



下载APP



## 10 | 集成 Nacos : 如何通过服务发现机制向服务提供者发起调用 ?

2022-01-03 姚秋辰

《Spring Cloud 微服务项目实战》

课程介绍 >



讲述 : 姚秋辰

时长 21:39 大小 19.83M



你好，我是姚秋辰。

在上一课里，我们对 coupon-template-serv 和 coupon-calculation-serv 这两个服务做了微服务化改造，通过服务注册流程将它们注册到了 Nacos Server。这两个服务是以服务提供者的身份注册的，它们之间不会发生相互调用。为了发起一次完整的服务调用请求，我们还需要构建一个服务消费者去访问 Nacos 上的已注册服务。

领资料


coupon-customer-serv 就扮演了服务消费者的角色，它需要调用 coupon-template-serv 和 coupon-calculation-serv 完成自己的业务流程。今天我们就来动手改造 coupon-customer-serv 服务，借助 Nacos 的服务发现功能从注册中心获取可供调用的服务列表，并发起一个远程服务调用。



通过今天的内容，你可以了解如何使用 Webflux 发起远程调用，并熟练掌握如何搭建一套基于 Nacos 的服务治理方案。

## 添加 Nacos 依赖项和配置信息

在开始写代码之前，你需要将以下依赖项添加到 customer-customer-impl 子模块的 pom.xml 文件中。

 复制代码

```
1 <!-- Nacos服务发现组件 -->
2 <dependency>
3     <groupId>com.alibaba.cloud</groupId>
4     <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
5 </dependency>
6
7 <!-- 负载均衡组件 -->
8 <dependency>
9     <groupId>org.springframework.cloud</groupId>
10    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
11 </dependency>
12
13 <!-- webflux服务调用 -->
14 <dependency>
15     <groupId>org.springframework.boot</groupId>
16     <artifactId>spring-boot-starter-webflux</artifactId>
17 </dependency>
```

第一个依赖项你一定很熟悉了，它是 Nacos 服务治理的组件，我们在上一节课程中也添加了同款依赖项到 coupon-template-impl 和 coupon-calculation-impl 两个模块。

后面两个依赖项你应该是第一回见到，我来向你简单介绍一下。

**spring-cloud-starter-loadbalancer**：Spring Cloud 御用负载均衡组件

Loadbalancer，用来代替已经进入维护状态的 Netflix Ribbon 组件。我会在下一课带你深入了解 Loadbalancer 的功能，今天我们只需要简单了解下它的用法就可以了；

**spring-boot-starter-webflux**：Webflux 是 Spring Boot 提供的响应式编程框架，响应式编程是基于异步和事件驱动的非阻塞程序。Webflux 实现了 Reactive Streams 规范，内置了丰富的响应式编程特性。今天我将用 Webflux 组件中一个叫做 WebClient 的小工具发起远程服务调用。

添加好这两个依赖之后，你还需要做一番清理门户的工作，让 coupon-customer-serv 和另外两个微服务之间划清界限。

1. **删除实现层依赖**：从 coupon-customer-impl 的依赖项中删除 coupon-template-impl 和 coupon-calculation-impl；
2. **添加接口层依赖**：在 coupon-customer-impl 的依赖项中添加 coupon-template-api 和 coupon-calculation-api。

这样做的目的是**划清服务之间的依赖关系**，由于 coupon-customer-serv 是一个独立的微服务，它不需要将其他服务的“代码逻辑实现层”打包到自己的启动程序中一同启动。如果某个应用场景需要调用其它微服务，我们应该使用远程接口调用的方式对目标服务发起请求。因此，我们需要将对应接口的 Impl 实现层从 coupon-customer-impl 的依赖中删除，同时引入 API 层的依赖，以便构造请求参数和接收服务响应。

接下来，你还需要在 coupon-customer-impl 项目的 application.yml 文件中添加 Nacos 的配置项，我们直接从 coupon-template-impl 的配置项里抄作业就好了。将 spring.cloud.nacos 路径下的配置项 copy 到 coupon-customer-impl 项目中，如果你通过 spring.cloud.nacos.discovery.service 参数指定了服务名称，那你要**记得在抄作业的时候把名字改掉**，改成 coupon-customer-impl。

修改完依赖项和配置信息之后，你的代码一定冒出了不少编译错误。因为尽管我们已经将 coupon-template-impl 和 coupon-calculation-impl 依赖项删除，但 coupon-customer-impl 中的 CouponCustomerServiceImpl 仍然使用 Autowire 注入的方式调用本地服务。

所以接下来，我们就需要对调用层做一番改造，将 Autowire 注入本地服务的方式，替换为使用 WebClient 发起远程调用。

## 添加 WebClient 对象

为了可以用 WebClient 发起远程调用，你还需要在 Spring 上下文中构造一个 WebClient 对象。标准的做法是创建一个 Configuration 类，并在这个类中通过 @Bean 注解创建需要的对象。

