主讲老师： Fox老师

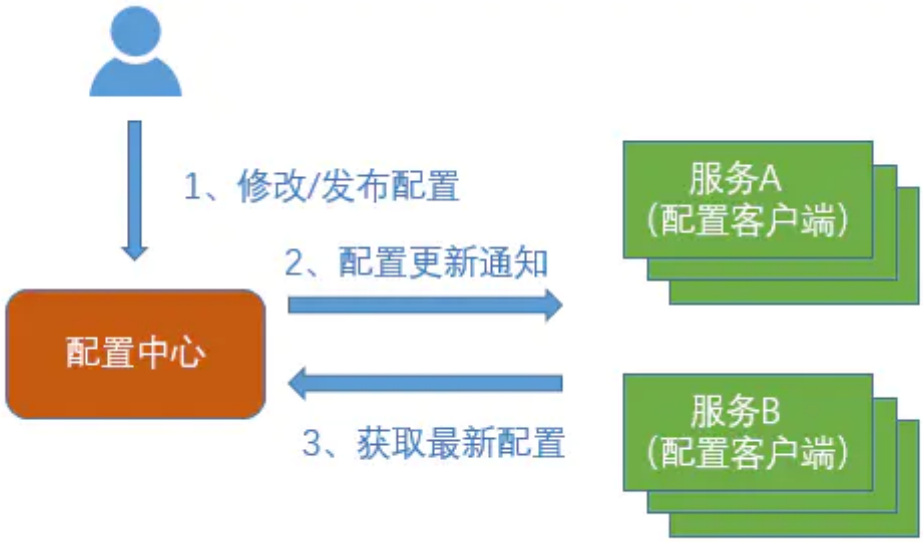
> 有道笔记：https://note.youdao.com/s/b4JgbD0R

1. Nacos配置中心

1.1 微服务为什么需要配置中心

在微服务架构中，当系统从一个单体应用，被拆分成分布式系统上一个个服务节点后，配置文件也必须跟着迁移（分割），这样配置就分散了，不仅如此，分散中还包含着冗余。

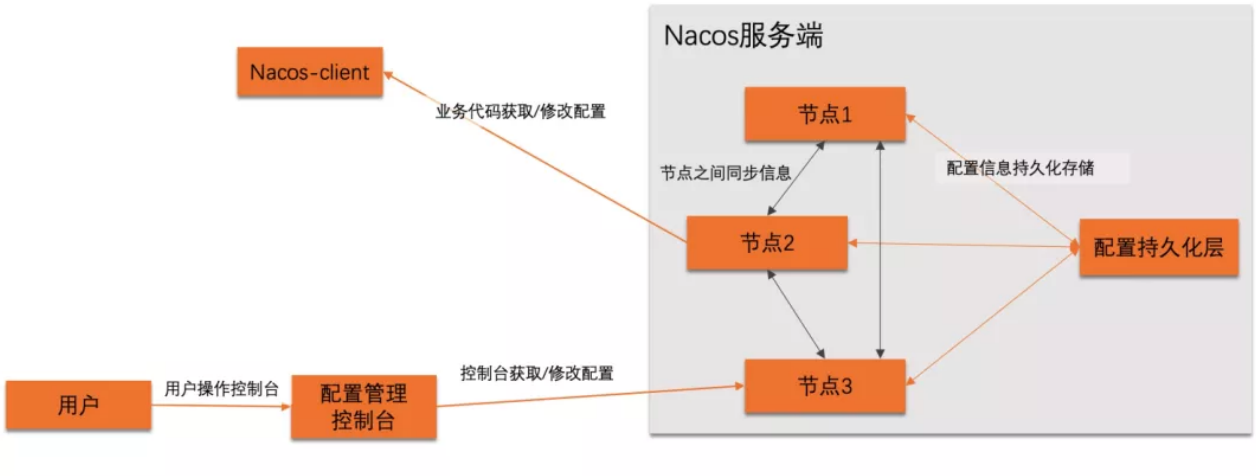
配置中心就是一种统一管理各种应用配置的基础服务组件。配置中心的出现，可以解决这些问题，使得配置信息集中管理，易于维护，并且可以动态更新配置，使得分布式系统更加稳定可靠。



