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

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

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);

```
Ribbon进行调用
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
return new RestTemplate();
}
```

缺点:

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

①：我们不难发现，我们构建上诉的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=0xa25bbeba000047fd&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

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

```
<build>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-jar-plugin</artifactId>
 </plugin>
 </plugins>
</build>
```

第三步:编写声明式接口

```
@FeignClient(name = "product-center")
public interface ProductCenterFeignApi {

 /**
 * 声明式接口,远程调用http://product-center/selectProductInfoById/{productNo}
 * @param productNo
 * @return
 */
 @RequestMapping("/selectProductInfoById/{productNo}")
  ProductInfo selectProductInfoById(@PathVariable("productNo") String productNo);
}
```

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

```
<dependency>
 <groupId>com.tuling</groupId>
 <artifactId>tulingvip03-ms-alibaba-feign-api</artifactId>
 <version>0.0.1-SNAPSHOT</version>
</dependency>
```

第二步: 开启注解加入 @EnableFeignClients

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class Tulingvip03MsAlibabaFeignOrderApplication {

 public static void main(String[] args) {
 SpringApplication.run(Tulingvip03MsAlibabaFeignOrderApplication.class, args);
 }
}
```

第三步:调用方式(像调用本地方式一样调用远程服务)

```
@Autowired
private ProductCenterFeignApi productCenterFeignApi;

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

```
@RestController
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)

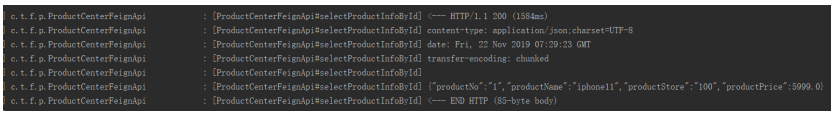
级别	打印内容
NONE（默认值）	不记录任何日志
BASIC	仅记录请求方法、URL、响应状态代码以及执行时间
HEADERS	记录BASIC级别的基础上，记录请求和响应的header
FULL	记录请求和响应的header、body和元数据

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

```
/**
 * 这个类上千万不要添加@Configuration,不然会被作为为全局配置文件共享
 * Created by smlz on 2019/11/22.
 */
public class ProductCenterFeignConfig {

 @Bean
 public Logger.Level level() {
 //return Logger.Level.FULL;
  return Logger.Level.BASIC;
  }
}

@FeignClient(name = "product-center",configuration = ProductCenterFeignConfig.class)
public interface ProductCenterFeignApi {

 /**
  * 声明式接口,远程调用http://product-center/selectProductInfoById/{productNo}
  * @param productNo
  * @return
  */
 @RequestMapping("/selectProductInfoById/{productNo}")
  ProductInfo selectProductInfoById(@PathVariable("productNo") String productNo);
}
```

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

```
logging:
 level:
 com:
 tuling:
 feignapi: debug
```



2.2)基于yml文件细粒度配置

ProductCenterFeignApi 不用指定configuration的选项

```
@FeignClient(name = "product-center")
public interface ProductCenterFeignApi {

 /**
  * 声明式接口,远程调用http://product-center/selectProductInfoById/{productNo}
  * @param productNo
  * @return
  */
 @RequestMapping("/selectProductInfoById/{productNo}")
  ProductInfo selectProductInfoById(@PathVariable("productNo") String productNo);
}
```

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

```yaml
logging:
 level:
 com:
 tuling:
 feignapi: debug
feign:
 client:
 config:
 product-center:
  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的配置中的默认锲约是springmvc（也就是说支持SpingMvc注解）

```
@Bean
@ConditionalOnMissingBean
public Contract feignContract(ConversionService feignConversionService) {
 return new SpringMvcContract(this.parameterProcessors, feignConversionService);
}

所以我们在FeignClient上可以使用Mvc的注解
@RequestMapping("/selectProductInfoById/{productNo}")
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锲约

```

public class ProductCenterFeignConfig{
 /**
 * 根据SpringBoot自动装配FeignClientsConfiguration 的FeignClient的契约是SpringMvc
 *
 通过修改契约为默认的Feign的锲约，那么就可以使用默认的注解
 * @return
 */
 @Bean
  public Contract feiContract() {
  return new Contract.Default();
  }
}


```

FeignClient类ProductCenterFeignApi使用Feign原生的注解

