**视图调试工具**

**背景**

在潍柴二次换肤的开发过程中发现，定位某个图片会很耗时，而且开发中遇到的一些非技术性问题，会占用开发过程不少的时间。

1. 对于一些很久没改动的模块或者不熟悉的模块，无法快速定位问题
2. 页面中有一个图片，想立即知道这个图片的名称
3. 需要修改某个页面布局，但页面镶嵌层级很深，无法快速找到要修改的位置
4. 在fragment镶嵌时无法快速找到要修改的fragment对象
5. 对于一些需要频繁修改并验证的代码，需要不停构建项目
6. 对布局进行微调需要多次构建项目
7. 针对无法断点调试的设备，需要插入日志，如果没有考虑全面，则需要多次插入日志并构建

针对以上问题，期望的解决方式：

1. 页面上能看到的元素，能很快知道这个元素的各种信息
2. 能快速找出某个view所在的layout文件（比如recyclerView的item布局），而且能直接复制layout的名称，用于在Android studio中搜索
3. 能快速找到View位于的Fragment（导航搜索模块很多同名Fragment，里面还镶嵌ViewPager）
4. 修改了代码后能立即生效，最好不重启应用就生效

所以针对以上问题，在潍柴二次换肤的期间，开发了一个小工具用以解决以上问题。

**可行性分析和原理**

**快速定位**

通过触摸事件的xy坐标，递归找到所有包含该点的view。

**获取View各种信息**

View的信息主要包括：

**View的图片相关名称（比如background、src、thumb、progressDrawable等与图片相关的属性）**

通过换肤框架能够拿到View的各种属性（换肤工具通过拦截布局xml的解析，能拿到xml中设置给View的所有属性）

View的一些xml属性记录在view.getViewUnion()中

对于一些在代码中动态设置的图片资源，如果需要能够捕获，则需要完整接入换肤框架。

**View所在布局名称**

View如何与layout进行关联

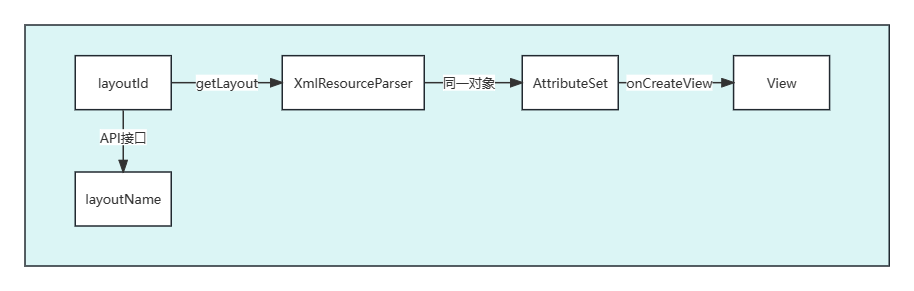
|  |
| --- |
| Java /LayoutInflater.java public View inflate(@LayoutRes int resource, @Nullable ViewGroup root, boolean attachToRoot) { //....  XmlResourceParser parser = res.getLayout(resource); //.... } |

|  |
| --- |
| Java /Resources.java @NonNull public XmlResourceParser getLayout(@LayoutRes int id) throws NotFoundException {  return loadXmlResourceParser(id, "layout"); } |

|  |
| --- |
| Java /LayoutInflater.Factory2.java @Nullable View onCreateView(@Nullable View parent, @NonNull String name,  @NonNull Context context, @NonNull AttributeSet attrs); |

|  |
| --- |
| Java /XmlResourceParser.java public interface XmlResourceParser extends XmlPullParser, AttributeSet, AutoCloseable {  String getAttributeNamespace (int index);  public void close(); } |

总结：



通过重写Resource.getLayout方法，能将layoutId和XmlResourceParser进行映射，通过重写onCreateView能将View和AttributeSet进行映射，而AttributeSet和XmlResourceParser是同一个对象，即View和layoutId能够进行映射。最终通过view的带layoutId，通过layoutId得到layoutName。

