中国地质大学（北京）

本科毕业设计外文资料翻译

**院（系）： 信息工程**

**专 　　业：**  **软件工程**

**姓 名：** **王彬龙**

**学 号：** 1004126121

**外文出处：** Configure Apps with Over 64K

(用外文写)

**附 件：** 1.外文资料翻译译文；2.外文原文。

**完成日期：** 　2016 年　2　月　23　日

**突破65k限制**

目录

[错误出现 2](#_Toc445311350)

[65k限制 2](#_Toc445311351)

[5.0之前的Multidex 2](#_Toc445311352)

[5.0之后的Multidex 3](#_Toc445311353)

[避免65k限制 3](#_Toc445311354)

[配置Multidex通过gradle 3](#_Toc445311355)

[Multidex支持库的限制 4](#_Toc445311356)

[优化Multidex开发版本 5](#_Toc445311357)

[Build Variants 6](#_Toc445311358)

[测试 Multidex App 7](#_Toc445311359)

# 错误出现

伴随着Android平台的继续增长，Android应用程序的尺寸也在变大。当你的应用程序或者引用库达到一定规模，你会碰到一个build错误：你的应用到达了Android应用构架架构的最大限制。早期版本构建系统报告如下错误：

Conversion to Dalvik format failed:  
Unable to execute dex: method ID not in [0, 0xffff]: 65536

最近版本构建系统会报告不同的错误：

trouble writing output:  
Too many field references: 131000; max is 65536.  
You may try using --multi-dex option.

这两种错误显示一个共同的数字：65536。这个数字表示在一个Dalvik Executable(dex)字节码可以调用的引用总数。如果你创建一个app，并收到这个错误。首先恭喜你，你有很多代码！接下来帮你突破这个限制，继续构建应用程序。

# 65k限制

Android应用程序包含的可执行文件Dex,其中包含用于运行你的应用程序的可执行字节码文件。Dalvik可执行规范限制了可以在单个dex文件内引用65536：包括Android框架方法，库方法，并在自己的代码中实现的方法总数。突破这个限制需要配置你的应用程序构建过程，生成多个dex文件，被称为multidex配置。

# 5.0之前的Multidex

5.0之前的版本使用Dalvik运行时执行应用程序代码。默认情况下，Dalvik限制应用中每个apk的classes.dex字节码文件。为了解决这个限制，可以使用multidex支持库，成为你的应用程序主dex文件的一部分。然后设法获得额外的dex文件和它们包含的代码。

# 5.0之后的Multidex

5.0之后的版本使用ART运行时执行应用程序代码。ART本身支持应用程序apk文件加载多个Dex文件。ART在应用程序执行预编译安装时它会扫描所有的dex文件，并将其编译成由Android设备执行的单个文件：.oat文件。

# 避免65k限制

在配置您的应用程序，以便使用65K或以上方法的引用，您应该采取措施，以减少你的应用程序代码调用引用的总次数，包括您的应用程序代码或包含的库定义的方法。以下策略可以帮助您避免击中DEX参考限值：

* 查看应用的直接和传递依赖 - 确保依赖大的library胜过代码直接被添加到应用程序中使用。一个常见的​​反模式是包含一个非常大的library，只有很少的方法是有用的。减少你的应用程序代码依赖性往往可以帮助你避免DEX参考限值。
* 使用ProGuard删除未使用的代码-为您的应用程序配置ProGuard的设置来运行ProGuard，并将其确定您已经缩水的版本。确保你不发布未使用的代码在APK中 。

使用这些方法可以使你的应用减少引用。可以降低APK 的尺寸，这对其中带宽成本很高的市场是很重要的。

# 配置Multidex通过gradle

在Android SDK中为gradle提供的构建工具 Android SDK Build Tools 21.1和更高版本支持multidex作为构建配置的一部分。在配置之前，请确保更新了Android SDK Build Tools和Android Support Repository 到最新的版本。

设置应用开发项目中使用multidex配置要求做一些修改应用程序开发项目。特别需要执行以下步骤：

* 改变你的gradle构建配置multidex
* 修改你的manifest文件

修改gradle的配置，包括支持库。 在gradle构建文件中指定multiDexEnabled，可以在defaultConfig ，buildType或productFlavor节设置。

android {  
    compileSdkVersion 21  
    buildToolsVersion "21.1.0"  
  
    defaultConfig {  
        ...  
        minSdkVersion 14  
        targetSdkVersion 21  
        ...  
  
