

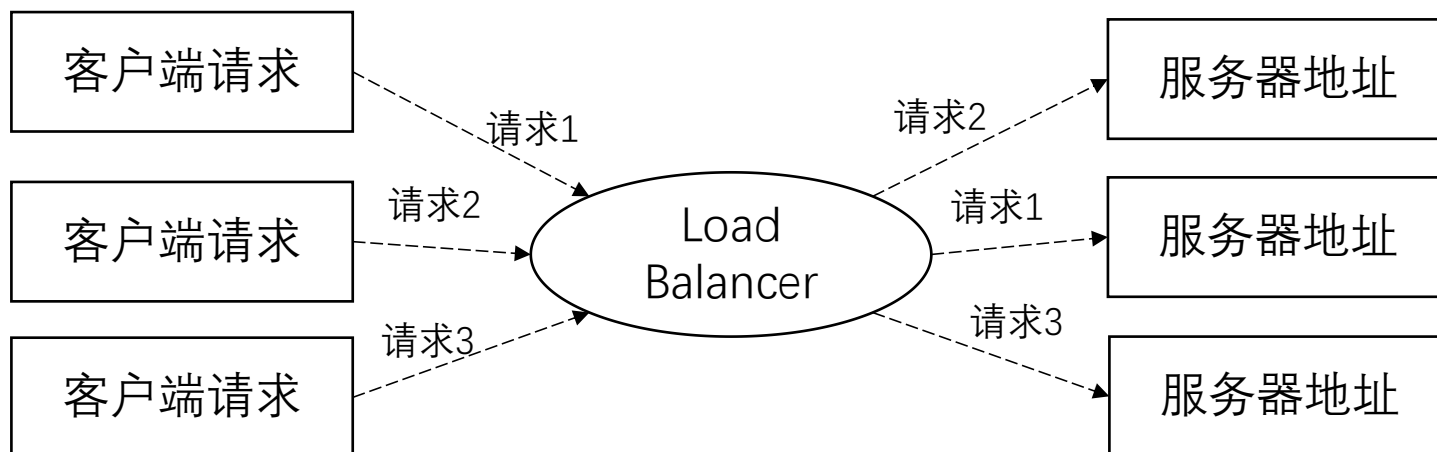
第九章 Spring Cloud服务调用 与负载均衡

负载均衡

- 集群的负载均衡机制可以将业务请求分摊到多台单机性能不一定非常出众的服务器
- 集群的容错机制确保当前集群中的某台机器无法正常提供服务时整个集群仍然可用

负载均衡

负载为了在服务的多个实例之间分配负载，将请求分摊到多个操作单元上进行执行，以达到更优化资源使用、最大化吞吐率，同时避免过载



负载均衡示意图

负载均衡

- 负载均衡功能由负载均衡器完成，负载均衡器通常需要与服务注册中心共同完成服务发现功能，以及时获取服务实例的最新消息
- 根据服务器地址列表所存放的位置分为两大类：
 - 服务器负载均衡
 - 客户端负载均衡

