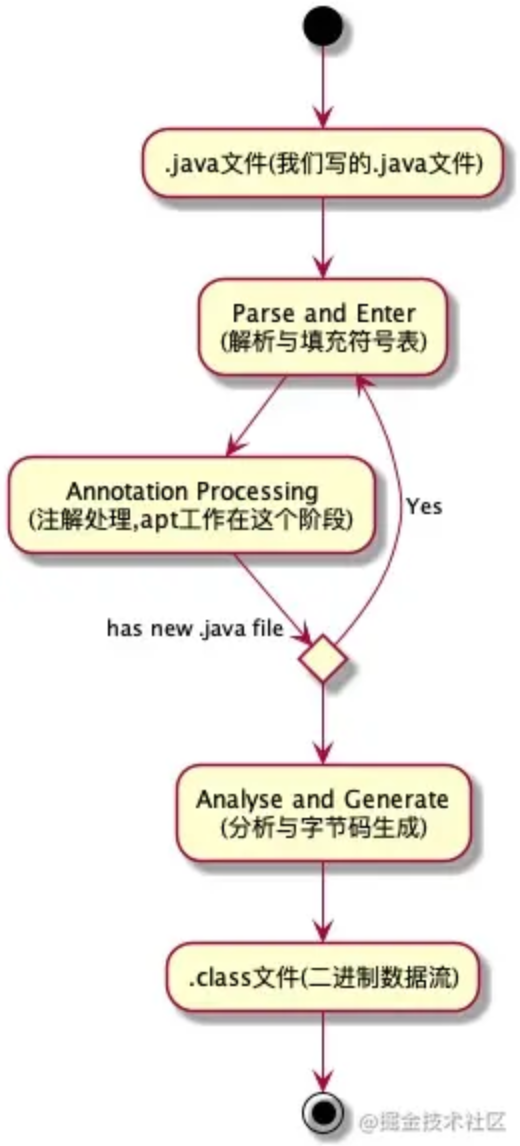
**名词介绍**

* apt:**APT**(Annotation Processing Tool)即**注解处理器**，是一种处理注解的工具，确切的说它是javac的一个工具，它用来在**编译时**扫描和处理注解。注解处理器以**Java代码**(或者编译过的字节码)作为输入，生成**java文件**作为输出。ARouter中通过处理注解生成相关路由信息类。



* asm:ASM 库是一款基于 Java 字节码层面的代码分析和修改工具。ASM 可以直接生产二进制的 class 文件，也可以在类被加载入 JVM 之前动态修改类行为。在ARouter中用于arouter\_register插件插入初始化代码。

**ARouter 路由使用分析**

ARouter.getInstance().build("/test/activity").navigation();

### ****要点步骤如下：****

1. 通过PathReplaceService 预处理路径，并从path:"/test/activity" 中抽取出 group: "test"
2. 将path 和group 作为参数创建 Postcard 实例
3. 调用 postcard#navigation ，最终导航到\_ARouter#navigation
4. 通过 group 和 path 从Warehouse.routes\*获取具体路径信息RouteMeta，完善postcard。

#### 第一步：抽取 group

不管是跳转Activity,获取Fragment实例，还是获取Provider 实例。 都是通过ARouter.getInstance().build("")方式来进行，这也是 ARouter最核心的路由api。这个build出来的产物，就是Postcard类。

protected Postcard build(String path) {

if (TextUtils.isEmpty(path)) {

throw new HandlerException(Consts.TAG + "Parameter is invalid!");

} else {

/// 用户自定义路径处理类。默认为空。 ARouter.getInstance().navigation 直接获取Provider后文分析

PathReplaceService pService = ARouter.getInstance().navigation(PathReplaceService.class);

if (null != pService) {

path = pService.forString(path);

}

///获取path中包含的 group 作为参数二

return build(path, extractGroup(path), true);

}

}

上面的代码中，先是通过ARouter获取了PathReplaceService 实例，对路径进行了预处理操作(默认没有自定义实现类)。再通过extractGroup方法从 路径中获取了 group信息。

private String extractGroup(String path) {

if (TextUtils.isEmpty(path) || !path.startsWith("/")) {

throw new HandlerException(Consts.TAG + "Extract the default group failed, the path must be start with '/' and contain more than 2 '/'!");

}

try {

String defaultGroup = path.substring(1, path.indexOf("/", 1));

if (TextUtils.isEmpty(defaultGroup)) {

throw new HandlerException(Consts.TAG + "Extract the default group failed! There's nothing between 2 '/'!");

} else {

return defaultGroup;

}

} catch (Exception e) {

logger.warning(Consts.TAG, "Failed to extract default group! " + e.getMessage());

return null;

}

}

从上面extractGroup 源码可知，抽取group的时候对路由路径 "/test/activity" 做了校验：

1. 一定要"/" 开头
2. 至少要有两个"/"
3. 第一个反斜杠后面的就是group

#### 第二步：创建Postcard实例

return new Postcard(path, group);

#### 第三步：调用\_ARouter#navigation

protected Object navigation(final Context context, final Postcard postcard, final int requestCode, final NavigationCallback callback) {

///1.自定义预处理代码

PretreatmentService pretreatmentService = ARouter.getInstance().navigation(PretreatmentService.class);

if (null != pretreatmentService && !pretreatmentService.onPretreatment(context, postcard)) {

