

# 一:什么是Feign

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

**在Spring Cloud中，使用Feign非常简单——只需创建接口，并在接口上添加注解即可。**

Feign支持多种注解，例如Feign自带的注解或者JAX-RS注解等。Spring Cloud对Feign进行了增强，使其支持Spring MVC注解，另外还整合了Ribbon和Eureka，从而使得Feign的使用更加方便

eg: 以前我们调用dao的时候，我们是不是一个**接口加 注解**然后在service中就可以进行调用了

```
1 @Mapper
2 public interface OrderDao {
3     List<Order> queryOrdersByUserId(Integer userId);
4 }
```

## 1.2) Ribbon Vs Feign

### ①:Ribbon+ RestTemplate进行微服务调用 模式

```
ResponseEntity<List> responseEntity = restTemplate.getForEntity("http://order-
service/order/queryOrdersByUserId/"+userId,List.class);
```

```
1 Ribbon进行调用
2 @Bean
3 @LoadBalanced
4 public RestTemplate restTemplate() {
5     return new RestTemplate();
6 }
```

#### 缺点:

```
ResponseEntity<List> responseEntity = restTemplate.getForEntity("http://order-
service/order/queryOrdersByUserId/"+userId,List.class);
```

①: 我们不难发现，我们构建上游的URL 是比较简单的，假如我们业务系统十分复杂，类似如下节点  
`https://www.baidu.com/s?`

`wd=asf&rsv_spt=1&rsv_iqid=0xa25bbaba000047fd&issp=1&f=8&rsv_bp=0&rsv_idx=2&ie=utf-8&tn=baiduhome_pg&rsv_enter=1&rsv_sug3=3&rsv_sug1=2&rsv_sug7=100&rsv_sug2=0&inputT:`  
那么我们构建这个请求的URL是不是很复杂，若我们请求的参数有可能变动，那么是否这个URL是不是很复杂

②:

如果系统业务非常复杂，而你是一个新人，当你看到这行代码，恐怕很难一眼看出其用途是什么！此时，你很可能需要寻求老同事的帮助（往往是这行代码的作者，哈哈，可万一离职了呢？），或者查阅该目标地址对应的文档（文档常常还和代码不匹配），才能清晰了解这行代码背后的含义！否则，你只能陷入蛋疼的境地！

## 1.3)在我们工程中怎么添加Feign

### 1.3.1)我们采取开发中常用的套路 定义一个tulingvip03-ms-alibaba-feign-api工程

#### ①:第一步，引入依赖

```
1 <dependency>
2 <groupId>org.springframework.cloud</groupId>
3 <artifactId>spring-cloud-starter-openfeign</artifactId>
4 </dependency>
```

## ②:第二步:修改打包方式(因为该工程式一个普通的jar 不需要打可执行的jar)

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.apache.maven.plugins</groupId>
5       <artifactId>maven-jar-plugin</artifactId>
6     </plugin>
7   </plugins>
8 </build>
```

## 第三步:编写声明式接口

```
1 @FeignClient(name = "product-center")
2 public interface ProductCenterFeignApi {
3
4   /**
5    * 声明式接口, 远程调用http://product-center/selectProductInfoById/{productNo}
6    * @param productNo
7    * @return
8    */
9   @RequestMapping("/selectProductInfoById/{productNo}")
10   ProductInfo selectProductInfoById(@PathVariable("productNo") String productNo);
11 }
```

## 1.3.2)调用者工程tulingvip03-ms-alibaba-feign-order

### 第一步:引入依赖包 tulingvip03-ms-alibaba-feign-api

```
1 <dependency>
2   <groupId>com.tuling</groupId>
3   <artifactId>tulingvip03-ms-alibaba-feign-api</artifactId>
4   <version>0.0.1-SNAPSHOT</version>
5 </dependency>
```

### 第二步: 开启注解加入 @EnableFeignClients

