# 1 Hook类型

## 1.1 Hook概念

Hook技术也称“钩子技术”，其原理是先将需要修改的函数“勾住”，然后使用自定义的函数替换原函数，让程序在运行时执行自定义函数，从而达到动态修改软件的目的。

## 1.2 Hook类型

Hook分为Java层Hook与Native层Hook。

Java层： Dalvik Hook、ART Hook。

Native层：给予动态库加载劫持的LD\_PRELOAD Hook、针对.got.plt节区的GOT Hook、针对汇编指令级别的Inline Hook。

### 1.2.1 Dalvik Hook

Dalvik Hook能够实现，得益于以下三个方面：

（1）Java的反射机制：

对任何一个java类的方法进行Hook，其本质就是对Method类进行Hook。通过反射机制，DEX文件中的代码很容易就能访问自己内存空间的任何类方法信息。

（2）Dalvik虚拟机底层的java方法实现：

在Dalvik虚拟机中，Method结构体中的字段完整地描述了一个java方法的信息，可以将它们的值修改为目标方法的相应字段，实现“移花接木”的效果。

（3）可以修改自身进程空间的内存。

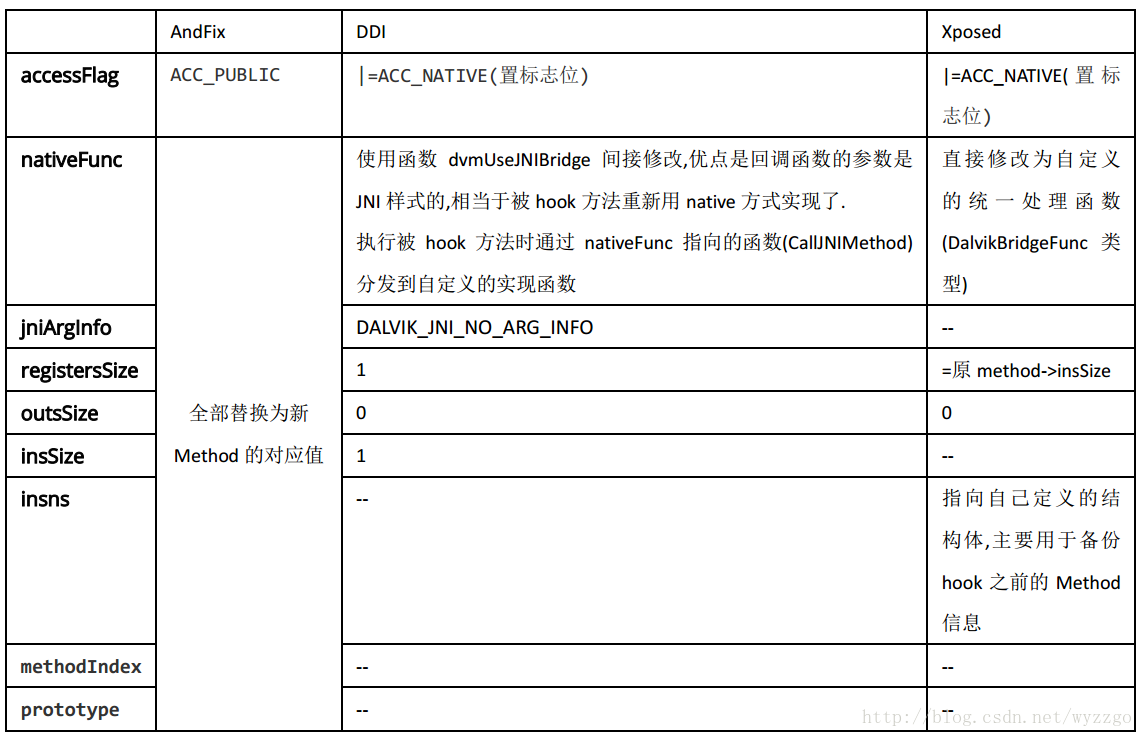
进程对自身内存空间的数据有绝对的控制权。

常见的两种hook思路:

（1）将被hook方法Method结构体的accessFlags置native标志位,并将nativeFunc指向自定义的函数.可以将多个被hook的方法都转向统一的自定义函数,在此函数中进行方法的分发处理.特点是比较灵活,实现起来有一定难度.

（2）声明一个与被hook方法完全相同的新方法,将新方法的Method结构体覆盖到被hook的Method结构体上. 特点是相对简单,但是不够灵活.

几种常见的Dalvik Hook框架实现细节：



其中–表示未修改,ACC\_NATIVE,ACC\_PUBLIC定义于/dalvik/libdex/DexFile.h, DALVIK\_JNI\_NO\_ARG\_INFO定义于/dalvik/vm/JniInternal.h.

值得注意的是DDI框架将registersSize和insSize全部修改为1,是因为其代码只hook了部分类的toString函数,并没有通用性.另外将jniArgInfo修改为DALVIK\_JNI\_NO\_ARG\_INFO在部分CPU架构上会出现问题(x86强制要求jniArgInfo为有效值,否则调用方法时直接崩溃).

几种常见的Dalvik Hook框架比较：



参考链接：

<https://blog.csdn.net/wyzzgo/article/details/53647994>

<https://blog.csdn.net/wyzzgo/article/details/53706685>

### 1.2.2 ART Hook

参考链接：

<https://www.cnblogs.com/twlqx/p/4449451.html>

<https://www.cnblogs.com/lanrenxinxin/p/5207174.html>

<https://www.jianshu.com/p/7dcb32f8a0ce>

### 1.2.3 LD\_PRELOAD Hook

LD\_PRELOAD是Linux系统中的一个环境变量。LD\_PRELOAD Hook是针对此环境变量的特殊的Hook技术。

LD\_PRELOAD可以影响程序运行时的动态链接，只需通过对该环境变量预先指定想要加载的动态链接库文件列表，在程序运行时，系统就会优先加载这个动态链接库列表中的动态库，并通过它的函数信息来设置程序的.plt节区中的函数。

### 1.2.4 GOT Hook

### 1.2.5 Inline Hook

# 2 Hook框架Xposed

## 2.1 Xposed概述

Xposed是GitHUB上rovo89设计的一个针对Android平台的动态劫持项目，**通过替换/system/bin/app\_process程序控制zygote进程**，**使得app\_process在启动过程中会加载XposedBridge.jar这个jar包，从而完成对Zygote进程及其创建的Dalvik虚拟机的劫持**。与采取传统的Inhook方式（详见Dynamic Dalvik Instrumentation分析这篇本章 ）相比，Xposed在开机的时候完成对所有的Hook Function劫持，在原Function执行的前后加上自定义代码。

Xposed共有五个部分：

（1）Xposed：

Xposed的C++部分，主要是用来替换/system/bin/app\_process，并为 XposedBridge提供JNI方法;

（2）XposedBridge：

Xposed的Java部分，以XposedBridge.jar文件存在，负责在Native 层与FrameWork层进行交互。/system/bin/app\_process进程启动过程中会加载该jar包，其它的Modules的开发与运行都是基于这个jar；

（3）XposedInstaller：

Xposed的安装包，负责配置Xposed工作的环境并且提供对基于Xposed框架的Modules的管理。在安装XposedInstaller之后，app\_process与XposedBridge.jar放置在了/data/data/de.robv.android.xposed.installer；

（4）XposedMods：

用Xposed Framework实现的实现的demo；

（5）XposedAppSettings:

用Xposed开发的一些Modules，其中AppSettings是一个可以进行权限动态管理的应用;

## 2.2 Xposed原理

**Zygote：**

在Android系统中，应用程序进程都是由Zygote进程孵化出来的，而Zygote进程是由Init进程启动的。Zygote进程在启动时会创建一个Dalvik虚拟机实例，每当它孵化一个新的应用程序进程时，都会将这个Dalvik虚拟机实例复制到新的应用程序进程里面去，从而使得每一个应用程序进程都有一个独立的Dalvik虚拟机实例。这也是Xposed选择替换app\_process的原因。

Zygote进程在启动的过程中，除了会创建一个Dalvik虚拟机实例之外，还会将Java运行时库加载到进程中来，以及注册一些Android核心类的JNI方法来前面创建的Dalvik虚拟机实例中去。注意，一个应用程序进程被Zygote进程孵化出来的时候，不仅会获得Zygote进程中的Dalvik虚拟机实例拷贝，还会与Zygote一起共享Java运行时库。这也就是可以将XposedBridge这个jar包加载到每一个Android应用程序中的原因。XposedBridge有一个私有的Native（JNI）方法hookMethodNative，这个方法也在app\_process中使用。这个函数提供一个方法对象利用Java的Reflection机制来对内置方法覆写。具体的实现可以看下文的Xposed源代码分析。

**Hook：**

Xposed 框架中真正起作用的是对方法的hook。在Repackage技术中，如果要对APK做修改，则需要修改Smali代码中的指令。而另一种动态修改指令的技术需要在程序运行时基于匹配搜索来替换smali代码，但因为方法声明的多样性与复杂性，这种方法也比较复杂。

