- Andfix
  - 即时生效原理
    - app启动到一半的时候，所有需要发生变更的类已经被加载过了，

在Android系统中是无法对一个已经加载的类进行卸载的。腾讯的Tinker的方案是让ClassLoader去加载新的类，如果不重启app，原有的类还在虚拟机中，就无法加载新的类。因此需要冷启动后，抢先加载修复补丁中的新类，从而达到热修复的目的。
    - AndFix采用的方法是直接在已加载的类中的native层替换掉原方法，是在原有类的基础上进行修改的。
  - 底层替换原理
    - 每一个Java方法在Art虚拟机中都对应一个ArtMethod，ArtMethod记录了该方法的所有信息，包括所属类、访问权限、代码执行地址等。
    - 通过env->FromReflectedMethod，可以由Method对象得到这个方法对应的ArtMethod的真正起始地址，然后强转为ArtMethod指针，通过指针的操作对其成员属性进行修改替换。
  - 为什么这样替换后就可以实现热修复呢？
    - 就需要先了解虚拟机调用方法的原理
  - 虚拟机调用方法的原理
    - Android 5.1.1版本中的ArtMethod结构体路径：

art/runtime/mirror/art_method.h#560
      - 最重要的字段就是entry_point_from_interpreter_和entry_point_from_quick_compiled_code_，从名字可以看出它们就是方法的执行入口。
    - Java代码在Android中会被编译成Dex code