负载均衡

- 服务器端负载均衡是指负载均衡器位于服务端，也称作集中式负载均衡

客户端发送请求到负载均衡器，负载均衡器负责将接收到的各个请求转送到运行中的某台微服务节点上，然后接收到请求的微服务做响应处理

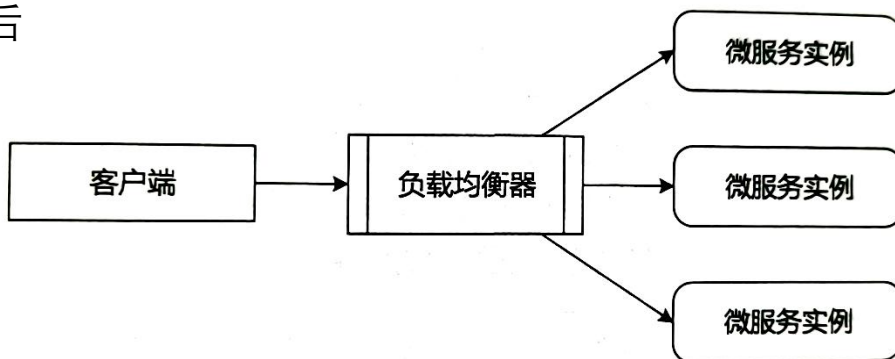


图 4-15 服务器端负载均衡示意图

负载均衡

- 常见的服务端负载均衡器有：HAProxy、Nginx和LVS

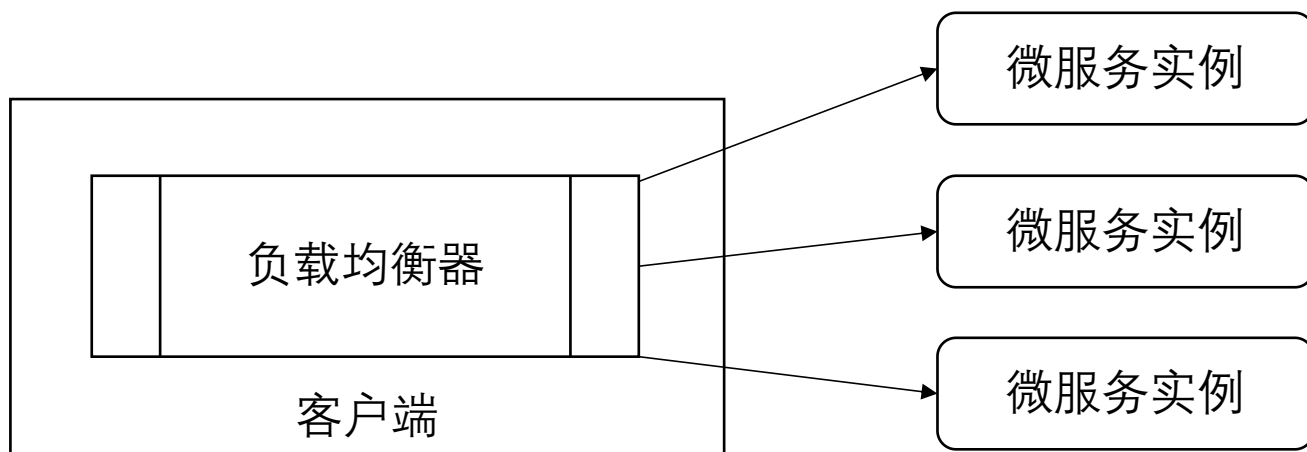
名称	协议支持	Session保持
HAProxy	网络4(TCP)、7层(HTTP)协议	支持
Nginx	网络4(TCP)、7层(HTTP)协议	支持
LVS	网络4层协议	支持

负载均衡

- 基于服务端的负载均衡机制优点：
 - 实现简单：只需要在客户端与各个微服务实例之间架设集中式的负载均衡器
 - 负载均衡器检测到某个服务已经不可用时就会自动移除该服务
 - 负载均衡对于服务消费者来说是透明的，服务消费者完全感知不到负载均衡的存在
- 基于服务端的负载均衡机制弊端：
 - 负载均衡器将会成为系统的瓶颈
 - 一旦负载均衡器自身发生失败，整个微服务的调用过程都将发生失败

负载均衡

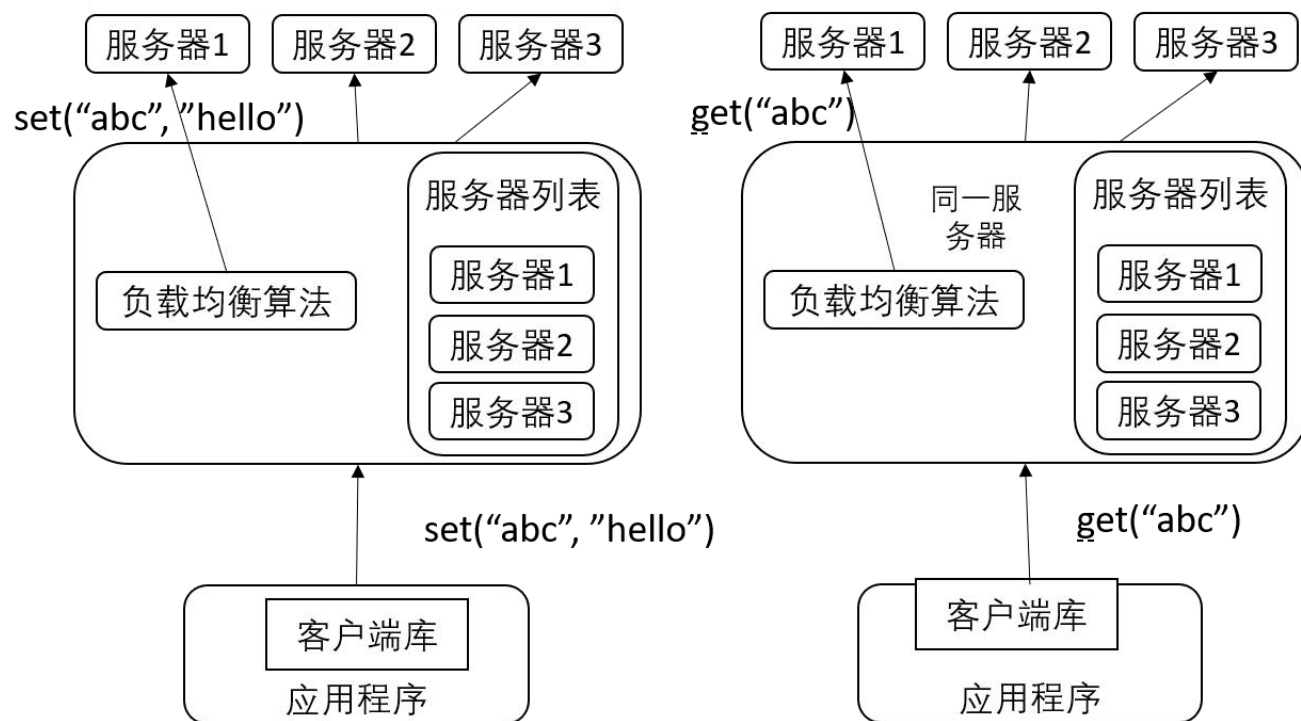
- 客户端负载均衡：将负载均衡的功能以库的形式集成到服务消费中，在客户端程序里面自己设定一个调度算法，在向服务器发起请求的时候，先执行调度算法计算出目标服务器地址



