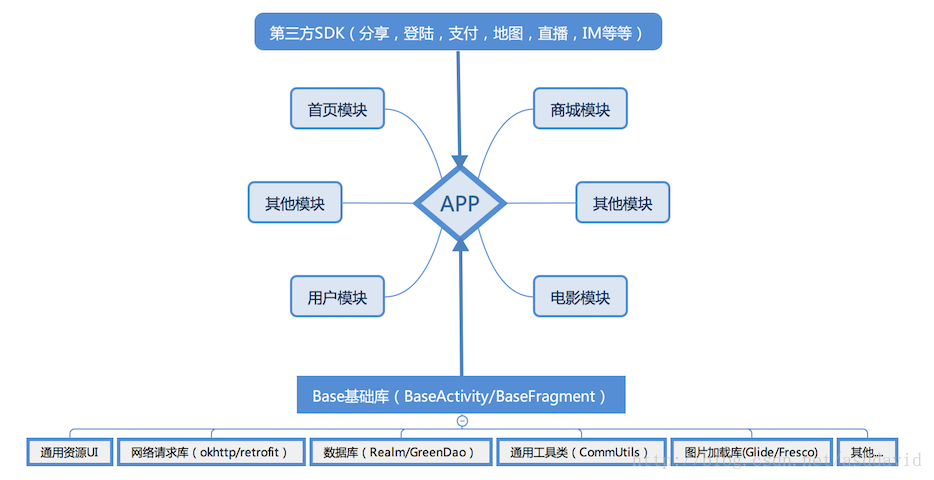
前言   
鉴于个人文笔有限，上篇博文Android组件化文章写的太烂[Android组件化、模块化开发](http://blog.csdn.net/asddavid/article/details/53436848" \t "_blank)图文以及解释做的太过粗糙。   
本篇咱们根据图表对比，优缺点，讲述具体的实现步骤以及gradle自动化脚本的书写等。

[**组件化案例视频代码传送门:https://www.jikexueyuan.com/zhiye/course/84.html?type=18**](https://www.jikexueyuan.com/zhiye/course/84.html?type=18)

为什么组件化   
随着移动互联网的发展，或许中小型项目还可以用单工程+MVC/MVP/MVVM的架构来完成，但当项目到了一定程度之后，**编译时间** 原来越长，**测试或者开发任何一个模块功能都需要整个项目重启运行。**