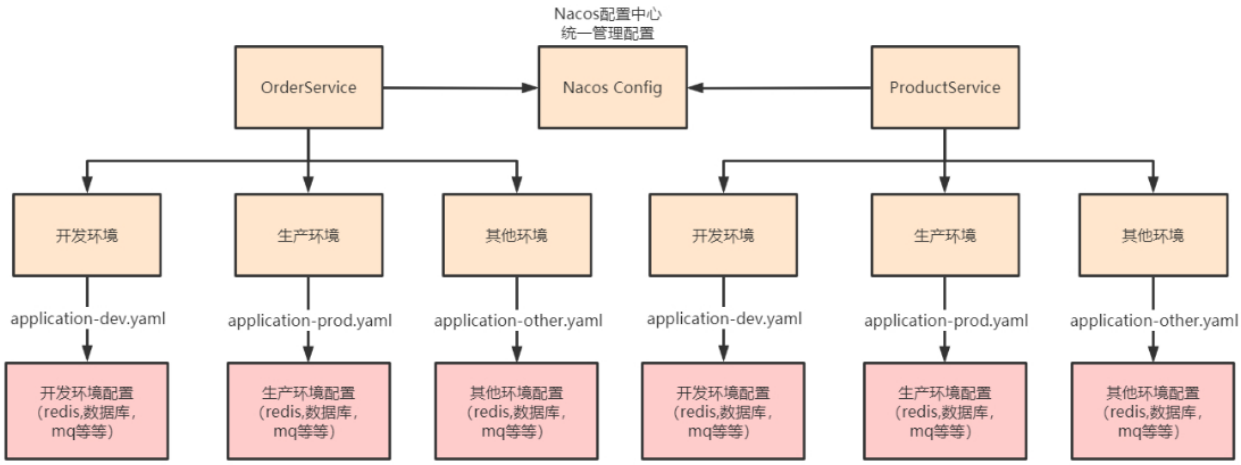
1.2 什么是Nacos配置中心

Nacos 提供用于存储配置和其他元数据的 key/value 存储，为分布式系统中的外部化配置提供服务器端和客户端支持。使用 Spring Cloud Alibaba Nacos Config，您可以在 Nacos Server 集中管理你 Spring Cloud 应用的外部属性配置。

配置中心的架构



应用场景



2. Nacos配置中心实战

2.1 微服务整合Nacos配置中心快速开始

1）准备Nacos Server环境

参考Nacos注册中心笔记搭建Nacos Server环境

2）微服务端整合Nacos配置中心

以mall-user-config-demo为例

2.1）引入依赖

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

常见错误： No spring.config.import set

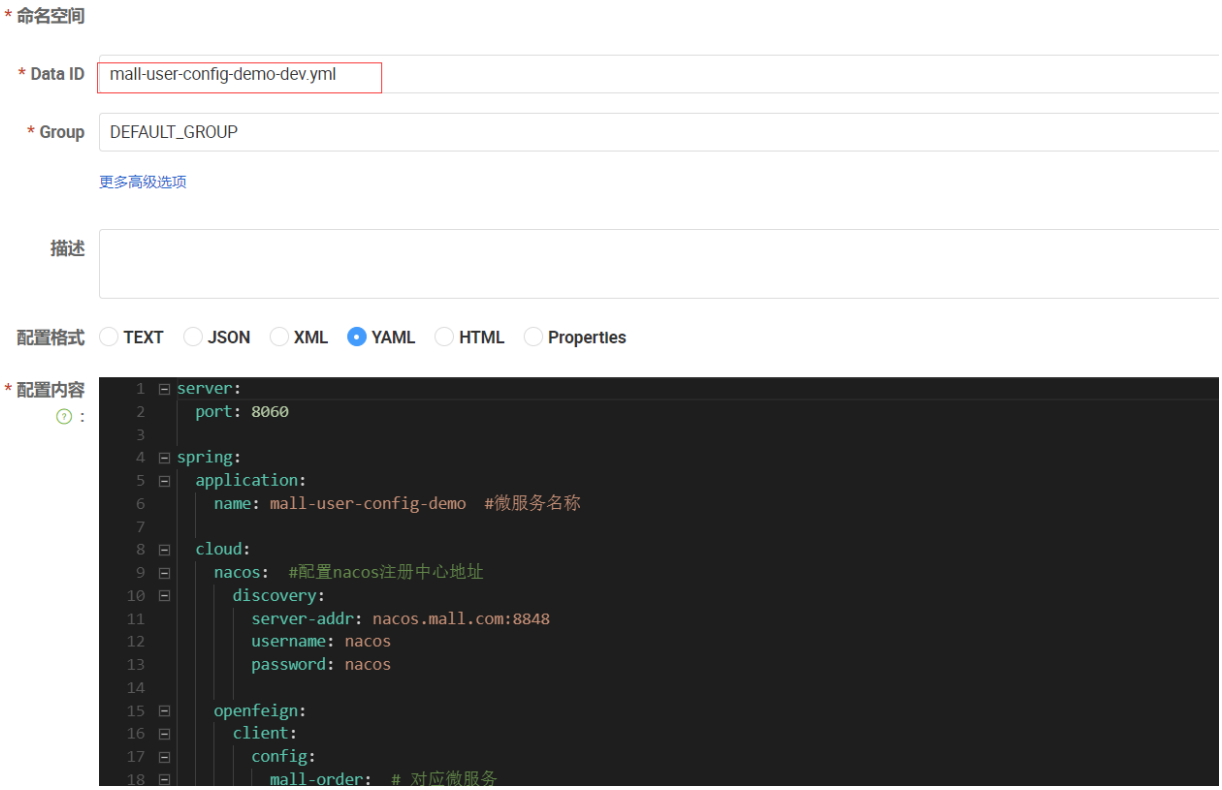
Spring Cloud2020之后，默认没有使用bootstarp的依赖，重新引入spring-cloud-starter-bootstarp的依赖即可解决。

