当程序越来越大之后，出现了一个 dex 包装不下的情况，通过 `MultiDex` 的方法解决了这个问题，但是在底端机器上又出现了 `INSTALL_FAILED_DEXOPT` 的情况，那再解决这个问题吧。等解决完这个问题之后，发现需要填的坑越来越多了，文章讲的是我在分包处理中填的坑，比如 65536、LinearAlloc、NoClassDefFoundError等等。

INSTALL_FAILED_DEXOPT

INSTALL_FAILED_DEXOPT 出现的原因大部分都是两种，一种是 65536 了，另外一种是 `LinearAlloc` 太小了。两者的限制不同，但是原因却是相似，那就是App太大了，导致没办法安装到手机上。

65536

> trouble writing output: Too many method references: 70048; max is 65536.

或者

> UNEXPECTED TOP-LEVEL EXCEPTION: java.lang.IllegalArgumentException: method ID not in [0, 0xffff]: 65536
> at com.android.dx.merge.DexMerger$6.updateIndex(DexMerger.java:501)
> at com.android.dx.merge.DexMerger$IdMerger.mergeSorted(DexMerger.java:276)
> at com.android.dx.merge.DexMerger.mergeMethodIds(DexMerger.java:490)
> at com.android.dx.merge.DexMerger.mergeDexes(DexMerger.java:167)
> at com.android.dx.merge.DexMerger.merge(DexMerger.java:188)
> at com.android.dx.command.dexer.Main.mergeLibraryDexBuffers(Main.java:439)
> at com.android.dx.command.dexer.Main.runMonoDex(Main.java:287)
> at com.android.dx.command.dexer.Main.run(Main.java:230)
> at com.android.dx.command.dexer.Main.main(Main.java:199)
> at com.android.dx.command.Main.main(Main.java:103):Derp:dexDerpDebug FAILED

编译环境

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.3.0'
    }
}

android {
    compileSdkVersion 23
    buildToolsVersion "24.0.0rc1"
    //....
    defaultConfig {
        minSdkVersion 14
        targetSdkVersion 23
        //....
    }
}
```

为什么是65536

根据 StackOverFlow – Does the Android ART runtime have the same method limit limitations as Dalvik? 上面的说法，是因为 Dalvik 的 invoke-kind 指令集中，method reference index 只留了 16 bits，最多能**引用** 65535 个方法。Dalvik bytecode：

> | Op & Format | Mnemonic / Syntax | Arguments |
> | --- | --- | --- |
> | 6e..72 35c | invoke-*kind* {vC, vD, vE, vF, vG}, meth@BBBB6e: invoke-virtual6f: invoke-super70: invoke-direct71: invoke-static72: invoke-interface | `A:` argument word count (4 bits)`B:` method reference index (16 bits)`C..G:` argument registers (4 bits each) |

即使 dex 里面的引用方法数超过了 65536，那也只有前面的 65536 得的到调用。所以这个不是 dex 的原因。其次，既然和 dex 没有关系，那在打包 dex 的时候为什么会报错。我们先定位 `Too many` 关键字，定位到了MemberIdsSection：

```
public abstract class MemberIdsSection extends UniformItemSection {
  /** {@inheritDoc} */
    @Override
    protected void orderItems() {
        int idx = 0;

        if (items().size() > DexFormat.MAX_MEMBER_IDX + 1) {
            throw new DexIndexOverflowException(getTooManyMembersMessage());
        }

        for (Object i : items()) {
            ((MemberIdItem) i).setIndex(idx);
            idx++;
        }
    }

    private String getTooManyMembersMessage() {
        Map<String, AtomicInteger> membersByPackage = new TreeMap<String,
AtomicInteger>();
        for (Object member : items()) {
            String packageName = ((MemberIdItem)
member).getDefiningClass().getPackageName();
            AtomicInteger count = membersByPackage.get(packageName);
            if (count == null) {
                count = new AtomicInteger();
                membersByPackage.put(packageName, count);
            }
            count.incrementAndGet();
        }

        Formatter formatter = new Formatter();
        try {
            String memberType = this instanceof MethodIdsSection ? "method" :
"field";
            formatter.format("Too many %s references: %d; max is %d.%n" +
                    Main.getTooManyIdsErrorMessage() + "%n" +
                    "References by package:",
                    memberType, items().size(), DexFormat.MAX_MEMBER_IDX + 1);
```

```
            for (Map.Entry<String, AtomicInteger> entry :
membersByPackage.entrySet()) {
                formatter.format("%n%6d %s", entry.getValue().get(),
entry.getKey());
            }
            return formatter.toString();
        } finally {
            formatter.close();
        }
    }
}
```

`items().size() > DexFormat.MAX_MEMBER_IDX + 1`，那 DexFormat 的值是：

```
public final class DexFormat {
  /**
     * Maximum addressable field or method index.
     * The largest addressable member is 0xffff, in the "instruction formats"
spec as field@CCCC or
     * meth@CCCC.
     */
    public static final int MAX_MEMBER_IDX = 0xFFFF;
}
```

dx 在这里做了判断，当大于 65536 的时候就抛出异常了。所以在生成 dex 文件的过程中，当调用方法数不能超过 65535 。那我们再跟一跟代码，发现 `MemberIdsSection` 的一个子类叫 MethodidsSection：

```
public final class MethodIdsSection extends MemberIdsSection {}
```

回过头来，看一下 `orderItems()` 方法在哪里被调用了，跟到了 `MemberIdsSection` 的父类 UniformItemSection：

```
public abstract class UniformItemSection extends Section {
    @Override
    protected final void prepare0() {
        DexFile file = getFile();

        orderItems();

        for (Item one : items()) {
            one.addContents(file);
        }
    }

    protected abstract void orderItems();
}
```

再跟一下 `prepare0` 在哪里被调用，查到了 `UniformItemSection` 父类 Section:

```
public abstract class Section {
    public final void prepare() {
        throwIfPrepared();
        prepare0();
        prepared = true;
    }

    protected abstract void prepare0();
}
```

那现在再跟一下 `prepare()`，查到 DexFile 中有调用：

```
public final class DexFile {
  private ByteArrayAnnotatedOutput toDex0(boolean annotate, boolean verbose) {
        classDefs.prepare();
        classData.prepare();
        wordData.prepare();
        byteData.prepare();
        methodIds.prepare();
        fieldIds.prepare();
        protoIds.prepare();
        typeLists.prepare();
        typeIds.prepare();
        stringIds.prepare();
        stringData.prepare();
        header.prepare();
        //blablabla......
    }
}
```

那再看一下 `toDex0()` 吧，因为是 private 的，直接在类中找调用的地方就可以了：

