

Android 高级开发面试题以及答案整理

网上高级工程师面试相关文章鱼龙混杂，要么一堆内容，要么内容质量太浅，鉴于此我整理了如下安卓开发高级工程师面试题以及答案帮助大家顺利进阶为高级工程师，目前我就职于某大厂安卓高级工程师职位，在当下大环境下也想为安卓工程师出一份力，通过我的技术经验整理了面试经常问的题，答案部分会是一篇文章或者几篇文章，都是我认真看过并且觉得不错才整理出来，大家知道高级工程师不会像刚入门那样被问的问题一句话两句话就能表述清楚，所以我通过过滤好文章来帮助大家理解，进入正题：

一、Handler 相关知识

1、Handler Looper Message 关系是什么？

分析 Handler

首先我们来分析分析一下 Handler 的用法，我们知道，要创建一个 Handler 对象非常的简单明了，直接进行 new 一个对象即可，但是你有没有想过，这里会隐藏着什么注意点呢。现在可以试着写一下下面的一小段代码，然后自己运行看看：

```
public class MainActivity extends ActionBarActivity {  
  
    private Handler mHandler0;  
  
    private Handler mHandler1;
```

```
@Override

protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_main);

    mHandler0 = new Handler();

    new Thread(new Runnable() {

        @Override

        public void run() {

            mHandler1 = new Handler();

        }

    }).start();

}

}
```

这一小段程序代码主要创建了两个 Handler 对象，其中，一个在主线程中创建，而另外一个则在子线程中创建，现在运行一下程序，则你会发现，在子线程创建的 Handler 对象竟然会导致程序直接崩溃，提示的错误竟然是 **Can't create handler inside thread that has not called Looper.prepare()**

于是我们按照 logcat 中所说，在子线程中加入 Looper.prepare(), 即代码如下：

```
new Thread(new Runnable(){

    @Override

    public void run(){

        Looper.prepare();

        mHandler1 = new Handler()1

    }}).start();
```

再次运行一下程序，发现程序不会再崩溃了，可是，单单只加这句 Looper.prepare() 是否就能解决问题了。我们探讨问题，就要知其然，才能了解得更多。我们还是先分析一下源码吧，看看为什么在子线程中没有加 Looper.prepare() 就会出现崩溃，而主线程中为什么不用加这句代码？我们看下 Handler() 构造函数：

```
public Handler() {
```

```
    this(null, false);}
```

构造函数直接调用 `this(null, false)`，于是接着看其调用的函数，

```
public Handler(Callback callback, boolean async) {

    if (FIND_POTENTIAL_LEAKS) {

        final Class<? extends Handler> klass = getClass();

        if ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass()) &&

            (klass.getModifiers() & Modifier STATIC) == 0) {

            Log.w(TAG, "The following Handler class should be static or leaks might occur: " +

                klass.getCanonicalName());
        }
    }

    mLooper = Looper.myLooper();

    if (mLooper == null) {

        throw new RuntimeException(
            "Can't create handler inside thread that has not called Looper.prepare()");
    }

    mQueue = mLooper.mQueue;
```

```
mCallback = callback;  
  
mAsynchronous = async;  
  
}
```

不难看出，源码中调用了 `mLooper = Looper.myLooper()` 方法获取一个 Looper 对象，若此时 Looper 对象为 null，则会直接抛出一个“Can't create handler inside thread that has not called Looper.prepare()” 异常，那什么时候造成 `mLooper` 是为空呢？那就接着分析 `Looper.myLooper()`，

```
public static Looper myLooper() {  
  
    return sThreadLocal.get();  
  
}
```

这个方法在 `sThreadLocal` 变量中直接取出 Looper 对象，若 `sThreadLocal` 变量中存在 Looper 对象，则直接返回，若不存在，则直接返回 null，而 `sThreadLocal` 变量是什么呢？

```
static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>()  
();
```

它是本地线程变量，存放在 Looper 对象，由这也可看出，每个线程只有存有一个 Looper 对象，可是，是在哪里给 `sThreadLocal` 设置 Looper 的呢，通过前面的试验，我们不难猜到，应该是在 `Looper.prepare()` 方法中，现在来看看它的源码：

```
private static void prepare(boolean quitAllowed) {  
  
    if (sThreadLocal.get() != null) {  
  
        throw new RuntimeException("Only one Looper may be created per thread");  
  
    }  
  
    sThreadLocal.set(new Looper(quitAllowed));}
```

由此看到，我们的判断是正确的，在 `Looper.prepare()` 方法中给 `sThreadLocal` 变量设置 Looper 对象，这样也就理解了为什么要先调用 `Looper.prepare()` 方法，才能创建 Handler 对象，才不会导致崩溃。但是，仔细想想，为什么主线程就不用调用呢？不要急，我们接着分析一下主线程，我们查看一下 `ActivityThread` 中的 `main()` 方法，代码如下：

```
public static void main(String[] args) {  
  
    SamplingProfilerIntegration.start();
```

```
// CloseGuard defaults to true and can be quite spammy.  We
// disable it here, but selectively enable it later (via
// StrictMode) on debug builds, but using DropBox, not logs.

CloseGuard.setEnabled(false);

Environment.initForCurrentUser();

// Set the reporter for event logging in libcore
EventLogger.setReporter(new EventLoggingReporter());

Security.addProvider(new AndroidKeyStoreProvider());

// Make sure TrustedCertificateStore looks in the right place for CA
certificates

final File configDir = Environment.getUserConfigDirectory(UserHandle.
myUserId());

TrustedCertificateStore.setDefaultUserDirectory(configDir);

Process.setArgV0("<pre-initialized>");

Looper.prepareMainLooper();

ActivityThread thread = new ActivityThread();
```

```
        thread.attach(false);

    if (sMainThreadHandler == null) {

        sMainThreadHandler = thread.getHandler();

    }

    if (false) {

        Looper.myLooper().setMessageLogging(new
            LogPrinter(Log.DEBUG, "ActivityThread"));

    }

    Looper.loop();

}

throw new RuntimeException("Main thread loop unexpectedly exited");}
```

代码中调用了 Looper.prepareMainLooper()方法，而这个方法又会继续调用了 Looper.prepare()方法，代码如下：

```
public static void prepareMainLooper() {

    prepare(false);

    synchronized (Looper.class) {

        if (sMainLooper != null) {

            throw new IllegalStateException("The main Looper has already
been prepared.");

        }

    }

}
```

```
sMainLooper = myLooper();  
}  
}
```

分析到这里已经真相大白，主线程中 `google` 工程师已经自动帮我们创建了一个 `Looper` 对象了，因而我们不再需要手动再调用 `Looper.prepare()` 再创建，而子线程中，因为没有自动帮我们创建 `Looper` 对象，因此需要我们手动添加，调用方法是 `Looper.prepare()`，这样，我们才能正确地创建 `Handler` 对象。

发送消息

当我们正确的创建 `Handler` 对象后，接下来我们来了解一下怎么发送消息，有一点基础的朋友肯定对这个方法已经了如指掌了。具体是先创建出一个 `Message` 对象，然后可以利用一些方法，如 `setData()` 或者使用 `arg` 参数等方式来存放数据于消息中，再借助 `Handler` 对象将消息发送出去就可以了。

```
new Thread(new Runnable() {  
  
    @Override  
  
    public void run() {  
  
        Message msg = Message.obtain();  
  
        msg.arg1 = 1;  
  
        msg.arg2 = 2;  
  
        Bundle bundle = new Bundle();  
  
        bundle.putChar("key", 'v');  
  
        bundle.putString("key", "value");  
  
        msg.setData(bundle);  
  
        mHandler0.sendMessage(msg);  
  
    }  
  
}).start();
```

通过 `Message` 对象进行传递消息，在消息中添加各种数据，之后再消息通过 `mHandler0` 进行传递，之后我们再利用 `Handler` 中的 `handleMessage()` 方法将此时传递的 `Message` 进行捕获出来，再分析得到存储在 `msg` 中的数据。但是，这个

流程到底是怎么样的呢？具体我们还是来分析一下源码。首先分析一下发送方法 `sendMessage()`：

```
public final boolean sendMessage(Message msg){  
  
    return sendMessageDelayed(msg, 0);}
```

通过调用 `sendMessageDelayed(msg, 0)` 方法

```
public final boolean sendMessageDelayed(Message msg, long delayMillis){  
  
    if (delayMillis < 0) {  
  
        delayMillis = 0;  
  
    }  
  
    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);}
```

再能过调用 `sendMessageDelayed(Message msg, long delayMillis)`，方法中第一个参数是指发送的消息 `msg`, 第二个参数是指延迟多少毫秒发送，我们着重看一下此方法：

```
public boolean sendMessageAtTime(Message msg, long uptimeMillis) {  
  
    MessageQueue queue = mQueue;  
  
    if (queue == null) {  
  
        RuntimeException e = new RuntimeException(  
  
            this + " sendMessageAtTime() called with no mQueue");  
  
        Log.w("Looper", e.getMessage(), e);  
  
        return false;  
  
    }  
  
    return enqueueMessage(queue, msg, uptimeMillis);}
```

由这里可以分析得出，原来消息 `Message` 对象是建立一个消息队列 `MessageQueue`，这个对象 `MessageQueue` 由 `mQueue` 赋值，而由源码分析得出 `mQueue = mLooper.mQueue`，而 `mLooper` 则是 `Looper` 对象，我们由上面已经知道，每个线程只有一个 `Looper`，因此，一个 `Looper` 也就对应了一个 `MessageQueue` 对象，之后调用 `enqueueMessage(queue, msg, uptimeMillis)` 直接入队操作：

```
private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {

    msg.target = this;

    if (mAsynchronous) {

        msg.setAsynchronous(true);

    }

    return queue.enqueueMessage(msg, uptimeMillis);}


```

方法通过调用 MessageQueue 对 enqueueMessage(Message msg, long uptimeMills)方法:

```
boolean enqueueMessage(Message msg, long when) {

    if (msg.target == null) {

        throw new IllegalArgumentException("Message must have a target.");
    };

    if (msg.isInUse()) {

        throw new IllegalStateException(msg + " This message is already in use.");
    };

    synchronized (this) {

        if (mQuitting) {

            IllegalStateException e = new IllegalStateException(
                msg.target + " sending message to a Handler on a dead thread");

            Log.w("MessageQueue", e.getMessage(), e);
        }
    }
}
```

```
    msg.recycle();

    return false;
}

msg.markInUse();

msg.when = when;

Message p = mMessages;

boolean needWake;

if (p == null || when == 0 || when < p.when) {

    // New head, wake up the event queue if blocked.

    msg.next = p;

    mMessages = msg;

    needWake = mBlocked;

} else {

    // Inserted within the middle of the queue.  Usually we don't
    // have to wake

    // up the event queue unless there is a barrier at the head of
    // the queue

    // and the message is the earliest asynchronous message in the
    // queue.

    needWake = mBlocked && p.target == null && msg.isAsynchronous
();

    Message prev;

    for (;;) {

        prev = p;
```

```

    p = p.next;

    if (p == null || when < p.when) {
        break;
    }

    if (needWake && p.isAsynchronous()) {
        needWake = false;
    }
}

msg.next = p; // invariant: p == prev.next
prev.next = msg;
}
}

// We can assume mPtr != 0 because mQuitting is false.

if (needWake) {
    nativeWake(mPtr);
}
}

return true;
}

```

首先要知道，源码中用 `mMessages` 代表当前等待处理的消息，`MessageQueue` 也没有使用一个集合保存所有的消息。观察中间的代码部分，队列中根据时间 `when` 来时间排序，这个时间也就是我们传进来延迟的时间 `uptimeMillis` 参数，之后再根据时间的顺序调用 `msg.next`，从而指定下一个将要处理的消息是什么。如果只是通过 `sendMessageAtFrontOfQueue()` 方法来发送消息

```

public final boolean sendMessageAtFrontOfQueue(Message msg) {
    MessageQueue queue = mQueue;

```

```
if (queue == null) {

    RuntimeException e = new RuntimeException(
        this + " sendMessageAtTime() called with no mQueue");

    Log.w("Looper", e.getMessage(), e);

    return false;
}

return enqueueMessage(queue, msg, 0);}
```

它也是直接调用 `enqueueMessage()` 进行入队，但没有延迟时间，此时会将传递的此消息直接添加到队头处，现在入队操作已经了解得差不多了，接下来应该来了解一下出队操作，那么出队在哪里进行的呢，不要忘记 `MessageQueue` 对象是在 Looper 中赋值，因此我们可以在 Looper 类中找，来看一看 `Looper.loop()` 方法：

```
public static void loop() {

    final Looper me = myLooper();

    if (me == null) {

        throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");
    }

    final MessageQueue queue = me.mQueue;

    // Make sure the identity of this thread is that of the local process,
    // and keep track of what that identity token actually is.

    Binder.clearCallingIdentity();

    final long ident = Binder.clearCallingIdentity();
```

```
for (;;) {

    Message msg = queue.next(); // might block

    if (msg == null) {

        // No message indicates that the message queue is quitting.

        return;

    }

    // This must be in a local variable, in case a UI event sets the logger

    Printer logging = me.mLogging;

    if (logging != null) {

        logging.println(">>>> Dispatching to " + msg.target + " " +
                       msg.callback + ": " + msg.what);

    }

    msg.target.dispatchMessage(msg);

}

if (logging != null) {

    logging.println("<<<< Finished to " + msg.target + " " + msg.
callback);

}

// Make sure that during the course of dispatching the

// identity of the thread wasn't corrupted.
```

```
final long newIdent = Binder.clearCallingIdentity();

if (ident != newIdent) {

    Log.wtf(TAG, "Thread identity changed from 0x"

        + Long.toHexString(ident) + " to 0x"

        + Long.toHexString(newIdent) + " while dispatching to

"
        + msg.target.getClass().getName() + " "

        + msg.callback + " what=" + msg.what);

}

msg.recycleUnchecked();

}}
```

代码比较多，我们只挑重要的分析一下，我们可以看到下面的代码用 `for(;;)` 进入了一个死循环，之后不断的从 MessageQueue 对象 queue 中取出消息 msg，而我们不难知道，此时的 `next()` 就是进行队列的出队方法，`next()` 方法代码有点长，有兴趣的话可以自行翻阅查看，主要逻辑是判断当前的 MessageQueue 是否存在待处理的 `mMessages` 消息，如果有，则将这个消息出队，然后让下一个消息成为 `mMessages`，否则就进入一个阻塞状态，一直等到有新的消息入队唤醒。回看 `loop()` 方法，可以发现当执行 `next()` 方法后会执行 `msg.target.dispatchMessage(msg)` 方法，而不难看出，此时 `msg.target` 就是 Handler 对象，继续看一下 `dispatchMessage()` 方法：

```
public void dispatchMessage(Message msg) {

    if (msg.callback != null) {

        handleCallback(msg);

    } else {

        if (mCallback != null) {

            if (mCallback.handleMessage(msg)) {

                return;
            }
        }
    }
}
```

```
        }

    }

    handleMessage(msg);

}}
```