2.2）创建bootstrap.yml文件，配置nacos配置中心的地址

```yaml
spring:
  application:
    name: mall-user-config-demo  #微服务名称

  cloud:
    nacos:
      config:  #配置nacos配置中心地址
        server-addr: nacos.mall.com:8848
        username: nacos
        password: nacos
```

3）将application.yml中的配置移到配置中心， 在配置中心中创建微服务的配置



DateID

Nacos 中的某个配置集的 ID。配置集 ID 是组织划分配置的维度之一。Data ID 通常用于组织划分系统的配置集。一个系统或者应用可以包含多个配置集，每个配置集都可以被一个有意义的名称标识。

Group

Nacos 中的一组配置集，是组织配置的维度之一。通过一个有意义的字符串（如 Buy 或 Trade ）对配置集进行分组，从而区分 Data ID 相同的配置集。当您在 Nacos 上创建一个配置时，如果未填写配置分组的名称，则配置分组的名称默认采用 DEFAULT_GROUP 。配置分组的常见场景：不同的应用或组件使用了相同的配置类型，如 database_url 配置和 MQ_topic 配置。

4）注释掉application.yml中的配置，并在bootstrap.yml中指定需要加载的配置文件的路径

在 Nacos Spring Cloud 中，dataId 的完整格式如下：${prefix}-${spring.profiles.active}.${file-extension}

- prefix 默认为 spring.application.name 的值，也可以通过配置项 spring.cloud.nacos.config.prefix来配置。
- spring.profiles.active 即为当前环境对应的 profile，详情可以参考 Spring Boot文档。 注意：当 spring.profiles.active 为空时，对应的连接符 - 也将不存在，dataId 的拼接格式变成 ${prefix}.${file-extension}
- file-exetension 为配置内容的数据格式，可以通过配置项 spring.cloud.nacos.config.file-extension 来配置。

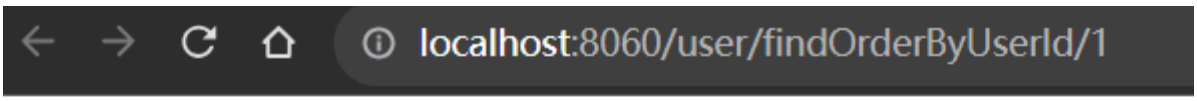
```yaml
spring:
  application:
    name: mall-user-config-demo   #微服务名称

  profiles:
    active: dev #加载开发环境的配置文件

  cloud:
    nacos:
      config:  #配置nacos配置中心地址
        server-addr: nacos.mall.com:8848
        username: nacos
        password: nacos
        file-extension: yml    # 指定配置文件的扩展名为yaml
```

对应的DataID: mall-user-config-demo-dev.yml

5）启动mall-user-config-demo服务，测试调用http://localhost:8060/user/findOrderByUserId/1**，可以正常访问**



