Android热修复技术选型——三大流派解析

所为 InfoQ 2016-09-09 08:00

2015年以来，Android开发领域里对热修复技术的讨论和分享越来越多，同时也出现了一些不同的解决方案，如QQ空间补丁方案、阿里AndFix以及微信Tinker，它们在原理各有不同，适用场景各异，到底采用哪种方案，是开发者比较头疼的问题。本文希望通过介绍QQ空间补丁、Tinker以及基于AndFix的阿里百川HotFix技术的原理分析和横向比较，帮助开发者更深入了解热修复方案。

技术背景

一、正常开发流程

从流程来看，传统的开发流程存在很多弊端：

- 重新发布版本代价太大
- 用户下载安装成本太高
- BUG修复不及时，用户体验太差

二、热修复开发流程

而热修复的开发流程显得更加灵活，优势很多：

- 无需重新发版，实时高效热修复

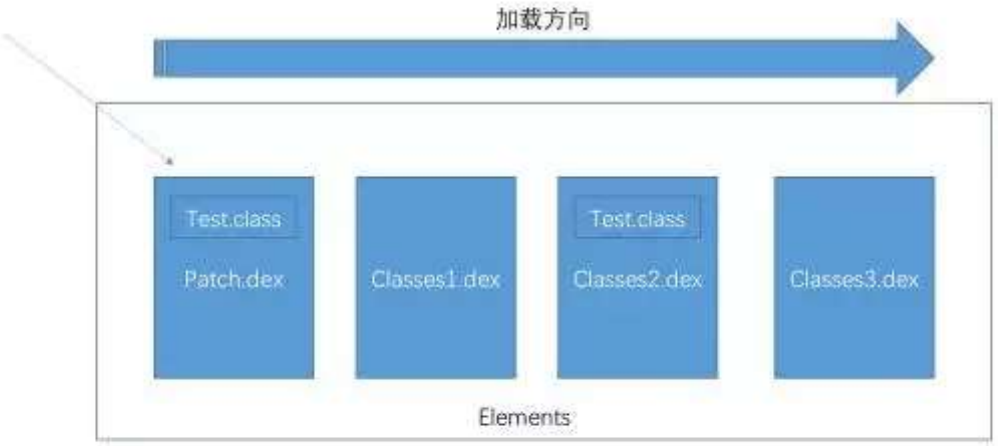
- 用户无感知修复，无需下载新的应用，代价小
- 修复成功率高，把损失降到最低

业界热门的热修复技术

热修复作为当下热门的技术，在业界内比较著名的有阿里巴巴的AndFix、Dexposed，腾讯QQ空间的超级补丁技术和微信的Tinker。最近阿里百川推出的HotFix热修复服务就基于AndFix技术，定位于线上紧急BUG的即时修复，所以AndFix技术这块我们重点分析阿里百川HotFix。下面，我们就分别介绍QQ空间超级热补丁技术和微信的Tinker以及阿里百川HotFix技术。

一、QQ空间超级补丁技术

超级补丁技术基于DEX分包方案，使用了多DEX加载的原理，大致的过程就是：把BUG方法修复以后，放到一个单独的DEX里，插入到dexElements数组的最前面，让虚拟机去加载修复完后的方法。



当patch.dex中包含Test.class时就会优先加载，在后续的DEX中遇到Test.class的话就会直接返回而不去加载，这样就达到了修复的目的。

但是有一个问题是，当两个调用关系的类不在同一个DEX时，就会产生异常报错。我们知道，在APK安装时，虚拟机需要将classes.dex优化成odex文件，然后才会执行。在这个过程中，会进行类的verify操作，如果调用关系的类都在同一个DEX中的话就会被打上CLASS_ISPREVERIFIED的标志，然后才会写入odex文件。

所以，为了可以正常的进行打补丁修复，必须避免类被打上CLASS_ISPREVERIFIED标志，具体的做法就是单独放一个类在另外DEX中，让其他类调用。

我们来逆向手机QQ空间APK看一下具体的实现：

先进入程序入口QZoneRealApplication，在attachBaseContext中进行了两步操作：修复CLASS_ISPREVERIFIED标志导致的unexpected DEX problem异常、加载修复的DEX。

```
protected void attachBaseContext(Context paramContext)
{
  int j = 0;
  try
  {
    super.attachBaseContext(paramContext);
    LoaderContext.init(this);
    Global.a(this);
    b();
    PatchProtector.a(this).a(15);
    PatchLibLoader.loadAndCopyDalvikHackDexFile(this);
    PatchLibLoader.loadPatchDex(LoaderContext.context(), 0);
    c();
    TimePrinter.a("application attachBaseContext >>>>>");
    paramContext = FundamentalStepManager.b;
    int k = paramContext.length;
    i = 0;
    while (i < k)
    {
      FundamentalStepManager.a(paramContext[i]).run();
      i += 1;
    }
    paramContext = Envi.process().processName();
    if ((paramContext != null) && (paramContext.contains(":service")))
    {
      i = 1;
      if (i == 0)
      {
        paramContext = FundamentalStepManager.a;
        k = paramContext.length;
        i = j;
        while (i < k)
        {
          FundamentalStepManager.a(paramContext[i]).run();
          i += 1;
        }
      }
      WnsdaemonHelper.a().a(this);
      TimePrinter.c("application attachBaseContext end >>>>>");
      return;
    }
  }
  catch (IllegalStateException paramContext)
  {
    for (;;)
    {
      continue;
      int i = 0;
    }
  }
}
```

