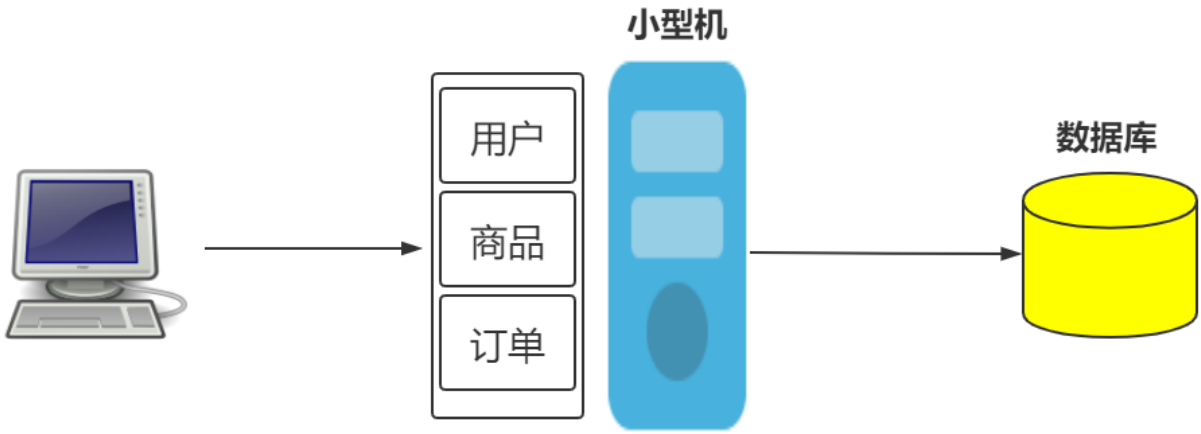
《Jack-Spring Cloud从入门到入魔》

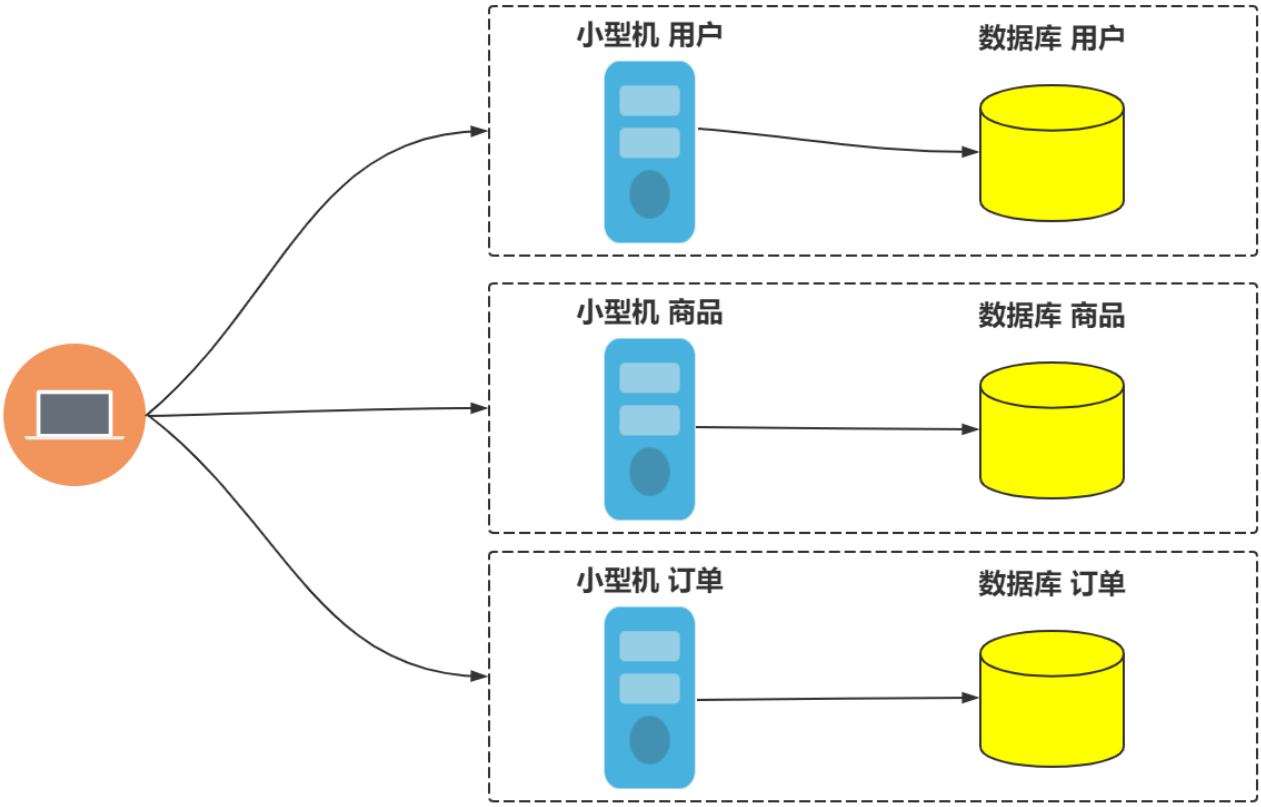
01 架构演进

1.1 单机小型机

- 1969年，阿帕网诞生，最初是为了军事目的，后来衍变成Internet
- 2000年左右，互联网在中国开始盛行起来，但是那时候网民数较少，所以多数企业服务单一，采用集中式部署的方式就能满足需求



- 一旦小型机或者数据库出现问题，会导致整个系统的故障，而且功能修改发布也不方便，所以不妨把大系统拆分成多个子系统，比如“用户”，“商品”，“论坛”等，也就是”垂直拆分“，并且每个子系统都有各自的硬件

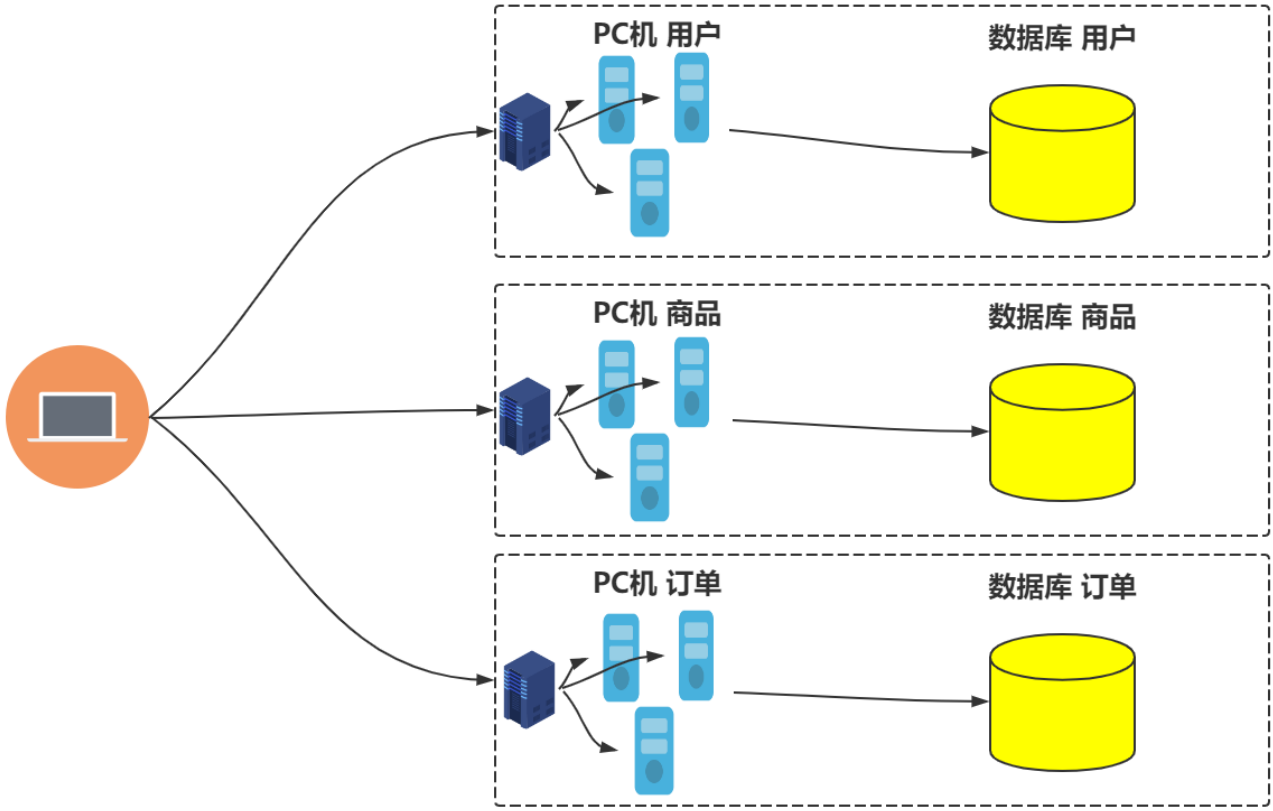


1.2 集群化负载均衡

- 用户量越来越大，就意味着需要更多的小型机，价格昂贵，操作维护成本高，不妨把小型机换成普通的PC机，采用多台PC机部署同一个应用的方案，但是此时就需要对这些应用做负载均衡，因为客户端不知道请求哪一个。

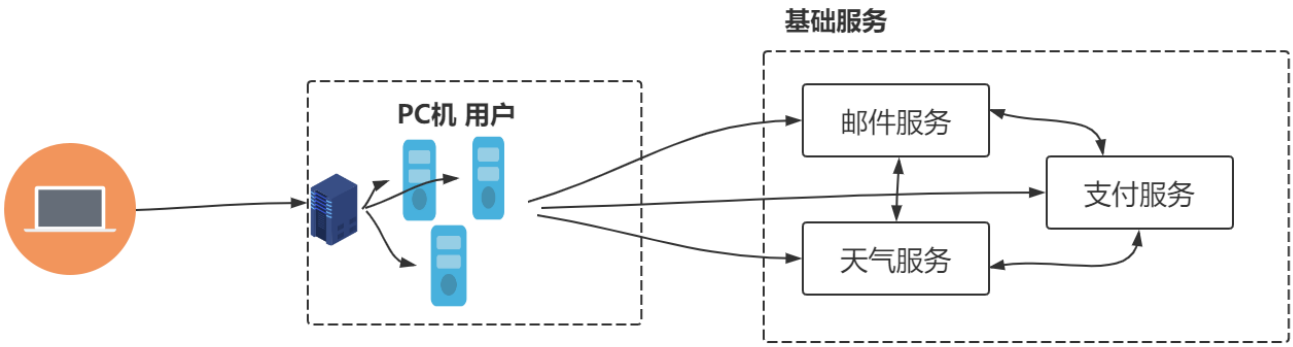
> 2013年5月17日，阿里集团最后一台IBM小型机在支付宝下线。
>
> 这是自2009年“去IOE”战略透露以来，“去IOE”非常重要的一个节点。“去IOE”指的是摆脱掉IT部署中原有的IBM小型机、Oracle数据库以及EMC存储的过度依赖。告别最后一台小机，意味着整个阿里集团尽管还有一些Oracle数据库和EMC存储，但是IBM小型机已全部消失。

- 负载均衡可以分为硬件和软件，硬件比如F5，软件比如LVS(Linux Virtual Server)。负载均衡的思路是对外暴露一个统一的接口，然后根据用户的请求进行对应规则的转发。同时负载均衡还可以做健康检查、限流等
- 有了负载均衡，后端的应用就可以根据流量的大小进行动态的扩缩容了，也就是“水平扩展”



1.3 服务化(SOA)

- 此时发现在用户，商品和订单等中有重复的功能，比如登录、注册、邮件等。一旦项目大了，集群部署多了，这些重复的功能无疑是浪费资源，不妨把重复的功能抽取出来，起个名字叫“服务Service”，我觉得挺好的



- 其实对于“服务”现在已经比较广泛了，比如“基础设施即服务IaaS”、“平台即服务PaaS”、“软件即服务SaaS“等

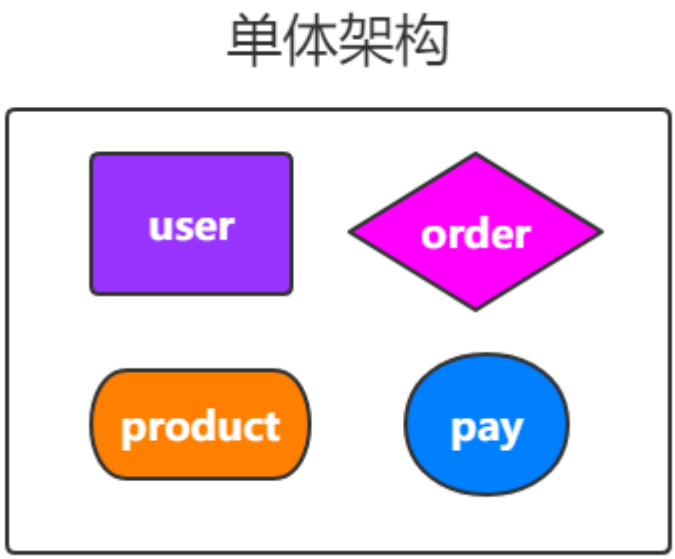
1.4 微服务(Microservices)

> Martin Folwer：https://www.martinfowler.com/articles/microservices.html

```
In short, the microservice architectural style [1] is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.
```

02 单体架构VS微服务架构

2.1 单体架构



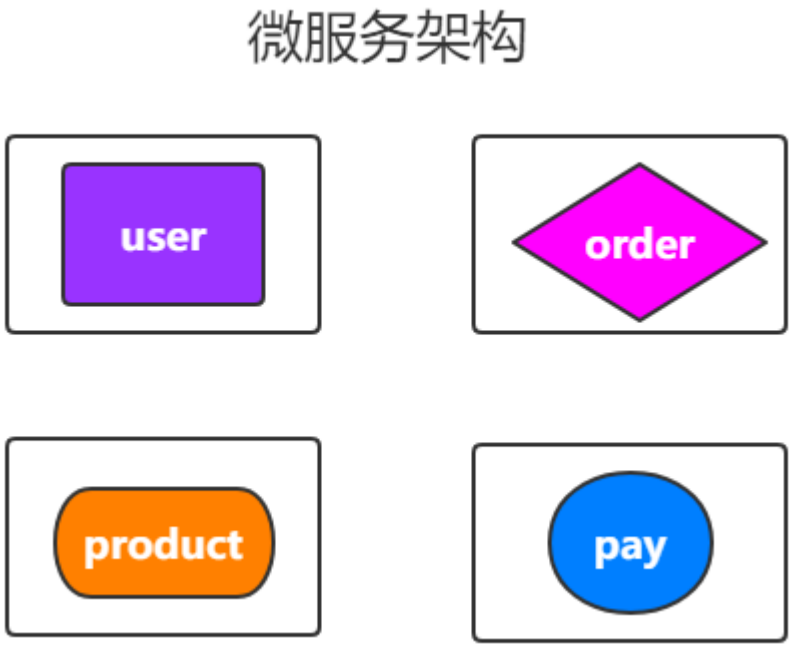
- 优点

功能业务集中在同一个发布包中，部署运行在同一个进程中，方便测试与部署

- 缺点

(1)开发效率低
(2)代码维护难
(3)部署不灵活：构建的时间特别长
(4)稳定性不高：一个微不足道的小问题，导致整个系统不可用
(5)扩展性不够：比如某个服务容量大一些，某个服务容量小一些

2.2 微服务架构



- 优点

> 独立性，敏捷性，技术栈灵活，高效团队

- 缺点

> 额外的工作[DDD领域驱动设计]，数据一致性，沟通成本

03 What is Spring Cloud

官网：https://spring.io/projects/spring-cloud#overview

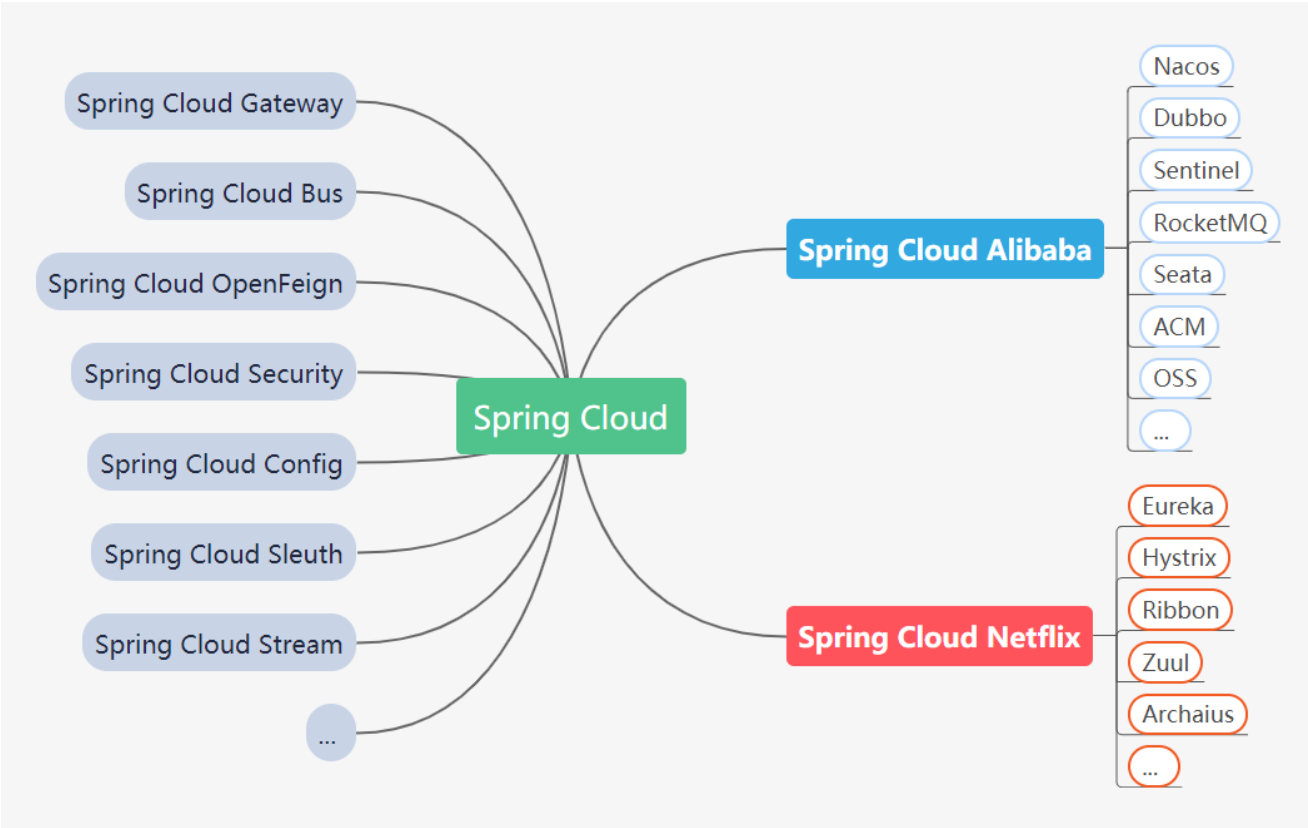
```
Spring Cloud provides tools for developers to quickly build some of the common patterns in
distributed systems (e.g. configuration management, service discovery, circuit breakers,
intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership
election, distributed sessions, cluster state).
```

Features:

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Global locks
- Leadership election and cluster state

- Distributed messaging

> 所以Spring Cloud可以理解为Spring生态用于解决微服务架构下各种问题而制定的标准与规范
>
> 基于该规范实现的组件有很多，可以看官网，也可以看体系图



3.1 Spring Cloud Netflix

官网：https://spring.io/projects/spring-cloud-netflix

```
Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through
autoconfiguration and binding to the Spring Environment and other Spring programming model
idioms.
```

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 registration of Zuul filters, and a simple convention over configuration approach to reverse proxy creation

3.2 Spring Cloud Alibaba

官网：https://spring.io/projects/spring-cloud-alibaba

```
Spring Cloud Alibaba provides a one-stop solution for distributed application development. It contains all the components required to develop distributed applications, making it easy for you to develop your applications using Spring Cloud.
```

Features:

- **Flow control and service degradation**：flow control, circuit breaking and system adaptive protection with Alibaba Sentinel
- **Service registration and discovery**：instances can be registered with Alibaba Nacos and clients can discover the instances using Spring-managed beans. Supports Ribbon, the client side load-balancer via Spring Cloud Netflix
- **Distributed Configuration**：using Alibaba Nacos as a data store
- **Event-driven**：building highly scalable event-driven microservices connected with Spring Cloud Stream RocketMQ Binder
- **Message Bus**: link nodes of a distributed system with Spring Cloud Bus RocketMQ
- **Distributed Transaction**：support for distributed transaction solution with high performance and ease of use with Seata
- **Dubbo RPC**：extend the communication protocols of Spring Cloud service-to-service calls by Apache Dubbo RPC

04 Spring&Boot&Cloud

4.1 Spring Framework

> IoC、DI、AOP、MVC、JDBC和Webflux等

```
The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform.
```

4.2 Spring Boot

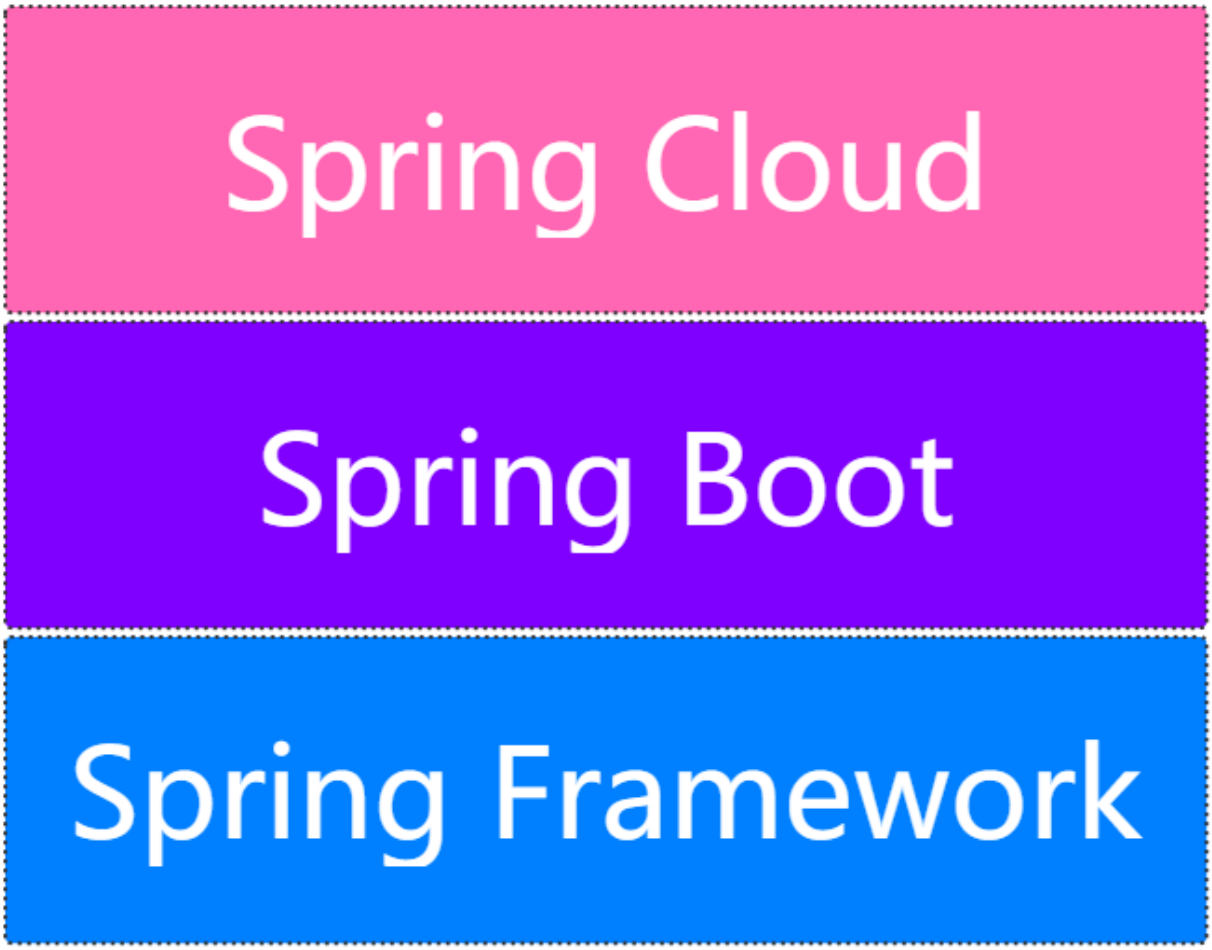
> Embed Tomcat、Starter dependency、Automatically Assembly和Production-ready等

```
Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".
```

