

2019 一线互联网三方源码高频面试总结

1.Glide : 加载、缓存、LRU 算法 (如何自己设计一个大图加载框架) (LRUCache 原理)

如何阅读源码

在开始解析 Glide 源码之前,我想先和大家谈一下该如何阅读源码,这个问题也是我平时被问得比较多的,因为很多人都觉得阅读源码是一件比较困难的事情。

那么阅读源码到底困难吗?这个当然主要还是要视具体的源码而定。比如同样是图片加载框架,我读 Volley 的源码时就感觉酣畅淋漓,并且对 Volley 的架构设计和代码质量深感佩服。读 Glide 的源码时却让我相当痛苦,代码极其难懂。当然这里我并不是说 Glide 的代码写得不好,只是因为 Glide 和复杂程度和 Volley 完全不是在一个量级上的。

那么,虽然源码的复杂程度是外在的不可变条件,但我们却可以通过一些技巧来提升自己阅读源码的能力。这里我和大家分享一下我平时阅读源码时所使用的技巧,简单概括就是八个字:抽丝剥茧、点到即止。应该认准一个功能点,然后去分析这个功能点是如何实现的。但只要去追寻主体的实现逻辑即可,千万不要试图去搞懂每一行代码都是什么意思,那样很容易会陷入到思维黑洞当中,而且越陷越深。因为这些庞大的系统都不是由一个人写出来的,每一行代码都想搞明白,就会感觉自己是在盲人摸象,永远也研究不透。如果只是去分析主体的实现逻辑,那么就有比较明确的目的性,这样阅读源码会更加轻松,也更加有成效。

而今天带大家阅读的 Glide 源码就非常适合使用这个技巧,因为 Glide 的源码太复杂了,千万不要试图去搞明白它每行代码的作用,而是应该只分析它的主体实现逻辑。那么我们本篇文章就先确立好一个目标,就是要通过阅读源码搞明白下面这行代码:

```
Glide.with(this).load(url).into(imageView);
```

到底是如何实现将一张网络图片展示到 ImageView 上面的。先将 Glide 的一整套图片加载机制的基本流程梳理清楚,然后我们再通过后面的几篇文章具体去了解 Glide 源码方方面面的细节。

准备好了吗?那么我们现在开始。

源码下载

既然是要阅读 Glide 的源码,那么我们自然需要先将 Glide 的源码下载下来。其实如果你是在使用在 build.gradle 中添加依赖的方式将 Glide 引入到项目中的,那么源码自动就已经下载下来了,在 Android Studio 中就可以直接进行查看。

不过，使用添加依赖的方式引入的 **Glide**，我们只能看到它的源码，但不能做任何修改，如果你还需要修改它的源码的话，可以到 **GitHub** 上面将它的完整源码下载下来。

Glide 的 **GitHub** 主页的地址是：<https://github.com/bumptech/glide>

不过在这个地址下载到的永远都是最新的源码，有可能还正在处于开发当中。而我们整个系列都是使用 **Glide 3.7.0** 这个版本来进行讲解的，因此如果你需要专门去下载 **3.7.0** 版本的源码，可以到这个地址进行下载：<https://github.com/bumptech/glide/tree/v3.7.0>

开始阅读

我们在上一篇文章中已经学习过了，**Glide** 最基本的用法就是三步走：先 **with()**，再 **load()**，最后 **into()**。那么我们开始一步步阅读这三步走的源码，先从 **with()** 看起。

1. with()

with() 方法是 **Glide** 类中的一组静态方法，它有好几个方法重载，我们来看一下 **Glide** 类中所有 **with()** 方法的方法重载：

```
public class Glide {  
  
    ...  
  
    public static RequestManager with(Context context) {  
        RequestManagerRetriever retriever = RequestManagerRetriever.get();  
        return retriever.get(context);  
    }  
  
    public static RequestManager with(Activity activity) {  
        RequestManagerRetriever retriever = RequestManagerRetriever.get();  
        return retriever.get(activity);  
    }  
  
    public static RequestManager with(FragmentActivity activity) {  
        RequestManagerRetriever retriever = RequestManagerRetriever.get();  
        return retriever.get(activity);  
    }  
  
    @TargetApi(Build.VERSION_CODES.HONEYCOMB)  
    public static RequestManager with(android.app.Fragment fragment) {  
        RequestManagerRetriever retriever = RequestManagerRetriever.get();  
        return retriever.get(fragment);  
    }  
  
    public static RequestManager with(Fragment fragment) {
```

```

        RequestManagerRetriever retriever = RequestManagerRetriever.get();
        return retriever.get(fragment);
    }
}

```

可以看到，with()方法的重载种类非常多，既可以传入 Activity，也可以传入 Fragment 或者是 Context。每一个 with()方法重载的代码都非常简单，都是先调用 RequestManagerRetriever 的静态 get()方法得到一个 RequestManagerRetriever 对象，这个静态 get()方法就是一个单例实现，没什么需要解释的。然后再调用 RequestManagerRetriever 的实例 get()方法，去获取 RequestManager 对象。

而 RequestManagerRetriever 的实例 get()方法中的逻辑是什么样的呢？我们一起来看看：

```

public class RequestManagerRetriever implements Handler.Callback {

    private static final RequestManagerRetriever INSTANCE = new RequestManagerRetriever();

    private volatile RequestManager applicationManager;

    ...

    /**
     * Retrieves and returns the RequestManagerRetriever singleton.
     */
    public static RequestManagerRetriever get() {
        return INSTANCE;
    }

    private RequestManager getApplicationManager(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 fragment or activity.
                    // However, in this case since the manager attached to the application
                    will not receive lifecycle
                    // events, we must force the manager to start resumed using
                    ApplicationLifecycle.
                    applicationManager = new
                    RequestManager(context.getApplicationContext(),
                                    new ApplicationLifecycle(), new
                    EmptyRequestManagerTreeNode());
                }
            }
        }
    }
}

```

```

    }
    return applicationManager;
}

public RequestManager get(Context context) {
    if (context == null) {
        throw new IllegalArgumentException("You cannot start a load on a null Context");
    } else if (Util.isOnMainThread() && !(context instanceof Application)) {
        if (context instanceof FragmentActivity) {
            return get((FragmentActivity) context);
        } else if (context instanceof Activity) {
            return get((Activity) context);
        } else if (context instanceof ContextWrapper) {
            return get(((ContextWrapper) context).getBaseContext());
        }
    }
    return getApplicationManager(context);
}

public RequestManager get(FragmentActivity activity) {
    if (Util.isOnBackgroundThread()) {
        return get(activity.getApplicationContext());
    } else {
        assertNotDestroyed(activity);
        FragmentManager fm = activity.getSupportFragmentManager();
        return supportFragmentGet(activity, fm);
    }
}

public RequestManager get(Fragment fragment) {
    if (fragment.getActivity() == null) {
        throw new IllegalArgumentException("You cannot start a load on a fragment before it is attached");
    }
    if (Util.isOnBackgroundThread()) {
        return get(fragment.getActivity().getApplicationContext());
    } else {
        FragmentManager fm = fragment.getChildFragmentManager();
        return supportFragmentGet(fragment.getActivity(), fm);
    }
}

@TargetApi(Build.VERSION_CODES.HONEYCOMB)
public RequestManager get(Activity activity) {

```

```

        if (Util.isOnBackgroundThread() || Build.VERSION.SDK_INT <
Build.VERSION_CODES.HONEYCOMB) {
            return get(activity.getApplicationContext());
        } else {
            assertNotDestroyed(activity);
            android.app.FragmentManager fm = activity.getFragmentManager();
            return fragmentGet(activity, fm);
        }
    }
}

```

```

@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
private static void assertNotDestroyed(Activity activity) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1 &&
activity.isDestroyed()) {
        throw new IllegalArgumentException("You cannot start a load for a destroyed
activity");
    }
}

```

```

@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
public RequestManager get(android.app.Fragment fragment) {
    if (fragment.getActivity() == null) {
        throw new IllegalArgumentException("You cannot start a load on a fragment
before it is attached");
    }
    if (Util.isOnBackgroundThread() || Build.VERSION.SDK_INT <
Build.VERSION_CODES.JELLY_BEAN_MR1) {
        return get(fragment.getActivity().getApplicationContext());
    } else {
        android.app.FragmentManager fm = fragment.getChildFragmentManager();
        return fragmentGet(fragment.getActivity(), fm);
    }
}

```

```

@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
RequestManagerFragment getRequestManagerFragment(final
android.app.FragmentManager fm) {
    RequestManagerFragment current = (RequestManagerFragment)
fm.findFragmentByTag(FRAGMENT_TAG);
    if (current == null) {
        current = pendingRequestManagerFragments.get(fm);
        if (current == null) {
            current = new RequestManagerFragment();
            pendingRequestManagerFragments.put(fm, current);

```

```

        fm.beginTransaction().add(current,
FRAGMENT_TAG).commitAllowingStateLoss();
        handler.obtainMessage(ID_REMOVE_FRAGMENT_MANAGER,
fm).sendToTarget();
    }
}
return current;
}

```

```

@TargetApi(Build.VERSION_CODES.HONEYCOMB)
RequestManager fragmentGet(Context context, android.app.FragmentManager fm) {
    RequestManagerFragment current = getRequestManagerFragment(fm);
    RequestManager requestManager = current.getRequestManager();
    if (requestManager == null) {
        requestManager = new RequestManager(context, current.getLifecycle(),
current.getRequestManagerTreeNode());
        current.setRequestManager(requestManager);
    }
    return requestManager;
}

```

```

SupportRequestManagerFragment            getSupportRequestManagerFragment(final
FragmentManager fm) {
    SupportRequestManagerFragment current = (SupportRequestManagerFragment)
fm.findFragmentByTag(FRAGMENT_TAG);
    if (current == null) {
        current = pendingSupportRequestManagerFragments.get(fm);
        if (current == null) {
            current = new SupportRequestManagerFragment();
            pendingSupportRequestManagerFragments.put(fm, current);
            fm.beginTransaction().add(current,
FRAGMENT_TAG).commitAllowingStateLoss();
            handler.obtainMessage(ID_REMOVE_SUPPORT_FRAGMENT_MANAGER,
fm).sendToTarget();
        }
    }
    return current;
}

```

```

RequestManager supportFragmentGet(Context context, FragmentManager fm) {
    SupportRequestManagerFragment            current =
getSupportRequestManagerFragment(fm);
    RequestManager requestManager = current.getRequestManager();
    if (requestManager == null) {

```

```

        requestManager = new RequestManager(context, current.getLifecycle(),
current.getRequestManagerTreeNode());
        current.setRequestManager(requestManager);
    }
    return requestManager;
}

...
}

```

上述代码虽然看上去逻辑有点复杂，但是将它们梳理清楚后还是很简单的。`RequestManagerRetriever` 类中看似有很多个 `get()` 方法的重载，什么 `Context` 参数，`Activity` 参数，`Fragment` 参数等等，实际上只有两种情况而已，即传入 `Application` 类型的参数，和传入非 `Application` 类型的参数。

我们先来看传入 `Application` 参数的情况。如果在 `Glide.with()` 方法中传入的是一个 `Application` 对象，那么这里就会调用带有 `Context` 参数的 `get()` 方法重载，然后会在第 44 行调用 `getApplicationManager()` 方法来获取一个 `RequestManager` 对象。其实这是最简单的一种情况，因为 `Application` 对象的生命周期即应用程序的生命周期，因此 `Glide` 并不需要做什么特殊的处理，它自动就是和应用程序的生命周期是同步的，如果应用程序关闭的话，`Glide` 的加载也会同时终止。

接下来我们看传入非 `Application` 参数的情况。不管你在 `Glide.with()` 方法中传入的是 `Activity`、`FragmentActivity`、`v4` 包下的 `Fragment`、还是 `app` 包下的 `Fragment`，最终的流程都是一样的，那就是会向当前的 `Activity` 当中添加一个隐藏的 `Fragment`。具体添加的逻辑是在上述代码的第 117 行和第 141 行，分别对应的 `app` 包和 `v4` 包下的两种 `Fragment` 的情况。那么这里为什么要添加一个隐藏的 `Fragment` 呢？因为 `Glide` 需要知道加载的生命周期。很简单的一个道理，如果你在某个 `Activity` 上正在加载着一张图片，结果图片还没加载出来，`Activity` 就被用户关掉了，那么图片还应该继续加载吗？当然不应该。可是 `Glide` 并没有办法知道 `Activity` 的生命周期，于是 `Glide` 就使用了添加隐藏 `Fragment` 的这种小技巧，因为 `Fragment` 的生命周期和 `Activity` 是同步的，如果 `Activity` 被销毁了，`Fragment` 是可以监听到的，这样 `Glide` 就可以捕获这个事件并停止图片加载了。

这里额外再提一句，从第 48 行代码可以看出，如果我们是在非主线程当中使用的 `Glide`，那么不管你是传入的 `Activity` 还是 `Fragment`，都会被强制当成 `Application` 来处理。不过其实这就属于是在分析代码的细节了，本篇文章我们将会把目光主要放在 `Glide` 的主线工作流程上面，后面不会过多去分析这些细节方面的内容。

总体来说，第一个 `with()` 方法的源码还是比较好理解的。其实就是为了得到一个 `RequestManager` 对象而已，然后 `Glide` 会根据我们传入 `with()` 方法的参数来确定图片加载的生命周期，并没有什么特别复杂的逻辑。不过复杂的逻辑还在后面等着我们呢，接下来我们开始分析第二步，`load()` 方法。

2. load()

由于 `with()` 方法返回的是一个 `RequestManager` 对象，那么很容易就能想到，`load()` 方法是在

RequestManager 类当中的，所以说我们首先要看的就是 RequestManager 这个类。不过在上一篇文章中我们学过，Glide 是支持图片 URL 字符串、图片本地路径等等加载形式的，因此 RequestManager 中也有很多个 load() 方法的重载。但是这里我们不可能把每个 load() 方法的重载都看一遍，因此我们就只选其中一个加载图片 URL 字符串的 load() 方法来进行研究吧。

RequestManager 类的简化代码如下所示：

```
public class RequestManager implements LifecycleListener {

    ...

    /**
     * Returns a request builder to load the given {@link String}.
     * signature.
     *
     * @see #fromString()
     * @see #load(Object)
     *
     * @param string A file path, or a uri or url handled by {@link
com.bumptech.glide.load.model.UriLoader}.
     */
    public DrawableTypeRequest<String> load(String string) {
        return (DrawableTypeRequest<String>) fromString().load(string);
    }

    /**
     * Returns a request builder that loads data from {@link String}s using an empty signature.
     *
     * <p>
     * Note - this method caches data using only the given String as the cache key. If the
data is a Uri outside of
     * your control, or you otherwise expect the data represented by the given String to
change without the String
     * identifier changing, Consider using
     * {@link GenericRequestBuilder#signature(Key)} to mixin a signature
     * you create that identifies the data currently at the given String that will invalidate
the cache if that data
     * changes. Alternatively, using {@link DiskCacheStrategy#NONE} and/or
     * {@link DrawableRequestBuilder#skipMemoryCache(boolean)} may be appropriate.
     * </p>
     *
     * @see #from(Class)
     * @see #load(String)
     */
}
```



```

        public DrawableTypeRequest<String> fromString() {
            return loadGeneric(String.class);
        }

        private <T> DrawableTypeRequest<T> loadGeneric(Class<T> modelClass) {
            ModelLoader<T,
                InputStream> streamModelLoader =
            Glide.buildStreamModelLoader(modelClass, context);
            ModelLoader<T, ParcelFileDescriptor> fileDescriptorModelLoader =
                Glide.buildFileDescriptorModelLoader(modelClass, context);
            if (modelClass != null && streamModelLoader == null && fileDescriptorModelLoader ==
            null) {
                throw new IllegalArgumentException("Unknown type " + modelClass + ". You must
            provide a Model of a type for"
                    + " which there is a registered ModelLoader, if you are using a custom
            model, you must first call"
                    + " Glide#register with a ModelLoaderFactory for your custom model
            class");
            }
            return optionsApplier.apply(
                new
                DrawableTypeRequest<T>(modelClass,
                    streamModelLoader,
            fileDescriptorModelLoader, context,
                    glide, requestTracker, lifecycle, optionsApplier));
        }

        ...
    }

```

RequestManager 类的代码是非常多的，但是经过我这样简化之后，看上去就比较清爽了。在我们只探究加载图片 URL 字符串这一个 **load()** 方法的情况下，那么比较重要的方法就只剩下上述代码中的这三个方法。

那么我们先来看 **load()** 方法，这个方法中的逻辑是非常简单的，只有一行代码，就是先调用了 **fromString()** 方法，再调用 **load()** 方法，然后把传入的图片 URL 地址传进去。而 **fromString()** 方法也极为简单，就是调用了 **loadGeneric()** 方法，并且指定参数为 **String.class**，因为 **load()** 方法传入的是一个字符串参数。那么看上去，好像主要的工作都是在 **loadGeneric()** 方法中进行的了。

其实 **loadGeneric()** 方法也没几行代码，这里分别调用了 **Glide.buildStreamModelLoader()** 方法和 **Glide.buildFileDescriptorModelLoader()** 方法来获得 **ModelLoader** 对象。**ModelLoader** 对象是用于加载图片的，而我们给 **load()** 方法传入不同类型的参数，这里也会得到不同的 **ModelLoader** 对象。不过 **buildStreamModelLoader()** 方法内部的逻辑还是蛮复杂的，这里就不展开介绍了，要不然篇幅实在收不住，感兴趣的话你可以自己研究。由于我们刚才传入的参数是 **String.class**，因此最终得到的是 **StreamStringLoader** 对象，它是实现了 **ModelLoader** 接口的。

最后我们可以看到，loadGeneric()方法是要返回一个 DrawableTypeRequest 对象的，因此在 loadGeneric()方法的最后又去 new 了一个 DrawableTypeRequest 对象，然后把刚才获得的 ModelLoader 对象，还有一大堆杂七杂八的东西都传了进去。具体每个参数的含义和作用就不解释了，我们只看主线程流程。

那么这个 DrawableTypeRequest 的作用是什么呢？我们来看下它的源码，如下所示：

```
public class DrawableTypeRequest<ModelType> extends DrawableRequestBuilder<ModelType>
implements DownloadOptions {
    private final ModelLoader<ModelType, InputStream> streamModelLoader;
    private final ModelLoader<ModelType, ParcelFileDescriptor> fileDescriptorModelLoader;
    private final RequestManager.OptionsApplier optionsApplier;

    private static <A, Z, R> FixedLoadProvider<A, ImageVideoWrapper, Z, R> buildProvider(Glide
glide,
        ModelLoader<A, InputStream> streamModelLoader,
        ModelLoader<A, ParcelFileDescriptor> fileDescriptorModelLoader, Class<Z>
resourceClass,
        Class<R> transcodedClass,
        ResourceTranscoder<Z, R> transcoder) {
        if (streamModelLoader == null && fileDescriptorModelLoader == null) {
            return null;
        }

        if (transcoder == null) {
            transcoder = glide.buildTranscoder(resourceClass, transcodedClass);
        }

        DataLoadProvider<ImageVideoWrapper, Z> dataLoadProvider =
glide.buildDataProvider(ImageVideoWrapper.class,
            resourceClass);

        ImageVideoModelLoader<A> modelLoader = new
ImageVideoModelLoader<A>(streamModelLoader,
            fileDescriptorModelLoader);

        return new FixedLoadProvider<A, ImageVideoWrapper, Z, R>(modelLoader, transcoder,
dataLoadProvider);
    }

    DrawableTypeRequest(Class<ModelType> modelClass, ModelLoader<ModelType,
InputStream> streamModelLoader,
        ModelLoader<ModelType, ParcelFileDescriptor> fileDescriptorModelLoader,
Context context, Glide glide,
        RequestTracker requestTracker, Lifecycle lifecycle, RequestManager.OptionsApplier
optionsApplier) {
```

```

        super(context, modelClass,
                buildProvider(glide, streamModelLoader, fileDescriptorModelLoader,
GifBitmapWrapper.class,
                GlideDrawable.class, null),
                glide, requestTracker, lifecycle);
        this.streamModelLoader = streamModelLoader;
        this.fileDescriptorModelLoader = fileDescriptorModelLoader;
        this.optionsApplier = optionsApplier;
    }

    /**
     * Attempts to always load the resource as a {@link android.graphics.Bitmap}, even if it
     could actually be animated.
     *
     * @return A new request builder for loading a {@link android.graphics.Bitmap}
     */
    public BitmapTypeRequest<ModelType> asBitmap() {
        return optionsApplier.apply(new BitmapTypeRequest<ModelType>(this,
streamModelLoader,
                fileDescriptorModelLoader, optionsApplier));
    }

    /**
     * Attempts to always load the resource as a {@link
com.bumptech.glide.load.resource.gif.GifDrawable}.
     * <p>
     * If the underlying data is not a GIF, this will fail. As a result, this should only be used
     if the model
     * represents an animated GIF and the caller wants to interact with the GifDrawable
     directly. Normally using
     * just an {@link DrawableTypeRequest} is sufficient because it will determine
     whether or
     * not the given data represents an animated GIF and return the appropriate animated
     or not animated
     * {@link android.graphics.drawable.Drawable} automatically.
     * </p>
     *
     * @return A new request builder for loading a {@link
com.bumptech.glide.load.resource.gif.GifDrawable}.
     */
    public GifTypeRequest<ModelType> asGif() {
        return optionsApplier.apply(new GifTypeRequest<ModelType>(this,
streamModelLoader, optionsApplier));
    }

```

```
...  
}
```

这个类中的代码本身就不多，我只是稍微做了一点简化。可以看到，最主要的就是它提供了 `asBitmap()` 和 `asGif()` 这两个方法。这两个方法我们在上一篇文章当中都是学过的，分别是用于强制指定加载静态图片和动态图片。而从源码中可以看出，它们分别又创建了一个 `BitmapTypeRequest` 和 `GifTypeRequest`，如果没有进行强制指定的话，那默认就是使用 `DrawableTypeRequest`。

好的，那么我们再回到 `RequestManager` 的 `load()` 方法中。刚才已经分析过了，`fromString()` 方法会返回一个 `DrawableTypeRequest` 对象，接下来会调用这个对象的 `load()` 方法，把图片的 URL 地址传进去。但是我们刚才看到了，`DrawableTypeRequest` 中并没有 `load()` 方法，那么很容易就能猜想到，`load()` 方法是在父类当中的。

`DrawableTypeRequest` 的父类是 `DrawableRequestBuilder`，我们来看下这个类的源码：

```
public class DrawableRequestBuilder<ModelType>  
    extends GenericRequestBuilder<ModelType, ImageVideoWrapper, GifBitmapWrapper,  
    GlideDrawable>  
    implements BitmapOptions, DrawableOptions {  
  
    DrawableRequestBuilder(Context context, Class<ModelType> modelClass,  
        LoadProvider<ModelType> loadProvider, ImageVideoWrapper, GifBitmapWrapper,  
    GlideDrawable> glide, Lifecycle lifecycle) {  
        super(context, modelClass, loadProvider, GlideDrawable.class, glide, requestTracker,  
    lifecycle);  
        // Default to animating.  
        crossFade();  
    }  
  
    public DrawableRequestBuilder<ModelType> thumbnail(  
        DrawableRequestBuilder<?> thumbnailRequest) {  
        super.thumbnail(thumbnailRequest);  
        return this;  
    }  
  
    @Override  
    public DrawableRequestBuilder<ModelType> thumbnail(  
        GenericRequestBuilder<?, ?, ?, GlideDrawable> thumbnailRequest) {  
        super.thumbnail(thumbnailRequest);  
        return this;  
    }  
}
```

```

@Override
public DrawableRequestBuilder<ModelType> thumbnail(float sizeMultiplier) {
    super.thumbnail(sizeMultiplier);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType> sizeMultiplier(float sizeMultiplier) {
    super.sizeMultiplier(sizeMultiplier);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
decoder(ResourceDecoder<ImageVideoWrapper, GifBitmapWrapper> decoder) {
    super.decoder(decoder);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType> cacheDecoder(ResourceDecoder<File,
GifBitmapWrapper> cacheDecoder) {
    super.cacheDecoder(cacheDecoder);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
encoder(ResourceEncoder<GifBitmapWrapper> encoder) {
    super.encoder(encoder);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType> priority(Priority priority) {
    super.priority(priority);
    return this;
}

public DrawableRequestBuilder<ModelType> transform(BitmapTransformation...
transformations) {
    return bitmapTransform(transformations);
}

```

```

    public DrawableRequestBuilder<ModelType> centerCrop() {
        return transform(glide.getDrawableCenterCrop());
    }

    public DrawableRequestBuilder<ModelType> fitCenter() {
        return transform(glide.getDrawableFitCenter());
    }

    public DrawableRequestBuilder<ModelType> bitmapTransform(Transformation<Bitmap>...
    bitmapTransformations) {
        GifBitmapWrapperTransformation[] transformations =
            new GifBitmapWrapperTransformation[bitmapTransformations.length];
        for (int i = 0; i < bitmapTransformations.length; i++) {
            transformations[i] = new GifBitmapWrapperTransformation(glide.getBitmapPool(),
            bitmapTransformations[i]);
        }
        return transform(transformations);
    }

    @Override
    public DrawableRequestBuilder<ModelType>
    transform(Transformation<GifBitmapWrapper>... transformation) {
        super.transform(transformation);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> transcoder(
        ResourceTranscoder<GifBitmapWrapper, GlideDrawable> transcoder) {
        super.transcoder(transcoder);
        return this;
    }

    public final DrawableRequestBuilder<ModelType> crossFade() {
        super.animate(new DrawableCrossFadeFactory<GlideDrawable>());
        return this;
    }

    public DrawableRequestBuilder<ModelType> crossFade(int duration) {
        super.animate(new DrawableCrossFadeFactory<GlideDrawable>(duration));
        return this;
    }

    public DrawableRequestBuilder<ModelType> crossFade(int animationId, int duration) {

```

```

        super.animate(new DrawableCrossFadeFactory<GlideDrawable>(context, animationId,
            duration));
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> dontAnimate() {
        super.dontAnimate();
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> animate(ViewPropertyAnimation.Animator
    animator) {
        super.animate(animator);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> animate(int animationId) {
        super.animate(animationId);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> placeholder(int resourceId) {
        super.placeholder(resourceId);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> placeholder(Drawable drawable) {
        super.placeholder(drawable);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> fallback(Drawable drawable) {
        super.fallback(drawable);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> fallback(int resourceId) {

```

```

        super.fallback(resourceId);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> error(int resourceId) {
        super.error(resourceId);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> error(Drawable drawable) {
        super.error(drawable);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> listener(
        RequestListener<? super ModelType, GlideDrawable> requestListener) {
        super.listener(requestListener);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> diskCacheStrategy(DiskCacheStrategy strategy)
    {
        super.diskCacheStrategy(strategy);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> skipMemoryCache(boolean skip) {
        super.skipMemoryCache(skip);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> override(int width, int height) {
        super.override(width, height);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> sourceEncoder(Encoder<ImageVideoWrapper>
sourceEncoder) {

```



```

        super.sourceEncoder(sourceEncoder);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> dontTransform() {
        super.dontTransform();
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> signature(Key signature) {
        super.signature(signature);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> load(ModelType model) {
        super.load(model);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> clone() {
        return (DrawableRequestBuilder<ModelType>) super.clone();
    }

    @Override
    public Target<GlideDrawable> into(ImageView view) {
        return super.into(view);
    }

    @Override
    void applyFitCenter() {
        fitCenter();
    }

    @Override
    void applyCenterCrop() {
        centerCrop();
    }
}

```

DrawableRequestBuilder 中有很多个方法，这些方法其实就是 **Glide** 绝大多数的 API 了。里面有不少我们在上篇文章中已经用过了，比如说 **placeholder()** 方法、**error()** 方法、

`diskCacheStrategy()`方法、`override()`方法等。当然还有很多暂时还没用到的 API，我们会在后面的文章当中学习。

到这里，第二步 `load()` 方法也就分析结束了。为什么呢？因为你会发现 `DrawableRequestBuilder` 类中有一个 `into()`方法（上述代码第 220 行），也就是说，最终 `load()` 方法返回的其实就是一个 `DrawableTypeRequest` 对象。那么接下来我们就要进行第三步了，分析 `into()`方法中的逻辑。

3. `into()`

如果说前面两步都是在准备开胃小菜的话，那么现在终于要进入主菜了，因为 `into()`方法也是整个 `Glide` 图片加载流程中逻辑最复杂的地方。

不过从刚才的代码来看，`into()`方法中并没有任何逻辑，只有一句 `super.into(view)`。那么很显然，`into()`方法的具体逻辑都是在 `DrawableRequestBuilder` 的父类当中了。

`DrawableRequestBuilder` 的父类是 `GenericRequestBuilder`，我们来看一下 `GenericRequestBuilder` 类中的 `into()`方法，如下所示：

```
public Target<TranscodeType> into(ImageView view) {
    Util.assertMainThread();
    if (view == null) {
        throw new IllegalArgumentException("You must pass in a non null View");
    }
    if (!isTransformationSet && view.getScaleType() != null) {
        switch (view.getScaleType()) {
            case CENTER_CROP:
                applyCenterCrop();
                break;
            case FIT_CENTER:
            case FIT_START:
            case FIT_END:
                applyFitCenter();
                break;
            //$CASES-OMITTED$
            default:
                // Do nothing.
        }
    }
    return into(glide.buildImageViewTarget(view, transcodeClass));
}
```

这里前面一大堆的判断逻辑我们都可以先不用管，等到后面文章讲 `transform` 的时候会再进行解释，现在我们只需要关注最后一行代码。最后一行代码先是调用了 `glide.buildImageViewTarget()`方法，这个方法会构建出一个 `Target` 对象，`Target` 对象则是用来最终展示图片用的，如果我们跟进去的话会看到如下代码：

```
<R> Target<R> buildImageViewTarget(ImageView imageView, Class<R> transcodedClass) {
    return imageViewTargetFactory.buildTarget(imageView, transcodedClass);
}
```

这里其实又是调用了 `ImageViewTargetFactory` 的 `buildTarget()` 方法，我们继续跟进去，代码如下所示：

```
public class ImageViewTargetFactory {

    @SuppressWarnings("unchecked")
    public <Z> Target<Z> buildTarget(ImageView view, Class<Z> clazz) {
        if (GlideDrawable.class.isAssignableFrom(clazz)) {
            return (Target<Z>) new GlideDrawableImageViewTarget(view);
        } else if (Bitmap.class.equals(clazz)) {
            return (Target<Z>) new BitmapImageViewTarget(view);
        } else if (Drawable.class.isAssignableFrom(clazz)) {
            return (Target<Z>) new DrawableImageViewTarget(view);
        } else {
            throw new IllegalArgumentException("Unhandled class: " + clazz
                + ", try .as*(Class).transcode(ResourceTranscoder)");
        }
    }
}
```

可以看到，在 `buildTarget()` 方法中会根据传入的 `class` 参数来构建不同的 `Target` 对象。那如果你要分析这个 `class` 参数是从哪儿传过来的，这可有得你分析了，简单起见我直接帮大家梳理清楚。这个 `class` 参数其实基本上只有两种情况，如果你在使用 `Glide` 加载图片的时候调用了 `asBitmap()` 方法，那么这里就会构建出 `BitmapImageViewTarget` 对象，否则的话构建的都是 `GlideDrawableImageViewTarget` 对象。至于上述代码中的 `DrawableImageViewTarget` 对象，这个通常都是用不到的，我们可以暂时不用管它。

也就是说，通过 `glide.buildImageViewTarget()` 方法，我们构建出了一个 `GlideDrawableImageViewTarget` 对象。那现在回到刚才 `into()` 方法的最后一行，可以看到，这里又将这个参数传入到了 `GenericRequestBuilder` 另一个接收 `Target` 对象的 `into()` 方法当中了。我们来看一下这个 `into()` 方法的源码：

```
public <Y extends Target<TranscodeType>> Y into(Y target) {
    Util.assertMainThread();
    if (target == null) {
        throw new IllegalArgumentException("You must pass in a non null Target");
    }
    if (!isModelSet) {
        throw new IllegalArgumentException("You must first set a model (try #load())");
    }
    Request previous = target.getRequest();
```

```

        if (previous != null) {
            previous.clear();
            requestTracker.removeRequest(previous);
            previous.recycle();
        }
        Request request = buildRequest(target);
        target.setRequest(request);
        lifecycle.addListener(target);
        requestTracker.runRequest(request);
        return target;
    }

```

这里我们还是只抓核心代码，其实只有两行是最关键的，第 15 行调用 `buildRequest()` 方法构建出了一个 `Request` 对象，还有第 18 行来执行这个 `Request`。

`Request` 是用来发出加载图片请求的，它是 `Glide` 中非常关键的一个组件。我们先来看 `buildRequest()` 方法是如何构建 `Request` 对象的：

```

private Request buildRequest(Target<TranscodeType> target) {
    if (priority == null) {
        priority = Priority.NORMAL;
    }
    return buildRequestRecursive(target, null);
}

private Request buildRequestRecursive(Target<TranscodeType> target,
    ThumbnailRequestCoordinator parentCoordinator) {
    if (thumbnailRequestBuilder != null) {
        if (isThumbnailBuilt) {
            throw new IllegalStateException("You cannot use a request as both the main
request and a thumbnail, "
                + "consider using clone() on the request(s) passed to thumbnail()");
        }
        // Recursive case: contains a potentially recursive thumbnail request builder.
        if (thumbnailRequestBuilder.animationFactory.equals(NoAnimation.getFactory())) {
            thumbnailRequestBuilder.animationFactory = animationFactory;
        }

        if (thumbnailRequestBuilder.priority == null) {
            thumbnailRequestBuilder.priority = getThumbnailPriority();
        }

        if (Util.isValidDimensions(overrideWidth, overrideHeight)
            && !Util.isValidDimensions(thumbnailRequestBuilder.overrideWidth,
                thumbnailRequestBuilder.overrideHeight)) {

```

```

        thumbnailRequestBuilder.override(overrideWidth, overrideHeight);
    }

    ThumbnailRequestCoordinator coordinator = new
ThumbnailRequestCoordinator(parentCoordinator);
    Request fullRequest = obtainRequest(target, sizeMultiplier, priority, coordinator);
    // Guard against infinite recursion.
    isThumbnailBuilt = true;
    // Recursively generate thumbnail requests.
    Request thumbRequest = thumbnailRequestBuilder.buildRequestRecursive(target,
coordinator);
    isThumbnailBuilt = false;
    coordinator.setRequests(fullRequest, thumbRequest);
    return coordinator;
} else if (thumbSizeMultiplier != null) {
    // Base case: thumbnail multiplier generates a thumbnail request, but cannot recurse.
    ThumbnailRequestCoordinator coordinator = new
ThumbnailRequestCoordinator(parentCoordinator);
    Request fullRequest = obtainRequest(target, sizeMultiplier, priority, coordinator);
    Request thumbnailRequest = obtainRequest(target, thumbSizeMultiplier,
getThumbnailPriority(), coordinator);
    coordinator.setRequests(fullRequest, thumbnailRequest);
    return coordinator;
} else {
    // Base case: no thumbnail.
    return obtainRequest(target, sizeMultiplier, priority, parentCoordinator);
}
}

```

```

private Request obtainRequest(Target<TranscodeType> target, float sizeMultiplier, Priority
priority,
    RequestCoordinator requestCoordinator) {
    return GenericRequest.obtain(
        loadProvider,
        model,
        signature,
        context,
        priority,
        target,
        sizeMultiplier,
        placeholderDrawable,
        placeholderId,
        errorPlaceholder,
        errorId,

```

```

        fallbackDrawable,
        fallbackResource,
        requestListener,
        requestCoordinator,
        glide.getEngine(),
        transformation,
        transcodeClass,
        isCacheable,
        animationFactory,
        overrideWidth,
        overrideHeight,
        diskCacheStrategy);
    }

```

可以看到，`buildRequest()` 方法的内部其实又调用了 `buildRequestRecursive()` 方法，而 `buildRequestRecursive()` 方法中的代码虽然有点长，但是其中 90% 的代码都是在处理缩略图的。如果我们只追主线流程的话，那么只需要看第 47 行代码就可以了。这里调用了 `obtainRequest()` 方法来获取一个 `Request` 对象，而 `obtainRequest()` 方法中又去调用了 `GenericRequest` 的 `obtain()` 方法。注意这个 `obtain()` 方法需要传入非常多的参数，而其中很多的参数我们都是比较熟悉的，像什么 `placeholderId`、`errorPlaceholder`、`diskCacheStrategy` 等等。因此，我们就有理由猜测，刚才在 `load()` 方法中调用的所有 API，其实都是在这里组装到 `Request` 对象当中的。那么我们进入到这个 `GenericRequest` 的 `obtain()` 方法瞧一瞧：

```

public final class GenericRequest<A, T, Z, R> implements Request, SizeReadyCallback,
    ResourceCallback {

```

```

    ...