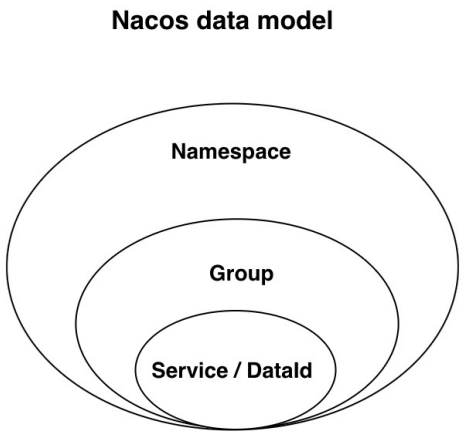

```
{
    msg: "success",
    code: 0,
  - orders: [
      - {
            id: 10,
            userId: 1,
            commodityCode: "C000001457",
            count: 10,
            amount: 200,
        },
      - {
            id: 11,
            userId: 1,
            commodityCode: "C000002356",
            count: 2,
            amount: 50,
        },
    ],
}
```


查看控制台日志，会发现openFeign的配置也是生效的

```
[OrderFeignService#findOrderByUserId] <--- HTTP/1.1 200  (284ms)
[OrderFeignService#findOrderByUserId] connection: keep-alive
[OrderFeignService#findOrderByUserId] content-type: application/json
[OrderFeignService#findOrderByUserId] date: Fri, 18 Aug 2023 06:16:05 GMT
[OrderFeignService#findOrderByUserId] keep-alive: timeout=60
[OrderFeignService#findOrderByUserId] transfer-encoding: chunked
[OrderFeignService#findOrderByUserId]
[OrderFeignService#findOrderByUserId] {"msg":"success","code":0,"orders":[{"id":10,"userId":1,"commodityCo
[OrderFeignService#findOrderByUserId] <--- END HTTP (183-byte body)
[d8b35ba8-9390-40e5-8a69-e3c054976f18] Receive server push request, request = NotifySubscriberRequest, requ
[d8b35ba8-9390-40e5-8a69-e3c054976f18] Ack server push request, request = NotifySubscriberRequest, request
```

2.2 Nacos配置中心常用配置

Nacos 数据模型 Key 由三元组唯一确定, Namespace默认是空串，公共命名空间（public），分组默认是 DEFAULT_GROUP



支持profile粒度的配置

spring-cloud-starter-alibaba-nacos-config 在加载配置的时候，不仅仅加载了以 dataid 为 ${spring.application.name}.${file-extension:properties} 为前缀的基础配置，还加载了dataid为 ${spring.application.name}-${profile}.${file-extension:properties} 的基础配置。在日常开发中如果遇到多套环境下的不同配置，可以通过Spring 提供的 ${spring.profiles.active} 这个配置项来配置。

```
spring.profiles.active=dev
```

支持自定义 namespace 的配置

用于进行租户粒度的配置隔离。不同的命名空间下，可以存在相同的 Group 或 Data ID 的配置。Namespace 的常用场景之一是不同环境的配置的区分隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等。

在没有明确指定 ${spring.cloud.nacos.config.namespace} 配置的情况下， 默认使用的是 Nacos 上 Public 这个namespace。如果需要使用自定义的命名空间，可以通过以下配置来实现：

```
spring.cloud.nacos.config.namespace=71bb9785-231f-4eca-b4dc-6be446e12ff8
```

支持自定义 Group 的配置

Group是组织配置的维度之一。通过一个有意义的字符串（如 Buy 或 Trade ）对配置集进行分组，从而区分 Data ID 相同的配置集。当您在 Nacos 上创建一个配置时，如果未填写配置分组的名称，则配置分组的名称默认采用 DEFAULT_GROUP 。配置分组的常见场景：不同的应用或组件使用了相同的配置类型，如 database_url 配置和 MQ_topic 配置。

在没有明确指定 ${spring.cloud.nacos.config.group} 配置的情况下，默认是DEFAULT_GROUP 。如果需要自定义自己的 Group，可以通过以下配置来实现：

```
spring.cloud.nacos.config.group=DEVELOP_GROUP
```

支持自定义扩展的 Data Id 配置

Data ID 是组织划分配置的维度之一。Data ID 通常用于组织划分系统的配置集。一个系统或者应用可以包含多个配置集，每个配置集都可以被一个有意义的名称标识。Data ID 通常采用类 Java 包（如 com.taobao.tc.refund.log.level）的命名规则保证全局唯一性。此命名规则非强制。

