---

title: Android gradle 3.0 插件之旅

date: 2017-10-29 12:00:21

tags: Android

top: 20

```
昨天，官网（https://developer.android.com/） 更新了
Androd studio 版本和 gradle-plugin 版本，迫不及待尝试下，
针对自己的项目做了些相应的修改，相比之前，这个版本的改动是最多
的，出来了许多新特性。下面总结下升级过程中遇到的坑。
```

1. 升级大概流程：

- 1）升级 gradle 版本：

```
distributionUrl=https\://services.gradle.org/distribu
tions/gradle-4.1-all.zip
```

- 2）修改 maven 仓库：

```
buildscript {
    repositories {
        google()
    }
}

allprojects {
    repositories {
        google()
    }
}
```

- 3）升级 gradle-plugin 插件版本：

```
dependencies {
        classpath
'com.android.tools.build:gradle:3.0.0'
    }
```

2. 踩到的坑以及插件变更的方面：

1) META-INF/MANIFEST.MF 问题：

```
* What went wrong:
Execution failed for task
':app:transformResourcesWithMergeJavaResForJdDebug'.
> More than one file was found with OS independent
path 'META-INF/MANIFEST.MF'
```

```
android {
        packagingOptions.excludes = [
                'META-INF/MANIFEST.MF'
        ]
    }
```

2) 依赖问题：

```
Could not resolve project :code:module:MdmBasicInfo.
```

```
       Required by:
           project :code:app:MdmApp
        > Unable to find a matching configuration of
project :code:module:MdmBasicInfo:
            - Configuration 'debugApiElements':
                - Required
com.android.build.api.attributes.BuildTypeAttr
'rtest' and found incompatible value 'debug'.
                - Required
com.android.build.gradle.internal.dependency.AndroidT
ypeAttr 'Aar' and found compatible value 'Aar'.
                - Found
com.android.build.gradle.internal.dependency.VariantA
ttr 'debug' but wasn't required.
                - Required
org.gradle.api.attributes.Usage 'java-api' and found
compatible value 'java-api'.
                - Required type_product 'jd' but no
value provided.
```

按照后面改进的依赖方式配置

3) AAPT2 问题：

```
> Task :app:MdmApp:processJdRtestResources
Failed to execute aapt
com.android.ide.common.process.ProcessException:
Failed to execute aapt

* What went wrong:
Execution failed for task
':app:MdmApp:processJdRtestResources'.
> Failed to execute aapt

* Try:
Run with --stacktrace option to get the stack trace.
Run with --info or --debug option to get more log
output.
```

在根项目的 gradle.properties 文件中添加:

```
android.enableAapt2=false
```

4) flavor dimension 必须设置:

```
Error:All flavors must now belong to a named flavor
dimension.
The flavor 'flavor_name' is not assigned to a flavor
dimension.
```

依赖机制改动：在使用 library 时会自动匹配 variant（debug、release)，即 app 的 debug 会自动匹配 library 的 debug；同样地，如果使用 flavor 的时候，比如 app 的 freeDebug 也会自动匹配 library 的 freeDebug。尽管如此，但在使用到 flavor 时，必须为所有的 flavor 配置一个 flavor dimension

```
// Specifies two flavor dimensions.
flavorDimensions "tier", "minApi"
productFlavors {
     free {
      // Assigns this product flavor to the "tier"
flavor dimension. Specifying
      // this property is optional if you are using
only one dimension.
      dimension "tier"
      ...
    }
```

```
        paid {
          dimension "tier"
          ...
        }

        minApi23 {
            dimension "minApi"
            ...
        }

        minApi18 {
            dimension "minApi"
            ...
        }
    }
```

6) 依赖配置：

解决依赖构建错误: 如果 app 依赖 library A, app 的 buildType 为 debug、rtest、release, 但是 library buildType 只有 debug、release，当构建 debug app 的时候，插件会自动去寻找 library rtest，最后会发现不到，错误信息如下：

```
Error:Failed to resolve: Could not resolve project
:mylibrary.
Required by:
    project :app
```

出现该错误的原因大致可以分为三类：

- app 存在的 buildType，但是 library 没有；通过 matchingFallbacks 去给 buildType 设置依赖：

```
// In the app's build.gradle file.
android {
    buildTypes {
        debug {}
        release {}
        staging {
            // Specifies a sorted list of fallback
build types that the
            // plugin should try to use when a
dependency does not include a
            // "staging" build type. You may specify
as many fallbacks as you
            // like, and the plugin selects the first
build type that's
            // available in the dependency.
            matchingFallbacks = ['debug', 'qa',
'release']
        }
    }
}
```