4.3 Spring Cloud

> Distributed/version Configuration、Service registration and discovery、Routing和Load Balancing等

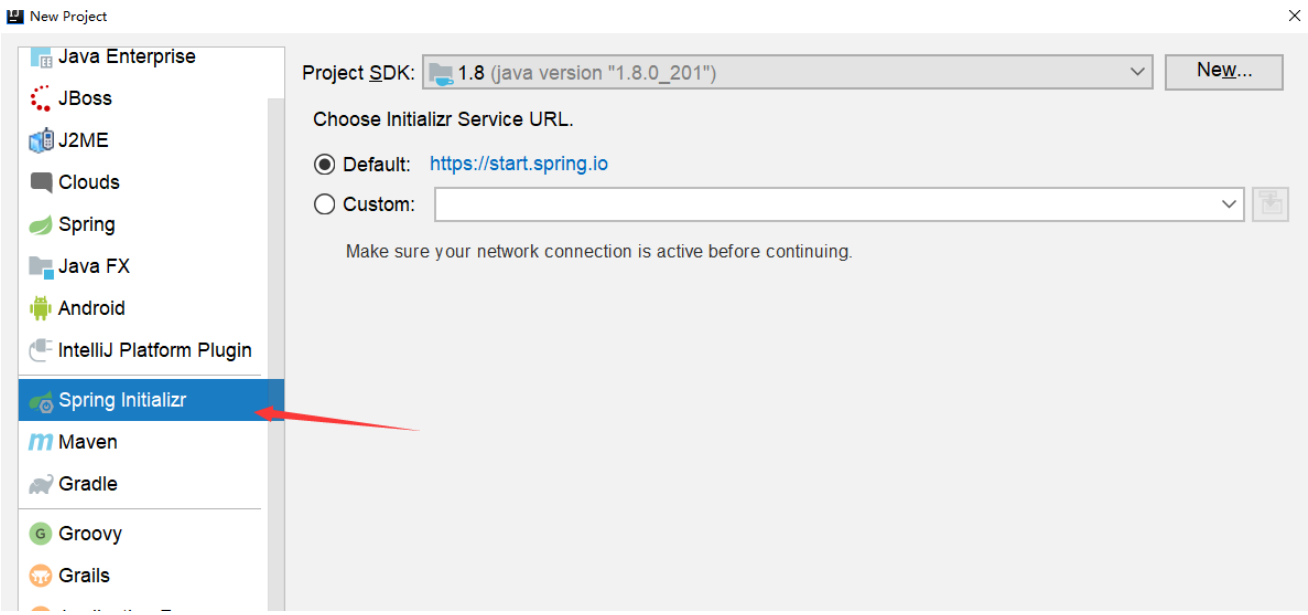
```
Spring Cloud provides tools for developers to quickly build some of the common patterns in
distributed systems
```



05 Spring Boot简单入门

5.1 创建Spring Boot工程

> 比如用IDEA这种方式
>
> （1）JDK版本8+
>
> （2）Spring Boot版本2.1.x
>
> （3）Web starter



5.2 编写Controller

> 启动测试，访问http://localhost:8080/hello

```
@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String hello(){
        return "hello spring boot";
    }
}
```

5.3 Actuator

> 引入actuator依赖，编写配置，访问端点

- 添加依赖

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

- 访问端点

```
{
  - _links: {
      - self: {
            href: "http://localhost:8080/actuator",
            templated: false
        },
      - health: {
            href: "http://localhost:8080/actuator/health",
            templated: false
        },
      - health-path: {
            href: "http://localhost:8080/actuator/health/{*path}",
            templated: true
        },
      - info: {
            href: "http://localhost:8080/actuator/info",
            templated: false
        }
    }
}
```


- 配置health详情

> 再次访问

```
management:
endpoint:
 health:
   show-details: always
```

- 激活所有端点

> 再次访问
>
> 测试访问：http://localhost:8080/actuator/metrics/jvm.threads.live

```
endpoints:
web:
exposure:
include: '*'
```

5.4 项目打包

> 若有问题，可以加上如下配置

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

mvn -Dmaven.test.skip -U clean install

java -jar xxx.jar

5.6 Spring Boot开发三板斧

- 添加依赖
- 写注解
- 写配置

06 准备两个微服务

> Spring Boot:2.1.16.RELEASE
>
> Spring Cloud:Greenwich.SR3
>
> Spring Cloud Aliaba:0.9.0.RELEASE
>
> Spring Cloud Netflix:2.1.4.RELEASE

6.1 order-service

> 就是一个普通的Spring Boot Web项目+Controller+配置端口
>
> 访问：http://localhost:9091/order/query

```
@RestController
@RequestMapping(path="/order",method = RequestMethod.GET)
public class OrderController {
    @RequestMapping("/query")
    public String query(){
        return "list orders";
    }
}
```

```
server:
  port: 9091
```

6.2 user-service

> 就是一个普通的Spring Boot Web项目+Controller+配置端口
>
> 访问：http://localhost:8081/user/test

```
@RestController
@RequestMapping(path="/user",method = RequestMethod.GET)
public class UserController {
    @RequestMapping("/test")
    public String test(){
        return "test";
    }
}
```

```
server:
  port: 8081
```

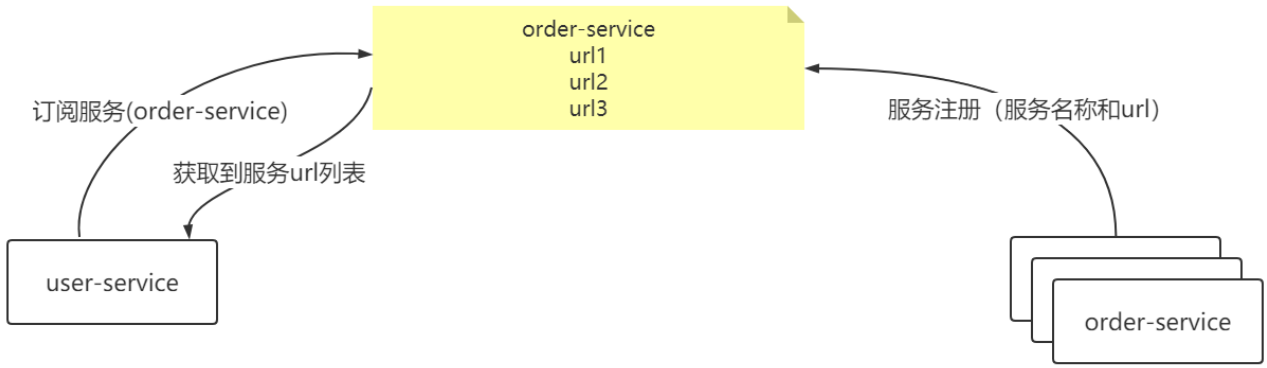
07 整合Spring Cloud

> 整合的好处在于子项目中不需要指定特定的版本，也不需要考虑版本之间的兼容性问题

```
<dependencyManagement>
    <dependencies>
        <!--整合Spring Cloud-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!--整合Spring Cloud Netflix-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-netflix-dependencies</artifactId>
            <version>${spring-cloud-netflix.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!--整合Spring Cloud Alibaba-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>${spring-cloud-alibaba.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

08 服务注册与发现Nacos

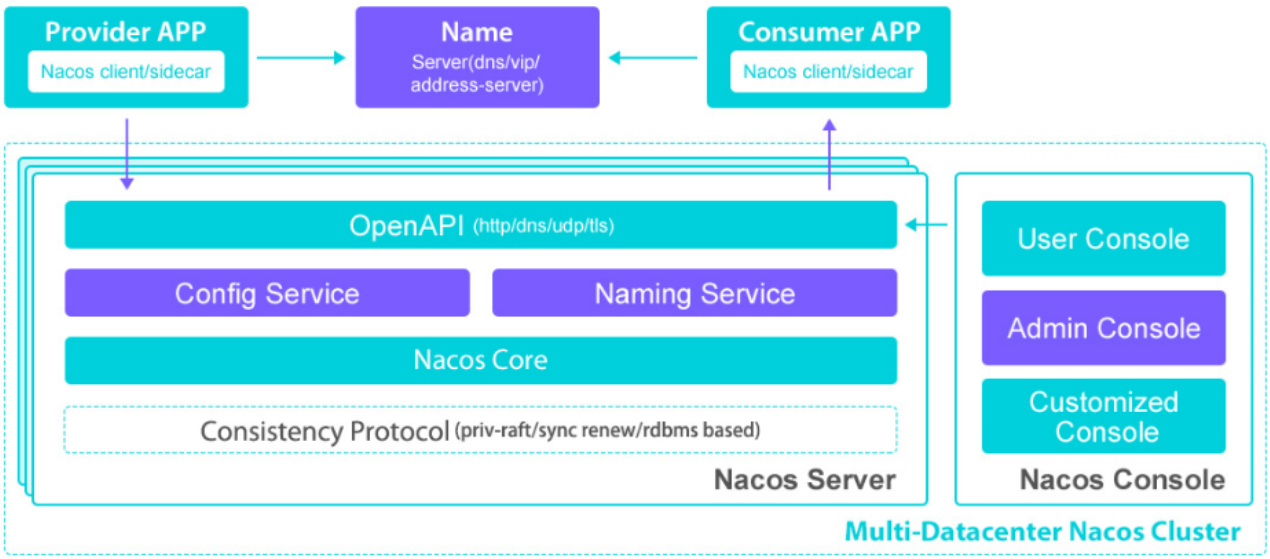
8.1 服务注册与发现架构图



8.2 技术选型-Nacos

> Spring Cloud Alibaba Nacos

官网：https://nacos.io/en-us/docs/quick-start.html



8.3 Nacos基本使用

> (1)下载与构建

```
git clone https://github.com/alibaba/nacos.git

cd nacos/

mvn -Prelease-nacos -Dmaven.test.skip=true clean install -U
ls -al distribution/target/
```

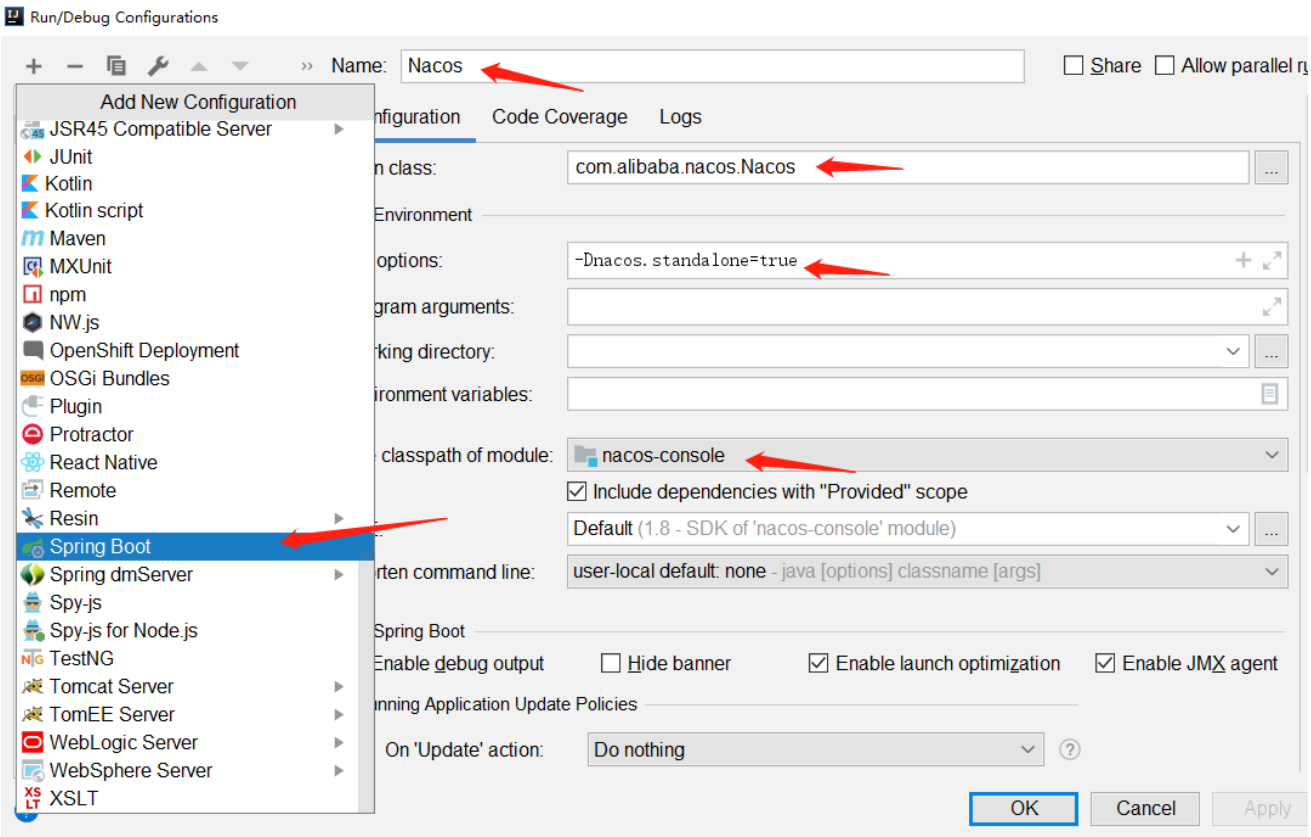
```
// change the $version to your actual path
cd distribution/target/nacos-server-$version/nacos/bin
```

> (2)启动

- 直接启动

  nacos/bin下面的startup.cmd

- 通过源码启动



> (3)访问

http://localhost:8848/nacos

用户名：nacos

密码：nacos

> (4)服务注册

```
curl -X POST 'http://127.0.0.1:8848/nacos/v1/ns/instance?
serviceName=nacos.naming.serviceName&ip=20.18.7.10&port=8080'
```

> (5)服务发现

```
curl -X GET 'http://127.0.0.1:8848/nacos/v1/ns/instance/list?
serviceName=nacos.naming.serviceName'
```

8.4 启动时服务注册

> (1)整合nacos-discovery
>
> 两个项目中都要引入

```
<!--整合nacos-starter-discovery-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

> (2)在order的application.yml文件中操作

```
spring:
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
  application:
    name: order-service
```

> (3)启动order项目，观察启动日志&查看nacos的控制面板

> (4)更改order的port，并且设置允许Parallel，注册3个服务

8.5 服务发现DiscoveryClient

> 可以在user-service中找到order-service的服务url列表
>
> 下面的操作在user-service中编写和配置

- Java代码

```
@Autowired
private DiscoveryClient discoveryClient;

@RequestMapping("/getinstances")
public List<ServiceInstance> getinstances(){
    List<ServiceInstance> instances = discoveryClient.getInstances("order-service");
    return instances;
}
```

- application.yml

```yaml
spring:
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
  application:
    name: user-service
```

- 访问localhost:8081/user/getinstances


```
[
  - {
        serviceId: "order-service",
        host: "192.168.1.3",
        port: 9093,
        secure: false,
      - metadata: {
            nacos.instanceId: "192.168.1.3#9093#DEFAULT#DEFAULT_GROUP@@order-service",
            nacos.weight: "1.0",
            nacos.cluster: "DEFAULT",
            nacos.healthy: "true",
            preserved.register.source: "SPRING_CLOUD"
        },
        uri: "http://192.168.1.3:9093",
        scheme: null,
        instanceId: null
    },
  - {
        serviceId: "order-service",
        host: "192.168.1.3",
        port: 9092,
        secure: false,
      - metadata: {
            nacos.instanceId: "192.168.1.3#9092#DEFAULT#DEFAULT_GROUP@@order-service",
            nacos.weight: "1.0",
            nacos.cluster: "DEFAULT",
            nacos.healthy: "true",
            preserved.register.source: "SPRING_CLOUD"
        },
        uri: "http://192.168.1.3:9092",
        scheme: null,
        instanceId: null
    },
  - {
        serviceId: "order-service",
        host: "192.168.1.3",
        port: 9091,
        secure: false,
      - metadata: {
            nacos.instanceId: "192.168.1.3#9091#DEFAULT#DEFAULT_GROUP@@order-service",
            nacos.weight: "1.0",
            nacos.cluster: "DEFAULT",
            nacos.healthy: "true",
            preserved.register.source: "SPRING_CLOUD"
        },
        uri: "http://192.168.1.3:9091",
        scheme: null,
        instanceId: null
    }
]
```

09 服务调用RestTemplate

9.1 RestTemplate准备

- UserServiceApplication

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

- UserController

```
@Autowired
private RestTemplate restTemplate;
```

9.2 指定uri

- Java代码

```
@RestController
@RequestMapping(path="/user",method = RequestMethod.GET)
public class UserController {
    @RequestMapping("/test")
    public String test(){
        RestTemplate restTemplate = new RestTemplate();
        String url="http://localhost:9091/order/query";
        String result = restTemplate.getForObject(url, String.class);
        return result;
    }
}
```

- 访问

```
http://localhost:8081/user/test
```

9.3 从服务发现列表中获取

> 从DiscoveryClient中获取第一个uri进行访问

```
@RequestMapping("/test2")
public String test2(){
    List<ServiceInstance> instances = discoveryClient.getInstances("order-service");
    String url= instances.stream().map(instance-> instance.getUri().toString()+"/order/query").
        findFirst().orElseThrow(()-> new IllegalArgumentException("找不到实例"));
    return this.restTemplate.getForObject(url, String.class);
}
```

10 负载均衡Ribbon

10.1 手写实现客户端负载均衡

> 既然能够通过DiscoveryClient获取所有服务列表的信息，那到底选择哪个向服务端发起调用请求呢?
>
> 所以这时候就涉及到负载均衡的方式咯，不妨手动先实现一下。

```
// 通过restTemplate访问随机一个url地址
@RequestMapping("/test3")
public String test3(){
    List<ServiceInstance> instances = discoveryClient.getInstances("order-service");
    List<String> uris = instances.stream().map(instance -> instance.getUri().toString() +
"/order/query").
        collect(Collectors.toList());
    int i = ThreadLocalRandom.current().nextInt(uris.size());
    String uri = uris.get(i);
    log.info("访问的uri是"+uri);
    return this.restTemplate.getForObject(uri, String.class);
}
```

10.2 Ribbon实现客户端负载均衡

> Ribbon是netflix的一款客户端负载均衡器，可以从注册中心上自动获取服务列表并且进行负载均衡选择
>
> Ribbon的负载均衡功能，主要是通过LoadBalancerClient来实现的，后面可以看一下源码
>
> 这样就不用使用DiscoveryClient来进行服务发现了，因为Ribbon也能做到
>
> **也就是说**：Ribbon先进行服务发现，然后进行负载均衡。

- 引入ribbon依赖

> 其实starter-discovery里面已经包含了ribbon，所以其实不用手动添加

```
<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-netflix-ribbon -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
    <version>2.1.4.RELEASE</version>
</dependency>
```

- 写注解

> 在restTemplate的Bean添加@LoadBalance注解，就相当于为restTemplate整合了ribbon

```
@Bean
@LoadBalanced   // 相当于为RestTemplate整合了Ribbon的功能
public RestTemplate restTemplate(){
    return new RestTemplate();
}
```

- 写配置

> 暂时没有配置

```
// 使用Ribbon和RestTemplate进行服务的调用
@RequestMapping("/test4")
public String test4(){
    return this.restTemplate.getForObject("http://order-service/order/query", String.class);
}
```

```
Flipping property: order-service.ribbon.ActiveConnectionsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 214748
Client: order-service instantiated a LoadBalancer: DynamicServerListLoadBalancer:{NFLoadBalancer:name=order-service,current list of Servers=[],Load balancer sta
Using serverListUpdater PollingServerListUpdater
Flipping property: order-service.ribbon.ActiveConnectionsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 214748
DynamicServerListLoadBalancer for client order-service initialized: DynamicServerListLoadBalancer:{NFLoadBalancer:name=order-service,current list of Servers=[19
on failure:0;    Total blackout seconds:0;   Last connection made:Thu Jan 01 08:00:00 CST 1970;  First connection made: Thu Jan 01 08:00:00 CST 1970;    Active C
Total blackout seconds:0;   Last connection made:Thu Jan 01 08:00:00 CST 1970;  First connection made: Thu Jan 01 08:00:00 CST 1970;    Active Connections:0;
Total blackout seconds:0;   Last connection made:Thu Jan 01 08:00:00 CST 1970;  First connection made: Thu Jan 01 08:00:00 CST 1970;    Active Connections:0;

Flipping property: order-service.ribbon.ActiveConnectionsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 214748
```

10.3 Ribbon基本使用

> 这里只先描述结合RestTemplate的使用，比如改变其负载均衡规则

10.3.1 Java代码

- UserServiceRibbonConfiguration

```
@Configuration
@RibbonClient(name="user-service",configuration = RibbonConfiguration.class)
public class UserServiceRibbonConfiguration {
}
```

- RibbonConfiguration

```
@Configuration
public class RibbonConfiguration {
    @Bean
    public IRule rule(){
        return new RandomRule();
    }
}
```

10.3.2 配置文件

```
user-service:
  ribbon:
    NFLoadBalanceRuleClassName: com.netflix.loadbalancer.RandomRule
```

11 声明式HTTP调用Feign

官网：https://spring.io/projects/spring-cloud-openfeign

GitHub：https://github.com/openfeign/feign

```
Feign is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it.
```

> 虽然RestTemplate能够完成基于http协议的服务调用，但是在代码中写类似于
>
> `http://order-service/order/query`这样的http请求代码总觉得不够优雅，而且不符合面向对象的开发思想，试想一下，要是能够在服务消费端调用服务提供者的接口，像调用本地方法接口一样简单，该有多好，而feign就可以理解为这样的中间件。

11.1 Feign的基本使用

- 为user-service整合feign

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

- 写注解

```
@SpringBootApplication
@EnableFeignClients
public class UserServiceApplication {
    // ...
}
```

- 编写feignclient

```
@FeignClient(name="order-service")
public interface OrderServiceFeignClient {
    /**
     * 最终会构建出这样的uri：http://order-service/order/query
     * @return
     */
    @RequestMapping("/order/query")
    public String query();
}
```

- controller中调用

```
@Autowired
private OrderServiceFeignClient orderServiceFeignClient;
// 测试feign
@RequestMapping("/testfeign")
public String testfeign(){
    return orderServiceFeignClient.query();
}
```

> 启动user-service，访问localhost:8081/user/testfeign接口，发现同样可以访问，并且能够做负载均衡
>
> feign中的负载均衡也是通过ribbon来实现的，所以ribbon相关的配置在feign中同样生效

11.2 Feign的常用配置

> 既然Feign是声明式的HTTP请求，那么关于HTTP请求的一些日志情况是否可以打印出来呢?
>
> 答案是肯定的，只是默认feign的日志并没有开启，需要通过配置的方式开启。

11.2.1 Java代码

> 在com.csdn.userservice包下创建一个文件夹config

- 定义一个配置类

```
public class OrderServiceFeignConfiguration {
    @Bean
    public Logger.Level level(){
        return Logger.Level.FULL;
    }
}
```

- 将配置类放到feignclient上

```
@FeignClient(name="order-service",configuration = OrderServiceFeignConfiguration.class)
public interface OrderServiceFeignClient {
}
```

- yml文件中配置日志级别

```
logging:
  level:
    # 这边写自己FeignClient的全路径
    com.csdn.userservice.feignclient.OrderServiceFeignClient: debug
```

> 重启项目，然后通过feign访问order看效果

```
: [OrderServiceFeignClient#query] <--- HTTP/1.1 200 (652ms)
: [OrderServiceFeignClient#query] connection: keep-alive
: [OrderServiceFeignClient#query] content-length: 11
: [OrderServiceFeignClient#query] content-type: text/plain;charset=UTF-8
: [OrderServiceFeignClient#query] date: Thu, 13 Aug 2020 19:45:20 GMT
: [OrderServiceFeignClient#query] keep-alive: timeout=60
: [OrderServiceFeignClient#query]
: [OrderServiceFeignClient#query] list orders
: [OrderServiceFeignClient#query] <--- END HTTP (11-byte body)
: Flipping property: order-service.ribbon.ActiveConnectionsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.ac
```

11.2.2 配置文件

```
# feign日志级别属性配置
feign:
  client:
    config:
      # feign调用的微服务名称
      order-service:
        loggerLevel: full
logging:
  level:
    com.csdn.userservice.feign: debug
```

11.3 更改Feign的HTTP请求方式

> 将feign默认的HttpURLConnection方式 更改为 apache的HttpClient

- 添加httpclient依赖

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

- 写配置

```
feign:
  httpclient:
    enabled: true
    max-connections: 200
    max-connections-per-route: 50
```

12 服务配置中心Nacos

> 配置文件每次修改之后就要重启，这样会比较麻烦，有办法实现配置文件动态刷新吗?

