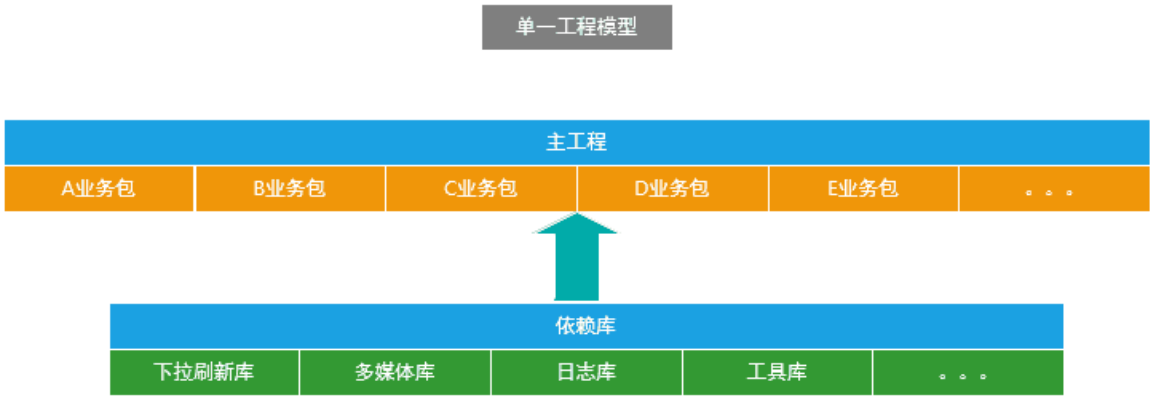
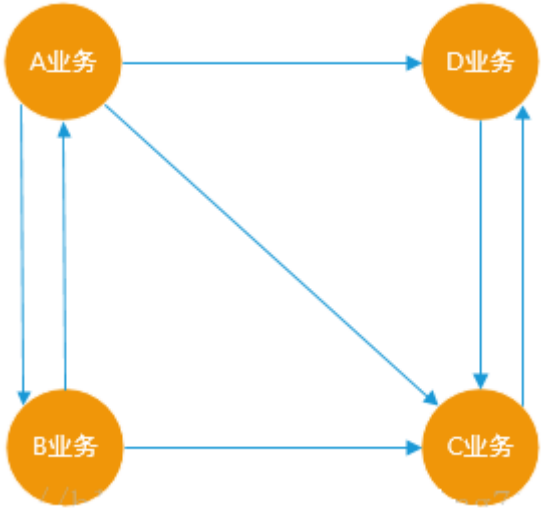
一、为什么要项目组件化

随着APP版本不断的迭代，新功能的不断增加，业务也会变的越来越复杂，APP业务模块的数量有可能还会继续增加，而且每个模块的代码也变的越来越多，这样发展下去单一工程下的APP架构势必会影响开发效率，增加项目的维护成本，每个工程师都要熟悉如此之多的代码，将很难进行多人协作开发，而且Android项目在编译代码的时候电脑会非常卡，又因为单一工程下代码耦合严重，每修改一处代码后都要重新编译打包测试，导致非常耗时，最重要的是这样的代码想要做单元测试根本无从下手，所以必须要有更灵活的架构代替过去单一的工程架构。



上图是目前比较普遍使用的Android APP技术架构，往往是在一个界面中存在大量的业务逻辑，而业务逻辑中充斥着各种网络请求、数据操作等行为，整个项目中也没有模块的概念，只有简单的以业务逻辑划分的文件夹，并且业务之间也是直接相互调用、高度耦合在一起的；

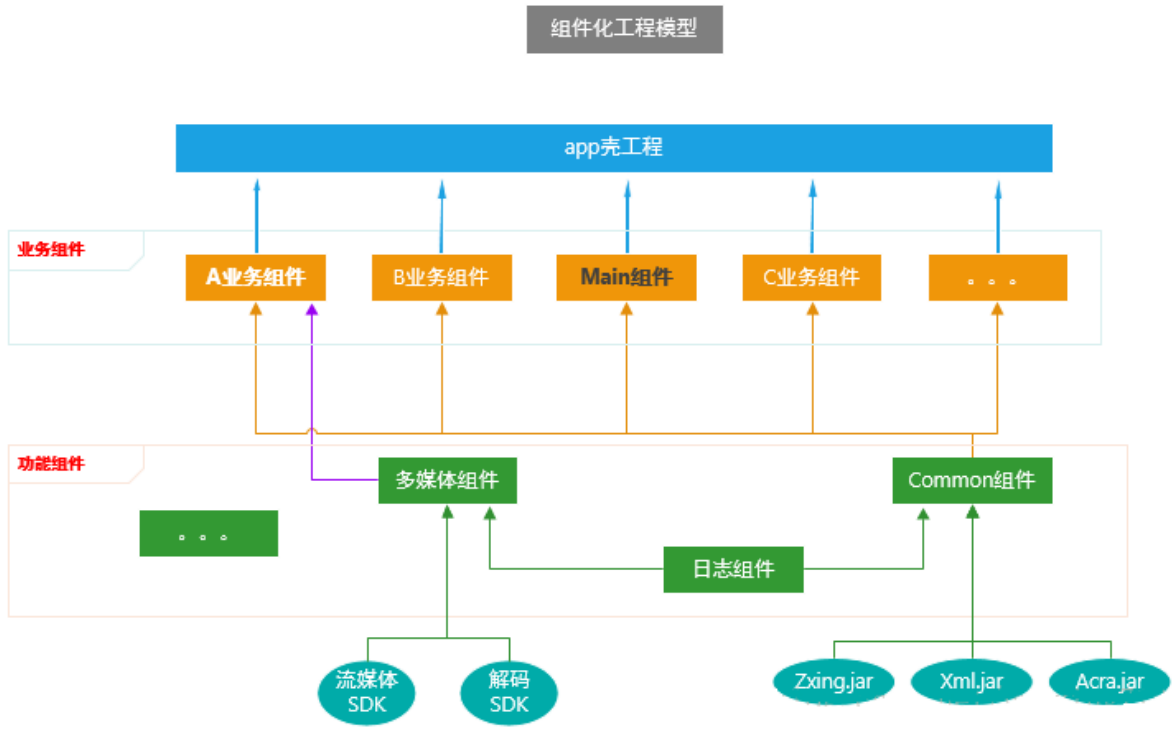


上图单一工程模型下的业务关系，总的来说就是：你中有我，我中有你，相互依赖，无法分离。
然而随着产品的迭代，业务越来越复杂，随之带来的是项目结构复杂度的极度增加，此时我们会面临如下几个问题：