```
public final class DexFile {
    public byte[] toDex(Writer humanOut, boolean verbose) throws IOException {
        boolean annotate = (humanOut != null);
        ByteArrayAnnotatedOutput result = toDex0(annotate, verbose);

        if (annotate) {
            result.writeAnnotationsTo(humanOut);
        }

        return result.getArray();
    }

    public void writeTo(OutputStream out, Writer humanOut, boolean verbose)
throws IOException {
        boolean annotate = (humanOut != null);
        ByteArrayAnnotatedOutput result = toDex0(annotate, verbose);

        if (out != null) {
            out.write(result.getArray());
        }

        if (annotate) {
            result.writeAnnotationsTo(humanOut);
        }
```

```
    }
}
```

先搜搜 `toDex()` 方法吧，最终发现在 com.android.dx.command.dexer.Main 中：

```
public class Main {
    private static byte[] writeDex(DexFile outputDex) {
        byte[] outArray = null;
        //blablabla......
        if (args.methodToDump != null) {
            outputDex.toDex(null, false);
            dumpMethod(outputDex, args.methodToDump, humanOutWriter);
        } else {
            outArray = outputDex.toDex(humanOutWriter, args.verboseDump);
        }
        //blablabla......
        return outArray;
    }
    //调用writeDex的地方
    private static int runMonoDex() throws IOException {
        //blablabla......
        outArray = writeDex(outputDex);
        //blablabla......
    }
    //调用runMonoDex的地方
    public static int run(Arguments arguments) throws IOException {
        if (args.multiDex) {
            return runMultiDex();
        } else {
            return runMonoDex();
        }
    }
}
```

`args.multiDex` 就是是否分包的参数，那么问题找着了，如果不选择分包的情况下，引用方法数超过了 65536 的话就会抛出异常。

同样分析第二种情况，根据错误信息可以具体定位到代码，但是很奇怪的是 `DexMerger` ，我们没有设置分包参数或者其他参数，为什么会有 `DexMerger` ，而且依赖工程最终不都是 aar 格式的吗？那我们还是来跟一跟代码吧。

```
public class Main {
    private static byte[] mergeLibraryDexBuffers(byte[] outArray) throws
IOException {
        ArrayList<Dex> dexes = new ArrayList<Dex>();
        if (outArray != null) {
            dexes.add(new Dex(outArray));
        }
        for (byte[] libraryDex : libraryDexBuffers) {
            dexes.add(new Dex(libraryDex));
        }
        if (dexes.isEmpty()) {
            return null;
        }
        Dex merged = new DexMerger(dexes.toArray(new Dex[dexes.size()]),
CollisionPolicy.FAIL).merge();
```

```
        return merged.getBytes();
    }
}
```

这里可以看到变量 `libraryDexBuffers` ，是一个 List 集合，那么我们看一下这个集合在哪里添加数据的：

```
public class Main {
    private static boolean processFileBytes(String name, long lastModified,
byte[] bytes) {
        boolean isClassesDex = name.equals(DexFormat.DEX_IN_JAR_NAME);
        //blablabla...
        } else if (isClassesDex) {
            synchronized (libraryDexBuffers) {
                libraryDexBuffers.add(bytes);
            }
            return true;
        } else {
        //blablabla...
    }
    //调用processFileBytes的地方
    private static class FileBytesConsumer implements ClassPathOpener.Consumer {

        @Override
        public boolean processFileBytes(String name, long lastModified,
                byte[] bytes)   {
            return Main.processFileBytes(name, lastModified, bytes);
        }
        //blablabla...
    }
    //调用FileBytesConsumer的地方
    private static void processOne(String pathname, FileNameFilter filter) {
        ClassPathOpener opener;

        opener = new ClassPathOpener(pathname, true, filter, new
FileBytesConsumer());

        if (opener.process()) {
          updateStatus(true);
        }
    }
    //调用processOne的地方
    private static boolean processAllFiles() {
        //blablabla...
        // forced in main dex
        for (int i = 0; i < fileNames.length; i++) {
            processOne(fileNames[i], mainPassFilter);
        }
        //blablabla...
    }
    //调用processAllFiles的地方
    private static int runMonoDex() throws IOException {
        //blablabla...
        if (!processAllFiles()) {
            return 1;
        }
        //blablabla...
```

```
    }

}
```

跟了一圈又跟回来了，但是注意一个变量：`fileNames[i]`，传进去这个变量，是个地址，最终在 `processFileBytes` 中处理后添加到 `libraryDexBuffers` 中，那跟一下这个变量：

```
public class Main {
    private static boolean processAllFiles() {
        //blablabla...
        String[] fileNames = args.fileNames;
        //blablabla...
    }
    public void parse(String[] args) {
        //blablabla...
        }else if(parser.isArg(INPUT_LIST_OPTION + "=")) {
            File inputListFile = new File(parser.getLastValue());
            try{
                inputList = new ArrayList<String>();
                readPathsFromFile(inputListFile.getAbsolutePath(), inputList);
            } catch(IOException e) {
                System.err.println("Unable to read input list file: " +
inputListFile.getName());
                throw new UsageException();
            }
        } else {
        //blablabla...
        fileNames = parser.getRemaining();
        if(inputList != null && !inputList.isEmpty()) {
            inputList.addAll(Arrays.asList(fileNames));
            fileNames = inputList.toArray(new String[inputList.size()]);
        }
    }

    public static void main(String[] argArray) throws IOException {
        Arguments arguments = new Arguments();
        arguments.parse(argArray);

        int result = run(arguments);
        if (result != 0) {
            System.exit(result);
        }
    }
}
```

跟到这里发现是传进来的参数，那我们再看看 gradle 里面传的是什么参数吧，查看 Dex task :

```
public class Dex extends BaseTask {
    @InputFiles
    Collection<File> libraries
}
```

我们把这个参数打印出来：

```
afterEvaluate {
    tasks.matching {
        it.name.startsWith('dex')
    }.each { dx ->
        if (dx.additionalParameters == null) {
            dx.additionalParameters = []
        }
        println dx.libraries
    }
}
```

打印出来发现是 `build/intermediates/pre-dexed/` 目录里面的 jar 文件，再把 jar 文件解压发现里面就是 dex 文件了。所以 `DexMerger` 的工作就是合并这里的 dex 。

更改编译环境

```
buildscript {
    //...
    dependencies {
        classpath 'com.android.tools.build:gradle:2.1.0-alpha3'
    }
}
```

将 gradle 设置为 2.1.0-alpha3 之后，在项目的 `build.gradle` 中即使没有设置 `multiDexEnabled true` 也能够编译通过，但是生成的 apk 包依旧是两个 dex ，我想的是可能为了设置 `instantRun` 。

解决 65536

Google MultiDex 解决方案：

在 gradle 中添加 `MultiDex` 的依赖：

```
dependencies { compile 'com.android.support:MultiDex:1.0.0' }
```

在 gradle 中配置 `MultiDexEnable` ：

```
android {
    buildToolsVersion "21.1.0"
    defaultConfig {
        // Enabling MultiDex support.
        MultiDexEnabled true
  }
}
```

在 AndroidManifest.xml 的 application 中声明:

```
<application
  android:name="android.support.multidex.MultiDexApplication">
<application/>
```

如果有自己的 Application 了，让其继承于 MultiDexApplication 。

如果继承了其他的 Application ，那么可以重写 `attachBaseContext(Context)` ：