1. 修复unexpectedDEX problem异常

先看代码：

```
public static boolean loadAndCopyDalvikHackDexFile(Context paramContext)
{
  boolean bool = ExtraLibLoader.load(paramContext, "libs/dalvikhack.jar", "com.qzone.dalvikhack.AntiLazyLoad", true, false);
  ExtraLibLoader.loadFailCheckAndKillProcess(bool);
  return bool;
}
```

可以看到，这里是要加载一个libs目录下的dalvikhack.jar。在项目的assets/libs找到该文件，解压得到classes.dex文件，逆向打开该DEX文件，

```
package com.qzone.dalvikhack;

public class AntiLazyLoad {}
```

通过不同的DEX加载进来，然后在每一个类的构造方法中引用其他dex中的唯一类AnitLazyLoad，避免类被打上CLASS_ISPREVERIFIED标志。

```
public class AndPatchLoader
{
  private static final String APATCH_PATH = "/out.jar";
  public static final String TAG = "AndPatchLoader";
  private static PatchManager mPatchManager = null;

  public AndPatchLoader()
  {
    if (NotDoVerifyClasses.DO_VERIFY_CLASSES) {
      System.out.print(AntiLazyLoad.class);
    }
  }

  public static void clear()
  {
    if (mPatchManager != null) {
      mPatchManager.a();
    }
  }
```

在无修复的情况下，将DO_VERIFY_CLASSES设置为false，提高性能。只有在需要修复的时候，才设置为true。

```
package com.qzone.dalvikhack;

public class NotDoVerifyClasses
{
    public static boolean DO_VERIFY_CLASSES = false;
}
```

至于如何加载进来，与接下来第二个步骤基本相同。

2. 加载修复的DEX

从 loadPatchDex() 方法进入，经过几次跳转，到达核心的代码段，SystemClassLoaderInjector.c()。由于进行了混淆和多次方法的跳转，于是将核心代码段做了如下整理：

```
//修复
public static void c(Context paramContext, String paramString1, String paramString2, String paramString3, boolean paramBoolean){
   Object localObject1 = paramContext.getClassLoader();
   PathClassLoader localPathClassLoader = (PathClassLoader)localObject1;
   paramContext = new DexClassLoader(paramString1, paramContext.getDir("odex", 0).getAbsolutePath(), paramString2, (ClassLoader)localObject1);
   localObject1 = combingArray(getElements(getPathList(localPathClassLoader)), getElements(getPathList(paramContext)), paramBoolean);
   Object localObject2 = getPathList(localPathClassLoader);
   Class paramClass = localObject2.getClass();
   paramClass = paramClass.getDeclaredField("dexElements");
   paramClass.setAccessible(true);
   paramClass.set(localObject2, localObject1);
   if (!TextUtils.isEmpty(paramString3)) {
     localPathClassLoader.loadClass(paramString3).newInstance();
   }
}

//得到DexPathList属性对象pathList
private static Object getPathList(Object paramObject){
   Class paramClass = Class.forName("dalvik.system.BaseDexClassLoader");
   paramClass = paramClass.getDeclaredField("pathList");
   paramClass.setAccessible(true);
   return paramClass.get(paramObject);
}

//得到Element[]
private static Object getElements(Object paramObject){
   Class paramClass = paramObject.getClass;
   paramClass = paramClass.getDeclaredField("dexElements");
   paramClass.setAccessible(true);
   return paramClass.get(paramObject);
}

//合并得到新的Elements数组
private static Object combineArray(Object firstArray, Object secondArray,boolean paramBoolean) {
      Object localObject;
      if (paramBoolean) {
        localClass = firstArray;
        firstArray = secondArray;
        secondArray = localObject;
      }
      Class<?> localClass = firstArray.getClass().getComponentType();
      int firstArrayLength = Array.getLength(firstArray);
      int allLength = firstArrayLength + Array.getLength(secondArray);
      Object result = Array.newInstance(localClass, allLength);
      for (int k = 0; k < allLength; ++k) {
          if (k < firstArrayLength) {
              Array.set(result, k, Array.get(firstArray, k));
          } else {
              Array.set(result, k, Array.get(secondArray, k - firstArrayLength));
          }
      }
      return result;
  }
```

