Spring 动态代理实现新老路径的一键切换

前言

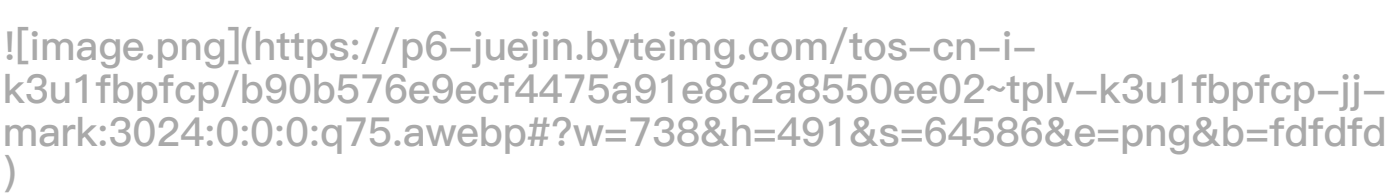
本篇文章主要介绍了代码迁移开关的技术需要，以及**使用 Spring 动态代理以及动态 Bean 注册**的功能，实现迁移路径收束的一键控制。

背景

众所周知，由于 usercenter 中的业务域在银行架构中，应该处于其他业务域的上层，不应被业务域服务所依赖。

但现实就是，usercenter 管理了客户在 APP 上的用户信息，其他各种场景不免需要依赖这些信息，简直”倒反天罡”。

所以，就需要把这些用户信息下沉到业务域之下的统一域，产生了这次迁移。



加开关

虽然，本次行为，主要是对数据读写逻辑的迁移，数据库表不做迁移。但这依然是一个危险的动作。

* 1.usercenter 是一个基础服务，一旦出现故障，影响面很广
* 2.new-usercenter 会把一些读写行为合并，可能会有相互影响。
* 3.usercenter 从直接读写数据库模型，到读写接口 DTO，并且受到接口合并的影响，势必会对业务逻辑的代码做些调整。

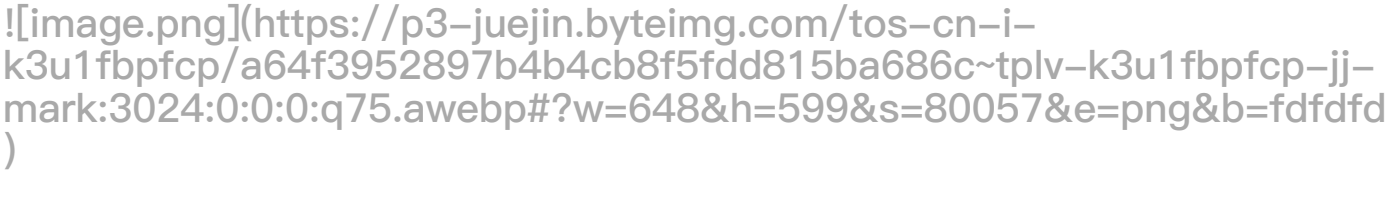
所以基于风险考虑，最好有一个开关，如果生产出现问题，可以快速切回旧逻辑。

而且切回以后，由于数据库表没有变化，仅需对问题数据做处理即可。

那么问题来了，怎么加开关?

