主讲老师：Fox老师

课程安排：

1. 微服务专题前三节课讲完，会进入Nacos源码专题，Nacos注册中心源码课安排：
    - Nacos 1.4.x 注册中心源码分析录播 （3节课）
    - grpc课程学习地址
    https://vip.tulingxueyuan.cn/detail/p_6192513ce4b09b5fe0b30066/6
    - Nacos 2.x 注册中心源码分析直播（2节课）
2. Ribbon&Feign的源码分析学习地址：
https://vip.tulingxueyuan.cn/detail/v_6079631fe4b09890f0e441e0/3

问题：

feign的调用链路

1. RequestInterceptor　如果扩展了请求一定会调
2. Client

```
有道云笔记：
文档：03 微服务调用组件Feign实战.note
链接：http://note.youdao.com/noteshare?id=c90bd2616b59aa073f9a1e330989ef04&sub=9124CCFE78124FC3A291F84C663C9AA6
```

1. RPC概述

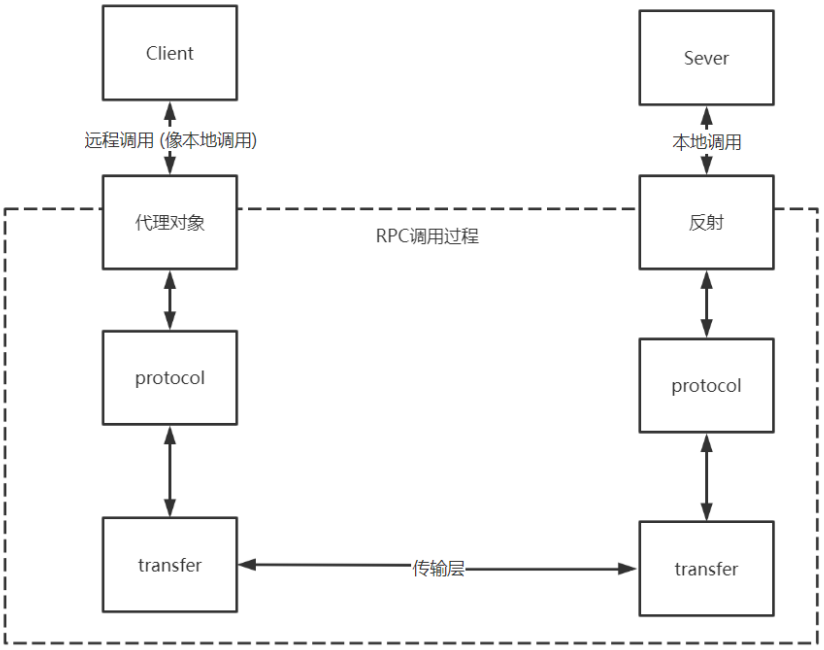
思考： 微服务之间如何方便优雅的实现服务间的远程调用?

RPC 全称是 Remote Procedure Call，即远程过程调用，其对应的是我们的本地调用。

RPC 的目的是：让我们调用远程方法像调用本地方法一样。

```
//本地调用
R result = orderService.findOrderByUserId(id);
//RPC远程调用 orderService为代理对象
R result = orderService.findOrderByUserId(id);
```

RPC框架设计架构



2. 什么是Feign

Feign是Netflix开发的声明式、模板化的HTTP客户端，Feign可帮助我们更加便捷、优雅地调用HTTP API。

Feign可以做到使用 HTTP 请求远程服务时就像调用本地方法一样的体验，开发者完全感知不到这是远程方法，更感知不到这是个 HTTP 请求。它像 Dubbo 一样，consumer 直接调用接口方法调用 provider，而不需要通过常规的 Http Client 构造请求再解析返回数据。它解决了让开发者调用远程接口就跟调用本地方法一样，无需关注与远程的交互细节，更无需关注分布式环境开发。

Spring Cloud openfeign对Feign进行了增强，使其支持Spring MVC注解，另外还整合了Ribbon和Eureka，从而使得Feign的使用更加方便。