所以我们在 coupon-customer-impl 子模块下创建了 com.geekbang.coupon.customer.Configuration 类，并声明 WebClient 的 Builder 对象。

[复制代码](#)

```
1 // Configuration注解声明配置类
2 @org.springframework.context.annotation.Configuration
3 public class Configuration {
4
5     // 注册Bean并添加负载均衡功能
6     @Bean
7     @LoadBalanced
8     public WebClient.Builder register() {
9         return WebClient.builder();
10    }
11
12 }
```

虽然上面的代码没几行，但我足足用了三个注解，这些注解各有用途。

**@Configuration 注解**：定义一个配置类。在 Configuration 类中定义的 @Bean 注解方法会被 AnnotationConfigApplicationContext 或者 AnnotationConfigWebApplicationContext 扫描并在上下文中进行构建；

**@Bean 注解**：声明一个受 Spring 容器托管的 Bean；

**@LoadBalanced 注解**：为 WebClient.Build 构造器注入特殊的 Filter，实现负载均衡功能，我在下一课会详细解释负载均衡的知识点。今天咱就好读书不求甚解就可以了，只需要知道这个注解的作用是在远程调用发起之前选定目标服务器地址。

WebClient 创建好了之后，你就可以在业务类中注入 WebClient 对象，并发起服务调用了。接下来，我就手把手带你将 CouponCustomerServiceImpl 里的本地方法调用替换成 WebClient 远程调用。

## 使用 WebClient 发起远程方法调用

首先，我们将 Configuration 类中声明的 WebClient 的 Builder 对象注入到 CouponCustomerServiceImpl 类中，两行代码简单搞定：

```
1 @Autowired
2 private WebClient.Builder webClientBuilder;
```

[复制代码](#)

接下来，我们开始改造第一个接口 requestCoupon。你需要将 requestCoupon 接口实现的第一行代码中的 CouponTemplateService 本地调用替换为 WebClient 远程调用。下面是改造之前的代码。

```
1 CouponTemplateInfo templateInfo = templateService.loadTemplateInfo(request.get
```

[复制代码](#)

远程接口调用的代码改造可以通过 WebClient 提供的“链式编程”轻松实现，下面是代码的完整实现。

```
1 CouponTemplateInfo templateInfo = webClientBuilder.build()
2     .get()
3     .uri("http://coupon-template-serv/template/getTemplate?id=" + request.
4     .retrieve()
5     .bodyToMono(CouponTemplateInfo.class)
6     .block();
```

[复制代码](#)

在这段代码中，我们应用了几个关键方法发起远程调用。

get：指明了 Http Method 是 GET，如果是其他请求类型则使用对应的 post、put、patch、delete 等方法；

uri：指定了访问的请求地址；

retrieve + bodyToMono：指定了 Response 的返回格式；

block：发起一个阻塞调用，在远程服务没有响应之前，当前线程处于阻塞状态。

在使用 uri 指定调用服务的地址时，你并不需要提供目标服务的 IP 地址和端口号，只需要将目标服务的服务名称 coupon-template-serv 告诉 WebClient 就好了。Nacos 在背后会通过服务发现机制，帮你获取到目标服务的所有可用节点列表。然后，WebClient 会通

过负载均衡过滤器，从列表中选取一个节点进行调用，整个流程对开发人员都是**透明的、无感知的**。

你可以看到，在代码中我使用了 `retrieve + bodyToMono` 的方式接收 `Response` 响应，并将其转换为 `CouponTemplateInfo` 对象。在这个过程中，我只接收了 `Response` 返回的 `Body` 内容，并没有对 `Response` 中包含的其它字段进行处理。

如果你需要获取完整的 `Response`，包括 `Http status`、`headers` 等额外数据，就可以使用 `retrieve + toEntity` 的方式，获取包含完整 `Response` 信息的 `ResponseEntity` 对象。示例如下，你可以自己在项目中尝试这种调用方式，体验下 `toEntity` 和 `bodyToMono` 的不同之处。

[复制代码](#)

```
1 Mono<ResponseEntity<CouponTemplateInfo>> entityMono = client.get()
2   .uri("http://coupon-template-serv/template/xxxx")
3   .accept(MediaType.APPLICATION_JSON)
4   .retrieve()
5   .toEntity(CouponTemplateInfo.class);
```

`WebClient` 使用了一种**链式编程**的风格来构造请求对象，链式编程就是我们熟悉的 `Builder` 建造者模式。仔细观察你会发现，大部分开源应用都在使用这种设计模式简化对象的构建。如果你需要在自己的项目中使用 `Builder` 模式，你可以借助 `Lombok` 组件的 `@Builder` 注解来实现。如果你对此感兴趣，可以自行了解 `Lombok` 组件的相关用法。

