# 阿里 P8 大牛手写 Android 高级工程师三方库学习笔记

目录

# 三方库源码笔记（1）-EventBus 源码详解

我们知道，EventBus 在有消息被发送出来时，可以直接为我们回调该消息的所有监听方法，回调操作是通过反射 `method.invoke` 来实现的。那么 EventBus 在回调之前也必须先拿到所有的监听方法才行，这样才知道该消息类型对应什么监听方法以及对应多少监听方法

EventBus 获取监听方法的方式有两种：

- 不配置注解处理器。在 subscriber 进行 register 时通过反射获取到，这种方式是在运行时实现的
- 配置注解处理器。预先解析监听方法到辅助文件中，在运行时就可以直接拿到所有的解析结果而不必依靠反射来实现，这种方式是在编译阶段实现的，相比第一种方式性能会高很多

这里先介绍第一种方式，这种方式只需要导入如下依赖即可

```
implementation "org.greenrobot:eventbus:3.2.0"

复制代码
```

## 一、注册

### EventBus.java

```
    public void register(Object subscriber) {

        Class<?> subscriberClass = subscriber.getClass();

        List<SubscriberMethod> subscriberMethods = subscriberMethodFinder.findSubscr
iberMethods(subscriberClass);

        synchronized (this) {

            for (SubscriberMethod subscriberMethod : subscriberMethods) {
```

```
                subscribe(subscriber, subscriberMethod);

            }

        }

    }
```

复制代码

EventBus 的注册操作是通过 `register(Object)`方法来完成的。该方法会对注册类进行解析，将注册类包含的所有声明了 `@Subscribe` 注解的方法的签名信息保存到内存中，这样当有消息被 Post 时，就可以直接在内存中查找到目标方法了

从 `SubscriberMethod` 类包含的所有参数可以看出来，它包含了我们对 `@Subscribe` 的配置信息以及对应的方法签名信息

```
public class SubscriberMethod {

    final Method method;

    final ThreadMode threadMode;

    final Class<?> eventType;

    final int priority;

    final boolean sticky;

    /** Used for efficient comparison */

    String methodString;



    ...



}
```

复制代码

这个查找的过程是通过 `SubscriberMethodFinder` 类来完成的

## SubscriberMethodFinder.java

这里来看下 `SubscriberMethodFinder` 是如何遍历获取到所有声明了`@Subscribe` 注解的方法

```
    private static final Map<Class<?>, List<SubscriberMethod>> METHOD_CACHE = new Con
currentHashMap<>();



    List<SubscriberMethod> findSubscriberMethods(Class<?> subscriberClass) {

        List<SubscriberMethod> subscriberMethods = METHOD_CACHE.get(subscriberClas
s);

        if (subscriberMethods != null) {

            return subscriberMethods;

        }



        if (ignoreGeneratedIndex) {

            subscriberMethods = findUsingReflection(subscriberClass);

        } else {

            subscriberMethods = findUsingInfo(subscriberClass);

        }

        if (subscriberMethods.isEmpty()) {

            //如果为空，说明不包含使用 @Subscribe 注解的方法，那么 register 操作就是没有意
义的，直接抛出异常

            throw new EventBusException("Subscriber " + subscriberClass
```

```
                    + " and its super classes have no public methods with the @Subscri
be annotation");

        } else {

            METHOD_CACHE.put(subscriberClass, subscriberMethods);

            return subscriberMethods;

        }

    }
```

复制代码

`SubscriberMethodFinder` 会将每次的查找结果缓存到 `METHOD_CACHE` 中，这对某些会先后经历**多次注册和反注册**操作的页面来说会比较有用，因为每次查找可能需要依靠多次循环遍历和反射操作，会稍微有点消耗性能

因为 `ignoreGeneratedIndex` 默认值是 false，所以这里直接看 `findUsingInfo(subscriberClass)` 方法

其主要逻辑是：

1. 通过 `prepareFindState()` 方法从对象池 `FIND_STATE_POOL` 中获取空闲的 `FindState` 对象，如果不存在则初始化一个新的，并在使用过后通过 `getMethodsAndRelease` 方法将对象还给对象池。通过对象池来避免无限制地创建 `FindState` 对象，这也算做是一个优化点
2. 在不使用注解处理器的情况下 `findState.subscriberInfo` 和 `subscriberInfoIndexes` 默认都是等于 null 的，所以主要看 `findUsingReflectionInSingleClass` 方法即可，从该方法名可知是通过反射操作来进行解析的。解析结果会被存到 `findState` 中
3. 因为父类注册的监听方法会被子类继承到，而解析过程是会从子类向其父类依次遍历的，所以在解析完子类后需要通过 `findState.moveToSuperclass()` 方法将下一个查找的 class 对象指向父类

```
    private static final int POOL_SIZE = 4;
```

```
    private static final FindState[] FIND_STATE_POOL = new FindState[POOL_SIZE];



    private List<SubscriberMethod> findUsingInfo(Class<?> subscriberClass) {

        //步骤1

        FindState findState = prepareFindState();

        findState.initForSubscriber(subscriberClass);

        while (findState.clazz != null) {

            findState.subscriberInfo = getSubscriberInfo(findState);

            if (findState.subscriberInfo != null) {

                SubscriberMethod[] array = findState.subscriberInfo.getSubscriberMeth
ods();

                for (SubscriberMethod subscriberMethod : array) {

                    if (findState.checkAdd(subscriberMethod.method, subscriberMethod.
eventType)) {

                        findState.subscriberMethods.add(subscriberMethod);

                    }

                }

            } else {

                //步骤2

                findUsingReflectionInSingleClass(findState);

            }

            //步骤3

            findState.moveToSuperclass();

        }
```

```
        return getMethodsAndRelease(findState);
    }

    private List<SubscriberMethod> getMethodsAndRelease(FindState findState) {
        List<SubscriberMethod> subscriberMethods = new ArrayList<>(findState.subscriberMethods);
        findState.recycle();
        synchronized (FIND_STATE_POOL) {
            //回收 findState，尝试将之存到对象池中
            for (int i = 0; i < POOL_SIZE; i++) {
                if (FIND_STATE_POOL[i] == null) {
                    FIND_STATE_POOL[i] = findState;
                    break;
                }
            }
        }
        return subscriberMethods;
    }

    //如果对象池中有可用的对象则取出来使用，否则的话就构建一个新的
    private FindState prepareFindState() {
        synchronized (FIND_STATE_POOL) {
            for (int i = 0; i < POOL_SIZE; i++) {
```

```
                FindState state = FIND_STATE_POOL[i];
                if (state != null) {
                    FIND_STATE_POOL[i] = null;
                    return state;
                }
            }
        }
        return new FindState();
    }
```

复制代码

这里来主要看下 `findUsingReflectionInSingleClass` 方法是如何完成反射操作的。如果解析到的方法签名不符合要求，则会在开启了**严格检查**的情况下会直接抛出异常；如果方法签名符合要求，则会将方法签名保存到 `subscriberMethods` 中

```
    private void findUsingReflectionInSingleClass(FindState findState) {
        Method[] methods;
        try {
            // This is faster than getMethods, especially when subscribers are fat cla
sses like Activities
            //获取 clazz 包含的所有方法，不包含继承得来的方法
            methods = findState.clazz.getDeclaredMethods();
        } catch (Throwable th) {
            // Workaround for java.lang.NoClassDefFoundError, see https://github.com/
greenrobot/EventBus/issues/149
            try {
```

```
                //获取 clazz 以及其父类的所有 public 方法

                methods = findState.clazz.getMethods();

            } catch (LinkageError error) { // super class of NoClassDefFoundError to b
e a bit more broad...

                String msg = "Could not inspect methods of " + findState.clazz.getName
();

                if (ignoreGeneratedIndex) {

                    msg += ". Please consider using EventBus annotation processor to a
void reflection.";

                } else {

                    msg += ". Please make this class visible to EventBus annotation pr
ocessor to avoid reflection.";

                }

                throw new EventBusException(msg, error);

            }

            //由于 getDeclaredMethods() 都抛出异常了，就不再继续向下循环了，所以指定下次循
环时忽略父类

            findState.skipSuperClasses = true;

        }

        for (Method method : methods) {

            int modifiers = method.getModifiers();

            if ((modifiers & Modifier.PUBLIC) != 0 && (modifiers & MODIFIERS_IGNORE)
== 0) {

                //method 是 public 的，且不是 ABSTRACT、STATIC、BRIDGE、SYNTHETIC



                Class<?>[] parameterTypes = method.getParameterTypes();
```

```
                if (parameterTypes.length == 1) {  //方法包含的参数个数是一

                    Subscribe subscribeAnnotation = method.getAnnotation(Subscribe.cla
ss);

                    if (subscribeAnnotation != null) { //方法签名包含 Subscribe 注解

                        Class<?> eventType = parameterTypes[0];

                        if (findState.checkAdd(method, eventType)) {

                            //校验通过后，就将 Subscribe 注解的配置信息及 method 方法签名保
存起来

                            ThreadMode threadMode = subscribeAnnotation.threadMode();

                            findState.subscriberMethods.add(new SubscriberMethod(metho
d, eventType, threadMode,

                                    subscribeAnnotation.priority(), subscribeAnnotation.
sticky()));

                        }

                    }

                } else if (strictMethodVerification && method.isAnnotationPresent(Sub
scribe.class)) {

                    //因为 EventBus 只支持包含一个入参参数的注解函数，所以如果开启了严格的方
法校验那么就抛出异常

                    String methodName = method.getDeclaringClass().getName() + "." + m
ethod.getName();

                    throw new EventBusException("@Subscribe method " + methodName +

                            "must have exactly 1 parameter but has " + parameterTypes.l
ength);

                }

            } else if (strictMethodVerification && method.isAnnotationPresent(Subscri
be.class)) {
```

```
                //如果 method 的方法签名不符合要求且开启了严格的方法校验那么就抛出异常
                String methodName = method.getDeclaringClass().getName() + "." + metho
d.getName();
                throw new EventBusException(methodName +
                        " is a illegal @Subscribe method: must be public, non-static, a
nd non-abstract");
            }
        }
    }
```

复制代码

## SubscriberMethodFinder.FindState

`findUsingReflectionInSingleClass` 方法的一个重点是 `findState.checkAdd` 方法。如果往简单了想，只要把注册类每个声明了 Subscribe 注解的方法都给保存起来就可以了，可是还需要考虑一些特殊情况：

1. Java 中类是可以有继承关系的，如果父类声明了 Subscribe 方法，那么就相当于子类也持有了该监听方法，那么子类在 register 后就需要拿到父类的所有 Subscribe 方法
2. 如果子类继承并重写了父类的 Subscribe 方法，那么子类在 register 后就需要以自己重写后的方法为准，忽略父类的相应方法

`checkAdd` 方法就用于进行上述判断

```
        //以 eventType 作为 key，method 或者 FindState 作为 value
        final Map<Class, Object> anyMethodByEventType = new HashMap<>();
        //以 methodKey 作为 key，methodClass 作为 value
        final Map<String, Class> subscriberClassByMethodKey = new HashMap<>();
```

```
        boolean checkAdd(Method method, Class<?> eventType) {
            // 2 level check: 1st level with event type only (fast), 2nd level with co
mplete signature when required.
            // Usually a subscriber doesn't have methods listening to the same event t
ype.

            Object existing = anyMethodByEventType.put(eventType, method);
            if (existing == null) {
                //existing 等于 null 说明之前未解析到监听相同事件的方法，检查通过
                //因为大部分情况下监听者不会声明多个监听相同事件的方法，所以先进行这步检查效
率上会比较高
                return true;
            } else { //existing 不等于 null 说明之前已经解析到同样监听这个事件的方法了

                if (existing instanceof Method) {
                    if (!checkAddWithMethodSignature((Method) existing, eventType)) {
                        // Paranoia check
                        throw new IllegalStateException();
                    }
                    // Put any non-Method object to "consume" the existing Method
                    //会执行到这里，说明存在多个方法监听同个 Event，那么将将 eventType 对应
的 value 置为 this
                    //避免多次检查，让其直接去执行 checkAddWithMethodSignature 方法
                    anyMethodByEventType.put(eventType, this);
```

```
                }

                return checkAddWithMethodSignature(method, eventType);

            }

        }


        private boolean checkAddWithMethodSignature(Method method, Class<?> eventType)
 {

            methodKeyBuilder.setLength(0);

            methodKeyBuilder.append(method.getName());

            methodKeyBuilder.append('>').append(eventType.getName());


            //以 methodName>eventTypeName 字符串作为 key

            //通过这个 key 来判断是否存在子类重写了父类方法的情况

            String methodKey = methodKeyBuilder.toString();

            //获取声明了 method 的类对应的 class 对象

            Class<?> methodClass = method.getDeclaringClass();


            Class<?> methodClassOld = subscriberClassByMethodKey.put(methodKey, metho
dClass);

            //1. 如果 methodClassOld == null 为 true，说明 method 是第一次解析到，允许添
加

            //2. 如果 methodClassOld.isAssignableFrom(methodClass) 为 true

            //2.1、说明 methodClassOld 是 methodClass 的父类，需要以子类重写的方法 method
 为准，允许添加

            //     实际上应该不存在这种情况，因为 EventBus 是从子类开始向父类进行遍历的
```

```
            //2.2、说明 methodClassOld 是 methodClass 是同个类，即 methodClass 声明了多个
方法对同个事件进行监听 ， 也允许添加

            if (methodClassOld == null || methodClassOld.isAssignableFrom(methodClas
s)) {

                // Only add if not already found in a sub class

                return true;

            } else {

                // Revert the put, old class is further down the class hierarchy

                //由于 EventBus 是从子类向父类进行解析

                //会执行到这里就说明之前已经解析到了相同 key 的方法，对应子类重写了父类方法
的情况

                //此时需要以子类重写的方法 method 为准，所以又将 methodClassOld 重新设回去

                subscriberClassByMethodKey.put(methodKey, methodClassOld);

                return false;

            }

        }
```

复制代码

## EventBus.java

进行上述操作后，就拿到了注册类所有的包含了注解声明的方法了，这些方法都会保存到 `List<SubscriberMethod>` 中。拿到所有方法后，就需要对注册者及其所有监听方法进行归类了

归类的目的是既是为了方便后续操作也是为了提高效率。 因为在同个页面或者多个页面间可能存在多个对同种类型消息的监听方法，那么就需要将每种消息类型和其当前的所有监听方法对应起来，提高消息的发送效率。而且在 subscriber 解除注册时， 也需要将 subscriber 包含的所有监听方法都给移除掉，那么就需要预先进行归类。监听方法也可以设定自己对消息处理的优先级顺序，所以需要预先对监听方法进行排序

```
    public void register(Object subscriber) {

        Class<?> subscriberClass = subscriber.getClass();

        List<SubscriberMethod> subscriberMethods = subscriberMethodFinder.findSubscr
iberMethods(subscriberClass);

        synchronized (this) {

            for (SubscriberMethod subscriberMethod : subscriberMethods) {

                subscribe(subscriber, subscriberMethod);

            }

        }

    }



    private final Map<Class<?>, CopyOnWriteArrayList<Subscription>> subscriptionsByE
ventType;

    private final Map<Object, List<Class<?>>> typesBySubscriber;



    // Must be called in synchronized block

    private void subscribe(Object subscriber, SubscriberMethod subscriberMethod) {

        Class<?> eventType = subscriberMethod.eventType;

        Subscription newSubscription = new Subscription(subscriber, subscriberMetho
d);

        //subscriptionsByEventType 以消息类型 eventType 作为 key，value 存储了所有对该 e
ventType 的订阅者，提高后续在发送消息时的效率

        CopyOnWriteArrayList<Subscription> subscriptions = subscriptionsByEventType.
get(eventType);

        if (subscriptions == null) {
```

```
            subscriptions = new CopyOnWriteArrayList<>();
            subscriptionsByEventType.put(eventType, subscriptions);
        } else {
            if (subscriptions.contains(newSubscription)) {
                //说明某个 Subscriber 重复注册了
                throw new EventBusException("Subscriber " + subscriber.getClass() + "
already registered to event "
                        + eventType);
            }
        }

        //将订阅者根据消息优先级高低进行排序
        int size = subscriptions.size();
        for (int i = 0; i <= size; i++) {
            if (i == size || subscriberMethod.priority > subscriptions.get(i).subscri
berMethod.priority) {
                subscriptions.add(i, newSubscription);
                break;
            }
        }

        //typesBySubscriber 以订阅者 subscriber 作为 key，value 存储了其订阅的所有 event
Type
        //用于向外提供某个类是否已注册的功能，也方便后续在 unregister 时移除 subscriber 下
的所有监听方法
```

```
        List<Class<?>> subscribedEvents = typesBySubscriber.get(subscriber);
        if (subscribedEvents == null) {
            subscribedEvents = new ArrayList<>();
            typesBySubscriber.put(subscriber, subscribedEvents);
        }
        subscribedEvents.add(eventType);



        //下面是关于粘性事件的处理，后续再进行介绍
        if (subscriberMethod.sticky) {
            if (eventInheritance) {
                // Existing sticky events of all subclasses of eventType have to be co
nsidered.
                // Note: Iterating over all events may be inefficient with lots of sti
cky events,
                // thus data structure should be changed to allow a more efficient loo
kup
                // (e.g. an additional map storing sub classes of super classes: Class
 -> List<Class>).
                Set<Map.Entry<Class<?>, Object>> entries = stickyEvents.entrySet();
                for (Map.Entry<Class<?>, Object> entry : entries) {
                    Class<?> candidateEventType = entry.getKey();
                    if (eventType.isAssignableFrom(candidateEventType)) {
                        Object stickyEvent = entry.getValue();
                        checkPostStickyEventToSubscription(newSubscription, stickyEven
t);
```

```
                }

            }

        } else {

            Object stickyEvent = stickyEvents.get(eventType);

            checkPostStickyEventToSubscription(newSubscription, stickyEvent);

        }

    }

}
```

复制代码

# 二、发送消息

## 1、消息的执行策略

在介绍消息的具体发送步骤前，先来了解下 EventBus 几种不同的消息执行策略。执行策略由枚举 `ThreadMode` 来执行，在 `Subscribe` 注解中进行声明。执行策略决定了消息接收方是在哪一个线程接收到消息的

| ThreadMode | 执行线程 | |
|---|---|---|
| POSTING | 在发送事件的线程中执行 | 直接调用消息接收方 |
| MAIN | 在主线程中执行 | 如果事件就是在主线程发送的，则直接调用消息接收 mainThreadPoster 进行处理 |
| MAIN_ORDERED | 在主线程中按顺序执行 | 通过 mainThreadPoster 进行处理，以此保证消息处 |
| BACKGROUND | 在后台线程中按顺序执行 | 如果事件是在主线程发送的，则提交给 backgroundP 用消息接收方 |
| ASYNC | 提交给空闲的后台线程执行 | 将消息提交到 asyncPoster 进行处理 |

执行策略的具体细分逻辑是在 `postToSubscription` 方法完成的

```
   private void postToSubscription(Subscription subscription, Object event, boolean
isMainThread) {

       switch (subscription.subscriberMethod.threadMode) {

           case POSTING:

               invokeSubscriber(subscription, event);

               break;

           case MAIN:

               if (isMainThread) {

                   invokeSubscriber(subscription, event);

               } else {

                   mainThreadPoster.enqueue(subscription, event);

               }

               break;

           case MAIN_ORDERED:

               if (mainThreadPoster != null) {

                   mainThreadPoster.enqueue(subscription, event);

               } else {

                   // temporary: technically not correct as poster not decoupled from
subscriber

                   invokeSubscriber(subscription, event);

               }

               break;
```

```
            case BACKGROUND:
                if (isMainThread) {
                    backgroundPoster.enqueue(subscription, event);
                } else {
                    invokeSubscriber(subscription, event);
                }
                break;
            case ASYNC:
                asyncPoster.enqueue(subscription, event);
                break;
            default:
                throw new IllegalStateException("Unknown thread mode: " + subscription.
subscriberMethod.threadMode);
        }
    }


复制代码
```

例如，对于 `AsyncPoster` 来说，其每接收到一个消息，都会直接在 `enqueue` 方法中将自己（Runnable）提交给线程池进行处理，而使用的线程池默认是 `Executors.newCachedThreadPool()`，该线程池每接收到一个任务都会马上交由线程进行处理，所以 `AsyncPoster` 并不保证消息处理的有序性，但在消息处理的及时性方面会比较高，且每次提交给 `AsyncPoster` 的消息可能都是由不同的线程来处理

```
class AsyncPoster implements Runnable, Poster {
```

```
    private final PendingPostQueue queue;
    private final EventBus eventBus;

    AsyncPoster(EventBus eventBus) {
        this.eventBus = eventBus;
        queue = new PendingPostQueue();
    }

    public void enqueue(Subscription subscription, Object event) {
        PendingPost pendingPost = PendingPost.obtainPendingPost(subscription, event);
        queue.enqueue(pendingPost);
        eventBus.getExecutorService().execute(this);
    }

    @Override
    public void run() {
        PendingPost pendingPost = queue.poll();
        if(pendingPost == null) {
            throw new IllegalStateException("No pending post available");
        }
        eventBus.invokeSubscriber(pendingPost);
    }
```

```
}



复制代码
```

而 `BackgroundPoster` 只会在当前自己并没有正在处理消息的情况下才会将自己（Runnable）提交给线程池进行处理，所以 `BackgroundPoster` 会保证消息队列在处理时的有序性，但在消息处理的及时性方面相比 `AsyncPoster` 要低一些

```
final class BackgroundPoster implements Runnable, Poster {



    private final PendingPostQueue queue;

    private final EventBus eventBus;



    private volatile boolean executorRunning;



    BackgroundPoster(EventBus eventBus) {

        this.eventBus = eventBus;

        queue = new PendingPostQueue();

    }



    public void enqueue(Subscription subscription, Object event) {

        PendingPost pendingPost = PendingPost.obtainPendingPost(subscription, event);

        synchronized (this) {

            queue.enqueue(pendingPost);

            if (!executorRunning) {
```

```
                executorRunning = true;

                eventBus.getExecutorService().execute(this);

            }

        }

    }



    ...

}



复制代码
```

而不管是使用什么消息处理策略，最终都是通过调用以下方法来完成监听方法的反射调用

```
    void invokeSubscriber(PendingPost pendingPost) {

        Object event = pendingPost.event;

        Subscription subscription = pendingPost.subscription;

        PendingPost.releasePendingPost(pendingPost);

        if (subscription.active) {

            invokeSubscriber(subscription, event);

        }

    }



    void invokeSubscriber(Subscription subscription, Object event) {
```

```
        try {
            subscription.subscriberMethod.method.invoke(subscription.subscriber, even
t);
        } catch (InvocationTargetException e) {
            handleSubscriberException(subscription, event, e.getCause());
        } catch (IllegalAccessException e) {
            throw new IllegalStateException("Unexpected exception", e);
        }
    }
```

复制代码

## 2、发送非黏性消息

`EventBus.getDefault().post(Any)`方法用于发送非黏性消息。EventBus 会通过 `ThreadLocal` 为每个发送消息的线程维护一个 `PostingThreadState` 对象，用于为每个线程维护一个消息队列及其它辅助参数

```
    /**
     * For ThreadLocal, much faster to set (and get multiple values).
     */
    final static class PostingThreadState {
        final List<Object> eventQueue = new ArrayList<>();
        boolean isPosting;
        boolean isMainThread;
        Subscription subscription;
        Object event;
```

```
        boolean canceled;

    }



    private final ThreadLocal<PostingThreadState> currentPostingThreadState = new Th
readLocal<PostingThreadState>() {

        @Override

        protected PostingThreadState initialValue() {

            return new PostingThreadState();

        }

    };



    /**

     * Posts the given event to the event bus.

     */

    public void post(Object event) {

        PostingThreadState postingState = currentPostingThreadState.get();

        List<Object> eventQueue = postingState.eventQueue;

        //将消息添加到消息队列

        eventQueue.add(event);



        if (!postingState.isPosting) {

            //是否在主线程发送的消息

            postingState.isMainThread = isMainThread();
```

```
            //标记当前正在发送消息中
            postingState.isPosting = true;
            if (postingState.canceled) {
                throw new EventBusException("Internal error. Abort state was not reset
");
            }
            try {
                while (!eventQueue.isEmpty()) {
                    postSingleEvent(eventQueue.remove(0), postingState);
                }
            } finally {
                postingState.isPosting = false;
                postingState.isMainThread = false;
            }
        }
    }
```

复制代码

每次 post 进来的消息都会先存到消息队列 `eventQueue` 中，然后通过 while 循环进行处理，消息处理逻辑是通过 `postSingleEvent` 方法来完成的

其主要逻辑是：

1. 假设 EventA 继承于 EventB，那么当发送的消息类型是 EventA 时，就需要考虑 EventB 的监听方法是否可以接收到 EventA，即需要考虑消息类型是否具有继承关系
2. 具有继承关系。此时就需要拿到 EventA 的所有父类型，然后根据 EventA 本身和其父类型关联到的所有监听方法依次进行消息发送
3. 不具有继承关系。此时只需要向 EventA 的监听方法进行消息发送即可
4. 如果发送的消息最终没有找到任何接收者，且 `sendNoSubscriberEvent` 为 true，那么就主动发送一个 `NoSubscriberEvent` 事件，用于向外通知消息没有找到任何接收者
5. 监听方法之间可以设定消息处理的优先级高低，高优先级的方法可以通过调用 `cancelEventDelivery` 方法来拦截事件，不再继续向下发送。但只有在 `POSTING` 模式下才能拦截事件，因为只有在这个模式下才能保证监听方法是按照严格的先后顺序被执行的

最终，发送的消息都会通过 `postToSubscription` 方法来完成，根据接收者方法不同的处理策略进行处理

```
    private void postSingleEvent(Object event, PostingThreadState postingState) thro
ws Error {

        Class<?> eventClass = event.getClass();

        //用于标记是否有找到消息的接收者

        boolean subscriptionFound = false;

        if (eventInheritance) {

            //步骤2

            List<Class<?>> eventTypes = lookupAllEventTypes(eventClass);

            int countTypes = eventTypes.size();

            for (int h = 0; h < countTypes; h++) {

                Class<?> clazz = eventTypes.get(h);
```

```
                subscriptionFound |= postSingleEventForEventType(event, postingState,
 clazz);

            }

        } else {

            //步骤3

            subscriptionFound = postSingleEventForEventType(event, postingState, even
tClass);

        }

        if (!subscriptionFound) {

            if (logNoSubscriberMessages) {

                logger.log(Level.FINE, "No subscribers registered for event " + eventC
lass);

            }

            if (sendNoSubscriberEvent && eventClass != NoSubscriberEvent.class &&

                    eventClass != SubscriberExceptionEvent.class) {

                //步骤4

                post(new NoSubscriberEvent(this, event));

            }

        }

    }



    private boolean postSingleEventForEventType(Object event, PostingThreadState pos
tingState, Class<?> eventClass) {

        CopyOnWriteArrayList<Subscription> subscriptions;

        synchronized (this) {
```

```
            //找到所有监听者
            subscriptions = subscriptionsByEventType.get(eventClass);
        }
        if (subscriptions != null && !subscriptions.isEmpty()) {
            for (Subscription subscription : subscriptions) {
                postingState.event = event;
                postingState.subscription = subscription;
                boolean aborted;
                try {
                    postToSubscription(subscription, event, postingState.isMainThrea
d);
                    aborted = postingState.canceled;
                } finally {
                    postingState.event = null;
                    postingState.subscription = null;
                    postingState.canceled = false;
                }
                //步骤5
                if (aborted) {
                    break;
                }
            }
            return true;
```

```
        }

        return false;

    }
```

复制代码

## 3、发送黏性消息

黏性消息的意义是为了使得在消息发出来后，即使是后续再进行 `register` 的 `subscriber` 也可以收到之前发送的消息，这需要 `@Subscribe` 注解的 `sticky` 属性设为 true，即表明消息接收方希望接收黏性消息

`EventBus.getDefault().postSticky(Any)`方法就用于发送黏性消息。黏性事件会被保存到 `stickyEvents` 这个 Map 中，key 是 event 的 Class 对象，value 是 event 本身，这也说明对于同一类型的黏性消息来说，只会保存其最后一个消息

```
    private final Map<Class<?>, Object> stickyEvents;



    /**

     * Posts the given event to the event bus and holds on to the event (because it i
s sticky). The most recent sticky

     * event of an event's type is kept in memory for future access by subscribers us
ing {@link Subscribe#sticky()}.

     */

    public void postSticky(Object event) {

        synchronized (stickyEvents) {

            stickyEvents.put(event.getClass(), event);

        }
```

```
        // Should be posted after it is putted, in case the subscriber wants to remov
e immediately

        post(event);

    }
```

复制代码

对于一个黏性消息，会有两种不同的时机被 subscriber 接收到

1. 调用 postSticky 方法时，被其现有的 subscriber 直接接收到，这种方式通过在 postSticky 方法里调用 post 方法来实现
2. 调用 register 方法时，新添加的 subscriber 会判断 stickyEvents 中是否存在关联的 event 需要进行分发

这里主要看第二种情况。register 操作会在 subscribe 方法里完成黏性事件的分发。和 post 操作一样，发送黏性事件时也需要考虑 event 的继承关系

```
    private void subscribe(Object subscriber, SubscriberMethod subscriberMethod) {



        ...



        if (subscriberMethod.sticky) {

            if (eventInheritance) {

                // Existing sticky events of all subclasses of eventType have to be co
nsidered.

                // Note: Iterating over all events may be inefficient with lots of sti
cky events,

                // thus data structure should be changed to allow a more efficient loo
kup
```

```
            // (e.g. an additional map storing sub classes of super classes: Class
 -> List<Class>).



            //事件类型需要考虑其继承关系

            //因此需要判断每一个 stickyEvent 的父类型是否存在监听者，有的话就需要都进行
回调

            Set<Map.Entry<Class<?>, Object>> entries = stickyEvents.entrySet();

            for (Map.Entry<Class<?>, Object> entry : entries) {

                Class<?> candidateEventType = entry.getKey();

                if (eventType.isAssignableFrom(candidateEventType)) {

                    Object stickyEvent = entry.getValue();

                    checkPostStickyEventToSubscription(newSubscription, stickyEven
t);

                }

            }

        } else {

            //事件类型不需要考虑其继承关系

            Object stickyEvent = stickyEvents.get(eventType);

            checkPostStickyEventToSubscription(newSubscription, stickyEvent);

        }

    }

}



    private void checkPostStickyEventToSubscription(Subscription newSubscription, Ob
ject stickyEvent) {
```

```
        if (stickyEvent != null) {
            // If the subscriber is trying to abort the event, it will fail (event is not tracked in posting state)
            // --> Strange corner case, which we don't take care of here.
            postToSubscription(newSubscription, stickyEvent, isMainThread());
        }
    }
```

复制代码

## 4、移除黏性事件

移除指定的黏性事件可以通过以下方法来实现，都是用于将指定事件从 `stickyEvents` 中移除

```
    /**
     * Remove and gets the recent sticky event for the given event type.
     *
     * @see #postSticky(Object)
     */
    public <T> T removeStickyEvent(Class<T> eventType) {
        synchronized (stickyEvents) {
            return eventType.cast(stickyEvents.remove(eventType));
        }
    }

    /**
```

```
     * Removes the sticky event if it equals to the given event.
     *
     * @return true if the events matched and the sticky event was removed.
     */
    public boolean removeStickyEvent(Object event) {
        synchronized (stickyEvents) {
            Class<?> eventType = event.getClass();
            Object existingEvent = stickyEvents.get(eventType);
            if (event.equals(existingEvent)) {
                stickyEvents.remove(eventType);
                return true;
            } else {
                return false;
            }
        }
    }
```

复制代码

## 三、解除注册

解除注册的目的是为了避免内存泄露，EventBus 使用了单例模式，如果不主动解除注册的话，EventBus 就会一直持有注册对象。解除注册的操作是通过 `unregister` 方法来实现的，该方法逻辑也比较简单，只是将 subscriber 以及其关联的所有 method 对象从集合中移除而已

而此处虽然会将关于 subscriber 的信息均给移除掉，但是在 SubscriberMethodFinder 中的静态成员变量 METHOD_CACHE 依然会缓存着已经注册过的 subscriber 的信息，这也是为了在某些页面会先后多次注册 EventBus 时可以做到信息复用，避免多次循环反射

```
    /**
     * Unregisters the given subscriber from all event classes.
     */
    public synchronized void unregister(Object subscriber) {
        List<Class<?>> subscribedTypes = typesBySubscriber.get(subscriber);
        if (subscribedTypes != null) {
            for (Class<?> eventType : subscribedTypes) {
                unsubscribeByEventType(subscriber, eventType);
            }
            typesBySubscriber.remove(subscriber);
        } else {
            logger.log(Level.WARNING, "Subscriber to unregister was not registered before: " + subscriber.getClass());
        }
    }


    /**
     * Only updates subscriptionsByEventType, not typesBySubscriber! Caller must update typesBySubscriber.
     */
    private void unsubscribeByEventType(Object subscriber, Class<?> eventType) {
```

```
        List<Subscription> subscriptions = subscriptionsByEventType.get(eventType);
        if (subscriptions != null) {
            int size = subscriptions.size();
            for (int i = 0; i < size; i++) {
                Subscription subscription = subscriptions.get(i);
                if (subscription.subscriber == subscriber) {
                    subscription.active = false;
                    subscriptions.remove(i);
                    i--;
                    size--;
                }
            }
        }
    }
```

复制代码

## 四、注解处理器

使用注解处理器可以避免 subscriber 进行注册时的多次循环反射操作，极大提升了 EventBus 的运行效率

APT(**Annotation Processing Tool**) 即注解处理器，是一种注解处理工具，用来在编译期扫描和处理注解，通过注解来生成 Java 文件。即以注解作为桥梁，通过预先规定好的代码生成规则来自动生成 Java 文件。此类注解框架的代表有 **ButterKnife**、**Dragger2**、**EventBus** 等

Java API 已经提供了扫描源码并解析注解的框架，开发者可以通过继承 **AbstractProcessor** 类来实现自己的注解解析逻辑。APT 的原理就是在注解了某些代码元素（如字段、函数、类等）后，在编译时编译器会检

查 **AbstractProcessor** 的子类，并且自动调用其 **process()** 方法，然后将添加了指定注解的所有代码元素作为参数传递给该方法，开发者再根据注解元素在编译期输出对应的 Java 代码

关于 APT 技术的原理和应用可以看这篇文章：Android APT 实例讲解

在 Kotlin 环境引入注解处理器的方法如下所示：

```
apply plugin: 'kotlin-kapt'



kapt {

    arguments {

        arg('eventBusIndex', 'github.leavesc.demo.MyEventBusIndex')

    }

}



dependencies {

    implementation "org.greenrobot:eventbus:3.2.0"

    kapt "org.greenrobot:eventbus-annotation-processor:3.2.0"

}

复制代码
```

当中，`github.leavesc.demo.MyEventBusIndex` 就是生成的辅助文件的包名路径，可以由我们自己定义

原始文件：

```
/**

 * 作者：leavesC
```

```
 * 时间：2020/10/01 12:17

 * 描述：

 * GitHub：https://github.com/leavesC

 */

class MainActivity : AppCompatActivity() {



    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity_main)

    }



    @Subscribe

    fun fun1(msg: String) {



    }



    @Subscribe(threadMode = ThreadMode.MAIN, priority = 100)

    fun fun2(msg: String) {



    }



}
```

复制代码

生成的辅助文件如下所示。可以看出，MyEventBusIndex 文件中封装了 subscriber 和其所有监听方法的签名信息，这样我们就无需在运行时再来进行解析了，而是直接在编译阶段就拿到了

```
/** This class is generated by EventBus, do not edit. */public class MyEventBusIndex implements SubscriberInfoIndex {

    private static final Map<Class<?>, SubscriberInfo> SUBSCRIBER_INDEX;



    static {

        SUBSCRIBER_INDEX = new HashMap<Class<?>, SubscriberInfo>();



        putIndex(new SimpleSubscriberInfo(MainActivity.class, true, new SubscriberMethodInfo[] {

            new SubscriberMethodInfo("fun1", String.class),

            new SubscriberMethodInfo("fun2", String.class, ThreadMode.MAIN, 100, false),

        }));



    }



    private static void putIndex(SubscriberInfo info) {

        SUBSCRIBER_INDEX.put(info.getSubscriberClass(), info);

    }
```

```
    @Override
    public SubscriberInfo getSubscriberInfo(Class<?> subscriberClass) {
        SubscriberInfo info = SUBSCRIBER_INDEX.get(subscriberClass);
        if (info != null) {
            return info;
        } else {
            return null;
        }
    }
}
复制代码
```

需要注意的是，在生成了辅助文件后，还需要通过这些类文件来初始化EventBus

```
EventBus.builder().addIndex(MyEventBusIndex()).installDefaultEventBus()

复制代码
```

注入的辅助文件会被保存到 `SubscriberMethodFinder` 类的成员变量 `subscriberInfoIndexes` 中，`findUsingInfo` 方法会先尝试从辅助文件中获取SubscriberMethod，只有在获取不到的时候才会通过性能较低的反射操作来完成

```
    private List<SubscriberMethod> findUsingInfo(Class<?> subscriberClass) {
        FindState findState = prepareFindState();
        findState.initForSubscriber(subscriberClass);
        while (findState.clazz != null) {
```

```
            //在没有使用注解处理器的情况下，findState.subscriberInfo 和 subscriberInfoIn
dexes 的默认值都是为 null，所以 getSubscriberInfo 会返回 null

            //此时就需要通过 findUsingReflectionInSingleClass 方法来进行反射获取



            //而在有使用注解处理器的情况下，subscriberInfoIndexes 就存储了自动生成的辅助文
件，此时 getSubscriberInfo 就可以从辅助文件中拿到目标信息

            //从而避免了反射操作



            findState.subscriberInfo = getSubscriberInfo(findState);

            if (findState.subscriberInfo != null) {

                SubscriberMethod[] array = findState.subscriberInfo.getSubscriberMeth
ods();

                for (SubscriberMethod subscriberMethod : array) {

                    if (findState.checkAdd(subscriberMethod.method, subscriberMethod.
eventType)) {

                        findState.subscriberMethods.add(subscriberMethod);

                    }

                }

            } else {

                findUsingReflectionInSingleClass(findState);

            }

            findState.moveToSuperclass();

        }

        return getMethodsAndRelease(findState);

    }
```

```
    private SubscriberInfo getSubscriberInfo(FindState findState) {
        if (findState.subscriberInfo != null && findState.subscriberInfo.getSuperSubscriberInfo() != null) {
            SubscriberInfo superclassInfo = findState.subscriberInfo.getSuperSubscriberInfo();
            if (findState.clazz == superclassInfo.getSubscriberClass()) {
                return superclassInfo;
            }
        }
        if (subscriberInfoIndexes != null) {
            for (SubscriberInfoIndex index : subscriberInfoIndexes) {
                SubscriberInfo info = index.getSubscriberInfo(findState.clazz);
                if (info != null) {
                    return info;
                }
            }
        }
        return null;
    }
```

复制代码

## 五、一些坑

### 1、奇怪的继承关系

上文有介绍到，子类可以继承父类的 Subscribe 方法。但有一个比较奇怪的地方是：如果子类重写了父类多个 Subscribe 方法的话，就会抛出 IllegalStateException。例如，在下面的例子中。父类 BaseActivity 声明了两个 Subscribe 方法，子类 MainActivity 重写了这两个方法，此时运行后就会抛出 IllegalStateException。而如果 MainActivity 不重写或者只重写一个方法的话，就可以正常运行

```
/**

 * 作者：leavesC

 * 时间：2020/10/01 12:49

 * 描述：

 * GitHub：https://github.com/leavesC

 */

open class BaseActivity : AppCompatActivity() {



    @Subscribe

    open fun fun1(msg: String) {



    }



    @Subscribe

    open fun fun2(msg: String) {



    }



}
```

```
class MainActivity : BaseActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        EventBus.getDefault().register(this)
    }

    override fun onDestroy() {
        super.onDestroy()
        EventBus.getDefault().unregister(this)
    }

    @Subscribe
    override fun fun1(msg: String) {

    }

    @Subscribe
    override fun fun2(msg: String) {

    }
```

```
}
```

复制代码

按道理来说，如果子类重写了父类一个 Subscribe 方法都可以正常使用的话，那么重写两个也应该可以正常使用才对。可是上述例子就表现得 EventBus 好像有 bug 似的。通过定位堆栈信息，可以发现是在 `FindState` 的 `checkAdd` 方法抛出了异常

其抛出异常的步骤是这样的：

1. EventBus 对 Subscribe 方法的解析方向是子类向父类进行的，同个类下的 Subscribe 方法按照声明顺序进行解析
2. 当 `checkAdd` 方法开始解析 `BaseActivity` 的 `fun2` 方法时，`existing` 对象就是 `BaseActivity.fun1`，此时就会执行到操作 1，而由于子类已经重写了 `fun1` 方法，此时 `checkAddWithMethodSignature` 方法就会返回 false，最终导致抛出异常

```
        boolean checkAdd(Method method, Class<?> eventType) {

            // 2 level check: 1st level with event type only (fast), 2nd level with co
mplete signature when required.

            // Usually a subscriber doesn't have methods listening to the same event t
ype.

            Object existing = anyMethodByEventType.put(eventType, method);

            if (existing == null) {

                return true;

            } else {

                if (existing instanceof Method) {

                    //操作1

                    if (!checkAddWithMethodSignature((Method) existing, eventType)) {
```

```
                    // Paranoia check
                    throw new IllegalStateException();
                }
                // Put any non-Method object to "consume" the existing Method
                anyMethodByEventType.put(eventType, this);
            }
            return checkAddWithMethodSignature(method, eventType);
        }
    }
```

复制代码

EventBus 中有一个 issues 也反馈了这个问题：issues，该问题在 2018 年时就已经存在了，EeventBus 的作者也只是回复说：**只在子类进行方法监听**

## 2、移除黏性消息

`removeStickyEvent` 方法会有一个比较让人误解的点：对于通过 `EventBus.getDefault().postSticky(XXX)`方法发送的黏性消息无法通过 `removeStickyEvent` 方法来使现有的监听者拦截该事件

例如，假设下面的两个方法都已经处于注册状态了，postSticky 后，即使在 fun1 方法中移除了黏性消息，fun2 方法也可以接收到消息。这是因为 `postSticky` 方法最终也是要靠调用 post 方法来完成消息发送，而 post 方法并不受 `stickyEvents` 的影响

```
@Subscribe(sticky = true)
fun fun1(msg: String) {
    EventBus.getDefault().removeStickyEvent(msg)
}
```

```
    @Subscribe(sticky = true)
    fun fun2(msg: String) {


    }
```
复制代码

而如果 EventBus 中已经存储了黏性事件，那么在上述两个方法刚 register 时，fun1 方法就可以拦截住消息使 fun2 方法接收不到消息。这是因为 register 方法是在 for 循环中遍历 method，如果之前的方法已经移除了黏性消息的话，那么后续方法就没有黏性消息需要处理了

```
    public void register(Object subscriber) {
        Class<?> subscriberClass = subscriber.getClass();
        List<SubscriberMethod> subscriberMethods = subscriberMethodFinder.findSubscr
iberMethods(subscriberClass);
        synchronized (this) {
            //在 for 循环中遍历 method
            for (SubscriberMethod subscriberMethod : subscriberMethods) {
                subscribe(subscriber, subscriberMethod);
            }
        }
    }
```
复制代码

