转载请注明链接：https://blog.csdn.net/feather_wch/article/details/87376462

> 详细分析ARouter的源码。
>
> ARouter原理是比较简单的，就两方面内容: 初始化init()和路由navigation()。而这两者的本质就是围绕注册、加载所进行的。
>
> 1. 初始化：本质是为了注册Group、拦截器和Provider。注册就是将ARouter生成的中间类的className放置到索引列表中。便于以后通过反射进行实例化。
> 2. 路由：就是加载出目标后，根据对面是Activity、Fragment还是Provider进行相应的处理。过程中涉及到加载、路径动态替换、局部监听路由操作、全局降级策略、数据自动注入。
>   1. 加载：从上面注册的索引列表中获取到ClassName，判断类型。对于Provider和拦截器，就是直接构造出对象，执行其init()进行初始化，并加入到对应列表中。对于Group，实例化后调用loadInto()，将该组下面的Activity、Fragment、Provider进行注册。

ARouter源码详解

版本：2019/2/23-14:50

1、ARouter源码的四个组成部分

> 1. arouter-register
> 2. arouter-annotation
> 3. arouter-compiler
> 4. arouter-api

2、arouter-register的作用?

> 1. 1.3.0版本新添加的gradle插件.
> 2. 用于路由表的自动注册
> 3. 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。

3、arouter-annotation的作用?

> 1. 注解类以及携带数据的bean

4、 arouter-compiler的作用?

> 1. 注解处理类
> 2. 通过java的 `APT` (Annotation Processor Tool)按照定义的Processor生成所需要的类。
> 3. 生成的类位于文件夹 `build/generated/source/apt/debug/com.alibaba.android.arouter/routes` 中。

5、arouter-api的作用?

> 1. 提供了实现路由功能所需要的api

arouter-api

源码工程目录(1题)

1、com.alibaba:arouter-api-1.4.0工程目录

> 1. 大致描述arouter-api源工程中各个部分的作用。适合所有源码大致都看过后再看。

```
/**========================
 * 1、ARouter
 *========================*/
-launcher
 -ARouter.java // 提供ARouter的各项功能
 -_ARouter.java // ARouter.java内部类，实际实现ARouter功能，隐藏不想对外开放的方法。

/**========================
 * 2、核心
 *========================*/
-core
 -LogisticsCenter.java // 物流中心。填充路由需要的必要信息。
 -Warehouse.java // 仓库。存放Group、Providers、Interceptors。
 -AutowiredServiceImpl.java // 数据自动注入的管理器。
 -InterceptorServiceImpl.java // 拦截器的管理器。

/**========================
 * 3、定义所有接口。
 *========================*/
-facade
 -Postcard.java // 唯一class。明信片，路由到目标需要借助该Postcard中存储的必要信息。

 // 3.1、模板
 -template
  -IInterceptor.java // 拦截器必须实现该类。
  -IInterceptorGroup.java // 用于注册拦截器，初始化时，实现该接口的类会被调用loadInto来注册同一Group下的所有拦截器。
  -IProvider.java // 服务必须实现该类。
  -IProviderGroup.java // 用于注册Providers，初始化时，实现该接口的类会被调用loadInto来注册同一Group下的所有Providers。
  -IRouteGroup.java // 用于加载Route节点。路由时，通过该接口的实现类，来加载同一Group下的所有Route。
  -IRouteRoot.java // 根元素。初始化时，用于注册所有Group类-实现IRouteGroup接口的类。
  -ISyringe.java // 注射器接口。所有使用@Autowired注解的页面，都会生成对应的注射器实现类，用于实现自动注入。
  -ILogger.java // 日志。

 // 3.2、接口
 -callback
  -InterceptorCallback.java // 拦截器的处理接口。onContinue()和onInterrupt()
  -NavigationCallback.java // 局部监控路由过程的接口。
  -NavCallback.java // 该类用于方便使用NavigationCallback接口的功能。

 // 3.3、服务
 -service
  -AutowiredService.java // 自动注入服务接口。
  -DegradeService.java // 全局降级策略服务接口。
  -InterceptorService.java // 拦截器管理服务接口。
  -PathReplaceService.java // 路径动态替换服务接口。
  -SerializationService.java // 注入Bundle不支持的数据的序列化/反序列化接口。
  -ClassLoaderService.java // 用于InstantRun和 move Dex文件

/**========================
 * 4、ARouter系统级中间类
 *========================*/
-routes
 // 1、arouter根元素,初始化阶段,注册ARouter$$Group$$arouter到Warehouse.groupsIndex
 -ARouter$$Root$$arouterapi.java
 // 2、arouter拦截器,初始化阶段,注册AutowiredServiceImpl和InterceptorServiceImpl到Warehouse.providersIndex
 -ARouter$$Providers$$arouterapi.java
 // 3、实际加载AutowiredServiceImpl和InterceptorServiceImpl(第一次使用时)
 -ARouter$$Group$$arouter.java

/**========================
 * 5、线程相关
 *========================*/
-thread
 -DefaultPoolExecutor.java    // 默认线程池。
 -DefaultThreadFactory.java   // 线程池工厂。
 -CancelableCountDownLatch.java // 可取消的CountDownLatch。用于拦截器管理类InterceptorServiceImpl处理all拦截器。

/**========================
```

```
 * 6、工具类
 *========================*/
-utils
 -ClassUtils.java // 扫描包下面所有"com.alibaba.android.arouter.routes"开头的类名，也就是ARouter生成的中间类。
 -Consts.java     // ARouter相关常量。如: "ARouter"、"com.alibaba.android.arouter.routes"、"Root"、"Interceptors"、"Providers"
 -DefaultLogger.java // 日志工具
 -MapUtils.java
 -PackageUtils.java
 -TextUtils.java

/**========================
 * 7、ARouter相关的自定义异常
 *========================*/
-exception
 -HandlerException.java // 主流程的处理出现异常
 -InitException.java // 初始化异常
 -NoRouteFoundException.java // 没有找到目标Route

/**========================
 * 8、base
 *========================*/
-base
 -UniqueKeyTreeMap.java // key唯一的TreeMap,用于存储拦截器的Warehouse.interceptorsIndex, 保证key(优先级)唯一。
```

开启日志(4题)

1、ARouter的初始化

```
// 1.必须在init之前调用
if (isDebugARouter) {
    ARouter.openLog();     // 开启Log
    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)
}
// 2.初始化
ARouter.init(BaseApp.this);
```

流程图

2、日志初始化流程图

openLog()

3、ARouter.openLog()源码分析

> 1. 设置DefaultLogger的标志位(isShowLog)为true
> 2. 后续打印的日志，在isShowLog=true时，调用 Log的相关方法 打印出日志。

```
/**==================
 * 1、打开Log
 * // ARouter.java
 *==================*/
public static synchronized void openLog() {
    _ARouter.openLog();
}
/**==================================
 * 2、交给默认日志工具:DefaultLogger
 *     1. 设置logger的标志位(isShowLog)为true
 *     2. 打印出日志
 * // _ARouter.java
 *==================================*/
static ILogger logger = new DefaultLogger(Consts.TAG); // 日志工具
static synchronized void openLog() {
    logger.showLog(true); // 1. 设置logger的标志位(isShowLog)为true
    logger.info(Consts.TAG, "ARouter openLog"); // 2. 打印出日志
}
// DefaultLogger.java
public void showLog(boolean showLog) {
    isShowLog = showLog;
}

/**==================================
 * 3、日志的打印内部都是通过Log来实现。例如info就是Log.i
 * // DefaultLogger.java
 *==================================*/
public void info(String tag, String message) {
    if (isShowLog) {
        StackTraceElement stackTraceElement = Thread.currentThread().getStackTrace()[3];
        // Log打印出日志
        Log.i(TextUtils.isEmpty(tag) ? getDefaultTag() : tag, message + getExtInfo(stackTraceElement));
    }
}
```

4、 `_ARouter` 的作用?

> 1. ARouter的相关操作，内部都是通过 `_ARouter` 实现。
> 2. Arouter是对外暴露api的类， `_ARouter` 是真正的实现类
> 3. 好处: 解耦，可以有选择的去暴露想要给用户使用的方法，并且将其他方法隐藏在内部。比使用 `private` 的灵活性更强。

openDebug()

4、ARouter.openDebug()源码分析

```
/**==================
 * // ARouter.java
 * 1、打开Debug模式
 *==================*/
public static synchronized void openDebug() {
    _ARouter.openDebug();
}
/**==================================
 * // _ARouter.java
 * 2、设置debuggable = true
 *   1. debuggable的作用:
 *      在navigation等路由操作遇到问题的时候,如果debuggable=true,
 *      就会弹出相关的toast。如"There's no route matched!"。
 *==================================*/
static synchronized void openDebug() {
    debuggable = true;
    logger.info(Consts.TAG, "ARouter openDebug");
}
```

初始化(10题)

流程图

1、ARouter.init()流程图

init()

2、ARouter.init()源码分析

> 1. 内部通过 `_ARouter.init()` 转交给 `LogisticsCenter` (后勤中心)进行初始化工作。
> 2. `LogisticsCenter.init()` 需要得到ARouter框架生成的所有中间类的类名集合。如果有本地缓存直接读取，没有缓存会找到app的所有Dex路径，然后遍历出其中的属于com.alibaba.android.arouter.routes包下的所有类名，将这些ARouter框架生成的中间类打包成集合返回。
> 3. 获取到ARouter生成的所有中间类类名集合后，会遍历该集合并且对其中的Root、Interceptors拦截器、Providors服务进行初始化，并且加入到Map中。
> 4. `LogisticsCenter.init()` 还会实现检测是否已经通过插件完成了注册和初始化，如果是，则跳过上面这些操作。
> 5. 初始化完成后，会调用 `_ARouter.afterInit()` 触发所有拦截器的初始化，会遍历拦截器的Map，实例化所有拦截器并且调用其init方法，然后将拦截器对象加入到烂机器列表中。

```
/**==================
 * // ARouter.java
 * 1、初始化
 *==================*/
public static void init(Application application) {
    // 1. hasInit保证初始化代码只执行一次
    if (!hasInit) {
        // 2. 使用_ARouter的日至工具
        logger = _ARouter.logger;
        _ARouter.logger.info(Consts.TAG, "ARouter init start.");
        // 3. 通过_ARouter进行初始化
        hasInit = _ARouter.init(application);

        if (hasInit) {
            // 4. 触发拦截器的初始化(内部通过interceptorService实现)。实例化所有拦截器，调用其init方法，并将对象存储到列表Ware
            _ARouter.afterInit();
        }

        _ARouter.logger.info(Consts.TAG, "ARouter init over.");
    }
}
/**======================
 * // _ARouter.java
 * 2、_ARouter实际进行初始化工作
 *======================*/
private volatile static ThreadPoolExecutor executor = DefaultPoolExecutor.getInstance();
protected static synchronized boolean init(Application application) {
    // 1. 保存application为成员变量
    mContext = application;
    // 2. 重要的初始化工作
    LogisticsCenter.init(mContext, executor);
    // 3. 打印日志
    logger.info(Consts.TAG, "ARouter init success!");
    // 4. hasInit = true;
    hasInit = true;
    // 5. 创建主线程的Handler
    mHandler = new Handler(Looper.getMainLooper());
    return true;
}

/**================================
 * // LogisticsCenter.java
 * 3、LogisticsCenter初始化，在内存中加载所有的元信息。请求初始化.
 *================================*/
public synchronized static void init(Context context, ThreadPoolExecutor tpe) throws HandlerException {
    mContext = context;
    // 1. 该线程池是_ARouter中获取的DefaultPoolExecutor对象
    executor = tpe;

    try {
        /**-------------------------------------
         * 2. 由插件加载(该方法默认是空方法)
         *   1)arouter-auto-register插件会在该方法内生成代码
         *   2)调用该方法去注册所有的Routers、Interceptors、Providers
         *-------------------------------------*/
        loadRouterMap();
        if (registerByPlugin) {
            // 3. 如果由插件进行了注册，完成初始化
            logger.info(TAG, "Load router map by arouter-auto-register plugin.");
        } else {
            // 4. 插件没有成功进行注册。该集合存放生成类的类名集合.
            Set<String> routerMap;

            /**================================================
             * 5. 如果是debug模式或者App是新版本，从apt生成的包中加载类
             *================================================*/
            if (ARouter.debuggable()
                    || PackageUtils.isNewVersion(context)) {  // 对比App的新旧versioncode和versionname是否相同
```

```
                // 6. 找到app的dex，然后遍历出其中的属于com.alibaba.android.arouter.routes包下的所有类名，打包成集合返回。
                //    public static final String ROUTE_ROOT_PAKCAGE = "com.alibaba.android.arouter.routes";
                routerMap = ClassUtils.getFileNameByPackageName(mContext, ROUTE_ROOT_PAKCAGE);
                // 7. 通过sp将类名集合存储到本地
                if (!routerMap.isEmpty()) {
                    context.getSharedPreferences(AROUTER_SP_CACHE_KEY, Context.MODE_PRIVATE).edit().putStringSet(AROUTER_
                }
                // 8. 更新版本(保存versionname和versioncode)
                PackageUtils.updateVersion(context);
            } else {
                /**=============================================================
                 * 9. 不是debug模式，也不是新版本的app，直接从本地缓存中读取RouterMap(apt生成类的类名集合)
                 *=============================================================*/
                routerMap = new HashSet<>(context.getSharedPreferences(AROUTER_SP_CACHE_KEY, Context.MODE_PRIVATE).getStr
            }

            // 10. 成功找到Router Map打印RouterMap的尺寸
            logger.info(TAG, "Find router map finished, map size = " + routerMap.size());

            /**=============================================================
             * 11. 不是debug模式，也不是新版本的app，直接从本地缓存中读取RouterMap(apt生成类的类名集合)
             *=============================================================*/
            for (String className : routerMap) {
                // 12. 加载Root元素。className如:com.alibaba.android.arouter.routes.ARouter$$Root + $$app。反射并实例化对象
                if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAME + SEPARATOR + SUFFIX_ROOT)) {
                    // 将分组生成的类，例如: ARouter$$Group$$app.class和ARouter$$Group$$fragment.class添加到Map中
                    ((IRouteRoot) (Class.forName(className).getConstructor().newInstance())).loadInto(Warehouse.groupsInd
                }
                // 13. 加载拦截器interceptors。className如: com.alibaba.android.arouter.routes.ARouter$$Interceptors$$app
                else if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAME + SEPARATOR + SUFFIX_INTERCEPTORS)) {
                    ((IInterceptorGroup) (Class.forName(className).getConstructor().newInstance())).loadInto(Warehouse.ir
                }
                // 14. 加载Providers，将元数据(RouteMeta)存储到Map中。(ARouter提供服务管理，用于将一部分功能和组件封装成接口，
                else if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAME + SEPARATOR + SUFFIX_PROVIDERS)) {
                    ((IProviderGroup) (Class.forName(className).getConstructor().newInstance())).loadInto(Warehouse.provi
                }
            }
        }

        logger.info(TAG, "Load root element finished, cost " + (System.currentTimeMillis() - startInit) + " ms.");

        // 不包含任何Group，需要检查是否配置有问题.
        if (Warehouse.groupsIndex.size() == 0) {
            logger.error(TAG, "No mapping files were found, check your configuration please!");
        }

        if (ARouter.debuggable()) {
            logger.debug(TAG, String.format(Locale.getDefault(), "LogisticsCenter has already been loaded, GroupIndex[%d]
        }
    } catch (Exception e) {
        throw new HandlerException(TAG + "ARouter init logistics center exception! [" + e.getMessage() + "]");
    }
}


/**================================
 * // ClassUtils.java
 * 4、获取到所有的dex路径
 *================================*/
public static List<String> getSourcePaths(Context context) throws PackageManager.NameNotFoundException, IOException {
    ApplicationInfo applicationInfo = context.getPackageManager().getApplicationInfo(context.getPackageName(), 0);
    // 1. 获取到APK所在文件
    File sourceApk = new File(applicationInfo.sourceDir);

    List<String> sourcePaths = new ArrayList<>();
    // 2. 添加默认的apk路径到Dex路径集合中。例如:/data/app/com.feather.imageview-1/base.apk
    sourcePaths.add(applicationInfo.sourceDir);

    // 3. APK文件名 + EXTRACTED_NAME_EX(= .class)。例如: base.apk.classes
    String extractedFilePrefix = sourceApk.getName() + EXTRACTED_NAME_EXT;
```

```
    /**=======================================================
     * 4. 如果VM不支持MultiDex,进一步处理。
     *    目前手上的Phone和Pad都已经支持MultiDex，不会进入该分支，不仔细研究其中的细节。
     *    // 如果VM已经支持了MultiDex，就不要去Secondary Folder加载 Classesx.zip了，那里已经么有了
     *    // 通过是否存在sp中的multidex.version是不准确的，因为从低版本升级上来的用户，是包含这个sp配置的
     *=======================================================*/
    if (!isVMMultidexCapable()) {
        //the total dex numbers
        int totalDexNumber = getMultiDexPreferences(context).getInt(KEY_DEX_NUMBER, 1);
        File dexDir = new File(applicationInfo.dataDir, SECONDARY_FOLDER_NAME);

        for (int secondaryNumber = 2; secondaryNumber <= totalDexNumber; secondaryNumber++) {
            //for each dex file, ie: test.classes2.zip, test.classes3.zip...
            String fileName = extractedFilePrefix + secondaryNumber + EXTRACTED_SUFFIX;
            File extractedFile = new File(dexDir, fileName);
            if (extractedFile.isFile()) {
                sourcePaths.add(extractedFile.getAbsolutePath());
                //we ignore the verify zip part
            } else {
                throw new IOException("Missing extracted secondary dex file '" + extractedFile.getPath() + "'");
            }
        }
    }

    // 5. ARouter处于Debug模式,搜寻IntentRun的Dex文件
    if (ARouter.debuggable()) { // Search instant run support only debuggable
        sourcePaths.addAll(tryLoadInstantRunDexFile(applicationInfo));
    }
    // 6. 最终返回Dex路径集合。本例中只返回了路径: /data/app/com.feather.imageview-1/base.apk
    return sourcePaths;
}

/**================================
 * // ARouter$$Root$$app.java
 * 5、加载Root根元素。将group-app、group-fragment, 都添加到Map中(Warehouse.groupsIndex)
 *================================*/
public class ARouter$$Root$$app implements IRouteRoot {
    @Override
    public void loadInto(Map<String, Class<? extends IRouteGroup>> routes) {
        // 1. 分组名app和fragment都是自定义的
        routes.put("app", ARouter$$Group$$app.class);
        routes.put("fragment", ARouter$$Group$$fragment.class);
    }
}

/**=======================================
 * // ARouter$$Interceptors$$app.java
 * 6、将拦截器加载到Map中(Warehouse.interceptorsIndex)
 *     1. key=priority优先级，例如: 8
 *     2. value = 拦截器的class对象
 *=======================================*/
public class ARouter$$Interceptors$$app implements IInterceptorGroup {
    @Override
    public void loadInto(Map<Integer, Class<? extends IInterceptor>> interceptors) {
        interceptors.put(8, MyInterceptor.class);
    }
}

/**=======================================
 * // ARouter$$Providers$$arouterapi.java
 * 7、将Provider服务加载到privoders Map中(Warehouse.providersIndex)
 *=======================================*/
public class ARouter$$Providers$$arouterapi implements IProviderGroup {
    public ARouter$$Providers$$arouterapi() {
    }

    public void loadInto(Map<String, RouteMeta> providers) {
        providers.put("com.alibaba.android.arouter.facade.service.AutowiredService", RouteMeta.build(RouteType.PROVIDER,
        providers.put("com.alibaba.android.arouter.facade.service.InterceptorService", RouteMeta.build(RouteType.PROVIDEF
```

```
    }
}

/**=======================
 * // _ARouter.java
 * 8、触发拦截器的初始化。实例化所有拦截器，调用其init方法，并将对象存储到列表Warehouse.interceptors中.
 *=======================*/
static void afterInit() {
    // Trigger interceptor init, use byName.
    interceptorService = (InterceptorService) ARouter.getInstance().build("/arouter/service/interceptor").navigation();
}

/**============================================================
 * // InterceptorServiceImpl.java
 * 9、管理拦截器的服务。InterceptorService实现了IProvider接口。
 *============================================================*/
@Route(path = "/arouter/service/interceptor")
public class InterceptorServiceImpl implements InterceptorService {
    private static boolean interceptorHasInit;
    private static final Object interceptorInitLock = new Object();

    /**============================================================
     * 1. 初始化所有拦截器，调用其init方法，并且实例化拦截器对象并且存储到拦截器列表中(List<IInterceptor> interceptors)
     *============================================================*/
    public void init(final Context context) {
        LogisticsCenter.executor.execute(new Runnable() {
            public void run() {
                if (MapUtils.isNotEmpty(Warehouse.interceptorsIndex)) {
                    // 2. 遍历Map: Warehouse.interceptorsIndex
                    for (Map.Entry<Integer, Class<? extends IInterceptor>> entry : Warehouse.interceptorsIndex.entrySet()
                        Class<? extends IInterceptor> interceptorClass = entry.getValue();
                        try {
                            // 3. 构建拦截器对象，调用其init方法
                            IInterceptor iInterceptor = interceptorClass.getConstructor().newInstance();
                            iInterceptor.init(context);
                            // 4. 将拦截器添加到List中
                            Warehouse.interceptors.add(iInterceptor);
                        } catch (Exception ex) {
                            throw new HandlerException(TAG + "ARouter init interceptor error! name = [" + interceptorClas
                        }
                    }

                    interceptorHasInit = true;

                    synchronized (interceptorInitLock) {
                        interceptorInitLock.notifyAll();
                    }
                }
            }
        });
    }
    public void doInterceptions(final Postcard postcard, final InterceptorCallback callback) {...}
    private static void _excute(final int index, final CancelableCountDownLatch counter, final Postcard postcard) {...}
    private static void checkInterceptorsInitStatus() {...}
}
```