在实际的业务场景中应用和共享配置间的关系可能如下：

- 从单个应用的角度来看： 应用可能会有多套(develop/beta/product)发布环境，多套发布环境之间有不同的基础配置，例如数据库。
- 从多个应用的角度来看：多个应用间可能会有一些共享通用的配置，比如多个应用之间共用一套zookeeper集群。

通过自定义扩展的 Data Id 配置，既可以解决多个应用间配置共享的问题，又可以支持一个应用有多个配置文件。

```
spring:
  application:
    name: mall-user-config-demo  #微服务名称

  profiles:
    active: dev #加载开发环境的配置文件
```

```yaml

  cloud:
    nacos:
      config:  #配置nacos配置中心地址
        server-addr: nacos.mall.com:8848
        username: nacos
        password: nacos
        file-extension: yml   # 指定配置文件的扩展名为yml

        # 自定义 Data Id 的配置
        shared-configs:  #不同工程的通用配置 支持共享的 DataId
          - data-id: nacos.yml
            group: GLOBALE_GROUP
          - data-id: openfeign.yml
            group: GLOBALE_GROUP

        extension-configs:  # 支持一个应用多个 DataId 的配置
          - data-id: common.yml
            group: REFRESH_GROUP
            refresh: true   #支持动态刷新
```

- 通过 spring.cloud.nacos.config.shared-configs[n].data-id 来支持多个共享 Data Id 的配置，多个之间用逗号隔开。 多个共享配置间的一个优先级的关系我们约定：按照配置出现的先后顺序，即后面的优先级要高于前面。 如果没有明确配置，默认情况下所有共享配置的 Data Id 都不支持动态刷新。
- 通过spring.cloud.nacos.config.extension-configs[n].data-id 的配置方式来支持多个 Data Id 的配置。多个 Data Id 同时配置时，他的优先级关系是 n 的值越大，优先级越高。
- 通过spring.cloud.nacos.config.extension-configs[n].group 的配置方式自定义 Data Id 所在的组，不明确配置的话，默认是 DEFAULT_GROUP。
- 通过spring.cloud.nacos.config.extension-configs[n].refresh 的配置方式来控制该 Data Id 在配置变更时，是否支持应用中可动态刷新， 感知到最新的配置值。默认是不支持的。

配置的优先级

Spring Cloud Alibaba Nacos Config 目前提供了三种配置能力从 Nacos 拉取相关的配置。

- A: 通过 spring.cloud.nacos.config.shared-configs 支持多个共享 Data Id 的配置
- B: 通过 spring.cloud.nacos.config.extension-configs[n].data-id 的方式支持多个扩展 Data Id 的配置
- C: 通过内部相关规则(应用名、应用名+ Profile)自动生成相关的 Data Id 配置

当三种方式共同使用时，他们的一个优先级关系是:A < B < C

完整的配置优先级从高到低：

- ${spring.application.name}-${profile}.${file-extension}
- ${spring.application.name}.${file-extension}
- ${spring.application.name}
- extensionConfigs
- sharedConfigs

完全关闭配置

通过设置 spring.cloud.nacos.config.enabled = false 来完全关闭 Spring Cloud Nacos Config。

通过 Nacos Config 对外暴露的 Endpoint查看相关的配置

Nacos Config 内部提供了一个 Endpoint, 对应的 endpoint id 为 nacosconfig。

Endpoint 暴露的 json 中包含了三种属性:

- Sources: 当前应用配置的数据信息
- RefreshHistory: 配置刷新的历史记录
- NacosConfigProperties: 当前应用 Nacos 的基础配置信息

Endpoint 信息查看

1）引入依赖

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

2）暴露监控端点

```
# 暴露所有的端点
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

3) 查看Endpoint信息，访问：http://localhost:8060/actuator/nacosconfig