## 六、总结

EventBus 的源码解析到这里就结束了，本文所讲的内容应该也已经涵盖了大部分内容了。这里再来为 EventBus 的实现流程做一个总结

1. EventBus 包含 register 和 unregister 方法用于标记当前 subscriber 是否需要接收消息，内部对应**向 CopyOnWriteArrayList 添加和移除元素**这两个操作
2. 每当有 event 被 post 出来时，就需要根据 eventClass 对象找到所有所有声明了 @Subscribe 注解且对这种消息类型进行监听的方法，这些方法都是在 subscriber 进行 register 的时候，从 subscriber 中获取到的
3. 从 subscriber 中获取所有声明了 @Subscribe 注解的方法有两种。第一种是通过反射的方式拿到 subscriber 这个类中包含的所有声明了 @Subscribe 注解的方法，对应的是没有配置注解处理器的情况。第二种对应的是有配置注解处理器的情况，通过在编译阶段全局扫描 @Subscribe 注解并生成辅助文件，从而在 register 的时候省去了效率低下的反射操作。不管是通过什么方式进行获取，拿到所有方法后都会将 methods 按照消息类型 eventType 进行归类，方便后续遍历
4. 每当有消息被发送出来时，就根据 event 对应的 Class 对象找到相应的监听方法，然后通过反射的方式来回调方法。外部可以在初始化 EventBus 的时候选择是否要考虑 event 的继承关系，即在 event 被 Post 出来时，对 event 的父类型进行监听的方法是否需要被回调

EventBus 的实现思路并不算多难，难的是在实现的时候可以方方面面都考虑周全，做到稳定高效，从 2018 年到现在 2020 年也才发布了两个版本（也许是作者懒得更新？）。原理懂了，那么下一篇就进入实战篇，自己来动手实现一个 EventBus

# 三方库源码笔记（2）-EventBus 自己实现一个？

上一篇文章中对 EventBus 的源码进行了一次全面解析，原理懂得了，那么就也需要进行一次实战才行。对于一个优秀的第三方库，开发者除了要学会如何使用外，更有难度的用法就是去了解实现原理、懂得如何改造甚至自己实现。本篇文章就来自己动手实现一个 EventBus，不求功能多齐全，就来实现简单的**注册、反注册、发送消息、接收消息**这些功能即可

先来看下最终的实现效果

对于以下两个监听者：`EasyEventBusActivity` 和 `EasyBusEventTest`，通过标注 `@Event` 注解来修饰监听方法，然后使用 `EasyEventBus` 这个自定义类来进行**注册、反注册和发送消息**

```
/**

 * 作者：LeavesC

 * 时间：2020/10/2 13:14

 * 描述：

 * GitHub：https://github.com/LeavesC

 */class EasyEventBusActivity : BaseActivity() {



    private val eventTest = EasyBusEventTest()



    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity_easy_event_bus)

        EasyEventBus.register(this)

        eventTest.register()



        btn_postString.setOnClickListener {

            EasyEventBus.post("Hello")

        }

        btn_postBean.setOnClickListener {

            EasyEventBus.post(HelloBean("hi"))
```

```
        }
    }

    @Event
    fun stringFun(msg: String) {
        showToast("$msg EasyEventBusActivity")
    }

    @Event
    fun benFun(msg: HelloBean) {
        showToast("${msg.data} EasyEventBusActivity")
    }

    override fun onDestroy() {
        super.onDestroy()
        EasyEventBus.unregister(this)
        eventTest.unregister()
    }

}
class EasyBusEventTest {

    @Event
```

```
    fun stringFun(msg: String) {

        showToast("$msg EasyBusEventTest")

    }



    @Event

    fun benFun(msg: HelloBean) {

        showToast("${msg.data} EasyBusEventTest")

    }



    fun register() {

        EasyEventBus.register(this)

    }



    fun unregister() {

        EasyEventBus.unregister(this)

    }



}

data class HelloBean(val data: String)
```

复制代码

使用起来和真正的 EvnetBus 差不多

![img](data:image/svg+xml;utf8,<?xml version="1.0"?><svg xmlns="http://www.w3.org/2000/svg" version="1.1" width="800" height="600"></svg>)

最终自定义的 EasyEventBus 也只有五十行左右的代码量，仅向外部提供了三个方法用于进行**注册、反注册和发送消息**

```
/**

 * 作者：leavesC

 * 时间：2020/10/3 11:44

 * 描述：

 * GitHub：https://github.com/leavesC

 */object EasyEventBus {



    private val subscriptions = mutableSetOf<Any>()



    private const val PACKAGE_NAME = "github.leavesc.easyeventbus"



    private const val CLASS_NAME = "EventBusInject"



    private const val CLASS_PATH = "$PACKAGE_NAME.$CLASS_NAME"



    private val clazz = Class.forName(CLASS_PATH)



    //通过反射生成 EventBusInject 对象

    private val instance = clazz.newInstance()
```

```
@Synchronized
fun register(subscriber: Any) {
    subscriptions.add(subscriber)
}

@Synchronized
fun unregister(subscriber: Any) {
    subscriptions.remove(subscriber)
}

@Synchronized
fun post(event: Any) {
    subscriptions.forEach { subscriber ->
        val subscriberInfo =
            getSubscriberInfo(subscriber.javaClass)
        if (subscriberInfo != null) {
            val methodList = subscriberInfo.methodList
            methodList.forEach { method ->
                if (method.eventType == event.javaClass) {
                    val declaredMethod = subscriber.javaClass.getDeclaredMethod(
                        method.methodName,
                        method.eventType
```

```
                    )
                    declaredMethod.invoke(subscriber, event)
                }
            }
        }
    }

    //通过反射调用 EventBusInject 的 getSubscriberInfo 方法
    private fun getSubscriberInfo(subscriberClass: Class<*>): SubscriberInfo? {
        val method = clazz.getMethod("getSubscriberInfo", Class::class.java)
        return method.invoke(instance, subscriberClass) as? SubscriberInfo
    }

}
```

复制代码

# 一、需要做什么

这里先来想下这个自定义的 EasyEventBus 应该实现什么功能，以及怎么实现

EasyEventBus 的核心重点就在于其通过**注解处理器**生成辅助文件这个过程，这个过程使用者是感知不到的，这块逻辑也只会在编译阶段被触发到。我们希望在编译阶段就能够拿到所有声明了 `@Event` 的方法，免得在运行时才来反射。即在编译阶段就希望能够生成以下的辅助文件：

```
/**
```

```
 * 这是自动生成的代码 by leavesC

 */public class EventBusInject {

    private static final Map<Class<?>, SubscriberInfo> subscriberIndex = new HashMap<
Class<?>, SubscriberInfo>();



    {

        List<EventMethodInfo> eventMethodInfoList = new ArrayList<EventMethodInfo>();

        eventMethodInfoList.add(new EventMethodInfo("stringFun", String.class));

        eventMethodInfoList.add(new EventMethodInfo("benFun", HelloBean.class));

        SubscriberInfo subscriberInfo = new SubscriberInfo(EasyBusEventTest.class, ev
entMethodInfoList);

        putIndex(subscriberInfo);

    }



    {

        List<EventMethodInfo> eventMethodInfoList = new ArrayList<EventMethodInfo>();

        eventMethodInfoList.add(new EventMethodInfo("stringFun", String.class));

        eventMethodInfoList.add(new EventMethodInfo("benFun", HelloBean.class));

        SubscriberInfo subscriberInfo = new SubscriberInfo(EasyEventBusActivity.class,
 eventMethodInfoList);

        putIndex(subscriberInfo);

    }



    private static final void putIndex(SubscriberInfo info) {
```

```
        subscriberIndex.put(info.getSubscriberClass(), info);
    }

    public final SubscriberInfo getSubscriberInfo(Class<?> subscriberClass) {
        return subscriberIndex.get(subscriberClass);
    }
}
```

复制代码

可以看到，`subscriberIndex`中存储了所有的监听方法的签名信息，在应用运行时我们我们只需要通过 `getSubscriberInfo` 方法就可以拿到 `subscriberClass` 的所有监听方法

最后，还需要向外提供一个 API 调用入口，即上面贴出来的自定义的 EasyEventBus 这个自定义类，是提供给使用者运行时调用的，在有消息需要发送的时候通过外部传入的 `subscriberClass` 从 `EventBusInject` 取出所有监听方法进行反射回调

所以，EasyEventBus 逻辑上会拆分为两个 moudle：

- event-api。向外暴露 API 调用入口
- evnet-processor。不对外暴露，只在编译阶段生效

## 二、注解处理器

首先，我们需要提供一个注解对监听方法进行标记

```
@MustBeDocumented@kotlin.annotation.Retention(AnnotationRetention.SOURCE)@Target(An
notationTarget.FUNCTION)annotation class Event
```

复制代码

然后，我们在编译阶段需要预先把所有监听方法抽象保存起来，所以就需要定义两个 JavaBean 来作为承载体

```
/**

 * 作者：LeavesC

 * 时间：2020/10/3 17:33

 * 描述：

 * GitHub：https://github.com/LeavesC

 */data class EventMethodInfo(val methodName: String, val eventType: Class<*>)

data class SubscriberInfo(

    val subscriberClass: Class<*>,

    val methodList: List<EventMethodInfo>

)
```

复制代码

然后声明一个 `EasyEventBusProcessor` 类继承于 `AbstractProcessor`，由编译器在编译阶段传入我们关心的代码元素

```
/**

 * 作者：LeavesC

 * 时间：2020/10/3 15:55

 * 描述：

 * GitHub：https://github.com/LeavesC

 */class EasyEventBusProcessor : AbstractProcessor() {



    companion object {
```

```
        private const val PACKAGE_NAME = "github.leavesc.easyeventbus"

        private const val CLASS_NAME = "EventBusInject"

        private const val DOC = "这是自动生成的代码 by leavesC"

    }

    private lateinit var elementUtils: Elements

    private val methodsByClass = LinkedHashMap<TypeElement, MutableList<ExecutableEl
ement>>()

    override fun init(processingEnvironment: ProcessingEnvironment) {
        super.init(processingEnvironment)
        elementUtils = processingEnv.elementUtils
    }

    override fun getSupportedAnnotationTypes(): MutableSet<String> {
        //只需要处理 Event 注解
        return mutableSetOf(Event::class.java.canonicalName)
    }
```

```
    override fun getSupportedSourceVersion(): SourceVersion {
        return SourceVersion.RELEASE_8
    }

    ...

}
```

复制代码

通过 `collectSubscribers` 方法拿到所有的监听方法，保存到 `methodsByClass` 中，同时需要对方法签名进行校验：**只能是实例方法，且必须是 public 的，最多且只能包含一个入参参数**

```
override fun process(
        set: Set<TypeElement>,
        roundEnvironment: RoundEnvironment
    ): Boolean {
        val messager = processingEnv.messager
        collectSubscribers(set, roundEnvironment, messager)
        if (methodsByClass.isEmpty()) {
            messager.printMessage(Diagnostic.Kind.WARNING, "No @Event annotations fou
nd")
        } else {
```

```
        ...



    }

    return true

}



private fun collectSubscribers(

    annotations: Set<TypeElement>,

    env: RoundEnvironment,

    messager: Messager

) {

    for (annotation in annotations) {

        val elements = env.getElementsAnnotatedWith(annotation)

        for (element in elements) {

            if (element is ExecutableElement) {

                if (checkHasNoErrors(element, messager)) {

                    val classElement = element.enclosingElement as TypeElement

                    var list = methodsByClass[classElement]

                    if (list == null) {

                        list = mutableListOf()

                        methodsByClass[classElement] = list

                    }

                    list.add(element)
```

```
                }
            } else {
                //@Event 只能用于修改方法
                messager.printMessage(
                    Diagnostic.Kind.ERROR,
                    "@Event is only valid for methods",
                    element
                )
            }
        }
    }
}

/**
 * 校验方法签名是否合法
 */
private fun checkHasNoErrors(element: ExecutableElement, messager: Messager): Boolean {
    //不能是静态方法
    if (element.modifiers.contains(Modifier.STATIC)) {
        messager.printMessage(Diagnostic.Kind.ERROR, "Event method must not be static", element)
        return false
    }
```

```
        //必须是 public 方法
        if (!element.modifiers.contains(Modifier.PUBLIC)) {
            messager.printMessage(Diagnostic.Kind.ERROR, "Event method must be public
", element)
            return false
        }
        //方法最多且只能包含一个参数
        val parameters = element.parameters
        if (parameters.size != 1) {
            messager.printMessage(
                Diagnostic.Kind.ERROR,
                "Event method must have exactly 1 parameter",
                element
            )
            return false
        }
        return true
    }
复制代码
```

然后，再来生成 `subscriberIndex` 这个静态常量，以及对应的静态方法块、`putIndex` 方法

```
    //生成 subscriberIndex 这个静态常量
    private fun generateSubscriberField(): FieldSpec {
```

```
    val subscriberIndex = ParameterizedTypeName.get(
        ClassName.get(Map::class.java),
        getClassAny(),
        ClassName.get(SubscriberInfo::class.java)
    )
    return FieldSpec.builder(subscriberIndex, "subscriberIndex")
        .addModifiers(
            Modifier.PRIVATE,
            Modifier.STATIC,
            Modifier.FINAL
        )
        .initializer(
            "new ${"$"}T<Class<?>, ${"$"}T>()",
            HashMap::class.java,
            SubscriberInfo::class.java
        )
        .build()
}

//生成静态方法块
private fun generateInitializerBlock(builder: TypeSpec.Builder) {
    for (item in methodsByClass) {
        val methods = item.value
```

```
            if (methods.isEmpty()) {
                break
            }
            val codeBuilder = CodeBlock.builder()
            codeBuilder.add(
                "${"$"}T<${"$"}T> eventMethodInfoList = new ${"$"}T<${"$"}T>();",
                List::class.java,
                EventMethodInfo::class.java,
                ArrayList::class.java,
                EventMethodInfo::class.java
            )
            methods.forEach {
                val methodName = it.simpleName.toString()
                val eventType = it.parameters[0].asType()
                codeBuilder.add(
                    "eventMethodInfoList.add(new EventMethodInfo(${"$"}S, ${"$"}T.cla
ss));",
                    methodName,
                    eventType
                )
            }
            codeBuilder.add(
                "SubscriberInfo subscriberInfo = new SubscriberInfo(${"$"}T.class, ev
entMethodInfoList); putIndex(subscriberInfo);",
```

```
                item.key.asType()

            )

            builder.addInitializerBlock(

                codeBuilder.build()

            )

        }

    }



    //生成 putIndex 方法

    private fun generateMethodPutIndex(): MethodSpec {

        return MethodSpec.methodBuilder("putIndex")

            .addModifiers(Modifier.PRIVATE, Modifier.STATIC, Modifier.FINAL)

            .returns(Void.TYPE)

            .addParameter(SubscriberInfo::class.java, "info")

            .addCode(

                CodeBlock.builder().add("subscriberIndex.put(info.getSubscriberClass()
, info);")

                    .build()

            )

            .build()

    }
```

复制代码

然后，再来生成 `getSubscriberInfo` 这个公开方法，用于运行时调用

```
    //生成 getSubscriberInfo 方法
    private fun generateMethodGetSubscriberInfo(): MethodSpec {
        return MethodSpec.methodBuilder("getSubscriberInfo")
            .addModifiers(Modifier.PUBLIC, Modifier.FINAL)
            .returns(SubscriberInfo::class.java)
            .addParameter(getClassAny(), "subscriberClass")
            .addCode(
                CodeBlock.builder().add("return subscriberIndex.get(subscriberClass);
")
                    .build()
            )
            .build()
    }
```

复制代码

完成以上方法的定义后，就可以在 `process` 完成 `EventBusInject` 整个类文件的构建了

```
    override fun process(
        set: Set<TypeElement>,
        roundEnvironment: RoundEnvironment
    ): Boolean {
        val messager = processingEnv.messager
        collectSubscribers(set, roundEnvironment, messager)
        if (methodsByClass.isEmpty()) {
```

```
            messager.printMessage(Diagnostic.Kind.WARNING, "No @Event annotations fou
nd")

        } else {

            val typeSpec = TypeSpec.classBuilder(CLASS_NAME)

                .addModifiers(Modifier.PUBLIC)

                .addJavadoc(DOC)

                .addField(generateSubscriberField())

                .addMethod(generateMethodPutIndex())

                .addMethod(generateMethodGetSubscriberInfo())

            generateInitializerBlock(typeSpec)

            val javaFile = JavaFile.builder(PACKAGE_NAME, typeSpec.build())

                .build()

            try {

                javaFile.writeTo(processingEnv.filer)

            } catch (e: Throwable) {

                e.printStackTrace()

            }

        }

        return true

    }
```

复制代码

## 三、EasyEventBus

EasyEventBus 的逻辑就很简单了，主要是通过反射来生成 `EventBusInject` 对象，拿到 `subscriber` 关联的 `SubscriberInfo`，然后在有消息被 Post 出来的时候进行遍历调用即可

```
/**

 * 作者：LeavesC

 * 时间：2020/10/3 11:44

 * 描述：

 * GitHub：https://github.com/LeavesC

 */object EasyEventBus {



    private val subscriptions = mutableSetOf<Any>()



    private const val PACKAGE_NAME = "github.leavesc.easyeventbus"



    private const val CLASS_NAME = "EventBusInject"



    private const val CLASS_PATH = "$PACKAGE_NAME.$CLASS_NAME"



    private val clazz = Class.forName(CLASS_PATH)



    //通过反射生成 EventBusInject 对象

    private val instance = clazz.newInstance()
```

```
@Synchronized
fun register(subscriber: Any) {
    subscriptions.add(subscriber)
}

@Synchronized
fun unregister(subscriber: Any) {
    subscriptions.remove(subscriber)
}

@Synchronized
fun post(event: Any) {
    subscriptions.forEach { subscriber ->
        val subscriberInfo =
            getSubscriberInfo(subscriber.javaClass)
        if (subscriberInfo != null) {
            val methodList = subscriberInfo.methodList
            methodList.forEach { method ->
                if (method.eventType == event.javaClass) {
                    val declaredMethod = subscriber.javaClass.getDeclaredMethod(
                        method.methodName,
                        method.eventType
                    )
```

```
                    declaredMethod.invoke(subscriber, event)

                }

            }

        }

    }



    //通过反射调用 EventBusInject 的 getSubscriberInfo 方法

    private fun getSubscriberInfo(subscriberClass: Class<*>): SubscriberInfo? {

        val method = clazz.getMethod("getSubscriberInfo", Class::class.java)

        return method.invoke(instance, subscriberClass) as? SubscriberInfo

    }



}
```

复制代码

## 四、结尾

文本实现的 EasyEventBus 挺简陋的 AndroidOpenSourceDemo

# 三方库源码笔记（3）-ARouter 源码详解

## 一、ARouter

路由框架在大型项目中比较常见，特别是在项目中拥有多个 moudle 的时候。为了实现组件化，多个 module 间的通信就不能直接以模块间的引用来实现，此时就需要依赖路由框架来实现模块间的通信和解耦:sunglasses:

而 ARouter 就是一个用于帮助 Android App 进行组件化改造的框架，支持模块间的路由、通信、解耦

## 1、支持的功能

1. 支持直接解析标准 URL 进行跳转，并自动注入参数到目标页面中
2. 支持多模块工程使用
3. 支持添加多个拦截器，自定义拦截顺序
4. 支持依赖注入，可单独作为依赖注入框架使用
5. 支持 InstantRun
6. 支持 MultiDex(Google 方案)
7. 映射关系按组分类、多级管理，按需初始化
8. 支持用户指定全局降级与局部降级策略
9. 页面、拦截器、服务等组件均自动注册到框架
10. 支持多种方式配置转场动画
11. 支持获取 Fragment
12. 完全支持 Kotlin 以及混编(配置见文末 其他#5)
13. 支持第三方 App 加固(使用 arouter-register 实现自动注册)
14. 支持生成路由文档
15. 提供 IDE 插件便捷的关联路径和目标类

## 2、典型应用

1. 从外部 URL 映射到内部页面，以及参数传递与解析
2. 跨模块页面跳转，模块间解耦
3. 拦截跳转过程，处理登陆、埋点等逻辑
4. 跨模块 API 调用，通过控制反转来做组件解耦

以上介绍来自于 ARouter 的 Github 官网：README_CN

本文就基于其当前（2020/10/04）ARouter 的最新版本，对 ARouter 进行一次全面的源码解析和原理介绍，做到知其然也知其所以然，希望对你有所帮助

```
dependencies {

    implementation 'com.alibaba:arouter-api:1.5.0'

    kapt 'com.alibaba:arouter-compiler:1.2.2'

}

复制代码
```

## 二、前言

假设存在一个包含多个 moudle 的项目，在名为 **user** 的 moudle 中存在一个 `UserHomeActivity`，其对应的**路由路径**是 `/account/userHome`。那么，当我们要从其它 moudle 跳转到该页面时，只需要指定 path 来跳转即可

```
package github.leavesc.user



/**

 * 作者：leavesC

 * 时间：2020/10/3 18:05

 * 描述：

 * GitHub：https://github.com/leavesC

 */

@Route(path = RoutePath.USER_HOME)

class UserHomeActivity : AppCompatActivity() {



    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity_user_home)
```

```
    }



}



//其它页面使用如下代码来跳转到 UserHomeActivity

ARouter.getInstance().build(RoutePath.USER_HOME).navigation()

复制代码
```

只根据一个 path，ARouter 是如何定位到特定的 Activity 的呢？

这就需要通过在**编译阶段**生成辅助代码来实现了。我们都知道，想要跳转到某个 Activity，那么就需要拿到该 Activity 的 Class 对象才行。在编译阶段，ARouter 会根据我们设定的路由跳转规则来自动生成映射文件，映射文件中就包含了 path 和 ActivityClass 之间的对应关系

例如，对于 `UserHomeActivity`，在编译阶段就会自动生成以下辅助文件。可以看到，`ARouter$$Group$$account` 类中就将 path 和 ActivityClass 作为键值对保存到了 Map 中。ARouter 就是依靠此来进行跳转的

```
package com.alibaba.android.arouter.routes;

/**

 * DO NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER. */public class ARouter$$Grou
p$$account implements IRouteGroup {

  @Override

  public void loadInto(Map<String, RouteMeta> atlas) {

    atlas.put("/account/userHome", RouteMeta.build(RouteType.ACTIVITY, UserHomeActiv
ity.class, "/account/userhome", "account", null, -1, -2147483648));

  }

}
```

复制代码

还有一个重点需要注意，就是这类自动生成的文件的包名路径都是 `com.alibaba.android.arouter.routes`，且类名前缀也是有特定规则的。虽然 `ARouter$$Group$$account` 类实现了将对应关系保存到 Map 的逻辑，但是 `loadInto` 方法还是需要由 ARouter 在运行时来调用，那么 ARouter 就需要拿到 `ARouter$$Group$$account` 这个类才行，而 ARouter 就是通过扫描 `com.alibaba.android.arouter.routes` 这个包名路径来获取所有辅助文件的

ARouter 的基本实现思路就是：

1. 开发者自己维护**特定 path** 和**特定的目标类**之间的对应关系，ARouter 只要求开发者使用包含了 path 的 `@Route` 注解修饰**目标类**
2. ARouter 在**编译阶段**通过**注解处理器**来自动生成 path 和特定的目标类之间的对应关系，即将 path 作为 key，将目标类的 Class 对象作为 value 之一存到 Map 之中
3. 在运行阶段，应用通过 path 来发起请求，ARouter 根据 path 从 Map 中取值，从而拿到目标类

# 三、初始化

ARouter 的一般是放在 `Application` 中调用 `init` 方法来完成初始化的，这里先来看下其初始化流程

```
/**

 * 作者：leavesC

 * 时间：2020/10/4 18:05

 * 描述：

 * GitHub：https://github.com/leavesC

 */class MyApp : Application() {



    override fun onCreate() {
```

```
        super.onCreate()
        if (BuildConfig.DEBUG) {
            ARouter.openDebug()
            ARouter.openLog()
        }
        ARouter.init(this)
    }

}
```

复制代码

`ARouter` 类使用了单例模式，逻辑比较简单，因为 `ARouter` 类只是负责对外暴露可以由外部调用的 API，大部分的实现逻辑还是转交由 `_ARouter` 类来完成

```
public final class ARouter {

    private volatile static ARouter instance = null;

    private ARouter() {
    }

    /**
     * Get instance of router. A
     * All feature U use, will be starts here.
```

```
     */
    public static ARouter getInstance() {
        if (!hasInit) {
            throw new InitException("ARouter::Init::Invoke init(context) first!");
        } else {
            if (instance == null) {
                synchronized (ARouter.class) {
                    if (instance == null) {
                        instance = new ARouter();
                    }
                }
            }
            return instance;
        }
    }

    /**
     * Init, it must be call before used router.
     */
    public static void init(Application application) {
        if (!hasInit) { //防止重复初始化
            logger = _ARouter.logger;
            _ARouter.logger.info(Consts.TAG, "ARouter init start.");
```

```
            //通过 _ARouter 来完成初始化
            hasInit = _ARouter.init(application);
            if (hasInit) {
                _ARouter.afterInit();
            }
            _ARouter.logger.info(Consts.TAG, "ARouter init over.");
        }
    }

    ...


}
```

复制代码

_ARouter` 类是**包私有权限**，也使用了单例模式，其 `init(Application)` 方法的重点就在于 `LogisticsCenter.init(mContext, executor)

```
final class _ARouter {

    private volatile static _ARouter instance = null;

    private _ARouter() {
    }

    protected static _ARouter getInstance() {
        if (!hasInit) {
```

```
            throw new InitException("ARouterCore::Init::Invoke init(context) first!
");

        } else {

            if (instance == null) {

                synchronized (_ARouter.class) {

                    if (instance == null) {

                        instance = new _ARouter();

                    }

                }

            }

            return instance;

        }

    }



    protected static synchronized boolean init(Application application) {

        mContext = application;

        //重点

        LogisticsCenter.init(mContext, executor);

        logger.info(Consts.TAG, "ARouter init success!");

        hasInit = true;

        mHandler = new Handler(Looper.getMainLooper());

        return true;

    }
```

```
    ...


}
```

复制代码

`LogisticsCenter` 就实现了前文说的**扫描特定包名路径拿到所有自动生成的辅助文件**的逻辑，即在进行初始化的时候，我们就需要加载到当前项目一共包含的所有 group，以及每个 group 对应的路由信息表，其主要逻辑是：

1. 如果当前开启了 debug 模式或者通过本地 SP 缓存判断出 app 的版本前后发生了变化，那么就重新获取全局路由信息，否则就从使用之前缓存到 SP 中的数据
2. 获取全局路由信息是一个比较耗时的操作，所以 ARouter 就通过将全局路由信息缓存到 SP 中来实现复用。但由于在开发阶段开发者可能随时就会添加新的路由表，而每次发布新版本正常来说都是会加大应用的版本号的，所以 ARouter 就只在开启了 debug 模式或者是版本号发生了变化的时候才会重新获取路由信息
3. 获取到的路由信息中包含了在 `com.alibaba.android.arouter.routes` 这个包下自动生成的辅助文件的全路径，通过判断路径名的前缀字符串，就可以知道该类文件对应什么类型，然后通过反射构建不同类型的对象，通过调用对象的方法将路由信息存到 `Warehouse` 的 Map 中。至此，整个初始化流程就结束了

```
public class LogisticsCenter {


    /**
     * LogisticsCenter init, load all metas in memory. Demand initialization
     */
```

```
    public synchronized static void init(Context context, ThreadPoolExecutor tpe) thr
ows HandlerException {

        mContext = context;

        executor = tpe;



        try {

            long startInit = System.currentTimeMillis();

            //billy.qi modified at 2017-12-06

            //load by plugin first

            loadRouterMap();

            if (registerByPlugin) {

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

            } else {

                Set<String> routerMap;



                //如果当前开启了 debug 模式或者通过本地 SP 缓存判断出 app 的版本前后发生了
变化

                //那么就重新获取路由信息，否则就从使用之前缓存到 SP 中的数据

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

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

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

                    // These class was generated by arouter-compiler.

                    //获取 ROUTE_ROOT_PAKCAGE 包名路径下包含的所有的 ClassName
```

```
                routerMap = ClassUtils.getFileNameByPackageName(mContext, ROUTE_ROOT_PAKCAGE);

                if (!routerMap.isEmpty()) {

                    //缓存到 SP 中

                    context.getSharedPreferences(AROUTER_SP_CACHE_KEY, Context.MODE_PRIVATE).edit().putStringSet(AROUTER_SP_KEY_MAP, routerMap).apply();

                }

                //更新 App 的版本信息

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

            } else {

                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();



            for (String className : routerMap) {

                //通过 className 的前缀来判断该 class 对应的什么类型，并同时缓存到 Warehouse 中

                //1.IRouteRoot

                //2.IInterceptorGroup
```

```
                //3.IProviderGroup

                if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAME + SEP
ARATOR + SUFFIX_ROOT)) {

                    // This one of root elements, load root.

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

                } else if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAM
E + SEPARATOR + SUFFIX_INTERCEPTORS)) {

                    // Load interceptorMeta

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

                } else if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAM
E + SEPARATOR + SUFFIX_PROVIDERS)) {

                    // Load providerIndex

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

                }

            }

        }



        ...



    } catch (Exception e) {

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

    }
```

```
    }



}
```

复制代码

对于第三步，可以举个例子来加强理解。对于上文所讲的 `UserHomeActivity`，其对应的 path 是 `/account/userHome`，ARouter 默认会将 path 的第一个单词即 `account` 作为其 `group`，而且 `UserHomeActivity` 是放在名为 `user` 的 module 中

而 ARouter 在通过注解处理器生成辅助文件的时候，类名就会根据以上信息来生成，所以最终就会生成以下两个文件：

```
package com.alibaba.android.arouter.routes;

/**

 * DO NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER. */public class ARouter$$Root
$$user implements IRouteRoot {

  @Override

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

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

  }

}
```

复制代码

```
package com.alibaba.android.arouter.routes;

/**

 * DO NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER. */public class ARouter$$Grou
p$$account implements IRouteGroup {

  @Override

  public void loadInto(Map<String, RouteMeta> atlas) {
```

```
    atlas.put("/account/userHome", RouteMeta.build(RouteType.ACTIVITY, UserHomeActiv
ity.class, "/account/userhome", "account", null, -1, -2147483648));

  }

}
```

复制代码

`LogisticsCenter` 的 `init` 方法就会根据文件名的固定前缀 `ARouter$$Root$$` 定位到 `ARouter$$Root$$user` 这个类，然后通过反射构建出该对象，然后通过调用其 `loadInto` 方法将键值对保存到 `Warehouse.groupsIndex` 中。等到后续需要跳转到 `group` 为 `account` 的页面时，就会再来反射调用 `ARouter$$Group$$account` 的 `loadInto` 方法，即按需加载，等到需要的时候再来获取详细的路由对应信息

因为对于一个大型的 App 来说，可能包含一百或者几百个页面，如果一次性将所有路由信息都加载到内存中，对于内存的压力是比较大的，而用户每次使用可能也只会打开十几个页面，所以这里必须是按需加载

## 四、跳转到 Activity

讲完初始化流程，那就再来看下 ARouter 实现 Activity 跳转的流程

跳转到 Activity 最简单的方式就是只指定 path：

```
    ARouter.getInstance().build(RoutePath.USER_HOME).navigation()
```

复制代码

`build()` 方法会通过 `ARouter` 中转调用到 `_ARouter` 的 `build()` 方法，最终返回一个 `Postcard` 对象

```
    /**

     * Build postcard by path and default group

     */
```

```
    protected Postcard build(String path) {
        if (TextUtils.isEmpty(path)) {
            throw new HandlerException(Consts.TAG + "Parameter is invalid!");
        } else {
            PathReplaceService pService = ARouter.getInstance().navigation(PathReplac
eService.class);
            if (null != pService) {
                //用于路径替换，这对于某些需要控制页面跳转流程的场景比较有用
                //例如，如果某个页面需要登录才可以展示的话
                //就可以通过 PathReplaceService 将 path 替换 loginPagePath
                path = pService.forString(path);
            }
            //使用字符串 path 包含的第一个单词作为 group
            return build(path, extractGroup(path));
        }
    }


    /**
     * Build postcard by path and group
     */
    protected Postcard build(String path, String group) {
        if (TextUtils.isEmpty(path) || TextUtils.isEmpty(group)) {
            throw new HandlerException(Consts.TAG + "Parameter is invalid!");
```

```
        } else {
            PathReplaceService pService = ARouter.getInstance().navigation(PathReplaceService.class);
            if (null != pService) {
                path = pService.forString(path);
            }
            return new Postcard(path, group);
        }
    }
```

复制代码

返回的 `Postcard` 对象可以用于传入一些跳转配置参数，例如：携带参数 `mBundle`、开启绿色通道 `greenChannel` 、跳转动画 `optionsCompat` 等

```
public final class Postcard extends RouteMeta {
    // Base
    private Uri uri;
    private Object tag;             // A tag prepare for some thing wrong.
    private Bundle mBundle;         // Data to transform
    private int flags = -1;         // Flags of route
    private int timeout = 300;      // Navigation timeout, TimeUnit.Second
    private IProvider provider;     // It will be set value, if this postcard was provider.
    private boolean greenChannel;
    private SerializationService serializationService;
```

```
}
```

复制代码

`Postcard` 的 `navigation()` 方法又会调用到 `_ARouter` 的以下方法来完成 Activity 的跳转。该方法逻辑上并不复杂，注释也写得很清楚了

```
final class _ARouter {



    /**
     * Use router navigation.
     *
     * @param context     Activity or null.
     * @param postcard    Route metas
     * @param requestCode RequestCode
     * @param callback    cb
     */
    protected Object navigation(final Context context, final Postcard postcard, final
 int requestCode, final NavigationCallback callback) {
        PretreatmentService pretreatmentService = ARouter.getInstance().navigation(P
retreatmentService.class);
        if (null != pretreatmentService && !pretreatmentService.onPretreatment(contex
t, postcard)) {
            // Pretreatment failed, navigation canceled.
            //用于执行跳转前的预处理操作，可以通过 onPretreatment 方法的返回值决定是否取消
跳转
```

```
            return null;
        }


        try {
            LogisticsCenter.completion(postcard);
        } catch (NoRouteFoundException ex) {
            //没有找到匹配的目标类
            //下面就执行一些提示操作和事件回调通知


            logger.warning(Consts.TAG, ex.getMessage());
            if (debuggable()) {
                // Show friendly tips for user.
                runInMainThread(new Runnable() {
                    @Override
                    public void run() {
                        Toast.makeText(mContext, "There's no route matched!\n" +
                                " Path = [" + postcard.getPath() + "]\n" +
                                " Group = [" + postcard.getGroup() + "]", Toast.LENGTH_
LONG).show();
                    }
                });
            }
            if (null != callback) {
```

```
                callback.onLost(postcard);
            } else {
                // No callback for this invoke, then we use the global degrade servic
e.
                DegradeService degradeService = ARouter.getInstance().navigation(Degr
adeService.class);
                if (null != degradeService) {
                    degradeService.onLost(context, postcard);
                }
            }
            return null;
        }

        if (null != callback) {
            //找到了匹配的目标类
            callback.onFound(postcard);
        }

        if (!postcard.isGreenChannel()) {   // It must be run in async thread, maybe
interceptor cost too mush time made ANR.
            //没有开启绿色通道，那么就还需要执行所有拦截器
            //外部可以通过拦截器实现：控制是否允许跳转、更改跳转参数等逻辑

            interceptorService.doInterceptions(postcard, new InterceptorCallback() {
```

```
            /**
             * Continue process
             *
             * @param postcard route meta
             */
            @Override
            public void onContinue(Postcard postcard) {
                //拦截器允许跳转
                _navigation(context, postcard, requestCode, callback);
            }


            /**
             * Interrupt process, pipeline will be destory when this method calle
d.
             *
             * @param exception Reson of interrupt.
             */
            @Override
            public void onInterrupt(Throwable exception) {
                if (null != callback) {
                    callback.onInterrupt(postcard);
                }
```

```
                    logger.info(Consts.TAG, "Navigation failed, termination by interce
ptor : " + exception.getMessage());

                }

            });

        } else {

            //开启了绿色通道，直接跳转，不需要遍历拦截器

            return _navigation(context, postcard, requestCode, callback);

        }



        return null;

    }



    //由于本例子的目标页面是 Activity，所以只看 ACTIVITY 即可

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

        final Context currentContext = null == context ? mContext : context;

        switch (postcard.getType()) {

            case ACTIVITY:

                // Build intent

                //Destination 就是指向目标 Activity 的 class 对象

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

                //塞入携带的参数

                intent.putExtras(postcard.getExtras());
```

```
            // Set flags.
            int flags = postcard.getFlags();
            if (-1 != flags) {
                intent.setFlags(flags);
            } else if (!(currentContext instanceof Activity)) {    // Non activity, need less one flag.
                intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            }

            // Set Actions
            String action = postcard.getAction();
            if (!TextUtils.isEmpty(action)) {
                intent.setAction(action);
            }

            // Navigation in main looper.
            //最终在主线程完成跳转
            runInMainThread(new Runnable() {
                @Override
                public void run() {
                    startActivity(requestCode, currentContext, intent, postcard, callback);
                }
```

```
                });


                break;

            ... //省略其它类型判断

        }


        return null;

    }


}
```

复制代码

`navigation` 方法的重点在于 `LogisticsCenter.completion(postcard)` 这一句代码。在讲 ARouter 初始化流程的时候有讲到：等到后续需要跳转到 `group` 为 `account` 的页面时，就会再来反射调用 `ARouter$$Group$$account` 的 `loadInto` 方法，即按需加载，等到需要的时候再来获取详细的路由对应信息

```
completion` 方法就是用来获取详细的路由对应信息的。该方法会通过 `postcard` 携带的 path 和 group 信息从 `Warehouse` 取值，如果值不为 null 的话就将信息保存到 `postcard` 中，如果值为 null 的话就抛出 `NoRouteFoundException

    /**

     * Completion the postcard by route metas

     *

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

     */

    public synchronized static void completion(Postcard postcard) {
```

```
        if (null == postcard) {

            throw new NoRouteFoundException(TAG + "No postcard!");

        }



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

        if (null == routeMeta) {    //为 null 说明目标类不存在或者是该 group 还未加载过

            Class<? extends IRouteGroup> groupMeta = Warehouse.groupsIndex.get(postca
rd.getGroup());  // Load route meta.

            if (null == groupMeta) {

                //groupMeta 为 null，说明 postcard 的 path 对应的 group 不存在，抛出异常

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

            } else {

                // Load route and cache it into memory, then delete from metas.

                try {

                    if (ARouter.debuggable()) {

                        logger.debug(TAG, String.format(Locale.getDefault(), "The grou
p [%s] starts loading, trigger by [%s]", postcard.getGroup(), postcard.getPath()));

                    }

                  //会执行到这里，说明此 group 还未加载过，那么就来反射加载 group 对应的所有
 path 信息

                   //获取后就保存到 Warehouse.routes

                    IRouteGroup iGroupInstance = groupMeta.getConstructor().newInstanc
e();

                    iGroupInstance.loadInto(Warehouse.routes);
```

```
                //移除此 group
                Warehouse.groupsIndex.remove(postcard.getGroup());

                if (ARouter.debuggable()) {
                    logger.debug(TAG, String.format(Locale.getDefault(), "The grou
p [%s] has already been loaded, trigger by [%s]", postcard.getGroup(), postcard.getP
ath()));
                }
            } catch (Exception e) {
                throw new HandlerException(TAG + "Fatal exception when loading gro
up meta. [" + e.getMessage() + "]");
            }

            //重新执行一遍
            completion(postcard);   // Reload
        }
    } else {
        //拿到详细的路由信息了，将这些信息存到 postcard 中

        postcard.setDestination(routeMeta.getDestination());
        postcard.setType(routeMeta.getType());
        postcard.setPriority(routeMeta.getPriority());
        postcard.setExtra(routeMeta.getExtra());
```

```
            //省略一些和本例子无关的代码
            ...
        }
    }
```

复制代码

## 五、跳转到 Activity 并注入参数

ARouter 也支持在跳转到 Activity 的同时向目标页面自动注入参数

在跳转的时候指定要携带的键值对参数：

```
ARouter.getInstance().build(RoutePath.USER_HOME)
        .withLong(RoutePath.USER_HOME_PARAMETER_ID, 20)
        .withString(RoutePath.USER_HOME_PARAMETER_NAME, "leavesC")
        .navigation()
object RoutePath {

    const val USER_HOME = "/account/userHome"

    const val USER_HOME_PARAMETER_ID = "userHomeId"

    const val USER_HOME_PARAMETER_NAME = "userName"
```

```
}
```

复制代码

在目标页面通过 `@Autowired` 注解修饰变量。注解可以同时声明其 `name` 参数，用于和传递的键值对中的 key 对应上，这样 ARouter 才知道应该向哪个变量赋值。如果没有声明 `name` 参数，那么 `name` 参数就默认和**变量名**相等

这样，在我们调用 `ARouter.getInstance().inject(this)` 后，ARouter 就会自动完成参数的赋值

```
package github.leavesc.user

/**
 * 作者：leavesC
 * 时间：2020/10/4 14:05
 * 描述：
 * GitHub：https://github.com/leavesC
 */@Route(path = RoutePath.USER_HOME)class UserHomeActivity : AppCompatActivity() {


    @Autowired(name = RoutePath.USER_HOME_PARAMETER_ID)
    @JvmField
    var userId: Long = 0


    @Autowired
    @JvmField
    var userName = ""
```

```
    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity_user_home)

        ARouter.getInstance().inject(this)

        tv_hint.text = "$userId $userName"

    }



}
```

复制代码

ARouter 实现参数自动注入也需要依靠注解处理器生成的辅助文件来实现，即会生成以下的辅助代码：

```
package github.leavesc.user;

/**

 * DO NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER. */public class UserHomeActiv
ity$$ARouter$$Autowired implements ISyringe {



  //用于实现序列化和反序列化

  private SerializationService serializationService;



  @Override

  public void inject(Object target) {

    serializationService = ARouter.getInstance().navigation(SerializationService.cla
ss);
```

```
    UserHomeActivity substitute = (UserHomeActivity)target;

    substitute.userId = substitute.getIntent().getLongExtra("userHomeId", substitute.
userId);

    substitute.userName = substitute.getIntent().getStringExtra("userName");

  }

}
```

复制代码

因为在跳转到 Activity 时携带的参数也是需要放到 `Intent` 里的，所以 `inject` 方法也只是帮我们实现了从 `Intent` 取值然后向变量赋值的逻辑而已，这就要求相应的变量必须是 `public` 的，这就是在 Kotlin 代码中需要同时向变量加上 `@JvmField` 注解的原因

现在来看下 ARouter 是如何实现**参数自动注入**的，其起始方法就是：`ARouter.getInstance().inject(this)`，其最终会调用到以下方法

```
final class _ARouter {



    static void inject(Object thiz) {

        AutowiredService autowiredService = ((AutowiredService) ARouter.getInstance().
build("/arouter/service/autowired").navigation());

        if (null != autowiredService) {

            autowiredService.autowire(thiz);

        }

    }



}
```

