代码修复

代码修复有两大主要方案，一种是阿里系的底层替换方案，另一种是腾讯系的类加载方案。

两种方案各有优劣：

底层替换方案限制颇多，但时效性最好，加载轻快，立即见效。

类加载方案时效性差，需要重新冷启动才能见效，但修复范围广，限制少。

底层替换方案

底层替换方案是在已经加载了的类中直接替换掉原有方法，是在原来类的基础上进行修改的。因而无法实现对与原有类进行方法和字段的增减，因为这样将破坏原有类的结构。

一旦补丁类中出现了方法的增加和减少，就会导致这个类以及整个 Dex 的方法数的变化。方法数的变化伴随着方法索引的变化，这样在访问方法时就无法正常地索引到正确的方法了。如果字段发生了增加和减少，和方法变化的情况一样，所有字段 的索引都会发生变化。并且更严重的问题是，如果在程序运行中间某个类突然增加了 —个字段，那么对于原先已经产生的这个类的实例，它们还是原来的结构，这是无法改变的。而新方法使用到这些老的实例对象时，访问新增字段就会产生不可预期 的结果。

这是这类方案的固有限制，而底层替换方案最为人诟病的地方，在于底层替换的不稳定性。

通过对代码的底层替换原理重新进行了深入思考，从克服其限制和兼容性入手，以一种更加优雅的替换思路，实现了即时生效的代码热修复。

采用一种无视底层具体结构的替换方式，这种方式不仅解决了兼容性问题，并且由于忽略了底层ArtMethod结构的差异，对于所有的Android版本都不 再需要区分，代码量大大减少。即使以后的Android版本不断修改ArtMethod的成员，只要保证ArtMethod数组仍是以线性结构排列，就能直接适用于将来的 Android 8.0、9.0等新版本，无需再针对新的系统版本进行适配了。

类加载方案

类加载方案的原理是在 app 重新启动后让 Classloader 去加载新的类。因为在 app运行到一半的时候，所有需要发生变更的类已经被加载过了，在 Android 上是无法对一个类进行卸载的。如果不重启，原来的类还在虚拟机中，就无法加载新类。 因此，只有在下次重启的时候，在还没走到业务逻辑之前抢先加载补丁中的新类，这样后续访问这个类时，就会Resolve为新类。从而达到热修复的目的。

系统默认使用的是 PathClassLoader，继承自 BaseDexClassLoader，在 BaseDexClassLoader 里，有一个 DexPathList 变量，在 DexPathList 的实现里，有一个 Element[] dexElements 变量，这里面保存了所有的 dex。在加载 Class 的时候，就遍历 dexElements 成员，依次查找 Class，找到以后就返回。那么自然就想到了把补丁 dex 插入到 dexElements 最前面，这样遍历 dexElements 就会优先从补丁 dex 中查找 Class 了。

微信的 Tinker 方案是完整的全量 dex 加载，并且可谓是将补丁合成做到了极致。Tinker 的合成方案，是从 dex 的方法和指令维度进行全量合成，整个过程都是自己研发的。虽然可以很大地节省空间，但由于对dex内容的比较粒度过细，实现较为复杂，性能消耗比较严重。实际上，dex 的大小占整个apk的比例是比较低的，一个 app 里面的dex文件大小并不是主要部分，而占空间大的主要还是资源文件。因此，Tinker 方案的时空代价转换的性价比不高。

dex 比较的最佳粒度，应该是在类的维度。它既不像方法和指令维度那样的细微，也不像 bsbiff 比较那般的粗糙。在类的维度，可以达到时间和空间平衡的最 佳效果。基于这个准则，另辟蹊径，实现了一种完全不同的全量dex替换方案。

直接利用 Android 原先的类查找和合成机制，快速合成新的全量 dex。这么一来，既不需要处理合成时方法数超过的情况，对于 dex 的结构也不用进行破坏性重构。

重新编排了包中dex的顺序。虚拟机查找类的时候，会优先找到 classes.dex 中的类，然后才是 classes2.dex、classes3.dex，也可以看做是 dex 文件级别的类插桩方案。这个方式对旧包与补丁包中 classes.dex 的顺 序进行了打破与重组，最终使得系统可以自然地识别到这个顺序，以实现类覆盖的目的。大大减少合成补丁的开销。

Sophix的代码修复体系正是同时涵盖了这两种方案。两种方案的结合，可以实现优势互补，完全兼顾的作用，可以灵活地根据实际情况自动切换。

在补丁生成阶段，补丁工具会根据实际代码变动情况进行自动选择，

针对小修改，在底层替换方案限制范围内的，就直接采用底层替换修复吗，这样可以做到代码修复即时生效。

对于代码修改超出底层替换限制的，会使用类加载替换，这样虽然及时性没那么好，但总归可以达到热修复的目的。

运行时阶段，Sophix 还会再判断所运行的机型是否支持热修复，这样即使补丁支持热修复，但由于机型底层虚拟机构造不支持，还是会走类加载修复，从而达到最好的兼容性。

资源修复

Instant Run 中的资源热修复分为两步:

构造一个新的AssetManager,并通过反射调用 addAssetPath，把这个完 整的新资源包加入到 AssetManager 中。这样就得到了一个含有所有新资源 的 AssetManager。

找到所有之前引用到原有AssetManager的地方，通过反射，把引用处替换为 AssetManager 。

新的实现方式：构造了一个 package id 为 0x66 的资源包，这个包里只包含改变了的资源项，然后直接在原有 AssetManager 中 addAssetPath 这个包就可以了。由于补丁包的 package id 为 0x66，不与目前已经加载的 0x7f 冲突，因此直接加入到已有的 AssetManager 中就可以直接使用了。

新的实现方式：构造了一个 package id 为 0x66 的资源包，这个包里只包含改变了的资源项，然后直接在原有 AssetManager 中 addAssetPath 这个包就可以了。由于补丁包的 package id 为 0x66，不与目前已经加载的 0x7f 冲突，因此直接加入到已有的 AssetManager 中就可以直接使用了。

补丁包里面的资源，只包含原有包里面没有而新的包里面有的新增资源，以及原有内容发生了改变的资源。并且，我们采用了更加优雅的替换方式，直接在原有的 AssetManager 对象上进行析构和重构，这样所有原先对 AssetManager对象的引用是没有发生改变的，所以就不需要像Instant Run 那样 进行繁琐的修改了。

可以说，我们的资源修复方案，优越性超过了 Google官方的Instant Run方 案。整个资源替换的方案优势在于：

不修改 AssetManager 的引用处，替换更快更完全。（对比 Instanat Run 以 及所有 copycat 的实现）

不必下发完整包，补丁包中只包含有变动的资源。（对比 Instanat Runs Amigo 等方式的实现）

不需要在运行时合成完整包。不占用运行时计算和内存资源。（对比 Tinker 的实现）

SO库修复

SO 库的修复本质上是对 native 方法的修复和替换。

我们采用的是类似类修复反射注入方式。把补丁 so 库的路径插入到 nativeLibraryDirectories 数组的最前面，就能够达到加载 so 库的时候是补丁 so 库，而不是原来 so 库的目录，从而达到修复的目的。

采用这种方案，完全由 Sophix 在启动期间反射注入 patch 中的 so 库。对开发者依然是透明的。不用像某些其他方案需要手动替换系统的 System.load 来实现替 换目的。