先进行判断 mCallback 是否为空, 若不为空则调用 mCallback 的 handleMessage() 方法, 否则直接调用 handleMessage() 方法, 并将消息作为参数传出去。这样我们就完全一目了然, 为什么我们要使用 handleMessage() 来捕获我们之前传递过去的信息。

现在我们根据上面的理解, 不难写出异步消息处理机制的线程了。

```
class myThread extends Thread{

    public Handler myHandler;

    @Override

    public void run() {

        Looper.prepare();

        myHandler = new Handler(){

            @Override

            public void handleMessage(Message msg) {

                super.handleMessage(msg);

                //处理消息

            }

        };

        Looper.loop();

    }

}}
```

当然除了发送消息外, 还有以下几个方法可以在子线程中进行 UI 操作:

- View 的 post()方法
- Handler 的 post()方法
- Activity 的 runOnUiThread()方法

其实这几个方法的本质都是一样的，只要我们勤于查看这几个方法的源码，不难看出最后调用的也是 Handler 机制，也是借用了异步消息处理机制来实现的。

总结

通过上面对异步消息处理线程的讲解，我们不难真正地理解到了 Handler、Looper 以及 Message 之间的关系，概括性来说，Looper 负责的是创建一个 MessageQueue 对象，然后进入到一个无限循环体中不断取出消息，而这些消息都是由一个或者多个 Handler 进行创建处理。

2、Messagequeue 的数据结构是什么？为什么要用这个数据结构？

为什么要用 Message Queue

•

解耦

在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息队列在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。这允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束

•

•

冗余

有些情况下，处理数据的过程会失败。除非数据被持久化，否则将造成丢失。消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。在被许多消息队列所采用的“插入-获取-删除”范式中，在把一个消息从队列中删除之前，需要你的处理过程明确的指出该消息已经被处理完毕，确保你的数据被安全的保存直到你使用完毕。

•

•

扩展性

因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的；只要另外增加处理过程即可。不需要改变代码、不需要调节参数。扩展就像调大电力按钮一样简单。

•

•

灵活性 & 峰值处理能力

在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见；如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

-
-

可恢复性

当体系的一部分组件失效，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。而这种允许重试或者延后处理请求的能力通常是造就一个略感不便的用户和一个沮丧透顶的用户之间的区别。

-
-

送达保证

消息队列提供的冗余机制保证了消息能被实际的处理，只要一个进程读取了该队列即可。在此基础上，IronMQ 提供了一个“只送达一次”保证。无论有多少进程在从队列中领取数据，每一个消息只能被处理一次。这之所以成为可能，是因为获取一个消息只是“预定”了这个消息，暂时把它移出了队列。除非客户端明确的表示已经处理完了这个消息，否则这个消息会被放回队列中去，在一段可配置的时间之后可再次被处理。

-
-

顺序保证

在大多使用场景下，数据处理的顺序都很重要。消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。IronMQ 保证消息通过 FIFO（先进先出）的顺序来处理，因此消息在队列中的位置就是从队列中检索他们的位置。

-
-

缓冲

在任何重要的系统中，都会有需要不同的处理时间的元素。例如，加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率的执行—写入队列的处理会尽可能的快速，而不受从队列读的预备处理的约束。该缓冲有助于控制和优化数据流经过系统的速度。

-
-

理解数据流

在一个分布式系统里，要得到一个关于用户操作会用多长时间及其原因的总体印象，是个巨大的挑战。消息系列通过消息被处理的频率，来方便的辅助确定那些表现不佳的处理过程或领域，这些地方的数据流都不够优化。

•
•

异步通信

很多时候，你不想也不需要立即处理消息。消息队列提供了异步处理机制，允许你把一个消息放入队列，但并不立即处理它。你想向队列中放入多少消息就放多少，然后在你乐意的时候再去处理它们。

Messagequeue 的数据结构是什么？

基础数据结构中“先进先出”的一种数据结构

3、如何在子线程中创建 Handler?

在子线程中创建 handler，要确保子线程有 Looper,UI 线程默认包含 Looper

我们需要用到一个特殊类

HandlerThread

这个类可以轻松的创建子线程 handler

创建步骤：

1：创建一个 HandlerThread，即创建一个包含 Looper 的线程

HandlerThread 的构造函数有两个

```
public HandlerThread(String name) {  
    super(name);  
    mPriority = Process.THREAD_PRIORITY_DEFAULT;  
}  
  
/**  
 * Constructs a HandlerThread.  
 * @param name  
 * @param priority The priority to run the thread at. The value supplied must be from  
 * {@link android.os.Process} and not from java.lang.Thread.  
 */  
public HandlerThread(String name, int priority) {
```

```
    super(name);
    mPriority = priority;
}
```

这里我们使用第一个就好：

```
HandlerThread handlerThread=new HandlerThread("xuan");
```

```
handlerThread.start();//创建 HandlerThread 后一定要记得 start();
```

通过 HandlerThread 的 getLooper 方法可以获取 Looper

```
Looper looper=handlerThread.getLooper();
```

通过 Looper 我们就可以创建子线程的 handler 了

```
Handlr handler=new Handler(looper);
```

通过该 handler 发送消息，就会在子线程执行；

提示：如果要 handlerThread 停止：handlerThread.quit();

完整测试代码：

```
HandlerThread hanlerThread = new HandlerThread("子线程");
hanlerThread.start();
final Handler handler = new Handler(hanlerThread.getLooper()) {
    @Override
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        Log.d("---->", "线程：" + Thread.currentThread().getName());
    }
};
```

```
        findViewById(R.id.bt).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                handler.sendEmptyMessage(100);
            }
        });
    
```

结果：

```
12-24 10:13:15.881 5024-5052/gitosctest.gitosc_studyproject D/---->: 线程:子线程
```

像在 intentService(子线程) 中, 如果要回掉在 UI 线程怎么办呢?

```
new Handler(getMainLooper()).post(new Runnable() {
    @Override
    public void run() {
        // person.getName() Realm objects can only be accessed on
        the thread they were created.
        Toast.makeText(getApplicationContext(), "Loaded Person from
        broadcast-receiver->intent-service: " + info, Toast.LENGTH_LONG).show();
    }
});
```

4、Handler post 方法原理?

一. 源码分析

1.点进去看 postDelayed()中的方法。里面调用 sendMessageDelayed 方法，和 post() 里面调用的方法一样。

```
public final boolean postDelayed(Runnable r, long delayMillis)
{
    return sendMessageDelayed(getPostMessage(r), delayMillis);
}
```

2. 我们再点进去看下 sendMessageDelayed()方法,

```
public final boolean sendMessageDelayed(Message msg, long delayMillis)
```

```
{  
  
    if (delayMillis < 0) {  
  
        delayMillis = 0;  
  
    }  
  
    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);  
  
}
```

里面调用了 `sendMessageAtTime()`，这里的 `SystemClock.uptimeMillis()` 是获取系统从开机启动到现在的时间，期间不包括休眠的时间，这里获得到的时间是一个相对的时间，而不是通过获取当前的时间（绝对时间）。

而之所以使用这种方式来计算时间，而不是获得当前 `currenttime` 来计算，在于 `handler` 会受到阻塞，挂起状态，睡眠等，这些时候是不应该执行的；如果使用绝对时间的话，就会抢占资源来执行当前 `handler` 的内容，显然这是不应该出现的情况，所以要避免。

3. 点进 `sendMessageAtTime()` 方法看看

```
public boolean sendMessageAtTime(Message msg, long uptimeMillis) {  
  
    MessageQueue queue = mQueue;  
  
    if (queue == null) {  
  
        RuntimeException e = new RuntimeException(  
            this + " sendMessageAtTime() called with no mQueue");  
  
        Log.w("Looper", e.getMessage(), e);  
  
        return false;  
  
    }  
  
    return enqueueMessage(queue, msg, uptimeMillis);  
}
```

追到这里依然没有看到，他在存放的时候有什么不同，但是显然证实了消息不是延迟放进 `MessageQueen` 的，那是肿么处理的，是在轮训的时候处理的吗？，

4. 我们点进 `Looper` 中看一下，主要代码，`Looper` 中的 `looper` 调用了 `MessageQueen` 中的 `next` 方法，难道是在 `next()` 方法中处理的？

```
public static void loop() {  
    ...  
  
    for (;;) {  
  
        Message msg = queue.next(); // might block  
  
        if (msg == null) {  
  
            // No message indicates that the message queue is quitting.  
            return;  
  
        }  
  
        ...  
    }  
}
```

5. 我们点进 MessageQueen 中的 next()方法

```
for (;;) {  
  
    if (nextPollTimeoutMillis != 0) {  
  
        Binder.flushPendingCommands();  
  
    }  
  
    nativePollOnce(ptr, nextPollTimeoutMillis);  
  
    ...  
  
    if (msg != null) {  
  
        if (now < msg.when) {  
  
            // Next message is not ready. Set a timeout to wake up when it is ready.  
        }  
    }  
}
```

```
        nextPollTimeoutMillis = (int) Math.min(msg.when -  
now, Integer.MAX_VALUE);  
  
    } else {  
  
        // Got a message.  
  
        mBlocked = false;  
  
        if (prevMsg != null) {  
  
            prevMsg.next = msg.next;  
  
        } else {  
  
            mMessages = msg.next;  
  
        }  
  
        msg.next = null;  
  
        if (DEBUG) Log.v(TAG, "Returning message: " + ms  
g);  
  
        msg.markInUse();  
  
        return msg;  
  
    }  
  
} else {  
  
    // No more messages.  
  
    nextPollTimeoutMillis = -1;  
  
}  
  
...  

```

```
}}
```

很贴心的给出了注释解释“`Next message is not ready. Set a timeout to wake up when it is ready.`”，翻译“下一条消息尚未准备好。设置一个超时，以便在准备就绪时唤醒。”

`when` 就是 `uptimeMillis`, `for (;;)相当与 while(true)`, 如果头部的这个 `Message` 是有延迟而且延迟时间没到的 (`now < msg.when`) , 不返回 `message` 而且会计算一下时间 (保存为变量 `nextPollTimeoutMillis`) , 然后再循环的时候判断如果这个 `Message` 有延迟, 就调用 `nativePollOnce(ptr, nextPollTimeoutMillis)`进行阻塞。`nativePollOnce()`的作用类似与 `object.wait()`。得出结论是通过阻塞实现的。

6.但是如果在阻塞这段时间里有无延迟 `message` 又加入 `MessageQueen` 中又是怎么实现立即处理这个 `message` 的呢? , 我们看 `MessageQueen` 中放入消息 `enqueueMessage()`方法

```
boolean enqueueMessage(Message msg, long when) {  
  
    if (msg.target == null) {  
  
        throw new IllegalArgumentException("Message must have a target.");  
  
    }  
  
    if (msg.isInUse()) {  
  
        throw new IllegalStateException(msg + " This message is already in use.");  
  
    }  
  
    synchronized (this) {  
  
        if (mQuitting) {  
  
            IllegalStateException e = new IllegalStateException(  
  
                msg.target + " sending message to a Handler on a dead thread");  
  
            Log.w(TAG, e.getMessage(), e);  
  
            msg.recycle();  
  
        }  
    }  
}
```

```
}

msg.markInUse();

msg.when = when;

Message p = mMessages;

boolean needWake;

if (p == null || when == 0 || when < p.when) {

    // New head, wake up the event queue if blocked.

    msg.next = p;

    mMessages = msg;

    needWake = mBlocked;

} else {

    // Inserted within the middle of the queue. Usually we do
    // n't have to wake

    // up the event queue unless there is a barrier at the hea
    d of the queue

    // and the message is the earliest asynchronous message in
    // the queue.

    needWake = mBlocked && p.target == null && msg.isAsynchron
    ous();

    Message prev;

    for (;;) {

        prev = p;

        p = p.next;

        if (p == null || when < p.when) {
```

```

        break;

    }

    if (needWake && p.isAsynchronous()) {

        needWake = false;

    }

    msg.next = p; // invariant: p == prev.next

    prev.next = msg;

}

// We can assume mPtr != 0 because mQuitting is false.

if (needWake) {

    nativeWake(mPtr);

}

return true;

}

```

在这里 p 是现在消息队列中的头部消息，我们看到 | when < p.when 的时候它交换了放入 message 与原来消息队列头部 P 的位置，并且 needWake = mBlocked; (在 next() 中当消息为延迟消息的时候 mBlocked=true)，继续向下看 当 needWake =true 的时候 nativeWake(mPtr) (唤起线程)
一切都解释的通了，如果当前插入的消息不是延迟 message，或比当前的延迟短，这个消息就会插入头部并且唤起线程来

二. 整理

我们把我们跟踪的所有信息整理下

1. 消息是通过 MessageQueue 中的 enqueueMessage() 方法加入消息队列中的，并且它在放入中就进行好排序，链表头的延迟时间小，尾部延迟时间最大
2. Looper.loop() 通过 MessageQueue 中的 next() 去取消息
3. next() 中如果当前链表头部消息是延迟消息，则根据延迟时间进行消息队列会阻塞，不返回给 Looper message，知道时间到了，返回给 message
4. 如果在阻塞中有新的消息插入到链表头部则唤醒线程
5. Looper 将新消息交给回调给 handler 中的 handleMessage 后，继续调用 MessageQueue 的 next() 方法，如果刚刚的延迟消息还是时间未到，则计算时间继续阻塞

三总结

handler.postDelay() 的实现 是通过 MessageQueue 中执行时间顺序排列，消息队列阻塞，和唤醒的方式结合实现的。

如果真的是通过延迟将消息放入到 MessageQueue 中，那放入多个延迟消息就要维护多个定时器，

5、Android 消息机制的原理及源码解析

一、消息机制概述

1. 消息机制的简介

在 Android 中使用消息机制，我们首先想到的就是 Handler。没错，Handler 是 Android 消息机制的上层接口。Handler 的使用过程很简单，通过它可以轻松地将一个任务切换到 Handler 所在的线程中去执行。通常情况下，Handler 的使用场景就是更新 UI。

如下就是使用消息机制的一个简单实例：

```
public class Activity extends android.app.Activity {  
  
    private Handler mHandler = new Handler(){  
  
        @Override  
  
        public void handleMessage(Message msg) {  
  
            super.handleMessage(msg);  
  
            System.out.println(msg.what);  
  
        }  
  
    };
```

```
    @Override

    public void onCreate(Bundle savedInstanceState, PersistableBundle persistentState) {

        super.onCreate(savedInstanceState, persistentState);

        setContentView(R.layout.activity_main);

        new Thread(new Runnable() {

            @Override

            public void run() {

                .....耗时操作

                Message message = Message.obtain();

                message.what = 1;

                mHandler.sendMessage(message);

            }

        }).start();

    }

}
```

在子线程中，进行耗时操作，执行完操作后，发送消息，通知主线程更新 UI。这便是消息机制的典型应用场景。我们通常只会接触到 Handler 和 Message 来完成消息机制，其实内部还有两大助手来共同完成消息传递。