之前考虑有两种方案

* 在业务层与数据层做开关，在核心业务代码不改动的情况下，实现对数据读写的转变。
* 在业务层上做开关，通过接口屏蔽对调用方（接口层或其他业务层类）。



最后选择了在业务层之上做了一层开关，为了是最大限度保留原代码，减少本次迁移对其的影响。

就像上面所说的风险，由于调用方式、模型、逻辑收束的变化，数据层方案，即使要做一层防腐层，也势必会对原有 inner service 造成修改。

所以在复制了 inner service 的基础上，以 new service 为新路径做迁移。

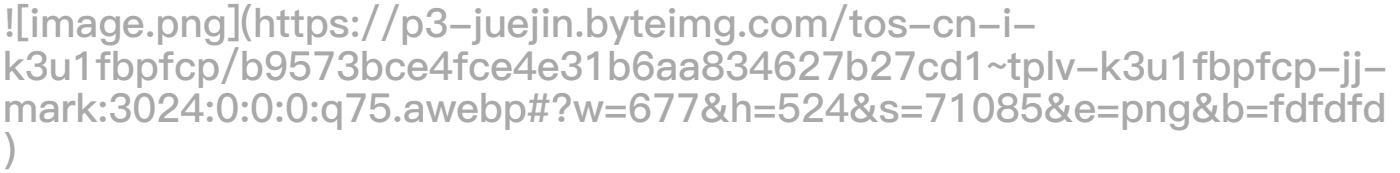
硬编码方案
=====

好的，经过了一段长时间的复制、接入、调整，我们已经创建了新的 service，接入了 new usercenter。

那剩下就简单了，就是在 Controller 调用 Service 或 Service 相互调用的地方，分别注入 inner service 和 new service，根据开关的值，选择调用 inner service 和 new service。

那么，第一个问题来了，一个方法，可能会被多方调用，不能每次都重复写一次吧。

所以，大家都会的，搞一个实现两个 service 共有接口的代理类来处理开关，让 Controller 只调代理类。



但是，还有第二个问题，inner service 很多，20多个。

然后方法还多，一个类可以多达十几二十个，这个乘法就很明白了。

所以，“懒惰”是第一生产力，这不能自己搞，让 Spring 来做。

动态代理方案

生成代理类

就像上面图里写的，这20几个类，共几百个方法的Proxy Service，得让 Spring 来写。

众所周知，Spring 代理方式有两种 JDK 以及 Cglib：

* 基于JDK的动态代理 基于接口的动态代理，用到的类是Proxy的newProxyInstance静态方法创建，要求被代理对象至少实现一个接口，如果没有，则不能创建代理对象。

* 基于cglib的动态代理 要导入cglib第三方库，使用的类是Enhancer的create静态方法创建，要求被代理类不能是最终类，即不能用final修饰，如String类。

这里不细说，可见 [Spring学习(五)：动态代理的两种实现方式（全网最容易懂）_spring动态代理的两种方式-CSDN博客](http://cxyroad.com/”https://blog.csdn.net/qq_40205337/article/details/111410861”)

因为所有的 service 都有实现接口，所以优先采用了 JDK 的方式。

```
Proxy.newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)
```

核心方法需要三个参数：

* ClassLoader：咱们直接使用接口对应的 ClassLoader 即可
* Class[]：就是代理类需要实现的接口
* **InvocationHandler：具体实现的代理逻辑**

ClassLoader & Class[] 都很清楚，InvocationHandler 也很简单，上述的图中已经画出来了。

```
public class InnerServiceProxy<T> implements InvocationHandler {
    private final ZaLogger log = ZaLoggerFactory.getLogger(getClass());

    private ApplicationContext applicationContext;
    private Class<T> interfaceClass;
    private INewService newService;
    private T originService;

    // 注入两个 service
    public Object bind(Class<T> cls, ApplicationContext applicationContext) {
        interfaceClass = cls;
        this.applicationContext = applicationContext;
        String[] beanNames =
```

```
applicationContext.getBeanNamesForType(interfaceClass);
        Arrays.stream(beanNames).forEach(name -> {
            if (name.endsWith("Proxy")) {
                log.info("跳过代理类{}", name);
                return;
            }
            T bean = applicationContext.getBean(name, interfaceClass);
            if (bean instanceof INewService) {
                log.info("{}注入迁移 Service{}", interfaceClass, bean);
                newService= (INewService) bean;
            } else {
                log.info("{}注入原始 Service{}", interfaceClass, bean);
                originService = bean;
            }
        });
        Assert.notNull(originService, "原始 Service 不能为空");
        return Proxy.newProxyInstance(cls.getClassLoader(), new Class[]
{cls}, this);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        if (Objects.isNull(newService)) {
            log.info("{} 执行, 无迁移 Service, 执行原始 Service,bean={}",
interfaceClass.getSimpleName(), originService);
            return method.invoke(originService, args);
        }
        Boolean migrateSwitch =
applicationContext.getEnvironment().getProperty("migrate.switch",
Boolean.class);
        log.info("迁移配置={}", migrateSwitch);
        if (BooleanUtils.isTrue(migrateSwitch)) {
            log.info("执行 {}, 迁移 service, config={}, bean={}",
interfaceClass.getSimpleName(), migrateSwitch , newService);
            return method.invoke(newService, args);
        }
        log.info("执行 {}, 原始 service, config={}, bean={}",
interfaceClass.getSimpleName(), migrateSwitch , originService);
        return method.invoke(originService, args);
    }
}