到这里，我们已经完成了 `requestCoupon` 方法的改造，接下来我们趁热打铁，动手去替换 `findCoupon` 和 `placeOrder` 方法中的本地调用。有了之前的基础，这次替换对你来说已经是小菜一碟了。

在 `findCoupon` 方法中，我们需要调用 `coupon-template-serv` 的服务批量查询 `CouponTemplate`。这里的方式和前面一样，我使用 `WebClient` 对本地调用进行了替换，你可以参考下面的源码。

[复制代码](#)

```
1 Map<Long, CouponTemplateInfo> templateMap = webClientBuilder.build().get()
2   .uri("http://coupon-template-serv/template/getBatch?ids=" + templateId
3   .retrieve()
```

```
4         .bodyToMono(new ParameterizedTypeReference<Map<Long, CouponTemplateInf  
5         block());
```

由于方法的返回值不是一个标准的 Json 对象，而是 `Map<Long, CouponTemplateInfo>` 类型，因此你需要构造一个 `ParameterizedTypeReference` 实例丢给 `WebClient`，告诉它应该将 `Response` 转化成什么类型。

现在，我们还剩下一个关键方法没有改造，那就是 `placeOrder`，它调用了 `coupon-calculation-serv` 计算最终的订单价格，你可以参考以下源码。

[复制代码](#)

```
1 ShoppingCart checkoutInfo = webClientBuilder.build()  
2     .post()  
3     .uri("http://coupon-calculation-serv/calculator/checkout")  
4     .bodyValue(order)  
5     .retrieve()  
6     .bodyToMono(ShoppingCart.class)  
7     .block();
```

和前面几处改造不同的是，这是一个 POST 请求，因此在使用 `webClient` 构造器的时候我调用了 `post` 方法；除此之外，它还需要接收订单的完整信息作为请求参数，因此我这里调用了 `bodyValue` 方法，将封装好的 `Order` 对象塞了进去。在 `coupon-customer-impl` 中剩下的一些远程调用方法，就留给你来施展拳脚做改造了。

到这里，我们整个 Nacos 服务改造就已经完成了。你可以在本地依次启动 `coupon-template-serv`、`coupon-calculation-serv` 和 `coupon-customer-serv`。启动成功后，再到 Nacos 控制台查看这三个服务是否已经全部注册到了 Nacos。

如果你是以集群模式启动了多台 Nacos 服务器，那么即便你在实战项目中只配置了一个 Nacos URL，并没有使用虚拟 IP 搭建单独的集群地址，注册信息也会传播到 Nacos 集群中的所有节点。





现在，动手搭建一套基于 Nacos 的服务治理方案对你而言一定不是难事儿了。动手能力是有了，但我们也不能仅仅满足于学会使用一套技术，**你必须要深入到技术的具体实现方案，才能从中汲取到养分，为你将来的技术方案设计提供参考。**

那么接下来，就让我带你去了解一下 Nacos 服务发现的底层实现，学习一下 Client 端是通过什么途径从 Nacos Server 获取服务注册表的。

## Nacos 服务发现底层实现

Nacos Client 通过一种**主动轮询**的机制从 Nacos Server 获取服务注册信息，包括地址列表、group 分组、cluster 名称等一系列数据。简单来说，Nacos Client 会开启一个本地的定时任务，每间隔一段时间，就尝试从 Nacos Server 查询服务注册表，并将最新的注册信息更新到本地。这种方式也被称之为“Pull”模式，即客户端主动从服务端拉取的模式。

负责拉取服务的任务是 UpdateTask 类，它实现了 Runnable 接口。Nacos 以开启线程的方式调用 UpdateTask 类中的 run 方法，触发本地的服务发现查询请求。