        // Enabling multidex support.  
        multiDexEnabled true  
    }  
    ...  
}  
  
dependencies {  
  compile 'com.android.support:multidex:1.0.0'  
}

在mainfest文件中添加MultidexApplication

<?xml version="1.0" encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.example.android.multidex.myapplication">  
    <application  
        ...  
        android:name="android.support.multidex.MultiDexApplication">  
        ...  
    </application>  
</manifest>

当这些配置设置添加到应用程序， Android编译工具会构建一个主DEX （ classes.dex ）及其配套（ classes2.dex ， classes3.dex ） 。然后，构建系统将它们打包成APK文件。

如果应用程序使用扩展的Application，则可以覆盖attachBaseContext （）方法，并调用MultiDex.install （this） ，以使multidex 。欲了解更多信息，请参阅MultiDexApplication参考文档。

# Multidex支持库的限制

multidex支持库有一些已知的限制，当你将其纳入应用程序构建配置，你应该了解和测试。

* .dex文件启动期间安装到设备的数据分区是复杂的，如果二次DEX文件是大的，可能导致ANR。在这种情况下，你应该使用代码缩减技术使用ProGuard尽量减少DEX文件的大小，并删除代码的未使用的部分。
* 早于Android 4.0的（API级别14）平台版本的设备启动应用程序,使用multidex可能无法运行由于Dalvik的linearAlloc bug（ issue 22586）。如果你是早于14 API平台，确保执行与这些版本的平台测试您的应用程序可以在启动时的问题或当类特殊群体被加载。代码收缩可以减少或可能消除这些潜在的问题。
* 使用multidex配置过的应用，非常大的内存分配请求可能在运行时由于Dalvik的linearAlloc limit（issue 78035 ）崩溃。此分配限制是在Android 4.0的（API级别14 ）增加，但应用仍有可能Android 5.0 （API级别21）之前碰上在Android版本中此限制。
* 很多复杂的需求包括：在基础的dex运行在Dalvik中需要的类文件。Android编译工具的更新处理Android的要求，但它可能是其他包括library有额外的依赖性要求，包括使用内省或Java方法调用的本地代码。某些库可能无法使用，不可以指定必须包括在主DEX文件类，除非multidex构建工具的更新，，。

# 优化Multidex开发版本

一个multidex配置要求显著增加构建的处理时间，因为构建系统必须做出什么类必须包含在主DEX文件，哪些类可以包含在二次DEX文件复杂的决定。这意味着，multidex常规编译通常需要更长的时间，并有可能减缓你的开发过程。  
为了减少编译multidex时间，你应该创建使用Android插件gradle productFlavors的构建输出两个味道：一个开发flavors和产品flavors。  
对于开发味，设置一个最小的SDK版本21.此设置更快产生multidex输出采用ART-支持的格式。对于释放香味，设置符合你的实际的最低支持率的最低SDK版本。此设置生成APK multidex是与更多的设备兼容，但需要更长的时间来建立。

下面构建配置示例演示了如何设置这些味道在摇篮构建文件：

android {  
    productFlavors {  
        // Define separate dev and prod product flavors.  
        dev {  
            // dev utilizes minSDKVersion = 21 to allow the Android gradle plugin  
            // to pre-dex each module and produce an APK that can be tested on  
            // Android Lollipop without time consuming dex merging processes.  
            minSdkVersion 21  
        }  
        prod {  
            // The actual minSdkVersion for the application.  
            minSdkVersion 14  
        }  
    }  
          ...  
    buildTypes {  
        release {  
            runProguard true  
            proguardFiles getDefaultProguardFile('proguard-android.txt'),  
                                                 'proguard-rules.pro'  
        }  
    }  
}  
dependencies {  
  compile 'com.android.support:multidex:1.0.0'  
}

完成此配置更改后，它结合了dev productFlavor和debug buildType的属性devDebug变种。使用此目标创建了ProGuard的disable， multidex enabled应用程序启用，并设置的minSdkVersion到Android API 21级这些设置导致了Android gradle这个插件来做到以下几点:

* 将每个module（包括dependency）创建成单独的dex文件。这个过程叫做pre-dexing.
* 包括不加修改地在每个DEX 文件。
* 最重要的是，模块DEX文件将不会合并，所以长期运行计算来确定主DEX文件的内容是可以避免的。