客户端负载均衡示意图

负载均衡

- 客户端负载均衡流程：在微服务中包含着各个服务器的配置信息，然后通过负载均衡算法计算目标服务器实现负载均衡



负载均衡

- 典型的客户端负载均衡机制应用：
 - Memcache缓存：使用Memcache的应用程序会依赖客户端库中所包含的分布式算法获取目标Memcache服务地址
 - Nginx等能够实现代理功能的负载均衡器部署到运行微服务的同一台机器上。

负载均衡

- 基于客户端的负载均衡的主要优势：
 - 不会出现集中式负载均衡所产生的瓶颈问题
 - 负载均衡器的失败也不会造成严重的后果
- 基于客户端的负载均衡的弊端：
 - 由于所有的运行时信息都需要在多个负载均衡器之间进行服务配置信息的传递，会在一定程度上加重网络流量负载

负载均衡策略

- 负载均衡器需要采用一种或多种负载均衡策略完成负载分配，以达到资源最优惠的使用
- 常见的负载均衡策略：
 - 随机策略：负载均衡器将请求随机分配到服务的实例上
 - 轮询策略：负载均衡器将请求按顺序逐一分配到不同的实例上
 - 权重策略：负载均衡器为各实例设置权重，各实例的访问比率约等于权重比例
 - 响应时间策略：负载均衡器按实例的响应时间来分配请求，响应时间短的优先分配

负载均衡策略

- 轮询策略：

- 每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器 *down* 掉，能自动剔除

```
upstream backserver {  
    server 192.168.0.14;  
    server 192.168.0.15;  
}
```

- 权重策略：

- 指定轮询几率，*weight*和访问比率成正比，用于后端服务器性能不均的情况。
- 权重越高，在被访问的概率越大，如例，分别是30%，70%。

```
upstream backserver {  
    server 192.168.0.14 weight=3;  
    server 192.168.0.15 weight=7;  
}
```

负载均衡策略

- 响应时间策略：

- 按后端服务器的响应时间来分配请求，响应时间短的优先分配。
- *fair*根据响应时间返回服务器。

```
upstream backserver {  
    server server1;  
    server server2;  
    fair;  
}
```

负载均衡策略

- 另一类负载均衡策略基于会话保持，不过微服务的设计通常是无状态的，因此在实际场景中不常使用：
 - 源地址hash：按请求访问的IP的hash结果进行分配，这样每个访客会被路由到特定实例上
 - Cookie识别：在返回结果中插入特定的字符串
 - 基于session：在负载均衡器中存储session信息



