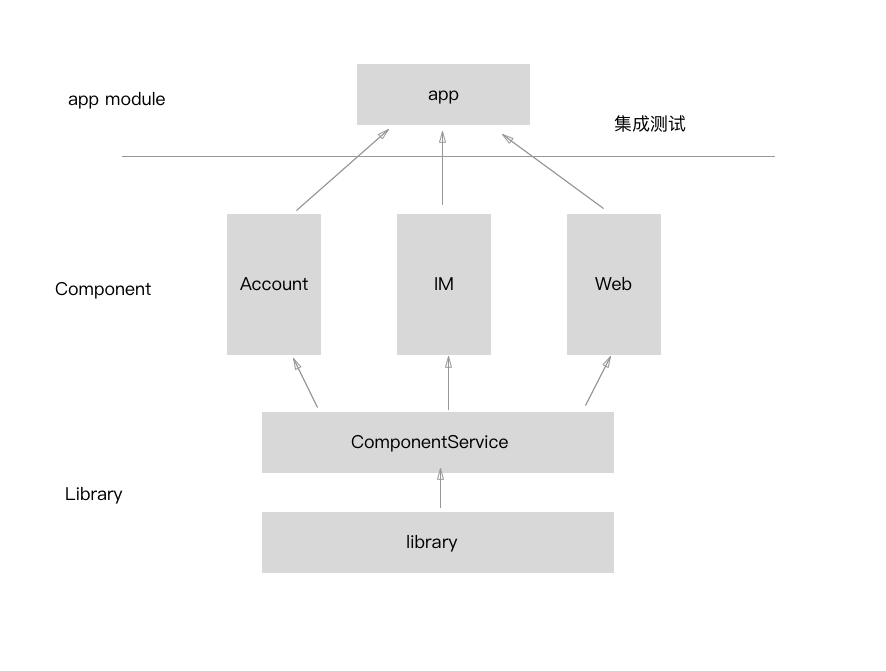
# 组件化方案介绍



## 一. 什么是组件化

组件，可以简单理解为独立的功能模块，所以，对于项目来说，组件化就是将项目拆分成多个相互独立的功能模块。

组件化的核心思想是分而治之，解耦分层。

## 二. 为什么要使用组件化

1. 极大地缩短编译时间。理想的组件化是，开发阶段，按需编译，每个组件只依赖最小的依赖库（如基础 UI 类、基础样式、基础工具类以及网络库等），这样编译速度会快很多。

2. 并行开发。因为组件相互独立， 人员多的团队可以实现并行开发，极大地提高开发效率

3. 测试更加高效。对于业务复杂，代码耦合严重的项目，每个改动都需要进行大范围的回归测试。而组件化之后，只需要对修改的组件进行测试即可。

## 三. 如何实现组件化

要实现组件化，主要考虑一下几个方面：

- 代码解耦。如何将一个庞大的工程拆分成有机的整体？

- 组件单独调试。每个组件都是一个完整的整体，如何让其单独运行和调试呢？

- 数据传递。因为每个组件都会给其他组件提供的服务，那么主项目（Host）与组件、组件与组件之间如何传递数据？

- UI跳转。UI跳转可以认为是一种特殊的数据传递

- 集成调试。在开发阶段如何做到按需的编译组件？一次调试中可能只有一两个组件参与集成，这样编译的时间就会大大降低，提高开发效率。

- 代码隔离。如何从根本上避免组件之间的直接引用呢？只有做到这一点才是彻底的组件化。

### 1. 代码解耦

Android Studio 工程支持多 module。我们对两种 module 进行区分

- 基础库 library，这些代码被其他组件直接引用。如网路库、图片加载库、常用工具类等

- Component（组件），这种 module 是一个完整的功能模块。比如 IM模块、用户模块等

将 library 称之为依赖库，Component 称为组件，而负责拼装这些组件的形成完整 app 的 module，称之为主项目或主 module 或 app module

### 2. 组件单独调试

要实现单独调试需要对组件 module 做一些修改：

1. 修改组件的 `build.gradle` 把 `apply plugin: 'com.android.library'` 换成 `apply plugin: 'com.android.application'`