3、为什么不同module使用了相同的group名导致出现错误 `There is no route match the path` ？

> 1. 不同的Module都会生成不同的 `IRouteGroup`实现(如: `ARouter$$Group$$fragment.class`)
> 2. 在加载 `Root`元素 的时候，会先后执行两次 `put(xxx)` 方法，但是因为key相同，因此前一个会被覆盖，导致前一个定义的路由无法找到。
> 3. 官方建议不同Module的group名不能相同。

```
public class ARouter$$Root$$app implements IRouteRoot {
    @Override
    public void loadInto(Map<String, Class<? extends IRouteGroup>> routes) {
        routes.put("fragment", ARouter$$Group$$fragmentA.class);
    }
}
public class ARouter$$Root$$home implements IRouteRoot {
    @Override
    public void loadInto(Map<String, Class<? extends IRouteGroup>> routes) {
        routes.put("fragment", ARouter$$Group$$fragmentB.class);
    }
}
```

Warehouse

4、Warehouse的源码和作用分析

> 1. 存储了Providers、Interceptors以及Group相关的RouteMeta(路由元数据)

```
class Warehouse {
    // 1、存储Group的类对象和Group行管的RouteMeta
    static Map<String, Class<? extends IRouteGroup>> groupsIndex = new HashMap<>();
    static Map<String, RouteMeta> routes = new HashMap<>();

    // 2、存储Provider(服务)
    static Map<String, RouteMeta> providersIndex = new HashMap<>();
    static Map<Class, IProvider> providers = new HashMap<>();

    // 3、存储拦截器，使用UniqueKeyTreeMap在key值相同时报错(key = 优先级)。
    static Map<Integer, Class<? extends IInterceptor>> interceptorsIndex = new UniqueKeyTreeMap<>("More than one intercep
    static List<IInterceptor> interceptors = new ArrayList<>();

    static void clear() {
        routes.clear();
        groupsIndex.clear();
        providers.clear();
        providersIndex.clear();
        interceptors.clear();
        interceptorsIndex.clear();
    }
}
```

接口

IRouteRoot、IInterceptorGroup、IProviderGroup

5、IRouteRoot、IInterceptorGroup、IProviderGroup接口的作用?

> 1. init()初始化工作时，会扫描所有dex中和ARouter相关的中间类。
> 2. 只有实现这三种接口的类，才会通过反射实例化，并调用其 `loadInto` 来加载 `Root`元素、拦截器、`Provider`服务

IRouteGroup、IProvider、IInterceptor

6、IRouteGroup、IProvider、IInterceptor接口的作用?

```
/**=============================================================
 * 1、路由分组(Group)需要实现IRouteGroup接口。
 *  用于将ActvityA、ActivityB、FragmentA、FragmentB的RouteMeta(路由元数据)
 *   以路径path为key值，存储到Map atlas中
 *=============================================================*/
public interface IRouteGroup {
    void loadInto(Map<String, RouteMeta> atlas);
}
/**=============================================================
 * 2、服务接口需要实现IProvider接口。
 *  init方法用于处理初始化工作。
 *=============================================================*/
public interface IProvider {
    void init(Context context);
}
/**=============================================================
 * 3、拦截器需要实现IInterceptor接口。该接口实现了IProvider接口。
 *  主要进行两个操作.
 *  1. init方法用于处理初始化工作。
 *  2. process方法进行拦截和处理工作。
 *=============================================================*/
public interface IInterceptor extends IProvider {
    void process(Postcard postcard, InterceptorCallback callback);
}
```

RouteMeta、RouteType

7、RouteMeta和RouteType的作用?

> 1. RouteMeta是一个数据bean，封装了被注解类的一些信息
> 2. RouteType是路由类型，该枚举表明是Provider、Activity、Fragment等类型。
> 3. 详情见: `arouter-annotation->model->RouteMeta` 和 `arouter-annotation->enums->RouteType`

ClassUtils

getFileNameByPackageName()

8、ClassUtils.getFileNameByPackageName()源码分析

> 1. 找到app的dex，然后遍历出其中的属于com.alibaba.android.arouter.routes包下的所有类名.
> 2. 这些类都是编译期间生成的中间类。()

```
public static Set<String> getFileNameByPackageName(Context context, final String packageName) throws PackageManager.NameN
    final Set<String> classNames = new HashSet<>();

    // 1. 获取到所有dex的路径。
    List<String> paths = getSourcePaths(context);
    final CountDownLatch parserCtl = new CountDownLatch(paths.size());
    // 2. 遍历所有dex路径
    for (final String path : paths) {
        DefaultPoolExecutor.getInstance().execute(new Runnable() {
            @Override
            public void run() {
                DexFile dexfile = null;
                try {
                    // 3. 加载Dex文件
                    if (path.endsWith(EXTRACTED_SUFFIX)) {
                        //NOT use new DexFile(path), because it will throw "permission error in /data/dalvik-cache"
                        dexfile = DexFile.loadDex(path, path + ".tmp", 0);
                    } else {
                        dexfile = new DexFile(path);
                    }

                    // 4. 找到Dex文件中所有以"com.alibaba.android.arouter.routes"开头的类，并将其类名存储到集合中
                    Enumeration<String> dexEntries = dexfile.entries();
                    while (dexEntries.hasMoreElements()) {
                        String className = dexEntries.nextElement();
                        if (className.startsWith(packageName)) {
                            classNames.add(className);
                        }
                    }
                } catch (Throwable ignore) {
                    Log.e("ARouter", "Scan map file in dex files made error.", ignore);
                } finally {
                    if (null != dexfile) {
                        try {
                            dexfile.close();
                        } catch (Throwable ignore) {
                        }
                    }
                    parserCtl.countDown();
                }
            }
        });
    }
    parserCtl.await();
    // 5. 返回"com.alibaba.android.arouter.routes"包下的所有类名的集合
    return classNames;
}
```

> 返回的 `classNames` 例如:

```
0 = "com.alibaba.android.arouter.routes.ARouter$$Root$$app"
1 = "com.alibaba.android.arouter.routes.ARouter$$Group$$app$1"
2 = "com.alibaba.android.arouter.routes.ARouter$$Group$$fragment"
3 = "com.alibaba.android.arouter.routes.ARouter$$Providers$$arouterapi"
4 = "com.alibaba.android.arouter.routes.ARouter$$Group$$app"
6 = "com.alibaba.android.arouter.routes.ARouter$$Group$$arouter"
5 = "com.alibaba.android.arouter.routes.ARouter$$Interceptors$$app"
7 = "com.alibaba.android.arouter.routes.ARouter$$Root$$arouterapi"
8 = "com.alibaba.android.arouter.routes.ARouter$$Providers$$app"
```

9、CountDownLatch的作用?

> 1. CountDownLatch这个类能够使一个线程等待其他线程完成各自的工作后再执行。
> 2. `getFileNameByPackageName` 需要等所有的Dex路径都扫描好后，才返回类名的集合。

10、ClassUtils.getFileNameByPackageName()的效率问题和改进方法

1. 遍历所有Dex路径寻找指定包名下所有类的操作工作量过大，从而会导致效率问题。
2. arouter-register就是用来解决这个问题。

Navigation路由(19题)

流程图

1、ARouter.getInstance().build(xxx).navigation()流程图

build()

2、ARouter.getInstance().build()源码分析

1. 本质就是构建出内部保存了 `path` 和 `group` 的 `Postcard`
2. 会先通过官方预留的 `PathReplaceService` 对 `path` 进行 动态处理
3. 需要注意在 `PathReplaceService` 处理path前进行判断处理，避免 `build(path, group)`和`build(path)` 对路径进行了重复的两次动态处理。
4. Provider、Activity、Fragment区别:

区别	Activity	Fragment	Provider
PathReplaceService(路径动态变化)	√	√	√
NavigationCallback(局部监控)	√	√	√
DegradeService(全局降级服务)	√	√	√
绿色通道	√	×	×
InterceptorService(拦截器)	√	×	×
数据传递	intent.putExtras	setArguments	无
路由结果	跳转到Activity	获取Fragment	获取Provider

```
/**==============================================
 * 1、构建加载的路径。例如: path = /app/HostActivity
 * //ARouter.java
 *==============================================*/
public Postcard build(String path) {
    // 交给_ARouter构建出path相关的Postcard
    return _ARouter.getInstance().build(path);
}
/**==============================================
 * 2、_ARouter通过路径path和默认分组group构建出postcard。
 *   例如: path = /app/HostActivity
 * //_ARouter.java
 *==============================================*/
protected Postcard build(String path) {
    // ...确保path不为空
    // 1、Navigation出PathReplaceService对象。PathReplaceService继承自IProvider接口，是预留给用户实现路径动态变化的功能。
    PathReplaceService pService = ARouter.getInstance().navigation(PathReplaceService.class);
    if (null != pService) {
        // 2、通过该方法中由用户自定义实现的路径path动态变化的逻辑进行处理后，获取到最终的path路径。
        path = pService.forString(path);
    }
    /**==========================================================
     * 3、进一步构建。extractGroup(path)是提取出该path路径中的默认group分组。
     *    1. 该build(path, group)和build(path)中都会通过PathReplaceService对path进行处理。
     *    2. 在PathReplaceService的路径动态变化前需要进行判断工作，避免两次变化.
     *     path = /app/HostActivity
     *     group = app
     *==========================================================*/
    return build(path, extractGroup(path));
}
/**==============================================
 * 3、通过路径path和分组group构建出postcard。
 *
 * //_ARouter.java
 *==============================================*/
protected Postcard build(String path, String group) {
    // ...确保path不为空
    // 1、PathReplaceService的动态路径变换。
    PathReplaceService pService = ARouter.getInstance().navigation(PathReplaceService.class);
    if (null != pService) {
        path = pService.forString(path);
    }
    // 2、使用该path和group构造出Postcard(明信片)对象
    return new Postcard(path, group);
}

/**==============================================
 * 4、Postcard继承自RouterMeta
 *  构建明信片Postcard就是内部保存path(/app/HostActivity)和group(app)
 * //Postcard.java
 *==============================================*/
public final class Postcard extends RouteMeta {
    // xxx
    // 属性均省略
    public Postcard(String path, String group) {
        this(path, group, null, null);
    }

    public Postcard(String path, String group, Uri uri, Bundle bundle) {
        setPath(path);
        setGroup(group);
        // xxx
    }
}
```

navigation()

路由分析(Activity、Fragment、Provider)