```
@Override
protected void attachBaseContext(Context base) {
    super.attachBaseContext(base);
    MultiDex.install(this);
}
```

LinearAlloc

> ERROR/dalvikvm(4620): LinearAlloc exceeded capacity (5242880), last=...

LinearAlloc 是什么

LinearAlloc 主要用来管理 Dalvik 中 class 加载时的内存，就是让 App 在执行时减少系统内存的占用。在 App 的安装过程中，系统会运行一个名为 dexopt 的程序为该应用在当前机型中运行做准备。dexopt 使用 LinearAlloc 来存储应用的方法信息。App 在执行前会将 class 读进 LinearAlloc 这个 buffer 中，这个 LinearAlloc 在 Android 2.3 之前是 4M 或 5M，到 4.0 之后变为 8M 或 16M。因为 5M 实在是太小了，可能还没有 65536 就已经超过 5M 了，什么意思呢，就是只有一个包的情况下也有可能出现 INSTALL_FAILED_DEXOPT，原因就在于 LinearAlloc。

解决 LinearAlloc

gradle:

```
afterEvaluate {
  tasks.matching {
    it.name.startsWith('dex')
  }.each { dx ->
    if (dx.additionalParameters == null) {
      dx.additionalParameters = []
    }
    dx.additionalParameters += '--set-max-idx-number=48000'
  }
}
```

`--set-max-idx-number=` 用于控制每一个 dex 的最大方法个数。

这个参数在查看 dx.jar 找到:

```
//blablabla...
} else if (parser.isArg("--set-max-idx-number=")) { // undocumented test option
  maxNumberOfIdxPerDex = Integer.parseInt(parser.getLastValue());
} else if(parser.isArg(INPUT_LIST_OPTION + "=")) {
//blablabla...
```

FB 的工程师们曾经还想到过直接修改 LinearAlloc 的□大小，比如从 5M 修改到 8M： Under the Hood: Dalvik patch for Facebook for Android。

dexopt && dex2oat

dexopt_dex2oat

dexopt

当 Android 系统安装一个应用的时候，有一步是对 Dex 进行优化，这个过程有一个专门的工具来处理，叫 DexOpt。DexOpt 是在第一次加载 Dex 文件的时候执行的，将 dex 的依赖库文件和一些辅助数据打包成 odex 文件，即 Optimised Dex，存放在 cache/dalvik_cache 目录下。保存格式为 `apk路径 @ apk名 @ classes.dex` 。执行 ODEX 的效率会比直接执行 Dex 文件的效率要高很多。

dex2oat

Android Runtime 的 dex2oat 是将 dex 文件编译成 oat 文件。而 oat 文件是 elf 文件，是可以在本地执行的文件，而 Android Runtime 替换掉了虚拟机读取的字节码转而用本地可执行代码，这就被叫做 AOT(ahead-of-time)。dex2oat 对所有 apk 进行编译并保存在 dalvik-cache 目录里。PackageManagerService 会持续扫描安装目录，如果有新的 App 安装则马上调用 dex2oat 进行编译。

NoClassDefFoundError

现在 INSTALL_FAILED_DEXOPT 问题是解决了，但是有时候编译完运行的时候一打开 App 就 crash 了，查看 log 发现是某个类找不到引用。

Build Tool 是如何分包的

为什么会这样呢？是因为 build-tool 在分包的时候只判断了直接引用类。什么是直接引用类呢？举个栗子：

```
public class MainActivity extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        DirectReferenceClass test = new DirectReferenceClass();
    }
}

public class DirectReferenceClass {
    public DirectReferenceClass() {
        InDirectReferenceClass test = new InDirectReferenceClass();
    }
}

public class InDirectReferenceClass {
    public InDirectReferenceClass() {

    }
}
```

上面有 MainActivity、DirectReferenceClass 、InDirectReferenceClass 三个类，其中 DirectReferenceClass 是 MainActivity 的直接引用类，InDirectReferenceClass 是 DirectReferenceClass 的直接引用类。而 InDirectReferenceClass 是 MainActivity 的间接引用类(即直接引用类的所有直接引用类)。

如果我们代码是这样写的：

```
public class HelloMultiDexApplication extends Application {
    @Override
    protected void attachBaseContext(Context base) {
        super.attachBaseContext(base);
        DirectReferenceClass test = new DirectReferenceClass();
        MultiDex.install(this);
    }
}
```

这样直接就 crash 了。同理还要单例模式中拿到单例之后直接调用某个方法返回的是另外一个对象，并非单例对象。

build tool 的分包操作可以查看 sdk 中 build-tools 文件夹下的 `mainDexClasses` 脚本，同时还发现了 `mainDexClasses.rules` 文件，该文件是主 dex 的匹配规则。该脚本要求输入一个文件组（包含编译后的目录或jar包），然后分析文件组中的类并写入到–output所指定的文件中。实现原理也不复杂，主要分为三步：

1. 环境检查，包括传入参数合法性检查，路径检查以及proguard环境检测等。
2. 使用mainDexClasses.rules规则，通过Proguard的shrink功能，裁剪无关类，生成一个tmp.jar包。
3. 通过生成的tmp jar包，调用MainDexListBuilder类生成主dex的文件列表。

Gradle 打包流程中是如何分包的

在项目中，可以直接运行 gradle 的 task 。

- `collect{flavor}{buildType}MultiDexComponents` Task 。这个 task 是获取 AndroidManifest.xml 中 Application 、Activity 、Service 、 Receiver 、 Provider 等相关类，以及 Annotation ， 之后将内容写到 `build/intermediates/multi-dex/{flavor}/{buildType}/maindexlist.txt` 文件中去。
- `packageAll{flavor}DebugClassesForMultiDex` Task 。该 task 是将所有类打包成 jar 文件存在 `build/intermediates/multi-dex/{flavor}/debug/allclasses.jar` 。 当 BuildType 为 Release 的时候，执行的是 `proguard{flavor}Release` Task，该 task 将 proguard 混淆后的类打包成 jar 文件存在 `build/intermediates/classes-proguard/{flavor}/release/classes.jar`
- `shrink{flavor}{buildType}MultiDexComponents` Task 。该 task 会根据 maindexlist.txt 生成 componentClasses.jar ， 该 jar 包里面就只有 maindexlist.txt 里面的类，该 jar 包的位置在 `build/intermediates/multi-dex/{flavor}/{buildType}/componentClasses.jar`
- `create{flavor}{buildType}MainDexClassList` Task 。该 task 会根据生成的 componentClasses.jar 去找这里面的所有的 class 中直接依赖的 class ， 然后将内容写到 `build/intermediates/multi-dex/{flavor}/{buildType}/maindexlist.txt` 中。最终这个文件里面列出来的类都会被分配到第一个 dex 里面。

解决 NoClassDefFoundError

gradle :