```

```

    public static <A, T, Z, R> GenericRequest<A, T, Z, R> obtain(
        LoadProvider<A, T, Z, R> loadProvider,
        A model,
        Key signature,
        Context context,
        Priority priority,
        Target<R> target,
        float sizeMultiplier,
        Drawable placeholderDrawable,
        int placeholderResourceId,
        Drawable errorDrawable,
        int errorResourceId,
        Drawable fallbackDrawable,
        int fallbackResourceId,
        RequestListener<? super A, R> requestListener,
        RequestCoordinator requestCoordinator,
        Engine engine,

```

```

        Transformation<Z> transformation,
        Class<R> transcodeClass,
        boolean isMemoryCacheable,
        GlideAnimationFactory<R> animationFactory,
        int overrideWidth,
        int overrideHeight,
        DiskCacheStrategy diskCacheStrategy) {
    @SuppressWarnings("unchecked")
    GenericRequest<A, T, Z, R> request = (GenericRequest<A, T, Z, R>)
REQUEST_POOL.poll();
    if (request == null) {
        request = new GenericRequest<A, T, Z, R>();
    }
    request.init(loadProvider,
        model,
        signature,
        context,
        priority,
        target,
        sizeMultiplier,
        placeholderDrawable,
        placeholderResourceId,
        errorDrawable,
        errorResourceId,
        fallbackDrawable,
        fallbackResourceId,
        requestListener,
        requestCoordinator,
        engine,
        transformation,
        transcodeClass,
        isMemoryCacheable,
        animationFactory,
        overrideWidth,
        overrideHeight,
        diskCacheStrategy);
    return request;
}

...
}

```

可以看到，这里在第 33 行去 new 了一个 GenericRequest 对象，并在最后一行返回，也就是说，obtain()方法实际上获得的就是一个 GenericRequest 对象。另外这里又是在第 35 行调用了 GenericRequest 的 init()，里面主要就是一些赋值的代码，将传入的这些参数赋值到

GenericRequest 的成员变量当中，我们就不再跟进去看了。

好，那现在解决了构建 Request 对象的问题，接下来我们看一下这个 Request 对象又是怎么执行的。回到刚才的 into() 方法，你会发现在第 18 行调用了 requestTracker.runRequest() 方法来执行这个 Request，那么我们跟进去瞧一瞧，如下所示：

```
/**
 * Starts tracking the given request.
 */
public void runRequest(Request request) {
    requests.add(request);
    if (!isPaused) {
        request.begin();
    } else {
        pendingRequests.add(request);
    }
}
```

这里有一个简单的逻辑判断，就是先判断 Glide 当前是不是处理暂停状态，如果不是暂停状态就调用 Request 的 begin() 方法来执行 Request，否则的话就先将 Request 添加到待执行队列里面，等暂停状态解除了之后再执行。

暂停请求的功能仍然不是这篇文章所关心的，这里就直接忽略了，我们重点来看这个 begin() 方法。由于当前的 Request 对象是一个 GenericRequest，因此这里就需要看 GenericRequest 中的 begin() 方法了，如下所示：

```
@Override
public void begin() {
    startTime = LogTime.getLogTime();
    if (model == null) {
        onException(null);
        return;
    }
    status = Status.WAITING_FOR_SIZE;
    if (Util.isValidDimensions(overrideWidth, overrideHeight)) {
        onSizeReady(overrideWidth, overrideHeight);
    } else {
        target.getSize(this);
    }
    if (!isComplete() && !isFailed() && canNotifyStatusChanged()) {
        target.onLoadStarted(getPlaceholderDrawable());
    }
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logV("finished run method in " + LogTime.getElapsedMillis(startTime));
    }
}
```



```
}
```

这里我们来注意几个细节，首先如果 `model` 等于 `null`，`model` 也就是我们在第二步 `load()` 方法中传入的图片 URL 地址，这个时候会调用 `onException()` 方法。如果你跟到 `onException()` 方法里面去看看，你会发现它最终会调用到一个 `setErrorPlaceholder()` 当中，如下所示：

```
private void setErrorPlaceholder(Exception e) {
    if (!canNotifyStatusChanged()) {
        return;
    }
    Drawable error = model == null ? getFallbackDrawable() : null;
    if (error == null) {
        error = getErrorDrawable();
    }
    if (error == null) {
        error = getPlaceholderDrawable();
    }
    target.onLoadFailed(e, error);
}
```

这个方法中会先去获取一个 `error` 的占位图，如果获取不到的话会再去获取一个 `loading` 占位图，然后调用 `target.onLoadFailed()` 方法并将占位图传入。那么 `onLoadFailed()` 方法中做了什么呢？我们看一下：

```
public abstract class ImageViewTarget<Z> extends ViewTarget<ImageView, Z> implements
GlideAnimation.ViewAdapter {
```

```
...
```

```
@Override
```

```
public void onLoadStarted(Drawable placeholder) {
    view.setImageDrawable(placeholder);
}
```

```
@Override
```

```
public void onLoadFailed(Exception e, Drawable errorDrawable) {
    view.setImageDrawable(errorDrawable);
}
```

```
...
```

```
}
```

很简单，其实就是将这张 `error` 占位图显示到 `ImageView` 上而已，因为现在出现了异常，没办法展示正常的图片了。而如果你仔细看下刚才 `begin()` 方法的第 15 行，你会发现它又调用了 `target.onLoadStarted()` 方法，并传入了一个 `loading` 占位图，在也就是说，在图片请求开始之前，会先使用这张占位图代替最终的图片显示。这也是我们在上一篇文章中学过的 `placeholder()` 和 `error()` 这两个占位图 API 底层的实现原理。

好，那么我们继续回到 `begin()` 方法。刚才讲了占位图的实现，那么具体的图片加载又是从哪里开始的呢？是在 `begin()` 方法的第 10 行和第 12 行。这里要分两种情况，一种是你使用了 `override()` API 为图片指定了一个固定的宽高，一种是没有指定。如果指定了的话，就会执行第 10 行代码，调用 `onSizeReady()` 方法。如果没指定的话，就会执行第 12 行代码，调用 `target.getSize()` 方法。这个 `target.getSize()` 方法的内部会根据 `ImageView` 的 `layout_width` 和 `layout_height` 值做一系列的计算，来算出图片应该的宽高。具体的计算细节我就不带着大家分析了，总之在计算完之后，它也会调用 `onSizeReady()` 方法。也就是说，不管是哪种情况，最终都会调用到 `onSizeReady()` 方法，在这里进行下一步操作。那么我们跟到这个方法里面来：

```
@Override
public void onSizeReady(int width, int height) {
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logV("Got onSizeReady in " + LogTime.getElapsedMillis(startTime));
    }
    if (status != Status.WAITING_FOR_SIZE) {
        return;
    }
    status = Status.RUNNING;
    width = Math.round(sizeMultiplier * width);
    height = Math.round(sizeMultiplier * height);
    ModelLoader<A, T> modelLoader = loadProvider.getModelLoader();
    final DataFetcher<T> dataFetcher = modelLoader.getResourceFetcher(model, width, height);
    if (dataFetcher == null) {
        onException(new Exception("Failed to load model: \"" + model + "\""));
        return;
    }
    ResourceTranscoder<Z, R> transcoder = loadProvider.getTranscoder();
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logV("finished setup for calling load in " + LogTime.getElapsedMillis(startTime));
    }
    loadedFromMemoryCache = true;
    loadStatus = engine.load(signature, width, height, dataFetcher, loadProvider, transformation,
transcoder,
        priority, isMemoryCacheable, diskCacheStrategy, this);
    loadedFromMemoryCache = resource != null;
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logV("finished onSizeReady in " + LogTime.getElapsedMillis(startTime));
    }
}
```

从这里开始，真正复杂的地方来了，我们需要慢慢进行分析。先来看一下，在第 12 行调用了 `loadProvider.getModelLoader()` 方法，那么我们第一个要搞清楚的就是，这个 `loadProvider` 是什么？要搞清楚这点，需要先回到第二步的 `load()` 方法当中。还记得 `load()` 方法是返回一个 `DrawableTypeRequest` 对象吗？刚才我们只是分析了 `DrawableTypeRequest` 当中的

asBitmap() 和 asGif() 方法，并没有仔细看它的构造函数，现在我们重新来看一下 DrawableTypeRequest 类的构造函数：

```
public class DrawableTypeRequest<ModelType> extends DrawableRequestBuilder<ModelType>
implements DownloadOptions {

    private final ModelLoader<ModelType, InputStream> streamModelLoader;
    private final ModelLoader<ModelType, ParcelFileDescriptor> fileDescriptorModelLoader;
    private final RequestManager.OptionsApplier optionsApplier;

    private static <A, Z, R> FixedLoadProvider<A, ImageVideoWrapper, Z, R> buildProvider(Glide
glide,
        ModelLoader<A, InputStream> streamModelLoader,
        ModelLoader<A, ParcelFileDescriptor> fileDescriptorModelLoader, Class<Z>
resourceClass,
        Class<R> transcodedClass,
        ResourceTranscoder<Z, R> transcoder) {
        if (streamModelLoader == null && fileDescriptorModelLoader == null) {
            return null;
        }
        if (transcoder == null) {
            transcoder = glide.buildTranscoder(resourceClass, transcodedClass);
        }
        DataLoadProvider<ImageVideoWrapper, Z> dataLoadProvider =
glide.buildDataProvider(ImageVideoWrapper.class,
            resourceClass);
        ImageVideoModelLoader<A> modelLoader = new
ImageVideoModelLoader<A>(streamModelLoader,
            fileDescriptorModelLoader);
        return new FixedLoadProvider<A, ImageVideoWrapper, Z, R>(modelLoader, transcoder,
dataLoadProvider);
    }

    DrawableTypeRequest(Class<ModelType> modelClass, ModelLoader<ModelType,
InputStream> streamModelLoader,
        ModelLoader<ModelType, ParcelFileDescriptor> fileDescriptorModelLoader,
Context context, Glide glide,
        RequestTracker requestTracker, Lifecycle lifecycle, RequestManager.OptionsApplier
optionsApplier) {
        super(context, modelClass,
            buildProvider(glide, streamModelLoader, fileDescriptorModelLoader,
GifBitmapWrapper.class,
                GlideDrawable.class, null),
            glide, requestTracker, lifecycle);
    }
}
```

```

        this.streamModelLoader = streamModelLoader;
        this.fileDescriptorModelLoader = fileDescriptorModelLoader;
        this.optionsApplier = optionsApplier;
    }

    ...
}

```

可以看到，这里在第 29 行，也就是构造函数中，调用了一个 `buildProvider()` 方法，并把 `streamModelLoader` 和 `fileDescriptorModelLoader` 等参数传入到这个方法中，这两个 `ModelLoader` 就是之前在 `loadGeneric()` 方法中构建出来的。

那么我们再来看一下 `buildProvider()` 方法里面做了什么，在第 16 行调用了 `glide.buildTranscoder()` 方法来构建一个 `ResourceTranscoder`，它是用于对图片进行转码的，由于 `ResourceTranscoder` 是一个接口，这里实际会构建出一个 `GifBitmapWrapperDrawableTranscoder` 对象。

接下来在第 18 行调用了 `glide.buildDataProvider()` 方法来构建一个 `DataLoadProvider`，它是用于对图片进行编解码的，由于 `DataLoadProvider` 是一个接口，这里实际会构建出一个 `ImageVideoGifDrawableLoadProvider` 对象。

然后在第 20 行，`new` 了一个 `ImageVideoModelLoader` 的实例，并把之前 `loadGeneric()` 方法中构建的两个 `ModelLoader` 封装到了 `ImageVideoModelLoader` 当中。

最后，在第 22 行，`new` 出一个 `FixedLoadProvider`，并把刚才构建的出来的 `GifBitmapWrapperDrawableTranscoder`、`ImageVideoModelLoader`、`ImageVideoGifDrawableLoadProvider` 都封装进去，这个也就是 `onSizeReady()` 方法中的 `loadProvider` 了。

好的，那么我们回到 `onSizeReady()` 方法中，在 `onSizeReady()` 方法的第 12 行和第 18 行，分别调用了 `loadProvider` 的 `getModelLoader()` 方法和 `getTranscoder()` 方法，那么得到的对象也就是刚才我们分析的 `ImageVideoModelLoader` 和 `GifBitmapWrapperDrawableTranscoder` 了。而在第 13 行，又调用了 `ImageVideoModelLoader` 的 `getResourceFetcher()` 方法，这里我们又需要跟进去瞧一瞧了，代码如下所示：

```

public class ImageVideoModelLoader<A> implements ModelLoader<A, ImageVideoWrapper> {
    private static final String TAG = "IVML";

    private final ModelLoader<A, InputStream> streamLoader;
    private final ModelLoader<A, ParcelFileDescriptor> fileDescriptorLoader;

    public ImageVideoModelLoader(ModelLoader<A, InputStream> streamLoader,
        ModelLoader<A, ParcelFileDescriptor> fileDescriptorLoader) {
        if (streamLoader == null && fileDescriptorLoader == null) {
            throw new NullPointerException("At least one of streamLoader and

```

```

fileDescriptorLoader must be non null");
    }
    this.streamLoader = streamLoader;
    this.fileDescriptorLoader = fileDescriptorLoader;
}

@Override
public DataFetcher<ImageVideoWrapper> getResourceFetcher(A model, int width, int height)
{
    DataFetcher<InputStream> streamFetcher = null;
    if (streamLoader != null) {
        streamFetcher = streamLoader.getResourceFetcher(model, width, height);
    }
    DataFetcher<ParcelFileDescriptor> fileDescriptorFetcher = null;
    if (fileDescriptorLoader != null) {
        fileDescriptorFetcher = fileDescriptorLoader.getResourceFetcher(model, width,
height);
    }

    if (streamFetcher != null || fileDescriptorFetcher != null) {
        return new ImageVideoFetcher(streamFetcher, fileDescriptorFetcher);
    } else {
        return null;
    }
}

static class ImageVideoFetcher implements DataFetcher<ImageVideoWrapper> {
    private final DataFetcher<InputStream> streamFetcher;
    private final DataFetcher<ParcelFileDescriptor> fileDescriptorFetcher;

    public ImageVideoFetcher(DataFetcher<InputStream> streamFetcher,
        DataFetcher<ParcelFileDescriptor> fileDescriptorFetcher) {
        this.streamFetcher = streamFetcher;
        this.fileDescriptorFetcher = fileDescriptorFetcher;
    }

    ...
}
}

```

可以看到，在第 20 行会先调用 `streamLoader.getResourceFetcher()` 方法获取一个 `DataFetcher`，而这个 `streamLoader` 其实就是我们在 `loadGeneric()` 方法中构建出的 `StreamStringLoader`，调用它的 `getResourceFetcher()` 方法会得到一个 `HttpUrlFetcher` 对象。然后在第 28 行 `new` 出了一个 `ImageVideoFetcher` 对象，并把获得的 `HttpUrlFetcher` 对象传进去。也就是说，`ImageVideoModelLoader` 的 `getResourceFetcher()` 方法得到的是一个 `ImageVideoFetcher`。

那么我们再次回到 `onSizeReady()` 方法，在 `onSizeReady()` 方法的第 23 行，这里将刚才获得的 `ImageVideoFetcher`、`GifBitmapWrapperDrawableTranscoder` 等等一系列的值一起传入到了 `Engine` 的 `load()` 方法当中。接下来我们就要看一看，这个 `Engine` 的 `load()` 方法当中，到底做了什么？代码如下所示：

```
public class Engine implements EngineJobListener,
    MemoryCache.ResourceRemovedListener,
    EngineResource.ResourceListener {

    ...

    public <T, Z, R> LoadStatus load(Key signature, int width, int height, DataFetcher<T> fetcher,
        DataLoadProvider<T, Z> loadProvider, Transformation<Z> transformation,
        ResourceTranscoder<Z, R> transcoder,
        Priority priority, boolean isMemoryCacheable, DiskCacheStrategy
        diskCacheStrategy, ResourceCallback cb) {
        Util.assertMainThread();
        long startTime = LogTime.getLogTime();

        final String id = fetcher.getId();
        EngineKey key = keyFactory.buildKey(id, signature, width, height,
        loadProvider.getCacheDecoder(),
            loadProvider.getSourceDecoder(), transformation, loadProvider.getEncoder(),
            transcoder, loadProvider.getSourceEncoder());

        EngineResource<?> cached = loadFromCache(key, isMemoryCacheable);
        if (cached != null) {
            cb.onResourceReady(cached);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Loaded resource from cache", startTime, key);
            }
            return null;
        }

        EngineResource<?> active = loadFromActiveResources(key, isMemoryCacheable);
        if (active != null) {
            cb.onResourceReady(active);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Loaded resource from active resources", startTime,
key);
            }
            return null;
        }
    }
}
```

```

        EngineJob current = jobs.get(key);
        if (current != null) {
            current.addCallback(cb);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Added to existing load", startTime, key);
            }
            return new LoadStatus(cb, current);
        }

        EngineJob engineJob = engineJobFactory.build(key, isMemoryCacheable);
        DecodeJob<T, Z, R> decodeJob = new DecodeJob<T, Z, R>(key, width, height, fetcher,
loadProvider, transformation,
            transcoder, diskCacheProvider, diskCacheStrategy, priority);
        EngineRunnable runnable = new EngineRunnable(engineJob, decodeJob, priority);
        jobs.put(key, engineJob);
        engineJob.addCallback(cb);
        engineJob.start(runnable);

        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Started new load", startTime, key);
        }
        return new LoadStatus(cb, engineJob);
    }

    ...
}

```

`load()`方法中的代码虽然有点长，但大多数的代码都是在处理缓存的。关于缓存的内容我们会在下一篇文章当中学习，现在只需要从第 45 行看起就行。这里构建了一个 `EngineJob`，它的主要作用就是用来开启线程的，为后面的异步加载图片做准备。接下来第 46 行创建了一个 `DecodeJob` 对象，从名字上来看，它好像是用来对图片进行解码的，但实际上它的任务十分繁重，待会我们就知道了。继续往下看，第 48 行创建了一个 `EngineRunnable` 对象，并且在 51 行调用了 `EngineJob` 的 `start()` 方法来运行 `EngineRunnable` 对象，这实际上就是让 `EngineRunnable` 的 `run()` 方法在子线程当中执行了。那么我们现在就可以去看看 `EngineRunnable` 的 `run()` 方法里做了些什么，如下所示：

```

@Override
public void run() {
    if (isCancelled) {
        return;
    }
    Exception exception = null;
    Resource<?> resource = null;
    try {

```

```

        resource = decode();
    } catch (Exception e) {
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, "Exception decoding", e);
        }
        exception = e;
    }
    if (isCancelled) {
        if (resource != null) {
            resource.recycle();
        }
        return;
    }
    if (resource == null) {
        onLoadFailed(exception);
    } else {
        onLoadComplete(resource);
    }
}

```

这个方法中的代码并不多，但我们仍然还是要抓重点。在第 9 行，这里调用了一个 `decode()` 方法，并且这个方法返回了一个 `Resource` 对象。看上去所有的逻辑应该都在这个 `decode()` 方法执行的了，那我们跟进去瞧一瞧：

```

private Resource<?> decode() throws Exception {
    if (isDecodingFromCache()) {
        return decodeFromCache();
    } else {
        return decodeFromSource();
    }
}

```

`decode()` 方法中又分了两 种 情况，从缓存当中去 `decode` 图片的话就会执行 `decodeFromCache()`，否则的话就执行 `decodeFromSource()`。本篇文章中我们不讨论缓存的情况，那么就 直接来看 `decodeFromSource()` 方法的代码吧，如下所示：

```

private Resource<?> decodeFromSource() throws Exception {
    return decodeJob.decodeFromSource();
}

```

这里又调用了 `DecodeJob` 的 `decodeFromSource()` 方法。刚才已经说了，`DecodeJob` 的任务十分繁重，我们继续跟进看一看吧：

```

class DecodeJob<A, T, Z> {

```

```

    ...

```



```

public Resource<Z> decodeFromSource() throws Exception {
    Resource<T> decoded = decodeSource();
    return transformEncodeAndTranscode(decoded);
}

private Resource<T> decodeSource() throws Exception {
    Resource<T> decoded = null;
    try {
        long startTime = LogTime.getLogTime();
        final A data = fetcher.loadData(priority);
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Fetched data", startTime);
        }
        if (isCancelled) {
            return null;
        }
        decoded = decodeFromSourceData(data);
    } finally {
        fetcher.cleanup();
    }
    return decoded;
}

...
}

```

主要的方法就这些，我都帮大家提取出来了。那么我们先来看一下 `decodeFromSource()` 方法，其实它的工作分为两部，第一步是调用 `decodeSource()` 方法来获得一个 `Resource` 对象，第二步是调用 `transformEncodeAndTranscode()` 方法来处理这个 `Resource` 对象。

那么我们先来看第一步，`decodeSource()` 方法中的逻辑也并不复杂，首先在第 14 行调用了 `fetcher.loadData()` 方法。那么这个 `fetcher` 是什么呢？其实就是刚才在 `onSizeReady()` 方法中得到的 `ImageVideoFetcher` 对象，这里调用它的 `loadData()` 方法，代码如下所示：

```

@Override
public ImageVideoWrapper loadData(Priority priority) throws Exception {
    InputStream is = null;
    if (streamFetcher != null) {
        try {
            is = streamFetcher.loadData(priority);
        } catch (Exception e) {
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                Log.v(TAG, "Exception fetching input stream, trying ParcelFileDescriptor", e);
            }
            if (fileDescriptorFetcher == null) {

```

```

        throw e;
    }
}
ParcelFileDescriptor fileDescriptor = null;
if (fileDescriptorFetcher != null) {
    try {
        fileDescriptor = fileDescriptorFetcher.loadData(priority);
    } catch (Exception e) {
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, "Exception fetching ParcelFileDescriptor", e);
        }
        if (is == null) {
            throw e;
        }
    }
}
return new ImageVideoWrapper(is, fileDescriptor);
}

```

可以看到，在 `ImageVideoFetcher` 的 `loadData()` 方法的第 6 行，这里又去调用了 `streamFetcher.loadData()` 方法，那么这个 `streamFetcher` 是什么呢？自然就是刚才在组装 `ImageVideoFetcher` 对象时传进来的 `HttpUrlFetcher` 了。因此这里又会去调用 `HttpUrlFetcher` 的 `loadData()` 方法，那么我们继续跟进去瞧一瞧：

```

public class HttpUrlFetcher implements DataFetcher<InputStream> {

    ...

    @Override
    public InputStream loadData(Priority priority) throws Exception {
        return loadDataWithRedirects(glideUrl.toURL(), 0 /*redirects*/, null /*lastUrl*/,
        glideUrl.getHeaders());
    }

    private InputStream loadDataWithRedirects(URL url, int redirects, URL lastUrl, Map<String,
    String> headers)
        throws IOException {
        if (redirects >= MAXIMUM_REDIRECTS) {
            throw new IOException("Too many (> " + MAXIMUM_REDIRECTS + ") redirects!");
        } else {
            // Comparing the URLs using .equals performs additional network I/O and is
            generally broken.
            //

```

See

<http://michaelscharf.blogspot.com/2006/11/javaneturlequals-and-hashcode-make.html>.

```

        try {
            if (lastUrl != null && url.toURI().equals(lastUrl.toURI())) {
                throw new IOException("In re-direct loop");
            }
        } catch (URISyntaxException e) {
            // Do nothing, this is best effort.
        }
    }

    urlConnection = connectionFactory.build(url);
    for (Map.Entry<String, String> headerEntry : headers.entrySet()) {
        urlConnection.addRequestProperty(headerEntry.getKey(), headerEntry.getValue());
    }
    urlConnection.setConnectTimeout(2500);
    urlConnection.setReadTimeout(2500);
    urlConnection.setUseCaches(false);
    urlConnection.setDoInput(true);

    // Connect explicitly to avoid errors in decoders if connection fails.
    urlConnection.connect();
    if (isCancelled) {
        return null;
    }
    final int statusCode = urlConnection.getResponseCode();
    if (statusCode / 100 == 2) {
        return getStreamForSuccessfulRequest(urlConnection);
    } else if (statusCode / 100 == 3) {
        String redirectUrlString = urlConnection.getHeaderField("Location");
        if (TextUtils.isEmpty(redirectUrlString)) {
            throw new IOException("Received empty or null redirect url");
        }
        URL redirectUrl = new URL(url, redirectUrlString);
        return loadDataWithRedirects(redirectUrl, redirects + 1, url, headers);
    } else {
        if (statusCode == -1) {
            throw new IOException("Unable to retrieve response code from
HttpURLConnection.");
        }
        throw new IOException("Request failed " + statusCode + ": " +
urlConnection.getResponseMessage());
    }
}

private InputStream getStreamForSuccessfulRequest(HttpURLConnection urlConnection)
    throws IOException {

```

```

        if (TextUtils.isEmpty(urlConnection.getContentEncoding())) {
            int contentLength = urlConnection.getContentLength();
            stream = ContentLengthInputStream.obtain(urlConnection.getInputStream(),
contentLength);
        } else {
            if (Log.isLoggable(TAG, Log.DEBUG)) {
                Log.d(TAG, "Got non empty content encoding: " +
urlConnection.getContentEncoding());
            }
            stream = urlConnection.getInputStream();
        }
        return stream;
    }

    ...
}

```

经过一层一层地跋山涉水，我们终于在这里找到网络通讯的代码了！之前有朋友跟我讲过，说 **Glide** 的源码实在是太复杂了，甚至连网络请求是在哪里发出去的都找不到。我们也是经过一段一段又一段的代码跟踪，终于把网络请求的代码给找出来了，实在是太不容易了。

不过也别高兴得太早，现在离最终分析完还早着呢。可以看到，**loadData()**方法只是返回了一个 **InputStream**，服务器返回的数据连读都还没开始读呢。所以我们还是要静下心来继续分析，回到刚才 **ImageVideoFetcher** 的 **loadData()**方法中，在这个方法的最后一行，创建了一个 **ImageVideoWrapper** 对象，并把刚才得到的 **InputStream** 作为参数传了进去。

然后我们回到再上一层，也就是 **DecodeJob** 的 **decodeSource()**方法当中，在得到了这个 **ImageVideoWrapper** 对象之后，紧接着又将这个对象传入到了 **decodeFromSourceData()**当中，来去解码这个对象。**decodeFromSourceData()**方法的代码如下所示：

```

private Resource<T> decodeFromSourceData(A data) throws IOException {
    final Resource<T> decoded;
    if (diskCacheStrategy.cacheSource()) {
        decoded = cacheAndDecodeSourceData(data);
    } else {
        long startTime = LogTime.getLogTime();
        decoded = loadProvider.getSourceDecoder().decode(data, width, height);
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Decoded from source", startTime);
        }
    }
    return decoded;
}

```

可以看到，这里在第 7 行调用了 **loadProvider.getSourceDecoder().decode()**方法来进行解码。**loadProvider** 就是刚才在 **onSizeReady()**方法中得到的 **FixedLoadProvider**，而 **getSourceDecoder()**

得到的则是一个 GifBitmapWrapperResourceDecoder 对象,也就是要调用这个对象的 decode() 方法来对图片进行解码。那么我们来看下 GifBitmapWrapperResourceDecoder 的代码:

```
public class GifBitmapWrapperResourceDecoder implements
ResourceDecoder<ImageVideoWrapper, GifBitmapWrapper> {

    ...

    @SuppressWarnings("resource")
    // @see ResourceDecoder.decode
    @Override
    public Resource<GifBitmapWrapper> decode(ImageVideoWrapper source, int width, int
height) throws IOException {
        ByteArrayPool pool = ByteArrayPool.get();
        byte[] tempBytes = pool.getBytes();
        GifBitmapWrapper wrapper = null;
        try {
            wrapper = decode(source, width, height, tempBytes);
        } finally {
            pool.releaseBytes(tempBytes);
        }
        return wrapper != null ? new GifBitmapWrapperResource(wrapper) : null;
    }

    private GifBitmapWrapper decode(ImageVideoWrapper source, int width, int height, byte[]
bytes) throws IOException {
        final GifBitmapWrapper result;
        if (source.getInputStream() != null) {
            result = decodeStream(source, width, height, bytes);
        } else {
            result = decodeBitmapWrapper(source, width, height);
        }
        return result;
    }

    private GifBitmapWrapper decodeStream(ImageVideoWrapper source, int width, int height,
byte[] bytes)
        throws IOException {
        InputStream bis = streamFactory.build(source.getInputStream(), bytes);
        bis.mark(MARK_LIMIT_BYTES);
        ImageHeaderParser.ImageType type = parser.parse(bis);
        bis.reset();
        GifBitmapWrapper result = null;
        if (type == ImageHeaderParser.ImageType.GIF) {
```

```

        result = decodeGifWrapper(bis, width, height);
    }
    // Decoding the gif may fail even if the type matches.
    if (result == null) {
        // We can only reset the buffered InputStream, so to start from the beginning of
the stream, we need to
        // pass in a new source containing the buffered stream rather than the original
stream.
        ImageVideoWrapper forBitmapDecoder = new ImageVideoWrapper(bis,
source.getFileDescriptor());
        result = decodeBitmapWrapper(forBitmapDecoder, width, height);
    }
    return result;
}

private GifBitmapWrapper decodeBitmapWrapper(ImageVideoWrapper toDecode, int width,
int height) throws IOException {
    GifBitmapWrapper result = null;
    Resource<Bitmap> bitmapResource = bitmapDecoder.decode(toDecode, width, height);
    if (bitmapResource != null) {
        result = new GifBitmapWrapper(bitmapResource, null);
    }
    return result;
}

...
}

```

首先，在 `decode()` 方法中，又去调用了另外一个 `decode()` 方法的重载。然后在第 23 行调用了 `decodeStream()` 方法，准备从服务器返回的流当中读取数据。`decodeStream()` 方法中会先从流中读取 2 个字节的数据，来判断这张图是 GIF 图还是普通的静图，如果是 GIF 图就调用 `decodeGifWrapper()` 方法来进行解码，如果是普通的静图就用调用 `decodeBitmapWrapper()` 方法来进行解码。这里我们只分析普通静图的实现流程，GIF 图的实现有点过于复杂了，无法在本篇文章当中分析。

然后我们来看一下 `decodeBitmapWrapper()` 方法，这里在第 52 行调用了 `bitmapDecoder.decode()` 方法。这个 `bitmapDecoder` 是一个 `ImageVideoBitmapDecoder` 对象，那么我们来看一下它的代码，如下所示：

```

public class ImageVideoBitmapDecoder implements ResourceDecoder<ImageVideoWrapper,
Bitmap> {
    private final ResourceDecoder<InputStream, Bitmap> streamDecoder;
    private final ResourceDecoder<ParcelFileDescriptor, Bitmap> fileDescriptorDecoder;

    public ImageVideoBitmapDecoder(ResourceDecoder<InputStream, Bitmap> streamDecoder,

```

```

        ResourceDecoder<ParcelFileDescriptor, Bitmap> fileDescriptorDecoder) {
    this.streamDecoder = streamDecoder;
    this.fileDescriptorDecoder = fileDescriptorDecoder;
}

@Override
public Resource<Bitmap> decode(ImageVideoWrapper source, int width, int height) throws
IOException {
    Resource<Bitmap> result = null;
    InputStream is = source.getInputStream();
    if (is != null) {
        try {
            result = streamDecoder.decode(is, width, height);
        } catch (IOException e) {
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                Log.v(TAG, "Failed to load image from stream, trying FileDescriptor", e);
            }
        }
    }
    if (result == null) {
        ParcelFileDescriptor fileDescriptor = source.getFileDescriptor();
        if (fileDescriptor != null) {
            result = fileDescriptorDecoder.decode(fileDescriptor, width, height);
        }
    }
    return result;
}

...
}

```

代码并不复杂，在第 14 行先调用了 `source.getInputStream()` 来获取到服务器返回的 `InputStream`，然后在第 17 行调用 `streamDecoder.decode()` 方法进行解码。`streamDecode` 是一个 `StreamBitmapDecoder` 对象，那么我们再来看这个类的源码，如下所示：

```

public class StreamBitmapDecoder implements ResourceDecoder<InputStream, Bitmap> {

    ...

    private final Downsampler downsampler;
    private BitmapPool bitmapPool;
    private DecodeFormat decodeFormat;

    public StreamBitmapDecoder(Downsampler downsampler, BitmapPool bitmapPool,
        DecodeFormat decodeFormat) {

```

```

        this.downsampler = downsampler;
        this.bitmapPool = bitmapPool;
        this.decodeFormat = decodeFormat;
    }

    @Override
    public Resource<Bitmap> decode(InputStream source, int width, int height) {
        Bitmap bitmap = downsampler.decode(source, bitmapPool, width, height,
decodeFormat);
        return BitmapResource.obtain(bitmap, bitmapPool);
    }

    ...
}

```

可以看到，它的 `decode()` 方法又去调用了 `Downsampler` 的 `decode()` 方法。接下来又到了激动人心的时刻了，`Downsampler` 的代码如下所示：

```

public abstract class Downsampler implements BitmapDecoder<InputStream> {

    ...

    @Override
    public Bitmap decode(InputStream is, BitmapPool pool, int outWidth, int outHeight,
DecodeFormat decodeFormat) {
        final ByteArrayPool byteArrayPool = ByteArrayPool.get();
        final byte[] bytesForOptions = byteArrayPool.getBytes();
        final byte[] bytesForStream = byteArrayPool.getBytes();
        final BitmapFactory.Options options = getDefaultOptions();
        // Use to fix the mark limit to avoid allocating buffers that fit entire images.
        RecyclableBufferedInputStream bufferedStream = new RecyclableBufferedInputStream(
            is, bytesForStream);
        // Use to retrieve exceptions thrown while reading.
        // TODO(#126): when the framework no longer returns partially decoded Bitmaps or
provides a way to determine
        // if a Bitmap is partially decoded, consider removing.
        ExceptionCatchingInputStream exceptionStream =
            ExceptionCatchingInputStream.obtain(bufferedStream);
        // Use to read data.
        // Ensures that we can always reset after reading an image header so that we can still
attempt to decode the
        // full image even when the header decode fails and/or overflows our read buffer. See
#283.
        MarkEnforcingInputStream invalidatingStream = new
MarkEnforcingInputStream(exceptionStream);
    }
}

```



```

try {
    exceptionStream.mark(MARK_POSITION);
    int orientation = 0;
    try {
        orientation = new ImageHeaderParser(exceptionStream).getOrientation();
    } catch (IOException e) {
        if (Log.isLoggable(TAG, Log.WARN)) {
            Log.w(TAG, "Cannot determine the image orientation from header", e);
        }
    } finally {
        try {
            exceptionStream.reset();
        } catch (IOException e) {
            if (Log.isLoggable(TAG, Log.WARN)) {
                Log.w(TAG, "Cannot reset the input stream", e);
            }
        }
    }
    options.inTempStorage = bytesForOptions;
    final int[] inDimens = getDimensions(invalidatingStream, bufferedStream, options);
    final int inWidth = inDimens[0];
    final int inHeight = inDimens[1];
    final int degreesToRotate =
TransformationUtils.getExifOrientationDegrees(orientation);
    final int sampleSize = getRoundedSampleSize(degreesToRotate, inWidth, inHeight,
outWidth, outHeight);
    final Bitmap downsampled =
        downsampleWithSize(invalidatingStream, bufferedStream, options, pool,
inWidth, inHeight, sampleSize,
        decodeFormat);
    // BitmapFactory swallows exceptions during decodes and in some cases when
inBitmap is non null, may catch
    // and log a stack trace but still return a non null bitmap. To avoid displaying
partially decoded bitmaps,
    // we catch exceptions reading from the stream in our
ExceptionCatchingInputStream and throw them here.
    final Exception streamException = exceptionStream.getException();
    if (streamException != null) {
        throw new RuntimeException(streamException);
    }
    Bitmap rotated = null;
    if (downsampled != null) {
        rotated = TransformationUtils.rotateImageExif(downsampled, pool,
orientation);

```

```

        if (!downsampled.equals(rotated) && !pool.put(downsampled)) {
            downsampled.recycle();
        }
    }
    return rotated;
} finally {
    byteArrayPool.releaseBytes(bytesForOptions);
    byteArrayPool.releaseBytes(bytesForStream);
    exceptionStream.release();
    releaseOptions(options);
}
}

private Bitmap downsampleWithSize(MarkEnforcingInputStream is,
    RecyclableBufferedInputStream bufferedStream,
    BitmapFactory.Options options, BitmapPool pool, int inWidth, int inHeight, int
sampleSize,
    DecodeFormat decodeFormat) {
    // Prior to KitKat, the inBitmap size must exactly match the size of the bitmap we're
    decoding.
    Bitmap.Config config = getConfig(is, decodeFormat);
    options.inSampleSize = sampleSize;
    options.inPreferredConfig = config;
    if ((options.inSampleSize == 1 || Build.VERSION_CODES.KITKAT <=
Build.VERSION.SDK_INT) && shouldUsePool(is)) {
        int targetWidth = (int) Math.ceil(inWidth / (double) sampleSize);
        int targetHeight = (int) Math.ceil(inHeight / (double) sampleSize);
        // BitmapFactory will clear out the Bitmap before writing to it, so getDirty is safe.
        setInBitmap(options, pool.getDirty(targetWidth, targetHeight, config));
    }
    return decodeStream(is, bufferedStream, options);
}

/**
 * A method for getting the dimensions of an image from the given InputStream.
 *
 * @param is The InputStream representing the image.
 * @param options The options to pass to
 *             {@link BitmapFactory#decodeStream(InputStream, android.graphics.Rect,
 *             BitmapFactory.Options)}.
 * @return an array containing the dimensions of the image in the form {width, height}.
 */
public int[] getDimensions(MarkEnforcingInputStream is, RecyclableBufferedInputStream
bufferedStream,

```

```

        BitmapFactory.Options options) {
    options.inJustDecodeBounds = true;
    decodeStream(is, bufferedStream, options);
    options.inJustDecodeBounds = false;
    return new int[] { options.outWidth, options.outHeight };
}

private static Bitmap decodeStream(MarkEnforcingInputStream is,
    RecyclableBufferedInputStream bufferedStream,
    BitmapFactory.Options options) {
    if (options.inJustDecodeBounds) {
        // This is large, but jpeg headers are not size bounded so we need something
        // large enough to minimize
        // the possibility of not being able to fit enough of the header in the buffer to get
        // the image size so
        // that we don't fail to load images. The BufferedInputStream will create a new
        // buffer of 2x the
        // original size each time we use up the buffer space without passing the mark so
        // this is a maximum
        // bound on the buffer size, not a default. Most of the time we won't go past our
        // pre-allocated 16kb.
        is.mark(MARK_POSITION);
    } else {
        // Once we've read the image header, we no longer need to allow the buffer to
        // expand in size. To avoid
        // unnecessary allocations reading image data, we fix the mark limit so that it is
        // no larger than our
        // current buffer size here. See issue #225.
        bufferedStream.fixMarkLimit();
    }
    final Bitmap result = BitmapFactory.decodeStream(is, null, options);
    try {
        if (options.inJustDecodeBounds) {
            is.reset();
        }
    } catch (IOException e) {
        if (Log.isLoggable(TAG, Log.ERROR)) {
            Log.e(TAG, "Exception loading inDecodeBounds=" +
options.inJustDecodeBounds
                + " sample=" + options.inSampleSize, e);
        }
    }

    return result;
}