Spring Cloud实现服务调用与负载均衡的方式

1. RestTemplate（服务调用）+Ribbon（负载均衡）
2. OpenFeign（服务调用）+ 内置Ribbon（负载均衡）

推荐使用第二种方式，简单易用！



负载均衡组件-Ribbon

问题：实际环境中，我们往往会开启很多个用户微服务的集群。此时我们获取的服务列表中就会有多个，到底购票微服务该访问哪一个呢？

```
@RequestMapping(value = "/order", method = RequestMethod.GET)
public String order(){
    // 模拟当前用户
    Integer id = 2;

    // 到Eureka发现用户微服务
    // 返回值为什么是List集合呢？ 获取到所有同名的微服务
    List<ServiceInstance> instances = discoveryClient.getInstances( servicelId: "myshop-user");
    // 没有使用负载均衡，只能获取第一个服务
    ServiceInstance serviceInstance = instances.get(0);
    User user = restTemplate.getForObject( url: "http://" + serviceInstance.getHost() + ":" +
        +serviceInstance.getPort() + "/user/" + id, User.class);
    System.out.println(user + "正在购票...");
    return "购票成功";
}
```



开启两个用户微服务，用户微服务1的controller代码

```
/** 根据id查询用户 */  
@RequestMapping(value = "{id}", method = RequestMethod.GET)  
public User findById(@PathVariable Integer id){  
    System.out.println("用户微服务11111");  
    return userService.findById(id);  
}
```

用户微服务2的controller代码

```
public User findById(@PathVariable Integer id){  
    System.out.println("用户微服务22222");  
    return userService.findById(id);  
}
```

用户微服务1的application.yml

```
server:  
  port: 9001
```

用户微服务2的application.yml

```
server:  
  port: 9101
```

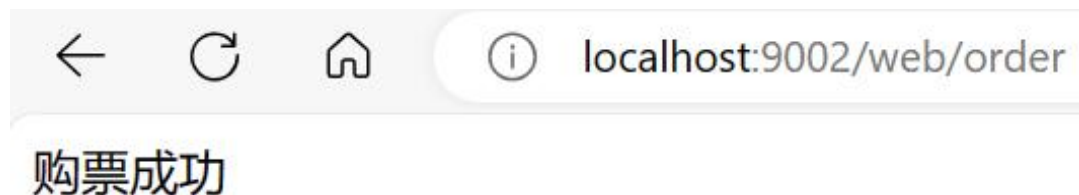


开启两个用户微服务，并在Eureka上查看服务状态

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EUREKA-SERVER	n/a (1)	(1)	UP (1) - TF-laptop:eureka-server:9999
MYSHOP-USER	n/a (2)	(2)	UP (2) - TF-laptop:myshop-user:9001 , TF-laptop:myshop-user:9101
MYSHOP-WEB	n/a (1)	(1)	UP (1) - TF-laptop:myshop-web:9002

多次请求购票微服务



用户微服务1的后台控制台显示

控制台 Actuator

```
2023-10-13 20:30:08.690 INFO 15308 ---
用户微服务11111
Hibernate: select user0_.id as id1_0_0_,
用户微服务11111
Hibernate: select user0_.id as id1_0_0_,
用户微服务11111
Hibernate: select user0_.id as id1_0_0_,
用户微服务11111
Hibernate: select user0_.id as id1_0_0_,
```

用户微服务2的控制台显示


控制台 Actuator

```
2023-10-13 20:32:15.018 INFO
```


负载均衡组件-Ribbon

问题：实际环境中，我们往往会开启很多个用户微服务的集群。此时我们获取的服务列表中就会有多个，到底电影微服务该访问哪一个呢？

答案：Eureka中已经帮我们集成了负载均衡组件：**Ribbon**，使用起来非常简单。


 myshop-web


>  生命周期

>  插件

√  依赖项

√  org.springframework.cloud:spring-cloud-starter-netflix-eureka-client:2.0.0.M8

>  org.springframework.cloud:spring-cloud-starter:2.0.0.M9


>  org.springframework.cloud:spring-cloud-netflix-core:2.0.0.M8


 org.springframework.cloud:spring-cloud-netflix-eureka-client:2.0.0.M8

>  com.netflix.eureka:eureka-client:1.8.7

>  com.netflix.eureka:eureka-core:1.8.7


>  org.springframework.cloud:spring-cloud-starter-netflix-archaius:2.0.0.M8

>  org.springframework.cloud:spring-cloud-starter-netflix-ribbon:2.0.0.M8

>  com.netflix.ribbon:ribbon-eureka:2.2.5

>  com.thoughtworks.xstream:xstream:1.4.10

>  org.springframework.cloud:spring-cloud-netflix-eureka-client:2.0.0.M8

>  org.springframework.boot:spring-boot-starter-web:2.0.1.RELEASE



负载均衡组件-Ribbon

Ribbon是Netflix 发布的负载均衡器，它有助于控制HTTP和TCP客户端的行为。为Ribbon配置服务提供者地址列表后，Ribbon就可基于某种负载均衡算法，自动地帮助服务消费者去请求。Ribbon默认为我们提供了很多的负载均衡算法，例如轮询、随机、哈希等。当然,我们也可为Ribbon实现自定义的负载均衡算法。