```
1 @SpringBootApplication
2 @EnableDiscoveryClient
3 @EnableFeignClients
4 public class Tulingvip03MsAlibabaFeignOrderApplication {
5
6   public static void main(String[] args) {
7     SpringApplication.run(Tulingvip03MsAlibabaFeignOrderApplication.class, args);
8   }
9 }
```

### 第三步:调用方式(像调用本地方式一样调用远程服务)

```
1 @Autowired
2 private ProductCenterFeignApi productCenterFeignApi;
3
4 ProductInfo productInfo = productCenterFeignApi.selectProductInfoById(orderNo);
```

1.3.3)我们服务提供者tulingvip03-ms-alibaba-feign-product 的controller 需要实现我们的productCenterFeignApi接口, 防止修改(比如我们的productCenterFeignApi修改了, 若没有实现该接口, 服务提供者感知不到)

### 第一步:引入依赖 tulingvip03-ms-alibaba-feign-api

### 第二步:我们的ProductInfoController实现productCenterFeignApi接口

```
1 @RestController
2 public class ProductInfoController implements ProductCenterFeignApi
```

## 二:如何自定义Feign

2.1)默认情况下, Feign的调用式不打印日志, 我们需要通过自定义来打印我们的Feign的日志 (basic适用于生产环境)

级别	打印内容
NONE（默认值）	不记录任何日志
BASIC	仅记录请求方法、URL、响应状态代码以及执行时间
HEADERS	记录BASIC级别的基础上，记录请求和响应的header
FULL	记录请求和响应的header、body和元数据

### ①: 我们在tulingvip03-ms-alibaba-feign-api工程中添加Feign的自定义配置

```

1 /**
2  * 这个类上千万不要添加@Configuration,不然会被作为全局配置文件共享
3  * Created by smlz on 2019/11/22.
4  */
5 public class ProductCenterFeignConfig {
6
7     @Bean
8     public Logger.Level level() {
9         //return Logger.Level.FULL;
10        return Logger.Level.BASIC;
11    }
12 }
13
14 @FeignClient(name = "product-center", configuration = ProductCenterFeignConfig.class)
15 public interface ProductCenterFeignApi {
16
17     /**
18      * 声明式接口,远程调用http://product-center/selectProductInfoById/{productNo}
19      * @param productNo
20      * @return
21      */
22     @RequestMapping("/selectProductInfoById/{productNo}")
23     ProductInfo selectProductInfoById(@PathVariable("productNo") String productNo);
24 }

```

### ②: 针对调用端工程tulingvip03-ms-alibaba-customcfg-feign-order针对日志com.tuling.feignapi 包下的日志级别必须调整为DEBUG级别的 不然是不会打印日志的

```

1 logging:
2   level:
3     com:
4       tuling:
5         feignapi: debug

```

```

c.t.f.p.ProductCenterFeignapi : [ProductCenterFeignapi$selectProductInfoById] <--- HTTP/1.1 200 (358ms)
c.t.f.p.ProductCenterFeignapi : [ProductCenterFeignapi$selectProductInfoById] content-type: application/json;charset=UTF-8
c.t.f.p.ProductCenterFeignapi : [ProductCenterFeignapi$selectProductInfoById] date: Fri, 22 Nov 2019 07:29:23 GMT
c.t.f.p.ProductCenterFeignapi : [ProductCenterFeignapi$selectProductInfoById] transfer-encoding: chunked
c.t.f.p.ProductCenterFeignapi : [ProductCenterFeignapi$selectProductInfoById]
c.t.f.p.ProductCenterFeignapi : [ProductCenterFeignapi$selectProductInfoById] {"productNo":"1","productName":"iphone11","productStore":"100","productPrice":9999.0}
c.t.f.p.ProductCenterFeignapi : [ProductCenterFeignapi$selectProductInfoById] <--- END HTTP (85-byte body)

```

## 2.2)基于yaml文件细粒度配置

### ProductCenterFeignApi 不用指定configuration的选项