```

```

    }

    ...

}

```

可以看到，对服务器返回的 `InputStream` 的读取，以及对图片的加载全都在这里了。当然这里其实处理了很多的逻辑，包括对图片的压缩，甚至还有旋转、圆角等逻辑处理，但是我们目前只需要关注主线逻辑就行了。`decode()`方法执行之后，会返回一个 `Bitmap` 对象，那么图片在这里其实也已经被加载出来了，剩下的工作就是如果让这个 `Bitmap` 显示到界面上，我们继续往下分析。

回到刚才的 `StreamBitmapDecoder` 当中，你会发现，它的 `decode()` 方法返回的是一个 `Resource<Bitmap>` 对象。而我们从 `Downsampler` 中得到的是一个 `Bitmap` 对象，因此这里在第 18 行又调用了 `BitmapResource.obtain()` 方法，将 `Bitmap` 对象包装成了 `Resource<Bitmap>` 对象。代码如下所示：

```

public class BitmapResource implements Resource<Bitmap> {
    private final Bitmap bitmap;
    private final BitmapPool bitmapPool;

    /**
     * Returns a new {@link BitmapResource} wrapping the given {@link Bitmap} if the Bitmap
     * is non-null or null if the
     *   * given Bitmap is null.
     *   *
     *   * @param bitmap A Bitmap.
     *   * @param bitmapPool A non-null {@link BitmapPool}.
     */
    public static BitmapResource obtain(Bitmap bitmap, BitmapPool bitmapPool) {
        if (bitmap == null) {
            return null;
        } else {
            return new BitmapResource(bitmap, bitmapPool);
        }
    }

    public BitmapResource(Bitmap bitmap, BitmapPool bitmapPool) {
        if (bitmap == null) {
            throw new NullPointerException("Bitmap must not be null");
        }
        if (bitmapPool == null) {
            throw new NullPointerException("BitmapPool must not be null");
        }
        this.bitmap = bitmap;
        this.bitmapPool = bitmapPool;
    }
}

```

```

    }

    @Override
    public Bitmap get() {
        return bitmap;
    }

    @Override
    public int getSize() {
        return Util.getBitmapByteSize(bitmap);
    }

    @Override
    public void recycle() {
        if (!bitmapPool.put(bitmap)) {
            bitmap.recycle();
        }
    }
}

```

BitmapResource 的源码也非常简单，经过这样一层包装之后，如果我还需要获取 Bitmap，只需要调用 Resource<Bitmap>的 get()方法就可以了。

然后我们需要一层层继续向上返回，StreamBitmapDecoder 会将值返回到 ImageVideoBitmapDecoder 当中，而 ImageVideoBitmapDecoder 又会将值返回到 GifBitmapWrapperResourceDecoder 的 decodeBitmapWrapper()方法当中。由于代码隔得有点太远了，我重新把 decodeBitmapWrapper()方法的代码贴一下：

```

private GifBitmapWrapper decodeBitmapWrapper(ImageVideoWrapper toDecode, int width, int height) throws IOException {
    GifBitmapWrapper result = null;
    Resource<Bitmap> bitmapResource = bitmapDecoder.decode(toDecode, width, height);
    if (bitmapResource != null) {
        result = new GifBitmapWrapper(bitmapResource, null);
    }
    return result;
}

```

可以看到，decodeBitmapWrapper()方法返回的是一个 GifBitmapWrapper 对象。因此，这里在第 5 行，又将 Resource<Bitmap>封装到了一个 GifBitmapWrapper 对象当中。这个 GifBitmapWrapper 顾名思义，就是既能封装 GIF，又能封装 Bitmap，从而保证了不管是什么类型的图片 Glide 都能从容应对。我们顺便来看下 GifBitmapWrapper 的源码吧，如下所示：

```

public class GifBitmapWrapper {
    private final Resource<GifDrawable> gifResource;
    private final Resource<Bitmap> bitmapResource;
}

```

```

    public GifBitmapWrapper(Resource<Bitmap> bitmapResource, Resource<GifDrawable>
gifResource) {
        if (bitmapResource != null && gifResource != null) {
            throw new IllegalArgumentException("Can only contain either a bitmap resource
or a gif resource, not both");
        }
        if (bitmapResource == null && gifResource == null) {
            throw new IllegalArgumentException("Must contain either a bitmap resource or a
gif resource");
        }
        this.bitmapResource = bitmapResource;
        this.gifResource = gifResource;
    }

    /**
     * Returns the size of the wrapped resource.
     */
    public int getSize() {
        if (bitmapResource != null) {
            return bitmapResource.getSize();
        } else {
            return gifResource.getSize();
        }
    }

    /**
     * Returns the wrapped {@link Bitmap} resource if it exists, or null.
     */
    public Resource<Bitmap> getBitmapResource() {
        return bitmapResource;
    }

    /**
     * Returns the wrapped {@link GifDrawable} resource if it exists, or null.
     */
    public Resource<GifDrawable> getGifResource() {
        return gifResource;
    }
}

```

还是比较简单的，就是分别对 `gifResource` 和 `bitmapResource` 做了一层封装而已，相信没有什么解释的必要。

然后这个 `GifBitmapWrapper` 对象会一直向上返回，返回到 `GifBitmapWrapperResourceDecoder`

最外层的 `decode()` 方法的时候，会对它再做一次封装，如下所示：

```
@Override
public Resource<GifBitmapWrapper> decode(ImageVideoWrapper source, int width, int height)
throws IOException {
    ByteArrayPool pool = ByteArrayPool.get();
    byte[] tempBytes = pool.getBytes();
    GifBitmapWrapper wrapper = null;
    try {
        wrapper = decode(source, width, height, tempBytes);
    } finally {
        pool.releaseBytes(tempBytes);
    }
    return wrapper != null ? new GifBitmapWrapperResource(wrapper) : null;
}
```

可以看到，这里在第 11 行，又将 `GifBitmapWrapper` 封装到了一个 `GifBitmapWrapperResource` 对象当中，最终返回的是一个 `Resource<GifBitmapWrapper>` 对象。这个 `GifBitmapWrapperResource` 和刚才的 `BitmapResource` 是相似的，它们都实现的 `Resource` 接口，都可以通过 `get()` 方法来获取封装起来的具体内容。`GifBitmapWrapperResource` 的源码如下所示：

```
public class GifBitmapWrapperResource implements Resource<GifBitmapWrapper> {
    private final GifBitmapWrapper data;

    public GifBitmapWrapperResource(GifBitmapWrapper data) {
        if (data == null) {
            throw new NullPointerException("Data must not be null");
        }
        this.data = data;
    }

    @Override
    public GifBitmapWrapper get() {
        return data;
    }

    @Override
    public int getSize() {
        return data.getSize();
    }

    @Override
    public void recycle() {
        Resource<Bitmap> bitmapResource = data.getBitmapResource();
    }
}
```

```

        if (bitmapResource != null) {
            bitmapResource.recycle();
        }
        Resource<GifDrawable> gifDataResource = data.getGifResource();
        if (gifDataResource != null) {
            gifDataResource.recycle();
        }
    }
}

```

经过这一层的封装之后，我们从网络上得到的图片就能够以 `Resource` 接口的形式返回，并且还能同时处理 `Bitmap` 图片和 `GIF` 图片这两种情况。

那么现在我们可以回到 `DecodeJob` 当中了，它的 `decodeFromSourceData()` 方法返回的是一个 `Resource<T>` 对象，其实也就是 `Resource<GifBitmapWrapper>` 对象了。然后继续向上返回，最终返回到 `decodeFromSource()` 方法当中，如下所示：

```

public Resource<Z> decodeFromSource() throws Exception {
    Resource<T> decoded = decodeSource();
    return transformEncodeAndTranscode(decoded);
}

```

刚才我们就是从这里跟进到 `decodeSource()` 方法当中，然后执行了一大堆一大堆的逻辑，最终得到了这个 `Resource<T>` 对象。然而你会发现，`decodeFromSource()` 方法最终返回的却是一个 `Resource<Z>` 对象，那么这到底是怎么回事呢？我们就需要跟进到 `transformEncodeAndTranscode()` 方法来瞧一瞧了，代码如下所示：

```

private Resource<Z> transformEncodeAndTranscode(Resource<T> decoded) {
    long startTime = LogTime.getLogTime();
    Resource<T> transformed = transform(decoded);
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logWithTimeAndKey("Transformed resource from source", startTime);
    }
    writeTransformedToCache(transformed);
    startTime = LogTime.getLogTime();
    Resource<Z> result = transcode(transformed);
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logWithTimeAndKey("Transcoded transformed from source", startTime);
    }
    return result;
}

private Resource<Z> transcode(Resource<T> transformed) {
    if (transformed == null) {
        return null;
    }
}

```



```

        return transcoder.transcode(transformed);
    }

```

首先，这个方法开头的几行 `transform` 还有 `cache`，这都是我们后面才会学习的东西，现在不用管它们就可以了。需要注意的是第 9 行，这里调用了一个 `transcode()` 方法，就把 `Resource<T>` 对象转换成 `Resource<Z>` 对象了。

而 `transcode()` 方法中又是调用了 `transcoder` 的 `transcode()` 方法，那么这个 `transcoder` 是什么呢？其实这也是 `Glide` 源码特别难懂的原因之一，就是它用到的很多对象都是很早很早之前就初始化的，在初始化的时候你可能完全就没有留意过它，因为一时半会根本就用不着，但是真正需要用到时候你却早就记不起来这个对象是从哪儿来的了。

那么这里我来提醒一下大家吧，在第二步 `load()` 方法返回的那个 `DrawableTypeRequest` 对象，它的构造函数中去构建了一个 `FixedLoadProvider` 对象，然后将三个参数传入到了 `FixedLoadProvider` 当中，其中就有一个 `GifBitmapWrapperDrawableTranscoder` 对象。后来在 `onSizeReady()` 方法中获取到了这个参数，并传递到了 `Engine` 当中，然后又由 `Engine` 传递到了 `DecodeJob` 当中。因此，这里的 `transcoder` 其实就是这个 `GifBitmapWrapperDrawableTranscoder` 对象。那么我们来看一下它的源码：

```

public class GifBitmapWrapperDrawableTranscoder implements
ResourceTranscoder<GifBitmapWrapper, GlideDrawable> {
    private final ResourceTranscoder<Bitmap, GlideBitmapDrawable>
bitmapDrawableResourceTranscoder;

    public GifBitmapWrapperDrawableTranscoder(
        ResourceTranscoder<Bitmap, GlideBitmapDrawable>
bitmapDrawableResourceTranscoder) {
        this.bitmapDrawableResourceTranscoder = bitmapDrawableResourceTranscoder;
    }

    @Override
    public Resource<GlideDrawable> transcode(Resource<GifBitmapWrapper> toTranscode) {
        GifBitmapWrapper gifBitmap = toTranscode.get();
        Resource<Bitmap> bitmapResource = gifBitmap.getBitmapResource();
        final Resource<? extends GlideDrawable> result;
        if (bitmapResource != null) {
            result = bitmapDrawableResourceTranscoder.transcode(bitmapResource);
        } else {
            result = gifBitmap.getGifResource();
        }
        return (Resource<GlideDrawable>) result;
    }

    ...
}

```

这里我来简单解释一下，GifBitmapWrapperDrawableTranscoder 的核心作用就是用来转码的。因为 GifBitmapWrapper 是无法直接显示到 ImageView 上面的，只有 Bitmap 或者 Drawable 才能显示到 ImageView 上。因此，这里的 transcode()方法先从 Resource<GifBitmapWrapper>中取出 GifBitmapWrapper 对象，然后再从 GifBitmapWrapper 中取出 Resource<Bitmap>对象。

接下来做了一个判断，如果 Resource<Bitmap>为空，那么说明此时加载的是 GIF 图，直接调用 getGifResource()方法将图片取出即可，因为 Glide 用于加载 GIF 图片是使用的 GifDrawable 这个类，它本身就是一个 Drawable 对象了。而如果 Resource<Bitmap>不为空，那么就需要再做一次转码，将 Bitmap 转换成 Drawable 对象才行，因为要保证静图和动图的类型一致性，不然逻辑上是不好处理的。

这里在第 15 行又进行了一次转码，是调用的 GlideBitmapDrawableTranscoder 对象的 transcode()方法，代码如下所示：

```
public class GlideBitmapDrawableTranscoder implements ResourceTranscoder<Bitmap,
GlideBitmapDrawable> {
    private final Resources resources;
    private final BitmapPool bitmapPool;

    public GlideBitmapDrawableTranscoder(Context context) {
        this(context.getResources(), Glide.get(context).getBitmapPool());
    }

    public GlideBitmapDrawableTranscoder(Resources resources, BitmapPool bitmapPool) {
        this.resources = resources;
        this.bitmapPool = bitmapPool;
    }

    @Override
    public Resource<GlideBitmapDrawable> transcode(Resource<Bitmap> toTranscode) {
        GlideBitmapDrawable drawable = new GlideBitmapDrawable(resources,
toTranscode.get());
        return new GlideBitmapDrawableResource(drawable, bitmapPool);
    }

    ...
}
```

可以看到，这里 new 出了一个 GlideBitmapDrawable 对象，并把 Bitmap 封装到里面。然后对 GlideBitmapDrawable 再进行一次封装，返回一个 Resource<GlideBitmapDrawable>对象。

现在再返回到 GifBitmapWrapperDrawableTranscoder 的 transcode()方法中，你会发现它们的类型就一致了。因为不管是静图的 Resource<GlideBitmapDrawable>对象，还是动图的 Resource<GifDrawable>对象，它们都是属于父类 Resource<GlideDrawable>对象的。因此 transcode()方法也是直接返回了 Resource<GlideDrawable>，而这个 Resource<GlideDrawable>

其实也就是转换过后的 `Resource<Z>` 了。

那么我们继续回到 `DecodeJob` 当中，它的 `decodeFromSource()` 方法得到了 `Resource<Z>` 对象，当然也就是 `Resource<GlideDrawable>` 对象。然后继续向上返回会回到 `EngineRunnable` 的 `decodeFromSource()` 方法，再回到 `decode()` 方法，再回到 `run()` 方法当中。那么我们重新再贴一下 `EngineRunnable run()` 方法的源码：

```
@Override
public void run() {
    if (isCancelled) {
        return;
    }
    Exception exception = null;
    Resource<?> resource = null;
    try {
        resource = decode();
    } catch (Exception e) {
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, "Exception decoding", e);
        }
        exception = e;
    }
    if (isCancelled) {
        if (resource != null) {
            resource.recycle();
        }
        return;
    }
    if (resource == null) {
        onLoadFailed(exception);
    } else {
        onLoadComplete(resource);
    }
}
```

也就是说，经过第 9 行 `decode()` 方法的执行，我们最终得到了这个 `Resource<GlideDrawable>` 对象，那么接下来就是如何将它显示出来了。可以看到，这里在第 25 行调用了 `onLoadComplete()` 方法，表示图片加载已经完成了，代码如下所示：

```
private void onLoadComplete(Resource resource) {
    manager.onResourceReady(resource);
}
```

这个 `manager` 就是 `EngineJob` 对象，因此这里实际上调用的是 `EngineJob` 的 `onResourceReady()` 方法，代码如下所示：

```

class EngineJob implements EngineRunnable.EngineRunnableManager {

    private static final Handler MAIN_THREAD_HANDLER = new
Handler(Looper.getMainLooper(), new MainThreadCallback());

    private final List<ResourceCallback> cbs = new ArrayList<ResourceCallback>();

    ...

    public void addCallback(ResourceCallback cb) {
        Util.assertMainThread();
        if (hasResource) {
            cb.onResourceReady(engineResource);
        } else if (hasException) {
            cb.onException(exception);
        } else {
            cbs.add(cb);
        }
    }

    @Override
    public void onResourceReady(final Resource<?> resource) {
        this.resource = resource;
        MAIN_THREAD_HANDLER.obtainMessage(MSG_COMPLETE, this).sendToTarget();
    }

    private void handleResultOnMainThread() {
        if (isCancelled) {
            resource.recycle();
            return;
        } else if (cbs.isEmpty()) {
            throw new IllegalStateException("Received a resource without any callbacks to
notify");
        }
        engineResource = engineResourceFactory.build(resource, isCacheable);
        hasResource = true;
        engineResource.acquire();
        listener.onEngineJobComplete(key, engineResource);
        for (ResourceCallback cb : cbs) {
            if (!isIgnoredCallbacks(cb)) {
                engineResource.acquire();
                cb.onResourceReady(engineResource);
            }
        }
    }
}

```

```

        engineResource.release();
    }

    @Override
    public void onException(final Exception e) {
        this.exception = e;
        MAIN_THREAD_HANDLER.obtainMessage(MSG_EXCEPTION, this).sendToTarget();
    }

    private void handleExceptionOnMainThread() {
        if (isCancelled) {
            return;
        } else if (cbs.isEmpty()) {
            throw new IllegalStateException("Received an exception without any callbacks to
notify");
        }
        hasException = true;
        listener.onEngineJobComplete(key, null);
        for (ResourceCallback cb : cbs) {
            if (!isIgnoredCallbacks(cb)) {
                cb.onException(exception);
            }
        }
    }

    private static class MainThreadCallback implements Handler.Callback {

        @Override
        public boolean handleMessage(Message message) {
            if (MSG_COMPLETE == message.what || MSG_EXCEPTION == message.what) {
                EngineJob job = (EngineJob) message.obj;
                if (MSG_COMPLETE == message.what) {
                    job.handleResultOnMainThread();
                } else {
                    job.handleExceptionOnMainThread();
                }
                return true;
            }
            return false;
        }
    }

    ...
}

```

可以看到，这里在 `onResourceReady()` 方法使用 `Handler` 发出了一条 `MSG_COMPLETE` 消息，那么在 `MainThreadCallback` 的 `handleMessage()` 方法中就会收到这条消息。从这里开始，所有的逻辑又回到主线程当中进行了，因为很快就需要更新 UI 了。

然后在第 72 行调用了 `handleResultOnMainThread()` 方法，这个方法中又通过一个循环，调用了所有 `ResourceCallback` 的 `onResourceReady()` 方法。那么这个 `ResourceCallback` 是什么呢？答案在 `addCallback()` 方法当中，它会向 `cbs` 集合中去添加 `ResourceCallback`。那么这个 `addCallback()` 方法又是哪里调用的呢？其实调用的地方我们早就已经看过了，只不过之前没有注意，现在重新来看一下 `Engine` 的 `load()` 方法，如下所示：

```
public class Engine implements EngineJobListener,
    MemoryCache.ResourceRemovedListener,
    EngineResource.ResourceListener {

    ...

    public <T, Z, R> LoadStatus load(Key signature, int width, int height, DataFetcher<T> fetcher,
        DataLoadProvider<T, Z> loadProvider, Transformation<Z> transformation,
        ResourceTranscoder<Z, R> transcoder, Priority priority,
        boolean isMemoryCacheable, DiskCacheStrategy diskCacheStrategy,
        ResourceCallback cb) {

        ...

        EngineJob engineJob = engineJobFactory.build(key, isMemoryCacheable);
        DecodeJob<T, Z, R> decodeJob = new DecodeJob<T, Z, R>(key, width, height, fetcher,
        loadProvider, transformation,
            transcoder, diskCacheProvider, diskCacheStrategy, priority);
        EngineRunnable runnable = new EngineRunnable(engineJob, decodeJob, priority);
        jobs.put(key, engineJob);
        engineJob.addCallback(cb);
        engineJob.start(runnable);

        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Started new load", startTime, key);
        }
        return new LoadStatus(cb, engineJob);
    }

    ...
}
```

这次把目光放在第 18 行上面，看到了吗？就是在这里调用的 `EngineJob` 的 `addCallback()` 方法来注册的一个 `ResourceCallback`。那么接下来的问题就是，`Engine.load()` 方法的 `ResourceCallback` 参数又是谁传过来的呢？这就需要回到 `GenericRequest` 的 `onSizeReady()` 方

法当中了，我们看到 ResourceCallback 是 load()方法的最后一个参数，那么在 onSizeReady()方法中调用 load()方法时传入的最后一个参数是什么？代码如下所示：

```
public final class GenericRequest<A, T, Z, R> implements Request, SizeReadyCallback,
    ResourceCallback {

    ...

    @Override
    public void onSizeReady(int width, int height) {
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logV("Got onSizeReady in " + LogTime.getElapsedMillis(startTime));
        }
        if (status != Status.WAITING_FOR_SIZE) {
            return;
        }
        status = Status.RUNNING;
        width = Math.round(sizeMultiplier * width);
        height = Math.round(sizeMultiplier * height);
        ModelLoader<A, T> modelLoader = loadProvider.getModelLoader();
        final DataFetcher<T> dataFetcher = modelLoader.getResourceFetcher(model, width,
height);
        if (dataFetcher == null) {
            onException(new Exception("Failed to load model: \" + model + "\"));
            return;
        }
        ResourceTranscoder<Z, R> transcoder = loadProvider.getTranscoder();
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logV("finished setup for calling load in " + LogTime.getElapsedMillis(startTime));
        }
        loadedFromMemoryCache = true;
        loadStatus = engine.load(signature, width, height, dataFetcher, loadProvider,
transformation,
            transcoder, priority, isMemoryCacheable, diskCacheStrategy, this);
        loadedFromMemoryCache = resource != null;
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logV("finished onSizeReady in " + LogTime.getElapsedMillis(startTime));
        }
    }

    ...
}
```

请将目光锁定在第 29 行的最后一个参数，this。没错，就是 this。GenericRequest 本身就实现了 ResourceCallback 的接口，因此 EngineJob 的回调最终其实就是回调到了 GenericRequest

的 `onResourceReady()`方法当中了，代码如下所示：

```
public void onResourceReady(Resource<?> resource) {
    if (resource == null) {
        onException(new Exception("Expected to receive a Resource<R> with an object of " +
transcodeClass
        + " inside, but instead got null.));
        return;
    }
    Object received = resource.get();
    if (received == null || !transcodeClass.isAssignableFrom(received.getClass())) {
        releaseResource(resource);
        onException(new Exception("Expected to receive an object of " + transcodeClass
        + " but instead got " + (received != null ? received.getClass() : "") + "{" +
received + "}"
        + " inside Resource{" + resource + "}. "
        + (received != null ? "" : " "
        + "To indicate failure return a null Resource object, "
        + "rather than a Resource object containing null data.")
        ));
        return;
    }
    if (!canSetResource()) {
        releaseResource(resource);
        // We can't set the status to complete before asking canSetResource().
        status = Status.COMPLETE;
        return;
    }
    onResourceReady(resource, (R) received);
}

private void onResourceReady(Resource<?> resource, R result) {
    // We must call isFirstReadyResource before setting status.
    boolean isFirstResource = isFirstReadyResource();
    status = Status.COMPLETE;
    this.resource = resource;
    if (requestListener == null || !requestListener.onResourceReady(result, model, target,
loadedFromMemoryCache,
        isFirstResource)) {
        GlideAnimation<R> animation = animationFactory.build(loadedFromMemoryCache,
isFirstResource);
        target.onResourceReady(result, animation);
    }
    notifyLoadSuccess();
}
```



```

        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logV("Resource ready in " + LogTime.getElapsedMillis(startTime) + " size: "
                + (resource.getSize() * TO_MEGABYTE) + " fromCache: " +
loadedFromMemoryCache);
        }
    }
}

```

这里有两个 `onResourceReady()` 方法，首先在第一个 `onResourceReady()` 方法当中，调用 `resource.get()` 方法获取到了封装的图片对象，也就是 `GlideBitmapDrawable` 对象，或者是 `GifDrawable` 对象。然后将这个值传入到了第二个 `onResourceReady()` 方法当中，并在第 36 行调用了 `target.onResourceReady()` 方法。

那么这个 `target` 又是什么呢？这个又需要向上翻很久了，在第三步 `into()` 方法的一开始，我们就分析了在 `into()` 方法的最后一行，调用了 `glide.buildImageViewTarget()` 方法来构建出一个 `Target`，而这个 `Target` 就是一个 `GlideDrawableImageViewTarget` 对象。

那么我们去查看 `GlideDrawableImageViewTarget` 的源码就可以了，如下所示：

```

public class GlideDrawableImageViewTarget extends ImageViewTarget<GlideDrawable> {
    private static final float SQUARE_RATIO_MARGIN = 0.05f;
    private int maxLoopCount;
    private GlideDrawable resource;

    public GlideDrawableImageViewTarget(ImageView view) {
        this(view, GlideDrawable.LOOP_FOREVER);
    }

    public GlideDrawableImageViewTarget(ImageView view, int maxLoopCount) {
        super(view);
        this.maxLoopCount = maxLoopCount;
    }

    @Override
    public void onResourceReady(GlideDrawable resource, GlideAnimation<? super
GlideDrawable> animation) {
        if (!resource.isAnimated()) {
            float viewRatio = view.getWidth() / (float) view.getHeight();
            float drawableRatio = resource.getIntrinsicWidth() / (float)
resource.getIntrinsicHeight();
            if (Math.abs(viewRatio - 1f) <= SQUARE_RATIO_MARGIN
                && Math.abs(drawableRatio - 1f) <= SQUARE_RATIO_MARGIN) {
                resource = new SquaringDrawable(resource, view.getWidth());
            }
        }
        super.onResourceReady(resource, animation);
    }
}

```

```

        this.resource = resource;
        resource.setLoopCount(maxLoopCount);
        resource.start();
    }

    @Override
    protected void setResource(GlideDrawable resource) {
        view.setImageDrawable(resource);
    }

    @Override
    public void onStart() {
        if (resource != null) {
            resource.start();
        }
    }

    @Override
    public void onStop() {
        if (resource != null) {
            resource.stop();
        }
    }
}

```

在 `GlideDrawableImageViewTarget` 的 `onResourceReady()` 方法中做了一些逻辑处理，包括如果是 GIF 图片的话，就调用 `resource.start()` 方法开始播放图片，但是好像并没有看到哪里有将 `GlideDrawable` 显示到 `ImageView` 上的逻辑。

确实没有，不过父类里面有，这里在第 25 行调用了 `super.onResourceReady()` 方法，`GlideDrawableImageViewTarget` 的父类是 `ImageViewTarget`，我们来看下它的代码吧：

```

public abstract class ImageViewTarget<Z> extends ViewTarget<ImageView, Z> implements
GlideAnimation.ViewAdapter {

    ...

    @Override
    public void onResourceReady(Z resource, GlideAnimation<? super Z> glideAnimation) {
        if (glideAnimation == null || !glideAnimation.animate(resource, this)) {
            setResource(resource);
        }
    }

    protected abstract void setResource(Z resource);
}

```

```
}
```

可以看到，在 `ImageViewTarget` 的 `onResourceReady()` 方法当中调用了 `setResource()` 方法，而 `ImageViewTarget` 的 `setResource()` 方法是一个抽象方法，具体的实现还是在子类那边实现的。

那子类的 `setResource()` 方法是怎么实现的呢？回头再来看一下 `GlideDrawableImageViewTarget` 的 `setResource()` 方法，没错，调用的 `view.setImageDrawable()` 方法，而这个 `view` 就是 `ImageView`。代码执行到这里，图片终于也就显示出来了。

那么，我们对 `Glide` 执行流程的源码分析，到这里也终于结束了。

2. EventBus

1.1、EventBus

`EventBus` 是一个 `Android` 事件发布/订阅框架，通过解耦发布者和订阅者简化 `Android` 事件传递，这里的事件可以理解为消息，本文中统一称为事件。事件传递既可用于 `Android` 四大组件间通讯，也可以用户异步线程和主线程间通讯等等。

传统的事件传递方式包括：`Intent`、`Handler`、`BroadcastReceiver`、`Interface` 回调，相比之下 `EventBus` 的优点是代码简洁，使用简单，并将事件发布和订阅充分解耦。可简化 `Activities`、`Fragments`、`Threads`、`Services` 等组件间的消息传递，可替代 `Intent`、`Handler`、`Broadcast`、接口等传统方案，更快，代码更小，50K 左右的 `jar` 包，代码更优雅，彻底解耦。

1.2、概念

事件(Event)：又可称为消息，本文中统一用事件表示。其实就是一个对象，可以是网络请求返回的字符串，也可以是某个开关状态等等。事件类型(`EventType`)指事件所属的 `Class`。

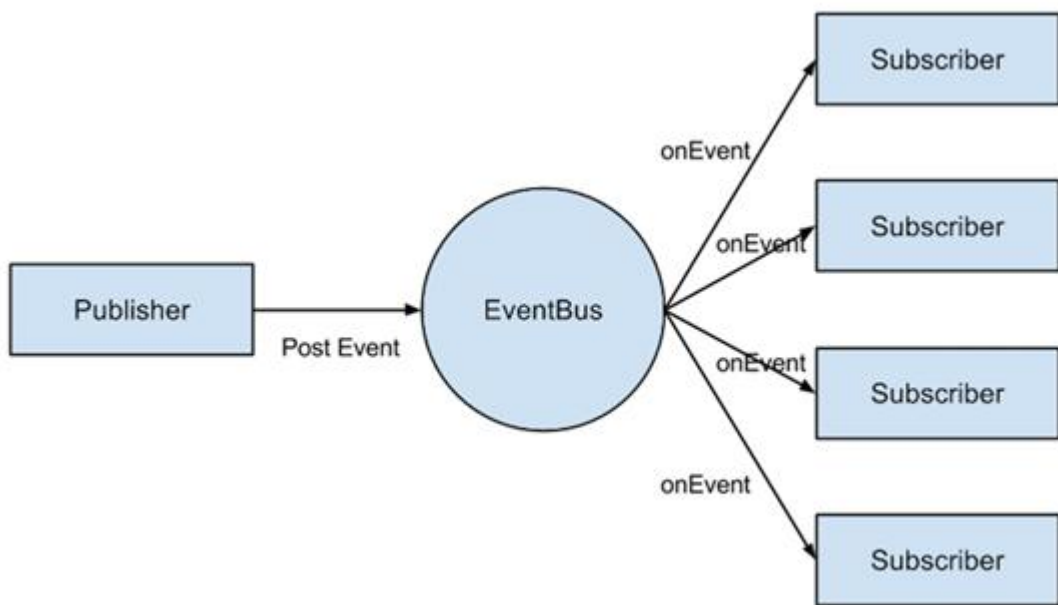
事件分为一般事件和 `Sticky` 事件，相对于一般事件，`Sticky` 事件不同之处在于，当事件发布后，再有订阅者开始订阅该类型事件，依然能收到该类型事件最近一个 `Sticky` 事件。

订阅者(Subscriber)：订阅某种事件类型的对象。当有发布者发布这类事件后，`EventBus` 会执行订阅者的 `onEvent` 函数，这个函数叫事件响应函数。订阅者通过 `register` 接口订阅某个事件类型，`unregister` 接口退订。订阅者存在优先级，优先级高的订阅者可以取消事件继续向优先级低的订阅者分发，默认所有订阅者优先级都为 0。

发布者(Publisher)：发布某事件的对象，通过 `post` 接口发布事件。

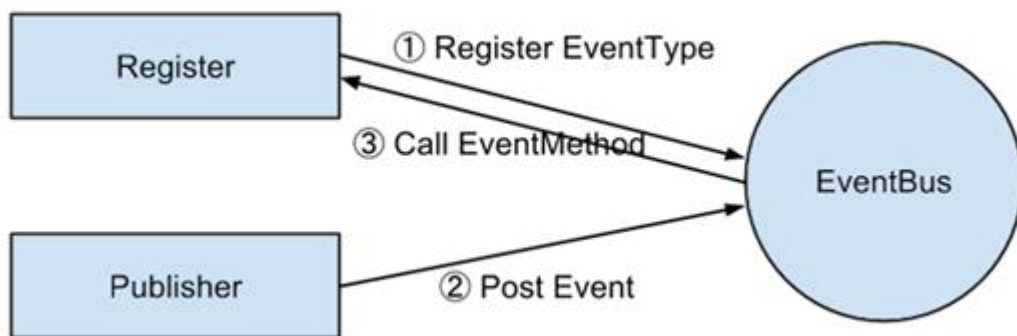
1.3、订阅者、发布者、EventBus 关系图

`EventBus` 负责存储订阅者、事件相关信息，订阅者和发布者都只和 `EventBus` 关联。



事件响应流程

订阅者首先调用 EventBus 的 register 接口订阅某种类型的事件，当发布者通过 post 接口发布该类型的事件时，EventBus 执行调用者的事件响应函数。



二、EventBus 优势

2.1、对比 Java 监听器接口（Listener Interfaces）

在 Java 中，特别是 Android，一个常用的模式就是使用“监听器（Listeners）”接口。在此模式中，一个实现了监听器接口的类必须将自身注册到它想要监听的类中去。这就意味着监听与被监听之间属于强关联关系。这种关系就使得单元测试很难进行开展。

2.2、对比本地广播管理器（LocalBroadcastManager）

另一项技术就是在组件间通过本地广播管理器（LocalBroadcastManager）进行消息的发送与监听。虽然这对于解耦有很好的帮助，但它的 API 不如 EventBus 那样简洁。此外，如果你不注意保持 Intent extras 类型的一致，它还可能引发潜在的运行时/类型检测错误。

使用 EventBus 不仅使代码变得清晰，而且增强了类型安全（type-safe）。当用 Intent 传递数据时，在编译时并不能检查出所设的 extra 类型与收到时的类型一致。所以一个很常见的错误便是你或者你团队中的其他人改变了 Intent 所传递的数据，但忘记了对全部的接收器（receiver）进行更新。这种错误在编译时是无法被发现的，只有在运行时才会发现问题。

而使用 EventBus 所传递的消息则是通过你所定义的 Event 类。由于接收者方法是直接与这些类实例打交道，所以所有的数据均可以进行类型检查，这样任何由于类型不一致所导致的错误都可以在编译时刻被发现。

另外就是你的 Event 类可以定义成任何类型。通常会为了表示事件而显式地创建明确命名的类，你也通过 EventBus 发送/接收任何类。通过这种方法，你就不必受限于那些只能添加到 Intent extras 中的简单数据类型了。例如，你可以发送一个和 ORM 模型类实例，并且在接收端直接处理与 ORM 操作相关的类实例。

三、EventBus 使用

3.1 基本使用

(1) 自定义一个类，可以是空类，比如：

```
[[java] 1 2 3 C P
01. public class AnyEventType {
02.     public AnyEventType(){}
03. }
```

(2) 在要接收消息的页面注册：

```
[[java] 1 2 3 C P
01. EventBus.register(this);
```

(3) 发送消息

```
[[java] 1 2 3 C P
01. EventBus.post(new AnyEventType event);
```

(4) 接受消息的页面实现(共有四个函数，各功能不同，这是其中之一，可以选择性的实现，这里先实现一个)：

```
[[java] 1 2 3 C P
01. public void onEvent(AnyEventType event) {}
```

(5) 解除注册

```
[[java] 1 2 3 C P
01. EventBus.unregister(this);
```

3.2 具体案例

```
[java] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

01. package com.example.tryeventbus_simple;
02.
03. import com.harvic.other.FirstEvent;
04.
05. import de.greenrobot.event.EventBus;
06. import android.app.Activity;
07. import android.os.Bundle;
08. import android.view.View;
09. import android.widget.Button;
10.
11. public class SecondActivity extends Activity {
12.     private Button btn_FirstEvent;
13.
14.     @Override
15.     protected void onCreate(Bundle savedInstanceState) {
16.         super.onCreate(savedInstanceState);
17.         setContentView(R.layout.activity_second);
18.         btn_FirstEvent = (Button) findViewById(R.id.btn_first_event);
19.
20.         btn_FirstEvent.setOnClickListener(new View.OnClickListener() {
21.
22.             @Override
23.             public void onClick(View v) {
24.                 // TODO Auto-generated method stub
25.                 EventBus.getDefault().post(
26.                     new FirstEvent("FirstEvent btn clicked"));
27.             }
28.         });
29.     }
30. }
```

```

15. public class MainActivity extends Activity {
16.
17.     Button btn;
18.     TextView tv;
19.
20.     @Override
21.     protected void onCreate(Bundle savedInstanceState) {
22.         super.onCreate(savedInstanceState);
23.         setContentView(R.layout.activity_main);
24.
25.         EventBus.getDefault().register(this);
26.
27.         btn = (Button) findViewById(R.id.btn_try);
28.         tv = (TextView) findViewById(R.id.tv);
29.
30.         btn.setOnClickListener(new View.OnClickListener() {
31.
32.             @Override
33.             public void onClick(View v) {
34.                 // TODO Auto-generated method stub
35.                 Intent intent = new Intent(getApplicationContext(),
36.                     SecondActivity.class);
37.                 startActivity(intent);
38.             }
39.         });
40.     }
41.
42.     public void onEventMainThread(FirstEvent event) {
43.
44.         String msg = "onEventMainThread收到了消息: " + event.getMsg();
45.         Log.d("harvic", msg);
46.         tv.setText(msg);
47.         Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
48.     }
49.
50.     @Override
51.     protected void onDestroy(){
52.         super.onDestroy();
53.         EventBus.getDefault().unregister(this);

```

```

01. package com.harvic.other;
02.
03. public class FirstEvent {
04.
05.     private String mMsg;
06.     public FirstEvent(String msg) {
07.         // TODO Auto-generated constructor stub
08.         mMsg = msg;
09.     }
10.     public String getMsg(){
11.         return mMsg;
12.     }
13. }