常规单工程+MVC/MVP/MVVM项目：



乍一看，这样的结构只要咱们模块分层明确，是不存在大问题的，但是随着业务的快速迭代，面临以下问题：

1.需求疯狂变化，上周刚讨论出一套方案，你花了两天搞定，这个时候PM告诉你，这个咱们修改或者不要了，是否想抓狂呢。

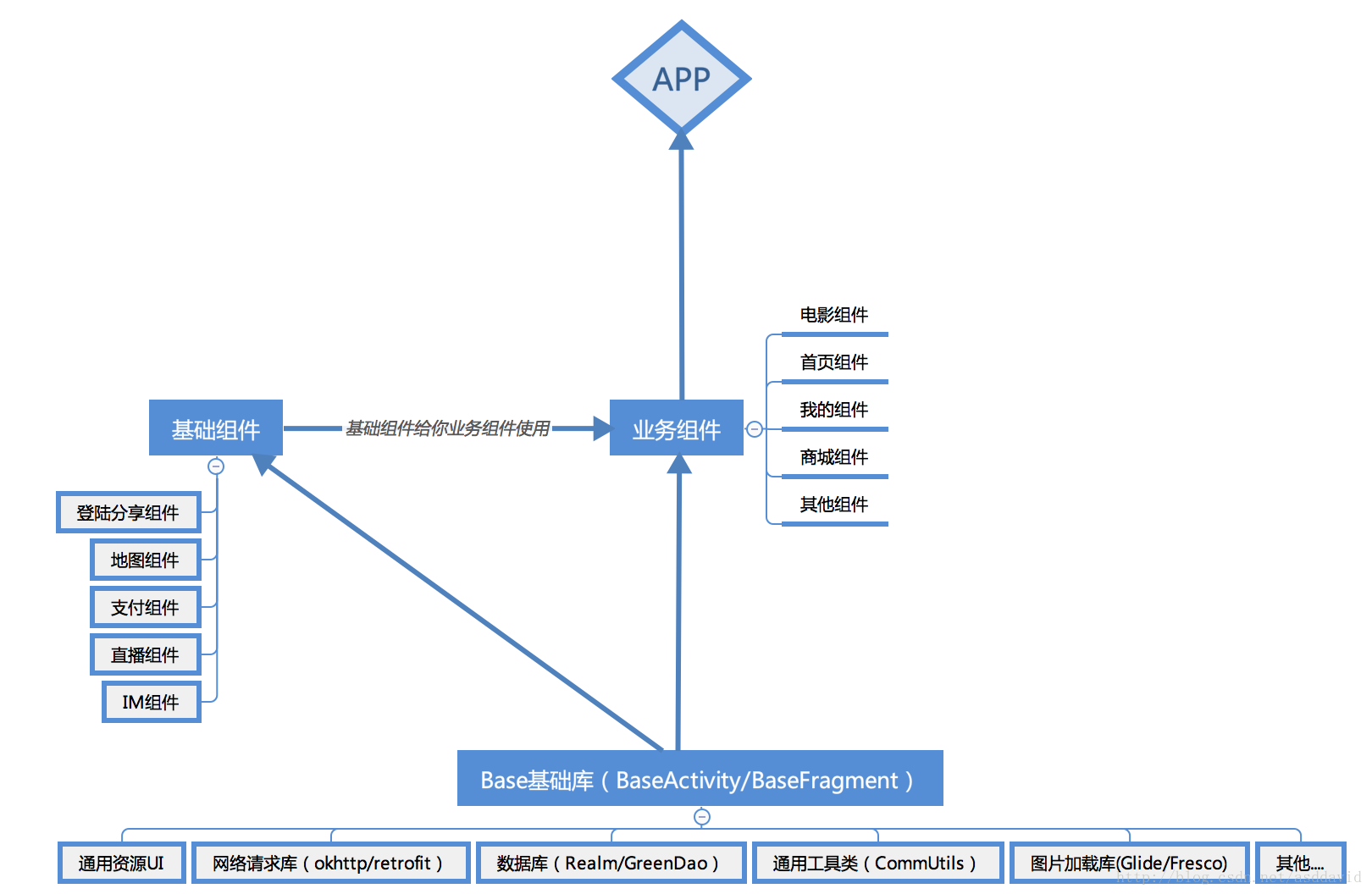
2.所有业务都在一个项目，不管基于什么原因，有时候咱们为了快速完成一个功能，或多或少存在耦合，任何改动都可能显的比较吃力，解决了一个BUG又出现另外一个BUG。

3.团队人数达到一定程度后，并行开发过程中，如果某个成员不小心犯错并且提交了代码，可能导致项目暂时无法运行，不得不停下来协同查找问题，严重影响开发效率

4.业务越来越多，项目越来越大，编译运行一次要10秒…20秒…1分钟…5分钟…累计几个月下来的时间说不得抽出来都可以去找个女朋友了…

基于以上问题，咱们的组件化应运而生。

组件化结构图



对比上张图，这里的APP主要由业务组件构成，严格来说这5个业务组件也可以是5个App,当实现以上架构图，看看组件化的优缺点：

组件化优点

* 业务组件可以单独分配并行开发
* 单个组件业务可以由开发者自行决定采取MVC/MVP/MVVM架构而不影响整体大局
* 新人接手项目分配任务可单独分配某一个模块任务，不必关心整个项目
* 开发效率提升，开发过程仅仅需要维护开发自己的组件内容
* 若公司有多个团队，优秀代码组件可快速移植复用
* 积累个人的组件仓库，摆脱粘贴复制的“搬砖工”身份
* 测试可单独测试某个模块

组件化的坑

* 组件与组件之间的调用，数据等交互
* 多个组件，在使用application的时候怎办
* 多个组件资源命名重复
* 多个组件引用不同版本的相同的库

了解了优缺点，咱们进入正式的组件化开发集成，后续将会描述如何解决组件化的一些坑。

前文说过，咱们的5个组件可以理解为5个app，下面开始集成。

先看看咱们的组件化效果，手机展示效果，   


**1：首先统一组件之间的版本以及第三方库版本**   
利用Gradle统一版本号，可参考[android使用Gradle统一配置依赖版本](http://blog.csdn.net/asddavid/article/details/53322689" \t "_blank)

**2:咱们的组件又是Lib，又是application，如何控制调试，如何在主APP选择**

在config.build处新增一个布尔isBuildApp作为标志判断依赖，true表示作为application存在，false表示lib存在

ext {

isBuildApp=false;//false:作为Lib组件存在， true:作为application存在

...

}

**在每个组件的build根据isBuildApp来选择依赖**

if(rootProject.ext.isBuildApp){

apply plugin: "com.android.application"

}else{

apply plugin: 'com.android.library'

}

android{

...

defaultConfig {

if(rootProject.ext.isBuildApp){

applicationId "com.allure.shop"

}

...