2. 新建一个 AndroidManifest，增加入口 Activity

3. 为了避免组件资源冲突，在组件的 `build.gradle`中增加 `resourcePrefix xxx\_`，从而固定每个组件的资源前缀。

### 3. 数据传递 和 UI 跳转

主项目和组件、组件与组件之间不能直接使用类的相互引用来进行数据交互。那么如何做到这个隔离呢？

可以采用接口+实现的结构。每个组件声明自己提供的服务Service，这些Service都是一些抽象类或者接口，

组件负责将这些Service实现并注册到一个统一的路由Router中去。要使用某个组件的功能，只需向Router

请求这个Service的实现，具体的实现细节我们全然不关心，只要能返回我们需要的结果就可以了。

数据传递和 UI 跳转我们使用 ARouter 这个三方库实现。具体可参看 ARouter 介绍

### 4. 集成调试

在开发后期我们需要把几个组件机集成到一个app里面去验证。由于我们上面的机制保证了组件之间的隔离，

所以我们可以任意选择几个组件参与集成。这种按需索取的加载机制可以保证在集成调试中有很大的灵活性，

并且可以加大的加快编译速度。

### 5. 代码隔离

最终的集成测试还是需要在 app module 引入各个组件的依赖，我们可以通过编写 gradle 插件，

在编译期间引入依赖，以避免由于人为因素导致的直接依赖实现类的问题。

# ARouter 使用指南  
  
ARouter 是一个用于帮助 Android App 进行组件化改造的框架 —— 支持模块间的路由、通信、解耦  
  
## ARouter 配置  
  
1. 在 module (如app) `build.gradle` 添加依赖。\*\*\*注意，如果有多个 module 用到`ARouter`注解，每个 module 都需要添加\*\*\*  
  
```  
android {  
 defaultConfig {  
 ...  
 // 用到注解的需要加  
 javaCompileOptions {  
 annotationProcessorOptions {  
 arguments = [AROUTER\_MODULE\_NAME: project.getName()]  
 }  
 }  
 }  
}  
  
dependencies {  
 // 这个最好放到基础库，如果基础库已经有，module这里就不用再添加  
 api 'com.alibaba:arouter-api:x.x.x'  
   
 // 用到注解的需要加依赖  
 annotationProcessor 'com.alibaba:arouter-compiler:x.x.x'  
 ...  
}  
```  
  
2. 初始化 SDK  
  
注意：最好在 Application 中初始化，debug 模式下开启调试模式  
  
```  
private void initARouter(Application app) {  
 // 开启调试必须在 init 之前调用，否则会无效  
 if (BuildConfig.DEBUG) {  
 ARouter.openLog(); // 开启日志  
 ARouter.openDebug(); // 开启调试模式(如果在InstantRun模式下运行，必须开启调试模式！线上版本需要关闭,否则有安全风险)  
 }  
  
 ARouter.init(app);  
}  
```  
  
3. 混淆规则（proguard）  
  
```  
-keep public class com.alibaba.android.arouter.routes.\*\*{\*;}  
-keep public class com.alibaba.android.arouter.facade.\*\*{\*;}  
-keep class \* implements com.alibaba.android.arouter.facade.template.ISyringe{\*;}  
  
# 如果使用了 byType 的方式获取 Service，需添加下面规则，保护接口  
-keep interface \* implements com.alibaba.android.arouter.facade.template.IProvider  
  
# 如果使用了 单类注入，即不定义接口实现 IProvider，需添加下面规则，保护实现  
# -keep class \* implements com.alibaba.android.arouter.facade.template.IProvider  
```  
  
4. 使用 gradle 插件实现路由表自动加载（可选）  
  
可选使用，通过 ARouter 提供的注册插件进行路由表的自动加载(power by [AutoRegister](https://github.com/luckybilly/AutoRegister))， 默认通过扫描 dex 的方式  
进行加载通过 gradle 插件进行自动注册可以缩短初始化时间解决应用加固导致无法直接访问  
dex 文件，初始化失败的问题，需要注意的是，该插件必须搭配 api 1.3.0 以上版本使用！  
  
project `build.gradle`  
  
```  
buildscript {  
 repositories {  
 jcenter()  
 }  
  
 dependencies {  
 classpath "com.alibaba:arouter-register:1.0.2"  
 }  
}  
```  
  
module `build.gradle`  
  
```  
apply plugin: 'com.alibaba.arouter'  
```  
  
5. 生成路由文档  
  
module `build.gradle`  
  
```  
  
// 添加参数 AROUTER\_GENERATE\_DOC = enable  
// 生成的文档路径 : build/generated/source/apt/(debug or release)/com/alibaba/android/arouter/docs/arouter-map-of-${moduleName}.json  
   
  
android {  
 defaultConfig {  
 ...  
 // 用到注解的需要加  
 javaCompileOptions {  
 annotationProcessorOptions {  
 arguments = [AROUTER\_MODULE\_NAME: project.getName(), AROUTER\_GENERATE\_DOC: "enable"]  
 }  
 }  
 }  
}  
```