2.1 Ribbon&Feign对比

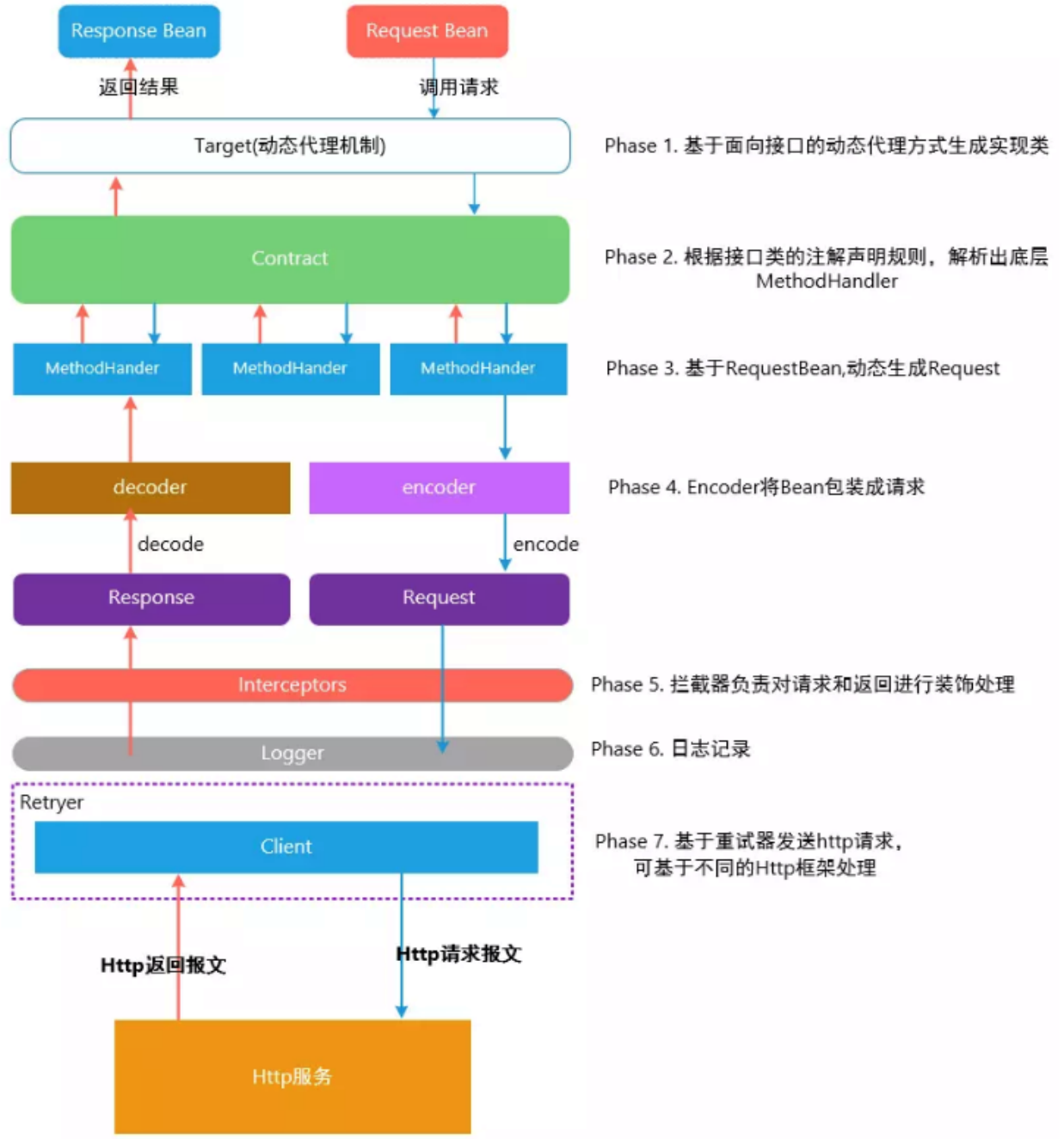
Ribbon+RestTemplate进行微服务调用

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

//调用方式
String url = "http://mall-order/order/findOrderByUserId/"+id;
R result = restTemplate.getForObject(url,R.class);
```

Feign进行微服务调用

```
@FeignClient(value = "mall-order",path = "/order")
public interface OrderFeignService {
 @RequestMapping("/findOrderByUserId/{userId}")
 public R findOrderByUserId(@PathVariable("userId") Integer userId);
}