3、ARouter.getInstance().build(path).navigation()源码分析

> 1. 也就是对 `Postcard` 的 `navigation` 进行源码分析

```
/**=============================================
 * 1、进行路由。路由到postcard中path所指定的路径中。
 *   1. 没有参数，默认使用application的context
 * // Postcard.java
 *=============================================*/
public Object navigation() {
    return navigation(null);
}
/**=============================================
 * 2、使用参数指定的Context进行路由。
 * // Postcard.java
 *=============================================*/
public Object navigation(Context context) {
    return navigation(context, null);
}
/**=============================================
 * 3、使用参数指定的Context进行路由。并且使用NavigationCallback对路由操作进行监听回调.
 * // Postcard.java
 *=============================================*/
public Object navigation(Context context, NavigationCallback callback) {
    return ARouter.getInstance().navigation(context, //Context
            this, // 该Postcard
            -1,  // requestCode
            callback); // 路由操作监听的回调接口
}
/**=============================================
 * 4、启动路由。
 * // ARouter.java
 *=============================================*/
public Object navigation(Context mContext, Postcard postcard, int requestCode, NavigationCallback callback) {
    return _ARouter.getInstance().navigation(mContext, postcard, requestCode, callback);
}

/**=============================================
 * 5、_ARouter实际进行路由操作。
 * // _ARouter.java
 *=============================================*/
protected Object navigation(final Context context, final Postcard postcard, final int requestCode, final NavigationCallba
    try {
        // 1、补全Postcard(该postcard的数据并不完整，目前只有一个path和group)
        LogisticsCenter.completion(postcard);
    } catch (NoRouteFoundException ex) {
        // 2、没有找到符合该path和group的Route
        if (debuggable()) { // Show friendly tips for user.
            Toast.makeText(mContext, "There's no route matched!\n" +
                    " Path = [" + postcard.getPath() + "]\n" +
                    " Group = [" + postcard.getGroup() + "]", Toast.LENGTH_LONG).show();
        }
        // 3、存在NavigationCallback，回调其onLost()方法表明路径没有找到.
        if (null != callback) {
            callback.onLost(postcard);
        } else {
            // 4、没有NavigationCallback，交给全局降级服务进行处理.
            DegradeService degradeService = ARouter.getInstance().navigation(DegradeService.class);
            if (null != degradeService) {
                // 5、调用降级服务的onLost()进行处理
                degradeService.onLost(context, postcard);
            }
        }
        return null;
    }
    // 6、表明已经找到该Route
    if (null != callback) {
        callback.onFound(postcard);
    }

    /**=======================================
     * 7、没有开启绿色通道时，进行拦截器的拦截处理。
     *    1. 此处的拦截器是InterceptorServiceImpl，内部的doInterceptions()方法采用了线程池进行处理
```

```
     *    2. 如果拦截器的处理不在异步线程中处理，拦截器的耗时操作可能会导致ANR
     *==========================================*/
    if (!postcard.isGreenChannel()) {
        // 8、interceptorService在初始化时的_ARouter.afterInit()中设置，实现类为InterceptorServiceImpl。
        interceptorService.doInterceptions(postcard, new InterceptorCallback() {
            // 9、onContinue()后继续进行路由
            @Override
            public void onContinue(Postcard postcard) {
                _navigation(context, postcard, requestCode, callback);
            }
            // 10、中断路由操作。回调NavigationCallback.onInterrupt()方法表明路由被中断。
            @Override
            public void onInterrupt(Throwable exception) {
                if (null != callback) {
                    callback.onInterrupt(postcard);
                }
            }
        });
    } else {
        // 11、存在绿色通道直接进行路由。
        return _navigation(context, postcard, requestCode, callback);
    }

    return null;
}


/**==============================================
 * 6、最终的路由操作，根据Postcard的类型进行处理.
 * // _ARouter.java
 *==============================================*/
private Object _navigation(final Context context, final Postcard postcard, final int requestCode, final NavigationCallbac
    final Context currentContext = null == context ? mContext : context;

    switch (postcard.getType()) {
        // 1、Activity的路由
        case ACTIVITY:
            // 2、创建Intent。Destination = HostActivity.class
            final Intent intent = new Intent(currentContext, postcard.getDestination());
            // 3、intent存入Extras
            intent.putExtras(postcard.getExtras());
            // 4、设置Flag, Activity的启动模式
            int flags = postcard.getFlags();
            if (-1 != flags) {
                intent.setFlags(flags);
            } else if (!(currentContext instanceof Activity)) {
                // 5、当前的Context不是Activity的Context，因此采用FLAG_ACTIVITY_NEW_TASK进行启动.
                intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            }

            // 6、设置Action。常用于隐式意图，可以传递参数。如: intent.setAction("android.intent.action.VIEW");
            String action = postcard.getAction();
            if (!TextUtils.isEmpty(action)) {
                intent.setAction(action);
            }

            // 7、在主线程中打开该Activity
            if (Looper.getMainLooper().getThread() != Thread.currentThread()) {
                mHandler.post(new Runnable() {
                    @Override
                    public void run() {
                        startActivity(requestCode, currentContext, intent, postcard, callback);
                    }
                });
            } else {
                startActivity(requestCode, currentContext, intent, postcard, callback);
            }

            break;
        case PROVIDER:
```

```
            return postcard.getProvider();
        case BOARDCAST:
        case CONTENT_PROVIDER:
        case FRAGMENT:
            // 8、Fragment目标的Class对象
            Class fragmentMeta = postcard.getDestination();
            try {
                // 9、创建Fragment对象，并且传入参数.
                Object instance = fragmentMeta.getConstructor().newInstance();
                if (instance instanceof Fragment) {
                    ((Fragment) instance).setArguments(postcard.getExtras());
                } else if (instance instanceof android.support.v4.app.Fragment) {
                    ((android.support.v4.app.Fragment) instance).setArguments(postcard.getExtras());
                }
                // 10、直接返回Fragment对象.
                return instance;
            } catch (Exception ex) {
                logger.error(Consts.TAG, "Fetch fragment instance error, " + TextUtils.formatStackTrace(ex.getStackTrace(
            }
        case METHOD:
        case SERVICE:
        default:
            return null;
    }

    return null;
}

/**=============================================
 * 7、startActivity，两种启动方式。
 *    1. 需要返回值：startActivityForResult
 *    2. 无返回值：startActivity
 * // _ARouter.java
 *=============================================*/
private void startActivity(int requestCode, Context currentContext, Intent intent, Postcard postcard, NavigationCallback
    // 1、根据情况选择启动activity的方式
    if (requestCode >= 0) {  // Need start for result
        ActivityCompat.startActivityForResult((Activity) currentContext, intent, requestCode, postcard.getOptionsBundle()
    } else {
        ActivityCompat.startActivity(currentContext, intent, postcard.getOptionsBundle());
    }
    // 2、处理Activity进出的动画效果
    if ((-1 != postcard.getEnterAnim() && -1 != postcard.getExitAnim()) && currentContext instanceof Activity) {    // Ol
        ((Activity) currentContext).overridePendingTransition(postcard.getEnterAnim(), postcard.getExitAnim());
    }
    // 3、回调NavigationCallback的onArrival表明，已经跳转到目标页面
    if (null != callback) { // Navigation over.
        callback.onArrival(postcard);
    }
}
```

Provider

4、加载Provider的两种方法

> 1. ARouter.getInstance().build(服务的`path`).navigation()
> 2. ARouter.getInstance().navigation(`Provider.class`)

5、ARouter.getInstance().navigation(Provider.class): 加载服务的源码流程

> 1. 该加载方式没有Build的流程。

```
    /**================================
     * 1、转交_ARouter加载Provider
     * // ARouter.java
     * ================================*/
    public <T> T navigation(Class<? extends T> service) {
        return _ARouter.getInstance().navigation(service);
    }
    /**================================
     * 2、加载Provider。
     *    1. 没有注册过，返回null
     *    2. 注册过，进行加载，返回Provider
     * // _ARouter.java
     * ================================*/
    protected <T> T navigation(Class<? extends T> service) {
        // 1、构造Provider相关的RouteMeta，存入group和path，
        Postcard postcard = LogisticsCenter.buildProvider(service.getName());
        // 2、老版本适配
        if (null == postcard) {
            postcard = LogisticsCenter.buildProvider(service.getSimpleName());
        }
        // 3、没有找到，表明没有注册过，直接返回。Provider是init初始化阶段一定会注册的。
        if (null == postcard) {
            return null;
        }
        // 4、对于Provider，直接加载。实例化，并且调用init方法。
        LogisticsCenter.completion(postcard);
        // 5、返回Provider
        return (T) postcard.getProvider();
    }
```

6、 为什么初始化时已经对所有Provider进行注册，还会生成该Provider相关的Group中间类?

> 0-自定义拦截器:

```
@Route(path = "/degrade/Service")
public class DegradeServiceImpl implements DegradeService {}
```

> 1-Provider中间类: `ARouter$$Providers$$app.java`

```
public class ARouter$$Providers$$app implements IProviderGroup {

  // Warehouse.providersIndex
  public void loadInto(Map<String, RouteMeta> providers) {
    providers.put("com.alibaba.android.arouter.facade.service.DegradeService", RouteMeta.build(RouteType.PROVIDER, Degrac
  }
}
```

> 2-自定义Provider相关的Group中间类: `ARouter$$Group$$degrade.java`

```
public class ARouter$$Group$$degrade implements IRouteGroup {

  // Warehouse.routes
  public void loadInto(Map<String, RouteMeta> atlas) {
    atlas.put("/degrade/Service", RouteMeta.build(RouteType.PROVIDER, DegradeServiceImpl.class, "/degrade/service", "degr
  }
}
```

> 3-init初始化时，通过 `ARouter$$Providers$$app` 加入到providers的索引列表中。
> 4-通过 `"/degrade/Service"` 路由到目标Provider时，用不到这个索引信息，会直接通过 `ARouter$$Group$$degrade` 将 自定义`Provider` 注册到 `Warehouse.routes`列表中 。然后构造出Provider并加入到 `Warehouse.providers` 中。

> 5-结论：`ARouter$$Providers$$app` 是用于系统需要使用 全局降级服务 时来寻找到 `DegradeService` 的。
> 而 `ARouter$$Group$$degrade` 是用户级需要Provider时所用到的。

监听路由操作/全局降级服务

7、监听路由操作的功能是如何实现的?

> 1. `navigation()` 时传入的 `NavigationCallback接口` 相关的回调方法会在 `_ARouter.navigation()` 中进行处理。

```
protected Object navigation(final Context context, final Postcard postcard, final int requestCode, final NavigationCallba
     try {
         LogisticsCenter.completion(postcard);
     } catch (NoRouteFoundException ex) {
         // 1、路径没有找到.
        callback.onLost(postcard);
         return null;
     }
     // 2、表明已经找到该Route
     callback.onFound(postcard);

     if (!postcard.isGreenChannel()) {
         interceptorService.doInterceptions(postcard, new InterceptorCallback() {
             @Override
             public void onInterrupt(Throwable exception) {
                // 3、中断路由操作。
                callback.onInterrupt(postcard);
             }
         });
     } else {
         return _navigation(context, postcard, requestCode, callback);
     }
     return null;
 }
 private Object _navigation(final Context context, final Postcard postcard, final int requestCode, final NavigationCallba
    // xxx
    startActivity(requestCode, currentContext, intent, postcard, callback);
    return null;
}
private void startActivity(int requestCode, Context currentContext, Intent intent, Postcard postcard, NavigationCallback
    // 4、成功跳转到该页面。
    callback.onArrival(postcard);
}
```

8、如果DegradeService有多个实现类，系统是如何处理的?

> 1. path = "xxx"，进行字符串比较，较小的在前，较大的在后。
> 2. 因为是Map.put，因此较大是最终存储的。
> 3. 绝对不要有多个实现类!

9、局部监听路由操作和全局监听路由操作的优先级?

> 1. 如果存在局部监听路由操作的 `NavigationCallback` 时，直接处理不会再调用 `DegradeService`的`onLost()`方法 。
> 2. 如果不存在 `NavigationCallback` 才交给 `DegradeService` 处理。

10、为什么在路径找不到时DegradeService的onLost()没有被调用?

> 在 `navigation()` 时，已经设置了 `NavigationCallback` ， 因此直接交给 `NavigationCallback` 进行处理。

拦截器ANR

11、拦截器interceptor中如果有耗时操作会导致ANR吗?

> 1. 不会
> 2. `navigation` 进行路由时，内部处理 `拦截器相关操作` 是在 `线程池LogisticsCenter.executor` 中进行处理的。不会ANR。

绿色通道

12、为什么绿色通道GreenChannel不会导致路由被拦截?

> 1. `_ARouter.navigation()` 中会判断是否是 绿色通道
> 2. 非绿色通道才会通过拦截器进行处理。

13、为什么Fragment不会触发拦截器?

> 1. Fragment会设置 绿色通道
> 2. 在 `_ARouter.navigation()` 中通过 `LogisticsCenter.completion(postcard)` 对 `Postcard`进行填充时 ， 发现是 `Fragment` 会直接调用 `postcard.greenChannel();` 进行设置。

LogisticsCenter

14、LogisticsCenter的作用?

> 1. 物流中心，能补全Postcard，并且注册一级Group下所有二级元素。

completion(postcard)

15、LogisticsCenter.completion(postcard)源码分析

> 1.

```
/**==============================================================
     * // LogisticsCenter.java
     * 1、补全不完整的Postcard.
     *==============================================================================*/
    public synchronized static void completion(Postcard postcard) {

        // 2、通过path查询，路由Map中的RouteMeta-路由元数据。key = postcard.getPath() = /app/HostActivity
        RouteMeta routeMeta = Warehouse.routes.get(postcard.getPath());
        if (null == routeMeta) {
            /**=============================================
             * 不存在该路由点，可能是没有加载过，加载该group下所有的RouteMeta
             *=============================================*/
             // 1. 获取所处group所对应的中间类-如: ARouter$$Group$$arouter
            Class<? extends IRouteGroup> groupMeta = Warehouse.groupsIndex.get(postcard.getGroup());
            // 2. group不存在，报错。
            if (null == groupMeta) {
                throw new NoRouteFoundException(TAG + "There is no route match the path [" + postcard.getPath() + "], in
            } else {
                // 3. 实例化Group
                IRouteGroup iGroupInstance = groupMeta.getConstructor().newInstance();
                // 4. 加载Group下所有子节点到Routes Map中
                iGroupInstance.loadInto(Warehouse.routes);
                // 5. 从GroupIndex分组索引中移除该【一级分组】
                Warehouse.groupsIndex.remove(postcard.getGroup());
                // 6. 重新进行填充

                completion(postcard);   // Reload
            }
        } else {
            // 3、设置该Postcard所跳转到的目的地。如: class com.feather.imageview.HostActivity
            postcard.setDestination(routeMeta.getDestination());
            // 4、设置该Postcard的类型。如: ACTIVITY
            postcard.setType(routeMeta.getType());
            // 5、设置该Postcard的优先级。如: -1(默认值，最高优先级。)
            postcard.setPriority(routeMeta.getPriority());
            // 6、设置该Postcard的额外数据。
            postcard.setExtra(routeMeta.getExtra());

            // 7、rawUri暂时不知道是干啥的
            Uri rawUri = postcard.getUri();
            if (null != rawUri) {   // Try to set params into bundle.
                Map<String, String> resultMap = TextUtils.splitQueryParameters(rawUri);
                Map<String, Integer> paramsType = routeMeta.getParamsType();

                if (MapUtils.isNotEmpty(paramsType)) {
                    // Set value by its type, just for params which annotation by @Param
                    for (Map.Entry<String, Integer> params : paramsType.entrySet()) {
                        setValue(postcard,
                                params.getValue(),
                                params.getKey(),
                                resultMap.get(params.getKey()));
                    }

                    // Save params name which need auto inject.
                    postcard.getExtras().putStringArray(ARouter.AUTO_INJECT, paramsType.keySet().toArray(new String[]{}));
                }

                // Save raw uri
                postcard.withString(ARouter.RAW_URI, rawUri.toString());
            }

            switch (routeMeta.getType()) {
                // 8、如果类型是Provider,会通过反射构造Provider对象，调用其init方法，并且存入到Providers Map中
                case PROVIDER:
                    // 9、获取Provider的Class对象
                    Class<? extends IProvider> providerMeta = (Class<? extends IProvider>) routeMeta.getDestination();
                    // 10、Providers Map中查找对应的对象
                    IProvider instance = Warehouse.providers.get(providerMeta);
                    if (null == instance) { // There's no instance of this provider
```

```
                    // 11、没有查找到，构造对象.
                    IProvider provider = providerMeta.getConstructor().newInstance();
                    // 12、初始化
                    provider.init(mContext);
                    // 13、添加到Map中
                    Warehouse.providers.put(providerMeta, provider);
                    instance = provider;
                }
                // 14、设置Postcard的Provider
                postcard.setProvider(instance);
                // 15、Provider开通绿色通道。
                postcard.greenChannel();
                break;
            // 16、Fragment都开启绿色通道。
            case FRAGMENT:
                postcard.greenChannel();
            default:
                break;
        }
    }
}
```

buildProvider()

16、buildProvider()源码分析

```
// _ARouter.java - 构造Provider的RouteMeta
public static Postcard buildProvider(String serviceName) {
    // 1、索引Map中查找(ARouter初始化时会注册所有Provider)
    RouteMeta meta = Warehouse.providersIndex.get(serviceName);
    if (null == meta) {
        return null;
    } else {
        // 2、构造Postcard
        return new Postcard(meta.getPath(), meta.getGroup());
    }
}
```

Postcard

17、Postcard是什么?有什么用?

> 1. ARouter.build()方法就是构造出一个 `Postcard(明信片)` ，包含了所有路由到目标所需要的必要信息。
> 2. `Postcard` 继承自 `RouteMeta`
> 3. `build()` 中仅仅是构造出具有 `path和group` 的 `Postcard`
> 4. `navigation()->LogisticsCenter.completion()` 补全出包含路由所必要的信息的 `Postcard` (该过程中还会去尝试加载同一个group下的所有元素)

18、Postcard所包含的必要信息(除了携带的参数)?