```

1 @FeignClient(name = "product-center")
2 public interface ProductCenterFeignApi {
3
4     /**
5      * 声明式接口,远程调用http://product-center/selectProductInfoById/{productNo}
6      * @param productNo
7      * @return
8      */
9     @RequestMapping("/selectProductInfoById/{productNo}")
10    ProductInfo selectProductInfoById(@PathVariable("productNo") String productNo);
11 }

```

在调用方: tulingvip03-ms-alibaba-customcfg-feign-order

通过feign:client:config:微服务名称:loggerLevel: 日志级别来指定

```
1 logging:
2 level:
3 com:
4 tuling:
5 feignapi: debug
6 feign:
7 client:
8 config:
9 product-center:
10 loggerLevel: full
```

```
ProductCenterFeignApi : [ProductCenterFeignApi#selectProductInfoById] <--- HTTP/1.1 200 (1466ms)
ProductCenterFeignApi : [ProductCenterFeignApi#selectProductInfoById] content-type: application/json;charset=UTF-8
ProductCenterFeignApi : [ProductCenterFeignApi#selectProductInfoById] date: Fri, 22 Nov 2019 07:37:54 GMT
ProductCenterFeignApi : [ProductCenterFeignApi#selectProductInfoById] transfer-encoding: chunked
ProductCenterFeignApi : [ProductCenterFeignApi#selectProductInfoById]
ProductCenterFeignApi : [ProductCenterFeignApi#selectProductInfoById] {"productNo":"1","productName":"iphone11","produ
ProductCenterFeignApi : [ProductCenterFeignApi#selectProductInfoById] <--- END HTTP (85-byte body)
```

## 2.3)使用Feign原生的注解配置(需要修改契约)

根据自动装配我们FeignClients的配置中的默认契约是springmvc (也就是说支持SpringMvc注解)

```
1 @Bean
2 @ConditionalOnMissingBean
3 public Contract feignContract(ConversionService feignConversionService) {
4     return new SpringMvcContract(this.parameterProcessors, feignConversionService);
5 }
6
7 所以我们在FeignClient上可以使用Mvc的注解
8 @RequestMapping("/selectProductInfoById/{productNo}")
9 ProductInfo selectProductInfoById(@PathVariable("productNo") String productNo);
```

现在我们需要想使用Feign的原生注解来标识方法需要修改契约

```
1
2 public class ProductCenterFeignConfig{
3     /**
4      * 根据SpringBoot自动装配FeignClientsConfiguration 的FeignClient的契约是SpringMvc
5      *
6      * 通过修改契约为默认的Feign的契约, 那么就可以使用默认的注解
7      * @return
8      */
9     @Bean
10    public Contract feiContract() {
11        return new Contract.Default();
12    }
13 }
14
15
```

FeignClient类ProductCenterFeignApi使用Feign原生的注解

```
1 @FeignClient(name = "product-center",configuration = ProductCenterFeignConfig.class)
2 public interface ProductCenterFeignApi {
3     /**
4      * 修改契约为Feign的 那么就可以使用默认的注解
5      * @param productNo
6      * @return
7      */
8     @RequestMapping("GET /selectProductInfoById/{productNo}")
9     ProductInfo selectProductInfoById(@Param("productNo") String productNo);
10 }
```

## 也可以通过配置文件的形式来指定我们的契约

```
1 feign:
2   client:
3   config:
4   product-center:
5     loggerLevel: full
6   contract: feign.Contract.Default #指定默认契约
```

## 2.4) 拦截器的应用配置

```

1 public class TulingRequestInterceptor implements RequestInterceptor {
2     @Override
3     public void apply(RequestTemplate template) {
4         template.header("token", UUID.randomUUID().toString());
5     }
6 }
7
8 public class ProductCenterFeignConfig {
9     @Bean
10    public RequestInterceptor requestInterceptor() {
11        return new TulingRequestInterceptor();
12    }
13
14 }
15
16
17 @FeignClient(name = "product-center", configuration = ProductCenterFeignConfig.class)
18 // @FeignClient(name = "product-center")
19 public interface ProductCenterFeignApi {
20     @RequestLine("GET /getToken4Header")
21     String getToken4Header(@RequestHeader("token") String token);
22 }

```

服务提供者:

```
1 @RestController
2 @Slf4j
3 public class ProductInfoController implements ProductCenterFeignApi {
4
5     @RequestMapping("/getToken4Header")
6     public String getToken4Header(@RequestHeader("token") String token) {
7         log.info("token:{}", token);
8         return token;
9     }
10 }
```

```

2019-11-22 17:47:10.738 INFO 73896 --- [main] jn.mgc3mllibm.org.ProductApplication : Started [jn.mgc3mllibm.org.ProductApplication in 6.997 sec
2019-11-22 17:26:00.838 INFO 73896 --- [nio-8082-exec-1] o.a.c.c.([localhost/]) : Initializing Spring FrameworkServlet 'dispatcherServlet'
2019-11-22 17:26:00.839 INFO 73896 --- [nio-8082-exec-1] o.a.c.c.([localhost/]) : FrameworkServlet 'dispatcherServlet': initialization started
2019-11-22 17:26:00.852 INFO 73896 --- [nio-8082-exec-1] o.a.c.c.([localhost/]) : FrameworkServlet 'dispatcherServlet': initialization completed
2019-11-22 17:26:00.890 INFO 73896 --- [nio-8082-exec-1] c.t.controller.ProductInfoController : token:847e20c2-4870-4a97-915e-8a3e4995e119

```

## 2.4) Feign调用优化方案

## 第一步:开启连接池配置

```
1 feign:
2   client:
3   config:
4   product-center:
5     loggerLevel: full
6   contract: feign.Contract.Default
7   httpclient:
8   #让feign底层使用HttpClient去调用
```

```
9  enabled: true
10 max-connections: 200 #最大连接数
11 max-connections-per-route: 50 #为每个url请求设置最大连接数
```

## 第二步：调整Feign的日志级别（强烈推荐使用Basic级别的）

### 2.5) Feign的生产实践（已Feign的超时说了算）

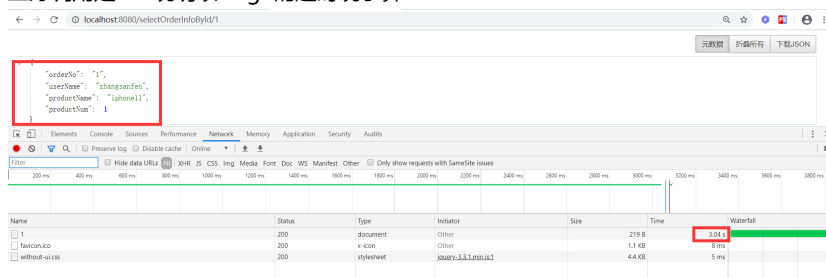
问题①:Feign的底层用的是Ribbon，那么我们怎么配置超时时间

服务提供方模拟耗时 睡眠3S

```
@RequestMapping("/selectProductInfoById/{productNo}")
public ProductInfo selectProductInfoById(@PathVariable("productNo") String productNo) {
    log.info("接口被调用了!!");
    Thread.sleep( millis: 3000);
    ProductInfo productInfo = productInfoMapper.selectProductInfoById(productNo);
    return productInfo;
}
```

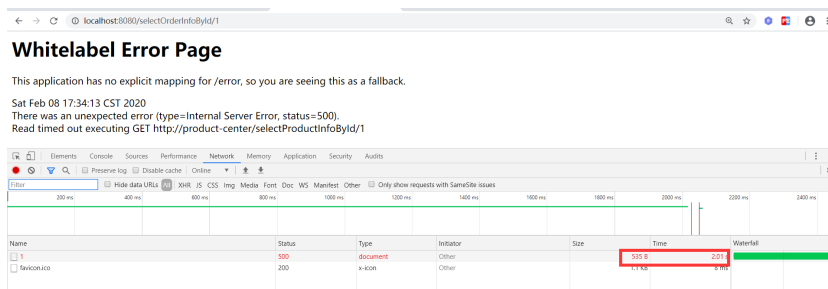
```
1 配置这个 ribbon肯定会超时
2 ribbon:
3   connectTimeout: 2000
4   readTimeout: 2000
5
6 (Feign不会超时)
7 feign:
8   client:
9   config:
10  default:
11   connectTimeout: 5000
12   readTimeout: 5000
```

正好调用是3S 说明以Feign的超时说了算

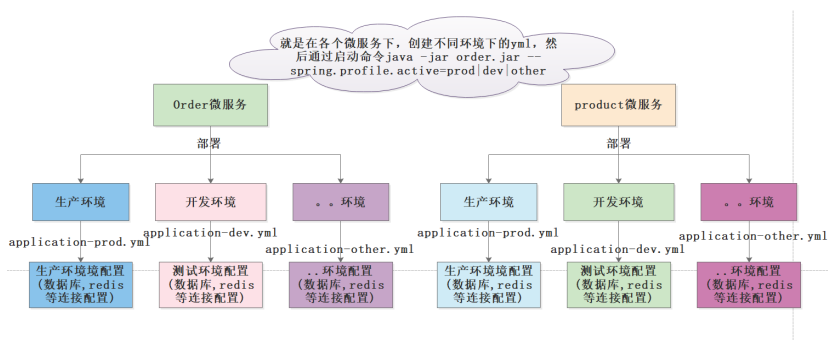


这种配置 Feign超时了