修复的步骤为：

1. 可以看出是通过获取到当前应用的Classloader，即为BaseDexClassloader

2. 通过反射获取到他的DexPathList属性对象pathList

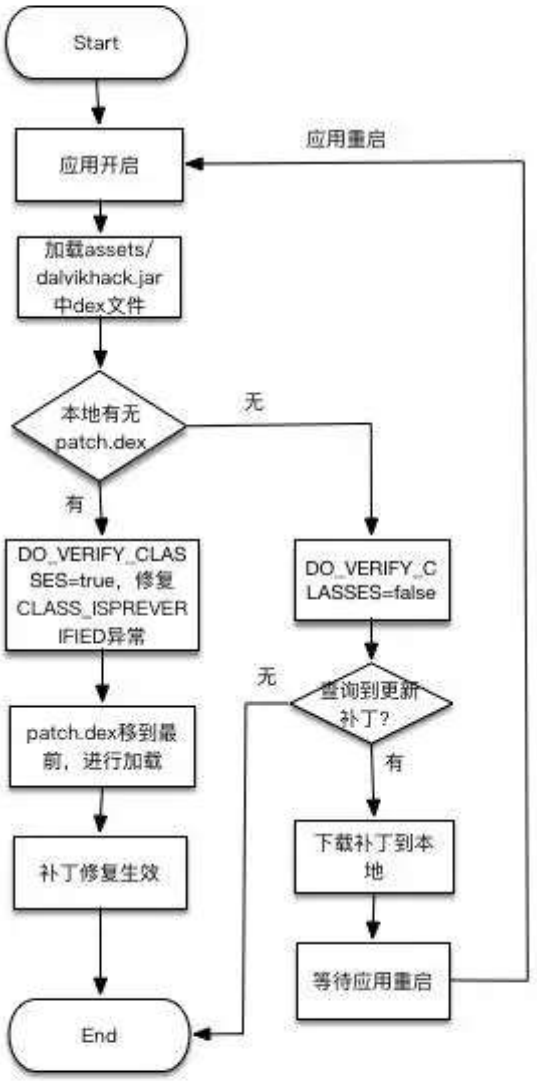
3. 通过反射调用pathList的dexElements方法把patch.dex转化为Element[]

4. 两个Element[]进行合并，把patch.dex放到最前面去

5. 加载Element[]，达到修复目的

整体的流程图如下：

手机QQ空间超级补丁技术流程图



从流程图来看，可以很明显的找到这种方式的特点：

优势：

1. 没有合成整包（和微信Tinker比起来），产物比较小，比较灵活

2. 可以实现类替换，兼容性高。（某些三星手机不起作用）

不足：

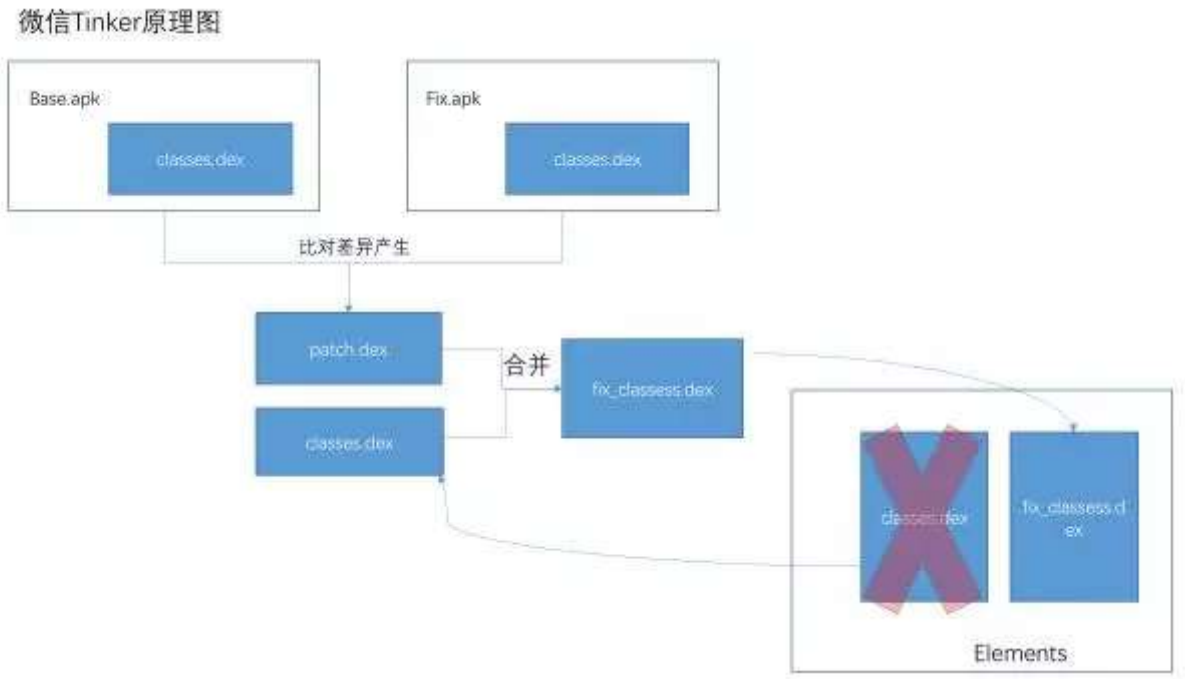
1. 不支持即时生效，必须通过重启才能生效。

2. 为了实现修复这个过程，必须在应用中加入两个dex！dalvikhack.dex中只有一个类，对性能影响不大，但是对于patch.dex来说，修复的类到了一定数量，就需要花不少的时间加载。对手淘这种航母级应用来说，启动耗时增加2s以上是不能够接受的事。

3. 在ART模式下，如果类修改了结构，就会出现内存错乱的问题。为了解决这个问题，就必须把所有相关的调用类、父类子类等等全部加载到patch.dex中，导致补丁包异常的大，进一步增加应用启动加载的时候，耗时更加严重。