```
afterEvaluate {
  tasks.matching {
    it.name.startsWith('dex')
  }.each { dx ->
    if (dx.additionalParameters == null) {
      dx.additionalParameters = []
    }
    dx.additionalParameters += '--set-max-idx-number=48000'
    dx.additionalParameters += "--main-dex-
list=$projectDir/multidex.keep".toString()
  }
}
```

`--main-dex-list=` 参数是一个类列表的文件，在该文件中的类会被打包在第一个 dex 中。

multidex.keep 里面列上需要打包到第一个 dex 的 class 文件，注意，如果需要混淆的话需要写混淆之后的 class 。

Application Not Responding

因为第一次运行（包括清除数据之后）的时候需要 dexopt ，然而 dexopt 是一个比较耗时的操作，同时 `MultiDex.install()` 操作是在 `Application.attachBaseContext()` 中进行的，占用的是UI线程。那么问题来了，当我的第二个包、第三个包很大的时候，程序就阻塞在 `MultiDex.install()` 这个地方了，一旦超过规定时间，那就 ANR 了。那怎么办？放子线程？如果 Application 有一些初始化操作，到初始化操作的地方的时候都还没有完成 install + dexopt 的话，那不是又 NoClassDefFoundError 了吗？同时 ClassLoader 放在哪个线程都让主线程挂起。好了，那在 multidex.keep 的加上相关的所有的类吧。好像这样成了，但是第一个 dex 又大起来了，而且如果用户操作快，还没完成 install + dexopt 但是已经把 App 所以界面□都打开了一遍。。。虽然这不现实。。

微信加载方案

首次加载在地球中页中, 并用线程去加载（但是 5.0 之前加载 dex 时还是会挂起主线程一段时间（不是全程都挂起））。

- dex 形式

  微信是将包放在 `assets` 目录下的，在加载 Dex 的代码时，实际上传进去的是 zip，在加载前需要验证 MD5，确保所加载的 Dex 没有被篡改。

- dex 类分包规则

  分包规则即将所有 Application、ContentProvider 以及所有 export 的 Activity、Service 、Receiver 的间接依赖集都必须放在主 dex。

- 加载 dex 的方式

  加载逻辑这边主要判断是否已经 dexopt，若已经 dexopt，即放在 attachBaseContext 加载，反之放于地球中用线程加载。怎么判断？因为在微信中，若判断 revision 改变，即将 dex 以及 dexopt 目录清空。只需简单判断两个目录 dex 名称、数量是否与配置文件的一致。

总的来说，这种方案用户体验较好，缺点在于太过复杂，每次都需重新扫描依赖集，而且使用的是比较大的间接依赖集。

Facebook 加载方案

Facebook的思路是将 `MultiDex.install()` 操作放在另外一个经常进行的。

- dex 形式

  与微信相同。

- dex 类分包规则

  Facebook 将加载 dex 的逻辑单独放于一个单独的 nodex 进程中。

```
  <activity
  android:exported="false"
  android:process=":nodex"

android:name="com.facebook.nodex.startup.splashscreen.NodexSplashActivity">
```

所有的依赖集为 Application、NodexSplashActivity 的间接依赖集即可。

- 加载 dex 的方式

  因为 `NodexSplashActivity` 的 intent-filter 指定为 `Main` 和 `LAUNCHER` ，所以一打开 App 首先拉起 nodex 进程，然后打开 `NodexSplashActivity` 进行 `MultiDex.install()` 。如果已经进行了 dexpot 操作的话就直接跳转主界面，没有的话就等待 dexpot 操作完成再跳转主界面。

这种方式好处在于依赖集非常简单，同时首次加载 dex 时也不会卡死。但是它的缺点也很明显，即每次启动主进程时，都需先启动 nodex 进程。尽管 nodex 进程逻辑非常简单，这也需100ms以上。

美团加载方案

- dex 形式

  在 gradle 生成 dex 文件的这步中，自定义一个 task 来干预 dex 的生产过程，从而产生多个 dex 。

```
  tasks.whenTaskAdded { task ->
      if (task.name.startsWith('proguard') && (task.name.endsWith('Debug') ||
task.name.endsWith('Release'))) {
          task.doLast {
              makeDexFileAfterProguardJar();
          }
          task.doFirst {
              delete "${project.buildDir}/intermediates/classes-proguard";

              String flavor = task.name.substring('proguard'.length(),
task.name.lastIndexOf(task.name.endsWith('Debug') ? "Debug" : "Release"));
              generateMainIndexKeepList(flavor.toLowerCase());
          }
      } else if (task.name.startsWith('zipalign') && (task.name.endsWith('Debug')
|| task.name.endsWith('Release'))) {
          task.doFirst {
              ensureMultiDexInApk();
          }
      }
  }
```

- dex 类分包规则

把 Service、Receiver、Provider 涉及到的代码都放到主 dex 中，而把 Activity 涉及到的代码进行了一定的拆分，把首页 Activity、Laucher Activity 、欢迎页的 Activity 、城市列表页 Activity 等所依赖的 class 放到了主 dex 中，把二级、三级页面的 Activity 以及业务频道的代码放到了第二个 dex 中，为了减少人工分析 class 的依赖所带了的不可维护性和高风险性，美团编写了一个能够自动分析 class 依赖的脚本， 从而能够保证□主 dex 包含 class 以及他们所依赖的所有 class 都在其内，这样这个脚本就会在打包之前自动分析出启动到主 dex 所涉及的所有代码，保证主 dex 运行正常。

- 加载 dex 的方式

  通过分析 Activity 的启动过程，发现 Activity 是由 ActivityThread 通过 Instrumentation 来启动的，那么是否可以在 Instrumentation 中做一定的手脚呢？通过分析代码 ActivityThread 和 Instrumentation 发现，Instrumentation 有关 Activity 启动相关的方法大概有：execStartActivity、 newActivity 等等，这样就可以在这些方法中添加代码逻辑进行判断这个 class 是否加载了，如果加载则直接启动这个 Activity，如果没有加载完成则启动一个等待的 Activity 显示给用户，然后在这个 Activity 中等待后台第二个 dex 加载完成，完成后自动跳转到用户实际要跳转的 Activity；这样在代码充分解耦合，以及每个业务代码能够做到颗粒化的前提下，就做到第二个 dex 的按需加载了。

美团的这种方式对主 dex 的要求非常高，因为第二个 dex 是等到需要的时候再去加载。重写 Instrumentation 的 execStartActivity 方法，hook 跳转 Activity 的总入口做判断，如果当前第二个 dex 还没有加载完成，就弹一个 loading Activity等待加载完成。

综合加载方案

微信的方案需要将 dex 放于 `assets` 目录下，在打包的时候太过负责；Facebook 的方案每次进入都是开启一个 nodex 进程，而我们希望节省资源的同时快速打开 App；美团的方案确实很 hack，但是对于项目已经很庞大，耦合度又比较高的情况下并不适合。所以这里尝试结合三个方案，针对自己的项目来进行优化。

- dex 形式

  第一，为了能够继续支持 Android 2.x 的机型，我们将每个包的方法数控制在 48000 个，这样最后分出来 dex 包大约在 5M 左右；第二，为了防止 NoClassDefFoundError 的情况，我们找出来启动页、引导页、首页比较在意的一些类，比如 Fragment 等（因为在生成 maindexlist.txt 的时候只会找 Activity 的直接引用，比如首页 Activity 直接引用 AFragemnt，但是 AFragment 的引用并没有去找）。

- dex 类分包规则

  第一个包放 Application、Android四大组件以及启动页、引导页、首页的直接引用的 Fragment 的引用类，还放了推送消息过来点击 Notification 之后要展示的 Activity 中的 Fragment 的引用类。

  Fragment 的引用类是写了一个脚本，输入需要找的类然后将这些引用类写到 multidex.keep 文件中，如果是 debug 的就直接在生成的 jar 里面找，如果是 release 的话就通过 mapping.txt 找，找不到的话再去 jar 里面找，所以在 gradle 打包的过程中我们人为干扰一下：

