#### 腾讯-热修复连环炮(热修复是什么 有接触过tinker吗，tinker原理是什么)

##### 热修复是什么

答：

热修复无疑是这2年较火的新技术，是作为安卓工程师必学的技能之一。在热修复出现之前，一个已经上线的app中如果出现了bug，即使是一个非常小的bug，不及时更新的话有可能存在风险，若要及时更新就得将app重新打包发布到应用市场后，让用户再一次下载，这样就大大降低了用户体验，当热修复出现之后，这样的问题就不再是问题了。

目前较火的热修复方案大致分为两派，分别是：

1. 阿里系：spohix、andfix：从底层二进制入手（c语言）。
2. 腾讯系：tinker：从java加载机制入手。

##### 有接触过tinker吗

答: 有接触过Tinker的 Tinker是一个比较优异修复架构

##### 修复的原理是什么

答: 关于bug的概念自己百度百科吧，我认为的bug一般有2种（可能不太准确）：

* 代码功能不符合项目预期，即代码逻辑有问题。
* 程序代码不够健壮导致App运行时崩溃。

这两种情况一般是一个或多个class出现了问题，在一个理想的状态下，我们只需将修复好的这些个class更新到用户手机上的app中就可以修复这些bug了。但说着简单，要怎么才能动态更新这些class呢？其实，不管是哪种热修复方案，肯定是如下几个步骤：

1. 下发补丁（内含修复好的class）到用户手机，即让app从服务器上下载（网络传输）
2. app通过**"某种方式"**，使补丁中的class被app调用（本地更新）

这里的**"某种方式"**，对本篇而言，就是使用Android的类加载器，通过类加载器加载这些修复好的class，覆盖对应有问题的class，理论上就能修复bug了。所以，下面就先来了解和分析Android中的类加载器吧。

#### Tinker源码分析

##### Tinker工程结构

直接从github上clone Tinker的源码进行食用如下：

img

img

##### 接入流程

1. gradle相关配置主项目中build.gradle加入

buildscript {
 dependencies {
 classpath ('com.tencent.tinker:tinker-patch-gradle-plugin:1.8.1')
 }
}

在app工程中build.gradle加入

dependencies {
 //可选，用于生成application类
 provided('com.tencent.tinker:tinker-android-anno:1.8.1')
 //tinker的核心库
 compile('com.tencent.tinker:tinker-android-lib:1.8.1')
}
...
...
//apply tinker插件
apply plugin: 'com.tencent.tinker.patch'

这里需要注意tinker编译阶段会判断一个TinkerId的字段，该字段默认由git提交记录生成HEAD(git rev-parse --short HEAD)而且是在rootproject中执行的git命令,所以个别工程可能在rootproject目录没有git init过，可以选择在那初始化git或者自定义gradle修改gitSha方法。

出包还是使用正常的build过程，测试阶段选择assembleDebug，Tinker产出patch使用gradle tinkerPatchDebug同样也支持Flavor和Variant，Tiner会在主工程build目录下创建bakApk，下面会有一个app-yydd-hh-mm-ss的目录里面对应有Favor子目录里面包含了通过assemble出的apk包。在build目录下的outputs中有tinkerPatch里面同样也区分了build variant产物。

img

img

image

*需要注意的是在debug出包测试过程中需要修改gradle的参数*

ext {
 //for some reason, you may want to ignore tinkerBuild, such as instant run debug build?
 tinkerEnabled = true

 //for normal build
 //old apk file to build patch apk
 tinkerOldApkPath = "${bakPath}/app-debug-1018-17-58-54.apk"
 //proguard mapping file to build patch apk
 tinkerApplyMappingPath = "${bakPath}/app-debug-1018-17-32-47-mapping.txt"
 //resource R.txt to build patch apk, must input if there is resource changed
 tinkerApplyResourcePath = "${bakPath}/app-debug-1018-17-32-47-R.txt"

 //使用buildvariants修改此处app信息作为基准包
 tinkerBuildFlavorDirectory = "${bakPath}/app-1020-11-52-37"
}

而release出包可以直接在gradle命令带上后缀-POLD\_APK= -PAPPLY\_MAPPING= -PAPPLY\_RESOURCE=

1. Application改造

Tinker采用了代码框架的方案来解决应用启动加载默认Application导致patch无法修复它。原理就是使用一个ApplicationLike代理类来完成原Application的功能，把所有原理Application中的代码逻辑移动到ApplicationLike中，然后删除原来的Application类通过注解让Tinker自动生成默认Application。

