> 最近几年，腾讯对于开源事业也是越来越支持，今天要说的就是在腾讯被广泛使用的Shadow框架，一个经过线上亿级用户量检验的反射全动态Android插件框架。 首先，让我们来看一下官方对于Shadow的简介：

Shadow是一个腾讯自主研发的Android插件框架，经过线上亿级用户量检验。 Shadow不仅开源分享了插件技术的关键代码，还完整的分享了上线部署所需要的所有设计。

与市面上其他插件框架相比，Shadow主要具有以下特点：

- **复用独立安装App的源码**：插件App的源码原本就是可以正常安装运行的。
- **零反射无Hack实现插件技术**：从理论上就已经确定无需对任何系统做兼容开发，更无任何隐藏API调用，和Google限制非公开SDK接口访问的策略完全不冲突。
- **全动态插件框架**：一次性实现完美的插件框架很难，但Shadow将这些实现全部动态化起来，使插件框架的代码成为了插件的一部分。插件的迭代不再受宿主打包了旧版本插件框架所限制。
- **宿主增量极小**：得益于全动态实现，真正合入宿主程序的代码量极小（15KB，160方法数左右）。
- **Kotlin支持**：core.loader，core.transform核心代码完全用Kotlin实现，代码简洁易维护。

除此之外， Shadow支持的特性有：

- 四大组件
- Fragment（代码添加和Xml添加）
- DataBinding（无需特别支持，但已验证可正常工作）
- 跨进程使用插件Service
- 自定义Theme
- 插件访问宿主类
- So加载
- 分段加载插件（多Apk分别加载或多Apk以此依赖加载）
- 一个Activity中加载多个Apk中的View
- ……

Shadow解决的问题

非公开SDK接口访问

众所周知，Android 9.0出现限制非公开SDK接口访问之后，可以说当时我们已知的所有插件框架实现都或多或少的出现了适配问题。大家的应对方式基本上都是一种对抗的思想，有的去破解限制，有的通过和Google沟通浅灰名单有效期暂时续命。

遇到了这个问题，我们没有选择和这个策略进行对抗，我们非常理解Google为什么限制使用非公开SDK接口。所以我们重新Review了插件框架的本质原理和设计缺陷，进而设计了全新的插件框架Shadow。Shadow没有使用任何非公开SDK接口，实现了和原本在用的使用了大量非公开SDK接口的实现一样的功能。

在Shadow的Sample中，可以添加如下所示的代码来开启严格检查模式运行，而其他插件框架并不能做到。

```
StrictMode.VmPolicy.Builder builder = new StrictMode.VmPolicy.Builder();
builder.penaltyDeath();
builder.detectNonSdkApiUsage();
StrictMode.setVmPolicy(builder.build());
```

比如，我们看到的一款也宣传未使用非公开SDK接口，支持Android 9.0的插件框架，在它的Sample中开启严格模式运行后，出现了如下Crash信息：

```
W/.xxx.sampl: Accessing hidden method Landroid/view/View;-
>computeFitSystemWindows(Landroid/graphics/Rect;Landroid/graphics/Rect;)Z (light
greylist, reflection)
W/System.err: StrictMode VmPolicy violation with POLICY_DEATH; shutting down.
```

可见，即使它的实现代码中没有出现任何非公开SDK的引用，实际上它依赖的第三方组件内部也使用了非公开SDK接口。

插件框架全动态化

所谓全动态，指的就是除了插件代码之外，插件框架本身的所有逻辑代码也都是动态的。并且，Shadow框架实际上也做到了这一点，即插件框架的代码我们是和插件打包在一起发布的。

全动态化插件框架有多重要呢？其实它比无Hack、零反射实现还要重要！因为有了这个特性之后，就算是我们用了Hack的方案，需要兼容各种手机厂商的系统。我们也不需要等宿主App更新才能解决问题。

实际上，Shadow的这个特性是更早实现的。我们早在2015年就开始大量使用插件技术了。在2016年就实现了这个特性。凭借这个特性不断的动态发布插件框架的代码，去适配各种兼容性问题。在今年更是应用这个特性，**在完全不跟宿主版本的前提下，将原本的具有上百个反射Hack调用的旧实现更新为了Shadow无Hack实现。**新的Shadow自然也具备这个特性。

在实际使用过程中，我们的宿主对于业务接入在增量上有极其苛刻的要求。Shadow接入时只使用了15.1KB，160个方法。而我们已知的其他插件框架对宿主的增量一般在110KB，900个方法到370KB，2300个方法之间。

