Android实战之组件化开发：

问题场景：

笔者一直想使用组件化开发框架来进行实现模块解耦，这一点在协作开发的时候很有用，比如ABC三人开发一款app，其中A需要用到BC的功能，B需要用到AC的功能，C需要用到A功能，那么问题来了，如果ABC同时开发各自模块，那么谁都不想等待，怎么解决这个问题？

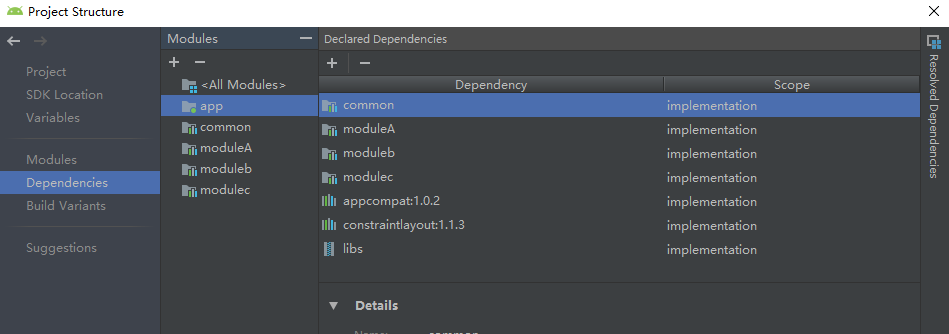
解决方法：

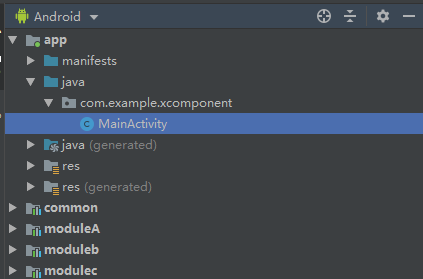
很多人会想到ABC定义业务接口，先提供出去，然后给需要的调用，这个方案是好的，我们在提供接口的时候，需要定义各种模块，下面我们开始编写一个组件化开发架构（目前笔者项目正在切换的架构，笔者认为不错，打算模拟源码实现开源，这样的结构够我玩几年了~）

下面我们就开始组件化开发吧：

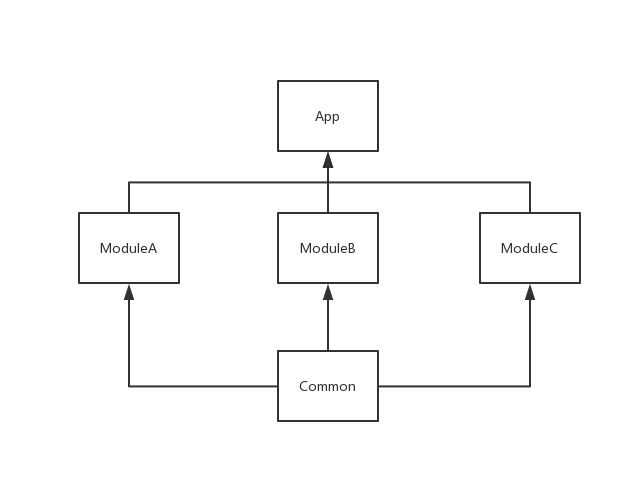
首先创建一个app项目，同时创建moudleA,moduleB,moduleC模块，app模块依赖moudleA,moduleB,moduleC模块，同时创建一个common作为base能力库，这个模块用于我们后面编写组件化核心代码

项目结构如下：





其中moudleA,moduleB,moduleC, common模块作为lib存在（apply plugin: 'com.android.library'）,图解如下：