Ribbon实现用户微服务负载均衡

- 修改购票微服务controller代码
 1. 注入负载均衡客户端LoadBalancerClient
 2. 使用Ribbon的choose方法选择合适的服务

```
@Autowired
private LoadBalancerClient client;

/**
 * 购票方法(使用Ribbon负载均衡组件)
 * @return
 */
@RequestMapping(value = "/order", method = RequestMethod.GET)
public String order(){
    // 模拟当前用户
    Integer id = 2;
    // 使用Ribbon帮助我们选择一个合适的服务实例 (默认算法: 轮询)
    ServiceInstance serviceInstance= client.choose( serviceId: "myshop-user");
    User user = restTemplate.getForObject( url: "http://" + serviceInstance.getHost() + ":"
        + serviceInstance.getPort() + "/user/" + id, User.class);
    System.out.println(user+"==正在购票...");
    return "购票成功";
}
```



重启购票微服务，并6次请求该服务



购票成功

用户微服务1的后台控制台显示

```
用户微服务11111  
Hibernate: select user0_.id as :  
用户微服务11111  
Hibernate: select user0_.id as :  
用户微服务11111  
Hibernate: select user0_.id as :
```

用户微服务2的控制台显示

```
用户微服务22222  
Hibernate: select user0_  
用户微服务22222  
Hibernate: select user0_  
用户微服务22222  
Hibernate: select user0_
```

Ribbon默认使用轮询算法！！！！



• 再简化购票微服务

1. 在启动类的RestTemplate实例方法上添加@LoadBalance注解

```
@SpringBootApplication
@EnableEurekaClient // 开启Eureka客户端自动配置
public class WebApplication {
    public static void main(String[] args) { SpringApplication.run(WebApplication.class,
        /** 实例化RestTemplate */
        @Bean
        @LoadBalanced // 添加Ribbon负载均衡组件
        public RestTemplate restTemplate() { return new RestTemplate(); }
    }
```

2. 修改购票方法controller代码

```
@RequestMapping(value = "/order", method = RequestMethod.GET)
public String order(){
    // 模拟当前用户
    Integer id = 2;
    // 使用Ribbon帮助我们选择一个合适的服务实例（默认算法：轮询）
    //ServiceInstance serviceInstance= client.choose("myshop-user");
    User user = restTemplate.getForObject( url: "http://myshop-user/user/"+id, User.class)
    System.out.println(user+"==正在购票...");
    return "购票成功";
}
```




LoadBalancerInterceptor负载均衡拦截器

```
public ClientHttpResponse intercept(final HttpRequest request, final byte[] body,  
    URI originalUri = request.getURI(); originalUri: "http://myshop-user/user/2"  
    String serviceName = originalUri.getHost(); serviceName: "myshop-user"  
    Assert.state(expression: serviceName != null, message: "Request URI does not con  
    return (ClientHttpResponse) this.loadBalancer.execute(serviceName, this.request  
}
```


> {RibbonLoadBalancerClient@8182}



execute方法

```
public <T> T execute(String serviceId, LoadBalancerRequest<T> req  
    ILoadBalancer loadBalancer = this.getLoadBalancer(serviceId);  
    Server server = this.getServer(loadBalancer); loadBalancer:  
    if (server == null) { server: "192.168.31.202:9001"  
        throw new IllegalStateException("No instances available f  
    } else {  
        RibbonServer ribbonServer = new RibbonServer(serviceId, s  
        return this.execute(serviceId, ribbonServer, request);
```

开启两个用户微服务，分别是9001和9101，execute方法中设置断点，并请求购票微服务



```
public <T> T execute(String serviceId, LoadBalancerRequest<T> request)
    ILoadBalancer loadBalancer = this.getLoadBalancer(serviceId);
    Server server = this.getServer(loadBalancer); server: "192.168.31.202:9101"
    if (server == null) { server: "192.168.31.202:9101"
```

```
public <T> T execute(String serviceId, LoadBalancerRequest<T> req
    ILoadBalancer loadBalancer = this.getLoadBalancer(serviceId);
    Server server = this.getServer(loadBalancer); loadBalancer:
    if (server == null) { server: "192.168.31.202:9001"
```

轮询调用了9001和9101，可知，getServer方法将微服务名转换为地址时采用了负载均衡机制



chooseServer方法

```
protected Server getServer(ILoadBalancer loadBalancer) { loadBalancer  
    return loadBalancer == null ? null : loadBalancer.chooseServer(o);
```

