更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

设计思想解读开源框架

目录

第一章、 热修复设计 2
第一节、 AOT/JIT & dexopt 与 dex2oat 2
第二节、 热修复设计之 CLASS_ISPREVERIFIED 问题 12
第三节、热修复设计之热修复原理 27
第四节、Tinker 的集成与使用（自动补丁包生成） 33
第二章、 插件化框架设计 78
第一节、 Class 文件与 Dex 文件的结构解读 78
第二节、 Android 资源加载机制详解 85
第三节、 四大组件调用原理 98
第四节、 so 文件加载机制 123
第五节、 Android 系统服务实现原理 158
第三章、 组件化框架设计 179
第一节、阿里巴巴开源路由框——ARouter 原理分析 179
第二节、APT 编译时期自动生成代码&动态类加载 195
第三节、 Java SPI 机制 220
第四节、 AOP&IOC 235
第五节、 手写组件化架构 255
第四章、图片加载框架 290
4.1 图片加载框架选型 290
4.2 Glide 原理分析 386
4.3 手写图片加载框架实战 664
第五章、网络访问框架设计 664
5.1 网络通信必备基础 664
5.2 OkHttp 源码解读 698
5.3 Retrofit 源码解析 751
第六章、 RXJava 响应式编程框架设计 784
6.1 链式调用 785
6.2 扩展的观察者模式 791
6.3 事件变换设计 800
6.4 Scheduler 线程控制 814
第七章、 IOC 架构设计 819
7.1 依赖注入与控制反转 819
7.2 ButterKnife 原理上篇、中篇、下篇 822
7.3 Dagger 架构设计核心解密 897
第八章、 Android 架构组件 Jetpack 916
8.1 LiveData 原理 916
8.2 Navigation 如何解决 tabLayout 问题 925
8.3 ViewModel 如何感知 View 生命周期及内核原理 925
8.4 Room 架构方式方法 944
8.5 dataBinding 为什么能够支持 MVVM 956

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

8.6 WorkManager 内核揭秘.....987
8.7 Lifecycles 生命周期.....999

第一章、热修复设计

第一节、AOT/JIT & dexopt 与 dex2oat

一、AOT/JIT

一个程序的编译过程可以是步骤迭代式的，即每一轮步骤结束后得到的结果都可独立运行，比如，先构造 AST 再输出字节码，中间状态 AST 也是可以解释执行的。由于编译的本质就是代码转换，因此对一个语言可以有多个独立的编译器，每个负责一轮步骤

AOT Compiler 和 JIT Compiler 就是针对编译形式做的分类：

AOT：Ahead Of Time，指在运行前编译，比如普通的静态编译

JIT：Just In Time，指在运行时编译，边运行边编译，比如 java 虚拟机在运行时就用到 JIT 技术

JIT 可能知道的人多些，AOT 这个名词就相对少见一些了，其实除了 JIT，剩下的都是 AOT。wiki 上 JIT 的解释也比 AOT 详尽很多，如果按 wiki 上的理解，一般来说，是从形式上来区分这两个概念，即看编译是不是在“运行时”进行

然而，这两个概念又有模糊性，问题在于这个“运行时”怎么来区分，比方说，从这个概念来看，python 是用到 JIT 技术的，因为：

```
... import a ...
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

当执行到 import a 的时候，当然是运行时，这时候如果只找到了 a.py，则会进行编译工作，并生成 a.pyc，这就是 python 的 JIT 特性，但是一般来说，认为 python 的 JIT 是 psyco、pypy 之类，并不认为 python 本身的动态性属于 JIT 范畴，或者说，它的这种"形式上"的 JIT 特性不纳入讨论范围。其他脚本语言，动态语言也有类似的情况。具体原因我觉得有几点

首先被主流理论认定的 JIT 编译器对于被其编译的语言来说属于附加品，也就是说，就算去掉 JIT，并不影响语言本身的运行，例如 java，如果关闭 JIT，依然可以解释执行，而上述 python 的运行时 import 的特性虽然形式上符合 JIT，但这个机制是语言本身规定的，如果去掉，语言（的主流实现）就不完整了。反过来说，如果 python 采用源码直接解析执行，则编译为字节码的行为就可以看做是 JIT，因为做不做都不影响解析执行过程

其次，python 的这种编译并非每次执行都会进行，因为一般来说会生成字节码结果 pyc 文件存在磁盘，它更像是对 java 源代码转 class 文件这一过程的惰性化，在需要的时候进行

最后，JIT 会消耗运行时资源，可能导致进程卡顿，而 java 等语言之所以引入 JIT，是因为 JIT 对字节码编译后能以更快的速度运行，卡顿的时间能补救回来，因此从工程角度讲，JIT 几乎就等于是运行时优化（虽然从概念和形式上并非如此），而 python 的 import 就只有卡顿，对速度没啥好处

于是，虽然从概念来说，上面的例子的确符合 JIT，但一般来说也不这么认为，出发角度问题，说 python 自带 JIT 特性或没有 JIT 都算说得通的

之所以先举这个例子，因为我觉得能体现 AOT 和 JIT 概念的对立和统一，对立是形式上的，以"运行"为分界线，而统一则是说，其实所有需要执行的指令序列，都是需要先编译再执行的，比如 import a，这个相对于整个进程当然是 JIT，但

相对于 a.py 这个模块（python 进程首次 import 某个模块时会执行它）不妨看做 AOT，如果有人觉得这么做不妥，那换个更明显的例子，如果一个 python 程序的所有 import 都在进程开启时立即运行，然后才进入执行，那按照概念来说，这是 JIT，因为进程已经开始运行了，但是，为什么不能看做是先编译再执行的 AOT 模式，只是整个过程被批处理化了呢？

带着这个问题再考虑很多资料（包括 wiki）对 JIT 的另一个描述，JIT 是在运行时将解释执行的语言（比如字节码）编译成机器指令，以提高运行速度。这个看法在前面的某篇也提过，的确很多 JIT 编译器，比如 java 的就是这么干的（我们下面就拿 java 举例），但是，既然字节码编译成机器指令可以提高速度，为何一定要放在运行时进行，做成 AOT 模式不是可以运行得更流畅吗，而且还能一次编译，N 次执行，为啥非要做成运行时做，JIT 本来是要提高运行速度，但这岂不是降低了效率？

这种看法是有道理的，事实上，java 的确有一些 AOT 编译器，可以将字节码甚至 java 源码直接编译成机器指令的可执行文件，微软当初的 VJ++似乎就这么搞的，和 sun 打了很久的架，sun 还喊出了 pure java（纯粹的 java，即按照 sun 的设计理念和标准来实现 java）的口号，有兴趣可以去搜一下这段历史，挺搞笑的

另一方面，sun 的 jvm 虽然采用了 JIT 编译，但同时也提供了 client 和 server 模式，在 server 模式下，虚拟机在一开始执行的时候会先尽可能多地对字节码进行编译，且优化程度也尽量高，这样可以使得服务器在运行过程中能尽量少卡顿，根据上面的讨论，这实际上相当于 AOT 批处理了。client 模式下则不会这样做，主要是为了尽量缩短启动延迟，提高用户体验

BAT交流/2246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

顺便说一句，对于 JIT 将字节码编译成机器指令，wiki 的描述比较暧昧，有时候用 machine code，有时候用 native code，比方说我们用 java 实现一个 A 语言的虚拟机，解释 A 的字节码执行，并将字节码编译成 java 自己的字节码，这也是 JIT，因为 A 跑在 jvm 上，则 java 字节码就看做是 native code，而 machine code 这个 machine 也不见得是真实机器，jvm 也是一种机器

由于 JIT 编译耗费运行时间，则对于某些优化点就无法做到百分百支持，必须在代码优化和执行卡顿之间做一个权衡，AOT 就没有这个问题，另外，AOT 可以做到编译后持久化到存储，而 JIT 一般是每运行一次就会搞一遍重复的编译

如果我们不考虑 AOT 本身耗费的时间（比如编译一次，N 次运行），也不考虑使用上的方便性（AOT 可能会有多次编译过程），那是不是可以认为，AOT 编译可以完全替换 JIT 编译，JIT 就完全没必要了，实际情况当然不是这样，JIT 还是有它的优势和必要性的，否则研究它的那群人岂不都是傻子

从动静态来看这个问题，AOT 是静态编译，而 JIT 是运行时动态编译，则 JIT 的优势在于，它不但能看到静态信息（代码），还能看到运行时的情况，这就是 JIT 的优势。接下来讨论的 JIT 是一种狭义的 JIT，即在 AOT 搞不定的地方使用的 JIT，而非上述形式上的

关于 JIT 的优势，wiki 上给出了四点理由，但有意思的是，其中有两条连它自己都承认并非只有 JIT 能做，也就是说至少理论上，用 AOT 实现（或部分实现）是可行的，这四条是：

1、JIT 可以根据当前的硬件情况实时编译成最优机器指令，比如 cpu 中如果含 FPU，MMX，SSE2，或者 Intel cpu 的并行计算特性，则可以做到同一份字节码，在不同机器运行时最大限度利用硬件资源。而如果是 AOT 编译一个程序放出去

,12246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

给不同用户使用，就只能去兼容特性最少的 cpu，或者内部实现多个版本

2、JIT 可以根据当前进程实际运行状态，将字节码编译成适合最优化的机器指令序列。wiki 认为静态编译也可以通过分析 profile 来实现这方面的优化（可能有点麻烦）

3、当程序需要支持动态链接时，即在静态编译阶段，可能不知道运行时会引入什么样的代码来和程序协作执行，这时候就只能依靠 JIT

4、考虑到垃圾收集，JIT 可以根据进程中的内存实际情况来调整代码，使得 cache 能更充分地使用，wiki 认为静态编译也可以做到，但 JIT 做起来更容易实现

对于第一条，JIT 的确可以实现这种优化，但是 AOT 一样可以实现，虽然 AOT 编译一个程序给不同用户执行无法做到，但是可以编译字节码发布，用户使用时再根据当前机器再做一次 AOT

对于第二条，首先我认为大多数程序的运行状态不会经常变动，比如同一个程序有时候是整数计算居多，有时候是浮点计算居多，一般来说程序应用场景是固定的；其次对于特定场景也可以 AOT

对于第三条，的确动态链接的全文静态优化 AOT 无法做到，但是如上篇所说，必要时候我们可以直接砍掉语言的动态性，再者静态编译时候也不是什么都感知不到，比如 C 语言做静态链接时，至少是知道头文件的，动态性没那么强

对于第四条，AOT 也是有可能实现的，虽然麻烦很多。另一方面，静态编译时也有指令乱序来提高 cache 使用效果，再者这块也和垃圾收集算法、程序本身的局部性有很大关系，如果程序本身写的烂，这个调整效果可能也比较有限

所以我觉得，这四条虽然都有道理，但没精确说到点子上。再来审视这个问题，我们可以看出，从理论上讲，AOT 可以完全代替 JIT，因为一个进程的状态是有限的，AOT 可以预测所有可能情况并进行优化，实际运行时的状态不会超出 AOT 的预测，采用最优代码执行即可，而 JIT 在这里的优势就是，它能精准地得知运

BA/2246

行时状态，而不是像 AOT 那样预测，成本更低，如果一个 AOT 优化的成本过高，则应该选择 JIT。AOT 不是不能做，而是不可行

JIT 相关的资料，相比 wiki 我更推荐这篇论文：《Representation-based Just-in-time Specialization and the Psyco prototype for Python》 by Armin Rigo，这个论文是以 python 和其 JIT 插件库 psyco 为例来分析，论文题目中的单词 Specialization 可谓画龙点睛，它指出至少在动态类型语言中，JIT 的关键作用之一是特化，用上篇的话说，就是动态行为静态化，而这些场景中 AOT 不可行的原因是它很难找到特化的方向，而枚举所有特化是不可行的

一个典型的特化案例，也是论文中提到的，假设有一个函数 f(x,y)，则对于 x 的输入 x1,x2,x3...，我们可以特化这个函数为 f1(y),f2(y),f3(y)...，其中 fk(y)在功能上对应 f(xk,y)，这样一来，每个 fk 可以单独地做优化，与其他函数无关，而特化后的函数列表至少不会比原来的 f(x,y)慢。唯一的问题是，x 的取值可能很多，比如 x 是一个 int，则如果采用 AOT 方式来特化，则需要编译 42 亿多个函数，这显然是不现实的，但是 JIT 就有可能对这个场景做优化，原因在于，x 的取值虽然很多，但在一个具体运行过程中范围相对小，甚至是很小，这符合二八定律

于是，在运行时我们可以对函数 f 做监控，统计每次输入的 x 的值，如果发现这些值的分布不平均，比如 x 为 123 的情况占大多数，则动态特化一个 f123(y)，对其进行高度优化，然后修改 f 函数为：

```
func f(x, y):
    if x == 123:
        return f123(y)
    ... //f 的正常流程
```

于是只需要一个特化函数，就能带来运行时效率的提升，这就是 JIT 特化的优势

对很多程序来说，对这种数值做监控和特化可能性价比不高，因为不是每个函数的输入值范围都呈现不平衡状态，或者说不是那么明显，但上面这个例子中，x 和 y 不一定是变量，也可以是类型，这样一来对动态类型语言就有很大的意义

前面讲过，在 C++中可以用模板来实现鸭子类型，实质是通过代码替换来实现类型静态化，C++这个方式虽然效率高，但渠道是通过静态编译中的全文分析，是 AOT 编译，如果改成稍微动态性强一些的语言，就用不上了。在动态类型中，一个函数如果有 k 个参数，有 n 个可能类型，则 AOT 需要将一个函数扩展为 n^k 个特化实例，n 和 k 稍大一点就不可操作了，何况本身就是动态类型，n 的范围都不一定在编译期能知道

对这种场景，JIT 就可以通过统计的方式来选择性地特化，这个的可行性和现实意义更大，原因在于，程序员在用动态类型写程序的时候，比如写一个函数：

```
func f(x, y):

    return x + y
```

理论上，这个函数可以接受任意类型的 x 和 y，只要 x 能和 y 相加即可，但具体到一个确定的程序，这个函数的业务意义一般是固定的，或者是做字符串拼接，或者是数值相加，很少说写一个函数，接收八竿子打不着的不同的类型还能运算，而且还是程序员刻意这么设计，就像前面讲过的 C++模板的二义性一样，基本见不到这种需求，所以在函数的输入参数类型上，符合二八定律。于是对于上述代码，假设 x 和 y 绝大多数情况下都是整数，则进行特化（假设这个伪代码中不考虑整数溢出）：

```
func f(x, y):

    if not (x instanceof int and y instanceof int):

        //有一个不是整数，走原有流程
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
    return x + y

//整数加法的特化流程

internal_code:

    int ix = get_internal_int(x)

    int iy = get_internal_int(y)

    int iresult

    asm:

        push ... //当前状态压栈

        mov eax, ix

        mov ebx, iy

        add eax, ebx

        mov iresult, eax

        pop ... //状态出栈

    return build_int_object(iresult)
```

当然这只是个例子，如果只是为了一个加法，这多少有点小题大做，但如果 f 的逻辑较为复杂，优化就很明显了

还可以逆向思维一下，AOT 难以实现特化的原因是无法考虑所有情况，但我们也没有必要考虑所有情况，实际上类型使用的二八定律本身也在另一个二八定律里，具体到 int 类型，一个绝大多数使用到的类型都是 int 的程序在所有程序中占绝大多数，至少在一个有限的领域是这样，因此干脆对于每个函数都只做 int 相关的特化，这样 2k 种情况还算能接受（实际情况数比 2k 低很多，因为很多参数如果被假定为 int，会语法错误，就不用假设了），如果再做的好一点，还可以做成编译器选项，由用户来指定 AOT 的时候对哪个类型特化，这样就比较完美了

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

除类型的动态性外，其他动态性也可以类似讨论，仅拿上篇的例子，不赘述了：

```
for i in range(n):

    print(i)

转换为:  if not (range is builtins.range and print is builtins.print):

    for i in range(n):

        print(i) else:

    internal_code:

        long tmp = get_internal_long(n)

        long i

        //这里应该用汇编，仅表个意思

        for (i = 0; i < tmp; ++ i):

            print_long(i)
```

需要在程序启动时在 builtins 里面保存默认函数，用于检测当前运行环境是否被用户修改过，这样就兼顾了效率和动态性，跟上面一样，这里 JIT 或 AOT 实现都可以。

二、dexopt 与 dex2oat

从应用层开发来说有个原理上的大致理解也是必须掌握的，具体区别可用如下图概述（图片来自网络）。

BAT交流群：892872246

通过上图可以很明显的看出 dexopt 与 dex2oat 的区别，前者针对 Dalvik 虚拟机，后者针对 Art 虚拟机。

dexopt 是对 dex 文件 进行 verification 和 optimization 的操作，其对 dex 文件的优化结果变成了 odex 文件，这个文件和 dex 文件很像，只是使用了一些优化操作码（譬如优化调用虚拟指令等）。

dex2oat 是对 dex 文件的 AOT 提前编译操作，其需要一个 dex 文件，然后对其进行编译，结果是一个本地可执行的 ELF 文件，可以直接被本地处理器执行。

除此之外在上图还可以看到 Dalvik 虚拟机中有使用 JIT 编译器，也就是说其也能将程序运行的热点 java 字节码编译成本地 code 执行，所以其与 Art 虚拟机还是有区别的。Art 虚拟机的 dex2oat 是提前编译所有 dex 字节码，而 Dalvik 虚拟机只编译使用启发式检测中最频繁执行的热点字节码。

第二节、热修复设计之 CLASS_ISPREVERIFIED 问题

一、前言

本文将解决两个问题。

1. 怎么将修复后的 Bug 类打包成 dex
2. 怎么将外部的 dex 插入到 ClassLoader 中

二、建立测试 Demo

BAT交流群：892872246

2.1 目录结构

2.2 源码

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?><RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:paddingBottom="@dimen/activity_vertical_margin"

    android:paddingLeft="@dimen/activity_horizontal_margin"

    android:paddingRight="@dimen/activity_horizontal_margin"

    android:paddingTop="@dimen/activity_vertical_margin"

    tools:context=".MainActivity">
```

```
    <Button

        android:layout_width="wrap_content"

        android:layout_height="wrap_content"

        android:onClick="click"

        android:text="小喵叫一声"/></RelativeLayout>
```

MainActivity.class

```
package com.aitsuki.bugfix;

import android.os.Bundle;import android.support.v7.app.AppCompatActivity;
import android.view.View;import android.widget.Toast;import com.aitsuki.
bugfix.animal.Cat;

public class MainActivity extends AppCompatActivity {



    private Cat mCat;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        mCat = new Cat();

    }



    public void click(View view) {

        Toast.makeText(this, mCat.say(),Toast.LENGTH_SHORT).show();

    }}
```

Cat.class

```
package com.aitsuki.bugfix.animal;

/**

 * Created by AItsuki on 2016/3/14.

 */public class Cat {

    public String say() {

        return "汪汪汪！";

    }}
```

BAT交流群：892872246

2.3 运行结果

假设这是我们公司的开发项目，刚刚上线就发现了严重 bug，猫会狗叫。

想修复 bug，让用户再立刻更新一次显然很不友好，此时热补丁修复技术就有用了。

三、制作补丁

在加载 dex 的代码之前，我们先来制作补丁。

1. 首先我们将 Cat 类修复，汪汪汪改成喵喵喵，然后重新编译项目。（Rebuild

92872246

一下就行了）

2. 去保存项目的地方，将 Cat.class 文件拷贝出来，在这里

3. 新建文件夹，要和该 Cat.class 文件的包名一致，然后将 Cat.class 复制到这里，如图

4. 命令行进入到图中的 test 目录，运行一下命令，打包补丁。如图：

BAT交流群/2246

然后 test 目录是这样的

patch_dex.jar 就是我们打包好的补丁了，我们将它放到 sdCard 中，待会从这里加载补丁。

关于什么用这么复杂的方法打包补丁的说明：

你也可以直接将 java 文件拷出来，通过 javac -d 带包编译再转成 jar。

但我这么麻烦是有原因的，因为用这种方法你可能会遇到 ParseException，原因是 jar 包版本和 dx 工具版本不一致。

而从项目中直接将编译好的 class 直接转成 jar 就没问题，因为 java 会向下兼容，

,12246

打出来的 jar 包和 class 版本是一致的。

总而言之，dx 版本要和 class 编译版本对应。

四、加载补丁

4.1 思路

通过上一篇博文，我们知道 dex 保存在这个位置

BaseDexClassLoader–>pathList–>dexElements

apk 的 classes.dex 可以从应用本身的 DexClassLoader 中获取。

path_dex 的 dex 需要 new 一个 DexClassLoader 加载后再获取。

分别通过反射取出 dex 文件，重新合并成一个数组，然后赋值给盈通本身的 ClassLoader 的 dexElements

4.2 代码实现

加载外部 dex，我们可以在 Application 中操作。

首先新建一个 HotPatchApplication，然后在清单文件中配置，顺便加上读取 sdcard 的权限，因为补丁就保存在那里。

HotPatchApplication 代码如下：

```
package com.aitsuki.hotpatchdemo;

import android.app.Application;import android.os.Environment;import andr
oid.util.Log;import java.io.File;import java.lang.reflect.Array;import j
ava.lang.reflect.Field;import dalvik.system.DexClassLoader;

/**
```

892872246

```
 * Created by hp on 2016/4/6.

 */

public class HotPatchApplication extends Application {



    @Override

    public void onCreate() {

        super.onCreate();



        // 获取补丁，如果存在就执行注入操作

        String dexPath = Environment.getExternalStorageDirectory().getAb
solutePath().concat("/patch_dex.jar");

        File file = new File(dexPath);

        if (file.exists()) {

            inject(dexPath);

        } else {

            Log.e("BugFixApplication", dexPath + "不存在");

        }

    }



    /**

     * 要注入的 dex 的路径

     *

     * @param path
```

```
    */

    private void inject(String path) {

        try {

            // 获取 classes 的 dexElements

            Class<?> cl = Class.forName("dalvik.system.BaseDexClassLoader
");

            Object pathList = getField(cl, "pathList", getClassLoader());

            Object baseElements = getField(pathList.getClass(), "dexEleme
nts", pathList);



            // 获取 patch_dex 的 dexElements（需要先加载 dex）

            String dexopt = getDir("dexopt", 0).getAbsolutePath();

            DexClassLoader dexClassLoader = new DexClassLoader(path, dexo
pt, dexopt, getClassLoader());

            Object obj = getField(cl, "pathList", dexClassLoader);

            Object dexElements = getField(obj.getClass(), "dexElements",
obj);



            // 合并两个 Elements

            Object combineElements = combineArray(dexElements, baseElemen
ts);



            // 将合并后的 Element 数组重新赋值给 app 的 classLoader

            setField(pathList.getClass(), "dexElements", pathList, combin
eElements);
```

```
            //======== 以下是测试是否成功注入 =================

            Object object = getField(pathList.getClass(), "dexElements",
pathList);

            int length = Array.getLength(object);

            Log.e("BugFixApplication", "length = " + length);



        } catch (ClassNotFoundException e) {

            e.printStackTrace();

        } catch (IllegalAccessException e) {

            e.printStackTrace();

        } catch (NoSuchFieldException e) {

            e.printStackTrace();

        }

    }



    /**

     * 通过反射获取对象的属性值

     */

    private Object getField(Class<?> cl, String fieldName, Object object)
throws NoSuchFieldException, IllegalAccessException {

        Field field = cl.getDeclaredField(fieldName);

        field.setAccessible(true);

        return field.get(object);

    }
```

```
    /**
     * 通过反射设置对象的属性值
     */
    private void setField(Class<?> cl, String fieldName, Object object, O
bject value) throws NoSuchFieldException, IllegalAccessException {
        Field field = cl.getDeclaredField(fieldName);
        field.setAccessible(true);
        field.set(object, value);
    }

    /**
     * 通过反射合并两个数组
     */
    private Object combineArray(Object firstArr, Object secondArr) {
        int firstLength = Array.getLength(firstArr);
        int secondLength = Array.getLength(secondArr);
        int length = firstLength + secondLength;

        Class<?> componentType = firstArr.getClass().getComponentType();
        Object newArr = Array.newInstance(componentType, length);
        for (int i = 0; i < length; i++) {
            if (i < firstLength) {
```

```
                Array.set(newArr, i, Array.get(firstArr, i));

            } else {

                Array.set(newArr, i, Array.get(secondArr, i - firstLengt
h));

            }

        }

        return newArr;

    }

}
```

五、CLASS_ISPREVERIFIED

运行一下 Demo，报以下错误。（AndroidStudio 2.0 可能不会报错，需要打包的时候才会出现错误，这是 Instant run 导致的）

```
E/BugFixApplication: length = 2
W/dalvikvm: VFY: unable to find class referenced in signature (Landroid/view/SearchEvent;)
W/dalvikvm: VFY: unable to resolve interface method 14480: Landroid/view/Window$Callback;.onSearchRequested (Landroid/view/SearchEvent;)Z
W/dalvikvm: VFY: unable to resolve interface method 14484: Landroid/view/Window$Callback;.onWindowStartingActionMode (Landroid/view/ActionMode$Callba
W/dalvikvm: VFY: unable to resolve virtual method 408: Landroid/content/res/TypedArray;.getChangingConfigurations ()I
W/dalvikvm: VFY: unable to resolve virtual method 430: Landroid/content/res/TypedArray;.getType (I)I
W/dalvikvm: Class resolved by unexpected DEX: Lcom/aitsuki/bugfix/MainActivity;(0x424e2f30):0x752ab000 ref [Lcom/aitsuki/bugfix/animal/Cat;] Lcom/ait
W/dalvikvm: (Lcom/aitsuki/bugfix/MainActivity; had used a different Lcom/aitsuki/bugfix/animal/Cat; during pre-verification)
W/dalvikvm: threadid=1: thread exiting with uncaught exception (group=0x41804e00)
E/AndroidRuntime: FATAL EXCEPTION: main
E/AndroidRuntime: Process: com.aitsuki.bugfix, PID: 5557
E/AndroidRuntime: java.lang.IllegalAccessError: Class ref in pre-verified class resolved to unexpected implementation
E/AndroidRuntime:     at com.aitsuki.bugfix.MainActivity.onCreate(MainActivity.java:16)   new Cat()
E/AndroidRuntime:     at android.app.Activity.performCreate(Activity.java:5266)
E/AndroidRuntime:     at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1091)
E/AndroidRuntime:     at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2189)
E/AndroidRuntime:     at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2279)
```

dexElements 的 length = 2，看来我们的 patch_dex 已经成功添加进去了。

但是从黄色框框和黄色框上面那一段 log 提示中可以看出，MainActivity 引用了 Cat，但是发现他们在不同的 Dex 中。

看到这里可能就会问：
为什么之前那么多项目都采用分包方案，但是却不会出现这个错误呢？
我在这里总结了一个过程，想知道详细分析过程的请看 QQ 空间开发团队的原文。

在 apk 安装的时候，虚拟机会将 dex 优化成 odex 后才拿去执行。在这个过程中会对所有 class 一个校验。

校验方式：假设 A 该类在它的 static 方法，private 方法，构造函数，override 方法中直接引用到 B 类。如果 A 类和 B 类在同一个 dex 中，那么 A 类就会被打上 CLASS_ISPREVERIFIED 标记

被打上这个标记的类不能引用其他 dex 中的类，否则就会报图中的错误

在我们的 Demo 中，MainActivity 和 Cat 本身是在同一个 dex 中的，所以 MainActivity 被打上了 CLASS_ISPREVERIFIED。而我们修复 bug 的时候却引用了另外一个 dex 的 Cat.class，所以这里就报错了
而普通分包方案则不会出现这个错误，因为引用和被引用的两个类一开始就不在同一个 dex 中，所以校验的时候并不会被打上 CLASS_ISPREVERIFIED

补充一下第二条：A 类如果还引用了一个 C 类，而 C 类在其他 dex 中，那么 A 类并不会被打上标记。换句话说，只要在 static 方法，构造方法，private 方法，override 方法中直接引用了其他 dex 中的类，那么这个类就不会被打上 CLASS_ISPREVERIFIED 标记。

5.1 解决方案

根据上面的第六条，我们只要让所有类都引用其他 dex 中的某个类就可以了。

下面是 QQ 控件给出的解决方案

在所有类的构造函数中插入这行代码 System.out.println(AntilazyLoad.class);

这样当安装 apk 的时候，classes.dex 内的类都会引用一个在不相同 dex 中的 AntilazyLoad 类，这样就防止了类被打上 CLASS_ISPREVERIFIED 的标志了，只要没被打上这个标志的类都可以进行打补丁操作。

hack.dex 在应用启动的时候就要先加载出来，不然 AntilazyLoad 类会被标记为不存在，即使后面再加载 hack.dex，AntilazyLoad 类还是会提示不存在。该类只要一次找不到，那么就会永远被标上找不到的标记了。

我们一般在 Application 中执行 dex 的注入操作，所以在 Application 的构造中不能加上 System.out.println(AntilazyLoad.class);这行代码，因为此时 hack.dex 还没有加载进来，AntilazyLoad 并不存在。

之所以选择构造函数是因为他不增加方法数，一个类即使没有显式的构造函数，也会有一个隐式的默认构造函数。

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

5.2 插入代码的难点

1.首先在源码中手动插入不太可行，hack.dex 此时并没有加载进来，AntilazyLoad.class 并不存在，编译不通过。

2.所以我们需要在源码编译成字节码之后，在字节码中进行插入操作。对字节码进行操作的框架有很多，但是比较常用的则是 ASM 和 javaassist

3.但 AndroidStudio 是使用 Gradle 构建项目，编译-打包都是自动化的，我们怎么操作呢。

第三节、热修复设计之热修复原理

一、Android 热修复

在 Android 的热修复中主要用来替换类，资源，so 的过程；

Java 虚拟机

栈架构指令集的主要缺点是执行速度相对来说稍微慢一些;基于堆栈的机器需要更多指令，（内存）

Android 虚拟机

而基于寄存器（硬件在 CPU 内部）的机器指令更长

速度： CPU - > 寄存器 -> 内存 -> 外存

http://blog.csdn.net/ljtyzhr/article/details/39859659

Android 目前有 2 中虚拟机， Dalvik 和 ART 虚拟机；

72246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

二、Android 虚拟机和编译加载顺序

Android 热修复其实主要是针对 Android 虚拟机加载类的一个过程，所以首先先我们应该知道 Android 常用的虚拟机是 Dalvik 虚拟和 ART 虚拟机；

Android 4.0 之前是主要是的 Dalvik 虚拟机。 Android 4.4 之后开始支持 ART 虚拟机（可选）， Android 5.0 之后就是 ART 虚拟机；

```
Android 4.0 --> Android 4.4 --> Android 5.0 ---> Android 7.0
```

Dalvik 虚拟机在 Android 2.2 的时候引入了 JIT (Just in time), 也就是一边运行，一边编译成机器码在运行；这种编译成机器码的过程在应用重启的时候需要重新编译成机器码再运行，有点浪费性能（不是持久化），重复翻译，编译，运行；

在 Android 5.0 之后就使用完全使用 ART 虚拟机；因为在 AOT (ahead of time) 的操作在安装的时候把 dex 优化成 odex; 在 Android N (7) 之前是全量 dex 优化成 odex, 这样导致安装 apk 的时候，或者系统更新重启的时候非常耗时，特别的慢；

所以在 Android N 后引入了 JIT 和 AOT 的混合模式; 可以理解为"全时段的编译"（All-Of-the-Time compilation， 也叫 AOT；是不是发现有 2 个 AOT, 一开始我也很懵逼的；其实和 Android 5.0 中的 AOT 是不太一样的；还有要注意的是 Android N 之后的 JIT 和 Davlik 虚拟机中的 JIT 是不一样的，简单理解就是高级版的 JIT；这个高级版本的 JIT 过程会把处理后的 odex 缓存到 base.art（有些地方也叫 image ） 中；等下次 app 启动的时候，就先直接把这个优化后的 base.art 加载到内存中；这样就不会重复的 JIT 了；

/2246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

三、混合模式的理解

ART 初期是使用全量的 AOT (aheader of time) 变成机器码（指令）；因为耗安装时间和系统升级后的启动时间；所以在 ART 的时候，先把 dex 解释成中间态的，不编译成机器码。在运行的时候，或者充电的时候，只编译"热代码"

引用 infoq 的一篇文章 http://www.infoq.com/cn/news/2016/04/android-n-aot-jit

> Android N（7.x）开发者预览版包含了一个混合模式的运行时。应用在安装时不做编译，而是解释字节码，所以可以快速启动。ART 中有一种新的、更快的解释器，通过一种新的 JIT 完成，但是这种 JIT 的信息不是持久化的。取而代之的是，代码在执行期间被分析，分析结果保存起来。然后，当设备空转和充电的时候，ART 会执行针对"热代码"进行的基于分析的编译，其他代码不做编译。为了得到更优的代码，ART 采用了几种技巧包括深度内联。
>
> 对同一个应用可以编译数次，或者找到变"热"的代码路径或者对已经编译的代码进行新的优化，这取决于分析器在随后的执行中的分析数据。这个步骤仍被简称为 AOT，可以理解为"全时段的编译"（All-Of-the-Time compilation）。

四、源码类到机器执行的文件过程

主要过程： java - (dex, class) - opt/oat -- odex;

892872246
BAT交

Davlik 虚拟机: java - dex - opt -- D 类型的 odex （优化过后的还是需要翻译）; JIT

ART 虚拟机: java -- dex -- oat -- A 类型的 odex （机器码类型文件）; AOT (Ahead of time)

混合模式： java -- dex -- oat -- (D 类型的 odex, base.art: 热缓存，image) JIT （高级的 JIT）; AOT （All-Of-the-Time compilation）可以理解为"全时段的编译" ,profile

注意以上的 A 类型 odex 和 D 类型的 odex，只是用来代码 2 种是不同的 odex，便于理解；

五、补丁包

补丁包主要有 class， 资源文件，so 的改动；

1. 资源文件方案；替换 AssertManager， ---> 加载 resources.arsc
2. os 还不清楚，没研究过, 可能使用 classLoader 加载放入 so 目录
3. 类 ：底层替换和类加载器方案

本文主要诉说类加载器的方案；

六、类补丁生效原理

1. 底层替换方案 （阿里系）AndFix，
2. 类加器载方案 （腾讯系）QFix, tinker(粒度太细了), Sophix 使用类的替换（粒度大一点）

BAT交流群：892872246

因为 Android 有 2 款虚拟机，所以应该针对 2 款虚拟机加载过程进行分析和处理；

七、Davlik 虚拟机的限制

Android 在编译的时候有个 65536 （2 的 16 次方）问题。受限于 Dlv 虚拟机的限制；不只是方法名有限制，其实字段也是有限制的；

八、Davlik Class resolved by unexpected DEX: 限制和处理方式

Davlik 类加载处理 源码

手 Q 的方案

```
if (!fromUnverifiedConstant && // 条件 3
        IS_CLASS_FLAG_SET(referrer, CLASS_ISPREVERIFIED)) // 条件 1
    {
        ClassObject* resClassCheck = resClass;
        if (dvmIsArrayClass(resClassCheck))
            resClassCheck = resClassCheck->elementClass;
        if (referrer->pDvmDex != resClassCheck->pDvmDex &&
            resClassCheck->classLoader != NULL) // 条件 2
        {
            ALOGW("Class resolved by unexpected DEX:"
                 " %s(%p):%p ref [%s] %s(%p):%p",
                referrer->descriptor, referrer->classLoader,
                referrer->pDvmDex,
```

```
                resClass->descriptor, resClassCheck->descriptor,
                resClassCheck->classLoader, resClassCheck->pDvmDex);
            ALOGW("(%s had used a different %s during pre-verification)
",
                referrer->descriptor, resClass->descriptor);
            dvmThrowIllegalAccessError(
                "Class ref in pre-verified class resolved to unexpecte
d "
                "implementation");
            return NULL;
        }
    }
```

三个条件：

CLASS_ISPREVERIFIED 这个标记被打上的原理是，只要一个类里面和这个类的依赖都在一个 dex 中，那么这些类就被打上 PREVERIFIED 的标记；

单独有个[特殊的.dex]这个 dex 里面只有一个类，且其他的 dex 中的类都引用一下这个单独 dex 里面的类，那么，项目中所有的类，就不会被打上 CLASS_ISPREVERIFIED 这个标记

```
dex 数组:  [ 1.dex, 2.dex, 3.dex, 特殊的.dex(插桩类)]
```

referrer->pDvmDex != resClassCheck->pDvmDex

BAT交流群：892872246

补丁类和引用类应该在同一个 dex 里面；

fromUnverifiedConstant hook 使用 native 方式

九、类加载器的双亲委派加载机制

类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父类去完成，每一个层次类加载器都是如此，因此所有的加载请求都应该传送到启动类加载其中，只有当父类加载器反馈自己无法完成这个请求的时候（在它的加载路径下没有找到所需加载的 Class），子类加载器才会尝试自己去加载

相关日志

```
12-07 10:05:27.859 11471-11471/app.mj.com.ihour I/MyApp: getClassLoader() : dalvik.system.PathClassLoader[DexPathList[[zip file "/data/app/app.mj.com.ihour-2/base.apk"],nativeLibraryDirectories=[/data/app/app.mj.com.ihour-2/lib/arm64, /system/lib64, /vendor/lib64]]]12-07 10:05:27.860 11471-11471/app.mj.com.ihour I/MyApp: getClassLoader().getParent(): java.lang.BootClassLoader@36e062412-07 10:05:27.860 11471-11471/app.mj.com.ihour I/MyApp: getClassLoader()
```

第四节、Tinker 的集成与使用（自动补丁包生成）

一、简述

> Tinker 是微信官方的 Android 热补丁解决方案，它支持动态下发代码、So 库以及资源，让应用能够在不需要重新安装的情况下实现更新。当然，你也可以使用 Tinker 来更新你的插件。

上面是 Tinker 官方 Wiki 的原话，意思嘛相信大家都看得明白，但注意啦，它并没有说 Tinker 可以让补丁实时生效（也叫无感知更新），它必须在打上补丁后重启 App（重启进程），补丁才会发挥作用，这跟阿里的热修复方案有着本质的

BAT交流群：892872246

区别。在开始集成 Tinker 之前，我们有必要了解清楚，Tinker 有那些不足，下面是 Tinker 的已知问题：

1.Tinker 不支持修改 AndroidManifest.xml，Tinker 不支持新增四大组件(1.9.0 支持新增非 export 的 Activity)；

2.由于 Google Play 的开发者条款限制，不建议在 GP 渠道动态更新代码；

3.在 Android N 上，补丁对应用启动时间有轻微的影响；

4.不支持部分三星 android-21 机型，加载补丁时会主动抛出 "TinkerRuntimeException:checkDexInstall failed"；

5.对于资源替换，不支持修改 remoteView。例如 transition 动画，notification icon 以及桌面图标。

> 上述不足是由于原理与系统限制，我们在编程中要清楚这些，尽量避免以上问题的出现。

尽管 Tinker 有着这些“小缺点”，但也丝毫不影响 Tinker 在国内众多热修复方案中的地位，一方面 Tinker 是开源的（这意味着 Tinker 本身免费），另一方面则是 Tinker 已运行在微信的数亿 Android 设备上（说明该方案相当稳定）。下面开始进行对 Tinker 的集成与使用。

二、Tinker 组件依赖

1、在项目的 build.gradle 中：

添加 tinker-patch-gradle-plugin 的依赖

BAT交流群：892872246

```
buildscript {
    dependencies {
        classpath ('com.tencent.tinker:tinker-patch-gradle-plugin:1.9.1')
    }
}
```
复制代码

2、在 app 的 gradle 文件（app/build.gradle）中：

> 需要注意一点，Tinker 需要使用到 MulitDex，原话在 Bugly 文档的热更新 API 接口部分。

1）添加 tinker 的库依赖

Gradle 版本小于 2.3 的这么写：

```
dependencies {
    compile "com.android.support:multidex:1.0.1"
    //可选，用于生成 application 类
    provided('com.tencent.tinker:tinker-android-anno:1.9.1')
    //tinker 的核心库
    compile('com.tencent.tinker:tinker-android-lib:1.9.1')
}
```
复制代码

Gradle 版本大等于 2.3 的这么写：

BAT交流 2246

```
dependencies {
    implementation "com.android.support:multidex:1.0.1"
    //tinker 的核心库
    implementation("com.tencent.tinker:tinker-android-lib:1.9.1") { changing = true }
    //可选，用于生成 application 类
    annotationProcessor("com.tencent.tinker:tinker-android-anno:1.9.1") { changing =
true }
    compileOnly("com.tencent.tinker:tinker-android-anno:1.9.1") { changing = true }
}
```
复制代码

2）开启 multiDex

```
defaultConfig {
        ...
        multiDexEnabled true
}
```
复制代码

3）应用 tinker 的 gradle 插件

这部分可先不管，在第三部分《Tinker 的配置及任务》的第 2 节《配置 Tinker 与任务》中会添加。可跳过这部分继续往下看。

BAT交流群：892872246

```
//apply tinker 插件

apply plugin: 'com.tencent.tinker.patch
```

复制代码

三、Tinker 的配置及任务

1、开启支持大工程模式

Tinker 文档中推荐将 jumboMode 设置为 true。

```
android {

    dexOptions {

        // 支持大工程模式

        jumboMode = true

    }

    ...

}
```

复制代码

2、配置 Tinker 与任务

将下面的配置全部复制粘贴到 app 的 gradle 文件（app/build.gradle）末尾，内容很多，但现在只需要看懂 bakPath 与 ext 括号内的东东就好了。

892872246

```
// Tinker 配置与任务

def bakPath = file("${buildDir}/bakApk/")
```

```
ext {
    // 是否使用 Tinker(当你的项目处于开发调试阶段时，可以改为 false)
    tinkerEnabled = true
    // 基础包文件路径(名字这里写死为 old-app.apk。用于比较新旧 app 以生成补丁包，不管是 debug 还是 release 编译)
    tinkerOldApkPath = "${bakPath}/old-app.apk"
    // 基础包的 mapping.txt 文件路径（用于辅助混淆补丁包的生成，一般在生成 release 版 app 时会使用到混淆，所以这个 mapping.txt 文件一般只是用于 release 安装包补丁的生成）
    tinkerApplyMappingPath = "${bakPath}/old-app-mapping.txt"
    // 基础包的 R.txt 文件路径（如果你的安装包中资源文件有改动，则需要使用该 R.txt 文件来辅助生成补丁包）
    tinkerApplyResourcePath = "${bakPath}/old-app-R.txt"
    //only use for build all flavor, if not, just ignore this field
    tinkerBuildFlavorDirectory = "${bakPath}/flavor"
}

def getOldApkPath() {
    return hasProperty("OLD_APK") ? OLD_APK : ext.tinkerOldApkPath
}

def getApplyMappingPath() {
    return hasProperty("APPLY_MAPPING") ? APPLY_MAPPING :
ext.tinkerApplyMappingPath
}
```

```
def getApplyResourceMappingPath() {
    return hasProperty("APPLY_RESOURCE") ? APPLY_RESOURCE :
ext.tinkerApplyResourcePath
}

def getTinkerIdValue() {
    return hasProperty("TINKER_ID") ? TINKER_ID : android.defaultConfig.versionName
}

def buildWithTinker() {
    return hasProperty("TINKER_ENABLE") ? TINKER_ENABLE : ext.tinkerEnabled
}

def getTinkerBuildFlavorDirectory() {
    return ext.tinkerBuildFlavorDirectory
}

if (buildWithTinker()) {
    //apply tinker 插件
    apply plugin: 'com.tencent.tinker.patch'

    // 全局信息相关的配置项
    tinkerPatch {
        tinkerEnable = buildWithTinker()// 是否打开 tinker 的功能。
```

oldApk = getOldApkPath()　　// 基准 apk 包的路径，必须输入，否则会报错。

ignoreWarning = false　　// 是否忽略有风险的补丁包。这里选择不忽略，当补丁包风险时会中断编译。

useSign = true　　// 在运行过程中，我们需要验证基准 apk 包与补丁包的签名是否一致，我们是否需要为你签名。

// 编译相关的配置项

buildConfig {

applyMapping = getApplyMappingPath()

// 可选参数；在编译新的 apk 时候，我们希望通过保持旧 apk 的 proguard 混淆方式，从而减少补丁包的大小。这个只是推荐设置，不设置 applyMapping 也不会影响任何的 assemble 编译。

applyResourceMapping = getApplyResourceMappingPath()

// 可选参数；在编译新的 apk 时候，我们希望通过旧 apk 的 R.txt 文件保持 ResId 的分配，这样不仅可以减少补丁包的大小，同时也避免由于 ResId 改变导致 remote view 异常。

tinkerId = getTinkerIdValue()

// 在运行过程中，我们需要验证基准 apk 包的 tinkerId 是否等于补丁包的 tinkerId。这个是决定补丁包能运行在哪些基准包上面，一般来说我们可以使用 git 版本号、versionName 等等。

keepDexApply = false

// 如果我们有多个 dex,编译补丁时可能会由于类的移动导致变更增多。若打开 keepDexApply 模式，补丁包将根据基准包的类分布来编译。

isProtectedApp = false // 是否使用加固模式，仅仅将变更的类合成补丁。注意，这种模式仅仅可以用于加固应用中。

supportHotplugComponent = false // 是否支持新增非 export 的 Activity（1.9.0 版本开始才有的新功能）

}

```
        // dex 相关的配置项
        dex {
            dexMode = "jar"
// 只能是'raw'或者'jar'。 对于'raw'模式，我们将会保持输入 dex 的格式。对于'jar'模式，我们将会把输入 dex 重新压缩封装到 jar。如果你的 minSdkVersion 小于 14，你必须选择'jar'模式，而且它更省存储空间，但是验证 md5 时比'raw'模式耗时。默认我们并不会去校验 md5,一般情况下选择 jar 模式即可。
            pattern = ["classes*.dex",
                       "assets/secondary-dex-?.jar"]
            // 需要处理 dex 路径，支持*、?通配符，必须使用'/'分割。路径是相对安装包的，例如 assets/...
            loader = [
                    // 定义哪些类在加载补丁包的时候会用到。这些类是通过 Tinker 无法修改的类，也是一定要放在 main dex 的类。
                    // 如果你自定义了 TinkerLoader，需要将它以及它引用的所有类也加入 loader 中；
                    // 其他一些你不希望被更改的类，例如 Sample 中的 BaseBuildInfo 类。这里需要注意的是，这些类的直接引用类也需要加入到 loader 中。或者你需要将这个类变成非 preverify。
            ]
        }
        //  lib 相关的配置项
        lib {
            pattern = ["lib/*/*.so","src/main/jniLibs/*/*.so"]
            // 需要处理 lib 路径，支持*、?通配符，必须使用'/'分割。与 dex.pattern 一致，路径是相对安装包的，例如 assets/...
```

```
    }
    // res 相关的配置项
    res {
        pattern = ["res/*", "assets/*", "resources.arsc", "AndroidManifest.xml"]
        // 需要处理 res 路径，支持*、?通配符，必须使用'/'分割。与 dex.pattern 一致，路径是相对安装包的，例如 assets/...，务必注意的是，只有满足 pattern 的资源才会放到合成后的资源包。
        ignoreChange = [
            // 支持*、?通配符，必须使用'/'分割。若满足 ignoreChange 的 pattern，在编译时会忽略该文件的新增、删除与修改。 最极端的情况，ignoreChange 与上面的 pattern 一致，即会完全忽略所有资源的修改。
            "assets/sample_meta.txt"
        ]
        largeModSize = 100
        // 对于修改的资源，如果大于 largeModSize，我们将使用 bsdiff 算法。这可以降低补丁包的大小，但是会增加合成时的复杂度。默认大小为 100kb
    }
    // 用于生成补丁包中的'package_meta.txt'文件
    packageConfig {
        // configField("key", "value"), 默认我们自动从基准安装包与新安装包的 Manifest 中读取 tinkerId,并自动写入 configField。
        // 在这里，你可以定义其他的信息，在运行时可以通过 TinkerLoadResult.getPackageConfigByName 得到相应的数值。
        // 但是建议直接通过修改代码来实现，例如 BuildConfig。
```

```
            configField("platform", "all")
            configField("patchVersion", "1.0")
//              configField("patchMessage", "tinker is sample to use")
        }
        // 7zip 路径配置项，执行前提是 useSign 为 true
        sevenZip {
            zipArtifact = "com.tencent.mm:SevenZip:1.1.10"
        }
    }
    List<String> flavors = new ArrayList<>();
    project.android.productFlavors.each { flavor ->
        flavors.add(flavor.name)
    }
    boolean hasFlavors = flavors.size() > 0
    def date = new Date().format("MMdd-HH-mm-ss")

    /**
     * bak apk and mapping
     */
    android.applicationVariants.all { variant ->
        /**
         * task type, you want to bak
         */
```

```
        def taskName = variant.name

        tasks.all {
            if ("assemble${taskName.capitalize()}".equalsIgnoreCase(it.name)) {

                it.doLast {
                    copy {
                        def fileNamePrefix = "${project.name}-${variant.baseName}"
                        def newFileNamePrefix = hasFlavors ? "${fileNamePrefix}" :
"${fileNamePrefix}-${date}"

                        def destPath = hasFlavors ?
file("${bakPath}/${project.name}-${date}/${variant.flavorName}") : bakPath
                        from variant.outputs.first().outputFile
                        into destPath
                        rename { String fileName ->
                            fileName.replace("${fileNamePrefix}.apk",
"${newFileNamePrefix}.apk")
                        }

                        from "${buildDir}/outputs/mapping/${variant.dirName}/mapping.txt"
                        into destPath
                        rename { String fileName ->
                            fileName.replace("mapping.txt",
```

```
"${newFileNamePrefix}-mapping.txt")
                        }

                        from "${buildDir}/intermediates/symbols/${variant.dirName}/R.txt"
                        into destPath
                        rename { String fileName ->
                            fileName.replace("R.txt", "${newFileNamePrefix}-R.txt")
                        }
                    }
                }
            }
        }
    }
    project.afterEvaluate {
        //sample use for build all flavor for one time
        if (hasFlavors) {
            task(tinkerPatchAllFlavorRelease) {
                group = 'tinker'
                def originOldPath = getTinkerBuildFlavorDirectory()
                for (String flavor : flavors) {
                    def tinkerTask =
tasks.getByName("tinkerPatch${flavor.capitalize()}Release")
                    dependsOn tinkerTask
```

```
                def preAssembleTask =
tasks.getByName("process${flavor.capitalize()}ReleaseManifest")
                preAssembleTask.doFirst {
                    String flavorName = preAssembleTask.name.substring(7,
8).toLowerCase() + preAssembleTask.name.substring(8, preAssembleTask.name.length() - 15)
                    project.tinkerPatch.oldApk =
"${originOldPath}/${flavorName}/${project.name}-${flavorName}-release.apk"
                    project.tinkerPatch.buildConfig.applyMapping =
"${originOldPath}/${flavorName}/${project.name}-${flavorName}-release-mapping.txt"
                    project.tinkerPatch.buildConfig.applyResourceMapping =
"${originOldPath}/${flavorName}/${project.name}-${flavorName}-release-R.txt"

                }

            }
        }

        task(tinkerPatchAllFlavorDebug) {
            group = 'tinker'
            def originOldPath = getTinkerBuildFlavorDirectory()
            for (String flavor : flavors) {
                def tinkerTask = tasks.getByName("tinkerPatch${flavor.capitalize()}Debug")
                dependsOn tinkerTask
                def preAssembleTask =
```

```
tasks.getByName("process${flavor.capitalize()}DebugManifest")

                    preAssembleTask.doFirst {

                        String flavorName = preAssembleTask.name.substring(7,

8).toLowerCase() + preAssembleTask.name.substring(8, preAssembleTask.name.length() - 13)

                        project.tinkerPatch.oldApk =

"${originOldPath}/${flavorName}/${project.name}-${flavorName}-debug.apk"

                        project.tinkerPatch.buildConfig.applyMapping =

"${originOldPath}/${flavorName}/${project.name}-${flavorName}-debug-mapping.txt"

                        project.tinkerPatch.buildConfig.applyResourceMapping =

"${originOldPath}/${flavorName}/${project.name}-${flavorName}-debug-R.txt"

                    }


                }

            }

        }

    }

}
```

复制代码

其中，有几点配置在这里说明一下，方便理解后续的操作（当 tinkerEnabled = true 的情况下）：

1.app 的生成目录是：主 Module（一般是名为 app）/build/bakApk 文件夹。

BAT交流群：892872246

2.补丁包的生成路径：主 Module（一般是名为 app）/build/outputs/apk/tinkerPatch/debug/patch_signed_7zip.apk。

3.基础包的名字：old-app.apk，放于 bakApk 文件夹下。

4.基础包的 mapping.txt 和 R.txt 文件一般在编译 release 签名的 apk 时才会用到。

5.在用到 mapping.txt 文件时，需要重命名为 old-app-mapping.txt，放于 bakApk 文件夹下。

6.在用到 R.txt 文件时，需要重命名为 old-app-R.txt，放于 bakApk 文件夹下。

> 对于 mapping.txt 和 R.txt 文件，在配置中有说明，请回配置中仔细看。 上面只是我项目中的配置，这些其实都是可以自定义的，建议在搞清楚配置内容之后再去自定义修改。

3、什么是基础包？？

基础包就是已经上架的 apk 文件（假设是 1.0 版本）。这其实很好理解，在新版本的 App 上架之前（假设是 2.0 版本），我们会用到 Tinker 来修复 1.0 版 App 中存在的 bug，这时就需要用到 Tinker 来产生补丁包文件，而补丁包文件的本质，就是修复好 Bug 的 App 与 1.0 版本 App 之间的文件差异。在 2.0 版本上架之前，我们可能会多次产生新的补丁包，用于修复在用户手机上的 1.0 版 App，所以补丁包必须以 1.0 版 App 作为参考标准，也就是说用户手机上的 app 就是基础包，即当前应用市场上的 apk 文件（前面说的 1.0 版本）。

四、Tinker 封装与拓展

BAT交流群：892872246

1、拷贝文件

将 Demo 中提供的 tinker 包下的所有文件及文件夹都拷贝到自己项目中。

这些文件其实就是 Tinker 官方 Demo 中的文件完全复制过来的，只是多加了一些注释。

简单说明下，这几个文件的作用：

1.SampleUncaughtExceptionHandler：Tinker 的全局异常捕获器。

2.MyLogImp：Tinker 的日志输出实现类。

3.SampleLoadReporter：加载补丁时的一些回调。

4.SamplePatchListener：过滤 Tinker 收到的补丁包的修复、升级请求。

5.SamplePatchReporter：修复或者升级补丁时的一些回调。

6.SampleTinkerReport：修复结果（成功、冲突、失败等）。

BAT交流群：892872246

7.SampleResultService：:patch 补丁合成进程将合成结果返回给主进程的类。

8.TinkerManager：Tinker 管理器（安装、初始化 Tinker）。

9.TinkerUtils：拓展补丁条件判定、锁屏或后台时应用重启功能的工具类。

> 这些只是对 Tinker 功能的拓展和封装罢了，都是可选的，但这些文件对项目的功能完善会有所帮助，建议加入到自己的项目中。如果你仅仅只是为了修复 bug，而不做过多的工作（如：上传打补丁信息到服务器等），则无须理会这些文件的作用，当然你也可以自己封装。

对于这些自定义类及错误码的详细说明，请参考：「Tinker 官方 Wiki：可选的自定义类」。

2、清单文件中添加服务

前面添加的文件中，有一个 SampleResultService 文件，是四大组件之一，所以必须在清单文件中声明。

```
<service

    android:name="com.lqr.tinker.service.SampleResultService"

    android:exported="false"/>
```

复制代码

五、编写 Application 的代理类

Tinker 表示，Application 无法动态修复，所以有两种选择：

1.使用「继承 TinkerApplication + DefaultApplicationLike」。

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

2.使用「DefaultLifeCycle 注解 + DefaultApplicationLike」。

当然，如果你觉得你自定义的 Application 不会用到热修复，可无视这部分； 但下方代码中的 initTinker()方法记得要拷贝到你项目中，用于初始化 Tinker。

第 1 种方式感觉比较鸡肋，这里使用第 2 种(Tinker 官方推荐的方式):「DefaultLifeCycle 注解 + TinkerApplicationLike」，DefaultLifeCycle 注解生成 Application，下面就来编写 Application 的代理类：

1、编写 TinkerApplicationLike

将下方的代码拷贝到项目中，注释简单明了，不多解释：

```
@SuppressWarnings("unused")
@DefaultLifeCycle(application = "com.lqr.tinker.MyApplication",// application 类名。只能用字符串，这个 MyApplication 文件是不存在的，但可以在 AndroidManifest.xml 的 application 标签上使用（name）
        flags = ShareConstants.TINKER_ENABLE_ALL,// tinkerFlags
        loaderClass = "com.tencent.tinker.loader.TinkerLoader",//loaderClassName，我们这里使用默认即可!（可不写）
        loadVerifyFlag = false)//tinkerLoadVerifyFlag
public class TinkerApplicationLike extends DefaultApplicationLike {

    private Application mApplication;
    private Context mContext;
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
    private Tinker mTinker;

    // 固定写法
    public TinkerApplicationLike(Application application, int tinkerFlags, boolean
tinkerLoadVerifyFlag, long applicationStartElapsedTime, long applicationStartMillisTime,
Intent tinkerResultIntent) {
        super(application, tinkerFlags, tinkerLoadVerifyFlag, applicationStartElapsedTime,
applicationStartMillisTime, tinkerResultIntent);
    }

    // 固定写法
    @TargetApi(Build.VERSION_CODES.ICE_CREAM_SANDWICH)
    public void registerActivityLifecycleCallbacks(Application.ActivityLifecycleCallbacks
callback) {
        getApplication().registerActivityLifecycleCallbacks(callback);
    }

    @TargetApi(Build.VERSION_CODES.ICE_CREAM_SANDWICH)
    @Override
    public void onBaseContextAttached(Context base) {
        super.onBaseContextAttached(base);
        mApplication = getApplication();
        mContext = getApplication();
        initTinker(base);
```

2246

```
        // 可以将之前自定义的 Application 中 onCreate()方法所执行的操作搬到这里...
    }

    private void initTinker(Context base) {
        // tinker 需要你开启 MultiDex
        MultiDex.install(base);

        TinkerManager.setTinkerApplicationLike(this);
        // 设置全局异常捕获
        TinkerManager.initFastCrashProtect();
        //开启升级重试功能（在安装 Tinker 之前设置）
        TinkerManager.setUpgradeRetryEnable(true);
        //设置 Tinker 日志输出类
        TinkerInstaller.setLogIml(new MyLogImp());
        //安装 Tinker(在加载完 multiDex 之后，否则你需要将 com.tencent.tinker.**手动放到 main
dex 中)
        TinkerManager.installTinker(this);
        mTinker = Tinker.with(getApplication());
    }

}
```
复制代码

BAT交流群 892872246

2、搬运自定义 Application 中的操作

把项目中在自定义 Application 的操作移到 TinkerApplicationLike 的 onCreate() 或 onBaseContextAttached()方法中。

```
public class TinkerApplicationLike extends DefaultApplicationLike {

        ...

    @Override

    public void onCreate() {

        super.onCreate();

                    // 将之前自定义的 Application 中 onCreate()方法所执行的操作搬到这里...

    }



    @TargetApi(Build.VERSION_CODES.ICE_CREAM_SANDWICH)

    @Override

    public void onBaseContextAttached(Context base) {

        super.onBaseContextAttached(base);

        mApplication = getApplication();
```

```
        mContext = getApplication();

        initTinker(base);

        // 或搬到这里...

    }

}
```

复制代码

3、清单文件中注册

将@DefaultLifeCycle 中 application 对应的值，即

"com.lqr.tinker.MyApplication"，赋值给清单文件的 application 标签的 name 属性，如下：

```
<application

    android:name="com.lqr.tinker.MyApplication"

    android:allowBackup="true"

    android:icon="@mipmap/ic_launcher"

    android:label="@string/app_name"

    android:roundIcon="@mipmap/ic_launcher_round"

    android:supportsRtl="true"
```

```
    android:theme="@style/AppTheme">

    ...

</application>
```

复制代码

> 注意： 此时 name 属性会报红，因为项目源码中根本不存在 MyApplication.java 文件，但不必担心，因为它是动态生成的，Build 一下项目就好了，不管它也无所谓。

对于 Application 代理类的详细说明，请参考：「Tinker 官方 Wiki：Application 代理类」。

> 到这里就已经集成好 Tinker 了，但只是本地集成而已，服务端下发补丁包到 app 的文章之后会陆续发布更新。

六、常用 API

现在来了解下代码中会用到的几个 Tinker 的重要 API。

1、请求打补丁

```
TinkerInstaller.onReceiveUpgradePatch(context, 补丁包的本地路径);
```

复制代码

BAT交流群：892872246

2、卸载补丁

```
Tinker.with(getApplicationContext()).cleanPatch();// 卸载所有的补丁

Tinker.with(getApplicationContext()).cleanPatchByVersion(版本号)// 卸载指定版本的补丁
```

复制代码

3、杀死应用的其他进程

```
ShareTinkerInternals.killAllOtherProcess(getApplicationContext());
```

复制代码

4、Hack 方式修复 so

```
TinkerLoadLibrary.installNavitveLibraryABI(this, abi);
```

复制代码

abi：cpu 架构类型

5、非 Hack 方式修复 so

```
TinkerLoadLibrary.loadLibraryFromTinker(getApplicationContext(), "lib/" + abi, so 库的模块名); // 加载任意 abi 库

TinkerLoadLibrary.loadArmLibrary(getApplicationContext(), so 库的模块名); // 只适用于加载 armeabi 库

TinkerLoadLibrary.loadArmV7Library(getApplicationContext(), so 库的模块名); // 只适用于加载 armeabi-v7a 库
```

复制代码

loadArmLibrary()与 loadArmV7Library()本质是调用了 loadLibraryFromTinker()，有兴趣的可以查看下源码。

对于 Tinker 所有 API 的详细说明，请参考：「Tinker 官方 Wiki：Tinker-API 概览」。

七、测试

因为布局简单且不是重点，这里就给出一张 Demo 的运行图片，剩下的就靠想像了。

BAT交流群：892872246

1、编译基础包

没有基础包，那要补丁有什么用？所以，第一步就是打包一个 apk。

在 Terminal 中使用命令行./gradlew assembleDebug。不会命令行无所谓，

Android Studio 为我们提供了图形化操作，根据下图操作即可：

如果你是要 release 签名的打包，则双击 assembleRelease，不过还要配置签名文件，这个后面再说。

编译完成后，可以在 build 目录下会自动创建一个 bakApk 文件夹，里面就有打包好的 apk 文件，因为之后的所有生成的补丁包都以这个 apk 会标准，所以这就是那个基础包文件（相当于应用市场上的 app）。

如果这个 apk 文件是 release 签名且是要放到应用市场上的，那么你必须将 apk 与 R.txt（如果有使用混淆的话，还会有一个 mapping.txt）这几个文件保存好，切记。

现在就把这个 tinker-local-debug-1206-11-48-42.apk 安装到手机上（相当于是用户手机上的 app）。

BAT交流群：892872246

1.点击"say someting"按钮吐司"Hello"。

2.点击"get string from .so"按钮吐司"hello LQR"。

3.点击"show info"按钮显示"patch is not loaded"，说明当前没有加载补丁。

2、修复 java 代码

下面是"say someting"按钮点击时，调用的方法，使用 Toast 显示 Hello 字符串：

BAT交流群：892872246

```
public void say(View view) {
    Toast.makeText(getApplicationContext(), "Hello", Toast.LENGTH_SHORT).show();
}
```
复制代码

1）修复代码

现在我想让它吐司 Hello World，所以代码修改为：

```
public void say(View view) {
    Toast.makeText(getApplicationContext(), "Hello World", Toast.LENGTH_SHORT).show();
}
```
复制代码

2）制作补丁包

先将基础包（前面那个 tinker-local-debug-1206-11-48-42.apk 文件）重命名为 old-app.apk，然后双击 tinkerPatchDebug，操作如下图所示：

BAT交流群：892872246

编译完成后，build/outputs/apk/tinkerPatch 会产生 3 个补丁包，我们要的就是 patch_signed_7zip.apk。

对于 build/outputs/apk/tinkerPatch 目录下文件及文件夹的详细说明，请参考：

「Tinker 官方 Wiki：输出文件详解」。

3）下发补丁包

将 patch_signed_7zip.apk 放到手机的 SD 卡目录下：

> 不一定是 SD 卡目录，位置由我们开发者决定，Demo 中调用 TinkerInstaller.onReceiveUpgradePatch(context, 补丁包的本地路径)方法时，第二个参数指定了是 SD 卡，故如此操作。

BAT交流群：892872246

4）打补丁

可以看到，在 install patch 之前，点击"say someting"按钮时，还是吐司"Hello"。点击"install patch"按钮后，会提示"patch success,please restart process"，说明 Tinker 已经打上补丁了。 这时点击"show info"，可以看到"patch is not loaded"，说明当前补丁还没有生效。 最后，点击"kill myself"按钮，杀死当前 app（进程）。

BAT交流群：892872246

在重新打开 app，再点击"say someting"按钮，吐司"Hello World"。 再点击"show info"，可以看到"patch is loaded"，说明 app 重启后补丁生效了。

BAT交流群：892872246

小结： Tinker 热修复无法让补丁实时生效，在重启进程后，补丁才会生效。 Tinker 会在 app 重启后自动应用补丁。

3、修复 so 库

在一开始制作基础包时，工程中就已经加入了一些 so 文件，存放在 src/main/jniLibs 目录下，因为 Android Studio 默认的库目录是 libs（与 src 同级），所以这里需要在 app 的 build.gradle 文件中进行配置，指定 so 库所在文件夹。

BAT交流群：892872246

下面是"get string from .so"按钮点击时调用的方法：

```
public void string_from_so(View view) {

    String string = JniUtil.hello();

    Toast.makeText(getApplicationContext(), string, Toast.LENGTH_SHORT).show();

}
```

复制代码

这个 JniUtil 的代码如下：

```
public class JniUtil {

    public static String LIB_NAME = "LQRJni";

    public JniUtil() {}

    static {

        System.loadLibrary(LIB_NAME);

    }
```

```
    public static native String hello();

}
```

复制代码

加载 so 库有 2 点需要注意：

1.System.loadLibrary(libname)加载固定目录下的库文件，而 System.load(filename)加载指定目录下的库文件。

2.System.loadLibrary(libname)的参数 libname 指的是库的模块名，不是 so 文件的名字，如 libLQRJni.so 文件的模块名实际上是 LQRJni。

> so 文件的制作代码包含在 Demo 中，有兴趣的朋友可以尝试自己制作。

1）替换 so 文件

回归正题，现在 so 库中得到的文字是"Hello LQR"，现在变一下，我需要得到的文字是"Hello CSDN_LQR"，将新的 so 文件替换掉旧的 so 文件即可。

2）检查 Tinker 的 lib 匹配规则

BAT交流群：892872246

在 app 的 build.gradle 文件中，我们前面在第三部分《Tinker 的配置及任务》的第 2 节《配置 Tinker 与任务》中，有如下一段配置：

```
lib {

    pattern = ["lib/*/*.so", "src/main/jniLibs/*/*.so"]

}
```

复制代码

这就是 Tinker 的 lib 匹配规则，在生成补丁的过程中，它会去把符合这个规则的库文件拿出来与基础包中的库文件进行匹配，从而将有差异的库文件放入到补丁包中。而 Tinker 官方 Demo 的配置中是没有"src/main/jniLibs//.so"这一段的，这将导致 Tinker 在产生补丁包时不会去检查 src/main/jniLibs 目录下的文件变化，进而补丁包中不会包含修复好的 so 文件，这很重要，切记。

3）生成补丁与下发补丁包

生成补丁与下发补丁包的过程与之前的操作一致，这里不再重覆，不过我们来看看 tinkerPatch 跟之前有什么区别吧：

BAT交流群：892872246

最后记得将 patch_signed_7zip.apk 放到手机的 SD 卡目录下。

4）卸载补丁

补丁是可以打多个的，用补丁的版本号做区分，在卸载的时候，可以根据补丁的版本号来卸载，也可以把之前所有的补丁卸载掉，实际开发中，看项目需求来解决用哪种方式来卸载补丁，这里我选择清理之前所有的补丁，下面是"uninstall patch"按钮的点击事件：

```
public void uninstall_patch(View view) {

    ShareTinkerInternals.killAllOtherProcess(getApplicationContext());

    Tinker.with(getApplicationContext()).cleanPatch();

}
```

复制代码

> 卸载补丁之前，要先杀死当前 App 的其他进程。 卸载补丁之后，App 是不安全的（因为此时 Tinker 已经初始化完成），最好，需要重启一下 App。

5）打补丁

现在我们再来打一次补丁，操作如下图所示：

BAT交流群：892872246

可以看到，在 install patch 之前，点击"get string from .so"按钮时，还是吐司"Hello LQR"。 点击"install patch"按钮后，会提示"patch success,please restart process"，说明 Tinker 已经打上补丁了。 这时点击"show info"，可以看到"patch is not loaded"，说明当前补丁还没有生效。 最后，点击"kill myself"按钮，杀死当前 app（进程）。

现在重启 app，理想状态下，当我们再次点击"get string from .so"按钮时，会吐司"Hello CSDN_LQR"。

BAT交流群：892872246

然而，吐司还是"Hello LQR"，并没有变化，而且点击"show info"按钮后，可以看到"patch is loaded"，说明补丁已经加载了，这是为啥？

再来看看下面的操作：

重启 app 后，先点击"load library(hack)"按钮，再点击"get string from .so"按钮，出现了，吐司变成了"Hello CSDN_LQR"。

上图是 Tinker 官网 Wiki 的文档部分截图，从红线部分可以知道，因为部分手机判断 abi 并不准确（可能因为 Android 碎片化比较严重吧），Tinker 没有区分 abi，自然也不会在 app 启动时，自动加载对应的 so 库，这需要开发者自己判断。

下面是"load library(hack)"按钮点击调用的方法：

```
public void load_library_hack(View view) {
    String CPU_ABI = android.os.Build.CPU_ABI;
    // 将 tinker library 中的 CPU_ABI 架构的 so 注册到系统的 library path 中。
    TinkerLoadLibrary.installNavitveLibraryABI(this, CPU_ABI);
}
```

复制代码

这是 Tinker 提供的使用 Hack 方式加载补丁中的 so 库，只是一个方法调用而已，并没有什么特别。对于非 Hack 方式加载补丁的方式，我本人是没有测试成功的，很奇怪，搞不明白问题的原因，官方的文档也写得不清不楚的，有知道本 Demo 加载不成功的原因的朋友请不吝赐教一下哈，thx。

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

小结： Tinker 虽然会在 app 重启后自动加载补丁，但不会自动加载补丁中的 so 文件，开发者需自己判定好 abi 来加载 so 文件。

4、修复资源文件

这部分跟前面的重合度极高，故不做演示了，你可以在补丁包中对本 demo 中的头像进行替换试试，与修复 java 文件的操作基本一致，这部分需要提醒的是，app 的 build.gradle 文件中 Tinker 配置有如下这一段：

```
res {
    pattern = ["res/*", "assets/*", "resources.arsc", "AndroidManifest.xml"]
    ...
}
```

复制代码

不难理解，这就是 Tinker 对资源文件的匹配规则，日常开发够用，如果你的项目中把资源文件放到了这里没有的目录下，需要修改这部分的配置。

八、细节

1、打包步骤

1）debug 打包

1.调用 assembleDebug 编译得到一个 debug 签名的 apk（old apk），这是基础 apk。

2.修改代码、更新 res 文件、so 等。

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

3.将 old apk 按 gradle 中的参数规则，重命名为指定名字，还是放在 bakApk 目录下（该目录可更改）。

4.调用 tinkerPatchDebug 生成补丁包于/build/outputs/tinkerPatch/目录（默认是 patch_signed_7zip.apk）。

5.将补丁包复制到 SD 卡目录下（目录可更改），在程序中调用打补丁方法，重启 app 即可实现热修复。

2）release 打包步骤

1.调用 assembleRelease 编译得到一个 release 签名的 apk（old apk），这是基础 apk，还有一个 mapping 文件。

2.修改代码、更新 res 文件、so 等。

3.将 old apk 与 mapping 文件按 gradle 中的参数规则，分别重命名为指定名字，还是放在 bakApk 目录下（该目录可更改）。

4.调用 tinkerPatchRelease 生成补丁包于/build/outputs/tinkerPatch/目录（默认是 patch_signed_7zip.apk）。

5.将补丁包复制到 SD 卡目录下（目录可更改），在程序中调用打补丁方法，重启 app 即可实现热修复。

因为调用 tinker 的 release 打包需要用到签名文件的信息，所以还必须在 app 的 build.gradle 中配置好签名文件。

BAT交流群：892872246

```
android {
    ...
    signingConfigs {
        release {
            try {
                storeFile file("./keystore/release.keystore")
                storePassword "testres"
                keyAlias "testres"
                keyPassword "testres"
            } catch (ex) {
                throw new InvalidUserDataException(ex.toString())
            }
        }

        debug {
            storeFile file("./keystore/debug.keystore")
        }
    }
    ...
    buildTypes {
        release {
            minifyEnabled true
```

```
            signingConfig signingConfigs.release
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
project.file('proguard-rules.pro')
        }
        debug {
            debuggable true
            minifyEnabled false
            signingConfig signingConfigs.debug
        }
    }
}
```

复制代码

其实说白了，debug 与 release 打包的差别，除了执行的命令不一样之外，release 打包比 debug 打包多用到 2 个文件（mapping.txt、R.txt）。

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

2、使用 tinker 的注意事项与发现

1.tinker 编译时需要禁用 instant run。

2.tinker 需要 MultiDex。

3.上架前用 assembleRelease 编译得到的 apk、mapping.txt、R.txt 这 3 个文件要备份好，制作补丁时会用到。

4.多个补丁包的版本一样时，不影响打补丁（如：第一次补丁版本是 1.0，第二次补丁还是 1.0 版本，是可以成功打上第二次补丁的）。

5.成功打上补丁后，补丁原文件会被删除，故项目中不必担心补丁原文件清理的问题。

3、可能会遇到的错误

1）onLoadPatchListenerReceiveFail code 为-2

报错原文如下：

```
receive a patch file: /storage/emulated/0/patch_signed_7zip.apk, file size:3604

patch loadReporter onLoadPatchListenerReceiveFail: patch receive fail:

/storage/emulated/0/patch_signed_7zip.apk, code: -2
```

复制代码

出现这种情况，请按如下两步进行排查：

1.查看文件路径是否正常。

2.查看清单文件中是否有添加 SD 卡访问权限。

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

如果你的手机是 Android7.0 请要考虑 FileProvider（Android7.0 不支持直接访问 sd 卡）。

2）onLoadPatchListenerReceiveFail code 为-24

报错原文如下：

```
receive a patch file: /storage/emulated/0/patch_signed_7zip.apk, file size:3665

get platform:null

patch loadReporter onLoadPatchListenerReceiveFail: patch receive fail:

/storage/emulated/0/patch_signed_7zip.apk, code: -24
```

复制代码

提示很明显，Tinker 获取不到 platform 的值，请检查在 app 的 build.gradle 文件中是否有如下配置，这部分配置了 Tinker 补丁包支持的平台与版本号：

```
packageConfig {

    configField("platform", "all")

    configField("patchVersion", "1.0")

}
```

复制代码

第二章、插件化框架设计

第一节、Class 文件与 Dex 文件的结构解读

一、Class 文件

BAT交流群：892872246

Class 文件是 Java 虚拟机定义并被其所识别的文件格式，通俗地讲，每一个接口或者类对应一个 Class 文件。

1.Class 文件格式

1.Class 文件是一组以 8 位字节为基础单位的的二进制流。

2.各数据项目之间没有任何分隔符

3.Class 文件格式采用的结构只有两种数据结构：无符号数和表。

4.无符号数：以 u1,u2 等表示 1，2 个字节的无符号数，无符号数可用于描述数字、索引引用、数量值、字符串值。

5.表：以多个无符号数或者表组成，通常以_info 结尾，整个 Class 文件就是一张表。

BAT交流群：892872246

2.Class 文件的组成：

类型	名称	数量
u4	magic(魔术)	1
u2	minor_version(次版本号)	1
u2	major_version(主版本号)	1
u2	constant_pool_count(常量个数)	1
cp_info	constant_pool(常量池表)	constant_pool_count-1
u2	access_flags(类的访问控制权限)	1
u2	this_class(类名)	1
u2	super_class(父类名)	1
u2	interfaces_count(接口个数)	1
u2	interfaces(接口名)	interfaces_count
u2	fields_count(域个数)	1
field_info	fields(域的表)	fields_count
u2	methods_count(方法的个数)	1
method_info	methods(方法表)	methods_count
u2	attributes_count(附加属性的个数)	1
attribute_info	attributes(附加属性的表)	attributes_count

- 魔数：Class 文件开始四个字节是魔数，用于验证该文件是否能被虚拟机执行
- 版本号：魔数后四个字节则是 Class 文件的主版本号和次版本号
- 常量池：版本号后面紧跟着的则是常量池部分。常量池不同与 java 中，它是由 1 开始计数的，常量痴表的数目是常量个数减一。第 0 项为保留项目，用于指示当前 Class 文件不需要引用任何一个常量池。常量池中主要包含两大类常量：
  - 字面量：包括文本字符串，被声明为 final 的常量值等。

BAT交流群：892872246

- 符号引用：包括类和接口的权限定名，字段的名称和描述符，方法的名称和描述符
- 访问标志：常量池部分结束后则是访问标志部分，该部分包括信息：是类还是接口，是否定义为 public，是否 final，是否 abstract 等信息。
- 访问标志之后则是类索引，父类索引，接口索引集合部分，该部分的主要信息为：该类的信息的索引，父类的信息的索引，实现接口的个数和对应接口信息的索引。（通过指向一个类型为 CONSTANT_Class_info 的类描述符常量，从而在对应的常量池找到相关信息）
- 字段表：接下来的部分则是用于描述类声明的变量信息，包括变量的作用域，是否为 static，是否为 final 等信息。
- 方法表：方法表则是用于描述类中方法的信息，与字段表类似。
- 属性表：属性表在字段表和方法表中都会出现，主要用于描述某些场景特有的信息。比如方法是否抛出异常，被 final 声明的变量的值，内部类列表等信息。

二、Dex 文件

Dex 是 Android 平台上(Dalvik 虚拟机，art 虚拟机)的可执行文件，每个 APK 压缩包中都包含一个（或者多个 MultiDex）Dex 文件，Dex 文件中包含了 app 的所有源码。

1.Dex 文件的结构

1.Dex 文件是一组以 8 位字节为基础单位的的二进制流。

2.Dex 文件的各数据项目之间也没有任何分隔符

3.Dex 文件由文件头，索引区，数据区三个部分组成

BAT交流群：892872246

其各个元素的解释如下：

-
- header：dex 文件头部，记录整个 dex 文件的相关属性
- string_ids：字符串数据索引，记录了每个字符串在数据区的偏移量
- type_ids：类型数据索引，记录了每个类型的字符串索引
- proto_ids：原型数据索引，记录了方法声明的字符串，返回类型字符串，参数列表
- field_ids：字段数据索引，记录了所属类，类型以及方法名
- method_ids：类方法索引，记录方法所属类名，方法声明以及方法名等信息

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

- class_defs：类定义数据索引，记录指定类各类信息，包括接口，超类，类数据偏移量
- data：数据区，保存了各个类的真实数据
- link_data：连接数据区

2.header

文件头记录了 dex 文件的一些基本信息，以及大致的数据分布。其各字段及解释如下：

字段名称	偏移值	长度	说明
magic	0x0	8	魔数字段，值为"dex\n035\0"
checksum	0x8	4	校验码
signature	0xc	20	sha-1签名
file_size	0x20	4	dex文件总长度
header_size	0x24	4	文件头长度，009版本=0x5c,035版本=0x70
endian_tag	0x28	4	标示字节顺序的常量
link_size	0x2c	4	链接段的大小，如果为0就是静态链接
link_off	0x30	4	链接段的开始位置
map_off	0x34	4	map数据基址
string_ids_size	0x38	4	字符串列表中字符串个数
string_ids_off	0x3c	4	字符串列表基址
type_ids_size	0x40	4	类列表里的类型个数
type_ids_off	0x44	4	类列表基址
proto_ids_size	0x48	4	原型列表里面的原型个数
proto_ids_off	0x4c	4	原型列表基址
field_ids_size	0x50	4	字段个数
field_ids_off	0x54	4	字段列表基址
method_ids_size	0x58	4	方法个数
method_ids_off	0x5c	4	方法列表基址
class_defs_size	0x60	4	类定义标中类的个数
class_defs_off	0x64	4	类定义列表基址
data_size	0x68	4	数据段的大小，必须4k对齐
data_off	0x6c	4	数据段基址

,/2246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

3.索引区

索引区中索引了整个 dex 中的字符串、类型、方法声明、字段以及方法的信息，其结构体的开始位置和个数均来自 dex 文件头中的记录

- 字符串索引区：描述 dex 文件中所有的字符串信息
- 类型索引区：描述 dex 文件中所有的类型，如类类型、基本类型、返回值类型等
- 方法声明索引区：描述 dex 文件中所有的方法声明
- 字段索引区：描述 dex 文件中所有的字段声明，这个结构中的数据全部都是索引值，指明了字段所在的类、字段的类型以及字段名称
- 方法索引区：描述 Dex 文件中所有的方法，指明了方法所在的类、方法的声明以及方法名字

4.数据区

索引区中的最终数据偏移以及文件头中描述的偏移都指向数据区，还包括了即将要解析的 class_def_item

- class_def_item：这个结构由 dex 文件头中的 classDefsSize 和 classDefsOff 所指向，描述 Dex 文件中所有类定义信息，每一个 DexClassDef 中包含一个 DexClassData 的结构(classDataOff),每一个 DexClassData 中包含了一个 Class 的数据, Class 数据中包含了所有的方法，方法中包含了该方法中的所有指令

BAT交流群：892872246

三、Class 文件和 Dex 文件对比

- Class 文件是 JVM 的执行文件，而 Dex 文件是安卓虚拟机（Dalvik，art）的执行文件
- Class 文件对应 Java 中每一个类，而 Dex 文件则包含整个 APK 文件中的 JAVA 代码，故 Dex 文件可以大大缩小体积，减少 Class 文件的数据冗余，有利于在移动端中运行。（可以通过 SDK 中的 dx 命令将若干个 Class 文件转化为 Dex 文件）

第二节、Android 资源加载机制详解

Android 提供了一种非常灵活的资源系统，可以根据不同的条件提供可替代资源。因此，系统基于很少的改造就能支持新特性，比如 Android N 中的分屏模式。这也是 Android 强大部分之一。本文主要讲述 Android 资源系统的实现原理，以及在应用开发中需要注意的事项。

一、定义资源

Android 使用 XML 文件描述各种资源，包括字符串、颜色、尺寸、主题、布局、甚至是图片（selector，layer-list）。

资源可分为两部分，一部分是属性，另一部分是值。对于 `android:text="hello, world"`，`text` 就是属性，`hello, world` 就是值。

BAT交流群：892872246

1. 属性的定义

在 APK 程序中，属性定义在 `res/values/attrs.xml` 中，在系统中属性位于 `framework/base/core/res/res/values/attrs.xml` 文件中。具体定义如下所示：

```
<declare-styleable name="Window">    <attr name="windowBackground" forma
t="reference"/>    <attr name="windowContentOverlaly" />    <attr name="
windowFrame" />    <attr name="windowTitle" /></declare-styleable>
```

styleable 相当于一个属性集合，其在 R.java 文件中对应一个 int[]数组，aapt 为 styleable 中的每个 attr（属性）分配一个 id 值，int[]中的每个 id 对应着 styleable 中的每一个 attr。

对于`<declare-styleable name="Window">`，Window 相当于属性集合的名称。

对于`<attr name="windowBackground">`，windowBackground 相当于属性的名称；属性名称在应用程序范围内必须唯一，既无论定义几个资源文件，无论定义几个 styleable，windowBackground 必须唯一。

在 Java 代码中，变量在一个作用域内只能声明一次，但可以多次使用。`attr` 也是一样，只能声明一次，但可以多处引用。如上代码所示，在 Window 中声明了一个名为 windowBackground 的 `attr`，在 Window 中引用了一个名为 windowTitle 的 `attr`。

如果一个 `attr` 后面仅仅有一个 `name`，那么这就是引用；如果不光有 `name` 还有 `format` 那就是声明。windowBackground 是属性的声明，其不能在其他 styleable 中再次声明；windowTitle 则是属性的引用，其声明是在别的 styleable 中。

2. 值的定义

常见的值一般有以下几种：

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

- String，Color，boolean，int 类型：在 `res/values/xxx.xml` 文件中指定
- Drawable 类型：在 `res/drawable/xxx` 中指定
- layout（布局）：在 `res/layout/xxx.xml` 中指定
- style（样式）：在 `res/values/xxx.xml` 中指定

值的类型大致分为两类，一类是基本类型，一类是引用类型；对于 int，boolean 等类型在声明属性时使用如下方式：

```
<attr name="width" format="integer"/>
<attr name="text" format="string" />
<attr name="centerInParent"="boolean"/>
```

对于 Drawable，layout 等类型在声明属性时：

```
<attr name="background" format="reference"/>
```

二、解析资源

资源解析主要涉及到两个类，一个是 AttributeSet，另一个是 TypedArray。

1. AttributeSet

该类位于 android.util.AttributeSet，纯粹是一个辅助类，当从 XML 文件解析时会返回 AttributeSet 对象，该对象包含了解析元素的所有属性及属性值。并且在解析的属性名称与 attrs.xml 中定义的属性名称之间建立联系。AttributeSet 还提供了一组 API 接口从而可以方便的根据 attrs.xml 中已有的名称获取相应的值。

如果使用一般的 XML 解析工具，则可以通过类似 getElementById()等方法获取属性的名称和属性值，然而这样并没有在获取的属性名称与 attrs.xml 定义的属性名称之间建立联系。

Attribute 对象一般作为 View 的构造函数的参数传递过来，例如：

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
publlic TextView(Context context,AttributeSet attrs,int defStyle)
```

AttributeSet 中的 API 可按功能分为以下几类，假定 TextView 定义如下所示：

```
<TextView

android:id="@+id/tv"

android:layout_width="@dimen/width"

android:layout_height="wrap_content"

style="@stylel/text"/>
```

第一类，操作特定属性：

- `public String getIdAttribute()`，获取 id 属性对应的字符串，此处返回"@+id/tv"
- `public String getStyleAttribute()`，获取 style 属性对应的字符串，返回"@style/text"
- `public int getIdAttributeResourceValue(int defaultValue)`，返回 id 属性对应的 int 值，此处对应 R.id.tv。

第二类，操作通用属性：

- `public int getAttributeCount()`，获取属性的数目，本例中返回 4
- `public String getAttributeName(int index)`，根据属性所在位置返回相应的属性名称。例如，id=0，layout_width=1，layout_height=2，style=3，如果 getAttributeName(2)，则返回 android:layout_height
- `public String getAttributeValue(int index)`,根据位置返回值。本例中，getAttributeValue(2)则返回"wrap_content"。

BAT交流群：892872246

- `public String getAttributeValue(String namespace,String name)`，返回指定命名空间，指定名称的属性值，该方法说明 AttributeSet 允许给一个 XML Element 的属性增加多个命名空间的属性值。
- `public int getAttributeResource(int index)`，返回指定位置的属性 id 值。本例中，getAttributeResource(2)返回 R.attr.layout_width。前面也说过，系统会为每一个 attr 分配一个唯一的 id。

第三类，获取特定类型的值：

- `public XXXType getAttributeXXXType(int index,XXXType defaultValue)`，其中 XXXType 包括 int、unsigned int、boolean、float 类型。使用该方法时，必须明确知道某个位置（index）对应的数据类型，否则会返回错误。而且该方法仅适用于特定的类型，如果某个属性值为一个 style 类型，或者为一个 layout 类型，那么返回值都将无效。

2. TypedArray

程序员在开发应用程序时，在 XML 文件中引用某个变量通常是 android:background="@drawable/background"，该引用对应的元素一般为某个 View/ViewGroup，而 View/ViewGroup 的构造函数中会通过 obatinStyledAttributes 方法返回一个 TypedArray 对象，然后再调用对象中的 getDrawable()方法获取背景图片。

TypedArray 是对 AttributeSet 数据类的某种抽象。对于 `andorid:layout_width="@dimen/width"`，如果使用 AttributeSet 的方法，仅仅能获取"@dimen/width"字符串。而实际上该字符串对应了一个 dimen 类型的数据。

TypedArray 可以将某个 AttributeSet 作为参数构造 TypedArray 对象，并提供更方便的方法直接获取该 dimen 的值。

```
TypedArray a = context.obtainStyledAttributes(attrs,com.android.internal.
R.styleable.XXX,defStyle,0);
```

方法 obtainStyledAttributes()的第一个参数是一个 AttributeSet 对象，它包含了一个 XML 元素中定义的所有属性。第二个参数是前面定义的 styleable，appt 会把一个 styleable 编译成一个 int[]数组，该数组的内部实现正是通过遍历 AttributeSet 中的每一个属性，找到用户感兴趣的属性，然后把值和属性经过重定位，返回一个 TypedArray 对象。想要获取某个属性的值则调用相关的方法即可，比如 TypedArray.getDrawbale()，TypedArray.getString()等。getDrawable()，getString()方法内部均通过 Resources 获取属性值。

三、加载资源

在使用资源时首先要把资源加载到内存。Resources 的作用主要就是加载资源，应用程序需要的所有资源（包括系统资源）都是通过此对象获取。一般情况下每个应用都会仅有一个 Resources 对象。

要访问资源首先要获取 Resources 对象。获取 Resources 对象有两种方法，一种是通过 Context，一种是通过 PackageManager。

BAT交流群：892872246

1. 使用 Context 获取 Resources

抽象类Context内部个有getResources()方法，一般是在Activity对象或者Service对象中调用，因为 Activity 或者 Service 的本质是一个 Context，而真正实现 Context 接口的是 ContextImpl 类。

ContextImpl 对象是在 ActivityThread 类中创建，所以 getResources()方法实际上是调用 ContextImpl.getResources()方法。在 ContextImpl 类中，该方法仅仅是返回内部的 mResources 变量，而对该变量赋值是在 init()方法中。在创建 ContextImpl 对象后，一般会调用 init()方法对 ContextImpl 对象内部变量初始化，其中就包括 mResources 变量，如以下代码所示：

```
final void init(ActivityThread.PackageInfo packageInfo, IBinder activity
Token, ActivityThread mainThread, Resources container){

mPackageInfo = packageInfo;

mResources = mPackageInfo.getResources(mainThread);}
```

从以上代码可以看出，mResources 又是调用 mPackageInfo 的 getResources()方法进行赋值。一个应用程序中可以有多个 ContextImpl，但多个 ContextImpl 对象共享一个 PackageInfo 对象。所以多个 ContextImpl 对象中的 mResources 变量实际上是同一个 Resources 对象。

PackageInfo.getResources()方法如下所示：

```
public Resources getResources(ActivityThread mainThread){  if(mResources
 == null){

mResources = mainThread.getTopLevelResources(mResDir,this);  }}
```

以上代码中，参数 mainThread 指的就是 ActivityThread 对象，每个应用程序只有一个 ActivityThread 对象。getTopLevelResources()方法就是获取本应用程序中的 Resources 对象。

在 ActivityThread 对象中，使用 `HashMap<ResourcesKey,WeakReference<Resources>> mActiveResources` 保存该应用程序所有的 Resources 对象，并且这些 Resources 都是以一个弱引用保存起来的，这样在内存紧张时可以释放 Resources 所占的内存。

在 mActiveResources 中，使用 ResourcesKey 映射 Resources 类，ResourcesKey 仅仅是一个数据类，其创建方式如下所示：

```
ResourcesKey key = new ResourcesKey(resDir,compInfo.applicatioScale);
```

resDir 变量代表资源文件所在路径，实际是指 APK 程序所在路径，例如 `/data/app/xxx.apk`。该 APK 会对应/data/dalvik-cache 目录下的 data@app@xxx.apk@classes.dex 文件，这两个文件也是应用程序安装后自动生成的文件。

如果一个应用程序没有访问该应用程序以外的资源，那么 mActivieResources 变量中就仅有一个 Resources 对象。当应用程序想要访问其他应用程序的资源则需要构建不同的 ResourcesKey，也就是需要不同的 resDir，毕竟每一个 ResourcesKey 对应一个 Resources 对象，这样该应用程序就可以访问其他应用程序中的资源。

如果 mActiveResources 中还没有包含所要的 Resources 对象，那就需要重新创建一个：

BAT交流/2246

```
AssetManager assets = new AssetManager();if(assets.addAssetPath(resDir)
== 0){  return null;}DisplayMetrics metrics = getDisplayMetricsLocked(fa
lse);

r = new Resources(assets,metrics,getConfiguration(),compInfo);
```

创建 Resources 需要一个 AssetManager 对象。在开发应用程序时，使用 Resources.getAssets()获取的就是这里创建的 AssetManager 对象。

AssetManager 其实并不只是访问 res/assets 目录下的资源，而是可以访问 res 目录下的所有资源。

AssetManager 在初始化的时候会被赋予两个路径，一个是应用程序资源路径 `/data/app/xxx.apk`，一个是 Framework 资源路径 `/system/framework/framework-res.apk`(系统资源会被打包到此 apk 中）。所以应用程序使用本地 Resources 既可访问应用程序资源，又可访问系统资源。

AssetManager 中很多获取资源的关键方法都是 native 实现，当使用 getXXX(int id)访问资源时，如果 id 小于 0x1000 0000 时表示访问系统资源，如果 id 都大于 0x7000 0000 则表示应用资源。aapt 在对系统资源进行编译时，所有资源 id 都被编译为小于 0x1000 0000。

当创建好 Resources 后就把该对象放到 mActivieResources 中以便以后继续使用。

2. 使用 PackageManager 获取 Resources

892872246

该方法主要是用来访问其他应用程序中的资源，最典型的就是切换主题，但这种主题一般仅限于一个应用程序内部。获取 Resources 的过程如下所示：

使用 PackageManager 获取 Resources 对象：

```
PackageManager pm = mContext.getPackageManager();pm.getResourcesForAppli
cation("com.android...your package name");
```

其中 getPackageManager()返回一个 PackageManager 对象，PackageManager 本身是一个 abstract 类，其真正实现类是 ApplicationPackageManager。其内部方法一般调用远程 PackageManagerService。ApplicationPackageManager 在构造时传入一个远程服务的引用 IPackageManager，该对象是通过调用 getPackageManager()静态方法获取的。这种获取远程服务的方法和大多数获取远程服务的方法类似：

```
public static IPackageManager getPackageManager(){  if(sPackageManager !
=null){    return sPackageManager;  }  IBinder b = ServiceManager.getServ
ice("package");

sPackageManager = IPackageManager.Stub.asInterface(b);  return sPackageM
anager;}
```

获得了 PackageManager 对象后，接着调用 getResourcesForApplication()方法，该方法位于 ContextImpl.ApplicationPackageManager 中：

```
@Override  public Resources getResourcesForApplication(ApplicationInfo a
pp) throws NameNotFoundException{  if(app.packageName.equals("system"))
{    return mContext.mMainThread.getSystemContext().getResources();  }
Resources r = mContext.mMainThread.getTopLevelResources(app.uid == Proce
ss.myUid() ? app.sourceDir : app.publicSourceDir,mContext.mPackageInfo);
  if(r != null){    return r;  }  throw new NameNotFoundException("Unable
 to open " + app.publicSourceDir);}
```

以上代码内部调用 mMainThread.getTopLevelResources()方法，又回到了使用 Context 获取 Resources 对象的过程中。注意，此处调用参数的含义：如果目标资源程序和当前程序是同一个 uid，那么就使用目标程序的 sourceDir 作为路径，否则就使用目标程序的 publicSourceDir 目录，该目录可以在 AndroidManifest.xml 中指定。在大多数情况下，目标程序和当前程序不属于同

12246

一个 uid，因此，多为 publicSourceDir，而该值默认情况下和 sourceDir 的值相同。

当进入 mMainThread.getTopLevelResources()方法后，ActivityThread 对象就会在 mActivieResources 变量中保存一个新的 Resources 对象，其键值对应目标程序的包名。

3. 加载应用程序资源

应用程序打包的最终文件是 xxx.apk。APK 本身是一个 zip 文件，可以使用压缩工具解压。系统在安装应用程序时首先解压，并将其中的文件放到指定目录。其中有一个文件名为 resources.arsc，APK 所有的资源均在其中定义。

resources.arsc 是一种二进制格式的文件。aapt 在对资源文件进行编译时，会为每一个资源分配唯一的 id 值，程序在执行时会根据这些 id 值读取特定的资源，而 resources.arsc 文件正是包含了所有 id 值得一个数据集合。在该文件中，如果某个 id 对应的资源是 String 或者数值（包括 int，long 等），那么该文件会直接包含相应的值，如果 id 对应的资源是某个 layout 或者 drawable 资源，那么该文件会存入对应资源的路径地址。

事实上，当程序运行时，所需要的资源都要从原始文件中读取（APK 在安装时都会被系统拷贝到`/data/app`目录下）。加载资源时，首先加载 resources.arsc，然后根据 id 值找到指定的资源。

4. 加载 Framework 资源

系统资源是在 zygote 进程启动时被加载的，并且只有当加载了系统资源后才开始启动其他应用进程，从而实现其他应用进程共享系统资源的目标。

群：892872246

启动第一步就是加载系统资源，加载完毕后再调用 startSystemServer()启动系统进程，并最后调用 runSelectLoopMode()开始监听 Socket，并启动指定的应用进程。加载系统资源是通过 preLoadResources()完成的，该方法关键代码如下所示：

```
mResources = Resources.getSystem();

mResources.startPreLoading();if(PRELOAD_RESOURCES){  long startTime = Sy
stemClock.uptimeMillis();

TypeArray ar = mResources.obtainTypedArray(com.android.internal.R.array.
preloadingdrawables);  int N = prelaodDrawables(runtime,ar);

Log.i(TAG,"...preloading " + N + "resources in " + (SystemClock.uptimeMi
llis()-startTime) + "ms.");

startTime = SystemClock.uptimeMillis();

ar = mResources.obtainTypedArray(com.android.internal.R.array.preloading
_color_state_lists);

N = preloadingColorStateLists(runtime,ar);

Log.i(TAG,"...preloaded " + N + "resources in " + (SystemClock.uptimeMil
lis()-startTime) + "ms.");}

mResources.finishPreloading();
```

在以上代码中使用 Resources.getSystem()创建 Resources 对象，一般情况下应用程序不应该调用此方法，因为该方法返回的 Resources 仅能访问 Framework 资源。

当 Resources 对象创建完成后，调用 preloadDrawables()和 preloadColorStateLists()装在需要"预装载"的资源。这两个方法都需要传入一个 TypeArray，其来源是 res/values/arrays.xml 中定义的一个 array 数组资源，例如：

```
<array name="preloaded_drawables">  <item>@drawable/sym_def_app_icon</it
em>  <item>@drawable/arrow_down_float</item></array>
```

```
<array name="preloaded_color_state_lists">  <item>@color/hint_foreground
_dark</item>  <item>@color/hint_foreground_light</item></array>
```

在 Resources 类中，相关资源读取函数需要将读取到的资源缓冲起来，以便以后使用，Resources 类中定义了四个静态变量缓冲这些资源：

```
private static final LongSparseArray<Drawable.ConstantState> sPreloadedD
rawables = new LongSparseArray<Drawable.ConstantState>();private static
final LongSparseArray<ColorStateList> sPreloadedColorStateLists = new Lo
ngSparseArray<ColorStateList>();private static final LongSparseArray<Dra
wable.ConstantState> sPreloadedColorDrawables = new LongSparseArray<Draw
able.ConstantState>();private static boolean mPreloaded;
```

其中前三个变量是列表类型，并且被 static 修饰，所有 Resources 对象均共享这三个变量。所以当应用程序创建新的 Resources 对象时可以访问系统资源。

第四个变量用来区分是 zygote 装在资源还是普通应用进程装在资源。因为 zygote 与普通进程装载资源的方式类似，所以增加 mPreloaded 变量进行区分。

mPreloaded 在 startPreloading()中被置为 true，在 finishPreloading()中被置为 false，而 startPreloading()和 finishPreloading()正是在 ZygoteInit.java 的 preloadResources()中被调用，这就区别了 zygote 调用和普通进程调用。

最后，在 Resources 的具体资源读取方法中，会判断 mPreloaded 变量，如果为 true，则同时把读取到的资源存储到三个静态列表中，否则把资源放到非静态列表中，这些非静态列表的作用范围为调用者所在进程。

Resources.loadDrawable()方法代码如下所示：

```
if(mPreloading){  if(isColorDrawable){

sPreloadedColorDrawables.put(key,cs);  } else {
```

```
sPreloadedDrawables.put(key,cs);  }} else {  synchronized(mTmpValue){
 if(isColorDrawbale){

mColorDrawableCache.put(key,new WeakReference<ColorDrawable>(cs));    }
else {

mDrawableCache.put(key,new WeakReference<Drawable>(cs));    }  }}
```

上面所介绍的资源加载仅仅只是加载在 res/values/arrays.xml 中预先定义的资源值，Framework 包含了更多的资源，zygote 所加载的仅仅是一小部分。对于那些非"预装载"的系统资源则不会被缓冲到静态列表变量中，这时应用进程如果需要一个非预装载资源则会在各自进程中保持一个资源缓冲。

第三节、四大组件调用原理

一、Activity

一、开启插件 Activity 流程

第 1 步

开启插件 Activity 的入口在 Replugin.StartActivity(Context context, Intent intent)，其他 intent 内部一定要传入想要开启的插件名。

```
public static boolean startActivity(Context context, Intent intent) {

        // TODO 先用旧的开启 Activity 方案，以后再优化

        ComponentName cn = intent.getComponent();

        if (cn == null) {

            // TODO 需要支持 Action 方案

            return false;

        }
```

```
        String plugin = cn.getPackageName();

        String cls = cn.getClassName();

        return Factory.startActivityWithNoInjectCN(context, intent, plug
in, cls, IPluginManager.PROCESS_AUTO);

    }
```

Factory.startActivityWithNoInjectCN 内部调用 sPluginManager.startActivity 方法，那么 sPluginManager 是哪里传进去的？

PMF.java

```
Factory.sPluginManager = PMF.getLocal();

Factory2.sPluginManager = PMF.getInternal();
```

看下 PMF.init 的方法可以看到 sPluginManager 是 PMF 内部维护的 sPluginMgr.mLocal，也就是 Pmbase 内部的 PmLocalImpl mLocal 对象，继续跟踪发现真正实现 startActivity 逻辑的是 PmBase 内部的 PmInternalImpl.startActivity 下面这个方法。

```
public boolean startActivity(Context context, Intent intent, String plug
in, String activity, int process, boolean download) {

        。。。

    }
```

第 2 步

重点走读下上面 startActivity 方法，首先会判断要跳转的插件是否已经存在了，如果不存在则回调 callback 接口去提示用户下载等逻辑操作，如果插件状态不正确，则回调外部 callback 去提示用户插件不可用或者去升级，如果插件首次加载

并且是大插件则异步加载并弹窗显示正在加载。然后在调用 context.startActivity 方法前会去通过下面的 loadPluginActivity 方法将目标插件 Activity class 替换为“坑位”Activity，这样其实传给 AMS 的还是宿主的坑位 Activity。

PmInternalmpl.java

```
public boolean startActivity(Context context, Intent intent, String plug
in, String activity, int process, boolean download) {

        if (LOG) {

            LogDebug.d(PLUGIN_TAG, "start activity: intent=" + intent + "
 plugin=" + plugin + " activity=" + activity + " process=" + process + "
download=" + download);

        }



        // 1. 是否启动下载

        // 若插件不可用（不存在或版本不匹配），则直接弹出“下载插件”对话框

        // 因为已经打开 UpdateActivity，故在这里返回 True，告诉外界已经打开，无
需处理

        if (download) {

            if (PluginTable.getPluginInfo(plugin) == null) {

                if (LOG) {

                    LogDebug.d(PLUGIN_TAG, "plugin=" + plugin + " not foun
d, start download ...");

                }



                // 如果用户在下载即将完成时突然点按“取消”，则有可能出现插件已下
载成功，但没有及时加载进来的情况
```

```
                // 因此我们会判断这种情况，如果是，则重新加载一次即可，反之则提示用户下载

                // 原因：“取消”会触发 Task.release 方法，最终调用 mDownloadTask.destroy，导致“下载服务”的 Receiver 被注销，即使文件下载了也没有回调回来

                // NOTE isNeedToDownload 方法会调用 pluginDownloaded 再次尝试加载

                if (isNeedToDownload(context, plugin)) {

                    return RePlugin.getConfig().getCallbacks().onPluginNotExistsForActivity(context, plugin, intent, process);

                }

            }

        }

          ...

        // 2. 如果插件状态出现问题，则每次弹此插件的 Activity 都应提示无法使用，或提示升级（如有新版）

        // Added by Jiongxuan Zhang

        if (PluginStatusController.getStatus(plugin) < PluginStatusController.STATUS_OK) {

            if (LOG) {

                LogDebug.d(PLUGIN_TAG, "PmInternalImpl.startActivity(): Plugin Disabled. pn=" + plugin);

            }

            return RePlugin.getConfig().getCallbacks().onPluginNotExistsForActivity(context, plugin, intent, process);

        }



        // 3. 若为首次加载插件，且是“大插件”，则应异步加载，同时弹窗提示“加载中”
```

```
        // Added by Jiongxuan Zhang

        if (!RePlugin.isPluginDexExtracted(plugin)) {

            PluginDesc pd = PluginDesc.get(plugin);

            if (pd != null && pd.isLarge()) {

                ...

                return RePlugin.getConfig().getCallbacks().onLoadLargePlu
ginForActivity(context, plugin, intent, process);

            }

        }



        // WARNING: 千万不要修改 intent 内容，尤其不要修改其 ComponentName

        // 因为一旦分配坑位有误（或压根不是插件 Activity），则外界还需要原封不
动的 startActivity 到系统中

        // 可防止出现“本来要打开宿主，结果被改成插件”，进而无法打开宿主 Activit
y 的问题



        // 缓存打开前的 Intent 对象，里面将包括 Action 等内容

        Intent from = new Intent(intent);



        // 帮助填写打开前的 Intent 的 ComponentName 信息（如有。没有的情况如直接
通过 Action 打开等）

        if (!TextUtils.isEmpty(plugin) && !TextUtils.isEmpty(activity)) {

            from.setComponent(new ComponentName(plugin, activity));

        }
```

```
        //4. 会去将目标 Activity 替换为坑位 Activity

        ComponentName cn = mPluginMgr.mLocal.loadPluginActivity(intent, p
lugin, activity, process);

        if (cn == null) {

            if (LOG) {

                LogDebug.d(PLUGIN_TAG, "plugin cn not found: intent=" + in
tent + " plugin=" + plugin + " activity=" + activity + " process=" + proc
ess);

            }

            return false;

        }



        // 将 Intent 指向到“坑位”。这样：

        // from：插件原 Intent

        // to：坑位 Intent

        intent.setComponent(cn);





        //5. 调用系统 startActivity 方法

        context.startActivity(intent);



        // 6. 通知外界，已准备好要打开 Activity 了

        // 其中：from 为要打开的插件的 Intent，to 为坑位 Intent

        RePlugin.getConfig().getEventCallbacks().onPrepareStartPitActivi
ty(context, from, intent);
```

```
        return true;
    }
```

坑位 Activity 的替换规格，逻辑是在 PmLocalImpl.allocActivityContainer 方法中执行，这个有兴趣的可以自己去跟踪下，坑位 Activity 替换规格，这里就不深入去分析了。

第 3 步

上述几步使用坑位 Activity 来替换目标 Activity 之后，AMS 内部都是基于这个坑位 Activity 来实现对其合法性、生命周期进行回调啦，AMS 一系列流程走完，然后 AMS 再通过宿主进程的 ActivityThread 提供的 IApplicationThread Binder 代理对象去实现宿主进程中类加载坑位 Activity 啦，并对其回调相关接口如 onCreate。正常 startActivity 是这样没错，但是我们明明是希望调用的宿主的 Activity 啊，又不是坑位 Activity？是的这里需要对坑位 Activity 进行恢复真身处理，恢复为目标插件 Activity。在哪里恢复呢？没错，就是在我们唯一 hook 点：RePluginClassLoader 中。

我们在《Replugin 插件化技术解读之框架初始化、插件安装与加载（二）》一文中已经分析的很明白，从 RePluginClassLoader 的 loadClass 方法中会首先去使用 PMF.loadClass 去加载插件，去生成插件的 Loader 对象，

初始化好插件的 DexClassLoader、Resource、PluginContext 等资源，然后用插件的 ClassLoader 去尝试加载目标 Activity。具体执行逻辑在 PluginProcessPer.resolveActivityClass 方法中。

```
/**
```

BAT /2246

```
 * 类加载器根据容器解析到目标的 activity
 * @param container
 * @return
 */
 final Class<?> resolveActivityClass(String container) {
     String plugin = null;
     String activity = null;

     // 先找登记的，如果找不到，则用 forward activity
     PluginContainers.ActivityState state = mACM.lookupByContainer(cont
ainer);
     if (state == null) {
         // PACM: loadActivityClass, not register, use forward activity,
 container=
         if (LOGR) {
             LogRelease.w(PLUGIN_TAG, "use f.a, c=" + container);
         }
         return ForwardActivity.class;
     }
     plugin = state.plugin;
     activity = state.activity;

     Plugin p = mPluginMgr.loadAppPlugin(plugin);
     if (p == null) {
```

```
        return null;
    }
    ClassLoader cl = p.getClassLoader();
    if (LOG) {
    Class<?> c = null;
    try {
        c = cl.loadClass(activity);
    } catch (Throwable e) {
        if (LOGR) {
            LogRelease.e(PLUGIN_TAG, e.getMessage(), e);
        }
    }
    return c;
}
```

这样我们插件目标 Activity 就恢复成功，目标 Activity 内部逻辑就能正常执行啦。

总结：

开启插件 Activity 的大致流程为：

插件 Activity -> Replugin.startActivity -> 解析出插件 Activity 对应的 pluginName -> 挑选坑位 Activity 替换 ->调用系统 startActivity 方法 -> AMS 回调执行坑位 Activity 类加载 ->加载并初始化目标插件资源、上下文、类加载器(首次) ->Hook 掉 ClassLoader 的类加载中恢复为插件 Activity -> 插件 Activity 启动，并拥有完整生命周期。

BAT交流,12246

二、插件内部 startActivity 流程

插件中 startActiviy 有两种场景：调用自己插件本身的 Activity 或者调用宿主的 Activity。显然光凭插件自己是无法成功开启的。因为插件的 Manifest 根本就没有被 AMS 识别到呀，必须还是要用统一的 Replugin.startActivity 接口去按照上述分析的流程重走一遍才行的啊。

在《Replugin 插件化技术解读之目录结构解读》一文中，我们了解到其实插件在动态编译过程中会在字节码层做适量修改，如插件的所有 Activity 都会被继承 replugin-plugin-lib 中的 PluginActivity 这样一个基类。所以我们插件内部其实 Activity 内部最后执行 startActivity 都会先走 PluginActivity 的此方法；

PluginActivity.java

```
static void initLocked(final ClassLoader classLoader) {



            final String factory2 = "com.qihoo360.i.Factory2";

            final String factory = "com.qihoo360.i.Factory";



            // 初始化 Factory2 相关方法

            createActivityContext = new MethodInvoker(classLoader, factor
y2, "createActivityContext", new Class<?>[]{Activity.class, Context.clas
s});

            handleActivityCreateBefore = new MethodInvoker(classLoader, f
actory2, "handleActivityCreateBefore", new Class<?>[]{Activity.class, Bu
ndle.class});

            handleActivityCreate = new MethodInvoker(classLoader, factory
2, "handleActivityCreate", new Class<?>[]{Activity.class, Bundle.class});
```

```
            handleActivityDestroy = new MethodInvoker(classLoader, factor
y2, "handleActivityDestroy", new Class<?>[]{Activity.class});

            handleRestoreInstanceState = new MethodInvoker(classLoader, f
actory2, "handleRestoreInstanceState", new Class<?>[]{Activity.class, Bu
ndle.class});

            startActivity = new MethodInvoker(classLoader, factory2, "sta
rtActivity", new Class<?>[]{Activity.class, Intent.class});

            startActivityForResult = new MethodInvoker(classLoader, facto
ry2, "startActivityForResult", new Class<?>[]{Activity.class, Intent.cla
ss, int.class, Bundle.class});



            // 初始化 Factory 相关方法

            loadPluginActivity = new MethodInvoker(classLoader, factory,
"loadPluginActivity", new Class<?>[]{Intent.class, String.class, String.
class, int.class});
```

显然，反射的是 Factory2 中的 startActivity 方法，而 Factory.startActivity 其实就是调用的 PmInternalImpl.startActivity 方法，显然这个跟上面宿主启动插件 Activity 对上啦。

那么 RepluginInternal.initLocked 方法是什么时候走的呢？容易看到是在 RepluginFramework.init 中调用，而 RepluginFramework.init 初始化时在 Entry 的 create 方法中，这里是不是很熟悉啦。没错，在《Replugin 插件化技术解读之框架初始化、插件安装与加载（二）》一文中我们知道在加载插件的时候会通过回调执行插件的 Entry 的 create 方法初始化插件，显然这里这条线又连上啦。

这样插件内部启动 startActivity 说白了就是通过反射调用宿主 Factory2 中 startActivity 方法去重走了遍上述分析的标准启动插件 Activity 流程。注意，这

里其实仅仅是通过插件 Activity 继承 PluginActivity 重写了 Activity 内部 StartActivity 方法，改变了 Activity 内 startActivity 方法启动步骤。

我们显然还要修改 Context 上下文对应的 startActivity 方法才能保证这里调用的时候也会走到我们设定好的宿主 startActivityq 启动流程，

那么插件内部其他 startActivity 的地方都要进行这样的调整才行，这个就大家自己跟下就能明白了，原理差不多。

```
public abstract class PluginActivity extends Activity {



    @Override

    protected void attachBaseContext(Context newBase) {

        newBase = RePluginInternal.createActivityContext(this, newBase);

        super.attachBaseContext(newBase);

    }
```

PluginActivity 的 attachBaseContext 方法中我们会使用 RepluginInternal.createActivityContext 方法，通过反射，最终吊起来 PmInternalImpl.createActivityContext 方法。创建对应插件的 PluginContext 对象，PluginContext 前文已经介绍，然后用这个 PluginContext 替换掉插件内部的上下文 BaseContext，其实也就是 Activity 继承的 ContextWrapper 类的 Context mBase 对象。跟踪 Android 源码很容易发现，ContextWrapper 内部封装了 startActivity startService 等方法，所以这里我们要将这个 Context 替换掉。那么 PluginContext 内部又做了什么呢？显然它也重写了 startActivity 方法。

```
@Override
    public void startActivity(Intent intent) {
        if (mContextInjector != null) {
            mContextInjector.startActivityBefore(intent);
        }

        super.startActivity(intent);

        if (mContextInjector != null) {
            mContextInjector.startActivityAfter(intent);
        }
    }
```

这里 super.startActivity 其实就是对应宿主上下文了，直接走宿主 startActivity 流程。

其他还有在 PluginApplicationClient 为插件创建的 Application 对象，在 PluginApplicationClient.callAttachBaseContext 方法中使用反射调用了 Application.attachf 方法，用 PluginContext 对象替换了原来的 Context，这个就各位看官自己去梳理了哈。

二、Service

BAT交流群：892872246

一、开启插件内 Service

第 1 步

在《Replugin 插件化技术解读之框架初始化、插件安装与加载（二）》中我们知道插件在加载的时候会自动生成 PluginContext 来作为插件内上下文使用，而插件中 startService 其实调用的就是已经被重写的 startService 方法了。

PluginContext.java

```
public ComponentName startService(Intent service) {
        if (mContextInjector != null) {
            mContextInjector.startServiceBefore(service);
        }


        。。。
        try {
            return PluginServiceClient.startService(this, service, true);
        } catch (PluginClientHelper.ShouldCallSystem e) {
            // 若打开插件出错，则直接走系统逻辑
            return super.startService(service);
        } finally {
            if (mContextInjector != null) {
                mContextInjector.startServiceAfter(service);
            }
```

```
        }

    }
```

第 2 步

PluginServiceClient.startService 中会首先获取 ComponentName 对象，得到插件包名和对应 Service Class 类型对象，然后获得开启的 Service 所属的进程号，如 UI 进程或者常驻进程。接着会通过进程号获得对应的 IPluginServiceService pss 对象，进行 pss.startService 操作。

```
public static ComponentName startService(Context context, Intent intent,
 boolean throwOnFail) {



        // 1. 从 Intent 中获取 ComponentName，包含想要开启的 ServiceName 以及
插件包名

        ComponentName cn = getServiceComponentFromIntent(context, inten
t);



        // 2. 获取将要使用服务的进程 ID，

        int process = getProcessByComponentName(cn);

        if (process == PROCESS_UNKNOWN) {

            // 有可能是不支持的插件，也有可能本意就是想调用 Main 工程的服务。则
直接调用系统方法来做

            if (LOG) {

                LogDebug.d(PLUGIN_TAG, "PSS.startService(): Call SystemAP
I: in=" + intent);

            }

            if (throwOnFail) {
```

```
                throw new PluginClientHelper.ShouldCallSystem();
            } else {
                return context.startService(intent);
            }
        }

        // 既然确认是插件，则将之前获取的插件信息填入
        intent.setComponent(cn);

        //3. 获取对应进程 ID 的 IPluginServiceServer 对象，IBinder 实现提是 Plu
ginServiceService.Stub
        IPluginServiceServer pss = sServerFetcher.fetchByProcess(proces
s);
        if (pss == null) {
            if (LOGR) {
                LogRelease.e(PLUGIN_TAG, "psc.ss: pss n");
            }
            return null;
        }

        // 开启服务
        try {
            return pss.startService(intent, sClientMessenger);
        } catch (Throwable e) {
```

```
            if (LOGR) {
                LogRelease.e(PLUGIN_TAG, "psc.ss: pss e", e);
            }
        }
        return null;
    }
```

第 3 步

重点看下如果根据不同进程去生成相应的 PSS 对象，也就是方法

sServerFetcher.fetchByProcess 方法

```
public IPluginServiceServer fetchByProcess(int process) {
        if (process == PluginServiceClient.PROCESS_UNKNOWN) {
            return null;
        }
        // 取之前的缓存
        IPluginServiceServer pss;
        synchronized (PSS_LOCKER) {
            pss = mServiceManagerByProcessMap.get(process);
            if (pss != null) {
                if (LOG) {
                    LogDebug.d(PLUGIN_TAG, "PluginServiceClient.fsmbp():
Exists! p=" + process);
                }
                return pss;
```

```
            }

        }



        // 缓存没有？则去目标进程获取新的

        if (LOG) {

            LogDebug.d(PLUGIN_TAG, "PluginServiceClient.fsmbp(): Create a
 new one! p=" + process);

        }

        try {

            if (process == IPluginManager.PROCESS_PERSIST) {

                //1. 如果是常驻进程，则直接获取得到 Pss 对象，PluginServiceSer
ver.mStub

                IPluginHost ph = PluginProcessMain.getPluginHost();

                pss = ph.fetchServiceServer();

            } else {

                //2. 如果是非常驻其他进程，则会通过 MP.startPluginProcess 方
法，利用动态编译自动生成的进程挂在宿主 Manifest 中的 Provider

                //去通过 PluginProviderStub.proxyStartPluginProcess 利用 in
sert 方法去吊起 provider 对应的进程号，开启进程

                //IPluginClient 的实现体就是 PluginProcessPer 中，获得对象同样
是 PluginServiceServer.mStub

                PluginBinderInfo pbi = new PluginBinderInfo(PluginBinderI
nfo.NONE_REQUEST);

                IPluginClient pc = MP.startPluginProcess(null, process, p
bi);

                pss = pc.fetchServiceServer();
```

```
            }



            // 挂死亡周期，如果出问题了就置空重来，防止外界调用 psm 出现 DeadObject 问题

            pss.asBinder().linkToDeath(new PSSDeathMonitor(process, pss.asBinder()), 0);

        } catch (Throwable e) {

            if (LOGR) {

                LogRelease.e(PLUGIN_TAG, "psc.fsm: e", e);

            }

        }

        if (pss != null) {

            synchronized (PSS_LOCKER) {

                mServiceManagerByProcessMap.put(process, pss);

            }

        }

        return pss;

    }
```

1. 显然，针对不同进程号，使用了一个 ArrayMap 来缓存对应 pss 对象，如果是常驻进程，也就是进程号

```
process == IPluginManager.PROCESS_PERSIST
```

会直接获得 PluginServiceServer.mStub 的 IPluginServiceService 对象来进行相关 Service 操作。

1. 如果 Service 所处的是非常驻进程，这里会采用一个比较巧妙的方法去首先拉起该进程，跟踪下面方法：

```
IPluginClient pc = MP.startPluginProcess(null, process, pbi);
```

MP.java

```
public static final IPluginClient startPluginProcess(String plugin, int
process, PluginBinderInfo info) throws RemoteException {

        return PluginProcessMain.getPluginHost().startPluginProcess(plug
in, process, info);

    }
```

显然，会调用常驻进程服务的 PmHostSvc.startPluginProcess 方法去拉起对应进程，PmHostSvc 常驻进程服务再整个 Replugin 确实起到了对各插件管理的作用，包括拉起插件 Service 进程。继续往下看，调用了

PmBase.startPluginProcessLocked 方法：

```
final IPluginClient startPluginProcessLocked(String plugin, int process,
 PluginBinderInfo info) {

    ......