这些设置会导致快速，增量编译，因为只有修改模块的DEX文件重新计算和重新包装成APK文件。从这些结果APK建立可用于仅5.0设备在Android上进行测试。然而，通过实现配置作为香味后，将保留执行正常的生成与释放，适当的最低水平SDK和ProGuard的设置的能力。  
还可以构建其它变体，包括一个prodDebug变种的构建，这需要更长的时间来建立，但可以用于测试开发之外。在所示的配置中， prodRelease变异将是最后的测试和发布版本。如果您在命令行中执行任务的gradle ，您可以使用带有DevDebug标准命令追加到结尾（例如./gradlew installDevDebug ） 。有关使用香精与摇篮任务的详细信息，请参阅Gradle Plugin用户指南。

你也可以提供一个自定义的清单，或者针对每种口味的自定义应用程序类，允许您使用支持库MultiDexApplication类，或调用MultiDex.install （）只对需要它的变种。

# 配置Build Varians

此页面建立在配置编译概述为您展示如何可以配置版本的变种，从一个单一的项目建立不同版本的应用程序，以及如何正确地管理你的依赖和签名配置。

每个构建变量代表了不同版本的应用程序，你可以建立的。他们用的是特定的规则相结合的设置，代码，并在构建种类和口味的产品配置资源的摇篮结果。虽然你没有直接配置构建变种，你做配置构建类型和产品风味形成它们。

例如， “演示”产品风味可以指定不同的功能和设备的要求，如自定义源代码，资源和最低API级别，而“调试”构筑类型采用不同的构建和打包设置，如调试选项和签名键。产生的构建变种是您的应用程序的“ demoDebug ”版本，它包含了配置和资源的组合包含在“演示”产品的风味， “调试”构筑类型，主/源集。

# 测试 Multidex App

当使用instrumentation测试 multidex应用程序，则需要额外的配置，以使测试仪器。由于代码在multidex应用类的位置不是一个单一的DEX文件内，instrumentation测试不正确除非配置为multidex运行。  
为了测试与仪器测试中multidex应用程序，从multidex测试支持库配置MultiDexTestRunner 。下面的示例的build.gradle文件演示了如何配置您的构建来使用这个测试运行：

android {  
  defaultConfig {  
      ...  
      testInstrumentationRunner "com.android.test.runner.MultiDexTestRunner"  
  }  
}

有了Android插件gradle版本低于1.1时，您需要添加以下依赖于multidex的instrument:

dependencies {  
    androidTestCompile('com.android.support:multidex-instrumentation:1.0.1') {  
         exclude group: 'com.android.support', module: 'multidex'  
    }         
}

你可以直接使用instrumentation测试运行的类或扩展，以满足您的测试需求。或者，您可以在这样现有的仪器仪表覆盖的onCreate ：

public void onCreate(Bundle arguments) {  
    MultiDex.install(getTargetContext());  
    super.onCreate(arguments);  
    ...  
}

目前不支持创建测试APK使用multidex的。

# Configure Apps with Over 64K Methods

In this document

[About the 64K Reference Limit](https://developer.android.com/studio/build/multidex.html" \l "about)

[Multidex support prior to Android 5.0](https://developer.android.com/studio/build/multidex.html" \l "mdex-pre-l)

[Multidex support for Android 5.0 and higher](https://developer.android.com/studio/build/multidex.html" \l "mdex-on-l)

[Avoiding the 64K Limit](https://developer.android.com/studio/build/multidex.html" \l "avoid)

[Configuring Your App for Multidex with Gradle](https://developer.android.com/studio/build/multidex.html" \l "mdex-gradle)

[Limitations of the multidex support library](https://developer.android.com/studio/build/multidex.html" \l "limitations)

[Optimizing Multidex Development Builds](https://developer.android.com/studio/build/multidex.html" \l "dev-build)

[Using Build Variants in Android Studio](https://developer.android.com/studio/build/multidex.html" \l "variants-studio)

[Testing Multidex Apps](https://developer.android.com/studio/build/multidex.html" \l "testing)

See Also

[ProGuard](https://developer.android.com/tools/help/proguard.html)

As the Android platform has continued to grow, so has the size of Android apps. When your application and the libraries it references reach a certain size, you encounter build errors that indicate your app has reached a limit of the Android app build architecture. Earlier versions of the build system report this error as follows:

Conversion to Dalvik format failed:  
Unable to execute dex: method ID not in [0, 0xffff]: 65536

More recent versions of the Android build system display a different error, which is an indication of the same problem:

trouble writing output:  
Too many field references: 131000; max is 65536.  
You may try using --multi-dex option.

