动态代理使用的场景还有哪些呢？

（1）利用动态代理，能够实现在方法执行前后加入额外的逻辑处理；例如Hook Activity的启动流程，常用在插件化的框架中;

（2）利用动态代理，能够实现解耦，使得调用层与实现层分离，例如Retrofit框架；

（3）动态代理不需要接口的实现类，常用于IPC进程间通信；

（4）动态代理可以解决程序的执行流程，例如反射调用某个方法，需要传入一个接口实现类，就会使用到动态代理；

动态代理深入

动态代理的例子

private fun testProxy() {

val proxy = Proxy.newProxyInstance(

classLoader,

arrayOf(IProxyInterface::class.java)

) { obj, method, args ->

Log.e("TAG", "方法调用前------")

return@newProxyInstance handleMethod()

} as IProxyInterface

/\*\*调用方法\*/

val result = proxy.getName()

Log.e("TAG", "result==>$result")

}

private fun handleMethod(): Any? {

Log.e("TAG", "开始执行方法--")

return "小明~"

}

当通过Proxy的newProxyInstance方法创建一个IProxyInterface的代理对象的时候，其实这个接口并没有任何实现类

interface IProxyInterface {

fun getName(): String

}

只有一个getName方法，那么当这个代理对象调用getName()方法的时候，就会先走到InvocationHandler的方法体内部，handleMethod方法我们可以认为是接口方法的实现，所以在方法实现之前，可以做一些前置的操作。

2022-11-26 20:25:07.960 403-403/com.lay.mvi E/TAG: 方法调用前------

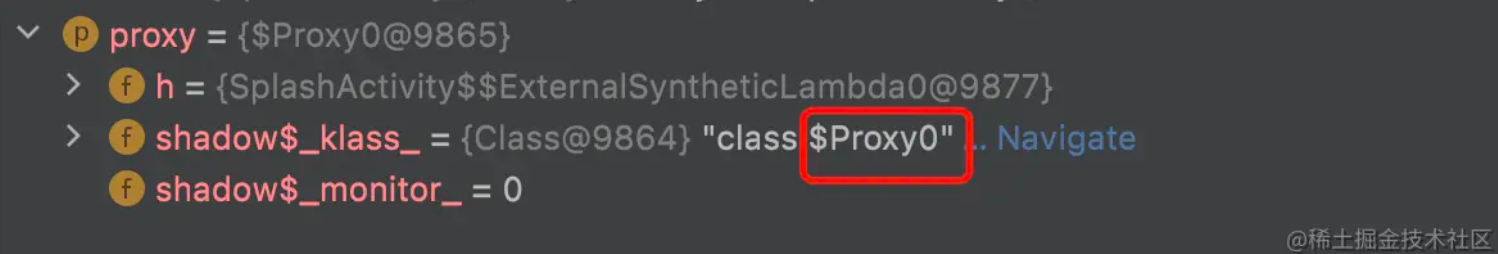
2022-11-26 20:25:07.960 403-403/com.lay.mvi E/TAG: 开始执行方法--

2022-11-26 20:25:07.960 403-403/com.lay.mvi E/TAG: result==>小明~

$Proxy0

所以，当我们创建一个接口之后，并不需要实例化该接口，而是采用动态代理的方式生成一个代理对象，从而实现调用层与实现层的分离，这样也是解耦的一种方式。

那么生成的IProxyInterface代理对象是接口吗？肯定不是，因为接口不可实例化，那么生成的对象是什么呢？



过断点，我们发现这个对象是$Proxy0，那么这个对象是怎么生成的呢？

public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)

throws IllegalArgumentException

{

Objects.requireNonNull(h);

final Class<?>[] intfs = interfaces.clone();

/\*

\* Look up or generate the designated proxy class.

\*/

Class<?> cl = getProxyClass0(loader, intfs);

/\*

\* Invoke its constructor with the designated invocation handler.

\*/

try {

final Constructor<?> cons = cl.getConstructor(constructorParams);

final InvocationHandler ih = h;

if (!Modifier.isPublic(cl.getModifiers())) {

// BEGIN Android-removed: Excluded AccessController.doPrivileged call.

/\*

AccessController.doPrivileged(new PrivilegedAction<Void>() {

public Void run() {

cons.setAccessible(true);

return null;

}

});

\*/

cons.setAccessible(true);

// END Android-removed: Excluded AccessController.doPrivileged call.