// 预处理拦截了 返回

return null;

}

// 设置context

postcard.setContext(null == context ? mContext : context);

try {

///2.通过路由信息，找到对应的路由信息 RouteMeta ，根据路由类型 RouteType

///完善posrcard

LogisticsCenter.completion(postcard);

} catch (NoRouteFoundException ex) {

///... 省略异常日志和弹窗展示。以及相关回调方法

///值得一提的是走了 DegradeService 的自定义丢失回调

}

if (null != callback) {

callback.onFound(postcard);

}

///3.如果不是绿色通道，需要走拦截器：InterceptorServiceImpl

if (!postcard.isGreenChannel()) { // It must be run in async thread, maybe interceptor cost too mush time made ANR.

interceptorService.doInterceptions(postcard, new InterceptorCallback() {

@Override

public void onContinue(Postcard postcard) {

///4.继续导航方法

\_navigation(postcard, requestCode, callback);

}

@Override

public void onInterrupt(Throwable exception) {

///省略拦截后的一些代码

}

});

} else {

///4.继续导航方法

return \_navigation(postcard, requestCode, callback);

}

return null;

}

简单总结一下主要代码步骤：

1. 如有自定义预处理导航逻辑，执行并检查拦截
2. 通过path路径找到对应的routemeta路由信息，用该信息完善postcard对象（LogisticsCenter.completion方法中完成，细节后文分析）
3. 如果不是绿色通道，需要走拦截器：InterceptorServiceImpl 。该拦截器服务类中完成拦截器一一执行。(2的源码细节可知，PROVIDER和FRAGMENT类型是绿色通道)
4. 继续导航方法，调用\_navigation。

private Object \_navigation(final Postcard postcard, final int requestCode, final NavigationCallback callback) {

final Context currentContext = postcard.getContext();

switch (postcard.getType()) {

case ACTIVITY:

// Build intent

final Intent intent = new Intent(currentContext, postcard.getDestination());

//...省略完善intent代码

// Navigation in main looper.

runInMainThread(new Runnable() {

@Override

public void run() {

startActivity(requestCode, currentContext, intent, postcard, callback);

}

});

break;

case PROVIDER:

return postcard.getProvider();

case BOARDCAST:

case CONTENT\_PROVIDER:

case FRAGMENT:

Class<?> fragmentMeta = postcard.getDestination();

try {

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());

}

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;

}

很明显，代码中注意对各种类型的路由做了处理。

* \*ACTIVITY：\*新建Intent ，通过postcard信息，完善intent走context.startActivity 或者 context.startActivityForResult。
* PROVIDER：postcard.getProvider() 获取provider实例（实例化代码在LogisticsCenter.completion）
* FRAGMENT，BOARDCAST，CONTENT\_PROVIDER：routeMeta.getConstructor().newInstance() 通过路由信息实例化出实例，如果是Fragment的话，则另外再设置extras信息。
* \*METHOD，SERVICE：\*返回空，啥也不做。说明该类型路由调用navigation没啥意义。

看到这里，对于Activity 的路由跳转就很直观了，就是调用了startActivity 或者 startActivityForResult 方法，其他provider fragment等实例的获取也十分得清晰明了了，接下来讲讲上面提到的补全postcard关键代码。

#### 关键代码：LogisticsCenter.completion 分析

完善postcard信息代码是通过LogisticsCenter.completion 方法完成的。

/\*\*

\* 通过RouteMate 完善 postcard

\* @param postcard Incomplete postcard, should complete by this method.

\*/

public synchronized static void completion(Postcard postcard) {

//省略空判断

RouteMeta routeMeta = Warehouse.routes.get(postcard.getPath());

if (null == routeMeta) {

// 如果路由的组group没有找到，直接抛异常

if (!Warehouse.groupsIndex.containsKey(postcard.getGroup())) {

throw new NoRouteFoundException(TAG + "There is no route match the path [" + postcard.getPath() + "], in group [" + postcard.getGroup() + "]");

} else {

//...省略一些日志代码

// 1.动态添加组元素（从groupsIndex 中找到对应 IRouteGroup的生成类，再对组元素进行加载）

addRouteGroupDynamic(postcard.getGroup(), null);

completion(postcard); // Reload

}

} else {

postcard.setDestination(routeMeta.getDestination());

postcard.setType(routeMeta.getType());

postcard.setPriority(routeMeta.getPriority());

postcard.setExtra(routeMeta.getExtra());

Uri rawUri = postcard.getUri();

///2.如果有uri 信息，解析uri相关参数。解析出AutoWired的参数的值

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());

}

///3.获取provider实例，如果初始获取，初始化该provider， 最后赋值给postcard

switch (routeMeta.getType()) {

case PROVIDER: // if the route is provider, should find its instance

// Its provider, so it must implement IProvider

Class<? extends IProvider> providerMeta = (Class<? extends IProvider>) routeMeta.getDestination();

IProvider instance = Warehouse.providers.get(providerMeta);

if (null == instance) { // There's no instance of this provider

IProvider provider;

try {

provider = providerMeta.getConstructor().newInstance();

provider.init(mContext);

Warehouse.providers.put(providerMeta, provider);

instance = provider;

} catch (Exception e) {

logger.error(TAG, "Init provider failed!", e);

throw new HandlerException("Init provider failed!");

