Android热更新：Tinker框架学习笔记

笔记本：	里克分享		
创建时间：	2018/3/29 17:01	**更新时间：**	2018/4/3 17:15
作者：	里克Rick_tan		

热更新两大门派：native、Dex

1、Native流派的心法口诀：替换函数

原理：将Java方法的属性设为native转到JNI层处理,在JNI中又把方法指针指向了Java Hook,在hook中回调其他Java方法,Java->Native Hook->Java Fix,最终回调到任意的目标方法.

代表：阿里 AndFix

直接使用dalvik_replaceMethod替换class中方法的实现。由于它并没有整体替换class, 而field在class中的相对地址在class加载时已确定，所以AndFix无法支持新增或者删除filed的情况(通过替换init与clinit只可以修改field的数值)。

2、Dex流派的心法口诀：替换dex

原理：一个ClassLoader可以包含多个dex文件，每个dex文件是一个Element，多个dex文件排列成一个有序的数组dexElements，当找类的时候，会按顺序遍历dex文件，然后从当前遍历的dex文件中找类，如果找类则返回，如果找不到从下一个dex文件继续查找。

代表：腾讯 Qzone、微信Tinker

热补丁技术最难的地方不是原理，不是注入dex ，而是字节码的注入。

一、Qzone方案

一个严重的问题： unexpected DEX problem

问题现场还原：ModuleManager引用了QzoneActivityManager，两个类都在classes.dex里面，在安装在Dalvik机器的过程中，会经历dex优化过程，这个过程会进行类的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。这个时候ModuleManager被打上了CLASS_ISPREVERIFIED标识。后来因修复问题，将QzoneActivityManager patch到了补丁 patch.dex里面，在加载QzoneActivityManager的时候，对QzoneActivityManager和ModuleManager进行dex校验，发现相关联的类在不同的dex中，于是系统就抛出了问题....

问题跟进记录：

当一个apk在安装的时候，apk中的classes.dex会被虚拟机(dexopt)优化成odex文件，然后才会拿去执行。 虚拟机在启动的时候，会有许多的启动参数，其中一项就是verify选项，当verify选项被打开的时候，上面doVerify变量为true，那么就会执行dvmVerifyClass进行类的校验，如果dvmVerifyClass校验类成功，那么这个类会被打上CLASS_ISPREVERIFIED的标志。源码： **DexPrepare.cpp**、DexVerify.cpp、...

简单说：以下情况会打上CLASS_ISPREVERIFIED这个标记：

1. static方法
2. private方法
3. 构造函数
4. 虚函数

如果QzoneActivityManager.class在以上方法中被引用时，并且QzoneActivityManager.class和引用QzoneActivityManager.class的类（比如：ModuleManager.class）处于同一个dex文件时【并且QzoneActivityManager没有引用其他dex类】，就会打上那个标记。后面的问题...（就是上面的描述的）

问题解决 - 插桩方案：

向所有类的构造函数中插入了以下代码：

```
if (ClassVerifier.PREVENT_VERIFY) {
    System.out.println(AntilazyLoad.class);
}
```