**View所在Fragment实例名称**

如何通过View获取view所在Fragment？

|  |
| --- |
| Java /ViewTreeLifecycleOwner.java @Nullable public static LifecycleOwner get(@NonNull View view) {  LifecycleOwner found = (LifecycleOwner) view.getTag(R.id.view\_tree\_lifecycle\_owner);  if (found != null) return found;  ViewParent parent = view.getParent();  while (found == null && parent instanceof View) {  final View parentView = (View) parent;  found = (LifecycleOwner) parentView.getTag(R.id.view\_tree\_lifecycle\_owner);  parent = parentView.getParent();  }  return found; } |

|  |
| --- |
| Java /Fragment.java void performCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container,  @Nullable Bundle savedInstanceState) {  //...  mViewLifecycleOwner = new FragmentViewLifecycleOwner(this, getViewModelStore());  mView = onCreateView(inflater, container, savedInstanceState);  if (mView != null) {  //....  ViewTreeLifecycleOwner.set(mView, mViewLifecycleOwner);   } else {  //....  } } |

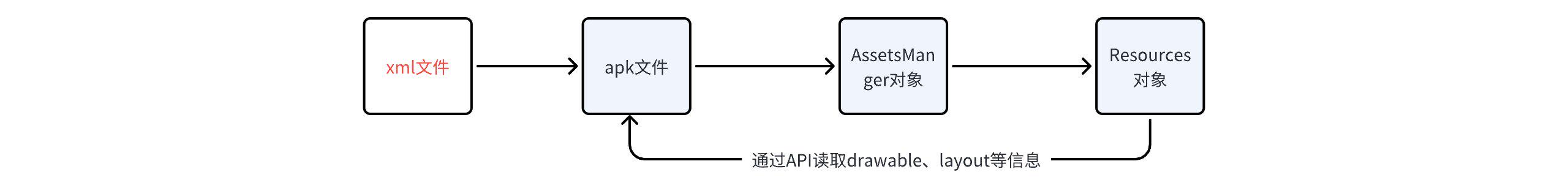
总结：

ViewTreeLifecycleOwner.get(view)方法能通过循环，向上拿到离View最近的一个LifecycleOwner 对象（FragmentViewLifecycleOwner），该对象在Fragment创建View时设置到Fragment的根View上。通过反射即可拿到FragmentViewLifecycleOwner里面的Fragment对象。

**代码修改即时生效**

**layout、drawable等xml文件的动态修改**

要动态加载xml的首要问题：使用什么API来加载外部 xml？需要将外部xml转换成何种对象方可使用？



参考换肤逻辑，通过构造皮肤包（apk），就可以加载外部apk的图片、drawable等信息。如果能将当前编辑的xml打入一个apk，那么就可以通过构造AssetsManger对象来构造Resource对象。

如何将xml打入apk？

apk就是一个压缩包，java中通过ZipOutputStream可以输出一个zip压缩文件。

另一个问题：xml文件应该放到apk哪个目录下？/drawable？/layout？/assets?

理论上都可以，通过Resources对象，均能读取这些文件夹下的文件，但是读取/drawable、/layout文件夹下的文件需要资源id，需要资源id才能获取。而/assets目录下直接通过文件名称就能获取，故放/assets目录。

|  |
| --- |
| Kotlin // 读取xml文件 val parser: XmlResourceParser = context.resources.assets.openXmlResourceParser("x.xml") // xml转drawable对象 val drawable = Drawable.createFromXml(context.resources, parser) |