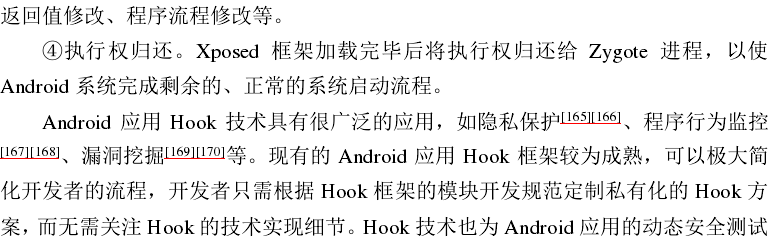
在Android系统启动的时候，zygote进程加载XposedBridge将所有需要替换的Method通过JNI方法hookMethodNative指向Native方法xposedCallHandler，xposedCallHandler在转入handleHookedMethod这个Java方法执行用户规定的Hook Func。

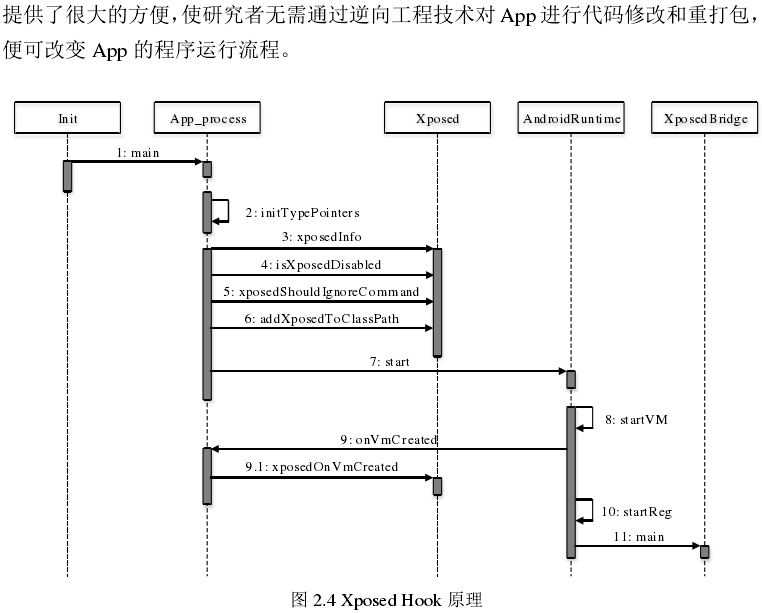
XposedBridge这个jar包含有一个私有的本地方法：hookMethodNative，该方法在附加的app\_process程序中也得到了实现。它将一个方法对象作为输入参数（你可以使用Java的反射机制来获取这个方法）并且改变Dalvik虚拟机中对于该方法的定义。它将该方法的类型改变为native并且将这个方法的实现链接到它的本地的通用类的方法。换言之，当调用那个被hook的方法时候，通用的类方法会被调用而不会对调用者有任何的影响。在hookMethodNative的实现中，会调用XposedBridge中的handleHookedMethod这个方法来传递参数。handleHookedMethod这个方法类似于一个统一调度的Dispatch例程，其对应的底层的C++函数是xposedCallHandler。而handleHookedMethod实现里面会根据一个全局结构hookedMethodCallbacks来选择相应的hook函数，并调用他们的before, after函数。

当多模块同时Hook一个方法的时候，Xposed会自动根据Module的优先级来排序，调用顺序如下：

A.before -> B.before -> original method -> B.after -> A.after







参考链接：

<https://blog.csdn.net/wxyyxc1992/article/details/17320911>

## 2.3 Hook场景与应用

1.开启日志调试输出

可以使用Xposed来hook xxLog 类的初始化代码，修改条件变量的值以使其满足调试输出。更简单的方法是使用XC\_MethodReplacement直接替换日志输出方法的内容。

2.APK插件

3.绕过SSL Pinning（安全回话绑定或证书绑定）

APK允许使用自签名的CA来构建HTTPS通信网络。在所有的通信过程中，对SSL通信证书合法性的验证由KeySore提供的API来完成，它们是Android SDK API的一部分，使用java方法实现。这也意味着，所有的通信过程都可以被Xposed编写的插件接管。

## 2.4 动态注入

动态注入与Hook区别：进行Hook后，会修改原方法或原函数的指针，让其转去执行Hook的方法，而动态注入则通过进程读写技术将一段代码或程序写到目标程序的内存空间中，然后修改目标程序的指令指针，让加载的代码得以执行。

1.so动态注入

So动态注入指将so动态库注入目标apk，然后执行库中的某个函数的技术。分以下四步实现：

①编写Native代码，定义so加载后需要执行的函数，将其编译成so文件。此阶段使用AS配合Android NDK实现。

②编写注入程序，实现远程进程注入。注入程序用Native代码编写，并需要使用practe系统的进程调试接口函数。

③将so文件注入目标进程。

④运行so文件指定函数。

2.DEX注入

Dex注入也需要注入so文件，只是so文件中指定的方法是用来加载dex文件的，步骤如下：

①编写java代码，编译成dex或apk文件，

②编写Native代码，定义so加载后执行的函数（必须包含dex代码），编译成so文件。需要用到JNI技术。

③编写注入程序，实现远程进程注入。这一步与so动态库注入一样。

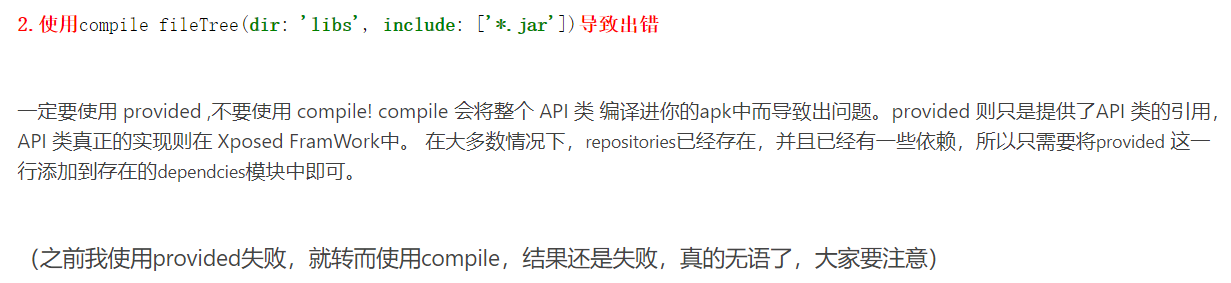
④将so文件注入目标程序。

⑤运行so文件中指定的函数。

## 2.5 Xposed应用实例

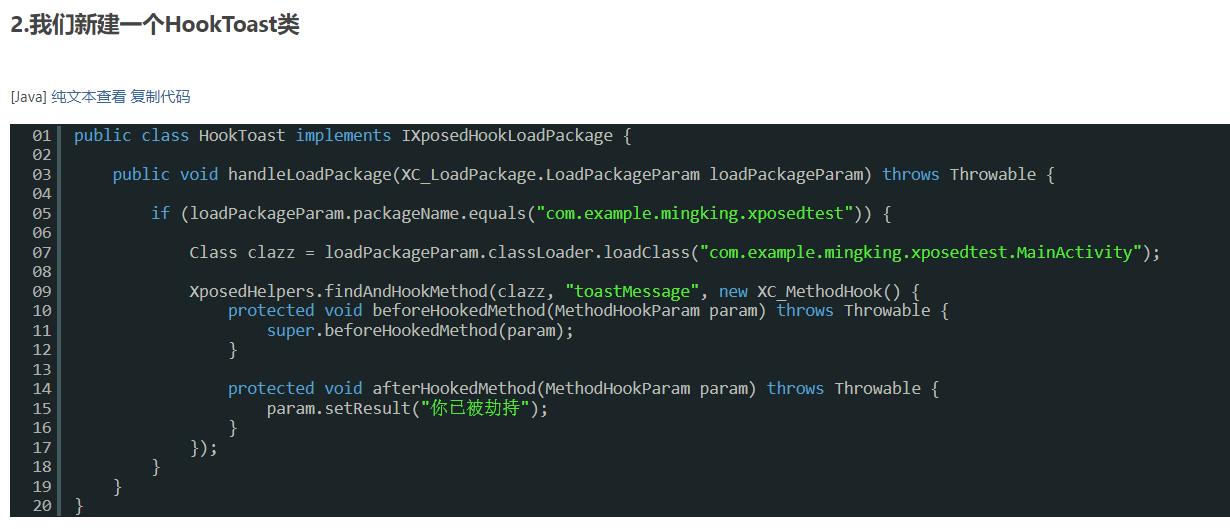
<https://www.52pojie.cn/thread-688466-1-1.html>

