SpringCloud 入门

> 此处使用的版本是
>
> springboot 1.5.10
>
> SpringCloud 版本 Edgware.SR2
>
> jdk 1.8.161
>
> 参考书籍 SpringCloud 与 Docker 微服务架构实战

根项目依赖

```
<!--标记为是 springboot 项目-->
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.10.RELEASE</version>
    </parent>
    <dependencyManagement>
        <dependencies>
            <!-- Feign 依赖-->
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Edgware.SR2</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <!--springboot 需要的额外依赖-->
            <dependency>
                <groupId>ch.qos.logback</groupId>
                <artifactId>logback-core</artifactId>
                <version>1.2.3</version>
            </dependency>
            <dependency>
                <groupId>ch.qos.logback</groupId>
                <artifactId>logback-classic</artifactId>
                <version>1.2.3</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>

        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-core</artifactId>
```

```
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
        </dependency>
    </dependencies>

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

一 微服务介绍

近年来，在软件开发领域关于微服务的讨论呈现出火爆的局面，有人倾向于在系统设计与开发中采用微服务方式实现软件系统的松耦合、跨部门开发，被认为是IT软件架构的未来方向，Martin Fowler也给微服务架构极高的评价；同时，反对之声也很强烈，持反对观点的人表示微服务增加了系统维护、部署的难度，导致一些功能模块或代码无法复用，同时微服务允许使用不同的语言和框架来开发各个系统模块，这又会增加系统集成与测试的难度，而且随着系统规模的日渐增长，微服务在一定程度上也会导致系统变得越来越复杂。尽管一些公司已经在生产系统中采用了微服务架构，并且取得了良好的效果；但更多公司还是处在观望的态度。

什么是微服务架构呢？简单说就是将一个完整的应用（单体应用）按照一定的拆分规则（后文讲述）拆分成多个不同的服务，每个服务都能独立地进行开发、部署、扩展。服务于服务之间通过注入RESTful api或其他方式调用。大家可以搜索到很多相关介绍和文章。本文暂不细表。在此推荐两个比较好的博客：

> http://microservices.io/ http://martinfowler.com/articles/microservices.html

微服务的优点:

1. 易于开发和维护,只需要关注每个服务的单独业务即可,不需要关心其他
2. 启动较快
3. 局部修改容易部署
4. 技术栈不受限
5. 按需伸缩
6. DevOps

微服务的缺点:

1. 运维成本较高
2. 分布式复杂
3. 接口调整成本高

4. 重复劳动,因为非集中项目,当使用不同技术栈的时候,一些功能可能需要多次开发

二 SpringCloud

2.1 SpringCloud 简介

Spring Cloud是在Spring Boot的基础上构建的，用于简化分布式系统构建的工具集，为开发人员提供快速建立分布式系统中的一些常见的模式。

> 例如：配置管理（configuration management），服务发现（service discovery），断路器（circuit breakers），智能路由（ intelligent routing），微代理（micro-proxy），控制总线（control bus），一次性令牌（ one-time tokens），全局锁（global locks），领导选举（leadership election），分布式会话（distributed sessions），集群状态（cluster state）。

Spring Cloud 包含了多个子项目：

> 例如：Spring Cloud Config、Spring Cloud Netflix等

Spring Cloud 项目主页：http://projects.spring.io/spring-cloud/

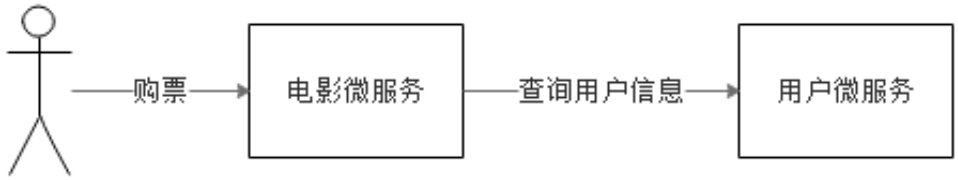
2.2 SpringCloud 的特点

1. 约定优于配置
2. 开箱即用、快速启动
3. 适用于各种环境
4. 轻量级的组件,比如 服务发现组件 Eureka
5. 组件的支持很丰富，功能很齐全 包含:配置中心,注册中心,智能路由
6. 选型中立 比如服务发现组件,不限制必须使用某一种,例如 Eureka,Zookeeper,Consul

三 服务器提供者与消费者

> 参考代码
>
> microservice-simple-provider-user
>
> microservice-simple-consumer-movie

名词	概念
服务提供者	服务的被调用方（即：为其他服务提供服务的服务）
服务消费者	服务的调用方（即：依赖其他服务的服务）



> 案例中, user 为提供者, movie 为消费者,此案例仅为模拟演示功能,没有实际意义
>
> user 中通过 jpa 访问数据库获取数据并返回,注意,不是有 Controller 就代表一定是一个带网页的项目,此处的 user 只提供接口访问而已
>
> movie 中通过使用 RestTemplate 访问 user 中 controller 提供的接口地址,获取数据,这样 user 就相当于为 movie 提供了一个微服务

3.1 user 提供者

3.1.1 yml 配置

```yaml
server:
  port: 7900
spring:
  jpa: #jpa 配置
    generate-ddl: false
    show-sql: true
    hibernate:
      ddl-auto: none
    database-platform: org.hibernate.dialect.MySQL5Dialect
  datasource: #数据库
    url: jdbc:mysql://localhost:3306/test?characterEncoding=UTF-8
    username: root
    password: qishimeiyoumima
    driver-class-name: com.mysql.jdbc.Driver
logging:
  level:
    root: INFO
    org.hibernate: INFO
    org.hibernate.type.descriptor.sql.BasicBinder: TRACE
    org.hibernate.type.descriptor.sql.BasicExtractor: TRACE
    com.qianfeng: DEBUG
```

3.1.2 User

```
@Entity
```

```java
public class User {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;

  @Column
  private String username;

  @Column
  private String name;

  @Column
  private Short age;

  @Column
  private BigDecimal balance;

  public Long getId() {
    return this.id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public String getUsername() {
    return this.username;
  }

  public void setUsername(String username) {
    this.username = username;
  }

  public String getName() {
    return this.name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public Short getAge() {
    return this.age;
  }

  public void setAge(Short age) {
    this.age = age;
  }
```

```
  public BigDecimal getBalance() {
    return this.balance;
  }

  public void setBalance(BigDecimal balance) {
    this.balance = balance;
  }
}
```

3.1.3 UserRepository

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
//使用 JPA 的默认功能,此处为演示项目,所以不需要其他功能
}
```

3.1.4 controller

```
@RestController
public class UserController {

  @Autowired
  private UserRepository userRepository;

  @GetMapping("/simple/{id}")//根据 id 查询数据库, rest 风格,微服务一般都使用 rest 风格
  public User findById(@PathVariable Long id) {
    return this.userRepository.findOne(id);
  }
}
```

3.1.5 user 主程序

```
@SpringBootApplication
public class MicroserviceSimpleProviderUserApplication {

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

3.1.6启动测试

> 启动后 访问 http://localhost:7900/simple/1 查看结果

3.2 movie 消费者

> 消费者在自己的接口内容通过调用 user 的接口获取数据

3.2.1 yml

```
  port: 7901
user: #此处 user没有任何含义,主要是一个名字而已
  #即便在这里也类似于硬编码, 还是不够灵活,定义提供者的位置
  userServicePath: http://localhost:7900/simple/
```

3.2.2 User

> 注意此处的 User 的权限定名称必须和上面的 user 一样,也就是必须在一样的包下面,虽然不在一个项目

```
public class User {
  private Long id;

  private String username;

  private String name;

  private Short age;

  private BigDecimal balance;

  public Long getId() {
    return this.id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public String getUsername() {
    return this.username;
  }

  public void setUsername(String username) {
    this.username = username;
  }

  public String getName() {
    return this.name;
```

```
  }

  public void setName(String name) {
    this.name = name;
  }

  public Short getAge() {
    return this.age;
  }

  public void setAge(Short age) {
    this.age = age;
  }

  public BigDecimal getBalance() {
    return this.balance;
  }

  public void setBalance(BigDecimal balance) {
    this.balance = balance;
  }

}
```

3.2.3 MovieController

```
@RestController
public class MovieController {
  @Autowired
  private RestTemplate restTemplate;// rest 请求模板类

  @Value("${user.userServicePath}")//从配置文件中读取指定属性,名字和配置中保持一致即可
  private String userServicePath;

  @GetMapping("/movie/{id}")
  public User findById(@PathVariable Long id) {
      //调用指定的地址,传递参数过去,将返回的数据解析为 user 格式
    // return this.restTemplate.getForObject("http://localhost:7900/simple/"+ id,
User.class);//硬编码,不好
    return this.restTemplate.getForObject(this.userServicePath + id,
User.class);//通过配置文件注入地址的方式
  }
}
```

3.2.4 movie 主程序

```
@SpringBootApplication
public class MicroserviceSimpleConsumerMovieApplication {

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

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

3.2.5 启动测试

> 启动 movie 之后,通过访问 http://localhost:7901/movie/1 发现可以获取到指定的 user,这样说明微服务调用成功

四 Eureka 服务发现

> 通过上面的例子,我们发现当我们使用硬编码的时候,只要其中一个位置变化,我们的所有的相关地方都需要变化,会导致出现严重问题,甚至引起雪崩效应,所以,我们需要对上面的例子进行改进,改进为自动发现的方式,用于解决上面的一些问题

4.1 服务发现与注册

在微服务架构中，服务发现（Service Discovery）是关键原则之一。手动配置每个客户端或某种形式的约定是很难做的，并且很脆弱。Spring Cloud提供了多种服务发现的实现方式，例如：Eureka、Consul、Zookeeper。

Spring Cloud支持得最好的是Eureka，其次是Consul，最次是Zookeeper。

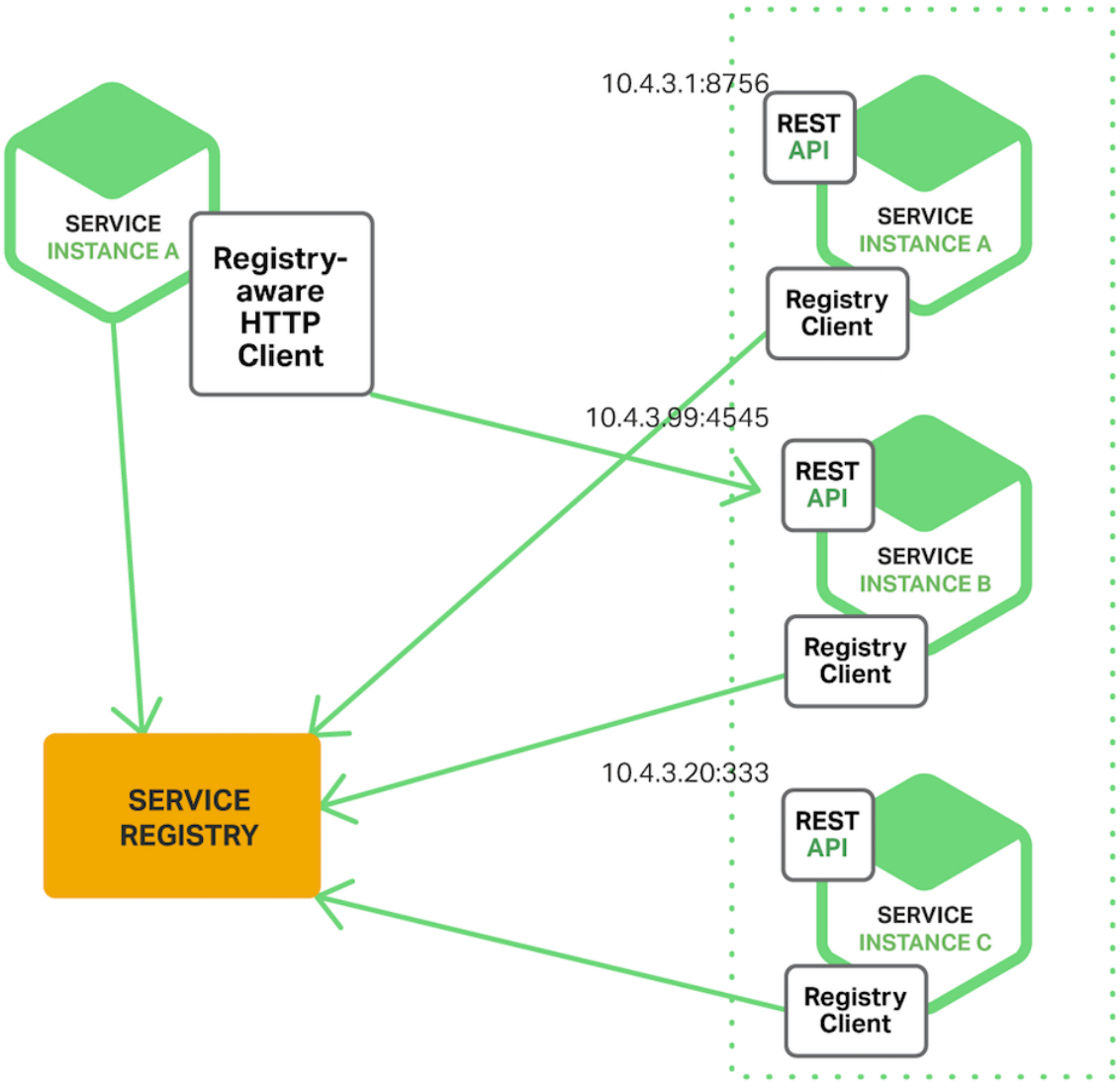
4.2 服务发现有两大模式

4.2.1客户端发现模式

使用客户端发现模式时，客户端决定相应服务实例的网络位置，并且对请求实现负载均衡。客户端查询服务注册表，后者是一个可用服务实例的数据库；然后使用负载均衡算法从中选择一个实例，并发出请求。

客户端从服务注册服务中查询，其中是所有可用服务实例的库。客户端使用负载均衡算法从多个服务实例中选择出一个，然后发出请求。

下图显示了这种模式的架构：



服务实例的网络位置在启动时被记录到服务注册表，等实例终止时被删除。服务实例的注册信息通常使用心跳机制来定期刷新。

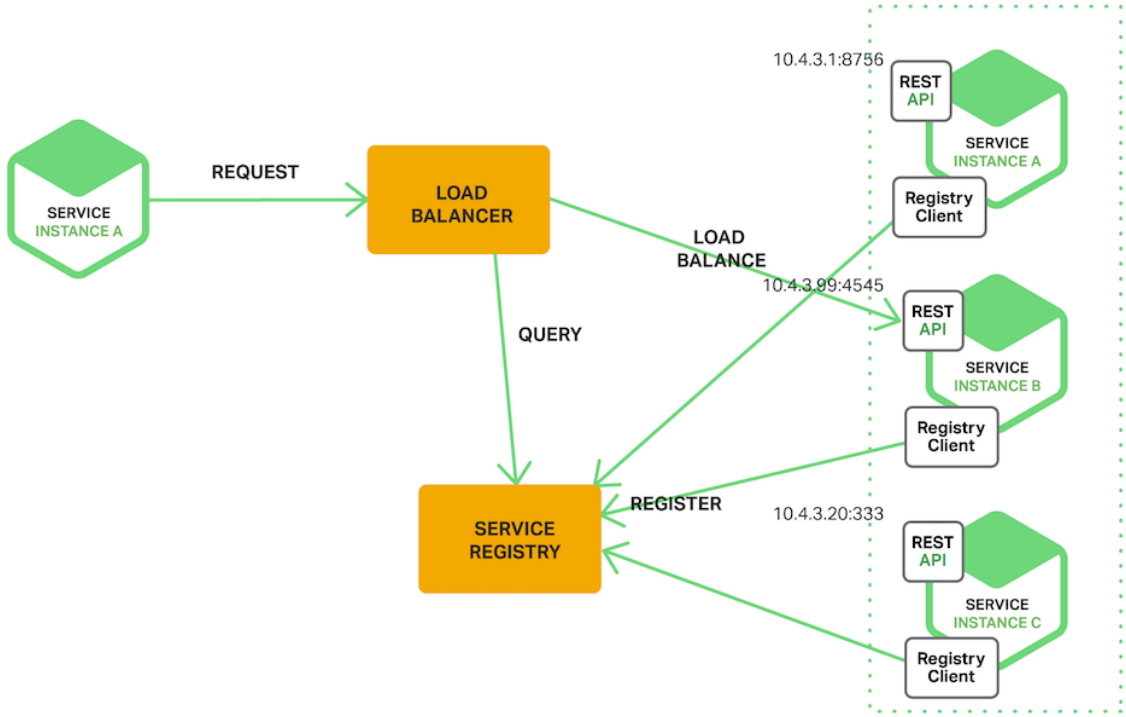
Netflix OSS 是客户端发现模式的绝佳范例。Netflix Eureka 是一个服务注册表，为服务实例注册管理和查询可用实例提供了 REST API 接口。Netflix Ribbon 是 IPC 客户端，与 Eureka 一起实现对请求的负载均衡。我们会在后面深入讨论 Eureka。

客户端发现模式优缺点兼有。这一模式相对直接，除了服务注册外，其它部分无需变动。此外，由于客户端知晓可用的服务实例，能针对特定应用实现智能负载均衡，比如使用哈希一致性。这种模式的一大缺点就是客户端与服务注册绑定，要针对服务端用到的每个编程语言和框架，实现客户端的服务发现逻辑。

分析过客户端发现后，我们来了解服务端发现。

4.2.2服务端发现模式

另外一种服务发现的模式是服务端发现模式，下图展现了这种模式的架构：



客户端通过负载均衡器向某个服务提出请求，负载均衡器查询服务注册表，并将请求转发到可用的服务实例。如同客户端发现，服务实例在服务注册表中注册或注销。

AWS Elastic Load Balancer（ELB）是服务端发现路由的例子，ELB 通常均衡来自互联网的外部流量，也可用来负载均衡 VPC（Virtual private cloud）的内部流量。客户端使用 DNS 通过 ELB 发出请求（HTTP 或 TCP），ELB 在已注册的 EC2 实例或 ECS 容器之间负载均衡。这里并没有单独的服务注册表，相反，EC2 实例和 ECS 容器注册在 ELB。

HTTP 服务器与类似 NGINX PLUS 和 NGINX 这样的负载均衡起也能用作服务端的发现均衡器。Graham Jenson 的 Scalable Architecture DR CoN: Docker, Registrator, Consul, Consul Template and Nginx 一文就描述如何使用 Consul Template 来动态配置 NGINX 反向代理。Consul Template 定期从 Consul Template 注册表中的配置数据中生成配置文件；文件发生更改即运行任意命令。在这篇文章中，Consul Template 生成 nginx.conf 文件，用于配置反向代理，然后运行命令，告诉 NGINX 重新加载配置文件。在更复杂的实现中，需要使用 HTTP API 或 DNS 来动态配置 NGINX Plus。