Both these error conditions display a common number: 65,536. This number is significant in that it represents the total number of references that can be invoked by the code within a single Dalvik Executable (dex) bytecode file. If you have built an Android app and received this error, then congratulations, you have a lot of code! This document explains how to move past this limitation and continue building your app.

Note: The guidance provided in this document supersedes the guidance given in the Android Developers blog post [Custom Class Loading in Dalvik](http://android-developers.blogspot.com/2011/07/custom-class-loading-in-dalvik.html).

About the 64K Reference Limit

Android application (APK) files contain executable bytecode files in the form of [Dalvik](https://source.android.com/devices/tech/dalvik/) Executable (DEX) files, which contain the compiled code used to run your app. The Dalvik Executable specification limits the total number of methods that can be referenced within a single DEX file to 65,536—including Android framework methods, library methods, and methods in your own code. In the context of computer science, the term *[Kilo, K](https://en.wikipedia.org/wiki/Kilo-)*, denotes 1024 (or 2^10). Because 65,536 is equal to 64 X 1024, this limit is referred to as the '64K reference limit'.

Getting past this limit requires that you configure your app build process to generate more than one DEX file, known as a multidex configuration.

Multidex support prior to Android 5.0

Versions of the platform prior to Android 5.0 (API level 21) use the Dalvik runtime for executing app code. By default, Dalvik limits apps to a single classes.dex bytecode file per APK. In order to get around this limitation, you can use the [multidex support library](https://developer.android.com/tools/support-library/features.html" \l "multidex), which becomes part of the primary DEX file of your app and then manages access to the additional DEX files and the code they contain.

Note: If your project is configured for multidex with minSdkVersion 20 or lower, and you deploy to target devices running Android 4.4 (API level 20) or lower, Android Studio disables [Instant Run](https://developer.android.com/tools/building/building-studio.html" \l "instant-run).

Multidex support for Android 5.0 and higher

Android 5.0 (API level 21) and higher uses a runtime called ART which natively supports loading multiple dex files from application APK files. ART performs pre-compilation at application install time which scans for classes(..N).dex files and compiles them into a single .oat file for execution by the Android device. For more information on the Android 5.0 runtime, see [Introducing ART](https://source.android.com/devices/tech/dalvik/art.html).

Note: While using [Instant Run](https://developer.android.com/tools/building/building-studio.html" \l "instant-run), Android Studio automatically configures your app for multidex when your app's minSdkVersion is set to 21 or higher. Because Instant Run only works with the debug version of your app, you still need to configure your release build for multidex to avoid the 64K limit.

Avoiding the 64K Limit

Before configuring your app to enable use of 64K or more method references, you should take steps to reduce the total number of references called by your app code, including methods defined by your app code or included libraries. The following strategies can help you avoid hitting the dex reference limit:

Review your app's direct and transitive dependencies - Ensure any large library dependency you include in your app is used in a manner that outweighs the amount of code being added to the application. A common anti-pattern is to include a very large library because a few utility methods were useful. Reducing your app code dependencies can often help you avoid the dex reference limit.

Remove unused code with ProGuard - Configure the [ProGuard](https://developer.android.com/tools/help/proguard.html) settings for your app to run ProGuard and ensure you have shrinking enabled for release builds. Enabling shrinking ensures you are not shipping unused code with your APKs.

Using these techniques can help you avoid the build configuration changes required to enable more method references in your app. These steps can also decrease the size of your APKs, which is particularly important for markets where bandwidth costs are high.

Configuring Your App for Multidex with Gradle

The Android plugin for Gradle available in Android SDK Build Tools 21.1 and higher supports multidex as part of your build configuration. Make sure you update the Android SDK Build Tools tools and the Android Support Repository to the latest version using the [SDK Manager](https://developer.android.com/tools/help/sdk-manager.html) before attempting to configure your app for multidex.

Setting up your app development project to use a multidex configuration requires that you make a few modifications to your app development project. In particular you need to perform the following steps:

Change your Gradle build configuration to enable multidex

Modify your manifest to reference the [MultiDexApplication](https://developer.android.com/reference/android/support/multidex/MultiDexApplication.html) class

Modify the module-level build.gradle file configuration to include the support library and enable multidex output, as shown in the following code snippet:

android {  
    compileSdkVersion 21  
    buildToolsVersion "21.1.0"  
  
    defaultConfig {  
        ...  
        minSdkVersion 14  
        targetSdkVersion 21  
        ...  
  