复制代码

ARouter 通过控制反转的方式拿到 `AutowiredService` 对应的实现类 `AutowiredServiceImpl` 的实例对象，然后调用其 `autowire` 方法完成参数注入

由于生成的**参数注入辅助类**的类名具有**固定的包名和类名**，即包名和目标类所在包名一致，类名是**目标类类名+** `$$ARouter$$Autowired`，所以在 `AutowiredServiceImpl` 中就可以根据传入的 `instance` 参数和反射来生成辅助类对象，最终调用其 `inject` 方法完成参数注入

```
@Route(path = "/arouter/service/autowired")public class AutowiredServiceImpl impleme
nts AutowiredService {

    private LruCache<String, ISyringe> classCache;

    private List<String> blackList;



    @Override

    public void init(Context context) {

        classCache = new LruCache<>(66);

        blackList = new ArrayList<>();

    }



    @Override

    public void autowire(Object instance) {

        String className = instance.getClass().getName();

        try {

            //如果在白名单中了的话，那么就不再执行参数注入

            if (!blackList.contains(className)) {
```

```
                ISyringe autowiredHelper = classCache.get(className);

                if (null == autowiredHelper) {  // No cache.

                    autowiredHelper = (ISyringe) Class.forName(instance.getClass().get
Name() + SUFFIX_AUTOWIRED).getConstructor().newInstance();

                }

                //完成参数注入

                autowiredHelper.inject(instance);

                //缓存起来，避免重复反射

                classCache.put(className, autowiredHelper);

            }

        } catch (Exception ex) {

            //如果参数注入过程抛出异常，那么就将其加入白名单中

            blackList.add(className);    // This instance need not autowired.

        }

    }

}
```

复制代码

## 六、控制反转

上一节所讲的**跳转到 Activity 并自动注入参数**属于**依赖注入**的一种，ARouter 同时也支持**控制反转**：通过接口来获取其实现类实例

例如，假设存在一个 `ISayHelloService` 接口，我们需要拿到其实现类实例，但是不希望在使用的时候和特定的实现类 `SayHelloService` 绑定在一起从而造成强耦合，此时就可以使用 ARouter 的控制反转功能，但这也要求 `ISayHelloService` 接口继承了 `IProvider` 接口才行

```
/**
 * 作者：leavesC
 * 时间：2020/10/4 13:49
 * 描述：
 * GitHub：https://github.com/leavesC
 */interface ISayHelloService : IProvider {

    fun sayHello()

}
@Route(path = RoutePath.SERVICE_SAY_HELLO)class SayHelloService : ISayHelloService {

    override fun init(context: Context) {

    }

    override fun sayHello() {
        Log.e("SayHelloService", "$this sayHello")
    }

}
```

复制代码

在使用的时候直接传递 `ISayHelloService` 的 Class 对象即可，ARouter 会将 `SayHelloService` 以单例模式的形式返回，无需开发者手动去构建 `SayHelloService` 对象，从而达到解耦的目的

```
ARouter.getInstance().navigation(ISayHelloService::class.java).sayHello()

复制代码
```

和实现 Activity 跳转的时候一样，ARouter 也会自动生成以下几个文件，包含了路由表的映射关系

```
package com.alibaba.android.arouter.routes;

/**

 * DO NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER. */public class ARouter$$Grou
p$$account implements IRouteGroup {

  @Override

  public void loadInto(Map<String, RouteMeta> atlas) {

    atlas.put("/account/sayHelloService", RouteMeta.build(RouteType.PROVIDER, SayHel
loService.class, "/account/sayhelloservice", "account", null, -1, -2147483648));

  }

}

/**

 * DO NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER. */public class ARouter$$Prov
iders$$user implements IProviderGroup {

  @Override

  public void loadInto(Map<String, RouteMeta> providers) {

    providers.put("github.leavesc.user.ISayHelloService", RouteMeta.build(RouteType.
PROVIDER, SayHelloService.class, "/account/sayHelloService", "account", null, -1, -2
147483648));

  }
```

```
}

/**

 * DO NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER. */public class ARouter$$Root
$$user implements IRouteRoot {

  @Override

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

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

  }

}
```

复制代码

这里再来看下其具体的实现原理

在讲初始化流程的时候有讲到，`LogisticsCenter` 实现了**扫描特定包名路径拿到所有自动生成的辅助文件**的逻辑。所以，最终 Warehouse 中就会在初始化的时候拿到以下数据

Warehouse.groupsIndex：

- `account -> class com.alibaba.android.arouter.routes.ARouter$$Group$$account`

Warehouse.providersIndex：

- `github.leavesc.user.ISayHelloService -> RouteMeta.build(RouteType.PROVIDER, SayHelloService.class, "/account/sayHelloService", "account", null, -1, -2147483648)`

`ARouter.getInstance().navigation(ISayHelloService::class.java)` 最终会中转调用到 `_ARouter` 的以下方法

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

        try {
```

```
            //从 Warehouse.providersIndex 取值拿到 RouteMeta 中存储的 path 和 group
            Postcard postcard = LogisticsCenter.buildProvider(service.getName());

            // Compatible 1.0.5 compiler sdk.
            // Earlier versions did not use the fully qualified name to get the servic
e
            if (null == postcard) {
                // No service, or this service in old version.
                postcard = LogisticsCenter.buildProvider(service.getSimpleName());
            }

            if (null == postcard) {
                return null;
            }
           //重点
            LogisticsCenter.completion(postcard);
            return (T) postcard.getProvider();
        } catch (NoRouteFoundException ex) {
            logger.warning(Consts.TAG, ex.getMessage());
            return null;
        }
    }
```

复制代码

`LogisticsCenter.completion(postcard)` 方法的流程和之前讲解的差不多，只是在获取对象实例的时候同时将实例缓存起来，留待之后复用，至此就完成了控制反转的流程了

```
    /**
     * Completion the postcard by route metas
     *
     * @param postcard Incomplete postcard, should complete by this method.
     */
    public synchronized static void completion(Postcard postcard) {
       ... //省略之前已经讲解过的代码


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


        switch (routeMeta.getType()) {
                case PROVIDER:  // if the route is provider, should find its instance
                    // Its provider, so it must implement IProvider
                    //拿到 SayHelloService Class 对象
                    Class<? extends IProvider> providerMeta = (Class<? extends IProvid
er>) routeMeta.getDestination();
                    IProvider instance = Warehouse.providers.get(providerMeta);
                    if (null == instance) { // There's no instance of this provider
                        //instance 等于 null 说明是第一次取值
                        //那么就通过反射构建 SayHelloService 对象，然后将之缓存到 Warehous
e.providers 中
```

```
                    //所以通过控制反转获取的对象在应用的整个生命周期内只会有一个实例
                    IProvider provider;
                    try {
                        provider = providerMeta.getConstructor().newInstance();
                        provider.init(mContext);
                        Warehouse.providers.put(providerMeta, provider);
                        instance = provider;
                    } catch (Exception e) {
                        throw new HandlerException("Init provider failed! " + e.get
Message());
                    }
                }
                //将获取到的实例存起来
                postcard.setProvider(instance);
                postcard.greenChannel();    // Provider should skip all of interce
ptors
                break;
            case FRAGMENT:
                postcard.greenChannel();    // Fragment needn't interceptors
            default:
                break;
        }

    }
```

复制代码

## 七、拦截器

ARouter 的拦截器对于某些需要控制页面跳转流程的业务逻辑来说是十分有用的功能。例如，用户如果要跳转到个人资料页面时，我们就可以通过拦截器来判断用户是否处于已登录状态，还未登录的话就可以拦截该请求，然后自动为用户打开登录页面

我们可以同时设定多个拦截器，每个拦截器设定不同的优先级

```
/**

 * 作者：leavesC

 * 时间：2020/10/5 11:49

 * 描述：

 * GitHub：https://github.com/leavesC

 */@Interceptor(priority = 100, name = "啥也不做的拦截器")class NothingInterceptor : IInterceptor {



    override fun init(context: Context) {



    }



    override fun process(postcard: Postcard, callback: InterceptorCallback) {

        //不拦截，任其跳转

        callback.onContinue(postcard)

    }
```

```
}

@Interceptor(priority = 200, name = "登陆拦截器")class LoginInterceptor : IIntercepto
r {


    override fun init(context: Context) {


    }


    override fun process(postcard: Postcard, callback: InterceptorCallback) {
        if (postcard.path == RoutePath.USER_HOME) {
            //拦截
            callback.onInterrupt(null)
            //跳转到登陆页
            ARouter.getInstance().build(RoutePath.USER_LOGIN).navigation()
        } else {
            //不拦截，任其跳转
            callback.onContinue(postcard)
        }
    }


}
```

复制代码

这样，当我们执行 `ARouter.getInstance().build(RoutePath.USER_HOME).navigation()` 想要跳转的时候，就会发现打开的其实是登录页 `RoutePath.USER_LOGIN`

来看下拦截器是如何实现的

对于以上的两个拦截器，会生成以下的辅助文件。辅助文件会拿到所有我们自定义的拦截器实现类并根据优先级高低存到 Map 中

```
package com.alibaba.android.arouter.routes;

/**

 * DO NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER. */public class ARouter$$Inte
rceptors$$user implements IInterceptorGroup {

  @Override

  public void loadInto(Map<Integer, Class<? extends IInterceptor>> interceptors) {

    interceptors.put(100, NothingInterceptor.class);

    interceptors.put(200, LoginInterceptor.class);

  }

}
```

复制代码

而这些拦截器一样是会在初始化的时候，通过 `LogisticsCenter.init` 方法存到 `Warehouse.interceptorsIndex` 中

```
 /**

    * LogisticsCenter init, load all metas in memory. Demand initialization

    */

   public synchronized static void init(Context context, ThreadPoolExecutor tpe) thr
ows HandlerException {
```

```
        ...



        for (String className : routerMap) {

                    if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAME + SEP
ARATOR + SUFFIX_ROOT)) {

                        // This one of root elements, load root.

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

                    } else if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAM
E + SEPARATOR + SUFFIX_INTERCEPTORS)) {

                        // Load interceptorMeta

                        //拿到自定义的拦截器实现类

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

                    } else if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAM
E + SEPARATOR + SUFFIX_PROVIDERS)) {

                        // Load providerIndex

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

                    }

                }



            ...



    }
```

复制代码

然后，在 `_ARouter` 的 `navigation` 方法中，如何判断到此次路由请求没有开启绿色通道模式的话，那么就会将此次请求转交给 `interceptorService`，让其去遍历每个拦截器

```
final class _ARouter {



    /**

     * Use router navigation.

     *

     * @param context     Activity or null.

     * @param postcard    Route metas

     * @param requestCode RequestCode

     * @param callback    cb

     */

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



        ...



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



            //遍历拦截器

            interceptorService.doInterceptions(postcard, new InterceptorCallback() {
```

```
                /**
                 * Continue process
                 *
                 * @param postcard route meta
                 */
                @Override
                public void onContinue(Postcard postcard) {
                    _navigation(context, postcard, requestCode, callback);
                }

                /**
                 * Interrupt process, pipeline will be destory when this method calle
d.
                 *
                 * @param exception Reson of interrupt.
                 */
                @Override
                public void onInterrupt(Throwable exception) {
                    if (null != callback) {
                        callback.onInterrupt(postcard);
                    }

                    logger.info(Consts.TAG, "Navigation failed, termination by interce
ptor : " + exception.getMessage());
```

```
                }

            });

        } else {

            return _navigation(context, postcard, requestCode, callback);

        }


        return null;

    }


}
```

复制代码

`interceptorService` 变量属于 `InterceptorService` 接口类型，该接口的实现类是 `InterceptorServiceImpl`，ARouter 内部在初始化的过程中也是根据控制反转的方式来拿到 `interceptorService` 这个实例的

`InterceptorServiceImpl` 的主要逻辑是：

1. 在第一次获取 `InterceptorServiceImpl` 实例的时候，其 `init` 方法会马上被调用，该方法内部会交由线程池来执行，通过反射生成每个拦截器对象，并调用每个拦截器的 `init` 方法来完成拦截器的初始化，并将每个拦截器对象都存到 `Warehouse.interceptors` 中。如果初始化完成了，则唤醒等待在 `interceptorInitLock` 上的线程
2. 当拦截器逻辑被触发，即 `doInterceptions` 方法被调用时，如果此时第一个步骤还未执行完的话，则会通过 `checkInterceptorsInitStatus()`方法等待第一个步骤执行完成。如果十秒内都未完成的话，则走失败流程直接返回

3. 在线程池中遍历拦截器列表，如果有某个拦截器拦截了请求的话则调用 `callback.onInterrupt` 方法通知外部，否则的话则调用 `callback.onContinue()` 方法继续跳转逻辑

```
@Route(path = "/arouter/service/interceptor")public class InterceptorServiceImpl imp
lements InterceptorService {

    private static boolean interceptorHasInit;

    private static final Object interceptorInitLock = new Object();



    @Override

    public void init(final Context context) {

        LogisticsCenter.executor.execute(new Runnable() {

            @Override

            public void run() {

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

                    //遍历拦截器列表，通过反射构建对象并初始化

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

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

                        try {

                            IInterceptor iInterceptor = interceptorClass.getConstructo
r().newInstance();

                            iInterceptor.init(context);

                            //存起来

                            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();

                }

            }

        }

    });

}



    @Override

    public void doInterceptions(final Postcard postcard, final InterceptorCallback c
allback) {

        if (null != Warehouse.interceptors && Warehouse.interceptors.size() > 0) {
```

```
            checkInterceptorsInitStatus();


            if (!interceptorHasInit) {
                //初始化太久，不等了，直接走失败流程
                callback.onInterrupt(new HandlerException("Interceptors initializatio
n takes too much time."));
                return;
            }


            LogisticsCenter.executor.execute(new Runnable() {
                @Override
                public void run() {
                    CancelableCountDownLatch interceptorCounter = new CancelableCountD
ownLatch(Warehouse.interceptors.size());
                    try {
                        _excute(0, interceptorCounter, postcard);
                        interceptorCounter.await(postcard.getTimeout(), TimeUnit.SECON
DS);
                        if (interceptorCounter.getCount() > 0) {    // Cancel the navig
ation this time, if it hasn't return anythings.
                            //大于 0 说明此次请求被某个拦截器拦截了，走失败流程
                            callback.onInterrupt(new HandlerException("The interceptor
 processing timed out."));
                        } else if (null != postcard.getTag()) {    // Maybe some except
ion in the tag.
```

```
                        callback.onInterrupt(new HandlerException(postcard.getTag
().toString()));

                    } else {

                        callback.onContinue(postcard);

                    }

                } catch (Exception e) {

                    callback.onInterrupt(e);

                }

            }

        });

    } else {

        callback.onContinue(postcard);

    }

}



/**

 * Excute interceptor

 *

 * @param index    current interceptor index

 * @param counter  interceptor counter

 * @param postcard routeMeta

 */

private static void _excute(final int index, final CancelableCountDownLatch count
er, final Postcard postcard) {
```

```
        if (index < Warehouse.interceptors.size()) {

            IInterceptor iInterceptor = Warehouse.interceptors.get(index);

            iInterceptor.process(postcard, new InterceptorCallback() {

                @Override

                public void onContinue(Postcard postcard) {

                    // Last interceptor excute over with no exception.

                    counter.countDown();

                    _excute(index + 1, counter, postcard);  // When counter is down, i
t will be execute continue ,but index bigger than interceptors size, then U know.

                }



                @Override

                public void onInterrupt(Throwable exception) {

                    // Last interceptor excute over with fatal exception.



                    postcard.setTag(null == exception ? new HandlerException("No messa
ge.") : exception.getMessage());    // save the exception message for backup.

                    counter.cancel();

                    // Be attention, maybe the thread in callback has been changed,

                    // then the catch block(L207) will be invalid.

                    // The worst is the thread changed to main thread, then the app wi
ll be crash, if you throw this exception!//                    if (!Looper.getMainLoop
er().equals(Looper.myLooper())) {    // You shouldn't throw the exception if the thre
ad is main thread.//                        throw new HandlerException(exception.getMe
ssage());//                    }

                }
```

```
            });

        }

    }



    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 t
ime error! reason = [" + e.getMessage() + "]");

                }

            }

        }

    }



}
```

复制代码

## 八、注解处理器

通篇读下来，读者应该能够感受到注解处理器在 ARouter 中起到了很大的作用，依靠注解处理器生成的辅助文件，ARouter 才能完成**参数自动注入**等功能。这里就再来介绍下 ARouter 关于注解处理器的实现原理

APT(**Annotation Processing Tool**) 即注解处理器，是一种注解处理工具，用来在编译期扫描和处理注解，通过注解来生成 Java 文件。即以注解作为桥梁，通过预先规定好的代码生成规则来自动生成 Java 文件。此类注解框架的代表有 **ButterKnife、Dragger2、EventBus** 等

Java API 已经提供了扫描源码并解析注解的框架，开发者可以通过继承 **AbstractProcessor** 类来实现自己的注解解析逻辑。APT 的原理就是在注解了某些代码元素（如字段、函数、类等）后，在编译时编译器会检查 **AbstractProcessor** 的子类，并且自动调用其 **process()** 方法，然后将添加了指定注解的所有代码元素作为参数传递给该方法，开发者再根据注解元素在编译期输出对应的 Java 代码

关于 APT 技术的原理和应用可以看这篇文章：Android APT 实例讲解

ARouter 源码中和注解处理器相关的 module 有两个：

- arouter-annotation。Java Module，包含了像 Autowired、Interceptor 这些注解以及 RouteMeta 等 JavaBean
- arouter-compiler。Android Module，包含了多个 AbstractProcessor 的实现类用于生成代码

这里主要来看 `arouter-compiler`，这里以自定义的拦截器 `NothingInterceptor` 作为例子

```
package github.leavesc.user

/**

 * 作者：leavesC

 * 时间：2020/10/5 11:49

 * 描述：

 * GitHub：https://github.com/leavesC

 */@Interceptor(priority = 100, name = "啥也不做的拦截器")class NothingInterceptor : IInterceptor {



    override fun init(context: Context) {
```

```
    }


    override fun process(postcard: Postcard, callback: InterceptorCallback) {
        //不拦截，任其跳转
        callback.onContinue(postcard)
    }


}
```

复制代码

生成的辅助文件：

```
package com.alibaba.android.arouter.routes;

import com.alibaba.android.arouter.facade.template.IInterceptor;import com.alibaba.android.arouter.facade.template.IInterceptorGroup;import github.leavesc.user.NothingInterceptor;import java.lang.Class;import java.lang.Integer;import java.lang.Override;import java.util.Map;

/**
 * DO NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER. */public class ARouter$$Interceptors$$user implements IInterceptorGroup {
  @Override
  public void loadInto(Map<Integer, Class<? extends IInterceptor>> interceptors) {
    interceptors.put(100, NothingInterceptor.class);
  }
}
```

复制代码

那么，生成的辅助文件我们就要包含以下几个元素：

1. 包名
2. 导包
3. 注释
4. 实现类及继承的接口
5. 包含的方法及方法参数
6. 方法体
7. 修饰符

如果通过硬编码的形式，即通过拼接字符串的方式来生成以上代码也是可以的，但是这样会使得代码不好维护且可读性很低，所以 ARouter 是通过 `JavaPoet` 这个开源库来生成代码的。`JavaPoet` 是 square 公司开源的 Java 代码生成框架，可以很方便地通过其提供的 API 来生成指定格式（修饰符、返回值、参数、函数体等）的代码

拦截器对应的 `AbstractProcessor` 子类就是 `InterceptorProcessor`，其主要逻辑是：

1. 在 process 方法中通过 RoundEnvironment 拿到所有使用了 @Interceptor 注解进行修饰的代码元素 elements，然后遍历所有 item
2. 判断每个 item 是否继承了 IInterceptor 接口，是的话则说明该 item 就是我们要找的拦截器实现类
3. 获取每个 item 包含的 @Interceptor 注解对象，根据我们为之设定的优先级 priority，将每个 item 按顺序存到 interceptors 中
4. 如果存在两个拦截器的优先级相同，那么就抛出异常
5. 将所有拦截器按顺序存入 interceptors 后，通过 JavaPoet 提供的 API 来生成**包名、导包、注释、实现类**等多个代码元素，并最终生成一个完整的类文件

```
@AutoService(Processor.class)@SupportedAnnotationTypes(ANNOTATION_TYPE_INTECEPTOR)p
ublic class InterceptorProcessor extends BaseProcessor {
```

```
    //用于保存拦截器，按照优先级高低进行排序
    private Map<Integer, Element> interceptors = new TreeMap<>();

    private TypeMirror iInterceptor = null;

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

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

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

    /**
     * {@inheritDoc}
     *
     * @param annotations
     * @param roundEnv
     */
    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment
roundEnv) {
```

```
        if (CollectionUtils.isNotEmpty(annotations)) {

            //拿到所有使用了 @Interceptor 进行修饰的代码元素

            Set<? extends Element> elements = roundEnv.getElementsAnnotatedWith(Inter
ceptor.class);

            try {

                parseInterceptors(elements);

            } catch (Exception e) {

                logger.error(e);

            }

            return true;

        }



        return false;

    }



    /**

     * Parse tollgate.

     *

     * @param elements elements of tollgate.

     */

    private void parseInterceptors(Set<? extends Element> elements) throws IOExcepti
on {

        if (CollectionUtils.isNotEmpty(elements)) {
```

```
            logger.info(">>> Found interceptors, size is " + elements.size() + " <<<
");



            // Verify and cache, sort incidentally.

            for (Element element : elements) {

                //判断使用了 @Interceptor 进行修饰的代码元素是否同时实现了 com.alibaba.an
droid.arouter.facade.template.IInterceptor 这个接口

                //两者缺一不可

                if (verify(element)) {  // Check the interceptor meta

                    logger.info("A interceptor verify over, its " + element.asType());

                    Interceptor interceptor = element.getAnnotation(Interceptor.clas
s);



                    Element lastInterceptor = interceptors.get(interceptor.priority
());

                    if (null != lastInterceptor) { // Added, throw exceptions

                        //不为 null 说明存在两个拦截器其优先级相等，这是不允许的，直接抛出异
常

                        throw new IllegalArgumentException(

                                String.format(Locale.getDefault(), "More than one inter
ceptors use same priority [%d], They are [%s] and [%s].",

                                        interceptor.priority(),

                                        lastInterceptor.getSimpleName(),

                                        element.getSimpleName())

                        );

                    }
```

```
                    //将拦截器按照优先级高低进行排序保存
                    interceptors.put(interceptor.priority(), element);
                } else {
                    logger.error("A interceptor verify failed, its " + element.asType
());
                }
            }

            // Interface of ARouter.
            //拿到 com.alibaba.android.arouter.facade.template.IInterceptor 这个接口的
类型抽象
            TypeElement type_ITollgate = elementUtils.getTypeElement(IINTERCEPTOR);
            //拿到 com.alibaba.android.arouter.facade.template.IInterceptorGroup 这个
接口的类型抽象
            TypeElement type_ITollgateGroup = elementUtils.getTypeElement(IINTERCEPTO
R_GROUP);

            /**
             *  Build input type, format as :
             *
             *  ```Map<Integer, Class<? extends ITollgate>>```
             */
            //生成对 Map<Integer, Class<? extends IInterceptor>> 这段代码的抽象封装
            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.
        //生成 loadInto 方法的入参参数 interceptors
        ParameterSpec tollgateParamSpec = ParameterSpec.builder(inputMapTypeOfTollgate, "interceptors").build();

        // Build method : 'loadInto'
        //生成 loadInto 方法
        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()) {
```

```
                    //遍历每个拦截器，生成 interceptors.put(100, NothingInterceptor.clas
s); 这类型的代码

                    loadIntoMethodOfTollgateBuilder.addStatement("interceptors.put("
+ entry.getKey() + ", $T.class)", ClassName.get((TypeElement) entry.getValue()));

                }

            }



            // Write to disk(Write file even interceptors is empty.)

            //包名固定是 PACKAGE_OF_GENERATE_FILE，即 com.alibaba.android.arouter.rout
es

            JavaFile.builder(PACKAGE_OF_GENERATE_FILE,

                    TypeSpec.classBuilder(NAME_OF_INTERCEPTOR + SEPARATOR + moduleName)
 //设置类名

                            .addModifiers(PUBLIC) //添加 public 修饰符

                            .addJavadoc(WARNING_TIPS) //添加注释

                            .addMethod(loadIntoMethodOfTollgateBuilder.build()) //添加
loadInto 方法

                            .addSuperinterface(ClassName.get(type_ITollgateGroup)) //
最后生成的类同时实现了 IInterceptorGroup 接口

                            .build()

            ).build().writeTo(mFiler);



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

        }

    }
```

```
    /**
     * Verify inteceptor meta
     *
     * @param element Interceptor taw type
     * @return verify result
     */
    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);
    }
}
```

复制代码

## 九、结尾

ARouter 的实现原理和源码解析都讲得差不多了，文本应该讲得挺全面的了，那么下一篇就再来进入实战篇吧，自己来动手实现一个 ARouter

# 三方库源码笔记（4）-ARouter 自己实现一个？

上一篇文章中对 ARouter 的源码进行了一次全面解析，原理懂得了，那么就也需要进行一次实战才行。对于一个优秀的第三方库，开发者除了要学会如何使用外，更有难度的用法就是去了解实现原理、懂得如何改造甚至自己实现。本文就来自己动手实现一个路由框架，因为自己实现的目的不在于做到和 ARouter 一样功能完善，而只是一个练手项目，目的是在于加深对 ARouter 的原理理解，所以自己的自定义实现就叫做 EasyArouter 吧

EasyArouter 支持同个模块间及跨模块实现 Activity 的跳转，仅需要指定一个字符串 path 即可：

```
    EasyRouter.navigation(EasyRouterPath.PATH_HOME)

复制代码
```

最终实现的效果：


img

EasyArouter 的实现及使用一共涉及以下几个模块：

1. app。即项目的主模块，从这里跳到子模块
2. base。用于在多个模块间共享 path

3. easyrouter-annotation。用于定义和 EasyArouter 实现相关的**注解和 Bean 对象**
4. easyrouter-api。用于定义和 EasyArouter 实现相关的 API 入口
5. easyrouter-processor。用于定义和 EasyArouter 实现相关的注解处理器，在编译阶段使用
6. easyrouter-test。子模块，用于测试 app 模块跳转到子模块是否正常

![img](data:image/svg+xml;utf8,<?xml version="1.0"?><svg xmlns="http://www.w3.org/2000/svg" version="1.1" width="800" height="600"></svg>)

EasyArouter 的实现思路和 ARouter 略有不同。EasyArouter 将同个模块下的所有路由信息通过静态方法块来进行存储并初始化，最终会生成以下的辅助文件：

```
package github.leavesc.easyrouter;

import java.util.HashMap;import java.util.Map;

import github.leavesc.ctrlcv.easyrouter.EasyRouterHomeActivity;import github.leavesc.ctrlcv.easyrouter.EasyRouterSubPageActivity;import github.leavesc.easyrouterannotation.RouterBean;

/**

 * 这是自动生成的代码 by leavesC

 */public class EasyRouterappLoader {

    public static final Map<String, RouterBean> routerMap = new HashMap<>();



    {

        routerMap.put("app/home", new RouterBean(EasyRouterHomeActivity.class, "app/home", "app"));

        routerMap.put("app/subPage", new RouterBean(EasyRouterSubPageActivity.class, "app/subPage", "app"));
```

```
    }

}
```

复制代码

由于静态变量和静态方法块在类被加载前是不会被初始化的，所以也可以做到按需加载。即只有在外部发起跳转到 app 这个模块的请求的时候，EasyArouter 才会去实例化 `EasyRouterappLoader` 类，此时才会去加载 app 模块的所有路由表信息，从而避免了内存浪费

下面再来简单介绍下 EasyArouter 的实现过程

# 一、前置准备

由于路由框架是以模块为单位的，所以同个模块内的路由信息都可以存到同一个辅助文件中，而为了避免多个模块间出现生成的辅助文件重名的情况，所以外部需要主动配置每个模块的特定唯一标识，然后在编译阶段通过 `AbstractProcessor` 拿到这个唯一标识

例如，我为 `easyrouter-test` 这个模块设置的唯一标识就是 **RouterTest**

```
kapt {

    arguments {

        arg("EASYROUTER_MODULE_NAME", "RouterTest")

    }

}
```

复制代码

最终生成的辅助文件对应的包名会是固定的，但类名会包含这个唯一标识。而由于包名和类名的生成规则是有规律的，也方便在运行时拿到这个类，同时这也就要求同个模块下的路由路径 path 必须是属于同个 group

```
package github.leavesc.easyrouter;

import java.util.HashMap;import java.util.Map;
```

```
import github.leavesc.easyrouter_test.EasyRouterTestAActivity;import github.leavesc.easyrouterannotation.RouterBean;

/**

 * 这是自动生成的代码 by leavesC

 */public class EasyRouterRouterTestLoader {

    public static final Map<String, RouterBean> routerMap = new HashMap<>();



    {

        routerMap.put("RouterTest/testA", new RouterBean(EasyRouterTestAActivity.class, "RouterTest/testA", "RouterTest"));

    }

}
```

复制代码@Router` 用于对 `Activity` 进行标注，仅需要设置一个参数 `path` 即可，`path` 包含的第一个单词就是 `group/**

```
 * 作者：leavesC

 * 时间：2020/10/5 22:08

 * 描述：

 * GitHub：https://github.com/leavesC

 */

 @MustBeDocumented@kotlin.annotation.Retention(AnnotationRetention.SOURCE)@Target(AnnotationTarget.CLASS)annotation class Router(val path: String)



data class RouterBean(val targetClass: Class<*>, val path: String, val group: String)
```

复制代码

## 二、注解处理器

声明一个 `EasyRouterProcessor` 类继承于 `AbstractProcessor`，在编译阶段通过扫描代码元素从而拿到 `@Router` 注解的信息

```
/**
 * 作者：leavesC
 * 时间：2020/10/5 22:17
 * 描述：
 * GitHub：https://github.com/leavesC
 */class EasyRouterProcessor : AbstractProcessor() {

    companion object {

        private const val KEY_MODULE_NAME = "EASYROUTER_MODULE_NAME"

        private const val PACKAGE_NAME = "github.leavesc.easyrouter"

        private const val DOC = "这是自动生成的代码 by leavesC"

    }

    private lateinit var elementUtils: Elements

    private lateinit var messager: Messager
```

```
    private lateinit var moduleName: String

    override fun init(processingEnvironment: ProcessingEnvironment) {
        super.init(processingEnvironment)
        elementUtils = processingEnv.elementUtils
        messager = processingEnv.messager
        val options = processingEnv.options
        moduleName = options[KEY_MODULE_NAME] ?: ""
        if (moduleName.isBlank()) {
            messager.printMessage(Diagnostic.Kind.ERROR, "$KEY_MODULE_NAME must not b
e null")
        }
    }

    ...

    override fun getSupportedAnnotationTypes(): MutableSet<String> {
        return mutableSetOf(Router::class.java.canonicalName)
    }

    override fun getSupportedSourceVersion(): SourceVersion {
        return SourceVersion.RELEASE_8
```

```
    }


    override fun getSupportedOptions(): Set<String> {
        return hashSetOf(KEY_MODULE_NAME)
    }


}
```

复制代码

首先需要生成的 `routerMap` 这个用于存储路由表信息的 Map 字段，其 key 值即 path，value 值即 path 对应的页面信息

```
    //生成 routerMap 这个静态常量
    private fun generateSubscriberField(): FieldSpec {
        val subscriberIndex = ParameterizedTypeName.get(
            ClassName.get(Map::class.java),
            ClassName.get(String::class.java),
            ClassName.get(RouterBean::class.java)
        )
        return FieldSpec.builder(subscriberIndex, "routerMap")
            .addModifiers(
                Modifier.PUBLIC,
                Modifier.STATIC,
                Modifier.FINAL
```

```
            )
            .initializer("new ${"$"}T<>()", HashMap::class.java)
            .build()
    }
```

复制代码

之后就需要生成静态方法块。拿到 `@Router` 注解包含的 path 属性，及被注解的类对应的 Class 对象，以此来构建一个 `RouterBean` 对象并存到 `routerMap` 中

```
    //生成静态方法块
    private fun generateInitializerBlock(
        elements: MutableSet<out Element>,
        builder: TypeSpec.Builder
    ) {
        val codeBuilder = CodeBlock.builder()
        elements.forEach {
            val router = it.getAnnotation(Router::class.java)
            val path = router.path
            val group = path.substring(0, path.indexOf("/"))
            codeBuilder.add(
                "routerMap.put(${"$"}S, new ${"$"}T(${"$"}T.class, ${"$"}S, ${"$"}S));
",
                path,
                RouterBean::class.java,
                it.asType(),
```

```
                path,
                group
            )
        }
        builder.addInitializerBlock(
            codeBuilder.build()
        )
    }
```

复制代码

然后在 `process` 方法中完成辅助文件的生成

```
    override fun process(
        mutableSet: MutableSet<out TypeElement>,
        roundEnvironment: RoundEnvironment
    ): Boolean {
        val elements: MutableSet<out Element> =
            roundEnvironment.getElementsAnnotatedWith(Router::class.java)
        if (elements.isNullOrEmpty()) {
            return true
        }
        val typeSpec = TypeSpec.classBuilder("EasyRouter" + moduleName + "Loader")
            .addModifiers(Modifier.PUBLIC)
            .addField(generateSubscriberField())
```

```
            .addJavadoc(DOC)

        generateInitializerBlock(elements, typeSpec)

        val javaFile = JavaFile.builder(PACKAGE_NAME, typeSpec.build())

            .build()

        try {

            javaFile.writeTo(processingEnv.filer)

        } catch (e: Throwable) {

            e.printStackTrace()

        }

        return true

    }
```

复制代码

## 三、EasyRouter

`EasyRouter` 这个单例对象即最终提供给外部的调用入口，总代码行数不到五十行。外部通过调用 `navigation` 方法并传入目标页面 path 来实现跳转，通过 path 来判断其所属 group，并尝试加载其所在模块生成的辅助文件，如果加载成功则能成功跳转，否则就 Toast 提示

```
/**

 * 作者：leavesC

 * 时间：2020/10/5 23:45

 * 描述：

 * GitHub：https://github.com/leavesC

 */object EasyRouter {
```

```
    private const val PACKAGE_NAME = "github.leavesc.easyrouter"

    private lateinit var context: Application

    private val routerByGroupMap = hashMapOf<String, Map<String, RouterBean>>()

    fun init(application: Application) {
        this.context = application
    }

    fun navigation(path: String) {
        val routerBean = getRouterLoader(path)
        if (routerBean == null) {
            Toast.makeText(context, "找不到匹配的路径：$path", Toast.LENGTH_SHORT).show
()
            return
        }
        val intent = Intent(context, routerBean.targetClass)
        intent.flags = Intent.FLAG_ACTIVITY_NEW_TASK
        context.startActivity(intent)
    }
```

```
    private fun getRouterLoader(path: String): RouterBean? {

        val group = path.substring(0, path.indexOf("/"))

        val map = routerByGroupMap[group]

        if (map == null) {

            var routerMap: Map<String, RouterBean>? = null

            try {

                val classPath = PACKAGE_NAME + "." + "EasyRouter" + group + "Loader"

                val clazz = Class.forName(classPath)

                val instance = clazz.newInstance()

                val routerMapField = clazz.getDeclaredField("routerMap")

                routerMap =

                    (routerMapField.get(instance) as? Map<String, RouterBean>) ?: hash
MapOf()

                routerByGroupMap[group] = routerMap

            } catch (e: Throwable) {

                e.printStackTrace()

            } finally {

                if (routerMap == null) {

                    routerByGroupMap[group] = hashMapOf()

                }

            }

        }

        return routerByGroupMap[group]?.get(path)
```

```
    }



}
```

复制代码

## 四、结尾

由于只是为了加深对 ARouter 的实现原理的理解，所以才来尝试实现 EasyArouter，也不打算实现得多么功能齐全，但对于一些读者来说我觉得还是有参考价值的 AndroidOpenSourceDemo

# 三方库源码笔记（5）-LeakCanary 源码详解

LeakCanary 是由 Square 公司开源的用于 Android 的内存泄漏检测工具，可以帮助开发者发现内存泄露情况并且找出泄露源头，有助于减少 `OutOfMemoryError` 情况的发生。在目前的应用开发中也算作是性能优化的一个重要实现途径，很多面试官在考察性能优化时都会问到 LeakCanary 的实现原理

本文就基于其当前（2020/10/06）的最新一次提交来进行源码分析，具体的 Git 版本节点是：**9f62126e**，来了解 LeakCanary 的整体运行流程和实现原理

## 一、支持的内存泄露类型

我们经常说 LeakCanary 能检测到应用内发生的内存泄露，那么它到底具体支持什么类型的内存泄露情况呢？LeakCanary 官网有对此进行介绍：

LeakCanary automatically detects leaks of the following objects:

- destroyed `Activity` instances
- destroyed `Fragment` instances
- destroyed fragment `View` instances
- cleared `ViewModel` instances

我们也可以从 LeakCanary 的 `AppWatcher.Config` 这个类找到答案。Config 类用于配置是否开启内存检测，从其配置项就可以看出来 leakcanary 支持：**Activity、Fragment、FragmentView、ViewModel** 等四种类型

```
data class Config(

    /**

     * Whether AppWatcher should automatically watch destroyed activity instances.

     *

     * Defaults to true.

     */

    val watchActivities: Boolean = true,



    /**

     * Whether AppWatcher should automatically watch destroyed fragment instances.

     *

     * Defaults to true.

     */

    val watchFragments: Boolean = true,



    /**

     * Whether AppWatcher should automatically watch destroyed fragment view instance
s.

     *

     * Defaults to true.

     */
```

```
    val watchFragmentViews: Boolean = true,



    /**
     * Whether AppWatcher should automatically watch cleared [androidx.lifecycle.View
Model]
     * instances.
     *
     * Defaults to true.
     */
    val watchViewModels: Boolean = true,



    /**
     * How long to wait before reporting a watched object as retained.
     *
     * Default to 5 seconds.
     */
    val watchDurationMillis: Long = TimeUnit.SECONDS.toMillis(5),



    /**
     * Deprecated, this didn't need to be a part of the API.
     * Used to indicate whether AppWatcher should watch objects (by keeping weak refe
rences to
     * them). Currently a no-op.
     *
```

```
    * If you do need to stop watching objects, simply don't pass them to [objectWatcher].

    */

    @Deprecated("This didn't need to be a part of LeakCanary's API. No-Op.")

    val enabled: Boolean = true

  )
```

复制代码

## 二、初始化

如今，我们在项目中引入 LeakCanary 只需要添加如下依赖即可，无须任何的初始化行为等附加操作，当应用启动时 LeakCanary 就会自动启动并开始监测了

```
dependencies {

  // debugImplementation because LeakCanary should only run in debug builds.

  debugImplementation 'com.squareup.leakcanary:leakcanary-android:2.4'

}
```

复制代码

一般来说，像这类第三方库都是需要由外部传入一个 `ApplicationContext` 对象进行初始化并启动的，LeakCanary 的 1.x 版本也是如此，但在 2.x 版本中，LeakCanary 将初始过程交由 `AppWatcherInstaller` 这个 `ContentProvider` 来自动完成

```
internal sealed class AppWatcherInstaller : ContentProvider() {



    /**
```

```
     * [MainProcess] automatically sets up the LeakCanary code that runs in the main
app process.
     */
    internal class MainProcess : AppWatcherInstaller()


    /**
     * When using the `leakcanary-android-process` artifact instead of `leakcanary-an
droid`,
     * [LeakCanaryProcess] automatically sets up the LeakCanary code
     */
    internal class LeakCanaryProcess : AppWatcherInstaller()


    override fun onCreate(): Boolean {
        val application = context!!.applicationContext as Application
        AppWatcher.manualInstall(application)
        return true
    }


    ...
}
```

复制代码

由于 `ContentProvider` 会在 `Application` 被创建之前就由系统调用其 `onCreate()` 方法来完成初始化，所以 LeakCanary 通过 `AppWatcherInstaller` 就可以拿到 `Context` 来完成初始化并随应用一起启动，通过这种方式就简化了使用者的引

入成本。而且由于我们的引用方式是 `debugImplementation`，所以**正式版本**会自动移除对 LeakCanary 的所有引用，进一步简化了引入成本

Jetpack 也包含了一个组件来实现**通过 ContentProvider 来完成初始化的逻辑**：AppStartup。在实现思路上两者很类似，但是如果每个三方库都通过自定义 `ContentProvider` 来实现初始化的话，那么应用的启动速度就会很感人了吧 :joy:，所以 Jetpack 官方推出的 `AppStartup` 应该是以后的主流才对

`AppWatcherInstaller` 最终会将 `Application` 对象传给 `InternalAppWatcher` 的 `install(Application)` 方法

```
/**

 * Note: this object must load fine in a JUnit environment

 */internal object InternalAppWatcher {



  ...



  val objectWatcher = ObjectWatcher(

      clock = clock,

      checkRetainedExecutor = checkRetainedExecutor,

      isEnabled = { true }

  )



  fun install(application: Application) {

    //检测是否在 main 线程

    checkMainThread()

    //避免重复初始化

    if (this::application.isInitialized) {
```

```
      return
    }
    InternalAppWatcher.application = application
    if (isDebuggableBuild) {
      SharkLog.logger = DefaultCanaryLog()
    }
    //拿到默认配置，默认四种类型都进行检测
    val configProvider = { AppWatcher.config }
    //检测 Activity
    ActivityDestroyWatcher.install(application, objectWatcher, configProvider)
    //检测 Fragment、FragmentView、ViewModel
    FragmentDestroyWatcher.install(application, objectWatcher, configProvider)
    onAppWatcherInstalled(application)
  }

  ...

}
复制代码
```

LeakCanary 具体进行内存泄露检测的逻辑可以分为三类：

- ObjectWatcher。检测 Object 的内存泄露情况
- ActivityDestroyWatcher。检测 Activity 的内存泄露情况

- FragmentDestroyWatcher。检测 Fragment、FragmentView、ViewModel 的内存泄露情况

当中，`ActivityDestroyWatcher` 和 `FragmentDestroyWatcher` 都需要依靠 `ObjectWatcher` 来完成，因为 `Activity、Fragment、FragmentView、ViewModel` 本质上都属于不同类型的 `Object`

## 三、ObjectWatcher：检测任意对象

我们知道，当一个对象不再被我们引用时，如果该对象由于代码错误或者其它原因导致迟迟无法被系统回收，此时就是发生了内存泄露。那么 LeakCanary 是怎么知道应用是否发生了内存泄露呢？

这个可以依靠引用队列 `ReferenceQueue` 来实现。先来看个小例子

```
/**

 * 作者：leavesC

 * 时间：2020/10/06 14:26

 * 描述：

 * GitHub：https://github.com/leavesC

 */fun main() {

    val referenceQueue = ReferenceQueue<Pair<String, Int>?>()

    var pair: Pair<String, Int>? = Pair("name", 24)

    val weakReference = WeakReference(pair, referenceQueue)



    println(referenceQueue.poll()) //null



    pair = null
```

```
    System.gc()

    //GC 后休眠一段时间，等待 pair 被回收

    Thread.sleep(4000)



    println(referenceQueue.poll()) //java.lang.ref.WeakReference@d716361

}
```

复制代码

可以看到，在 GC 过后 `referenceQueue.poll()` 的返回值变成了**非 null**，这是由于 `WeakReference` 和 `ReferenceQueue` 的一个组合特性导致的：在声明一个 `WeakReference` 对象时如果同时传入了 `ReferenceQueue` 作为构造参数的话，那么当 `WeakReference` 持有的对象被 GC 回收时，JVM 就会把这个**弱引用**存入与之关联的引用队列之中。依靠这个特性，我们就可以实现内存泄露的检测了

例如，当用户按返回键退出 Activity 时，正常情况下该 Activity 对象应该在不久后就被系统回收，我们可以监听 Activity 的 `onDestroy` 回调，在回调时把 Activity 对象保存到和 `ReferenceQueue` 关联的 `WeakReference` 中，在一段时间后（可以主动触发几次 GC）检测 `ReferenceQueue` 中是否有值，如果一直为 null 的话就说明发生了内存泄露。LeakCanary 就是通过这种方法来实现的

`ObjectWatcher` 中就封装了上述逻辑，这里来看看其实现逻辑

`ObjectWatcher` 的起始方法是 `watch(Any, String)`，该方法就用于监听指定对象

```
    /**

     * References passed to [watch].