二、微信Tinker

微信针对QQ空间超级补丁技术的不足提出了一个提供DEX差量包，整体替换DEX的方案。主要的原理是与QQ空间超级补丁技术基本相同，区别在于不再将patch.dex增加到elements数组中，而是差量的方式给出patch.dex，然后将patch.dex与应用的classes.dex合并，然后整体替换掉旧的DEX，达到修复的目的。



我们来逆向微信APK看一下具体的实现：

先找到应用入口TinkerApplication，在onBaseContextAttached()调用了loadTinker(),

```
private void loadTinker()
{
  if (this.tinkerFlags == 0) {
    return;
  }
  this.tinkerResultIntent = new Intent();
  try
  {
    Class localClass = Class.forName(this.loaderClassName);
    this.tinkerResultIntent = ((Intent)localClass.getMethod("tryLoad",
      new Class[] { Application.class, Integer.TYPE, Boolean.TYPE })
    .invoke(localClass.getConstructor(new Class[0]).newInstance(new Object[0]),
      new Object[] { this, Integer.valueOf(this.tinkerFlags), Boolean.valueOf(this.tinkerLoadVerifyFlag) }));
    return;
  }
  catch (Throwable localThrowable)
  {
    this.tinkerResultIntent.putExtra("intent_patch_exception", localThrowable);
  }
}
```

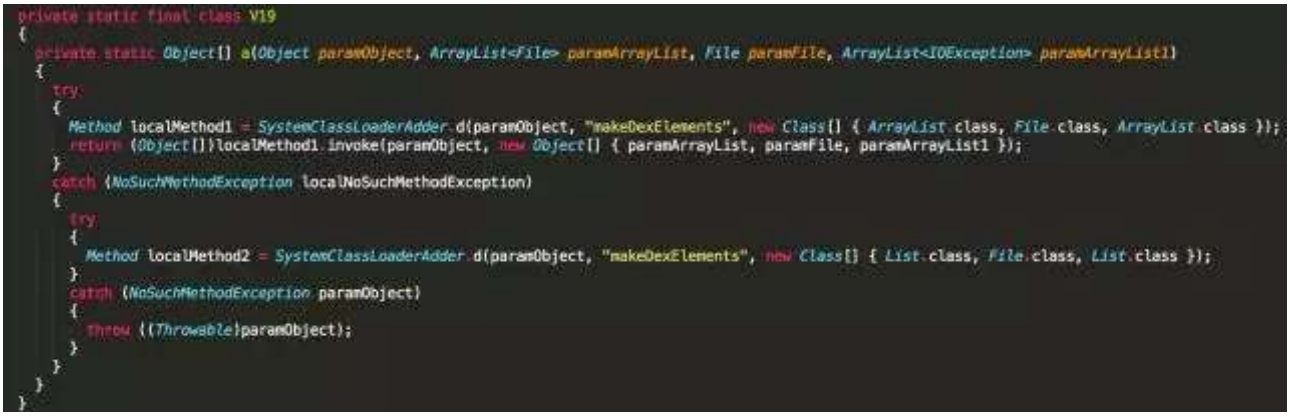
进入TinkerLoader的tryLoad()方法中，

```
public Intent tryLoad(Application paramApplication, int paramInt, boolean paramBoolean)
{
  Intent localIntent = new Intent();
  long l = SystemClock.elapsedRealtime();
  tryLoadPatchFilesInternal(paramApplication, paramInt, paramBoolean, localIntent);
  localIntent.putExtra("intent_patch_cost_time", SystemClock.elapsedRealtime() - l);
  return localIntent;
}
```

从方法名可以预见，在tryLoadPatchFilesInternal()中尝试加载本地的补丁，再经过跳转进入核心修复功能类SystemClassLoaderAdder.class中。

```
public static void a(ClassLoader paramClassLoader, File paramFile, List<File> paramList)
{
  if (!paramList.isEmpty())
  {
    if (Build.VERSION.SDK_INT >= 23) {
      V23.c(paramClassLoader, paramList, paramFile, true);
    }
  }
  else {
    return;
  }
  if (Build.VERSION.SDK_INT >= 19)
  {
    V19.b(paramClassLoader, paramList, paramFile, true);
    return;
  }
  if (Build.VERSION.SDK_INT >= 14)
  {
    V14.a(paramClassLoader, paramList, paramFile, true);
    return;
  }
  V4.d(paramClassLoader, paramList, paramFile, true);
}
```

代码中可以看出，根据Android版本的不同，分别采取具体的修复操作，不过原理都是一样的。我们以V19为例，



```
private static final class V19
{
  private static Object[] a(Object paramObject, ArrayList<File> paramArrayList, File paramFile, ArrayList<IOException> paramArrayList1)
  {
    try
    {
      Method localMethod1 = SystemClassLoaderAdder.d(paramObject, "makeDexElements", new Class[] { ArrayList.class, File.class, ArrayList.class });
      return (Object[])localMethod1.invoke(paramObject, new Object[] { paramArrayList, paramFile, paramArrayList1 });
    }
    catch (NoSuchMethodException localNoSuchMethodException)
    {
      try
      {
        Method localMethod2 = SystemClassLoaderAdder.d(paramObject, "makeDexElements", new Class[] { List.class, File.class, List.class });
      }
      catch (NoSuchMethodException paramObject)
      {
        throw ((Throwable)paramObject);
      }
    }
  }
}
```