@Autowired
OrderFeignService orderFeignService;
//feign调用
 R result = orderFeignService.findOrderByUserId(id);
```

2.2 Feign的设计架构



2.3 Spring Cloud Alibaba快速整合Feign

1）引入依赖

```
<!-- openfeign 远程调用 -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

2）编写调用接口+@FeignClient注解

```
@FeignClient(value = "mall-order",path = "/order")
```

```
public interface OrderFeignService {

 @RequestMapping("/findOrderByUserId/{userId}")
 public R findOrderByUserId(@PathVariable("userId") Integer userId);
}
```

3）调用端在启动类上添加@EnableFeignClients注解

```
@SpringBootApplication
@EnableFeignClients //扫描和注册feign客户端的beanDefinition
public class MallUserFeignDemoApplication {
 public static void main(String[] args) {
 SpringApplication.run(MallUserFeignDemoApplication.class, args);
 }
}
```

4）发起调用，像调用本地方式一样调用远程服务

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

 @Autowired
 OrderFeignService orderFeignService;

 @RequestMapping(value = "/findOrderByUserId/{id}")
 public R findOrderByUserId(@PathVariable("id") Integer id) {
  //feign调用
  R result = orderFeignService.findOrderByUserId(id);
  return result;
 }
}
```

提示： Feign 的继承特性可以让服务的接口定义单独抽出来，作为公共的依赖，以方便使用。

2.3 Spring Cloud Feign扩展

Feign 提供了很多的扩展机制，让用户可以更加灵活的使用。

日志配置

有时候我们遇到 Bug，比如接口调用失败、参数没收到等问题，或者想看看调用性能，就需要配置 Feign 的日志了，以此让 Feign 把请求信息输出来。

1）定义一个配置类，指定日志级别

```
// 注意： 此处配置@Configuration注解就会全局生效，如果想指定对应微服务生效，就不能配置@Configuration
@Configuration
public class FeignConfig {
 /**
 * 日志级别
 *
 * @return
 */
 @Bean
  public Logger.Level feignLoggerLevel() {
  return Logger.Level.FULL;
  }
}
```

通过源码可以看到日志等级有 4 种，分别是：

- **NONE【性能最佳，适用于生产】**：不记录任何日志（默认值）。
- **BASIC【适用于生产环境追踪问题】**：仅记录请求方法、URL、响应状态代码以及执行时间。
- **HEADERS**：记录BASIC级别的基础上，记录请求和响应的header。
- **FULL【比较适用于开发及测试环境定位问题】**：记录请求和响应的header、body和元数据。

2) 局部配置，让调用的微服务生效，在@FeignClient 注解中指定使用的配置类

```
@FeignClient(value = "mall-order",path = "/order",configuration = FeignConfig.class)
public interface OrderFeignService {

    @RequestMapping("/findOrderByUserId/{userId}")
    public R findOrderByUserId(@PathVariable("userId") Integer userId);
}
```

指定configuration，注意：FeignConfig不能添加@Configuration

3) 在yml配置文件中配置 Client 的日志级别才能正常输出日志，格式是"logging.level.feign接口包路径=debug"

```
logging:
 level:
 com.tuling.mall.feigndemo.feign: debug
```

测试：BASIC级别日志

```
: [OrderFeignService#findOrderByUserId] ---> GET http://mall-order/order/findOrderByUserId/1 HTTP/1.1
: [OrderFeignService#findOrderByUserId] <--- HTTP/1.1 200 (11ms)
```

补充：局部配置可以在yml中配置

对应属性配置类：

org.springframework.cloud.openfeign.FeignClientProperties.FeignClientConfiguration

```yaml
feign:
 client:
 config:
 mall-order: #对应微服务
 loggerLevel: FULL
```

契约配置

Spring Cloud 在 Feign 的基础上做了扩展，可以让 Feign 支持 Spring MVC 的注解来调用。原生的 Feign 是不支持 Spring MVC 注解的，如果你想在 Spring Cloud 中使用原生的注解方式来定义客户端也是可以的，通过配置契约来改变这个配置，Spring Cloud 中默认的是 SpringMvcContract。

1）修改契约配置，支持Feign原生的注解