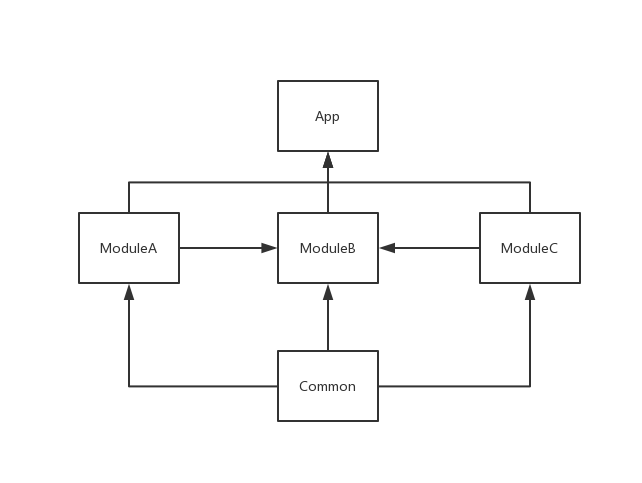
现在创建了ModuleA,B,C模块，但是目前了结构无法满足我们的要求，ABC之间需要定义接口

下面开始我们的编码：

在moudleA里面定义外部接口A，moudleB里面定义外部接口B，moudleC里面定义外部接口C，如果ModuleB需要用到A或者C的接口，需要引用A,C项目

implementation project(path: ':moduleA')   
implementation project(path: ':modulec')

那么B的结构看上去如下：



这样MoudleB就可以用到AC的接口，但是有心的读者可以发现一个问题，那这样我还要区分moudleA，moudleB，moudleC干什么？他们互相依赖，并且处于同一层那就相当于合在了一个Lib里面，彼此相互依赖无法拆分，这样的定义模块就没有意义了

笔者也认为确实是这样的，我们还需要进行进一步改造，争取让B只能访问AC的接口，对于moduleA，moduleC的具体实现不关心

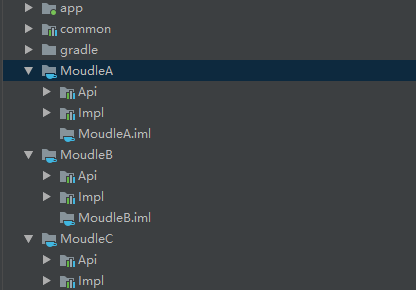
实现方法：

我们将MoudleA与MoudleC再次拆分，分成ModuleA-API与ModuleA-IMPL，将ModuleA-API提供给B使用，MoudleC同样的方法拆分

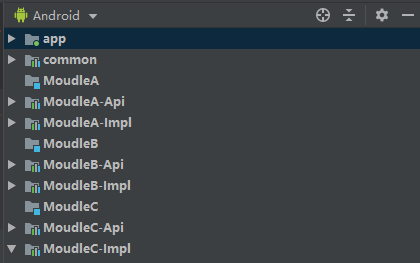
这里我们需要创建文件夹了，

首先在我们的工程目录下面创建moudleA,moduleB,moduleC文件夹，通知将之前的moudle分别嵌入对应目录，创建APImoudule与Implmoudle最后修改settings.gradle，看上去像这样：

Project这样：



Android表现如下：

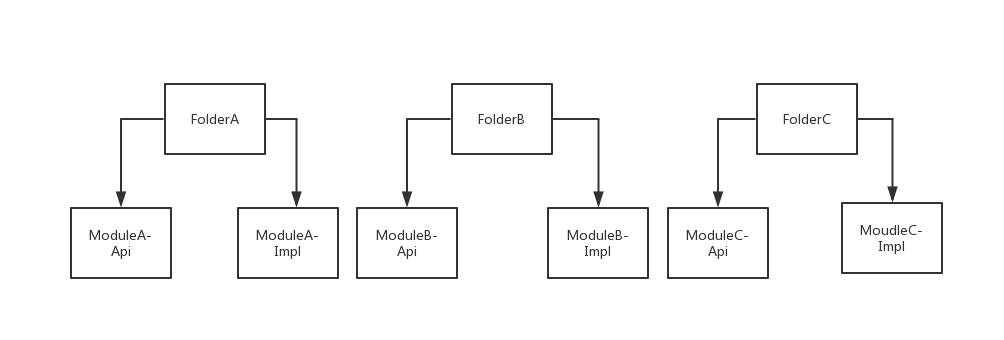


Setting配置如下：

include ':app', ':common',  
 ":MoudleA:Api",":MoudleA:Impl",  
 ":MoudleB:Api",":MoudleB:Impl",  
 ":MoudleC:Api",":MoudleC:Impl"  
rootProject.name='ComponentApp'

