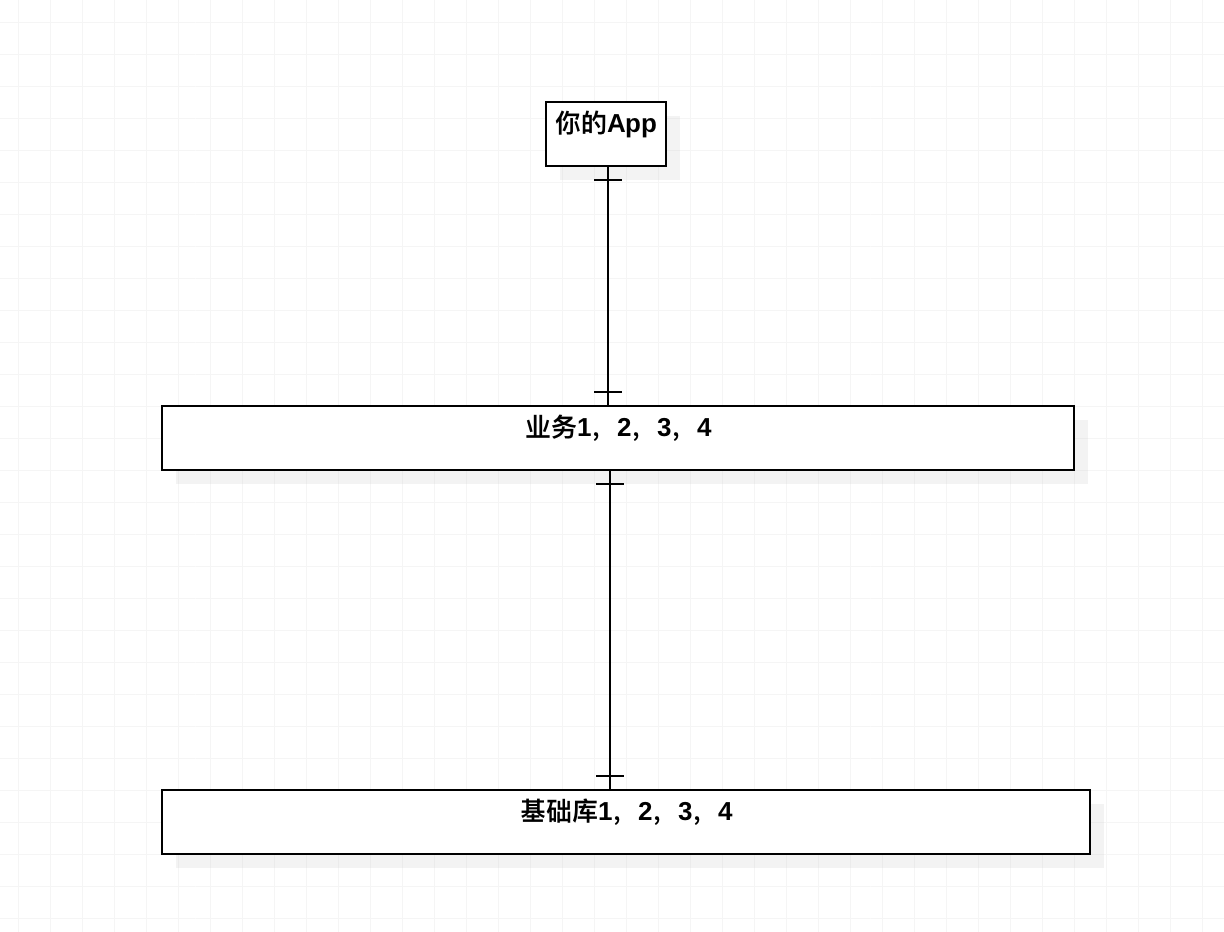
# 关于Android业务组件化的一些思考

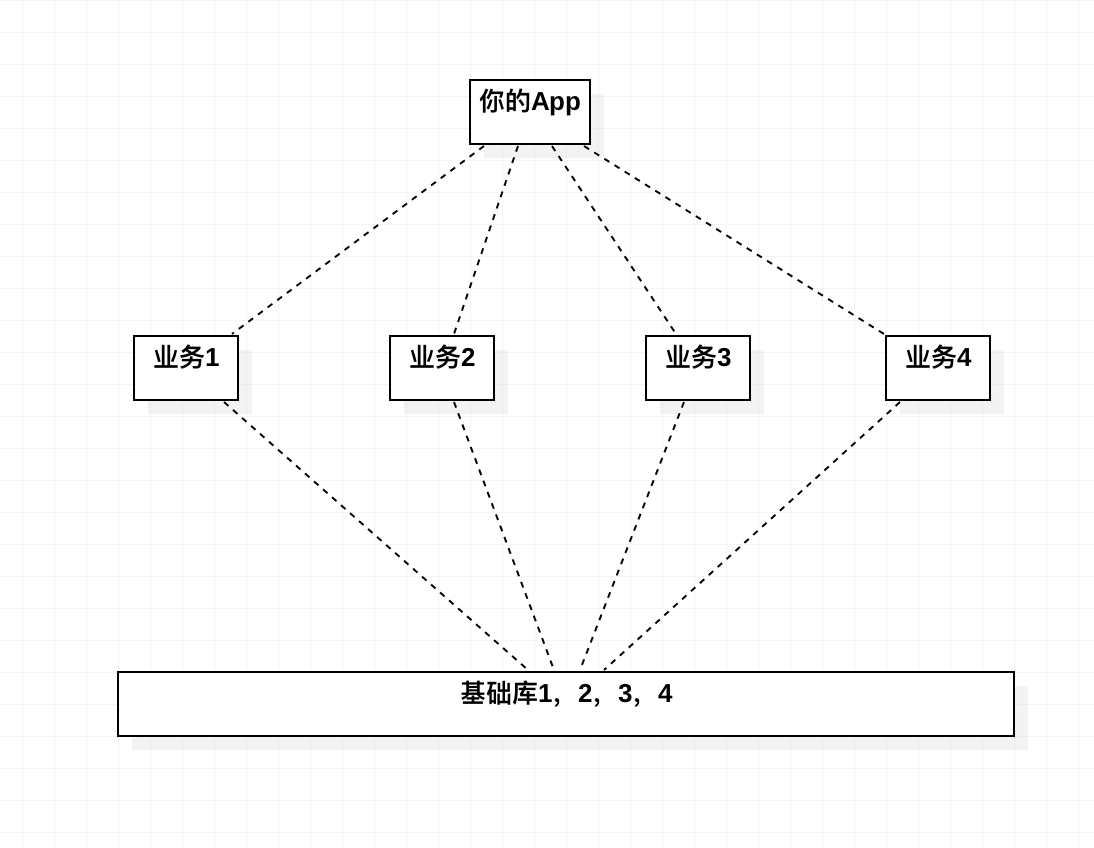
# **什么是组件化**

在说组件化之前，相信大家对Android上的插件化肯定或多或少都有一定的了解，插件化的出现，让我们的App的运行的时候能够动态的进行组装，就像我们使用chrome的插件一样，非常的方便，甚至演变到后期，插件化越来越像［虚拟机］发展，使用一个类似［boot］的壳，就可以像Java虚拟机加载Java文件一样加载一个App。

组件化可以说和插件化有异曲同工之妙，只不过插件化是在［运行时］，而组件化是在［编译时］。换句话说，插件化是基于多APK的，而组件化本质上还是只有一个APK。相信大家在平时工作中会有这样一个苦恼，那就是编译一个App的时间实在是太太太太太长了，特别是一个比较大的应用，可能2，3分钟都不够，对于一些耐心好的同学，可以去喝一杯咖啡，思考一下人生，而像我这样急性子的人嘛，简直就是噩梦，在等待编译的时候经常手足无措。［组件化］就可以很好的解决这样的问题，此外，由于整个App的各个业务被分离了，所以它们之间的耦合度也就被降低了，各个业务线可以由专门的开发同学进行开发，相互之间也不会有干扰，提升开发效率。