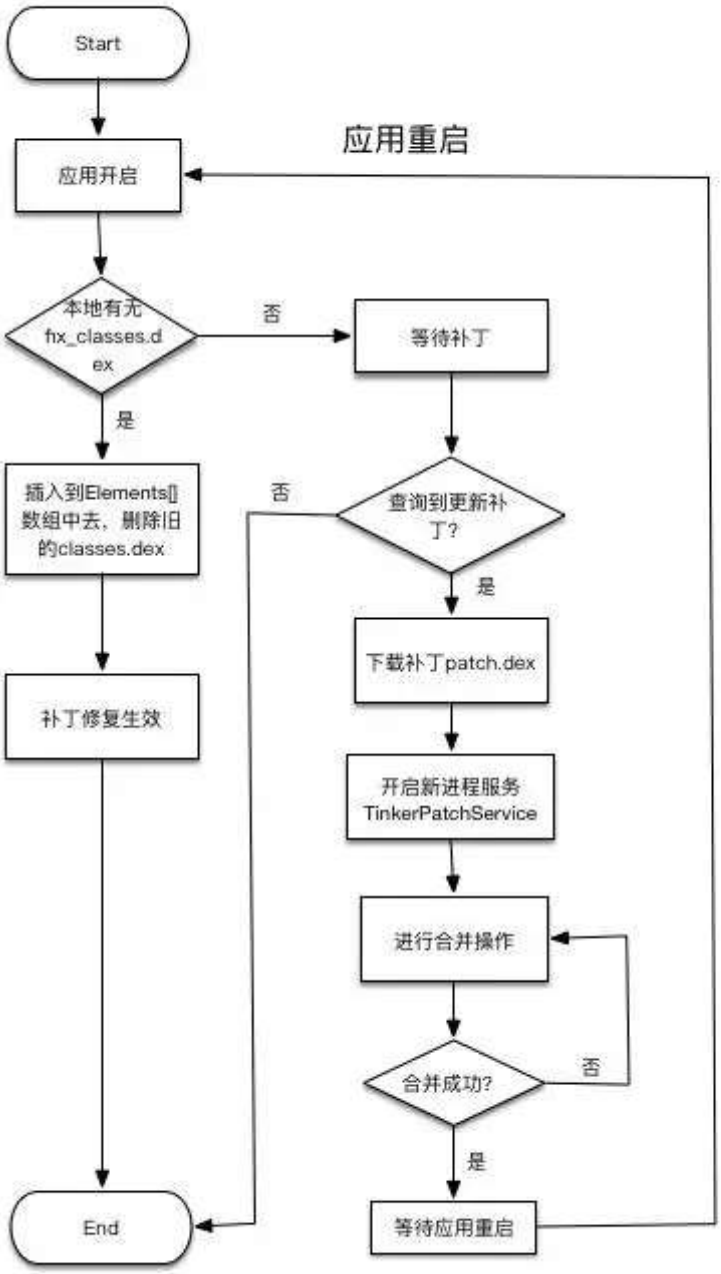
从代码中可以看到，通过反射操作得到PathClassLoader的DexPatchList,反射调用patchlist的makeDexElements()方法吧本地的dex文件直接替换到Element[]数组中去，达到修复的目的。

对于如何进行patch.dex与classes.dex的合并操作，这里微信开启了一个新的进程，开启新进程的服务TinkerPatchService进行合并。

```
public static void k(Context paramContext, String paramString, boolean paramBoolean)
{
  Intent localIntent = new Intent(paramContext, TinkerPatchService.class);
  localIntent.putExtra("patch_path_extra", paramString);
  localIntent.putExtra("patch_new_extra", paramBoolean);
  paramContext.startService(localIntent);
}
```