```
@FeignClient(name = "product-center",configuration = ProductCenterFeignConfig.class)
 public interface ProductCenterFeignApi {
 /**
 * 修改锲约为Feign的 那么就可以使用默认的注解
 * @param productNo
 * @return
 */
 @RequestLine("GET /selectProductInfoById/{productNo}")
 ProductInfo selectProductInfoById(@Param("productNo") String productNo);
}
```

```
11
```

也可以通过配置文件的形式来指定我们的契约

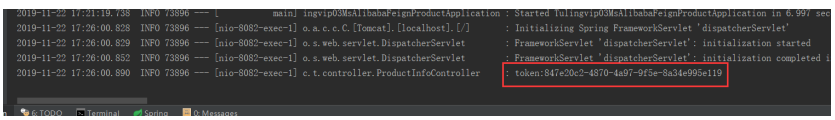
```
feign:
 client:
 config:
 product-center:
 loggerLevel: full
 contract: feign.Contract.Default #指定默认契约
```

2.4）拦截器的应用配置

```
public class TulingRequestInterceptor implements RequestInterceptor {
 @Override
 public void apply(RequestTemplate template) {
 template.header("token", UUID.randomUUID().toString());
 }
}

public class ProductCenterFeignConfig {
 @Bean
  public RequestInterceptor requestInterceptor() {
  return new TulingRequestInterceptor();
  }

}


@FeignClient(name = "product-center",configuration = ProductCenterFeignConfig.class)
//@FeignClient(name = "product-center")
public interface ProductCenterFeignApi {
 @RequestLine("GET /getToken4Header")
 String getToken4Header(@RequestHeader("token") String token);
}
```

服务提供者:

```
@RestController
@Slf4j
public class ProductInfoController implements ProductCenterFeignApi {

 @RequestMapping("/getToken4Header")
 public String getToken4Header(@RequestHeader("token") String token) {
 log.info("token:{}",token);
 return token;
 }
}
```



2.4）Feign调用优化方案

第一步:开启连接池配置

```
feign:
 client:
 config:
 product-center:
 loggerLevel: full
 contract: feign.Contract.Default
 httpclient:
 #让feign底层使用HttpClient去调用
```

```
  enabled: true
  max-connections: 200 #最大连接数
  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）

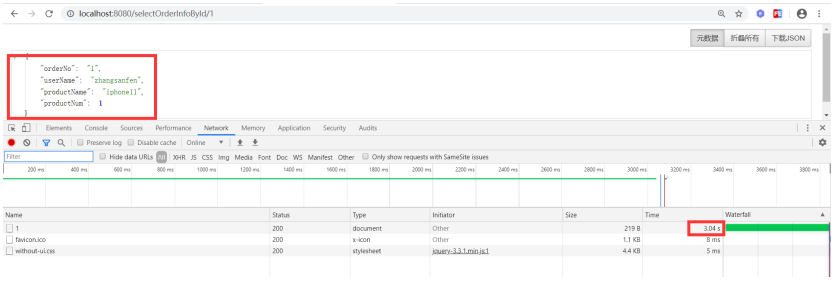
问题①:Feign的底层用的是Ribbon，那么我们怎么配置超时时间

服务提供方模拟耗时 睡眠3S

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

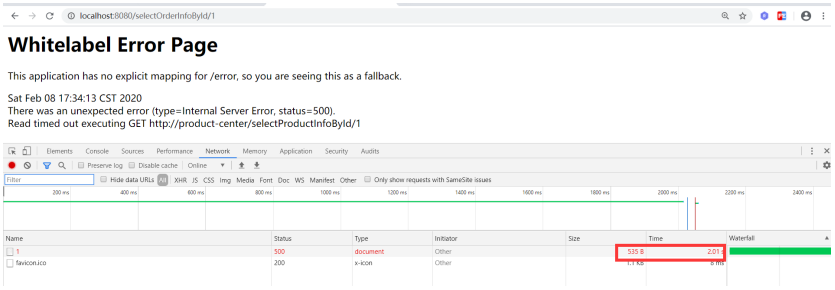
```
配置这个 ribbon肯定会超时
ribbon:
  connectTimeout: 2000
  readTimeout: 2000

（Feign不会超时）
feign:
  client:
  config:
   default:
   connectTimeout: 5000
   readTimeout: 5000
```