```
  tasks.whenTaskAdded { task ->
      if (task.name.startsWith("create") &&
task.name.endsWith("MainDexClassList")) {
          task.doLast {
              def flavorAndBuildType = task.name.substring("create".length(),
task.name.length() - "MainDexClassList".length())
              autoSplitDex.configure {
                  description = flavorAndBuildType
              }
              autoSplitDex.execute()
          }
      }
  }
```

- 加载 dex 的方式

  在防止 ANR 方面，我们采用了 Facebook 的思路。但是稍微有一点区别，差别在于我们并不在一开启 App 的时候就去起进程，而是一开启 App 的时候在主进程里面判断是否 dexopt 过没，没有的话再去起另外的进程的 Activity 专门做 dexopt 操作 。一旦拉起了去做 dexopt 的进程，那么让主进程进入一个死循环，一直等到 dexopt 进程结束再结束死循环往下走。那么问题来了，第一，主进程进入死循环会 ANR 吗？第二，如何判断是否 dexopt 过；第三，为了界面友好，dexopt 的进程该怎么做；第四，主进程怎么知道 dexopt 进程结束了，也就是怎么去做进程间通信。

  一个一个问题的解决，先第一个：因为当拉起 dexopt 进程之后，我们在 dexopt 进程的 Activity 中进行 `MultiDex.install()` 操作，此时主进程□不再是前台进程了，所以不会 ANR 。

  第二个问题：因为第一次启动是什么数据都没有的，那么我们就建立一个 `SharedPreference` ，启动的时候先去从这里获取数据，如果没有数据那么也就是没有 dexopt 过，如果有数据那么肯定是 dexopt 过的，但是这个 `SharedPreference` 我们得保证我们的程序只有这个地方可以修改，其他地方不能修改。

  第三个问题：因为 App 的启动也是一张图片，所以在 dexopt 的 Activity 的 layout 中，我们就把这张图片设置上去就好了，当关闭 dexopt 的 Activity 的时候，我们得关闭 Activity 的动画。同时为了不让 dexopt 进程发生 ANR ，我们将 `MultiDex.install()` 过程放在了子线程中进行。

  第四个问题：Linux 的进程间通信的方式有很多，Android 中还有 Binder 等，那么我们这里采用哪种方式比较好呢？首先想到的是既然 dexopt 进程结束了自然在主进程的死循环中去判断 dexopt 进程是否存在。但是在实际操作中发现，dexopt 虽然已经退出了，但是进程并没有马上被回收掉，所以这个方法走不通。那么用 Broadcast 广播可以吗？可是可以，但是增加了 Application 的负担，在拉起 dexopt 进程前还得注册一个动态广播，接收到广播之后还得注销掉，所以这个也没有采用。那么最终采用的方式是判断文件是否存在，在拉起 dexopt 进程前在某个安全的地方建立一个临时文件，然后死循环判断这个文件是否存在，在 dexopt 进程结束的时候删除这个临时文件，那么在主进程的死循环中发现此文件不存在了，就直接跳出循环，继续 Application 初始化操作。

```
  public class NoteApplication extends Application {
  @Override
      protected void attachBaseContext(Context base) {
          super.attachBaseContext(base);
          //开启dex进程的话也会进入application
          if (isDexProcess()) {
              return;
          }
          doInstallBeforeLollipop();
          MultiDex.install(this);
      }
```

```
    @Override
    public void onCreate() {
        super.onCreate();
        if (isDexProcess()) {
            return;
        }
      //其他初始化
    }

    private void doInstallBeforeLollipop() {
        //满足3个条件，1.第一次安装开启，2.主进程，3.API<21(因为21之后ART的速度比dalvik快接近10倍(毕竟5.0之后的手机性能也要好很多))
        if (isAppFirstInstall() && !isDexProcessOrOtherProcesses() && Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
            try {
                createTempFile();
                startDexProcess();
                while (true) {
                    if (existTempFile()) {
                        try {
                            Thread.sleep(50);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    } else {
                        setAppNoteFirstInstall();
                        break;
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

总的来说，这种方式好处在于依赖集非常简单，同时它的集成方式也是非常简单，我们无须去修改与加载无关的代码。但是当没有启动过 App 的时候，被推送全家桶唤醒或者收到了广播，虽然这里都是没有界面的过程，但是运用了这种加载方式的话会弹出 dexopt 进程的 Activity，用户看到会一脸懵比的。

坑

Too many classes in –main-dex-list

> UNEXPECTED TOP-LEVEL EXCEPTION:com.android.dex.DexException: Too many classes in –main-dex-list, main dex capacity exceeded at com.android.dx.command.dexer.Main.processAllFiles(Main.java:494) at com.android.dx.command.dexer.Main.runMultiDex(Main.java:332) at com.android.dx.command.dexer.Main.run(Main.java:243) at com.android.dx.command.dexer.Main.main(Main.java:214) at com.android.dx.command.Main.main(Main.java:106)

通过 sdk 的 `mainDexClasses.rules` 知道主 dex 里面会有 Application、Activity、Service、Receiver、Provider、Instrumentation、BackupAgent 和 Annotation。当这些类以及直接引用类比较多的时候，都要塞进主 dex ，就引发了 main dex capacity exceeded build error 。

为了解决这个问题，当执行 `Create{flavor}{buildType}ManifestKeepList` task 之前将其中的 `activity` 去掉，之后会发现 /build/intermediates/multi_dex/{flavor}/{buildType}/manifest_keep.txt 文件中已经没有 Activity 相关的类了。

```
def patchKeepSpecs() {
    def taskClass =
"com.android.build.gradle.internal.tasks.multidex.CreateManifestKeepList";
    def clazz = this.class.classLoader.loadClass(taskClass)
    def keepSpecsField = clazz.getDeclaredField("KEEP_SPECS")
    keepSpecsField.setAccessible(true)
    def keepSpecsMap = (Map) keepSpecsField.get(null)
    if (keepSpecsMap.remove("activity") != null) {
        println "KEEP_SPECS patched: removed 'activity' root"
    } else {
        println "Failed to patch KEEP_SPECS: no 'activity' root found"
    }
}

patchKeepSpecs()
```

Too many classes in –main-dex-list

没错，还是 Too many classes in –main-dex-list 的错误。在美团的自动拆包中讲到：

> 实际应用中我们还遇到另外一个比较棘手的问题， 就是Field的过多的问题，Field过多是由我们目前采用的代码组织结构引入的，我们为了方便多业务线、多团队并发协作的情况下开发，我们采用的aar的方式进行开发，并同时在aar依赖链的最底层引入了一个通用业务aar，而这个通用业务aar中包含了很多资源，而ADT14以及更高的版本中对Library资源处理时，Library的R资源不再是static final的了，这样在最终打包时Library中的R没法做到内联，这样带来了R field过多的情况，导致需要拆分多个Secondary DEX，为了解决这个问题我们采用的是在打包过程中利用脚本把Libray中R field（例如ID、Layout、Drawable等）的引用替换成常量，然后删去Library中R.class中的相应Field。