}

}

postcard.setProvider(instance);

postcard.greenChannel(); // Provider should skip all of interceptors

break;

case FRAGMENT:

postcard.greenChannel(); // Fragment needn't interceptors

default:

break;

}

}

}

梳理一下这一块的代码,这一部分代码完善了postcard信息,总共分成了三个要点

1. \*\*获取路由信息：\*\*如果路由信息找不到，通过组信息，重新动态添加组group内所有路由 ，调用addRouteGroupDynamic 。
2. \*\*获取uri内的参数：\*\*如果postcard创建的时候有传递uri。解析uri里面所有需要AutoInject的参数。放置到postcard中。
3. \*\*获取Provider实例，配置是否不走拦截器的绿色通道：\*\*不存在的Provider通过路由信息的 getDestination 反射创建实例并初始化，存在的直接获取。

分析到这里。各种RouteType的跳转，实例获取都已经明了了。 现在剩下的问题是，WareHouse里面的路由信息数据是哪里来的？前面提到了动态添加组内路由的方法addRouteGroupDynamic。

public synchronized static void addRouteGroupDynamic(String groupName, IRouteGroup group) throws NoSuchMethodException, IllegalAccessException, InvocationTargetException, InstantiationException {

if (Warehouse.groupsIndex.containsKey(groupName)){

// If this group is included, but it has not been loaded

// load this group first, because dynamic route has high priority.

Warehouse.groupsIndex.get(groupName).getConstructor().newInstance().loadInto(Warehouse.routes);

Warehouse.groupsIndex.remove(groupName);

}

// cover old group.

if (null != group) {

group.loadInto(Warehouse.routes);

}

}

可以看到Warehouse.routes 里面的所有路由信息，都是从 IRouteGroup.loadInto 加载出来的。而IRouteGroup 都存在Warehouse.groupsIndex *中。* 这时候新的问题出现了，Warehouse.groupsIndex的数据是哪里来的呢?

#### tips:上文中提到的对外可自定义配置:

简单罗列下源码中提到的可自定义配置的IProvider。便于使用的时候自定义。

* PathReplaceService ///路由自定义处理替换
* DegradeService //没有找到路由的通用回调
* PretreatmentService ///navigation 预处理拦截

### 通过Class 获取IProvider实例

前面提到的PathReplaceService 等用户自定义类，都是通过ARouter.getInstance().navigation(clazz) 方式获取的。

protected <T> T navigation(Class<? extends T> service) {

try {

//1.通过类名从Provider路由信息索引中，获取路由信息，组建postcart

Postcard postcard = LogisticsCenter.buildProvider(service.getName());

// Compatible 1.0.5 compiler sdk.

// Earlier versions did not use the fully qualified name to get the service

if (null == postcard) {

// No service, or this service in old version.

//1.通过类名从Provider路由信息索引中，获取路由信息，组建postcart

postcard = LogisticsCenter.buildProvider(service.getSimpleName());

}

if (null == postcard) {

return null;

}

// Set application to postcard.

postcard.setContext(mContext);

//2.完善postcard ,该方法里面创建provider

LogisticsCenter.completion(postcard);

return (T) postcard.getProvider();

} catch (NoRouteFoundException ex) {

logger.warning(Consts.TAG, ex.getMessage());

return null;

}

}

public static Postcard buildProvider(String serviceName) {

RouteMeta meta = Warehouse.providersIndex.get(serviceName);

if (null == meta) {

return null;

} else {

return new Postcard(meta.getPath(), meta.getGroup());

}

}

和路由组信息获取类似，Provider的路由信息从 Warehouse.providersIndex 维护的映射表中获取。 所以providersIndex是专门用来给没有@Route 路由信息的Provider创建实例用的。这就是维护providersIndex的用途。

### 小结

路由跳转以及获取Provider等实例的原理可以简单总结下：

1. 先是获取postcard，可能是直接通过路由路径和uri构建, 如"/test/activity1"，也可能是通过Provider 类名从索引获取，如PathReplaceService.class.getName()
2. 然后通过RouteMate完善 postcard。获取诸如类名信息,路由类型，provider实例等信息。
3. 最后导航，根据路由类型作出跳转或者返回对应实例。

关键点在于WareHouse 维护的路由图谱。

# ARouter初始化分析

public static void init(Application application) {

if (!hasInit) {

logger = \_ARouter.logger;

\_ARouter.logger.info(Consts.TAG, "ARouter init start.");

///调用初始化代码

hasInit = \_ARouter.init(application);

///初始化完成后，加载拦截器服务，并初始化所有拦截器

if (hasInit) {

\_ARouter.afterInit();

}

\_ARouter.logger.info(Consts.TAG, "ARouter init over.");

}

}

代码关键就两步，

1. 初始化ARouter
2. 获取拦截器服务实例初始化所有拦截器

### 初始化ARouter

public synchronized static void init(Context context, ThreadPoolExecutor tpe) throws HandlerException {

mContext = context;

executor = tpe;

try {

long startInit = System.currentTimeMillis();

//load by plugin first

//1.加载路由映射表 （通过 ARouter 插件 注册）

loadRouterMap();

//2.是否通过插件注册初始化

if (registerByPlugin) {

logger.info(TAG, "Load router map by ARouter-auto-register plugin.");

} else {

Set<String> routerMap;

// It will rebuild router map every times when debuggable.

