从服务化到微服务

- 微服务架构与传统单体架构的对比
- 微服务架构与 SOA 服务化的对比
- 微服务架构的核心要点和实现原理
- 微服务架构中职能团队的划分
- 微服务的去中心化治理
- 微服务的交互模式
- 共享数据集成的缺点
- 微服务的分解和组合模式
- 组合微服务方式
- 微服务的容错模式
- 微服务的粒度
- Java平台微服务架构的项目组织形式
- 微服务项目的依赖关系
- 微服务项目的层级结构
- 微服务项目的持续发布
- 服务化管理和治理框架的技术选型
- springcloud：
- springcloud alibaba
- springcloud netflix
- 快速启动

Ribbon

- **客户端负载均衡和服务端负载均衡的区别**
- Ribbon介绍

RestTemplate

- GET请求
- POST请求
- PUT请求
- DELETE请求
- 总结：

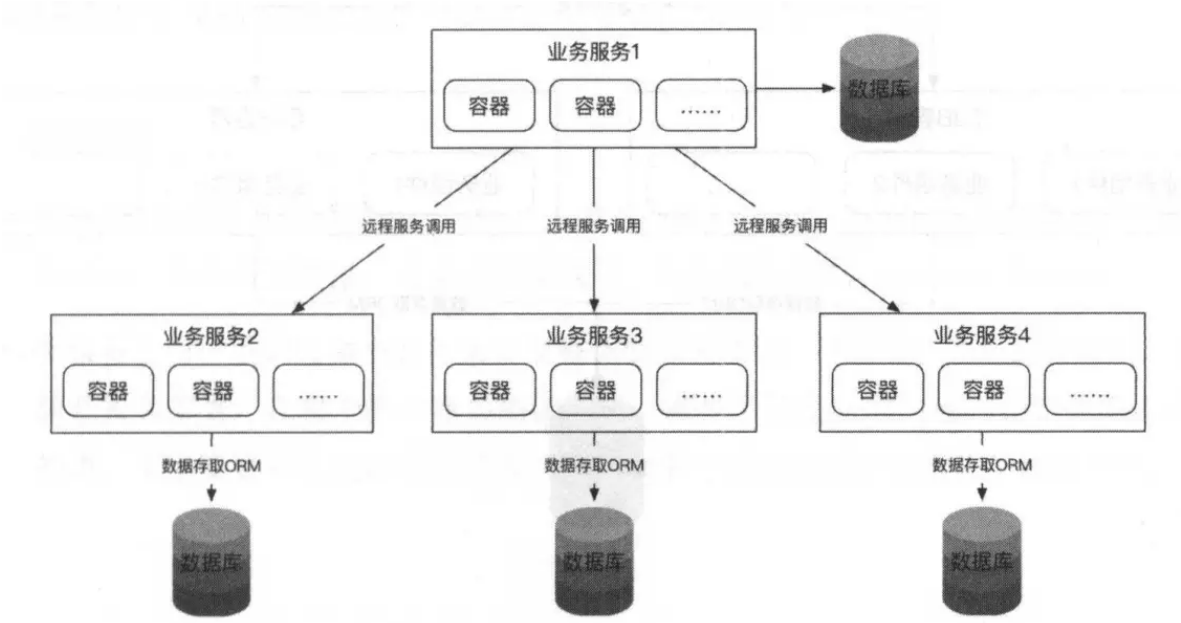
Feign的目标

- 引入Feign
- Feign的使用
- FeignClient注解的一些属性
- Feign自定义处理返回的异常
- Feign使用OKhttp发送request
- Feign原理简述
- Feign开启GZIP压缩
- 作用在所有Feign Client上的配置方式
- Feign Client开启日志
- Feign 的GET的多参数传递
- Feign解决了什么问题?
- Feign是如何设计的?
- PHASE 1. 基于面向接口的动态代理方式生成实现类
- PHASE 2. 根据Contract协议规则，解析接口类的注解信息，解析成内部表现：
- 默认Contract 实现
- 基于Spring MVC的协议规范SpringMvcContract:
- PHASE 3. 基于 RequestBean，动态生成Request
- PHASE 4. 使用Encoder 将Bean转换成 Http报文正文（消息解析和转码逻辑）
- PHASE 5. 拦截器负责对请求和返回进行装饰处理
- PHASE 6. 日志记录
- PHASE 7 . 基于重试器发送HTTP请求
- PHASE 8. 发送Http请求
- Feign 的性能怎么样?

从服务化到微服务

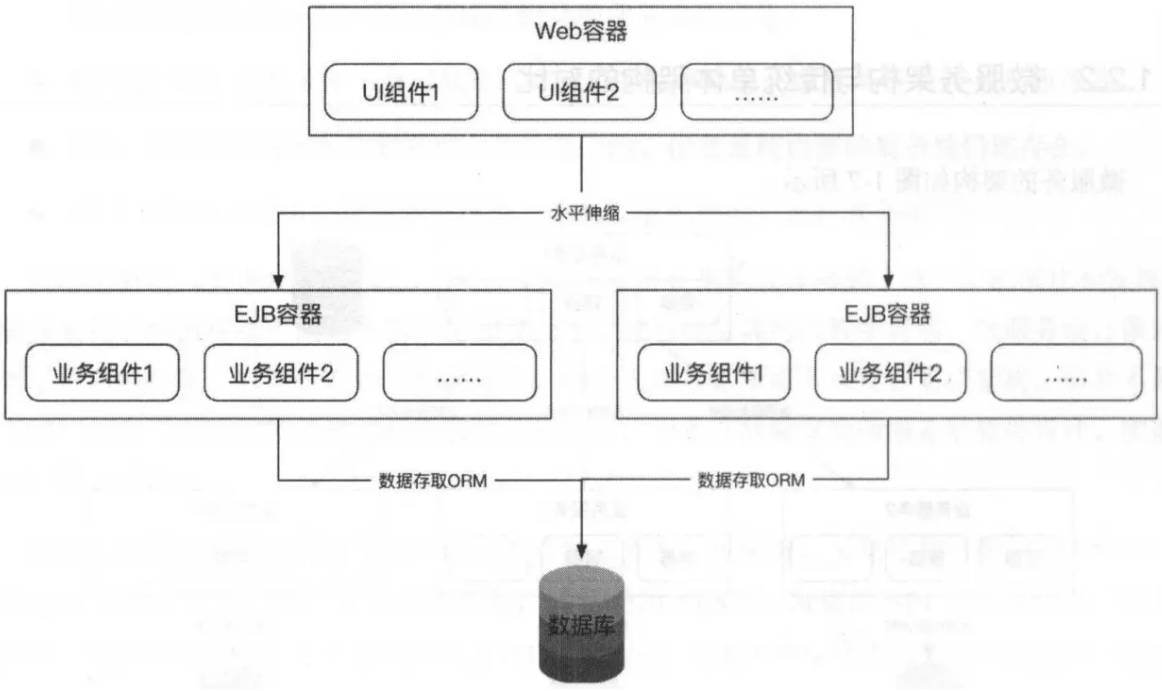
微服务架构与传统单体架构的对比

微服务的架构：



- 微服务把每一个职责单一的功能放在一个独立的服务中 。
- 每个服务运行在一个单独的进程中。
- 每个服务有多个实例运行。运行在容器化的平台，可以平滑伸缩。
- 每个服务有自己的数据存储。独立的数据，缓存，消息队列等。
- 每个服务有独立的运营平台。每个服务高度自治，内部变化对外透明。
- 每个服务可以根据性能独立地水平伸缩。

传统单体架构的伸缩架构：



- 传统单体架构将所有模块化组件混合后运行在同一个服务JVM进程中 。

- 可对包含多个模块化组件的整体JVM进程进行水平扩展，而无法对某个模块化组件进行水平扩展。
- 某个模块化组件发生变化时，需要所有的模块化组件进行编译、打包和上线。
- 模块间的依赖将会不清晰，互相糯合、互相依赖。

微服务架构与 SOA 服务化的对比

微服务架构与 SOA 服务化虽然一脉相承，却略有不同。

- 目的不同。SOA强调异构服务之间协作和集成。微服务目的是拆分，实现敏捷开发部署。
- 部署方式不同。拆分成多个小服务，使用敏捷扩容，Docker实现自动化容器管理。SOA服务将多个服务打包在一起，部署在一个服务器上。
- 服务粒度不同。微服务拆分粒度更细，职责单一。通过服务组合实现业务流程。SOA对粒度没有要求，通常是粗粒度的。
- 微服务架构 = 80%的SOA服务架构思想 + 100%的组件化架构思想 + 80%的领域建模思想

微服务架构的核心要点和实现原理

微服务架构中职能团队的划分

微服务架构按照业务的功能进行划分，每个单一的业务功能叫作一个服务，每个服务对应一个独立的职能团队，团队里包含用户交互UI设计师、后台服务开发人员、DBA、运营和运维人员。

微服务的去中心化治理

微服务架构倡导去中心化的服务管理和治理，尽量不设置中心化的管理服务，最差也需要在中心化的管理服务宕机时有替代方案和设计。

微服务的交互模式

- 读者容错模式：微服务化中服务提供者和消费者之间如何对接口的改变进行容错。
- 消费者驱动契约模式：用来定义服务化中服务之间交互接口改变的最佳规则。
  +去数据共享模式：不要共享缓存和数据库等资源，也不要使用总线模式，服务之间的通信和交互只能依赖定义良好的接口，通常使用RESTful样式的API或者透明的RPC调用框架。

共享数据集成的缺点

- 微服务之间的交互除了接口契约，还存在数据存储契约。
- 上游数据格式变化，可能导致下游的处理逻辑出问题。
- 多个服务共享一个资源服务，资源服务的运维难以划清职责和界限。
- 多机房部署，需要考虑到服务和资源的路由情况，跨机房调用，难以实现服务自治。

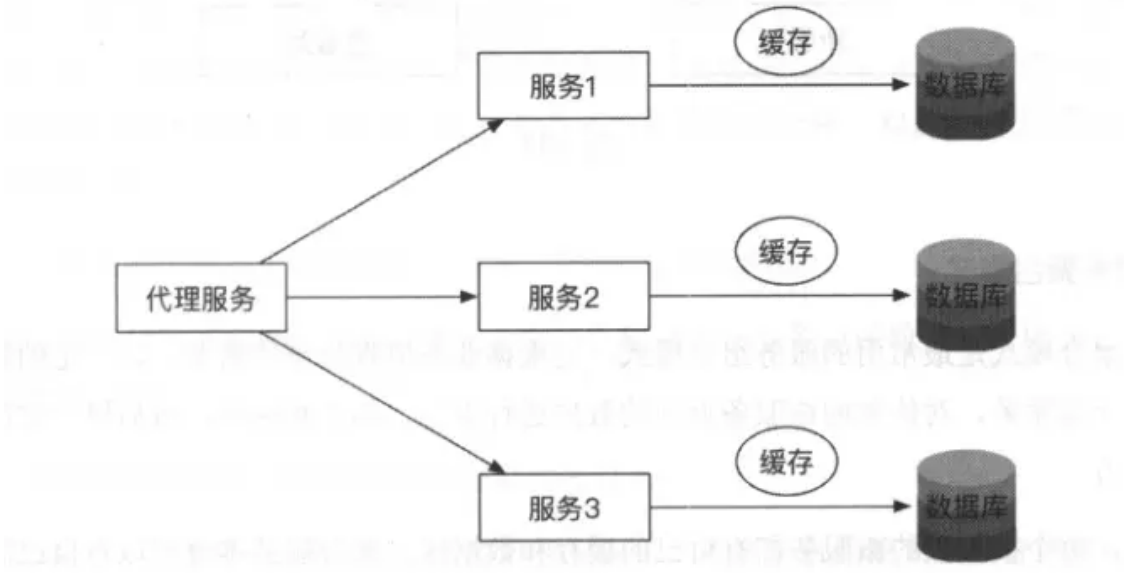
微服务的分解和组合模式

使用微服务架构划分服务和团队是微服务架构实施的重要一步，良好的划分和拆分使系统达到松耦合和高内聚的效果，然后通过微服务的灵活组装可以满足上层的各种各样的业务处理需求。