```java
/**
 * 修改契约配置，支持Feign原生的注解
 * @return
 */
@Bean
public Contract feignContract() {
 return new Contract.Default();
}
```

注意：修改契约配置后，OrderFeignService 不再支持springmvc的注解，需要使用Feign原生的注解

2）OrderFeignService 中配置使用Feign原生的注解

```java
@FeignClient(value = "mall-order",path = "/order")
public interface OrderFeignService {
 @RequestLine("GET /findOrderByUserId/{userId}")
 public R findOrderByUserId(@Param("userId") Integer userId);
}
```

3）补充，也可以通过yml配置契约

```yaml
feign:
 client:
 config:
 mall-order: #对应微服务
 loggerLevel: FULL
 contract: feign.Contract.Default #指定Feign原生注解契约配置
```

通过拦截器实现参数传递

通常我们调用的接口都是有权限控制的，很多时候可能认证的值是通过参数去传递的，还有就是通过请求头去传递认证信息，比如 Basic 认证方式。

Feign 中我们可以直接配置 Basic 认证

```
1
2 @Configuration // 全局配置
3 public class FeignConfig {
4  @Bean
5  public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
6  return new BasicAuthRequestInterceptor("fox", "123456");
7  }
8 }
```

扩展点： feign.RequestInterceptor

每次 feign 发起http调用之前，会去执行拦截器中的逻辑。

```
1 public interface RequestInterceptor {
2
3  /**
4  * Called for every request. Add data using methods on the supplied {@link
RequestTemplate}.
5  */
6  void apply(RequestTemplate template);
7 }
```

使用场景

1. 统一添加 header 信息；
2. 对 body 中的信息做修改或替换；

自定义拦截器实现认证逻辑

```
1 public class FeignAuthRequestInterceptor implements RequestInterceptor {
2  @Override
3  public void apply(RequestTemplate template) {
4  // 业务逻辑
5  String access_token = UUID.randomUUID().toString();
6  template.header("Authorization",access_token);
7  }
8 }
9
10 @Configuration // 全局配置
11 public class FeignConfig {
12  @Bean
```

```
13   public Logger.Level feignLoggerLevel() {
14   return Logger.Level.FULL;
15   }
16   /**
17   * 自定义拦截器
18   * @return
19   */
20   @Bean
21   public FeignAuthRequestInterceptor feignAuthRequestInterceptor(){
22   return new FeignAuthRequestInterceptor();
23   }
24  }
```

测试

```
eignService#findOrderByUserId] ---> GET http://mall-order/order/findOrderByUserId/1 HTTP/1.1
eignService#findOrderByUserId] Authorization: 09558987-0e31-409b-b808-15663176a375
eignService#findOrderByUserId] ---> END HTTP (0-byte body)
eignService#findOrderByUserId] <--- HTTP/1.1 200 (60ms)
eignService#findOrderByUserId] connection: keep-alive
eignService#findOrderByUserId] content-type: application/json
```

补充：可以在yml中配置

```
1  feign:
2   client:
3   config:
4   mall-order: #对应微服务
5   requestInterceptors[0]: #配置拦截器
6   com.tuling.mall.feigndemo.interceptor.FeignAuthRequestInterceptor
```

mall-order端可以通过 @RequestHeader获取请求参数，建议在filter,interceptor中处理

超时时间配置

通过 Options 可以配置连接超时时间和读取超时时间，Options 的第一个参数是连接的超时时间（ms），默认值是 2s；第二个是请求处理的超时时间（ms），默认值是 5s。

全局配置

```
1  @Configuration
2  public class FeignConfig {
3   @Bean
4   public Request.Options options() {
5   return new Request.Options(5000, 10000);
6   }
7  }
```

yml中配置

```
1  feign:
2   client:
```

```
config:
mall-order: #对应微服务
# 连接超时时间，默认2s
connectTimeout: 5000
# 请求处理超时时间，默认5s
readTimeout: 10000
```

补充说明： Feign的底层用的是Ribbon，但超时时间以Feign配置为准

测试超时情况：

```
java.net.SocketTimeoutException: Read timed out
    at java.net.SocketInputStream.socketRead0(Native Method) ~[na:1.8.0_181]
    at java.net.SocketInputStream.socketRead(SocketInputStream.java:116) ~[na:1.8.0_181]
    at java.net.SocketInputStream.read(SocketInputStream.java:171) ~[na:1.8.0_181]
    at java.net.SocketInputStream.read(SocketInputStream.java:141) ~[na:1.8.0_181]
```

返回结果