```

将代理类 Bean Definition 注册到 Spring 容器中

有了代理类实现，就把代理类的 Bean Definition 注册到容器中，且是涉及的所有接口。

那就需要用到Spring的扩展点：public interface BeanDefinitionRegistryPostProcessor。

所有实现了该接口的类，会在 Spring 容器准备好后，就自动被 Spring 容器调用执行实现的方法。

图源：[Spring IOC源码解析：Spring容器启动流程_spring ioc容器启动过程-CSDN博客](http://cxyroad.com/”https://blog.csdn.net/zzti_erlie/article/details/114897454?ops_request_misc=&request_id=&biz_id=102&utm_term=Spring%20%E5%AE%B9%E5%99%A8%E5%88%9D%E5%A7%8B%E5%8C%96%E8%BF%87%E7%A8%8B&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-3-114897454.142%5Ev100%5Epc_search_result_base3&spm=1018.2226.3001.4187”)

```
public class InnerServiceRegistryBean implements ApplicationContextAware, BeanDefinitionRegistryPostProcessor {
    private final ZaLogger log = ZaLoggerFactory.getLogger(getClass());
    private ApplicationContext ctx;
    @Autowired
    private ResourceLoader resourceLoader;
    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {// do nothing}
    @Override
    public void setApplicationContext(ApplicationContext ctx) throws BeansException {this.ctx = ctx;}

    @Override
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry beanDefinitionRegistry) throws BeansException {
```

```
        ResourcePatternResolver resolver =
ResourcePatternUtils.getResourcePatternResolver(resourceLoader);
        MetadataReaderFactory metaReader = new
CachingMetadataReaderFactory(resourceLoader);
        Resource[] resources;
        try {
            // 扫描 inner 包下所有文件
            resources =
resolver.getResources("classpath*:com/usercenter/service/inner/**/*.cl
ass");
            for (Resource resource : resources) {
                MetadataReader reader =
metaReader.getMetadataReader(resource);
                String className =
reader.getClassMetadata().getClassName();
                Class<?> cls = Class.forName(className);
                // 只需要代理实现接口
                if (!cls.isInterface()) {
                    log.info("不是接口, 跳过代理{}", cls);
                    continue;
                }
                // 根据接口创建一个 BeanDefinition
                BeanDefinitionBuilder builder =
BeanDefinitionBuilder.genericBeanDefinition(cls);
                GenericBeanDefinition definition = (GenericBeanDefinition)
builder.getRawBeanDefinition();
                // 注入属性，接口类型、以及 上下文（用于后续初始化或执行时
，获取新老 bean 以及属性）
                definition.getPropertyValues().add("interfaceClass", cls);
                definition.getPropertyValues().add("applicationContext", ctx);
                // 注意，这里的实现类是 InnerServiceProxyFactory，不是
InnerServiceProxy
                definition.setBeanClass(InnerServiceProxyFactory.class);

 definition.setAutowireMode(GenericBeanDefinition.AUTOWIRE_BY_TYP
E);
                // 将代理类设为 Primary，所有依赖该接口的地方，优先注入该代
理类的实现
                definition.setPrimary(true);
                // 固定后缀，方便后续判断是代理类

 beanDefinitionRegistry.registerBeanDefinition(cls.getSimpleName() +
"Proxy", definition);
            }
        } catch (IOException | ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
```