> 总结: 1.URI 2.路由超时时间 3.绿色通道 4.intent的flags标志 5.intent的action 6.动画相关的OptionsCompat、入场动画、出场动画
> 1-携带的URI

```
// uri
private Uri uri;
// 设置uri
public Postcard setUri(Uri uri) {
    this.uri = uri;
    return this;
}
```

> 2-设置navigation路由的超时时间(单位：秒)

```
// Navigation的超时时间(300s)
private int timeout = 300;
// 设置timeout
public Postcard setTimeout(int timeout) {
    this.timeout = timeout;
    return this;
}
```

3-设置Intent的Flags标志(Intent)

```
// Intent的Flags标志(启动模式)
private int flags = -1;         // Flags of route
// 设置启动模式
public Postcard withFlags(@FlagInt int flag) {
    this.flags = flag;
    return this;
}
// 预设的Intent的Flag选项
@IntDef({
        Intent.FLAG_ACTIVITY_SINGLE_TOP,
        Intent.FLAG_ACTIVITY_NEW_TASK,
        Intent.FLAG_GRANT_WRITE_URI_PERMISSION,
        Intent.FLAG_DEBUG_LOG_RESOLUTION,
        Intent.FLAG_FROM_BACKGROUND,
        Intent.FLAG_ACTIVITY_BROUGHT_TO_FRONT,
        Intent.FLAG_ACTIVITY_CLEAR_TASK,
        Intent.FLAG_ACTIVITY_CLEAR_TOP,
        Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS,
        Intent.FLAG_ACTIVITY_FORWARD_RESULT,
        Intent.FLAG_ACTIVITY_LAUNCHED_FROM_HISTORY,
        Intent.FLAG_ACTIVITY_MULTIPLE_TASK,
        Intent.FLAG_ACTIVITY_NO_ANIMATION,
        Intent.FLAG_ACTIVITY_NO_USER_ACTION,
        Intent.FLAG_ACTIVITY_PREVIOUS_IS_TOP,
        Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED,
        Intent.FLAG_ACTIVITY_REORDER_TO_FRONT,
        Intent.FLAG_ACTIVITY_TASK_ON_HOME,
        Intent.FLAG_RECEIVER_REGISTERED_ONLY
})
@Retention(RetentionPolicy.SOURCE)
public @interface FlagInt {
}
```

4-设置Intent的Action

```
//增加设置intent的action
private String action;
public String getAction(){
    return action;
}
public Postcard withAction(String action){
    this.action=action;
    return this;
}
```

5-开启绿色通道

```
// 是否开启绿色通道
private boolean greenChannel;
// 开启绿色通道，不允许手动关闭(Fragment默认是绿色通道)
public Postcard greenChannel() {
    this.greenChannel = true;
    return this;
}
```

> 6-设置动画(入场动画、出场动画等)

```
// Animation
private Bundle optionsCompat;    // The transition animation of activity
private int enterAnim = -1;
private int exitAnim = -1;
// 设置动画1:
@RequiresApi(16)
public Postcard withOptionsCompat(ActivityOptionsCompat compat) {
    if (null != compat) {
        this.optionsCompat = compat.toBundle();
    }
    return this;
}
// 设置动画2: 设置常规动画
public Postcard withTransition(int enterAnim, int exitAnim) {
    this.enterAnim = enterAnim;
    this.exitAnim = exitAnim;
    return this;
}
```

Interceptor拦截器(4题)

流程图

1、InterceptorServiceImpl.doInterceptions()流程图

InterceptorServiceImpl

2、InterceptorServiceImpl.doInterceptions()进行拦截的源码分析

> 1. `InterceptorServiceImpl` 实现 `InterceptorService`接口
> 2. `_ARouter` 的navigation方法中如果当前的Postcard没有开启绿色通道，会调用 `interceptorService.doInterceptions` 进行拦截器处理。

```java
// 1、InterceptorService.java - 拦截服务，继承自IProvider
public interface InterceptorService extends IProvider {
    void doInterceptions(Postcard postcard, InterceptorCallback callback);
}
/**=======================================
 *  2、InterceptorServiceImpl.java - InterceptorService的具体实现，包含方法：init()、doInterceptions()
 *=======================================*/
@Route(path = "/arouter/service/interceptor")
public class InterceptorServiceImpl implements InterceptorService {
    private static boolean interceptorHasInit;
    private static final Object interceptorInitLock = new Object();
    // xxx
}
/**=======================================
 *  3、InterceptorServiceImpl.java - 拦截操作
 *=======================================*/
public void doInterceptions(final Postcard postcard, final InterceptorCallback callback) {
    // 1、存在拦截器。如：此时有个拦截器MainInterceptor.java
    if (null != Warehouse.interceptors && Warehouse.interceptors.size() > 0) {
        // 2、等待init()初始化完成后，才继续执行，否则wait。
        checkInterceptorsInitStatus();
        // 3、等待10s都没有初始化完成，调用拦截的回调，报错“Interceptors initialization takes too much time.”
        if (!interceptorHasInit) {
            callback.onInterrupt(new HandlerException("Interceptors initialization takes too much time."));
            return;
        }
        // 4、线程池中调用拦截器的方法，避免拦截操作耗时导致ANR。
        LogisticsCenter.executor.execute(new Runnable() {
            @Override
            public void run() {

                // 5、设置interceptorCounter的count = 拦截器数量。每处理一个拦截器，interceptorCounter - 1。
                CancelableCountDownLatch interceptorCounter = new CancelableCountDownLatch(Warehouse.interceptors.siz
                try {
                    /**=======================================
                     * 6、层层处理interceptors列表中index = 0、1、2...的拦截器，执行其process方法。
                     *   1. 如果回调了onContinue则继续层层处理。
                     *   2. 如果回调了onInterrupt则中断处理，设置Postcard的Tag为对应的Exception信息。
                     *=======================================*/
                    _excute(0, interceptorCounter, postcard);
                    // 7、阻塞，知道计数归0，或者等待300秒。
                    interceptorCounter.await(postcard.getTimeout(), TimeUnit.SECONDS);
                    /**=======================================
                     * 8、interceptorCounter的count > 0，表明没有处理完所有拦截器。
                     *   1. 此时一定是拦截器处理出现了超时。等待了300秒。
                     *   2. 如果是拦截一定会进入第二个分支，而不是当前第一个分支。
                     *   3. 会传递到上层去执行NavigationCallback.onInterrupt()方法
                     *=======================================*/
                    if (interceptorCounter.getCount() > 0) {
                        callback.onInterrupt(new HandlerException("The interceptor processing timed out."));
                    }
                    /**=======================================
                     * 9、出现了拦截操作，传递到上层去执行NavigationCallback.onInterrupt()方法。
                     *   1. onInterrupt()拦截操作后，会调用counter.cancel()调用，此时count = 0。
                     *   2. 一定会设置postcard的Tag，因此一定进入该分支。
                     *=======================================*/
                    else if (null != postcard.getTag()) {
                        callback.onInterrupt(new HandlerException(postcard.getTag().toString()));
                    }
                    // 10、剩下的情况就是处理好所有拦截器，继续进行路由。调用_Arouter._navigation()继续路由。
                    else {
                        callback.onContinue(postcard);
                    }
                } catch (Exception e) {
                    // 11、过程中出现异常，都直接拦截，不继续路由。
                    callback.onInterrupt(e);
                }
            }
        });
```

```
        } else {
            // 12、不存在拦截器，继续路由。
            callback.onContinue(postcard);
        }
    }

    /**========================================
     *  4、MainInterceptor.java - 用户自定义的一个Interceptor
     *======================================*/
    @Interceptor(priority = 1)
    public class MainInterceptor implements IInterceptor {
        // xxx
    }

    /**========================================
     *  5、InterceptorServiceImpl.java
     *   - 在InterceptorServiceImpl调用init()初始化后，才进行后续操作，否则进行等待wait 10秒钟。
     *======================================*/
    private static void checkInterceptorsInitStatus() {
        synchronized (interceptorInitLock) {
            while (!interceptorHasInit) {
                try {
                    interceptorInitLock.wait(10 * 1000);
                } catch (InterruptedException e) {
                    throw new HandlerException(TAG + "Interceptor init cost too much time error! reason = [" + e.getMessa
                }
            }
        }
    }

    /**========================================
     *  6、InterceptorServiceImpl.java - 遍历执行所有拦截器
     *======================================*/
    private static void _excute(final int index, final CancelableCountDownLatch counter, final Postcard postcard) {
        // 1、有下一个拦截器。
        if (index < Warehouse.interceptors.size()) {
            // 2、获取到interceptors中的拦截器(下标为index)。
            IInterceptor iInterceptor = Warehouse.interceptors.get(index);
            // 3、执行拦截器的process方法
            iInterceptor.process(postcard, new InterceptorCallback() {
                // 4、拦截器中选择执行onContinue()，继续路由或者处理下个拦截器。
                public void onContinue(Postcard postcard) {
                    // 5、计数减1。
                    counter.countDown();
                    // 6、执行后一个拦截器。依次处理index = 1、2、3、4、5.....的拦截器
                    _excute(index + 1, counter, postcard);
                }

                @Override
                public void onInterrupt(Throwable exception) {
                    // 7、拦截，将exception存入postcard的tag字段
                    postcard.setTag(null == exception ? new HandlerException("No message.") : exception.getMessage());
                    // 8、计数器归零
                    counter.cancel();
                }
            });
        }
        // 9、不存在拦截器，直接返回。
    }
```

3、拦截器的 `process()` 方法是在子线程执行还是主线程?

> 1. 子线程

4、拦截器的 `process()` 中如何操作UI?

> 1. 需要切换到Main线程才能进行Dialog等UI操作

inject数据注入(15题)

流程图

1、withXXX()参数传入的流程图
2、inject()的流程图

withXXX()参数传入

3、ARouter的参数传入方式分为两类

> 1. 第一种: 借助传统的 `Bundle` 进行数据传递。如: withString(key, "String")
> 2. 第二种: `Bundle` 无法传递的数据，借助 `SerializationService` 进行序列化，将生成的 `JSON String` 通过Bundle进行传递。如: withObject(key, Object)
> 3. 本质两种方法都是通过 `Bundle` 进行传递

4、Postcard内部的Bundle是如何传递给目标页面的呢?

> 1. 如果是 `Activity` : 通过Intent的 `intent.putExtras(bundle)` 传递

```
private Object _navigation(Context context, Postcard postcard, xxx) {
    // xxx
    switch (postcard.getType()) {
        case ACTIVITY:
            // 存到intent中
            intent.putExtras(postcard.getExtras());
    }
    // xxx
}
```

> 2. 如果是 `Frgament` : 通过 `Fragment` 的 `setArguments(bundle)` 传递。

```
private Object _navigation(Context context, Postcard postcard, xxx) {
    switch (postcard.getType()) {
        case FRAGMENT:
            Class fragmentMeta = postcard.getDestination();
            Object instance = fragmentMeta.getConstructor().newInstance();
            if (instance instanceof Fragment) {
                // setArguments()传入Bundle
                ((Fragment) instance).setArguments(postcard.getExtras());
            } else if (instance instanceof android.support.v4.app.Fragment) {
                // setArguments()传入Bundle
                ((android.support.v4.app.Fragment) instance).setArguments(postcard.getExtras());
            }
            return instance;
    }
    // xxx
}
```

Postcard

5、Postcard Bundle相关源码分析

> 1. 内部单纯的保存了一个 `Bundle`
> 2. 支持所有Bundle能传递的数据。
> 3. 借助SerializationService将任何Object都能存入到Bundle中

```
public final class Postcard extends RouteMeta {

    /**=====================================
     *  1、内部的Bundle
     *=====================================*/
    private Bundle mBundle;
    // 1. 该方法会直接覆盖原有的Bundle，而不是增加！
    public Postcard with(Bundle bundle) {
        if (null != bundle) {
            mBundle = bundle;
        }
        return this;
    }
    // 2. 获取到Bundle
    public Bundle getExtras() {
        return mBundle;
    }

    /**=====================================
     *  2、支持所有Bundle能传递的数据。
     *    1、 也支持在Bundle中以key-value形式存入Bundle
     *=====================================*/
    public Postcard withBundle(@Nullable String key, @Nullable Bundle value) {
        mBundle.putBundle(key, value);
        return this;
    }
    public Postcard withString(@Nullable String key, @Nullable String value) {
        mBundle.putString(key, value);
        return this;
    }
    public Postcard withBoolean(@Nullable String key, boolean value) {}
    public Postcard withShort(@Nullable String key, short value) {}
    public Postcard withInt(@Nullable String key, int value) {}
    public Postcard withLong(@Nullable String key, long value) {}
    public Postcard withDouble(@Nullable String key, double value) {}
    public Postcard withByte(@Nullable String key, byte value) {}
    public Postcard withChar(@Nullable String key, char value) {}
    public Postcard withFloat(@Nullable String key, float value) {}
    public Postcard withCharSequence(@Nullable String key, @Nullable CharSequence value) {}
    public Postcard withParcelable(@Nullable String key, @Nullable Parcelable value){}
    public Postcard withParcelableArray(@Nullable String key, @Nullable Parcelable[] value){}
    public Postcard withParcelableArrayList(@Nullable String key, @Nullable ArrayList<? extends Parcelable> value){}
    public Postcard withSparseParcelableArray(@Nullable String key, @Nullable SparseArray<? extends Parcelable> value) {}
    public Postcard withIntegerArrayList(@Nullable String key, @Nullable ArrayList<Integer> value) {}
    public Postcard withStringArrayList(@Nullable String key, @Nullable ArrayList<String> value) {}
    public Postcard withCharSequenceArrayList(@Nullable String key, @Nullable ArrayList<CharSequence> value) {}
    public Postcard withSerializable(@Nullable String key, @Nullable Serializable value) {}
    public Postcard withByteArray(@Nullable String key, @Nullable byte[] value) {}
    public Postcard withShortArray(@Nullable String key, @Nullable short[] value) {}
    public Postcard withCharArray(@Nullable String key, @Nullable char[] value) {}
    public Postcard withFloatArray(@Nullable String key, @Nullable float[] value) {}
    public Postcard withCharSequenceArray(@Nullable String key, @Nullable CharSequence[] value) {}

    /**=====================================
     *  3、借助SerializationService将任何Object都能存入到Bundle中
     *=====================================*/
    public Postcard withObject(@Nullable String key, @Nullable Object value) {
        serializationService = ARouter.getInstance().navigation(SerializationService.class);
        mBundle.putString(key, serializationService.object2Json(value));
        return this;
    }
}
```

inject()

6、ARouter路由时是如何传递参数的?

> 1. `@Autowired` 的属性会在 `ARouter.getInstance().inject(this);` 调用时实现自动注入。
> 2. 原生Activity、Fragment传递数据都是通过 `Bundle` 实现的。
> 3. ARouter传递数据也是基于 `Bundle` 实现，并且自动赋值。

```
// 携带数据
ARouter.getInstance()
        .build("/app/HostActivity")
        .withString("type", "/fragment/one")
        .navigation();
```

> HostActivity: 注入数据

```
@Route(path = "/app/HostActivity")
public class HostActivity extends AppCompatActivity {

    @Autowired(name = "type")
    String type;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // 注入数据
        ARouter.getInstance().inject(this);
    }
}
```

7、inject(this)源码分析

```
/**============================
 * 1、注入参数和服务
 * // ARouter.java
 *============================*/
public void inject(Object thiz) {
    _ARouter.inject(thiz);
}
/**============================
 * 2、通过AutowiredService注入参数
 *  @param thiz 路由的目标，例如:HostActivity
 * // _ARouter.java
 *============================*/
static void inject(Object thiz) {
    // 1、实例化AutowiredService
    AutowiredService autowiredService = ((AutowiredService) ARouter.getInstance().build("/arouter/service/autowired").nav
    if (null != autowiredService) {
        // 2、注入参数
        autowiredService.autowire(thiz);
    }
}

/**============================
 * 3、AutowiredService的实现类: AutowiredServiceImpl
 * // AutowiredServiceImpl.java
 *============================*/
@Route(path = "/arouter/service/autowired")
public class AutowiredServiceImpl implements AutowiredService {
    private LruCache<String, ISyringe> classCache; // Size = 66
    private List<String> blackList;
    /**============================
     * 1、参数注入
     *============================*/
    @Override
    public void autowire(Object instance) {
        // 2、获取到目标的className，如: HostActivity
        String className = instance.getClass().getName();
        try {
            // 3、不需要自动注入的目标会存到blackList中。
            if (!blackList.contains(className)) {
                // 4、缓存中去获取该页面所对应的ISyringe(注射器)实现
                ISyringe autowiredHelper = classCache.get(className);
                if (null == autowiredHelper) {
                    // 5、不存在缓存，创建
                    autowiredHelper = (ISyringe) Class.forName(instance.getClass().getName() + SUFFIX_AUTOWIRED).getConst
                }
                // 6、注射器的实现类进行注入
                autowiredHelper.inject(instance);
                // 7、缓存
                classCache.put(className, autowiredHelper);
            }
        } catch (Exception ex) {
            // 8、不需要自动注入的目标，存放到blackList中
            blackList.add(className);
        }
    }
}

/**========================================
 * 4、所有具有@Autowired注解的类，都会通过apt生成ISyringe(注射器)的实现类
 * // ISyringe.java
 *====================================*/
public interface ISyringe {
    void inject(Object target);
}

/**========================================
 * 5、HostActivity的注射器实现类。由apt生成。
 * // HostActivity$$ARouter$$Autowired.java
 *====================================*/
```

```
public class HostActivity$$ARouter$$Autowired implements ISyringe {
    private SerializationService serializationService;
    @Override
    public void inject(Object target) {
        // 1、目标类: HostActivity
        HostActivity substitute = (HostActivity)target;
        /**=====================================================
         * 2、赋值，@Autowired注解的属性。
         *  1. substitute.type 这种表示代表private的属性无法注入。
         * // 2.“目标类对象.属性”遇到private属性会抛出异常。会导致将该HostActivity存入不需要注入的列表中。
         * // 3. 对private属性用 @Autowired注解，会导致同一个类其他非private属性也无法注入.
         *  4. 使用@Autowired注解private属性，直接会导致编译不过。
         *=====================================================*/
        substitute.type = substitute.getIntent().getStringExtra("type");
        // xxx
        /**=====================================================
         * 3、如果存在Bundle无法携带的属性。
         *  1. 会通过SerializationService序列化成json的String传递。
         *  2. SerializationService没有提供默认实现，需要用户自己实现
         *=====================================================*/
        serializationService = ARouter.getInstance().navigation(SerializationService.class);
        if (null != serializationService) {
            substitute.obj = serializationService.parseObject(substitute.getArguments().getString("obj"), new com.alibaba
        } else {
            Log.e("ARouter::", "You want automatic inject the field 'obj' in class 'BlankFragment' , then you should impl
        }
        if (null == substitute.obj) {
            Log.e("ARouter::", "The field 'obj' is null, in class '" + BlankFragment.class.getName() + "!");
        }
    }
}
```