整体的流程如下：

微信Tinker流程图



从流程图来看，同样可以很明显的找到这种方式的特点：

优势：

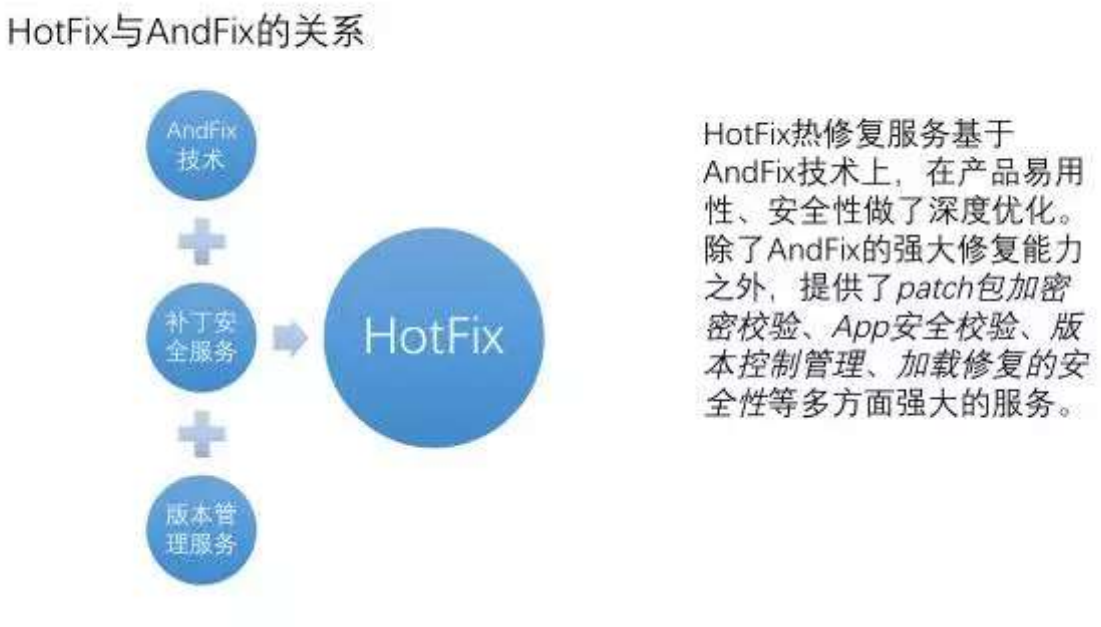
1. 合成整包，不用在构造函数插入代码，防止verify，verify和opt在编译期间就已经完成，不会在运行期间进行
2. 性能提高。兼容性和稳定性比较高。
3. 开发者透明，不需要对包进行额外处理。

不足：

1. 与超级补丁技术一样，不支持即时生效，必须通过重启应用的方式才能生效。
2. 需要给应用开启新的进程才能进行合并，并且很容易因为内存消耗等原因合并失败。
3. 合并时占用额外磁盘空间，对于多DEX的应用来说，如果修改了多个DEX文件，就需要下发多个patch.dex与对应的classes.dex进行合并操作时这种情况会更严重，因此合并过程的失败率也会更高。

三、阿里百川HotFix

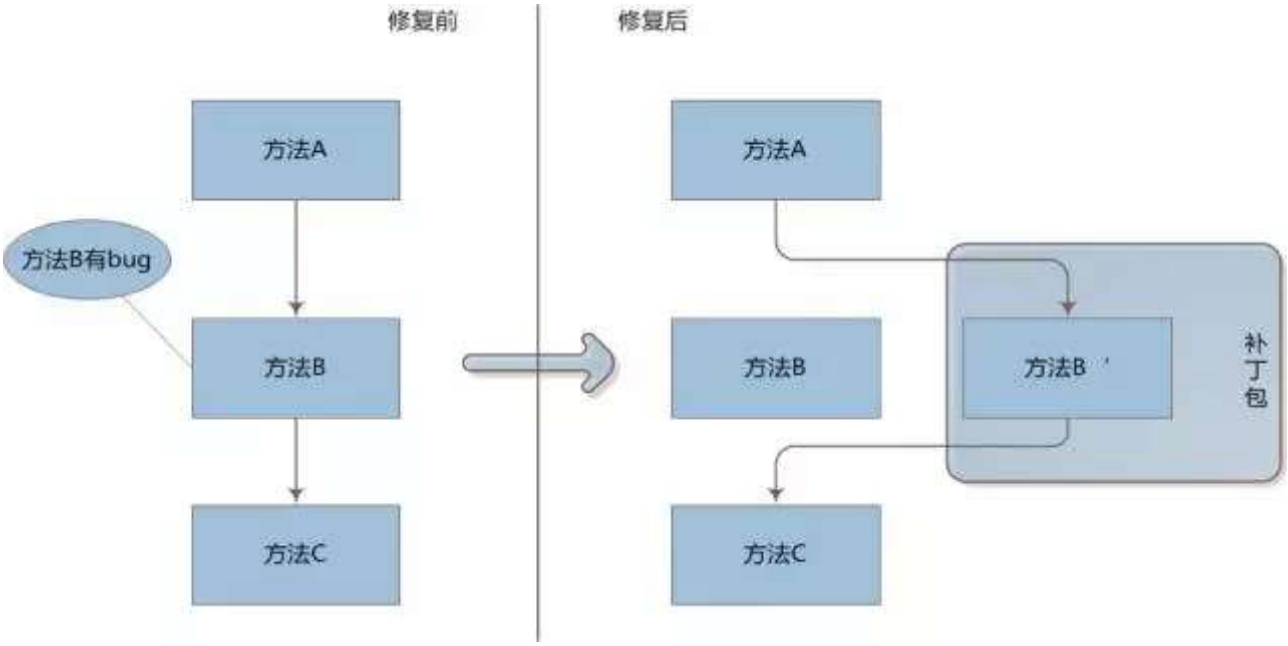
阿里百川推出的热修复HotFix服务，相对于QQ空间超级补丁技术和微信Tinker来说，定位于紧急bug修复的场景下，能够最及时的修复bug，下拉补丁立即生效无需等待。



1、AndFix实现原理

AndFix不同于QQ空间超级补丁技术和微信Tinker通过增加或替换整个DEX的方案，提供了一种运行时在Native修改Filed指针的方式，实现方法的替换，达到即时生效无需重启，对应用无性能消耗的目的。