2. 消息机制的模型

消息机制主要包含：MessageQueue, Handler 和 Looper 这三大部分，以及 Message，下面我们一一介绍。

Message: 需要传递的消息，可以传递数据；

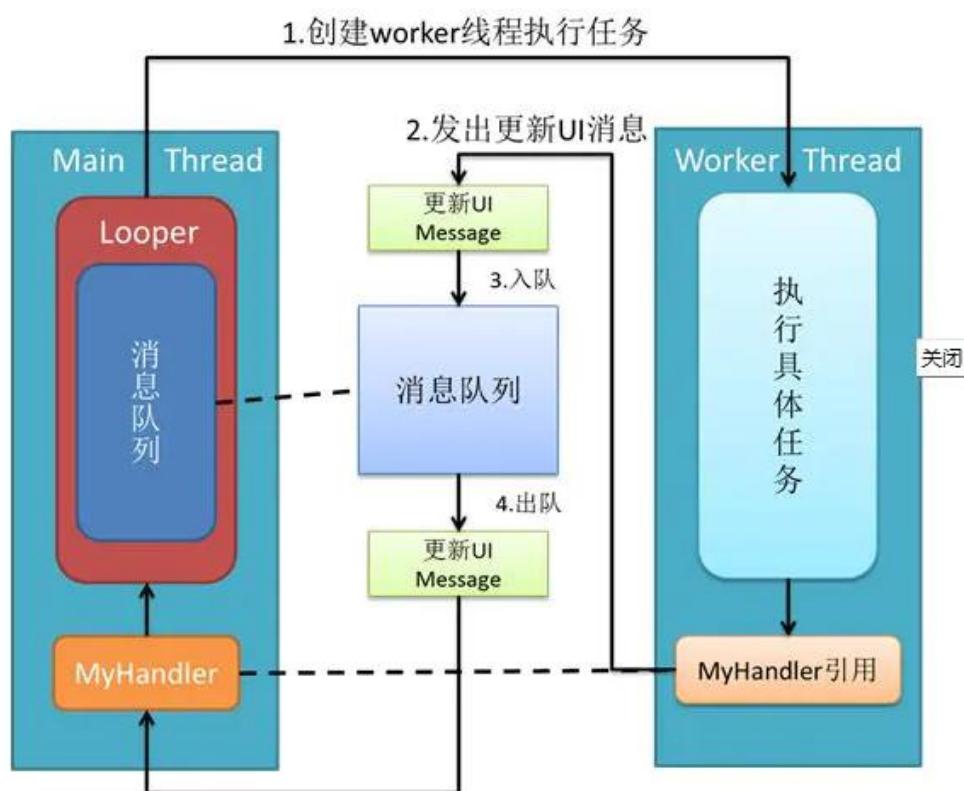
MessageQueue: 消息队列，但是它的内部实现并不是用的队列，实际上是通过一个单链表的数据结构来维护消息列表，因为单链表在插入和删除上比较有优势。主要功能向消息池投递消息(MessageQueue.enqueueMessage)和取走消息池的消息(MessageQueue.next);

Handler: 消息辅助类，主要功能向消息池发送各种消息事件(Handler.sendMessage)和处理相应消息事件(Handler.handleMessage);

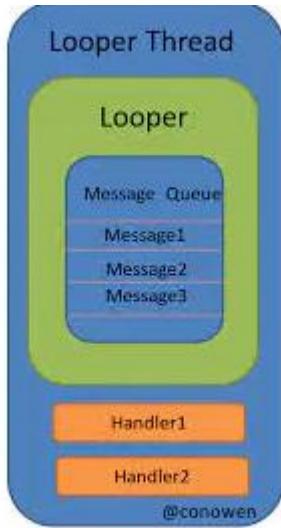
Looper: 不断循环执行(Looper.loop), 从 MessageQueue 中读取消息, 按分发机制将消息分发给目标处理者。

3. 消息机制的架构

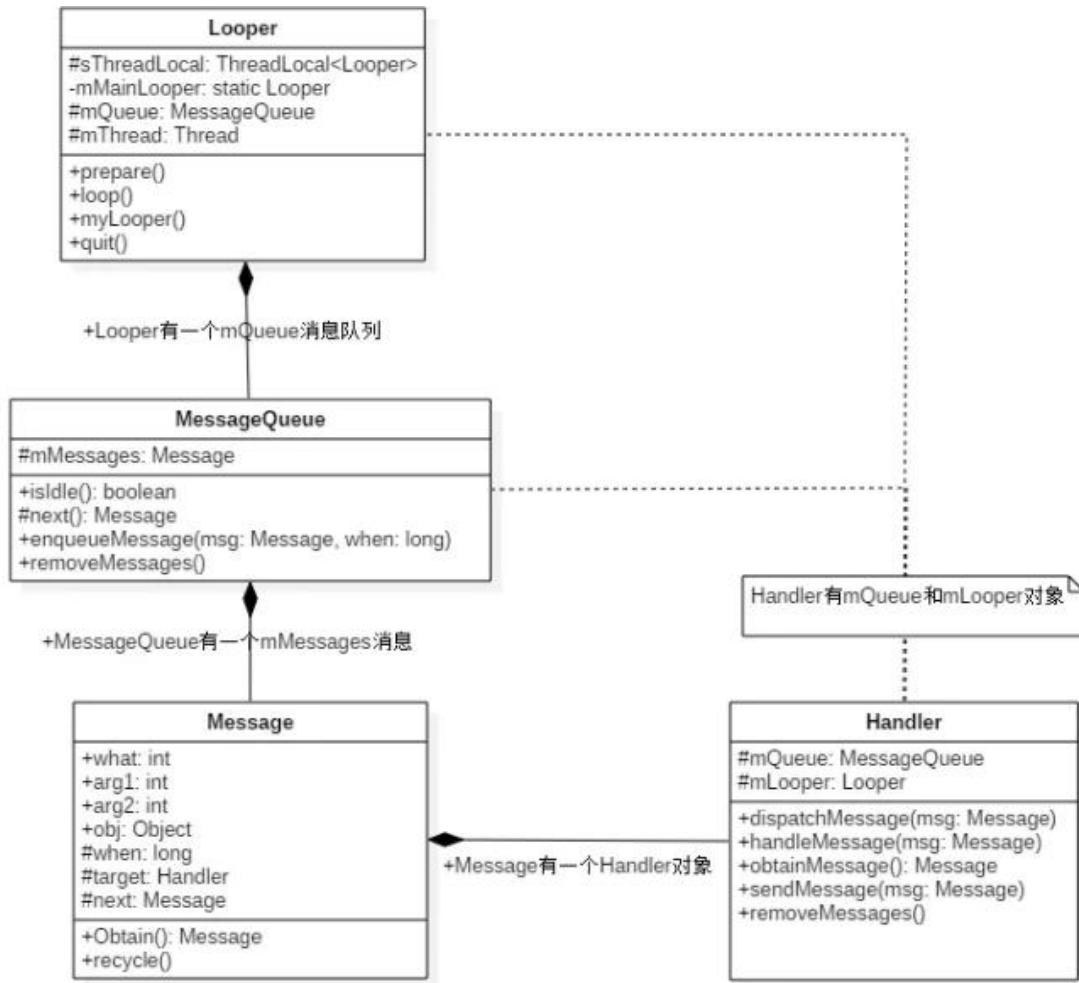
消息机制的运行流程: 在子线程执行完耗时操作, 当 Handler 发送消息时, 将会调用 `MessageQueue.enqueueMessage`, 向消息队列中添加消息。当通过 `Looper.loop` 开启循环后, 会不断地从线程池中读取消息, 即调用 `MessageQueue.next`, 然后调用目标 Handler (即发送该消息的 Handler) 的 `dispatchMessage` 方法传递消息, 然后返回到 Handler 所在线程, 目标 Handler 收到消息, 调用 `handleMessage` 方法, 接收消息, 处理消息。



MessageQueue, Handler 和 Looper 三者之间的关系: 每个线程中只能存在一个 Looper, Looper 是保存在 ThreadLocal 中的。主线程 (UI 线程) 已经创建了一个 Looper, 所以在主线程中不需要再创建 Looper, 但是在其他线程中需要创建 Looper。每个线程中可以有多个 Handler, 即一个 Looper 可以处理来自多个 Handler 的消息。Looper 中维护一个 MessageQueue, 来维护消息队列, 消息队列中的 Message 可以来自不同的 Handler。



下面是消息机制的整体架构图，接下来我们将慢慢解剖整个架构。



从中我们可以看出：

- Looper 有一个 MessageQueue 消息队列；
- MessageQueue 有一组待处理的 Message；
- Message 中记录发送和处理消息的 Handler；
- Handler 中有 Looper 和 MessageQueue。

二、消息机制的源码解析

1. Looper

要想使用消息机制，首先要创建一个 Looper。

初始化 Looper

无参情况下，默认调用 `prepare(true)`；表示的是这个 Looper 可以退出，而对于 `false` 的情况则表示当前 Looper 不可以退出。。

这里看出，不能重复创建 Looper，只能创建一个。创建 Looper，并保存在 ThreadLocal。其中 ThreadLocal 是线程本地存储区（Thread Local Storage，简称为 TLS），每个线程都有自己的私有的本地存储区域，不同线程之间彼此不能访问对方的 TLS 区域。

开启 Looper

```
public static void loop() {  
  
    final Looper me = myLooper(); //获取 TLS 存储的 Looper 对象  
  
    if (me == null) {  
  
        throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");  
  
    }  
}
```

```
final MessageQueue queue = me.mQueue; //获取 Looper 对象中的消息队列

Binder.clearCallingIdentity();

final long ident = Binder.clearCallingIdentity();

for (;;) { //进入 loop 的主循环方法

    Message msg = queue.next(); //可能会阻塞,因为 next()方法可能会无限循
环

    if (msg == null) { //消息为空, 则退出循环

        return;

    }

    Printer logging = me.mLogging; //默认为 null, 可通过 setMessageLog
ging()方法来指定输出, 用于 debug 功能

    if (logging != null) {

        logging.println(">>>> Dispatching to " + msg.target + " " +
msg.callback + ": " + msg.what);

    }

    msg.target.dispatchMessage(msg); //获取 msg 的目标 Handler, 然后用于
分发 Message

    if (logging != null) {

        logging.println("<<<< Finished to " + msg.target + " " + msg.
callback);

    }

}
```

```
final long newIdent = Binder.clearCallingIdentity();

if (ident != newIdent) {

}

msg.recycleUnchecked();

}}
```

loop()进入循环模式，不断重复下面的操作，直到消息为空时退出循环：

读取 MessageQueue 的下一条 Message（关于 next()，后面详细介绍）；

把 Message 分发给相应的 target。

当 next()取出下一条消息时，队列中已经没有消息时，next()会无限循环，产生阻塞。等待 MessageQueue 中加入消息，然后重新唤醒。

主线程中不需要自己创建 Looper，这是由于在程序启动的时候，系统已经帮我们自动调用了 Looper.prepare()方法。查看 ActivityThread 中的 main()方法，代码如下所示：

```
public static void main(String[] args) {.....  
  
    Looper.prepareMainLooper();  
  
    .....  
  
    Looper.loop();  
  
    .....  
  
}
```

其中`prepareMainLooper()`方法会调用 `prepare(false)`方法。

2. Handler

创建 Handler

```
public Handler() {  
  
    this(null, false);}
```

```
public Handler(Callback callback, boolean async) {  
    .....  
    //必须先执行 Looper.prepare(), 才能获取 Looper 对象, 否则为 null.  
    mLooper = Looper.myLooper(); //从当前线程的 TLS 中获取 Looper 对象  
  
    if (mLooper == null) {  
  
        throw new RuntimeException("");  
    }  
  
    mQueue = mLooper.mQueue; //消息队列, 来自 Looper 对象  
    mCallback = callback; //回调方法  
  
    mAsynchronous = async; //设置消息是否为异步处理方式}  
}
```

对于 Handler 的无参构造方法，默认采用当前线程 TLS 中的 Looper 对象，并且 callback 回调方法为 null，且消息为同步处理方式。只要执行的 Looper.prepare() 方法，那么便可以获取有效的 Looper 对象。

3. 发送消息

发送消息有几种方式，但是归根结底都是调用了 sendMessageAtTime() 方法。
在子线程中通过 Handler 的 post() 方式或 send() 方式发送消息，最终都是调用了 sendMessageAtTime() 方法。

post 方法

```
public final boolean post(Runnable r)  
{  
  
    return sendMessageDelayed(getPostMessage(r), 0);  
}  
public final boolean postAtTime(Runnable r, long uptimeMillis)  
{  
  
    return sendMessageAtTime(getPostMessage(r), uptimeMillis);  
}
```

```
public final boolean postAtTime(Runnable r, Object token, long uptimeMillis)

{
    return sendMessageAtTime(getPostMessage(r, token), uptimeMillis);
}

public final boolean postDelayed(Runnable r, long delayMillis)

{
    return sendMessageDelayed(getPostMessage(r), delayMillis);
}
```

send 方法

```
public final boolean sendMessage(Message msg)

{
    return sendMessageDelayed(msg, 0);
}

public final boolean sendEmptyMessage(int what)

{
    return sendEmptyMessageDelayed(what, 0);
}

} public final boolean sendEmptyMessageDelayed(int what, long delayMillis) {

    Message msg = Message.obtain();

    msg.what = what;

    return sendMessageDelayed(msg, delayMillis);
}
```

```
public final boolean sendEmptyMessageAtTime(int what, long uptimeMillis)
{
    Message msg = Message.obtain();
    msg.what = what;
    return sendMessageAtTime(msg, uptimeMillis);
}

public final boolean sendMessageDelayed(Message msg, long delayMillis)
{
    if (delayMillis < 0) {
        delayMillis = 0;
    }
    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
}
```

就连子线程中调用 Activity 中的 runOnUiThread() 中更新 UI，其实也是发送消息通知主线程更新 UI，最终也会调用 `sendMessageAtTime()` 方法。

```
public final void runOnUiThread(Runnable action) {
    if (Thread.currentThread() != mUiThread) {
        mHandler.post(action);
    } else {
        action.run();
    }
}
```

如果当前的线程不等于 UI 线程(主线程), 就去调用 Handler 的 post()方法, 最终会调用 sendMessageAtTime()方法。否则就直接调用 Runnable 对象的 run()方法。下面我们就来一探究竟, 到底 sendMessageAtTime()方法有什么作用?

sendMessageAtTime()

```
public boolean sendMessageAtTime(Message msg, long uptimeMillis) {  
  
    //其中 mQueue 是消息队列, 从 Looper 中获取的  
  
    MessageQueue queue = mQueue;  
  
    if (queue == null) {  
  
        RuntimeException e = new RuntimeException(  
            this + " sendMessageAtTime() called with no mQueue");  
  
        Log.w("Looper", e.getMessage(), e);  
  
        return false;  
    }  
  
    //调用 enqueueMessage 方法  
  
    return enqueueMessage(queue, msg, uptimeMillis);  
}  
  
private boolean enqueueMessage(MessageQueue queue, Message msg, long up  
timeMillis) {  
  
    msg.target = this;  
  
    if (mAsynchronous) {  
  
        msg.setAsynchronous(true);  
    }  
  
    //调用 MessageQueue 的 enqueueMessage 方法  
  
    return queue.enqueueMessage(msg, uptimeMillis);  
}
```

可以看到 `sendMessageAtTime()` 方法的作用很简单，就是调用 `MessageQueue` 的 `enqueueMessage()` 方法，往消息队列中添加一个消息。