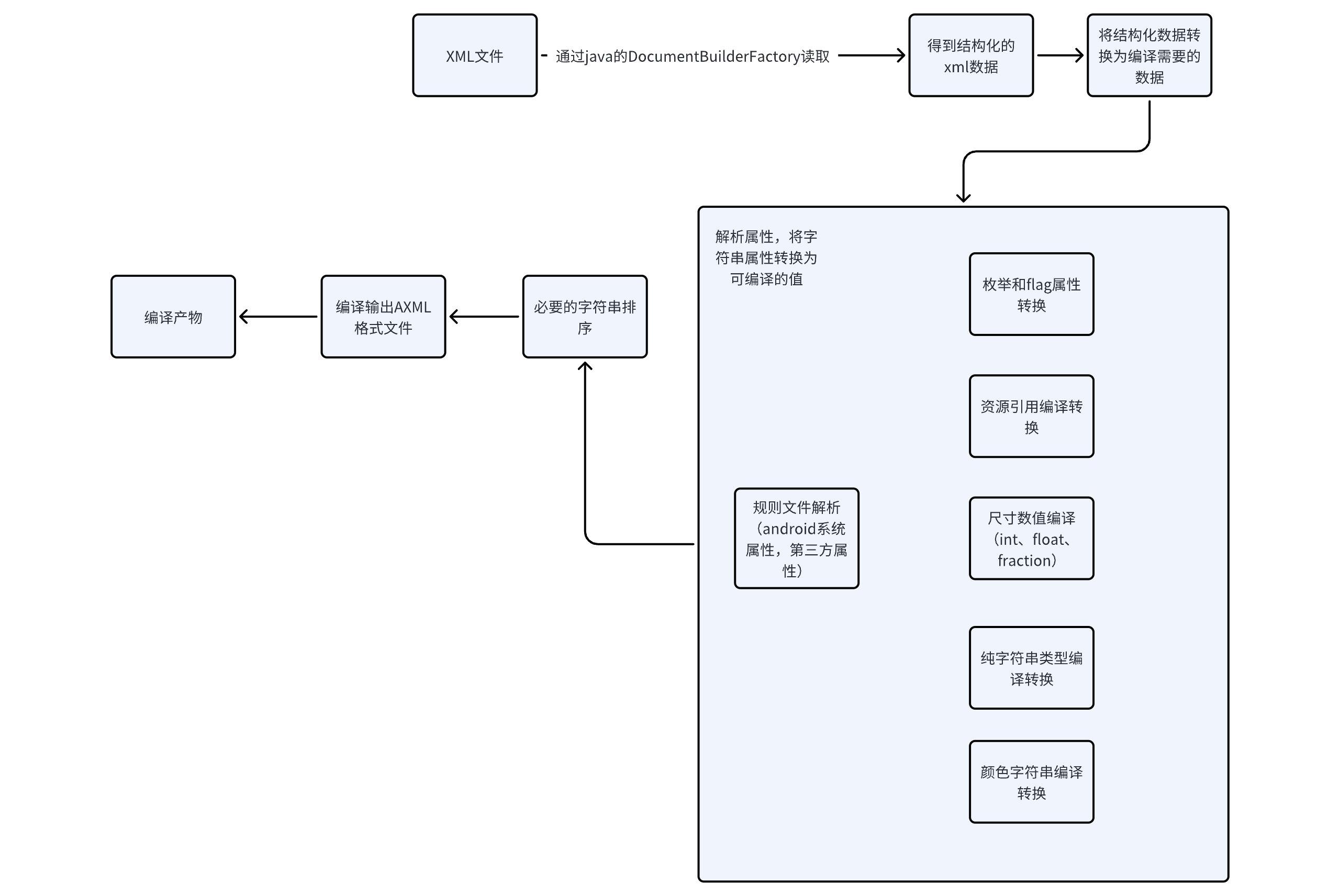
流程没问题了？

不，还有一个更复杂的问题，apk里面的xml文件和原始xml文件不一样，apk里面的xml文件是一种AXML格式的文件。总之，就是说原始xml不能直接打入apk，需要将原始xml编译为AXML格式。