实现原理

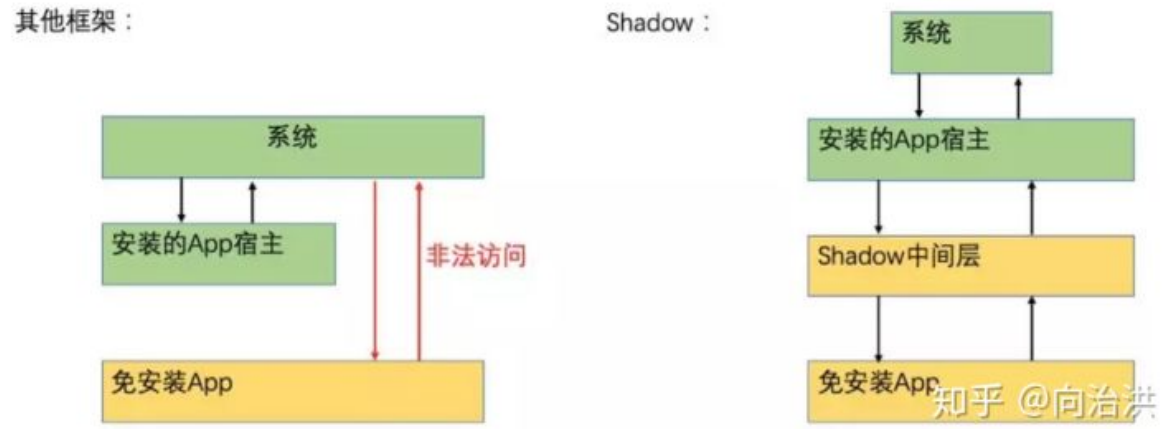
以前的插件框架总是想用一些Hack手段去修改系统行为，找到系统的漏洞达到目的。Shadow的原则是不去跟系统对抗。既然只是限制非公开SDK接口访问，而没有限制动态加载代码。那么肯定有办法在不使用非公开SDK接口的前提下实现原来的目的。因为我们插件技术的目的本质上来说还是动态加载代码。

那么一个重要的原则就是，如果一个组件需要安装才能使用，那么就别在没安装的情况下把它交给系统。我们已知的插件框架中，做的最好的也不符合这个原则，所以尽管它的Hook点少，但就是由于它将没有安装的Activity交给系统了，所以后面就不得不做一些Hack的事修补。

所以套一个壳子的方案就非常好。这种思路其他框架很早就有了，但是它们一直想把一个插件Activity套在一个宿主Activity之中，然后想办法实现一个转调关系。如果插件Activity是一个真的Activity，那这个插件就可以正常编译安装运行，对开发插件或者直接上架插件App非常有利。但是由于它是个系统的Activity子类，它就有很多方法不能直接调用，甚至还可能需要避免它的super方法被调用。如果插件Activity不是一个真的Activity，只是一个跟Activity有差不多方法的普通类，这件事就简单多了，只需要让壳子Activity持有它，转调它就行了。但这种插件的代码正常编译成独立App安装运行会比较麻烦，代码中可能会出现很多插件相关的if-else，也不好。

Shadow做了一个非常简单事，通过运用AOP思想，利用字节码编辑工具，在编译期把插件中的所有Activity的父类都改成一个普通类，然后让壳子持有这个普通类型的父类去转调它就不用Hack任何系统实现了。虽然说是非常简单的事，实际上这样修改后还带来一些额外的问题需要解决，比如getActivity()方法返回的也不是Activity了。不过Shadow的实现中都解决了这些问题。

Shadow框架的原理示意图如下：



集成Shadow

环境准备

第一次clone Shadow的代码到本地后，建议先在命令行编译一次。

- 在编译前，必须设置ANDROID_HOME环境变量。
- 在编译时，必须使用gradlew脚本，以保证采用了项目配置的Gradle版本。

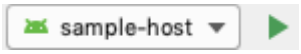
在命令行测试编译时可以执行如下编译任务：

```
./gradlew build
```

如果没有出错，再尝试用Android Studio打开工程。

1. 必须使用3.4或更高版本的Android Studio打开工程。
2. 必须关闭Android Studio的Instant Run功能。

然后就可以在IDE中选择sample-host模块直接运行即可，如下：



Shadow的所有代码都位于projects目录下的3个目录：