     * 用于保存要监听的对象，mapKey 是该对象的唯一标识、mapValue 是该对象的弱引用

     */

    private val watchedObjects = mutableMapOf<String, KeyedWeakReference>()



    //KeyedWeakReference 关联的引用队列
```

```
    private val queue = ReferenceQueue<Any>()



    /**
     * Watches the provided [watchedObject].
     *
     * @param description Describes why the object is watched.
     */
    @Synchronized
    fun watch(watchedObject: Any, description: String) {
        if (!isEnabled()) {
            return
        }
        removeWeaklyReachableObjects()
        //为 watchedObject 生成一个唯一标识
        val key = UUID.randomUUID().toString()
        val watchUptimeMillis = clock.uptimeMillis()
        //创建 watchedObject 关联的弱引用
        val reference = KeyedWeakReference(watchedObject, key, description, watchUpti
meMillis, queue)
        ...
        watchedObjects[key] = reference
        checkRetainedExecutor.execute {
            moveToRetained(key)
```

```
        }

    }
```

复制代码

`watch()` 方法的主要逻辑：

1. 为每个 `watchedObject` 生成一个**唯一标识 key**，通过该 key 构建一个 `watchedObject` 的弱引用 `KeyedWeakReference`，将该弱引用保存到 `watchedObjects` 中。`ObjectWatcher` 可以先后监测多个对象，每个对象都会先被存入到 `watchedObjects` 中
2. 外部通过传入的 `checkRetainedExecutor` 来指定检测内存泄露的触发时机，通过 `moveToRetained` 方法来判断是否真的发生了内存泄露

`KeyedWeakReference` 是一个自定义的 `WeakReference` 子类，包含一个唯一 key 来标识特定对象，也包含一个 `retainedUptimeMillis` 字段用来标记是否发生了内存泄露

```
class KeyedWeakReference(

        referent: Any,

        val key: String,

        val description: String,

        val watchUptimeMillis: Long,

        referenceQueue: ReferenceQueue<Any>

) : WeakReference<Any>(

        referent, referenceQueue

) {

    /**

     * Time at which the associated object ([referent]) was considered retained, or -
1 if it hasn't
```

```
     * been yet.
     * 用于标记 referent 是否还未被回收，是的话则值不为 -1
     */
    @Volatile
    var retainedUptimeMillis = -1L


    companion object {
        @Volatile
        @JvmStatic
        var heapDumpUptimeMillis = 0L
    }


}
```

复制代码

`moveToRetained` 方法就用于判断指定 key 关联的对象是否已经泄露，如果没有泄露则移除对该对象的弱引用，有泄露的话则更新其 `retainedUptimeMillis` 值，以此来标记其发生了泄露，并同时通过回调 `onObjectRetainedListeners` 来分析内存泄露链

```
    @Synchronized
    private fun moveToRetained(key: String) {
        removeWeaklyReachableObjects()
        val retainedRef = watchedObjects[key]
        if (retainedRef != null) {
```

```
            //记录当前时间
            retainedRef.retainedUptimeMillis = clock.uptimeMillis()
            onObjectRetainedListeners.forEach { it.onObjectRetained() }
        }
    }

    //如果判断到一个对象没有发生内存泄露，那么就移除对该对象的弱引用
    //此方法会先后调用多次
    private fun removeWeaklyReachableObjects() {
        // WeakReferences are enqueued as soon as the object to which they point to b
ecomes weakly
        // reachable. This is before finalization or garbage collection has actually
happened.
        var ref: KeyedWeakReference?
        do {
            ref = queue.poll() as KeyedWeakReference?
            if (ref != null) {
                //如果 ref 不为 null，说明 ref 关联的对象没有发生内存泄露，那么就移除对该对
象的引用
                watchedObjects.remove(ref.key)
            }
        } while (ref != null)
    }
```

复制代码

## 四、ActivityDestroyWatcher：检测 Activity

理解了 `ObjectWatcher` 的流程后来看 `ActivityDestroyWatcher` 就会比较简单了。`ActivityDestroyWatcher` 会向 `Application` 注册一个 `ActivityLifecycleCallbacks` 回调，当收到每个 Activity 执行了 `onDestroy` 的回调后，就会将将 Activity 对象转交由 `ObjectWatcher` 来进行监听

```
internal class ActivityDestroyWatcher private constructor(private val objectWatcher:
 ObjectWatcher, private val configProvider: () -> Config) {



    private val lifecycleCallbacks = object : Application.ActivityLifecycleCallbacks
 by noOpDelegate() {

        override fun onActivityDestroyed(activity: Activity) {

            if (configProvider().watchActivities) {

                objectWatcher.watch(activity, "${activity::class.java.name} received
Activity#onDestroy() callback"

                )

            }

        }

    }



    companion object {

        fun install(application: Application, objectWatcher: ObjectWatcher, configPro
vider: () -> Config) {

            val activityDestroyWatcher = ActivityDestroyWatcher(objectWatcher, config
Provider)

            application.registerActivityLifecycleCallbacks(activityDestroyWatcher.lif
ecycleCallbacks)

        }
```

```
    }



}
```

复制代码

## 五、FragmentDestroyWatcher：检测 Fragment

做 Android 应用开发的应该都知道，现在 Google 提供的基础依赖包分为了 **Support** 和 **AndroidX** 两种，**Support** 版本已经不再维护，主流的都是使用 **AndroidX** 了。而 LeakCanary 为了照顾老项目，就贴心的为这两种版本分别提供了 Fragment 的内存检测功能

`FragmentDestroyWatcher` 可以看做是一个分发器，它会根据外部环境的不同来选择不同的检测手段，其主要逻辑是：

- 系统版本大于等于 8.0。使用 AndroidOFragmentDestroyWatcher 来检测 Fragment、FragmentView 的内存泄露
- 开发者使用的是 Support 包。使用 AndroidSupportFragmentDestroyWatcher 来检测 Fragment、FragmentView 的内存泄露
- 开发者使用的是 AndroidX 包。使用 AndroidXFragmentDestroyWatcher 来检测 Fragment、FragmentView、ViewModel 的内存泄露
- 通过反射 Class.forName 来判断开发者使用的是 Support 包还是 AndroidX 包
- 由于 Fragment 都需要被挂载在 Activity 上，所有向 Application 注册一个 ActivityLifecycleCallback，每当有 Activity 被创建时就监听该 Activity 内可能存在的 Fragment

  这里令我很疑惑的一个点就是：当系统版本大于等于 8.0 时，AndroidOFragmentDestroyWatcher 不就会和 AndroidSupportFragmentDestroyWatcher 或者 AndroidXFragmentDestroyWatcher 重复了吗？这算咋回事:joy:

```
internal object FragmentDestroyWatcher {
```

```
  private const val ANDROIDX_FRAGMENT_CLASS_NAME = "androidx.fragment.app.Fragment"
  private const val ANDROIDX_FRAGMENT_DESTROY_WATCHER_CLASS_NAME =
    "leakcanary.internal.AndroidXFragmentDestroyWatcher"

  // Using a string builder to prevent Jetifier from changing this string to Android
X Fragment
  @Suppress("VariableNaming", "PropertyName")
  private val ANDROID_SUPPORT_FRAGMENT_CLASS_NAME =
    StringBuilder("android.").append("support.v4.app.Fragment")
        .toString()
  private const val ANDROID_SUPPORT_FRAGMENT_DESTROY_WATCHER_CLASS_NAME =
    "leakcanary.internal.AndroidSupportFragmentDestroyWatcher"

  fun install(
    application: Application,
    objectWatcher: ObjectWatcher,
    configProvider: () -> AppWatcher.Config
  ) {
    val fragmentDestroyWatchers = mutableListOf<(Activity) -> Unit>()

    if (SDK_INT >= O) {
      fragmentDestroyWatchers.add(
```

```
      AndroidOFragmentDestroyWatcher(objectWatcher, configProvider)
  )
}

//AndroidX
getWatcherIfAvailable(
    ANDROIDX_FRAGMENT_CLASS_NAME,
    ANDROIDX_FRAGMENT_DESTROY_WATCHER_CLASS_NAME,
    objectWatcher,
    configProvider
)?.let {
  fragmentDestroyWatchers.add(it)
}

//Support
getWatcherIfAvailable(
    ANDROID_SUPPORT_FRAGMENT_CLASS_NAME,
    ANDROID_SUPPORT_FRAGMENT_DESTROY_WATCHER_CLASS_NAME,
    objectWatcher,
    configProvider
)?.let {
  fragmentDestroyWatchers.add(it)
}
```

```
    if (fragmentDestroyWatchers.size == 0) {
      return
    }

    application.registerActivityLifecycleCallbacks(object : Application.ActivityLife
cycleCallbacks by noOpDelegate() {
      override fun onActivityCreated(
        activity: Activity,
        savedInstanceState: Bundle?
      ) {
        for (watcher in fragmentDestroyWatchers) {
          watcher(activity)
        }
      }
    })
  }

  ...
}
```

复制代码

由于 `AndroidXFragmentDestroyWatcher`、`AndroidSupportFragmentDestroyWatcher`、`AndroidOFragmentDestroyWatcher` 在逻辑上很类似，且

就 AndroidXFragmentDestroyWatcher 同时提供了 ViewModel 内存泄露的检测功能，所以这里只看 AndroidXFragmentDestroyWatcher 就行

AndroidXFragmentDestroyWatcher 的主要逻辑是：

- 在 invoke 方法里向 Activity 的 FragmentManager 以及 childFragmentManager 注册一个 FragmentLifecycleCallback，通过该回调拿到 onFragmentViewDestroyed 和 onFragmentDestroyed 的事件通知，收到通知时就通过 ObjectWatcher 启动检测
- 在 onFragmentCreated 回调里通过 ViewModelClearedWatcher 来启动和 Fragment 关联的 ViewModel 的内存泄露检测逻辑
- 在 invoke 方法里通过 ViewModelClearedWatcher 来启动和 Activity 关联的 ViewModel 的内存泄露检测

```
internal class AndroidXFragmentDestroyWatcher(

  private val objectWatcher: ObjectWatcher,

  private val configProvider: () -> Config

) : (Activity) -> Unit {



  private val fragmentLifecycleCallbacks = object : FragmentManager.FragmentLifecycl
eCallbacks() {



    override fun onFragmentCreated(

      fm: FragmentManager,

      fragment: Fragment,

      savedInstanceState: Bundle?

    ) {

      ViewModelClearedWatcher.install(fragment, objectWatcher, configProvider)
```

```
    }

    override fun onFragmentViewDestroyed(
      fm: FragmentManager,
      fragment: Fragment
    ) {
      val view = fragment.view
      if (view != null && configProvider().watchFragmentViews) {
        objectWatcher.watch(
            view, "${fragment::class.java.name} received Fragment#onDestroyView() cal
lback " +
            "(references to its views should be cleared to prevent leaks)"
        )
      }
    }

    override fun onFragmentDestroyed(
      fm: FragmentManager,
      fragment: Fragment
    ) {
      if (configProvider().watchFragments) {
        objectWatcher.watch(
            fragment, "${fragment::class.java.name} received Fragment#onDestroy() cal
lback"
```

```
        )
      }
    }
  }

  override fun invoke(activity: Activity) {
    if (activity is FragmentActivity) {
      val supportFragmentManager = activity.supportFragmentManager
      supportFragmentManager.registerFragmentLifecycleCallbacks(fragmentLifecycleCal
lbacks, true)
      ViewModelClearedWatcher.install(activity, objectWatcher, configProvider)
    }
  }
}
```

复制代码

Fragment 和 FragmentView 走向 `Destroyed` 时，正常情况下它们都是不会被复用的，应该会很快就被 GC 回收，且它们本质上都只是一种对象，所以直接使用 `ObjectWatcher` 进行检测即可

## 六、ViewModelClearedWatcher：检测 ViewModel

和 Fragment、FragmentView 相比，ViewModel 就比较特殊了，由于可能存在一个 Activity 和多个 Fragment 同时持有一个 ViewModel 实例的情况，而 leakcanary 无法知道 ViewModel 到底是同时被几个持有者所持有，所以无法通过单独一个 Activity 和 Fragment 的 `Destroyed` 回调来启动对 ViewModel 的检测。幸好 ViewMode 也提供了 `onCleared()` 的回调事件，leakcanary 就通过该回调来知道 ViewModel 是什么时候需要被回收。对 ViewModel 的实现原理不清楚的同学可以看我的这篇文章：从源码看 Jetpack（6）-ViewModel 源码详解

`ViewModelClearedWatcher` 的主要逻辑是：

- ViewModelClearedWatcher 继承于 ViewModel，当拿到 ViewModelStoreOwner 实例（Activity 或者 Fragment）后，就创建一个和该实例绑定的 ViewModelClearedWatcher 对象
- ViewModelClearedWatcher 通过反射获取到 ViewModelStore 中的 mMap 变量，该变量就存储了所有的 Viewmodel 实例
- 当 ViewModelClearedWatcher 的 onCleared() 方法被回调了，就说明了所有和 Activity 或者 Fragment 绑定的 ViewModel 实例都不再被需要了，此时就可以开始监测所有的 ViewModel 实例了

```
internal class ViewModelClearedWatcher(

        storeOwner: ViewModelStoreOwner,

        private val objectWatcher: ObjectWatcher,

        private val configProvider: () -> Config

) : ViewModel() {



    private val viewModelMap: Map<String, ViewModel>?



    init {

        // We could call ViewModelStore#keys with a package spy in androidx.lifecycle
 instead,

        // however that was added in 2.1.0 and we support AndroidX first stable relea
se. viewmodel-2.0.0

        // does not have ViewModelStore#keys. All versions currently have the mMap fi
eld.

        viewModelMap = try {

            val mMapField = ViewModelStore::class.java.getDeclaredField("mMap")
```

```
            mMapField.isAccessible = true

            @Suppress("UNCHECKED_CAST")

            mMapField[storeOwner.viewModelStore] as Map<String, ViewModel>

        } catch (ignored: Exception) {

            null

        }

    }



    override fun onCleared() {

        if (viewModelMap != null && configProvider().watchViewModels) {

            viewModelMap.values.forEach { viewModel ->

                objectWatcher.watch(

                        viewModel, "${viewModel::class.java.name} received ViewModel#o
nCleared() callback"

                )

            }

        }

    }



    companion object {

        fun install(storeOwner: ViewModelStoreOwner, objectWatcher: ObjectWatcher, co
nfigProvider: () -> Config) {

            val provider = ViewModelProvider(storeOwner, object : Factory {

              @Suppress("UNCHECKED_CAST")
```

```
            override fun <T : ViewModel?> create(modelClass: Class<T>): T =

                    ViewModelClearedWatcher(storeOwner, objectWatcher, configProvide
r) as T

            })

            provider.get(ViewModelClearedWatcher::class.java)

        }

    }

}
```

复制代码

## 七、检测到内存泄露后的流程

我们不可能在 Activity 刚被回调了 `onDestroy` 方法就马上来判断 `ReferenceQueue` 中是否有值，因为 JVM 的 GC 时机是不确定的，Activity 对象可能不会那么快就被回收，所以需要延迟一段时间后再来检测。而即使延迟检测了，也可能会存在**应用没有发生内存泄露只是系统还未执行 GC** 的情况，所以就需要去主动触发 GC，经过几轮检测后才可以确定当前应用是否的确发生了内存泄露

这里就来看下具体的检测流程

`ObjectWatcher` 对象包含了一个 `Executor` 参数：`checkRetainedExecutor`。检测操作的触发时机就取决于向 `checkRetainedExecutor` 提交的任务在什么时候会被执行

```
class ObjectWatcher constructor(private val clock: Clock, private val checkRetainedE
xecutor: Executor,

        /**

         * Calls to [watch] will be ignored when [isEnabled] returns false

         */

        private val isEnabled: () -> Boolean = { true }
```

```
) {

    ...

    /**
     * Watches the provided [watchedObject].
     *
     * @param description Describes why the object is watched.
     */
    @Synchronized
    fun watch(
            watchedObject: Any,
            description: String
    ) {
        if (!isEnabled()) {
            return
        }
        removeWeaklyReachableObjects()
        val key = UUID.randomUUID()
                .toString()
        val watchUptimeMillis = clock.uptimeMillis()
        val reference =
```

```
            KeyedWeakReference(watchedObject, key, description, watchUptimeMillis,
 queue)

        SharkLog.d {

            "Watching " +

                    (if (watchedObject is Class<*>) watchedObject.toString() else "ins
tance of ${watchedObject.javaClass.name}") +

                    (if (description.isNotEmpty()) " ($description)" else "") +

                    " with key $key"

        }

        watchedObjects[key] = reference

        //重点

        checkRetainedExecutor.execute {

            moveToRetained(key)

        }

    }



    //判断 key 关联的对象是否已经泄露

    //是的话则将更新其 retainedUptimeMillis 值，以此来标记其发生了泄露

    @Synchronized

    private fun moveToRetained(key: String) {

        removeWeaklyReachableObjects()

        val retainedRef = watchedObjects[key]

        if (retainedRef != null) {

            retainedRef.retainedUptimeMillis = clock.uptimeMillis()
```

```
            //重点，向外发出可能有内存泄露的通知
            onObjectRetainedListeners.forEach { it.onObjectRetained() }
        }
    }

    ...

}
```

复制代码

`ObjectWatcher` 对象又是在 `InternalAppWatcher` 里初始化的，`checkRetainedExecutor` 在收到任务后会通过 `Handler` 来延时五秒执行

```
internal object InternalAppWatcher {

  ...

  private val mainHandler by lazy {
    Handler(Looper.getMainLooper())
  }

  private val checkRetainedExecutor = Executor {
    mainHandler.postDelayed(it, AppWatcher.config.watchDurationMillis)
  }
```

```
  val objectWatcher = ObjectWatcher(
      clock = clock,
      checkRetainedExecutor = checkRetainedExecutor,
      isEnabled = { true }
  )

  ...

}
```

复制代码

`ObjectWatcher` 的 `moveToRetained` 方法又会通过 `onObjectRetained` 向外发出通知：**当前可能发生了内存泄露**。`InternalLeakCanary` 会收到这个通知，然后交由 `HeapDumpTrigger` 来进行检测

```
internal object InternalLeakCanary : (Application) -> Unit, OnObjectRetainedListener
 {

    private lateinit var heapDumpTrigger: HeapDumpTrigger

    override fun onObjectRetained() = scheduleRetainedObjectCheck()

    fun scheduleRetainedObjectCheck() {
        if (this::heapDumpTrigger.isInitialized) {
```

```
            heapDumpTrigger.scheduleRetainedObjectCheck()

        }

    }


    ...



}
```

复制代码

当 LeakCanary 判定当前真的存在内存泄露时，就会进行 **DumpHeap**，找到泄露对象的引用链，而这个操作是比较**费时费内存**的，可能会直接导致应用页面无响应，所以 LeakCanary 进行 DumpHeap 前会有许多前置检查操作和前置条件，就是为了尽量减少 DumpHeap 次数以及在 DumpHeap 时尽量减少对开发人员的干扰

`heapDumpTrigger` 的 `scheduleRetainedObjectCheck()` 方法的主要逻辑是：

1. 获取当前还未回收的对象个数 retainedKeysCount。如果个数大于 0，则先主动触发 GC，尽量尝试回收对象，避免误判，然后执行第二步；如果个数为 0，那么流程就结束了
2. GC 过后再次更新 retainedKeysCount 值，如果对象都被回收了（即 retainedKeysCount 值为 0），那么流程就结束了，否则就执行第三步
3. 如果 retainedKeysCount 小于阈值 5，且当前“应用处于前台”或者是“应用处于后台但退到后台的时间还未超出五秒”，那么就启动一个定时任务，在二十秒后重新执行第一步，否则执行第四步
4. 如果上一次 DumpHeap 离现在不足一分钟，那么就启动一个定时任务，满一分钟后重新执行第一步，否则执行第五步
5. 此时各个条件都满足了，已经可以确定发生了内存泄漏，去执行 DumpHeap

```
internal class HeapDumpTrigger(
```

```
        private val application: Application,
        private val backgroundHandler: Handler,
        private val objectWatcher: ObjectWatcher,
        private val gcTrigger: GcTrigger,
        private val heapDumper: HeapDumper,
        private val configProvider: () -> Config
) {

    ...

    fun scheduleRetainedObjectCheck(
            delayMillis: Long = 0L
    ) {
        val checkCurrentlyScheduledAt = checkScheduledAt
        if (checkCurrentlyScheduledAt > 0) {
            //如果当前已经在进行检测了，则直接返回
            return
        }
        checkScheduledAt = SystemClock.uptimeMillis() + delayMillis
        backgroundHandler.postDelayed({
          checkScheduledAt = 0
          checkRetainedObjects()
        }, delayMillis)
```

```
    }

    private fun checkRetainedObjects() {
        val iCanHasHeap = HeapDumpControl.iCanHasHeap()

        val config = configProvider()

        if (iCanHasHeap is Nope) {
            if (iCanHasHeap is NotifyingNope) {
                // Before notifying that we can't dump heap, let's check if we still h
ave retained object.
                var retainedReferenceCount = objectWatcher.retainedObjectCount

                if (retainedReferenceCount > 0) {
                    gcTrigger.runGc()
                    retainedReferenceCount = objectWatcher.retainedObjectCount
                }

                val nopeReason = iCanHasHeap.reason()
                val wouldDump = !checkRetainedCount(
                        retainedReferenceCount, config.retainedVisibleThreshold, nopeR
eason
                )
```

```
                if (wouldDump) {
                    val uppercaseReason = nopeReason[0].toUpperCase() + nopeReason.sub
string(1)
                    onRetainInstanceListener.onEvent(DumpingDisabled(uppercaseReaso
n))
                    showRetainedCountNotification(
                            objectCount = retainedReferenceCount,
                            contentText = uppercaseReason
                    )
                }
            } else {
                SharkLog.d {
                    application.getString(
                            R.string.leak_canary_heap_dump_disabled_text, iCanHasHeap.
reason()
                    )
                }
            }
            return
        }


        //获取当前还未回收的对象个数
        var retainedReferenceCount = objectWatcher.retainedObjectCount
```

```
        if (retainedReferenceCount > 0) {

            //主动触发 GC，尽量尝试回收对象，避免误判

            gcTrigger.runGc()

            retainedReferenceCount = objectWatcher.retainedObjectCount

        }



        if (checkRetainedCount(retainedReferenceCount, config.retainedVisibleThresho
ld))

            return



        val now = SystemClock.uptimeMillis()

        val elapsedSinceLastDumpMillis = now - lastHeapDumpUptimeMillis



        //如果上一次 DumpHeap 离现在不足一分钟，那么就启动一个定时任务，满一分钟后再次检查

        if (elapsedSinceLastDumpMillis < WAIT_BETWEEN_HEAP_DUMPS_MILLIS) {

            onRetainInstanceListener.onEvent(DumpHappenedRecently)

            showRetainedCountNotification(

                    objectCount = retainedReferenceCount,

                    contentText = application.getString(R.string.leak_canary_notifica
tion_retained_dump_wait)

            )

            scheduleRetainedObjectCheck(

                    delayMillis = WAIT_BETWEEN_HEAP_DUMPS_MILLIS - elapsedSinceLastDum
pMillis
```

```
            )
            return
        }

        dismissRetainedCountNotification()
        //各个条件都满足了，已经可以确定发生了内存泄漏，去执行 DumpiHeap
        dumpHeap(retainedReferenceCount, retry = true)
    }

    /**
     * 判断当前是否符合 DumpHeap 的条件，符合的话返回 false
     * @param retainedKeysCount 当前还未回收的对象个数
     * @param retainedVisibleThreshold 触发 DumpHeap 的阈值
     * 只有当 retainedKeysCount 大于等于 retainedVisibleThreshold 时才会触发 DumpHeap，默认值是 5
     * @param nopeReason
     */
    private fun checkRetainedCount(
            retainedKeysCount: Int,
            retainedVisibleThreshold: Int,
            nopeReason: String? = null
    ): Boolean {
        //用于标记本次检测相对上次，未回收的对象个数是否发生了变化
```

```
        val countChanged = lastDisplayedRetainedObjectCount != retainedKeysCount
        lastDisplayedRetainedObjectCount = retainedKeysCount
        if (retainedKeysCount == 0) {
            if (countChanged) {
                //如果 retainedKeysCount 为 0，且值相对上次检测减少了，则说明有对象被回收了
                SharkLog.d { "All retained objects have been garbage collected" }
                onRetainInstanceListener.onEvent(NoMoreObjects)
                showNoMoreRetainedObjectNotification()
            }
            return true
        }

        //应用是否还在前台
        val applicationVisible = applicationVisible
        val applicationInvisibleLessThanWatchPeriod = applicationInvisibleLessThanWa
tchPeriod

        ...

        if (retainedKeysCount < retainedVisibleThreshold) { //还未达到阈值
            if (applicationVisible || applicationInvisibleLessThanWatchPeriod) {
                if (countChanged) {
                    onRetainInstanceListener.onEvent(BelowThreshold(retainedKeysCoun
t))
```

```
            }

            //在通知栏显示当前未回收的对象个数

            showRetainedCountNotification(

                    objectCount = retainedKeysCount,

                    contentText = application.getString(

                            R.string.leak_canary_notification_retained_visible, re
tainedVisibleThreshold

                    )

            )

            //retainedKeysCount 还未达到阈值，且当前“应用处于前台”或者是“应用处于后台但退到后台的时间还未超出五秒”

            //此时就启动一个定时任务，在二十秒后重新再检测一遍

            scheduleRetainedObjectCheck(

                    delayMillis = WAIT_FOR_OBJECT_THRESHOLD_MILLIS

            )

            return true

        }

    }

    return false

  }



 ...



}
```

复制代码

更后面的流程就涉及具体的 DumpHeap 操作了，这里就不再展开了，因为我也不太懂，后续有机会再单独写一篇文章来介绍了~~

# 八、小提示

## 1、检测任意对象

除了 LeakCanary 默认支持的四种类型外，我们还可以主动检测任意对象。例如，可以检测 Service：

```
class MyService : Service {



  // ...



  override fun onDestroy() {

    super.onDestroy()

    AppWatcher.objectWatcher.watch(

      watchedObject = this,

      description = "MyService received Service#onDestroy() callback"

    )

  }

}
```

复制代码

## 2、更改配置项

LeakCanary 提供的默认配置项大多数情况已经很适合我们在项目中直接使用了，而如果我们想要更改 LeakCanary 的默认配置项（例如不希望检测 FragmentView），可以在 Application 中进行更改：

```
class DebugExampleApplication : Application() {



  override fun onCreate() {

    super.onCreate()

    AppWatcher.config = AppWatcher.config.copy(watchFragmentViews = false)

  }

}
```

复制代码

由于 LeakCanary 的引用方式是 `debugImplementation`，在 `releas` 环境下是引用不到 LeakCanary 的，所以为了避免在生成 `release` 包时需要主动来删除这行配置项，需要将 `DebugExampleApplication` 放到 `src/debug/java` 文件夹中

## 九、结尾

可以看出 **Activity、Fragment、FragmentView、ViewModel** 等四种类型的内存检测都是需要依靠 `ObjectWatcher` 来完成的，因为这四种类型本质上都是属于不同的对象。而 `ObjectWatcher` 需要依靠引用队列 `ReferenceQueue` 来实现，因此 LeakCanary 的基本实现基础就是来源于 Java 的原生特性

LeakCanary 的整体源码讲得也差不多了，后边就再来写一篇关于**内存泄露**的扩展阅读

# 三方库源码笔记（6）-LeakCanary 扩展阅读

上篇文章对 LeakCanary 进行了一次比较全面的源码解析，按流程来说本篇文章应该是属于实战篇的，可是由于某些原因就不打算写实战内容了（其实就是自己有点菜，写不太出来），就还是来写一篇关于内存泄露相关的扩展阅读吧

Java 的一个很显著的优点就在于内存自动回收机制，Java 通过垃圾收集器(Garbage Collection，GC)来自动管理内存的回收过程，而无需开发者来主动释放内存。这种自动化行为有效地节省了开发人员的开发成本，但也让一些开发者误以为 Java 就不存在**内存泄漏**的问题了，或者是误认为内存泄露应该是 GC 或者 JVM 层面来关心和解决的问题。这种想法是不正确的，因为内存泄露大多时候是由于程序本身存在缺陷而导致的，GC 和 JVM 并无法精准理解程序的实现初衷，所以还是需要由开发人员来主动解决问题

# 一、内存泄露和内存溢出

**内存泄露(Memory Leak)** 和 **内存溢出(Out Of Memory)** 两个概念经常会一起被提及，两者有相互关联的地方，但实质上还是有着很大的区别：

- 内存泄露。内存泄漏属于代码错误，这种错误会导致应用程序长久保留那些不再被需要的对象的引用，从而导致分配给该对象的内存无法被回收，造成程序的可用内存逐步降低，严重时甚至会导致 OOM。例如，当 Activity 的 onDestroy() 方法被调用后，正常来说 Activity 本身以及它涉及到的 View、Bitmap 等对象都应该被回收。但如果有一个后台线程持续持有对这个 Activity 的引用的话，那么 Activity 占据的内存就无法被回收，严重时将导致 OOM，最终 Crash
- 内存溢出。指一个应用在申请内存时，系统没有足够的内存空间可以供其使用

两者都会导致应用运行出现问题、性能下降或崩溃。不同点主要在于：

- 内存泄露是导致内存溢出的原因之一，内存泄露严重时将导致内存溢出
- 内存泄露是由于代码缺陷引起的，可以通过完善代码来避免；内存溢出可以通过调整配置来减少发生频率，但无法彻底避免

对于一个存在内存泄露的程序来说，即使每次仅会泄露少量内存，程序的可用内存也是会逐步降低，在长期运行过后，程序也是隐藏着崩溃的危险

# 二、内存管理

为了判断程序是否存在内存泄露的情况，我们首先必须先了解 Java 是如何管理内存的，Java 的内存管理就是对象的分配和释放过程

在 Java 中，我们都是通过关键字 new 来申请内存空间并创建对象的（基本类型除外），所有的对象都在堆 (Heap)中分配空间。总的来说，Java 的内存区域可以分为三类：

1. 静态存储区：在程序整个运行期间都存在，编译时就分配好空间，主要用于存放静态数据和常量
2. 栈区：当一个方法被执行时会在栈区内存中创建方法体内部的局部变量，方法结束后自动释放内存
3. 堆区：通常存放 new 出来的对象

对象的释放则由 GC 来完成。GC 负责监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等行为。当某个对象被 GC 判断为不再被引用了时，GC 就会回收并释放该对象对应的内存空间

一个对象的引用方式可以分为四种：

1. 强引用(StrongReference)：JVM 宁可抛出 OOM 也不会让 GC 回收具有强引用的对象
2. 软引用(SoftReference)：如果一个对象只具有软引用，那么在内存空间不足时就会回收该对象
3. 弱引用(WeakReference)：如果一个对象只具有弱引用，那么在 GC 时不管当前内存空间是否足够，都会回收该对象
4. 虚引用(PhantomReference)：任何时候都可以被 GC 回收，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否存在该对象的虚引用，来了解这个对象是否将要被回收

而一个对象不再被引用的标记就是其不再被强引用，JVM 会通过**引用计数法**或者是**可达性分析**等方法来判断一个对象是否还被强引用着

在 Java 中，内存泄露的就意味着发生了这么一种情况：一个对象是可达的，存在其它对象强引用着该对象，但该对象是无用的，程序以后不会再使用这些对象。满足这种情况的对象就意味着该对象已经泄露，该对象不会被 GC 所回收（因为该对象可达，还未达到 GC 的标准），然而却一直持续占用着内存。例如，由于非静态内部类会持有对外部类的隐式引用，所以当非静态内部类在被回收之前，外部类也无法被回收

# 三、常见的内存泄露

以下列举九种常见的内存泄露场景及相应的解决方案，内容来自于国外的一篇文章：9 ways to avoid memory leaks in Android

## 1、Broadcast Receivers

如果在 Activity 中注册了 BroadcastReceiver 而忘记了 **unregister** 的话，BroadcastReceiver 就将一直持有对 Activity 的引用，即使 Activity 已经执行了 `onDestroy`

```
public class BroadcastReceiverLeakActivity extends AppCompatActivity {



    private BroadcastReceiver broadcastReceiver;



    @Override

    protected void onCreate(@Nullable Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_first);

    }



    private void registerBroadCastReceiver() {

        broadcastReceiver = new BroadcastReceiver() {

            @Override

            public void onReceive(Context context, Intent intent) {

                //your receiver code goes here!

            }
```

```
        };

        registerReceiver(broadcastReceiver, new IntentFilter("SmsMessage.intent.MAIN
"));

    }




    @Override

    protected void onStart() {

        super.onStart();

        registerBroadCastReceiver();

    }




    @Override

    protected void onStop() {

        super.onStop();



        /*

         * Uncomment this line in order to avoid memory leak.

         * You need to unregister the broadcast receiver since the broadcast receiver
 keeps a reference of the activity.

         * Now when its time for your Activity to die, the Android framework will call
 onDestroy() on it

         * but the garbage collector will not be able to remove the instance from memo
ry because the broadcastReceiver
```

```
         * is still holding a strong reference to it.

         * */



        if(broadcastReceiver != null) {

            unregisterReceiver(broadcastReceiver);

        }

    }

}
```

复制代码

开发者必须谨记在 `Activity.onStop()` 的时候调用 `unregisterReceiver`。但需要注意的是，如果 BroadcastReceiver 是在 `onCreate()` 中进行注册的，那么当应用进入后台并再次切换回来时，BroadcastReceiver 将不会被再次注册。所以，最好在 Activity 的 `onStart()` 或者 `onResume()` 方法中进行注册，然后在 `onStop()` 时进行注销

## 2、Static Activity or View Reference

看下面的示例代码，将 TextView 声明为了静态变量（无论出于什么原因）。不管是直接还是间接通过静态变量引用了 Activity 或者 View，在 Activity 被销毁后都无法对其进行垃圾回收

```
public class StaticReferenceLeakActivity extends AppCompatActivity {



    /*

     * This is a bad idea!

     */

    private static TextView textView;
```

```
    private static Activity activity;



    @Override

    protected void onCreate(@Nullable Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_first);




        textView = findViewById(R.id.activity_text);

        textView.setText("Bad Idea!");



        activity = this;

    }

}
```

复制代码

永远不要通过静态变量来引用 Activity、View 和 Context

## 3、Singleton Class Reference

看下面的例子，定义了一个 Singleton 类，该类需要传递 Context 以便从本地存储中获取一些文件

```
public class SingletonLeakExampleActivity extends AppCompatActivity {



    private SingletonSampleClass singletonSampleClass;
```

```
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

    /*
     * Option 1: Do not pass activity context to the Singleton class. Instead pass ap
plication Context
     */
        singletonSampleClass = SingletonSampleClass.getInstance(this);
    }


   @Override
   protected void onDestroy() {
        super.onDestroy();

    /*
     * Option 2: Unregister the singleton class here i.e. if you pass activity contex
t to the Singleton class,
     * then ensure that when the activity is destroyed, the context in the singleton
class is set to null.
     */
     singletonSampleClass.onDestroy();
```

```
    }
}
复制代码public class SingletonSampleClass {

    private Context context;
    private static SingletonSampleClass instance;

    private SingletonSampleClass(Context context) {
        this.context = context;
    }

    public synchronized static SingletonSampleClass getInstance(Context context) {
        if (instance == null) instance = new SingletonSampleClass(context);
        return instance;
    }

    public void onDestroy() {
       if(context != null) {
          context = null;
       }
    }
}
```

复制代码

此时如果没有主动将 SingletonSampleClass 包含的 context 置空的话，就将导致内存泄露。那如何解决这个问题？

- 可以传递 ApplicationContext，而不是将 ActivityContext 传递给 singleton 类
- 如果真的必须使用 ActivityContext，那么当 Activity 被销毁的时候，需要确保传递将 singleton 类的 Context 设置为 null

## 4、Inner Class Reference

看下面的例子，定义了一个 LeakyClass 类，你需要传递 Activity 才能重定向到新的 Activity

```
public class InnerClassReferenceLeakActivity extends AppCompatActivity {



  /*

   * Mistake Number 1:

   * Never create a static variable of an inner class

   * Fix I:

   * private LeakyClass leakyClass;

   */

  private static LeakyClass leakyClass;



    @Override

    protected void onCreate(@Nullable Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_first);


        new LeakyClass(this).redirectToSecondScreen();


        /*
         * Inner class is defined here
         * */
         leakyClass = new LeakyClass(this);
         leakyClass.redirectToSecondScreen();
    }


/*
 * Mistake Number 2:
 * 1. Never create a inner variable of an inner class
 * 2. Never pass an instance of the activity to the inner class
 */
    private class LeakyClass {


        private Activity activity;
        public LeakyClass(Activity activity) {
            this.activity = activity;
        }
```

```
        public void redirectToSecondScreen() {
            this.activity.startActivity(new Intent(activity, SecondActivity.class));
        }
    }
}
```

复制代码

如何解决这个问题？

- 就如之前所述，不要创建内部类的静态变量
- LeakyClass 设置为静态类，静态内部类不会持有对其外部类的隐式引用
- 对任何 View/Activity 都使用 weakReference。如果只有弱引用指向某个对象，那么垃圾回收器就可以回收该对象

```
public class InnerClassReferenceLeakActivity extends AppCompatActivity {


  /*
   * Mistake Number 1:
   * Never create a static variable of an inner class
   * Fix I:
   */
  private LeakyClass leakyClass;


    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_first);



        new LeakyClass(this).redirectToSecondScreen();



        /*
         * Inner class is defined here
         * */
         leakyClass = new LeakyClass(this);
         leakyClass.redirectToSecondScreen();
    }




    /*
     * How to fix the above class:
     * Fix memory leaks:
     * Option 1: The class should be set to static
     * Explanation: Instances of anonymous classes do not hold an implicit reference to their outer class
     * when they are "static".
     *
     * Option 2: Use a weakReference of the textview or any view/activity for that matter
     * Explanation: Weak References: Garbage collector can collect an object if only weak references
```

```
     * are pointing towards it.
     * */
    private static class LeakyClass {


        private final WeakReference<Activity> messageViewReference;
        public LeakyClass(Activity activity) {
            this.activity = new WeakReference<>(activity);
        }


        public void redirectToSecondScreen() {
            Activity activity = messageViewReference.get();
            if(activity != null) {
               activity.startActivity(new Intent(activity, SecondActivity.class));
            }
        }
    }
}
复制代码
```

## 5、Anonymous Class Reference

匿名内存类带来的内存泄漏问题和上一节内容相同，解决办法如下所示：

```
public class AnonymousClassReferenceLeakActivity extends AppCompatActivity {
```

```
    private TextView textView;



    @Override

    protected void onCreate(@Nullable Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_first);




        textView = findViewById(R.id.activity_text);

        textView.setText(getString(R.string.text_inner_class_1));

        findViewById(R.id.activity_dialog_btn).setVisibility(View.INVISIBLE);



        /*

         * Runnable class is defined here

         * */

         new Thread(new LeakyRunnable(textView)).start();

    }







    private static class LeakyRunnable implements Runnable {



        private final WeakReference<TextView> messageViewReference;
```

```
        private LeakyRunnable(TextView textView) {
            this.messageViewReference = new WeakReference<>(textView);
        }

        @Override
        public void run() {
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            TextView textView = messageViewReference.get();
            if(textView != null) {
                textView.setText("Runnable class has completed its work");
            }
        }
    }
}
```

复制代码

## 6、AsyncTask Reference

看下面的示例，通过 AsyncTask 来获取一个字符串值，该值用于在 `onPostExecute()` 方法中更新 textView

```
public class AsyncTaskReferenceLeakActivity extends AppCompatActivity {



    private TextView textView;

    private BackgroundTask backgroundTask;



    @Override

    protected void onCreate(@Nullable Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_first);



        /*

         * Executing AsyncTask here!

         * */

        backgroundTask = new BackgroundTask(textView);

        backgroundTask.execute();

    }



    /*

     * Couple of things we should NEVER do here:

     * Mistake number 1. NEVER reference a class inside the activity. If we definitel
y need to, we should set the class as static as static inner classes don't hold

     *    any implicit reference to its parent activity class

     * Mistake number 2. We should always cancel the asyncTask when activity is destr
oyed. This is because the asyncTask will still be executing even if the activity
```

```
     *    is destroyed.
     * Mistake number 3. Never use a direct reference of a view from acitivty inside an asynctask.
     * */
 private class BackgroundTask extends AsyncTask<Void, Void, String> {
        @Override
        protected String doInBackground(Void... voids) {

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "The task is completed!";
        }

        @Override
        protected void onPostExecute(String s) {
            super.onPostExecute(s);
            textView.setText(s);
        }
    }
}
```

复制代码

如何解决这个问题？

- 当 Activity 被销毁时，我们应该取消异步任务，这是因为即使 Activity 已经走向 Destoryed，未结束的 AsyncTask 仍将继续执行
- 永远不要在 Activity 中引用内部类。如果确实需要，我们应该将其设置为静态类，因为静态内部类不会包含对其外部类的任何隐式引用
- 通过 weakReference 来引用 textview

```
public class AsyncTaskReferenceLeakActivity extends AppCompatActivity {



    private TextView textView;

    private BackgroundTask backgroundTask;



    @Override

    protected void onCreate(@Nullable Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_first);



        /*

         * Executing AsyncTask here!