```
{
  - NacosConfigProperties: {
        serverAddr: "nacos.mall.com:8848",
        username: "nacos",
        password: "nacos",
        encode: null,
        group: "DEFAULT_GROUP",
        prefix: null,
        fileExtension: "yml",
        timeout: 3000,
        maxRetry: null,
        configLongPollTimeout: null,
        configRetryTime: null,
        enableRemoteSyncConfig: false,
        endpoint: null,
        namespace: null,
        accessKey: null,
        secretKey: null,
        ramRoleName: null,
        contextPath: null,
        clusterName: null,
        name: null,
      + sharedConfigs: [2],
      + extensionConfigs: [2],
        refreshEnabled: true,
      + configServiceProperties: {23},
        refreshableDataids: "",
        sharedDataids: "nacos.yml,openfeign.yml",
      + extConfig: [2],
    },
    RefreshHistory: [ ],
  + Sources: [7],
}
```


2.3 配置的动态刷新

spring-cloud-starter-alibaba-nacos-config 也支持配置的动态更新。

测试：当动态配置刷新时，会更新到 Enviroment中

1）修改启动类，每隔3s从Enviroment中获取common.user和common.age中的值

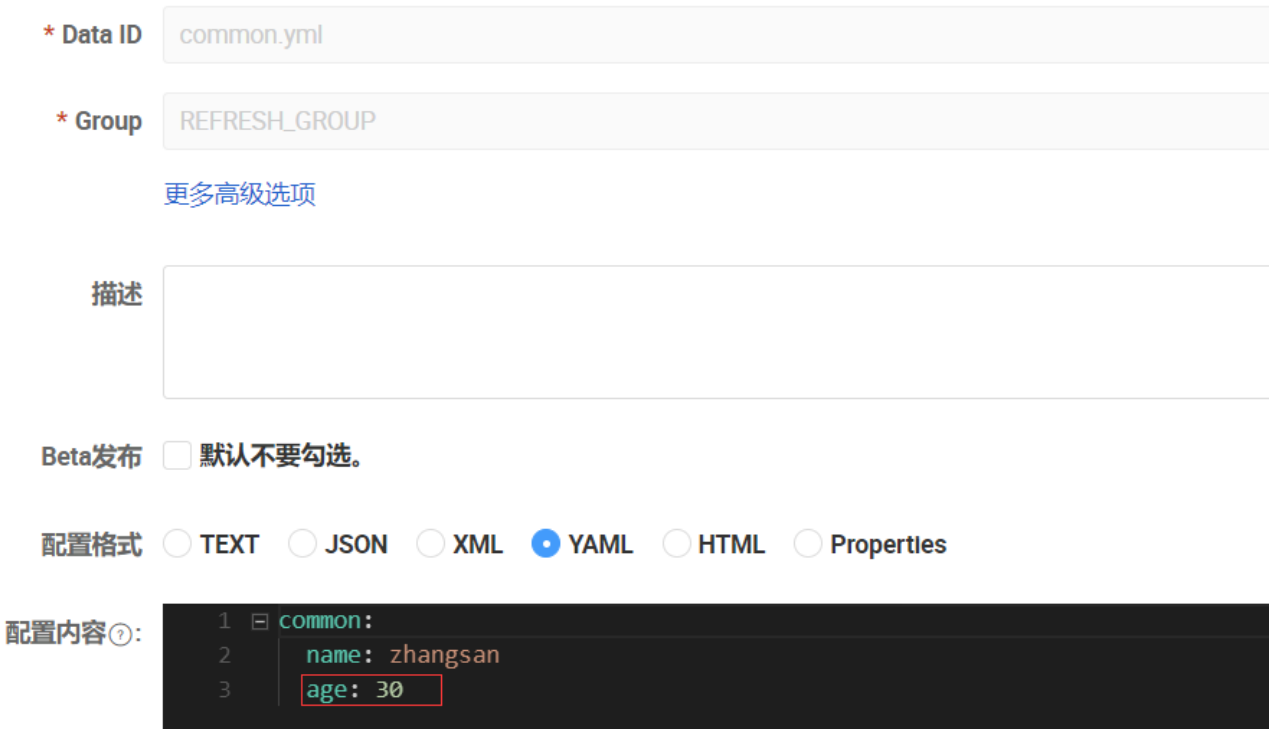
```
public static void main(String[] args) throws InterruptedException{
    ConfigurableApplicationContext applicationContext = SpringApplication.run(MallUserConfigDemoApplication.class, args);

    while (true) {
        //当动态配置刷新时，会更新到 Enviroment中，因此这里每隔3秒中从Enviroment中获取配置
        String userName = applicationContext.getEnvironment().getProperty("common.name");
        String userAge = applicationContext.getEnvironment().getProperty("common.age");
        System.err.println("common name:" + userName + "; age: " + userAge);
        TimeUnit.SECONDS.sleep(3);

    }

}
```

2）进入配置中心，修改common.yml的配置，common.age从10改成30



3）查看控制台的输出，common.age的值是否发生变化

OpenFeign开启对feign.Request.Options属性的刷新支持

> https://docs.spring.io/spring-cloud-openfeign/docs/current/reference/html/#spring-refreshscope-support