> (1)引入maven依赖

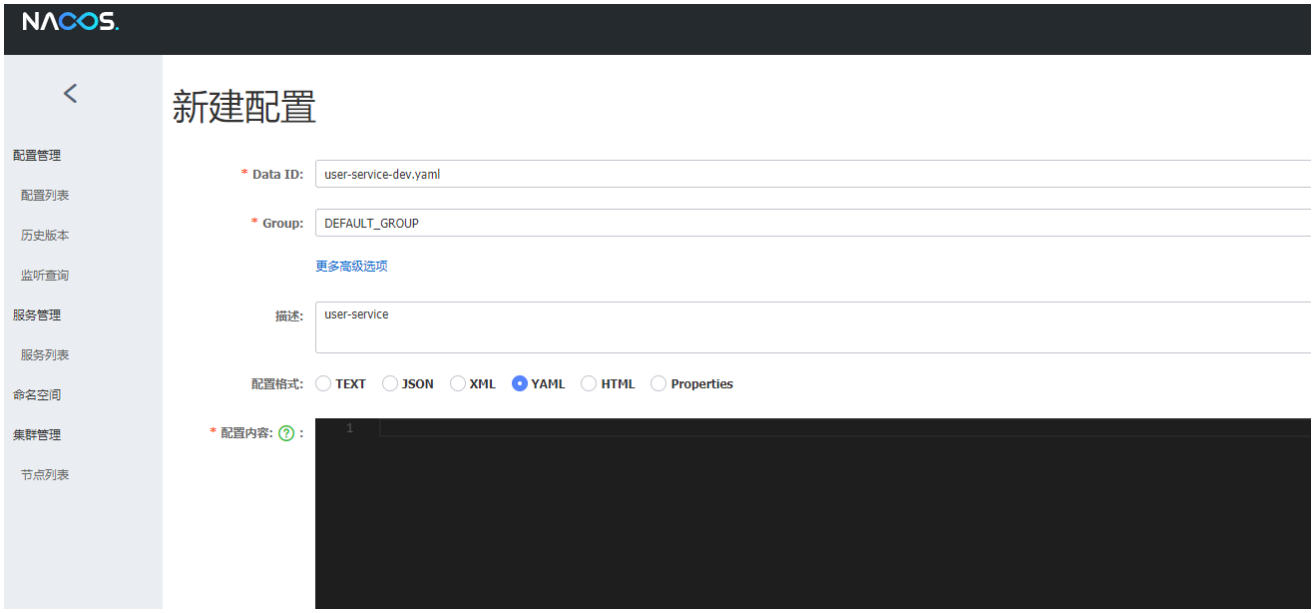
```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

```
<!--使bootstrap配置文件生效-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-context</artifactId>
</dependency>
```

> (2)创建编写bootstrap.yaml文件

```
spring:
  cloud:
    nacos:
      config:
        server-addr: localhost:8848
        file-extension: yaml
  application:
    name: user-service
  profiles:
    active: dev
```

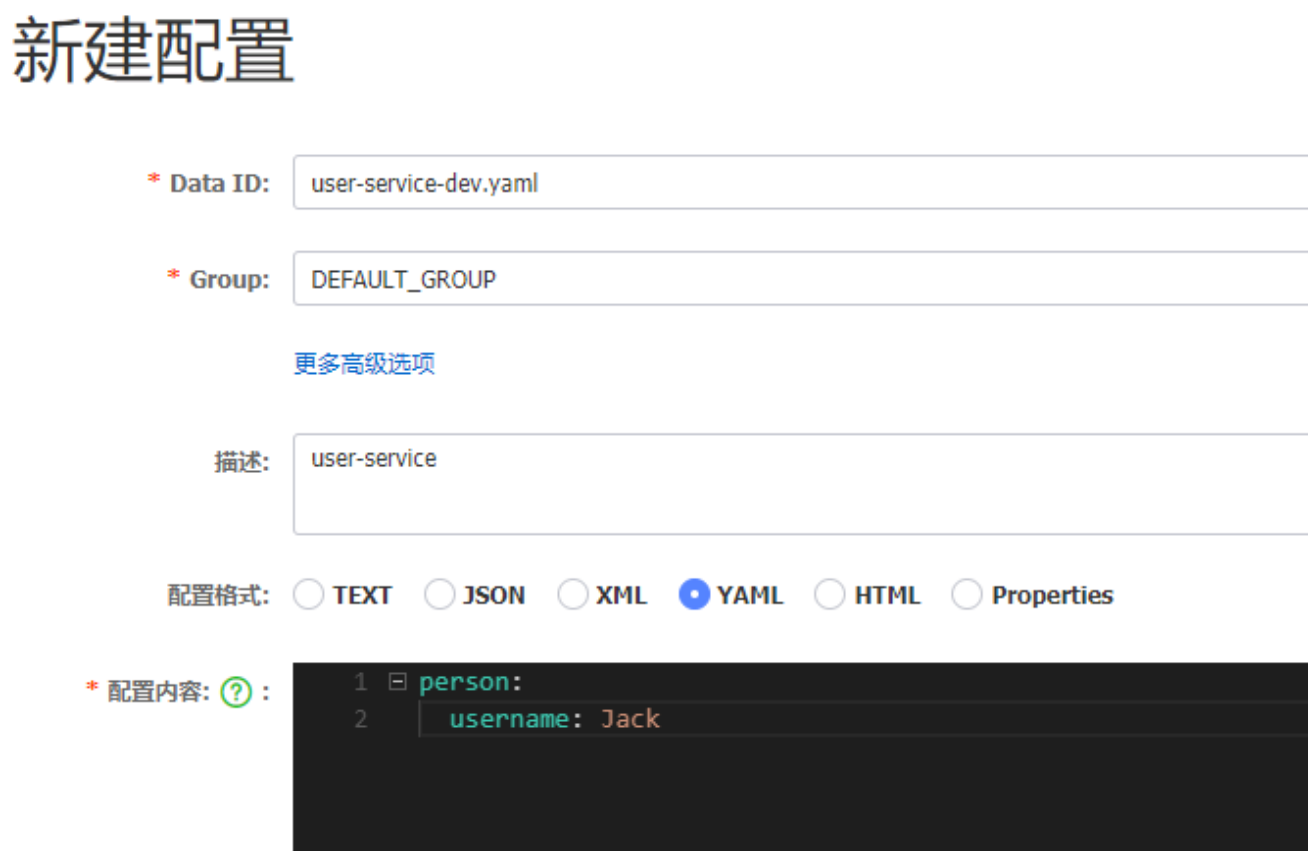
> (3)在nacos上创建指定配置文件
>
> 命名：user-service-dev.yaml



> (4)准备测试接口

```
// 测试nacosconfig
@Value("${person.username}")   //org.springframework.beans.factory.annotation.Value;
private String personUsername;

@RequestMapping("/nacosconfig")
public String nacosconfig(){
    return this.personUsername;
}
```

> (5)编写nacos上的配置文件



> (6)重启user-service项目访问测试
>
> localhost:8081/nacosconfig
>
> **注意**：记得在UserController上加**@RefreshScope**

```
@RefreshScope
public class UserController {
// ...
}
```

不断在nacos控制台改变person.username的值查看结果

13 服务容错Sentinel

> 微服务中经常会有级联调用，为了对某些服务进行保护，或者保证某些服务不被其他服务拖死，经常采用一些容错的方案，比如：超时、限流、熔断等

github：https://github.com/alibaba/Sentinel

```
As distributed systems become increasingly popular, the reliability between services is becoming more important than ever before. Sentinel takes "flow" as breakthrough point, and works on multiple fields including flow control, traffic shaping, circuit breaking and system adaptive protection, to guarantee reliability and resilience for microservices.
```

13.1 单独使用

> 参照sentinel的github地址下面的步骤即可，so easy

13.2 在Spring Cloud

比如不妨对user/hello接口进行容错设置

> (1)引入sentinel的依赖

```
<!--整合sentinel-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

> (2)写配置

```
spring:
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
    sentinel:
      transport:
        dashboard: localhost:8080
```

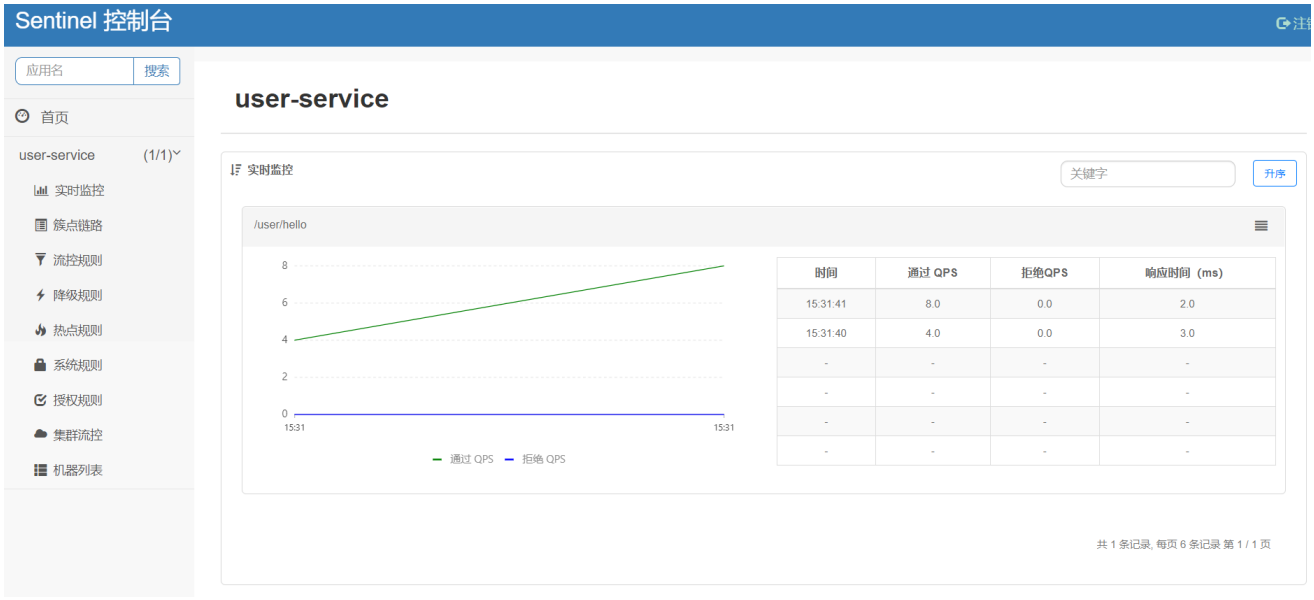
> (3)下载sentinel的jar包
>
> https://github.com/alibaba/Sentinel/releases
>
> 版本选择1.6.3

> (4)启动sentinel

```
java -jar sentinel-dashboard-1.6.3.jar
```

> (5)访问sentinel控制台
>
> localhost:8080
>
> 账号密码都是：sentinel

> (6)启动order-service项目，访问/user/hello接口
>
> localhost:8081/user/hello
>
> 等待一会，查看sentinel控制台的变化



13.3 流控规则

> 体验一下流控规则

> 不断刷新访问localhost:8081/user/hello

```
Blocked by Sentinel (flow limiting)
```

14 服务网关Gateway

官网：https://spring.io/projects/spring-cloud-gateway

相比Netflix的Zuul功能和性能更加强大

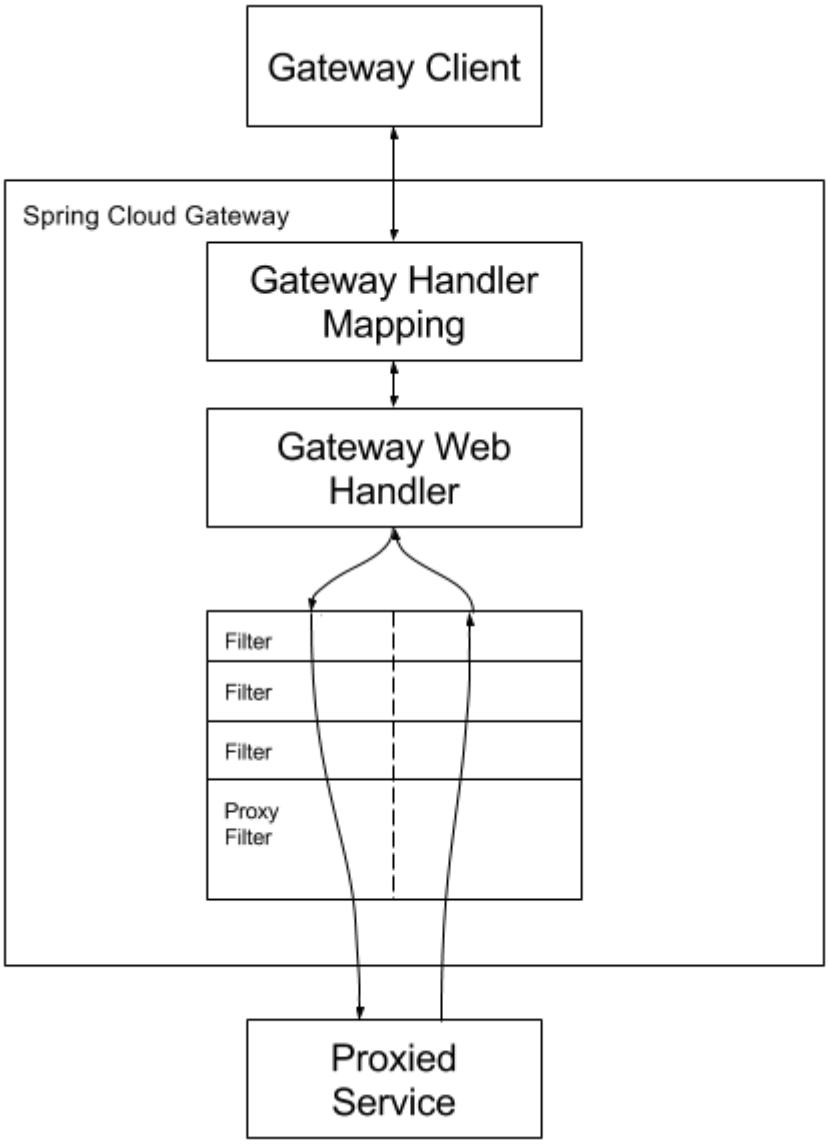
Features

- Built on Spring Framework 5, Project Reactor and Spring Boot 2.0
- Able to match routes on any request attribute.
- Predicates and filters are specific to routes.
- Hystrix Circuit Breaker integration.

- Spring Cloud DiscoveryClient integration
- Easy to write Predicates and Filters
- Request Rate Limiting
- Path Rewriting

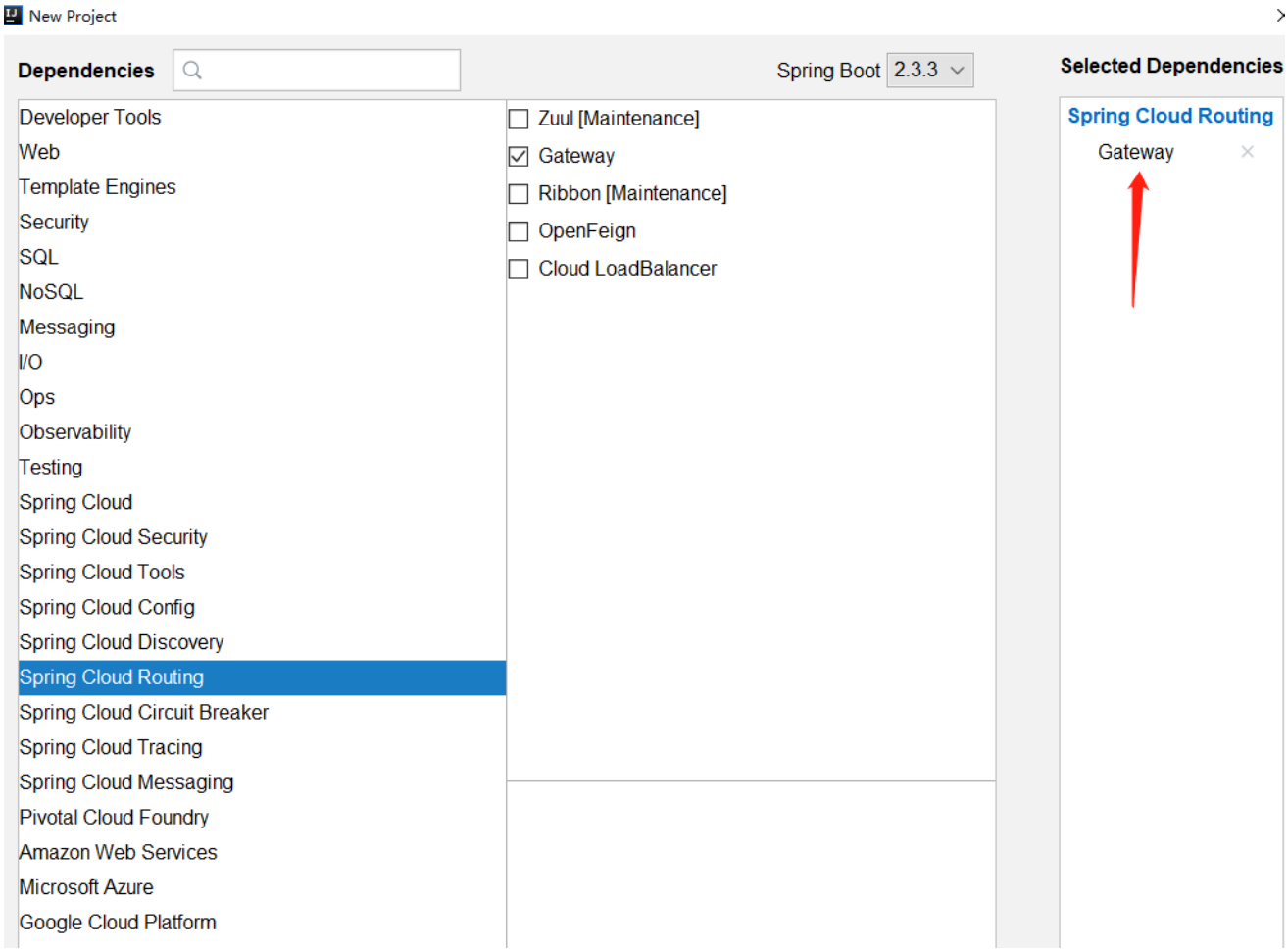
> Gateway基于Netty构建，而Zuul基于BIO Servlet模型构建

架构图



14.1 基本使用

> (1)创建spring boot项目，引入Gateway依赖



> (2)保证Spring Boot和Spring Cloud版本和服务保持一致

Spring Boot:2.1.16

Spring Cloud:Greenwich.SR3

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.16.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.csdn</groupId>
<artifactId>shop-gateway</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>shop-gateway</name>
<description>Demo project for Spring Boot</description>

<properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>Greenwich.SR3</spring-cloud.version>
</properties>
```

> (3)添加Spring Cloud Alibaba Dependencies 和 Nacos Discovery

> (4)编写gateway的基本配置

```
server:
  port: 8090
spring:
  application:
    name: gateway
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
    gateway:
      discovery:
        locator:
          # gateway能够进行服务发现
          enabled: true
```

> (5)通过gateway访问测试

http://localhost:8090

http://localhost:8090/user-service/user/testfeign

http://localhost:8090/order-service/order/query

14.2 路由规则配置

> 其实是希望通过gateway的路由规则配置进行转发到对应的服务

```
server:
  port: 8090
spring:
  application:
    name: gateway
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
    gateway:
      discovery:
        locator:
          # gateway能够进行服务发现
          enabled: true
      routes:
        # 路由的匹配条件
        - predicates:
            - Path=/jack/**
          filters:
            # 表示在将请求发送到下游之前从请求中剥离的路径个数，1表示从二级url路径转发
            - StripPrefix=1
          uri: http://localhost:8081/
```

访问测试：http://localhost:8090/jack/user/testfeign

15 服务调用链路Sleuth&Zipkin

> 微服务的调用基本上都是比较复杂的，当出现问题时，如何能够快速定位到问题，这时候就需要排查调用链咯，那怎么知道调用的过程呢？可以使用Sleuth

官网：https://spring.io/projects/spring-cloud-sleuth

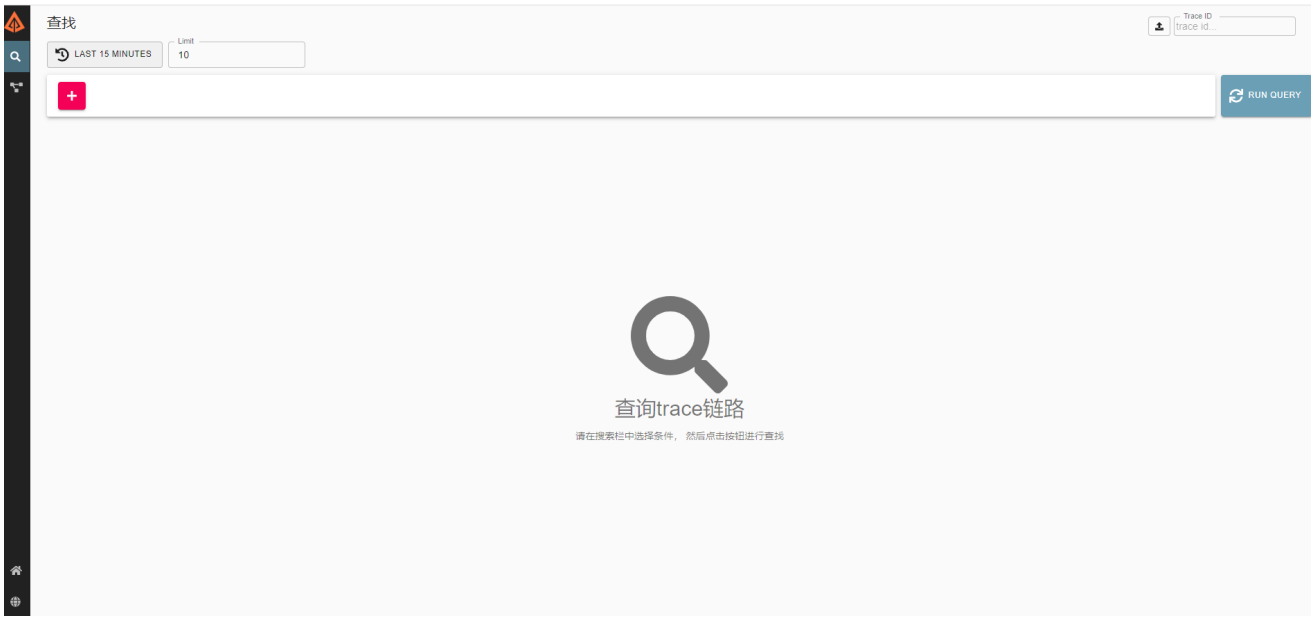
```
Spring Cloud Sleuth provides Spring Boot auto-configuration for distributed tracing.
```

> 可以使用Zipkin进行图形化展示，也就是Zipkin+Sleuth实现分布式链路追踪

> (1)引入zipkin的依赖，其中包含了sleuth的依赖[user-service、order-service、gateway]

```
<!--整合zipkin和sleuth-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

> (2)下载搭建zipkin server
>
> https://github.com/openzipkin/zipkin/tree/master/zipkin-server
>
> java -jar zipkin.jar
>
> localhost:9411