文件对比

axml读取：https://www.jianshu.com/p/5c31fc5b70f4

写入流程：



其中编译需要注意和解决的问题：

编译后的文件是小端存储；

字符串常量写完需进行4字节对齐；

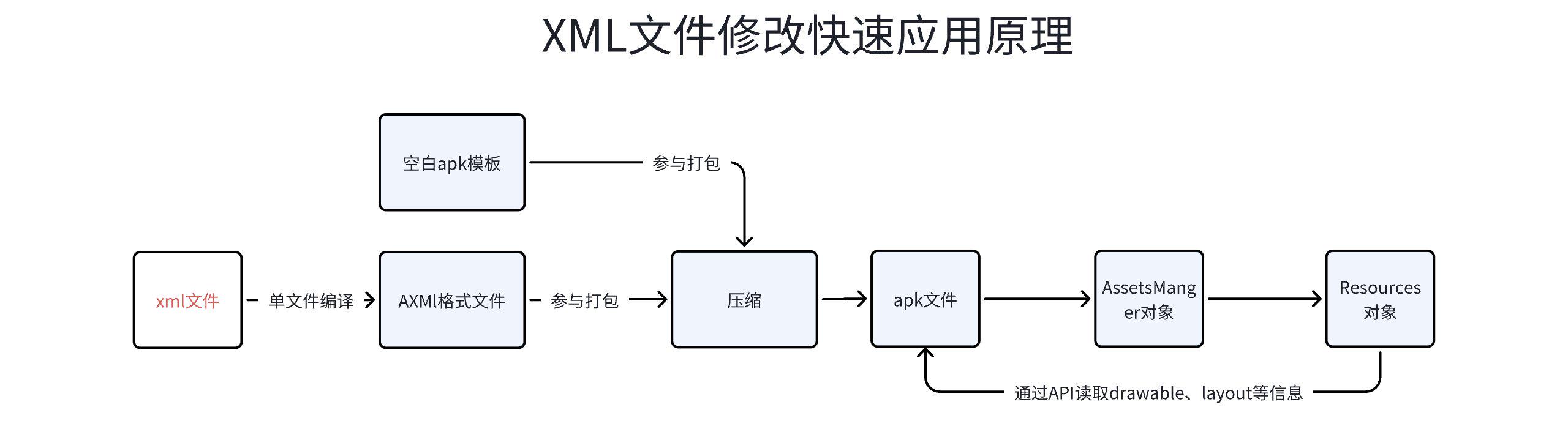
属性字符串排序方式是按照属性id进行排序；

字符串长度小于128则存储一个字节，大于则进行扩充，注意扩充规则；

尺寸(dimensions)编译需了解转换原理，如何用一个int值存储10px、10dp、10sp、10.5dp等不同单位的整形和浮点数据；

由于是在应用内编译，还需考虑Android版本问题，即遇到当前运行环境无法识别的属性如何处理；

总结：



**kotlin类（文件）的热修复**

很常见的的一种技术，冷启动修复

1. 类何时被加载？

初次new一个类时会加载；

静态类初次引用时会加载；

Class.forName初次执行时会加载对应类

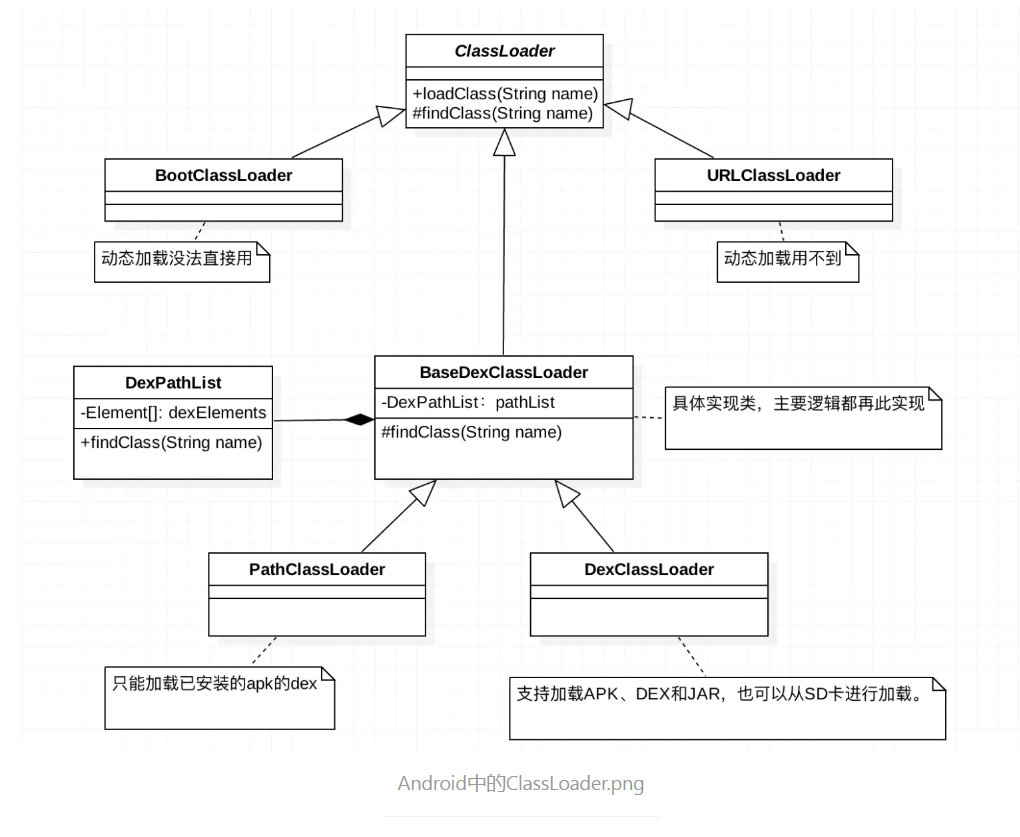
2. ClassLoader的双亲委托机制

|  |
| --- |
| Kotlin /ClassLoader.java protected Class<?> loadClass(String name, boolean resolve)  throws ClassNotFoundException {  // First, check if the class has already been loaded  Class<?> c = findLoadedClass(name);  if (c == null) {  try {  if (parent != null) {  c = parent.loadClass(name, false);  } else {  c = findBootstrapClassOrNull(name);  }  } catch (ClassNotFoundException e) {  // ClassNotFoundException thrown if class not found  // from the non-null parent class loader  }   if (c == null) {  // If still not found, then invoke findClass in order  // to find the class.  c = findClass(name);  }  }  return c; } |

加载一个类时总是先从缓存加载，没有则让父加载器加载，还是没有则让BootstrapClassLoader加载（比如java库中的类），最终还是没有则自己加载。如果自己也无法加载，则抛出ClassNotFoundException 的异常。