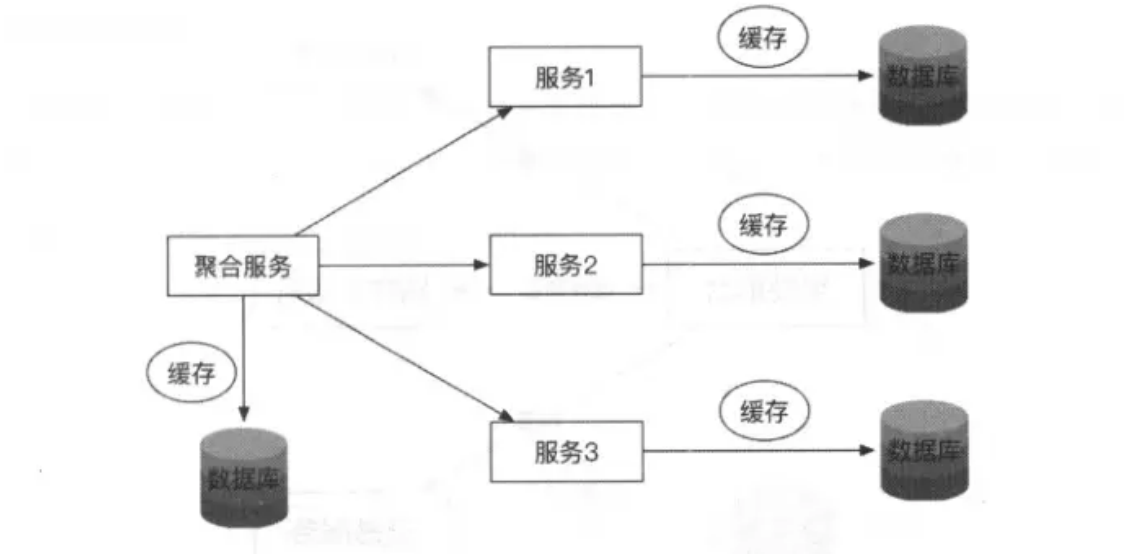
组合微服务方式

- 服务代理模式：最简单的服务组合模式，它根据业务的需求选择调用后端的某个服务。在返回给使用端之前，代理可以对后端服务的输出进行加工，也可以直接把后端服务的返回结果返回给使用端。一般会对读请求切换设计一个开关，开关打开时查询新系统，开关关闭时查询老系统。典型的

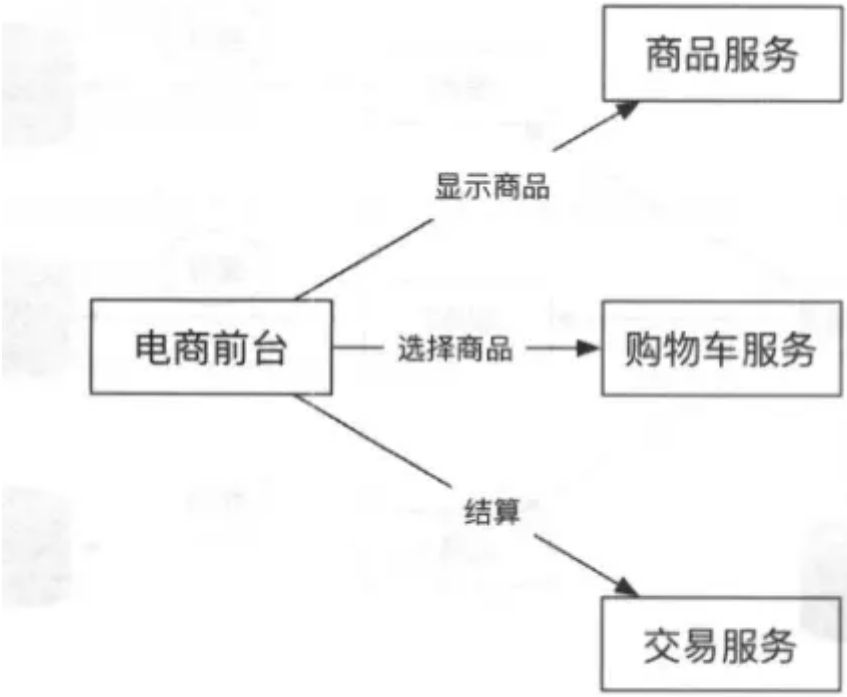
案例：平滑的系统迁移。



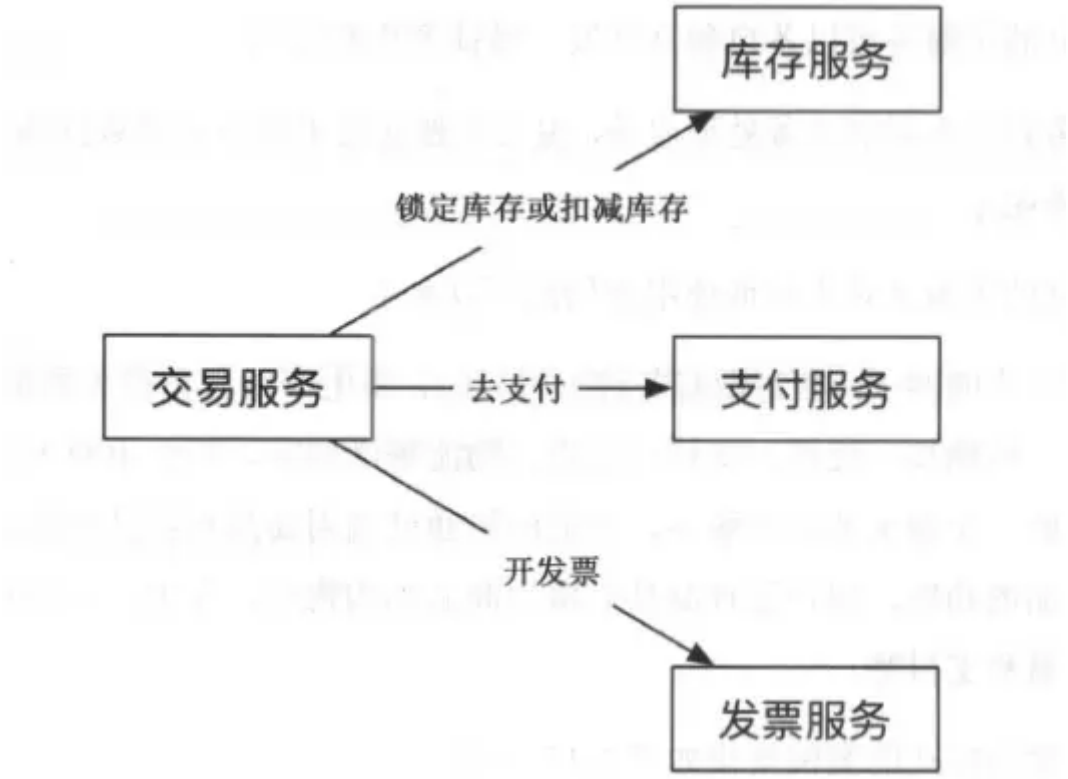
- 服务聚合模式：最常用的服务组合模式，它根据业务流程处理的需要，以一定的顺序调用依赖的多个微服务，对依赖的微服务返回的数据进行组合、加工和转换，最后以一定的形式返回给使用方。



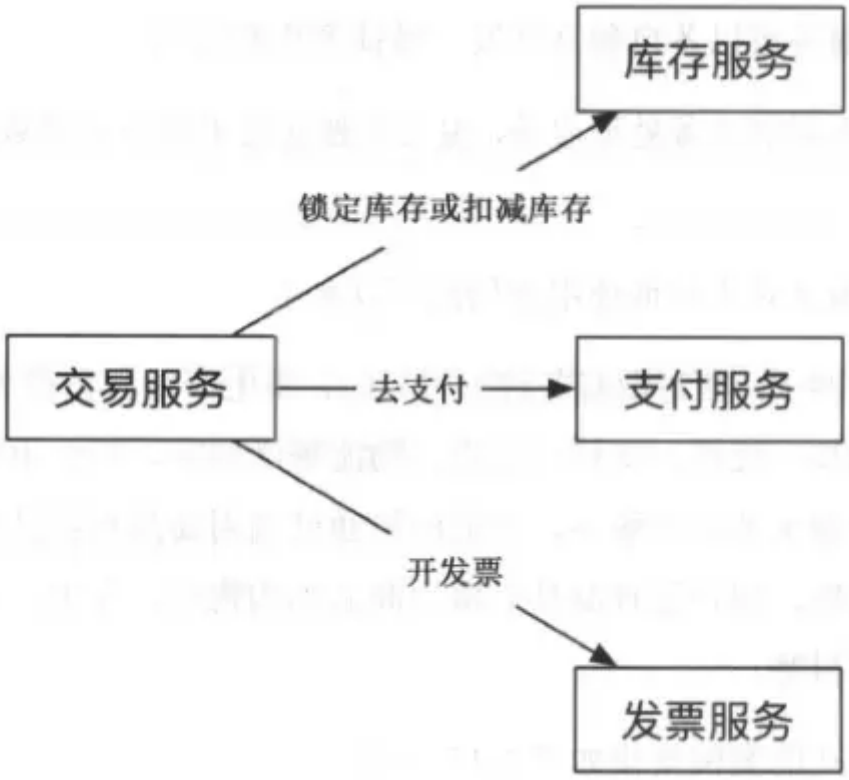
- 聚合服务可以是前端应用



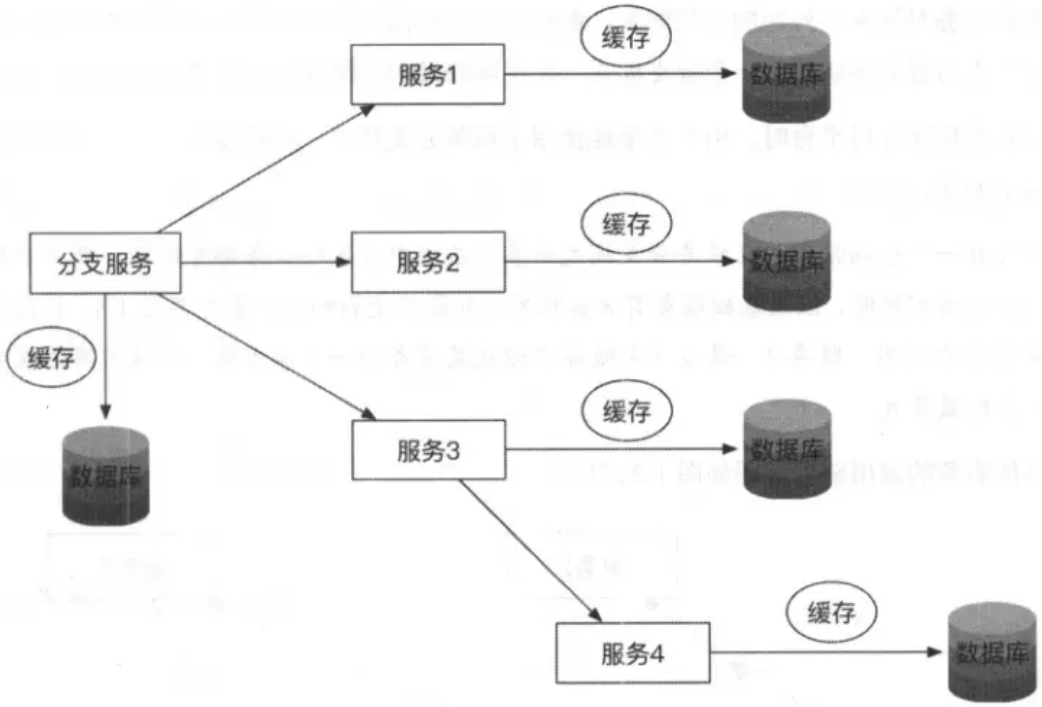
- 也可以是纯后台服务



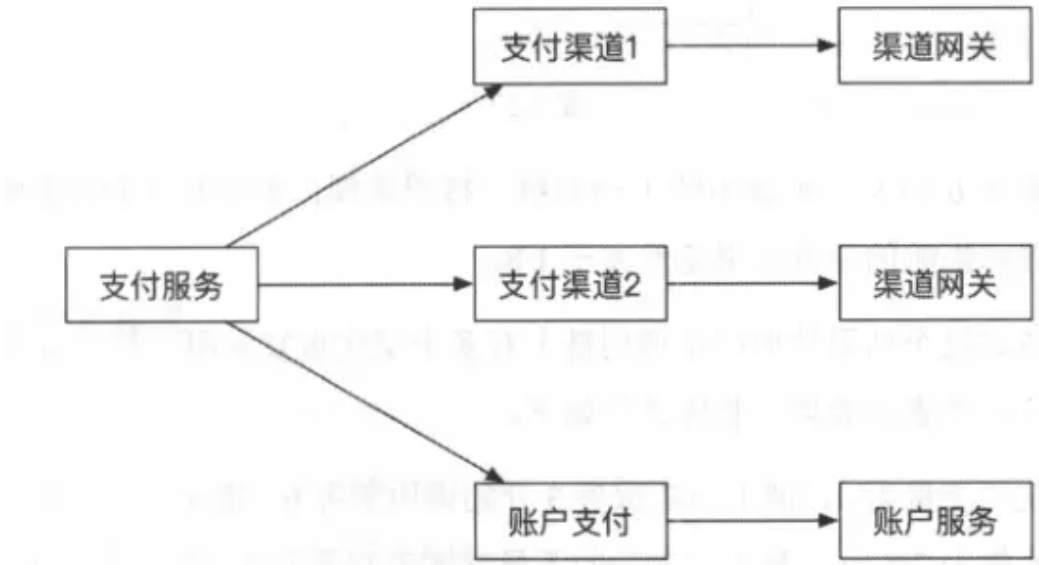
- 服务串联模式：类似于一个工作流，服务直接的调用通常使用同步的RESTful风格的远程调用实现。优点是在非串联服务的正后面增加节点，串联服务无感知；缺点是不建议服务的层级太多。