         * */

        backgroundTask = new BackgroundTask(textView);

        backgroundTask.execute();

    }
```

```java
/*
 * Fix number 1
 * */
private static class BackgroundTask extends AsyncTask<Void, Void, String> {

    private final WeakReference<TextView> messageViewReference;
    private BackgroundTask(TextView textView) {
        this.messageViewReference = new WeakReference<>(textView);
    }


    @Override
    protected String doInBackground(Void... voids) {

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "The task is completed!";
    }
```

```
    @Override
    protected void onPostExecute(String s) {
        super.onPostExecute(s);
      /*
       * Fix number 3
       * */
        TextView textView = messageViewReference.get();
        if(textView != null) {
            textView.setText(s);
        }
    }
}


@Override
protected void onDestroy() {
    super.onDestroy();


    /*
     * Fix number 2
     * */
    if(backgroundTask != null) {
        backgroundTask.cancel(true);
```

```
        }

    }

}
```

复制代码

## 7、Handler Reference

看下面的例子，通过 Handler 在五秒后更新 UI

```
public class HandlersReferenceLeakActivity extends AppCompatActivity {



    private TextView textView;



    /*

     * Mistake Number 1

     * */

     private Handler leakyHandler = new Handler();




    @Override

    protected void onCreate(@Nullable Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_first);



        /*
```

```
        * Mistake Number 2
        * */
        leakyHandler.postDelayed(new Runnable() {
            @Override
            public void run() {
                textView.setText(getString(R.string.text_handler_1));
            }
        }, 5000);
    }
```

复制代码

如何解决这个问题？

- 永远不要在 Activity 中引用内部类。如果确实需要，我们应该将其设置为静态类。这是因为当 Handler 在主线程上实例化后，它将与 Looper 的 MessageQueue 相关联，发送到 MessageQueue 的 Message 将持有对 Handler 的引用，以便当 Looper 最终处理消息时，framework 可以调用 Handler#handleMessage(message) 方法
- 通过 weakReference 来引用 Activity

```
public class HandlersReferenceLeakActivity extends AppCompatActivity {


    private TextView textView;


    /*
     * Fix number I
```

```
 * */
private final LeakyHandler leakyHandler = new LeakyHandler(this);


@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_first);


    leakyHandler.postDelayed(leakyRunnable, 5000);
}


/*
 * Fix number II - define as static
 * */
private static class LeakyHandler extends Handler {


/*
 * Fix number III - Use WeakReferences
 * */
    private WeakReference<HandlersReferenceLeakActivity> weakReference;
    public LeakyHandler(HandlersReferenceLeakActivity activity) {
        weakReference = new WeakReference<>(activity);
    }
```

```
        @Override
        public void handleMessage(Message msg) {
            HandlersReferenceLeakActivity activity = weakReference.get();
            if (activity != null) {
                activity.textView.setText(activity.getString(R.string.text_handler_
2));
            }
        }
    }

    private static final Runnable leakyRunnable = new Runnable() {
        @Override
        public void run() { /* ... */ }
    }
```

复制代码

## 8、Threads Reference

Thread 和 TimerTask 也可能会导致内存泄露问题

```
public class ThreadReferenceLeakActivity extends AppCompatActivity {

    /*
     * Mistake Number 1: Do not use static variables
```

```
 * */
private static LeakyThread thread;


@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_first);


    createThread();
    redirectToNewScreen();
}



private void createThread() {
    thread = new LeakyThread();
    thread.start();
}


private void redirectToNewScreen() {
    startActivity(new Intent(this, SecondActivity.class));
}
```

```
    /*
     * Mistake Number 2: Non-static anonymous classes hold an
     * implicit reference to their enclosing class.
     * */
    private class LeakyThread extends Thread {
        @Override
        public void run() {
            while (true) {
            }
        }
    }
```

复制代码

如何解决这个问题？

- 非静态匿名类会包含对其外部类的隐式引用，将 LeakyThread 改为静态内部类
- 在 Activity 的 onDestroy() 方法中停止线程，以避免线程泄漏

```
public class ThreadReferenceLeakActivity extends AppCompatActivity {



    /*
     * FIX I: make variable non static
     * */
    private LeakyThread leakyThread = new LeakyThread();
```

```
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_first);

    createThread();
    redirectToNewScreen();
}

private void createThread() {
    leakyThread.start();
}

private void redirectToNewScreen() {
    startActivity(new Intent(this, SecondActivity.class));
}

@Override
protected void onDestroy() {
    super.onDestroy();
    // FIX II: kill the thread
```

```
        leakyThread.interrupt();
    }



    /*
     * Fix III: Make thread static
     * */
    private static class LeakyThread extends Thread {
        @Override
        public void run() {
            while (!isInterrupted()) {
            }
        }
    }
}
```

复制代码

## 9、TimerTask Reference

对于 TimerTask 也可以遵循相同的原则，修复内存泄漏的示例如下所示：

```
public class TimerTaskReferenceLeakActivity extends Activity {



    private CountDownTimer countDownTimer;
```

```
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_first);


    startTimer();
}


/*
 * Mistake 1: Cancel Timer is never called
 * even though activity might be completed
 * */
public void cancelTimer() {
    if(countDownTimer != null) countDownTimer.cancel();
}




private void startTimer() {
    countDownTimer = new CountDownTimer(1000, 1000) {
        @Override
        public void onTick(long millisUntilFinished) {
            final int secondsRemaining = (int) (millisUntilFinished / 1000);
            //update UI
```

```
            }


            @Override
            public void onFinish() {
                //handle onFinish
            }
        };
        countDownTimer.start();
    }
}
```

复制代码

如何解决这个问题？

- 在 Activity 的 onDestroy() 方法中停止计时器，以避免内存泄漏

```
public class TimerTaskReferenceLeakActivity extends Activity {


    private CountDownTimer countDownTimer;


    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_first);
```

```
    startTimer();
}



public void cancelTimer() {
    if(countDownTimer != null) countDownTimer.cancel();
}



private void startTimer() {
    countDownTimer = new CountDownTimer(1000, 1000) {
        @Override
        public void onTick(long millisUntilFinished) {
            final int secondsRemaining = (int) (millisUntilFinished / 1000);
            //update UI
        }


        @Override
        public void onFinish() {
            //handle onFinish
        }
    };
    countDownTimer.start();
```

```
    }




    /*
     * Fix 1: Cancel Timer when
     * activity might be completed
     * */
   @Override
    protected void onDestroy() {
        super.onDestroy();
        cancelTimer();
    }
}
```

复制代码

## 10、总结

最后再来简单总结一下：

1. 尽可能使用 ApplicationContext 而不是 ActivityContext。如果真的必须使用 ActivityContext，那么当 Activity 被销毁时，请确保将传递的 Context 置为 null
2. 不要通过静态变量来引用 View 和 Activity
3. 不要在 Activity 中引用内部类，如果确实需要，那么应该将它声明为静态的，不管它是 Thread、Handler、Timer 还是 AsyncTask

4. 务必记住注销 Activity 中的 BroadcastReceiver 和 Timer，在 onDestroy() 方法中取消任何异步任务和线程
5. 通过 weakReference 来持有对 Activity 和 View 的引用

# 三方库源码笔记（7）-超详细的 Retrofit 源码解析

## 一、前言

Retrofit 也是现在 Android 应用开发中的标配之一了吧？笔者使用 Retrofit 蛮久的了，一直以来用着也挺舒心的，没遇到啥大的坑。总这样用着不来了解下其实现原理也好像不太好，趁着动笔写 三方库源码笔记 系列文章就来对 Retrofit 进行一次（自我感觉的）全面的源码解析吧

Retrofit 的源码并不算太复杂，但由于应用了很多种设计模式，所以在流程上会比较绕。笔者从 **2020/10/10** 开始看源码，陆陆续续看了几天源码后就开始动笔，但总感觉没法阐述得特别清晰，写着写着就成了目前的样子。读者如果觉得我有哪里写得不太好的地方也欢迎给下建议

Retrofit 是这么自我介绍的：**A type-safe HTTP client for Android and Java.**

说明 Retrofit 的内部实现并不需要依赖于 Android 平台，而是可以用于任意的 Java 客户端，Retrofit 只是对 Android 平台进行了特殊实现而已（例如，在 main 线程进行回调）。但现在 Android 平台的主流开发语言早已是 Kotlin 了，所以本篇文章所写的例子就都还是用 Kotlin 语言来实现吧~

对 Kotlin 语言不熟悉的同学可以看我的这篇文章来入门：两万六千字带你 Kotlin 入门

## 二、小例子

先来看几个简单的小例子，后续的讲解都会围绕这几个例子来展开

先引入当前 Retrofit 的最新版本：

```
dependencies {

    implementation 'com.squareup.retrofit2:retrofit:2.9.0'
```

```
}
```

复制代码

在除了 Retrofit 外不引入其它任何依赖库的情况下，我们发起一个网络请求的流程都大致如下所示：

```
/**
 * 作者：leavesC
 * 时间：2020/10/13 0:05
 * 描述：
 * GitHub：https://github.com/leavesC
 */interface ApiService {

    @GET("getUserData")
    fun getUserData(): Call<ResponseBody>

}
fun main() {
    val retrofit = Retrofit.Builder()
        .baseUrl("https://mockapi.eolinker.com/9IiwI82f58c23411240ed608ceca204b2f185014507cbe3/")
        .build()
    val service = retrofit.create(ApiService::class.java)
    val call: Call<ResponseBody> = service.getUserData()
    call.enqueue(object : Callback<ResponseBody> {
```

```
        override fun onResponse(call: Call<ResponseBody>, response: Response<Response
Body>) {

            val userBean = response.body()?.string()

            println("userBean: $userBean")

        }



        override fun onFailure(call: Call<ResponseBody>, t: Throwable) {

            println("onFailure: $t")

        }

    })

}
```

复制代码

输出结果：

```
userBean: {"userName":"JBl","userAge":7816977017260632}
```

复制代码

Retrofit 是建立在 OkHttp 之上的一个网络请求封装库，内部依靠 OkHttp 来完成实际的网络请求。Retrofit 在使用上很简洁，API 是通过 interface 来声明的。不知道读者第一次使用 Retrofit 的时候是什么感受，我第一次使用的时候就觉得 Retrofit 好神奇，我只需要通过 interface 来声明 **API 路径、请求方式、请求参数、返回值类型**等各个配置项，然后调用方法就可以发起网络请求了，相比 OkHttp 和 Volley 这些网络请求库真的是简洁到没朋友

从上述例子可以看到，`getUserData()`方法所代表的 API 的请求结果是一个 Json 格式的字符串，其返回值类型被定义为 `Call<ResponseBody>`，此处的 ResponseBody 即 `okhttp3.ResponseBody`，是 OkHttp 提供的对网络请求结果的包装类，Call 即 `retrofit2.Call`，是 Retrofit 对 `okhttp3.Call` 做的一层包装，OkHttp 在实际发起请求的时候使用的回调是 `okhttp3.Call`，回调内部就会来中转调用 `retrofit2.Call`，以便将请求结果转交给外部

## 1、converter-gson

上述例子虽然简单，但还不够方便，因为既然 API 的返回值我们已知就是 Json 格式的了，那么我们自然就希望 `getUserData()` 方法的返回值直接就是一个 Bean 对象，而不是拿到一个 String 对象后还需要自己再去进行反序列化，这可以通过 `converter-gson` 这个库来达到这个效果

```
dependencies {

    implementation 'com.squareup.retrofit2:retrofit:2.9.0'

    implementation 'com.squareup.retrofit2:converter-gson:2.5.0'

}
```

复制代码

代码再做点小改动，之后就可以直接在 Callback 中拿到 UserBean 对象了

```
/**

 * 作者：LeavesC

 * 时间：2020/10/13 0:05

 * 描述：

 * GitHub：https://github.com/LeavesC

 */interface ApiService {



    @GET("getUserData")

    fun getUserData(): Call<UserBean>



}

data class UserBean(val userName: String, val userAge: Long)
```

```
fun main() {

    val retrofit = Retrofit.Builder()

        .baseUrl("https://mockapi.eolinker.com/9IiwI82f58c23411240ed608ceca204b2f185
014507cbe3/")

        .addConverterFactory(GsonConverterFactory.create())

        .build()

    val service = retrofit.create(ApiService::class.java)

    val call: Call<UserBean> = service.getUserData()

    call.enqueue(object : Callback<UserBean> {

        override fun onResponse(call: Call<UserBean>, response: Response<UserBean>) {

            val userBean = response.body()

            println("userBean: $userBean")

        }



        override fun onFailure(call: Call<UserBean>, t: Throwable) {

            println("onFailure: $t")

        }

    })

}
```

复制代码

## 2、adapter-rxjava2

再然后，如果我们也看 `Call<UserBean>`不爽，想要通过 RxJava 的方式来进行网络请求可不可以？也行，此时就需要再使用到 `adapter-rxjava2` 这个库了

```
dependencies {

    implementation 'com.squareup.retrofit2:retrofit:2.9.0'

    implementation 'com.squareup.retrofit2:converter-gson:2.5.0'

    implementation 'com.squareup.retrofit2:adapter-rxjava2:2.9.0'

}
```

复制代码

代码再来做点小改动，此时就完全不用使用到 `Call.enqueue` 来显式发起网络请求了，当进行 subscribe 的时候就会触发网络请求

```
/**

 * 作者：LeavesC

 * 时间：2020/10/13 0:05

 * 描述：

 * GitHub：https://github.com/LeavesC

 */interface ApiService {



    @GET("getUserData")

    fun getUserData(): Observable<UserBean>



}

data class UserBean(val userName: String, val userAge: Long)

fun main() {

    val retrofit = Retrofit.Builder()
```

```
        .baseUrl("https://mockapi.eolinker.com/9IiwI82f58c23411240ed608ceca204b2f185
014507cbe3/")

        .addConverterFactory(GsonConverterFactory.create())

        .addCallAdapterFactory(RxJava2CallAdapterFactory.create())

        .build()

    val service = retrofit.create(ApiService::class.java)

    val call: Observable<UserBean> = service.getUserData()

    call.subscribe(object : Consumer<UserBean> {

        override fun accept(userBean: UserBean?) {

            println("userBean: $userBean")

        }



    }, object : Consumer<Throwable> {

        override fun accept(t: Throwable?) {

            println("onFailure: $t")

        }

    })

}
```

复制代码

## 3、小疑问

可以看到，Retrofit 在抽象程度上是很高的。不管是需要 Call 类还是 Observable 类型的包装类，只需要添加不同的 `CallAdapterFactory` 即可，就算想返回 LiveData 类型都是可以实现的。也不管是需要 ResponseBody 类型还是

具体的 Bean 对象类型，也只需要添加不同的 `ConverterFactory` 即可，就算 API 的返回值是 XML 格式也可以进行映射解析

那么，我们就需要带着几个问题来逐步看 Retrofit 的源码：

1. Retrofit 是如何将 interface 内部的方法转化为一个个实际的 GET、POST、DELETE 等各式各样的网络请求的呢？例如，Retrofit 是如何将 getUserData() 转换为一个 OkHttp 的 GET 请求的呢？
2. Retrofit 是如何将 API 的返回值映射为具体的 Bean 对象的呢？例如，ResponseBody 是如何映射为 UserBean 的呢？
3. Retrofit 是如何抽象不同的 API 返回值包装类的呢？例如，Call 是如何替换为 Observable 的呢？

## 三、Retrofit.create()

先来看下 `retrofit.create` 方法做了什么

```
public <T> T create(final Class<T> service) {

    validateServiceInterface(service);

    return (T)

        Proxy.newProxyInstance(

            service.getClassLoader(),

            new Class<?>[] {service},

            new InvocationHandler() {

              private final Platform platform = Platform.get();

              private final Object[] emptyArgs = new Object[0];



              @Override
```

```
            public @Nullable Object invoke(Object proxy, Method method, @Nullable O
bject[] args)

                throws Throwable {

              // If the method is a method from Object then defer to normal invocati
on.

              if (method.getDeclaringClass() == Object.class) {

              //如果外部调用的是 Object 中声明的方法的话则直接调用

              //例如 toString()、hashCode() 等方法

                return method.invoke(this, args);

              }

              args = args != null ? args : emptyArgs;

              //根据 method 是否默认方法来决定如何调用

              return platform.isDefaultMethod(method)

                  ? platform.invokeDefaultMethod(method, service, proxy, args)

                  : loadServiceMethod(method).invoke(args);

            }

          });

  }
```

复制代码

这里的重点就是 `Proxy.newProxyInstance` 所实现的**动态代理模式**了。通过动态代理，Retrofit 会将我们对 ApiService 的调用操作转发给 **InvocationHandler** 来完成。Retrofit 在后续会通过反射拿到我们在声明 `getUserData()` 时标注的各个配置项，例如 **API 路径、请求方式、请求参数、返回值类型**等各个信息，然后将这些配置项拼接为 OkHttp 的一个原始网络请求。当我们调用了 `call.enqueue` 方法时，这个操作就会触发 **InvocationHandler** 去发起 OkHttp 网络请求了

Retrofit 会根据 method 是否是**默认方法**来决定如何调用，这里主要看 `loadServiceMethod(method)`方法，该方法的主要逻辑是：

1. 将每个代表接口方法的 method 对象转换为 ServiceMethod 对象，该对象中就包含了 API 的具体信息
2. 因为 API 可能先后会被调用多次，所以将构造出来的 ServiceMethod 对象缓存到 serviceMethodCache 中以实现复用

```
  private final Map<Method, ServiceMethod<?>> serviceMethodCache = new ConcurrentHas
hMap<>();



  ServiceMethod<?> loadServiceMethod(Method method) {

    ServiceMethod<?> result = serviceMethodCache.get(method);

    if (result != null) return result;



    synchronized (serviceMethodCache) {

      result = serviceMethodCache.get(method);

      if (result == null) {

        //重点

        result = ServiceMethod.parseAnnotations(this, method);

        serviceMethodCache.put(method, result);

      }

    }

    return result;

  }
```

复制代码

# 四、ServiceMethod

从上面可知，`loadServiceMethod(method)`方法返回的是一个 ServiceMethod 对象，从名字上来看也可以猜出，每个 ServiceMethod 对象就对应一个 API 接口方法，其内部就包含了对 API 的解析结果。`loadServiceMethod(method).invoke(args)` 这个操作就对应**调用 API 方法并传递参数**这个过程，即对应 `val call: Call<ResponseBody> = service.getUserData()` 这个过程

ServiceMethod 是一个抽象类，仅包含一个抽象的 `invoke(Object[] args)`方法。ServiceMethod 使用到了工厂模式，由于 API 的最终请求方式可能是多样化的，既可能是通过线程池来执行，也可能是通过 Kotlin 协程来执行，使用工厂模式的意义就在于可以将这种差异都隐藏在不同的 ServiceMethod 实现类中，而外部统一都是通过 `parseAnnotations` 方法来获取 ServiceMethod 的实现类

`parseAnnotations` 方法返回的 ServiceMethod 实际上是 **HttpServiceMethod**，所以重点就要来看 `HttpServiceMethod.parseAnnotations` 方法返回的 HttpServiceMethod 具体是如何实现的，并是如何拼接出一个完整的 OkHttp 请求调用链

```
abstract class ServiceMethod<T> {

  static <T> ServiceMethod<T> parseAnnotations(Retrofit retrofit, Method method) {

    //requestFactory 包含了对 API 的注解信息进行解析后的结果

    RequestFactory requestFactory = RequestFactory.parseAnnotations(retrofit, metho
d);



    Type returnType = method.getGenericReturnType();

    //如果返回值包含未确定的泛型类型或者是包含通配符的话，那么就抛出异常

    //因为 Retrofit 无法构造出一个不具有确定类型的对象作为返回值

    if (Utils.hasUnresolvableType(returnType)) {

      throw methodError(

          method,

          "Method return type must not include a type variable or wildcard: %s",
```

```
          returnType);

    }

    //返回值类型不能是 void

    if (returnType == void.class) {

      throw methodError(method, "Service methods cannot return void.");

    }



    //重点

    return HttpServiceMethod.parseAnnotations(retrofit, method, requestFactory);

  }



  abstract @Nullable T invoke(Object[] args);

}
```

复制代码

## 五、HttpServiceMethod

通过查找引用，可以知道 ServiceMethod 这个抽象类的直接子类只有一个，即 **HttpServiceMethod**。HttpServiceMethod 也是一个抽象类，其包含两个泛型声明，`ResponseT` 表示的是 **API 方法返回值的外层包装类型**，`ReturnT` 表示的是我们**实际需要的数据类型**。例如，对于 `fun getUserData(): Call<UserBean>` 方法，`ResponseT` 对应的是 `Call`，`ReturnT` 对应的是 `UserBean`。此外，HttpServiceMethod 也实现了父类的 `invoke` 方法，并将操作转交给了另一个抽象方法 `adapt` 来完成，所以说，API 方法对应的网络请求具体的发起操作主要看 `adapt` 方法即可

```
abstract class HttpServiceMethod<ResponseT, ReturnT> extends ServiceMethod<ReturnT>
{
```

```
  @Override

  final @Nullable ReturnT invoke(Object[] args) {

    Call<ResponseT> call = new OkHttpCall<>(requestFactory, args, callFactory, respo
nseConverter);

    return adapt(call, args);

  }



  protected abstract @Nullable ReturnT adapt(Call<ResponseT> call, Object[] args);



  ...



}
```

复制代码

再来看 `HttpServiceMethod.parseAnnotations(retrofit, method, requestFactory)`方法是如何构建出一个 HttpServiceMethod 对象的，并且该对象的 `adapt` 方法是如何实现的

```
abstract class HttpServiceMethod<ResponseT, ReturnT> extends ServiceMethod<ReturnT>
{

  /**

   * Inspects the annotations on an interface method to construct a reusable service
method that

   * speaks HTTP. This requires potentially-expensive reflection so it is best to bui
ld each service

   * method only once and reuse it.

   */
```

```
  static <ResponseT, ReturnT> HttpServiceMethod<ResponseT, ReturnT> parseAnnotations
(

      Retrofit retrofit, Method method, RequestFactory requestFactory) {

    //是否是 Suspend 函数，即是否以 Kotlin 协程的方式来进行请求

    boolean isKotlinSuspendFunction = requestFactory.isKotlinSuspendFunction;

    boolean continuationWantsResponse = false;

    boolean continuationBodyNullable = false;



    Annotation[] annotations = method.getAnnotations();

    Type adapterType;

    if (isKotlinSuspendFunction) {

        //省略 Kotlin 协程的一些处理逻辑

    } else {

      adapterType = method.getGenericReturnType();

    }



    //重点1

    CallAdapter<ResponseT, ReturnT> callAdapter =

        createCallAdapter(retrofit, method, adapterType, annotations);



    //拿到包装类内部的具体类型，例如，Observable<UserBean> 内部的 UserBean

    //responseType 不能是 okhttp3.Response 或者是不包含具体泛型类型的 Response

    Type responseType = callAdapter.responseType();
```

```
    if (responseType == okhttp3.Response.class) {
      throw methodError(
          method,
          "'"
              + getRawType(responseType).getName()
              + "' is not a valid response body type. Did you mean ResponseBody?");
    }
    if (responseType == Response.class) {
      throw methodError(method, "Response must include generic type (e.g., Response<String>)");
    }
    // TODO support Unit for Kotlin?
    if (requestFactory.httpMethod.equals("HEAD") && !Void.class.equals(responseType)) {
      throw methodError(method, "HEAD method must use Void as response type.");
    }


    //重点2
    Converter<ResponseBody, ResponseT> responseConverter =
        createResponseConverter(retrofit, method, responseType);


    okhttp3.Call.Factory callFactory = retrofit.callFactory;
    if (!isKotlinSuspendFunction) {
      //重点3
```

```
      return new CallAdapted<>(requestFactory, callFactory, responseConverter, callA
dapter);

    }



    //省略 Kotlin 协程的一些处理逻辑

    ...


  }


  ...



}
```

复制代码

Retrofit 目前已经支持以 Kotlin 协程的方式来进行调用了，但本例子和协程无关，所以此处先忽略协程相关的处理逻辑，后面会再讲解，`parseAnnotations` 方法的主要逻辑是：

1. 先通过 `createCallAdapter(retrofit, method, adapterType, annotations)` 方法拿到 CallAdapter 对象，CallAdapter 就用于实现 API 方法的不同返回值**包装类**处理逻辑。例如，`getUserData()`方法的返回值**包装类**类型如果是 `Call` ，那么返回的 CallAdapter 对象就对应 `DefaultCallAdapterFactory` 包含的 Adapter；如果是 `Observable`，那么返回的就是 `RxJava2CallAdapterFactory` 包含的 Adapter
2. 再通过 `createResponseConverter(retrofit, method, responseType)` 方法拿到 Converter 对象，Converter 就用于实现 API 方法的不同返回值处理逻辑。例如，`getUserData()` 方法的目标返回值类型如果是 `ResponseBody` ，那么 Converter 对象就对应 `BuiltInConverters`；如果是 `UserBean`，那么就对应 `GsonConverterFactory`

3. 根据前两个步骤拿到的值，构造出一个 CallAdapted 对象并返回

CallAdapted 正是 HttpServiceMethod 的子类，在 **InvocationHandler** 中通过 `loadServiceMethod(method).invoke(args)`发起的调用链，会先创建出一个 OkHttpCall 对象，并最后调用到 `callAdapter.adapt(call)`方法

```
abstract class HttpServiceMethod<ResponseT, ReturnT> extends ServiceMethod<ReturnT>
{



    @Override

    final @Nullable ReturnT invoke(Object[] args) {

        Call<ResponseT> call = new OkHttpCall<>(requestFactory, args, callFactory, re
sponseConverter);

        return adapt(call, args);

    }



}



static final class CallAdapted<ResponseT, ReturnT> extends HttpServiceMethod<Respons
eT, ReturnT> {

    private final CallAdapter<ResponseT, ReturnT> callAdapter;



    CallAdapted(

        RequestFactory requestFactory,

        okhttp3.Call.Factory callFactory,

        Converter<ResponseBody, ResponseT> responseConverter,

        CallAdapter<ResponseT, ReturnT> callAdapter) {
```

```
      super(requestFactory, callFactory, responseConverter);
      this.callAdapter = callAdapter;
    }

    @Override
    protected ReturnT adapt(Call<ResponseT> call, Object[] args) {
      return callAdapter.adapt(call);
    }
  }
```

复制代码

## 六、OkHttpCall

OkHttpCall 是实际发起 OkHttp 请求的地方。当我们调用 `fun getUserData(): Call<ResponseBody>` 方法时，返回的 Call 对象实际上是 OkHttpCall 类型，而当我们调用 `call.enqueue(Callback)`方法时，`enqueue` 方法中会发起一个 OkHttp 请求，传入的 Callback 对象就会由 `okhttp3.Callback` 本身收到回调时再进行中转调用

```
final class OkHttpCall<T> implements Call<T> {

  private final RequestFactory requestFactory;

  private final Object[] args;

  private final okhttp3.Call.Factory callFactory;

  private final Converter<ResponseBody, T> responseConverter;



  private volatile boolean canceled;
```

```
@GuardedBy("this")
private @Nullable okhttp3.Call rawCall;

@GuardedBy("this") // Either a RuntimeException, non-fatal Error, or IOException.
private @Nullable Throwable creationFailure;

@GuardedBy("this")
private boolean executed;

@Override
public void enqueue(final Callback<T> callback) {
  ...
  okhttp3.Call call;
  ...
  call.enqueue( new okhttp3.Callback() {
        @Override
        public void onResponse(okhttp3.Call call, okhttp3.Response rawResponse) {
          Response<T> response;
          try {
            response = parseResponse(rawResponse);
          } catch (Throwable e) {
            throwIfFatal(e);
            callFailure(e);
```

```
      return;
    }

    try {
      callback.onResponse(OkHttpCall.this, response);
    } catch (Throwable t) {
      throwIfFatal(t);
      t.printStackTrace(); // TODO this is not great
    }
  }

  @Override
  public void onFailure(okhttp3.Call call, IOException e) {
    callFailure(e);
  }

  private void callFailure(Throwable e) {
    try {
      callback.onFailure(OkHttpCall.this, e);
    } catch (Throwable t) {
      throwIfFatal(t);
      t.printStackTrace(); // TODO this is not great
    }
```

```
        }
      });
  }


   ...


}
```

复制代码

## 七、小总结

以上几个小节的内容讲了在发起如下请求的过程中涉及到的所有流程，但单纯这样看的话其实有点难把握各个小点（**我自己看着都有点绕**），所以这里就再来回顾下以上内容，把所有知识点给串联起来

```
/**
 * 作者：leavesC
 * 时间：2020/10/13 0:05
 * 描述：
 * GitHub：https://github.com/leavesC
 */interface ApiService {


    @GET("getUserData")
    fun getUserData(): Call<ResponseBody>
```

```
}

fun main() {

    val retrofit = Retrofit.Builder()

        .baseUrl("https://mockapi.eolinker.com/9IiwI82f58c23411240ed608ceca204b2f185
014507cbe3/")

        .build()

    val service = retrofit.create(ApiService::class.java)

    val call: Call<ResponseBody> = service.getUserData()

    call.enqueue(object : Callback<ResponseBody> {

        override fun onResponse(call: Call<ResponseBody>, response: Response<Response
Body>) {

            val userBean = response.body()?.string()

            println("userBean: $userBean")

        }



        override fun onFailure(call: Call<ResponseBody>, t: Throwable) {

            println("onFailure: $t")

        }

    })

}
```

复制代码

首先，我们通过 `retrofit.create(ApiService::class.java)` 得到一个 ApiService 的**动态实现类**，这是通过 Java 原生提供的 `Proxy.newProxyInstance` 代表的动态代理功能来实现的。在拿到 ApiService 的实现类后，我们就可以直接调用 ApiService 中声明的所有方法了

而当我们调用了`service.getUserData()`方法时，Retrofit 会将每一个 API 方法都抽象封装为一个 ServiceMethod 对象并缓存起来，我们的操作会转交给 ServiceMethod 来完成，由 ServiceMethod 来负责返回我们的目标类型，对应的是`ServiceMethod.invoke(Object[] args)`方法，args 代表的是我们调用 API 方法时需要传递的参数，对应本例子就是一个空数组

```
abstract class ServiceMethod<T> {

  static <T> ServiceMethod<T> parseAnnotations(Retrofit retrofit, Method method) {

    //requestFactory 包含了对 API 的注解信息进行解析后的结果

    RequestFactory requestFactory = RequestFactory.parseAnnotations(retrofit, metho
d);



    Type returnType = method.getGenericReturnType();

    //如果返回值包含未确定的泛型类型或者是包含通配符的话，那么就抛出异常

    //因为 Retrofit 无法构造出一个不具有确定类型的对象作为返回值

    if (Utils.hasUnresolvableType(returnType)) {

      throw methodError(

          method,

          "Method return type must not include a type variable or wildcard: %s",

          returnType);

    }

    //返回值类型不能是 void

    if (returnType == void.class) {

      throw methodError(method, "Service methods cannot return void.");

    }
```

```
    //重点
    return HttpServiceMethod.parseAnnotations(retrofit, method, requestFactory);
  }

  abstract @Nullable T invoke(Object[] args);
}
```

复制代码

而实际上，ServiceMethod 只具有一个唯一的直接子类，即 HttpServiceMethod，而 HttpServiceMethod 会在 `invoke` 方法中构建出一个 OkHttpCall 对象，然后调用其抽象方法 `adapt`

此外，对于不同的请求方式，`ServiceMethod.parseAnnotations` 方法最终会返回不同的 HttpServiceMethod 子类。对应本例子，最终返回的会是 CallAdapted 对象

```
abstract class HttpServiceMethod<ResponseT, ReturnT> extends ServiceMethod<ReturnT>
{

  @Override
  final @Nullable ReturnT invoke(Object[] args) {
    Call<ResponseT> call = new OkHttpCall<>(requestFactory, args, callFactory, respo
nseConverter);
    return adapt(call, args);
  }

  protected abstract @Nullable ReturnT adapt(Call<ResponseT> call, Object[] args);
```

```
  ...



}



static final class CallAdapted<ResponseT, ReturnT> extends HttpServiceMethod<Respons
eT, ReturnT> {

    private final CallAdapter<ResponseT, ReturnT> callAdapter;



    CallAdapted(

        RequestFactory requestFactory,

        okhttp3.Call.Factory callFactory,

        Converter<ResponseBody, ResponseT> responseConverter,

        CallAdapter<ResponseT, ReturnT> callAdapter) {

      super(requestFactory, callFactory, responseConverter);

      this.callAdapter = callAdapter;

    }



    @Override

    protected ReturnT adapt(Call<ResponseT> call, Object[] args) {

      return callAdapter.adapt(call);

    }

  }
```

复制代码

所以，当我们调用 `val call: Call<ResponseBody> = service.getUserData()`时，返回的 `Call<ResponseBody>`实际上是 `OkHttpCall<ResponseBody>`

而当我们调用 `call.enqueue`方法时，OkHttpCall 的 `enqueue` 方法内部就会发起一个 OkHttp 请求，并为这个请求设置一个回调对象 `okhttp3.Callback`，然后在这个回调中再来回调我们传递进去的 `retrofit2.Callback`。这样，外部也就可以收到网络请求成功或者失败的事件回调了

```
final class OkHttpCall<T> implements Call<T> {



  @Override

  public void enqueue(final Callback<T> callback) {

    ...

    okhttp3.Call call;

    ...

    call.enqueue( new okhttp3.Callback() {

          @Override

          public void onResponse(okhttp3.Call call, okhttp3.Response rawResponse) {

            Response<T> response;

            try {

              response = parseResponse(rawResponse);

            } catch (Throwable e) {

              throwIfFatal(e);

              callFailure(e);

              return;

            }
```

```
      try {
        callback.onResponse(OkHttpCall.this, response);
      } catch (Throwable t) {
        throwIfFatal(t);
        t.printStackTrace(); // TODO this is not great
      }
    }

    @Override
    public void onFailure(okhttp3.Call call, IOException e) {
      callFailure(e);
    }

    private void callFailure(Throwable e) {
      try {
        callback.onFailure(OkHttpCall.this, e);
      } catch (Throwable t) {
        throwIfFatal(t);
        t.printStackTrace(); // TODO this is not great
      }
    }
  });
```

```
  }



   ...



}
```

复制代码

## 八、API 方法是如何解析的？

**Retrofit 是如何将 interface 内部的方法转化为一个个实际的 GET、POST、DELETE 等各式各样的网络请求的呢？例如，Retrofit 是如何将 getUserData() 转换为一个 OkHttp 的 GET 请求的呢？**

这个过程在 ServiceMethod 的 `parseAnnotations` 方法中就已经完成的了，对应的是 `RequestFactory.parseAnnotations(retrofit, method)`方法

```
abstract class ServiceMethod<T> {

  static <T> ServiceMethod<T> parseAnnotations(Retrofit retrofit, Method method) {

    //重点

    RequestFactory requestFactory = RequestFactory.parseAnnotations(retrofit, metho
d);



    Type returnType = method.getGenericReturnType();

    if (Utils.hasUnresolvableType(returnType)) {

      throw methodError(

          method,
```

```
          "Method return type must not include a type variable or wildcard: %s",
          returnType);
    }
    if (returnType == void.class) {
      throw methodError(method, "Service methods cannot return void.");
    }

    return HttpServiceMethod.parseAnnotations(retrofit, method, requestFactory);
  }

  abstract @Nullable T invoke(Object[] args);
}
```

复制代码

前文说了，Retrofit 是建立在 OkHttp 之上的一个网络请求封装库，内部依靠 OkHttp 来完成实际的网络请求。而 OkHttp 的一般请求方式如下所示

```
OkHttpClient client = new OkHttpClient();

String run(String url) throws IOException {
  Request request = new Request.Builder()
      .url(url)
      .build();

  try (Response response = client.newCall(request).execute()) {
```

```
    return response.body().string();

  }

}
```

复制代码

OkHttp 需要构建一个 Request 对象来配置请求方式和请求参数，并以此来发起网络请求。所以，Retrofit 也需要一个构建 Request 对象的过程，这个过程就隐藏在 RequestFactory 中

RequestFactory 采用了 Builder 模式，这里无需过多理会其构建过程，我们只要知道 RequestFactory 中包含了对 API 方法的各项解析结果即可。
其 `create(Object[] args)`方法就会根据各项解析结果，最终返回一个 `okhttp3.Request` 对象

```
final class RequestFactory {

  static RequestFactory parseAnnotations(Retrofit retrofit, Method method) {

    return new Builder(retrofit, method).build();

  }



  private final Method method;

  private final HttpUrl baseUrl;

  final String httpMethod;

  private final @Nullable String relativeUrl;

  private final @Nullable Headers headers;

  private final @Nullable MediaType contentType;

  private final boolean hasBody;

  private final boolean isFormEncoded;
```

```
  private final boolean isMultipart;

  private final ParameterHandler<?>[] parameterHandlers;

  final boolean isKotlinSuspendFunction;



  okhttp3.Request create(Object[] args) throws IOException {

    @SuppressWarnings("unchecked") // It is an error to invoke a method with the wron
g arg types.

    ParameterHandler<Object>[] handlers = (ParameterHandler<Object>[]) parameterHand
lers;



    int argumentCount = args.length;

    if (argumentCount != handlers.length) {

      throw new IllegalArgumentException(

          "Argument count ("

              + argumentCount

              + ") doesn't match expected count ("

              + handlers.length

              + ")");

    }



    RequestBuilder requestBuilder =

        new RequestBuilder(

            httpMethod,

            baseUrl,
```

```
          relativeUrl,
          headers,
          contentType,
          hasBody,
          isFormEncoded,
          isMultipart);

    if (isKotlinSuspendFunction) {
      // The Continuation is the last parameter and the handlers array contains null at that index.
      argumentCount--;
    }

    List<Object> argumentList = new ArrayList<>(argumentCount);
    for (int p = 0; p < argumentCount; p++) {
      argumentList.add(args[p]);
      handlers[p].apply(requestBuilder, args[p]);
    }

    return requestBuilder.get().tag(Invocation.class, new Invocation(method, argumentList)).build();
  }

}
```

```
复制代码
```

我们现在知道，OkHttpCall 是实际上发起网络请求的地方，所以最终 RequestFactory 的 `create` 方法会由 OkHttpCall 的 `createRawCall()` 方法来调用

```
final class OkHttpCall<T> implements Call<T> {



    private okhttp3.Call createRawCall() throws IOException {

        okhttp3.Call call = callFactory.newCall(requestFactory.create(args));

        if (call == null) {

            throw new NullPointerException("Call.Factory returned null.");

        }

        return call;

    }



}

复制代码
```

## 九、ResponseBody 是如何映射为 UserBean 的？

**Retrofit 是如何将 API 的返回值映射为具体的 Bean 对象的呢？例如，ResponseBody 是如何映射为 UserBean 的呢？**

OkHttp 默认的接口返回值对象就是 ResponseBody，所以在使用 Retrofit 时如果不引入 `converter-gson`，我们只能将接口请求结果都定义为 ResponseBody，而不能是具体的 Bean 对象，因为 Retrofit 无法自动地完成 ResponseBody 到 UserBean 之间的转换操作，需要我们将这种转换规则告知 Retrofit

```
interface ApiService {
```

```
    @GET("getUserData")
    fun getUserData1(): Call<ResponseBody>

    @GET("getUserData")
    fun getUserData2(): Call<UserBean>

}
```

复制代码

这种转换规则被 Retrofit 定义为 Converter 接口，对应它的 `responseBodyConverter` 方法

```
public interface Converter<F, T> {

  @Nullable
  T convert(F value) throws IOException;

  abstract class Factory {

    //将 ResponseBody 转换为目标类型 type
    public @Nullable Converter<ResponseBody, ?> responseBodyConverter(
        Type type, Annotation[] annotations, Retrofit retrofit) {
      return null;
```

```
    }



    ...

  }

}
```

复制代码

为了能直接获取到 UserBean 对象，我们需要在构建 Retrofit 对象的时候添加 GsonConverterFactory

```
    val retrofit = Retrofit.Builder()

        .baseUrl("https://mockapi.eolinker.com/9IiwI82f58c23411240ed608ceca204b2f185
014507cbe3/")

        .addConverterFactory(GsonConverterFactory.create())

        .build()
```

复制代码

GsonConverterFactory 会根据目标类型 type，通过 Gson 来进行反序列化出 UserBean 对象

```
public final class GsonConverterFactory extends Converter.Factory {



  @Override

  public Converter<ResponseBody, ?> responseBodyConverter(Type type, Annotation[] an
notations,

      Retrofit retrofit) {

    TypeAdapter<?> adapter = gson.getAdapter(TypeToken.get(type));
```

```
    return new GsonResponseBodyConverter<>(gson, adapter);
  }

  ...

}

final class GsonResponseBodyConverter<T> implements Converter<ResponseBody, T> {
  private final Gson gson;
  private final TypeAdapter<T> adapter;

  GsonResponseBodyConverter(Gson gson, TypeAdapter<T> adapter) {
    this.gson = gson;
    this.adapter = adapter;
  }

  @Override public T convert(ResponseBody value) throws IOException {
    JsonReader jsonReader = gson.newJsonReader(value.charStream());
    try {
      T result = adapter.read(jsonReader);
      if (jsonReader.peek() != JsonToken.END_DOCUMENT) {
        throw new JsonIOException("JSON document was not fully consumed.");
      }
```

```
      return result;
    } finally {
      value.close();
    }
  }
}
```

复制代码

那么，问题又来了，**Retrofit 是如何知道什么类型才可以交由 GsonConverterFactory 来进行处理的呢？至少 ResponseBody 就不应该交由 GsonConverterFactory 来处理，Retrofit 如何进行选择呢？**

首先，当我们在构建 Retrofit 对象时传入了 GsonConverterFactory，最终 Retrofit 会对所有 Converter.Factory 进行排序，converterFactories 中 BuiltInConverters 会被默认排在第一位，BuiltInConverters 是 Retrofit 自带的对 ResponseBody 进行默认解析的 Converter.Factory 实现类

```
public final class Retrofit {


   public static final class Builder {
         public Retrofit build() {


      ...


      // Make a defensive copy of the converters.
      List<Converter.Factory> converterFactories =
          new ArrayList<>(
```

```
            1 + this.converterFactories.size() + platform.defaultConverterFactories
Size());



     // Add the built-in converter factory first. This prevents overriding its behav
ior but also

     // ensures correct behavior when using converters that consume all types.

     converterFactories.add(new BuiltInConverters());

     converterFactories.addAll(this.converterFactories);

     converterFactories.addAll(platform.defaultConverterFactories());



     ...

   }



}
```

复制代码

而 BuiltInConverters 的 `responseBodyConverter` 方法在目标类型并非 **ResponseBody、Void、Unit** 等三种类型的情况下会返回 null

```
final class BuiltInConverters extends Converter.Factory {



  @Override

  public @Nullable Converter<ResponseBody, ?> responseBodyConverter(

      Type type, Annotation[] annotations, Retrofit retrofit) {

    if (type == ResponseBody.class) {
```

```
      return Utils.isAnnotationPresent(annotations, Streaming.class)
          ? StreamingResponseBodyConverter.INSTANCE
          : BufferingResponseBodyConverter.INSTANCE;
    }
    if (type == Void.class) {
      return VoidResponseBodyConverter.INSTANCE;
    }
    if (checkForKotlinUnit) {
      try {
        if (type == Unit.class) {
          return UnitResponseBodyConverter.INSTANCE;
        }
      } catch (NoClassDefFoundError ignored) {
        checkForKotlinUnit = false;
      }
    }
    return null;
  }

  ...