并且，将AntilazyLoad.class放入一个单独的dex文件里。【因此就不会有类打上CLASS_ISPREVERIFIED标签】

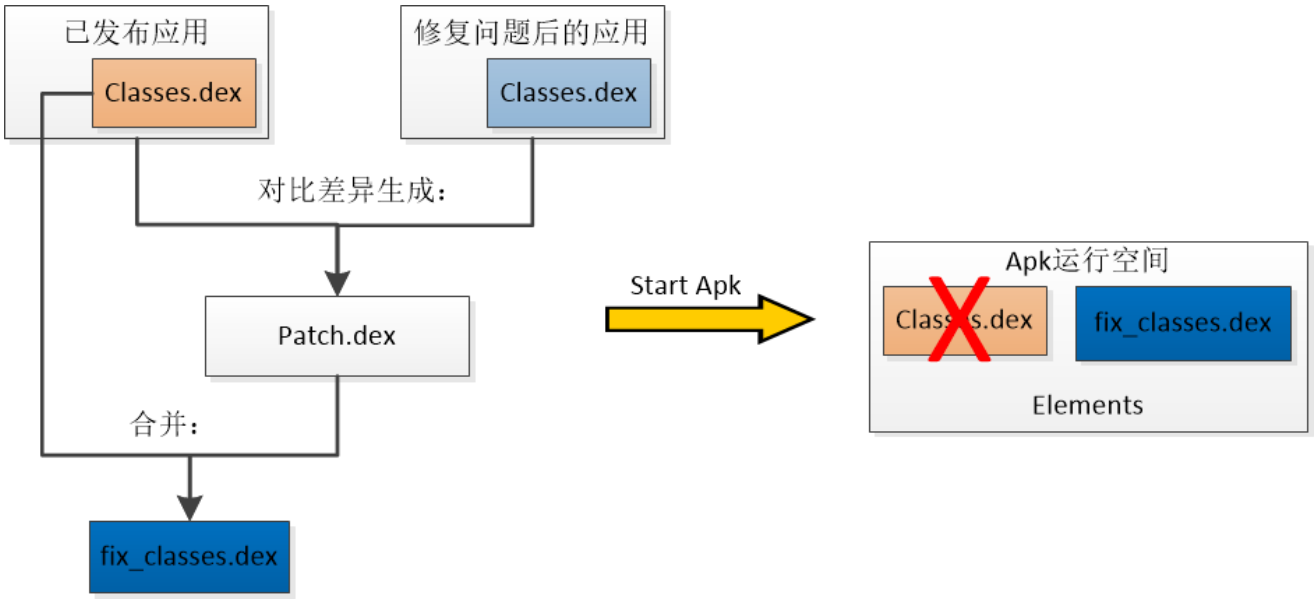
带来的问题：

补丁包大小与性能损耗上有一定的局限性

程序运行时的性能产生影响

二、Tinker原理

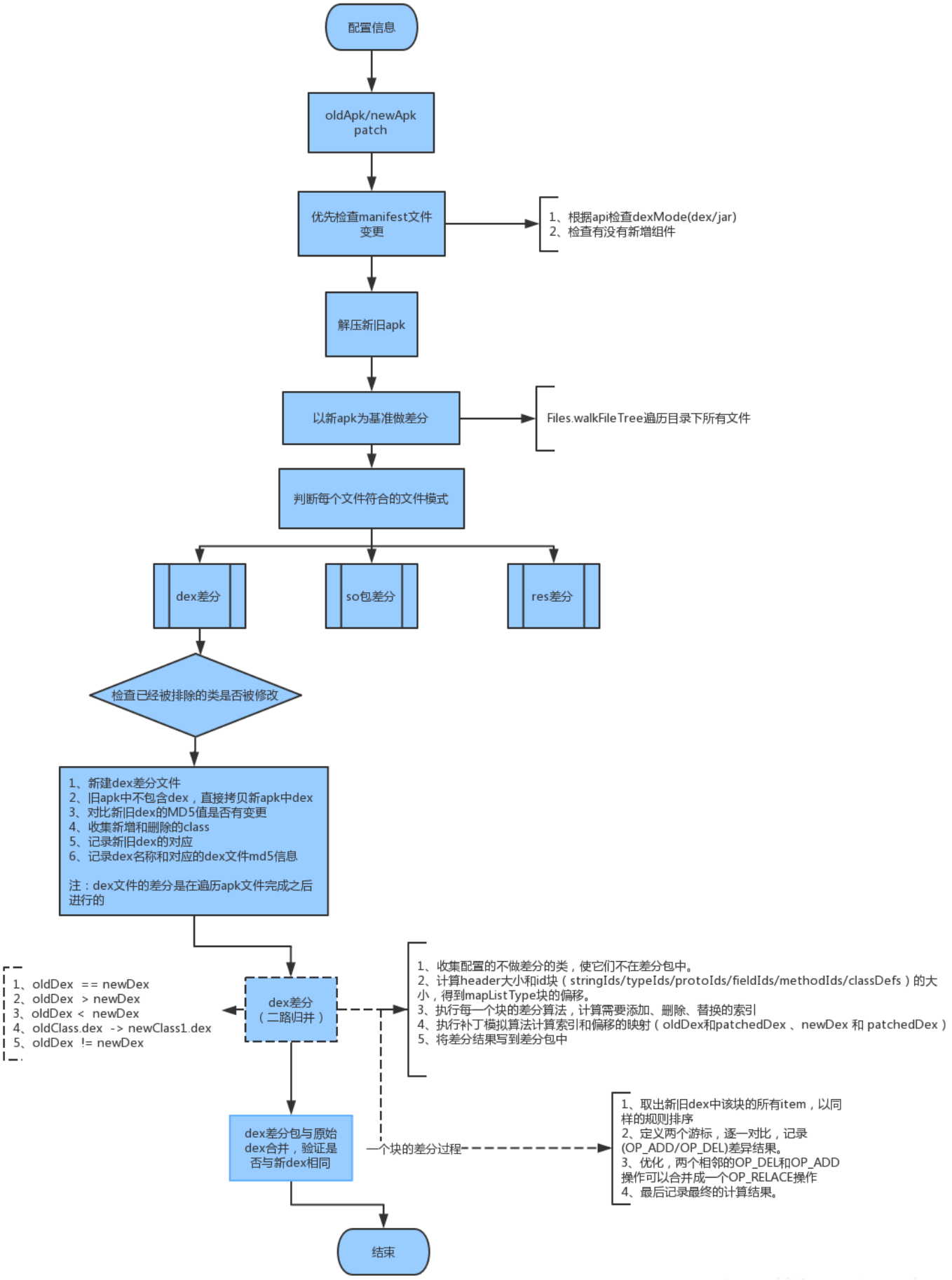
核心思想：**利用DexDiff算法对比差异生成Patch补丁包，分平台合成，全量替换新的Dex。**



