探索Android路由框架-ARouter

https://mp.weixin.qq.com/s?__biz=MzA5MzI3NjE2MA==&mid=2650243312&idx=1&sn=5bee03192d18ebed5270492058f67288&chksm=8863719fbf14f889209328260d1650be1196a445af645c52a3ecc068193ab7d1d44c82310037&scene=38#wechat_redirect

探索Android路由框架-ARouter

原创： 骑小猪看流星 郭霖 7月2日

今日科技快讯

6月29日，据CNBC报道，亚马逊周四完成了收购在线药房PillPack的交易，不仅表明这家电子商务巨头对医药市场的兴趣日益浓厚，也突显出其更为激进的并购模式。分析人士称，这可能是出于亚马逊需要保持增长，并迎合不同受众和人口结构所致。

作者简介

大家周一好，新的一周！继续加油！

本篇来自 **骑小猪看流星** 的投稿，分享了阿里路由框架ARouter相关的知识，一起来看看！希望大家喜欢。

骑小猪看流星 的博客地址：

https://www.jianshu.com/u/0111a7da544b

前言

首先借用阿里云栖社区的一段话：我们所使用的原生路由方案一般是通过显式intent和隐式intent两种方式实现的（这里主要是指跳转Activity or Fragment）。在显式intent的情况下，因为会存在直接的类依赖的问题，导致耦合非常严重；而在隐式intent情况下，则会出现规则集中式管理，导致协作变得非常困难。一般而言配置规则都是在Manifest中的，这就导致了扩展性较差。除此之外，使用原生的路由方案会出现跳转过程无法控制的问题，因为一旦使用了StartActivity()就无法插手其中任何环节了，只能交给系统管理，这就导致了在跳转失败的情况下无法降级，而是会直接抛出运营级的异常。这时候如果考虑使用自定义的路由组件就可以解决以上问题，比如通过URL索引就可以解决类依赖的问题；通过分布式管理页面配置可以解决隐式intent中集中式管理Path的问题；自己实现整个路由过程也可以拥有良好的扩展性，还可以通过AOP的方式解决跳转过程无法控制的问题，与此同时也能够提供非常灵活的降级方式。

因此本文意在快速集成并掌握阿里Android技术团队开源的这款路由框架。这款框架可以为应用开发提供更好更丰富的跳转方案。比如支持解析标准URL进行跳转，并自动注入参数到目标页面中；支持添加多个拦截器，自定义拦截顺序（满足拦截器设置的条件才允许跳转，所以这一特性对于某些问题又提供了新的解决思路）。基本的使用介绍完毕之后，还会对框架源码进行分析解读。

封装

添加依赖

在项目的build.gradle添加：

```
javaCompileOptions {
annotationProcessorOptions {
arguments = [moduleName :project.getName() ]
} }
compile'com.alibaba:arouter-api:1.3.1'
annotationProcessor'com.alibaba:arouter-compiler:1.1.4
```

初始化

官方建议我们在Application里面进行ARouter初始化，于是我们可以在自定义的Application onCreate() 方法里面添加如下代码：

```
ARouter.init(HomeApplication.this);
```

然后别忘记了在清单文件里面配置自定义的Application和使用到的Activity。

项目依赖导入和初始化就已经完成了，下面就开始正式的功能使用以及简单的封装。

开始使用

- 首先：在Activity/Fragment类上面写上 Route path 注解。

注意：这里的路径需要注意的是至少需要有两级，/xx/xx

- 然后：在Activity/Fragment类里面进入Arouter 注入，也就是：　　ARouter.getInstance().inject(this);
- 接着：目标的Activity类上面需要声明Route path 注解，以此对应（跳转如果不对应路径，框架会Toast说路径不匹配）

上述说明的使用代码如下：

```
// 在支持路由的页面上添加注解(必选)
// 这里的路径需要注意的是至少需要有两级，/xx/xx
// 路径标签个人建议写在一个类里面 这样方便统一管理和维护
@Route(path = "/app/MainActivity")
public class MainActivity extends BaseActivity {
    public static final String TAG = "app";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
         ARouter.getInstance().inject(this);
        /**
         * Activity 跳转 (普通跳转)
         * 跳转到SimpleActivity
         */
        findViewById(R.id.skip).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
            ARouter.getInstance().build( "/app/SimpleActivity").
                        navigation();
            }
    });
```