Art虚拟机中可以采用解释模式或AOT机器码模式执行Dex Code
      - 解释模式就是取出Dex Code，逐条解释执行。如果方法的调用者是以解释模式运行的，调用该方法时，就会获取它的entry_point_from_interpreter_,然后跳转执行。
      - 如果是AOT的方式，就会预编译Dex code对应的机器码，然后再运行期间直接执行机器码，不需要逐条解释执行dex code。如果方法的调用者是以AOT机器码方式执行的，在调用该方法时就是跳转到entry_point_from_quick_compiled_code_中执行的。
  - 那是不是替换方法的执行入口就可以了呢？
    - 当然不是，无论是解释模式还是AOT机器码模式，在运行期间还会需要调用ArtMethod中的其他成员字段。
  - 原理分析
    - 源码证明（类加载过程）
      - /frameworks/base/core/jni/AndroidRuntime.cpp#951
        - 参数className的值等于“com.android.internal.os.ZygoteInit”，本地变量env是从调用另外一个成员函数startVm创建的ART虚拟机获得的JNI接口。函数的目标就是要找到一个名称为com.android.internal.os.ZygoteInit的类，以及它的静态成员函数main，然后就以这个函数为入口，开始运行ART虚拟机
          - 1. 调用JNI接口FindClass加载com.android.internal.os.ZygoteInit类。
            - 1028
          - 2. 调用JNI接口GetStaticMethodID找到com.android.internal.os.ZygoteInit类的静态成员函数main。
            - 1033
            - http://androidxref.com/5.1.1_r6/xref/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java#648
          - 3. 调用JNI接口CallStaticVoidMethod开始执行com.android.internal.os.ZygoteInit类的静态成员函数main。
            - 1039
        - 需要注意的是这里执行main方法的是本地机器指令，而不是dex字节码
      - env->FindClass
      - /art/runtime/jni_internal.cc#FindClass()#599
        - 在ART虚拟机进程中，存在着一个Runtime单例，用来描述ART运行时。通过调用Runtime类的静态成员函数Current可以获得上述Runtime单例。获得了这个单例之后，就可以调用它的成员函数GetClassLinker来获得一个ClassLinker对象。
          - ClassLinker对象是在创建ART虚拟机的过程中创建的，用来加载类以及链接类方法
        - JNI类的静态成员函数FindClass首先是判断ART运行时是否已经启动起来。如果已经启动，那么就通过调用函数GetClassLoader来获得当前线程所关联的ClassLoader，并且以此为参数，调用前面获得的ClassLinker对象的成员函数FindClass来加载由参数name指定的类。
        - 如果ART运行时还没有启动，那么这时候只可以加载系统类。这个通过前面获得的ClassLinker对象的成员函数FindSystemClass来实现的。
      - /art/runtime/class_linker.cc#FindClass()#2117
        - 参数descriptor指向的是要加载的类的签名，而参数class_loader指向的是一个类加载器
        - 首先是调用另外一个成员函数LookupClass来检查参数descriptor指定的类是否已经被加载过。如果是的话，那么ClassLinker类的成员函数LookupClass就会返回一个对应的Class对象，这个Class对象接着就会返回给调用者，表示加载已经完成。
        - 如果参数descriptor指定的类还没有被加载过，这时候主要就是要看参数class_loader的值了。如果参数class_loader的值等于NULL，那么就需要调用FindInClassPath来在系统启动类路径寻找对应的类。一旦寻找到，那么就会获得包含目标类的DEX文件，因此接下来就调用ClassLinker类的另外一个成员函数DefineClass从获得的DEX文件中加载参数descriptor指定的类了
          - 这里调用FindInClassPath，实际要完成的工作就是从ClassLinker类的成员变量boot_class_path_描述的一系列的DEX文件中检查哪一个DEX文件包含有参数descriptor指定的类。
          - 所谓的系统启动类路径，其实就是一系列指定的由系统提供的DEX文件，这些DEX文件保存在ClassLinker类的成员变量boot_class_path_描述的一个向量中
        - 知道了参数descriptor指定的类定义在哪一个DEX文件之后，就可以通过ClassLinker类的另外一个成员函数DefineClass来从中加载它了。
      - /art/runtime/class_linker.cc#DefineClass()#2218
        - ClassLinker类有一个类型为bool的成员变量init_done_，用来表示ClassLinker是否已经初始化完成。
        - 如果ClassLinker正处于初始化过程，即其成员变量init_done_的值等于false，并且参数descriptor描述的是特定的内部类，那么就将本地变量klass指向它们，其余情况则会通过成员函数AllocClass为其分配存储空间，以便后面通过成员函数LoadClass进行初始化。
        - ClassLinker类的成员函数LoadClass用来从指定的DEX文件中加载指定的类。指定的类从DEX文件中加载完成后，需要通过另外一个成员函数InsertClass添加到ClassLinker的已加载类列表中去。如果指定的类之前已经加载过，即调用成员函数InsertClass得到的返回值不等于空，那么就说明有另外的一个线程也正在加载指定的类。这时候就需要调用成员函数EnsureResolved来保证（等待）该类已经加载并且解析完成。另一方面，如果没有其它线程加载指定的类，那么当前线程从指定的DEX文件加载完成指定的类后，还需要调用成员函数LinkClass来对加载后的类进行解析。最后，一个类型为Class的对象就可以返回给调用者了，用来表示一个已经加载和解析完成的类。
      - /art/runtime/class_linker.cc#LoadClass()#2727
        - dex_file: 类型为DexFile，描述要加载的类所在的DEX文件。