DexException: Library dex files are not supported in multi-dex mode

> com.android.dex.DexException: Library dex files are not supported in multi-dex mode
> at com.android.dx.command.dexer.Main.runMultiDex(Main.java:322)
> at com.android.dx.command.dexer.Main.run(Main.java:228)
> at com.android.dx.command.dexer.Main.main(Main.java:199)
> at com.android.dx.command.Main.main(Main.java:103)

解决：

```
android {
    dexOptions {
        preDexLibraries = false
    }
}
```

OutOfMemoryError: Java heap space

> UNEXPECTED TOP-LEVEL ERROR:
> java.lang.OutOfMemoryError: Java heap space

解决：

```
android {
    dexOptions {
        javaMaxHeapSize "2g"
    }
}
```

MultiDex.install(Context)

回过头来我们看看 `MultiDex.install()` 做了一些什么：

```
public final class MultiDex {

    static {
        //secondary-dexes的位置：/code_cache/secondary-dexes
        SECONDARY_FOLDER_NAME = "code_cache" + File.separator + "secondary-
dexes";
        installedApk = new HashSet();
        IS_VM_MULTIDEX_CAPABLE =
isVMMultidexCapable(System.getProperty("java.vm.version"));
    }

    public static void install(Context context) {
        Log.i("MultiDex", "install");
        if(IS_VM_MULTIDEX_CAPABLE) {//针对ART
            Log.i("MultiDex", "VM has multidex support, MultiDex support library
is disabled.");
        } else if(VERSION.SDK_INT < 4) {
            throw new RuntimeException("Multi dex installation failed. SDK " +
VERSION.SDK_INT + " is unsupported. Min SDK version is " + 4 + ".");
        } else {
            try {
                ApplicationInfo e = getApplicationInfo(context);
                if(e == null) {
                    return;
                }

                Set var2 = installedApk;
                synchronized(installedApk) {
                    String apkPath = e.sourceDir;
                    //installedApk的类型是：Set<String>，如果这个apk已经安装，则不重复
安装
                    if(installedApk.contains(apkPath)) {
                        return;
                    }

                    installedApk.add(apkPath);
                    if(VERSION.SDK_INT > 20) {
```

```
                        Log.w("MultiDex", "MultiDex is not guaranteed to work in
SDK version " + VERSION.SDK_INT + ": SDK version higher than " + 20 + " should be
backed by " + "runtime with built-in multidex capabilty but it\'s not the " +
"case here: java.vm.version=\"" + System.getProperty("java.vm.version") + "\"");
                    }
          //类加载器，PathClassLoader
                    ClassLoader loader;
                    try {
                        loader = context.getClassLoader();
                    } catch (RuntimeException var9) {
                        Log.w("MultiDex", "Failure while trying to obtain Context
class loader. Must be running in test mode. Skip patching.", var9);
                        return;
                    }

                    if(loader == null) {
                        Log.e("MultiDex", "Context class loader is null. Must be
running in test mode. Skip patching.");
                        return;
                    }

                    try {
                        //删除以前的dex
                        clearOldDexDir(context);
                    } catch (Throwable var8) {
                        Log.w("MultiDex", "Something went wrong when trying to
clear old MultiDex extraction, continuing without cleaning.", var8);
                    }
          //dex将会输出到/data/data/{packagename}/code_cache/secondary-dexes目录。

                    File dexDir = new File(e.dataDir, SECONDARY_FOLDER_NAME);
                    List files = MultiDexExtractor.load(context, e, dexDir,
false);
                  //blablabla......
                }
            } catch (Exception var11) {
                Log.e("MultiDex", "Multidex installation failure", var11);
                throw new RuntimeException("Multi dex installation failed (" +
var11.getMessage() + ").");
            }

            Log.i("MultiDex", "install done");
        }
    }
}
```

之前都是准备工作，然后进入 `MultiDexExtractor.load()` 看一下：

```
final class MultiDexExtractor {
  static List<File> load(Context context, ApplicationInfo applicationInfo, File
dexDir, boolean forceReload) throws IOException {
        //dexDir地址是/data/data/{packagename}/code_cache/secondary-dexes
        Log.i("MultiDex", "MultiDexExtractor.load(" + applicationInfo.sourceDir +
", " + forceReload + ")");
        //applicationInfo.sourceDir地址是/data/app/{packagename}-1.apk
        File sourceApk = new File(applicationInfo.sourceDir);
        //Crc校验
```

```
        long currentCrc = getZipCrc(sourceApk);
        List files;
        //isModified()判断apk是否被修改过
        if(!forceReload && !isModified(context, sourceApk, currentCrc)) {
            try {
                // 加载已经存在的文件，如果有的文件不存在，或者不是zip文件，则会抛出异常
                files = loadExistingExtractions(context, sourceApk, dexDir);
            } catch (IOException var9) {
                Log.w("MultiDex", "Failed to reload existing extracted secondary
dex files, falling back to fresh extraction", var9);
                // 从apk中提取出多dex，然后将这些dex逐个打包为zip文件，最终返回提取出来的
zip文件列表
                files = performExtractions(sourceApk, dexDir);
        // getTimeStamp方法中调用的是sourceApk.lastModified()方法
                // putStoredApkInfo方法存储apk的信息：时间戳、crc值、apk中dex的总个数
                putStoredApkInfo(context, getTimeStamp(sourceApk), currentCrc,
files.size() + 1);
            }
        } else {
            Log.i("MultiDex", "Detected that extraction must be performed.");
            files = performExtractions(sourceApk, dexDir);
            putStoredApkInfo(context, getTimeStamp(sourceApk), currentCrc,
files.size() + 1);
        }

        Log.i("MultiDex", "load found " + files.size() + " secondary dex
files");
        return files;
    }

    private static boolean isModified(Context context, File archive, long
currentCrc) {
        SharedPreferences prefs = getMultiDexPreferences(context);
        return prefs.getLong("timestamp", -1L) != getTimeStamp(archive) ||
prefs.getLong("crc", -1L) != currentCrc;
    }
}
```