@DefaultLifeCycle(application = "com.\*.Application",
 flags = ShareConstants.TINKER\_ENABLE\_ALL,
 loadVerifyFlag = false)
public class ApplicationLike extends DefaultApplicationLike {
 @Override
 public void onBaseContextAttached(Context base) {
 super.onBaseContextAttached(base);
 //you must install multiDex whatever tinker is installed!
 MultiDex.install(base);

 TinkerManager.setTinkerApplicationLike(this);

 TinkerManager.initFastCrashProtect();
 //should set before tinker is installed
 TinkerManager.setUpgradeRetryEnable(true);

 //installTinker after load multiDex
 //or you can put com.tencent.tinker.\*\* to main dex
 TinkerManager.installTinker(this);
 }

}

##### TinkerManager.java

public static void installTinker(ApplicationLike appLike) {
 if (isInstalled) {
 TinkerLog.w(TAG, "install tinker, but has installed, ignore");
 return;
 }
 //or you can just use DefaultLoadReporter
 LoadReporter loadReporter = new TinkerLoadReporter(appLike.getApplication());
 //or you can just use DefaultPatchReporter
 PatchReporter patchReporter = new TinkerPatchReporter(appLike.getApplication());
 //or you can just use DefaultPatchListener
 PatchListener patchListener = new TinkerPatchListener(appLike.getApplication());
 //you can set your own upgrade patch if you need
 AbstractPatch upgradePatchProcessor = new UpgradePatch();

 TinkerInstaller.install(appLike,
 loadReporter, patchReporter, patchListener,
 TinkerResultService.class, upgradePatchProcessor);

 isInstalled = true;
 }

其中参数application代表自动生成的application包名路径，flags代表tinker作用域包括res、so、dex，loadVerifyFlag代表是否开启加载patch前各个文件进行md5校验,还有一个loaderClass默认是"com.tencent.tinker.loader.TinkerLoader"表示加载Tinker的主类名。

在onBaseContextAttached方法里需要初始化一些Tinker相关回调(在installTinker方法中)PatchReporter是对patch进程中合成过程的回调接口实现，LoadReporter是对主进程加载patch dex补丁过程的回调接口实现。PatchListener可以对接收到patch补丁后做自定义的check操作比如渠道检查和存储空间检查。

设置AbstractResultService的实现类TinkerResultService作为合成补丁完成后的处理重启逻辑的IntentService。

设置AbstractPatch的实现类UpgradePatch类作为合成patch方法tryPatch实现类。

## Tinker原理

先上github官方首页的图

img

img

image

BaseApk就是我们的基准包，也就是渠道上线的包。

NewApk就是我们的hotfix包，包括修复的代码资源以及so文件。

Tinker做了对应的DexDiff、ResDiff、BsDiff来产出一个patch.apk,里面具体内容也是由lib、res和dex文件组成，assets中还有对应的dex、res和so信息

img

img

image

然后Tinker通过找到基准包data/app/packagename/base.apk通过DexPatch合成新的dex，并且合成一个tinker\_classN.apk(其实就是包含了所有合成dex的zip包)接着在运行时通过反射把这个合成dex文件插入到PathClassLoader中的dexElements数组的前面，保证类加载时优先加载补丁dex中的class。

接下来我们就从加载patch和合成patch来弄清Tinker的整个工作流程。

## Tinker源码分析之加载补丁Patch流程

默认情况如果使用了Tinker注解产生Application可以看到它继承了TinkerApplication

/\*\*
 \*
 \* Generated application for tinker life cycle
 \*
 \*/
public class Application extends TinkerApplication {

 public Application() {
 super(7, "com.jiuyan.infashion.ApplicationLike", "com.tencent.tinker.loader.TinkerLoader", false);
 }

}

跟踪到TinkerApplication在方法attachBaseContext中找到最终会调用loadTinker方法来,最后反射调用了变量loaderClassName定义类中的tryLoad方法，默认是com.tencent.tinker.loader.TinkerLoader这个类中的tryLoad方法。该方法调用tryLoadPatchFilesInternal来执行相关代码逻辑。

private void tryLoadPatchFilesInternal(TinkerApplication app, Intent resultIntent) {
 //..省略一大段校验相关逻辑代码