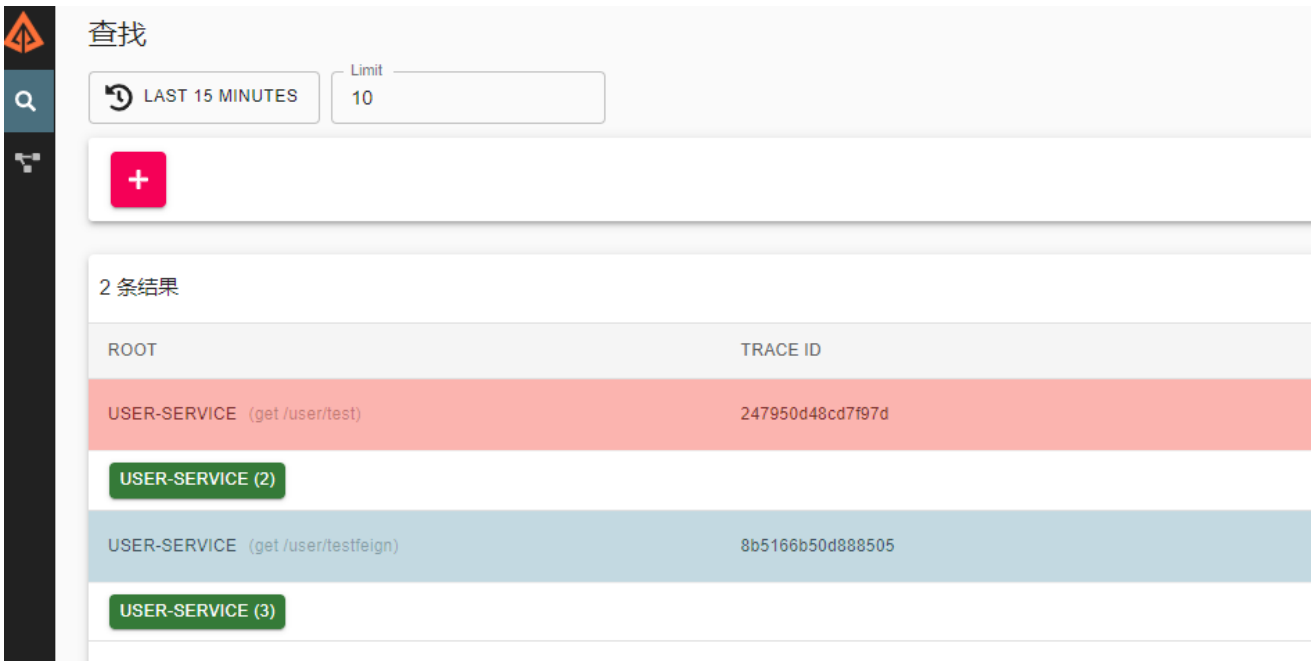


> (3)写配置[user-service、order-service、gateway]

```
zipkin:
  base-url: http://localhost:9411
sleuth:
  sampler:

    probability: 1.0
```

> (4)重新启动user-service、order-service和gateway这3个项目

> (5)访问测试
>
> http://localhost:8081/user/testfeign
>
> http://localhost:8090/user-service/user/testfeign [通过网关访问]



> (5)user-service后台报错解决

```
zipkin:
  base-url: http://localhost:9411
  discoveryClientEnabled: false
```

16 Spring Cloud部分组件源码解读

16.1 Nacos服务注册流程

（1）查找nacos-server中注册接口

```
curl -X POST 'http://127.0.0.1:8848/nacos/v1/ns/instance?
serviceName=nacos.naming.serviceName&ip=20.18.7.10&port=8080'
```

推断：在server.jar中一定有一个/nacos/v1/ns/instance的接口，而且在order项目启动的时候会访问该接口

在nacos-devlop源码中搜索“/v1/ns/instance"，同时在order项目中查找一下，发现依然没有

```
@RestController
@RequestMapping(UtilsAndCommons.NACOS_NAMING_CONTEXT + "/instance")
public class InstanceController {
    //...
}
```

（2）寻找register方法

InstanceController#register()，打个断点，debug启动。

```
curl访问测试
启动order-service项目测试
```

发现都会来到register方法，说明是register方法就是服务注册的接口。

（3）谁调用服务注册接口

既然已经找到server中的InstanceController#register是服务注册的接口

关键是在order项目启动的时候，谁来调用该服务端的接口呢？

```
curl -X POST http://127.0.0.1:8848/nacos/v1/ns/instance?
serviceName=nacos.naming.serviceName&ip=20.18.7.10&port=8080
```

不妨猜测一下是有nacos-client.jar中的类来调用nacos-server.jar中的接口的

断点调试一下

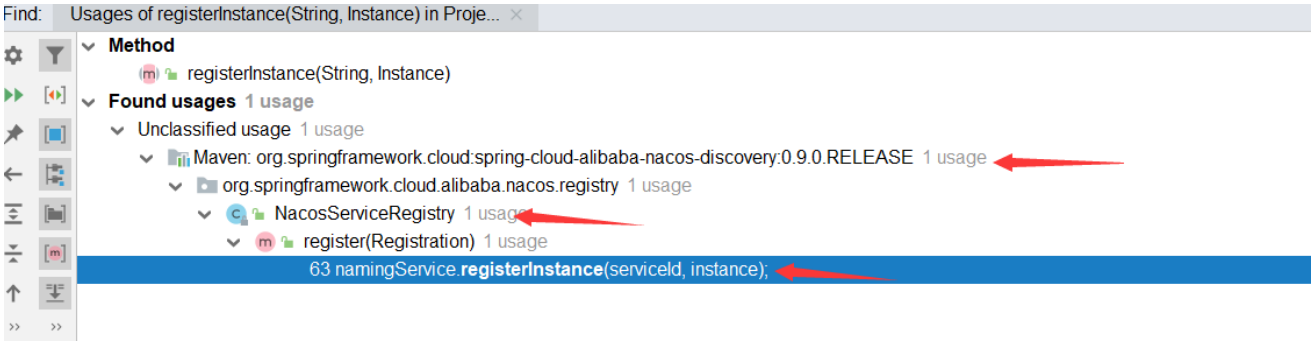
```
@SuppressWarnings("PMD.ServiceOrDaoClassShouldEndWithImplRule")
public class NacosNamingService implements NamingService {
    @Override
    public void registerInstance(String serviceName, Instance instance) throws NacosException {
        registerInstance(serviceName, Constants.DEFAULT_GROUP, instance);
    }
}
```

（4）谁调用NacosNamingService#registerInstance

Shift+Alt+Ctrl+F7 && Project and Libraries

spring-cloud-nacos-discovery.jar：NacosServiceRegistry#register

断点调试一下



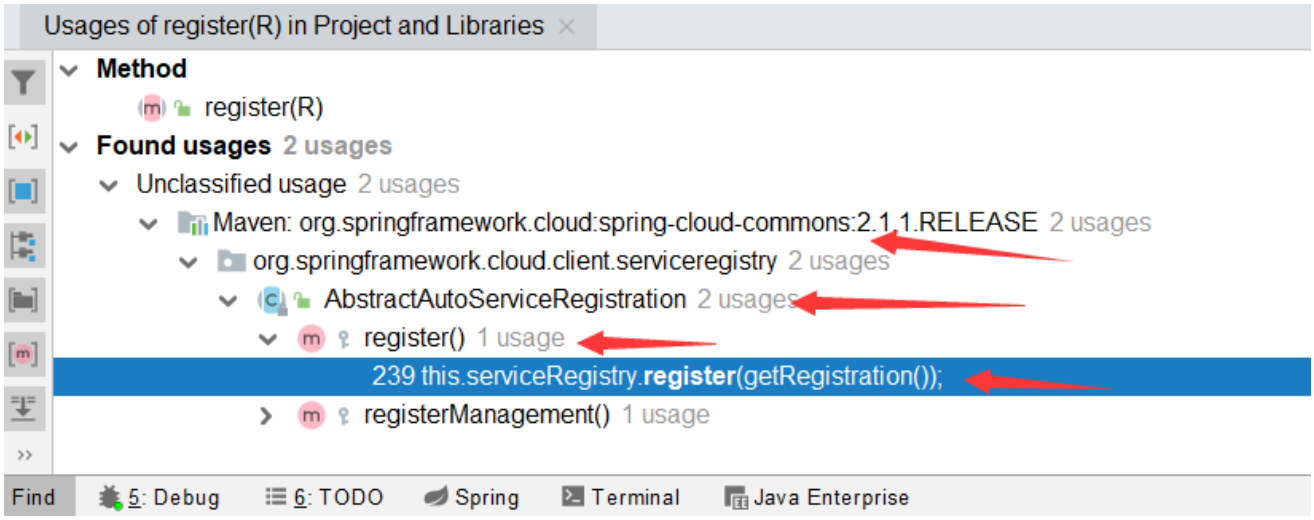
（5）谁调用NacosServiceRegistry#register

Shift+Alt+Ctrl+F7 && Project and Libraries

spring-cloud-commons.jar：AbstractAutoServiceRegistration#register()

进一步调用了具体的实现NacosAutoServiceRegistration#register()

断点调试一下



（6）Spring Cloud规范

> 上面的register方法的定义和调用不禁让我们联想到Spring Cloud这一微服务生态，通过spring-cloud-commons.jar来定义的，而像不同的组件就是对整个规范的实现。

- eureka的服务注册

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
    <version>1.4.7.RELEASE</version>
</dependency>
```

> 搜索EurekaServiceRegitry，查找register方法，查看其使用，发现也是spring-cloud-commons定义规范

- consul的服务注册

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
    <version>2.2.1.RELEASE</version>
</dependency>
```

> 搜索ConsulServiceRegitry，查找register方法，查看其使用，发现也是spring-cloud-commons定义规范

- zookeeper的服务注册

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
    <version>2.2.1.RELEASE</version>
</dependency>
```

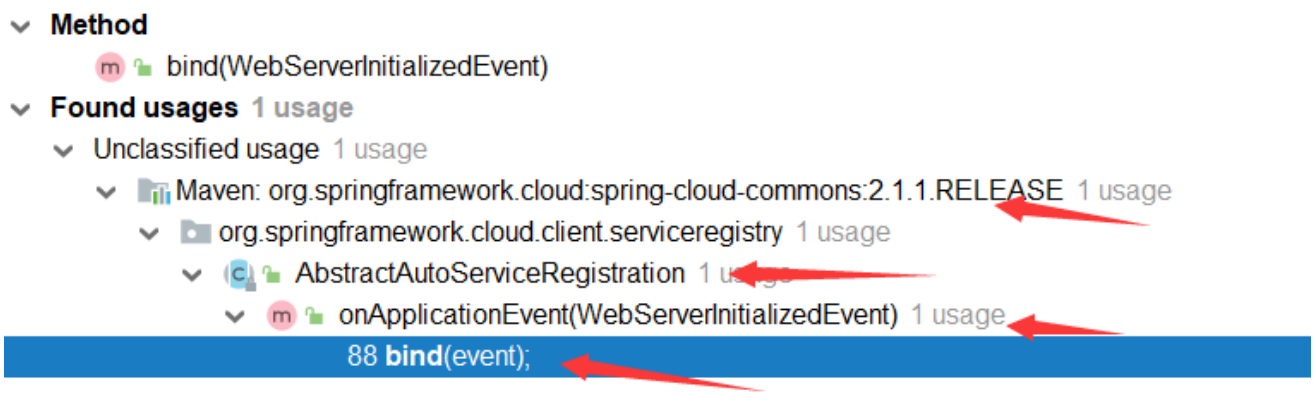
> 搜索ZookeeperServiceRegitry，查找register方法，查看其使用，发现也是spring-cloud-commons定义规范

（7）谁调用AbstractAutoServiceRegistration#register

Shift+Alt+Ctrl+F7 && Project and Libraries -->查看第二个usages

register()->start()->bind()->onApplicationEvent()->ApplicationListener

断点调试一下



显然：这是一个listener与event事件机制的使用，典型的观察者模式

（8）ApplicationListener

Spring ApplicationListener 是Spring事件机制的一部分

与ApplicationEvent抽象类结合完成ApplicationContext的事件通知机制.

- ContextRefreshedEvent事件监听

当Application被初始化或者刷新时，会触发ContextRreshedEvent事件

```
@Component
public class Test implements ApplicationListener<ContextRefreshedEvent> {
    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        System.out.println("我被调用了...");
    }
}
```

因为在Spring Boot启动的时候会调用事件的发布

```
AbstractApplicationContext#refresh()->finishRefresh()->
    publishEvent(new ContextRefreshedEvent(this));
```

- WebServerInitializedEvent

监听WebServerInitializedEvent事件，发现也能打印出信息

说明在Spring Boot启动过程中，也会发布该类型的事件

比如：publishEvent(WebServerInitializedEvent)之类的

```
@Component   //千万别忘了
public class Test implements ApplicationListener<WebServerInitializedEvent> {
    @Override
    public void onApplicationEvent(WebServerInitializedEvent event) {
        System.out.println("我被调用了...");
    }
}
```

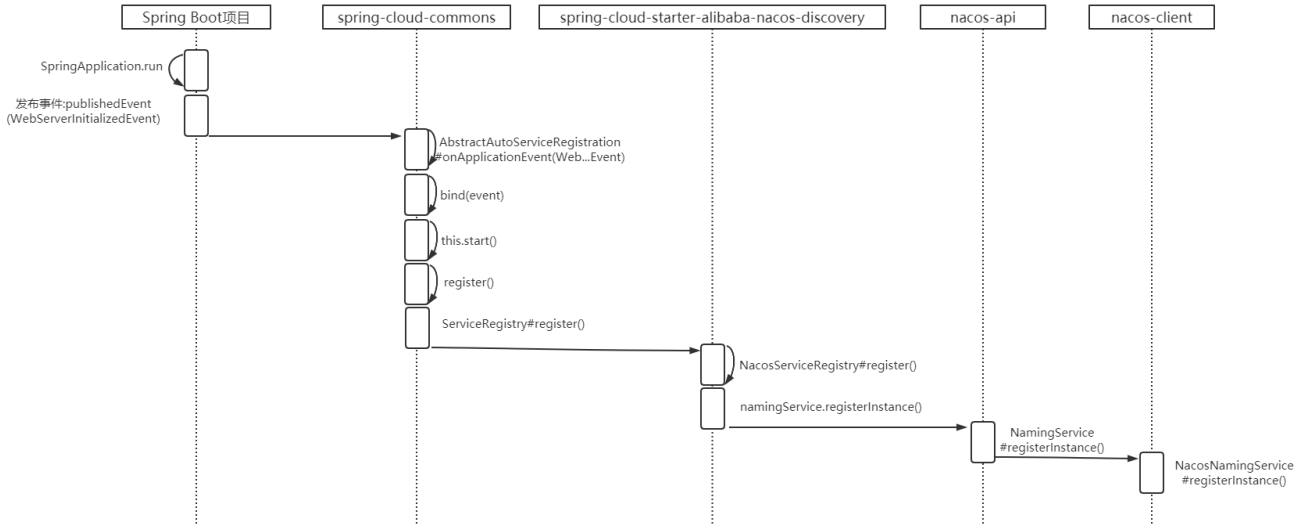
（9）寻找WebServerInitializedEvent的发布

经过上面的分析，现在就只需要寻找到Spring Boot启动过程中WebServerInitializedEvent事件的发布咯

Tomcat启动完成之后，会发布一个ServletWebServerInitializedEvent事件

（10）服务注册时序图

不妨用一张时序图把上述的过程串一串



（11）证明nacos-client调用了nacos-server

要想整个过程完整，还需要证明一下NacosNamingService#registerInstance()方法就是调用了rest api接口

类似于这样的接口

```
curl -X POST http://127.0.0.1:8848/nacos/v1/ns/instance?
serviceName=nacos.naming.serviceName&ip=20.18.7.10&port=8080
```

- 定位到NacosNamingService#registerInstance
- 断点

```
serverProxy.registerService(NamingUtils.getGroupedName(serviceName, groupName), groupName,
instance);
```

- debug启动程序

停留在serverProxy.registerService这里，按F7进入方法

- 断点reqAPI

```
serviceName=order
groupName=default_group
instance=192.168.1.3 and 8081 等
NACOS_URL_INSTANCE=/nacos/v1/ns/instance
```

- 断点reqAPI

```
return reqAPI(api, params, snapshot, method);
```

- 断点callServer(api，params，nacosDomain)

```
NamingProxy#callServer();
```

> 最终发起的是HttpClient请求，调用request方法，使用的是HttpURLConnection类

16.2 Nacos服务注册的心跳机制

> 在nacos-client通过NacosNamingService#registerInstance方法进行服务注册时
>
> 在此之前会封装一个BeatInfo，用于向nacos-server发送心跳，从而来判断当前的服务是否正常

- beatReactor#addBeatInfo

```
beatReactor.addBeatInfo(NamingUtils.getGroupedName(serviceName, groupName), beatInfo);
```

- NacosNamingService构造函数#init()

```
beatReactor = new BeatReactor(serverProxy, initClientBeatThreadCount(properties));
```

- BeatReactor

```
private ScheduledExecutorService executorService;

private volatile long clientBeatInterval = 5 * 1000;

private NamingProxy serverProxy;

public final Map<String, BeatInfo> dom2Beat = new ConcurrentHashMap<String, BeatInfo>();
```

```
executorService.schedule(new BeatProcessor(), 0, TimeUnit.MILLISECONDS);
```

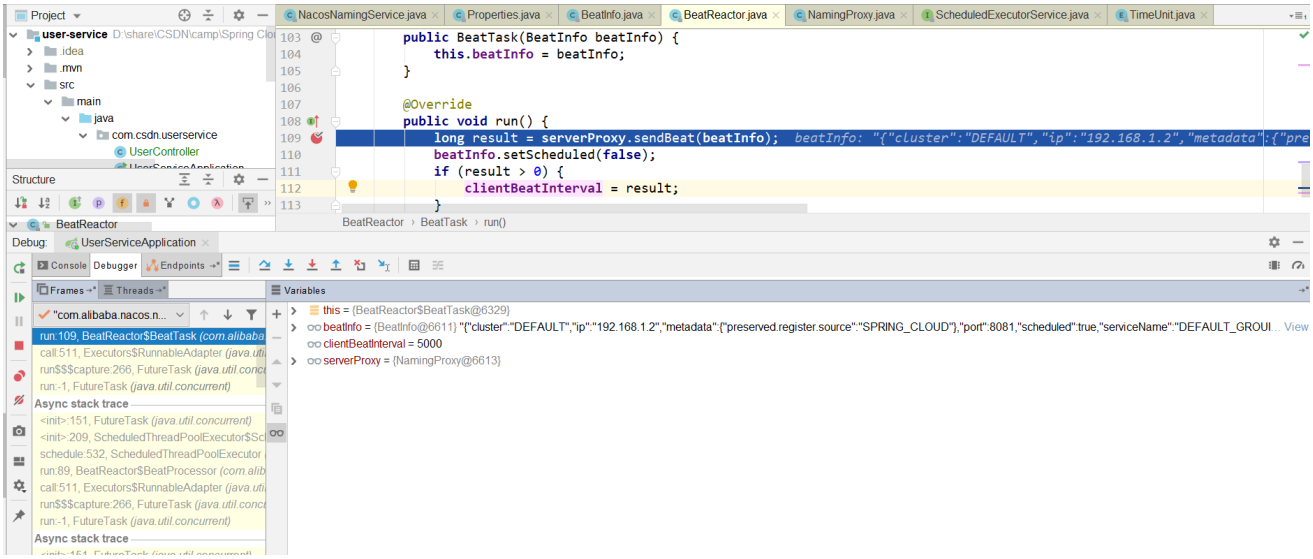
- BeatProcessor

```
executorService.schedule(new BeatTask(beatInfo), 0, TimeUnit.MILLISECONDS);
```

- BeatTask

```
@Override
public void run() {
    long result = serverProxy.sendBeat(beatInfo);
    beatInfo.setScheduled(false);
    if (result > 0) {
        clientBeatInterval = result;
    }
}
```

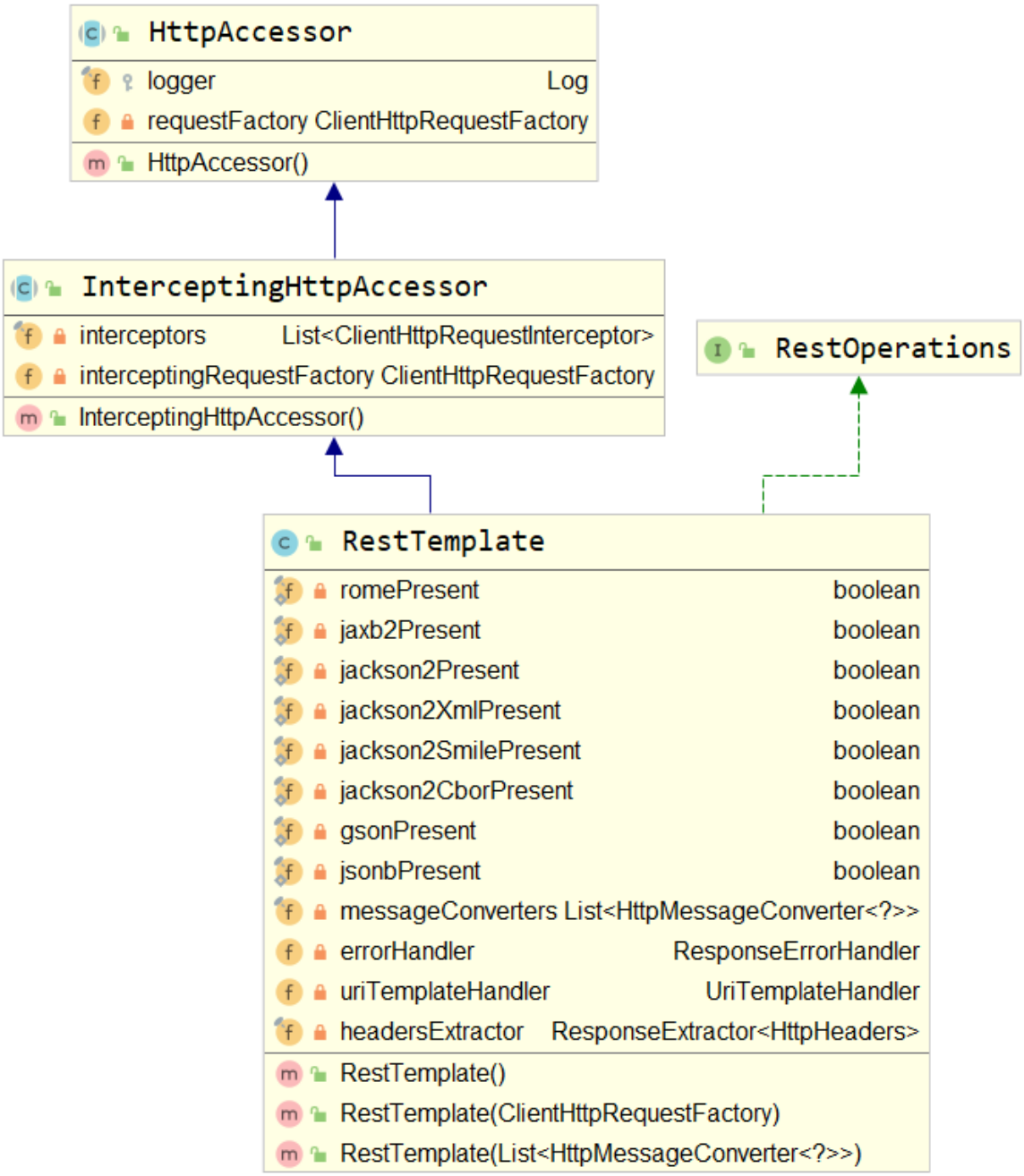
> 可以在serverProxy#sendBeat(beatInfo)方法上打一个断点，然后重启服务，等待5秒，观察nacos控制台



> 同时可以继续查看sendBeat方法，然后找到nacos-server中的访问url

16.3 RestTemplate源码解析

> RestTemplate其实就是Spring封装的用于HTTP请求的模板工具类，相比Apache的Http Client要更加方便




```
String url="http://localhost:9091/order/query";
String result = this.restTemplate.getForObject(url, String.class);
```

RestTemplate#getForObject->RestTemplate#execute->RestTemplate#doExecute->HttpAccessor#createRequest->getRequestFactory().createRequest(url, method);

> 查看一下getRequestFactory()->this.requestFactory->new SimpleClientHttpRequestFactory()
>
> 也就是默认情况下使用的是SimpleClientHttpRequestFactory工厂类

ClientHttpRequestFactory#createRequest->SimpleClientHttpRequestFactory#createRequest

> 可以发现默认SimpleClientHttpRequestFactory使用的是java.net.HttpURLConnection

注意：其实getRequestFactory()方法不是直接调用的HttpAccessor#getRequestFactory，而是先调用的InterceptingHttpAccessor#getRequestFactory