1、实际业务变化非常快，但是单一工程的业务模块耦合度太高，牵一发而动全身；
2、对工程所做的任何修改都必须要编译整个工程；
3、功能测试和系统测试每次都要进行；
4、团队协同开发存在较多的冲突.不得不花费更多的时间去沟通和协调，并且在开发过程中，任何一位成员没办法专注于自己的功能点，影响开发效率；
5、不能灵活的对业务模块进行配置和组装；

为了满足各个业务模块的迭代而彼此不受影响，更好的解决上面这种让人头疼的依赖关系，就需要整改App的架构。

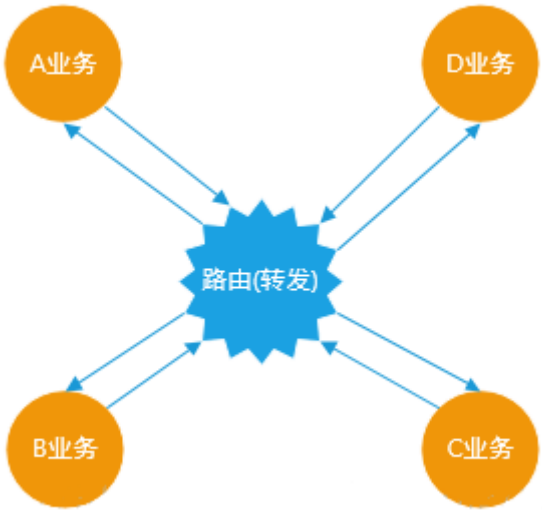
二、如何组件化



上图是组件化工程模型，为了方便理解这张架构图，下面会列举一些组件化工程中用到的名词的含义：

名词	含义
集成模式	所有的业务组件被“app壳工程”依赖，组成一个完整的APP；
组件模式	可以独立开发业务组件，每一个业务组件就是一个APP；
app壳工程	负责管理各个业务组件，和打包apk，没有具体的业务功能；
业务组件	根据公司具体业务而独立形成一个的工程；
功能组件	提供开发APP的某些基础功能，例如打印日志、树状图等；
Main组件	属于业务组件，指定APP启动页面、主界面；
Common组件	属于功能组件，支撑业务组件的基础，提供多数业务组件需要的功能，例如提供网络请求功能；

Android APP组件化架构的目标是告别结构臃肿，让各个业务变得相对独立，业务组件在组件模式下可以独立开发，而在集成模式下又可以变为arr包集成到“app壳工程”中，组成一个完整功能的APP； 从组件化工程模型中可以看到，**业务组件之间是独立的，没有关联的**，这些业务组件在集成模式下是一个个library，被app壳工程所依赖，组成一个具有完整业务功能的APP应用，但是在组件开发模式下，业务组件又变成了一个个application，它们可以独立开发和调试，由于在组件开发模式下，业务组件们的代码量相比于完整的项目差了很远，因此在运行时可以显著减少编译时间。



这是组件化工程模型下的业务关系，业务之间将不再直接引用和依赖，而是通过“路由”这样一个中转站间接产生联系，而Android中的路由实际就是对URL Scheme的封装； 如此规模大的架构整改需要付出更高的成本，还会涉及一些潜在的风险，但是整改后的架构能够带来很多好处：

1、加快业务迭代速度，各个业务模块组件更加独立，不再出现业务耦合情况；
2、稳定的公共模块采用依赖库方式，提供给各个业务线使用，减少重复开发和维护工作量；
3、迭代频繁的业务模块采用组件方式，各业务研发可以互不干扰、提升协作效率，并控制产品质量；
4、为新业务随时集成提供了基础，所有业务可上可下，灵活多变；
5、降低团队成员熟悉项目的成本，降低项目的维护难度；
6、加快编译速度，提高开发效率；
7、控制代码权限，将代码的权限细分到更小的粒度；

三、组件化实施流程

1、组件模式和集成模式的转换

Android Studio中的Module主要有两种属性，分别为：

1、**application属性**，可以独立运行的Android程序，也就是我们的APP；

```
apply plugin: ‘com.android.application’1
```

2、**library属性**，不可以独立运行，一般是Android程序依赖的库文件；

```
apply plugin: ‘com.android.library’1
```

Module的属性是在每个组件的 **build.gradle** 文件中配置的，当我们在组件模式开发时，业务组件应处于application属性，这时的业务组件就是一个 Android App，可以独立开发和调试；而当我们转换到集成模式开发时，业务组件应该处于 library 属性，这样才能被我们的“app壳工程”所依赖，组成一个具有完整功能的APP；

但是我们如何让组件在这两种模式之间自动转换呢？总不能每次需要转换模式的时候去每个业务组件的 Gralde 文件中去手动把 Application 改成 library 吧？如果我们的项目只有两三个组件那么这个办法肯定是可行的，手动去改一遍也用不了多久，但是在大型项目中我们可能会有十几个业务组件，再去手动改一遍必定费时费力，这时候就需要程序员发挥下懒的本质了。

试想，我们经常在写代码的时候定义静态常量，那么定义静态常量的目的什么呢？当一个常量需要被好几处代码引用的时候，把这个常量定义为静态常量的好处是当这个常量的值需要改变时我们只需要改变静态常量的值，其他引用了这个静态常量的地方都会被改变，**做到了一次改变，到处生效**；根据这个思想，那么我们就可以在我们的代码中的某处定义一个决定业务组件属性的常量，然后让所有业务组件的build.gradle都引用这个常量，这样当我们改变了常量值的时候，所有引用了这个常量值的业务组件就会根据值的变化改变自己的属性；可是问题来了？静态常量是用Java代码定义的，而改变组件属性是需要在Gradle中定义的，Gradle能做到吗？

Gradle自动构建工具有一个重要属性，可以帮助我们完成这个事情。每当我们用AndroidStudio创建一个Android项目后，就会在项目的根目录中生成一个文件 **gradle.properties**，我们将使用这个文件的一个重要属性：**在Android项目中的任何一个build.gradle文件中都可以把gradle.properties中的常量读取出来**；那么我们在上面提到解决办法就有了实际行动的方法，首先我们在gradle.properties中定义一个常量值 **isModule（是否是组件开发模式，true为是，false为否）**：

```
# 每次更改“isModule”的值后，需要点击 "Sync Project" 按钮
isModule=false12
```

然后我们在业务组件的build.gradle中读取 **isModule**，但是 gradle.properties 还有一个重要属性：**gradle.properties 中的数据类型都是String类型，使用其他数据类型需要自行转换；**也就是说我们读到 isModule 是个String类型的值，而我们需要的是Boolean值，代码如下：