SerializationService

8、需要传递自定义Bean需要实现SerializationService接口

> 1. 将自定义Bean等数据序列化成Json字符串
> 2. 注入时再将Json转换为对应的Bean。
> 3. 自定义实现: JsonServiceImpl

```
@Route(path = "/service/json")
public class JsonServiceImpl implements SerializationService{
    Gson mGson = new Gson();
    // 1、Object转为Json
    @Override
    public String object2Json(Object instance) {
        return mGson.toJson(instance);
    }

    // 2、Json转为Object
    @Override
    public <T> T parseObject(String input, Type clazz) {
        return mGson.fromJson(input, clazz);
    }

    // 废弃。
    @Override
    public <T> T json2Object(String input, Class<T> clazz) {
        return mGson.fromJson(input, clazz);
    }

    @Override
    public void init(Context context) {
    }
}
```

9、withObject()崩溃报错

> 必须要实现如上 `SerializationService`接口的实现类 ， 如: JsonServiceImpl

10、@Autowired能否注解private属性?

> 1. 不可以。
> 2. 会直接导致编译报错。
> 3. 特殊情况下 注解 `private`属性 会导致当前目标，被加入到 不自动注入列表 中，从而导致 非`private`属性 无法 注入数据 。

11、为什么@Autowired注解的属性没有被注入数据?

> 1. 没有调用 `ARouter.getInstance().inject(this);`
> 2. 该属性为 `private`属性
> 3. 该属性虽然为 非`private`属性 ， 但是用 `@Autowired` 注解了 `private`属性 导致该页面被加入到 不自动注入列表 中
> 4. 跳转到该页面时没有携带对应参数数据。

12、@Autowired对于public、protected、default、private修饰的属性是否可以注入数据?

> 1. public: 可以
> 2. protected: 可以。虽然可能navigation的源对象位于其他包，但是注射器实现
>   类 `HostActivity$$ARouter$$Autowired` 和 `Hostctivity` 位于同一个包，因此 目标类对象.属性 可以注入数据。
> 3. default: 可以。同理 `protected`
> 4. private: 编译都失败。

13、ARouter路由传递数据的流程

> 1. 发起路由请求。通过 `LogisticsCenter.completion` 自动包装好 `Postcard`的参数(需要传递的数据)
> 2. 将 `Postcard`的数据 放入 `Intent` ,并且启动Activity。
> 3. `inject()` 方法中通过编译器产生的中间类- `xxx$$ARouter$$Autowired` 进行赋值操作。

14、ARouter处理数据的两种思路

> 1. Bundle能处理的数据，通过Bundle传输。
> 2. Bundle不能传递的数据，通过 `SerializationService` 将对象转为 `Json`字符串 进行传递，然后反序列化。

AutowiredServiceImpl

15、AutowiredServiceImpl如何缓存的？LruCache的应用场景?

> 1. 利用 `LruCache` 对所有具有@Autowired注解的类所生成ISyringe(注射器)的实现进行缓存。
> 2. 应用于 `AutowiredServiceImpl` (自动注入服务)

arouter-annotation(12题)

注解

Route

1、Route源码

```
// 1、表明该注解用于：类、接口(注解类型)、枚举
@Target({ElementType.TYPE})
// 2、进保留到编译器生成的`class`文件中
@Retention(RetentionPolicy.CLASS)
public @interface Route {

    // 1、路径
    String path();
    // 2、分组
    String group() default "";
    // 3、名字，用于生成javadoc
    String name() default "";
     // 4、额外的数据
    int extras() default Integer.MIN_VALUE;
     // 5、优先级。-1表示最高优先级，这个优先级暂时没啥用。
    int priority() default -1;
}
```

2、@Target注解是什么？有什么用?

> 1. 元注解之一，用于注解其他注解。
> 2. 表明了 `Annotation` 所修饰的对象范围。

```
@Documented
// 运行时也会保留
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    // 表明该注解类型可以用于哪些元素
    ElementType[] value();
}
```

3、@Target可以修饰的元素类型ElementType有哪些(10种)?

```
public enum ElementType {
    /** Class, interface (including annotation type), or enum declaration */
    // 1、描述类：类、接口(包括Annotation)、enum枚举声明
    TYPE,
    // 2、描述域：成员变量-包括enmu枚举常量
    FIELD,
    // 3、方法
    METHOD,
    // 4、参数
    PARAMETER,
    // 5、构造器
    CONSTRUCTOR,
    // 6、局部变量
    LOCAL_VARIABLE,
    // 7、Annotation注解类型
    ANNOTATION_TYPE,
    // 8、包
    PACKAGE,
    // 9、类型参数声明，1.8开始
    TYPE_PARAMETER,
    // 10、 Use of a type，1.8开始
    TYPE_USE
}
```

4、@Retention注解是什么？有什么用?

> 1. 定义该Annotation被保留的时间长短。
> 2. RetentionPolicy(保留策略)有三种类型:
>   1. SOURCE: 仅保留在源码中。
>   2. CLASS: 保留到class文件

> 3. RUNTIME: VM的运行时依旧保留。
> 3. 注释类型声明中不存在 Retention 注释，则默认为 `CLASS` Retention:

```
@Documented
// 1、运行时有效
@Retention(RetentionPolicy.RUNTIME)
// 2、注解
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    // 3、保留策略
    RetentionPolicy value();
}
```

> RetentionPolicy: 记忆策略

```
public enum RetentionPolicy {
    // 1、源文件中有效。被编译器丢弃。
    SOURCE,
     // 2、class文件中有效。在VM的运行时，被遗弃。这是默认的行为。
    CLASS,
     // 3、运行时有效。会在编译器的class文件中和VM的运行时中保留，该类型要深思熟虑。
    RUNTIME
}
```

5、元注解是什么？(meta-annotation)

> 1. 元注解的作用就是负责注解其他注解。
> 2. Java5.0定义了4个标准的meta-annotation类型:
> 3. `@Target`
> 4. `@Retention`
> 5. `@Documented`
> 6. `@Inherited`

Autowired

6、Autowired注解的源码分析

> 1. 该注解用于 需要自动注入的成员变量

```
// 1、修饰成员变量
@Target({ElementType.FIELD})
// 2、保留到class文件中
@Retention(RetentionPolicy.CLASS)
public @interface Autowired {
    // 3、参数或者服务的name
    String name() default "";
    /**==========================================
     * 4、required = true时，如果value = null，app会崩溃。
     *     1. 只检查引用类型。
     *     2. 不检查原始类型
     *==========================================*/
    boolean required() default false;
    // 5、描述
    String desc() default "";
}
```

7、如何检查说是否注入的某些必要的数据?

> 1. 如果是引用类型，可以使用 `@Autowired(name = "xxx", required = true)`
> 2. 该参数的值如果为 `null` ， app会直接崩溃。让开发者在调试阶段就发现问题。

Interceptor

8、Interceptor注解源码分析

```
// 1、用于描述class、interface
@Target({ElementType.TYPE})
// 2、class文件中有效
@Retention(RetentionPolicy.CLASS)
public @interface Interceptor {
    // 3、拦截器的优先级。ARouter会按顺序从0开始，依次处理拦截器(0、1、2、3、4、....)
    int priority();
    // 4、name，用于生成javadoc
    String name() default "Default";
}
```

enums

TypeKind

9、TypeKind是什么？有什么作用?

> 1-用于表明 参数类型 ， 便于ARouter调用相应的 `withBoolean()/withByte()` 去装填入参数。
> 2-共有 12种类型的参数

```
public enum TypeKind {
    // Base type
    BOOLEAN,
    BYTE,
    SHORT,
    INT,
    LONG,
    CHAR,
    FLOAT,
    DOUBLE,

    // Other type
    STRING,
    SERIALIZABLE,
    PARCELABLE,
    OBJECT;
}
```

> 3-会在 中间类ARouter$$Group$$xxx 中去设置参数类型。

```
public class ARouter$$Group$$app implements IRouteGroup {
    @Override
    public void loadInto(Map<String, RouteMeta> atlas) {
        atlas.put("/app/HostActivity",
                RouteMeta.build(RouteType.ACTIVITY, HostActivity.class, "/app/hostactivity", "app",
                        /**=====================================
                         * 填入参数类型。
                         * 这里:
                         *   11 = OBJECT，student是一个自定义Bean对象
                         *   8 = STRING，type是一个字符串
                         *=====================================*/
                        new HashMap<String, Integer>(){{put("student", 11); put("type", 8); }},
                        -1, -2147483648));
    }
}
```

10、ARouter根据参数类型，向postcard装填入参数的流程。

> 1. `LogisticsCenter.completion()` -> `LogisticsCenter.setValue()`

```
/**==================================
 * 1、Postcard的补全
 * // LogisticsCenter.java
 * ================================*/
public synchronized static void completion(Postcard postcard) {

    RouteMeta routeMeta = Warehouse.routes.get(postcard.getPath());
    if (null != routeMeta) {
        // 1、Postcard设置各种参数.
        postcard.setXXX();
        // 2、获取到URI
        Uri rawUri = postcard.getUri();
        if (null != rawUri) {
            // 3、从URL中拆分出参数的value
            Map<String, String> resultMap = TextUtils.splitQueryParameters(rawUri);
            // 4、参数类型的HashMap
            Map<String, Integer> paramsType = routeMeta.getParamsType();

            if (MapUtils.isNotEmpty(paramsType)) {
                // 5、遍历所有的参数。根据参数类型，设置参数到Postcard中。
                for (Map.Entry<String, Integer> params : paramsType.entrySet()) {
                    setValue(postcard,
                            // 6、参数类型
                            params.getValue(),
                            // 7、Key值。
                            params.getKey(),
                            // 8、该参数对应的Value
                            resultMap.get(params.getKey()));
                }

                // 9、存入需要自动注入的参数。此处是所有参数.
                postcard.getExtras().putStringArray(ARouter.AUTO_INJECT, paramsType.keySet().toArray(new String[]{}));
            }
            // 10、存入原始的URI
            postcard.withString(ARouter.RAW_URI, rawUri.toString());
        }
    }
}

/**==================================
 * 2、设置URI中解析出来的参数。
 *   根据参数类型和TypeKind对比，调用相应的方法存入参数Value.
 * // LogisticsCenter.java
 *================================*/
private static void setValue(Postcard postcard, Integer typeDef, String key, String value) {
    //
    if (null != typeDef) {
        if (typeDef == TypeKind.BOOLEAN.ordinal()) {
            postcard.withBoolean(key, Boolean.parseBoolean(value));
        } else if (typeDef == TypeKind.BYTE.ordinal()) {
            postcard.withByte(key, Byte.valueOf(value));
        } else if (typeDef == TypeKind.SHORT.ordinal()) {
            postcard.withShort(key, Short.valueOf(value));
        } else if (typeDef == TypeKind.INT.ordinal()) {
            postcard.withInt(key, Integer.valueOf(value));
        } else if (typeDef == TypeKind.LONG.ordinal()) {
            postcard.withLong(key, Long.valueOf(value));
        } else if (typeDef == TypeKind.FLOAT.ordinal()) {
            postcard.withFloat(key, Float.valueOf(value));
        } else if (typeDef == TypeKind.DOUBLE.ordinal()) {
            postcard.withDouble(key, Double.valueOf(value));
        } else if (typeDef == TypeKind.STRING.ordinal()) {
            postcard.withString(key, value);
        } else if (typeDef == TypeKind.PARCELABLE.ordinal()) {
            // TODO : How to description parcelable value with string?
        } else if (typeDef == TypeKind.OBJECT.ordinal()) {
            postcard.withString(key, value);
        } else {    // Compatible compiler sdk 1.0.3, in that version, the string type = 18
            postcard.withString(key, value);
```

```
            }
        } else {
            postcard.withString(key, value);
        }
    }
```

RouteType

11、RouteType的作用

> 1. 路由类型

```
/**=======================================
 * 1、路由类型。枚举，表示被注解类的路由类型。
 *=======================================*/
public enum RouteType {
    ACTIVITY(0, "android.app.Activity"),
    SERVICE(1, "android.app.Service"),
    PROVIDER(2, "com.alibaba.android.arouter.facade.template.IProvider"),
    CONTENT_PROVIDER(-1, "android.app.ContentProvider"),
    BOARDCAST(-1, ""),
    METHOD(-1, ""),
    FRAGMENT(-1, "android.app.Fragment"),
    UNKNOWN(-1, "Unknown route type");

    int id;
    String className;
}
```

model

RouteMeta

12、RouteMeta是什么?

> 1. RouteMeta是一个数据bean，封装了被注解类的一些信息
> 2. 所有需要跳转的页面(Activity、Fragment)都会封装成 `RouteMeta` 存放到 `Warehouse.routes` 中
> 3. 所有的Provider的索引都会封装成 `RouteMeta` 存放到 `Warehouse.providersIndex` 中，后续实际加载 `provider` 时会从索引中提取出关键信息， 实例化 provider并且存入 `Warehouse` 的 `Map<Class, IProvider> providers` 中。

```
public class RouteMeta {
    /**=======================================
     * 1、路由类型。是一个枚举，表示被注解类的路由类型。例如: PROVIDER
     *   1. Activity
     *   2. Service // 目前不支持
     *   3. Provider
     *   4. Content Provider // 目前不支持
     *   5. Broastcast // 目前不支持
     *   6. Method
     *   7. Fragment
     *   8. UNKNOWN
     *=======================================*/
    private RouteType type;
    private Element rawType;        // Raw type of route
    /**=======================================
     *  2、路由的目标。
     *   如: HostActivity.class、MainFragment.class、MyInterceptor.cass、
     *=======================================*/
    private Class<?> destination;   // Destination. 例如: class com.alibaba.android.arouter.core.AutowiredServiceImpl; cl
    /**=======================================
     *  3、路由的路径。
     *   如: /app/MainActivity
     *=======================================*/
    private String path;            // Path of route. 例如: /arouter/service/autowired; /arouter/service/interceptor
    /**=======================================
     *  4、路由的分组(一级路径)。
     *   如: app、arouter
     *=======================================*/
    private String group;           // Group of route. 例如: arouter
    /**=======================================
     *  5、属性类型。包含了所有注解了Autowired的属性的信息。
     *   key为属性名，value为属性类型，ARouter将可被intent传递的数据类型定义了对应的int类型:
     *   BOOLEAN,BYTE,SHORT,INT,LONG,CHAR,FLOAT,DOUBLE,STRING,PARCELABLE,OBJECT分别对应0，1，2，3...
     *=======================================*/
    private Map<String, Integer> paramsType;  // Param type
    /**=======================================
     *  6、优先级。-1默认为最高。
     *   1. 该属性目前没啥用。
     *   2. 拦截器才需要优先级。并且放在Warehouse的
     *      Map<Integer, Class<? extends IInterceptor>> interceptorsIndex中。
     *      也不是RouteMeta属性的。
     *=======================================*/
    private int priority = -1;      // The smaller the number, the higher the priority
    /**=======================================
     *  7、name
     *=======================================*/
    private String name;
    /**=======================================
     *  8、额外数据
     *=======================================*/
    private int extra;              // Extra data
    /**=======================================
     *  9、注解配置，目前不知道有什么用。
     *=======================================*/
    private Map<String, Autowired> injectConfig;  // Cache inject config.
}
```

arouter-compiler(10题)

1、arouter-compiler的作用和使用?