}

}

**ibrary与application运行时需要manifest,依然根据isBuildApp判断**

sourceSets {

main {

if (rootProject.ext.isBuildApp) {

manifest.srcFile 'src/main/debug/AndroidManifest.xml'

} else {

manifest.srcFile 'src/main/release/AndroidManifest.xml'

java {

exclude 'debug/\*\*'

}

}

}

}

**资源的命名为了避免重复，建议按照组件名开头，如Login组件，命名login\_xxx，BaiDuMap组件命bd\_map\_xxx**   
可用gradle进行强制检测

resourcePrefix "login\_"

**主项目的引用**

if (rootProject.ext.isBuildApp) {

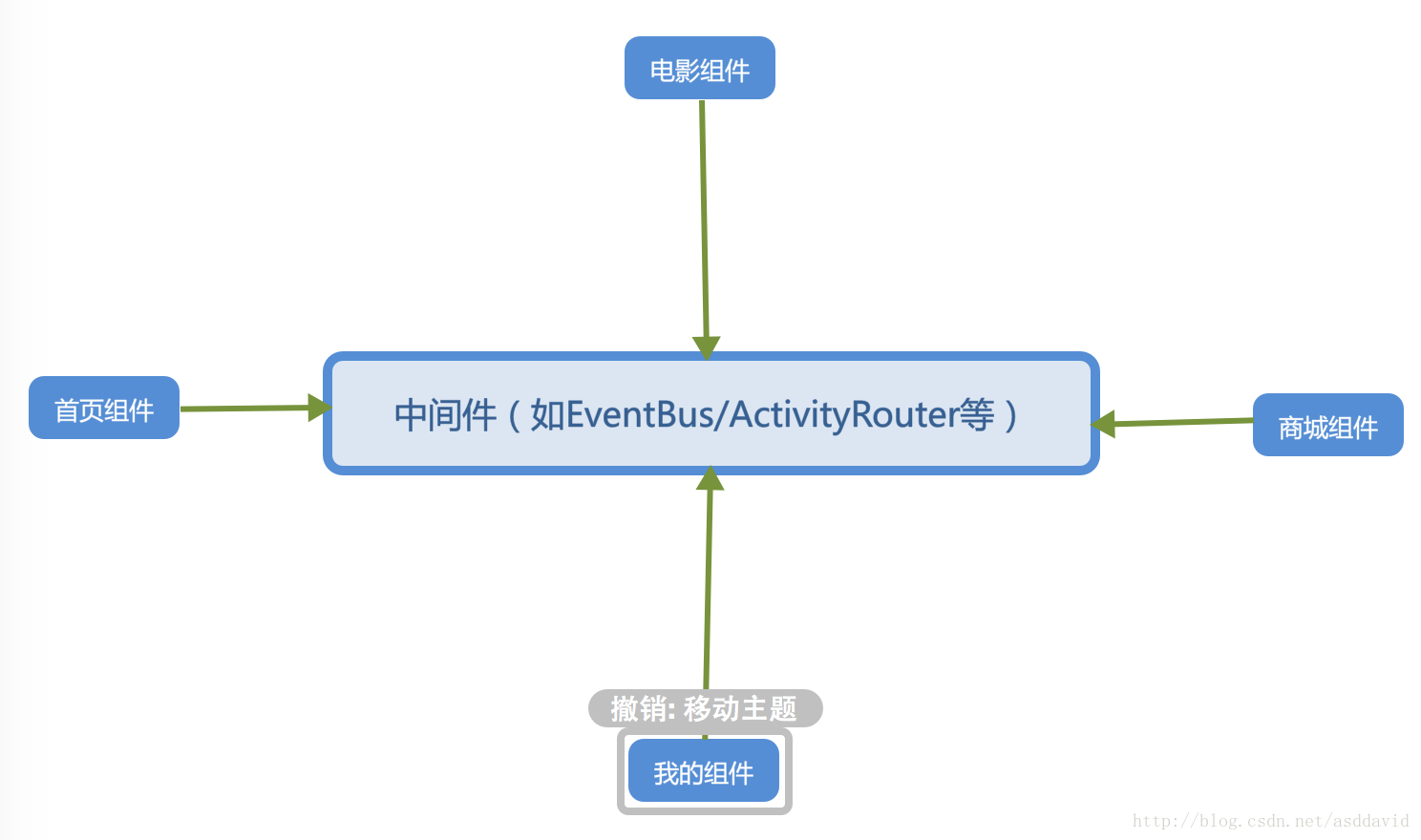
compile project(':modulebase')

} else {

compile project(':modulecore:moduleLogin')

compile project(':modulecore:moduleShop')