```
@Override
public ClientHttpRequestFactory getRequestFactory() {
    List<ClientHttpRequestInterceptor> interceptors = getInterceptors();
    // interceptors不为空时，调用interceptingRequestFactory
    if (!CollectionUtils.isEmpty(interceptors)) {
        ClientHttpRequestFactory factory = this.interceptingRequestFactory;
        if (factory == null) {
            factory = new InterceptingClientHttpRequestFactory(super.getRequestFactory(),
interceptors);
            this.interceptingRequestFactory = factory;
        }
        return factory;
    }
    // interceptors为空时，调用HttpAccessor#getRequestFactory
    else {
        return super.getRequestFactory();
    }
}
```

> 这样一来，就会每一步都调用ClientHttpRequestInterceptor#intercept方法

16.4 Ribbon源码解析

> **使用Ribbon和RestTemplate进行服务的调用**
>
> 当restTemplate请求http://order-service/order/query地址时，ribbon会根据order-service的名字，在注册中心找到对应的uri地址，并且进行负载均衡，替换order-service的值，让restTemplate进行访问

16.4.1 @LoadBalanced

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

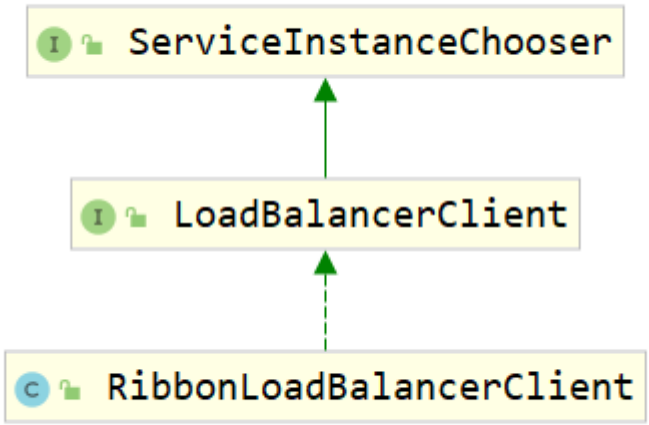
> 查看@LoadBalanced
>
> 发现RestTemplate能够具备负载均衡的能力，是因为@LoadBalanced注解通过LoadBalancerClient实现的

```
/**
 * Annotation to mark a RestTemplate bean to be configured to use a LoadBalancerClient.
 * @author Spencer Gibb
 */
@Target({ ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Qualifier
public @interface LoadBalanced {

}
```

16.4.2 LoadBalancerClient接口

> spring-cloud-commons.jar：interface LoadBalancerClient
>
> 查看其实现，发现是spring-cloud-netflix-ribbon.jar：class RibbonLoadBalancerClient

- 查看其类关系图



- 将RibbonLoadBalancerClient交给Spring IoC容器管理

```
@Configuration
@Conditional(RibbonAutoConfiguration.RibbonClassesConditions.class)
@RibbonClients
@AutoConfigureAfter(name =
"org.springframework.cloud.netflix.eureka.EurekaClientAutoConfiguration")
@AutoConfigureBefore({ LoadBalancerAutoConfiguration.class,
        AsyncLoadBalancerAutoConfiguration.class })
@EnableConfigurationProperties({ RibbonEagerLoadProperties.class,
        ServerIntrospectorProperties.class })
public class RibbonAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean(LoadBalancerClient.class)
    public LoadBalancerClient loadBalancerClient() {
```

```
        return new RibbonLoadBalancerClient(springClientFactory());
    }
}
```

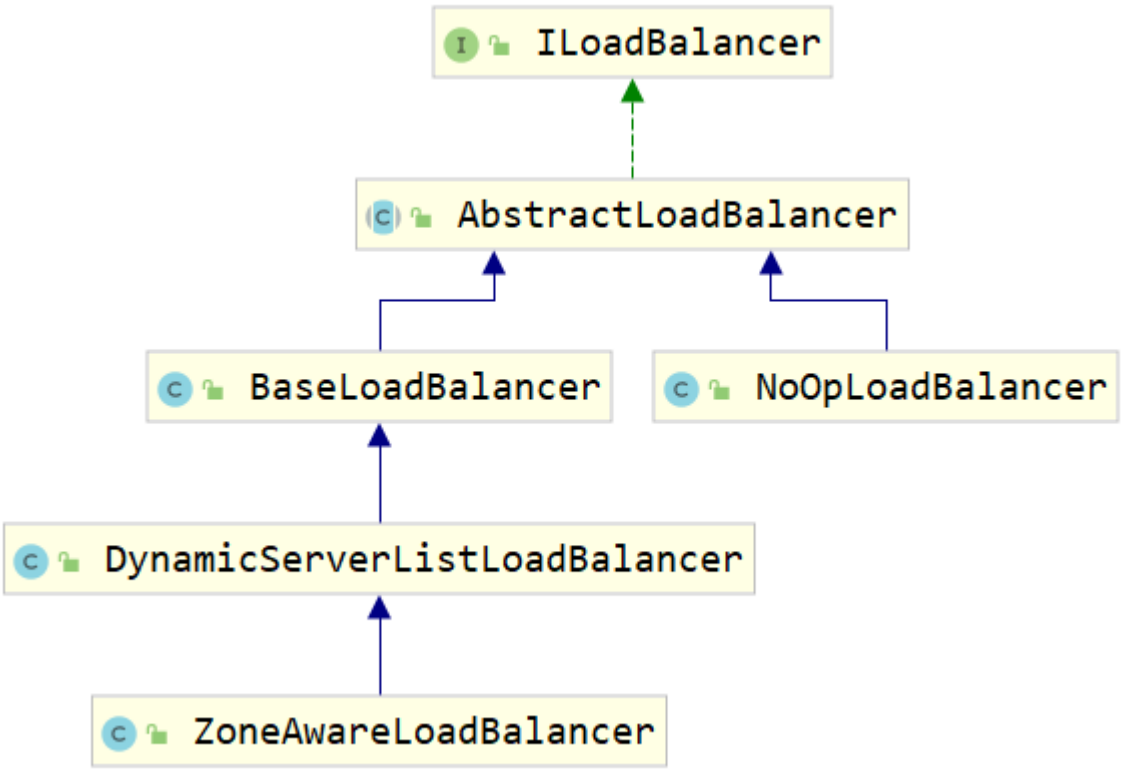
- LoadBalancerClient中的方法

```
public interface LoadBalancerClient extends ServiceInstanceChooser {
    <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws IOException;

    <T> T execute(String serviceId, ServiceInstance serviceInstance,
            LoadBalancerRequest<T> request) throws IOException;

    URI reconstructURI(ServiceInstance instance, URI original);

}
```

16.4.3 ILoadBalancer接口

> Ribbon的入口

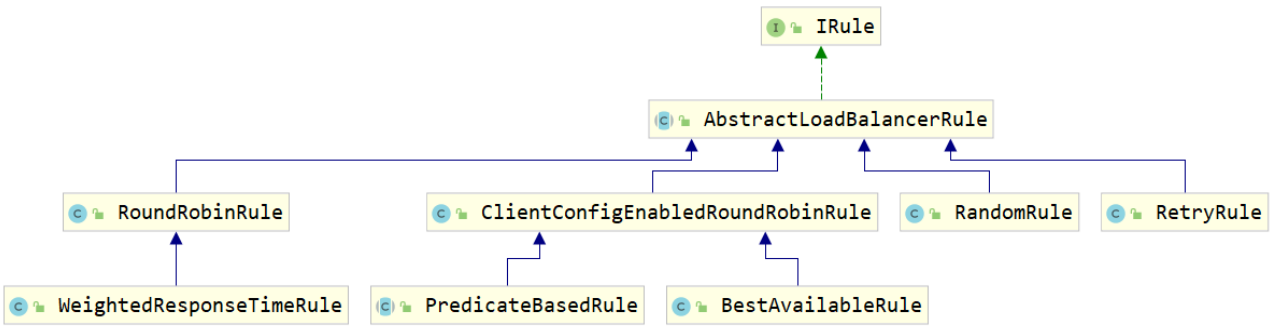
```
public interface ILoadBalancer {
    // 向负载均衡器中维护的实例列表增加服务实例
    public void addServers(List<Server> newServers);

    // 通过某种策略，从负载均衡器中挑选出一个具体的服务实例
    public Server chooseServer(Object key);

    // 用来通知和标识负载均衡器中某个具体实例已经停止服务
    // 不然负载均衡器在下一次获取服务实例清单前都会认为服务实例均是正常服务的
    public void markServerDown(Server server);

    // 获取当前服务的实例列表
    @Deprecated
    public List<Server> getServerList(boolean availableOnly);

    // 获取当前正常服务的实例列表
    public List<Server> getReachableServers();

    // 获取所有已知的服务实例列表，包括正常服务和停止服务的实例
    public List<Server> getAllServers();
}
```



16.6.4 IRule

> 负载均衡的规则，选择实例

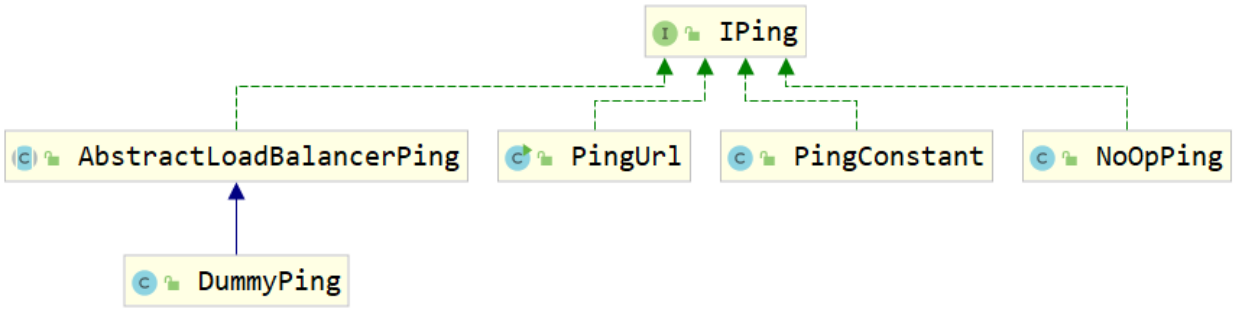
```
public interface IRule{
    public Server choose(Object key);
    public void setLoadBalancer(ILoadBalancer lb);
    public ILoadBalancer getLoadBalancer();
}
```



16.4.5 IPing

> 检查某个Server是否可以ping通，从而达到过滤的条件

```
public interface IPing {
    public boolean isAlive(Server server);
}
```



16.4.6 LoadBalancerAutoCon..

> 添加了@LoadBalanced注解后，Spring Boot会通过LoadBalancerAutoConfiguration完成自动装配
>
> 其中包括拦截器interceptors的装配，这边要联想到之前的restTemplate源码中的拦截器

```
@Configuration
@ConditionalOnClass(RestTemplate.class)
@ConditionalOnBean(LoadBalancerClient.class)
@EnableConfigurationProperties(LoadBalancerRetryProperties.class)
public class LoadBalancerAutoConfiguration {
    // 创建LoadBalancerInterceptor
    @Bean
    public LoadBalancerInterceptor ribbonInterceptor(
        LoadBalancerClient loadBalancerClient,
        LoadBalancerRequestFactory requestFactory) {
        return new LoadBalancerInterceptor(loadBalancerClient, requestFactory);
    }
    // 给RestTemplate添加LoadBalancerInterceptor
    @Bean
    @ConditionalOnMissingBean
    public RestTemplateCustomizer restTemplateCustomizer(
        final LoadBalancerInterceptor loadBalancerInterceptor) {
        return restTemplate -> {
            List<ClientHttpRequestInterceptor> list = new ArrayList<>(
                restTemplate.getInterceptors());
            list.add(loadBalancerInterceptor);
            restTemplate.setInterceptors(list);
        };
    }
}
```

16.4.7 LoadBalancerInterceptor

> 断点+访问：final URI originalUri = request.getURI();

```
return this.loadBalancer.execute(serviceName,this.requestFactory.createRequest(request, body,
execution));
```

> loadBalancer.execute()->LoadBalancerClient#execute->RibbonLoadBalancerClient#execute
>
> 正好这块也验证了之前@LoadBalancer这个注解是通过LoadBalancerClient来配置的意义

> 根据service-id获得具体的实例
>
> **断点+调试执行**

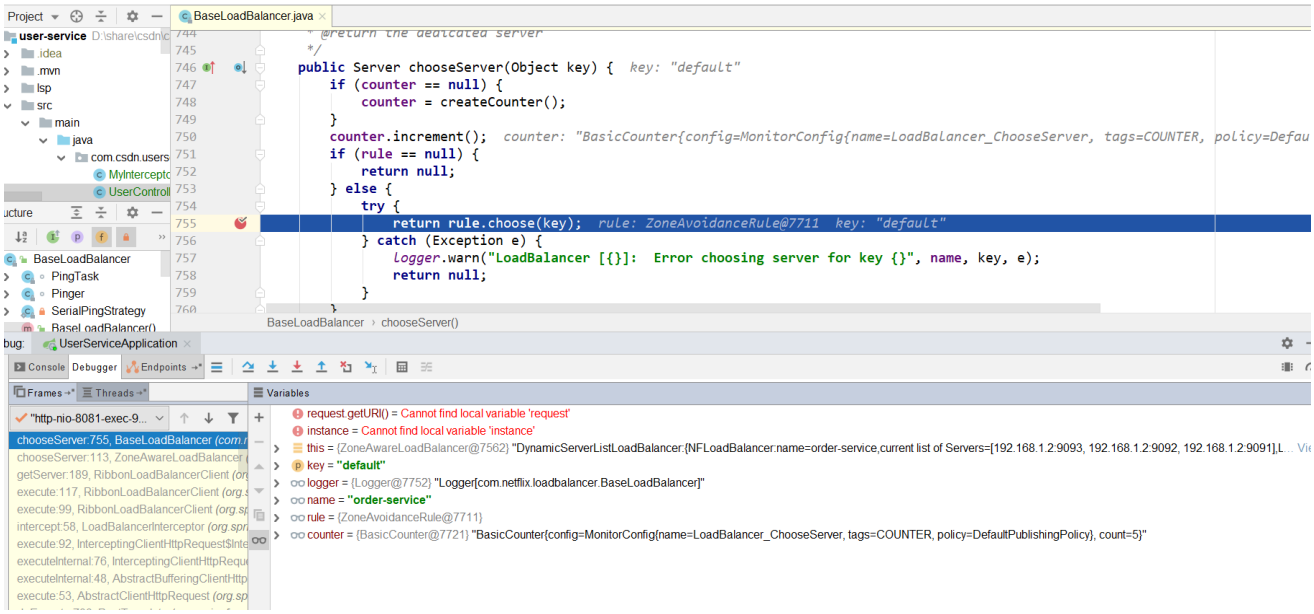
```
ILoadBalancer loadBalancer = getLoadBalancer(serviceId);
Server server = getServer(loadBalancer, hint);
```

getServer()->loadBalancer#chooseServer->ILoadBalancer#chooseServer->默认ZoneAwareLoadBalancer#chooseServer[**可以查看RibbonClientConfiguration配置类**]

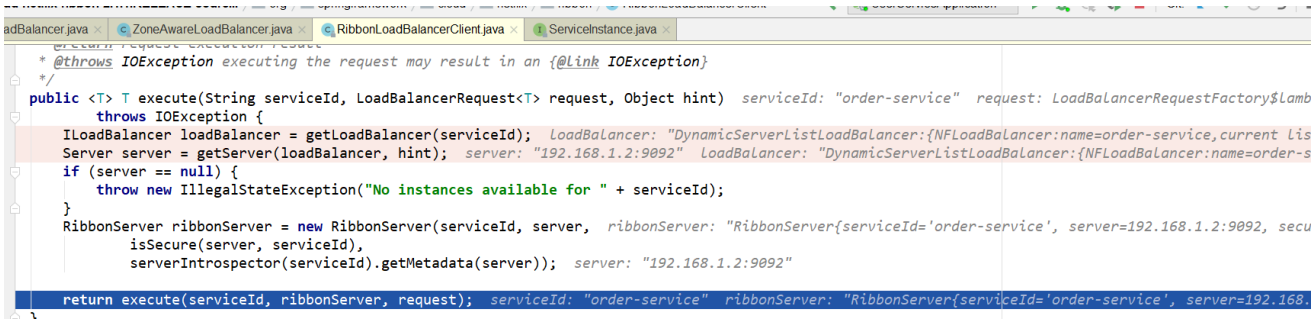
> 但是要注意，要是没有Zone空间，则调用父类的chooseServer方法：BaseLoadBalancer

BaseLoadBalancer#chooseServer->IRule#choose->RoundRobinRule#choose

> 此时在BaseLoadBalancer中choose方法上打个断点就能发现服务列表已经获取到了
>
> 获取的步骤是在RibbonLoadBalancerClient#execute中的
>
> ILoadBalancer loadBalancer = getLoadBalancer(serviceId);



> 然后来到RibbonLoadBalancerClient#execute方法，在执行完第1,2步之后，肯定会获取到一个Server，并且会包装成RibbonServer



16.4.8 LoadBalancerInter..回调

> 当获取到RibbonServer之后，会回调apply方法
>
> 注意RibbonServer实现了ServerInstance接口

```
public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint)
            throws IOException {
    return execute(serviceId, ribbonServer, request);
}
```

> 此时已经获取到具体的Server了，接下来就是要进行uri的重构，然后发起调用请求
>
> **断点+调试**

```
@Override
public <T> T execute(String serviceId, ServiceInstance serviceInstance,
                     LoadBalancerRequest<T> request) throws IOException {
    T returnVal = request.apply(serviceInstance);
}
```

> 在ServerInstance中维护了serviceid、host、port等信息，所以完成可以进行相应的重构条件的基础准备

```
public interface LoadBalancerRequest<T> {
    T apply(ServiceInstance instance) throws Exception;
}
```

LoadBalancerRequest#apply->ServiceRequestWrapper->ServiceRequestWrapper#getURI

> 下面的loadBalancer就是LoadBalancerClient
>
> 因此就能重构出类似于这样的uri：http://192.168.1.2:9093/order/query

```
URI uri = this.loadBalancer.reconstructURI(this.instance, getRequest().getURI());
```

16.5 Feign源码解析

> 回想一个我们使用feign的经历，添加依赖&写注解&写接口，依赖暂且先不说
>
> @EnableFeignClients：开启Feign

> @FeignClient：标记需要用Feign拦截的请求接口

16.5.1 为什么要用feign

> 如果不用feign，我们可以使用restTemplate

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

```
String result = this.restTemplate.getForObject("http://localhost:9091/order/query",
String.class);
```

把url写到代码中，这种方式其实不便于我们阅读，也不符合开发习惯。

而feign的声明式http调用的方式就特别符合开发习惯，对于开发者而言，在consumer中调用provider中的方法，就好像在写本地的接口一样。

16.5.2 动态代理类

> feignclient接口OrderServiceFeignClient

```
@FeignClient(name="order-service")
public interface OrderServiceFeignClient {
    @GetMapping("/order/query")
    public String query();
}
```

> UserController中调用

```
@Autowired
private OrderServiceFeignClient orderServiceFeignClient;
// 测试feign
@RequestMapping("/testfeign")
public String testfeign(){
    return orderServiceFeignClient.query();
}
```

思考：接口并不能实例化，query方法也没有实现，为何能通过接口来调用query方法？

> 不妨在`return orderServiceFeignClient.query();`处打个断点，观察一下接口到底是什么

```
$Proxy77@8030 --->由此可以推断出，该接口通过动态代理做了实现，得到一个动态代理类，并且实现了query方法
```

联想：先从感性的角度思考一下，根据我们之前的开发经验，是不是也有框架这么做过？比如MyBatis中的Mapper接口

```
@Repository
@Mapper
public interface PersonMapper {
    Person find(String username);
}
```

```
@Service
public class PersonService {
    @Autowired
    private PersonMapper personMapper;
    public Person findByUsername(String username) {
        return personMapper.find(username);
    }
}
```

16.5.3 哪些接口要生成动态代理类

- 标注@FeignClient的接口

```
@FeignClient(name="order-service")
public interface OrderServiceFeignClient {
}
```

- 扫描指定的包

```
// 可以指定扫描哪些包路径下的接口，看接口是否标注了@FeignClient注解
@EnableFeignClients
public class UserServiceApplication {
}
```

16.5.4 @EnableFeignClients

> 主要是注解扫描的配置

- @EnableFeignClients

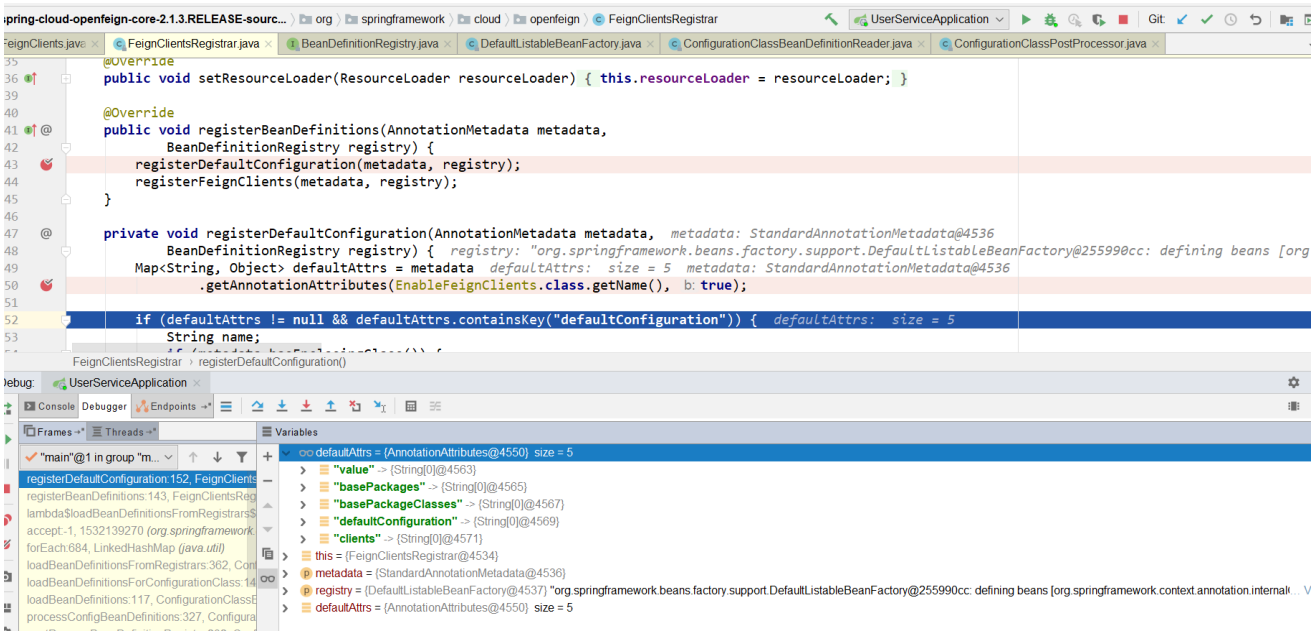
```
// Scans for interfaces that declare they are feign clients (via @FeignClient)
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Import(FeignClientsRegistrar.class)
public @interface EnableFeignClients {
}
```

- @Import(FeignClientsRegistrar.class)

```
class FeignClientsRegistrar
        implements ImportBeanDefinitionRegistrar, ResourceLoaderAware, EnvironmentAware {
    @Override
    public void registerBeanDefinitions(AnnotationMetadata metadata,
            BeanDefinitionRegistry registry) {
        // 注册Configuration
        registerDefaultConfiguration(metadata, registry);
        // 注册@FeignClient注解
        registerFeignClients(metadata, registry);
    }
}
```

16.5.5 r..DefaultConfiguration

> 如图所示，打上两个断点，然后调试



16.5.6 registerFeignClients

- ClassPathScanningCandidateComponentProvider

```
// spring中提供的一个类
A component provider that provides candidate components from a base package
```

- basePackages