//可调试 和新版本的时候 重建路由表

if (ARouter.debuggable() || PackageUtils.isNewVersion(context)) {

logger.info(TAG, "Run with debug mode or new install, rebuild router map.");

// These class was generated by ARouter-compiler.

//3.在线程池中,扫描所有dex文件,通过包名获取路由映射表类名

//包名 ROUTE\_ROOT\_PAKCAGE :com.alibaba.android.ARouter.routes

routerMap = ClassUtils.getFileNameByPackageName(mContext, ROUTE\_ROOT\_PAKCAGE);

///保存路由映射表类名们到缓存中

if (!routerMap.isEmpty()) {

context.getSharedPreferences(ARouter\_SP\_CACHE\_KEY, Context.MODE\_PRIVATE).edit().putStringSet(ARouter\_SP\_KEY\_MAP, routerMap).apply();

}

PackageUtils.updateVersion(context); // Save new version name when router map update finishes.

} else {

///从 SharedPreferences 缓存中获取所有的路由类文件

logger.info(TAG, "Load router map from cache.");

routerMap = new HashSet<>(context.getSharedPreferences(ARouter\_SP\_CACHE\_KEY, Context.MODE\_PRIVATE).getStringSet(ARouter\_SP\_KEY\_MAP, new HashSet<String>()));

}

logger.info(TAG, "Find router map finished, map size = " + routerMap.size() + ", cost " + (System.currentTimeMillis() - startInit) + " ms.");

startInit = System.currentTimeMillis();

///4.加载 IRouteRoot，IInterceptorGroup，IProviderGroup 类，填充Warehouse 的路由信息组索引

for (String className : routerMap) {

if (className.startsWith(ROUTE\_ROOT\_PAKCAGE + DOT + SDK\_NAME + SEPARATOR + SUFFIX\_ROOT)) {

// This one of root elements, load root.

((IRouteRoot) (Class.forName(className).getConstructor().newInstance())).loadInto(Warehouse.groupsIndex);

} else if (className.startsWith(ROUTE\_ROOT\_PAKCAGE + DOT + SDK\_NAME + SEPARATOR + SUFFIX\_INTERCEPTORS)) {

// Load interceptorMeta

((IInterceptorGroup) (Class.forName(className).getConstructor().newInstance())).loadInto(Warehouse.interceptorsIndex);

} else if (className.startsWith(ROUTE\_ROOT\_PAKCAGE + DOT + SDK\_NAME + SEPARATOR + SUFFIX\_PROVIDERS)) {

// Load providerIndex

((IProviderGroup) (Class.forName(className).getConstructor().newInstance())).loadInto(Warehouse.providersIndex);

}

}

}

//..省略路由类初始化结果的日志代码

} catch (Exception e) {

throw new HandlerException(TAG + "ARouter init logistics center exception! [" + e.getMessage() + "]");