三、Tinker重要流程

按过程分三个阶段：1、构建补丁包；2、重新合成 - 补丁包合并；3、加载

1、构建补丁包



Tinker框架提供了一套gradle插件，包括了他们自己研究的一套用于生成补丁的DexDiff算法。

Tinker和以往的HotFix库思路不太一样，它更像是APP的增量更新，在服务器端通过差异性算法，计算出新旧dex之间的差异包，推送到客户端，进行合成。传统的差异性算法有BsDiff，而Tinker的牛逼之处就在于它自己基于Dex的文件格式，研发出了DexDiff算法。

补丁生成是在编译阶段进行的，调用链为：

（Gradle）tinkerPatchRelease
TinkerPatchSchemaTask.tinkerPatch()
->Runner.gradleRun()
->Runner.run()
->Runner.tinkerPatch()
->ApkDecoder.patch()　　// 会先对manifest文件进行检测，看其是否有更改，如果发现manifest的组件有新增，则抛出异常，因为目前Tinker暂不支持四大组件的新增。
// 检测通过后解压apk文件，遍历新旧apk，交给ApkFilesVisitor进行处理，

// ApkFilesVisitor的visitFile函数中，对于dex类型的文件，调用dexDecoder进行patch操作；对于so类型的文件，使用soDecoder进行patch操作；对于Res类型文件，使用resDecoder进行操作。

->FileVisitResult.visitFile //下面针对dex来阐述

->DexDiffDecoder.patch　// 首先检测输入的dex文件中是否有不允许修改的类被修改了，如loader相关的类是不允许被修改的，这种情况下会抛出异常； 如果dex是新增的，直接将该dex拷贝到结果文件； 如果dex是修改的，收集增加和删除的class。oldAndNewDexFilePairList将新旧dex对应关系保存起来，用于后面的分析。

->UniqueDexDiffDecoder.patch　// 将新的dex文件加入到addedDexFiles

->DexFiffDecoder.generatePatchInfoFile //首先遍历oldAndNewDexFilePairList，取出新旧文件对。判断新旧文件的MD5是否相等，不相等，说明有变化，会根据新旧文件创建DexPatchGenerator，

->DexPatchGenerator.executeAndSaveTo()　　// 根据上面DexPatchGenerator提供的15个Dex区域的比较算法对dex的各个区域进行比较，最后生成dex文件的差异，

executeAndSaveTo()方法是DexDiff算法的真正入口，DexDiff算法的特点在于它深入分析了Dex文件格式，深度利用Dex的格式来减少差异大小。

【注意】： Tinker在生成补丁阶段会生成一个test.dex，这个test.dex的作用就是用来验证dex的加载是否成功。test.dex中含有com.tencent.tinker.loader.TinkerTestDexLoad类，该类中包含一个字段isPatch，checkDexInstall就是通过findField该字段判断是否加载成功。