```
com.csdn.userservice
```

- AnnotationMetadata

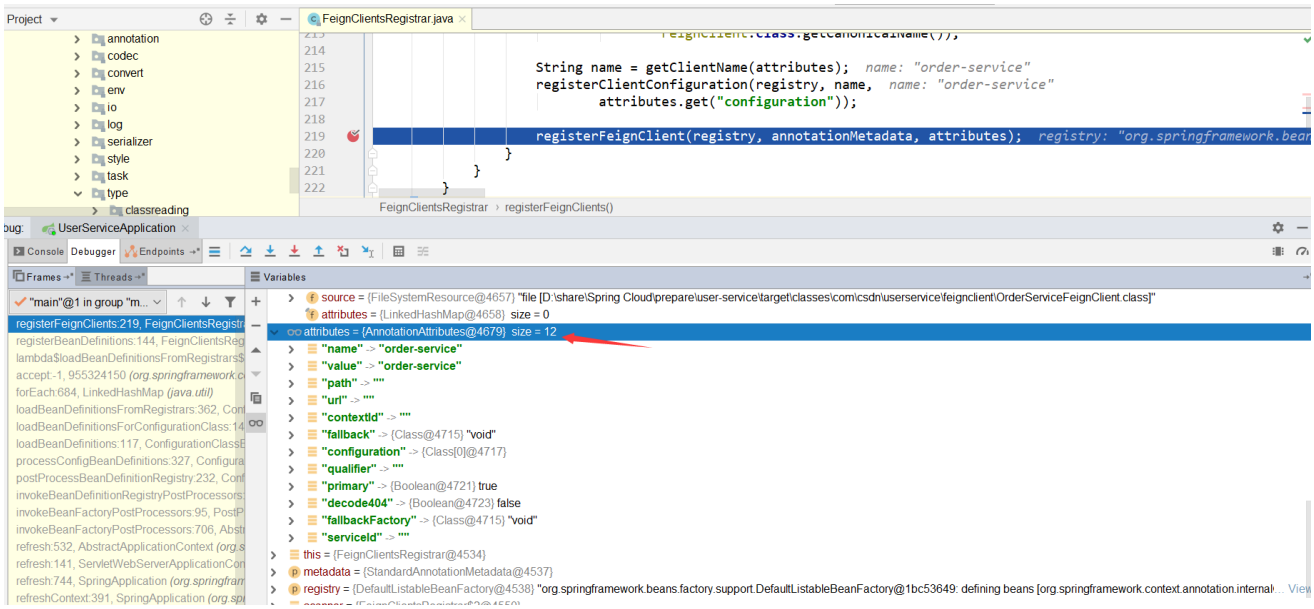
```
Interface that defines abstract access to the annotations of a specific
class, in a form that does not require that class to be loaded yet.
```

- AnnotatedBeanDefinition

```
// ClassPathScanningCandidateComponentProvider 扫描 basePackages下的类是否有标注@FeignClient
// 如果有，通过AnnotatedBeanDefinition进行返回
// AnnotatedBeanDefinition中包含了@FeignClient注解的属性，来源于AnnotationMetadata
Extended interface that exposes about its bean class - without requiring the class to be loaded
yet.
```

16.5.7 registerFeignClient

> attributes



> 代理类的创建
>
> **思考**：代理类肯定也是Bean，那怎样创建一个Bean，而且能定义一个Bean的各种创建过程呢？->FactoryBean

```
// Interface to be implemented by objects used within a {@link BeanFactory} which
public interface FactoryBean<T> {
    /*
       Return an instance (possibly shared or independent) of the object
       managed by this factory.
    */
    @Nullable
    T getObject() throws Exception;
}
```

```
BeanDefinitionBuilder definition = BeanDefinitionBuilder
                .genericBeanDefinition(FeignClientFactoryBean.class);
```

```
class FeignClientFactoryBean
        implements FactoryBean<Object>, InitializingBean, ApplicationContextAware {
    // 返回对象实例
    @Override
    public Object getObject() throws Exception {
        return getTarget();
    }
}
```

16.5.8 getTarget()

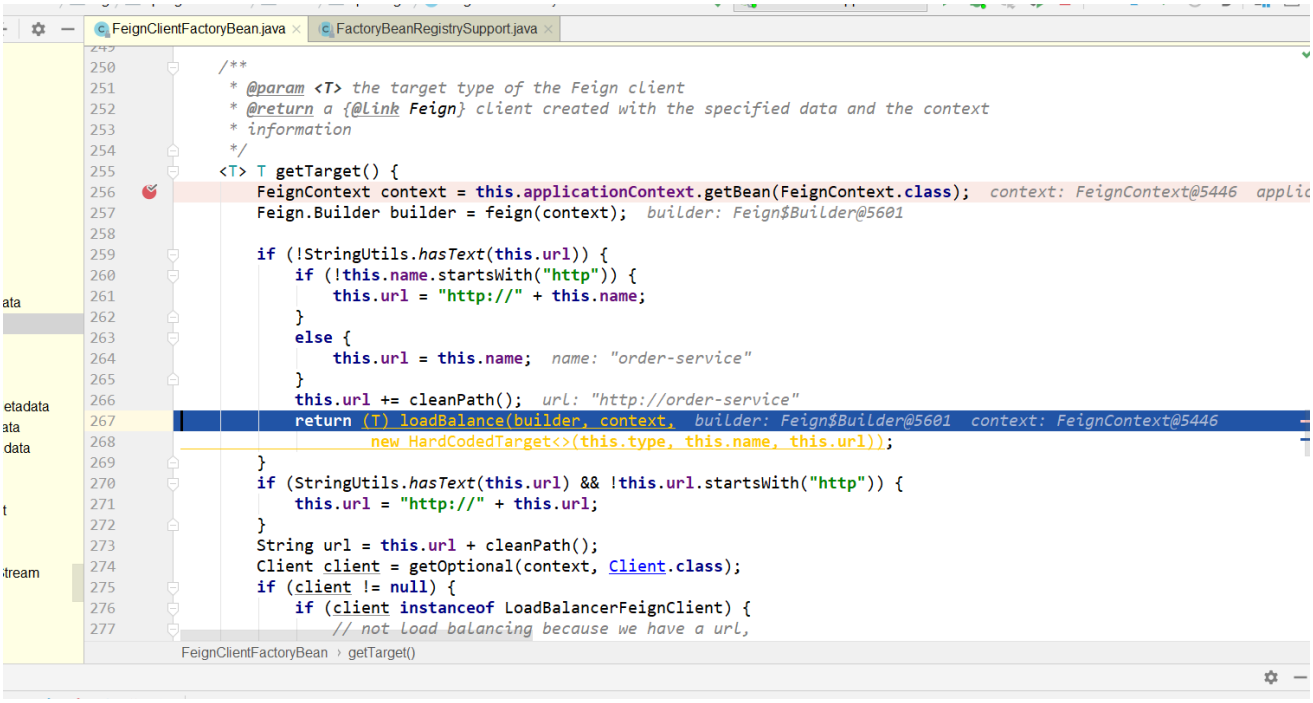
> 生成Feign接口的代理对象

- 代理类对象前置准备之Feign.Builder builder

```
private final List<RequestInterceptor> requestInterceptors =
    new ArrayList<RequestInterceptor>();
private Logger.Level logLevel = Logger.Level.NONE;
private Contract contract = new Contract.Default();
private Client client = new Client.Default(null, null);
private Retryer retryer = new Retryer.Default();
private Logger logger = new NoOpLogger();
private Encoder encoder = new Encoder.Default();
private Decoder decoder = new Decoder.Default();
private QueryMapEncoder queryMapEncoder = new QueryMapEncoder.Default();
private ErrorDecoder errorDecoder = new ErrorDecoder.Default();
private Options options = new Options();
private InvocationHandlerFactory invocationHandlerFactory =
    new InvocationHandlerFactory.Default();
private boolean decode404;
private boolean closeAfterDecode = true;
private ExceptionPropagationPolicy propagationPolicy = NONE;
```

- 代理类对象前置准备之uri

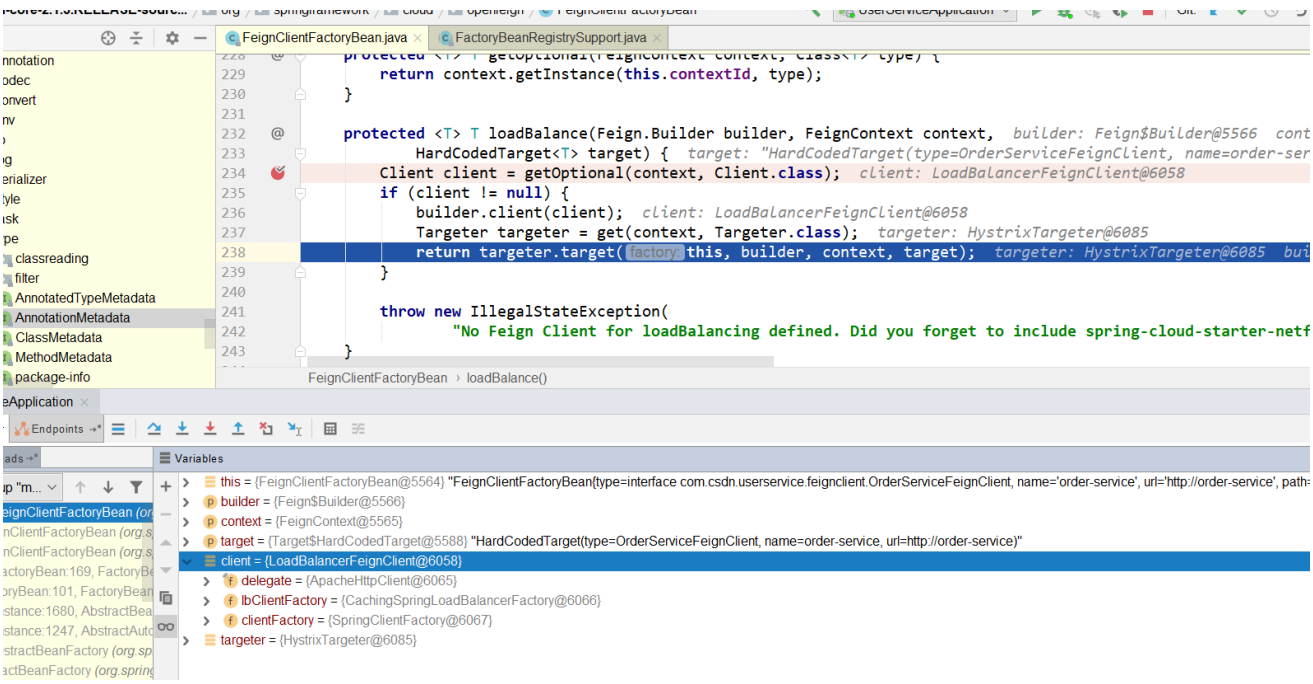
肯定需要有目标地址的uri，这个uri其实就是从接口的元数据中获得的，不妨这边调试一下



- 代理类对象前置准备之负载均衡

仅仅拿到http://order-service是不够的，因为这里的order-service不能直接访问，肯定需要做负载均衡，默认使用的是ribbon的方式

> 对于ribbon的负载均衡方式之前已经分享过，就不做过多的赘述咯
>
> 最终会获得一个类似于http://192.168.0.3:9091的地址，也就是将order-service做了替换



- 生成代理对象

```
// getTarget
return (T) targeter.target(this, builder, context,
      new HardCodedTarget<>(this.type, this.name, url));
```

```
Targeter#target
```

```
DefaultTargeter#target
```

```
feign.target(target)
```

```
build().newInstance(target)
```

> build()
>
> ```
> public Feign build() {
> SynchronousMethodHandler.Factory synchronousMethodHandlerFactory =
>   new SynchronousMethodHandler.Factory(client, retryer, requestInterceptors, logger,
>                                        logLevel, decode404, closeAfterDecode,
> propagationPolicy);
> ParseHandlersByName handlersByName =
>   new ParseHandlersByName(contract, options, encoder, decoder, queryMapEncoder,
>                           errorDecoder, synchronousMethodHandlerFactory);
> return new ReflectiveFeign(handlersByName, invocationHandlerFactory, queryMapEncoder);
> }
> ```

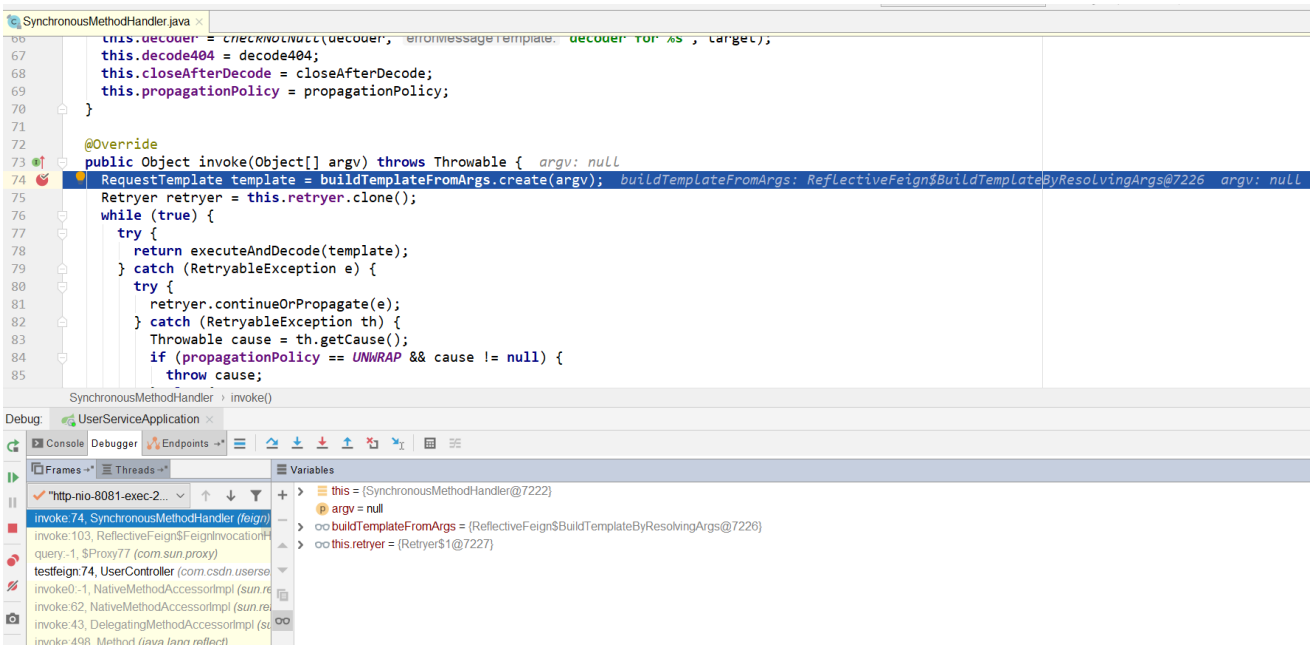
```
ReflectiveFeign#newInstance
```

```
targetToHandlersByName.apply(target)
```

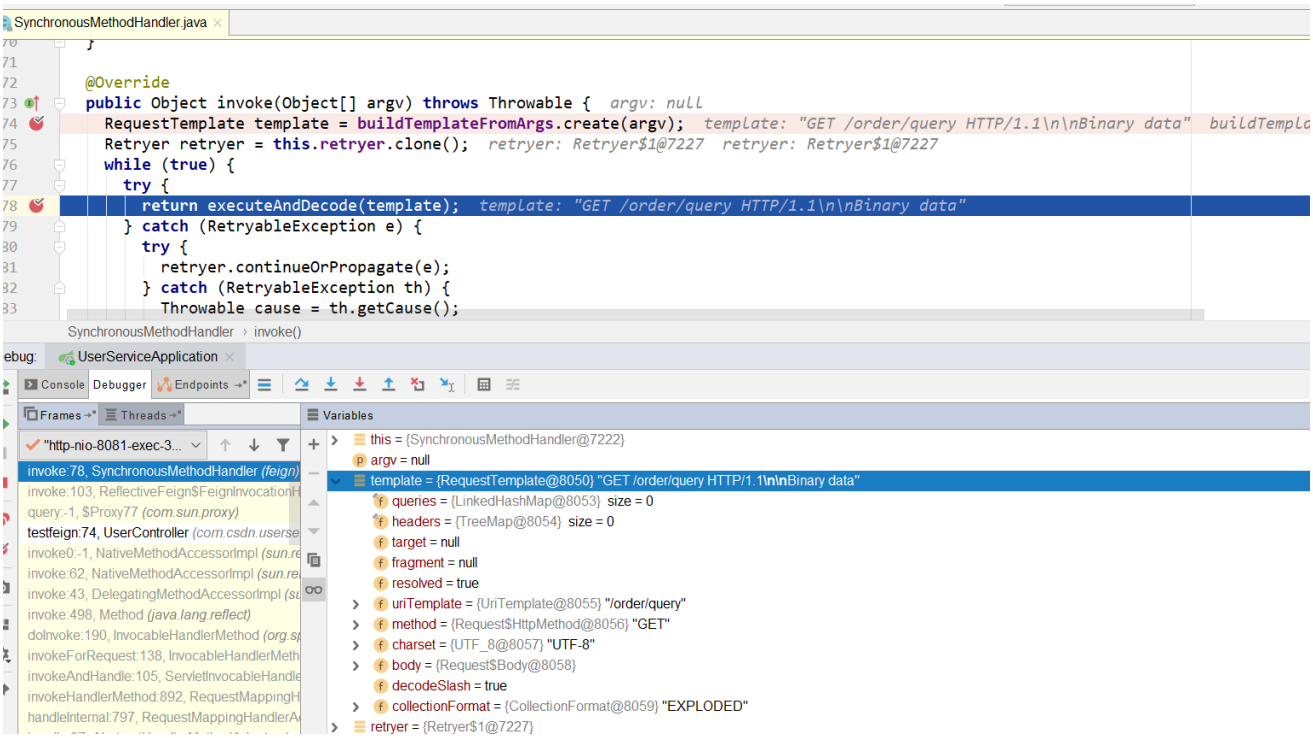
> 其实最终调用的是SynchronousMethodHandler的invoke方法[反射]

16.5.9 调用invoke

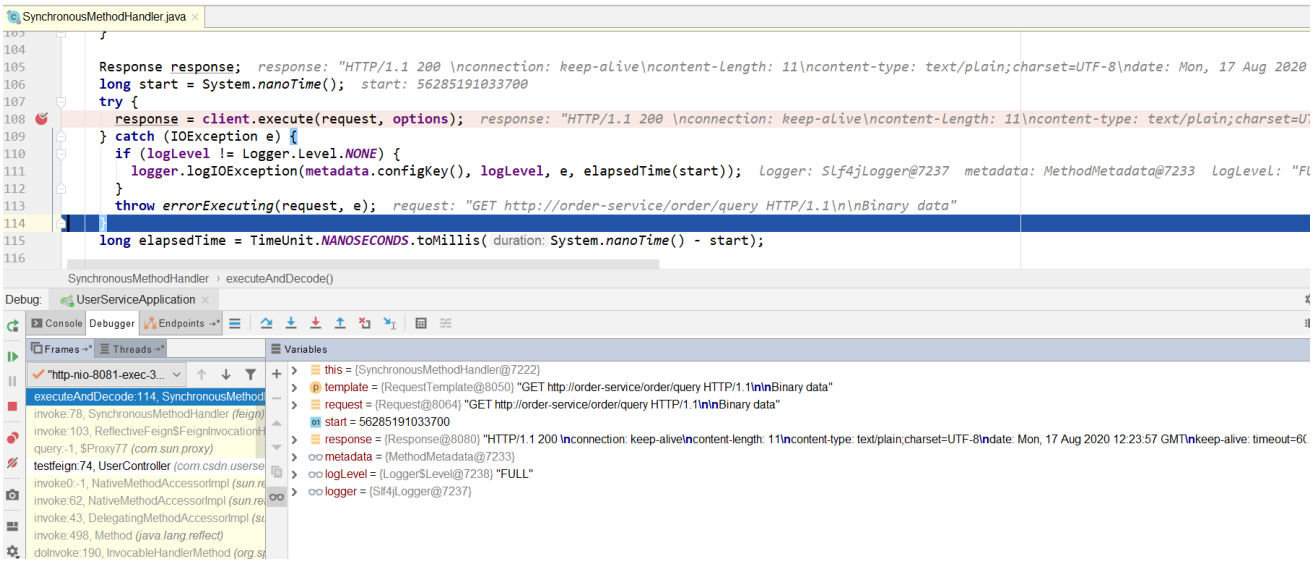
> 在SynchronousMethodHandler的invoke方法上打个断点，然后访问testfeign接口



> SynchronousMethodHandler#invoke#executeAndDecode



> SynchronousMethodHandler#executeAndDecode#client.execute
>
> 可以发现client是LoadBalancerFeignClient类型，更加说明其做了服务发现与负载均衡



16.5.10 其他思考

> **可以基于Feign.Builder来切入**

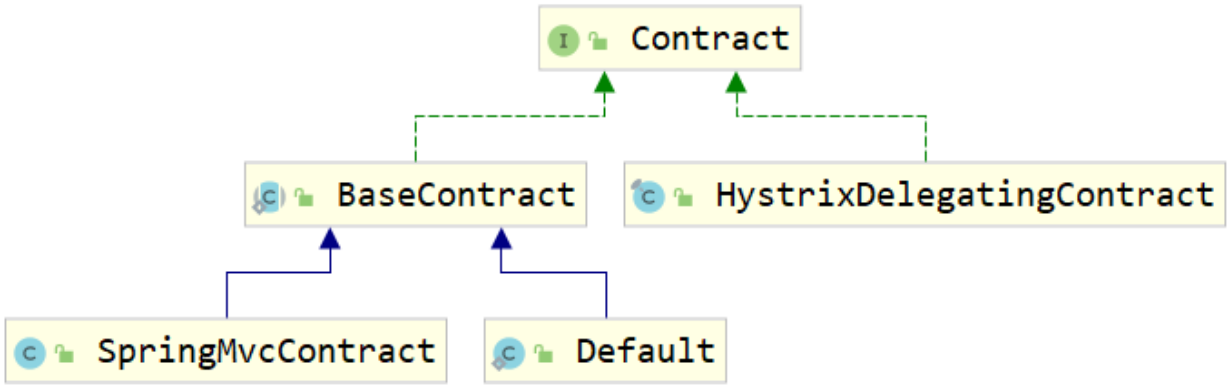
- Contract

> Feign是如何支持Spring MVC的注解的?
>
> 比如@RequestMapping

```
private Contract contract = new Contract.Default();
```

```
/**
 * Defines what annotations and values are valid on interfaces.
 */
public interface Contract {
    List<MethodMetadata> parseAndValidatateMetadata(Class<?> targetType);
}
```

```
public class SpringMvcContract extends Contract.BaseContract
        implements ResourceLoaderAware {
    // 比如RequestMapping
    @Override
    protected void processAnnotationOnMethod(MethodMetadata data,
            Annotation methodAnnotation, Method method) {
        if (!RequestMapping.class.isInstance(methodAnnotation) && !methodAnnotation
                .annotationType().isAnnotationPresent(RequestMapping.class)) {
            return;
        }
    // ...
}
```



- 序列化和返回序列化

> 从上述源码的过程可以看出，发起的是Request请求，接收的是Response响应，那这种JavaBean在网络中是如何进行传输的?