下面用两张图来辅助说明一下：





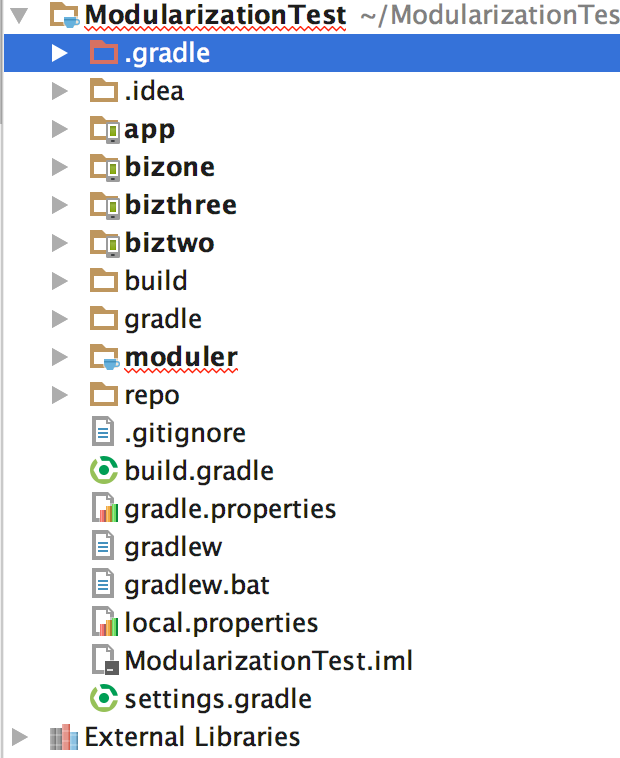
可以看到，在没有进行组件化改造的情况下，我们的App依赖的各条业务线是捆绑在一起的，根本没办法解耦，更别说模块的复用了，可以说整个App就是各条业务线混合在一起的一个大容器。

下面那张图是进行了组件化分离之后，各个业务线分离，逻辑变得清晰，每个业务线都可以成为另外一个业务线的上游或者下游。更重要的是，它们每一个都可以单独编译，缩减了编译的时间。也正因为这一点，各个业务线的研发也可以做到互相不干扰，加快了开发的速度。

# **如何进行组件化**

对于组件化的实践，之前的方式就是各个组件以AAR的形式输出，主App去依赖那些AAR，但是这种方式，对于开发人员来说成本会有一点大，主要是以AAR形式输出的话，在每次修改之后都要重新去打一次AAR，非常的麻烦，所以冯老师提出了一种新的方式，就是以Debug模式和Release模式去区分，在Debug模式下，各个业务线作为Application可以单独运行，而在Release模式下，则作为Library，可以提供给主App进行依赖。这样一来就可以做到每个业务线的平行开发，在Release模式下再合到一起，非常的灵活。

那具体应该如何实施呢？这里以一个demo为例，让我们先看看整个as工程的结构。



忽略那个moduler，它是一个gradle插件，可以看到主App就是app模块，而bizone，biztwo和bizthree代表的就是三条业务线，每一个都是一个独立的module。

然后，我们可以通过一个全局的变量去标记是Debug模式还是Release模式，当然大家也可以使用gradle的build type，但是我个人感觉还是自己设一个标记会好一点，因为这样更加灵活，对于一些混淆上的问题，我们可以通过设置Debug模式，再启用build type的release就可以去验证了。

|  |  |
| --- | --- |
| 1 2 3 4 5 | if (isDebug.toBoolean()) {  apply plugin: 'com.android.application' } else {  apply plugin: 'com.android.library' } |

通过在业务module的gradle配置文件中判断是否是Debug模式去区分是Application还是Library，这和之前提供的观点是一致的，也是这种组件化方式最核心的地方。

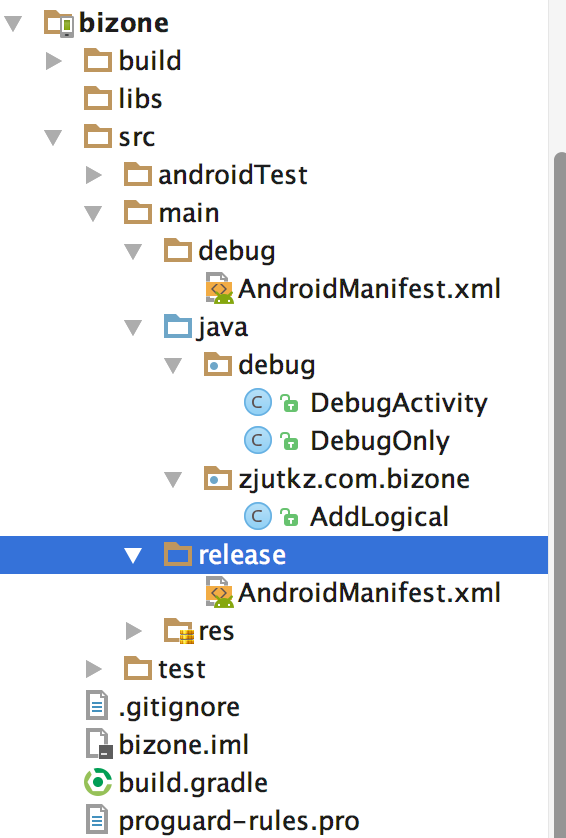
|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | dependencies {  compile fileTree(include: ['\*.jar'], dir: 'libs')  compile 'com.android.support:appcompat-v7:23.2.1'  testCompile 'junit:junit:4.12'  if(!isDebug.toBoolean()){  compile project(':bizone')  compile project(':biztwo')  compile project(':bizthree')  } } |