注意事项：

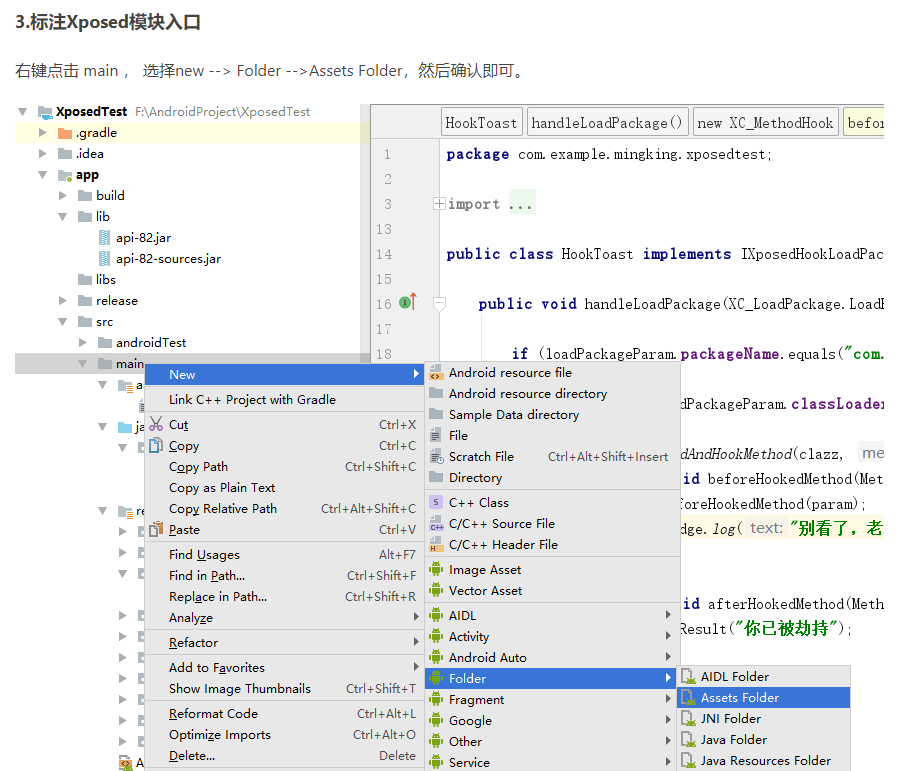


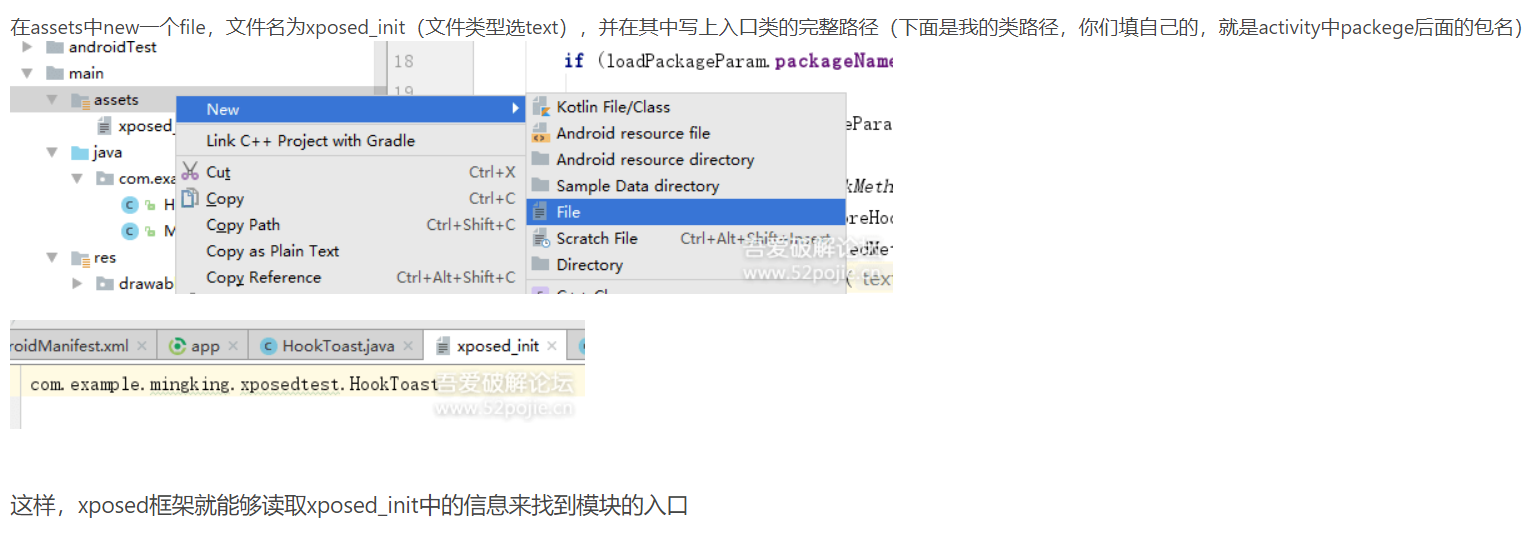


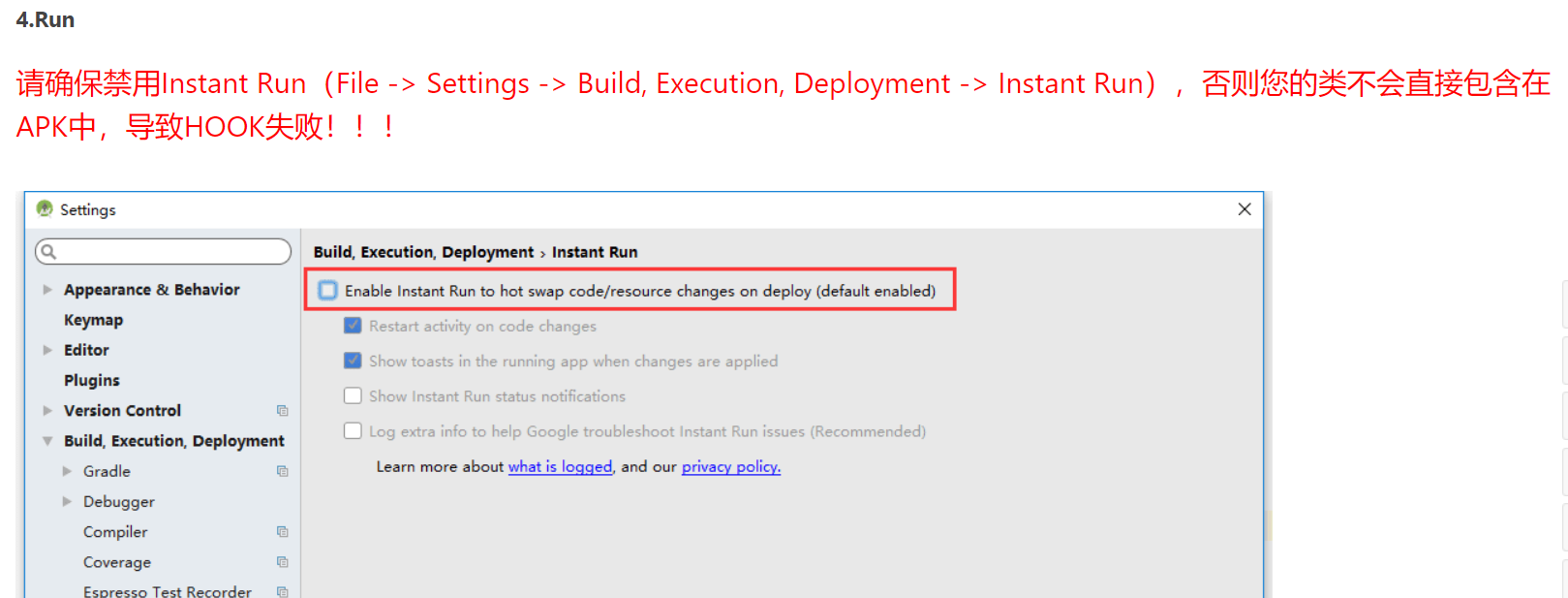












# 3 注入框架Frida

Frida 是一款基于 Python + JavaScript 的 Hook 与调试框架。

Firda 是一款易用的跨平 Hook 工具， Java 层到 Native 层的 Hook 无所不能，是一种 动态 的插桩工具，可以插入代码到原生 App 的内存空间中，动态的去监视和修改行为，原生平台包括 Win、Mac、Linux、Android、iOS 全平台。

（1）静态二进制插桩：在程序执行前插入额外的代码和数据，生成一个永久改变的可执行文件。

（2）动态二进制插桩：在程序运行时实时地插入额外代码和数据，对可执行文件没有任何永久改变。

大部分 App 对于我们来说都是黑盒，对 App 进行逆向和动态调试、或自动化分析、需要不断的进行动态调试，Frida 通过使用 Python 注入 JavaScript 脚本，都是通过 JS 脚本来操作设备上的 Java代码。