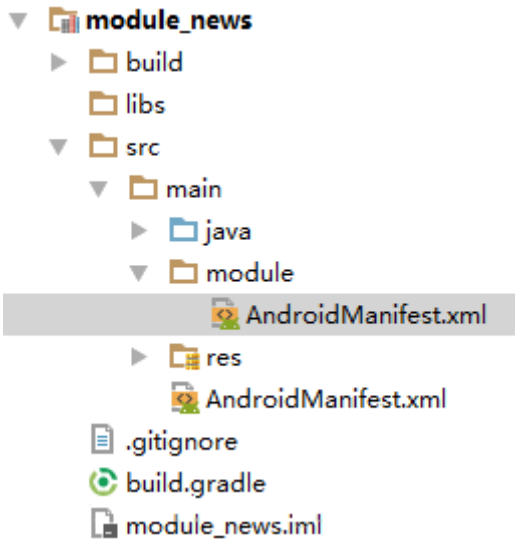
```
if (isModule.toBoolean()) {
    apply plugin: 'com.android.application'
} else {
    apply plugin: 'com.android.library'
}12345
```

这样我们第一个问题就解决了，当然了 **每次改变isModule的值后，都要同步项目才能生效；**

2、组件之间AndroidManifest合并问题

在 AndroidStudio 中每一个组件都会有对应的 AndroidManifest.xml，用于声明需要的权限、Application、Activity、Service、Broadcast等，当项目处于组件模式时，业务组件的 AndroidManifest.xml 应该具有一个 Android APP 所具有的的所有属性，尤其是声明 Application 和要launch的Activity，但是当项目处于集成模式的时候，每一个业务组件的 AndroidManifest.xml 都要合并到“app壳工程”中，要是每一个业务组件都有自己的 Application 和 launch的Activity，那么合并的时候肯定会冲突，试想一个APP怎么可能会有多个 Application 和 launch 的Activity呢？

但是大家应该注意到这个问题是在组件开发模式和集成开发模式之间转换引起的问题，而在上一节中我们已经解决了组件模式和集成模式转换的问题，另外大家应该都经历过将 Android 项目从 Eclipse 切换到 AndroidStudio 的过程，由于 Android 项目在 Eclipse 和 AndroidStudio开发时 AndroidManifest.xml 文件的位置是不一样的，我们需要在**build.gradle 中指定下 AndroidManifest.xml 的位置**，AndroidStudio 才能读取到 AndroidManifest.xml，这样解决办法也就有了，我们可以**为组件开发模式下的业务组件再创建一个 AndroidManifest.xml，然后根据isModule指定AndroidManifest.xml的文件路径，让业务组件在集成模式和组件模式下使用不同的AndroidManifest.xml**，这样表单冲突的问题就可以规避了。



上图是组件化项目中一个标准的业务组件目录结构，首先我们在main文件夹下创建一个module文件夹用于存放组件开发模式下业务组件的 AndroidManifest.xml，而 AndroidStudio 生成的 AndroidManifest.xml 则依然保留，并用于集成开发模式下业务组件的表单；然后我们需要在业务组件的 build.gradle 中指定表单的路径，代码如下：

```
  sourceSets {
        main {
            if (isModule.toBoolean()) {
                manifest.srcFile 'src/main/module/AndroidManifest.xml'
            } else {
                manifest.srcFile 'src/main/AndroidManifest.xml'
            }
        }
    }123456789
```

这样在不同的开发模式下就会读取到不同的 AndroidManifest.xml ，然后我们需要修改这两个表单的内容以为我们不同的开发模式服务。

首先是集成开发模式下的 AndroidManifest.xml，前面我们说过集成模式下，业务组件的表单是绝对不能拥有自己的 Application 和 launch 的 Activity的，也不能声明APP名称、图标等属性，总之app壳工程有的属性，业务组件都不能有，下面是一份**标准的集成开发模式**下业务组件的 AndroidManifest.xml：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.guiying.girls">

    <application android:theme="@style/AppTheme">
        <activity
            android:name=".main.GirlsActivity"
            android:screenOrientation="portrait" />
        <activity
            android:name=".girl.GirlActivity"
            android:screenOrientation="portrait"
            android:theme="@style/AppTheme.NoActionBar" />
    </application>

</manifest>
123456789101112131415
```

我在这个表单中只声明了应用的主题，而且这个主题还是跟app壳工程中的主题是一致的，都引用了common组件中的资源文件，在这里声明主题是为了方便这个业务组件中有使用默认主题的Activity时就不用再给Activity单独声明theme了。

然后是组件开发模式下的表单文件：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.guiying.girls">

    <application
        android:name="debug.GirlsApplication"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/girls_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".main.GirlsActivity"
            android:screenOrientation="portrait">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".girl.GirlActivity"
            android:screenOrientation="portrait"
            android:theme="@style/AppTheme.NoActionBar" />
    </application>

</manifest>12345678910111213141516171819020212223242526
```

组件模式下的业务组件表单就是一个Android项目普通的AndroidManifest.xml，这里就不在过多介绍了。

3、全局Context的获取及组件数据初始化

当Android程序启动时，Android系统会为每个程序创建一个 Application 类的对象，并且只创建一个，application对象的生命周期是整个程序中最长的，它的生命周期就等于这个程序的生命周期。在默认情况下应用系统会自动生成 Application 对象，但是如果我们自定义了 Application，那就需要在 AndroidManifest.xml 中声明告知系统，实例化的时候，是实例化我们自定义的，而非默认的。

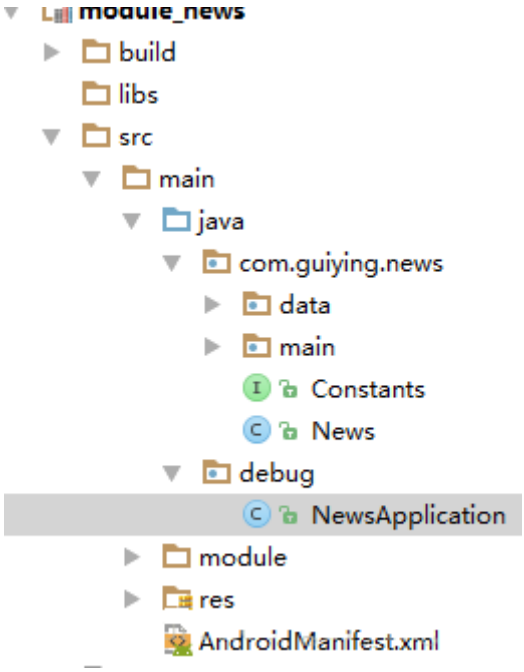
但是我们在组件化开发的时候，可能为了数据的问题每一个组件都会自定义一个Application类，如果我们在自己的组件中开发时需要获取 **全局的Context**，一般都会直接获取 application 对象，但是当所有组件要打包合并在一起的时候就会出现问题，因为最后程序只有一个 Application，我们组件中自己定义的 Application 肯定是没法使用的，因此我们需要想办法再任何一个业务组件中都能获取到全局的 Context，而且这个 Context 不管是在组件开发模式还是在集成开发模式都是生效的。