}

return cons.newInstance(new Object[]{h});

} catch (IllegalAccessException|InstantiationException e) {

throw new InternalError(e.toString(), e);

} catch (InvocationTargetException e) {

Throwable t = e.getCause();

if (t instanceof RuntimeException) {

throw (RuntimeException) t;

} else {

throw new InternalError(t.toString(), t);

}

} catch (NoSuchMethodException e) {

throw new InternalError(e.toString(), e);

}

}

其实我们也能够看到，通过getProxyClass0方法目的就是查找或者生成一个代理的Class对象，并通过反射创建一个实体类，其实就是$Proxy0

那么调用getName方法，其实就是调用$Proxy0的getName方法，最终内部就是调用了InvocationHandler的invoke方法。

动态代理实现Xutils

如果没有使用过ViewBinding的伙伴，可能在项目中大多都是用ButterKnife这些注入框架，那么对于这类依赖注入工具，我们该如何亲自实现呢？这就使用到了注解配合动态代理，这里我们先忘记ViewBinding。

Android属性注入

在日常的开发过程中，我们经常需要通过findViewById获取组件，并设置点击事件；或者为页面设置一个layout布局，每个页面几乎都需要设置一番，那么通过事件注入，就可以大大简化我们的流程。

/\*\*运行时注解，放在类上使用\*/

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.TYPE)

public @interface setContentView {

/\*\*布局id\*/

int value();

}

那么我们以布局注入为例，介绍一下事件是如何被注入进去的。

@RequiresApi(api = Build.VERSION\_CODES.N)

public class InjectUtils2 {

public static void inject(Context context) {

injectContentView(context);

}

private static void injectContentView(Context context) {

/\*\*获取布局id\*/

Class<?> aClass = context.getClass();

try {

setContentView setContentView = aClass.getDeclaredAnnotation(setContentView.class);

if (setContentView == null) {

return;

}

int layoutId = setContentView.value();

/\*\*反射获取Activity的setContentView方法\*/

Method setContentViewMethod = aClass.getMethod("setContentView", int.class);

setContentViewMethod.setAccessible(true);

setContentViewMethod.invoke(context, layoutId);

} catch (Exception e) {

}

}

}

这里我们采用反射的方式，判断类上方是否存在setContentView注解，如果存在，那么就反射调用Activity的setContentView方法。

这里为什么使用Java，是因为在反射的时候，如果反射的源码为Java代码，最好使用Java，否则与Kotlin的类型不匹配会导致反射失败。

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.FIELD)

public @interface viewId {

int value();

}

对于控件的注入，类似于ViewBinding

private static void injectView(Context context) {

Class<?> aClass = context.getClass();

try {

Field[] declaredFields = aClass.getDeclaredFields();

if (declaredFields.length == 0) {

return;

}

for (Field field : declaredFields) {

/\*\*判断当前属性是否包含viewId注解\*/

viewId viewId = field.getDeclaredAnnotation(viewId.class);

if (viewId != null) {

/\*\*获取id值\*/

int id = viewId.value();

/\*\*执行findViewById操作\*/

Method findViewById = aClass.getMethod("findViewById", int.class);

findViewById.setAccessible(true);

field.setAccessible(true);

field.set(context, findViewById.invoke(context, id));

}

}

} catch (Exception e) {

Log.e("TAG","exp===>"+e.getMessage());

}

}

具体的使用如下

@setContentView(R.layout.activity\_splash)

class SplashActivity : BaseActivity() {

@viewId(R.id.tv\_music)

private var tv\_music: TextView? = null

override fun initView() {

JUCTest.test()

Singleton.getInstance().increment()

testProxy()

tv\_music?.setOnClickListener {

Toast.makeText(this, "点击了", Toast.LENGTH\_SHORT).show()

}

}

动态代理实现事件注入

前面我们介绍了布局的注入以及属性的注入，其实这两个事件还是很简单的，通过反射赋值即可。但是如果是一个点击事件，就不是单纯的赋值了，就需要使用到动态代理了。

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

public @interface OnClick {

int[] value();

}

对于Android的事件来说有很多种，像点击事件、长按事件、滑动事件等等，如果只是像上面的注解一样，只有一个id，显然是不够的。

拿点击事件来说，需要三要素：setOnClickListener、OnClickListener对象、回调onClick

tv\_music?.setOnClickListener {

Toast.makeText(this, "点击了", Toast.LENGTH\_SHORT).show()

}

那么这些可以放在注解中，在调用的时候传入，但是对于用户来说，肯定只需要传入id就可以了，而不需要在外层传一堆乱七八糟的东西

@OnClick(value = [R.id.tv\_music],function="setOnClickListener",......)

private fun clickButton() {

}

那么这些操作就需要在注解内部处理。

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.ANNOTATION\_TYPE)

public @interface EventBase {

/\*\*设置监听的类型，例如setOnClickListener、setOnTouchListener......\*/

String listenerSetter();

/\*\*匿名内部类类型，例如OnClickListener.class\*/

Class<?> listenerType();

/\*\*回调方法\*/

String callbackMethod();