## API 介绍  
  
### 注解  
  
#### 1. `@Route` 路由注解  
  
```java  
  
/\*\*  
\* 作用范围：类。只支持 Activity、Service、Fragment (android.app.Fragment 和 android.support.v4.app.Fragment)、IProvider实现类 4种   
\*/  
public @interface Route {  
  
 /\*\*  
 \* 路径，通过路径可以跳转到 Activity 或者找到类实例  
 \* <b>至少要两层 如: /user/detail</b>  
 \*/  
 String path();  
  
 /\*\*  
 \* 指定分组，默认组名为第一层。如 /user/detail 的默认组名为 user  
 \* 注意：如果主动指定分组，应用内路由需要使用 ARouter.getInstance().build(path, group) 进行跳转  
 \* 不推荐手动指定  
 \*/  
 String group() default "";  
  
 /\*\*  
 \* 用于生成 javadoc 的 description 字段  
 \*   
 \*/  
 String name() default "";  
  
 /\*\*  
 \* 这个属性是一个 int值，换句话说，单个int有4字节，也就是32位，可以配置32个开关  
 \* 剩下的可以自行发挥，通过字节操作可以标识32个开关，通过开关标记目标页面的一些属性，在拦截器中可以拿到这个标记进行业务逻辑判断  
 \*/  
 int extras() default Integer.MIN\_VALUE;  
  
 /\*\*  
 \* 优先级，主要用于 IInterceptor。，  
 \*/  
 int priority() default -1;  
}  
  
```  
  
#### 2. `@interface` 拦截器注解  
  
```java  
  
/\*\*  
\* 作用范围：类  
\*/  
public @interface Interceptor {  
 /\*\*  
 \* 拦截器优先级。多个拦截器会按优先级顺序依次执行，数字越大，优先级越高  
 \*/  
 int priority();  
  
 /\*\*  
 \* 拦截器名字，用于生成 javadoc  
 \*/  
 String name() default "Default";  
}  
  
```  
  
#### 3. `@Autowired` 注入属性注解  
  
```java  
/\*\*  
\* 作用范围：类属性  
\*/  
public @interface Autowired {  
  
 /\*\*  
 \* 设置参数名，或者 服务（IProvider）path  
 \*/  
 String name() default "";  
  
 /\*\*  
 \* 能否必传。设置为 true 之后，必须要传递，否则会 crash  
 \*/  
 boolean required() default false;  
  
 // Description of the field  
 /\*\*  
 \* 用于生成 javadoc 的 description 字段  
 \*/  
 String desc() default "";  
}  
```  
  
### 4. `ARouter` API  
  
```  
ARouter.openLog(); // 开启日志，init 前调用（推荐开发时打开）  
ARouter.openDebug(); // 开启调试模式，init 前调用 (如果在InstantRun模式下运行，必须开启调试模式！线上版本需要关闭,否则有安全风险)  
ARouter.setExecutor(); // 自己提供线程池  
ARouter.setLogger(); // 自己提供 logger  
ARouter.init(getApplication()); // 初始化 （推荐在 Application 调用）  
  
ARouter aRouter = ARouter.getInstance(); // 获取 ARouter 单例  
  
ARouter.getInstance().inject(this); // 注入@Autowired 标注的类属性  
  
AppImpl app = ARouter.getInstance().navigation(AppImpl.class); // 获取 IProvider 子类对象  
  
Postcard postcard = ARouter.getInstance().build("/user/detail"); // 获取一个 Postcard 实例  
  
Postcard postcard2 = ARouter.getInstance().build(Uri.parse("/user/detail")); // 获取一个 Postcard 实例  
  