```
1 ribbon:
2   connectTimeout: 5000
3   readTimeout: 5000
4 http:
5   client:
6   enabled: true
7 feign:
8   client:
9   config:
10  default:
11   connectTimeout: 2000
12   readTimeout: 2000
```



### 3.什么是配置管理??



#### 上图缺点:

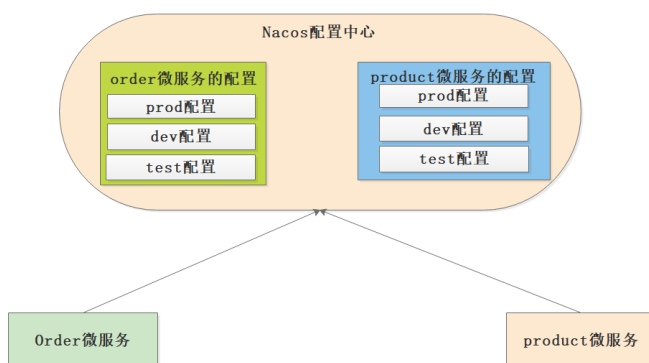
所有的环境的配置都是明文的 被太多开发人员都知道了。

业务场景：张三开发了一个新功能,业务需要，保留原来老逻辑的代码，所有他抽取了一个开关变量isNewBusi来控制，突然新功能上了生产后，发现有bug,怎么做到修改isNewBusi的值不需要重启。

### 3.1)根据上图我们知道配置管理的作用可以主要总结如下

- 1) 不同环境不管配置
- 2) 配置属性动态刷新

### 引入配置中心



### 根据这幅图，我们微服务需要解决的问题

- 1)我微服务怎么知道配置中心的地址
- 2) 我微服务到底需要连接哪个环境
- 3)怎么找到nacos config上的对应的配置文件

## 微服务接入配置中心的步骤

### ①:添加依赖包spring-cloud-alibaba-nacos-config