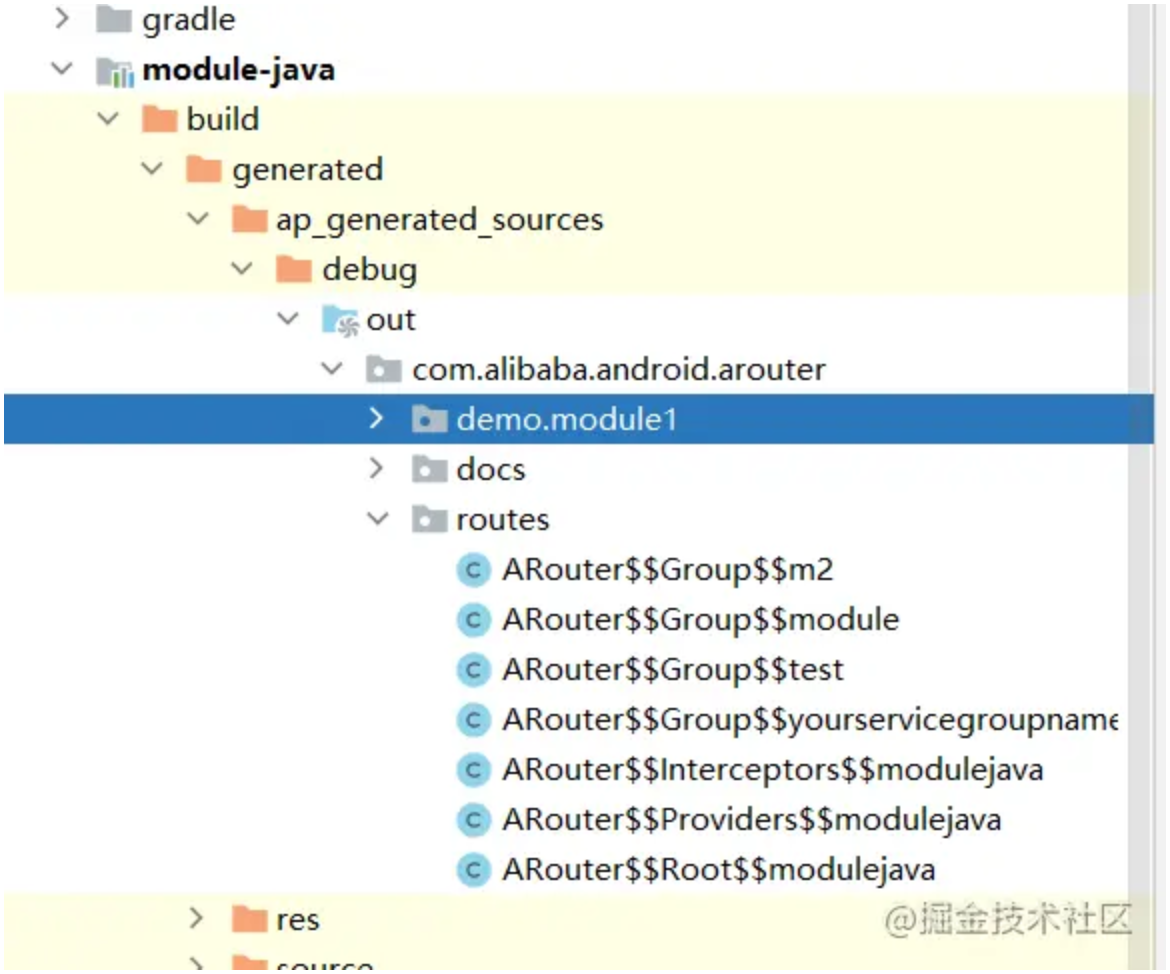
}

}

通过代码我们了解到了这么几个过程:

1. 方式一 : ARouter-auto-register 插件加载路由表（如果有该插件）。
2. 方式二 :
3. 在需要的时候扫描所有 dex文件,找到所有包名为com.alibaba.android.ARouter.routes的类,类名放到routerMap 集里面。
4. 实例化上面找到的所有类，并通过这些集类加载对应的集映射索引到WareHouse中。

很显然，com.alibaba.android.ARouter.routes 包名下面的类都是自动生成的路由表类。 通过搜索我们能找到样例代码中生成的该包名对象们:



module\_java 生成的IRouteRoot 代码如下所示

public class ARouter$$Root$$modulejava implements IRouteRoot {

@Override

public void loadInto(Map<String, Class<? extends IRouteGroup>> routes) {

routes.put("m2", ARouter$$Group$$m2.class);

routes.put("module", ARouter$$Group$$module.class);

routes.put("test", ARouter$$Group$$test.class);

routes.put("yourservicegroupname", ARouter$$Group$$yourservicegroupname.class);

}

}

初始化ARouter的过程,其实就是填充Warehouse 的providersIndex，groupsIndex，interceptorsIndex。

### 初始化后续:初始化所有拦截器

static void afterInit() {

// Trigger interceptor init, use byName.

interceptorService = (InterceptorService) ARouter.getInstance().build("/ARouter/service/interceptor").navigation();

}

这一块代码就是navigation 获取到了InterceptorService。上面也讲过,在执行navigation的时候，会调用IProvider的init方法。所以我们需要找到InterceptorService 的实现类，并看看他的init做了什么。项目中其实现类是InterceptorServiceImpl,找到init代码如下：

@Override

public void init(final Context context) {

LogisticsCenter.executor.execute(new Runnable() {

@Override

public void run() {

if (MapUtils.isNotEmpty(Warehouse.interceptorsIndex)) {

for (Map.Entry<Integer, Class<? extends IInterceptor>> entry : Warehouse.interceptorsIndex.entrySet()) {

Class<? extends IInterceptor> interceptorClass = entry.getValue();

try {

IInterceptor iInterceptor = interceptorClass.getConstructor().newInstance();

iInterceptor.init(context);

Warehouse.interceptors.add(iInterceptor);

} catch (Exception ex) {

throw new HandlerException(TAG + "ARouter init interceptor error! name = [" + interceptorClass.getName() + "], reason = [" + ex.getMessage() + "]");

}

}

interceptorHasInit = true;

logger.info(TAG, "ARouter interceptors init over.");

synchronized (interceptorInitLock) {

interceptorInitLock.notifyAll();

}

}

}

});

}

代码很明白的告诉我们，该初始化代码从拦截器路由信息索引里面加载并实例化了所有拦截器。然后通知等待的拦截器开始拦截。

### 小结

看完初始化代码之后,明白了WareHouse的数据来源,现在问题变成了com.alibaba.android.ARouter.routes 包名的代码何时生成。我们且看下回分解。

# ARouter 注解处理器：arouter-compiler

ARouter 生成路由信息代码利用了注解处理器的特性。 arouter-compiler 就是注解处理代码模块

### RouteProcessor注解处理器处理流程说明

@AutoService(Processor.class)

@SupportedAnnotationTypes({ANNOTATION\_TYPE\_ROUTE, ANNOTATION\_TYPE\_AUTOWIRED})

public class RouteProcessor extends BaseProcessor {

@Override

//在该方法中可以获取到processingEnvironment对象，

//借由该对象可以获取到生成代码的文件对象, debug输出对象，以及一些相关工具类

public synchronized void init(ProcessingEnvironment processingEnv) {

//...

super.init(processingEnv);

}

@Override

//返回所支持的java版本，一般返回当前所支持的最新java版本即可

public SourceVersion getSupportedSourceVersion() {

//...

return super.getSupportedSourceVersion();