Kubernetes 和 Marathon 这样的部署环境会在每个集群上运行一个代理，将代理用作服务端发现的负载均衡器。客户端使用主机 IP 地址和分配的端口通过代理将请求路由出去，向服务发送请求。代理将请求透明地转发到集群中可用的服务实例。

服务端发现模式兼具优缺点。它最大的优点是客户端无需关注发现的细节，只需要简单地向负载均衡器发送请求，这减少了编程语言框架需要完成的发现逻辑。并且如上文所述，某些部署环境免费提供这一功能。这种模式也有缺点。除非负载均衡器由部署环境提供，否则会成为一个需要配置和管理的高可用系统组件。

4.2.3服务注册表

服务注册表是服务发现的核心部分，是包含服务实例的网络地址的数据库。服务注册表需要高可用而且随时更新。客户端能够缓存从服务注册表中获取的网络地址，然而，这些信息最终会过时，客户端也就无法发现服务实例。因此，服务注册表会包含若干服务端，使用复制协议保持一致性。

如前所述，Netflix Eureka 是服务注册表的上好案例，为注册和请求服务实例提供了 REST API。服务实例使用 POST 请求来注册网络地址，每三十秒使用 PUT 请求来刷新注册信息。注册信息也能通过 HTTP DELETE 请求或者实例超时来被移除。以此类推，客户端能够使用 HTTP GET 请求来检索已注册的服务实例。

Netflix 通过在每个 AWS EC2 域运行一个或者多个 Eureka 服务实现高可用性。每个 Eureka 服务器都运行在拥有弹性 IP 地址的 EC2 实例上。DNS TEXT 记录被用来保存 Eureka 集群配置，后者包括可用域和 Eureka 服务器的网络地址列表。Eureka 服务在启动时会查询 DNS 去获取 Eureka 集群配置，确定同伴位置，以及给自己分配一个未被使用的弹性 IP 地址。

Eureka 客户端，包括服务和服务客户端，查询 DNS 去发现 Eureka 服务的网络地址。客户端首选同一域内的 Eureka 服务。然而，如果没有可用服务，客户端会使用其它可用域中的 Eureka 服务。

其它的服务注册表包括：

- etcd – 高可用、分布式、一致性的键值存储，用于共享配置和服务发现。Kubernetes 和 Cloud Foundry 是两个使用 etcd 的著名项目。
- consul – 发现和配置的服务，提供 API 实现客户端注册和发现服务。Consul 通过健康检查来判断服务的可用性。
- Apache ZooKeeper – 被分布式应用广泛使用的高性能协调服务。Apache ZooKeeper 最初是 Hadoop 的子项目，现在已成为顶级项目。

此外，如前所强调，像 Kubernetes、Marathon 和 AWS 并没有明确的服务注册，相反，服务注册已经内置在基础设施中。

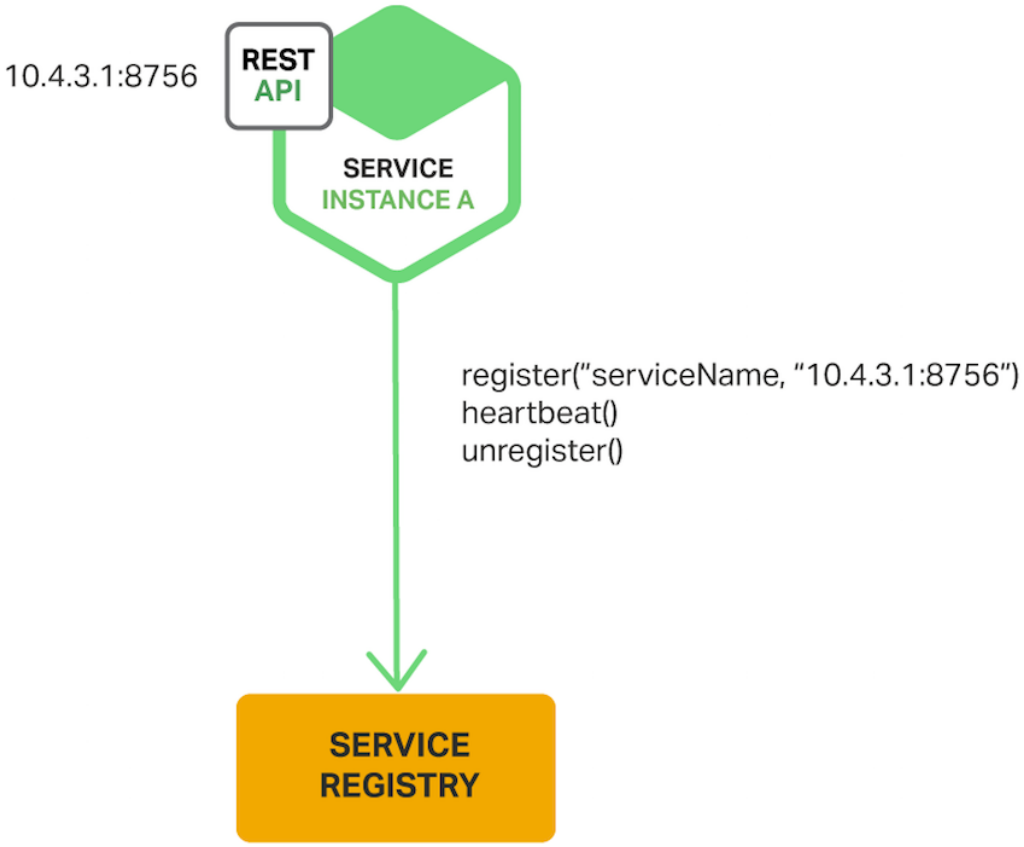
了解了服务注册的概念后，现在了解服务实例如何在注册表中注册。

4.2.4服务注册的方式

如前所述，服务实例必须在注册表中注册和注销。注册和注销有两种不同的方法。方法一是服务实例自己注册，也叫自注册模式（self-registration pattern）；另一种是采用管理服务实例注册的其它系统组件，即第三方注册模式。

4.2.4.1自注册方式

当使用自注册模式时，服务实例负责在服务注册表中注册和注销。另外，如果需要的话，一个服务实例也要发送心跳来保证注册信息不会过时。下图描述了这种架构：



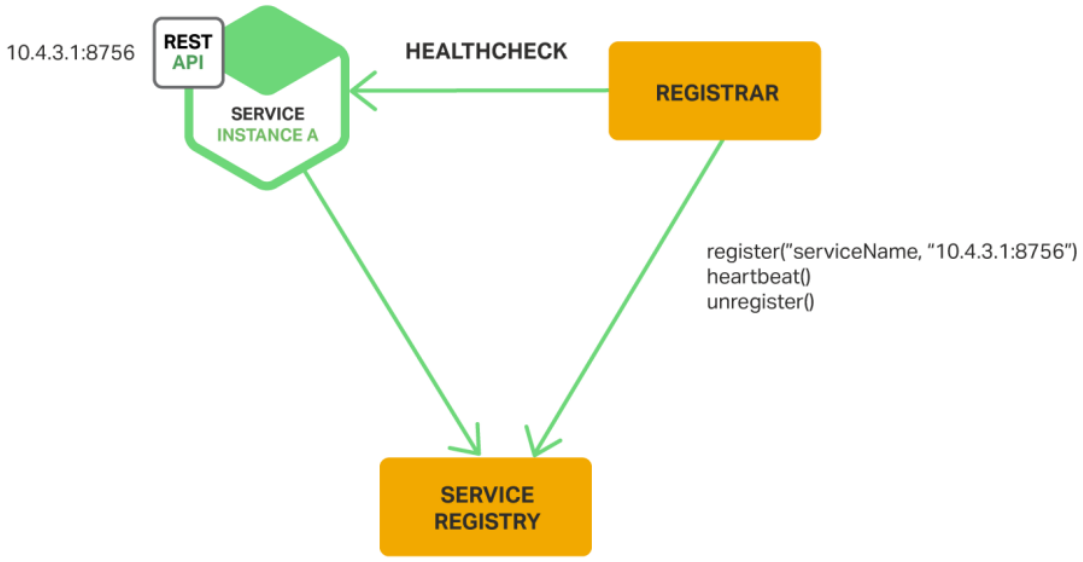
Netflix OSS Eureka 客户端是非常好的案例，它负责处理服务实例的注册和注销。Spring Cloud 能够执行包括服务发现在内的各种模式，使得利用 Eureka 自动注册服务实例更简单，只需要给 Java 配置类注释 @EnableEurekaClient。

自注册模式优缺点兼备。它相对简单，无需其它系统组件。然而，它的主要缺点是把服务实例和服务注册表耦合，必须在每个编程语言和框架内实现注册代码。

另一个方案将服务与服务注册表解耦合，被称作第三方注册模式。

4.2.4.2第三方注册模式

使用第三方注册模式，服务实例则不需要向服务注册表注册；相反，被称为服务注册器的另一个系统模块会处理。服务注册器会通过查询部署环境或订阅事件的方式来跟踪运行实例的更改。一旦侦测到有新的可用服务实例，会向注册表注册此服务。服务管理器也负责注销终止的服务实例。下面是这种模式的架构图。



Registrator 是一个开源的服务注册项目，它能够自动注册和注销被部署为 Docker 容器的服务实例。Registrator 支持包括 etcd 和 Consul 在内的多种服务注册表。

NetflixOSS Prana 是另一个服务注册器，主要面向非 JVM 语言开发的服务，是一款与服务实例一起运行的并行应用。Prana 使用 Netflix Eureka 来注册和注销服务实例。

服务注册器是部署环境的内置组件。由 Autoscaling Group 创建的 EC2 实例能够自动向 ELB 注册。Kubernetes 服务自动注册并能够被发现。

第三方注册模式也是优缺点兼具。在第三方注册模式中，服务与服务注册表解耦合，无需为每个编程语言和框架实现服务注册逻辑；相反，服务实例通过一个专有服务以中心化的方式进行管理。它的不足之处在于，除非该服务内置于部署环境，否则需要配置和管理一个高可用的系统组件。

4.2.5总结

在微服务应用中，服务实例的运行环境会动态变化，实例网络地址也是如此。因此，客户端为了访问服务必须使用服务发现机制。

服务注册表是服务发现的关键部分。服务注册表是可用服务实例的数据库，提供管理 API 和查询 API。服务实例使用管理 API 来实现注册和注销，系统组件使用查询 API 来发现可用的服务实例。

服务发现有两种主要模式：客户端发现和服务端发现。在使用客户端服务发现的系统中，客户端查询服务注册表，选择可用的服务实例，然后发出请求。在使用服务端发现的系统中，客户端通过路由转发请求，路由器查询服务注册表并转发请求到可用的实例。

服务实例的注册和注销也有两种方式。一种是服务实例自己注册到服务注册表中，即自注册模式；另一种则是由其它系统组件处理注册和注销，也就是第三方注册模式。

在一些部署环境中，需要使用 Netflix Eureka、etcd、Apache Zookeeper 等服务发现来设置自己的服务发现基础设施。而另一些部署环境则内置了服务发现。例如，Kubernetes 和 Marathon 处理服务实例的注册和注销，它们也在每个集群主机上运行代理，这个代理具有服务端发现路由的功能。

HTTP 反向代理和 NGINX 这样的负载均衡器能够用做服务器端的服务发现均衡器。服务注册表能够将路由信息推送到 NGINX，激活配置更新，譬如使用 Cosul Template。NGINX Plus 支持额外的动态配置机制，能够通过 DNS 从注册表中获取服务实例的信息，并为远程配置提供 API

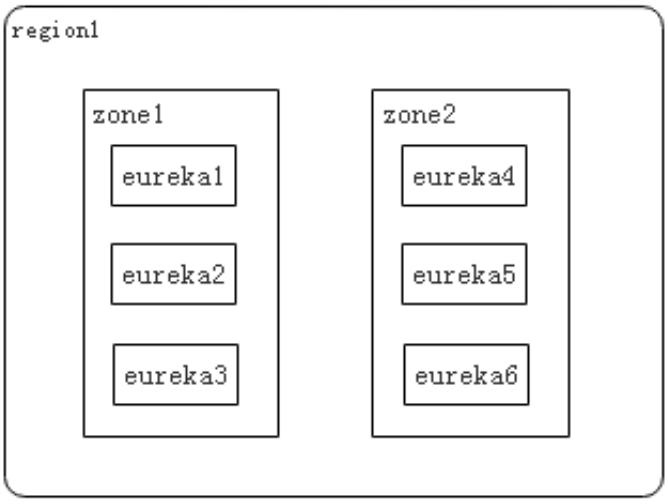
4.3 Eureka 介绍与 Eureka Server

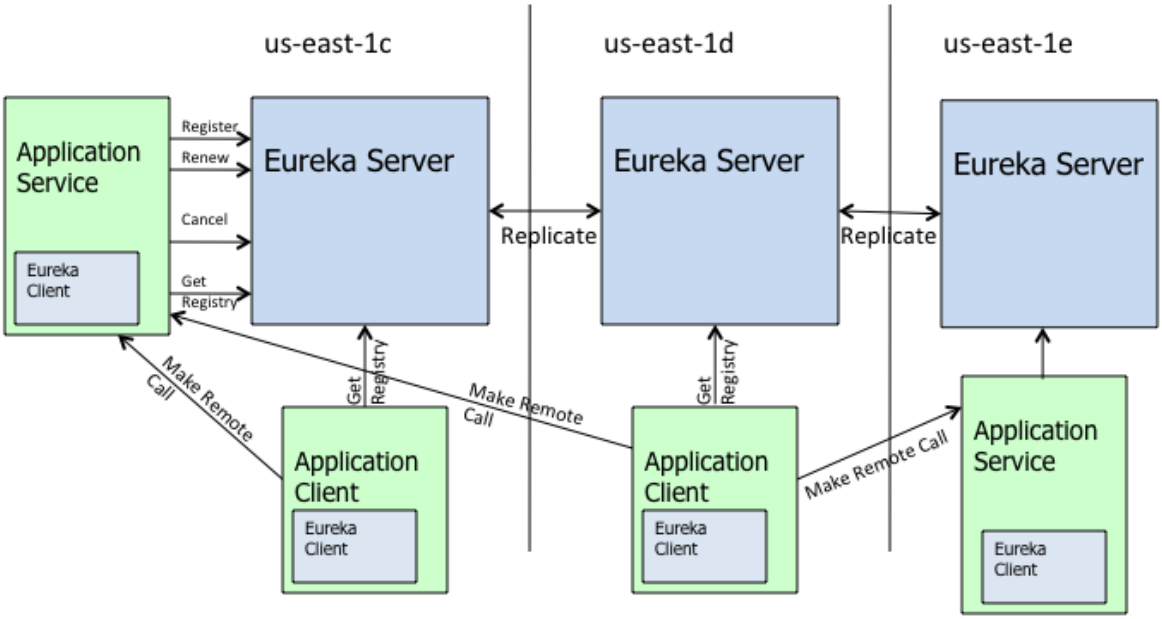
4.3.1 Eureka 介绍

Eureka是Netflix开发的服务发现框架，本身是一个基于REST的服务，主要用于定位运行在AWS域中的中间层服务，以达到负载均衡和中间层服务故障转移的目的。Spring Cloud将它集成在其子项目spring-cloud-netflix中，以实现Spring Cloud的服务发现功能。

4.3.2 Eureka 原理

4.3.2.1Region和Zone的关系





上图是来自Eureka官方的架构图，大致描述了Eureka集群的工作过程。由于图比较复杂，可能比较难看懂，这边用通俗易懂的语言翻译一下：

- Application Service 就相当于本书中的服务提供者（用户微服务），Application Client就相当于本示例中的服务消费者（电影微服务）；

- Make Remote Call，可以简单理解为调用RESTful的接口；

- us-east-1c、us-east-1d等是zone，它们都属于us-east-1这个region；

由图可知，Eureka包含两个组件：Eureka Server 和 Eureka Client。

Eureka Server提供服务注册服务，各个节点启动后，会在Eureka Server中进行注册，这样Eureka Server中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到。

Eureka Client是一个Java客户端，用于简化与Eureka Server的交互，客户端同时也具备一个内置的、使用轮询（round-robin）负载算法的负载均衡器。

在应用启动后，将会向Eureka Server发送心跳（默认周期为30秒）。如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，Eureka Server将会从服务注册表中把这个服务节点移除（默认90秒）。

Eureka Server之间将会通过复制的方式完成数据的同步.

Eureka还提供了客户端缓存的机制，即使所有的Eureka Server都挂掉，客户端依然可以利用缓存中的信息消费其他服务的API.