BaseLoadBalancer.class的chooseServer方法

```
public Server chooseServer(Object key) { key: "default"  
    if (this.counter == null) { counter: "BasicCounter"  
        this.counter = this.createCounter();  
    }  
  
    this.counter.increment();  
    if (this.rule == null) {  
        return null;  
    } else {  
        try {  
            return this.rule.choose(key);  
        } catch (Exception vars) {  
            logger.warn("LoadBalancer [{}]: Error choosi  
            return null;  
        }  
    }  
}
```


rule是IRule接口对象，IRule是负载均衡接口

```
public class BaseLoadBalancer extends AbstractLoadBalancer implements PrimeConnecti
= private static Logger logger = LoggerFactory.getLogger(BaseLoadBalancer.class);
private static final IRule DEFAULT_RULE = new RoundRobinRule();
private static final SerialPingStrategy DEFAULT_PING_STRATEGY = new SerialPingS
private static final String DEFAULT_NAME = "default";
private static final String PREFIX = "LoadBalancer_";
protected IRule rule; rule: ZoneAvoidanceRule@9947
```

```
public Server chooseServer(Object key) { key: "default"
    if (this.counter == null) { counter: "BasicCounter{
        this.counter = this.createCounter();
    }

    this.counter.increment();
    if (this.rule == null) {
        return null;
    } else {
        try {
            return this.rule.choose(key);
        } catch (Exception var3) {
            logger.warn("LoadBalancer [{}]: Error choosi
            return null;
        }
    }
}
```

执行到rule.choose方法
里面



找到choose方法中真正执行轮询算法的chooseRoundRobinAfterFiltering()

```
public Server choose(Object key) { key: "default"
    ILoadBalancer lb = this.getLoadBalancer();
    Optional<Server> server = this.getPredicate().chooseRoundRobinAfterFiltering(lb.getAllServers(), key);
    return server.isPresent() ? (Server) server.get() : null;
```

查看chooseRoundRobinAfterFiltering方法:

```
public Optional<Server> chooseRoundRobinAfterFiltering(List<Server> servers, Object loadBalancerKey) { loadBalancerKey: "default"
    List<Server> eligible = this.getEligibleServers(servers, loadBalancerKey); loadBalancerKey: "default" servers: size = 2
    return eligible.size() == 0 ? Optional.absent() : Optional.of(eligible.get(this.incrementAndGetModulo(eligible.size())));
```

incrementAndGetModulo方法:

```
private int incrementAndGetModulo(int modulo) { modulo: 2
    int current;
    int next;
    do {
        current = this.nextIndex.get(); nextIndex: "1"
        next = (current + 1) % modulo;
    } while (!this.nextIndex.compareAndSet(current, next) || current >= modulo);

    return current;
```



第一次请求购票微服务

```
do {  
    current = this.nextIndex.get();    nextIndex: "0"  
    next = (current + 1) % modulo;    modulo: 2    current: 0  
} while (!this.nextIndex.compareAndSet(current, next) || curre
```

第二次请求购票微服务

```
do {  
    current = this.nextIndex.get();    nextIndex: "1"  
    next = (current + 1) % modulo;    modulo: 2    current: 1  
} while (!this.nextIndex.compareAndSet(current, next) || current >= modulo);
```

```
@RequestMapping(value = "/order", method = RequestMethod.GET)  
public String order(){  
    // 模拟当前用户  
    Integer id = 2;  
    // 使用Ribbon帮助我们选择一个合适的服务实例（默认算法：轮询）  
    //ServiceInstance serviceInstance= client.choose("myshop-user");  
    User user = restTemplate.getForObject(url: "http://myshop-user/user/"+id, User.class)  
    System.out.println(user+"==正在购票...");  
    return "购票成功";  
}
```


修改Ribbon负载均衡算法

```
public interface IRule {  
    Server choose(Object var1)  
  
    void setLoadBalancer(ILoadBalancer balancer)  
  
    ILoadBalancer getLoadBalancer()  
}
```

- RandomRule
- ResponseTimeWeightedRule
- RetryRule
- RoundRobinRule
- Server

查看有哪些规则类

购票微服务application.yml中增加:

serviceld:

ribbon:

NFLoadBalancerRuleClassName: xxxxx

```
server:  
  port: 9002  
spring:  
  application:  
    name: myshop-web  
eureka:  
  client:  
    register-with-eureka: true # 作为客户端需要注册到Eureka  
    fetch-registry: true # 作为客户端需要从Eureka获取注册信息  
    service-url: # 客户端注册地址  
    defaultZone: http://localhost:8888/eureka
```