DexDiff的主要步骤如下：

Step1:计算出new dex中每项Section的大小，比如string_ids在dex文件中所占大小。

```
int patchedStringIdsSize = newDex.getTableOfContents().stringIds.size * SizeOf.STRING_ID_ITEM;
```

step2:根据表中前一项的偏移地址和大小，计算出每项Section的偏移地址。

```
this.patchedStringIdsOffset = patchedHeaderOffset + patchedheaderSize;
```

step3:调用DexSectionDiffAlgorithm.execute()，将new dex与old dex中的每项section进行对比，对于每项Section，遍历其每一项Item，进行新旧对比，记录ADD，DEL标识，存放于patchOperationList中。接着遍历patchOperationList，添加REPLACE标识，最后将ADD，DEL，REPLACE操作分别记录到各自的List中。

step4:调用DexPatchGenerator.writePatchOperations()，将记录写入补丁。

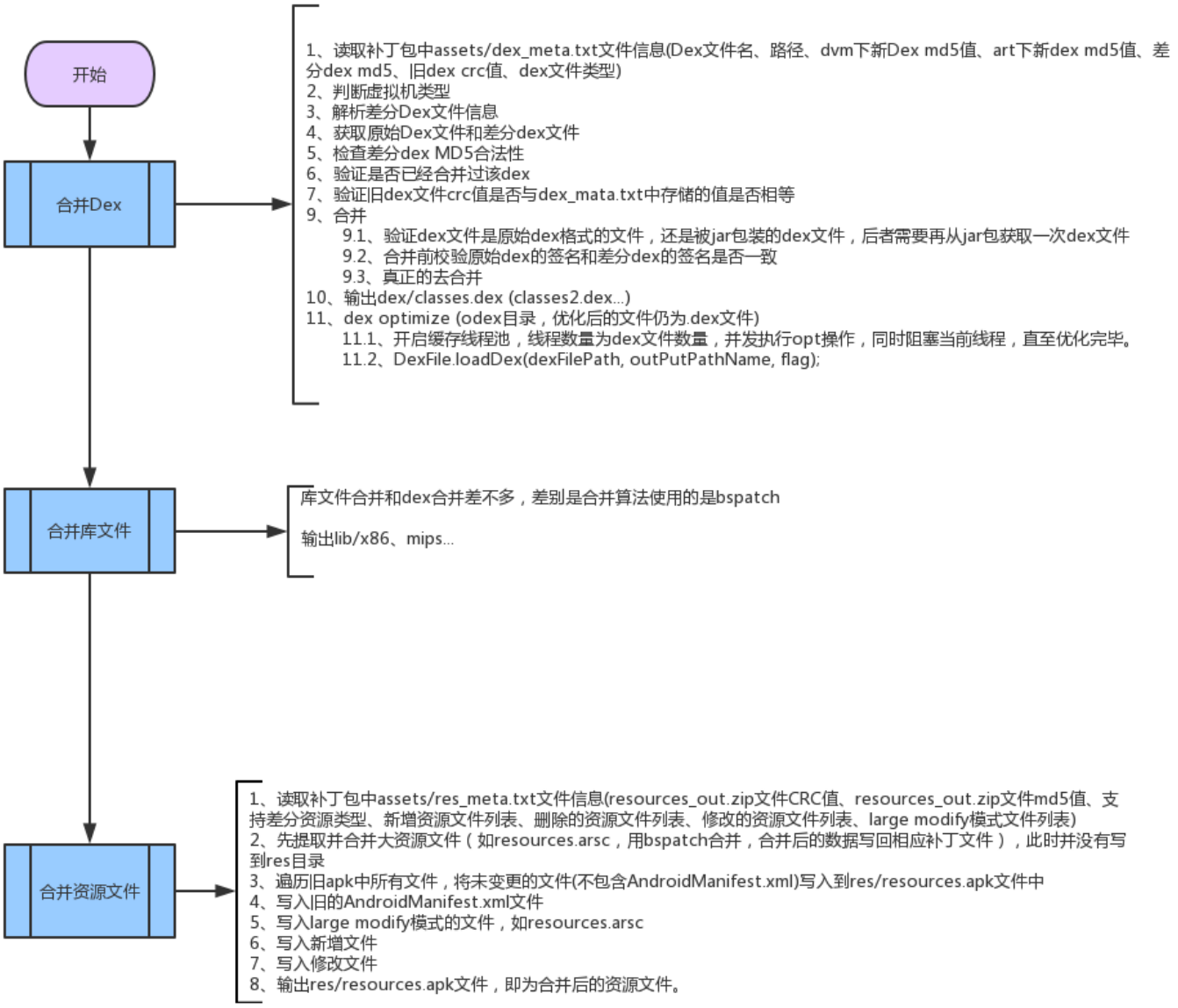
参考：

Android 热修复 Tinker 源码分析之DexDiff / DexPatch
Tinker Dexdiff算法解析

2、重新合成 - 补丁包合并

app收到服务器下发的补丁后，会触发**DefaultPatchListener.onPatchReceived**事件，

调用**TinkerPatchService.runPatchService**启动**patch**进程进行补丁**patch**工作。
[流程如下**]**：
-> TinkerPatchService.runPatchService //启动patch进程进行补丁patch工作
-> UpgradePatch.tryPatch() // 先检查补丁的合法性，签名，以及是否安装过补丁，检查通过后会尝试dex，so以及res文件的patch
-> DexDiffPatchInternal.tryRecoverDexFiles // 调用DexDiffPatchInternal.patchDexFile, 最终通过DexPatchApplier.executeAndSaveTo进行执行及生产全量dex
-> DexPatchApplier.executeAndSaveTo // 会对15个dex区域进行patch操作，针对old dex和patch dex进行合并,生成全量dex文件。
//每个区域的合并算法采用二路归并，在old dex的基础上对元素进行删除，增加，替换操作。这里的算法和生成补丁的DexDiff是一个逆向的过程。