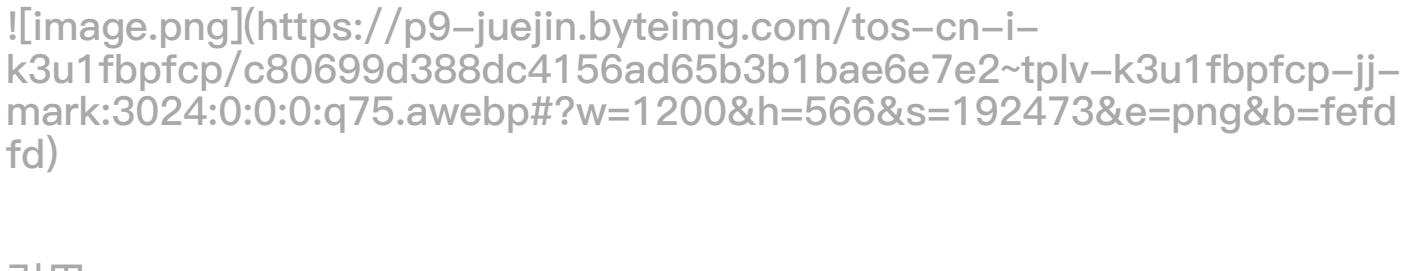
```

可以看到上面注入 BeanClass 是 InnerServiceProxyFactory，不是之前写的 InnerServiceProxy。

是因为 InnerServiceProxy 本身是对原有方法进行增强（代理）的一段逻辑，本身并不是原有方法所在接口或类的继承。

关系类似于这样：

```
 class Proxy {
    private InvocationHandler invocationHandler;
    public void doSomeThing() {
        invocationHandler.invoke();
    }
  }
```

又因为代理类是直接通过 Proxy 类在运行时创建的，且只有一个参数为 InvocationHandler 的构造器。所以无法直接把代理类直接设为 BeanClass。

因此，通过 FactoryBean 做一层包装，以保证 BeanClass 既是原始接口，又能将初始化后的代理对象放入容器管理。

```
public class InnerServiceProxyFactory<T> implements FactoryBean<T> {
    @Getter
    @Setter
    private Class<T> interfaceClass;
    @Getter
    @Setter
    private ApplicationContext applicationContext;
    @Override
    public T getObject() throws Exception {
        // InnerServiceProxy 创建指定接口和增强逻辑的对象
        return (T) new InnerServiceProxy().bind(interfaceClass,
applicationContext);
    }
```
```

```
    @Override
    public Class<?> getObjectType() {
        return interfaceClass;
    }

    @Override
    public boolean isSingleton() {
        // 单例模式
        return true;
    }
}

```

当然，不想用 Factory ，就从 Proxy 拿到代理类，然后直接指定构造函数

```
definition.setBeanClass(Proxy.getProxyClass(IUserService.class.getClass
Loader(), cls));
ConstructorArgumentValues constructorArgumentValues = new
ConstructorArgumentValues();
constructorArgumentValues.addIndexedArgumentValue(0, new
ConstructorArgumentValues.ValueHolder(new InnerServiceProxy<>()));
definition.setConstructorArgumentValues(constructorArgumentValues);

```

至此，方案讲完了，总体的流程像是下图呈现的。



引用
==

* [Spring学习(五)：动态代理的两种实现方式（全网最容易懂）_spring动态代理的两种方式-CSDN博客](http://cxyroad.com/
”https://blog.csdn.net/qq_40205337/article/details/111410861”)
 原文链接: https://juejin.cn/post/7367286576127262732