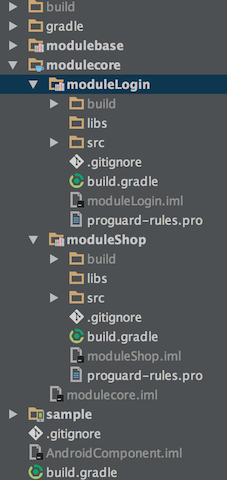
}

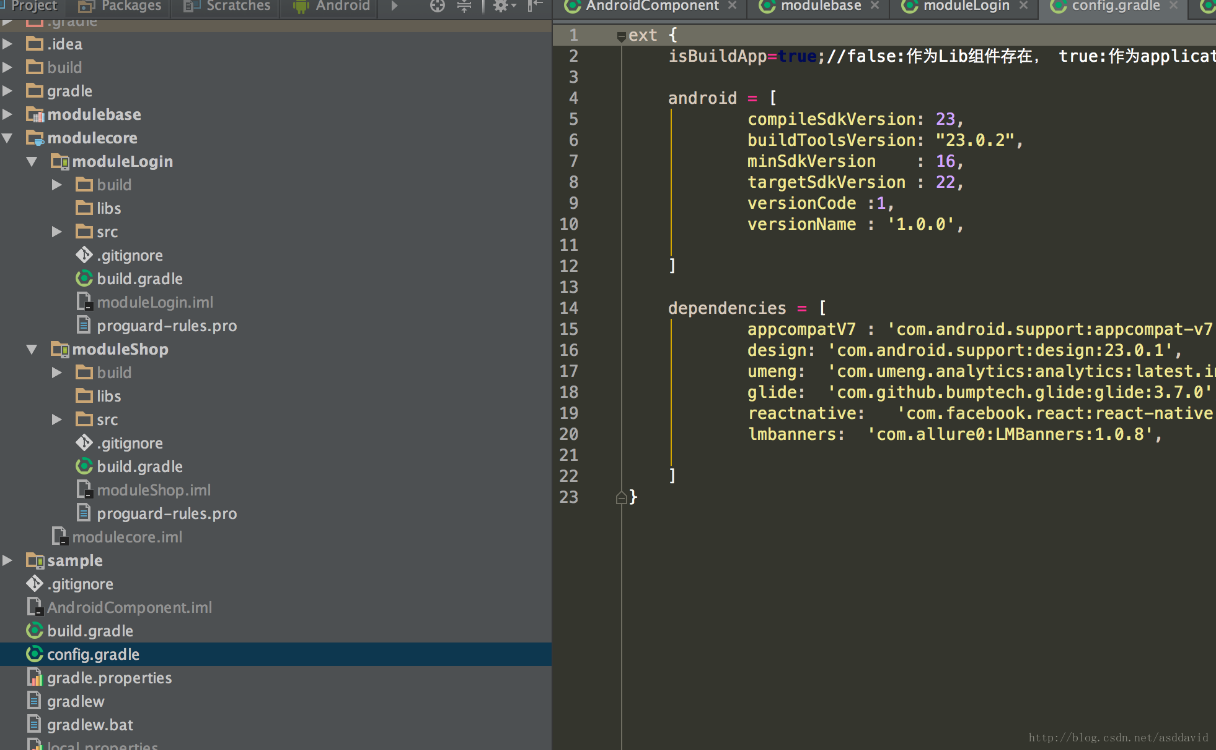
**解决组件与组件的交互：**   
方案1：可采用建立中间件的方式来统一管理组件之间的交互,如电影组件与首页组件需要跳转传值等可采用开源的[ActivityRouter](https://github.com/mzule/ActivityRouter" \t "_blank)，[EventBus](https://github.com/greenrobot/EventBus" \t "_blank)来完成   


方案2：在主项目APP建立统一的入口类，针对组件与组件的交互建立方法，实现接口等，但此方式有一定沟通成本，组件与组件之间的交互维护可能需要一份文档来约束。

**application的使用：**   
方案1：统一使用基础库的单例BaseApplication   
方案2：反射ActivityThread

**Lib与Application的切换**   
修改config.build里的isBuildApp属性并且重新sync

项目结构图：   
作为组件Lib   


作为单独的application   


**总结**

组件化技术难度不大，难点在于业务的解耦。具体是否选择组件化方式还是要根据项目大小来确定。 当然采取了组件化是极好的。   
本身组件化是应用于复杂业务的场景，DEMO也不大好做，简单的从项目抽取做了一份案例，后续考虑在此基础上更新

[Github源码地址](https://github.com/Allure0/AndroidComponent)