-> TinkerParallelDexOptimizer.optimizeAll // 在extractDexDiffInternals调用完以后, 会调用TinkerParallelDexOptimizer.optimizeAll对生成的全量dex进行optimize操作，生成odex文件, 最终合成的文件会放到/data/data/${package_name}/tinker目录下



微信团队最后实现下面这一套方案，这也是**其他全量合成方案所不能做到的**:

- Dalvik全量合成，解决了插桩带来的性能损耗;
- Art平台合成small dex，解决了全量合成方案占用Rom体积大，OTA升级以及Android N的问题;
- 大部分情况下Art.info仅仅1-20K，解决由于补丁包可能过大的问题;

补丁合成入口:

```
TinkerInstaller.onReceiveUpgradePatch(getApplicationContext(), path);
```

3、加载

在TinkerApplication【通过反射的方式将实际的app业务隔离，这样可以在热更新的时候修改实际的app内容】中的onBaseContextAttached中会通过反射调用TinkerLoader的tryLoad加载已经合成的dex。

```
-> TinkerApplication. onBaseContextAttached     // 通过反射调用TinkerLoader的tryLoad加载已经合成的dex
-> TinkerLoader.tryLoad()     //调用tryLoadPatchFilesInternal
-> TinkerLoader.tryLoadPatchFilesInternal // 加载Patch文件的核心函数，  主要做了以下的事情:
```

- tinkerFlag是否开启，否则不加载
- tinker目录是否生成，没有则表示没有生成全量的dex，不需要重新加载
- tinker/patch.info是否存在，否则不加载

- 读取patch.info，读取失败则不加载
- 比较patchInfo的新旧版本，都为空则不加载
- 判断版本号是否为空，为空则不加载
- 判断patch version directory (//tinker/patch.info/patch-641e634c) 是否存在
- 判断patchVersionDirectoryFile (//tinker/patch.info/patch-641e634c/patch-641e634c.apk) 是否存在
- checkTinkerPackage，(如tinkerId和oldTinkerId不能相等，否则不加载)
- 检测dex的完整性，包括dex是否全部生产，是否对dex做了优化，优化后的文件是否存在 (//tinker/patch.info/patch-641e634c/dex)
- 同样对so res文件进行完整性检测
- 尝试超过3次不加载
- loadTinkerJars/loadTinkerResources/