来看一下 `performExtractions()`：

```
final class MultiDexExtractor {
  private static List<File> performExtractions(File sourceApk, File dexDir)
throws IOException {
        //extractedFilePrefix的内容是{packagename}-1.apk.classes
        String extractedFilePrefix = sourceApk.getName() + ".classes";
        //准备文件夹
        prepareDexDir(dexDir, extractedFilePrefix);
        ArrayList files = new ArrayList();
        ZipFile apk = new ZipFile(sourceApk);

        try {
            int e = 2;

            for(ZipEntry dexFile = apk.getEntry("classes" + e + ".dex"); dexFile
!= null; dexFile = apk.getEntry("classes" + e + ".dex")) {//那第e个dex
                //{packagename}-1.apk.classes2.zip
                String fileName = extractedFilePrefix + e + ".zip";
```

```
                //extractedFile地址
是/data/data/{packagename}/code_cache/secondary-dexes/com.yydcdut.note-
1.apk.classes2.zip
                File extractedFile = new File(dexDir, fileName);
                files.add(extractedFile);
                Log.i("MultiDex", "Extraction is needed for file " +
extractedFile);
                int numAttempts = 0;
                boolean isExtractionSuccessful = false;

                while(numAttempts < 3 && !isExtractionSuccessful) {
                    ++numAttempts;
                    //提取apk中的dex文件，然后打包成一个zip文
                    extract(apk, dexFile, extractedFile, extractedFilePrefix);
                    //验证提取的文件是否是一个zip文件。
                    isExtractionSuccessful = verifyZipFile(extractedFile);
                    Log.i("MultiDex", "Extraction " +
(isExtractionSuccessful?"success":"failed") + " - length " +
extractedFile.getAbsolutePath() + ": " + extractedFile.length());
                    if(!isExtractionSuccessful) {
                        extractedFile.delete();
                        if(extractedFile.exists()) {
                            Log.w("MultiDex", "Failed to delete corrupted
secondary dex \'" + extractedFile.getPath() + "\'");
                        }
                    }
                }

                if(!isExtractionSuccessful) {
                    throw new IOException("Could not create zip file " +
extractedFile.getAbsolutePath() + " for secondary dex (" + e + ")");
                }

                ++e;
            }
        } finally {
            try {
                apk.close();
            } catch (IOException var16) {
                Log.w("MultiDex", "Failed to close resource", var16);
            }

        }

        return files;
    }

    private static void prepareDexDir(File dexDir, final String
extractedFilePrefix) throws IOException {
        //dexDir的地址是：/data/data/{packagename}/code_cache/secondary-dexes
        //extractedFilePrefix的内容是data/app/{packagename}-1.apk.classes
        File cache = dexDir.getParentFile();
        //创建文件夹
        mkdirChecked(cache);
        mkdirChecked(dexDir);
        FileFilter filter = new FileFilter() {
            public boolean accept(File pathname) {
                return !pathname.getName().startsWith(extractedFilePrefix);
```

```
            }
        };
        File[] files = dexDir.listFiles(filter);
        if(files == null) {
            Log.w("MultiDex", "Failed to list secondary dex dir content (" +
dexDir.getPath() + ").");
        } else {
            File[] arr$ = files;
            int len$ = files.length;

            for(int i$ = 0; i$ < len$; ++i$) {
                File oldFile = arr$[i$];
                Log.i("MultiDex", "Trying to delete old file " +
oldFile.getPath() + " of size " + oldFile.length());
                if(!oldFile.delete()) {
                    Log.w("MultiDex", "Failed to delete old file " +
oldFile.getPath());
                } else {
                    Log.i("MultiDex", "Deleted old file " + oldFile.getPath());
                }
            }

        }
    }
}
```

自习看一下 `extract()` 做的事:

```
final class MultiDexExtractor {
  private static void extract(ZipFile apk, ZipEntry dexFile, File extractTo,
String extractedFilePrefix) throws IOException, FileNotFoundException {
        InputStream in = apk.getInputStream(dexFile);
        ZipOutputStream out = null;
        //创建临时文件，地址/data/data/{packagename}/code_cache/secondary-
dexes/{packagename}-1.apk.classes941893003.zip
        File tmp = File.createTempFile(extractedFilePrefix, ".zip",
extractTo.getParentFile());
        Log.i("MultiDex", "Extracting " + tmp.getPath());
    //变zip
        try {
            out = new ZipOutputStream(new BufferedOutputStream(new
FileOutputStream(tmp)));

            try {
                ZipEntry classesDex = new ZipEntry("classes.dex");
                classesDex.setTime(dexFile.getTime());
                out.putNextEntry(classesDex);
                byte[] buffer = new byte[16384];

                for(int length = in.read(buffer); length != -1; length =
in.read(buffer)) {
                    out.write(buffer, 0, length);
                }

                out.closeEntry();
            } finally {
                out.close();
```

```
            }
      //extractTo的地址是 /data/data/{packagename}/code_cache/secondary-
dexes/{packagename}-1.apk.classes2.zip
            Log.i("MultiDex", "Renaming to " + extractTo.getPath());
            //重明明
            if(!tmp.renameTo(extractTo)) {
                throw new IOException("Failed to rename \"" +
tmp.getAbsolutePath() + "\" to \"" + extractTo.getAbsolutePath() + "\"");
            }
        } finally {
            closeQuietly(in);
            tmp.delete();
        }

    }
}
```

执行完这里，`List files = MultiDexExtractor.load(context, e, dexDir, false);`也就执行完了，继续看`MultiDex.install()`：

```
public final class MultiDex {


    public static void install(Context context) {
        Log.i("MultiDex", "install");
        if(IS_VM_MULTIDEX_CAPABLE) {//针对ART
            Log.i("MultiDex", "VM has multidex support, MultiDex support library
is disabled.");
        } else if(VERSION.SDK_INT < 4) {
            throw new RuntimeException("Multi dex installation failed. SDK " +
VERSION.SDK_INT + " is unsupported. Min SDK version is " + 4 + ".");
        } else {
            try {
                  //blablabla......
          //dex将会输出到/data/data/{packagename}/code_cache/secondary-dexes目录。

                    File dexDir = new File(e.dataDir, SECONDARY_FOLDER_NAME);
                    List files = MultiDexExtractor.load(context, e, dexDir,
false);
          //校验这些zip文件是否合法
                    if(checkValidZipFiles(files)) {
                        //安装提取出来的zip文件
                        installSecondaryDexes(loader, dexDir, files);
                    } else {//不合法的情况下强制load一遍
                        Log.w("MultiDex", "Files were not valid zip files.
Forcing a reload.");
                         //最后一个参数是true，代表强制加载
                        files = MultiDexExtractor.load(context, e, dexDir,
true);
                        //还是不合法的就抛异常
                        if(!checkValidZipFiles(files)) {
                            throw new RuntimeException("Zip files were not
valid.");
                        }
            //终于合法了，安装zip文件
                        installSecondaryDexes(loader, dexDir, files);
                    }
```

```
                }
            } catch (Exception var11) {
                Log.e("MultiDex", "Multidex installation failure", var11);
                throw new RuntimeException("Multi dex installation failed (" +
var11.getMessage() + ").");
            }

            Log.i("MultiDex", "install done");
        }
    }
}
```

`installSecondaryDexes(loader, dexDir, files)` 比较重要，在这个方法里面进行将第二个 dex 的代码加载到程序中。

```
public final class MultiDex {
  private static void installSecondaryDexes(ClassLoader loader, File dexDir,
List<File> files) throws IllegalArgumentException, IllegalAccessException,
NoSuchFieldException, InvocationTargetException, NoSuchMethodException,
IOException {
        //分版本加载
        if(!files.isEmpty()) {
            if(VERSION.SDK_INT >= 19) {
                MultiDex.V19.install(loader, files, dexDir);
            } else if(VERSION.SDK_INT >= 14) {
                MultiDex.V14.install(loader, files, dexDir);
            } else {
                MultiDex.V4.install(loader, files);
            }
        }
    }
}
```