```
#启用刷新功能,可以刷新connectTimeout和readTimeout
spring.cloud.openfeign.client.refresh-enabled=true
```

@RefreshScope实现Bean的动态刷新

下面例子中，使用@Value注解可以获取到配置中心的值，但是无法让IndexController动态感知修改后的值，需要利用@RefreshScope注解修饰。

```
@RestController
@RefreshScope
public class IndexController {

    @Value("${common.age}")
    private String age;
    @Value("${common.name}")
    private String name;

    @GetMapping("/index")
    public String hello() {
        return name+","+age;
    }

}

```

测试结果： 使用@RefreshScope修饰的IndexController，访问/index结果可以获取最新的值

@RefreshScope 导致@Scheduled定时任务失效问题

当利用@RefreshScope刷新配置后会导致定时任务失效

```
@SpringBootApplication
@EnableScheduling   // 开启定时任务功能
public class NacosConfigApplication {
}

@RestController
@RefreshScope  //动态感知修改后的值
public class TestController {

    @Value("${common.age}")
     String age;
    @Value("${common.name}")
     String name;

    @GetMapping("/common")
    public String hello() {
        return name+","+age;
    }

    //触发@RefreshScope执行逻辑会导致@Scheduled定时任务失效
    @Scheduled(cron = "*/3 * * * * ?")  //定时任务每隔3s执行一次
    public void execute() {
        System.out.println("定时任务正常执行。。。。。。");
    }


}
```

测试结果：

- 当在配置中心变更属性后，定时任务失效
- 当再次访问http://localhost:8060/common，定时任务生效

原因：@RefreshScope修饰的bean的属性发生变更后，会从缓存中清除。此时没有这个bean，定时任务当然也就不生效了。

详细原因如下：

1. @RefreshScope 注解标注了@Scope 注解，并默认了ScopedProxyMode.TARGET_CLASS属性，此属性的功能就是创建一个代理，在每次调用的时候都用它来调用GenericScope#get 方法来获取bean对象。
2. 在GenericScope 里面包装了一个内部类 BeanLifecycleWrapperCache 来对加了 @RefreshScope 的bean进行缓存，使其在不刷新时获取的都是同一个对象。
3. 如属性发生变更会调用 ContextRefresher#refresh()——>RefreshScope#refreshAll() 进行缓存清理方法调用，并发送刷新事件通知 ——> 调用GenericScope#destroy() 实现清理缓存
4. 当下一次使用此bean对象时，代理对象会调用GenericScope#get(String name, ObjectFactory<?> objectFactory) 方法创建一个新的bean对象，并存入缓存中，此时新对象因为Spring 的装配机制就是新的属性了

后面会结合源码分析，核心源码：GenericScope#get

解决方案

实现Spring事件监听器，监听 RefreshScopeRefreshedEvent事件，监听方法中进行一次定时方法的调用

```
@RestController
@RefreshScope  //动态感知修改后的值
public class TestController implements ApplicationListener<RefreshScopeRefreshedEvent>{

    @Value("${common.age}")
     String age;
    @Value("${common.name}")
     String name;

    @GetMapping("/common")
    public String hello() {
        return name+","+age;
    }

    //触发@RefreshScope执行逻辑会导致@Scheduled定时任务失效
    @Scheduled(cron = "*/3 * * * * ?")  //定时任务每隔3s执行一次
    public void execute() {
        System.out.println("定时任务正常执行。。。。。。");
    }


    @Override
```

```
    public void onApplicationEvent(RefreshScopeRefreshedEvent event) {
        this.execute();
    }
}
```

2.4 Nacos插件扩展：配置加密

为保证用户敏感配置数据的安全，Nacos 提供了配置加密的新特性。降低了用户使用的风险，也不需要再对配置进行单独的加密处理。

> 参考文档：https://nacos.io/zh-cn/docs/v2/plugin/config-encryption-plugin.html

Nacos 加解密插件是可插拔的，有没有都不影响 Nacos 的核心功能的运行。如果想要使用 Naocs 的配置加解密功能需要单独引用加密算法的实现。

在 Nacos 服务端启动的时候就会加载所有依赖的加解密算法，然后通过发布配置的 `dataId` 的前缀(cipher-[加密算法名称])来进行匹配是否需要加解密和使用的加解密算法。

客户端发布的配置会在客户端通过filter完成加解密，也就是配置在传输过程中都是密文的。而控制台发布的配置会在服务端进行处理。

客户端和服务端都通过添加以下依赖来使用 AES 加解密算法，服务端推荐添加到 config 模块下。