 //now we can load patch jar
 if (isEnabledForDex) {
 boolean loadTinkerJars = TinkerDexLoader.loadTinkerJars(app, patchVersionDirectory, oatDex, resultIntent, isSystemOTA);
 if (isSystemOTA) {
 // update fingerprint after load success
 patchInfo.fingerPrint = Build.FINGERPRINT;
 patchInfo.oatDir = loadTinkerJars ? ShareConstants.INTERPRET\_DEX\_OPTIMIZE\_PATH : ShareConstants.DEFAULT\_DEX\_OPTIMIZE\_PATH;
 // reset to false
 oatModeChanged = false;

 if (!SharePatchInfo.rewritePatchInfoFileWithLock(patchInfoFile, patchInfo, patchInfoLockFile)) {
 ShareIntentUtil.setIntentReturnCode(resultIntent, ShareConstants.ERROR\_LOAD\_PATCH\_REWRITE\_PATCH\_INFO\_FAIL);
 Log.w(TAG, "tryLoadPatchFiles:onReWritePatchInfoCorrupted");
 return;
 }
 // update oat dir
 resultIntent.putExtra(ShareIntentUtil.INTENT\_PATCH\_OAT\_DIR, patchInfo.oatDir);
 }
 if (!loadTinkerJars) {
 Log.w(TAG, "tryLoadPatchFiles:onPatchLoadDexesFail");
 return;
 }
 }

 //now we can load patch resource
 if (isEnabledForResource) {
 boolean loadTinkerResources = TinkerResourceLoader.loadTinkerResources(app, patchVersionDirectory, resultIntent);
 if (!loadTinkerResources) {
 Log.w(TAG, "tryLoadPatchFiles:onPatchLoadResourcesFail");
 return;
 }
 }
 // kill all other process if oat mode change
 if (oatModeChanged) {
 ShareTinkerInternals.killAllOtherProcess(app);
 Log.i(TAG, "tryLoadPatchFiles:oatModeChanged, try to kill all other process");
 }
 //all is ok!
 ShareIntentUtil.setIntentReturnCode(resultIntent, ShareConstants.ERROR\_LOAD\_OK);
 Log.i(TAG, "tryLoadPatchFiles: load end, ok!");
 return;
}

这里省略了非常多的Tinker校验，一共有包括tinker自身enable属性以及md5和文件存在等相关检查。

先看加载dex部分，TinkerDexLoader.loadTinkerJars传入四个参数，分别为application，patchVersionDirectory当前patch文件目录，oatDir当前patch的oat文件目录，intent，当前patch是否需要进行oat(由于系统OTA更新需要dex oat重新生成缓存)。

/\*\*
 \* Load tinker JARs and add them to
 \* the Application ClassLoader.
 \*
 \* @param application The application.
 \*/
@TargetApi(Build.VERSION\_CODES.ICE\_CREAM\_SANDWICH)
public static boolean loadTinkerJars(final TinkerApplication application, String directory, String oatDir, Intent intentResult, boolean isSystemOTA) {
 if (loadDexList.isEmpty() && classNDexInfo.isEmpty()) {
 Log.w(TAG, "there is no dex to load");
 return true;
 }

 PathClassLoader classLoader = (PathClassLoader) TinkerDexLoader.class.getClassLoader();
 if (classLoader != null) {
 Log.i(TAG, "classloader: " + classLoader.toString());
 } else {
 Log.e(TAG, "classloader is null");
 ShareIntentUtil.setIntentReturnCode(intentResult, ShareConstants.ERROR\_LOAD\_PATCH\_VERSION\_DEX\_CLASSLOADER\_NULL);
 return false;
 }
 String dexPath = directory + "/" + DEX\_PATH + "/";

 ArrayList<File> legalFiles = new ArrayList<>();

 for (ShareDexDiffPatchInfo info : loadDexList) {
 //for dalvik, ignore art support dex
 if (isJustArtSupportDex(info)) {
 continue;
 }

 String path = dexPath + info.realName;
 File file = new File(path);