```
{
    "timestamp": "2021-01-30T13:24:25.589+0000",
    "status": 500,
    "error": "Internal Server Error",
    "message": "Read timed out executing GET http://mall-order/order/findOrderByUserId/1",
    "path": "/user/findOrderByUserId/1"
}
```

客户端组件配置

Feign 中默认使用 JDK 原生的 URLConnection 发送 HTTP 请求，我们可以集成别的组件来替换掉 URLConnection，比如 Apache HttpClient，OkHttp。

Feign发起调用真正执行逻辑：**feign.Client#execute （扩展点）**

```
@Override
public Response execute(Request request, Options options) throws IOException {
  HttpURLConnection connection = convertAndSend(request, options);
  return convertResponse(connection, request);
}
```

配置Apache HttpClient

引入依赖

```
<!-- Apache HttpClient -->
<dependency>
 <groupId>org.apache.httpcomponents</groupId>
 <artifactId>httpclient</artifactId>
 <version>4.5.7</version>
</dependency>
<dependency>
 <groupId>io.github.openfeign</groupId>
```

```
 <artifactId>feign-httpclient</artifactId>
 <version>10.1.0</version>
</dependency>
```

然后修改yml配置，将 Feign 的 Apache HttpClient启用：

```
feign:
 #feign 使用 Apache HttpClient 可以忽略，默认开启
 httpclient:
 enabled: true
```

关于配置可参考源码： org.springframework.cloud.openfeign.FeignAutoConfiguration

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(ApacheHttpClient.class)
@ConditionalOnMissingClass("com.netflix.loadbalancer.ILoadBalancer")
@ConditionalOnMissingBean(CloseableHttpClient.class)
@ConditionalOnProperty(value = "feign.httpclient.enabled", matchIfMissing = true)
protected static class HttpClientFeignConfiguration {
```

测试：调用会进入feign.httpclient.ApacheHttpClient#execute

配置 OkHttp

引入依赖

```
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-okhttp</artifactId>
</dependency>
```

然后修改yml配置，将 Feign 的 HttpClient 禁用，启用 OkHttp，配置如下：

```
feign:
 #feign 使用 okhttp
 httpclient:
 enabled: false
 okhttp:
 enabled: true
```

关于配置可参考源码： org.springframework.cloud.openfeign.FeignAutoConfiguration

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(OkHttpClient.class)
@ConditionalOnMissingClass("com.netflix.loadbalancer.ILoadBalancer")
@ConditionalOnMissingBean(okhttp3.OkHttpClient.class)
@ConditionalOnProperty("feign.okhttp.enabled")
protected static class OkHttpFeignConfiguration {

    private okhttp3.OkHttpClient okHttpClient;
```

测试：调用会进入feign.okhttp.OkHttpClient#execute

GZIP 压缩配置

开启压缩可以有效节约网络资源，提升接口性能，我们可以配置 GZIP 来压缩数据：

```yaml
feign:
 # 配置 GZIP 来压缩数据
 compression:
 request:
 enabled: true
 # 配置压缩的类型
 mime-types: text/xml,application/xml,application/json
 # 最小压缩值
 min-request-size: 2048
  response:
  enabled: true
```

注意：只有当 Feign 的 Http Client 不是 okhttp3 的时候，压缩才会生效，配置源码在 FeignAcceptGzipEncodingAutoConfiguration

```java
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(FeignClientEncodingProperties.class)
@ConditionalOnClass(Feign.class)
@ConditionalOnBean(Client.class)
@ConditionalOnProperty(value = "feign.compression.response.enabled",
        matchIfMissing = false)
// The OK HTTP client uses "transparent" compression.
// If the accept-encoding header is present it disable transparent compression
@ConditionalOnMissingBean(type = "okhttp3.OkHttpClient")
@AutoConfigureAfter(FeignAutoConfiguration.class)
public class FeignAcceptGzipEncodingAutoConfiguration {