在 组件化工程模型图中，功能组件集合中有一个 **Common 组件**， Common 有公共、公用、共同的意思，所以这个组件中主要封装了项目中需要的基础功能，并且每一个业务组件都要依赖Common组件，Common 组件就像是万丈高楼的地基，而业务组件就是在 Common 组件这个地基上搭建起来我们的 APP的，Common 组件会专门在一个章节中讲解，这里只讲 Common组件中的一个功能，在Common 组件中我们封装了项目中用到的各种Base类，这些基类中就有**BaseApplication 类。**

BaseApplication 主要用于各个业务组件和app壳工程中声明的 Application 类继承用的，只要各个业务组件和app壳工程中声明的Application类继承了 BaseApplication，当应用启动时 BaseApplication 就会被动实例化，这样从 BaseApplication 获取的 Context 就会生效，也就从根本上解决了我们不能直接从各个组件获取全局 Context 的问题；

这时候大家肯定都会有个疑问？不是说了业务组件不能有自己的 Application 吗，怎么还让他们继承 BaseApplication 呢？其实我前面说的是业务组件不能在集成模式下拥有自己的 Application，但是这不代表业务组件也不能在组件开发模式下拥有自己的Application，其实业务组件在组件开发模式下必须要有自己的 Application 类，一方面是为了让 BaseApplication 被实例化从而获取 Context，还有一个作用是，**业务组件自己的 Application 可以在组件开发模式下初始化一些数据**，例如在组件开发模式下，A组件没有登录页面也没法登录，因此就无法获取到 Token，这样请求网络就无法成功，因此我们需要在A组件这个 APP 启动后就应该已经登录了，这时候组件自己的 Application 类就有了用武之地，我们在组件的 Application的 onCreate 方法中模拟一个登陆接口，在登陆成功后将数据保存到本地，这样就可以处理A组件中的数据业务了；另外我们也**可以在组件Application中初始化一些第三方库。**

但是，实际上业务组件中的Application在最终的集成项目中是没有什么实际作用的，组件自己的 Application 仅限于在组件模式下发挥功能，因此我们需要在将项目从组件模式转换到集成模式后将组件自己的Application剔除出我们的项目；在 AndroidManifest 合并问题小节中介绍了如何在不同开发模式下让 Gradle 识别组件表单的路径，这个方法也同样适用于Java代码；



我们在Java文件夹下创建一个 debug 文件夹，用于存放不会在业务组件中引用的类，例如上图中的 NewsApplication，你甚至**可以在 debug 文件夹中创建一个Activity，然后组件表单中声明启动这个 Activity，在这个Activity中不用setContentView，只需要在启动你的目标Activity的时候传递参数就行，这样就就可以解决组件模式下某些Activity需要getIntent数据而没有办法拿到的情况**，代码如下；

```
public class LauncherActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        request();
        Intent intent = new Intent(this, TargetActivity.class);
        intent.putExtra("name", "avcd");
        intent.putExtra("syscode", "023e2e12ed");
        startActivity(intent);
        finish();
    }
```

```
    //申请读写权限
    private void request() {
        AndPermission.with(this)
                .requestCode(110)
                .permission(Manifest.permission.WRITE_EXTERNAL_STORAGE,
                        Manifest.permission.CAMERA,
Manifest.permission.READ_PHONE_STATE)
                .callback(this)
                .start();
    }

}12345678910111213141516171819202122232 4
```

接下来在业务组件的 build.gradle 中，根据 isModule 是否是集成模式将 debug 这个 Java代码文件夹排除：

```
    sourceSets {
        main {
            if (isModule.toBoolean()) {
                manifest.srcFile 'src/main/module/AndroidManifest.xml'
            } else {
                manifest.srcFile 'src/main/AndroidManifest.xml'
                //集成开发模式下排除debug文件夹中的所有Java文件
                java {
                    exclude 'debug/**'
                }
            }
        }
    }12345678910111213 14
```

4、library依赖问题

在介绍这一节的时候，先说一个问题，在**组件化工程模型图**中，多媒体组件和Common组件都依赖了日志组件，而A业务组件有同时依赖了多媒体组件和Common组件，这时候就会有人问，你这样搞岂不是日志组件要被重复依赖了，而且Common组件也被每一个业务组件依赖了，这样不出问题吗？

其实大家完全没有必要担心这个问题，如果真有重复依赖的问题，在你编译打包的时候就会报错，如果你还是不相信的话可以反编译下最后打包出来的APP，看看里面的代码你就知道了。组件只是我们在代码开发阶段中为了方便叫的一个术语，在组件被打包进APP的时候是没有这个概念的，这些组件最后都会被打包成arr包，然后被app壳工程所依赖，在构建APP的过程中Gradle会自动将重复的arr包排除，APP中也就不会存在相同的代码了；

但是虽然组件是不会重复了，但是我们还是要考虑另一个情况，我们在build.gradle中compile的第三方库，例如AndroidSupport库经常会被一些开源的控件所依赖，而我们自己一定也会compile AndroidSupport库 ，这就会造成第三方包和我们自己的包存在重复加载，解决办法就是找出那个多出来的库，并将多出来的库给排除掉，而且Gradle也是支持这样做的，分别有两种方式：**根据组件名排除或者根据包名排除**，下面以排除support-v4库为例：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile("com.jude:easyrecyclerview:$rootProject.easyRecyclerVersion") {
        exclude module: 'support-v4'//根据组件名排除
        exclude group: 'android.support.v4'//根据包名排除
    }
}1234567
```

library重复依赖的问题算是都解决了，但是我们在开发项目的时候会依赖很多开源库，而这些库每个组件都需要用到，要是每个组件都去依赖一遍也是很麻烦的，尤其是给这些库升级的时候，为了方便我们统一管理第三方库，我们将给给整个工程提供统一的依赖第三方库的入口，前面介绍的Common库的作用之一就是统一依赖开源库，因为其他业务组件都依赖了Common库，所以这些业务组件也就间接依赖了Common所依赖的开源库。

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    //Android Support
    compile "com.android.support:appcompat-
v7:$rootProject.supportLibraryVersion"
    compile "com.android.support:design:$rootProject.supportLibraryVersion"
    compile "com.android.support:percent:$rootProject.supportLibraryVersion"
    //网络请求相关
    compile "com.squareup.retrofit2:retrofit:$rootProject.retrofitVersion"
    compile "com.squareup.retrofit2:retrofit-mock:$rootProject.retrofitVersion"
    compile
"com.github.franmontiel:PersistentCookieJar:$rootProject.cookieVersion"
    //稳定的
    compile "com.github.bumptech.glide:glide:$rootProject.glideVersion"
    compile "com.orhanobut:logger:$rootProject.loggerVersion"
    compile "org.greenrobot:eventbus:$rootProject.eventbusVersion"
    compile "com.google.code.gson:gson:$rootProject.gsonVersion"
    compile "com.github.chrisbanes:PhotoView:$rootProject.photoViewVersion"

    compile "com.jude:easyrecyclerview:$rootProject.easyRecyclerVersion"
    compile "com.github.GrenderG:Toasty:$rootProject.toastyVersion"

    //router
    compile
"com.github.mzule.activityrouter:activityrouter:$rootProject.routerVersion"
}12345678910111213141516171819202122223
```