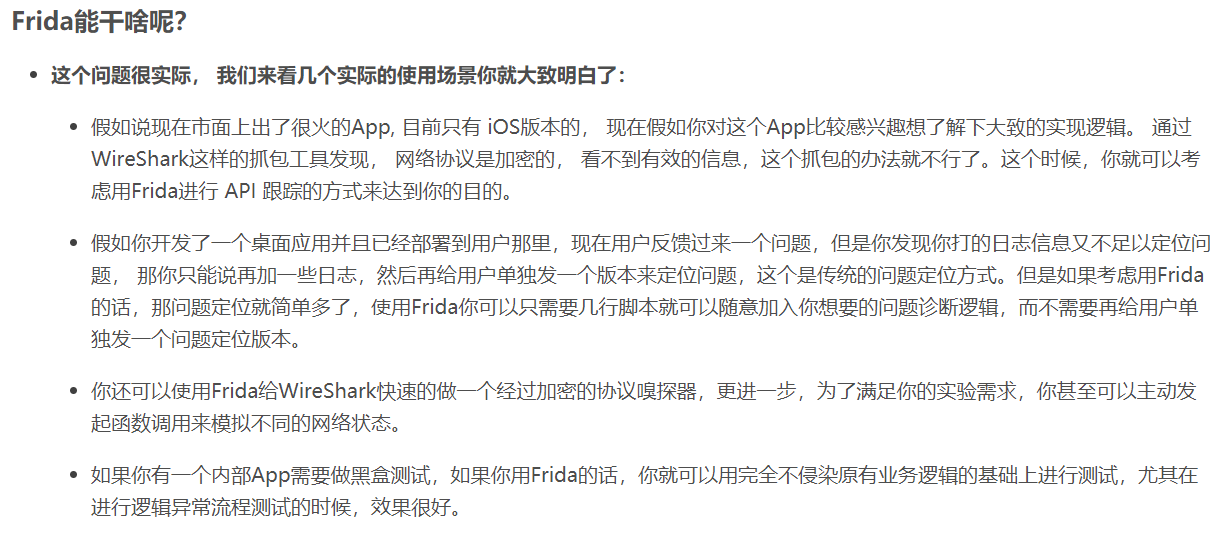
如果需要持久化的 Hook 还是需要通过 Xposed 等框架，但是 Frida 的动态和灵活性对逆向和自动化逆向提供了很大帮助。

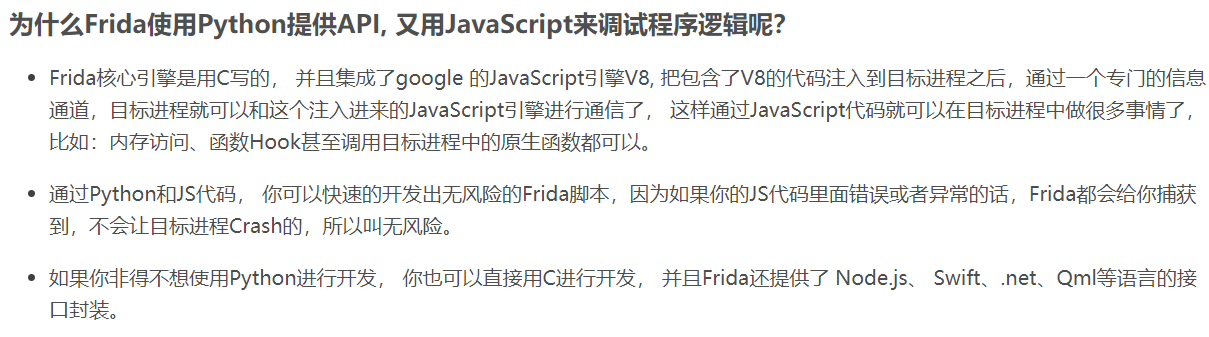
对于逆向操作破解来说方便，安装环境和配置环境要求简单兼容性好。

选择动态调试函数会遇到各种反调试、崩溃，相比xposed而言，frida优势是其动态执行不需要重启。

因为Frida大致原理是手机端安装一个server程序，然后把手机端的端口转到PC端，PC端写python脚本进行通信，而python脚本中需要hook的代码采用javascript语言。

**缺点：**因为Frida只适用于破解者在开发阶段，也就是它没法像Xposed用于实践生产中，比如我写一个微信外挂用Frida写肯定不行的，因为他无法在手机端运行。也就是破解者用的比较多。



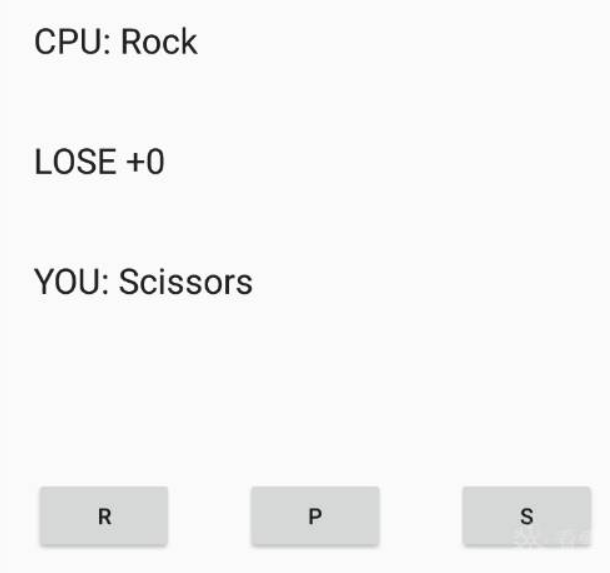


**参考链接：**

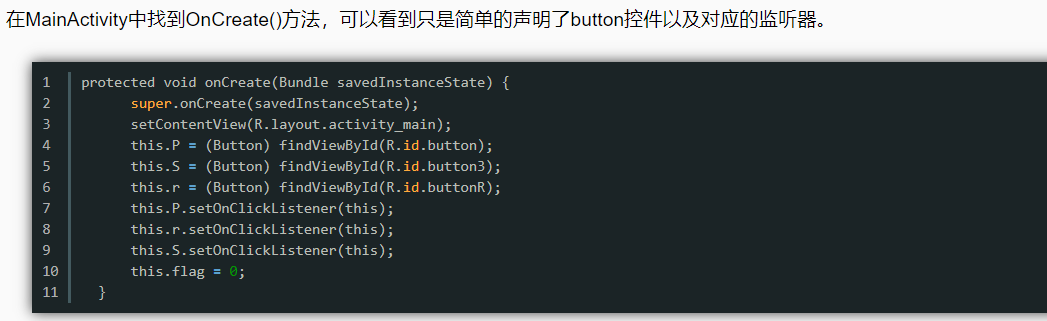
<https://blog.csdn.net/freakishfox/article/details/78278452>