dex_class_def: 类型为ClassDef，描述要加载的类在DEX文件里面的信息。

klass: 类型为Class，描述加载完成的类。

class_loader: 类型为ClassLoader，描述所使用的类加载器。
        - 总的来说，ClassLinker类的成员函数LoadClass的任务就是要用dex_file、dex_class_def、class_loader三个参数包含的相关信息设置到参数klass描述的Class对象去，以便可以得到一个完整的已加载类信息。
        - 主要工作
          - setClassLoader：将class_loader描述的ClassLoader设置到klass描述的Class对象中，即给每一个已加载的类关联一个类加载器。
          - SetDexClassDefIndex：通过DexFile的成员函数GetIndexForClassDef获得正在加载的类在Dex文件中的索引号，并设置到klass中
        - FindOatClass：从相应的OAT文件中找到与正在加载的类对应的一个OatClass结构体oat_class。这需要利用到上面提到的DEX类索引号，这是因为DEX类和OAT类根据索引号存在一一对应关系。
      - /art/runtime/class_linker.cc#LoadClassMembers()#2767
        - 从参数dex_file描述的DEX文件中获得正在加载的类的静态成员变量和实例成员变量个数，并且为每一个静态成员变量和实例成员变量都分配一个ArtField对象，接着通过ClassLinker类的成员函数LoadField对这些ArtField对象进行初始化。初始化得到的ArtField对象全部保存在klass描述的Class对象中。
        - 从参数dex_file描述的DEX文件中获得正在加载的类的直接成员函数和虚拟成员函数个数，并且为每一个直接成员函数和虚拟成员函数都分配一个ArtMethod对象，接着通过ClassLinker类的成员函数LoadMethod对这些ArtMethod对象进行初始化。初始好得到的ArtMethod对象全部保存在klass描述的Class对象中。
        - 总结来说，参数klass描述的Class对象包含了一系列的ArtField对象和ArtMethod对象，其中，ArtField对象用来描述成员变量信息，而ArtMethod用来描述成员函数信息。
        - 接下来，我们继续分析全局函数LinkCode的实现，以便可以了解如何在一个OAT文件中找到一个DEX类方法的本地机器指令。
      - /art/runtime/class_linker.cc#2627_LinkCode()#2627
        - 参数method表示要设置本地机器指令的类方法。

参数oat_class表示类方法method在OAT文件中对应的OatClass结构体。

参数dex_file表示在dex文件中对应的DexFile

参数dex_method_index表示DexFile中方法method的索引号。

参数method_index表示类方法method的索引号。
        - 通过参数method_index描述的索引号可以在oat_class表示的OatClass结构体中找到一个OatMethod结构体oat_method。这个OatMethod结构描述了类方法method的本地机器指令相关信息，通过调用它的成员函数LinkMethod可以将这些信息设置到参数method描述的ArtMethod对象中去.
        - #2643_LinkMethod()

/art/runtime/oat_file.cc#595
          - 其中，最重要的就是通过OatMethod类的成员函数GetPortableCode和GetQuickCode获得OatMethod结构体中的code_offset_字段，并且通过调用ArtMethod类的成员函数SetEntryPointFromCompiledCode设置到参数method描述的ArtMethod对象中去。OatMethod结构体中的code_offset_字段指向的是一个本地机器指令函数，这个本地机器指令函数正是通过翻译参数method描述的类方法的DEX字节码得到的。
          - /art/runtime/oat_file.h#GetPortableCode()#100
          - /art/runtime/oat_file.h#GetQuickCode()#109
        - #2647_NeedsInterpreter()

/art/runtime/class_linker.cc#2525
          - 检查参数method描述的类方法是否需要通过解释器执行
          - 在以下两种情况下，一个类方法需要通过解释器来执行：

1. 没有对应的本地机器指令，即参数quick_code和portable_code的值等于NULL。

2. ART虚拟机运行在解释模式中，并且类方法不是JNI方法，并且也不是代理方法。
            - 因为JNI方法是没有对应的DEX字节码的，因此即使ART虚拟机运行在解释模式中，JNI方法也不能通过解释器来执行。至于代理方法，由于是动态生成的（没有对应的DEX字节码），因此即使ART虚拟机运行在解释模式中，它们也不通过解释器来执行
          - 调用Runtime类的静态成员函数Current获得的是描述ART运行时的一个Runtime对象。调用这个Runtime对象的成员函数GetInstrumentation获得的是一个Instrumentation对象。这个Instrumentation对象是用来调试ART运行时的，通过调用它的成员函数InterpretOnly可以知道ART虚拟机是否运行在解释模式中。
        - #2654
          - 如果调用函数NeedsInterpreter得到的返回值enter_interpreter等于true，那么就意味着参数method描述的类方法需要通过解释器来执行，而且不是Native方法，这时候就将函数artInterpreterToInterpreterBridge设置为解释器执行该类方法的入口点。否则的话，就将另外一个函数artInterpreterToCompiledCodeBridge设置为解释器执行该类方法的入口点。
        - #2660
          - 判断参数method描述的类方法是否是一个抽象方法。抽象方法声明类中是没有实现的，必须要由子类实现。因此抽象方法在声明类中是没有对应的本地机器指令的，它们必须要通过解释器来执行。不过，为了能够进行统一处理，我们仍然假装抽象方法有对应的本地机器指令函数。
        - #2669
          - 当参数method描述的类方法是一个非类静态初始化函数(class initializer)的静态方法时，我们不能直接执行翻译其DEX字节码得到的本地机器指令。这是因为类静态方法可以在不创建类对象的前提下执行。这意味着一个类静态方法在执行的时候，对应的类可能还没有初始化好。这时候我们就需要先将对应的类初始化好，再执行相应的静态方法。
        - #2703
          - 判断参数method描述的类方法是否是一个JNI方法。如果是的话，那么就调用ArtMethod类的成员函数UnregisterNative来初始化它的JNI方法调用接口。
        - #2705_UnregisterNative()