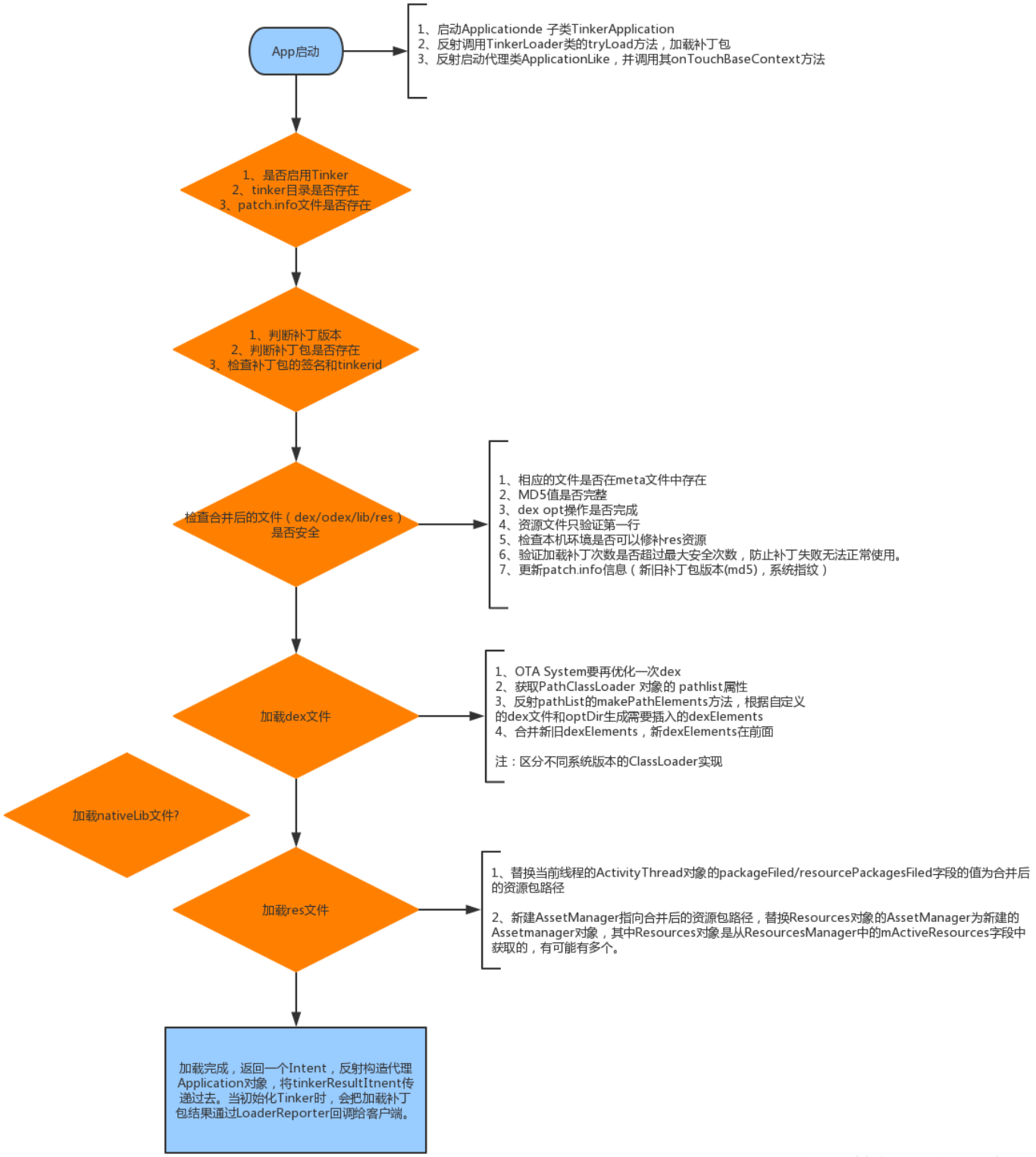
```
-> TinkerDexLoader.loadTinkerJars          // 处理加载dex文件。
-> SystemClassLoaderAdder.installDexes     // 按照安卓的版本对dex进行install
```

- install的做法就是，先获取BaseDexClassLoader的dexPathList对象，然后通过dexPathList的makeDexElements函数将我们要安装的dex转化成Element[]对象， 最后将其和dexPathList的dexElements对象进行合并，就是新的Element[]对象。
- 以V19的install为例：

```
private static final class V19 {

    private static void install(ClassLoader loader, List<File> additionalClassPathEntries,
                                File optimizedDirectory)
        throws IllegalArgumentException, IllegalAccessException,
        NoSuchFieldException, InvocationTargetException, NoSuchMethodException, IOException {
        /* The patched class loader is expected to be a descendant of
         * dalvik.system.BaseDexClassLoader. We modify its
         * dalvik.system.DexPathList pathList field to append additional DEX
         * file entries.
         */

        //V19.install()中先通过反射获取BaseDexClassLoader中的dexPathList,然后调用了ShareReflectUtil.expandFieldArray().
        // 值得一提的是微信对异常的处理很细致,用List<IOException>接收dexElements数组中每一个dex加载抛出的异常而不是笼统的抛出一个大异常.
        Field pathListField = ShareReflectUtil.findField(loader, name: "pathList");
        Object dexPathList = pathListField.get(loader);
        ArrayList<IOException> suppressedExceptions = new ArrayList<~>();

        //不要被它的注释误导了,这里不是替换普通的Field,调用这个方法的入参fieldName正是上一步中的"dexElements",
        // 在这么不起眼的一个工具类中终于找到了Dex流派的核心方法
        ShareReflectUtil.expandFieldArray(dexPathList, fieldName: "dexElements", makeDexElements(dexPathList,
            new ArrayList<File>(additionalClassPathEntries), optimizedDirectory,
            suppressedExceptions));
        if (suppressedExceptions.size() > 0) {
            for (IOException e : suppressedExceptions) {
                Log.w(TAG, msg: "Exception in makeDexElement", e);
                throw e;
            }
        }
    }
```