```
private Encoder encoder = new Encoder.Default();
private Decoder decoder = new Decoder.Default();
```

```
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 class Default extends StringDecoder {
    @Override
    public Object decode(Response response, Type type) throws IOException {
        if (response.status() == 404 || response.status() == 204)
            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);
    }
}
```

- Feign的日志管理

> 当user-service通过feign调用order-service的时候，发现是没有任何日志打印的，可以配置一下，然后访问观察控制台
>
> ```
> private Logger logger = new NoOpLogger();
> ```

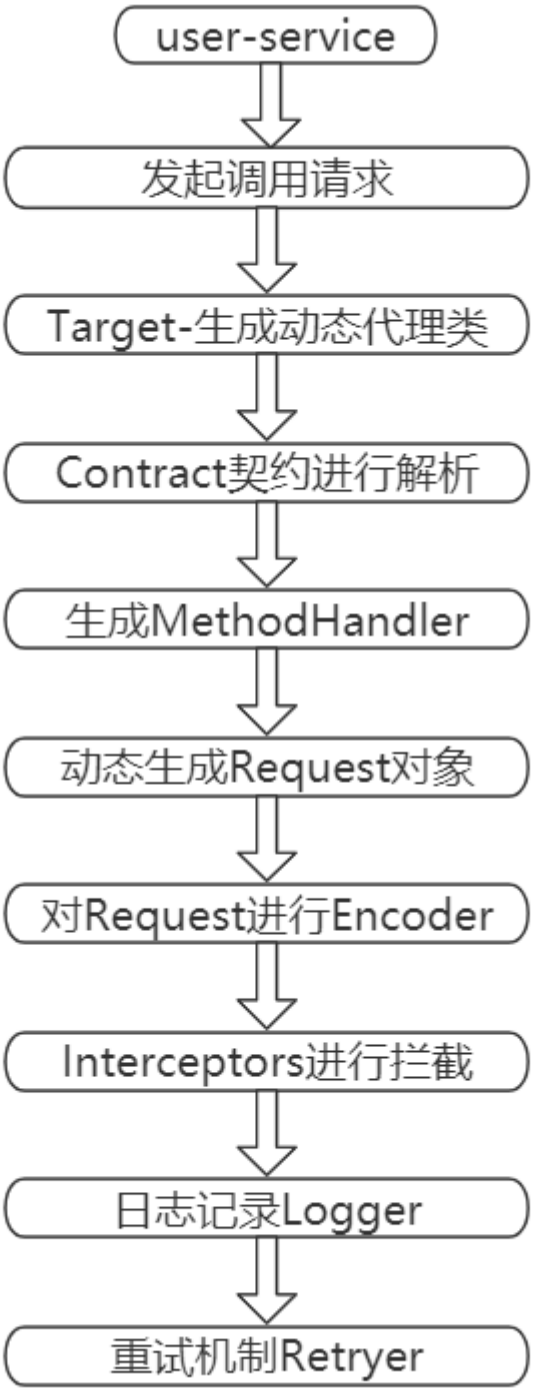
```
public enum Level {
    NONE,
    BASIC,
    HEADERS,
    FULL
}
```

- Feign的重拾机制

> 如果调用不成功，重拾机制是如何的?
>
> ```
> public interface Retryer extends Cloneable {
> }
> ```

16.5.11 Feign整体调用流程

> 好了，有了上面的源码分析之后，我们大概知道了feign的调用流程。
>
> 本质上就是解析标注了@FeignClient注解的接口中的元数据，然后通过该接口的动态代理类完成调用返回，封装了http请求和返回的流程，其中过程中可能会经历各种拦截操作，仅此而已。



17 Docker

17.1 部署官方镜像之tomcat

docker run -d --name tomcat01 -p 6060:8080 tomcat

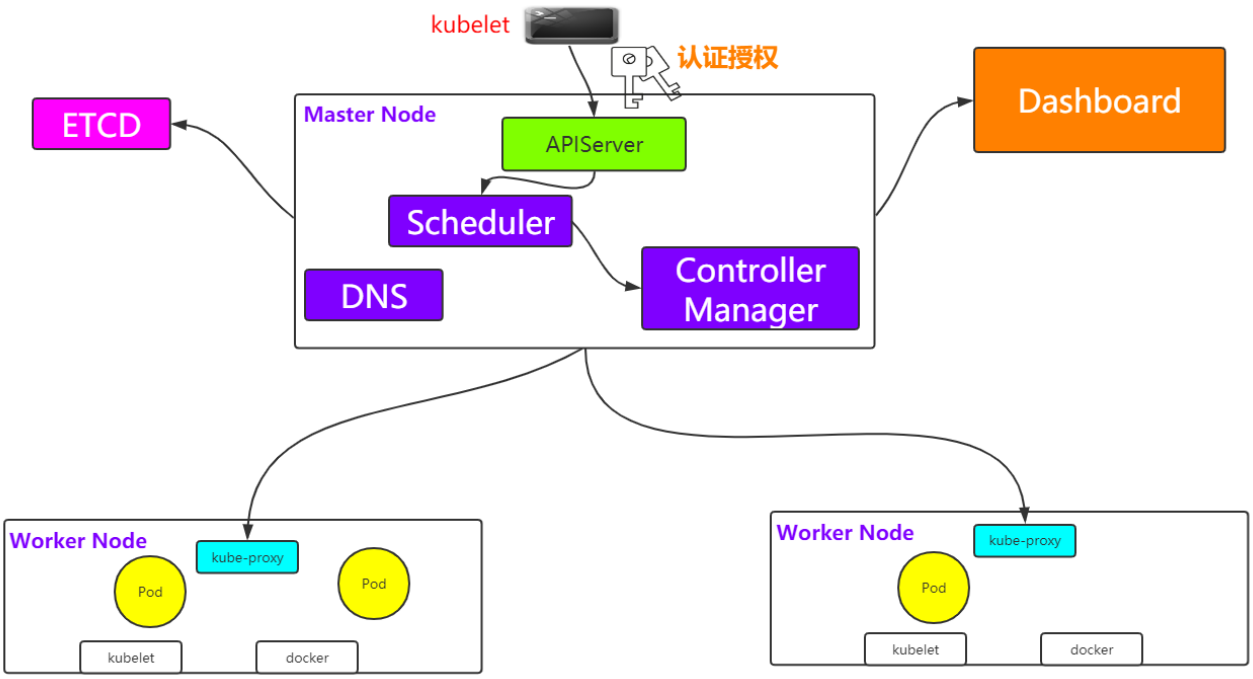
17.2 部署Spring Boot项目

> 这里以一个简单的SpringBoot项目为例

(1)在本地测试该项目的功能

```
    修改端口为8888
    @RequestMapping("/ops")
    public String ops(){
        return "Spring Cloud Ops";
    }

(2)在项目根目录下执行mvn clean package打成一个jar包
    mvn clean package -Dmaven.test.skip=true
    在target下找到"xxx.jar"

(3)在docker环境中新建一个目录"springboogt-demo"

(4)安装文件传输工具yum install lrzsz，然后上传"xxx.jar"到该目录下，并且在此目录创建Dockerfile

(5)编写Dockerfile内容
    FROM openjdk:8
    MAINTAINER itcrazy2016
    LABEL name="springboot-demo" version="1.0" author="itcrazy2016"
    COPY xxx.jar springboot-intro.jar
    CMD ["java","-jar","springboot-demo.jar"]

(6)基于Dockerfile构建镜像
    docker build -t sbd .

(7)上传镜像到镜像仓库
    docker tag sbd registry.cn-hangzhou.aliyuncs.com/itcrazy2016/sbd:v1
    docker login --username=itcrazy2016@163.com registry.cn-hangzhou.aliyuncs.com
    docker push registry.cn-hangzhou.aliyuncs.com/itcrazy2016/sbd:v1

(7)基于image创建container
    docker run -d --name sb01 -p 8081:8080 sbd

(8)查看启动日志docker logs sb01

(9)在浏览器访问http://192.168.2.11:8081/ops
```

18 K8s



> 基于kubeadm的3台K8s集群，一台master，两台worker

18.1 安装vagrant和virtualbox

18.1.1 下载安装vagrant

```
01 访问Vagrant官网
https://www.vagrantup.com/

02 点击Download
Windows, MacOS, Linux等

03 选择对应的版本

04 傻瓜式安装

05 命令行输入vagrant，测试是否安装成功
```

18.1.2 下载安装virtual box

```
01 访问VirtualBox官网
        https://www.virtualbox.org/

02 选择左侧的“Downloads”

03 选择对应的操作系统版本

04 傻瓜式安装

05 [win10中若出现]安装virtualbox快完成时立即回滚，并提示安装出现严重错误
    (1)打开服务
    (2)找到Device Install Service和Device Setup Manager，然后启动
    (3)再次尝试安装
```

18.1.3 安装centos7

```
01 创建centos7文件夹，并进入其中[不要有中文字符]

02 在此目录下打开cmd，运行vagrant init centos/7
   此时会在当前目录下生成Vagrantfile

03 运行vagrant up[注意不要运行，拉取远端的centos7太慢]
   此时会找centos7的镜像，本地有就用本地的，本地没有就会拉取远端的

04 准备centos7的box
        (1)选中命令行中提示的链接
      比如https://vagrantcloud.com/centos/boxes/7/versions/1905.1/providers/virtualbox.box
    (2)复制到迅雷中下载
    (3)vagrant box add centos/7 D:\迅雷下载\virtualbox.box
    (4)vagrant box list  查看本地的box[这时候可以看到centos/7]

05 根据本地的centos7 box创建虚拟机
        vagrant up[打开virtual box，可以发现centos7创建成功]

06 vagrant基本操作
        (1)vagrant ssh
        进入刚才创建的centos7中
    (2)vagrant status
        查看centos7的状态
    (3)vagrant halt
        停止centos7
    (4)vagrant destroy
        删除centos7
    (5)vagrant status
        查看当前vagrant创建的虚拟机
    (6)Vagrantfile中也可以写脚本命令，使得centos7更加丰富
        但是要注意，修改了Vagrantfile，要想使正常运行的centos7生效，必须使用vagrant reload
```

> 至此，使用vagrant+virtualbox搭建centos7完成，后面可以修改Vagrantfile对虚拟机进行相应配置

18.1.4 若想通过Xshell连接

```
01 查看centos7相关信息
        vagrant ssh-config
        关注:Hostname  Port  IdentityFile

02 Xshell连接
        IP:127.0.0.1
        port:2222
        用户名:vagrant
        密码:vagrant
        文件:Identityfile指向的路径

03 使用root账户登录
        sudo -i
        vi /etc/ssh/sshd_config
        修改PasswordAuthentication yes
        passwd修改密码
        systemctl restart sshd
        root vagrant 登录
```

18.1.5 Vagrantfile通用写法

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# All Vagrant configuration is done below. The "2" in Vagrant.configure
# configures the configuration version (we support older styles for
# backwards compatibility). Please don't change it unless you know what
# you're doing.
Vagrant.configure("2") do |config|
  # The most common configuration options are documented and commented below.
  # For a complete reference, please see the online documentation at
  # https://docs.vagrantup.com.

  # Every Vagrant development environment requires a box. You can search for
  # boxes at https://vagrantcloud.com/search.
  config.vm.box = "centos/7"

  # Disable automatic box update checking. If you disable this, then
  # boxes will only be checked for updates when the user runs
  # `vagrant box outdated`. This is not recommended.
  # config.vm.box_check_update = false

  # Create a forwarded port mapping which allows access to a specific port
  # within the machine from a port on the host machine. In the example below,
  # accessing "localhost:8080" will access port 80 on the guest machine.
  # NOTE: This will enable public access to the opened port
  # config.vm.network "forwarded_port", guest: 80, host: 8080

  # Create a forwarded port mapping which allows access to a specific port
  # within the machine from a port on the host machine and only allow access
  # via 127.0.0.1 to disable public access
  # config.vm.network "forwarded_port", guest: 80, host: 8080, host_ip: "127.0.0.1"
```

```
  # Create a private network, which allows host-only access to the machine
  # using a specific IP.
  # config.vm.network "private_network", ip: "192.168.33.10"

  # Create a public network, which generally matched to bridged network.
  # Bridged networks make the machine appear as another physical device on
  # your network.
  config.vm.network "public_network"

  # Share an additional folder to the guest VM. The first argument is
  # the path on the host to the actual folder. The second argument is
  # the path on the guest to mount the folder. And the optional third
  # argument is a set of non-required options.
  # config.vm.synced_folder "../data", "/vagrant_data"

  # Provider-specific configuration so you can fine-tune various
  # backing providers for Vagrant. These expose provider-specific options.
  # Example for VirtualBox:
  #
  # config.vm.provider "virtualbox" do |vb|
  #   # Display the VirtualBox GUI when booting the machine
  #   vb.gui = true
  #
  #   # Customize the amount of memory on the VM:
  #   vb.memory = "1024"
  # end
    config.vm.provider "virtualbox" do |vb|
        vb.memory = "4000"
        vb.name= "jack-centos7"
        vb.cpus= 2
    end
  #
  # View the documentation for the provider you are using for more
  # information on available options.

  # Enable provisioning with a shell script. Additional provisioners such as
  # Puppet, Chef, Ansible, Salt, and Docker are also available. Please see the
  # documentation for more information about their specific syntax and use.
  # config.vm.provision "shell", inline: <<-SHELL
  #   apt-get update
  #   apt-get install -y apache2
  # SHELL
end
```

18.2 准备3台centos

> (1)在labs下创建一个文件夹

```
kubeadm-k8s-cluster-centos7
```

> (2)新建Vagrantfile

```
boxes = [
    {
        :name => "master-kubeadm-k8s",
        :eth1 => "192.168.2.51",
        :mem => "2048",
        :cpu => "2"
    },
    {
        :name => "worker01-kubeadm-k8s",
        :eth1 => "192.168.2.61",
        :mem => "2048",
        :cpu => "2"
    },
    {
        :name => "worker02-kubeadm-k8s",
        :eth1 => "192.168.2.62",
        :mem => "2048",
        :cpu => "2"
    }
]

Vagrant.configure(2) do |config|

  config.vm.box = "centos/7"

   boxes.each do |opts|
      config.vm.define opts[:name] do |config|
        config.vm.hostname = opts[:name]
        config.vm.provider "vmware_fusion" do |v|
          v.vmx["memsize"] = opts[:mem]
          v.vmx["numvcpus"] = opts[:cpu]
        end

        config.vm.provider "virtualbox" do |v|
          v.customize ["modifyvm", :id, "--memory", opts[:mem]]
          v.customize ["modifyvm", :id, "--cpus", opts[:cpu]]
          v.customize ["modifyvm", :id, "--name", opts[:name]]
        end

        config.vm.network :public_network, ip: opts[:eth1]
      end
  end

end
```

(3)使用XShell连接3个节点，记得进入分别的机器修改密码：vagrant ssh master-kubeadm-k8s

```
sudo -i
vi /etc/ssh/sshd_config
PasswordAuthentication yes
passwd
systemctl restart sshd
```

18.3 更新并安装依赖

> 在XShell中，选择“查看”-->“撰写”-->“撰写窗口”-->”全部会话“

```
yum -y update
yum install -y conntrack ipvsadm ipset jq sysstat curl iptables libseccomp
```

18.4 安装Docker

```
01 安装必要的依赖
    sudo yum install -y yum-utils \
    device-mapper-persistent-data \
    lvm2

02 设置docker仓库
    sudo yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
【设置阿里云镜像加速器】
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "registry-mirrors": ["https://orptaaqe.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload

03 安装docker

  yum install -y docker-ce docker-ce-cli containerd.io

04 启动docker
    sudo systemctl start docker && sudo systemctl enable docker

05 卸载docker
    systemctl stop docker

    yum list installed|grep docker
    [
        yum -y remove containerd.io.x86_64
        yum -y remove docker-ce.x86_64
        yum -y remove docker-ce-cli.x86_64
    ]
    [
```

```
        sudo yum remove docker \
                  docker-client \
                  docker-client-latest \
                  docker-common \
                  docker-latest \
                  docker-latest-logrotate \
                  docker-logrotate \
                  docker-engine
    ]
```

18.5 修改hosts文件

> (1)master

```
sudo hostnamectl set-hostname m
vi /etc/hosts
192.168.2.51 m
192.168.2.61 w1
192.168.2.62 w2
```

> (2)两个worker

```
sudo hostnamectl set-hostname w1
sudo hostnamectl set-hostname w2
vi /etc/hosts
192.168.0.51 m
192.168.0.61 w1
192.168.0.62 w2
```

> (3)使用ping测试一下

18.6 系统基础前提配置

```
# (1)关闭防火墙
systemctl stop firewalld && systemctl disable firewalld

# (2)关闭selinux
【SELinux 全称 Security Enhanced Linux (安全强化 Linux)，是 MAC (Mandatory Access Control，强制访问控制系统)的一个实现，目的在于明确的指明某个进程可以访问哪些资源(文件、网络端口等)。】

setenforce 0
sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/selinux/config

# (3)关闭swap
【在Linux下，SWAP的作用类似Windows系统下的“虚拟内存”。当物理内存不足时，拿出部分硬盘空间当SWAP分区（虚拟成内存）使用，从而解决内存容量不足的情况。】

swapoff -a
sed -i '/swap/s/^\(.*\)$/#\1/g' /etc/fstab
```

```
# (4)配置iptables的ACCEPT规则
iptables -F && iptables -X && iptables -F -t nat && iptables -X -t nat && iptables -P FORWARD
ACCEPT

# (5)设置系统参数
cat <<EOF >  /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF

sysctl --system
```

18.7 安装kubeadm, kubelet and kubectl

> You will install these packages on all of your machines:
>
> - `kubeadm`: the command to bootstrap the cluster.
> - `kubelet`: the component that runs on all of the machines in your cluster and does things like starting pods and containers.
> - `kubectl`: the command line util to talk to your cluster.

> (1)配置yum源

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=http://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=0
repo_gpgcheck=0
gpgkey=http://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg
       http://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
EOF
```

> (2)安装kubeadm&kubelet&kubectl

```
# 列出版本
yum list kubeadm --showduplicates | sort -r

# 选择版本进行安装
yum install -y kubeadm-1.18.5-0 kubelet-1.18.5-0 kubectl-1.18.5-0
yum install -y kubeadm kubelet kubectl
yum remove -y kubeadm kubelet kubectl
```

> (3)docker和k8s设置同一个

```
# docker
vi /etc/docker/daemon.json
    "exec-opts": ["native.cgroupdriver=systemd"],
systemctl restart docker

# kubelet
sed -i "s/cgroup-driver=systemd/cgroup-driver=cgroupfs/g"
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf

systemctl enable kubelet && systemctl start kubelet
```

18.8 proxy/pause/scheduler等国内镜像

> 查看kubeadm使用的镜像
>
> kubeadm config images list
>
> 记得把这些镜像拉取到每台机器上

```
k8s.gcr.io/kube-apiserver:v1.18.8
k8s.gcr.io/kube-controller-manager:v1.18.8
k8s.gcr.io/kube-scheduler:v1.18.8
k8s.gcr.io/kube-proxy:v1.18.8
k8s.gcr.io/pause:3.2
k8s.gcr.io/etcd:3.4.3-0
k8s.gcr.io/coredns:1.6.7
```

18.9 kube init初始化master

> (1)初始化master节点
>
> 官网：https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/
>
> 注意：**此操作也是在主节点上进行**

```
# 本地有镜像
kubeadm init --kubernetes-version=1.18.8 --apiserver-advertise-address=192.168.2.51 --pod-
network-cidr=10.244.0.0/16
【若要重新初始化集群状态：kubeadm reset，然后再进行上述操作】

# 本地没有镜像，使用自己的阿里云仓库
kubeadm init --kubernetes-version=1.18.8 --apiserver-advertise-address=192.168.2.51 --pod-
network-cidr=10.244.0.0/16 --image-repository [填写自己的镜像仓库地址]
```

> (2)根据日志提示

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

此时kubectl cluster-info查看一下，发现连接成功了，跟之前minikube打印的信息很类似

> (3)查看pod验证一下
>
> 等待一会儿，同时可以发现像etc，controller，scheduler等组件都以pod的方式安装成功了
>
> 注意：coredns没有启动，需要安装网络插件

```
kubectl get pods -n kube-system
```

> (4)健康检查

```
curl -k https://localhost:6443/healthz
返回ok
```

> (5)kube init流程

```
01-进行一系列检查，以确定这台机器可以部署kubernetes

02-生成kubernetes对外提供服务所需要的各种证书可对应目录
/etc/kubernetes/pki/*

03-为其他组件生成访问kube-ApiServer所需的配置文件
    ls /etc/kubernetes/
    admin.conf  controller-manager.conf  kubelet.conf  scheduler.conf

04-为 Master组件生成Pod配置文件。
    ls /etc/kubernetes/manifests/*.yaml
    kube-apiserver.yaml
    kube-controller-manager.yaml
    kube-scheduler.yaml

05-生成etcd的Pod YAML文件。
    ls /etc/kubernetes/manifests/*.yaml
    kube-apiserver.yaml
    kube-controller-manager.yaml
    kube-scheduler.yaml
        etcd.yaml

06-一旦这些 YAML 文件出现在被 kubelet 监视的/etc/kubernetes/manifests/目录下，kubelet就会自动创建这些yaml文件定义的pod，即master组件的容器。master容器启动后，kubeadm会通过检查localhost：6443/healthz这个master组件的健康状态检查URL，等待master组件完全运行起来

07-为集群生成一个bootstrap token

08-将ca.crt等 Master节点的重要信息，通过ConfigMap的方式保存在etcd中，工后续部署node节点使用

09-最后一步是安装默认插件。kubernetes默认kube-proxy和DNS两个插件是必须安装的
```

18.10 部署calico网络插件

> 选择网络插件：https://kubernetes.io/docs/concepts/cluster-administration/addons/

> calico网络插件：https://docs.projectcalico.org/v3.9/getting-started/kubernetes/

> `calico，同样在master节点上操作`

> `kubernetes版本过高，比如是1.18的话`：https://github.com/datawire/ambassador/issues/1848

> `注意`：calico镜像拉取可能拉取比较慢，可以先手动pull一下

```
docker pull calico/pod2daemon-flexvol:v3.9.1
docker pull calico/kube-controllers:v3.9.1
docker pull calico/cni:v3.9.1
```

```
01-确保已经安装了kubeadm
        As a regular user with sudo privileges, open a terminal on the host that you installed
kubeadm on.

02-初始化master节点pod网络
        Initialize the master using the following command.

sudo kubeadm init --pod-network-cidr=192.168.0.0/16

        Note: If 192.168.0.0/16 is already in use within your network you must select a different
pod network CIDR, replacing 192.168.0.0/16 in the above command as well as in any manifests
applied below.

03-kube init之前的配置
    mkdir -p $HOME/.kube
    sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
    sudo chown $(id -u):$(id -g) $HOME/.kube/config

04-安装calico
        Install Calico with the following command.

**【kubectl apply -f https://docs.projectcalico.org/v3.9/manifests/calico.yaml】**

        Note: You can also view the YAML in a new tab.

05-确认一下calico是否安装成功
watch kubectl get pods --all-namespaces

        Wait until each pod has the STATUS of Running.