<https://blog.csdn.net/qq_1290259791/article/details/100381831>

## 3.1 Frida应用实例（rps.apk）

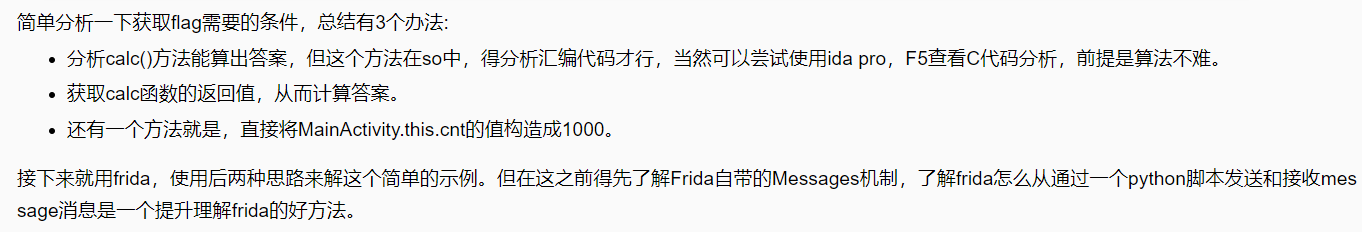


源代码分析：







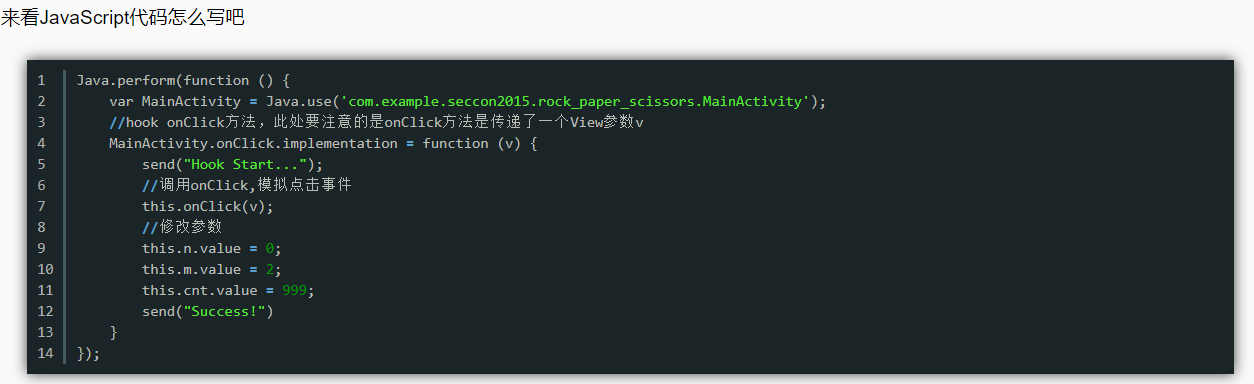










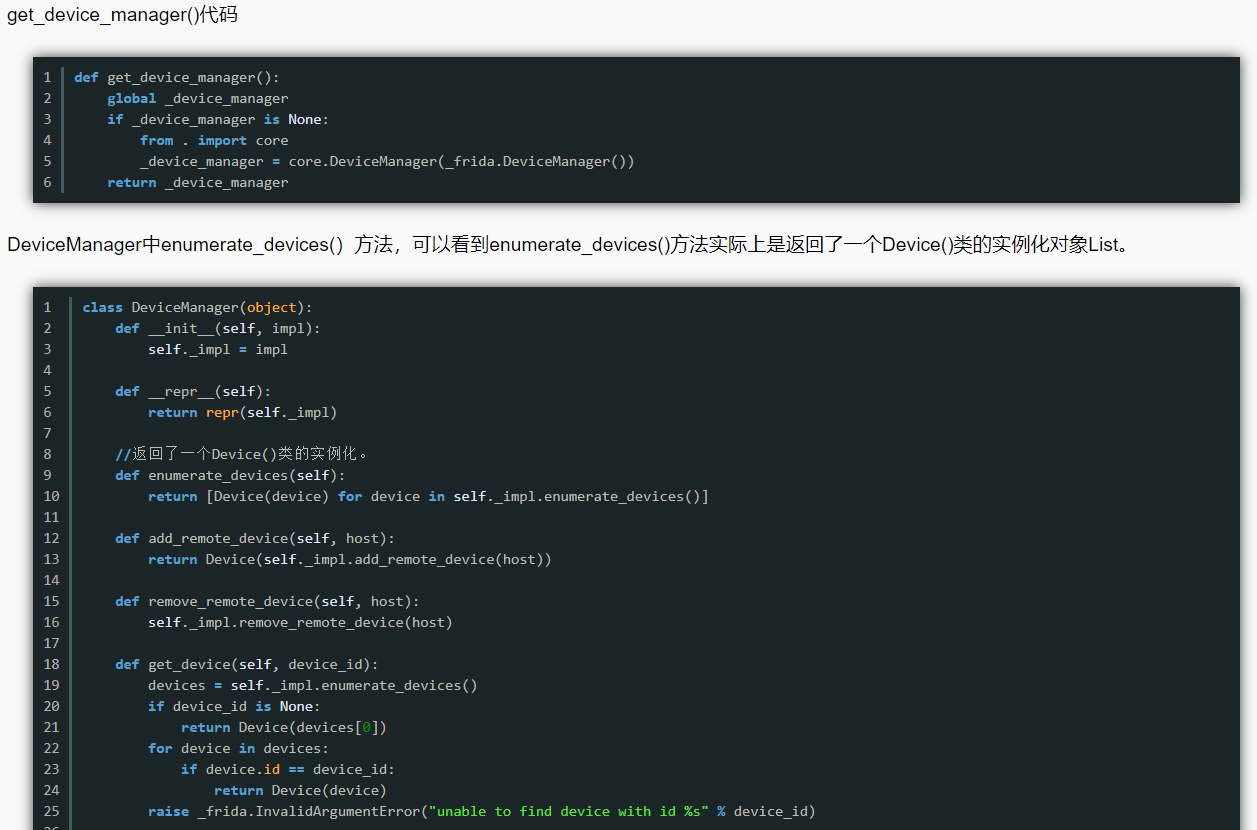


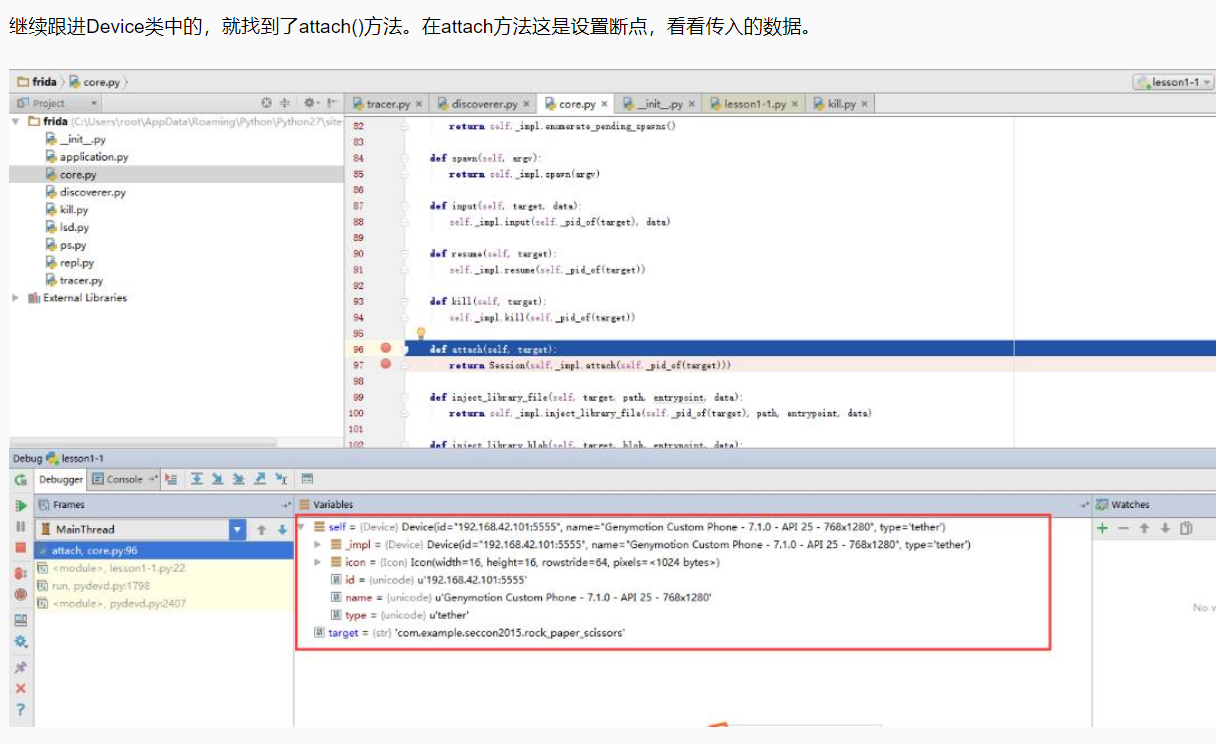


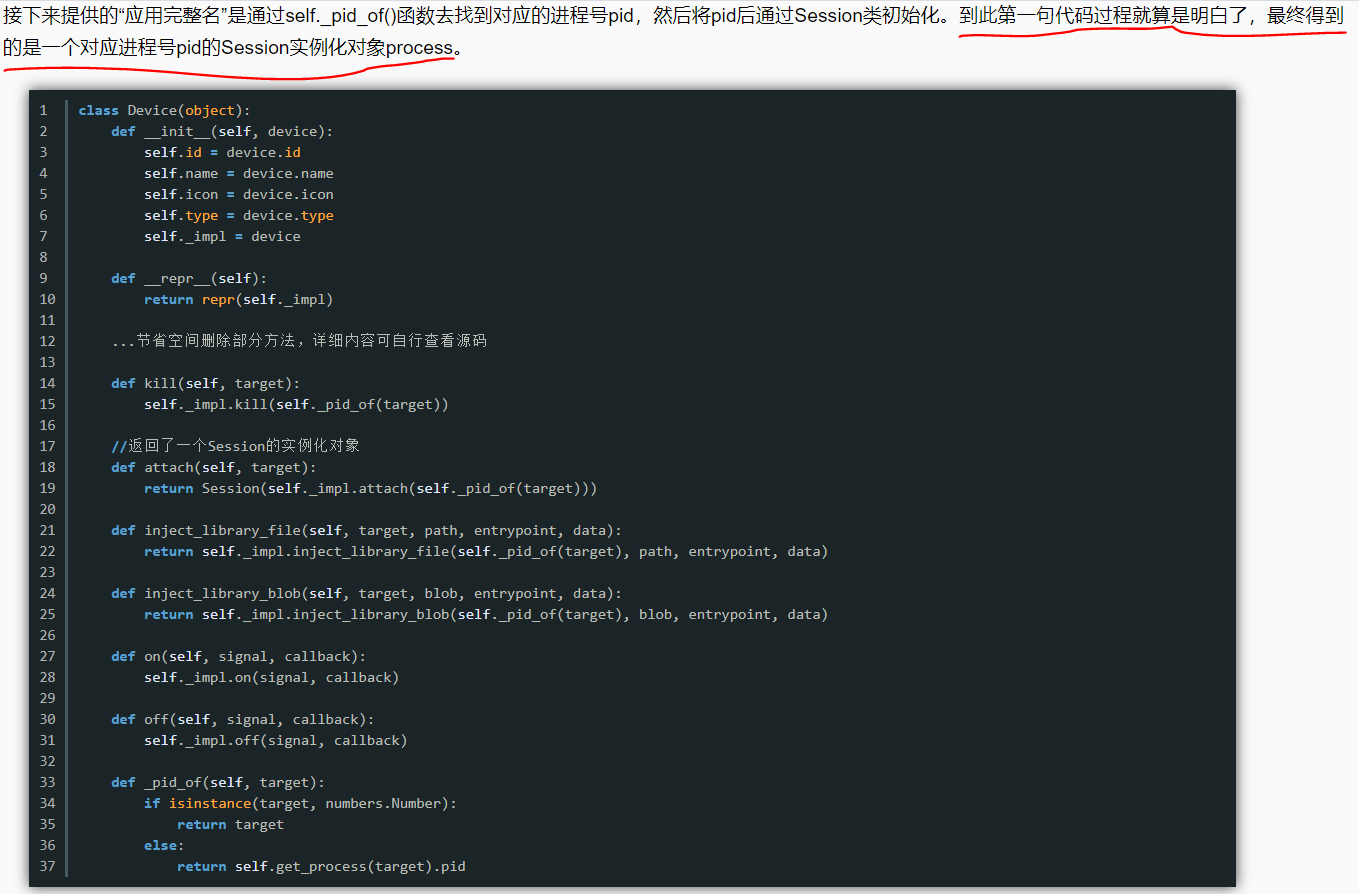
## 3.2 frida自带的Messages机制与进程交互



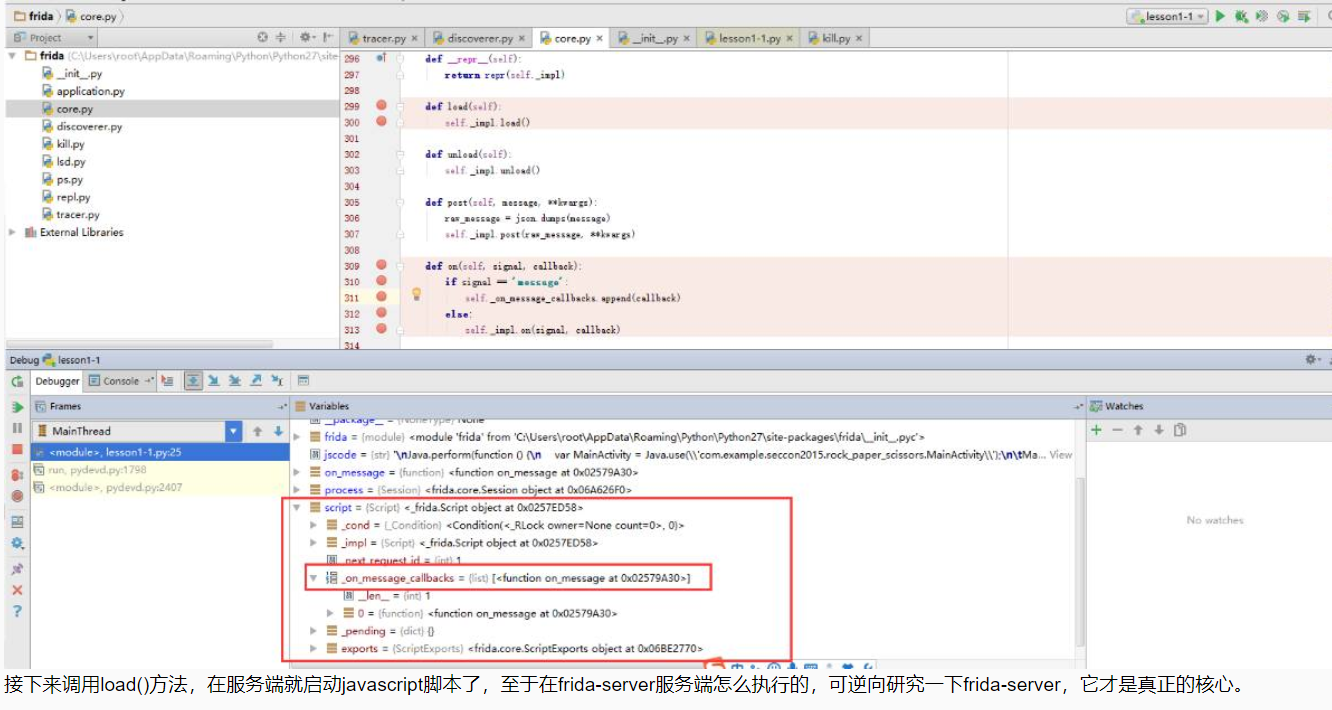