不同android版本里面的DexPathList等类的函数和字段都有一些变化，因此要在install的时候需要对不同版本进行适配。

因为我们添加的dex都被放在dexElements数组的最前面，所以当通过findClass来查找这个类时，就是使用的我们最新的dex里面的类。



Tinker虽然原理不变,但它也有拿得出手的重大优化:传统的插桩步骤会导致第一次加载类时耗时变长.应用启动时通常会加载大量类,所以对启动时间的影响很可观.Tinker的亮点是通过全量替换dex的方式避免unexpectedDEX,这样做所有的类自然都在同一个dex中.但这会带来补丁包dex过大的问题,由此微信自研了DexDiff算法来取代传统的BsDiff,极大降低了补丁包大小,又规避了运行性能问题又减小了补丁包大小,可以说是Dex流派的一大进步.

一、Tinker的优势

当前市面的热补丁方案有很多，其中比较出名的有阿里的AndFix、美团的Robust以及QZone的超级补丁方案。但它们都存在无法解决的问题，这也是正是我们推出Tinker的原因。

	Tinker	QZone	阿里的AndFix	美团的Robust
类替换	yes	yes	no	no
So替换	yes	no	no	no
资源替换	yes	yes	no	no
全平台支持	yes	yes	yes	yes
即时生效	no	no	yes	yes
性能损耗	较小	较大	较小	较小
补丁包大小	较小	较大	一般	一般
开发透明	yes	yes	no	no
复杂度	较低	较低	复杂	复杂
gradle支持	yes	no	no	no
Rom体积	较大	较小	较小	较小
成功率	较高	较高	一般	最高

Tinker的已知问题：

1.Tinker不支持修改AndroidMainfest.xml，Tinker不支持新增四大组件。
2.由于Google Pay的开发者条款限制，不建议在GP渠道动态更新代码。
3.在Android N上，补丁对应用启动时有轻微的影响。
4.不支持部分三星android-21机型，加载补丁时会主动抛出“TinkerRuntimeException:checkDexInstall failed”异常。
5.由于各个厂商加固实现并不一致，在1.7.6以后的版本，Tinker不在支持加固的动态更新。
6.对于资源替换，不支持修改remoteView，例如transition动画，notification icon以及桌面图标。

总的来说：

1. AndFix作为native解决方案，首先面临的是稳定性与兼容性问题，更重要的是它无法实现类替换，它是需要大量额外的开发成本的；
2. Robust兼容性与成功率较高，但是它与AndFix一样，无法新增变量与类只能用做的bugFix方案；
3. Qzone方案可以做到发布产品功能，但是它主要问题是插桩带来Dalvik的性能问题，以及为了解决Art下内存地址问题而导致补丁包急速增大的。

【Tinker 官方github接入指南】

二、gradle参数详解

我们将原apk包称为基准apk包，tinkerPatch直接使用基准apk包与新编译出来的apk包做差异，得到最终的补丁包。
gradle配置的参数详细解释如下：