```  
  
### 5. `Postcard` API  
  
`Postcard.withXxx` 设置参数或者转场动画。  
参数与 Intent 的设置类似，如果要传递普通对象（非序列化），需要实现 Json 序列化，见后文  
  
```  
// 直接传递Bundle  
 Bundle params = new Bundle();  
 ARouter.getInstance()  
 .build("/home/main")  
 .with(params)  
 .navigation();  
  
 // 指定Flag  
 ARouter.getInstance()  
 .build("/home/main")  
 .withFlags();  
 .navigation();  
  
 // 获取Fragment  
 Fragment fragment = (Fragment) ARouter.getInstance().build("/test/fragment").navigation();  
   
 // 对象传递  
 ARouter.getInstance()  
 .withObject("key", new TestObj("Jack", "Rose"))  
 .navigation();  
  
 // 觉得接口不够多，可以直接拿出Bundle赋值  
 ARouter.getInstance()  
 .build("/home/main")  
 .getExtra();  
  
 // 转场动画(常规方式)  
 ARouter.getInstance()  
 .build("/test/activity2")  
 .withTransition(R.anim.slide\_in\_bottom, R.anim.slide\_out\_bottom)  
 .navigation(this);  
  
 // 转场动画(API16+)  
 ActivityOptionsCompat compat = ActivityOptionsCompat.  
 makeScaleUpAnimation(v, v.getWidth() / 2, v.getHeight() / 2, 0, 0);  
  
 // ps. makeSceneTransitionAnimation 使用共享元素的时候，需要在navigation方法中传入当前Activity  
  
 ARouter.getInstance()  
 .build("/test/activity2")  
 .withOptionsCompat(compat)  
 .navigation();  
   
 // 使用绿色通道(跳过所有的拦截器)  
 ARouter.getInstance().build("/home/main").greenChannel().navigation();  
  
```  
  
`postcard.navigation()` 如果 `@Route` 标识的是 Activity，则跳转到 Activity。其他返回该类实例  
  
```  
  
@Route(path = "/user/list")  
public class UserListActivity extends BaseActivity {  
}  
  
ARouter.getInstance().build("/user/detail").navigation(); // 跳转 Activity  
  
// startActivityForResult  
// navigation的第一个参数是 Activity，第二个参数是 RequestCode  
ARouter.getInstance().build("/user/detail").navigation(activity, 11);  
  
@Route(path = Constants.USER.LIST\_FRAGMENT)  
public class UserListFragment extends BaseFragment {  
}  
  
Fragment fragment = (Fragment) ARouter.getInstance().build("/user/list-fragment").navigation(); // 获取 Fragmetn 实例  
  
```  
  
## 项目实战  
  
### 1. 统一路由定义  
  
```java  
public final class Constants {  
  
 /\*\*  
 \* 用户模块路径  
 \*/  
 public static final class USER {  
 /\*\*  
 \* 前缀，标识跟用户相关  
 \*/  
 public static final String PREFIX = "/user";  
 /\*\*  
 \* 登录  
 \*/  
 public static final String LOGIN = "/user/login";  
 /\*\*  
 \* 用户列表  
 \*/  
 public static final String LIST = "/user/list";  
 /\*\*  
 \* 用户详情  
 \* @param   
 \*/  
 public static final String DETAIL = "/user/detail";  
 /\*\*  
 \* 返回一个用户列表 fragment  
 \*/  
 public static final String LIST\_FRAGMENT = "/user/list-fragment";  
  
  
 }  
  
 /\*\*  
 \* web 模块路径  
 \*/  
 public static final class WEB {  
 /\*\*  
 \* 前缀，标识web相关  
 \*/  
 public static final String PREFIX = "/web";  
 /\*\*  
 \* 一般的web 页  
 \* @param url String  
 \*/  
 public static final String COMMON = "/web/common";  
 }  
}  
  
```  
  
### 2. 设置 Activity、Fragment 路由  
  
```java  
@Route(path = Constants.USER.LIST\_FRAGMENT)  
public class UserListFragment extends BaseFragment {  
  
 @Autowired(name = "select")  
 boolean mIsSelect;  
  
 @Override  
 public int getLayoutRes() {  
 return R.layout.user\_fragment\_list;  
 }  
  
 @Override  
 public void render(@Nullable Bundle savedInstanceState) {  
 ARouter.getInstance().inject(this);  
 // ...  
 }  
}  
```  
  
```java  
@Route(path = Constants.USER.LIST)  
public class UserListActivity extends BaseActivity {  
  
 @Autowired  
 boolean select;  
  
 @Override  
 public int getLayoutRes() {  
 return R.layout.fragment\_container;  
 }  
  
 @Override  
 public void render(@Nullable Bundle savedInstanceState) {  
 // 别忘了调用注入  
 ARouter.getInstance().inject(this);  
  
 // 获取 Fragment   
 Fragment fragment = (Fragment) ARouter.getInstance().build(Constants.USER.LIST\_FRAGMENT)  
 .withBoolean("select", select)  
 .navigation();  
  
 addFragment(fragment);  
 }  
  
 private void addFragment(Fragment fragment) {  
 getSupportFragmentManager().beginTransaction().replace(R.id.fragment\_container, fragment, fragment.getClass().getSimpleName()).commit();  
 }  
}  
```  
  