5、组件之间调用和通信

在组件化开发的时候，组件之间是没有依赖关系，我们不能在使用显示调用来跳转页面了，因为我们组件化的目的之一就是解决模块间的强依赖问题，假如现在要从A业务组件跳转到业务B组件，并且要携带参数跳转，这时候怎么办呢？而且组件这么多怎么管理也是个问题，这时候就需要引入“路由”的概念了，由本文开始的组件化模型下的业务关系图可知路由就是起到一个转发的作用。

这里我将介绍开源库的**“ActivityRouter”**，有兴趣的同学情直接去ActivityRouter的Github主页学习：ActivityRouter，ActivityRouter支持给Activity定义 URL，这样就可以通过 URL 跳转到Activity，并且支持从浏览器以及 APP 中跳入我们的Activity，而且还支持通过 url 调用方法。下面将介绍如何将ActivityRouter集成到组件化项目中以实现组件之间的调用；

1、首先我们需要在 Common 组件中的 build.gradle 将ActivityRouter 依赖进来，方便我们在业务组件中调用：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    //router
    compile
"com.github.mzule.activityrouter:activityrouter:$rootProject.routerVersion"
}12345
```

2、这一步我们需要先了解 **APT**这个概念，**APT(Annotation Processing Tool)是一种处理注解的工具，它对源代码文件进行检测找出其中的Annotation，使用Annotation进行额外的处理。Annotation处理器在处理Annotation时可以根据源文件中的Annotation生成额外的源文件和其它的文件(文件具体内容由Annotation处理器的编写者决定)，APT还会编译生成的源文件和原来的源文件，将它们一起生成class文件。**在这里我们将在每一个**业务组件**的 build.gradle 都引入ActivityRouter 的Annotation处理器，我们将会在声明组件和Url的时候使用，annotationProcessor是Android官方提供的Annotation处理器插件，代码如下：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    annotationProcessor
"com.github.mzule.activityrouter:compiler:$rootProject.annotationProcessor"
}12345
```

3、接下来需要在 **app壳工程**的 AndroidManifest.xml 配置，到这里ActivityRouter配置就算完成了：

```
 <!--声明整个应用程序的路由协议-->
        <activity
            android:name="com.github.mzule.activityrouter.router.RouterActivity"
            android:theme="@android:style/Theme.NoDisplay">
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />

                <category android:name="android.intent.category.DEFAULT" />
                <category android:name="android.intent.category.BROWSABLE" />

                <data android:scheme="@string/global_scheme" /> <!-- 改成自己的
scheme -->
            </intent-filter>
        </activity>
        <!--发送崩溃日志界面-->1234567891011121314
```

4、接下来我们将声明项目中的业务组件，声明方法如下：

```
@Module("girls")
public class Girls {
}123
```

在每一个业务组件的java文件的根目录下创建一个类，用 **注解@Module** 声明这个业务组件；
然后在“**app壳工程**”的 **应用Application** 中使用 **注解@Modules** 管理我们声明的所有业务组件，方法如下：

```
@Modules({"main", "girls", "news"})
public class MyApplication extends BaseApplication {
}123
```

到这里组件化项目中的所有业务组件就被声明和管理起来了，组件之间的也就可以互相调用了，当然前提是要给业务组件中的Activity定义 URL。

5、例如我们给 Girls组件 中的 GirlsActivity 使用 **注解@Router** 定义一个 URL：“news”，方法如下：

```
@Router("girls")
public class GirlsActivity extends BaseActionBarActivity {

    private GirlsView mView;
    private GirlsContract.Presenter mPresenter;

    @Override
    protected int setTitleId() {
        return R.string.girls_activity_title;
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mView = new GirlsView(this);
        setContentView(mView);
        mPresenter = new GirlsPresenter(mView);
        mPresenter.start();
    }
}12345678910111213141516171819201
```

然后我们就可以**在项目中的任何一个地方通过 URL地址： module://girls, 调用 GirlsActivity**，方法如下：

```
        Routers.open(MainActivity.this, "module://girls");1
```

组件之间的调用解决后，另外需要解决的就是组件之间的通信，例如A业务组件中有消息列表，而用户在B组件中操作某个事件后会产生一条新消息，需要通知A组件刷新消息列表，这样业务场景需求可以使用Android广播来解决，也可以使用第三方的事件总线来实现，比如EventBus。

6、组件之间资源名冲突

因为我们拆分出了很多业务组件和功能组件，在把这些组件合并到**“app壳工程”**时候就有可能会出现资源名冲突问题，例如**A组件和B组件都有一张叫做“ic_back”的图标**，这时候在集成模式下打包APP就会编译出错，解决这个问题最简单的办法就是**在项目中约定资源文件命名规约，比如强制使每个资源文件的名称以组件名开始，这个可以根据实际情况和开发人员制定规则。**当然了万能的Gradle构建工具也提供了解决方法，通过在在组件的build.gradle中添加如下的代码：

```
    //设置了resourcePrefix值后，所有的资源名必须以指定的字符串做前缀，否则会报错。
    //但是resourcePrefix这个值只能限定xml里面的资源，并不能限定图片资源，所有图片资源仍然需要手动去修改资源名。
    resourcePrefix "girls_"123
```

但是设置了这个属性后有个问题，所有的资源名必须以指定的字符串做前缀，否则会报错，而且resourcePrefix这个值只能限定xml里面的资源，并不能限定图片资源，所有图片资源仍然需要手动去修改资源名；所以我并不推荐使用这种方法来解决资源名冲突。

四、组件化项目的工程类型

在组件化工程模型中主要有：**app壳工程、业务组件和功能组件3种类型，而业务组件中的Main组件和功能组件中的Common组件比较特殊**，下面将分别介绍。

1、app壳工程

app壳工程是从名称来解释就是一个空壳工程，没有任何的业务代码，也不能有Activity，但它又必须被单独划分成一个组件，而不能融合到其他组件中，是因为它有如下几点重要功能：