- 对于 app 和 library 都存在的 flavor dimension，app 中存在的 flavor，但是 library 没有；通过 matchingFallbacks 去给 flavor 设置依赖:

```
// In the app's build.gradle file.
android {
    defaultConfig{
    // Do not configure matchingFallbacks in the
defaultConfig block.
    // Instead, you must specify fallbacks for a
given product flavor in the
    // productFlavors block, as shown below.
  }
    flavorDimensions 'tier'
    productFlavors {
        paid {
            dimension 'tier'
            // Because the dependency already
includes a "paid" flavor in its
            // "tier" dimension, you don't need to
provide a list of fallbacks
            // for the "paid" flavor.
        }
        free {
            dimension 'tier'
            // Specifies a sorted list of fallback
flavors that the plugin
            // should try to use when a dependency's
matching dimension does
            // not include a "free" flavor. You may
specify as many
            // fallbacks as you like, and the plugin
selects the first flavor
            // that's available in the dependency's
"tier" dimension.
            matchingFallbacks = ['demo', 'trial']
        }
    }
}
```

- library 依赖包括的 flavor dimension，但是 app 没有：通过 missingDimensionStrategy 在 defaultConfig 中指定默认的 flavor，这样插件找不到 dimension 时，就会去使用默认的 flvor；同时也可以在 productFlavors 中为每个匹配指定默认的 flavor：

```
// In the app's build.gradle file.
android {
    defaultConfig{
    // Specifies a sorted list of flavors that the
plugin should try to use from
```

```
    // a given dimension. The following tells the
plugin that, when encountering
    // a dependency that includes a "minApi"
dimension, it should select the
    // "minApi18" flavor. You can include additional
flavor names to provide a
    // sorted list of fallbacks for the dimension.
    missingDimensionStrategy 'minApi', 'minApi18',
'minApi23'
    // You should specify a missingDimensionStrategy
property for each
    // dimension that exists in a local dependency
but not in your app.
    missingDimensionStrategy 'abi', 'x86', 'arm64'
    }
    flavorDimensions 'tier'
    productFlavors {
        free {
            dimension 'tier'
            // You can override the default selection
at the product flavor
            // level by configuring another
missingDimensionStrategy property
            // for the "minApi" dimension.
            missingDimensionStrategy 'minApi',
'minApi23', 'minApi18'
        }
        paid {}
    }
}
```

7) 本地模块以来配置：自动匹配 variant 的机制，无须手动指定 variant 的配置，而且不支持如下配置了：

```
dependencies {
    debugCompile project(path: ':lib:appstore',
configuration: 'debug')
}
```

可修改为以下配置：

```
dependencies {
    // This is the old method and no longer works for
local
    // library modules:
    // debugImplementation project(path:
':lib:appstore', configuration: 'debug')
    // releaseImplementation project(path:
':lib:appstore', configuration: 'release')

    // Instead, simply use the following to take
advantage of
    // variant-aware dependency resolution. You can
learn more about
    // the 'implementation' configuration in the
section about
    // new dependency configurations.
    implementation project(':lib:appstore')

    // You can, however, keep using variant-specific
configurations when
    // targeting external dependencies. The following
line adds 'app-magic'
    // as a dependency to only the "debug" version of
your module.

    debugImplementation
'ggg.android:cm.android.mdmsdk:1.77'
}
```

8) Gradle 3.4 引入了新的 Java 库插件配置, 允许您控制是否将依赖项发布到使用该库的项目的编译和运行时路径。Android 插件正在采用这些新的依赖配置, 迁移大型项目来使用它们可以大大缩短构建时间。

新配置	旧配置	概述
implementation	compile	module 编译时可用，但 module 的依赖者运行时可用；只在内部使用了该 module，不会向外部暴露其依赖的 module 内容
api	compile	module 编译时可用，module 的依赖者编译和运行时可用，和 compile 的作用一样，当前 module 会暴露其依赖的其他 module 内容
compileOnly	provided	module 编译时可用，但是 module 的依赖者，在编译和运行时均不可用，和之前的 provided 类似
runtimeOnly	apk	module 和 module 依赖者，仅仅在运行时可用

9) 发布依赖：如下配置保存了 library 的传递依赖性, 供其使用者使用

```
// 编译时使用
variant_nameApiElements

// 运行时使用
variant_nameRuntimeElements
```