### 3. Activity 跳转  
  
```  
private void pickAUser() {  
// 跳转到 Account 组件的用户列表页面  
ARouter.getInstance().build(Constants.USER.LIST)  
 .withBoolean("select", true)// 传递参数  
 .navigation(this, REQUEST\_PICK\_USER); // 类似于调用 startActivityForResult  
}  
```  
  
### 4. 定义拦截器  
  
拦截器通过 aop 方式自动注册，只需要实现 `IInterceptor` 接口即可  
  
```java  
@Interceptor(priority = 8, name = "登录拦截器")  
public class LoginInterceptor implements IInterceptor {  
  
 private boolean mIsLogin;  
  
 @Override  
 public void process(Postcard postcard, InterceptorCallback callback) {  
 final String path = postcard.getPath();  
 // 可以通过 path 来过滤拦截。具体写法可以参考 demo 源码  
   
 if (mIsLogin) {  
 callback.onContinue(postcard);  
 return;  
 } else {  
 showLogin();  
 }  
 }  
  
 /\*\*  
 \* 拦截器被初始化时调用，只会调用一次  
 \* 即ARouter.init(Application) 的时候会调用  
 \* @param context 这个 context 就是 Application  
 \*/  
 @Override  
 public void init(Context context) {  
 }  
}  
```  
  
### 5. 通过依赖注入解耦（可以用于组件间通信）  
  
```java  
// 声明接口，其他组件通过接口来调用服务  
public interface IApp extends IProvider {  
  
 Application getApp();  
}  
  
// 实现接口  
@Route(path = "/service/app")  
public class AppImpl implements IApp {  
  
 private Application mApp;  
  
 @Override  
 public Application getApp() {  
 return mApp;  
 }  
  
 @Override  
 public void init(Context context) {  
 mApp = (Application) context;  
 }  
}  
```  
  
使用服务  
  
```java  
public final class ToastUtils {  
  
 private static Application mContext;  
  
 private static Application getContext() {  
 if (null == mContext) {  
 // 使用依赖查找的方式发现服务  
 IApp iApp = ARouter.getInstance().navigation(IApp.class);  
// IApp app = (IApp) ARouter.getInstance().build("/service/app").navigation(); // 这种方式也可以  
 mContext = iApp.getApp();  
 }  
  
 return mContext;  
 }  
  
 public static void toast(CharSequence text, int duration) {  
 Toast.makeText(getContext(), text, duration).show();  
 }  
}  
  
```  
  
### 6. 设置通过 url 跳转  
  
``` java  
// 新建一个Activity用于监听Schame事件,之后直接把url传递给ARouter即可  
public class SchameFilterActivity extends Activity {  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
  
 Uri uri = getIntent().getData();  
 ARouter.getInstance().build(uri).navigation();  
 finish();  
 }  
}  
```  
  
`AndroidManifest.xml`  
  
``` xml  
<activity android:name=".activity.SchameFilterActivity">  
 <!-- Schame -->  
 <intent-filter>  
 <data  
 android:host="sc.com"  
 android:scheme="router"/>  
  
 <action android:name="android.intent.action.VIEW"/>  
  
 <category android:name="android.intent.category.DEFAULT"/>  
 <category android:name="android.intent.category.BROWSABLE"/>  
 </intent-filter>  
</activity>  
```  
  
### 7. 自定义全局降级策略  
  
当路由失败的时候回触发  
  
``` java  
// 实现DegradeService接口，并加上一个Path内容任意的注解即可  
@Route(path = "/xxx/xxx")  
public class DegradeServiceImpl implements DegradeService {  
 @Override  
 public void onLost(Context context, Postcard postcard) {  
 // do something.  
 }  
  
 @Override  
 public void init(Context context) {  
   
 }  
}  
```  
  