综上，Eureka通过心跳检测、健康检查、客户端缓存等机制，确保了系统的高可用性、灵活性和可伸缩性。

4.3.3源码

4.3.3.1 源码获取、构建

https://github.com/Netflix/eureka 下载到原生 Eureka 代码，在 https://github.com/spring-cloud/spring-cloud-netflix 下载Spring Cloud针对于Eureka的Spring Cloud适配。

4.3.3.2 程序构成

Eureka:

1. 是纯正的 servlet 应用，需构建成war包部署
2. 使用了 Jersey 框架实现自身的 RESTful HTTP接口
3. peer之间的同步与服务的注册全部通过 HTTP 协议实现
4. 定时任务(发送心跳、定时清理过期服务、节点同步等)通过 JDK 自带的 `Timer` 实现
5. 内存缓存使用Google的guava包实现

4.3.3.3 代码结构

模块概览：

eureka-core 模块包含了功能的核心实现:

1. com.netflix.eureka.cluster - 与peer节点复制(replication)相关的功能
2. com.netflix.eureka.lease - 即"租约", 用来控制注册信息的生命周期(添加、清除、续约)
3. com.netflix.eureka.registry - 存储、查询服务注册信息
4. com.netflix.eureka.resources - RESTful风格中的"R", 即资源。相当于SpringMVC中的Controller
5. com.netflix.eureka.transport - 发送HTTP请求的客户端，如发送心跳
6. com.netflix.eureka.aws - 与amazon AWS服务相关的类

eureka-client模块: Eureka客户端，微服务通过该客户端与Eureka进行通讯，屏蔽了通讯细节

eureka-server模块: 包含了 servlet 应用的基本配置，如 web.xml。构建成功后在该模块下会生成可部署的war包。

4.3.4代码入口

4.3.4.1 作为纯Servlet应用的入口

由于是Servlet应用，所以Eureka需要通过servlet的相关监听器 `ServletContextListener` 嵌入到Servlet 的生命周期中。`EurekaBootStrap` 类实现了该接口，在servlet标准的 `contextInitialized()` 方法中完成了初始化工作：

```
@Override
    public void contextInitialized(ServletContextEvent event) {
        try {
            // 读取配置信息
            initEurekaEnvironment();
            // 初始化Eureka Client(用来与其它节点进行同步)
```

```
            // 初始化server
            initEurekaServerContext();
            ServletContext sc = event.getServletContext();
            sc.setAttribute(EurekaServerContext.class.getName(), serverContext);
        } catch (Throwable e) {
            logger.error("Cannot bootstrap eureka server :", e);
            throw new RuntimeException("Cannot bootstrap eureka server :", e);
        }
    }10
```

4.3.4.2 与Spring Cloud结合的胶水代码

Eureka是一个纯正的Servlet应用，而Spring Boot使用的是嵌入式Tomcat, 因此就需要一定的胶水代码让Eureka跑在Embedded Tomcat中。这部分工作是在 `EurekaServerBootstrap` 中完成的。与上面提到的 `EurekaBootStrap` 相比，它的代码几乎是直接将原生代码copy过来的，虽然它并没有继承 `ServletContextListener`, 但是相应的生命周期方法都还在，然后添加了 `@Configuration` 注解使之能被Spring容器感知：

原生的 `EurekaBootStrap` 类实现了标准的 `ServletContextListener` 接口

Spring Cloud的 `EurekaServerBootstrap` 类没有实现servlet接口，但是**保留了接口方法的完整实现**

我们可以推测，框架一定是在某处调用了这些方法，然后才是执行原生Eureka的启动逻辑。 `EurekaServerInitializerConfiguration` 类证实了我们的推测。该类实现了 `ServletContextAware` (拿到了tomcat的ServletContext对象)、 `SmartLifecycle` (Spring容器初始化该bean时会调用相应生命周期方法)：

```
@Configuration
@CommonsLog
public class EurekaServerInitializerConfiguration
        implements ServletContextAware, SmartLifecycle, Ordered {
}
```

在 start() 方法中可以看到

eurekaServerBootstrap.contextInitialized(EurekaServerInitializerConfiguration.this.servletContext);的调用，也就是说，在Spring容器初始化该组件时，Spring调用其生命周期方法start()从而触发了Eureka的启动。

```
    @Override
    public void start() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
```

```
 eurekaServerBootstrap.contextInitialized(EurekaServerInitializerConfiguration.thi
s.servletContext); // 调用 servlet 接口方法手工触发启动
                    log.info("Started Eureka Server");
                    // ... ...
                }
                catch (Exception ex) {
                    // Help!
                    log.error("Could not initialize Eureka servlet context", ex);
                }
            }
        }).start();
    }
```

4.3.4.3 其它几个重要的代码入口

了解以上入口信息后，我们就可以根据自己的需要自行研读相关的代码了。这里再提示几个代码入口：

1. `com.netflix.appinfo.InstanceInfo`类封装了服务注册所需的全部信息 \2. Eureka Client探测本机IP是通过`org.springframework.cloud.commons.util.InetUtils`工具类实现的
2. `com.netflix.eureka.resources.ApplicationResource`类相当于Spring MVC中的控制器，是服务的注册、查询功能的代码入口点

4.3.5可能会被坑的几处原理

4.3.5.1 Eureka的几处缓存

Eureka的wiki上有一句话，大意是一个服务启动后最长可能需要2分钟时间才能被其它服务感知到，但是文档并没有解释为什么会有这2分钟。其实这是由三处缓存 + 一处延迟造成的。

首先，Eureka对HTTP响应做了缓存。在Eureka的"控制器"类`ApplicationResource`的109行可以看到有一行

```
String payLoad = responseCache.get(cacheKey);11
```

的调用，该代码所在的`getApplication()`方法的功能是响应客户端查询某个服务信息的HTTP请求：

```
String payLoad = responseCache.get(cacheKey); // 从cache中拿响应数据
if (payLoad != null) {
        logger.debug("Found: {}", appName);
        return Response.ok(payLoad).build();
} else {
        logger.debug("Not Found: {}", appName);
        return Response.status(Status.NOT_FOUND).build();
}
```

上面的代码中，responseCache引用的是ResponseCache类型，该类型是一个接口，其get()方法首先会去缓存中查询数据，如果没有则生成数据返回（即真正去查询注册列表），且缓存的有效时间为30s。也就是说，客户端拿到Eureka的响应并不一定是即时的，大部分时候只是缓存信息。

其次，Eureka Client对已经获取到的注册信息也做了30s缓存。即服务通过eureka客户端第一次查询到可用服务地址后会将结果缓存，下次再调用时就不会真正向Eureka发起HTTP请求了。

再次， 负载均衡组件Ribbon也有30s缓存。Ribbon会从上面提到的Eureka Client获取服务列表，然后将结果缓存30s。

最后，如果你并不是在Spring Cloud环境下使用这些组件(Eureka, Ribbon)，你的服务启动后并不会马上向Eureka注册，而是需要等到第一次发送心跳请求时才会注册。心跳请求的发送间隔也是30s。（Spring Cloud对此做了修改，服务启动后会马上注册）

以上这四个30秒正是官方wiki上写服务注册最长需要2分钟的原因。

4.3.5.2服务注册信息不会被二次传播

如果Eureka A的peer指向了B, B的peer指向了C，那么当服务向A注册时，B中会有该服务的注册信息，但是C中没有。也就是说，如果你希望只要向一台Eureka注册其它所有实例都能得到注册信息，那么就必须把其它所有节点都配置到当前Eureka的`peer`属性中。这一逻辑是在`PeerAwareInstanceRegistryImpl#replicateToPeers()`方法中实现的：

```
private void replicateToPeers(Action action, String appName, String id,
                                  InstanceInfo info /* optional */,
                                  InstanceStatus newStatus /* optional */, boolean
isReplication) {
        Stopwatch tracer = action.getTimer().start();
        try {
            if (isReplication) {
                numberOfReplicationsLastMin.increment();
            }
            // 如果这条注册信息是其它Eureka同步过的则不会再继续传播给自己的peer节点
            if (peerEurekaNodes == Collections.EMPTY_LIST || isReplication) {
                return;
            }
            for (final PeerEurekaNode node : peerEurekaNodes.getPeerEurekaNodes())
{
                // 不要向自己发同步请求
                if (peerEurekaNodes.isThisMyUrl(node.getServiceUrl())) {
                    continue;
                }
                replicateInstanceActionsToPeers(action, appName, id, info,
newStatus, node);
            }
        } finally {
            tracer.stop();
        }
    }
```

4.3.5.3多网卡环境下的IP选择问题

如果服务部署的机器上安装了多块网卡，它们分别对应IP地址A, B, C，此时： Eureka会**选择IP合法**(标准ipv4地址)、**索引值最小**(eth0, eth1中eth0优先)且**不在忽略列表中**(可在`application.properites`中配置忽略哪些网卡)的网卡地址作为服务IP。 这个坑的详细分析见：http://blog.csdn.net/neosmith/article/details/53126924

4.3.6作为服务注册中心，Eureka比Zookeeper好在哪里

著名的CAP理论指出，一个分布式系统不可能同时满足C(一致性)、A(可用性)和P(分区容错性)。由于分区容错性在是分布式系统中必须要保证的，因此我们只能在A和C之间进行权衡。在此Zookeeper保证的是CP, 而Eureka则是AP。

4.3.6.1 Zookeeper保证CP

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

4.3.6.2 Eureka保证AP

Eureka看明白了这一点，因此在设计时就优先保证可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1. Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务
2. Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)
3. 当网络稳定时，当前实例新的注册信息会被同步到其它节点中

因此， Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪。

4.3.7 总结

Eureka作为单纯的服务注册中心来说要比zookeeper更加“专业”，因为注册服务更重要的是可用性，我们可以接受短期内达不到一致性的状况。不过Eureka目前1.X版本的实现是基于servlet的Java web应用，它的极限性能肯定会受到影响。期待正在开发之中的2.X版本能够从servlet中独立出来成为单独可部署执行的服务。

4.4 注册微服务到Eureka Server

> 参考示例中的

> microservice-provider-user
>
> microservice-discovery-eureka
>
> microservice-consumer-movie

4.4.1POM

> Eureka pom

```
<dependencies>
    <!--注意此处是 eureka server不是 eureka -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka-server</artifactId>
        </dependency>
        <!--如果访问 eureka 需要密码,则添加此依赖-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
    </dependencies>
```

> 提供者 user pom

```
<dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    <!--注意此处是 eureka 不是 eureka server-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
    <!--如果访问 eureka 需要密码,则添加此依赖-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
```

```
    </dependencies>
```

> movie 消费者 pom

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
  <!--注意此处是 eureka 不是 eureka server-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
    <!--如果访问 eureka 需要密码,则添加此依赖-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
    </dependencies>
```

4.4.2 创建 Eureka Server 项目

4.4.2.1 application.yml

> 用于指定 eureka 的相关配置

```
#配置需要密码,如果不需要,将下面地址中的相关参数取消即可,配置密码需要相关的依赖
security:
  basic:
    enabled: true
  user:
    name: user
    password: password123
server: ##eureka后台端口地址
  port: 8761
eureka:
  client:
  #因为是单机版,所以设置当前 server 只是一个 server, 而不是其他 eureka 的客户端
    register-with-eureka: false
    fetch-registry: false
    service-url:
```

```
      ##地址中的 user password 代表需要密码,为 curl 风格,会从上面配置的 user 和
password 中获取数据 用于登录,配置用户名和密码后,客户端必须也使用这样的地址
      defaultZone: http://user:password123@localhost:8761/eureka
```

4.4.2.2 主程序

```
@SpringBootApplication
@EnableEurekaServer//设置为 Eureka 服务端,开启此注解后,启动项目,会自动加载配置文件,配
置 server, 然后即可通过上面配置的端口号访问
public class EurekaApplication {
  public static void main(String[] args) {
    SpringApplication.run(EurekaApplication.class, args);
  }
}
```

4.4.3 修改原始的提供者项目 user

4.4.3.1 application.yml

> 在原始的 yml 文件中添加以下内容即可

```
eureka:
  client:
    healthcheck:
      enabled: true  #这个配置只能放在application.yml 中,不能放在 bootstrap.yml, 参
考http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-
cloud.html#_eureka_s_health_checks
    serviceUrl:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true   #在 eureka 后台中访问当前项目的时候使用ip 地址访问,默认
是机器名的方式
    ##在 eureka 后台显示的项目相关信息的格式
    instance-id:
${spring.application.name}:${spring.cloud.client.ipAddress}:${spring.application.i
nstance_id:${server.port}}
   #元数据
    metadata-map: #可以在Eureka 地址/apps/{appname}中查看该数据
      zone: ABC       # eureka可以理解的元数据
      jackiechan: sdfsdf  # 不会影响客户端行为
      ##心跳时间为5秒,加快刷新速度,但是不建议修改,防止打乱默认的一些设置
    lease-renewal-interval-in-seconds: 5
```

4.4.3.2 主程序

> 主程序只需要添加一个@EnableEurekaClient注解即可

```
@SpringBootApplication
@EnableEurekaClient//标记为是 Eureka 客户端,会自动去配置中的服务器地址注册
public class MicroserviceSimpleConsumerMovieApplication {

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

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

4.4.4 UserController 中的其他方法

> 为了演示一些功能的,非必须的

```
@RestController
public class UserController {

  @Autowired
  private UserRepository userRepository;

  @Autowired
  private EurekaClient eurekaClient;

  @Autowired
  private DiscoveryClient discoveryClient;

  @GetMapping("/simple/{id}")
  public User findById(@PathVariable Long id) {
    return this.userRepository.findOne(id);
  }

  /**
   * 访问当前地址可以获取当前服务器的信息
   * @return
   */
  @GetMapping("/eureka-instance")
  public String serviceUrl() {
    InstanceInfo instance =
this.eurekaClient.getNextServerFromEureka("MICROSERVICE-PROVIDER-USER", false);
    return instance.getHomePageUrl();
  }

  /**
```

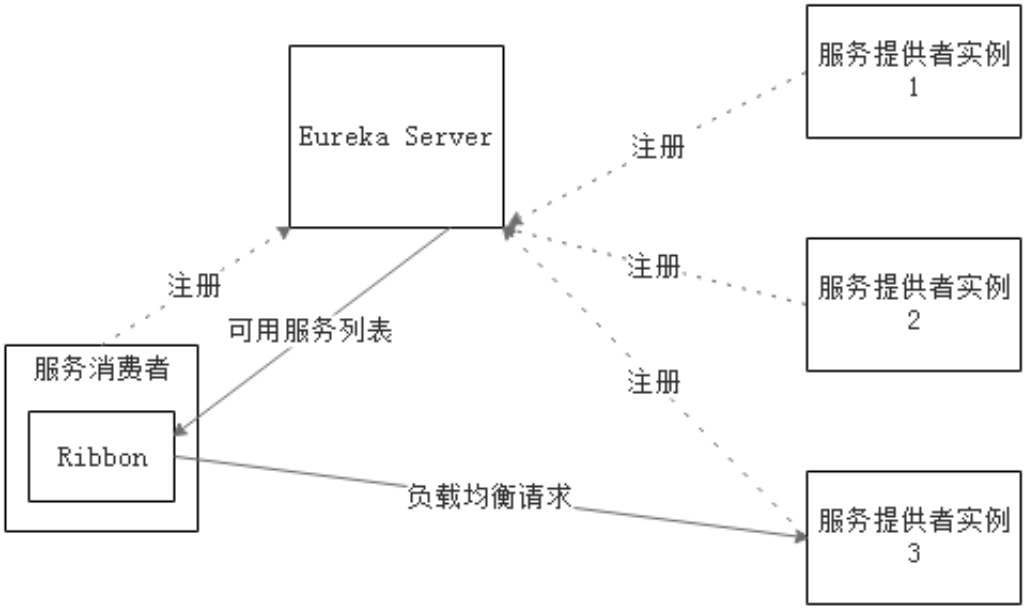
```
     * 获取相信信息
     * @return
     */
    @GetMapping("/instance-info")
    public ServiceInstance showInfo() {
      ServiceInstance localServiceInstance =
this.discoveryClient.getLocalServiceInstance();
      return localServiceInstance;
    }

    /**
     * 获取对象方式的参数
     * @param user
     * @return
     */
    @PostMapping("/user")
    public User postUser(@RequestBody User user) {
      return user;
    }

    // 该请求不会成功,因为必须是 post 方式
    @GetMapping("/get-user")
    public User getUser(User user) {
      return user;
    }

    /**
     * 获取多条数据
     * @return
     */
    @GetMapping("list-all")
    public List<User> listAll() {
      ArrayList<User> list = Lists.newArrayList();
      User user = new User(1L, "zhangsan");
      User user2 = new User(2L, "zhangsan");
      User user3 = new User(3L, "zhangsan");
      list.add(user);
      list.add(user2);
      list.add(user3);
      return list;

    }
```

> 通过访问 http://localhost:7900 加相关地址即可测试上述接口

4.4.5 修改消费者 movie

> 将 movie 的 yml文件添加上述配置 以及主类添加注解即可

五 Ribbbon

> 使用 Eureka 后我们可以通过服务器找到我们注册的客户端,但是如果我们的微服务有好几个,比如我们的 movie 使用了 user 服务,如果 user 服务有好多个,如何实现负载均衡,而且上述的配置文件中还是有一些硬编码的存在,如何解决,我们就需要用到 ribbon
>
> Feign 中自带 ribbon, 所以不需要导入依赖
>
> Ribbon是Netflix发布的云中间层服务开源项目，其主要功能是提供客户端侧负载均衡算法。Ribbon客户端组件提供一系列完善的配置项如连接超时，重试等。简单的说，Ribbon是一个客户端负载均衡器，我们可以在配置文件中列出Load Balancer后面所有的机器，Ribbon会自动的帮助你基于某种规则（如简单轮询，随机连接等）去连接这些机器，我们也很容易使用Ribbon实现自定义的负载均衡算法。
>
> 下图展示了Eureka使用Ribbon时候的大致架构：
>
> 
>
>
> Ribbon工作时分为两步：第一步先选择 Eureka Server, 它优先选择在同一个Zone且负载较少的Server；第二步再根据用户指定的策略，在从Server取到的服务注册列表中选择一个地址。其中Ribbon提供了多种策略，例如轮询round robin、随机Random、根据响应时间加权等。

5.1 Ribbon 负载均衡

> 参考示例代码
>
> microservice-provider-user
>
> microservice-discovery-eureka
>
> microservice-consumer-movie-ribbon
>
> 我们启动多个 user 服务,注册到 eureka 中,然后movie 去调用 user, 通过在 movie 中配置负载均衡来保证可以访问到多个 user 服务

5.1.1 配置 movie

5.1.1.1 配置 yml 文件

```
spring:
  application:
    name: microservice-consumer-movie-ribbon   #修改自己的名字
server:
  port: 8010
eureka:
  client:
    healthcheck:
      enabled: true
    serviceUrl:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
```

5.1.1.2 修改 RestTemplate 的获取方式

> 将主程序中的RestTemplate添加 @LoadBalanced注解即可实现负载均衡

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

5.1.1.3 修改 controller 中的方法

```
@GetMapping("/movie/{id}")
  public User findById(@PathVariable Long id) {
    // http://localhost:7900/simple/
    // VIP virtual IP
    // HAProxy Heartbeat
      //microservice-provider-user代表 user 微服务在 eureka 中的名字,也就是 user 的
yml 配置文件中的 applicationname, 这样子它会从 eureka 中找这个名字的服务
    return this.restTemplate.getForObject("http://microservice-provider-
user/simple/" + id, User.class);
  }
```

5.1.1.4 测试

> 启动 eureka ,然后通过修改 user 的 yml 中的端口实现启动多个 user, 然后启动 movie,通过访问 movie 中的接口,发现可以实现轮询方式的负载均衡
>
> 如果一开始已经有 user 服务器启动,后续新开的 user,开始的时候可能没有负载均衡效果,因为如果是中途新添加的 user 服务器,需要时间去刷新到客户端

5.2 自定义配置Ribbon

> 参考 Feign 的 ribbon 的文档,注意配置时候的配置文件的位置,详情参考 waring
>
> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_customizing_the_ribbon_client