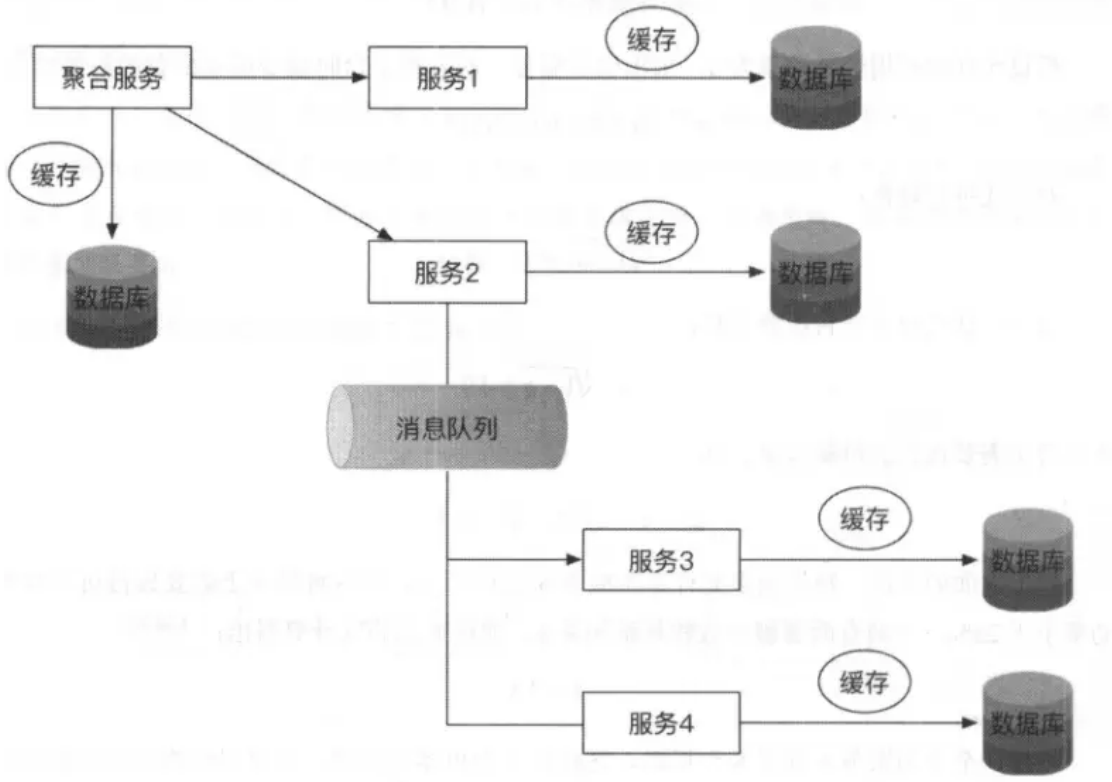
- 服务分支模式：是服务代理模式、服务聚合模式和服务串联模式相结合的产物。



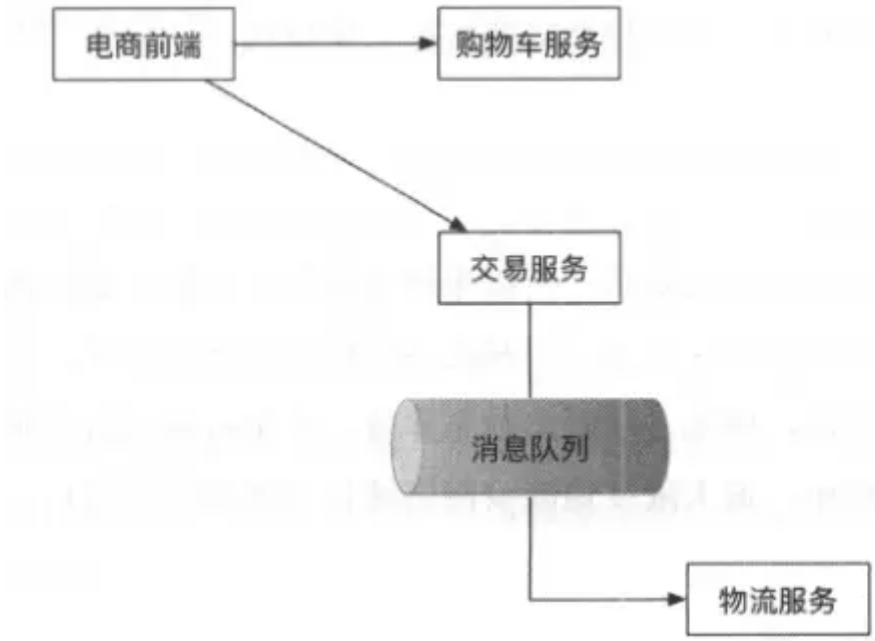
- 以电商平台的支付服务架构为例



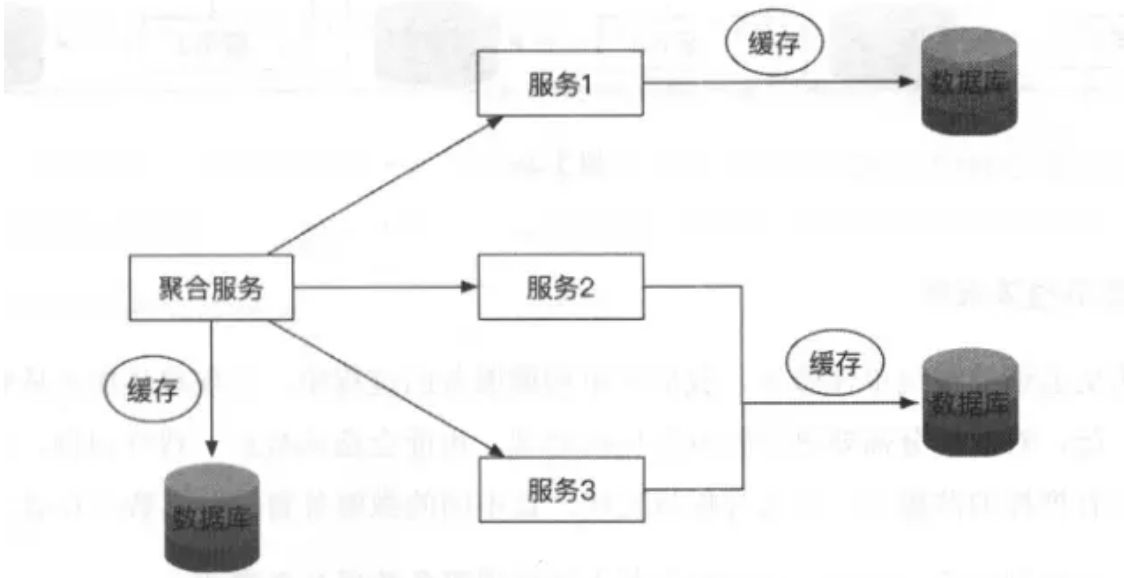
- 服务异步消息模式：



- 以电商平台交易完成后向物流系统发起消息通知为例



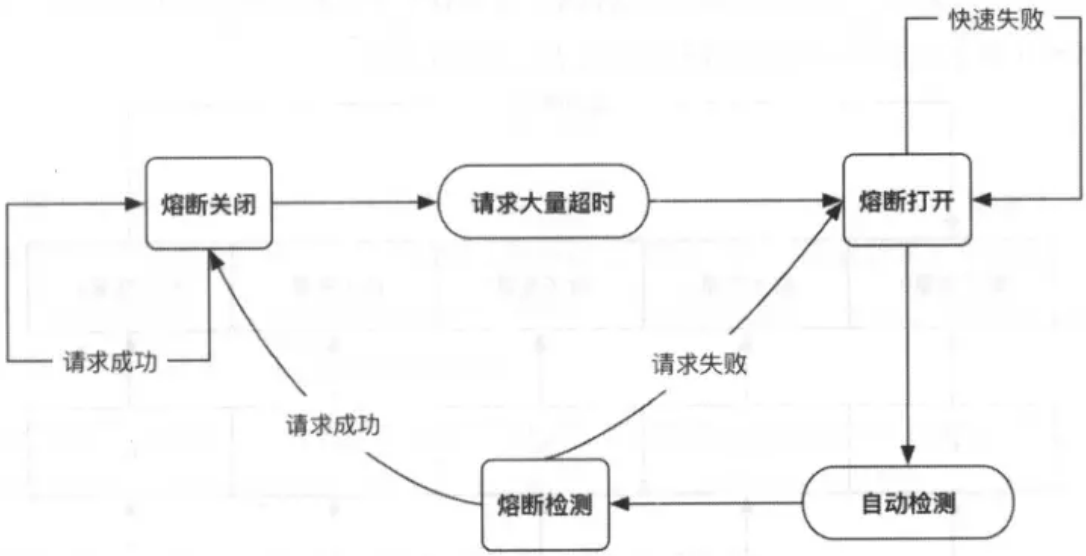
- 服务共享数据模式：其实是反模式，用于单元化架构和遗留的整体服务。



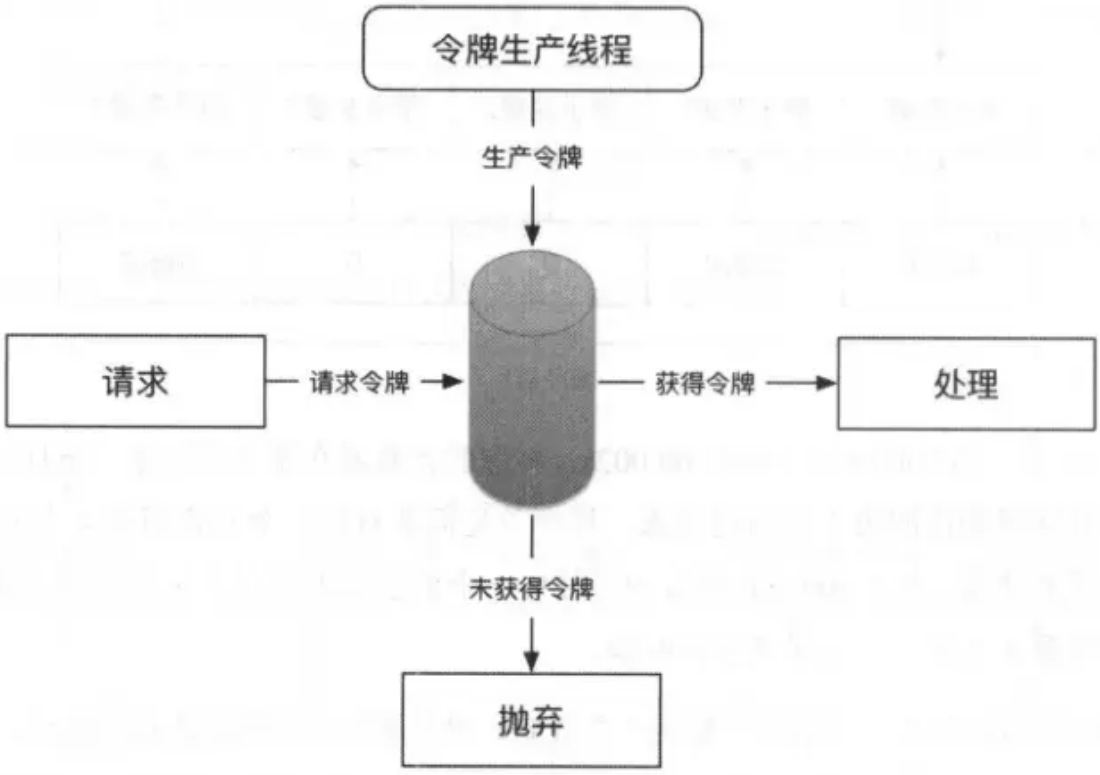
微服务的容错模式

网络通信是不稳定、不可靠的，一个服务依赖的服务可能出错、超时或者宕机。

- 舱壁隔离模式：微服务容器分组和线程池隔离
- 熔断模式



- 限流模式：计数器，令牌桶，信号量



- 失效转移模式：当发生了熔断和限流时，采用快速失败的策略，直接返回使用方错误；若有备份服务，迅速切换；有可能是某台机器出问题，采用重试方式。

微服务的粒度

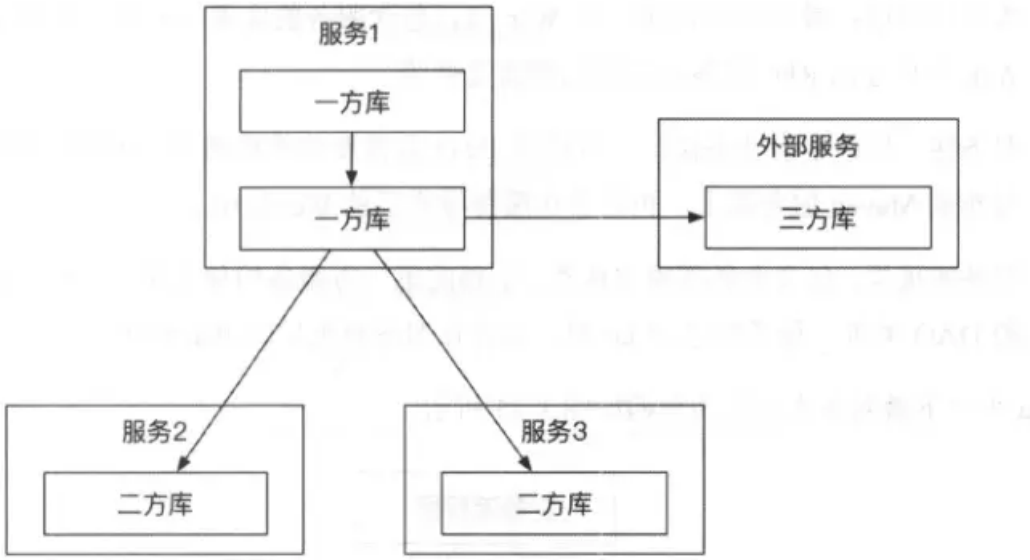
微服务初衷是按照业务的功能进行拆分，直到服务功能和职责单一，甚至拆到不可再拆。原则是拆分到可以合理排版底层的自服务来获得相应的组合服务，同时考虑团队人员分配。