```

3.3 onEvent 函数使用解析

前一篇给大家简单演示了 EventBus 的 onEventMainThread () 函数的接收，其实 EventBus 还有另外有个不同的函数，他们分别是：

- 1、onEvent
- 2、onEventMainThread
- 3、onEventBackgroundThread
- 4、onEventAsync

这四种订阅函数都是使用 onEvent 开头的，它们的功能稍有不同，在介绍不同之前先介绍两个概念：告知观察者事件发生时通过 EventBus.post 函数实现，这个过程叫做事件的发布，观察者被告知事件发生叫做事件的接收，是通过下面的订阅函数实现的。

onEvent: 如果使用 onEvent 作为订阅函数，那么该事件在哪个线程发布出来的，onEvent 就会在这个线程中运行，也就是说发布事件和接收事件线程在同一个线程。使用这个方法时，在 onEvent 方法中不能执行耗时操作，如果执行耗时操作容易导致事件分发延迟。

onEventMainThread: 如果使用 onEventMainThread 作为订阅函数，那么不论事件是在哪个线程中发布出来的，onEventMainThread 都会在 UI 线程中执行，接收事件就会在 UI 线程中运行，这个在 Android 中是非常有用的，因为在 Android 中只能在 UI 线程中更新 UI，所以在 onEventMainThread 方法中是不能执行耗时操作的。

onEventBackground: 如果使用 onEventBackground 作为订阅函数，那么如果事件是在 UI 线程中发布出来的，那么 onEventBackground 就会在子线程中运行，如果事件本来就是子线程中发布出来的，那么 onEventBackground 函数直接在该子线程中执行。

onEventAsync: 使用这个函数作为订阅函数，那么无论事件在哪个线程发布，都会创建新的子线程在执行 onEventAsync。

3.4 EventBus3.0 使用解析

注册一般是在 onCreate 和 onStart 里注册，尽量不要在 onResume，可能出现多次注册的情况，比如下面这个异常：

可以先判断下：

```
1      if (!EventBus.getDefault().isRegistered(this)) {
2          EventBus.getDefault().register(this);
3      }
```

取消注册 要写到 onDestroy 方法里，不要写到 onStop 里，有时会出现异常的哦

EventBus 3 和之前版本的 EventBus 不兼容，这里采用注解的方法来接收事件，四种注解 @Subscribe、@Subscribe(threadMode = ThreadMode.ASYNC)、@Subscribe(threadMode = ThreadMode.BACKGROUND)、@Subscribe(threadMode = ThreadMode.MAIN)分别对应之前的 onEvent()、onEventAsync()、onEventBackground()、onEventMainThread()。

EventBus 3 采用注解后，方法名没有限制了，参数只有一个，和发送者 `post` 的参数对应配对，在未声明 `threadMode` 时，默认的线程模式为 `ThreadMode.POSTING`，只有在该模式下才可以取消线程。

由于可在任何地方都可以 `post` 一个事件，那么在不同线程之间传递事件，比如在工作线程传递一个事件更新 UI 线程中的一个控件，则需要注意 `threadMode` 的切换。

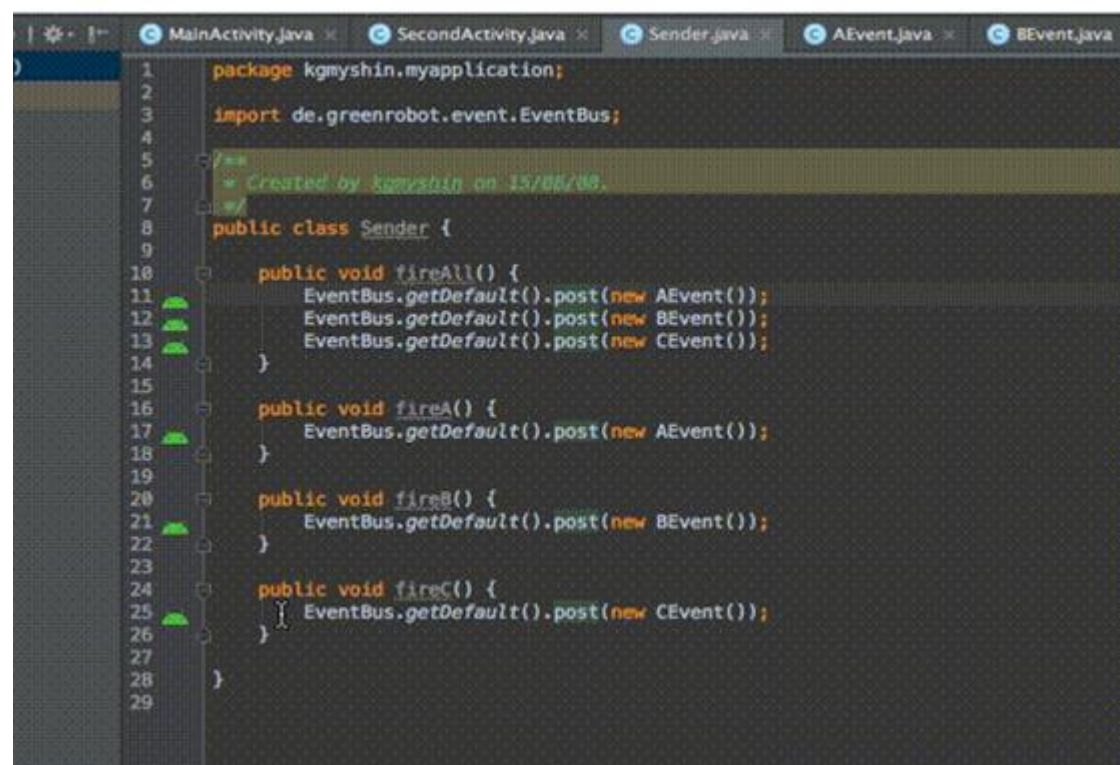
如果遇到订阅事件无法执行的情况，分析后发现是订阅事件的 `Activity` 还未执行的原因。找到原因就好办了，这时候就需要用到 `postSticky`。

发布事件时用 `postSticky` 操作：

```
1 EventBus.getDefault().postSticky(event);
```

订阅时，添加 `sticky = true`

```
1 @Subscribe(sticky = true)    //看下 `@Subscribe` 源码知道 `sticky` 默认是 `false`
2 public void onEvent(Event e) {
3     ---
4 }
```



```

1 package kgmyshin.myapplication;
2
3 import android.app.Activity;
4 import android.os.Bundle;
5
6 import de.greenrobot.event.Subscribe;
7 import de.greenrobot.event.ThreadMode;
8
9 public class MainActivity extends Activity {
10
11     @Override
12     protected void onCreate(Bundle savedInstanceState) {
13         super.onCreate(savedInstanceState);
14         setContentView(R.layout.activity_main);
15     }
16
17     @Subscribe
18     public void handleAEvent(AEvent event) {
19
20     }
21
22     @Subscribe(threadMode = ThreadMode.MainThread)
23     public void handleAEventMainThread(AEvent event) {
24
25     }
26
27     @Subscribe
28     public void handleBEvent(BEvent event) {
29
30     }
31
32 }

```

```

1 package kgmyshin.myapplication;
2
3 import android.app.Activity;
4 import de.greenrobot.event.Subscribe;
5
6 //
7 // Created by kgmyshin on 15/05/09.
8 //
9
10 public class SecondActivity extends Activity {
11
12     @Subscribe
13     public void handle(CEvent event) {
14
15     }
16
17     @Subscribe
18     public void handle(AEvent event) {
19
20     }
21
22     @Subscribe
23     public void handle(BEvent event) {
24
25     }
26
27 }
28

```

四、 EventBus 源码解析

4.1 register

EventBus.getDefault().register(this); EventBus.getDefault()其实就是个单例，和我们传统的 getInstance 一个意思：

```

01.  /** Convenience singleton for apps using a process-wide EventBus instance. */
02.  public static EventBus getDefault() {
03.      if (defaultInstance == null) {
04.          synchronized (EventBus.class) {
05.              if (defaultInstance == null) {
06.                  defaultInstance = new EventBus();
07.              }
08.          }
09.      }
10.      return defaultInstance;
11.  }

```

使用了双重判断的方式，防止并发的问題，还能极大的提高效率。

register 公布给我们使用的有 4 个：

```

01.  public void register(Object subscriber) {
02.      register(subscriber, DEFAULT_METHOD_NAME, false, 0);
03.  }
04.  public void register(Object subscriber, int priority) {
05.      register(subscriber, DEFAULT_METHOD_NAME, false, priority);
06.  }
07.  public void registerSticky(Object subscriber) {
08.      register(subscriber, DEFAULT_METHOD_NAME, true, 0);
09.  }
10.  public void registerSticky(Object subscriber, int priority) {
11.      register(subscriber, DEFAULT_METHOD_NAME, true, priority);
12.  }

```

```

01.  private synchronized void register(Object subscriber, String methodName, boolean sticky, int priority) {
02.      List<SubscriberMethod> subscriberMethods = subscriberMethodFinder.findSubscriberMethods(subscriber.getClass(),
03.          methodName);
04.      for (SubscriberMethod subscriberMethod : subscriberMethods) {
05.          subscribe(subscriber, subscriberMethod, sticky, priority);
06.      }
07.  }

```

调用内部类 **SubscriberMethodFinder** 的 **findSubscriberMethods** 方法，传入了 **subscriber** 的 **class**，以及 **methodName**，返回一个 **List<SubscriberMethod>**。

那么不用说，肯定是去遍历该类内部所有方法，然后根据 **methodName** 去匹配，匹配成功的封装成 **SubscriberMethod**，最后返回一个 **List**。


```

01. List<SubscriberMethod> findSubscriberMethods(Class<?> subscriberClass, String eventMethodName) {
02.     String key = subscriberClass.getName() + '.' + eventMethodName;
03.     List<SubscriberMethod> subscriberMethods;
04.     synchronized (methodCache) {
05.         subscriberMethods = methodCache.get(key);
06.     }
07.     if (subscriberMethods != null) {
08.         return subscriberMethods;
09.     }
10.     subscriberMethods = new ArrayList<SubscriberMethod>();
11.     Class<?> clazz = subscriberClass;
12.     HashSet<String> eventTypesFound = new HashSet<String>();
13.     StringBuilder methodKeyBuilder = new StringBuilder();
14.     while (clazz != null) {
15.         String name = clazz.getName();
16.         if (name.startsWith("java.") || name.startsWith("javax.") || name.startsWith("android.")) {
17.             // Skip system classes, this just degrades performance
18.             break;
19.         }
20.         // Starting with EventBus 2.2 we enforce
21.         Method[] methods = clazz.getMethods();
22.         for (Method method : methods) {
23.             String methodName = method.getName();
24.             if (methodName.startsWith(eventMethodName)) {
25.                 int modifiers = method.getModifiers();
26.                 if ((modifiers & Modifier.PUBLIC) != 0 && (modifiers & Modifier.ABSTRACT) == 0) {
27.                     Class<?>[] parameterTypes = method.getParameterTypes();
28.                     if (parameterTypes.length == 1) {
29.                         String modifierString = methodName.substring(eventMethodName.length());
30.                         ThreadMode threadMode;
31.                         if (modifierString.length() == 0) {
32.                             threadMode = ThreadMode.PostThread;
33.                         } else if (modifierString.equals("MainThread")) {
34.                             threadMode = ThreadMode.MainThread;
35.                         } else if (modifierString.equals("BackgroundThread")) {
36.                             threadMode = ThreadMode.BackgroundThread;
37.                         } else if (modifierString.equals("Async")) {
38.

```

25-29行：分别判断

了是否以onEvent开
头，是否是public且

非static和abstract
方法，是否是一个参

数。如果都复合，才

进入封装的部分

32-45行：也比较简
单，根据方法的后缀，
来确定threadMode，
threadMode是个枚举
类型：就四种情况

```

33.         threadMode = ThreadMode.PostThread;
34.     } else if (modifierString.equals("MainThread")) {
35.         threadMode = ThreadMode.MainThread;
36.     } else if (modifierString.equals("BackgroundThread")) {
37.         threadMode = ThreadMode.BackgroundThread;
38.     } else if (modifierString.equals("Async")) {
39.         threadMode = ThreadMode.Async;
40.     } else {
41.         if (skipMethodVerificationForClasses.containsKey(clazz)) {
42.             continue;
43.         } else {
44.             throw new EventBusException("Illegal onEvent method, check for typos: " + method);
45.         }
46.     }
47.     Class<?> eventType = parameterTypes[0];
48.     methodKeyBuilder.setLength(0);
49.     methodKeyBuilder.append(methodName);
50.     methodKeyBuilder.append('>').append(eventType.getName());
51.     String methodKey = methodKeyBuilder.toString();
52.     if (eventTypesFound.add(methodKey)) {
53.         // Only add if not already found in a sub-class
54.         subscriberMethods.add(new SubscriberMethod(method, threadMode, eventType));
55.     }
56. }
57. } else if (!skipMethodVerificationForClasses.containsKey(clazz)) {
58.     Log.d(EventBus.TAG, "Skipping method (not public, static or abstract): " + clazz + "."
59.         + methodName);
60. }
61. }
62. }
63. class = class.getSuperclass();
64. }
65. if (subscriberMethods.isEmpty()) {
66.     throw new EventBusException("Subscriber " + subscriberClass + " has no public methods called "
67.         + eventMethodName);
68. } else {
69.     synchronized (methodCache) {
70.         methodCache.put(key, subscriberMethods);
71.     }
72.     return subscriberMethods;

```

54行 将method, threadMode, eventType传入构造了: new SubscriberMethod(method, threadMode, eventType), 添加到List, 最终放回

63行: class = class.getSuperclass(); 可以看到, 会扫描所有的父类, 不仅仅是当前类

继续回到 register:

```

01.     for (SubscriberMethod subscriberMethod : subscriberMethods) {
02.         subscribe(subscriber, subscriberMethod, sticky, priority);
03.     }

```

```

01. // Must be called in synchronized block
02. private void subscribe(Object subscriber, SubscriberMethod subscriberMethod, boolean sticky, int priority) {
03.     subscribed = true;
04.     Class<?> eventType = subscriberMethod.eventType;
05.     CopyOnWriteArrayList<Subscription> subscriptions = subscriptionsByEventType.get(eventType);
06.     Subscription newSubscription = new Subscription(subscriber, subscriberMethod, priority);
07.     if (subscriptions == null) {
08.         subscriptions = new CopyOnWriteArrayList<Subscription>();
09.         subscriptionsByEventType.put(eventType, subscriptions);
10.     } else {
11.         for (Subscription subscription : subscriptions) {
12.             if (subscription.equals(newSubscription)) {
13.                 throw new EventBusException("Subscriber " + subscriber.getClass() + " already registered to event "
14.                     + eventType);
15.             }
16.         }
17.     }
18.
19.     // Starting with EventBus 2.2 we enforced methods to be public (might change with annotations again)
20.     // subscriberMethod.method.setAccessible(true);
21.
22.     int size = subscriptions.size();
23.     for (int i = 0; i <= size; i++) {
24.         if (i == size || newSubscription.priority > subscriptions.get(i).priority) {
25.             subscriptions.add(i, newSubscription);
26.             break;
27.         }
28.     }
29.
30.     List<Class<?>> subscribedEvents = typesBySubscriber.get(subscriber);
31.     if (subscribedEvents == null) {
32.         subscribedEvents = new ArrayList<Class<?>>();
33.         typesBySubscriber.put(subscriber, subscribedEvents);
34.     }
35.     subscribedEvents.add(eventType);
36.
37.     if (sticky) {
38.         Object stickyEvent;
39.         synchronized (stickyEvents) {
40.             stickyEvent = stickyEvents.get(eventType);
41.         }
42.         if (stickyEvent != null) {
43.             // If the subscriber is trying to abort the event, it will fail (event is not tracked in posting state)
44.             // --> Strange corner case, which we don't take care of here.
45.             postToSubscription(newSubscription, stickyEvent, Looper.getMainLooper() == Looper.myLooper());
46.         }
47.     }
48. }

```

4-17. 这里的 `subscriptionsByEventType` 是个 Map, key: `eventType`; value: `CopyOnWriteArrayList<Subscription>`; 这个 Map 其实就是 EventBus 存储方法的地方, 一定要记住

22-28行: 实际上, 就是添加 `newSubscription`; 并且是按照优先级添加的。可以看到, 优先级越高, 会插到在当前 List 的前面

30-35. 根据 `subscriber` 存储它所有的 `eventType`; 依然是 map; key: `subscriber`, value: `List<eventType>`; 知道就行, 非核心代码, 主要用于 `isRegister` 的判断

37-47 判断 `sticky`; 如果为 `true`, 从 `stickyEvents` 中根据 `eventType` 去查找有没有 `stickyEvent`, 如果有则立即发布去执行。 `stickyEvent` 其实就是我们 `post` 时的参数

到此, 我们 `register` 就介绍完了。

你只要记得一件事: 扫描了所有的方法, 把匹配的方法最终保存在 **`subscriptionsByEventType` (Map, key: `eventType`; value: `CopyOnWriteArrayList<Subscription>`)** 中;

`eventType` 是我们方法参数的 **Class**, **`Subscription`** 中则保存着 **`subscriber`, `subscriberMethod` (`method, threadMode, eventType`), `priority`**; 包含了执行改方法所需的一切。

4.2 post


```
01. /** Posts the given event to the event bus. */
```

```
02. public void post(Object event) {
```

```
03.     PostingThreadState postingState = currentPostingThreadState.get();
```

```
04.     List<Object> eventQueue = postingState.eventQueue;
```

```
05.     eventQueue.add(event);
```

```
06.
```

```
07.     if (postingState.isPosting) {
```

```
08.         return;
```

```
09.     } else {
```

```
10.         postingState.isMainThread = Looper.getMainLooper() == Looper.myLooper();
```

```
11.         postingState.isPosting = true;
```

```
12.         if (postingState.canceled) {
```

```
13.             throw new EventBusException("Internal error. Abort state was not reset");
```

```
14.         }
```

```
15.         try {
```

```
16.             while (!eventQueue.isEmpty()) {
```

```
17.                 postSingleEvent(eventQueue.remove(0), postingState);
```

```
18.             }
```

```
19.         } finally {
```

```
20.             postingState.isPosting = false;
```

```
21.             postingState.isMainThread = false;
```

```
22.         }
```

```
23.     }
```

```
24. }
```

把我们传入的event，保存

到了当前线程中的一个变

量PostingThreadState的

eventQueue中。

10行：判断当前是否是UI

线程

16-18行：遍历队列中的所有的event，调用
postSingleEvent (eventQueue.remove
(0), postingState) 方法。

这里大家会不会有疑问，每次post都会去调
用整个队列么，那么不会造成方法多次调用
么？

可以看到第7-8行，有个判断，就是防止该问
题的，isPosting=true了，就不会往下走了

```
01. private void postSingleEvent(Object event, PostingThreadState postingState) throws Error {
```

```
02.     Class<? extends Object> eventClass = event.getClass();
```

```
03.     List<Class<?>> eventTypes = findEventTypes(eventClass);
```

```
04.     boolean subscriptionFound = false;
```

```
05.     int countTypes = eventTypes.size();
```

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

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

```
08.         CopyOnWriteArrayList<Subscription> subscriptions;
```

```
09.         synchronized (this) {
```

```
10.             subscriptions = subscriptionsByEventType.get(clazz);
```

```
11.         }
```

```
12.         if (subscriptions != null && !subscriptions.isEmpty()) {
```

```
13.             for (Subscription subscription : subscriptions) {
```

```
14.                 postingState.event = event;
```

```
15.                 postingState.subscription = subscription;
```

```
16.                 boolean aborted = false;
```

```
17.                 try {
```

```
18.                     postToSubscription(subscription, event, postingState.isMainThread);
```

```
19.                     aborted = postingState.canceled;
```

```
20.                 } finally {
```

```
21.                     postingState.event = null;
```

```
22.                     postingState.subscription = null;
```

```
23.                     postingState.canceled = false;
```

```
24.                 }
```

```
25.                 if (aborted) {
```

```
26.                     break;
```

```
27.                 }
```

```
28.             }
```

```
29.             subscriptionFound = true;
```

```
30.         }
```

```
31.     }
```

```
32.     if (!subscriptionFound) {
```

```
33.         Log.d(TAG, "No subscribers registered for event " + eventClass);
```

```
34.         if (eventClass != NoSubscriberEvent.class && eventClass != SubscriberExceptionEvent.class) {
```

```
35.             post(new NoSubscriberEvent(this, event));
```

2-3行：根据event的Class，去得到

一个List<Class<?>>；其实就是得

到event当前对象的Class，以及父类

和接口的Class类型；主要用于匹

配，比如你传入Dog extends

Dog，他会把Animal也装到该List中

6-31行：遍历所有的Class，
到subscriptionsByEventTy
pe去查找subscriptions

```

01. private void postToSubscription(Subscription subscription, Object event, boolean isMainThread) {
02.     switch (subscription.subscriberMethod.threadMode) {
03.         case PostThread:
04.             invokeSubscriber(subscription, event);
05.             break;
06.         case MainThread:
07.             if (isMainThread) {
08.                 invokeSubscriber(subscription, event);
09.             } else {
10.                 mainThreadPoster.enqueue(subscription, event);
11.             }
12.             break;
13.         case BackgroundThread:
14.             if (isMainThread) {
15.                 backgroundPoster.enqueue(subscription, event);
16.             } else {
17.                 invokeSubscriber(subscription, event);
18.             }
19.             break;
20.         case Async:
21.             asyncPoster.enqueue(subscription, event);
22.             break;
23.         default:
24.             throw new IllegalStateException("Unknown thread mode: " + subscription.subscriberMethod.threadMode);
25.     }
26. }

01. void invokeSubscriber(Subscription subscription, Object event) throws Error {
02.     subscription.subscriberMethod.method.invoke(subscription.subscriber, event);
03. }

```

到此，我们完整的源码分析就结束了，总结一下：**register** 会把当前类中匹配的方法，存入一个 **map**，而 **post** 会根据实参去 **map** 查找进行反射调用。分析这么久，一句话就说完了~~

其实不用发布者，订阅者，事件，总线这几个词或许更好理解，以后大家问了 **EventBus**，可以说，就是在一个单例内部维持着一个 **map** 对象存储了一堆的方法；**post** 无非就是根据参数去查找方法，进行反射调用。

介绍了register和post；大家获取还能想到一个词sticky，在register中，如何sticky为true，会去stickyEvents去查找事件，然后立即去post；

那么这个stickyEvents何时进行保存事件呢？

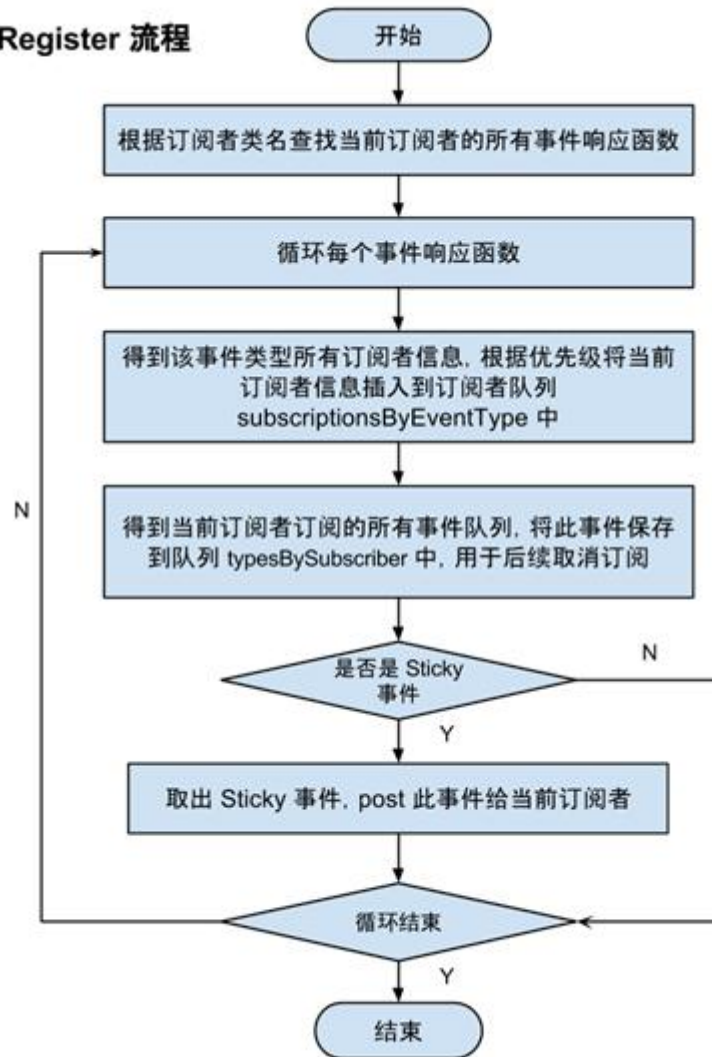
其实eventbus中，除了post发布事件，还有一个方法也可以：

```

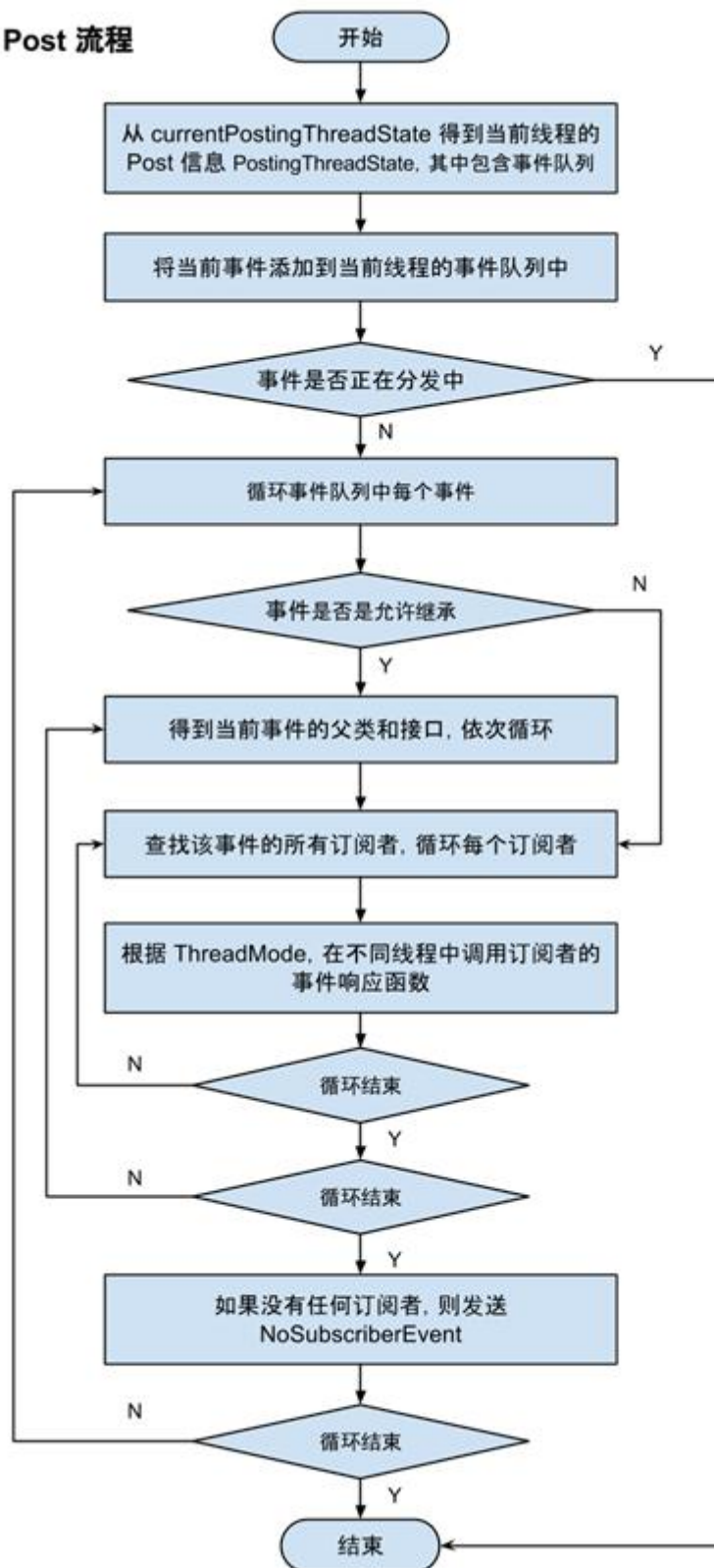
[Java] 1 2 3 4 5 6 7
01. public void postSticky(Object event) {
02.     synchronized (stickyEvents) {
03.         stickyEvents.put(event.getClass(), event);
04.     }
05.     // Should be posted after it is putted, in case the subscriber wants to remove immediately
06.     post(event);
07. }

```


Register 流程



Post 流程



3. LeakCanary

「Leakcanary」是我们经常用于检测内存泄漏的工具，简单的使用方式，内存泄漏的可视化，是我们开发中必备的工具之一。

分析源码之前

[Leakcanary](#) 大神的 [github](#)，最好的老师。

一、使用

1、配置

```
dependencies {
    debugImplementation
    'com.squareup.leakcanary:leakcanary-android:1.6.3'
    releaseImplementation
    'com.squareup.leakcanary:leakcanary-android-no-op:1.6.3'
    // Optional, if you use support library fragments:
    debugImplementation
    'com.squareup.leakcanary:leakcanary-support-fragment:1.6.3' }
```

2、简单使用

```
public class ExampleApplication extends Application {

    @Override public void onCreate() {
        super.onCreate();
        if (LeakCanary.isInAnalyzerProcess(this)) {
            // This process is dedicated to LeakCanary for heap analysis.
            // You should not init your app in this process.
            return;
        }
        LeakCanary.install(this);
        // Normal app init code...
    }
}
```

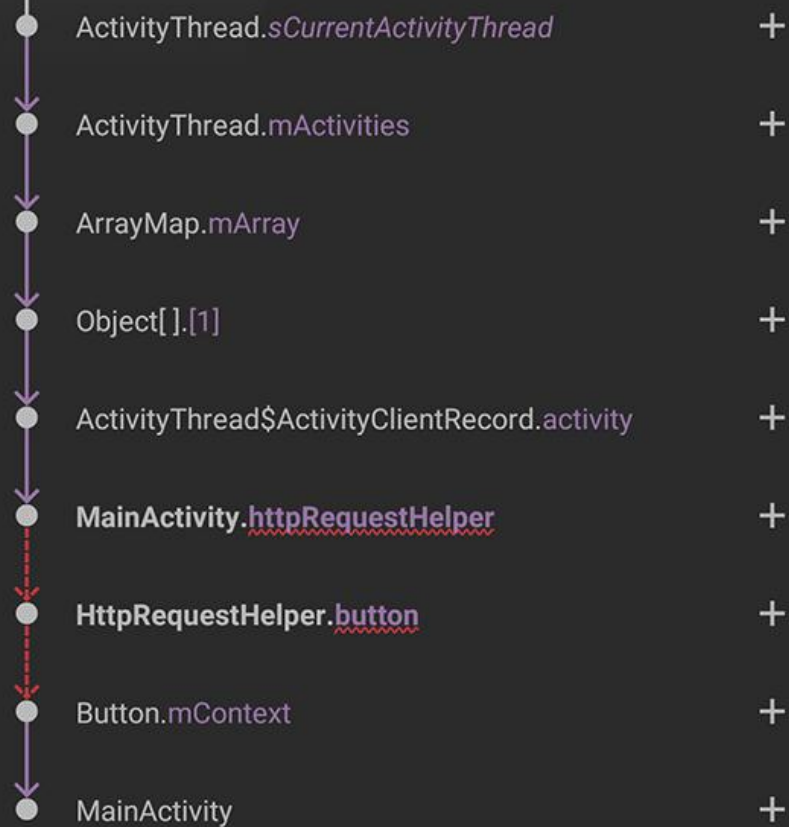
超级简单的配置和使用方式。最后就会得出以下的事例说明。

LTE 05:35

← MainActivity leaked

com.example.leakcanary

Tap here to learn more



DELETE

二、准备工作

1、Reference

Reference 把内存分为 4 种状态，Active 、 Pending 、 Enqueued 、 Inactive。

- Active 一般说来内存一开始被分配的状态都是 Active
- Pending 快要放入队列（ReferenceQueue）的对象，也就是马上要回收的对象
- Enqueued 对象已经进入队列，已经被回收的对象。方便我们查询某个对象是否被回收
- Inactive 最终的状态，无法变成其他的状态。

2、ReferenceQueue

引用队列，在 Reference 被回收的时候，Reference 会被添加到 ReferenceQueue 中

3、如果检测一个对象是否被回收

需要采用 Reference + ReferenceQueue

- 创建一个引用队列 queue
- 创建 Reference 对象（通常用弱引用）并关联引用队列
- 在 Reference 被回收的时候，Reference 会被添加到 queue 中

```
//创建一个引用队列 ReferenceQueue queue = new ReferenceQueue();  
// 创建弱引用，此时状态为 Active，并且 Reference.pending 为空，// 当前  
Reference.queue = 上面创建的 queue，并且 next=null // reference 创建并  
关联 queueWeakReference reference = new WeakReference(new Object(),  
queue);  
// 当 GC 执行后，由于是弱引用，所以回收该 object 对象，并且置于 pending  
上，此时 reference 的状态为 PENDING System.gc();  
// ReferenceHandler 从 pending 中取下该元素，并且将该元素放入到 queue  
中，//此时 Reference 状态为 ENQUEUED, Reference.queue = ReferenceENQUEUED  
// 当从 queue 里面取出该元素，则变为 INACTIVE，Reference.queue =  
Reference.NULL Reference reference1 = queue.remove();
```

在 Reference 类加载的时候，Java 虚拟机会创建一个最大优先级的后台线程，这个线程的工作就是不断检测 pending 是否为 null，如果不为 null，那么就将它放到 ReferenceQueue。因为 pending 不为 null，就说明引用所指向的对象已经被 GC，变成了不也达。

4、ActivityLifecycleCallbacks

用于监听所有 **Activity** 生命周期的回调方法。

```
private final Application.ActivityLifecycleCallbacks
lifecycleCallbacks =
    new Application.ActivityLifecycleCallbacks() {
        @Override public void onActivityCreated(Activity activity, Bundle
savedInstanceState) {
        }

        @Override public void onActivityStarted(Activity activity) {
        }

        @Override public void onActivityResumed(Activity activity) {
        }

        @Override public void onActivityPaused(Activity activity) {
        }

        @Override public void onActivityStopped(Activity activity) {
        }

        @Override public void onActivitySaveInstanceState(Activity
activity, Bundle outState) {
        }

        @Override public void onActivityDestroyed(Activity activity) {
            ActivityRefWatcher.this.onActivityDestroyed(activity);
        }
    };
```

5、Heap Dump

Heap Dump 也叫堆转储文件，是一个 **Java** 进程在某个时间点上的内存快照。

三、原理说明

1、监听 **Activity** 的生命周期。
2、在 **onDestory** 的时候，创建对应的 **Activity** 的 **Refrence** 和 相应的 **RefrenceQueue**，启动后台进程去检测。
3、一段时间后，从 **RefrenceQueue** 中读取，如果有这个 **Activity** 的 **Refrence**，那么说明这个 **Activity** 的 **Refrence** 已经被回收，但是如果 **RefrenceQueue** 没有这个 **Activity** 的 **Refrence** 那就说明出现了内存泄漏。
4、dump 出 **hprof** 文件，找到泄漏路径。

分析源码

程序的唯一入口 `LeakCanary.install(this);`

1、install

DisplayLeakService 这个类负责发起 **Notification** 以及将结果记录下来写在文件里面。以后每次启动 **LeakAnalyzerActivity** 就从这个文件里读取历史结果，并展示给我们。

```
public static RefWatcher install(Application application) {
    return install(application, DisplayLeakService.class);
}
public static RefWatcher install(Application application,
    Class<? extends AbstractAnalysisResultService>
listenerServiceClass) {
    //如果在主线程 那么返回一个无用的 RefWatcher 详解 1.1
    if (isInAnalyzerProcess(application)) {
        return RefWatcher.DISABLED;
    }
    //把 DisplayLeakActivity 设置为可用 用于显示 DisplayLeakActivity
    //就是我们看到的那个分析界面
    enableDisplayLeakActivity(application);
    // 详解 1.2
    HeapDump.Listener heapDumpListener =
        new ServiceHeapDumpListener(application, listenerServiceClass);
    //详解 2
    RefWatcher refWatcher = androidWatcher(application,
heapDumpListener);
    //详解 3
    ActivityRefWatcher.installOnIcsPlus(application, refWatcher);
    return refWatcher;
}
```

1.1 isInAnalyzerProcess

因为 分析的进程是硬外一个独立进程 所以要判断是否是主进程,这个工作需要 在 **AnalyzerProcess** 中进行。

```
public static boolean isInAnalyzerProcess(Context context) {
    return isInServiceProcess(context, HeapAnalyzerService.class);
}
```

把 App 的进程 和 这个 Service 进程进行对比 。

```
private static boolean isInServiceProcess(Context context,
    Class<? extends Service> serviceClass) {
    PackageManager packageManager = context.getPackageManager();
    PackageInfo packageInfo;
    try {
        packageInfo =
packageManager.getPackageInfo(context.getPackageName(),
GET_SERVICES);
    } catch (Exception e) {
        Log.e("AndroidUtils", "Could not get package info for " +
context.getPackageName(), e);
        return false;
    }
    String mainProcess = packageInfo.applicationInfo.processName;

    ComponentName component = new ComponentName(context, serviceClass);
    ServiceInfo serviceInfo;
    try {
        serviceInfo = packageManager.getServiceInfo(component, 0);
    } catch (PackageManager.NameNotFoundException ignored) {
        // Service is disabled.
        return false;
    }

    if (serviceInfo.processName.equals(mainProcess)) {
        Log.e("AndroidUtils",
            "Did not expect service " + serviceClass + " to run in main
process " + mainProcess);
        // Technically we are in the service process, but we're not in the
service dedicated process.
        return false;
    }

    int myPid = android.os.Process.myPid();
    ActivityManager activityManager =
        (ActivityManager)
context.getSystemService(Context.ACTIVITY_SERVICE);
    ActivityManager.RunningAppProcessInfo myProcess = null;

    for (ActivityManager.RunningAppProcessInfo process :
activityManager.getRunningAppProcesses()) {
        if (process.pid == myPid) {
```



```

        myProcess = process;
        break;
    }
}
if (myProcess == null) {
    Log.e("AndroidUtils", "Could not find running process for " +
myPid);
    return false;
}
//把 App 的进程 和 这个 Service 进程进行对比
return myProcess.processName.equals(serviceInfo.processName);
}