5.2.1 Config 文件符合上述要求位置的方式

5.2.1.1 在非 @ComponentScan 或者@SpringBootApplication能扫描到的位置下创建该类

```
@Configuration
//@RibbonClient(name = "microservice-provider-user", configuration =
TestConfiguration.class) //挪到主程序类上面去了,通过点击configuration查看 默认类型是
RibbonClientConfiguration ,内部有一个ribbonRule方法是用于创建规则的,我们也编写这个方法
public class TestConfiguration {
   @Autowired
  IClientConfig config;
/**
* 用于创建负载均衡规则对象的方法
**/
  @Bean
  public IRule ribbonRule(IClientConfig config) {
    return new RandomRule();//创建一个随机的规则,用于负载均衡时的随机算法
  }
}
```

5.2.1.2 修改主程序

```
@SpringBootApplication
@EnableEurekaClient
//name 代表要给哪个微服务指定配置  configuration代表配置所在的类
@RibbonClient(name = "microservice-provider-user", configuration =
TestConfiguration.class)//为microservice-provider-user"这个微服务 指定配置文件为
TestConfiguration这个类
public class ConsumerMovieRibbonApplication {

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

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

5.2.1.3 测试

> 启动 Eureka server, user 提供者(通过修改配置文件端口实现多次启动), 启动 movie, 访问 movie 的方法,测试,发现为随机访问了

5.2.2 不同微服务使用不同负载均衡模式

> 如果有多个不同的微服务,想指定不同的模式,如何配置呢
>
> 比如,我们通过修改 user 服务的端口和 applicationname 来模拟启动多个不同的服务
>
> microservice-provider-user 使用7900 7901 端口
>
> microservice-provider-user2 使用 7902 7903端口

5.2.2.1 修改 Movie 的 controller

> 在 controller 中 添加相关变量和方法

```
 @Autowired
  private LoadBalancerClient loadBalancerClient; //添加一个负载均衡的 client


    //添加一个新的访问地址,通过负载均衡去获取指定服务,然后打印相关参数
  @GetMapping("/test")
  public String test() {
    ServiceInstance serviceInstance =
this.loadBalancerClient.choose("microservice-provider-user");//获取microservice-
provider-user的一个实例
    System.out.println("111" + ":" + serviceInstance.getServiceId() + ":" +
serviceInstance.getHost() + ":" + serviceInstance.getPort());

    ServiceInstance serviceInstance2 =
this.loadBalancerClient.choose("microservice-provider-user2");//获取一个
microservice-provider-user2的实例
    System.out.println("222" + ":" + serviceInstance2.getServiceId() + ":" +
serviceInstance2.getHost() + ":" + serviceInstance2.getPort());

    return "1";
  }
```

5.2.3 Config 文件在特定位置

> 上面的配置中 我们的规则文件不在 SCAN 或者是 SpringBootApplication 的扫描目录中,那如果 Config 文件必须得在这些目录中呢
>
> 我们只要让ComponentScan扫描的时候忽略掉我们的配置文件即可

5.2.3.1 创建一个注解

```
public @interface ExcludeFromComponentScan {
//空注解即可
}
```

5.2.3.2 移动我们的配置文件到扫描目录中,并添加注解

```
@Configuration
@ExcludeFromComponentScan//添加我们的注解,用于后面过滤用
public class TestConfiguration {
  //  @Autowired
  //  IClientConfig config;

  @Bean
  public IRule ribbonRule() {
    return new RandomRule();
  }
}
```

5.2.3.3 修改主程序

> 添加注解ComponentScan 设置过滤条件

```
@SpringBootApplication
@EnableEurekaClient
@RibbonClient(name = "microservice-provider-user", configuration =
TestConfiguration.class)
@ComponentScan(excludeFilters = { @ComponentScan.Filter(type =
FilterType.ANNOTATION, value = ExcludeFromComponentScan.class) })//添加包扫描注解,
并设置过滤条件,过滤的类型为包含特定注解的类不会被扫描,当前指定的特定注解是我们自己写的注
解, 也就是如果对应的类上面有我们的自定义注解,就不会被扫描
public class ConsumerMovieRibbonApplication {

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

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

5.3 配置文件配置 Ribbon

> 参考官方文档,配置文件的优先级高于@RibbonClient 注解的优先级,同时高于 Feign 的默认配置
>
> 参考项目
>
> microservice-consumer-movie-ribbon-properties-customizing
>
> microservice-discovery-eureka 代码不变
>
> microservice-provider-user 代码不变

5.3.1 配置 movie的配置文件

> 添加以下内容

```
microservice-provider-user: #代表配置当前这个微服务的负载均衡
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule  #默认规则类 WeightedResponseTimeRule 参考官方文档 ,因为默认的是按照延迟等优先级做选择的,本地很难实现,所以使用随机的
```

5.3.2 修改 Controller

> 为了方便演示效果,修改 Controller 的findById方法代码

```
 @GetMapping("/movie/{id}")
  public User findById(@PathVariable Long id) {
    // http://localhost:7900/simple/
    // VIP virtual IP
    // HAProxy Heartbeat
    //使用负载均衡的方式去测试,方便演示效果
    ServiceInstance serviceInstance = this.loadBalancerClient.choose("microservice-provider-user");//注意调用的微服务的 id 必须和启动的 user 的微服务的 id 一致,因为前面修改过
    System.out.println("===" + ":" + serviceInstance.getServiceId() + ":" + serviceInstance.getHost() + ":" + serviceInstance.getPort());

    return this.restTemplate.getForObject("http://microservice-provider-user/simple/" + id, User.class);
  }
```

5.3.3 不同微服务不同负载规则

> 因为在配置文件中配置的是某个微服务的规则,所以启动其他微服务的时候会使用默认的规则,如果每个都不使用默认,单独配置即可,可以修改 user 的端口和名字来测试

5.4 Eureka中禁用 Ribbon

> 参考代码
>
> microservice-consumer-movie-ribbon-without-eureka
>
> microservice-discovery-eureka 代码不变
>
> microservice-provider-user 代码不变

5.4.1 修改 movie 的配置文件

```yaml
spring:
  application:
    name: microservice-consumer-movie-ribbon-without-eureka
server:
  port: 8010
eureka:
  client:
    healthcheck:
      enabled: true
    serviceUrl:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
ribbon:
  eureka:
   enabled: false   #禁用 rabbion
microservice-provider-user:  #给当前的微服务指定具体的地址,将不再有负载均衡
  ribbon:
    listOfServers: localhost:7900
```

5.4.2 测试

> 启动 Eureka user 和 movie 即可

六 Fegin

6.1 Fegin 简介使用

6.1.1 简介

Feign是一个声明式的Web服务客户端,要想使用 Fegin 只需要定义一个接口并且添加响应的注解即可,支持包括Feign注解和JAX-RS注解。Feign还支持可插拔编码器和解码器。Spring Cloud增加了对Spring MVC注解的支持，并使用Spring Web中默认使用的`HttpMessageConverters`。Spring Cloud集成Ribbon和Eureka以在使用Feign时提供负载均衡的http客户端

6.1.2 基本使用

> 参考项目
>
> microservice-consumer-movie-feign
>
> microservice-discovery-eureka 代码不变
>
> microservice-provider-user

6.1.2.1 定义 Feign 的接口

```
@FeignClient("microservice-provider-user")//定义使用哪个微服务的方法
public interface UserFeignClient {
  @RequestMapping(value = "/simple/{id}", method = RequestMethod.GET)
  public User findById(@PathVariable("id") Long id); // 两个坑：1. @GetMapping不支持   2. @PathVariable得设置value

  @RequestMapping(value = "/user", method = RequestMethod.POST)
  public User postUser(@RequestBody User user);

  // 该请求不会成功，只要参数是复杂对象，即使指定了是GET方法，feign依然会以POST方法进行发送请求。如果仍要使用,则可以在此处挨个定义参数
  @RequestMapping(value = "/get-user", method = RequestMethod.GET)
  public User getUser(User user);
}
```

6.1.2.2修改 movie 的 controller

> 将 controller 原先的 RestTemplate 删除,改为使用 Feign 的客户端

```
@RestController
public class MovieController {

  @Autowired
  private UserFeignClient userFeignClient;//注入我们自己写的客户端

  @GetMapping("/movie/{id}")
  public User findById(@PathVariable Long id) {
    return this.userFeignClient.findById(id);
  }

  @GetMapping("/test")
  public User testPost(User user) {
    return this.userFeignClient.postUser(user);
  }
```

```
  @GetMapping("/test-get")
  public User testGet(User user) {
    return this.userFeignClient.getUser(user);
  }
}
```

6.1.2.3 修改主程序

```
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients//启动 Feign 注解
public class ConsumerMovieFeignApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConsumerMovieFeignApplication.class, args);
  }
}
```

6.1.2.4 movie yml 配置文件

> 只需要配置 Eureka 服务端即可

```
spring:
  application:
    name: microservice-consumer-movie-feign
server:
  port: 7901
eureka:
  client:
    healthcheck:
      enabled: true
    serviceUrl:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
```

6.1.2.5 userController 添加几个测试方法

```
/**
   * 获取对象方式的参数
   * @param user
   * @return
   */
  @PostMapping("/user")
  public User postUser(@RequestBody User user) {
    return user;
```

```
    }

    // 该请求不会成功,因为在通过 Feign 请求复杂参数的时候 传递到这里的是 POST 请求,所以此处必须是 post 方式
    @GetMapping("/get-user")
    public User getUser(User user) {
        return user;
    }
```

6.2 自定义 Fegin 配置

> 参考
>
> microservice-consumer-movie-feign-customizing
>
> microservice-discovery-eureka 代码不变
>
> microservice-provider-user 代码不变

6.2.1 配置介绍

Spring Cloud 中心支持为每个 Feign 通过 @FeignClient注解name 属性进行命名(name 其实是 value 属性的别。Spring Cloud根据需要，使用`FeignClientsConfiguration`为每个已命名的客户端创建一个新的`ApplicationContext`集合(通过查看FeignClient注解的configuration属性可以得知FeignClientsConfiguration是默认属性类)。也就是FeignClient注解会创建一个 Spring 的子容器.这包含（除其他外）`feign.Decoder`，`feign.Encoder`和`feign.Contract`。

注意和 Ribbon 一样,官方不建议我们将配置文件放到`@ComponentScan` or `@SpringBootApplication`的扫描目录中

`FooConfiguration` does not need to be annotated with `@Configuration`. However, if it is, then take care to exclude it from any `@ComponentScan` that would otherwise include this configuration as it will become the default source for `feign.Decoder`, `feign.Encoder`, `feign.Contract`, etc., when specified. This can be avoided by putting it in a separate, non-overlapping package from any `@ComponentScan` or `@SpringBootApplication`, or it can be explicitly excluded in `@ComponentScan`.

serviceId已经过时了,现在使用 name

The `serviceId` attribute is now deprecated in favor of the `name` attribute.

注意 name 属性现在是必须的了

Previously, using the `url` attribute, did not require the `name` attribute. Using `name` is now required.

6.2.2 自定义配置

6.2.2.1 创建配置类

```
@Configuration
public class Configuration1 {
  @Bean
  public Contract feignContract() {
    return new feign.Contract.Default();// 默认的契约模式
  }
}
```

6.2.2.2 修改 UserFeignClient

```
@FeignClient(name = "microservice-provider-user", configuration =
Configuration1.class)//指定配置类为Configuration1
public interface UserFeignClient {
  @RequestLine("GET /simple/{id}")//注意此处不是 RequestMapping 注解
  public User findById(@Param("id") Long id);//注意参数的注解由PathVariable变成
Param注解
}
```

6.2.2.3 启动 Eureka ,user 和 movie 测试

> 由于上面的 FeignClient 删除了其他方法,所以 movie 的 controller中也要删除对应的请求
>
> 访问 movie 的对应方法

```
//保留着一个方法
  @GetMapping("/movie/{id}")
  public User findById(@PathVariable Long id) {
    return this.userFeignClient.findById(id);
  }
```

6.2.3 带 url 属性的使用方式

> Eurake 给我们提供了一个地址,可以查看当前注册的所有 app
>
> 地址是 Eurake 的访问地址/apps
>
> 在/apps 后面添加对应的 app 名字可以查看 app 信息

6.2.3.1 创建 Feign 客户端

```
//指定配置类为Configuration2,在指定了url的情况下 name 可以随便写
@FeignClient(name = "xxxx", url = "http://localhost:8761/", configuration =
Configuration2.class)
public interface FeignClient2 {
  @RequestMapping(value = "/eureka/apps/{serviceName}")//此处使用的是
RequestMapping 注解,代表访问的是在 url 的基础上添加此路径
  public String
findServiceInfoFromEurekaByServiceName(@PathVariable("serviceName") String
serviceName);
}
```

6.2.3.2 创建Configuration2配置文件

```
@Configuration
public class Configuration2 {
  @Bean
  public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
    return new BasicAuthRequestInterceptor("user", "password123");//此方法的作用是
用于返回 Eurake 的登录账号和密码 用于授权,否则会出现401异常
  }
}
```

6.2.3.3 修改 movieController

> 添加一下内容,

```
 @Autowired
 private FeignClient2 feignClient2;//注入客户端

@GetMapping("/{serviceName}")
  public String findServiceInfoFromEurekaByServiceName(@PathVariable String
serviceName) {
    return
this.feignClient2.findServiceInfoFromEurekaByServiceName(serviceName);//调用方法
  }
```

6.2.3.4 启动 Eurake .user 和 movie, 访问 movie 的地址进行测试

6.3 Fegin 日志

6.3.1 设置 yml

> 只有在 DEBUG 级别下, Feign 才会打印日志,
>
> 打印日志的设置方式 在 yml 中添加logging: level:(要打印日志的 Feign 的接口的权限定名称): DEBUG

6.3.2 修改 Feign 的配置文件

> 对要输出日志的 FeignClient 对应的 configuration 类中添加指定方法,比如此处指定的是 Configuration1

```
/**
    * 当配置输入日志属性的时候,设置输出的日志的内容
    * @return
    */
  @Bean
  Logger.Level feignLoggerLevel() {
    return Logger.Level.FULL;
  }
```

6.4 Fegin超时设置

> 参考最后的错误

6.5 其他

6.5.1 注册服务慢

> 参考地址http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_why_is_it_so_slow_to_register_a_service
>
> Being an instance also involves a periodic heartbeat to the registry (via the client's `serviceUrl`) with default duration 30 seconds. A service is not available for discovery by clients until the instance, the server and the client all have the same metadata in their local cache (so it could take 3 heartbeats). You can change the period using `eureka.instance.leaseRenewalIntervalInSeconds` and this will speed up the process of getting clients connected to other services. In production it's probably better to stick with the default because there are some computations internally in the server that make assumptions about the lease renewal period.
>
> 在服务器和客户端两边的元数据一直之前,服务注册可能需要发送三次心跳,每次的时间为30秒,所以时间比较长,可以通过eureka.instance.leaseRenewalIntervalInSeconds来修改心跳时间,但是不建议

6.5.2 Eureka 高可用

> 参考地址http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#spring-cloud-eureka-server-zones-and-regions

> The Eureka server does not have a backend store, but the service instances in the registry all have to send heartbeats to keep their registrations up to date (so this can be done in memory). Clients also have an in-memory cache of eureka registrations (so they don't have to go to the registry for every single request to a service).
>
> By default every Eureka server is also a Eureka client and requires (at least one) service URL to locate a peer. If you don't provide it the service will run and work, but it will shower your logs with a lot of noise about not being able to register with the peer.
>
> See also below for details of Ribbon support on the client side for Zones and Regions.
>
> Eureka 至少要有两台来保持高可用,每个 Eureka 既是一个 server 又是一个 client,每个 Eureka 都至少有一个对等 url(也就是它作为客户端,另外一个服务端就是它的对等 url)

高可用模式配置 参考地址http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_peer_awareness

```
---
spring:
  profiles: peer1
eureka:
  instance:
    hostname: peer1
  client:
    serviceUrl:
      defaultZone: http://peer2/eureka/  #将自己注册到了 peer2上面

---
spring:
  profiles: peer2
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1/eureka/  #将自己注册到了 peer1
```

> 参考代码
>
> microservice-discovery-eureka-ha
>
> 注意将依赖中的 安全认证删除,以免每次要求输入密码.仅测试用

```
#application-peer1.yml
spring:
  application:
    name: EUREKA-HA
    profiles: peer1  #参数为 peer1的时候执行
server:
```

```
  port: 8761
eureka:
  instance:
    hostname: peer1
  client:
    serviceUrl:
      defaultZone: http://peer2:8762/eureka/,http://peer3:8763/eureka/



#application-peer2.yml
server:
  port: 8762
spring:
  application:
    name: EUREKA-HA
  profiles: peer2  #参数为 peer2的时候执行
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/,http://peer3:8763/eureka/

#application-peer3.yml
server:
  port: 8763
spring:
  application:
    name: EUREKA-HA
  profiles: peer3   #参数为 peer3的时候执行
eureka:
  instance:
    hostname: peer3
  client:
    serviceUrl:
      defaultZon
```

在启动 ha项目的时候通过输入参数名(—spring.profiles=peer1 类推)来执行不同的操作,注意,启动第一个的时候因为连接不到另外的节点,会出错,忽略即可

> 微服务注册到高可用的 eureka 的时候,只需要将自己配置文件中的
>
> defaultZone: server1,server2 以此类推

6.5.3 其他配置属性

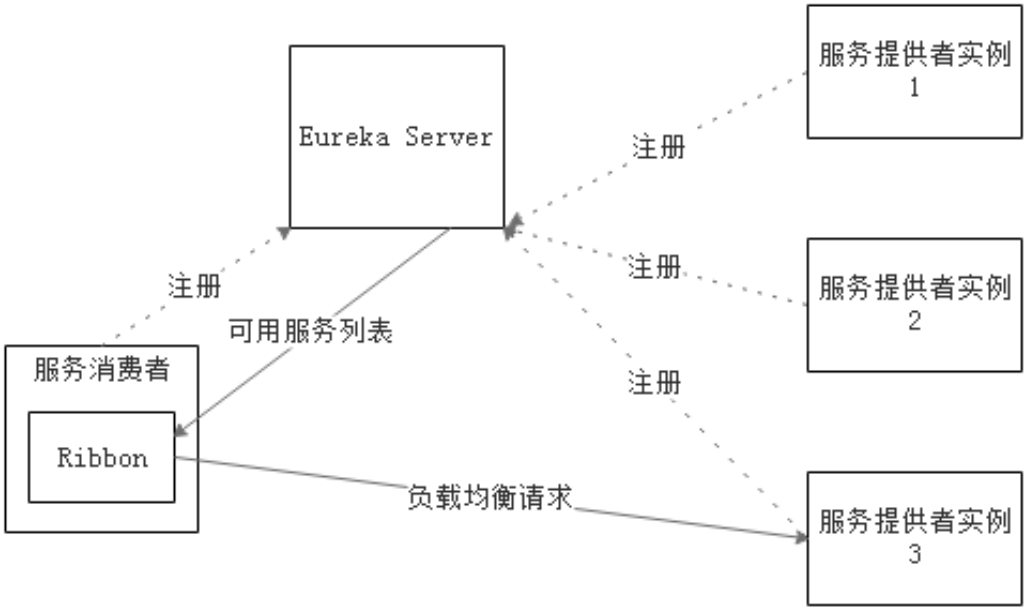
> 参考http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_appendix_compendium_of_configuration_properties