参数	默认值	描述

`tinkerPatch`		全局信息相关的配置项
tinkerEnable	true	是否打开tinker的功能。
oldApk	null	基准apk包的路径，必须输入，否则会报错。
newApk	null	选填，用于编译补丁apk路径。如果路径合法，即不再编译新的安装包，使用oldApk与newApk直接编译。
outputFolder null	选填，设置编译输出路径。默认在`build/outputs/tinkerPatch`中	
ignoreWarning	false	如果出现以下的情况，并且ignoreWarning为false，我们将中断编译。因为这些情况可能会导致编译出来的patch包带来风险： 1. minSdkVersion小于14，但是`dexMode`的值为"raw"; 2. 新编译的安装包出现新增的四大组件(Activity, BroadcastReceiver...); 3. 定义在dex.loader用于加载补丁的类不在main dex中; 4. 定义在dex.loader用于加载补丁的类出现修改; 5. resources.arsc改变，但没有使用applyResourceMapping编译。
useSign	true	在运行过程中，我们需要验证基准apk包与补丁包的签名是否一致，我们是否需要为你签名。
`buildConfig`		编译相关的配置项
applyMapping	null	可选参数；在编译新的apk时候，我们希望通过保持旧apk的proguard混淆方式，从而减少补丁包的大小。这个只是推荐设置，不设置`applyMapping`也不会影响任何的`assemble`编译。
applyResourceMapping	null	可选参数；在编译新的apk时候，我们希望通过旧apk的`R.txt`文件保持ResId的分配，这样不仅可以减少补丁包的大小，同时也避免由于`ResId`改变导致`remote view`异常。
tinkerId	null	在运行过程中，我们需要验证基准apk包的tinkerId是否等于补丁包的tinkerId。这个是决定补丁包能运行在哪些基准包上面，一般来说我们可以使用git版本号、versionName等等。
keepDexApply	false	如果我们有多个dex,编译补丁时可能会由于类的移动导致变更增多。若打开`keepDexApply`模式，补丁包将根据基准包的类分布来编译。
isProtectedApp	false	是否使用加固模式，仅仅将变更的类合成补丁。**注意，这种模式仅仅可以用于加固应用中。**
supportHotplugComponent(**added 1.9.0**)	false	是否支持新增非export的Activity

dex		dex相关的配置项
dexMode	jar	只能是'raw'或者'jar'。 对于'raw'模式，我们将会保持输入dex的格式。 对于'jar'模式，我们将会把输入dex重新压缩封装到jar。如果你的minSdkVersion小于14，你必须选择‘jar’模式，而且它更省存储空间，但是验证md5时比'raw'模式耗时。默认我们并不会去校验md5,一般情况下选择jar模式即可。
pattern	[]	需要处理dex路径，支持*、?通配符，必须使用'/'分割。路径是相对安装包的，例如assets/...
loader	[]	这一项非常重要，它定义了哪些类在加载补丁包的时候会用到。这些类是通过Tinker无法修改的类，也是一定要放在main dex的类。 这里需要定义的类有： 1．你自己定义的Application类； 2．Tinker库中用于加载补丁包的部分类，即com.tencent.tinker.loader.*； 3．如果你自定义了TinkerLoader，需要将它以及它引用的所有类也加入loader中； 4．其他一些你不希望被更改的类，例如Sample中的BaseBuildInfo类。**这里需要注意的是，这些类的直接引用类也需要加入到loader中。或者你需要将这个类变成非preverify。** 5．**使用1.7.6版本之后的gradle版本，参数1、2会自动填写。若使用newApk或者命令行版本编译，1、2依然需要手动填写**
lib		lib相关的配置项
pattern	[]	需要处理lib路径，支持*、?通配符，必须使用'/'分割。与dex.pattern一致，路径是相对安装包的，例如assets/...
res		res相关的配置项
pattern	[]	需要处理res路径，支持*、?通配符，必须使用'/'分割。与dex.pattern一致，路径是相对安装包的，例如assets/...，务必注意的是，只有满足pattern的资源才会放到合成后的资源包。
ignoreChange	[]	支持*、?通配符，必须使用'/'分割。若满足ignoreChange的pattern，在编译时会忽略该文件的新增、删除与修改。 **最极端的情况，ignoreChange与上面的pattern一致，即会完全忽略所有资源的修改。**
largeModSize	100	对于修改的资源，如果大于largeModSize，我们将使用bsdiff算法。这可以降低补丁包的大小，但是会增加合成时的复杂度。默认大小为100kb
packageConfig		用于生成补丁包中的'package_meta.txt'文件

configField	TINKER_ID, NEW_TINKER_ID	configField("key", "value"), 默认我们自动从基准安装包与新安装包的Manifest中读取tinkerId,并自动写入configField。在这里，你可以定义其他的信息，在运行时可以通过TinkerLoadResult.getPackageConfigByName得到相应的数值。但是建议直接通过修改代码来实现，例如BuildConfig。
sevenZip		7zip路径配置项，执行前提是useSign为true
zipArtifact	null	例如"com.tencent.mm:SevenZip:1.1.10"，将自动根据机器属性获得对应的7za运行文件，推荐使用。
path	7za	系统中的7za路径，例如"/usr/local/bin/7za"。path设置会覆盖zipArtifact，若都不设置，将直接使用7za去尝试。

具体的参数设置事例可参考sample中的app/build.gradle。