## 参考链接

<https://bbs.pediy.com/thread-227232.htm>

<https://bbs.pediy.com/thread-227233.htm>

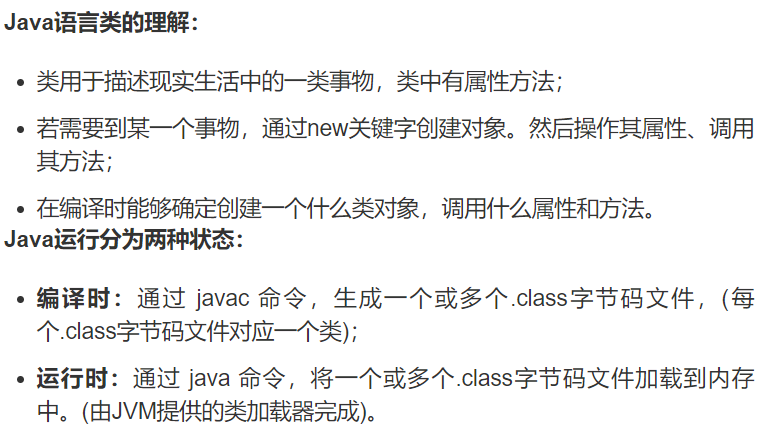
# 4 java反射机制

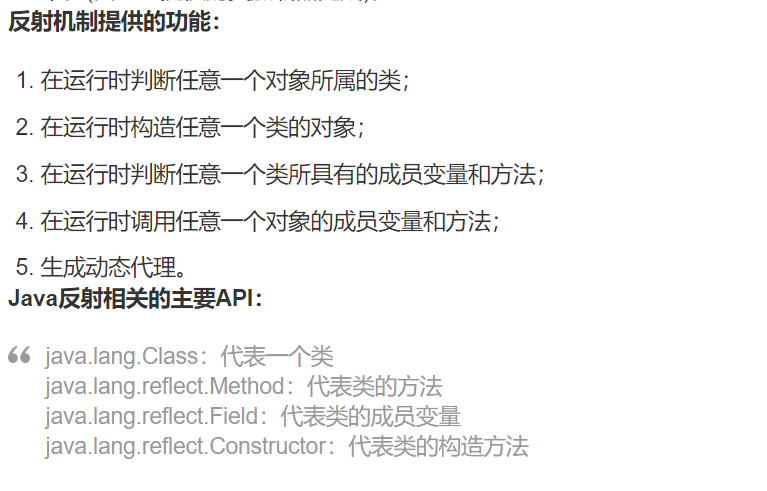
## 4.1 反射机制

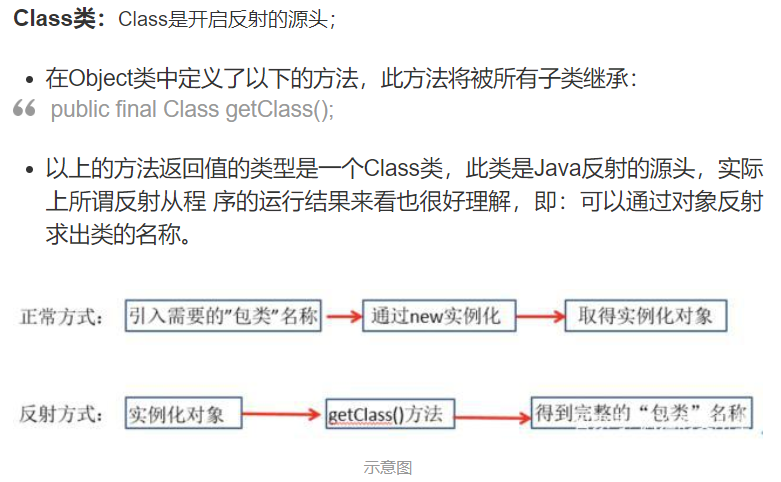
在运行状态中，**对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性**；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