> eureka.instance.appname 这个属性会覆盖spring.application.name ,目的是在整合一些其他项目的时候防止因为与 springcloud 冲突导致无法获取spring.application.name的情况,比如swagger2
>
> eureka.instance.hostname 主机名:配置该地址后, 请去掉prefer-ip-address和instance-id两个属性,在 eureka 后台查看当前服务方位地址的时候可以发现是 hostname, 所以必须保证 hostname 对应的地址是可以访问的

七 Hystrix 熔断机制

7.1 超时,熔断机制简介

7.1.1 存在问题

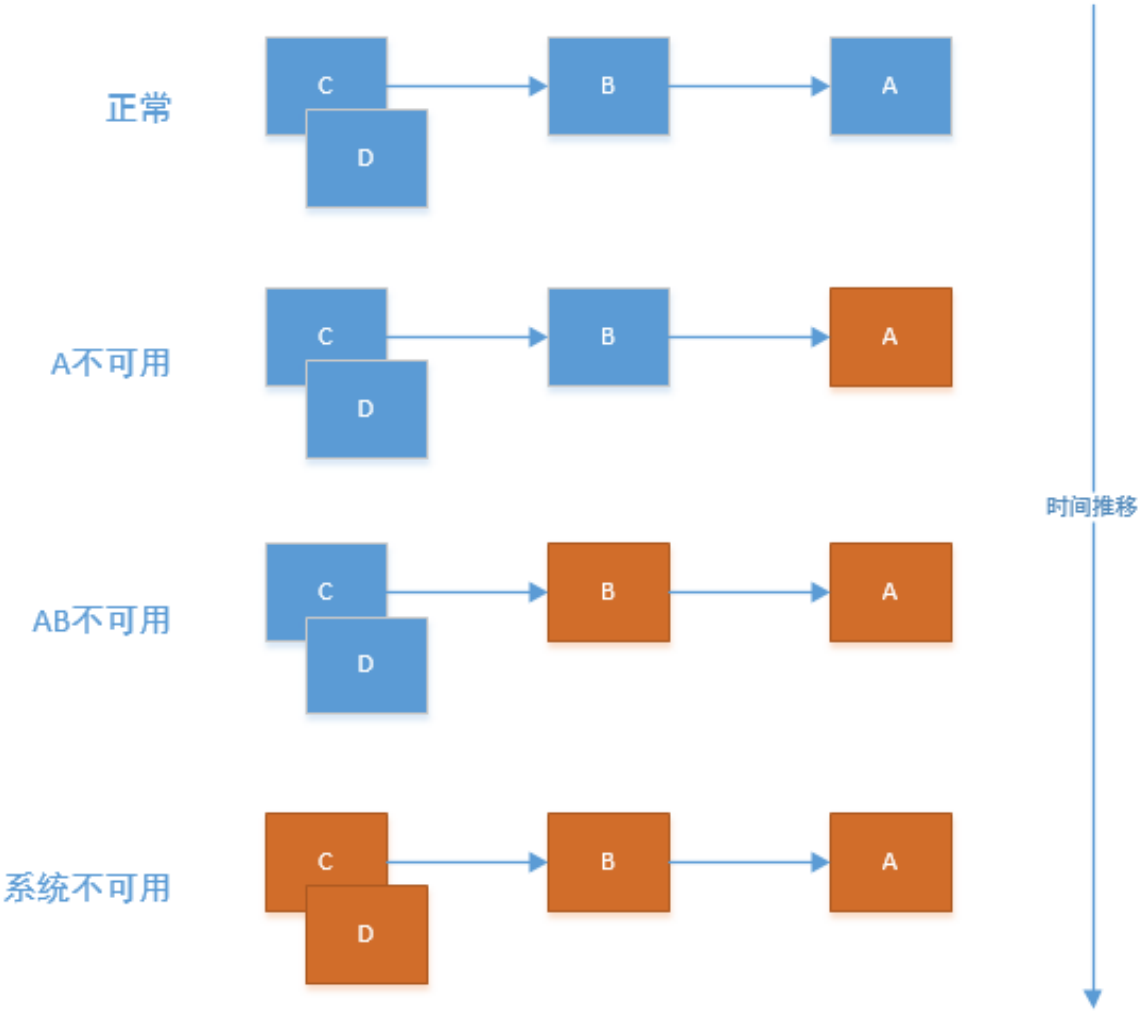


现在我们假设一下，服务提供者响应非常缓慢，那么消费者对提供者的请求就会被强制等待，直到服务返回。在高负载场景下，如果不做任何处理，这种问题很可能造成所有处理用户请求的线程都被耗竭，而不能响应用户的进一步请求。

7.1.2 雪崩效应

在微服务架构中通常会有多个服务层调用，大量的微服务通过网络进行通信，从而支撑起整个系统。各个微服务之间也难免存在大量的依赖关系。然而任何服务都不是100%可用的，网络往往也是脆弱的，所以难免有些请求会失败。基础服务的故障导致级联故障，进而造成了整个系统的不可用，这种现象被称为服务雪崩效应。服务雪崩效应描述的是一种因服务提供者的不可用导致服务消费者的不可用，并将不可用逐渐放大的过程。

A作为服务提供者，B为A的服务消费者，C和D是B的服务消费者。A不可用引起了B的不可用，并将不可用像滚雪球一样放大到C和D时，雪崩效应就形成了。



7.1.3 解决方案

7.1.3.1超时机制

> 通过网络请求其他服务时，都必须设置超时。正常情况下，一个远程调用一般在几十毫秒内就返回了。当依赖的服务不可用，或者因为网络问题，响应时间将会变得很长（几十秒）。而通常情况下，一次远程调用对应了一个线程/进程，如果响应太慢，那这个线程/进程就会得不到释放。而线程/进程都对应了系统资源，如果大量的线程/进程得不到释放，并且越积越多，服务资源就会被耗尽，从而导致资深服务不可用。所以必须为每个请求设置超时。

7.1.3.2 断路器模式

试想一下，家庭里如果没有断路器，电流过载了（例如功率过大、短路等），电路不断开，电路就会升温，甚至是烧断电路、起火。有了断路器之后，当电流过载时，会自动切断电路（跳闸），从而保护了整条电路与家庭的安全。当电流过载的问题被解决后，只要将关闭断路器，电路就又可以工作了。

同样的道理，当依赖的服务有大量超时时，再让新的请求去访问已经没有太大意义，只会无谓的消耗现有资源。譬如我们设置了超时时间为1秒，如果短时间内有大量的请求（譬如50个）在1秒内都得不到响应，就往往意味着异常。此时就没有必要让更多的请求去访问这个依赖了，我们应该使用断路器避免资源浪费。

断路器可以实现快速失败，如果它在一段时间内侦测到许多类似的错误（譬如超时），就会强迫其以后的多个调用快速失败，不再请求所依赖的服务，从而防止应用程序不断地尝试执行可能会失败的操作，这样应用程序可以继续执行而不用等待修正错误，或者浪费CPU时间去等待长时间的超时。断路器也可以使应用程序能够诊断错误是否已经修正，如果已经修正，应用程序会再次尝试调用操作。

断路器模式就像是那些容易导致错误的操作的一种代理。这种代理能够记录最近调用发生错误的次数，然后决定使用允许操作继续，或者立即返回错误。

7.2 Hystrix 基本使用

> 参考http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_circuit_breaker_hystrix_clients
>
> microservice-consumer-movie-ribbon-with-hystrix
>
> microservice-provider-user
>
> microservice-discovery-eureka
>
> 步骤:添加依赖,给当前项目主程序添加注解,开启 hystrix, 给具体方法添加注解,配置失败后的执行方法

7.2.1 添加依赖

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

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

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

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

7.2.2 主程序添加注解

```
@SpringBootApplication
```

```
@EnableEurekaClient
@EnableCircuitBreaker//启用 hystrix
public class ConsumerMovieRibbonApplication {

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

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

7.2.3 给对应的方法添加注解

```
@RestController
public class MovieController {
  @Autowired
  private RestTemplate restTemplate;

  @GetMapping("/movie/{id}")
  @HystrixCommand(fallbackMethod = "findByIdFallback")//给当前方法启用熔断,失败的时候执行findByIdFallback方法
  public User findById(@PathVariable Long id) {
    return this.restTemplate.getForObject("http://microservice-provider-user/simple/" + id, User.class);
  }

  public User findByIdFallback(Long id) {//参数必须和原始方法一致
    User user = new User();
    user.setId(0L);
    return user;
  }
}
```

7.3 上下文切换

> 参考http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_propagating_the_security_context_or_using_spring_scopes
>
> HystrixProperty的value属性的意思参考
>
> https://github.com/Netflix/Hystrix/wiki/Configuration#execution.isolation.strategy
>
> 参考代码

> microservice-consumer-movie-ribbon-with-hystrix-propagation
>
> user,eureka

7.3.1 修改要添加熔断方法的注解

```
@RestController
public class MovieController {
  @Autowired
  private RestTemplate restTemplate;

  @GetMapping("/movie/{id}")
  @HystrixCommand(fallbackMethod = "findByIdFallback", commandProperties =
@HystrixProperty(name = "execution.isolation.strategy", value = "SEMAPHORE"))//指
定失败时候执行方法的线程和当前线程一致
  public User findById(@PathVariable Long id) {
    return this.restTemplate.getForObject("http://microservice-provider-
user/simple/" + id, User.class);
  }

  public User findByIdFallback(Long id) {
    User user = new User();
    user.setId(0L);
    return user;
  }
}
```

7.3.2问题

> The same thing applies if you are using `@SessionScope` or `@RequestScope`. You will know when you need to do this because of a runtime exception that says it can't find the scoped context.
>
> You also have the option to set the `hystrix.shareSecurityContext` property to `true`. Doing so will auto configure an Hystrix concurrency strategy plugin hook who will transfer the `SecurityContext` from your main thread to the one used by the Hystrix command. Hystrix does not allow multiple hystrix concurrency strategy to be registered so an extension mechanism is available by declaring your own `HystrixConcurrencyStrategy` as a Spring bean. Spring Cloud will lookup for your implementation within the Spring context and wrap it inside its own plugin.
>
> 如果在使用@SessionScope or @RequestScope的时候提示找不到上下文 context, 则可以指定value = "SEMAPHORE"
>
> 或者设置hystrix.shareSecurityContext为 true 它将会自动配置一个钩子来切换到你当前添加了HystrixCommand注解的线程 ,注意此注解是netfilx s1.2.0才出现的

7.3 Hystrix 健康状态

> 参考项目microservice-consumer-movie-ribbon-with-hystrix
>
> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_health_indicator_4
>
> 访问微服务的/health 查看状态
>
> 注意 hysrix 的
>
> 注意To enable the Hystrix metrics stream include a dependency on `spring-boot-starter-actuator`. 必须添加这个依赖

7.4 Feign 整合 Hystrix

> 参考http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#spring-cloud-feign-hystrix
>
> microservice-consumer-movie-feign-with-hystrix
>
> 注意自Dalston版本 开始 springcloud 默认是关闭了feign 的断路器功能,需要单独打开
>
> 在需要使用断路器的配置文件中
>
> feign: hystrix: enabled: true

7.4.1 定义 FeignClient接口

```
@FeignClient(name = "microservice-provider-user", fallback =
HystrixClientFallback.class)//fallback代表失败时候执行指定类中的方法,方法和当前接口的方法是一样
public interface UserFeignClient {
  @RequestMapping(value = "/simple/{id}", method = RequestMethod.GET)
  public User findById(@PathVariable("id") Long id);
}
```

7.4.2 定义HystrixClientFallback类

```
@Component
public class HystrixClientFallback implements UserFeignClient {

  @Override
  public User findById(Long id) {
    User user = new User();
    user.setId(0L);
    return user;
  }
}
```

7.4.3 主类添加 feign 注解

```
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
@EnableCircuitBreaker
public class ConsumerMovieFeignApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConsumerMovieFeignApplication.class, args);
  }
}
```

7.5 Feign 禁用 HyStrix

> 参考示例代码microservice-consumer-movie-feign-customizing-without-hystrix

7.5.1 全局禁用

> 配置文件中添加
>
> feign: hystrix: enabled: false

7.5.2 局部禁用

> 对要禁用的 FeignClient 接口的注解中添加 configuration 参数,指定一个配置类configuration = Configuration2.class

```
@Configuration
public class Configuration2 {
  @Bean
  public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
    return new BasicAuthRequestInterceptor("user", "password123");
  }

  @Bean
  @Scope("prototype")
  public Feign.Builder feignBuilder() {//禁用 hystrix
    return Feign.builder();
  }
}
```

7.6 Feign fallbackFactory

> 参考microservice-consumer-movie-feign-with-hystrix-hystrix-factory
>
> user ,eureka

> 步骤,1定义 Feign 接口,在注解内指定参数fallbackFactory 为某个类
>
> 2 编写fallbackFactory的具体类,重写 create 方法,在内部返回 fallback 的时候真正执行类的代码
>
> 3 编写真正的执行类,此类可以是一个累也可以是一个接口,但是必须是上面定义的 Feign 接口的子类

7.6.1 Feign 接口

```
@FeignClient(name = "microservice-provider-user", /*fallback =
HystrixClientFallback.class, */fallbackFactory =
HystrixClientFactory.class)//fallbackFactory的时候会创建HystrixClientFactory对象,并执行 create 获取实际类
public interface UserFeignClient {
  @RequestMapping(value = "/simple/{id}", method = RequestMethod.GET)
  public User findById(@PathVariable("id") Long id);
}
```

7.6.2 HystrixClientFactory

```
/**
 * 实现指定接口,泛型是用于给哪个 Feign 接口实现 fallback
 */
@Component
public class HystrixClientFactory implements FallbackFactory<UserFeignClient> {

  private static final Logger LOGGER =
LoggerFactory.getLogger(HystrixClientFactory.class);

  @Override
  public UserFeignClient create(Throwable cause) {
    HystrixClientFactory.LOGGER.info("fallback; reason was: {}",
cause.getMessage());
    return new UserFeignClientWithFactory() { //返回最终处理失败方法的对象,此对象内部必须包含和 Feign 接口一样的方法,所以必须是 Feign 的子类
      @Override
      public User findById(Long id) {
        User user = new User();
        user.setId(-1L);
        return user;
      }
    };
  }
}
```

7.6.3UserFeignClientWithFactory 接口

```
public interface UserFeignClientWithFactory extends UserFeignClient {
//此处可以是接口,也可以是实现类,取决于具体的业务需求
}
```

7.7 Hystrix DashBoard

> 用于图形化监视的
>
> microservice-hystrix-dashboard
>
> microservice-consumer-movie-ribbon-with-hystrix 用于被监视
>
> microservice-provider-user
>
> microservice-hystrix-dashboard

7.7.1 dashboard 依赖

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

7.7.2 主程序

```
@EnableHystrixDashboard//启用 dashboard
@SpringBootApplication
public class EurekaApplication {
  public static void main(String[] args) {
    SpringApplication.run(EurekaApplication.class, args);
  }
}
```

7.8 Turbine 集群监控

7.8.1监控某个应用

> 参考microservice-hystrix-turbine
>
> 步骤:添加依赖,注解@EnableTurbine
>
> Turbine需要注册到 eureka

```yaml
server:
  port: 8031
spring:
  application:
    name: microservice-hystrix-turbine
eureka:
  client:
    serviceUrl:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
turbine:
  aggregator:
    clusterConfig: microservice-consumer-movie-ribbon-with-hystrix # 指定聚合哪些集群,多个使用,分割
    appConfig: microservice-consumer-movie-ribbon-with-hystrix #配置要监控哪些应用
   # clusterNameExpression: "'default'" #集群的名字
```

启动 eureka 和 user, 通过修改端口的方式启动多个microservice-consumer-movie-ribbon-with-hystrix

通过访问 turbine 项目 localhost:8031/trubine.stream?cluster=microservice-consumer-movie-ribbon-with-hystrix 的方式可以看到结果

启动 dashboard 项目,将上面的地址添加到 dashboard 中进行查看

分别访问两个microservice-consumer-movie-ribbon-with-hystrix的 movie/1 ,然后查看 dashboard. 发现不管访问哪个地址都可以被监控到

7.8.2 监控多个应用

修改 turbine 的配置文件

```yaml
server:
  port: 8031
spring:
  application:
    name: microservice-hystrix-turbine
eureka:
  client:
    serviceUrl:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
turbine:
  aggregator:
    clusterConfig: default # 指定聚合哪些集群,多个使用,分割,此处使用 default
```

```
  appConfig: microservice-consumer-movie-ribbon-with-hystrix, microservice-
consumer-movie-ribbon-with-hystrix #配置要监控哪些应用
  clusterNameExpression: "'default'" #集群的名字
```

> 启动 microservice-consumer-movie-ribbon-with-hystrix,microservice-consumer-movie-ribbon-with-hystrix ,user,eureka,dashboard
>
> 访问两个movie/1,然后访问 localhost:8031/trubine.stream ,可以看到数据
>
> 然后将 localhost:8031/trubine.stream 放到 dashboard 中可以查看到多个应用

八 Zuul

8.1 API GateWay

> 介绍参考http://blog.daocloud.io/microservices-2/

8.2 zuul 基本使用

> 参考代码microservice-gateway-zuul
>
> 每次使用不同的配置文件
>
> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_router_and_filter_zuul
>
> **hystrix 的默认隔离级别是 Thread, zuul 是SEMAPHORE**
>
> Default Hystrix isolation pattern (ExecutionIsolationStrategy) for all routes is SEMAPHORE. `zuul.ribbonIsolationStrategy` can be changed to THREAD if this isolation pattern is preferred.

To enable it, annotate a Spring Boot main class with `@EnableZuulProxy`, and this forwards local calls to the appropriate service 使用EnableZuulProxy注解开启

the Zuul starter does not include a discovery client, so for routes based on service IDs you need to provide one of those on the classpath as well (e.g. Eureka is one choice). zuul 不包括客户端发现,所以我们需要配置一个服务发现,比如 eureka

8.2.1 创建 zuul 项目

> 参考microservice-gateway-zuul的 bak1版本
>
> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#netflix-zuul-reverse-proxy

8.2.1.1 pom 文件