意思就是我们在每个模块加入一个父目录

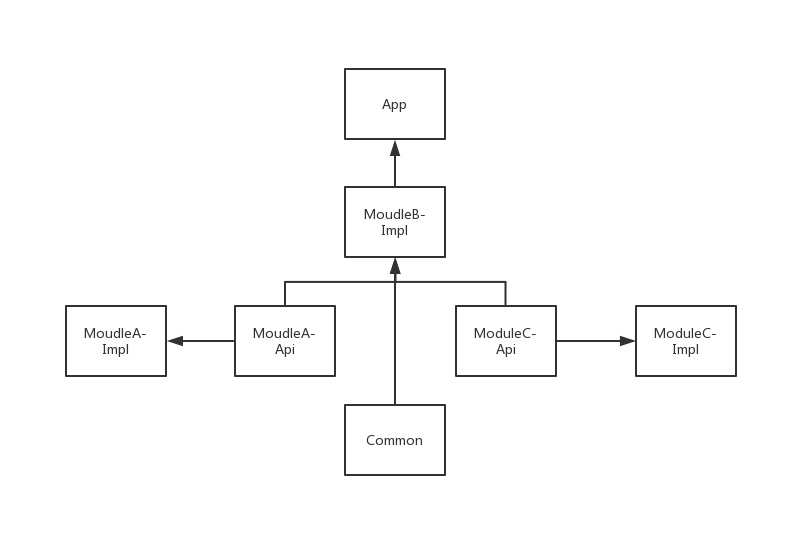
结构看上去如下：



其中Folder只是父目录，不是工程，其他都是工程

这样的话，如果ModuleB需要使用A与C的接口，只需要依赖MoudleA/C – Api模块就可以，不需要依赖Impl的具体实现，而MoudleA/C-Impl代表具体实现其Api接口

那么结构就变成这样了：



模块B需要用到moduleA,moduleC的接口，用到基础库Common，而真正具体实现ModuleA/C由协助者同步开发，这样就不产生过度耦合，注意：图中ModuleA/C-Impl只是具体实现，不会被其他模块直接使用，而我们最后将所有组件的Api与Impl将注册到总组件Common中，这样每个模块Impl可以拿到Api接口的具体实现

下面我们开始编写模拟代码

1.创建ModuleABC的接口InterfaceABC

2.MoudleB-Impl项目添加ModuleA-Api与ModuleC-Api依赖，用于调用其他模块功能，添加ModuleB-Api，用于具体实现自己外部提供的接口

implementation project(path: ':MoudleB:Api')  
implementation project(path: ':MoudleC:Api')  
implementation project(path: ':MoudleA:Api')

3.

3.1 先实现自己模块功能，ModuleB-Api，我们定义InterfaceB，提供一个printModuleB接口方法：

*/\*\*  
 \* 用于其他模块调用  
 \*/*public interface InterfaceB {  
 void printModuleB();  
}

3.2 ModuleB-Impl定义Impl实现类

public class ModuleBImpl implements InterfaceB {  
 @Override  
 public void printModuleB() {  
 Log.*i*("MoudleImpl", "print ModuleB");  
 }  
}

4. 那么MoudleAC-Impl引用了ModuleB-Api，如何获取其具体实现呢？

下面就是重点了，我们需要将所有Api与Impl链接起来，并且注册到一个组件存储器里面，而外部只能拿到接口

4.1 步骤4需要定义在Common库里面，并且所有Impl都有依赖Common库，用于获取Api服务，定义一个存储器HashMap<String,Object>存储api名与impl实现，并且提供注册方法registerApiAndImpl，有了注册，还需要提供获取接口方法。定义ComponentServiceStore类，代码看起来如下：

public class ComponentServiceStore implements IComponentService {  
  
 private Map<String, Object> apiStores = new HashMap<>();  
  
 @Override  
 public <T> void registerApiAndImpl(Class<T> api, Class<? extends T> impl) {  
 if (api.isInterface() && !impl.isInterface()) {  
 Object o = NewInstanceFactory.*create*(impl);  
 apiStores.put(api.getCanonicalName(), o);  
 } else {  
 throw new IllegalStateException("impl is not api subclass");  
 }  
 }  
  
 public void init(ApiService.IRegisterCallBack callBack) {  
 callBack.onInit(this);  
 }  
  