    PluginProcessMain.schedulePluginProcessLoop(PluginProcessMain.CHECK_
STAGE1_DELAY);



    // 尝试从缓存中查找

    IPluginClient client = PluginProcessMain.probePluginClient(plugin, p
rocess, info);

    ......

    int index = IPluginManager.PROCESS_AUTO;
```

```
    try {

        index = PluginProcessMain.allocProcess(plugin, process);

    } catch (Throwable e) {

    }

    // 启动

    boolean rc = PluginProviderStub.proxyStartPluginProcess(mContext, in
dex);

    ......

    // 再次获取

    client = PluginProcessMain.probePluginClient(plugin, process, info);

    ......

    return client;}
```

PluginProviderStub.proxyStartPluginProcess 方法去会吊起对应进程，代码如下：

PluginProviderStub.java

```
static final boolean proxyStartPluginProcess(Context context, int index)
 {

        //

        ContentValues values = new ContentValues();

        values.put(KEY_METHOD, METHOD_START_PROCESS);

        values.put(KEY_COOKIE, PMF.sPluginMgr.mLocalCookie);

        Uri uri = context.getContentResolver().insert(ProcessPitProvider
Base.buildUri(index), values);

        if (LOG) {

            LogDebug.d(PLUGIN_TAG, "proxyStartPluginProcess insert.rc=" +
 (uri != null ? uri.toString() : "null"));
```

```
        }

        if (uri == null) {

            if (LOG) {

                LogDebug.d(PLUGIN_TAG, "proxyStartPluginProcess failed");

            }

            return false;

        }



        return true;

    }
```

这里是个对数据库 Provider 的插入操作，那么究竟是对哪个 Provider 进行了数据库操作呢？

关键在这里：

```
ProcessPitProviderBase.buildUri(index)
```

最终生成的 Uri 是这样的：

```
content://com.qihoo360.replugin.sample.host.loader.p.mainN99/main
```

这个玩意对应的啥呢？貌似宿主 Manifest 中并没有配置这个，如果分析过 gradle 插件应该可以想到，估计又是动态编译自动生成的，反编译宿主 apk 可以在 Manifest 中发现：

```
<provider

            android:name="com.qihoo360.replugin.component.process.Process
PitProviderP0"
```

```
            android:exported="false"

            android:process=":p0"

            android:authorities="com.qihoo360.replugin.sample.host.loader.
p.mainN100" />
```

这样就全部串起来啦，想要开启分配在自定义进程中的 Service 首先就需要通过启动动态编译自动生成的对应进程 Provider 来拉起进程。

最终返回的 IPluginClient 的 IBinder 实现体其实就是 PluginProcessPer，生成的 PSS 也就是 PluginServiceServer.mStub 了。跟上述常驻进程生成的 PSS 方式是一样的。

第 4 步

PSS 中 startService 最终调用的是 PluginServiceServer.this.startServiceLocked(intent,client)方法，其内部首先通过 retrieveServiceLocked 方法生成 ServiceRecord 对象，接着利用 installServiceIfNeededLocked 方法去加载 Service 对象，利用发射将 PluginContext 通过 attachBaseContextLocked 方法赋值到上下文内部，接着执行 Service 的 onCreate 方法，然后记录以及被加载，同时 startPitService 开启一个坑位 Service，提高进程优先级防止被杀，后面继续调用 onStartCommand 方法。

```
// 加载插件，获取 Service 对象，并将其缓存起来

    private boolean installServiceLocked(ServiceRecord sr) {

        // 通过 ServiceInfo 创建 Service 对象

        Context plgc = Factory.queryPluginContext(sr.plugin);

        if (plgc == null) {
```

```
            if (LogDebug.LOG) {

                Log.e(TAG, "installServiceLocked(): Fetch Context Error!
pn=" + sr.plugin);

            }

            return false;

        }

        ClassLoader cl = plgc.getClassLoader();

        if (cl == null) {

            if (LOGR) {

                LogRelease.e(PLUGIN_TAG, "psm.is: cl n " + sr.className);

            }

            return false;

        }



        // 构建 Service 对象

        Service s;

        try {

            s = (Service) cl.loadClass(sr.serviceInfo.name).newInstance
();

        } catch (Throwable e) {

            if (LOGR) {

                LogRelease.e(TAG, "isl: ni f " + sr.plugin, e);

            }

            return false;
```

```
        }


        // 只复写 Context，别的都不做
        try {
            attachBaseContextLocked(s, plgc);
        } catch (Throwable e) {
            if (LOGR) {
                LogRelease.e(PLUGIN_TAG, "psm.is: abc e", e);
            }
            return false;
        }
        s.onCreate();
        sr.service = s;


        // 开启“坑位”服务，防止进程被杀
        startPitService();
        return true;
    }
```

二、总结

针对插件中 startService 的 Service 对象，其实 Replugin 框架仅仅将此 Service 当做一个普通的类进行代码强制执行其对应的生命周期各回调方法，当然 Android FWK 启动 Service 证明周期也是类似的方式做的，只是原始启动 Service

BAT交流群：892872246

的各生命周期是在进程对应的 ActivityThread 中，而 Replugin 则是在 PSS 内部代码执行。这一点跟启动插件 Activity 是不太一样的。同时还有一个就是启动自定义进程的 Service 会利用动态编译自动生成的对应该进程的 Provider，首先调用该 Provider 的 insert 数据库操作去吊起此进程，然后再获得 IPluginClient 中的 PSS，显然，Replugin 框架的使用动态编译的地方无处不在。

第四节、so 文件加载机制

本文的结论是跟着 `System.loadlibrary()` 一层层源码走进去，个人对其的理解所整理的，那么开始看源码之前，先来提几个问题：

Q1：你知道 so 文件的加载流程吗？

Q2：设备存放 so 的路径有 system/lib，vendor/lib，system/lib64，vendor/lib64，知道在哪里规定了这些路径吗？清楚哪些场景下系统会去哪个目录下寻找 so 文件吗？还是说，所有的目录都会去寻找？

Q3：Zygote 进程是分 32 位和 64 位的，那么，系统是如何决定某个应用应该运行在 32 位上，还是 64 位上？

Q4：如果程序跑在 64 位的 Zygote 进程上时，可以使用 32 位的 so 文件么，即应用的 primaryCpuAbi 为 arm64-v8a，那么是否可使用 armeabi-v7a 的 so 文件，兼容的吗？

Q2，Q3，Q4，这几个问题都是基于设备支持 64 位的前提下，在旧系统版本中，只支持 32 位，也就没这么多疑问需要处理了。

12246

一、准备工作

由于这次的源码会涉及很多 framework 层的代码，包括 java 和 c++，直接在 AndroidStudio 跟进 SDK 的源码已不足够查看到相关的代码了。所以，此次是借助 Source Insight 软件，而源码来源如下：

https://android.googlesource.com/platform/

我并没有将所有目录下载下来，只下载了如下目录的源码：

- system/core
- bionic
- libcore
- dalvik
- frameworks/base
- frameworks/native

我没有下载最新版本的代码，而是选择了 Tags 下的 More 按钮，然后选择 tag 为： **android-5.1.1 r24** 的代码下载。所以，此次分析的源码是基于这个版本，其余不同版本的代码可能会有所不一样，但大体流程应该都是一致的。

二、分析

源码分析的过程很长很长，不想看过程的话，你也可以直接跳到末尾看结论，但就会错失很多细节的分析了。

那么下面就开始来过下源码吧，分析的入口就是跟着 `System.loadlibrary()` 走 ：

```
//System#loadlibrary()public static void loadLibrary(String libName) {
```

```
    Runtime.getRuntime().loadLibrary(libName, VMStack.getCallingClassLoa
der());}

//Runtime#loadLibrary()void loadLibrary(String libraryName, ClassLoader
loader) {

    //1\. 程序中通过 System.loadlibrary() 方式，这个 loader 就不会为空，流程
走这边

    if (loader != null) {

        //2\. loader.findLibrary() 这是个重点，这个方法用于寻找 so 文件是否存
在

        String filename = loader.findLibrary(libraryName);

        if (filename == null) {

            throw new UnsatisfiedLinkError(loader + " couldn't find \""
+ System.mapLibraryName(libraryName) + "\"");

        }

        //3\. 如果 so 文件找到，那么加载它

        String error = doLoad(filename, loader);

        if (error != null) {

            //4\. 如果加载失败，那么抛异常

            throw new UnsatisfiedLinkError(error);

        }

        return;

    }



    //1.1 以下代码的运行场景我不清楚，但有几个方法可以蛮看一下

    //mapLibraryName 用于拼接 so 文件名的前缀:lib，和后缀.so
```

```
String filename = System.mapLibraryName(libraryName);

//...省略

//1.2 mLibPaths 存储着设备存放 so 文件的目录地址

for (String directory: mLibPaths) {

    String candidate = directory + filename;

    candidates.add(candidate);

    if (IoUtils.canOpenReadOnly(candidate))

        // 1.3 调用 native 层方法加载 so 库

        String error = doLoad(candidate, loader);

        if (error == null) {

            return; // We successfully loaded the library. Job done.

        }

        lastError = error;

    }

}

//...省略}
```

所以，其实 System 的 `loadlibrary()` 是调用的 Runtime 的 `loadLibrary()`，不同系统版本，这些代码是有些许差别的，但不管怎样，重点都还是 `loadLibrary()` 中调用的一些方法，这些方法基本没变，改变的只是其他代码的优化写法。

那么，要理清 so 文件的加载流程，或者说，要找出系统是去哪些地址加载 so 文件的，就需要梳理清这些方法：

- `loader.findLibrary()`
- `doLoad()`

BAT交流群 892872246

第一个方法用于寻找 so 文件，所涉及的整个流程应该都在这个方法里，如果可以找到，会返回 so 文件的绝对路径，然后交由 `doLoad()` 去加载。

java.library.path

但在深入去探索之前，我想先探索另一条分支，loader 为空的场景。loader 什么时候为空，什么时候不为空，我并不清楚，只是看别人的文章分析时说，程序中通过 `System.loadlibrary()` 方式加载 so，那么 loader 就不会为空。那，我就信你了，不然我也不知道去哪分析为不为空的场景。

既然程序不会走另一个分支，为什么我还要先来探索它呢？因为，第一个分支太不好探索了，先从另一个分支摸索点经验，而且还发现了一些感觉可以拿来讲讲的方法：

- `System.mapLibraryName()`

用于拼接 so 文件名的前缀 `lib`，和后缀 `.so`。

- `mLibPaths`

在其他版本的源码中，可能就没有这个变量了，直接就是调用一个方法，但作用都一样，我们看看这个变量的赋值：

```
//Runtime.mLibPathsprivate final String[] mLibPaths = initLibPaths();

//Runtime#initLibPaths()private static String[] initLibPaths() {

    String javaLibraryPath = System.getProperty("java.library.path");

    //...省略}
```

最后都是通过调用 System 的 `getProperty()` 方法，读取 `java.library.path` 的属性值。

也就是说，通过读取 `java.library.path` 的系统属性值，是可以获取到设备存放 so 库的目录地址的，那么就来看看在哪里有设置这个属性值进去。

System 内部有一个类型为 Properties 的静态变量，不同版本，这个变量名可能不一样，但作用也都一样，用来存储这些系统属性值，这样程序需要的时候，调用 `getProperty()` 读取属性值时其实是来这个静态变量中读取。而变量的初始化地方在类中的 static 代码块中：

```
//Systemstatic {

    //...省略

    //1.初始化一些不变的系统属性值

    unchangeableSystemProperties = initUnchangeableSystemProperties();

    //2.将上述的属性值以及一些默认的系统属性值设置到静态变量中

    systemProperties = createSystemProperties();

    //...}

//System#initUnchangeableSystemProperties()private static Properties ini
tUnchangeableSystemProperties() {

    //...省略一些属性值设置

    p.put("java.vm.vendor", projectName);

    p.put("java.vm.version", runtime.vmVersion());

    p.put("file.separator", "/");

    p.put("line.separator", "\n");

    p.put("path.separator", ":");

    //...



    //1.这里是重点

    parsePropertyAssignments(p, specialProperties());
```

```
    //...

    return p;}

//System#createSystemProperties()private static Properties createSystemP
roperties() {

    //1.拷贝不可变的一些系统属性值

    Properties p = new PropertiesWithNonOverrideableDefaults(unchangeabl
eSystemProperties);

    //2.设置一些默认的属性值

    setDefaultChangeableProperties(p);

    return p;}

//System#setDefaultChangeableProperties()private static void setDefaultC
hangeableProperties(Properties p) {

    p.put("java.io.tmpdir", "/tmp");

    p.put("user.home", "");}
```

static 静态代码块中的代码其实就是在初始化系统属性值，分两个步骤，一个是先设置一些不可变的属性值，二是设置一些默认的属性值，然后将这些存储在静态变量中。

但其实，不管在哪个方法中，都没找到有设置 `java.library.path` 属性值的代码，那这个属性值到底是在哪里设置的呢？

关键点在于设置不可变的属性时，有调用了一个 native 层的方法：

```
//System/**

* Returns an array of "key=value" strings containing information not oth
erwise

* easily available, such as #defined library versions.
```

```
*/private static native String[] specialProperties();
```

这方法会返回 key=value 形式的字符串数组，然后 `parsePropertyAssignments()` 方法会去遍历这些数组，将这些属性值填充到存储系统属性值的静态变量中。

也就是说，在 native 层还会设置一些属性值，而 `java.library.path` 有可能就是在 native 中设置的，那么就跟下去看看吧。

System 连同包名的全名是：java.lang.System；那么，通常，所对应的 native 层的 cpp 文件名为：java_lang_System.cpp，到这里去看看：

```
//platform/libcore/luni/src/main/native/java_lang_System.cpp#System_spe
cialProperties()static jobjectArray System_specialProperties(JNIEnv* env,
 jclass) {

    std::vector<std::string> properties;



    //...



    //1\. 获取 LD_LIBRARY_PATH 环境变量值

    const char* library_path = getenv("LD_LIBRARY_PATH");#if defined(HAV
E_ANDROID_OS)

    if (library_path == NULL) {

        //2.如果 1 步骤没获取到路径，那么通过该方法获取 so 库的目录路径

        android_get_LD_LIBRARY_PATH(path, sizeof(path));

        library_path = path;

    }#endif

    if (library_path == NULL) {
```

```
        library_path = "";

    }

    //3.设置 java.library.path 属性值

    properties.push_back(std::string("java.library.path=") + library_path);



    return toStringArray(env, properties);}
```

没错吧，对应的 native 层的方法是上述这个，它干的事，其实也是设置一些属性值，我们想要的 `java.library.path` 就是在这里设置的。那么，这个属性值来源的逻辑是这样的：

1. 先读取 LD_LIBRARY_PATH 环境变量值，如果不为空，就以这个值为准。但我测试过，貌似，程序运行时读取的这个值一直是 null，在 Runtime 的 `doLoad()` 方法注释中，Google 有解释是说由于 Android 的进程都是通过 Zygote 进程 fork 过来，所以不能使用 LD_LIBRARY_PATH 。应该，大概，可能是这个意思吧，我英文不大好，你们可以自行去确认一下。
2. 也就是说，第一步读取的 LD_LIBRARY_PATH 值是为空，所以会进入第二步，调用 android_get_LD_LIBRARY_PATH 方法来读取属性值。（进入这个步骤有个条件是定义了 HAVE_ANDROID_OS 宏变量，我就不去找到底哪里在什么场景下会定义了，看命名我直接猜测 Android 系统就都有定义的了）

那么，继续看看 android_get_LD_LIBRARY_PATH 这个方法做了些什么：

```
//platform/libcore/luni/src/main/native/java_lang_System.cpp#if defined(HAVE_ANDROID_OS)extern "C" void android_get_LD_LIBRARY_PATH(char*, size_t);#endif
```

emmm，看不懂，头疼。那，直接全局搜索下这个方法名试试看吧，结果在另一个 cpp 中找到它的实现：

```
//platform/bionic/linker/dlfcn.cppvoid android_get_LD_LIBRARY_PATH(char*
 buffer, size_t buffer_size) {

  ScopedPthreadMutexLocker locker(&g_dl_mutex);

  do_android_get_LD_LIBRARY_PATH(buffer, buffer_size);}
```

第一行估计是加锁之类的意思吧，不管，第二行是调用另一个方法，继续跟下去看看：

```
//platform/bionic/linker/linker.cppvoid do_android_get_LD_LIBRARY_PATH(c
har* buffer, size_t buffer_size) {

  //...

  char* end = stpcpy(buffer, kDefaultLdPaths[0]);

  *end = ':';

  strcpy(end + 1, kDefaultLdPaths[1]);}

static const char* const kDefaultLdPaths[] = {#if defined(__LP64__)

  "/vendor/lib64",

  "/system/lib64",#else

  "/vendor/lib",

  "/system/lib",#endif

  nullptr};
```

还好 Source Insight 点击方法时有时可以支持直接跳转过去，调用的这个方法又是在另一个 cpp 文件中了。开头省略了一些大小空间校验的代码，然后直接复制了静态常量的值，而这个静态常量在这份文件顶部定义。

BAT文 /2246

终于跟到底了吧，也就是说，如果有定义了 **LP64** 这个宏变量，那么就将 `java.library.path` 属性值赋值为 "/vendor/lib64:/system/lib64"，否则，就赋值为 "/vendor/lib:/system/lib"。

也就是说，so 文件的目录地址其实是在 native 层通过硬编码方式写死的，网上那些理所当然的说 so 文件的存放目录也就是这四个，是这么来的。那么，说白了，系统默认存放 so 文件的目录就两个，只是有两种场景。

而至于到底什么场景下会有这个 **LP64** 宏变量的定义，什么时候没有，我实在没能力继续跟踪下去了，网上搜索了一些资料后，仍旧不是很懂，如果有清楚的大佬，能够告知、指点下就最棒了。

我自己看了些资料，以及，自己也做个测试：同一个 app，修改它的 primaryCpuAbi 值，调用 System 的 `getProperty()` 来读取 `java.library.path`，它返回的值是会不同的。

所以，以我目前的能力以及所掌握的知识，我是这么猜测的，纯属个人猜测：

LP64 这个宏变量并不是由安卓系统代码来定义的，而是 Linux 系统层面所定义的。在 Linux 系统中，可执行文件，也可以说所运行的程序，如果是 32 位的，那么是没有定义这个宏变量的，如果是 64 位的，那么是有定义这个宏变量的。

总之，通俗的联想解释，**LP64** 这个宏变量表示着当前程序是 32 位还是 64 位的意思。（个人理解）

有时间再继续研究吧，反正这里清楚了，系统默认存放 so 文件的目录只有两个，但有两种场景。vendor 较少用，就不每次都打出来了。也就是说，如果应用在 system/lib 目录中没有找到 so 文件，那么它是不会再自动去 system/lib64 中

寻找的，两者它只会选其一。至于选择哪个，因为 Zygote 是有分 32 位还是 64 位进程的，那么刚好可以根据这个为依据。

findLibrary

该走回主线了，在支线中的探索已经摸索了些经验了。

大伙应该还记得吧，System 调用了 `loadlibrary()` 之后，内部其实是调用了 Runtime 的 `loadLibrary()` 方法，这个方法内部会去调用 ClassLoader 的 `findLibrary()` 方法，主要是去寻找这个 so 文件是否存在，如果存在，会返回 so 文件的绝对路径，接着交由 Runtime 的 `doLoad()` 方法去加载 so 文件。

所以，我们想要梳理清楚 so 文件的加载流程，`findLibrary()` 是关键。那么，接下去，就来跟着 `findLibrary()` 走下去看看吧：

```
//ClassLoader#findLibrary()protected String findLibrary(String libName)
{

    return null;}
```

ClassLoader 只是一个基类，具体实现在其子类，那这里具体运行的是哪个子类呢？

```
//System#loadlibrary()public static void loadLibrary(String libName) {

    Runtime.getRuntime().loadLibrary(libName, VMStack.getCallingClassLoa
der());}
```

所以这里是调用了 VMStack 的一个方法来获取 ClassLoader 对象，那么继续跟进看看：

```
native public static ClassLoader getCallingClassLoader();
```

BAT交流群892872246

又是一个 native 的方法，我尝试过跟进去，没有看懂。那么，换个方向来找出这个基类的具体实现子类是哪个吧，很简单的一个方法，打 log 输出这个对象本身：

```
ClassLoader classLoader = getClassLoader();Log.v(TAG, "classLoader = " +
 classLoader.toString());//输出// classLoader = dalvik.system.PathClassL
oader[dexPath=/data/app/com.qrcode.qrcode-1.apk,libraryPath=/data/app-l
ib/com.qrcode.qrcode-1]
```

以上打 Log 代码是从 Java 中 System.loadLibrary() 的执行过程 这篇文章中截取出来的，使用这个方法的前提是你得清楚 VMStack 的 `getCallingClassLoader()` 含义其实是获取调用这个方法的类它的类加载器对象。

或者，你对 Android 的类加载机制有所了解，知道当启动某个 app 时，经过层层工作后，会接着让 LoadedApk 去加载这个 app 的 apk，然后通过 ApplicationLoader 来加载相关代码文件，而这个类内部是实例化了一个 PathClassLoader 对象去进行 dex 的加载。

不管哪种方式，总之清楚了这里实际上是调用了 PathClassLoader 的 `findLibrary()` 方法，但 PathClassLoader 内部并没有这个方法，它继承自 BaseDexClassLoader，所以实际上还是调用了父类的方法，跟进去看看：

```
//platform/libcore/dalvik/src/main/java/dalvik/system/BaseDexClassLoade
r.javapublic String findLibrary(String name) {

    return pathList.findLibrary(name);}

private final DexPathList pathList;
```

内部又调用了 DexPathList 的 `findLibrary()` 方法，继续跟进看看：

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
//platform/libcore/dalvik/src/main/java/dalvik/system/DexPathList.javap
ublic String findLibrary(String libraryName) {

    //1\. 拼接前缀：lib，和后缀：.so

    String fileName = System.mapLibraryName(libraryName);

    //2\. 遍历所有存放 so 文件的目录，确认指定文件是否存在以及是只读文件

    for (File directory: nativeLibraryDirectories) {

        String path = new File(directory, fileName).getPath();

        if (IoUtils.canOpenReadOnly(path)) {

            return path;

        }

    }

    return null;}

/** List of native library directories. */private final File[] nativeLib
raryDirectories;
```

到了这里，会先进行文件名补全操作，拼接上前缀：lib 和后缀：.so，然后遍历所有存放 so 文件的目录，当找到指定文件，且是只读属性，则返回该 so 文件的绝对路径。

所以，重点就是 nativeLibraryDirectories 这个变量了，这里存放着 so 文件存储的目录路径，那么得看看它在哪里被赋值了：

```
//platform/libcore/dalvik/src/main/java/dalvik/system/DexPathList.javap
ublic DexPathList(ClassLoader definingContext, String dexPath, String li
braryPath, File optimizedDirectory) {

    //...

    //1\. 唯一赋值的地方，构造函数

    this.nativeLibraryDirectories = splitLibraryPath(libraryPath);}
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
private static File[] splitLibraryPath(String path) {

    // Native libraries may exist in both the system and

    // application library paths, and we use this search order:

    //

    //   1\. this class loader's library path for application libraries

    //   2\. the VM's library path from the system property for system li
braries

    //   (翻译下，大体是说，so 文件的来源有两处：1 是应用自身存放 so 文件的目录，
2 是系统指定的目录)

    // This order was reversed prior to Gingerbread; see http://b/293345
6.

    ArrayList < File > result = splitPaths(path, System.getProperty("jav
a.library.path"), true);

    return result.toArray(new File[result.size()]);}

//将传入的两个参数的目录地址解析完都存放到集合中 private static ArrayList < F
ile > splitPaths(String path1, String path2, boolean wantDirectories) {

    ArrayList < File > result = new ArrayList < File > ();



    splitAndAdd(path1, wantDirectories, result);

    splitAndAdd(path2, wantDirectories, result);

    return result;}

private static void splitAndAdd(String searchPath, boolean directoriesOn
ly, ArrayList < File > resultList) {

    if (searchPath == null) {

        return;

    }
```

```
    //因为获取系统的 java.library.path 属性值返回的路径是通过 : 拼接的,所以先拆分，然后判断这些目录是否可用

    for (String path: searchPath.split(":")) {

        try {

            StructStat sb = Libcore.os.stat(path);

            if (!directoriesOnly || S_ISDIR(sb.st_mode)) {

                resultList.add(new File(path));

            }

        } catch(ErrnoException ignored) {}

    }}
```

所以，nativeLibraryDirectories 这个变量是在构造函数中被赋值。代码不多，总结一下，构造函数会传入一个 libraryPath 参数，表示应用自身存放 so 文件的路径，然后内部会再去调用 System 的 `getProperty("java.library.path")` 方法获取系统指定的 so 文件目录地址。最后，将这些路径都添加到集合中。

而且，看添加的顺序，是先添加应用自身的 so 文件目录，然后再添加系统指定的 so 文件目录，也就是说，当加载 so 文件时，是先去应用自身的 so 文件目录地址寻找，没有找到，才会去系统指定的目录。

而系统指定的目录地址在 native 层的 linker.cpp 文件定义，分两种场景，取决于应用当前的进程是 32 位还是 64 位，32 位的话，则按顺序分别去 vendor/lib 和 system/lib 目录中寻找，64 位则是相对应的 lib64 目录中。

虽然，so 文件加载流程大体清楚了，但还有两个疑问点：

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

- 构造方法参数传入的表示应用自身存放 so 文件目录的 libraryPath 值是哪里来的；
- 应用什么时候运行在 32 位或 64 位的进程上；

nativeLibraryDir

先看第一个疑问点，应用自身存放 so 文件目录的这个值，要追究的话，这是一个很漫长的故事。

这个过程，我不打算全部都贴代码了，因为很多步骤，我自己也没有去看源码，也是看的别人的文章，我们以倒着追踪的方式来进行追溯吧。

首先，这个 libraryPath 值是通过 DexPathList 的构造方法传入的，而 BaseDexClassLoader 内部的 DexPathList 对象实例化的地方也是在它自己的构造方法中，同样，它也接收一个 libraryPath 参数值，所以 BaseDexClassLoader 只是做转发，来源并不在它这里。

那么，再往回走，就是 LoadedApk 实例化 PathClassLoader 对象的地方了，在它的 `getClassLoader()` 方法中：

```
//platform/frameworks/base/core/java/android/app/LoadedApk.javapublic ClassLoader getClassLoader() {

    synchronized(this) {

        //...

        final ArrayList < String > libPaths = new ArrayList < >();

        //...

        libPaths.add(mLibDir);

        //...

        final String lib = TextUtils.join(File.pathSeparator, libPaths);
```

892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
        //...

        mClassLoader = ApplicationLoaders.getDefault().getClassLoader(zi
p, lib, mBaseClassLoader);

        //...

    }}

public LoadedApk(ActivityThread activityThread, ApplicationInfo aInfo, C
ompatibilityInfo compatInfo, ClassLoader baseLoader, boolean securityVio
lation, boolean includeCode, boolean registerPackage) {

   //...

    mLibDir = aInfo.nativeLibraryDir;

   //...}
```

无关代码都省略掉了，也就是说，传给 DexPathList 的 libraryPath 值，其实是将要启动的这个 app 的 ApplicationInfo 中的 nativeLibraryDir 变量值。

可以看看 ApplicationInfo 中这个变量的注释：

```
//ApplicationInfo /**

* Full path to the directory where native JNI libraries are stored.

* 存放 so 文件的绝对路径

*/

public String nativeLibraryDir;
```

通俗点解释也就是，存放应用自身 so 文件的目录的绝对路径。那么问题又来了，传给 LoadedApk 的这个 ApplicationInfo 对象哪里来的呢？

这个就又涉及到应用的启动流程了，大概讲一下：

BAT文 ,12246

我们知道，当要启动其他应用时，其实是通过发送一个 Intent 去启动这个 app 的 LAUNCHER 标志的 Activity。而当这个 Intent 发送出去后，是通过 Binder 通信方式通知了 ActivityManagerServer 去启动这个 Activity。

AMS 在这个过程中会做很多事，但在所有事之前，它得先解析 Intent，知道要启动的是哪个 app 才能继续接下去的工作，这个工作在 ActivityStackSupervisor 的 `resolveActivity()`：

```
//ActivityStackSupervisor.javaActivityInfo resolveActivity(Intent intent,
 String resolvedType, int startFlags, ProfilerInfo profilerInfo, int use
rId) {

    // Collect information about the target of the Intent.

    ActivityInfo aInfo;

    try {

        ResolveInfo rInfo = AppGlobals.getPackageManager().resolveIntent
(intent, resolvedType, PackageManager.MATCH_DEFAULT_ONLY | ActivityManag
erService.STOCK_PM_FLAGS, userId);

        aInfo = rInfo != null ? rInfo.activityInfo: null;

    } catch(RemoteException e) {

        aInfo = null;

    }

    //...}
```

不同版本，可能不是由这个类负责这个工作了，但可以跟着 ActivityManagerService 的 `startActivity()` 走下去看看，不用跟很深就能找到，因为这个工作是比较早进行的。

BAT交流 892246

所以，解析 Intent 获取 app 的相关信息就又交给 PackageManagerService 的 `resolveIntent()` 进行了，PKMS 的工作不贴了，我直接说了吧：

PKMS 会根据 Intent 中目标组件的 packageName，通过一个只有包权限的类 Settings 来获取对应的 ApplicationInfo 信息，这个 Settings 类全名：com.android.server.pm.Settings，它的职责之一是存储所有 app 的基本信息，也就是在 data/system/packages.xml 中各 app 的信息都交由它维护缓存。

所以，一个 app 的 ApplicationInfo 信息，包括 nativeLibraryDir 我们都可以在 data/system/packages.xml 这份文件中查看到。这份文件的角色我把它理解成类似 PC 端上的注册表，所有 app 的信息都注册在这里。

那这份 packages.xml 文件的数据又是从哪里来的呢，这又得涉及到 apk 的安装机制过程了。

简单说一下，一个 app 的安装过程，在解析 apk 包过程中，还会结合各种设备因素等等来决定这个 app 的各种属性，比如说 nativeLibraryDir 这个属性值的确认，就需要考虑这个 app 是三方应用还是系统应用，这个应用的 primaryCpuAbi 属性值是什么，apk 文件的地址等等因素后，最后才确定了应用存放 so 文件的目录地址是哪里。

举个例子，对于系统应用来说，这个 nativeLibraryDir 值有可能最后是 /system/lib/xxx，也有可能是 system/app/xxx/lib 等等；而对于三方应用来说，这值有可能就是 data/app/xxx/lib；

BAT交流群/2246

也就是说，当 app 安装完成时，这些属性值也就都解析到了，就都会保存到 Settings 中，同时会将这些信息写入到 data/system/packages.xml 中。

到这里，先来小结一下，梳理下前面的内容：

当一个 app 安装的时候，系统会经过各种因素考量，最后确认 app 的一个 nativeLibraryDir 属性值，这个属性值代表应用自身的 so 文件存放地址，这个值也可以在 data/system/packages.xml 中查看。

当应用调用了 System 的 `loadlibrary()` 时，这个 so 文件的加载流程如下：

1. 先到 nativeLibraryDir 指向的目录地址中寻找这个 so 文件是否存在、可用；
2. 如果没找到，那么根据应用进程是 32 位或者 64 位来决定下去应该去哪个目录寻找 so 文件；
3. 如果是 32 位，则先去 vendor/lib 找，最后再去 system/lib 中寻找；
4. 如果是 64 位，则先去 vendor/lib64 找，最后再去 system/lib64 中寻找；
5. 系统默认的目录是在 native 层中的 linker.cpp 文件中指定，更严谨的说法，不是进程是不是 32 位或 64 位，而是是否有定义了 **LP64** 这个宏变量。

primaryCpuAbi

我们已经清楚了，加载 so 文件的流程，其实就分两步，先去应用自身存放 so 文件的目录（nativeLibraryDir）寻找，找不到，再去系统指定的目录中寻找。

BAT交流 892872246

而系统指定是目录分两种场景，应用进程是 32 位或者 64 位，那么，怎么知道应用是运行在 32 位还是 64 位的呢？又或者说，以什么为依据来决定一个应用是应该跑在 32 位上还是跑在 64 位上？

这个就取决于一个重要的属性了 primaryCpuAbi，它代表着这个应用的 so 文件使用的是哪个 abi 架构。

abi 常见的如：arm64-v8a，armeabi-v7a，armeabi，mips，x86_64 等等。

我们在打包 apk 时，如果不指定，其实默认是会将所有 abi 对应的 so 文件都打包一份，而通常，为了减少 apk 包体积，我们在 build.gradle 脚本中会指定只打其中一两份。但不管 apk 包有多少种不同的 abi 的 so 文件，在 app 安装过程中，最终拷贝到 nativeLibraryDir 中的通常都只有一份，除非你手动指定了要多份。

那么，app 在安装过程中，怎么知道，应该拷贝 apk 中的 lib 下的哪一份 so 文件呢？这就是由应用的 primaryCpuAbi 属性决定。

而同样，这个属性一样是在 app 安装过程中确定的，这个过程更加复杂，末尾有给了篇链接，感兴趣可以去看看，大概来说，就是 apk 包中的 so 文件、系统应用、相同 UID 的应用、设备的 abilist 等都对这个属性值的确定过程有所影响。同样，这个属性值也可以在 data/system/packages.xml 中查看。

,12246

那么，这个 primaryCpuAbi 属性值是如何影响应用进程是 32 位还是 64 位的呢？

这就涉及到 Zygote 方面的知识了。

在系统启动之后，系统会根据设备的 ro.zygote 属性值决定启动哪个 Zygote，可以通过执行 `getprop | grep ro.zygote` 来查看这个属性值，属性值与对应的 Zygote 进程关系如下：

- zygote32：只启动一个 32 位的 Zygote 进程
- zygote32_64：启动两个 Zygote 进程，分别为 32 位和 64 位，32 位的进程名为 zygote，表示以它为主，64 位进程名为 zygote_secondary ，表示它作为辅助
- zygote64：只启动一个 64 位的 Zygote 进程
- zygote64_32：启动两个 Zygote 进程，分别为 32 位和 64 位，64 位的进程名为 zygote，表示以它为主，32 位进程名为 zygote_secondary ，表示它作为辅助

而 Zygote 进程启动之后，会打开一个 socket 端口，等待 AMS 发消息过来启动新的应用时 fork 当前 Zygote 进程，所以，如果 AMS 是发给 64 位的 Zygote，那么新的应用自然就是跑在 64 位的进程上；同理，如果发给了 32 位的 Zygote 进程，那么 fork 出来的进程自然也就是 32 位的。

那么，可以跟随着 AMS 的 `startProcessLocked()` 方法，去看看是以什么为依据选择 32 位或 64 位的 Zygote：

```
//ActivityManagerService

private final void startProcessLocked(ProcessRecord app, String hostingT
ype, String hostingNameStr, String abiOverride, String entryPoint, Strin
g[] entryPointArgs) {

    //...省略
```

```
    //1\. 获取要启动的 app 的 primaryCpuAbi 属性值，abiOverride 不知道是什么，可能是 Google 开发人员写测试用例用的吧，或者其他一些场景

    String requiredAbi = (abiOverride != null) ? abiOverride: app.info.primaryCpuAbi;

    if (requiredAbi == null) {

        //2\. 如果为空，以设备支持的首个 abi 属性值，可执行 getprot ro.product.cpu.abilist 查看

        requiredAbi = Build.SUPPORTED_ABIS[0];

    }

    //...



    //3\. 调用 Precess 的 start 方法，将 requiredAbi 传入

    Process.ProcessStartResult startResult = Process.start(entryPoint, app.processName, uid, uid, gids, debugFlags, mountExternal, app.info.targetSdkVersion, app.info.seinfo, requiredAbi, instructionSet, app.info.dataDir, entryPointArgs);

    //...}
```

AMS 会先获取要启动的 app 的 primaryCpuAbi 属性值，至于这个 app 的相关信息怎么来的，跟上一小节一样，解析 Intent 时交由 PKMS 去它模块内部的 Settings 读取的。

如果 primaryCpuAbi 为空，则以设备支持的首个 abi 属性值为主，设备支持的 abi 列表可以通过执行 `getprot ro.product.cpu.abilist` 查看，最后调用 Precess 的 `start()` 方法，将读取的 abi 值传入：

```
//Process
```

BAT文 ,/2246

```
public static final ProcessStartResult start(final String processClass,
final String niceName, int uid, int gid, int[] gids, int debugFlags, int
mountExternal, int targetSdkVersion, String seInfo, String abi, String i
nstructionSet, String appDataDir, String[] zygoteArgs) {

    //...

    return startViaZygote(processClass, niceName, uid, gid, gids, debugF
lags, mountExternal, targetSdkVersion, seInfo, abi, instructionSet, appD
ataDir, zygoteArgs);

    //...}



private static ProcessStartResult startViaZygote(final String processCla
ss, final String niceName, final int uid, final int gid, final int[] gids,
 int debugFlags, int mountExternal, int targetSdkVersion, String seInfo,
 String abi, String instructionSet, String appDataDir, String[] extraArg
s) throws ZygoteStartFailedEx {

    //...

    //所以 abi 最终是调用 openZygoteSocketIfNeeded() 方法，传入给它使用

    return zygoteSendArgsAndGetResult(openZygoteSocketIfNeeded(abi), arg
sForZygote);}
```

abi 值又是一层传一层，最终交到了 Process 的 `openZygoteSocketIfNeeded()` 方法中使用，跟进看看：

```
//Processprivate static ZygoteState openZygoteSocketIfNeeded(String abi)
 throws ZygoteStartFailedEx {

    if (primaryZygoteState == null || primaryZygoteState.isClosed()) {

        try {

            //ZYGOTE_SOCKET 值为 zygote，

            //通过 ZygoteState 的 connect 方法，连接进程名为 zygote 的 Zygot
e 进程

            primaryZygoteState = ZygoteState.connect(ZYGOTE_SOCKET);
```

```
        } catch(IOException ioe) {

            throw new ZygoteStartFailedEx("Error connecting to primary zy
gote", ioe);

        }

    }

    //在进程名为 zygote 的 Zygote 进程支持的 abi 列表中，查看是否支持要启动的
app 的需要的 abi

    if (primaryZygoteState.matches(abi)) {

        return primaryZygoteState;

    }



    // The primary zygote didn't match. Try the secondary.

    if (secondaryZygoteState == null || secondaryZygoteState.isClosed())
 {

        try {

             //SECONDARY_ZYGOTE_SOCKET 的值为 zygote_secondary,

            //通过 ZygoteState 的 connect 方法，连接进程名为 zygote_secondar
y 的 Zygote 进程

            secondaryZygoteState = ZygoteState.connect(SECONDARY_ZYGOTE_S
OCKET);

        } catch(IOException ioe) {

            throw new ZygoteStartFailedEx("Error connecting to secondary
zygote", ioe);

        }

    }
```

BAT文 892246

```
    //在进程名为 zygote_secondary 的 Zygote 进程支持的 abi 列表中，查看是否支持要启动的 app 的需要的 abi

    if (secondaryZygoteState.matches(abi)) {

        return secondaryZygoteState;

    }



    throw new ZygoteStartFailedEx("Unsupported zygote ABI: " + abi);}

static ZygoteState primaryZygoteState;static ZygoteState secondaryZygoteState;public static final String ZYGOTE_SOCKET = "zygote";public static final String SECONDARY_ZYGOTE_SOCKET = "zygote_secondary";
```

到了这里，是先获取进程名 zygote 的 Zygote 进程，查看它支持的 abi 列表中是否满足要启动的 app 所需的 abi，如果满足，则使用这个 Zygote 来 fork 新进程，否则，获取另一个进程名为 zygote_secondary 的 Zygote 进程，同样查看它支持的 abi 列表中是否满足 app 所需的 abi，如果都不满足，抛异常。

那么，名为 zygote 和 zygote_secondary 分别对应的是哪个 Zygote 进程呢？哪个对应 32 位，哪个对应 64 位？

还记得上述说过的，系统启动后，会去根据设备的 ro.zygote 属性决定启动哪个 Zygote 进程吗？对应关系就是这个属性值决定的，举个例子，可以看看 zygote64_32 对应的 Zygote 启动配置文件：

```
//platform/system/core/rootdir/init.zygote64_32.rc

service zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --start-system-server --socket-name=zygote

    class main
```

```
    socket zygote stream 660 root system

    onrestart write /sys/android_power/request_state wake

    onrestart write /sys/power/state on

    onrestart restart media

    onrestart restart netd



service zygote_secondary /system/bin/app_process32 -Xzygote /system/bin
--zygote --socket-name=zygote_secondary

    class main

    socket zygote_secondary stream 660 root system

    onrestart restart zygote
```

这份代码前半段的意思就表示，让 Linux 启动一个 service，进程名为 zygote，可执行文件位于 /system/bin/app_process64，后面是参数以及其他命令。

所以，名为 zygote 和 zygote_secondary 分别对应的是哪个 Zygote 进程，就取决于设备的 ro.zygote 属性。

而，获取 Zygote 支持的 abi 列表是通过 ZygoteState 的 `connect()` 方法，我们继续跟进看看：

```
//Process$ZygoteStatepublic static ZygoteState connect(String socketAddr
ess) throws IOException {

    //...



    String abiListString = getAbiList(zygoteWriter, zygoteInputStream);
```

BAT文 892246

```
    Log.i("Zygote", "Process: zygote socket opened, supported ABIS: " + a
biListString);



    return new ZygoteState(zygoteSocket, zygoteInputStream, zygoteWriter,
 Arrays.asList(abiListString.split(",")));}
```

发现没有，源码内部将 Zygote 支持的 abi 列表输出日志了，你们可以自己尝试下，过滤下 TAG 为 Zygote，然后重启下设备，因为如果本来就连着 Zygote，那么是不会走到这里的了，最后看一下相关日志，如：

```
01-01 08:00:13.509 2818-2818/? D/AndroidRuntime: >>>>>> START com.androi
d.internal.os.ZygoteInit uid 0 <<<<<<01-01 08:00:15.068 2818-2818/? D/Zy
gote: begin preload01-01 08:00:15.081 2818-3096/? I/Zygote: Preloading c
lasses...01-01 08:00:15.409 2818-3097/? I/Zygote: Preloading resources...
01-01 08:00:16.637 2818-3097/? I/Zygote: ...preloaded 343 resources in 1
228ms.01-01 08:00:16.669 2818-3097/? I/Zygote: ...preloaded 41 resources
 in 33ms.01-01 08:00:17.242 2818-3096/? I/Zygote: ...preloaded 3005 clas
ses in 2161ms.01-01 08:00:17.373 2818-2818/? I/Zygote: Preloading shared
 libraries...01-01 08:00:17.389 2818-2818/? D/Zygote: end preload01-01 0
8:00:17.492 2818-2818/? I/Zygote: System server process 3102 has been cr
eated01-01 08:00:17.495 2818-2818/? I/Zygote: Accepting command socket c
onnections01-01 08:00:32.789 3102-3121/? I/Zygote: Process: zygote socke
t opened, supported ABIS: armeabi-v7a,armeabi
```

系统启动后，Zygote 工作的相关内容基本都打日志出来了。

最后，再来稍微理一理：

app 安装过程，会确定 app 的一个属性值：primaryCpuAbi，它代表着这个应用的 so 文件使用的是哪个 abi 架构，而且它的确定过程很复杂，apk 包中的 so 文件、系统应用、相同 UID 的应用、设备的 abilist 等都对这个属性值的确定过程有所影响。安装成功后，可以在 data/system/packages.xml 中查看这个属性值。

BAT交流 892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

每启动一个新的应用，都是运行在新的进程中，而新的进程是从 Zygote 进程 fork 过来的，系统在启动时，会根据设备的 ro.zygote 属性值决定启动哪几个 Zygote 进程，然后打开 socket，等待 AMS 发送消息来 fork 新进程。

当系统要启动一个新的应用时，AMS 在负责这个工作进行到 Process 类的工作时，会先尝试在进程名为 zygote 的 Zygote 进程中，查看它所支持的 abi 列表中是否满足要启动的 app 所需的 abi，如果满足，则以这个 Zygote 为主，fork 新进程，运行在 32 位还是 64 位就跟这个 Zygote 进程一致，而 Zygote 运行在几位上取决于 ro.zygote 对应的文件，如值为 zygote64_32 时，对应着 init.zygote64_32.rc 这份文件，那么此时名为 zygote 的 Zygote 就是运行在 64 位上的。

而当上述所找的 Zygote 支持的 abi 列表不满足 app 所需的 abi 时，那么再去名为 zygote_secondary 的 Zygote 进程中看看，它所支持的 abi 列表是否满足。

另外，Zygote 的相关工作流程，包括支持的 abi 列表，系统都有打印相关日志，可过滤 Zygote 查看，如没发现，可重启设备查看。

abi 兼容

so 文件加载的流程，及应用运行在 32 位或 64 位的依据我们都梳理完了，以上内容足够掌握什么场景下，该去哪些目录下加载 so 文件的判断能力了。

,12246

那么，还有个问题，如果应用运行在 64 位上，那么此时，它是否能够使用 armeabi-v7a 的 so 文件？

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

首先，先来罗列一下常见的 abi ：

- arm64-v8a，armeabi-v7a，armeabi，mips，mips64，x86，x86_64

其中，运行在 64 位的 Zygote 进程上的是：

- arm64-v8a，mips64，x86_64

同样，运行在 32 位的 Zygote 进程上的是：

- armeabi-v7a，armeabi，mips，x86

你们如果去网上搜索如下关键字：so 文件，abi 兼容等，你们会发现，蛮多文章里都会说：arm64-v8a 的设备能够向下兼容，支持运行 32 位的 so 文件，如 armeabi-v7a。

这句话没错，64 位的设备能够兼容运行 32 位的 so 文件，但别只看到这句话啊，良心一些的文章里还有另一句话：**不同 cpu 架构的 so 文件不能够混合使用**，例如，程序运行期间，要么全部使用 arm64-v8a 的 so 文件，要么全部使用 armeabi-v7a 的 so 文件，你不能跑在 64 位进程上，却使用着 32 位的 so 文件。

我所理解的兼容，并不是说，64 位的设备，支持你运行在 64 位的 Zygote 进程上时仍旧可以使用 32 位的 so 文件。有些文章里也说了，如果在 64 位的设备上，你选择使用 32 位的 so 文件，那么此时，你就丢失了专门为 64 位优化过的性能（ART，webview，media 等等 ）。这个意思就是说，程序启动时

BAT文 ,/2246

是从 32 位的 Zygote 进程 fork 过来的，等于你在 64 位的设备上，但却只运行在 32 位的进程上。

至于程序如何决定运行在 32 位还是 64 位，上面的章节中也分析过了，以 app 的 primaryCpuAbi 属性值为主，而这个属性值的确定因素之一就是含有的 so 文件所属的 abi。

如果，你还想自己验证，那么可以跟着 Runtime 的 `doLoad()` 方法跟到 native 层去看看，由于我下载的源码版本可能有些问题，我没找到 Runtime 对应的 cpp 文件，但我找到这么段代码：

```
//platform/bionic/linker/linker_phdr.cppbool ElfReader::VerifyElfHeader()
 {

  //...

  //1.读取 elf 文件的 header 的 class 信息

  int elf_class = header_.e_ident[EI_CLASS];#if defined(__LP64__)

  //2\. 如果当前进程是 64 位的，而 elf 文件属于 32 位的，则报错

  if (elf_class != ELFCLASS64) {

    if (elf_class == ELFCLASS32) {

      DL_ERR("\"%s\" is 32-bit instead of 64-bit", name_);

    } else {

      DL_ERR("\"%s\" has unknown ELF class: %d", name_, elf_class);

    }

    return false;

  }#else
```

```
  //3\. 如果当前进程是 32 位的，而 elf 文件属于 64 位的，则报错
 if (elf_class != ELFCLASS32) {
   if (elf_class == ELFCLASS64) {
     DL_ERR("\"%s\" is 64-bit instead of 32-bit", name_);
   } else {
     DL_ERR("\"%s\" has unknown ELF class: %d", name_, elf_class);
   }
   return false;
 }#endif
```

加载 so 文件，最终还是交由 native 层去加载，在 Linux 中，so 文件其实就是一个 elf 文件，elf 文件有个 header 头部信息，里面记录着这份文件的一些信息，如所属的是 32 位还是 64 位，abi 的信息等等。

而 native 层在加载 so 文件之前，会去解析这个 header 信息，当发现，如果当前进程运行在 64 位时，但要加载的 so 文件却是 32 位的，就会报 `xxx is 32-bit instead of 64-bit` 异常，同样，如果当前进程是运行在 32 位的，但 so 文件却是 64 位的，此时报 `xxx is 64-bit instead of 32-bit` 异常。

这个异常应该也有碰见过吧：

```
java.lang.UnsatisfiedLinkError: dlopen failed: "libimagepipeline.so" is
32-bit instead of 64-bit
```

所以说，64 位设备的兼容，并不是说，允许你运行在 64 位的进程上时，仍旧可以使用 32 位的 so 文件。它的兼容是说，允许你在 64 位的设备上运行 32 位的进程。

其实，想想也能明白，这就是为什么三方应用安装的时候，并不会将 apk 包中所有 abi 目录下的 so 文件都解压出来，只会解压一种，因为应用在安装过程中，系统已经确定你这个应用是应该运行在 64 位还是 32 位的进程上了，并将这个结果保存在 app 的 primaryCpuAbi 属性值中。

既然系统已经明确你的应用所运行的进程是 32 位还是 64 位，那么只需拷贝对应的一份 so 文件即可，毕竟 64 位的 so 文件和 32 位的又不能混合使用。

以上，是我的理解，如果有误，欢迎指点下。

三、总结

整篇梳理下来，虽然梳理 so 的加载流程不难，但要掌握知其所以然的程度，就需要多花费一点心思了。

毕竟都涉及到应用的安装机制，应用启动流程，系统启动机制，Zygote 相关的知识点了。如果你是开发系统应用的，建议还是花时间整篇看一下，毕竟系统应用的集成不像三方应用那样在 apk 安装期间自动将相关 so 文件解压到 nativeLibraryDirectories 路径下了。三方应用很少需要了解 so 的加载流程，但开发系统应用还是清楚点比较好。

不管怎么说，有时间，可以稍微跟着过一下整篇，相信多少是会有些收获的，如果发现哪里有误，也欢迎指点。没时间的话，那就看看总结吧。

- 一个应用在安装过程中，系统会经过一系列复杂的逻辑确定两个跟 so 文件加载相关的 app 属性值：nativeLibraryDirectories ，primaryCpuAbi ；
- nativeLibraryDirectories 表示应用自身存放 so 文件的目录地址，影响着 so 文件的加载流程；

BAT交流群：892872246

- primaryCpuAbi 表示应用应该运行在哪种 abi 上，如（armeabi-v7a），它影响着应用是运行在 32 位还是 64 位的进程上，进而影响到寻找系统指定的 so 文件目录的流程；
- 以上两个属性，在应用安装结束后，可在 data/system/packages.xml 中查看；
- 当调用 System 的 `loadLibrary()` 加载 so 文件时，流程如下：
- 先到 nativeLibraryDirectories 指向的目录中寻找，是否存在且可用的 so 文件，有则直接加载这里的 so 文件；
- 上一步没找到的话，则根据当前进程如果是 32 位的，那么依次去 vendor/lib 和 system/lib 目录中寻找；
- 同样，如果当前进程是 64 位的，那么依次去 vendor/lib64 和 system/lib64 目录中寻找；
- 当前应用是运行在 32 位还是 64 位的进程上，取决于系统的 ro.zygote 属性和应用的 primaryCpuAbi 属性值，系统的 ro.zygote 可通过执行 getprop 命令查看；
- 如果 ro.zygote 属性为 zygote64_32，那么应用启动时，会先在 ro.product.cpu.abilist64 列表中寻找是否支持 primaryCpuAbi 属性，有，则该应用运行在 64 位的进程上；
- 如果上一步不支持，那么会在 ro.product.cpu.abilist32 列表中寻找是否支持 primaryCpuAbi 属性，有，则该应用运行在 32 位的进程上；
- 如果 ro.zygote 属性为 zygote32_64，则上述两个步骤互换；

BAT交流群：892872246

- 如果应用的 primaryCpuAbi 属性为空，那么以 ro.product.cpu.abilist 列表中第一个 abi 值作为应用的 primaryCpuAbi；
- 运行在 64 位的 abi 有：arm64-v8a，mips64，x86_64
- 运行在 32 位的 abi 有：armeabi-v7a，armeabi，mips，x86
- 通常支持 arm64-v8a 的 64 位设备，都会向下兼容支持 32 位的 abi 运行；
- 但应用运行期间，不能混合着使用不同 abi 的 so 文件；
- 比如，当应用运行在 64 位进程中时，无法使用 32 位 abi 的 so 文件，同样，应用运行在 32 位进程中时，也无法使用 64 位 abi 的 so 文件；

第五节、Android 系统服务实现原理

一、系统服务提供方式

1、我们平时最常见的系统服务使用方式

```
 WifiManager wifimanager = (WifiManager) getSystemService(Context.WIFI_S
ERVICE);

 wifimanager.disconnect();
```

这里我们通过 Context 的 getSystemService(String name)方法，根据 Context.WIFI_SERVICE 标识获取到了 WifiManager，然后调用 disconnerct()这个 api 实现了下发 wifi 断开指令的功能，这里就是一个很经典的系统服务使用的例子了，getSystemService()通过我们传入的字符串标识返回我们需要的具体 WifiManager 实例，那么我们不妨从 getSystemService()入手，看看 google 的工程师是如何做的。

2、getSystemService()方法

源码路径:xxx/framework/base/core/java/android/content/Context.java

```
public abstract Object getSystemService(@ServiceName @NonNull String name);
```

Context 中的 getSystemService()方法是一个抽象方法，意料之中，他的具体实现是在 ContextImpl 中:

源码路径:xxx/framework/base/core/java/android/app/ContextImpl.java

```
@Override
public Object getSystemService(String name) {
    return SystemServiceRegistry.getSystemService(this, name);
}
```

这里要注意，在 5.0 之前是在 ContextImpl 中实现的，而 5.0 之后就去调 SystemServiceRegistry 中的 getSystemService()方法:

源码路径:xxx/framework/base/core/java/android/app/SystemServiceRegistry.java

```
/**
 * Gets a system service from a given context.
 */
public static Object getSystemService(ContextImpl ctx, String name) {
    ServiceFetcher<?> fetcher = SYSTEM_SERVICE_FETCHERS.get(name);
    return fetcher != null ? fetcher.getService(ctx) : null;
```

```
}
```

这里我们看到，通过我们传入的标识 Context.WIFI_SERVICE 从 SYSTEM_SERVICE_FETCHERS 中获取了 ServiceFetcher 对象，并通过 ServiceFetcher 的 getService()方法返回我们所需 WifiManager 对象，那么 SYSTEM_SERVICE_FETCHERS 是什么? ServiceFetcher 又是什么，我们看下下面的代码:

源码路径:xxx/framework/base/core/java/android/app/SystemServiceRegistry.java

```
 private static final HashMap<String, ServiceFetcher<?>> SYSTEM_SERVICE_
FETCHERS = new HashMap<String, ServiceFetcher<?>>();



    /**
     * Base interface for classes that fetch services.
     * These objects must only be created during static initialization.
     */
    static abstract interface ServiceFetcher<T> {
        T getService(ContextImpl ctx);
    }


    /**
     * Override this class when the system service constructor needs a
     * ContextImpl and should be cached and retained by that context.
     */
```

```
    static abstract class CachedServiceFetcher<T> implements ServiceFetc
her<T> {

        private final int mCacheIndex;



        public CachedServiceFetcher() {

            mCacheIndex = sServiceCacheSize++;

        }



        @Override

        @SuppressWarnings("unchecked")

        public final T getService(ContextImpl ctx) {

            final Object[] cache = ctx.mServiceCache;

            synchronized (cache) {

                // Fetch or create the service.

                Object service = cache[mCacheIndex];

                if (service == null) {

                    service = createService(ctx);

                    cache[mCacheIndex] = service;

                }

                return (T)service;

            }

        }

        public abstract T createService(ContextImpl ctx);
```

```
    }
```

看到 SYSTEM_SERVICE_FETCHERS 的定义我们发现它其实是一个 HashMap，而 ServiceFetcher 是一个接口，而 CachedServiceFetcher 就是它的其中一个实现，我们看到当调用个 getService()方法的时候，先去 mServiceCache 这个数组中查找，如果取不到再调用 createService()这个抽象方法创建，至于 createService()的具体实现，请往下看。。。

3、WifiManager 如何的创建

既然 SYSTEM_SERVICE_FETCHERS 是一个 Map 我们不妨找找他是在什么地方被添加数据的:

源码路径:xxx/framework/base/core/java/android/app/SystemServiceRegistry.java

```
/**

 * Statically registers a system service with the context.

 * This method must be called during static initialization only.

 */private static <T> void registerService(String serviceName, Class<T>
serviceClass,

          ServiceFetcher<T> serviceFetcher) {

   SYSTEM_SERVICE_NAMES.put(serviceClass, serviceName);

   SYSTEM_SERVICE_FETCHERS.put(serviceName, serviceFetcher);}
```

我们发现 SYSTEM_SERVICE_FETCHERS 中的数据是通过 registerService()方法添加的，继续查看 SystemServiceRegistry.java 文件你会发现好多类似下面的方法在静态代码块中被调用:

```
registerService(Context.WIFI_SERVICE, WifiManager.class,
                new CachedServiceFetcher<WifiManager>() {
    @Override
    public WifiManager createService(ContextImpl ctx) {
        IBinder b = ServiceManager.getService(Context.WIFI_SERVICE);
        IWifiManager service = IWifiManager.Stub.asInterface(b);
        return new WifiManager(ctx.getOuterContext(), service);}});
```

在这段代码中，我们看到 createService()这个抽象方法被实现，到目前为止，我们已经明确了系统是如何一步步的给我们提供 WifiManager 这个对象的了，至于 WifiManager 到底做了什么，我们继续往后看!

二、系统服务的创建以及管理

1、WifiManager

既然我们拿到手的最后是 WifiManager，那么我们先看看 WifiManager 什么东西，他又是如何工作的，就以我们刚开始的时候调用的 disconnect()方法为例:

```
public boolean disconnect() {
    try {
        mService.disconnect();
        return true;
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }}
```

在代码中我们看到，程序最终是走了 mService.disconnect();的，那么 mService 又是什么呢?

```
public WifiManager(Context context, IWifiManager service, Looper looper)
 {

    mContext = context;

    mService = service;

    mLooper = looper;

    mTargetSdkVersion = context.getApplicationInfo().targetSdkVersion;}
```

到此我们看到其实 mService 是 IWifiManager，搜索源代码发现，其实它是一个 IWifiManager.aidl 文件，既然是 aidl 那么就肯定有具体的实现，至于它的具体实现是什么我们现在肯定是不知道的，我们需要一步步分析 mService 的来源，最终确定其具体的实现类是什么，我们继续往下走。。。

源码路径:xxx/frameworks/base/wifi/java/android/net/wifi/IWifiManager.aidl

2、IWifiManager service = IWifiManager.Stub.asInterface(b)

既然 IWifiManager 是通过 IWifiManager.Stub.asInterface(b)返回的，那么我们不妨分析分析这段代码;看到这句代码的时候，相信很多朋友都觉得很熟悉吧，根据 AIDL 的特殊性，正常情况下编译后会根据 IWifiManager.aidl 这个文件生成一个 IWifiManager.java 的接口，而这个接口中维护了 Stub 怎么一个内部类，我们在 AS 上创建一个 AIDL 编译生成一个 Stub 分析分析:

```
public interface IWifiManager extends android.os.IInterface {

    /**

     * Local-side IPC implementation stub class.
```

```
    */

    public static abstract class Stub extends android.os.Binder implemen
ts android.discovery

            .light.IWifiManager {

        private static final java.lang.String DESCRIPTOR = "android.disco
very.light.IWifiManager";



        /**

         * Construct the stub at attach it to the interface.

         */

        public Stub() {

            this.attachInterface(this, DESCRIPTOR);

        }



        /**

         * Cast an IBinder object into an android.discovery.light.IWifiMa
nager interface,

         * generating a proxy if needed.

         */

        public static android.discovery.light.IWifiManager asInterface(a
ndroid.os.IBinder obj) {

            if ((obj == null)) {

                return null;

            }
```

BAT交 892246

```
            android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTO
R);

            if (((iin != null) && (iin instanceof android.discovery.light.
IWifiManager))) {

                return ((android.discovery.light.IWifiManager) iin);

            }

            return new android.discovery.light.IWifiManager.Stub.Proxy(ob
j);

        }

       //此处省略 N 行代码==========================================}
```

在代码中我们可以看到 asInterface(android.os.IBinder obj)通过 obj.queryLocalInterface(DESCRIPTOR)返回了一个 IInterface，而在 Binder 中 queryLocalInterface()方法的实现如下:

```
public IInterface queryLocalInterface(String descriptor) {

   if (mDescriptor.equals(descriptor)) {

       return mOwner;

   }

   return null;}
```

这里首先对传入的 descriptor 进行判断，然后返回 mOwner 对象，他们在 Binder 中的定义以及赋值如下:

```
private IInterface mOwner;private String mDescriptor;public void attachI
nterface(IInterface owner, String descriptor) {

   mOwner = owner;

   mDescriptor = descriptor;}
```

看到 attachInterface()方法朋友们应该想起点什么了吧?没想其没关系，看看下面的代码:

```
private static final java.lang.String DESCRIPTOR = "android.discovery.light.IMyAidlInterface";

 public Stub() {

     this.attachInterface(this, DESCRIPTOR);}
```

这里我们看到其实当我们创建 Stub 实例的时候，通过构造方法把 this 以及包名为保存到 mOwner，mDescriptor 两个属性中，当我们调用 asInterface(IBinder obj)方法的时候，根据 mDescriptor 标识进行判断，如果匹配就返回 mOwner，返回的其实就是 obj 本身，这种是最正常的情况了，我们暂时以这种情况为准！

好了，扯远了，回到我们之前的代码：

```
registerService(Context.WIFI_SERVICE, WifiManager.class,

              new CachedServiceFetcher<WifiManager>() {

   @Override

   public WifiManager createService(ContextImpl ctx) {

        IBinder b = ServiceManager.getService(Context.WIFI_SERVICE);

        IWifiManager service = IWifiManager.Stub.asInterface(b);

        return new WifiManager(ctx.getOuterContext(), service);}});
```

按照我们上面的分析，正常情况下，其实 IWifiManager.Stub.asInterface(b)返回的其实就是 b 本身，那么接下来我们要确定 b 这个 IBinder 对象是什么即可：

BAT交流群,12246

3、那么接下来我们分析 ServiceManager.getService(Context.WIFI_SERVICE)

源码路径:xxx/frameworks/base/core/java/android/os/ServiceManager.java

```
/**
 * Returns a reference to a service with the given name.
 *
 * @param name the name of the service to get
 * @return a reference to the service, or <code>null</code> if the servic
e doesn't exist
 */public static IBinder getService(String name) {
    try {
        IBinder service = sCache.get(name);
        if (service != null) {
            return service;
        } else {
            return getIServiceManager().getService(name);
        }
    } catch (RemoteException e) {
        Log.e(TAG, "error in getService", e);
    }
    return null;}
```

BAT交流 892246

我们看到程序首先通过 sCache 这个 HasMap 中去取，如果不存在就通过 getIServiceManager().getService(name)去取，而 getIServiceManager()返回的是一个 IServiceManager 接口，他的具体实现是在 ServiceManagerNative#ServiceManagerProxy 这个内部类中的，看下源码：

源码路径:xxx/frameworks/base/core/java/android/os/ServiceManagerNative#ServiceManagerProxy

```
public IBinder getService(String name) throws RemoteException {

    Parcel data = Parcel.obtain();

    Parcel reply = Parcel.obtain();

    data.writeInterfaceToken(IServiceManager.descriptor);

    data.writeString(name);

    mRemote.transact(GET_SERVICE_TRANSACTION, data, reply, 0);

    IBinder binder = reply.readStrongBinder();

    reply.recycle();

    data.recycle();

    return binder;}
```

至此，我们就知道了 ServiceManager.getService(Context.WIFI_SERVICE)中返回的 IBinder 是怎么来的了，这里的 Parcel 大家先不用关心，这个东西只不过是一个容器而已，我们只要知道这里是通过 Context.WIFI_SERVICE 这个字符串标识获取到了对应的IBinder对象即可，但是直到此处，我们依然没能知道这个IBinder对象到底是什么，这个时候别急，既然是容器，能取出来东西自然也要能存东西，取出来的时候不知道是什么，存进去的时候总该知道吧，继续往下走。。。

4、ServiceManager.addService()

细心的朋友应该已经发现了 ServiceManager 中还有另一个方法 addService()，这个方法就是我们创建服务之后注册到系统去管理的方法，他的调用是在 SystemServer.java 中，那我们去 SystemServer 找找看。。。找了半天是否发现没有在 SystemServer 中找不到哪个是队 Wifi 服务进行注册的?别急，其实他是通过了 WifiService 这个东西进行注册的!

```
private static final String WIFI_SERVICE_CLASS = "com.android.server.wifi.WifiService";



    /**
     * Starts a miscellaneous grab bag of stuff that has yet to be refactored
     * and organized.
     */
    private void startOtherServices() {
            //省略很多。。。
            mSystemServiceManager.startService(WIFI_SERVICE_CLASS);
            //省略很多。。。
    }
```

这里我们看到 mSystemServiceManager.startService(WIFI_SERVICE_CLASS)怎么一句代码，查看其定义发现 mSystemServiceManager 是 SystemServiceManager:

源码路径:

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

xxx/frameworks/base/services/core/java/com/android/server/SystemServiceManager.java

```
 * Starts a service by class name.
 *
 * @return The service instance.
 */
 @SuppressWarnings("unchecked")
 public SystemService startService(String className) {
        final Class<SystemService> serviceClass;
        try {
            serviceClass = (Class<SystemService>)Class.forName(classNam
e);
        } catch (ClassNotFoundException ex) {
            Slog.i(TAG, "Starting " + className);
            throw new RuntimeException("Failed to create service " + clas
sName
                    + ": service class not found, usually indicates that t
he caller should "
                    + "have called PackageManager.hasSystemFeature() to ch
eck whether the "
                    + "feature is available on this device before trying t
o start the "
                    + "services that implement it", ex);
        }
        return startService(serviceClass);}
```

892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

这里我们看到，通过反射的方式返回了 WifiService 的 Class 对象，然后作为参数传给了 startService(serviceClass)：

```
/**
    * Creates and starts a system service. The class must be a subclass of
    * {@link com.android.server.SystemService}.
    *
    * @param serviceClass A Java class that implements the SystemService interface.
    * @return The service instance, never null.
    * @throws RuntimeException if the service fails to start.
    */
   @SuppressWarnings("unchecked")
   public <T extends SystemService> T startService(Class<T> serviceClass) {
       final String name = serviceClass.getName();
       Slog.i(TAG, "Starting " + name);


       // Create the service.
       if (!SystemService.class.isAssignableFrom(serviceClass)) {
           throw new RuntimeException("Failed to create " + name
                   + ": service must extend " + SystemService.class.getName());
       }
```

```
        final T service;

        try {

            Constructor<T> constructor = serviceClass.getConstructor(Cont
ext.class);

            service = constructor.newInstance(mContext);

        } catch (InstantiationException ex) {

            throw new RuntimeException("Failed to create service " + name

                    + ": service could not be instantiated", ex);

        } catch (IllegalAccessException ex) {

            throw new RuntimeException("Failed to create service " + name

                    + ": service must have a public constructor with a Con
text argument", ex);

        } catch (NoSuchMethodException ex) {

            throw new RuntimeException("Failed to create service " + name

                    + ": service must have a public constructor with a Con
text argument", ex);

        } catch (InvocationTargetException ex) {

            throw new RuntimeException("Failed to create service " + name

                    + ": service constructor threw an exception", ex);

        }



        // Register it.

        mServices.add(service);
```

```
        // Start it.
        try {
            service.onStart();
        } catch (RuntimeException ex) {
            throw new RuntimeException("Failed to start service " + name
                    + ": onStart threw an exception", ex);
        }
        return service;
    }
```

我们发现，代码通过 service = constructor.newInstance(mContext)方法创建了 WifiService 的实例，并且调用了 service.onStart();方法，继续查看 WifiService 的源码：

源码路径：

xxx/frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiService.java

```
package com.android.server.wifi;import android.content.Context;import an
droid.net.wifi.ScanResult;import android.net.wifi.WifiScanner;import and
roid.util.Log;import com.android.server.SystemService;import java.util.L
ist;

 public final class WifiService extends SystemService {

    private static final String TAG = "WifiService";

    final WifiServiceImpl mImpl;



    public WifiService(Context context) {
```

```
        super(context);

        mImpl = new WifiServiceImpl(context);

    }


    @Override

    public void onStart() {

        Log.i(TAG, "Registering " + Context.WIFI_SERVICE);

        publishBinderService(Context.WIFI_SERVICE, mImpl);

    }


    @Override

    public void onBootPhase(int phase) {

        if (phase == SystemService.PHASE_SYSTEM_SERVICES_READY) {

            mImpl.checkAndStartWifi();

        }

    }}
```

这里，我们看到了在 WifiService 的构造方法中创建了 WifiServiceImpl 实例，并且通过 onStart()方法，将 WifiServiceImpl 实例 mImpl 传入到 publishBinderService(Context.WIFI_SERVICE, mImpl)中，我们继续到 SystemService 看看：

源码路径：

xxx/frameworks/base/services/core/java/com/android/server/SystemService.java

```
/**
 * Publish the service so it is accessible to other services and apps.
 */protected final void publishBinderService(String name, IBinder servic
e,
    boolean allowIsolated) {
     ServiceManager.addService(name, service, allowIsolated);}
```

历尽千辛万苦，终于看到我们想要找的东西了，原来 publishBinderService()方法通过 ServiceManager.addService(name, service, allowIsolated)将 WifiServiceImpl 注册到了 ServiceManager 中，我们看下注册的方式：

源码路径:xxx/frameworks/base/core/java/android/os/ServiceManagerNative#ServiceManagerProxy

```
public void addService(String name, IBinder service, boolean allowIsolat
ed)
            throws RemoteException {
   Parcel data = Parcel.obtain();
   Parcel reply = Parcel.obtain();
   data.writeInterfaceToken(IServiceManager.descriptor);
   data.writeString(name);
   data.writeStrongBinder(service);
   data.writeInt(allowIsolated ? 1 : 0);
   mRemote.transact(ADD_SERVICE_TRANSACTION, data, reply, 0);
   reply.recycle();
```

```
data.recycle();}
```

这个保存方式跟我们 getService()的方式是一致的！

至此，我们算是找到总算找到了 IWifiManager.aidl 的具体实现了，就是 WifiServiceImpl，那么也就意味着我们的这整个 WIfi 的系统服务的接口定义、实现、注册、访问等流程都有两个一个大致的了解！

5、抱着怀疑的态度，我们看下 WifiServiceImpl 中是否真的实现了 disconnect()方法：

源码路径：xxx/frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiServiceImpl.java

```
/**
 * see {@link android.net.wifi.WifiManager#disconnect()}
 */
public void disconnect() {
    enforceChangePermission();
    mWifiStateMachine.disconnectCommand();
}
```

ok，验证通过！

三、总结

这整个 Wifi 服务是如何被创建以及注册到系统管理，然后如何向上层提供接口进行调用的过程看起来比较复杂，但是其实系统服务里面多数的实现方式都差不

多是这样，只要理解了其中一个，其它的就相差不是很大，更重要的是我们去了解怎么一个过程，对我们以后进行 framework 层的开发还是有非常大的帮助的，比如说我们如何添加一个自己的系统服务。那么接下来我们队这整个流程进行简单的总结。

1、创建相关的 AIDL 文件 IWifiManager.aidl，并定义相关功能方法。

2、创建 WifiServiceImpl.java 并实现 IWifiManager.Stub，并在 Contex 中定义服务的唯一标识字符串。

3、创建创建 WifiManager.java 内部维护了通过 ServiceManager 获取到的 WifiServiceImpl 实例。

4、在 SystemServiceRegistry.java 的静态太代码块中通过 registerService()方法创建 WifiManager 实例，并保存到 SYSTEM_SERVICE_FETCHERS 这个 map 中，这里要强调一点的是，由于 WifiManager 是在静态代码块中被创建，所以是单例的。

5、在 SystemServer.java 中通过 SystemServiceManager 创建 WifiService 实例，并调用其 onStart()方法。

6、在 WifiService 的构造方法中创建 WifiServiceImpl 实例。

7、最终通过 SystemService 的 publishBinderService() 方法调用 ServiceManager.addService(name, service, allowIsolated);将 WifiServiceImpl 注册到 ServiceManager 中。

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

第三章、组件化框架设计

第一节、阿里巴巴开源路由框——ARouter 原理分析

一、背景

当项目的业务越来越复杂，业务线越来越多的时候，就需要按照业务线去分不同的模块去开发，这样专门的人负责专门的业务模块，最终上线由壳工程去负责进行组合打包各个 module，完成业务的快速迭代。整个过程会涉及到各个模块间进行通信，比如订单模块和个人中心模块，可能会需要频繁的传递数据和页面跳转，这个时候怎么去处理呢？我们能想到的方案就是采用类名反射，来动态创建需要跳转和交互的类，这样编译时就不会报错，运行时又可以完成模块间的交互。阿里巴巴推出的开源路由框架——ARouter 就是基于反射和注解来解决这个问题的，本文不讲基本使用（基本使用在项目的 github 主页上已经将的非常详细了），通过分析整个路由过程来讲解它的基本原理。

二、说在前面

首先在我们需要用到的类的类名加上注解@Route(“/group/name”)，注意这里需要至少两层路径(第一个是分组，第二个一般是类名)。这个注解就是代表这个类可以被其他模块找到的一个路径的注解，并且它是一个编译时注解，这就意味着在编译时就已经生成了相应的辅助类。ARouter 把路由一共分为以下几类：

```
ACTIVITY(0, “android.app.Activity”), SERVICE(1, “android.app.Service”),
PROVIDER(2, “com.alibaba.android.arouter.facade.template.IProvider”),

CONTENT_PROVIDER(-1, “android.app.ContentProvider”), BOARDCAST(-1, “”),
METHOD(-1, “”), FRAGMENT(-1, “android.app.Fragment”), UNKNOWN(-1, “Unkno
wn route type”);
```

其中我们常用的就是 ACTIVITY，PROVIDER，FRAGMENT 这三个了，也基本上满足了我们模块化开发的需求。另外一点就是分组的概念，ARouter 是按照组来进行整理的，也就是第一层的路径，所以前面说必须要两层路径，否则不知道归

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

到哪里去，一般一个 module 按照模块名采用统一的分组标识。我们来看看注解生成的类（这里只包含了 Activity,Fragment,Provider）：

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

/ / 。 。 。import 省略/**

 * DO NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER. */public class A
Router$$Group$$Personal implements IRouteGroup {

  @Override

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

    atlas.put("/Personal/EARNING", RouteMeta.build(RouteType.ACTIVITY, P
erEarningActivity.class, "/personal/earning", "personal", null, -1, -214
7483648));

    //...省略 Activity，Fragment

    atlas.put("/Personal/main", RouteMeta.build(RouteType.FRAGMENT, PerM
ainFragment.class, "/personal/main", "personal", null, -1, -214748364
8));

    atlas.put("/Personal/service", RouteMeta.build(RouteType.PROVIDER, P
erServiceImpl.class, "/personal/service", "personal", null, -1, -2147483
648));

  }}
```

以上就是所有注解的路径的信息集合，包含了所有的 Activity，Fragment，Provider(一般一个 module 一个 Provider 就够用了，专门用来跟其他模块交互)，并都以路径为 key 放到这个 map 中。

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

/ / 。 。 。import 省略/**

 * DO NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER. */public class A
Router$$Providers$$modlue_personal implements IProviderGroup {
```

```
 @Override

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

   providers.put("com.tb.test.service.ModulePersonalService", RouteMeta.
build(RouteType.PROVIDER, PerServiceImpl.class, "/Personal/service", "pe
rsonal", null, -1, -2147483648));

 }}
```

这个类是专门的 Provider 的索引的集合，所有的 provider 都被以全类名为索引放到一个 map 中。

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

/ / 。 。 。import 省略

/**

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

 @Override

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

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

 }}
```

这个类是所有的 group 的信息收集，全部都以 group 的名字为 key，以注解生成的不同的 group 的类的 class 对象为 value 放入到一个 map 中。

总共就生成这三种类型的类，当然，如果你有不同的分组还会生成其他的类，不过都是这三种里面的一种。

完成了这些注解信息的收集，下面就会去使用这些信息来完成我们的跨模块交互了。

BAT文/2246

三、初始化过程

使用 ARouter 必须先要进行初始化：

```
if (isDebug()) { // These two lines must be written before init, otherwis
e these configurations will be invalid in the init process

ARouter.openLog(); // Print log

ARouter.openDebug(); // Turn on debugging mode (If you are running in In
stantRun mode, you must turn on debug mode! Online version needs to be cl
osed, otherwise there is a security risk) }



ARouter.init(mApplication); // As early as possible, it is recommended t
o initialize in the Application
```

上面这段话就是去初始化 Arouter，我们来看看 init 里面到底做了什么事。。。

```
/**

     * Init, it must be call before used router.

     */

    public static void init(Application application) {

        if (!hasInit) {

            logger = _ARouter.logger;

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

            hasInit = _ARouter.init(application);



            if (hasInit) {

                _ARouter.afterInit();
```

```
            }


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

        }

    }
```

可以看到，这里使用了外观模式，最终调用都是在_ARouter 这个类里面，跟进去：

```
protected static synchronized boolean init(Application application) {

        mContext = application;

        LogisticsCenter.init(mContext, executor);

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

        hasInit = true;


        // It's not a good idea.

        // if (Build.VERSION.SDK_INT > Build.VERSION_CODES.ICE_CREAM_SAND
WICH) {

        //     application.registerActivityLifecycleCallbacks(new Autowir
edLifecycleCallback());

        // }

        return true;

    }
```

代码也很简单，核心就是 LogisticsCenter.init 这句话，跟进去看看，核心代码如下：

BAT文 892246

```
List<String> classFileNames = ClassUtils.getFileNameByPackageName(mConte
xt, ROUTE_ROOT_PAKCAGE);



            //

            for (String className : classFileNames) {

                if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_N
AME + SEPARATOR + SUFFIX_ROOT)) {

                    // This one of root elements, load root.

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

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

                    // Load interceptorMeta

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

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

                    // Load providerIndex

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

                }

            }
```

我们可以看到，首先是去获取到所有的 app 里的由 ARouter 注解生成的类的类名，他们的统一特点就是在同一个包下，包名为：

com.alibaba.android.arouter.routes

然后就是循环遍历这些类，也就是刚才我们说的那三种类。在这里，有一个 Warehouse 类，看下代码：

```
/**
 * Storage of route meta and other data.
 *
 * @author zhilong <a href="mailto:zhilong.lzl@alibaba-inc.com">Contact me.</a>
 * @version 1.0
 * @since 2017/2/23 下午1:39
 */class Warehouse {
    // Cache route and metas
    static Map<String, Class<? extends IRouteGroup>> groupsIndex = new HashMap<>();
    static Map<String, RouteMeta> routes = new HashMap<>();

    // Cache provider
    static Map<Class, IProvider> providers = new HashMap<>();
    static Map<String, RouteMeta> providersIndex = new HashMap<>();

    // Cache interceptor
    static Map<Integer, Class<? extends IInterceptor>> interceptorsIndex = new UniqueKeyTreeMap<>("More than one interceptors use same priority [%s]");
    static List<IInterceptor> interceptors = new ArrayList<>();

    static void clear() {
        routes.clear();
```

```
        groupsIndex.clear();

        providers.clear();

        providersIndex.clear();

        interceptors.clear();

        interceptorsIndex.clear();

    }}
```

很简单，定义了几个静态 map，在初始化的时候来存放之前的注解生成的那些相关信息。初始化里面存的就是所有 group 索引的 map，所有拦截器(本文不讲)索引的 map，所有 provider 索引的 map。至此，之前的那些注解类里面的信息都被存储起来了，这样后续在查找的时候就很方便可以找到对应的类，我们继续看初始化之后的 afterInit 方法：

```
static void afterInit() {

        // Trigger interceptor init, use byName.

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

    }
```

我们跟踪之后，发现最终会调用 LogisticsCenter 中的 completion 方法：

```
/**

     * Completion the postcard by route metas

     *

     * @param postcard Incomplete postcard, should completion by this met
hod.

     */
```

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

        if (null == postcard) {

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

        }



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

        if (null == routeMeta) {    // Maybe its does't exist, or didn't l
oad.

            Class<? extends IRouteGroup> groupMeta = Warehouse.groupsInde
x.get(postcard.getGroup());  // Load route meta.

            if (null == groupMeta) {

                throw new NoRouteFoundException(TAG + "There is no route m
atch the path [" + postcard.getPath() + "], in group [" + postcard.getGro
up() + "]");

            } else {

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

                try {

                    if (ARouter.debuggable()) {

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

                    }



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

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

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



                    if (ARouter.debuggable()) {

                        logger.debug(TAG, String.format(Locale.getDefault
(), "The group [%s] has already been loaded, trigger by [%s]", postcard.
getGroup(), postcard.getPath()));

                    }

                } catch (Exception e) {

                    throw new HandlerException(TAG + "Fatal exception when
 loading group meta. [" + e.getMessage() + "]");

                }



                completion(postcard);   // Reload

            }

        } else {

            postcard.setDestination(routeMeta.getDestination());

            postcard.setType(routeMeta.getType());

            postcard.setPriority(routeMeta.getPriority());

            postcard.setExtra(routeMeta.getExtra());



            Uri rawUri = postcard.getUri();

            if (null != rawUri) {   // Try to set params into bundle.

                Map<String, String> resultMap = TextUtils.splitQueryParam
eters(rawUri);
```

BAT文 892246

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



            if (MapUtils.isNotEmpty(paramsType)) {

                // Set value by its type, just for params which annota
tion by @Param

                for (Map.Entry<String, Integer> params : paramsType.en
trySet()) {

                    setValue(postcard,

                            params.getValue(),

                            params.getKey(),

                            resultMap.get(params.getKey()));

                }



                // Save params name which need autoinject.

                postcard.getExtras().putStringArray(ARouter.AUTO_INJE
CT, paramsType.keySet().toArray(new String[]{}));

            }



            // Save raw uri

            postcard.withString(ARouter.RAW_URI, rawUri.toString());

        }



        switch (routeMeta.getType()) {
```

```
                case PROVIDER:  // if the route is provider, should find i
ts instance

                    // Its provider, so it must be implememt IProvider

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

                    IProvider instance = Warehouse.providers.get(providerM
eta);

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

                        IProvider provider;

                        try {

                            provider = providerMeta.getConstructor().newIn
stance();

                            provider.init(mContext);

                            Warehouse.providers.put(providerMeta, provide
r);

                            instance = provider;

                        } catch (Exception e) {

                            throw new HandlerException("Init provider fail
ed! " + e.getMessage());

                        }

                    }

                    postcard.setProvider(instance);

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

                    break;

                case FRAGMENT:
```

```
                postcard.greenChannel();    // Fragment needn't interc
eptors

            default:

                break;

        }

    }

}
```

这个方法有点长，不过我们可以看到，核心功能就是 postcard 的信息完善。postcard 就是整个路由过程中的信使，类似于生活中的明信片功能，包含了路由所有需要的信息。通过第 34 行的递归调用，根据 groupsIndex 和 providersIndex 保证了 Warehouse 里面的另外两个静态 map(routes,providers)的赋值，这样最终都会走到 36 行 else 分支，去保证所有路由信息的完整性，另外 swtich...case 里面的 postcard.greenChannel()其实是 activity 跳转专用的，目的是用来拦截 activity 跳转，来对跳转过程进行干预，在之前或者之后做一些自己的处理，所以 greenChannel 就是绿色通道，不进行拦截。另外代码里面也可以看到，类的生成都是采用 getConstructor().newInstance()这种反射来进行的，最终调用：

```
ModulePersonalService service = (ModulePersonalService) ARouter.getInsta
nce().build("/Personal/service").navigation();
```

得到这个跨模块服务之后，里面的所有方法都可以去调用来实现功能需求了。

调用过程

Activity 的跳转如下：

```
        ARouter.getInstance().build("/Personal/main").navigation(acti
vity);
```

BAT/2246

最终调用代码则是_ARouter 类里面的_navigation 方法：

```
private Object _navigation(final Context context, final Postcard postcar
d, final int requestCode, final NavigationCallback callback) {

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



        switch (postcard.getType()) {

            case ACTIVITY:

                // Build intent

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

                intent.putExtras(postcard.getExtras());



                // Set flags.

                int flags = postcard.getFlags();

                if (-1 != flags) {

                    intent.setFlags(flags);

                } else if (!(currentContext instanceof Activity)) {    //
Non activity, need less one flag.

                    intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);

                }



                // Navigation in main looper.

                new Handler(Looper.getMainLooper()).post(new Runnable() {
```

```
                    @Override

                    public void run() {

                        if (requestCode > 0) {  // Need start for result

                            ActivityCompat.startActivityForResult((Activit
y) currentContext, intent, requestCode, postcard.getOptionsBundle());

                        } else {

                            ActivityCompat.startActivity(currentContext, i
ntent, postcard.getOptionsBundle());

                        }



                        if ((0 != postcard.getEnterAnim() || 0 != postcard.
getExitAnim()) && currentContext instanceof Activity) {    // Old versio
n.

                            ((Activity) currentContext).overridePendingTra
nsition(postcard.getEnterAnim(), postcard.getExitAnim());

                        }



                        if (null != callback) { // Navigation over.

                            callback.onArrival(postcard);

                        }

                    }

                });



                break;

            case PROVIDER:
```

```
                return postcard.getProvider();

            case BOARDCAST:

            case CONTENT_PROVIDER:

            case FRAGMENT:

                Class fragmentMeta = postcard.getDestination();

                try {

                    Object instance = fragmentMeta.getConstructor().newIns
tance();

                    if (instance instanceof Fragment) {

                        ((Fragment) instance).setArguments(postcard.getExt
ras());

                    } else if (instance instanceof android.support.v4.app.
Fragment) {

                        ((android.support.v4.app.Fragment) instance).setAr
guments(postcard.getExtras());

                    }



                    return instance;

                } catch (Exception ex) {

                    logger.error(Consts.TAG, "Fetch fragment instance erro
r, " + TextUtils.formatStackTrace(ex.getStackTrace()));

                }

            case METHOD:

            case SERVICE:

            default:
```

```
            return null;
        }


        return null;
    }
```

可以看到对 Activity 的处理，最终就是调用 startActivity 方法，对 provider 就是返回一个类的实例，而 BOARDCAST、CONTENT_PROVIDER、FRAGMENT 也都是生成一个实例返回，对于 METHOD、SERVICE 暂时是没有处理的。

拦截器和自动注入的功能，本文没有去分析，一般跳到某一个页面需要判断是否登陆的时候，可以使用拦截器，自动注入可以在页面间传递数据，非常方便。

第二节、APT 编译时期自动生成代码&动态类加载

本篇文章将继续从以下两个内容来介绍组件化框架设计:

- apt 编译时期自动生成代码
- Android 动态加载技术基础之类加载（ClassLoader）

一、apt 编译时期自动生成代码

第一步

新建一个 android 项目。

第二步

新建立一个 java 的 Module。注意是 javalib。这个 lib 用来专门写注解就好。

这个 lib 里面就先放一个注解，叫 TestAnno。

,/2246

```
import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;



@Retention(RetentionPolicy.CLASS)@Target(ElementType.TYPE)

public @interface TestAnno {}
```

RetentionPolicy.CLASS 表示编译时候注解。你需要关系的就是

@Target(ElementType.TYPE)这个 type 是类的注解，可以有方法的，属性的等等。

然后这个 javalib 的 gradle 文件要这么写。

```
apply plugin: 'java'



dependencies {

    compile fileTree(dir: 'libs', include: ['*.jar'])}

sourceCompatibility = "1.7"

targetCompatibility = "1.7"
```

注解库弄好了，在弄新建一个 java lib 叫 inject_comiler。这个是就是核心代码了，在编译时候，执行这个个库里面的代码，然后 生成代码。这个工程 三个类。一个是注解注解处理器的核心。一个是定义生成 java 文件的，方法拼接。还有一个就是常量包名类名了。

先看这个的 gradle 文件。

```
apply plugin: 'java'
```

```
sourceCompatibility = "1.7"

targetCompatibility = "1.7"

dependencies {

    compile fileTree(include: ['*.jar'], dir: 'libs')

    compile 'com.google.auto.service:auto-service:1.0-rc2'//谷歌的帮助我
们快速实现注解处理器

    compile project(':inject_annotation')//自己定义的注解的 java lib

    compile 'com.squareup:javapoet:1.7.0'//用来生成 java 文件的，避免字符串
拼接的尴尬}

//这个注解是谷歌提供了，快速实现注解处理器，会帮你生成配置文件啥的 。直接用就好

@AutoService(Processor.class)public class ActivityInjectProcesser extend
s AbstractProcessor {

    private Filer mFiler; //文件相关的辅助类

    private Elements mElementUtils; //元素相关的辅助类  许多元素

    private Messager mMessager; //日志相关的辅助类



    private Map<String, AnnotatedClass> mAnnotatedClassMap;



    @Override

    public synchronized void init(ProcessingEnvironment processingEnv) {

        super.init(processingEnv);

        mFiler = processingEnv.getFiler();

        mElementUtils = processingEnv.getElementUtils();

        mMessager = processingEnv.getMessager();
```

```
        mAnnotatedClassMap = new TreeMap<>();

    }



//这个方法是核心方法，在这里处理的你的业务。检测类别参数，生成 java 文件等

    @Override

    public boolean process(Set<? extends TypeElement> annotations, Round
Environment roundEnv) {

        mAnnotatedClassMap.clear();



        try {

            processActivityCheck(roundEnv);

        } catch (Exception e) {

            e.printStackTrace();

            error(e.getMessage());

        }



        for (AnnotatedClass annotatedClass : mAnnotatedClassMap.values())
 {

            try {

                annotatedClass.generateActivityFile().writeTo(mFiler);

            } catch (Exception e) {

                error("Generate file failed, reason: %s", e.getMessage
());

            }
```

```
        }

        return true;

    }





    private void processActivityCheck(RoundEnvironment roundEnv) throws
IllegalArgumentException, ClassNotFoundException {

        //check ruleslass forName(String className

        for (Element element : roundEnv.getElementsAnnotatedWith((Class<?
 extends Annotation>) Class.forName(TypeUtil.ANNOTATION_PATH))) {

            if (element.getKind() == ElementKind.CLASS) {

                getAnnotatedClass(element);

            } else

                error("ActivityInject only can use  in ElementKind.CLASS
");

        }

    }



    private AnnotatedClass getAnnotatedClass(Element element) {

        // tipe . can not use chines  so  ....

        // get TypeElement  element is class's --->class  TypeElement typ
eElement = (TypeElement) element

        //  get TypeElement  element is method's ---> TypeElement typeEle
ment = (TypeElement) element.getEnclosingElement();

        TypeElement typeElement = (TypeElement) element;
```

```
        String fullName = typeElement.getQualifiedName().toString();

        AnnotatedClass annotatedClass = mAnnotatedClassMap.get(fullNam
e);

        if (annotatedClass == null) {

            annotatedClass = new AnnotatedClass(typeElement, mElementUtil
s, mMessager);

            mAnnotatedClassMap.put(fullName, annotatedClass);

        }

        return annotatedClass;

    }




    @Override

    public SourceVersion getSupportedSourceVersion() {

        return SourceVersion.latestSupported();

    }




    //这个个方法返回你要处理注解的类型

    @Override

    public Set<String> getSupportedAnnotationTypes() {

        Set<String> types = new LinkedHashSet<>();

        types.add(TypeUtil.ANNOTATION_PATH);

        return types;
```

```
    }


    private void error(String msg, Object... args) {

        mMessager.printMessage(Diagnostic.Kind.ERROR, String.format(msg,
 args));

    }


    private void log(String msg, Object... args) {

        mMessager.printMessage(Diagnostic.Kind.NOTE, String.format(msg,
args));

    }

}
```

然后是生成 java 文件的辅助类。

```
public class AnnotatedClass {


    private TypeElement mTypeElement;//activity  //fragmemt

    private Elements mElements;

    private Messager mMessager;//日志打印


    public AnnotatedClass(TypeElement typeElement, Elements elements, Me
ssager messager) {

        mTypeElement = typeElement;

        mElements = elements;

        this.mMessager = messager;
```

```
    }

    public JavaFile generateActivityFile() {

        // build inject method

        MethodSpec.Builder injectMethod = MethodSpec.methodBuilder(TypeU
til.METHOD_NAME)

                .addModifiers(Modifier.PUBLIC)

                .addParameter(TypeName.get(mTypeElement.asType()), "activ
ity", Modifier.FINAL);

        injectMethod.addStatement("android.widget.Toast.makeText" +"(act
ivity, $S,android.widget.Toast.LENGTH_SHORT).show();", "from build");

        //generaClass

        TypeSpec injectClass = TypeSpec.classBuilder(mTypeElement.getSim
pleName() + "$$InjectActivity")

                .addModifiers(Modifier.PUBLIC)

                .addMethod(injectMethod.build())

                .build();

        String packgeName = mElements.getPackageOf(mTypeElement).getQual
ifiedName().toString();

        return JavaFile.builder(packgeName, injectClass).build();

    }

    }
```

这里就生成了一个 类名+$$的类，有一个方法叫 inject 参数是这个类本身，弹出一个 toast。最后一个就是一个字符常量类。

```
public class TypeUtil {

    public static final String METHOD_NAME = "inject";

    public static final String ANNOTATION_PATH = "com.example.TestAnno
";}
```

好了 lib 工程完毕。然后是主工程引用。

首先是在工程的 gradle 里面配置下 apt。

```
dependencies {

        classpath 'com.android.tools.build:gradle:2.3.2'

        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'

    }
```

然后在 app 的 gradle 里面配置如下

```
apply plugin: 'com.neenbedankt.android-apt'

……

 compile project(':inject_annotation')

  apt project(':inject_comiler')
```

这样就行了。注意是 apt 。为什么要新建立 javalib。因为 javalib 不能在引用 adnroidlib，。而注解处理器是 javalib 来完成，app 里面，可以这引用这 2 个，apt 如果换成 complie 会提示错误，但不会影响啥。配置都好了，就是最后的使用了。

BAT交流群：892872246

总结构

在我们 MainActivity 上面加上注解，使用下。

```
@TestAnnopublic class MainActivity extends AppCompatActivity {

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        InjectActivity.inject(this);//调用 build 生成的类
```

```
    }}
```

最后一个类就是我们的使用这个在运行时，调用生成 build 的类

```
public class InjectActivity {

    private static final ArrayMap<String, Object> injectMap = new ArrayMa
p<>();



    public static void inject(AppCompatActivity activity) {

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

        try {



            Object inject = injectMap.get(className);



            if (inject == null) {

               //加载 build 生成的类

                Class<?> aClass = Class.forName(className + "$$InjectActi
vity");

                inject = aClass.newInstance();

                injectMap.put(className, inject);

            }

            //反射执行方法

            Method m1 = inject.getClass().getDeclaredMethod("inject", act
ivity.getClass());

            m1.invoke(inject, activity);

        } catch (Exception e) {
```

```
            e.printStackTrace();

        }

    }}
```

这个类用了反射的方式查找了指定类执行了指定方法。有一个 map 来缓存，不用每次都重新反射。

当然你可可以不用反射，在创建 build 生成类的时候，实现一个接口，这里直接强转为接口，直接调用接口的方法也可以。这里简单一些用的反射

通过一个注解，就自动生成弹出 toast 的代码看下自动 生成的代码

```
public class MainActivity$$InjectActivity {

  public void inject(final MainActivity activity) {

    android.widget.Toast.makeText(activity, "from build",android.widget.
Toast.LENGTH_SHORT).show();;

  }}
```

最后看下运行效果。

BAT交流群：892872246

二、Android 动态加载技术基础之类加载（ClassLoader）

虚拟机类加载机制

- 类加载过程是指虚拟机将描述类的数据从 Class 文件中加载到内存，并对数据进行校验，转化解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型的过程。

BAT交流群：892872246

- 在 Java 中，类的加载和连接过程都是在程序运行期间完成。虽然会增加运行时的性能开销，但可以提高程序灵活性，这也是 Java 能够实现动态加载的原因之一。

类加载的过程

虚拟机类加载过程分为加载，验证，准备，解析，初始化，使用，卸载七个阶段。其中验证，准备，解析三个部分成为连接阶段。

加载

一般来说，在遇到了以下四种情况的时候，虚拟机会开始进行类的加载：

- 使用 new 关键字实例化对象，读取或者设置一个类的静态变量（被 final 修饰的除外，已经在编译器被加入常量池），以及调用一个类的静态方法的时候
- 对类进行反射调用的时候
- 当初始化一个类时，如果其父类没有被加载，则先对其父类进行加载
- 当虚拟机启动的时候，用户指定的（包含 main 方法）的类会被加载

在类的加载阶段，虚拟机会完成一下三件事情:

- 通过一个类的全限定名获取定义类的二进制字节流
- 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
- 在 Java 堆中生成一个代表这个类的 Class 对象，作为方法区这些数据的访问入口。

验证

这一阶段是为了确保 class 文件的字节流包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。主要包括以下几个过程：

- 文件格式验证

892872246
BAT交

- 元数据验证
- 字节码验证
- 符号引用验证

准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配。这个阶段中进行内存分配的变量只有被 static 修饰的变量，

并将其设置为默认值，而真正的赋值则在初始化阶段。另外，被 final static 字段修饰的常量在编译器就已经被赋值。

解析

解析阶段主要是虚拟机将常量池内的符号引用替换为直接引用的过程。

初始化

初始化阶段是执行类构造器<cinit>()方法的过程。

<cinit>()与类的构造方法不同，<cinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块中的语句合并而成的。编译器收集的顺序是按语句在源文件中出现的顺序决定的，静态语句块中只能访问定义在它之前的静态变量，定义在它之后的静态变量，只可以赋值，不可以访问。

虚拟机会保证子类的<cinit>()方法执行之前，其父类的<cinit>()方法一定被执行（父类先与子类完成加载过程）

三、Java 中的 ClassLoader

类加载阶段中，实现“通过一个类的全限定名获取定义类的二进制字节流”的动作被放在了虚拟机外部去实现，以便应用程序决定如何去加载所需要的类，实现这个动作的代码模块被称为“类加载器”

从 Java 虚拟机的角度上讲，只存在两种不同的类加载器：

- Bootstrap ClassLoader：使用 C++实现，是虚拟机的一部分。它主要负责加载存放在%JAVAHOME%/lib 目录中的，或者被-Xbootclasspath 指定的类库到虚拟机内存中，Bootstrap ClassLoader 无法被 java 程序直接引用。
- 继承自 java.lang.ClassLoader 的类加载器：
- Extension ClassLoader：主要负责加载%JAVAHOME%/lib/ext 目录中的，或者被 java.ext.dirs 系统变量指定路径的所有类。
- Application ClassLoader：也被称为系统类加载器（因为其实 getSystemClassLoader 的返回对象），主要负责加载用户类路径（ClassPath）下的类库

类的双亲委派模型

这些类加载器之间的关系如下：

双亲委派模型中，除了顶层的 BootstrapClassLoader，其他类加载器都要求有自己的父类加载器，这里的类加载器一般以组合的方式实现。

- 双亲委派模型的工作过程是：当一个类加载器收到一个类加载请求的时候，他首先不会自己加载这个类，而是将这个请求委派给父类加载器去完成，只有当父类加载器无法完成这个加载请求时，子加载器才会尝试自己去加载。
- 双亲委派模型的作用：
- 使得 java 类随着它的类加载器一起具备了一种带有优先级的层次关系
- 保证 Java 环境的稳定性
- 避免重复加载，如果已经加载过一次 Class，就不需要再次加载，而是先从缓存中直接读取。

四、Android 中的 ClassLoader

- 由于 Android 虚拟机的执行文件是 Dex 文件，而不是 JVM 中的 Class 文件，所以 Java 中的类加载器是无法加载 Dex 文件的，因此，Android 中存在另外一套 ClassLoader。

Android 的 ClassLoader 类型

Android 中的 ClassLoader 根据用途可分为一下几种：

- BootClassLoader：主要用于加载系统的类，包括 java 和 android 系统的类库，和 JVM 中不同，BootClassLoader 是 ClassLoader 内部类，是由 Java 实现的，它也是所有系统 ClassLoader 的父 ClassLoader

BAT交流群：892872246

- PathClassLoader：用于加载 Android 系统类和开发编写应用的类，只能加载已经安装应用的 dex 或 apk 文件，也是 getSystemClassLoader 的返回对象
- DexClassLoader：可以用于加载任意路径的 zip,jar 或者 apk 文件，也是进行安卓动态加载的基础

Android 中 ClassLoader 的继承关系

BAT交流群：892872246

ClassLoader 的执行过程

- 从上面的 ClasLoader 结构图可以看到，ClassLoader 的主要逻辑主要集中在 ClassLoader 和 BaseDexClassLoder 这两个类中。

ClasLoader

- ClasLoader 是所有 ClassLoader 的父类，它定义了加载 Class 的一般行为。
- 与 Java 中不同，Android 中加载类的过程主要是由 loadClass 方法实现，而在 Java 中则是 findClass 方法。

```
protected Class<?> loadClass(String className, boolean resolve) throws C
lassNotFoundException {

    Class<?> clazz = findLoadedClass(className);

    if (clazz == null) {

        ClassNotFoundException suppressed = null;

        try {

            clazz = parent.loadClass(className, false);

        } catch (ClassNotFoundException e) {

            suppressed = e;

        }

        if (clazz == null) {

            try {

                clazz = findClass(className);

            } catch (ClassNotFoundException e) {
```

```
            e.addSuppressed(suppressed);
            throw e;
        }
    }
}
return clazz;}
```

- 可以看到，当收到一个加载类请求的时候，ClassLoader 会先调用 findLoadedClass 查询是否本类是否被本加载器加载过（调用 JNI 方法），如果没有被加载则把请求委托给父加载器，当父加载器无法完成加载行为的时候，才会调用 findClass 方法尝试自己加载，而 ClassLoader 中的 findClass 方法并没有实现，而是交给子类去是实现。

```
protected Class<?> findClass(String name) throws ClassNotFoundException {
    throw new ClassNotFoundException(name);
}
```

BaseDexClassLoader

- 作为 ClassLoader 的直接子类，BaseDexClassLoader 实现加载 findClass 方法的主要逻辑，而其子类 DexClassLoader 和 PathClassLoader 都只是加载路径以及某些行为不同而已

BAT交流群：892872246

```
@Override
protected Class<?> findClass(String name) throws ClassNotFoundException {
    List<Throwable> suppressedExceptions = new ArrayList<Throwable>();
    Class c = pathList.findClass(name, suppressedExceptions);
    if (c == null) {
        ClassNotFoundException cnfe = new ClassNotFoundException("Didn't find class \"
        for (Throwable t : suppressedExceptions) {
            cnfe.addSuppressed(t);
        }
        throw cnfe;
    }
    return c;
}
```

- 可以看到 findClass 方法是又是调用了 pathList 的 findClass 方法去加载类，而 pathList 则是一个 DexPathList 对象，它的 findClass 对象是这样实现的：

```
public Class findClass(String name, List<Throwable> suppressed) {
    for (Element element : dexElements) {
        DexFile dex = element.dexFile;

        if (dex != null) {
            Class clazz = dex.loadClassBinaryName(name, definingContext, suppressed);
            if (clazz != null) {
                return clazz;
            }
        }
    }
    if (dexElementsSuppressedExceptions != null) {
        suppressed.addAll(Arrays.asList(dexElementsSuppressedExceptions));
    }
    return null;
}
```

- 其中 DexFile 是 Dex 文件在 Java 中的表现形式，而它的 loadClassBinaryName 方法则是最后调用了 JNI 方法去完成在 dex 文件在加载 Class 对象。

BAT交流群：892872246

```
private static native Class defineClassNative(String name, ClassLoader loader, Object cookie,
                                              DexFile dexFile)
        throws ClassNotFoundException, NoClassDefFoundError;
```

DexClassLoader 和 PathClassLoader

•

DexClassLoader 和 PathClassLoader 都是 BaseDexClassLoader 的子类，他们的实现也很简单，只是构造方法传入了不同的参数而已：

DexClassLoader ：

```
public DexClassLoader(String dexPath, String optimizedDirectory,
        String librarySearchPath, ClassLoader parent) {
    super(dexPath, new File(optimizedDirectory), librarySearchPath, parent);
}
```

PathClassLoader：

```
public PathClassLoader(String dexPath, String librarySearchPath, ClassLoader parent) {
    super(dexPath, null, librarySearchPath, parent);
}
```

可以看到，DexClassLoader 和 PathClassLoader 的区别就是，PathClassLoader 的第二个参数传为 NULL，回到 BaseDexClassLoader 中可以看到：

BAT交流群：892872246

```
/**
 * Constructs an instance.
 *
 * @param dexPath the list of jar/apk files containing classes and
 * resources, delimited by {@code File.pathSeparator}, which
 * defaults to {@code ":"} on Android
 * @param optimizedDirectory directory where optimized dex files
 * should be written; may be {@code null}
 * @param librarySearchPath the list of directories containing native
 * libraries, delimited by {@code File.pathSeparator}; may be
 * {@code null}
 * @param parent the parent class loader
 */
public BaseDexClassLoader(String dexPath, File optimizedDirectory,
        String librarySearchPath, ClassLoader parent) {
    super(parent);
    this.pathList = new DexPathList(this, dexPath, librarySearchPath, optimizedDirectory);
}
```

根据注释可以看到 optimizedDirectory 参数是用来放置 DexFile 的，那么具体是怎么回事呢，再进入 DexPathList

BAT交流群：892872246

```java
public DexPathList(ClassLoader definingContext, String dexPath,
        String librarySearchPath, File optimizedDirectory) {

    if (definingContext == null) {
        throw new NullPointerException("definingContext == null");
    }

    if (dexPath == null) {
        throw new NullPointerException("dexPath == null");
    }

    if (optimizedDirectory != null) {
        if (!optimizedDirectory.exists())  {
            throw new IllegalArgumentException(
                    "optimizedDirectory doesn't exist: "
                    + optimizedDirectory);
        }

        if (!(optimizedDirectory.canRead()
                        && optimizedDirectory.canWrite())) {
            throw new IllegalArgumentException(
                    "optimizedDirectory not readable/writable: "
                    + optimizedDirectory);
        }
    }

    this.definingContext = definingContext;

    ArrayList<IOException> suppressedExceptions = new ArrayList<IOException>();
    // save dexPath for BaseDexClassLoader
    this.dexElements = makeDexElements(splitDexPath(dexPath), optimizedDirectory,
                                       suppressedExceptions, definingContext);
```

- optimizedDirectory 被传进了 makeDexElements 方法

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
private static Element[] makeElements(List<File> files, File optimizedDirectory,
                                      List<IOException> suppressedExceptions,
                                      boolean ignoreDexFiles,
                                      ClassLoader loader) {
    Element[] elements = new Element[files.size()];
    int elementsPos = 0;
    /*
     * Open all files and load the (direct or contained) dex files
     * up front.
     */
    for (File file : files) {
        File zip = null;
        File dir = new File("");
        DexFile dex = null;
        String path = file.getPath();
        String name = file.getName();

        if (path.contains(zipSeparator)) {
            String split[] = path.split(zipSeparator, 2);
            zip = new File(split[0]);
            dir = new File(split[1]);
        } else if (file.isDirectory()) {
            // We support directories for looking up resources and native libraries.
            // Looking up resources in directories is useful for running libcore tests.
            elements[elementsPos++] = new Element(file, true, null, null);
        } else if (file.isFile()) {
            if (!ignoreDexFiles && name.endsWith(DEX_SUFFIX)) {
                // Raw dex file (not inside a zip/jar).
                try {
                    dex = loadDexFile(file, optimizedDirectory, loader, elements);
                } catch (IOException suppressed) {
```

又被传进了 loadDexFile

```
    */
    private static DexFile loadDexFile(File file, File optimizedDirectory, ClassLoader loader,
                                       Element[] elements)
            throws IOException {
        if (optimizedDirectory == null) {
            return new DexFile(file, loader, elements);
        } else {
            String optimizedPath = optimizedPathFor(file, optimizedDirectory);
            return DexFile.loadDex(file.getPath(), optimizedPath, 0, loader, elements);
        }
    }
```

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```java
private static String optimizedPathFor(File path,
        File optimizedDirectory) {
    /*
     * Get the filename component of the path, and replace the
     * suffix with ".dex" if that's not already the suffix.
     *
     * We don't want to use ".odex", because the build system uses
     * that for files that are paired with resource-only jar
     * files. If the VM can assume that there's no classes.dex in
     * the matching jar, it doesn't need to open the jar to check
     * for updated dependencies, providing a slight performance
     * boost at startup. The use of ".dex" here matches the use on
     * files in /data/dalvik-cache.
     */
    String fileName = path.getName();
    if (!fileName.endsWith(DEX_SUFFIX)) {
        int lastDot = fileName.lastIndexOf(".");
        if (lastDot < 0) {
            fileName += DEX_SUFFIX;
        } else {
            StringBuilder sb = new StringBuilder(lastDot + 4);
            sb.append(fileName, 0, lastDot);
            sb.append(DEX_SUFFIX);
            fileName = sb.toString();
        }
    }

    File result = new File(optimizedDirectory, fileName);
    return result.getPath();
}
```

可以看到，如果 optimizedDirectory 为 NULL，则会以原来的路径创建 DexFile，否则会以 optimizedDirectory 为路径创建 DexFile

其实 optimizedDirectory 是要求一个内部路径的，因为动态加载去加载的可执行文件一定要存放在内部存储。而 DexClassLoader 可以指定 optimizedDirectory，所以它可以加载外部的 dex，并且这个 dex 会被复制到内部路径的 optimizedDirectory；而 PathClassLoader 没有 optimizedDirectory，所以它只能加载内部路径的 dex，也就是存在于已经安装过的 apk 里面的。

第三节、Java SPI 机制

BAT交流群：892872246

一、SPI 机制定义

SPI 机制（Service Provider Interface)其实源自服务提供者框架（Service Provider Framework，参考【EffectiveJava】page6)，是一种将服务接口与服务实现分离以达到解耦、大大提升了程序可扩展性的机制。引入服务提供者就是引入了 spi 接口的实现者，通过本地的注册发现获取到具体的实现类，轻松可插拔。

二、典型实例：jdbc 的设计

通常各大厂商（如 Mysql、Oracle）会根据一个统一的规范(java.sql.Driver)开发各自的驱动实现逻辑。客户端使用 jdbc 时不需要去改变代码，直接引入不同的 spi 接口服务即可。

Mysql 的则是 com.mysql.jdbc.Drive,Oracle 则是

oracle.jdbc.driver.OracleDriver

伪代码如下:

```
    //注:从 jdbc4.0 之后无需这个操作,spi 机制会自动找到相关的驱动实现

    //Class.forName(driver);



    //1.getConnection()方法，连接 MySQL 数据库。有可能注册了多个 Driver，这里
通过遍历成功连接后返回。

    con = DriverManager.getConnection(mysqlUrl,user,password);

    //2.创建 statement 类对象，用来执行 SQL 语句！！

    Statement statement = con.createStatement();

    //3.ResultSet 类，用来存放获取的结果集！！
```

```
    ResultSet rs = statement.executeQuery(sql);
```

jdbc 连接源码分析

1. java.sql.DriverManager 静态块初始执行，其中使用 spi 机制加载 jdbc 具体实现

```
//java.sql.DriverManager.java

//当调用 DriverManager.getConnection(..)时，static 会在 getConnection(..)执行之前被触发执行

    /**

     * Load the initial JDBC drivers by checking the System property

     * jdbc.properties and then use the {@code ServiceLoader} mechanism

     */

    static {

        loadInitialDrivers();

        println("JDBC DriverManager initialized");

    }
```

2.loadInitialDrivers()中完成了引入的数据库驱动的查找以及载入，本示例只引入了 oracle 厂商的 mysql，我们具体看看。

//java.util.serviceLoader.java

```
    private static void loadInitialDrivers() {

        String drivers;

        try {
```

BAT/2246

```
            drivers = AccessController.doPrivileged(new PrivilegedAction<
String>() {

                public String run() {

                //使用系统变量方式加载

                    return System.getProperty("jdbc.drivers");

                }

            });

        } catch (Exception ex) {

            drivers = null;

        }

        //如果 spi 存在将使用 spi 方式完成提供的 Driver 的加载

        // If the driver is packaged as a Service Provider, load it.

        // Get all the drivers through the classloader

        // exposed as a java.sql.Driver.class service.

        // ServiceLoader.load() replaces the sun.misc.Providers()



        AccessController.doPrivileged(new PrivilegedAction<Void>() {

            public Void run() {//查找具体的 provider,就是在 META-INF/service
s/***.Driver 文件中查找具体的实现。

                ServiceLoader<Driver> loadedDrivers = ServiceLoader.load
(Driver.class);

                Iterator<Driver> driversIterator = loadedDrivers.iterator
();



                /* Load these drivers, so that they can be instantiated.
```

```
                 * It may be the case that the driver class may not be the
re

                 * i.e. there may be a packaged driver with the service cl
ass

                 * as implementation of java.sql.Driver but the actual cla
ss

                 * may be missing. In that case a java.util.ServiceConfigu
rationError

                 * will be thrown at runtime by the VM trying to locate

                 * and load the service.

                 *

                 * Adding a try catch block to catch those runtime errors

                 * if driver not available in classpath but it's

                 * packaged as service and that service is there in classp
ath.

                 */

                 //查找具体的实现类的全限定名称

                try{

                    while(driversIterator.hasNext()) {

                        driversIterator.next();//加载并初始化实现类

                    }

                } catch(Throwable t) {

                // Do nothing

                }

                return null;
```

```
            }

        });


        println("DriverManager.initialize: jdbc.drivers = " + drivers);


        if (drivers == null || drivers.equals("")) {

            return;

        }

        String[] driversList = drivers.split(":");....

        }

    }
```

3.java.util.ServiceLoader 加载 spi 实现类.