 //...check md5
 legalFiles.add(file);
 }
 //... verify merge classN.apk

 File optimizeDir = new File(directory + "/" + oatDir);

 if (isSystemOTA) {
 final boolean[] parallelOTAResult = {true};
 final Throwable[] parallelOTAThrowable = new Throwable[1];
 String targetISA;
 try {
 targetISA = ShareTinkerInternals.getCurrentInstructionSet();
 } catch (Throwable throwable) {
 Log.i(TAG, "getCurrentInstructionSet fail:" + throwable);
// try {
// targetISA = ShareOatUtil.getOatFileInstructionSet(testOptDexFile);
// } catch (Throwable throwable) {
 // don't ota on the front
 deleteOutOfDateOATFile(directory);

 intentResult.putExtra(ShareIntentUtil.INTENT\_PATCH\_INTERPRET\_EXCEPTION, throwable);
 ShareIntentUtil.setIntentReturnCode(intentResult, ShareConstants.ERROR\_LOAD\_PATCH\_GET\_OTA\_INSTRUCTION\_SET\_EXCEPTION);
 return false;
// }
 }

 deleteOutOfDateOATFile(directory);

 Log.w(TAG, "systemOTA, try parallel oat dexes, targetISA:" + targetISA);
 // change dir
 optimizeDir = new File(directory + "/" + INTERPRET\_DEX\_OPTIMIZE\_PATH);

 TinkerDexOptimizer.optimizeAll(
 legalFiles, optimizeDir, true, targetISA,
 new TinkerDexOptimizer.ResultCallback() {
 //... callback
 }
 );

 if (!parallelOTAResult[0]) {
 Log.e(TAG, "parallel oat dexes failed");
 intentResult.putExtra(ShareIntentUtil.INTENT\_PATCH\_INTERPRET\_EXCEPTION, parallelOTAThrowable[0]);
 ShareIntentUtil.setIntentReturnCode(intentResult, ShareConstants.ERROR\_LOAD\_PATCH\_OTA\_INTERPRET\_ONLY\_EXCEPTION);
 return false;
 }
 }
 try {
 SystemClassLoaderAdder.installDexes(application, classLoader, optimizeDir, legalFiles);
 } catch (Throwable e) {
 Log.e(TAG, "install dexes failed");
// e.printStackTrace();
 intentResult.putExtra(ShareIntentUtil.INTENT\_PATCH\_EXCEPTION, e);
 ShareIntentUtil.setIntentReturnCode(intentResult, ShareConstants.ERROR\_LOAD\_PATCH\_VERSION\_DEX\_LOAD\_EXCEPTION);
 return false;
 }

 return true;
}

省略了几处md5校验代码，首先获取到PathClassLoader并且通过判断系统是否art过滤出对应legalFiles，如果发现系统进行过OTA升级则通过ProcessBuilder命令行执行dex2oat进行并行的oat优化dex，最后调用installDexes来安装dex。

 @SuppressLint("NewApi")
public static void installDexes(Application application, PathClassLoader loader, File dexOptDir, List<File> files)
 throws Throwable {
 Log.i(TAG, "installDexes dexOptDir: " + dexOptDir.getAbsolutePath() + ", dex size:" + files.size());

 if (!files.isEmpty()) {
 files = createSortedAdditionalPathEntries(files);
 ClassLoader classLoader = loader;
 if (Build.VERSION.SDK\_INT >= 24 && !checkIsProtectedApp(files)) {
 classLoader = AndroidNClassLoader.inject(loader, application);
 }
 //because in dalvik, if inner class is not the same classloader with it wrapper class.
 //it won't fail at dex2opt
 if (Build.VERSION.SDK\_INT >= 23) {
 V23.install(classLoader, files, dexOptDir);
 } else if (Build.VERSION.SDK\_INT >= 19) {
 V19.install(classLoader, files, dexOptDir);
 } else if (Build.VERSION.SDK\_INT >= 14) {
 V14.install(classLoader, files, dexOptDir);
 } else {
 V4.install(classLoader, files, dexOptDir);
 }
 //install done
 sPatchDexCount = files.size();
 Log.i(TAG, "after loaded classloader: " + classLoader + ", dex size:" + sPatchDexCount);

 if (!checkDexInstall(classLoader)) {
 //reset patch dex
 SystemClassLoaderAdder.uninstallPatchDex(classLoader);
 throw new TinkerRuntimeException(ShareConstants.CHECK\_DEX\_INSTALL\_FAIL);
 }
 }
}