 <T> T getServiceImpl(Class<T> apiClazz) {  
 String canonicalName = apiClazz.getCanonicalName();  
 Object o = apiStores.containsKey(canonicalName) ? apiStores.get(canonicalName) : null;  
 return o != null ? (T) o : null;  
 }  
  
 public static class NewInstanceFactory {  
 @SuppressWarnings("ClassNewInstance")  
 @NonNull  
 public static <T> T create(@NonNull Class<T> modelClass) {  
 // noinspection TryWithIdenticalCatches  
 try {  
 return modelClass.newInstance();  
 } catch (InstantiationException e) {  
 throw new RuntimeException("Cannot create an instance of " + modelClass, e);  
 } catch (IllegalAccessException e) {  
 throw new RuntimeException("Cannot create an instance of " + modelClass, e);  
 }  
 }  
 }  
}

代码就是一个简单的map存取功能，这就是核心实现，对此我们还需要扩展包装一下，提供一个初始化方法：

public class ApiService {  
  
 private static ComponentServiceStore *componentService*;  
  
 public interface IRegisterCallBack {  
 void onInit(IComponentService componentService);  
 }  
  
 public static void initService(IRegisterCallBack callBack) {  
 // init componentService  
 if (*componentService* == null) {  
 *componentService* = new ComponentServiceStore();  
 }  
 *componentService*.init(callBack);  
 }  
  
 public static <T> T getServiceImpl(Class<T> apiClazz) {  
 if (*componentService* == null) {  
 throw new IllegalStateException("you must call ApiComponentService#initService first");  
 }  
 return *componentService*.getServiceImpl(apiClazz);  
 }  
}

上面就是注册组件化的核心代码，我们可以在自定义Application#onCreate里面进行初始化，使用方法如下：（注意，App模块目前需要将所有模块都依赖进去，这里后续我们还需要改造，如果app删除某个模块，最好我们只需要删除gradle里面依赖不改动代码就好）

public class XApplication extends Application {  
  
 @Override  
 public void onCreate() {  
 super.onCreate();  
 initService();  
 }  
 private void initService() {  
 ApiService.*initService*(new ApiService.IRegisterCallBack() {  
 @Override  
 public void onInit(IComponentService componentService) {  
 componentService.registerApiAndImpl(InterfaceA.class, ModuleAImpl.class);  
 componentService.registerApiAndImpl(InterfaceB.class, ModuleBImpl.class);  
 componentService.registerApiAndImpl(InterfaceC.class, ModuleCImpl.class);  
 }  
 });  
 }  
}

所有组件都在App模块里面注册好了，那么我们如果在MoudleB-Impl模块使用其他某块呢

1. MoudleB-Impl已经依赖ModuleA-Api与ModuleC-Api

implementation project(path: ':MoudleC:Api')  
implementation project(path: ':MoudleA:Api')

2. 获取其他模块接口：

public class ModuleBImpl implements InterfaceB {  
 @Override  
 public void printModuleB() {  
 Log.*i*("MoudleImpl", "print ModuleB");  
  
 //这里我们调用一下模块C的接口实现  
 new TestMocker().testModuleC();  
 }  
}

public class TestMocker {  
 public void testModuleC() {  
 InterfaceC implC = ApiService.*getServiceImpl*(InterfaceC.class);  
 if (implC != null) {  
 implC.printModuleC();  
 }  
 }  
}

我们在组件B的Impl实现里面去获取C的接口并调用C的接口方法，看看C-Impl是否实现了接口



好了，上面就是组件化解耦，笔者目前项目正在迁移的框架简介（很有用哦~）

这里遗留一个小问题，有兴趣的读者可以研究一下：

前面我们提到的，app模块如果注册所有api与impl需要将模块全部引入，如果删除某个模块还需要手动修改报错代码，这里我们是否可以能够在动态添加删除模块的时候，不修改代码，让组件注册的时候自动加载实例化，如果加载不到那就不加载呢？

感谢阅读，有不当之处或者建议，评论回复，笔者定努力完善。