10) 自定义依赖：如下配置来解决变量的所有依赖项

新配置(运行时)	旧配置(编译时)
variant_nameRuntimeClasspath	

	_variant_nameCompile 不再生效
variant_nameCompileClasspath	_variant_nameApk 不再生效

如果继续使用旧有的配置，则会抛出以下错误：

```
Error:Configuration with old name _debugCompile
found.
Use new name debugCompileClasspath instead.
```

由于新的依赖方式延迟了依赖 resolution, 因此可以使用 Variant API 来设置 resolution strategy：

```
// Previously, you had to apply a custom resolution
strategy during the configuration phase, rather than
in the execution phase. That's because, by the time
the variant was created and the Variant API was
called, the dependencies were already resolved. But
now these configurations DO NOT WORK with the 3.0.0
Gradle plugin:

// configurations {
//     _debugCompile
//     _debugApk
// }
//
// configurations._debugCompile.resolutionStrategy {
//     ...
// }
//
// configurations.all {
//     resolutionStrategy {
//     ...
//     }
// }
// Instead, because the new build model delays
dependency resolution, you should query and modify
the resolution strategy using the Variant API:
android {
    applicationVariants.all { variant ->

variant.getCompileConfiguration().resolutionStrategy
{
            ...
        }

variant.runtimeConfiguration.resolutionStrategy {
            ...
        }

variant.getAnnotationProcessorConfiguration().resolut
ionStrategy {
            ...
        }
    }
}
```

11) 从 test 配置中排除 app 依赖项

- 原有的排除依赖方式：

```
dependencies {
    implementation
"com.jakewharton.threetenabp:threetenabp:1.0.5"
    // Note: You can still use the exclude keyword to
omit certain artifacts of dependencies you add only
to your test configurations.

androidTestImplementation("org.threeten:threetenbp:1.
3.3") {
        exclude group: 'com.jakewharton.threetenabp',
module: 'threetenabp'
    }
}
```

- 新的排除依赖方式：这是因为 androidTestImplementation 和 androidTestApi 扩展了 Implementation 和 api 配置

```
android.testVariants.all { variant ->
    variant.getCompileConfiguration().exclude group:
'com.jakewharton.threetenabp', module: 'threetenabp'
    variant.getRuntimeConfiguration().exclude group:
'com.jakewharton.threetenabp', module: 'threetenabp'
}
```

12) 原有的 variant outputs 在构建时不再生效，不过新的插件仍然支持修改生成的 apk 的名称：

```
// If you use each() to iterate through the variant
objects,
// you need to start using all(). That's because
each() iterates
// through only the objects that already exist during
configuration time—
// but those object don't exist at configuration time
with the new model.
// However, all() adapts to the new model by picking
up object as they are
// added during execution.
android.applicationVariants.all { variant ->
    variant.outputs.all {
        outputFileName = "${variant.name}-
${variant.versionName}.apk"
    }
}
```

13) processManifest.manifestOutputFile() 方法已经废弃，如果继续使用，会抛出如下错误：

```
A problem occurred configuring project ':myapp'.
   Could not get unknown property
'manifestOutputFile' for task
':myapp:processDebugManifest'
   of type
com.android.build.gradle.tasks.ProcessManifest.
```

但是可以调用 processManifest.manifestOutputDirectory () 返回包含所有生成的 manifest 的目录的路径，然后可以根据需要修改某个 manifest：

```
android.applicationVariants.all { variant ->
    variant.outputs.all { output ->
        output.processManifest.doLast {
            // Stores the path to the maifest.
            String manifestPath =
"$manifestOutputDirectory/AndroidManifest.xml"
            // Stores the contents of the manifest.
            def manifestContent =
file(manifestPath).getText()
            // Changes the version code in the stored
text.
            manifestContent =
manifestContent.replace('android:versionCode="1"',

String.format('android:versionCode="%s"',
generatedCode))
            // Overwrites the manifest with the new
text.
            file(manifestPath).write(manifestContent)
        }
    }
}
```

14) 使用 annotation processor 依赖配置：

在之前的插件中，compile classpath 上的依赖项被自动添加到processor classpath 中。即你可以将 annotation processor添加到 compile classpath 中；但是，通过向 processor 添加大量不必要的依赖关系，这将对性能产生重大影响。

在 Android 3.0.0 插件中，规定必须使用 annotationProcessor 将依赖项配置将 annotation processors 添加到 processor classpath 中：

```
dependencies {
    ...
    annotationProcessor 'com.google.dagger:dagger-
compiler:<version-number>'
}
```