```
<dependencies>
    <!--zuul 依赖-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-zuul</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
    <!--新版需要添加,否则访问 routes 的时候会无权限-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>

    </dependencies>
```

8.2.1.2 配置文件

```
spring:
  application:
    name: microservice-gateway-zuul
server:
  port: 8040
eureka:
  client:
    service-url:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
 #用于查看 routes 操作时候的密码
security:
    user:
      name: user
      password: password123
```

8.2.1.3 主类

```
@SpringBootApplication
@EnableZuulProxy //EnableZuulProxy是一个组合注解,可以自动注册到 eurake 以及做
fallback
public class ZuulApplication {
  public static void main(String[] args) {
    SpringApplication.run(ZuulApplication.class, args);
  }
}
```

8.2.1.4 启动测试

> 启动我们的 eurake,user 和 zuul
>
> 通过访问 zuul 的地址 localhost:8040/${user服务的 serviceid}/接口 http://localhost:8040/microservice-provider-user/simple/1 发现也可以访问 user 的服务器,也就是我们的 zuul 已经代理了我们的 user 服务

8.2.1.5 问题

> 每次我们需要通过服务名称去带你服务,非常麻烦.我们可以给服务添加路径映射,然后快速访问服务
>
> 在外面的 zuul 的配置文件中添加
>
> ```
> zuul:
>   ignoredServices: microservice-consumer-movie-ribbon-with-hystrix  #zuul 会
> 自动代理 eureka 上面的所有服务,如果我们不想让它代理某个服务,可以这样设置,如果多个
> 以,分割
>   routes:
>     microservice-provider-user: /user/**   #指定服务的代理路径  microservice-
> provider-user的访问路径是/ user
> ```

8.3 path 方式反向代理

> 参考配置文件 bak2
>
> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/multi/multi__router_and_filter_zuul.html#netflix-zuul-reverse-proxy

8.3.1 修改配置文件

```
spring:
  application:
    name: microservice-gateway-zuul
server:
  port: 8040
```

```yaml
eureka:
  client:
    service-url:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
 #用于查看 routes 操作时候的密码
security:
    user:
      name: user
      password: password123
zuul:
  routes:
    abc: #只要保证唯一即可,随便写
      path: /user-path  #代理后的地址是当前地址
      serviceId: microservice-provider-user  #代理当前的微服务
    #相当于上面的问题中的microservice-provider-user: /user/**
```

8.3.2 匹配注意事项

> This means that http calls to "/myusers" get forwarded to the "users_service" service. The route has to have a "path" which can be specified as an ant-style pattern, so "/myusers/*" only matches one level, but "/myusers/**" matches hierarchically.
>
> 路径的匹配规则 **代表任意 *代表一级 ?匹配单个字符
>
> The location of the backend can be specified as either a "serviceId" (for a service from discovery) or a "url" (for a physical location),
>
> 我们可以使用 url 来代替serviceId,注意 url 就是对应 id 的service的实际地址 ,参考 bak3配置文件
>
> ```yaml
> zuul:
>   routes:
>     abc:
>       path: /user-url/**
>       url: http://192.168.85.1:7900/  #此处http://192.168.85.1:7900/ 实际是microservice-provider-user 的访问地址
> ```
>
> 但是注意 These simple url-routes don't get executed as a `HystrixCommand` nor do they loadbalance multiple URLs with Ribbon. To achieve this, you can specify a `serviceId` with a static list of servers:
>
> 使用 url 的方式不会被 ribbon 进行负载均衡

8.3 zuul 负载均衡

> 我们如果启动多个服务想要通过 zuul 实现负载聚恒如何设置呢

> 参考 bak4配置文件

8.3.1 zuul 项目的配置文件

```
spring:
  application:
    name: microservice-gateway-zuul
server:
  port: 8040
eureka:
  client:
    service-url:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
  #用于查看 routes 操作时候的密码
security:
    user:
      name: user
      password: password123
zuul:
  routes:
    abc:
      path: /user-url/**
      service-id: microservice-provider-user #代理当前的 service, 这里要和下面的服务器列表所在的服务名一致
ribbon:
  eureka:
    enabled: false
microservice-provider-user:     # 这边是ribbon要请求的微服务的serviceId,要和上面代理的 service-id 一样
  ribbon:
    listOfServers: http://localhost:7900,http://localhost:7901 ##当前 service 的对应地址
```

8.3.2 启动测试

> 通过修改 user 的端口,实现启动多个 user 服务器,然后通过 zuul 代理访问服务器,查看项目输出的控制台内容发现可以实现负载均衡

8.4 zuul 正则表达式配置规则

> 参考microservice-gateway-zuul-reg-exp
>
> user, eureka

8.4.1 主程序

```
@SpringBootApplication
@EnableZuulProxy //启用代理
public class ZuulApplication {
  public static void main(String[] args) {
    SpringApplication.run(ZuulApplication.class, args);
  }

  /**
   * 此方法的作用,设置正则表达式的方式进行地址映射
   * 映射规则 将 服务器的 appname-version 映射为 /version/appname 的格式
   * @return
   */
  @Bean
  public PatternServiceRouteMapper serviceRouteMapper() {
    return new PatternServiceRouteMapper("(?<name>^.+)-(?<version>v.+$)",
"${version}/${name}");
  }
}
```

8.4.2 修改 user 服务器的名称

> 修改yml 配置文件中的spring.application.name 为:microservice-provider-user- v1

8.4.3 启动 eureka,user,zuul,测试

> 通过访问 localhost:8040/v1/microservice-provider-user/simple/1 发现可以访问到 user 的服务

8.5 地址前缀

> 参考microservice-gateway-zuul
>
> bak5配合文件

8.5.1 全局路径前缀

```
spring:
  application:
    name: microservice-gateway-zuul
server:
  port: 8040
eureka:
  client:
    service-url:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
 #用于查看 routes 操作时候的密码
security:
```

```
    user:
      name: user
      password: password123
zuul:
  prefix: /api #代表访问任何代理的服务前加前缀 api
  strip-prefix: false #此属性需要配合prefix使用,当配置为false 的时候 prefix应该是被代理的服务器的实际接口的访问 前缀,比如我们的microservice-provider-user内有个/simple的接口,那么此处就应该填写/sipmle,使用场景是以免有些微服务定义的 homepage 是带有自己的 path 的情况
logging:
  level:
    com.netflix: DEBUG
```

8.5.2 局部路径前缀

> 给某个服务添加前缀

```
zuul:
 routes:
   users: #代表当前服务的访问路径
     path: /myusers/**
     stripPrefix: false
```

8.5.3 过滤某些路径

```
zuul:
 ignoredPatterns: /**/admin/**  #细粒度的过滤某些路径
```

8.5.4 批量代理

> If you need your routes to have their order preserved you need to use a YAML file as the ordering will be lost using a properties file. For example: 注意使用以下方式的时候必须使用 yml 文件

```
zuul:
 routes:
   users: #users 服务使用以下路径代理
     path: /myusers/**
   legacy: #除了上面的之外使用以下代理路径
     path: /**
```

8.6 zuul 上传文件

> 参考代码microservice-file-upload

> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/multi/multi_*router_and_filter_zuul.html*#uploading_files_through_zuul

8.6.1 yml 配置文件

```
server:
  port: 8050
eureka:
  client:
    serviceUrl:
      defaultZone: http://user:password123@localhost:8761/eureka/
  instance:
    prefer-ip-address: true
spring:
  application:
    name: microservice-file-upload
  http:
    multipart: ##定义上传文件的相关信息,此处的配置并不是 zul 的配置
      max-file-size: 2000Mb      # Max file size, 默认1M  mb 不可以缺少
      max-request-size: 2500Mb   # Max request size, 默认10M
 #用于查看 routes 操作时候的密码
security:
    user:
      name: user
      password: password123
```

8.6.2 FileUploadController

```
@Controller
public class FileUploadController {
  /**
   * 上传文件
   * 测试方法:
   * 有界面的测试: http://localhost:8050/index.html
   * 使用命令: curl -F "file=@文件全名" localhost:8050/upload
   * ps.该示例比较简单, 没有做IO异常、文件大小、文件非空等处理
   * @param file 待上传的文件
   * @return 文件在服务器上的绝对路径
   * @throws IOException IO异常
   */
  @RequestMapping(value = "/upload", method = RequestMethod.POST)
  public @ResponseBody String handleFileUpload(@RequestParam(value = "file",
required = true) MultipartFile file) throws IOException {
    byte[] bytes = file.getBytes();
    File fileToSave = new File(file.getOriginalFilename());
    FileCopyUtils.copy(bytes, fileToSave);
    return fileToSave.getAbsolutePath();
```

```
  }
}
```

8.6.3 主程序

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

8.6.4 index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
    <form method="POST" enctype="multipart/form-data" action="/upload">
        File to upload:
        <input type="file" name="file">
        <input type="submit" value="Upload">
    </form>
</body>
</html>
```

8.6.5 测试

> 项目中的 index.html 是需要访问的当前项目直接上传的方式,而并不是通过 zuul 的方式,所以我们需要将 form 表单的地址修改掉,将上传地址中的/upload 修改为zuul 代理后的地址
>
> 此处通过 curl 的方式上传
>
> 在文件所在的目录 执行 curl -F "file@文件名" localhost:8040/microservice-file-upload/upload

8.6.6 问题

> 1.文件太大
>
> 如果上传的文件较大的时候,我们需要在 zuul 那边也配置文件上传大小的限制
>
> 如果不想配置 则可以通过在 请求地址的服务名字前添加 zuul 的方式绕过 springmvc 的限制

```
If you @EnableZuulProxy you can use the proxy paths to upload files and it
should just work as long as the files are small. For large files there is an
alternative path which bypasses the Spring DispatcherServlet (to avoid
multipart processing) in "/zuul/*". I.e. if
zuul.routes.customers=/customers/** then you can POST large files to
"/zuul/customers/*". The servlet path is externalized via zuul.servletPath.
Extremely large files will also require elevated timeout settings if the
proxy route takes you through a Ribbon load balancer,
```

执行 curl -F "file@文件名" localhost:8040/zuul/microservice-file-upload/upload 测试上传大文件发现新的问题2

问题2:超时

在 zuul 的配置文件中添加 设置超时时间

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:
60000
ribbon:
  ConnectTimeout: 3000
  ReadTimeout: 60000
```

问题3: 文件太大的情况下可能会出现堆内存移除

解决方案,修改tomcat 的内存 -Xms200M -Xmx 4000M 之类的方式设置大内存 或者在开发工具的运行选项中给 jvm 添加内容

问题4 Feign 提供文件上传支持,

```
@FeignClient(name = "xxx", url = "http://www.itmuch.com/", configuration =
TestFeignClient.FormSupportConfig.class)//在 Feign的配置文件类内部提供解码对象
public interface TestFeignClient {
    @PostMapping(value = "/test",
            consumes = {MediaType.APPLICATION_FORM_URLENCODED_VALUE},// form
表单数据
            produces = {MediaType.APPLICATION_JSON_UTF8_VALUE}
            )
    void post(Map<String, ?> queryParam);
    class FormSupportConfig {
        @Autowired
        private ObjectFactory<HttpMessageConverters> messageConverters;
        // new一个form编码器，实现支持form表单提交
        @Bean
        public Encoder feignFormEncoder() {
            return new SpringFormEncoder(new
SpringEncoder(messageConverters));
        }
        // 开启Feign的日志
```

```
        @Bean
        public Logger.Level logger() {
            return Logger.Level.FULL;
        }
    }
}
```

8.7 禁用 zuul 的一些功能

> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/multi/multi_*router_and_filter_zuul.html*#plain_embedded_zuul

You can also run a Zuul server without the proxying, or switch on parts of the proxying platform selectively, if you use `@EnableZuulServer` (instead of `@EnableZuulProxy`). Any beans that you add to the application of type `ZuulFilter` will be installed automatically, as they are with `@EnableZuulProxy`, but without any of the proxy filters being added automatically.

我们可以使用EnableZuulServer来代替EnableZuulProxy注解,EnableZuulServer只提供了最基本的使用,并不包含反向代理,通过查看EnableZuulServer源代码可以发现,我们可以通过ZuulProperties属性来进行配置自己想要的功能

8.8 zuul fallback

> 如果 zuul 代理的微服务出现问题,会出现404或者其他错误,这样的话页面体验效果不佳,我们知道 hystrix 和 feign 都 fallback 功能, 其实 zuul 也有 fallback 功能,而且使用非常简单
>
> 参考microservice-gateway-zuul-fallback

8.8.1 yml 文件

```yaml
spring:
  application:
    name: microservice-gateway-zuul
server:
  port: 8040
eureka:
  client:
    service-url:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000
ribbon:
  ConnectTimeout: 3000
```

```
  ReadTimeout: 60000
 #用于查看 routes 操作时候的密码
security:
    user:
      name: user
      password: password123
```

8.8.2 MyFallbackProvider

> 用于处理 fallback

```
package com.qianfeng.cloud.fallback;

import org.springframework.cloud.netflix.zuul.filters.route.ZuulFallbackProvider;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.client.ClientHttpResponse;
import org.springframework.stereotype.Component;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;

@Component//注意需要让 spring 创建对象,当存在此类型对象时候, springcloud 会自动注册
public class MyFallbackProvider implements ZuulFallbackProvider
{//FallbackProvider代替了ZuulFallbackProvider
  @Override
  public String getRoute() {
    return "microservice-provider-user";
  }

  /**
   * 获取fallback 时候的 http 响应
   * @return
   */
  @Override
  public ClientHttpResponse fallbackResponse() {
    return new ClientHttpResponse() {
      @Override
      public HttpStatus getStatusCode() throws IOException {//响应码,正常应该根据异
常类型进行判断
        return HttpStatus.BAD_REQUEST;
      }

      @Override
      public int getRawStatusCode() throws IOException {//错误码
        return HttpStatus.BAD_REQUEST.value();
      }
```

```
        @Override
        public String getStatusText() throws IOException {//错误码对应的文本描述
          return HttpStatus.BAD_REQUEST.getReasonPhrase();
        }

        @Override
        public void close() {
        }

        @Override
        public InputStream getBody() throws IOException {//响应正文
          return new ByteArrayInputStream(("fallback" +
MyFallbackProvider.this.getRoute()).getBytes());
        }

        @Override
        public HttpHeaders getHeaders() {//响应头
          HttpHeaders headers = new HttpHeaders();
          headers.setContentType(MediaType.APPLICATION_JSON);
          return headers;
        }
      };
    }
}
```

8.8.3 主程序

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

8.8.4 启动测试

1. 启动 Eureka,user 服务,和当前的 zuul
2. 测试 zuul 代理 user 是正常的
3. 关闭 user
4. 再次通过 zuul 代理 user 访问发现返回 fallback 内容

8.9 sidecar 整合异构语言微服务

> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/multi/multi_*polyglot_support_with_sidecar.html*#polyglot_support_with_sidecar
>
> 在我们的项目中,实际可能不止一种语言编写服务,那么不同的语言之间如何进行使用呢,我们可以使用 sidecar
>
> Sidecar是作为一个代理的服务来间接性的让其他语言可以使用Eureka等相关组件。通过与Zuul的来进行路由的映射，从而可以做到服务的获取，然后可以使用Ribbon，Feign对服务进行消费，以及对Config Server的间接性调用
>
> 在你的项目中使用Sidecar，需要添加依赖，其group为`org.springframework.cloud`，artifact id为`spring-cloud-netflix-sidecar`。(这是以maven依赖的方式)
>
> 启用Sidecar，创建一个Spring Boot应用程序，并在在应用主类上加上`@EnableSidecar`注解。该注解包含`@EnableCircuitBreaker`，`@EnableDiscoveryClient`以及`@EnableZuulProxy`。Run the resulting application on the same host as the non-jvm application. (这句不太会翻译，我的理解为：在与非jvm应用程序相同的主机上运行生成的应用程序)注：这里的生成应该是通过代理产生的服务。
>
> 配置Sidecar，在application.yml中添加`sidecar.port`和`sidecar.health-uri`。`sidecar.port`属性是非jre程序监听的端口号，这就是Sidecar可以正确注册应用到Eureka的原因。`sidecar.health-uri`是非jre应用提供的一个对外暴露的可访问uri地址，在该地址对应的接口中需要实现一个模仿Spring Boot健康检查指示器的功能。它需要返回如下的json文档。(**注**：通过返回一个json，其用status字段来标识你的应用的服务状态，是up还是down，sidecar会将该状态报告给eureka注册中心从而实现你的服务的状态可用情况。**简单的说就是用来控制sidecar代理服务的状态！**)
>
> 参考代码
>
> microservice-sidecar
>
> node-service

8.9.1 异构服务 node

> 此处使用 node 实现一个简单服务端

```
var http = require('http');//引入指定的服务
var url = require("url");
var path = require('path');

// 创建server
var server = http.createServer(function(req, res) {
  // 获得请求的路径
  var pathname = url.parse(req.url).pathname;
  res.writeHead(200, { 'Content-Type' : 'application/json; charset=utf-8' });
  // 访问http://localhost:8060/, 将会返回{"index":"欢迎来到首页"}
  if (pathname === '/') {//如果请求的是首页返回以下内容
    res.end(JSON.stringify({ "index" : "欢迎来到首页" }));
  }
```

```
  // 访问http://localhost:8060/health，将会返回{"status":"UP"}
  else if (pathname === '/health.json') {//如果请求的是 health.json 返回以下内容
    res.end(JSON.stringify({ "status" : "UP" }));//此处的 up down 会影响到 eureka
的状态,我们的 sidecar 的项目会请求这个地址 获取到状态后显示在 eureka 的后台,如果返回
DOWN 则 eureka 会显示 Down
  }
  // 其他情况返回404
  else {
    res.end("404");
  }
});
// 创建监听，并打印日志
server.listen(8060, function() {//监听8060端口
  console.log('listening on localhost:8060');
});
```

8.9.2 sidercar 模块

8.9.2.1 yml 文件

```
spring:
  application:
    name: microservice-sidecar
server:
  port: 8070
eureka: #注册到 eureka
  client:
    service-url:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
sidecar:
  port: 8060 #异构的端口
  health-uri: http://localhost:8060/health.json #异构的地址,这个接口的目的是检查健康
状态的,异构服务需要提供一个地址用于获取健康状态
```

8.9.2.2 主程序