1、**app壳工程中声明了我们Android应用的 Application**，这个 Application 必须继承自 Common组件中的 BaseApplication（如果你无需实现自己的Application可以直接在表单声明BaseApplication），因为只有这样，在打包应用后才能让BaseApplication中的Context生效，当然你还可以在这个 Application中初始化我们工程中使用到的库文件，还可以在这里解决Android引用方法数不能超过 65535 的限制，对崩溃事件的捕获和发送也可以在这里声明。

2、**app壳工程的 AndroidManifest.xml 是我Android应用的根表单**，应用的名称、图标以及是否支持备份等等属性都是在这份表单中配置的，其他组件中的表单最终在集成开发模式下都被合并到这份AndroidManifest.xml 中。

3、**app壳工程的 build.gradle 是比较特殊的**，app壳不管是在集成开发模式还是组件开发模式，它的属性始终都是：com.android.application，因为最终其他的组件都要被app壳工程所依赖，被打包进app壳工程中，这一点从组件化工程模型图中就能体现出来，所以app壳工程是不需要单独调试单独开发的。另外Android应用的打包签名，以及buildTypes和defaultConfig都需要在这里配置，而它的dependencies则需要根据isModule的值分别依赖不同的组件，在组件开发模式下app壳工程只需要依赖Common组件，或者为了防止报错也可以根据实际情况依赖其他功能组件，而在集成模式下app壳工程必须依赖所有在应用Application中声明的业务组件，并且不需要再依赖任何功能组件。

下面是一份 app壳工程 的 build.gradle文件：

```
apply plugin: 'com.android.application'

static def buildTime() {
    return new Date().format("yyyyMMdd");
}

android {
    signingConfigs {
        release {
            keyAlias 'guiying712'
            keyPassword 'guiying712'
            storeFile file('/mykey.jks')
            storePassword 'guiying712'
        }
    }

    compileSdkVersion rootProject.ext.compileSdkVersion
    buildToolsVersion rootProject.ext.buildToolsVersion
    defaultConfig {
        applicationId "com.guiying.androidmodulepattern"
        minSdkVersion rootProject.ext.minSdkVersion
        targetSdkVersion rootProject.ext.targetSdkVersion
        versionCode rootProject.ext.versionCode
        versionName rootProject.ext.versionName
        multiDexEnabled false
```

```
        //打包时间
        resValue "string", "build_time", buildTime()
    }

    buildTypes {
        release {
            //更改AndroidManifest.xml中预先定义好占位符信息
            //manifestPlaceholders = [app_icon: "@drawable/icon"]
            // 不显示Log
            buildConfigField "boolean", "LEO_DEBUG", "false"
            //是否zip对齐
            zipAlignEnabled true
            // 缩减resource文件
            shrinkResources true
            //Proguard
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
            //签名
            signingConfig signingConfigs.release
        }

        debug {
            //给applicationId添加后缀“.debug”
            applicationIdSuffix ".debug"
            //manifestPlaceholders = [app_icon: "@drawable/launch_beta"]
            buildConfigField "boolean", "LOG_DEBUG", "true"
            zipAlignEnabled false
            shrinkResources false
            minifyEnabled false
            debuggable true
        }
    }


}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    annotationProcessor
"com.github.mzule.activityrouter:compiler:$rootProject.annotationProcessor"
    if (isModule.toBoolean()) {
        compile project(':lib_common')
    } else {
        compile project(':module_main')
        compile project(':module_girls')
        compile project(':module_news')
    }
}12345678910111213141516171819202122232425262728293031323334353637383940414243444546474849505152535455565758596061626364656667686970717
2
```

2、功能组件和Common组件

功能组件是为了支撑业务组件的某些功能而独立划分出来的组件，功能实质上跟项目中引入的第三方库是一样的，功能组件的特征如下：

1、功能组件的 AndroidManifest.xml 是一张空表，这张表中只有功能组件的包名；

2、功能组件不管是在集成开发模式下还是组件开发模式下属性始终是： com.android.library，所以功能组件是不需要读取 gradle.properties 中的 isModule 值的；另外功能组件的 build.gradle 也无需设置 buildTypes ，只需要 dependencies 这个功能组件需要的jar包和开源库。

下面是一份 普通 的功能组件的 build.gradle文件：

```
apply plugin: 'com.android.library'

android {
    compileSdkVersion rootProject.ext.compileSdkVersion
    buildToolsVersion rootProject.ext.buildToolsVersion

    defaultConfig {
        minSdkVersion rootProject.ext.minSdkVersion
        targetSdkVersion rootProject.ext.targetSdkVersion
        versionCode rootProject.ext.versionCode
        versionName rootProject.ext.versionName
    }

}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
}12345678910111213141516171
```

Common组件除了有功能组件的普遍属性外，还具有其他功能：

1、Common组件的 AndroidManifest.xml 不是一张空表，这张表中声明了我们 Android应用用到的所有使用权限 uses-permission 和 uses-feature，放到这里是因为在组件开发模式下，所有业务组件就无需在自己的 AndroidManifest.xm 声明自己要用到的权限了。

2、Common组件的 build.gradle 需要统一依赖业务组件中用到的 第三方依赖库和jar包，例如我们用到的ActivityRouter、Okhttp等等。

3、Common组件中封装了Android应用的 Base类和网络请求工具、图片加载工具等等，公用的 widget控件也应该放在Common 组件中；业务组件中都用到的数据也应放于Common组件中，例如保存到 SharedPreferences 和 DataBase 中的登陆数据；

4、Common组件的资源文件中需要放置项目公用的 Drawable、layout、sting、dimen、color和style等等，另外项目中的 Activity 主题必须定义在 Common中，方便和 BaseActivity 配合保持整个Android应用的界面风格统一。

下面是一份 Common功能组件的 build.gradle文件：

```
apply plugin: 'com.android.library'

android {
    compileSdkVersion rootProject.ext.compileSdkVersion
    buildToolsVersion rootProject.ext.buildToolsVersion
```

```
    defaultConfig {
        minSdkVersion rootProject.ext.minSdkVersion
        targetSdkVersion rootProject.ext.targetSdkVersion
        versionCode rootProject.ext.versionCode
        versionName rootProject.ext.versionName
    }

}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    //Android Support
    compile "com.android.support:appcompat-
v7:$rootProject.supportLibraryVersion"
    compile "com.android.support:design:$rootProject.supportLibraryVersion"
    compile "com.android.support:percent:$rootProject.supportLibraryVersion"
    //网络请求相关
    compile "com.squareup.retrofit2:retrofit:$rootProject.retrofitVersion"
    compile "com.squareup.retrofit2:retrofit-mock:$rootProject.retrofitVersion"
    compile
"com.github.franmontiel:PersistentCookieJar:$rootProject.cookieVersion"
    //稳定的
    compile "com.github.bumptech.glide:glide:$rootProject.glideVersion"
    compile "com.orhanobut:logger:$rootProject.loggerVersion"
    compile "org.greenrobot:eventbus:$rootProject.eventbusVersion"
    compile "com.google.code.gson:gson:$rootProject.gsonVersion"
    compile "com.github.chrisbanes:PhotoView:$rootProject.photoViewVersion"

    compile "com.jude:easyrecyclerview:$rootProject.easyRecyclerVersion"
    compile "com.github.GrenderG:Toasty:$rootProject.toastyVersion"

    //router
    compile
"com.github.mzule.activityrouter:activityrouter:$rootProject.routerVersion"
}
12345678910111213141516171819202122232425262728293031323334353637383 9
```