修改Ribbon的负载均衡算法

```
myshop-user:
```

```
  ribbon:
```

```
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

以debug方式重启购票微服务，并多次请求购票微服务

```
public <T> T execute(String serviceId, LoadBalancerRequest<T> req  
    ILoadBalancer loadBalancer = this.getLoadBalancer(serviceId);  
    Server server = this.getServer(loadBalancer); loadBalancer:  
    if (server == null) { server: "192.168.31.202:9101"
```

```
public <T> T execute(String serviceId, LoadBalancerRequest<T> req  
    ILoadBalancer loadBalancer = this.getLoadBalancer(serviceId);  
    Server server = this.getServer(loadBalancer); loadBalancer:  
    if (server == null) { server: "192.168.31.202:9101"
```

随机算法生效!!!

yaml × C RibbonLoadBalancerClient.class × IRule.class × C RandomRule.class

件, 字节码版本: 51.0 (Java 7) [下载源代码](#)

```
@SuppressWarnings({ RCN_REDUNDANT_NULLCHECK_OF_NULL_VALUE })  
public Server choose(ILoadBalancer lb, Object key) {  
    if (lb == null) {  
        return null;  
    } else {  
        Server server = null;
```

选择了
RandomRule的
choose方法

服务调用组件-OpenFeign

- 在前面的学习中，我们使用了Ribbon的负载均衡功能，一定程度上简化了远程调用时的代码：

```
@RequestMapping(value = "/order", method = RequestMethod.GET)
public String order(){
    // 模拟当前用户
    Integer id = 2;
    // 使用Ribbon帮助我们选择一个合适的服务实例（默认算法：轮询）
    //ServiceInstance serviceInstance= client.choose("myshop-user");
    User user = restTemplate.getForObject( url: "http://myshop-user/user/"+id, User.class)
    System.out.println(user+"==正在购票...");
    return "购票成功";
}
```

- 但我们以后需要编写类似的大量重复代码，格式基本相同，无非参数不一样。有没有更优雅的方式，来对这些代码再次优化呢？
- Spring Cloud提供OpenFeign组件，简化了以上操作！



认识OpenFeign

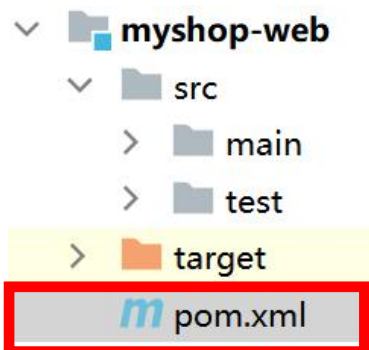
Feign是Netflix开发的声明式、模板化的HTTP客户端，其灵感来自 Retrofit、JAXRS-2.0以及 WebSocket。Feign可帮助我们更加便捷、优雅地调用HTTP API。

在Spring Cloud中，使用Feign非常简单——创建一个接口，并在接口上添加一些注解，代码就完成了。Feign支持多种注解，例如Feign自带的注解或者JAX-RS注解等。

Spring Cloud对 Feign进行了增强，使Feign支持了Spring MVC注解，并整合了Ribbon和 Eureka，从而让 Feign的使用更加方便。

OpenFeign使用步骤

- 修改服务调用者-购票微服务
 1. 导入OpenFeign的依赖

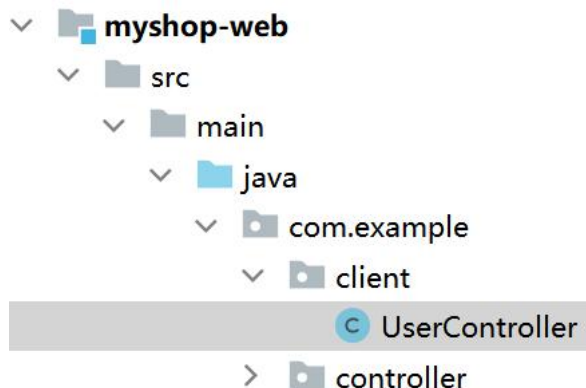