原理图如下：



2、AndFix实现过程

对于实现方法的替换，需要在Native层操作，经过三个步骤：

接下来以Dalvik设备为例，来分析具体的实现过程：

2.1 setup()

```
extern jboolean __attribute__ ((visibility ("hidden"))) dalvik_setup(
        JNIEnv* env, int apilevel) {
    //Davik虚拟机实现 是在libdvm.so中
    //dlopen()方法以指定模式打开动态链接库，RTLD_NOW立即打开
    void* dvm_hand = dlopen("libdvm.so", RTLD_NOW);
    if (dvm_hand) {
        //dlsym:通过句柄和连接符名称获取内部函数
        dvmDecodeIndirectRef_fnPtr = dvm_dlsym(dvm_hand,
                apilevel > 10 ?
                        "_Z20dvmDecodeIndirectRefP6ThreadP8_jobject" :
                        "dvmDecodeIndirectRef");
        if (!dvmDecodeIndirectRef_fnPtr) {
            return JNI_FALSE;
        }
        dvmThreadSelf_fnPtr = dvm_dlsym(dvm_hand,
                apilevel > 10 ? "_Z13dvmThreadSelfv" : "dvmThreadSelf");
        if (!dvmThreadSelf_fnPtr) {
            return JNI_FALSE;
        }
        //
        jclass clazz = env->FindClass("java/lang/reflect/Method");
        jClassMethod = env->GetMethodID(clazz, "getDeclaringClass",
                        "()Ljava/lang/Class;");

        return JNI_TRUE;
    } else {
        return JNI_FALSE;
    }
}
```

对于Dalvik来说，遵循JIT即时编译机制，需要在运行时装载libdvm.so动态库，获取以下内部函数：

1） dvmThreadSelf( )：查询当前的线程；

2）dvmDecodeIndirectRef()：根据当前线程获得ClassObject对象。

2.2 setFieldFlag

该操作的目的：让private、protected的方法和字段可被动态库看见并识别。原因在于动态库会忽略非public属性的字段和方法。

2.3 replaceMethod

```
extern void __attribute__ ((visibility ("hidden"))) dalvik_replaceMethod(
        JNIEnv* env, jobject src, jobject dest) {
    //clazz为被替换的类
    jobject clazz = env->CallObjectMethod(dest, jClassMethod);
    //clz 为被替换的类对象
    ClassObject* clz = (ClassObject*) dvmDecodeIndirectRef_fnPtr(
            dvmThreadSelf_fnPtr(), clazz);
    //将类状态设置为初始化完毕
    clz->status = CLASS_INITIALIZED;
    //得到指向新方法的指针
    Method* meth = (Method*) env->FromReflectedMethod(src);
    //得到指向需要修复的目标方法的指针
    Method* target = (Method*) env->FromReflectedMethod(dest);

    //新方法指向目标方法，实现方法的替换
    meth->clazz = target->clazz;
    //访问权限属性public
    meth->accessFlags |= ACC_PUBLIC;
    meth->methodIndex = target->methodIndex;
    meth->jniArgInfo = target->jniArgInfo;
    meth->registersSize = target->registersSize;
    meth->outsSize = target->outsSize;
    meth->insSize = target->insSize;
    meth->prototype = target->prototype;
    //指针指向新的替换方法
    meth->insns = target->insns;
    meth->nativeFunc = target->nativeFunc;
}
```

该步骤是方法替换的核心，替换的流程如下：

AndFix对ART设备同样支持，具体的过程与Dalvik相似，这里不再赘述。

从技术原理，不难看出阿里百川HotFix的几个特点：

最大的优势在于

1. BUG修复的即时性
2. 补丁包同样采用差量技术，生成的PATCH体积小
3. 对应用无侵入，几乎无性能损耗

不足：

1. 不支持新增字段，以及修改<init>方法，也不支持对资源的替换。
2. 由于厂商的自定义ROM，对少数机型暂不支持。

综合分析如下：

	HotFix（AndFix）	微信Tinker	QQ空间补丁技术
方法替换	✓	✓	✓
即时生效	✓	×	×
Dalvik	✓	✓	✓
ART	✓	✓	✓
类、资源新增或替换	×	✓	✓
性能损耗	低，几乎无损耗	较高	高
补丁包大小	小，小至3k	较大	大
占rom大小	小	大	一般
开发透明度	高	高	高
补丁管理	控制台提供版本管理、灰度测试等功能	未开源	未开源
接入文档	丰富，傻瓜式接入	未开源	未开源

热修复的坑和解

我们可以看到，QQ空间超级补丁技术和微信Tinker的修复原理都基于类加载，在功能上已经支持类、资源的替换和新增，功能非常强大。既然已经有了这么强大的热修复技术，为什么阿里百川还要推出自己的热修复方案HotFix呢？

一、多DEX带来的性能问题和影响

我们知道，多DEX方案用来解决应用方法数65k的问题，现在Google也官方支持了MultiDex的实现方案。但是，这实在是应用因方法数超出而作出的不得已的下策，但是

超级补丁技术和Tinker作为一种热修复的方案，平生给应用增加了多个DEX，而多DEX技术最大的问题在于性能上的坑，因此基于这种方案的补丁技术影响应用的性能是无疑的。