        // Enabling multidex support.  
        multiDexEnabled true  
    }  
    ...  
}  
  
dependencies {  
  compile 'com.android.support:multidex:1.0.0'  
}

In your manifest add the [MultiDexApplication](https://developer.android.com/reference/android/support/multidex/MultiDexApplication.html) class from the multidex support library to the application element.

<?xml version="1.0" encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.example.android.multidex.myapplication">  
    <application  
        ...  
        android:name="android.support.multidex.MultiDexApplication">  
        ...  
    </application>  
</manifest>

When these configuration settings are added to an app, the Android build tools construct a primary dex (classes.dex) and supporting (classes2.dex, classes3.dex) as needed. The build system will then package them into an APK file for distribution.

Note: If your app uses extends the [Application](https://developer.android.com/reference/android/app/Application.html) class, you can override the attachBaseContext() method and call MultiDex.install(this) to enable multidex. For more information, see the [MultiDexApplication](https://developer.android.com/reference/android/support/multidex/MultiDexApplication.html) reference documentation.

Limitations of the multidex support library

The multidex support library has some known limitations that you should be aware of and test for when you incorporate it into your app build configuration:

The installation of .dex files during startup onto a device's data partition is complex and can result in Application Not Responding (ANR) errors if the secondary dex files are large. In this case, you should apply code shrinking techniques with ProGuard to minimize the size of dex files and remove unused portions of code.

Applications that use multidex may not start on devices that run versions of the platform earlier than Android 4.0 (API level 14) due to a Dalvik linearAlloc bug (Issue [22586](http://b.android.com/22586)). If you are targeting API levels earlier than 14, make sure to perform testing with these versions of the platform as your application can have issues at startup or when particular groups of classes are loaded. Code shrinking can reduce or possibly eliminate these potential issues.

Applications using a multidex configuration that make very large memory allocation requests may crash during run time due to a Dalvik linearAlloc limit (Issue [78035](http://b.android.com/78035)). The allocation limit was increased in Android 4.0 (API level 14), but apps may still run into this limit on Android versions prior to Android 5.0 (API level 21).

There are complex requirements regarding what classes are needed in the primary dex file when executing in the Dalvik runtime. The Android build tooling updates handle the Android requirements, but it is possible that other included libraries have additional dependency requirements including the use of introspection or invocation of Java methods from native code. Some libraries may not be able to be used until the multidex build tools are updated to allow you to specify classes that must be included in the primary dex file.

Optimizing Multidex Development Builds

A multidex configuration requires significantly increased build processing time because the build system must make complex decisions about what classes must be included in the primary DEX file and what classes can be included in secondary DEX files. This means that routine builds performed as part of the development process with multidex typically take longer and can potentially slow your development process.

In order to mitigate the typically longer build times for multidex output, you should create two variations on your build output using the Android plugin for Gradle [productFlavors](http://tools.android.com/tech-docs/new-build-system/user-guide" \l "TOC-Product-flavors): a development flavor and a production flavor.

For the development flavor, set a minimum SDK version of 21. This setting generates multidex output much faster using the ART-supported format. For the release flavor, set a minimum SDK version which matches your actual minimum support level. This setting generates a multidex APK that is compatible with more devices, but takes longer to build.

The following build configuration sample demonstrates the how to set up these flavors in a Gradle build file:

android {  
    productFlavors {  
        // Define separate dev and prod product flavors.  
        dev {  
            // dev utilizes minSDKVersion = 21 to allow the Android gradle plugin  
            // to pre-dex each module and produce an APK that can be tested on  
            // Android Lollipop without time consuming dex merging processes.  
            minSdkVersion 21  
        }  
        prod {  
            // The actual minSdkVersion for the application.  
            minSdkVersion 14  
        }  
    }  
          ...  
    buildTypes {  
        release {  
            runProguard true  
            proguardFiles getDefaultProguardFile('proguard-android.txt'),  
                                                 'proguard-rules.pro'  
        }  
    }  
}  
dependencies {  
  compile 'com.android.support:multidex:1.0.0'  
}

After you have completed this configuration change, you can use the devDebug variant of your app, which combines the attributes of the devproductFlavor and the debug buildType. Using this target creates a debug app with proguard disabled, multidex enabled, and minSdkVersion set to Android API level 21. These settings cause the Android gradle plugin to do the following:

Build each module of the application (including dependencies) as separate dex files. This is commonly referred to as pre-dexing.