Java平台微服务架构的项目组织形式

微服务项目的依赖关系

- 一方库：本服务在JVM进程内依赖的Jar包。
- 二方库：在服务外通过网络通信或者RPC调用的服务的Jar包。

- 三方库：所依赖的其他公司或者组织提供的服务或者模块。



微服务项目的层级结构

Java 微服务项目的层级结构一般为：服务导出层、接口层和逻辑实现层，

微服务项目的持续发布

微服务项目需要实现自动化的持续部署和持续集成的功能，包括：代码管理、自动编译、发布QA、自动化测试、性能测试、准生产部署和测试、生产环境发布等。

服务化管理和治理框架的技术选型

springcloud：

Spring Cloud为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理，服务发现，断路器，智能路由，微代理，控制总线）。分布式系统的协调导致了样板模式, 使用Spring Cloud开发人员可以快速地支持实现这些模式的服务和应用程序

springcloud alibaba

- **服务限流降级**：默认支持 WebServlet、WebFlux, OpenFeign、RestTemplate、Spring Cloud Gateway, Zuul, Dubbo 和 RocketMQ 限流降级功能的接入，可以在运行时通过控制台实时修改限流降级规则，还支持查看限流降级 Metrics 监控。
- **服务注册与发现**：适配 Spring Cloud 服务注册与发现标准，默认集成了 Ribbon 的支持。
- **分布式配置管理**：支持分布式系统中的外部化配置，配置更改时自动刷新。
- **消息驱动能力**：基于 Spring Cloud Stream 为微服务应用构建消息驱动能力。
- **分布式事务**：使用 @GlobalTransactional 注解， 高效并且对业务零侵入地解决分布式事务问题。。
- **阿里云对象存储**：阿里云提供的海量、安全、低成本、高可靠的云存储服务。支持在任何应用、任何时间、任何地点存储和访问任意类型的数据。
- **分布式任务调度**：提供秒级、精准、高可靠、高可用的定时（基于 Cron 表达式）任务调度服务。同时提供分布式的任务执行模型，如网格任务。网格任务支持海量子任务均匀分配到所有 Worker（schedulerx-client）上执行。
- **阿里云短信服务**：覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。

springcloud netflix

Spring Cloud Netflix features:

- Service Discovery: Eureka instances can be registered and clients can discover the instances using Spring-managed beans
- Service Discovery: an embedded Eureka server can be created with declarative Java configuration
- Circuit Breaker: Hystrix clients can be built with a simple annotation-driven method decorator
- Circuit Breaker: embedded Hystrix dashboard with declarative Java configuration
- Declarative REST Client: Feign creates a dynamic implementation of an interface decorated with JAX-RS or Spring MVC annotations
- Client Side Load Balancer: Ribbon
- External Configuration: a bridge from the Spring Environment to Archaius (enables native configuration of Netflix components using Spring Boot conventions)
- Router and Filter: automatic regsitration of Zuul filters, and a simple convention over configuration approach to reverse proxy creation

快速启动

https://start.spring.io/

Ribbon

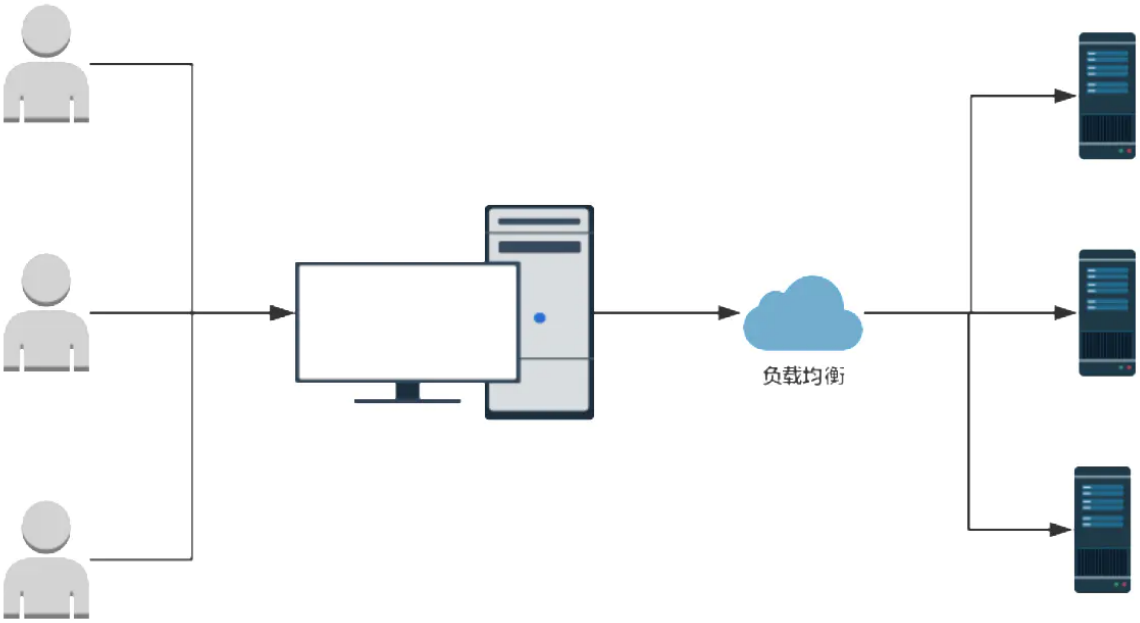
客户端负载均衡和服务端负载均衡的区别

客户端负载均衡和服务端负载均衡最大的不同在于服务清单所在的位置。客户端负载均衡中，客户端中都维护着自己要访问的服务段清单，而这些清单都来源于服务注册中心，但是服务端负载均衡的服务清单是无法自己来维护的。

Ribbon介绍

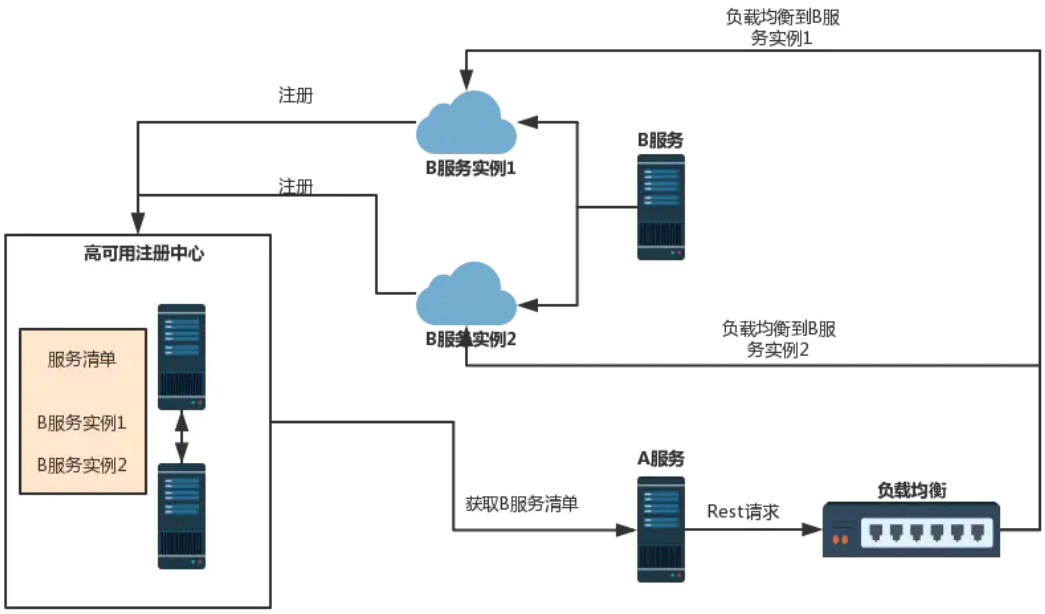
SpringCloud Ribbon 是一个基于HTTP 和TCP的客户端负载均衡工具，它基于Netflix Ribbon实现。通过Spring Cloud的封装，可以让我们轻松的，面向服务的REST模板请求自动转换成客户端负载均衡的服务调用。

负载均衡在系统架构中很重要，负载均衡是对系统的高可用，网络压力的环节和处理能力扩容的重要手段。在高并发的web请求中，大量用户同时点击一个登陆按钮，导致短时间内网络带宽急剧增加，服务器负载过重，这个时候我们就要对这些大量的请求做一个引流，把这些请求分发到不同的服务器上去，在这个过程中，使用什么样的算法把这些请求分发到不同的服务器上去，这是负载均衡需要做的。



在Spring Cloud 构建的微服务集群中，不同服务之间的通信是通过HTTP的Rest请求完成的，可能出现A服务对B的大量Rest请求，这个时候就要对A服务做客户端的负载均衡，B服务要创建多个实例来就收这些负载均衡后的Rest请求。

在客户端负载均衡中，所有客户端节点都维护这自己要访问的服务端清单，而这些服务端的清单来自于服务注册中心。




```
@RestController
@RequestMapping("/user")
public class UserRibbonController {
    @Autowired
    private RestTemplate restTemplate;
    @Value("${service-url.user-service}")
    private String userServiceUrl;

    @GetMapping("/{id}")
    public CommonResult getUser(@PathVariable Long id) {
        return restTemplate.getForObject(userServiceUrl + "/user/{1}",
CommonResult.class, id);
    }
}
```

创建一个返回值是RestTemplate的方法，并且使用@Bean注解，这样这个RestTemplate对象就会加载到Spring容器中。

使用@LoadBalanced注解，开启客户端的负载均衡，并且该注解使用在RestTemplate对象上，使得通过RestTemplate对象发送的Rest请求会实现客户端的负载均衡。这里使用Ribbon的默认负载均衡，**轮询服务注册清单**，分发Rest请求。

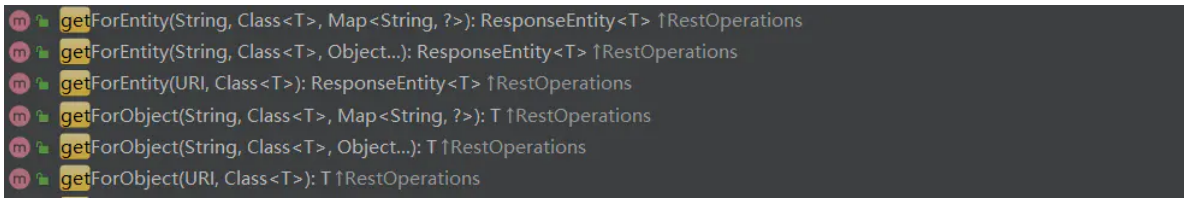
使用@Autowired注解将Spring容器中的RestTemplate对象注入进来

通过RestTemplate对象发送Rest请求给B服务，并获取响应结果

RestTemplate

RestTemplate:是一个REST请求的模板，封装了不同Rest请求类型,该对象会使用Ribbon的自动化配置，同时通过配置@LoadBalanced还能够开启客户端的负载均衡。下面介绍RestTemplate的不同请求类型

GET请求




```
RestTemplate restTemplate = new RestTemplate();

ResponseEntity<String> responseEntity = restTemplate .getForEntity(String
“http://HELLO-SERVICE/helllo/hello?name={1}”,String.class,"李四");

String body = responseEntity .getBody();
```

第一个参数：表示 Rest请求的URL 通常使用服务的服务名称，服务通过服务名称获取注册中心上该服务名称下的所有实例，在服务中心中保存着各个实例的元数据包括IP和端口号，这样就可以组成一个url，然后实行负载均衡。

第二个参数：表示Rest请求的响应返回值类型，上面这个例子表示返回值是String类型。当然也可以是一个实体类

第三个参数：便是URL中映射的参数值 可以使使用Map<参数名：参数值> 如果没有可以不填

> getForEntity方法返回的是整个Rest请求响应的结果 包含响应的body 当然还有一些其他的参数如响应状态，等信息。

getForObject()