### 8. 实现 Json 序列化服务，利用 json 序列化可以实现传递非序列化类的实例  
  
```  
@Route(path = "/service/json")  
public class JsonServiceImpl implements SerializationService {  
 @Override  
 public void init(Context context) {  
  
 }  
  
 @Override  
 public <T> T json2Object(String text, Class<T> clazz) {  
 return JSON.parseObject(text, clazz);  
 }  
  
 @Override  
 public String object2Json(Object instance) {  
 return JSON.toJSONString(instance);  
 }  
}  
```  
  
## 其他姿势  
  
### 1. 获取原始的URI  
  
``` java  
String uriStr = getIntent().getStringExtra(ARouter.RAW\_URI);  
```  
  
### 2. 重写跳转URL  
  
``` java  
// 实现PathReplaceService接口，并加上一个Path内容任意的注解即可  
@Route(path = "/xxx/xxx") // 必须标明注解  
public class PathReplaceServiceImpl implements PathReplaceService {  
 /\*\*  
 \* For normal path.  
 \*  
 \* @param path raw path  
 \*/  
 String forString(String path) {  
 return path; // 按照一定的规则处理之后返回处理后的结果  
 }  
  
 /\*\*  
 \* For uri type.  
 \*  
 \* @param uri raw uri  
 \*/  
 Uri forUri(Uri uri) {  
 return url; // 按照一定的规则处理之后返回处理后的结果  
 }  
}  
```  
  
### 3. 路由中的分组概念  
  
- SDK中针对所有的路径(/test/1 /test/2)进行分组，分组只有在分组中的某一个路径第一次被访问的时候，该分组才会被初始化  
- 可以通过 @Route 注解主动指定分组，否则使用路径中第一段字符串(/\*/)作为分组  
- 注意：一旦主动指定分组之后，应用内路由需要使用 ARouter.getInstance().build(path, group) 进行跳转，手动指定分组，否则无法找到  
``` java  
@Route(path = "/test/1", group = "app")  
```  
  
### 4. 拦截器和服务的异同  
  
- 拦截器和服务所需要实现的接口不同，但是结构类似，都存在 init(Context context) 方法，但是两者的调用时机不同  
- 拦截器因为其特殊性，会被任何一次路由所触发，拦截器会在ARouter初始化的时候异步初始化，如果第一次路由的时候拦截器还没有初始化结束，路由会等待，直到初始化完成。  
- 服务没有该限制，某一服务可能在App整个生命周期中都不会用到，所以服务只有被调用的时候才会触发初始化操作  
  
## Q&A  
  
1. 通过 URL跳转之后，在 intent 中拿不到参数如何解决？  
  
需要注意的是，如果不使用自动注入，可以不写 `ARouter.getInstance().inject(this)`，但是需要取值的字段仍然需要标上 `@Autowired` 注解。因为  
只有标上注解之后，`ARouter` 才能知道以哪一种数据类型提取 URL 中的参数并放入 Intent 中，这样才能在 Intent 中获取到对应的参数  
  
2. Fragment 如何接收数据回传？  
  
`ARouter` 并不提供数据回传的接口，这个需要我们自己实现。如 `Activity` 可以在`onActivityResult`接收。  
由于`ARouter` 没有`navigation(Fragment)`这种接口，`Fragment.onActivityResult`是不会调用的，  
我们可以在 `Activity.onActivityResult` 方法里面调用 `Fragment.onActivityResult`。  
  
```  
// 可以在 BaseActivity 添加如下代码  
@Override  
protected void onActivityResult(int requestCode, int resultCode, @Nullable Intent data) {  
 List<Fragment> fragments = getSupportFragmentManager().getFragments();  
 if (fragments != null && fragments.size() > 0) {  
 for (Fragment fragment : fragments) {  
 fragment.onActivityResult(requestCode, resultCode, data);  
 }  
 }  
}  
```  
  
也可以通过 `EventBus` 回传数据，需要注意数据类型最好能保持唯一，以免对其他页面产生影响。  
  
其他问题请到 [ARouter issues](https://github.com/alibaba/ARouter/issues) 里查找  
  
  
## 参考  
  
- [ARouter](https://github.com/alibaba/ARouter)  
- [JIMU](https://github.com/mqzhangw/JIMU)