针对不同的Android版本需要对DexPathList中的dexElements生成方法makeDexElements进行适配。

主要做的事情就是获取当前app运行时PathClassLoader的父类BaseDexClassLoader中的pathList对象，通过反射它的makePathElements方法传入对应的path参数构造出Element[]数组对象，然后拿到pathList中的Element[]数组对象dexElements两者进行合并排序，把patch的相关dex信息放在数组前端，最后合并数组结果赋值给pathList保证classloader优先到patch中查找加载。

## Tinker源码分析之合成补丁Patch流程

合并代码入口

Tinker.with(context).getPatchListener().onPatchReceived(patchLocation);

传入patch文件所在位置即可，推荐通过服务端下发下载到对应的/data/data/应用目录下防止被三方软件清理，onPatchReceived方法在DefaultPatchListener.java中。

@Override
public int onPatchReceived(String path) {
 File patchFile = new File(path);

 int returnCode = patchCheck(path, SharePatchFileUtil.getMD5(patchFile));

 if (returnCode == ShareConstants.ERROR\_PATCH\_OK) {
 TinkerPatchService.runPatchService(context, path);
 } else {
 Tinker.with(context).getLoadReporter().onLoadPatchListenerReceiveFail(new File(path), returnCode);
 }
 return returnCode;
}

先进行tinker的一些初始化配置检查还有patch文件的md5校验。如果check通过returnCode为0则执行runPatchService启动一个IntentService的子类TinkerPatchService来处理patch的合成。接下来看Service执行任务代码：

@Override
protected void onHandleIntent(Intent intent) {
 final Context context = getApplicationContext();
 Tinker tinker = Tinker.with(context);
 tinker.getPatchReporter().onPatchServiceStart(intent);

 if (intent == null) {
 TinkerLog.e(TAG, "TinkerPatchService received a null intent, ignoring.");
 return;
 }
 String path = getPatchPathExtra(intent);
 if (path == null) {
 TinkerLog.e(TAG, "TinkerPatchService can't get the path extra, ignoring.");
 return;
 }
 File patchFile = new File(path);

 long begin = SystemClock.elapsedRealtime();
 boolean result;
 long cost;
 Throwable e = null;

 increasingPriority();
 PatchResult patchResult = new PatchResult();
 try {
 if (upgradePatchProcessor == null) {
 throw new TinkerRuntimeException("upgradePatchProcessor is null.");
 }
 result = upgradePatchProcessor.tryPatch(context, path, patchResult);
 } catch (Throwable throwable) {
 e = throwable;
 result = false;
 tinker.getPatchReporter().onPatchException(patchFile, e);
 }

 cost = SystemClock.elapsedRealtime() - begin;
 tinker.getPatchReporter().
 onPatchResult(patchFile, result, cost);

 patchResult.isSuccess = result;
 patchResult.rawPatchFilePath = path;
 patchResult.costTime = cost;
 patchResult.e = e;

 AbstractResultService.runResultService(context, patchResult, getPatchResultExtra(intent));

}

回调PatchReporter接口的onPatchServiceStart方法，然后取到patch文件同时调用increasingPriority启动一个不可见前台Service*保活*这个TinkerPatchService，最后开始合成patchupgradePatchProcessor.tryPatch。同样省略一些常规check代码：

@Override
public boolean tryPatch(Context context, String tempPatchPath, PatchResult patchResult) {
 Tinker manager = Tinker.with(context);
 final File patchFile = new File(tempPatchPath);
 //...省略

 //check ok, we can real recover a new patch
 final String patchDirectory = manager.getPatchDirectory().getAbsolutePath();

 File patchInfoLockFile = SharePatchFileUtil.getPatchInfoLockFile(patchDirectory);
 File patchInfoFile = SharePatchFileUtil.getPatchInfoFile(patchDirectory);

 SharePatchInfo oldInfo = SharePatchInfo.readAndCheckPropertyWithLock(patchInfoFile, patchInfoLockFile);

 //it is a new patch, so we should not find a exist
 SharePatchInfo newInfo;