}
```

复制代码

而 Retrofit 类的 `nextResponseBodyConverter` 方法就是为每一个 API 方法选择 Converter 进行返回值数据类型转换的方法。该方法会先遍历到 BuiltInConverters，发现其返回了 null，就会最终选择到 GsonResponseBodyConverter，从而完成数据解析。如果最终没有找到一个合适的处理器的话，就会抛出 IllegalArgumentException

```
public <T> Converter<ResponseBody, T> nextResponseBodyConverter(

      @Nullable Converter.Factory skipPast, Type type, Annotation[] annotations) {

    Objects.requireNonNull(type, "type == null");

    Objects.requireNonNull(annotations, "annotations == null");



    int start = converterFactories.indexOf(skipPast) + 1;

    for (int i = start, count = converterFactories.size(); i < count; i++) {

      Converter<ResponseBody, ?> converter =

          converterFactories.get(i).responseBodyConverter(type, annotations, this);

      if (converter != null) {

        //noinspection unchecked

        return (Converter<ResponseBody, T>) converter;

      }

    }

    ...

    throw new IllegalArgumentException(builder.toString());

  }
```

复制代码

## 十、Call 是如何替换为 Observable 的？

**Retrofit 是如何抽象不同的 API 返回值包装类的呢？例如，Call 是如何替换为 Observable 的？**

与上一节内容相类似，Retrofit 在默认情况下也只支持将 `retrofit2.Call` 作为 API 接口的返回数据类型包装类，为了支持返回 `Observable` 类型，我们需要在构建 Retrofit 的时候添加 RxJava2CallAdapterFactory

```
    val retrofit = Retrofit.Builder()

        .baseUrl("https://mockapi.eolinker.com/9IiwI82f58c23411240ed608ceca204b2f185
014507cbe3/")

        .addConverterFactory(GsonConverterFactory.create())

        .addCallAdapterFactory(RxJava2CallAdapterFactory.create())

        .build()

复制代码
```

Retrofit 将 `retrofit2.Call` 转换为 `Observable` 的这种规则抽象为了 CallAdapter 接口

```
public interface CallAdapter<R, T> {



  //返回具体的内部类型，即 UserBean

  Type responseType();



  //用于将 Call 转换为 Observable

  T adapt(Call<R> call);



  abstract class Factory {
```

```
    //用于提供将 Call<UserBean> 转换为 Observable<UserBean> 的 CallAdapter 对象
    //此处的 returnType 即 Observable<UserBean>
    //如果此 CallAdapter 无法完成这种数据类型的转换，那么就返回 null
    public abstract @Nullable CallAdapter<?, ?> get(
        Type returnType, Annotation[] annotations, Retrofit retrofit);

    ...

  }
}
```

复制代码

对于 RxJava2CallAdapterFactory 的 `get` 方法而言，如何返回值类型并非 **Completable、Flowable、Single、Maybe** 等类型的话就会返回 null，否则就返回 RxJava2CallAdapter 对象

```
public final class RxJava2CallAdapterFactory extends CallAdapter.Factory {

  ...

  @Override
  public @Nullable CallAdapter<?, ?> get(Type returnType, Annotation[] annotations,
Retrofit retrofit) {
```

```
        Class<?> rawType = getRawType(returnType);

        if (rawType == Completable.class) {

            // Completable is not parameterized (which is what the rest of this method
 deals with) so it

            // can only be created with a single configuration.

            return new RxJava2CallAdapter(

            Void.class, scheduler, isAsync, false, true, false, false, false, true);

        }



        boolean isFlowable = rawType == Flowable.class;

        boolean isSingle = rawType == Single.class;

        boolean isMaybe = rawType == Maybe.class;

        if (rawType != Observable.class && !isFlowable && !isSingle && !isMaybe) {

            return null;

        }

        ...



        return new RxJava2CallAdapter(responseType, scheduler, isAsync, isResult, isB
ody, isFlowable, isSingle, isMaybe, false);

  }

}
```

复制代码

对于本例子而言，最终 RxJava2CallAdapter 又会返回 CallExecuteObservable，CallExecuteObservable 又会在外部进行 subscribe

的时候调用 `call.execute()` 方法来发起网络请求，所以在上面的例子中我们并不需要显式地发起网络请求，而是在进行 subscribe 的时候就自动触发请求了，Observer 只需要等待网络请求结果自动回调出来即可

```
final class RxJava2CallAdapter<R> implements CallAdapter<R, Object> {

  ...

  @Override
  public Type responseType() {
    return responseType;
  }

  @Override
  public Object adapt(Call<R> call) {
    Observable<Response<R>> responseObservable =
        isAsync ? new CallEnqueueObservable<>(call) : new CallExecuteObservable<>(cal
l);

    ...

    return RxJavaPlugins.onAssembly(observable);
  }
}
```

复制代码

```
final class CallExecuteObservable<T> extends Observable<Response<T>> {
  private final Call<T> originalCall;
```

```
CallExecuteObservable(Call<T> originalCall) {
  this.originalCall = originalCall;
}

@Override
protected void subscribeActual(Observer<? super Response<T>> observer) {
  // Since Call is a one-shot type, clone it for each new observer.
  Call<T> call = originalCall.clone();
  CallDisposable disposable = new CallDisposable(call);
  observer.onSubscribe(disposable);
  if (disposable.isDisposed()) {
    return;
  }

  boolean terminated = false;
  try {
    //发起网络请求
    Response<T> response = call.execute();
    if (!disposable.isDisposed()) {
      //将请求结果传给外部
      observer.onNext(response);
    }
```

```
      if (!disposable.isDisposed()) {
        terminated = true;
        observer.onComplete();
      }
    } catch (Throwable t) {
      Exceptions.throwIfFatal(t);
      if (terminated) {
        RxJavaPlugins.onError(t);
      } else if (!disposable.isDisposed()) {
        try {
          observer.onError(t);
        } catch (Throwable inner) {
          Exceptions.throwIfFatal(inner);
          RxJavaPlugins.onError(new CompositeException(t, inner));
        }
      }
    }
  }

  ...

}
```

复制代码

**那么，问题又来了，Retrofit 是如何知道什么类型才可以交由 RxJava2CallAdapterFactory 来进行处理的呢？**

首先，当我们在构建 Retrofit 对象时传入了 RxJava2CallAdapterFactory，最终 Retrofit 会按照添加顺序对所有 CallAdapter.Factory 进行保存，且默认会在队尾添加一个 DefaultCallAdapterFactory，用于对包装类型为 `retrofit2.Call` 的情况进行解析

而 Retrofit 类的 `nextCallAdapter` 方法就是为每一个 API 方法选择 CallAdapter 进行返回值数据类型转换的方法。该方法会先遍历到 RxJava2CallAdapter ，发现其返回了非 null 值，之后就交由其进行处理

```
 public CallAdapter<?, ?> nextCallAdapter(

     @Nullable CallAdapter.Factory skipPast, Type returnType, Annotation[] annotatio
ns) {

    Objects.requireNonNull(returnType, "returnType == null");

    Objects.requireNonNull(annotations, "annotations == null");



    int start = callAdapterFactories.indexOf(skipPast) + 1;

    for (int i = start, count = callAdapterFactories.size(); i < count; i++) {

      CallAdapter<?, ?> adapter = callAdapterFactories.get(i).get(returnType, annota
tions, this);

      if (adapter != null) {

        return adapter;

      }

    }



    ...
```

```
    throw new IllegalArgumentException(builder.toString());
  }
```

复制代码

## 十一、整个数据转换流程再总结下

这里再来总结下上面两节关于 Retrofit 整个数据转换的流程的内容

在默认情况下，我们从回调 Callback 中取到的最原始的返回值类型是 `Response<ResponseBody>`

```
interface ApiService {


    @GET("getUserData")
    fun getUserData(): Call<ResponseBody>


}
fun main() {
    val retrofit = Retrofit.Builder()
        .baseUrl("https://mockapi.eolinker.com/9IiwI82f58c23411240ed608ceca204b2f185014507cbe3/")
        .build()
    val service = retrofit.create(ApiService::class.java)
    val call: Call<ResponseBody> = service.getUserData()
    call.enqueue(object : Callback<ResponseBody> {
```

```
        override fun onResponse(call: Call<ResponseBody>, response: Response<Response
Body>) {

            val userBean = response.body()?.string()

            println("userBean: $userBean")

        }



        override fun onFailure(call: Call<ResponseBody>, t: Throwable) {

            println("onFailure: $t")

        }

    })

}
```

复制代码

而在引入了 `converter-gson` 和 `adapter-rxjava2` 之后，我们可以直接拿到目标类型 UserBean

```
/**

 * 作者：leavesC

 * 时间：2020/10/22 1:11

 * 描述：

 * GitHub：https://github.com/leavesC

 */data class UserBean(val userName: String, val userAge: Long)

interface ApiService {



    @GET("getUserData")
```

```
    fun getUserData(): Observable<UserBean>



}

fun main() {

    val retrofit = Retrofit.Builder()

        .baseUrl("https://mockapi.eolinker.com/9IiwI82f58c23411240ed608ceca204b2f185
014507cbe3/")

        .addConverterFactory(GsonConverterFactory.create())

        .addCallAdapterFactory(RxJava2CallAdapterFactory.createAsync())

        .build()

    val service = retrofit.create(ApiService::class.java)

    val call: Observable<UserBean> = service.getUserData()

    call.subscribe(object : Consumer<UserBean> {

        override fun accept(userBean: UserBean?) {

            println("userBean: $userBean")

        }



    }, object : Consumer<Throwable> {

        override fun accept(t: Throwable?) {

            println("onFailure: $t")

        }

    })

}
```

复制代码

那么，Retrofit 要达到这种转换效果，就要先后进行两个步骤：

1. 将 `ResponseBody` 转换为 `UserBean`，从而可以得到 API 返回值 `Response<UserBean>`
2. 将 `Call` 转换为 `Observable`，Observable 直接从 `Response<UserBean>` 中把 UserBean 取出来作为返回值来返回，从而直接得到目标类型 UserBean

第一个步骤即**第九节**所讲的内容，ResponseBody 转为 UserBean 的转换规则是通过 Converter 接口来定义的

```
public interface Converter<F, T> {



  //用于将 F 类型转换为 T 类型

  @Nullable

  T convert(F value) throws IOException;



  ...



}
```

复制代码

这个过程的转换就发生在 OkHttpCall 中，`enqueue` 方法在拿到 OkHttp 返回的 `okhttp3.Response` 对象后，就通过调用 `parseResponse` 方法来完成转化为 `Response<T>`的逻辑，当中就调用了 Converter 接口的 `convert` 方法，从而得到返回值 `Response<UserBean>`

```
final class OkHttpCall<T> implements Call<T> {
```

```
@Override

public void enqueue(final Callback<T> callback) {

  call.enqueue(

      new okhttp3.Callback() {

        @Override

        public void onResponse(okhttp3.Call call, okhttp3.Response rawResponse) {

          Response<T> response;

          try {

            //重点

            response = parseResponse(rawResponse);

          } catch (Throwable e) {

            throwIfFatal(e);

            callFailure(e);

            return;

          }



          try {

            callback.onResponse(OkHttpCall.this, response);

          } catch (Throwable t) {

            throwIfFatal(t);

            t.printStackTrace(); // TODO this is not great

          }

        }
```

```
        });
    }



 private final Converter<ResponseBody, T> responseConverter;


 Response<T> parseResponse(okhttp3.Response rawResponse) throws IOException {
    ...
    ExceptionCatchingResponseBody catchingBody = new ExceptionCatchingResponseBody(r
awBody);
    try {
      //catchingBody 就是 ResponseBody 类型，将其转换为 T 类型
      T body = responseConverter.convert(catchingBody);
      //然后再将其包装为 Response<T> 类型
      return Response.success(body, rawResponse);
    } catch (RuntimeException e) {
      // If the underlying source threw an exception, propagate that rather than indi
cating it was
      // a runtime exception.
      catchingBody.throwIfCaught();
      throw e;
    }
  }
```

```
}
```

复制代码

第二个步骤即**第十节**所讲的内容， Call 转换为 Observable 的转换规则是通过 CallAdapter 接口来定义的

```
public interface CallAdapter<R, T> {


  Type responseType();


  //此方法就用于将 Call<R> 转为你希望的目标类型 T，例如：Observable<UserBean>
  T adapt(Call<R> call);


  ...


}
```

复制代码

在 CallEnqueueObservable 这个类中，通过自定义回调接口 CallCallback 来发起网络请求，从而拿到在第一个步骤解析完成后的数据，
即 `Response<UserBean>` 对象

```
final class CallEnqueueObservable<T> extends Observable<Response<T>> {

  private final Call<T> originalCall;


  CallEnqueueObservable(Call<T> originalCall) {
    this.originalCall = originalCall;
```

```
}


@Override
protected void subscribeActual(Observer<? super Response<T>> observer) {
  // Since Call is a one-shot type, clone it for each new observer.
  Call<T> call = originalCall.clone();
  CallCallback<T> callback = new CallCallback<>(call, observer);
  observer.onSubscribe(callback);
  if (!callback.isDisposed()) {
    //自定义回调，发起请求
    call.enqueue(callback);
  }
}


private static final class CallCallback<T> implements Disposable, Callback<T> {
  private final Call<?> call;
  private final Observer<? super Response<T>> observer;
  private volatile boolean disposed;
  boolean terminated = false;


  CallCallback(Call<?> call, Observer<? super Response<T>> observer) {
    this.call = call;
    this.observer = observer;
```

```
    }


    @Override
    public void onResponse(Call<T> call, Response<T> response) {
        ...
        //直接将 Response<T>传递出去，即 Response<UserBean> 对象
        observer.onNext(response);
        ...
    }


    ...
  }
}
```

复制代码

CallCallback 类同时持有着一个 observer 对象，该 observer 对象实际上又属于 BodyObservable 类。BodyObservable 在拿到了 `Response<UserBean>` 对象后，如果判断到此次网络请求属于成功状态的话，那么就直接取出 body （即 UserBean）传递出去。因此我们才可以直接拿到目标类型，而不包含任何包装类

```
final class BodyObservable<T> extends Observable<T> {

  private final Observable<Response<T>> upstream;


  BodyObservable(Observable<Response<T>> upstream) {
    this.upstream = upstream;
```

```
}

@Override
protected void subscribeActual(Observer<? super T> observer) {
  upstream.subscribe(new BodyObserver<T>(observer));
}

private static class BodyObserver<R> implements Observer<Response<R>> {
  private final Observer<? super R> observer;
  private boolean terminated;

  BodyObserver(Observer<? super R> observer) {
    this.observer = observer;
  }

  @Override
  public void onSubscribe(Disposable disposable) {
    observer.onSubscribe(disposable);
  }

  @Override
  public void onNext(Response<R> response) {
    if (response.isSuccessful()) {
```

```
        //如果本次网络请求成功，那么就直接取出 body 传递出去
        observer.onNext(response.body());
      } else {
        terminated = true;
        Throwable t = new HttpException(response);
        try {
          observer.onError(t);
        } catch (Throwable inner) {
          Exceptions.throwIfFatal(inner);
          RxJavaPlugins.onError(new CompositeException(t, inner));
        }
      }
    }

    ...
  }
}
```

复制代码

## 十二、如何实现以 Kotlin 协程的方式来调用？

**Retrofit** 的当前版本已经支持以 **Kotlin** 协程的方式来调用了，这里就来看下 **Retrofit** 是如何支持协程调用的

先导入所有需要使用到的依赖，因为本例子是纯 Kotlin 项目，所以就不导入 Android 平台的 Kotlin 协程支持库了

```
dependencies {

    implementation 'com.squareup.retrofit2:retrofit:2.9.0'

    implementation 'com.squareup.retrofit2:converter-gson:2.5.0'

    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.9'

}
```

复制代码

本例子通过 runBlocking 来启动一个协程，避免网络请求还未结束 main 线程就停止了。**需要注意的是，在实际开发中应该避免这样来使用协程，否则使用协程就没有多少意义了**

```
/**

 * 作者：leavesC

 * 时间：2020/10/19 22:00

 * 描述：

 * GitHub：https://github.com/leavesC

 */interface ApiService {



    @GET("getUserData")

    suspend fun getUserData(): UserBean



}

data class UserBean(val userName: String, val userAge: Long)
```

```
fun main() {

    val retrofit = Retrofit.Builder()

        .baseUrl("https://mockapi.eolinker.com/9IiwI82f58c23411240ed608ceca204b2f185
014507cbe3/")

        .addConverterFactory(GsonConverterFactory.create())

        .build()

    val service = retrofit.create(ApiService::class.java)

    runBlocking {

        val job: Job = launch {

            try {

                val userBean: UserBean = service.getUserData()

                println("userBean: $userBean")

            } catch (e: Throwable) {

                println("onFailure: $e")

            }

        }

    }

}
```

复制代码

在本例子中，`getUserData()`方法的返回值就不需要任何包装类了，我们直接声明目标数据类型就可以了，在使用上会比之前更加简洁方便

**好了，开始来分析下流程**

我们先为 ApiService 多声明几个方法，方便来分析规律。每个方法都使用 `suspend` 关键字进行修饰，标明其只能用于在协程中来调用

```
interface ApiService {



    @GET("getUserData")

    fun getUserData(): UserBean



    @GET("getUserData")

    suspend fun getUserData1(): UserBean



    @GET("getUserData")

    suspend fun getUserData2(id: String): UserBean



    @GET("getUserData")

    suspend fun getUserData3(id: String, limit: Int): UserBean



}
```

复制代码

Retrofit 是以 Java 语言实现的，但 suspend 关键字只能用于 Kotlin，两者就存在着“沟通障碍”，但只要调用方也属于 JVM 语言的话，那么按道理来说 Retrofit 就都是可以使用的，此处就通过 IDEA 将 ApiService 反编译为了以下的 Java 类，看下 suspend 函数在 Retrofit 的角度来看是怎么实现的

```
public interface ApiService {

   @GET("getUserData")

   @NotNull
```

```
    UserBean getUserData();



    @GET("getUserData")

    @Nullable

    Object getUserData1(@NotNull Continuation var1);



    @GET("getUserData")

    @Nullable

    Object getUserData2(@NotNull String var1, @NotNull Continuation var2);



    @GET("getUserData")

    @Nullable

    Object getUserData3(@NotNull String var1, int var2, @NotNull Continuation var3);

}
```

复制代码

可以看到，非 suspend 函数的转换结果还符合我们的心理预期，但是 suspend 函数就相差得比较大了，方法返回值类型都变为 Object，且在方法的参数列表的最后都被添加了一个 `kotlin.coroutines.Continuation` 参数。**这个参数是重点，后面会使用到**

在 RequestFactory 类中包含一个 `isKotlinSuspendFunction` 的布尔变量，就用来标记当前解析到的 Method 是否是 suspend 函数。在 RequestFactory 的 `build()` 方法中，会对 API 方法的每一个参数进行解析，当中就包含了检测当前解析的参数是否是属于最后一个参数的逻辑

```
RequestFactory build() {

      ...
```

```
      int parameterCount = parameterAnnotationsArray.length;

      parameterHandlers = new ParameterHandler<?>[parameterCount];

      for (int p = 0, lastParameter = parameterCount - 1; p < parameterCount; p++) {

        parameterHandlers[p] =

            //p == lastParameter 如果为 true 就说明当前解析的 parameterTypes[p] 是 API
方法的最后一个参数

            parseParameter(p, parameterTypes[p], parameterAnnotationsArray[p], p == l
astParameter);

      }

      ...

      return new RequestFactory(this);

}
```

复制代码

如果检测到最后一个参数类型就是 `Continuation.class` 的话，那么 `isKotlinSuspendFunction` 就会变成 true。这个检测逻辑就符合了上面所介绍的 Kotlin 类型的 ApiService 代码转换为 Java 形式后的变化规则

```
private @Nullable ParameterHandler<?> parseParameter(

        int p, Type parameterType, @Nullable Annotation[] annotations, boolean allowC
ontinuation) {

      ...

      if (result == null) {

        if (allowContinuation) {

          try {

            if (Utils.getRawType(parameterType) == Continuation.class) {
```

```
            isKotlinSuspendFunction = true;
            return null;
          }
        } catch (NoClassDefFoundError ignored) {
        }
      }
      throw parameterError(method, p, "No Retrofit annotation found.");
    }

    return result;
  }
```

复制代码

然后，在 HttpServiceMethod 的 `parseAnnotations` 方法中我们就会用到 `isKotlinSuspendFunction` 这个变量

```
static <ResponseT, ReturnT> HttpServiceMethod<ResponseT, ReturnT> parseAnnotations(
      Retrofit retrofit, Method method, RequestFactory requestFactory) {

    boolean isKotlinSuspendFunction = requestFactory.isKotlinSuspendFunction;
    boolean continuationWantsResponse = false;
    boolean continuationBodyNullable = false;

    Annotation[] annotations = method.getAnnotations();
```

```
    Type adapterType;
    if (isKotlinSuspendFunction) {
      ...
      //虽然 getUserData() 方法我们直接定义返回类型为 UserBean
      //但实际上 Retrofit 还是需要将返回类型转为 Call<UserBean>，使之符合我们上述的数据解析流程
      //所以，此处的 responseType 为 UserBean，adapterType 确是 Call<UserBean>
      adapterType = new Utils.ParameterizedTypeImpl(null, Call.class, responseType);
      annotations = SkipCallbackExecutorImpl.ensurePresent(annotations);
    } else {
      adapterType = method.getGenericReturnType();
    }

    CallAdapter<ResponseT, ReturnT> callAdapter =
        createCallAdapter(retrofit, method, adapterType, annotations);
    Type responseType = callAdapter.responseType();

    ...

    //重点
      return (HttpServiceMethod<ResponseT, ReturnT>)
          new SuspendForBody<>(
              requestFactory,
```

```
            callFactory,
            responseConverter,
            (CallAdapter<ResponseT, Call<ResponseT>>) callAdapter,
            continuationBodyNullable);
  }
```

复制代码

最终，对于本例子来说，`parseAnnotations` 方法最终的返回值是 **SuspendForBody**，它也是 HttpServiceMethod 的子类。其主要逻辑是：

1. 将 API 方法的最后一个参数强转为 `Continuation<ResponseT>` 类型，这符合上述的分析
2. 因为 isNullable 固定为 false，所以最终会调用 `KotlinExtensions.await(call, continuation)` 这个 Kotlin 的扩展函数

```
static final class SuspendForBody<ResponseT> extends HttpServiceMethod<ResponseT, Object> {
    private final CallAdapter<ResponseT, Call<ResponseT>> callAdapter;
    private final boolean isNullable;

    SuspendForBody(
        RequestFactory requestFactory,
        okhttp3.Call.Factory callFactory,
        Converter<ResponseBody, ResponseT> responseConverter,
        CallAdapter<ResponseT, Call<ResponseT>> callAdapter,
        boolean isNullable) {
      super(requestFactory, callFactory, responseConverter);
```

```
      this.callAdapter = callAdapter;
      this.isNullable = isNullable;
    }

    @Override
    protected Object adapt(Call<ResponseT> call, Object[] args) {
      call = callAdapter.adapt(call);
      //noinspection unchecked Checked by reflection inside RequestFactory.
      Continuation<ResponseT> continuation = (Continuation<ResponseT>) args[args.len
gth - 1];
      try {
        return isNullable
            ? KotlinExtensions.awaitNullable(call, continuation)
            : KotlinExtensions.await(call, continuation);
      } catch (Exception e) {
        return KotlinExtensions.suspendAndThrow(e, continuation);
      }
    }
  }
```

复制代码

`await()`方法就会以 `suspendCancellableCoroutine` 这个支持 cancel 的 CoroutineScope 作为作用域，依旧以 `Call.enqueue` 的方式来发起 OkHttp 请求，拿到 responseBody 后就透传出来，至此就完成了整个调用流程了

```kotlin
suspend fun <T : Any> Call<T>.await(): T {
  return suspendCancellableCoroutine { continuation ->
    continuation.invokeOnCancellation {
      cancel()
    }
    enqueue(object : Callback<T> {
      override fun onResponse(call: Call<T>, response: Response<T>) {
        if (response.isSuccessful) {
          val body = response.body()
          if (body == null) {
            val invocation = call.request().tag(Invocation::class.java)!!
            val method = invocation.method()
            val e = KotlinNullPointerException("Response from " +
                method.declaringClass.name +
                '.' +
                method.name +
                " was null but response body type was declared as non-null")
            continuation.resumeWithException(e)
          } else {
            continuation.resume(body)
          }
        } else {
          continuation.resumeWithException(HttpException(response))
```

```
        }
      }

      override fun onFailure(call: Call<T>, t: Throwable) {
        continuation.resumeWithException(t)
      }
    })
  }
}
```

复制代码

## 十三、Retrofit 对 Android 平台做了什么特殊支持？

上文有讲到，**Retrofit 的内部实现并不需要依赖于 Android 平台，而是可以用于任意的 Java 客户端，Retrofit 只是对 Android 平台进行了特殊实现而已**

**那么，Retrofit 具体是对 Android 平台做了什么特殊支持呢？**

在构建 Retrofit 对象的时候，我们可以选择传递一个 Platform 对象用于标记调用方所处的平台

```
public static final class Builder {

    private final Platform platform;

    private @Nullable okhttp3.Call.Factory callFactory;

    private @Nullable HttpUrl baseUrl;

    private final List<Converter.Factory> converterFactories = new ArrayList<>();

    private final List<CallAdapter.Factory> callAdapterFactories = new ArrayList<>
();
```

```
    private @Nullable Executor callbackExecutor;
    private boolean validateEagerly;

    Builder(Platform platform) {
      this.platform = platform;
    }

    public Builder() {
      this(Platform.get());
    }

    ...
}
```

复制代码

Platform 类有两个作用：

1. 判断是否支持 Java 8。这在判断是否支持调用 interface 的默认方法，以及判断是否支持 Optional 和 CompletableFuture 时需要用到。因为 Android 应用如果想要支持 Java 8 的话，是需要在 Gradle 文件中进行主动配置的，且 Java 8 在 Android 平台上目前也支持得并不彻底，所以需要判断是否支持 Java 8 来决定是否启用特定功能
2. 实现 main 线程回调的 Executor。众所周知，Android 平台是不允许在 main 线程上执行耗时任务的，且 UI 操作都需要切换到 main 线程来完成。所以，对于 Android 平台来说，Retrofit 在回调网络请求结果时，都会通过 main 线程执行的 Executor 来进行线程切换

```
class Platform {

  private static final Platform PLATFORM = findPlatform();



  static Platform get() {

    return PLATFORM;

  }



  private static Platform findPlatform() {

    //根据 JVM 名字来判断使用方是否是 Android 平台

    return "Dalvik".equals(System.getProperty("java.vm.name"))

        ? new Android() //

        : new Platform(true);

  }



  //是否支持 Java 8

  private final boolean hasJava8Types;

  private final @Nullable Constructor<Lookup> lookupConstructor;



  Platform(boolean hasJava8Types) {

    this.hasJava8Types = hasJava8Types;



    Constructor<Lookup> lookupConstructor = null;

    if (hasJava8Types) {
```

```
      try {
        // Because the service interface might not be public, we need to use a MethodHandle lookup
        // that ignores the visibility of the declaringClass.
        lookupConstructor = Lookup.class.getDeclaredConstructor(Class.class, int.class);
        lookupConstructor.setAccessible(true);
      } catch (NoClassDefFoundError ignored) {
        // Android API 24 or 25 where Lookup doesn't exist. Calling default methods on non-public
        // interfaces will fail, but there's nothing we can do about it.
      } catch (NoSuchMethodException ignored) {
        // Assume JDK 14+ which contains a fix that allows a regular lookup to succeed.
        // See https://bugs.openjdk.java.net/browse/JDK-8209005.
      }
    }
    this.lookupConstructor = lookupConstructor;
  }


  //获取默认的 Executor 实现，用于 Android 平台
  @Nullable
  Executor defaultCallbackExecutor() {
    return null;
  }
```

```
  List<? extends CallAdapter.Factory> defaultCallAdapterFactories(

      @Nullable Executor callbackExecutor) {

    DefaultCallAdapterFactory executorFactory = new DefaultCallAdapterFactory(callba
ckExecutor);

    return hasJava8Types

        ? asList(CompletableFutureCallAdapterFactory.INSTANCE, executorFactory)

        : singletonList(executorFactory);

  }



  int defaultCallAdapterFactoriesSize() {

    return hasJava8Types ? 2 : 1;

  }



  List<? extends Converter.Factory> defaultConverterFactories() {

    return hasJava8Types ? singletonList(OptionalConverterFactory.INSTANCE) : emptyL
ist();

  }



  int defaultConverterFactoriesSize() {

    return hasJava8Types ? 1 : 0;

  }



  @IgnoreJRERequirement // Only called on API 24+.
```

```
  boolean isDefaultMethod(Method method) {

    return hasJava8Types && method.isDefault();

  }



  @IgnoreJRERequirement // Only called on API 26+.

  @Nullable

  Object invokeDefaultMethod(Method method, Class<?> declaringClass, Object object,
Object... args)

      throws Throwable {

    Lookup lookup =

        lookupConstructor != null

            ? lookupConstructor.newInstance(declaringClass, -1 /* trusted */)

            : MethodHandles.lookup();

    return lookup.unreflectSpecial(method, declaringClass).bindTo(object).invokeWith
Arguments(args);

  }



}
```

复制代码

Platform 类只具有一个唯一子类，即 Android 类。其主要逻辑就是重写了父类的 `defaultCallbackExecutor()`方法，通过 Handler 来实现在 main 线程执行特定的 Runnable，以此来实现网络请求结果都在 main 线程进行回调

```
static final class Android extends Platform {

    Android() {
```

```
      super(Build.VERSION.SDK_INT >= 24);
    }

    @Override
    public Executor defaultCallbackExecutor() {
      return new MainThreadExecutor();
    }

    @Nullable
    @Override
    Object invokeDefaultMethod(
        Method method, Class<?> declaringClass, Object object, Object... args) throws
Throwable {
      if (Build.VERSION.SDK_INT < 26) {
        throw new UnsupportedOperationException(
            "Calling default methods on API 24 and 25 is not supported");
      }
      return super.invokeDefaultMethod(method, declaringClass, object, args);
    }

    static final class MainThreadExecutor implements Executor {
      private final Handler handler = new Handler(Looper.getMainLooper());
```

```
    @Override
    public void execute(Runnable r) {
      handler.post(r);
    }
  }
}
```

复制代码

前文也有讲到，Retrofit 有个默认的 `CallAdapter.Factory` 接口实现类，用于对 API 返回值包装类型是 `Call` 的情形进行处理。DefaultCallAdapterFactory 会拿到 Platform 返回的 Executor 对象，如果 Executor 对象不为 null 且 API 方法没有标注 SkipCallbackExecutor 注解的话，就使用该 Executor 对象作为一个代理来中转所有的回调操作，以此实现线程切换。这里使用到了**装饰器模式**

```
final class DefaultCallAdapterFactory extends CallAdapter.Factory {
  private final @Nullable Executor callbackExecutor;

  DefaultCallAdapterFactory(@Nullable Executor callbackExecutor) {
    this.callbackExecutor = callbackExecutor;
  }

  @Override
  public @Nullable CallAdapter<?, ?> get(
      Type returnType, Annotation[] annotations, Retrofit retrofit) {

    ...
```

```
  final Executor executor =
      //判断 annotations 是否包含 SkipCallbackExecutor 注解
      //有的话说明希望直接在原来的线程进行方法调用，不需要进行线程切换
      Utils.isAnnotationPresent(annotations, SkipCallbackExecutor.class)
          ? null
          : callbackExecutor;

  return new CallAdapter<Object, Call<?>>() {
    @Override
    public Type responseType() {
      return responseType;
    }

    @Override
    public Call<Object> adapt(Call<Object> call) {
      //executor 不为 null 的话就将其作为一个中间代理
      //交由 ExecutorCallbackCall 来完成实际的回调操作
      return executor == null ? call : new ExecutorCallbackCall<>(executor, call);
    }
  };
}
```

```
  static final class ExecutorCallbackCall<T> implements Call<T> {
    final Executor callbackExecutor;
    final Call<T> delegate;

    ExecutorCallbackCall(Executor callbackExecutor, Call<T> delegate) {
      this.callbackExecutor = callbackExecutor;
      this.delegate = delegate;
    }

    @Override
    public void enqueue(final Callback<T> callback) {
      Objects.requireNonNull(callback, "callback == null");

      delegate.enqueue(
          new Callback<T>() {
            @Override
            public void onResponse(Call<T> call, final Response<T> response) {
              callbackExecutor.execute(
                  () -> {
                    if (delegate.isCanceled()) {
                      // Emulate OkHttp's behavior of throwing/delivering an IOExcepti
on on
                      // cancellation.
```

```
                    callback.onFailure(ExecutorCallbackCall.this, new IOException("C
anceled"));

                  } else {

                    callback.onResponse(ExecutorCallbackCall.this, response);

                  }

                });

            }



            @Override

            public void onFailure(Call<T> call, final Throwable t) {

              callbackExecutor.execute(() -> callback.onFailure(ExecutorCallbackCall.
this, t));

            }

          });

    }



    ...



  }

}
```

复制代码

## 十四、结尾

Retrofit 的源码就讲到这里了，自我感觉还是讲得挺全面的，虽然可能讲得没那么易于理解 =_= 从开始看源码到写完文章花了要十天出一些，断断续续地看源码，断断续续地写文章，总算写完了。觉得对你有所帮助就请点个赞吧

# 三方库源码笔记（8）-Retrofit 与 LiveData 的结合使用

在上篇文章中我讲解了 Retrofit 是如何实现支持不同的 API 返回值的。例如，对于同一个 API 接口，我们既可以使用 Retrofit 原生的 `Call<ResponseBody>`方式来作为返回值，也可以使用 `Observable<ResponseBody>`这种 RxJava 的方式来发起网络请求

```
/**

 * 作者：leavesC

 * 时间：2020/10/24 12:45

 * 描述：

 * GitHub：https://github.com/leavesC

 */interface ApiService {



    //Retrofit 原始请求方式

    @GET("getUserData")

    fun getUserDataA(): Call<ResponseBody>



    //RxJava 的请求方式

    @GET("getUserData")

    fun getUserDataB(): Observable<ResponseBody>
```

```
}
```

复制代码

我们在搭建项目的网络请求框架的时候，一个重要的设计环节就是要**避免由于网络请求结果的异步延时回调导致内存泄漏情况的发生**，所以在使用 RxJava 的时候我们往往是会搭配 RxLifecycle 来一起使用。而 Google 推出的 Jetpack 组件一个很大的亮点就是提供了生命周期安全保障的 LiveData：从源码看 Jetpack（3）-LiveData 源码解析

LiveData 是基于观察者模式来实现的，也完全符合我们在进行网络请求时的使用习惯。所以，本篇文章就来动手实现一个 **LiveDataCallAdapter**，即实现以下方式的网络请求回调

```
interface ApiService {



    @GET("getUserData")

    fun getUserData(): LiveData<HttpWrapBean<UserBean>>



}

class MainActivity : AppCompatActivity() {



    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity_main)

        RetrofitManager.apiService.getUserData().observe(this, Observer {

                val userBean = it.data

        })

    }
```

```
}
```

复制代码

# 一、基础定义

假设我们的项目中 API 接口的返回值的数据格式都是如下所示。通过 status 来标明本次网络请求结果是否成功，在 data 里面存放具体的目标数据

```
{
    "status": 200,
    "msg": "success",
    "data": {

    }
}
```

复制代码

对应我们项目中的实际代码就是一个泛型类

```
data class HttpWrapBean<T>(val status: Int, val msg: String, val data: T) {

    val isSuccess: Boolean
        get() = status == 200

}
```

复制代码

所以，ApiService 就可以如下定义，用 LiveData 作为目标数据的包装类

```
data class UserBean(val userName: String, val userAge: Int)

interface ApiService {



    @GET("getUserData")

    fun getUserData(): LiveData<HttpWrapBean<UserBean>>



}
```

复制代码

而网络请求不可避免会有异常发生，我们还需要预定义几个 Exception，对常见的异常类型：**无网络** 或者 **status!=200** 的情况进行封装

```
sealed class BaseHttpException(

    val errorCode: Int,

    val errorMessage: String,

    val realException: Throwable?

) : Exception(errorMessage) {



    companion object {



        const val CODE_UNKNOWN = -1024
```

```
        const val CODE_NETWORK_BAD = -1025



        fun generateException(throwable: Throwable?): BaseHttpException {
            return when (throwable) {
                is BaseHttpException -> {
                    throwable
                }
                is SocketException, is IOException -> {
                    NetworkBadException("网络请求失败", throwable)
                }
                else -> {
                    UnknownException("未知错误", throwable)
                }
            }
        }


    }


}
/**
 * 由于网络原因导致 API 请求失败
 * @param errorMessage
 * @param realException
```

```
 */class NetworkBadException(errorMessage: String, realException: Throwable) :
    BaseHttpException(CODE_NETWORK_BAD, errorMessage, realException)
/**
 * API 请求成功了，但 code != successCode
 * @param bean
 */class ServerCodeNoSuccessException(bean: HttpWrapBean<*>) :
    BaseHttpException(bean.status, bean.msg, null)
/**
 * 未知错误
 * @param errorMessage
 * @param realException
 */class UnknownException(errorMessage: String, realException: Throwable?) :
    BaseHttpException(CODE_UNKNOWN, errorMessage, realException)
复制代码
```

而在网络请求失败的时候，我们往往是需要向用户 Toast 失败原因的，所以此时一样需要向 LiveData postValue，以此将异常情况回调出去。因为还需要一个可以根据 Throwable 来生成对应的 HttpWrapBean 对象的方法

```
data class HttpWrapBean<T>(val status: Int, val msg: String, val data: T) {

    companion object {

        fun error(throwable: Throwable): HttpWrapBean<*> {
            val exception = BaseHttpException.generateException(throwable)
```

```
            return HttpWrapBean(exception.errorCode, exception.errorMessage, null)
        }


    }


    val isSuccess: Boolean
        get() = status == 200


}
```

复制代码

## 二、LiveDataCallAdapter

首先需要继承 CallAdapter.Factory 类，在 LiveDataCallAdapterFactory 类中判断是否支持特定的 API 方法，在类型校验通过后返回 LiveDataCallAdapter

```
class LiveDataCallAdapterFactory private constructor() : CallAdapter.Factory() {


    companion object {


        fun create(): LiveDataCallAdapterFactory {
            return LiveDataCallAdapterFactory()
        }


    }
```

```
    override fun get(
        returnType: Type,
        annotations: Array<Annotation>,
        retrofit: Retrofit
    ): CallAdapter<*, *>? {
        if (getRawType(returnType) != LiveData::class.java) {
            //并非目标类型的话就直接返回 null
            return null
        }
        //拿到 LiveData 包含的内部泛型类型
        val responseType = getParameterUpperBound(0, returnType as ParameterizedType)
        require(getRawType(responseType) == HttpWrapBean::class.java) {
            "LiveData 包含的泛型类型必须是 HttpWrapBean"
        }
        return LiveDataCallAdapter<Any>(responseType)
    }

}
```

复制代码

LiveDataCallAdapter 的逻辑也比较简单，如果**网络请求成功且状态码等于 200** 则直接返回接口值，否则就需要根据不同的失败原因构建出不同的 HttpWrapBean 对象

```
/**

 * 作者：leavesC

 * 时间：2020/10/22 21:06

 * 描述：

 * GitHub：https://github.com/leavesC

 */class LiveDataCallAdapter<R>(private val responseType: Type) : CallAdapter<R, LiveData<R>> {



    override fun responseType(): Type {

        return responseType

    }



    override fun adapt(call: Call<R>): LiveData<R> {

        return object : LiveData<R>() {



            private val started = AtomicBoolean(false)



            override fun onActive() {

                //避免重复请求

                if (started.compareAndSet(false, true)) {

                    call.enqueue(object : Callback<R> {

                        override fun onResponse(call: Call<R>, response: Response<R>)
{

                            val body = response.body() as HttpWrapBean<*>
```

```
                        if (body.isSuccess) {
                            //成功状态，直接返回 body
                            postValue(response.body())
                        } else {
                            //失败状态，返回格式化好的 HttpWrapBean 对象
                            postValue(HttpWrapBean.error(ServerCodeNoSuccessExcept
ion(body)) as R)
                        }
                    }

                    override fun onFailure(call: Call<R>, t: Throwable) {
                        //网络请求失败，根据 Throwable 类型来构建 HttpWrapBean
                        postValue(HttpWrapBean.error(t) as R)
                    }
                })
            }
        }


    }
}
```

复制代码

然后在构建 Retrofit 的时候添加 LiveDataCallAdapterFactory

```
object RetrofitManager {



    private val retrofit = Retrofit.Builder()

        .baseUrl("https://getman.cn/mock/")

        .addConverterFactory(GsonConverterFactory.create())

        .addCallAdapterFactory(LiveDataCallAdapterFactory.create())

        .build()



    val apiService = retrofit.create(ApiService::class.java)



}
```

复制代码

然后就可以直接在 Activity 中发起网络请求了。当 Activity 处于后台时 LiveData 不会回调任何数据，避免了常见的内存泄漏和 NPE 问题

```
/**

 * 作者：leavesC

 * 时间：2020/10/24 12:39

 * 描述：

 * GitHub：https://github.com/leavesC
```

```
 */@Router(EasyRouterPath.PATH_RETROFIT)class LiveDataCallAdapterActivity : AppCompa
tActivity() {



    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity_live_data_call_adapter)

        btn_success.setOnClickListener {

            RetrofitManager.apiService.getUserDataSuccess().observe(this, Observer {

                if (it.isSuccess) {

                    showToast(it.toString())

                } else {

                    showToast("failed: " + it.msg)

                }

            })

        }

    }



    private fun showToast(msg: String) {

        Toast.makeText(this, msg, Toast.LENGTH_SHORT).show()

    }



}
```

复制代码

## 三、结尾

LiveDataCallAdapter 的实现逻辑挺简单的，在使用上也很简单。本篇文章也算作是在了解了 Retrofit 源码后所做的一个实战 AndroidOpenSourceDemo

# 三方库源码笔记（9）-超详细的 Glide 源码详解

Glide 的源码有点复杂，如果要细细展开来讲解，那么写个十篇文章也囊括不完 **以小点来划分，每个小点只包含 Glide 实现某个功能或目的时所涉及的流程，以此来简化理解难度，通过整合多个小的功能点来把控住 Glide 大的实现方向**

本文基于 Glide 当前的最新版本来进行讲解

```
dependencies {

    implementation 'com.github.bumptech.glide:glide:4.11.0'

    kapt 'com.github.bumptech.glide:compiler:4.11.0'

}
```

复制代码

## 一、前置准备

在开始看 Glide 源码前，需要先对 Glide 有一些基本的了解

Glide 的缓存机制分为**内存缓存**和**磁盘缓存**两级。默认情况下，Glide 会自动对加载的图片进行缓存，缓存途径就分为内存缓存和磁盘缓存两种，缓存逻辑均采用 LruCache 算法。例如，Glide 在加载一张网络图片前，会先后判断当前内存和磁盘中是否已经缓存了目标图片，有的话则进行复用，没有的话则再进行网络请求

在默认情况下，Glide 对于一张网络图片的取值路径按顺序如下所示：

1. 当启动一个加载图片的请求时，会先检查 ActiveResources 中是否有符合条件的图片，如果存在则直接取值，否则就执行下一步。ActiveResources 存储了当前**正在使用**的图片资源（例如，某个 ImageView 正在展示这张图片），ActiveResources 通过弱引用来持有该图片资源
2. 检查 MemoryCache 中是否有符合条件的图片，如果存在则直接取值，否则就执行下一步。MemoryCache 使用了 Lru 算法，用于在内存中缓存曾使用过但目前非使用状态的图片资源
3. 检查本地磁盘缓存 DiskCache 中是否有符合条件的图片，如果存在则进行解码取值，否则就执行下一步
4. 联网请求图片。当加载到图片后，会将图片缓存到内存和磁盘中，以便后续复用

所以说，Glide 的内存缓存分为 ActiveResources 和 MemoryCache 两级

此外，Glide 最终会缓存到磁盘的图片类型可以分为两类，一类是原始图片，一类是将原始图片进行各种压缩裁剪变换等各种转换操作后得到的图片。Glide 的磁盘缓存策略（DiskCacheStrategy）就分为以下五种，用于决定如何对这两类图片进行磁盘保存

| 磁盘缓存策略 | 缓存策略说明 |
| --- | --- |
| DiskCacheStrategy.NONE | 不缓存任何内容 |
| DiskCacheStrategy.ALL | 既缓存原始图片，也缓存转换过后的图片 |
| DiskCacheStrategy.DATA | 只缓存原始图片 |
| DiskCacheStrategy.RESOURCE | 只缓存转换过后的图片 |
| DiskCacheStrategy.AUTOMATIC | 由 Glide 根据图片资源类型来自动选择使用哪一种缓存策 |

当中，比较特殊的缓存策略是 **DiskCacheStrategy.AUTOMATIC**，该策略会根据要加载的图片来源类型采用最佳的缓存策略。如果加载的是远程图片，仅会存储原始图片，不存储转换过后的图片，因为下载远程图片相比调整磁盘上已经存在的数据要昂贵得多。如果加载的是本地图片，则仅会存储转换过后的图片，因为即使需要再次生成另一个尺寸或类型的图片，取回原始图片也很容易

由于磁盘空间是有限的，所以 AUTOMATIC 是在衡量**所占磁盘空间大小**和**获取图片的成本**两者所做的一个居中选择

## 二、如何监听生命周期

通常，我们加载的图片最终是要显示在 ImageView 中的，而 ImageView 是会挂载在 Activity 或者 Fragment 等容器上的，当容器处于后台或者已经被 finish 时，此时加载图片的操作就应该被取消或者停止，否则就容易发生内存泄露或者 NPE 问题。那么，显而易见的一个问题就是，Glide 是如何判断容器是否还处于活跃状态的呢？

**类似于 Jetpack 组件中的 Lifecycle 的实现思路，Glide 也是通过一个无界面的 Fragment 来间接获取容器的生命周期状态的。Lifecycle 的实现思路可以看我的这篇源码讲解文章：从源码看 Jetpack（1） -Lifecycle 源码解析**

Glide 实现**生命周期监听**涉及到的类包含以下几个：

1. LifecycleListener
2. Lifecycle
3. ActivityFragmentLifecycle
4. ApplicationLifecycle
5. SupportRequestManagerFragment

首先，LifecycleListener 定义了三种事件通知回调，用于通知容器的活跃状态（是处于前台、后台、还是已经退出了）。Lifecycle 用于注册和移除 LifecycleListener

```
/**

 * An interface for listener to {@link android.app.Fragment} and {@link android.app.
Activity}

 * lifecycle events.