}

****这里首先定义了一个注解的基类，里面定义了事件的三要素，目的就是给上层注解提供实现类似于继承的方式****。

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

@EventBase(listenerSetter = "setOnClickListener", listenerType = View.OnClickListener.class, callbackMethod = "onClick")

public @interface OnClick {

int[] value();

}

接下来就可以通过反射获取方法上的注解

private static void injectClick(Context context) {

Class<?> aClass = context.getClass();

try {

Method[] methods = aClass.getDeclaredMethods();

if (methods.length == 0) {

return;

}

/\*\*处理单击事件\*/

for (Method method : methods) {

Annotation[] annotations = method.getDeclaredAnnotations();

if (annotations.length > 0) {

for (Annotation annotation : annotations) {

EventBase eventBase = annotation.annotationType().getAnnotation(EventBase.class);

if (eventBase == null) {

continue;

}

/\*\*拿到事件三要素\*/

String listenerSetter = eventBase.listenerSetter();

Class<?> listenerType = eventBase.listenerType();

String callbackMethod = eventBase.callbackMethod();

/\*\*拿到注解中传入的id\*/

Method values = annotation.getClass().getDeclaredMethod("values");

values.setAccessible(true);

int[] componentIds = (int[]) values.invoke(annotation);

for (int id : componentIds) {

/\*\*反射获取到这个id对应的组件\*/

Method findViewById = aClass.getMethod("findViewById", int.class);

findViewById.setAccessible(true);

View view = (View) findViewById.invoke(context, id);

/\*\*反射获取事件方法，注意这里类型是动态的\*/

Method setListenerMethod = view.getClass().getMethod(listenerSetter, listenerType);

/\*\*执行这个事件\*/

setListenerMethod.setAccessible(true);

setListenerMethod.invoke(view, buildProxyInstance(listenerType, context, method));

}

}

}

}

} catch (Exception e) {

Log.e("TAG", "injectClick exp===>" + e.getMessage());

}

}

/\*\*

\* 根据listener类型创建动态代理对象

\*

\*/

private static Object buildProxyInstance(Class<?> listenerType, Context context, Method callbackMethod) {

return Proxy.newProxyInstance(listenerType.getClassLoader(), new Class<?>[]{listenerType}, new InvocationHandler() {

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

Log.e("TAG", "调用前处理--");

callbackMethod.setAccessible(true);

return callbackMethod.invoke(context);

}

});

}

这里通过反射获取时，完全是根据listenerSetter属性动态查找，而不是写死一个方法，这种方式使用起来具备扩展性。

public interface OnClickListener {

/\*\*

\* Called when a view has been clicked.

\*

\* @param v The view that was clicked.

\*/

void onClick(View v);

}

因为这里采用的是动态代理的方式，动态创建一个OnClickListener对象，并作为setOnclickListener方法的参数传入进去，所以当onClick执行的时候，会走到InvocationHandler的invoke方法中，在这里执行了应用层的方法。

@OnClick(values = [R.id.tv\_music])

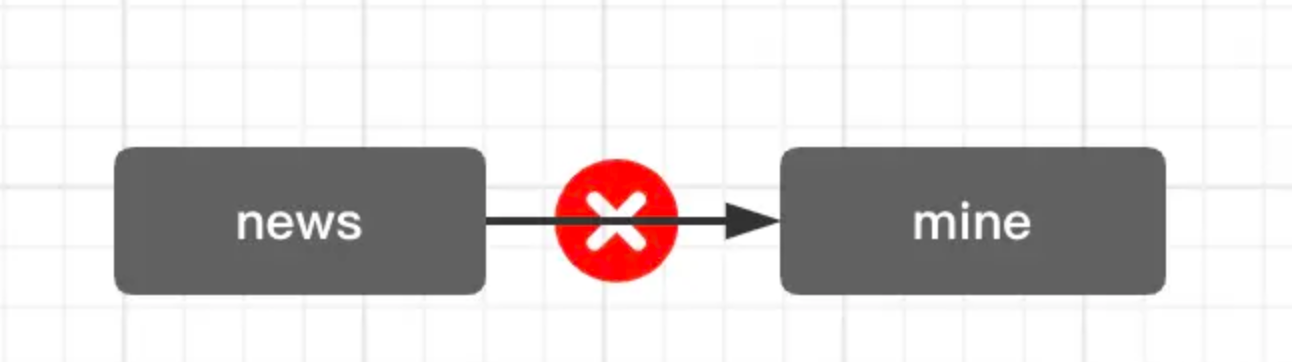
private fun clickButton() {

Toast.makeText(this, "点击了", Toast.LENGTH\_SHORT).show()