 //already have patch
 if (oldInfo != null) {
 if (oldInfo.oldVersion == null || oldInfo.newVersion == null || oldInfo.oatDir == null) {
 TinkerLog.e(TAG, "UpgradePatch tryPatch:onPatchInfoCorrupted");
 manager.getPatchReporter().onPatchInfoCorrupted(patchFile, oldInfo.oldVersion, oldInfo.newVersion);
 return false;
 }

 if (!SharePatchFileUtil.checkIfMd5Valid(patchMd5)) {
 TinkerLog.e(TAG, "UpgradePatch tryPatch:onPatchVersionCheckFail md5 %s is valid", patchMd5);
 manager.getPatchReporter().onPatchVersionCheckFail(patchFile, oldInfo, patchMd5);
 return false;
 }
 // if it is interpret now, use changing flag to wait main process
 final String finalOatDir = oldInfo.oatDir.equals(ShareConstants.INTERPRET\_DEX\_OPTIMIZE\_PATH)
 ? ShareConstants.CHANING\_DEX\_OPTIMIZE\_PATH : oldInfo.oatDir;
 newInfo = new SharePatchInfo(oldInfo.oldVersion, patchMd5, Build.FINGERPRINT, finalOatDir);
 } else {
 newInfo = new SharePatchInfo("", patchMd5, Build.FINGERPRINT, ShareConstants.DEFAULT\_DEX\_OPTIMIZE\_PATH);
 }

 //it is a new patch, we first delete if there is any files
 //don't delete dir for faster retry
// SharePatchFileUtil.deleteDir(patchVersionDirectory);
 final String patchName = SharePatchFileUtil.getPatchVersionDirectory(patchMd5);

 final String patchVersionDirectory = patchDirectory + "/" + patchName;

 TinkerLog.i(TAG, "UpgradePatch tryPatch:patchVersionDirectory:%s", patchVersionDirectory);

 //copy file
 File destPatchFile = new File(patchVersionDirectory + "/" + SharePatchFileUtil.getPatchVersionFile(patchMd5));

 //...省略

 if (!DexDiffPatchInternal.tryRecoverDexFiles(manager, signatureCheck, context, patchVersionDirectory, destPatchFile)) {
 TinkerLog.e(TAG, "UpgradePatch tryPatch:new patch recover, try patch dex failed");
 return false;
 }

 if (!BsDiffPatchInternal.tryRecoverLibraryFiles(manager, signatureCheck, context, patchVersionDirectory, destPatchFile)) {
 TinkerLog.e(TAG, "UpgradePatch tryPatch:new patch recover, try patch library failed");
 return false;
 }

 if (!ResDiffPatchInternal.tryRecoverResourceFiles(manager, signatureCheck, context, patchVersionDirectory, destPatchFile)) {
 TinkerLog.e(TAG, "UpgradePatch tryPatch:new patch recover, try patch resource failed");
 return false;
 }

 //...省略
}

1.检查是否有之前的patch信息oldInfo,查看旧补丁是否正在执行oat过程,后续会等待主进程oat执行完毕。 2.拷贝new patch到app的data目录的tinker目录下，防止被三方软件删除。 3.分别判断执行tryRecoverDexFiles合成dex，tryRecoverLibraryFiles合成so以及tryRecoverResourceFiles合成资源。

主要看下dex合成过程，这也是我们最关心的地方。

protected static boolean tryRecoverDexFiles(Tinker manager, ShareSecurityCheck checker, Context context,
 String patchVersionDirectory, File patchFile) {
 if (!manager.isEnabledForDex()) {
 TinkerLog.w(TAG, "patch recover, dex is not enabled");
 return true;
 }
 String dexMeta = checker.getMetaContentMap().get(DEX\_META\_FILE);

 if (dexMeta == null) {
 TinkerLog.w(TAG, "patch recover, dex is not contained");
 return true;
 }

 long begin = SystemClock.elapsedRealtime();
 boolean result = patchDexExtractViaDexDiff(context, patchVersionDirectory, dexMeta, patchFile);
 long cost = SystemClock.elapsedRealtime() - begin;
 TinkerLog.i(TAG, "recover dex result:%b, cost:%d", result, cost);
 return result;
}

读取patch包assets/dex\_meta.txt信息转换成String，进入patchDexExtractViaDexDiff方法。

private static boolean patchDexExtractViaDexDiff(Context context, String patchVersionDirectory, String meta, final File patchFile) {
 String dir = patchVersionDirectory + "/" + DEX\_PATH + "/";

 if (!extractDexDiffInternals(context, dir, meta, patchFile, TYPE\_DEX)) {
 TinkerLog.w(TAG, "patch recover, extractDiffInternals fail");
 return false;
 }