```

1.2 ServiceHeapDumpListener

设置 DisplayLeakService 和 HeapAnalyzerService 的可用。
analyze 方法，开始分析 HeapDump。

```

public final class ServiceHeapDumpListener implements HeapDump.Listener
{

    private final Context context;
    private final Class<? extends AbstractAnalysisResultService>
listenerServiceClass;

    public ServiceHeapDumpListener(Context context,
        Class<? extends AbstractAnalysisResultService>
listenerServiceClass) {
        LeakCanary.setEnabled(context, listenerServiceClass, true);
        LeakCanary.setEnabled(context, HeapAnalyzerService.class, true);
        this.listenerServiceClass = checkNotNull(listenerServiceClass,
"listenerServiceClass");
        this.context = checkNotNull(context,
"context").getApplicationContext();
    }

    @Override public void analyze(HeapDump heapDump) {
        checkNotNull(heapDump, "heapDump");
        HeapAnalyzerService.runAnalysis(context, heapDump,
listenerServiceClass);
    }
}

```

2、RefWatcher

```

private final Executor watchExecutor;
private final DebuggerControl debuggerControl;
private final GcTrigger gcTrigger;
private final HeapDumper heapDumper;
private final Set<String> retainedKeys;
private final ReferenceQueue<Object> queue;
private final HeapDump.Listener heapdumpListener;

```

- **watchExecutor:** 执行内存泄漏检测的 **Executor**。
- **debuggerControl:** 用于查询是否在 **debug** 调试模式下，调试中不会执行内存泄漏检测。
- **gcTrigger:** **GC** 开关，调用系统 **GC**。
- **heapDumper:** 用于产生内存泄漏分析用的 **dump** 文件。即 **dump** 内存 **head**。
- **retainedKeys:** 保存待检测和产生内存泄漏的引用的 **key**。
- **queue:** 用于判断弱引用持有的对象是否被 **GC**。
- **heapdumpListener:** 用于分析 **dump** 文件，生成内存泄漏分析报告。

这里创建我们所需要的 **RefWatcher**。

```

public static RefWatcher androidWatcher(Application app,
HeapDump.Listener heapDumpListener) {
    DebuggerControl debuggerControl = new AndroidDebuggerControl();
    AndroidHeapDumper heapDumper = new AndroidHeapDumper(app);
    heapDumper.cleanup();
    return new RefWatcher(new AndroidWatchExecutor(), debuggerControl,
GcTrigger.DEFAULT,
        heapDumper, heapDumpListener);
}

```

3、ActivityRefWatcher

```

public static void installOnIcsPlus(Application application,
RefWatcher refWatcher) {
    if (SDK_INT < ICE_CREAM_SANDWICH) {
        // If you need to support Android < ICS, override onDestroy() in your
base activity.
        return;
    }
    ActivityRefWatcher activityRefWatcher = new
ActivityRefWatcher(application, refWatcher);
    activityRefWatcher.watchActivities();
}
//注册 lifecycleCallbacks
public void watchActivities() {

```

```

        // Make sure you don't get installed twice.
        stopWatchingActivities();

application.registerActivityLifecycleCallbacks(lifecycleCallbacks);
    }
    //ArrayList<ActivityLifecycleCallbacks> mActivityLifecycleCallbacks
    //就是 mActivityLifecycleCallbacks 的添加
    public void
registerActivityLifecycleCallbacks(ActivityLifecycleCallbacks
callback) {
        synchronized (mActivityLifecycleCallbacks) {
            mActivityLifecycleCallbacks.add(callback);
        }
    }
    // 注销 lifecycleCallbacks
    public void stopWatchingActivities() {

application.unregisterActivityLifecycleCallbacks(lifecycleCallbacks);
    }
    // //就是 mActivityLifecycleCallbacks 的 移除
    public void
unregisterActivityLifecycleCallbacks(ActivityLifecycleCallbacks
callback) {
        synchronized (mActivityLifecycleCallbacks) {
            mActivityLifecycleCallbacks.remove(callback);
        }
    }
}

```

本质就是在 Activity 的 **onActivityDestroyed** 方法里 执行 **refWatcher.watch(activity);**

```

private final Application.ActivityLifecycleCallbacks
lifecycleCallbacks =
    new Application.ActivityLifecycleCallbacks() {
        @Override public void onActivityCreated(Activity activity, Bundle
savedInstanceState) {
        }

        @Override public void onActivityStarted(Activity activity) {
        }

        @Override public void onActivityResumed(Activity activity) {
        }
    }

```

```

        @Override public void onActivityPaused(Activity activity) {
        }

        @Override public void onActivityStopped(Activity activity) {
        }

        @Override public void onActivitySaveInstanceState(Activity
activity, Bundle outState) {
        }

        @Override public void onActivityDestroyed(Activity activity) {
            ActivityRefWatcher.this.onActivityDestroyed(activity);
        }
    };

    void onActivityDestroyed(Activity activity) {
        refWatcher.watch(activity);
    }

```

4、watch

```

public void watch(Object watchedReference, String referenceName) {
    checkNotNull(watchedReference, "watchedReference");
    checkNotNull(referenceName, "referenceName");
    if (debuggerControl.isDebuggerAttached()) {
        return;
    }
    //随机生成 watchedReference 的 key 保证其唯一性
    final long watchStartNanoTime = System.nanoTime();
    String key = UUID.randomUUID().toString();
    retainedKeys.add(key);
    //这个一个弱引用的子类拓展类 用于 我们之前所说的 watchedReference
和 queue 的联合使用
    final KeyedWeakReference reference =
        new KeyedWeakReference(watchedReference, key, referenceName,
queue);

    watchExecutor.execute(new Runnable() {
        @Override public void run() {
            //重要方法，确然是否 内存泄漏
            ensureGone(reference, watchStartNanoTime);
        }
    });
}

```

```

    }
    final class KeyedWeakReference extends WeakReference<Object> {
        public final String key;
        public final String name;

        KeyedWeakReference(Object referent, String key, String name,
            ReferenceQueue<Object> referenceQueue) {
            super(checkNotNull(referent, "referent"),
                checkNotNull(referenceQueue, "referenceQueue"));
            this.key = checkNotNull(key, "key");
            this.name = checkNotNull(name, "name");
        }
    }

```

5、ensureGone

```

    void ensureGone(KeyedWeakReference reference, long watchStartNanoTime)
    {
        long gcStartNanoTime = System.nanoTime();

        long watchDurationMs = NANOSECONDS.toMillis(gcStartNanoTime -
            watchStartNanoTime);
        //把 queue 的引用 根据 key 从 retainedKeys 中引出 。
        //retainedKeys 中剩下的就是没有分析和内存泄漏的引用的 key
        removeWeaklyReachableReferences();
        //如果内存没有泄漏 或者处于 debug 模式那么就直接返回
        if (gone(reference) || debuggerControl.isDebugEnabled()) {
            return;
        }
        //如果内存依旧没有被释放 那么在 GC 一次
        gcTrigger.runGc();
        //再次 清理下 retainedKeys
        removeWeaklyReachableReferences();
        //最后还有 就是说明内存泄漏了
        if (!gone(reference)) {
            long startDumpHeap = System.nanoTime();
            long gcDurationMs = NANOSECONDS.toMillis(startDumpHeap -
                gcStartNanoTime); //dump 出 Head 报告
            File heapDumpFile = heapDumper.dumpHeap();

            if (heapDumpFile == null) {
                // Could not dump the heap, abort.
                return;
            }
        }
    }

```

```

        long heapDumpDurationMs = NANSECONDS.toMillis(system.nanoTime() -
startDumpHeap);
        //最后进行分析 这份 HeapDump
        //LeakCanary 分析内存泄露用的是一个第三方工具 HAHA 别笑 真的是这
        个名字
        heapdumpListener.analyze(
            new HeapDump(heapDumpFile, reference.key, reference.name,
watchDurationMs, gcDurationMs,
                heapDumpDurationMs));
    }
}
private void removeWeaklyReachableReferences() {
    // WeakReferences are enqueued as soon as the object to which they
point to becomes weakly
    // reachable. This is before finalization or garbage collection has
actually happened.
    KeyedWeakReference ref;
    while ((ref = (KeyedWeakReference) queue.poll()) != null) {
        retainedKeys.remove(ref.key);
    }
}

private boolean gone(KeyedWeakReference reference) {
    return !retainedKeys.contains(reference.key);
}

```

6、[haha](#)

大家有兴趣可以分析下这个分析库的原理。在这里就不深入研究了。

最后把分析的引用链 写入文件中，发通知。

```

@TargetApi(HONEYCOMB) @Override
protected final void onHeapAnalyzed(HeapDump heapDump, AnalysisResult
result) {
    String leakInfo = leakInfo(this, heapDump, result);
    Log.d("LeakCanary", leakInfo);

    if (!result.leakFound || result.excludedLeak) {
        afterDefaultHandling(heapDump, result, leakInfo);
        return;
    }

    File leakDirectory = DisplayLeakActivity.leakDirectory(this);

```

```

        int maxStoredLeaks =
getResources().getInteger(R.integer.__leak_canary_max_stored_leaks);
        File renamedFile = findNextAvailableHprofFile(leakDirectory,
maxStoredLeaks);

        if (renamedFile == null) {
            // No file available.
            Log.e("LeakCanary",
                "Leak result dropped because we already store " + maxStoredLeaks
+ " leak traces.");
            afterDefaultHandling(heapDump, result, leakInfo);
            return;
        }

        heapDump = heapDump.renameFile(renamedFile);

        File resultFile = DisplayLeakActivity.leakResultFile(renamedFile);
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream(resultFile);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(heapDump);
            oos.writeObject(result);
        } catch (IOException e) {
            Log.e("LeakCanary", "Could not save leak analysis result to disk",
e);
            afterDefaultHandling(heapDump, result, leakInfo);
            return;
        } finally {
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException ignored) {
                }
            }
        }

        PendingIntent pendingIntent =
            DisplayLeakActivity.createPendingIntent(this,
heapDump.referenceKey);

        String contentTitle =
            getString(R.string.__leak_canary_class_has_leaked,
classSimpleName(result.className));

```

```

        String contentText =
getString(R.string.__leak_canary_notification_message);

        NotificationManager notificationManager =
            (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);

        Notification notification;
        if (SDK_INT < HONEYCOMB) {
            notification = new Notification();
            notification.icon = R.drawable.__leak_canary_notification;
            notification.when = System.currentTimeMillis();
            notification.flags |= Notification.FLAG_AUTO_CANCEL;
            notification.setLatestEventInfo(this, contentTitle, contentText,
pendingIntent);
        } else {
            Notification.Builder builder = new Notification.Builder(this) //
                .setSmallIcon(R.drawable.__leak_canary_notification)
                .setWhen(System.currentTimeMillis())
                .setContentTitle(contentTitle)
                .setContentText(contentText)
                .setAutoCancel(true)
                .setContentIntent(pendingIntent);
            if (SDK_INT < JELLY_BEAN) {
                notification = builder.getNotification();
            } else {
                notification = builder.build();
            }
        }
        notificationManager.notify(0xDEAFBEEF, notification);
        afterDefaultHandling(heapDump, result, leakInfo);
    }
}

```

总结

其实沿着源码分析很容易让人无法自拔，所以我们更要跳出来看到本质。

1、监听 Activity 的生命周期，在 onDestory 方法里调用 RefWatcher 的 watch 方法。
 watch 方法监控的是 Activity 对象
 2、给 Activity 的 Reference 生成唯一性的 key 添加到 retainedKeys 。生成 KeyedWeakReference 对象，Activity 的弱引用和 ReferenceQueue 关联。执行 ensureGone 方法。
 3、如果 retainedKeys 中没有 该 Reference 的 key 那么就说明没有内存泄漏。
 4、如果有，那么 analyze 分析我们 HeadDump 文件。建立导致泄漏的引用链。

5、引用链传递给 APP 进程的 DisplayLeakService，以通知的形式展示出来。

4. ARouter

前言

阅读本文之前，建议读者：

- 对 Arouter 的使用有一定的了解。
- 对 Apt 技术有所了解。

Arouter 是一款 Alibaba 出品的优秀的路由框架，本文不对其进行全面的分析，只对其最重要的功能进行源码以及思路分析，至于其拦截器，降级，ioc 等功能感兴趣的同学请自行阅读源码，强烈推荐阅读云栖社区的[官方介绍](#)。

对于一个框架的学习和讲解，我个人喜欢先将其最核心的思路用简单一两句话总结出来：
ARouter 通过 Apt 技术，生成保存路径(路由 path)和被注解(@Router)的组件类的映射关系的类，利用这些保存了映射关系的类，Arouter 根据用户的请求 postcard (明信片) 寻找到要跳转的目标地址(class),使用 Intent 跳转。

原理很简单，可以看出来，该框架的核心是利用 apt 生成的映射关系，这里要用到 Apt 技术，读者可以自行搜索了解一下。

分析

我们先看最简单的代码的使用：

首先需要在需要跳转的组件添加注解

```
@Route(path = "/main/homepage")public class HomeActivity extends BaseActivity {  
  
    onCreate()  
  
    ....  
  
}
```

然后在需要跳转的时候调用

```
Arouter.getInstance().build("main/hello").navigation;
```

这里的路径“main/hello”是用户唯一配置的东西，我们需要通过这个 path 找到对应的 Activity。最简单的思路就是通过 APT 技术，寻找到所有带有注解@Router 的组件，将其注解值 path 和对应的 Activity 保存到一个 map 里，比如像下面这样：

```
class RouterMap {  
  
    public Map getAllRoutes {
```

```

        Map map = new HashMap<String, Class<?>>;

        map.put("/main/homepage", HomeActivity.class);

        map.put("/main/setting", SettingActivity.class);

        map.put("/login/register", LoginRegisterActivity.class);

        ....

    return map;

}

}

```

然后在工程代码中将这个 **map** 加载到内存中，需要的时候直接 **get(path)** 就可以了，这种方案似乎能解决我们的问题。

发现问题

上面的思路确实能够实现路由功能，但是这么做会存在一个较大的问题：对于一个大型项目，组件数量会很多，可能会有一两百或者更多，把这么多组件都放到这个 **Map** 里，显然会对内存造成很大的压力，因此，**Arouter** 作为一款阿里出品的优秀框架，显然是要解决这个问题的。

这里建议读者自行思考一下，如何解决一次性加载所有映射关系带来的内存损耗问题，我在思考这个问题的时候首先想到的是“懒加载”，但是仅仅懒加载是不够的，因为懒加载后如果还是一次性把所有映射关系加载进来，内存损耗还是一样大的。因此，再深入思考一下，可能还能想出解决一个思路：分段懒加载，思路有了，如何实现呢？这里还是建议大家在阅读下面的内容之前思考一下，或许你能想到一套不同于 **Arouter** 的方案来哦。

Arouter 采用的方法就是“分组+按需加载”，分组还带来的另一个好处是便于管理，下面我们来看一下实现原理。

解决步骤一：分组

首先看如何分组的，**Arouter** 在一层 **map** 之外，增加了一层 **map**，我们看 **WareHouse** 这个类，里面有两个静态 **Map**：

```

static Map<String, Class<? extends IRouteGroup>> groupsIndex = new
HashMap<>();

static Map<String, RouteMeta> routes = new HashMap<>();

```

-

groupsIndex 保存了 **group** 名字到 **IRouteGroup** 类的映射，这一层映射就是 **Arouter** 新增的一层映射关系。

-

-

`routes` 保存了路径 `path` 到 `RouteMeta` 的映射，其中，`RouteMeta` 是目标地址的包装。这一层映射关系跟我们自己方案里的 `map` 是一致的，我们路径跳转也是要用这一层来映射。

•

这里出现了两个我们不认识的类，`IRouteGroup` 和 `RouteMeta`，后者很简单，就是对跳转目标的封装，我们后续称其为“目标”，其内包含了目标组件的信息，比如 `Activity` 的 `Class`。那 `IRouteGroup` 是个什么东西？

```
public interface IRouteGroup {  
  
    /**  
     * Fill the atlas with routes in group.  
     */  
  
    void loadInto(Map<String, RouteMeta> atlas);  
}
```

一个接口，只有一个方法 `loadInto`，都有谁实现了这个接口呢？我拿我手上的一个项目为例，`Arouter` 通过 `apt` 生成了下面几个类：



这几个类都以 `Arouter$$Group` 开头，我们随便拿一个看看：

```
public class ARouter$$Group$$$main implements IRouteGroup {  
  
    @Override  
  
    public void loadInto(Map<String, RouteMeta> atlas) {  
  
        atlas.put("/main/fa/leakscan", RouteMeta.build(RouteType.ACTIVITY,  
MainFaLeakScanActivity.class, "/main/fa/leakscan", "main", {}, -1, 1));  
  
        atlas.put("/main/login", RouteMeta.build(RouteType.ACTIVITY,  
LoginActivity.class, "/main/login", "main", null, -1, -2147483648));  
  
        atlas.put("/main/register", RouteMeta.build(RouteType.ACTIVITY,  
RegPhoneActivity.class, "/main/register", "main", null, -1, -2147483648));  
  
    }  
}
```

我们看到，他实现了 `loadInto` 方法，在这个方法中，它往这个 `HashMap` 中填充了好多数
据，填充的是什么呢？填充的是路径 `path` 和它对应的目标 `RouteMeta`，也就是我们最终

需要的那层映射关系。而且，我们还能观察到：这个类下面所有的路由 **path** 都有一个共同点，即全是“/main”开头的，也就是说，这个类加载的映射关系，都是在一个组内的。因此我们总结出：

Arouter 通过 **apt** 技术，为每个组生成了一个以 **Arouter\$\$Group** 开头的类，这个类负责向 **atlas** 这个 **HashMap** 中填充组内路由数据。

IRouteGroup 正如其名字，它就是一个能装载该组路由映射数据的类，其实有点像个工具类，为了方便后续讲解，我们姑且称上面这样一个实现了 **IRouteGroup** 的类叫做“组加载器”，本质是一个类。上图中的类是一个组加载器，其他所有以 **Arouter\$\$Group** 开头的类都是一个“组加载器”。回到之前的主线，**Warehoust** 中的两个 **HashMap**，其中 **groupsIndex** 这个 **map** 中保存的是什么呢？我们通过它的调用找到这一行代码(已简化)：

```
for (String className : routerMap) {  
  
    if (className.startsWith(ROUTE_ROOT_PACKAGE + DOT + SDK_NAME + SEPARATOR  
+SUFFIX_ROOT)) {  
  
        ((IRouteRoot)  
(Class.forName(className).getConstructor().newInstance()).loadInto(Warehous  
e.groupsIndex));  
  
    }  
  
}
```

其中 `ROUTE_ROOT_PACKAGE + DOT + SDK_NAME + SEPARATOR +SUFFIX_ROOT` 这行代码是几个静态字符串拼起来的，它等于 `com.alibaba.android.arouter.routes.Arouter$$Root`。另外 **routerMap** 是什么呢？它是一个 **HashSet<String>**：

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

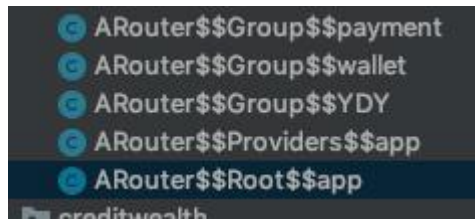
这一行代码对它进行了初始化，目的是找到 `com.alibaba.android.arouter.routes` 这个包名下所有的类，将其类名保存到 **routerMap** 中。因此，上面的代码意思就是将 `com.alibaba.android.arouter.routes` 包下所有名字

以 `com.alibaba.android.arouter.routes.Arouter$$Root` 开头的类找出来，通过反射实例化并强转成 **IRouteRoot**，然后调用 **loadInto** 方法。这里又出来一个新的接口：

IRouteRoot，我们看代码：

```
public interface IRouteRoot {  
  
    /**  
        * Load routes to input  
        * @param routes input  
        */  
  
    void loadInto(Map<String, Class<? extends IRouteGroup>> routes);  
  
}
```

跟 `IRouteGroup` 长得还挺像，也是 `loadInto`，我们看它的实现。还是以我的项目为例，在 `apt` 生成的文件夹下查找：



最底下一行，有个 `Arouter$$Root$$app`，它符合前面名字规则，我们进去看看：

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

这个类实现了 `IRouteRoot`，在 `loadInto` 方法中，他将组名和组对应的“组加载器”保存到了 `routes` 这个 `map` 中。也就是说，这个类将所有的“组加载器”给索引了下来，通过任意一个组名，可以找到对应的“组加载器”，我们再回到前面讲的初始化 `Arouter` 时候的方法中：

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

理解了吧，这个方法的意义就在于将所有的组路由加载类索引到了 `groupsIndex` 这个 `map` 中。因此我们就明白了：

Warehouse 中的 `groupsIndex` 保存的是组名到“组加载器”的映射关系

说句题外话：回过头想想前面用到的两个接口：`IRouteGroup` 和 `IRouteRoot`，它们其实是 `apt` 生成的类和我们项目中代码之间沟通的桥梁，熟悉 `AIDL` 的同学可能会觉得很熟悉，二者其实是异曲同工的，两个系统进行交互的时候都是通过接口来沟通的。当然，在使用 `apt` 生成的类时，我们需要用到反射技术。

总结一下 `Arouter` 的分组设计：`Arouter` 在原来 `path` 到目标的 `map` 外，加了一个新的 `map`，该 `map` 保存了组名到“组加载器”的映射关系。其中“组加载器”是一个类，可以加载其组内的 `path` 到目标的映射关系。

到此为止，**Arouter** 只是完成了分组工作，但这么做的目的是什么呢？别着急，前面的都只是铺垫，接下来才是这个分组设计发挥作用的地方，我们进入“按需加载”的代码分析：

解决步骤二：按需加载

之前说过，**Arouter** 使用的是分组按需加载，分组是为了按需做准备的。我们看 **Arouter** 是怎么按需加载的，我们还是从代码的使用入手：

```
Arouter.getInstance().build("main/hello").navigation;
```

在 **navigation** 这个方法中，最终会跳转到这里：

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

```
    try {
```

```
        //请关注这一行
```

```
        LogisticsCenter.completion(postcard);
```

```
    } catch (NoRouteFoundException ex) {
```

```
        logger.warning(Constants.TAG, ex.getMessage());
```

```
        ....//简化代码
```

```
    }
```

```
    //调用 Intent 跳转
```

```
    return navigation(context, postcard, requestCode, callback)
```

最后一行的 **return** 语句很简单，就是去调用 **Intent** 唤起组件了，我们看前面 **try** 中的第一行 `LogisticsCenter.completion(postcard)`，我们进到这个函数里：

```
//从缓存里取路由信息
```

```
RouteMeta routeMeta = Warehouse.routes.get(postcard.getPath()); //如果为空，需要
加载该组的路由 if (null == routeMeta) {
```

```
    Class<? extends IRouteGroup> groupMeta =
```

```
Warehouse.groupsIndex.get(postcard.getGroup());
```

```
IRouteGroup iGroupInstance = groupMeta.getConstructor().newInstance();
```

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

```
Warehouse.groupsIndex.remove(postcard.getGroup());
```

```
} //如果不为空，走后续流程 else {
```

```
    postcard.setDestination(routeMeta.getDestination());
```

```
    ...
```

```
}
```

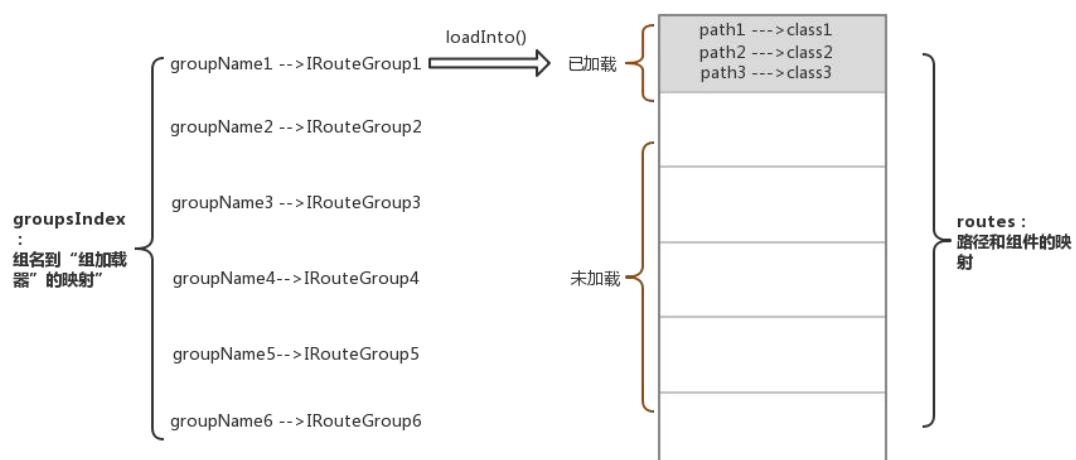
这段代码就是“按需加载”的核心逻辑所在了，我对其进行了简化，分析其逻辑是这样的：

- 首先从 `Warehouse.routes`(前面说了，这里存放的是 `path` 到目标的映射)里拿到目标信息，如果找不到，说明这个信息还没加载，需要加载，实际上，刚开始这个 `routes` 里面什么都没有。
- 加载流程：首先从 `Warehouse.groupsIndex` 里获取“组加载器”，组加载器是一个类，需要通过反射将其实例化，实例化为 `iGroupInstance`，接着调用组加载器的加载方法 `loadInto`，将该组的路由映射关系加载到 `Warehouse.routes` 中，加载完成后，`routes` 中就缓存下来当前组的所有路由映射了，因此这个组加载器其实就没用了，为了节省内存，将其从 `Warehouse.groupsIndex` 移除。
- 如果之前加载过，则在 `Warehouse.routes` 里面是可以找到路由映射关系的，因此直接将目标信息 `routeMeta` 传递给 `postcard`，保存在 `postcard` 中，这样 `postcard` 就知道了最终要去哪个组件了。

到此为止分组按需加载的逻辑就都分析完了，通过这两个步骤，解决了路由映射一次性加载到内存占用内存过大的缺点，这是 `Arouter` 这个框架优秀的重要原因之一。当然 `Arouter` 还有一些优秀的功能，比如拦截器，依赖注入等，总之，功能全，性能好，使用方便，这些都是 `Arouter` 受欢迎的原因，这点值得我们所有开发者去学习。

总结

最后结合一张图总结一下 `Arouter` 的分组按需加载的逻辑：



图中左侧 `groupsIndex` 是“组映射”，右侧 `routes` 是“路由映射”。`Arouter` 在初始化的时候，通过反射技术，将所有的“组加载器”索引到 `groupsIndex` 这个 `map` 中，而此时，右侧的 `routes` 还是空的。在用户调用 `navigation()` 进行跳转的时候，会根据路径提取组名，由组名根据 `groupsIndex` 获取到相应组的“组加载器”，由组加载器加载对应组内的路由信息，此时保存全局“路由由目标映射的”`routes` 这个 `map` 中就保存了刚才组的所有路由映射关系了。同样，当其他组请求时，其他组也会加载组对应的路由映射，这样就实现了整个 `App` 运行时，只有用到的组才会加到内存中，没有去过的组就不会加载到内存中，达到了节省内存的目的。

5. 插件化（不同插件化机制原理与流派，优缺点。局限性）

1.动态加载 so 库（其实可以放在外部存储，我们常用的是放在内部存储）

2.classloader 动态加载外部可执行文件，如 dex,jar,apk

关于动态加载，细分为三种：

第一，简单的动态加载；这种不适用插件的 `activity`。使用插件的 `fragment`，或者只是用 `dex` 中的逻辑，或者用代码编写布局的方式替换布局。实现起来比较简单，比较稳定，但是有限制，适用于界面改动小，或者不改动，逻辑 `sdk`，登录注册界面等等。

第二，静态代理 `activity`；这种是使用插件的 `activity`，但是实际上使用的是 `host` 的 `activity`。使用插件的 `activity`，需要解决两个问题，

一是生命周期，

关于生命周期，通过在 `host` 提前注册好需要的 `activity`，然后拿到插件 `activity` 引用（两种方式，通过反射拿到，再则通过把插件 `activity` 接口化，通过接口调用的方式，更推荐这种，方便 `host` 控制插件），在各个生命周期方法中同步调用。

二是 `R` 资源。

关于 `R` 资源，需要通过反射拿到 `AssertManager`,调用其 `addAssertPath`，得到 `Resource` 资源对象。相当于多维护了一套 `R` 资源。实际上，我们平日用的就有两套 `R` 资源，应用层的一套，然后系统层的一套。

第三，动态注册 `activity`

关于动态注册 `activity`，他解决生命周期和 `R` 资源的方式就更简单了，相当于是通过标准模式启动了一个 `activity`，自然而然该 `activity` 就具备了生命周期和 `R` 资源。那么这种该如何做呢？

首先，host 中，要有一个通用的 `PluginActivity`，当要启动 `PluginActivity` 的时候，我们复写 `ClassLoader` 的 `loadClass` 方法，当要找的事 `PluginActivity`，我们通过 `DexMaker` 生成一个 `TargetActivity extends PluginActivity`。然后 `return TargetActivity` 实例。这样就完美解决了生命周期和 R 资源的问题。

如果插件里面需要有多多个 `activity`，如何从 `TargetActivity` 中跳转到其他 `Activity` 呢，实际上我们知道所有的跳转都会走到 `startActivityForResult`，我们只需要在用 `DexMaker` 生成 `TargetActivity` 的时候，复写对应的 `startActivityForResult` 方法就行了。

6. 热修复

。首先看下 Demo 里面 `Application` 的代码。

```

13  * sample application
14  *
15  */
16  public class MainApplication extends Application {
17      private static final String TAG = "andrew";
18
19      private static final String APATCH_PATH = "/out.apatch";
20
21      private static final String DIR = "apatch"; //补丁文件夹
22      /**
23       * patch manager
24       */
25      private PatchManager mPatchManager;
26
27      @Override
28      public void onCreate() {
29          super.onCreate();
30          // initialize
31          mPatchManager = new PatchManager(this);
32          mPatchManager.init("1.0");
33          Log.d(TAG, "inited.")http://blog.csdn.net/
34
35          // load patch
36          mPatchManager.loadPatch();
37          try {
38              // .apatch file path
39              String patchFileString = Environment.getExternalStorageDirectory()
40                  .getAbsolutePath() + APATCH_PATH;
41              mPatchManager.addPatch(patchFileString);
42              Log.d(TAG, "apatch:" + patchFileString + " added.");
43
44              //复制且加载补丁成功后，删除下载的补丁
45              File f = new File(this.getFilesDir(), DIR + APATCH_PATH);
46              if (f.exists()) {
47                  boolean result = new File(patchFileString).delete();
48                  if (!result)
49                      Log.e(TAG, patchFileString + " delete fail");
50              }
51          } catch (IOException e) {
52              Log.e(TAG, "", e);
53          }
54      }
55  }

```

2.一开始实例化 PatchManager。然后调用 init()这种方法，我们跟进去看看。

我凝视的非常具体，大致就是从 SharedPreferences 读取曾经存的版本号和你传过来的版本号进行比对，假设两者版本号不一致就删除本地 patch。否则调用 initPatches()这种方法。

```

/**
 * initialize
 *
 * @param appVersion App version
 */
public void init(String appVersion) {
    if (!mPatchDir.exists() && !mPatchDir.mkdirs()) { // make directory fail
        Log.e(TAG, "patch dir create error.");
        return;
    } else if (!mPatchDir.isDirectory()) { //如果遇到同名的文件，则将该同名文件删除
        mPatchDir.delete();
        return;
    }
    //在该文件下放入一个名为_andfix_的SharedPreferences文件
    SharedPreferences sp = mContext.getSharedPreferences(SP_NAME,
        Context.MODE_PRIVATE); //存储关于patch文件的信息
    //根据你传入的版本号和之前的对比，做不同的处理
    String ver = sp.getString(SP_VERSION, null);
    //如果从_andfix_这个文件获取的ver不是null，而且这个ver和外部初始化时传进来的版本号一致
    if (ver == null || !ver.equalsIgnoreCase(appVersion)) {
        cleanPatch(); //删除本地patch文件
        sp.edit().putString(SP_VERSION, appVersion).commit(); //并把传入的版本号保存
    } else {
        initPatches(); //mPatchDir文件夹下的文件作为参数传给了addPatch(File)方法
    }
}
}

```

3.分析下 initPatches()它做了什么，事实上代码非常 easy，就是把 mPatchDir 目录下的文件作为参数传给了 addPatch(File)方法，然后调用 addPatch()方法。addPatch 方法的作用看以下的凝视，写的非常清楚。

```

private void initPatches() {
    File[] files = mPatchDir.listFiles();
    for (File file : files) {
        addPatch(file);
    }
}

/**
 * add patch file
 *
 * @param file
 * @return patch
 */
//把扩展名为.apatch的文件传给Patch做参数，初始化对应的Patch，
//并把刚初始化的Patch加入到我们之前看到的Patch集合mPatches中
private Patch addPatch(File file) {
    Patch patch = null;
    if (file.getName().endsWith(SUFFIX)) {
        try {
            patch = new Patch(file); //实例化Patch对象
            mPatches.add(patch); //把patch实例存储到内存的集合中，在PatchManager实例化集合
        } catch (IOException e) {
            Log.e(TAG, "addPatch", e);
        }
    }
    return patch;
}

```

4. 我们能够看到 addPatch() 方法里面会实例化 Patch，我们跟进去看看实例化过程中，它又干了什么事。

```
AndFixDemo AndFix src com alipay euler andfix patch Patch
Patch.java
* classes of patch
*/
private Map<String, List<String>> mClassesMap;

public Patch(File file) throws IOException {
    mFile = file;
    init();
}

/deprecation/
private void init() throws IOException {
    JarFile jarFile = null;
    InputStream inputStream = null;
    try {
        jarFile = new JarFile(mFile); // 使用JarFile读取Patch文件
        JarEntry entry = jarFile.getJarEntry(ENTRY_NAME); // 获取META-INF/PATCH.MF文件
        inputStream = jarFile.getInputStream(entry);
        Manifest manifest = new Manifest(inputStream);
        Attributes main = manifest.getMainAttributes();
        mName = main.getValue(PATCH_NAME); // 获取PATCH.MF属性Patch-Name
        mTime = new Date(main.getValue(CREATED_TIME)); // 获取PATCH.MF属性Created-Time

        mClassesMap = new HashMap<String, List<String>>();
        Attributes.Name attrName;
        String name;
        List<String> strings;
        for (Iterator<?> it = main.keySet().iterator(); it.hasNext(); ) {
            attrName = (Attributes.Name) it.next();
            name = attrName.toString();
            // 判断name的后缀是否是-Classes，并把name对应的值加入到集合中，对应的值就是class类名的列表
            if (name.endsWith(CLASSES)) {
                strings = Arrays.asList(main.getValue(attrName).split(","));
                if (name.equalsIgnoreCase(PATCH_CLASSES)) {
                    mClassesMap.put(mName, strings);
                } else {
                    mClassesMap.put(
                        name.trim().substring(0, name.length() - 8), // remove
                        // "-Classes"
                        strings);
                }
            }
        }
    }
}
```

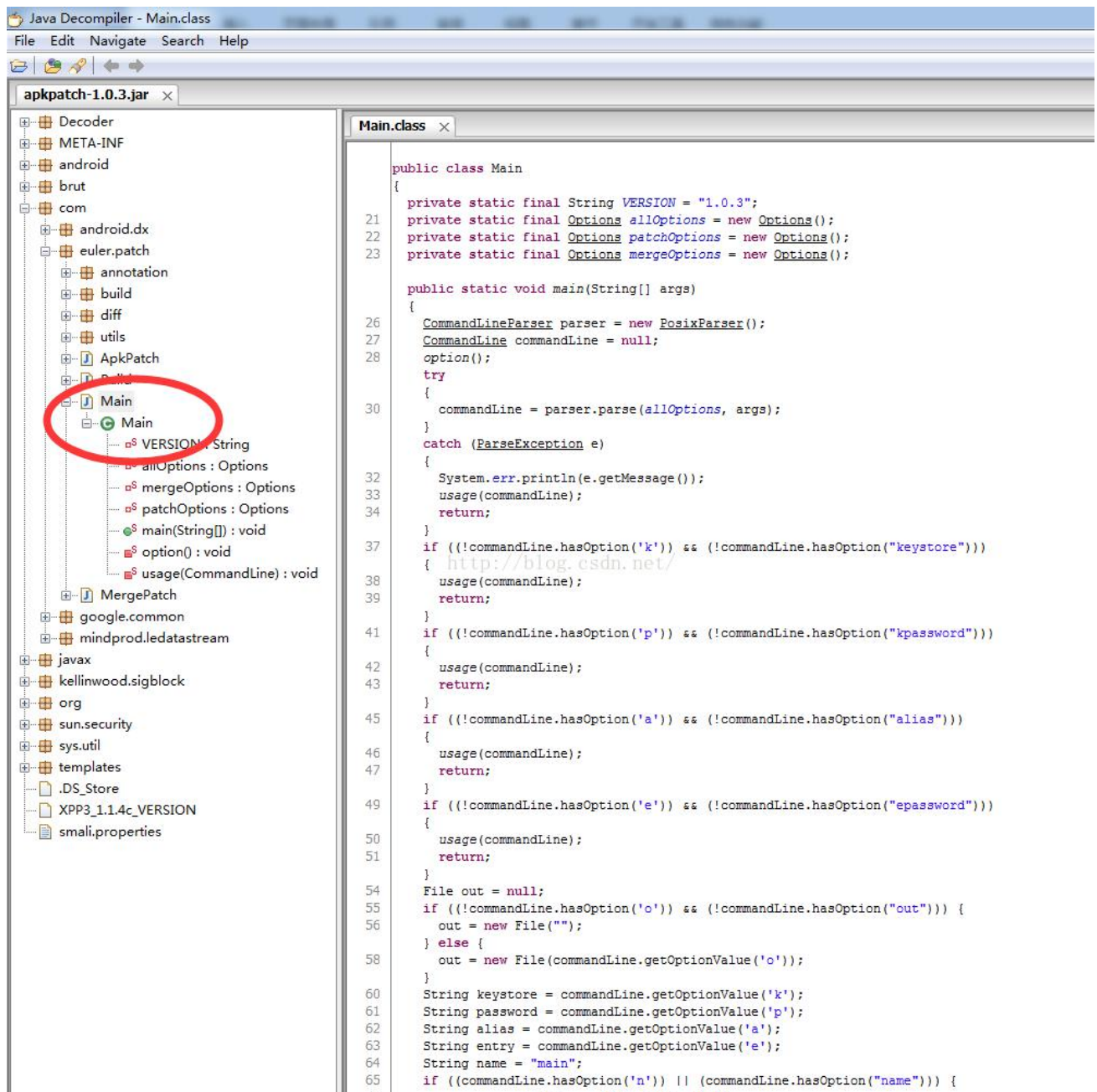
它里面调用了 `init()` 方法，能够看到里面有

`JarFile`, `JarEntry`, `Manifest`, `Attributes`, 通过它们一层层的从 `Jar` 文件里获取对应的值，提到这里大家可能会奇怪,明明是 `.patch` 文件，怎么又变成 `Jar` 文件了？事实上是通过阿里打补丁包工具生成补丁的时候写入对应的值。补丁文件事实上就相等于 `jar` 包。仅仅只是它们的扩展名