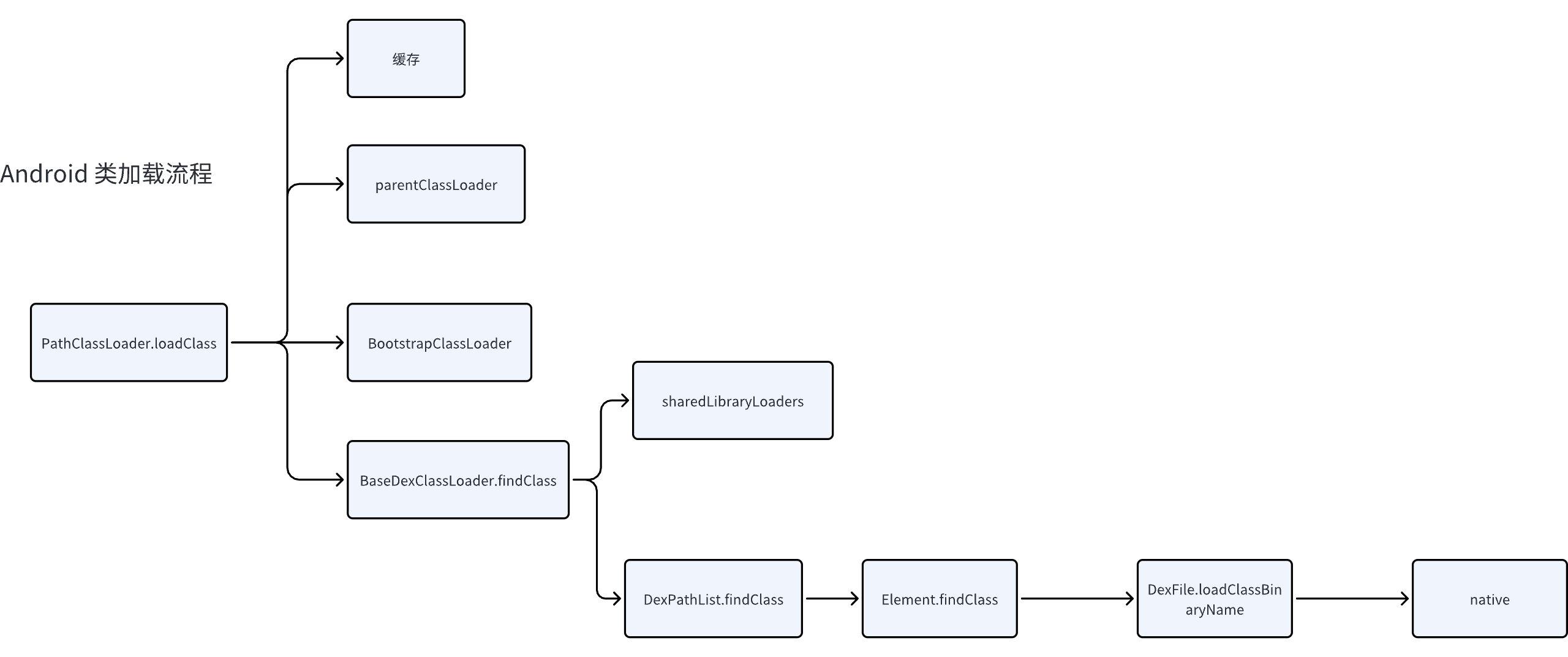
总结：

一旦一个类被加载过，则无法重新加载，要修复一个类，需要在类被加载之前执行替换。



Android应用的默认类加载器是dalvike.system.PathClassLoader。

Android中类的加载流程



热修复就是在findClass时，优先从外部提供的dex中读取类信息

如何将当前正在编辑的代码转换成可加载的dex？

java/ktolin ----> class文件 ----> dex

在这个流程中，java转class可以用javac、kotlin转class可以用kotlinc，class转dex可以用dx工具。

新的问题又产生了，通过javac或者kotlinc的方式编译代码会出现一个问题，即需要手动指定classpath，一个Android项目中一般都有好几十个依赖，需要在执行javac或者kotlinc时一一指定，很麻烦。

而且，如果java调用kotlin代码，那么仅仅javac是无法编译成功的，需要进行java、kotlin混合编译，十分麻烦。

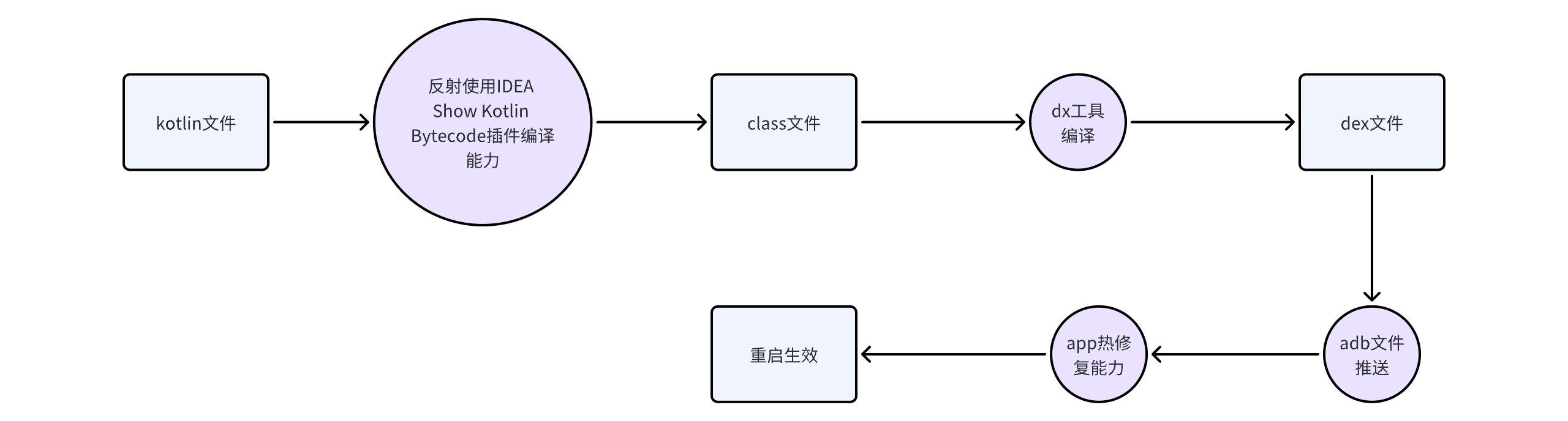
针对kotlin，Android studio自带的Kotlin插件支持即时查看字节码信息：Tools->Kotlin->Show Kotlin Bytecode