UpdateTask 这个类隐藏得非常深，它是 HostReactive 的一个内部类，我带你看一下经过详细注释的代码走读：

复制代码

```
1 public class UpdateTask implements Runnable {
2
3     // ....省略部分代码
```



```
4
5 // 获取服务列表
6 @Override
7 public void run() {
8     long delayTime = DEFAULT_DELAY;
9
10    try {
11        // 根据service name获取到当前服务的信息, 包括服务器地址列表
12        ServiceInfo serviceObj = serviceInfoMap
13            .get(ServiceInfo.getKey(serviceName, clusters));
14
15        // 如果为空, 则重新拉取最新的服务列表
16        if (serviceObj == null) {
17            updateService(serviceName, clusters);
18            return;
19        }
20
21        // 如果时间戳<=上次更新的时间, 则进行更新操作
22        if (serviceObj.getLastRefTime() <= lastRefTime) {
23            updateService(serviceName, clusters);
24            serviceObj = serviceInfoMap.get(ServiceInfo.getKey(serviceName
25        } else {
26            // 如果serviceObj的refTime更晚,
27            // 则表示服务通过主动push机制已被更新, 这时我们只进行刷新操作
28            refreshOnly(serviceName, clusters);
29        }
30        // 刷新服务的更新时间
31        lastRefTime = serviceObj.getLastRefTime();
32
33        // 如果订阅被取消, 则停止更新任务
34        if (!notifier.isSubscribed(serviceName, clusters) && !futureMap
35            .containsKey(ServiceInfo.getKey(serviceName, clusters))) {
36            // abort the update task
37            NAMING_LOGGER.info("update task is stopped, service:" + servic
38            return;
39        }
40        // 如果没有可供调用的服务列表, 则统计失败次数+1
41        if (CollectionUtils.isEmpty(serviceObj.getHosts())) {
42            incFailCount();
43            return;
44        }
45        // 设置延迟一段时间后进行查询
46        delayTime = serviceObj.getCacheMillis();
47        // 将失败查询次数重置为0
48        resetFailCount();
49    } catch (Throwable e) {
50        incFailCount();
51        NAMING_LOGGER.warn("[NA] failed to update serviceName: " + service
52    } finally {
53        // 设置下一次查询任务的触发时间
54        executor.schedule(this, Math.min(delayTime << failCount, DEFAULT_D
55    }
```

```
56     }  
57 }
```

在 UpdateTask 的源码中，它通过调用 updateService 方法实现了服务查询和本地注册表更新，在每次任务执行结束的时候，在结尾处它通过 finally 代码块设置了下一次 executor 查询的时间，周而复始循环往复。

以上，就是 Nacos 通过 UpdateTask 来查询服务端注册表的底层原理了。

那么现在我就要考考你了，你知道 UpdateTask 是在什么阶段由哪一个类首次触发的吗？我已经把这个藤交到你手上了，希望你能顺藤摸瓜，顺着 UpdateTask 类，从源码层面找到它的上游调用方，理清整个服务发现链路的流程。

## 总结

到这里，我们就完成了 geekbang-coupon-center 的 Nacos 服务治理改造。通过这两节课，你完整搭建了整个 Nacos 服务治理链路。在这条链路中，你通过**服务注册**流程实现了服务提供者的注册，又通过**服务发现**机制让服务消费者获取服务注册信息，还能通过 WebClient 发起**远程调用**。

在这段学习过程中，学会如何使用技术是一件很容易的事儿，而学会它背后的原理却需要花上数倍的功夫。在每节实战课里我都会加上一些源码分析，不仅授之以鱼，更要授之以渔，让你学会如何通过深入源码去学习一个框架。

为什么学习源码这么重要呢？我这么说吧，这就像你学习写作一样，小学刚开始练习写作的时候，我们是从“模仿”开始的。随着阅历、知识和阅读量的增多，你逐渐有了自己的思考和想法，建立了属于你的写作风格。学习技术也是类似的，好的开源框架就像一本佳作，Spring 社区孵化的框架更是如此，你从中可以汲取很多营养，进而完善自己的架构理念和技术细节。若干年后当你成为独当一面的架构师，这些平日里的积累终会为你所用。

## 思考题

如果某个服务节点碰到了某些异常状况，比如网络故障或者磁盘空间已满，导致无法响应服务请求。你知道 Nacos 通过什么途径来识别故障服务，并从 Nacos Server 的服务注册表中将故障服务剔除的吗？

好啦，这节课就结束啦。欢迎你把这节课分享给更多对 Spring Cloud 感兴趣的朋友。我是姚秋辰，我们下节课再见！

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 09 | 集成 Nacos：如何将服务提供者注册到 Nacos 服务器？

### 精选留言 (2)

 写留言



请叫我和尚

2022-01-03

再加一个问题，类UpdateTask是在HostReactor中，文中应该写错了不是在HostReactive中

作者回复: 同学看的很仔细，确实是个笔误



请叫我和尚

2022-01-03

2 个问题：

1. 项目 customer 上的注解：`@LoadBalancerClient(value = "coupon-template-serv", configuration = CanaryRuleConfiguration.class)`，没有说明是啥作用
2. 运行 customer 这个项目的时候会报错，Access to DialectResolutionInfo cannot be null when 'hibernate.dialect' not set , ...

展开

作者回复: Loadbalancer注解是spring cloud loadbalancer的内容，在Nacos讲完之后紧接着下一课里会讲到。如果项目版本和MySQL版本都保持和源码一致的话（在环境准备篇里有版本信息），那么新版里database-platform不用显式配置，默认auto-detect后会自动在日志里打印这一行

org.hibernate.dialect.Dialect: HHH000400: Using dialect: org.hibernate.dialect.MySQL8Dialect