不同而已。提到这里我们就来单独的探索下，补丁文件是怎么一步步生成的。由于阿里没有对打补丁工具进行加密和混淆。我们能够使用 **jdgui** 打开查看。

所需对应的工具代码 **demo** 等我都统一放在以下的下载链接里面。有须要可自行取下。

5.好了，我们如今来分析下补丁文件怎样生成的，用 **jdgui** 打开 **apkpatch-1.0.3**。先从 **main** 方法開始。



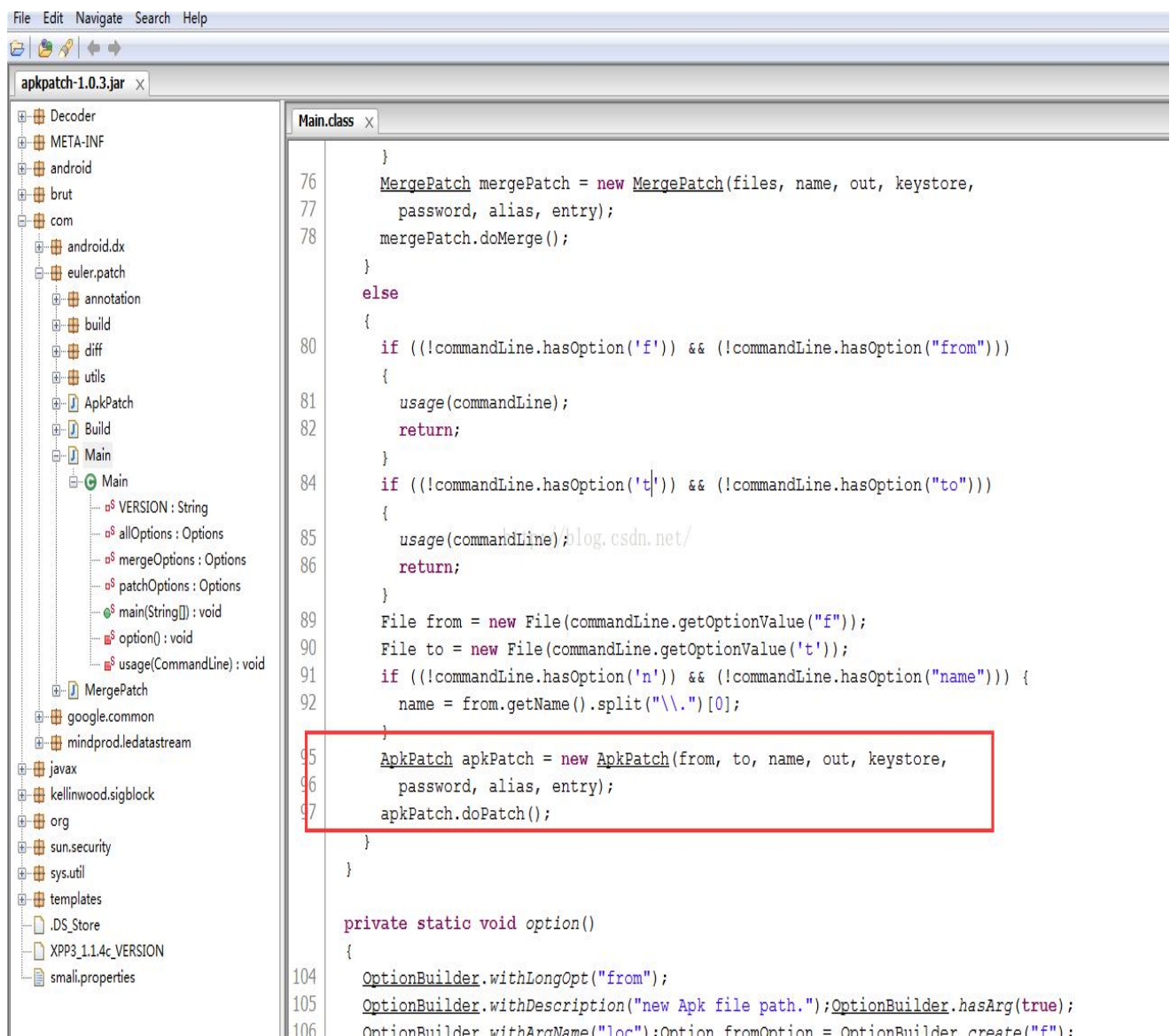
能够看到：下图 1 部分就是我们前面介绍怎样使用命令行打补丁包的命令，检查命令行是否有那些参数。假设没有要求的参数，就给用户对应的提示。

第二部分。我们在打正式包的时候，会指定 keystore, password, alias, entry 相关参数。

另外 name 就是最后生成的文件，能够忽略。

```
Main.class x
{
32     System.err.println(e.getMessage());
33     usage(commandLine);
34     return;
35 }
37 if ((!commandLine.hasOption('k')) && (!commandLine.hasOption("keystore")))
38 {
39     usage(commandLine);
40     return;
41 }
42 if ((!commandLine.hasOption('p')) && (!commandLine.hasOption("kpassword")))
43 {
44     usage(commandLine);
45     return;
46 }
47 if ((!commandLine.hasOption('a')) && (!commandLine.hasOption("alias")))
48 {
49     usage(commandLine);
50     return;
51 }
52 if ((!commandLine.hasOption('e')) && (!commandLine.hasOption("epassword")))
53 {
54     usage(commandLine);
55     return;
56 }
57 File out = null;
58 if ((!commandLine.hasOption('o')) && (!commandLine.hasOption("out"))) {
59     out = new File("");
60 } else {
61     out = new File(commandLine.getOptionValue('o'));
62 }
63 String keystore = commandLine.getOptionValue('k');
64 String password = commandLine.getOptionValue('p');
65 String alias = commandLine.getOptionValue('a');
66 String entry = commandLine.getOptionValue('e');
67 String name = "main";
68 if ((commandLine.hasOption('n')) || (commandLine.hasOption("name"))) {
69     name = commandLine.getOptionValue('n');
70 }
71 if ((commandLine.hasOption('m')) || (commandLine.hasOption("merge")))
72 {
73     String[] merges = commandLine.getOptionValues('m');
74 }
```

Main 函数最后一个方法是我们的大头戏。上面的参数传给 ApkPatch 进行初始化。然后调用 doPatch()方法。



我们再跟进去。看看 **ApkPatch** 初始化的过程中，做了什么。

```
Main.class  ApkPatch.class x
package com.euler.patch;

import brut.androlib.mod.SmaliMod;

public class ApkPatch
    extends Build
{
    private File from;
    private File to;
    private Set<String> classes;

    public ApkPatch(File from, File to, String name, File out, String keystore, String password, String alias, String entry)
    {
        super(name, out, keystore, password, alias, entry);
        this.from = from;
        this.to = to;
    }

    public void doPatch()
    {
        try
        {
            File smaliDir = new File(this.out, "smali");

```

调用了父类的方法，我们再看看父类 **Build**.

```
package com.euler.patch;

import com.euler.patch.build.PatchBuilder;

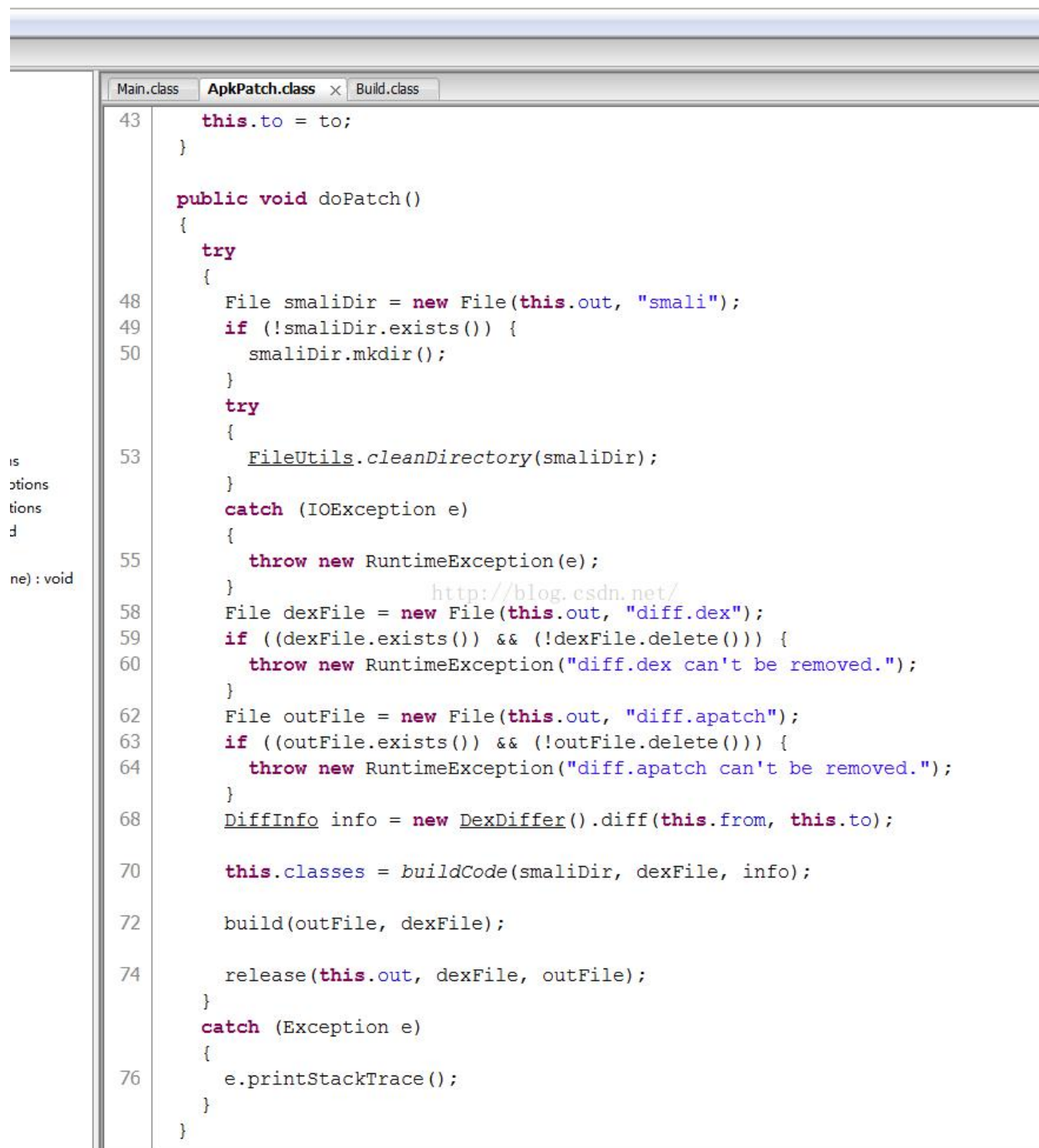
public abstract class Build
{
    protected static final String SUFFIX = ".apatch";
    protected String name;
    private String keystore;
    private String password;
    private String alias;
    private String entry;
    protected File out;

    public Build(String name, File out, String keystore, String password, String alias, String entry)
    {
        this.name = name;
        this.out = out;
        this.keystore = keystore;
        this.password = password;
        this.alias = alias;
        this.entry = entry;
        if (!out.exists()) {
            out.mkdirs();
        } else if (!out.isDirectory()) {
            throw new RuntimeException("output path must be directory.");
        }
    }
}
```

干的事情事实上比较简单。就是给变量进行赋值。能够看到 out,我们的输出文件就是这么来的,没有的话,它会自己创建一个。

然后我们再回到 apkPatch.doPatch()这种方法。

看看这种方法里面是什么。



这种方法主要做的就是在我们的 out 输出文件里生成一个 smali 目录，还有 diff.dex, diff.apatch 文件。

能够找到你的输出文件确认下。

```
}  
DiffInfo info = new DexDiffer().diff(this.from, this.to);  
  
this.classes = buildCode(smaliDir, dexFile, info);  
                http://blog.csdn.net/  
build(outFile, dexFile);  
  
release(this.out, dexFile, outFile);
```

DiffInfo 相当于一个存储新包和旧包差异信息的容器来，通过 diff 方法将二者的差异信息给 info.然后就是三个最重要的方法，buildCode(), build(),release()。我们接下来一个个的看下，他们到底为何这么重要。

```
Main.class  ApkPatch.class  Build.class
76      e.printStackTrace();
      }
  }

  private static Set<String> buildCode(File smaliDir, File dexFile, DiffInfo info)
      throws IOException, RecognitionException, FileNotFoundException
  {
83      Set<String> classes = new HashSet();
84      Set<DexBackedClassDef> list = new HashSet();
85      list.addAll(info.getAddedClasses());
86      list.addAll(info.getModifiedClasses());

88      baksmaliOptions options = new baksmaliOptions();

90      options.deodex = false;
91      options.noParameterRegisters = false;
92      options.useLocalsDirective = true;
93      options.useSequentialLabels = true;
94      options.outputDebugInfo = true;
95      options.addCodeOffsets = false;
96      options.jobs = -1;
97      options.noAccessorComments = false;
98      options.registerInfo = 0;
99      options.ignoreErrors = false;
100     options.inlineResolver = null;
101     options.checkPackagePrivateAccess = false;
102     if (!options.noAccessorComments) {
103         options.syntheticAccessorResolver = new SyntheticAccessorResolver(
104             list);
    }

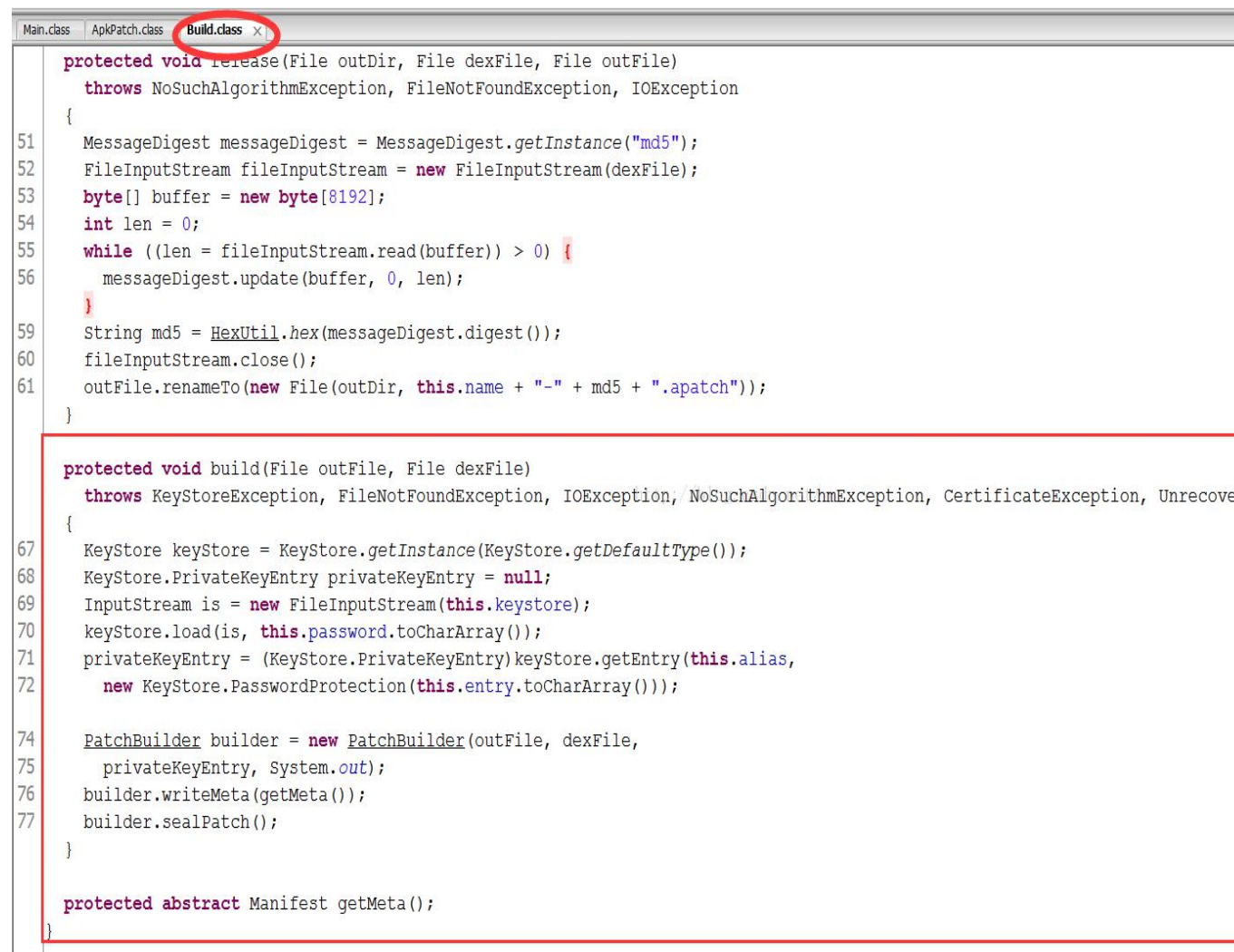
106     ClassFileNameHandler outFileNameHandler = new ClassFileNameHandler(
107         smaliDir, ".smali");
108     ClassFileNameHandler inFileNameHandler = new ClassFileNameHandler(
109         smaliDir, ".smali");
110     DexBuilder dexBuilder = DexBuilder.makeDexBuilder();
111     for (DexBackedClassDef classDef : list)
112     {
114         String className = classDef.getType();
115         baksmali.disassembleClass(classDef, outFileNameHandler, options);
116         File smaliFile = inFileNameHandler.getUniqueFilenameForClass(
117             TypeGenUtil.newType(className));
118         classes.add(TypeGenUtil.newType(className)
119             .substring(1, TypeGenUtil.newType(className).length() - 1)
120             .replace('/', '.'));
121         SmaliMod.assembleSmaliFile(smaliFile, dexBuilder, true, true);
    }
124     dexBuilder.writeTo(new FileDataStore(dexFile));

126     return classes;
  }
```

看到 baksmali 和 smali，反编译过 apk 的同学一定不陌生，这就是 dex 的打包工具和解包工具。关于这个详细就不深入了，有兴趣的同学能够深入了解下。这种方法的返回值将 DiffInfo 中新加入的 classes

和改动过的 `classes` 做了一个重命名。然后保存了起来，同一时候，将相关内容写入 `smali` 文件里。

为什么要进行重命名，事实上是为了防止和之前安装的 `Dex` 文件名称冲突。



```
protected void release(File outDir, File dexFile, File outFile)
    throws NoSuchAlgorithmException, FileNotFoundException, IOException
{
    MessageDigest messageDigest = MessageDigest.getInstance("md5");
    FileInputStream fileInputStream = new FileInputStream(dexFile);
    byte[] buffer = new byte[8192];
    int len = 0;
    while ((len = fileInputStream.read(buffer)) > 0) {
        messageDigest.update(buffer, 0, len);
    }
    String md5 = HexUtil.hex(messageDigest.digest());
    fileInputStream.close();
    outFile.renameTo(new File(outDir, this.name + "-" + md5 + ".apatch"));
}

protected void build(File outFile, File dexFile)
    throws KeyStoreException, FileNotFoundException, IOException, NoSuchAlgorithmException, CertificateException, UnrecoverableException
{
    KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
    KeyStore.PrivateKeyEntry privateKeyEntry = null;
    InputStream is = new FileInputStream(this.keyStore);
    keyStore.load(is, this.password.toCharArray());
    privateKeyEntry = (KeyStore.PrivateKeyEntry) keyStore.getEntry(this.alias,
        new KeyStore.PasswordProtection(this.entry.toCharArray()));

    PatchBuilder builder = new PatchBuilder(outFile, dexFile,
        privateKeyEntry, System.out);
    builder.writeMeta(getMeta());
    builder.sealPatch();
}

protected abstract Manifest getMeta();
}
```

接下来看看 `build(outFile, dexFile)`，首先从 `keystone` 里面获取应用相关签名，将 `getMeta()` 中获取的 `Manifest` 内容写入 "META-INF/PATCH.MF" 文件里。 `getMeta()` 方法上面，实例化 `PatchBuilder`，然后调用 `writeMeta(getMeta())`。我们走进去先看看。

```
x
Main.class  ApkPatch.class  Build.class  PatchBuilder.class x
package com.euler.patch.build;

import java.io.File;

public class PatchBuilder
{
    private SignedJarBuilder mBuilder;

    public PatchBuilder(File outFile, File dexFile, KeyStore.PrivateKeyEntry key, PrintStream verboseStream)
    {
        try
        {
22         this.mBuilder = new SignedJarBuilder(new FileOutputStream(outFile, false), key.getPrivateKey(),
24             (X509Certificate)key.getCertificate());
25         this.mBuilder.writeFile(dexFile, "classes.dex");
        }
        catch (Exception e)
        {
27         e.printStackTrace();
        }
    }
}
```

这个就是将 dexFile 和签名相关信息写入 classes.dex 文件里。能够有点蒙。

我们就看看 writeFile()方法。


```

public SignedJarBuilder(OutputStream out, PrivateKey key, X509Certificate certi
    throws IOException, NoSuchAlgorithmException
{
    this.mOutputJar = new JarOutputStream(new BufferedOutputStream(out));
    this.mOutputJar.setLevel(9);
    this.mKey = key;
    this.mCertificate = certificate;
    if ((this.mKey != null) && (this.mCertificate != null))
    {
        this.mManifest = new Manifest();
        Attributes main = this.mManifest.getMainAttributes();
        main.putValue("Manifest-Version", "1.0");
        main.putValue("Created-By", "1.0 (ApkPatch)");
        this.mBase64Encoder = new BASE64Encoder();
        this.mMessageDigest = MessageDigest.getInstance("SHA1");
    }
}
}
http://blog.csdn.net/

public void writeFile(File inputFile, String jarPath)
    throws IOException
{
    FileInputStream fis = new FileInputStream(inputFile);
    try
    {
        JarEntry entry = new JarEntry(jarPath);
        entry.setTime(inputFile.lastModified());
        writeEntry(fis, entry);
    }
    finally
    {
        fis.close();
    }
}
}

```

SignedJarBuilder 的构造方法做了一些初始化和赋值操作。提到这个
是方便可以理解 writeFile()这种方法。

writeFile 里面调用了 writeEntry(),我们看看它。

```

Main.class  ApkPatch.class  Build.class  PatchBuilder.class  SignedJarBuilder.class x
238         this.mOutputJar.close();
        }
        catch (IOException localIOException) {}
    }

    private void writeEntry(InputStream input, JarEntry entry)
        throws IOException
    {
252         this.mOutputJar.putNextEntry(entry);
        int count;
255         while ((count = input.read(this.mBuffer)) != -1)
        {
            int count;
256             this.mOutputJar.write(this.mBuffer, 0, count);
258             if (this.mMessageDigest != null) {
259                 this.mMessageDigest.update(this.mBuffer, 0, count);
            }
        }
263         this.mOutputJar.closeEntry();
264         if (this.mManifest != null)
        {
266             Attributes attr = this.mManifest.getAttributes(entry.getName());
267             if (attr == null)
            {
268                 attr = new Attributes();
269                 this.mManifest.getEntries().put(entry.getName(), attr);
            }
271             attr.putValue("SHA1-Digest", this.mBase64Encoder.encode(this.mMessageDigest.digest()));
        }
    }
}
```

这种方法就是从 input 输入流中读取 buffer 数据然后写入到 entry。然后联系到我上面提到的将 dexfile 和签名相关信息写入到 classes.dex 里面。应该能好理解点。

上面提了一大堆，我们的东西准备的差点儿相同了，如今就看看最后一个方法 `ApkPatch release(this.out, dexFile, outFile)`

```

main.class  ApkPatch.class  Build.class x PatchBuilder.class  SignedJarBuilder.class
5      throw new RuntimeException("output path must be directory.");
      }
    }

    protected void release(File outDir, File dexFile, File outFile)
        throws NoSuchAlgorithmException, FileNotFoundException, IOException
    {
1      MessageDigest messageDigest = MessageDigest.getInstance("md5");
2      FileInputStream fileInputStream = new FileInputStream(dexFile);
3      byte[] buffer = new byte[8192];
4      int len = 0;
5      while ((len = fileInputStream.read(buffer)) > 0) {
6          messageDigest.update(buffer, 0, len);
7      }
9      String md5 = HexUtil.hex(messageDigest.digest());
0      fileInputStream.close();
1      outFile.renameTo(new File(outDir, this.name + "-" + md5 + ".apatch"));
    }

```

这种方法就是将 dexFile 进行 md5 加密，把 build(outFile, dexFile); 函数中生成的 outFile 重命名。哈哈。看到“.patch”有没有非常激动！

！

我们的补丁包一开始的命名就是一长串。好了，到这里，补丁文件就生成了。接下来我们看看，怎么来使用它。

坚持就是胜利，立即你就要熬过头了...没办法。别人团队花了这么长时间做的，想分析就得花时间。

相关资料工具及 demo 下载地址：

<http://pan.baidu.com/s/1hsdcs7a>

7. RxJava （RxJava 的线程切换原理）

前言

很多项目使用流行的 Rxjava2 + Retrofit 搭建网络框架，Rxjava 现在已经发展到 Rxjava2，之前一直都只是再用 Rxjava，但从来没有了解下 Rxjava 的内部实现，接下来一步步来分析 Rxjava2 的源码，Rxjava2 分 Observable 和 Flowable 两种（无被压和有被压），我们今天先从简单的无背压的 observable 来分析。源码基于 rxjava:2.1.1。

一、Rxjava 如何创建事件源、发射事件、何时发射事件、如何将观察者和被观察者关联起来

简单的例子

先来段最简单的代码，直观的了解下整个 Rxjava 运行的完整流程。

```
1 private void doSomeWork() { 2     Observable<String> observable =
Observable.create(new ObservableOnSubscribe<String>() { 3         @Override 4
public void subscribe(Observer<String> e) throws Exception { 5
e.onNext("a"); 6         e.onComplete(); 7         } 8     }); 9
Observer observer = new Observer<String>() {10 11         @Override12
public void onSubscribe(Disposable d) {13             Log.i("lx", "
onSubscribe : " + d.isDisposed());14         }15 16         @Override17
public void onNext(String str) {18             Log.i("lx", " onNext : " +
str);19         }20 21         @Override22         public void
onError(Throwable e) {23             Log.i("lx", " onError : " +
e.getMessage());24         }25 26         @Override27         public
void onComplete() {28             Log.i("lx", "
onComplete");29         }30     };31
observable.subscribe(observer);32 }
```

上面代码之所以将 observable 和 observer 单独声明，最后再调用 observable.subscribe(observer); 是为了分步来分析：

1. 被观察者 Observable 如何生产事件的
2. 被观察者 Observable 何时生产事件的
3. 观察者 Observer 是何时接收到上游事件的
4. Observable 与 Observer 是如何关联在一起的

Observable

Observable 是数据的上游，即事件生产者


首先来分析事件是如何生成的，直接看代码 Observable.create()方法。

```

1 @SchedulerSupport (SchedulerSupport.NONE) 2     public static <T> Observable<T>
create(ObservableOnSubscribe<T> source) {    // ObservableOnSubscribe 是个接口,
只包含 subscribe 方法, 是事件生产的源头。3         ObjectHelper.requireNonNull(source,
"source is null"); // 判空 4         return RxJavaPlugins.onAssembly(new
ObservableCreate<T>(source));5     }

```


最重要的是 `RxJavaPlugins.onAssembly(new ObservableCreate<T>(source));` 这句代码。继续跟踪进去



```

1 /** 2     * Calls the associated hook function. 3     * @param <T> the value
type 4     * @param source the hook's input value 5     * @return the value
returned by the hook 6     */ 7     @SuppressWarnings({ "rawtypes", "unchecked" })
8     @NonNull 9     public static <T> Observable<T> onAssembly(@NonNull
Observable<T> source) {10         Function<? super Observable, ? extends
Observable> f = onObservableAssembly;11         if (f != null) {12
return apply(f, source);13         }14         return source;15     }

```



看注释, 原来这个方法是个 **hook function**。通过调试得知静态对象 `onObservableAssembly` 默认为 `null`, 所以此方法直接返回传入的参数 `source`。

`onObservableAssembly` 可以通过静态方法 `RxJavaPlugins.setOnObservableAssembly()` 设置全局的 **Hook** 函数, 有兴趣的同学可以自己去试试。这里暂且不谈, 我们继续返回代码。

现在我们明白了:

```

1 Observable<String> observable=Observable.create(new
ObservableOnSubscribe<String>() {2     ...3     ...4 })

```

相当于:

```

1 Observable<String> observable=new ObservableCreate(new
ObservableOnSubscribe<String>() {2     ...3     ...4 })

```

好了, 至此我们明白了, 事件的源就是 `new ObservableCreate()` 对象, 将 `ObservableOnSubscribe` 作为参数传递给 `ObservableCreate` 的构造函数。事件是由接口 `ObservableOnSubscribe` 的 `subscribe` 方法上产的, 至于何时生产事件, 稍后再分析。

Observer

Observer 是数据的下游，即事件消费者

Observer 是个 interface，包含：

```
1 void onSubscribe(@NonNull Disposable d); 2 void onNext(@NonNull T t); 3
void onError(@NonNull Throwable e); 4 void onComplete();
```

上游发送的事件就是再这几个方法中被消费的。上游何时发送事件、如何发送，稍后再表。

subscribe

重点来了，接下来最重要的方法来了：observable.subscribe(observer);

从这个方法的名字就知道，subscribe 是订阅，是将观察者(observer)与被观察者(observable)连接起来的方法。只有 subscribe 方法执行后，上游产生的事件才能被下游接收并处理。其实自然的方式应该是 observer 订阅(subscribe) observable，但这样会打断 rxjava 的链式结构。所以采用相反的方式。

接下来看源码，只列出关键代码



```
1 public final void subscribe(Observer<? super T> observer) { 2
ObjectHelper.requireNonNull(observer, "observer is null"); 3 ..... 4
observer = RxJavaPlugins.onSubscribe(this, observer); // hook，默认直接返回
observer 5 ..... 6 subscribeActual(observer); // 这个才是真正实现订
阅的方法。 7 ..... 8 } 9 10 // subscribeActual 是抽象方法，所以需要到实现
类中去看具体实现，也就是说实现是在上文中提到的 ObservableCreate 中 11 protected abstract
void subscribeActual(Observer<? super T> observer);
```



接下来我们来看 ObservableCreate.java:



```
1 public ObservableCreate(ObservableOnSubscribe<T> source) { 2
this.source = source; // 事件源，生产事件的接口，由我们自己实现 3 } 4 5
@Override 6 protected void subscribeActual(Observer<? super T> observer) { 7
CreateEmitter<T> parent = new CreateEmitter<T>(observer); // 发射器 8
observer.onSubscribe(parent); //直接回调了观察者的 onSubscribe 9 10 try {11
// 调用了事件源 subscribe 方法生产事件，同时将发射器传给事件源。 12 // 现在我们
明白了，数据源生产事件的 subscribe 方法只有在 observable.subscribe(observer)被执行 13
后才执行的。换言之，事件流是在订阅后才产生的。 14 //而 observable 被创建出来时
并不生产事件，同时也不发射事件。 15 source.subscribe(parent); 16 }
catch (Throwable ex) {17 Exceptions.throwIfFatal(ex); 18
parent.onError(ex); 19 } 20 }
```




现在我们明白了，数据源生产事件的 **subscribe** 方法只有在 **observable.subscribe(observer)** 被执行后才执行的。换言之，事件流是在订阅后才产生的。而 **observable** 被创建出来时并不生产事件，同时也不发射事件。

接下来我们再来看看事件是如何被发射出去，同时 **observer** 是如何接收到发射的事件的
`CreateEmitter<T> parent = new CreateEmitter<T>(observer);`

`CreateEmitter` 实现了 `ObservableEmitter` 接口，同时 `ObservableEmitter` 接口又继承了 `Emitter` 接口。

`CreateEmitter` 还实现了 `Disposable` 接口，这个 `disposable` 接口是用来判断是否中断事件发射的。

从名称上就能看出，这个是发射器，故名思议是用来发射事件的，正是它将上游产生的事件发射到下游的。

`Emitter` 是事件源与下游的桥梁。

`CreateEmitter` 主要包括方法：

```
1 void onNext(@NonNull T value); 2 void onError(@NonNull Throwable error); 3
void onComplete(); 4 public void dispose(); 5 public boolean isDisposed();
```

是不是跟 `observer` 的方法很像？

我们来看看 `CreateEmitter` 中这几个方法的具体实现：

只列出关键代码



```
1 public void onNext(T t) { 2     if (!isDisposed()) { // 判断事件是否需要被
    丢弃 3         observer.onNext(t); // 调用 Emitter 的 onNext, 它会直接调用 observer
    的 onNext 4     } 5     } 6     public void onError(Throwable t) { 7
    if (!isDisposed()) { 8         try { 9             observer.onError(t);
    // 调用 Emitter 的 onError, 它会直接调用 observer 的 onError 10         } finally
    { 11             dispose(); // 当 onError 被触发时, 执行 dispose(), 后续 onNext,
    onError, onComplete 就不会继 12             续发射事件了
    13         } 14     } 15     } 16 17     @Override 18
    public void onComplete() { 19         if (!isDisposed()) { 20             try
    { 21                 observer.onComplete(); // 调用 Emitter 的 onComplete, 它会直
    接调用 observer 的 onComplete 22             } finally { 23
    dispose(); // 当 onComplete 被触发时, 也会执行 dispose(), 后续 onNext, onError,
    onComplete 24             同样不会继续发射事件了
    25         } 26     } 27     }
```



CreateEmitter 的 onError 和 onComplete 方法任何一个执行完都会执行 dispose()中断事件发射，所以 observer 中的 onError 和 onComplete 也只能有一个被执行。

现在终于明白了，事件是如何被发射给下游的。当订阅成功后，数据源

ObservableOnSubscribe 开始生产事件，调用 Emitter 的 onNext, onComplete 向下游发射事件，

Emitter 包含了 observer 的引用，又调用了 observer onNext, onComplete，这样下游 observer 就接收到了上游发射的数据。

总结

Rxjava 的流程大概是：

1. **Observable.create** 创建事件源，但并不生产也不发射事件。
2. 实现 **observer** 接口，但此时没有也无法接受到任何发射来的事件。
3. 订阅 **observable.subscribe(observer)**，此时会调用具体 **Observable** 的实现类中的 **subscribeActual** 方法，此时才会真正触发事件源生产事件，事件源生产出来的事件通过 **Emitter** 的 **onNext, onError, onComplete** 发射给 **observer** 对应的方法由下游 **observer** 消费掉。从而完成整个事件流的处理。

observer 中的 onSubscribe 在订阅时即被调用，并传回了 Disposable，observer 中可以利用 Disposable 来随时中断事件流的发射。


今天所列举的例子是最简单的一个事件处理流程，没有使用线程调度，Rxjava 最强大的就是异步时对线程的调度和随时切换观察者线程，未完待续。

上面分析了 Rxjava 是如何创建事件源，如何发射事件，何时发射事件，也清楚了上游和下游是如何关联起来的。

下面着重来分析下 Rxjava 强大的线程调度是如何实现的。

二、RxJava 的线程调度机制

简单的例子



```
1 private void doSomeWork() { 2     Observable.create(new
ObservableOnSubscribe<String>() { 3         @Override 4         public void
subscribe(ObservableEmitter<String> e) throws Exception { 5
Log.i("lx", " subscribe: " + Thread.currentThread().getName()); 6
Thread.sleep(2000); 7         e.onNext("a"); 8
e.onComplete(); 9         }10     }).subscribe(new Observer<String>() {11
@Override12         public void onSubscribe(Disposable d) {13
Log.i("lx", " onSubscribe: " +
```



```

Thread.currentThread().getName());14          }15          @Override16
public void onNext(String str) {17          Log.i("lx", " onNext: " +
Thread.currentThread().getName());18          }19          @Override20
public void onError(Throwable e) {21          Log.i("lx", " onError: " +
Thread.currentThread().getName());22          }23          @Override24
public void onComplete() {25          Log.i("lx", " onComplete: " +
Thread.currentThread().getName());26          }27          });28      }

```



运行结果:

```

1 com.reactivex2.android.samples I/lx:  onSubscribe: main2
com.reactivex2.android.samples I/lx:  subscribe: main3
com.reactivex2.android.samples I/lx:  onNext: main4 com.reactivex2.android.samples
I/lx:  onComplete: main