理论上来说，如果只是进行简单的跳转页面，ARouter.getInstance().build(“目标界面对应的路径”).navigation();这一行代码即可完成跳转界面。由于项目的扩大、界面的跳转成为了很常见的功能、路径的标签数量也会相对增多。因此需要更好管理的界面路径（所以更好的选择是写一个类，在这个类里面统一管理和维护路径标签，不仅利于维护也方便后期拓展，看到路径就一目了然，哇~这个路径对应的是登录界面，这个路径对应的是详情界面而且更加容易排错）；

其次，每个页面的注入，也就是ARouter.getInstance().inject(this);这句代码出现的几率会写的很多，（而且一般的常规逻辑是有注入就有解绑或者释放资源）所以我们应该简单封装起来提高效率。

简单封装

首先是路径管理（只是参考）：

```
/**
 * ARouter上面的注解需要我们写路径标识
 * 我们可以写一个常量文件，在这里统一管理路径标签
 */
public final class Constance {
    public static final String TAG = "app";
    public static final boolean UseIInterceptor = true;
    public static final String ACTIVITY_URL_MAIN = "/app/MainActivity";
    public static final String ACTIVITY_URL_SIMPLE = "/app/SimpleActivity";
    public static final String ACTIVITY_URL_PARSE = "/app/SimpleUriActivity";
    public static final String ACTIVITY_URL_SECOND = "/app/SecondActivity";
    public static final String ACTIVITY_URL_INTERCEPTOR = "/app/Interceptor";
    public static final String ACTIVITY_URL_FRAGMENT = "/app/fragment";
    public static final String GROUP_FIRST = "group_first";
}
```

然后是注入封装；这里多提一嘴，优秀的第三方框架如果一般有注入或者绑定的API，那与之对应的一般就会有释放或者解绑资源的API。（这样做的本质是优化内存）其中， ARouter.getInstance().destroy()；这个API一目了然，就是释放资源的API。下面就是开始注入和释放资源的封装：

```
public class BaseActivity extends AppCompatActivity {
    private SparseArray<View> mViews;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // ARouter inject 注入
        ARouter.getInstance().inject(this);
     setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
        mViews = new SparseArray<>();
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
//         这样写会报错
//         ARouter.getInstance().destroy();
    }
```

笔者在Activity的基类里面通过生命周期进行了注入和解绑，但是项目运行后发现了一个问题，就是如果在onDestroy（）里面调用了 ARouter.getInstance().destroy()；在进入目标Activity之后，然后按back键返回原界面的时候，APP会报错崩溃，下面是崩溃日志：

仔细一看，初始化有问题？在前面我们说到在自定义Application里面已经初始化了ARouter，且在清单文件里面配置了自定义的Application，但是依旧提示没有初始化，这就纳了个闷？然后想了想，可能是 ARouter.getInstance().destroy()；这行代码的使用位置可能用错了。然后既然是在Application里面进行的初始化，那么就可以将这行释放资源的代码，写在Application生命周期的onTerminate( )里面，果不其然，项目运行后就没什么问题了。当然，这是我自己的思路，有更好的意见和想法请在评论区指出，谢谢。因此调整之后的Application写法如下：

```
public class HomeApplication extends Application {
//  ARouter 调试开关
    private boolean isDebugARouter = true;
    @Override
    public void onCreate() {
        super.onCreate();
        if (isDebugARouter) {
            // 下面两行必须写在init之前，否则这些配置在init过程中将无效
            ARouter.openLog();     // 打印日志
            // 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.openDebug();
        }
        // 官方建议推荐在Application中初始化
        ARouter.init(HomeApplication.this);
    }
    @Override
    public void onTerminate() {
        super.onTerminate();
        ARouter.getInstance().destroy();
    }
}
```