NAMESPACE    NAME                                       READY  STATUS   RESTARTS  AGE
kube-system  calico-kube-controllers-6ff88bf6d4-tgtzb   1/1    Running  0         2m45s
kube-system  calico-node-24h85                          1/1    Running  0         2m43s
kube-system  coredns-846jhw23g9-9af73                   1/1    Running  0         4m5s
kube-system  coredns-846jhw23g9-hmswk                   1/1    Running  0         4m5s
kube-system  etcd-jbaker-1                              1/1    Running  0         6m22s
kube-system  kube-apiserver-jbaker-1                    1/1    Running  0         6m12s
kube-system  kube-controller-manager-jbaker-1           1/1    Running  0         6m16s
kube-system  kube-proxy-8fzp2                           1/1    Running  0         5m16s
kube-system  kube-scheduler-jbaker-1                    1/1    Running  0         5m41s
```

```
06-查看某个pod的具体状态
       kubectl describe pods -n kube-system pod_name
```

18.11 kube join加入worker

> 记得保存初始化master节点的最后打印信息
>
> 若之前worker node上也join过，需要先`kubeadm reset`

```
Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.0.51:6443 --token yu1ak0.2dcecvmpozsy8loh \
    --discovery-token-ca-cert-hash
sha256:5c4a69b3bb05b81b675db5559b0e4d7972f1d0a61195f217161522f464c307b0
```

> (1)在woker01和worker02上执行上述命令

> (2)在master节点上检查集群信息

```
kubectl get nodes

NAME                   STATUS   ROLES    AGE     VERSION
master-kubeadm-k8s     Ready    master   19m     v1.14.0
worker01-kubeadm-k8s   Ready    <none>   3m6s    v1.14.0
worker02-kubeadm-k8s   Ready    <none>   2m41s   v1.14.0
```

18.12 体验Pod之Nginx

> (1)定义pod.yml文件，比如pod_nginx_rs.yaml

```
cat > pod_nginx_rs.yaml <<EOF
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx
  labels:
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      name: nginx
      labels:
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx
```

```
        ports:
        - containerPort: 80
EOF
```

(2)根据pod_nginx_rs.yml文件创建pod

```
kubectl apply -f pod_nginx_rs.yaml
```

(3)查看pod

```
kubectl get pods
kubectl get pods -o wide
kubectl describe pod nginx
```

(4)感受通过rs将pod扩容

```
kubectl scale rs nginx --replicas=5
kubectl get pods -o wide
```

(5)删除pod

```
kubectl delete -f pod_nginx_rs.yaml
```

18.13 体验Pod之Tomcat

tomcat.yaml文件

```
cat > tomcat.yaml <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tomcat-deployment
  labels:
    app: tomcat
spec:
  replicas: 3
  selector:
    matchLabels:
      app: tomcat
  template:
    metadata:
      labels:
        app: tomcat
    spec:
      containers:
      - name: tomcat
        image: tomcat
        ports:
        - containerPort: 8080
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: tomcat
EOF
```

> 基本操作

```
kubectl apply -f tomcat.yaml
kubectl get pods
kubectl get pods -o wide
curl pod_id:8080
kubectl get svc
curl tomcat-service_ip
kubectl scale deploy tomcat-deployment --replicas=3
```

18.14 体验Pod之Spring Boot

```
# 以Deployment部署Pod
cat > springboot.yaml <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: springboot-demo
spec:
  selector:
    matchLabels:
      app: springboot-demo
  replicas: 1
  template:
    metadata:
      labels:
        app: springboot-demo
    spec:
      containers:
      - name: springboot-demo
        image: registry.cn-hangzhou.aliyuncs.com/itcrazy2016/sbd:v1.0
        ports:
        - containerPort: 8888
---
apiVersion: v1

kind: Service
```

```
metadata:
  name: springboot-demo
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8888
  selector:
    app: springboot-demo
  type: NodePort
EOF
```

> 基本操作

```
kubectl apply -f springboot.yaml
kubectl get pods
kubectl get pods -o wide
curl pod_id:8080/k8s
kubectl get svc
curl springboot-demo-ip
kubectl scale deploy springboot-demo --replicas=5
curl 192.168.2.51:30342
```

19 Service Mesh

> 到此，大家能够感受到Docker与K8s的方便，但是有没有发现Spring Cloud项目在K8s中部署的时候依然有问题需要解决，就是非业务代码相关的问题：通信。

（1）最初是为了业务而写代码，比如登录功能、支付功能等，到后面会发现要解决网络通信的问题，虽然有Spring Cloud里面的组件帮我们解决了，但是细想一下它是怎么解决的？引入以来dependency，加注解，写配置，最后将这些非业务功能的代码打包一起部署，和业务代码在一起，称为“侵入式框架”；
（2）微服务中的服务支持不同语言开发，维护不同语言和非业务代码的成本；
（3）业务代码开发者应该把更多的精力投入到业务熟悉度上，而不应该是非业务上，Spring Cloud虽然能解决微服务领域的很多问题，但是学习成本还是较大的；
（4）对于Spring Cloud而言，并不是微服务领域的所有问题都有对应的解决方案，也就是功能有限，比如Content Based Routing和Version Based Routing。当然可以将Spring Cloud和Service Mesh结合起来使用，也就是对SC的补充、扩展和加强等；
（5）互联网公司产品的版本升级是非常频繁的，为了维护各个版本的兼容性、权限、流量等，因为Spring Cloud是“代码侵入式的框架”，这时候版本的升级就注定要让非业务代码一起，一旦出现问题，再加上多语言之间的调用，工程师会非常痛苦；
（6）其实大家有没有发现，服务拆分的越细，只是感觉上轻量级解耦了，但是维护成本却越高了，那么怎么办呢？网络的问题当然还是要交给网络本身来解决

19.1 问题解决的思路

- 本质上是要解决服务之间通信的问题，不应该将非业务的代码融合到业务代码中
- 也就是从客户端发出的请求，要能够顺利到达对应的服务，这中间的网络通信的过程要和业务代码尽量无关

通信的问题：服务发现、负载均衡、版本控制、蓝绿部署等

- 在很早之前的单体架构中，其实通信问题也是需要写在业务代码中的，那时候怎么解决的呢？

把网络通信，流量转发等问题放到了计算机网络模型中的TCP/UDP层，也就是非业务功能代码下沉

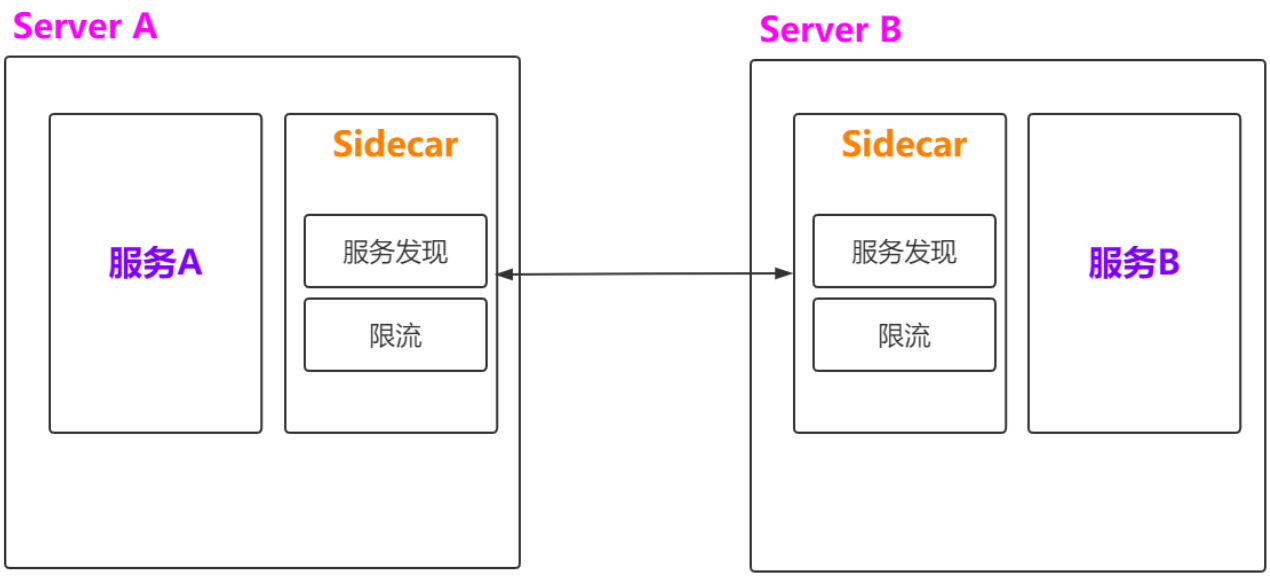
- 那就不妨把这些通信的问题都交给计算机网络模型组织去解决咯？别人肯定不会答应，怎么办？

既然不答应，那我们就自己想办法解决通信问题，搞一些产品岂不快哉？没错，代理嘛

- 不妨这样在帮每个服务配置一个代理，所有通信的问题都交给这个代理去做
- 大家肯定接触过，比如最初Nginx、HaProxy等，其实它们做反向代理把请求转发给其他服务器，也就为Service Mesh的诞生和完成提供了一个解决思路

19.2 对于代理的探索Sidecar

> 很多公司借鉴了Proxy模式，推出了Sidecar的产品
>
> 比如像14年Netflix的Prana、15年唯品会的local proxy
>
> 其实Sidecar模式和Proxy很类似，但是Sidecar功能更全面，也就是Sidecar功能和侵入式框架的功能对应

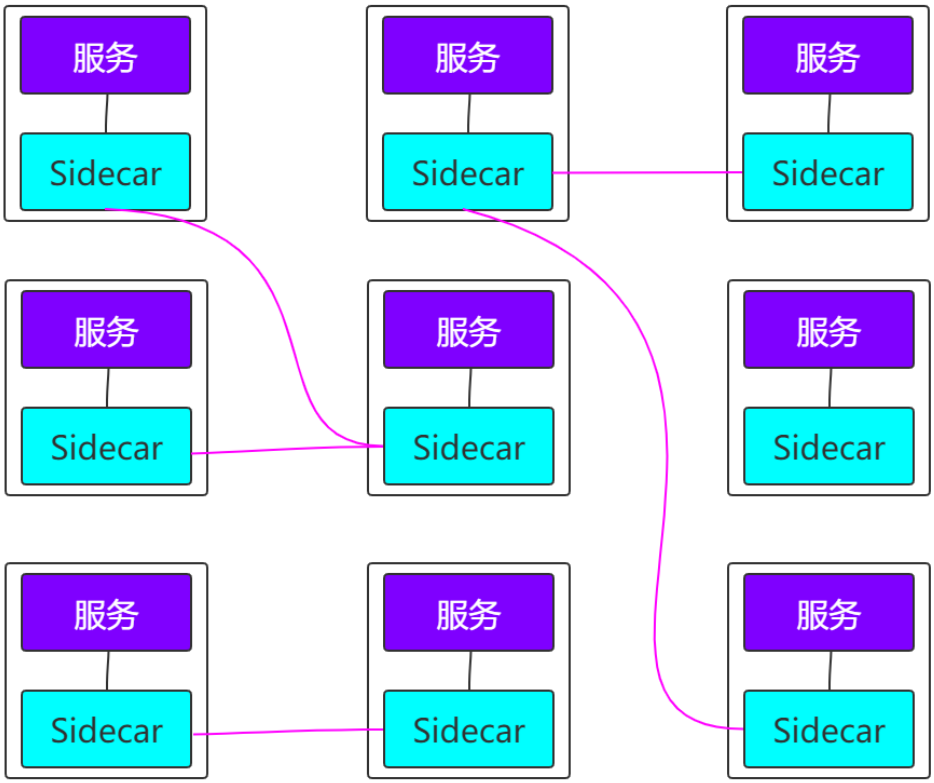


问题：这种Sidecar是为特定的基础设施而设计的，也就是跟公司原有的框架技术绑定在一起，不能成为通用性的产品，所以还需要继续探索。

19.3 Service Mesh之Linkerd

> 2016年1月，离开Twitter的基础设施工程师William Morgan和Oliver Gould，在github上发布了Linkerd 0.0.7版本，从而第一个Service Mesh项目由此诞生。并且Linkerd是基于Twitter的Finagle开源项目，实现了通用性。[**找一下github，scala**]
>
> `William Morgan`:https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one
>
> 后面又出现了Service Mesh的第二个项目Envoy，并且在17年都加入了CNCF项目。

Linkerd设计思想

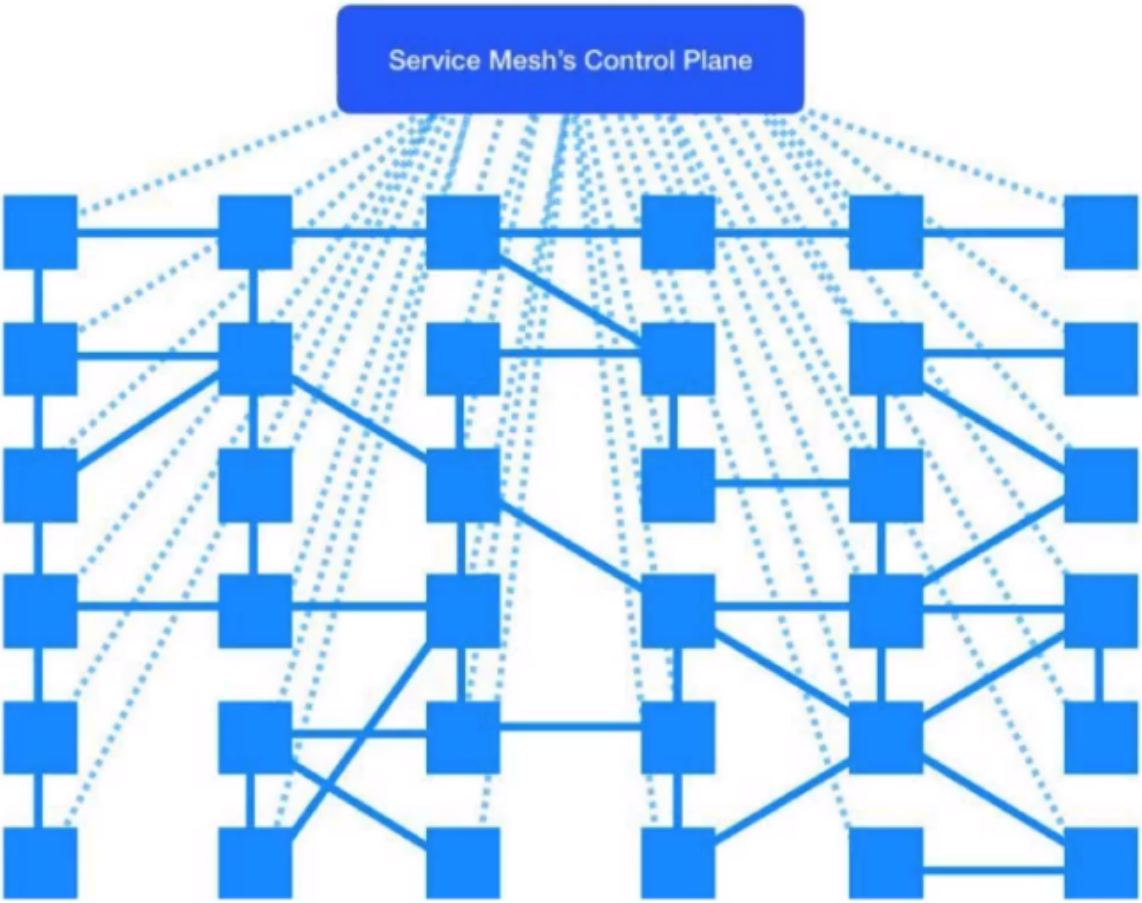


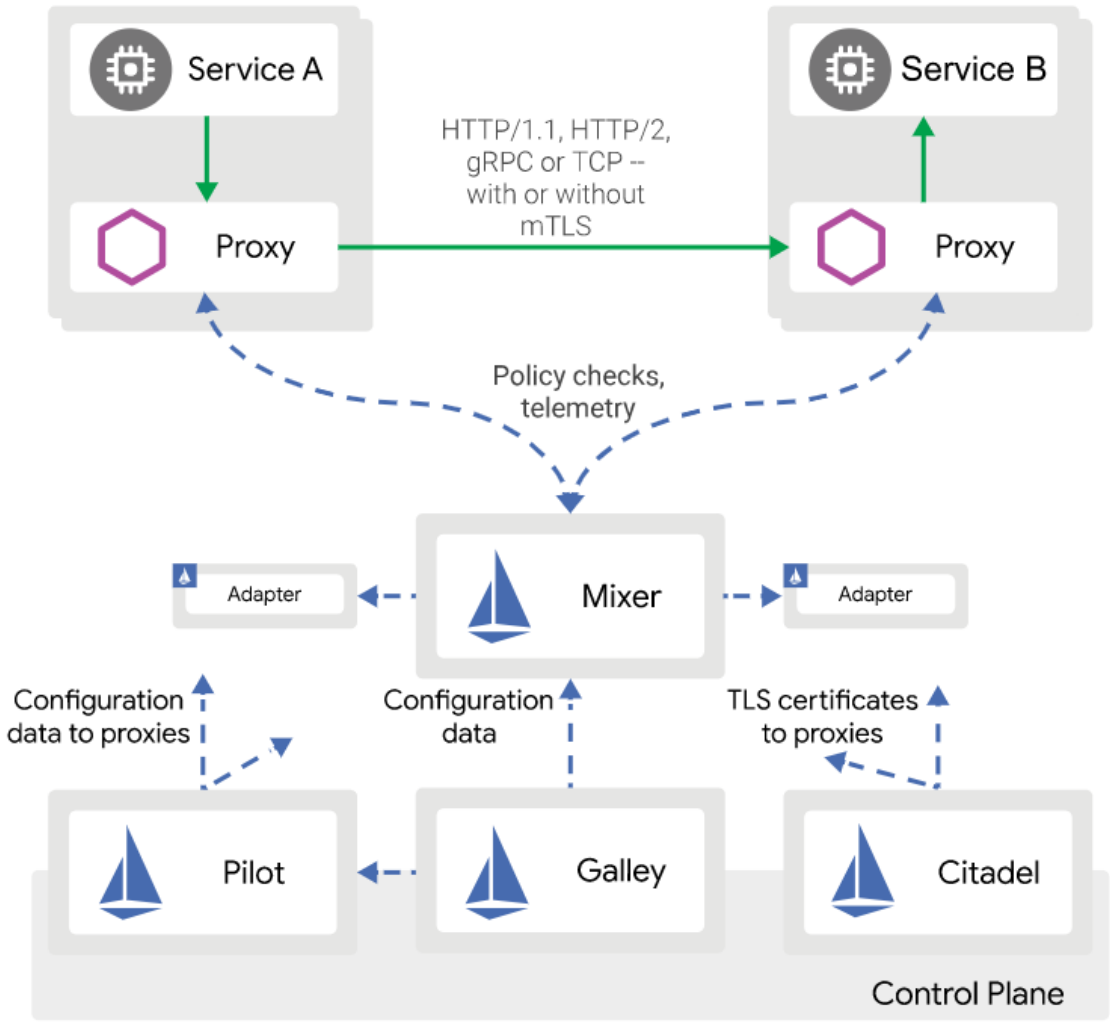
小结：Linkerd解决了通用性问题，在Linkerd思想要求所有的流量都走Sidecar，而原来的Sidecar方式可以走可以直连，这样一来Linkerd就帮业务人员屏蔽了通信细节，也不需要侵入到业务代码中，让业务开发者更加专注于业务本身。

问题：但是Linkerd的设计思想在传统运维方式中太难部署和维护了[**何为难部署？**]，所以就后来没有得到广泛的关注，其实主要的问题是Linkerd只是实现了数据层面的问题，但没有对其进行很好的管理。

19.4 Service Mesh之Istio

> 由Google、IBM和Lyft共同发起的开源项目
>
> 17年5月发布0.1 release版本，17年10月发布0.2 release版本，18年7月发布1.0 release版本
>
> 很明显Istio不仅拥有“数据平面（Data Plane）”，而且还拥有“控制平面（Control Plane），也就是拥有了数据接管与集中控制能力。
>
> `官网Why use Istio?`：https://istio.io/docs/concepts/what-is-istio/#why-use-istio
>
> ```
> Istio makes it easy to create a network of deployed services with load balancing, service-to-service authentication, monitoring, and more, with few or no code changes in service code. You add Istio support to services by deploying a special sidecar proxy throughout your environment that intercepts all network communication between microservices, then configure and manage Istio using its control plane functionality
> ```


Service Mesh's Control Plane



Istio Architecture

19.5 国内发展情况

- 蚂蚁进入的SOFA

```
全称Scalable Open Financial Architecture
前身是SOFA RPC
18年07月正式开源
```

- 腾讯的Tencent Service Mesh
- 华为的CSE Mesher

小结：基本上都借鉴了Sidecar、Envoy和Istio等设计思想

最终目的：TCP/IP协议解决了计算机之间连接的问题，其实我们很少感知到它的存在。

而目前的服务治理也面临相似的问题，也就是要让计算机中的服务更好的连接起来，而且要做到业务代码尽可能无感知。

19.6 Service Mesh发展历程

- 借鉴反向代理，对Proxy模式进行探索，让非业务的代码下沉
- 使用Sidecar弥补Proxy模式功能的不足，解决“侵入式框架”中非业务代码的问题
- Linkerd解决传统Sidecar模式中通用性的问题
- Istio增加了控制平面，解决整个系统中的流量完全控制的问题

19.7 为何Service Mesh开始走红

随着微服务和容器化技术的发展，使得开发和运维的方式变得非常方便。

从而让服务能够方便地迁移到不同的云平台上。

20 云原生

20.1 发展历程

官网：https://www.cncf.io

```
The Cloud Native Computing Foundation (CNCF) hosts critical components of the global technology infrastructure. CNCF brings together the world’s top developers, end users, and vendors and runs the largest open source developer conferences. CNCF is part of the nonprofit Linux Foundation.
```

```
CNCF（Cloud Native Computing Foundation）于 2015 年 7 月成立，隶属于 Linux 基金会，初衷围绕“云原生”服务云计算，致力于维护和集成开源技术，支持编排容器化微服务架构应用。
```

(1)早在 2006 年 8 月 9 日，埃里克·施密特（EricSchmidt）在搜索引擎大会上首次提出了“云计算”（Cloud Computing）的概念。一转眼十年过去了，它的发展势如破竹，不断渗透当代的 IT 市场。

(2)云计算被视为继 80 年代大型计算机到客户端-服务器的大转变之后的又一次革命，为我们带来了工作方式和商业模式的根本性改变，进而成为推动企业创新的引擎。2013年，Pivotal 的 MattStine 提出了云原生（Cloud Native）的概念，凭借其优良的可伸缩性，云原生应用和服务也越来越受到青睐。

(3)为了统一云计算接口和相关标准，2015 年 7 月隶属于 Linux 基金会的云原生计算基金会（CNCF）应运而生。谈到 CNCF，首先它是一个非营利组织，致力于通过技术优势和用户价值创造一套新的通用容器技术，推动本土云计算和服务的发展。CNCF 关注于容器如何管理而不是如何创建，因为如果没有一个成熟的平台去管理容器，那么大型企业无法真正放心接受并使用容器。

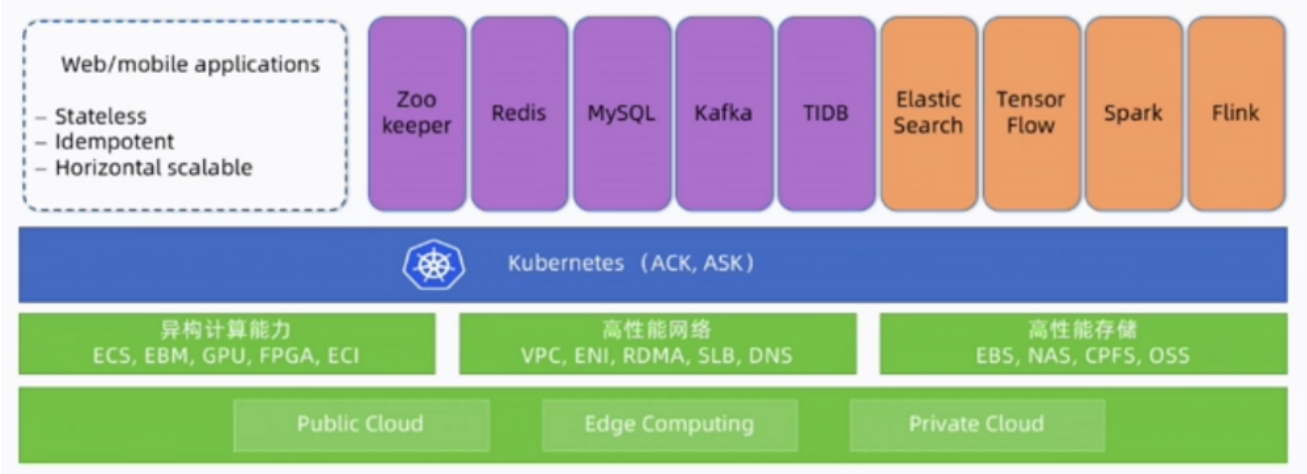
20.2 云原生的项目

正如基金会目标中所描述的任务、角色以及价值观，基金会自创立以来名下已经管理了多个云端原生技术项目，包括：
- Kubernetes ：集群中管理跨多台主机容器化应用的开源系统；
- Prometheus ：专注于时间序列数据，为客户端依赖及第三方数据消费提供广泛集成支持的开源监控解决方案；
- OpenTracing：与厂商无关的分布式追踪开源标准；
- Fluentd：创建统一日志层的开源数据收集器。

另外，2017 开年以来基金会新接纳了五个项目：
- Linkerd：为微服务提供可靠性支持、自动化负载均衡、服务发现和运行时可恢复性的开源“服务网格”项目；
- gRPC：现代化高性能开源远程调用框架；
- CoreDNS：快速灵活的构建 DNS 服务器的方案；
- containerd：将容器运行时及其管理功能从 Docker Daemon 剥离的镜像管理和容器执行技术；
- rkt：帮助开发者打包应用和依赖包，简化搭环境等部署工作，提高容器安全性和易用性的容器引擎。

CNCF 包含的明星项目如此之多，其中广为人知的有 Kubernetes、Prometheus 和目前炙手可热的 gRPC。

20.3 K8s成为基础设施



20.4 现状与未来展望

（1）云原生代表云计算先进生产力 （2）生产环境得云原生应用增长超过200% （3）容器成为Linux之后最受欢迎的项目 （4）预测到2022年，75%的全球企业将使用云原生的容器化应用