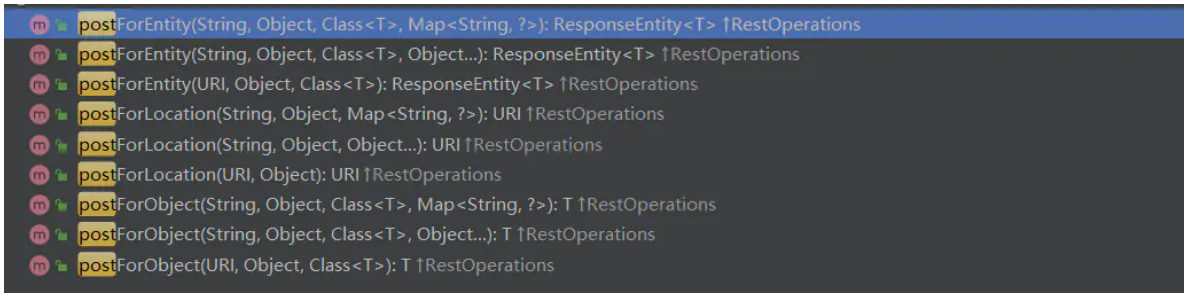
对于只要求获取响应body的情况可以直接使用getForObject()方法，该方法在getForObject()方法的基础上封装，直接返回响应body值

```
RestTemplate restTemplate = new RestTemplate();

String body = restTemplate .getForObject(String “http://HELLO-
SERVICE/helllo/hello?name={1}”,String.class,"李四");
```

POST请求

post请求和get请求类似



多了一个postForLocation()方法，该方法实现以POST请求提交资源，并返回新资源的URL

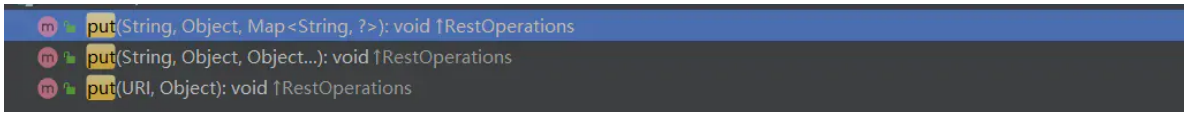
User user = new User("李四"，33);

URI responseUrl = restTemplate.postForLocation(url，user);

该方法不需要指定返回值类型，因为返回值类型指定为URI

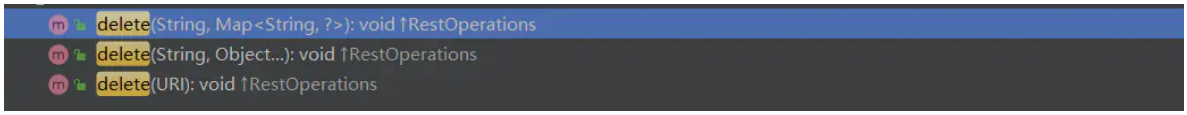
PUT请求

put请求和get请求类似



DELETE请求

delete请求和get请求类似



总结：

在微服务架构中使用客户端负载均衡调用是需要两个步骤

1：服务提供者只需要启动多个服务实例并注册到一个注册中心或者多个关联的注册中心

2：服务消费者直接通过调用被@LoadBalanced注解修饰过的RestTemplate对象实现面向服务的接口调用

Feign的目标

feign是声明式的web service客户端，它让微服务之间的调用变得更简单了，类似controller调用service。Spring Cloud集成了Ribbon和Eureka，可在使用Feign时提供负载均衡的http客户端。

引入Feign

因为feign底层是使用了ribbon作为负载均衡的客户端，而ribbon的负载均衡也是依赖于eureka 获得各个服务的地址，所以要引入eureka-client。

SpringbootApplication启动类加上@FeignClient注解，以及@EnableDiscoveryClient。

```
@EnableFeignClients
@EnableDiscoveryClient
@SpringBootApplication
public class ProductApplication {

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

yaml配置：

```
server:
  port: 8082

#配置eureka
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka
  instance:
    status-page-url-path: /info
    health-check-url-path: /health

#服务名称
spring:
  application:
    name: product
  profiles:
    active: ${boot.profile:dev}
#feign的配置，连接超时及读取超时配置
feign:
  client:
    config:
      default:
        connectTimeout: 5000
        readTimeout: 5000
        loggerLevel: basic
```

Feign的使用

```
@FeignClient(value = "CART")
public interface CartFeignClient {

    @PostMapping("/cart/{productId}")
    Long addCart(@PathVariable("productId")Long productId);
}
```

上面是最简单的feign client的使用，声明完为feign client后，其他spring管理的类，如service就可以直接注入使用了，例如：

```
//这里直接注入feign client
@Autowired
private CartFeignClient cartFeignClient;

@PostMapping("/toCart/{productId}")
public ResponseEntity addCart(@PathVariable("productId") Long productId){
    Long result = cartFeignClient.addCart(productId);
    return ResponseEntity.ok(result);
}
```

可以看到，使用feign之后，我们调用eureka 注册的其他服务，在代码中就像各个service之间相互调用那么简单。

FeignClient注解的一些属性

属性名	默认值	作用	备注
value	空字符串	调用服务名称，和name属性相同	
serviceId	空字符串	服务id，作用和name属性相同	已过期
name	空字符串	调用服务名称，和value属性相同	
url	空字符串	全路径地址或hostname，http或https可选	
decode404	false	配置响应状态码为404时是否应该抛出FeignExceptions	

属性名	默认值	作用	备注
configuration	{}	自定义当前feign client的一些配置	参考FeignClientsConfiguration
fallback	void.class	熔断机制，调用失败时，走的一些回退方法，可以用来抛出异常或给出默认返回数据。	底层依赖hystrix，启动类要加上@EnableHystrix
path	空字符串	自动给所有方法的requestMapping前加上前缀，类似与controller类上的requestMapping	
primary	true		

此外，还有qualifier及fallbackFactory，这里就不再赘述。

Feign自定义处理返回的异常

这里贴上GitHub上openFeign的wiki给出的自定义errorDecoder例子。

```
public class StashErrorDecoder implements ErrorDecoder {

    @Override
    public Exception decode(String methodKey, Response response) {
        if (response.status() >= 400 && response.status() <= 499) {
            //这里是给出的自定义异常
            return new StashClientException(
                    response.status(),
                    response.reason()
            );
        }
        if (response.status() >= 500 && response.status() <= 599) {
            //这里是给出的自定义异常
            return new StashServerException(
                    response.status(),
                    response.reason()
            );
        }
        //这里是其他状态码处理方法
        return errorStatus(methodKey, response);
    }
}
```

自定义好异常处理类后，要在@Configuration修饰的配置类中声明此类。

Feign使用OKhttp发送request

Feign底层默认是使用jdk中的HttpURLConnection发送HTTP请求，feign也提供了OKhttp来发送请求，具体配置如下：

```yaml
feign:
  client:
    config:
      default:
        connectTimeout: 5000
        readTimeout: 5000
        loggerLevel: basic
  okhttp:
    enabled: true
  hystrix:
    enabled: true
```

Feign原理简述

- 启动时，程序会进行包扫描，扫描所有包下所有@FeignClient注解的类，并将这些类注入到spring的IOC容器中。当定义的Feign中的接口被调用时，通过JDK的动态代理来生成RequestTemplate。
- RequestTemplate中包含请求的所有信息，如请求参数，请求URL等。
- RequestTemplate声场Request，然后将Request交给client处理，这个client默认是JDK的HTTPUrlConnection，也可以是OKhttp、Apache的HTTPClient等。
- 最后client封装成LoadBaLanceClient，结合ribbon负载均衡地发起调用。

详细原理请参考源码解析。

Feign、hystrix与retry的关系请参考https://xli1224.github.io/2017/09/22/configure-feign/

Feign开启GZIP压缩

Spring Cloud Feign支持对请求和响应进行GZIP压缩，以提高通信效率。

application.yml配置信息如下：

```yaml
feign:
  compression:
    request: #请求
      enabled: true #开启
      mime-types: text/xml,application/xml,application/json #开启支持压缩的MIME
TYPE
      min-request-size: 2048 #配置压缩数据大小的下限
    response: #响应
      enabled: true #开启响应GZIP压缩
```

注意：

由于开启GZIP压缩之后，Feign之间的调用数据通过二进制协议进行传输，返回值需要修改为ResponseEntity<byte[]>才可以正常显示，否则会导致服务之间的调用乱码。

示例如下：

```java
@PostMapping("/order/{productId}")
ResponseEntity<byte[]> addCart(@PathVariable("productId") Long productId);
```

作用在所有Feign Client上的配置方式

方式一：通过java bean 的方式指定。

@EnableFeignClients注解上有个defaultConfiguration属性，可以指定默认Feign Client的一些配置。

```
@EnableFeignClients(defaultConfiguration = DefaultFeignConfiguration.class)
@EnableDiscoveryClient
@SpringBootApplication
@EnableCircuitBreaker
public class ProductApplication {

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

DefaultFeignConfiguration内容：

```
@Configuration
public class DefaultFeignConfiguration {

    @Bean
    public Retryer feignRetryer() {
        return new Retryer.Default(1000,3000,3);
    }
}
```

方式二：通过配置文件方式指定。

```
feign:
  client:
    config:
      default:
        connectTimeout: 5000 #连接超时
        readTimeout: 5000 #读取超时
        loggerLevel: basic #日志等级
```

Feign Client开启日志

日志配置和上述配置相同，也有两种方式。

方式一：通过java bean的方式指定

```
@Configuration
public class DefaultFeignConfiguration {
    @Bean
    public Logger.Level feignLoggerLevel(){
        return Logger.Level.BASIC;
    }
}
```

方式二：通过配置文件指定

```
logging:
  level:
    com.xt.open.jmall.product.remote.feignclients.CartFeignClient: debug
```

Feign 的GET的多参数传递

目前，feign不支持GET请求直接传递POJO对象的，目前解决方法如下：

1. 把POJO拆散城一个一个单独的属性放在方法参数中
2. 把方法参数编程Map传递
3. 使用GET传递@RequestBody，但此方式违反restful风格

介绍一个最佳实践，通过feign的拦截器来实现。

```
@Component
@Slf4j
public class FeignCustomRequestInteceptor implements RequestInterceptor {

    @Autowired
    private ObjectMapper objectMapper;

    @Override
    public void apply(RequestTemplate template) {
        if (HttpMethod.GET.toString() == template.method() && template.body() !=
null) {
            //feign 不支持GET方法传输POJO 转换成json，再换成query
            try {
                Map<String, Collection<String>> map =
objectMapper.readValue(template.bodyTemplate(), new TypeReference<Map<String,
Collection<String>>>() {

                });
                template.body(null);
                template.queries(map);
            } catch (IOException e) {
                log.error("cause exception", e);
            }
        }
    }
```

Feign解决了什么问题?

封装了Http调用流程，更适合面向接口化的变成习惯

在服务调用的场景中，我们经常调用基于Http协议的服务，而我们经常使用到的框架可能有HttpURLConnection、Apache HttpComponnets、OkHttp3 、Netty等等，这些框架在基于自身的专注点提供了自身特性。而从角色划分上来看，他们的职能是一致的提供Http调用服务。具体流程如下：

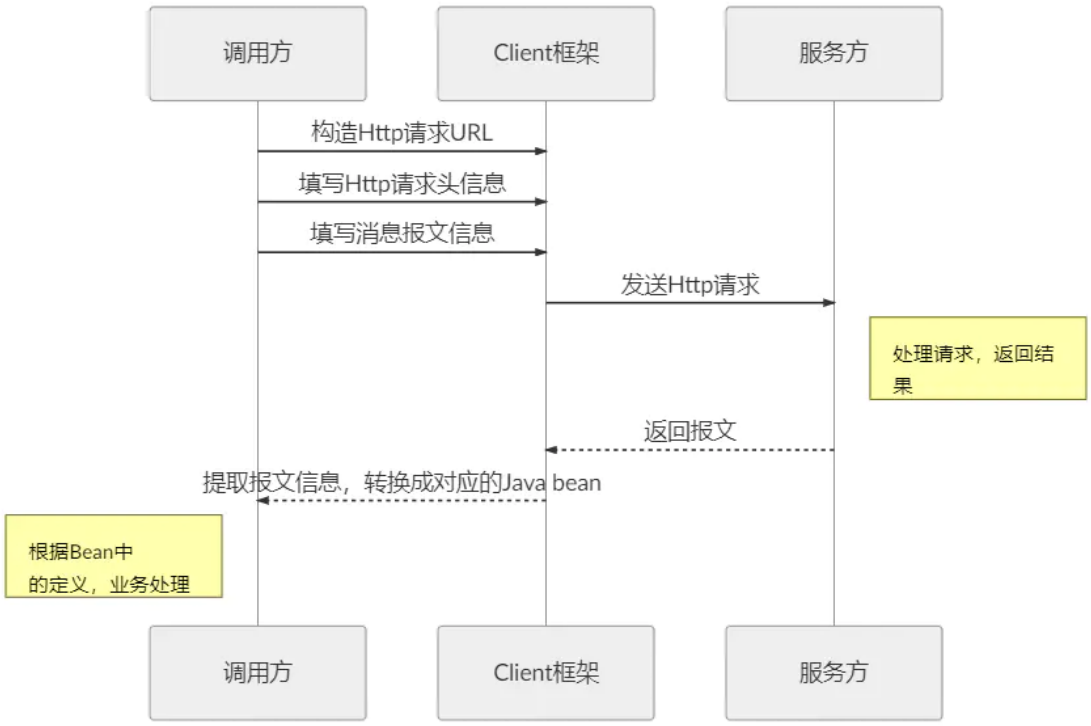


image.png

Feign是如何设计的?