```

因为此方法笔者是在 **main** 线程中调用的, 所以没有进行线程调度的情况下, 所有方法都运行在 **main** 线程中。但我们知道 **Android** 的 **UI** 线程是不能做网络操作, 也不能做耗时操作, 所以一般我们把网络或耗时操作都放在非 **UI** 线程中执行。接下来我们就来感受下 **Rxjava** 强大的线程调度能力。



```

1 private void doSomeWork() { 2          Observable.create(new
ObservableOnSubscribe<String>() { 3          @Override 4          public void
subscribe(ObservableEmitter<String> e) throws Exception { 5
Log.i("lx", " subscribe: " + Thread.currentThread().getName()); 6
Thread.sleep(2000); 7          e.onNext("a"); 8
e.onComplete(); 9          }10          }).subscribeOn(Schedulers.io()) //增加了
这一句11          .subscribe(new Observer<String>() {12          @Override13
public void onSubscribe(Disposable d) {14          Log.i("lx", " onSubscribe:
" + Thread.currentThread().getName());15          }16          @Override17
public void onNext(String str) {18          Log.i("lx", " onNext: " +
Thread.currentThread().getName());19          }20          @Override21
public void onError(Throwable e) {22          Log.i("lx", " onError: " +
Thread.currentThread().getName());23          }24          @Override25
public void onComplete() {26          Log.i("lx", " onComplete: " +
Thread.currentThread().getName());27          }28          });29      }

```



运行结果:

```
1 com.rxjava2.android.samples I/lx:  onSubscribe: main2
com.rxjava2.android.samples I/lx:  subscribe: RxCachedThreadScheduler-13
com.rxjava2.android.samples I/lx:  onNext: RxCachedThreadScheduler-14
com.rxjava2.android.samples I/lx:  onComplete: RxCachedThreadScheduler-1
```

只增加了 **subscribeOn** 这一句代码，就发生如此神奇的现象，除了 **onSubscribe** 方法还运行在 **main** 线程（订阅发生的线程）其它方法全部都运行在一个名为 **RxCachedThreadScheduler-1** 的线程中。我们来看看 **rxjava** 是怎么完成这个线程调度的。

线程调度 subscribeOn

首先我们先分析下 **Schedulers.io()**这个东东。

```
1 @NonNull2     public static Scheduler io() {3         return
RxJavaPlugins.onIoScheduler(IO); // hook function4         // 等价于 5
return IO;6     }
```

再看看 **IO** 是什么，**IO** 是个 **static** 变量，初始化的地方是

```
1 IO = RxJavaPlugins.initIoScheduler(new IOTask()); // 又是 hook function2 // 等
价于 3 IO = callRequireNonNull(new IOTask());4 // 等价于 5 IO = new IOTask().call();
```

继续看看 **IOTask**

```
1 static final class IOTask implements Callable<Scheduler> {2         @Override3
public Scheduler call() throws Exception {4             return IoHolder.DEFAULT;5
// 等价于 6             return new IoScheduler();7         }8     }
```

代码层次很深，为了便于记忆，我们再回顾一下：

```
1 Schedulers.io() 等价于 new IoScheduler()2 3     // Schedulers.io() 等价于 4
@NonNull5     public static Scheduler io() {6         return new
IoScheduler();7     }
```

好了，排除了其他干扰代码，接下来看看 **IoScheduler()**是什么东东了

IoScheduler 看名称就知道是个 **IO** 线程调度器，根据代码注释得知，它就是一个用来创建和缓存线程的线程池。看到这个豁然开朗了，原来 **Rxjava** 就是通过这个调度器来调度线程的，至于具体怎么实现我们接着往下看

```

1 public IoScheduler() { 2         this(WORKER_THREAD_FACTORY); 3     } 4     5
public IoScheduler(ThreadFactory threadFactory) { 6         this.threadFactory =
threadFactory; 7         this.pool = new AtomicReference<CachedWorkerPool>(NONE);
8         start(); 9     }10 11     @Override12     public void start() {13
CachedWorkerPool update = new CachedWorkerPool(KEEP_ALIVE_TIME, KEEP_ALIVE_UNIT,
threadFactory);14         if (!pool.compareAndSet(NONE, update)) {15
update.shutdown();16         }17     }18     19     CachedWorkerPool(long
keepAliveTime, TimeUnit unit, ThreadFactory threadFactory) {20
this.keepAliveTime = unit != null ? unit.toNanos(keepAliveTime) : 0L;21
this.expiringWorkerQueue = new ConcurrentLinkedQueue<ThreadWorker>();22
this.allWorkers = new CompositeDisposable();23         this.threadFactory =
threadFactory;24 25         ScheduledExecutorService evictor = null;26
Future<?> task = null;27         if (unit != null) {28             evictor
= Executors.newScheduledThreadPool(1, EVICTOR_THREAD_FACTORY);29
task = evictor.scheduleWithFixedDelay(this, this.keepAliveTime,
this.keepAliveTime, TimeUnit.NANOSECONDS);30         }31
evictorService = evictor;32         evictorTask = task;33     }

```



从上面的代码可以看出, `new IoScheduler()` 后 Rxjava 会创建 `CachedWorkerPool` 的线程池, 同时也创建并运行了一个名为 `RxCachedWorkerPoolEvictor` 的清除线程, 主要作用是清除不再使用的一些线程。

但目前只创建了线程池并没有实际的 `thread`, 所以 `Schedulers.io()` 相当于只做了线程调度的前期准备。

OK, 终于可以开始分析 Rxjava 是如何实现线程调度的。回到 Demo 来看 `subscribeOn()` 方法的内部实现:

```

1 public final Observable<T> subscribeOn(Scheduler scheduler) {2
ObjectHelper.requireNonNull(scheduler, "scheduler is null");3     return
RxJavaPlugins.onAssembly(new ObservableSubscribeOn<T>(this,
scheduler));4     }

```

很熟悉的代码 `RxJavaPlugins.onAssembly`, 上一篇已经分析过这个方法, 就是个 `hook function`, 等价于直接 `return new ObservableSubscribeOn<T>(this, scheduler);`, 现在知道了这里的 `scheduler` 其实就是 `IoScheduler`。

跟踪代码进入 `ObservableSubscribeOn`,

可以看到这个 `ObservableSubscribeOn` 继承自 `Observable`, 并且扩展了一些属性, 增加了 `scheduler`。各位看官, 这不就是典型的装饰模式嘛, Rxjava 中大量用到了装饰模式, 后面还会经常看到这种 `wrap` 类。

上篇文章我们已经知道了 `Observable.subscribe()` 方法最终都是调用了对应的实现类的 `subscribeActual` 方法。我们重点分析下 `subscribeActual`:



```
1 @Override 2     public void subscribeActual(final Observer<? super T> s) { 3
final SubscribeOnObserver<T> parent = new SubscribeOnObserver<T>(s); 4 5
// 没有任何线程调度，直接调用的，所以下游的 onSubscribe 方法没有切换线程， 6     //本文 demo 中下游就是观察者，所以我们明白了为什么只有 onSubscribe 还运行在 main 线程 7
s.onSubscribe(parent); 8 9
parent.setDisposable(scheduler.scheduleDirect(new
SubscribeTask(parent)));10 }
```



SubscribeOnObserver 也是装饰模式的体现，是对下游 observer 的一个 wrap，只是添加了 Disposable 的管理。

接下来分析最重要的 scheduler.scheduleDirect(new SubscribeTask(parent))



```
1 // 这个类很简单，就是一个 Runnable，最终运行上游的 subscribe 方法 2     final class
SubscribeTask implements Runnable { 3         private final
SubscribeOnObserver<T> parent; 4 5
SubscribeTask(SubscribeOnObserver<T> parent) { 6             this.parent = parent;
7         } 8 9         @Override10         public void run() {11
source.subscribe(parent);12         }13     }14     @Nonnull15     public
Disposable scheduleDirect(@Nonnull Runnable run, long delay, @Nonnull TimeUnit
unit) {16         // IoScheduler 中的 createWorker()17         final Worker w =
createWorker();18         // hook decoratedRun=run;19         final Runnable
decoratedRun = RxJavaPlugins.onSchedule(run);20         // decoratedRun 的 wrap,
增加了 Dispose 的管理 21         DisposeTask task = new DisposeTask(decoratedRun,
w);22         // 线程调度 23         w.schedule(task, delay, unit);24 25
return task;26     }
```



回到 IoScheduler



```
1 public Worker createWorker() { 2         // 工作线程是在此时创建的 3         return
new EventLoopWorker(pool.get()); 4     } 5     6     public Disposable
schedule(@Nonnull Runnable action, long delayTime, @Nonnull TimeUnit unit) { 7
if (tasks.isDisposed()) { 8         // don't schedule, we are unsubscribed
9         return EmptyDisposable.INSTANCE;10     }11         //
```

```

        action 中就包含上游 subscribe 的 runnable12        return
threadWorker.scheduleActual(action, delayTime, unit, tasks);13    }

```



最终线程是在这个方法内调度并执行的。



```

1 public ScheduledRunnable scheduleActual(final Runnable run, long delayTime,
@NonNull TimeUnit unit, @Nullable DisposableContainer parent) { 2    //
decoratedRun = run, 包含上游 subscribe 方法的 runnable 3    Runnable
decoratedRun = RxJavaPlugins.onSchedule(run); 4 5    // decoratedRun 的 wrap,
增加了 dispose 的管理 6    ScheduledRunnable sr = new
ScheduledRunnable(decoratedRun, parent); 7 8    if (parent != null) { 9
if (!parent.add(sr)) {10        return sr;11    }12    }13 14
// 最终 decoratedRun 被调度到之前创建或从线程池中取出的线程, 15    // 也就是说在
RxCachedThreadScheduler-x 运行 16    Future<?> f;17    try {18
if (delayTime <= 0) {19        f =
executor.submit((Callable<Object>)sr);20    } else {21        f =
executor.schedule((Callable<Object>)sr, delayTime, unit);22    }23
sr.setFuture(f);24    } catch (RejectedExecutionException ex) {25
if (parent != null) {26        parent.remove(sr);27    }28
RxJavaPlugins.onError(ex);29    }30 31    return sr;32    }

```



至此我们终于明白了 Rxjava 是如何调度线程并执行的，通过 `subscribeOn` 方法将上游生产事件的方法运行在指定的调度线程中。

```

1 com.rxjava2.android.samples I/lx: onSubscribe: main2
com.rxjava2.android.samples I/lx: subscribe: RxCachedThreadScheduler-13
com.rxjava2.android.samples I/lx: onNext: RxCachedThreadScheduler-14
com.rxjava2.android.samples I/lx: onComplete: RxCachedThreadScheduler-1

```

从上面的运行结果来看，因为上游生产者已被调度到 `RxCachedThreadScheduler-1` 线程中，同时发射事件并没有切换线程，所以发射后消费事件的 `onNext onErro onComplete` 也在 `RxCachedThreadScheduler-1` 线程中。

总结

1. `Schedulers.io()`等价于 `new IoScheduler()`。
2. `new IoScheduler()` Rxjava 创建了线程池，为后续创建线程做准备，同时创建并运行了一个清理线程 `RxCachedWorkerPoolEvictor`，定期执行清理任务。
3. `subscribeOn()`返回一个 `ObservableSubscribeOn` 对象，它是 `Observable` 的一个装饰类，增加了 `scheduler`。

4. 调用 `subscribe()` 方法，在这个方法调用后，`subscribeActual()` 被调用，才真正执行了 `IoScheduler` 中的 `createWorker()` 创建线程并运行，最终将上游 `Observable` 的 `subscribe()` 方法调度到新建的线程中运行。

现在了解了被观察者执行线程是如何被调度到指定线程中执行的，但很多情况下，我们希望观察者（事件下游）处理事件最好在 `UI` 线程执行，比如更新 `UI` 操作等。下面分析下游何时调度，如何调度由于篇幅问题。

三、Rxjava 如何对观察者线程进行调度

简单的例子

```
1 private void doSomeWork() { 2     Observable.create(new
ObservableOnSubscribe<String>() { 3         @Override 4         public void
subscribe(ObservableEmitter<String> e) throws Exception { 5
Log.i("lx", " subscribe: " + Thread.currentThread().getName()); 6
e.onNext("a"); 7         e.onComplete(); 8     }
9     }).subscribeOn(Schedulers.io())10         .observeOn(AndroidSchedu
rs.mainThread())11         .subscribe(new Observer<String>() {12
@Override13         public void onSubscribe(Disposable d) {14
Log.i("lx", " onSubscribe: " +
Thread.currentThread().getName());15         }16         @Override17
public void onNext(String str) {18             Log.i("lx", " onNext: " +
Thread.currentThread().getName());19         }20         @Override21
public void onError(Throwable e) {22             Log.i("lx", " onError: " +
Thread.currentThread().getName());23         }24         @Override25
public void onComplete() {26             Log.i("lx", " onComplete: " +
Thread.currentThread().getName());27         }28     });29 }
```

看看运行结果：

```
1 com.rxjava2.android.samples I/lx: onSubscribe: main2
com.rxjava2.android.samples I/lx: subscribe: RxCachedThreadScheduler-13
com.rxjava2.android.samples I/lx: onNext: main4 com.rxjava2.android.samples
I/lx: onComplete: main
```

从结果可以看出，事件的生产线程运行在 `RxCachedThreadScheduler-1` 中，而事件的消费线程则被调度到了 `main` 线程中。关键代码是因为这句 `observeOn(AndroidSchedulers.mainThread())`。下面我们着重分析下这句代码都做了哪些事情。

`AndroidSchedulers.mainThread()`

先来看看 `AndroidSchedulers.mainThread()` 是什么？贴代码

```
1 /** A {@link Scheduler} which executes actions on the Android main thread. */2
public static Scheduler mainThread() {3     return
RxAndroidPlugins.onMainThreadScheduler(MAIN_THREAD);4 }
```

注释已经说的很明白了，是一个在主线程执行任务的 **scheduler**，接着看

```
1 private static final Scheduler MAIN_THREAD =
RxAndroidPlugins.initMainThreadScheduler( 2     new Callable<Scheduler>()
{ 3         @Override public Scheduler call() throws Exception { 4
return MainHolder.DEFAULT; 5         } 6     }); 7     8
public static Scheduler initMainThreadScheduler(Callable<Scheduler> scheduler)
{ 9     if (scheduler == null) {10         throw new
NullPointerException("scheduler == null");11     }12
Function<Callable<Scheduler>, Scheduler> f = onInitMainThreadHandler;13     if
(f == null) {14         return callRequireNonNull(scheduler);15     }16     return
applyRequireNonNull(f, scheduler);17 }
```

代码很简单，这个 `AndroidSchedulers.mainThread()` 相当于 `new HandlerScheduler(new Handler(Looper.getMainLooper()))`，原来是利用 Android 的 Handler 来调度到 main 线程的。

我们再看看 `HandlerScheduler`，它与我们上节分析的 `IOScheduler` 类似，都是继承自 `Scheduler`，所以 `AndroidSchedulers.mainThread()` 其实就是创建了一个运行在 main thread 上的 scheduler。

好了，我们再回过头来看 `observeOn` 方法。

observeOn

```
1 public final Observable<T> observeOn(Scheduler scheduler) { 2     return
observeOn(scheduler, false, bufferSize()); 3     } 4     5     public final
Observable<T> observeOn(Scheduler scheduler, boolean delayError, int bufferSize)
{ 6         ObjectHelper.requireNonNull(scheduler, "scheduler is null"); 7
ObjectHelper.verifyPositive(bufferSize, "bufferSize"); 8         return
RxJavaPlugins.onAssembly(new ObservableObserveOn<T>(this, scheduler,
delayError, bufferSize)); 9     }10 }
```

重点是这个 `new ObservableObserveOn`，看名字是不是有种似成相识的感觉，还记得上篇的 `ObservableSubscribeOn` 吗？它俩就是亲兄弟，是继承自同一个父类。

重点还是这个方法，我们前文已经提到了，Observable 的 subscribe 方法最终都是调用 subscribeActual 方法。下面看看这个方法的实现：

```
1  @Override 2      protected void subscribeActual (Observer<? super T> observer)
{ 3          // scheduler 就是前面提到的 HandlerScheduler，所以进入 else 分支 4
if (scheduler instanceof TrampolineScheduler) { 5
source.subscribe(observer); 6      } else { 7          // 创建 HandlerWorker
8          Scheduler.Worker w = scheduler.createWorker(); 9          // 调用
上游 Observable 的 subscribe，将订阅向上传递 10      source.subscribe(new
ObserveOnObserver<T>(observer, w, delayError, bufferSize));11      }12  }
```

从上面代码可以看到使用了 ObserveOnObserver 类对 observer 进行装饰，好了，我们再来看看 ObserveOnObserver。

我们已经知道了，事件源发射的事件，是通过 observer 的 onNext,onError,onComplete 发射到下游的。所以看看 ObserveOnObserver 的这三个方法是如何实现的。

由于篇幅问题，我们只分析 onNext 方法，onError 和 onComplete 方法有兴趣的同学可以自己分析下。

```
1  @Override 2      public void onNext(T t) { 3          if (done) { 4              return;
5          } 6          7          // 如果是非异步方式，将上游发射的时间加入到队列 8
if (sourceMode != QueueDisposable.ASYNC) { 9
queue.offer(t);10      }11      schedule();12      }13      14      void
schedule() {15          // 保证只有唯一任务在运行 16          if (getAndIncrement() == 0)
{17              // 调用的就是 HandlerWorker 的 schedule 方法 18
worker.schedule(this);19      }20      }21      22      @Override23
public Disposable schedule(Runnable run, long delay, TimeUnit unit) {24
if (run == null) throw new NullPointerException("run == null");25      if
(unit == null) throw new NullPointerException("unit == null");26 27      if
(disposed) {28          return Disposables.disposed();29      }30 31
run = RxJavaPlugins.onSchedule(run);32 33      ScheduledRunnable scheduled
= new ScheduledRunnable(handler, run);34 35      Message message =
Message.obtain(handler, scheduled);36      message.obj = this; // Used as
token for batch disposal of this worker's runnables.37 38
handler.sendMessageDelayed(message, Math.max(0L, unit.toMillis(delay)));39 40
// Re-check disposed state for removing in case we were racing a call to dispose().41
if (disposed) {42          handler.removeCallbacks(scheduled);43
return Disposables.disposed();44      }45 46      return
scheduled;47  }
```




`schedule` 方法将传入的 `run` 调度到对应的 `handle` 所在的线程来执行，这个例子里就是有 `main` 线程来完成。再回去看看前面传入的 `run` 吧。

回到 `ObserveOnObserver` 中的 `run` 方法：



```
1 @Override 2     public void run() { 3         // 此例子中代码不会进入这个分支，至于
这个 drainFused 是什么，后面章节再讨论。 4         if (outputFused) { 5
drainFused(); 6         } else { 7             drainNormal(); 8         } 9     } 10
11     void drainNormal() { 12         int missed = 1; 13 14         final
SimpleQueue<T> q = queue; 15         final Observer<? super T> a = actual; 16 17
for (;;) { 18             if (checkTerminated(done, q.isEmpty(), a)) { 19
return; 20             } 21 22             for (;;) { 23                 boolean d =
done; 24                 T v; 25 26                 try { 27                     // 从队列
中 queue 中取出事件 28                     v = q.poll(); 29                 } catch
(Throwable ex) { 30                     Exceptions.throwIfFatal(ex); 31
s.dispose(); 32                     q.clear(); 33
a.onError(ex); 34                     worker.dispose(); 35
return; 36                 } 37                 boolean empty = v == null; 38 39
if (checkTerminated(d, empty, a)) { 40
return; 41                 } 42 43                 if (empty) { 44
break; 45                 } 46                 //调用下游 observer 的 onNext 将事件 v 发射出
去 47                 a.onNext(v); 48                 } 49 50                 missed =
addAndGet(-missed); 51                 if (missed == 0) { 52
break; 53                 } 54             } 55         }
```



至此我们明白了 `RXjava` 是如何调度消费者线程了。

消费者线程调度流程概括

`Rxjava` 调度消费者现在的流程，以 `observeOn(AndroidSchedulers.mainThread())` 为例。

1. `AndroidSchedulers.mainThread()` 先创建一个包含 `handler` 的 `Scheduler`，这个 `handler` 是主线程的 `handler`。
2. `observeOn` 方法创建 `ObservableObserveOn`，它是上游 `Observable` 的一个装饰类，其中包含前面创建的 `Scheduler` 和 `bufferSize` 等。
3. 当订阅方法 `subscribe` 被调用后，`ObservableObserveOn` 的 `subscribeActual` 方法创建 `Scheduler.Worker` 并调用上游的 `subscribe` 方法，同时将自身接收的参数 'observer' 用装饰类 `ObserveOnObserver` 装饰后传递给上游。

4. 当上游调用被 `ObserveOnObserver` 的 `onNext`、`onError` 和 `onComplete` 方法时，`ObserveOnObserver` 将上游发送的事件通通加入到队列 `queue` 中，然后再调用 `scheduler` 将处理事件的方法调度到对应的线程中（本例会调度到 `main thread`）。处理事件的方法将 `queue` 中保存的事件取出来，调用下游原始的 `observer` 再发射出去。
5. 经过以上流程，下游处理事件的消费者线程就运行在了 `observeOn` 调度后的 `thread` 中。

总结

经过前面两节的分析，我们已经明白了 `Rxjava` 是如何对线程进行调度的。

- `Rxjava` 的 `subscribe` 方法是由下游一步步向上游进行传递的。会调用上游的 `subscribe`，直到调用到事件源。

如：`source.subscribe(xxx);`

而上游的 `source` 往往是经过装饰后的 `Observable`，`Rxjava` 就是利用 `ObservableSubscribeOn` 将 `subscribe` 方法调度到了指定线程运行，生产者线程最终会运行在被调度后的线程中。但多次调用 `subscribeOn` 方法会怎么样呢？我们知道因为 `subscribe` 方法是由下而上传递的，所以事件源的生产者线程最终都只会运行在第一次执行 `subscribeOn` 所调度的线程中，换句话说就是多次调用 `subscribeOn` 方法，只有第一次有效。

- `Rxjava` 发射事件是由上而下发射的，上游的 `onNext`、`onError`、`onComplete` 方法会调用下游传入的 `observer` 的对应方法。往往下游传递的 `observer` 对象也是经过装饰后的 `observer` 对象。`Rxjava` 就是利用 `ObserveOnObserver` 将执行线程调度后，再调用下游对应的 `onNext`、`onError`、`onComplete` 方法，这样下游消费者就运行在了指定的线程内。那么多次调用 `observeOn` 调度不同的线程会怎么样呢？因为事件是由上而下发射的，所以每次用 `observeOn` 切换完线程后，对下游的事件消费都有效，比如下游的 `map` 操作符。最终的事件消费线程运行在最后一个 `observeOn` 切换后线程中。
- 另外通过源码可以看到 `onSubscribe` 运行在 `subscribe` 的调用线程中，这个就不具体分析了。

8. Retrofit （Retrofit 在 OkHttpClient 上做了哪些封装？动态代理和静态代理的区别，是怎么实现的）

源码解析

从 **Builder** 模式创建实例开始看起

首先我们先从上面的第 4 步开始解析源码，有下面这段代码：

```
retrofit = new Retrofit.Builder()

    .baseUrl(GankConfig.HOST)

    .addConverterFactory(GsonConverterFactory.create(date_gson)) // 添加
    一个转换器，将 gson 数据转换为 bean 类

    .addCallAdapterFactory(RxJava2CallAdapterFactory.create()) // 添加一
    个适配器，与 RxJava 配合使用

    .build();
```

很明显这个是使用了 **Builder** 模式，接下来我们一步一步来看里面做了什么？首先是 **Builder()**。

```
public Builder() {

    this(Platform.get());

}

Builder(Platform platform) {

    this.platform = platform;

    // 添加转换器，请见下面关于 addConverterFactory() 的讲解

    converterFactories.add(new BuiltInConverters());

}
```

构造方法中的参数是 **Platform** 的静态方法 **get()**，接下来就看看 **get()**。

```
private static final Platform PLATFORM = findPlatform();

static Platform get() {

    return PLATFORM;

}

private static Platform findPlatform() {

    try {

        Class.forName("android.os.Build");

        if (Build.VERSION.SDK_INT != 0) {

            return new Android();

        }

    } catch (ClassNotFoundException e) {

        return new Android();

    }

}
```

```

    }

    } catch (ClassNotFoundException ignored) {

    }

    try {

        Class.forName("java.util.Optional");

        return new Java8();

    } catch (ClassNotFoundException ignored) {

    }

    return new Platform();

    }

}

```

可以看到，Retrofit 支持多平台，包括 Android 与 JAVA8，它会根据不同的平台设置不同的线程池。先来看看到目前为止我们分析到哪里了



接下来看一下 **baseUrl()** 方法。

```

public Builder baseUrl(String baseUrl) {

    checkNotNull(baseUrl, "baseUrl == null");

    HttpUrl httpUrl = HttpUrl.parse(baseUrl);

    if (httpUrl == null) {

        throw new IllegalArgumentException("Illegal URL: " + baseUrl);

    }

    return baseUrl(httpUrl);

}

```

很容易理解，**baseUrl()** 是配置服务器的地址的，如果为空，那么就会抛出异常。

接着是 **addConverterFactory()**

```

private final List<Converter.Factory> converterFactories = new ArrayList<>();

public Builder addConverterFactory(Converter.Factory factory) {

    converterFactories.add(checkNotNull(factory, "factory == null"));

}

```

```
        return this;
```

```
    }
```

大家是不是还记得刚才在 **Builder()** 方法初始化的时候，有这样一行代码：

```
converterFactories.add(new BuiltInConverters());
```

可以看到，**converterFactories** 在初始化的时候就已经添加了一个默认的 **Converter**，那我们手动添加的这个 **GsonConverter** 是干什么用的呢？

```
public final class GsonConverterFactory extends Converter.Factory {
```

```
    public static GsonConverterFactory create() {
```

```
        return create(new Gson());
```

```
    }
```

```
    public static GsonConverterFactory create(Gson gson) {
```

```
        return new GsonConverterFactory(gson);
```

```
    }
```

```
    private final Gson gson;
```

```
    private GsonConverterFactory(Gson gson) {
```

```
        if (gson == null) throw new NullPointerException("gson == null");
```

```
        this.gson = gson;
```

```
    }
```

```
    @Override
```

```
    public Converter<ResponseBody, ?> responseBodyConverter(Type type,
        Annotation[] annotations,
```

```
        Retrofit retrofit) {
```

```
        TypeAdapter<?> adapter = gson.getAdapter(TypeToken.get(type));
```

```
        return new GsonResponseBodyConverter<>(gson, adapter);
```

```
    }
```

```

@Override

public Converter<?, RequestBody> requestBodyConverter(Type type,
    Annotation[] parameterAnnotations, Annotation[] methodAnnotations,
    Retrofit retrofit) {

    TypeAdapter<?> adapter = gson.getAdapter(TypeToken.get(type));

    return new GsonRequestBodyConverter<>(gson, adapter);

}

}

```

其实这个 **Converter** 主要的作用就是将 HTTP 返回的数据解析成 Java 对象，我们常见的网络传输数据有 Xml、Gson、protobuf 等等，而 **GsonConverter** 就是将 **Gson** 数据转换为我们的 Java 对象，而不用我们重新去解析这些 **Gson** 数据。

接着看 **addCallAdapterFactory()**

```

private final List<CallAdapter.Factory> adapterFactories = new ArrayList<>();

public Builder addCallAdapterFactory(CallAdapter.Factory factory) {

    adapterFactories.add(checkNotNull(factory, "factory == null"));

    return this;

}

```

可以看到，**CallAdapter** 同样也被一个 **List** 维护，也就是说用户可以添加多个 **CallAdapter**，那 **Retrofit** 总得有一个默认的吧，默认的是什么呢？请看接下来的 **build()**。

最后看一下 **build()**

```

public Retrofit build() {

    //检验 baseUrl

    if (baseUrl == null) {

        throw new IllegalStateException("Base URL required.");

    }

    //创建一个 call，默认情况下使用 okhttp 作为网络请求器

    okhttp3.Call.Factory callFactory = this.callFactory;

    if (callFactory == null) {

        callFactory = new OkHttpClient();

    }

```

```
}
```

```
Executor callbackExecutor = this.callbackExecutor;
```

```
if (callbackExecutor == null) {
```

```
    callbackExecutor = platform.defaultCallbackExecutor();
```

```
}
```

```
List<CallAdapter.Factory> adapterFactories = new
```

```
ArrayList<>(this.adapterFactories);
```

```
//添加一个默认的 callAdapter
```

```
adapterFactories.add(platform.defaultCallAdapterFactory(callbackExecutor));
```

```
List<Converter.Factory> converterFactories = new
```

```
ArrayList<>(this.converterFactories);
```

```
return new Retrofit(callFactory, baseUrl, converterFactories,  
adapterFactories,
```

```
callbackExecutor, validateEagerly);
```

```
}
```

首先 **Retrofit** 会新建一个 **call**，其实质就是 **OKHttp**，作用就是网络请求器；接着在上一点中我们困惑的 **callAdapter** 也已经能够得到解决了，首先 **Retrofit** 有一个默认的 **callAdapter**，请看下面这段代码：

```
adapterFactories.add(platform.defaultCallAdapterFactory(callbackExecutor));
```

```
CallAdapter.Factory defaultCallAdapterFactory(Executor callbackExecutor) {
```

```
    if (callbackExecutor != null) {
```

```
        return new ExecutorCallAdapterFactory(callbackExecutor);
```

```
    }
```

```
    return DefaultCallAdapterFactory.INSTANCE;
```

```
}
```

```
final class ExecutorCallAdapterFactory extends CallAdapter.Factory {
```

```

    final Executor callbackExecutor;

    ExecutorCallAdapterFactory(Executor callbackExecutor) {

        this.callbackExecutor = callbackExecutor;
    }

    @Override

    public CallAdapter<?, ?> get(Type returnType, Annotation[] annotations,
Retrofit retrofit) {

        if (getRawType(returnType) != Call.class) {

            return null;

        }

        final Type responseType = Utils.getCallResponseType(returnType);

        return new CallAdapter<Object, Call<?>>() {

            @Override public Type responseType() {

                return responseType;

            }

            @Override public Call<Object> adapt(Call<Object> call) {

                return new ExecutorCallbackCall<>(callbackExecutor, 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) {

        if (callback == null) throw new NullPointerException("callback == null");

        delegate.enqueue(new Callback<T>() {

            @Override public void onResponse(Call<T> call, final Response<T> response)
            {

                callbackExecutor.execute(new Runnable() {

                    @Override public void run() {

                        if (delegate.isCanceled()) {

                            // Emulate OkHttp's behavior of throwing/delivering an IOException
                            on cancellation.

                            callback.onFailure(ExecutorCallbackCall.this, new
                            IOException("Canceled"));

                        } else {

                            callback.onResponse(ExecutorCallbackCall.this, response);

                        }

                    }

                });

            }

        });

    }

    @Override public void onFailure(Call<T> call, final Throwable t) {

        callbackExecutor.execute(new Runnable() {

            @Override public void run() {

                callback.onFailure(ExecutorCallbackCall.this, t);

            }

        });

    }

}

```

```
    }
```

```
    @Override public boolean isExecuted() {
```

```
        return delegate.isExecuted();
```

```
    }
```

```
    @Override public Response<T> execute() throws IOException {
```

```
        return delegate.execute();
```

```
    }
```

```
    @Override public void cancel() {
```

```
        delegate.cancel();
```

```
    }
```

```
    @Override public boolean isCanceled() {
```

```
        return delegate.isCanceled();
```

```
    }
```

```
    @SuppressWarnings("CloneDoesntCallSuperClone") // Performing deep clone.
```

```
    @Override public Call<T> clone() {
```

```
        return new ExecutorCallbackCall<>(callbackExecutor, delegate.clone());
```

```
    }
```

```
    @Override public Request request() {
```

```
        return delegate.request();
```

```
    }
```

```
    }
```

```
}
```

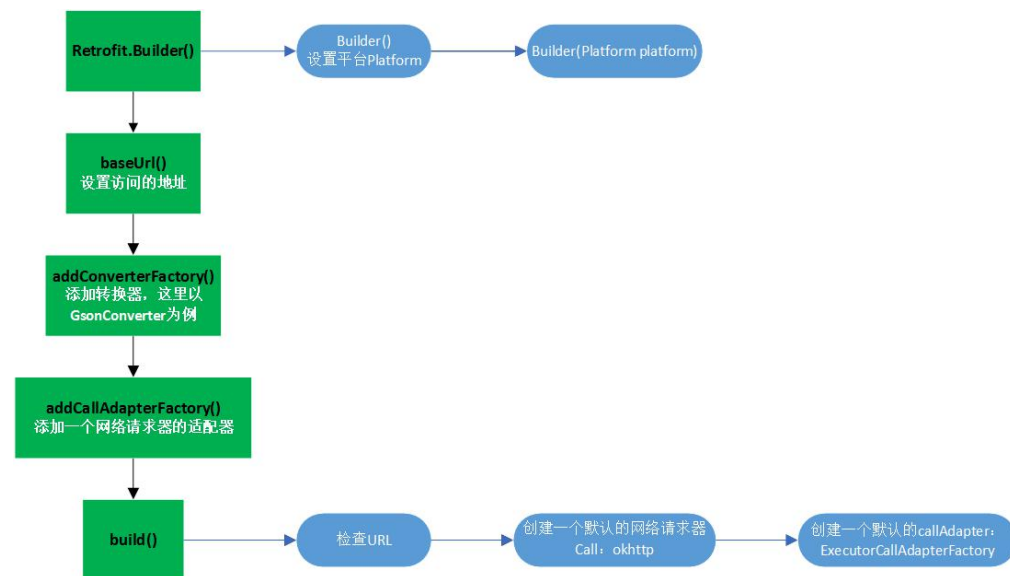
可以看到默认的 `callAdapter` 是 `ExecutorCallAdapterFactory`。`callAdapter` 其实也是运用了适配器模式，其实质就是网络请求器 `Call` 的适配器，而在 `Retrofit` 中 `Call` 就是指 `OKHttp`，那么

CallAdapter 就是用来将 OKHttp 适配给不同的平台的，在 Retrofit 中提供了四种 CallAdapter，分别如下：

- ExecutorCallAdapterFactory（默认使用）
- GuavaCallAdapterFactory
- Java8CallAdapterFactory
- RxJavaCallAdapterFactory

为什么要提供如此多的适配器呢？首先是易于扩展，例如用户习惯使用什么适配器，只需要添加即可使用；再者 RxJava 如此火热，因为其切换线程十分的方便，不需要手动使用 handler 切换线程，而 Retrofit 使用了支持 RxJava 的适配器之后，功能也会更加强大。

综上所述我们已经将使用 Builder 模式创建出来的 Retrofit 实例分析完毕了，我们只需要对相关的功能进行配置即可，Retrofit 负责接收我们配置的功能然后进行对象的初始化，这个也就是 Builder 模式屏蔽掉创建对象的复杂过程的好处。现在我们再次用流程图来梳理一下刚才的思路。



网络请求接口的创建

我最初使用 Retrofit 的时候觉得有一个地方十分神奇，如下：

```
GankRetrofit gankRetrofit=retrofit.create(GankRetrofit.class);
```

```
GankData data= gankRetrofit.getDailyData(2017, 9, 1);
```

要想解惑，首先得对动态代理有所了解，如果你对动态代理还不是很清楚，请点击[这里](#)了解动态代理的原理，之后再接着往下看。



我们就以这里为切入点开始分析吧！首先是 **create()**

```
public <T> T create(final Class<T> service) {  
  
    Utils.validateServiceInterface(service);  
  
    if (validateEagerly) {  
  
        eagerlyValidateMethods(service);  
  
    }  
  
    //重点看这里  
  
    return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>[] {  
        service },  
  
        new InvocationHandler() {  
  
            private final Platform platform = Platform.get();  
  
  
            @Override public Object invoke(Object proxy, Method method, Object[]  
args)  
  
                throws Throwable {  
  
                // If the method is a method from Object then defer to normal invocation.  
  
                if (method.getDeclaringClass() == Object.class) {
```

```

        return method.invoke(this, args);
    }

    if (platform.isDefaultMethod(method)) {

        return platform.invokeDefaultMethod(method, service, proxy, args);
    }

    //下面就会讲到哦

    ServiceMethod<Object, Object> serviceMethod =

        (ServiceMethod<Object, Object>) loadServiceMethod(method);

    //下一小节讲到哦

    OkHttpClient<Object> okHttpClient = new OkHttpClient<>(serviceMethod,
args);

    //下两个小节讲哦

    return serviceMethod.callAdapter.adapt(okHttpClient);

}

});

}

```

我们主要看 `Proxy.newProxyInstance` 方法，它接收三个参数，第一个是一个类加载器，其实哪个类的加载器都无所谓，这里为了方便就选择了我们所定义的借口的类加载器；第二个参数是我们定义的接口的 `class` 对象，第三个则是一个 `InvocationHandler` 匿名内部类。

那大家应该会有疑问了，这个 `newProxyInstance` 到底有什么用呢？其实他就是通过动态代理生成了网络请求接口的代理类，代理类生成之后，接下来我们就可以使用

`ankRetrofit.getDailyData(2017, 9, 1);`这样的语句去调用 `getDailyData` 方法，当我们调用这个方法的时候就会被动态代理拦截，直接进入 `InvocationHandler` 的 `invoke` 方法。下面就来讲讲它。

invoke 方法

它接收三个参数，第一个是动态代理，第二个是我们要调用的方法，这里就是指 `getDailyData`，第三个是一个参数数组，同样的这里就是指 `[2017, 9, 1]`，收到方法名和参数之后，紧接着会调用 `loadServiceMethod` 方法来生产过一个 `ServiceMethod` 对象，这里的一个 `ServiceMethod` 对象就对应我们在网络接口里定义的一个方法，相当于做了一层封装。接下来重点来看 `loadServiceMethod` 方法。

loadServiceMethod 方法

```
ServiceMethod<?, ?> loadServiceMethod(Method method) {
```

```

        ServiceMethod<?, ?> result = serviceMethodCache.get(method);

        if (result != null) return result;

        synchronized (serviceMethodCache) {

            result = serviceMethodCache.get(method);

            if (result == null) {

                result = new ServiceMethod.Builder<>(this, method).build();

                serviceMethodCache.put(method, result);

            }

        }

        return result;

    }
}