```
@SpringBootApplication
@EnableSidecar//启用 sidecar
public class SidecarApplication {
  public static void main(String[] args) {
    SpringApplication.run(SidecarApplication.class, args);
  }
}
```

8.9.3测试

1. 启用 eureka
2. 启动 microservice-sidecar
3. 启动 zuul
4. 通过 zuul 访问microservice-sidecar的首页,发现返回的是 node 的数据

8.9.4 Ribbon 中调用 sidecar

如果要使用 ribbon 调用 sidercar 的服务,只需要 restTemplate.getForObject("http://sidecar微服务地址" +参数 返回值类型.class); 这种方式即可

8.9.5 sidecard 缺陷

> To enable the Sidecar, create a Spring Boot application with `@EnableSidecar`. This annotation includes `@EnableCircuitBreaker`, `@EnableDiscoveryClient`, and `@EnableZuulProxy`. Run the resulting application on the same host as the non-jvm application.
>
> 异构服务必须运行在和当前调用异构服务的程序一样的 host 上面,否则需要配置 eureka.instance.hostname 来将服务的地址设置为异构的地址
>
> 如果异构微服务太多, sidecar 也不是很适合,因为还要部署高可用,需要更多的sidecar

8.10 zuul 过滤器

> 通过Filter，我们可以实现安全控制，比如，只有请求参数中有用户名和密码的客户端才能访问服务端的资源
>
> 通过继承ZuulFilter然后覆写4个方法，就可以实现一个简单的过滤器，下面就相关注意点进行说明
>
> filterType：返回一个字符串代表过滤器的类型，在zuul中定义了四种不同生命周期的过滤器类型，具体如下：
>
> - `pre`：可以在请求被路由之前调用
> - `route`：在路由请求时候被调用
> - `post`：在route和error过滤器之后被调用
> - `error`：处理请求时发生错误时被调用
>
> Zuul的主要请求生命周期包括“pre”，“route”和“post”等阶段。对于每个请求，都会运行具有这些类型的所有过滤器。
>
> `filterOrder`：通过int值来定义过滤器的执行顺序
>
> `shouldFilter`：返回一个boolean类型来判断该过滤器是否要执行，所以通过此函数可实现过滤器的开关。在上例中，我们直接返回true，所以该过滤器总是生效

> `run`：过滤器的具体逻辑。需要注意，这里我们通过`ctx.setSendZuulResponse(false)`令zuul过滤该请求，不对其进行路由，然后通过`ctx.setResponseStatusCode(401)`设置了其返回的错误码
>
> 过滤器间的协调 过滤器没有直接的方式来访问对方。 它们可以使用RequestContext共享状态，这是一个类似Map的结构，具有一些显式访问器方法用于被认为是Zuul的原语，内部是使用ThreadLocal实现的，有兴趣的同学可以看下源码。
>
> 参考代码microservice-gateway-zuul-filter
>
> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/multi/multi**router_and_filter_zuul.html#**literal_enablezuulserver_literal_filters

8.10.1 编写过滤器

```
package com.qianfeng.cloud;

import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.servlet.http.HttpServletRequest;

public class PreZuulFilter extends ZuulFilter {
  private static final Logger LOGGER =
LoggerFactory.getLogger(PreZuulFilter.class);

  /**
   * 是否启用过滤器，true 启用
   * @return
   */
  @Override
  public boolean shouldFilter() {
    return true;
  }

  /**
   * 过滤器执行的时候
   * @return
   */
  @Override
  public Object run() {
    HttpServletRequest request = RequestContext.getCurrentContext().getRequest();
    String host = request.getRemoteHost();
    PreZuulFilter.LOGGER.info("请求的host:{}", host);
    return null;
  }
```

```
  /**
   * 过滤器的类型 比如前置后置等
   * @return
   */
  @Override
  public String filterType() {
    return "pre";
  }

  /**
   * 执行顺序,越小级别越高
   * @return
   */
  @Override
  public int filterOrder() {
    return 1;
  }

}
```

8.10.2主程序

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

  @Bean
  public PreZuulFilter preZuulFilter() {//创建过滤器,也可以使用 Commpent 等注解创建
    return new PreZuulFilter();
  }
}
```

8.10.3 测试

> 1 启动 eureka,user,和当前 zuul
>
> 2 通过 zuul 代理访问 user 可以发现过滤器执行了

8.10.4 禁用 Filter

> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/multi/multi**router_and_filter_zuul.html#**literal_enablezuulserver_literal_filters

> Zuul for Spring Cloud comes with a number of `ZuulFilter` beans enabled by default in both proxy and server mode. See the zuul filters package for the possible filters that are enabled. If you want to disable one, simply set `zuul.<SimpleClassName>.<filterType>.disable=true`. By convention, the package after `filters` is the Zuul filter type. For example to disable `org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter` set `zuul.SendResponseFilter.post.disable=true`.
>
> SimpleClassName是简单类名,不是权限定名称
>
> 比如我们要禁用 A 过滤器的 pre 类型
>
> 则在 yml 中配置 zuul.A.pre.disable=true

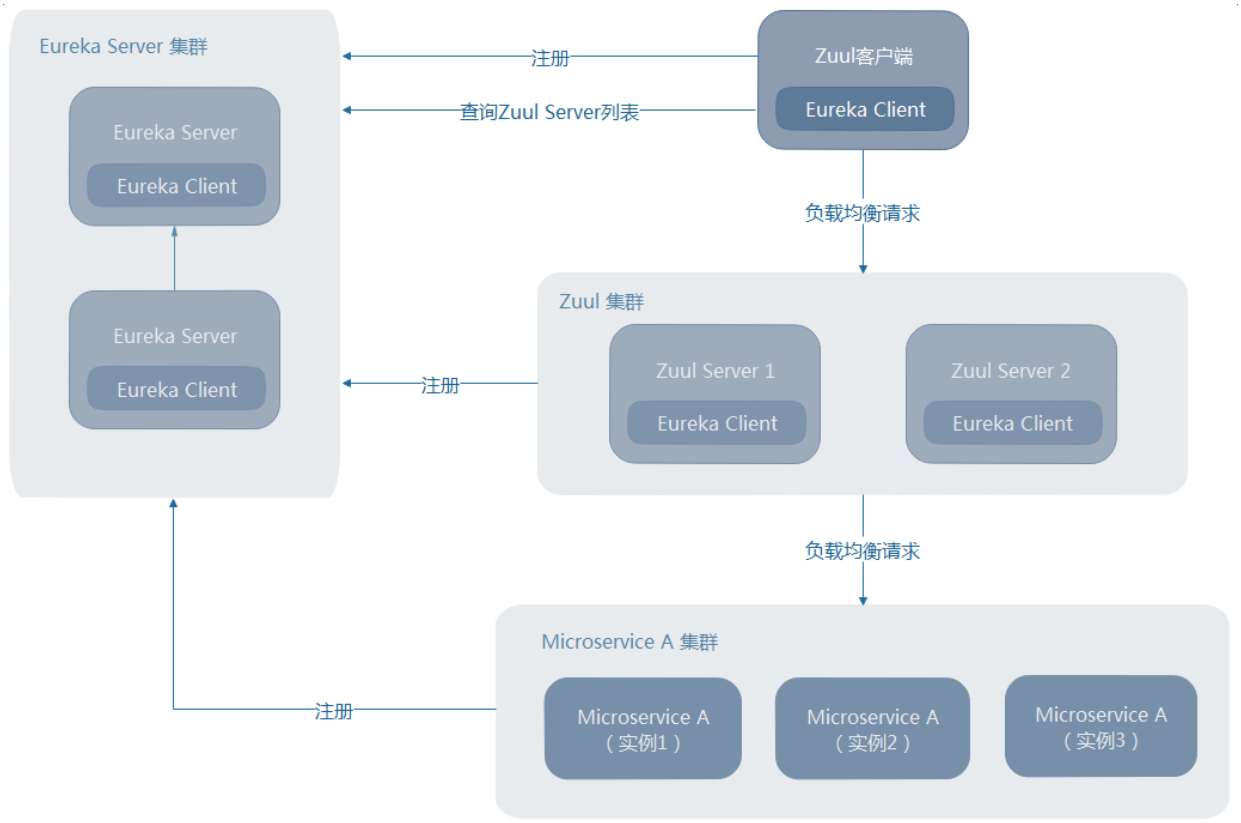
8.11 zuul 高可用

Zuul的高可用非常关键，因为外部请求到后端微服务的流量都会经过Zuul。故而在生产环境中，我们一般都需要部署高可用的Zuul以避免单点故障。

笔者分两种场景讨论Zuul的高可用。

8.11.1Zuul客户端也注册到了Eureka Server上

这种情况下，Zuul的高可用非常简单，只需将多个Zuul节点注册到Eureka Server上，就可实现Zuul的高可用。此时，Zuul的高可用与其他微服务的高可用没什么区别。

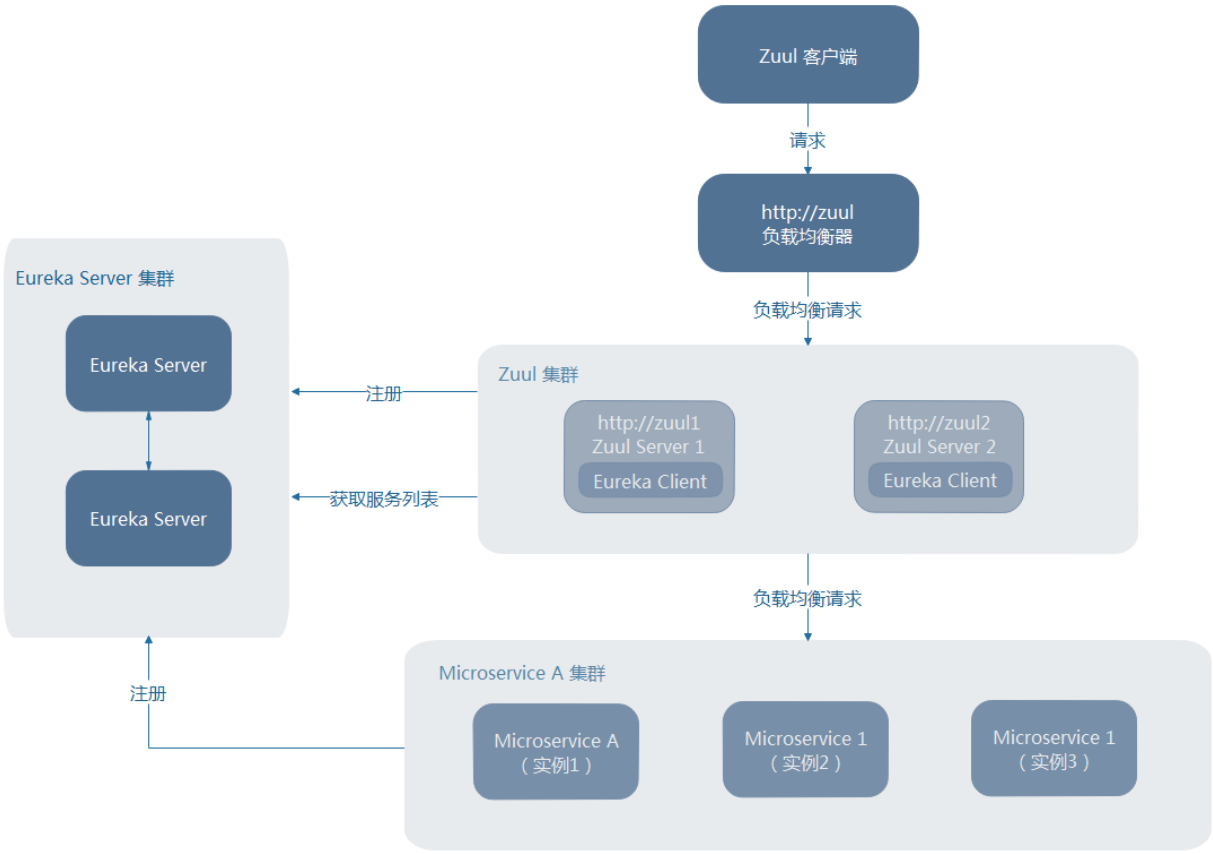


如图，当Zuul客户端也注册到Eureka Server上时，只需部署多个Zuul节点即可实现其高可用。Zuul客户端会自动从Eureka Server中查询Zuul Server的列表，并使用Ribbon负载均衡地请求Zuul集群。

这种场景一般用于**Sidecar**。

8.11.2Zuul客户端未注册到Eureka Server上

现实中，这种场景往往更常见，例如，Zuul客户端是一个手机APP——我们不可能让所有的手机终端都注册到Eureka Server上。这种情况下，我们可借助一个额外的负载均衡器来实现Zuul的高可用，例如Nginx、HAProxy、F5等。



Zuul高可用架构图

如图Zuul客户端将请求发送到负载均衡器，负载均衡器将请求转发到其代理的其中一个Zuul节点。这样，就可以实现Zuul的高可用。

九 SpringCloud Config

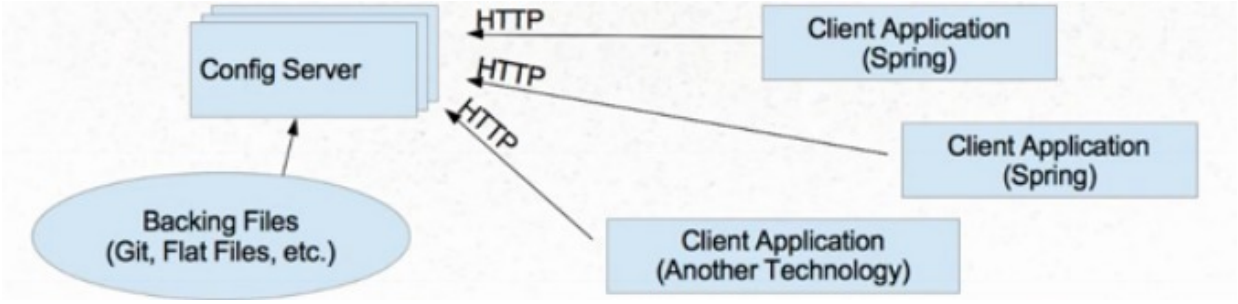
9.1 SpringCloud Config 简介

随着项目的变化,我们的微服务数量越来越多,配置也越来越多,也会遇到各种不同的环境,甚至修改了配置文件后需要自动刷新,等等需求,所以我们需要一个能帮我们集中管理配置的一个功能,这个就是springcloud 的 configserver 功能

一个Application中不只是代码,还需要连接资源和其它应用,经常有很多需要外部设置的项去调整Application行为,如切换不同的数据库,i18n国际化 等.应用中的会经常见到的xml,properties,yaml等就是配置信息.

常见的实现信息配置的方法:

- 硬编码(缺点:需要修改代码,风险大)
- 放在xml等配置文件中,和应用一起打包(缺点:需要重新打包和重启)
- 文件系统中(缺点:依赖操作系统等)
- 环境变量(缺点:有大量的配置需要人工设置到环境变量中,不便于管理,且依赖平台)
- 云端存储(缺点:与其他应用耦合) Spring Cloud Config 就是云端存储配置信息的,它具有中心化,版本控制,支持动态更新,平台独立,语言独立等特性.



Spring Cloud Config的原理如图所示,真正的数据存在Git等repository中,Config Server去获取相应的信息,然后开发给Client Application,相互间的通信基于HTTP,TCP,UDP等协议.它包含 ConfigServer 和 ConfigClient

ConfigServer 和 ConfigClient都实现了 Spring Environment 和 PropertySource抽象的映射,因此 SpringCloud Config非常适合 Spring 程序,当然也可以和其他语言整合

> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_spring_cloud_config

9.1.1 ConfigServer

Config Server 是一个可以横向扩展,集中式的配置服务器,它用于集中管理应用程序下的各种环境配置,默认使用 Git 存储配置内容,也可以使用 svn 本地文件等存储,因此可以方便的是思安对配置的版本控制和审计

9.1.2 Config Client

Config Client 是 Config Server 的客户端,用于操作在 Config Server 中的配置属性

9.4 Git Config Server 使用

9.4.1 创建 git 仓库

> 此处以码云做测试,实际开发请自行改变

9.4.2 基本使用模式

9.4.2.1 在仓库根目录提交一个配置文件

> 里面就这一个属性,我们做测试用,所以一个属性和多个属性没区别

```
profile: profile-default
```

9.4.2.2 创建一个项

9.4.2.2.1 pom

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

9.4.2.2.2 yml

```
server:
  port: 8080
spring:
  cloud:
    config:
      server:
        git:
          uri: https://gitee.com/jackiechan/spring-cloud-config
```

9.4.2.2.3 主程序

```
@SpringBootApplication
@EnableConfigServer//开启 ConfigServer 功能
public class ConfigServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServerApplication.class, args);
  }
}
```

9.4.2.2.4 使用配置文件

启动项目

> The HTTP service has resources in the form:
>
> 我们可以通过以下方式中一种方式去获取实用配置文件
>
> 默认情况下是通过查找文件名字的方式找文件,找不到则匹配默认的

```
/{application}/{profile}[/{label}]  application代表程序名称, profile 代表哪个文件
lable 代表版本,比如 git 是 master
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

此处我们通过http://localhost:8080/abc-default.properties 或者http://localhost:8080/abc-default.yml 可以访问到我们刚才提交的文件内容,其中 abc 是任意内容

9.4.2.3 其他匹配

1.新创建 foobar-dev.yml

```
profile: foo-dev
```

启动 configserver 访问http://localhost:8080/foobar-dev.properties 或者http://localhost:8080/foobar-dev.yml 可以查看到内容,其他地址发现输出默认文件的信息

2. 访问http://localhost:8080/abc/default/master 发现也可以访问并且获取到内容,注意 name 属性的值只是一个描述值,并不是文件的实际访问路径

9.5 ConfigClient

> 在微服务中使用我们的 configserver
>
> 参考microservice-config-client
>
> microservice-config-server

9.5.1 client pom

```
<dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>
```

9.5.2 application.yml

```
server:
  port: 8081
  #profile: dev 如果当前文件和 bootstrap 里面的重复,则以 bootstrap 为准
```

9.5.3 bootstrap.yml

> 注意连接 configserver 的配置必须写在bootstrap.yml

```yaml
spring:
  cloud:
    config:
      uri: http://localhost:8080
      profile: dev
      label: master   # 当configserver的后端存储是Git时，默认就是master
  application: #此属性可以放到 application 中
    name: foobar  #程序 name，参考路径的匹配规则,会找 foobar 开头的叫 dev 的文件
```

9.5.4 controller