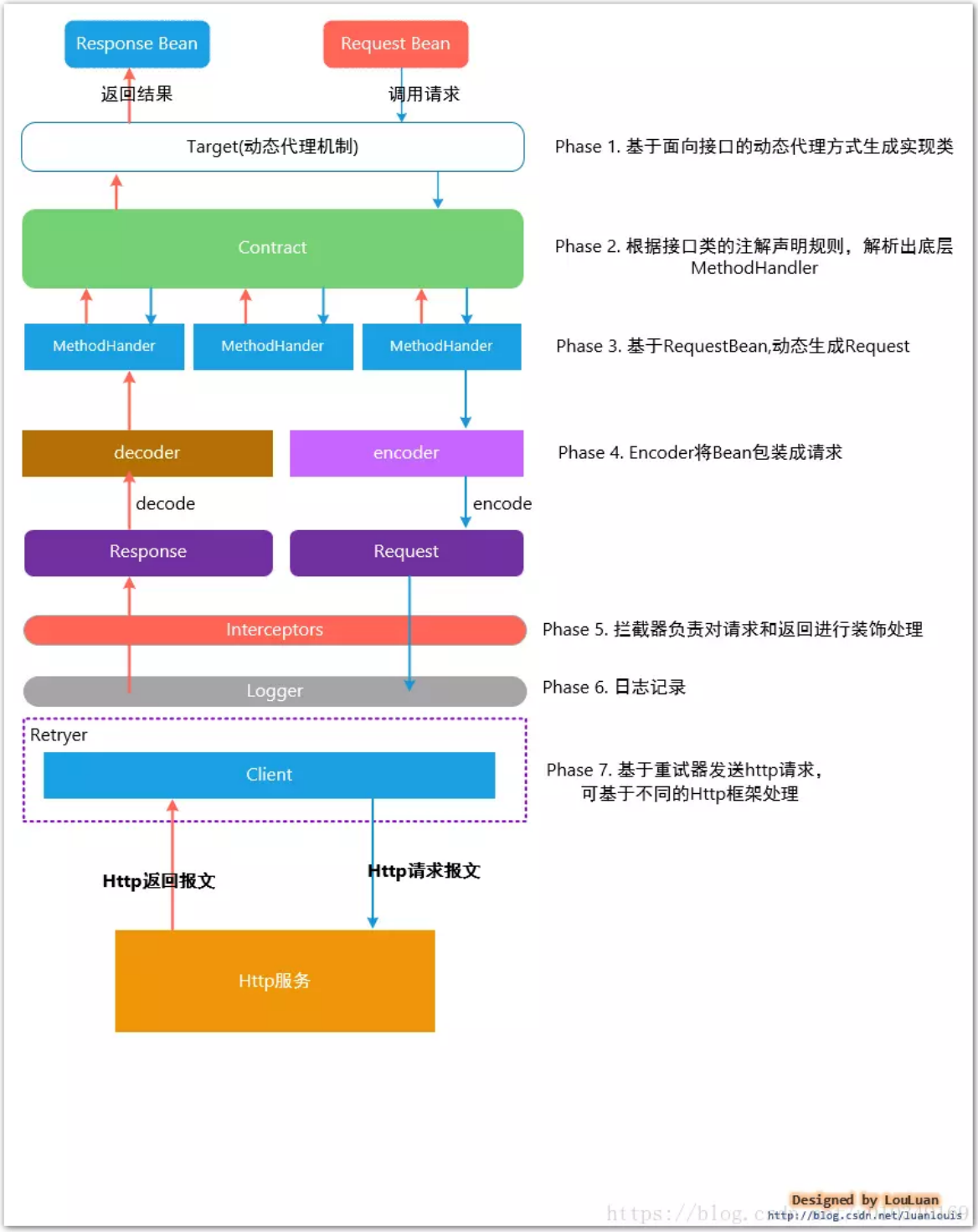


image.png

PHASE 1. 基于面向接口的动态代理方式生成实现类

在使用feign 时，会定义对应的接口类，在接口类上使用Http相关的注解，标识HTTP请求参数信息,如下所示:

```
interface GitHub {
  @RequestLine("GET /repos/{owner}/{repo}/contributors")
  List<Contributor> contributors(@Param("owner") String owner, @Param("repo")
String repo);
}

public static class Contributor {
```

```
  String login;
  int contributions;
}

public class MyApp {
  public static void main(String... args) {
    GitHub github = Feign.builder()
                         .decoder(new GsonDecoder())
                         .target(GitHub.class, "https://api.github.com");

    // Fetch and print a list of the contributors to this library.
    List<Contributor> contributors = github.contributors("OpenFeign", "feign");
    for (Contributor contributor : contributors) {
      System.out.println(contributor.login + " (" + contributor.contributions +
")");
    }
  }
}
```

在Feign 底层，通过基于面向接口的动态代理方式生成实现类，将请求调用委托到动态代理实现类，基本原理如下所示：

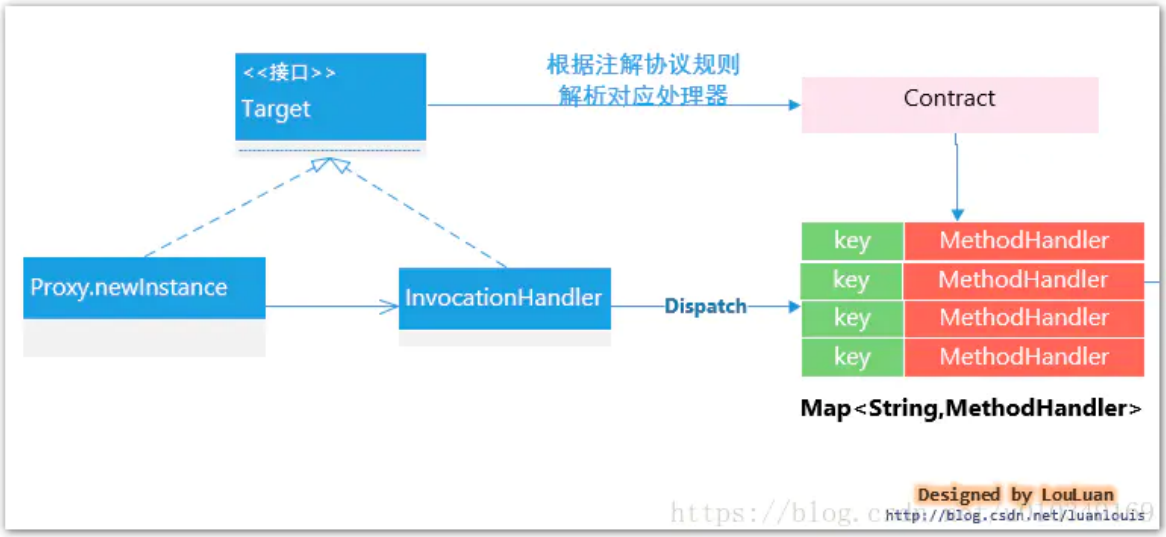


image.png

```
 public class ReflectiveFeign extends Feign{
  ///省略部分代码
  @Override
  public <T> T newInstance(Target<T> target) {
    //根据接口类和Contract协议解析方式，解析接口类上的方法和注解，转换成内部的MethodHandler
处理方式
    Map<String, MethodHandler> nameToHandler =
targetToHandlersByName.apply(target);
    Map<Method, MethodHandler> methodToHandler = new LinkedHashMap<Method,
MethodHandler>();
    List<DefaultMethodHandler> defaultMethodHandlers = new
LinkedList<DefaultMethodHandler>();

    for (Method method : target.type().getMethods()) {
```

```
      if (method.getDeclaringClass() == Object.class) {
        continue;
      } else if(Util.isDefault(method)) {
        DefaultMethodHandler handler = new DefaultMethodHandler(method);
        defaultMethodHandlers.add(handler);
        methodToHandler.put(method, handler);
      } else {
        methodToHandler.put(method,
nameToHandler.get(Feign.configKey(target.type(), method)));
      }
    }
    InvocationHandler handler = factory.create(target, methodToHandler);
    // 基于Proxy.newProxyInstance 为接口类创建动态实现，将所有的请求转换给
InvocationHandler 处理。
    T proxy = (T) Proxy.newProxyInstance(target.type().getClassLoader(), new
Class<?>[]{target.type()}, handler);

    for(DefaultMethodHandler defaultMethodHandler : defaultMethodHandlers) {
      defaultMethodHandler.bindTo(proxy);
    }
    return proxy;
  }
  //省略部分代码
```

PHASE 2. 根据Contract协议规则，解析接口类的注解信息，解析成内部表现：

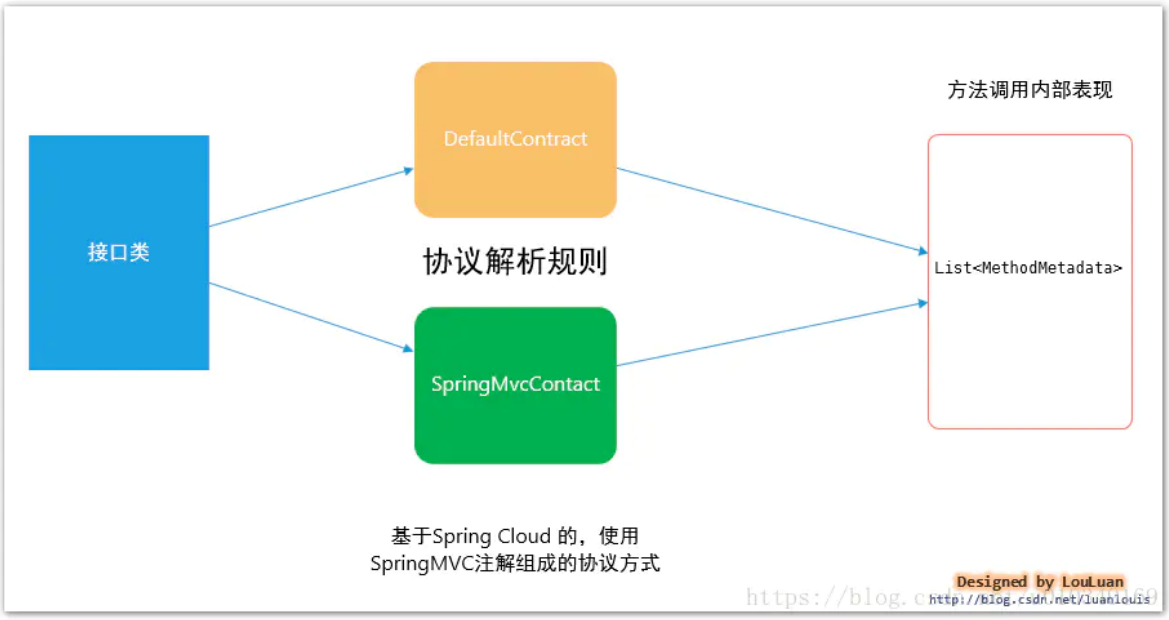


image.png

Feign 定义了转换协议，定义如下：

```
/**
 * Defines what annotations and values are valid on interfaces.
 */
public interface Contract {

  /**
   * Called to parse the methods in the class that are linked to HTTP requests.
   * 传入接口定义，解析成相应的方法内部元数据表示
   * @param targetType {@link feign.Target#type() type} of the Feign interface.
   */
  // TODO: break this and correct spelling at some point
  List<MethodMetadata> parseAndValidatateMetadata(Class<?> targetType);
}
```

默认Contract 实现

Feign 默认有一套自己的协议规范，规定了一些注解，可以映射成对应的Http请求，如官方的一个例子：

```
public interface GitHub {

  @RequestLine("GET /repos/{owner}/{repo}/contributors")
  List<Contributor> getContributors(@Param("owner") String owner, @Param("repo")
String repository);

  class Contributor {
    String login;
    int contributions;
  }
}
```

上述的例子中，尝试调用GitHub.getContributors("foo","myrepo")的的时候，会转换成如下的HTTP请求：

```
GET /repos/foo/myrepo/contributors
HOST XXXX.XXX.XXX
```

Feign 默认的协议规范

注解	接口Target	使用说明
`@RequestLine`	方法上	定义HttpMethod 和 UriTemplate. UriTemplate 中使用`{}` 包裹的表达式，可以通过在方法参数上使用@Param 自动注入
`@Param`	方法参数	定义模板变量，模板变量的值可以使用名称的方式使用模板注入解析
`@Headers`	类上或者方法上	定义头部模板变量，使用@Param 注解提供参数值的注入。如果该注解添加在接口类上，则所有的请求都会携带对应的Header信息；如果在方法上，则只会添加到对应的方法请求上
`@QueryMap`	方法上	定义一个键值对或者 pojo，参数值将会被转换成URL上的 query 字符串上
`@HeaderMap`	方法上	定义一个HeaderMap, 与 UrlTemplate 和HeaderTemplate 类型，可以使用@Param 注解提供参数值

具体FeignContract 是如何解析的，不在本文的介绍范围内，详情请参考代码：
https://github.com/OpenFeign/feign/blob/master/core/src/main/java/feign/Contract.java

基于Spring MVC的协议规范SpringMvcContract:

当前Spring Cloud 微服务解决方案中，为了降低学习成本，采用了Spring MVC的部分注解来完成 请求协议解析，也就是说 ，写客户端请求接口和像写服务端代码一样：客户端和服务端可以通过SDK的方式进行约定，客户端只需要引入服务端发布的SDK API，就可以使用面向接口的编码方式对接服务：