3、业务组件和Main组件

业务组件就是根据业务逻辑的不同拆分出来的组件，业务组件的特征如下：

1、业务组件中要有两张AndroidManifest.xml，分别对应组件开发模式和集成开发模式，这两张表的区别请查看 **组件之间AndroidManifest合并问题** 小节。

2、业务组件在集成模式下是不能有自己的Application的，但在组件开发模式下又必须实现自己的Application并且要继承自Common组件的BaseApplication，并且这个Application不能被业务组件中的代码引用，因为它的功能就是为了使业务组件从BaseApplication中获取的全局Context生效，还有初始化数据之用。

3、业务组件有debug文件夹，这个文件夹在集成模式下会从业务组件的代码中排除掉，所以debug文件夹中的类不能被业务组件强引用，例如组件模式下的 Application 就是置于这个文件夹中，还有组件模式下开发给目标 Activity 传递参数的用的 launch Activity 也应该置于 debug 文件夹中；

4、业务组件必须在自己的 Java文件夹中创建业务组件声明类，以使 **app壳工程 中的 应用Application能够引用**，实现组件跳转，具体请查看 **组件之间调用和通信** 小节；

5、**业务组件必须在自己的 build.gradle 中根据 isModule 值的不同改变自己的属性**，在组件模式下是：com.android.application，而在集成模式下com.android.library；同时还需要在build.gradle配置资源文件，如 指定不同开发模式下的AndroidManifest.xml文件路径，排除debug文件夹等；业务组件还必须在dependencies中依赖Common组件，并且引入ActivityRouter的注解处理器annotationProcessor，以及依赖其他用到的功能组件。

下面是一份普通业务组件的 build.gradle文件：

```
if (isModule.toBoolean()) {
    apply plugin: 'com.android.application'
} else {
    apply plugin: 'com.android.library'
}

android {
    compileSdkVersion rootProject.ext.compileSdkVersion
    buildToolsVersion rootProject.ext.buildToolsVersion

    defaultConfig {
        minSdkVersion rootProject.ext.minSdkVersion
        targetSdkVersion rootProject.ext.targetSdkVersion
        versionCode rootProject.ext.versionCode
        versionName rootProject.ext.versionName
    }

    sourceSets {
        main {
            if (isModule.toBoolean()) {
                manifest.srcFile 'src/main/module/AndroidManifest.xml'
            } else {
                manifest.srcFile 'src/main/AndroidManifest.xml'
                //集成开发模式下排除debug文件夹中的所有Java文件
                java {
                    exclude 'debug/**'
                }
            }
        }
    }

    //设置了resourcePrefix值后，所有的资源名必须以指定的字符串做前缀，否则会报错。
    //但是resourcePrefix这个值只能限定xml里面的资源，并不能限定图片资源，所有图片资源仍然需要手动去修改资源名。
    //resourcePrefix "girls_"


}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    annotationProcessor
"com.github.mzule.activityrouter:compiler:$rootProject.annotationProcessor"
    compile project(':lib_common')
}12345678910111213141516171819202122232425262728293031323334353637383940414243
```

Main组件除了有业务组件的普遍属性外，还有一项重要功能：

1、Main组件集成模式下的AndroidManifest.xml是跟其他业务组件不一样的，Main组件的表单中声明了我们整个Android应用的launch Activity，这就是Main组件的独特之处；所以我建议SplashActivity、登陆Activity以及主界面都应属于Main组件，也就是说Android应用启动后要调用的页面应置于Main组件。

```
        <activity
            android:name=".splash.SplashActivity"
            android:launchMode="singleTop"
            android:screenOrientation="portrait"
            android:theme="@style/SplashTheme">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>1234567891011
```

五、组件化项目的混淆方案

组件化项目的Java代码混淆方案采用在集成模式下集中在app壳工程中混淆，各个业务组件不配置混淆文件。集成开发模式下在app壳工程中build.gradle文件的release构建类型中开启混淆属性，其他buildTypes配置方案跟普通项目保持一致，Java混淆配置文件也放置在app壳工程中，各个业务组件的混淆配置规则都应该在app壳工程中的混淆配置文件中添加和修改。

之所以不采用在每个业务组件中开启混淆的方案，是因为 **组件在集成模式下都被 Gradle 构建成了 release 类型的arr包**，一旦业务组件的代码被混淆，而这时候代码中又出现了bug，将很难根据日志找出导致bug的原因；另外每个业务组件中都保留一份混淆配置文件非常不便于修改和管理，这也是不推荐在业务组件的 build.gradle 文件中配置 buildTypes （构建类型）的原因。

六、工程的build.gradle和gradle.properties文件

1、组件化工程的build.gradle文件

在组件化项目中因为每个组件的 build.gradle 都需要配置 compileSdkVersion、buildToolsVersion和 defaultConfig 等的版本号，而且每个组件都需要用到 annotationProcessor，**为了能够使组件化项目中的所有组件的 build.gradle 中的这些配置都能保持统一，并且也是为了方便修改版本号**，我们统一在Android工程根目录下的build.gradle中定义这些版本号，当然为了方便管理Common组件中的第三方开源库的版本号，最好也在这里定义这些开源库的版本号，然后在各个组件的build.gradle中引用Android工程根目录下的build.gradle定义的版本号，组件化工程的 build.gradle 文件代码如下：

```
buildscript {
    repositories {
        jcenter()
        mavenCentral()
    }

    dependencies {
        //classpath "com.android.tools.build:gradle:$localGradlePluginVersion"
        //$localGradlePluginVersion是gradle.properties中的数据
        classpath "com.android.tools.build:gradle:$localGradlePluginVersion"
    }
```

```
}

allprojects {
    repositories {
        jcenter()
        mavenCentral()
        //Add the JitPack repository
        maven { url "https://jitpack.io" }
        //支持arr包
        flatDir {
            dirs 'libs'
        }
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}

// Define versions in a single place
//时间：2017.2.13；每次修改版本号都要添加修改时间
ext {
    // Sdk and tools
    //localBuildToolsVersion是gradle.properties中的数据
    buildToolsVersion = localBuildToolsVersion
    compileSdkVersion = 25
    minSdkVersion = 16
    targetSdkVersion = 25
    versionCode = 1
    versionName = "1.0"
    javaVersion = JavaVersion.VERSION_1_8

    // App dependencies version
    supportLibraryVersion = "25.3.1"
    retrofitVersion = "2.1.0"
    glideVersion = "3.7.0"
    loggerVersion = "1.15"
    eventbusVersion = "3.0.0"
    gsonVersion = "2.8.0"
    photoViewVersion = "2.0.0"

    //需检查升级版本
    annotationProcessor = "1.1.7"
    routerVersion = "1.2.2"
    easyRecyclerVersion = "4.4.0"
    cookieVersion = "v1.0.1"
    toastyVersion = "1.1.3"
}
12345678910111213141516171819202122232425262728293031323334353637383940414243444
5464748495051525354555657585960
```