使用

封装完毕了路径标识以及注入释放等基本功能，我们回到ARouter的基本使用：

简单页面跳转

如果只是简单的页面跳转，上面也说了一行代码即可完成，其中，build里面是页面的标签路径，对应的就是目标Activity的这里，也就是类注释标签路径要一致。

Ps:不要忘了在清单文件里面配置Activity。

带参数的界面跳转

带参数的跳转是很常见的功能，Android可以通过Bundle去传递参数，如果使用ARouter框架，它传递参数通过以下去操

作：

ARouter传递对象的时候，首先该对象需要Parcelable或者Serializable序列化，可能Parcelable这个序列化大家觉得手写起来比较麻烦，但是Android Studio已经有一些插件帮我们自动生成Parcelable序列化了（因为Android用Parcelable序列化优势会更加明显一些）。字符串、char、int等基本数据类型当然都是可以传递的。当然，它也可以直接传Bundle、数组、列表等很多对象，传递类型如下图

携带参数的界面跳转，简单使用如下：

```
/**
 * Activity 跳转 (携带参数跳转)
 */
findViewById(R.id.skipAddParams).setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        ARouter.getInstance().
                build(Constance.ACTIVITY_URL_SECOND).
                withString("name", "android").
                withInt("age", 3).
                withParcelable("test", new ManualBean("tzw", 26)).
                navigation(MainActivity.this,123);
    }
});
```

其中，第一个参数代表的是参数的key,第二个参数对应的是我们要传递的属性值，也就是value。那么目标界面如何获取传递过来的值？这个时候，我们需要在目标界面（目标界面也要声明类注解，也就是@Route(path= “路径”)），使用Autowired注解。

```
@Route(path = Constance.ACTIVITY_URL_SECOND)
public class SecondActivity extends BaseActivity {
    private TextView mTextView;
    @Autowired()
String name;
@Autowired()
    int age;
    @Autowired(name = "test")
    ManualBean manualBean;
```

这样就可以获取到传递过来的值了

值得注意的是，只有当@Autowired(name = "test")，也就是key标签一致的情况下，才可以获取到对象的值，如果不写标签名，结果会为null，

所以为了规避每一个可能会遇到的风险，建议在@Autowired里面 都写上与之对应具体的key名。

界面跳转动画

直接调用withTransition，里面传入两个动画即可（R.anim.xxx）

使用URI进行跳转

ARouter框架也可以使用URI进行匹配跳转，代码也很少，只需匹配路径一致即可完成跳转：

```
/**
 * Activity 跳转 (使用URI进行跳转)
 * 跳转到SimpleUriActivity
 */
findViewById(R.id.skip_uri).setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Uri uri = Uri.parse(Constance.ACTIVITY_URL_PARSE);
        ARouter.getInstance().build(uri).navigation();
    }
});
```

Fragment跳转

Fragment的跳转也可以参照Activity跳转，第一步依旧是先写上类注释，然后是强转，代码如下

```
Fragment fragment = (Fragment) ARouter.getInstance().build(Constance.ACTIVITY_URL_FRAGMENT).navigation();
```

进阶用法之拦截器

拦截器是ARouter这一款框架的亮点。说起拦截器这个概念，可能印象更加深刻的是OkHttp的拦截器，OkHttp的拦截器主要是用来拦截请求体（比如添加请求Cookie）和拦截响应体（判断Token是否过期），在真正的请求和响应前做一些判断

和修改然后在去进行操作，大抵这就是拦截器的简单概念。那么，ARouter框架的拦截器是怎么实现的？

ARouter的拦截器，是通过实现 IInterceptor接口，重写init（）和process（）方法去完成拦截器内部操作的。

首先我们定义两个拦截器：

首先，定义ARouter拦截器必须要使用Interceptor类注解。注解里面的 priority（也就是红色框） 这个是声明拦截器的优先级、里面的属性值是int类型。既然是定义优先级，我们这里定义2个拦截器来测试看看优先级是如何区分谁先谁后的？两个拦截器写完之后，运行下项目看下效果：

结论 1：根据实验得知，使用Interceptor类注解的priority数值越小，越先执行，优先级越高。（四大组件中的广播，优先级的取值是-1000到1000，数值越大优先级越高）。那么，还有一种情况，如果两个拦截器定义的优先级都是一样的，那么谁的优先级会高？是根据类的字符串长度来判断嘛还是别的条件来判断的？？

首先，将上面的拦截器的优先级改成一样（都改成1），项目编译试试，结果发现项目就会直接报错！

看下具体的错误原因：

翻译过来就是他们使用了相同的优先级。

结论 2：如果两个拦截器的优先级一样，项目编译就会报错。所以，不同拦截器定义的优先级属性值不能相同。

我们到这两个拦截器里面加一点筛选条件的代码：

```
if(postcard.getPath().equals(Constance.ACTIVITY_URL_INTERCEPTOR)) {
    Log.i(Constance.TAG, UseIInterceptor.class.getName() + " 进行了拦截处理！");
}
```