 */public interface LifecycleListener {



  /**

   * Callback for when {@link android.app.Fragment#onStart()}} or {@link

   * android.app.Activity#onStart()} is called.

   */
```

```
  void onStart();



  /**

   * Callback for when {@link android.app.Fragment#onStop()}} or {@link

   * android.app.Activity#onStop()}} is called.

   */

  void onStop();



  /**

   * Callback for when {@link android.app.Fragment#onDestroy()}} or {@link

   * android.app.Activity#onDestroy()} is called.

   */

  void onDestroy();

}
```

复制代码

```
/** An interface for listening to Activity/Fragment lifecycle events. */publi
c interface Lifecycle {

  /** Adds the given listener to the set of listeners managed by this Lifecycle imple
mentation. */

  void addListener(@NonNull LifecycleListener listener);



  /**

   * Removes the given listener from the set of listeners managed by this Lifecycle i
mplementation,

   * returning {@code true} if the listener was removed successfully, and {@code fals
e} otherwise.
```

```
     *
     * <p>This is an optimization only, there is no guarantee that every added listener
     will
     * eventually be removed.
     */
    void removeListener(@NonNull LifecycleListener listener);
}
```

复制代码

对于一个容器实例，例如在一个 Activity 的整个生命周期中，Activity 可能会先后加载多张图片，相应的就需要先后启动多个加载图片的后台任务，当 Activity 的生命周期状态发生变化时，就需要通知到每个后台任务。这一整个通知过程就对应 ActivityFragmentLifecycle 这个类

ActivityFragmentLifecycle 用 **isStarted** 和 **isDestroyed** 两个布尔变量来标记 Activity 的当前活跃状态，并提供了保存并通知多个 LifecycleListener 的能力

```
class ActivityFragmentLifecycle implements Lifecycle {
    private final Set<LifecycleListener> lifecycleListeners =
        Collections.newSetFromMap(new WeakHashMap<LifecycleListener, Boolean>());
    private boolean isStarted;
    private boolean isDestroyed;

    /**
     * Adds the given listener to the list of listeners to be notified on each lifecycl
    e event.
     *
```

```
   * <p>The latest lifecycle event will be called on the given listener synchronously
 in this

   * method. If the activity or fragment is stopped, {@link LifecycleListener#onStop
()}} will be

   * called, and same for onStart and onDestroy.

   *

   * <p>Note - {@link com.bumptech.glide.manager.LifecycleListener}s that are added
more than once

   * will have their lifecycle methods called more than once. It is the caller's resp
onsibility to

   * avoid adding listeners multiple times.

   */

  @Override

  public void addListener(@NonNull LifecycleListener listener) {

    lifecycleListeners.add(listener);



    if (isDestroyed) {

      listener.onDestroy();

    } else if (isStarted) {

      listener.onStart();

    } else {

      listener.onStop();

    }

  }
```

```
  @Override
  public void removeListener(@NonNull LifecycleListener listener) {
    lifecycleListeners.remove(listener);
  }

  void onStart() {
    isStarted = true;
    for (LifecycleListener lifecycleListener : Util.getSnapshot(lifecycleListeners))
{
      lifecycleListener.onStart();
    }
  }

  void onStop() {
    isStarted = false;
    for (LifecycleListener lifecycleListener : Util.getSnapshot(lifecycleListeners))
{
      lifecycleListener.onStop();
    }
  }

  void onDestroy() {
    isDestroyed = true;
```

```
    for (LifecycleListener lifecycleListener : Util.getSnapshot(lifecycleListeners))
 {

      lifecycleListener.onDestroy();

    }

  }

}
```

复制代码

ActivityFragmentLifecycle 用于 SupportRequestManagerFragment 这个 Fragment 中来使用（省略了部分代码）。可以看到，在 Fragment 的三个生命周期回调事件中，都会相应通知 ActivityFragmentLifecycle。那么，不管 ImageView 的载体是 Activity 还是 Fragment，我们都可以向其注入一个无界面的 SupportRequestManagerFragment，以此来监听载体在整个生命周期内活跃状态的变化

```
public class SupportRequestManagerFragment extends Fragment {



  private static final String TAG = "SupportRMFragment";



  private final ActivityFragmentLifecycle lifecycle;



  public SupportRequestManagerFragment() {

    this(new ActivityFragmentLifecycle());

  }



  @VisibleForTesting

  @SuppressLint("ValidFragment")
```

```
public SupportRequestManagerFragment(@NonNull ActivityFragmentLifecycle lifecycle)
{
  this.lifecycle = lifecycle;
}

@NonNull
ActivityFragmentLifecycle getGlideLifecycle() {
  return lifecycle;
}

@Override
public void onStart() {
  super.onStart();
  lifecycle.onStart();
}

@Override
public void onStop() {
  super.onStop();
  lifecycle.onStop();
}

@Override
```

```
  public void onDestroy() {
    super.onDestroy();
    lifecycle.onDestroy();
    unregisterFragmentWithRoot();
  }

  @Override
  public String toString() {
    return super.toString() + "{parent=" + getParentFragmentUsingHint() + "}";
  }

}
```

复制代码

还有种特殊情况，就是我们加载的图片并不是最终要挂载在 Activity 上的，而只是想下载图片而已，此时我们传给 Glide 的 Context 可能就是 Application 了，此时 Lifecycle 对应的实现类就是 ApplicationLifecycle，默认且一直都处于 onStart 状态

```
class ApplicationLifecycle implements Lifecycle {

  @Override
  public void addListener(@NonNull LifecycleListener listener) {
    listener.onStart();
  }
```

```
  @Override
  public void removeListener(@NonNull LifecycleListener listener) {
    // Do nothing.
  }
}
```

复制代码

## 三、怎么注入 Fragment

SupportRequestManagerFragment 用于通知事件，那么 SupportRequestManagerFragment 是如何挂载到 Activity 或者 Fragment 上的呢？

通过查找引用，可以定位到是在 RequestManagerRetriever 的 `getSupportRequestManagerFragment` 方法中完成 SupportRequestManagerFragment 的注入

```
public class RequestManagerRetriever implements Handler.Callback {



  @NonNull
  private SupportRequestManagerFragment getSupportRequestManagerFragment(
      @NonNull final FragmentManager fm, @Nullable Fragment parentHint, boolean isPar
entVisible) {
    //通过 TAG 判断 FragmentManager 中是否已经包含了 SupportRequestManagerFragment
    SupportRequestManagerFragment current =
        (SupportRequestManagerFragment) fm.findFragmentByTag(FRAGMENT_TAG);
    if (current == null) {
      //current 为 null 说明还未注入过 SupportRequestManagerFragment
```

```
      //那么就构建一个 SupportRequestManagerFragment 实例并添加到 FragmentManager 中



      current = pendingSupportRequestManagerFragments.get(fm);

      if (current == null) {

        current = new SupportRequestManagerFragment();

        current.setParentFragmentHint(parentHint);

        if (isParentVisible) {

          current.getGlideLifecycle().onStart();

        }

        pendingSupportRequestManagerFragments.put(fm, current);

        fm.beginTransaction().add(current, FRAGMENT_TAG).commitAllowingStateLoss();

        handler.obtainMessage(ID_REMOVE_SUPPORT_FRAGMENT_MANAGER, fm).sendToTarget
();

      }

    }

    return current;

  }



}
```

复制代码

那具体的注入时机是在什么时候呢？

我们使用 Glide 来加载一张图片往往是像以下所示那么的朴实无华，一行代码就搞定，Glide 在背后悄悄做了成吨的工作量

```
Glide.with(FragmentActivity).load(url).into(ImageView)

复制代码
```

当调用 `Glide.with(FragmentActivity)` 时，最终是会中转调用到 RequestManagerRetriever 的 `get(FragmentActivity)` 方法，在内部调用 `supportFragmentGet` 方法完成 SupportRequestManagerFragment 的注入，并最终返回一个 RequestManager 对象

```
public class RequestManagerRetriever implements Handler.Callback {



  @NonNull

  public RequestManager get(@NonNull FragmentActivity activity) {

    if (Util.isOnBackgroundThread()) {

      //如果是后台线程的话，那么就使用 ApplicationLifecycle

      return get(activity.getApplicationContext());

    } else {

      assertNotDestroyed(activity);

      FragmentManager fm = activity.getSupportFragmentManager();

      return supportFragmentGet(activity, fm, /*parentHint=*/ null, isActivityVisibl
e(activity));

    }

  }



  @NonNull

  private RequestManager supportFragmentGet(

      @NonNull Context context,
```

```
      @NonNull FragmentManager fm,
      @Nullable Fragment parentHint,
      boolean isParentVisible) {
    //在这里完成 SupportRequestManagerFragment 的注入操作
    SupportRequestManagerFragment current =
        getSupportRequestManagerFragment(fm, parentHint, isParentVisible);
    RequestManager requestManager = current.getRequestManager();
    if (requestManager == null) {
      // TODO(b/27524013): Factor out this Glide.get() call.
      Glide glide = Glide.get(context);
      requestManager =
          factory.build(
              glide, current.getGlideLifecycle(), current.getRequestManagerTreeNode(),
 context);
      current.setRequestManager(requestManager);
    }
    return requestManager;
  }


}
```

复制代码

所以说，当我们调用 `Glide.with(FragmentActivity)`方法时，此时就已经完成了 SupportRequestManagerFragment 的注入

而 RequestManagerRetriever 一共包含几种入参类型的 get 方法重载

1. Context
2. androidx.fragment.app.FragmentActivity
3. android.app.Activity
4. androidx.fragment.app.Fragment
5. android.app.Fragment（已废弃）
6. View

这几个 get 方法的逻辑可以总结为：

1. 如果外部是通过子线程来调用的，那么就统一使用 Application，此时就不需要注入 Fragment，直接使用 ApplicationLifecycle，不进行生命周期观察，默认外部会一直处于活跃状态
2. 如果外部传入的是 Application，那么步骤同上
3. 如果外部传入的 View 并没有关联到 Activity（例如，View 包含的 Context 属于 ServiceContext 类型），那么步骤同上
4. 除以上情况外，最终都会通过外部传入的参数查找到关联的 Activity 或者 Fragment，最终向其注入 RequestManagerFragment 或者 SupportRequestManagerFragment

    RequestManagerFragment 的功能和 SupportRequestManagerFragment 相同，但目前已经是废弃状态，此处就不再赘述

例如，`get(@NonNull Context context)`就会根据调用者所在线程以及 Context 所属类型，来获取不同的 RequestManager

```
@NonNull

public RequestManager get(@NonNull Context context) {

  if (context == null) {
```

```
      throw new IllegalArgumentException("You cannot start a load on a null Context
");

    } else if (Util.isOnMainThread() && !(context instanceof Application)) {

      //在主线程调用，且 context 并非 Application



      if (context instanceof FragmentActivity) {

        return get((FragmentActivity) context);

      } else if (context instanceof Activity) {

        return get((Activity) context);

      } else if (context instanceof ContextWrapper

          // Only unwrap a ContextWrapper if the baseContext has a non-null applicati
on context.

          // Context#createPackageContext may return a Context without an Application
 instance,

          // in which case a ContextWrapper may be used to attach one.

          && ((ContextWrapper) context).getBaseContext().getApplicationContext() != n
ull) {

        return get(((ContextWrapper) context).getBaseContext());

      }

    }



    //在子线程调用或者 context 是 Application

    return getApplicationManager(context);

  }
```

复制代码

如果不注入 SupportRequestManagerFragment，那么最终使用的 RequestManager 对象就属于全员唯一的 Application 级别的 RequestManager

```
  /** The top application level RequestManager. */
  private volatile RequestManager applicationManager;


  @NonNull
  private RequestManager getApplicationManager(@NonNull Context context) {
    // Either an application context or we're on a background thread.
    if (applicationManager == null) {
      synchronized (this) {
        if (applicationManager == null) {
          // Normally pause/resume is taken care of by the fragment we add to the frag
ment or
          // activity. However, in this case since the manager attached to the applic
ation will not
          // receive lifecycle events, we must force the manager to start resumed usi
ng
          // ApplicationLifecycle.


          // TODO(b/27524013): Factor out this Glide.get() call.
          Glide glide = Glide.get(context.getApplicationContext());
          applicationManager =
```

```
            factory.build(
                glide,
                new ApplicationLifecycle(),
                new EmptyRequestManagerTreeNode(),
                context.getApplicationContext());
        }
      }
    }

    return applicationManager;
  }

复制代码
```

## 四、如何启动加载图片的任务

前文介绍了 Glide 是如何实现监听 Activity 的生命周期变化的，那么，Glide 是如何发起加载图片的任务的呢？

上面提到了，当我们调用了 `Glide.with(FragmentActivity)`时，就会完成 SupportRequestManagerFragment 的注入操作。且对于同一个 Activity 实例，在其整个完整的生命周期过程中只会注入一次。从 `supportFragmentGet` 方法也可以看到，每个 SupportRequestManagerFragment 也会包含一个 RequestManager 实例

```
public class RequestManagerRetriever implements Handler.Callback {


  @NonNull
```

```
  private RequestManager supportFragmentGet(

      @NonNull Context context,

      @NonNull FragmentManager fm,

      @Nullable Fragment parentHint,

      boolean isParentVisible) {

    //在这里完成 SupportRequestManagerFragment 的注入操作

    SupportRequestManagerFragment current =

        getSupportRequestManagerFragment(fm, parentHint, isParentVisible);

    RequestManager requestManager = current.getRequestManager();

    if (requestManager == null) {

      //如果 requestManager 为 null 就进行生成并设置到 SupportRequestManagerFragment 中

      // TODO(b/27524013): Factor out this Glide.get() call.

      Glide glide = Glide.get(context);

      requestManager =

          factory.build(

              glide, current.getGlideLifecycle(), current.getRequestManagerTreeNode(),
 context);

      current.setRequestManager(requestManager);

    }

    return requestManager;

  }



}
```

复制代码

RequestManager 类就是用于启动并管理某个 Activity 前后启动的所有加载图片的任务的地方，当我们完整调用 `Glide.with(FragmentActivity).load(url).into(ImageView)` 的 `into` 方法后，就会构建出一个代表当前加载任务的 Request 对象，并且将该任务传递给 RequestManager，以此开始跟踪该任务

```
@NonNull
public ViewTarget<ImageView, TranscodeType> into(@NonNull ImageView view) {
  ...
  return into(
      glideContext.buildImageViewTarget(view, transcodeClass),
      /*targetListener=*/ null,
      requestOptions,
      Executors.mainThreadExecutor());
}



private <Y extends Target<TranscodeType>> Y into(
    @NonNull Y target,
    @Nullable RequestListener<TranscodeType> targetListener,
    BaseRequestOptions<?> options,
    Executor callbackExecutor) {
  Preconditions.checkNotNull(target);
```

```
    if (!isModelSet) {

      throw new IllegalArgumentException("You must call #load() before calling #into()
");

    }

    //构建一个代表加载任务的 Request 对象

    Request request = buildRequest(target, targetListener, options, callbackExecuto
r);

    ...

    requestManager.clear(target);

    target.setRequest(request);

    //将 request 传递给 requestManager，以此开始跟踪该任务

    requestManager.track(target, request);

    return target;

  }



复制代码
```

重点还是 `requestManager.track(target, request)`这一句代码，这就是任务的发起点

```
public class RequestManager

    implements ComponentCallbacks2, LifecycleListener, ModelTypes<RequestBuilder<Dra
wable>> {



    //存储所有任务

    @GuardedBy("this")
```

```
    private final RequestTracker requestTracker;



    @GuardedBy("this")
    private final TargetTracker targetTracker = new TargetTracker();



    synchronized void track(@NonNull Target<?> target, @NonNull Request request) {
        targetTracker.track(target);
        //运行任务
        requestTracker.runRequest(request);
    }



}



复制代码
```

当中，RequestTracker 就用于存储所有的 Request，即存储所有加载图片的任务，并提供了**开始、暂停和重启**所有任务的方法。外部通过改变 **isPaused** 变量值，用来控制当前是否允许启动任务，`runRequest` 方法中就会根据 isPaused 来判断当前是**马上启动任务 begin() 还是将任务暂存到待处理列表 pendingRequests** 中

```
public class RequestTracker {

  private static final String TAG = "RequestTracker";



  private final Set<Request> requests =

      Collections.newSetFromMap(new WeakHashMap<Request, Boolean>());
```

```java
@SuppressWarnings("MismatchedQueryAndUpdateOfCollection")
private final List<Request> pendingRequests = new ArrayList<>();

private boolean isPaused;

/** Starts tracking the given request. */
public void runRequest(@NonNull Request request) {
  //先将任务保存起来
  requests.add(request);
  //如果并非暂停状态，那么就开启任务，否则就将任务存入待处理列表
  if (!isPaused) {
    request.begin();
  } else {
    request.clear();
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
      Log.v(TAG, "Paused, delaying request");
    }
    pendingRequests.add(request);
  }
}

/** Stops any in progress requests. */
```

```
  public void pauseRequests() {
    isPaused = true;
    for (Request request : Util.getSnapshot(requests)) {
      if (request.isRunning()) {
        // Avoid clearing parts of requests that may have completed (thumbnails) to a
void blinking
        // in the UI, while still making sure that any in progress parts of requests a
re immediately
        // stopped.
        request.pause();
        pendingRequests.add(request);
      }
    }
  }


  /** Restarts failed requests and cancels and restarts in progress requests. */
  public void restartRequests() {
    for (Request request : Util.getSnapshot(requests)) {
      if (!request.isComplete() && !request.isCleared()) {
        request.clear();
        if (!isPaused) {
          request.begin();
        } else {
          // Ensure the request will be restarted in onResume.
```

```
        pendingRequests.add(request);
      }
    }
  }

  ...

}

复制代码
```

当 SupportRequestManagerFragment 走到 `onStop()` 状态时，就会中转调用到 RequestTracker，将其 isPaused 变量置为 true。此外，当 SupportRequestManagerFragment 执行到 `onDestroy()` 时，就意味着 Activity 已经被 finish 了，此时就会回调通知到 RequestManager 的 `onDestroy()`方法，在这里完成任务的清理以及解除各种注册事件

```
  @Override
  public synchronized void onDestroy() {
    targetTracker.onDestroy();
    for (Target<?> target : targetTracker.getAll()) {
      clear(target);
    }
    targetTracker.clear();
    requestTracker.clearRequests();
```

```
    lifecycle.removeListener(this);

    lifecycle.removeListener(connectivityMonitor);

    mainHandler.removeCallbacks(addSelfToLifecycle);

    glide.unregisterRequestManager(this);

  }



复制代码
```

## 五、加载图片的具体流程

Request 是一个接口，对于本例子来说，其实际实现类是 SingleRequest，那么就来看其 `request.begin()`方法是如何实现的，即具体的加载图片的流程

`begin` 方法会先对当前的任务状态进行校验，防止重复加载，然后去获取**目标宽高**或者 **ImageView 的宽高**，之后还会判断是否需要先展示占位符

```
public final class SingleRequest<R> implements Request, SizeReadyCallback, ResourceC
allback {



  @Override

  public void begin() {

    synchronized (requestLock) {

      assertNotCallingCallbacks();

      stateVerifier.throwIfRecycled();

      startTime = LogTime.getLogTime();

      if (model == null) {

        if (Util.isValidDimensions(overrideWidth, overrideHeight)) {
```

```
        width = overrideWidth;
        height = overrideHeight;
      }
      // Only log at more verbose log levels if the user has set a fallback drawable, because
      // fallback Drawables indicate the user expects null models occasionally.
      int logLevel = getFallbackDrawable() == null ? Log.WARN : Log.DEBUG;
      //model 为 null，说明外部没有传入图片来源地址，直接走失败流程
      onLoadFailed(new GlideException("Received null model"), logLevel);
      return;
    }

    //防止任务正在运行时重复启动
    if (status == Status.RUNNING) {
      throw new IllegalArgumentException("Cannot restart a running request");
    }

    // If we're restarted after we're complete (usually via something like a notifyDataSetChanged
    // that starts an identical request into the same Target or View), we can simply use the
    // resource and size we retrieved the last time around and skip obtaining a new size, starting
    // a new load etc. This does mean that users who want to restart a load because they expect
```

```
      // that the view size has changed will need to explicitly clear the View or Tar
get before

      // starting the new load.

      if (status == Status.COMPLETE) {

        //任务已经完成，直接返回已加载好的图片资源

        onResourceReady(resource, DataSource.MEMORY_CACHE);

        return;

      }



      // Restarts for requests that are neither complete nor running can be treated a
s new requests

      // and can run again from the beginning.



      //先获取目标宽高或者 ImageView 的宽高，按需加载

      status = Status.WAITING_FOR_SIZE;

      if (Util.isValidDimensions(overrideWidth, overrideHeight)) {

        onSizeReady(overrideWidth, overrideHeight);

      } else {

        target.getSize(this);

      }



      if ((status == Status.RUNNING || status == Status.WAITING_FOR_SIZE)

          && canNotifyStatusChanged()) {

        //先把占位符传出去
```

```
      target.onLoadStarted(getPlaceholderDrawable());
    }
    if (IS_VERBOSE_LOGGABLE) {
      logV("finished run method in " + LogTime.getElapsedMillis(startTime));
    }
  }
}

复制代码
```

可以看到，以上逻辑还没有涉及到具体的加载图片的逻辑，因为这个过程还需要在获取到目标宽高后才能进行。如果外部有传入具体的宽高值，那么就以外部值为准，否则就以 target（例如 ImageView）的宽高大小为准。只有在获取到宽高后才会真正开始加载，这都是为了实现按需加载，避免内存浪费

所以，重点还是要看 `onSizeReady` 方法。其内部会将当前的所有配置信息（图片地址，宽高、优先级、是否允许使用缓存等等）都转交给 Engine 的 load 方法，由其来完成图片的加载

```
  private volatile Engine engine;


  /** A callback method that should never be invoked directly. */
  @Override
  public void onSizeReady(int width, int height) {
    stateVerifier.throwIfRecycled();
```

```
    synchronized (requestLock) {
      if (IS_VERBOSE_LOGGABLE) {
        logV("Got onSizeReady in " + LogTime.getElapsedMillis(startTime));
      }
      if (status != Status.WAITING_FOR_SIZE) {
        return;
      }
      status = Status.RUNNING;


      //进行缩放处理
      float sizeMultiplier = requestOptions.getSizeMultiplier();
      this.width = maybeApplySizeMultiplier(width, sizeMultiplier);
      this.height = maybeApplySizeMultiplier(height, sizeMultiplier);


      if (IS_VERBOSE_LOGGABLE) {
        logV("finished setup for calling load in " + LogTime.getElapsedMillis(startTi
me));
      }
      //重点，正式开始加载图片
      loadStatus =
          engine.load(
              glideContext,
              model,
```

```
            requestOptions.getSignature(),

            this.width,

            this.height,

            requestOptions.getResourceClass(),

            transcodeClass,

            priority,

            requestOptions.getDiskCacheStrategy(),

            requestOptions.getTransformations(),

            requestOptions.isTransformationRequired(),

            requestOptions.isScaleOnlyOrNoTransform(),

            requestOptions.getOptions(),

            requestOptions.isMemoryCacheable(),

            requestOptions.getUseUnlimitedSourceGeneratorsPool(),

            requestOptions.getUseAnimationPool(),

            requestOptions.getOnlyRetrieveFromCache(),

            this,

            callbackExecutor);



    // This is a hack that's only useful for testing right now where loads complete
 synchronously

    // even though under any executor running on any thread but the main thread, th
e load would

    // have completed asynchronously.

    if (status != Status.RUNNING) {
```

```
        loadStatus = null;

      }

      if (IS_VERBOSE_LOGGABLE) {

        logV("finished onSizeReady in " + LogTime.getElapsedMillis(startTime));

      }

    }

  }
```

复制代码

转交给 Engine 的配置信息同时还包含一个 ResourceCallback 对象，即 SingleRequest 本身，因为 SingleRequest 实现了 ResourceCallback 接口。从 ResourceCallback 包含的方法的名称来看，就可以知道当 Engine 在加载图片成功或者失败时，就会通过这两个方法将结果回调出来

```
public interface ResourceCallback {



  /**

   * Called when a resource is successfully loaded.

   *

   * @param resource The loaded resource.

   */

  void onResourceReady(Resource<?> resource, DataSource dataSource);



  /**
```

```
     * Called when a resource fails to load successfully.
     *
     * @param e a non-null {@link GlideException}.
     */
    void onLoadFailed(GlideException e);

    /** Returns the lock to use when notifying individual requests. */
    Object getLock();
}
```

复制代码

load 方法会先为本次请求生成一个唯一 key，这个 key 就是判定是否可以实现图片复用的依据，然后根据这个 key 从内存缓存中取值，如果取得到的话就直接进行复用，否则就启动一个新任务来从磁盘加载或者联网加载，或者是为已存在的任务添加一个回调

```
public <R> LoadStatus load(
        GlideContext glideContext,
        Object model,
        Key signature,
        int width,
        int height,
        Class<?> resourceClass,
        Class<R> transcodeClass,
```

```
    Priority priority,
    DiskCacheStrategy diskCacheStrategy,
    Map<Class<?>, Transformation<?>> transformations,
    boolean isTransformationRequired,
    boolean isScaleOnlyOrNoTransform,
    Options options,
    boolean isMemoryCacheable,
    boolean useUnlimitedSourceExecutorPool,
    boolean useAnimationPool,
    boolean onlyRetrieveFromCache,
    ResourceCallback cb,
    Executor callbackExecutor) {
  long startTime = VERBOSE_IS_LOGGABLE ? LogTime.getLogTime() : 0;


  //为本次请求生成一个唯一 key，这个 key 就是判定是否可以实现图片复用的依据
  EngineKey key =
      keyFactory.buildKey(
          model,
          signature,
          width,
          height,
          transformations,
          resourceClass,
```

```
            transcodeClass,
            options);


    EngineResource<?> memoryResource;
    synchronized (this) {
      //先从内存缓存中取值
      memoryResource = loadFromMemory(key, isMemoryCacheable, startTime);


      if (memoryResource == null) {
        //当前内存中不存在目标资源，那么就启动一个新任务来加载，或者是为已存在的任务添加一个回调
        return waitForExistingOrStartNewJob(
            glideContext,
            model,
            signature,
            width,
            height,
            resourceClass,
            transcodeClass,
            priority,
            diskCacheStrategy,
            transformations,
            isTransformationRequired,
```

```
            isScaleOnlyOrNoTransform,

            options,

            isMemoryCacheable,

            useUnlimitedSourceExecutorPool,

            useAnimationPool,

            onlyRetrieveFromCache,

            cb,

            callbackExecutor,

            key,

            startTime);

      }

    }



    // Avoid calling back while holding the engine lock, doing so makes it easier for
 callers to

    // deadlock.

    cb.onResourceReady(memoryResource, DataSource.MEMORY_CACHE);

    return null;

  }
```

复制代码

## 1、内存缓存

再来看下 Glide 的内存缓存机制

前文说了，Glide 的内存缓存分为 ActiveResources 和 MemoryCache 两级。首先，Glide 会先根据 key 从 ActiveResources 中取值，如果取得到的话则调用 `acquire()` 方法将该资源的引用数加一。从 ActiveResources 取不到值的话则再根据 key 从 MemoryCache 取值，如果取得到的话则调用 `acquire()` 方法将该资源的引用数加一，并同时将该资源从 MemoryCache 中移除并存入 ActiveResources 中，取不到值的话则最终返回 null

```
private final ActiveResources activeResources;



private final MemoryCache cache;



//尝试从内存中加载图片资源

@Nullable

private EngineResource<?> loadFromMemory(

    EngineKey key, boolean isMemoryCacheable, long startTime) {

  if (!isMemoryCacheable) { //如果配置了不允许使用内存缓存则直接返回

    return null;

  }



  //从 ActiveResources 加载

  EngineResource<?> active = loadFromActiveResources(key);

  if (active != null) {

    if (VERBOSE_IS_LOGGABLE) {

      logWithTimeAndKey("Loaded resource from active resources", startTime, key);

    }
```

```
    return active;
  }

  //从 MemoryCache 加载
  EngineResource<?> cached = loadFromCache(key);
  if (cached != null) {
    if (VERBOSE_IS_LOGGABLE) {
      logWithTimeAndKey("Loaded resource from cache", startTime, key);
    }
    return cached;
  }

  return null;
}

@Nullable
private EngineResource<?> loadFromActiveResources(Key key) {
  EngineResource<?> active = activeResources.get(key);
  if (active != null) {
    active.acquire();
  }

  return active;
```

```
  }

  private EngineResource<?> loadFromCache(Key key) {
    EngineResource<?> cached = getEngineResourceFromCache(key);
    if (cached != null) {
      cached.acquire();
      activeResources.activate(key, cached);
    }
    return cached;
  }

  private EngineResource<?> getEngineResourceFromCache(Key key) {
    Resource<?> cached = cache.remove(key);

    final EngineResource<?> result;
    if (cached == null) {
      result = null;
    } else if (cached instanceof EngineResource) {
      // Save an object allocation if we've cached an EngineResource (the typical cas
e).
      result = (EngineResource<?>) cached;
    } else {
      result =
```

```
        new EngineResource<>(

            cached, /*isMemoryCacheable=*/ true, /*isRecyclable=*/ true, key, /*lis
tener=*/ this);

    }

    return result;

  }



复制代码
```

ActiveResources 是通过弱引用的方式来保存当前所有正在被使用的图片资源。我们知道，当一个对象只具有弱引用而不再被强引用，那么当发生 GC 时，弱引用中持有的引用就会被直接置空，同时弱引用对象本身就会被存入关联的 ReferenceQueue 中

当有一张新图片加载成功且被使用了，且当前配置项允许内存缓存，那么该图片资源就会通过 `activate` 方法保存到 activeEngineResources 中。当一张图片资源的引用计数 acquired 变为 0 时，说明该资源当前已经不再被外部使用了，此时就会通过 `deactivate` 方法将其从 activeEngineResources 中移除，消除对资源的引用，如果当前允许内存缓存的话则还会将该资源存入到 MemoryCache 中

```
final class ActiveResources {



  final Map<Key, ResourceWeakReference> activeEngineResources = new HashMap<>();



  private final ReferenceQueue<EngineResource<?>> resourceReferenceQueue = new Refer
enceQueue<>();



  synchronized void activate(Key key, EngineResource<?> resource) {

    ResourceWeakReference toPut =
```

```
      new ResourceWeakReference(
          key, resource, resourceReferenceQueue, isActiveResourceRetentionAllowed);


  ResourceWeakReference removed = activeEngineResources.put(key, toPut);
  if (removed != null) {
    removed.reset();
  }
}


synchronized void deactivate(Key key) {
  ResourceWeakReference removed = activeEngineResources.remove(key);
  if (removed != null) {
    removed.reset();
  }
}


@Synthetic
void cleanupActiveReference(@NonNull ResourceWeakReference ref) {
  synchronized (this) {
    activeEngineResources.remove(ref.key);


    if (!ref.isCacheable || ref.resource == null) {
      return;
```

```
      }
    }

    EngineResource<?> newResource =
        new EngineResource<>(
            ref.resource, /*isMemoryCacheable=*/ true, /*isRecyclable=*/ false, ref.key, listener);
    listener.onResourceReleased(ref.key, newResource);
  }

}


 //对应 Engine 类
 @Override
 public void onResourceReleased(Key cacheKey, EngineResource<?> resource) {
    //从 activeResources 中移除该图片资源
    activeResources.deactivate(cacheKey);
    if (resource.isMemoryCacheable()) {
      //如果允许内存缓存的话则再将图片资源存到 MemoryCache 中
      cache.put(cacheKey, resource);
    } else {
      resourceRecycler.recycle(resource, /*forceNextFrame=*/ false);
```

```
    }

  }
```

复制代码

MemoryCache 的默认实现则对应着 LruResourceCache 类。从名字也可以看出来，MemoryCache 使用的是 Lru 算法，其会根据外部传入的最大内存缓存大小来进行图片缓存，本身逻辑比较简单，不过多赘述

LruResourceCache 主要是包含了一个 ResourceRemovedListener 对象，用于当从内存缓存中移除了某个图片对象时回调通知 Engine，由 Engine 来回收该图片资源

```
public class LruResourceCache extends LruCache<Key, Resource<?>> implements MemoryCa
che {



  @Override

  public void setResourceRemovedListener(@NonNull ResourceRemovedListener listener)
{

    this.listener = listener;

  }



  @Override

  protected void onItemEvicted(@NonNull Key key, @Nullable Resource<?> item) {

    if (listener != null && item != null) {

      listener.onResourceRemoved(item);

    }

  }
```

```
}
```

复制代码

好了，那就再来总结下 ActiveResources 和 MemoryCache 的逻辑和关系

1. ActiveResources 通过弱引用来保存当前处于使用状态的图片资源，当一张图片被加载成功且还处于使用状态时 ActiveResources 就会一直持有着对其的引用，当图片不再被使用时就会从 ActiveResources 中移除并存入到 MemoryCache 中
2. MemoryCache 使用了 Lrc 算法在内存中缓存图片资源，仅用于缓存当前并非处于使用状态的图片资源。当缓存在 MemoryCache 中的图片被外部重用时，该图片就会从 MemoryCache 中移除并再次存入 ActiveResources 中
3. ActiveResources 中保存的图片是当前处于强引用状态的资源，正常来说即使系统当前可用内存不足，系统即使抛出 OOM 也不会回收强引用，所以 Glide 的内存缓存先从 ActiveResources 取值就不会增大当前的已用内存。而硬件内存大小是有限的，MemoryCache 使用 Lrc 算法就是为了尽量节省内存且尽量让最大概率还会被重用的图片可以被保留下来
4. Glide 将内存缓存分为 ActiveResources 和 MemoryCache 两级而不是全都放到 MemoryCache 中，就避免了误将当前正处于活跃状态的图片资源给移除队列。且 ActiveResources 内部也一直在循环判断保存的图片资源是否已经不再被外部使用了，从而可以及时更新 MemoryCache，提高了 MemoryCache 的利用率和准确度

## 2、磁盘缓存

Glide 的磁盘缓存逻辑要从 Engine 类的 `waitForExistingOrStartNewJob` 方法开始看起。当判断到当前内存缓存中没有目标图片时，就会启动 EngineJob 和 DecodeJob 进行磁盘缓存加载、本地文件加载或者联网加载

```
private <R> LoadStatus waitForExistingOrStartNewJob(
      GlideContext glideContext,
      Object model,
      Key signature,
      int width,
      int height,
      Class<?> resourceClass,
      Class<R> transcodeClass,
      Priority priority,
      DiskCacheStrategy diskCacheStrategy,
      Map<Class<?>, Transformation<?>> transformations,
      boolean isTransformationRequired,
      boolean isScaleOnlyOrNoTransform,
      Options options,
      boolean isMemoryCacheable,
      boolean useUnlimitedSourceExecutorPool,
      boolean useAnimationPool,
      boolean onlyRetrieveFromCache,
      ResourceCallback cb,
      Executor callbackExecutor,
      EngineKey key,
      long startTime) {
```

```
EngineJob<?> current = jobs.get(key, onlyRetrieveFromCache);
if (current != null) {
  //如果已经启动了同个请求任务，那么就向其添加一个回调即可
  current.addCallback(cb, callbackExecutor);
  if (VERBOSE_IS_LOGGABLE) {
    logWithTimeAndKey("Added to existing load", startTime, key);
  }
  return new LoadStatus(cb, current);
}

EngineJob<R> engineJob =
    engineJobFactory.build(
        key,
        isMemoryCacheable,
        useUnlimitedSourceExecutorPool,
        useAnimationPool,
        onlyRetrieveFromCache);

DecodeJob<R> decodeJob =
    decodeJobFactory.build(
        glideContext,
        model,
        key,
```

```
        signature,
        width,
        height,
        resourceClass,
        transcodeClass,
        priority,
        diskCacheStrategy,
        transformations,
        isTransformationRequired,
        isScaleOnlyOrNoTransform,
        onlyRetrieveFromCache,
        options,
        engineJob);


jobs.put(key, engineJob);


engineJob.addCallback(cb, callbackExecutor);
//启动 decodeJob
engineJob.start(decodeJob);


if (VERBOSE_IS_LOGGABLE) {
  logWithTimeAndKey("Started new load", startTime, key);
}
```

```
    return new LoadStatus(cb, engineJob);

  }
```

复制代码

这里主要 DecodeJob 类

前文有讲到，Glide 缓存的图片类型可以分为两类，一类是原始图片，一类是将原始图片进行各种压缩裁剪变换等各种转换操作后得到的图片。那么如果我们本次请求配置了允许复用磁盘缓存，DecodeJob 会根据我们的请求配置来选择相应的 DataFetcherGenerator 来进行处理，最终图片的来源类型就有三种可能：

1. 复用转换过的图片资源。对应 ResourceCacheGenerator，当缓存未命中时就从 DataCacheGenerator 取值
2. 复用原始的图片资源。对应 DataCacheGenerator，当缓存未命中时就从 SourceGenerator 取值
3. 本地没有符合条件的已缓存资源，需要全新加载（例如，联网请求）。对应 SourceGenerator

```
private DataFetcherGenerator getNextGenerator() {

  switch (stage) {

    case RESOURCE_CACHE:

      return new ResourceCacheGenerator(decodeHelper, this);

    case DATA_CACHE:

      return new DataCacheGenerator(decodeHelper, this);

    case SOURCE:

      return new SourceGenerator(decodeHelper, this);

    case FINISHED:
```

```
      return null;
    default:
      throw new IllegalStateException("Unrecognized stage: " + stage);
  }
}
```

复制代码

例如，DataCacheGenerator 的主要逻辑就是 `startNext()` 方法，该方法会从 DiskCache 中取值，拿到缓存文件 cacheFile 以及相应的处理器 modelLoaders，modelLoaders 就包含了所有可以实现本次转换操作（例如，File 转 Drawable、File 转 Bitmap 等）的实现器，如果最终判定到存在缓存文件及相应的转换器，那么方法就会返回 true

当 DataCacheGenerator 加载目标数据成功后，就会回调 DecodeJob 的 `onDataFetcherReady` 方法，最终将目标数据存到 ActiveResources 中并通知所有 Target

```
@Override
public boolean startNext() {
  while (modelLoaders == null || !hasNextModelLoader()) {
    sourceIdIndex++;
    if (sourceIdIndex >= cacheKeys.size()) {
      return false;
    }

    Key sourceId = cacheKeys.get(sourceIdIndex);
```

```
      // PMD.AvoidInstantiatingObjectsInLoops The loop iterates a limited number of t
imes
      // and the actions it performs are much more expensive than a single allocatio
n.
      @SuppressWarnings("PMD.AvoidInstantiatingObjectsInLoops")
      Key originalKey = new DataCacheKey(sourceId, helper.getSignature());
      //从磁盘缓存中取值
      cacheFile = helper.getDiskCache().get(originalKey);
      if (cacheFile != null) {
        this.sourceKey = sourceId;
        //拿到所有的数据类型转换器
        modelLoaders = helper.getModelLoaders(cacheFile);
        modelLoaderIndex = 0;
      }
    }


    loadData = null;
    boolean started = false;
    while (!started && hasNextModelLoader()) {
      ModelLoader<File, ?> modelLoader = modelLoaders.get(modelLoaderIndex++);
      loadData =
          modelLoader.buildLoadData(
              cacheFile, helper.getWidth(), helper.getHeight(), helper.getOptions());
      if (loadData != null && helper.hasLoadPath(loadData.fetcher.getDataClass())) {
```

```
        started = true;
        loadData.fetcher.loadData(helper.getPriority(), this);
      }
    }
    return started;
  }
```

复制代码

DataCacheGenerator 代表的是从本地磁盘缓存中取到目标图片的情况，而将图片资源写入本地磁盘的逻辑还要看 SourceGenerator

SourceGenerator 负责全新加载一张图片资源，在加载成功后就会调用到 `onDataReadyInternal` 方法。如果本次请求不允许进行磁盘缓存，就会直接回调 DecodeJob 的 `onDataFetcherReady` 方法完成整个流程，这个过程就和 DataCacheGenerator 一致。而如果允许进行磁盘缓存，那么就会调用到 `reschedule()`方法重新触发 `startNext()` 方法，在 `cacheData` 方法中完成磁盘文件的写入，在写入成功后就会构造一个 DataCacheGenerator，由 DataCacheGenerator 再来从磁盘中取值