那么，就代表着这个插件可以实时将kotlin转换成字节码，但其显示的是字符串，而需要的是字节码。

通过查看IDEA Kotlin插件源码，最终能够通过反射其插件，调用插件方法从而得到对应kotlin的字节码([Kotlin Bytecode](https://github.com/search?q=repo%3AJetBrains%2Fintellij-community+kotlin+Bytecode&type=code))。

由于过程有点复杂，就不再描述。

整体流程



总结

整个过程分为两部分，Android studio插件负责将当前文件发送到app（adb push），如果发送的是kotlin文件，则先编译成dex文件再发送。app监听收到文件后，负责将发送的xml文件编译成axml格式然后打包成apk，最后加载apk动态替换原有xml。如果接收的是dex文件，则在下次启动时加载dex文件。

**其它**

目前存在的缺陷：

1. 输出地址需要手动指定
2. 需手动设置dx工具路径
3. 需手动设置R文件路径
4. 如果是在代码中设置的图片信息则无法抓取，需要适配换肤框架才能抓取。
5. 无法快速编译java代码
6. 针对某些需要gradle在编译期间生成的代码无法生成，比如某些编译时生效的注解
7. 目前还不支持多个kotlin文件的编译
8. kotlin编译生效只能是未混淆的类
9. dialog需要指定style

工具源码：

https://github.com/DFFXT/SkinSwitch

https://github.com/DFFXT/ViewDebug-Trans

**使用方式**

前提：Android studio安装特定插件，否则无法发送文件到app，无法动态编译kotlin。（[视图调试工具--Android studio插件](https://mxnavi-com.feishu.cn/docx/T1eKdVhTjoFR09xNBKxcsuQundd) ）

|  |
| --- |
| Kotlin // 添加jitpack仓库；不过一般项目都是添加了的 maven { url 'https://jitpack.io' } |

|  |
| --- |
| Groovy // 解决依赖冲突，如果有冲突的话 debugImplementation 'com.google.guava:listenablefuture:9999.0-empty-to-avoid-conflict-with-guava' // View创建拦截库，具有换肤能力 debugImplementation 'com.github.DFFXT.SkinSwitch:SkinCore:0.19' // 调试工具核心库 debugImplementation 'com.github.DFFXT.SkinSwitch:ViewDebug:0.19' |

由于该工具依赖换肤框架，所以需要初始化换肤框架。

|  |
| --- |
| Kotlin SkinManager.init(context as Application, 0, DefaultProviderFactory()) |

由于上面采用debugImplementation 方式依赖，所以这部分代码需要放在在src/debug/java目录（没有自己创建），通过startUp插件初始化，否则编译release版本会找不到类

|  |
| --- |
| Kotlin <?xml version="1.0" encoding="utf-8"?> <manifest xmlns:android="http://schemas.android.com/apk/res/android"  xmlns:tools="http://schemas.android.com/tools">   <application>  <provider  android:name="androidx.startup.InitializationProvider"  android:authorities="ViewDebugInitializer.androidx-startup"  android:exported="false"  tools:node="merge">  <meta-data  android:name="DebugInitializer"  android:value="androidx.startup" />  </provider>  </application> </manifest> |

|  |
| --- |
| Kotlin // 使用startup 的方式来初始化 class DebugInitializer: Initializer<DebugInitializer> {  override fun create(context: Context): ViewDebugProvider {  // 初始化  SkinManager.init(context as Application, 0, DefaultProviderFactory())  return this  }   override fun dependencies(): MutableList<Class<out Initializer<\*>>> {  // 添加依赖，依赖内部ViewDebugInitializer对象  return mutableListOf(com.example.viewdebug.ViewDebugInitializer::class.java)  }  } |

调试工具需要悬浮窗权限，需要手动开启