> 1. 作用是在编译期间，过滤第一个模块中定义的几个注解(Route，Autowired，Interceptor)，然后生成编译期间的中间代码.
> 2. `build.gradle` 中引入

```
    annotationProcessor 'com.alibaba:arouter-compiler:1.2.0'
```

AbstractProcessor简介

2、AbstractProcessor的作用?

> 1. 通过AbstractProcessor以 Java编译时生成代码的方式 实现 注解处理器 。
> 2. 抽象类AbstractProcessor以及接口Processor都是位于 包javax.annotation.processing 中。
> 3. 该抽象类有四个主要方法:

```
public abstract class AbstractProcessor implements Processor {
    protected ProcessingEnvironment processingEnv;
    private boolean initialized = false;

    /**=================================================================================
     * 1、初始化。用于初始化操作，利用参数提供的ProcessingEnvironment，可能以获取一些有用的工具类。
     * ================================================================================*/
    public synchronized void init(ProcessingEnvironment var1) {
        if(this.initialized) {
            throw new IllegalStateException("Cannot call init more than once.");
        } else {
            Objects.requireNonNull(var1, "Tool provided null ProcessingEnvironment");
            this.processingEnv = var1;
            this.initialized = true;
        }
    }
    /**=================================================================================
     * 2、注解处理器的核心方法，处理具体的注解。
     * ================================================================================*/
    public abstract boolean process(Set<? extends TypeElement> var1, RoundEnvironment var2);

    /**=================================================================================
     * 3、返回此注释 Processor 支持的最新的源版本。
     *      1. 可以通过注解指定: @SupportedSourceVersion(SourceVersion.RELEASE_7)
     * ================================================================================*/
    public SourceVersion getSupportedSourceVersion() {
        // xxx
    }
    /**=================================================================================
     * 4、返回此 Processor 支持的注释类型的名称。
     *      1. 可能是"name."形式的名称，表示所有以"name."开头的规范名称的注释类型集合。
     *      2. 不应该声明"*"，除非实际处理了所有文件。如此声明可能导致性能下降。
     *      3. 可以通过注解指定: @SupportedAnnotationTypes()
     * ================================================================================*/
    public Set<String> getSupportedAnnotationTypes() {
        // xxx
    }
    /**=================================================================================
     * 5、可以通过注解指定: @SupportedOptions()
     * ================================================================================*/
    public Set<String> getSupportedOptions() {
        SupportedOptions var1 = (SupportedOptions)this.getClass().getAnnotation(SupportedOptions.class);
        return var1 == null?Collections.emptySet():arrayToSet(var1.value());
    }
    public Iterable<? extends Completion> getCompletions(xxx){xxx}
    protected synchronized boolean isInitialized() {return this.initialized;}
    private static Set<String> arrayToSet(String[] var0) {xxx}
}
```

3、ProcessingEnvironment的作用

> 提供有用的工具类。

```
public interface ProcessingEnvironment {

        /**
         * 返回用来在元素上进行操作的某些实用工具方法的实现。<br>
         *
         * Elements是一个工具类，可以处理相关Element（包括ExecutableElement, PackageElement, TypeElement, TypeParameterElem
         */
        Elements getElementUtils();

        /**
         * 返回用来报告错误、警报和其他通知的 Messager。
         */
        Messager getMessager();

        /**
         *  用来创建新源、类或辅助文件的 Filer。
         */
        Filer getFiler();

        /**
         *  返回用来在类型上进行操作的某些实用工具方法的实现。
         */
        Types getTypeUtils();

        // 返回任何生成的源和类文件应该符合的源版本。
        SourceVersion getSourceVersion();

        // 返回当前语言环境；如果没有有效的语言环境，则返回 null。
        Locale getLocale();

        // 返回传递给注释处理工具的特定于 processor 的选项
        Map<String, String> getOptions();
    }
```

RouteProcessor

4、RouteProcessor的作用

> 1. 在编译期间获取Route注解的类，生成中间类文件。
> 2. 生成唯一的Root文件: `ARouter$$Root$$ + ModuleName`
> 3. 生成唯一的Provider文件: `ARouter$$Providers + ModuleName`
> 4. 生成各个分组对应的Group文件:
>   1. `ARouter$$Group$$ + GroupA`
>   2. `ARouter$$Group$$ + GroupB`
>   3. `ARouter$$Group$$ + GroupC`

5、RouteProcessor源码解析

> 1. 生成 `Root`中间类、`Group`中间类、`Provider`中间类 文件
> 2. 分别构造这三种文件的 `loadInto()`

```
@AutoService(Processor.class)
// 1、支持的选项: "AROUTER_MODULE_NAME"和"AROUTER_GENERATE_DOC"
//        javaCompileOptions {
//                annotationProcessorOptions {
//                includeCompileClasspath = true
//                arguments = [AROUTER_MODULE_NAME: project.getName()]
//               }
//        }
@SupportedOptions({KEY_MODULE_NAME, KEY_GENERATE_DOC_NAME})
// 2、支持的最新的源版本：1.7
@SupportedSourceVersion(SourceVersion.RELEASE_7)
// 3、支持的注解类型的名称: "com.alibaba.android.arouter.facade.annotation.Route"和"com.alibaba.android.arouter.facade.ann
@SupportedAnnotationTypes({ANNOTATION_TYPE_ROUTE, ANNOTATION_TYPE_AUTOWIRED})
public class RouteProcessor extends AbstractProcessor {
    private Map<String, Set<RouteMeta>> groupMap = new HashMap<>(); // ModuleName and routeMeta.
    private Map<String, String> rootMap = new TreeMap<>();  // Map of root metas, used for generate class file in order.
    // xxx省略xxx

    /**=============================================================
     * 1、初始化“注解处理器”，通过ProcessingEnvironment提供的相应工具类。
     *    1. 获取并处理用户配置的 module name
     *    2. 生成用户配置的 doc文件
     *=============================================================*/
    public synchronized void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);

        // 0、工具初始化
        mFiler = processingEnv.getFiler();                  // Generate class.
        types = processingEnv.getTypeUtils();            // Get type utils.
        elements = processingEnv.getElementUtils();      // Get class meta.
        iProvider = elements.getTypeElement(Consts.IPROVIDER).asType();
        typeUtils = new TypeUtils(types, elements);
        logger = new Logger(processingEnv.getMessager());   // Package the log utils.

        // 1、获取到用户配置的Module Name。
        Map<String, String> options = processingEnv.getOptions();
        if (MapUtils.isNotEmpty(options)) {
            moduleName = options.get(KEY_MODULE_NAME);
            generateDoc = VALUE_ENABLE.equals(options.get(KEY_GENERATE_DOC_NAME));
        }

        // 2、处理用户配置的module name
        if (StringUtils.isNotEmpty(moduleName)) {
            moduleName = moduleName.replaceAll("[^0-9a-zA-Z_]+", "");
        }

        // 3、生成doc文件
        if (generateDoc) {
            docWriter = mFiler.createResource(
                        StandardLocation.SOURCE_OUTPUT,
                        PACKAGE_OF_GENERATE_DOCS,
                        "arouter-map-of-" + moduleName + ".json"
                        ).openWriter();
        }
    }

    /**=============================================================
     * 2、处理注解。会获取到环境变量中过滤的元素集合，最终生成编译期间的中间类。
     *      命名规则：工程名+$$+Group+$$+模块名
     *=============================================================*/
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        if (CollectionUtils.isNotEmpty(annotations)) {
            Set<? extends Element> routeElements = roundEnv.getElementsAnnotatedWith(Route.class);
            // 1. 解析路由元素
            this.parseRoutes(routeElements);
            return true;
        }
        return false;
    }
```

```
/**=============================================================
 * 3、解析路由元素。
 *=============================================================*/
private void parseRoutes(Set<? extends Element> routeElements) throws IOException {
    if (CollectionUtils.isNotEmpty(routeElements)) {
        // 1、找到所有用@Route注解的目标
        logger.info(">>> Found routes, size is " + routeElements.size() + " <<<");

        rootMap.clear();

        TypeMirror type_Activity = elements.getTypeElement(ACTIVITY).asType();
        TypeMirror type_Service = elements.getTypeElement(SERVICE).asType();
        TypeMirror fragmentTm = elements.getTypeElement(FRAGMENT).asType();
        TypeMirror fragmentTmV4 = elements.getTypeElement(Consts.FRAGMENT_V4).asType();
        // Interface of ARouter
        TypeElement type_IRouteGroup = elements.getTypeElement("xxx.IRouteGroup");
        TypeElement type_IProviderGroup = elements.getTypeElement("xxx..IProviderGroup");
        ClassName routeMetaCn = ClassName.get(RouteMeta.class);
        ClassName routeTypeCn = ClassName.get(RouteType.class);

        /**=============================================================
         * 2、构造Root元素中，loadInto的入参
         *     // ARouter$$Root$$... loadInto()方法入参:
         *     Map<String, Class<? extends IRouteGroup>> routes
         *=============================================================*/
        ParameterizedTypeName inputMapTypeOfRoot = ParameterizedTypeName.get(
                ClassName.get(Map.class),
                ClassName.get(String.class),
                ParameterizedTypeName.get(
                        ClassName.get(Class.class),
                        WildcardTypeName.subtypeOf(ClassName.get(type_IRouteGroup))
                )
        );

        /**=============================================================
         * 3、构造Group和Providers中，loadInto的入参
         *     // ARouter$$Group$$... loadInto()方法入参:
         *        1. Map<String, RouteMeta> atlas
         *     // ARouter$$Providers$$... loadInto()方法入参:
         *        2. Map<String, RouteMeta> providers
         *=============================================================*/
        ParameterizedTypeName inputMapTypeOfGroup = ParameterizedTypeName.get(
                ClassName.get(Map.class),
                ClassName.get(String.class),
                ClassName.get(RouteMeta.class)
        );

        /**=============================================
         * 4、构造入参名称
         *     1. Root元素: "routes"
         *     2. Group: "atlas"
         *     3. Providers: "providers"
         *=============================================*/
        ParameterSpec rootParamSpec = ParameterSpec.builder(inputMapTypeOfRoot, "routes").build();
        ParameterSpec groupParamSpec = ParameterSpec.builder(inputMapTypeOfGroup, "atlas").build();
        ParameterSpec providerParamSpec = ParameterSpec.builder(inputMapTypeOfGroup, "providers").build();  // Ps. it


        //  Follow a sequence, find out metas of group first, generate java file, then statistics them as root.
        for (Element element : routeElements) {
            TypeMirror tm = element.asType();
            Route route = element.getAnnotation(Route.class);
            RouteMeta routeMeta;

            /**=============================================
             * 5、目标是Activity。处理所有Autowire注解的字段的参数类型。
             *=============================================*/
            if (types.isSubtype(tm, type_Activity)) {                 // Activity
```

```
            logger.info(">>> Found activity route: " + tm.toString() + " <<<");

            // 1. 获取到所有@Autowired注解的字段
            Map<String, Integer> paramsType = new HashMap<>();
            Map<String, Autowired> injectConfig = new HashMap<>();
            // 2. 遍历这些字段，处理数据类型，存入到RouteMeta的paramsType中。
            for (Element field : element.getEnclosedElements()) {
                if (field.getKind().isField() && field.getAnnotation(Autowired.class) != null && !types.isSubtype
                    // It must be field, then it has annotation, but it not be provider.
                    Autowired paramConfig = field.getAnnotation(Autowired.class);
                    String injectName = StringUtils.isEmpty(paramConfig.name()) ? field.getSimpleName().toString(
                    paramsType.put(injectName, typeUtils.typeExchange(field));
                    injectConfig.put(injectName, paramConfig);
                }
            }
            // 3. 构建RouteMeta
            routeMeta = new RouteMeta(route, element, RouteType.ACTIVITY, paramsType);
            // 4. 存入Map<String, Autowired> injectConfig
            routeMeta.setInjectConfig(injectConfig);
        }
        /**============================================
         * 6、目标是Provider，构建对应的RouteMeta
         *============================================*/
        else if (types.isSubtype(tm, iProvider)) {         // IProvider
            logger.info(">>> Found provider route: " + tm.toString() + " <<<");
            routeMeta = new RouteMeta(route, element, RouteType.PROVIDER, null);
        }
        /**============================================
         * 7、目标是Service，构建对应的RouteMeta
         *============================================*/
        else if (types.isSubtype(tm, type_Service)) {           // Service
            logger.info(">>> Found service route: " + tm.toString() + " <<<");
            routeMeta = new RouteMeta(route, element, RouteType.parse(SERVICE), null);
        }
        /**============================================
         * 8、目标是Fragment，构建对应的RouteMeta
         *============================================*/
        else if (types.isSubtype(tm, fragmentTm) || types.isSubtype(tm, fragmentTmV4)) {
            logger.info(">>> Found fragment route: " + tm.toString() + " <<<");
            routeMeta = new RouteMeta(route, element, RouteType.parse(FRAGMENT), null);
        }

        /**============================================
         * 9、对RouteMeta进行分类.
         *============================================*/
        categories(routeMeta);
    }

    /**============================================
     * 10、Root的方法loadInto()的构造器
     *============================================*/
    MethodSpec.Builder loadIntoMethodOfRootBuilder = MethodSpec.methodBuilder(METHOD_LOAD_INTO)
            .addAnnotation(Override.class)
            .addModifiers(PUBLIC)
            .addParameter(rootParamSpec);

    /**============================================
     * 11、Provider的方法loadInto()的构造器
     *============================================*/
    MethodSpec.Builder loadIntoMethodOfProviderBuilder = MethodSpec.methodBuilder(METHOD_LOAD_INTO)
            .addAnnotation(Override.class)
            .addModifiers(PUBLIC)
            .addParameter(providerParamSpec);

    Map<String, List<RouteDoc>> docSource = new HashMap<>();

    // Start generate java source, structure is divided into upper and lower levels, used for demand initializati
    /**============================================
     * 12、根据group分组，对相同group的进行统一处理。
     *   group = app
```

```
     *   group = home
     *   ...各种分组...
     *
     *   1. 构造出
     *==============================================*/
    for (Map.Entry<String, Set<RouteMeta>> entry : groupMap.entrySet()) {
        String groupName = entry.getKey();

        /**============================================
         * 13、Group的方法loadInto()的构造器
         *============================================*/
        MethodSpec.Builder loadIntoMethodOfGroupBuilder = MethodSpec.methodBuilder(METHOD_LOAD_INTO)
                .addAnnotation(Override.class)
                .addModifiers(PUBLIC)
                .addParameter(groupParamSpec);

        List<RouteDoc> routeDocList = new ArrayList<>();

        /**============================================
         * 14、遍历同一个group的所有RouteMeta构造方法体
         *============================================*/
        Set<RouteMeta> groupData = entry.getValue();
        for (RouteMeta routeMeta : groupData) {
            RouteDoc routeDoc = extractDocInfo(routeMeta);

            ClassName className = ClassName.get((TypeElement) routeMeta.getRawType());

            switch (routeMeta.getType()) {
                /**============================================
                 * 15、Provider构造loadInto()方法的构造器，中添加"添加provider的语句":
                 *   1. providers.put("xxx", RouteMeta.build(xxx, JsonServiceImpl.class, xxx));
                 *============================================*/
                case PROVIDER:  // Need cache provider's super class
                    List<? extends TypeMirror> interfaces = ((TypeElement) routeMeta.getRawType()).getInterfaces(
                    for (TypeMirror tm : interfaces) {
                        routeDoc.addPrototype(tm.toString());


                        if (types.isSameType(tm, iProvider)) {   // Its implements iProvider interface himself.
                            String className = (routeMeta.getRawType()).toString();
                        } else if (types.isSubtype(tm, iProvider)) {
                            String className = tm.toString();
                        }

                        loadIntoMethodOfProviderBuilder.addStatement(
                                    "providers.put($S, $T.build($T." + routeMeta.getType() + ", $T.class, $S, $S,
                                    className,
                                    routeMetaCn,
                                    routeTypeCn,
                                    className,
                                    routeMeta.getPath(),
                                    routeMeta.getGroup());
                    }
                    break;
                default:
                    break;
            }

            /**============================================
             * 16、构造paramsType(参数类型)的Map Body
             *============================================*/
            StringBuilder mapBodyBuilder = new StringBuilder();
            Map<String, Integer> paramsType = routeMeta.getParamsType();
            Map<String, Autowired> injectConfigs = routeMeta.getInjectConfig();
            if (MapUtils.isNotEmpty(paramsType)) {
                List<RouteDoc.Param> paramList = new ArrayList<>();

                for (Map.Entry<String, Integer> types : paramsType.entrySet()) {
                    mapBodyBuilder.append("put(\"").append(types.getKey()).append("\", ").append(types.getValue()
```

```
                RouteDoc.Param param = new RouteDoc.Param();
                Autowired injectConfig = injectConfigs.get(types.getKey());
                param.setKey(types.getKey());
                param.setType(TypeKind.values()[types.getValue()].name().toLowerCase());
                param.setDescription(injectConfig.desc());
                param.setRequired(injectConfig.required());

                paramList.add(param);
            }

            routeDoc.setParams(paramList);
        }
        String mapBody = mapBodyBuilder.toString();

        /**============================================
         * 17、Group构造loadInto()方法的构造器, 中添加语句:
         *   1. atlas.put("/app/HostActivity", RouteMeta.build(xxx, HostActivity.class, "/app/hostactivity",
         *   2. atlas.put("/app/MainActivity", RouteMeta.build(xxx, MainActivity.class, "/app/hostactivity",
         *============================================*/
        loadIntoMethodOfGroupBuilder.addStatement(
                "atlas.put($S, $T.build($T." + routeMeta.getType() + ", $T.class, $S, $S, " + (StringUtils.is
                routeMeta.getPath(),
                routeMetaCn,
                routeTypeCn,
                className,
                routeMeta.getPath().toLowerCase(),
                routeMeta.getGroup().toLowerCase());

        routeDoc.setClassName(className.toString());
        routeDocList.add(routeDoc);
    }

    /**============================================
     * 18、生成Group文件
     *   1. group = app: ARouter$$Group$$app
     *   2. group = fragment: ARouter$$Group$$fragment
     *============================================*/
    String groupFileName = NAME_OF_GROUP + groupName;
    JavaFile.builder(PACKAGE_OF_GENERATE_FILE,
            TypeSpec.classBuilder(groupFileName)
                    .addJavadoc(WARNING_TIPS)
                    .addSuperinterface(ClassName.get(type_IRouteGroup))
                    .addModifiers(PUBLIC)
                    .addMethod(loadIntoMethodOfGroupBuilder.build())
                    .build()
    ).build().writeTo(mFiler);

    logger.info(">>> Generated group: " + groupName + "<<<");
    rootMap.put(groupName, groupFileName);
    docSource.put(groupName, routeDocList);
}

// Output route doc
if (generateDoc) {
    docWriter.append(JSON.toJSONString(docSource, SerializerFeature.PrettyFormat));
    docWriter.flush();
    docWriter.close();
}

/**============================================
 * 19、生成Provider中间类文件
 *   例如: 1. ARouter$$Providers$$app
 *============================================*/
String providerMapFileName = NAME_OF_PROVIDER + SEPARATOR + moduleName;
JavaFile.builder(PACKAGE_OF_GENERATE_FILE,
        TypeSpec.classBuilder(providerMapFileName)
                .addJavadoc(WARNING_TIPS)
                .addSuperinterface(ClassName.get(type_IProviderGroup))
                .addModifiers(PUBLIC)
                .addMethod(loadIntoMethodOfProviderBuilder.build())
```

```
                    .build()
        ).build().writeTo(mFiler);

        logger.info(">>> Generated provider map, name is " + providerMapFileName + " <<<");



        /**==========================================
         * 20、Root构造器中，添加语句(将Group中间类添加到routes中)
         *   例如: 1. routes.put("app", ARouter$$Group$$app.class);
         *         2. routes.put("fragment", ARouter$$Group$$fragment.class);
         *         3. routes.put("service", ARouter$$Group$$service.class);
         *==========================================*/
        if (MapUtils.isNotEmpty(rootMap)) {
            // Generate root meta by group name, it must be generated before root, then I can find out the class of g
            for (Map.Entry<String, String> entry : rootMap.entrySet()) {
                loadIntoMethodOfRootBuilder.addStatement("routes.put($S, $T.class)", entry.getKey(), ClassName.get(PA
            }
        }
        // Write root meta into disk.
        String rootFileName = NAME_OF_ROOT + SEPARATOR + moduleName;
        JavaFile.builder(PACKAGE_OF_GENERATE_FILE,
                TypeSpec.classBuilder(rootFileName)
                        .addJavadoc(WARNING_TIPS)
                        .addSuperinterface(ClassName.get(elements.getTypeElement(ITROUTE_ROOT)))
                        .addModifiers(PUBLIC)
                        .addMethod(loadIntoMethodOfRootBuilder.build())
                        .build()
        ).build().writeTo(mFiler);

        logger.info(">>> Generated root, name is " + rootFileName + " <<<");
    }
}


/**=============================================================
 * 4、对RouteMeta进行排序。 相同group的RouteMeta放到同一个set中，并且将该set存入groupMap中
 *=============================================================================*/
private void categories(RouteMeta routeMete) {
    if (routeVerify(routeMete)) {
        logger.info(">>> Start categories, group = " + routeMete.getGroup() + ", path = " + routeMete.getPath() + " <
        Set<RouteMeta> routeMetas = groupMap.get(routeMete.getGroup());
        if (CollectionUtils.isEmpty(routeMetas)) {
            Set<RouteMeta> routeMetaSet = new TreeSet<>(new Comparator<RouteMeta>() {
                @Override
                public int compare(RouteMeta r1, RouteMeta r2) {
                    try {
                        return r1.getPath().compareTo(r2.getPath());
                    } catch (NullPointerException npe) {
                        logger.error(npe.getMessage());
                        return 0;
                    }
                }
            });
            routeMetaSet.add(routeMete);
            groupMap.put(routeMete.getGroup(), routeMetaSet);
        } else {
            routeMetas.add(routeMete);
        }
    } else {
        logger.warning(">>> Route meta verify error, group is " + routeMete.getGroup() + " <<<");
    }
}

/**================================
 * 5、验证RouteMeta的合法性
 *================================*/
private boolean routeVerify(RouteMeta meta) {
    // xxx
```

```
        return true;
    }

    /**==================================
     * 6、RouteMeta中提取出doc所需要的信息
     *==================================*/
    private RouteDoc extractDocInfo(RouteMeta routeMeta) {
        RouteDoc routeDoc = new RouteDoc();
        routeDoc.setGroup(routeMeta.getGroup());
        routeDoc.setPath(routeMeta.getPath());
        routeDoc.setDescription(routeMeta.getName());
        routeDoc.setType(routeMeta.getType().name().toLowerCase());
        routeDoc.setMark(routeMeta.getExtra());

        return routeDoc;
    }
}
```