}

@Override

//必须实现 扫描所有被注解的元素，并作处理，最后生成文件。该方法的返回值为boolean类型，若返回true,

//则代表本次处理的注解已经都被处理，不希望下一个注解处理器继续处理，

//否则下一个注解处理器会继续处理。

public boolean process(Set<? extends TypeElement> set, RoundEnvironment roundEnvironment) {

//...

return false;

}

}

RouteProcessor 继承 BaseProcessor 间接继承了AbstractProcessor，在BaseProcessor#init 方法中，获取到processingEnv 中的各种实用工具，以供处理注解使用。 值得一提的是，init 中获取了moduleName 和 generateDoc 参数代码如下:

if (MapUtils.isNotEmpty(options)) {

///AROUTER\_MODULE\_NAME

moduleName = options.get(KEY\_MODULE\_NAME);

///AROUTER\_GENERATE\_DOC

generateDoc = VALUE\_ENABLE.equals(options.get(KEY\_GENERATE\_DOC\_NAME));

}

这一块就是我们常常需要在gradle中配置的arguments 的由来：

android {

defaultConfig {

javaCompileOptions {

annotationProcessorOptions {

arguments = [AROUTER\_MODULE\_NAME: project.getName(), AROUTER\_GENERATE\_DOC: "enable"]

}

}

}

}

//或者kotlin

kapt {

arguments {

arg("AROUTER\_MODULE\_NAME", project.getName())

}

}

接下来看RouteProcessor#process方法的具体实现:

Set<? extends Element> routeElements = roundEnv.getElementsAnnotatedWith(Route.class);

this.parseRoutes(routeElements);

代码中拿到了所有标注@Route注解的相关类元素。 然后在parseRoutes方法中进行处理。

private void parseRoutes(Set<? extends Element> routeElements) throws IOException {

if (CollectionUtils.isNotEmpty(routeElements)) {

// prepare the type an so on.

logger.info(">>> Found routes, size is " + routeElements.size() + " <<<");

rootMap.clear();

///省略类型获取代码

/\*...省略构建'loadInto'方法描述,通过定义变量名，

定义类型最后得出 MethodSpec.Builder

void loadInto(Map<String, Class<? extends IRouteGroup>> atlas);

\*/

MethodSpec.Builder loadIntoMethodOfRootBuilder;

// Follow a sequence, find out metas of group first, generate java file, then statistics them as root.

for (Element element : routeElements) {

//..省略相关代码，根据element类型，创建出对应的RouteMate实例，得到路由信息，

//并且通过injectParamCollector 方法将Activity和Fragmentr内部的所有@AutoWired

//注解 的信息放到MetaData的 paramsType 和injectConfig 中

v

//对 routeMate进行分类,在groupMap中填充对应数据

categories(routeMeta);

}

/\*...省略构建'loadInto'方法描述,通过定义变量名，

定义类型最后得出 MethodSpec.Builder,主要用来构建providers索引。

void loadInto(Map<String, RouteMeta> providers);

\*/

MethodSpec.Builder loadIntoMethodOfProviderBuilder;

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

// Start generate java source, structure is divided into upper and lower levels, used for demand initialization.

for (Map.Entry<String, Set<RouteMeta>> entry : groupMap.entrySet()) {

String groupName = entry.getKey();

/\*\*获取方法描述

void loadInto(Map<String, RouteMeta> atlas)\*/

MethodSpec.Builder loadIntoMethodOfGroupBuilder;

///文档列表，后续转为json文件 存在arouter-map-of-{module\_name}.json 中

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

// 为loadInto方法添加方法体，

// 第一步：遍历组下面的所有RouteMate,如果是IProvider子类，添加到 providers 的loadInto方法中

// 第二步：添加参数列表

Set<RouteMeta> groupData = entry.getValue();

for (RouteMeta routeMeta : groupData) {

RouteDoc routeDoc = extractDocInfo(routeMeta);

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

//省略第一步provider的添加代码

// Make map body for paramsType

StringBuilder mapBodyBuilder = new StringBuilder();

Map<String, Integer> paramsType = routeMeta.getParamsType();

Map<String, Autowired> injectConfigs = routeMeta.getInjectConfig();

if (MapUtils.isNotEmpty(paramsType)) {

//..省略构建参数方法体字符串，以及添加参数们到doc 数据中

}

String mapBody = mapBodyBuilder.toString();

//添加IRouteGroup#loadInto的方法体

loadIntoMethodOfGroupBuilder.addStatement(

"atlas.put($S, $T.build($T." + routeMeta.getType() + ", $T.class, $S, $S, " + (StringUtils.isEmpty(mapBody) ? null : ("new java.util.HashMap<String, Integer>(){{" + mapBodyBuilder.toString() + "}}")) + ", " + routeMeta.getPriority() + ", " + routeMeta.getExtra() + "))",

/\*...\*/);

routeDoc.setClassName(className.toString());

routeDocList.add(routeDoc);

}

// Generate groups

//1.生成对应的IRouteGroup 类代码文件 ARouter$$Group$$[GroupName]

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);

}

if (MapUtils.isNotEmpty(rootMap)) {

// Generate root meta by group name, it must be generated before root, then I can find out the class of group.

for (Map.Entry<String, String> entry : rootMap.entrySet()) {

loadIntoMethodOfRootBuilder.addStatement("routes.put($S, $T.class)", entry.getKey(), ClassName.get(PACKAGE\_OF\_GENERATE\_FILE, entry.getValue()));

}

}

// 2.Output route doc 写入json到doc文档中

if (generateDoc) {

docWriter.append(JSON.toJSONString(docSource, SerializerFeature.PrettyFormat));

docWriter.flush();

docWriter.close();

}

// Write provider into disk

//3.生成对应的IProviderGroup 类代码文件 ARouter$$Providers$$[moduleName]

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 + " <<<");

// Write root meta into disk.