上一步的核心代码如下，我们接着分析：

```
//java.util.serviceLoader.java

ServiceLoader<Driver> loadedDrivers = ServiceLoader.load(Driver.class);I
terator<Driver> driversIterator = loadedDrivers.iterator();try{

  //查找具体的实现类的全限定名称

     while(driversIterator.hasNext()) {

     //加载并初始化实现

         driversIterator.next();

     }

 } catch(Throwable t) {

 // Do nothing
```

```
}
```

主要是通过 ServiceLoader 来完成的,我们按照执行顺序来看看 ServiceLoader 实现：

```
//初始化一个 ServiceLoader,load 参数分别是需要加载的接口 class 对象,当前类加载器

    public static <S> ServiceLoader<S> load(Class<S> service) {

        ClassLoader cl = Thread.currentThread().getContextClassLoader();

        return ServiceLoader.load(service, cl);

    }

    public static <S> ServiceLoader<S> load(Class<S> service,

                                            ClassLoader loader)

    {

        return new ServiceLoader<>(service, loader);

    }
```

遍历所有存在的 service 实现

```
        public boolean hasNext() {

            if (acc == null) {

                return hasNextService();

            } else {

                PrivilegedAction<Boolean> action = new PrivilegedAction<Boolean>() {

                    public Boolean run() { return hasNextService(); }

                };
```

```
            return AccessController.doPrivileged(action, acc);
        }
    }
//写死的一个目录
    private static final String PREFIX = "META-INF/services/";

    private boolean hasNextService() {
        if (nextName != null) {
            return true;
        }
        if (configs == null) {
            try {
                String fullName = PREFIX + service.getName();
                //通过相对路径读取 classpath 中 META-INF 目录的文件，也就是
读取服务提供者的实现类全限定名
                if (loader == null)
                    configs = ClassLoader.getSystemResources(fullNam
e);
                else
                    configs = loader.getResources(fullName);
            } catch (IOException x) {
                fail(service, "Error locating configuration files",
x);
            }
```

```
        }
        //判断是否读取到实现类全限定名,比如 mysql 的“com.mysql.jdbc.Drive
r
”
        while ((pending == null) || !pending.hasNext()) {
            if (!configs.hasMoreElements()) {
                return false;
            }
            pending = parse(service, configs.nextElement());
        }
        nextName = pending.next();//nextName 保存,后续初始化实现类使用
        return true;//查到了 返回 true，接着调用 next()
    }
    public S next() {
        if (acc == null) {//用来判断 serviceLoader 对象是否完成初始化
            return nextService();
        } else {
            PrivilegedAction<S> action = new PrivilegedAction<S>() {
                public S run() { return nextService(); }
            };
            return AccessController.doPrivileged(action, acc);
        }
    }
```

```
    private S nextService() {

        if (!hasNextService())

            throw new NoSuchElementException();

        String cn = nextName;//上一步找到的服务实现者全限定名

        nextName = null;

        Class<?> c = null;

        try {

        //加载字节码返回 class 对象.但并不去初始化(换句话就是说不去执行这个
类中的 static 块与 static 变量初始化)

        //

            c = Class.forName(cn, false, loader);

        } catch (ClassNotFoundException x) {

            fail(service,

                 "Provider " + cn + " not found");

        }

        if (!service.isAssignableFrom(c)) {

            fail(service,

                 "Provider " + cn  + " not a subtype");

        }

        try {

            //初始化这个实现类.将会通过 static 块的方式触发实现类注册到 Dri
verManager(其中组合了一个 CopyOnWriteArrayList 的 registeredDrivers 成员变量)
中

            S p = service.cast(c.newInstance());
```

```
                providers.put(cn, p);//本地缓存 （全限定名，实现类对象）

                return p;

            } catch (Throwable x) {

                fail(service,

                     "Provider " + cn + " could not be instantiated",

            throw new Error();          // This cannot happen

        }
```

上一步中，Sp = service.cast(c.newInstance()) 将会导致具体实现者的初始化，

比如 mysqlJDBC，会触发如下代码：

```
//com.mysql.jdbc.Driver.java......

    private final static CopyOnWriteArrayList<DriverInfo> registeredDriv
ers = new CopyOnWriteArrayList<>();......



    static {

        try {

             //并发安全的想一个 copyOnWriteList 中方

            java.sql.DriverManager.registerDriver(new Driver());

        } catch (SQLException E) {

            throw new RuntimeException("Can't register driver!");

        }

    }
```

BAT交流群

4.最终 Driver 全部注册并初始化完毕，开始执行

DriverManager.getConnection(url, “root”, “root”)方法并返回。

使用实例

四个项目:spiInterface、spiA、spiB、spiDemo

spiInterface 中定义了一个 com.zs.IOperation 接口。

spiA、spiB 均是这个接口的实现类，服务提供者。

spiDemo 作为客户端,引入 spiA 或者 spiB 依赖，面向接口编程，通过 spi 的方式获取具体实现者并执行接口方法。

```
├─spiA
│  └─src
│      ├─main
│      │  ├─java
│      │  │  └─com
│      │  │      └─zs
│      │  ├─resources
│      │  │  └─META-INF
│      │  │      └─services
│      │  └─webapp
│      │      └─WEB-INF
│      └─test
│          └─java
```

BA 2246

```
├─spiB
│  └─src
│      ├─main
│      │  ├─java
│      │  │  └─com
│      │  │      └─zs
│      │  ├─resources
│      │  │  └─META-INF
│      │  │      └─services
│      │  └─webapp
│      │      └─WEB-INF
│      └─test
│          └─java
├─spiDemo
│  └─src
│      ├─main
│      │  ├─java
│      │  │  └─com
│      │  │      └─zs
│      │  ├─resources
│      │  └─webapp
│      │      └─WEB-INF
│      └─test
```

```
│         └─java
└─spiInterface
    └─src
        ├─main
        │  ├─java
        │  │  └─com
        │  │      └─zs
        │  ├─resources
        │  └─webapp
        │      └─WEB-INF
        └─test
            └─java
                └─spiInterface
```

spiDemo

```
package com.zs;

import java.sql.Connection;import java.sql.DriverManager;import java.sql.
ResultSet;import java.sql.SQLException;import java.sql.Statement;import
java.util.Iterator;import java.util.ServiceLoader;

public class Launcher {



    public static void main(String[] args) throws Exception {//      jdbc
Test();

        showSpiPlugins();
```

```
    }

    private static void jdbcTest() throws SQLException {

        String url = "jdbc:mysql://localhost:3306/test";

        Connection conn = DriverManager.getConnection(url, "root", "root
");

        Statement statement = conn.createStatement();

        ResultSet set = statement.executeQuery("select * from test.user
");

        while (set.next()) {

            System.out.println(set.getLong("id"));

            System.out.println(set.getString("userName"));

            System.out.println(set.getInt("age"));

        }

    }

    private static void showSpiPlugins() {

        ServiceLoader<IOperation> operations = ServiceLoader.load(IOpera
tion.class);

        Iterator<IOperation> operationIterator = operations.iterator();


        while (operationIterator.hasNext()) {

            IOperation operation = operationIterator.next();

            System.out.println(operation.operation(6, 3));

        }
```

```
}}
```

SPI 示例 完整代码。

第四节、AOP&IOC

一、AOP（面向切面编程）、Filter（过虑器）、Interceptor（拦截器）

1.1 AOP（面向切面编程）

面向切面编程（AOP 是 Aspect Oriented Program 的首字母缩写），我们知道，面向对象的特点是继承、多态和封装。而封装就要求将功能分散到不同的对象中去，这在软件设计中往往称为职责分配。实际上也就是说，让不同的类设计不同的方法。这样代码就分散到一个个的类中去了。这样做的好处是降低了代码的复杂程度，使类可重用。

但是人们也发现，在分散代码的同时，也增加了代码的重复性。什么意思呢？比如说，我们在两个类中，可能都需要在每个方法中做日志。按面向对象的设计方法，我们就必须在两个类的方法中都加入日志的内容。也许他们是完全相同的，但就是因为面向对象的设计让类与类之间无法联系，而不能将这些重复的代码统一起来。

也许有人会说，那好办啊，我们可以将这段代码写在一个独立的类独立的方法里，然后再在这两个类中调用。但是，这样一来，这两个类跟我们上面提到的独立的类就有耦合了，它的改变会影响这两个类。那么，有没有什么办法，能让我们在需要的时候，随意地加入代码呢？这种在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。

一般而言，我们管切入到指定类指定方法的代码片段称为切面，而切入到哪些类、哪些方法则叫切入点。有了 AOP，我们就可以把几个类共有的代码，抽取到一个切片中，等到需要时再切入对象中去，从而改变其原有的行为。这样看来，AOP 其实只是 OOP 的补充而已。OOP 从横向上区分出一个个的类来，而 AOP 则从纵向上向对象中加入特定的代码。有了 AOP，OOP 变得立体了。如果加上时间维度，AOP 使 OOP 由原来的二维变为三维了，由平面变成立体了。从技术

BAT交 ,/2246

上来说，AOP 基本上是通过代理机制实现的。 AOP 在编程历史上可以说是里程碑式的，对 OOP 编程是一种十分有益的补充。

AOP 不一定都像 Spring AOP 那样，是在运行时生成代理对象来织入的，还可以在编译期、类加载期织入，比如 AspectJ。AOP 像 OOP 一样，只是一种编程范式，AOP 并没有规定说，实现 AOP 协议的代码，要用什么方式去实现。

什么是代理模式？
就是再生成一个代理类，去代理 UserController 的 saveUser()方法，代码大概就长这样：

```
class UserControllerProxy {

    private UserController userController;



    public void saveUser() {

        checkAuth();

        userController.saveUser();

    }}
```

代理分为静态代理和动态代理。
静态代理，顾名思义，就是你自己写代理对象。
动态代理，则是在运行期，生成一个代理对象。Spring AOP 就是基于动态代理的，但是不是所有 AOP 的实现都是在运行时进行织入的，因为这样效率太低了，而且只能针对方法进行 AOP，无法针对构造函数、字段进行 AOP。我完全可以在编译成 class 时就织入啊，比如 AspectJ，当然 AspectJ 还提供了后编译器织入和类加载期织入。

1.1.1 AOP 概念

方面（Aspect）：一个关注点的模块化，这个关注点实现可能另外横切多个对象。事务管理是 J2EE 应用中一个很好的横切关注点例子。方面用 Spring 的 Advisor 或拦截器实现。

连接点（Joinpoint）：程序执行过程中明确的点，如方法的调用或特定的异常被抛出。

通知（Advice）：在特定的连接点，AOP 框架执行的动作。各种类型的通知包括“around”、“before”和“throws”通知。通知类型将在下面讨论。许多 AOP 框架包括 Spring 都是以拦截器做通知模型，维护一个“围绕”连接点的拦截器链。

切入点（Pointcut）：指定一个通知将被引发的一系列连接点的集合。AOP 框架必须允许开发者指定切入点，例如，使用正则表达式。

引入（Introduction）：添加方法或字段到被通知的类。Spring 允许引入新的接口到任何被通知的对象。例如，你可以使用一个引入使任何对象实现 IsModified 接口，来简化缓存。

目标对象（Target Object）：包含连接点的对象，也被称作被通知或被代理对象。

AOP 代理（AOP Proxy）：AOP 框架创建的对象，包含通知。在 Spring 中，AOP 代理可以是 JDK 动态代理或 CGLIB 代理。

编织（Weaving）：组装方面来创建一个被通知对象。这可以在编译时完成（例如使用 AspectJ 编译器），也可以在运行时完成。Spring 和其他纯 Java AOP 框架一样，在运行时完成织入。

BAT交流群：/2246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

1.1.2 通知类型

Around 通知：包围一个连接点的通知，如方法调用。这是最强大的通知。Aroud 通知在方法调用前后完成自定义的行为，它们负责选择继续执行连接点或通过返回它们自己的返回值或抛出异常来短路执行。

Before 通知：在一个连接点之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。

Throws 通知：在方法抛出异常时执行的通知。Spring 提供强制类型的 Throws 通知，因此你可以书写代码捕获感兴趣的异常（和它的子类），不需要从 Throwable 或 Exception 强制类型转换。

After returning 通知：在连接点正常完成后执行的通知，例如，一个方法正常返回，没有抛出异常。

Around 通知是最通用的通知类型。大部分基于拦截的 AOP 框架（如 Nanning 和 Jboss 4）只提供 Around 通知。

如同 AspectJ，Spring 提供所有类型的通知，我们推荐你使用最为合适的通知类型来实现需要的行为。例如，如果只是需要用一个方法的返回值来更新缓存，你最好实现一个 after returning 通知，而不是 around 通知，虽然 around 通知也能完成同样的事情。使用最合适的通知类型使编程模型变得简单，并能减少潜在错误。例如，你不需要调用在 around 通知中所需使用的 MethodInvocation 的 proceed()方法，因此就调用失败。

切入点的概念是 AOP 的关键，它使 AOP 区别于其他使用拦截的技术。切入点使通知独立于 OO 的层次选定目标。例如，提供声明式事务管理的 around 通知可以被应用到跨越多个对象的一组方法上。 因此切入点构成了 AOP 的结构要素。

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

1.2 Filter（过虑器）

Filter 技术是 Servlet2.3 新增加的功能，Servlet2.3 是 sun 公司于 2000 年 10 月发布的，它的开发者包括许多个人和公司团体，充分体现了 sun 公司所倡导的代码开放性原则。在众多参与者的共同努力下，Servlet2.3 比以往功能都强大了很多，而且性能也有了提高。那么 Filter 有什么功能呢？Filter 可以用来设置字符集、控制权限、控制转向等等。简单点说：就是定义在访问某对象之前和访问之后要做的事情。

1.3Interceptor（拦截器）

1，拦截器的概念

java 里的拦截器是动态拦截 Action 调用的对象，它提供了一种机制可以使开发者在一个 Action 执行的前后执行一段代码，也可以在一个 Action 执行前阻止其执行，同时也提供了一种可以提取 Action 中可重用部分代码的方式。在 AOP 中，拦截器用于在某个方法或者字段被访问之前，进行拦截然后再之前或者之后加入某些操作。目前，我们需要掌握的主要是 Spring 的拦截器，Struts2 的拦截器不用深究，知道即可。

2，拦截器的原理

大部分时候，拦截器方法都是通过代理的方式来调用的。Struts2 的拦截器实现相对简单。当请求到达 Struts2 的 ServletDispatcher 时，Struts2 会查找配置文件，并根据配置实例化相对的拦截器对象，然后串成一个列表（List），最后一个一个的调用列表中的拦截器。Struts2 的拦截器是可插拔的，拦截器是 AOP 的一个实现。Struts2 拦截器栈就是将拦截器按一定的顺

BA /2246

序连接成一条链。在访问被拦截的方法或者字段时，Struts2 拦截器链中的拦截器就会按照之前定义的顺序进行调用。

3，自定义拦截器的步骤

第一步：自定义一个实现了 Interceptor 接口的类，或者继承抽象类

AbstractInterceptor。

第二步：在配置文件中注册定义的拦截器。

第三步：在需要使用 Action 中引用上述定义的拦截器，为了方便也可以将拦截器定义为默认的拦截器，这样在不加特殊说明的情况下，所有的Action 都被这个拦截器拦截。

4，过滤器与拦截器的区别

过滤器可以简单的理解为“取你所想取”，过滤器关注的是 web 请求；拦截器可以

简单的理解为“拒你所想拒”，拦截器关注的是方法调用，比如拦截敏感词汇。

4.1，拦截器是基于 java 反射机制来实现的，而过滤器是基于函数回调来实现的。（有人说，拦截器是基于动态代理来实现的）

4.2，拦截器不依赖 servlet 容器，过滤器依赖于 servlet 容器。

4.3，拦截器只对 Action 起作用，过滤器可以对所有请求起作用。

4.4，拦截器可以访问 Action 上下文和值栈中的对象，过滤器不能。

4.5，在 Action 的生命周期中，拦截器可以多次调用，而过滤器只能在容器初始化时调用一次。

,/2246

5，Spring 拦截器

5.1，抽象类 HandlerInterceptorAdapter

我们如果在项目中使用了 Spring 框架，那么，我们可以直接继承

HandlerInterceptorAdapter.java 这个抽象类，来实现我们自己的拦截器。

Spring 框架，对 java 的拦截器概念进行了包装，这一点和 Struts2 很类似。HandlerInterceptorAdapter 继承了抽象接口 HandlerInterceptor。

二、Android IOC 注入框架

2.1 什么是 IOC 注入框架

ButterKnife 大家都应该使用过，对于 view 的注入减少了大量篇幅的 findViewById 操作，而注解注入的方式也显得更加**优雅**。这里介绍一下我的 IOC 简单注入框架，项目地址移步这里

2.2 IOC 使用简单介绍

添加依赖

项目根目录下的 build.gradle 文件添加如下内容

```
allprojects {

    repositories {

        ...

        maven { url "https://raw.githubusercontent.com/demoless/ioc/master/repo" }

    }

}
```

然后在 app 模块的 build.gradle 文件添加如下内容

```
implementation 'com.demoless:ioc:1.0.0'
```

这里我贴出 demo 的调用示例代码看看如何使用：

要实现这样一套 IOC 框架我们还要先注册一下，看 BaseActiivty 的代码：

可以看到相比于传统的 Activity 的写法，IOC 注入框架颇具诱惑，下面我就带大家了解一下我的 IOC 实现思路。

如何实现 IOC

IOC 是一套注解注入框架，所以主要是通过 Java 的反射与注解来实现的，这里就不介绍了，不了解的可以看看这篇文章。

布局注入

```
@ContentView（R.layout.activity_main）
```

首先创建 ContentView 这个注解

创建过程跟创建类的过程是一样的，只需要将 Kind 选择为 Annotation 即可：

BAT交流群：892872246

注解的编写

```
package com.demo.iocinject.ioc;



import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

/**

 * Create by Zhf on 2019/7/13

 **/@Target(ElementType.TYPE)@Retention(RetentionPolicy.RUNTIME)

public @interface ContentView {
```

```
    int value();}
```

我们可以看到在这个注解的上面还有两个注解，这是两个元注解，首先@Target(ElementType.TYPE)代表这个 ContentView 注解作用在类上面，然后@Retention(RetentionPolicy.RUNTIME)表示注解在运行时执行，因为一个类只会有一个布局文件，所以这里 value 方法的返回值为 int，而不是数组。

Java 实现注解的执行逻辑

定义好了这个注解之后，我们就要考虑如何将 ContentView 里传入的布局文件设置给 Activity，我们知道传统的 activity 是通过在 onCreate 方法里的 setContentView 来将布局文件设置给 activity 的，那我们也只需要通过反射将传入注解的布局再传入 setContentView 并且让它自动执行不就可以实现了嘛，思路好像没错，我们来实现以下：

```
//布局注入

    private static void injectLayout(Activity activity) {

        Class<? extends Activity> clazz = activity.getClass();

        //获取类之上的注解

        ContentView contentView = clazz.getAnnotation(ContentView.clas
s);

        if (contentView != null){

            //获取注解的返回值

            int layoutId = contentView.value();



            //第一种方法
```

```
            //activity.setContentView(layoutId);



           //第二种方法

            try {

                Method setContentView = clazz.getMethod("setContentView",
int.class);

                setContentView.invoke(activity,layoutId);

            } catch (Exception e) {

                e.printStackTrace();

            }

        }



    }
```

这段代码首先通过 activity.getClass()拿到这个 Class 对象，再通过 clazz.getAnnotation(ContentView.class)获取类上的注解，拿到注解之后自然要获取他的返回值，所以再调用他的 value 方法，这样我们就拿到了对应的布局文件，最好要完成的是传入 setContentView 这个布局方法并执行。这里我给出了两种方法，第一种很简单直接调用 activity 的 setContentView 方法，第二种通过反射拿到 setContentView 这个 Method 对象，在调用 invoke 方法自动执行。这样一个简易布局注入就实现了。

BAT交流群：/2246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

控件注入

@InjectView

注解文件

与 ContentView 一样这里就不在赘述了：

```
/**
 * Create by Zhf on 2019/7/13
 **/
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface InjectView {
    int value();
}
```

Java 执行逻辑

```
//控件的注入

    private static void injectViews(Activity activity) {

        Class<? extends Activity> clazz = activity.getClass();



        //获取类的全部属性

        Field[] fields = clazz.getDeclaredFields();

        for (Field field : fields) {



            //获取属性上的注解

            InjectView injectView = field.getAnnotation(InjectView.clas
s);
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
            if (injectView != null){


                //获取注解的值

                int viewId = injectView.value();


                //View view = activity.findViewById(viewId);

                try {

                    //获取 findViewById 方法

                    Method findViewById = clazz.getMethod("findViewById",
int.class);


                    //执行 findViewById 方法

                    Object view = findViewById.invoke(activity, viewId);


                    //设置访问权限 private

                    field.setAccessible(true);

                    //为属性赋值

                    field.set(activity,view);

                } catch (Exception e) {

                    e.printStackTrace();

                }

            }

        }
```

```
    }
```

这段代码也跟布局注入的实现很相似，这是回去类的全部属性的时候，需要调用的是 clazz.getDeclaredFields()方法，如果用 getFields 方法程序会崩溃，这个很容易想到，因为在父类和子类中可能会有相同命名的一个控件，就行这样：**private Button mButton；**，所以就造成崩溃，这只能获取类自身的属性；第二个需要注意的地方是，通常我们声明一个控件，都是用的 private 关键字，所以这里还需要设置一下访问权限，调用 field.setAccessible(true),这样对稀有属性进行操作;另一个与布局注入实现不同的是，setContentView 方法没有返回值，而 findViewById 则相反，所以我们需要为属性（这里就是一些 View）赋值，调用的是 field.set(activity，view)。

事件的注入

@InjectEvent

Android 事件监听规律

事件的注入相比之前的布局和控件注入，难度和复杂度大大提高了。通过对 Android 中的事件监听代码的观察，我们得出如下三部曲：

- setListener
- new Listener
- doCallback

就像 View 的点击事件和长按时间监听那样，首先 setListener：View.setOnClickListener()，然后 new 一个 Listener 传入，

BAT交流群：892872246

View.setOnClickListener(new OnClickListener(View v){}),最后执行回调方法：

onClick（View v）{...}

定义事件监听规律的注解

@EventBase

通过上述规律总结，我们要先定义这个注解：

```
/**
 * Create by Zhf on 2019/7/13
 **/
@Target(ElementType.ANNOTATION_TYPE)
@Retention(RetentionPolicy.RUNTIME)public @interface EventBase {

    //setListener
    Class<?> listenerType();

    //new View.OnxxxListener
    String listenerSetter();

    //回调 最终执行方法
    String callBackListener();}
```

我们看到这个注解是放在注解类之上的，那么这个注解怎么使用呢，就以 View 的长按事件监听为例：

```
/**
```

```
 * Create by Zhf on 2019/7/13

 **/

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

@EventBase(listenerSetter = "setOnLongClickListener",

        listenerType = View.OnLongClickListener.class,

        callBackListener = "onLongClick")public @interface OnLongClick {

    int[] value();}
```

在这个注解上面调用了刚才定义的 EventBase 注解，根据传入的值大家似乎就什么都看明白了吧，没错这里传入了 View.OnLongClickListener 事件监听三部曲，因为在一个类中可能不止一个控件会设置长按事件监听，所以这里的返回值是数组。

事件注入的逻辑

```
    //事件的注入

    private static void injectEvents(Activity activity) {

        Class<? extends Activity> clazz = activity.getClass();



        //获取一个类的所有方法

        Method[] methods = clazz.getDeclaredMethods();

        //遍历所有方法

        for (Method method : methods) {

            Annotation[] annotations = method.getAnnotations();
```

BAT/892246

```
            //遍历所有注解

            for (Annotation annotation : annotations) {

                Class<? extends Annotation> annotationType = annotation.a
nnotationType();

                if (annotationType != null) {

                    EventBase eventBase = annotationType.getAnnotation(Eve
ntBase.class);

                    if (eventBase != null) {

                        String listenerSetter = eventBase.listenerSetter
();

                        Class<?> listenerType = eventBase.listenerType();

                        String callBackListener = eventBase.callBackListen
er();

                        try {

                            Method valueMethod = annotationType.getDeclare
dMethod("value");



                            int[] viewIds = (int[]) valueMethod.invoke(ann
otation);



                            //设置 private 权限可见

                            method.setAccessible(true);



                            //AOP 切面

                            ListenerInvocationHandler handler = new Listen
erInvocationHandler(activity);
```

```
                        handler.addMethods(callBackListener, method);


                        //代理模式
                        Object listener = Proxy.newProxyInstance(liste
nerType.getClassLoader(),
                                new Class[]{listenerType}, handler);


                        for (int viewId : viewIds) {


                            View view = activity.findViewById(viewId);


                            Method setter = view.getClass().getMethod(l
istenerSetter, listenerType);


                            setter.invoke(view, listener);


                        }
                    } catch (Exception e) {
                        e.printStackTrace();
                    }


                }
            }
```

```
            }

        }

    }
```

这里其他的不多介绍了，主要不同的就是这里使用了动态代理和 AOP 切面技术：

```
import android.util.Log;

import java.lang.reflect.InvocationHandler;import java.lang.reflect.Meth
od;import java.util.HashMap;

/**

 * Create by Zhf on 2019/7/13

 **/public class ListenerInvocationHandler implements InvocationHandler
{



    private final static long QUICK_EVENT_TIME_SPAN = 300;

    private long lastClickTime;



    private Object target;//需要拦截的对象



    private HashMap<String, Method> map = new HashMap<>();



    public ListenerInvocationHandler(Object target){

        this.target = target;

    }
```

```
    @Override

    public Object invoke(Object proxy, Method method, Object[] args) thro
ws Throwable {

        if (target != null){

            String methodName = method.getName();

            method = map.get(methodName);



            long timeSpan = System.currentTimeMillis() - lastClickTime;

            if (timeSpan < QUICK_EVENT_TIME_SPAN){

                Log.e("点击阻塞，防止误点", String.valueOf(timeSpan));

                return null;

            }

            lastClickTime = System.currentTimeMillis();

            if (method != null){

                if (method.getGenericParameterTypes().length == 0) return
 method.invoke(target);

                return method.invoke(target,args);

            }

        }

        return null;}



    public void addMethods(String methodName, Method method){

        map.put(methodName, method);
```

```
    }}
```

这个类实现了 InvocationHandler 接口，可以实现点击事件不传参数以及点击阻塞，防误点，具体的逻辑比较简单，可以看看代码以及注释。

第五节、手写组件化架构

一、此次手写架构，解决的问题是：

1、让 App 内 各个功能模块能够独立开发单元测试，也可以 所有模块集成打包，统一测试

独立开发

更改 gradle.properties 的配置，使得每个功能模块都成为 application， 可以独立打包成 apk，单独运行。单个模块，独立测试。

集成打包

更改 gradle.properties 的配置，使得原先每个单独模块，都变成 library，被 主模块引用，这时候只有主模块能够打包 apk，所有功能都集成在这个 apk 内。

2、实现 功能模块的整体移植，灵活拔插

故事背景

当你们公司有多个安卓开发人员，开发出核心业务相同，但是 UI 不

BAT交流群：892872246

同，其他业务不同的一系列 App 时（如果核心业务是 X，你们有 5 个开发人员，做出了 A,B,C,D,E 5 个 app，都包含核心业务 X，但是除了 X 之外，其他的业务模块各不相同）这时候，如果领导要把 A 里面的一个非核心功能，挪到 B 里面...

现状

开发 B 的程序猿可能要骂娘,因为他在从移植 A 的代码中剥离代码 遇到了很多高耦合，低内聚 的类结构，挪过来之后，牵一发而动全身，动一点小地方，整个代码满江红。

理想

如果这个时候,我们通过代码框架的配置,能够把 A 里面的一个模块，作为一个 module 移植到 工程内部，然后主 module 来引用这个 module，略微写一些代码来使得这个功能模块在 app 中生效。那么无论是多少个功能模块，都可以作为整体来 给其他 app 复用。这样开发人员也不用相互骂娘了，如果挪过来的模块存在 bug 或者其他问题，也不用甩锅，模块原本是谁开发的，找谁就好了。

3、保证 App 内 业务模块的相互隔离，但是又不妨碍业务模块之间的数据交互

我们开发 app 的功能模块，一个业务，可能是通过一个 Activity 或者 一个 Fragment 作为对外的窗口,也可能是。所谓窗口，就是这个业务，相对于其他模块，"有且只有"一个入口，没有任何其他可以触达到这个业务的途径。业务代码之间相互隔离，绝对不可以有相互引用。那么，既然相互不会引用，那 A 模块一定要用到 B 模块的数据，

BAT交流群：892872246

怎么办呢？下文提供解决方案。

二、正文大纲

1、代码结构现状以及理想状态一览

2、功能组件化的实现思路，实现组件移植拔插

3、参考 ARouter 源码，写出自己的 Router 框架，统一通过 Router 来进行模块的切换 以及 组件之间数据的交互

4、使用组件 api 化，在模块很多的情况下优化公共模块的结构

三、正文

1、代码结构现状以及理想状态一览

现状；

BAT交流群：892872246

代码有模块化的迹象，但是没有对业务模块进行非常明显的模块化（不明白啥意思是吧？不明白就对了，app 这个 module 里面其实还有很多东西没有展示出来，请看下图：试想，把所有的模块集中到一个 module 的一个包里面，当你要移植某一个功能的时候，想想那酸爽....当然如果你口味别致，那当我没说）

BAT交流群：892872246

理想：

BAT交流群：892872246

理想化的话，参照：理想.png; 项目结构层次分明，脉络清晰

按照图中的分层，详细解释一下：

外壳层：app module

内部代码只写 app 的骨骼框架，比如说，你的 app 是这个样子的结构：

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

,/2246

下方有 N 个 TAB，通过 Fragment 来进行切换模块。这种架构肯定不少见。

这个时候，外壳层 app module，就只需要写上 上面这种 UI 架构的框架代码就行了，至于有多少个模块，需要代码去读取配置进行显示。你问我怎么写这种 UI 框架吗？网上一大把的，**如果实在找不到，来我的 github 项目地址**

业务层

我们的业务模块，对外接口可能是一个 `Activity`* （比如说，登录模块，只对外提供一个 `LoginActivity`,有且仅有这一个窗口）或者 **是一个** `Fragment`，就像上图（典型的 app 架构.png），如果 app 的 UI 框架是通过切换 `Fragment` 来却换业务模块的话。用 `business`*这个目录，将所有的业务模块包含进去，每个模块又是独立的 `module`，这样既实现了业务代码隔离，又能一眼看到所有的业务模块，正所谓，一目了然。

功能组件层

每一个业务模块，不可避免的需要用到一些公用工具类，有的是**第三方 SDK 的再次封装**，有的是**自己的工具类**，或者自己写的**自定义控件**，还有可能是 **所有业务模块都需要的 辅助模块**，都放在这里。

路由框架层

设计这一层，是想让 app 内的所有 Activity，业务模块 Fragment，以及模块之间的数据交互，都由 这一层开放出去的接口来负责

gradle 统一配置文件

892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

工程内部的一些全局 gradle 变量，放在这里，整个工程都有效

module 编译设置

setting.gradle 配置要编译的 module; 也可以做更复杂的操作，比如，写 gradle 代码去自动生成一些 module，免除人为创建的麻烦.

---

2. 功能组件化的实现思路，实现组件移植拔插

能够兼顾 每个模块的单独开发，单独测试 和 整体打包，统一测试。 听起来很神奇的样子，但是其实就一个核心：gradle 编程。

打开 gradle.properties 文件：

```
#true: 集成模式运行 false: 组件模式运行
isModule=true
```

注解应该很清晰了，通过一个全局变量，就可以控制当前是要 模块化单元测试呢？还是要集成打包 apk 测试。

那么，只写一个 isModule 就完事了吗？当然不是，还有一堆杂事 需要我们处理，我们要使用这个全局变量。

一堆杂事，分为两类；

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

1- app 外壳层 module 的 build.gradle(注意：写在 dependencies)

```
if (isModule.toBoolean()) {

   implementation project(":business:activity_XXX")

   //...在这里引用更多业务模块}
```

2- 每个业务 module 的 build.gradle

第一处：判定 isModule，决定当前 module 是要当成 library 还是 application

```
if (isModule.toBoolean()) {

    apply plugin:'com.android.library'

} else {

    apply plugin:'com.android.application'*

}
```

第二处：更改 defaultConfig 里面的部分代码，为什么要改？因为当当前 module 作为 library 的时候，不能有 applicationId "XXXX"这一句

```
defaultConfig {

    if (!isModule.toBoolean()) {

       applicationId"study.hank.com.XXXX"*

    }

    ....

}
```

第三处：当业务模块 module 作为 library 的时候，不可以在 AndroidManifest.xml 中写 Launcher Activity,否则，你打包 app module 的时候，安装完毕，**手机桌面上将会出现不止一个 icon**。而，当业务模块 module 作为 application 单独运行的时候，必须有一个 Launcher Activity ，不然...launcher 都没有，你测个什么鬼 ``` 所以这里针对 manifest 文件进行区分对待。

```
sourceSets {

        main {

            if (isModule.toBoolean()) {

                manifest.srcFile 'src/main/module/AndroidManifest.xml'

            } else {

                manifest.srcFile 'src/main/AndroidManifest.xml'

            }

        }

    }
```

由于要区分对待，我们就需要另外创建一个 manifest 文件，移除 launcher 配置即可。参考下图：

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="activity_chart"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity
        android:name=".ChartActivity"
        android:screenOrientation="portrait">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

</application>
```

存在launcher

原本的manifest文件

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="study.hank.com.activity_chart">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="activity_chart"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true">
        <activity android:name=".ChartActivity">
            </activity>
    </application>

</manifest>
```

移除了原来的launcher配置

另外创建的manifest

这就是业务模块组件化的秘密了。

什么，你问我怎么 进行功能拔插吗？

当你不需要某一个模块的时候，

1）在 app 的 build.gradle 里面 把 引用该模块的配置去掉；

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
if (isModule.toBoolean()) {
    implementation project(":business:activity_chart")
    implementation project(":business:fragment_test")
    implementation project(":business:fragment_home")不要这个模块了？
    implementation project(":business:fragment_mine")
    implementation project(":business:fragment_test2")
}
```

2）setting.gradle 的 include 去掉它

```
include ':app', ':business:fragment_home', ':business:fragment_test', ':business:fragment_test2', ':store'
```

转手的小姐

3）app module 里面，改动代码，不再使用这个模块。（这个我就不截图了，因为 app module 的 UI 框架代码不是一句话说得清的。请运行我的 demo 源码自己看吧）

功能的插入，同理，上面的过程倒过来走一遍，就不浪费篇幅了。

---

3. 参考 ARouter 源码，写出自己的 Router 框架，统一通过 Router 来进行模块的切换 以及组件之间数据的交互

说到路由框架的使用价值，有两点：

1、在 app 实现了组件化之后，由于组件之间由于代码隔离，不允许相互引用，导致 相互不能直接沟通，那么，就需要一个 **"中间人角色"** 来帮忙*" **带话**"了.

BAT交流群：892872246

2、app 内部，不可避免地要进行 Activity 跳转，Fragment 切换。

把这些重复性的代码，都统一让路由来做吧。省了不少代码行数。

阅读了阿里巴巴 ARouter 的源码，参照阿里大神的主要思路，简化了一些流程，去掉了一些我不需要的功能，增加了一些我独有的功能，加入了一些自己的想法，写出了自己的 ZRouter 路由 框架。那就不罗嗦了，上干货吧。

基础知识

如果以下基础知识不具备，建议先去学习基础知识。 或者 也可以跟着笔者的思路来看代码，慢慢理解这些知识的实用价值。

java 反射机制（路由框架里大量地使用了 class 反射创建 对象）

APT 注解，注解解析机制（注解解析机制贯穿了整个路由框架）

javapoet ， java 类的元素结构（一些人为写起来很麻烦的代码，

一些脏活累活，就通过自动生成代码来解决）

如何使用

1- 在 app module 的自定义 Application 类里面，进行初始化， ZRouter 准备就绪

```
public class FTApplication extends Application {

    @Override

    public void onCreate() {

        super.onCreate();

        ZRouter.getInstance().initRegister(this);

    }}
```

2- 就绪之后才可以直接使用（RouterPathConst 里面都是我自己定义的 String 常量）：

切换 Fragment

```
ZRouter.getInstance().build(RouterPathConst.PATH_FRAGMENT_MINE).navigat
ion();
```

跳转 Activity

```
ZRouter.getInstance().build(RouterPathConst.PATH_ACTIVITY_CHART).naviga
tion();
```

组件之间的通信,取得 Mine 模块的 accountNo 然后 toast 出来

```
String accountNo = ZRouter.getInstance().navigation(MineOpenServiceApi.c
lass).accountNo();

Toast.makeText(getActivity(), accountNo, Toast.LENGTH_LONG).show();
```

如我们之前所设想的，切换 Fragment，跳转 Activity，组件之间的通信 全部只能通过 ZRouter 框架来执行。

3- 退出 app 时，要释放 ARouer 的资源（主要是静态变量）

```
ZRouter.getInstance().release();
```

4- 每个业务模块，在将要暴露出去的 Fragment 或者 Activity 上，要加上注解

```
@ZRoute(RouterPathConst.PATH_ACTIVITY_CHART)//注册 Activitypublic class C
hartActivity extends AppCompatActivity {···}
```

或者

```
@ZRoute(RouterPathConst.PATH_FRAGMENT_HOME)//注册 Fragmentpublic class Ho
meFragment extends Fragment {···}
```

或者

```
@ZRoute(RouterPathConst.PATH_PROVIDER_MINE) // 注册数据接口 public class M
ineServiceImpl implements MineOpenServiceApi  {···}
```

设计思路

讲解设计思路，必须用源码进行参照，请务必参照源码 。

源码地址为：https://github.com/18598925736/EvolutionPro

说明一下我本人 阅读源码的方法。也许很多人都和曾经的我一样，看到一份第三方 SDK 的源码，不知道从何下手，要么看了半天还在原地打转转，要么就是这里看一点，那里看一点，没有中心思想，看了半天毫无收获。

> 干货：看源码要思路清晰，目的明确。一切技术的价值都只有一个，那就是解决实际问题。既然是解决实际问题，那我们就从这个 SDK 暴露出来的最外围接口为起点，看这个接口的作用是什么，解决了什么问题，顺藤摸瓜，找找它解决问题的核心方法，至于顺藤摸瓜道路上遇到的枝枝脉脉，要分清哪些是辅助类（每个人写辅助类的习惯可能都不同，所以不必太在意），哪些是核心类

892872246

（核心思想一般都是大同小异）。找到了核心思想，再从头重新过几遍，SDK 的设计思路就了然于胸了哈.

按照我的上面提供的"干货"，如果你现在下载了我的 Demo 源码，那么我们继续：

如果把看源码的结构，理解为 警察查案。那么就要从最表层的现象开始着手，慢慢查找根源。

HomeFragment.java 的 54 行， 这里要进行 Activity 跳转。

```
ZRouter.getInstance().build(RouterPathConst.PATH_ACTIVITY_CHART).navigation();
```

这里有 getInstance()方法，build()方法，还有 navigation()方法,一个一个看

- `getInstance()`是处在 ZRouter 类内部，是 ZRouter 的单例模式的 get 方法，单例模式就不赘述了，我写了注释
- `build()`方法也是在 ZRouter 类内部，逻辑很简单，就是 `new Postcard(path)` 参数 `path` 是一个 `string`,方法返回值是一个 `Postcard` 对象
- `navigation()`方法是在 Postcard 类内部，但是，具体的执行逻辑，依然是在 `ZRouter` 类里面 `getInstance()`和 `build()`方法都很简单，不需要花太多精力。下面继续跟随 `ZRouter` 的 `navigation()`方法"追查"

`ZRouter` 的 `navigation()` 方法内容如下：

```
Object navigation(Postcard postcard) {
        LogisticsCenter.complete(postcard);
        switch (postcard.getRouteType()) {
            case ACTIVITY://如果是 Activity，那就跳吧
```

```
            return startActivity(postcard);
        case FRAGMENT://如果是 Fragment，那就切换吧
            return switchFragment(postcard);
        case PROVIDER://如果是 Provider，那就执行业务逻辑
            return postcard.getProvider();//那就直接返回 provider 对象
        default:
            break;
    }
    return null;
}
```

发现一个可疑的代码：`LogisticsCenter.complete(postcard);` 看方法名，应该是对 postcard 对象进行完善。

进去追查

```
/**
 * Postcard 字段补全
 *
 * @param postcard
 */
public static void complete(Postcard postcard) {
    if (null == postcard) {
        throw new RuntimeException("err:postcard 是空的，怎么搞的？");
    }
```

```
        RouteMeta routeMeta = Warehouse.routeMap.get(postcard.getPath());
//

        if (null == routeMeta) {//如果路由 meta 是空，说明可能这个路由没注册，
也有可能路由表没有去加载到内存中

            throw new RuntimeException("err:路由寻址失败，请检查是否 path 写
错了");

        } else {

            postcard.setDestination(routeMeta.getDestination());

            postcard.setRouteType(routeMeta.getRouteType());



            ...

        }

    }
```

这段代码，从一个 `map` 中，用 `path` 作为 `key`，`get` 出了一个 `RouteMeat` 对象，然后用这个对象的字段值，对参数 `postcard` 的属性进行赋值。好像有点莫名其妙。看不太懂。不着急，继续。

刚才的 `navigation()`方法这里存在 `switch` 分支，分支设计到 `ACTIVITY,FRAGMENT,PROVIDER`，由于我们这次追查的只是 `activity` 相关，所以，忽略掉其他分支，只追查 `startActivity(postcard);` 下面是该方法的代码：

```
private Object startActivity(Postcard postcard) {

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

        if (cls == null) {

            if (cls == null)
```

BAT文 2246

```
            throw new RuntimeException("没找到对应的 activity，请检查路由寻址标识是否写错");

        }

        final Intent intent = new Intent(mContext, cls);

        if (Postcard.FLAG_DEFAULT != postcard.getFlag()) {//如果不是初始值，也就是说，flag 值被更改过，那就用更改后的值

            intent.setFlags(postcard.getFlag());

        } else {//如果没有设定启动模式，即 flag 值没有被更改，就用常规模式启动

            intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);//常规模式启动 Activity

        }

        //跳转只能在主线程中进行

        runInMainThread(new Runnable() {

            @Override

            public void run() {

                mContext.startActivity(intent);

            }

        });

        return null;

    }
```

这里只是一个简单的跳转操作，但是，发现了一个关键点，跳转的“目的地”`class` 是来自 `postcard` 的 `destination`．发现规律了，原来刚才

在 `LogisticsCenter.complete(postcard);` 里面进行 `postcard`"完善"的时候，`set` 进去的 `destination` 原来在这里被使用到。

那么问题的关键点就发生了转移了， 这个 `destination Class` 是从 `map` 里面 `get` 出来的，那么，又是什么时候被 `put` 进去的呢？

开始追踪这个 `map`：`Warehouse.routeMap`，通过代码追踪，可以发现，唯一可能往 `map` 里面 `put` 东西的代码只有这一句：

```
    /**
     * 反射执行 APT 注册文件的注册方法
     */
    private static void registerComm() {
        try {
            Set<String> classNames = ClassUtils.getFileNameByPackageName
(mContext, RouterConst.GENERATION_PACKAGE_NAME);//找到包名下的所有 class
            for (String className : classNames) {
                Class<?> clz = Class.forName(className);
                if (IRouterZ.class.isAssignableFrom(clz)) {
                    IRouterZ iRouterComm = (IRouterZ) clz.getConstructor().
newInstance();
                    iRouterComm.onLoad(Warehouse.routeMap);
```

```
            }
        }
    } finally {
        Warehouse.traversalCommMap();
    }
}
```

利用 java 反射机制，反射创建类的实例，然后执行 `onLoad` 方法，参数，正是这个 `map`

OK,关于查看源码的详细步骤，就写到这么多，再罗嗦，大佬们要打人啦。

目前为止的结论：通过追踪 `ZRouter.getInstance().build(RouterPathConst.PATH_ACTIVITY_CHART).navigation();`

我们一路上遭遇了这些类或接口：

核心类：

ZRouter（提供 Activity 跳转的接口）；

辅助类或接口

Postcard (“明信片”，封装我们要执行操作，这次的操作是 跳 Activity）

RouteMeta (“路由参数”,Postcard 的基类)

RouteType （“路由类型”，我们要执行的操作，用枚举来进行区分）

LogisticsCenter ("物流中心"，主要封装 ZRouter 类的一些特殊逻辑，比如对 Postcard 对象进行完善补充)

BAT交流群：892872246

Warehouse (“货舱”，用 hashMap 来存储“路由”对象)

IRouterZ ("路由注册"接口类 ，用于反射创建对象，从而进行路由的注册)

上面用大量篇幅详述了 追踪源码， 追查框架结构的方法，那么下面的篇幅就直接说结论了：

路由框架的结构，可以用一张图表示：

针对这张图 简单说两句：

路由框架必然有 3 个部分，注解定义，注解解析，以及路由对外接口。

`demo` 中我把这 3 个部分定义成了 3 个 `module`.

其中，每个部分的核心代码是：

`zrouter-annotation` 模块的 `ZRoute @interface，IRouterZ` 接口

`zrouter-api` 模块的 `ZRouter` 类

BAT交流群：892872246

`zrouter-compiler` 模块的 `RouterProcessor` 类

具体的代码，不加以说明了。

如何用路由进行 `Activity` 跳转，我写了详细步骤，相信没人看不懂了。那么 `Fragment` 的切换，是我自定义的方法，可能有点粗糙，但是也是通俗易懂，就点到为止。但是，我们组件化的思想，就是要隔离所有的业务模块，彼此之间不能进行直接通信，如果 A 模块一定要使用 B 模块的一些数据，通过路由框架也能实现。

HomeFragment 类的第 72 行代码：

```
String accountNo =
ZRouter.getInstance().navigation(MineOpenServiceApi.class).accountNo()
;
```

这句代码的意义是：在 Home 模块中，通过路由框架，调用 Mine 模块对外开放的接口 accountNo();

追踪这句代码的 `navigation()` 方法，找到真正的执行逻辑 `ZRouter 类` 141 行起:

```
public <T> T navigation(String serviceName) {
        Postcard postcard = LogisticsCenter.buildProvider(serviceName);
        if (null == postcard)
            return null;
        LogisticsCenter.complete(postcard);//补全 postcard 字段值
        return (T) postcard.getProvider();
    }
```

这里：最终返回了一个 Provider 对象.

BAT交流群

`LogisticsCenter` 类又有了戏份：

`LogisticsCenter.buildProvider(serviceName)` 和 `LogisticsCenter.complete(postcard);`

分别点进去看：

`buildProvider(String)` 方法，其实就是从 `map` 中找出 `RouteMeta` 对象，然后返回一个 `Postcard`.

```
public static Postcard buildProvider(String name) {

        RouteMeta routeMeta = Warehouse.routeMap.get(name);

        if (null == routeMeta) {

            return null;

        } else {

            return new Postcard(routeMeta.getPath());

        }

    }
```

`complete(Postcard)`方法，其实就是完善 `postcard` 的字段，且，针对 `Provider`，进行特别处理，反射创建 `Provider` 对象，并建立 `Provider` 的缓存机制，防止多次进行数据交互时进行无意义的反射创建对象。

```
/**

     * Postcard 字段补全

     *

     * @param postcard

     */

    public static void complete(Postcard postcard) {
```

```
        if (null == postcard) {

            throw new RuntimeException("err:postcard 是空的，怎么搞的？");

        }



        RouteMeta routeMeta = Warehouse.routeMap.get(postcard.getPath());
//

        if (null == routeMeta) {//如果路由 meta 是空，说明可能这个路由没注册，
也有可能路由表没有去加载到内存中

            throw new RuntimeException("err:路由寻址失败，请检查是否 path 写
错了");

        } else {

            postcard.setDestination(routeMeta.getDestination());

            postcard.setRouteType(routeMeta.getRouteType());



            switch (routeMeta.getRouteType()) {

                case PROVIDER://如果是数据接口 Provider 的话

                    Class<? extends IProvider> clz = (Class<? extends IPro
vider>) routeMeta.getDestination();

                    //从 map 中找找看

                    IProvider provider = Warehouse.providerMap.get(clz);

                    //如果没找到

                    if (null == provider) {

                        //执行反射方法创建，并且存入到 map

                        try {

                            provider = clz.getConstructor().newInstance();
```

```
                    provider.init(mContext);
                    Warehouse.providerMap.put(clz, provider);
                } catch (Exception e) {
                }
            }
            postcard.setProvider(provider);
            break;
        default:
            break;
    }
}
}
```

看到这里，整个路由框架，包括模块间的通信，就讲解完毕了。

做下结论吧：

使用路由框架的目的，是 在项目代码组件化的背景之下，优化 Activity 跳转，Fragment 切换的重复代码的编写，而统一使用路由框架的对外接口执行跳转或者切换。同时，通过路由框架的对外接口，实现组件之间的无障碍通信，保证组件的独立性。

在探索框架的过程中，我们遇到了很多辅助类，但是辅助类怎么写，完全看个人习惯，我是看了阿里巴巴的 ARtouer 框架之后得到启发，按照它的思路来写自己的路由框架，但是很多辅助类的写法，我并完全按它的意思来。但是，核心思想，APT 注解+反射+自动生成代码 是完全一样的。

BAT交流：892872246

所以说，打蛇打七寸，看框架要看核心，拿住核心之后，其他的东西，就算代码量再大，也是狐假虎威。

4、使用组件 api 化，在模块很多的情况下优化公共模块的结构

回顾一下理想中的项目结构:

背景

这里的功能组件层 function，是存放各个业务模块都需要的公共类或者接口。这里说的公共类，也包含了刚才所提及的 业务模块之间进行通信所需要的接口。

举例说明：A 模块，需要调用 B 模块的 test()接口，由于 A 不能直接引用 B 模块，那这个 test 接口，只能放在 function 这个公共模块内，然后 A,B 同时引用，B 对 test 接口进行实现并通过注解进行路由注册，A 通过路由对外接口调用 B 的 test 方法。

现状

诚然，这种做法没毛病，能够实现功能。但是随着项目模块的增多，function 里面会存在很多的业务模块数据接口。有一种情况：如果存在 A,B,C,D,E 5 个模块，它们都在 function 内存放了 数据接口，并且 5 个模块都引用了 function 模块。那么，当 A 需要，并且只需要 B 的数据接口，而不需要 C,D,E 的接口时，它还是不得不去引用这些用不着的接口。A 不需要这些接口，但是，还不得不引用！这显然会不合逻辑。并且这种 全部业务数据接口都塞到 function 模块里面的做法，会导致 function 出现不必要的臃肿。

理想

每个业务模块的数据接口，只和本模块的业务有关，所以最好是放在本模块之内，但是，如果放在本模块之内，又会导致组件之间不能通信. 那么就创建一个专门的 Module 来存放每个业务模块的接口。想法可行，但是每个业务模块的 module 数量一下子加倍了，又会造成维护困难的问题。那么有没有方法可以自动生成这些数据接口模块呢？ 还真有~ 神奇的 gradle 编程 >_<*

关键词

892872246

组件 API 化技术 使用 gradle 配置，对 module 内的特殊后缀文件进行检索，并以当前 module 为基础，自动生成新的 module.

上干货：

这个名叫 MineOpenServiceApi 的接口，原本是.java 后缀，现在改成.api

BAT交流群：892872246

打开 demo 的 setting.gradle 文件：

找到下面的代码：

```
include_with_api(':business:fragment_mine')



def include_with_api(String moduleName) {

    include(moduleName)

    //获得工程根目录

    String originDir = project(moduleName).projectDir

    //制作的 SDK 工程的目录

    String targetDir = "${originDir}_api"

    //制作的 SDK 工程的名字

    String sdkName = "${project(moduleName).name}_api"

    System.out.println("-----------------------------------SDK name:"
+ sdkName)

    //删除掉 SDK 工程目录 除了 iml

    FileTree targetFiles = fileTree(targetDir)

    targetFiles.exclude "*.iml"

    targetFiles.each { File file ->

        file.delete()

    }

    //从待制作 SDK 工程拷贝目录到 SDK 工程 只拷贝目录

    copy {

        from originDir

        into targetDir
```

```
        //拷贝文件

        include '**/*.api'

        include '**/AndroidManifest.xml'

        include 'api.gradle'

    }

    //读取实现模块的 manifest 并将 package 的值后加 .api 作为 API 工程的 manifes
t package

    FileTree manifests = fileTree(targetDir).include("**/AndroidManifest.
xml")

    manifests.each {

        File file ->

            def parser = new XmlParser().parse(file)

            def node = parser.attribute('package')

            parser.attributes().replace('package', "${node}.api")

            new XmlNodePrinter(new PrintWriter(file)).print(parser)

    }



    //将 api.gradle 改为 build.gradle

    File build = new File(targetDir + "/api.gradle")

    if (build.exists()) {

        build.renameTo(new File(targetDir + "/build.gradle"))

    }



    // 将.api 文件改为 .java
```

```
    FileTree files = fileTree(targetDir).include("**/*.api")

    files.each {

        File file ->

            file.renameTo(new File(file.absolutePath.replace(".api", ".ja
va")))

    }

    //加入 SDK 工程

    include ":business:" + "$sdkName"}
```

这段代码来自一位很厉害的大神，它的作用是，检索指定模块里面，有没有指定后缀名（.api）的文件，有的话，找出来，经过一系列处理(注解很详细，应该能看懂)，自动生成一个 module. 生成的 module 名字比原来的 module 多一个_api. 表示这个模块，包含原模块的所有对外数据接口

有几处细节需要注意:

- 数据接口的.java 后缀需要改成.api(整个.api 完全和 setting.gradle 代码里的.api 对应，你可以都换成其他后缀，比如.apixxxxx)
- 原模块里面，会多出一个 api.gradle, 这个文件的名字也和 setting.gradle 里的 api.gradle 对应，也可以修改

BAT交流群：892872246

这个 api.gradle 并不会在本模块被编译的时候起作用，但是它最终会变成 _api 新模块的 build.gradle，并保持完全一样的代码。新的_api 模块只是一个 library，所以，要去掉 本模块里面的 build.gradle 里面针对 isModule 的判定。

OK,感受一下组件 API 化的成果：

BAT交流群：892872246

理想实现了

现在不用把所有的数据接口都放到 function 公共模块内，而只需要在本模块之内将数据接口文件后缀改成.api，然后在 setting.gradle 里面使用自定义的方法进

行 include。 就可以只引用本模块需要的 数据接口 module，而不需要引用多余的 module,而且，防止了 function 模块的无意义的膨胀。简直破费。

第四章、图片加载框架

4.1 图片加载框架选型

4.1.1 Universal ImangeLoader、Glide、Picasso 与 Fresco

基本介绍

相信大家平时做 Android 应用的时候，多少会接触到异步加载图片，或者加载大量图片的问题，而加载图片我们常常会遇到许多的问题，比如说图片的错乱，OOM 等问题，对于新手来说，这些问题解决起来会比较吃力，所以就有很多的开源图片加载框架应运而生，比较著名的就是 Universal-Image-Loader，相信很多朋友都听过或者使用过这个强大的图片加载框架，今天这篇文章就是对这个框架的基本介绍以及使用，主要是帮助那些没有使用过这个框架的朋友们。该项目存在于 Github 上面 Android-Universal-Image-Loader，我们可以先看看这个开源库存在哪些特征

- 多线程下载图片，图片可以来源于网络，文件系统，项目文件夹 assets 中以及 drawable 中等
-
- 支持随意的配置 ImageLoader，例如线程池，图片下载器，内存缓存策略，硬盘缓存策略，图片显示选项以及其他的一些配置
-
-

BAT交流群：892872246

支持图片的内存缓存，文件系统缓存或者 SD 卡缓存

-
-

支持图片下载过程的监听

-
-

根据控件(ImageView)的大小对 Bitmap 进行裁剪，减少 Bitmap 占用过多的内存

-
-

较好的控制图片的加载过程，例如暂停图片加载，重新开始加载图片，一般使用在 ListView,GridView 中，滑动过程中暂停加载图片，停止滑动的时候去加载图片

-
-

提供在较慢的网络下对图片进行加载

-

当然上面列举的特性可能不全，要想了解一些其他的特性只能通过我们的使用慢慢去发现了

ImageLoaderConfiguratio

图片加载器 ImageLoader 的配置参数，使用 Builder 模式。

常用的配置属性有

```
//通过 StorageUtils 获取内置的内存目录/data/data/.../cache

File cacheDir = StorageUtils.getCacheDirectory(context);

ImageLoaderConfiguration config = new ImageLoaderConfiguration

                .Builder(getApplicationContext())

                .memoryCacheExtraOptions(480, 800) //即保存的每个缓存文件的
最大长宽

                .threadPoolSize(3) //线程池内加载的数量
```

```
                .threadPriority(Thread.NORM_PRIORITY - 2)

                //解释：当同一个 Uri 获取不同大小的图片，缓存到内存时，只缓存一个。默认会缓存多个不同的大小的相同图片

                .denyCacheImageMultipleSizesInMemory()  //拒绝缓存多个图片。

                .memoryCache(new WeakMemoryCache()) //缓存策略你可以通过自己的内存缓存实现 ，这里用弱引用，缺点是太容易被回收了，不是很好！

                .memoryCacheSize(2 * 1024 * 1024) //设置内存缓存的大小

                .diskCacheSize(50 * 1024 * 1024) //设置磁盘缓存大小 50M

                .diskCacheFileNameGenerator(new Md5FileNameGenerator()) //将保存的时候的 URI 名称用 MD5 加密

                .tasksProcessingOrder(QueueProcessingType.LIFO) //设置图片下载和显示的工作队列排序

                .diskCacheFileCount(100) //缓存的文件数量

                .diskCache(new UnlimitedDiskCache(cacheDir)) //自定义缓存路径

                .defaultDisplayImageOptions(defaultOptions) //显示图片的参数，默认：DisplayImageOptions.createSimple()

                .imageDownloader(new BaseImageDownloader(this, 5 * 1000, 30 * 1000)) // connectTimeout (5 s), readTimeout (30 s)超时时间

                .writeDebugLogs() //打开调试日志

                .build();//开始构建  //配置使用

ImageLoader.getInstance().init(configuration);
```

可以设置内存缓存，硬盘缓存的相关参数等。

设置完相关的参数后就可进行图片加载显示

图片加载

BAT交流群：892872246

ImageLader 提供了几个图片加载的方法，主要是这几个 displayImage(), loadImage(),loadImageSync()，loadImageSync()方法是同步的，android4.0 有个特性，网络操作不能在主线程，所以 loadImageSync()方法我们就不去使用

loadimage()加载图片

我们先使用 ImageLoader 的 loadImage()方法来加载网络图片

。

```
final ImageView mImageView = (ImageView) findViewById(R.id.image);

        String imageUrl = "https://lh6.googleusercontent.com/-55osAWw3x0
Q/URquUtcFr5I/AAAAAAAAAbs/rWlj1RUKrYI/s1024/A%252520Photographer.jpg";



        ImageLoader.getInstance().loadImage(imageUrl, new ImageLoadingLi
stener() {



            @Override

            public void onLoadingStarted(String imageUri, View view) {



            }



            @Override

            public void onLoadingFailed(String imageUri, View view,

                    FailReason failReason) {



            }



            @Override
```

```
            public void onLoadingComplete(String imageUri, View view, Bit
map loadedImage) {

                mImageView.setImageBitmap(loadedImage);

            }



            @Override

            public void onLoadingCancelled(String imageUri, View view) {




            }

        });
```

传入图片的 url 和 ImageLoaderListener，在回调方法 onLoadingComplete()中将 loadedImage 设置到 ImageView 上面就行了，如果你觉得传入 ImageLoaderListener 太复杂了，我们可以使用 SimpleImageLoadingListener 类，该类提供了 ImageLoaderListener 接口方法的空实现，使用的是缺省适配器模式

```
final ImageView mImageView = (ImageView) findViewById(R.id.image);

        String imageUrl = "https://lh6.googleusercontent.com/-55osAWw3x0
Q/URquUtcFr5I/AAAAAAAAAbs/rWlj1RUKrYI/s1024/A%252520Photographer.jpg";



        ImageLoader.getInstance().loadImage(imageUrl, new SimpleImageLoa
dingListener(){



            @Override

            public void onLoadingComplete(String imageUri, View view,

                    Bitmap loadedImage) {

                super.onLoadingComplete(imageUri, view, loadedImage);
```

```
            mImageView.setImageBitmap(loadedImage);

        }



    });
```

如果我们要指定图片的大小该怎么办呢，这也好办，初始化一个 ImageSize 对象，指定图片的宽和高，代码如下

```
final ImageView mImageView = (ImageView) findViewById(R.id.image);

    String imageUrl = "https://lh6.googleusercontent.com/-55osAWw3x0
Q/URquUtcFr5I/AAAAAAAAAbs/rWlj1RUKrYI/s1024/A%252520Photographer.jpg";



    ImageSize mImageSize = new ImageSize(100, 100);



    ImageLoader.getInstance().loadImage(imageUrl, mImageSize, new Si
mpleImageLoadingListener(){



        @Override

        public void onLoadingComplete(String imageUri, View view,

                Bitmap loadedImage) {

            super.onLoadingComplete(imageUri, view, loadedImage);

            mImageView.setImageBitmap(loadedImage);

        }



    });
```

BAT文 /2246

上面只是很简单的使用 ImageLoader 来加载网络图片，在实际的开发中，我们并不会这么使用，那我们平常会怎么使用呢？我们会用到 DisplayImageOptions，他可以配置一些图片显示的选项，比如图片在加载中 ImageView 显示的图片，是否需要使用内存缓存，是否需要使用文件缓存等等

DisplayImageOptions

可以配置一些图片显示的选项，比如图片在加载中 ImageView 显示的图片，是否需要使用内存缓存，是否需要使用文件缓存等等，可供我们选择的配置如下

```
DisplayImageOptions options = new DisplayImageOptions.Builder()

        .showImageOnLoading(R.drawable.ic_stub) // resource or drawable

        .showImageForEmptyUri(R.drawable.ic_empty) // resource or drawabl
e

        .showImageOnFail(R.drawable.ic_error) // resource or drawable

        .resetViewBeforeLoading(false)  // default

        .delayBeforeLoading(1000)

        .cacheInMemory(false) // default

        .cacheOnDisk(false) // default

        .preProcessor(...)

        .postProcessor(...)

        .extraForDownloader(...)

        .considerExifParams(false) // default

        .imageScaleType(ImageScaleType.IN_SAMPLE_POWER_OF_2) // default

        .bitmapConfig(Bitmap.Config.ARGB_8888) // default

        .decodingOptions(...)

        .displayer(new SimpleBitmapDisplayer()) // default

        .handler(new Handler()) // default
```

```
        .build();
```

大家就可以根据实际情况去设置。

displayImage()加载图片

接下来我们就来看看网络图片加载的另一个方法 displayImage()，代码如下

```
        String imageUrl = "https://lh6.googleusercontent.com/-55osAWw3x0
Q/URquUtcFr5I/AAAAAAAAAbs/rWlj1RUKrYI/s1024/A%252520Photographer.jpg";



        //显示图片的配置

        DisplayImageOptions options = new DisplayImageOptions.Builder()

                .showImageOnLoading(R.drawable.ic_stub)

                .showImageOnFail(R.drawable.ic_error)

                .cacheInMemory(true)

                .cacheOnDisk(true)

                .bitmapConfig(Bitmap.Config.RGB_565)

                .build();



        ImageLoader.getInstance().displayImage(imageUrl, mImageView, opt
ions);
```

可以看到这里是直接传递了 ImageView 进行设置显示，并不需要监听后设置，这样更为简便，这也是 `displayImage` 和 `loadImage` 的区别。

BAT交流群：892872246

加载其他来源的图片

使用 Universal-Image-Loader 框架不仅可以加载网络图片，还可以加载 sd 卡中的图片，Content provider 等，使用也很简单，只是将图片的 url 稍加的改变下就行了，下面是加载文件系统的图片

我们只需要给每个图片来源的地方加上 Scheme 包裹起来(Content provider 除外)，然后当做图片的 url 传递到 imageLoader 中，Universal-Image-Loader 框架会根据不同的 Scheme 获取到输入流

```
        //图片来源于文件

        String imagePath = "/mnt/sdcard/image.png";

        String imageUrl = Scheme.FILE.wrap(imagePath);

        //相当于 file:/mnt/sdcard/image.png



        //图片来源于 Content provider

        String contentprividerUrl = "content://media/external/audio/albu
mart/13";



        //图片来源于 assets

        String assetsUrl = Scheme.ASSETS.wrap("image.png");



        //图片来源于

        String drawableUrl = Scheme.DRAWABLE.wrap("R.drawable.image");
```

获取到对应 URL 后就可以调用 display/loadImage 方法进行显示。

BAT交流群：892872246

GirdView,ListView 加载图片

相信大部分人都是使用 GridView，ListView 来显示大量的图片，而当我们快速滑动 GridView，ListView，我们希望能停止图片的加载，而在 GridView，ListView 停止滑动的时候加载当前界面的图片，这个框架当然也提供这个功能，使用起来也很简单，它提供了 PauseOnScrollListener 这个类来控制 ListView,GridView 滑动过程中停止去加载图片，该类使用的是代理模式

```
listView.setOnScrollListener(new PauseOnScrollListener(imageLoader, paus
eOnScroll, pauseOnFling));



gridView.setOnScrollListener(new PauseOnScrollListener(imageLoader, paus
eOnScroll, pauseOnFling));
```

第一个参数就是我们的图片加载对象 ImageLoader,
第二个是控制是否在滑动过程中暂停加载图片，如果需要暂停传 true 就行了，
第三个参数控制猛的滑动界面的时候图片是否加载

OutOfMemoryError

虽然这个框架有很好的缓存机制，有效的避免了 OOM 的产生，一般的情况下产生 OOM 的概率比较小，但是并不能保证 OutOfMemoryError 永远不发生，这个框架对于 OutOfMemoryError 做了简单的 catch,保证我们的程序遇到 OOM 而不被 crash 掉，但是如果我们使用该框架经常发生 OOM，我们应该怎么去改善呢？

-

减少线程池中线程的个数，在 ImageLoaderConfiguration 中的（.threadPoolSize）中配置，推荐配置 1-5

-
-

在 DisplayImageOptions 选项中配置 bitmapConfig 为 Bitmap.Config.RGB_565，因为默认是 ARGB_8888， 使用 RGB_565 会比使用 ARGB_8888 少消耗 2 倍的内存

-
-

BAT交流群：892872246

在 ImageLoaderConfiguration 中配置图片的内存缓存为 memoryCache(new WeakMemoryCache()) 或者不使用内存缓存

-
-

在 DisplayImageOptions 选项中设置.imageScaleType(ImageScaleType.IN_SAMPLE_INT)或者 imageScaleType(ImageScaleType.EXACTLY)

-

[Universal-Image-Loader 解析内部缓存原理]

对于我们所知道的缓存，常用的是内存缓存 MemoryCache 和硬盘缓存 DiscCache。一个读取快容量小，一个读取慢容量大。

对于各自使用哪种缓存，则可以在前面配置 `ImageLoaderConfiguration` 进行缓存设置，当然也可以自己自定义适合的缓存。

```
ImageLoaderConfiguration configuration = new ImageLoaderConfiguration.Builder(this)

        .memoryCache(new WeakMemoryCache())

        .build();
```

对于 Universal-Image-Loader 来说它的缓存结构也是分为内存缓存 MemoryCache 和硬盘缓存 DiskCache

MemoryCache 内存缓存

首先先看个结构图，理解 UIL 里面内存缓存的结构

BAT交流群：892872246

image

由于空间有限就没画成标准的 UML 类图形式。

对于基类 `MemoryCache` 它则是一个接口，里面定义了 put,get 图片的方法

```
public interface MemoryCache {

    ...

    boolean put(String key, Bitmap value);


    Bitmap get(String key);


    Bitmap remove(String key);


    Collection<String> keys();


    void clear();}
```

都是大家比较所熟悉的方法，而对于其他的类

我们一个个看

LruMemoryCache

这个类就是这个开源框架默认的内存缓存类，缓存的是 bitmap 的强引用。直接实现了 `MemoryCache` 方法

```
public class LruMemoryCache implements MemoryCache {



    private final LinkedHashMap<String, Bitmap> map;

    //最大容量

    private final int maxSize;

    /** 目前缓存的容量大小 */

    private int size;

    public LruMemoryCache(int maxSize) {

        ...

        this.maxSize = maxSize;

        this.map = new LinkedHashMap<String, Bitmap>(0, 0.75f, true);

    }

    @Override

    public final Bitmap get(String key) {

        ...

        synchronized (this) {

            return map.get(key);

        }
```

```
    }

    @Override

    public final boolean put(String key, Bitmap value) {

        ...

        synchronized (this) {

            size += sizeOf(key, value);

            Bitmap previous = map.put(key, value);

            if (previous != null) {

                size -= sizeOf(key, previous);

            }

        }


        trimToSize(maxSize);

        return true;

    }


    /**

     * Lru 算法，当容量超过最大缓存容量，则移除最久的条目

     */

    private void trimToSize(int maxSize) {

        while (true) {

            String key;

            Bitmap value;
```

```
            synchronized (this) {

                if (size < 0 || (map.isEmpty() && size != 0)) {

                    throw new IllegalStateException(getClass().getName() +
 ".sizeOf() is reporting inconsistent results!");

                }



                if (size <= maxSize || map.isEmpty()) {

                    break;

                }



                Map.Entry<String, Bitmap> toEvict = map.entrySet().iterat
or().next();

                if (toEvict == null) {

                    break;

                }

                key = toEvict.getKey();

                value = toEvict.getValue();

                map.remove(key);

                size -= sizeOf(key, value);

            }

        }

    }



    @Override
```

```
    public final Bitmap remove(String key) {

        ...

        synchronized (this) {

            Bitmap previous = map.remove(key);

            if (previous != null) {

                size -= sizeOf(key, previous);

            }

            return previous;

        }

    }

    ...

    //返回图片的字节大小

    private int sizeOf(String key, Bitmap value) {

        return value.getRowBytes() * value.getHeight();

    }

    ...}
```

LruMemoryCache 的源码也比较简单，内部有个成员变量 LinkedHashMap<String, Bitmap> map 这里直接进行保存的话则是强引用的形式。
主要看 get, put 方法。
对于 get 方法来说，比较简单，直接根据指定的 key 返回对应的图片。
而对于 put 方法来说，则需要考虑容量的问题。

```
@Override

    public final boolean put(String key, Bitmap value) {

        ...
```

```
        synchronized (this) {

            size += sizeOf(key, value);

            Bitmap previous = map.put(key, value);

            if (previous != null) {

                size -= sizeOf(key, previous);

            }

        }



        trimToSize(maxSize);

        return true;

    }
```

put 方法首先调用了 `sizeof` 方法，该方法则是返回指定 Bitmap 的字节大小，之后 size +=，总缓存量增加，之后调用 `trimToSize` 该方法则是进行缓存容量判断的。

```
private void trimToSize(int maxSize) {

        while (true) {

            String key;

            Bitmap value;

            synchronized (this) {

                if (size < 0 || (map.isEmpty() && size != 0)) {

                    throw new IllegalStateException(getClass().getName() +
 ".sizeOf() is reporting inconsistent results!");

                }
```

```
            if (size <= maxSize || map.isEmpty()) {
                break;
            }

            Map.Entry<String, Bitmap> toEvict = map.entrySet().iterat
or().next();
            if (toEvict == null) {
                break;
            }
            key = toEvict.getKey();
            value = toEvict.getValue();
            map.remove(key);
            size -= sizeOf(key, value);
        }
    }
}
```

如果加入后的 size 缓存容量 <= maxSize 最大缓存容量，则直接 break,不用进行判定处理。
如果大于的话，则直接移除最久未使用的。

大家肯定有疑问，它到底怎么判断最久未使用的？没看到相关代码呀？

相信知道 `LinkedHashMap` 的话可能就知道。
`LinkedHashMap` 自身已经实现了顺序存储，默认情况下是按照元素的添加顺序存储，也可以启用按照访问顺序存储，即最近读取的数据放在最前面，最早读取的数据放在最后面，然后它还有一个判断是否删除最老数据的方法，默认是返回 false，即不删除数据。大家常见也就是按顺序存储，很少忘了它还可以根据最近未使用的方法。

BAT交流群：892872246

```
//LinkedHashMap 的一个构造函数，当参数 accessOrder 为 true 时，即会按照访问顺序排序，最近访问的放在最前，最早访问的放在后面 public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder) {

        super(initialCapacity, loadFactor);

        this.accessOrder = accessOrder;}

//LinkedHashMap 自带的判断是否删除最老的元素方法，默认返回 false，即不删除老数据//我们要做的就是重写这个方法，当满足一定条件时删除老数据 protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {

        return false;}
```

回看我们前面 `LinkedHashMap` 的创建

```
  this.map = new LinkedHashMap<String, Bitmap>(0, 0.75f, true);
```

再举个列子

```
9
10 public class Test {
11
12     public static void main(String[] args) {
13         LinkedHashMap<String , String> map = new LinkedHashMap<String , String>(0 , 0.7f , true);
14         map.put("1", "1");
15         map.put("2", "2");
16         map.put("3", "3");
17         map.put("4", "4");
18
19         System.out.println(map);
20         map.get("1");
21         System.out.println(map);

Problems  Javadoc  Declaration  Console
<terminated> Test [Java Application] E:\jdk\jdk1.8.0_40\bin\javaw.exe (2017年2月16日 下午10:29:14)
{1=1, 2=2, 3=3, 4=4}
{2=2, 3=3, 4=4, 1=1}
```

image

就比较明了了。

BaseMemoryCache

`BaseMemoryCache` 同样也是实现了 `MemoryCache` 方法，不过它还是一个抽象类。它是一个内存缓存的基类，实现了内存缓存中常用的方法，只不过它里面提供了一个非强引用的 `Reference` 作为扩展，方便 GC 的回收，避免 OOM.

```
public abstract class BaseMemoryCache implements MemoryCache {
```

IT交流群：892872246

```
    /** Stores not strong references to objects */

    private final Map<String, Reference<Bitmap>> softMap = Collections.s
ynchronizedMap(new HashMap<String, Reference<Bitmap>>());


    @Override

    public Bitmap get(String key) {

        Bitmap result = null;

        Reference<Bitmap> reference = softMap.get(key);

        if (reference != null) {

            result = reference.get();

        }

        return result;

    }


    @Override

    public boolean put(String key, Bitmap value) {

        softMap.put(key, createReference(value));

        return true;

    }


    @Override

    public Bitmap remove(String key) {
```

```
        Reference<Bitmap> bmpRef = softMap.remove(key);

        return bmpRef == null ? null : bmpRef.get();

    }



    /** Creates {@linkplain Reference not strong} reference of value */

    protected abstract Reference<Bitmap> createReference(Bitmap value);}
```

代码也比较简单，内存持有一个 `Map<String, Reference<Bitmap>> softMap` 来保存非强引用对象，具体的引用类型则看它实现的抽象方法 `createReference`。

WeakMemoryCache

我们看它的一个子类 `WeakMemoryCache` 则是继承与 `BaseMemory`，实现 `createReference`

```
public class WeakMemoryCache extends BaseMemoryCache {

    @Override

    protected Reference<Bitmap> createReference(Bitmap value) {

        return new WeakReference<Bitmap>(value);

    }}
```

很明显是来保存弱引用对象的。

LimitedMemoryCache

我们看它的另外一个子类 `LimitedMemoryCache`，但它并没有实现 `BaseMemoryCache` 里的 `createReference` 方法，它也是一个抽象类，在 `BaseMemoryCache` 基础上封装了个抽象方法 `protected abstract Bitmap removeNext();`用来处理当缓存容量不足时的情况。

```
public abstract class LimitedMemoryCache extends BaseMemoryCache {

    ...
```

```
    //当前保存的 Bitmap，用来统计缓存数

    private final List<Bitmap> hardCache = Collections.synchronizedList
(new LinkedList<Bitmap>());

    ...

    @Override

    public boolean put(String key, Bitmap value) {

        boolean putSuccessfully = false;

        // Try to add value to hard cache

        int valueSize = getSize(value);

        int sizeLimit = getSizeLimit();

        int curCacheSize = cacheSize.get();

        if (valueSize < sizeLimit) {

            while (curCacheSize + valueSize > sizeLimit) {

                Bitmap removedValue = removeNext();

                if (hardCache.remove(removedValue)) {

                    curCacheSize = cacheSize.addAndGet(-getSize(removedVa
lue));

                }

            }

            hardCache.add(value);

            cacheSize.addAndGet(valueSize);



            putSuccessfully = true;

        }
```

```
        // Add value to soft cache
        super.put(key, value);
        return putSuccessfully;
    }

    @Override
    public Bitmap remove(String key) {
        Bitmap value = super.get(key);
        if (value != null) {
            if (hardCache.remove(value)) {
                cacheSize.addAndGet(-getSize(value));
            }
        }
        return super.remove(key);
    }
...
    protected abstract int getSize(Bitmap value);

    protected abstract Bitmap removeNext();}
```

可以看到在 `LimitedMemoryCache` 里面又有一个 `List<Bitmap>`保存的是强引用，而在 `BaseMemoryCache` 里面也有个 `Map<String, Reference<Bitmap>> softMap` 来保存 Bitmap，为什么要这样。

这主要是因为在 `BaseMemoryCache` 里面并没有做缓存限制处理，它只是封装实现了基本的 Bitmap 的 put,get。而当面对缓存容量有限的情况下，则需要交给子类去处理。

BAT交流群：892872246

我们看下这里的 put 方法，关键在

```
while (curCacheSize + valueSize > sizeLimit) {

                Bitmap removedValue = removeNext();

                if (hardCache.remove(removedValue)) {

                    curCacheSize = cacheSize.addAndGet(-getSize(removedVa
lue));

                }

            }
```

当超过容量时，调用抽象方法 `removeNext` 由子类自行实现，之后 hardCache 移除，但此时并没有调用 softMap 的移除。

也就是对于 `List<Bitmap>`来说，当它的缓存容量超过的时候，它会移除第一个对象来缓解容量，但是保存在 `Map<String, Reference<Bitmap>> softMap` 里面的 Bitmap 并没有被移除。
如果这样下去 softMap 岂不是会无限大？

这是因为在 `Map<String, Reference<Bitmap>> softMap` 里面保存的 Bitmap 是弱引用的存在，而在 `List<Bitmap>`里面保存的是强引用，当内存不足的时候，GC 则会先清除 softMap 里面的对象。

FIFOLimitedMemoryCache

我们看下 `LimitedMemoryCache` 的一个子类 `FIFOLimitedMemoryCache`，看到 FIFO 也就是先进先出了。

```
public class FIFOLimitedMemoryCache extends LimitedMemoryCache {



    private final List<Bitmap> queue = Collections.synchronizedList(new
LinkedList<Bitmap>());

    ...

    @Override

    public boolean put(String key, Bitmap value) {
```

```
        if (super.put(key, value)) {
            queue.add(value);
            return true;
        } else {
            return false;
        }
    }

    @Override
    public Bitmap remove(String key) {
        Bitmap value = super.get(key);
        if (value != null) {
            queue.remove(value);
        }
        return super.remove(key);
    }
    ...
    @Override
    protected Bitmap removeNext() {
        return queue.remove(0);
    }

    @Override
```

```
protected Reference<Bitmap> createReference(Bitmap value) {

    return new WeakReference<Bitmap>(value);

}}
```

可以看到，这里的 LRU 处理则是使用 `LinkedHashMap`，在它的构造方法中第三个参数为 `true` 表示使用 LRU，之后再 `removeNext` 返回那个 Bitmap。

同理其他子类也如下，就不一一列举。

MemoryCache 小结

1. 只使用的是强引用缓存

- LruMemoryCache（这个类就是这个开源框架默认的内存缓存类，缓存的是 bitmap 的强引用）

2.使用强引用和弱引用相结合的缓存有

-

UsingFreqLimitedMemoryCache（如果缓存的图片总量超过限定值，先删除使用频率最小的 bitmap）

-
-

LRULimitedMemoryCache（这个也是使用的 lru 算法，和 LruMemoryCache 不同的是，他缓存的是 bitmap 的弱引用）

-
-

FIFOLimitedMemoryCache（先进先出的缓存策略，当超过设定值，先删除最先加入缓存的 bitmap）

-
-

LargestLimitedMemoryCache(当超过缓存限定值，先删除最大的 bitmap 对象)

-
-

BAT交流群：892872246

LimitedAgeMemoryCache（当 bitmap 加入缓存中的时间超过我们设定的值，将其删除）

-

3.只使用弱引用缓存

- WeakMemoryCache（这个类缓存 bitmap 的总大小没有限制，唯一不足的地方就是不稳定，缓存的图片容易被回收掉）

二.DiskCache 硬盘缓存

同样先来看个结构

image

DiskCache 的设计其实和 MemoryCache 一样，对于基类 `DiskCache`，它同样是一个接口

```
public interface DiskCache {

    //返回硬盘缓存的根目录

    File getDirectory();



    File get(String imageUri);
```

```
    boolean save(String imageUri, InputStream imageStream, IoUtils.CopyL
istener listener) throws IOException;



    boolean save(String imageUri, Bitmap bitmap) throws IOException;



    boolean remove(String imageUri);



    void close();



    void clear();}
```

同样一个个看

LruDiskCache

LruDiskCache 则是直接实现了 DiskCache 接口，采用 LRU 算法来进行缓存处理。再理解 LruDiskCache 前，先理解另一个类 DiskLruCache

```
final class DiskLruCache implements Closeable {

    static final String JOURNAL_FILE = "journal";

    static final String JOURNAL_FILE_TEMP = "journal.tmp";

    static final String JOURNAL_FILE_BACKUP = "journal.bkp";

    static final String MAGIC = "libcore.io.DiskLruCache";

    ...

    private final LinkedHashMap<String, Entry> lruEntries =

            new LinkedHashMap<String, Entry>(0, 0.75f, true);

    ...
```

```
    final ThreadPoolExecutor executorService =

            new ThreadPoolExecutor(0, 1, 60L, TimeUnit.SECONDS, new Linke
dBlockingQueue<Runnable>());

    ...

    public static DiskLruCache open(File directory, int appVersion, int v
alueCount, long maxSize, int maxFileCount)

            throws IOException {

        ...

    }

    ...

    public synchronized Snapshot get(String key) throws IOException {

        ...

    }

    ...

    private synchronized Editor edit(String key, long expectedSequenceNu
mber) throws IOException {

        ...

    }

    /** A snapshot of the values for an entry. */

    public final class Snapshot implements Closeable {

        private final String key;

        private final long sequenceNumber;

        private File[] files;

        private final InputStream[] ins;
```

```
    private final long[] lengths;
    ...
}
...
public final class Editor {
    private final Entry entry;
    private final boolean[] written;
    private boolean hasErrors;
    private boolean committed;
    ...
}
...
private final class Entry {
    private final String key;

    private final long[] lengths;

    private boolean readable;

    private Editor currentEditor;

    private long sequenceNumber;
    ...
```

```
    }
```

这个 DiskLruCache 比较长也比较复杂，它是 LruDiskCache 的一个文件工具类。这里的缓存数据存储在文件系统上的一个目录。
同时也注意到这里的一个成员变量
private final LinkedHashMap<String, Entry> lruEntries =new LinkedHashMap<String, Entry>(0, 0.75f, true);
可以知道这是用来处理 LRU 的。

同时这里的 value 则是 Entry，Entry 则是封装了当前文件的编辑情况 Ediotr 以及 key。
而这里 Editor 封装了文件的写入情况 OutputStream，Snapshot 封装了文件的读取情况 InputStream。

回头看回 LruDiskCache

```
public class LruDiskCache implements DiskCache {

    protected DiskLruCache cache;

    private File reserveCacheDir;



    protected final FileNameGenerator fileNameGenerator;

    ...

    public LruDiskCache(File cacheDir, File reserveCacheDir, FileNameGen
erator fileNameGenerator, long cacheMaxSize,

            int cacheMaxFileCount) throws IOException {

        ...

        this.reserveCacheDir = reserveCacheDir;

        this.fileNameGenerator = fileNameGenerator;

        initCache(cacheDir, reserveCacheDir, cacheMaxSize, cacheMaxFileC
ount);

    }
```

```
    private void initCache(File cacheDir, File reserveCacheDir, long cac
heMaxSize, int cacheMaxFileCount)

        ...

            cache = DiskLruCache.open(cacheDir, 1, 1, cacheMaxSize, cache
MaxFileCount);

        ...

    }

    @Override

    public File get(String imageUri) {

        DiskLruCache.Snapshot snapshot = null;

        try {

            snapshot = cache.get(getKey(imageUri));

            return snapshot == null ? null : snapshot.getFile(0);

        }

        ...

    }

    @Override

    public boolean save(String imageUri, Bitmap bitmap) throws IOExcepti
on {

        DiskLruCache.Editor editor = cache.edit(getKey(imageUri));

        ...

        OutputStream os = new BufferedOutputStream(editor.newOutputStream
(0), bufferSize);

        boolean savedSuccessfully = false;
```

```
        try {

            savedSuccessfully = bitmap.compress(compressFormat, compressQ
uality, os);

        }

        ...

        return savedSuccessfully;

    }
```

首先 `LruDiskCache` 内部成员变量带有 `DiskLruCache` 还有文件的保存目录等，在它的构造方法中调用 `DiskLruCache.open` 方法创建了 `DiskLruCache` 对象，而在它的 open 方法里，则根据文件的目录情况创建了对应的文件系统。

再看它的 save 方法，先调用 `getKey` 方法将 uri 转换为对应的 key，而在 cache,edit 中

```
private synchronized Editor edit(String key, long expectedSequenceNumber)
 throws IOException {

        ...

        Entry entry = lruEntries.get(key);

        if (expectedSequenceNumber != ANY_SEQUENCE_NUMBER && (entry == nu
ll

                || entry.sequenceNumber != expectedSequenceNumber)) {

            return null; // Snapshot is stale.

        }

        if (entry == null) {

            entry = new Entry(key);

            lruEntries.put(key, entry);

        } else if (entry.currentEditor != null) {

            return null; // Another edit is in progress.
```

```
        }



        Editor editor = new Editor(entry);

        entry.currentEditor = editor;

        ...

        return editor;

    }
```

则是根据指定的 key 先判断缓存文件中有没有相应的 key，如果没有则创建一个 `Entry` 对象持有它，之后保存在 `lruEntries` 之后，创建一个当前 `Entry` 的编辑对象 `Editor`，以便之后写入到文件中。

s 之后调用了

```
        OutputStream os = new BufferedOutputStream(editor.newOutputStream
(0), bufferSize);

        OutputStream os = new BufferedOutputStream(editor.newOutputStream
(0), bufferSize);
```

在 `editor.newOutputStream` 则是根据当前目录和 key 创建出一个文件，之后打开这个文件的一个输出流情况，获取到之后就进行 Bitmap 的写入。

同理，看下 `LruDiskCache` 的 get 方法

```
@Override

    public File get(String imageUri) {

        DiskLruCache.Snapshot snapshot = null;

        try {

            snapshot = cache.get(getKey(imageUri));

            return snapshot == null ? null : snapshot.getFile(0);

        }
```

```
        ...

    }
```

调用了 `cache,get`

```
public synchronized Snapshot get(String key) throws IOException {

        。。。

        Entry entry = lruEntries.get(key);

        ...

        File[] files = new File[valueCount];

        InputStream[] ins = new InputStream[valueCount];

        try {

            File file;

            for (int i = 0; i < valueCount; i++) {

                file = entry.getCleanFile(i);

                files[i] = file;

                ins[i] = new FileInputStream(file);

            }

        }

        ...

        return new Snapshot(key, entry.sequenceNumber, files, ins, entry.
lengths);

    }
```

在 get 方法中，先根据 key 拿到对应的 `Entry`，再拿到对应的文件打开输入流，之后传入到 `Snapshot`，
而在 `snapshot.getFile` 中

```
/** Returns file with the value for {@code index}. */
```

```
        public File getFile(int index) {

            return files[index];

        }
```

返回的则是对应的文件。

BaseDiskCache

`BaseDiskCache` 同样也是直接实现了 `DiskCache` 方法，实现的方法也比较简单

```
public abstract class BaseDiskCache implements DiskCache {

    ...

    protected final File cacheDir;

    protected final File reserveCacheDir;



    protected final FileNameGenerator fileNameGenerator;



    public BaseDiskCache(File cacheDir, File reserveCacheDir, FileNameGe
nerator fileNameGenerator) {

        ...

        this.cacheDir = cacheDir;

        this.reserveCacheDir = reserveCacheDir;

        this.fileNameGenerator = fileNameGenerator;

    }



    @Override
```

BAT交流群892246

```
    public boolean save(String imageUri, Bitmap bitmap) throws IOExcepti
on {

        File imageFile = getFile(imageUri);

        File tmpFile = new File(imageFile.getAbsolutePath() + TEMP_IMAGE_
POSTFIX);

        OutputStream os = new BufferedOutputStream(new FileOutputStream(t
mpFile), bufferSize);

        boolean savedSuccessfully = false;

        try {

            savedSuccessfully = bitmap.compress(compressFormat, compressQ
uality, os);

        } finally {

            IoUtils.closeSilently(os);

            if (savedSuccessfully && !tmpFile.renameTo(imageFile)) {

                savedSuccessfully = false;

            }

            if (!savedSuccessfully) {

                tmpFile.delete();

            }

        }

        bitmap.recycle();

        return savedSuccessfully;

    }



    @Override
```

```
    public File get(String imageUri) {

        return getFile(imageUri);

    }



    protected File getFile(String imageUri) {

        String fileName = fileNameGenerator.generate(imageUri);

        File dir = cacheDir;

        if (!cacheDir.exists() && !cacheDir.mkdirs()) {

            if (reserveCacheDir != null && (reserveCacheDir.exists() || r
eserveCacheDir.mkdirs())) {

                dir = reserveCacheDir;

            }

        }

        return new File(dir, fileName);

    }
```

比较简单，根据对应的文件去打开获取。它的两个子类 `LimitedAgeDiskCache` 和 `UnlimitedDiskCache` 也都不一一扩展开了。

三、Universal-Image-Loader 解析之源代码解析

当我们配置好 `ImageConfiguration` 和 `ImageLoader` 后，我们就会开始调用

```
ImageLoader.getInstance().loadImage(...);

ImageLoader.getInstance().displayImage(...);
```

这两个方法其中一个来显示图片。
先看 `loadImage`

BAT交流 72246

```
public void loadImage(String uri, ImageSize targetImageSize, DisplayImag
eOptions options,

            ImageLoadingListener listener, ImageLoadingProgressListener p
rogressListener) {

        checkConfiguration();

        if (targetImageSize == null) {

            targetImageSize = configuration.getMaxImageSize();

        }

        if (options == null) {

            options = configuration.defaultDisplayImageOptions;

        }



        NonViewAware imageAware = new NonViewAware(uri, targetImageSize,
ViewScaleType.CROP);

        displayImage(uri, imageAware, options, listener, progressListene
r);

    }
```

首先调用了 `checkConfiguration` 用来判断是否有初始化 `ImageLoaderConfiguration`
如果有设置 ImageView 的大小，则设置，没则默认 Configuration 的大小。
如果没有设置 `DisplayImageOptions`，则设置上一个默认的 options
之后创建了个 `NonViewAware`，再调用 `displayImage`。
也就是说，`loadImage` 最终还是调用到了 `displayImage`。

ImageAware

这里的 `NonViewAware` 实现了 `ImageAware` 接口。先来看个结构图

BAT交流群：892872246

ImageAware 是一个接口，内部提供了一系列操作图片的一些方法。
对于 NonViewAware 来说，它内部只是简单的保存图片一些必要的数据，比如图片大小尺寸，URI，ScaleType 这些。主要封装成 ImageAware 来给 displayImage 调用。

看下 displayImage 的使用

```
public void displayImage(String uri, ImageView imageView) {

        displayImage(uri, new ImageViewAware(imageView), null, null, nul
l);

    }
```

这里把 ImageView 封装成 ImageViewAware 再去调用 displayImage 这个就跟 loadImage 一样。
而这里 ImageViewAware 继承与 ViewAware,ViewAware 则实现了 ImageAware 接口。

BAT交流群892246

与 NonViewAware 不同的是 ViewAware 内部持有一个 Reference<View> viewRef 的成员变量，它是用来保存当前 ImageView 的一个弱引用，以便之后来直接设置显示图片。
ViewAware 很多方法都是依赖于这个 View

```
@Override

    public boolean setImageDrawable(Drawable drawable) {

        if (Looper.myLooper() == Looper.getMainLooper()) {

            View view = viewRef.get();

            if (view != null) {

                setImageDrawableInto(drawable, view);

                return true;

            }

        } else {

            L.w(WARN_CANT_SET_DRAWABLE);

        }

        return false;

    }
```

之后就可以在 `ImageViewAware` 中设置显示。

好了回过头看他们最终调用的方法。
这个方法有点长，我们拆分成一部分一部分来看

```
public void displayImage(String uri, ImageAware imageAware, DisplayImage
Options options,

            ImageSize targetSize, ImageLoadingListener listener, ImageLoa
dingProgressListener progressListener) {

        checkConfiguration();

        if (imageAware == null) {
```

```
            throw new IllegalArgumentException(ERROR_WRONG_ARGUMENTS);

        }

        if (listener == null) {

            listener = defaultListener;

        }

        if (options == null) {

            options = configuration.defaultDisplayImageOptions;

        }



        if (TextUtils.isEmpty(uri)) {

            engine.cancelDisplayTaskFor(imageAware);

            listener.onLoadingStarted(uri, imageAware.getWrappedView());

            if (options.shouldShowImageForEmptyUri()) {

                imageAware.setImageDrawable(options.getImageForEmptyUri(c
onfiguration.resources));

            } else {

                imageAware.setImageDrawable(null);

            }

            listener.onLoadingComplete(uri, imageAware.getWrappedView(),
null);

            return;

        }

        ...//下一部分看

    }
```

首先先检查是否有初始化设置 `ImageLoaderConfiguration` 没则抛出异常，没设置 listener 和 DisplayImageOptions 则设置一个默认值。

之后调用 `TextUtils.isEmpty(uri)` 判断是否当前的 uri 为空，则调用 `engine.cancelDisplayTaskFor(imageAware);`
之后则用 listener 通知开始和结束，也比较好理解，主要是这个 engine。

这个 engine 就是 `ImageLoaderEngine`，主要用来负责显示加载图片的一个类。`ImageLoaderEngine` 中存在一个 HashMap,用来记录正在加载的任务，加载图片的时候会将 ImageView 的 id 和图片的 url 加上尺寸加入到 HashMap 中，加载完成之后会将其移除。

接着看下面

```
public void displayImage(String uri, ImageAware imageAware, DisplayImage
Options options,

            ImageSize targetSize, ImageLoadingListener listener, ImageLoa
dingProgressListener progressListener) {

        ...//前一部分

        if (targetSize == null) {

            targetSize = ImageSizeUtils.defineTargetSizeForView(imageAwar
e, configuration.getMaxImageSize());

        }

        String memoryCacheKey = MemoryCacheUtils.generateKey(uri, targetS
ize);

        engine.prepareDisplayTaskFor(imageAware, memoryCacheKey);



        listener.onLoadingStarted(uri, imageAware.getWrappedView());



        Bitmap bmp = configuration.memoryCache.get(memoryCacheKey);

        if (bmp != null && !bmp.isRecycled()) {

            L.d(LOG_LOAD_IMAGE_FROM_MEMORY_CACHE, memoryCacheKey);
```

```
            if (options.shouldPostProcess()) {

                ImageLoadingInfo imageLoadingInfo = new ImageLoadingInfo
(uri, imageAware, targetSize, memoryCacheKey,

                        options, listener, progressListener, engine.getLoc
kForUri(uri));

                ProcessAndDisplayImageTask displayTask = new ProcessAndDi
splayImageTask(engine, bmp, imageLoadingInfo,

                        defineHandler(options));

                if (options.isSyncLoading()) {

                    displayTask.run();

                } else {

                    engine.submit(displayTask);

                }

            } else {

                options.getDisplayer().display(bmp, imageAware, LoadedFro
m.MEMORY_CACHE);

                listener.onLoadingComplete(uri, imageAware.getWrappedView
(), bmp);

            }

        }

        ...//下一部分

    }
```

当 URI 不为空的时候来加载显示。首先根据 uri 获取对应 uri 对应唯一的一个 Key，之后调用 `engine.prepareDisplayTaskFor(imageAware, memoryCacheKey)`；来记录当前加载的任务，开启 listener 的 start 回调，接着调用 `Bitmap bmp =`

BAT交流群 892246

configuration.memoryCache.get(memoryCacheKey);来获取内存缓存中的图片，这里默认的内存缓存是 LruMemoryCache，前篇文章有分析到。

如果缓存中存在相应的 Bitmap 的话，进入到 if 里面
我们如果在 DisplayImageOptions 中设置了 postProcessor 就进入 true 逻辑，不过默认 postProcessor 是为 null 的，BitmapProcessor 接口主要是对 Bitmap 进行处理，这个框架并没有给出相对应的实现，如果我们有自己的需求的时候可以自己实现 BitmapProcessor 接口（比如将图片设置成圆形的）.

然后到了 27 行
将 Bitmap 设置到 ImageView 上面，这里我们可以在 DisplayImageOptions 中配置显示需求 displayer，默认使用的是 SimpleBitmapDisplayer，直接将 Bitmap 设置到 ImageView 上面，我们可以配置其他的显示逻辑， 他这里提供了 FadeInBitmapDisplayer（透明度从 0-1）RoundedBitmapDisplayer（4 个角是圆弧）等， 然后回调到 ImageLoadingListener 接口。

我们知道 loadImage 和 displayImage 的区别在于 loadImage 依靠返回的 Bitmap 进行设置显示，而 displayImage 则是直接显示。而 loadImage 最终也是调用了 displayImage，原因就在于这个 display 和 imageAware

```
public final class SimpleBitmapDisplayer implements BitmapDisplayer {

    @Override

    public void display(Bitmap bitmap, ImageAware imageAware, LoadedFrom
 loadedFrom) {

        imageAware.setImageBitmap(bitmap);

    }}
```

loadImage 的 ImageAware 是 NonImageAware 并没有处理 setImageBitmap 的方法，而 displayImage 的 ImageViewAware 则有处理显示。

好，继续前面，当从内存缓存获取到的 Bitmap 为空的情况下

```
public void displayImage(String uri, ImageAware imageAware, DisplayImage
Options options,

            ImageSize targetSize, ImageLoadingListener listener, ImageLoa
dingProgressListener progressListener) {

            ...//前两部分
```

```
        //如果 Bitmap 为空

        } else {

            if (options.shouldShowImageOnLoading()) {

                imageAware.setImageDrawable(options.getImageOnLoading(con
figuration.resources));

            } else if (options.isResetViewBeforeLoading()) {

                imageAware.setImageDrawable(null);

            }



            ImageLoadingInfo imageLoadingInfo = new ImageLoadingInfo(uri,
 imageAware, targetSize, memoryCacheKey,

                    options, listener, progressListener, engine.getLockFor
Uri(uri));

            LoadAndDisplayImageTask displayTask = new LoadAndDisplayImage
Task(engine, imageLoadingInfo,

                    defineHandler(options));

            if (options.isSyncLoading()) {

                displayTask.run();

            } else {

                engine.submit(displayTask);

            }

        }

    }
```

如果需要设置显示加载中的图片，则进行设置显示。
`ImageLoadingInfo` 则是一个加载显示图片任务信息的一个类。
之后根据它创建了一个 `LoadAndDisplayImageTask` 类，它实现了 `Runnable`。

如果配置了 isSyncLoading 为 true，直接执行 LoadAndDisplayImageTask 的 run 方法，表示同步，默认是 false,将 LoadAndDisplayImageTask 提交给线程池对象

接下来我们就看 LoadAndDisplayImageTask 的 run()，这个类还是蛮复杂的，我们还是一段一段的分析。

```
@Override

    public void run() {

        if (waitIfPaused()) return;

        if (delayIfNeed()) return;



        ...

    }

如果 waitIfPaused(), delayIfNeed()返回 true 的话，直接从 run()方法中返回了，

不执行下面的逻辑，接下来我们先看看 waitIfPaused（）

private boolean waitIfPaused() {

        AtomicBoolean pause = engine.getPause();

        if (pause.get()) {

            synchronized (engine.getPauseLock()) {

                if (pause.get()) {

                    L.d(LOG_WAITING_FOR_RESUME, memoryCacheKey);

                    try {

                        engine.getPauseLock().wait();

                    } catch (InterruptedException e) {

                        L.e(LOG_TASK_INTERRUPTED, memoryCacheKey);

                        return true;
```

```
                }

                L.d(LOG_RESUME_AFTER_PAUSE, memoryCacheKey);

            }

        }

    }

    return isTaskNotActual();

}
```

这个方法是干嘛用呢，主要是我们在使用 ListView，GridView 去加载图片的时候，有时候为了滑动更加的流畅，我们会选择手指在滑动或者猛地一滑动的时候不去加载图片，所以才提出了这么一个方法，那么要怎么用呢？ 这里用到了 PauseOnScrollListener 这个类，使用很简单 ListView.setOnScrollListener(new PauseOnScrollListener(pauseOnScroll, pauseOnFling))， pauseOnScroll 控制我们缓慢滑动 ListView，GridView 是否停止加载图片，pauseOnFling 控制猛的滑动 ListView，GridView 是否停止加载图片。

我们可以看下这个 `PauseOnScrollListener` 的处理

```
@Override

    public void onScrollStateChanged(AbsListView view, int scrollState)
{

        switch (scrollState) {

            case OnScrollListener.SCROLL_STATE_IDLE:

                imageLoader.resume();

                break;

            case OnScrollListener.SCROLL_STATE_TOUCH_SCROLL:

                if (pauseOnScroll) {

                    imageLoader.pause();

                }
```

```
                break;

            case OnScrollListener.SCROLL_STATE_FLING:

                if (pauseOnFling) {

                    imageLoader.pause();

                }

                break;

        }

        if (externalListener != null) {

            externalListener.onScrollStateChanged(view, scrollState);

        }

    }

滑动停止的话会调用到imageLoader.pause
public void pause() {

        engine.pause();

    }...void pause() {

        paused.set(true);

    }

这里的pause是private final AtomicBoolean paused = new AtomicBoolean(fals
e);
```

所以调用 `pause.get` 则会返回 true。

除此之外，这个方法的返回值由 isTaskNotActual()决定，我们接着看看

isTaskNotActual()的源码

```
private boolean isTaskNotActual() {
```

```
        return isViewCollected() || isViewReused();

    }

private boolean isTaskNotActual() {

        return isViewCollected() || isViewReused();

    }
```

isViewCollected()是判断我们 ImageView 是否被垃圾回收器回收了，如果回收了，LoadAndDisplayImageTask 方法的 run()就直接返回了，isViewReused()判断该 ImageView 是否被重用，被重用 run()方法也直接返回，为什么要用 isViewReused()方法呢？主要是 ListView，GridView 我们会复用 item 对象，假如我们先去加载 ListView，GridView 第一页的图片的时候，第一页图片还没有全部加载完我们就快速的滚动，isViewReused()方法就会避免这些不可见的 item 去加载图片，而直接加载当前界面的图片。

回头继续看 run 方法

```
@Override

    public void run() {

        ...

        ReentrantLock loadFromUriLock = imageLoadingInfo.loadFromUriLoc
k;

        L.d(LOG_START_DISPLAY_IMAGE_TASK, memoryCacheKey);

        if (loadFromUriLock.isLocked()) {

            L.d(LOG_WAITING_FOR_IMAGE_LOADED, memoryCacheKey);

        }



        loadFromUriLock.lock();

        Bitmap bmp;

        try {
```

```
            checkTaskNotActual();



            bmp = configuration.memoryCache.get(memoryCacheKey);

            if (bmp == null || bmp.isRecycled()) {

                bmp = tryLoadBitmap();

                if (bmp == null) return; // listener callback already was
fired



                checkTaskNotActual();

                checkTaskInterrupted();



                if (options.shouldPreProcess()) {

                    L.d(LOG_PREPROCESS_IMAGE, memoryCacheKey);

                    bmp = options.getPreProcessor().process(bmp);

                    if (bmp == null) {

                        L.e(ERROR_PRE_PROCESSOR_NULL, memoryCacheKey);

                    }

                }



                if (bmp != null && options.isCacheInMemory()) {

                    L.d(LOG_CACHE_IMAGE_IN_MEMORY, memoryCacheKey);

                    configuration.memoryCache.put(memoryCacheKey, bmp);

                }
```

```
            } else {
                loadedFrom = LoadedFrom.MEMORY_CACHE;
                L.d(LOG_GET_IMAGE_FROM_MEMORY_CACHE_AFTER_WAITING, memory
CacheKey);
            }


            if (bmp != null && options.shouldPostProcess()) {
                L.d(LOG_POSTPROCESS_IMAGE, memoryCacheKey);
                bmp = options.getPostProcessor().process(bmp);
                if (bmp == null) {
                    L.e(ERROR_POST_PROCESSOR_NULL, memoryCacheKey);
                }
            }
            checkTaskNotActual();
            checkTaskInterrupted();
        } catch (TaskCancelledException e) {
            fireCancelEvent();
            return;
        } finally {
            loadFromUriLock.unlock();
        }


        DisplayBitmapTask displayBitmapTask = new DisplayBitmapTask(bmp,
imageLoadingInfo, engine, loadedFrom);
```

```
        runTask(displayBitmapTask, syncLoading, handler, engine);

    }
```

第 4 行代码有一个 loadFromUriLock，这个是一个锁，获取锁的方法在

ImageLoaderEngine 类的 getLockForUri()方法中 c

```
ReentrantLock getLockForUri(String uri) {

        ReentrantLock lock = uriLocks.get(uri);

        if (lock == null) {

            lock = new ReentrantLock();

            uriLocks.put(uri, lock);

        }

        return lock;

    }

ReentrantLock getLockForUri(String uri) {

        ReentrantLock lock = uriLocks.get(uri);

        if (lock == null) {

            lock = new ReentrantLock();

            uriLocks.put(uri, lock);

        }

        return lock;

    }
```

从上面可以看出，这个锁对象与图片的 url 是相互对应的，为什么要这么做？也行你还有点不理解，不知道大家有没有考虑过一个场景，假如在一个 ListView 中，某个 item 正在获取图片的过程中，而此时我们将这个 item 滚出界面之后又将其滚进来，滚进来之后如果没有加锁，该 item 又会去加载一次图片，假设在很短的时间内滚动很频繁，那么就会出现多次去网络上面请求图片，所以这里根据图

片的 Url 去对应一个 ReentrantLock 对象，让具有相同 Url 的请求就会在第 10 行等待，等到这次图片加载完成之后，ReentrantLock 就被释放，刚刚那些相同 Url 的请求就会继续执行第 10 行下面的代码。

之后来到第 13 行，先调用 `checkTaskNotActual` 判断当前 View 是否被 GC 回收使用，是则抛出异常。
接着 15 行，它们会先从内存缓存中获取一遍，如果内存缓存中没有在去执行下面的逻辑，所以 ReentrantLock 的作用就是避免这种情况下重复的去从网络上面请求图片。

17 行的方法 `tryLoadBitmap()`,这个方法确实也有点长，我先告诉大家，这里面的逻辑是先从文件缓存中获取有没有 Bitmap 对象，如果没有在去从网络中获取，然后将 bitmap 保存在文件系统中，我们还是具体分析下

```
private Bitmap tryLoadBitmap() throws TaskCancelledException {

        Bitmap bitmap = null;

        try {

            File imageFile = configuration.diskCache.get(uri);

            if (imageFile != null && imageFile.exists() && imageFile.leng
th() > 0) {

                L.d(LOG_LOAD_IMAGE_FROM_DISK_CACHE, memoryCacheKey);

                loadedFrom = LoadedFrom.DISC_CACHE;



                checkTaskNotActual();

                bitmap = decodeImage(Scheme.FILE.wrap(imageFile.getAbsolu
tePath()));

            }

            if (bitmap == null || bitmap.getWidth() <= 0 || bitmap.getHei
ght() <= 0) {

                L.d(LOG_LOAD_IMAGE_FROM_NETWORK, memoryCacheKey);

                loadedFrom = LoadedFrom.NETWORK;
```

```
            String imageUriForDecoding = uri;

            if (options.isCacheOnDisk() && tryCacheImageOnDisk()) {

                imageFile = configuration.diskCache.get(uri);

                if (imageFile != null) {

                    imageUriForDecoding = Scheme.FILE.wrap(imageFile.g
etAbsolutePath());

                }

            }



            checkTaskNotActual();

            bitmap = decodeImage(imageUriForDecoding);



            if (bitmap == null || bitmap.getWidth() <= 0 || bitmap.get
Height() <= 0) {

                fireFailEvent(FailType.DECODING_ERROR, null);

            }

        }

    }

    ...

    return bitmap;

}
```

首先在第 4 行会去磁盘缓存中去获取图片，如果图片已经保存在磁盘了，则直接获取对应的 File 路径，调用 `bitmap = decodeImage(Scheme.FILE.wrap(imageFile.getAbsolutePath()));`进行解析。

如果在磁盘中没有的话，则到了 12 行，开始进行网络下载获取。
在 17 行会去调用 `isCacheOnDisk` 判断是否要保持在磁盘中，如果默认 false，如果是则调用 `tryCacheImageOnDisk` 来下载图片并且保持在磁盘

```
private boolean tryCacheImageOnDisk() throws TaskCancelledException {

        L.d(LOG_CACHE_IMAGE_ON_DISK, memoryCacheKey);



        boolean loaded;

        try {

            loaded = downloadImage();

            ...