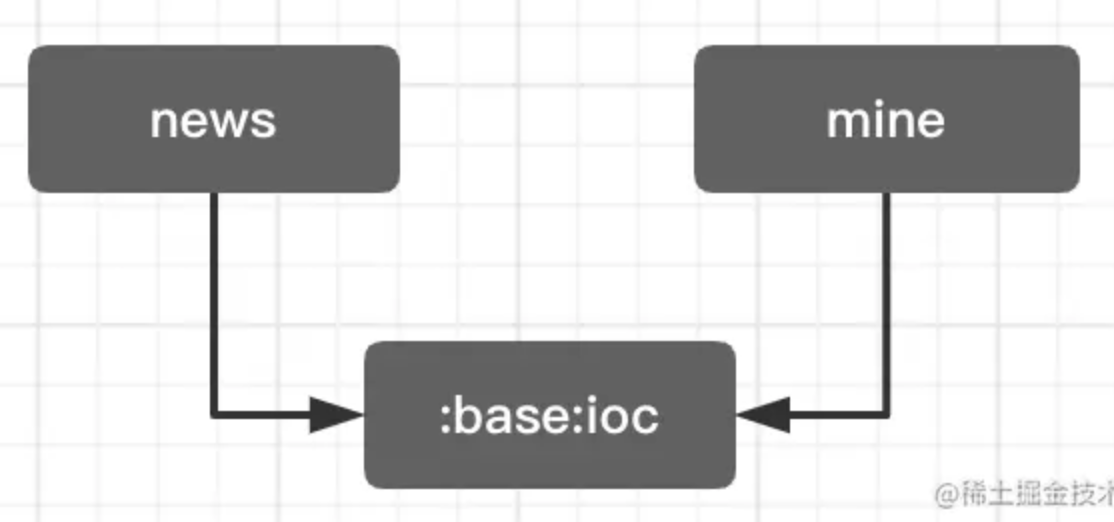
}

组件化依赖注入

如果在项目中使用到组件化的伙伴可能有遇到这样的问题，两个模块需要通信，通常采用的是模块依赖直接通信



这种方式其实是不可行的，因为不管是模块化还是组件化，这种方式会使得两个模块间耦合非常严重，两个模块应该相对独立，并向下继承，所以在下层需要有一个module专门负责依赖注入。



因为所有的业务模块会向下依赖，因此在:base:ioc库中会创建与业务相关的代理接口。

# :base:ioc module

interface ILoginDelegate {

fun openLoginActivity(context: Context, src: (Intent.() -> Unit)? = null)

}

既然有接口出现，那么就会有对应的实现类，该实现类是在登录模块中实现的。

# login module

class LoginDelegateImpl : ILoginDelegate{

override fun openLoginActivity(context: Context, src: (Intent.() -> Unit)?) {

val intent = Intent()

if (src != null){

intent.src()

}

intent.setClass(context,LoginActivity::class.java)

context.startActivity(intent)

}

}

所以登录模块需要向ioc模块注入这个实现类，其中比较简单的方式就是通过接口名与实现类名存储在一个Map中，当任意一个模块想要调用时，只需要拿到接口名就可以得到注入的实现类。

object InjectUtils {

/\*\*接口名与实现类名一一对应的map\*/

private val routerMap: MutableMap<String, String> by lazy {

mutableMapOf()

}

/\*\*接口名与实现类的一一对应\*/

private val implMap: MutableMap<String, WeakReference<\*>> by lazy {

mutableMapOf()

}

/\*\*注册\*/

fun inject(interfaceName: String, implName: String) {

if (routerMap.containsKey(interfaceName) || routerMap.containsValue(interfaceName)) {

return

}

routerMap[interfaceName] = implName

}

/\*\*获取实现类\*/

fun <T> getApiService(clazz: Class<T>): T? {

try {

val weakInstance = implMap[clazz.name]

if (weakInstance != null) {

val instance = weakInstance.get()

if (instance != null) {

return instance as T

}

}

/\*\*如果实例为空，需要新建一个实现类\*/

val implName = routerMap[clazz.name]

val instance = Class.forName(implName).newInstance()

implMap[clazz.name] = WeakReference(instance)

return instance as T

} catch (e: Exception) {

Log.i("InjectUtils", "error==>${e.message}")

return null

}

}

}

例如在news模块想要跳转到登录，首先需要全局注入

InjectUtils.inject(ILoginDelegate::class.java.name, LoginDelegateImpl::class.java.name)

然后在任何一个模块中都能够拿到这个实例。

InjectUtils.getApiService(ILoginDelegate::class.java)?.openLoginActivity(this)

其实想要实现这种注入方式有很多，像通过注解修饰这个实现类，****配合注解处理器全局扫描就可以少一部自己手动存储的这一步****，就是APT的思路；还有就是****Dagger2或者Hilt实现的隔离层架构****，同样也是一种方式。总之想要实现模块解耦，依赖注入是必须的。