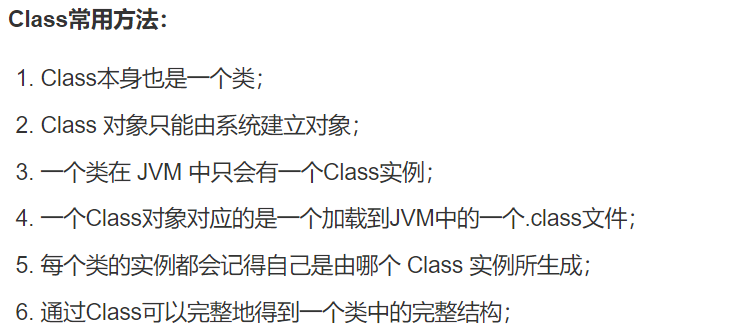
**反射(Reflection)**被视为动态语言的关键，反射机制允许程序在执行期借助于Reflection API取得任何类的内部信息，并能直接操作任意对象的内部属性及方法。

## 4.2 反射研究与应用



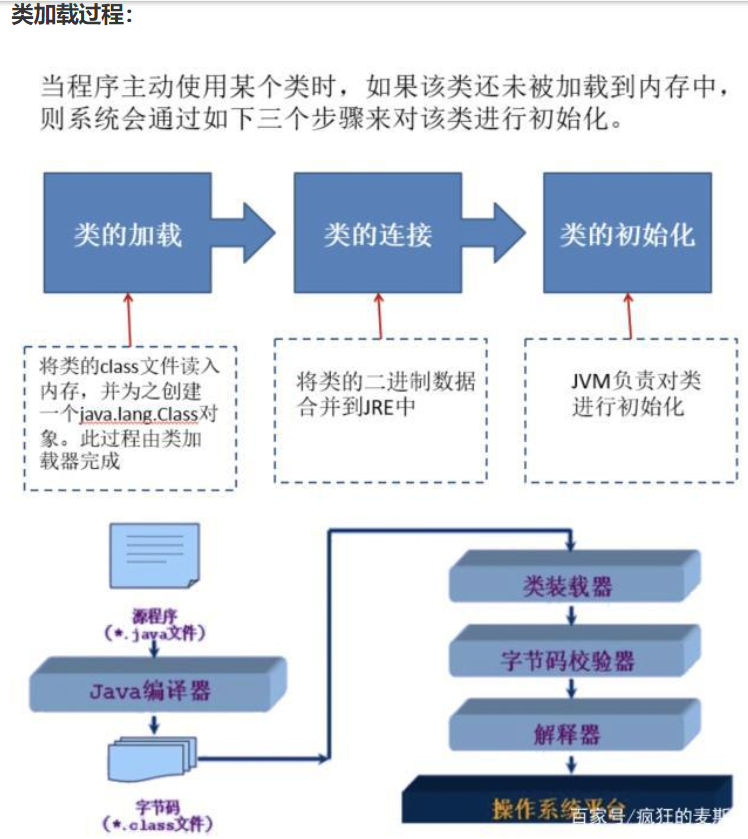


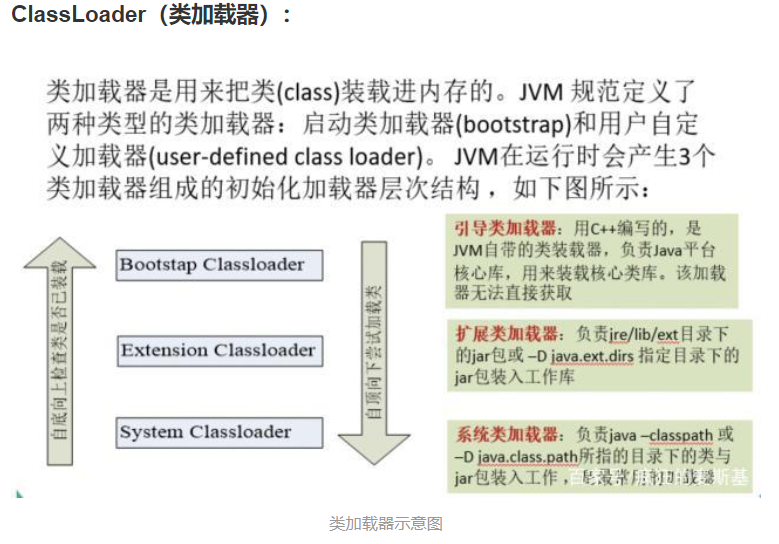






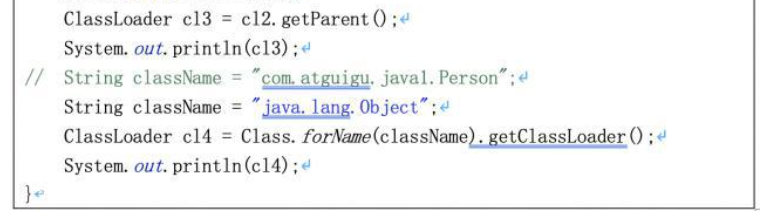




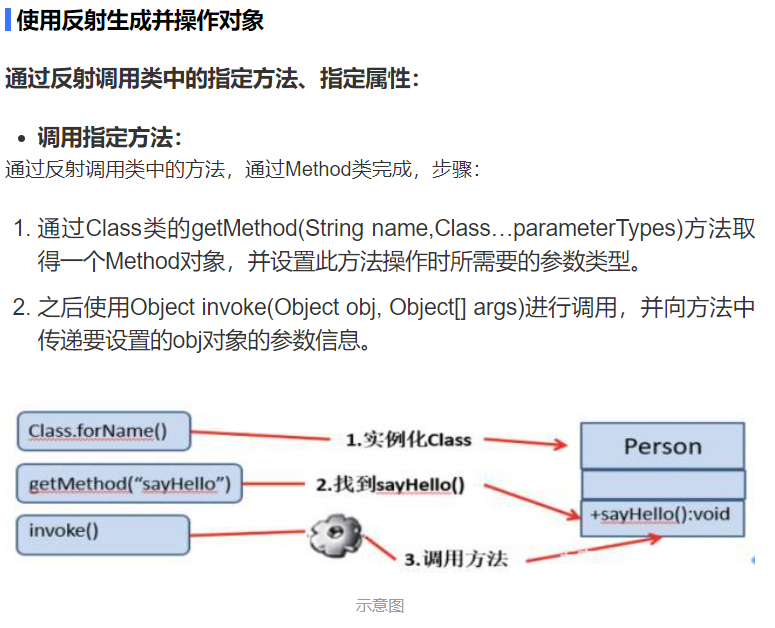


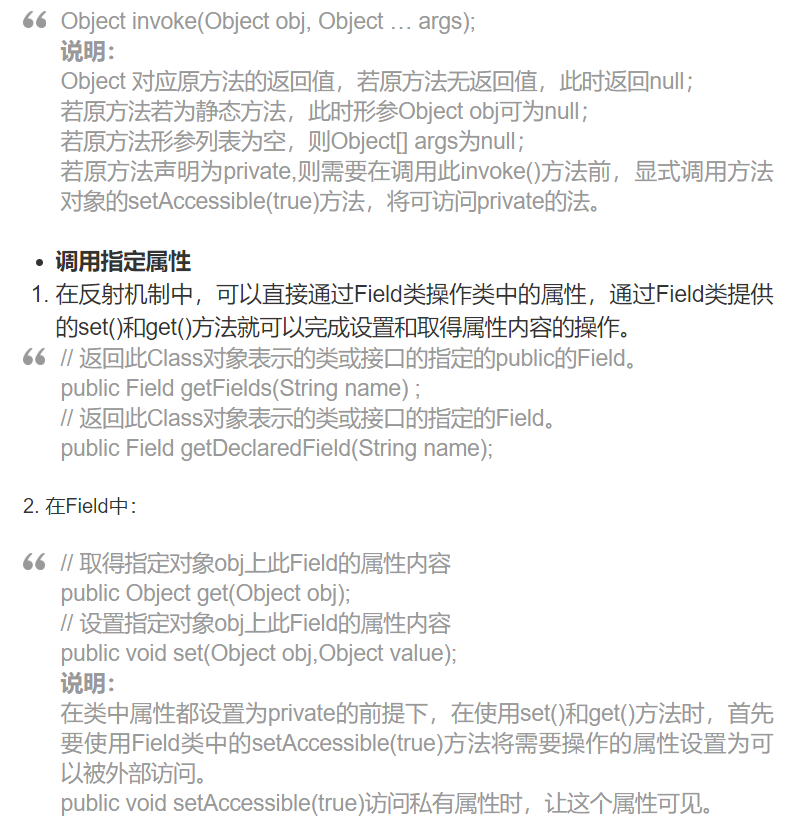












## 参考链接

<https://baijiahao.baidu.com/s?id=1645023991123759354&wfr=spider&for=pc>

<https://blog.csdn.net/a745233700/article/details/82893076>

# 5 静态代理和动态代理

## 5.1 代理概念

为某个对象提供一个代理，以控制对这个对象的访问。 代理类和委托类有共同的父类或父接口，这样在任何使用委托类对象的地方都可以用代理对象替代。代理类负责请求的预处理、过滤、将请求分派给委托类处理、以及委托类执行完请求后的后续处理。