        } ...

        return loaded;

    }
```

调用了 `downloadImage` 进行下载图片

```
private boolean downloadImage() throws IOException {

        InputStream is = getDownloader().getStream(uri, options.getExtraF
orDownloader());

        if (is == null) {

            L.e(ERROR_NO_IMAGE_STREAM, memoryCacheKey);

            return false;

        } else {

            try {

                return configuration.diskCache.save(uri, is, this);

            } finally {

                IoUtils.closeSilently(is);
```

```
            }

        }

    }
```

可以看到这里调用了 `getDownloader().getStream` 来下载，这里先不扩展，在后面会说到

下载之后则保存在磁盘中。

回来前面

```
String imageUriForDecoding = uri;

                if (options.isCacheOnDisk() && tryCacheImageOnDisk()) {

                    imageFile = configuration.diskCache.get(uri);

                    if (imageFile != null) {

                        imageUriForDecoding = Scheme.FILE.wrap(imageFile.g
etAbsolutePath());

                    }

                }

                checkTaskNotActual();

                bitmap = decodeImage(imageUriForDecoding);
```

这里有个 String 变量 imageUriForDecoding，初始值是 uri，如果有设置磁盘缓存的话，则会调用 `tryCacheImageOnDisk` 来下载并且保持图片，此时的 imageUriForDecoding 则是文件 File 的路径。

如果没有设置磁盘缓存的话，则 imageUriForDecoding 还是 uri。
关键则是在 `decodeImage`，它能根据对应的 uri 来加载图片。

```
    private Bitmap decodeImage(String imageUri) throws IOException {

        ViewScaleType viewScaleType = imageAware.getScaleType();

        ImageDecodingInfo decodingInfo = new ImageDecodingInfo(memoryCach
eKey, imageUri, uri, targetSize, viewScaleType,

                getDownloader(), options);
```

```
        return decoder.decode(decodingInfo);

    }
```

把传递进来的 imageUri（可能是文件的 uri，也可能是图片的 uri）封装到 ImageDecodingInfo 进行解析。
这里的 decoder 是 ImageDecode，它的默认实现类是 BaseImageDecode

```
@Override

    public Bitmap decode(ImageDecodingInfo decodingInfo) throws IOExcept
ion {

        Bitmap decodedBitmap;

        ImageFileInfo imageInfo;



        InputStream imageStream = getImageStream(decodingInfo);

        ...}
```

可以看到，在这里，已经做了多种情况的读取判断。第一篇文章就有介绍到 UIL 可以根据不同的 uri 来解析图片，其原理就是在这里。
而前面通过 `tryCacheImageOnDisk` 来下载图片也是根据这个。这里就不一一扩展开。
这里的网络下载图片内部则是使用 `HttpUrlConnection` 来下载的。

回到最前面 `LoadAndDisplayImageTask` 的 run 方法后面，当我们获取到 Bitmap 后，到了

```
    DisplayBitmapTask displayBitmapTask = new DisplayBitmapTask(bmp, ima
geLoadingInfo, engine, loadedFrom);

        runTask(displayBitmapTask, syncLoading, handler, engine);

这两个代码就是一个显示任务

直接看 DisplayBitmapTask 类的 run（）方法

    @Override

    public void run() {

        if (imageAware.isCollected()) {
```

```
            L.d(LOG_TASK_CANCELLED_IMAGEAWARE_COLLECTED, memoryCacheKey);

            listener.onLoadingCancelled(imageUri, imageAware.getWrappedVi
ew());

        } else if (isViewWasReused()) {

            L.d(LOG_TASK_CANCELLED_IMAGEAWARE_REUSED, memoryCacheKey);

            listener.onLoadingCancelled(imageUri, imageAware.getWrappedVi
ew());

        } else {

            L.d(LOG_DISPLAY_IMAGE_IN_IMAGEAWARE, loadedFrom, memoryCacheK
ey);

            displayer.display(bitmap, imageAware, loadedFrom);

            engine.cancelDisplayTaskFor(imageAware);

            listener.onLoadingComplete(imageUri, imageAware.getWrappedVie
w(), bitmap);

        }

    }
```

这两个代码就是一个显示任务

直接看 DisplayBitmapTask 类的 run（）方法

假如 ImageView 被回收了或者被重用了，回调给 ImageLoadingListener 接口，否则就调用 BitmapDisplayer 去显示 Bitmap。到这里 Bitmap 已经显示加载完成，调用 engine 移除图片显示任务。

当然在最前面那里

如果此时的显示加载是异步的话，则交由 engine 的 `Executor` 线程池去处理，最终也是调用了 `LoadAndDisplayImageTask` 的 run 方法去加载显示。

到这里 Universal-Image-Loader 的分析也算完了，从基本使用到内存模型在加载显示，可以看到 UIL 这个开源框架十分的灵活，比如建造者模式，装饰模式，代

BAT交流群：892872246

理模式，策略模式等等，这样方便我们去扩展，实现我们想要的功能，当然，也带给我们更多的想象空间。

4.1.2 Glide

文章将通过 Glide 使用来阐述图片加载框架选型：

首先，添加依赖

```
implementation 'com.github.bumptech.glide:glide:4.5.0'

annotationProcessor 'com.github.bumptech.glide:compiler:4.5.0'
```

之后添加访问网络权限

```
<uses-permission android:name="android.permission.INTERNET" />
```

一、常用的方法

1、加载图片到 imageView

```
Glide.with(Context context).load(Strint url).into(ImageView imageView);
```

2、各种形式的图片加载到 ImageView

```
// 加载本地图片 File file = new File(getExternalCacheDir() + "/image.jpg
");

Glide.with(this).load(file).into(imageView);

// 加载应用资源 int resource = R.drawable.image;

Glide.with(this).load(resource).into(imageView);

// 加载二进制流 byte[] image = getImageBytes();

Glide.with(this).load(image).into(imageView);

// 加载 Uri 对象 Uri imageUri = getImageUri();
```

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

3、加载带有占位图

```
Glide.with(this).load(url).placeholder(R.drawable.loading).into(imageVi
ew);
```

占位图目的为在目的图片还未加载出来的时候，提前展示给用户的一张图片；

4、加载失败 放置占位符

```
Glide.with(this).load(url).placeholder(R.drawable.loading).error(R.draw
able.error)

     .diskCacheStrategy(DiskCacheStrategy.NONE)//关闭 Glide 的硬盘缓存机制

     .into(imageView);

//DiskCacheStrategy.NONE： 表示不缓存任何内容。//DiskCacheStrategy.SOURCE:
 表示只缓存原始图片。//DiskCacheStrategy.RESULT： 表示只缓存转换过后的图片（默
认选项）。//DiskCacheStrategy.ALL ： 表示既缓存原始图片，也缓存转换过后的图片。
```

4、加载指定格式的图片--指定为静止图片

```
Glide.with(this)

     .load(url)

     .asBitmap()//只加载静态图片，如果是 git 图片则只加载第一帧。

     .placeholder(R.drawable.loading)

     .error(R.drawable.error)

     .diskCacheStrategy(DiskCacheStrategy.NONE)

     .into(imageView);
```

5、加载动态图片

```
Glide.with(this)

     .load(url)

     .asGif()//加载动态图片，若现有图片为非 gif 图片，则直接加载错误占位图。
```

BA 92872246

```
    .placeholder(R.drawable.loading)

    .error(R.drawable.error)

    .diskCacheStrategy(DiskCacheStrategy.NONE)

    .into(imageView);
```

6、加载动态图片

```
Glide.with(this)

    .load(url)

    .asGif()//加载动态图片，若现有图片为非 gif 图片，则直接加载错误占位图。

    .placeholder(R.drawable.loading)

    .error(R.drawable.error)

    .diskCacheStrategy(DiskCacheStrategy.NONE)

    .into(imageView);
```

7、加载指定大小的图片

```
Glide.with(this)

    .load(url)

    .placeholder(R.drawable.loading)

    .error(R.drawable.error)

    .diskCacheStrategy(DiskCacheStrategy.NONE)

    .override(100, 100)//指定图片大小

    .into(imageView);
```

8、关闭框架的内存缓存机制

```
Glide.with(this)

    .load(url)
```

```
    .skipMemoryCache(true)  //传入参数为 false 时，则关闭内存缓存。

    .into(imageView);
```

9、关闭硬盘的缓存

```
Glide.with(this)

    .load(url)

    .diskCacheStrategy(DiskCacheStrategy.NONE)     //关闭硬盘缓存操作

    .into(imageView);

//其他参数表示：//DiskCacheStrategy.NONE： 表示不缓存任何内容。//DiskCacheS
trategy.SOURCE： 表示只缓存原始图片。//DiskCacheStrategy.RESULT： 表示只缓存
转换过后的图片（默认选项）。//DiskCacheStrategy.ALL ： 表示既缓存原始图片，也
缓存转换过后的图片。
```

10、当引用的 url 存在 token 时解决方法-->重写 Glide 的 GlideUrl 方法

```
public class MyGlideUrl extends GlideUrl {



    private String mUrl;



    public MyGlideUrl(String url) {

        super(url);

        mUrl = url;

    }



    @Override

    public String getCacheKey() {

        return mUrl.replace(findTokenParam(), "");
```

```
    }



    private String findTokenParam() {

        String tokenParam = "";

        int tokenKeyIndex = mUrl.indexOf("?token=") >= 0 ? mUrl.indexOf("?
token=") : mUrl.indexOf("&token=");

        if (tokenKeyIndex != -1) {

            int nextAndIndex = mUrl.indexOf("&", tokenKeyIndex + 1);

            if (nextAndIndex != -1) {

                tokenParam = mUrl.substring(tokenKeyIndex + 1, nextAndInd
ex + 1);

            } else {

                tokenParam = mUrl.substring(tokenKeyIndex);

            }

        }

        return tokenParam;

    }

}
```

然后加载图片的方式为：

```
Glide.with(this)

     .load(new MyGlideUrl(url))

     .into(imageView);
```

11、利用 Glide 将图片加载到不同控件或加载成不同使用方式

（1）、拿到图片实例

BAT交流群

```
//1、通过自己构造 target 可以获取到图片实例

SimpleTarget<GlideDrawable> simpleTarget = new SimpleTarget<GlideDrawabl
e>() {

    @Override

    public void onResourceReady(GlideDrawable resource, GlideAnimation g
lideAnimation) {

        imageView.setImageDrawable(resource);

    }};

//2、将图片实例记载到指定的 imageview 上，也可以做其他的事情 public void loadI
mage(View view) {

    String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473
_1920x1080.jpg";

    Glide.with(this)

         .load(url)

         .into(simpleTarget);}
```

（2）、将图片加载到任何位置

```
/*

*将图片加载为控件背景

*/public class MyLayout extends LinearLayout {



    private ViewTarget<MyLayout, GlideDrawable> viewTarget;



    public MyLayout(Context context, AttributeSet attrs) {

        super(context, attrs);

        viewTarget = new ViewTarget<MyLayout, GlideDrawable>(this) {
```

```
            @Override

            public void onResourceReady(GlideDrawable resource, GlideAnim
ation glideAnimation) {

                MyLayout myLayout = getView();

                myLayout.setImageAsBackground(resource);

            }

        };

    }



    public ViewTarget<MyLayout, GlideDrawable> getTarget() {

        return viewTarget;

    }



    public void setImageAsBackground(GlideDrawable resource) {

        setBackground(resource);

    }

}

//引用图片到指定控件作为背景 public class MainActivity extends AppCompatAct
ivity {



    MyLayout myLayout;



    @Override

    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        myLayout = (MyLayout) findViewById(R.id.background);

    }



    public void loadImage(View view) {

        String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN733679
5473_1920x1080.jpg";

        Glide.with(this)

            .load(url)

            .into(myLayout.getTarget());

    }

}
```

12、Glide 实现预加载

```
//a、预加载代码

Glide.with(this)

     .load(url)

     .diskCacheStrategy(DiskCacheStrategy.SOURCE)

     .preload();

//preload() 有两种重载

 // 1、带有参数的重载,参数作用是设置预加载的图片大小; //2、不带参数的表示加载的
图片为原始尺寸;

//b、使用预加载的图片

Glide.with(this)
```

```
    .load(url)

    .diskCacheStrategy(DiskCacheStrategy.SOURCE)

    .into(imageView);
```

切记：diskCacheStrategy() 方法内必须设置参数为：“ DiskCacheStrategy.SOURCE ”，否则可能预加载失败，导致显示图片时，需要重新加载。

13、Glide 实现图片下载

使用 `downloadOnly(int width, int height)` 或 `downloadOnly(Y target)` 方法替代 `into(view)` 方法。

```
public void downloadImage(View view) {

    new Thread(new Runnable() {

        @Override

        public void run() {

            try {

                String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-C
N7336795473_1920x1080.jpg";

                final Context context = getApplicationContext();

                FutureTarget<File> target = Glide.with(context)

                                            .load(url)

                                            .downloadOnly(Target.SIZE_O
RIGINAL, Target.SIZE_ORIGINAL);

                final File imageFile = target.get();

                runOnUiThread(new Runnable() {

                    @Override

                    public void run() {

                        Toast.makeText(context, imageFile.getPath(), Toast.
LENGTH_LONG).show();
```

```
                }

            });

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}).start();}
```

(1)、有两个参数的 `downloadOnly(int width, int height)` 方法表示指定下载尺寸，用于在子线程内进行下载；
(2)、一个参数的 `downloadOnly(Y target)` 方法 在主线程内进行下载
(3)、`target.get()` 方法可以获取到下载文件保存路径；

使用下载完的图片的方式

```
public void loadImage(View view) {

    String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473
_1920x1080.jpg";

    Glide.with(this)

            .load(url)

            .diskCacheStrategy(DiskCacheStrategy.SOURCE)

            .into(imageView);}
```

注意： diskCacheStrategy() 方法的参数应该为 DiskCacheStrategy.SOURCE 或者 DiskCacheStrategy.ALL 否则可能导致加载图片到控件的时候，需要重新加载。

13、监听 Glide 加载的状态

```
public void loadImage(View view) {

    String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473
_1920x1080.jpg";
```

892872246

```
    Glide.with(this)

            .load(url)

            .listener(new RequestListener<String, GlideDrawable>() {

                @Override

                public boolean onException(Exception e, String model, Tar
get<GlideDrawable> target,

                    boolean isFirstResource) {

                    return false;

                }



                @Override

                public boolean onResourceReady(GlideDrawable resource, St
ring model,

                    Target<GlideDrawable> target, boolean isFromMemoryCach
e, boolean isFirstResource) {

                    return false;

                }

            })

            .into(imageView);

}
```

（1）、onException() 方法表示加载失败，onResourceReady() 表示加载成功；
（2）、 每个方法都有一个 boolean 的返回值，false 表示未处理、true 表示处理。

14、Glide 的图形变换功能
（1）、禁用图形变换功能

```
Glide.with(this)
```

IT交流群：892872246

```
    .load(url)

    .dontTransform()

    .into(imageView);
```

这个方法时全局的，导致其他地方的图片也不可进行图形变换了。
修改方法

```
Glide.with(this)

    .load(url)

    .override(Target.SIZE_ORIGINAL, Target.SIZE_ORIGINAL)

    .into(imageView);
```

通过 override() 方法设置大小

（3）、简单的图形变换

```
Glide.with(this)

    .load(url)

    .centerCrop()

    .into(imageView);



Glide.with(this)

    .load(url)

    .fitCenter()

    .into(imageView);
```

通过 centerCrop()方法 按照原始的长宽比充满全屏和 fitCenter() 方法 对原图的中心区域进行裁剪对图片进行相关设置。

（4）、override() 方法与 centerCrop() 方法配合使用

BAT交流群892246

```
String url = "http://cn.bing.com/az/hprichbg/rb/AvalancheCreek_ROW11173354624_1920x1080.jpg";

Glide.with(this)

    .load(url)

    .override(500, 500)

    .centerCrop()

    .into(imageView);
```

image

（4）、复杂的图像变换

首先需要再引入一个 第三方框架 。

```
dependencies {

    implementation 'jp.wasabeef:glide-transformations:3.3.0'

    // If you want to use the GPU Filters

    implementation 'jp.co.cyberagent.android.gpuimage:gpuimage-library:1.
4.1'}
```

其次,部分样例:

图片虚化

```
Glide.with(this)

     .load(url)

     .bitmapTransform(new BlurTransformation(this))

     .into(imageView);
```

图片黑白化

```
Glide.with(this)

     .load(url)

     .bitmapTransform(new GrayscaleTransformation(this))

     .into(imageView);
```

多个属性同时使用

```
Glide.with(this)

     .load(url)

     .bitmapTransform(new BlurTransformation(this), new GrayscaleTransfo
rmation(this))

     .into(imageView);
```

4.1.3 Picasso

BAT交流 892246

一、 Picasso 分析

<meta charset="utf-8">

Picasso 是 Square 公司开源的一个 Android 平台上的图片加载框架

核心类：

Picasso：负责图片下载、变换、缓存的管理器，当它收到一个图片下载请求的时候，它会创建 Request 并提交给 Dispatcher

Dispatcher: Dispatcher 会寻找对应的处理器 RequestHandler，并将请求与该处理器一起提交给线程池执行

BAT交流群：892872246

图片加载步骤：

创建 -> 入队 -> 执行 -> 解码 -> 变化 -> 批处理 -> 完成 -> 分发 -> 显示

使用

```
Picasso.get().load(url).into(first_img)

//加载一张图片 旋转 45°

Picasso.get()

        .load(URL)

        .rotate(45f)

        .placeholder(R.drawable.placeholder_disk)

        .into(imgOne);
```

```
//加载一张图片并设置一个回调接口

Picasso.get()

        .load(URL)

        .placeholder(R.drawable.placeholder_disk)

        .into(imgOne, new Callback() {



            @Override

            public void onSuccess() {



            }



            @Override

            public void onError(Exception e) {



            }

        });

//预加载一张图片

Picasso.get().load(URL).fetch();

//同步加载一张图片,注意只能在子线程中调用并且Bitmap不会被缓存到内存里.new Thr
ead() {

    @Override

    public void run() {

        try {
```

```
            final Bitmap bitmap = Picasso.get().load(URL).get();

        } catch (IOException e) {

            e.printStackTrace();

        }

    }}.start();

//加载一张图片旋转并且添加一个 Transformation,可以对图片进行各种变化处理,例如圆形头像.

Picasso.get().load("url").rotate(10).transform(new Transformation() {

    @Override

    public Bitmap transform(Bitmap source) {

        //处理 Bitmap

        return null;

    }



    @Override

    public String key() {

        return null;

    }}).into(imgTwo);

//加载一张图片并按照指定尺寸以 centerInside()的形式缩放.并设置加载的优先级为高.注意 centerInside()或 centerCrop()//只能同时使用一种,而且必须指定 resize()或者 resizeDimen();

Picasso.get().load(URL)

        .resize(400, 400)

        .centerInside()
```

```
        .priority(Picasso.Priority.HIGH)

        .into(imgTwo);

//加载一张图片并设置 tag,可以通过 tag 来暂定或者继续加载,可以用于当 ListView 滚动是暂定加载.停止滚动恢复加载.

Picasso.get().load(URL).tag(getContext()).into(imgTwo);

Picasso.get().pauseTag(getContext());

Picasso.get().resumeTag(getContext());
```

二、Fresco 的使用

当下有很多图片加载框架，常见的有 Glide、Fresco、Picasso 等。Glide 因为其体积小、缓存机制强大等优点，受到了广大程序员的青睐；Fresco 虽然体积比较大，缓存机制也没有 Glide 强大，但它胜在拥有一些炫酷的效果（进度条、淡入效果）等，也有很多人在使用。今天我们就来介绍一下 Fresco 的使用。

1、配置环境

使用 Fresco 需要先在 build.gradle 中导入依赖：

```
// Fresco 所需依赖

   compile 'com.facebook.fresco:fresco:0.12.0'

   // 在 API < 14 上的机器支持 WebP 时，需要添加

   compile 'com.facebook.fresco:animated-base-support:0.12.0'

   // 支持 GIF 动图，需要添加

   compile 'com.facebook.fresco:animated-gif:0.12.0'

   // 支持 WebP （静态图+动图），需要添加

   compile 'com.facebook.fresco:animated-webp:0.12.0'

   compile 'com.facebook.fresco:webpsupport:0.12.0'
```

如果需要从网络中下载图片，则需要在 AndroidManifest.xml 文件中配置网络权限：

```
<uses-permission android:name="android.permission.INTERNET" />
```

最后，在项目的 Application 中初始化 Fresco：

```
public class MyApplication extends Application {



    @Override

    public void onCreate() {

        super.onCreate();

        // 初始化 Fresco

        Fresco.initialize(this);

    }}
```

别忘了在 AndroidManifest.xml 文件中为 Application 添加 name 属性：

```
<application

        android:name=".MyApplication"

        ......

    </application>
```

2、SimpleDraweeView

2.1、XML 属性

Fresco 为我们提供了一个 SimpleDraweeView 控件，我们可以直接在这个控件中加载图片。Fresco 在这个控件中集成了很多属性，简单实用。以下是 SimpleDraweeView 中的常用属性：

```
fresco:actualImageScaleType：实际加载的图片的伸缩样式
```

IT交流群：892872246

```
        fresco:backgroundImage：底层图片资源

        fresco:fadeDuration：进度条和占位符图片逐渐消失、加载的图片逐渐显示的时间间隔

        fresco:failureImage：加载失败时显示的图片资源

        fresco:failureImageScaleType：加载失败时加载的图片的伸缩样式

        fresco:overlayImage：在显示的图片表层覆盖一张图片的图片资源

        fresco:placeholderImage：占位符图片资源

        fresco:placeholderImageScaleType：占位符图片的伸缩样式

        fresco:progressBarAutoRotateInterval：进度条图片转动周期

        fresco:progressBarImage：进度条图片资源

        fresco:progressBarImageScaleType：进度条图片的伸缩样式

        fresco:retryImage：提示重新加载的图片资源

        fresco:retryImageScaleType：提示重新加载的图片的伸缩样式

        fresco:roundAsCircle：将图片剪切成圆形

        fresco:viewAspectRatio：图片宽高比
```

我们的 XML 布局文件中的代码如下：

```
<?xml version="1.0" encoding="utf-8"?><RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

    xmlns:fresco="http://schemas.android.com/apk/res-auto"

    android:id="@+id/activity_main"

    android:layout_width="match_parent"

    android:layout_height="match_parent">
```

```
<com.facebook.drawee.view.SimpleDraweeView
    android:id="@+id/id_main_sdv_sdv"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    fresco:actualImageScaleType="focusCrop"
    fresco:fadeDuration="3000"
    fresco:failureImage="@mipmap/ic_launcher"
    fresco:failureImageScaleType="centerInside"
    fresco:placeholderImage="@mipmap/ic_launcher"
    fresco:placeholderImageScaleType="fitCenter"
    fresco:progressBarAutoRotateInterval="1000"
    fresco:progressBarImage="@mipmap/ic_launcher"
    fresco:progressBarImageScaleType="centerInside"
    fresco:retryImage="@mipmap/ic_launcher"
    fresco:retryImageScaleType="centerCrop"
    fresco:roundAsCircle="false"
    fresco:viewAspectRatio="1.6" />


<Button
    android:id="@+id/id_main_btn_load"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
```

```
        android:layout_margin="20.0dip"

        android:text="Start Load Image" />

</RelativeLayout>
```

2.2、注意事项

（1）SimpleDraweeView 控件的宽高不能设置为 wrap_content，只能是 match_parent、具体值或使用 viewAspectRatio 属性设置宽高比。

（2）如果在一个页面中加载多张图片，不要将 SimpleDraweeView 直接放在 ScollView 中，建议使用 RecyclerView、ListView 或 GridView，因为 ScrollView 中的 SimpleDraweeView 会持有内存直到 Activity 或 Fragment 销毁。

（3）使用 SimpleDraweeView 时，不要使用 imageView 中有、View 中没有的属性。

3、框架使用

3.1、简单使用

像 TextView、Button 等其他控件一样后去到 SimpleDraweeView，然后调用如下代码即可加载网络图片：

```
SimpleDraweeView sdv = (SimpleDraweeView) findViewById(R.id.id_main_sdv_sdv);
sdv.setImageURI("http://image5.tuku.cn/pic/wallpaper/fengjing/menghuandaziranmeijingbizhi/009.jpg");
```

3.2、JAVA 代码设置属性

我们可以通过 GenericDraweeHierarchy，在 JAVA 代码中动态的设置 SimpleDraweeView 控件的属性。需要注意的是，如果在 JAVA 代码中设置了属性，那么 XML 文件中设置的属性就都无效了。一个实例代码如下：

```
// 代码设置 SimpleDraweeView 的属性（会覆盖 XML 设置的所有属性，即在 XML 中有在这里没有的属性都会失效）

        // 注意：一个 GenericDraweeHierarchy 是不能被多个 SimpleDraweeView 共用的

        GenericDraweeHierarchy hierarchy = new GenericDraweeHierarchyBuilder(getResources())

                .setFadeDuration(3000)

                .setPlaceholderImage(R.mipmap.ic_launcher)
```

```
                .setPlaceholderImageScaleType(ScalingUtils.ScaleType.FIT_XY)

                .setProgressBarImage(new ProgressBarDrawable()) // 显示进度条（Fresco 自带的进度条）

                .build();

        // 设置图片圆角

        RoundingParams roundingParams = new RoundingParams();

        roundingParams.setRoundAsCircle(false); // 不将图片剪切成圆形

        roundingParams.setCornersRadius(200);

        hierarchy.setRoundingParams(roundingParams);

        // 为 SimpleDraweeView 设置属性

        sdv.setHierarchy(hierarchy);
```

注意：不能把同一个 GenericDraweeHierarchy 对象设置给多个 SimpleDraweeView！

3.3、设置下载事件

```
        sdv.setController(Fresco.newDraweeControllerBuilder()

                .setControllerListener(new BaseControllerListener<ImageInfo>() {

                    @Override

                    public void onFinalImageSet(String id, ImageInfo imageInfo, Animatable animatable) {

                        // 所有图片都加载成功时触发的方法

                        int width = imageInfo.getWidth();

                        int height = imageInfo.getHeight();
```

BAT交 892246

```
                    QualityInfo qualityInfo = imageInfo.getQualityInfo
();

                    int quality = qualityInfo.getQuality();

                    boolean isOfFullQuality = qualityInfo.isOfFullQual
ity();

                    boolean isOfGoodEnoughQuality = qualityInfo.isOfGo
odEnoughQuality();

                }



                @Override

                public void onIntermediateImageSet(String id, ImageInf
o imageInfo) {

                    // 加载渐进式图片时回调的方法

                }



                @Override

                public void onFailure(String id, Throwable throwable)
{

                    // 加载图片失败时回调的方法

                }

            })

            .setUri("http://image5.tuku.cn/pic/wallpaper/fengjing/men
ghuandaziranmeijingbizhi/009.jpg")

            .build());
```

3.4、渐进式图片

渐进式图片是一种支持图片从模糊到清晰的加载模式。代码如下：

```
  sdv.setController(Fresco.newDraweeControllerBuilder()
                        .setImageRequest(
                                ImageRequestBuilder.newBuilderWithSource(
                                        Uri.parse("http://image5.tuku.cn/pic/wallpaper/fengjing/menghuandaziranmeijingbizhi/009.jpg"))
                                        .setProgressiveRenderingEnabled(true)
                                        .build())
                        .setOldController(sdv.getController())
                        .build());
```

以上就是对 Fresco 的基本使用方法的介绍

4.1.4 Fresco

一、 Picasso 分析

<meta charset="utf-8">

Picasso 是 Square 公司开源的一个 Android 平台上的图片加载框架

BAT交流群：892872246

核心类：

Picasso：负责图片下载、变换、缓存的管理器，当它收到一个图片下载请求的时候，它会创建 Request 并提交给 Dispatcher

Dispatcher: Dispatcher 会寻找对应的处理器 RequestHandler，并将请求与该处理器一起提交给线程池执行

BAT交流群：892872246

图片加载步骤：

创建 -> 入队 -> 执行 -> 解码 -> 变化 -> 批处理 -> 完成 -> 分发 -> 显示

使用

```
Picasso.get().load(url).into(first_img)

//加载一张图片 旋转 45°

Picasso.get()

        .load(URL)

        .rotate(45f)

        .placeholder(R.drawable.placeholder_disk)

        .into(imgOne);

//加载一张图片并设置一个回调接口
```

```
Picasso.get()
        .load(URL)
        .placeholder(R.drawable.placeholder_disk)
        .into(imgOne, new Callback() {

            @Override
            public void onSuccess() {

            }

            @Override
            public void onError(Exception e) {

            }
        });
//预加载一张图片
Picasso.get().load(URL).fetch();
//同步加载一张图片,注意只能在子线程中调用并且Bitmap不会被缓存到内存里.new Thr
ead() {
    @Override
    public void run() {
        try {
            final Bitmap bitmap = Picasso.get().load(URL).get();
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }}.start();
//加载一张图片旋转并且添加一个 Transformation,可以对图片进行各种变化处理,例如圆形头像.
Picasso.get().load("url").rotate(10).transform(new Transformation() {
    @Override
    public Bitmap transform(Bitmap source) {
        //处理 Bitmap
        return null;
    }

    @Override
    public String key() {
        return null;
    }}).into(imgTwo);
//加载一张图片并按照指定尺寸以 centerInside()的形式缩放.并设置加载的优先级为高.注意 centerInside()或 centerCrop()//只能同时使用一种,而且必须指定 resize()或者 resizeDimen();
Picasso.get().load(URL)
        .resize(400, 400)
        .centerInside()
        .priority(Picasso.Priority.HIGH)
```

```
        .into(imgTwo);

//加载一张图片并设置 tag,可以通过 tag 来暂定或者继续加载,可以用于当 ListView 滚动是暂定加载.停止滚动恢复加载.

Picasso.get().load(URL).tag(getContext()).into(imgTwo);

Picasso.get().pauseTag(getContext());

Picasso.get().resumeTag(getContext());
```

二、Fresco 的使用

当下有很多图片加载框架，常见的有 Glide、Fresco、Picasso 等。Glide 因为其体积小、缓存机制强大等优点，受到了广大程序员的青睐；Fresco 虽然体积比较大，缓存机制也没有 Glide 强大，但它胜在拥有一些炫酷的效果（进度条、淡入效果）等，也有很多人在使用。今天我们就来介绍一下 Fresco 的使用。

1、配置环境

使用 Fresco 需要先在 build.gradle 中导入依赖：

```
// Fresco 所需依赖

   compile 'com.facebook.fresco:fresco:0.12.0'

   // 在 API < 14 上的机器支持 WebP 时，需要添加

   compile 'com.facebook.fresco:animated-base-support:0.12.0'

   // 支持 GIF 动图，需要添加

   compile 'com.facebook.fresco:animated-gif:0.12.0'

   // 支持 WebP （静态图+动图），需要添加

   compile 'com.facebook.fresco:animated-webp:0.12.0'

   compile 'com.facebook.fresco:webpsupport:0.12.0'
```

如果需要从网络中下载图片，则需要在 AndroidManifest.xml 文件中配置网络权限：

BAT交流 892246

```
<uses-permission android:name="android.permission.INTERNET" />
```

最后，在项目的 Application 中初始化 Fresco：

```
public class MyApplication extends Application {



    @Override

    public void onCreate() {

        super.onCreate();

        // 初始化 Fresco

        Fresco.initialize(this);

    }}
```

别忘了在 AndroidManifest.xml 文件中为 Application 添加 name 属性：

```
<application

        android:name=".MyApplication"

        ......

    </application>
```

2、SimpleDraweeView

2.1、XML 属性

Fresco 为我们提供了一个 SimpleDraweeView 控件，我们可以直接在这个控件中加载图片。Fresco 在这个控件中集成了很多属性，简单实用。以下是 SimpleDraweeView 中的常用属性：

```
fresco:actualImageScaleType：实际加载的图片的伸缩样式

        fresco:backgroundImage：底层图片资源
```

BAT交流 892872246

```
        fresco:fadeDuration：进度条和占位符图片逐渐消失、加载的图片逐渐显示的时间间隔

        fresco:failureImage：加载失败时显示的图片资源

        fresco:failureImageScaleType：加载失败时加载的图片的伸缩样式

        fresco:overlayImage：在显示的图片表层覆盖一张图片的图片资源

        fresco:placeholderImage：占位符图片资源

        fresco:placeholderImageScaleType：占位符图片的伸缩样式

        fresco:progressBarAutoRotateInterval：进度条图片转动周期

        fresco:progressBarImage：进度条图片资源

        fresco:progressBarImageScaleType：进度条图片的伸缩样式

        fresco:retryImage：提示重新加载的图片资源

        fresco:retryImageScaleType：提示重新加载的图片的伸缩样式

        fresco:roundAsCircle：将图片剪切成圆形

        fresco:viewAspectRatio：图片宽高比
```

我们的 XML 布局文件中的代码如下：

```
<?xml version="1.0" encoding="utf-8"?><RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

    xmlns:fresco="http://schemas.android.com/apk/res-auto"

    android:id="@+id/activity_main"

    android:layout_width="match_parent"

    android:layout_height="match_parent">



    <com.facebook.drawee.view.SimpleDraweeView
```

```
        android:id="@+id/id_main_sdv_sdv"

        android:layout_width="match_parent"

        android:layout_height="wrap_content"

        fresco:actualImageScaleType="focusCrop"

        fresco:fadeDuration="3000"

        fresco:failureImage="@mipmap/ic_launcher"

        fresco:failureImageScaleType="centerInside"

        fresco:placeholderImage="@mipmap/ic_launcher"

        fresco:placeholderImageScaleType="fitCenter"

        fresco:progressBarAutoRotateInterval="1000"

        fresco:progressBarImage="@mipmap/ic_launcher"

        fresco:progressBarImageScaleType="centerInside"

        fresco:retryImage="@mipmap/ic_launcher"

        fresco:retryImageScaleType="centerCrop"

        fresco:roundAsCircle="false"

        fresco:viewAspectRatio="1.6" />


    <Button

        android:id="@+id/id_main_btn_load"

        android:layout_width="match_parent"

        android:layout_height="wrap_content"

        android:layout_alignParentBottom="true"

        android:layout_margin="20.0dip"
```

```
        android:text="Start Load Image" />

</RelativeLayout>
```

2.2、注意事项

（1）SimpleDraweeView 控件的宽高不能设置为 wrap_content，只能是 match_parent、具体值或使用 viewAspectRatio 属性设置宽高比。

（2）如果在一个页面中加载多张图片，不要将 SimpleDraweeView 直接放在 ScollView 中，建议使用 RecyclerView、ListView 或 GridView，因为 ScrollView 中的 SimpleDraweeView 会持有内存直到 Activity 或 Fragment 销毁。

（3）使用 SimpleDraweeView 时，不要使用 imageView 中有、View 中没有的属性。

3、框架使用

3.1、简单使用

像 TextView、Button 等其他控件一样后去到 SimpleDraweeView，然后调用如下代码即可加载网络图片：

```
SimpleDraweeView sdv = (SimpleDraweeView) findViewById(R.id.id_main_sdv_sdv);
sdv.setImageURI("http://image5.tuku.cn/pic/wallpaper/fengjing/menghuandaziranmeijingbizhi/009.
jpg");
```

3.2、JAVA 代码设置属性

我们可以通过 GenericDraweeHierarchy，在 JAVA 代码中动态的设置 SimpleDraweeView 控件的属性。需要注意的是，如果在 JAVA 代码中设置了属性，那么 XML 文件中设置的属性就都无效了。一个实例代码如下：

```
 // 代码设置 SimpleDraweeView 的属性（会覆盖 XML 设置的所有属性，即在 XML 中有在
这里没有的属性都会失效）

        // 注意：一个 GenericDraweeHierarchy 是不能被多个 SimpleDraweeView 共
用的

        GenericDraweeHierarchy hierarchy = new GenericDraweeHierarchyBuil
der(getResources())

                .setFadeDuration(3000)

                .setPlaceholderImage(R.mipmap.ic_launcher)
```

BAT交流 892246

```
                .setPlaceholderImageScaleType(ScalingUtils.ScaleType.FIT_XY)

                .setProgressBarImage(new ProgressBarDrawable()) // 显示进度条（Fresco 自带的进度条）

                .build();

        // 设置图片圆角

        RoundingParams roundingParams = new RoundingParams();

        roundingParams.setRoundAsCircle(false); // 不将图片剪切成圆形

        roundingParams.setCornersRadius(200);

        hierarchy.setRoundingParams(roundingParams);

        // 为 SimpleDraweeView 设置属性

        sdv.setHierarchy(hierarchy);
```

注意：不能把同一个 GenericDraweeHierarchy 对象设置给多个 SimpleDraweeView！

3.3、设置下载事件

```
        sdv.setController(Fresco.newDraweeControllerBuilder()

                .setControllerListener(new BaseControllerListener<ImageInfo>() {

                    @Override

                    public void onFinalImageSet(String id, ImageInfo imageInfo, Animatable animatable) {

                        // 所有图片都加载成功时触发的方法

                        int width = imageInfo.getWidth();

                        int height = imageInfo.getHeight();
```

BAT交 892246

```
                    QualityInfo qualityInfo = imageInfo.getQualityInfo
();

                    int quality = qualityInfo.getQuality();

                    boolean isOfFullQuality = qualityInfo.isOfFullQual
ity();

                    boolean isOfGoodEnoughQuality = qualityInfo.isOfGo
odEnoughQuality();

                }



                @Override

                public void onIntermediateImageSet(String id, ImageInf
o imageInfo) {

                    // 加载渐进式图片时回调的方法

                }



                @Override

                public void onFailure(String id, Throwable throwable)
{

                    // 加载图片失败时回调的方法

                }

            })

            .setUri("http://image5.tuku.cn/pic/wallpaper/fengjing/men
ghuandaziranmeijingbizhi/009.jpg")

            .build());
```

BAT交流群 892246

3.4、渐进式图片

渐进式图片是一种支持图片从模糊到清晰的加载模式。代码如下：

```
sdv.setController(Fresco.newDraweeControllerBuilder()
                        .setImageRequest(
                                ImageRequestBuilder.newBuilderWithSource(
                                        Uri.parse("http://image5.tuku.cn/pic/wallpaper/fengjing/menghuandaziranmeijingbizhi/009.jpg"))
                                        .setProgressiveRenderingEnabled(true)
                                        .build())
                        .setOldController(sdv.getController())
                        .build());
```

4.2 Glide 原理分析

4.2.1 Glide 的基本用法

现在 Android 上的图片加载框架非常成熟，从最早的老牌图片加载框架 UniversalImageLoader，到后来 Google 推出的 Volley，再到后来的新兴军 Glide 和 Picasso，当然还有 Facebook 的 Fresco。每一个都非常稳定，功能也都十分强大。但是它们的使用场景基本都是重合的，也就是说我们基本只需要选择其中一个来进行学习和使用就足够了，每一个框架都尝试去掌握的话则有些浪费时间。

在这几个框架当中，我对 Volley 和 Glide 研究得比较深入，对 UniversalImageLoader、Picasso 和 Fresco 都只是有一些基本的了解。从易用性上来讲，Glide 和 Picasso 应该都是完胜其他框架的，这两个框架都实在是太简单好用了，大多数情况下加载图片都是一行代码就能解决的，而 UniversalImageLoader 和 Fresco 则在这方面略逊一些。

那么再拿 Glide 和 Picasso 对比呢，首先这两个框架的用法非常相似，但其实它们各有特色。Picasso 比 Glide 更加简洁和轻量，Glide 比 Picasso 功能更为丰富。之

BAT交流群：892872246

前已经有人对这两个框架进行过全方面的对比，大家如果想了解更多的话可以去参考一下 这篇文章 。

总之，没有最好的框架，只有最适合自己的框架。经过多方面对比之后，我还是决定选择了 Glide 来进行研究，并且这也是 Google 官方推荐的图片加载框架。

说实话，关于 Glide 的文章我已经筹备了好久，去年这个时候本来就打算要写了，但是一直都没有动笔。因为去年我的大部分时间都放在了写《第二行代码》上面，只能用碎片时间来写写博客，但是 Glide 的难度远超出了我用碎片时间所能掌握的难度。当然，这里我说的是对它的源码进行解析的难度，不是使用上的难度，Glide 的用法是很简单的。所以，我觉得去年我写不好 Glide 这个题材的文章，也就一直拖到了今年。

而现在，我花费了大量的精力去研究 Glide 的源码和各种用法，相信现在已经可以将它非常好地掌握了，因此我准备将我掌握的这些知识整理成一个新的系列，帮忙大家更好地学习 Glide。这个 Glide 系列大概会有 8 篇左右文章，预计花半年时间写完，将会包括 Glide 的基本用法、源码解析、高级用法、功能扩展等内容，可能会是目前互联网上最详尽的 Glide 教程。

那么本篇文章是这个系列的第一篇文章，我们先来了解一下 Glide 的基本用法吧。

开始

Glide 是一款由 Bump Technologies 开发的图片加载框架，使得我们可以在 Android 平台上以极度简单的方式加载和展示图片。

目前，Glide 最新的稳定版本是 3.7.0，虽然 4.0 已经推出 RC 版了，但是暂时问题还比较多。因此，我们这个系列的博客都会使用 Glide 3.7.0 版本来进行讲解，这个版本的 Glide 相当成熟和稳定。

要想使用 Glide，首先需要将这个库引入到我们的项目当中。新建一个 GlideTest 项目，然后在 app/build.gradle 文件当中添加如下依赖：

```
dependencies {

    compile 'com.github.bumptech.glide:glide:3.7.0'

}
```

如果你还在使用 Eclipse，可以点击 这里 下载 Glide 的 jar 包。

```
<uses-permission android:name="android.permission.INTERNET" />
```

BA… 892246

另外，Glide 中需要用到网络功能，因此你还得在 AndroidManifest.xml 中声明一下网络权限才行：

就是这么简单，然后我们就可以自由地使用 Glide 中的任意功能了。

加载图片

现在我们就来尝试一下如何使用 Glide 来加载图片吧。比如这是必应上一张首页美图的地址：

http://cn.bing.com/az/hprichbg/rb/Dongdaemun_ZH-CN10736487148_1920x1080.jpg

然后我们想要在程序当中去加载这张图片。

那么首先打开项目的布局文件，在布局当中加入一个 Button 和一个 ImageView，如下所示：

```
<?xml version="1.0" encoding="utf-8"?><LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:orientation="vertical">



    <Button

        android:layout_width="wrap_content"

        android:layout_height="wrap_content"

        android:text="Load Image"

        android:onClick="loadImage"

        />
```

```
    <ImageView

        android:id="@+id/image_view"

        android:layout_width="match_parent"

        android:layout_height="match_parent" />

</LinearLayout>
```

为了让用户点击 Button 的时候能够将刚才的图片显示在 ImageView 上，我们需要修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {



    ImageView imageView;



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        imageView = (ImageView) findViewById(R.id.image_view);

    }



    public void loadImage(View view) {

        String url = "http://cn.bing.com/az/hprichbg/rb/Dongdaemun_ZH-CN
10736487148_1920x1080.jpg";

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

    }
```

```
}
```

没错，就是这么简单。现在我们来运行一下程序，效果如下图所示：

可以看到，一张网络上的图片已经被成功下载，并且展示到 ImageView 上了。

而我们到底做了什么？实际上核心的代码就只有这一行而已：

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

千万不要小看这一行代码，实际上仅仅就这一行代码，你已经可以做非常非常多的事情了，包括加载网络上的图片、加载手机本地的图片、加载应用资源中的图片等等。

下面我们就来详细解析一下这行代码。

首先，调用 Glide.with()方法用于创建一个加载图片的实例。with()方法可以接收 Context、Activity 或者 Fragment 类型的参数。也就是说我们选择的范围非常广，

技术交流群：892872246

不管是在 Activity 还是 Fragment 中调用 with()方法，都可以直接传 this。那如果调用的地方既不在 Activity 中也不在 Fragment 中呢？也没关系，我们可以获取当前应用程序的 ApplicationContext，传入到 with()方法当中。注意 with()方法中传入的实例会决定 Glide 加载图片的生命周期，如果传入的是 Activity 或者 Fragment 的实例，那么当这个 Activity 或 Fragment 被销毁的时候，图片加载也会停止。如果传入的是 ApplicationContext，那么只有当应用程序被杀掉的时候，图片加载才会停止。

接下来看一下 load()方法，这个方法用于指定待加载的图片资源。Glide 支持加载各种各样的图片资源，包括网络图片、本地图片、应用资源、二进制流、Uri 对象等等。因此 load()方法也有很多个方法重载，除了我们刚才使用的加载一个字符串网址之外，你还可以这样使用 load()方法：

```
// 加载本地图片 File file = new File(getExternalCacheDir() + "/image.jpg
");

Glide.with(this).load(file).into(imageView);

// 加载应用资源 int resource = R.drawable.image;

Glide.with(this).load(resource).into(imageView);

// 加载二进制流 byte[] image = getImageBytes();

Glide.with(this).load(image).into(imageView);

// 加载 Uri 对象 Uri imageUri = getImageUri();

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

最后看一下 into()方法，这个方法就很简单了，我们希望让图片显示在哪个 ImageView 上，把这个 ImageView 的实例传进去就可以了。当然，into()方法不仅仅是只能接收 ImageView 类型的参数，还支持很多更丰富的用法，不过那个属于高级技巧，我们会在后面的文章当中学习。

那么回顾一下 Glide 最基本的使用方式，其实就是关键的三步走：先 with()，再 load()，最后 into()。熟记这三步，你就已经入门 Glide 了。

占位图

现在我们来学一些 Glide 的扩展内容。其实刚才所学的三步走就是 Glide 最核心的东西，而我们后面所要学习的所有东西都是在这个三步走的基础上不断进行扩展而已。

BAT交流群：892872246

观察刚才加载网络图片的效果，你会发现，点击了 Load Image 按钮之后，要稍微等一会图片才会显示出来。这其实很容易理解，因为从网络上下载图片本来就是需要时间的。那么我们有没有办法再优化一下用户体验呢？当然可以，Glide 提供了各种各样非常丰富的 API 支持，其中就包括了占位图功能。

顾名思义，占位图就是指在图片的加载过程中，我们先显示一张临时的图片，等图片加载出来了再替换成要加载的图片。

下面我们就来学习一下 Glide 占位图功能的使用方法，首先我事先准备好了一张 loading.jpg 图片，用来作为占位图显示。然后修改 Glide 加载部分的代码，如下所示：

```
Glide.with(this)

    .load(url)

    .placeholder(R.drawable.loading)

    .into(imageView);
```

没错，就是这么简单。我们只是在刚才的三步走之间插入了一个 placeholder()方法，然后将占位图片的资源 id 传入到这个方法中即可。另外，这个占位图的用法其实也演示了 Glide 当中绝大多数 API 的用法，其实就是在 load()和 into()方法之间串接任意想添加的功能就可以了。

不过如果你现在重新运行一下代码并点击 Load Image，很可能是根本看不到占位图效果的。因为 Glide 有非常强大的缓存机制，我们刚才加载那张必应美图的时候 Glide 自动就已经将它缓存下来了，下次加载的时候将会直接从缓存中读取，不会再去网络下载了，因而加载的速度非常快，所以占位图可能根本来不及显示。

因此这里我们还需要稍微做一点修改，来让占位图能有机会显示出来，修改代码如下所示：

```
Glide.with(this)

    .load(url)

    .placeholder(R.drawable.loading)

    .diskCacheStrategy(DiskCacheStrategy.NONE)

    .into(imageView);
```

可以看到，这里串接了一个 diskCacheStrategy()方法，并传入 DiskCacheStrategy.NONE 参数，这样就可以禁用掉 Glide 的缓存功能。

关于 Glide 缓存方面的内容我们将会在后面的文章进行详细的讲解，这里只是为了测试占位图功能而加的一个额外配置，暂时你只需要知道禁用缓存必须这么写就可以了。

现在重新运行一下代码，效果如下图所示：

可以看到，这里串接了一个 diskCacheStrategy()方法，并传入 DiskCacheStrategy.NONE 参数，这样就可以禁用掉 Glide 的缓存功能。

关于 Glide 缓存方面的内容我们将会在后面的文章进行详细的讲解，这里只是为了测试占位图功能而加的一个额外配置，暂时你只需要知道禁用缓存必须这么写就可以了。

现在重新运行一下代码，效果如下图所示：

可以看到，当点击 Load Image 按钮之后会立即显示一张占位图，然后等真正的图片加载完成之后会将占位图替换掉。

BAT交流群：892872246

当然，这只是占位图的一种，除了这种加载占位图之外，还有一种异常占位图。异常占位图就是指，如果因为某些异常情况导致图片加载失败，比如说手机网络信号不好，这个时候就显示这张异常占位图。

异常占位图的用法相信你已经可以猜到了，首先准备一张 error.jpg 图片，然后修改 Glide 加载部分的代码，如下所示：

```
Glide.with(this)
    .load(url)
    .placeholder(R.drawable.loading)
    .error(R.drawable.error)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .into(imageView);
```

很简单，这里又串接了一个 error()方法就可以指定异常占位图了。

现在你可以将图片的 url 地址修改成一个不存在的图片地址，或者干脆直接将手机的网络给关了，然后重新运行程序，效果如下图所示：

BAT交流群：892872246

这样我们就把 Glide 提供的占位图功能都掌握了。

指定图片格式

我们还需要再了解一下 Glide 另外一个强大的功能，那就是 Glide 是支持加载 GIF 图片的。这一点确实非常牛逼，因为相比之下 Jake Warton 曾经明确表示过，Picasso 是不会支持加载 GIF 图片的。

而使用 Glide 加载 GIF 图并不需要编写什么额外的代码，Glide 内部会自动判断图片格式。比如这是一张 GIF 图片的 URL 地址：

```
http://p1.pstatp.com/large/1662000198500628 39d3
```

我们只需要将刚才那段加载图片代码中的 URL 地址替换成上面的地址就可以了，现在重新运行一下代码，效果如下图所示：

BAT交流群：892872246

也就是说，不管我们传入的是一张普通图片，还是一张 GIF 图片，Glide 都会自动进行判断，并且可以正确地把它解析并展示出来。

但是如果我想指定图片的格式该怎么办呢？就比如说，我希望加载的这张图必须是一张静态图片，我不需要 Glide 自动帮我判断它到底是静图还是 GIF 图。

想实现这个功能仍然非常简单，我们只需要再串接一个新的方法就可以了，如下所示：

```
Glide.with(this)
    .load(url)
    .asBitmap()
    .placeholder(R.drawable.loading)
    .error(R.drawable.error)
```

```
    .diskCacheStrategy(DiskCacheStrategy.NONE)

    .into(imageView);
```

可以看到，这里在 load()方法的后面加入了一个 asBitmap()方法，这个方法的意思就是说这里只允许加载静态图片，不需要 Glide 去帮我们自动进行图片格式的判断了。

现在重新运行一下程序，效果如下图所示：

由于调用了 asBitmap()方法，现在 GIF 图就无法正常播放了，而是会在界面上显示第一帧的图片。

那么类似地，既然我们能强制指定加载静态图片，就也能强制指定加载动态图片。比如说我们想要实现必须加载动态图片的功能，就可以这样写：

```
Glide.with(this)
```

BA交流群：892872246

```
    .load(url)

    .asGif()

    .placeholder(R.drawable.loading)

    .error(R.drawable.error)

    .diskCacheStrategy(DiskCacheStrategy.NONE)

    .into(imageView);
```

这里调用了 asGif()方法替代了 asBitmap()方法，很好理解，相信不用我多做什么解释了。

那么既然指定了只允许加载动态图片，如果我们传入了一张静态图片的 URL 地址又会怎么样呢？试一下就知道了，将图片的 URL 地址改成刚才的必应美图，然后重新运行代码，效果如下图所示。

BAT交流群：892872246

没错，如果指定了只能加载动态图片，而传入的图片却是一张静图的话，那么结果自然就只有加载失败喽。

指定图片大小

实际上，使用 Glide 在绝大多数情况下我们都是不需要指定图片大小的。

在学习本节内容之前，你可能还需要先了解一个概念，就是我们平时在加载图片的时候很容易会造成内存浪费。什么叫内存浪费呢？比如说一张图片的尺寸是 1000*1000 像素，但是我们界面上的 ImageView 可能只有 200*200 像素，这个时候如果你不对图片进行任何压缩就直接读取到内存中，这就属于内存浪费了，因为程序中根本就用不到这么高像素的图片。

关于图片压缩这方面，我之前也翻译过 Android 官方的一篇文章，感兴趣的朋友可以去阅读一下 Android 高效加载大图、多图解决方案，有效避免程序 OOM 。

而使用 Glide，我们就完全不用担心图片内存浪费，甚至是内存溢出的问题。因为 Glide 从来都不会直接将图片的完整尺寸全部加载到内存中，而是用多少加载多少。Glide 会自动判断 ImageView 的大小，然后只将这么大的图片像素加载到内存当中，帮助我们节省内存开支。

当然，Glide 也并没有使用什么神奇的魔法，它内部的实现原理其实就是上面那篇文章当中介绍的技术，因此掌握了最基本的实现原理，你也可以自己实现一套这样的图片压缩机制。

也正是因为 Glide 是如此的智能，所以刚才在开始的时候我就说了，在绝大多数情况下我们都是不需要指定图片大小的，因为 Glide 会自动根据 ImageView 的大小来决定图片的大小。

不过，如果你真的有这样的需求，必须给图片指定一个固定的大小，Glide 仍然是支持这个功能的。修改 Glide 加载部分的代码，如下所示：

```
Glide.with(this)
    .load(url)
    .placeholder(R.drawable.loading)
    .error(R.drawable.error)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .override(100, 100)
```

BA… 2246

```
    .into(imageView);
```

仍然非常简单，这里使用 override()方法指定了一个图片的尺寸，也就是说，Glide 现在只会将图片加载成 100*100 像素的尺寸，而不会管你的 ImageView 的大小是多少了。

4.2.2 从源码的角度理解 Glide 的执行流程上篇、下篇

在多数情况下，我们想要在界面上加载并展示一张图片只需要一行代码就能实现，如下所示：

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

虽说只有这简简单单的一行代码，但大家可能不知道的是，Glide 在背后帮我们默默执行了成吨的工作。这个形容词我想了很久，因为我觉得用非常多这个形容词不足以描述 Glide 背后的工作量，我查到的英文资料是用 tons of work 来进行形容的，因此我觉得这里使用成吨来形容更加贴切一些。

虽说我们在平时使用 Glide 的时候格外地简单和方便，但是知其然也要知其所以然。那么今天我们就来解析一下 Glide 的源码，看看它在这些简单用法的背后，到底执行了多么复杂的工作。

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。

BAT资料：892872246

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，但不能做任何的修改，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 {
```

892872246

```
    ...


    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？我们一起来看一看：

```
public class RequestManagerRetriever implements Handler.Callback {


    private static final RequestManagerRetriever INSTANCE = new RequestM
anagerRetriever();


    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 fragm
ent 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.getApp
licationContext(),

                            new ApplicationLifecycle(), new EmptyRequestMa
nagerTreeNode());

                }

            }

        }

        return applicationManager;

    }
```

```
    public RequestManager get(Context context) {

        if (context == null) {

            throw new IllegalArgumentException("You cannot start a load o
n a null Context");

        } else if (Util.isOnMainThread() && !(context instanceof Applicat
ion)) {

            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 o
n 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);
```

BAT交 892246

```
            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 f
or 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 o
n 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.getChildFragmentMan
ager();

            return fragmentGet(fragment.getActivity(), fm);

        }

    }



    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)

    RequestManagerFragment getRequestManagerFragment(final android.app.F
ragmentManager fm) {

        RequestManagerFragment current = (RequestManagerFragment) fm.find
FragmentByTag(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).commitAl
lowingStateLoss();

                handler.obtainMessage(ID_REMOVE_FRAGMENT_MANAGER, fm).sen
dToTarget();

            }

        }

        return current;

    }
```

```
    @TargetApi(Build.VERSION_CODES.HONEYCOMB)

    RequestManager fragmentGet(Context context, android.app.FragmentMana
ger fm) {

        RequestManagerFragment current = getRequestManagerFragment(fm);

        RequestManager requestManager = current.getRequestManager();

        if (requestManager == null) {

            requestManager = new RequestManager(context, current.getLifec
ycle(), current.getRequestManagerTreeNode());

            current.setRequestManager(requestManager);

        }

        return requestManager;

    }



    SupportRequestManagerFragment getSupportRequestManagerFragment(final
 FragmentManager fm) {

        SupportRequestManagerFragment current = (SupportRequestManagerFr
agment) 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).commitAl
lowingStateLoss();

                handler.obtainMessage(ID_REMOVE_SUPPORT_FRAGMENT_MANAGER,
 fm).sendToTarget();
```

```
            }

        }

        return current;

    }



    RequestManager supportFragmentGet(Context context, FragmentManager f
m) {

        SupportRequestManagerFragment current = getSupportRequestManager
Fragment(fm);

        RequestManager requestManager = current.getRequestManager();

        if (requestManager == null) {

            requestManager = new RequestManager(context, current.getLifec
ycle(), 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

BAT交流群：892872246

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.b
umptech.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 usi
ng 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 signat
ure
```

```
     *     you create that identifies the data currently at the given Str
ing that will invalidate the cache if that data

     *     changes. Alternatively, using {@link DiskCacheStrategy#NONE} a
nd/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.buildStream
ModelLoader(modelClass, context);

        ModelLoader<T, ParcelFileDescriptor> fileDescriptorModelLoader =

                Glide.buildFileDescriptorModelLoader(modelClass, contex
t);

        if (modelClass != null && streamModelLoader == null && fileDescri
ptorModelLoader == null) {

            throw new IllegalArgumentException("Unknown type " + modelCla
ss + ". You must provide a Model of a type for"
```

BAT交流群892246

```
                    + " which there is a registered ModelLoader, if you ar
e 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, optionsApplie
r));

    }



    ...

}
```

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，

BAT交流群：892872246

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：
你好啊！请问是通过简书过来学习交流和来免费领取阿里 P7 级别架构技术视频，BAT 面试资料集锦，项目专题资料、源码的吗？

```
public class DrawableTypeRequest<ModelType> extends DrawableRequestBuild
er<ModelType> implements DownloadOptions {

    private final ModelLoader<ModelType, InputStream> streamModelLoader;

    private final ModelLoader<ModelType, ParcelFileDescriptor> fileDescr
iptorModelLoader;

    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> fileDescriptorModelLoade
r, Class<Z> resourceClass,

            Class<R> transcodedClass,

            ResourceTranscoder<Z, R> transcoder) {

        if (streamModelLoader == null && fileDescriptorModelLoader == nul
l) {

            return null;

        }



        if (transcoder == null) {

            transcoder = glide.buildTranscoder(resourceClass, transcodedC
lass);
```

```
        }

        DataLoadProvider<ImageVideoWrapper, Z> dataLoadProvider = glide.b
uildDataProvider(ImageVideoWrapper.class,

                resourceClass);

        ImageVideoModelLoader<A> modelLoader = new ImageVideoModelLoader<
A>(streamModelLoader,

                fileDescriptorModelLoader);

        return new FixedLoadProvider<A, ImageVideoWrapper, Z, R>(modelLoa
der, transcoder, dataLoadProvider);

    }



    DrawableTypeRequest(Class<ModelType> modelClass, ModelLoader<ModelTy
pe, InputStream> streamModelLoader,

            ModelLoader<ModelType, ParcelFileDescriptor> fileDescriptorMo
delLoader, Context context, Glide glide,

            RequestTracker requestTracker, Lifecycle lifecycle, RequestMa
nager.OptionsApplier optionsApplier) {

        super(context, modelClass,

                buildProvider(glide, streamModelLoader, fileDescriptorMod
elLoader, 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.graphic
s.Bitmap}
     */
    public BitmapTypeRequest<ModelType> asBitmap() {
        return optionsApplier.apply(new BitmapTypeRequest<ModelType>(thi
s, streamModelLoader,
                fileDescriptorModelLoader, optionsApplier));
    }


    /**
     * Attempts to always load the resource as a {@link com.bumptech.glid
e.load.resource.gif.GifDrawable}.
     * <p>
     *     If the underlying data is not a GIF, this will fail. As a resul
t, this should only be used if the model
     *     represents an animated GIF and the caller wants to interact wi
th the GIfDrawable directly. Normally using
     *     just an {@link DrawableTypeRequest} is sufficient because it w
ill determine whether or
     *     not the given data represents an animated GIF and return the a
ppropriate animated or not animated
     *     {@link android.graphics.drawable.Drawable} automatically.
```

```
     * </p>

     *

     * @return A new request builder for loading a {@link com.bumptech.gl
ide.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, GifB
itmapWrapper, GlideDrawable>

        implements BitmapOptions, DrawableOptions {



    DrawableRequestBuilder(Context context, Class<ModelType> modelClass,
```

```
            LoadProvider<ModelType, ImageVideoWrapper, GifBitmapWrapper,
GlideDrawable> loadProvider, Glide glide,

            RequestTracker requestTracker, Lifecycle lifecycle) {

        super(context, modelClass, loadProvider, GlideDrawable.class, gli
de, requestTracker, lifecycle);

        // Default to animating.

        crossFade();

    }



    public DrawableRequestBuilder<ModelType> thumbnail(

            DrawableRequestBuilder<?> thumbnailRequest) {

        super.thumbnail(thumbnailRequest);

        return this;

    }



    @Override

    public DrawableRequestBuilder<ModelType> thumbnail(

            GenericRequestBuilder<?, ?, ?, GlideDrawable> thumbnailReques
t) {

        super.thumbnail(thumbnailRequest);

        return this;

    }



    @Override
```

```
    public DrawableRequestBuilder<ModelType> thumbnail(float sizeMultipl
ier) {

        super.thumbnail(sizeMultiplier);

        return this;

    }



    @Override

    public DrawableRequestBuilder<ModelType> sizeMultiplier(float sizeMu
ltiplier) {

        super.sizeMultiplier(sizeMultiplier);

        return this;

    }



    @Override

    public DrawableRequestBuilder<ModelType> decoder(ResourceDecoder<Ima
geVideoWrapper, GifBitmapWrapper> decoder) {

        super.decoder(decoder);

        return this;

    }



    @Override

    public DrawableRequestBuilder<ModelType> cacheDecoder(ResourceDecode
r<File, GifBitmapWrapper> cacheDecoder) {

        super.cacheDecoder(cacheDecoder);

        return this;
```

```
    }


    @Override

    public DrawableRequestBuilder<ModelType> encoder(ResourceEncoder<Gif
BitmapWrapper> encoder) {

        super.encoder(encoder);

        return this;

    }


    @Override

    public DrawableRequestBuilder<ModelType> priority(Priority priority)
 {

        super.priority(priority);

        return this;

    }


    public DrawableRequestBuilder<ModelType> transform(BitmapTransformat
ion... transformations) {

        return bitmapTransform(transformations);

    }


    public DrawableRequestBuilder<ModelType> centerCrop() {

        return transform(glide.getDrawableCenterCrop());

    }
```

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

    public DrawableRequestBuilder<ModelType> bitmapTransform(Transformat
ion<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<Gi
fBitmapWrapper>... transformation) {
        super.transform(transformation);
        return this;
    }

    @Override
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
    public DrawableRequestBuilder<ModelType> transcoder(

            ResourceTranscoder<GifBitmapWrapper, GlideDrawable> transcode
r) {

        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>(durati
on));

        return this;

    }



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

        super.animate(new DrawableCrossFadeFactory<GlideDrawable>(contex
t, animationId,

                duration));

        return this;

    }
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

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

    @Override
    public DrawableRequestBuilder<ModelType> animate(ViewPropertyAnimati
on.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 drawab
le) {

        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> requestList
ener) {

        super.listener(requestListener);

        return this;

    }

    @Override

    public DrawableRequestBuilder<ModelType> diskCacheStrategy(DiskCache
Strategy strategy) {

        super.diskCacheStrategy(strategy);

        return this;

    }
```

```
    @Override

    public DrawableRequestBuilder<ModelType> skipMemoryCache(boolean ski
p) {

        super.skipMemoryCache(skip);

        return this;

    }



    @Override

    public DrawableRequestBuilder<ModelType> override(int width, int hei
ght) {

        super.override(width, height);

        return this;

    }



    @Override

    public DrawableRequestBuilder<ModelType> sourceEncoder(Encoder<Image
VideoWrapper> sourceEncoder) {

        super.sourceEncoder(sourceEncoder);

        return this;

    }



    @Override

    public DrawableRequestBuilder<ModelType> dontTransform() {
```

BA 2246

```
        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) {
```

BAT交流群：892872246

```
    Util.assertMainThread();

    if (view == null) {

        throw new IllegalArgumentException("You must pass in a non null V
iew");

    }

    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> transco
dedClass) {

    return imageViewTargetFactory.buildTarget(imageView, transcodedClas
s);}
```

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: " + claz
z

                    + ", 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 T
arget");

    }

    if (!isModelSet) {

        throw new IllegalArgumentException("You must first set a model (t
ry #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, Thum
bnailRequestCoordinator 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 req
uest 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.overri
deWidth,
                        thumbnailRequestBuilder.overrideHeight)) {
          thumbnailRequestBuilder.override(overrideWidth, overrideHeigh
t);
        }

        ThumbnailRequestCoordinator coordinator = new ThumbnailRequestCoo
rdinator(parentCoordinator);
        Request fullRequest = obtainRequest(target, sizeMultiplier, prior
ity, coordinator);
        // Guard against infinite recursion.
        isThumbnailBuilt = true;
        // Recursively generate thumbnail requests.
        Request thumbRequest = thumbnailRequestBuilder.buildRequestRecur
sive(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 ThumbnailRequestCoo
rdinator(parentCoordinator);

        Request fullRequest = obtainRequest(target, sizeMultiplier, prior
ity, coordinator);

        Request thumbnailRequest = obtainRequest(target, thumbSizeMultipl
ier, getThumbnailPriority(), coordinator);

        coordinator.setRequests(fullRequest, thumbnailRequest);

        return coordinator;

    } else {

        // Base case: no thumbnail.

        return obtainRequest(target, sizeMultiplier, priority, parentCoor
dinator);

    }}

private Request obtainRequest(Target<TranscodeType> target, float sizeMu
ltiplier, 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：

BAT交流群 892872246

```
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()方法来去执行这个 Request，那么我们跟进去瞧一瞧，如下所示：

BAT交流群：892872246

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

```
@Overridepublic void begin() {
```

IT交流群：892872246

```
    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(startTi
me));

    }}
```

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一个 target.onLoadStarted()方法，并传入了一个 loading 占位图，在也就说，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：

```
@Overridepublic void onSizeReady(int width, int height) {

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logV("Got onSizeReady in " + LogTime.getElapsedMillis(startTim
e));

    }

    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(mo
del, 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.getElapsedMi
llis(startTime));

    }

    loadedFromMemoryCache = true;

    loadStatus = engine.load(signature, width, height, dataFetcher, load
Provider, transformation, transcoder,

            priority, isMemoryCacheable, diskCacheStrategy, this);

    loadedFromMemoryCache = resource != null;

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logV("finished onSizeReady in " + LogTime.getElapsedMillis(startT
ime));

    }}
```

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

BAT交流群：892872246

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, ImageVid
eoWrapper> {

    private static final String TAG = "IVML";



    private final ModelLoader<A, InputStream> streamLoader;

    private final ModelLoader<A, ParcelFileDescriptor> fileDescriptorLoa
der;



    public ImageVideoModelLoader(ModelLoader<A, InputStream> streamLoade
r,

            ModelLoader<A, ParcelFileDescriptor> fileDescriptorLoader) {

        if (streamLoader == null && fileDescriptorLoader == null) {
```

BAT文 892246

```
            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, in
t 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.getResourceFetch
er(model, width, height);

        }



        if (streamFetcher != null || fileDescriptorFetcher != null) {

            return new ImageVideoFetcher(streamFetcher, fileDescriptorFet
cher);

        } else {
```

```
            return null;

        }

    }



    static class ImageVideoFetcher implements DataFetcher<ImageVideoWrap
per> {

        private final DataFetcher<InputStream> streamFetcher;

        private final DataFetcher<ParcelFileDescriptor> fileDescriptorFe
tcher;



        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

BAT交流群：892872246

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> transf
ormation, ResourceTranscoder<Z, R> transcoder,

            Priority priority, boolean isMemoryCacheable, DiskCacheStrate
gy 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, loadProv
ider.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, isMemoryC
acheable);

        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, ke
y);
```

```
            }

            return new LoadStatus(cb, current);

        }



        EngineJob engineJob = engineJobFactory.build(key, isMemoryCacheab
le);

        DecodeJob<T, Z, R> decodeJob = new DecodeJob<T, Z, R>(key, width,
 height, fetcher, loadProvider, transformation,

                transcoder, diskCacheProvider, diskCacheStrategy, priorit
y);

        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> {



    ...
```

BA 92872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
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;
}
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
    ...}
```

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：

```
@Overridepublic ImageVideoWrapper loadData(Priority priority) throws Exc
eption {

    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 Parce
lFileDescriptor", 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*/, n
ull /*lastUrl*/, glideUrl.getHeaders());
```

```
    }



    private InputStream loadDataWithRedirects(URL url, int redirects, UR
L 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 netwo
rk I/O and is generally broken.

            // See http://michaelscharf.blogspot.com/2006/11/javaneturleq
uals-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(), headerE
ntry.getValue());

        }

        urlConnection.setConnectTimeout(2500);

        urlConnection.setReadTimeout(2500);

        urlConnection.setUseCaches(false);

        urlConnection.setDoInput(true);



        // Connect explicitly to avoid errors in decoders if connection f
ails.

        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("Loca
tion");

            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 fr
om 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.getInp
utStream(), contentLength);

        } else {

            if (Log.isLoggable(TAG, Log.DEBUG)) {

                Log.d(TAG, "Got non empty content encoding: " + urlConnect
ion.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, hei
ght);

        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, i
nt 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);

        }
```

BAT文 /2246

```
        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.getStream() != 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.getStream(), 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 fro
m 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(bi
s, source.getFileDescriptor());

            result = decodeBitmapWrapper(forBitmapDecoder, width, heigh
t);

        }

        return result;

    }



    private GifBitmapWrapper decodeBitmapWrapper(ImageVideoWrapper toDec
ode, 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<ImageVid
eoWrapper, Bitmap> {

    private final ResourceDecoder<InputStream, Bitmap> streamDecoder;

    private final ResourceDecoder<ParcelFileDescriptor, Bitmap> fileDesc
riptorDecoder;



    public ImageVideoBitmapDecoder(ResourceDecoder<InputStream, Bitmap>
streamDecoder,

            ResourceDecoder<ParcelFileDescriptor, Bitmap> fileDescriptorD
ecoder) {

        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.getStream();

        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 F
ileDescriptor", e);

                }

            }

        }

        if (result == null) {

            ParcelFileDescriptor fileDescriptor = source.getFileDescripto
r();

            if (fileDescriptor != null) {

                result = fileDescriptorDecoder.decode(fileDescriptor, wid
th, height);

            }

        }

        return result;

    }



    ...}
```

代码并不复杂，在第 14 行先调用了 source.getStream()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 bitma
pPool, DecodeFormat decodeFormat) {

        this.downsampler = downsampler;

        this.bitmapPool = bitmapPool;

        this.decodeFormat = decodeFormat;

    }



    @Override

    public Resource<Bitmap> decode(InputStream source, int width, int he
ight) {

        Bitmap bitmap = downsampler.decode(source, bitmapPool, width, hei
ght, 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, i
nt 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 RecyclableBuff
eredInputStream(

                is, bytesForStream);

        // Use to retrieve exceptions thrown while reading.

        // TODO(#126): when the framework no longer returns partially dec
oded 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 MarkEnforcingIn
putStream(exceptionStream);

        try {

            exceptionStream.mark(MARK_POSITION);

            int orientation = 0;

            try {

                orientation = new ImageHeaderParser(exceptionStream).getO
rientation();

            } catch (IOException e) {

                if (Log.isLoggable(TAG, Log.WARN)) {

                    Log.w(TAG, "Cannot determine the image orientation fro
m 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, buff
eredStream, options);

        final int inWidth = inDimens[0];

        final int inHeight = inDimens[1];

        final int degreesToRotate = TransformationUtils.getExifOrient
ationDegrees(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 so
me cases when inBitmap is non null, may catch

        // and log a stack trace but still return a non null bitmap. T
o avoid displaying partially decoded bitmaps,

        // we catch exceptions reading from the stream in our Excepti
onCatchingInputStream and throw them here.

        final Exception streamException = exceptionStream.getExceptio
n();

        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(downsample
d)) {

                    downsampled.recycle();

                }

            }

            return rotated;

        } finally {

            byteArrayPool.releaseBytes(bytesForOptions);

            byteArrayPool.releaseBytes(bytesForStream);

            exceptionStream.release();

            releaseOptions(options);

        }

    }



    private Bitmap downsampleWithSize(MarkEnforcingInputStream is, Recyc
lableBufferedInputStream  bufferedStream,

            BitmapFactory.Options options, BitmapPool pool, int inWidth,
int inHeight, int sampleSize,

            DecodeFormat decodeFormat) {
```

BAT交 /2246

```
        // 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 <= B
uild.VERSION.SDK_INT) && shouldUsePool(is)) {

            int targetWidth = (int) Math.ceil(inWidth / (double) sampleSi
ze);

            int targetHeight = (int) Math.ceil(inHeight / (double) sample
Size);

            // 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 Inp
utStream.

     *

     * @param is The InputStream representing the image.

     * @param options The options to pass to

     *         {@link BitmapFactory#decodeStream(InputStream, android.gr
aphics.Rect,
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
     *             BitmapFactory.Options)}.

     * @return an array containing the dimensions of the image in the for
m {width, height}.

     */

    public int[] getDimensions(MarkEnforcingInputStream is, RecyclableBu
fferedInputStream 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, Recy
clableBufferedInputStream 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 hea
der in the buffer to get the image size so

             // that we don't fail to load images. The BufferedInputStrea
m 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.
```

892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
            is.mark(MARK_POSITION);

        } else {

            // Once we've read the image header, we no longer need to all
ow the buffer to expand in size. To avoid

            // unnecessary allocations reading image data, we fix the ma
rk 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, option
s);

        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就已经被加载出来了，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 Bit
map} 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 bitmap
Pool) {

        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：

```
public class GifBitmapWrapper {

    private final Resource<GifDrawable> gifResource;

    private final Resource<Bitmap> bitmapResource;



    public GifBitmapWrapper(Resource<Bitmap> bitmapResource, Resource<Gi
fDrawable> 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：

```
@Overridepublic Resource<GifBitmapWrapper> decode(ImageVideoWrapper sour
ce, 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) : nul
l;}
```

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：

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<GifBitmapWrapp
er> {

    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", startTim
e);

    }

    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。

BAT交流群：892872246

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，但是真正需要用到的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，它的构建函数中去构建了一个 FixedLoadProvider 对象，然后我们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 ResourceTrans
coder<GifBitmapWrapper, GlideDrawable> {

    private final ResourceTranscoder<Bitmap, GlideBitmapDrawable> bitmap
DrawableResourceTranscoder;



    public GifBitmapWrapperDrawableTranscoder(

            ResourceTranscoder<Bitmap, GlideBitmapDrawable> bitmapDrawabl
eResourceTranscoder) {

        this.bitmapDrawableResourceTranscoder = bitmapDrawableResourceTr
anscoder;

    }



    @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(bitmapRes
ource);

        } 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> toTr
anscode) {

        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。

BAT交流群：892872246

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);

    }}
```

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(Loope
r.getMainLooper(), new MainThreadCallback());



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



    ...



    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).sendToTarg
et();
    }

    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, isCacheab
le);

        hasResource = true;

        engineResource.acquire();

        listener.onEngineJobComplete(key, engineResource);

        for (ResourceCallback cb : cbs) {

            if (!isInIgnoredCallbacks(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).sendToTar
get();

    }



    private void handleExceptionOnMainThread() {

        if (isCancelled) {

            return;
```

```
        } else if (cbs.isEmpty()) {

            throw new IllegalStateException("Received an exception withou
t any callbacks to notify");

        }

        hasException = true;

        listener.onEngineJobComplete(key, null);

        for (ResourceCallback cb : cbs) {

            if (!isInIgnoredCallbacks(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> transf
ormation, ResourceTranscoder<Z, R> transcoder, Priority priority,

            boolean isMemoryCacheable, DiskCacheStrategy diskCacheStrateg
y, ResourceCallback cb) {



        ...



        EngineJob engineJob = engineJobFactory.build(key, isMemoryCacheab
le);

        DecodeJob<T, Z, R> decodeJob = new DecodeJob<T, Z, R>(key, width,
 height, fetcher, loadProvider, transformation,

                transcoder, diskCacheProvider, diskCacheStrategy, priorit
y);

        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.getElaps
edMillis(startTime));

        }

        loadedFromMemoryCache = true;

        loadStatus = engine.load(signature, width, height, dataFetcher, l
oadProvider, transformation,

                transcoder, priority, isMemoryCacheable, diskCacheStrateg
y, this);

        loadedFromMemoryCache = resource != null;

        if (Log.isLoggable(TAG, Log.VERBOSE)) {

            logV("finished onSizeReady in " + LogTime.getElapsedMillis(st
artTime));

        }

    }



    ...}
```

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.ge
tClass())) {

        releaseResource(resource);

        onException(new Exception("Expected to receive an object of " + t
ranscodeClass

                + " but instead got " + (received != null ? received.getCl
ass() : "") + "{" + 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 canSetResour
ce().

        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(resu
lt, model, target, loadedFromMemoryCache,

            isFirstResource)) {

        GlideAnimation<R> animation = animationFactory.build(loadedFromM
emoryCache, 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: " + lo
adedFromMemoryCache);

    }}
```

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。

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

```
public class GlideDrawableImageViewTarget extends ImageViewTarget<GlideD
rawable> {

    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
```

BAT交流群892246

```
    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> gl
ideAnimation) {
```

```
        if (glideAnimation == null || !glideAnimation.animate(resource, t
his)) {

            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。

总结

真是好长的一篇文章，这也可能是我目前所写过的最长的一篇文章了。如果你之前没有读过 Glide 的源码，真的很难相信，这短短一行代码：

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

背后竟然蕴藏着如此极其复杂的逻辑吧？

不过 Glide 也并不是有意要将代码写得如此复杂，实在是因为 Glide 的功能太强大了，而上述代码只是使用了 Glide 最最基本的功能而已。

4.2.3 深入探究 Glide 的缓存机制

BAT交流群 892872246

Glide 缓存简介

Glide 的缓存设计可以说是非常先进的，考虑的场景也很周全。在缓存这一功能上，Glide 又将它分成了两个模块，一个是内存缓存，一个是硬盘缓存。

这两个缓存模块的作用各不相同，内存缓存的主要作用是防止应用重复将图片数据读取到内存当中，而硬盘缓存的主要作用是防止应用重复从网络或其他地方重复下载和读取数据。

内存缓存和硬盘缓存的相互结合才构成了 Glide 极佳的图片缓存效果，那么接下来我们就分别来分析一下这两种缓存的使用方法以及它们的实现原理。

缓存 Key

既然是缓存功能，就必然会有用于进行缓存的 Key。那么 Glide 的缓存 Key 是怎么生成的呢？我不得不说，Glide 的缓存 Key 生成规则非常繁琐，决定缓存 Key 的参数竟然有 10 个之多。不过繁琐归繁琐，至少逻辑还是比较简单的，我们先来看一下 Glide 缓存 Key 的生成逻辑。

生成缓存 Key 的代码在 Engine 类的 load()方法当中，这部分代码我们在上一篇文章当中已经分析过了，只不过当时忽略了缓存相关的内容，那么我们现在重新来看一下：

```
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> transf
ormation, ResourceTranscoder<Z, R> transcoder,

            Priority priority, boolean isMemoryCacheable, DiskCacheStrate
gy 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, loadProv
ider.getEncoder(),

                transcoder, loadProvider.getSourceEncoder());



        ...

    }



    ...}
```

可以看到，这里在第 11 行调用了 fetcher.getId()方法获得了一个 id 字符串，这个字符串也就是我们要加载的图片的唯一标识，比如说如果是一张网络上的图片的话，那么这个 id 就是这张图片的 url 地址。

接下来在第 12 行，将这个 id 连同着 signature、width、height 等等 10 个参数一起传入到 EngineKeyFactory 的 buildKey()方法当中，从而构建出了一个 EngineKey 对象，这个 EngineKey 也就是 Glide 中的缓存 Key 了。

可见，决定缓存 Key 的条件非常多，即使你用 override()方法改变了一下图片的 width 或者 height，也会生成一个完全不同的缓存 Key。

EngineKey 类的源码大家有兴趣可以自己去看一下，其实主要就是重写了 equals()和 hashCode()方法，保证只有传入 EngineKey 的所有参数都相同的情况下才认为是同一个 EngineKey 对象，我就不在这里将源码贴出来了。

内存缓存

有了缓存 Key，接下来就可以开始进行缓存了，那么我们先从内存缓存看起。

首先你要知道，默认情况下，Glide 自动就是开启内存缓存的。也就是说，当我们使用 Glide 加载了一张图片之后，这张图片就会被缓存到内存当中，只要在它还没从内存中被清除之前，下次使用 Glide 再加载这张图片都会直接从内存当中

BAT交流群：892872246

读取，而不用重新从网络或硬盘上读取了，这样无疑就可以大幅度提升图片的加载效率。比方说你在一个 RecyclerView 当中反复上下滑动，RecyclerView 中只要是 Glide 加载过的图片都可以直接从内存当中迅速读取并展示出来，从而大大提升了用户体验。

而 Glide 最为人性化的是，你甚至不需要编写任何额外的代码就能自动享受到这个极为便利的内存缓存功能，因为 Glide 默认就已经将它开启了。

那么既然已经默认开启了这个功能，还有什么可讲的用法呢？只有一点，如果你有什么特殊的原因需要禁用内存缓存功能，Glide 对此提供了接口：

```
Glide.with(this)

    .load(url)

    .skipMemoryCache(true)

    .into(imageView);
```

可以看到，只需要调用 skipMemoryCache()方法并传入 true，就表示禁用掉 Glide 的内存缓存功能。

没错，关于 Glide 内存缓存的用法就只有这么多，可以说是相当简单。但是我们不可能只停留在这么简单的层面上，接下来就让我们就通过阅读源码来分析一下 Glide 的内存缓存功能是如何实现的。

其实说到内存缓存的实现，非常容易就让人想到 LruCache 算法（Least Recently Used），也叫近期最少使用算法。它的主要算法原理就是把最近使用的对象用强引用存储在 LinkedHashMap 中，并且把最近最少使用的对象在缓存值达到预设定值之前从内存中移除。LruCache 的用法也比较简单，我在 Android 高效加载大图、多图解决方案，有效避免程序 OOM 这篇文章当中有提到过它的用法，感兴趣的朋友可以去参考一下。

那么不必多说，Glide 内存缓存的实现自然也是使用的 LruCache 算法。不过除了 LruCache 算法之外，Glide 还结合了一种弱引用的机制，共同完成了内存缓存功能，下面就让我们来通过源码分析一下。

首先回忆一下，在上一篇文章的第二步 load()方法中，我们当时分析到了在 loadGeneric()方法中会调用 Glide.buildStreamModelLoader()方法来获取一个 ModelLoader 对象。当时没有再跟进到这个方法的里面再去分析，那么我们现在来看下它的源码：

```
public class Glide {
```

892872246

```
    public static <T, Y> ModelLoader<T, Y> buildModelLoader(Class<T> mod
elClass, Class<Y> resourceClass,

            Context context) {

        if (modelClass == null) {

            if (Log.isLoggable(TAG, Log.DEBUG)) {

                Log.d(TAG, "Unable to load null model, setting placeholder
 only");

            }

            return null;

        }

        return Glide.get(context).getLoaderFactory().buildModelLoader(mo
delClass, resourceClass);

    }



    public static Glide get(Context context) {

        if (glide == null) {

            synchronized (Glide.class) {

                if (glide == null) {

                    Context applicationContext = context.getApplicationCon
text();

                    List<GlideModule> modules = new ManifestParser(applica
tionContext).parse();

                    GlideBuilder builder = new GlideBuilder(applicationCon
text);

                    for (GlideModule module : modules) {

                        module.applyOptions(applicationContext, builder);
```

```
                }

                glide = builder.createGlide();

                for (GlideModule module : modules) {

                    module.registerComponents(applicationContext, glid
e);

                }

            }

        }

    }

    return glide;

}



...}
```

这里我们还是只看关键，在第 11 行去构建 ModelLoader 对象的时候，先调用了一个 Glide.get()方法，而这个方法就是关键。我们可以看到，get()方法中实现的是一个单例功能，而创建 Glide 对象则是在第 24 行调用 GlideBuilder 的 createGlide()方法来创建的，那么我们跟到这个方法当中：

```
public class GlideBuilder {

    ...



    Glide createGlide() {

        if (sourceService == null) {

            final int cores = Math.max(1, Runtime.getRuntime().availableP
rocessors());

            sourceService = new FifoPriorityThreadPoolExecutor(cores);

        }
```

```
        if (diskCacheService == null) {

            diskCacheService = new FifoPriorityThreadPoolExecutor(1);

        }

        MemorySizeCalculator calculator = new MemorySizeCalculator(contex
t);

        if (bitmapPool == null) {

            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {

                int size = calculator.getBitmapPoolSize();

                bitmapPool = new LruBitmapPool(size);

            } else {

                bitmapPool = new BitmapPoolAdapter();

            }

        }

        if (memoryCache == null) {

            memoryCache = new LruResourceCache(calculator.getMemoryCacheS
ize());

        }

        if (diskCacheFactory == null) {

            diskCacheFactory = new InternalCacheDiskCacheFactory(contex
t);

        }

        if (engine == null) {

            engine = new Engine(memoryCache, diskCacheFactory, diskCacheS
ervice, sourceService);

        }
```

```
        if (decodeFormat == null) {

            decodeFormat = DecodeFormat.DEFAULT;

        }

        return new Glide(engine, memoryCache, bitmapPool, context, decode
Format);

    }}
```

这里也就是构建 Glide 对象的地方了。那么观察第 22 行，你会发现这里 new 出了一个 LruResourceCache，并把它赋值到了 memoryCache 这个对象上面。你没有猜错，这个就是 Glide 实现内存缓存所使用的 LruCache 对象了。不过我这里并不打算展开来讲 LruCache 算法的具体实现，如果你感兴趣的话可以自己研究一下它的源码。

现在创建好了 LruResourceCache 对象只能说是把准备工作做好了，接下来我们就一步步研究 Glide 中的内存缓存到底是如何实现的。

刚才在 Engine 的 load()方法中我们已经看到了生成缓存 Key 的代码，而内存缓存的代码其实也是在这里实现的，那么我们重新来看一下 Engine 类 load()方法的完整源码：

```
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> transf
ormation, ResourceTranscoder<Z, R> transcoder,

            Priority priority, boolean isMemoryCacheable, DiskCacheStrate
gy 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, loadProv
ider.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, isMemoryC
acheable);

        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, ke
y);

            }

            return new LoadStatus(cb, current);

        }



        EngineJob engineJob = engineJobFactory.build(key, isMemoryCacheab
le);

        DecodeJob<T, Z, R> decodeJob = new DecodeJob<T, Z, R>(key, width,
 height, fetcher, loadProvider, transformation,

                transcoder, diskCacheProvider, diskCacheStrategy, priorit
y);

        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);

    }



    ...}
```

可以看到，这里在第 17 行调用了 loadFromCache()方法来获取缓存图片，如果获取到就直接调用 cb.onResourceReady()方法进行回调。如果没有获取到，则会在第 26 行调用 loadFromActiveResources()方法来获取缓存图片，获取到的话也直接进行回调。只有在两个方法都没有获取到缓存的情况下，才会继续向下执行，从而开启线程来加载图片。

也就是说，Glide 的图片加载过程中会调用两个方法来获取内存缓存，loadFromCache()和 loadFromActiveResources()。这两个方法中一个使用的就是 LruCache 算法，另一个使用的就是弱引用。我们来看一下它们的源码：

```
public class Engine implements EngineJobListener,

        MemoryCache.ResourceRemovedListener,

        EngineResource.ResourceListener {



    private final MemoryCache cache;

    private final Map<Key, WeakReference<EngineResource<?>>> activeResou
rces;

    ...
```

```
    private EngineResource<?> loadFromCache(Key key, boolean isMemoryCac
heable) {

        if (!isMemoryCacheable) {

            return null;

        }

        EngineResource<?> cached = getEngineResourceFromCache(key);

        if (cached != null) {

            cached.acquire();

            activeResources.put(key, new ResourceWeakReference(key, cache
d, getReferenceQueue()));

        }

        return cached;

    }



    private EngineResource<?> getEngineResourceFromCache(Key key) {

        Resource<?> cached = cache.remove(key);

        final EngineResource result;

        if (cached == null) {

            result = null;

        } else if (cached instanceof EngineResource) {

            result = (EngineResource) cached;

        } else {

            result = new EngineResource(cached, true /*isCacheable*/);

        }
```

```
        return result;

    }



    private EngineResource<?> loadFromActiveResources(Key key, boolean i
sMemoryCacheable) {

        if (!isMemoryCacheable) {

            return null;

        }

        EngineResource<?> active = null;

        WeakReference<EngineResource<?>> activeRef = activeResources.get
(key);

        if (activeRef != null) {

            active = activeRef.get();

            if (active != null) {

                active.acquire();

            } else {

                activeResources.remove(key);

            }

        }

        return active;

    }



    ...

}
```

在 loadFromCache()方法的一开始，首先就判断了 isMemoryCacheable 是不是 false，如果是 false 的话就直接返回 null。这是什么意思呢？其实很简单，我们刚刚不是学了一个 skipMemoryCache()方法吗？如果在这个方法中传入 true，那么这里的 isMemoryCacheable 就会是 false，表示内存缓存已被禁用。

我们继续往下看，接着调用了 getEngineResourceFromCache()方法来获取缓存。在这个方法中，会使用缓存 Key 来从 cache 当中取值，而这里的 cache 对象就是在构建 Glide 对象时创建的 LruResourceCache，那么说明这里其实使用的就是 LruCache 算法了。

但是呢，观察第 22 行，当我们从 LruResourceCache 中获取到缓存图片之后会将它从缓存中移除，然后在第 16 行将这个缓存图片存储到 activeResources 当中。activeResources 就是一个弱引用的 HashMap，用来缓存正在使用中的图片，我们可以看到，loadFromActiveResources()方法就是从 activeResources 这个 HashMap 当中取值的。使用 activeResources 来缓存正在使用中的图片，可以保护这些图片不会被 LruCache 算法回收掉。

好的，从内存缓存中读取数据的逻辑大概就是这些了。概括一下来说，就是如果能从内存缓存当中读取到要加载的图片，那么就直接进行回调，如果读取不到的话，才会开启线程执行后面的图片加载逻辑。

现在我们已经搞明白了内存缓存读取的原理，接下来的问题就是内存缓存是在哪里写入的呢？这里我们又要回顾一下上一篇文章中的内容了。还记不记得我们之前分析过，当图片加载完成之后，会在 EngineJob 当中通过 Handler 发送一条消息将执行逻辑切回到主线程当中，从而执行 handleResultOnMainThread()方法。那么我们现在重新来看一下这个方法，代码如下所示：

```
class EngineJob implements EngineRunnable.EngineRunnableManager {



    private final EngineResourceFactory engineResourceFactory;

    ...



    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, isCacheab
le);

        hasResource = true;

        engineResource.acquire();

        listener.onEngineJobComplete(key, engineResource);

        for (ResourceCallback cb : cbs) {

            if (!isInIgnoredCallbacks(cb)) {

                engineResource.acquire();

                cb.onResourceReady(engineResource);

            }

        }

        engineResource.release();

    }



    static class EngineResourceFactory {

        public <R> EngineResource<R> build(Resource<R> resource, boolean
isMemoryCacheable) {

            return new EngineResource<R>(resource, isMemoryCacheable);

        }

    }
```

```
    ...}
```

在第 13 行，这里通过 EngineResourceFactory 构建出了一个包含图片资源的 EngineResource 对象，然后会在第 16 行将这个对象回调到 Engine 的 onEngineJobComplete()方法当中，如下所示：

```
public class Engine implements EngineJobListener,

        MemoryCache.ResourceRemovedListener,

        EngineResource.ResourceListener {

    ...



    @Override

    public void onEngineJobComplete(Key key, EngineResource<?> resource)
 {

        Util.assertMainThread();

        // A null resource indicates that the load failed, usually due to
 an exception.

        if (resource != null) {

            resource.setResourceListener(key, this);

            if (resource.isCacheable()) {

                activeResources.put(key, new ResourceWeakReference(key, r
esource, getReferenceQueue()));

            }

        }

        jobs.remove(key);

    }



    ...}
```

现在就非常明显了，可以看到，在第 13 行，回调过来的 EngineResource 被 put 到了 activeResources 当中，也就是在这里写入的缓存。

那么这只是弱引用缓存，还有另外一种 LruCache 缓存是在哪里写入的呢？这就要介绍一下 EngineResource 中的一个引用机制了。观察刚才的 handleResultOnMainThread()方法，在第 15 行和第 19 行有调用 EngineResource 的 acquire()方法，在第 23 行有调用它的 release()方法。其实，EngineResource 是用一个 acquired 变量用来记录图片被引用的次数，调用 acquire()方法会让变量加 1，调用 release()方法会让变量减 1，代码如下所示：

```
class EngineResource<Z> implements Resource<Z> {



    private int acquired;

    ...



    void acquire() {

        if (isRecycled) {

            throw new IllegalStateException("Cannot acquire a recycled re
source");

        }

        if (!Looper.getMainLooper().equals(Looper.myLooper())) {

            throw new IllegalThreadStateException("Must call acquire on t
he main thread");

        }

        ++acquired;

    }



    void release() {

        if (acquired <= 0) {
```

```
            throw new IllegalStateException("Cannot release a recycled or
 not yet acquired resource");

        }

        if (!Looper.getMainLooper().equals(Looper.myLooper())) {

            throw new IllegalThreadStateException("Must call release on t
he main thread");

        }

        if (--acquired == 0) {

            listener.onResourceReleased(key, this);

        }

    }}
```

也就是说，当 acquired 变量大于 0 的时候，说明图片正在使用中，也就应该放到 activeResources 弱引用缓存当中。而经过 release()之后，如果 acquired 变量等于 0 了，说明图片已经不再被使用了，那么此时会在第 24 行调用 listener 的 onResourceReleased()方法来释放资源，这个 listener 就是 Engine 对象，我们来看下它的 onResourceReleased()方法：

```
public class Engine implements EngineJobListener,

        MemoryCache.ResourceRemovedListener,

        EngineResource.ResourceListener {



    private final MemoryCache cache;

    private final Map<Key, WeakReference<EngineResource<?>>> activeResou
rces;

    ...



    @Override
```

BAT交 892246

```
    public void onResourceReleased(Key cacheKey, EngineResource resource)
{

        Util.assertMainThread();

        activeResources.remove(cacheKey);

        if (resource.isCacheable()) {

            cache.put(cacheKey, resource);

        } else {

            resourceRecycler.recycle(resource);

        }

    }



    ...}
```

可以看到，这里首先会将缓存图片从 activeResources 中移除，然后再将它 put 到 LruResourceCache 当中。这样也就实现了正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用 LruCache 来进行缓存的功能。

这就是 Glide 内存缓存的实现原理。

硬盘缓存

接下来我们开始学习硬盘缓存方面的内容。

不知道你还记不记得，在本系列的第一篇文章中我们就使用过硬盘缓存的功能了。当时为了禁止 Glide 对图片进行硬盘缓存而使用了如下代码：

```
Glide.with(this)

    .load(url)

    .diskCacheStrategy(DiskCacheStrategy.NONE)

    .into(imageView);
```

BA… 2246

调用 diskCacheStrategy()方法并传入 DiskCacheStrategy.NONE，就可以禁用掉 Glide 的硬盘缓存功能了。

这个 diskCacheStrategy()方法基本上就是 Glide 硬盘缓存功能的一切，它可以接收四种参数：

DiskCacheStrategy.NONE： 表示不缓存任何内容。
DiskCacheStrategy.SOURCE： 表示只缓存原始图片。
DiskCacheStrategy.RESULT： 表示只缓存转换过后的图片（默认选项）。
DiskCacheStrategy.ALL ： 表示既缓存原始图片，也缓存转换过后的图片。
上面四种参数的解释本身并没有什么难理解的地方，但是有一个概念大家需要了解，就是当我们使用 Glide 去加载一张图片的时候，Glide 默认并不会将原始图片展示出来，而是会对图片进行压缩和转换（我们会在后面学习这方面的内容）。总之就是经过种种一系列操作之后得到的图片，就叫转换过后的图片。而 Glide 默认情况下在硬盘缓存的就是转换过后的图片，我们通过调用 diskCacheStrategy()方法则可以改变这一默认行为。

好的，关于 Glide 硬盘缓存的用法也就只有这么多，那么接下来还是老套路，我们通过阅读源码来分析一下，Glide 的硬盘缓存功能是如何实现的。

首先，和内存缓存类似，硬盘缓存的实现也是使用的 LruCache 算法，而且 Google 还提供了一个现成的工具类 DiskLruCache。我之前也专门写过一篇文章对这个 DiskLruCache 工具进行了比较全面的分析，感兴趣的朋友可以参考一下 Android DiskLruCache 完全解析，硬盘缓存的最佳方案 。当然，Glide 是使用的自己编写的 DiskLruCache 工具类，但是基本的实现原理都是差不多的。

接下来我们看一下 Glide 是在哪里读取硬盘缓存的。这里又需要回忆一下上篇文章中的内容了，Glide 开启线程来加载图片后会执行 EngineRunnable 的 run()方法，run()方法中又会调用一个 decode()方法，那么我们重新再来看一下这个 decode()方法的源码：

```
private Resource<?> decode() throws Exception {

    if (isDecodingFromCache()) {

        return decodeFromCache();

    } else {

        return decodeFromSource();

    }}
```

可以看到，这里会分为两种情况，一种是调用 decodeFromCache()方法从硬盘缓存当中读取图片，一种是调用 decodeFromSource()来读取原始图片。默认情况

下 Glide 会优先从缓存当中读取，只有缓存中不存在要读取的图片时，才会去读取原始图片。那么我们现在来看一下 decodeFromCache()方法的源码，如下所示：

```
private Resource<?> decodeFromCache() throws Exception {

    Resource<?> result = null;

    try {

        result = decodeJob.decodeResultFromCache();

    } catch (Exception e) {

        if (Log.isLoggable(TAG, Log.DEBUG)) {

            Log.d(TAG, "Exception decoding result from cache: " + e);

        }

    }

    if (result == null) {

        result = decodeJob.decodeSourceFromCache();

    }

    return result;

}
```

可以看到，这里会先去调用 DecodeJob 的 decodeResultFromCache()方法来获取缓存，如果获取不到，会再调用 decodeSourceFromCache()方法获取缓存，这两个方法的区别其实就是 DiskCacheStrategy.RESULT 和 DiskCacheStrategy.SOURCE 这两个参数的区别，相信不需要我再做什么解释吧。

那么我们来看一下这两个方法的源码吧，如下所示：

```
public Resource<Z> decodeResultFromCache() throws Exception {

    if (!diskCacheStrategy.cacheResult()) {

        return null;

    }
```

```
    long startTime = LogTime.getLogTime();

    Resource<T> transformed = loadFromCache(resultKey);

    startTime = LogTime.getLogTime();

    Resource<Z> result = transcode(transformed);

    return result;}

public Resource<Z> decodeSourceFromCache() throws Exception {

    if (!diskCacheStrategy.cacheSource()) {

        return null;

    }

    long startTime = LogTime.getLogTime();

    Resource<T> decoded = loadFromCache(resultKey.getOriginalKey());

    return transformEncodeAndTranscode(decoded);}
```

可以看到，它们都是调用了 loadFromCache()方法从缓存当中读取数据，如果是 decodeResultFromCache()方法就直接将数据解码并返回，如果是 decodeSourceFromCache()方法，还要调用一下 transformEncodeAndTranscode()方法先将数据转换一下再解码并返回。

然而我们注意到，这两个方法中在调用 loadFromCache()方法时传入的参数却不一样，一个传入的是 resultKey，另外一个却又调用了 resultKey 的 getOriginalKey()方法。这个其实非常好理解，刚才我们已经解释过了，Glide 的缓存 Key 是由 10 个参数共同组成的，包括图片的 width、height 等等。但如果我们是缓存的原始图片，其实并不需要这么多的参数，因为不用对图片做任何的变化。那么我们来看一下 getOriginalKey()方法的源码：

```
public Key getOriginalKey() {

    if (originalKey == null) {

        originalKey = new OriginalKey(id, signature);

    }

    return originalKey;}
```

可以看到，这里其实就是忽略了绝大部分的参数，只使用了 id 和 signature 这两个参数来构成缓存 Key。而 signature 参数绝大多数情况下都是用不到的，因此基本上可以说就是由 id（也就是图片 url）来决定的 Original 缓存 Key。

搞明白了这两种缓存 Key 的区别，那么接下来我们看一下 loadFromCache()方法的源码吧：

```
private Resource<T> loadFromCache(Key key) throws IOException {

    File cacheFile = diskCacheProvider.getDiskCache().get(key);

    if (cacheFile == null) {

        return null;

    }

    Resource<T> result = null;

    try {

        result = loadProvider.getCacheDecoder().decode(cacheFile, width,
height);

    } finally {

        if (result == null) {

            diskCacheProvider.getDiskCache().delete(key);

        }

    }

    return result;}
```

这个方法的逻辑非常简单，调用 getDiskCache()方法获取到的就是 Glide 自己编写的 DiskLruCache 工具类的实例，然后调用它的 get()方法并把缓存 Key 传入，就能得到硬盘缓存的文件了。如果文件为空就返回 null，如果文件不为空则将它解码成 Resource 对象后返回即可。

这样我们就将硬盘缓存读取的源码分析完了，那么硬盘缓存又是在哪里写入的呢？趁热打铁我们赶快继续分析下去。

BAT交流群：892872246

刚才已经分析过了，在没有缓存的情况下，会调用 decodeFromSource()方法来读取原始图片。那么我们来看下这个方法：

```
public Resource<Z> decodeFromSource() throws Exception {

    Resource<T> decoded = decodeSource();

    return transformEncodeAndTranscode(decoded);}
```

这个方法中只有两行代码，decodeSource()顾名思义是用来解析原图片的，而transformEncodeAndTranscode()则是用来对图片进行转换和转码的。我们先来看 decodeSource()方法：

```
private Resource<T> decodeSource() throws Exception {

    Resource<T> decoded = null;

    try {

        long startTime = LogTime.getLogTime();

        final A data = fetcher.loadData(priority);

        if (isCancelled) {

            return null;

        }

        decoded = decodeFromSourceData(data);

    } finally {

        fetcher.cleanup();

    }

    return decoded;}

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, hei
ght);

    }

    return decoded;}

private Resource<T> cacheAndDecodeSourceData(A data) throws IOException
{

    long startTime = LogTime.getLogTime();

    SourceWriter<A> writer = new SourceWriter<A>(loadProvider.getSourceE
ncoder(), data);

    diskCacheProvider.getDiskCache().put(resultKey.getOriginalKey(), wri
ter);

    startTime = LogTime.getLogTime();

    Resource<T> result = loadFromCache(resultKey.getOriginalKey());

    return result;}
```

这里会在第 5 行先调用 fetcher 的 loadData()方法读取图片数据，然后在第 9 行调用 decodeFromSourceData()方法来对图片进行解码。接下来会在第 18 行先判断是否允许缓存原始图片，如果允许的话又会调用 cacheAndDecodeSourceData()方法。而在这个方法中同样调用了 getDiskCache()方法来获取 DiskLruCache 实例，接着调用它的 put()方法就可以写入硬盘缓存了，注意原始图片的缓存 Key 是用的 resultKey.getOriginalKey()。

好的，原始图片的缓存写入就是这么简单，接下来我们分析一下 transformEncodeAndTranscode()方法的源码，来看看转换过后的图片缓存是怎么写入的。代码如下所示：

```
private Resource<Z> transformEncodeAndTranscode(Resource<T> decoded) {

    long startTime = LogTime.getLogTime();

    Resource<T> transformed = transform(decoded);
```

```
    writeTransformedToCache(transformed);

    startTime = LogTime.getLogTime();

    Resource<Z> result = transcode(transformed);

    return result;}

private void writeTransformedToCache(Resource<T> transformed) {

    if (transformed == null || !diskCacheStrategy.cacheResult()) {

        return;

    }

    long startTime = LogTime.getLogTime();

    SourceWriter<Resource<T>> writer = new SourceWriter<Resource<T>>(loa
dProvider.getEncoder(), transformed);

    diskCacheProvider.getDiskCache().put(resultKey, writer);}
```

这里的逻辑就更加简单明了了。先是在第 3 行调用 transform()方法来对图片进行转换，然后在 writeTransformedToCache()方法中将转换过后的图片写入到硬盘缓存中，调用的同样是 DiskLruCache 实例的 put()方法，不过这里用的缓存 Key 是 resultKey。

这样我们就将 Glide 硬盘缓存的实现原理也分析完了。虽然这些源码看上去如此的复杂，但是经过 Glide 出色的封装，使得我们只需要通过 skipMemoryCache()和 diskCacheStrategy()这两个方法就可以轻松自如地控制 Glide 的缓存功能了。

了解了 Glide 缓存的实现原理之后，接下来我们再来学习一些 Glide 缓存的高级技巧吧。

高级技巧

虽说 Glide 将缓存功能高度封装之后，使得用法变得非常简单，但同时也带来了一些问题。

比如之前有一位群里的朋友就跟我说过，他们项目的图片资源都是存放在七牛云上面的，而七牛云为了对图片资源进行保护，会在图片 url 地址的基础之上再加上一个 token 参数。也就是说，一张图片的 url 地址可能会是如下格式：

BAT交流群：892872246

```
http://url.com/image.jpg?token=d9caa6e02c990b0a

http://url.com/image.jpg?token=d9caa6e02c990b0a
```

而使用 Glide 加载这张图片的话，也就会使用这个 url 地址来组成缓存 Key。

但是接下来问题就来了，token 作为一个验证身份的参数并不是一成不变的，很有可能时时刻刻都在变化。而如果 token 变了，那么图片的 url 也就跟着变了，图片 url 变了，缓存 Key 也就跟着变了。结果就造成了，明明是同一张图片，就因为 token 不断在改变，导致 Glide 的缓存功能完全失效了。

这其实是个挺棘手的问题，而且我相信绝对不仅仅是七牛云这一个个例，大家在使用 Glide 的时候很有可能都会遇到这个问题。

那么该如何解决这个问题呢？我们还是从源码的层面进行分析，首先再来看一下 Glide 生成缓存 Key 这部分的代码：

```
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> transf
ormation, ResourceTranscoder<Z, R> transcoder,

            Priority priority, boolean isMemoryCacheable, DiskCacheStrate
gy diskCacheStrategy, ResourceCallback cb) {

        Util.assertMainThread();

        long startTime = LogTime.getLogTime();



        final String id = fetcher.getId();
```

BAT文 /2246

```
        EngineKey key = keyFactory.buildKey(id, signature, width, height,
 loadProvider.getCacheDecoder(),

                loadProvider.getSourceDecoder(), transformation, loadProv
ider.getEncoder(),

                transcoder, loadProvider.getSourceEncoder());



        ...

    }



    ...}
```

这里我将代码稍微进行了一点简化，这样看上去更加简单明了。GlideUrl 类的构造函数接收两种类型的参数，一种是 url 字符串，一种是 URL 对象。然后 getCacheKey()方法中的判断逻辑非常简单，如果传入的是 url 字符串，那么就直接返回这个字符串本身，如果传入的是 URL 对象，那么就返回这个对象 toString()后的结果。

其实看到这里，我相信大家已经猜到解决方案了，因为 getCacheKey()方法中的逻辑太直白了，直接就是将图片的 url 地址进行返回来作为缓存 Key 的。那么其实我们只需要重写这个 getCacheKey()方法，加入一些自己的逻辑判断，就能轻松解决掉刚才的问题了。

创建一个 MyGlideUrl 继承自 GlideUrl，代码如下所示：

```
public class MyGlideUrl extends GlideUrl {



    private String mUrl;



    public MyGlideUrl(String url) {

        super(url);

        mUrl = url;
```

```
    }

    @Override
    public String getCacheKey() {
        return mUrl.replace(findTokenParam(), "");
    }

    private String findTokenParam() {
        String tokenParam = "";
        int tokenKeyIndex = mUrl.indexOf("?token=") >= 0 ? mUrl.indexOf("?
token=") : mUrl.indexOf("&token=");
        if (tokenKeyIndex != -1) {
            int nextAndIndex = mUrl.indexOf("&", tokenKeyIndex + 1);
            if (nextAndIndex != -1) {
                tokenParam = mUrl.substring(tokenKeyIndex + 1, nextAndInd
ex + 1);
            } else {
                tokenParam = mUrl.substring(tokenKeyIndex);
            }
        }
        return tokenParam;
    }
}
```

BAT交 892246

可以看到，这里我们重写了 getCacheKey()方法，在里面加入了一段逻辑用于将图片 url 地址中 token 参数的这一部分移除掉。这样 getCacheKey()方法得到的就是一个没有 token 参数的 url 地址，从而不管 token 怎么变化，最终 Glide 的缓存 Key 都是固定不变的了。

当然，定义好了 MyGlideUrl，我们还得使用它才行，将加载图片的代码改成如下方式即可：

```
Glide.with(this)

    .load(new MyGlideUrl(url))

    .into(imageView);
```

也就是说，我们需要在 load()方法中传入这个自定义的 MyGlideUrl 对象，而不能再像之前那样直接传入 url 字符串了。不然的话 Glide 在内部还是会使用原始的 GlideUrl 类，而不是我们自定义的 MyGlideUrl 类。

这样我们就将这个棘手的缓存问题给解决掉了。

4.2.4 玩转 Glide 的回调与监听

回调的源码实现

作为一名 Glide 老手，相信大家对于 Glide 的基本用法已经非常熟练了。我们都知道，使用 Glide 在界面上加载并展示一张图片只需要一行代码：

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

而在这一行代码的背后，Glide 帮我们执行了成千上万行的逻辑。其实在第二篇文章当中，我们已经分析了这一行代码背后的完整执行流程，但是这里我准备再带着大家单独回顾一下回调这部分的源码，这将有助于我们今天这篇文章的学习。

首先来看一下 into()方法，这里我们将 ImageView 的实例传入到 into()方法当中，Glide 将图片加载完成之后，图片就能显示到 ImageView 上了。这是怎么实现的呢？我们来看一下 into()方法的源码：

```
public Target<TranscodeType> into(ImageView view) {

    Util.assertMainThread();
```

892872246

```
    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;
            default:
                // Do nothing.
        }
    }
    return into(glide.buildImageViewTarget(view, transcodeClass));}
```

可以看到，最后一行代码会调用 glide.buildImageViewTarget()方法构建出一个 Target 对象，然后再把它传入到另一个接收 Target 参数的 into()方法中。Target 对象则是用来最终展示图片用的，如果我们跟进到 glide.buildImageViewTarget()方法中，你会看到如下的源码：

```
public class ImageViewTargetFactory {
```

892872246

```
    @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: " + claz
z

                    + ", try .as*(Class).transcode(ResourceTranscoder)");

        }

    }}
```

buildTarget()方法会根据传入的 class 参数来构建不同的 Target 对象，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。

之后就会把这里构建出来的 Target 对象传入到 GenericRequest 当中，而 Glide 在图片加载完成之后又会回调 GenericRequest 的 onResourceReady()方法，我们来看一下这部分源码：

```
public final class GenericRequest<A, T, Z, R> implements Request, SizeRe
adyCallback,

        ResourceCallback {
```

```
    private Target<R> target;

    ...



    private void onResourceReady(Resource<?> resource, R result) {

        boolean isFirstResource = isFirstReadyResource();

        status = Status.COMPLETE;

        this.resource = resource;

        if (requestListener == null || !requestListener.onResourceReady(r
esult, model, target,

                loadedFromMemoryCache, isFirstResource)) {

            GlideAnimation<R> animation = animationFactory.build(loadedFr
omMemoryCache, isFirstResource);

            target.onResourceReady(result, animation);

        }

        notifyLoadSuccess();

    }

    ...}
```

这里在第 14 行调用了 target.onResourceReady()方法，而刚才我们已经知道，这里的 target 就是 GlideDrawableImageViewTarget 对象，那么我们再来看一下它的源码：

```
public class GlideDrawableImageViewTarget extends ImageViewTarget<GlideD
rawable> {

    ...