//4. 生成对应的IRouteRoot 类代码文件 ARouter$$Root$$[moduleName]

String rootFileName = NAME\_OF\_ROOT + SEPARATOR + moduleName;

JavaFile.builder(PACKAGE\_OF\_GENERATE\_FILE,

TypeSpec.classBuilder(rootFileName)

.addJavadoc(WARNING\_TIPS)

.addSuperinterface(ClassName.get(elementUtils.getTypeElement(ITROUTE\_ROOT)))

.addModifiers(PUBLIC)

.addMethod(loadIntoMethodOfRootBuilder.build())

.build()

).build().writeTo(mFiler);

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

}

}

代码很长，关键结果就是三点，

1. 在com.alibaba.android.arouter.routes 包名下生成ARouter$$Group$$[GroupName] 类，包含路由组的所有路由信息。
2. 在该包名下生成ARouter$$Root$$[moduleName]类，包含所有组的信息。
3. 在该包名下生成ARouter$$Providers$$[moduleName] 类，包含所有Providers索引
4. 在docs下，生成文件名为 "arouter-map-of-" + moduleName + ".json" 的文档。

### 其他注解处理器说明

剩下还有两个注解处理器 InterceptorProcessor 和 AutowiredProcessor。 生成代码逻辑大同小异，只是逻辑复杂度的区别，

* AutowiredProcessor :处理@Autowired注解的参数，以参数所在对应的类分类，生成[classSimpleName]$$ARouter$$Autowired 代码文件，以在Activity或者Fragment跳转的时候自动从intent中获取数据，并对activity 和 fragment 对象赋值。
* InterceptorProcessor: 处理@Interceptor注解。生成对应的ARouter$$Interceptors$$[modulename]代码文件，提供拦截器功能。

值得一提的是，对于自定义类型的@AutoWired，ARouter提供了 SerializationService进行自定义，用户只需要实现该解析类就行。

### 小结

这个模块完成了之前ARouter初始化所需要的所有代码的生成。

# ARouter 自动注册插件：arouter-register

### 注册转换器

ARouter-register 插件通过 registerTransform api。添加了一个自定义Transform，对dex进行自定义处理。 直接看 该源码中的入口代码 PluginLaunch#apply

def isApp = project.plugins.hasPlugin(AppPlugin)

//only application module needs this plugin to generate register code

if (isApp) {

def android = project.extensions.getByType(AppExtension)

def transformImpl = new RegisterTransform(project)

android.registerTransform(transformImpl)

}

### 扫描class文件和jar文件，保存路由类信息

那工序做了什么呢？看看代码RegisterTransform#transform：

@Override

void transform(Context context, Collection<TransformInput> inputs

, Collection<TransformInput> referencedInputs

, TransformOutputProvider outputProvider

, boolean isIncremental) throws IOException, TransformException, InterruptedException {

Logger.i('Start scan register info in jar file.')

long startTime = System.currentTimeMillis()

boolean leftSlash = File.separator == '/'

inputs.each { TransformInput input ->

//通过AMS 的 ClassVisistor 扫描所有的jar 文件，将所有扫描到的IRouteRoot IInterceptorGroup IInterceptorGroup类

//都加到ScanSetting 的 classList中

//详情可以看看 ScanClassVisitor

//如果jar包是 LogisticsCenter.class，标记该类文件到 fileContainsInitClass

input.jarInputs.each { JarInput jarInput ->

//排除对于support库，以及m2repository 内第三方库的扫描。scan jar file to find classes

if (ScanUtil.shouldProcessPreDexJar(src.absolutePath)) {

//扫描

ScanUtil.scanJar(src, dest)

}

//..省略重命名扫描过的jar包相关代码

}

// scan class files

//..省略扫描class文件相关代码，方式类似扫描jar包

}

Logger.i('Scan finish, current cost time ' + (System.currentTimeMillis() - startTime) + "ms")

//如果存在 LogisticsCenter.class 类文件

//插入注册代码到 LogisticsCenter.class 中

if (fileContainsInitClass) {

registerList.each { ext ->

//...省略一些判空和日志代码

///插入初始化代码

RegisterCodeGenerator.insertInitCodeTo(ext)

}

}

Logger.i("Generate code finish, current cost time: " + (System.currentTimeMillis() - startTime) + "ms")

}

从代码中可知，这一块代码有四个关键点。

1. 通过ASM扫描了对应的jar 文件和class文件，并将扫描到的对应routes包下的类加入到ScanSetting 的classList属性 中
2. 如果扫描到包含LogisticsCenter.class 类文件，将该文件记录到fileContainsInitClass 字段中。
3. 扫描完成的文件重命名。
4. 最后通过RegisterCodeGenerator.insertInitCodeTo(ext) 方法插入初始化代码到LogisticsCenter.class中。