正好调用是3S 说明以Feign的超时说了算

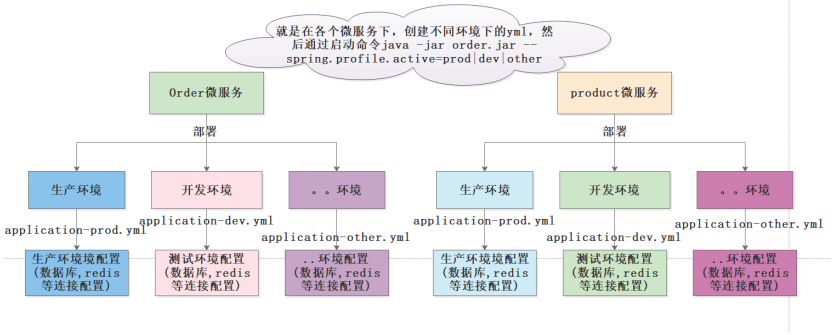


这种配置 Feign超时了

```
ribbon:
  connectTimeout: 5000
  readTimeout: 5000
  http:
  client:
  enabled: true
feign:
  client:
  config:
   default:
   connectTimeout: 2000
   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。

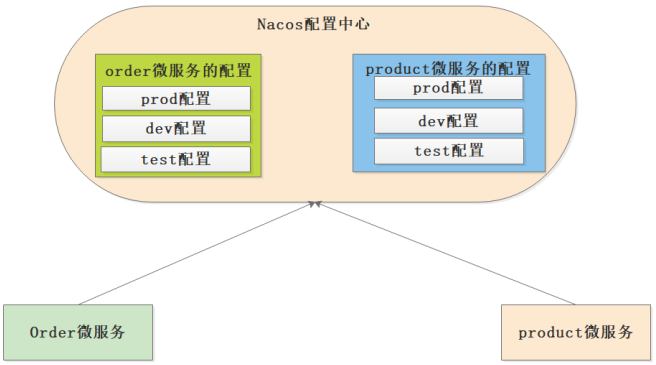
业务场景：张三开发了一个新功能,业务需要，保留原来老逻辑的代码，所有他抽取了一个开关变量isNewBusi来控制，突然新功能上了生产后，发现有bug,怎么做到修改isNewBusi的值不需要重启。

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

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

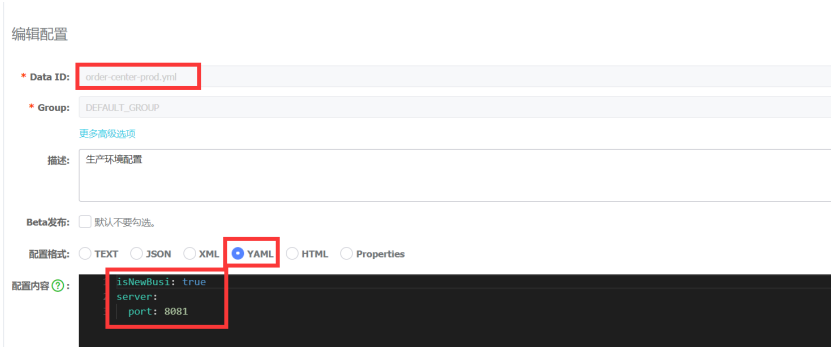
spring.profiles.active=prod 表示我需要向配置中心索要order-center的生产环境的配置

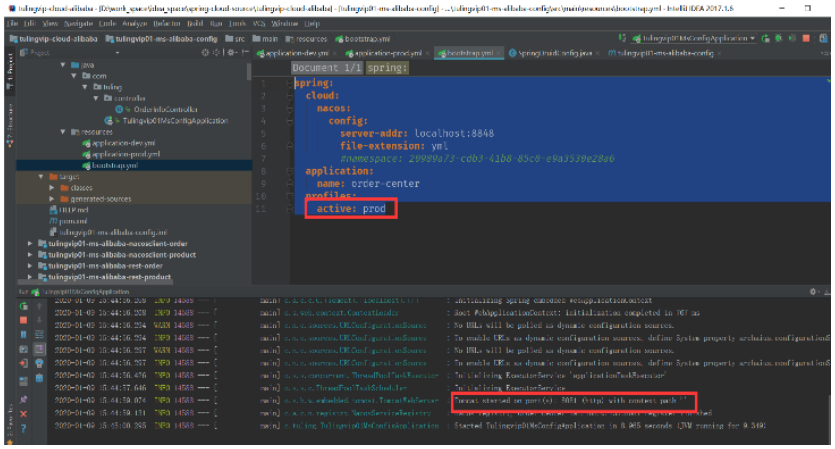
索要文件的格式为

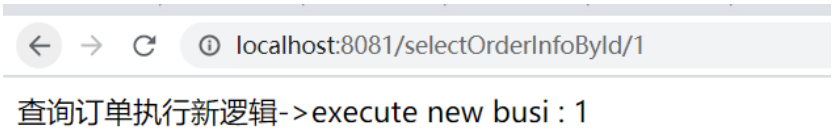
${application.name}- ${spring.profiles.active}.${file-extension}

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