```

它调用了 **ServiceMethod** 类，而 **ServiceMethod** 也使用了 **Builder** 模式，直接先看 **Builder** 方法。

```

    Builder(Retrofit retrofit, Method method) {

        this.retrofit = retrofit;

        //获取接口中的方法名

        this.method = method;

        //获取方法里的注解

        this.methodAnnotations = method.getAnnotations();

        //获取方法里的参数类型

        this.parameterTypes = method.getGenericParameterTypes();

        //获取接口方法里的注解内容

        this.parameterAnnotationsArray = method.getParameterAnnotations();

    }
}

```

再来看 **build** 方法

```

public ServiceMethod build() {

    callAdapter = createCallAdapter();
}

```

```

        responseType = callAdapter.responseType();

        if (responseType == Response.class || responseType ==
okhttp3.Response.class) {

            throw methodError("'"

                + Utils.getRawType(responseType).getName()

                + "' is not a valid response body type. Did you mean ResponseBody?");

        }

        responseConverter = createResponseConverter();

        for (Annotation annotation : methodAnnotations) {

            parseMethodAnnotation(annotation);

        }

        if (httpMethod == null) {

            throw methodError("HTTP method annotation is required (e.g., @GET, @POST,
etc.).");

        }

        if (!hasBody) {

            if (isMultipart) {

                throw methodError(

                    "Multipart can only be specified on HTTP methods with request body
(e.g., @POST).");

            }

            if (isFormEncoded) {

                throw methodError("FormUrlEncoded can only be specified on HTTP methods
with "

                    + "request body (e.g., @POST).");

            }

        }

```

```

int parameterCount = parameterAnnotationsArray.length;

parameterHandlers = new ParameterHandler<?>[parameterCount];

for (int p = 0; p < parameterCount; p++) {

    Type parameterType = parameterTypes[p];

    if (Utils.hasUnresolvableType(parameterType)) {

        throw parameterError(p, "Parameter type must not include a type variable
or wildcard: %s",

            parameterType);

    }

    Annotation[] parameterAnnotations = parameterAnnotationsArray[p];

    if (parameterAnnotations == null) {

        throw parameterError(p, "No Retrofit annotation found.");

    }

    parameterHandlers[p] = parseParameter(p, parameterType,
parameterAnnotations);

}

if (relativeUrl == null && !gotUrl) {

    throw methodError("Missing either @%s URL or @Url parameter.",
httpMethod);

}

if (!isFormEncoded && !isMultipart && !hasBody && gotBody) {

    throw methodError("Non-body HTTP method cannot contain @Body.");

}

if (isFormEncoded && !gotField) {

    throw methodError("Form-encoded method must contain at least one
@Field.");

}

if (isMultipart && !gotPart) {

```



```

        throw methodError("Multipart method must contain at least one @Part.");
    }
}

```

```

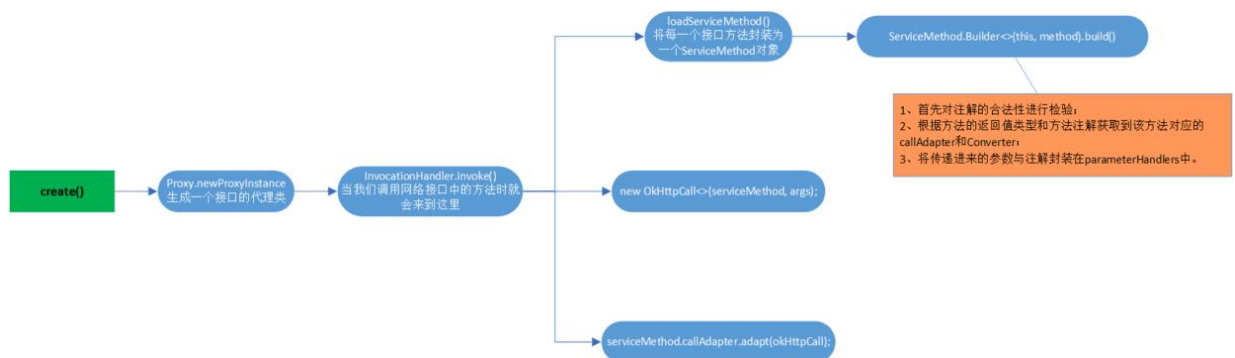
return new ServiceMethod<>(this);
}

```

代码稍微有点长，但是思路很清晰，主要的工作有

- 1、首先对注解的合法性进行检验，例如，HTTP 的请求方法是 GET 还是 POST，如果不是就会抛出异常；
- 2、根据方法的返回值类型和方法注解从 Retrofit 对象的的 `callAdapter` 列表和 `Converter` 列表中分别获取到该方法对应的 `callAdapter` 和 `Converter`；
- 3、将传递进来的参数与注解封装在 `parameterHandlers` 中，为后面的网络请求做准备。

先用流程图梳理一下刚才的思路：



分析到这里，我们总算是明白了最初的两行代码原来干了这么多事情，J神真的是流弊啊！接下来我们就来看一下网络请求部分。

使用 OkHttpClient 进行网络请求

回头看一下上一小节讲解 `create` 方法时我们有这一行代码：

```
OkHttpClient<Object> okHttpClient = new OkHttpClient<>(serviceMethod, args);
```

他将我们刚才得到的 `serviceMethod` 与我们实际传入的参数传递给了 `OkHttpClient`，接下来就来瞧瞧这个类做了些什么？

```

final class OkHttpClient<T> implements Call<T> {
    private final ServiceMethod<T, ?> serviceMethod;
    private final Object[] args;

    private volatile boolean canceled;

    // All guarded by this.
    private okhttp3.Call rawCall;
}

```

```

private Throwable creationFailure; // Either a RuntimeException or IOException.

private boolean executed;

    OkHttpCall (ServiceMethod<T, ?> serviceMethod, Object[] args) {

        this.serviceMethod = serviceMethod;

        this.args = args;

    }

}

```

很可惜，我们好像没有看到比较有用的东西，只是将传进来的参数进行了赋值，那我们就接着看 `create` 方法中的最后一行吧！

callAdapter 的使用

`create` 方法的最后一行是这样的：

```
return serviceMethod.callAdapter.adapt(okHttpCall);
```

最后是调用了 `callAdapter` 的 `adapt` 方法，上面我们讲到 `Retrofit` 在决定使用什么 `callAdapter` 的时候是看我们在接口中定义的方法的返回值的，而在我们的例子中使用的是 `RxJava2CallAdapter`，因此我们就直接看该类中的 `adapt` 方法吧！

```

@Override public Object adapt(Call<R> call) {

    Observable<Response<R>> responseObservable = isAsync

        ? new CallEnqueueObservable<>(call)

        : new CallExecuteObservable<>(call);

    Observable<?> observable;

    if (isResult) {

        observable = new ResultObservable<>(responseObservable);

    } else if (isBody) {

        observable = new BodyObservable<>(responseObservable);

    } else {

        observable = responseObservable;

    }

    if (scheduler != null) {

```

```

        observable = observable.subscribeOn(scheduler);
    }

    if (isFlowable) {
        return observable.toFlowable(BackpressureStrategy.LATEST);
    }

    if (isSingle) {
        return observable.singleOrError();
    }

    if (isMaybe) {
        return observable.singleElement();
    }

    if (isCompletable) {
        return observable.ignoreElements();
    }

    return observable;
}

```

首先在 **adapt** 方法中会先判断是同步请求还是异步请求，这里我们以同步请求为例，直接看 **CallExecuteObservable**。

```

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();

        observer.onSubscribe(new CallDisposable(call));
    }
}

```

```

        boolean terminated = false;

        try {

            //重点看这里

            Response<T> response = call.execute();

            if (!call.isCanceled()) {

                observer.onNext(response);

            }

            if (!call.isCanceled()) {

                terminated = true;

                observer.onComplete();

            }

        } catch (Throwable t) {

            Exceptions.throwIfFatal(t);

            if (terminated) {

                RxJavaPlugins.onError(t);

            } else if (!call.isCanceled()) {

                try {

                    observer.onError(t);

                } catch (Throwable inner) {

                    Exceptions.throwIfFatal(inner);

                    RxJavaPlugins.onError(new CompositeException(t, inner));

                }

            }

        }

    }

}

private static final class CallDisposable implements Disposable {

    private final Call<?> call;

```

```

    CallDisposable(Call<?> call) {

        this.call = call;

    }

    @Override public void dispose() {

        call.cancel();

    }

    @Override public boolean isDisposed() {

        return call.isCanceled();

    }

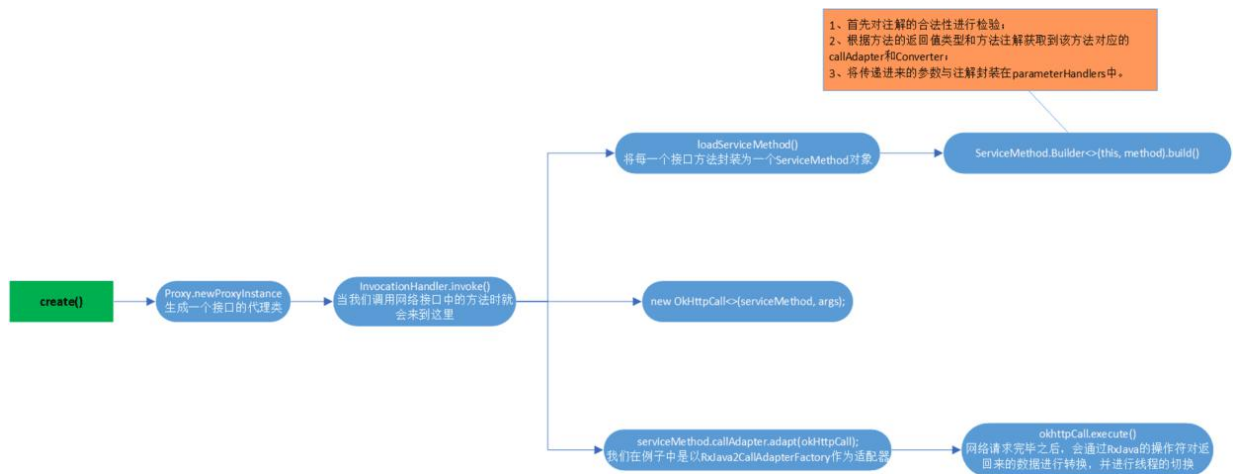
}

}

}

```

在 `subscribeActual` 方法中去调用了 `OKHttpClient` 的 `execute` 方法开始进行网络请求，网络请求完毕之后，会通过 `RxJava` 的操作符对返回来的数据进行转换，并进行线程的切换，至此，`Retrofit` 的一次使用也就结束了。最后我们再用一张完整的流程图总结上述的几个过程。

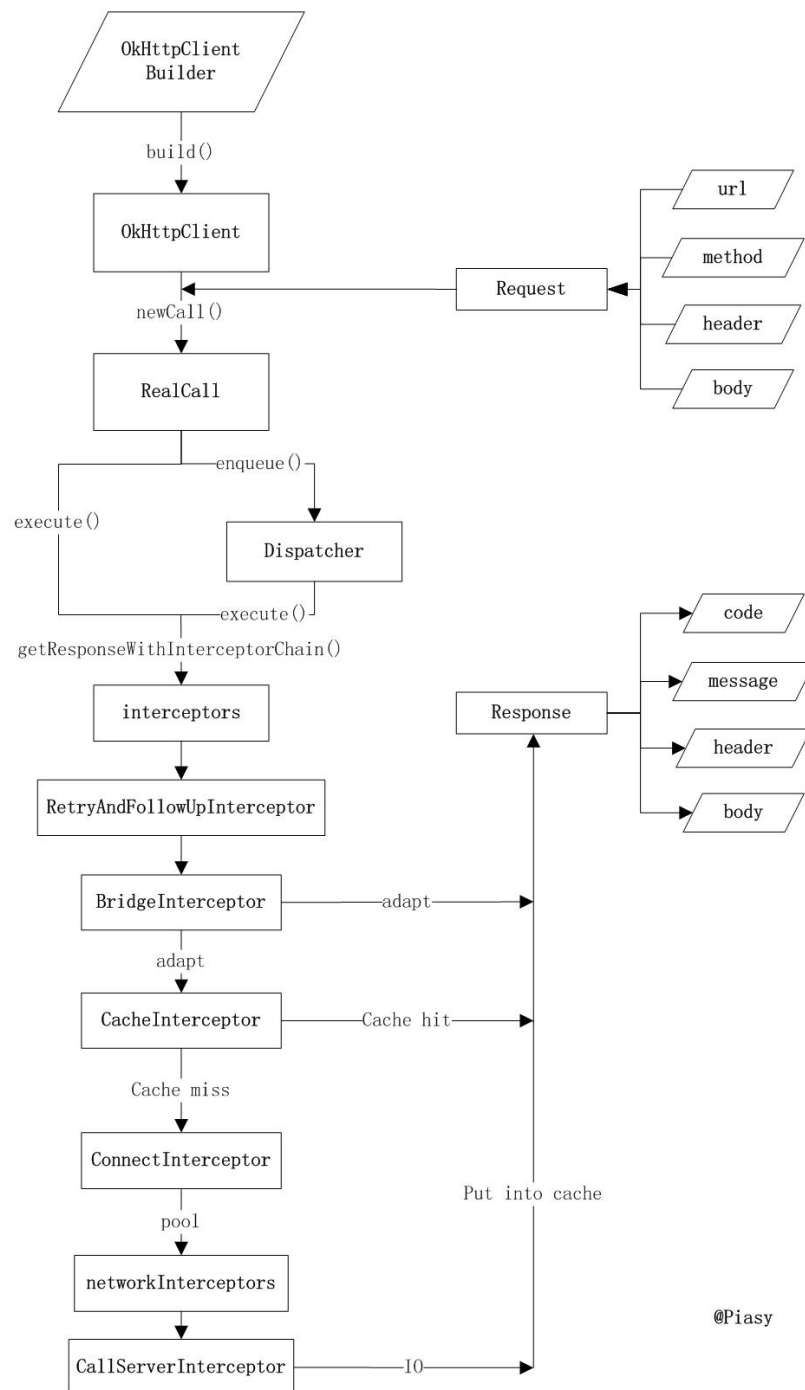


9. OkHttp

1，整体思路

从使用方法出发,首先是怎么使用,其次是我们使用的功能在内部是如何实现的,实现方案上有什么技巧,有什么范式。全文基本上是对 **OkHttp** 源码的一个分析与导读,非常建议大家下载 **OkHttp** 源码之后,跟着本文,过一遍源码。对于技巧和范式,由于目前我的功力还不到位,分析内容没多少,欢迎大家和我一起讨论。

首先放一张完整流程图(看不懂没关系,慢慢往后看):



2，基本用例

来自 [OkHttp 官方网站](#)。

2.1，创建 OkHttpClient 对象

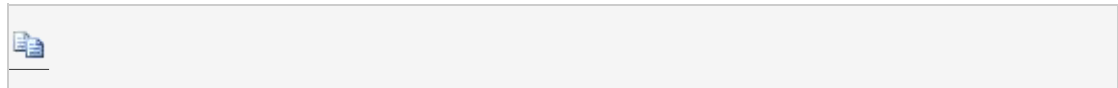
```
OkHttpClient client = new OkHttpClient();
```

咦，怎么不见 **builder**? 莫急，且看其构造函数：

```
public OkHttpClient() {  
    this(new Builder());  
}
```

原来是方便我们使用，提供了一个“快捷操作”，全部使用了默认的配置。

OkHttpClient.Builder 类成员很多，后面我们再慢慢分析，这里先暂时略过：



```
public Builder() {  
    dispatcher = new Dispatcher();  
    protocols = DEFAULT_PROTOCOLS;  
    connectionSpecs = DEFAULT_CONNECTION_SPECS;  
    proxySelector = ProxySelector.getDefault();  
    cookieJar = CookieJar.NO_COOKIES;  
    socketFactory = SocketFactory.getDefault();  
    hostnameVerifier = OkHostnameVerifier.INSTANCE;  
    certificatePinner = CertificatePinner.DEFAULT;  
    proxyAuthenticator = Authenticator.NONE;  
    authenticator = Authenticator.NONE;  
    connectionPool = new ConnectionPool();  
    dns = Dns.SYSTEM;  
    followSslRedirects = true;
```

```

followRedirects = true;
retryOnConnectionFailure = true;
connectTimeout = 10_000;
readTimeout = 10_000;
writeTimeout = 10_000;
}

```



2.2, 发起 HTTP 请求



```

String run(String url) throws IOException {
    Request request = new Request.Builder()
        .url(url)
        .build();

    Response response = client.newCall(request).execute();
    return response.body().string();
}

```



OkHttpClient 实现了 **Call.Factory**, 负责根据请求创建新的 **Call**, 在 拆轮子系列: 拆 Retrofit 中我们曾和它发生过一次短暂的遭遇:

callFactory 负责创建 HTTP 请求, HTTP 请求被抽象为了 **okhttp3.Call** 类, 它表示一个已经准备好, 可以随时执行的 HTTP 请求

那我们现在就来看看它是如何创建 **Call** 的：

```
/**
 * Prepares the {@code request} to be executed at some point in the future.
 */
@Override public Call newCall(Request request) {
    return new RealCall(this, request);
}
```

如此看来功劳全在 **RealCall** 类了，下面我们一边分析同步网络请求的过程，一边了解 **RealCall** 的具体内容。

2.2.1，同步网络请求

我们首先看 **RealCall#execute**：



```
@Override public Response execute() throws IOException {
    synchronized (this) {
        if (executed) throw new IllegalStateException("Already Executed"); // (1)
        executed = true;
    }
    try {
        client.dispatcher().executed(this); // (2)
        Response result = getResponseWithInterceptorChain(); // (3)
        if (result == null) throw new IOException("Canceled");
        return result;
    } finally {
        client.dispatcher().finished(this); // (4) }
}
```




这里我们做了 4 件事：

1. 检查这个 `call` 是否已经被执行了，每个 `call` 只能被执行一次，如果想要一个完全一样的 `call`，可以利用 `call#clone` 方法进行克隆。
2. 利用 `client.dispatcher().executed(this)` 来进行实际执行，
`dispatcher` 是刚才看到的 `OkHttpClient.Builder` 的成员之一，它的文档说自己是异步 HTTP 请求的执行策略，现在看来，同步请求它也有掺和。
3. 调用 `getResponseWithInterceptorChain()` 函数获取 HTTP 返回结果，从函数名可以看出，这一步还会进行一系列“拦截”操作。
4. 最后还要通知 `dispatcher` 自己已经执行完毕。

`dispatcher` 这里我们不过度关注，在同步执行的流程中，涉及到 `dispatcher` 的内容只不过是告知它我们的执行状态，比如开始执行了（调用 `executed`），比如执行完毕了（调用 `finished`），在异步执行流程中它会有更多的参与。

真正发出网络请求，解析返回结果的，还是 `getResponseWithInterceptorChain`：



```
private Response getResponseWithInterceptorChain() throws IOException {  
    // Build a full stack of interceptors.  
    List<Interceptor> interceptors = new ArrayList<>();  
    interceptors.addAll(client.interceptors());  
    interceptors.add(retryAndFollowUpInterceptor);  
    interceptors.add(new BridgeInterceptor(client.cookieJar()));  
    interceptors.add(new CacheInterceptor(client.internalCache()));  
    interceptors.add(new ConnectInterceptor(client));  
    if (!retryAndFollowUpInterceptor.isForWebSocket()) {  

```

```
        interceptors.addAll(client.networkInterceptors());  
    }  
  
    interceptors.add(new CallServerInterceptor(  
        retryAndFollowUpInterceptor.isForWebSocket()));  
  
    Interceptor.Chain chain = new RealInterceptorChain(  
        interceptors, null, null, null, 0, originalRequest);  
    return chain.proceed(originalRequest);  
}
```



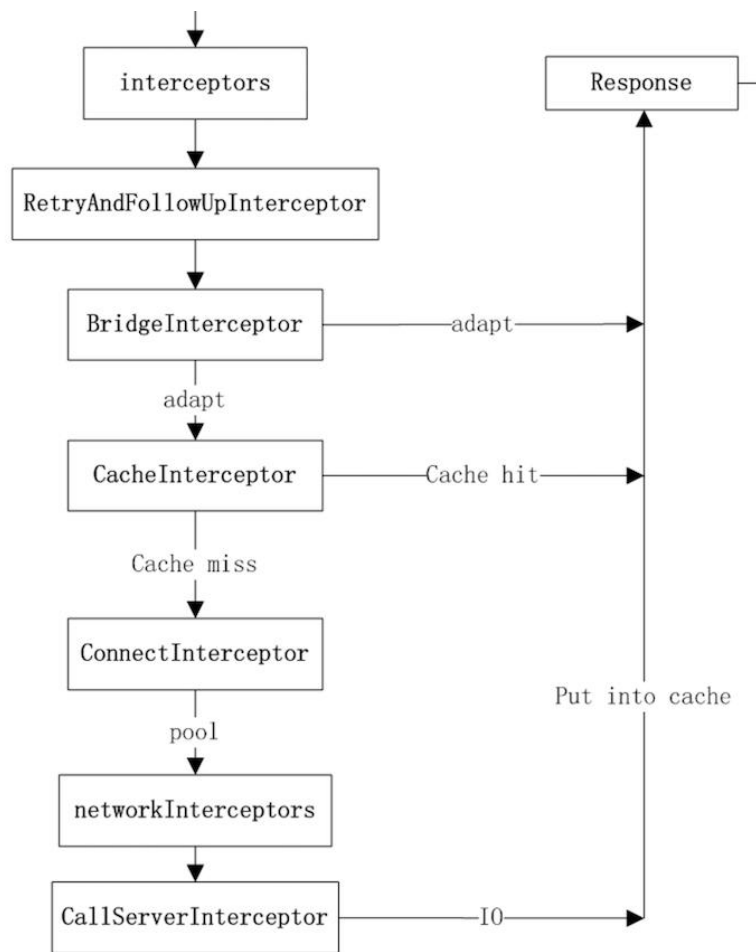
在 [OkHttp 开发者之一介绍 OkHttp 的文章](#)里面，作者讲到：

the whole thing is just a stack of built-in interceptors.

可见 **Interceptor** 是 **OkHttp** 最核心的一个东西，不要误以为它只负责拦截请求进行一些额外的处理（例如 **cookie**），实际上它把实际的网络请求、缓存、透明压缩等功能都统一了起来，每一个功能都只是一个 **Interceptor**，它们再连接成一个 **Interceptor.Chain**，环环相扣，最终圆满完成一次网络请求。

从 `getResponseWithInterceptorChain` 函数我们可以看到，

Interceptor.Chain 的分布依次是：



1. 在配置 `OkHttpClient` 时设置的 `interceptors`;
2. 负责失败重试以及重定向的 `RetryAndFollowUpInterceptor`;
3. 负责把用户构造的请求转换为发送到服务器的请求、把服务器返回的响应转换为用户友好的响应的 `BridgeInterceptor`;
4. 负责读取缓存直接返回、更新缓存的 `CacheInterceptor`;
5. 负责和服务建立连接的 `ConnectInterceptor`;
6. 配置 `OkHttpClient` 时设置的 `networkInterceptors`;
7. 负责向服务器发送请求数据、从服务器读取响应数据的 `CallServerInterceptor`。

在这里，位置决定了功能，最后一个 **Interceptor** 一定是负责和服务端实际通讯的，重定向、缓存等一定是在实际通讯之前的。

责任链模式在这个 **Interceptor** 链条中得到了很好的实践（感谢 **Stay** 一语道破，自愧弗如）。

它包含了一些命令对象和一系列的处理对象，每一个处理对象决定它能处理哪些命令对象，它也知道如何将它不能处理的命令对象传递给该链中的下一个处理对象。该模式还描述了往该处理链的末尾添加新的处理对象的方法。

对于把 **Request** 变成 **Response** 这件事来说，每个 **Interceptor** 都可能完成这件事，所以我们循着链条让每个 **Interceptor** 自行决定能否完成任务以及怎么完成任务（自力更生或者交给下一个 **Interceptor**）。这样一来，完成网络请求这件事就彻底从 **RealCall** 类中剥离了出来，简化了各自的责任和逻辑。两个字：优雅！

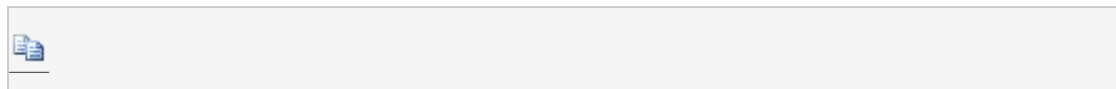
责任链模式在安卓系统中也有比较典型的实践，例如 **view** 系统对点击事件

（**TouchEvent**）的处理，具体可以参考 [Android 设计模式源码解析之责任链模式中相关的分析](#)。

回到 **OkHttp**，在这里我们先简单分析一

下 **ConnectInterceptor** 和 **CallServerInterceptor**，看看 **OkHttp** 是怎么进行和服务器的实际通信的。

2.2.1.1，建立连接：**ConnectInterceptor**



```
@Override public Response intercept(Chain chain) throws IOException {  
    RealInterceptorChain realChain = (RealInterceptorChain) chain;  
    Request request = realChain.request();  
    StreamAllocation streamAllocation = realChain.streamAllocation();
```

```
// We need the network to satisfy this request. Possibly for validating a
conditional GET.

boolean doExtensiveHealthChecks = !request.method().equals("GET");

HttpCodec httpCodec = streamAllocation.newStream(client,
doExtensiveHealthChecks);

RealConnection connection = streamAllocation.connection();

return realChain.proceed(request, streamAllocation, httpCodec, connection);
}
```



实际上建立连接就是创建了一个 **HttpCodec** 对象，它将在后面的步骤中被使用，那它又是何方神圣呢？它是对 **HTTP** 协议操作的抽象，有两个实现：

Http1Codec 和 **Http2Codec**，顾名思义，它们分别对应 **HTTP/1.1** 和 **HTTP/2** 版本的实现。

在 **Http1Codec** 中，它利用 **Okio** 对 **Socket** 的读写操作进行封装，**Okio** 以后有机会再进行分析，现在让我们对它们保持一个简单地认识：它对 **java.io** 和 **java.nio** 进行了封装，让我们更便捷高效的进行 **IO** 操作。

而创建 **HttpCodec** 对象的过程涉及到 **StreamAllocation**、**RealConnection**，代码较长，这里就不展开，这个过程概括来说，就是找到一个可用的 **RealConnection**，再利用 **RealConnection** 的输入输出

（**BufferedSource** 和 **BufferedSink**）创建 **HttpCodec** 对象，供后续步骤使用。

2.2.1.2，发送和接收数据：CallServerInterceptor



```
@Override public Response intercept(Chain chain) throws IOException {
    HttpCodec httpCodec = ((RealInterceptorChain) chain).httpStream();
```

```

StreamAllocation streamAllocation = ((RealInterceptorChain)
chain).streamAllocation();

Request request = chain.request();

long sentRequestMillis = System.currentTimeMillis();

httpCodec.writeRequestHeaders(request);

if (HttpMethod.permitsRequestBody(request.method()) && request.body() != null)
{
    Sink requestBodyOut = httpCodec.createRequestBody(request,
request.body().contentLength());

    BufferedSink bufferedRequestBody = Okio.buffer(requestBodyOut);

    request.body().writeTo(bufferedRequestBody);

    bufferedRequestBody.close();
}

httpCodec.finishRequest();

Response response = httpCodec.readResponseHeaders()
    .request(request)
    .handshake(streamAllocation.connection().handshake())
    .sentRequestAtMillis(sentRequestMillis)
    .receivedResponseAtMillis(System.currentTimeMillis())
    .build();

if (!forWebSocket || response.code() != 101) {
    response = response.newBuilder()
        .body(httpCodec.openResponseBody(response))
        .build();
}

if ("close".equalsIgnoreCase(response.request().header("Connection"))
|| "close".equalsIgnoreCase(response.header("Connection"))) {
    streamAllocation.noNewStreams();
}

```

```
// 省略部分检查代码
```

```
return response;
```

```
}
```



我们抓住主干部分：

1. 向服务器发送 **request header**;
2. 如果有 **request body**，就向服务器发送;
3. 读取 **response header**，先构造一个 **Response** 对象;
4. 如果有 **response body**，就在 3 的基础上加上 **body** 构造一个新的 **Response** 对象;

这里我们可以看到，核心工作都由 **HttpCodec** 对象完成，而 **HttpCodec** 实际上利用的是 **Okio**，而 **Okio** 实际上还是用的 **Socket**，所以没什么神秘的，只不过一层套一层，层数有点多。

其实 **Interceptor** 的设计也是一种分层的思想，每个 **Interceptor** 就是一层。

为什么要套这么多层呢？分层的思想在 **TCP/IP** 协议中就体现得淋漓尽致，分层简化了每一层的逻辑，每层只需要关注自己的责任（单一原则思想也在此体现），而各层之间通过约定的接口/协议进行合作（面向接口编程思想），共同完成复杂的任务。

简单应该是我们的终极追求之一，尽管有时为了达成目标不得不复杂，但如果有一种更简单的方式，我想应该没有人不愿意替换。

2.2.2，发起异步网络请求



```
client.newCall(request).enqueue(new Callback() {  
    @Override  
    public void onFailure(Call call, IOException e) {  
    }  
  
    @Override  
    public void onResponse(Call call, Response response) throws IOException {  
        System.out.println(response.body().string());  
    }  
});  
  
// RealCall#enqueue  
@Override public void enqueue(Callback responseCallback) {  
    synchronized (this) {  
        if (executed) throw new IllegalStateException("Already Executed");  
        executed = true;  
    }  
    client.dispatcher().enqueue(new AsyncCall(responseCallback));  
}  
  
// Dispatcher#enqueueSynchronized void enqueue(AsyncCall call) {  
    if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerHost) {  
        runningAsyncCalls.add(call);  
        executorService().execute(call);  
    } else {  
        readyAsyncCalls.add(call);  
    }  
}
```



这里我们就能看到 **dispatcher** 在异步执行时发挥的作用了，如果当前还能执行一个并发请求，那就立即执行，否则加入 **readyAsyncCalls** 队列，而正在执行

的请求执行完毕之后，会调用 `promoteCalls()` 函数，来把 `readyAsyncCalls` 队列中的 `AsyncCall` “提升”为 `runningAsyncCalls`，并开始执行。

这里的 `AsyncCall` 是 `RealCall` 的一个内部类，它实现了 `Runnable`，所以可以被提交到 `ExecutorService` 上执行，而它在执行时会调用 `getResponseWithInterceptorChain()` 函数，并把结果通过 `responseCallback` 传递给上层使用者。

这样看来，同步请求和异步请求的原理是一样的，都是在 `getResponseWithInterceptorChain()` 函数中通过 `Interceptor` 链条来实现的网络请求逻辑，而异步则是通过 `ExecutorService` 实现。

2.3，返回数据的获取

在上述同步（`Call#execute()` 执行之后）或者异步（`Callback#onResponse()` 回调中）请求完成之后，我们就可以从 `Response` 对象中获取到响应数据了，包括 `HTTP status code`，`status message`，`response header`，`response body` 等。

这里 `body` 部分最为特殊，因为服务器返回的数据可能非常大，所以必须通过数据流的方式来进行访问（当然也提供了诸如 `string()` 和 `bytes()` 这样的方法将流内的数据一次性读取完毕），而响应中其他部分则可以随意获取。

响应 `body` 被封装到 `ResponseBody` 类中，该类主要有两点需要注意：

1. 每个 `body` 只能被消费一次，多次消费会抛出异常；
2. `body` 必须被关闭，否则会发生资源泄漏；

在 2.2.1.2，发送和接收数据：CallServerInterceptor 小节中，我们就看过了 `body` 相关的代码：

```
if (!forWebSocket || response.code() != 101) {  
    response = response.newBuilder()  
        .body(httpCodec.openResponseBody(response))  
        .build();  
}
```

由 `HttpCodec#openResponseBody` 提供具体 HTTP 协议版本的响应 `body`，而 `HttpCodec` 则是利用 `Okio` 实现具体的数据 IO 操作。

这里有一点值得一提，`OkHttp` 对响应的校验非常严格，`HTTP status line` 不能有任何杂乱的数据，否则就会抛出异常，在我们公司项目的实践中，由于服务器的问题，偶尔 `status line` 会有额外数据，而服务端的问题也毫无头绪，导致我们不得不忍痛继续使用 `URLConnection`，而后者在一些系统上又存在各种其他的问题，例如魅族系统发送 `multi-part form` 的时候就会出现没有响应的问题。

2.4, HTTP 缓存

在 [2.2.1, 同步网络请求](#) 小节中，我们已经看到了 `Interceptor` 的布局，在建立连接、和服务器通讯之前，就是 `CacheInterceptor`，在建立连接之前，我们检查响应是否已经被缓存、缓存是否可用，如果是则直接返回缓存的数据，否则就进行后面的流程，并在返回之前，把网络的数据写入缓存。

这块代码比较多，但也很直观，主要涉及 HTTP 协议缓存细节的实现，而具体的缓存逻辑 `OkHttp` 内置封装了一个 `Cache` 类，它利用 `DiskLruCache`，用磁盘上的有限大小空间进行缓存，按照 `LRU` 算法进行缓存淘汰，这里也不再展开。

我们可以在构造 `OkHttpClient` 时设置 `Cache` 对象，在其构造函数中我们可以指定目录和缓存大小：

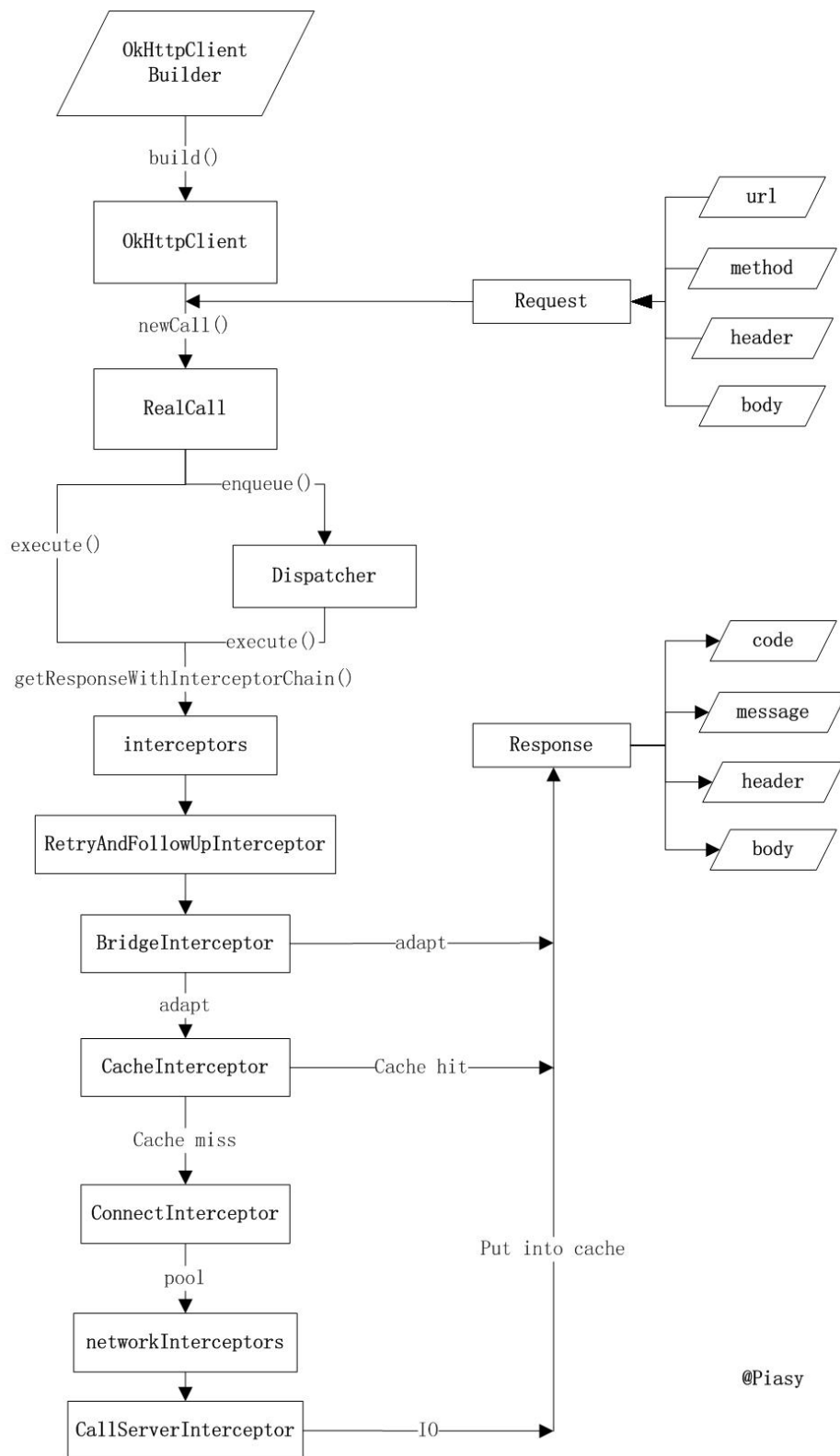
```
public Cache(File directory, long maxSize);
```

而如果我们对 `OkHttp` 内置的 `Cache` 类不满意，我们可以自行实现 `InternalCache` 接口，在构造 `OkHttpClient` 时进行设置，这样就可以使用我们自定义的缓存策略了。

3，总结

`OkHttp` 还有很多细节部分没有在本文展开，例如 `HTTP2/HTTPS` 的支持等，但建立一个清晰的概览非常重要。对整体有了清晰认识之后，细节部分如有需要，再单独深入将更加容易。

在文章最后我们再来回顾一下完整的流程图：



- **OkHttpClient** 实现 **Call.Factory**，负责为 **Request** 创建 **Call**;

- `RealCall` 为具体的 `Call` 实现, 其 `enqueue()` 异步接口通过 `Dispatcher` 利用 `ExecutorService` 实现, 而最终进行网络请求时和同步 `execute()` 接口一致, 都是通过 `getResponseWithInterceptorChain()` 函数实现;
- `getResponseWithInterceptorChain()` 中利用 `Interceptor` 链条, 分层实现缓存、透明压缩、网络 IO 等功能;