6、RouteProcessor能处理哪些功能的中间类?

> 1. 所有 `@Route` 注解的目标。
>   1. Activity、Frgment
>   2. Provider

InterceptorProcessor

7、InterceptorProcessor的作用

> 1. 生成 拦截器 相关的中间类:

```
public class ARouter$$Interceptors$$app implements IInterceptorGroup {
  @Override
  public void loadInto(Map<Integer, Class<? extends IInterceptor>> interceptors) {
    interceptors.put(8, MyInterceptor.class);
  }
}
```

8、InterceptorProcessor的源码(未解析)

```
@AutoService(Processor.class)
@SupportedOptions(KEY_MODULE_NAME)
@SupportedSourceVersion(SourceVersion.RELEASE_7)
@SupportedAnnotationTypes(ANNOTATION_TYPE_INTECEPTOR)
public class InterceptorProcessor extends AbstractProcessor {
    private Map<Integer, Element> interceptors = new TreeMap<>();
    private Filer mFiler;       // File util, write class file into disk.
    private Logger logger;
    private Elements elementUtil;
    private String moduleName = null;   // Module name, maybe its 'app' or others
    private TypeMirror iInterceptor = null;

    // 初始化
    @Override
    public synchronized void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);

        mFiler = processingEnv.getFiler();                  // Generate class.
        elementUtil = processingEnv.getElementUtils();      // Get class meta.
        logger = new Logger(processingEnv.getMessager());   // Package the log utils.

        // Attempt to get user configuration [moduleName]
        Map<String, String> options = processingEnv.getOptions();
        if (MapUtils.isNotEmpty(options)) {
            moduleName = options.get(KEY_MODULE_NAME);
        }

        if (StringUtils.isNotEmpty(moduleName)) {
            moduleName = moduleName.replaceAll("[^0-9a-zA-Z_]+", "");
            logger.info("The user has configuration the module name, it was [" + moduleName + "]");
        } else {
            logger.error(NO_MODULE_NAME_TIPS);
            throw new RuntimeException("ARouter::Compiler >>> No module name, for more information, look at gradle log.")
        }

        iInterceptor = elementUtil.getTypeElement(Consts.IINTERCEPTOR).asType();

        logger.info(">>> InterceptorProcessor init. <<<");
    }

    /**
     * 生成中间类文件
     */
    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        if (CollectionUtils.isNotEmpty(annotations)) {
            Set<? extends Element> elements = roundEnv.getElementsAnnotatedWith(Interceptor.class);
            try {
                parseInterceptors(elements);
            } catch (Exception e) {
                logger.error(e);
            }
            return true;
        }

        return false;
    }

    /**
     * 生成拦截器中间类文件
     */
    private void parseInterceptors(Set<? extends Element> elements) throws IOException {
        if (CollectionUtils.isNotEmpty(elements)) {
            logger.info(">>> Found interceptors, size is " + elements.size() + " <<<");

            // Verify and cache, sort incidentally.
            for (Element element : elements) {
                if (verify(element)) {  // Check the interceptor meta
                    logger.info("A interceptor verify over, its " + element.asType());
```

```
                Interceptor interceptor = element.getAnnotation(Interceptor.class);

                Element lastInterceptor = interceptors.get(interceptor.priority());
                if (null != lastInterceptor) { // Added, throw exceptions
                    throw new IllegalArgumentException(
                            String.format(Locale.getDefault(), "More than one interceptors use same priority [%d], Th
                                    interceptor.priority(),
                                    lastInterceptor.getSimpleName(),
                                    element.getSimpleName())
                    );
                }

                interceptors.put(interceptor.priority(), element);
            } else {
                logger.error("A interceptor verify failed, its " + element.asType());
            }
        }

        // Interface of ARouter.
        TypeElement type_ITollgate = elementUtil.getTypeElement(IINTERCEPTOR);
        TypeElement type_ITollgateGroup = elementUtil.getTypeElement(IINTERCEPTOR_GROUP);

        /**
         *  Build input type, format as :
         *
         *  ```Map<Integer, Class<? extends ITollgate>>```
         */
        ParameterizedTypeName inputMapTypeOfTollgate = ParameterizedTypeName.get(
                ClassName.get(Map.class),
                ClassName.get(Integer.class),
                ParameterizedTypeName.get(
                        ClassName.get(Class.class),
                        WildcardTypeName.subtypeOf(ClassName.get(type_ITollgate))
                )
        );

        // Build input param name.
        ParameterSpec tollgateParamSpec = ParameterSpec.builder(inputMapTypeOfTollgate, "interceptors").build();

        // Build method : 'loadInto'
        MethodSpec.Builder loadIntoMethodOfTollgateBuilder = MethodSpec.methodBuilder(METHOD_LOAD_INTO)
                .addAnnotation(Override.class)
                .addModifiers(PUBLIC)
                .addParameter(tollgateParamSpec);

        // Generate
        if (null != interceptors && interceptors.size() > 0) {
            // Build method body
            for (Map.Entry<Integer, Element> entry : interceptors.entrySet()) {
                loadIntoMethodOfTollgateBuilder.addStatement("interceptors.put(" + entry.getKey() + ", $T.class)", Cl
            }
        }

        // Write to disk(Write file even interceptors is empty.)
        JavaFile.builder(PACKAGE_OF_GENERATE_FILE,
                TypeSpec.classBuilder(NAME_OF_INTERCEPTOR + SEPARATOR + moduleName)
                        .addModifiers(PUBLIC)
                        .addJavadoc(WARNING_TIPS)
                        .addMethod(loadIntoMethodOfTollgateBuilder.build())
                        .addSuperinterface(ClassName.get(type_ITollgateGroup))
                        .build()
        ).build().writeTo(mFiler);

        logger.info(">>> Interceptor group write over. <<<");
    }
}

/**
 * 验证拦截器数据的合法性
 */
```

```
    private boolean verify(Element element) {
        Interceptor interceptor = element.getAnnotation(Interceptor.class);
        // It must be implement the interface IInterceptor and marked with annotation Interceptor.
        return null != interceptor && ((TypeElement) element).getInterfaces().contains(iInterceptor);
    }
}
```

AutowiredProcessor

9、AutowiredProcessor的作用

> 1. 用于生成使用 `@Autowired` 注解过的页面的自动注入辅助类。例如 `HostActivity$$ARouter$$Autowired.java`

```
/**=========================================
 * 5、HostActivity的注射器实现类。由apt生成。
 * // HostActivity$$ARouter$$Autowired.java
 *=======================================*/
public class HostActivity$$ARouter$$Autowired implements ISyringe {
    private SerializationService serializationService;
    @Override
    public void inject(Object target) {
        HostActivity substitute = (HostActivity)target;
        substitute.type =
        // 注入数据
        substitute.getIntent().getStringExtra("type");
        // xxx
    }
}
```