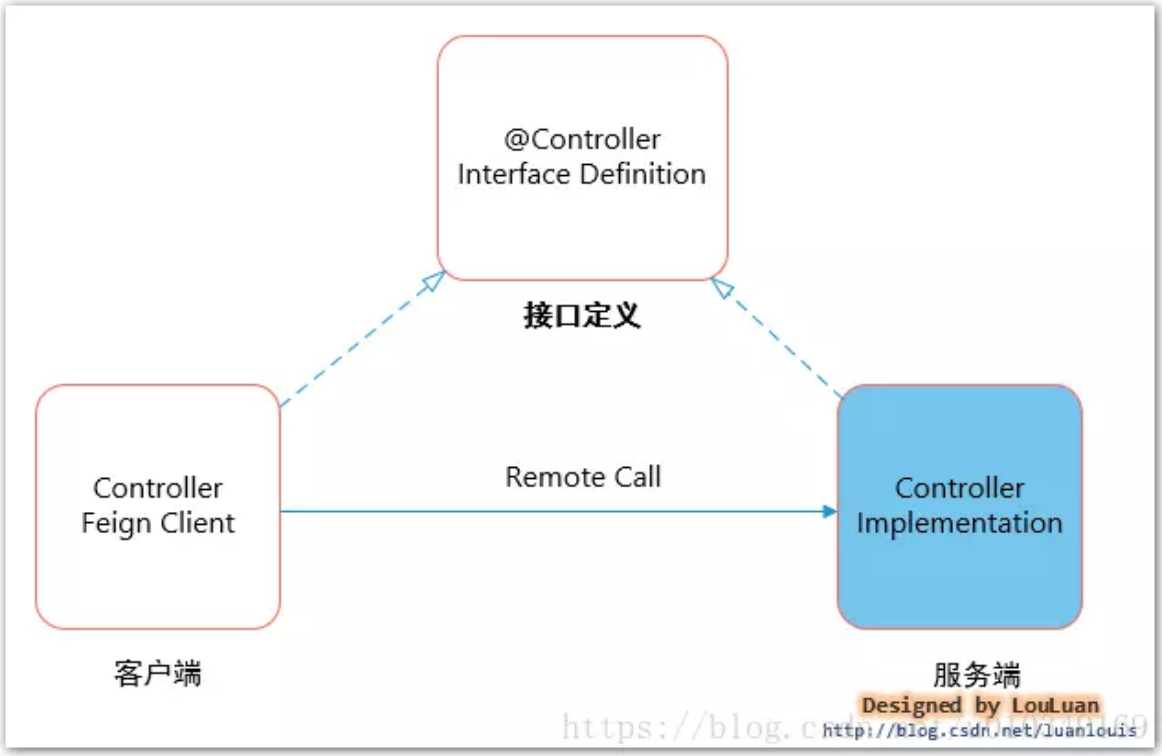


image.png

> 我们团队内部就是按照这种思路，结合Spring Boot Starter 的特性，定义了服务端starter，服务消费者在使用的时候，只需要引入Starter，就可以调用服务。这个比较适合平台无关性，接口抽象出来的好处就是可以根据服务调用实现方式自有切换：

> 1. 可以基于简单的Http服务调用；
> 2. 可以基于Spring Cloud 微服务架构调用；
> 3. 可以基于Dubbo SOA服务治理
>
> 这种模式比较适合在SaSS混合软件服务的模式下自有切换，根据客户的硬件能力选择合适的方式部署，也可以基于自身的服务集群部署微服务

至于Spring Cloud 是如何实现 协议解析的，可参考代码：
https://github.com/spring-cloud/spring-cloud-openfeign/blob/master/spring-cloud-openfeign-core/src/main/java/org/springframework/cloud/openfeign/support/SpringMvcContract.java

当然，目前的Spring MVC的注解并不是可以完全使用的，有一些注解并不支持,如`@GetMapping`,`@PutMapping`等，仅支持使用`@RequestMapping`等，另外注解继承性方面也有些问题；具体限制细节，每个版本能会有些出入，可以参考上述的代码实现，比较简单。

> Spring Cloud 没有基于Spring MVC 全部注解来做Feign 客户端注解协议解析，个人认为这个是一个不小的坑。在刚入手Spring Cloud 的时候，就碰到这个问题。后来是深入代码才解决的.... 这个应该有人写了增强类来处理，暂且不表，先MARK一下，是一个开源代码练手的好机会。

PHASE 3. 基于 RequestBean，动态生成Request

根据传入的Bean对象和注解信息，从中提取出相应的值，来构造Http Request 对象：

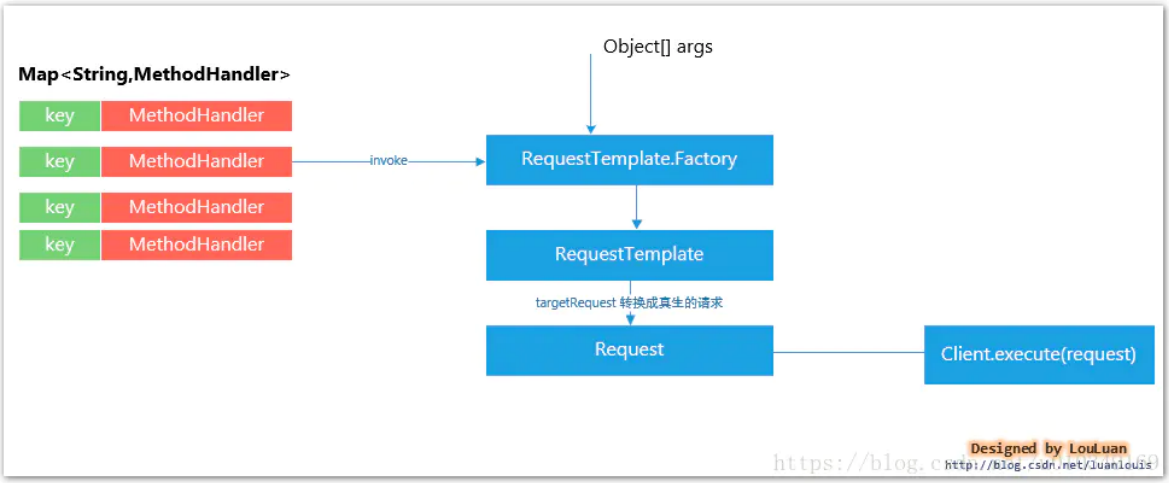


image.png

PHASE 4. 使用Encoder 将Bean转换成 Http报文正文（消息解析和转码逻辑）

Feign 最终会将请求转换成Http 消息发送出去，传入的请求对象最终会解析成消息体，如下所示：

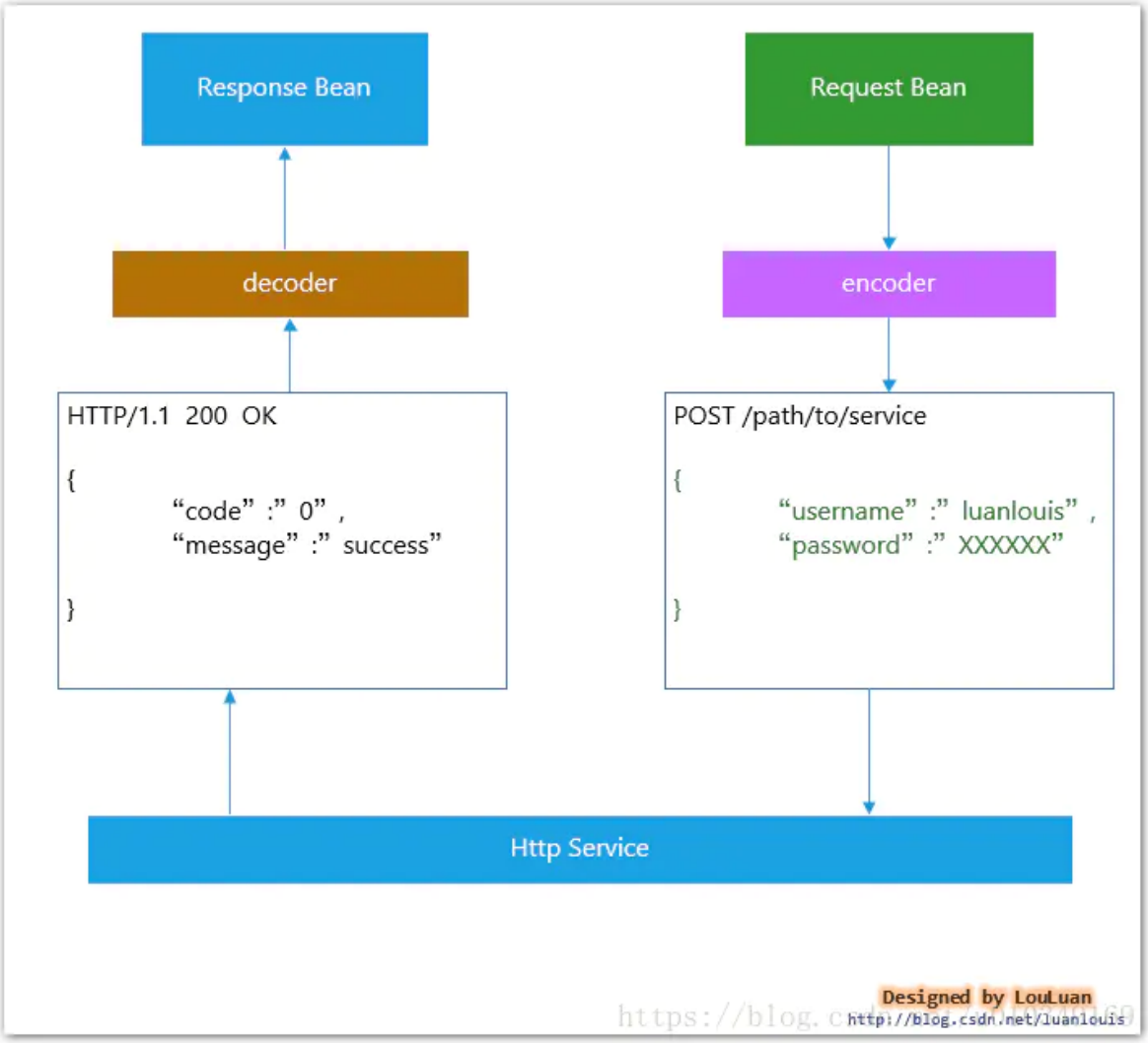


image.png

在接口定义上Feign做的比较简单，抽象出了Encoder 和decoder 接口：

```
public interface Encoder {
  /** Type literal for {@code Map<String, ?>}, indicating the object to encode
is a form. */
  Type MAP_STRING_WILDCARD = Util.MAP_STRING_WILDCARD;

  /**
   * Converts objects to an appropriate representation in the template.
   *  将实体对象转换成Http请求的消息正文中
   * @param object   what to encode as the request body.
   * @param bodyType the type the object should be encoded as. {@link
#MAP_STRING_WILDCARD}
   *                 indicates form encoding.
   * @param template the request template to populate.
   * @throws EncodeException when encoding failed due to a checked exception.
   */
  void encode(Object object, Type bodyType, RequestTemplate template) throws
EncodeException;

  /**
   * Default implementation of {@code Encoder}.
   */
```

```
  class Default implements Encoder {

    @Override
    public void encode(Object object, Type bodyType, RequestTemplate template) {
      if (bodyType == String.class) {
        template.body(object.toString());
      } else if (bodyType == byte[].class) {
        template.body((byte[]) object, null);
      } else if (object != null) {
        throw new EncodeException(
            format("%s is not a type supported by this encoder.",
object.getClass()));
      }
    }
  }
}
```

```
public interface Decoder {

  /**
   * Decodes an http response into an object corresponding to its {@link
   * java.lang.reflect.Method#getGenericReturnType() generic return type}. If
you need to wrap
   * exceptions, please do so via {@link DecodeException}.
   *  从Response 中提取Http消息正文，通过接口类声明的返回类型，消息自动装配
   * @param response the response to decode
   * @param type     {@link java.lang.reflect.Method#getGenericReturnType()
generic return type} of
   *                 the method corresponding to this {@code response}.
   * @return instance of {@code type}
   * @throws IOException     will be propagated safely to the caller.
   * @throws DecodeException when decoding failed due to a checked exception
besides IOException.
   * @throws FeignException  when decoding succeeds, but conveys the operation
failed.
   */
  Object decode(Response response, Type type) throws IOException,
DecodeException, FeignException;

  /** Default implementation of {@code Decoder}. */
  public class Default extends StringDecoder {

    @Override
    public Object decode(Response response, Type type) throws IOException {
      if (response.status() == 404) return Util.emptyValueOf(type);
      if (response.body() == null) return null;
      if (byte[].class.equals(type)) {
        return Util.toByteArray(response.body().asInputStream());
      }
      return super.decode(response, type);
    }
  }
}
```