### 遍历包含入口class的jar文件，准备插入代码

在RegisterCodeGenerator.insertInitCodeTo(ext)代码中，先判断ScanSetting#classList是否为空，再判断文件是否是jar文件。如果判断都过了，最后走到 RegisterCodeGenerator#insertInitCodeIntoJarFile代码：

private File insertInitCodeIntoJarFile(File jarFile) {

//将包含 LogisticsCenter.class 的 jar文件，插入初始化代码

//操作在 \*\*\*.jar.opt 临时文件做

if (jarFile) {

def optJar = new File(jarFile.getParent(), jarFile.name + ".opt")

if (optJar.exists())

optJar.delete()

///通过JarFile 和JarEntry

def file = new JarFile(jarFile)

Enumeration enumeration = file.entries()

JarOutputStream jarOutputStream = new JarOutputStream(new FileOutputStream(optJar))

//遍历jar中的所有class，查询修改

while (enumeration.hasMoreElements()) {

JarEntry jarEntry = (JarEntry) enumeration.nextElement()

String entryName = jarEntry.getName()

ZipEntry zipEntry = new ZipEntry(entryName)

InputStream inputStream = file.getInputStream(jarEntry)

jarOutputStream.putNextEntry(zipEntry)

///如果是LogisticsCenter.class文件，调用referHackWhenInit 插入代码

///如果不是，不改变数据直接写入

if (ScanSetting.GENERATE\_TO\_CLASS\_FILE\_NAME == entryName) {

Logger.i('Insert init code to class >> ' + entryName)

//！！！！重点代码,插入初始化代码

def bytes = referHackWhenInit(inputStream)

jarOutputStream.write(bytes)

} else {

jarOutputStream.write(IOUtils.toByteArray(inputStream))

}

inputStream.close()

jarOutputStream.closeEntry()

}

jarOutputStream.close()

file.close()

if (jarFile.exists()) {

jarFile.delete()

}

optJar.renameTo(jarFile)

}

return jarFile

}

从代码中可知，按步骤梳理：

1. 创建临时文件，\*\*\*.jar.opt
2. 通过输入输出流，遍历jar文件下面的所有class，判断是否LogisticCenter.class
3. 对LogisticCenter.class 调用 referHackWhenInit 方法插入初始化代码，写入到opt临时文件
4. 对其他class 原封不动写入opt临时文件
5. 删除原来的jar文件，将临时文件改名为原来的jar文件名

这一步完成了对于jar文件的修改。插入了ARouter的自动注册初始化代码。

### 插入初始化代码

插入操作主要是找到 LogisticCenter 关键的插入代码在于RegisterCodeGenerator#referHackWhenInit：

private byte[] referHackWhenInit(InputStream inputStream) {

ClassReader cr = new ClassReader(inputStream)

ClassWriter cw = new ClassWriter(cr, 0)

ClassVisitor cv = new MyClassVisitor(Opcodes.ASM5, cw)

cr.accept(cv, ClassReader.EXPAND\_FRAMES)

return cw.toByteArray()

}

可以看到代码中利用了ams 框架的 ClassVisitor 来访问入口类。 再看MyClassVisistor 的visitMethod 实现：

@Override

MethodVisitor visitMethod(int access, String name, String desc,

String signature, String[] exceptions) {

MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions)

//generate code into this method

//针对loadRouterMap 方法进行处理

if (name == ScanSetting.GENERATE\_TO\_METHOD\_NAME) {

mv = new RouteMethodVisitor(Opcodes.ASM5, mv)

}

return mv

}

可以看到，当asm访问的方法名为loadRouterMap时候，就通过RouteMethodVisitor 对齐进行操作，具体代码如下：

class RouteMethodVisitor extends MethodVisitor {

RouteMethodVisitor(int api, MethodVisitor mv) {

super(api, mv)

}

@Override

void visitInsn(int opcode) {

//generate code before return

if ((opcode >= Opcodes.IRETURN && opcode <= Opcodes.RETURN)) {

extension.classList.each { name ->

name = name.replaceAll("/", ".")

mv.visitLdcInsn(name)//类名

// generate invoke register method into LogisticsCenter.loadRouterMap()

mv.visitMethodInsn(Opcodes.INVOKESTATIC

, ScanSetting.GENERATE\_TO\_CLASS\_NAME

, ScanSetting.REGISTER\_METHOD\_NAME

, "(Ljava/lang/String;)V"

, false)

}

}

super.visitInsn(opcode)

}

@Override

void visitMaxs(int maxStack, int maxLocals) {

super.visitMaxs(maxStack + 4, maxLocals)

}

}

* visitLdcInsn：访问ldc指令，向栈中压入参数
* visitMethodInsn：调用方法的指令，上面代码中，用来调用LogisticsCenter.register(String className)方法
* visitMaxs: 用以确定类方法在执行时候的堆栈大小。

### 小结

对这里我们就十分清晰了插入初始化代码的路径。

1. 首先是扫描所有的jar和class，找到对应的routes包名的类文件和 包含 LogisticsCenter.class 类的jar文件。类文件名依据类别存放在ScanSetting中。
2. 找到LogisticsCenter.class ，对他进行字节码操作，插入初始化代码。

整个register插件的流程就完成了

作者：guuguo  
链接：https://juejin.cn/post/6981263258015498248  
来源：掘金  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。