10、AutowiredProcessor源码(未解析)

```
@AutoService(Processor.class)
@SupportedOptions(KEY_MODULE_NAME)
@SupportedSourceVersion(SourceVersion.RELEASE_7)
@SupportedAnnotationTypes({ANNOTATION_TYPE_AUTOWIRED})
public class AutowiredProcessor extends AbstractProcessor {
    private Filer mFiler;       // File util, write class file into disk.
    private Logger logger;
    private Types types;
    private TypeUtils typeUtils;
    private Elements elements;
    private Map<TypeElement, List<Element>> parentAndChild = new HashMap<>();   // Contain field need autowired and his s
    private static final ClassName ARouterClass = ClassName.get("com.alibaba.android.arouter.launcher", "ARouter");
    private static final ClassName AndroidLog = ClassName.get("android.util", "Log");

    @Override
    public synchronized void init(ProcessingEnvironment processingEnvironment) {
        super.init(processingEnvironment);

        mFiler = processingEnv.getFiler();                  // Generate class.
        types = processingEnv.getTypeUtils();            // Get type utils.
        elements = processingEnv.getElementUtils();      // Get class meta.

        typeUtils = new TypeUtils(types, elements);

        logger = new Logger(processingEnv.getMessager());   // Package the log utils.

        logger.info(">>> AutowiredProcessor init. <<<");
    }

    @Override
    public boolean process(Set<? extends TypeElement> set, RoundEnvironment roundEnvironment) {
        if (CollectionUtils.isNotEmpty(set)) {
            try {
                logger.info(">>> Found autowired field, start... <<<");
                categories(roundEnvironment.getElementsAnnotatedWith(Autowired.class));
                generateHelper();

            } catch (Exception e) {
                logger.error(e);
            }
            return true;
        }

        return false;
    }

    private void generateHelper() throws IOException, IllegalAccessException {
        TypeElement type_ISyringe = elements.getTypeElement(ISYRINGE);
        TypeElement type_JsonService = elements.getTypeElement(JSON_SERVICE);
        TypeMirror iProvider = elements.getTypeElement(Consts.IPROVIDER).asType();
        TypeMirror activityTm = elements.getTypeElement(Consts.ACTIVITY).asType();
        TypeMirror fragmentTm = elements.getTypeElement(Consts.FRAGMENT).asType();
        TypeMirror fragmentTmV4 = elements.getTypeElement(Consts.FRAGMENT_V4).asType();

        // Build input param name.
        ParameterSpec objectParamSpec = ParameterSpec.builder(TypeName.OBJECT, "target").build();

        if (MapUtils.isNotEmpty(parentAndChild)) {
            for (Map.Entry<TypeElement, List<Element>> entry : parentAndChild.entrySet()) {
                // Build method : 'inject'
                MethodSpec.Builder injectMethodBuilder = MethodSpec.methodBuilder(METHOD_INJECT)
                        .addAnnotation(Override.class)
                        .addModifiers(PUBLIC)
                        .addParameter(objectParamSpec);

                TypeElement parent = entry.getKey();
                List<Element> childs = entry.getValue();

                String qualifiedName = parent.getQualifiedName().toString();
```

```
String packageName = qualifiedName.substring(0, qualifiedName.lastIndexOf("."));
String fileName = parent.getSimpleName() + NAME_OF_AUTOWIRED;

logger.info(">>> Start process " + childs.size() + " field in " + parent.getSimpleName() + " ... <<<");

TypeSpec.Builder helper = TypeSpec.classBuilder(fileName)
        .addJavadoc(WARNING_TIPS)
        .addSuperinterface(ClassName.get(type_ISyringe))
        .addModifiers(PUBLIC);

FieldSpec jsonServiceField = FieldSpec.builder(TypeName.get(type_JsonService.asType()), "serializationSer
helper.addField(jsonServiceField);

injectMethodBuilder.addStatement("serializationService = $T.getInstance().navigation($T.class)", ARouterC
injectMethodBuilder.addStatement("$T substitute = ($T)target", ClassName.get(parent), ClassName.get(parer

// Generate method body, start inject.
for (Element element : childs) {
    Autowired fieldConfig = element.getAnnotation(Autowired.class);
    String fieldName = element.getSimpleName().toString();
    if (types.isSubtype(element.asType(), iProvider)) {  // It's provider
        if ("".equals(fieldConfig.name())) {    // User has not set service path, then use byType.

            // Getter
            injectMethodBuilder.addStatement(
                    "substitute." + fieldName + " = $T.getInstance().navigation($T.class)",
                    ARouterClass,
                    ClassName.get(element.asType())
            );
        } else {    // use byName
            // Getter
            injectMethodBuilder.addStatement(
                    "substitute." + fieldName + " = ($T)$T.getInstance().build($S).navigation()",
                    ClassName.get(element.asType()),
                    ARouterClass,
                    fieldConfig.name()
            );
        }

        // Validater
        if (fieldConfig.required()) {
            injectMethodBuilder.beginControlFlow("if (substitute." + fieldName + " == null)");
            injectMethodBuilder.addStatement(
                    "throw new RuntimeException(\"The field '" + fieldName + "' is null, in class '\" + $
            injectMethodBuilder.endControlFlow();
        }
    } else {    // It's normal intent value
        String originalValue = "substitute." + fieldName;
        String statement = "substitute." + fieldName + " = " + buildCastCode(element) + "substitute.";
        boolean isActivity = false;
        if (types.isSubtype(parent.asType(), activityTm)) {  // Activity, then use getIntent()
            isActivity = true;
            statement += "getIntent().";
        } else if (types.isSubtype(parent.asType(), fragmentTm) || types.isSubtype(parent.asType(), fragm
            statement += "getArguments().";
        } else {
            throw new IllegalAccessException("The field [" + fieldName + "] need autowired from intent, i
        }

        statement = buildStatement(originalValue, statement, typeUtils.typeExchange(element), isActivity)
        if (statement.startsWith("serializationService.")) {   // Not mortals
            injectMethodBuilder.beginControlFlow("if (null != serializationService)");
            injectMethodBuilder.addStatement(
                    "substitute." + fieldName + " = " + statement,
                    (StringUtils.isEmpty(fieldConfig.name()) ? fieldName : fieldConfig.name()),
                    ClassName.get(element.asType())
            );
            injectMethodBuilder.nextControlFlow("else");
            injectMethodBuilder.addStatement(
                    "$T.e(\"" + Consts.TAG + "\", \"You want automatic inject the field '" + fieldName +
```

```
                        injectMethodBuilder.endControlFlow();
                    } else {
                        injectMethodBuilder.addStatement(statement, StringUtils.isEmpty(fieldConfig.name()) ? fieldNa
                    }

                    // Validator
                    if (fieldConfig.required() && !element.asType().getKind().isPrimitive()) {  // Primitive wont be
                        injectMethodBuilder.beginControlFlow("if (null == substitute." + fieldName + ")");
                        injectMethodBuilder.addStatement(
                                "$T.e(\"" + Consts.TAG + "\", \"The field '" + fieldName + "' is null, in class '\" +
                        injectMethodBuilder.endControlFlow();
                    }
                }
            }

            helper.addMethod(injectMethodBuilder.build());

            // Generate autowire helper
            JavaFile.builder(packageName, helper.build()).build().writeTo(mFiler);

            logger.info(">>> " + parent.getSimpleName() + " has been processed, " + fileName + " has been generated.
        }

        logger.info(">>> Autowired processor stop. <<<");
    }
}

private String buildCastCode(Element element) {
    if (typeUtils.typeExchange(element) == TypeKind.SERIALIZABLE.ordinal()) {
        return CodeBlock.builder().add("($T) ", ClassName.get(element.asType())).build().toString();
    }
    return "";
}

private String buildStatement(String originalValue, String statement, int type, boolean isActivity) {
    if (type == TypeKind.BOOLEAN.ordinal()) {
        statement += (isActivity ? ("getBooleanExtra($S, " + originalValue + ")") : ("getBoolean($S)"));
    } else if (type == TypeKind.BYTE.ordinal()) {
        statement += (isActivity ? ("getByteExtra($S, " + originalValue + ")") : ("getByte($S)"));
    } else if (type == TypeKind.SHORT.ordinal()) {
        statement += (isActivity ? ("getShortExtra($S, " + originalValue + ")") : ("getShort($S)"));
    } else if (type == TypeKind.INT.ordinal()) {
        statement += (isActivity ? ("getIntExtra($S, " + originalValue + ")") : ("getInt($S)"));
    } else if (type == TypeKind.LONG.ordinal()) {
        statement += (isActivity ? ("getLongExtra($S, " + originalValue + ")") : ("getLong($S)"));
    }else if(type == TypeKind.CHAR.ordinal()){
        statement += (isActivity ? ("getCharExtra($S, " + originalValue + ")") : ("getChar($S)"));
    } else if (type == TypeKind.FLOAT.ordinal()) {
        statement += (isActivity ? ("getFloatExtra($S, " + originalValue + ")") : ("getFloat($S)"));
    } else if (type == TypeKind.DOUBLE.ordinal()) {
        statement += (isActivity ? ("getDoubleExtra($S, " + originalValue + ")") : ("getDouble($S)"));
    } else if (type == TypeKind.STRING.ordinal()) {
        statement += (isActivity ? ("getStringExtra($S)") : ("getString($S)"));
    } else if (type == TypeKind.SERIALIZABLE.ordinal()) {
        statement += (isActivity ? ("getSerializableExtra($S)") : ("getSerializable($S)"));
    } else if (type == TypeKind.PARCELABLE.ordinal()) {
        statement += (isActivity ? ("getParcelableExtra($S)") : ("getParcelable($S)"));
    } else if (type == TypeKind.OBJECT.ordinal()) {
        statement = "serializationService.parseObject(substitute." + (isActivity ? "getIntent()." : "getArguments().'
    }

    return statement;
}

/**
 * Categories field, find his papa.
 *
 * @param elements Field need autowired
 */
private void categories(Set<? extends Element> elements) throws IllegalAccessException {
```

```
        if (CollectionUtils.isNotEmpty(elements)) {
            for (Element element : elements) {
                TypeElement enclosingElement = (TypeElement) element.getEnclosingElement();

                if (element.getModifiers().contains(Modifier.PRIVATE)) {
                    throw new IllegalAccessException("The inject fields CAN NOT BE 'private'!!! please check field ["
                            + element.getSimpleName() + "] in class [" + enclosingElement.getQualifiedName() + "]");
                }

                if (parentAndChild.containsKey(enclosingElement)) { // Has categries
                    parentAndChild.get(enclosingElement).add(element);
                } else {
                    List<Element> childs = new ArrayList<>();
                    childs.add(element);
                    parentAndChild.put(enclosingElement, childs);
                }
            }

            logger.info("categories finished.");
        }
    }
}
```

实例分析

单Module app

1、实例分析init初始化流程()

> 1-ARouter会生成8个中间类：3个ARouter级
> (`ARouter$$Root$$arouterapi`、`ARouter$$Providers$$arouterapi`、`ARouter$$Group$$arouter`)和5个app级
> (`ARouter$$Root$$app`、`ARouter$$Interceptors$$app`、`ARouter$$Providers$$app`、`ARouter$$Group$$app`、 `ARouter$$Group$$degrade`)
> 2-加载ARouter级的Root和Providers。1、将 `ARouter$$Group$$arouter` 存入 `groupsIndex` 2、将AutowiredServiceImpl和InterceptorServiceImpl存入 `providersIndex`

```
// 0 = "com.alibaba.android.arouter.routes.ARouter$$Root$$arouterapi"
public class ARouter$$Root$$arouterapi implements IRouteRoot {
    public void loadInto(Map<String, Class<? extends IRouteGroup>> routes) {
        // Warehouse.groupsIndex【将ARouter$$Group$$arouter加入到group索引map中】
        routes.put("arouter", arouter.class);
    }
}

// 1 = "com.alibaba.android.arouter.routes.ARouter$$Providers$$arouterapi"
public class ARouter$$Providers$$arouterapi implements IProviderGroup {

    public void loadInto(Map<String, RouteMeta> providers) {
        // Warehouse.providersIndex-AutowiredServiceImpl添加到Providers Map中
        providers.put("com.alibaba.android.arouter.facade.service.AutowiredService", RouteMeta.build(RouteType.PROVIDER,
        // Warehouse.providersIndex-InterceptorServiceImpl添加到Providers Map中
        providers.put("com.alibaba.android.arouter.facade.service.InterceptorService", RouteMeta.build(RouteType.PROVIDEF
    }
}

// 2 = "com.alibaba.android.arouter.routes.ARouter$$Group$$arouter"
public class ARouter$$Group$$arouter implements IRouteGroup {

    public void loadInto(Map<String, RouteMeta> atlas) {
        // Warehouse.routes 【AutowiredServiceImpl加载到Route列表中】
        atlas.put("/arouter/service/autowired", RouteMeta.build(RouteType.PROVIDER, AutowiredServiceImpl.class, "/arouter
        // Warehouse.routes 【InterceptorServiceImpl加载到Route列表中】
        atlas.put("/arouter/service/interceptor", RouteMeta.build(RouteType.PROVIDER, InterceptorServiceImpl.class, "/arc
    }
}
```

```
Warehouse.groupsIndex: size = 1
 0 = "arouter" -> "class com.alibaba.android.arouter.routes.ARouter$$Group$$arouter"

Warehouse.providersIndex: size = 2
0 = {HashMap$Node@10926} "com.alibaba.android.arouter.facade.service.AutowiredService" -> "RouteMeta{type=PROVIDER, rawType=null, destina
1 = {HashMap$Node@10927} "com.alibaba.android.arouter.facade.service.InterceptorService" -> "RouteMeta{type=PROVIDER, rawType=null, desti
```

> 3-加载app级别的Route: 1、将 `ARouter$$Root$$app` 和 `ARouter$$Group$$degrade` 存入 `Warehouse.groupsIndex` 2、将 `DegradeServiceImpl` 存入 `Warehouse.providersIndex` 3、将 `MainInterceptor` 存入 `Warehouse.interceptorsIndex`

```
// 0 = "com.alibaba.android.arouter.routes.ARouter$$Root$$app"
public class ARouter$$Root$$app implements IRouteRoot {
  @Override
  public void loadInto(Map<String, Class<? extends IRouteGroup>> routes) {
    // Warehouse.groupsIndex【group的索引，需要进一步加载Group中的内容】 - 存入Group app, 内部加载分组app中的Route
    routes.put("app", ARouter$$Group$$app.class);
    // Warehouse.groupsIndex【group的索引，需要进一步加载Group中的内容】 - 存入Group degrade
    routes.put("degrade", ARouter$$Group$$degrade.class);
  }
}

// 1 = "com.alibaba.android.arouter.routes.ARouter$$Interceptors$$app"
public class ARouter$$Interceptors$$app implements IInterceptorGroup {
  @Override
  public void loadInto(Map<Integer, Class<? extends IInterceptor>> interceptors) {
    // Warehouse.interceptorsIndex【拦截器的索引，需要进一步加载Interceptor中的内容】
    interceptors.put(1, MainInterceptor.class);
  }
}

// 2 = "com.alibaba.android.arouter.routes.ARouter$$Providers$$app"
public class ARouter$$Providers$$app implements IProviderGroup {
  @Override
  public void loadInto(Map<String, RouteMeta> providers) {
    // Warehouse.providersIndex【Providers服务的索引，需要进一步加载Provider中的内容】
    providers.put("com.alibaba.android.arouter.facade.service.DegradeService", RouteMeta.build(RouteType.PROVIDER, Degrac
  }
}

// 3 = "com.alibaba.android.arouter.routes.ARouter$$Group$$app"
public class ARouter$$Group$$app implements IRouteGroup {
  @Override
  public void loadInto(Map<String, RouteMeta> atlas) {
    // Warehouse.routes【存放具体的RouteMeta路由元数据】 - MainActivity的路由元数据
    atlas.put("/app/MainActivity", RouteMeta.build(RouteType.ACTIVITY, MainActivity.class, "/app/mainactivity", "app", nu
    // Warehouse.routes【存放具体的RouteMeta路由元数据】 - MainFragment的路由元数据
    atlas.put("/app/MainFragment", RouteMeta.build(RouteType.FRAGMENT, MainFragment.class, "/app/mainfragment", "app", nu
  }
}

// 4 = "com.alibaba.android.arouter.routes.ARouter$$Group$$degrade"
public class ARouter$$Group$$degrade implements IRouteGroup {
  @Override
  public void loadInto(Map<String, RouteMeta> atlas) {
    //  Warehouse.routes【存放具体的RouteMeta路由元数据】 - 存入全局降级策略Service(一种Provider)
    atlas.put("/degrade/Service", RouteMeta.build(RouteType.PROVIDER, DegradeServiceImpl.class, "/degrade/service", "degr
  }
}
```

> 4- `afterInit()` 会加载 `/arouter/service/interceptor` 也就是 `InterceptorServiceImpl`

```
    static void afterInit() {
        interceptorService = (InterceptorService) ARouter.getInstance().build("/arouter/service/interceptor").navigation(
    }
```

> 5- `build("/arouter/service/interceptor")` 中会先加载 `PathReplaceService`

```
    protected Postcard build(String path) {
        PathReplaceService pService = ARouter.getInstance().navigation(PathReplaceService.class);
        path = pService.forString(path);
        return build(path, extractGroup(path));
    }
```

> 6-加载 PathReplaceService 的 navigation(service)

```
/**=====================================================
     * 加载Service
     * // _ARouter.java
     * =====================================================*/
    protected <T> T navigation(Class<? extends T> service) {
        // 1. 用Service的Name构造对应的Postcard
        Postcard postcard = LogisticsCenter.buildProvider(service.getName());
        // 2. 可能是老版本，用Service的SimpleName构建
        if (null == postcard) {
            postcard = LogisticsCenter.buildProvider(service.getSimpleName());
        }
        // 3、不存在目标Service返回Null
        if (null == postcard) {
            return null;
        }
        // 4、存在目标Serivce，补全Postcard
        LogisticsCenter.completion(postcard);
        // 5、返回该Service(Provider)
        return (T) postcard.getProvider();
    }

    /**=========================================================
     * 2、构造Service(Provider)
     * 1. 查找到Warehouse.providersIndex中ServiceName对应的RouteMeta
     * 2. 用RouteMeta的path和group构造Postcard
     * // LogisticsCenter.java
     * =====================================================*/
    public static Postcard buildProvider(String serviceName) {
        // 1. 查找到Warehouse.providersIndex中ServiceName对应的RouteMeta
        RouteMeta meta = Warehouse.providersIndex.get(serviceName);
        if (null == meta) {
            return null;
        } else {
            // 2. 用RouteMeta的path和group构造Postcard
            return new Postcard(meta.getPath(), meta.getGroup());
        }
    }
```

> 7-构造出 /arouter/service/interceptor 也就是 InterceptorServiceImpl 对应的Postcard后正式开始navigation加载
> 8- navigation 中 LogisticsCenter.completion() 因为 Warehouse.routes 中找不到 InterceptorServiceImpl ， 因此会去加载 group = arouter 下面的所有节点，调用 ARouter$$Group$$arouter.loadInto() 加载 AutowiredServiceImpl 和 InterceptorServiceImpl
> 9- ARouter$$Group$$arouter 从 Warehouse.groupsIndex 中移除
> 10-构造出 InterceptorServiceImpl对象 并加入到 Warehouse.providers 中

2、实例分析navigation路由到Fragment的流程。

> 1. ARouter.getInstance().build(“/app/MainFragment”).navigation(); 会先加载 group = app 下面所有的Fragment、Activity
> 2. 实例化 MainFragment 并且返回。

3、实例分析navigation路由到Activity的流程。

> 1. ARouter.getInstance().build(“/app/MainActivity”).navigation(); 会先加载 group = app 下面所有的Fragment、Activity
> 2. 如果出错没找到，会调用 ARouter$$Group$$degrade.loadInto() 加载 DegradeServiceImpl 到providers中。并进行降级处理。
> 3. 如果没有出错，会通过 InterceptorServiceImpl 处理拦截器。
> 4. 最终实例化 Activity 并且携带参数跳转到目标页面。

问题补充

1、为什么会出现错误: There is no route match the path

> 1. 需要调用 `ARouter.openDebug()` 方法将标志位 `debuggable` 设置为 `true` 。不进入Debug模式不会弹出Toast，只会打印日志。
> 2. 不同module的一级路径相同，导致moudle中的一级路径失效，因此跳转到第二个module的某个页面时出现该错误。

参考资料

1. 可能是最详细的ARouter源码分析
2. Java注解处理器
3. 路由方案之ARouter源码分析
4. 什么时候使用CountDownLatch
5. CountDownLatch理解一：与join的区别
6. annotation(@Retention@Target)详解
7. Android编译时注解APT实战（AbstractProcessor）
8. Java AbstractProcessor实现自定义ButterKnife
9. JDK在线文档
10. Annotation实战【自定义AbstractProcessor】