```
spring:
 cloud:
 nacos:
 config:
 server-addr: localhost:8848
 file-extension: yml
 application:
 name: order-center
 profiles:
  active: prod
```

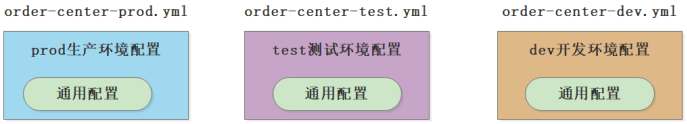






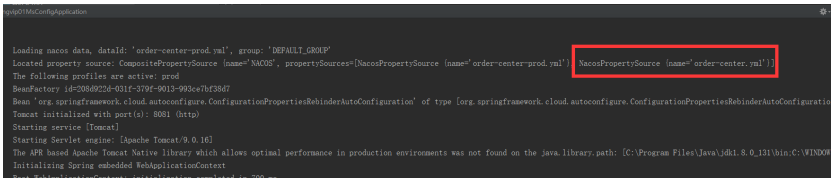
现在我们需要不停机改变我们的生产环境isNewBusi的值来控制我们的业务逻辑。我们需要在对应的Controller上添加一**@RefreshScope**进行动态刷新

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

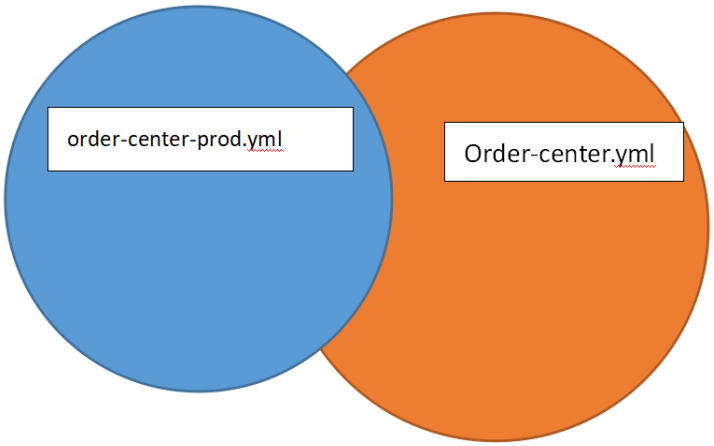


所以我们需要创建一个通用配置文件:order-center.yml配置

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

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

```yaml
spring:
  cloud:
  nacos:
  discovery:
  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

```yaml
spring:
  cloud:
  nacos:
  config:
  server-addr: localhost:8848
  file-extension: yml
  #各个微服务共享的配置,注意越拍到后面的公共配置yml优先级越高
  shared-dataids: common.yml,common2.yml
  #支持动态刷新的配置文件
   refreshable-dataids: common.yml,common2.yml
   application:
   name: order-center
   profiles:
   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.yml'},
NacosPropertySource {name='common.yml'}

]
}

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

```yaml
spring:
 cloud:
 nacos:
 config:
 server-addr: localhost:8848
 file-extension: yml
 ext-config:
 - data-id: common3.yml
 group: DEFAULT_GROUP
  refresh: true
  - data-id: common4.yml
  group: DEFAULT_GROUP
  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)

```yaml
spring:
 cloud:
 nacos:
 config:
 server-addr: localhost:8848
 file-extension: yml
 shared-dataids: common.yml,common2.yml
 refreshable-dataids: common.yml,common2.yml
 ext-config:
  - data-id: common3.yml
  group: DEFAULT_GROUP
  refresh: true
  - data-id: common4.yml
  group: DEFAULT_GROUP
  refresh: true

  application:
  name: order-center
  profiles:
  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