15) 禁用 annotation processor 错误检查：

```
android {
    ...
    defaultConfig {
        ...
        javaCompileOptions {
            annotationProcessorOptions {
                includeCompileClasspath false
            }
        }
    }
}
```

16) 使用独立的测试模块：

```
android {
    variantFilter { variant ->
        if (variant.buildType.name.equals('debug') {
            variant.setIgnore(true);
        }
    }
}
```

17) libraries 中的本地 jar 包可传递：

之前的插件在库模块中将以非标准方式处理本地 jar 的依赖关系，并将它们打包到 aar 中；新的插件3.0.0 和新的 Gradle api，支持本地 jar 可传递依赖性，类似于基于 maven 的依赖关系：

- 在 project 中发布：
    - library 模块不再处理本地 jar；
    - library 模块的改变只会影响 project，PROJECT_LOCAL_DEPS 已经废弃；
    - app 模块中的本地 jar 作为EXTERNAL stream 的一部分，PROJECT_LOCAL_DEPS 和 SUB_PROJECT_LOCAL_DEPS 总是为空；
    - library modules 中的 ProGuard 不会影响 library 代码，只需在依赖 library 的 app 中使用 ProGuard；
    - 之前，需要在 library 中解决和本地 jar 包中的资源冲突，现在需要在 app 中去解决冲突；

- Maven 仓库发布：这个和之前类似，没有改变；

18) AAPT2 的变化：

- Android manifest 中元素的层级：在之前的 aapt 中，Manifest 中不正确的层级会报警告或者被忽略，如下：

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/and
roid"
   package="com.example.myname.myapplication">
   <application
       ...
       <activity android:name=".MainActivity">
           <intent-filter>
               <action
android:name="android.intent.action.MAIN" />
               <category
android:name="android.intent.category.LAUNCHER" />
           </intent-filter>
           <action
android:name="android.intent.action.CUSTOM" />
       </activity>
   </application>
</manifest>
```

之前的 aapt 只会忽略 action，然而，在最新的 AAPT2 中，将会报错：

```
AndroidManifest.xml:15: error: unknown element
<action> found.
```

解决方法就是按照正确的语法结构去修改，确保 manifest 元素正确

- 资源定义：不支持从 name 属性中指定资源类型，如：

```
<style name="foo" parent="bar">
    <item name="attr/my_attr">@color/pink</item>
</style>
```

上述定义资源类型将会导致如下错误：

```
Error: style attribute 'attr/attr/my_attr (aka
my.package:attr/attr/my_attr)' not found.
```

解决方法：明确指定资源类型(type = "attr")

```
<style name="foo" parent="bar">
  <item type="attr" name="my_attr">@color/pink</item>
</style>
```

此外，定义 style 元素时，它的父元素也必须是 style 资源类型，否则会导致同样的错误：

```
Error: (...) invalid resource type 'attr' for parent
of style
```

- Android ForegroundLinearLayout 中的 namespace：ForegroundLinearLayout 包括三种 foregroundInsidePadding, android:foreground, and android:foregroundGravity 三种属性，注意 android namespace 不包括 foregroundInsidePadding；在之前的 AAPT 中，编译器会忽略 foregroundInsidePadding，但是在 AAPT2 中，则会抛出错误：

```
Error: (...) resource
android:attr/foregroundInsidePadding is private
```

解决方法：用 android:foregroundInsidePadding 替换 foregroundInsidePadding 即可：

- 不恰当地使用 @ 资源引用符号：检测到忽略或者不当使用 @ 时，AAPT2 会抛出错误，例如，当指定 style 属性时，遗漏了 @ 时

```
<style name="AppTheme"
parent="Theme.AppCompat.Light.DarkActionBar">
  ...
  <!-- Note the missing '@' symbol when specifying
the resource type. -->
  <item name="colorPrimary">color/colorPrimary</item>
</style>
```

将会导致如下错误：

```
ERROR: expected color but got (raw string)
color/colorPrimary
```

此外，如果不当的使用 @ 引用 android namespace 中的资源时，如下所示：

```
...
<!-- When referencing resources from the 'android'
namespace, omit the '@' symbol. -->
<item name="@android:windowEnterAnimation"/>
```

将会导致如下错误：

```
Error: style attribute
'@android:attr/windowEnterAnimation' not found
```

参考：
[1] https://developer.android.com/studio/build/gradle-plugin-3-0-0-migration.html#known_issues
[2] http://blog.csdn.net/ncuboy045wsq/article/details/73521856