将这段代码加进去之后，重新运行App，打印日志结果如下：

为了方便看清运行的日志，我用三种颜色的箭头去对应。首先是两个拦截器的初始化，然后，调用了NavigationCallback这个回调函数里面的onFound（），然后执行了拦截器里面的process（）方法；当拦截器的process（）方法执行完毕以后，最终回调了NavigationCallback里面的onArrival（）方法。拦截器的工作流程大抵就是这样。那么，NavigationCallback这个又是什么？实际上，NavigationCallback这个简单理解就是ARouter在路由跳转的过程中，我们可以监听路由的一个具体过程。它一共有四个方法：

```
ARouter.getInstance().build(Constance.ACTIVITY_URL_INTERCEPTOR  ).
                        navigation(MainActivity.this,
                                new NavigationCallback()
                                {
                    @Override
                    public void onFound(Postcard postcard) {
                        //路由目标被发现时调用
                        String group = postcard.getGroup(); String path = postcard.getPath();
                        Log.i(TAG, "onFound : group == "+group + "; path == "+path);
                    }
                     @Override
                    public void onArrival(Postcard postcard) {
                        //路由到达之后调用
                         String group = postcard.getGroup();String path = postcard.getPath();
                         Log.i(TAG, "onArrival : group == "+group + "; path == "+path);
                    }
                    @Override
                    public void onLost(Postcard postcard) {
                        //路由被丢失时调用
                        Log.i(TAG, "onLost : " );
                    }
                    @Override
                    public void onInterrupt(Postcard postcard) {
                        //路由被拦截时调用
                        Log.i(TAG, "onInterrupt : " );
                    }
                });
```

那么，这个回调里面的 Postcard 又是什么意思？点进去源码看看，类注释写的一目了然：

红色框翻译过来的类注释就是：一个包含路线图的容器。既然是路线图的容器，那肯定有些API会获取到相应的信息。

通过Postcard可以获取到路径的组以及全路径，那么，路径的组（Group）又是什么？是这样，一般来说，ARouter在编译期框架扫描了所有的注册页面 / 字段 / 拦截器等，那么很明显运行期不可能一股脑全部加载进来，这样就太不和谐了。所以就使用分组来管理，我们的类标签里面的注释，对于group默认是“”（空字符串）如下图：

在 Group简单使用 这张图上面，根据日志，打印了分组的信息，可以发现Group的值默认就是第一个 / /（两个分隔符）

之间的内容。

那么，我们也可以自定义分组，来进行界面跳转，所以ARouter又提供了一种解决方案：

自定义分组 实现跳转界面

如果使用自定义分组来跳转界面，只需要在源代码改动以下三个位置：

- 类注解新增 group，赋值我们自定义的组名，（依旧统一写在一个类里面这样便于管理）
- 在build方法里面（这是一个方法重载），添加我们的与之对应的组名
- 在被跳转的Activity里面的类注释，加上同样的组名

通过上面三个步骤即可完成 自定义分组 来完成界面跳转

通过日志显示，这里的组名已经被我们更改成自定义分组且成功完成了跳转。

ARouter如何实现类似startActivityForResult（）？

这种应用场景也是很常见的，那ARouter该如何实现？

第一步：为了方便看效果，我们在第一个Activity设置requestCode 为123，

```
@Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        switch (requestCode) {
            case 123:
                Log.i(TAG, "onActivityResult: "+"接受到了第二个界面返回来的数据");
            default:
                break;
        }
    }
```

第二步：需要在跳转的navigation方法（这是一个方法重载）里面的第二个参数，设置我们定义的requestCode，（通过匹配requestCode 来实现该功能）

第三步：在第二个界面的setResult方法里面，写上对应的resultCode(也就是 setResult(123);)，这里就不展示Intent数据了。综合上面三个步骤，项目编译运行，跳转到第二个界面然后返回上一个界面，日志成功打印：

总结

写到这里，ARouter框架的使用就结束了。项目地址如下所示：

https://github.com/zuowutan/ShareARouter