    @Override
```

BAT交 892246

```
    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);

    }



    ...}
```

可以看到，这里在 onResourceReady()方法中处理了图片展示，还有 GIF 播放的逻辑，那么一张图片也就显示出来了，这也就是 Glide 回调的基本实现原理。

好的，那么原理就先分析到这儿，接下来我们就来看一下在回调和监听方面还有哪些知识是可以扩展的。

into()方法

使用了这么久的 Glide，我们都知道 into()方法中是可以传入 ImageView 的。那么 into()方法还可以传入别的参数吗？我可以让 Glide 加载出来的图片不显示到 ImageView 上吗？答案是肯定的，这就需要用到自定义 Target 功能。

其实通过上面的分析，我们已经知道了，into()方法还有一个接收 Target 参数的重载。即使我们传入的参数是 ImageView，Glide 也会在内部自动构建一个 Target 对象。而如果我们能够掌握自定义 Target 技术的话，就可以更加随心所欲地控制 Glide 的回调了。

我们先来看一下 Glide 中 Target 的继承结构图吧，如下所示：

可以看到，Target 的继承结构还是相当复杂的，实现 Target 接口的子类非常多。不过你不用被这么多的子类所吓到，这些大多数都是 Glide 已经实现好的具备完整功能的 Target 子类，如果我们要进行自定义的话，通常只需要在两种 Target 的基础上去自定义就可以了，一种是 SimpleTarget，一种是 ViewTarget。

接下来我就分别以这两种 Target 来举例，学习一下自定义 Target 的功能。

BAT交流群：892872246

首先来看 SimpleTarget，顾名思义，它是一种极为简单的 Target，我们使用它可以将 Glide 加载出来的图片对象获取到，而不是像之前那样只能将图片在 ImageView 上显示出来。

那么下面我们来看一下 SimpleTarget 的用法示例吧，其实非常简单：

```
SimpleTarget<GlideDrawable> simpleTarget = new SimpleTarget<GlideDrawabl
e>() {

    @Override

    public void onResourceReady(GlideDrawable resource, GlideAnimation g
lideAnimation) {

        imageView.setImageDrawable(resource);

    }};

public void loadImage(View view) {

    String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473
_1920x1080.jpg";

    Glide.with(this)

        .load(url)

        .into(simpleTarget);}
```

怎么样？不愧是 SimpleTarget 吧，短短几行代码就搞了。这里我们创建了一个 SimpleTarget 的实例，并且指定它的泛型是 GlideDrawable，然后重写了 onResourceReady()方法。在 onResourceReady()方法中，我们就可以获取到 Glide 加载出来的图片对象了，也就是方法参数中传过来的 GlideDrawable 对象。有了这个对象之后你可以使用它进行任意的逻辑操作，这里我只是简单地把它显示到了 ImageView 上。

SimpleTarget 的实现创建好了，那么只需要在加载图片的时候将它传入到 into()方法中就可以了，现在运行一下程序，效果如下图所示。

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

虽然目前这个效果和直接在 into()方法中传入 ImageView 并没有什么区别，但是我们已经拿到了图片对象的实例，然后就可以随意做更多的事情了。

当然，SimpleTarget 中的泛型并不一定只能是 GlideDrawable，如果你能确定你正在加载的是一张静态图而不是 GIF 图的话，我们还能直接拿到这张图的 Bitmap 对象，如下所示：

```
SimpleTarget<Bitmap> simpleTarget = new SimpleTarget<Bitmap>() {

    @Override

    public void onResourceReady(Bitmap resource, GlideAnimation glideAni
mation) {

        imageView.setImageBitmap(resource);

    }};

public void loadImage(View view) {
```

892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
    String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473
_1920x1080.jpg";

    Glide.with(this)

        .load(url)

        .asBitmap()

        .into(simpleTarget);}
```

可以看到，这里我们将 SimpleTarget 的泛型指定成 Bitmap，然后在加载图片的时候调用了 asBitmap()方法强制指定这是一张静态图，这样就能在 onResourceReady()方法中获取到这张图的 Bitmap 对象了。

好了，SimpleTarget 的用法就是这么简单，接下来我们学习一下 ViewTarget 的用法。

事实上，从刚才的继承结构图上就能看出，Glide 在内部自动帮我们创建的 GlideDrawableImageViewTarget 就是 ViewTarget 的子类。只不过 GlideDrawableImageViewTarget 被限定只能作用在 ImageView 上，而 ViewTarget 的功能更加广泛，它可以作用在任意的 View 上。

这里我们还是通过一个例子来演示一下吧，比如我创建了一个自定义布局 MyLayout，如下所示：

```
public class MyLayout extends LinearLayout {



    private ViewTarget<MyLayout, GlideDrawable> viewTarget;



    public MyLayout(Context context, AttributeSet attrs) {

        super(context, attrs);

        viewTarget = new ViewTarget<MyLayout, GlideDrawable>(this) {

            @Override

            public void onResourceReady(GlideDrawable resource, GlideAnim
ation glideAnimation) {
```

```
                MyLayout myLayout = getView();

                myLayout.setImageAsBackground(resource);

            }

        };

    }



    public ViewTarget<MyLayout, GlideDrawable> getTarget() {

        return viewTarget;

    }



    public void setImageAsBackground(GlideDrawable resource) {

        setBackground(resource);

    }

}
```

在 MyLayout 的构造函数中，我们创建了一个 ViewTarget 的实例，并将 Mylayout 当前的实例 this 传了进去。ViewTarget 中需要指定两个泛型，一个是 View 的类型，一个图片的类型（GlideDrawable 或 Bitmap）。然后在 onResourceReady()方法中，我们就可以通过 getView()方法获取到 MyLayout 的实例，并调用它的任意接口了。比如说这里我们调用了 setImageAsBackground()方法来将加载出来的图片作为 MyLayout 布局的背景图。

接下来看一下怎么使用这个 Target 吧，由于 MyLayout 中已经提供了 getTarget()接口，我们只需要在加载图片的地方这样写就可以了：

```
public class MainActivity extends AppCompatActivity {



    MyLayout myLayout;
```

```
    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        myLayout = (MyLayout) findViewById(R.id.background);

    }



    public void loadImage(View view) {

        String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN733679
5473_1920x1080.jpg";

        Glide.with(this)

            .load(url)

            .into(myLayout.getTarget());

    }

}
```

就是这么简单，在 into()方法中传入 myLayout.getTarget()即可。现在重新运行一下程序，效果如下图所示。

BAT交流群：892872246

好的，关于自定义 Target 的功能我们就介绍这么多，这些虽说都是自定义 Target 最基本的用法，但掌握了这些用法之后，你就能应对各种各样复杂的逻辑了。

preload()方法

Glide 加载图片虽说非常智能，它会自动判断该图片是否已经有缓存了，如果有的话就直接从缓存中读取，没有的话再从网络去下载。但是如果我希望提前对图片进行一个预加载，等真正需要加载图片的时候就直接从缓存中读取，不想再等待慢长的网络加载时间了，这该怎么办呢？

对于很多 Glide 新手来说这确实是一个烦恼的问题，因为在没有学习本篇文章之前，into()方法中必须传入一个 ImageView 呀，而传了 ImageView 之后图片就显示出来了，这还怎么预加载呢？

不过在学习了本篇文章之后，相信你已经能够想到解决方案了。因为 into()方法中除了传入 ImageView 之后还可以传入 Target 对象，如果我们在 Target 对象的 onResourceReady()方法中做一个空实现，也就是不做任何逻辑处理，那么图片自然也就显示不出来了，而 Glide 的缓存机制却仍然还会正常工作，这样不就实现预加载功能了吗？

BAT交流群：892872246

没错，上述的做法完全可以实现预加载功能，不过有没有感觉这种实现方式有点笨笨的。事实上，Glide 专门给我们提供了预加载的接口，也就是 preload()方法，我们只需要直接使用就可以了。

preload()方法有两个方法重载，一个不带参数，表示将会加载图片的原始尺寸，另一个可以通过参数指定加载图片的宽和高。

preload()方法的用法也非常简单，直接使用它来替换 into()方法即可，如下所示：

```
Glide.with(this)
    .load(url)
    .diskCacheStrategy(DiskCacheStrategy.SOURCE)
    .preload();
```

需要注意的是，我们如果使用了 preload()方法，最好要将 diskCacheStrategy 的缓存策略指定成 DiskCacheStrategy.SOURCE。因为 preload()方法默认是预加载的原始图片大小，而 into()方法则默认会根据 ImageView 控件的大小来动态决定加载图片的大小。因此，如果不将 diskCacheStrategy 的缓存策略指定成 DiskCacheStrategy.SOURCE 的话，很容易会造成我们在预加载完成之后再使用 into()方法加载图片，却仍然还是要从网络上去请求图片这种现象。

调用了预加载之后，我们以后想再去加载这张图片就会非常快了，因为 Glide 会直接从缓存当中去读取图片并显示出来，代码如下所示：

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

注意，这里我们仍然需要使用 diskCacheStrategy()方法将硬盘缓存策略指定成 DiskCacheStrategy.SOURCE，以保证 Glide 一定会去读取刚才预加载的图片缓存。

preload()方法的用法大概就是这么简单，但是仅仅会使用显然层次有些太低了，下面我们就满足一下好奇心，看看它的源码是如何实现的。

和 into()方法一样，preload()方法也是在 GenericRequestBuilder 类当中的，代码如下所示：

BAT交流群：892872246

```
public class GenericRequestBuilder<ModelType, DataType, ResourceType, Tr
anscodeType> implements Cloneable {

    ...



    public Target<TranscodeType> preload(int width, int height) {

        final PreloadTarget<TranscodeType> target = PreloadTarget.obtain
(width, height);

        return into(target);

    }



    public Target<TranscodeType> preload() {

        return preload(Target.SIZE_ORIGINAL, Target.SIZE_ORIGINAL);

    }



    ...}
```

如刚才所说，preload()方法有两个方法重载，你可以调用带参数的 preload()方法来明确指定图片的宽和高，也可以调用不带参数的 preload()方法，它会在内部自动将图片的宽和高都指定成 Target.SIZE_ORIGINAL，也就是图片的原始尺寸。

然后我们可以看到，这里在第 5 行调用了 PreloadTarget.obtain()方法获取一个 PreloadTarget 的实例，并把它传入到了 into()方法当中。从刚才的继承结构图中可以看出，PreloadTarget 是 SimpleTarget 的子类，因此它是可以直接传入到 into()方法中的。

那么现在的问题就是，PreloadTarget 具体的实现到底是什么样子的了，我们看一下它的源码，如下所示：

```
public final class PreloadTarget<Z> extends SimpleTarget<Z> {



    public static <Z> PreloadTarget<Z> obtain(int width, int height) {
```

892872246

```
        return new PreloadTarget<Z>(width, height);

    }



    private PreloadTarget(int width, int height) {

        super(width, height);

    }



    @Override

    public void onResourceReady(Z resource, GlideAnimation<? super Z> gl
ideAnimation) {

        Glide.clear(this);

    }}
```

PreloadTarget 的源码非常简单，obtain()方法中就是 new 了一个 PreloadTarget 的实例而已，而 onResourceReady()方法中也没做什么事情，只是调用了 Glide.clear()方法。

这里的 Glide.clear()并不是清空缓存的意思，而是表示加载已完成，释放资源的意思，因此不用在这里产生疑惑。

其实 PreloadTarget 的思想和我们刚才提到设计思路是一样的，就是什么都不做就可以了。因为图片加载完成之后只将它缓存而不去显示它，那不就相当于预加载了嘛。

preload()方法不管是在用法方面还是源码实现方面都还是非常简单的，那么关于这个方法我们就学到这里。

downloadOnly()方法

一直以来，我们使用 Glide 都是为了将图片显示到界面上。虽然我们知道 Glide 会在图片的加载过程中对图片进行缓存，但是缓存文件到底是存在哪里的，以及如何去直接访问这些缓存文件？我们都还不知道。

其实 Glide 将图片加载接口设计成这样也是希望我们使用起来更加的方便，不用过多去考虑底层的实现细节。但如果我现在就是想要去访问图片的缓存文件该怎么办呢？这就需要用到 downloadOnly()方法了。

BAT交流群：892872246

和 preload()方法类似，downloadOnly()方法也是可以替换 into()方法的，不过 downloadOnly()方法的用法明显要比 preload()方法复杂不少。顾名思义，downloadOnly()方法表示只会下载图片，而不会对图片进行加载。当图片下载完成之后，我们可以得到图片的存储路径，以便后续进行操作。

那么首先我们还是先来看下基本用法。downloadOnly()方法是定义在 DrawableTypeRequest 类当中的，它有两个方法重载，一个接收图片的宽度和高度，另一个接收一个泛型对象，如下所示：

downloadOnly(int width, int height)
downloadOnly(Y target)
这两个方法各自有各自的应用场景，其中 downloadOnly(int width, int height)是用于在子线程中下载图片的，而 downloadOnly(Y target)是用于在主线程中下载图片的。

那么我们先来看 downloadOnly(int width, int height)的用法。当调用了 downloadOnly(int width, int height)方法后会立即返回一个 FutureTarget 对象，然后 Glide 会在后台开始下载图片文件。接下来我们调用 FutureTarget 的 get()方法就可以去获取下载好的图片文件了，如果此时图片还没有下载完，那么 get()方法就会阻塞住，一直等到图片下载完成才会有值返回。

下面我们通过一个例子来演示一下吧，代码如下所示：

```
public void downloadImage(View view) {

    new Thread(new Runnable() {

        @Override

        public void run() {

            try {

                String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-C
N7336795473_1920x1080.jpg";

                final Context context = getApplicationContext();

                FutureTarget<File> target = Glide.with(context)

                                                .load(url)

                                                .downloadOnly(Target.SIZE_O
RIGINAL, Target.SIZE_ORIGINAL);
```

```
                final File imageFile = target.get();

                runOnUiThread(new Runnable() {

                    @Override

                    public void run() {

                        Toast.makeText(context, imageFile.getPath(), Toast.
LENGTH_LONG).show();

                    }

                });

            } catch (Exception e) {

                e.printStackTrace();

            }

        }

    }).start();}
```

这段代码稍微有一点点长，我带着大家解读一下。首先刚才说了，downloadOnly(int width, int height)方法必须要用在子线程当中，因此这里的第一步就是 new 了一个 Thread。在子线程当中，我们先获取了一个 Application Context，这个时候不能再用 Activity 作为 Context 了，因为会有 Activity 销毁了但子线程还没执行完这种可能出现。

接下来就是 Glide 的基本用法，只不过将 into()方法替换成了 downloadOnly()方法。downloadOnly()方法会返回一个 FutureTarget 对象，这个时候其实 Glide 已经开始在后台下载图片了，我们随时都可以调用 FutureTarget 的 get()方法来获取下载的图片文件，只不过如果图片还没下载好线程会暂时阻塞住，等下载完成了才会把图片的 File 对象返回。

最后，我们使用 runOnUiThread()切回到主线程，然后使用 Toast 将下载好的图片文件路径显示出来。

BAT交流群：892872246

现在重新运行一下代码，效果如下图所示。

这样我们就能清晰地看出来图片完整的缓存路径是什么了。

之后我们可以使用如下代码去加载这张图片，图片就会立即显示出来，而不用再去网络上请求了：

```
public void loadImage(View view) {

    String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473
_1920x1080.jpg";

    Glide.with(this)

            .load(url)

            .diskCacheStrategy(DiskCacheStrategy.SOURCE)

            .into(imageView);}
```

需要注意的是，这里必须将硬盘缓存策略指定成 DiskCacheStrategy.SOURCE 或者 DiskCacheStrategy.ALL，否则 Glide 将无法使用我们刚才下载好的图片缓存文件。

现在重新运行一下代码，效果如下图所示。

可以看到，图片的加载和显示是非常快的，因为 Glide 直接使用的是刚才下载好的缓存文件。

那么这个 downloadOnly(int width, int height)方法的工作原理到底是什么样的呢？我们来简单快速地看一下它的源码吧。

首先在 DrawableTypeRequest 类当中可以找到定义这个方法的地方，如下所示：

```
public class DrawableTypeRequest<ModelType> extends DrawableRequestBuild
er<ModelType>

        implements DownloadOptions {

    ...
```

```
    public FutureTarget<File> downloadOnly(int width, int height) {

        return getDownloadOnlyRequest().downloadOnly(width, height);

    }



    private GenericTranscodeRequest<ModelType, InputStream, File> getDow
nloadOnlyRequest() {

        return optionsApplier.apply(new GenericTranscodeRequest<ModelTyp
e, InputStream, File>(

            File.class, this, streamModelLoader, InputStream.class, File.
class, optionsApplier));

    }}
```

这里会先调用 getDownloadOnlyRequest()方法得到一个

GenericTranscodeRequest 对象，然后再调用它的 downloadOnly()方法，代码如下所示：

```
public class GenericTranscodeRequest<ModelType, DataType, ResourceType>

    implements DownloadOptions {

    ...



    public FutureTarget<File> downloadOnly(int width, int height) {

        return getDownloadOnlyRequest().into(width, height);

    }



    private GenericRequestBuilder<ModelType, DataType, File, File> getDo
wnloadOnlyRequest() {
```

```
        ResourceTranscoder<File, File> transcoder = UnitTranscoder.get();

        DataLoadProvider<DataType, File> dataLoadProvider = glide.buildDa
taProvider(dataClass, File.class);

        FixedLoadProvider<ModelType, DataType, File, File> fixedLoadProvi
der =

            new FixedLoadProvider<ModelType, DataType, File, File>(modelL
oader, transcoder, dataLoadProvider);

        return optionsApplier.apply(

                new GenericRequestBuilder<ModelType, DataType, File, Fil
e>(fixedLoadProvider,

                File.class, this))

                .priority(Priority.LOW)

                .diskCacheStrategy(DiskCacheStrategy.SOURCE)

                .skipMemoryCache(true);

    }}
```

这里又是调用了一个 getDownloadOnlyRequest()方法来构建了一个图片下载的请求，getDownloadOnlyRequest()方法会返回一个 GenericRequestBuilder 对象，接着调用它的 into(width, height)方法，我们继续跟进去瞧一瞧：

```
public FutureTarget<TranscodeType> into(int width, int height) {

    final RequestFutureTarget<ModelType, TranscodeType> target =

            new RequestFutureTarget<ModelType, TranscodeType>(glide.getMa
inHandler(), width, height);

    glide.getMainHandler().post(new Runnable() {

        @Override

        public void run() {

            if (!target.isCancelled()) {

                into(target);
```

```
            }

        }

    });

    return target;}
```

可以看到，这里首先是 new 出了一个 RequestFutureTarget 对象，RequestFutureTarget 也是 Target 的子类之一。然后通过 Handler 将线程切回到主线程当中，再将这个 RequestFutureTarget 传入到 into()方法当中。

那么也就是说，其实这里就是调用了接收 Target 参数的 into()方法，然后 Glide 就开始执行正常的图片加载逻辑了。那么现在剩下的问题就是，这个 RequestFutureTarget 中到底处理了些什么逻辑？我们打开它的源码看一看：

```
public class RequestFutureTarget<T, R> implements FutureTarget<R>, Runna
ble {

    ...



    @Override

    public R get() throws InterruptedException, ExecutionException {

        try {

            return doGet(null);

        } catch (TimeoutException e) {

            throw new AssertionError(e);

        }

    }



    @Override

    public R get(long time, TimeUnit timeUnit) throws InterruptedExcepti
on, ExecutionException,
```

```
        TimeoutException {

        return doGet(timeUnit.toMillis(time));

    }



    @Override

    public void getSize(SizeReadyCallback cb) {

        cb.onSizeReady(width, height);

    }



    @Override

    public synchronized void onLoadFailed(Exception e, Drawable errorDra
wable) {

        exceptionReceived = true;

        this.exception = e;

        waiter.notifyAll(this);

    }



    @Override

    public synchronized void onResourceReady(R resource, GlideAnimation<?
 super R> glideAnimation) {

        resultReceived = true;

        this.resource = resource;

        waiter.notifyAll(this);

    }
```

```
    private synchronized R doGet(Long timeoutMillis) throws ExecutionException, InterruptedException,
        TimeoutException {
        if (assertBackgroundThread) {
            Util.assertBackgroundThread();
        }

        if (isCancelled) {
            throw new CancellationException();
        } else if (exceptionReceived) {
            throw new ExecutionException(exception);
        } else if (resultReceived) {
            return resource;
        }

        if (timeoutMillis == null) {
            waiter.waitForTimeout(this, 0);
        } else if (timeoutMillis > 0) {
            waiter.waitForTimeout(this, timeoutMillis);
        }

        if (Thread.interrupted()) {
```

```
            throw new InterruptedException();
        } else if (exceptionReceived) {
            throw new ExecutionException(exception);
        } else if (isCancelled) {
            throw new CancellationException();
        } else if (!resultReceived) {
            throw new TimeoutException();
        }

        return resource;
    }

    static class Waiter {

        public void waitForTimeout(Object toWaitOn, long timeoutMillis) t
hrows InterruptedException {
            toWaitOn.wait(timeoutMillis);
        }

        public void notifyAll(Object toNotify) {
            toNotify.notifyAll();
        }
    }
```

```
    ...}
```

这里我对 RequestFutureTarget 的源码做了一些精简，我们只看最主要的逻辑就可以了。

刚才我们已经学习过了 downloadOnly()方法的基本用法，在调用了 downloadOnly()方法之后，再调用 FutureTarget 的 get()方法，就能获取到下载的图片文件了。而 downloadOnly()方法返回的 FutureTarget 对象其实就是这个 RequestFutureTarget，因此我们直接来看它的 get()方法就行了。

RequestFutureTarget 的 get()方法中又调用了一个 doGet()方法，而 doGet()方法才是真正处理具体逻辑的地方。首先在 doGet()方法中会判断当前是否是在子线程当中，如果不是的话会直接抛出一个异常。然后下面会判断下载是否已取消、或者已失败，如果是已取消或者已失败的话都会直接抛出一个异常。接下来会根据 resultReceived 这个变量来判断下载是否已完成，如果这个变量为 true 的话，就直接把结果进行返回。

那么如果下载还没有完成呢？我们继续往下看，接下来就进入到一个 wait()当中，把当前线程给阻塞住，从而阻止代码继续往下执行。这也是为什么 downloadOnly(int width, int height)方法要求必须在子线程当中使用，因为它会对当前线程进行阻塞，如果在主线程当中使用的话，那么就会让主线程卡死，从而用户无法进行任何其他操作。

那么现在线程被阻塞住了，什么时候才能恢复呢？答案在 onResourceReady()方法中。可以看到，onResourceReady()方法中只有三行代码，第一行把 resultReceived 赋值成 true，说明图片文件已经下载好了，这样下次再调用 get()方法时就不会再阻塞线程，而是可以直接将结果返回。第二行把下载好的图片文件赋值到一个全局的 resource 变量上面，这样 doGet()方法就也可以访问到它。第三行 notifyAll 一下，通知所有 wait 的线程取消阻塞，这个时候图片文件已经下载好了，因此 doGet()方法也就可以返回结果了。

好的，这就是 downloadOnly(int width, int height)方法的基本用法和实现原理，那么下面我们来看一下 downloadOnly(Y target)方法。

回想一下，其实 downloadOnly(int width, int height)方法必须使用在子线程当中，最主要还是因为它在内部帮我们自动创建了一个 RequestFutureTarget，是这个 RequestFutureTarget 要求必须在子线程当中执行。而 downloadOnly(Y target)方法则要求我们传入一个自己创建的 Target，因此就不受 RequestFutureTarget 的限制了。

但是 downloadOnly(Y target)方法的用法也会相对更复杂一些，因为我们又要自己创建一个 Target 了，而且这次必须直接去实现最顶层的 Target 接口，比之前的 SimpleTarget 和 ViewTarget 都要复杂不少。

BAT交流群：892872246

那么下面我们就来实现一个最简单的 DownloadImageTarget 吧，注意 Target 接口的泛型必须指定成 File 对象，这是 downloadOnly(Y target)方法要求的，代码如下所示：

```
public class DownloadImageTarget implements Target<File> {



    private static final String TAG = "DownloadImageTarget";



    @Override

    public void onStart() {

    }



    @Override

    public void onStop() {

    }



    @Override

    public void onDestroy() {

    }



    @Override

    public void onLoadStarted(Drawable placeholder) {

    }



    @Override
```

```
    public void onLoadFailed(Exception e, Drawable errorDrawable) {

    }



    @Override

    public void onResourceReady(File resource, GlideAnimation<? super Fi
le> glideAnimation) {

        Log.d(TAG, resource.getPath());

    }



    @Override

    public void onLoadCleared(Drawable placeholder) {

    }



    @Override

    public void getSize(SizeReadyCallback cb) {

        cb.onSizeReady(Target.SIZE_ORIGINAL, Target.SIZE_ORIGINAL);

    }



    @Override

    public void setRequest(Request request) {

    }



    @Override
```

```
    public Request getRequest() {

        return null;

    }}
```

由于是要直接实现 Target 接口，因此需要重写的方法非常多。这些方法大多是数 Glide 加载图片生命周期的一些回调，我们可以不用管它们，其中只有两个方法是必须实现的，一个是 getSize()方法，一个是 onResourceReady()方法。

在第二篇 Glide 源码解析的时候，我带着大家一起分析过，Glide 在开始加载图片之前会先计算图片的大小，然后回调到 onSizeReady()方法当中，之后才会开始执行图片加载。而这里，计算图片大小的任务就交给我们了。只不过这是一个最简单的 Target 实现，我在 getSize()方法中就直接回调了 Target.SIZE_ORIGINAL，表示图片的原始尺寸。

然后 onResourceReady()方法我们就非常熟悉了，图片下载完成之后就会回调到这里，我在这个方法中只是打印了一下下载的图片文件的路径。

这样一个最简单的 DownloadImageTarget 就定义好了，使用它也非常的简单，我们不用再考虑什么线程的问题了，而是直接把它的实例传入 downloadOnly(Y target)方法中即可，如下所示：

```
public void downloadImage(View view) {

    String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473
_1920x1080.jpg";

    Glide.with(this)

            .load(url)

            .downloadOnly(new DownloadImageTarget());}
```

现在重新运行一下代码并点击 Download Image 按钮，然后观察控制台日志的输出，结果如下图所示。

这样我们就使用了 downloadOnly(Y target)方法同样获取到下载的图片文件的缓存路径了。

BAT交流 372246

好的，那么关于 downloadOnly()方法我们就学到这里。

listener()方法

今天学习的内容已经够多了，下面我们就以一个简单的知识点结尾吧，Glide 回调与监听的最后一部分——listener()方法。

其实 listener()方法的作用非常普遍，它可以用来监听 Glide 加载图片的状态。举个例子，比如说我们刚才使用了 preload()方法来对图片进行预加载，但是我怎样确定预加载有没有完成呢？还有如果 Glide 加载图片失败了，我该怎样调试错误的原因呢？答案都在 listener()方法当中。

首先来看下 listener()方法的基本用法吧，不同于刚才几个方法都是要替换 into()方法的，listener()是结合 into()方法一起使用的，当然也可以结合 preload()方法一起使用。最基本的用法如下所示：

```
public void loadImage(View view) {

    String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473
_1920x1080.jpg";

    Glide.with(this)

            .load(url)

            .listener(new RequestListener<String, GlideDrawable>() {

                @Override

                public boolean onException(Exception e, String model, Tar
get<GlideDrawable> target,

                    boolean isFirstResource) {

                    return false;

                }



                @Override

                public boolean onResourceReady(GlideDrawable resource, St
ring model,
```

BAT文 /2246

```
                    Target<GlideDrawable> target, boolean isFromMemoryCach
e, boolean isFirstResource) {

                    return false;

                }

            })

            .into(imageView);

}
```

这里我们在 into()方法之前串接了一个 listener()方法，然后实现了一个 RequestListener 的实例。其中 RequestListener 需要实现两个方法，一个 onResourceReady()方法，一个 onException()方法。从方法名上就可以看出来了，当图片加载完成的时候就会回调 onResourceReady()方法，而当图片加载失败的时候就会回调 onException()方法，onException()方法中会将失败的 Exception 参数传进来，这样我们就可以定位具体失败的原因了。

没错，listener()方法就是这么简单。不过还有一点需要处理，onResourceReady()方法和 onException()方法都有一个布尔值的返回值，返回 false 就表示这个事件没有被处理，还会继续向下传递，返回 true 就表示这个事件已经被处理掉了，从而不会再继续向下传递。举个简单点的例子，如果我们在 RequestListener 的 onResourceReady()方法中返回了 true，那么就不会再回调 Target 的 onResourceReady()方法了。

关于 listener()方法的用法就讲这么多，不过还是老规矩，我们再来看一下它的源码是怎么实现的吧。

首先，listener()方法是定义在 GenericRequestBuilder 类当中的，而我们传入到 listener()方法中的实例则会赋值到一个 requestListener 变量当中，如下所示：

```
public class GenericRequestBuilder<ModelType, DataType, ResourceType, Tr
anscodeType> implements Cloneable {



    private RequestListener<? super ModelType, TranscodeType> requestLis
tener;

    ...
```

```
    public GenericRequestBuilder<ModelType, DataType, ResourceType, Tran
scodeType> listener(

            RequestListener<? super ModelType, TranscodeType> requestList
ener) {

        this.requestListener = requestListener;

        return this;

    }



    ...}
```

接下来在构建 GenericRequest 的时候这个变量也会被一起传进去，最后在图片加载完成的时候，我们会看到如下逻辑：

```
public final class GenericRequest<A, T, Z, R> implements Request, SizeRe
adyCallback,

        ResourceCallback {



    private RequestListener<? super A, R> requestListener;

    ...



    private void onResourceReady(Resource<?> resource, R result) {

        boolean isFirstResource = isFirstReadyResource();

        status = Status.COMPLETE;

        this.resource = resource;

        if (requestListener == null || !requestListener.onResourceReady(r
esult, model, target,

                loadedFromMemoryCache, isFirstResource)) {
```

```
            GlideAnimation<R> animation = animationFactory.build(loadedFr
omMemoryCache, isFirstResource);

            target.onResourceReady(result, animation);

        }

        notifyLoadSuccess();

    }

    ...}
```

可以看到，这里在第 11 行会先回调 requestListener 的 onResourceReady()方法，只有当这个 onResourceReady()方法返回 false 的时候，才会继续调用 Target 的 onResourceReady()方法，这也就是 listener()方法的实现原理。

另外一个 onException()方法的实现机制也是一模一样的，代码同样是在 GenericRequest 类中，如下所示：

```
public final class GenericRequest<A, T, Z, R> implements Request, SizeRe
adyCallback,

        ResourceCallback {

    ...



    @Override

    public void onException(Exception e) {

        status = Status.FAILED;

        if (requestListener == null ||

                !requestListener.onException(e, model, target, isFirstRea
dyResource())) {

            setErrorPlaceholder(e);

        }

    }
```

```
    ...}
```

可以看到，这里会在第 9 行回调 requestListener 的 onException()方法，只有在 onException()方法返回 false 的情况下才会继续调用 setErrorPlaceholder()方法。也就是说，如果我们在 onException()方法中返回了 true，那么 Glide 请求中使用 error(int resourceId)方法设置的异常占位图就失效了。

这样我们也就将 listener()方法的全部实现原理都分析完了。

4.2.5 Glide 强大的图片变换功能

Glide 的这个框架的功能实在是太强大了，它所能做的事情远远不止于目前我们所学的这些。因此，今天我们就再来学习一个新的功能模块，并且是一个非常重要的模块——Glide 的图片变化功能。

一个问题

在正式开始学习 Glide 的图片变化功能之前，我们先来看一个问题，这个问题可能有不少人都在使用 Glide 的时候都遇到过，正好在本篇内容的主题之下我们顺带着将这个问题给解决了。

首先我们尝试使用 Glide 来加载一张图片，图片 URL 地址是：

```
https://www.baidu.com/img/bd_logo1.png
```

这是百度首页 logo 的一张图片，图片尺寸是 540*258 像素。

接下来我们编写一个非常简单的布局文件，如下所示：

```
<LinearLayout

    xmlns:android="http://schemas.android.com/apk/res/android"

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:orientation="vertical">
```

```
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Load Image"
        android:onClick="loadImage"
        />

    <ImageView
        android:id="@+id/image_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        /></LinearLayout>
```

布局文件中只有一个按钮和一个用于显示图片的 ImageView。注意，ImageView 的宽和高这里设置的都是 wrap_content。

然后编写如下的代码来加载图片：

```
public class MainActivity extends AppCompatActivity {

    ImageView imageView;

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

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

        imageView = (ImageView) findViewById(R.id.image_view);

    }



    public void loadImage(View view) {

        String url = "https://www.baidu.com/img/bd_logo1.png";

        Glide.with(this)

            .load(url)

            .into(imageView);

    }}
```

这些简单的代码对于现在的你而言应该都是小儿科了，相信我也不用再做什么解释。现在运行一下程序并点击加载图片按钮，效果如下图所示。

BAT交流群：892872246

图片是正常加载出来了，不过大家有没有发现一个问题。百度这张 logo 图片的尺寸只有 540*258 像素，但是我的手机的分辨率却是 1080*1920 像素，而我们将 ImageView 的宽高设置的都是 wrap_content，那么图片的宽度应该只有手机屏幕宽度的一半而已，但是这里却充满了全屏，这是为什么呢？

如果你之前也被这个问题困扰过，那么恭喜，本篇文章正是你所需要的。之所以会出现这个现象，就是因为 Glide 的图片变换功能所导致的。那么接下来我们会先分析如何解决这个问题，然后再深入学习 Glide 图片变化的更多功能。

稍微对 Android 有点了解的人应该都知道 ImageView 有 scaleType 这个属性，但是可能大多数人却不知道，如果在没有指定 scaleType 属性的情况下，ImageView 默认的 scaleType 是什么？

这个问题如果直接问我，我也答不上来。不过动手才是检验真理的唯一标准，想知道答案，自己动手试一下就知道了。

```
public class MainActivity extends AppCompatActivity {
```

BAT交流群：892872246

```
private static final String TAG = "MainActivity";

ImageView imageView;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    imageView = (ImageView) findViewById(R.id.image_view);
    Log.d(TAG, "imageView scaleType is " + imageView.getScaleType());
}

...}
```

可以看到，我们在 onCreate()方法中打印了 ImageView 默认的 scaleType，然后重新运行一下程序，结果如下图所示：

```
D/MainActivity: imageView scaleType is FIT_CENTER
```

由此我们可以得知，在没有明确指定的情况下，ImageView 默认的 scaleType 是 FIT_CENTER。

有了这个前提条件，我们就可以继续去分析 Glide 的源码了。当然，本文中的源码还是建在第二篇源码分析的基础之上，还没有看过这篇文章的朋友，建议先去阅读 Android 图片加载框架最全解析（二），从源码的角度理解 Glide 的执行流程 。

BAT交流群：892872246

回顾一下第二篇文章中我们分析过的 into()方法，它是在 GenericRequestBuilder 类当中的，代码如下所示：

```
public Target<TranscodeType> into(ImageView view) {

    Util.assertMainThread();

    if (view == null) {

        throw new IllegalArgumentException("You must pass in a non null V
iew");

    }

    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));}
```

还记得我们当初分析这段代码的时候，直接跳过前面的所有代码，直奔最后一行。因为那个时候我们的主要任务是分析 Glide 的主线执行流程，而不去仔细阅读它的细节，但是现在我们是时候应该阅读一下细节了。

可以看到，这里在第 7 行会进行一个 switch 判断，如果 ImageView 的 scaleType 是 CENTER_CROP，则会去调用 applyCenterCrop()方法，如果 scaleType 是 FIT_CENTER、FIT_START 或 FIT_END，则会去调用 applyFitCenter()方法。这里的 applyCenterCrop()和 applyFitCenter()方法其实就是向 Glide 的加载流程中添加了一个图片变换操作，具体的源码我们就不跟进去看了。

那么现在我们就基本清楚了，由于 ImageView 默认的 scaleType 是 FIT_CENTER，因此会自动添加一个 FitCenter 的图片变换，而在这个图片变换过程中做了某些操作，导致图片充满了全屏。

那么我们该如何解决这个问题呢？最直白的一种办法就是看着源码来改。当 ImageView 的 scaleType 是 CENTER_CROP、FIT_CENTER、FIT_START 或 FIT_END 时不是会自动添加一个图片变换操作吗？那我们把 scaleType 改成其他值不就可以了。ImageView 的 scaleType 可选值还有 CENTER、CENTER_INSIDE、FIT_XY 等。这当然是一种解决方案，不过只能说是一种比较笨的解决方案，因为我们为了解决这个问题而去改动了 ImageView 原有的 scaleType，那如果你真的需要 ImageView 的 scaleType 为 CENTER_CROP 或 FIT_CENTER 时可能就傻眼了。

上面只是我们通过分析源码得到的一种解决方案，并不推荐大家使用。实际上，Glide 给我们提供了专门的 API 来添加和取消图片变换，想要解决这个问题只需要使用如下代码即可：

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

可以看到，这里调用了一个 dontTransform()方法，表示让 Glide 在加载图片的过程中不进行图片变换，这样刚才调用的 applyCenterCrop()、applyFitCenter()就统统无效了。

现在我们重新运行一下代码，效果如下图所示：

BAT交流群：892872246

这样图片就只会占据半个屏幕的宽度了，说明我们的代码奏效了。

但是使用 dontTransform()方法存在着一个问题，就是调用这个方法之后，所有的图片变换操作就全部失效了，那如果我有一些图片变换操作是必须要执行的该怎么办呢？不用担心，总归是有办法的，这种情况下我们只需要借助 override()方法强制将图片尺寸指定成原始大小就可以了，代码如下所示：

```
Glide.with(this)
    .load(url)
    .override(Target.SIZE_ORIGINAL, Target.SIZE_ORIGINAL)
    .into(imageView);
```

通过 override()方法将图片的宽和高都指定成 Target.SIZE_ORIGINAL，问题同样被解决了。程序的最终运行结果和上图是完全一样的，我就不再重新截图了。

BAT交流群892246

由此我们可以看出，之所以会出现这个问题，和 Glide 的图片变换功能是撇不开关系的。那么也是通过这个问题，我们对 Glide 的图片变换有了一个最基本的认识。接下来，就让我们正式开始进入本篇文章的正题吧。

图片变换的基本用法

顾名思义，图片变换的意思就是说，Glide 从加载了原始图片到最终展示给用户之前，又进行了一些变换处理，从而能够实现一些更加丰富的图片效果，如图片圆角化、圆形化、模糊化等等。

添加图片变换的用法非常简单，我们只需要调用 transform()方法，并将想要执行的图片变换操作作为参数传入 transform()方法即可，如下所示：

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

至于具体要进行什么样的图片变换操作，这个通常都是需要我们自己来写的。不过 Glide 已经内置了两种图片变换操作，我们可以直接拿来使用，一个是 CenterCrop，一个是 FitCenter。

但这两种内置的图片变换操作其实都不需要使用 transform()方法，Glide 为了方便我们使用直接提供了现成的 API：

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


Glide.with(this)
    .load(url)
    .fitCenter()
```

BAT /2246

```
    .into(imageView);
```

当然，centerCrop()和 fitCenter()方法其实也只是对 transform()方法进行了一层封装而已，它们背后的源码仍然还是借助 transform()方法来实现的，如下所示：

```
public class DrawableRequestBuilder<ModelType>

        extends GenericRequestBuilder<ModelType, ImageVideoWrapper, GifB
itmapWrapper, GlideDrawable>

        implements BitmapOptions, DrawableOptions {

    ...



    /**

     * Transform {@link GlideDrawable}s using {@link com.bumptech.glide.
load.resource.bitmap.CenterCrop}.

     *

     * @see #fitCenter()

     * @see #transform(BitmapTransformation...)

     * @see #bitmapTransform(Transformation[])

     * @see #transform(Transformation[])

     *

     * @return This request builder.

     */

    @SuppressWarnings("unchecked")

    public DrawableRequestBuilder<ModelType> centerCrop() {

        return transform(glide.getDrawableCenterCrop());
```

```
    }



    /**

     * Transform {@link GlideDrawable}s using {@link com.bumptech.glide.
load.resource.bitmap.FitCenter}.

     *

     * @see #centerCrop()

     * @see #transform(BitmapTransformation...)

     * @see #bitmapTransform(Transformation[])

     * @see #transform(Transformation[])

     *

     * @return This request builder.

     */

    @SuppressWarnings("unchecked")

    public DrawableRequestBuilder<ModelType> fitCenter() {

        return transform(glide.getDrawableFitCenter());

    }



    ...}
```

那么这两种内置的图片变换操作到底能实现什么样的效果呢？FitCenter 的效果其实刚才我们已经见识过了，就是会将图片按照原始的长宽比充满全屏。那么 CenterCrop 又是什么样的效果呢？我们来动手试一下就知道了。

为了让效果更加明显，这里我就不使用百度首页的 Logo 图了，而是换成必应首页的一张美图。在不应用任何图片变换的情况下，使用 Glide 加载必应这张图片效果如下所示。

BAT交流群：892872246

现在我们添加一个 CenterCrop 的图片变换操作，代码如下：

```
String url = "http://cn.bing.com/az/hprichbg/rb/AvalancheCreek_ROW11173354624_1920x1080.jpg";

Glide.with(this)

    .load(url)

    .centerCrop()

    .into(imageView);
```

BAT交流群：892872246

可以看到，现在展示的图片是对原图的中心区域进行裁剪后得到的图片。

另外，centerCrop()方法还可以配合 override()方法来实现更加丰富的效果，比如指定图片裁剪的比例：

```
String url = "http://cn.bing.com/az/hprichbg/rb/AvalancheCreek_ROW11173354624_1920x1080.jpg";

Glide.with(this)

    .load(url)

    .override(500, 500)

    .centerCrop()

    .into(imageView);
```

BA 892246

可以看到，这里我们将图片的尺寸设定为 500*500 像素，那么裁剪的比例也就变成 1：1 了，现在重新运行一下程序，效果如下图所示。、

这样我们就把 Glide 内置的图片变换接口的用法都掌握了。不过不得不说，Glide 内置的图片变换接口功能十分单一且有限，完全没有办法满足我们平时的开发需求。因此，掌握自定义图片变换功能就显得尤为重要了。

不过，在正式开始学习自定义图片变换功能之前，我们先来探究一下 CenterCrop 这种图片变换的源码，理解了它的源码我们再来进行自定义图片变换就能更加得心应手了。

源码分析

那么就话不多说，我们直接打开 CenterCrop 类来看一下它的源码吧，如下所示：

```
public class CenterCrop extends BitmapTransformation {
```

资料：892872246

```
    public CenterCrop(Context context) {

        super(context);

    }



    public CenterCrop(BitmapPool bitmapPool) {

        super(bitmapPool);

    }



    // Bitmap doesn't implement equals, so == and .equals are equivalent
here.

    @SuppressWarnings("PMD.CompareObjectsWithEquals")

    @Override

    protected Bitmap transform(BitmapPool pool, Bitmap toTransform, int
outWidth, int outHeight) {

        final Bitmap toReuse = pool.get(outWidth, outHeight, toTransform.
getConfig() != null

                ? toTransform.getConfig() : Bitmap.Config.ARGB_8888);

        Bitmap transformed = TransformationUtils.centerCrop(toReuse, toTr
ansform, outWidth, outHeight);

        if (toReuse != null && toReuse != transformed && !pool.put(toReus
e)) {

            toReuse.recycle();

        }

        return transformed;

    }
```

```
    @Override

    public String getId() {

        return "CenterCrop.com.bumptech.glide.load.resource.bitmap";

    }}
```

这段代码并不长，但是我还是要划下重点，这样大家看起来的时候会更加轻松。

首先，CenterCrop 是继承自 BitmapTransformation 的，这个是重中之重，因为整个图片变换功能都是建立在这个继承结构基础上的。

接下来 CenterCrop 中最重要的就是 transform()方法，其他的方法我们可以暂时忽略。transform()方法中有四个参数，每一个都很重要，我们来一一解读下。第一个参数 pool，这个是 Glide 中的一个 Bitmap 缓存池，用于对 Bitmap 对象进行重用，否则每次图片变换都重新创建 Bitmap 对象将会非常消耗内存。第二个参数 toTransform，这个是原始图片的 Bitmap 对象，我们就是要对它来进行图片变换。第三和第四个参数比较简单，分别代表图片变换后的宽度和高度，其实也就是 override()方法中传入的宽和高的值了。

下面我们来看一下 transform()方法的细节，首先第一行就从 Bitmap 缓存池中尝试获取一个可重用的 Bitmap 对象，然后把这个对象连同 toTransform、outWidth、outHeight 参数一起传入到了 TransformationUtils.centerCrop()方法当中。那么我们就跟进去来看一下这个方法的源码，如下所示：

```
public final class TransformationUtils {

    ...



    public static Bitmap centerCrop(Bitmap recycled, Bitmap toCrop, int w
idth, int height) {

        if (toCrop == null) {

            return null;

        } else if (toCrop.getWidth() == width && toCrop.getHeight() == he
ight) {

            return toCrop;
```

```
        }

        // From ImageView/Bitmap.createScaledBitmap.

        final float scale;

        float dx = 0, dy = 0;

        Matrix m = new Matrix();

        if (toCrop.getWidth() * height > width * toCrop.getHeight()) {

            scale = (float) height / (float) toCrop.getHeight();

            dx = (width - toCrop.getWidth() * scale) * 0.5f;

        } else {

            scale = (float) width / (float) toCrop.getWidth();

            dy = (height - toCrop.getHeight() * scale) * 0.5f;

        }

        m.setScale(scale, scale);

        m.postTranslate((int) (dx + 0.5f), (int) (dy + 0.5f));



        final Bitmap result;

        if (recycled != null) {

            result = recycled;

        } else {

            result = Bitmap.createBitmap(width, height, getSafeConfig(toC
rop));

        }
```

```
        // We don't add or remove alpha, so keep the alpha setting of the
Bitmap we were given.

        TransformationUtils.setAlpha(toCrop, result);



        Canvas canvas = new Canvas(result);

        Paint paint = new Paint(PAINT_FLAGS);

        canvas.drawBitmap(toCrop, m, paint);

        return result;

    }



    ...}
```

这段代码就是整个图片变换功能的核心代码了。可以看到，第 5-9 行主要是先做了一些校验，如果原图为空，或者原图的尺寸和目标裁剪尺寸相同，那么就放弃裁剪。接下来第 11-22 行是通过数学计算来算出画布的缩放的比例以及偏移值。第 24-29 行是判断缓存池中取出的 Bitmap 对象是否为空，如果不为空就可以直接使用，如果为空则要创建一个新的 Bitmap 对象。第 32 行是将原图 Bitmap 对象的 alpha 值复制到裁剪 Bitmap 对象上面。最后第 34-37 行是裁剪 Bitmap 对象进行绘制，并将最终的结果进行返回。全部的逻辑就是这样，总体来说还是比较简单的，可能也就是数学计算那边需要稍微动下脑筋。

那么现在得到了裁剪后的 Bitmap 对象，我们再回到 CenterCrop 当中，你会看到，在最终返回这个 Bitmap 对象之前，还会尝试将复用的 Bitmap 对象重新放回到缓存池当中，以便下次继续使用。

好的，这样我们就将 CenterCrop 图片变换的工作原理完整地分析了一遍，FitCenter 的源码也是基本类似的，这里就不再重复分析了。了解了这些内容之后，接下来我们就可以开始学习自定义图片变换功能了。

自定义图片变换

Glide 给我们定制好了一个图片变换的框架，大致的流程是我们可以获取到原始的图片，然后对图片进行变换，再将变换完成后的图片返回给 Glide，最终由 Glide 将图片显示出来。理论上，在对图片进行变换这个步骤中我们可以进行任何的操

BAT交流群：892872246

作，你想对图片怎么样都可以。包括圆角化、圆形化、黑白化、模糊化等等，甚至你将原图片完全替换成另外一张图都是可以的。

但是这里显然我不可能向大家演示所有图片变换的可能，图片变换的可能性也是无限的。因此这里我们就选择一种常用的图片变换效果来进行自定义吧——对图片进行圆形化变换。

图片圆形化的功能现在在手机应用中非常常见，比如手机 QQ 就会将用户的头像进行圆形化变换，从而使得界面变得更加好看。

自定义图片变换功能的实现逻辑比较固定，我们刚才看过 CenterCrop 的源码之后，相信你已经基本了解整个自定义的过程了。其实就是自定义一个类让它继承自 BitmapTransformation ，然后重写 transform()方法，并在这里去实现具体的图片变换逻辑就可以了。一个空的图片变换实现大概如下所示：

```
public class CircleCrop extends BitmapTransformation {



    public CircleCrop(Context context) {

        super(context);

    }



    public CircleCrop(BitmapPool bitmapPool) {

        super(bitmapPool);

    }



    @Override

    public String getId() {

        return "com.example.glidetest.CircleCrop";

    }
```

BA... 2246

```
    @Override

    protected Bitmap transform(BitmapPool pool, Bitmap toTransform, int
outWidth, int outHeight) {

        return null;

    }}
```

这里有一点需要注意，就是 getId()方法中要求返回一个唯一的字符串来作为 id，以和其他的图片变换做区分。通常情况下，我们直接返回当前类的完整类名就可以了。

另外，这里我们选择继承 BitmapTransformation 还有一个限制，就是只能对静态图进行图片变换。当然，这已经足够覆盖日常 95%以上的开发需求了。如果你有特殊的需求要对 GIF 图进行图片变换，那就得去自己实现 Transformation 接口才可以了。不过这个就非常复杂了，不在我们今天的讨论范围。

好了，那么我们继续实现对图片进行圆形化变换的功能，接下来只需要在 transform()方法中去做具体的逻辑实现就可以了，代码如下所示：

```
public class CircleCrop extends BitmapTransformation {



    public CircleCrop(Context context) {

        super(context);

    }



    public CircleCrop(BitmapPool bitmapPool) {

        super(bitmapPool);

    }



    @Override

    public String getId() {
```

```
        return "com.example.glidetest.CircleCrop";

    }



    @Override

    protected Bitmap transform(BitmapPool pool, Bitmap toTransform, int
outWidth, int outHeight) {

        int diameter = Math.min(toTransform.getWidth(), toTransform.getHe
ight());



        final Bitmap toReuse = pool.get(outWidth, outHeight, Bitmap.Confi
g.ARGB_8888);

        final Bitmap result;

        if (toReuse != null) {

            result = toReuse;

        } else {

            result = Bitmap.createBitmap(diameter, diameter, Bitmap.Confi
g.ARGB_8888);

        }



        int dx = (toTransform.getWidth() - diameter) / 2;

        int dy = (toTransform.getHeight() - diameter) / 2;

        Canvas canvas = new Canvas(result);

        Paint paint = new Paint();

        BitmapShader shader = new BitmapShader(toTransform, BitmapShader.
TileMode.CLAMP,
```

```
                                        BitmapShader.TileMode.CLAMP);

        if (dx != 0 || dy != 0) {

            Matrix matrix = new Matrix();

            matrix.setTranslate(-dx, -dy);

            shader.setLocalMatrix(matrix);

        }

        paint.setShader(shader);

        paint.setAntiAlias(true);

        float radius = diameter / 2f;

        canvas.drawCircle(radius, radius, radius, paint);



        if (toReuse != null && !pool.put(toReuse)) {

            toReuse.recycle();

        }

        return result;

    }}
```

下面我来对 transform()方法中的逻辑做下简单的解释。首先第 18 行先算出原图宽度和高度中较小的值，因为对图片进行圆形化变换肯定要以较小的那个值作为直径来进行裁剪。第 20-26 行则和刚才一样，从 Bitmap 缓存池中尝试获取一个 Bitmap 对象来进行重用，如果没有可重用的 Bitmap 对象的话就创建一个。第 28-41 行是具体进行圆形化变换的部分，这里算出了画布的偏移值，并且根据刚才得到的直径算出半径来进行画圆。最后，尝试将复用的 Bitmap 对象重新放回到缓存池当中，并将圆形化变换后的 Bitmap 对象进行返回。

这样，一个自定义图片变换的功能就写好了，那么现在我们就来尝试使用一下它吧。使用方法非常简单，刚才已经介绍过了，就是把这个自定义图片变换的实例传入到 transform()方法中即可，如下所示：

```
Glide.with(this)
```

IT交流群：892872246

```
    .load(url)

    .transform(new CircleCrop(this))

    .into(imageView);
```

现在我们重新运行一下程序，效果如下图所示。

更多图片变换功能

虽说 Glide 的图片变换功能框架已经很强大了，使得我们可以轻松地自定义图片变换效果，但是如果每一种图片变换都要我们自己去写还是蛮吃力的。事实上，确实也没有必要完全靠自己去实现各种各样的图片变换效果，因为大多数的图片变换都是比较通用的，各个项目会用到的效果都差不多，我们每一个都自己去重新实现无异于重复造轮子。

也正是因此，网上出现了很多 Glide 的图片变换开源库，其中做的最出色的应该要数 glide-transformations 这个库了。它实现了很多通用的图片变换效果，如裁

BAT交流群：892872246

剪变换、颜色变换、模糊变换等等，使得我们可以非常轻松地进行各种各样的图片变换。

glide-transformations 的项目主页地址是 https://github.com/wasabeef/glide-transformations 。

下面我们就来体验一下这个库的强大功能吧。首先需要将这个库引入到我们的项目当中，在 app/build.gradle 文件当中添加如下依赖：

```
dependencies {

    compile 'jp.wasabeef:glide-transformations:2.0.2'

}
```

现在如果我想对图片进行模糊化处理，那么就可以使用 glide-transformations

库中的 BlurTransformation 这个类，代码如下所示：

```
Glide.with(this)

     .load(url)

     .bitmapTransform(new BlurTransformation(this))

     .into(imageView);
```

注意这里我们调用的是 bitmapTransform()方法而不是 transform()方法，因为 glide-transformations 库都是专门针对静态图片变换来进行设计的。现在重新运行一下程度，效果如下图所示。

BAT交流群：892872246

没错，我们就这样轻松地实现模糊化的效果了。

接下来我们再试一下图片黑白化的效果，使用的是 GrayscaleTransformation 这个类，代码如下所示：

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

现在重新运行一下程度，效果如下图所示。

BAT交流群：892872246

而且我们还可以将多个图片变换效果组合在一起使用，比如同时执行模糊化和黑白化的变换：

```
Glide.with(this)

    .load(url)

    .bitmapTransform(new BlurTransformation(this), new GrayscaleTransformation(this))

    .into(imageView);
```

可以看到，同时执行多种图片变换的时候，只需要将它们都传入到bitmapTransform()方法中即可。现在重新运行一下程序，效果如下图所示。

BAT交流群：892872246

当然，这些只是 glide-transformations 库的一小部分功能而已，更多的图片变换效果你可以到它的 GitHub 项目主页去学习，所有变换的用法都是这么简单哦。

4.2.6 探究 Glide 的自定义模块功能

自定义模块的基本用法

学到这里相信你已经知道，Glide 的用法是非常非常简单的，大多数情况下，我们想要实现的图片加载效果只需要一行代码就能解决了。但是 Glide 过于简洁的 API 也造成了一个问题，就是如果我们想要更改 Glide 的某些默认配置项应该怎么操作呢？很难想象如何将更改 Glide 配置项的操作串联到一行经典的 Glide 图片加载语句中当中吧？没错，这个时候就需要用到自定义模块功能了。

自定义模块功能可以将更改 Glide 配置，替换 Glide 组件等操作独立出来，使得我们能轻松地对 Glide 的各种配置进行自定义，并且又和 Glide 的图片加载逻辑

BAT交流群：892872246

没有任何交集，这也是一种低耦合编程方式的体现。那么接下来我们就学习一下自定义模块的基本用法。

首先需要定义一个我们自己的模块类，并让它实现 GlideModule 接口，如下所示：

```
public class MyGlideModule implements GlideModule {

    @Override

    public void applyOptions(Context context, GlideBuilder builder) {

    }



    @Override

    public void registerComponents(Context context, Glide glide) {

    }}
```

可以看到，在 MyGlideModule 类当中，我们重写了 applyOptions()和 registerComponents()方法，这两个方法分别就是用来更改 Glide 和配置以及替换 Glide 组件的。我们待会儿只需要在这两个方法中加入具体的逻辑，就能实现更改 Glide 配置或者替换 Glide 组件的功能了。

不过，目前 Glide 还无法识别我们自定义的 MyGlideModule，如果想要让它生效，还得在 AndroidManifest.xml 文件当中加入如下配置才行：

```
<manifest>



    ...



    <application>



        <meta-data

            android:name="com.example.glidetest.MyGlideModule"
```

BA 892246

```
        android:value="GlideModule" />

    ...

</application></manifest>
```

在<application>标签中加入一个 meta-data 配置项，其中 android:name 指定成我们自定义的 MyGlideModule 的完整路径，android:value 必须指定成 GlideModule，这个是固定值。

这样的话，我们就将 Glide 自定义模块的功能完成了，是不是非常简单？现在 Glide 已经能够识别我们自定义的这个 MyGlideModule 了，但是在编写具体的功能之前，我们还是按照老规矩阅读一下源码，从源码的层面上来分析一下，Glide 到底是如何识别出这个自定义的 MyGlideModule 的。

自定义模块的原理

这里我不会带着大家从 Glide 代码执行的第一步一行行重头去解析 Glide 的源码，而是只分析和自定义模块相关的部分。如果你想将 Glide 的源码通读一遍的话，可以去看本系列的第二篇文章 Android 图片加载框架最全解析（二），从源码的角度理解 Glide 的执行流程 。

显然我们已经用惯了 Glide.with(context).load(url).into(imageView)这样一行简洁的 Glide 图片加载语句，但是我们好像从来没有注意过 Glide 这个类本身的实例。然而事实上，Glide 类确实是有创建实例的，只不过是在内部由 Glide 自动帮我们创建和管理了，对于开发者而言，大多数情况下是不用关心它的，只需要调用它的静态方法就可以了。

那么 Glide 的实例到底是在哪里创建的呢？我们来看下 Glide 类中的 get()方法的源码，如下所示：

```
public class Glide {

    private static volatile Glide glide;

    ...
```

```
    public static Glide get(Context context) {
        if (glide == null) {
            synchronized (Glide.class) {
                if (glide == null) {
                    Context applicationContext = context.getApplicationCon
text();
                    List<GlideModule> modules = new ManifestParser(applica
tionContext).parse();
                    GlideBuilder builder = new GlideBuilder(applicationCon
text);
                    for (GlideModule module : modules) {
                        module.applyOptions(applicationContext, builder);
                    }
                    glide = builder.createGlide();
                    for (GlideModule module : modules) {
                        module.registerComponents(applicationContext, glid
e);
                    }
                }
            }
        }
        return glide;
    }
```

```
    ...}
```

我们来仔细看一下上面这段代码。首先这里使用了一个单例模式来获取 Glide 对象的实例，可以看到，这是一个非常典型的双重锁模式。然后在第 12 行，调用 ManifestParser 的 parse()方法去解析 AndroidManifest.xml 文件中的配置，实际上就是将 AndroidManifest 中所有值为 GlideModule 的 meta-data 配置读取出来，并将相应的自定义模块实例化。由于你可以自定义任意多个模块，因此这里我们将会得到一个 GlideModule 的 List 集合。

接下来在第 13 行创建了一个 GlideBuilder 对象，并通过一个循环调用了每一个 GlideModule 的 applyOptions()方法，同时也把 GlideBuilder 对象作为参数传入到这个方法中。而 applyOptions()方法就是我们可以加入自己的逻辑的地方了，虽然目前为止我们还没有编写任何逻辑。

再往下的一步就非常关键了，这里调用了 GlideBuilder 的 createGlide()方法，并返回了一个 Glide 对象。也就是说，Glide 对象的实例就是在这里创建的了，那么我们跟到这个方法当中瞧一瞧：

```
public class GlideBuilder {

    private final Context context;



    private Engine engine;

    private BitmapPool bitmapPool;

    private MemoryCache memoryCache;

    private ExecutorService sourceService;

    private ExecutorService diskCacheService;

    private DecodeFormat decodeFormat;

    private DiskCache.Factory diskCacheFactory;



    ...



    Glide createGlide() {
```

```
        if (sourceService == null) {
            final int cores = Math.max(1, Runtime.getRuntime().availableP
rocessors());
            sourceService = new FifoPriorityThreadPoolExecutor(cores);
        }
        if (diskCacheService == null) {
            diskCacheService = new FifoPriorityThreadPoolExecutor(1);
        }
        MemorySizeCalculator calculator = new MemorySizeCalculator(contex
t);
        if (bitmapPool == null) {
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
                int size = calculator.getBitmapPoolSize();
                bitmapPool = new LruBitmapPool(size);
            } else {
                bitmapPool = new BitmapPoolAdapter();
            }
        }
        if (memoryCache == null) {
            memoryCache = new LruResourceCache(calculator.getMemoryCacheS
ize());
        }
        if (diskCacheFactory == null) {
            diskCacheFactory = new InternalCacheDiskCacheFactory(contex
t);
```

```
        }

        if (engine == null) {

            engine = new Engine(memoryCache, diskCacheFactory, diskCacheS
ervice, sourceService);

        }

        if (decodeFormat == null) {

            decodeFormat = DecodeFormat.DEFAULT;

        }

        return new Glide(engine, memoryCache, bitmapPool, context, decode
Format);

    }}
```

这个方法中会创建 BitmapPool、MemoryCache、DiskCache、DecodeFormat 等对象的实例，并在最后一行创建一个 Glide 对象的实例，然后将前面创建的这些实例传入到 Glide 对象当中，以供后续的图片加载操作使用。

但是大家有没有注意到一个细节，createGlide()方法中创建任何对象的时候都做了一个空检查，只有在对象为空的时候才会去创建它的实例。也就是说，如果我们可以在 applyOptions()方法中提前就给这些对象初始化并赋值，那么在 createGlide()方法中就不会再去重新创建它们的实例了，从而也就实现了更改 Glide 配置的功能。关于这个功能我们待会儿会进行具体的演示。

现在继续回到 Glide 的 get()方法中，得到了 Glide 对象的实例之后，接下来又通过一个循环调用了每一个 GlideModule 的 registerComponents()方法，在这里我们可以加入替换 Glide 的组件的逻辑。

好了，这就是 Glide 自定义模块的全部工作原理。了解了它的工作原理之后，接下来所有的问题就集中在我们到底如何在 applyOptions()和 registerComponents()这两个方法中加入具体的逻辑了，下面我们马上就来学习一下。

更改 Glide 配置

刚才在分析自定义模式工作原理的时候其实就已经提到了，如果想要更改 Glide 的默认配置，其实只需要在 applyOptions()方法中提前将 Glide 的配置项进行初始化就可以了。那么 Glide 一共有哪些配置项呢？这里我给大家做了一个列举：

BAT交流群：892872246

setMemoryCache()
用于配置 Glide 的内存缓存策略，默认配置是 LruResourceCache。

setBitmapPool()
用于配置 Glide 的 Bitmap 缓存池，默认配置是 LruBitmapPool。

setDiskCache()
用于配置 Glide 的硬盘缓存策略，默认配置是 InternalCacheDiskCacheFactory。

setDiskCacheService()
用于配置 Glide 读取缓存中图片的异步执行器，默认配置是 FifoPriorityThreadPoolExecutor，也就是先入先出原则。

setResizeService()
用于配置 Glide 读取非缓存中图片的异步执行器，默认配置也是 FifoPriorityThreadPoolExecutor。

setDecodeFormat()
用于配置 Glide 加载图片的解码模式，默认配置是 RGB_565。

其实 Glide 的这些默认配置都非常科学且合理，使用的缓存算法也都是效率极高的，因此在绝大多数情况下我们并不需要去修改这些默认配置，这也是 Glide 用法能如此简洁的一个原因。

但是 Glide 科学的默认配置并不影响我们去学习自定义 Glide 模块的功能，因此总有某些情况下，默认的配置可能将无法满足你，这个时候就需要我们自己动手来修改默认配置了。

下面就通过具体的实例来看一下吧。刚才说到，Glide 默认的硬盘缓存策略使用的是 InternalCacheDiskCacheFactory，这种缓存会将所有 Glide 加载的图片都存储到当前应用的私有目录下。这是一种非常安全的做法，但同时这种做法也造成了一些不便，因为私有目录下即使是开发者自己也是无法查看的，如果我想要去验证一下图片到底有没有成功缓存下来，这就有点不太好办了。

这种情况下，就非常适合使用自定义模块来更改 Glide 的默认配置。我们完全可以自己去实现 DiskCache.Factory 接口来自定义一个硬盘缓存策略，不过却大大没有必要这么做，因为 Glide 本身就内置了一个 ExternalCacheDiskCacheFactory，可以允许将加载的图片都缓存到 SD 卡。

那么接下来，我们就尝试使用这个 ExternalCacheDiskCacheFactory 来替换默认的 InternalCacheDiskCacheFactory，从而将所有 Glide 加载的图片都缓存到 SD 卡上。

由于在前面我们已经创建好了一个自定义模块 MyGlideModule，那么现在就可以直接在这里编写逻辑了，代码如下所示：

```
public class MyGlideModule implements GlideModule {
```

AT交流群：892872246

```
    @Override
    public void applyOptions(Context context, GlideBuilder builder) {
        builder.setDiskCache(new ExternalCacheDiskCacheFactory(contex
t));
    }

    @Override
    public void registerComponents(Context context, Glide glide) {

    }
}
```

没错，就是这么简单，现在所有 Glide 加载的图片都会缓存到 SD 卡上了。

另外，InternalCacheDiskCacheFactory 和 ExternalCacheDiskCacheFactory 的默认硬盘缓存大小都是 250M。也就是说，如果你的应用缓存的图片总大小超出了 250M，那么 Glide 就会按照 DiskLruCache 算法的原则来清理缓存的图片。

当然，我们是可以对这个默认的缓存大小进行修改的，而且修改方式非常简单，如下所示：

```
public class MyGlideModule implements GlideModule {

    public static final int DISK_CACHE_SIZE = 500 * 1024 * 1024;

    @Override
    public void applyOptions(Context context, GlideBuilder builder) {
```

BAT交流群892872246

```
        builder.setDiskCache(new ExternalCacheDiskCacheFactory(context,
DISK_CACHE_SIZE));

    }



    @Override

    public void registerComponents(Context context, Glide glide) {



    }

}
```

只需要向 ExternalCacheDiskCacheFactory 或者 InternalCacheDiskCacheFactory 再传入一个参数就可以了，现在我们就将 Glide 硬盘缓存的大小调整成了 500M。

好了，更改 Glide 配置的功能就是这么简单，那么接下来我们就来验证一下更改的配置到底有没有生效吧。

这里还是使用最基本的 Glide 加载语句来去加载一张网络图片：

```
String url = "http://guolin.tech/book.png";

Glide.with(this)

     .load(url)

     .into(imageView);
```

运行一下程序，效果如下图所示：

BAT交流群：892872246

OK，现在图片已经加载出现了，那么我们去找一找它的缓存吧。

ExternalCacheDiskCacheFactory 的默认缓存路径是在 sdcard/Android/包名/cache/image_manager_disk_cache 目录当中，我们使用文件浏览器进入到这个目录，结果如下图所示。

BAT交流群：892872246

可以看到，这里有两个文件，其中 journal 文件是 DiskLruCache 算法的日志文件，这个文件必不可少，且只会有一个。想了解更多关于 DiskLruCache 算法的朋友，可以去阅读我的这篇博客 Android DiskLruCache 完全解析，硬盘缓存的最佳方案 。

而另外一个文件就是那张缓存的图片了，它的文件名虽然看上去很奇怪，但是我们只需要把这个文件的后缀改成.png，然后用图片浏览器打开，结果就一目了然了，如下图所示。

BAT交流群：892872246

由此证明，我们已经成功将 Glide 的硬盘缓存路径修改到 SD 卡上了。

另外这里再提一点，我们都知道 Glide 和 Picasso 的用法是非常相似的，但是有一点差别却很大。Glide 加载图片的默认格式是 RGB_565，而 Picasso 加载图片的默认格式是 ARGB_8888。ARGB_8888 格式的图片效果会更加细腻，但是内存开销会比较大。而 RGB_565 格式的图片则更加节省内存，但是图片效果上会差一些。

Glide 和 Picasso 各自采取的默认图片格式谈不上熟优熟劣，只能说各自的取舍不一样。但是如果你希望 Glide 也能使用 ARGB_8888 的图片格式，这当然也是可以的。我们只需要在 MyGlideModule 中更改一下默认配置即可，如下所示：

```
public class MyGlideModule implements GlideModule {



    public static final int DISK_CACHE_SIZE = 500 * 1024 * 1024;
```

```
    @Override
    public void applyOptions(Context context, GlideBuilder builder) {
        builder.setDiskCache(new ExternalCacheDiskCacheFactory(context,
DISK_CACHE_SIZE));
        builder.setDecodeFormat(DecodeFormat.PREFER_ARGB_8888);
    }

    @Override
    public void registerComponents(Context context, Glide glide) {

    }
}
```

通过这样配置之后，使用 Glide 加载的所有图片都将会使用 ARGB_8888 的格式，虽然图片质量变好了，但同时内存开销也会明显增大，所以你要做好心理准备哦。

好了，关于更改 Glide 配置的内容就介绍这么多，接下来就让我们进入到下一个非常重要的主题，替换 Glide 组件。

替换 Glide 组件

替换 Glide 组件功能需要在自定义模块的 registerComponents()方法中加入具体的替换逻辑。相比于更改 Glide 配置，替换 Glide 组件这个功能的难度就明显大了不少。Glide 中的组件非常繁多，也非常复杂，但其实大多数情况下并不需要我们去做什么替换。不过，有一个组件却有着比较大的替换需求，那就是 Glide 的 HTTP 通讯组件。

默认情况下，Glide 使用的是基于原生 HttpURLConnection 进行订制的 HTTP 通讯组件，但是现在大多数的 Android 开发者都更喜欢使用 OkHttp，因此将 Glide 中的 HTTP 通讯组件修改成 OkHttp 的这个需求比较常见，那么今天我们也会以这个功能来作为例子进行讲解。

BAT交流群：892872246

首先来看一下 Glide 中目前有哪些组件吧，在 Glide 类的构造方法当中，如下所示：

```
public class Glide {



    Glide(Engine engine, MemoryCache memoryCache, BitmapPool bitmapPool,
 Context context, DecodeFormat decodeFormat) {

        ...



        register(File.class, ParcelFileDescriptor.class, new FileDescrip
torFileLoader.Factory());

        register(File.class, InputStream.class, new StreamFileLoader.Fac
tory());

        register(int.class, ParcelFileDescriptor.class, new FileDescript
orResourceLoader.Factory());

        register(int.class, InputStream.class, new StreamResourceLoader.
Factory());

        register(Integer.class, ParcelFileDescriptor.class, new FileDesc
riptorResourceLoader.Factory());

        register(Integer.class, InputStream.class, new StreamResourceLoa
der.Factory());

        register(String.class, ParcelFileDescriptor.class, new FileDescr
iptorStringLoader.Factory());

        register(String.class, InputStream.class, new StreamStringLoader.
Factory());

        register(Uri.class, ParcelFileDescriptor.class, new FileDescript
orUriLoader.Factory());

        register(Uri.class, InputStream.class, new StreamUriLoader.Facto
ry());
```

BAT文 892246

```
        register(URL.class, InputStream.class, new StreamUrlLoader.Facto
ry());

        register(GlideUrl.class, InputStream.class, new HttpUrlGlideUrlL
oader.Factory());

        register(byte[].class, InputStream.class, new StreamByteArrayLoa
der.Factory());



        ...

    }

}
```

可以看到，这里都是以调用 register()方法的方式来注册一个组件，register()方法中传入的参数表示 Glide 支持使用哪种参数类型来加载图片，以及如何去处理这种类型的图片加载。举个例子：

```
register(GlideUrl.class, InputStream.class, new HttpUrlGlideUrlLoader.Fa
ctory());
```

register(GlideUrl.class, InputStream.class, new HttpUrlGlideUrlLoader.Factory());

这句代码就表示，我们可以使用 Glide.with(context).load(new GlideUrl("url...")).into(imageView)的方式来加载图片，而 HttpUrlGlideUrlLoader.Factory 则是要负责处理具体的网络通讯逻辑。如果我们想要将 Glide 的 HTTP 通讯组件替换成 OkHttp 的话，那么只需要在自定义模块当中重新注册一个 GlideUrl 类型的组件就行了。

说到这里有的朋友可能会疑问了，我们平时使用 Glide 加载图片时，大多数情况下都是直接将图片的 URL 字符串传入到 load()方法当中的，很少会将它封装成 GlideUrl 对象之后再传入到 load()方法当中，那为什么只需要重新注册一个 GlideUrl 类型的组件，而不需要去重新注册一个 String 类型的组件呢？其实道理很简单，因为 load(String)方法只是 Glide 给我们提供一种简易的 API 封装而已，它的底层仍然还是调用的 GlideUrl 组件，因此我们在替换组件的时候只需要直接替换最底层的，这样就一步到位了。

那么接下来我们就开始学习到底如何将 Glide 的 HTTP 通讯组件替换成 OkHttp。

BAT交流群 892872246

首先第一步，不用多说，肯定是要先将 OkHttp 的库引入到当前项目中，如下所示：

```
dependencies {

    compile 'com.squareup.okhttp3:okhttp:3.9.0'

}
```

然后接下来该怎么做呢？我们只要依葫芦画瓢就可以了。刚才不是说 Glide 的网络通讯逻辑是由 HttpUrlGlideUrlLoader.Factory 来负责的吗，那么我们就来看一下它的源码：

```
public class HttpUrlGlideUrlLoader implements ModelLoader<GlideUrl, Inpu
tStream> {



    private final ModelCache<GlideUrl, GlideUrl> modelCache;



    public static class Factory implements ModelLoaderFactory<GlideUrl,
InputStream> {

        private final ModelCache<GlideUrl, GlideUrl> modelCache = new Mod
elCache<GlideUrl, GlideUrl>(500);



        @Override

        public ModelLoader<GlideUrl, InputStream> build(Context context,
GenericLoaderFactory factories) {

            return new HttpUrlGlideUrlLoader(modelCache);

        }



        @Override
```

```
        public void teardown() {

        }

    }



    public HttpUrlGlideUrlLoader() {

        this(null);

    }



    public HttpUrlGlideUrlLoader(ModelCache<GlideUrl, GlideUrl> modelCac
he) {

        this.modelCache = modelCache;

    }



    @Override

    public DataFetcher<InputStream> getResourceFetcher(GlideUrl model, i
nt width, int height) {

        GlideUrl url = model;

        if (modelCache != null) {

            url = modelCache.get(model, 0, 0);

            if (url == null) {

                modelCache.put(model, 0, 0, model);

                url = model;

            }

        }
```

```
        return new HttpUrlFetcher(url);

    }}
```

可以看到，HttpUrlGlideUrlLoader.Factory 是一个内部类，外层的 HttpUrlGlideUrlLoader 类实现了 ModelLoader<GlideUrl, InputStream>这个接口，并重写了 getResourceFetcher()方法。而在 getResourceFetcher()方法中，又创建了一个 HttpUrlFetcher 的实例，在这里才是真正处理具体网络通讯逻辑的地方，代码如下所示：

```
public class HttpUrlFetcher implements DataFetcher<InputStream> {

    private static final String TAG = "HttpUrlFetcher";

    private static final int MAXIMUM_REDIRECTS = 5;

    private static final HttpUrlConnectionFactory DEFAULT_CONNECTION_FAC
TORY = new DefaultHttpUrlConnectionFactory();



    private final GlideUrl glideUrl;

    private final HttpUrlConnectionFactory connectionFactory;



    private HttpURLConnection urlConnection;

    private InputStream stream;

    private volatile boolean isCancelled;



    public HttpUrlFetcher(GlideUrl glideUrl) {

        this(glideUrl, DEFAULT_CONNECTION_FACTORY);

    }



    HttpUrlFetcher(GlideUrl glideUrl, HttpUrlConnectionFactory connectio
nFactory) {
```

```
        this.glideUrl = glideUrl;

        this.connectionFactory = connectionFactory;

    }



    @Override

    public InputStream loadData(Priority priority) throws Exception {

        return loadDataWithRedirects(glideUrl.toURL(), 0 , null , glideUr
l.getHeaders());

    }



    private InputStream loadDataWithRedirects(URL url, int redirects, UR
L lastUrl, Map<String, String> headers)

            throws IOException {

        if (redirects >= MAXIMUM_REDIRECTS) {

            throw new IOException("Too many (> " + MAXIMUM_REDIRECTS + ")
 redirects!");

        } else {

            try {

                if (lastUrl != null && url.toURI().equals(lastUrl.toURI
())) {

                    throw new IOException("In re-direct loop");

                }

            } catch (URISyntaxException e) {

            }

        }
```

```
        urlConnection = connectionFactory.build(url);

        for (Map.Entry<String, String> headerEntry : headers.entrySet())
{

          urlConnection.addRequestProperty(headerEntry.getKey(), headerE
ntry.getValue());

        }

        urlConnection.setConnectTimeout(2500);

        urlConnection.setReadTimeout(2500);

        urlConnection.setUseCaches(false);

        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("Loca
tion");

            if (TextUtils.isEmpty(redirectUrlString)) {

                throw new IOException("Received empty or null redirect url
");

            }

            URL redirectUrl = new URL(url, redirectUrlString);
```

BAT交... 892246

```
            return loadDataWithRedirects(redirectUrl, redirects + 1, url,
 headers);

        } else {

            if (statusCode == -1) {

                throw new IOException("Unable to retrieve response code fr
om 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.getInp
utStream(), contentLength);

        } else {

            stream = urlConnection.getInputStream();

        }

        return stream;

    }
```

```
@Override
public void cleanup() {
    if (stream != null) {
        try {
            stream.close();
        } catch (IOException e) {
        }
    }
    if (urlConnection != null) {
        urlConnection.disconnect();
    }
}

@Override
public String getId() {
    return glideUrl.getCacheKey();
}

@Override
public void cancel() {
    isCancelled = true;
}
```

```
    interface HttpUrlConnectionFactory {

        HttpURLConnection build(URL url) throws IOException;

    }



    private static class DefaultHttpUrlConnectionFactory implements Http
UrlConnectionFactory {

        @Override

        public HttpURLConnection build(URL url) throws IOException {

            return (HttpURLConnection) url.openConnection();

        }

    }}
```

上面这段代码看上去应该不费力吧？其实就是一些 HttpURLConnection 的用法而已。那么我们只需要仿照着 HttpUrlFetcher 的代码来写，并且把 HTTP 的通讯组件替换成 OkHttp 就可以了。

现在新建一个 OkHttpFetcher 类，并且同样实现 DataFetcher<InputStream>接口，代码如下所示：

```
public class OkHttpFetcher implements DataFetcher<InputStream> {



    private final OkHttpClient client;

    private final GlideUrl url;

    private InputStream stream;

    private ResponseBody responseBody;

    private volatile boolean isCancelled;



    public OkHttpFetcher(OkHttpClient client, GlideUrl url) {
```

```
        this.client = client;

        this.url = url;

    }



    @Override

    public InputStream loadData(Priority priority) throws Exception {

        Request.Builder requestBuilder = new Request.Builder()

                .url(url.toStringUrl());

        for (Map.Entry<String, String> headerEntry : url.getHeaders().ent
rySet()) {

            String key = headerEntry.getKey();

            requestBuilder.addHeader(key, headerEntry.getValue());

        }

        requestBuilder.addHeader("httplib", "OkHttp");

        Request request = requestBuilder.build();

        if (isCancelled) {

            return null;

        }

        Response response = client.newCall(request).execute();

        responseBody = response.body();

        if (!response.isSuccessful() || responseBody == null) {

            throw new IOException("Request failed with code: " + response.
code());

        }
```

```
        stream = ContentLengthInputStream.obtain(responseBody.byteStream
(),
                responseBody.contentLength());
        return stream;
    }

    @Override
    public void cleanup() {
        try {
            if (stream != null) {
                stream.close();
            }
            if (responseBody != null) {
                responseBody.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String getId() {
        return url.getCacheKey();
```

```
    }



    @Override

    public void cancel() {

        isCancelled = true;

    }}
}
```

上面这段代码完全就是我照着 HttpUrlFetcher 依葫芦画瓢写出来的，用的也都是一些 OkHttp 的基本用法，相信不需要再做什么解释了吧。可以看到，使用 OkHttp 来编写网络通讯的代码要比使用 HttpURLConnection 简单很多，代码行数也少了很多。注意在第 22 行，我添加了一个 httplib: OkHttp 的请求头，这个是待会儿我们用来进行测试验证的，大家实际项目中的代码无须添加这个请求头。

那么我们就继续发挥依葫芦画瓢的精神，仿照着 HttpUrlGlideUrlLoader 再写一个 OkHttpGlideUrlLoader 吧。新建一个 OkHttpGlideUrlLoader 类，并且实现 ModelLoader<GlideUrl, InputStream>接口，代码如下所示：

```
public class OkHttpGlideUrlLoader implements ModelLoader<GlideUrl, Input
Stream> {



    private OkHttpClient okHttpClient;



    public static class Factory implements ModelLoaderFactory<GlideUrl,
InputStream> {



        private OkHttpClient client;



        public Factory() {

        }
```

```
        public Factory(OkHttpClient client) {
            this.client = client;
        }

        private synchronized OkHttpClient getOkHttpClient() {
            if (client == null) {
                client = new OkHttpClient();
            }
            return client;
        }

        @Override
        public ModelLoader<GlideUrl, InputStream> build(Context context,
GenericLoaderFactory factories) {
            return new OkHttpGlideUrlLoader(getOkHttpClient());
        }

        @Override
        public void teardown() {
        }
    }
```

```
    public OkHttpGlideUrlLoader(OkHttpClient client) {

        this.okHttpClient = client;

    }



    @Override

    public DataFetcher<InputStream> getResourceFetcher(GlideUrl model, i
nt width, int height) {

        return new OkHttpFetcher(okHttpClient, model);

    }}
```

注意这里的 Factory 我提供了两个构造方法，一个是不带任何参数的，一个是带 OkHttpClient 参数的。如果对 OkHttp 不需要进行任何自定义的配置，那么就调用无参的 Factory 构造函数即可，这样会在内部自动创建一个 OkHttpClient 实例。但如果你需要想添加拦截器，或者修改 OkHttp 的默认超时等等配置，那么就自己创建一个 OkHttpClient 的实例，然后传入到 Factory 的构造方法当中就行了。

好了，现在就只差最后一步，将我们刚刚创建的 OkHttpGlideUrlLoader 和 OkHttpFetcher 注册到 Glide 当中，将原来的 HTTP 通讯组件给替换掉，如下所示：

```
public class MyGlideModule implements GlideModule {



    ...



    @Override

    public void registerComponents(Context context, Glide glide) {

        glide.register(GlideUrl.class, InputStream.class, new OkHttpGlid
eUrlLoader.Factory());

    }

}
```

可以看到，这里也是调用了 Glide 的 register()方法来注册组件的。register()方法中使用的 Map 类型来存储已注册的组件，因此我们这里重新注册了一遍 GlideUrl.class 类型的组件，就把原来的组件给替换掉了。

理论上来说，现在我们已经成功将 Glide 的 HTTP 通讯组件替换成 OkHttp 了，现在唯一的问题就是我们该如何去验证一下到底有没有替换成功呢？

验证的方式我倒是想了很多种，比如添加 OkHttp 拦截器，或者自己架设一个测试用的服务器都是可以的。不过为了让大家最直接地看到验证结果，这里我准备使用 Fiddler 这个抓包工具来进行验证。这个工具的用法非常简单，但是限于篇幅我就不在本篇文章中介绍这个工具的用法了，还没用过这个工具的朋友们可以通过 这篇文章 了解一下。

在开始验证之前，我们还得要再修改一下 Glide 加载图片的代码才行，如下所示：

```
String url = "http://guolin.tech/book.png";

Glide.with(this)

    .load(url)

    .skipMemoryCache(true)

    .diskCacheStrategy(DiskCacheStrategy.NONE)

    .into(imageView);
```

这里我把 Glide 的内存缓存和硬盘缓存都禁用掉了，不然的话，Glide 可能会直接读取刚才缓存的图片，而不会再重新发起网终请求。

好的，现在我们重新使用 Glide 加载一下图片，然后观察 Fiddler 中的抓包情况，如下图所示。

92872246
BAT交

可以看到，在 HTTP 请求头中确实有我们刚才自己添加的 httplib: OkHttp。也就说明，Glide 的 HTTP 通讯组件的确被替换成功了。

更简单的组件替换

上述方法是我们纯手工地将 Glide 的 HTTP 通讯组件进行了替换，如果你不想这么麻烦也是可以的，Glide 官方给我们提供了非常简便的 HTTP 组件替换方式。并且除了支持 OkHttp3 之外，还支持 OkHttp2 和 Volley。

我们只需要在 gradle 当中添加几行库的配置就行了。比如使用 OkHttp3 来作为 HTTP 通讯组件的配置如下：

```
dependencies {

    compile 'com.squareup.okhttp3:okhttp:3.9.0'

    compile 'com.github.bumptech.glide:okhttp3-integration:1.5.0@aar'

}
```

使用 OkHttp2 来作为 HTTP 通讯组件的配置如下：

```
dependencies {

    compile 'com.github.bumptech.glide:okhttp-integration:1.5.0@aar'

    compile 'com.squareup.okhttp:okhttp:2.7.5'

}
```

使用 Volley 来作为 HTTP 通讯组件的配置如下：

```
dependencies {

    compile 'com.github.bumptech.glide:volley-integration:1.5.0@aar'

    compile 'com.mcxiaoke.volley:library:1.0.19'

}
```

BAT交流 892246

当然了，这些库背后的工作原理和我们刚才自己手动实现替换 HTTP 组件的原理是一模一样的。而学会了手动替换组件的原理我们就能更加轻松地扩展更多丰富的功能，因此掌握这一技能还是非常重要的。

4.2.7 实现带进度的 Glide 图片加载功能

扩展目标

首先来确立一下功能扩展的目标。虽说 Glide 本身就已经十分强大了，但是有一个功能却长期以来都不支持，那就是监听下载进度功能。

我们都知道，使用 Glide 来加载一张网络上的图片是非常简单的，但是让人头疼的是，我们却无从得知当前图片的下载进度。如果这张图片很小的话，那么问题也不大，反正很快就会被加载出来。但如果这是一张比较大的 GIF 图，用户耐心等了很久结果图片还没显示出来，这个时候你就会觉得下载进度功能是十分有必要的了。

好的，那么我们今天的目标就是对 Glide 进行功能扩展，使其支持监听图片下载进度的功能。

开始

今天这篇文章我会带着大家从零去创建一个新的项目，一步步地进行实现，最终完成一个带进度的 Glide 图片加载的 Demo。当然，在本篇文章的最后我会提供这个 Demo 的完整源码，但是这里我仍然希望大家能用心跟着我一步步来编写。

那么我们现在就开始吧，首先创建一个新项目，就叫做 GlideProgressTest 吧。

项目创建完成后的第一件事就是要将必要的依赖库引入到当前的项目当中，目前我们必须要依赖的两个库就是 Glide 和 OkHttp。在 app/build.gradle 文件当中添加如下配置：

```
dependencies {

    compile 'com.github.bumptech.glide:glide:3.7.0'

    compile 'com.squareup.okhttp3:okhttp:3.9.0'

}
```

BAT交流 892246

另外，由于 Glide 和 OkHttp 都需要用到网络功能，因此我们还得在

AndroidManifest.xml 中声明一下网络权限才行：

```
<uses-permission android:name="android.permission.INTERNET" />
```

好了，这样准备工作就完成了。

替换通讯组件

通过第二篇文章的源码分析，我们知道了 Glide 内部 HTTP 通讯组件的底层实现是基于 HttpUrlConnection 来进行定制的。但是 HttpUrlConnection 的可扩展性比较有限，我们在它的基础之上无法实现监听下载进度的功能，因此今天的第一个大动作就是要将 Glide 中的 HTTP 通讯组件替换成 OkHttp。

关于 HTTP 通讯组件的替换原理和替换方式，我在第六篇文章当中都介绍得比较清楚了，这里就不再赘述。下面我们就来开始快速地替换一下。

新建一个 OkHttpFetcher 类，并且实现 DataFetcher 接口，代码如下所示：

```
public class OkHttpFetcher implements DataFetcher<InputStream> {



    private final OkHttpClient client;

    private final GlideUrl url;

    private InputStream stream;

    private ResponseBody responseBody;

    private volatile boolean isCancelled;



    public OkHttpFetcher(OkHttpClient client, GlideUrl url) {

        this.client = client;

        this.url = url;

    }
```

```
    @Override
    public InputStream loadData(Priority priority) throws Exception {
        Request.Builder requestBuilder = new Request.Builder()
                .url(url.toStringUrl());
        for (Map.Entry<String, String> headerEntry : url.getHeaders().ent
rySet()) {
            String key = headerEntry.getKey();
            requestBuilder.addHeader(key, headerEntry.getValue());
        }
        Request request = requestBuilder.build();
        if (isCancelled) {
            return null;
        }
        Response response = client.newCall(request).execute();
        responseBody = response.body();
        if (!response.isSuccessful() || responseBody == null) {
            throw new IOException("Request failed with code: " + response.
code());
        }
        stream = ContentLengthInputStream.obtain(responseBody.byteStream
(),
                responseBody.contentLength());
        return stream;
```

```
    }

    @Override
    public void cleanup() {
        try {
            if (stream != null) {
                stream.close();
            }
            if (responseBody != null) {
                responseBody.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String getId() {
        return url.getCacheKey();
    }

    @Override
    public void cancel() {
```

```
        isCancelled = true;

    } }
```

然后新建一个 OkHttpGlideUrlLoader 类，并且实现 ModelLoader

```
public class OkHttpGlideUrlLoader implements ModelLoader<GlideUrl, Input
Stream> {



    private OkHttpClient okHttpClient;



    public static class Factory implements ModelLoaderFactory<GlideUrl,
InputStream> {



        private OkHttpClient client;



        public Factory() {

        }



        public Factory(OkHttpClient client) {

            this.client = client;

        }



        private synchronized OkHttpClient getOkHttpClient() {

            if (client == null) {

                client = new OkHttpClient();
```

```
            }

            return client;

        }



        @Override

        public ModelLoader<GlideUrl, InputStream> build(Context context,
GenericLoaderFactory factories) {

            return new OkHttpGlideUrlLoader(getOkHttpClient());

        }



        @Override

        public void teardown() {

        }

    }



    public OkHttpGlideUrlLoader(OkHttpClient client) {

        this.okHttpClient = client;

    }



    @Override

    public DataFetcher<InputStream> getResourceFetcher(GlideUrl model, i
nt width, int height) {

        return new OkHttpFetcher(okHttpClient, model);

    } }
```

接下来，新建一个 MyGlideModule 类并实现 GlideModule 接口，然后在 registerComponents()方法中将我们刚刚创建的 OkHttpGlideUrlLoader 和 OkHttpFetcher 注册到 Glide 当中，将原来的 HTTP 通讯组件给替换掉，如下所示：

```
public class MyGlideModule implements GlideModule {

    @Override

    public void applyOptions(Context context, GlideBuilder builder) {

    }



    @Override

    public void registerComponents(Context context, Glide glide) {

        glide.register(GlideUrl.class, InputStream.class, new OkHttpGlid
eUrlLoader.Factory());

    } }
```

最后，为了让 Glide 能够识别我们自定义的 MyGlideModule，还得在

AndroidManifest.xml 文件当中加入如下配置才行：

```
<manifest>

    ...

    <application>

        <meta-data

            android:name="com.example.glideprogresstest.MyGlideModule"

            android:value="GlideModule" />

        ...

    </application> </manifest>
```

OK，这样我们就把 Glide 中的 HTTP 通讯组件成功替换成 OkHttp 了。

实现下载进度监听

那么，将 HTTP 通讯组件替换成 OkHttp 之后，我们又该如何去实现监听下载进度的功能呢？这就要依靠 OkHttp 强大的拦截器机制了。

我们只要向 OkHttp 中添加一个自定义的拦截器，就可以在拦截器中捕获到整个 HTTP 的通讯过程，然后加入一些自己的逻辑来计算下载进度，这样就可以实现下载进度监听的功能了。

拦截器属于 OkHttp 的高级功能，不过即使你之前并没有接触过拦截器，我相信你也能轻松看懂本篇文章的，因为它本身并不难。

确定了实现思路之后，那我们就开始动手吧。首先创建一个没有任何逻辑的空拦截器，新建 ProgressInterceptor 类并实现 Interceptor 接口，代码如下所示：

```
public class ProgressInterceptor implements Interceptor {



    @Override

    public Response intercept(Chain chain) throws IOException {

        Request request = chain.request();

        Response response = chain.proceed(request);

        return response;

    }

}
```

这个拦截器中我们可以说是什么都没有做。就是拦截到了 OkHttp 的请求，然后调用 proceed()方法去处理这个请求，最终将服务器响应的 Response 返回。

接下来我们需要启用这个拦截器，修改 MyGlideModule 中的代码，如下所示：

```
public class MyGlideModule implements GlideModule {

    @Override

    public void applyOptions(Context context, GlideBuilder builder) {
```

BAT 892872246

```
    }



    @Override

    public void registerComponents(Context context, Glide glide) {

        OkHttpClient.Builder builder = new OkHttpClient.Builder();

        builder.addInterceptor(new ProgressInterceptor());

        OkHttpClient okHttpClient = builder.build();

        glide.register(GlideUrl.class, InputStream.class, new OkHttpGlid
eUrlLoader.Factory(okHttpClient));

    } }
```

这里我们创建了一个 OkHttpClient.Builder，然后调用 addInterceptor()方法将刚才创建的 ProgressInterceptor 添加进去，最后将构建出来的新 OkHttpClient 对象传入到 OkHttpGlideUrlLoader.Factory 中即可。

好的，现在自定义的拦截器已经启用了，接下来就可以开始去实现下载进度监听的具体逻辑了。首先新建一个 ProgressListener 接口，用于作为进度监听回调的工具，如下所示：

```
public interface ProgressListener {



    void onProgress(int progress);

}
```

然后我们在 ProgressInterceptor 中加入注册下载监听和取消注册下载监听的方法。修改 ProgressInterceptor 中的代码，如下所示：

```
public class ProgressInterceptor implements Interceptor {
```

892872246
BAT文

```
    static final Map<String, ProgressListener> LISTENER_MAP = new HashMa
p<>();



    public static void addListener(String url, ProgressListener listener)
 {

        LISTENER_MAP.put(url, listener);

    }



    public static void removeListener(String url) {

        LISTENER_MAP.remove(url);

    }



    @Override

    public Response intercept(Chain chain) throws IOException {

        Request request = chain.request();

        Response response = chain.proceed(request);

        return response;

    }

}
```

可以看到，这里使用了一个 Map 来保存注册的监听器，Map 的键是一个 URL 地址。之所以要这么做，是因为你可能会使用 Glide 同时加载很多张图片，而这种情况下，必须要能区分出来每个下载进度的回调到底是对应哪个图片 URL 地址的。

接下来就要到今天最复杂的部分了，也就是下载进度的具体计算。我们需要新建一个 ProgressResponseBody 类，并让它继承自 OkHttp 的 ResponseBody，然后在这个类当中去编写具体的监听下载进度的逻辑，代码如下所示：

BAT交流群：892872246

```java
public class ProgressResponseBody extends ResponseBody {


    private static final String TAG = "ProgressResponseBody";


    private BufferedSource bufferedSource;


    private ResponseBody responseBody;


    private ProgressListener listener;


    public ProgressResponseBody(String url, ResponseBody responseBody) {

        this.responseBody = responseBody;

        listener = ProgressInterceptor.LISTENER_MAP.get(url);

    }


    @Override

    public MediaType contentType() {

        return responseBody.contentType();

    }


    @Override

    public long contentLength() {

        return responseBody.contentLength();
```

```
    }


    @Override

    public BufferedSource source() {

        if (bufferedSource == null) {

            bufferedSource = Okio.buffer(new ProgressSource(responseBody.
source()));

        }

        return bufferedSource;

    }


    private class ProgressSource extends ForwardingSource {


        long totalBytesRead = 0;


        int currentProgress;


        ProgressSource(Source source) {

            super(source);

        }


        @Override

        public long read(Buffer sink, long byteCount) throws IOException
{
```

```
            long bytesRead = super.read(sink, byteCount);
            long fullLength = responseBody.contentLength();
            if (bytesRead == -1) {
                totalBytesRead = fullLength;
            } else {
                totalBytesRead += bytesRead;
            }
            int progress = (int) (100f * totalBytesRead / fullLength);
            Log.d(TAG, "download progress is " + progress);
            if (listener != null && progress != currentProgress) {
                listener.onProgress(progress);
            }
            if (listener != null && totalBytesRead == fullLength) {
                listener = null;
            }
            currentProgress = progress;
            return bytesRead;
        }
    }
}
```

其实这段代码也不是很难，下面我来简单解释一下。首先，我们定义了一个 ProgressResponseBody 的构造方法，该构造方法中要求传入一个 url 参数和一个 ResponseBody 参数。那么很显然，url 参数就是图片的 url 地址了，而 ResponseBody 参数则是 OkHttp 拦截到的原始的 ResponseBody 对象。然后在构造方法中，我们

调用了 ProgressInterceptor 中的 LISTENER_MAP 来去获取该 url 对应的监听器回调对象，有了这个对象，待会就可以回调计算出来的下载进度了。

由于继承了 ResponseBody 类之后一定要重写 contentType()、contentLength()和 source()这三个方法，我们在 contentType()和 contentLength()方法中直接就调用传入的原始 ResponseBody 的 contentType()和 contentLength()方法即可，这相当于一种委托模式。但是在 source()方法中，我们就必须加入点自己的逻辑了，因为这里要涉及到具体的下载进度计算。

那么我们具体看一下 source()方法，这里先是调用了原始 ResponseBody 的 source()方法来去获取 Source 对象，接下来将这个 Source 对象封装到了一个 ProgressSource 对象当中，最终再用 Okio 的 buffer()方法封装成 BufferedSource 对象返回。

那么这个 ProgressSource 是什么呢？它是一个我们自定义的继承自 ForwardingSource 的实现类。ForwardingSource 也是一个使用委托模式的工具，它不处理任何具体的逻辑，只是负责将传入的原始 Source 对象进行中转。但是，我们使用 ProgressSource 继承自 ForwardingSource，那么就可以在中转的过程中加入自己的逻辑了。

可以看到，在 ProgressSource 中我们重写了 read()方法，然后在 read()方法中获取该次读取到的字节数以及下载文件的总字节数，并进行一些简单的数学计算就能算出当前的下载进度了。这里我先使用 Log 工具将算出的结果打印了一下，再通过前面获取到的回调监听器对象将结果进行回调。

好的，现在计算下载进度的逻辑已经完成了，那么我们快点在拦截器当中使用它吧。修改 ProgressInterceptor 中的代码，如下所示：

```
public class ProgressInterceptor implements Interceptor {



    ...



    @Override

    public Response intercept(Chain chain) throws IOException {

        Request request = chain.request();

        Response response = chain.proceed(request);

        String url = request.url().toString();
```

```
        ResponseBody body = response.body();

        Response newResponse = response.newBuilder().body(new ProgressRes
ponseBody(url, body)).build();

        return newResponse;

    }

}
```

这里也都是一些 OkHttp 的简单用法。我们通过 Response 的 newBuilder()方法来创建一个新的 Response 对象，并把它的 body 替换成刚才实现的 ProgressResponseBody，最终将新的 Response 对象进行返回，这样计算下载进度的逻辑就能生效了。

代码写到这里，我们就可以来运行一下程序了。现在无论是加载任何网络上的图片，都应该是可以监听到它的下载进度的。

修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"


    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:orientation="vertical">



    <Button

        android:layout_width="wrap_content"

        android:layout_height="wrap_content"

        android:text="Load Image"

        android:onClick="loadImage"

        />
```

```
<ImageView
    android:id="@+id/image"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" /> </LinearLayout>
```

现在就可以运行一下程序了，效果如下图所示。

OK，图片已经加载出来了。那么怎么验证有没有成功监听到图片的下载进度呢？还记得我们刚才在 ProgressResponseBody 中加的打印日志吗？现在只要去 logcat 中观察一下就知道了，如下图所示：

BAT交流群：892872246

```
D/ProgressResponseBody: download progress is 0
D/ProgressResponseBody: download progress is 6
D/ProgressResponseBody: download progress is 8
D/ProgressResponseBody: download progress is 9
D/ProgressResponseBody: download progress is 10
D/ProgressResponseBody: download progress is 11
D/ProgressResponseBody: download progress is 12
D/ProgressResponseBody: download progress is 13
D/ProgressResponseBody: download progress is 14
D/ProgressResponseBody: download progress is 15
D/ProgressResponseBody: download progress is 16
D/ProgressResponseBody: download progress is 17
D/ProgressResponseBody: download progress is 17
D/ProgressResponseBody: download progress is 18
D/ProgressResponseBody: download progress is 19
D/ProgressResponseBody: download progress is 20
D/ProgressResponseBody: download progress is 21
D/ProgressResponseBody: download progress is 22
D/ProgressResponseBody: download progress is 23
D/ProgressResponseBody: download progress is 24
D/ProgressResponseBody: download progress is 25
D/ProgressResponseBody: download progress is 26
D/ProgressResponseBody: download progress is 27
```

http://blog.csdn.net/guolin_blog

由此可见，下载进度监听功能已经成功实现了。

进度显示

虽然现在我们已经能够监听到图片的下载进度了，但是这个进度目前还只能显示在控制台打印当中，这对于用户来说是没有任何意义的，因此我们下一步就是要想办法将下载进度显示到界面上。

现在修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {


    String url = "http://guolin.tech/book.png";


    ImageView image;


    ProgressDialog progressDialog;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        image = (ImageView) findViewById(R.id.image);
        progressDialog = new ProgressDialog(this);
        progressDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTA
L);
        progressDialog.setMessage("加载中");
    }

    public void loadImage(View view) {
        ProgressInterceptor.addListener(url, new ProgressListener() {
            @Override
            public void onProgress(int progress) {
                progressDialog.setProgress(progress);
            }
        });
        Glide.with(this)
             .load(url)
             .diskCacheStrategy(DiskCacheStrategy.NONE)
             .override(Target.SIZE_ORIGINAL, Target.SIZE_ORIGINAL)
```

```
            .into(new GlideDrawableImageViewTarget(image) {

                @Override

                public void onLoadStarted(Drawable placeholder) {

                    super.onLoadStarted(placeholder);

                    progressDialog.show();

                }



                @Override

                public void onResourceReady(GlideDrawable resource, Glid
eAnimation<? super GlideDrawable> animation) {

                    super.onResourceReady(resource, animation);

                    progressDialog.dismiss();

                    ProgressInterceptor.removeListener(url);

                }

            });

    }
```

代码并不复杂。这里我们新增了一个 ProgressDialog 用来显示下载进度，然后在 loadImage()方法中，调用了 ProgressInterceptor.addListener()方法来去注册一个下载监听器，并在 onProgress()回调方法中更新当前的下载进度。

最后，Glide 的 into()方法也做了修改，这次是 into 到了一个 GlideDrawableImageViewTarget 当中。我们重写了它的 onLoadStarted()方法和 onResourceReady()方法，从而实现当图片开始加载的时候显示进度对话框，当图片加载完成时关闭进度对话框的功能。

现在重新运行一下程序，效果如下图所示。

BAT交流群：892872246

当然，不仅仅是静态图片，体积比较大的 GIF 图也是可以成功监听到下载进度的。

比如我们把图片的 url 地址换成 http://guolin.tech/test.gif，重新运行程序，效果如下图所示。

BAT交流群：892872246

好了，这样我们就把带进度的 Glide 图片加载功能完整地实现了一遍。虽然这个例子当中的界面都比较粗糙，下载进度框也是使用的最简陋的，不过只要将功能学会了，界面那都不是事，大家后期可以自己进行各种界面优化。
好了，这样我们就把带进度的 Glide 图片加载功能完整地实现了一遍。虽然这个例子当中的界面都比较粗糙，下载进度框也是使用的最简陋的，不过只要将功能学会了，界面那都不是事，大家后期可以自己进行各种界面优化。

4.2.8 带你全面了解 Glide 4 的用法

本篇将是我们这个 Glide 系列的最后一篇文章。

其实在写这个系列第一篇文章的时候，Glide 就推出 4.0.0 的 RC 版了。那个时候因为我一直研究的都是 Glide 3.7.0 版本，再加上 RC 版本还不太稳定，因此整个系列也都是基于 3.7.0 版本来写的。

而现在，Glide 的最新版本已经出到了 4.4.0，可以说 Glide 4 已经是相当成熟和稳定了。而且也不断有朋友一直在留言，想让我讲一讲 Glide 4 的用法，因为 Glide

BAT必流群：892872246

4 相对于 Glide 3 改动貌似还是挺大的，学完了 Glide 3 再去使用 Glide 4，发现根本就无法使用。

OK，那么今天就让我们用《带你全面了解 Glide 4 的用法》这样一篇文章，给这个 Glide 系列画上一个圆满的句号。

Glide 4 概述

刚才有说到，有些朋友觉得 Glide 4 相对于 Glide 3 改动非常大，其实不然。之所以大家会有这种错觉，是因为你将 Glide 3 的用法直接搬到 Glide 4 中去使用，结果 IDE 全面报错，然后大家可能就觉得 Glide 4 的用法完全变掉了。

其实 Glide 4 相对于 Glide 3 的变动并不大，只是你还没有了解它的变动规则而已。一旦你掌握了 Glide 4 的变动规则之后，你会发现大多数 Glide 3 的用法放到 Glide 4 上都还是通用的。

我对 Glide 4 进行了一个大概的研究之后，发现 Glide 4 并不能算是有什么突破性的升级，而更多是一些 API 工整方面的优化。相比于 Glide 3 的 API，Glide 4 进行了更加科学合理地调整，使得易读性、易写性、可扩展性等方面都有了不错的提升。但如果你已经对 Glide 3 非常熟悉的话，并不是就必须要切换到 Glide 4 上来，因为 Glide 4 上能实现的功能 Glide 3 也都能实现，而且 Glide 4 在性能方面也并没有什么提升。

但是对于新接触 Glide 的朋友而言，那就没必要再去学习 Glide 3 了，直接上手 Glide 4 就是最佳的选择了。

好了，对 Glide 4 进行一个基本的概述之后，接下来我们就要正式开始学习它的用法了。刚才我已经说了，Glide 4 的用法相对于 Glide 3 其实改动并不大。在前面的七篇文章中，我们已经学习了 Glide 3 的基本用法、缓存机制、回调与监听、图片变换、自定义模块等用法，那么今天这篇文章的目标就很简单了，就是要掌握如何在 Glide 4 上实现之前所学习过的所有功能，那么我们现在就开始吧。

开始

要想使用 Glide，首先需要将这个库引入到我们的项目当中。新建一个 Glide4Test 项目，然后在 app/build.gradle 文件当中添加如下依赖：

```
dependencies {

    implementation 'com.github.bumptech.glide:glide:4.4.0'

    annotationProcessor 'com.github.bumptech.glide:compiler:4.4.0'
```

BA 2872246

```
}
```

注意，相比于 Glide 3，这里要多添加一个 compiler 的库，这个库是用于生成 Generated API 的，待会我们会讲到它。

另外，Glide 中需要用到网络功能，因此你还得在 AndroidManifest.xml 中声明一下网络权限才行：

```
<uses-permission android:name="android.permission.INTERNET" />
```

就是这么简单，然后我们就可以自由地使用 Glide 中的任意功能了。

加载图片

现在我们就来尝试一下如何使用 Glide 来加载图片吧。比如这是一张图片的地址：

```
http://guolin.tech/book.png
```

然后我们想要在程序当中去加载这张图片。

那么首先打开项目的布局文件，在布局当中加入一个 Button 和一个 ImageView，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:orientation="vertical">



    <Button

        android:layout_width="wrap_content"

        android:layout_height="wrap_content"

        android:text="Load Image"
```

```
        android:onClick="loadImage"
        />

    <ImageView
        android:id="@+id/image_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

为了让用户点击 Button 的时候能够将刚才的图片显示在 ImageView 上，我们需要修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    ImageView imageView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        imageView = (ImageView) findViewById(R.id.image_view);
    }

    public void loadImage(View view) {
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
        String url = "http://guolin.tech/book.png";

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

    }

}
```

没错，就是这么简单。现在我们来运行一下程序，效果如下图所示：

可以看到，一张网络上的图片已经被成功下载，并且展示到 ImageView 上了。

你会发现，到目前为止，Glide 4 的用法和 Glide 3 是完全一样的，实际上核心的代码就只有这一行而已：

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

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

仍然还是传统的三步走：先 with()，再 load()，最后 into()。对这行代码的解读，我在 Android 图片加载框架最全解析（一），Glide 的基本用法 这篇文章中讲解的很清楚了，这里就不再赘述。

好了，现在你已经成功入门 Glide 4 了，那么接下来就让我们学习一下 Glide 4 的更多用法吧。

占位图

观察刚才加载网络图片的效果，你会发现，点击了 Load Image 按钮之后，要稍微等一会图片才会显示出来。这其实很容易理解，因为从网络上下载图片本来就是需要时间的。那么我们有没有办法再优化一下用户体验呢？当然可以，Glide 提供了各种各样非常丰富的 API 支持，其中就包括了占位图功能。

顾名思义，占位图就是指在图片的加载过程中，我们先显示一张临时的图片，等图片加载出来了再替换成要加载的图片。

下面我们就来学习一下 Glide 占位图功能的使用方法，首先我事先准备好了一张 loading.jpg 图片，用来作为占位图显示。然后修改 Glide 加载部分的代码，如下所示：

```
RequestOptions options = new RequestOptions()

        .placeholder(R.drawable.loading);

Glide.with(this)

     .load(url)

     .apply(options)

     .into(imageView);
```

没错，就是这么简单。这里我们先创建了一个 RequestOptions 对象，然后调用它的 placeholder()方法来指定占位图，再将占位图片的资源 id 传入到这个方法中。最后，在 Glide 的三步走之间加入一个 apply()方法，来应用我们刚才创建的 RequestOptions 对象。

不过如果你现在重新运行一下代码并点击 Load Image，很可能是根本看不到占位图效果的。因为 Glide 有非常强大的缓存机制，我们刚才加载图片的时候 Glide 自动就已经将它缓存下来了，下次加载的时候将会直接从缓存中读取，不会再去网络下载了，因而加载的速度非常快，所以占位图可能根本来不及显示。

BAT交流群：892872246

因此这里我们还需要稍微做一点修改，来让占位图能有机会显示出来，修改代码如下所示：

```
RequestOptions options = new RequestOptions()

        .placeholder(R.drawable.loading)

        .diskCacheStrategy(DiskCacheStrategy.NONE);

Glide.with(this)

     .load(url)

     .apply(options)

     .into(imageView);
```

可以看到，这里在 RequestOptions 对象中又串接了一个 diskCacheStrategy()方法，并传入 DiskCacheStrategy.NONE 参数，这样就可以禁用掉 Glide 的缓存功能。

关于 Glide 缓存方面的内容我们待会儿会进行更详细的讲解，这里只是为了测试占位图功能而加的一个额外配置，暂时你只需要知道禁用缓存必须这么写就可以了。

现在重新运行一下代码，效果如下图所示：

BAT交流群：892872246

可以看到，当点击 Load Image 按钮之后会立即显示一张占位图，然后等真正的图片加载完成之后会将占位图替换掉。

除了这种加载占位图之外，还有一种异常占位图。异常占位图就是指，如果因为某些异常情况导致图片加载失败，比如说手机网络信号不好，这个时候就显示这张异常占位图。

异常占位图的用法相信你已经可以猜到了，首先准备一张 error.jpg 图片，然后修改 Glide 加载部分的代码，如下所示：

```
RequestOptions options = new RequestOptions()

        .placeholder(R.drawable.ic_launcher_background)

        .error(R.drawable.error)
```

```
        .diskCacheStrategy(DiskCacheStrategy.NONE);

Glide.with(this)

     .load(url)

     .apply(options)

     .into(imageView);
```

很简单，这里又串接了一个 error()方法就可以指定异常占位图了。

其实看到这里，如果你熟悉 Glide 3 的话，相信你已经掌握 Glide 4 的变化规律了。在 Glide 3 当中，像 placeholder()、error()、diskCacheStrategy()等等一系列的 API，都是直接串联在 Glide 三步走方法中使用的。

而 Glide 4 中引入了一个 RequestOptions 对象，将这一系列的 API 都移动到了 RequestOptions 当中。这样做的好处是可以使我们摆脱冗长的 Glide 加载语句，而且还能进行自己的 API 封装，因为 RequestOptions 是可以作为参数传入到方法中的。

比如你就可以写出这样的 Glide 加载工具类：

```
public class GlideUtil {



    public static void load(Context context,

                            String url,

                            ImageView imageView,

                            RequestOptions options) {

        Glide.with(context)

             .load(url)

             .apply(options)

             .into(imageView);

    }
```

BAT 892246

```
}
```

指定图片大小

实际上，使用 Glide 在大多数情况下我们都是不需要指定图片大小的，因为 Glide 会自动根据 ImageView 的大小来决定图片的大小，以此保证图片不会占用过多的内存从而引发 OOM。

不过，如果你真的有这样的需求，必须给图片指定一个固定的大小，Glide 仍然是支持这个功能的。修改 Glide 加载部分的代码，如下所示：

```
RequestOptions options = new RequestOptions()

        .override(200, 100);

Glide.with(this)

     .load(url)

     .apply(options)

     .into(imageView);
```

仍然非常简单，这里使用 override()方法指定了一个图片的尺寸。也就是说，Glide 现在只会将图片加载成 200*100 像素的尺寸，而不会管你的 ImageView 的大小是多少了。

如果你想加载一张图片的原始尺寸的话，可以使用 Target.SIZE_ORIGINAL 关键字，如下所示：

```
RequestOptions options = new RequestOptions()

        .override(Target.SIZE_ORIGINAL);

Glide.with(this)

     .load(url)

     .apply(options)

     .into(imageView);
```

这样的话，Glide 就不会再去自动压缩图片，而是会去加载图片的原始尺寸。当然，这种写法也会面临着更高的 OOM 风险。

BAT/2246

缓存机制

Glide 的缓存设计可以说是非常先进的，考虑的场景也很周全。在缓存这一功能上，Glide 又将它分成了两个模块，一个是内存缓存，一个是硬盘缓存。

这两个缓存模块的作用各不相同，内存缓存的主要作用是防止应用重复将图片数据读取到内存当中，而硬盘缓存的主要作用是防止应用重复从网络或其他地方重复下载和读取数据。

内存缓存和硬盘缓存的相互结合才构成了 Glide 极佳的图片缓存效果，那么接下来我们就来分别学习一下这两种缓存的使用方法。

首先来看内存缓存。

你要知道，默认情况下，Glide 自动就是开启内存缓存的。也就是说，当我们使用 Glide 加载了一张图片之后，这张图片就会被缓存到内存当中，只要在它还没从内存中被清除之前，下次使用 Glide 再加载这张图片都会直接从内存当中读取，而不用重新从网络或硬盘上读取了，这样无疑就可以大幅度提升图片的加载效率。比方说你在一个 RecyclerView 当中反复上下滑动，RecyclerView 中只要是 Glide 加载过的图片都可以直接从内存当中迅速读取并展示出来，从而大大提升了用户体验。

而 Glide 最为人性化的是，你甚至不需要编写任何额外的代码就能自动享受到这个极为便利的内存缓存功能，因为 Glide 默认就已经将它开启了。

那么既然已经默认开启了这个功能，还有什么可讲的用法呢？只有一点，如果你有什么特殊的原因需要禁用内存缓存功能，Glide 对此提供了接口：

```
RequestOptions options = new RequestOptions()

        .skipMemoryCache(true);

Glide.with(this)

     .load(url)

     .apply(options)

     .into(imageView);
```

可以看到，只需要调用 skipMemoryCache()方法并传入 true，就表示禁用掉 Glide 的内存缓存功能。

接下来我们开始学习硬盘缓存方面的内容。

BAT交流群：892246

其实在刚刚学习占位图功能的时候，我们就使用过硬盘缓存的功能了。当时为了禁止 Glide 对图片进行硬盘缓存而使用了如下代码：

```
RequestOptions options = new RequestOptions()

        .diskCacheStrategy(DiskCacheStrategy.NONE);

Glide.with(this)

     .load(url)

     .apply(options)

     .into(imageView);
```

调用 diskCacheStrategy()方法并传入 DiskCacheStrategy.NONE，就可以禁用掉 Glide 的硬盘缓存功能了。

这个 diskCacheStrategy()方法基本上就是 Glide 硬盘缓存功能的一切，它可以接收五种参数：

DiskCacheStrategy.NONE： 表示不缓存任何内容。
DiskCacheStrategy.DATA： 表示只缓存原始图片。
DiskCacheStrategy.RESOURCE： 表示只缓存转换过后的图片。
DiskCacheStrategy.ALL ： 表示既缓存原始图片，也缓存转换过后的图片。
DiskCacheStrategy.AUTOMATIC： 表示让 Glide 根据图片资源智能地选择使用哪一种缓存策略（默认选项）。
其中，DiskCacheStrategy.DATA 对应 Glide 3 中的 DiskCacheStrategy.SOURCE，DiskCacheStrategy.RESOURCE 对应 Glide 3 中的 DiskCacheStrategy.RESULT。而 DiskCacheStrategy.AUTOMATIC 是 Glide 4 中新增的一种缓存策略，并且在不指定 diskCacheStrategy 的情况下默认使用就是的这种缓存策略。

上面五种参数的解释本身并没有什么难理解的地方，但是关于转换过后的图片这个概念大家可能需要了解一下。就是当我们使用 Glide 去加载一张图片的时候，Glide 默认并不会将原始图片展示出来，而是会对图片进行压缩和转换（我们会在稍后学习这方面的内容）。总之就是经过种种一系列操作之后得到的图片，就叫转换过后的图片。

好的，关于 Glide 4 硬盘缓存的内容就讲到这里。想要了解更多 Glide 缓存方面的知识，可以参考 Android 图片加载框架最全解析（三），深入探究 Glide 的缓存机制 这篇文章。

BAT交流群：892872246

指定加载格式

我们都知道，Glide 其中一个非常亮眼的功能就是可以加载 GIF 图片，而同样作为非常出色的图片加载框架的 Picasso 是不支持这个功能的。

而且使用 Glide 加载 GIF 图并不需要编写什么额外的代码，Glide 内部会自动判断图片格式。比如我们将加载图片的 URL 地址改成一张 GIF 图，如下所示：

```
Glide.with(this)

    .load("http://guolin.tech/test.gif")

    .into(imageView);
```

现在重新运行一下代码，效果如下图所示：

BAT交流群：892872246

也就是说，不管我们传入的是一张普通图片，还是一张 GIF 图片，Glide 都会自动进行判断，并且可以正确地把它解析并展示出来。

但是如果我想指定加载格式该怎么办呢？就比如说，我希望加载的这张图必须是一张静态图片，我不需要 Glide 自动帮我判断它到底是静图还是 GIF 图。

想实现这个功能仍然非常简单，我们只需要再串接一个新的方法就可以了，如下所示：

```
Glide.with(this)
    .asBitmap()
    .load("http://guolin.tech/test.gif")
    .into(imageView);
```

可以看到，这里在 with()方法的后面加入了一个 asBitmap()方法，这个方法的意思就是说这里只允许加载静态图片，不需要 Glide 去帮我们自动进行图片格式的判断了。如果你传入的还是一张 GIF 图的话，Glide 会展示这张 GIF 图的第一帧，而不会去播放它。

熟悉 Glide 3 的朋友对 asBitmap()方法肯定不会陌生对吧？但是千万不要觉得这里就没有陷阱了，在 Glide 3 中的语法是先 load()再 asBitmap()的，而在 Glide 4 中是先 asBitmap()再 load()的。乍一看可能分辨不出来有什么区别，但如果你写错了顺序就肯定会报错了。

那么类似地，既然我们能强制指定加载静态图片，就也能强制指定加载动态图片，对应的方法是 asGif()。而 Glide 4 中又新增了 asFile()方法和 asDrawable()方法，分别用于强制指定文件格式的加载和 Drawable 格式的加载，用法都比较简单，就不再进行演示了。

回调与监听

回调与监听这部分的内容稍微有点多，我们分成四部分来学习一下。

1. into()方法

我们都知道 Glide 的 into()方法中是可以传入 ImageView 的。那么 into()方法还可以传入别的参数吗？我们可以让 Glide 加载出来的图片不显示到 ImageView 上吗？答案是肯定的，这就需要用到自定义 Target 功能。

Glide 中的 Target 功能多样且复杂，下面我就先简单演示一种 SimpleTarget 的用法吧，代码如下所示：

BAT交流群：892872246

```
SimpleTarget<Drawable> simpleTarget = new SimpleTarget<Drawable>() {

    @Override

    public void onResourceReady(Drawable resource, Transition<? super Dr
awable> transition) {

        imageView.setImageDrawable(resource);

    }};

public void loadImage(View view) {

    Glide.with(this)

         .load("http://guolin.tech/book.png")

         .into(simpleTarget);}
```

这里我们创建了一个 SimpleTarget 的实例，并且指定它的泛型是 Drawable，然后重写了 onResourceReady()方法。在 onResourceReady()方法中，我们就可以获取到 Glide 加载出来的图片对象了，也就是方法参数中传过来的 Drawable 对象。有了这个对象之后你可以使用它进行任意的逻辑操作，这里我只是简单地把它显示到了 ImageView 上。

SimpleTarget 的实现创建好了，那么只需要在加载图片的时候将它传入到 into()方法中就可以了。

这里限于篇幅原因我只演示了自定义 Target 的简单用法，想学习更多相关的内容可以去阅读 Android 图片加载框架最全解析（四），玩转 Glide 的回调与监听 。

2. preload()方法

Glide 加载图片虽说非常智能，它会自动判断该图片是否已经有缓存了，如果有的话就直接从缓存中读取，没有的话再从网络去下载。但是如果我希望提前对图片进行一个预加载，等真正需要加载图片的时候就直接从缓存中读取，不想再等待慢长的网络加载时间了，这该怎么办呢？

不用担心，Glide 专门给我们提供了预加载的接口，也就是 preload()方法，我们只需要直接使用就可以了。

preload()方法有两个方法重载，一个不带参数，表示将会加载图片的原始尺寸，另一个可以通过参数指定加载图片的宽和高。

preload()方法的用法也非常简单，直接使用它来替换 into()方法即可，如下所示：

BAT交流群：892872246

```
Glide.with(this)
    .load("http://guolin.tech/book.png")
    .preload();
```

调用了预加载之后，我们以后想再去加载这张图片就会非常快了，因为 Glide 会直接从缓存当中去读取图片并显示出来，代码如下所示：

```
Glide.with(this)
    .load("http://guolin.tech/book.png")
    .into(imageView);
```

3. submit()方法

一直以来，我们使用 Glide 都是为了将图片显示到界面上。虽然我们知道 Glide 会在图片的加载过程中对图片进行缓存，但是缓存文件到底是存在哪里的，以及如何去直接访问这些缓存文件？我们都还不知道。

其实 Glide 将图片加载接口设计成这样也是希望我们使用起来更加的方便，不用过多去考虑底层的实现细节。但如果我现在就是想要去访问图片的缓存文件该怎么办呢？这就需要用到 submit()方法了。

submit()方法其实就是对应的 Glide 3 中的 downloadOnly()方法，和 preload()方法类似，submit()方法也是可以替换 into()方法的，不过 submit()方法的用法明显要比 preload()方法复杂不少。这个方法只会下载图片，而不会对图片进行加载。当图片下载完成之后，我们可以得到图片的存储路径，以便后续进行操作。

那么首先我们还是先来看下基本用法。submit()方法有两个方法重载：

submit()
submit(int width, int height)
其中 submit()方法是用于下载原始尺寸的图片，而 submit(int width, int height)则可以指定下载图片的尺寸。

这里就以 submit()方法来举例。当调用了 submit()方法后会立即返回一个 FutureTarget 对象，然后 Glide 会在后台开始下载图片文件。接下来我们调用 FutureTarget 的 get()方法就可以去获取下载好的图片文件了，如果此时图片还没有下载完，那么 get()方法就会阻塞住，一直等到图片下载完成才会有值返回。

下面我们通过一个例子来演示一下吧，代码如下所示：

```
public void downloadImage() {
```

BAT交流群：892872246

```
    new Thread(new Runnable() {

        @Override

        public void run() {

            try {

                String url = "http://www.guolin.tech/book.png";

                final Context context = getApplicationContext();

                FutureTarget<File> target = Glide.with(context)

                        .asFile()

                        .load(url)

                        .submit();

                final File imageFile = target.get();

                runOnUiThread(new Runnable() {

                    @Override

                    public void run() {

                        Toast.makeText(context, imageFile.getPath(), Toast.
LENGTH_LONG).show();

                    }

                });

            } catch (Exception e) {

                e.printStackTrace();

            }

        }

    }).start();}
```

这段代码稍微有一点点长，我带着大家解读一下。首先，submit()方法必须要用在子线程当中，因为刚才说了 FutureTarget 的 get()方法是会阻塞线程的，因此这里的第一步就是 new 了一个 Thread。在子线程当中，我们先获取了一个 Application Context，这个时候不能再用 Activity 作为 Context 了，因为会有 Activity 销毁了但子线程还没执行完这种可能出现。

接下来就是 Glide 的基本用法，只不过将 into()方法替换成了 submit()方法，并且还使用了一个 asFile()方法来指定加载格式。submit()方法会返回一个 FutureTarget 对象，这个时候其实 Glide 已经开始在后台下载图片了，我们随时都可以调用 FutureTarget 的 get()方法来获取下载的图片文件，只不过如果图片还没下载好线程会暂时阻塞住，等下载完成了才会把图片的 File 对象返回。