下面来看 `enqueueMessage()` 方法的具体执行逻辑。

enqueueMessage()

```
boolean enqueueMessage(Message msg, long when) {  
  
    // 每一个 Message 必须有一个 target  
  
    if (msg.target == null) {  
  
        throw new IllegalArgumentException("Message must have a target.  
    ");  
  
    }  
  
    if (msg.isInUse()) {  
  
        throw new IllegalStateException(msg + " This message is already i  
n use.");  
  
    }  
  
    synchronized (this) {  
  
        if (mQuitting) { // 正在退出时，回收 msg，加入到消息池  
  
            msg.recycle();  
  
            return false;  
  
        }  
  
        msg.markInUse();  
  
        msg.when = when;  
  
        Message p = mMessages;  
  
        boolean needWake;  
  
        if (p == null || when == 0 || when < p.when) {  
  
            // p 为 null(代表 MessageQueue 没有消息) 或者 msg 的触发时间是队列中  
            最早的，则进入该分支
```

```
msg.next = p;

mMessages = msg;

needWake = mBlocked;

} else {

    //将消息按时间顺序插入到 MessageQueue。一般地，不需要唤醒事件队列，  
除非

    //消息队头存在 barrier，并且同时 Message 是队列中最早的异步消息。

    needWake = mBlocked && p.target == null && msg.isAsynchronous()  
();

    Message prev;

    for (;;) {

        prev = p;

        p = p.next;

        if (p == null || when < p.when) {

            break;
        }

        if (needWake && p.isAsynchronous()) {

            needWake = false;
        }
    }

    msg.next = p;

    prev.next = msg;
}

if (needWake) {
```

```
    nativeWake(mPtr);

}

return true;
}
```

MessageQueue 是按照 Message 触发时间的先后顺序排列的，队头的消息是将要最早触发的消息。当有消息需要加入消息队列时，会从队列头开始遍历，直到找到消息应该插入的合适位置，以保证所有消息的时间顺序。

4. 获取消息

当发送了消息后，在 MessageQueue 维护了消息队列，然后在 Looper 中通过 loop() 方法，不断地获取消息。上面对 loop() 方法进行了介绍，其中最重要的是调用了 queue.next() 方法，通过该方法来提取下一条信息。下面我们来看一下 next() 方法的具体流程。

next()

```
Message next() {

    final long ptr = mPtr;

    if (ptr == 0) { //当消息循环已经退出，则直接返回

        return null;
    }

    int pendingIdleHandlerCount = -1; // 循环迭代的首次为-1

    int nextPollTimeoutMillis = 0;

    for (;;) {

        if (nextPollTimeoutMillis != 0) {

            Binder.flushPendingCommands();
        }

        //阻塞操作，当等待 nextPollTimeoutMillis 时长，或者消息队列被唤醒，都会返回

        nativePollOnce(ptr, nextPollTimeoutMillis);
    }
}
```

```
synchronized (this) {  
  
    final long now = SystemClock.uptimeMillis();  
  
    Message prevMsg = null;  
  
    Message msg = mMessages;  
  
    if (msg != null && msg.target == null) {  
  
        //当消息 Handler 为空时，查询 MessageQueue 中的下一条异步消息 msg，为空则退出循环。  
  
        do {  
  
            prevMsg = msg;  
  
            msg = msg.next;  
  
        } while (msg != null && !msg.isAsynchronous());  
  
    }  
  
    if (msg != null) {  
  
        if (now < msg.when) {  
  
            //当异步消息触发时间大于当前时间，则设置下一次轮询的超时时间  
            nextPollTimeoutMillis = (int) Math.min(msg.when - now,  
Integer.MAX_VALUE);  
  
        } else {  
  
            // 获取一条消息，并返回  
            mBlocked = false;  
  
            if (prevMsg != null) {  
  
                prevMsg.next = msg.next;  
  
            } else {
```

```
mMessages = msg.next;

}

msg.next = null;

//设置消息的使用状态，即 flags |= FLAG_IN_USE

msg.markInUse();

return msg; //成功地获取 MessageQueue 中的下一条即将要执行的消息

}

} else {

//没有消息

nextPollTimeoutMillis = -1;

}

//消息正在退出，返回 null

if (mQuitting) {

dispose();

return null;

}

.....



}}
```

nativePollOnce 是阻塞操作，其中 nextPollTimeoutMillis 代表下一个消息到来前，还需要等待的时长；当 nextPollTimeoutMillis = -1 时，表示消息队列中无消息，会一直等待下去。

可以看出 next()方法根据消息的触发时间，获取下一条需要执行的消息，队列中消息为空时，则会进行阻塞操作。

5. 分发消息

在 `loop()` 方法中，获取到下一条消息后，执行 `msg.target.dispatchMessage(msg)`，来分发消息到目标 `Handler` 对象。

下面就来具体看下 `dispatchMessage(msg)` 方法的执行流程。

`dispatchMessage()`

```
public void dispatchMessage(Message msg) {  
  
    if (msg.callback != null) {  
  
        //当 Message 存在回调方法，回调 msg.callback.run()方法;  
  
        handleCallback(msg);  
  
    } else {  
  
        if (mCallback != null) {  
  
            //当 Handler 存在 Callback 成员变量时，回调方法 handleMessage();  
  
            if (mCallback.handleMessage(msg)) {  
  
                return;  
  
            }  
  
        }  
  
        //Handler 自身的回调方法 handleMessage()  
  
        handleMessage(msg);  
  
    }  
  
}  
  
private static void handleCallback(Message message) {  
  
    message.callback.run();  
  
}
```

分发消息流程：

当 `Message` 的 `msg.callback` 不为空时，则回调方法 `msg.callback.run()`；

当 `Handler` 的 `mCallback` 不为空时，则回调方法 `mCallback.handleMessage(msg)`；

最后调用 `Handler` 自身的回调方法 `handleMessage()`，该方法默认为空，`Handler` 子类通过覆写该方法来完成具体的逻辑。

消息分发的优先级：

`Message` 的回调方法：`message.callback.run()`，优先级最高；

Handler 中 Callback 的回调方法: `Handler.mCallback.handleMessage(msg)`, 优先级仅次于 1;

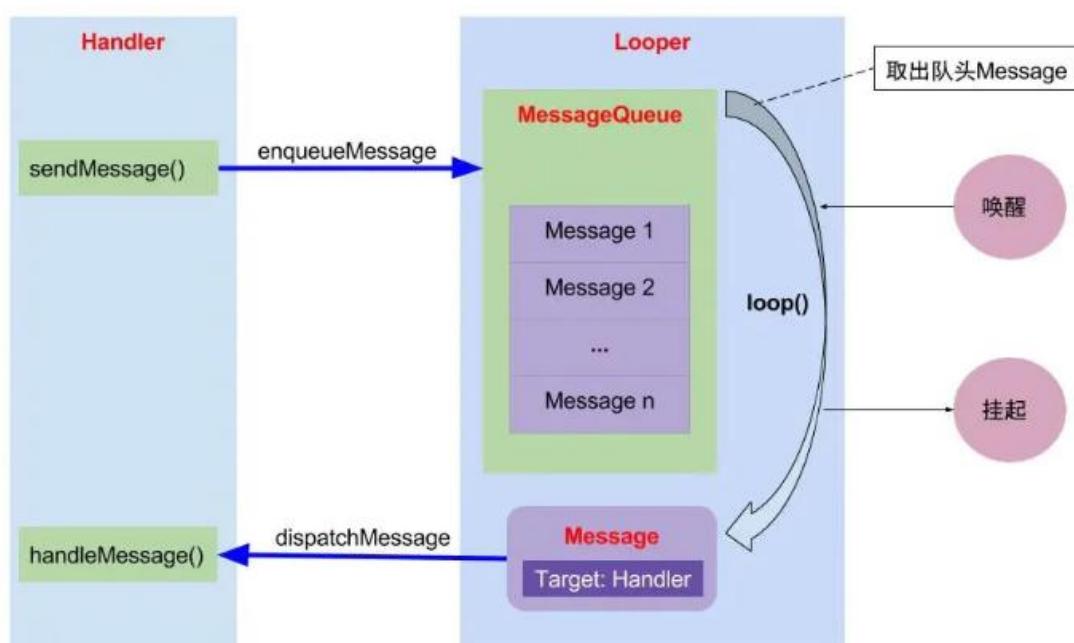
Handler 的默认方法: `Handler.handleMessage(msg)`, 优先级最低。

对于很多情况下, 消息分发后的处理方法是第 3 种情况, 即

`Handler.handleMessage()`, 一般地往往通过覆写该方法从而实现自己的业务逻辑。

三、总结

以上便是消息机制的原理, 以及从源码角度来解析消息机制的运行过程。可以简单地用下图来理解。



6、Handler 都没搞懂，拿什么去跳槽啊？

0. 前言

做 Android 开发肯定离不开跟 Handler 打交道, 它通常被我们用来做主线程与子线程之间的通信工具, 而 Handler 作为 Android 中消息机制的重要一员也确实给我们的开发带来了极大的便利。

可以说只要有异步线程与主线程通信的地方就一定会有 Handler。

那么, Handler 的通信机制背后的原理是什么?

本文带你揭晓。

注意：本文所展示的系统源码基于 Android-27，并有所删减。

[](#)

1. 重识 Handler

我们可以使用 Handler **发送并处理**与一个线程关联的 Message 和 Runnable 。

(注意: Runnable 会被封装进一个 Message, 所以它本质上还是一个 Message)

每个 Handler 都会跟一个线程绑定, 并与该线程的 MessageQueue 关联在一起, 从而实现消息的管理以及线程间通信。

1. 1 Handler 的基本用法

```
android.os.Handler handler = new Handler(){

    @Override

    public void handleMessage(final Message msg) {

        //这里接受并处理消息

    };//发送消息

    handler.sendMessage(message);

    handler.post(runnable);
}
```

实例化一个 Handler 重写 handleMessage 方法 , 然后在需要的时候调用它的 send 以及 post **系列方法**就可以了, 非常简单易用, 并且支持延时消息。 (更多方法可查询 API 文档)

但是奇怪, 我们并没有看到任何 MessageQueue 的身影, 也没看到它与线程绑定的逻辑, 这是怎么回事?

2. Handler 原理解析

相信大家早就听说过了 Looper 以及 MessageQueue 了, 我就不多绕弯子了。

不过在开始分析原理之前, 先**明确我们的问题**:

1. Handler 是如何与线程关联的?
2. Handler 发出去的消息是谁管理的?
3. 消息又是怎么回到 handleMessage() 方法的?
4. 线程的切换是怎么回事?

2. 1 Handler 与 Looper 的关联

实际上我们在实例化 Handler 的时候 Handler 会去检查当前线程的 Looper 是否存在，如果不存在则会报异常，也就是说**在创建 Handler 之前一定需要先创建 Looper**。

代码如下：

```
public Handler(Callback callback, boolean async) {  
  
    // 检查当前的线程是否有 Looper  
  
    mLooper = Looper.myLooper();  
  
    if (mLooper == null) {  
  
        throw new RuntimeException(  
            "Can't create handler inside thread that has not called Looper.prepare()");  
  
    }  
  
    // Looper 持有一个 MessageQueue  
  
    mQueue = mLooper.mQueue;}
```

这个异常相信很多同学遇到过，而我们平时直接使用感受不到这个异常是因为主线程已经为我们创建好了 Looper，先记住，后面会讲。（见【3.2】）

一个完整的 Handler 使用例子其实是这样的：

```
class LooperThread extends Thread {  
  
    public Handler mHandler;  
  
    public void run() {  
  
        Looper.prepare();  
  
        mHandler = new Handler() {  
  
            public void handleMessage(Message msg) {  
  
                // process incoming messages here  
  
            }  
        };  
    }  
}
```

```
};

    Looper.loop();

}
}
```

Looper.prepare() :

```
//Looperprivate static void prepare(boolean quitAllowed) {

    if (sThreadLocal.get() != null) {

        throw new RuntimeException("Only one Looper may be created per thread");
    }

    sThreadLocal.set(new Looper(quitAllowed));
}
```

Looper 提供了 Looper.prepare() 方法来创建 Looper , 并且会借助 ThreadLocal 来实现与当前线程的绑定功能。Looper.loop() 则会开始不断尝试从 MessageQueue 中获取 Message , 并分发给对应的 Handler (见【2.3】) 。也就是说 Handler 跟线程的关联是靠 Looper 来实现的。

2.2 Message 的存储与管理

Handler 提供了一些列的方法让我们来发送消息, 如 send() 系列 post() 系列 。不过不管我们调用什么方法, 最终都会走到 Message.enqueueMessage(Message, long) 方法。
以 sendEmptyMessage(int) 方法为例:

```
//HandlersendEmptyMessage(int)

    -> sendEmptyMessageDelayed(int,int)

    -> sendMessageAtTime(Message,long)

    -> enqueueMessage(MessageQueue,Message,long)

    -> queue.enqueueMessage(Message, long);
```

到了这里, 消息的管理者 MessageQueue 也就露出了水面。

MessageQueue 顾名思议, 就是个队列, 负责消息的入队出队。

2.3 Message 的分发与处理

了解清楚 `Message` 的发送与存储管理后，就该揭开分发与处理的面纱了。

前面说到了 `Looper.loop()` 负责对消息的分发，本章节进行分析。

先来看看所涉及到的方法：

```
//Looperpublic static void loop() {  
  
    final Looper me = myLooper();  
  
    if (me == null) {  
  
        throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");  
  
    }  
  
    final MessageQueue queue = me.mQueue;  
  
    //...  
  
    for (;;) {  
  
        // 不断从 MessageQueue 获取 消息  
  
        Message msg = queue.next(); // might block  
  
        //退出 Looper  
  
        if (msg == null) {  
  
            // No message indicates that the message queue is quitting.  
  
            return;  
  
        }  
  
        //...  
  
        try {  
  
            msg.target.dispatchMessage(msg);  
  
            end = (slowDispatchThresholdMs == 0) ? 0 : SystemClock.uptimeMillis();  
        }  
    }  
}
```

```
    } finally {
        //...
    }
    //...
    //回收 message, 见【3.5】
    msg.recycleUnchecked();
}
}
```

`loop()` 里调用了 `MessageQueue.next()`:

```
//MessageQueueMessage next() {
    //...
    for (;;) {
        //...
        nativePollOnce(ptr, nextPollTimeoutMillis);
    }
    synchronized (this) {
        // Try to retrieve the next message.  Return if found.
        final long now = SystemClock.uptimeMillis();
        Message prevMsg = null;
        Message msg = mMessages;
        //...
        if (msg != null) {
            if (now < msg.when) {
```

```
        // Next message is not ready. Set a timeout to wake up
        // when it is ready.

        nextPollTimeoutMillis = (int) Math.min(msg.when - now,
Integer.MAX_VALUE);

    } else {

        // Got a message.

        mBlocked = false;

        if (prevMsg != null) {

            prevMsg.next = msg.next;

        } else {

            mMessages = msg.next;

        }

        msg.next = null;

        return msg;

    }

} else {

    // No more messages.

    nextPollTimeoutMillis = -1;

}

// Process the quit message now that all pending messages have
// been handled.

if (mQuitting) {

    dispose();
}
```

```
        return null;

    }

}

// Run the idle handlers. 关于 IdleHandler 自行了解

//...

}}
```