这是主App的gradle依赖，在Release模式下，直接compile那几个业务module就好了。

通过上面的操作，我们已经可以进行组件化开发了，很简单有没有。

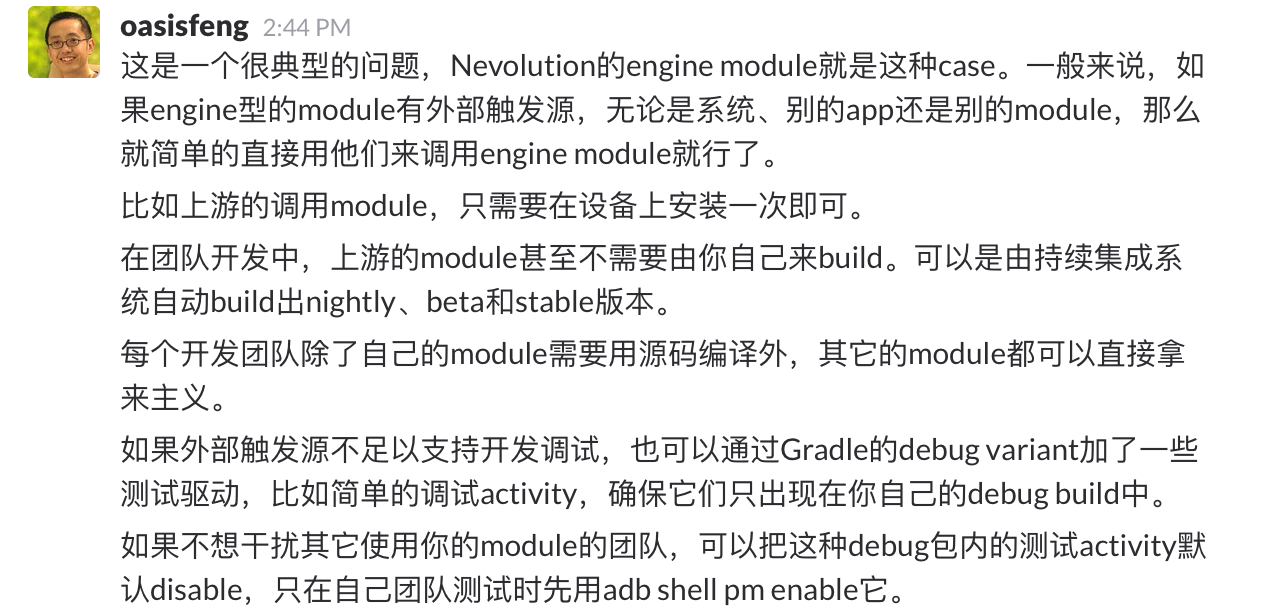
但是需要考虑的东西还有很多。比如有一些不存在UI的业务，或者说一些业务没办法独立运行，需要一个触发源。这种情况下，最理想的方式是通过其他某个已存在的module去触发它们，或者使用一个类似于DebugActivity的东西来当作触发源，而这样的DebugOnly的东西是不应该打包到Release模式中的，所以我们需要通过gradle配置去做一些自动化的东西。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | sourceSets {  main {  if (isDebug.toBoolean()) {  manifest.srcFile 'src/main/debug/AndroidManifest.xml'  } else {  manifest.srcFile 'src/main/release/AndroidManifest.xml'  java {  exclude 'debug/\*\*'  }  }  } } |



通过上面这种方式，我们就可以在Release的环境下去除掉debug包里的东西 。另外需要两个Manifest文件的原因是在Debug模式下，我们需要一个Activity标示为MAIN和LAUNCHER，而Release模式下则不需要。

关于这一点，我和冯老师也进行过沟通，下面是他的回答：



还有一个问题就是，如果你想在Debug模式下使用主App去依赖业务，是没有办法通过compile的方式去实现的，因为Debug模式下各个业务线是Application，没有办法compile，这种时候，你就需要手动去将业务module的AAR添加到App中进行依赖。业务module的AAR可以在对应的build目录下找到。这个操作可以通过gradle插件去进行自动化的完成，至于怎么写对应的插件大家自己去实现吧，比较的简单。