```
1 <dependency>
2   <groupId>com.alibaba.cloud</groupId>
3   <artifactId>spring-cloud-alibaba-nacos-config</artifactId>
4 </dependency>
```

## ②:编写配置文件,需要写一个bootstrap.yml配置文件

### 配置解释:

server-addr: localhost:8848 表示我微服务怎么去找我的配置中心

spring.application.name=order-center 表示当前微服务需要向配置中心索要order-center的配置

spring.profiles.active=prod 表示我需要向配置中心索要order-center的生产环境的配置

索要文件的格式为

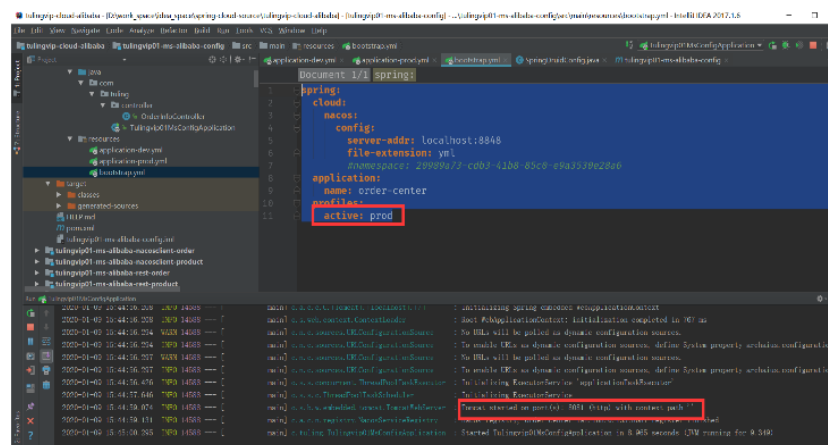
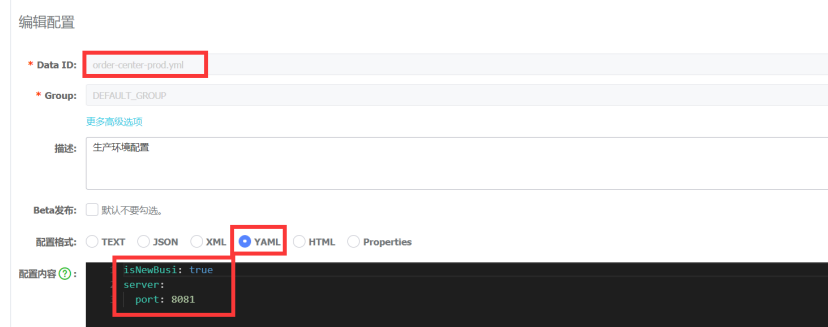
```

${application.name}- ${spring.profiles.active}.${file-extension}

```

真正在nacos配置中心上 就是 **order-center-prod.yml**

```
1 spring:
2   cloud:
3     nacos:
4       config:
5         server-addr: localhost:8848
6         file-extension: yml
7       application:
8         name: order-center
9       profiles:
10        active: prod
```



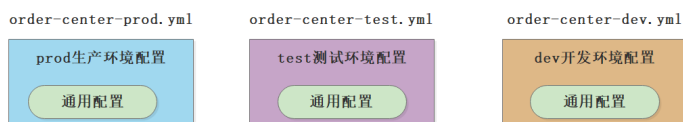


localhost:8081/selectOrderInfoById/1

查询订单执行新逻辑->execute new busi : 1

现在我们需要不停机改变我们的生产环境isNewBusi的值来控制我们的业务逻辑。我们需要在对应的Controller上添加一@RefreshScope进行动态刷新

### 3.2) 怎么解决 生产环境，测试环境，开发环境相同的配置。（配置通用）



比如我们的servlet-context 为order-center

查看工程启动日志

```
Loading nacos data, dataId: 'order-center-prod.yml', group: 'DEFAULT_GROUP'
Located property source: CompositePropertySource [name='NACOS', propertySources=[NacosPropertySource (name='order-center-prod.yml'), NacosPropertySource (name='order-center.yml')]]
The following profiles are active: prod
BootstrapFactory id=2066222-011c-379f-2013-992ce7b03047
Bean 'org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebinderAutoConfiguration' of type [org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebinderAutoConfiguration]
Tomcat initialized with port(s): 8081 (http)
Starting service [Tomcat]
Starting Servlet engine: [Apache Tomcat/9.0.16]
The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: [C:\Program Files\Java\jdk1.8.0_131\bin.C\WINDOWS]
Initializing Spring embedded WebApplicationContext
Root WebApplicationContext: initialization completed in 1250ms
```

所以我们需要创建一个通用配置文件:order-center.yml配置

那么order-center.yml就是一个通用配置了，不管是启动prod,还是dev 都会有该段配置order-server的 context-path 配置

## 新建配置

\* Data ID: order-center.yml

\* Group: DEFAULT\_GROUP

更多高级选项

描述: order各个环境的通用配置

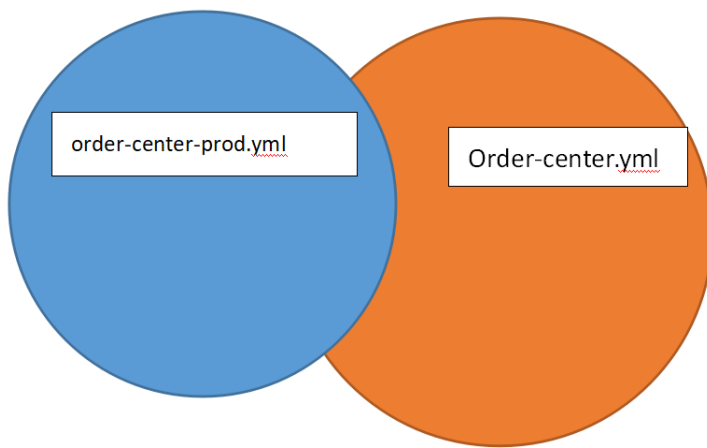
配置格式: ☐ TEXT ☐ JSON ☐ XML ☒ YAML ☐ HTML ☐ Properties

\* 配置内容:

?