```
@RestController
public class ConfigClientController {

  @Value("${profile}")//注意我们的配置文件中并没有这个属性,我们的目的是通过连接配置中心获取当前数据的值
  private String profile;

  @GetMapping("/profile")
  public String getProfile() {//此接口无意义,仅用于测试
    return this.profile;
  }
}
```

9.5.5 主程序

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

9.5.6 启动测试

> 启动 server 和 client 访问http://localhost:8081/profile 可以发现获取到了 foobar 的属性

> It is a good idea to set the `spring.application.name` (in `bootstrap.yml` or `application.yml`) if your application needs any application-specific configuration from the server.
>
> 建议最好是在 bootstrap 或者 application.yml文件中添加spring.application.name
>
> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_the_bootstrap_application_context

9.4.3 匹配方式2通配符

> 在 git 上面新建两个项目 一个是 simple 一个是special ,在里面分别放置 application.yml
>
> 通配符实现一个项目一个配置

9.4.3.1 修改 yml

```yaml
server:
  port: 8080
spring:
  cloud:
    config:
      server:
        git:
          uri: https://gitee.com/jackiechan/{application}
```

9.4.3.2 启动测试

> 按照映射方式(只要符合映射模式即可)访问http://localhost:8080/simple-dev.properties 或者http://localhost:8080/special-dev.properties 可以分别获取到对应的内容, 可以做到不同项目不同配置,以免修改的时候影响其他项目

9.4.4 匹配方式3模式匹配

> 按照不同的模式映射不同的配置地址
>
> 在special 目录中新建special-dev.yml 和special-test.yml

9.4.4.1 修改 yml

```yaml
server:
  port: 8080
spring:
  cloud:
    config:
      server:
        git:
          uri: https://gitee.com/jackiechan/spring-cloud-config     # 公用,如果其他的无法匹配到,则使用这个默认的
          repos: #指定具体项目配置
             simple:
              uri: https://gitee.com/jackiechan/simple #simple 配置这个地址
            special: #special的下面地址匹配
              pattern: special*/dev*,special*/test* #special-dev 匹配 special-dev 文件, special-sfsdf 匹配  special 只匹配符合当前格式的,不符合的会走上面公用的
              uri: https://gitee.com/jackiechan/special
```

9.4.4.2 访问测试

> 访问http://localhost:8080/master/special-dev.properties发现输出 dev 信息 http://localhost:8080/master/special-test.properties 输出 test 信息 http://localhost:8080/master/special-default.properties 匹配默认的

9.4.3 匹配方式3 搜索匹配

> 在spring-cloud-config仓库中新建 foo 和 bar 目录
>
> 在内部分别创建 foo-dev.yml 和 bar-dev.yml

9.4.3.1 yml

```yaml
server:
  port: 8080
spring:
  cloud:
    config:
      server:
        git:
          uri: https://gitee.com/jackiechan/spring-cloud-config      # 公用
          search-paths:
            - foo   # foo路径,以 foo 开头的请求会被转到 foo文件夹中
            - bar   # bar路径
```

9.4.3.2 访问测试

> 访问http://localhost:8080/master/bar-dev.properties 和http://localhost:8080/master/foo-dev.properties
>
> 可以访问对应文件

9.4.4 启动加载文件

> 上面我们的配置文件是在访问的时候才会去加载,我们可以通过设置来开启

```yaml
server:
  port: 8080
spring:
  cloud:
    config:
      server:
        git:
          uri: https://gitee.com/jackiechan/spring-cloud-config      # 公用
          clone-on-start: true #启动加载 ,全局配置
          repos:
            simple:
              uri: https://gitee.com/jackiechan/simple
            special:
              pattern: special*/dev*,special*/test*
              uri: https://gitee.com/jackiechan/special
              cloneOnStart: false #关闭启动加载 默认是 false, 这是局部配置
```

9.4.5 仓库密码

> 如果仓库访问需要密码的话可以设置密码

```yaml
server:
  port: 8080
spring:
  cloud:
    config:
      server:
        git:
          uri: https://gitee.com/jackiechan/spring-cloud-config
          username: #没有密码可以不写
          password:
```

9.5 Config Server 加密

> 加密的原因是方式他人看到请求的数据信息,比如配置文件中存放的是一些敏感信息等,所以需要加密
>
> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_encryption_and_decryption
>
> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_encryption_and_decryption_2
>
> 首先下载JCE
>
> 参考microservice-config-server-encrypt

9.5.1 对称加密

9.5.1.1 yml

```yaml
server:
  port: 8080
spring:
  cloud:
    config:
      server:
        git:
          uri: https://gitee.com/jackiechan/spring-cloud-config
          username:
          password:

encrypt:
  key: foo #密码
```

9.5.1.2 主程序

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

9.5.1.3 启动测试

> curl -X POST http://localhost:8080/encrypt -d foobar 请求我们的配置文件,发现返回的是密文
>
> curl -X POST http://localhost:8080/decrypt -d foobar

9.5.2 注意点

> 正常来说,我们的服务端应该配置的是密文
>
> 密文配合 方式:
>
> ```
> xxxx :'{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'  #在 yml 中给 xxxx属性设置密码 格式为'{cipher}内容'
> ```
>
> ```
> xxxx :{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ  #在 properties 中没有两侧的''
> ```

9.5.3 自动解密

> 前面我们是通过 curl 的方式访问获取数据的,实际上我们的 configserver 会自动帮我们解密,我们只需要设置密码即可,也就是我们的 config-client 也不需要做改动,直接使用即可
>
> 在spring-cloud-config-test 仓库中添加两个文件foobar-production.yml 和foobar-test.properties 分别放入加密的内容
>
> 启动microservice-config-server-encrypt
>
> 访问http://localhost:8080/master/foobar-test.properties 和http://localhost:8080/master/foobar-production.yml发现可以自动解密

9.5.4 非对称加密

> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_creating_a_key_store_for_testing
>
> 非对称加密使用证书,我们第一步要生成证书,然后倒入证书即可

> 参考microservice-config-server-encrypt-rsa

9.5.4.1 yml

```yaml
server:
  port: 8080
spring:
  cloud:
    config:
      server:
        git:
          uri: https://gitee.com/jackiechan/spring-cloud-config
          username:
          password:

encrypt:
  keyStore:
    location: classpath:/server.jks  #指定 jsk 的位置
    password: letmein  #key 的密码
    alias: mytestkey #key 的别名
    secret: changeme #别名的密码
```

9.5.4.2 其他配置同对称加密

> 配置中新内使用的是加密后的内容,然后通过它解密即可

9.6 Config Server 认证

> 参考microservice-config-server-authc
>
> microservice-config-client-authc

9.6.1 server 配置

9.6.1.1 serverpom

```xml
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-config-server</artifactId>
        </dependency>

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

9.6.1.2 server yml

```yaml
security:
  basic:
    enabled: true #开启认证
  user: #配置访问当前配置中心的密码
    name: user
    password: password123
server:
  port: 8080
spring:
  cloud:
    config:
      server:
        git:
          uri: https://gitee.com/jackiechan/spring-cloud-config
          username:
          password:
```

9.6.1.3 server主程序

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

9.6.1.4 启动 server 测试

> 访问 http://localhost:8080/master/simple-default.properties 发现需要密码

9.6.2 client 配置

9.6.2.1 client pom

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>
```

9.6.2.2 client bottstrap.yml

> 注意配置密码需要放在bottstrap.yml

```
spring:
  cloud:
    config:
      uri: http://localhost:8080  # curl style
      username: user
      password: password123
      profile: dev
      label: master   # 当configserver的后端存储是Git时，默认就是master
  application:
    name: foobar #此处 foobar 是因为我们的配置中心的文件 foobar 的原因
```

9.6.2.3 client application.yml

```
server:
  port: 8081
#此处没有别的功能所以不需要其他东西
```

9.6.2.4 controller

```
@RestController
public class ConfigClientController {

  @Value("${profile}")
  private String profile;

  @GetMapping("/profile")
  public String getProfile() {
    return this.profile;
  }
}
```

9.6.2.5 主程序

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

9.6.2.6 启动测试

> 启动 configserver 和 client,, 访问 client 的 controller 发现可以访问了,

9.7 Config Server 整合 Eureka

> 因为实际开发中,我们可能会有很多个 configserver, 所以我们需要解决负载均衡的问题,我们需要在客户端配置多个 configserver, 这样问题就都跑到了客户端,如果我们的服务端发生了改变,那么客户端都得改变,会导致不灵活,因此 我们希望如果 configserver 也能注册到 eureka 上,自动发现就好了
>
> 参考microservice-config-server-eureka
>
> microservice-discovery-eureka
>
> microservice-config-client-eureka

9.7.1 server 端配置

9.7.1.1 server pom

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

9.7.1.2 yml

```
server:
  port: 8080
spring:
  cloud:
    config:
```

```
      server:
        git:
          uri: https://gitee.com/jackiechan/spring-cloud-config
          username:
          password:
  application:
    name: microservice-config-server-eureka

eureka:
  client:
    serviceUrl:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
```

9.7.1.3 主程序

```
@SpringBootApplication
@EnableConfigServer// configserver
@EnableDiscoveryClient//客户端注册
public class ConfigServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServerApplication.class, args);
  }
}
```

通过上面我们发现其实 server 上面只需要添加注册到 eureka 就行

9.7.1.4 启动测试

> 启动 eureka ,我们的 server,然后访问 eureka 发现可以找到我们的 server

9.7.2 client 配置

9.7.2.1 client pom

```
<dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
```

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

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

9.7.2.2 client yml

> bootstrap.yml

```
spring:
  cloud:
    config:
      discovery:
        enabled: true
        service-id: microservice-config-server-eureka
     # username: user # 如果configserver 需要认证的话,在这里配置
     # password: password123 #
  application:
    name: foobar

eureka:
  client:
    serviceUrl:
      defaultZone: http://user:password123@localhost:8761/eureka
  instance:
    prefer-ip-address: true
```

> application.yml

```
server:
  port: 8081
```

9.7.2.3 client controller

```
@RestController
public class ConfigClientController {

  @Value("${profile}")
  private String profile;

  @GetMapping("/profile")
  public String getProfile() {
    return this.profile;
  }
}
```

9.7.2.4 client 主程序

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

9.7.2.5 启动测试

> 启动我们的 client,访问我们的 controller 接口,发现可以从 configserver 中获取数据

9.8 手动刷新Config Server 配置

> 当我们的 git 中的数据变化后,希望能刷新到 configserver 中,而不需要我们重启项目
>
> 刷新数据不需要我们变化我们的 configserver 代码,我们只需要修改 client 即可
>
> 参考
>
> microservice-config-client-refresh

9.8.1 pom

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
```

```
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

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

9.8.2 yml 配置

> application.yml 同上 server 配置
>
> Bootstrap.yml

```
spring:
  cloud:
    config:
      #username: user
      #password: password123
      uri: http://localhost:8080
      profile: dev
      label: master   # 当configserver的后端存储是Git时，默认就是master
  application:
    name: foobar
#management:
 # security:
  #  enabled: false
security:
  user:
    name: user
    password: password123
```

9.8.3 controller

> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_refresh_scope

> A Spring `@Bean` that is marked as `@RefreshScope` will get special treatment when there is a configuration change. This addresses the problem of stateful beans that only get their configuration injected when they are initialized. For instance if a `DataSource` has open connections when the database URL is changed via the `Environment`, we probably want the holders of those connections to be able to complete what they are doing. Then the next time someone borrows a connection from the pool he gets one with the new URL.
>
> 被RefreshScope修饰的会在数据发生变化 变化,但是它会在下次请求的时候发生变化,而不是数据一旦变化后自动变化

```
@RestController
@RefreshScope//添加此注解,开启自动刷新
public class ConfigClientController {

  @Value("${profile}")
  private String profile;

  @GetMapping("/profile")
  public String getProfile() {
    return this.profile;
  }
}
```

9.8.4 主程序

> 同上面 serever配置

9.8.5启动测试

> 访问http://192.168.3.84:8081/profile 可以获取到数据
>
> 但是更新 git 中数据后无法获取
>
> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#_endpoints 解释/ refresh
>
> 通过 post 方式 请求 http://user:password123@localhost:8081/refresh (curl -X POST http://user:password123@localhost:8081/refresh)后,再重新请求http://192.168.3.84:8081/profile

9.9 自动刷新Config Server 配置

> Spring cloud 支持使用 spring cloud bus 来通过消息推送方式来自动刷新配置
>
> 此处使用的是 amqp 的 rabbitmq
>
> 同上 config server 是不需要改变代码的
>
> 参考microservice-config-client-refresh-bus

此方式需要提前安装 rabbitmq

9.9.1 client pom

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

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

9.9.2 yml

> application.yml 同上client
>
> bootstrap.yml

```
spring:
  cloud:
    config:
      uri: http://localhost:8080
      profile: dev
      label: master   # 当configserver的后端存储是Git时，默认就是master
    bus:
      trace:
        enabled: true
  application:
    name: foobar
  rabbitmq: #rabbit 的配置
    host: localhost  #主机
    port: 5672 #端口
```

```
    username: guest
    password: guest
  security:
   user:
    name: user
    password: password123
```

9.9.3 controller 和主程序同上client

9.9.4 启动我们的 configserver 和 client(修改端口,启动两次,好观察效果)

> 访问我们的 http://localhost:8082/profile 可以获取到数据
>
> 更新我们的 git 数据,然后 post 访问 http://localhost:8081/bus/refresh (curl -X POST http://user:password123@localhost:8081/bus/refresh) 调用 rabbitmq 的消息刷新

9.9.5 配置自动刷新

> 配置我们的 git 仓库,此配置的作用是当我们的 git 仓库改变的时候吗,向指定地址发起 post 请求,注意此处需要使用公网可以访问的地址,以后我们提交数据后就会自动刷新了

9.10 自动重试

> client 连接到 configserver 失败后如何重试呢
>
> http://cloud.spring.io/spring-cloud-static/Edgware.SR2/single/spring-cloud.html#config-client-retry
>
> If you expect that the config server may occasionally be unavailable when your app starts, you can ask it to keep trying after a failure. First you need to set `spring.cloud.config.failFast=true`, and then you need to add `spring-retry` and `spring-boot-starter-aop` to your classpath. The default behaviour is to retry 6 times with an initial backoff interval of 1000ms and an exponential multiplier of 1.1 for subsequent backoffs. You can configure these properties (and others) using

> `spring.cloud.config.retry.*` configuration properties.
>
> 要添加`spring-retry` and `spring-boot-starter-aop`的依赖,然后配置 spring.cloud.config.failFast=true

9.11 刷新扩展

> 参考http://www.itmuch.com/spring-cloud/spring-cloud-bus-auto-refresh-configuration

9.11 Config Server 高可用

> 如果使用 github 等服务商 他们已经自带高可用,如果是自己搭建的环境,需要自己解决高可用,此处不属于springcloud 的范畴

10 错误

10.1 连接 Eureka 超时

> 参考github 上面 spring-cloud- netflix 的768号 issues,因为hystrix 默认超时时间是1秒
>
> 解决方式 在 client 的 yml 中设置

```
# 解决第一次请求报超时异常的方案：设置超时时间长点
# hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 5000
# 或者禁用超时设置
# hystrix.command.default.execution.timeout.enabled: false
# 或者：
feign.hystrix.enabled: false ## 索性禁用feign的hystrix支持

# 超时的issue：https://github.com/spring-cloud/spring-cloud-netflix/issues/768
# 超时的解决方案： http://stackoverflow.com/questions/27375557/hystrix-command-fails-with-timed-out-and-no-fallback-available
# hystrix配置： https
```

10.2 RESTTemplate 获取 List 类型数据

> 参考microservice-consumer-movie-ribbon-test
>
> microservice-provider-user
>
> microservice-discovery-eureka

10.2.1 movie controller

> 添加以下方法测试

```
@GetMapping("/list-all")
public List<User> listAll() {
```

```
    // wrong
    //    List<User> list = this.restTemplate.getForObject("http://microservice-provider-user/list-all", List.class);
    //    for (User user : list) {
    //      System.out.println(user.getId());
    //    }

    // right
    User[] users = this.restTemplate.getForObject("http://microservice-provider-user/list-all", User[].class);//此处应该使用数组
    List<User> list2 = Arrays.asList(users);
    for (User user : list2) {
      System.out.println(user.getId());
    }

    return list2;
  }
```

10.2.2 user controller

```
/**
   * 获取多条数据
   * @return
   */
  @GetMapping("list-all")
  public List<User> listAll() {
    ArrayList<User> list = Lists.newArrayList();
    User user = new User(1L, "zhangsan");
    User user2 = new User(2L, "zhangsan");
    User user3 = new User(3L, "zhangsan");
    list.add(user);
    list.add(user2);
    list.add(user3);
    return list;

  }
```

10.3 Eureka Server不踢出已关停的节点的问题

```
#server端:
eureka.server.enable-self-preservation          # (设为false，关闭自我保护主要)
eureka.server.eviction-interval-timer-in-ms     #清理间隔（单位毫秒，默认是60*1000)
#client端:
eureka.client.healthcheck.enabled = true                #开启健康检查（需要spring-boot-starter-actuator依赖)
eureka.instance.lease-renewal-interval-in-seconds =10       #租期更新时间间隔（默认30秒)
eureka.instance.lease-expiration-duration-in-seconds =30  #租期到期时间（默认90秒)
```

11注意

11.1新特性

Spring Cloud发布Spring Cloud Edgware 版本。该版本解决了不少Bug，而且Edgware版本的出现一个重大改变—Edgware重命名了很多Starter的Artifact ID。虽然在Edgware版中，原Artifact ID依然可用，但一旦Spring Cloud Finchley 发布，老的Artifact ID将会废弃!

以下是Spring Cloud Edgware及之前版本中Starter改名前后的映射表:

```
spring-cloud-starter-eureka-server –> spring-cloud-starter-netflix-eureka-server
spring-cloud-starter-eureka –> spring-cloud-starter-netflix-eureka-client
spring-cloud-starter-ribbon –> spring-cloud-starter-netflix-ribbon
spring-cloud-starter-hystrix –>spring-cloud-starter-netflix-hystrix
spring-cloud-starter-hystrix-dashboard –> spring-cloud-starter-netflix-hystrix-dashboard
spring-cloud-starter-turbine –> spring-cloud-starter-netflix-turbine
spring-cloud-starter-turbine-stream –> spring-cloud-starter-netflix-turbine-stream
spring-cloud-starter-feign –> spring-cloud-starter-openfeign
spring-cloud-starter-zuul –> spring-cloud-starter-netflix-zuul
```