# **怎么样更好的组件化**

当我们完成了组件化的准备工作之后，我们需要利用一些框架或者说技术让我们的组件化之路更加完善。这里首当其冲所需要的就是URLRouter。

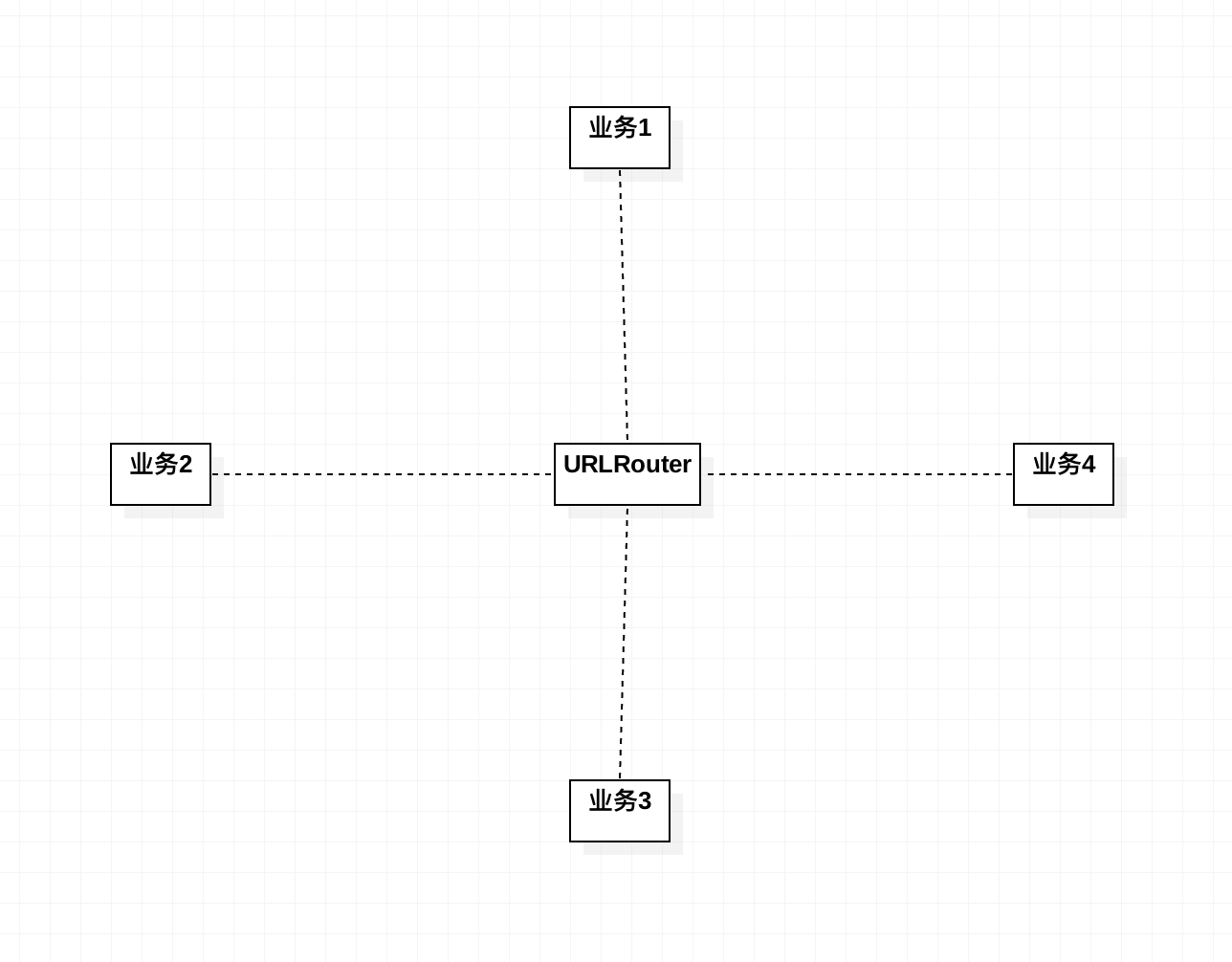
所谓URLRouter，就是通过类似打开网页的方式去通过路由打开一个Activity，而非直接通过显式Intent去方式去进行跳转。那是因为在进行了组件化开发之后，各个业务模块进行是可以独立运行的了，我们再使用显式Intent的方式去联系每个模块之间中间结构就会显得耦合性比较大，而且在Debug模式下也没办法使用显式Intent的方式去进行模块间的跳转。使用URLRouter的方式则可以很好的解耦，具体方法大家可以参考[这篇文章](http://zjutkz.net/2016/04/17/%E5%A5%BD%E5%A5%BD%E5%92%8Ch5%E6%B2%9F%E9%80%9A%EF%BC%81%E5%87%A0%E7%A7%8D%E5%B8%B8%E8%A7%81%E7%9A%84hybrid%E9%80%9A%E4%BF%A1%E6%96%B9%E5%BC%8F/" \t "http://zjutkz.net/2016/10/07/%E5%85%B3%E4%BA%8EAndroid%E4%B8%9A%E5%8A%A1%E7%BB%84%E4%BB%B6%E5%8C%96%E7%9A%84%E4%B8%80%E4%BA%9B%E6%80%9D%E8%80%83/_blank)里的第三种方式。