```
<!-- 引入配置加密插件 -->
<dependency>
    <groupId>com.alibaba.nacos</groupId>
    <artifactId>nacos-aes-encryption-plugin</artifactId>
    <version>1.0.0-SNAPSHOT</version>
</dependency>
```

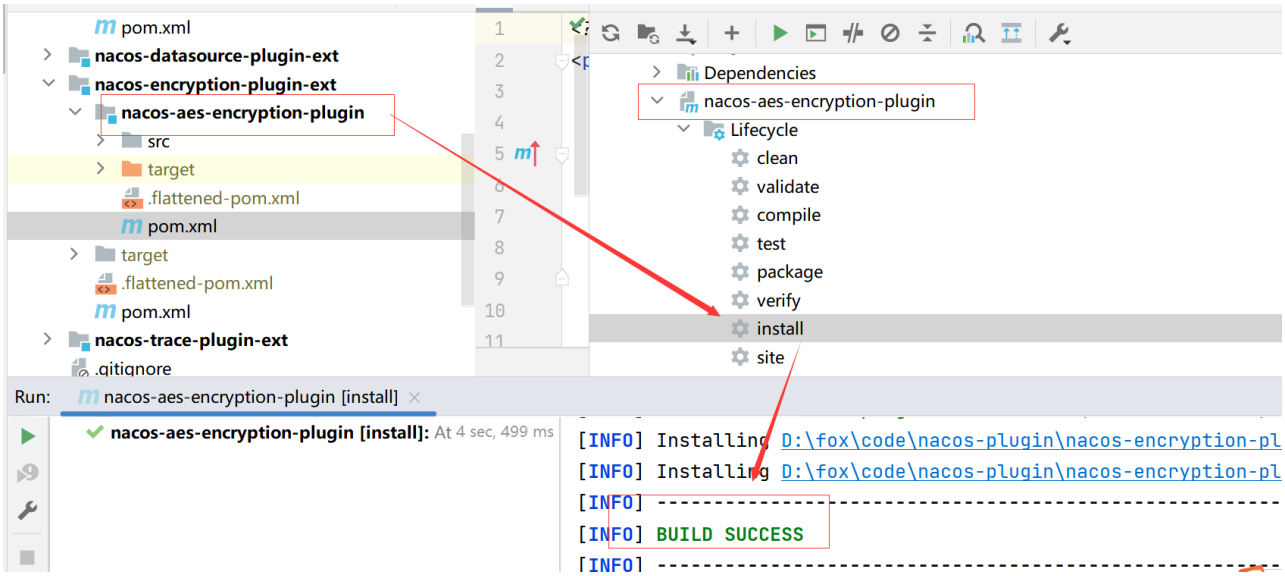
注意：目前插件需要自己编译,并未上传至maven中央仓库

1） 编译nacos-aes-encryption-plugin插件

通过 SPI 的机制抽象出加密和解密的操作，Nacos 默认提供 AES 的实现。用户也可以自定义加解密的实现方式。具体的实现在 nacos-plugin 仓库。

```
git clone git@github.com:nacos-group/nacos-plugin.git
mvn install
```

编译nacos-aes-encryption-plugin插件



如果不想自己编译，可以直接下载下面的jar包：

2) 在nacos的config包下引入nacos-aes-encryption-plugin依赖，重新编译nacos服务端源码

3）创建加密配置

配置前缀使用cipher-[加密算法名称]-dataId来标识这个配置需要加密，系统会自动识别并加密。例如使用 AES 算法来解密配置：cipher-aes-nacos.yml。

查看mysql数据库存储的数据，是否加密：

id	data_id	group_id	content	md5
10	cipher-aes-nacos.yml	DEFAULT_GROUP	3a2b7a4600f88fe3c4265f2228b88db35a442ddeeb18c15d444b5164339fe6d96e20052cf326aceb5cac2d459a945cd4cf1674db33107b432	fd768
11	nacos.yml	DEFAULT_GROUP	spring: cloud: nacos: #配置nacos注册中心地址 discovery: server-addr: nacos.mall.com:8848 username: nacos password: na	2d537

存储到mysql的是加密后的配置