还调用了 `msg.target.dispatchMessage(msg)`，`msg.target` 就是发送该消息的 Handler，这样就回调到了 Handler 那边去了：

```
//Handlerpublic void dispatchMessage(Message msg) {

    //msg.callback 是 Runnable，如果是 post 方法则会走这个 if

    if (msg.callback != null) {

        handleCallback(msg);

    } else {

        //callback 见【3.4】

        if (mCallback != null) {

            if (mCallback.handleMessage(msg)) {

                return;

            }

        }

        //回调到 Handler 的 handleMessage 方法

        handleMessage(msg);

    }

}}
```

注意：`dispatchMessage()` 方法针对 `Runnable` 的方法做了特殊处理，如果是，则会直接执行 `Runnable.run()`。

分析：`Looper.loop()` 是个死循环，会**不断调用 `MessageQueue.next()` 获取 `Message`，并调用 `msg.target.dispatchMessage(msg)` 回到了 `Handler` 来分发消息，以此来完成消息的回调。**

注意：`loop()`方法并不会卡死主线程，见【6】。

那么**线程的切换又是怎么回事呢？**

很多人搞不懂这个原理，但是其实非常简单，我们将所涉及的方法调用栈画出来，如下：

```
Thread.foo(){  
  
    Looper.loop()  
  
    -> MessageQueue.next()  
  
    -> Message.target.dispatchMessage()  
  
    -> Handler.handleMessage()  
}
```

显而易见，`Handler.handleMessage()` 所在的线程最终由调用 `Looper.loop()` 的线程所决定。

平时我们用的时候从异步线程发送消息到 `Handler`，这个 `Handler` 的 `handleMessage()` 方法是在主线程调用的，所以消息就从异步线程切换到了主线程。

2.3 图解原理

文字版的原理解析到这里就结束了，如果你看到这里还是没有懂，没关系，我特意给你们准备了些图，配合着前面几个章节，再多看几遍，一定可以吃透。

[图片上传失败...(image-d9e983-1551153280608)]

[图片上传失败...(image-2d8cb1-1551153280608)]
图片来源见【6】

2.4 小结

`Handler` 的背后有着 `Looper` 以及 `MessageQueue` 的协助，三者通力合作，分工明确。

尝试小结一下它们的职责，如下：

- `Looper`：负责**关联线程以及消息的分发**在该线程下**从 `MessageQueue` 获取 `Message`，分发给 `Handler`；
- `MessageQueue`：是个队列，负责**消息的存储与管理**，负责管理由 `Handler` 发送过来的 `Message`；
- `Handler`：负责**发送并处理消息**，面向开发者，提供 API，并隐藏背后实现的细节。

对【2】章节提出的问题用一句话总结：

Handler 发送的消息由 **MessageQueue** 存储管理，并由 **Looper** 负责回调消息到 **handleMessage()**。

线程的转换由 **Looper** 完成，**handleMessage()** 所在线程由 **Looper.loop()** 调用者所在线程决定。

3. Handler 的延伸

Handler 虽然简单易用，但是要用好它还是需要注意一点，另外 **Handler** 相关 还有些鲜为人知的知识技巧，比如 **IdleHandler**。

由于 **Handler** 的特性，它在 **Android** 里的应用非常广泛，比如：**AsyncTask**、**HandlerThread**、**Messenger**、**IdleHandler** 和 **IntentService** 等等。

这些我会讲解一些，我没讲到的可以自行搜索相关内容进行了解。

3.1 Handler 引起的内存泄露原因以及最佳解决方案

Handler 允许我们发送**延时消息**，如果在延时期间用户关闭了 **Activity**，那么该 **Activity** 会泄露。

这个泄露是因为 **Message** 会持有 **Handler**，而又因为 **Java** 的特性，**内部类会持有外部类**，使得 **Activity** 会被 **Handler** 持有，这样最终就导致 **Activity** 泄露。解决该问题的最有效的方法是：将 **Handler** 定义成静态的内部类，在内部持有 **Activity** 的弱引用，并及时移除所有消息。

示例代码如下：

```
private static class SafeHandler extends Handler {  
    private WeakReference<HandlerActivity> ref;  
  
    public SafeHandler(HandlerActivity activity) {  
        this.ref = new WeakReference(activity);  
    }  
  
    @Override  
    public void handleMessage(final Message msg) {  
        HandlerActivity activity = ref.get();  
        if (activity != null) {
```

```
        activity.handleMessage(msg);

    }

}}
```

并且再在 `Activity.onDestroy()` 前移除消息，加一层保障：

```
@Overrideprotected void onDestroy() {

    safeHandler.removeCallbacksAndMessages(null);

    super.onDestroy();}
```

这样双重保障，就能完全避免内存泄露了。

注意：单纯的在 `onDestroy` 移除消息并不保险，因为 `onDestroy` 并不一定执行。

3.2 为什么我们能在主线程直接使用 Handler，而不需要创建 Looper ?

前面我们提到了每个 Handler 的线程都有一个 Looper，主线程当然也不例外，但是我们不曾准备过主线程的 Looper 而可以直接使用，这是为何？

注意：通常我们认为 `ActivityThread` 就是主线程。事实上它并不是一个线程，而是主线程操作的管理者，所以吧，我觉得把 `ActivityThread` 认为就是主线程无可厚非，另外主线程也可以说成 UI 线程。

在 `ActivityThread.main()` 方法中有如下代码：

```
//android.app.ActivityThreadpublic static void main(String[] args) {

    //...

    Looper.prepareMainLooper();

    ActivityThread thread = new ActivityThread();

    thread.attach(false);

    if (sMainThreadHandler == null) {

        sMainThreadHandler = thread.getHandler();

    }

}
```

```
//...  
  
Looper.loop();  
  
throw new RuntimeException("Main thread loop unexpectedly exited");}
```

Looper.prepareMainLooper(); 代码如下：

```
/**  
  
 * Initialize the current thread as a looper, marking it as an  
  
 * application's main looper. The main looper for your application  
  
 * is created by the Android environment, so you should never need  
  
 * to call this function yourself. See also: {@link #prepare()}  
  
 */public static void prepareMainLooper() {  
  
    prepare(false);  
  
    synchronized (Looper.class) {  
  
        if (sMainLooper != null) {  
  
            throw new IllegalStateException("The main Looper has already  
been prepared.");  
  
        }  
  
        sMainLooper = myLooper();  
  
    }  
}
```

可以看到在 `ActivityThread` 里 调用了 `Looper.prepareMainLooper()` 方法创建了主线程的 `Looper`, 并且调用了 `loop()` 方法, 所以我们就可以直接使用 `Handler` 了。

注意: `Looper.loop()` 是个死循环, 后面的代码正常情况不会执行。

3.3 主线程的 `Looper` 不允许退出

如果你尝试退出 `Looper` , 你会得到以下错误信息:

```
Caused by: java.lang.IllegalStateException: Main thread not allowed to quit.

    at android.os.MessageQueue.quit(MessageQueue.java:415)

    at android.os.Looper.quit(Looper.java:240)
```

why? 其实原因很简单, **主线程不允许退出**, 退出就意味 APP 要挂。

3.4 Handler 里藏着的 Callback 能干什么?

在 Handler 的构造方法中有几个 要求传入 Callback , 那它是什么, 又能做什么呢?

来看看 Handler.dispatchMessage(msg) 方法:

```
public void dispatchMessage(Message msg) {

    //这里的 callback 是 Runnable

    if (msg.callback != null) {

        handleCallback(msg);

    } else {

        //如果 callback 处理了该 msg 并且返回 true, 就不会再回调 handleMessage

        if (mCallback != null) {

            if (mCallback.handleMessage(msg)) {

                return;

            }

        }

        handleMessage(msg);

    }

}
```

可以看到 Handler.Callback 有**优先处理消息的权利** , 当一条消息被 Callback 处理**并拦截 (返回 true)** , 那么 Handler 的 handleMessage(msg) 方法就不会被调用了; 如果 Callback 处理了消息, 但是并没有拦截, 那么就意味着**一个消息可以同时被 Callback 以及 Handler 处理**。

这个就很有意思了, 这有什么作用呢?

我们可以利用 **Callback** 这个拦截机制来拦截 **Handler** 的消息！

场景：Hook `ActivityThread.mH`， 在 `ActivityThread` 中有个成员变量 `mH`， 它是个 `Handler`， 又是个极其重要的类， 几乎所有的插件化框架都使用了这个方法。
[](#)

3.5 创建 Message 实例的最佳方式

由于 `Handler` 极为常用， 所以为了节省开销， Android 给 `Message` 设计了回收机制， 所以我们在使用的时候尽量复用 `Message`， 减少内存消耗。
方法有二：

1. 通过 `Message` 的静态方法 `Message.obtain()`； 获取；
2. 通过 `Handler` 的公有方法 `handler.obtainMessage()`。

3.6 子线程里弹 Toast 的正确姿势

当我们尝试在子线程里直接去弹 `Toast` 的时候， 会 `crash`：

```
java.lang.RuntimeException: Can't create handler inside thread that has  
not called Looper.prepare()
```

本质上是因为 `Toast` 的实现依赖于 `Handler`， 按子线程使用 `Handler` 的要求修改即可（见【2.1】）， 同理的还有 `Dialog`。

正确示例代码如下：

```
new Thread(new Runnable() {  
  
    @Override  
  
    public void run() {  
  
        Looper.prepare();  
  
        Toast.makeText(HandlerActivity.this, "不会崩溃啦！", Toast.LENGTH_SHORT).show();  
  
        Looper.loop();  
  
    }  
});
```

3.7 妙用 Looper 机制

我们可以利用 `Looper` 的机制来帮助我们做一些事情：

1. 将 Runnable post 到主线程执行;
2. 利用 Looper 判断当前线程是否是主线程。

完整示例代码如下：

```
public final class MainThread {  
  
    private MainThread() {  
    }  
  
    private static final Handler HANDLER = new Handler(Looper.getMainLooper());  
  
    public static void run(@NonNull Runnable runnable) {  
        if (isMainThread()) {  
            runnable.run();  
        } else {  
            HANDLER.post(runnable);  
        }  
    }  
  
    public static boolean isMainThread() {  
        return Looper.myLooper() == Looper.getMainLooper();  
    }  
}
```

能够省去不少样板代码。

4. 知识点汇总

由前文可得出一些知识点，汇总一下，方便记忆。

1. Handler 的背后有 Looper、MessageQueue 支撑，Looper 负责消息分发，MessageQueue 负责消息管理；
2. 在创建 Handler 之前一定需要先创建 Looper；
3. Looper 有退出的功能，但是主线程的 Looper 不允许退出；
4. 异步线程的 Looper 需要自己调用 Looper.myLooper().quit(); 退出；
5. Runnable 被封装进了 Message，可以说是一个特殊的 Message；
6. Handler.handleMessage() 所在线程是 Looper.loop() 方法被调用的线程，也可以说成 Looper 所在的线程，并不是创建 Handler 的线程；
7. 使用内部类的方式使用 Handler 可能会导致内存泄露，即便在 Activity.onDestroy 里移除延时消息，必须要写成静态内部类；

5. 总结

Handler 简单易用的背后藏着工程师大量的智慧，要努力向他们学习。

7、Android Handler 消息机制（解惑篇）

Android 中线程的分类

带有消息队列 ,用来执行循环性任务(例如主线程、android.os.HandlerThread)

•

有消息时就处理

•

•

没有消息时就睡眠

•

没有消息队列，用来执行一次性任务（例如 `java.lang.Thread`）

- 任务一旦执行完成便退出

带有消息队列线程概述

四要素

-

`Message`(消息)

-

-

`MessageQueue`(消息队列)

-

-

`Looper`(消息循环)

-

-

`Handler`(消息发送和处理)

-

四要素的交互过程

具体工作过程

-

消息队列的创建

-

-

消息循环

-

-

消息的发送

-

最基本的两个 API

-

-

Handler.sendMessage

-

-

带一个 Message 参数，用来描述消息的内容

-

Handler.post

-
- 带一个 Runnable 参数，会被转换为一个 Message 参数
-

消息的处理

-

基于消息的异步任务接口

-

android.os.HandlerThread

-

- 适合用来处于不需要更新 UI 的后台任务

-

android.os.AsyncTask

-

- 适合用来处于需要更新 UI 的后台任务

带有消息队列线程的具体实现

ThreadLocal

ThreadLocal 并不是一个 Thread，而是 Thread 的**局部变量**。当使用 ThreadLocal 维护变量时，ThreadLocal 为每个使用该变量的线程提供独立的变

量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

从线程的角度看，目标变量就象是**线程的本地变量**，这也是类名中“Local”所要表达的意思。

Looper

用于在指定线程中运行一个消息循环，一旦有新任务则执行，执行完继续等待下一个任务，即变成**Looper线程**。Looper类的注释里有这样一个例子：

```
class LooperThread extends Thread {  
  
    public Handler mHandler;  
  
    public void run() {  
  
        //将当前线程初始化为 Looper 线程  
        Looper.prepare();  
  
        // ...其他处理，如实例化 handler  
        mHandler = new Handler() {  
  
            public void handleMessage(Message msg) {  
                // process incoming messages here  
            }  
        };  
    }  
}
```

```
};

// 开始循环处理消息队列

Looper.loop();

}

}
```

其实核心代码就两行，我们先来看下 Looper.prepare()方法的具体实现

```
public final class Looper {

    private static final String TAG = "Looper";

    // sThreadLocal.get() will return null unless you've called prepare().
    static final ThreadLocal<Looper> sThreadLocal = new
    ThreadLocal<Looper>();

    private static Looper sMainLooper; // guarded by Looper.class

    //Looper 内的消息队列
    final MessageQueue mQueue;

    // 当前线程
    final Thread mThread;

    private Printer mLogging;
```

```
private Looper(boolean quitAllowed) {  
    mQueue = new MessageQueue(quitAllowed);  
    mThread = Thread.currentThread();  
}  
  
/** Initialize the current thread as a looper.  
 * This gives you a chance to create handlers that then reference  
 * this looper, before actually starting the loop. Be sure to call  
 * {@link #loop()} after calling this method, and end it by calling  
 * {@link #quit()}.  
 */  
public static void prepare() {  
    prepare(true);  
}  
  
private static void prepare(boolean quitAllowed) {  
    //试图在有 Looper 的线程中再次创建 Looper 将抛出异常  
    if (sThreadLocal.get() != null) {  
        throw new RuntimeException("Only one Looper may be  
created per thread");  
    }  
}
```

```
sThreadLocal.set(new Looper(quitAllowed));  
}  
  
/**  
 * Initialize the current thread as a looper, marking it as an  
 * application's main looper. The main looper for your application  
 * is created by the Android environment, so you should never need  
 * to call this function yourself. See also: {@link #prepare()}  
 */  
  
public static void prepareMainLooper() {  
    prepare(false);  
    synchronized (Looper.class) {  
        if (sMainLooper != null) {  
            throw new IllegalStateException("The main Looper has  
already been prepared.");  
        }  
        sMainLooper = myLooper();  
    }  
}  
  
//~省略部分无关代码~  
}
```