1. 启动加载时间过长

我们可以看到，超级补丁技术和Tinker都选择在Application的attachBaseContext()进行补丁dex的加载，即使这是加载dex的最佳时机，但是依然会带来很大的性能问题，首当其冲的就是启动时间太长。

对于补丁DEX来说，应用启动时虚拟机会进行dexopt操作，将patch.dex文件转换成odex文件，这个过程非常耗时。而这个过程，又要求需要在主线程中，以同步的方式执行，否则无法成功进行修复。就DEX的加载时间，大概做了以下的时间测试。

Dex大小	10k	100k	1M	2M
加载时间	基本无增加	几十ms左右	1s左右	2s左右

随着patch.dex的增加，在不做任何优化的情况下，启动时间也直线增长。对于一个应用来说，这简直是灾难性的。

2. 易造成应用的ANR和Crash

正是尤其多DEX加载导致了启动时间过长，很容易就会引发应用的ANR。我们知道当应用在主线程等待超过5s以后，就会直接导致长时间无响应而退出。超级补丁技术为保证ART不出现地址错乱问题，需要将所有关联的类全部加入到补丁中，而微信Tinker采取一种差量包合并加载的方式，都会使要加载的dex体积变得很大。这也很大程度上容易导致ANR情况的出现。

除了应用ANR以外，多DEX模式也同样很容易导致Crash情况的出现。我们知道，超级补丁技术为了保证ART设备下不出现地址错乱问题，需要把修改类的所有相关类全部加入到补丁中，这里会出现一个问题，为了保证补丁包的体积最小，能否保证引入全部的关联类而不引入无关的类呢？一旦没有引入关联的类，就会出现以下的异常：

- NoClassDefFoundError
- Could not find class
- Could not find method

出现这些异常，就会直接导致应用的Crash退出。

所以，不难看出如果我们需要修复一个不是Crash的BUG，但是因为未加入相关类而导致了更严重的Crash，就更加的得不偿失。

总的来说，热修复本质的目的是为了保证应用更加稳定，而不是为了更强大的功能引入更大的风险和不稳定性。

二、热修复 or 插件化?

我们经常提到热修复和插件化，这都是当下热门的新兴技术。在讲述之前，需要对这两个概念进行一下解释。

- **插件化**：一个程序划分为不同的部分，以插件的形式加载到应用中去，本质上它使用的技术还是热修复技术，只是加入了更多工程实践，让它支持大规模的代码更新以及资源和SO包的更新。
- **热修复**：当线上应用出现紧急BUG，为了避免重新发版，并且保证修复的及时性而进行的一项在线推送补丁的修复方案。

显然，从概念上我们可以看到，插件化使用场景更多是功能，热修复使用常见在于修复。从这个层面来说，插件化必然功能更加强大，能做的事情也更多。QQ空间超级补丁技术和微信Tinker从类、资源的替换和更新上来看，与其说是热修复，不如说是插件化。

当然，强大的功能也就增加了不稳定的因素。比如上文提到的增加启动时间，导致ANR、Crash的问题。

QQ空间超级补丁技术和微信Tinker提供了更加强大的功能，但是对应用的性能和稳定有较大的影响，就BUG修复的这个使用场景上还不够明确，并且显得过重。

针对应用的性能损耗，我们可以举例做一个对比。

某APP的启动载入时间为3s左右，本身就是基于多DEX模式的实现。

分别接入三种热修复服务，根据腾讯提供超级补丁技术和Tinker的数据，那么会变成以下的场景：

1. 阿里百川HotFix:启动时间几乎无增加，不增加运行期额外的磁盘消耗。
2. QQ空间超级补丁技术：如果应用有700个类，启动耗时增加超过2.5s，达到5.5s以上。

3. 微信Tinker：假设应用有5个DEX文件，分别修改了这5个DEX，产生5个patch.dex文件，就要进行5次的patch合并动作，假设每个补丁1M，那么就要多占用7.5M的磁盘空间。

显然对于修复紧急BUG这个场景，阿里百川HotFix的更为合适，它更加轻量，可以在不重启的情况下生效，且对性能几乎没有影响。微信Tinker、QQ空间超级补丁技术更多地把场景定位在发布小的新功能上，采用ClassLoader的模式，牺牲较高的性能代价去实现类、资源新增或替换的功能。阿里百川HotFix对应用本身做到无侵入，无性能损耗。

总结

QQ空间超级补丁技术和微信Tinker 支持新增类和资源的替换，在一些功能化的更新上更为强大，但对应用的性能和稳定会有的一定的影响；阿里百川HotFix虽然暂时不支持新增类和资源的替换，对新功能的发布也有所限制，但是作为一项定位为线上紧急BUG的热修复的服务来说，能够真正做到BUG即时修复用户无感知，同时保证对应用性能不产生不必要的损耗，在热修复方面不失为一个好的选择。

目前阿里百川HotFix已经开始公测，复制下面的链接到浏览器打开（没办法，淘宝链接被微信屏蔽了），点击立即使用，即可开始你的热修复之旅。

http://baichuan.taobao.com/product/hotfix.htm?infoq=01