```
  void onDataReadyInternal(LoadData<?> loadData, Object data) {
    DiskCacheStrategy diskCacheStrategy = helper.getDiskCacheStrategy();
    if (data != null && diskCacheStrategy.isDataCacheable(loadData.fetcher.getDataSo
urce())) {
      //允许进行磁盘缓存，先将 data 缓存到 dataToCache 变量
      dataToCache = data;
      // We might be being called back on someone else's thread. Before doing anythin
g, we should
      // reschedule to get back onto Glide's thread.
```

```
    cb.reschedule();
  } else {
    cb.onDataFetcherReady(
        loadData.sourceKey,
        data,
        loadData.fetcher,
        loadData.fetcher.getDataSource(),
        originalKey);
  }
}

@Override
public boolean startNext() {
  if (dataToCache != null) {
    Object data = dataToCache;
    dataToCache = null;
    cacheData(data);
  }

  if (sourceCacheGenerator != null && sourceCacheGenerator.startNext()) {
    return true;
  }
  ...
```

```
    return started;
  }




  private void cacheData(Object dataToCache) {
    long startTime = LogTime.getLogTime();
    try {
      Encoder<Object> encoder = helper.getSourceEncoder(dataToCache);
      DataCacheWriter<Object> writer =
          new DataCacheWriter<>(encoder, dataToCache, helper.getOptions());
      originalKey = new DataCacheKey(loadData.sourceKey, helper.getSignature());
      //写入磁盘缓存
      helper.getDiskCache().put(originalKey, writer);
      if (Log.isLoggable(TAG, Log.VERBOSE)) {
        Log.v(
            TAG,
            "Finished encoding source to cache"
                + ", key: "
                + originalKey
                + ", data: "
                + dataToCache
                + ", encoder: "
                + encoder
```

```
                + ", duration: "
                + LogTime.getElapsedMillis(startTime));
      }
    } finally {
      loadData.fetcher.cleanup();
    }

    sourceCacheGenerator =
        new DataCacheGenerator(Collections.singletonList(loadData.sourceKey), helper,
 this);
  }
```

复制代码

Glide 的磁盘缓存算法具体对应的是 DiskLruCache 类，这是 Glide 根据 JakeWharton 的 DiskLruCache 开源库修改而来的，这里不过多赘述

## 六、如何分辨不同的加载类型

`Glide.with(Context).load(Any)`的 `load` 方法是一个多重载形式的方法，支持 **Integer、String、Uri、File** 等多种入参类型。那么，Glide 是如何分辨我们不同的入参请求的呢？以及如何对不同的请求类型进行处理呢？

Glide 类中包含一个 `registry` 变量，相当于一个注册器，存储了对于特定的入参类型，其对应的处理逻辑，以及该入参类型希望得到的结果值类型

```
    registry
        .append(Uri.class, InputStream.class, new UriLoader.StreamFactory(contentRes
olver))
```

```
        .append(
            Uri.class,
            ParcelFileDescriptor.class,
            new UriLoader.FileDescriptorFactory(contentResolver))
        .append(
            Uri.class,
            AssetFileDescriptor.class,
            new UriLoader.AssetFileDescriptorFactory(contentResolver))
        .append(Uri.class, InputStream.class, new UrlUriLoader.StreamFactory())
        .append(URL.class, InputStream.class, new UrlLoader.StreamFactory())
        .append(Uri.class, File.class, new MediaStoreFileLoader.Factory(context))
        .append(GlideUrl.class, InputStream.class, new HttpGlideUrlLoader.Factory())
        .append(byte[].class, ByteBuffer.class, new ByteArrayLoader.ByteBufferFactor
y())
        .append(byte[].class, InputStream.class, new ByteArrayLoader.StreamFactory
())
        .append(Uri.class, Uri.class, UnitModelLoader.Factory.<Uri>getInstance())
        .append(Drawable.class, Drawable.class, UnitModelLoader.Factory.<Drawable>ge
tInstance())
        .append(Drawable.class, Drawable.class, new UnitDrawableDecoder())
        /* Transcoders */
        .register(Bitmap.class, BitmapDrawable.class, new BitmapDrawableTranscoder(r
esources))
        .register(Bitmap.class, byte[].class, bitmapBytesTranscoder)
        .register(
```

```
            Drawable.class,
            byte[].class,
            new DrawableBytesTranscoder(
                bitmapPool, bitmapBytesTranscoder, gifDrawableBytesTranscoder))
        .register(GifDrawable.class, byte[].class, gifDrawableBytesTranscoder);
```

复制代码

例如，我们最常见的一种请求方式就是通过图片的 Url 来从网络获取图片，这就对应着以下配置：

```
    append(GlideUrl.class, InputStream.class, new HttpGlideUrlLoader.Factory())
```

复制代码

当中，GlideUrl 就对应着我们传入的 ImageUrl，InputStream 即希望根据该 Url 从网络获取到相应的资源输入流，HttpGlideUrlLoader 就用来实现将 ImageUrl 转换为 InputStream 的过程

HttpGlideUrlLoader 会将 ImageUrl 传给 HttpUrlFetcher，由其来进行具体的网络请求

```
public class HttpGlideUrlLoader implements ModelLoader<GlideUrl, InputStream> {


    @Override
    public LoadData<InputStream> buildLoadData(
        @NonNull GlideUrl model, int width, int height, @NonNull Options options) {
        // GlideUrls memoize parsed URLs so caching them saves a few object instantia
tions and time
```

```
        // spent parsing urls.
        GlideUrl url = model;
        if (modelCache != null) {
            url = modelCache.get(model, 0, 0);
                if (url == null) {
            modelCache.put(model, 0, 0, model);
            url = model;
        }
        }
        int timeout = options.get(TIMEOUT);
        return new LoadData<>(url, new HttpUrlFetcher(url, timeout));
      }


}
```

HttpUrlFetcher 会在 `loadDataWithRedirects` 方法中通过 HttpURLConnection 来请求图片，最终通过 DataCallback 来将得到的图片输入流 InputStream 对象透传出去。此外，`loadDataWithRedirects` 方法会通过循环调用自己的方式来处理重定向的情况，不允许重复重定向到同个 Url，且最多重定向五次，否则就会直接走失败流程

```
public class HttpUrlFetcher implements DataFetcher<InputStream> {



    private static final int MAXIMUM_REDIRECTS = 5;
```

```
  @Override

  public void loadData(

      @NonNull Priority priority, @NonNull DataCallback<? super InputStream> callback)
 {

    long startTime = LogTime.getLogTime();

    try {

      InputStream result = loadDataWithRedirects(glideUrl.toURL(), 0, null, glideUrl.
getHeaders());

      callback.onDataReady(result);

    } catch (IOException e) {

      if (Log.isLoggable(TAG, Log.DEBUG)) {

        Log.d(TAG, "Failed to load data for url", e);

      }

      callback.onLoadFailed(e);

    } finally {

      if (Log.isLoggable(TAG, Log.VERBOSE)) {

        Log.v(TAG, "Finished http url fetcher fetch in " + LogTime.getElapsedMillis(s
tartTime));

      }

    }

  }



  private InputStream loadDataWithRedirects(

      URL url, int redirects, URL lastUrl, Map<String, String> headers) throws IOExce
ption {
```

```
    if (redirects >= MAXIMUM_REDIRECTS) {

       //重定向总次数达到五次，走失败流程

      throw new HttpException("Too many (> " + MAXIMUM_REDIRECTS + ") redirects!");

    } else {

      // Comparing the URLs using .equals performs additional network I/O and is gene
rally broken.

      // See http://michaelscharf.blogspot.com/2006/11/javaneturlequals-and-hashcode
-make.html.

      try {

        if (lastUrl != null && url.toURI().equals(lastUrl.toURI())) {

          //循环重定向到同个 Url，走失败流程

          throw new HttpException("In re-direct loop");

        }

      } catch (URISyntaxException e) {

        // Do nothing, this is best effort.

      }

    }



    urlConnection = connectionFactory.build(url);

    ...

    stream = urlConnection.getInputStream();

    if (isCancelled) {

      return null;

    }
```

```
    final int statusCode = urlConnection.getResponseCode();

    if (isHttpOk(statusCode)) {

      return getStreamForSuccessfulRequest(urlConnection);

    } else if (isHttpRedirect(statusCode)) {

      String redirectUrlString = urlConnection.getHeaderField("Location");

      if (TextUtils.isEmpty(redirectUrlString)) {

        throw new HttpException("Received empty or null redirect url");

      }

      URL redirectUrl = new URL(url, redirectUrlString);

      // Closing the stream specifically is required to avoid leaking ResponseBodys i
n addition

      // to disconnecting the url connection below. See #2352.

      cleanup();

      return loadDataWithRedirects(redirectUrl, redirects + 1, url, headers);

    } else if (statusCode == INVALID_STATUS_CODE) {

      throw new HttpException(statusCode);

    } else {

      throw new HttpException(urlConnection.getResponseMessage(), statusCode);

    }

  }



}
```

# 七、一共包含几个线程池

先说结论，如果我没看遗漏的话，Glide 是一共包含七个线程池。**此处我所指的线程池的概念不单单指 ThreadPoolExecutor 类，而是指 java.util.concurrent.Executor 接口的任意实现类**

其中，前四个线程池可以从 EngineJob 类的构造参数得到答案

```
class EngineJob<R> implements DecodeJob.Callback<R>, Poolable {



  EngineJob(

      GlideExecutor diskCacheExecutor,

      GlideExecutor sourceExecutor,

      GlideExecutor sourceUnlimitedExecutor,

      GlideExecutor animationExecutor,

      EngineJobListener engineJobListener,

      ResourceListener resourceListener,

      Pools.Pool<EngineJob<?>> pool) {

    this(

        diskCacheExecutor,

        sourceExecutor,

        sourceUnlimitedExecutor,

        animationExecutor,

        engineJobListener,

        resourceListener,

        pool,
```

```
        DEFAULT_FACTORY);
  }


}
```

其用途分别是：

1. diskCacheExecutor。用于加载磁盘缓存
2. sourceExecutor。用于执行非加载本地磁盘缓存的操作，例如，根据指定的 URI 或者 ImageUrl 去加载图片
3. sourceUnlimitedExecutor。同 sourceExecutor
4. animationExecutor。按官方的注释解释就是用于加载 Gif

这四个线程池的创建逻辑可以看 GlideExecutor 类，这四个线程池的区别是：

1. diskCacheExecutor。核心线程数和最大线程数均为 1，线程超时时间为 0 秒。因为 diskCacheExecutor 执行的是磁盘文件读写，核心线程数和最大线程数均为 1 就使得当线程池被启动后始终只有一个线程处于活跃状态，保证了文件读写时的有序性，避免了加锁操作
2. sourceExecutor。核心线程数和最大线程数根据设备的 CPU 个数来决定，至少是 4 个线程，线程超时时间为 0 秒。线程数量的设置就限制了 Glide 最多发起四个联网加载图片的请求
3. sourceUnlimitedExecutor。核心线程数为 0，最大线程数为 Integer.MAX_VALUE，超时时间为 10 秒，当线程闲置时就会被马上回收。sourceUnlimitedExecutor 的目的是为了应对需要同时处理大量加载图片请求的需求，允许近乎无限制地新建线程来处理每个请求，在及时性上相对 sourceExecutor 可能会有所提升，但也可能反而会因为多线程竞争而降低效率，且也容易发生 OOM

4. animationExecutor。如果设备的 CPU 个数大于 4，则核心线程数和最大线程数设为 2，否则设为 1；线程超时时间为 0 秒

这四个线程池都用于 EngineJob 类。diskCacheExecutor 只用于磁盘缓存，只要本次请求允许使用磁盘缓存， diskCacheExecutor 就会被使用到。而其它三个线程池在我看来都是用于加载本地文件或者联网请求图片，如果 useUnlimitedSourceGeneratorPool 为 true，就使用 sourceUnlimitedExecutor，否则如果 useAnimationPool 为 true，就使用 animationExecutor，否则就使用 sourceExecutor

**useUnlimitedSourceGeneratorPool 的意义还好理解，就是为了控制同时并发请求的最大线程数，但区分 useAnimationPool 的意义我就不太理解了，懂的同学麻烦解答下**

```
  public synchronized void start(DecodeJob<R> decodeJob) {

    this.decodeJob = decodeJob;

    GlideExecutor executor =

        decodeJob.willDecodeFromCache() ? diskCacheExecutor : getActiveSourceExecutor
();

    executor.execute(decodeJob);

  }



  private GlideExecutor getActiveSourceExecutor() {

    return useUnlimitedSourceGeneratorPool

        ? sourceUnlimitedExecutor

        : (useAnimationPool ? animationExecutor : sourceExecutor);

  }
```

第五个线程池就位于 ActiveResources 类中。该线程池就用于不断从 ReferenceQueue 中取值判断，将当前已经不再被外部使用的图片资源缓存到 MemoryCache 中

```
ActiveResources(boolean isActiveResourceRetentionAllowed) {

   this(

       isActiveResourceRetentionAllowed,

       java.util.concurrent.Executors.newSingleThreadExecutor(

           new ThreadFactory() {

             @Override

             public Thread newThread(@NonNull final Runnable r) {

               return new Thread(

                   new Runnable() {

                     @Override

                     public void run() {

                       Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);

                       r.run();

                     }

                   },

                   "glide-active-resources");

             }

           }));

 }
```

其余的两个线程池则在 Executors 类中

1. MAIN_THREAD_EXECUTOR。用于当图片加载完成后，通过 Handler 切换到主线程来更新 UI
2. DIRECT_EXECUTOR。可以看做是一个空实现，会在原来的线程上执行 Runnable，当我们想直接取得图片资源而非更新 UI 时，例如 `Glide.with(this).load(url).submit()`，此时就会使用到

```
public final class Executors {

  private Executors() {

    // Utility class.

  }



  private static final Executor MAIN_THREAD_EXECUTOR =

      new Executor() {

        private final Handler handler = new Handler(Looper.getMainLooper());



        @Override

        public void execute(@NonNull Runnable command) {

          handler.post(command);

        }

      };

  private static final Executor DIRECT_EXECUTOR =

      new Executor() {

        @Override
```

```
      public void execute(@NonNull Runnable command) {
        command.run();
      }
    };

/** Posts executions to the main thread. */
public static Executor mainThreadExecutor() {
  return MAIN_THREAD_EXECUTOR;
}

/** Immediately calls {@link Runnable#run()} on the current thread. */
public static Executor directExecutor() {
  return DIRECT_EXECUTOR;
}

@VisibleForTesting
public static void shutdownAndAwaitTermination(ExecutorService pool) {
  long shutdownSeconds = 5;
  pool.shutdownNow();
  try {
    if (!pool.awaitTermination(shutdownSeconds, TimeUnit.SECONDS)) {
      pool.shutdownNow();
      if (!pool.awaitTermination(shutdownSeconds, TimeUnit.SECONDS)) {
```

```
          throw new RuntimeException("Failed to shutdown");

        }

      }

    } catch (InterruptedException ie) {

      pool.shutdownNow();

      Thread.currentThread().interrupt();

      throw new RuntimeException(ie);

    }

  }

}
```

## 八、如何自定义网络请求库

默认情况下，Glide 是通过 HttpURLConnection 来联网加载图片的，相对于我们常用的 OkHttp 来说比较原始低效。而 Glide 也提供了 Registry 类，允许外部来自定义实现特定的请求逻辑

例如，如果你想要通过 OkHttp 来请求图片，那么可以依赖 Glide 官方提供的支持库：

```
dependencies {

    implementation "com.github.bumptech.glide:okhttp3-integration:4.11.0"

}
```

只要集成了 `okhttp3-integration`，那么 Glide 就会自动将网络类型的请求交由其内部的 OkHttp 来处理，因为其内部包含了一个声明了 **@GlideModule** 注解的

OkHttpLibraryGlideModule 类，可以在运行时被 Glide 解析到，之后就会将 GlideUrl 类型的加载请求交由 OkHttpUrlLoader 来进行处理

```
@GlideModulepublic final class OkHttpLibraryGlideModule extends LibraryGlideModule {

  @Override

  public void registerComponents(

      @NonNull Context context, @NonNull Glide glide, @NonNull Registry registry) {

    registry.replace(GlideUrl.class, InputStream.class, new OkHttpUrlLoader.Factory
());

  }

}
```

我们也可以将 `okhttp3-integration` 中的代码复制出来，在自定义的 AppGlideModule 类中传入自己实现的 OkHttpUrlLoader

```
@GlideModuleclass MyAppGlideModule : AppGlideModule() {



    override fun isManifestParsingEnabled(): Boolean {

        return false

    }



    override fun registerComponents(context: Context, glide: Glide, registry: Regist
ry) {

        val okHttClient = OkHttpClient.Builder()

            .connectTimeout(10, TimeUnit.SECONDS)

            .writeTimeout(10, TimeUnit.SECONDS)
```

```
            .readTimeout(15, TimeUnit.SECONDS)
            .eventListener(object : EventListener() {
                override fun callStart(call: okhttp3.Call) {
                    Log.e("TAG", "callStart:  " + call.request().url().toString())
                }
            }).build()
        registry.replace(
            GlideUrl::class.java, InputStream::class.java,
            OkHttpUrlLoader.Factory(okHttClient)
        )
    }

}
```

## 九、内存清理机制

Glide 的内存缓存机制是为了尽量复用图片资源频繁的内存读写，memoryCache、bitmapPool 和 arrayPool 的存在都是为了这个目的，但另一方面内存缓存也造成了有一部分内存空间一直被占用着，可能会造成系统的可用内存空间不足。当我们的应用退到后台时，如果之后系统的可用内存空间不足，那么系统就会按照优先级高低来清理掉一些后台进程，以便为前台进程腾出内存空间，为了提高应用在后台时的优先级避免被系统杀死，我们就需要主动降低我们的内存占用

所幸的是 Glide 也考虑到了这种情况，提供了缓存内存的自动清理机制。Glide 类的 `initializeGlide` 方法就默认向 Application 注册了一个 ComponentCallbacks，用于接收系统下发的内存状态变化的事件通知

```
@GuardedBy("Glide.class")
@SuppressWarnings("deprecation")
private static void initializeGlide(
    @NonNull Context context,
    @NonNull GlideBuilder builder,
    @Nullable GeneratedAppGlideModule annotationGeneratedModule) {
  Context applicationContext = context.getApplicationContext();
  ...
  applicationContext.registerComponentCallbacks(glide);
  Glide.glide = glide;
}
```

对应的 ComponentCallbacks 实现类即 Glide 类本身，其相关的方法实现对应以下两个

```
@Override
public void onTrimMemory(int level) {
  trimMemory(level);
}


@Override
public void onLowMemory() {
  clearMemory();
```

```
  }
```

这两个方法会自动触发对 memoryCache、bitmapPool 和 arrayPool 的清理工作

```
  public void trimMemory(int level) {
    // Engine asserts this anyway when removing resources, fail faster and consistent
ly
    Util.assertMainThread();
    // Request managers need to be trimmed before the caches and pools, in order for
the latter to
    // have the most benefit.
    for (RequestManager manager : managers) {
      manager.onTrimMemory(level);
    }
    // memory cache needs to be trimmed before bitmap pool to trim re-pooled Bitmaps
too. See #687.
    memoryCache.trimMemory(level);
    bitmapPool.trimMemory(level);
    arrayPool.trimMemory(level);
  }


  public void clearMemory() {
    // Engine asserts this anyway when removing resources, fail faster and consistent
ly
```

```
    Util.assertMainThread();

    // memory cache needs to be cleared before bitmap pool to clear re-pooled Bitmaps
too. See #687.

    memoryCache.clearMemory();

    bitmapPool.clearMemory();

    arrayPool.clearMemory();

  }
```

# 三方库源码笔记（10）-Glide 你可能不知道的知识点

## 一、利用 AppGlideModule 实现默认配置

在大多数情况下 Glide 的默认配置就已经能够满足我们的需求了，像缓存池大小，磁盘缓存策略等都不需要我们主动去设置，但 Glide 也提供了 AppGlideModule 让开发者可以去实现自定义配置。对于一个 App 来说，在加载图片的时候一般都是使用同一张 placeholder，如果每次加载图片时都需要来手动设置一遍的话就显得很多余了，此时就可以通过 AppGlideModule 来设置默认的 placeholder

首先需要继承于 AppGlideModule，在 `applyOptions` 方法中设置配置参数，然后为实现类添加 @GlideModule 注解，这样在编译阶段 Glide 就可以通过 APT 解析到我们的这一个实现类，然后将我们的配置参数设置为默认值

```
/**

 * 作者：leavesC

 * 时间：2020/11/5 23:16

 * 描述：

 * GitHub：https://github.com/leavesC

 */@GlideModuleclass MyAppGlideModule : AppGlideModule() {
```

```kotlin
//用于控制是否需要从 Manifest 文件中解析配置文件
override fun isManifestParsingEnabled(): Boolean {
    return false
}

override fun applyOptions(context: Context, builder: GlideBuilder) {
    builder.setDiskCache(
        //配置磁盘缓存目录和最大缓存
        DiskLruCacheFactory(
            (context.externalCacheDir ?: context.cacheDir).absolutePath,
            "imageCache",
            1024 * 1024 * 50
        )
    )
    builder.setDefaultRequestOptions {
        return@setDefaultRequestOptions RequestOptions()
            .placeholder(android.R.drawable.ic_menu_upload_you_tube)
            .error(android.R.drawable.ic_menu_call)
            .diskCacheStrategy(DiskCacheStrategy.AUTOMATIC)
            .format(DecodeFormat.DEFAULT)
            .encodeQuality(90)
    }
```

```
    }


    override fun registerComponents(context: Context, glide: Glide, registry: Regist
ry) {


    }


}
```

复制代码

在编译后，我们的工程目录中就会自动生成 GeneratedAppGlideModuleImpl 这个类，该类就包含了 MyAppGlideModule

```
@SuppressWarnings("deprecation")final class GeneratedAppGlideModuleImpl extends Gene
ratedAppGlideModule {

  private final MyAppGlideModule appGlideModule;



  public GeneratedAppGlideModuleImpl(Context context) {

    appGlideModule = new MyAppGlideModule();

    if (Log.isLoggable("Glide", Log.DEBUG)) {

      Log.d("Glide", "Discovered AppGlideModule from annotation: github.leavesc.glid
e.MyAppGlideModule");

    }

  }



  @Override
```

```
  public void applyOptions(@NonNull Context context, @NonNull GlideBuilder builder)
{

    appGlideModule.applyOptions(context, builder);

  }



  @Override

  public void registerComponents(@NonNull Context context, @NonNull Glide glide,

      @NonNull Registry registry) {

    appGlideModule.registerComponents(context, glide, registry);

  }



  @Override

  public boolean isManifestParsingEnabled() {

    return appGlideModule.isManifestParsingEnabled();

  }



  @Override

  @NonNull

  public Set<Class<?>> getExcludedModuleClasses() {

    return Collections.emptySet();

  }



  @Override
```

```
  @NonNull

  GeneratedRequestManagerFactory getRequestManagerFactory() {

    return new GeneratedRequestManagerFactory();

  }

}
```

复制代码

在运行阶段，Glide 就会通过反射生成一个 GeneratedAppGlideModuleImpl 对象，然后根据我们的默认配置项来初始化 Glide 实例

```
 @Nullable

  @SuppressWarnings({"unchecked", "TryWithIdenticalCatches", "PMD.UnusedFormalParame
ter"})

  private static GeneratedAppGlideModule getAnnotationGeneratedGlideModules(Context
context) {

    GeneratedAppGlideModule result = null;

    try {

      //通过反射来生成一个 GeneratedAppGlideModuleImpl 对象

      Class<GeneratedAppGlideModule> clazz =

          (Class<GeneratedAppGlideModule>)

              Class.forName("com.bumptech.glide.GeneratedAppGlideModuleImpl");

      result =

          clazz.getDeclaredConstructor(Context.class).newInstance(context.getApplica
tionContext());

    } catch (ClassNotFoundException e) {

      if (Log.isLoggable(TAG, Log.WARN)) {
```

```
       Log.w(
           TAG,
           "Failed to find GeneratedAppGlideModule. You should include an"
               + " annotationProcessor compile dependency on com.github.bumptech.gli
de:compiler"
               + " in your application and a @GlideModule annotated AppGlideModule im
plementation"
               + " or LibraryGlideModules will be silently ignored");
      }
      // These exceptions can't be squashed across all versions of Android.
    } catch (InstantiationException e) {
      throwIncorrectGlideModule(e);
    } catch (IllegalAccessException e) {
      throwIncorrectGlideModule(e);
    } catch (NoSuchMethodException e) {
      throwIncorrectGlideModule(e);
    } catch (InvocationTargetException e) {
      throwIncorrectGlideModule(e);
    }
    return result;
  }



 private static void initializeGlide(
```

```
      @NonNull Context context,
      @NonNull GlideBuilder builder,
      @Nullable GeneratedAppGlideModule annotationGeneratedModule) {
    Context applicationContext = context.getApplicationContext();
    ...
    if (annotationGeneratedModule != null) {
      //调用 MyAppGlideModule 的 applyOptions 方法，对 GlideBuilder 进行设置
      annotationGeneratedModule.applyOptions(applicationContext, builder);
    }
    //根据 GlideBuilder 来生成 Glide 实例
    Glide glide = builder.build(applicationContext);
    ...
    if (annotationGeneratedModule != null) {
        //配置自定义组件
        annotationGeneratedModule.registerComponents(applicationContext, glide, glid
e.registry);
    }
    applicationContext.registerComponentCallbacks(glide);
    Glide.glide = glide;
  }
```

复制代码

## 二、自定义网络请求组件

默认情况下，Glide 是通过 HttpURLConnection 来进行联网请求图片的，这个过程就由 HttpUrlFetcher 类来实现。HttpURLConnection 相对于我们常用的 OkHttp 来说比较原始低效，我们可以通过使用 Glide 官方提供的 `okhttp3-integration` 来将网络请求交由 OkHttp 完成

```
dependencies {

    implementation "com.github.bumptech.glide:okhttp3-integration:4.11.0"

}

复制代码
```

如果想方便后续修改的话，我们也可以将 `okhttp3-integration` 内的代码复制出来，通过 Glide 开放的 Registry 来注册一个自定义的 OkHttpStreamFetcher，这里我也提供一份 kotlin 版本的示例代码

首先需要继承于 DataFetcher，在拿到 GlideUrl 后完成网络请求，并将请求结果通过 DataCallback 回调出去

```
/**

 * 作者：leavesC

 * 时间：2020/11/5 23:16

 * 描述：

 * GitHub：https://github.com/leavesC

 */class OkHttpStreamFetcher(private val client: Call.Factory, private val url: Glid
eUrl) :

    DataFetcher<InputStream>, Callback {



    companion object {

        private const val TAG = "OkHttpFetcher"

    }
```

```
    private var stream: InputStream? = null

    private var responseBody: ResponseBody? = null

    private var callback: DataFetcher.DataCallback<in InputStream>? = null

    @Volatile
    private var call: Call? = null

    override fun loadData(priority: Priority, callback: DataFetcher.DataCallback<in
InputStream>) {
        val requestBuilder = Request.Builder().url(url.toStringUrl())
        for ((key, value) in url.headers) {
            requestBuilder.addHeader(key, value)
        }
        val request = requestBuilder.build()
        this.callback = callback
        call = client.newCall(request)
        call?.enqueue(this)
    }

    override fun onFailure(call: Call, e: IOException) {
```

```
        if (Log.isLoggable(TAG, Log.DEBUG)) {
            Log.d(TAG, "OkHttp failed to obtain result", e)
        }
        callback?.onLoadFailed(e)
    }

    override fun onResponse(call: Call, response: Response) {
        if (response.isSuccessful) {
            responseBody = response.body()
            val contentLength = Preconditions.checkNotNull(responseBody).contentLengt
h()
            stream = ContentLengthInputStream.obtain(responseBody!!.byteStream(), con
tentLength)
            callback?.onDataReady(stream)
        } else {
            callback?.onLoadFailed(HttpException(response.message(), response.code
()))
        }
    }

    override fun cleanup() {
        try {
            stream?.close()
        } catch (e: IOException) {
```

```
            // Ignored
        }
        responseBody?.close()
        callback = null
    }

    override fun cancel() {
        call?.cancel()
    }

    override fun getDataClass(): Class<InputStream> {
        return InputStream::class.java
    }

    override fun getDataSource(): DataSource {
        return DataSource.REMOTE
    }

}
```

复制代码

之后还需要继承于 ModelLoader，提供构建 OkHttpUrlLoader 的入口

```
/**
```

```
 * 作者：LeavesC

 * 时间：2020/11/5 23:16

 * 描述：

 * GitHub：https://github.com/LeavesC

 */class OkHttpUrlLoader(private val client: Call.Factory) : ModelLoader<GlideUrl, I
nputStream> {



    override fun buildLoadData(

        model: GlideUrl,

        width: Int,

        height: Int,

        options: Options

    ): LoadData<InputStream> {

        return LoadData(

            model,

            OkHttpStreamFetcher(client, model)

        )

    }



    override fun handles(model: GlideUrl): Boolean {

        return true

    }
```

```
    class Factory(private val client: Call.Factory) : ModelLoaderFactory<GlideUrl, I
nputStream> {



        override fun build(multiFactory: MultiModelLoaderFactory): ModelLoader<GlideU
rl, InputStream> {

            return OkHttpUrlLoader(client)

        }



        override fun teardown() {

            // Do nothing, this instance doesn't own the client.

        }



    }



}
```

复制代码

最后注册 OkHttpUrlLoader ， 之后 GlideUrl 类型的请求都会交由其处理

```
/**

 * 作者：leavesC

 * 时间：2020/11/5 23:16

 * 描述：

 * GitHub：https://github.com/leavesC

 */@GlideModuleclass MyAppGlideModule : AppGlideModule() {
```

```
    override fun isManifestParsingEnabled(): Boolean {
        return false
    }

    override fun applyOptions(context: Context, builder: GlideBuilder) {

    }

    override fun registerComponents(context: Context, glide: Glide, registry: Regist
ry) {
        registry.replace(
            GlideUrl::class.java,
            InputStream::class.java,
            OkHttpUrlLoader.Factory(OkHttpClient())
        )
    }

}
```

复制代码

## 三、实现图片加载进度监听

对于某些高清图片来说，可能一张就是十几 MB 甚至上百 MB 大小了，如果没有进度条的话用户可能就会等得有点难受了，这里我就提供一个基于 **OkHttp 拦截器**实现的监听图片加载进度的方法

首先需要对 OkHttp 原始的 ResponseBody 进行一层包装，在内部根据 **contentLength** 和**已读取到的流字节数**来计算当前进度值，然后向外部提供通过 imageUrl 来注册 ProgressListener 的入口

```
/**

 * 作者：leavesC

 * 时间：2020/11/6 21:58

 * 描述：

 * GitHub：https://github.com/leavesC

 */internal class ProgressResponseBody constructor(

    private val imageUrl: String,

    private val responseBody: ResponseBody?

) : ResponseBody() {



    interface ProgressListener {



        fun update(progress: Int)



    }



    companion object {
```

```
        private val progressMap = mutableMapOf<String, WeakReference<ProgressListene
r>>()



        fun addProgressListener(url: String, listener: ProgressListener) {

            progressMap[url] = WeakReference(listener)

        }



        fun removeProgressListener(url: String) {

            progressMap.remove(url)

        }



        private const val CODE_PROGRESS = 100



        private val mainHandler by lazy {

            object : Handler(Looper.getMainLooper()) {

                override fun handleMessage(msg: Message) {

                    if (msg.what == CODE_PROGRESS) {

                        val pair = msg.obj as Pair<String, Int>

                        val progressListener = progressMap[pair.first]?.get()

                        progressListener?.update(pair.second)

                    }

                }

            }
```

```
        }


    }


    private var bufferedSource: BufferedSource? = null


    override fun contentType(): MediaType? {
        return responseBody?.contentType()
    }


    override fun contentLength(): Long {
        return responseBody?.contentLength() ?: -1
    }


    override fun source(): BufferedSource {
        if (bufferedSource == null) {
            bufferedSource = source(responseBody!!.source()).buffer()
        }
        return bufferedSource!!
    }


    private fun source(source: Source): Source {
        return object : ForwardingSource(source) {
```

```
var totalBytesRead = 0L



@Throws(IOException::class)
override fun read(sink: Buffer, byteCount: Long): Long {
    val bytesRead = super.read(sink, byteCount)
    totalBytesRead += if (bytesRead != -1L) {
        bytesRead
    } else {
        0
    }
    val contentLength = contentLength()
    val progress = when {
        bytesRead == -1L -> {
            100
        }
        contentLength != -1L -> {
            ((totalBytesRead * 1.0 / contentLength) * 100).toInt()
        }
        else -> {
            0
        }
    }
```

```
                mainHandler.sendMessage(Message().apply {
                    what = CODE_PROGRESS
                    obj = Pair(imageUrl, progress)
                })
                return bytesRead
            }
        }
    }

}
```

复制代码

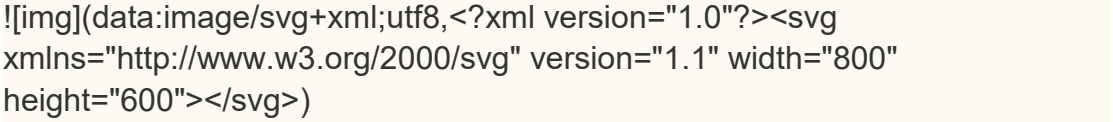
然后在 Interceptor 中使用 ProgressResponseBody 对原始的 ResponseBody 多进行一层包装，将我们的 ProgressResponseBody 作为一个代理，之后再将 ProgressInterceptor 添加给 OkHttpClient 即可

```
/**
 * 作者：leavesC
 * 时间：2020/11/6 22:08
 * 描述：
 * GitHub：https://github.com/leavesC
 */class ProgressInterceptor : Interceptor {

    override fun intercept(chain: Interceptor.Chain): Response {
        val request = chain.request()
```

```
        val originalResponse = chain.proceed(request)

        val url = request.url.toString()

        return originalResponse.newBuilder()

            .body(ProgressResponseBody(url, originalResponse.body))

            .build()

    }



}
```

复制代码

最终实现的效果：



## 四、自定义磁盘缓存 key

在某些时候，我们拿到的图片 Url 可能是带有时效性的，需要在 Url 的尾部加上一个 token 值，在指定时间后 token 就会失效，防止图片被盗链。这种类型的 Url 在一定时间内就需要更换 token 才能拿到图片，可是 Url 的变化就会导致 Glide 的磁盘缓存机制完全失效

```
https://images.pexels.com/photos/1425174/pexels-photo-1425174.jpeg?auto=compress&cs
=tinysrgb&dpr=2&h=750&w=1260&token=tokenValue
```

复制代码

从我的上篇文章内容可以知道，一张图片在进行磁盘缓存时必定会同时对应一个唯一 Key，这样 Glide 在后续加载同样的图片时才能复用已有的缓存文件。对于一张网络图片来说，其唯一 Key 的生成就依赖于 GlideUrl 类的 `getCacheKey()` 方法，该方法会直接返回网络图片的 Url 字符串。如果 Url 的 token 值会一直变化，那么 Glide 就无法对应上同一张图片了，导致磁盘缓存完全失效

```
/**

 * @Author: leavesC

 * @Date: 2020/11/6 15:13

 * @Desc:

 * GitHub: https://github.com/leavesC

 */public class GlideUrl implements Key {



 @Nullable private final String stringUrl;



 public GlideUrl(String url) {

   this(url, Headers.DEFAULT);

 }



 public GlideUrl(String url, Headers headers) {

   this.url = null;

   this.stringUrl = Preconditions.checkNotEmpty(url);

   this.headers = Preconditions.checkNotNull(headers);

 }
```

```
  public String getCacheKey() {

    return stringUrl != null ? stringUrl : Preconditions.checkNotNull(url).toString
();

  }



}



复制代码
```

想要解决这个问题，就需要来手动定义磁盘缓存时的唯一 Key。这可以通过继承 GlideUrl，修改`getCacheKey()`方法的返回值来实现，将 Url 移除 token 键值对后的字符串作为缓存 Key 即可

```
/**

 * @Author: leavesC

 * @Date: 2020/11/6 15:13

 * @Desc:

 * GitHub: https://github.com/leavesC

 */class TokenGlideUrl(private val selfUrl: String) : GlideUrl(selfUrl) {



    override fun getCacheKey(): String {

        val uri = URI(selfUrl)

        val querySplit = uri.query.split("&".toRegex())

        querySplit.forEach {

            val kv = it.split("=".toRegex())
```

```
            if (kv.size == 2 && kv[0] == "token") {

                //将包含 token 的键值对移除

                return selfUrl.replace(it, "")

            }

        }

        return selfUrl

    }



}



复制代码
```

然后在加载图片的时候使用 TokenGlideUrl 来传递图片 Url 即可

```
    Glide.with(Context).load(TokenGlideUrl(ImageUrl)).into(ImageView)



复制代码
```

## 五、如何直接拿到图片

如果想直接取得 Bitmap 而非显示在 ImageView 上的话，可以用以下同步请求的方式来获得 Bitmap。需要注意的是，`submit()`方法就会触发 Glide 去请求图片，此时请求操作还是运行于 Glide 内部的线程池的，但 `get()`操作就会直接阻塞所在线程，直到图片加载结束（不管成功与否）才会返回

```
            thread {

                val futureTarget = Glide.with(this)
```

```
                .asBitmap()
                .load(url)
                .submit()
            val bitmap = futureTarget.get()
            runOnUiThread {
                iv_tokenUrl.setImageBitmap(bitmap)
            }
        }
```

复制代码

也可以用类似的方式来拿到 File 或者 Drawable

```
        thread {
            val futureTarget = Glide.with(this)
                .asFile()
                .load(url)
                .submit()
            val file = futureTarget.get()
            runOnUiThread {
                showToast(file.absolutePath)
            }
        }
```

复制代码

Glide 也提供了以下的异步加载方式

```
            Glide.with(this)
                .asBitmap()
                .load(url)
                .into(object : CustomTarget<Bitmap>() {
                    override fun onLoadCleared(placeholder: Drawable?) {
                        showToast("onLoadCleared")
                    }

                    override fun onResourceReady(
                        resource: Bitmap,
                        transition: Transition<in Bitmap>?
                    ) {
                        iv_tokenUrl.setImageBitmap(resource)
                    }
                })
```

复制代码

## 六、Glide 如何实现网络监听

在上篇文章我有讲到，RequestTracker 就用于存储所有加载图片的任务，并提供了**开始、暂停和重启**所有任务的方法，一个常见的需要重启任务的情形就是用户的网络**从无信号状态恢复正常了**，此时就应该自动重启所有未完成的任务

```
ConnectivityMonitor  connectivityMonitor =

        factory.build(

            context.getApplicationContext(),

            new RequestManagerConnectivityListener(requestTracker));



private class RequestManagerConnectivityListener

      implements ConnectivityMonitor.ConnectivityListener {

    @GuardedBy("RequestManager.this")

    private final RequestTracker requestTracker;



    RequestManagerConnectivityListener(@NonNull RequestTracker requestTracker) {

      this.requestTracker = requestTracker;

    }



    @Override

    public void onConnectivityChanged(boolean isConnected) {

      if (isConnected) {

        synchronized (RequestManager.this) {

          //重启未完成的任务

          requestTracker.restartRequests();
```

```
      }

    }

  }

}


复制代码
```

可以看出来，RequestManagerConnectivityListener 本身就只是一个回调函数，重点还需要看 ConnectivityMonitor 是如何实现的。ConnectivityMonitor 实现类就在 DefaultConnectivityMonitorFactory 中获取，内部会判断当前应用是否具有 `NETWORK_PERMISSION` 权限，如果没有的话则返回一个空实现 NullConnectivityMonitor，有权限的话就返回 DefaultConnectivityMonitor，在内部根据 ConnectivityManager 来判断当前的网络连接状态

```
public class DefaultConnectivityMonitorFactory implements ConnectivityMonitorFactory
 {

  private static final String TAG = "ConnectivityMonitor";

  private static final String NETWORK_PERMISSION = "android.permission.ACCESS_NETWOR
K_STATE";



  @NonNull

  @Override

  public ConnectivityMonitor build(

      @NonNull Context context, @NonNull ConnectivityMonitor.ConnectivityListener li
stener) {

    int permissionResult = ContextCompat.checkSelfPermission(context, NETWORK_PERMIS
SION);

    boolean hasPermission = permissionResult == PackageManager.PERMISSION_GRANTED;
```

```
    if (Log.isLoggable(TAG, Log.DEBUG)) {
      Log.d(
          TAG,
          hasPermission
              ? "ACCESS_NETWORK_STATE permission granted, registering connectivity mo
nitor"
              : "ACCESS_NETWORK_STATE permission missing, cannot register connectivit
y monitor");
    }
    return hasPermission
        ? new DefaultConnectivityMonitor(context, listener)
        : new NullConnectivityMonitor();
  }
}
```

复制代码

DefaultConnectivityMonitor 的逻辑比较简单，不过多赘述。我觉得比较有价值的一点是：Glide 由于使用人数众多，有比较多的开发者会反馈 issues，DefaultConnectivityMonitor 内部就对各种可能抛出 Exception 的情况进行了捕获，这样相对来说会比我们自己实现的逻辑要考虑周全得多，所以我就把 DefaultConnectivityMonitor 复制出来转为 kotlin 以便后续自己复用了

```
/**
 * @Author: LeavesC
 * @Date: 2020/11/7 14:40
 * @Desc:
```

```
 */internal interface ConnectivityListener {

    fun onConnectivityChanged(isConnected: Boolean)

}

internal class DefaultConnectivityMonitor(

    context: Context,

    val listener: ConnectivityListener

) {



    private val appContext = context.applicationContext



    private var isConnected = false



    private var isRegistered = false



    private val connectivityReceiver: BroadcastReceiver = object : BroadcastReceiver
() {

        override fun onReceive(context: Context, intent: Intent) {

            val wasConnected = isConnected

            isConnected = isConnected(context)

            if (wasConnected != isConnected) {

                listener.onConnectivityChanged(isConnected)

            }

        }
```

```
    }


    private fun register() {
        if (isRegistered) {
            return
        }
        // Initialize isConnected.
        isConnected = isConnected(appContext)
        try {
            appContext.registerReceiver(
                connectivityReceiver,
                IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION)
            )
            isRegistered = true
        } catch (e: SecurityException) {
            e.printStackTrace()
        }
    }


    private fun unregister() {
        if (!isRegistered) {
            return
        }
```

```
        appContext.unregisterReceiver(connectivityReceiver)
        isRegistered = false
    }

    @SuppressLint("MissingPermission")
    private fun isConnected(context: Context): Boolean {
        val connectivityManager =
            context.getSystemService(Context.CONNECTIVITY_SERVICE) as? ConnectivityMa
nager
                ?: return true
        val networkInfo = try {
            connectivityManager.activeNetworkInfo
        } catch (e: RuntimeException) {
            return true
        }
        return networkInfo != null && networkInfo.isConnected
    }

    fun onStart() {
        register()
    }

    fun onStop() {
```

```
        unregister()

    }



}
```

## 七、结尾

关于 Glide 的知识点扩展也介绍完了，上述的所有示例代码我也都放到 GitHub 了，欢迎 star：AndroidOpenSourceDemo