从中我们可以看到以下几点：

prepare()其核心就是将 looper 对象定义为 ThreadLocal

一个 Thread 只能有一个 Looper 对象

prepare()方法会调用 Looper 的构造方法，初始化一个消息队列，并且指定当前线程

在调用 Looper.loop()方法之前，确保已经调用了 prepare(boolean quitAllowed)方法，并且我们可以调用 quite 方法结束循环

说到初始化 MessageQueue，我们来看下它是干什么的

```
/**  
 * Low-level class holding the list of messages to be dispatched by a  
 * {@link Looper}. Messages are not added directly to a MessageQueue,  
 * but rather through {@link Handler} objects associated with the Looper.  
 *  
 * You can retrieve the MessageQueue for the current thread with  
 * {@link Looper#myQueue()} Looper.myQueue().  
 */
```

它是一个低等级的持有 Messages 集合的类，被 Looper 分发。Messages 并不是直接加到 MessageQueue 的，而是通过 Handler 对象和 Looper 关联到一起。我们可以通过 Looper.myQueue()方法来检索当前线程的 MessageQueue。

接下来再看看 Looper.loop()

```
/**  
 * Run the message queue in this thread. Be sure to call  
 * {@link #quit()} to end the loop.
```

```
*/  
  
public static void loop() {  
  
    //得到当前线程 Looper  
  
    final Looper me = myLooper();  
  
    if (me == null) {  
  
        throw new RuntimeException("No Looper; Looper.prepare()  
wasn't called on this thread.");  
  
    }  
  
    //得到当前 looper 的 MessageQueue  
  
    final MessageQueue queue = me.mQueue;  
  
  
    // Make sure the identity of this thread is that of the local process,  
    // and keep track of what that identity token actually is.  
  
    Binder.clearCallingIdentity();  
  
    final long ident = Binder.clearCallingIdentity();  
  
  
    //开始循环  
  
    for (;;) {  
  
        Message msg = queue.next(); // might block  
  
        if (msg == null) {  
  
            // No message indicates that the message queue is quitting.  
  
            //没有消息表示消息队列正在退出  
        }  
    }  
}
```

```
        return;

    }

// This must be in a local variable, in case a UI event sets the
logger

    Printer logging = me.mLogging;

    if (logging != null) {

        logging.println(">>>> Dispatching to " + msg.target + " "
+
msg.callback + ": " + msg.what);

    }

//将真正的处理工作交给 message 的 target , 即 handler
msg.target.dispatchMessage(msg);

if (logging != null) {

    logging.println("<<<< Finished to " + msg.target + " " +
msg.callback);

}

// Make sure that during the course of dispatching the
// identity of the thread wasn't corrupted.
```

```
final long newIdent = Binder.clearCallingIdentity();

if (ident != newIdent) {

    Log.wtf(TAG, "Thread identity changed from 0x"
        + Long.toHexString(ident) + " to 0x"
        + Long.toHexString(newIdent) + " while
dispatching to "
        + msg.target.getClass().getName() + " "
        + msg.callback + " what=" + msg.what);

}

//回收消息资源
msg.recycleUnchecked();

}

}
```

通过这段代码可知，调用 **loop** 方法后，**Looper** 线程就开始真正工作了，它不断从自己的 **MessageQueue** 中取出队头的消息(或者说是任务)执行。

除了 **prepare()**和 **loop()**方法，**Looper** 类还有一些比较有用的方法，比如

Looper.myLooper()得到当前线程 looper 对象

getThread()得到 looper 对象所属线程

quit()方法结束 looper 循环

这里需要注意的一点是，quit() 方法其实调用的是 MessageQueue 的 quite
(boolean safe) 方法。

```
void quit(boolean safe) {  
    if (!mQuitAllowed) {  
        throw new IllegalStateException("Main thread not allowed to  
quit.");  
    }  
  
    synchronized (this) {  
        if (mQuitting) {  
            return;  
        }  
        mQuitting = true;  
  
        if (safe) {  
            Looper.myLooper().quit();  
        } else {  
            Looper.myLooper().quitSafely();  
        }  
    }  
}
```

```
removeAllFutureMessagesLocked();

} else {

    removeAllMessagesLocked();

}

// We can assume mPtr != 0 because mQuitting was previously
false.

nativeWake(mPtr);

}

}

•
```

我们看到其实主线程是不能调用这个方法退出消息队列的。至于
`mQuitAllowed` 参数是在 `Looper` 初始化的时候初始化的，主线程初始化调用
的是 `Looper.prepareMainLooper()` 方法，这个方法把参数设置为 `false`。

Message

在整个消息处理机制中，`message` 又叫 `task`，封装了任务携带的信息和处理该
任务的 `handler`。我们看下这个类的注释

```
/**  
 *  
 * Defines a message containing a description and arbitrary data object that can be  
 * sent to a {@link Handler}. This object contains two extra int fields and an  
 * extra object field that allow you to not do allocations in many cases.  
 *  
 * While the constructor of Message is public, the best way to get  
 * one of these is to call {@link #obtain Message.obtain()} or one of the  
 * {@link Handler#obtainMessage Handler.obtainMessage()} methods, which will pull  
 * them from a pool of recycled objects.  
 */
```

这个类定义了一个包含描述和一个任意类型对象的对象，它可以被发送给
Handler。

从注释里我们还可以了解到以下几点：

尽管 Message 有 public 的默认构造方法，但是你应该通过 Message.obtain()
来从消息池中获得空消息对象，以节省资源。

如果你的 message 只需要携带简单的 int 信息，请优先使用 Message.arg1 和
Message.arg2 来传递信息，这比用 Bundle 更省内存

用 message.what 来标识信息，以便用不同方式处理 message。

Handler

从 MessageQueue 的注释中，我们知道添加消息到消息队列是通过 Handler 来操作的。我们通过源码来看下具体是怎么实现的

```
/*
 * A Handler allows you to send and process {@link Message} and Runnable
 * objects associated with a thread's {@link MessageQueue}. Each Handler
 * instance is associated with a single thread and that thread's message
 * queue. When you create a new Handler, it is bound to the thread /
 * message queue of the thread that is creating it – from that point on,
 * it will deliver messages and runnables to that message queue and execute
 * them as they come out of the message queue.
 *
 *
 * There are two main uses for a Handler: (1) to schedule messages and
 * runnables to be executed as some point in the future; and (2) to enqueue
 * an action to be performed on a different thread than your own.
 *
 */

```

注释比较简单，这里就不过多翻译了，主要内容是：**每一个 Handler 实例关联了一个单一的 thread 和这个 thread 的 messagequeue ,当 Handler 的实例被创建的时候它就被绑定到了创建它的 thread。它用来调度 message 和 runnables 在未来某个时间点的执行，还可以排列其他线程里执行的操作。**

```
public class Handler {
```

```
//~省略部分无关代码~
```

```
final MessageQueue mQueue;
```

```
final Looper mLooper;
```

```
public Handler() {
```

```
    this(null, false);
```

```
}
```

```
public Handler(Looper looper) {
```

```
    this(looper, null, false);
```

```
}
```

```
public Handler(boolean async) {
```

```
    this(null, async);
```

```
}
```

```
public Handler(Callback callback, boolean async) {
```

```
    if (FIND_POTENTIAL_LEAKS) {
```

```
        final Class<? extends Handler> klass = getClass();
```

```
        if ((klass.isAnonymousClass() || klass.isMemberClass() ||
```

```
klass.isLocalClass()) &&
```

```
        (klass.getModifiers() & Modifier.STATIC) == 0) {
```

```
            Log.w(TAG, "The following Handler class should be
```

```
static or leaks might occur: " +
```

```
        klass.getCanonicalName());
```

```
}
```

```
}
```

```
mLooper = Looper.myLooper();

if (mLooper == null) {

    throw new RuntimeException(
        "Can't create handler inside thread that has not
called Looper.prepare()");

}

mQueue = mLooper.mQueue;

mCallback = callback;

mAsynchronous = async;

}

public Handler(Looper looper, Callback callback, boolean async) {

    mLooper = looper;

    mQueue = looper.mQueue;

    mCallback = callback;

    mAsynchronous = async;

}

//~省略部分无关代码~

}
```

先看构造方法，其实里边的重点是初始化了两个变量，把关联 looper 的 MessageQueue 作为自己的 MessageQueue，因此它的消息将发送到关联 looper 的 MessageQueue 上。

有了 handler 之后，我们就可以使用 Handler 提供的 post 和 send 系列方法向 MessageQueue 上发送消息了。其实 post 发出的 Runnable 对象最后都被封装成 message 对象

接下来我们看一下 handler 是如何发送消息的

```
/**  
 * Causes the Runnable r to be added to the message queue.  
 * The runnable will be run on the thread to which this handler is  
 * attached.  
 *  
 * @param r The Runnable that will be executed.  
 *  
 * @return Returns true if the Runnable was successfully placed in to  
 * the  
 * message queue. Returns false on failure, usually because  
 * the  
 * looper processing the message queue is exiting.  
 */  
  
public final boolean post(Runnable r)
```

```
{  
    return sendMessageDelayed(getPostMessage(r), 0);  
  
}  
  
/**  
 * Enqueue a message into the message queue after all pending  
messages  
 * before (current time + delayMillis). You will receive it in  
 * {@link #handleMessage}, in the thread attached to this handler.  
 *  
 * @return Returns true if the message was successfully placed in to  
the  
 * message queue. Returns false on failure, usually because  
the  
 * looper processing the message queue is exiting. Note  
that a  
 * result of true does not mean the message will be  
processed -- if  
 * the looper is quit before the delivery time of the message  
 * occurs then the message will be dropped.  
*/  
public final boolean sendMessageDelayed(Message msg, long
```

```
delayMillis)

{

    if (delayMillis < 0) {

        delayMillis = 0;

    }

    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);

}

/**



 * Enqueue a message into the message queue after all pending messages

 * before the absolute time (in milliseconds)

<var>uptimeMillis</var>.

 * <b>The time-base is {@link android.os.SystemClock#uptimeMillis}>.</b>

 * Time spent in deep sleep will add an additional delay to execution.

 * You will receive it in {@link #handleMessage}, in the thread attached

 * to this handler.

 *

 * @param uptimeMillis The absolute time at which the message
```

should be

- * delivered, using the
- * {@link android.os.SystemClock#uptimeMillis} time-base.
- *

* @return Returns true if the message was successfully placed in to
the

* message queue. Returns false on failure, usually because
the

* looper processing the message queue is exiting. Note
that a

* result of true does not mean the message will be
processed -- if

- * the looper is quit before the delivery time of the message
- * occurs then the message will be dropped.

*/

public boolean sendMessageAtTime(Message msg, long

uptimeMillis) {

 MessageQueue queue = mQueue;

 if (queue == null) {

 RuntimeException e = new RuntimeException(

 this + " sendMessageAtTime() called with no
mQueue");

```
        Log.w("Looper", e.getMessage(), e);

        return false;

    }

    return enqueueMessage(queue, msg, uptimeMillis);

}

private boolean enqueueMessage(MessageQueue queue, Message
msg, long uptimeMillis) {

    msg.target = this;

    if (mAsynchronous) {

        msg.setAsynchronous(true);

    }

    return queue.enqueueMessage(msg, uptimeMillis);

}
```

这里我们只列出了一种调用关系，其他调用关系大同小异，我们来分析一下

1. 调用 `getPostMessage(r)`，把 `Runnable` 对象添加到一个 `Message` 对象中。
2. `sendMessageDelayed(getPostMessage(r), 0)`，基本没做什么操作，又继续调用 `sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis)` 方法，在这个方法里拿到创建这个 `Handler` 对象的线程持有的 `MessageQueue`。

3. 调用 enqueueMessage(queue, msg, uptimeMillis)方法 , 给 msg 对象的 target 变量赋值为当前的 Handler 对象 然后放入到 MessageQueue。那发送消息说完了 , 那我们的消息是怎样被处理的呢 ?

我们看到 message.target 为该 handler 对象 , 这确保了 looper 执行到该 message 时能找到处理它的 handler , 即 loop() 方法中的关键代码。

```
/**  
 * Callback interface you can use when instantiating a Handler to  
 * avoid  
 * having to implement your own subclass of Handler.  
 *  
 * @param msg A {@link android.os.Message} object  
 * @return True if no further handling is desired  
 */  
  
public interface Callback {  
    public boolean handleMessage(Message msg);  
}  
  
/**  
 * Subclasses must implement this to receive messages.  
 */  
  
public void handleMessage(Message msg) {
```

```
}

/**
 * Handle system messages here.
 */

public void dispatchMessage(Message msg) {
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        handleMessage(msg);
    }
}

private static void handleCallback(Message message) {
    message.callback.run();
}

我们看到这里最终又调用到了我们重写的 handleMessage(Message msg)方
```

法来做处理子线程发来的消息或者调用 `handleCallback(Message message)` 去执行我们子线程中定义并传过来的操作。

8、Android 消息机制

Android UI 是线程不安全的，如果在子线程中尝试进行 UI 操作，程序就有可能会崩溃。相信大家在日常的工作当中都会经常遇到这个问题，解决的方案应该也是早已烂熟于心，即创建一个 Message 对象，然后借助 Handler 发送出去，之后在 Handler 的 `handleMessage()` 方法中获得刚才发送的 Message 对象，然后在这里进行 UI 操作就不会再出现崩溃了。

这种处理方式被称为异步消息处理线程，虽然我相信大家都会用，可是你知道它背后的原理是什么样的吗？今天我们就来一起深入探究一下 Handler 和 Message 背后的秘密。

首先来看一下如何创建 Handler 对象。你可能会觉得挺纳闷的，创建 Handler 有什么好看的呢，直接 new 一下不就行了？确实，不过即使只是简单 new 一下，还是有不少地方需要注意的，我们尝试在程序中创建两个 Handler 对象，一个在主线程中创建，一个在子线程中创建，代码如下所示：

```
public class MainActivity extends Activity {
```

```
private Handler handler1;

private Handler handler2;

@Override

protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_main);

    handler1 = new Handler();

    new Thread(new Runnable() {

        @Override

        public void run() {

            handler2 = new Handler();

        }

    }).start();

}

}
```

如果现在运行一下程序，你会发现，在子线程中创建的 Handler 是会导致程序崩溃的，提示的错误信息为 `Can't create handler inside thread that has not called Looper.prepare()`。说是不能在没有调用 `Looper.prepare()` 的线程中创建 Handler，那我们尝试在子线程中先调用一下 `Looper.prepare()` 呢，代码如下所示：

```
new Thread(new Runnable() {

    @Override

    public void run() {

        Looper.prepare();

    }

}).start();
```

```

        handler2 = new Handler();
    }
}).start();
果然这样就不会崩溃了，不过只满足于此显然是不够的，我们来看下 Handler 的源码，搞清楚为什么不调用 Looper.prepare()就不行呢。Handler 的无参构造函数如下所示：
public Handler() {
    if (FIND_POTENTIAL_LEAKS) {
        final Class<? extends Handler> klass = getClass();
        if ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass())
&&
            (klass.getModifiers() & Modifier.STATIC) == 0) {
            Log.w(TAG, "The following Handler class should be static or leaks might
occur: " +
                    klass.getCanonicalName());
        }
    }
    mLooper = Looper.myLooper();
    if (mLooper == null) {
        throw new RuntimeException(
                "Can't create handler inside thread that has not called Looper.prepare()");
    }
    mQueue = mLooper.mQueue;
    mCallback = null;
}