最后，我们使用 runOnUiThread()切回到主线程，然后使用 Toast 将下载好的图片文件路径显示出来。

现在重新运行一下代码，效果如下图所示。

BAT交流群：892872246

这样我们就能清晰地看出来图片完整的缓存路径是什么了。

4. listener()方法

其实 listener()方法的作用非常普遍，它可以用来监听 Glide 加载图片的状态。举个例子，比如说我们刚才使用了 preload()方法来对图片进行预加载，但是我怎样确定预加载有没有完成呢？还有如果 Glide 加载图片失败了，我该怎样调试错误的原因呢？答案都在 listener()方法当中。

下面来看下 listener()方法的基本用法吧，不同于刚才几个方法都是要替换 into()方法的，listener()是结合 into()方法一起使用的，当然也可以结合 preload()方法一起使用。最基本的用法如下所示：

```
Glide.with(this)

    .load("http://www.guolin.tech/book.png")

    .listener(new RequestListener<Drawable>() {

        @Override

        public boolean onLoadFailed(@Nullable GlideException e, Object m
odel, Target<Drawable> target, boolean isFirstResource) {

            return false;

        }



        @Override

        public boolean onResourceReady(Drawable resource, Object model,
Target<Drawable> target, DataSource dataSource, boolean isFirstResource)
 {

            return false;

        }

    })

    .into(imageView);
```

这里我们在 into()方法之前串接了一个 listener()方法，然后实现了一个 RequestListener 的实例。其中 RequestListener 需要实现两个方法，一个

onResourceReady()方法，一个 onLoadFailed()方法。从方法名上就可以看出来了，当图片加载完成的时候就会回调 onResourceReady()方法，而当图片加载失败的时候就会回调 onLoadFailed()方法，onLoadFailed()方法中会将失败的 GlideException 参数传进来，这样我们就可以定位具体失败的原因了。

没错，listener()方法就是这么简单。不过还有一点需要处理，onResourceReady()方法和 onLoadFailed()方法都有一个布尔值的返回值，返回 false 就表示这个事件没有被处理，还会继续向下传递，返回 true 就表示这个事件已经被处理掉了，从而不会再继续向下传递。举个简单点的例子，如果我们在 RequestListener 的 onResourceReady()方法中返回了 true，那么就不会再回调 Target 的 onResourceReady()方法了。

关于回调与监听的内容就讲这么多吧，如果想要学习更多深入的内容以及源码解析，还是请参考这篇文章 Android 图片加载框架最全解析（四），玩转 Glide 的回调与监听 。

图片变换

图片变换的意思就是说，Glide 从加载了原始图片到最终展示给用户之前，又进行了一些变换处理，从而能够实现一些更加丰富的图片效果，如图片圆角化、圆形化、模糊化等等。

添加图片变换的用法非常简单，我们只需要在 RequestOptions 中串接 transforms()方法，并将想要执行的图片变换操作作为参数传入 transforms()方法即可，如下所示：

```
RequestOptions options = new RequestOptions()

        .transforms(...);

Glide.with(this)

     .load(url)

     .apply(options)

     .into(imageView);
```

至于具体要进行什么样的图片变换操作，这个通常都是需要我们自己来写的。不过 Glide 已经内置了几种图片变换操作，我们可以直接拿来使用，比如 CenterCrop、FitCenter、CircleCrop 等。

但所有的内置图片变换操作其实都不需要使用 transform()方法，Glide 为了方便我们使用直接提供了现成的 API：

BAT交流群：892872246

```
RequestOptions options = new RequestOptions()
        .centerCrop();


RequestOptions options = new RequestOptions()
        .fitCenter();


RequestOptions options = new RequestOptions()
        .circleCrop();
```

当然，这些内置的图片变换 API 其实也只是对 transform()方法进行了一层封装而已，它们背后的源码仍然还是借助 transform()方法来实现的。

这里我们就选择其中一种内置的图片变换操作来演示一下吧，circleCrop()方法是用来对图片进行圆形化裁剪的，我们动手试一下，代码如下所示：

```
String url = "http://guolin.tech/book.png";

RequestOptions options = new RequestOptions()
        .circleCrop();

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

重新运行一下程序并点击加载图片按钮，效果如下图所示。

BAT交流群：892872246

可以看到，现在展示的图片是对原图进行圆形化裁剪后得到的图片。

当然，除了使用内置的图片变换操作之外，我们完全可以自定义自己的图片变换操作。理论上，在对图片进行变换这个步骤中我们可以进行任何的操作，你想对图片怎么样都可以。包括圆角化、圆形化、黑白化、模糊化等等，甚至你将原图片完全替换成另外一张图都是可以的。

不过由于这部分内容相对于 Glide 3 没有任何的变化，因此就不再重复进行讲解了。想学习自定义图片变换操作的朋友们可以参考这篇文章 Android 图片加载框架最全解析（五），Glide 强大的图片变换功能 。

关于图片变换，最后我们再来看一个非常优秀的开源库，glide-transformations。它实现了很多通用的图片变换效果，如裁剪变换、颜色变换、模糊变换等等，使得我们可以非常轻松地进行各种各样的图片变换。

BAT交流群：892872246

glide-transformations 的项目主页地址是 https://github.com/wasabeef/glide-transformations 。

下面我们就来体验一下这个库的强大功能吧。首先需要将这个库引入到我们的项目当中，在 app/build.gradle 文件当中添加如下依赖：

```
dependencies {

    implementation 'jp.wasabeef:glide-transformations:3.0.1'

}
```

我们可以对图片进行单个变换处理，也可以将多种图片变换叠加在一起使用。比如我想同时对图片进行模糊化和黑白化处理，就可以这么写：

```
String url = "http://guolin.tech/book.png";

RequestOptions options = new RequestOptions()

        .transforms(new BlurTransformation(), new GrayscaleTransformatio
n());

Glide.with(this)

     .load(url)

     .apply(options)

     .into(imageView);
```

可以看到，同时执行多种图片变换的时候，只需要将它们都传入到 transforms() 方法中即可。现在重新运行一下程序，效果如下图所示。

BAT交流群：892872246

当然，这只是 glide-transformations 库的一小部分功能而已，更多的图片变换效果你可以到它的 GitHub 项目主页去学习。

自定义模块

自定义模块属于 Glide 中的高级功能，同时也是难度比较高的一部分内容。

这里我不可能在这一篇文章中将自定义模块的内容全讲一遍，限于篇幅的限制我只能讲一讲 Glide 4 中变化的这部分内容。关于 Glide 自定义模块的全部内容，请大家去参考 Android 图片加载框架最全解析（六），探究 Glide 的自定义模块功能 这篇文章。

自定义模块功能可以将更改 Glide 配置，替换 Glide 组件等操作独立出来，使得我们能轻松地对 Glide 的各种配置进行自定义，并且又和 Glide 的图片加载逻辑没有任何交集，这也是一种低耦合编程方式的体现。下面我们就来学习一下自定义模块要如何实现。

BAT交流群：892872246

首先定义一个我们自己的模块类，并让它继承自 AppGlideModule，如下所示：

```
@GlideModulepublic class MyAppGlideModule extends AppGlideModule {



    @Override

    public void applyOptions(Context context, GlideBuilder builder) {



    }



    @Override

    public void registerComponents(Context context, Glide glide, Registr
y registry) {



    }

}
```

}

可以看到，在 MyAppGlideModule 类当中，我们重写了 applyOptions()和 registerComponents()方法，这两个方法分别就是用来更改 Glide 配置以及替换 Glide 组件的。

注意在 MyAppGlideModule 类在上面，我们加入了一个@GlideModule 的注解，这是 Gilde 4 和 Glide 3 最大的一个不同之处。在 Glide 3 中，我们定义了自定义模块之后，还必须在 AndroidManifest.xml 文件中去注册它才能生效，而在 Glide 4 中是不需要的，因为@GlideModule 这个注解已经能够让 Glide 识别到这个自定义模块了。

这样的话，我们就将 Glide 自定义模块的功能完成了。后面只需要在 applyOptions()和 registerComponents()这两个方法中加入具体的逻辑，就能实现更改 Glide 配置或者替换 Glide 组件的功能了。详情还是请参考 Android 图片加载框架最全解析（六），探究 Glide 的自定义模块功能 这篇文章，这里就不再展开讨论了。

BAT交流群：892872246

使用 Generated API

Generated API 是 Glide 4 中全新引入的一个功能，它的工作原理是使用注解处理器（Annotation Processor）来生成出一个 API，在 Application 模块中可使用该流式 API 一次性调用到 RequestBuilder，RequestOptions 和集成库中所有的选项。

这么解释有点拗口，简单点说，就是 Glide 4 仍然给我们提供了一套和 Glide 3 一模一样的流式 API 接口。毕竟有些人还是觉得 Glide 3 的 API 更好用一些，比如说我。

Generated API 对于熟悉 Glide 3 的朋友来说那是再简单不过了，基本上就是和 Glide 3 一模一样的用法，只不过需要把 Glide 关键字替换成 GlideApp 关键字，如下所示：

```
GlideApp.with(this)

        .load(url)

        .placeholder(R.drawable.loading)

        .error(R.drawable.error)

        .skipMemoryCache(true)

        .diskCacheStrategy(DiskCacheStrategy.NONE)

        .override(Target.SIZE_ORIGINAL)

        .circleCrop()

        .into(imageView);
```

不过，有可能你的 IDE 中会提示找不到 GlideApp 这个类。这个类是通过编译时注解自动生成的，首先确保你的代码中有一个自定义的模块，并且给它加上了 @GlideModule 注解，也就是我们在上一节所讲的内容。然后在 Android Studio 中点击菜单栏 Build -> Rebuild Project，GlideApp 这个类就会自动生成了。

当然，Generated API 所能做到的并不只是这些而已，它还可以对现有的 API 进行扩展，定制出任何属于你自己的 API。

下面我来具体举个例子，比如说我们要求项目中所有图片的缓存策略全部都要缓存原始图片，那么每次在使用 Glide 加载图片的时候，都去指定 diskCacheStrategy(DiskCacheStrategy.DATA)这么长长的一串代码，确实是让人比较心烦。这种情况我们就可以去定制一个自己的 API 了。

BAT交流群：892872246

定制自己的 API 需要借助@GlideExtension 和@GlideOption 这两个注解。创建一个我们自定义的扩展类，代码如下所示：

```
@GlideExtensionpublic class MyGlideExtension {



    private MyGlideExtension() {



    }



    @GlideOption

    public static void cacheSource(RequestOptions options) {

        options.diskCacheStrategy(DiskCacheStrategy.DATA);

    }

}
```

这里我们定义了一个 MyGlideExtension 类，并且给加上了一个@GlideExtension 注解，然后要将这个类的构造函数声明成 private，这都是必须要求的写法。

接下来就可以开始自定义 API 了，这里我们定义了一个 cacheSource()方法，表示只缓存原始图片，并给这个方法加上了@GlideOption 注解。注意自定义 API 的方法都必须是静态方法，而且第一个参数必须是 RequestOptions，后面你可以加入任意多个你想自定义的参数。

在 cacheSource()方法中，我们仍然还是调用的 diskCacheStrategy(DiskCacheStrategy.DATA)方法，所以说 cacheSource()就是一层简化 API 的封装而已。

然后在 Android Studio 中点击菜单栏 Build -> Rebuild Project，神奇的事情就会发生了，你会发现你已经可以使用这样的语句来加载图片了：

```
GlideApp.with(this)

        .load(url)

        .cacheSource()
```

892872246

```
        .into(imageView);
```

有了这个强大的功能之后，我们使用 Glide 就能变得更加灵活了。

结束语

这样我们基本上就将 Glide 4 的所有重要内容都介绍完了，如果你以前非常熟悉 Glide 3 的话，看完这篇文章之后相信你已经能够熟练使用 Glide 4 了。而如果你以前并未接触过 Glide，仅仅只看这一篇文章可能了解得还不够深入，建议最好还是把前面的七篇文章也去通读一下，这样你才能成为一名 Glide 好手。

4.3 手写图片加载框架实战

第五章、网络访问框架设计

5.1 网络通信必备基础

5.1.1 Restful URL

本文将先从通过 Restful 风格的 URL 请求来介绍网络通信必备基础

SpringMVC 的 URL 请求的配置方式有两种

第一种方式：.do,.action 类似于 /user/addUser.do /user/addUser.action

*第二种配置方式：/ 类似于/user/addUser /user/delUser**

注意:第二种方式其实是包含第一种方式,但是当我们的页面包含静态资源文件.js,.image,.css 的时候 springMVC 会将他们认为是一个 url 请求去匹配,结果 404。*

怎么解决？
三种方式：
一：在 springmvc.xml 配置文件里面配置：

```
<mvc:resources location="/jsAndCss/" mapping="/jsAndCss/**"></mvc:resour
ces>
```

二：在 springmvc.xml 配置文件里面配置：

BAT 892872246

```
<mvc:default-servlet-handler/>
```

二：在 springmvc.xml 配置文件里面配置：

```
<mvc:default-servlet-handler/>
```

三：在 web.xml 配置：

```
<servlet-mapping>

    <servlet-name>default</servlet-name>

    <url-pattern>*.js</url-pattern>        </servlet-mapping>
```

ps：实际应用中第一种方式比较常见，因为后两种方式使用的是容器默认的 servlet，不同的容器默认的 servlet 不同，实际项目部署时多用 WebLogic，WebSphere。Tomcat 多用于开发。为了防止部署的时候出现差错，所以一般用第一种方式。

Restful 是什么

中文名称：表现层状态转化。用 URL 表示要操作的资源，用不同的 HTTP 请求（GET，POST，PUT，DELETE）描述对资源的操作，通过 HTTP 的状态码来判断此次对资源操作的结果，这就是 Restful 风格。

其实 Restful 风格是对请求的一次解耦，提高了 url 的重用性。

状态转化：客户端通过 http 协议访问服务端资源,通过 http 协议里面定义好的四个动词：GET POST PUT DELETE 来讲资源进行状态转化。
GET 用来获取资源
POST 用来新增资源
PUT 用来更新资源
DELETE 用来删除资源
put 和 delete 虽然是 http 协议的规范 ,但是遗憾的是,所有的无论是 html4 还是 h5 都不支持,所以我们在实际开开发过程当中要模拟这两种状态。

普通的 URL 开发方式 ： http://qujianlei.com/getUserList addUser deleteUser updateUser

Restful 风格的 URL：

BAT交流群：892872246

http://qujianlei.com/users/ method = “get” 获取用户列表
http://qujianlei.com/user method = “post” 更新用户
http://qujianlei.com/user method=“delete” 删除用户
http://qujianlei.com/user method = “put” 新增用户

Restful 怎么实现

SpringMVC 提供了对 restful 风格很友好的支持。

支持无后缀的 uri 匹配,是一种很优雅的 restful 风格的 URI。

对于 html 不支持的 delete 和 put 两种提交方式,提供了一个过滤器进行转化：设置一个 _method = delete / put 的元素,通过 springmvc 提供的过滤器, 能够将他转化成 http 协议的 delete 和 put 提交方式 ,可以在 springmvc 的 @RequestMapping 注解里面获取。
像下面这样来模拟 put 和 delete 两种方式：

```
url :'<%=path%>/user/'+userId+'?_method=delete'

url :'<%=path%>/user/'+userId+'?_method=put'
```

a：配置 HiddenHttpMethodFilter 过滤器

```
<!-- 配置 restful 风格 url 请求的 hidden 过滤器 --><filter>

    <filter-name>hiddenHttpMethodFilter</filter-name>

    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter<
/filter-class></filter><filter-mapping>

    <filter-name>hiddenHttpMethodFilter</filter-name>

    <url-pattern>/*</url-pattern></filter-mapping>
```

b：在 @RequesetMapping 注解用指定 method 的方式来匹配 url

```
/**

 * restful 风格的用户列表

 * @return
```

```
*/@RequestMapping(value="/users",method = RequestMethod.GET)

   public @ResponseBody List<User> getUserListRestful(){

   List<User> userList = userService.getUserList(new User());

   return userList;}
```

c：对于 http://qujianlei.com/user/1 带参数的这种方式,需要用 @PathVariable 注解来接收

```
/**

 * restful 风格的删除

 * @param userId

 * @return

 */@RequestMapping(value="/user/{userId}" ,method = RequestMethod.DELET
E)

public @ResponseBody Map<String,Object> userDelRestful(@PathVariable("us
erId") Long userId){

   Map<String,Object> resultMap = new HashMap<String,Object>();

   if(userId!=null){

       if(userService.delUser(userId)){

           resultMap.put("result", "删除用户成功") ;

       }else{

           resultMap.put("result", "删除用户失败");

       }

   }

   return resultMap;}
```

BAT交流群892246

Restful 有什么意义

网络上的所有的信息体都看作一种资源，对网络资源的某种操作，都是通过 method 来确定的。

对于现在的数据或者资源提供方，对外透露的接口一般是 restful 风格的，有利于不同系统之间的资源共享,而且只需要遵守规范不需要做另外的配置就能达到资源共享。

这是豆瓣提供的一个 restful 风格的 URL ：
https://api.douban.com/v2/book/2129650

获取到的 json 格式的数据如下：

{"rating":{"max":10,"numRaters":16,"average":"7.4","min":0},"subtitle":"","author":["廖雪峰"],"pubdate":"2007","tags":[{"count":21,"name":"spring","title":"spring"},{"count":13,"name":"Java","title":"Java"},{"count":6,"name":"javaee","title":"javaee"},{"count":5,"name":"j2ee","title":"j2ee"},{"count":4,"name":"计算机","title":"计算机"},{"count":4,"name":"编程","title":"编程"},{"count":3,"name":"藏书","title":"藏书"},{"count":3,"name":"POJO","title":"POJO"}],"origin_title":"","image":"https://img3.doubanio.com/view/subject/m/public/s2552283.jpg","binding":"平装","translator":[],"catalog":"","pages":"509","images":{"small":"https://img3.doubanio.com/view/subject/s/public/s2552283.jpg","large":"https://img3.doubanio.com/view/subject/l/public/s2552283.jpg","medium":"https://img3.doubanio.com/view/subject/m/public/s2552283.jpg"},"alt":"https://book.douban.com/subject/2129650/","id":"2129650","publisher":"电子工业出版社","isbn10":"7121042622","isbn13":"9787121042621","title":"Spring 2.0 核心技术与最佳实践","url":"https://api.douban.com/v2/book/2129650","alt_title":"","author_intro":"","summary":"本书注重实践而又深入理论，由浅入深且详细介绍了 Spring 2.0 框架的几乎全部的内容，并重点突出 2.0 版本的新特性。本书将为读者展示如何应用 Spring 2.0 框架创建灵活高效的 JavaEE 应用，并提供了一个真正可直接部署的完整的 Web 应用程序——Live 在线书店(http://www.livebookstore.net)。\n 在介绍 Spring 框架的同时，本书还介绍了与 Spring 相关的大量第三方框架，涉及领域全面，实用性强。本书另一大特色是实用性强，易于上手，以实际项目为出发点，介绍项目开发中应遵循的最佳开发模式。\n 本书还介绍了大量实践性极强的例子，并给出了完整的配置步骤，几乎覆盖了 Spring 2.0 版本的新特性。\n 本书适合有一定 Java 基础的读者，对 JavaEE 开发人员特别有帮助。本书既可以作为 Spring 2.0 的学习指南，也可以作为实际项目开发的参考手册。","price":"59.8"}

我们在后台获取到了这些数据之后，再通过一些 json 解析包，像：fastjson，gson…就可以把获取到的数据转换成 java 对象，然后再进行数据库的一些增删改查操作。

BAT交流群：892872246

restful 是用来外提供接口的，主要用于在不同的系统之间进行数据的交互。

5.1.2 HTTP 协议& TCP/IP 协议

一、Http 协议

当你在浏览器地址栏敲入“http://www.jianshu.com/”，然后猛按回车，呈现在你面前的，将是简书的首页了（这真是废话，你会认为这是理所当然的）。作为一个开发者，尤其是 web 开发人员，我想你有必要去了解这一系列的处理流程，在这期间，**浏览器和服务器到底是如何打交道的？服务器又是如何处理的？浏览器又是如何将网页显示给用户的呢？……**

疑惑和细节真是太多了。坦白讲，要想彻彻底底的弄清楚以上每个疑惑和处理细节，至少需要十本书的厚度，所谓“底层无极限”嘛，而且不同的 web 服务 器和服务器端编程语言的实现和处理流程不尽相同（但本质都是相通的）。本文中，我将根据 http 协议的有关知识，跟大家讲解一些 web 开发的本质。不管你 是从事.NET，还是 J2EE 或者 php 开发等等，都离不开这些本质。希望你读完本文，能有新的收获和见解。由于本人水平和经验有限，难免有误，望读者见 谅。

何为 http 协议（Hypertext Transfer Protocol，超文本传输协议）？

所谓协议，就是指双方遵循的规范。http 协议，就是浏览器和服务器之间进行“沟通”的一种规范。我们在看空间，刷微博...都是在使用 http 协议，当然，远远不止这些应用。

笔者一直听说 http 是属于“应用层的协议”，而且是基于 TCP/IP 协议的。这个不难理解，如果你上大学时候学过“计算机网络”的课程，就一定知 道 OSI 七层参考协议（我当时是死记硬背的）。如果你接触过 socket 网络编程，就应该明白 TCP 和 UDP 这两种使用广泛的通信协议（建立连接、三次握 手等等，当然，这不是本文讨论的重点）。如图：

BAT交流群：892872246

既然 TCP/UDP 是广泛使用的网络通信协议，那为啥有多出个 http 协议来呢？

笔者曾自己动手写过一个简单的 web 服务器处理软件，根据我的推断（不一定准确）。UDP 协议具有不可靠性和不安全性，显然这很难满足 web 应用的需要。

而 TCP 协议是基于连接和三次握手的，虽然具有可靠性，但人具有一定的缺陷。但试想一下，普通的 C/S 架构软件，顶多上千个 Client 同时连接，而 B/S 架构的网站，十万人同时在线也是很平常的事儿。如果十万个客户端和服务器一直保持连接状态，那服务器如何满足承载呢？

这就衍生出了 http 协议。基于 TCP 的可靠性连接。通俗点说，就是在请求之后，服务器端立即关闭连接、释放资源。这样既保证了资源可用，也吸取了 TCP 的可靠性的优点。

正因为这点，所以大家通常说 http 协议是“无状态”的，也就是“服务器不知道你客户端干了啥”，其实很大程度上是基于性能考虑的。以至于后来有了 session 之类的玩意。

实战准备工作：

在监视网络方面，windows 平台上有一款叫做 Sniffer 的优秀软件，这也是很多“黑客”经常使用的嗅探工具。 在研究 http 协议时，推荐大家使用一款

叫作 httpwatch 的工具。（遗憾的是，该工具是收费的。该咋办就咋办，你懂的）。安装完成后，可以在 IE 浏览器的 tools 中直接打开（目前也支持 firefox）。如图所示：

BAT交流群：892872246

点击 Record，就可以开始监视并记录 http 消息了。stop、Clear 等等按钮的功能，这里就不一一介绍了。拿实例来说话，下面就是我记录访问 main.aspx 页面的时候记录的，能够清晰的看到 http 报文消息的详细信息，如图：

学习 http 协议，主要需要了解 http 的请求和响应（当然，还有 get、post 等请求方式，状态码、URI、MIME 等）

首先看看 http 请求消息（就是浏览器丢给服务器的）：

一个 http 请求代表客户端浏览器向服务器发送的数据。一个完整的 http 请求消息，包含一个请求行，若干个消息头（请求头），换行，实体内容

请求行：描述客户端的请求方式、请求资源的名称、http 协议的版本号。例如：GET/BOOK/JAVA.HTML HTTP/1.1

请求头（消息头）包含（客户机请求的服务器主机名，客户机的环境信息等）：Accept：用于告诉服务器，客户机支持的数据类型 （例如：Accept:text/html,image/*）

BAT交流群：892872246

Accept-Charset：用于告诉服务器，客户机采用的编码格式
Accept-Encoding：用于告诉服务器，客户机支持的数据压缩格式
Accept-Language：客户机语言环境
Host:客户机通过这个服务器，想访问的主机名
If-Modified-Since：客户机通过这个头告诉服务器，资源的缓存时间
Referer：客户机通过这个头告诉服务器，它（客户端）是从哪个资源来访问服务器的（防盗链）
User-Agent：客户机通过这个头告诉服务器，客户机的软件环境（操作系统，浏览器版本等）
Cookie：客户机通过这个头，将 Coockie 信息带给服务器
Connection：告诉服务器，请求完成后，是否保持连接
Date：告诉服务器，当前请求的时间

（换行）
实体内容：
就是指浏览器端通过 http 协议发送给服务器的实体数据。例如：
name=dylan&id=110
（get 请求时，通过 url 传给服务器的值。post 请求时，通过表单发送给服务器的值）

再看看 HTTP 响应消息（服务器返回给浏览器的）：

一个 http 响应代表服务器端向客户端回送的数据，它包括：
一个状态行，若干个消息头，以及实体内容

响应头(消息头)包含:
Location：这个头配合 302 状态吗，用于告诉客户端找谁
Server：服务器通过这个头，告诉浏览器服务器的类型
Content-Encoding：告诉浏览器，服务器的数据压缩格式
Content-Length：告诉浏览器，回送数据的长度
Content-Type：告诉浏览器，回送数据的类型
Last-Modified：告诉浏览器当前资源缓存时间
Refresh：告诉浏览器，隔多长时间刷新
Content-Disposition：告诉浏览器以下载的方式打开数据。例如：
context.Response.AddHeader("Content-Disposition","attachment:filename=aa.jpg");
context.Response.WriteFile("aa.jpg");
Transfer-Encoding：告诉浏览器，传送数据的编码格式
ETag：缓存相关的头（可以做到实时更新）
Expries：告诉浏览器回送的资源缓存多长时间。如果是-1 或者 0，表示不缓存
Cache-Control：控制浏览器不要缓存数据 no-cache
Pragma：控制浏览器不要缓存数据 no-cache

Connection：响应完成后，是否断开连接。 close/Keep-Alive
Date：告诉浏览器，服务器响应时间

BAT交流群：892872246

状态行： 例如： HTTP/1.1 200 OK （协议的版本号是 1.1 响应状态码为 200 响应结果为 OK）

实体内容（实体头）：响应包含浏览器能够解析的静态内容，例如：html，纯文本，图片等等信息

理解了以上的 http 请求消息和响应消息，相信你对于 http 协议已经理解得足够深刻了。关于 http 协议的更多具体细节，可以参照 **http RFC 文档**。

大致步骤就是：浏览器先向服务器发送请求，服务器接收到请求后，做相应的处理，然后封装好响应报文，再回送给浏览器。浏览器拿到响应报文后，再通过 浏览器引擎去渲染网页，解析 DOM 树，javascript 引擎解析并执行脚本操作，插件去干插件该干的事儿…关于浏览器渲染、解析的原理，可以参考 http://kb.cnblogs.com/page/129756/

说白了，所谓 web 的本质，无非是：请求/处理/响应 ，任何的 web 服务器，任何的服务端编程语言，都没法脱离这个本质。 而浏览器端解析 html、图片等静态内容，呈现给用户，脚本引擎执行脚本代码，完成脚本代码要做的事儿（例如 dom 操作，css 属性更改，发送 ajax 请 求等等）。

二、TCP/IP 协议详解

学习 http 协议，主要需要了解 http 的请求和响应（当然，还有 get、post 等请求方式，状态码、URI、MIME 等）

首先看看 http 请求消息（就是浏览器丢给服务器的）：

一个 http 请求代表客户端浏览器向服务器发送的数据。一个完整的 http 请求消息，包含一个请求行，若干个消息头（请求头），换行，实体内容

请求行：描述客户端的请求方式、请求资源的名称、http 协议的版本号。 例如：GET/BOOK/JAVA.HTML HTTP/1.1

请求头（消息头）包含（客户机请求的服务器主机名，客户机的环境信息等）：
Accept：用于告诉服务器，客户机支持的数据类型 （例如：Accept:text/html,image/*）
Accept-Charset：用于告诉服务器，客户机采用的编码格式
Accept-Encoding：用于告诉服务器，客户机支持的数据压缩格式
Accept-Language：客户机语言环境
Host:客户机通过这个服务器，想访问的主机名
If-Modified-Since：客户机通过这个头告诉服务器，资源的缓存时间
Referer：客户机通过这个头告诉服务器，它（客户端）是从哪个资源来访问服务器的（防盗链）

BAT交流群：892872246

User-Agent：客户机通过这个头告诉服务器，客户机的软件环境（操作系统，浏览器版本等）
Cookie：客户机通过这个头，将 Coockie 信息带给服务器
Connection：告诉服务器，请求完成后，是否保持连接
Date：告诉服务器，当前请求的时间

（换行）
实体内容：
就是指浏览器端通过 http 协议发送给服务器的实体数据。例如：
name=dylan&id=110
（get 请求时，通过 url 传给服务器的值。post 请求时，通过表单发送给服务器的值）

再看看 HTTP 响应消息（服务器返回给浏览器的）：

一个 http 响应代表服务器端向客户端回送的数据，它包括：
一个状态行，若干个消息头，以及实体内容

响应头(消息头)包含:
Location：这个头配合 302 状态吗，用于告诉客户端找谁
Server：服务器通过这个头，告诉浏览器服务器的类型
Content-Encoding：告诉浏览器，服务器的数据压缩格式
Content-Length：告诉浏览器，回送数据的长度
Content-Type：告诉浏览器，回送数据的类型
Last-Modified：告诉浏览器当前资源缓存时间
Refresh：告诉浏览器，隔多长时间刷新
Content-Disposition：告诉浏览器以下载的方式打开数据。例如：
context.Response.AddHeader("Content-Disposition","attachment:filename=aa.jpg");
context.Response.WriteFile("aa.jpg");
Transfer-Encoding：告诉浏览器，传送数据的编码格式
ETag：缓存相关的头（可以做到实时更新）
Expries：告诉浏览器回送的资源缓存多长时间。如果是-1 或者 0，表示不缓存
Cache-Control：控制浏览器不要缓存数据 no-cache
Pragma：控制浏览器不要缓存数据 no-cache

Connection：响应完成后，是否断开连接。 close/Keep-Alive
Date：告诉浏览器，服务器响应时间

状态行： 例如： HTTP/1.1 200 OK （协议的版本号是 1.1 响应状态码为 200 响应结果为 OK）

实体内容（实体头）：响应包含浏览器能够解析的静态内容，例如：html，纯文本，图片等等信息

BAT交流群：892872246

理解了以上的 http 请求消息和响应消息，相信你对于 http 协议已经理解得足够深刻了。关于 http 协议的更多具体细节，可以参照 **http RFC 文档**。

大致步骤就是：浏览器先向服务器发送请求，服务器接收到请求后，做相应的处理，然后封装好响应报文，再回送给浏览器。浏览器拿到响应报文后，再通过 浏览器引擎去渲染网页，解析 DOM 树，javascript 引擎解析并执行脚本操作，插件去干插件该干的事儿...关于浏览器渲染、解析的原理，可以参考 http://kb.cnblogs.com/page/129756/

说白了，所谓 web 的本质，无非是：请求/处理/响应 ，任何的 web 服务器，任何的服务端编程语言，都没法脱离这个本质。 而浏览器端解析 html、图片等静态内容，呈现给用户，脚本引擎执行脚本代码，完成脚本代码要做的事儿（例如 dom 操作，css 属性更改，发送 ajax 请 求等等）。

2.1 什么是协议？

简单来说，协议就是计算机之间通过网络实现通信时事先达成的一种“约定”；这种“约定”使那些由不同厂商的设备，不同 CPU 及不同操作系统组成的计算机之间，只要遵循相同的协议就可以实现通信。TCP/IP 通常被认为是一个四层协议，如图：

链路层:数据链路层或网络接口层，包括设备驱动程序和网络接口卡，它们一起处理与电缆的物理接口细节。
网络层:处理分组在网络中的活动，如分组的选路；网络层的协议包括 IP 协议、ICMP 协议（Internet 互联网控制报文协议）、IGMP 协议（Internet 组管理协议）。
传输层:主要为两台主机上的应用程序提供端到端的通信，包括 TCP（传输控制协议）和 UDP（用户数据报协议）。
应用层:处理特定的应用程序细节。

BAT交流群：892872246

链路层的三个主要作用：
1)为 IP 模块发送和接受 IP 数据；
2)为 ARP 模块发送 ARP 请求和接受 ARP 应答；
3)为 RARP 发送 RARP 请求和接受 RARP 应答。

TCP 创建过程和链接折除过程是由 TCP/IP 协议栈自动创建的，所以开发者并不需要理解得十分透彻，只需要对 TCP 底层运作机制有一个大致理解即可，这一块有些复杂，先做个笔记，以后有需要，再做详细扩展

拓展：**检测网络是否联通的命令——Ping**
Ping 的目的是为了测试另一台主机是否可达，该命令会发送一份 ICMP 回显请求报文给主机，并等待返回 ICMP 回显应答。
Ping 程序还能测出到这台主机的往返时间，大多数的 TCP/IP 实现都在内核中直接支持 Ping 服务，这种服务器不是一个用户进程。Ping 程序通过在 ICMP 报文数据中存放请求的时间值来计算往返时间。当应答返回时，用当前时间减去存放在 ICMP 报文中的时间值，既是往返时间。

BAT交流群：892872246

2.2 什么是传输包？

将需要传输的数据分割为一个个小的单位进行传输的方法，就叫做包，如图:

通信过程中，每一个分组中都会附加上源主机地址和目标主机地址送给通信线路，这些发送端地址、接收端地址以及分组序号写入的部分就是“报文首部”；如果数据较大，分组过多，那么就有必要将分组序号写入包中，接收端会根据序号，分组按序重新装配为原始数据。(理解丢包就需要知道什么是包)

2.3 什么是“三次握手，四次挥手”？

TCP 是一种面向连接的单播协议，在发送数据前，通信双方必须在彼此间建立一条连接。该“连接”就是客户端和服务器的内存里保存的关于 ip 地址、端口号等对方的信息。TCP 可以看成是一种字节流，它会处理 IP 层或以下的层的丢包、重复以及错误问题。在连接的建立过程中，双方需要交换一些连接的参数。这些参数可以放在 TCP 头部。

TCP 提供了一种可靠、面向连接、字节流、传输层的服务，采用三次握手建立一个连接，采用 4 次挥手来关闭一个连接。

2.4 为什么要“三次握手，四次挥手”？

1）所谓三次握手(Three-way Handshake)，是指建立一个 TCP 连接时，需要客户端和服务器总共发送 3 个包。三次握手的目的是连接服务器指定端口，建立 TCP 连接,并同步连接双方的序列号和确认号并交换 TCP 窗口大小信息。在 socket

BAT交流群：892872246

编程中，客户端执行 connect()时，将触发三次握手：即客户端和服务端通信前要进行连接，“3 次握手”的作用就是双方都能明确自己和对方的收、发能力正常。

第一次握手：客户端发送网络包，服务端收到。服务端得出结论：客户端发送能力、服务端接收能力正常。

第二次握手：服务端发包，客户端收到。客户端得出结论：服务端接收、发送能力正常，客户端接收、发送能力正常。

第三次握手：客户端发包，服务端收到。服务端得出结论：客户端接收、发送能力正常，服务端发送、接收能力正常。
所以第一、二次握手，只是为了验证彼此的接收发送能力是否正常。经历上面的三次握手过程，客户端和服务端都确认了自己的接收、发送能力是正常的，就可以正常通信了。

2）所谓四次挥手(four-way handshake)，就是指 TCP 的连接的拆除过程需要发送四个包，客户端或服务器均可主动发起挥手动作，在 socket 编程中，任何一方执行 close()操作即可产生挥手操作。

BAT交流群：892872246

TCP 连接是双向传输的对等的模式，就是说双方都可以同时向对方发送或接收数据。客户端或者服务端皆可以主动发送指令告诉对方，我要关闭连接了，比如：
客户端：我要关闭连接了（客户端发起 Close）
服务端：好，你关吧（服务端收到关闭指令，回了一个 ACK）【此时客户端不可以向服务端传输数据，但是服务端仍然可以向客户端传输数据】
服务端：我传完所有数据啦（发送一个 FIN 段来关闭向客户端传输方向上的连接）
客户端：哦，好的，知道（发送 ACK 确认关闭连接）

注意：接收到 FIN 报文的一方只能回复一个 ACK，它是无法马上返回对方一个 FIN 报文段的，因为结束数据传输的“指令”是上层应用层给出的，它只负责“搬运”。

参考 https://www.cnblogs.com/haiyan123/p/7777924.html
https://www.jianshu.com/p/8e4eb47421f5

阿里 P7 移动互联网架构师进阶视频（每日更新中）免费学习请点击：
https://space.bilibili.com/474380680

结束语

希望读到这的您能**转发分享和关注一下我**，以后还会分享阿里 P7 Android 知识点及解析，您的支持就是我最大的动力！！

BAT交流群：892872246

5.1.3 SSL 握手与加密

本篇文章将继续从以下两个内容来介绍网络通信必备基础:

- [SSL 握手过程详解]
- [DNS 解析]

一、SSL 握手过程详解

1.1、客户端发出加密通信请求 ClientHello

提供：
1，协议版本（如 TSL1.0）
2，随机数 1（用于生成对话密钥）
3，支持的加密方法（如 RSA 公钥加密）
4，支持的压缩方法

1.2、服务器回应 SeverHello

回应内容：
1，确认使用的加密通信协议版本（TSL1.0）
2，随机数 2（用于生成对话密钥）

BAT交流群：892872246

3，确认加密方法（RSA）
4，服务器证书（包含非对称加密的公钥）
5，（可选）要求客户端提供证书的请求

1.3、客户端验证证书

如果证书不是可信机构颁布，或证书域名与实际域名不符，或者证书已经过期，就会向访问者显示一个警告，是否继续通信

1.4 、客户端回应

证书没有问题，就会取出证书中的服务器公钥
然后发送：
1，随机数 3（pre-master key，此随机数用服务器公钥加密，防止被窃听）
2，编码改变通知（表示随后的信息都将用双方商定的方法和密钥发送）
3，客户端握手结束通知

1.5、双方生成会话密钥

双方同时有了三个随机数，接着就用事先商定的加密方法，各自生成同一把“会话密钥”
服务器端用自己的私钥（非对称加密的）获取第三个随机数，会计算生成本次所用的会话密钥（对称加密的密钥），如果前一步要求客户端证书，会在这一步验证

1.6、服务器最后响应

服务器生成会话密钥后，向客户端发送：
1，编码改变通知（后面的信息都用双方的加密方法和密钥来发送）
2，服务器握手结束通知

至此，握手阶段全部结束，接下来客户端与服务器进入加密通信，用会话密钥加密内容

二、DNS 解析

我们知道，网络中的大部分通讯都是基于 TCP/IP 协议的，而 TCP/IP 协议中有个很重要的协议——IP 协议，它是基于 IP 地址的。我们无法记住大量的 IP 地址，所以就需要 DNS 服务器能把域名翻译成 IP 地址。

BAT交流群：892872246

2.1、什么是 DNS?

它所提供的服务是用来将主机名和域名转换为 IP 地址的工作。我们知道域名和 IP 地址是一一对应的关系，但是多个域名可以对应同一个 IP 地址。

2.2、DNS 查询过程

递归：DNS 服务器可使用其自身的资源记录信息缓存来应答查询，也可代表请求客户机来查询或联系其他 DNS 服务器，以完全解析该名称，并随后将应答返回至客户机。

迭代：客户机自己也可尝试 I 联系其他的 DNS 服务器来解析名称。如果客户机这么做，它会使用基于服务器应答的独立和附加的查询。

```
1、在浏览器中输入域名，操作系统会先检查自己本地的 hosts 文件是否有这个网址映射关系。
```

892872246

2、如果 hosts 里没有映射关系，查找本地 DNS 解析器缓存。

3、如果还是没有，首先会找 TCP/IP 参数中设置的首选 DNS 服务器。

参考 https://blog.csdn.net/colinandroid/article/details/79327847
阿里 P7 移动互联网架构师进阶视频（每日更新中）免费学习请点击：
https://space.bilibili.com/474380680

5.1.4 DNS 解析

阿里 P7 移动互联网架构师进阶视频（每日更新中）免费学习请点击：
https://space.bilibili.com/474380680
本篇文章将继续从以下两个内容来介绍网络通信必备基础:

- [SSL 握手过程详解]
- [DNS 解析]

一、SSL 握手过程详解

1.1、客户端发出加密通信请求 ClientHello

提供：
1，协议版本（如 TSL1.0）
2，随机数 1（用于生成对话密钥）

BAT交流群：892872246

3，支持的加密方法（如 RSA 公钥加密）
4，支持的压缩方法

1.2、服务器回应 SeverHello

回应内容：
1，确认使用的加密通信协议版本（TSL1.0）
2，随机数 2（用于生成对话密钥）
3，确认加密方法（RSA）
4，服务器证书（包含非对称加密的公钥）
5，（可选）要求客户端提供证书的请求

1.3、客户端验证证书

如果证书不是可信机构颁布，或证书域名与实际域名不符，或者证书已经过期，就会向访问者显示一个警告，是否继续通信

1.4 、客户端回应

证书没有问题，就会取出证书中的服务器公钥
然后发送：
1，随机数 3（pre-master key，此随机数用服务器公钥加密，防止被窃听）
2，编码改变通知（表示随后的信息都将用双方商定的方法和密钥发送）
3，客户端握手结束通知

1.5、双方生成会话密钥

双方同时有了三个随机数，接着就用事先商定的加密方法，各自生成同一把“会话密钥”
服务器端用自己的私钥（非对称加密的）获取第三个随机数，会计算生成本次所用的会话密钥（对称加密的密钥），如果前一步要求客户端证书，会在这一步验证

1.6、服务器最后响应

服务器生成会话密钥后，向客户端发送：
1，编码改变通知（后面的信息都用双方的加密方法和密钥来发送）
2，服务器握手结束通知

BAT交流群：892872246

至此，握手阶段全部结束，接下来客户端与服务器进入加密通信，用会话密钥加密内容

二、DNS 解析

我们知道，网络中的大部分通讯都是基于 TCP/IP 协议的，而 TCP/IP 协议中有个很重要的协议——IP 协议，它是基于 IP 地址的。我们无法记住大量的 IP 地址，所以就需要 DNS 服务器能把域名翻译成 IP 地址。

2.1、什么是 DNS?

它所提供的服务是用来将主机名和域名转换为 IP 地址的工作。我们知道域名和 IP 地址是一一对应的关系，但是多个域名可以对应同一个 IP 地址

BAT交流群：892872246

2.2、DNS 查询过程

递归：DNS 服务器可使用其自身的资源记录信息缓存来应答查询，也可代表请求客户机来查询或联系其他 DNS 服务器，以完全解析该名称，并随后将应答返回至客户机。

迭代：客户机自己也可尝试 I 联系其他的 DNS 服务器来解析名称。如果客户机这么做，它会使用基于服务器应答的独立和附加的查询。

```
1、在浏览器中输入域名，操作系统会先检查自己本地的 hosts 文件是否有这个网址映射关系。



2、如果 hosts 里没有映射关系，查找本地 DNS 解析器缓存。



3、如果还是没有，首先会找 TCP/IP 参数中设置的首选 DNS 服务器。
```

参考 https://blog.csdn.net/colinandroid/article/details/79327847
阿里 P7 移动互联网架构师进阶视频（每日更新中）免费学习请点击：
https://space.bilibili.com/474380680

5.1.5 Socket 通信原则

本篇文章将最后从 Socket 通信原则以下两个内容来介绍网络通信必备基础:

BAT交流群：892872246

- [SOCKS 代理]
- [HTTP 普通代理&隧道代理]

一、SOCKS 代理

全能代理，就像有很多跳线的转接板，它只是简单地将一端的系统连接到另外一端。支持多种协议，包括 http、ftp 请求及其它类型的请求。它分 socks 4 和 socks 5 两种类型，socks 4 只支持 TCP 协议而 socks 5 支持 TCP/UDP 协议，还支持各种身份验证机制等协议。其标准端口为 1080。

1.1 原理

socks 代理相应的采用 socks 协议的代理服务器就是 SOCKS 服务器，是一种通用的代理服务器。Socks 是个电路级的底层网关，是 DavidKoblas 在 1990 年开发的，此后就一直作为 Internet RFC 标准的开放标准。Socks 不要求应用程序遵循特定的操作系统平台，Socks 代理与应用层代理、 HTTP 层代理不同，Socks 代理只是简单地传递数据包，而不必关心是何种应用协议（比如 FTP、HTTP 和 NNTP 请求）。所以，Socks 代理比其他应用层代理要快得多。它通常绑定在代理服务器的 1080 端口上。如果您在企业网或校园网上，需要透过防火墙或通过代理服务器访问 Internet 就可能需要使用 SOCKS。一般情况下，对于拨号上网用户都不需要使用它。注意，浏览网页时常用的代理服务器通常是专门的 http 代理，它和 SOCKS 是不同的。因此，您能浏览网页不等于您一定可以通过 SOCKS 访问 Internet。 常用的防火墙，或代理软件都支持 SOCKS，但需要其管理员打开这一功能。如果您不确信您是否需要 SOCKS 或是否有 SOCKS 可用，请与您的网络管理员联系。为了使用 socks，您需要了解一下内容：

① SOCKS 服务器的 IP 地址

② SOCKS 服务所在的端口

③ 这个 SOCKS 服务是否需要用户认证？如果需要，您要向您的网络管理员申请一个用户和口令

知道了上述信息，您就可以把这些信息填入“网络配置”中，或者在第一次登记时填入，您就可以使用 socks 代理了。

1.2 实际应用

在实际应用中 SOCKS 代理可以用作为：电子邮件、新闻组软件、网络传呼 ICQ、网络聊天 MIRC 和使用代理服务器打游戏等等各种应用软件当中。

BAT交流群：892872246

二、HTTP 普通代理&隧道代理

2.1 普通代理

第一种 Web 代理原理特别简单：

HTTP 客户端向代理发送请求报文，代理服务器需要正确地处理请求和连接（例如正确处理 Connection: keep-alive），同时向服务器发送请求，并将收到的响应转发给客户端。

下面这张图片来自于《HTTP 权威指南》，直观地展示了上述行为：

假如我通过代理访问 A 网站，对于 A 来说，它会把代理当做客户端，完全察觉不到真正客户端的存在，这实现了隐藏客户端 IP 的目的。当然代理也可以修改 HTTP 请求头部，通过 X-Forwarded-IP 这样的自定义头部告诉服务端真正的客户端 IP。但服务器无法验证这个自定义头部真的是由代理添加，还是客户端修改了请求头，所以从 HTTP 头部字段获取 IP 时，需要格外小心。这部分内容可以参考我之前的《HTTP 请求头中的 X-Forwarded-For》这篇文章。

给浏览器显式的指定代理，需要手动修改浏览器或操作系统相关设置，或者指定 PAC（Proxy Auto-Configuration，自动配置代理）文件自动设置，还有些浏览器支持 WPAD（Web Proxy Autodiscovery Protocol，Web 代理自动发现协议）。显式指定浏览器代理这种方式一般称之为正向代理，浏览器启用正向代理后，会对 HTTP 请求报文做一些修改，来规避老旧代理服务器的一些问题，这部分内容可以参考我之前的《Http 请求头中的 Proxy-Connection》这篇文章。

还有一种情况是访问 A 网站时，实际上访问的是代理，代理收到请求报文后，再向真正提供服务的服务器发起请求，并将响应转发给浏览器。这种情况一般被称之为反向代理，它可以用来隐藏服务器 IP 及端口。一般使用反向代理后，需

BAT技术群：892872246

要通过修改 DNS 让域名解析到代理服务器 IP，这时浏览器无法察觉到真正服务器的存在，当然也就不需要修改配置了。反向代理是 Web 系统最为常见的一种部署方式，例如本博客就是使用 Nginx 的 `proxy_pass` 功能将浏览器请求转发到背后的 Node.js 服务。

了解完第一种代理的基本原理后，我们用 Node.js 实现一下它。只包含核心逻辑的代码如下：

```
var http = require('http');var net = require('net');var url = require('u
rl');

function request(cReq, cRes) {

    var u = url.parse(cReq.url);



    var options = {

        hostname : u.hostname,

        port     : u.port || 80,

        path     : u.path,

        method     : cReq.method,

        headers     : cReq.headers

    };



    var pReq = http.request(options, function(pRes) {

        cRes.writeHead(pRes.statusCode, pRes.headers);

        pRes.pipe(cRes);

    }).on('error', function(e) {

        cRes.end();

    });
```

```
    cReq.pipe(pReq);}

http.createServer().on('request', request).listen(8888, '0.0.0.0');
```

以上代码运行后，会在本地 8888 端口开启 HTTP 代理服务，这个服务从请求报文中解析出请求 URL 和其他必要参数，新建到服务端的请求，并把代理收到的请求转发给新建的请求，最后再把服务端响应返回给浏览器。修改浏览器的 HTTP 代理为 127.0.0.1:8888 后再访问 HTTP 网站，代理可以正常工作。

但是，使用我们这个代理服务后，HTTPS 网站完全无法访问，这是为什么呢？答案很简单，这个代理提供的是 HTTP 服务，根本没办法承载 HTTPS 服务。那么是否把这个代理改为 HTTPS 就可以了呢？显然也不可以，因为这种代理的本质是中间人，而 HTTPS 网站的证书认证机制是中间人劫持的克星。普通的 HTTPS 服务中，服务端不验证客户端的证书，中间人可以作为客户端与服务端成功完成 TLS 握手；但是中间人没有证书私钥，无论如何也无法伪造成服务端跟客户端建立 TLS 连接。当然如果你拥有证书私钥，代理证书对应的 HTTPS 网站当然就没问题了。

HTTP 抓包神器 Fiddler 的工作原理也是在本地开启 HTTP 代理服务，通过让浏览器流量走这个代理，从而实现显示和修改 HTTP 包的功能。如果要让 Fiddler 解密 HTTPS 包的内容，需要先将它自带的根证书导入到系统受信任的根证书列表中。一旦完成这一步，浏览器就会信任 Fiddler 后续的「伪造证书」，从而在浏览器和 Fiddler、Fiddler 和服务端之间都能成功建立 TLS 连接。而对于 Fiddler 这个节点来说，两端的 TLS 流量都是可以解密的。

如果我们不导入根证书，Fiddler 的 HTTP 代理还能代理 HTTPS 流量么？实践证明，不导入根证书，Fiddler 只是无法解密 HTTPS 流量，HTTPS 网站还是可以正常访问。这是如何做到的，这些 HTTPS 流量是否安全呢？这些问题将在下一节揭晓。

2.2 隧道代理

第二种 Web 代理的原理也很简单：

> HTTP 客户端通过 CONNECT 方法请求隧道代理创建一条到达任意目的服务器和端口的 TCP 连接，并对客户端和服务器之间的后继数据进行盲转发。

下面这张图片同样来自于《HTTP 权威指南》，直观地展示了上述行为：

假如我通过代理访问 A 网站，浏览器首先通过 CONNECT 请求，让代理创建一条到 A 网站的 TCP 连接；一旦 TCP 连接建好，代理无脑转发后续流量即可。所以这种代理，理论上适用于任意基于 TCP 的应用层协议，HTTPS 网站使用的 TLS 协议当然也可以。这也是这种代理为什么被称为隧道的原因。对于 HTTPS 来说，客户端透过代理直接跟服务端进行 TLS 握手协商密钥，所以依然是安全的，下图中的抓包信息显示了这种场景：

BAT交流群：892872246

可以看到，浏览器与代理进行 TCP 握手之后，发起了 CONNECT 请求，报文起始行如下：

CONNECT imququ.com:443 HTTP/1.1

对于 CONNECT 请求来说，只是用来让代理创建 TCP 连接，所以只需要提供服务器域名及端口即可，并不需要具体的资源路径。代理收到这样的请求后，需要与服务端建立 TCP 连接，并响应给浏览器这样一个 HTTP 报文：

HTTP/1.1 200 Connection Established

浏览器收到了这个响应报文，就可以认为到服务端的 TCP 连接已经打通，后续直接往这个 TCP 连接写协议数据即可。通过 Wireshark 的 Follow TCP Steam 功能，可以清楚地看到浏览器和代理之间的数据传递：

BAT交流群：892872246

可以看到，浏览器建立到服务端 TCP 连接产生的 HTTP 往返，完全是明文，这也是为什么 CONNECT 请求只需要提供域名和端口：如果发送了完整 URL、Cookie 等信息，会被中间人一览无余，降低了 HTTPS 的安全性。HTTP 代理承载的 HTTPS 流量，应用数据要等到 TLS 握手成功之后通过 Application Data 协议传输，中间节点无法得知用于流量加密的 master-secret，无法解密数据。而 CONNECT 暴露的域名和端口，对于普通的 HTTPS 请求来说，中间人一样可以拿到（IP 和端口很容易拿到，请求的域名可以通过 DNS Query 或者 TLS Client Hello 中的 Server Name Indication 拿到），所以这种方式并没有增加不安全性。

了解完原理后，再用 Node.js 实现一个支持 CONNECT 的代理也很简单。核心代码如下：

```
var http = require('http');var net = require('net');var url = require('u
rl');

function connect(cReq, cSock) {

    var u = url.parse('http://' + cReq.url);



    var pSock = net.connect(u.port, u.hostname, function() {

        cSock.write('HTTP/1.1 200 Connection Established\r\n\r\n');
```

```
        pSock.pipe(cSock);

    }).on('error', function(e) {

        cSock.end();

    });



    cSock.pipe(pSock);}



http.createServer().on('connect', connect).listen(8888, '0.0.0.0');
```

以上代码运行后，会在本地 8888 端口开启 HTTP 代理服务，这个服务从 CONNECT 请求报文中解析出域名和端口，创建到服务端的 TCP 连接，并和 CONNECT 请求中的 TCP 连接串起来，最后再响应一个 Connection Established 响应。修改浏览器的 HTTP 代理为 127.0.0.1:8888 后再访问 HTTPS 网站，代理可以正常工作。

最后，将两种代理的实现代码合二为一，就可以得到全功能的 Proxy 程序了，全部代码在 50 行以内（当然异常什么的基本没考虑，这是我博客代码的一贯风格）：

```
var http = require('http');var net = require('net');var url = require('u
rl');

function request(cReq, cRes) {

    var u = url.parse(cReq.url);



    var options = {

        hostname : u.hostname,

        port     : u.port || 80,

        path     : u.path,

        method     : cReq.method,
```

```
        headers     : cReq.headers
    };

    var pReq = http.request(options, function(pRes) {
        cRes.writeHead(pRes.statusCode, pRes.headers);
        pRes.pipe(cRes);
    }).on('error', function(e) {
        cRes.end();
    });

    cReq.pipe(pReq);}
function connect(cReq, cSock) {
    var u = url.parse('http://' + cReq.url);

    var pSock = net.connect(u.port, u.hostname, function() {
        cSock.write('HTTP/1.1 200 Connection Established\r\n\r\n');
        pSock.pipe(cSock);
    }).on('error', function(e) {
        cSock.end();
    });

    cSock.pipe(pSock);}
```

```
http.createServer()

    .on('request', request)

    .on('connect', connect)

    .listen(8888, '0.0.0.0');
```

需要注意的是，大部分浏览器显式配置了代理之后，只会让 HTTPS 网站走隧道代理，这是因为建立隧道需要耗费一次往返，能不用就尽量不用。但这并不代表 HTTP 请求不能走隧道代理，我们用 Node.js 写段程序验证下（先运行前面的代理服务）：

```
var http = require('http');

var options = {

    hostname : '127.0.0.1',

    port     : 8888,

    path     : 'imququ.com:80',

    method     : 'CONNECT'};

var req = http.request(options);



req.on('connect', function(res, socket) {

    socket.write('GET / HTTP/1.1\r\n' +

                 'Host: imququ.com\r\n' +

                 'Connection: Close\r\n' +

                 '\r\n');



    socket.on('data', function(chunk) {

        console.log(chunk.toString());
```

BAT交 892246

```
    });
```
可以看到，通过 CONNECT 让代理打开到目标服务器的 TCP 连接，用来承载 HTTP 流量也是完全没问题的。

最后，HTTP 的认证机制可以跟代理配合使用，使得必须输入正确的用户名和密码才能使用代理，这部分内容比较简单，这里略过。
参考 https://www.cnblogs.com/be-bright/p/5680599.html
https://baike.baidu.com/item/Socks 代理/10627160?fr=aladdin
阿里 P7 移动互联网架构师进阶视频（每日更新中）免费学习请点击：
https://space.bilibili.com/474380680

作者：Alvin 老师
链接：https://www.jianshu.com/p/02b7810c3289
来源：简书
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

```
    socket.on('end', function() {
        console.log('socket end.');
    });});


req.end();
```

这段代码运行完，结果如下：

```
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Thu, 19 Nov 2015 15:57:47 GMT
Content-Type: text/html
Content-Length: 178
Connection: close
Location: https://imququ.com/
```

```
<html><head><title>301 Moved Permanently</title></head><body bgcolor="white"><center><h1>301 Moved Permanently</h1></center><hr><center>nginx</center></body></html>



socket end.
```

可以看到，通过 CONNECT 让代理打开到目标服务器的 TCP 连接，用来承载 HTTP 流量也是完全没问题的。

最后，HTTP 的认证机制可以跟代理配合使用，使得必须输入正确的用户名和密码才能使用代理，这部分内容比较简单，这里略过。
参考 https://www.cnblogs.com/be-bright/p/5680599.html
https://baike.baidu.com/item/Socks 代理/10627160?fr=aladdin
阿里 P7 移动互联网架构师进阶视频（每日更新中）免费学习请点击：
https://space.bilibili.com/474380680

5.1.5.1 SOCKS 代理

5.1.5.2 HTTP 普通代理与隧道代理

5.2 OkHttp 源码解读

5.2.1 Socket 连接池复用机制

1、概述

提高网络性能优化，很重要的一点就是降低延迟和提升响应速度。

通常我们在浏览器中发起请求的时候 header 部分往往是这样的

BAT交流群：892872246

keep-alive 就是浏览器和服务端之间保持长连接，这个连接是可以复用的。在 HTTP1.1 中是默认开启的。

连接的复用为什么会提高性能呢？
通常我们在发起 http 请求的时候首先要完成 tcp 的三次握手，然后传输数据，最后再释放连接。三次握手的过程可以参考这里 TCP 三次握手详解及释放连接过程

一次响应的过程

在高并发的请求连接情况下或者同个客户端多次频繁的请求操作，无限制的创建会导致性能低下。

如果使用 keep-alive

在 timeout 空闲时间内，连接不会关闭，相同重复的 request 将复用原先的 connection，减少握手的次数，大幅提高效率。

并非 keep-alive 的 timeout 设置时间越长，就越能提升性能。长久不关闭会造成过多的僵尸连接和泄露连接出现。

那么 okttp 在客户端是如果类似于客户端做到的 keep-alive 的机制。

BAT交流群：892872246

2、连接池的使用

连接池的类位于 okhttp3.ConnectionPool。我们的主旨是了解到如何在 timeout 时间内复用 connection，并且有效的对其进行回收清理操作。

其成员变量代码片

```
/**

 * Background threads are used to cleanup expired connections. There wil
l be at most a single

 * thread running per connection pool. The thread pool executor permits t
he pool itself to be

 * garbage collected.

   */

  private static final Executor executor = new ThreadPoolExecutor(0 /* c
orePoolSize */,

      Integer.MAX_VALUE /* maximumPoolSize */, 60L /* keepAliveTime */, T
imeUnit.SECONDS,

      new SynchronousQueue<Runnable>(), Util.threadFactory("OkHttp Conne
ctionPool", true));



  /** The maximum number of idle connections for each address. */

  private final int maxIdleConnections;



  private final Deque<RealConnection> connections = new ArrayDeque<>();

  final RouteDatabase routeDatabase = new RouteDatabase();

  boolean cleanupRunning;
```

excutor：线程池，用来检测闲置 socket 并对其进行清理。
connections：connection 缓存池。Deque 是一个双端列表，支持在头尾插入元素，

这里用作 LIFO（后进先出）堆栈，多用于缓存数据。
routeDatabase ：用来记录连接失败 router

2.1 缓存操作

ConnectionPool 提供对 Deque<RealConnection>进行操作的方法分别为 put、get、connectionBecameIdle、evictAll 几个操作。分别对应放入连接、获取连接、移除连接、移除所有连接操作。

put 操作

```
void put(RealConnection connection) {

    assert (Thread.holdsLock(this));

    if (!cleanupRunning) {

      cleanupRunning = true;

      executor.execute(cleanupRunnable);

    }

    connections.add(connection);

  }
```

可以看到在新的 connection 放进列表之前执行清理闲置连接的线程。

既然是复用，那么看下他获取连接的方式。

```
/** Returns a recycled connection to {@code address}, or null if no such
connection exists. */

RealConnection get(Address address, StreamAllocation streamAllocation) {

    assert (Thread.holdsLock(this));

    for (RealConnection connection : connections) {

      if (connection.allocations.size() < connection.allocationLimit

          && address.equals(connection.route().address)
```

```
        && !connection.noNewStreams) {

      streamAllocation.acquire(connection);

      return connection;

    }

  }

  return null;

}
```

遍历 connections 缓存列表，当某个连接计数的次数小于限制的大小以及 request 的地址和缓存列表中此连接的地址完全匹配。则直接复用缓存列表中的 connection 作为 request 的连接。

streamAllocation.allocations 是个对象计数器，其本质是一个 List<Reference<StreamAllocation>> 存放在 RealConnection 连接对象中用于记录 Connection 的活跃情况。

连接池中 Connection 的缓存比较简单，就是利用一个双端列表，配合 CRD 等操作。那么 connection 在 timeout 时间类是如果失效的呢，并且如果做到有效的对连接进行清除操作以确保性能和内存空间的充足。

2.2 连接池的清理和回收

在看 ConnectionPool 的成员变量的时候我们了解到一个 Executor 的线程池是用来清理闲置的连接的。注释中是这么解释的：

Background threads are used to cleanup expired connections

我们在 put 新连接到队列的时候会先执行清理闲置连接的线程。调用的正是 executor.execute(cleanupRunnable); 方法。观察 cleanupRunnable

```
private final Runnable cleanupRunnable = new Runnable() {

  @Override public void run() {

    while (true) {

      long waitNanos = cleanup(System.nanoTime());
```

BA 2246

```
      if (waitNanos == -1) return;

      if (waitNanos > 0) {

        long waitMillis = waitNanos / 1000000L;

        waitNanos -= (waitMillis * 1000000L);

        synchronized (ConnectionPool.this) {

          try {

            ConnectionPool.this.wait(waitMillis, (int) waitNanos);

          } catch (InterruptedException ignored) {

          }

        }

      }

    }

  }

};
```

线程中不停调用 Cleanup 清理的动作并立即返回下次清理的间隔时间。继而进入 wait 等待之后释放锁，继续执行下一次的清理。所以可能理解成他是个监测时间并释放连接的后台线程。

了解 cleanup 动作的过程。这里就是如何清理所谓闲置连接的和行了。怎么找到闲置的连接是主要解决的问题。

```
long cleanup(long now) {

    int inUseConnectionCount = 0;

    int idleConnectionCount = 0;

    RealConnection longestIdleConnection = null;

    long longestIdleDurationNs = Long.MIN_VALUE;
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
    // Find either a connection to evict, or the time that the next evict
ion is due.
    synchronized (this) {
      for (Iterator<RealConnection> i = connections.iterator(); i.hasNex
t(); ) {
        RealConnection connection = i.next();

        // If the connection is in use, keep searching.
        if (pruneAndGetAllocationCount(connection, now) > 0) {
          inUseConnectionCount++;
          continue;
        }

        idleConnectionCount++;

        // If the connection is ready to be evicted, we're done.
        long idleDurationNs = now - connection.idleAtNanos;
        if (idleDurationNs > longestIdleDurationNs) {
          longestIdleDurationNs = idleDurationNs;
          longestIdleConnection = connection;
        }
      }
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
      if (longestIdleDurationNs >= this.keepAliveDurationNs

          || idleConnectionCount > this.maxIdleConnections) {

        // We've found a connection to evict. Remove it from the list, th
en close it below (outside

        // of the synchronized block).

        connections.remove(longestIdleConnection);

      } else if (idleConnectionCount > 0) {

        // A connection will be ready to evict soon.

        return keepAliveDurationNs - longestIdleDurationNs;

      } else if (inUseConnectionCount > 0) {

        // All connections are in use. It'll be at least the keep alive d
uration 'til we run again.

        return keepAliveDurationNs;

      } else {

        // No connections, idle or in use.

        cleanupRunning = false;

        return -1;

      }

    }



    closeQuietly(longestIdleConnection.socket());



    // Cleanup again immediately.

    return 0;
```

```
  }
```

在遍历缓存列表的过程中，使用连接数目 inUseConnectionCount 和闲置连接数目 idleConnectionCount 的计数累加值都是通过 pruneAndGetAllocationCount() 是否大于 0 来控制的。那么很显然 pruneAndGetAllocationCount() 方法就是用来识别对应连接是否闲置的。>0 则不闲置。否则就是闲置的连接。

进去观察

```
private int pruneAndGetAllocationCount(RealConnection connection, long n
ow) {

    List<Reference<StreamAllocation>> references = connection.allocation
s;

    for (int i = 0; i < references.size(); ) {

      Reference<StreamAllocation> reference = references.get(i);



      if (reference.get() != null) {

        i++;

        continue;

      }



      // We've discovered a leaked allocation. This is an application bu
g.

      Platform.get().log(WARN, "A connection to " + connection.route().ad
dress().url()

          + " was leaked. Did you forget to close a response body?", nul
l);

      references.remove(i);

      connection.noNewStreams = true;
```

```
    // If this was the last allocation, the connection is eligible for
immediate eviction.

    if (references.isEmpty()) {

      connection.idleAtNanos = now - keepAliveDurationNs;

      return 0;

    }

  }



  return references.size();

 }}
```

好了，原先存放在 RealConnection 中的 allocations 派上用场了。遍历 StreamAllocation 弱引用链表，移除为空的引用，遍历结束后返回链表中弱引用的数量。所以可以看出 List<Reference<StreamAllocation>> 就是一个记录 connection 活跃情况的 >0 表示活跃 =0 表示空闲。StreamAllocation 在列表中的数量就是就是物理 socket 被引用的次数

解释：StreamAllocation 被高层反复执行 aquire 与 release。这两个函数在执行过程中其实是在一直在改变 Connection 中的 List<WeakReference<StreamAllocation>>大小。

搞定了查找闲置的 connection 操作，我们回到 cleanup 的操作。计算了 inUseConnectionCount 和 idleConnectionCount 之后程序又根据闲置时间对 connection 进行了一个选择排序，选择排序的核心是：

```
// If the connection is ready to be evicted, we're done.

      long idleDurationNs = now - connection.idleAtNanos;

      if (idleDurationNs > longestIdleDurationNs) {

        longestIdleDurationNs = idleDurationNs;

        longestIdleConnection = connection;

      }
```

```
    }

  ....
```

通过对比最大闲置时间选择排序可以方便的查找出闲置时间最长的一个connection。如此一来我们就可以移除这个没用的 connection 了！

```
if (longestIdleDurationNs >= this.keepAliveDurationNs

        || idleConnectionCount > this.maxIdleConnections) {

      // We've found a connection to evict. Remove it from the list, th
en close it below (outside

      // of the synchronized block).

      connections.remove(longestIdleConnection);}
```

总结：清理闲置连接的核心主要是引用计数器 List<Reference<StreamAllocation>> 和 选择排序的算法以及 excutor 的清理线程池。
原文链接 https://www.cnblogs.com/ganchuanpu/p/9408081.html
阿里 P7 移动互联网架构师进阶视频（每日更新中）免费学习请点击：
https://space.bilibili.com/474380680

5.2.2 HTTP 协议重定向与缓存处理

阿里 P7 移动互联网架构师进阶视频（每日更新中）免费学习请点击：
https://space.bilibili.com/474380680
本篇文章将继续从以下两个内容来解析 OkHttp3 源码:

- [HTTP 重定向]
- [缓存的处理]

一、HTTP 重定向

1.1 重定向

> 重定向(Redirect)就是通过各种方法将各种网络请求重新定个方向转到其它位置（如：网页重定向、域名的重定向、路由选择的变化也是对数据报文经由路径的一种重定向）。 ----百度百科

BAT交流群：892872246

1.2 原理

在 HTTP 协议中，重定向操作由服务器通过发送特殊的响应（即 redirects）而触发。HTTP 协议的重定向响应的状态码为 3xx 。浏览器在接收到重定向响应的时候，会采用该响应提供的新的 URL ，并立即进行加载；大多数情况下，除了会有一小部分性能损失之外，重定向操作对于用户来说是不可见的。

不同类型的重定向映射可以划分为三个类别：永久重定向、临时重定向和特殊重定向。

1.3 永久重定向

这种重定向操作是永久性的。它表示原 URL 不应再被使用，而应该优先选用新的 URL。搜索引擎机器人会在遇到该状态码时触发更新操作，在其索引库中修改与该资源相关的 URL 。

注意了：浏览器自动重定向，不管代码怎么写都会自动重定向到第一次你永久重定向的 URL，没办法，这时候你只能清楚浏览器缓存

BAT交流群：892872246

编码	含义	处理方法	典型应用场景
301	Moved Permanently	GET 方法不会发生变更，其他方法有可能会变更为 GET 方法。	网站重构。
308	Permanent Redirect	方法和消息主体都不发生变化。	网站重构。with non-GET links/operations(?)

1.4 临时重定向

> 有时候请求的资源无法从其标准地址访问，但是却可以从另外的地方访问。在这种情况下可以使用临时重定向。搜索引擎不会记录该新的、临时的链接。在创建、更新或者删除资源的时候，临时重定向也可以用于显示临时性的进度页面。
> **那么临时重定向就是解决自己傻乎乎的用永久重定向的问题。**

编码	含义	处理方法	典型应用场景
302	Found	GET 方法不会发生变更，其他方法有可能会变更为 GET 方法。	由于不可预见的原因该页面暂不可用。在这种情况下，搜索引擎不会更新它们的链接。
303	See Other	GET 方法不会发生变更，其他方法会变更为 GET 方法（消息主体会丢失）。	用于PUT 或 POST 请求完成之后进行页面跳转来防止由于页面刷新导致的操作的重复触发。
307	Temporary Redirect	方法和消息主体都不发生变化。	由于不可预见的原因该页面暂不可用。在这种情况下，搜索引擎不会更新它们的链接。当站点支持非 GET 方法的链接或操作的时候，该状态码优于 302 状态码。

BAT交流群：892872246

1.5 特殊重定向

> 除了上述两种常见的重定向之外，还有两种特殊的重定向。304（Not Modified，资源未被修改）会使页面跳转到本地缓存的版本当中（该缓存已过期(?)），而 300（Multiple Choice，多项选择）则是一种手工重定向：以 Web 页面形式呈现在浏览器中的消息主体包含了一个可能的重定向链接的列表，用户可以从中进行选择。

编码	含义	典型应用场景
300	Multiple Choice	不会太多：所有的选项在消息主体的 HTML 页面中列出。也可以返回 200 OK 状态码。
304	Not Modified	缓存刷新：该状态码表示缓存值依然有效，可以使用。

二、缓存处理

2.1 缓存的优势

缓存的使用场景很多，通过它可以将数据通过一定的规则存储起来，再次请求数据的时候就可以快速从缓存中读取了，缓存有以下优势。

> 1. 减少向服务器请求的次数，减轻服务器的负载。
> 2. 加快了本地的响应速度，直接从缓存中取数据比从网络读取要快很多。
> 3. 提供无网模式下的浏览体验，没有网络的情况下也能显示内容。

2.2 HTTP 的缓存机制

HTTP 本身提供了一套缓存相关的机制。这套机制定义了相关的字段和规则，用来客户端和服务端进行缓存相关的协商，如响应的数据是否需要缓存，缓存有效期，缓存是否有效，服务器端给出指示，而客户端则根据服务端的指示做具体的缓存更新和读取缓存工作。http 缓存可分为两类

BAT交流群：892872246

> **强制缓存**
>
> 强制缓存，是直接向缓存数据库请求数据，如果找到了对应的缓存数据，并且是有效的，就直接返回缓存数据。如果没有找到或失效了，则向服务器请求数据，返回数据和缓存规则，同时将数据和缓存规则保存到缓存数据库中。

> **对比缓存**
>
> 对比缓存，是先向缓存数据库获取缓存数据的标识，然后用该标识去服务器请求该标识对应的数据是否失效，如果没有失效，服务器会返回304未失效响应，则客户端使用该标识对应的缓存。如果失效了，服务器会返回最新的数据和缓存规则，客户端使用返回的最新数据，同时将数据和缓存规则保存到缓存数据库中。

强制缓存

强制缓存，在缓存数据未失效的情况下，可以直接使用缓存数据，有两个字段 Expires 和 Cache-Control 用于标明失效规则。

Expires

表示过期时间，由服务端返回。那么下次请求数据时，判断这个 Expires 过期时间是否已经过了，如果还没有到过期时间，则使用缓存，如果过了过期时间，则重新请求服务器的数据。Expires 格式如下：

> Expires: Sat, 11 Nov 2017 10:30:01 GMT
>
> 表示到期时间是2017年11月11日10点30分，在这个时间之前可以使用缓存，过了这个时间就要重新请求服务器数据了。

不过因为服务器和客户端的时间并不是同步的，用一个绝对时间作为过期的标记并不是很明智，所以 HTTP1.1 之后更多的是 Cache-Control，它的控制更加灵活。

Cache-Control

表示缓存的控制，有服务端返回。它有以下几个取值：

BAT交流群：892872246

> **public**
>
> 表示数据内容都可以被储存起来，就连有密码保护的网页也储存，安全性很低
>
> **private**
>
> 表示数据内容只能被储存到私有的cache，仅对某个用户有效，不能共享
>
> **no-cache**
>
> 表示可以缓存，但是只有在跟WEB服务器验证了其有效后，才能返回给客户端，触发对比缓存
>
> **no-store**
>
> 表示请求和响应都禁止被缓存，强制缓存，对比缓存都不会触发
>
> **max-age**
>
> 表示返回数据的过期时间

默认情况下是private，也就是不能共享的。Cache-Control格式如下：

> Cache-Control:public, max-age=31536000
>
> 表示可以被公共缓存，有效时间是1年，也就是说一年时间内，请求该数据时，直接使用缓存，而不用请求服务器了。

对比缓存

对比缓存，表示需要和服务端进行相关信息的对比，由服务器决定是使用缓存还是最新内容，如果服务器判定使用缓存，返回响应吗 304，判定使用最新内容，则返回响应码 200 和最新数据。对比缓存的判定字段有两组：

ETag 和 If-None-Match

ETag 表示资源的一种标识信息，用于标识某个资源，由服务端返回，优先级更高。格式如下：

BAT交流群：892872246

```
Etag:"AFY10-6MddXmSerSiXP1ZTiU65VS"

表示该资源的标识是AFY10-6MddXmSerSiXP1ZTiU65VS
```

然后客户端再次请求时，加入字段If-None-Match，格式如下：

```
If-None-Match:"AFY10-6MddXmSerSiXP1ZTiU65VS"
```

服务端收到请求的该字段时（之前的 Etag 值），和资源的唯一标识进行对比，如果相同，说明没有改动，则返回状态码 304，如果不同，说明资源被改过了，则返回状态码 200 和整个内容数据。

Last-Modified 和 If-Modified-Since

Last-Modified 表示资源的最近修改时间，由服务端返回，优先级更低。格式如下：

```
Last-Modified: Sat, 11 Nov 2017 10:30:01 GMT

表示上次修改时间是2017年11月11日10点30分。

If-Modified-Since: Sat, 11 Nov 2017 10:30:01 GMT

客户端请求，表示我指定的这个2017年11月11日10点30分是不是你服务器最新的修改时间。
```

Last-Modified
由服务器返回，表示响应的数据最近修改的时间。

If-Modified-Since
由客户端请求，表示询问服务器这个时间是不是上次修改的时间。如果服务端该资源的修改时间小于等于 If-Modified-Since 指定的时间，说明资源没有改动，返回响应状态码 304，可以使用缓存。如果服务端该资源的修改时间大于 If-Modified-Since 指定的时间，说明资源又有改动了，则返回响应状态码 200 和最新数据给客户端，客户端使用响应返回的最新数据。

Last-Modified 字段的值（服务端返回的资源上次修改时间），常常被用于客户端下次请求时的 If-Modified-Since 字段中。

BAT交流群：892872246

两种缓存的区别

> 强制缓存的情况下，如果缓存是有效的，则直接使用缓存，而对比缓存不管缓存是否有效，都需要先去和服务器对比是否有新的数据，没有新的数据才使用缓存数据。

两种缓存的使用情景

> 对于强制缓存，服务器通知浏览器一个缓存时间，在缓存时间内，下次请求，直接用缓存，不在时间内，执行对比缓存策略。
>
> 对于对比缓存，将缓存信息中的Etag和Last-Modified通过请求发送给服务器，由服务器校验，返回304状态码时，浏览器直接使用缓存。

HTTP 的缓存规则总结

HTTP 的缓存规则是优先考虑强制缓存，然后考虑对比缓存。

1. 首先判断强制缓存中的数据的是否在有效期内。如果在有效期，则直接使用缓存。如果过了有效期，则进入对比缓存。
2. 在对比缓存过程中，判断 ETag 是否有变动，如果服务端返回没有变动，说明资源未改变，使用缓存。如果有变动，判断 Last-Modified。
3. 判断 Last-Modified，如果服务端对比资源的上次修改时间没有变化，则使用缓存，否则重新请求服务端的数据，并作缓存工作。

BAT交流群：892872246

2.3 Okhttp 缓存相关类

Okhttp 缓存相关的类有如下：

CacheControl（HTTP 中的 Cache-Control 和 Pragma 缓存控制）

CacheControl 是用于描述 HTTP 的 Cache-Control 和 Pragma 字段的类，用于指定缓存的规则。

CacheStrategy（缓存策略类）

CacheStrategy 是用于判定使用缓存数据还是网络请求的决策类。

Cache（缓存类）

对外开放的缓存类，提供了缓存的增删改查接口。

BAT交流群：892872246

InternalCache（内部缓存类）

对内使用的缓存类接口，没有具体实现，只是封装了 Cache 的使用。

DiskLruCache（文件化的 LRU 缓存类）

这是真正实现缓存功能的类，将数据存储在文件中，并使用 LRU 规则（由 LinkedHashMap 实现），控制对缓存文件的增删改查。

2.4 Okhttp 缓存的启用

要开启使用 Okhttp 的缓存其实很简单，只需要给 OkHttpClient 对象设置一个 Cache 对象即可，创建一个 Cache 时指定缓存保存的目录和缓存最大的大小即可。

```
//新建一个 cache，指定目录为外部目录下的 okhttp_cache 目录，大小为 100M

Cache cache = new Cache(new File(Environment.getExternalStorageDirectory
() + "/okhttp_cache/"), 100 * 1024 * 1024);

将 cache 设置到 OkHttpClient 中，这样缓存就开始生效了。

OkHttpClient client = new OkHttpClient.Builder().cache(cache).build();
```

那么下面我们来看看 Okhttp 缓存执行的大概流程

2.5 Okhttp 的缓存流程

Okhttp 的缓存流程分为读取缓存和存储缓存两个过程，我们分别分析。

Okhttp 读取缓存流程

读取使用缓存的流程从 HttpEngine 的 sendRequest 发送请求开始。

1. 首先获取 OkHttpClient 的 Cache 缓存对象，就是之前创建 OkHttpClient 时设置的 Cache。
2. 然后传入 Request 请求到 Cache 的 get 方法去查找缓存响应数据 Response。
3. 构造一个缓存策略，传入 Request 请求和缓存响应 Response，然后调用它的 get 方法去决策使用网络请求还是缓存响应。
4. 策略判定之后，如果是使用缓存，则它的 cacheResponse 不为空，networkRequest 为空，如果使用请求，则相反。然后再将策略给出的这两个值，继续处理。
5. 如果使用请求，但是之前又找到了缓存响应，则要关闭缓存响应资源。
6. 如果策略得出缓存响应为空，网络请求也为空，则返回请求不合理的响应。（比如强制使用缓存，但是找不到缓存的情况下）

BAT交流群：892872246

7. 如果请求为空，缓存不为空，也就是使用缓存的情况，则使用缓存响应来构造返回的响应数据。
8. 最后就是只使用网络请求的情况，走网络请求路线。
   总的来说就是，先查找是否有可用的 Cache，然后通过 Cache 找到请求对应的缓存，然后将请求和缓存交给缓存策略去判断使用请求还是缓存，得出结果后，自己再判断使用缓存还是请求，如果使用缓存，用缓存构造响应直接返回，如果使用请求，那么开始网络请求流程。

```
public final class HttpEngine {



  //发送请求

  public void sendRequest() throws RequestException, RouteException, IOE
xception {

    if (cacheStrategy != null) return; // Already sent.

    if (httpStream != null) throw new IllegalStateException();

    //根据用户请求得到实际的网络请求

    Request request = networkRequest(userRequest);

    //这里 InternalCache 就是对 Cache 的封装，它的实现在 Cache 的 internalCache
中。

    InternalCache responseCache = Internal.instance.internalCache(clien
t);

    //通过 Cache 的 get 方法查找缓存响应

    Response cacheCandidate = responseCache != null

        ? responseCache.get(request)

        : null;



    long now = System.currentTimeMillis();

    //构造缓存策略，然后进行策略判断
```

```
    cacheStrategy = new CacheStrategy.Factory(now, request, cacheCandida
te).get();

    //策略判定后的网络请求和缓存响应

    networkRequest = cacheStrategy.networkRequest;

    cacheResponse = cacheStrategy.cacheResponse;



    if (responseCache != null) {

      //使用缓存响应的话，记录一下使用记录

      responseCache.trackResponse(cacheStrategy);

    }



    if (cacheCandidate != null && cacheResponse == null) {

      //使用网络请求，但是之前又有缓存的话，要关闭缓存，释放资源

      closeQuietly(cacheCandidate.body()); // The cache candidate wasn't
applicable. Close it.

    }



    // If we're forbidden from using the network and the cache is insuffi
cient, fail.

    if (networkRequest == null && cacheResponse == null) {

      //强制使用缓存，又找不到缓存，就报不合理请求响应了

      userResponse = new Response.Builder()

          .request(userRequest)

          .priorResponse(stripBody(priorResponse))
```

```
        .protocol(Protocol.HTTP_1_1)
        .code(504)
        .message("Unsatisfiable Request (only-if-cached)")
        .body(EMPTY_BODY)
        .build();
    return;
  }

  //上面情况处理之后，就是使用缓存返回，还是网络请求的情况了

  // If we don't need the network, we're done.
  if (networkRequest == null) {
    //使用缓存返回响应
    userResponse = cacheResponse.newBuilder()
        .request(userRequest)
        .priorResponse(stripBody(priorResponse))
        .cacheResponse(stripBody(cacheResponse))
        .build();
    userResponse = unzip(userResponse);
    return;
  }

  //使用网络请求
```

```
    //下面就是网络请求流程了，略

    ...

  }  }
```

接下来我们分析

1. Cache 是如何获取缓存的。
2. 缓存策略是如何判断的。

Cache 获取缓存

从 Cache 的 get 方法开始。它按以下步骤进行。

1. 计算 request 对应的 key 值，md5 加密请求 url 得到。
2. 根据 key 值去 DiskLruCache 查找是否存在缓存内容。
3. 存在缓存的话，创建缓存 Entry 实体。ENTRY_METADATA 代表响应头信息，ENTRY_BODY 代表响应体信息。
4. 然后根据缓存 Entry 实体得到响应，其中包含了缓存的响应头和响应体信息。
5. 匹配这个缓存响应和请求的信息是否匹配，不匹配的话要关闭资源，匹配的话返回。

```
public final class Cache implements Closeable, Flushable {

  //获取缓存

  Response get(Request request) {

    //计算请求对应的 key

    String key = urlToKey(request);

    DiskLruCache.Snapshot snapshot;

    Entry entry;

    try {

      //这里从 DiskLruCache 中读取缓存信息

      snapshot = cache.get(key);

      if (snapshot == null) {
```

```
      return null;
    }
  } catch (IOException e) {
    // Give up because the cache cannot be read.
    return null;
  }


  try {
    //这里读取缓存的响应头信息
    entry = new Entry(snapshot.getSource(ENTRY_METADATA));
  } catch (IOException e) {
    Util.closeQuietly(snapshot);
    return null;
  }
  //然后得到响应信息，包含了缓存响应头和响应体信息
  Response response = entry.response(snapshot);
  //判断缓存响应和请求是否匹配，匹配 url,method,和其他响应头信息
  if (!entry.matches(request, response)) {
    //不匹配的话，关闭响应体
    Util.closeQuietly(response.body());
    return null;
  }
```

```
    //返回缓存响应
    return response;
  }

  //这里 md5 加密 url 得到 key 值
  private static String urlToKey(Request request) {
    return Util.md5Hex(request.url().toString());
  }
}
```

如果存在缓存的话，在指定的缓存目录中，会有两个文件“****.0”和“****.1”，分别存储某个请求缓存的响应头和响应体信息。（“****”是 url 的 md5 加密值）对应的 ENTRY_METADATA 响应头和 ENTRY_BODY 响应体。缓存的读取其实是由 DiskLruCache 来读取的，DiskLruCache 是支持 Lru（最近最少访问）规则的用于磁盘存储的类，对应 LruCache 内存存储。它在存储的内容超过指定值之后，就会根据最近最少访问的规则，把最近最少访问的数据移除，以达到总大小不超过限制的目的。

接下来我们分析 CacheStrategy 缓存策略是怎么判定的。

CacheStrategy 缓存策略

直接看 CacheStrategy 的 get 方法。缓存策略是由请求和缓存响应共同决定的。

1. 如果缓存响应为空，则缓存策略为不使用缓存。
2. 如果请求是 https 但是缓存响应没有握手信息，同上不使用缓存。
3. 如果请求和缓存响应都是不可缓存的，同上不使用缓存。
4. 如果请求是 noCache，并且又包含 If-Modified-Since 或 If-None-Match，同上不使用缓存。
5. 然后计算请求有效时间是否符合响应的过期时间，如果响应在有效范围内，则缓存策略使用缓存。
6. 否则创建一个新的有条件的请求，返回有条件的缓存策略。
7. 如果判定的缓存策略的网络请求不为空，但是只使用缓存，则返回两者都为空的缓存策略。

BAT交流群：892872246

```
public final class CacheStrategy {



    public Factory(long nowMillis, Request request, Response cacheRespon
se) {

      this.nowMillis = nowMillis;

      //网络请求和缓存响应

      this.request = request;

      this.cacheResponse = cacheResponse;



      if (cacheResponse != null) {

        //找到缓存响应的响应头信息

        Headers headers = cacheResponse.headers();

        for (int i = 0, size = headers.size(); i < size; i++) {

          //查看响应头信息中是否有以下字段信息

          String fieldName = headers.name(i);

          String value = headers.value(i);

          if ("Date".equalsIgnoreCase(fieldName)) {

            servedDate = HttpDate.parse(value);

            servedDateString = value;

          } else if ("Expires".equalsIgnoreCase(fieldName)) {

            expires = HttpDate.parse(value);

          } else if ("Last-Modified".equalsIgnoreCase(fieldName)) {

            lastModified = HttpDate.parse(value);
```

```
          lastModifiedString = value;
        } else if ("ETag".equalsIgnoreCase(fieldName)) {
          etag = value;
        } else if ("Age".equalsIgnoreCase(fieldName)) {
          ageSeconds = HeaderParser.parseSeconds(value, -1);
        } else if (OkHeaders.SENT_MILLIS.equalsIgnoreCase(fieldName)) {
          sentRequestMillis = Long.parseLong(value);
        } else if (OkHeaders.RECEIVED_MILLIS.equalsIgnoreCase(fieldNam
e)) {
          receivedResponseMillis = Long.parseLong(value);
        }
      }
    }
  }

  public CacheStrategy get() {
    //获取判定的缓存策略
    CacheStrategy candidate = getCandidate();

    if (candidate.networkRequest != null && request.cacheControl().onl
yIfCached()) {
      // 如果判定的缓存策略的网络请求不为空，但是只使用缓存，则返回两者都为空
的缓存策略。
      return new CacheStrategy(null, null);
```

```
    }


    return candidate;

  }


  /** Returns a strategy to use assuming the request can use the networ
k. */

  private CacheStrategy getCandidate() {

    // No cached response.

    //如果没有缓存响应，则返回没有缓存响应的策略

    if (cacheResponse == null) {

      return new CacheStrategy(request, null);

    }


    // Drop the cached response if it's missing a required handshake.

    //如果请求是 https，而缓存响应的握手信息为空，则返回没有缓存响应的策略

    if (request.isHttps() && cacheResponse.handshake() == null) {

      return new CacheStrategy(request, null);

    }


    // If this response shouldn't have been stored, it should never be
used

    // as a response source. This check should be redundant as long as
the
```

```
    // persistence store is well-behaved and the rules are constant.

    //如果请求对应的响应不能被缓存，则返回没有缓存响应的策略

    if (!isCacheable(cacheResponse, request)) {

      return new CacheStrategy(request, null);

    }



    //获取请求头中的 CacheControl 信息

    CacheControl requestCaching = request.cacheControl();

    //如果请求头中的 CacheControl 信息是不缓存的，则返回没有缓存响应的策略

    if (requestCaching.noCache() || hasConditions(request)) {

      return new CacheStrategy(request, null);

    }



    //获取响应的年龄

    long ageMillis = cacheResponseAge();

    //计算上次响应刷新的时间

    long freshMillis = computeFreshnessLifetime();

    //如果请求里有最大持续时间要求，则取较小的值作为上次响应的刷新时间

    if (requestCaching.maxAgeSeconds() != -1) {

      freshMillis = Math.min(freshMillis, SECONDS.toMillis(requestCach
ing.maxAgeSeconds()));

    }
```

```
      //如果请求里有最短刷新时间要求，则用它来作为最短刷新时间

      long minFreshMillis = 0;

      if (requestCaching.minFreshSeconds() != -1) {

        minFreshMillis = SECONDS.toMillis(requestCaching.minFreshSeconds
());

      }



      //最大过期时间

      long maxStaleMillis = 0;

      //获取缓存响应头中的 CacheControl 信息

      CacheControl responseCaching = cacheResponse.cacheControl();

      //如果缓存响应不是必须要再验证，并且请求有最大过期时间，则用请求的最大过
期时间作为最大过期时间

      if (!responseCaching.mustRevalidate() && requestCaching.maxStaleSe
conds() != -1) {

        maxStaleMillis = SECONDS.toMillis(requestCaching.maxStaleSeconds
());

      }



      //如果支持缓存，并且持续时间+最短刷新时间<上次刷新时间+最大验证时间 则可
以缓存

      if (!responseCaching.noCache() && ageMillis + minFreshMillis < fres
hMillis + maxStaleMillis) {

        Response.Builder builder = cacheResponse.newBuilder();

        if (ageMillis + minFreshMillis >= freshMillis) {
```

BAT文 892246

```
        builder.addHeader("Warning", "110 HttpURLConnection \"Response
is stale\"");

      }

      long oneDayMillis = 24 * 60 * 60 * 1000L;

      if (ageMillis > oneDayMillis && isFreshnessLifetimeHeuristic()) {

        builder.addHeader("Warning", "113 HttpURLConnection \"Heuristic
 expiration\"");

      }

      //返回响应缓存

      return new CacheStrategy(null, builder.build());

    }



    //构造一个新的有条件的 Request，添加 If-None-Match，If-Modified-Since
等信息

    Request.Builder conditionalRequestBuilder = request.newBuilder();



    if (etag != null) {

      conditionalRequestBuilder.header("If-None-Match", etag);

    } else if (lastModified != null) {

      conditionalRequestBuilder.header("If-Modified-Since", lastModifi
edString);

    } else if (servedDate != null) {

      conditionalRequestBuilder.header("If-Modified-Since", servedDate
String);

    }
```

```
    Request conditionalRequest = conditionalRequestBuilder.build();

    //根据是否有 If-None-Match，If-Modified-Since 信息，返回不同的缓存策略

    return hasConditions(conditionalRequest)

        ? new CacheStrategy(conditionalRequest, cacheResponse)

        : new CacheStrategy(conditionalRequest, null);

   }



   /**

    * Returns true if the request contains conditions that save the serv
er from sending a response

    * that the client has locally. When a request is enqueued with its o
wn conditions, the built-in

    * response cache won't be used.

    */

   private static boolean hasConditions(Request request) {

     return request.header("If-Modified-Since") != null || request.head
er("If-None-Match") != null;

   }}
```

接来下我们看看 CacheControl 类里有些什么。

CacheControl

```
public final class CacheControl {



  //表示这是一个优先使用网络验证，验证通过之后才可以使用缓存的缓存控制，设置了 n
oCache
```

```
  public static final CacheControl FORCE_NETWORK = new Builder().noCache
().build();



  //表示这是一个优先先使用缓存的缓存控制，设置了 onlyIfCached 和 maxStale 的最
大值

  public static final CacheControl FORCE_CACHE = new Builder()

      .onlyIfCached()

      .maxStale(Integer.MAX_VALUE, TimeUnit.SECONDS)

      .build();



  //以下的字段都是 HTTP 中 Cache-Control 字段相关的值

  private final boolean noCache;

  private final boolean noStore;

  private final int maxAgeSeconds;

  private final int sMaxAgeSeconds;

  private final boolean isPrivate;

  private final boolean isPublic;

  private final boolean mustRevalidate;

  private final int maxStaleSeconds;

  private final int minFreshSeconds;

  private final boolean onlyIfCached;

  private final boolean noTransform;



  //解析头文件中的相关字段，得到该缓存控制类
```

```
  public static CacheControl parse(Headers headers) {

    ...

  }

}
```

可以发现，它就是用于描述响应的缓存控制信息。

然后我们再看看 Okhttp 存储缓存是怎么进行的。

Okhttp 存储缓存流程

存储缓存的流程从 HttpEngine 的 readResponse 发送请求开始的。

```
public final class HttpEngine {

  /**

   * Flushes the remaining request header and body, parses the HTTP respo
nse headers and starts

   * reading the HTTP response body if it exists.

   */

  public void readResponse() throws IOException {

    //读取响应，略

    ...



    // 判断响应信息中包含响应体

    if (hasBody(userResponse)) {

      // 如果缓存的话，缓存响应头信息

      maybeCache();

      //缓存响应体信息，同时 zip 解压缩响应数据
```

BA 2246

```
      userResponse = unzip(cacheWritingResponse(storeRequest, userRespon
se));

    }

  }



  // 如果缓存的话，缓存响应头信息

  private void maybeCache() throws IOException {

    InternalCache responseCache = Internal.instance.internalCache(clien
t);

    if (responseCache == null) return;



    // Should we cache this response for this request?

    if (!CacheStrategy.isCacheable(userResponse, networkRequest)) {

      if (HttpMethod.invalidatesCache(networkRequest.method())) {

        try {

          responseCache.remove(networkRequest);

        } catch (IOException ignored) {

          // The cache cannot be written.

        }

      }

      return;

    }



    // Offer this request to the cache.
```

```
    //这里将响应头信息缓存到缓存文件中，对应缓存文件“\*\*\*\*.0”

    storeRequest = responseCache.put(stripBody(userResponse));

  }



  /**

   * Returns a new source that writes bytes to {@code cacheRequest} as th
ey are read by the source

   * consumer. This is careful to discard bytes left over when the stream
 is closed; otherwise we

   * may never exhaust the source stream and therefore not complete the c
ached response.

   */

  //缓存响应体信息

  private Response cacheWritingResponse(final CacheRequest cacheRequest,
 Response response)

      throws IOException {

    // Some apps return a null body; for compatibility we treat that like
 a null cache request.

    if (cacheRequest == null) return response;

    Sink cacheBodyUnbuffered = cacheRequest.body();

    if (cacheBodyUnbuffered == null) return response;



    final BufferedSource source = response.body().source();

    final BufferedSink cacheBody = Okio.buffer(cacheBodyUnbuffered);
```

```
    Source cacheWritingSource = new Source() {

      boolean cacheRequestClosed;



      //这里就是从响应体体读取数据，保存到缓存文件中，对应缓存文件“\*\*\*\*.1”

      @Override public long read(Buffer sink, long byteCount) throws IOEx
ception {

        long bytesRead;

        try {

          bytesRead = source.read(sink, byteCount);

        } catch (IOException e) {

          if (!cacheRequestClosed) {

            cacheRequestClosed = true;

            cacheRequest.abort(); // Failed to write a complete cache res
ponse.

          }

          throw e;

        }



        if (bytesRead == -1) {

          if (!cacheRequestClosed) {

            cacheRequestClosed = true;

            cacheBody.close(); // The cache response is complete!

          }

          return -1;
```

```
      }


      sink.copyTo(cacheBody.buffer(), sink.size() - bytesRead, bytesRea
d);

      cacheBody.emitCompleteSegments();

      return bytesRead;

    }


    @Override public Timeout timeout() {

      return source.timeout();

    }


    @Override public void close() throws IOException {

      if (!cacheRequestClosed

          && !discard(this, HttpStream.DISCARD_STREAM_TIMEOUT_MILLIS, M
ILLISECONDS)) {

        cacheRequestClosed = true;

        cacheRequest.abort();

      }

      source.close();

    }

  };


  return response.newBuilder()
```

```
        .body(new RealResponseBody(response.headers(), Okio.buffer(cache
WritingSource)))

        .build();

  }}
```

可以看到这里先通过 maybeCache 写入了响应头信息，再通过 cacheWritingResponse 写入了响应体信息。我们再进去看 Cache 的 put 方法实现。

```
private CacheRequest put(Response response) throws IOException {

    String requestMethod = response.request().method();



    // 响应的请求方法不支持缓存，只有 GET 方法支持缓存

    if (HttpMethod.invalidatesCache(response.request().method())) {

      try {

        remove(response.request());

      } catch (IOException ignored) {

        // The cache cannot be written.

      }

      return null;

    }

    // 同样，请求只支持 GET 方法的缓存

    if (!requestMethod.equals("GET")) {

      // Don't cache non-GET responses. We're technically allowed to cach
e
```

BAT交 892246

```
      // HEAD requests and some POST requests, but the complexity of doin
g

      // so is high and the benefit is low.

      return null;

    }



    //缓存不支持通配符

    if (OkHeaders.hasVaryAll(response)) {

      return null;

    }



    //开始缓存

    Entry entry = new Entry(response);

    DiskLruCache.Editor editor = null;

    try {

      editor = cache.edit(urlToKey(response.request()));

      if (editor == null) {

        return null;

      }

      entry.writeTo(editor);

      return new CacheRequestImpl(editor);

    } catch (IOException e) {

      abortQuietly(editor);
```

```
    return null;

  }}
```

我们继续看 Cache 的 writeTo 方法，可以看到是写入一些响应头信息。

```
public void writeTo(DiskLruCache.Editor editor) throws IOException {

    BufferedSink sink = Okio.buffer(editor.newSink(ENTRY_METADATA));



    sink.writeUtf8(url);

    sink.writeByte('\n');

    sink.writeUtf8(requestMethod);

    sink.writeByte('\n');

    sink.writeDecimalLong(varyHeaders.size());

    sink.writeByte('\n');

    for (int i = 0, size = varyHeaders.size(); i < size; i++) {

      sink.writeUtf8(varyHeaders.name(i));

      sink.writeUtf8(": ");

      sink.writeUtf8(varyHeaders.value(i));

      sink.writeByte('\n');

    }



    sink.writeUtf8(new StatusLine(protocol, code, message).toString
());

    sink.writeByte('\n');

    sink.writeDecimalLong(responseHeaders.size());
```

```
      sink.writeByte('\n');

      for (int i = 0, size = responseHeaders.size(); i < size; i++) {

        sink.writeUtf8(responseHeaders.name(i));

        sink.writeUtf8(": ");

        sink.writeUtf8(responseHeaders.value(i));

        sink.writeByte('\n');

      }



      if (isHttps()) {

        sink.writeByte('\n');

        sink.writeUtf8(handshake.cipherSuite().javaName());

        sink.writeByte('\n');

        writeCertList(sink, handshake.peerCertificates());

        writeCertList(sink, handshake.localCertificates());

        // The handshake’s TLS version is null on HttpsURLConnection and
on older cached responses.

        if (handshake.tlsVersion() != null) {

          sink.writeUtf8(handshake.tlsVersion().javaName());

          sink.writeByte('\n');

        }

      }

      sink.close();

    }
```

到这里 Okhttp 缓存的读取和存储流程我们就清楚了。可以说，缓存的使用策略基本都是按照 HTTP 的缓存定义来实现的，所以对 HTTP 缓存相关字段的理解是很重要的。然后关于 DiskLruCache 是如何管理缓存文件的，这个其实也很好理解，首先的原则就是按照 LRU 这种最近最少使用删除的原则，当总的大小超过限定大小后，删除最近最少使用的缓存文件，它的 LRU 算法是使用 LinkedHashMap 进行维护的，这样来保证，保留的缓存文件都是更常使用的。具体实现大家可以分析 DiskLruCache 和 LinkedHashMap 的实现原理。
参考 https://www.jianshu.com/p/00d281c226f6
https://www.jianshu.com/p/3f9128380cf7
阿里 P7 移动互联网架构师进阶视频（每日更新中）免费学习请点击：
https://space.bilibili.com/474380680

5.2.3 高并发请求队列：任务分发

1 概述

1.1 引言

android 完成非阻塞式的异步请求的时候都是通过启动子线程的方式来解决，子线程执行完任务的之后通过 handler 的方式来和主线程来完成通信。无限制的创建线程，会给系统带来大量的开销。如果在高并发的任务下，启用个线程池，可以不断的复用里面不再使用和有效的管理线程的调度和数量的管理。就可以节省系统的成本，有效的提高执行效率。

1.2 线程池 ThreadPoolExecutor

okhttp 的线程池对象存在于 Dispatcher 类中。实例过程如

```
1. public synchronized ExecutorService executorService() {2. if (executorService == null) {3. executorService = , Integer.MAX_VALUE, , TimeUnit.SECONDS,4. new SynchronousQueue<Runnable>(), Util.threadFactory("OkHttp Dispatcher", false));5. }6. return executorService;7. }
```

1.2 Call 对象

了解源码或使用过 okhttp 的都知道。 okttp 的操作元是 Call 对象。异步的实现是 RealCall.AsyncCall。而 AsyncCall 是实现的一个 Runnable 接口。

```
1. final class AsyncCall extends NamedRunnable {}
```

所以 Call 本质就是一个 Runable 线程操作元肯定是放进 excutorService 中直接启动的。

BAT交流群 892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

2 线程池的复用和管理

2.1 图解

为了完成调度和复用，定义了两个队列分别用作等待队列和执行任务的队列。这两个队列都是 `Dispatcher` 成员变量。Dispatcher 是一个控制执行，控制所有 Call 的分发和任务的调度、通信、清理等操作。这里只介绍异步调度任务。

```
1.  /** Ready async calls in the order they'll be run. */2.  private  final  Deque<AsyncCall> readyAsyncCalls =  new  ArrayDeque<>();

4.  /** Running asynchronous calls. Includes canceled calls that haven't finished yet. */5.  private  final  Deque<AsyncCall> runningAsyncCalls =  new  ArrayDeque<>();
```

在《okhttp 连接池复用机制》文章中我们在缓存 Connection 连接的时候也是使用的 Deque 双端队列。这里同样的方式，可以方便在队列头添加元素，移除尾部的元素。

2.2 过程分析

`Call` 代用 `equeue` 方法的时候

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
1. synchronized void enqueue(AsyncCall call) {2. if (runningAsyncCal
ls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerH
ost) {3. runningAsyncCalls.add(call);4. executorService().execute(cal
l);5. } else {6. readyAsyncCalls.add(call);7. }8. }
```

方法中满足执行队列里面不足最大线程数 maxRequests 并且 Call 对应的 host 数目不超过 maxRequestsPerHost 的时候直接把 call 对象直接推入到执行队列里，并启动线程任务（Call 本质是一个 Runnable）。否则，当前线程数过多，就把他推入到等待队列中。Call 执行完肯定需要在 runningAsyncCalls 队列中移除这个线程。那么 readyAsyncCalls 队列中的线程在什么时候才会被执行呢。

追溯下 AsyncCall 线程的执行方法

方法中满足执行队列里面不足最大线程数 maxRequests 并且 Call 对应的 host 数目不超过 maxRequestsPerHost 的时候直接把 call 对象直接推入到执行队列里，并启动线程任务（Call 本质是一个 Runnable）。否则，当前线程数过多，就把他推入到等待队列中。Call 执行完肯定需要在 runningAsyncCalls 队列中移除这个线程。那么 readyAsyncCalls 队列中的线程在什么时候才会被执行呢。

追溯下 AsyncCall 线程的执行方法

作者：Alvin 老师
链接：https://www.jianshu.com/p/3c427e2c0586
来源：简书
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

方法中满足执行队列里面不足最大线程数 maxRequests 并且 Call 对应的 host 数目不超过 maxRequestsPerHost 的时候直接把 call 对象直接推入到执行队列里，并启动线程任务（Call 本质是一个 Runnable）。否则，当前线程数过多，就把他推入到等待队列中。Call 执行完肯定需要在 runningAsyncCalls 队列中移除这个线程。那么 readyAsyncCalls 队列中的线程在什么时候才会被执行呢。

追溯下 AsyncCall 线程的执行方法

```
1. @Override2. protected void execute() {3. boolean signalledCallbac
k = false;4. try {5. Response response = getResponseWithInterceptorCh
ain(forWebSocket);6. if (canceled) {7. signalledCallback = true;8.
responseCallback.onFailure(RealCall.this, new IOException("Canceled"));
9. } else {10. signalledCallback = true;11. responseCallback.onResp
onse(RealCall.this, response);12. }13. } catch (IOException e) {14.
 if (signalledCallback) {15. // Do not signal the callback twice!16.
Platform.get().log(INFO, "Callback failure for " + toLoggableString(),
 e);17. } else {18. responseCallback.onFailure(RealCall.this, e);19.
```

```
}20. } finally {21. client.dispatcher().finished(this);22. }23. }
24. }
```

这里做了核心 request 的动作，并把失败和回复数据的结果通过 responseCallback 回调到 Dispatcher。执行操作完毕了之后不管有无异常都会进入到 dispactcher 的 finished 方法。

```
1. private <T> void finished(Deque<T> calls, T call, boolean promoteC
alls) {2. int runningCallsCount;3. Runnable idleCallback;4. synchron
ized (this) {5. if (!calls.remove(call)) throw new AssertionError
("Call wasn't in-flight!");6. if (promoteCalls) promoteCalls();7. run
ningCallsCount = runningCallsCount();8. idleCallback = this.idleCallba
ck;9. }

11. && idleCallback != null) {12. idleCallback.run();13. }14. }
```

在这里 call 在 runningAsyncCalls 队列中被移除了，重新计算了目前正在执行的线程数量。并且调用了 `promoteCalls()` 看来是来调整任务队列的，跟进去看下

```
1. private void promoteCalls() {2. if (runningAsyncCalls.size() >=
maxRequests) return; // Already running max capacity.3. if (readyAsyn
cCalls.isEmpty()) return; // No ready calls to promote.

5. for (Iterator<AsyncCall> i = readyAsyncCalls.iterator(); i.hasNext
(); ) {6. AsyncCall call = i.next();

8. if (runningCallsForHost(call) < maxRequestsPerHost) {9. i.remove
();10. runningAsyncCalls.add(call);11. executorService().execute(call);
12. }

14. if (runningAsyncCalls.size() >= maxRequests) return; // Reached
max capacity.15. }16. }
```

原来实在这里对 `readyAsyncCalls` 进行调度的。最终会在 `readyAsyncCalls` 中通过 `remove` 操作把元素迭代取出并移除之后加入到 runningAsyncCalls 的执行队列中执行操作。`ArrayDeque` 是非线程安全的所以 `finished` 在调用 `promoteCalls` 的时候都在 `synchronized` 块中执行的。执行等待队列线程当然的前提是 `runningAsyncCalls` 线程数没有超上线，而且等待队列里面有等待的任务。

以上完成了线程线程池的复用和线程的管理工作。

> 小结，Call 在执行任务通过 Dispatcher 把单元任务优先推到执行队列里进行操作，如果操作完成再执行等待队列的任务。

BAT交流群：892872246

作者：Alvin 老师
链接：https://www.jianshu.com/p/3c427e2c0586
来源：简书
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

原来实在这里对 readyAsyncCalls 进行调度的。最终会在 readyAsyncCalls 中通过 remove 操作把元素迭代取出并移除之后加入到 runningAsyncCalls 的执行队列中执行操作。ArrayDeque 是非线程安全的所以 finished 在调用 promoteCalls 的时候都在 synchronized 块中执行的。执行等待队列线程当然的前提是 runningAsyncCalls 线程数没有超上线，而且等待队列里面有等待的任务。

以上完成了线程线程池的复用和线程的管理工作。

> 小结，Call 在执行任务通过 Dispatcher 把单元任务优先推到执行队列里进行操作，如果操作完成再执行等待队列的任务。

5.2.4 责任链模式拦截器设计

阿里 P7 移动互联网架构师进阶视频（每日更新中）免费学习请点击：

https://space.bilibili.com/474380680

1.构造 Demo

首先构造一个简单的异步网络访问 Demo:

```
1. OkHttpClient client = new OkHttpClient();2. Request request = new
 Request.Builder()3. .url("http://publicobject.com/helloworld.txt")4.
 .build();

6. client.newCall(request).enqueue(new Callback() {7. @Override8. p
ublic void onFailure(Call call, IOException e) {9. Log.d("OkHttp", "
Call Failed:" + e.getMessage());10. }

12. @Override13. public void onResponse(Call call, Response response)
 throws IOException {14. Log.d("OkHttp", "Call succeeded:" + respon
se.message());15. }16. });
```

2. 发起请求

`OkHttpClient.newCall` 实际是创建一个 `RealCall` 实例：

```
1. @Override2. public Call newCall(Request request) {3. return new
 RealCall(this, request, false /* for web socket */);4. }
```

`RealCall.enqueue` 实际就是讲一个 `RealCall` 放入到任务队列中，等待合适的机会执行:

```
1. @Override2. public void enqueue(Callback responseCallback) {3. sy
nchronized (this) {4. if (executed) throw new IllegalStateExceptio
n("Already Executed");5. executed = true;6. }7. captureCallStackTrac
e();8. client.dispatcher().enqueue(new AsyncCall(responseCallback));9.
 }
```

从代码中可以看到最终 RealCall 被转化成一个 AsyncCall 并被放入到任务队列中，任务队列中的分发逻辑这里先不说，相关实现会放在 [OkHttp 源码分析——任务队列]疑问进行介绍。这里只需要知道 AsyncCall 的 excute 方法最终将会被执行:

```
1. [RealCall.java]2. @Override protected void execute() {3. boolean
 signalledCallback = false;4. try {5. Response response = getResponse
WithInterceptorChain();6. if (retryAndFollowUpInterceptor.isCanceled())
 {7. signalledCallback = true;8. responseCallback.onFailure(RealCall.
this, new IOException("Canceled"));9. } else {10. signalledCallback
 = true;11. responseCallback.onResponse(RealCall.this, response);12. }
13. } catch (IOException e) {14. if (signalledCallback) {15. // D
o not signal the callback twice!16. Platform.get().log(INFO, "Callback
 failure for " + toLoggableString(), e);17. } else {18. responseCall
back.onFailure(RealCall.this, e);19. }20. } finally {21. client.dis
patcher().finished(this);22. }23. }24. }
```

execute 方法的逻辑并不复杂，简单的说就是：

调用 getResponseWithInterceptorChain 获取服务器返回
通知任务分发器(client.dispatcher)该任务已结束
getResponseWithInterceptorChain 构建了一个拦截器链，通过依次执行该拦截器链中的每一个拦截器最终得到服务器返回。

3. 构建拦截器链

首先来看下 getResponseWithInterceptorChain 的实现：

源码路径：okhttp3/RealCall.java

```
// 开始执行整个请求

Response getResponseWithInterceptorChain() throws IOException {
```

群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
// Build a full stack of interceptors.

// 拦截器栈

List<Interceptor> interceptors = new ArrayList<>();

// 前文说过的 普通拦截器

interceptors.addAll(client.interceptors());

// 重试拦截器，网络错误、请求失败等

interceptors.add(retryAndFollowUpInterceptor);

// 桥接拦截器，主要是重构请求头即 header

interceptors.add(new BridgeInterceptor(client.cookieJar()));

// 缓存拦截器

interceptors.add(newCacheInterceptor(client.internalCache()));

// 连接拦截器，连接服务器，https 包装

interceptors.add(new ConnectInterceptor(client));

// 网络拦截器，websockt 不支持，同样是自定义

if (!forWebSocket) {

  interceptors.addAll(client.networkInterceptors());

}

// 服务拦截器，主要是发送（write、input）、读取（read、output）数据

interceptors.add(new CallServerInterceptor(forWebSocket));



// 开启调用链

Interceptor.Chain chain = new RealInterceptorChain(

    interceptors, , originalRequest);
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
    return chain.proceed(originalRequest);

  }
}
```

其逻辑大致分为两部分：

创建一系列拦截器，并将其放入一个拦截器数组中。这部分拦截器即包括用户自定义的拦截器也包括框架内部拦截器
创建一个拦截器链 RealInterceptorChain,并执行拦截器链的 proceed 方法
接下来看下 RealInterceptorChain 的实现逻辑：

```
public final class RealInterceptorChain implements Interceptor.Chain {

  private final List<Interceptor> interceptors;

  private final StreamAllocation streamAllocation;

  private final HttpCodec httpCodec;

  private final RealConnection connection;

  private final int index;

  private final Request request;

  private int calls;



  public RealInterceptorChain(List<Interceptor> interceptors, StreamAll
ocation streamAllocation,

                              HttpCodec httpCodec, RealConnection connecti
on, int index, Request request) {

    this.interceptors = interceptors;

    this.connection = connection;

    this.streamAllocation = streamAllocation;

    this.httpCodec = httpCodec;
```

```
    this.index = index;

    this.request = request;

  }


  @Override public Connection connection() {

    return connection;

  }


  public StreamAllocation streamAllocation() {

    return streamAllocation;

  }


  public HttpCodec httpStream() {

    return httpCodec;

  }


  @Override public Request request() {

    return request;

  }


  @Override public Response proceed(Request request) throws IOException
{

    return proceed(request, streamAllocation, httpCodec, connection);
```

BA 892246

```
  }



  public Response proceed(Request request, StreamAllocation streamAlloc
ation, HttpCodec httpCodec,

      RealConnection connection) throws IOException {



    ......

    // Call the next interceptor in the chain.