 File dexFiles = new File(dir);
 File[] files = dexFiles.listFiles();
 List<File> dexList = files != null ? Arrays.asList(files) : null;

 final String optimizeDexDirectory = patchVersionDirectory + "/" + DEX\_OPTIMIZE\_PATH + "/";
 return dexOptimizeDexFiles(context, dexList, optimizeDexDirectory, patchFile);

}

首先执行方法extractDexDiffInternals传入了合成后dex路径,前面读取的dex\_meta信息,patch文件以及type类型dex。为了节约篇幅只提取了主要的代码，详细代码参考github。

private static boolean extractDexDiffInternals(Context context, String dir, String meta, File patchFile, int type) {
 //parse
 patchList.clear();
 ShareDexDiffPatchInfo.parseDexDiffPatchInfo(meta, patchList);
 //获取base.apk
 String apkPath = applicationInfo.sourceDir;
 apk = new ZipFile(apkPath);
 patch = new ZipFile(patchFile);
 for (ShareDexDiffPatchInfo info : patchList) {
 String patchRealPath;
 if (infoPath.equals("")) {
 patchRealPath = info.rawName;
 } else {
 patchRealPath = info.path + "/" + info.rawName;
 }
 File extractedFile = new File(dir + info.realName);
 //..省略

 ZipEntry patchFileEntry = patch.getEntry(patchRealPath);
 ZipEntry rawApkFileEntry = apk.getEntry(patchRealPath);

 patchDexFile(apk, patch, rawApkFileEntry, patchFileEntry, info, extractedFile);
 }

 if (!mergeClassNDexFiles(context, patchFile, dir)) {
 return false;
 }
}

1.解析dex\_meta内容

img

img

image

对应的

ShareDexDiffPatchInfo

信息

final String name = kv[0].trim();
final String path = kv[1].trim();
final String destMd5InDvm = kv[2].trim();
final String destMd5InArt = kv[3].trim();
final String dexDiffMd5 = kv[4].trim();
final String oldDexCrc = kv[5].trim();
final String newDexCrc = kv[6].trim();
final String dexMode = kv[7].trim();

2.循环遍历获取到patch中各个classes.dex的crc和md5信息以及一大片校验代码，调用patchDexFile方法对base.apk和patch中的dex做合并生成新的dex。

3.把合成的dex压缩为一个tinker\_classN.apk

接下来看patchDexFile方法，同样只提取了关键代码。

private static void patchDexFile(
 ZipFile baseApk, ZipFile patchPkg, ZipEntry oldDexEntry, ZipEntry patchFileEntry,
 ShareDexDiffPatchInfo patchInfo, File patchedDexFile) throws IOException {
 InputStream oldDexStream = null;
 InputStream patchFileStream = null;

 oldDexStream = new BufferedInputStream(baseApk.getInputStream(oldDexEntry));
 patchFileStream = (patchFileEntry != null ? new BufferedInputStream(patchPkg.getInputStream(patchFileEntry)) : null);

 //...省略判断dex是否是jar类型或者是raw类型，做不同处理

 new DexPatchApplier(oldDexStream, patchFileStream).executeAndSaveTo(patchedDexFile);
}

下面是github官网上对raw和jar区别的解释

Tinker中的dex配置'raw'与'jar'模式应该如何选择？ 它们应该说各有优劣势，大概应该有以下几条原则： 如果你的minSdkVersion小于14, 那你务必要选择'jar'模式； 以一个10M的dex为例，它压缩成jar大约为4M，即'jar'模式能节省6M的ROM空间。 对于'jar'模式，我们需要验证压缩包流中dex的md5,这会更耗时，在小米2S上数据大约为'raw'模式126ms, 'jar'模式为246ms。 因为在合成过程中我们已经校验了各个文件的Md5，并将它们存放在/data/data/..目录中。默认每次加载时我们并不会去校验tinker文件的Md5,但是你也可通过开启loadVerifyFlag强制每次加载时校验，但是这会带来一定的时间损耗。 简单来说，'jar'模式更省空间，但是运行时校验的耗时大约为'raw'模式的两倍。如果你没有打开运行时校验，推荐使用'jar'模式。

最后通过ZipFile拿到base.apk和patch中对应dex文件进行合成为patchedDexFile。核心部分是如何把差分的dex和基准dex做合成处理产生新的dex，这部分涉及到了dex文件结构、DexDiff和DexPatch算法