2、组件化工程的gradle.properties文件

在组件化实施流程中我们了解到gradle.properties有两个属性对我们非常有用：

1、在Android项目中的任何一个build.gradle文件中都可以把gradle.properties中的常量读取出来，不管这个build.gradle是组件的还是整个项目工程的build.gradle；

2、gradle.properties中的数据类型都是String类型，使用其他数据类型需要自行转换；

利用gradle.properties的属性不仅可以解决集成开发模式和组件开发模式的转换，而且还可以解决在多人协同开发Android项目的时候，因为开发团队成员的Android开发环境（开发环境指Android SDK和AndroidStudio）不一致而导致频繁改变线上项目的build.gradle配置。

在每个Android组件的 build.gradle 中有一个属性：**buildToolsVersion**，表示构建工具的版本号，这个属性值对应 AndroidSDK 中的 **Android SDK Build-tools**，正常情况下 build.gradle 中的 buildToolsVersion 跟你电脑中 Android SDK Build-tools 的最新版本是一致的，比如现在 Android SDK Build-tools 的最新的版本是：25.0.3，那么我的Android项目中 build.gradle 中的 buildToolsVersion 版本号也是 25.0.3，但是一旦一个Android项目是由好几个人同时开发，总会出现每个人的开发环境 Android SDK Build-tools 是都是不一样的，并不是所有人都会经常升级更新 Android SDK，而且代码是保存到线上环境的（例如使用 SVN/Git 等工具），某个开发人员提交代码后线上Android项目中 build.gradle 中的 buildToolsVersion 也会被不断地改变。

另外一个原因是因为Android工程的根目录下的 build.gradle 声明了 Android Gradle 构建工具，而这个工具也是有版本号的，而且 **Gradle Build Tools** 的版本号跟 AndroidStudio 版本号一致的，但是有些开发人员基本很久都不会升级自己的 AndroidStudio 版本，导致团队中每个开发人员的 Gradle Build Tools 的版本号也不一致。

如果每次同步代码后这两个工具的版本号被改变了，开发人员可以自己手动改回来，并且不要把改动工具版本号的代码提交到线上环境，这样还可以勉强继续开发；但是很多公司都会使用持续集成工具（例如Jenkins）用于持续的软件版本发布，而Android出包是需要 Android SDK Build-tools 和 Gradle Build Tools 配合的，一旦提交到线上的版本跟持续集成工具所依赖的Android环境构建工具版本号不一致就会导致Android打包失败。

为了解决上面问题就必须将Android项目中 build.gradle 中的 buildToolsVersion 和 GradleBuildTools 版本号从线上代码隔离出来，保证线上代码的 buildToolsVersion 和 Gradle Build Tools 版本号不会被人为改变。

七、组件化项目Router的其他方案-ARouter

在组件化项目中使用到的跨组件跳转库ActivityRouter可以使用阿里巴巴的开源路由项目：阿里巴巴 ARouter；

ActivityRouter和ARouter的接入组件化项目的方式是一样的，ActivityRouter提供的功能目前ARouter也全部支持，但是ARouter还支持依赖注入解耦，页面、拦截器、服务等组件均会自动注册到框架。对于大家来说，没有最好的只有最适合的，大家可以根据自己的项目选择合适的Router。

下面将介绍ARouter的基础使用方法，更多功能还需大家去Github自己学习；

1、首先 ARouter 这个框架是需要初始化SDK的，所以你需要在“app壳工程”中的应用Application中加入下面的代码，**注意：在 debug 模式下一定要 openDebug**：

```
    if (BuildConfig.DEBUG) {
            //一定要在ARouter.init之前调用openDebug
            ARouter.openDebug();
            ARouter.openLog();
       }
       ARouter.init(this);123456
```

2、首先我们依然需要在 Common 组件中的 build.gradle 将ARouter 依赖进来，方便我们在业务组件中调用：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    //router
    compile 'com.alibaba:arouter-api:1.2.1.1'
}12345
```

3、然后在每一个**业务组件**的 build.gradle 都引入ARouter 的 Annotation处理器，代码如下：

```
android {
    defaultConfig {
    ...
    javaCompileOptions {
        annotationProcessorOptions {
        arguments = [ moduleName : project.getName() ]
        }
    }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    annotationProcessor 'com.alibaba:arouter-compiler:1.0.3'
}12345678910111213141516
```

4、**由于ARouter支持自动注册到框架，所以我们不用像ActivityRouter那样在各个组件中声明组件，当然更不需要在Application中管理组件了。** 我们给 Girls组件 中的 GirlsActivity 添加注解：**@Route(path = “/girls/list”)**，需要注意的是**这里的路径至少需要有两级，/xx/xx**，之所以这样是因为**ARouter使用了路径中第一段字符串(/*/)作为分组**，比如像上面的”girls”，而分组这个概念就有点类似于ActivityRouter中的组件声明 @Module ，代码如下：

```
@Route(path = "/girls/list")
public class GirlsActivity extends BaseActionBarActivity {

    private GirlsView mView;
    private GirlsContract.Presenter mPresenter;

    @Override
    protected int setTitleId() {
        return R.string.girls_activity_title;
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mView = new GirlsView(this);
```

```
        setContentView(mView);
        mPresenter = new GirlsPresenter(mView);
        mPresenter.start();
    }
}12345678910111213141516171819201
```

然后我们就可以**在项目中的任何一个地方通过 URL地址： /girls/list, 调用 GirlsActivity**，方法如下：

```
      ARouter.getInstance().build("/girls/list").navigation();1
```

八、结束语

组件化相比于单一工程优势是显而易见的：

1. 组件模式下可以加快编译速度，提高开发效率；
2. 自由选择开发框架（MVC /MVP / MVVM /）；
3. 方便做单元测试；
4. 代码架构更加清晰，降低项目的维护难度；
5. 适合于团队开发；