来看看 `MultiDex.V14.install(loader, files, dexDir);`：

```
public final class MultiDex {
    private static final class V14 {
    private static void install(ClassLoader loader, List<File>
additionalClassPathEntries, File optimizedDirectory) throws
IllegalArgumentException, IllegalAccessException, NoSuchFieldException,
InvocationTargetException, NoSuchMethodException {
            //optimizedDirectory地址
是/data/data/{packagename}/code_cache/secondary-dexes
            //反射找到PathClassLoader的pathList属性，是DexPathList类，这个属性是父类
BaseDexClassLoader
            Field pathListField = MultiDex.findField(loader, "pathList");
            Object dexPathList = pathListField.get(loader);
            //dexElements为dexPathList的一个属性，是Element数组
            MultiDex.expandFieldArray(dexPathList, "dexElements",
makeDexElements(dexPathList, new ArrayList(additionalClassPathEntries),
optimizedDirectory));
        }

    private static Object[] makeDexElements(Object dexPathList, ArrayList<File>
files, File optimizedDirectory) throws IllegalAccessException,
InvocationTargetException, NoSuchMethodException {
```

```
            //反射拿到dexPathList的方法makeDexElements，private static Element[]
makeDexElements(ArrayList<File> files, File optimizedDirectory,
ArrayList<IOException> suppressedExceptions)
            Method makeDexElements = MultiDex.findMethod(dexPathList,
"makeDexElements", new Class[]{ArrayList.class, File.class});
            //调用方法，该方法的作用是通过传入的files去加载jar或者zip，封装成DexFile，在封
装成Element返回
            return (Object[])((Object[])makeDexElements.invoke(dexPathList, new
Object[]{files, optimizedDirectory}));
        }
  }

  private static void expandFieldArray(Object instance, String fieldName,
Object[] extraElements) throws NoSuchFieldException, IllegalArgumentException,
IllegalAccessException {
        //找到dexElements这个属性
        Field jlrField = findField(instance, fieldName);
        //classloader中原始的dexElements
        Object[] original = (Object[])((Object[])jlrField.get(instance));
        //new一个新的出来
        Object[] combined = (Object[])
((Object[])Array.newInstance(original.getClass().getComponentType(),
original.length + extraElements.length));
        //将原有的复制到新的里面去
        System.arraycopy(original, 0, combined, 0, original.length);
        //将第二个dex的复制到新的里面去
        System.arraycopy(extraElements, 0, combined, original.length,
extraElements.length);
        //再塞回去
        jlrField.set(instance, combined);
    }
}
```

至此，`MultiDex.install()` 分析完了，这里只讲一下V14的，在 Java 层的主要流程将第二个 dex 取出（现在只考虑两个 dex 的情况），整成 Zip 形式的，然后通过反射将 zip 的地址等参数封装起来再塞给 PathClassLoader 。为什么是 Zip ， 因为在 BaseDexClassLoader 中 `DexFile.loadDex()` 只接受 jar 或者 zip。

Jack

通过 Experimental New Android Tool Chain - Jack and Jill 查看到 build tools 21.1.1 开始就支持 `Jack` 了。

那么我们直接使用文章给出的 jack 在 gradle 中的用法使用编译工程吧：

```
buildscript {
    //...
    dependencies {
        classpath 'com.android.tools.build:gradle:2.1.0-alpha3'
    }
}
android {
    //...
    buildToolsVersion '21.1.2'
    minSdkVersion 21
    defaultConfig {
```

```
    // Enable the experimental Jack build tools.
    useJack = true
    }
    //...
  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }
}
```

但是在实际编译过程中出现了问题：

> Error:Execution failed for task ':app:jillDebugPackagedLibraries'.
> Jack requires Build Tools 24.0.0 or later

那么把 build tool 更新到 `24.0.0rc1` 再试试吧：

```
android {
    ...
    buildToolsVersion '24.0.0rc1'
    ...
}
```

重新编译，然后当 gradle 运行到 `compileDebugJavaWithJack` 的时候：

> ERROR: Dex writing phase: classes.dex has too many IDs. Try using multi-dex
> com.android.jack.api.v01.CompilationException: Dex writing phase: classes.dex has too many IDs. Try using multi-dex
> at com.android.jack.api.v01.impl.Api01ConfigImpl$Api01CompilationTaskImpl.run(Api01ConfigImpl.java:113)
> at com.android.builder.core.AndroidBuilder.convertByteCodeUsingJackApis(AndroidBuilder.java:1904)
> at com.android.build.gradle.tasks.JackTask.doMinification(JackTask.java:148)
> at com.android.build.gradle.tasks.JackTask.access$000(JackTask.java:73)
> at com.android.build.gradle.tasks.JackTask$1.run(JackTask.java:112)
> at com.android.builder.tasks.Job.runTask(Job.java:51)
> at com.android.build.gradle.tasks.SimpleWorkQueue$EmptyThreadContext.runTask(SimpleWorkQueue.java:41)
> at com.android.builder.tasks.WorkQueue.run(WorkQueue.java:223)
> at java.lang.Thread.run(Thread.java:745)
> Caused by: com.android.jack.JackAbortException: Dex writing phase: classes.dex has too many IDs. Try using multi-dex
> at com.android.jack.backend.dex.DexFileWriter.run(DexFileWriter.java:90)
> at com.android.jack.backend.dex.DexFileWriter.run(DexFileWriter.java:41)
> at com.android.sched.scheduler.ScheduleInstance.runWithLog(ScheduleInstance.java:161)
> at com.android.sched.scheduler.MultiWorkersScheduleInstance$SequentialTask.process(MultiWorkersScheduleInstance.java:442)
> at com.android.sched.scheduler.MultiWorkersScheduleInstance$Worker.run(MultiWorkersSch

> eduleInstance.java:162)
> Caused by: com.android.jack.backend.dex.DexWritingException: Dex writing phase: classes.dex has too many IDs. Try using multi-dex
> at com.android.jack.backend.dex.SingleDexWritingTool.write(SingleDexWritingTool.java:64)
> at com.android.jack.backend.dex.DexFileWriter.run(DexFileWriter.java:87)
> ... 4 more
> Caused by: com.android.jack.backend.dex.SingleDexOverflowException: classes.dex has too many IDs. Try using multi-dex
> ... 6 more
> Caused by: com.android.jack.tools.merger.MethodIdOverflowException: Method ID overflow when trying to merge dex files
> at com.android.jack.tools.merger.ConstantManager.addDexFile(ConstantManager.java:177)
> at com.android.jack.tools.merger.JackMerger.addDexFile(JackMerger.java:69)
> at com.android.jack.backend.dex.DexWritingTool.mergeDex(DexWritingTool.java:107)
> at com.android.jack.backend.dex.SingleDexWritingTool.write(SingleDexWritingTool.java:62)
> ... 5 more

还是得使用 multidex 。jack 不支持 `instantRun` ，同样 gradle 是 `2.1.0-alpha3` ，这里就需要 multidex 了。 65536 的根本问题并不在于 jack 身上，而在于指令集身上，jack 也是采用的 multidex 来解决这个问题的。