## 5.2 静态代理

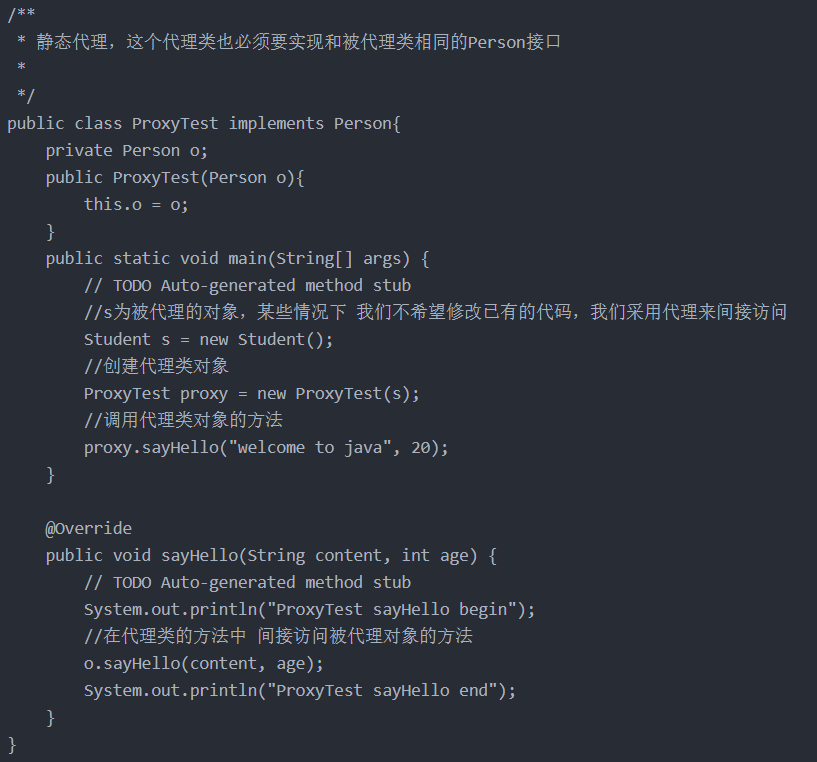
静态代理：在代码编译时就确定了被代理的类是哪一个。

这个静态代理比较简单，代理类和被代理类实现了同一接口，在代理类的构造函数中定义一个被代理类的对象即可。

Student类：实现Person接口（接口中有sayHello()方法）。

被代理类和Person类就比较简单，这里就不附上源码了。

代理类：StudentProxy类 实现Person接口



## 5.3 动态代理

动态代理：java的动态代理机制中，有两个重要的类或者接口，一个是InvocationHandler（Interface），还有一个是Proxy（是一个class）。

**其中InvocationHandler是每一个动态代理类都必须要实现的接口，我们通过代理对象调用一个方法的时候，该方法就会被转发由InvocationHandler这个接口的invoke方法来调用。**

**Proxy类，动态的创建一个代理对象的类，它提供了许多方法，我们用的最多的是newProxyInstance方法，该方法的作用就是得到一个动态的代理对象。**

动态代理模式主要由元素共同组成：

1.接口：定义具体实现的方法

2.被代理类：实现上述接口，执行接口中的方法

3.代理类：实现InvocationHandler，帮助被代理类实现方法。

代理类中要实现的内容有：

1.因为动态代理不知道被代理的类是哪一个，所以在实现了InvocationHandler的代理类中定义了一个Object类，在代理类的构函数中作为参数传递进来。

2.实现InvocationHandler中的invoke方法。

3.写main方法，并在main方法中根据定义的被代理类实现代理类的生成。

动态代理的步骤：

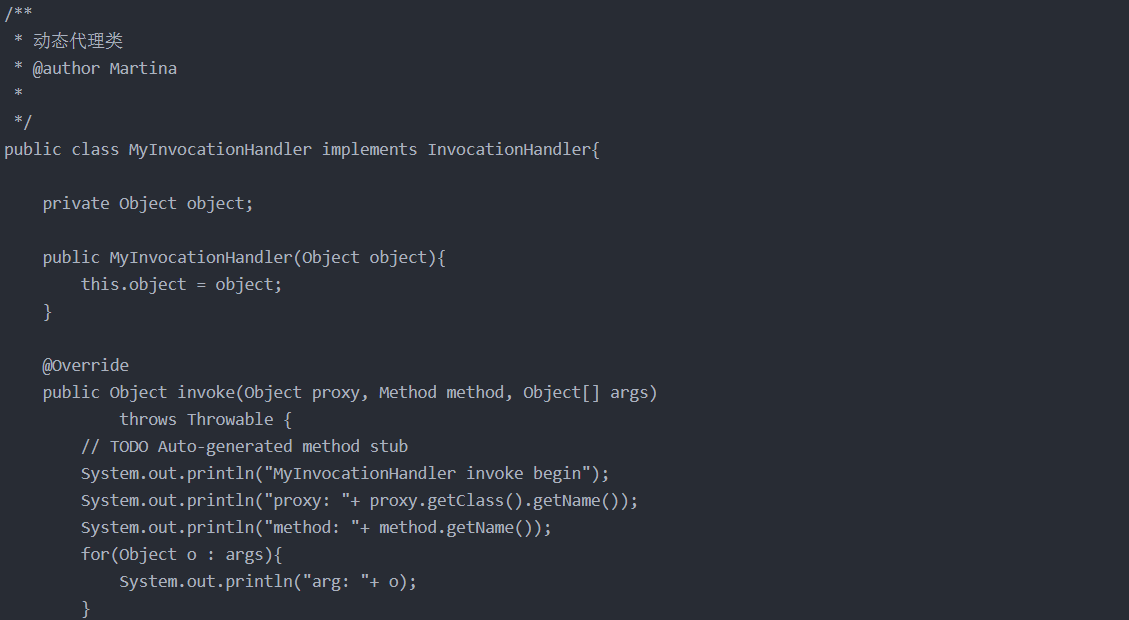
1. 首先获得一个被代理对象的引用，

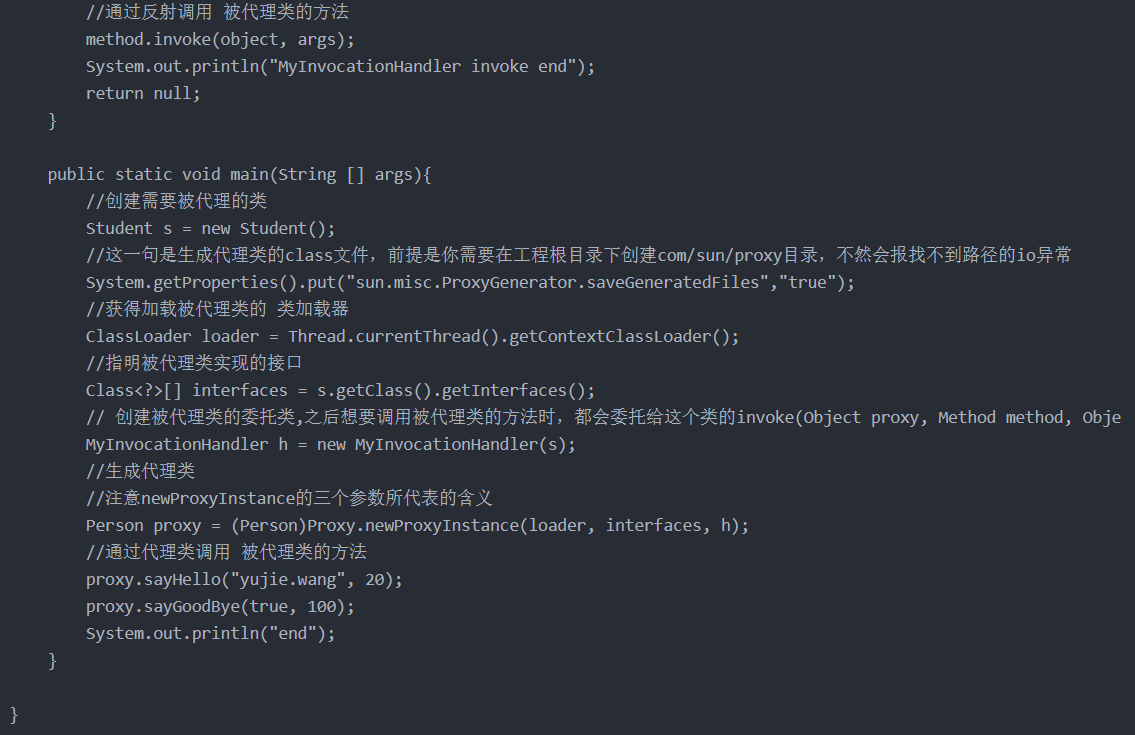
2. 获得该引用的接口

3. 生成一个类，这个类实现了我们给的代理对象所实现的接口

4. 上述类编译生成了.class字节码供JVM使用

5. 调用上述生成的class





## 参考链接

<https://blog.csdn.net/mulinsen77/article/details/84730087>

<https://blog.csdn.net/giserstone/article/details/17199755>

<https://blog.csdn.net/weixin_39079048/article/details/98852947>