    @Bean
    public FeignAcceptGzipEncodingInterceptor feignAcceptGzipEncodingInterceptor(
            FeignClientEncodingProperties properties) {
        return new FeignAcceptGzipEncodingInterceptor(properties);
    }
```

核心代码就是 @ConditionalOnMissingBean（type="okhttp3.OkHttpClient"），表示 Spring BeanFactory 中不包含指定的 bean 时条件匹配，也就是没有启用 okhttp3 时才会进行压缩配置。

编码器解码器配置

Feign 中提供了自定义的编码解码器设置，同时也提供了多种编码器的实现，比如 Gson、Jaxb、Jackson。我们可以用不同的编码解码器来处理数据的传输。如果你想传输 XML 格式的数据，可以自定义 XML 编码解码器来实现获取使用官方提供的 Jaxb。

扩展点：Encoder & Decoder

```java
public interface Encoder {
 void encode(Object object, Type bodyType, RequestTemplate template) throw
s EncodeException;
```

```
}
public interface Decoder {
 Object decode(Response response, Type type) throws IOException, DecodeException, FeignException;
}
```

Java配置方式

配置编码解码器只需要在 Feign 的配置类中注册 Decoder 和 Encoder 这两个类即可:

```
@Bean
public Decoder decoder() {
 return new JacksonDecoder();
}
@Bean
public Encoder encoder() {
 return new JacksonEncoder();
}
```

yml配置方式

```
feign:
 client:
 config:
 mall-order: #对应微服务
 # 配置编解码器
 encoder: feign.jackson.JacksonEncoder
 decoder: feign.jackson.JacksonDecoder
```

3. Spring Cloud整合Dubbo

3.1 provider端配置

1）引入依赖

```

<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-dubbo</artifactId>
 <version>2.2.7.RELEASE</version>
</dependency>

<dependency>
 <groupId>com.alibaba.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

注意：因为spring cloud alibaba 2.2.8这个版本没有整合dubbo，所以需要指定dubbo的版本

每个 Spring Cloud Alibaba 版本及其自身所适配的各组件对应版本如下表所示：

Spring Cloud Alibaba Version	Sentinel Version	Nacos Version	RocketMQ Version	Dubbo Version	Seata Version
2.2.8.RELEASE	1.8.4	2.1.0	4.9.3	~	1.5.1
2021.0.1.0	1.8.3	1.4.2	4.9.2	~	1.4.2
2.2.7.RELEASE	1.8.1	2.0.3	4.6.1	2.7.13	1.3.0

2）修改application.yml

```yaml
dubbo:
  scan:
  # 指定 Dubbo 服务实现类的扫描基准包
  base-packages: com.tuling.mall.user.service
# application:
# name: ${spring.application.name}
  protocol:
  # dubbo 协议
  name: dubbo
  # dubbo 协议端口（ -1 表示自增端口，从 20880 开始）
  port: -1
# registry:
# #挂载到 Spring Cloud 注册中心 高版本可选
# address: spring-cloud://127.0.0.1:8848

spring:
  application:
  name: spring-cloud-dubbo-provider-user
  main:
  # Spring Boot2.1及更高的版本需要设定
  allow-bean-definition-overriding: true
  cloud:
  nacos:
  # Nacos 服务发现与注册配置
  discovery:
  server-addr: 127.0.0.1:8848
```

3）服务实现类上配置@DubboService暴露服务

```java
@DubboService
public class UserServiceImpl implements UserService {

 @Autowired
 private UserMapper userMapper;

 @Override
 public List<User> list() {
 return userMapper.list();
  }

  @Override
  public User getById(Integer id) {
  return userMapper.getById(id);
  }
}
```

3.2 consumer端配置

1）引入依赖

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

<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-dubbo</artifactId>
 <version>2.2.7.RELEASE</version>
 </dependency>

 <dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
 </dependency>
```

2）修改application.yml

```yaml
dubbo:
 cloud:
 # 指定需要订阅的服务提供方，默认值*，会订阅所有服务，不建议使用
 subscribed-services: spring-cloud-dubbo-provider-user
# application:
```