目前Feign 有以下实现：

Encoder/ Decoder 实现	说明
JacksonEncoder，JacksonDecoder	基于 Jackson 格式的持久化转换协议
GsonEncoder，GsonDecoder	基于Google GSON 格式的持久化转换协议
SaxEncoder，SaxDecoder	基于XML 格式的Sax 库持久化转换协议
JAXBEncoder，JAXBDecoder	基于XML 格式的JAXB 库持久化转换协议
ResponseEntityEncoder，ResponseEntityDecoder	Spring MVC 基于 ResponseEntity< T > 返回格式的转换协议
SpringEncoder，SpringDecoder	基于Spring MVC HttpMessageConverters 一套机制实现的转换协议 ,应用于Spring Cloud 体系中

PHASE 5. 拦截器负责对请求和返回进行装饰处理

在请求转换的过程中，Feign 抽象出来了拦截器接口，用于用户自定义对请求的操作：

```
public interface RequestInterceptor {

  /**
   * 可以在构造RequestTemplate 请求时，增加或者修改Header, Method, Body 等信息
   * Called for every request. Add data using methods on the supplied {@link
RequestTemplate}.
   */
  void apply(RequestTemplate template);
}
```

比如，如果希望Http消息传递过程中被压缩，可以定义一个请求拦截器：

```
public class FeignAcceptGzipEncodingInterceptor extends BaseRequestInterceptor {

    /**
     * Creates new instance of {@link FeignAcceptGzipEncodingInterceptor}.
     *
     * @param properties the encoding properties
     */
    protected FeignAcceptGzipEncodingInterceptor(FeignClientEncodingProperties
properties) {
        super(properties);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void apply(RequestTemplate template) {
        //  在Header 头部添加相应的数据信息
        addHeader(template, HttpEncoding.ACCEPT_ENCODING_HEADER,
HttpEncoding.GZIP_ENCODING,
```

```
                HttpEncoding.DEFLATE_ENCODING);
    }
}
```

PHASE 6. 日志记录

在发送和接收请求的时候，Feign定义了统一的日志门面来输出日志信息，并且将日志的输出定义了四个等级：

级别	说明
NONE	不做任何记录
BASIC	只记录输出Http 方法名称、请求URL、返回状态码和执行时间
HEADERS	记录输出Http 方法名称、请求URL、返回状态码和执行时间 和 Header 信息
FULL	记录Request 和Response的Header，Body和一些请求元数据

```
public abstract class Logger {

  protected static String methodTag(String configKey) {
    return new StringBuilder().append('[').append(configKey.substring(0,
configKey.indexOf('(')))
        .append("] ").toString();
  }

  /**
   * Override to log requests and responses using your own implementation.
Messages will be http
   * request and response text.
   *
   * @param configKey value of {@link Feign#configKey(Class,
java.lang.reflect.Method)}
   * @param format    {@link java.util.Formatter format string}
   * @param args      arguments applied to {@code format}
   */
  protected abstract void log(String configKey, String format, Object... args);

  protected void logRequest(String configKey, Level logLevel, Request request) {
    log(configKey, "---> %s %s HTTP/1.1", request.method(), request.url());
    if (logLevel.ordinal() >= Level.HEADERS.ordinal()) {

      for (String field : request.headers().keySet()) {
        for (String value : valuesOrEmpty(request.headers(), field)) {
          log(configKey, "%s: %s", field, value);
        }
      }

      int bodyLength = 0;
      if (request.body() != null) {
        bodyLength = request.body().length;
        if (logLevel.ordinal() >= Level.FULL.ordinal()) {
          String
              bodyText =
```

```
              request.charset() != null ? new String(request.body(),
request.charset()) : null;
          log(configKey, ""); // CRLF
          log(configKey, "%s", bodyText != null ? bodyText : "Binary data");
        }
      }
      log(configKey, "---> END HTTP (%s-byte body)", bodyLength);
    }
  }

  protected void logRetry(String configKey, Level logLevel) {
    log(configKey, "---> RETRYING");
  }

  protected Response logAndRebufferResponse(String configKey, Level logLevel,
Response response,
                                            long elapsedTime) throws IOException
{
    String reason = response.reason() != null && logLevel.compareTo(Level.NONE)
> 0 ?
        " " + response.reason() : "";
    int status = response.status();
    log(configKey, "<--- HTTP/1.1 %s%s (%sms)", status, reason, elapsedTime);
    if (logLevel.ordinal() >= Level.HEADERS.ordinal()) {

      for (String field : response.headers().keySet()) {
        for (String value : valuesOrEmpty(response.headers(), field)) {
          log(configKey, "%s: %s", field, value);
        }
      }

      int bodyLength = 0;
      if (response.body() != null && !(status == 204 || status == 205)) {
        // HTTP 204 No Content "...response MUST NOT include a message-body"
        // HTTP 205 Reset Content "...response MUST NOT include an entity"
        if (logLevel.ordinal() >= Level.FULL.ordinal()) {
          log(configKey, ""); // CRLF
        }
        byte[] bodyData = Util.toByteArray(response.body().asInputStream());
        bodyLength = bodyData.length;
        if (logLevel.ordinal() >= Level.FULL.ordinal() && bodyLength > 0) {
          log(configKey, "%s", decodeOrDefault(bodyData, UTF_8, "Binary data"));
        }
        log(configKey, "<--- END HTTP (%s-byte body)", bodyLength);
        return response.toBuilder().body(bodyData).build();
      } else {
        log(configKey, "<--- END HTTP (%s-byte body)", bodyLength);
      }
    }
    return response;
  }

  protected IOException logIOException(String configKey, Level logLevel,
IOException ioe, long elapsedTime) {
    log(configKey, "<--- ERROR %s: %s (%sms)", ioe.getClass().getSimpleName(),
ioe.getMessage(),
        elapsedTime);
    if (logLevel.ordinal() >= Level.FULL.ordinal()) {
```

```
      StringWriter sw = new StringWriter();
      ioe.printStackTrace(new PrintWriter(sw));
      log(configKey, sw.toString());
      log(configKey, "<--- END ERROR");
    }
    return ioe;
  }
```

PHASE 7 . 基于重试器发送HTTP请求

Feign 内置了一个重试器，当HTTP请求出现IO异常时，Feign会有一个最大尝试次数发送请求，以下是Feign核心
代码逻辑:

```
final class SynchronousMethodHandler implements MethodHandler {

  // 省略部分代码

  @Override
  public Object invoke(Object[] argv) throws Throwable {
   //根据输入参数，构造Http 请求。
    RequestTemplate template = buildTemplateFromArgs.create(argv);
    // 克隆出一份重试器
    Retryer retryer = this.retryer.clone();
    // 尝试最大次数，如果中间有结果，直接返回
    while (true) {
      try {
        return executeAndDecode(template);
      } catch (RetryableException e) {
        retryer.continueOrPropagate(e);
        if (logLevel != Logger.Level.NONE) {
          logger.logRetry(metadata.configKey(), logLevel);
        }
        continue;
      }
    }
  }
```

重试器有如下几个控制参数:

重试参数	说明	默认值
period	初始重试时间间隔，当请求失败后，重试器将会暂停 初始时间间隔(线程 sleep 的方式)后再开始，避免强刷请求，浪费性能	100ms
maxPeriod	当请求连续失败时，重试的时间间隔将按照：`long interval = (long) (period * Math.pow(1.5, attempt - 1));` 计算，按照等比例方式延长，但是最大间隔时间为 maxPeriod, 设置此值能够避免 重试次数过多的情况下执行周期太长	1000ms
maxAttempts	最大重试次数	5

具体的代码实现可参考:
https://github.com/OpenFeign/feign/blob/master/core/src/main/java/feign/Retryer.java

PHASE 8. 发送Http请求

Feign 真正发送HTTP请求是委托给 `feign.Client` 来做的：

```
public interface Client {

  /**
   * Executes a request against its {@link Request#url() url} and returns a
response.
   *  执行Http请求，并返回Response
   * @param request safe to replay.
   * @param options options to apply to this request.
   * @return connected response, {@link Response.Body} is absent or unread.
   * @throws IOException on a network error connecting to {@link Request#url()}.
   */
  Response execute(Request request, Options options) throws IOException;
  }
```

Feign 默认底层通过JDK 的 `java.net.HttpURLConnection` 实现了 `feign.Client` 接口类,在每次发送请求的时候，都会创建新的HttpURLConnection 链接，这也就是为什么默认情况下Feign的性能很差的原因。可以通过拓展该接口，使用Apache HttpClient 或者OkHttp3等基于连接池的高性能Http客户端，我们项目内部使用的就是OkHttp3作为Http 客户端。

如下是Feign 的默认实现，供参考：

```
public static class Default implements Client {

    private final SSLSocketFactory sslContextFactory;
    private final HostnameVerifier hostnameVerifier;

    /**
     * Null parameters imply platform defaults.
     */
    public Default(SSLSocketFactory sslContextFactory, HostnameVerifier
hostnameVerifier) {
      this.sslContextFactory = sslContextFactory;
      this.hostnameVerifier = hostnameVerifier;
    }

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

    HttpURLConnection convertAndSend(Request request, Options options) throws
IOException {
      final HttpURLConnection
          connection =
          (HttpURLConnection) new URL(request.url()).openConnection();
      if (connection instanceof HttpsURLConnection) {
        HttpsURLConnection sslCon = (HttpsURLConnection) connection;
```

```
        if (sslContextFactory != null) {
          sslCon.setSSLSocketFactory(sslContextFactory);
        }
        if (hostnameVerifier != null) {
          sslCon.setHostnameVerifier(hostnameVerifier);
        }
      }
      connection.setConnectTimeout(options.connectTimeoutMillis());
      connection.setReadTimeout(options.readTimeoutMillis());
      connection.setAllowUserInteraction(false);
      connection.setInstanceFollowRedirects(true);
      connection.setRequestMethod(request.method());

      Collection<String> contentEncodingValues =
request.headers().get(CONTENT_ENCODING);
      boolean
          gzipEncodedRequest =
          contentEncodingValues != null &&
contentEncodingValues.contains(ENCODING_GZIP);
      boolean
          deflateEncodedRequest =
          contentEncodingValues != null &&
contentEncodingValues.contains(ENCODING_DEFLATE);

      boolean hasAcceptHeader = false;
      Integer contentLength = null;
      for (String field : request.headers().keySet()) {
        if (field.equalsIgnoreCase("Accept")) {
          hasAcceptHeader = true;
        }
        for (String value : request.headers().get(field)) {
          if (field.equals(CONTENT_LENGTH)) {
            if (!gzipEncodedRequest && !deflateEncodedRequest) {
              contentLength = Integer.valueOf(value);
              connection.addRequestProperty(field, value);
            }
          } else {
            connection.addRequestProperty(field, value);
          }
        }
      }
      // Some servers choke on the default accept string.
      if (!hasAcceptHeader) {
        connection.addRequestProperty("Accept", "*/*");
      }

      if (request.body() != null) {
        if (contentLength != null) {
          connection.setFixedLengthStreamingMode(contentLength);
        } else {
          connection.setChunkedStreamingMode(8196);
        }
        connection.setDoOutput(true);
        OutputStream out = connection.getOutputStream();
        if (gzipEncodedRequest) {
          out = new GZIPOutputStream(out);
        } else if (deflateEncodedRequest) {
          out = new DeflaterOutputStream(out);
```

```
        }
        try {
          out.write(request.body());
        } finally {
          try {
            out.close();
          } catch (IOException suppressed) { // NOPMD
          }
        }
      }
      return connection;
    }

    Response convertResponse(HttpURLConnection connection) throws IOException {
      int status = connection.getResponseCode();
      String reason = connection.getResponseMessage();

      if (status < 0) {
        throw new IOException(format("Invalid status(%s) executing %s %s",
status,
            connection.getRequestMethod(), connection.getURL()));
      }

      Map<String, Collection<String>> headers = new LinkedHashMap<String,
Collection<String>>();
      for (Map.Entry<String, List<String>> field :
connection.getHeaderFields().entrySet()) {
        // response message
        if (field.getKey() != null) {
          headers.put(field.getKey(), field.getValue());
        }
      }

      Integer length = connection.getContentLength();
      if (length == -1) {
        length = null;
      }
      InputStream stream;
      if (status >= 400) {
        stream = connection.getErrorStream();
      } else {
        stream = connection.getInputStream();
      }
      return Response.builder()
              .status(status)
              .reason(reason)
              .headers(headers)
              .body(stream, length)
              .build();
    }
  }
```

Feign 的性能怎么样?

Feign 整体框架非常小巧，在处理请求转换和消息解析的过程中，基本上没什么时间消耗。真正影响性能的，是处理Http请求的环节。

如上所述，由于默认情况下，Feign采用的是JDK的`HttpURLConnection`,所以整体性能并不高，刚开始接触Spring Cloud 的同学，如果没注意这些细节，可能会对Spring Cloud 有很大的偏见。

我们项目内部使用的是OkHttp3 作为连接客户端。

```
// 有一个数据文件，csv格式，记录了用户的登录时间，格式如下
//**
// user_id, time
// u1, 2019-02-10 00:00:00
// u2, 2019-02-11 00:00:00
// u1, 2019-02-12 00:00:00
// u1, 2019-02-11 00:00:00
// u1, 2019-02-14 00:00:00
//  ...
// **/

// 写一个程序处理，求得每个用户最近3个月内，最长的连续登录天数
```