/art/runtime/mirror/art_method.cc#364
          - 实际上就是将一个JNI方法的初始化入口设置为通过调用函数GetJniDlsymLookupStub获得的一个Stub。这个Stub的作用是，当一个JNI方法被调用时，如果还没有显示地注册有Native函数，那么它就会自动从已加载的SO文件查找是否存在一个对应的Native函数。如果存在的话，就将它注册为JNI方法的Native函数，并且执行它。这就是隐式的JNI方法注册。
        - #2717_UpdateMethodsCode()

/art/runtime/instrumentation.cc#679
          - 最后调用Instrumentation类的成员函数UpdateMethodsCode检查是否要进一步修改参数method描述的类方法的本地机器指令入口。
          - Instrumentation类是用来调用ART运行时的。例如，当我们需要监控类方法的调用时，就可以往Instrumentation注册一些Listener。这样当类方法调用时，这些注册的Listener就会得到回调。当Instrumentation注册有相应的Listener时，它的成员变量instrumentation_stubs_installed_的值就会等于true。
        - #2717_UpdateEntrypoints()

/art/runtime/instrumentation.cc#85
      - 总结
        - 通过上述源码跟踪分析，一个类的加载过程完成了。加载完成后得到的是一个Class对象。这个Class对象关联有一系列的ArtField对象和ArtMethod对象。其中，ArtField对象描述的是成员变量，而ArtMethod对象描述的是成员函数。对于每一个ArtMethod对象，它都有一个解释器入口点和一个本地机器指令入口点。这样，无论一个类方法是通过解释器执行，还是直接以本地机器指令执行，我们都可以以统一的方式来进行调用。同时，理解了上述的类加载过程后，我们就可以知道，我们在Native层通过JNI接口FindClass查找或者加载类时，得到的一个不透明的jclass值，实际上指向的是一个Class对象。
    - 类方法查找过程
      - JNI#GetStaticMethodID

/art/runtime/jni_internal.cc#943
        - 参数name和sig描述的分别是要查找的类方法的名称和签名，而参数java_class的是对应的类。参数java_class的类型是jclass，从前面类加载过程的分析可以知道，它实际上指向的是一个Class对象。
      - JNI#FindMethodID

/art/runtime/jni_internal.cc#140
        - 函数FindMethodID的执行过程:
          - 1. 将参数jni_class的值转换为一个Class指针c，因此就可以得到一个Class对象，并且通过ClassLinker类的成员函数EnsureInitialized确保该Class对象描述的类已经初始化。
          - 2. Class对象c描述的类在加载的过程中，经过解析已经关联上一系列的成员函数。这些成员函数可以分为两类：Direct和Virtual。Direct类的成员函数包括所有的静态成员函数、私有成员函数和构造函数，而Virtual则包括所有的虚成员函数。
            - FindInterFaceMethod其实也是Virtual
          - 3. 经过前面的查找过程，如果都不能在Class对象c描述的类中找到与参数name和sig对应的成员函数，那么就抛出一个NoSuchMethodError异常。否则的话，就将查找得到的ArtMethod对象封装成一个jmethodID值返回给调用者。
      - 我们通过调用JNI接口GetStaticMethodID获得的不透明jmethodID值指向的实际上是一个ArtMethod对象。