```
# name: ${spring.application.name}
 protocol:
 # dubbo 协议
 name: dubbo
  # dubbo 协议端口（ -1 表示自增端口，从 20880 开始）
  port: -1
# registry:
# #挂载到 Spring Cloud 注册中心 高版本可选
# address: spring-cloud://127.0.0.1:8848

spring:
 application:
 name: spring-cloud-dubbo-consumer-user
 main:
 # Spring Boot2.1及更高的版本需要设定
 allow-bean-definition-overriding: true
 cloud:
 nacos:
 # Nacos 服务发现与注册配置
 discovery:
 server-addr: 127.0.0.1:8848
```

当应用使用属性`dubbo.cloud.subscribed-services`为默认值时，日志中将会输出警告：

```
end you using the externalized property 'dubbo.cloud.subscribed-services' to specify the services
end you using the externalized property 'dubbo.cloud.subscribed-services' to specify the services
end you using the externalized property 'dubbo.cloud.subscribed-services' to specify the services
end you using the externalized property 'dubbo.cloud.subscribed-services' to specify the services
end you using the externalized property 'dubbo.cloud.subscribed-services' to specify the services
```

3）服务消费方通过@DubboReference引入服务

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

 @DubboReference
 private UserService userService;

 @RequestMapping("/info/{id}")
 public User info(@PathVariable("id") Integer id){

  return userService.getById(id);
  }

```

```
14   @RequestMapping("/list")
15   public List<User> list(){
16
17   return userService.list();
18   }
19  }
```

3.3 从Open Feign迁移到Dubbo

Dubbo Spring Cloud 提供了方案，可以从Open Feign迁移到Dubbo，即 @DubboTransported 注解。能够帮助服务消费端的 Spring Cloud Open Feign 接口以及 @LoadBalanced RestTemplate Bean 底层走 Dubbo 调用（可切换 Dubbo 支持的协议），而服务提供方则只需在原有 @RestController 类上追加 Dubbo @Servce 注解（需要抽取接口）即可，换言之，在不调整 Feign 接口以及 RestTemplate URL 的前提下，实现无缝迁移。

1）修改服务提供者

```
1  @DubboService
2  @Slf4j
3  @RestController
4  @RequestMapping("/user")
5  public class UserServiceImpl implements UserService {
6
7   @Autowired
8   private UserMapper userMapper;
9
10   @Override
11   @RequestMapping("/list")
12   public List<User> list() {
13   log.info("查询user列表");
14   return userMapper.list();
15   }
16
17   @Override
18   @RequestMapping("/getById/{id}")
19   public User getById(@PathVariable("id") Integer id) {
20   return userMapper.getById(id);
21   }
22  }
```

2）服务消费端引入依赖

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

<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-dubbo</artifactId>
 <version>2.2.7.RELEASE</version>
</dependency>

<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>

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

3）添加Feign的实现，启动类上添加@EnableFeignClients

```
@SpringBootApplication
@EnableFeignClients
public class SpringCloudDubboConsumerUserFeignApplication {

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

}
```

4）Feign接口添加 @DubboTransported 注解

```
@FeignClient(value = "spring-cloud-dubbo-provider-user-feign",path = "/use
r")
@DubboTransported(protocol = "dubbo")
public interface UserDubboFeignService {

 @RequestMapping("/list")
 public List<User> list();

```

```
8   @RequestMapping("/getById/{id}")
9   public User getById(@PathVariable("id") Integer id);
10  }
11
12  @FeignClient(value = "spring-cloud-dubbo-provider-user-feign",path = "/us
er")
13  public interface UserFeignService {
14
15   @RequestMapping("/list")
16   public List<User> list();
17
18
19   @RequestMapping("/getById/{id}")
20   public User getById(@PathVariable("id") Integer id);
21  }
```

5）调用对象添加 @DubboTransported 注解，发起调用

```
1  @RestController
2  @RequestMapping("/user")
3  public class UserConstroller {
4
5   @DubboReference
6   private UserService userService;
7
8   @RequestMapping("/info/{id}")
9   public User info(@PathVariable("id") Integer id){
10   return userService.getById(id);
11   }
12
13   @Autowired
14   private UserFeignService userFeignService;
15
16   @RequestMapping("/list")
17   public List<User> list(){
18   return userFeignService.list();
19   }
20
21   @Autowired
22   private UserDubboFeignService userDubboFeignService;
23
24   @RequestMapping("/list2")
```

```
25  public List<User> list2(){
26
27  return userDubboFeignService.list();
28  }
29
30 }
```