```
<!-- 导入OpenFeign依赖 -->
```

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

OpenFeign使用步骤

2. 编写代理接口（底层是动态代理，类似mybatis的mapper接口），代理接口使用@FeignClient指定调用的服务名



远程接口：client.UserController

myshop-user中UserController:

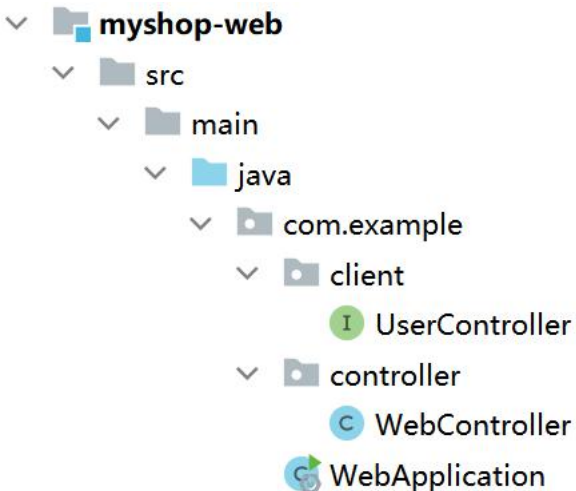
```
@RequestMapping("/user")
@RestController // @RestController=
public class UserController {

    /** 根据id查询用户 */
    @RequestMapping(value =("/{id}", method = RequestMethod.GET)
    public User findById(@PathVariable Integer id){
```

```
/**
 * 用户微服务远程接口
 * 三个注意事项:
 * 1) 使用@FeignClient, 声明需要调用微服务
 * 2) 检查@RequestMapping注解, value值(路径)是否完整
 * 3) @PathVariable注解的value不能省略的
 */
@FeignClient(value = "myshop-user")
public interface UserController {
    @RequestMapping(value = "/user/{id}", method = RequestMethod.GET)
    public User findById(@PathVariable(value = "id") Integer id);
}
```

OpenFeign使用步骤

3. WebController类中使用代理接口调用远程服务方法



```
User user =
restTemplate.getForObject("http://myshop-
user/user/"+id,User.class);
```

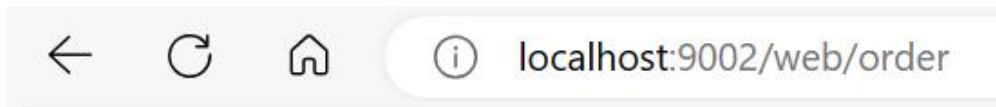


```
@Autowired
private UserController userController;
/* *
 * 购票方法（使用OpenFeign组件来简化调用代码）
 * @return
 */
@RequestMapping(value = "/order",method = RequestMethod.GET)
public String order(){
    // 模拟当前用户
    Integer id = 2;
    User user = userController.findById(id);
    System.out.println(user+"==正在购票...");
    return "购票成功";
}
```


4. 在启动类添加@EnableFeignClients注解，注释掉restTemplate

```
@SpringBootApplication
@EnableEurekaClient // 开启Eureka客户端自动配置
@EnableFeignClients // 开启Feign的自动配置
public class WebApplication {
    public static void main(String[] args) { SpringApplication.run(
//      /** 实例化RestTemplate */
//      @Bean
//      @LoadBalanced // 添加Ribbon负载均衡组件
//      public RestTemplate restTemplate(){
//          return new RestTemplate();
//      }
}
```

启动微服务，测试



购票成功

用户微服务22222

Hibernate: select user0_.id

用户微服务22222

Hibernate: select user0_.id

用户微服务22222

Hibernate: select user0_.id

2023-10-20 14:52:39.483 IN

用户微服务11111

Hibernate: select user0_.id

OpenFeign负载均衡

- Feign中本身已经集成了Ribbon依赖和自动配置，所以只要使用了Feign，就自动具备负载均衡啦！

```
▼ org.springframework.cloud:spring-cloud-starter-openfeign:2.0.0.M2
  org.springframework.cloud:spring-cloud-starter:2.0.0.M9 (omitted for duplicate)
  > org.springframework.cloud:spring-cloud-openfeign-core:2.0.0.M2
  > org.springframework:spring-web:5.0.5.RELEASE
  > org.springframework.cloud:spring-cloud-commons:2.0.0.M9
  io.github.openfeign:feign-core:9.5.1
  > io.github.openfeign:feign-slf4j:9.5.1
  > io.github.openfeign:feign-hystrix:9.5.1
  > io.github.openfeign:feign-java8:9.5.1
  org.springframework.cloud:spring-cloud-starter-netflix-ribbon:2.0.0.M8 (omitted for duplicate)
  org.springframework.cloud:spring-cloud-starter-netflix-archaius:2.0.0.M8 (omitted for duplicate)
```

- 也可以跟之前一样修改Ribbon的负载均衡算法