```
1 server:
2   servlet:
3     context-path: /order-server
```

配置的优先级 精准配置 会覆盖 与通用配置 相同的配置，然后再和通用配置互补。



### 3.3)不同微服务的通用配置。



#### 3.3.1)通过 shared-dataids 方式

比如 每一个微服务都要写服务注册地址等

```
1 spring:
2   cloud:
3     nacos:
4       discovery:
5         server-addr: localhost:8848
```

我们就可以把该配置提取为一个公共配置yml,提供给所有的工程使用

```
1 spring:
2   cloud:
3     nacos:
4       config:
5         server-addr: localhost:8848
6         file-extension: yaml
7         #各个微服务共享的配置,注意越拍到后面的公共配置yaml优先级越高
8         shared-dataids: common.yaml,common2.yaml
9         #支持动态刷新的配置文件
10        refreshable-dataids: common.yaml,common2.yaml
11      application:
12        name: order-center
13      profiles:
14        active: dev
```

配置优先级

```
2020-01-09 21:21:19.115 INFO 15316 --- [      main] b.c.PropertySourceBootstrapConfiguration : Located property source:
CompositePropertySource {name='NACOS',
propertySources=[
NacosPropertySource {name='order-center-dev.yml'},
NacosPropertySource {name='order-center.yml'},
NacosPropertySource {name='common2.yaml'},
NacosPropertySource {name='common.yaml'}
```

```
]
}
```

### 6.3.1)通过 ext-config方式

同样配置到越后面的配置 优先级越高

```
1 spring:
2   cloud:
3     nacos:
4       config:
5         server-addr: localhost:8848
6         file-extension: yaml
7         ext-config:
8           - data-id: common3.yaml
9             group: DEFAULT_GROUP
10            refresh: true
11           - data-id: common4.yaml
12             group: DEFAULT_GROUP
13            refresh: true
```

### 6.3.3)各个配置的优先级

精准配置>不同环境的通用配置>不同工程的(ext-config)>不同工程(shared- dataids)

```
1 spring:
2   cloud:
3     nacos:
4       config:
5         server-addr: localhost:8848
6         file-extension: yaml
7         shared-dataids: common.yaml,common2.yaml
8         refreshable-dataids: common.yaml,common2.yaml
9         ext-config:
10           - data-id: common3.yaml
11             group: DEFAULT_GROUP
12             refresh: true
13           - data-id: common4.yaml
14             group: DEFAULT_GROUP
15             refresh: true
16
17   application:
18     name: order-center
19     profiles:
20       active: dev
```

上述配置 加载的优先级

1)order-center-dev.yml 精准配置

2)order-center.yml 同工程不同环境的通用配置

3)ext-config: 不同工程 通用配置

3.1): common4.yml

3.2): common3.yml

4) shared-dataids 不同工程通用配置

4.1)common2.yml

4.2)common1.yml