```

可以看到，在第 10 行调用了 **Looper.myLooper()**方法获取了一个 **Looper** 对象，如果 **Looper** 对象为空，则会抛出一个运行时异常，提示的错误正是 **Can't create handler inside thread that has not called Looper.prepare()**！那什么时候 **Looper** 对象才可能为空呢？这就要看看 **Looper.myLooper()**中的代码了，如下所示：

```

public static final Looper myLooper() {
    return (Looper)sThreadLocal.get();
}

```

这个方法非常简单，就是从 **sThreadLocal** 对象中取出 **Looper**。如果 **sThreadLocal** 中有 **Looper** 存在就返回 **Looper**，如果没有 **Looper** 存在自然就返回空了。因此你可以想象得到是在哪里给 **sThreadLocal** 设置 **Looper** 了吧，当然是 **Looper.prepare()**方法！我们来看下它的源码：

```

public static final void prepare() {
    if (sThreadLocal.get() != null) {
        throw new RuntimeException("Only one Looper may be created per thread");
    }
    sThreadLocal.set(new Looper());
}

```

可以看到，首先判断 **sThreadLocal** 中是否已经存在 **Looper** 了，如果还没有则创建一个新的 **Looper** 设置进去。这样也就完全解释了为什么我们要先调用

Looper.prepare()方法，才能创建 Handler 对象。同时也可以看出每个线程中最多只会有一个 Looper 对象。

咦？不对呀！主线程中的 Handler 也没有调用 Looper.prepare()方法，为什么就没有崩溃呢？细心的朋友我相信都已经发现了这一点，这是由于在程序启动的时候，系统已经帮我们自动调用了 Looper.prepare()方法。查看 ActivityThread 中的 main()方法，代码如下所示：

```
public static void main(String[] args) {
    SamplingProfilerIntegration.start();
    CloseGuard.setEnabled(false);
    Environment.initForCurrentUser();
    EventLogger.setReporter(new EventLoggingReporter());
    Process.setArgV0("<pre-initialized>");
    Looper.prepareMainLooper();
    ActivityThread thread = new ActivityThread();
    thread.attach(false);
    if (sMainThreadHandler == null) {
        sMainThreadHandler = thread.getHandler();
    }
    AsyncTask.init();
    if (false) {
        Looper.myLooper().setMessageLogging(new LogPrinter(Log.DEBUG,
"ActivityThread"));
    }
    Looper.loop();
    throw new RuntimeException("Main thread loop unexpectedly exited");
}
```

可以看到，在第 7 行调用了 Looper.prepareMainLooper()方法，而这个方法又会再去调用 Looper.prepare()方法，代码如下所示：

```
public static final void prepareMainLooper() {
    prepare();
    setMainLooper(myLooper());
    if (Process.supportsProcesses()) {
        myLooper().mQueue.mQuitAllowed = false;
    }
}
```

}

因此我们应用程序的主线程中会始终存在一个 Looper 对象 ,从而不需要再手动去调用 Looper.prepare()方法了。

这样基本就将 Handler 的创建过程完全搞明白了 ,总结一下就是在主线程中可以直接创建 Handler 对象 ,而在子线程中需要先调用 Looper.prepare()才能创建 Handler 对象。

看完了如何创建 Handler 之后 ,接下来我们看一下如何发送消息 ,这个流程相信大家也已经非常熟悉了 ,new 出一个 Message 对象 ,然后可以使用 setData() 方法或 arg 参数等方式为消息携带一些数据 ,再借助 Handler 将消息发送出去就可以了 ,示例代码如下 :

```
new Thread(new Runnable() {  
    @Override  
    public void run() {  
        Message message = new Message();  
        message.arg1 = 1;  
        Bundle bundle = new Bundle();  
        bundle.putString("data", "data");  
        message.setData(bundle);  
        handler.sendMessage(message);  
    }  
}).start();
```

可是这里 Handler 到底是把 Message 发送到哪里去了呢 ?为什么之后又可以在 Handler 的 handleMessage()方法中重新得到这条 Message 呢 ?看来又需要通过阅读源码才能解除我们心中的疑惑了 ,Handler 中提供了很多个发送消息的

方法，其中除了 sendMessageAtFrontOfQueue() 方法之外，其它的发送消息方法最终都会辗转调用到 sendMessageAtTime() 方法中，这个方法的源码如下所示：

```
public boolean sendMessageAtTime(Message msg, long uptimeMillis)
{
    boolean sent = false;
    MessageQueue queue = mQueue;
    if (queue != null) {
        msg.target = this;
        sent = queue.enqueueMessage(msg, uptimeMillis);
    }
    else {
        RuntimeException e = new RuntimeException(
            this + " sendMessageAtTime() called with no mQueue");
        Log.w("Looper", e.getMessage(), e);
    }
    return sent;
}
```

sendMessageAtTime() 方法接收两个参数，其中 msg 参数就是我们发送的 Message 对象，而 uptimeMillis 参数则表示发送消息的时间，它的值等于自系统开机到当前时间的毫秒数再加上延迟时间，如果你调用的不是 sendMessageDelayed() 方法，延迟时间就为 0，然后将这两个参数都传递到

MessageQueue 的 enqueueMessage()方法中。这个 MessageQueue 又是什么东西呢？其实从名字上就可以看出了，它是一个消息队列，用于将所有收到的消息以队列的形式进行排列，并提供入队和出队的方法。这个类是在 Looper 的构造函数中创建的，因此一个 Looper 也就对应了一个 MessageQueue。

那么 enqueueMessage()方法毫无疑问就是入队的方法了，我们来看下这个方法的源码：

```
final boolean enqueueMessage(Message msg, long when) {  
    if (msg.when != 0) {  
        throw new AndroidRuntimeException(msg + " This message is  
already in use.");  
    }  
    if (msg.target == null && !mQuitAllowed) {  
        throw new RuntimeException("Main thread not allowed to  
quit");  
    }  
    synchronized (this) {  
        if (mQuiting) {  
            RuntimeException e = new RuntimeException(msg.target +  
                " sending message to a Handler on a dead thread");  
            Log.w("MessageQueue", e.getMessage(), e);  
        }  
        mMessages.add(msg);  
        if (msg.when == 0) {  
            if (mFirst == null) {  
                mFirst = msg;  
            } else {  
                msg.setNext(mFirst);  
                mFirst = msg;  
            }  
        } else {  
            if (mFirst == null) {  
                mFirst = msg;  
            } else {  
                msg.setNext(mFirst);  
                mFirst = msg;  
            }  
            if (mLast == null) {  
                mLast = msg;  
            } else {  
                mLast.setNext(msg);  
                mLast = msg;  
            }  
        }  
        if (mLast == null) {  
            mLast = msg;  
        }  
        if (mLast == msg) {  
            mLast.setNext(null);  
        }  
    }  
    return true;  
}
```

```
        return false;

    } else if (msg.target == null) {

        mQuiting = true;

    }

    msg.when = when;

    Message p = mMessages;

    if (p == null || when == 0 || when < p.when) {

        msg.next = p;

        mMessages = msg;

        this.notify();

    } else {

        Message prev = null;

        while (p != null && p.when <= when) {

            prev = p;

            p = p.next;

        }

        msg.next = prev.next;

        prev.next = msg;

        this.notify();

    }

    return true;
```

}

首先你要知道 ,MessageQueue 并没有使用一个集合把所有的消息都保存起来 , 它只使用了一个 mMessages 对象表示当前待处理的消息。然后观察上面的代码的 16~31 行我们就可以看出 , 所谓的入队其实就是将所有的消息按时间来进行排序 , 这个时间当然就是我们刚才介绍的 uptimeMillis 参数。具体的操作方法就根据时间的顺序调用 msg.next , 从而为每一个消息指定它的下一个消息是什么。当然如果你是通过 sendMessageAtFrontOfQueue() 方法来发送消息的 , 它也会调用 enqueueMessage() 来让消息入队 , 只不过时间为 0 , 这时会把 mMessages 赋值为新入队的这条消息 , 然后将这条消息的 next 指定为刚才的 mMessages , 这样也就完成了添加消息到队列头部的操作。

现在入队操作我们就已经看明白了 , 那出队操作是在哪里进行的呢?这个就需要看一看 Looper.loop() 方法的源码了 , 如下所示:

```
public static final void loop() {
    Looper me = myLooper();
    MessageQueue queue = me.mQueue;
    while (true) {
        Message msg = queue.next(); // might block
        if (msg != null) {
            if (msg.target == null) {
                return;
            }
            if (me.mLogging!= null) me.mLogging.println(
                    ">>>> Dispatching to " + msg.target + " "
                    + msg.callback + ": " + msg.what
                    );
            msg.target.dispatchMessage(msg);
            if (me.mLogging!= null) me.mLogging.println(
                    "<<<< Finished to      " + msg.target + " "
                    + msg.callback);
            msg.recycle();
        }
    }
}
```

```
    }  
}
```

可以看到，这个方法从第 4 行开始，进入了一个死循环，然后不断地调用的 `MessageQueue` 的 `next()` 方法，我想你已经猜到了，这个 `next()` 方法就是消息队列的出队方法。不过由于这个方法的代码稍微有点长，我就不贴出来了，它的简单逻辑就是如果当前 `MessageQueue` 中存在 `mMessages`(即待处理消息)，就将这个消息出队，然后让下一条消息成为 `mMessages`，否则就进入一个阻塞状态，一直等到有新的消息入队。继续看 `loop()` 方法的第 14 行，每当有一个消息出队，就将它传递到 `msg.target` 的 `dispatchMessage()` 方法中，那这里 `msg.target` 又是什么呢？其实就是 `Handler` 啦，你观察一下上面 `sendMessageAtTime()` 方法的第 6 行就可以看出来了。接下来当然就要看一看 `Handler` 中 `dispatchMessage()` 方法的源码了，如下所示：

```
public void dispatchMessage(Message msg) {  
    if (msg.callback != null) {  
        handleCallback(msg);  
    } else {  
        if (mCallback != null) {  
            if (mCallback.handleMessage(msg)) {  
                return;  
            }  
        }  
        handleMessage(msg);  
    }  
}
```

在第 5 行进行判断，如果 `mCallback` 不为空，则调用 `mCallback` 的 `handleMessage()` 方法，否则直接调用 `Handler` 的 `handleMessage()` 方法，并将消息对象作为参数传递过去。这样我相信大家就都明白了为什么 `handleMessage()` 方法中可以获取到之前发送的消息了吧！

因此，一个最标准的异步消息处理线程的写法应该是这样：

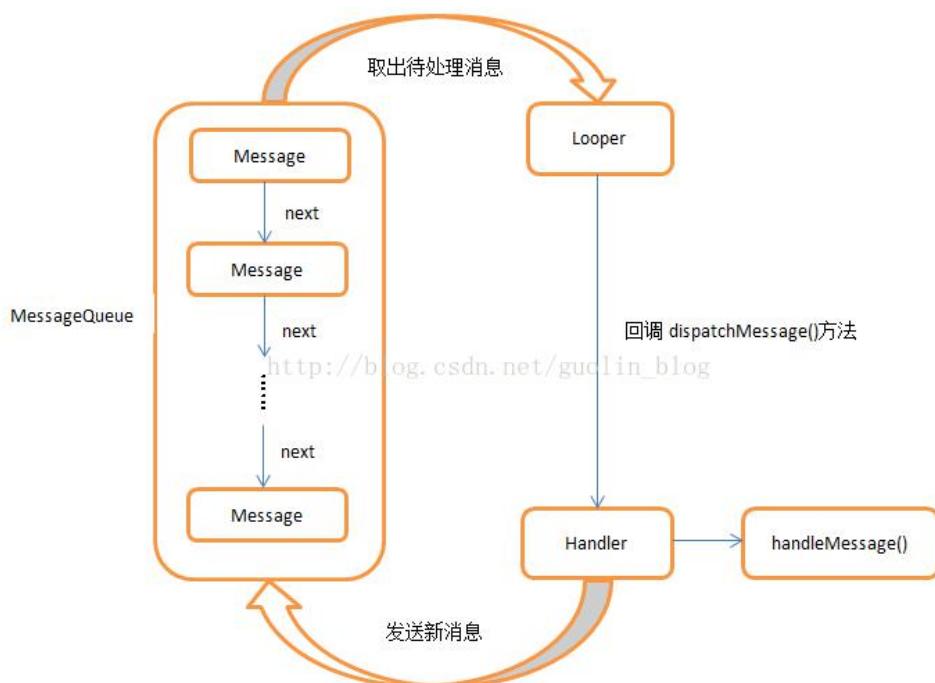
```
class LooperThread extends Thread {  
    public Handler mHandler;  
  
    public void run() {  
        Looper.prepare();  
  
        mHandler = new Handler() {  
            public void handleMessage(Message msg) {  
                // process incoming messages here  
            }  
        };  
  
        Looper.loop();  
    }  
}
```

```
    }  
}
```

当然，这段代码是从 Android 官方文档上复制的，不过大家现在再来看这段代码，是不是理解的更加深刻了？

那么我们还是要来继续分析一下，为什么使用异步消息处理的方式就可以对 UI 进行操作了呢？这是由于 Handler 总是依附于创建时所在的线程，比如我们的 Handler 是在主线程中创建的，而在子线程中又无法直接对 UI 进行操作，于是我们就通过一系列的发送消息、入队、出队等环节，最后调用到了 Handler 的 handleMessage()方法中，这时的 handleMessage()方法已经是在主线程中运行的，因而我们当然可以在这里进行 UI 操作了。整个异步消息处理流程的示意

图如下图所示：



另外除了发送消息之外，我们还有以下几种方法可以在子线程中进行 UI 操作：

1. Handler 的 post()方法

2. View 的 post()方法

3. Activity 的 runOnUiThread()方法

我们先来看下 Handler 中的 post()方法，代码如下所示：

```
public final boolean post(Runnable r)  
{  
    return sendMessageDelayed(getPostMessage(r), 0);  
}
```

原来这里还是调用了 sendMessageDelayed()方法去发送一条消息啊，并且还使用了 getPostMessage()方法将 Runnable 对象转换成了一条消息，我们来看下这个方法的源码：

```
private final Message getPostMessage(Runnable r) {  
    Message m = Message.obtain();  
    m.callback = r;  
    return m;  
}
```

原来这里还是调用了 sendMessageDelayed()方法去发送一条消息啊，并且还使用了 getPostMessage()方法将 Runnable 对象转换成了一条消息，我们来看下这个方法的源码：

```
private final Message getPostMessage(Runnable r) {  
  
    Message m = Message.obtain();  
  
    m.callback = r;  
  
    return m;  
  
}
```

也太简单了！竟然就是直接调用了一开始传入的 `Runnable` 对象的 `run()`方法。因此在子线程中通过 `Handler` 的 `post()`方法进行 UI 操作就可以这么写：

```
public class MainActivity extends Activity {  
  
    private Handler handler;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
  
        super.onCreate(savedInstanceState);  
  
        setContentView(R.layout.activity_main);  
  
        handler = new Handler();  
  
        new Thread(new Runnable() {  
  
            @Override  
            public void run() {  
  
                handler.post(new Runnable() {  
  
                    @Override  
                    public void run() {  
                }  
            }  
        }  
    }  
}
```

```
// 在这里进行 UI 操作

}

});

}

}).start();

}

}

}

虽然写法上相差很多，但是原理是完全一样的，我们在 Runnable 对象的 run() 方法里更新 UI，效果完全等同于在 handleMessage() 方法中更新 UI。
```

然后再来看一下 `View` 中的 `post()` 方法，代码如下所示：

```
public boolean post(Runnable action) {

    Handler handler;

    if (mAttachInfo != null) {

        handler = mAttachInfo.mHandler;

    } else {

        ViewRoot.getRunQueue().post(action);

        return true;

    }

    return handler.post(action);

}
```

原来就是调用了 `Handler` 中的 `post()` 方法，我相信已经没有什么必要再做解释了。

最后再来看一下 Activity 中的 runOnUiThread()方法，代码如下所示：

```
public final void runOnUiThread(Runnable action) {  
    if (Thread.currentThread() != mUiThread) {  
        mHandler.post(action);  
    } else {  
        action.run();  
    }  
}
```

果当前的线程不等于 UI 线程(主线程), 就去调用 Handler 的 post()方法, 否则就直接调用 Runnable 对象的 run()方法。还有什么会比这更清晰明了的吗?