    RealInterceptorChain next = new RealInterceptorChain(

        interceptors, streamAllocation, httpCodec, connection, index + ,
 request);

    Interceptor interceptor = interceptors.get(index);

    Response response = interceptor.intercept(next);



    ......



    return response;

  }

}
```

在 proceed 方法中的核心代码可以看到，proceed 实际上也做了两件事：

创建下一个拦截链。传入 index + 1 使得下一个拦截器链只能从下一个拦截器开始访问
执行索引为 index 的 intercept 方法，并将下一个拦截器链传入该方法。
原文链接：https://www.bbsmax.com/A/MAzAEmQMz9/
阿里 P7 移动互联网架构师进阶视频（每日更新中）免费学习请点击：
https://space.bilibili.com/474380680

BAT交流群：892872246

5.3 Retrofit 源码解析

阿里 P7 移动互联网架构师进阶视频（每日更新中）免费学习请点击：
https://space.bilibili.com/474380680
本文将最后通过 Retrofit2 源码解析来阐述网络访问框架设计。
<meta charset="utf-8">

Retrofit2 源码解析

注意： 本文是对源码的一个跟踪，会对每一行代码有具体的阐述，但是不会介绍 Retrofit 的设计模式。

Retrofit：一个 Restful 设计风格的 HTTP 网络请求框架的封装。基于 OkHttp

A type-safe HTTP client for Android and Java

0. 基本使用

1、Retrofit 将我们的 HTTP API 转换成一个 接口形式。所以我们第一步定义一个 interface

```
public interface GitHubService {

    @GET("user/{user}/repos")

    Call<List<Integer>> listRepos(@Path("user") String user);}
```

2、然后构建一个 Retrofit，通过 create 方法生成 GitHubService 的一个实现

```
Retrofit retrofit = new Retrofit.Builder()

    .baseUrl("https://api.github.com/")

    .build();

GitHubService service = retrofit.create(GitHubService.class);
```

3、调用 listRepos 拿到 Call 实例，可以做同步或异步请求。

每个 Call 实例只能使用一次，但调用 clone() 将创建一个可以使用的新实例。

BAT交流群：892872246

1. Retrofit 构建

1.1 Retrofit

首先看看 Retrofit 吧，这个类里面有 7 个实例变量。我们根据类型和变量名先猜猜是干什么用的，留个大体印象即可。

```
  // 一个线程安全的、支持高效并发的 HashMap，Key 是 Method，Value 是 ServiceMethod。Method 我们能猜到应该就是上面接口中定义的 listRepos，而这个方法中有很多注解，@GET、@Path 啥的，那这个 ServiceMethod 很有可能是这个方法的封装。而变量名带个 Cache 说明，会把这个 Method 对应的 ServiceMethod 缓存起来。

  private final Map<Method, ServiceMethod<?, ?>> serviceMethodCache = new ConcurrentHashMap<>();



  // 想必你知道 Retrofit 就是基于 OkHttp 的封装，那这个 Call.Factory，明显就是 Call 的工厂类。至于 Call 是干嘛的，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；

  final okhttp3.Call.Factory callFactory;

  // 这个很好理解了，就是上面 基本使用 中的 baseUrl，可是这是个 HttpUrl 类型的，我们传的可是 String 类型的呀，那估计是通过 Builder 做了处理的。

  final HttpUrl baseUrl;

  // Converter 根据字面意思可得 这应该是个转换器，用于把我们的 响应 转换成特定的格式

  final List<Converter.Factory> converterFactories;

  // CallAdapter 根据字面意思，难道是对 Call 的一个适配？

  final List<CallAdapter.Factory> callAdapterFactories;

  // Executor 很熟悉了，这是个回调 Executor，想必就是用来切换线程的了

  final @Nullable Executor callbackExecutor;

  // 这个就猜不出了，只能暂时理解为一个标志位
```

```
  final boolean validateEagerly;
```

再来看看 Retrofit 的构造函数

```
  Retrofit(okhttp3.Call.Factory callFactory, HttpUrl baseUrl,

      List<Converter.Factory> converterFactories, List<CallAdapter.Facto
ry> callAdapterFactories,

      @Nullable Executor callbackExecutor, boolean validateEagerly) {

    this.callFactory = callFactory;

    this.baseUrl = baseUrl;

    this.converterFactories = converterFactories; // Copy+unmodifiable a
t call site.

    this.callAdapterFactories = callAdapterFactories; // Copy+unmodifiab
le at call site.

    this.callbackExecutor = callbackExecutor;

    this.validateEagerly = validateEagerly;

  }
```

并没做什么特殊的处理，就是简单的赋值，那想必所有初始化的操作都在 Builder 里了。

那么成功建立一个 Retrofit 对象的标准就是：**配置好 Retrofit 里的成员变量。**

- callFactory：网络请求 工厂
- baseUrl ：网络请求的基本 Url 地址
- converterFactories ：数据转换器 工厂集合
- callAdapterFactories ：网络请求适配器 工厂集合
- callbackExecutor ：回调方法执行器

1.2 Retrofit.Builder

```
public static final class Builder {

    private final Platform platform;
```

群：892872246

```
    private @Nullable okhttp3.Call.Factory callFactory;

    private HttpUrl baseUrl;

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

    private final List<CallAdapter.Factory> callAdapterFactories = new A
rrayList<>();

    private @Nullable Executor callbackExecutor;

    private boolean validateEagerly;



    Builder(Platform platform) {

      this.platform = platform;

    }



    public Builder() {

      this(Platform.get());

    }



    // ... ...

    }
```

我们可以看到 Builder 与 Retrofit 的参数几乎一样，只是少了 serviceMethodCache，多了个 Platform。这个 Platform 很重要。我们通过 Builder 的构造函数可以知道，调用了 Platform.get()方法，然后赋值给自己的 platform 变量。 我们看看这个 Platform 类。

```
class Platform {

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

BA... 群：892872246

```
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 ignored) {
  }
  try {
    Class.forName("java.util.Optional");
    return new Java8();
  } catch (ClassNotFoundException ignored) {
  }
  return new Platform();
}

// ... ...
}
```

get 方法会去调用 findPlatform 方法，这个里面很明显跟平台相关，Class.forName 要求 JVM 根据 className 查找并加载指定的类，如果未找到则抛出 ClassNotFoundException 。这里很明显我们分析 Android 平台，所以会 return 一个 Android（）对象。

```
//Platform 内部 static class Android extends Platform {

    @Override public Executor defaultCallbackExecutor() {

      return new MainThreadExecutor();

    }



    @Override CallAdapter.Factory defaultCallAdapterFactory(@Nullable Ex
ecutor callbackExecutor) {

      if (callbackExecutor == null) throw new AssertionError();

      return new ExecutorCallAdapterFactory(callbackExecutor);

    }



    static class MainThreadExecutor implements Executor {

      private final Handler handler = new Handler(Looper.getMainLooper
());



      @Override public void execute(Runnable r) {

        handler.post(r);

      }

    }

  }
```

我们在这里面可以看到两个重要的方法

BAT交流 892246

- **defaultCallbackExecutor** ：这个方法返回的是个 Executor ，我们想到 Retrofit 正好有个 Executor 类型的变量，那么想必就是它了，它是 MainThreadExecutor 类型的，内部采用 handler 执行任务。
- **defaultCallAdapterFactory** ：这个方法返回的是个 CallAdapter.Factory，Retrofit 成员变量中也正好有个 CallAdapter.Factory 类型的变量，所以说这个 Platform 很重要嘛，跟我们 Retrofit 类中的两个成员变量都有重大的关系。这里最终返回的是个 ExecutorCallAdapterFactory ，话说我们一开始就不知道这个 CallAdapter 是什么，更不用说这个 Factory 了，那我们先看看这个 ExecutorCallAdapterFactory 吧。

```
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);

    }

  };

}



//... ... 省略}
```

这里我们可以看到，把我们传进来的 Executor 保存起来了，这个 Executor 想必就是 MainThreadExecutor 了。至于 get 方法，我们暂时还不知道哪里用到了，所以后面的暂时不看了，到了这里还是不知道 CallAdapter.Factory 干嘛用的。

看来 Builder 方法很复杂呀，写了这么多只是讲了个 Platform，不过幸好这里面也包括了 Executor 和 CallAdapter.Factory ，那么现在我们正式看看 Builder.build()方法。

```
public Retrofit build() {

    // 这一句告诉我们，baseUrl 是必不可少的。

    if (baseUrl == null) {

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

    }



    // 这里如果你没配置 callFactory ，会默认配置为 OkHttpClient

    okhttp3.Call.Factory callFactory = this.callFactory;

    if (callFactory == null) {

      callFactory = new OkHttpClient();

    }
```

```
    // 同样的，没配置的话，会默认配置为 Platform 的 defaultCallbackExecuto
r，这里我们之前分析过，它所返回的就是 MainThreadExecutor

    Executor callbackExecutor = this.callbackExecutor;

    if (callbackExecutor == null) {

      callbackExecutor = platform.defaultCallbackExecutor();

    }



    //这里会把你所配置的 CallAdapter.Factory 加到 List 里去，最后把 Platfo
rm 默认的 defaultCallAdapterFactory 即 ExecutorCallAdapterFactory 加到 Li
st 的最后边，

    // Make a defensive copy of the adapters and add the default Call a
dapter.

    List<CallAdapter.Factory> callAdapterFactories = new ArrayList<>(t
his.callAdapterFactories);

    callAdapterFactories.add(platform.defaultCallAdapterFactory(callba
ckExecutor));



    //这里一样会把你配置的 Converter.Factory 加到 List 里去，但是会把一个 B
uiltInConverters 加到第一个，而不是最后一个，请注意这点。

    // Make a defensive copy of the converters.

    List<Converter.Factory> converterFactories =

        new ArrayList<>(1 + this.converterFactories.size());



    // Add the built-in converter factory first. This prevents overridi
ng its behavior but also

    // ensures correct behavior when using converters that consume all
types.
```

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

    converterFactories.addAll(this.converterFactories);



    //最后返回一个 Retrofit 对象

    return new Retrofit(callFactory, baseUrl, unmodifiableList(convert
erFactories),

        unmodifiableList(callAdapterFactories), callbackExecutor, valid
ateEagerly);

  }
```

到这里，我们的 Retrofit 就构建完成了。如果按照我们 基本使用 中的例子，那么此刻，Retrofit 成员变量的值如下：

- serviceMethodCache ：暂时为空的 HashMap 集合
- callFactory ： OkHttpClient 对象
- baseUrl ： 根据配置的 baseUrl " https://api.github.com/ " 字符串， 构建出了一个 HttpUrl 对象
- converterFactories ：一个 ArrayList 对象，里面存放着一个 BuiltInConverters 对象
- callAdapterFactories ： 一个 ArrayList 对象，里面存放着一个 ExecutorCallAdapterFactory 对象
- callbackExecutor ：MainThreadExecutor 对象
- validateEagerly ：默认值 false

2. 创建网络请求接口实例，即 GitHubService service = retrofit.create(GitHubService.class);

接下来我们看看是怎样获得 GitHubService 实例的。
同样上源码，**注意**这里的 create 是非常重要的一个方法，这里使用了 外观模式和 代理模式。

```
  public <T> T create(final Class<T> service) {

  Utils.validateServiceInterface(service);

  if (validateEagerly) {

    eagerlyValidateMethods(service);
```

```
    }

    return (T) Proxy.newProxyInstance(service.getClassLoader(), new Clas
s<?>[] { service },

        new InvocationHandler() {

          private final Platform platform = Platform.get();



          @Override public Object invoke(Object proxy, Method method, @Nu
llable 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);

            }

            ServiceMethod<Object, Object> serviceMethod =

                (ServiceMethod<Object, Object>) loadServiceMethod(metho
d);

            OkHttpCall<Object> okHttpCall = new OkHttpCall<>(serviceMetho
d, args);

            return serviceMethod.adapt(okHttpCall);

          }

        });
```

```
}
```

这里我们看到了 validateEagerly 变量，让我们看看它到底控制了什么。进 eagerlyValidateMethods 方法。

```
private void eagerlyValidateMethods(Class<?> service) {
  Platform platform = Platform.get();
  for (Method method : service.getDeclaredMethods()) {
    if (!platform.isDefaultMethod(method)) {
      loadServiceMethod(method);
    }
  }
}

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;

  }
```

这里又见到了 Platform ，在 Retrofit.Builder 我们知道它返回的是 Android() 对象。 接着是个 循环 ，循环取出接口中的 Method ，接着调用 loadServiceMethod 。 loadServiceMethod 里面会根据 Method 生成一个 ServiceMethod，然后存入 serviceMethodCache ， 那么我们大概知道，这是属于**提前验证**，会提前把接口中每个方法进行**解析**得到一个 **ServiceMethod** 对象，然后存入缓存中。 在 loadServiceMethod 中会取缓存中的值，如果有就直接返回 ServiceMethod。

由此可以知道 **validateEagerly** 变量是用于 判断是否需要提前验证解析的。

create 方法中 继续往下走，会看到 return 一个 代理对象 Proxy ，并转成了 T 类型，即 GitHubService 。
此时我们这句代码 `GitHubService service = retrofit.create(GitHubService.class);` 中的 service 有值了，它指向一个实现了 GitHubService 接口的 代理对象 Proxy 。

3. 拿到 Call 对象 ， 即 Call<List<Repo>> repos = service.listRepos("octocat");

这里我们的 service 是个代理对象，所以执行 listRepos 方法时， 会先走 InvocationHandler 中的 invoke 方法。

```
  public <T> T create(final Class<T> service) {

    Utils.validateServiceInterface(service);

    if (validateEagerly) {

      eagerlyValidateMethods(service);

    }

    return (T) Proxy.newProxyInstance(service.getClassLoader(), new Clas
s<?>[] { service },

        new InvocationHandler() {
```

BA... 2246

```
        private final Platform platform = Platform.get();



        @Override public Object invoke(Object proxy, Method method, @Nu
llable Object[] args)

            throws Throwable {

          // If the method is a method from Object then defer to normal
invocation.

          // 如果这个方法是声明在 Object 类中，那么不拦截，直接执行

          if (method.getDeclaringClass() == Object.class) {

            return method.invoke(this, args);

          }

          // 这个总是返回的 false，所以不用关心

          if (platform.isDefaultMethod(method)) {

            return platform.invokeDefaultMethod(method, service, proxy,
 args);

          }



          // 下面三行代码非常重要，重点分析，分别对应 3.1 3.2 3.3 三个小节

          ServiceMethod<Object, Object> serviceMethod =

              (ServiceMethod<Object, Object>) loadServiceMethod(metho
d);

          OkHttpCall<Object> okHttpCall = new OkHttpCall<>(serviceMetho
d, args);

          return serviceMethod.adapt(okHttpCall);

        }
```

```
      });

  }
```

3.1 ServiceMethod<Object, Object> serviceMethod = (ServiceMethod<Object, Object>) loadServiceMethod(method);

首先 ServiceMethod 我们之前猜测过，应该是对 Method 的一个封装， 而这个 loadServiceMethod ，如果你还记得的话，我们在 create 的时候就碰到过，eagerlyValidateMethods 这个方法内部调用过 loadServiceMethod ，是为了加载这个 ServiceMethod 。现在我们来深入分析这个 loadServiceMethod 方法。

```
  ServiceMethod<?, ?> loadServiceMethod(Method method) {

    // 首先从 缓存 serviceMethodCache 中取 ServiceMethod ，如果存在就返回，
不存在继续往下走。

    // 也就是说 我们的 ServiceMethod 只会创建一次。

    ServiceMethod<?, ?> result = serviceMethodCache.get(method);

    if (result != null) return result;



    synchronized (serviceMethodCache) {

      //这里又从缓存取了一遍，看到这里有没有一种熟悉的感觉，是不是跟 DCL 单例模
式特别像，双重校验。

      result = serviceMethodCache.get(method);

      if (result == null) {

        result = new ServiceMethod.Builder<>(this, method).build();

        serviceMethodCache.put(method, result);

      }

    }

    return result;
```

```
}
```

到这里其实 loadServiceMethod 已经分析完了，很简单，就是个 DCL 单例模式，然后获得 ServiceMethod 。
那其实我们现在的分析任务就很明确了，弄清楚这个 ServiceMethod 究竟是什么 。

3.1.1 ServiceMethod 分析

```
final class ServiceMethod<R, T> {

  // ... 省略部分代码

  private final okhttp3.Call.Factory callFactory;

  private final CallAdapter<R, T> callAdapter;

  private final HttpUrl baseUrl;

  private final Converter<ResponseBody, R> responseConverter;



  // 同样先猜猜什么意思吧

  // 应该是网络请求的 Http 方法，比如 GET、POST 啥的

  private final String httpMethod;

  // 相对地址 ，应该就是 "user/{user}/repos" 这一段

  private final String relativeUrl;

  // http 请求头

  private final Headers headers;

  // 网络请求的 http 报文的 body 的类型

  private final MediaType contentType;

  // 是否有 body

  private final boolean hasBody;

  // post 提交数据时，是否使用 表单提交 方式
```

892872246

```
private final boolean isFormEncoded;

// post 提交数据时，是否使用 Mutipart 方式，一般用来文件上传

private final boolean isMultipart;

// 方法参数处理器，应该是解析方法中的 参数 的吧，这个估计也得详细分析下。

private final ParameterHandler<?>[] parameterHandlers;



ServiceMethod(Builder<R, T> builder) {

  this.callFactory = builder.retrofit.callFactory();

  this.callAdapter = builder.callAdapter;

  this.baseUrl = builder.retrofit.baseUrl();

  this.responseConverter = builder.responseConverter;

  this.httpMethod = builder.httpMethod;

  this.relativeUrl = builder.relativeUrl;

  this.headers = builder.headers;

  this.contentType = builder.contentType;

  this.hasBody = builder.hasBody;

  this.isFormEncoded = builder.isFormEncoded;

  this.isMultipart = builder.isMultipart;

  this.parameterHandlers = builder.parameterHandlers;

}



// ... 省略部分代码
```

首先看看 ServiceMethod 的构造方法。 也是通过建造者模式构建的。其中很多变量其实都很熟悉了，比如 callFactory 、 baseUrl 。 对于 callAdapter、responseConverter 我们别弄混了，我们在 Retrofit 类中的变量是 callAdapterFactories 和 converterFactories ， 是它们的工厂，是生产它们的地方。

接下来看 Builder 吧，毕竟这是真正做事的。

```
  public ServiceMethod build() {

      // 拿到具体的 CallAdapter 即 网络请求适配器，具体看 3.1.1.1

      callAdapter = createCallAdapter();

      // 根据上面拿到的 callAdapter 获取 响应类型，在 3.1.1.1 小节分析完后可知
道

      // 在我们的例子中 responseType = java.util.List<java.lang.Integer>

      responseType = callAdapter.responseType();

      if (responseType == Response.class || responseType == okhttp3.Respo
nse.class) {

        throw methodError("'"

            + Utils.getRawType(responseType).getName()

            + "' is not a valid response body type. Did you mean ResponseB
ody?");

      }

      // 获取 响应转换器 ，具体看 3.1.1.2 小节

      responseConverter = createResponseConverter();

      // 解析网络请求接口中方法的注解，这里我们就只有一个 @GET 注解，具体看 3.
1.1.3 小节

      // 这里解析完可以拿到 Http 请求方法、请求体、相对 url、相对 url 中的参数

      for (Annotation annotation : methodAnnotations) {

        parseMethodAnnotation(annotation);
```

```
    }

    //解析完方法上的注解后，做校验

    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 reques
t body (e.g., @POST).");

      }

      if (isFormEncoded) {

        throw methodError("FormUrlEncoded can only be specified on HTTP
 methods with "

            + "request body (e.g., @POST).");

      }

    }



    // 解析当前方法的参数，这里就我们的例子而言

    // parameterAnnotationsArray 就是 @Path ，所以这里的 length 就是 1

    int parameterCount = parameterAnnotationsArray.length;

    parameterHandlers = new ParameterHandler<?>[parameterCount];
```

```
    for (int p = 0; p < parameterCount; p++) {

      // parameterTypes 是参数类型，就本例而言是 String

      Type parameterType = parameterTypes[p];

      if (Utils.hasUnresolvableType(parameterType)) {

        throw parameterError(p, "Parameter type must not include a type
 variable or wildcard: %s",

            parameterType);

      }



      // 拿到第一个参数的 注解数组

      Annotation[] parameterAnnotations = parameterAnnotationsArray
[p];

      if (parameterAnnotations == null) {

        throw parameterError(p, "No Retrofit annotation found.");

      }

      // 解析参数

      // p : 0

      // parameterType : String

      // parameterAnnotations : 虽然是数组，但是就一个元素 @Path

      // 这个 parseParameter 就不分析了，大家自己看看源码就清楚了，无非就是
构建 ParameterHandler 数组，而这个 ParameterHandler 其实就是负责解析 API 定
义时每个方法的参数，并在构造 HTTP 请求时设置参数

      parameterHandlers[p] = parseParameter(p, parameterType, parameter
Annotations);

    }
```

```
      // 解析完方法中参数的注解后，做校验

      if (relativeUrl == null && !gotUrl) {

        throw methodError("Missing either @%s URL or @Url parameter.", ht
tpMethod);

      }

      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 @Pa
rt.");

      }



      return new ServiceMethod<>(this);

    }
```

3.1.1.1 createCallAdapter ()

```
  private CallAdapter<T, R> createCallAdapter() {

      // 拿到网络请求接口里方法的返回值类型，在我们的例子中会返回如下类型

      // retrofit2.Call<java.util.List<java.lang.Integer>>

      Type returnType = method.getGenericReturnType();
```

```
      if (Utils.hasUnresolvableType(returnType)) {

        throw methodError(

            "Method return type must not include a type variable or wildc
ard: %s", returnType);

      }

      // 如果返回类型是 void ，抛出异常

      if (returnType == void.class) {

        throw methodError("Service methods cannot return void.");

      }

      // 拿到方法的 注解 ，在我们的例子中就是如下所示，大家可以自己实验下

      // @retrofit2.http.GET(value=users/{user}/repos)

      Annotation[] annotations = method.getAnnotations();

      try {

        // 拿到注解后，返回个 CallAdapter ，跟进去看看究竟是做了什么

        return (CallAdapter<T, R>) retrofit.callAdapter(returnType, annot
ations);

      } catch (RuntimeException e) { // Wide exception range because fact
ories are user code.

        throw methodError(e, "Unable to create call adapter for %s", retu
rnType);

      }

    }



  public CallAdapter<?, ?> callAdapter(Type returnType, Annotation[] ann
otations) {
```

```
    // 这里会去调用 nextCallAdapter

    return nextCallAdapter(null, returnType, annotations);

  }



  // 这里的参数大家注意

  // skipPast 上面传的是 null

  // returnType 就是  retrofit2.Call<java.util.List<java.lang.Integer>>

  // annotations 在我们的例子中就是 @retrofit2.http.GET(value=users/{user}
/repos)

  public CallAdapter<?, ?> nextCallAdapter(@Nullable CallAdapter.Factory
 skipPast, Type returnType, Annotation[] annotations) {

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

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



    // callAdapterFactories 是一个 ArrayList 对象，里面存放着一个 ExecutorC
allAdapterFactory 对象 ，这个是在 Retrofit Builder 的时候创建的，也就是我们
上面所说的生产 CallAdapter 的地方，大家可以回过头去看看。 这里的 skipPast 是 n
ull，  所以 indexOf 肯定返回的 -1，  所以这里 start = 0

    int start = callAdapterFactories.indexOf(skipPast) + 1;

    // 循环，  这里由于我们的 callAdapterFactories 只有一个 元素，  所以直接看
ExecutorCallAdapterFactory 的 get 方法

    for (int i = start, count = callAdapterFactories.size(); i < count; i
++) {

      CallAdapter<?, ?> adapter = callAdapterFactories.get(i).get(return
Type, annotations, this);

      if (adapter != null) {

        return adapter;
```

```
      }

    }



    // 错误信息 builder

    StringBuilder builder = new StringBuilder("Could not locate call adap
ter for ")

        .append(returnType)

        .append(".\n");

    if (skipPast != null) {

      builder.append("  Skipped:");

      for (int i = 0; i < start; i++) {

        builder.append("\n   * ").append(callAdapterFactories.get(i).getC
lass().getName());

      }

      builder.append('\n');

    }

    builder.append("  Tried:");

    for (int i = start, count = callAdapterFactories.size(); i < count; i
++) {

      builder.append("\n   * ").append(callAdapterFactories.get(i).getCl
ass().getName());

    }

    throw new IllegalArgumentException(builder.toString());

  }
```

BAT交 892246

到这里，我们别忘了我们是在干嘛，我们是在获取 CallAdapter<T, R> ，好了，继续看 ExecutorCallAdapterFactory 的 get 方法。 解释都在代码注释里哟，一定要看看才知道现在到底是在干啥。话说源码分析，还是得靠自己认认真真读一次源码才行。

```
 @Override

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

   // getRawType 会返回该类型的原始类类型 ，比如传进去的是 List<? extends R
unnable> 会返回 List.class

   // 那么在我们的例子中，我们的 returnType 是 retrofit2.Call<java.util.Li
st<java.lang.Integer>>

   // 那么 getRawType 后，返回的是 retrofit2.Call ，所以这里是相等的

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

     return null;

   }

   // 根据 returnType 拿到 responseType ，这里就不跟进了，可以自己去看看

   // 在我们的例子中， responseType = java.util.List<java.lang.Integer>

   final Type responseType = Utils.getCallResponseType(returnType);

   // 最后返回一个 CallAdapter

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

     @Override public Type responseType() {

       return responseType;

     }



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

       return new ExecutorCallbackCall<>(callbackExecutor, call);
```

```
    }

  };

}
```

到这里，其实我们大概知道这个 CallAdapter 有什么用了，就是提供两个东西

- 网络请求响应要返回的类型 responseType
- retrofit2.Call< T > ，注意这里不是 okhttp3 下的 Call ，这里暂不深究。

因为我们不要忘了现在在做什么，我们现在是在获取 ServiceMethod 中的 callAdapter 变量值。所以看到这里返回了一个 CallAdapter 对象即可。

3.1.1.2 createResponseConverter ()

这里个方法是获取 响应转换器，就是把网络请求得到的响应数据转换成相应的格式。

```
private Converter<ResponseBody, T> createResponseConverter() {

    // 拿到方法上所有的注解，在我们的例子中就只有 @GET 注解

    Annotation[] annotations = method.getAnnotations();

    // 这里的 responseType 就是上面我们得到的 List<Integer>

    try {

      return retrofit.responseBodyConverter(responseType, annotation
s);

    } catch (RuntimeException e) { // Wide exception range because fact
ories are user code.

      throw methodError(e, "Unable to create converter for %s", respons
eType);

    }

  }
```

这里想必大家也知道套路了，跟获取 CallAdapter 是一样的，代码就不贴了，代码里同样是循环遍历 Retrofit 里的 converterFactories 变量。而这个 converterFactories 在我们的例子中是没有设置转换器的，所以它也只有一个

默认的元素，即 BuiltInConverters 。 那么我们直接查看 它的 responseBodyConverter 方法。

```
final class BuiltInConverters extends Converter.Factory {

  // 注意这里的参数，别忘了到底是什么

  // type : 就是我们的 responseType ，即 List<Integer>

  // annotations : 这里我们方法的注解只有一个，所以就是 @GET

  @Override

  public Converter<ResponseBody, ?> responseBodyConverter(Type type, Ann
otation[] annotations,

      Retrofit retrofit) {

    if (type == ResponseBody.class) {

      return Utils.isAnnotationPresent(annotations, Streaming.class)

          ? StreamingResponseBodyConverter.INSTANCE

          : BufferingResponseBodyConverter.INSTANCE;

    }

    if (type == Void.class) {

      return VoidResponseBodyConverter.INSTANCE;

    }

    return null;

  }
```

通过这里我们可以知道，其实它会返回 null 。 所以我们 **ServiceMethod 中的 Builder 中的 responseConverter 变量就等于 null 。**

3.1.1.3 parseMethodAnnotation ()

我们来看看 解析方法注解 ，注意我们例子中这个方法里传的参数是 @GET 注解

BAT交流群：892872246

```
  private void parseMethodAnnotation(Annotation annotation) {

    if (annotation instanceof DELETE) {

      parseHttpMethodAndPath("DELETE", ((DELETE) annotation).value(),
false);

    } else if (annotation instanceof GET) {

      //我们这里是 GET 注解，所以进这个方法

      parseHttpMethodAndPath("GET", ((GET) annotation).value(), fals
e);

    } else if (annotation instanceof HEAD) {

      parseHttpMethodAndPath("HEAD", ((HEAD) annotation).value(), fals
e);

      if (!Void.class.equals(responseType)) {

        throw methodError("HEAD method must use Void as response type.
");

      }

    }

    // 省略后续代码，后续还有很多其他类型的判断

   }



  // 这里的三个参数的值

  // httpMethod : GET

  // value : users/{user}/repos

  // hasBody : false

  private void parseHttpMethodAndPath(String httpMethod, String value,
boolean hasBody) {
```

```
    // 此处判断 httpMethod 的值是否存在，说明只允许一个 HTTP 方法存在
    if (this.httpMethod != null) {
      throw methodError("Only one HTTP method is allowed. Found: %s and
%s.",
          this.httpMethod, httpMethod);
    }
    this.httpMethod = httpMethod;
    this.hasBody = hasBody;


    if (value.isEmpty()) {
      return;
    }


    // 下面是解析 value 中的 相对 url
    // Get the relative URL path and existing query string, if present.
    int question = value.indexOf('?');
    if (question != -1 && question < value.length() - 1) {
      // Ensure the query string does not have any named parameters.
      String queryParams = value.substring(question + 1);
      Matcher queryParamMatcher = PARAM_URL_REGEX.matcher(queryParam
s);
      if (queryParamMatcher.find()) {
        throw methodError("URL query string \"%s\" must not have replac
e block. "
```

```
            + "For dynamic query parameters use @Query.", queryParams);

      }

    }



    this.relativeUrl = value;

    // 相对地址中的参数名字，这里不具体分析了，可以把结果告诉你

    // 在我们的例子中 value = “users/{user}/repos”

    // 这里的 relativeUrlParamNames 是个 Set<String> 集合 ，里面只有一个元素 user 。

    this.relativeUrlParamNames = parsePathParameters(value);

  }
```

至此，我们的 Builder 把 Http 的方法以及它的 Url 给分析完了，现在只剩 **参数解析了**。参数解析在 ServiceMethod 的 build 方法里已经讲过了 ，记得看注释。

呼~ 终于讲完了 ServiceMethod 的构造。这么大篇幅，由此可以看出 ServiceMethod 这个类非常重要。现在来总结一下，我们究竟拥有了些什么。

- callFactory : ExecutorCallAdapterFactory 实例
- callAdapter : ExecutorCallAdapterFactory 中的 get 方法返回的 CallAdapter 实例
- baseUrl ： HttpUrl 实例
- responseConverter : 由于我们没设置，所以为 null
- httpMethod : 字符串 GET
- relativeUrl ： 字符串 users/{user}/repos
- headers : 没有设置 Headers ，所以为 null
- contentType : null
- hasBody : false
- isFormEncoded : false
- isMultipart : false
- parameterHandlers : 就我们例子而已，该数组有一个元素，Path 对象，它是 ParameterHandler 抽象类里的一个静态内部类。

由此可以看出，**ServiceMethod 对象包含了访问网络的所有基本信息。**

好吧，接下来还是得继续前行，别忘了，我们构建 ServiceMethod 只是在 invoke 方法内，并且这还只是第一步。接下来看第二步。

BAT交流群：892872246

3.2 OkHttpCall<Object> okHttpCall = new OkHttpCall<>(serviceMethod, args);

这里是 new 一个 OkHttpCall 对象，这个 OkHttpCall 是 Retrofit 的 Call，它里面就是做请求的地方，会有 request、enqueue 等同步、异步请求方法，但是在这里面真正执行请求的是 okhttp3.Call ，即把请求委托给 okHttp 去执行。下面简要看看它的构造方法和一些成员变量吧，因为这里只是 new 操作，所以暂时不分析其余方法，用到的时候再看。

```
final class OkHttpCall<T> implements Call<T> {

  // 含有所有网络请求参数信息的 ServiceMethod

  private final ServiceMethod<T, ?> serviceMethod;

  private final @Nullable Object[] args;



  private volatile boolean canceled;



  // 实际进行网络请求的 Call

  private @Nullable okhttp3.Call rawCall;

  @GuardedBy("this") // Either a RuntimeException, non-fatal Error, or I
OException.

  private @Nullable Throwable creationFailure;

  @GuardedBy("this")

  private boolean executed;



  // 传入配置好的 ServiceMethod 和 请求参数

  OkHttpCall(ServiceMethod<T, ?> serviceMethod, @Nullable Object[] args)
{
```

```
    this.serviceMethod = serviceMethod;

    this.args = args;

  }
```

这样就把 OkHttpCall 给构建好了，接下来看第三步。

3.3 return serviceMethod.adapt(okHttpCall);

直接上代码

```
 T adapt(Call<R> call) {

    return callAdapter.adapt(call);

  }
```

这是 前面构建好的 ServiceMethod 中的 adapt 方法，会去调用 callAdapter 的 adapt 方法，我们知道 ServiceMethod 中的 callAdapter 是 ExecutorCallAdapterFactory 中的 get 方法返回的 CallAdapter 实例。而这个实例的 adapt 方法会返回一个 ExecutorCallbackCall 对象。

```
 <!-- ExecutorCallAdapterFactory 内部类 -->

 static final class ExecutorCallbackCall<T> implements Call<T> {

    // 这里在之前创建 ExecutorCallAdapterFactory 时，就知道它的值了，就是 Mai
nThreadExecutor ，用来切换线程的

    final Executor callbackExecutor;

    // 这就是刚刚传进来的 OkHttpCall

    final Call<T> delegate;



    ExecutorCallbackCall(Executor callbackExecutor, Call<T> delegate) {

      this.callbackExecutor = callbackExecutor;

      this.delegate = delegate;
```

```
    }
```

到这里为止，我们已经成功的返回了一个 `Call<List<Integer>>`

4. 调用 Call 的 enqueue

趁热打铁，我们执行异步请求，看看怎样切换线程的。

```
<!-- ExecutorCallbackCall 内部 -->

@Override

public void enqueue(final Callback<T> callback) {

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



    // 真正的 Call 去执行请求

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

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

        // 回调后 利用 MainThreadExecutor 中的 Handler 切换到主线程中去。

        callbackExecutor.execute(new Runnable() {

          @Override public void run() {

            if (delegate.isCanceled()) {

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

              callback.onFailure(ExecutorCallbackCall.this, new IOExcep
tion("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);
          }
        });
      }
    });
  }
```

可以看到是 delegate 执行了 enqueue 操作，而 delegate 就是我们的 OkHttpCall ，在 OkHttpCall 里的 enqueue 方法是这样工作的。
通过 `okhttp3.Call call = serviceMethod.toCall(args);` 构建一个真正执行请求的 Call ，即把请求交给 okhttp 去完成。而构建一个 Call 利用到了 ServiceMethod 中的 ParameterHandler 对象，这个对象是用来处理参数的。 它会把具体参数的值与 RequestBuilder 绑定起来。当然也用到了 ServiceMethod 自己，ServiceMethod 类似请求响应的大管家。

别忘了拿到响应后，在 okhttp3.Callback 中会去调用 `response = parseResponse(rawResponse);` 将响应转换成自己想要的格式，即定义的 Converter 。

第六章、RXJava 响应式编程框架设计

BAT交流群：892872246

6.1 链式调用

Rxjava 中链式调用怎么实现的？

```
        Observable.just("a")     //Observable1

                .map(new Func1<String, String>() {  //Observable2

                    @Override

                    public String call(String s) {

                        System.out.print(Thread.currentThread().getName()
+ ":first--" + s +"\n");

                        return s + s;

                    }

                })

                .subscribe(new Subscriber<String>() { //代码⑥ Subscriber

                    @Override

                    public void onCompleted() {

                        System.out.print(Thread.currentThread().getName()+
"\n");

                        System.out.print("completed"+"\n");

                    }



                    @Override

                    public void onError(Throwable e) {

                        System.out.print("error");

                    }
```

```
                @Override
                public void onNext(String s) {
                    System.out.println(s);
                }
            });
    }
```

先说说自己的理解，若把整个链条看成一个整体对象，那么 just 创建被观察者对象，而 subscribe()里的 Subscriber 作为观察者；若每一步都分开看，just()和 subscribe()中间的操作符即是观察者，又是被观察者。

Observable 中每个操作符基本都会创建出一个新的 Observable；因此可以解理成后一级的操作符去观察前一个 Observable 对象；以上例来说，.subscribe 的 Subscriber 所观察的对象就是.map 返回的 Observable2，而.map 的 Subscriber 所观察的对象就是 Observable.just("a")得到的对象 Observable1；

下面介绍实现代码，整个链式调用真正开始的地方是.subscribe()，我们就从这里开始。省略掉一些代码，只看关键部分如下：

```
private static <T> Subscription subscribe(Subscriber<? super T> subscrib
er, Observable<T> observable) {
    ...
    try {
        // allow the hook to intercept and/or decorate
        hook.onSubscribeStart(observable, observable.onSubscribe).cal
l(subscriber); //代码①
        return hook.onSubscribeReturn(subscriber);
    }
    ...}
```

BAT交...892246

hook.onSubscribeStart(observable, observable.onSubscribe)得到的对象就是observable.onSubscribe，而此处的 observable 明显就是 this，也就是上例中的observable2 对象，即把 subscriber 传入到了 observable2 里面以供其调用。

再跟着代码进入 observable2（.map 操作符）的实现。其主要实现是 lift 和OperatorMap。如下：

```
    public final <R> Observable<R> map(Func1<? super T, ? extends R> func)
{

        return lift(new OperatorMap<T, R>(func));

    }
```

lift 和 OperatorMap 各自干了什么事情呢？先看 OperatorMap，Func1 也作为构造参数传入。关键代码：

```
    @Override

    public Subscriber<? super T> call(final Subscriber<? super R> o) {  //代码②

        return new Subscriber<T>(o) {



            @Override

            public void onCompleted() {

                o.onCompleted();

            }



            @Override

            public void onError(Throwable e) {

                o.onError(e);

            }
```

```
        @Override
        public void onNext(T t) {
            try {
                o.onNext(transformer.call(t));
            } catch (Throwable e) {
                Exceptions.throwOrReport(e, this, t);
            }
        }

    };
}
```

这里 new 出了一个观察者对象 Subscriber，它实现了什么功能通过 o.onNext(transformer.call(t));即将例子中的 Func1 代码执行后将结果传入到下一层。即这里运行了 Func1 的代码。

再看 lift()操作符，看其返回值也就是我们定义的 observable2 对象。因此 subscribe 里的"代码①"的 call 即是此处 observable2 里 OnSubscribe 的 call 方法；再看 call 方法，“代码④”部分则是调用到了 observable1 对象里 OnSubscribe 的 call 方法，而“代码③”将 Func1 操作动作转变为 Subscriber，通过 call(o)完成对下一级 Subscriber 的引用。

```
public final <R> Observable<R> lift(final Operator<? extends R, ? super T> operator) {
    return new Observable<R>(new OnSubscribe<R>() {
        @Override
        public void call(Subscriber<? super R> o) {
            try {
```

```
                    Subscriber<? super T> st = hook.onLift(operator).call
(o); //代码③

                    try {

                        // new Subscriber created and being subscribed wit
h so 'onStart' it

                        st.onStart();

                        onSubscribe.call(st);  //代码④

                    } catch (Throwable e) {

                        // localized capture of errors rather than it skip
ping all operators

                        // and ending up in the try/catch of the subscribe
method which then

                        // prevents onErrorResumeNext and other similar ap
proaches to error handling

                        Exceptions.throwIfFatal(e);

                        st.onError(e);

                    }

                } catch (Throwable e) {

                    Exceptions.throwIfFatal(e);

                    // if the lift function failed all we can do is pass t
he error to the final Subscriber

                    // as we don't have the operator available to us

                    o.onError(e);

                }

            }

        });
```

```
    }
```

到这里“代码④”执行，即到了 observable1 对象，也就是例子中

Observable.just("a")所得到对象的 OnSubscribe 的 call()方法，如下：

```
 public final static <T> Observable<T> just(final T value) {

       return ScalarSynchronousObservable.create(value);

   }ScalarSynchronousObservable 类代码如下：



public static final <T> ScalarSynchronousObservable<T> create(T t) {

       return new ScalarSynchronousObservable<T>(t);

}

protected ScalarSynchronousObservable(final T t) {

       super(new OnSubscribe<T>() {



           @Override

           public void call(Subscriber<? super T> s) {

               /*

                * We don't check isUnsubscribed as it is a significant p
erformance impact in the fast-path use cases.

                * See PerfBaseline tests and https://github.com/Reactive
X/RxJava/issues/1383 for more information.

                * The assumption here is that when asking for a single i
tem we should emit it and not concern ourselves with

                * being unsubscribed already. If the Subscriber unsubscr
ibes at 0, they shouldn't have subscribed, or it will
```

```
                * filter it out (such as take(0)). This prevents us from
paying the price on every subscription.

                */

               s.onNext(t);  //代码⑤

               s.onCompleted();

           }



       });

       this.t = t;

   }
```

其中"代码⑤"是关键点，t 即是我们 just 传入的"a",s 则是代码④传入的 st，它其实是 observable2 的 Subscriber(观察者)，相当于 observable1 持有 observable2 的引用。通过 s.onNext(t)，完成了 observable1 向下一层的 observable2 的回调，也就是 Func1 对象所在的 Subscriber(OperatorMap)，再通过 o.onNext(transformer.call(t));回到例子中“代码⑥”，至此，整个调用链完成。

上面的分析比较混乱，重新梳理代码执行流程 ：
1、subscribe 里，hook.onSubscribeStart(observable, observable.onSubscribe).call(subscriber); //代码①
2、map 里，通过 lift()将 Func1 操作符生成 Subserber，Subscriber<? super T> st = hook.onLift(operator).call(o); //代码③
onSubscribe.call(st); //代码④
3、just 里 create()， s.onNext(t); //代码⑤
4、map 里， OperatorMap 里对象， o.onNext(transformer.call(t));
5、subscribe 的 Subscriber（）；

Observable 的所有链式调用，知道两个其两个关键点即可梳理清楚整个数据流传递原理；

Observable.onSubscribe 对象，完成以 call 方法来向上一层传递；
Subserber 向下一层的 Subserber 调用；

6.2 扩展的观察者模式

BAT交流群：892872246

1. RxJava 的观察者模式解读

RxJava 有四个基本概念：

Observable (可观察者，即被观察者)
Observer (观察者)
subscribe (订阅)
事件
Observable 和 Observer 通过 subscribe() 方法实现订阅关系，从而 Observable 可以在需要的时候发出事件来通知 Observer

Android 常见的观察者模式：

与传统观察者模式不同， RxJava 的事件回调方法除了普通事件 onNext() （相当于 onClick() / onEvent()）之外，还定义了两个特殊的事件：onCompleted() 和 onError()

onCompleted()：事件队列完结。RxJava 不仅把每个事件单独处理，还会把它们看做一个队列。RxJava 规定，当不会再有新的 onNext() 发出时，需要触发 onCompleted() 方法作为标志

onError()：事件队列异常。在事件处理过程中出异常时，onError() 会被触发，同时队列自动终止，不允许再有事件发出

在一个正确运行的事件序列中, onCompleted() 和 onError() 有且只有一个，并且是事件序列中的最后一个。需要注意的是，onCompleted() 和 onError() 二者也是互斥的，即在队列中调用了其中一个，就不应该再调用另一个。

BAT交流群：892872246

2. 扩展的观察者模式

Observable 和 Observer 通过 subscribe() 方法实现订阅关系
RxJava 的事件回调方法 onNext() 、onCompleted() 和 onError()
如果一个 Observable 中没有任何的 Observer ，那么这个 Observable 是不会发出任何事件的
第三点说明：

在 RxJava 中 Obseravable 有两种启动方式：热启动 和 冷启动

热启动 Obseravable 在任何时候都会发送消息，即使没有任何观察者监听它
冷启动 Obseravable 只有在至少有一个监听者时才会发送消息
热启动平时用不到，所以我们工作中用到的都是冷启动

3. RxJava 的实现和使用

RxJava 的实现主要有如下三步：

第一步：创建观察者 Observer。

Observer 即观察者，它决定事件触发的时候将有怎样的行为。注意:Observer 是个接口。
第二步：创建被观察者 Observable。

第三步：Subscribe(订阅)（被观察者 Observable 订阅观察者 Observer，注意：不同于普通观察者，这里是被观察者订阅观察者）。

创建 Observable 和 Observer 之后，再 subscribe() 法将它们联结起来，整条链子就可以 作 。代码形式很简单:

BAT交流群：892872246

所以贴出之前实现过的代码：

BAT交流群：892872246

RxHelloWorld.java

```java
    }

    @SuppressWarnings("deprecation")
    private static void testObservable() {
        // 创建观察者Observer
        Observer<String> observer = new Observer<String>() {

            @Override
            public void onCompleted() {
                System.out.println("observer onCompleted()");
            }

            @Override
            public void onError(Throwable e) {
                System.out.println("observer onError(): " + e.getMessage());
            }

            @Override
            public void onNext(String t) {
                System.out.println("observer onNext():" + t);
            }
        };
        // 创建被观察者Observable
        Observable<String> observable = Observable
                .create(new OnSubscribe<String>() {

                    @Override
                    public void call(Subscriber<? super String> subscriber) {
                        subscriber.onNext("Hello World!");
                        subscriber.onCompleted();
                    }
                });
        // 被观察者订阅(subscribe)观察者
        observable.subscribe(observer);

    }

}
```

编译运行：

Console

```
<terminated> RxHelloWorld [Java Application]
observer onNext():Hello World!
observer onCompleted()
```

这时制造一个异常再看下输出：

BAT交流群 892872246

RxHelloWorld.java

```java
    @SuppressWarnings("deprecation")
    private static void testObservable() {
        // 创建观察者Observer
        Observer<String> observer = new Observer<String>() {

            @Override
            public void onCompleted() {
                System.out.println("observer onCompleted()");
            }

            @Override
            public void onError(Throwable e) {
                System.out.println("observer onError(): " + e.getMessage());
            }

            @Override
            public void onNext(String t) {
                System.out.println("observer onNext():" + t);
            }
        };
        // 创建被观察者Observable
        Observable<String> observable = Observable
                .create(new OnSubscribe<String>() {

                    @Override
                    public void call(Subscriber<? super String> subscriber) {
                        subscriber.onNext("Hello World!");
                        int i = 1 / 0;
                        subscriber.onCompleted();
                    }
                });
        // 被观察者订阅(subscribe)观察者
        observable.subscribe(observer);

    }

}
```

故意制造一个异常

编译运行：

Console

```
<terminated> RxHelloWorld [Java Applicatio
observer onNext():Hello World!
observer onError(): / by zero
```

从打印中可以看出，onError()和onCompleted()是不共戴天滴。

除 Observer 接口之外，RxJava 还内置了一个实现 Observer 的抽象类:Subscriber(发[sə'skraɪbə])。 Subscriber 对 Observer 接口进行了一些扩展，但他们的基本使用方式是完全一样的:

RxHelloWorld.java

```
32              }
33          };
34
35          Subscriber<String> subscriber = new Subscriber<String>() {
36
37              @Override
38              public void onCompleted() {
39                  System.out.println("subscriber onCompleted()");
40              }
41
42              @Override
43              public void onError(Throwable e) {
44                  System.out.println("subscriber onError(): " + e.getMessage());
45              }
46
47              @Override
48              public void onNext(String t) {
49                  System.out.println("subscriber onNext():" + t);
50              }
51          };
52
53          // 创建被观察者Observable
54          Observable<String> observable = Observable
55                  .create(new OnSubscribe<String>() {
56
57                      @Override
58                      public void call(Subscriber<? super String> subscriber) {
59                          subscriber.onNext("Hello World!");
60                          int i = 1 / 0;
61                          subscriber.onCompleted();
62                      }
63                  });
64          // 被观察者订阅(subscribe)观察者
65          observable.subscribe(subscriber);
66
67      }
68
69  }
70
```

果真跟Observer的回调一模一样~

编译运行：

Console

```
<terminated> RxHelloWorld [Java Applicatior
subscriber onNext():Hello World!
subscriber onError(): / by zero
```

那既然观察者有两个：Observer 和 Subscriber，那这两者有啥区别呢？当然是有区别滴，不过在未来会进行学习，这里先有个印象。

BAT交流群：892872246

另外在 Rxjava 中，如果对于事件回调中，只想要 onNext()方法，onError()和 onCompleted()都不需要，有木有简化方案呢？当然有：

create() 方法是 RxJava 最基本的创造事件序列的方法。基于这个方法， RxJava 还提供了一些方法用来快捷创建事件队列，例如：

just(T...): 将传入的参数依次发送出来。

```
Observable observable = Observable.just("Hello", "Hi", "Aloha");
// 将会依次调用:
// onNext("Hello");
// onNext("Hi");
// onNext("Aloha");
// onCompleted();
```

上面 just(T...) 的例子和之前的 create(OnSubscribe) 的例子是等价的。

而之前我们创建 Observable 是通过它：

```
// 创建被观察者Observable
Observable<String> observable = Observable
        .create(new OnSubscribe<String>() {

            @Override
            public void call(Subscriber<? super String> subscriber) {
                subscriber.onNext("Hello World!");
                int i = 1 / 0;
                subscriber.onCompleted();
            }
        });
```

所以简化版采用 just 操作符来，如下：

BAT交流群：892872246

```
//更简单的代码：不关心onCompleted和onError，我们只需要在onNext的时候做一些处理，这时候就可以使用Action1类。
private static void testSimpleObservable() {
    Observable.just("Hello World!")
    .subscribe(new Observer<String>() {

        @Override
        public void onCompleted() {
            // TODO Auto-generated method stub

        }

        @Override
        public void onError(Throwable e) {
            // TODO Auto-generated method stub

        }

        @Override
        public void onNext(String t) {
            System.out.println("onNext():" + t);
        }
    });
}
```

另外因为Rxjava是一种链式调用，所以代码这块编写也改用更加简单的方式去写，而不像之前分成了几个步骤，更加体现其链式

编译运行：

```
Console
<terminated> RxHelloWorld [Jav
onNext():Hello World!
```

下面改用 Action1 类来达到我们所要的需求：

```
//更简单的代码：不关心onCompleted和onError，我们只需要在onNext的时候做一些处理，这时候就可以使用Action1类。
private static void testSimpleObservable() {
    Observable.just("Hello World!")
    .subscribe(new Action1<String>() {

        @Override
        public void call(String t) {
            System.out.println("value:" + t);
        }
    });
}
```

顺间回调减小，代码更加简洁了~

编译运行：

```
Console
<terminated> RxHelloWorld [Ja
value:Hello World!
```

BAT交流群892872246

但是，官方并不推荐使用该类来简化操作，因为 Action1 并不支持背压 (BackPressure)，而产生的大至原因是由于 Rxjava 的事件产生过快时导致。

6.3 事件变换设计

变换

RxJava 提供了对事件序列进行变换的支持，这是它的核心功能之一，也是大多数人说『RxJava 真是太好用了』的最大原因。**所谓变换，就是将事件序列中的对象或整个序列进行加工处理，转换成不同的事件或事件序列。**概念说着总是模糊难懂的，来看 API。

1) API

首先看一个 `map()` 的例子：

```
Observable.just("images/logo.png") // 输入类型 String
    .map(new Func1<String, Bitmap>() {
        @Override
        public Bitmap call(String filePath) { // 参数类型 String
            return getBitmapFromPath(filePath); // 返回类型 Bitmap
        }
    })
    .subscribe(new Action1<Bitmap>() {
        @Override
        public void call(Bitmap bitmap) { // 参数类型 Bitmap
            showBitmap(bitmap);
        }
```

```
    });
```

这里出现了一个叫做 Func1 的类。它和 Action1 非常相似，也是 RxJava 的一个接口，用于包装含有一个参数的方法。 Func1 和 Action 的区别在于， Func1 包装的是有返回值的方法。另外，和 ActionX 一样， FuncX 也有多个，用于不同参数个数的方法。FuncX 和 ActionX 的区别在 FuncX 包装的是有返回值的方法。

可以看到，map() 方法将参数中的 String 对象转换成一个 Bitmap 对象后返回，而在经过 map() 方法后，事件的参数类型也由 String 转为了 Bitmap。这种直接变换对象并返回的，是最常见的也最容易理解的变换。不过 RxJava 的变换远不止这样，它不仅可以针对事件对象，还可以针对整个事件队列，这使得 RxJava 变得非常灵活。我列举几个常用的变换：

- map()：事件对象的直接变换，具体功能上面已经介绍过。它是 RxJava 最常用的变换。 map() 的示意图：

map() 示意图

BAT交流

`flatMap()`：这是一个很有用但**非常难理解**的变换，因此我决定花多些篇幅来介绍它。 首先假设这么一种需求：假设有一个数据结构『学生』，现在需要打印出一组学生的名字。实现方式很简单：

```
Student[] students = ...;

Subscriber<String> subscriber = new Subscriber<String>() {

    @Override

    public void onNext(String name) {

        Log.d(tag, name);

    }

    ...};

Observable.from(students)

    .map(new Func1<Student, String>() {

        @Override

        public String call(Student student) {

            return student.getName();

        }

    })

    .subscribe(subscriber);
```

很简单。那么再假设：如果要打印出每个学生所需要修的所有课程的名称呢？（需求的区别在于，每个学生只有一个名字，但却有多个课程。）首先可以这样实现：

```
Student[] students = ...;

Subscriber<Student> subscriber = new Subscriber<Student>() {

    @Override
```

BA 2872246

```
    public void onNext(Student student) {

        List<Course> courses = student.getCourses();

        for (int i = 0; i < courses.size(); i++) {

            Course course = courses.get(i);

            Log.d(tag, course.getName());

        }

    }

    ...};

Observable.from(students)

    .subscribe(subscriber);
```

依然很简单。那么如果我不想在 `Subscriber` 中使用 for 循环，而是希望 `Subscriber` 中直接传入单个的 `Course` 对象呢（这对于代码复用很重要）？用 `map()` 显然是不行的，因为 `map()` 是一对一的转化，而我现在的要求是一对多的转化。那怎么才能把一个 Student 转化成多个 Course 呢？

这个时候，就需要用 `flatMap()` 了：

```
Student[] students = ...;

Subscriber<Course> subscriber = new Subscriber<Course>() {

    @Override

    public void onNext(Course course) {

        Log.d(tag, course.getName());

    }

    ...};

Observable.from(students)

    .flatMap(new Func1<Student, Observable<Course>>() {
```

```
        @Override
        public Observable<Course> call(Student student) {
            return Observable.from(student.getCourses());
        }
    })
    .subscribe(subscriber);
```

从上面的代码可以看出， flatMap() 和 map() 有一个相同点：它也是把传入的参数转化之后返回另一个对象。但需要注意，和 map()不同的是， flatMap() 中返回的是个 Observable 对象，并且这个 Observable 对象并不是被直接发送到了 Subscriber 的回调方法中。 flatMap() 的原理是这样的：1. 使用传入的事件对象创建一个 Observable 对象；2. 并不发送这个 Observable，而是将它激活，于是它开始发送事件；3. 每一个创建出来的 Observable 发送的事件，都被汇入同一个 Observable ，而这个 Observable 负责将这些事件统一交给 Subscriber 的回调方法。这三个步骤，把事件拆成了两级，通过一组新创建的 Observable 将初始的对象『铺平』之后通过统一路径分发了下去。而这个『铺平』就是 flatMap() 所谓的 flat。

flatMap() 示意图：

BAT交流群：892872246

扩展：由于可以在嵌套的 `Observable` 中添加异步代码，`flatMap()` 也常用于嵌套的异步操作，例如嵌套的网络请求。示例代码（Retrofit + RxJava）：

```
networkClient.token() // 返回 Observable<String>，在订阅时请求 token，并在响应后发送 token

    .flatMap(new Func1<String, Observable<Messages>>() {

        @Override

        public Observable<Messages> call(String token) {

            // 返回 Observable<Messages>，在订阅时请求消息列表，并在响应后发送请求到的消息列表

            return networkClient.messages();

        }
```

```
    })
    .subscribe(new Action1<Messages>() {
        @Override
        public void call(Messages messages) {
            // 处理显示消息列表
            showMessages(messages);
        }
    });
```

传统的嵌套请求需要使用嵌套的 Callback 来实现。而通过 `flatMap()` ，可以把嵌套的请求写在一条链中，从而保持程序逻辑的清晰。

- `throttleFirst()`: 在每次事件触发后的一定时间间隔内丢弃新的事件。常用作去抖动过滤，例如按钮的点击监听器：`RxView.clickEvents(button)` // RxBinding 代码，后面的文章有解释 `.throttleFirst(500, TimeUnit.MILLISECONDS)` // 设置防抖间隔为 500ms `.subscribe(subscriber)`;妈妈再也不怕我的用户手抖点开两个重复的界面啦。

此外， RxJava 还提供很多便捷的方法来实现事件序列的变换，这里就不一一举例了。

2) 变换的原理：lift()

这些变换虽然功能各有不同，但实质上都是**针对事件序列的处理和再发送**。而在 RxJava 的内部，它们是基于同一个基础的变换方法： `lift(Operator)`。首先看一下 `lift()` 的内部实现（仅核心代码）：

```
// 注意：这不是 lift() 的源码，而是将源码中与性能、兼容性、扩展性有关的代码剔除后的核心代码。// 如果需要看源码，可以去 RxJava 的 GitHub 仓库下载。public <R> Observable<R> lift(Operator<? extends R, ? super T> operator) {
    return Observable.create(new OnSubscribe<R>() {
        @Override
        public void call(Subscriber subscriber) {
```

```
        Subscriber newSubscriber = operator.call(subscriber);

        newSubscriber.onStart();

        onSubscribe.call(newSubscriber);

    }

});}
```

这段代码很有意思：它生成了一个新的 Observable 并返回，而且创建新 Observable 所用的参数 OnSubscribe 的回调方法 call() 中的实现竟然看起来和前面讲过的 Observable.subscribe() 一样！然而它们并不一样哟~不一样的地方关键就在于第二行 onSubscribe.call(subscriber) 中的 onSubscribe **所指代的对象不同**（高能预警：接下来的几句话可能会导致身体的严重不适）——

- subscribe() 中这句话的 onSubscribe 指的是 Observable 中的 onSubscribe 对象，这个没有问题，但是 lift() 之后的情况就复杂了点。
- 当含有 lift() 时：
  1.lift() 创建了一个 Observable 后，加上之前的原始 Observable，已经有两个 Observable 了；
  2.而同样地，新 Observable 里的新 OnSubscribe 加上之前的原始 Observable 中的原始 OnSubscribe，也就有了两个 OnSubscribe；
  3.当用户调用经过 lift() 后的 Observable 的 subscribe() 的时候，使用的是 lift() 所返回的新的 Observable，于是它所触发的 onSubscribe.call(subscriber)，也是用的新 Observable 中的新 OnSubscribe，即在 lift() 中生成的那个 OnSubscribe；
  4.而这个新 OnSubscribe 的 call() 方法中的 onSubscribe，就是指的原始 Observable 中的原始 OnSubscribe，在这个 call() 方法里，新 OnSubscribe 利用 operator.call(subscriber) 生成了一个新的 Subscriber（Operator 就是在这里，通过自己的 call() 方法将新 Subscriber 和原始 Subscriber 进行关联，并插入自己的『变换』代码以实现变换），然后利用这个新 Subscriber 向原始 Observable 进行订阅。
  这样就实现了 lift() 过程，有点**像一种代理机制，通过事件拦截和处理实现事件序列的变换。**

精简掉细节的话，也可以这么说：在 Observable 执行了 lift(Operator) 方法之后，会返回一个新的 Observable，这个新的 Observable 会像一个代理一样，负责接收原始的 Observable 发出的事件，并在处理后发送给 Subscriber。

如果你更喜欢具象思维，可以看图：

BAT交流群：892872246

lift() 原理图

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

或者可以看动图：

lift 原理动图

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

两次和多次的 `lift()` 同理，如下图：

两次 lift

举一个具体的 `Operator` 的实现。下面这是一个将事件中的 `Integer` 对象转换成 `String` 的例子，仅供参考：

```
observable.lift(new Observable.Operator<String, Integer>() {

    @Override

    public Subscriber<? super Integer> call(final Subscriber<? super Str
ing> subscriber) {

        // 将事件序列中的 Integer 对象转换为 String 对象
```

72246

```
        return new Subscriber<Integer>() {

            @Override

            public void onNext(Integer integer) {

                subscriber.onNext("" + integer);

            }



            @Override

            public void onCompleted() {

                subscriber.onCompleted();

            }



            @Override

            public void onError(Throwable e) {

                subscriber.onError(e);

            }

        };

    }});
```

讲述 lift() 的原理只是为了让你更好地了解 RxJava ，从而可以更好地使用它。然而不管你是否理解了 lift() 的原理，RxJava 都不建议开发者自定义 Operator 来直接使用 lift()，而是建议尽量使用已有的 lift() 包装方法（如 map() flatMap() 等）进行组合来实现需求，因为直接使用 lift() 非常容易发生一些难以发现的错误。

3) compose：对 Observable 整体的变换

BAT交流群：892872246

除了 lift() 之外， Observable 还有一个变换方法叫做 compose(Transformer)。它和 lift() 的区别在于， lift() **是针对事件项和事件序列的，而** compose() **是针对** Observable **自身进行变换。**举个例子，假设在程序中有多个 Observable ，并且他们都需要应用一组相同的 lift() 变换。你可以这么写：

```
observable1
    .lift1()
    .lift2()
    .lift3()
    .lift4()
    .subscribe(subscriber1);
observable2
    .lift1()
    .lift2()
    .lift3()
    .lift4()
    .subscribe(subscriber2);
observable3
    .lift1()
    .lift2()
    .lift3()
    .lift4()
    .subscribe(subscriber3);
observable4
    .lift1()
```

```
    .lift2()

    .lift3()

    .lift4()

    .subscribe(subscriber1);
```

你觉得这样太不软件工程了，于是你改成了这样：

```
private Observable liftAll(Observable observable) {

    return observable

        .lift1()

        .lift2()

        .lift3()

        .lift4();}...liftAll(observable1).subscribe(subscriber1);liftAll
(observable2).subscribe(subscriber2);liftAll(observable3).subscribe(sub
scriber3);liftAll(observable4).subscribe(subscriber4);
```

可读性、可维护性都提高了。可是 `Observable` 被一个方法包起来，这种方式对于 `Observale` 的灵活性似乎还是增添了那么点限制。怎么办？这个时候，就应该用 `compose()` 来解决了：

```
public class LiftAllTransformer implements Observable.Transformer<Intege
r, String> {

    @Override

    public Observable<String> call(Observable<Integer> observable) {

        return observable

            .lift1()

            .lift2()

            .lift3()
```

```
            .lift4();

    }}...Transformer liftAll = new LiftAllTransformer();

observable1.compose(liftAll).subscribe(subscriber1);

observable2.compose(liftAll).subscribe(subscriber2);

observable3.compose(liftAll).subscribe(subscriber3);

observable4.compose(liftAll).subscribe(subscriber4);
```

像上面这样，使用 `compose()` 方法，`Observable` 可以利用传入的 `Transformer` 对象的 `call` 方法直接对自身进行处理，也就不必被包在方法的里面了。像上面这样，使用 `compose()` 方法，`Observable` 可以利用传入的 `Transformer` 对象的 `call` 方法直接对自身进行处理，也就不必被包在方法的里面了。

6.4 Scheduler 线程控制

线程控制：Scheduler

除了灵活的变换，RxJava 另一个牛逼的地方，就是线程的自由控制。

1) Scheduler 的 API

前面讲到了，可以利用 `subscribeOn()` 结合 `observeOn()` 来实现线程控制，让事件的产生和消费发生在不同的线程。可是在了解了 `map()` `flatMap()` 等变换方法后，有些好事的（其实就是当初刚接触 RxJava 时的我）就问了：能不能多切换几次线程？

答案是：能。因为 `observeOn()` 指定的是 `Subscriber` 的线程，而这个 `Subscriber` 并不是（严格说应该为『不一定是』，但这里不妨理解为『不是』）`subscribe()` 参数中的 `Subscriber` ，而是 `observeOn()` 执行时的当前 `Observable` 所对应的 `Subscriber` ，即它的直接下级 `Subscriber` 。换句话说，`observeOn()` 指定的是它之后的操作所在的线程。因此如果有多次切换线程的需求，只要在每个想要切换线程的位置调用一次 `observeOn()` 即可。上代码：

```
Observable.just(1, 2, 3, 4) // IO 线程，由 subscribeOn() 指定
```

BAT交流群：892872246

```
    .subscribeOn(Schedulers.io())

    .observeOn(Schedulers.newThread())

    .map(mapOperator) // 新线程，由 observeOn() 指定

    .observeOn(Schedulers.io())

    .map(mapOperator2) // IO 线程，由 observeOn() 指定

    .observeOn(AndroidSchedulers.mainThread)

    .subscribe(subscriber);  // Android 主线程，由 observeOn() 指定
```

如上，通过 `observeOn()` 的多次调用，程序实现了线程的多次切换。

不过，不同于 `observeOn()` ， `subscribeOn()` 的位置放在哪里都可以，但它是只能调用一次的。

又有好事的（其实还是当初的我）问了：如果我非要调用多次 `subscribeOn()` 呢？会有什么效果？

这个问题先放着，我们还是从 RxJava 线程控制的原理说起吧。

2) Scheduler 的原理

其实， `subscribeOn()` 和 `observeOn()` 的内部实现，也是用的 `lift()`。具体看图（不同颜色的箭头表示不同的线程）：

`subscribeOn()` 原理图：

BAT交流群：892872246

subscribeOn() 原理

BAT交流群：892872246

`observeOn()` 原理图：

observeOn() 原理

从图中可以看出，`subscribeOn()` 和 `observeOn()` 都做了线程切换的工作（图中的 "schedule..." 部位）。不同的是， `subscribeOn()`的线程切换发生在 `OnSubscribe` 中，即在它通知上一级 `OnSubscribe` 时，这时事件还没有开始发送，因此 `subscribeOn()` 的线程控制可以从事件发出的开端就造成影响；而 `observeOn()` 的线程切换则发生在它内建的 `Subscriber` 中，即发生在它即将给下一级 `Subscriber` 发送事件时，因此 `observeOn()` 控制的是它后面的线程。

最后，我用一张图来解释当多个 `subscribeOn()` 和 `observeOn()` 混合使用时，线程调度是怎么发生的（由于图中对象较多，相对于上面的图对结构做了一些简化调整）：

BAT交流群：892872246

线程控制综合调用

图中共有 5 处含有对事件的操作。由图中可以看出，①和②两处受第一个 `subscribeOn()` 影响，运行在红色线程；③和④处受第一个 `observeOn()` 的影响，运行在绿色线程；⑤处受第二个 `onserveOn()` 影响，运行在紫色线程；而第二个 `subscribeOn()` ，由于在通知过程中线程就被第一个 `subscribeOn()` 截断，因此对整个流程并没有任何影响。这里也就回答了前面的问题：当使用了多个 `subscribeOn()` 的时候，只有第一个 `subscribeOn()` 起作用。

3) 延伸：doOnSubscribe()

然而，虽然超过一个的 `subscribeOn()` 对事件处理的流程没有影响，但在流程之前却是可以利用的。

在前面讲 `Subscriber` 的时候，提到过 `Subscriber` 的 `onStart()` 可以用作流程开始前的初始化。然而 `onStart()` 由于在 `subscribe()` 发生时就被调用了，

BAT交流群：892872246

因此不能指定线程，而是只能执行在 `subscribe()` 被调用时的线程。这就导致如果 `onStart()` 中含有对线程有要求的代码（例如在界面上显示一个 ProgressBar，这必须在主线程执行），将会有线程非法的风险，因为有时你无法预测 `subscribe()` 将会在什么线程执行。

而与 `Subscriber.onStart()` 相对应的，有一个方法 `Observable.doOnSubscribe()` 。它和 `Subscriber.onStart()` 同样是在 `subscribe()` 调用后而且在事件发送前执行，但区别在于它可以指定线程。默认情况下， `doOnSubscribe()` 执行在 `subscribe()` 发生的线程；而如果在 `doOnSubscribe()` 之后有 `subscribeOn()` 的话，它将执行在离它最近的 `subscribeOn()` 所指定的线程。

示例代码：

```
Observable.create(onSubscribe)
    .subscribeOn(Schedulers.io())
    .doOnSubscribe(new Action0() {
        @Override
        public void call() {
            progressBar.setVisibility(View.VISIBLE); // 需要在主线程执行
        }
    })
    .subscribeOn(AndroidSchedulers.mainThread()) // 指定主线程
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(subscriber);
```

如上，在 `doOnSubscribe()`的后面跟一个 `subscribeOn()` ，就能指定准备工作的线程了。

第七章、IOC 架构设计

7.1 依赖注入与控制反转

BAT交流群：892872246

1、IoC 是什么

Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。在 Java 开发中，Ioc 意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。如何理解好 Ioc 呢？理解好 Ioc 的关键是要明确“谁控制谁，控制什么，为何是反转（有反转就应该有正转了），哪些方面反转了”，那我们来深入分析一下：

●**谁控制谁，控制什么：**传统 Java SE 程序设计，我们直接在对象内部通过 new 进行创建对象，是程序主动去创建依赖对象；而 IoC 是有专门一个容器来创建这些对象，即由 Ioc 容器来控制对象的创建；谁控制谁？当然是 IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。

●**为何是反转，哪些方面反转了：**有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。

用图例说明一下，传统程序设计如图 2-1，都是主动去创建相关对象然后再组合起来：

传统应用程序示意图

BAT交流群：892872246

当有了IoC/DI的容器后，在客户端类中不再主动去创建这些对象了，如图2-2所示:

有IoC/DI容器后程序结构示意图

2、IoC 能做什么

IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了 IoC 容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

其实 IoC 对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在 IoC/DI 思想中，应用程序就变成被动的了，被动的等待 IoC 容器来创建并注入它所需要的资源了。

IoC 很好的体现了面向对象设计法则之一—— 好莱坞法则：“别找我们，我们找你”；即由 IoC 容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。

3、IoC 和 DI

BAT交流群：892872246

DI—Dependency Injection，即“依赖注入”：是组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

理解 DI 的关键是：“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”，那我们来深入分析一下：

●**谁依赖于谁：**当然是应用程序依赖于 IoC 容器；

●**为什么需要依赖：**应用程序需要 IoC 容器来提供对象需要的外部资源；

●**谁注入谁：**很明显是 IoC 容器注入应用程序某个对象，应用程序依赖的对象；

●**注入了什么：**就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）。

IoC 和 DI 由什么关系呢？其实它们是同一个概念的不同角度描述，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以 2004 年大师级人物 Martin Fowler 又给出了一个新的名字：“依赖注入”，相对 IoC 而言，**“****依赖注入”****明确描述了“被注入对象依赖 IoC****容器配置依赖对象”。**

7.2 ButterKnife 原理上篇、中篇、下篇

一、代码结构

BAT交流群：892872246

BAT交流群：892872246

我们这里对 ButterKnife 的最新版本 8.4.0 进行分析。
我们先 down 下来看下代码的结构，可以看到代码结构分的还是很好的。

- butterknife ;android library model 提供 android 使用的 API
- butterknife-annotations; java－model，使用时的注解
- butterknife-compiler；java－model，编译时用到的注解的处理器
- butterknife-gradle-plugin；自定义的 gradle 插件，辅助生成有关代码
- butterknife-integration-test；该项目的测试用例
- butterknife-lint；该项目的 lint 检查
- sample；demo
  这里重点分析 butterknife-compiler 及 butterknife

二、原理图

建议看完全篇了再返回看这个图，会更明白

BAT交流群：892872246

三　使用方法

这里不再给出。不过会在原代码分析的时候给出一些注意的地方。
我们拿官方的 demo－SimpleActivity
编译完后最后生成的文件为：SimpleActivity_ViewBinding
路径在：

BAT交流群：892872246

内容：

```
// Generated code from Butter Knife. Do not modify!package com.example.b
utterknife.library;

import android.support.annotation.CallSuper;import android.support.annot
ation.UiThread;import android.view.View;import android.widget.AdapterVie
w;import android.widget.Button;import android.widget.ListView;import and
roid.widget.TextView;import butterknife.Unbinder;import butterknife.inte
rnal.DebouncingOnClickListener;import butterknife.internal.Utils;import
java.lang.IllegalStateException;import java.lang.Override;

public class SimpleActivity_ViewBinding<T extends SimpleActivity> implem
ents Unbinder {

  protected T target;



  private View view2130968578;



  private View view2130968579;



  @UiThread

  public SimpleActivity_ViewBinding(final T target, View source) {

    this.target = target;
```

```
    View view;

    target.title = Utils.findRequiredViewAsType(source, R.id.title, "fie
ld 'title'", TextView.class);

    target.subtitle = Utils.findRequiredViewAsType(source, R.id.subtitle,
 "field 'subtitle'", TextView.class);

    view = Utils.findRequiredView(source, R.id.hello, "field 'hello', me
thod 'sayHello', and method 'sayGetOffMe'");

    target.hello = Utils.castView(view, R.id.hello, "field 'hello'", But
ton.class);

    view2130968578 = view;

    view.setOnClickListener(new DebouncingOnClickListener() {

      @Override

      public void doClick(View p0) {

        target.sayHello();

      }

    });

    view.setOnLongClickListener(new View.OnLongClickListener() {

      @Override

      public boolean onLongClick(View p0) {

        return target.sayGetOffMe();

      }

    });

    view = Utils.findRequiredView(source, R.id.list_of_things, "field 'l
istOfThings' and method 'onItemClick'");
```

```
    target.listOfThings = Utils.castView(view, R.id.list_of_things, "fie
ld 'listOfThings'", ListView.class);

    view2130968579 = view;

    ((AdapterView<?>) view).setOnItemClickListener(new AdapterView.OnIte
mClickListener() {

      @Override

      public void onItemClick(AdapterView<?> p0, View p1, int p2, long p3)
 {

        target.onItemClick(p2);

      }

    });

    target.footer = Utils.findRequiredViewAsType(source, R.id.footer, "f
ield 'footer'", TextView.class);

    target.headerViews = Utils.listOf(

        Utils.findRequiredView(source, R.id.title, "field 'headerViews'
"),

        Utils.findRequiredView(source, R.id.subtitle, "field 'headerView
s'"),

        Utils.findRequiredView(source, R.id.hello, "field 'headerViews'
"));

  }



  @Override

  @CallSuper

  public void unbind() {

    T target = this.target;
```

```
    if (target == null) throw new IllegalStateException("Bindings alread
y cleared.");


    target.title = null;

    target.subtitle = null;

    target.hello = null;

    target.listOfThings = null;

    target.footer = null;

    target.headerViews = null;


    view2130968578.setOnClickListener(null);

    view2130968578.setOnLongClickListener(null);

    view2130968578 = null;

    ((AdapterView<?>) view2130968579).setOnItemClickListener(null);

    view2130968579 = null;


    this.target = null;

  }}
```

我们看到了我们熟悉的代码，虽然比较乱（因为是生成的），
可以看出 在构造中 findview 在 unbind 中进行置 null 处理，让告诉 gc 在合适的机会回收占用的内存 ；但是这是后面真正生成代码我们看不到的，没关系 嘻嘻。

3.1 整体的原理一（编译时期一注解处理器）

在 java 代码的编译时期，javac 会调用 java 注解处理器来进行处理。因此我们可以定义自己的注解处理器来干一些事情。一个特定注解的处理器以 java 源代码（或者已编译的字节码）作为输入，然后生成一些文件（通常是.java 文件）作

BAT交流群：892872246

为输出。因此我们可以在用户已有的代码上添加一些方法，来帮我们做一些有用的事情。这些生成的 java 文件跟其他手动编写的 java 源代码一样，将会被 javac 编译。（个人参考及个人理解）

3.2 定义处理器 继承 AbstractProcessor

在 java 中定义自己的处理器都是继承自 AbstractProcessor
前 3 个方法都试固定写法，主要是 process 方法。

```
public class MyProcessor extends AbstractProcessor {



    //用来指定你使用的 java 版本。通常你应该返回
 SourceVersion.latestSupported()



    @Override

    public SourceVersion getSupportedSourceVersion() {

        return SourceVersion.latestSupported();

    }



    //会被处理器调用，可以在这里获取 Filer，Elements，Messager 等辅助类，后面
会解释

    @Override

    public synchronized void init(ProcessingEnvironment processingEnv) {

        super.init(processingEnv);

    }



//这个方法返回 stirng 类型的 set 集合，集合里包含了你需要处理的注解

    @Override
```

```
    public Set<String> getSupportedAnnotationTypes() {

        Set<String> annotataions = new LinkedHashSet<String>();

        annotataions.add("com.example.MyAnnotation");

        return annotataions;

    }




   //核心方法，这个一般的流程就是先扫描查找注解，再生成 java 文件

   //这 2 个步骤设计的知识点细节很多。


    @Override

    public boolean process(Set<? extends TypeElement> annoations,

            RoundEnvironment env) {

        return false;

    }}
```

3.3 注册你的处理器

要像 jvm 调用你写的处理器，你必须先注册，让他知道。怎么让它知道呢，其实很简单，google 为我们提供了一个库，简单的一个注解就可以。

首先是依赖

```
compile 'com.google.auto.service:auto-service:1.0-rc2'

@AutoService(Processor.class)

public class BindViewProcessor extends AbstractProcessor {
```

```
  //...省略非关键代码

}
```

3.4 基本概念

Elements：一个用来处理 Element 的工具类
Types：一个用来处理 TypeMirror 的工具类
Filer：你可以使用这个类来创建.java 文件

四、源码分析

分析之前呢先要有写基本的概念

可以看到 AutoService 注解

```
@AutoService(Processor.class)public final class ButterKnifeProcessor ext
ends AbstractProcessor {//...

}
```

(1) init 方法，这个主要是获取一些辅助类

```
private Filer mFiler; //文件相关的辅助类

   private Elements mElementUtils; //元素相关的辅助类

   private Messager mMessager; //日志相关的辅助类



 @Override public synchronized void init(ProcessingEnvironment env) {

   super.init(env);



   elementUtils = env.getElementUtils();

   typeUtils = env.getTypeUtils();

   filer = env.getFiler();
```

```
  try {

    trees = Trees.instance(processingEnv);

  } catch (IllegalArgumentException ignored) {

  }

}
```

(2) getSupportedSourceVersion()方法就是默认的，获取你该处理器使用的 java 版本

```
@Override public SourceVersion getSupportedSourceVersion() {

   return SourceVersion.latestSupported();

 }
```

(3) 接下来就是 process 方法，这个方法是核心方法。

```
 @Override public boolean process(Set<? extends TypeElement> elements, R
oundEnvironment env) {

 //1.查找所有的注解信息，并形成 BindingClass（是什么 后面会讲） 保存到 map 中

    Map<TypeElement, BindingClass> targetClassMap = findAndParseTargets
(env);

//2.遍历步骤 1 的 map 的生成.java 文件也就是上文的  类名_ViewBinding  的 java 文
件



  for (Map.Entry<TypeElement, BindingClass> entry : targetClassMap.entry
Set()) {

      TypeElement typeElement = entry.getKey();

      BindingClass bindingClass = entry.getValue();
```

```
      JavaFile javaFile = bindingClass.brewJava();

      try {

        javaFile.writeTo(filer);

      } catch (IOException e) {

        error(typeElement, "Unable to write binding for type %s: %s", typ
eElement, e.getMessage());

      }

    }



    return true;

  }
```

显然分两步执行，下面再仔细走一下该方法流程。

每个注解的查找与解析-findAndParseTargets

```
  private Map<TypeElement, BindingClass> findAndParseTargets(RoundEnviro
nment env) {

    Map<TypeElement, BindingClass> targetClassMap = new LinkedHashMap<>
();

    Set<TypeElement> erasedTargetNames = new LinkedHashSet<>();



    scanForRClasses(env);

    //...下面是每个注解的解析

 // Process each @BindArray element.

    for (Element element : env.getElementsAnnotatedWith(BindArray.class))
 {
```

```
    if (!SuperficialValidation.validateElement(element)) continue;

    try {

      parseResourceArray(element, targetClassMap, erasedTargetNames);

    } catch (Exception e) {

      logParsingError(element, BindArray.class, e);

    }

  }



   // Process each @BindView element.

  for (Element element : env.getElementsAnnotatedWith(BindView.class))
{

    // we don't SuperficialValidation.validateElement(element)

    // so that an unresolved View type can be generated by later proces
sing rounds

    try {

      parseBindView(element, targetClassMap, erasedTargetNames);

    } catch (Exception e) {

      logParsingError(element, BindView.class, e);

    }

  }

 //....




    // Process each annotation that corresponds to a listener.
```

```
    for (Class<? extends Annotation> listener : LISTENERS) {

      findAndParseListener(env, listener, targetClassMap, erasedTargetNa
mes);

    }



    // Try to find a parent binder for each.

    for (Map.Entry<TypeElement, BindingClass> entry : targetClassMap.ent
rySet()) {

      TypeElement parentType = findParentType(entry.getKey(), erasedTarg
etNames);

      if (parentType != null) {

        BindingClass bindingClass = entry.getValue();

        BindingClass parentBindingClass = targetClassMap.get(parentTyp
e);

        bindingClass.setParent(parentBindingClass);

      }

    }



    return targetClassMap;

}
```

第一步

我们先看一下参数 RoundEnvironment 这个是什么呢？个人理解是注解框架里的一个工具什么工具呢？一个可以在处理器处理该处理器 用来查询注解信息的工具，当然包含你在 getSupportedAnnotationTypes 注册的注解。

第二步

创建了一个 LinkedHashMap 保证了先后的顺序就是先注解的先生成 java 文件，其实也没有什么先后无所谓。最后将其返回。

我们这里只分析一个具有代表性的 BindView 注解，其它的都是一样的，连代码

BAT交流群：892872246

都一毛一样。
在这之前先看一下 Element 这个类我们看一下官方注释

```
 * Represents a program element such as a package, class, or method.

 * Each element represents a static, language-level construct

 * (and not, for example, a runtime construct of the virtual machine).
```

在注解处理器中，我们扫描 java 源文件，源代码中的每一部分都是 Element 的一个特定类型。换句话说：Element 代表程序中的元素，比如说 包，类，方法。每一个元素代表一个静态的，语言级别的结构.
比如：

```
public class ClassA { // TypeElement

    private int var_0; // VariableElement

    public ClassA() {} // ExecuteableElement



    public void setA( // ExecuteableElement

            int newA // TypeElement

    ) {

    }}
```

可以看到类为 TypeElement，变量为 VariableElement，方法为

ExecuteableElement

这些都是 Element 的子类，核对源码。

```
TypeElement aClass ; for (Element e : aClass.getEnclosedElements()){ //获取所有的子节点

    Element parent = e.getEnclosingElement();  // 获取父节点

    }
```

Elements 代表源代码，TypeElement 代表源代码中的元素类型，例如类。然后，TypeElement 并不包含类的相关信息。你可以从 TypeElement 获取类的名称，但你不能获取类的信息，比如说父类。这些信息可以通过 TypeMirror 获取。你可以通过调用 element.asType()来获取一个 Element 的 TypeMirror。

这个是对于理解源码的基础。
继续，我们看到了 for 循环查找所有包含 BindView 的注解。

```
parseBindView(element, targetClassMap, erasedTargetNames);
```

把 element 和 targetClassMap 传入

```
  private void parseBindView(Element element, Map<TypeElement, BindingCl
ass> targetClassMap,

      Set<TypeElement> erasedTargetNames) {

    TypeElement enclosingElement = (TypeElement) element.getEnclosingEle
ment();



     //1.检查用户使用的合法性

    // Start by verifying common generated code restrictions.

    boolean hasError = isInaccessibleViaGeneratedCode(BindView.class, "f
ields", element)

        || isBindingInWrongPackage(BindView.class, element);



    // Verify that the target type extends from View.

    TypeMirror elementType = element.asType();

    if (elementType.getKind() == TypeKind.TYPEVAR) {

      TypeVariable typeVariable = (TypeVariable) elementType;

      elementType = typeVariable.getUpperBound();

    }

    if (!isSubtypeOfType(elementType, VIEW_TYPE) && !isInterface(element
Type)) {

      if (elementType.getKind() == TypeKind.ERROR) {
```

```
      note(element, "@%s field with unresolved type (%s) "

              + "must elsewhere be generated as a View or interface. (%
s.%s)",

          BindView.class.getSimpleName(), elementType, enclosingElement.
getQualifiedName(),

          element.getSimpleName());

      } else {

        error(element, "@%s fields must extend from View or be an interfa
ce. (%s.%s)",

          BindView.class.getSimpleName(), enclosingElement.getQualified
Name(),

          element.getSimpleName());

        hasError = true;

      }

    }



    // 不合法的直接返回

    if (hasError) {

      return;

    }




    //2.获取 id 值



    // Assemble information on the field.
```

```
    int id = element.getAnnotation(BindView.class).value();



    // 3.获取 BindingClass，有缓存机制，没有则创建，下文会仔细分析



    BindingClass bindingClass = targetClassMap.get(enclosingElement);

    if (bindingClass != null) {

      ViewBindings viewBindings = bindingClass.getViewBinding(getId(i
d));

      if (viewBindings != null && viewBindings.getFieldBinding() != null)
 {

        FieldViewBinding existingBinding = viewBindings.getFieldBinding
();

        error(element, "Attempt to use @%s for an already bound ID %d on
'%s'. (%s.%s)",

            BindView.class.getSimpleName(), id, existingBinding.getName
(),

            enclosingElement.getQualifiedName(), element.getSimpleName
());

        return;

      }

    } else {

      bindingClass = getOrCreateTargetClass(targetClassMap, enclosingEle
ment);

    }

   //4 生成 FieldViewBinding 实体
```

```
    String name = element.getSimpleName().toString();

    TypeName type = TypeName.get(elementType);

    boolean required = isFieldRequired(element);



    FieldViewBinding binding = new FieldViewBinding(name, type, require
d);

     //5。加入到 bindingClass 成员变量的集合中

    bindingClass.addField(getId(id), binding);



    // Add the type-erased version to the valid binding        targets s
et.

    erasedTargetNames.add(enclosingElement);



  }
```

element.getEnclosingElement();是什么呢？是父节点。就是上面我们说的。

基本的步骤就是上面的 5 步

1.检查用户使用的合法

```
   // Start by verifying common generated code restrictions.

    boolean hasError = isInaccessibleViaGeneratedCode(BindView.class, "f
ields", element)

        || isBindingInWrongPackage(BindView.class, element);

 private boolean isInaccessibleViaGeneratedCode(Class<? extends Annotati
on> annotationClass,

      String targetThing, Element element) {
```

```
    boolean hasError = false;

    // 得到父节点

    TypeElement enclosingElement = (TypeElement) element.getEnclosingEle
ment();



 //判断修饰符，如果包含 private or static 就会抛出异常。

    // Verify method modifiers.

    Set<Modifier> modifiers = element.getModifiers();

    if (modifiers.contains(PRIVATE) || modifiers.contains(STATIC)) {

      error(element, "@%s %s must not be private or static. (%s.%s)",

          annotationClass.getSimpleName(), targetThing, enclosingElement.
getQualifiedName(),

          element.getSimpleName());

      hasError = true;

    }



        //判断父节点是否是类类型的，不是的话就会抛出异常

        //也就是说 BindView 的使用必须在一个类里

    // Verify containing type.

    if (enclosingElement.getKind() != CLASS) {

      error(enclosingElement, "@%s %s may only be contained in classes.
(%s.%s)",

          annotationClass.getSimpleName(), targetThing, enclosingElement.
getQualifiedName(),

          element.getSimpleName());
```

```
      hasError = true;

    }



     //判断父节点如果是 private 类，则抛出异常

    // Verify containing class visibility is not private.

    if (enclosingElement.getModifiers().contains(PRIVATE)) {

      error(enclosingElement, "@%s %s may not be contained in private cla
sses. (%s.%s)",

          annotationClass.getSimpleName(), targetThing, enclosingElement.
getQualifiedName(),

          element.getSimpleName());

      hasError = true;

    }



    return hasError;

  }
```

上面的代码里遇见注释了，这里说一下也就是我们在使用 bindview 注解的时候不能用使用

```
类不能是 private 修饰 ， 可以是默认的或者 public

  //in adapter

   private  static final class ViewHolder {

   //....

   }

   //成员变量不能是 private 修饰 ， 可以是默认的或者 public
```

```
  @BindView(R.id.word)

 private  TextView word;
```

接下来还有一个方法 isBindingInWrongPackage。
这个看名字也才出来个大概 就是不能在 android ，java 这种源码的 sdk 中使用。
如果你的包名是以 android 或者 java 开头就会抛出异常

```
  private boolean isBindingInWrongPackage(Class<? extends Annotation> a
nnotationClass,

     Element element) {

     //得到父节点

   TypeElement enclosingElement = (TypeElement) element.getEnclosingEle
ment();

   String qualifiedName = enclosingElement.getQualifiedName().toString
();



   if (qualifiedName.startsWith("android.")) {

     error(element, "@%s-annotated class incorrectly in Android framewor
k package. (%s)",

         annotationClass.getSimpleName(), qualifiedName);

     return true;

   }

   if (qualifiedName.startsWith("java.")) {

     error(element, "@%s-annotated class incorrectly in Java framework p
ackage. (%s)",

         annotationClass.getSimpleName(), qualifiedName);

     return true;

   }
```

```
    return false;

  }
```

到这里合法性检查就完了，如果你使用不当，就会抛出异常。

```
// 。。。 异常抛出了

// 不合法的直接返回

    if (hasError) {

      return;

    }
```

这里说明一点，在处理器中抛出异常，你不能直接像平常写 java 代码一样 new thow xxx 一样，这样抛出去的异常不太好看。所以 java 处理器帮我们提供了一个辅助类 Messager,这个可以帮助我们

```
     /**

    * Returns the messager used to report errors, warnings, and other

    * notices.

    *

    * @return the messager

    */

    Messager getMessager();
```

这里说明一点，在处理器中抛出异常，你不能直接像平常写 java 代码一样 new thow xxx 一样，这样抛出去的异常不太好看。所以 java 处理器帮我们提供了一个辅助类 Messager,这个可以帮助我

```
     /**

    * Returns the messager used to report errors, warnings, and other
```

BA 群：892872246

```
     * notices.
     *
     * @return the messager
     */
    Messager getMessager();
```

比如检查使用合法性抛出的异常信息－error 方法最后都会调用

```
  private void error(Element element, String message, Object... args) {
  //Kind.ERROR 级别，就像你使用 android 的 log 一样
    printMessage(Kind.ERROR, element, message, args);
  }

  private void note(Element element, String message, Object... args) {
    printMessage(Kind.NOTE, element, message, args);
  }

  private void printMessage(Kind kind, Element element, String message,
Object[] args) {
    if (args.length > 0) {
      message = String.format(message, args);
    }

    processingEnv.getMessager().printMessage(kind, message, element);}
```

IOC 架构设计之 ButterKnife 源码&原理（二）中篇

BAT交流群892246

第二步、parseBindView 的 获取值

```
BindingClass bindingClass = targetClassMap.get(enclosingElement);

   if (bindingClass != null) {

     ViewBindings viewBindings = bindingClass.getViewBinding(getId(i
d));

     if (viewBindings != null && viewBindings.getFieldBinding() != null)
 {

       FieldViewBinding existingBinding = viewBindings.getFieldBinding
();

       error(element, "Attempt to use @%s for an already bound ID %d on
'%s'. (%s.%s)",

           BindView.class.getSimpleName(), id, existingBinding.getName
(),

           enclosingElement.getQualifiedName(), element.getSimpleName
());

       return;

     }

   } else {

     bindingClass = getOrCreateTargetClass(targetClassMap, enclosingEle
ment);

   }
```

如果 map（缓存）中已经有了就直接到到这个 BindingClass 实例。BindingClass 这个我们后面还会再说。

1. 如果不为空，则根据 id 获取保存在 bindingClass 中的 ViewBindings 实例，如果 viewBindings 不为空且 viewBindings.getFieldBinding 不为空则抛出异常，什么意思呢？就是说一个 id 不能 bind 多次。
2.如果为空，则 bindingClass = getOrCreateTargetClass(targetClassMap, enclosingElement);
获取并返回，参数是最初的那个 map 和父节点。

BAT交流群：892872246

```
 private BindingClass getOrCreateTargetClass(Map<TypeElement, BindingCla
ss> targetClassMap,

     TypeElement enclosingElement) {

    BindingClass bindingClass = targetClassMap.get(enclosingElement);

    //再次判断

    if (bindingClass == null) {

     //获取父节点的类型名字，这个不用太关系

     TypeName targetType = TypeName.get(enclosingElement.asType());

     if (targetType instanceof ParameterizedTypeName) {

       targetType = ((ParameterizedTypeName) targetType).rawType;

     }



 //获取该 enclosingElement 就是父节点所在的包名称



     String packageName = getPackageName(enclosingElement);

    //类名字

     String className = getClassName(enclosingElement, packageName);

     //根据包名称和类名称获取 bindingClassName 实体

     //并且加入了_ViewBinding 哈哈，有点意思是了。不是吗？？

     ClassName bindingClassName = ClassName.get(packageName, className
+ "_ViewBinding");



     //是否是 final 类
```

```
      boolean isFinal = enclosingElement.getModifiers().contains(Modifie
r.FINAL);



      //创建了一个 BindingClass 实例

      bindingClass = new BindingClass(targetType, bindingClassName, isFi
nal);

      //加入集合，缓存

      targetClassMap.put(enclosingElement, bindingClass);

    }

    return bindingClass;

  }

第三步、parseBindView//@BindView(R.id.word)

    // TextView word;

    //name 就是 word

    String name = element.getSimpleName().toString();

     //类型的名字

    TypeName type = TypeName.get(elementType);

    //是否要求可为空

    boolean required = isFieldRequired(element);

     //生成 FieldViewBinding 实体

    FieldViewBinding binding = new FieldViewBinding(name, type, require
d);

//@BindView(R.id.word)

    // TextView word;
```

```
    //name 就是 word

    String name = element.getSimpleName().toString();

     //类型的名字

    TypeName type = TypeName.get(elementType);

    //是否要求可为空

    boolean required = isFieldRequired(element);

     //生成 FieldViewBinding 实体

    FieldViewBinding binding = new FieldViewBinding(name, type, require
d);
```

第三步、parseBindView

```
//@BindView(R.id.word)

    // TextView word;

    //name 就是 word

    String name = element.getSimpleName().toString();

     //类型的名字

    TypeName type = TypeName.get(elementType);

    //是否要求可为空

    boolean required = isFieldRequired(element);

     //生成 FieldViewBinding 实体

    FieldViewBinding binding = new FieldViewBinding(name, type, require
d);
```

第四步

```
添加到 bindingClass 中的成员变量的实体的集合中，方便生成 java 源文件也就是 xxxxx
_ViewBinding 文件的成员变量的初始化存在
```

```
bindingClass.addField(getId(id), binding);
```

其它的注解都是一样的。至此查找并解析成员变量的流程就完了。
接下来是处理控件事件的监听的流程。

注解事件源码流程分析（OnClick，OnItemClick 等）
我们回到 findAndParseTargets 方法

```
  //... 省略成员变量的注解



 // Process each annotation that corresponds to a listener.



   //处理方法的比如一些 OnClick，OnItemClick 等

   for (Class<? extends Annotation> listener : LISTENERS) {

     findAndParseListener(env, listener, targetClassMap, erasedTargetNa
mes);

   }



   // Try to find a parent binder for each.

   for (Map.Entry<TypeElement, BindingClass> entry : targetClassMap.ent
rySet()) {

     TypeElement parentType = findParentType(entry.getKey(), erasedTarg
etNames);

     if (parentType != null) {

       BindingClass bindingClass = entry.getValue();

       BindingClass parentBindingClass = targetClassMap.get(parentTyp
e);

       bindingClass.setParent(parentBindingClass);

     }
```

```
    }


    return targetClassMap;

   }
```

处理注解事件同样也分为查找和解析 2 个大步骤。

LISTENERS 是 butterknife 支持的注解集合

```
private static final List<Class<? extends Annotation>> LISTENERS = Arrays.asList(//

     OnCheckedChanged.class, //

     OnClick.class, //

     OnEditorAction.class, //

     OnFocusChange.class, //

     OnItemClick.class, //

     OnItemLongClick.class, //

     OnItemSelected.class, //

     OnLongClick.class, //

     OnPageChange.class, //

     OnTextChanged.class, //

     OnTouch.class //

  );

  private static final List<Class<? extends Annotation>> LISTENERS = Arrays.asList(//

     OnCheckedChanged.class, //

     OnClick.class, //
```

```
    OnEditorAction.class, //

    OnFocusChange.class, //

    OnItemClick.class, //

    OnItemLongClick.class, //

    OnItemSelected.class, //

    OnLongClick.class, //

    OnPageChange.class, //

    OnTextChanged.class, //

    OnTouch.class //

 );

private void findAndParseListener(RoundEnvironment env,

    Class<? extends Annotation> annotationClass, Map<TypeElement, Bind
ingClass> targetClassMap,

    Set<TypeElement> erasedTargetNames) {

   for (Element element : env.getElementsAnnotatedWith(annotationClass))
{

    //检查合法性问题

    if (!SuperficialValidation.validateElement(element)) continue;

    try {

       //解析注解

      parseListenerAnnotation(annotationClass, element, targetClassMap,
erasedTargetNames);

    } catch (Exception e) {

      StringWriter stackTrace = new StringWriter();
```

```
        e.printStackTrace(new PrintWriter(stackTrace));



        error(element, "Unable to generate view binder for @%s.\n\n%s",

            annotationClass.getSimpleName(), stackTrace.toString());

      }

    }

  }
```

我们看一下 parseListenerAnnotation 方法，传入了注解类 annotationClass，该节点 element，最初的那个集合 targetClassMap。 比较长，我们在方法里注释效果会比较好。

```
private void parseListenerAnnotation(Class<? extends Annotation> annotat
ionClass, Element element,

      Map<TypeElement, BindingClass> targetClassMap, Set<TypeElement> er
asedTargetNames)

      throws Exception {

    // This should be guarded by the annotation's @Target but it's worth
a check for safe casting.

    //必需是方法类型的，节点元素为 ExecutableElement

    if (!(element instanceof ExecutableElement) || element.getKind() !=
METHOD) {

      throw new IllegalStateException(

          String.format("@%s annotation must be on a method.", annotation
Class.getSimpleName()));

    }

//方法对应的是 ExecutableElement，前文我们已经简单的说明了一下

    ExecutableElement executableElement = (ExecutableElement) element;

    //获取父节点
```

```
    TypeElement enclosingElement = (TypeElement) element.getEnclosingEle
ment();



    // Assemble information on the method.

    // 获取注解信息

    Annotation annotation = element.getAnnotation(annotationClass);

    //该注解 value 方法，每一个注解都有（butterknife 提供的都是数组）

    //为什么是数组？因为支持下面这种

      @OnClick( {R.id.hello, R.id.hello2} )

      void sayHello() {

      }

      //反射注解方法 value

    Method annotationValue = annotationClass.getDeclaredMethod("value");

    //不是数组抛出异常

    if (annotationValue.getReturnType() != int[].class) {

      throw new IllegalStateException(

          String.format("@%s annotation value() type not int[].", annotat
ionClass));

    }



    //反射调用

    int[] ids = (int[]) annotationValue.invoke(annotation);

    //方法名字

    String name = executableElement.getSimpleName().toString();
```

```
    boolean required = isListenerRequired(executableElement);



    // Verify that the method and its containing class are accessible via
 generated code.

    //检查方法的修饰符，和成员变量一样，这里就不写了，嘻嘻

    boolean hasError = isInaccessibleViaGeneratedCode(annotationClass, "
methods", element);

    hasError |= isBindingInWrongPackage(annotationClass, element);



     //一个注解的方法不能有形同的 id，or 抛出异常

    Integer duplicateId = findDuplicate(ids);

    if (duplicateId != null) {

      error(element, "@%s annotation for method contains duplicate ID %d.
 (%s.%s)",

          annotationClass.getSimpleName(), duplicateId, enclosingElement.
getQualifiedName(),

          element.getSimpleName());

      hasError = true;

    }

     //获取该注解 ListenerClass.class 注解，什么意思呢？就是

     //butterknife 提供的方法注解 包含了另外一个注解

     //可以跳过代码看下面的文字说明。

    ListenerClass listener = annotationClass.getAnnotation(ListenerClass.
class);

    if (listener == null) {
```

```
      throw new IllegalStateException(
          String.format("No @%s defined on @%s.", ListenerClass.class.get
SimpleName(),
              annotationClass.getSimpleName()));
    }
//检查 id 的合法性，里面有个 Optional 注解
    for (int id : ids) {
      //id 为 -1 ，不合法
      if (id == NO_ID.value) {
        if (ids.length == 1) {
        //一个参数情况，且方法的参数适用了 Optional 注解，则抛出异常
          if (!required) {
            error(element, "ID-free binding must not be annotated with @O
ptional. (%s.%s)",
                enclosingElement.getQualifiedName(), element.getSimpleNam
e());
            hasError = true;
          }
        } else {
          error(element, "@%s annotation contains invalid ID %d. (%s.%s)
",
              annotationClass.getSimpleName(), id, enclosingElement.getQu
alifiedName(),
              element.getSimpleName());
          hasError = true;
```

```
    }
  }
}



  / / 获取实现的方法
ListenerMethod method;
ListenerMethod[] methods = listener.method();


// methods 就是 OnItemClick 注解的 ListenerClass 注解的初始化值，
比如下面这种，肯定是个
//    method = @ListenerMethod(
//   name = "onItemClick",
//   parameters = {
//   "android.widget.AdapterView<?>",
//    "android.view.View",
//       "int",
//       "long"
//    }
//  )
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
    if (methods.length > 1) {

    //抛异常，不可能走到这因为 butterknife 提供的都是 1 个默认的，能骗到我，我可
是上过小学的人，哈哈

      throw new IllegalStateException(String.format("Multiple listener m
ethods specified on @%s.",

          annotationClass.getSimpleName()));

    } else if (methods.length == 1) {

//如果有 method 属性值即这种 onItemClick，则 callbacks 必须为空，也就是 2 者不能
同时使用



      if (listener.callbacks() != ListenerClass.NONE.class) {

        throw new IllegalStateException(

            String.format("Both method() and callback() defined on @%s.",

                annotationClass.getSimpleName()));

      }

      method = methods[0];

    } else {

    // 否则使用 callback

    //反射 ListenerClass 注解中的 callback 方法



      Method annotationCallback = annotationClass.getDeclaredMethod("cal
lback");
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
      Enum<?> callback = (Enum<?>) annotationCallback.invoke(annotatio
n);

      Field callbackField = callback.getDeclaringClass().getField(callba
ck.name());

      method = callbackField.getAnnotation(ListenerMethod.class);



      //如果没有 ListenerMethod.class 注解 抛出异常，也就是说你使用了 callback，
则必须提供 ListenerMethod.class 注解



      if (method == null) {

        throw new IllegalStateException(

            String.format("No @%s defined on @%s's %s.%s.", ListenerMetho
d.class.getSimpleName(),

                annotationClass.getSimpleName(), callback.getDeclaringCla
ss().getSimpleName(),

                callback.name()));

      }

    }



    //检查方法的合法性，就是你使用的注解的方法的参数不能 butterknife 的参数的个
数（也就是 android 系统的那种）



    // Verify that the method has equal to or less than the number of par
ameters as the listener.

    List<? extends VariableElement> methodParameters = executableElement.
getParameters();

    if (methodParameters.size() > method.parameters().length) {
```

```
      error(element, "@%s methods can have at most %s parameter(s). (%s.%
s)",

          annotationClass.getSimpleName(), method.parameters().length,

          enclosingElement.getQualifiedName(), element.getSimpleName());

      hasError = true;

    }

//检查返回值，就是你使用的注解的方法的参数不能 butterknife 的参数的个数（也就是
android 系统的那种）



    // Verify method return type matches the listener.

    TypeMirror returnType = executableElement.getReturnType();

    if (returnType instanceof TypeVariable) {

      TypeVariable typeVariable = (TypeVariable) returnType;

      returnType = typeVariable.getUpperBound();

    }

    if (!returnType.toString().equals(method.returnType())) {

      error(element, "@%s methods must have a '%s' return type. (%s.%s)",

          annotationClass.getSimpleName(), method.returnType(),

          enclosingElement.getQualifiedName(), element.getSimpleName());

      hasError = true;

    }



    if (hasError) {

      return;
```

```
    }

//下面是方法参数的检查，不做分析了，太细了。记住一点就行了，你写的不和系统的实现方法一样就抛出异常



    Parameter[] parameters = Parameter.NONE;

    if (!methodParameters.isEmpty()) {

      parameters = new Parameter[methodParameters.size()];

      BitSet methodParameterUsed = new BitSet(methodParameters.size());

      String[] parameterTypes = method.parameters();

      for (int i = 0; i < methodParameters.size(); i++) {

        VariableElement methodParameter = methodParameters.get(i);

        TypeMirror methodParameterType = methodParameter.asType();

        if (methodParameterType instanceof TypeVariable) {

          TypeVariable typeVariable = (TypeVariable) methodParameterType;

          methodParameterType = typeVariable.getUpperBound();

        }



        for (int j = 0; j < parameterTypes.length; j++) {

          if (methodParameterUsed.get(j)) {

            continue;

          }

          if (isSubtypeOfType(methodParameterType, parameterTypes[j])

              || isInterface(methodParameterType)) {
```

```
          parameters[i] = new Parameter(j, TypeName.get(methodParameter
Type));

          methodParameterUsed.set(j);

          break;

        }

      }

      if (parameters[i] == null) {

        StringBuilder builder = new StringBuilder();

        builder.append("Unable to match @")

            .append(annotationClass.getSimpleName())

            .append(" method arguments. (")

            .append(enclosingElement.getQualifiedName())

            .append('.')

            .append(element.getSimpleName())

            .append(')');

        for (int j = 0; j < parameters.length; j++) {

          Parameter parameter = parameters[j];

          builder.append("\n\n  Parameter #")

              .append(j + 1)

              .append(": ")

              .append(methodParameters.get(j).asType().toString())

              .append("\n    ");

          if (parameter == null) {
```

```
            builder.append("did not match any listener parameters");
          } else {
            builder.append("matched listener parameter #")
                .append(parameter.getListenerPosition() + 1)
                .append(": ")
                .append(parameter.getType());
          }
        }
        builder.append("\n\nMethods may have up to ")
            .append(method.parameters().length)
            .append(" parameter(s):\n");
        for (String parameterType : method.parameters()) {
          builder.append("\n  ").append(parameterType);
        }
        builder.append(
            "\n\nThese may be listed in any order but will be searched f
or from top to bottom.");
        error(executableElement, builder.toString());
        return;
      }
    }
  }
//最后构造 MethodViewBinding 实体，形成方法的实体
```

```
    MethodViewBinding binding = new MethodViewBinding(name, Arrays.asLis
t(parameters), required);

    //构造 BindingClass

    BindingClass bindingClass = getOrCreateTargetClass(targetClassMap, e
nclosingElement);

    for (int id : ids) {



    //将生成的方法加入到 bindingClass 的方法集合中，一切都是为了生存 java 代码而
准备。



      if (!bindingClass.addMethod(getId(id), listener, method, binding))
 {

        error(element, "Multiple listener methods with return value speci
fied for ID %d. (%s.%s)",

            id, enclosingElement.getQualifiedName(), element.getSimpleNam
e());

        return;

      }

    }



    // Add the type-erased version to the valid binding targets set.

    erasedTargetNames.add(enclosingElement);

  }
```

ListenerClass / ListenerMethod 注解说明

```
@Target(METHOD)@Retention(CLASS)@ListenerClass(
```

```
    targetType = "android.widget.AdapterView<?>",

    setter = "setOnItemClickListener",

    type = "android.widget.AdapterView.OnItemClickListener",

    method = @ListenerMethod(

        name = "onItemClick",

        parameters = {

            "android.widget.AdapterView<?>",

            "android.view.View",

            "int",

            "long"

        }

    ))public @interface OnItemClick {

  /** View IDs to which the method will be bound. */

  @IdRes int[] value() default { View.NO_ID };}

import java.lang.annotation.Retention;

import java.lang.annotation.Target;



import static java.lang.annotation.ElementType.ANNOTATION_TYPE;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Retention(RUNTIME) @Target(ANNOTATION_TYPE)

public @interface ListenerClass {

  String targetType();
```

```
  /** Name of the setter method on the {@linkplain #targetType() target t
ype} for the listener. */

  String setter();



  /**

   * Name of the method on the {@linkplain #targetType() target type} to
remove the listener. If

   * empty {@link #setter()} will be used by default.

   */

  String remover() default "";



  /** Fully-qualified class name of the listener type. */

  String type();



  /** Enum which declares the listener callback methods. Mutually exclus
ive to {@link #method()}. */

  Class<? extends Enum<?>> callbacks() default NONE.class;



  /**

   * Method data for single-method listener callbacks. Mutually exclusiv
e with {@link #callbacks()}

   * and an error to specify more than one value.

   */

  ListenerMethod[] method() default { };
```

```
  /** Default value for {@link #callbacks()}. */

  enum NONE { }}

package butterknife.internal;



import java.lang.annotation.Retention;

import java.lang.annotation.Target;



import static java.lang.annotation.ElementType.FIELD;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Retention(RUNTIME) @Target(FIELD)

public @interface ListenerMethod {

  /** Name of the listener method for which this annotation applies. */

  String name();



  /** List of method parameters. If the type is not a primitive it must b
e fully-qualified. */

  String[] parameters() default { };



  /** Primitive or fully-qualified return type of the listener method. Ma
y also be {@code void}. */

  String returnType() default "void";



  /** If {@link #returnType()} is not {@code void} this value is returned
 when no binding exists. */
```

```
 String defaultReturn() default "null";}
```

可以把这 3 个整体来看。ListenerMethod 这个注解包含了方法的返回值，名字，参数，是实现的那些方法；ListenerClass 是 set 的那些方法属性，包含 setter 等，我们看到了 OnItemClick 设置的值就是我们平常写的那种，至此我们的 findAndParseTargets 方法算是走完了。

IOC 架构设计之 ButterKnife 源码&原理（二）下篇

处理器的核心方法 process 方法

```
 @Override public boolean process(Set<? extends TypeElement> elements, R
oundEnvironment env) {

 //这个方法查找所有的注解信息，并形成 BindingClass 集合

    Map<TypeElement, BindingClass> targetClassMap = findAndParseTargets
(env);

//遍历生存生成 java 文件

    for (Map.Entry<TypeElement, BindingClass> entry : targetClassMap.ent
rySet()) {

      TypeElement typeElement = entry.getKey();

      BindingClass bindingClass = entry.getValue();



      JavaFile javaFile = bindingClass.brewJava();

      try {

        javaFile.writeTo(filer);

      } catch (IOException e) {

        error(typeElement, "Unable to write binding for type %s: %s", typ
eElement, e.getMessage());

      }
```

```
    }

//表示处理器正确处理了

    return true;

  }
```

我们先看一下 BindingClass 的一些成员变量。这里我们主要是看关键的一些集合

```
  //这个是 BindView 注解在 findAndParseTargets 方法解析后 形成 ViewBindings
实体通过该类的 addxx 方法添加到 viewIdMap 集合的对象



  private final Map<Id, ViewBindings> viewIdMap = new LinkedHashMap<>();



     //这个是 BindViews 注解 在 findAndParseTargets 方法解析后 形成的.

     BindViews 用法:



     @BindViews({ R2.id.title, R2.id.subtitle, R2.id.hello }) List<View>
 headerViews;




     //key:FieldCollectionViewBinding 为实体，包含了名字，类型等信息，大家可
以看下源码，这里就不说了就是一个 java—bean ，你可以这么理解。

     //value:

     就是这些 id:  R2.id.title, R2.id.subtitle, R2.id.hello




  private final Map<FieldCollectionViewBinding, List<Id>> collectionBind
ings =
```

```
      new LinkedHashMap<>();



      //该集合是 BindBitmap 注解在 findAndParseTargets 注解在 findAndParseTar
gets 方法解析后 形成 FieldBitmapBinding 实体通过该类的 addxx 方法添加到 bitmap
Bindings 集合的对象



  private final List<FieldBitmapBinding> bitmapBindings = new ArrayList
<>();

  //这个就是 BindDrawable 注解....

  private final List<FieldDrawableBinding> drawableBindings = new ArrayL
ist<>();

  //这个就是 BindFloat,BindInt,BindString... 这些注解....

  private final List<FieldResourceBinding> resourceBindings = new ArrayL
ist<>();
```

上面的 addxxx 方法类似下面的,当然还有其它的就不贴出了

```
void addFieldCollection(List<Id> ids, FieldCollectionViewBinding binding)
 {

    collectionBindings.put(binding, ids);

  }



  boolean addMethod(

      Id id,

      ListenerClass listener,

      ListenerMethod method,

      MethodViewBinding binding) {

    ViewBindings viewBindings = getOrCreateViewBindings(id);
```

```
    if (viewBindings.hasMethodBinding(listener, method) && !"void".equal
s(method.returnType())) {

      return false;

    }

    viewBindings.addMethodBinding(listener, method, binding);

    return true;

  }



  void addResource(FieldResourceBinding binding) {

    resourceBindings.add(binding);

  }



  //......
```

至此我们算是完成了一个大步走，我们把所有的注解 方法等放倒了一个集合里，生成 java 文件的时候就可以遍历这些集合了，从而形成有一定规律的 java 源文件。
形成的文件就是：
这里我们拿官方的 demo－SimpleActivity
编译完后最后生成的文件为：SimpleActivity_ViewBinding
离目标越来越近了。

.java 文件生成
生成.java 文件你可以用字符串拼接的笨方法，但是我们的 jake 大神怎么可能那么弱。人家自己又写了一个 javapoet 库。

```
    compile 'com.squareup:javapoet:1.7.0'

这个库可以很方便的生成我们想要的 java 文件。

在 findAndParseTargets 方法 //...

//遍历生存生成 java 文件
```

```
    for (Map.Entry<TypeElement, BindingClass> entry : targetClassMap.entrySet()) {

      TypeElement typeElement = entry.getKey();

      BindingClass bindingClass = entry.getValue();



      JavaFile javaFile = bindingClass.brewJava();

      try {

        javaFile.writeTo(filer);

      } catch (IOException e) {

        error(typeElement, "Unable to write binding for type %s: %s", typeElement, e.getMessage());

      }

    }

 //...

 //遍历生存生成 java 文件

    for (Map.Entry<TypeElement, BindingClass> entry : targetClassMap.entrySet()) {

      TypeElement typeElement = entry.getKey();

      BindingClass bindingClass = entry.getValue();



      JavaFile javaFile = bindingClass.brewJava();

      try {

        javaFile.writeTo(filer);

      } catch (IOException e) {
```

```
        error(typeElement, "Unable to write binding for type %s: %s", typ
eElement, e.getMessage());

      }

    }
```

遍历 targetClassMap 集合，调用每一个类的 brewJava()方法，最后返回 JavaFile 对象，再通过 writeTo 方法生成 java 文件。
接下来我们就看下 bindingClass.brewJava（）方法。

```
//我们可以看到 javapoet 这个开源库使用了设计模式—builder 模式。// 我们看源码就
是看这些模式啦，思想啦，才能提高自己的代码编写能力//bug 少了才有时间干别的呀，哈
哈 。。。你懂我的



  JavaFile brewJava() {

  //参数 1:包名

   //参数 2:TypeSpec，这个可以生成 class ，interface 等 java 文件

    return JavaFile.builder(bindingClassName.packageName(), createBindin
gClass())

    //注释

        .addFileComment("Generated code from Butter Knife. Do not modify!
")

        .build();  }
```

主要步骤

1. 生成类名
2. 生成构造函数
3. 生成 unbind 方法。

892872246

```
  private TypeSpec createBindingClass() {



  1. 生成类名
```

```
 //bindingClassName.simpleName() 就是我们在 findAndParseTargets 方法形成的
类的名字：xxx_ViewBinding 名字

 xxx 也就是使用 butterknife 的类名



   TypeSpec.Builder result = TypeSpec.classBuilder(bindingClassName.sim
pleName())

   //增加类修饰符 public

       .addModifiers(PUBLIC);



 //使用 butterknife 的类是 final 类

   TypeName targetType;

   if (isFinal) {

    //增加类修饰符 FINAL

     result.addModifiers(FINAL);

     targetType = targetTypeName;

   } else {

   //不是 final 类，增加泛型参数 T

     targetType = TypeVariableName.get("T");

     result.addTypeVariable(TypeVariableName.get("T", targetTypeName));

   }

//如果有父类直接继承

   if (hasParentBinding()) {

result.superclass(ParameterizedTypeName.get(getParentBinding(), targetTy
pe));
```

```
    } else {

    //增加实现接口

      result.addSuperinterface(UNBINDER);

       //增加成员变量

      result.addField(targetType, "target", isFinal ? PRIVATE : PROTECTE
D);

    }



    //到这里生成的代码如下:

     / / 比如在 activity

    public class SimpleActivity_ViewBinding<T extends SimpleActivity> im
plements Unbinder {

  protected T target;

  }

  //比如在 adapter 的 viewholder 中 public final class SimpleAdapter$ViewHol
der_ViewBinding implements Unbinder {

  private SimpleAdapter.ViewHolder target;

}



2. 生成构造函数



    if (!bindNeedsView()) {

      // Add a delegating constructor with a target type + view signature
 for reflective use.
```

BAT交流群892246

```
    result.addMethod(createBindingViewDelegateConstructor(targetTyp
e));

  }

  result.addMethod(createBindingConstructor(targetType));



3. 生成 unbind 方法。

  if (hasViewBindings() || !hasParentBinding()) {

    result.addMethod(createBindingUnbindMethod(result, targetType));

  }



  return result.build();

}
```

（一）生成类名

代码里注释的比较详细了

（二）生成构造函数

主要是 createBindingConstructor 方法，主要是对成员变量赋值，以及设置监听事件。先看下 javapoet 提供的几个方法或类:

1. **MethodSpec：生成方法的辅助类**

这里主要是生成构造函数，当然也可以生成其它的普通方法，构造函数也是方法的一种吗。
通过 constructorBuilder 构造出一个方法,通过 addAnnotation 添加注解，通过 addModifiers 添加修饰符。

```
MethodSpec.Builder constructor =

MethodSpec.constructorBuilder()

      .addAnnotation(UI_THREAD)
```

```
        .addModifiers(PUBLIC);
```

通过如下方法添加参数 targetType 为参数类型，"target"为参数的变量名

```
    constructor.addParameter(targetType, "target");
```

通过如下方法添加代码语句
第一参数是 String 类型，可以有占位符 S（字符串类型），S（字符串类型），T（类型）等
第二个参数 Object… args，类型，多参数不固定。
就像你平时使用 String.format()方法一样的意思.