Include each dex file in the APK without modification.

Most importantly, the module dex files will not be combined, and so the long-running calculation to determine the contents of the primary dex file is avoided.

These settings result in fast, incremental builds, because only the dex files of modified modules are recomputed and repackaged into the APK file. The APK that results from these builds can be used to test on Android 5.0 devices only. However, by implementing the configuration as a flavor, you preserve the ability to perform normal builds with the release-appropriate minimum SDK level and proguard settings.

You can also build the other variants, including a prodDebug variant build, which takes longer to build, but can be used for testing outside of development. Within the configuration shown, the prodRelease variant would be the final testing and release version. If you are executing gradle tasks from the command line, you can use standard commands with DevDebug appended to the end (such as ./gradlew installDevDebug). For more information about using flavors with Gradle tasks, see the [Gradle Plugin User Guide](http://tools.android.com/tech-docs/new-build-system/user-guide).

Tip: You can also provide a custom manifest, or a custom application class for each flavor, allowing you to use the support library MultiDexApplication class, or calling MultiDex.install() only for the variants that need it.

Configure Build Variants

In this document

[Configure Build Types](https://developer.android.com/studio/build/build-variants.html" \l "build-types)

[Configure Product Flavors](https://developer.android.com/studio/build/build-variants.html" \l "product-flavors)

[Create Source Sets for Build Variants](https://developer.android.com/studio/build/build-variants.html" \l "sourcesets)

[Declare Dependencies](https://developer.android.com/studio/build/build-variants.html" \l "dependencies)

[Configure Signing Settings](https://developer.android.com/studio/build/build-variants.html" \l "signing)

See also

[Configure Your Build Overview](https://developer.android.com/studio/build/index.html)

[Manifest Merger](https://developer.android.com/studio/build/manifest-merge.html)

This page builds on the [Configure Your Build Overview](https://developer.android.com/studio/build/index.html) to show you how you can configure build variants to create different versions of your app from a single project, and how to properly manage your dependencies and signing configurations.

Each build variant represents a different version of your app that you can build. They are the result of Gradle using a [specific set of rules](https://developer.android.com/studio/build/build-variants.html" \l "sourceset-build) to combine settings, code, and resources configured in your build types and product flavors. Although you do not configure build variants directly, you do configure the build types and product flavors that form them.

For example, a "demo" product flavor can specify different features and device requirements, such as custom source code, resources, and minimum API levels, while the "debug" build type applies different build and packaging settings, such as debug options and signing keys. The resulting build variant is the "demoDebug" version of your app, and it includes a combination of the configurations and resources included in the "demo" product flavor, "debug" build type, and main/ source set.

Using Build Variants in Android Studio

Build variants can be very useful for managing the build process when using multidex. Android Studio allows you to select these build variants in the user interface.

To have Android Studio build the "devDebug" variant of your app:

Open the Build Variants window from the left-sidebar. The option is located next to Favorites.

Click the name of the build variant to select a different variant, as shown in Figure 1.

Figure 1. Screen shot of the Android Studio left panel showing a build variant.

Note: The option to open this window is only available after you have successfully synchronized Android Studio with your Gradle build file using theTools > Android > Sync Project with Gradle Files command.

Testing Multidex Apps

When using instrumentation tests with multidex apps, additional configuration is required to enable the test instrumentation. Because the location of code for classes in multidex apps is not within a single DEX file, instrumentation tests do not run properly unless configured for multidex.

To test a multidex app with instrumentation tests, configure the [MultiDexTestRunner](https://developer.android.com/reference/com/android/test/runner/MultiDexTestRunner.html) from the multidex testing support library. The following samplebuild.gradle file demonstrates how to configure your build to use this test runner:

android {  
  defaultConfig {  
      ...  
      testInstrumentationRunner "com.android.test.runner.MultiDexTestRunner"  
  }  
}

Note: With Android Plugin for Gradle versions lower than 1.1, you need to add the following dependency for multidex-instrumentation:

dependencies {  
    androidTestCompile('com.android.support:multidex-instrumentation:1.0.1') {  
         exclude group: 'com.android.support', module: 'multidex'  
    }  
}

You may use the instrumentation test runner class directly or extend it to fit your testing needs. Alternatively, you can override onCreate in existing instrumentations like this:

public void onCreate(Bundle arguments) {  
    MultiDex.install(getTargetContext());  
    super.onCreate(arguments);  
    ...  
}

Note: Use of multidex for creating a test APK is not currently supported.