无非就是在配置Activity的时候设置一些data。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | <activity android:name=".SubBizActivity">  <intent-filter>  <data  android:host="bizTwo"  android:path="/someWorks"  android:scheme="appName"/>  <action android:name="android.intent.action.VIEW"/>  <category android:name="android.intent.category.DEFAULT"/>  <category android:name="android.intent.category.BROWSABLE"/>  </intent-filter> </activity> |

然后在跳转的时候使用setData的方式进行跳转。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | String url = "appName://bizTwo/someWorks"; Intent intent = new Intent(Intent.ACTION\_VIEW, Uri.parse(url)); intent.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK); PackageManager packageManager = getPackageManager(); List<ResolveInfo> activities = packageManager.queryIntentActivities(intent, 0); if (!activities.isEmpty()) {  startActivity(intent); } |



就像图中所示的一样，通过URLRouter，我们就可以将业务线之间的关系解耦，互相调用也很方便。

当然要做到完善，这样的URLRouter最好是封装成一个库，减少上层的开发并且增强鲁棒性。

还有一点，通过这种方式实现组件化需要用到gradle的一些特性，比如上面说的分Debug和Release模式，在Release模式下不要编译DebugOnly的东西等等。而这样的特性我们可以通过自动化的方式去完成，而不需要开发每次新建一个module都去手写一遍。

具体的方式我这里想到的就是使用Android Studio的自定义模板。具体关于自定义模板的教程我这里不详细讲了，毕竟不是这篇文章的重点，大家可以参考[这篇文章](http://www.jianshu.com/p/81d03519dd33" \t "http://zjutkz.net/2016/10/07/%E5%85%B3%E4%BA%8EAndroid%E4%B8%9A%E5%8A%A1%E7%BB%84%E4%BB%B6%E5%8C%96%E7%9A%84%E4%B8%80%E4%BA%9B%E6%80%9D%E8%80%83/_blank)和[这篇文章](http://mp.weixin.qq.com/s?__biz=MzA3ODg4MDk0Ng==&mid=2651112554&idx=1&sn=95e57e0d6cb95f5c209ee8065380a395&scene=23&srcid=0812wCGnGEoqXAuvJWnZg3NA" \l "rd" \t "http://zjutkz.net/2016/10/07/%E5%85%B3%E4%BA%8EAndroid%E4%B8%9A%E5%8A%A1%E7%BB%84%E4%BB%B6%E5%8C%96%E7%9A%84%E4%B8%80%E4%BA%9B%E6%80%9D%E8%80%83/_blank)。

对于组件化这个场景，我们所需要做的就是改写gradle.properties.ftl这个文件，添加一个isDebug属性。然后改写的build.gradle.ftl文件，添加我前面所写的那些sourceSets闭包里的代码。这些一两句话也说不清，有机会的话下次可以专门写一篇文章讲如何自定义Android Studio的模板。

# **关于组件化的思考**

说了这么多，相信大家对组件化已经有了一个大致的概念，也知道了我们为什么要使用组件化。而在我看来，这样的技术其实对于纯开发而言难度是不大的，真正的难度在于如何剥离现有的业务线。粒度大拆分比较容易，但是不利于今后的维护。粒度小需要对业务有很深的理解，但是能很好的解耦并且提高灵活度，所以具体的情况需要在具体的实际开发中进行分析。

我的看法是组件化的前期可以先把业务线剥离的粗一点，尽管上手并且适应这样的开发，到了后期等开发熟悉了这样的形式并且对各条业务线有了很好的理解，再慢慢的细化也不迟。