```
   constructor.addStatement("this.target = target");
```

2. **构造函数中变量的赋值**

这里主要是对使用了如下的代码进行一个赋值操作。

```
 @BindView(R2.id.title) TextView title;

@OnClick(R2.id.hello) void sayHello() {}
```

这里我们主要看一下关键方法，因为都是类似的拼接代码字符串。
我们看一下变量的赋值：

```
  private void addViewBindings(MethodSpec.Builder result, ViewBindings b
indings) {

    if (bindings.isSingleFieldBinding()) {

      // Optimize the common case where there's a single binding directly
 to a field.

      FieldViewBinding fieldBinding = bindings.getFieldBinding();

      CodeBlock.Builder builder = CodeBlock.builder()

          .add("target.$L = ", fieldBinding.getName());



      boolean requiresCast = requiresCast(fieldBinding.getType());

      if (!requiresCast && !fieldBinding.isRequired()) {

        builder.add("source.findViewById($L)", bindings.getId().code);
```

```
      } else {

        builder.add("$T.find", UTILS);

        builder.add(fieldBinding.isRequired() ? "RequiredView" : "Optiona
lView");

        if (requiresCast) {

          builder.add("AsType");

        }

        builder.add("(source, $L", bindings.getId().code);

        if (fieldBinding.isRequired() || requiresCast) {

          builder.add(", $S", asHumanDescription(singletonList(fieldBindi
ng)));

        }

        if (requiresCast) {

          builder.add(", $T.class", fieldBinding.getRawType());

        }

        builder.add(")");

      }

      result.addStatement("$L", builder.build());

      return;

    }



    List<ViewBinding> requiredViewBindings = bindings.getRequiredBinding
s();

    if (requiredViewBindings.isEmpty()) {
```

```
      result.addStatement("view = source.findViewById($L)", bindings.get
Id().code);

    } else if (!bindings.isBoundToRoot()) {

      result.addStatement("view = $T.findRequiredView(source, $L, $S)",
UTILS,

          bindings.getId().code, asHumanDescription(requiredViewBinding
s));

    }



    addFieldBindings(result, bindings);

    addMethodBindings(result, bindings);

  }
```

主要是调用系统的 findViewById 方法，但是你看到了 findRequiredViewAsType,findRequiredView 方法和 castView 方法，findRequiredView,findRequiredViewAsType 是作者为乐代码的书写方便对 findViewById 的一层封装，你可以看一下源码,最后都会调用的 findRequiredView 方法的 findViewById 方法。

```
 public static View findRequiredView(View source, @IdRes int id, String
who) {

    View view = source.findViewById(id);

    if (view != null) {

      return view;

    }

    String name = getResourceEntryName(source, id);

    throw new IllegalStateException("Required view '"

        + name

        + "' with ID "
```

```
        + id
        + " for "
        + who
        + " was not found. If this view is optional add '@Nullable' (fiel
ds) or '@Optional'"
        + " (methods) annotation.");
 }
```

这个 castView 是什么方法呢？是 Class 类的方法，直接转换为指定的类型

```
  public static <T> T castView(View view, @IdRes int id, String who, Cla
ss<T> cls) {
    try {
      return cls.cast(view);
    } catch (ClassCastException e) {
      String name = getResourceEntryName(view, id);
      throw new IllegalStateException("View '"
          + name
          + "' with ID "
          + id
          + " for "
          + who
          + " was of the wrong type. See cause for more info.", e);
    }
  }
```

BAT交 892246

说白了都是调用系统的方法。
好了到这里成员变量的赋值算是完了。
注意一点 target.title target 就是我们的 activity 或者 view ； 也验证了为什么是使用了类似 BindView 注解不能是 private 修饰符的另一个原因了。

接下来是方法的监听 private void addMethodBindings(MethodSpec.Builder result, ViewBindings bindings) {} 方法，李 main 也是通过循环添加方法借助我们上文提到的
MethodSpec.methodBuilder 构造器

```
 for (ListenerMethod method : getListenerMethods(listener)) {

       MethodSpec.Builder callbackMethod = MethodSpec.methodBuilder(met
hod.name())

           .addAnnotation(Override.class)

           .addModifiers(PUBLIC)

           .returns(bestGuess(method.returnType()));

       String[] parameterTypes = method.parameters();

       for (int i = 0, count = parameterTypes.length; i < count; i++) {

         callbackMethod.addParameter(bestGuess(parameterTypes[i]), "p" +
 i);

         //...

       }
```

感兴趣的可以根据生成的代码来对照这查看，这里就不多说了。
最后生成的如下所示的代码。

```
 @UiThread

 public SimpleActivity_ViewBinding(final T target, View source) {

   this.target = target;



   View view;
```

```
    target.title = Utils.findRequiredViewAsType(source, R.id.title, "fie
ld 'title'", TextView.class);

    target.subtitle = Utils.findRequiredViewAsType(source, R.id.subtitle,
 "field 'subtitle'", TextView.class);

    view = Utils.findRequiredView(source, R.id.hello, "field 'hello', me
thod 'sayHello', and method 'sayGetOffMe'");

    target.hello = Utils.castView(view, R.id.hello, "field 'hello'", But
ton.class);

    view2130968578 = view;

    view.setOnClickListener(new DebouncingOnClickListener() {

      @Override

      public void doClick(View p0) {

        target.sayHello();

      }

    });

    view.setOnLongClickListener(new View.OnLongClickListener() {

      @Override

      public boolean onLongClick(View p0) {

        return target.sayGetOffMe();

      }

    });

    view = Utils.findRequiredView(source, R.id.list_of_things, "field 'l
istOfThings' and method 'onItemClick'");

    target.listOfThings = Utils.castView(view, R.id.list_of_things, "fie
ld 'listOfThings'", ListView.class);

    view2130968579 = view;
```

```
    ((AdapterView<?>) view).setOnItemClickListener(new AdapterView.OnIte
mClickListener() {

      @Override

      public void onItemClick(AdapterView<?> p0, View p1, int p2, long p3)
 {

        target.onItemClick(p2);

      }

    });

    target.footer = Utils.findRequiredViewAsType(source, R.id.footer, "f
ield 'footer'", TextView.class);

    target.headerViews = Utils.listOf(

        Utils.findRequiredView(source, R.id.title, "field 'headerViews'
"),

        Utils.findRequiredView(source, R.id.subtitle, "field 'headerView
s'"),

        Utils.findRequiredView(source, R.id.hello, "field 'headerViews'
"));

  }
```

（三）生成 unbind 方法

createBindingUnbindMethod 方法主要是把的成员变量啦，Listener 等 置为空，

比如 setOnClickListener（null）。

```
    private MethodSpec createBindingUnbindMethod(TypeSpec.Builder bindin
gClass,

      TypeName targetType) {

    MethodSpec.Builder result = MethodSpec.methodBuilder("unbind")
```

BA 892246

```
        .addAnnotation(Override.class)

        .addModifiers(PUBLIC);

    if (!isFinal && !hasParentBinding()) {

      result.addAnnotation(CALL_SUPER);

    }

    boolean rootBindingWithFields = !hasParentBinding() && hasFieldBindi
ngs();

    if (hasFieldBindings() || rootBindingWithFields) {

      result.addStatement("$T target = this.target", targetType);

    }

    if (!hasParentBinding()) {

      String target = rootBindingWithFields ? "target" : "this.target";

      result.addStatement("if ($N == null) throw new $T($S)", target, Ill
egalStateException.class,

          "Bindings already cleared.");

    } else {

      result.addStatement("super.unbind()");

    }



    if (hasFieldBindings()) {

      result.addCode("\n");

      for (ViewBindings bindings : viewIdMap.values()) {

        if (bindings.getFieldBinding() != null) {
```

```
        result.addStatement("target.$L = null", bindings.getFieldBindin
g().getName());

      }

    }

    for (FieldCollectionViewBinding fieldCollectionBinding : collectio
nBindings.keySet()) {

      result.addStatement("target.$L = null", fieldCollectionBinding.g
etName());

    }

  }



  if (hasMethodBindings()) {

    result.addCode("\n");

    for (ViewBindings bindings : viewIdMap.values()) {

      addFieldAndUnbindStatement(bindingClass, result, bindings);

    }

  }



  if (!hasParentBinding()) {

    result.addCode("\n");

    result.addStatement("this.target = null");

  }



  return result.build();
```

```
}
```

主要就是 addStatement 方法，上文已经说了，该方法的意思就是生成一句代码，第一参数是 String 类型，可以有占位符 S（字符串类型），S（字符串类型），T（类型）等
第二个参数 Object… args，类型，多参数不固定。
就像你平时使用 String.format()方法一样的意思.
比较简单，最后生成的方法类似：

```
@Override

public void unbind() {

  SimpleAdapter.ViewHolder target = this.target;

  if (target == null) throw new IllegalStateException("Bindings alread
y cleared.");



  target.word = null;

  target.length = null;

  target.position = null;



  this.target = null;

}
```

最后我们看下对外提供的 API。

bind 方法

那么还差一步，什么时候都要我们生成的 java 文件呢？
答案是： ButterKnife.bind(this);方法。
我们看一下 ButterKnife 对外提供的 API

```
/**

 * BindView annotated fields and methods in the specified {@link Activi
ty}. The current content
```

892872246

```
 * view is used as the view root.
 *
 * @param target Target activity for view binding.
 */
@NonNull @UiThread
public static Unbinder bind(@NonNull Activity target) {
  View sourceView = target.getWindow().getDecorView();
  return createBinding(target, sourceView);
}


/**
 * BindView annotated fields and methods in the specified {@link View}.
The view and its children
 * are used as the view root.
 *
 * @param target Target view for view binding.
 */
@NonNull @UiThread
public static Unbinder bind(@NonNull View target) {
  return createBinding(target, target);
}


/**
```

```
  * BindView annotated fields and methods in the specified {@link Dialo
g}. The current content

  * view is used as the view root.

  *

  * @param target Target dialog for view binding.

  */

 @NonNull @UiThread

 public static Unbinder bind(@NonNull Dialog target) {

   View sourceView = target.getWindow().getDecorView();

   return createBinding(target, sourceView);

 }



 /**

  * BindView annotated fields and methods in the specified {@code target}
 using the {@code source}

  * {@link Activity} as the view root.

  *

  * @param target Target class for view binding.

  * @param source Activity on which IDs will be looked up.

  */

 @NonNull @UiThread

 public static Unbinder bind(@NonNull Object target, @NonNull Activity
source) {

   View sourceView = source.getWindow().getDecorView();
```

```
    return createBinding(target, sourceView);
  }


  /**
   * BindView annotated fields and methods in the specified {@code target}
using the {@code source}
   * {@link View} as the view root.
   *
   * @param target Target class for view binding.
   * @param source View root on which IDs will be looked up.
   */
  @NonNull @UiThread
  public static Unbinder bind(@NonNull Object target, @NonNull View sour
ce) {
    return createBinding(target, source);
  }


  /**
   * BindView annotated fields and methods in the specified {@code target}
using the {@code source}
   * {@link Dialog} as the view root.
   *
   * @param target Target class for view binding.
   * @param source Dialog on which IDs will be looked up.
```

```
  */

 @NonNull @UiThread

 public static Unbinder bind(@NonNull Object target, @NonNull Dialog so
urce) {

   View sourceView = source.getWindow().getDecorView();

   return createBinding(target, sourceView);

 }



 private static Unbinder createBinding(@NonNull Object target, @NonNull
 View source) {

   Class<?> targetClass = target.getClass();

   if (debug) Log.d(TAG, "Looking up binding for " + targetClass.getName
());

   Constructor<? extends Unbinder> constructor = findBindingConstructor
ForClass(targetClass);



   if (constructor == null) {

     return Unbinder.EMPTY;

   }



   //noinspection TryWithIdenticalCatches Resolves to API 19+ only typ
e.

   try {

     return constructor.newInstance(target, source);

   } catch (IllegalAccessException e) {
```

```
      throw new RuntimeException("Unable to invoke " + constructor, e);

    } catch (InstantiationException e) {

      throw new RuntimeException("Unable to invoke " + constructor, e);

    } catch (InvocationTargetException e) {

      Throwable cause = e.getCause();

      if (cause instanceof RuntimeException) {

        throw (RuntimeException) cause;

      }

      if (cause instanceof Error) {

        throw (Error) cause;

      }

      throw new RuntimeException("Unable to create binding instance.", ca
use);

    }

  }
```

我们看到 bind 的一系列方法都会调用 createBinding 方法

```
   private static Unbinder createBinding(@NonNull Object target, @NonNul
l View source) {

    Class<?> targetClass = target.getClass();

    if (debug) Log.d(TAG, "Looking up binding for " + targetClass.getName
());

    Constructor<? extends Unbinder> constructor = findBindingConstructor
ForClass(targetClass);



    if (constructor == null) {
```

```
      return Unbinder.EMPTY;
    }

    //noinspection TryWithIdenticalCatches Resolves to API 19+ only typ
e.
    try {
      return constructor.newInstance(target, source);
    } catch (IllegalAccessException e) {
      throw new RuntimeException("Unable to invoke " + constructor, e);
    } catch (InstantiationException e) {
      throw new RuntimeException("Unable to invoke " + constructor, e);
    } catch (InvocationTargetException e) {
      Throwable cause = e.getCause();
      if (cause instanceof RuntimeException) {
        throw (RuntimeException) cause;
      }
      if (cause instanceof Error) {
        throw (Error) cause;
      }
      throw new RuntimeException("Unable to create binding instance.", ca
use);
    }
  }
```

BAT交流 892246

Class<?> （？是转义字符 csdn 要不支持）targetClass = target.getClass();获取类的实例，最后获取构造函数，最后 constructor.newInstance 方法来调用该类的构造函数。
而该类的构造函数是通过 findBindingConstructorForClass 方法，我可没来看下此方法：

```
  @Nullable @CheckResult @UiThread

 private static Constructor<? extends Unbinder> findBindingConstructorF
orClass(Class<?> cls) {

 //缓存中查找，找到直接返回

   Constructor<? extends Unbinder> bindingCtor = BINDINGS.get(cls);

   if (bindingCtor != null) {

     if (debug) Log.d(TAG, "HIT: Cached in binding map.");

     return bindingCtor;

   }



    //检查合法性

   String clsName = cls.getName();

   if (clsName.startsWith("android.") || clsName.startsWith("java.")) {

     if (debug) Log.d(TAG, "MISS: Reached framework class. Abandoning se
arch.");

     return null;

   }

   try {

   //构造一个 class，可以看类名就是我们生成的。

     Class<?> bindingClass = Class.forName(clsName + "_ViewBinding");

     //noinspection unchecked
```

```
        // 获取我们的构造函数

      bindingCtor = (Constructor<? extends Unbinder>) bindingClass.getCo
nstructor(cls, View.class);

      if (debug) Log.d(TAG, "HIT: Loaded binding class and constructor.
");

    } catch (ClassNotFoundException e) {

      if (debug) Log.d(TAG, "Not found. Trying superclass " + cls.getSupe
rclass().getName());

      bindingCtor = findBindingConstructorForClass(cls.getSuperclass());

    } catch (NoSuchMethodException e) {

      throw new RuntimeException("Unable to find binding constructor for
" + clsName, e);

    }

     //加入到缓存中

    BINDINGS.put(cls, bindingCtor);

    return bindingCtor;

  }
```

可以看到用到了简单的缓存。

unbind 方法

在新版的 8.4.0 中去除了 unbind 方法。

```
ButterKnife.unbind(this);
```

采用了接口的形式。让生成的类来实现，比如：

```
 public final class SimpleAdapter$ViewHolder_ViewBinding implements Unbi
nder {
```

```
  @UiThread

 public SimpleAdapter$ViewHolder_ViewBinding(SimpleAdapter.ViewHolder t
arget, View source) {

 //...

 }

//...

 @Override

 public void unbind() {

 /...

 }

}
```

那如何 unbind 呢？

ButterKnife.bind(this)返回值是一个我们生成的 java 文件类的实例对象。返回的是一个 Unbinder 正和此意。

使用如下：

```
 private  Unbinder unbinder;

 @Override protected void onCreate(Bundle savedInstanceState) {

   super.onCreate(savedInstanceState);

   setContentView(R.layout.simple_activity);

    unbinder=ButterKnife.bind(this);

    }

      @Override

 protected void onDestroy() {
```

```
    super.onDestroy();

    unbinder.unbind();

  }
```

至此我们 ButterKnife 源码&原理就结束了。

7.3 Dagger 架构设计核心解密

一、概述

Dagger2 依赖注入框架的好处：

- 依赖的注入和配置独立于组件之外
- 依赖对象是在一个独立、不耦合的地方初始化，当初始化方式改变的时候修改的代码少。
- 依赖注入使得单元测试更加简单。

Dagger2 相对于其它框架的优点：

- 编译期生成代码，有错误会在编译期报出。
- 错误可追踪。
- 易于调试。

Dagger2 的缺点：

- 缺少灵活性。
- 没有动态机制。

二、Dagger2 的注解

Dagger2 的注解主要有以下七类：

- @Inject：这个注解有两个作用：在目标类中标记成员变量告诉 Dagger 这个类型的变量需要一个实例对象；标记依赖类中的构造方法，告诉 Dagger 我可以提供这种类型的依赖实例。
- @Component：用来标记接口或者抽象类，也被称为注入器，是@Inject 和@Module 的桥梁，所有的 Component 都可以通过它的 modules 知道它所提供的依赖范围，一个 Componet 可以依赖一个或多个 Component，并拿到被依赖 Component 暴露出来的实例，Componenet 的 dependencies 属性就是确定依赖关系的实现。

BAT交流群：892872246

- @Module：用来标记类，一般类名以 Module 结尾，Module 的主要作用是用来集中管理@Provides 标记的方法，我们定义一个被@Module 注解的类，Dagger 就会知道在哪里找到依赖来满足创建类的实例，Module 的一个重要特征是被设计成区块并可以组合在一起。
- @Provides：对方法进行注解，并且这些方法都是有返回类型的，告诉 Dagger 我们向如何创建并提供该类型的依赖实例（一般会在方法中 new 出实例），用 @Provides 标记的方法，推荐用 provide 作为前缀。
- @Qualifier：限定符，当一个类的类型不足以标示一个依赖的时候，我们就可以用这个注解，它会调用 DataModule 中方法来返回合适的依赖类实例。
- @Scope：通过自定义注解来限定作用域，所有的对象都不再需要知道怎么管理它的实例，Dagger2 中有一个默认的作用域注解@Singleton，通常用来标记在 App 整个生命周期内存活的实例，也可以定义一个@PerActivity 注解，用来表明生命周期要与 Activity 一致。
- @SubComponent：如果我们需要父组件全部的提供对象，我们就可以用包含方式，而不是用依赖方式，包含方式不需要父组件显示显露对象，就可以拿到父组件全部对象，且 SubComponent 只需要在父 Component 接扣中声明就可以了。

三、Dagger2 的简单应用 - @Inject 和@Component

```
第一步：基础配置，在 build.gradle 中添加相应的依赖：//添加(1)

apply plugin: 'com.neenbedankt.android-apt'



buildscript {

    repositories {

        jcenter()

    }

    dependencies {

        //添加(2)

        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'

    }}



android {
```

```
    compileSdkVersion 23

    buildToolsVersion "23.0.0"



    defaultConfig {

        applicationId "com.demo.zejun.repodragger2"

        minSdkVersion 15

        targetSdkVersion 23

        versionCode 1

        versionName "1.0"

    }

    buildTypes {

        release {

            minifyEnabled false

            proguardFiles getDefaultProguardFile('proguard-android.txt'),
 'proguard-rules.pro'

        }

    }}



dependencies {

    compile fileTree(dir: 'libs', include: ['*.jar'])

    testCompile 'junit:junit:4.12'

    compile 'com.android.support:appcompat-v7:23.0.0'

    //添加(3)
```

```
    apt 'com.google.dagger:dagger-compiler:2.0'

    //添加(4)

    compile 'com.google.dagger:dagger:2.0'}
```

第二步：`User`作为目标类中需要实例化的成员对象，给其构造函数添加`@Inject`标签：

```
public class User {



    public String name;



    @Inject

    public User() {

        name = "lizejun";

    }



    public String getName() {

        return name;

    }}
```

第三步：声明`Component`：

```
@Component()public interface OnlyInjectComponent {

    void inject(AnnotationActivity annotationActivity);}
```

第四步：在目标类中添加注解`@Inject`，并根据我们第 3 步中声明的`Component`，调用`DaggerXXX`方法来进行注入：

```
public class AnnotationActivity extends AppCompatActivity {
```

BAT交流群：892872246

```
    @Inject
    public User mUser;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_dragger2);
        //在第 3 步声明的 Component 接口或者抽象类的基础上，添加 Dagger 前缀。
        DaggerOnlyInjectComponent.builder().build().inject(this);
    }
}
```

上面这个例子有两个缺点：

- 只能标记一个构造方法，因为如果标记两个以上，不知道要用哪一个构造提供实例。
- 不能标记其它我们不能修改的类，例如第三方库。
- 如果用`@Inject`标记的构造函数如果有参数，那么这个参数也需要其它地方提供依赖，而类似于 `String` 这些我们不能修改的类，只能用`@Module`中的`@Provides`来提供实例了。

四、采用@Module 来提供依赖

采用`@Module`标记的类提供依赖是常规套路，`@Module`标记的类起管理作用，真正提供依赖实例靠的是`@Provides`标记的带返回类型的方法。
第一步：和上面类似，我们定义一个依赖类，但是它的构造方法并不需要用`@Inject`标记：

```
public class Person {

    private String name;
```

892872246

```
    public Person() {
        this.name = "lizejun";
    }

    public String getName() {
        return name;
    }}
```

第二步：我们需要定义一个`@Module`来管理这些依赖类的实例：

```
@Modulepublic class PersonDataModule {

    @Provides
    public Person providePerson() {
        return new Person();
    }}
```

第三步：定义一个`@Component`，它指向上面定义的`@Module`

```
@Component(modules = {PersonDataModule.class})public interface PersonInj
ectComponent {
    void inject(PersonInjectActivity injectActivity);}
```

第四步：在目标类中进行依赖注入

```
public class PersonInjectActivity extends Activity {

    @Inject
```

```
    Person mPerson;



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        DaggerPersonInjectComponent.create().inject(this);

        System.out.println("Person name=" + mPerson.getName());

    }}
```

这里注入的方式有两种，一种是像上面这样的，它适合于 `PersonDataModule` 中只有一个无参的构造方法，否则我们需要这样调用：

```
DaggerPersonInjectComponent.builder().personDataModule(new PersonDataMod
ule()).build().inject(this);
```

五、初始化依赖实例的步骤

查找 `Module` 中是否存在创建该类型的方法（即`@Component` 标记的接口中包含了`@Module` 标记的 `Module` 类，如果没有则直接查找`@Inject` 对应的构造方法）。

如果存在创建类方法，则查看该方法是否有参数

-
-

如果不存在参数，直接初始化该类的实例，一次依赖注入到此结束。

-
-

如果存在参数，则从步骤 1 开始初始化每个参数。

-
-

BAT交流群：892872246

如果不存在创建类方法，则查找该类型的类中有@Inject标记的构造方法，查看构造方法是否有参数：

-
-

如果不存在参数，则直接初始化该类实例，一次依赖注入到此结束。

-
-

如果存在参数，则从步骤 1 开始初始化每个参数。

-

六、@Qualifier 限定符

在 Dagger 中，有一个已经定义好的限定符，@Name，下面我们也自己定义一个限定符：

```
@Qualifier@Retention(RetentionPolicy.RUNTIME)public @interface PeopleThr
eeQualifier {}
```

第一步：和前面类似，我们先定义一个需要实例化的依赖类：

```
public class People {



    private int count;



    public People() {

        count = 0;

    }



    public People(int count) {

        this.count = count;
```

```
    }


    public int getCount() {

        return count;

    }}
```

第二步：我定义一个 `DataModule`，和前面不同的是，在它的 `provideXXX` 方法的注解中，我们添加了 `@Name(xxx)` 和自定义的注解 `PeopleThreePeople`：

```
@Modulepublic class PeopleDataModule {


    @Provides

    @Named("Five People")

    People provideFivePeople() {

        return new People(5);

    }


    @Provides

    @Named("Ten People")

    People provideTenPeople() {

        return new People(10);

    }


    @Provides

    @PeopleThreeQualifier
```

```
    People provideThreePeople() {

        return new People(3);

    }}
```

第三步：定义 `Component`

```
@Component(modules = PeopleDataModule.class)public interface PeopleInjec
tComponent {

    void inject(PeopleInjectActivity peopleInjectActivity);}
```

第四步：在目标类中进行依赖注入，在提供`@Inject`注解时，我们还需要声明和`PeopleDataModule`中对应的限定符，这样`Dagger`就知道该用那个函数来生成目标类中的依赖类实例：

```
public class PeopleInjectActivity extends Activity {



    @Inject

    @Named("Five People")

    People mFivePeople;



    @Inject

    @Named("Ten People")

    People mTenPeople;



    @Inject

    @PeopleThreeQualifier

    People mThreePeople;



    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        DaggerPeopleInjectComponent.builder().peopleDataModule(new Peopl
eDataModule()).build().inject(this);

        System.out.println("Five People=" + mFivePeople.getCount() + ",Te
n People=" + mTenPeople.getCount() + ", Three People=" + mThreePeople.ge
tCount());

    }}
```

七、@Scope

@Scope 的作用主要是在组织 Component 和 Module 的时候起到一个提醒和管理的作用，在 Dagger 中，有一个默认的作用域@Singleton。
@Scope 的作用是：Dagger2 可以通过自定义 Scope 注解，来限定通过 Module 和 Inject 方式创建的类的实例的生命周期能够与目标类的生命周期相同。Scope 的真正作用在与 Component 的组织：

- 更好的管理 Component 之间的组织方式，不管是依赖方式还是包含方式，都有必要用自定的 Scope 注解标注这些 Component，而且编译器会检查有依赖关系或包含关系的 Component，若发现有 Component 没有用自定义 Scope 注解，则会报错。
- 更好地管理 Component 与 Module 之间地关系，编译器会检查 Component 管理的 Module，若发现 Component 的自定义 Scope 注解与 Module 中的标注创建类实例方法的注解不一样，就会报错。
- 提高程序的可读性。

下面是一个使用@Singleton 的例子：
第一步：定义需要实例化的类：

```
public class AnSingleObject {



    private String objectId;



    public AnSingleObject() {

        objectId = toString();
```

```
    }



    public String getObjectId() {

        return objectId;

    }}
```

第二步：定义 `DataModule`，在它的 `provideXXX` 方法，提供了 `@Singletion` 注解：

```
@Modulepublic class AnSingleObjectDataModule {



    @Provides

    @Singleton

    AnSingleObject provideAnSingleObject() {

        return new AnSingleObject();

    }}
```

第三步：定义 `Component`，和前面不同的是，需要给这个 `Component` 添加 `@Singleton` 注解：

```
@Component(modules = {AnSingleObjectDataModule.class})@Singletonpu
blic abstract class AnSingleObjectInjectComponent {



    private static AnSingleObjectInjectComponent sInstance;



    public abstract void inject(AnSingleObjectInjectActivity anSingleObj
ectInjectActivity);



    public static AnSingleObjectInjectComponent getInstance() {

        if (sInstance == null) {
```

```
            sInstance = DaggerAnSingleObjectInjectComponent.create();

        }

        return sInstance;

    }}

@Component(modules = {AnSingleObjectDataModule.class})@Singletonpublic a
bstract class AnSingleObjectInjectComponent {



    private static AnSingleObjectInjectComponent sInstance;



    public abstract void inject(AnSingleObjectInjectActivity anSingleObj
ectInjectActivity);



    public static AnSingleObjectInjectComponent getInstance() {

        if (sInstance == null) {

            sInstance = DaggerAnSingleObjectInjectComponent.create();

        }

        return sInstance;

    }}
```

第三步：定义 `Component`，和前面不同的是，需要给这个 `Component` 添加`@Singleton` 注解：

```
@Component(modules = {AnSingleObjectDataModule.class})@Singletonpublic a
bstract class AnSingleObjectInjectComponent {



    private static AnSingleObjectInjectComponent sInstance;
```

```
    public abstract void inject(AnSingleObjectInjectActivity anSingleObj
ectInjectActivity);



    public static AnSingleObjectInjectComponent getInstance() {

        if (sInstance == null) {

            sInstance = DaggerAnSingleObjectInjectComponent.create();

        }

        return sInstance;

    }}
```

第四步：在目标类中进行依赖注入，每次启动 `Activity` 的时候，我们可以发现打印出来的 `hash` 值都是相同的：

```
public class AnSingleObjectInjectActivity extends Activity {



    @Inject

    AnSingleObject object;



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        AnSingleObjectInjectComponent.getInstance().inject(this);

        System.out.println("AnSingleObject id=" + object.getObjectId());

    }}
```

八、组织 Component

BAT交流 892246

Component 有三种组织方式：

- 依赖：一个 Component 依赖一个或多个 Component，采用的是@Component 的 dependencies 属性。
- 包含：这里就用到了@SubComponent 注解，用它来标记接口或者抽象类，表示它可以被包干。一个 Component 可以包含一个或多个 Component，而且被包含的 Component 还可以继续包含其它的 Component。
- 继承：用一个 Component 继承另外一个 Component。

九、Google 官方框架分析

下面是 Google 官方框架的目录结构：

作者：Alvin 老师
链接：https://www.jianshu.com/p/3dd77c1b678c
来源：简书
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

BAT交流群：892872246

可以看出，它把每个界面都作为一个独立的包（addedittask、statistics、taskdetail、tasks），而数据、依赖类是其它的两个包（data、util），我们先从 ToDoApplication 开始分析

```
public class ToDoApplication extends Application {
```

```
    private TasksRepositoryComponent mRepositoryComponent;



    @Override

    public void onCreate() {

        super.onCreate();

        mRepositoryComponent = DaggerTasksRepositoryComponent.builder()

                .applicationModule(new ApplicationModule((getApplicationC
ontext())))

                .build();

    }



    public TasksRepositoryComponent getTasksRepositoryComponent() {

        return mRepositoryComponent;

    }}
```

在 ToDoApplication 中，我们实例化了一个变量 TasksRepositoryComponent，它相当于是项目中所有其它 Component 的管理者，它被声明为@Singleton 的，即在 App 的生命周期中只存在一个，同时用@Component 表明它和 TaskRepositoyModule、ApplicationModule 这两个 Module 关联。

```
@Singleton@Component(modules = {TasksRepositoryModule.class, Application
Module.class})public interface TasksRepositoryComponent {

    TasksRepository getTasksRepository();}
```

`TaskRpositotyModule` 提供了两种类型的数据源对象，它们是用`@Local`、`@Remote` 来区分的：

```
@Modulepublic class TasksRepositoryModule {



    @Singleton

    @Provides
```

```
    @Local
    TasksDataSource provideTasksLocalDataSource(Context context) {
        return new TasksLocalDataSource(context);
    }

    @Singleton
    @Provides
    @Remote
    TasksDataSource provideTasksRemoteDataSource() {
        return new FakeTasksRemoteDataSource();
    }
}
```

接下来再看一下 `ApplicationModule`

```
@Modulepublic final class ApplicationModule {

    private final Context mContext;

    ApplicationModule(Context context) {
        mContext = context;
    }

    @Provides
    Context provideContext() {
```

```
        return mContext;

    }}
```

下面我们用一个比较简单的界面来看一下 `TasksRepositoryComponent` 是怎么和其它的 `Component` 关联起来的，首先看 `StatisticsActivity`：

```
public class StatisticsActivity extends AppCompatActivity {



    private DrawerLayout mDrawerLayout;



    @Inject

    StatisticsPresenter mStatiticsPresenter; //依靠 Dagger 实例化的对象。



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.statistics_act);

        //初始化 Fragment 界面。

        StatisticsFragment statisticsFragment = (StatisticsFragment) getS
upportFragmentManager().findFragmentById(R.id.contentFrame);

        if (statisticsFragment == null) {

            statisticsFragment = StatisticsFragment.newInstance();

            ActivityUtils.addFragmentToActivity(getSupportFragmentManager
(),

                    statisticsFragment, R.id.contentFrame);

        }
```

```
        DaggerStatisticsComponent.builder()

            .statisticsPresenterModule(new StatisticsPresenterModule(stat
isticsFragment))

            .tasksRepositoryComponent(((ToDoApplication) getApplication
()))

            .getTasksRepositoryComponent())

            .build().inject(this);

    }

}
```

第八章、Android 架构组件 Jetpack

8.1 LiveData 原理

前言

本篇文章主要讲解 LiveData 工作的原理，如果还不知道 LiveData 如何用的话，请参考官方文档。
LiveData 的讲解涉及到了 Lifecycle 的知识，如果你还不了解 LifeCycle，请参考文档 LifeCycle 介绍。

介绍

LiveData 是一个数据持有类，它可以通过添加观察者被其他组件观察其变更。不同于普通的观察者，它最重要的特性就是遵从应用程序的生命周期，如在 Activity 中如果数据更新了但 Activity 已经是 destroy 状态，LivaeData 就不会通知 Activity(observer)。当然。LiveData 的优点还有很多，如不会造成内存泄漏等。

LiveData 通常会配合 ViewModel 来使用，ViewModel 负责触发数据的更新，更新会通知到 LiveData，然后 LiveData 再通知活跃状态的观察者。

原理分析

下面直接看代码：

BAT交流群：892872246

```
public class UserProfileViewModel extends ViewModel {

    private String userId;

    private MutableLiveData<User> user;

    private UserRepository userRepo;


    public void init(String userId) {

        this.userId = userId;

        userRepo = new UserRepository();

        user = userRepo.getUser(userId);

    }


    public void refresh(String userId) {

        user = userRepo.getUser(userId);

    }


    public MutableLiveData<User> getUser() {

        return user;

    }

}
```

上面 UserProfileViewModel 内部持有 UserRepository 中 MutableLiveData<User> 的引用，并且提供了获取 MutableLiveData 的方法 getUser()，UserRepository 负责从网络或数据库中获取数据并封装成 MutableLiveData 然后提供给 ViewModel。

我们在 UserProfileFragment 中为 MutableLiveData<User> 注册观察者，如下：

BAT交流群：892872246

```
@Overridepublic void onActivityCreated(@Nullable Bundle savedInstanceSta
te) {

    super.onActivityCreated(savedInstanceState);

    String userId = getArguments().getString(UID_KEY);

    viewModel = ViewModelProviders.of(this).get(UserProfileViewModel.cla
ss);

    viewModel.init(userId);

    //标注 1

    viewModel.getUser().observe(UserProfileFragment.this, new Observer<U
ser>() {

        @Override

        public void onChanged(@Nullable User user) {

            if (user != null) {

                tvUser.setText(user.toString());

            }

        }

    });}
```

看标注 1 处，viewModel.getUser()获取到 MutableLiveData<User> 也就是我们的 LiveData，然后调用 LiveData 的 observer 方法，并把 UserProfileFragment 作为参数传递进去。observer() 方法就是我们分析的入口了，接下来我们看 LiveData 的 observer()方法都做了什么：

```
@MainThreadpublic void observe(@NonNull LifecycleOwner owner, @NonNull O
bserver<T> observer) {

    //标注 1

    if (owner.getLifecycle().getCurrentState() == DESTROYED) {

        // ignore

        return;
```

```
    }

    //标注 2

    LifecycleBoundObserver wrapper = new LifecycleBoundObserver(owner, o
bserver);

    ObserverWrapper existing = mObservers.putIfAbsent(observer, wrappe
r);

    if (existing != null && !existing.isAttachedTo(owner)) {

        throw new IllegalArgumentException("Cannot add the same observer"

                + " with different lifecycles");

    }

    if (existing != null) {

        return;

    }

    owner.getLifecycle().addObserver(wrapper);}
```

可以看到，UserProfileFragment 是作为 LifeCycleOwner 参数传进来的，如果你的 support 包版本大于等于 26.1.0，support 包中的 Fragment 会默认继承自 LifecycleOwner，而 LifecycleOwner 可获取到该组件的 LifeCycle，也就知道了 UserProfileFragment 组件的生命周期（在这里默认大家已经了解过 LifeCycle 了）。

看标注 1 处，如果我们的 UserProfileFragment 组件已经是 destroy 状态的话，将直接返回，不会被加入观察者行列。如果不是 destroy 状态，就到标注 2 处，新建一个 LifecycleBoundObserver 将我们的 LifecycleOwner 和 observer 保存起来，然后调用 mObservers.putIfAbsent(observer, wrapper) 将 observer 和 wrapper 分别作为 key 和 value 存入 Map 中，putIfAbsent()方法会判断如果 value 已经能够存在，就返回，否则返回 null。
如果返回 existing 为 null，说明以前没有添加过这个观察者，就将 LifecycleBoundObserver 作为 owner 生命周期的观察者，也就是作为 UserProfileFragment 生命周期的观察者。

我们看下 LifecycleBoundObserver 源码：

```
class LifecycleBoundObserver extends ObserverWrapper implements GenericL
ifecycleObserver {
```

交流群：892872246

```
    @NonNull final LifecycleOwner mOwner;



    LifecycleBoundObserver(@NonNull LifecycleOwner owner, Observer<T> ob
server) {

        super(observer);

        mOwner = owner;

    }



    @Override

    boolean shouldBeActive() {

        return mOwner.getLifecycle().getCurrentState().isAtLeast(STARTE
D);

    }



    @Override

    public void onStateChanged(LifecycleOwner source, Lifecycle.Event ev
ent) {

        if (mOwner.getLifecycle().getCurrentState() == DESTROYED) {

            removeObserver(mObserver);

            return;

        }

        activeStateChanged(shouldBeActive());

    }
```

```
    @Override

    boolean isAttachedTo(LifecycleOwner owner) {

        return mOwner == owner;

    }



    @Override

    void detachObserver() {

        mOwner.getLifecycle().removeObserver(this);

    }}
```

代码并不多，LifecycleBoundObserver 继承自 ObserverWrapper 并实现了 GenericLifecycleObserver 接口，而 GenericLifecycleObserver 接口又继承自 LifecycleObserver 接口，那么根据 Lifecycle 的特性，实现了 LifecycleObserver 接口并且加入 LifecycleOwner 的观察者里就可以感知或主动获取 LifecycleOwner 的状态。

好了，看完了观察者，那么我们的 LiveData 什么时候会通知观察者呢？不用想，肯定是数据更新的时候，而数据的更新是我们代码自己控制的，如请求网络返回 User 信息后，我们会主动将 User 放入 MutableLiveData 中，这里我在 UserRepository 中直接模拟网络请求如下：

```
public class UserRepository {

    final MutableLiveData<User> data = new MutableLiveData<>();



    public MutableLiveData<User> getUser(final String userId) {

        if ("xiasm".equals(userId)) {

            data.setValue(new User(userId, "夏胜明"));

        } else if ("123456".equals(userId)) {

            data.setValue(new User(userId, "哈哈哈"));
```

```
        } else {
            data.setValue(new User(userId, "unknow"));
        }
        return data;
    }}
```

当调用 getUser()方法的时候，我们调用 MutableLiveData 的 setValue()方法将数据放入 LiveData 中，这里 MutableLiveData 实际上就是继承自 LiveData，没有什么特别：

```
public class MutableLiveData<T> extends LiveData<T> {
    @Override
    public void postValue(T value) {
        super.postValue(value);
    }

    @Override
    public void setValue(T value) {
        super.setValue(value);
    }}
```

setValue()在放入 User 的时候必须在主线程，否则会报错，而 postValue 则没有这个检查，而是会把数据传入到主线程。我们直接看 setValue()方法：

```
@MainThreadprotected void setValue(T value) {
    assertMainThread("setValue");
    mVersion++;
    mData = value;
```

```
    dispatchingValue(null);}
```

首先调用 assertMainThread()检查是否在主线程，接着将要更新的数据赋给 mData，然后调用 dispatchingValue()方法并传入 null，将数据分发给各个观察者，如我们的 UserProfileFragment。看 dispatchingValue()方法实现：

```
private void dispatchingValue(@Nullable ObserverWrapper initiator) {

    if (mDispatchingValue) {

        mDispatchInvalidated = true;

        return;

    }

    mDispatchingValue = true;

    do {

        mDispatchInvalidated = false;

        //标注 1

        if (initiator != null) {

            considerNotify(initiator);

            initiator = null;

        } else {

            //标注 2

            for (Iterator<Map.Entry<Observer<T>, ObserverWrapper>> iterat
or =

                    mObservers.iteratorWithAdditions(); iterator.hasNext
(); ) {

                considerNotify(iterator.next().getValue());

                if (mDispatchInvalidated) {

                    break;
```

```
            }

        }

    }

} while (mDispatchInvalidated);

mDispatchingValue = false;}
```

从标注 1 可以看出，dispatchingValue()参数传 null 和不传 null 的区别就是如果传 null 将会通知所有的观察者，反之仅仅通知传入的观察者。我们直接看标注 2，通知所有的观察者通过遍历 mObservers ，将所有的 ObserverWrapper 拿到，实际上就是我们上面提到的 LifecycleBoundObserver，通知观察者调用 considerNotify()方法，这个方法就是通知的具体实现了。

```
private void considerNotify(ObserverWrapper observer) {

    if (!observer.mActive) {

        return;

    }

    // Check latest state b4 dispatch. Maybe it changed state but we didn
't get the event yet.

    //

    // we still first check observer.active to keep it as the entrance fo
r events. So even if

    // the observer moved to an active state, if we've not received that
event, we better not

    // notify for a more predictable notification order.

    if (!observer.shouldBeActive()) {

        observer.activeStateChanged(false);

        return;

    }

    if (observer.mLastVersion >= mVersion) {
```

```
        return;

    }

    observer.mLastVersion = mVersion;

    //noinspection unchecked

    observer.mObserver.onChanged((T) mData);}
```

如果观察者不是活跃状态，将不会通知此观察者，看最后一行，observer.mObserver.onChanged((T) mData)，observer.mObserver 就是我们调用 LiveData 的 observer()方法传入的 Observer，然后调用 Observer 的 onChanged((T) mData)方法，将保存的数据 mData 传入，也就实现了更新。在看下我们实现的 Observer：

```
viewModel.getUser().observe(UserProfileFragment.this, new Observer<User>
() {

    @Override

    public void onChanged(@Nullable User user) {

        if (user != null) {

            tvUser.setText(user.toString());

        }

    }});
```

如果哪个控件要根据 user 的变更而及时更新，就在 onChanged()方法里处理就可以了。到这里，LiveData 已经能够分析完了，其实 LiveData 的实现还是要依赖于 Lifecycle。

8.2 Navigation 如何解决 tabLayout 问题

8.3 ViewModel 如何感知 View 生命周期及内核原理

概述

BAT交流群：892872246

ViewModel，从字面上理解的话，它肯定是跟视图(View)以及数据(Model)相关的。正像它字面意思一样，它是负责准备和管理和 UI 组件(Fragment/Activity)相关的数据类，也就是说 ViewModel 是用来管理 UI 相关的数据的，同时 ViewModel 还可以用来负责 UI 组件间的通信。

之前存在的问题

ViewModel 用来存储和管理 UI 相关的数据，可于将一个 Activity 或 Fragment 组件相关的数据逻辑抽象出来，并能适配组件的生命周期，如当屏幕旋转 Activity 重建后，ViewModel 中的数据依然有效。

引入 ViewModel 之前，存在如下几个问题：

通常 Android 系统来管理 UI controllers（如 Activity、Fragment）的生命周期，由系统响应用户交互或者重建组件，用户无法操控。当组件被销毁并重建后，原来组件相关的数据也会丢失，如果数据类型比较简单，同时数据量也不大，可以通过 onSaveInstanceState()存储数据，组件重建之后通过 onCreate()，从中读取 Bundle 恢复数据。但如果是大量数据，不方便序列化及反序列化，则上述方法将不适用。
UI controllers 经常会发送很多异步请求，有可能会出现 UI 组件已销毁，而请求还未返回的情况，因此 UI controllers 需要做额外的工作以防止内存泄露。
当 Activity 因为配置变化而销毁重建时，一般数据会重新请求，其实这是一种浪费，最好就是能够保留上次的数据。
UI controllers 其实只需要负责展示 UI 数据、响应用户交互和系统交互即可。但往往开发者会在 Activity 或 Fragment 中写许多数据请求和处理的工作，造成 UI controllers 类代码膨胀，也会导致单元测试难以进行。我们应该遵循职责分离原则，将数据相关的事情从 UI controllers 中分离出来。

ViewModel 基本使用

```
public class MyViewModel extends ViewModel {

    private MutableLiveData<List<User>> users;

    public LiveData<List<User>> getUsers() {

        if (users == null) {

            users = new MutableLiveData<List<Users>>();

            loadUsers();
```

```
        }

        return users;

    }



    private void loadUsers() {

        // 异步调用获取用户列表

    }}
```

新的 Activity 如下：

```
public class MyActivity extends AppCompatActivity {

    public void onCreate(Bundle savedInstanceState) {

        MyViewModel model = ViewModelProviders.of(this).get(MyViewModel.
class);

        model.getUsers().observe(this, users -> {

            // 更新 UI

        });

    }}
```

如果 Activity 被重新创建了，它会收到被之前 Activity 创建的相同 MyViewModel 实例。当所属 Activity 终止后，框架调用 ViewModel 的 onCleared()方法清除资源。

因为 ViewModel 在指定的 Activity 或 Fragment 实例外存活，它应该永远不能引用一个 View，或持有任何包含 Activity context 引用的类。如果 ViewModel 需要 Application 的 context（如获取系统服务），可以扩展 AndroidViewmodel，并拥有一个构造器接收 Application。

在 Fragment 间共享数据

一个 Activity 中的多个 Fragment 相互通讯是很常见的。之前每个 Fragment 需要定义接口描述，所属 Activity 将二者捆绑在一起。此外，每个 Fragment 必须处理

BA技术群：892872246

其他 Fragment 未创建或不可见的情况。通过使用 ViewModel 可以解决这个痛点，这些 Fragment 可以使用它们的 Activity 共享 ViewModel 来处理通讯：

```
public class SharedViewModel extends ViewModel {

    private final MutableLiveData<Item> selected = new MutableLiveData<I
tem>();



    public void select(Item item) {

        selected.setValue(item);

    }



    public LiveData<Item> getSelected() {

        return selected;

    }}

public class MasterFragment extends Fragment {

    private SharedViewModel model;

    public void onActivityCreated() {

        model = ViewModelProviders.of(getActivity()).get(SharedViewModel.
class);

        itemSelector.setOnClickListener(item -> {

            model.select(item);

        });

    }}

public class DetailFragment extends LifecycleFragment {

    public void onActivityCreated() {
```

```
        SharedViewModel model = ViewModelProviders.of(getActivity()).get
(SharedViewModel.class);

        model.getSelected().observe(this, { item ->

            // update UI

        });

    }}
```

注意:上面两个 Fragment 都用到了如下代码来获取 ViewModel，getActivity()返回的是同一个宿主 Activity，因此两个 Fragment 之间返回的是同一个 SharedViewModel 对象。

```
SharedViewModel model = ViewModelProviders.of(getActivity()).get(SharedV
iewModel.class);
```

这种方式的好处包括：

1.Activity 不需要做任何事情，也不需要知道通讯的事情
2.Fragment 不需要知道彼此，除了 SharedViewModel 进行联系。如果它们(Fragment)其中一个消失了，其余的仍然能够像往常一样工作。
3.每个 Fragment 有自己的生命周期，而且不会受其它 Fragment 生命周期的影响。事实上，一个 Fragment 替换另一个 Fragment，UI 的工作也不会受到任何影响。

ViewModel 的生命周期

ViewModel 对象的范围由获取 ViewModel 时传递至 ViewModelProvider 的 Lifecycle 所决定。ViewModel 始终处在内存中，直到 Lifecycle 永久地离开—对于 Activity 来说，是当它终止（finish）的时候，对于 Fragment 来说，是当它分离（detached）的时候。

BAT交流群：892872246

上图左侧为 Activity 的生命周期过程，期间有一个旋转屏幕的操作；右侧则为 ViewModel 的生命周期过程。

一般通过如下代码初始化 ViewModel：

```
viewModel = ViewModelProviders.of(this).get(UserProfileViewModel.class);
```

this 参数一般为 Activity 或 Fragment，因此 ViewModelProvider 可以获取组件的生命周期。

Activity 在生命周期中可能会触发多次 onCreate()，而 ViewModel 则只会在第一次 onCreate()时创建，然后直到最后 Activity 销毁。

ViewModel 相关类图

BAT交流群：892872246

ViewModelProviders 是 ViewModel 工具类，该类提供了通过 Fragment 和 Activity 得到 ViewModel 的方法，而具体实现又是由 ViewModelProvider 实现的。

ViewModelProvider 是实现 ViewModel 创建、获取的工具类。在 ViewModelProvider 中定义了一个创建 ViewModel 的接口类——Factory。ViewModelProvider 中有个 ViewModelStore 对象，用于存储 ViewModel 对象。

ViewModelStore 是存储 ViewModel 的类，具体实现是通过 HashMap 来保存 ViewModle 对象。

ViewModel 是个抽象类，里面只定义了一个 onCleared()方法，该方法在 ViewModel 不在被使用时调用。ViewModel 有一个子类 AndroidViewModel，这个类是便于要在 ViewModel 中使用 Context 对象，因为我们前面提到不能在 ViewModel 中持有 Activity 的引用。

ViewModelStores 是 ViewModelStore 的工厂方法类，它会关联 HolderFragment，HolderFragment 有个嵌套类——HolderFragmentManager。

ViewModel 相关时序图

追溯创建一个 ViewModel 的源码，会察觉需要的步骤有点多。下面以在 Fragment 中得到 ViewModel 对象为例看下整个过程的时序图。

BAT交流群：892872246

时序图看起来比较复杂，但是它只描述了两个过程：

得到 ViewModel 对象。
HolderFragment 被销毁时，ViewModel 收到 onCleared()通知。

ViewModel 相关源码分析

ViewModelProviders 类的具体实现：

```
public class ViewModelProviders {



    private static Application checkApplication(Activity activity) {

        Application application = activity.getApplication();

        if (application == null) {
```

```
            throw new IllegalStateException("Your activity/fragment is no
t yet attached to "

                    + "Application. You can't request ViewModel before onC
reate call.");

        }

        return application;

    }



    private static Activity checkActivity(Fragment fragment) {

        Activity activity = fragment.getActivity();

        if (activity == null) {

            throw new IllegalStateException("Can't create ViewModelProvid
er for detached fragment");

        }

        return activity;

    }



    @NonNull

    @MainThread

    public static ViewModelProvider of(@NonNull Fragment fragment) {

        ViewModelProvider.AndroidViewModelFactory factory =

                ViewModelProvider.AndroidViewModelFactory.getInstance(

                        checkApplication(checkActivity(fragment)));

        return new ViewModelProvider(ViewModelStores.of(fragment), facto
ry);
```

```
    }



    @NonNull

    @MainThread

    public static ViewModelProvider of(@NonNull FragmentActivity activit
y) {

        ViewModelProvider.AndroidViewModelFactory factory =

                ViewModelProvider.AndroidViewModelFactory.getInstance(

                        checkApplication(activity));

        return new ViewModelProvider(ViewModelStores.of(activity), facto
ry);

    }



    @NonNull

    @MainThread

    public static ViewModelProvider of(@NonNull Fragment fragment, @NonN
ull Factory factory) {

        checkApplication(checkActivity(fragment));

        return new ViewModelProvider(ViewModelStores.of(fragment), facto
ry);

    }



    @NonNull

    @MainThread
```

BAT文 /2246

```
    public static ViewModelProvider of(@NonNull FragmentActivity activit
y,

            @NonNull Factory factory) {

        checkApplication(activity);

        return new ViewModelProvider(ViewModelStores.of(activity), facto
ry);

    }
```

ViewModelProviders 提供了四个 of()方法，四个方法功能类似，其中 of(FragmentActivity activity, Factory factory)和 of(Fragment fragment, Factory factory)提供了自定义创建 ViewModel 的方法。

1. 判断 Fragment 的是否 Attached to Activity，Activity 的 Application 对象是否为空。
2. 创建 ViewModel 对象看似很简单，一行代码搞定。

```
new ViewModelProvider(ViewModelStores.of(fragment), factory)
```

先看看 ViewModelStores.of()方法:

```
    @NonNull

    @MainThread

    public static ViewModelStore of(@NonNull Fragment fragment) {

        if (fragment instanceof ViewModelStoreOwner) {

            return ((ViewModelStoreOwner) fragment).getViewModelStore();

        }

        return holderFragmentFor(fragment).getViewModelStore();

    }
```

继续深入发现其实是实现了一个接口：

```
public interface ViewModelStoreOwner {

    @NonNull
```

```
    ViewModelStore getViewModelStore();}
```

holderFragmentFor()是 HolderFragment 的静态方法，HolderFragment 继承自 Fragment。我们先看 holderFragment()方法的具体实现

```
    @RestrictTo(RestrictTo.Scope.LIBRARY_GROUP)

    public static HolderFragment holderFragmentFor(Fragment fragment) {

        return sHolderFragmentManager.holderFragmentFor(fragment);

    }



    @NonNull

    @Override

    public ViewModelStore getViewModelStore() {

        return mViewModelStore;

    }
```

继续看 HolderFragmentManager.holderFragmentFor()方法的具体实现

```
HolderFragment holderFragmentFor(Fragment parentFragment) {

            FragmentManager fm = parentFragment.getChildFragmentManager
();

            HolderFragment holder = findHolderFragment(fm);

            if (holder != null) {

                return holder;

            }

            holder = mNotCommittedFragmentHolders.get(parentFragment);
```

BA 892246

```
            if (holder != null) {
                return holder;
            }


            parentFragment.getFragmentManager()
                    .registerFragmentLifecycleCallbacks(mParentDestroyedC
allback, false);
            holder = createHolderFragment(fm);
            mNotCommittedFragmentHolders.put(parentFragment, holder);
            return holder;
        }
private FragmentLifecycleCallbacks mParentDestroyedCallback =
    new FragmentLifecycleCallbacks() {
        @Override
        public void onFragmentDestroyed(FragmentManager fm, Fragment pare
ntFragment) {
            super.onFragmentDestroyed(fm, parentFragment);
            HolderFragment fragment = mNotCommittedFragmentHolders.remove
(
                    parentFragment);
            if (fragment != null) {
                Log.e(LOG_TAG, "Failed to save a ViewModel for " + parentF
ragment);
            }
        }
```

```
    };

private static HolderFragment createHolderFragment(FragmentManager fragm
entManager) {

    HolderFragment holder = new HolderFragment(); // 创建 HolderFragment
对象

    fragmentManager.beginTransaction().add(holder, HOLDER_TAG).commitAll
owingStateLoss();

    return holder;}

public HolderFragment() {

    //这个是关键，这就使得 Activity 被 recreate 时，Fragment 的 onDestroy()和 o
nCreate()不会被调用

    setRetainInstance(true); }
```

setRetainInstance(boolean) 是 Fragment 中的一个方法。将这个方法设置为 true 就可以使当前 Fragment 在 Activity 重建时存活下来，如果不设置或者设置为 false，当前 Fragment 会在 Activity 重建时同样发生重建，以至于被新建的对象所替代。

在 setRetainInstance(boolean)为 true 的 Fragment 中放一个专门用于存储 ViewModel 的 Map，自然 Map 中所有的 ViewModel 都会幸免于 Activity 重建，让 Activity, Fragment 都绑定一个这样的 Fragment，将 ViewModel 存放到这个 Fragment 的 Map 中, ViewModel 组件就这样实现了。

到此为止，我们已经得到了 ViewStore 对象，前面我们在创建 ViewModelProvider 对象是通过这行代码实现的 new ViewModelProvider(ViewModelStores.of(fragment), sDefaultFactory)现在再看下 ViewModelProvider 的构造方法

```
public ViewModelProvider(@NonNull ViewModelStore store, @NonNull Factory
 factory) {

        mFactory = factory;

        this.mViewModelStore = store;

    }
```

BAT交 892246

```
public ViewModelProvider(@NonNull ViewModelStore store, @NonNull Factory
 factory) {

        mFactory = factory;

        this.mViewModelStore = store;

    }
```

现在就可以通过 ViewModelProvider.get()方法得到 ViewModel 对象，继续看下该方法的具体实现

```
    @NonNull

    @MainThread

    public <T extends ViewModel> T get(@NonNull String key, @NonNull Clas
s<T> modelClass) {

        ViewModel viewModel = mViewModelStore.get(key); //从缓存中查找是否
有已有 ViewModel 对象。



        if (modelClass.isInstance(viewModel)) {

            //noinspection unchecked

            return (T) viewModel;

        } else {

            //noinspection StatementWithEmptyBody

            if (viewModel != null) {

                // TODO: log a warning.

            }

        }



        viewModel = mFactory.create(modelClass); //创建 ViewModel 对象，然后
缓存起来。
```

```
        mViewModelStore.put(key, viewModel);

        //noinspection unchecked

        return (T) viewModel;

    }

    @NonNull

    @MainThread

    public <T extends ViewModel> T get(@NonNull String key, @NonNull Clas
s<T> modelClass) {

        ViewModel viewModel = mViewModelStore.get(key); //从缓存中查找是否
有已有 ViewModel 对象。



        if (modelClass.isInstance(viewModel)) {

            //noinspection unchecked

            return (T) viewModel;

        } else {

            //noinspection StatementWithEmptyBody

            if (viewModel != null) {

                // TODO: log a warning.

            }

        }



        viewModel = mFactory.create(modelClass); //创建 ViewModel 对象，然后
缓存起来。

        mViewModelStore.put(key, viewModel);
```

```
        //noinspection unchecked

        return (T) viewModel;

    }

    @NonNull

    @MainThread

    public <T extends ViewModel> T get(@NonNull String key, @NonNull Clas
s<T> modelClass) {

        ViewModel viewModel = mViewModelStore.get(key); //从缓存中查找是否
有已有 ViewModel 对象。



        if (modelClass.isInstance(viewModel)) {

            //noinspection unchecked

            return (T) viewModel;

        } else {

            //noinspection StatementWithEmptyBody

            if (viewModel != null) {

                // TODO: log a warning.

            }

        }



        viewModel = mFactory.create(modelClass); //创建 ViewModel 对象，然后
缓存起来。

        mViewModelStore.put(key, viewModel);

        //noinspection unchecked
```

```
        return (T) viewModel;

    }
```

ViewModelProvider.get()方法比较简单，注释中都写明了。最后我们看下ViewModelStore 类的具体实现

```
public class ViewModelStore {



    private final HashMap<String, ViewModel> mMap = new HashMap<>();



    final void put(String key, ViewModel viewModel) {

        ViewModel oldViewModel = mMap.get(key);

        if (oldViewModel != null) {

            oldViewModel.onCleared();

        }

        mMap.put(key, viewModel);

    }



    final ViewModel get(String key) {

        return mMap.get(key);

    }



    public final void clear() {

        for (ViewModel vm : mMap.values()) {
```

```
            vm.onCleared();

        }

        mMap.clear();

    }}
```

ViewModelStore 是缓存 ViewModel 的类，put()、get()方法用于存取 ViewModel 对象，另外提供了 clear()方法用于清空缓存的 ViewModel 对象，在该方法中会调用 ViewModel.onCleared()方法通知 ViewModel 对象不再被使用。

ViewModel 收到 onCleared()通知

HolderFragment 的 onDestroy()方法

```
    @Override

    public void onDestroy() {

        super.onDestroy();

        mViewModelStore.clear();

    }
```

在 onDestroy()方法中调用了 ViewModelStore.clear()方法，我们知道在该方法中会调用 ViewModel 的 onCleared()方法。在你看了 HolderFragment 源码后，或许你会有个疑问，mViewModelStore 保存的 ViewModel 对象是在哪里添加的呢？ 细心的话，你会发现在 ViewModelProvider 的构造方法中，已经将 HolderFragment 中的 ViwModelStore 对象 mViewModelStore 的引用传递给了 ViewModelProvider 中的 mViewModelStore，而在 ViewModelProvider.get()方法中会向 mViewModelStore 添加 ViewModel 对象。

总结

ViewModel 职责是为 Activity 或 Fragment 管理、请求数据，具体数据请求逻辑不应该写在 ViewModel 中，否则 ViewModel 的职责会变得太重，此处需要一个引入一个 Repository，负责数据请求相关工作。具体请参考 Android 架构组件。
ViewModel 可以用于 Activity 内不同 Fragment 的交互，也可以用作 Fragment 之间一种解耦方式。
ViewModel 也可以负责处理部分 Activity/Fragment 与应用其他模块的交互。

BAT交流群：892872246

ViewModel 生命周期（以 Activity 为例）起始于 Activity 第一次 onCreate()，结束于 Activity 最终 finish 时。

8.4 Room 架构方式方法

一、简介

Room 是 Google 推出的 Android 架构组件库中的数据持久化组件库，也可以说是在 SQLite 上实现的一套 ORM 解决方案。Room 主要包含三个部分：

- **Database**：持有 DB 和 DAO
- **Entity**：定义 POJO 类，即数据表结构
- **DAO**(Data Access Objects)：定义访问数据（增删改查）的接口

其关系如下图所示：

二、基本使用

1. 创建 Entity

1.1 一个简单的 Entitiy

一个简单 Entity 定义如下：

```
@Entity(tableName = "user"

        indices = {@Index(value = {"first_name", "last_name"})})public
class User {



    @PrimaryKey

    private int uid;



    @ColumnInfo(name = "first_name")

    private String firstName;



    @ColumnInfo(name = "last_name")

    private String lastName;



    @Ignore

    public User(String firstName, String lastName) {

        this.uid = UUID.randomUUID().toString();

        this.firstName = firstName;

        this. lastName = lastName;

    }
```

892872246

```
    public User(String id, String firstName, String lastName) {

        this.uid = id;

        this.firstName = userName;

        this. lastName = userName;

    }



    // Getters and setters}
```

- `@Entity(tableName = "table_name**")` 注解 POJO 类，定义数据表名称;
- `@PrimaryKey` 定义主键，如果一个 Entity 使用的是复合主键，可以通过`@Entity` 注解的 `primaryKeys` 属性定义复合主键：`@Entity(primaryKeys = {"firstName", "lastName"})`
- `@ColumnInfo(name = “column_name”)` 定义数据表中的字段名
- `@Ignore` 用于告诉 Room 需要忽略的字段或方法
- 建立索引：在`@Entity` 注解的 `indices` 属性中添加索引字段。例如：`indices = {@Index(value = {"first_name", "last_name"}, unique = true), ...}`，`unique = true` 可以确保表中不会出现`{"first_name", "last_name"}` 相同的数据。

1.2 Entitiy 间的关系

不同于目前存在的大多数 ORM 库，Room 不支持 Entitiy 对象间的直接引用。（具体原因可以参考： Understand why Room doesn't allow object references）
但 Room 允许通过**外键(Foreign Key)**来表示 Entity 之间的关系

```
@Entity(foreignKeys = @ForeignKey(entity = User.class,

                                  parentColumns = "id",

                                  childColumns = "user_id"))class Book {

    @PrimaryKey

    public int bookId;



    public String title;
```

```
    @ColumnInfo(name = "user_id")

    public int userId;}
```

如上面代码所示，Book 对象与 User 对象是属于的关系。Book 中的 user_id,对应 User 中的 id。 那么当一个 User 对象被删除时， 对应的 Book 会发生什么呢？

`@ForeignKey` 注解中有两个属性 `onDelete` 和 `onUpdate`， 这两个属性对应 `ForeignKey` 中的 `onDelete()` 和 `onUpdate()`，通过这两个属性的值来设置当 User 对象被删除 / 更新时，Book 对象作出的响应。这两个属性的可选值如下：

- `CASCADE`：User 删除时对应 Book 一同删除； 更新时，关联的字段一同更新
- `NO_ACTION`：User 删除时不做任何响应
- `RESTRICT`：禁止 User 的删除 / 更新。当 User 删除或更新时，Sqlite 会立马报错。
- `SET_NULL`：当 User 删除时， Book 中的 userId 会设为 NULL
- `SET_DEFAULT`：与 `SET_NULL` 类似，当 User 删除时，Book 中的 userId 会设为默认值

1.3 对象嵌套

在某些情况下， 对于一张表中的数据我们会用多个 POJO 类来表示，在这种情况下可以用`@Embedded` 注解嵌套的对象，比如：

```
class Address {

    public String street;

    public String state;

    public String city;



    @ColumnInfo(name = "post_code")

    public int postCode;}

@Entityclass User {

    @PrimaryKey

    public int id;
```

BA 2246

```
    public String firstName;

    @Embedded
    public Address address;}
```

以上代码所产生的 User 表中，Column 为 `id, firstName, street, state, city, post_code`

2 创建数据访问对象（DAO）

```
@Daopublic interface UserDao {
    @Query("SELECT * FROM user")
    List<User> getAll();

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    List<User> loadAllByIds(int[] userIds);

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND "
           + "last_name LIKE :last LIMIT 1")
    User findByName(String first, String last);

    @Insert
    void insertAll(List<User> users);
```

```
    @Insert(onConflict = OnConflictStrategy.REPLACE)

    public void insertUsers(User... users);



    @Delete

    void delete(User user);



    @Update

    public void updateUsers(List<User> users);}
```

DAO 可以是一个接口，也可以是一个抽象类， Room 会在编译时创建 DAO 的实现。

Tips:

- `@Insert` 方法也可以定义返回值， 当传入参数仅有一个时返回 `long`，传入多个时返回 `long[]`或 `List<Long>`，Room 在实现 insert 方法的实现时会在一个事务进行所有参数的插入。
- `@Insert` 的参数存在冲突时， 可以设置 `onConflict` 属性的值来定义冲突的解决策略， 比如代码中定义的是 `@Insert(onConflict = OnConflictStrategy.REPLACE)`，即发生冲突时替换原有数据
- `@Update` 和`@Delete` 可以定义 `int` 类型返回值，指更新 / 删除的函数

DAO 中的增删改方法的定义都比较简单，这里不展开讨论，下面更多的聊一下查询方法。

2.1 简单的查询

Talk is cheap，直接 show code:

```
@Query("SELECT * FROM user")

List<User> getAll();
```

Room 会在编译时校验 sql 语句，如果`@Query()` 中的 sql 语句存在语法错误，或者查询的表不存在，Room 会在编译时报错。

BAI交流群：892872246

2.2 查询参数传递

```
@Query("SELECT * FROM user WHERE uid IN (:userIds)")

List<User> loadAllByIds(int[] userIds);

@Query("SELECT * FROM user WHERE first_name LIKE :first AND "

          + "last_name LIKE :last LIMIT 1")

User findByName(String first, String last);
```

看代码应该比较好理解， 方法中传递参数 `arg`，在 sql 语句中用`:arg` 即可。编译时 Room 会匹配对应的参数。

如果在传参中没有匹配到:arg 对应的参数，Room 会在编译时报错。

2.3 查询表中部分字段的信息

在实际某个业务场景中， 我们可能仅关心一个表部分字段的值，这时我仅需要查询关心的列即可。

定义子集的 POJO 类:

```
public class NameTuple {

    @ColumnInfo(name="first_name")

    public String firstName;



    @ColumnInfo(name="last_name")

    public String lastName;}

public class NameTuple {

    @ColumnInfo(name="first_name")

    public String firstName;



    @ColumnInfo(name="last_name")
```

```
    public String lastName;}
```

在 DAO 中添加查询方法

```
@Query("SELECT first_name, last_name FROM user")public List<NameTuple> l
oadFullName();
```

这里定义的 POJO 也支持使用@Embedded

2.4 查询结果的返回类型

Room 中查询操作除了返回 POJO 对象及其 List 以外， 还支持：

- LiveData<T>:
  LiveData 是架构组件库中提供的另一个组件，可以很好满足数据变化驱动 UI 刷新的需求。Room 会实现更新 LiveData 的代码。

```
@Query("SELECT first_name, last_name FROM user WHERE region IN (:regions)
")

public LiveData<List<User>> loadUsersFromRegionsSync(List<String> region
s);
```

Flowablbe<T> Maybe<T> Single<T>:
Room 支持返回 RxJava2 的 Flowablbe, Maybe 和 Single 对象，对于使用 RxJava 的项目可以很好的衔接， 但需要在 gradle 添加该依赖：
android.arch.persistence.room:rxjava2。

```
@Query("SELECT * from user where id = :id LIMIT 1")public Flowable<User>
 loadUserById(int id);
```

- Cursor:
  返回 Cursor 是为了支持现有项目中使用 Cursor 的场景，官方不建议直接返回 Cursor.

  Caution: It's highly discouraged to work with the Cursor API because it doesn't guarantee whether the rows exist or what values the rows contain. Use this functionality only if you already have code that expects a cursor and that you can't refactor easily.

2.5 联表查询

Room 支持联表查询，接口定义上与其他查询差别不大， 主要还是 sql 语句的差别。

BAT交流群：892872246

```
@Daopublic interface MyDao {

    @Query("SELECT * FROM book "

            + "INNER JOIN loan ON loan.book_id = book.id "

            + "INNER JOIN user ON user.id = loan.user_id "

            + "WHERE user.name LIKE :userName")

   public List<Book> findBooksBorrowedByNameSync(String userName);}
```

3. 创建数据库

Room 中 DataBase 类似 SQLite API 中 SQLiteOpenHelper，是提供 DB 操作的切入点，但是除了持有 DB 外， 它还负责持有相关数据表（Entity）的数据访问对象（DAO），所以 Room 中定义 Database 需要满足三个条件：

- 继承 RoomDataBase，并且是一个抽象类
- 用@Database 注解，并定义相关的 entity 对象， 当然还有必不可少的数据库版本信息
- 定义返回 DAO 对象的抽象方法

```
@Database(entities = {User.class}, version = 1)public abstract class App
Database extends RoomDatabase {

    public abstract UserDao userDao();}
```

- 创建好以上 Room 的三大组件后， 在代码中就可以通过以下代码创建 Database 实例。

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),

        AppDatabase.class, "database-name").build();
```

三、数据库迁移

BAT交流群：892872246

3.1 Room 数据库升级

在传统的 SQLite API 中，我们如果要升级数据库， 通常在 `SQLiteOpenHelper.onUpgrade` 方法执行数据库升级的 sql 语句，这些 sql 语句的通常根据数据库版本以文件的方式或者用数组来管理。有人说这种方式升级数据库就像在拆炸弹，相比之下在 Room 中升级数据库简单的就像是按一个开关而已。

Room 提供了 Migration 类来实现数据库的升级:

```
Room.databaseBuilder(getApplicationContext(), MyDb.class, "database-name
")

        .addMigrations(MIGRATION_1_2, MIGRATION_2_3).build();

static final Migration MIGRATION_1_2 = new Migration(1, 2) {

    @Override

    public void migrate(SupportSQLiteDatabase database) {

        database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, "

                + "`name` TEXT, PRIMARY KEY(`id`))");

    }};

static final Migration MIGRATION_2_3 = new Migration(2, 3) {

    @Override

    public void migrate(SupportSQLiteDatabase database) {

        database.execSQL("ALTER TABLE Book "

                + " ADD COLUMN pub_year INTEGER");

    }};
```

在创建 Migration 类时需要指定 `startVersion` 和 `endVersion`，代码中 `MIGRATION_1_2` 和 `MIGRATION_2_3` 的 startVersion 和 endVersion 是递增的，Migration 其实是支持从版本 1 直接升到版本 3，只要其 `migrate()` 方法里执行的语句正常即可。那么 Room 是怎么实现数据库升级的呢？其实本质上还是调用 `SQLiteOpenHelper.onUpgrade`，Room 中自己实现了一个 `SQLiteOpenHelper`， 在

onUpgrade()方法被调用时触发 Migration，当第一次访问数据库时，Room 做了以下几件事：

- 创建 Room Database 实例
- SQLiteOpenHelper.onUpgrade 被调用，并且触发 Migration
- 打开数据库

这样一看， Room 中处理数据库升级确实很像是加一个开关。

3.2 原有 SQLite 数据库迁移至 Room

因为 Room 使用的也是 SQLite， 所以可以很好的支持原有 Sqlite 数据库迁移到 Room。

假设原有一个版本号为 1 的数据库有一张表 User，现在要迁移到 Room， 我们需要定义好 Entity, DAO, Database，然后创建 Database 时添加一个空实现的 Migraton 即可。需要注意的是，即使对数据库没有任何升级操作，也需要升级版本， 否则会抛异常 IllegalStateException.

```
@Database(entities = {User.class}, version = 2)public abstract class Use
rsDatabase extends RoomDatabase {

…

static final Migration MIGRATION_1_2 = new Migration(1, 2) {

    @Override

    public void migrate(SupportSQLiteDatabase database) {

        // Since we didn't alter the table, there's nothing else to do he
re.

    }};

…

database =  Room.databaseBuilder(context.getApplicationContext(),

        UsersDatabase.class, "Sample.db")

        .addMigrations(MIGRATION_1_2)

        .build();
```

四、复杂数据的处理

在某些场景下我们的应用可能需要存储复杂的数据类型，比如 `Date`，但是 Room 的 Entity 仅支持基本数据类型和其装箱类之间的转换，不支持其它的对象引用。所以 Room 提供了 `TypeConverter` 给使用者自己实现对应的转换。

一个 `Date` 类型的转换如下：

```
public class Converters {

    @TypeConverter

    public static Date fromTimestamp(Long value) {

        return value == null ? null : new Date(value);

    }



    @TypeConverter

    public static Long dateToTimestamp(Date date) {

        return date == null ? null : date.getTime();

    }}
```

定义好转换方法后，指定到对应的 Database 上即可，这样就可以在对应的 POJO（User）中使用 `Date` 类了。

```
@Database(entities = {User.class}, version = 1)@TypeConverters({Converte
rs.class})public abstract class AppDatabase extends RoomDatabase {

    public abstract UserDao userDao();}
```

五、总结

在 SQLite API 方式实现数据持久化的项目中，相信都有一个任务繁重的 `SQLiteOpenHelper` 实现，一堆维护表的字段的 `Constant` 类，一堆代码类似的数

BAT交流群：892872246

据库访问类（DAO），访问数据库时需要做 Cursor 的遍历，构建并返回对应的 POJO 类...相比之下，Room 作为在 SQLite 之上封装的 ORM 库确实有诸多优势，比较直观的体验是：

- 比 SQLite API 更简单的使用方式
- 省略了许多重复代码
- 能在编译时校验 sql 语句的正确性
- 数据库相关的代码分为 Entity, DAO, Database 三个部分，结构清晰
- 简单安全的数据库升级方案

想要了解更多 Room 相关内容可以戳下面的链接：

- Google Sample https://github.com/googlesamples/android-architecture-components
- Room 数据库迁移 https://medium.com/google-developers/understanding-migrations-with-room-f01e04b07929
- Room 使用引导说明 https://medium.com/google-developers/7-steps-to-room-27a5fe5f99b2
- Room https://medium.com/google-developers/room-rxjava-acb0cd4f3757
- 7 Pro-tips for Room https://medium.com/google-developers/7-pro-tips-for-room-fbadea4bfbd1

8.5 dataBinding 为什么能够支持 MVVM

前言：

2015 年谷歌 I/O 大会上介绍了一个数据绑定框架 DataBinding。2016 年，2017 年毫无意外成了项目实战中主流框架。使用它我们可以轻松实现 MVVM（模型-视图-视图模型）模式，来实现应用之间数据与视图的分离、视图与业务逻辑的分离、数据与业务逻辑的分离，从而达到低耦合、可重用性、易测试性等好处。而使用 DataBinding 不仅减少了 findViewById 的出现频率，而且还大大提高解析 XML 的速度。

1．基本使用

新建一个项目，在 app 的 build 文件加上：

```
android {

...

dataBinding{
```

892872246

```
        enabled =true;

    }

    ...

}
```

构建环境特别简单，接下来直接开始使用数据绑定

首先，新建一个 User.java 实体类：

```
public class User {

    public String name;

    public String myBlog;

    public int age;


    public User(String name,int age,String myBlog){

        this.name=name;

        this.age=age;

        this.myBlog=myBlog;

    }


    public String getName() {

        return name;

    }


    public void setName(String name) {

        this.name = name;
```

```
    }


    public String getMyBlog() {

        return myBlog;

    }


    public void setMyBlog(String myBlog) {

        this.myBlog = myBlog;

    }}
```

之后看下布局(activity_basic.xml)，跟传统的布局不一样，这里需要使用<layout></layout>作为根节点，在<layout>节点中我们可以通过<data>节点来引入我们要使用的数据源

```
<?xml version="1.0" encoding="utf-8"?><layout xmlns:android="http://sche
mas.android.com/apk/res/android">


    <data>


        <variable

            name="user"

            type="com.donkor.demo.databinding.bean.User" />


    </data>


    <LinearLayout

        android:layout_width="match_parent"
```

```
        android:layout_height="match_parent"
        android:orientation="vertical">



        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.name}" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{String.valueOf(user.age)}" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:autoLink="all"
            android:text="@{user.myBlog}" />

    </LinearLayout>
</layout>
```

</layout>

注意 ：

※ 切记，在<layout>节点下是没有“layout_width”和“layout_height”的

※ type 中声明的就是我们的用户实体类 User，连同包名一定要写全！！！我们给其命名（name）为“user”,然后在 TextView 中的@{user.name}就是把这个 user 中的名字展示出来，之后的“age”“myBlog”同理

※ age 年龄在实体类 User 中我们定义的是整数类型，所以在布局中需要使用 String.valueOf(user.age)转换为字符串

一个 Binding 类会基于 layout 文件的名称而产生，并且添加“Binding”后缀。上述的 layout 文件是 main_basic.xml，因此生成的类名是 ActivityBasicBinding。通过 DataBindingUtil.setContentView 获取 bing 实例，进行数据绑定

```
public class BasicActivity extends Activity {



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        ActivityBasicBinding bing= DataBindingUtil.setContentView(this,
R.layout.activity_basic);

        User user=new User("donkor",10,"http://blog.csdn.net/donkor_");

        bing.setUser(user);

    }}
```

运行之后看下效果图

2．显示照片

看完图之后，我们看下如何实现。
我们选择用 glide 加载一张网络图片。glide 怎么在 studio 中使用，这里就不再讨论。如何显示一张网络图片，需要使用到 DataBinding 自定义属性 @BindingAdapter。在布局文件中使用这个自定义属性，显示出我们要展示的布局。
还是那个实体类 User，添加自定义属性@BindingAdapter({"imageUrl"})

```
public class User {

    public String imageUrl;



    public User(String imageUrl){

        this.imageUrl=imageUrl;

    }



    /**

     * 使用 ImageLoader 显示图片

     * @param imageView
```

```
     * @param url
     */
    @BindingAdapter({"imageUrl"})
    public static void imageLoader(ImageView imageView, String url) {
        Glide.with(imageView.getContext()).load(url).into(imageView);
    }}
public class User {
    public String imageUrl;

    public User(String imageUrl){
        this.imageUrl=imageUrl;
    }

    /**
     * 使用 ImageLoader 显示图片
     * @param imageView
     * @param url
     */
    @BindingAdapter({"imageUrl"})
    public static void imageLoader(ImageView imageView, String url) {
        Glide.with(imageView.getContext()).load(url).into(imageView);
    }}
public class User {
```

```
    public String imageUrl;


    public User(String imageUrl){

        this.imageUrl=imageUrl;

    }


    /**

     * 使用 ImageLoader 显示图片

     * @param imageView

     * @param url

     */

    @BindingAdapter({"imageUrl"})

    public static void imageLoader(ImageView imageView, String url) {

        Glide.with(imageView.getContext()).load(url).into(imageView);

    }}

public class User {

    public String imageUrl;


    public User(String imageUrl){

        this.imageUrl=imageUrl;

    }


    /**
```

```
     * 使用 ImageLoader 显示图片
     * @param imageView
     * @param url
     */
    @BindingAdapter({"imageUrl"})
    public static void imageLoader(ImageView imageView, String url) {
        Glide.with(imageView.getContext()).load(url).into(imageView);
    }}
public class User {
    public String imageUrl;

    public User(String imageUrl){
        this.imageUrl=imageUrl;
    }

    /**
     * 使用 ImageLoader 显示图片
     * @param imageView
     * @param url
     */
    @BindingAdapter({"imageUrl"})
    public static void imageLoader(ImageView imageView, String url) {
        Glide.with(imageView.getContext()).load(url).into(imageView);
```

```
    }}
public class User {
    public String imageUrl;

    public User(String imageUrl){
        this.imageUrl=imageUrl;
    }

    /**
     * 使用 ImageLoader 显示图片
     * @param imageView
     * @param url
     */
    @BindingAdapter({"imageUrl"})
    public static void imageLoader(ImageView imageView, String url) {
        Glide.with(imageView.getContext()).load(url).into(imageView);
    }}
public class User {
    public String imageUrl;

    public User(String imageUrl){
        this.imageUrl=imageUrl;
    }
```

```
/**
 * 使用 ImageLoader 显示图片
 * @param imageView
 * @param url
 */
@BindingAdapter({"imageUrl"})
public static void imageLoader(ImageView imageView, String url) {
    Glide.with(imageView.getContext()).load(url).into(imageView);
}}
```

※ 这里要主要的是，imageLoader 方法必须声明是 static，否则会报错。

新建 activity_image.xml，因为使用了自定义属性，所以在 layout 节点下要添加 xmlns:app="http://schemas.android.com/apk/res-auto"。在 ImageView 中使用 app:imageUrl="@{user.imageUrl}"。完整代码如下：

```
<?xml version="1.0" encoding="utf-8"?><layout xmlns:android="http://sche
mas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>

        <variable
            name="user"
            type="com.donkor.demo.databinding.bean.User" />
```

```
    </data>


    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">


        <ImageView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:imageUrl="@{user.imageUrl}" />


    </LinearLayout>
</layout>
```

最后运行之后，结果如上图。我就不再发一遍了，有兴趣的朋友再拖回去看一遍，反正下面还有~~

3．更多用法

简单的字符拼接

```
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{`my Name is :`+user.name}" />
```

BA… 892246

```
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:autoLink="all"
        android:text="@{`my Blog is :`+user.myBlog}" />
```

简单的三目运算

判断名字是否为空，不为空只显示 user.name，否则显示 donkor11：

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text='@{user.name ?? "donkor11"}' />
```

相当于

```
<TextView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text='@{user.name !=null?user.name: "donkor11"}' />
```

这里需要注意的是当{}中使用了双引号“”，最外层要改成单引号”

根据数据判断，显示数据

判断是否为学生，是则显示 11，反则，显示 00

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

892872246

```
    android:text="@{user.isStudent?String.valueOf(11):String.valueOf(00)}
" />
```

**修改样式 **

判断是否为学生，是则修改背景颜色 0xFF0000FF，反则，显示 0xFFFF0000

```
<TextView

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:background="@{user.isStudent?0xFF0000FF:0xFFFF0000}"

    android:text="donkor" />
```

写在之后的话，这里我们需要知道 Databinding 支持与不支持的表达式，语法。如下

支持的表达式：

1.Mathematical + - / * %
2.String concatenation +
3.Logical && ||
4.Binary & | ^
5.Unary + - ! ~
6.Shift >> >>> <<
7.Comparison == > < >= <=
8.instanceof
9.Grouping ()
10.Literals - character, String, numeric, null
11.Cast
12.Method calls
13.Field access
14.Array access []
15.Ternary operator ?:

不支持的表达式：

1.this
2.super
3.new
4.Explicit generic invocation

最后运行之后，看下效果图

BAT交流群：892872246

4 ． 点击事件

单击事件

单击事件在实际开发中，使用频率有多高这里就不解释了。下面我们直接看怎么用

在 data 节点下的 variable 下 type 引用 android.view.View.OnClickListener。

```
<data>

    <variable

        name="myClick"

        type="android.view.View.OnClickListener" /></data>
```

Button 按钮直接引用"myClick"

```
<Button

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:text="单击按钮"

    android:onClick="@{myClick}" />
```

长按事件

长按事件的 button，首先需要分配一个 id 给 Button

```
<Button
```

```
    android:id="@+id/btnLongClick"

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:text="长按按钮"

    />
```

在 Activity 中，使用数据绑定，直接引用按钮 id。完整代码如下

```
public class ClickActivity extends Activity {



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        ActivityClickBinding bing=DataBindingUtil.setContentView(this, R.
layout.activity_click);



        //事件绑定 -- 单击

        bing.setMyClick(new View.OnClickListener() {

            @Override

            public void onClick(View v) {

                Toast.makeText(ClickActivity.this,"发生了点击事件",Toast.L
ENGTH_SHORT).show();

            }

        });
```

```
        //事件绑定 -- 长按

        bing.btnLongClick.setOnLongClickListener(new View.OnLongClickLis
tener() {

            @Override

            public boolean onLongClick(View v) {

                Toast.makeText(ClickActivity.this,"发生了长按事件",Toast.L
ENGTH_SHORT).show();

                return false;

            }

        });

    }}
```

最后运行之后，看下效果图。

BAT交流群 892246

▲5．绑定ListView

Listview在实际开发中使用频率同样很高，看了上面那么多例子。就知道Databinding绑定ListView一样简单。接下来看一组美图。

看完之后，能看出布局其实就是一张ImageView，底下加一个TextView。例子很简单，然后看下如何实现。首先需要新建一个美女的实体类，我们先定义Beauty,beautyNum就是美女底下的TextView文本。

```
public class Beauty {

    public String beautyNum;

    public String imageUrl;


    public Beauty(String beautyNum, String imageUrl) {

        this.beautyNum = beautyNum;

        this.imageUrl = imageUrl;

    }


    @BindingAdapter({"imageUrl"})

    public static void beautyImage(ImageView imageView, String url) {
```

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

    }}
```

然后看下主布局 activity_listview.xml，在 data 节点下的 variable 下 type 引用 android.widget.BaseAdapter。

```
<?xml version="1.0" encoding="utf-8"?><layout xmlns:android="http://sche
mas.android.com/apk/res/android"

    xmlns:app="http://schemas.android.com/apk/res-auto">



    <data>



        <variable

            name="adapter"

            type="android.widget.BaseAdapter" />

    </data>



    <LinearLayout

        android:layout_width="match_parent"

        android:layout_height="match_parent">



        <ListView

            android:layout_width="match_parent"

            android:layout_height="match_parent"

            app:adapter="@{adapter}" />
```

```
    </LinearLayout></layout>
```

然后 ListView 的 item 布局 item_listview.xml,同样比较简单，在这不做解释。

```
<?xml version="1.0" encoding="utf-8"?><layout xmlns:android="http://sche
mas.android.com/apk/res/android"

    xmlns:app="http://schemas.android.com/apk/res-auto">



    <data>

        <variable

            name="beauty"

            type="com.donkor.demo.databinding.bean.Beauty" />

    </data>



    <LinearLayout

        android:layout_width="match_parent"

        android:layout_height="match_parent"

        android:orientation="vertical">




        <ImageView

            android:layout_width="wrap_content"

            android:layout_height="wrap_content"

            app:imageUrl="@{beauty.imageUrl}" />
```

```
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{beauty.beautyNum}" />

    </LinearLayout>
</layout>
```

有 ListView 的地方，十有八九就有适配器。这里介绍一个终极适配器的写法 MyBaseAdapter。尽管这里仅仅是给美女使用的适配器，但是已经说明了是终极写法。我们就不叫它美女适配器了。

```
public class MyBaseAdapter<T> extends BaseAdapter {

    private List<T> list;
    private int layoutId;
    private int variableId;
    private LayoutInflater mInflater;

    public MyBaseAdapter(Context context, List<T> list, int layoutId, in
t variableId) {
        this.list = list;
        this.layoutId = layoutId;
        this.variableId = variableId;
        this.mInflater = LayoutInflater.from(context);
```

```
    }

    @Override
    public int getCount() {
        return list.size() == 0 ? 0 : list.size();
    }

    @Override
    public Object getItem(int position) {
        return list.get(position);
    }

    @Override
    public long getItemId(int position) {
        return position;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent)
{
        ViewDataBinding dataBinding;
        if (convertView == null) {
            dataBinding = DataBindingUtil.inflate(mInflater, layoutId, pa
rent, false);
```

```
        } else {

            dataBinding = DataBindingUtil.getBinding(convertView);

        }

        dataBinding.setVariable(variableId, list.get(position));



        return dataBinding.getRoot();

    }}
```

然后解释下其中几个变量

Context context：上下文，不比多说
List list：传进来的数据集合，不解释
int layoutId：　item 布局的资源 id
int variableId：系统自动生成的
※ 注意布局加载方式为 DataBindingUtil 类中的 inflate 方法
最后看下 ListViewActivity 完整代码

```
public class ListViewActivity extends Activity {

    private List<Beauty> list;



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        ActivityListviewBinding binding = DataBindingUtil.setContentView
(this, R.layout.activity_listview);



        list = new ArrayList<>();

        //加载数据

        initData();
```

```
        MyBaseAdapter<Beauty> adapter = new MyBaseAdapter<>(ListViewActiv
ity.this, list, R.layout.item_listview, BR.beauty);

        binding.setAdapter(adapter);

    }



    void initData() {

        Beauty beauty1 = new Beauty("第一个美女", "http://img2.imgtn.bdimg.
com/it/u=3988249408,1489015532&fm=21&gp=0.jpg");

        Beauty beauty2 = new Beauty("第二个美女", "http://img4.imgtn.bdimg.
com/it/u=2579627311,3580753633&fm=21&gp=0.jpg");

        Beauty beauty3 = new Beauty("第三个美女", "http://img5.imgtn.bdimg.
com/it/u=539171541,1245868076&fm=23&gp=0.jpg");

        Beauty beauty4 = new Beauty("第四个美女", "http://img1.imgtn.bdimg.
com/it/u=3494499027,4116428522&fm=23&gp=0.jpg");

        Beauty beauty5 = new Beauty("第五个美女", "http://img4.imgtn.bdimg.
com/it/u=645329305,336210525&fm=23&gp=0.jpg");

        list.add(beauty1);

        list.add(beauty2);

        list.add(beauty3);

        list.add(beauty4);

        list.add(beauty5);

    }}
```

最后运行之后，结果如上图。有兴趣的朋友再拖回去看一遍，反正下面没有美女了~~

6 ． 数据更新

BAT交流群 892246

Databinding 的数据更新同样惹人喜欢。最主要的是代码简洁明了，UI 还同步更新。
首先看下效果图

数据更新可以使用如下两种方式，分别对学生文本和老师文本进行修改
让实体类(Student)继承自 BaseObservable
给需要改变的字段的 get 方法添加上@Bindable 注解，然后给需要改变的字段(例如 name)的 set 方法加上 notifyPropertyChanged(BR.name)；字段 number 同理，在 set 方法内加上 notifyPropertyChanged(BR.number)；

```
public class Student extends BaseObservable {

    @Bindable

    public String name;

    @Bindable

    public String number;



    public Student() {

    }
```

```
public Student(String name, String number) {
    this.name = name;
    this.number = number;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
    notifyPropertyChanged(BR.name);
}

public String getNumber() {
    return number;
}

public void setNumber(String number) {
    this.number = number;
    notifyPropertyChanged(BR.number);
}}
```

** 使用 DataBinding 提供的 ObservableFields 来创建实体类**

Teacher 实体类,这个更加简单，几行代码搞定

```
public class Teacher extends BaseObservable {

    public ObservableField<String> name = new ObservableField<>();

    public ObservableField<String> number = new ObservableField<>();



    public Teacher() {

    }



    public Teacher(String name, String number) {

        this.name.set(name);

        this.number.set(number);

    }

}
```

然后看下布局文件 activity_data_notify 代码,跟上文描述的基本类似，这里不再过多赘述。

```
<?xml version="1.0" encoding="utf-8"?><layout xmlns:android="http://sche
mas.android.com/apk/res/android">



    <data>



        <variable

            name="student"

            type="com.donkor.demo.databinding.bean.Student" />
```

```
        <variable
            name="teacher"
            type="com.donkor.demo.databinding.bean.Teacher" />

    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:orientation="vertical">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{student.name}" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{student.number}" />
```

```
    <Button
        android:id="@+id/btnStudent"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="30dp"
        android:layout_marginRight="30dp"
        android:layout_marginTop="10dp"
        android:text="点击按钮修改学生文本"
        />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{teacher.name}" />


    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{teacher.number}" />

    <Button
```

```
            android:id="@+id/btnTeacher"

            android:layout_width="match_parent"

            android:layout_height="wrap_content"

            android:layout_marginLeft="30dp"

            android:layout_marginRight="30dp"

            android:layout_marginTop="10dp"

            android:text="点击按钮修改老师文本"

            />

    </LinearLayout>

</layout>
```

最后看下 DataNotifyActivity 完整代码

```
public class DataNotifyActivity extends Activity {



    private Student student;

    private Teacher teacher;



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        ActivityDataNotifyBinding binding = DataBindingUtil.setContentVi
ew(this, R.layout.activity_data_notify);

        student = new Student("studen_donkor", "number_aa");

        binding.setStudent(student);
```

```
        teacher = new Teacher("teacher_donkor", "number_bb");

        binding.setTeacher(teacher);



        //点击按钮，更新数据,同时更新 UI

        binding.btnStudent.setOnClickListener(new View.OnClickListener()
{

            @Override

            public void onClick(View v) {

                student.setName("donkor is not student");

                student.setNumber("number_00000");

            }

        });



        binding.btnTeacher.setOnClickListener(new View.OnClickListener()
{

            @Override

            public void onClick(View v) {

                teacher.name.set("donkor is not teacher");

                teacher.number.set("number_11111");

            }

        });

    }}
```

7．数据集合

DataBinding 中给我们提供了一些现成的集合 ObservableArrayList，

ObservableArrayMap。这里不做介绍。

8.6 WorkManager 内核揭秘

什么是 Workmanager

WorkManager 是 google 提供的异步执行任务的管理框架，会根据手机的 API 版本和应用程序的状态来选择适当的方式执行任务。当应用在运行的时候会在应用的进程中开一条线程来执行任务，当退出应用时，WorkManager 会选择根据设备的 API 版本使用适合的算法调用 JobScheduler 或者 Firebase JobDispatcher,或者 AlarmManager 来执行任务。如下图：

由上图可以看出 WorkManager 管理任务执行时底层还是调用 JobScheduler,JobDispatcher,AlarmManager，不过 WorkManager 会根据 Android 系统的 API 和应用的运行状态来选择合适的方式执行，并不用我们自己去考虑应用复杂的运行状态来进行选择使用 JobScheduler 还是 JobDispatcher 或者 AlarmManager 调用规则如下图：

BAT交流群：892872246

WorkManager 在项目中配置

使用 WorkManager 需要 gradle 依赖，进行一下简单配置就可以使用了。找到项目中的 app/build.gradle 目录下在上面加上下面的依赖。

```
dependencies {

    // 其他依赖配置

    def work_version = "1.0.0-beta02"

    implementation "android.arch.work:work-runtime:$work_version"}
```

可以在 https://developer.android.com/topic/libraries/architecture/adding-components#workmanager 获取当前的 work-runtime 版本并且设置正确的版本号。

BAT交流群

WorkManager 主要类及使用

如下图给出了 WorkManager 中主要的类以及关系图，黄色区域是最主要的三个类，构成了 WorkManager 的基本框架，红色部分和绿色部分是关联的黄色部分的具体实现或者类里面包含一些规则或数据。

1、Worker 处理要执行的任务的具体逻辑。

2、WorkerRequest 代表一个独立的可以执行的任务，以及任务执行时的条件和规则，比如说任务执行一次还是多次以及任务的触发条件是什么任务有什么约束等。

3、WorkManager 提供队列将要执行的 WorkerRequest 放到队列中管理和执行。
如下图，三个主要类的关系：

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

下面分别介绍三个类的作用和使用方法。

Worker

Worker 是一个抽象类，当有一个要执行的任务的时候可以继承 Worker 类，重写 doWork()方法在 doWork()方法中实现具体任务的逻辑。

```
public class MyWorker extends Worker {

    public MyWorker(

            @NonNull Context appContext,

            @NonNull WorkerParameters workerParams) {
```

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
        super(appContext, workerParams);

    }

    @NonNull

    @Override

    public Worker.Result doWork() {


        Context applicationContext = getApplicationContext();


        try {


            Bitmap picture = BitmapFactory.decodeResource(

                    applicationContext.getResources(),

                    R.drawable.test);


            return Worker.Result.SUCCESS;


        } catch (Throwable throwable) {


            return Worker.Result.FAILURE;


        }

    }}
}
```

在上面的 MyWorker 实例中，继承了 Worker 并且重写了 doWork()方法，需要注意的是 doWork()方法是有返回值 Worker.Result 的，可以在任务执行成功是返回 Worker.Result.SUCCESS，在任务执行出现异常时返回 Worker.Result.FAILURE doWork()方法的返回值主要有三种

1、Worker.Result.SUCCESS 表示任务执行成功

2、Worker.Result.FAILURE 表示任务执行失败

3、Worker.Result.RETRY 通知 WorkManager 之后再尝试执行该任务

WorkRequest

WorkRequest 要指定执行任务的 Worker，也可以给 WorkRequest 加一些规则，比如说什么时候执行任务，任务执行一次还是多次，每一个 WorkRequest 都有一个自动产生的唯一 ID，可以根据唯一 ID 获取对应任务的状态以及是否取消对应的任务。如下图 WorkRequest 有两个实现类如下图：

1、OneTimeWorkRequest 任务只执行一次

```
OneTimeWorkRequest myWorkRequest =

        new OneTimeWorkRequest.Builder(MyWorker.class)

    .build();

    //将上面定义的 MyWorker 加入到 OneTimeRequest.Builder 方法中 WorkManager.
getInstance().enqueue(myWorkRequest);//获取 WorkManager 实例并将 WorkReques
t 进队
```

2、PeriodicWorkRequest

PeriodicWorkRequest 重复执行任务，直到被取消才停止。首次执行是任务提交

后立即执行或者满足所给的 Constraints 条件。以后执行都会根据所给的时间间隔来执行。注意任务的执行可能会有延时，因为 WorkManager 会根据 OS 的电量进行优化。
假如设置的 Periodic Work 是 24 小时执行一次，有可能根据电池优化策略执行的过程如下：

```
1     | Jan 01, 06:00 AM

2     | Jan 02, 06:24 AM

3     | Jan 03, 07:15 AM

4     | Jan 04, 08:00 AM

5     | Jan 05, 08:00 AM

6     | Jan 06, 08:02 AM
```

由上面的执行时间可以看出，PeriodicWorkRequest 并不是准确的按照 24 小时来执行，会有一定的时间延迟。因此如果需要准确的间隔时间来执行任务的话不能使用 PeriodicWorkRequest。

```
Constraints constraints = new Constraints.Builder().setRequiredNetworkTy
pe(NetworkType.CONNECTED).build();PeriodicWorkRequest build = new Period
icWorkRequest.Builder(MyWorker.class, 25, TimeUnit.MILLISECONDS)

          .addTag(TAG)

          .setConstraints(constraints)

          .build();

WorkManager instance = WorkManager.getInstance();if (instance != null) {

         instance.enqueueUniquePeriodicWork(TAG, ExistingPeriodicWorkPo
licy.REPLACE, build);}
```

Constraints

可以给任务加一些运行的 Constraints 条件，比如说当设备空闲时或者正在充电或者连接 WiFi 时执行任务。

```
Constraints myConstraints = new Constraints.Builder()

    .setRequiresDeviceIdle(true)
```

BA 群：892872246

```
    .setRequiresCharging(true)

    // Many other constraints are available, see the

    // Constraints.Builder reference

     .build();OneTimeWorkRequest myWork =

                new OneTimeWorkRequest.Builder(CompressWorker.class)

     .setConstraints(myConstraints)

     .build();
```

WorkManager

WorkManager 管理 WorkRequest 队列。并根据设备和其他条件选择执行的具体方式。在大部分情况在如果没有给队列加 Contraints，WorkManager 会立即执行任务。

```
WorkManager.getInstance().enqueue(myWork);
```

如果要检查任务的执行状态可以通过获取 WorkInfo,WorkInfo 在 WorkManager 里面的 LiveData<WorkInfo>中。下面是判断任务是否结束的方式。

```
WorkManager.getInstance().getWorkInfoByIdLiveData(myWork.getId())

    .observe(lifecycleOwner, workInfo -> {

        // Do something with the status

        if (workInfo != null && workInfo.getState().isFinished()) {

            // ...

        }

    });
```

取消任务执行

通过任务的 ID 可以获取任务从而取消任务。任务 ID 可以从 WorkRequest 中获取。

```
UUID compressionWorkId = compressionWork.getId();

WorkManager.getInstance().cancelWorkById(compressionWorkId);
```

892872246

注意并不是所有的任务都可以取消，当任务正在执行时是不能取消的，当然任务执行完成了，取消也是意义的，也就是说当任务加入到 ManagerWork 的队列中但是还没有执行时才可以取消。

WorkManager 多任务调度

有时候可能有很多任务需要执行，并且这些任务之前可能有先后顺序或者某些依赖关系，WorkManager 提供了很好的方式。

1、先后顺序执行单个任务

比如说有三个任务 workA,workB,workC,并且执行顺序只能时 workA---->workB---->workC 可以用如下的方式处理

```
WorkManager.getInstance()
    .beginWith(workA)
    .then(workB)  instance
    .then(workC)
    .enqueue();
```

上面的workA,workB,workC,都是WorkRequest的子类实现对象。WorkManager会根据上面的先后顺序来执行workA,workB,workC,，但是如果执行过程中三个任务中有一个失败，整个执行都会结束。并且返回Result.failure()。

2、先后顺序执行多个任务列

有时候可能要先执行一组任务，然后再执行下一组任务，可以使用下面的方式来完成。

```
WorkManager.getInstance()
    // First, run all the A tasks (in parallel):
    .beginWith(Arrays.asList(workA1, workA2, workA3))
    // ...when all A tasks are finished, run the single B task:
    .then(workB)
    // ...then run the C tasks (in any order):
    .then(Arrays.asList(workC1, workC2))
    .enqueue();
```

3、多路径先后执行

上面两种方式都是单路径执行，可以实现更加复杂的多路径执行方式，使用 WorkContinuation.combine(List<OneTimeWorkRequest>)如下图要实现的执行方式：

BAT交流群：892872246

```
WorkContinuation chain1 = WorkManager.getInstance()
    .beginWith(workA)
    .then(workB);
WorkContinuation chain2 = WorkManager.getInstance()
    .beginWith(workC)
    .then(workD);
WorkContinuation chain3 = WorkContinuation
    .combine(Arrays.asList(chain1, chain2))
    .then(workE);
chain3.enqueue();
```

使用 WorkManager 遇到的问题

1、使用 PeriodicWorkRequest 只执行一次，并不重复执行。

```
    WorkManager instance= new PeriodicWorkRequest.Builder(PollingWorker.
class, 10, TimeUnit.MINUTES)

                .build();
```

原因：PeriodicWorkRequest 默认的时间间隔是 15 分钟如果设置的时间小于 15 分钟，就会出现问题。

解决方法：设置的默认时间必须大于或等于 15 分钟。另外要注意，就算设置的时间为 15 分钟也不一定间隔 15 分钟就执行。

2、在 doWork()方法中更新 UI 导致崩溃。
原因：doWork()方法是在 WorkManager 管理的后台线程中执行的，更新 UI 操作只能在主线程中进行。

解决方法：当 doWork()耗时方法执行完之后，将更新 UI 操作抛到主线中执行，可以用 handle 来实现，如下：

```
/**

 * Created time 15:32.

 *

 * @author huhanjun

 * @since 2019/1/23

 */public class PollingWorker extends Worker {

    public static final String TAG = "PollingWorker";



    @NonNull

    @Override

    public Result doWork() {

        Log.d(TAG, "doWork");
```

```
        try {
            polling();
            runOnUIThread(new Runnable() {
                @Override
                public void run() {
                    //更新 UI 操作
                }
            });
            return Result.SUCCESS;
        } catch (Exception e) {
            Log.d(TAG, "failure");
            return Result.FAILURE;
        }
    }

    private void polling() {
        Log.d(TAG, "Polling");
    }

    private void runOnUIThread(Runnable runnable) {
        new Handler(Looper.getMainLooper()).post(runnable);
    }}
```

BAT交... 892246

在执行完 Polling 方法后，将更新操作抛给了 runOnUIThread()方法，这样就可以在上面代码注释的地方执行更新操作。

8.7 Lifecycles 生命周期

使用生命周期感知组件处理生命周期

生命周期感知组件可以在其他组件（例如 activity 和 fragment）的生命周期状态更改时执行操作。这些组件可帮助我们生成更轻量级且更容易维护的代码。

我们通常会在 activity 和 fragment 的生命周期方法中实现自己的业务逻辑。但是，这种方式会导致我们代码组织的不良和出错的增加。但是通过使用生命周期感知组件，我们就可以将依赖组件生命周期的代码移出生命周期方法。

android.arch.lifecycle 包提供的类和接口，使我们可以构建生命周期感知组件 - 这些组件可以根据 activity 和 fragment 的当前生命周期状态自动调整其行为。

Android Framework 中定义的大多数应用程序组件都附加了生命周期。生命周期由操作系统或进程中运行的 Framework 代码管理。它们是 Android 工作原理的核心，我们的应用程序必须遵循它们，如果不这样做可能会触发内存泄漏甚至应用程序崩溃。

假设我们现在有一个 activity ，用来在手机屏幕上显示设备位置。常见的实现可能如下：

```
class MyLocationListener {

    public MyLocationListener(Context context, Callback callback) {

        // ...

    }



    void start() {

        // connect to system location service

    }
```

```
    void stop() {

        // disconnect from system location service

    }}


class MyActivity extends AppCompatActivity {

    private MyLocationListener myLocationListener;


    @Override

    public void onCreate(...) {

        myLocationListener = new MyLocationListener(this, (location) -> {

            // update UI

        });

    }


    @Override

    public void onStart() {

        super.onStart();

        myLocationListener.start();

        // manage other components that need to respond

        // to the activity lifecycle

    }


    @Override
```

```
    public void onStop() {

        super.onStop();

        myLocationListener.stop();

        // manage other components that need to respond

        // to the activity lifecycle

    }}
```

尽管这个示例看起来很好，但在实际的应用程序中，最终会有太多的回调来管理 UI 和 其他组件以响应生命周期的当前状态。管理多个组件会在生命周期方法中放置大量代码，导致如 onStart() 和 onStop() 这样的方法难以维护。

此外，我们无法保证 MyLocationListener 在 MyActivity 停止之前启动，如果我们需要执行长时间运行的操作（例如在 onStart() 中进行一些 配置检查），这会导致 MyActivity 的 onStop() 完成之前而这个操作才会回调，这就使组件保持活动的时间会长于其所需的时间。

```
class MyActivity extends AppCompatActivity {

    private MyLocationListener myLocationListener;



    public void onCreate(...) {

        myLocationListener = new MyLocationListener(this, location -> {

            // update UI

        });

    }



    @Override

    public void onStart() {

        super.onStart();
```

```
        Util.checkUserStatus(result -> {

            // what if this callback is invoked AFTER activity is stoppe
d?

            if (result) {

                myLocationListener.start();

            }

        });

    }



    @Override

    public void onStop() {

        super.onStop();

        myLocationListener.stop();

    }}
}
```

android.arch.lifecycle 包提供了类和接口，可以帮助我们以弹性和隔离的方式解决这些问题。

生命周期

Lifecycle 是一个持有关于组件（如 activity 或 fragment）生命周期状态信息并允许其他对象观察此状态的类。

Lifecycle 使用两个主要枚举来跟踪其关联组件的生命周期状态：

Event

从 Framework 和 Lifecycle 类分发生命周期事件 。这些事件映射到 activities 和 fragments 中的回调事件。

BAT交流群：892872246

State

通过 Lifecycle 对象来跟踪组件的当前状态 。

将 State 视为图形的节点，将 Event 视为这些节点之间的边缘。

一个类可以通过向组件的方法添加注解来监视其生命周期状态。我们可以通过调用 Lifecycle 类的 addObserver() 方法来添加观察者，并传递观察者的实例，如以下示例所示：

```
public class MyObserver implements LifecycleObserver {

    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)

    public void connectListener() {

        ...

    }



    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)

    public void disconnectListener() {

        ...

    }}
```

```
myLifecycleOwner.getLifecycle().addObserver(new MyObserver());
```

在上面的示例中，myLifecycleOwner 对象实现了 LifecycleOwner 接口，这个在下一节中进行说明。

LifecycleOwner

LifecycleOwner 是一个接口，它只有一个 getLifecycle() 方法，表明实现 LifecycleOwner 接口的类有一个 Lifecycle 。如果你正在尝试管理整个应用进程的生命周期，可以看下 ProcessLifecycleOwner。

LifecycleOwner 接口从个别类（例如 Fragment 和 AppCompatActivity）中抽象出了 Lifecycle 的所有权，并允许编写与其一起使用的组件。任何自定义类都可以实现 LifecycleOwner 接口。

实现了 LifecycleObserver 的组件与实现了 LifecycleOwner 的组件可以无缝工作，因为所有者可以提供生命周期，观察者可以注册观察。

例如位置跟踪的例子，我们可以使 MyLocationListener 类实现 LifecycleObserver ，然后在 activity 生命周期的 onCreate() 方法中初始化它，这意味着响应生命周期状态变化的逻辑可以被声明在 MyLocationListener 而不是 activity，当把逻辑代码置于在各个组件中后，就能使 activities 和 fragments 更易于管理。

```
class MyActivity extends AppCompatActivity {

    private MyLocationListener myLocationListener;



    public void onCreate(...) {

        myLocationListener = new MyLocationListener(this, getLifecycle(),
 location -> {

            // update UI

        });

        Util.checkUserStatus(result -> {

            if (result) {
```

```
            myLocationListener.enable();

        }

    });

}}
```

一个常见的用例是，如果 Lifecycle 现在不处于激活状态，则应避免调用某些回调 。例如：如果回调在 Activity 状态被保存到 onSaveInstanceState() 后运行 Fragment Transaction，就会触发崩溃，因此我们会避免调用该回调。

为了简化这个用例， Lifecycle 类允许其他对象查询当前状态。

```
class MyLocationListener implements LifecycleObserver {

    private boolean enabled = false;

    public MyLocationListener(Context context, Lifecycle lifecycle, Call
back callback) {

        ...

    }



    @OnLifecycleEvent(Lifecycle.Event.ON_START)

    void start() {

        if (enabled) {

            // connect

        }

    }



    public void enable() {

        enabled = true;
```

```
        if (lifecycle.getCurrentState().isAtLeast(STARTED)) {

            // connect if not connected

        }

    }



    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)

    void stop() {

        // disconnect if connected

    }}
```

通过这种实现，我们的 LocationListener 类完全可以感知生命周期。如果我们需要从另一个 activity 或 fragment 使用 LocationListener，我们只需要初始化它。所有设置和卸载操作都由 LocationListener 类本身管理。

如果一个库提供了需要使用 Android 生命周期的类，Google 建议我们使用生命周期感知组件。我们的客户端可以轻松地集成这些组件，而无需在客户端进行手动管理生命周期。

实现自定义 LifecycleOwner

在 Support Library 26.1.0 及更高版本的 Fragments 和 Activities 已实现了 LifecycleOwner 接口。

如果你想让自定义的类成为 LifecycleOwner，可以使用 LifecycleRegistry 类，但是你需要将生命周期事件转发到此类，如以下代码所示：

```
public class MyActivity extends Activity implements LifecycleOwner {

    private LifecycleRegistry mLifecycleRegistry;



    @Override

    protected void onCreate(Bundle savedInstanceState) {
```

BA… 2246

```
        super.onCreate(savedInstanceState);


        mLifecycleRegistry = new LifecycleRegistry(this);

        mLifecycleRegistry.markState(Lifecycle.State.CREATED);

    }


    @Override

    public void onStart() {

        super.onStart();

        mLifecycleRegistry.markState(Lifecycle.State.STARTED);

    }


    @NonNull

    @Override

    public Lifecycle getLifecycle() {

        return mLifecycleRegistry;

    }}
```

生命周期感知组件的最佳实践

尽可能精简我们的 UI 控制器（Activity 和 Fragment）。它们不应该尝试获取自己的数据， 相反，应该使用 ViewModel 来执行此操作，并观察 LiveData 对象以将数据更改反映到视图上。

编写数据驱动的 UI，UI 控制器负责在数据更改时更新视图，或者将用户操作通知给 ViewModel 。

BAT交流群：892872246

把我们的数据逻辑放在 ViewModel 类中。 ViewModel 应该作为 UI 控制器和应用程序其余部分之间的桥接器。不过 ViewModel 是没有获取数据（例如从网络）的职责的。相反， ViewModel 应该调用合适的组件（例如 Repository 组件）来获取数据，然后将结果提供给 UI 控制器。

使用 Data Binding 在 视图 和 UI 控制器之间维持一个干净的界面。这会使我们的视图更具声明性，并最大限度地减少在 Activity 和 Fragment 中编写所需的更新代码。如果你更喜欢使用 Java 编程语言执行此操作，那就使用 Butter Knife 之类的库来避免样板代码，使代码具有更好的抽象。

如果你的 UI 很复杂，可以考虑创建一个 presenter 类来处理 UI 修改。这可能是一项蛋疼的任务，但是它可以让我们的 UI 组件更容易测试。

避免在你的 ViewModel 中引用 View 或 Activity 的 context 。如果 ViewModel 的存活时间超过这个 activity（如在 configuration 更改的情况下），导致垃圾收集器未正确处理 activity 造成内存泄露。

生命周期感知组件的用例

生命周期感知组件可以让我们在各种情况下更轻松地管理生命周期。比如以下场景：

在粗粒度和细粒度的位置更新之间切换。使用生命周期感知组件可在我们的位置应用程序可见时启用细粒度位置更新，并在应用程序位于后台时切换到粗粒度更新。LiveData 是一个生命周期感知组件，它允许我们的应用在用户更改位置时自动更新 UI。

停止和开始视频缓冲。使用生命周期感知组件可以尽快开启视频缓冲，但推迟播放直到应用程序完全启动。我们还可以使用生命周期感知组件在销毁应用程序时终止缓冲。

开启和停止网络连接。使用生命周期感知组件在应用程序处于前台时启用网络数据的实时更新（流式传输），并在应用程序进入后台时自动暂停。

暂停和恢复动画绘制。当应用程序在后台时，使用生命周期感知组件暂停绘制动画内容，并在应用程序位于前台后恢复绘制内容。

处理 ON_STOP 事件

如果一个 Lifecycle 属于一个 AppCompatActivity 或 Fragment，当 AppCompatActivity 或 Fragment 的 onSaveInstanceState() 方法被调用时， 这个 Lifecycle 的状态就会变为 CREATED ，它的 ON_STOP 事件也会被分发 。

BAT交流群：892872246

当一个 AppCompatActivity 或 Fragment 的状态被保存到 onSaveInstanceState() 后，它的 UI 在 ON_START 被调用之前被认为是不可变的 。保存状态后尝试修改 UI 可能会导致应用程序的导航状态不一致，这就是为什么如果应用程序在保存状态后运行 FragmentTransaction ，FragmentManager 会抛出异常的原因，详细原因请查看 commit() 。

如果观察者的关联 Lifecycle 不是 STARTED 状态， LiveData 会通过避免调用其观察者来防止这种边缘情况开箱即用。在背后，它会确保在调用它的观察者之前调用 isAtLeast() 。

蛋疼的是，AppCompatActivity 的 onStop() 方法会在 onSaveInstanceState() 之后调用，这就导致程序留下了不允许更改 UI 状态但 Lifecycle 又还没转变为 CREATED 状态的间隔。

为了防止出现此问题， Lifecycle 类的 beta2 版本以及更低版本将 CREATED 状态标记为不调度事件，即使在系统调用 onStop() 之前未调度事件，任何检查当前状态的代码都会获得实际值。

悲催的是这个解决方案有两个重大问题：

在 API 23 和更低级别，Android 系统实际上保存了 activity 的状态，即使它被另一个 activity 部分覆盖。换句话说，Android 系统调用 onSaveInstanceState() 但不一定要调用 onStop() 。这会创建一个潜在的长间隔，即使无法修改其 UI 状态，观察者仍然- - 认为生命周期处于活跃状态。

任何类想要向 LiveData 类暴露类似行为 都必须实现 Lifecycle beta 2 版本 和更低版本提供的解决方法 。

注意

为了使此流程更简单并提供与旧版本的更好兼容性，从 1.0.0-rc1 版本开始，在调用 onSaveInstanceState() 时 Lifecycle 对象被标记为 CREATED 并且调度 ON_STOP ，而无须等待调用 onStop() 方法。这可能不会影响我们的代码，但我们需要注意这一点，因为它与 API 26 及更低级别的 Activity 类中的调用顺序不匹配。

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

。

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

。

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

BAT交流群：892872246

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246