当我们获得了一个ArtMethod对象之后，就可以轻松地得到它的本地机器指令入口，进而对它进行执行。
        - http://androidxref.com/5.1.1_r6/xref/art/runtime/scoped_thread_state_change.h#171
    - 原理总结
      - 当我们把旧方法(ArtMethod)的所有成员字段都替换为新方法(ArtMethod)的成员字段后，执行时所有的数据就可以保持和新方法的数据一致。
这样在所有执行到旧方法的地方，会获取新方法的执行入口、所属类型、方法索引号、以及所属dex信息，然后像调用旧方法一样，执行新方法的逻辑。
  - 兼容性问题的根源
    - 前面分析的实现原理就导致AndFix必定存在严重的兼容性问题
    - 虽然AndFix把底层结构强转为ArtMethod，但这里的ArtMethod并非App运行时所在设备虚拟机中的ArtMethod，而是自己构造的ArtMethod。虽然其中成员字段和AOSP开源代码中是完全一样，但是由于Android源码是开源的，各手机厂商都可以对代码进行修改，如果手机厂商对ArtMethod结构体进行了修改，那么AndFix的替换机制就会出问题了。
    - 从代码角度角度举例分析
      - 比如AndFix替换declaring_class_的地方
        - smeth->declaring_class_ = dmeth->declaring_class_;
      - 由于declaring_class_是AndFix自己构造的ArtMethod中的第一个成员字段，因此它等价于：
        - *(uint32_t*) (smeth + 0) = *(uint_32_t*) (dmeth + 0);
      - 如果手机厂商在ArtMethod结构体中的declaring_class_字段前添加了其他字段，比如additional_，那么additional_就成为了ArtMethod实际上第一个成员字段，所以smeth+0这个位置在这台设备上实际就变成了additional_，而不是declaring_class_。这行代码的真正意义就变成了：
        - smeth->additional_ = dmeth->additional_;
      - 这样就和原有替换declaring_class_的逻辑不一致，从而无法正常执行热修复逻辑。
  - 代码手写实现
  - 突破兼容性解决方案
    - 我们知道AndFix目前的替换思路是替换ArtMethod结构体的所有成员逐一替换。那么我们可以考虑整个结构体进行替换
    - 逐个替换的代码可以简化为：
      - memcpy(smeth, dmeth, sizeof(ArtMethod));
    - 这样通过整个结构体的替换，就能够把所有旧方法成员自动对应的替换成新方法的成员
    - 但是这里的关键在于sizeof(ArtMethohd)的计算结果，我们需要在运行时动态获取当前设备虚拟机的ArtMethod结构体的大小。
    - 从底层数据结构以及排列特点入手，在Art中初始化一个类的时候会给这个类的所有方法分配内存空间，看源码：
      - art/runtime/class_linker.cc#LoadClassMembers()#2767
    - 类的方法有direct和virtual之分。
      - direct方法包含static方法和所有不可继承的对象方法。
      - virtual方法包含所有可以继承的对象方法
    - 继续跟进源码：AllocArtMethodArray方法，分配内存，可以知道ArtMethod数组是紧密排列的，所以可以通过相邻ArtMethod结构体的起始地址差值的大小来确定
    - 代码实现角度分析
      - 我们可以自己构造一个类
        - public class NativeStructsModel{

final public static void f1(){}

final public static void f2(){}

}
      - 这里的f1()和f2()都是static方法，所以都属于direct方法，属于direct ArtMethod Array。由于我们自己构造的NativeStructsModel类中只存在两个方法，因此它们肯定是相邻的。
      - 那么我们就可以通过JNI层取得它们的地址差值：
        - size_t firstMethodId = (size_t) env->GetStaticMethodID(nativeStructsModelClazz, "f1", "()V");

size_t secondMethodId = (size_t) env->GetStaticMethodID(nativeStructsModelClazz, "f2", "()V");

size_t methodSize = secondMethodId - firstMethodId;
      - 然后就可以用methodSize来取代前面memcpy中的sizeof(ArtMethod)了
        - mememcpy(smeth, dmeth, methodSize);