通过以上所有源码的分析 , 我们已经发现了 , 不管是使用哪种方法在子线程中更新 UI , 其实背后的原理都是相同的 , 必须都要借助异步消息处理的机制来实现 , 而我们又已经将这个机制的流程完全搞明白了 , 真是一件一本万利的事情啊

二、Activity 相关

1、启动模式以及使用场景?

一、Activity 四种启动模式:

(一)、基本描述

1. **: 标准模式** : 如果在 manifest 中不设置就默认 standard ; standard 就是新建一个 Activity 就在栈中新建一个 activity 实例 ;

2. **栈顶复用模式** : 与 standard 相比栈顶复用可以有效减少 activity

重复创建对资源的消耗，但是这要根据具体情况而定，不能一概而论；

3. **栈内单例模式**，栈内只有一个 activity 实例，栈内已存 activity

实例，在其他 activity 中 start 这个 activity，Android 直接把这个实例上面其他 activity 实例踢出栈 GC 掉；

4. **堆内单例**：整个手机操作系统里面只有一个实例存在就是内存单例；

在 singleTop、singleTask、singleInstance 中如果在应用内存在

Activity 实例，并且再次发生 startActivity(Intent intent) 回到 Activity

后，由于并不是重新创建 Activity 而是复用栈中的实例，因此 Activity

再获取焦点后并没调用 onCreate、onStart，而是直接调用了

onNewIntent(Intent intent) 函数；

(二)、taskAffinity 属性

taskAffinity 属性和 Activity 的启动模式息息相关，而且 taskAffinity 属性比较特殊，在普通的开发中也是鲜有遇到，但是在有些特定场景下却有着出其不意的效果。

taskAffinity 是 Activity 在 manifest 中配置的一个属性，暂时可以理解为：

taskAffinity 为宿主 Activity 指定了存放的任务栈[不同于 App 中其他的 Activity 的栈]，为 activity 设置 taskAffinity 属性时不能和包名相同，因为 Android 团队为 taskAffinity 默认设置为包名任务栈。

taskAffinity 只有和 SingleTask 启动模式匹配使用时，启动的 Activity 才会运行在名字和 taskAffinity 相同的任务栈中。

(三)、Intent 中标志位设置启动模式

在上文中的四种模式都是在 mainfest 的 xml 文件中进行配置的，

GoogleAndroid 团队同时提供另种级别更高的设置方式，即通过

Intent.setFlags(int flags)设置启动模式；

1. **FLAG_ACTIVITY_CLEAR_TOP** : 等同于 mainfest 中配置的 singleTask，没啥好讲的；
2. **FLAG_ACTIVITY_SINGLE_TOP**: 同样等同于 mainfest 中配置的 singleTop;
3. **FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS**: 其对应在 AndroidManifest 中的属性为 android:excludeFromRecents= “true” ,当用户按了“最近任务列表”时候,该 Task 不会出现在最近任务列表中，可达到隐藏应用的目的。
4. **FLAG_ACTIVITY_NO_HISTORY**: 对应在 AndroidManifest 中的属性为: android:noHistory= “true” ,这个 FLAG 启动的 Activity ,一旦退出，它不会存在于栈中。
5. **-----** : 这个属性需要在被 start 的目标 Activity 在 AndroidManifest.xml 文件配置 taskAffinity 的值【必须和 startActivity 发其者 Activity 的包名不一样，如果是跳转另一个 App 的话可以

taskAffinity 可以省略】，则会在新标记的 Affinity 所存在的 taskAffinity 中压入这个 Activity。

- 个人认为在上述 Flag 中 **FLAG_ACTIVITY_NEW_TASK** 是最为重要的

一个 flag , 同时也需要注意的是网上有很多是瞎说的 ; 而且也是个人

唯一一个在实际开发中应用过的属性 ; 先说说个人应用示例 :

- 1. 在 Service 中启动 Activity ;
- 2. App 为系统 Launcher 时 , 跳转到微信无法退出时用到 ;

至于为啥看下文开发的案例就知道了。

(四)、startActivity 场景

Activity 的启动模式的应用的设置是和它的开发场景有关系的 , 在 App 中打开新的 Activity 的基本上分为两种情况 :

1. 目标是本应用中的 , 即它的启动模式是可以直接在 manifest 中配置或者默认为 standard , 任务栈也可以自己随意设置 ;
2. 目标是第三方中的 这个时候就需要先考虑打开新 Activity 的是和自己 App 放在同一任务栈中还是新的 task 中 【这个是很重要的因为在 Android 的机制中 : 同一个任务栈中的 activity 的生命周期是和这个 task 相关联的[具体实例见下文]】 , 然后考虑 Activity 的启动模式 ; 所以 Android 提供了优先级更高的设置方式在 Intent.setFlags(int flags), 通过这 setFlags 就可以为打开第三方的 App 中 Activity 设置任务栈和启动模式了 , 具体设置就自己去看源码了。

二、Activity 四种启动模式常见使用场景：

这也是面试中最为常见的面试题；当然也是个人工作经验和借鉴网友博文，如有错误纰漏尽请诸位批评指正；

LaunchMode	Instance
standard	邮件、manifest中没有配置就默认标准模式
singleTop	登录页面、WXPayEntryActivity、WXEntryActivity、推送通知栏
singleTask	程序模块逻辑入口：主页面（Fragment的containerActivity）、WebView页面、扫一扫页面、电商中：购物界面，确认订单界面，付款界面
singleInstance	系统Launcher、锁屏键、来电显示等系统应用

三、启动模式在实际开发中小插曲

最近的项目就出现了一个由于启动模式导致的问题，先开还是一脸懵还是同事最先想到问题缘由于是做个记录长个教训吧。最近搞的 App 是在一个 cpu 比较 low 的 Android 系统的设备上跑的，而且是 App 是作为 LAUNCHER 启动的，并且在 App 的一个功能就是点击直接跳转到微信登录 LauncherUI 页[如果有登录会自动到聊天页面]的，这个听起也是很普通的一个功能。但是还是出了一个小问题。

（一）、问题复现：

开机后 App 作为 launcher 启动，然后打开微信 LauncherUI 加载页【未登录微信】到微信登录页，然后点返回键【系统返回键】不能也退出微信返回到原来我们的 launcher 的 App 中去了，然后 Android 大法在底部导航栏中查看系统任务【有那些应用在后台】，结果显示一个也没有；这样就是一时间尴尬了退不出微信了；

(二)、问题定位：

1 点击返回键【系统返回键】微信页面没有退出：微信的登录页面 Activity 拦截了返回键事件【只是将应用隐藏在后台不退出，然后回到桌面】，所以登录 Activity 没有销毁；

2 系统导航栏查看正在运行的应用为空，实际上是正在运行我们的 App[launcher 而且放在在 system/app 下]和微信的，两个问题缘由：

1. 我们的 App 作为 launcher 被 Android 作为系统应用任务栈了所以没有显示；
2. 微信是在 launcherApp 中打开的而且打开代码是并没有设置微信 LauncherUI 的启动模式和任务栈，Android 默认是在同一个任务栈中了，所以在查看任务栈时看不到微信应用，而且由于 App 是 launcher 作为系统任务就显示没有应用程序了；

(三)、解决办法

我们的期望是即使不登录微信同样点返回键就可以直接返回到我们 App 的页面中；

根据已经点位到问题进行解决一个基本思路是：用户点返回键后杀死微信的 Activity，回到我们 App 页面中；

具体代码就是

```
//修改之前  
Intent intent = new Intent();  
intent.setClassName("com.tencent.mm", "com.tencent.mm.ui.LauncherUI");  
context.startActivity(intent);
```

```
//修改之后是这样的  
Intent intent = new Intent();  
intent.setClassName("com.tencent.mm", "com.tencent.mm.ui.LauncherUI");  
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);  
context.startActivity(intent);
```

其实代码中只是设置新打开微信在新的任务栈中，这样就是及时在微信 LauncherUI 的页面中拦截了返回键【只是将应用隐藏在后台不退出然后回到桌面】，这样我们 App 最为 launcher，所以就回到我们 App 了。同样这样在在导航栏中查看真正运行任务就可以看到微信了；

(四) 、新的问题

在添加上 Intent.FLAG_ACTIVITY_NEW_TASK 之后是需要新建一个任务栈打开 App 的，虽然是解决了不能返回的问题，但是也引发新的问题：

```
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
```

1 用户点返回键后微信并没有退出：在界面上看到是回到桌面 App 中了，其实微信**并没有退出**任然在后台运行；这个问题其实普遍存在，而且开发者基本上都是对用户采取理性欺骗【或者隐瞒】，现实中用户也是都已经接受；

2 点击跳转到微信明显卡顿：以前打开新的 App 是在一个任务栈中的，打开新 App 页面和在同一 App 页面跳转感觉基本上是一瞬间的，用户感觉不到是在 App 之间跳转的，设置 Intent.FLAG_ACTIVITY_NEW_TASK 之后是需要新建一个任务栈打开 App 的，点击后有明显的停顿【CPU 越差劲越明显】用户体验明显变差了。

四、Activity 中其他常见问题

(一) 生命周期

1.正常生命周期：

onCreate() onStart() onResume() onPause() onStop() onDestroy()

```
21:40:46.479 14501-14501/com.music E/com.music.MainActivity: onCreate: ----->
21:40:46.484 14501-14501/com.music E/com.music.MainActivity: onStart: ----->
21:40:46.488 14501-14501/com.music E/com.music.MainActivity: onResume: ----->
21:40:49.459 14501-14501/com.music E/com.music.MainActivity: onPause: ----->
21:40:49.675 14501-14501/com.music E/com.music.MainActivity: onStop: ----->
21:40:49.675 14501-14501/com.music E/com.music.MainActivity: onDestroy: ----->
```

3. 横竖屏切换

(1).未设置

```
21:46:12.197 14501-14501/com.music E/com.music.MainActivity: onCreate: ----->
21:46:12.202 14501-14501/com.music E/com.music.MainActivity: onStart: ----->
21:46:12.206 14501-14501/com.music E/com.music.MainActivity: onResume: ----->
21:46:16.671 14501-14501/com.music E/com.music.MainActivity: onPause: ----->
21:46:16.672 14501-14501/com.music E/com.music.MainActivity: onStop: ----->
21:46:16.673 14501-14501/com.music E/com.music.MainActivity: onDestroy: ----->
21:46:16.742 14501-14501/com.music E/com.music.MainActivity: onCreate: ----->
21:46:16.749 14501-14501/com.music E/com.music.MainActivity: onStart: ----->
21:46:16.756 14501-14501/com.music E/com.music.MainActivity: onResume: ----->
https://blog.csdn.net/black\_bird\_cn
```

configChanges :

```
21:51:23.812 16293-16293/com.music E/com.music.MainActivity: onCreate: ----->
21:51:23.820 16293-16293/com.music E/com.music.MainActivity: onStart: ----->
21:51:23.824 16293-16293/com.music E/com.music.MainActivity: onResume: ----->
```

(一)、Intent 的基本应用

在 Android 中 Intent 是在四大组件之间进行交互与通讯，也可以在应用之间通讯。其底层的通信是以 Binder 机制实现的，在物理层则是通过共享内存的方式实现的。

1.Intent 属性

Intent 的属性有 :component(组件)、action、category、data、type、extras、

flags ; 所有的属性也是各显神通 , 满足开发者的各种需要满足不同场景 ;

component: 显然就是设置四大组件的 , 将直接使用它指定的组件 , 借助这一属性可以实现不同应用组件之间通讯 ;

action : 是一个可以指定目标组件行为的字符串 , 开发人员可以自定义 action 通过匹配 action 实现组件之间的隐士跳转 , 当然 Android 系统也已经预定部分 String 作为系统应用 Action , 例如打开系统设置页面等等 ;

data : 通常是 URI 类型或者 MIME 类型格式定义的操作数据 ; 表示与动作要操作的数据

- data中常见的MIME类型数据 ;
tel:// : 号码数据格式 , 后跟电话号码。
mailto:// : 邮件数据格式 , 后跟邮件收件人地址。
smsto:// : 短信数据格式 , 后跟短信接收号码。
content:// : 内容数据格式 , 后跟需要读取的内容。
file:// : 文件数据格式 , 后跟文件路径。
market://search?q=pname:pkgname : 市场数据格式 , 在Google Market里搜
- data中URI
data元素组成的URI模型 : scheme://host:port/path 例如 : file:///com.android.jony.test:520/mnt/sdcard ;

Category : 属性用于指定当前动作 (Action) 被执行的环境 ;

type : 对于 data 范例的描写 ;

extras 和 flags 这两个太熟悉了就不在重复 ;

2.Intent 的隐士与显示

显示 Intent 是最长见的 , 也是使用最为频繁的 ; 但是很显然隐士 Intent 更为强大的尤其是在模块化的时代 ;

(二) 、 Activity 异常生命周期与应用

这个过程中几个核心的函数和参数是 : Bundle[onCreate(Bundle

savedInstanceState)] ,

;

1. 系统配置改变引起异常 ;

这里的系统配置改变是指由于横竖屏切换等引起的 Activity 生命周期的变化 ,进而引发的资源的变化。在 Android 中系统配置改变是会引起 Activity 销毁重建的 , 例如横竖屏切换 , 只是切换时间差太小 , 用户眼睛不能察觉而已。 activity 重建的时候就在之前 Activity 销毁前 , 系统会先调用 onSaveInstanceState(Bundle outState) 存储当时各种状态 , 在新建 Activity 中 Android 会通过 onRestoreInstanceState(Bundle savedInstanceState) 读取数据并自动恢复到之前 Activity 的 View , 当然在 Activity 中开发者自己的代码逻辑就需要自己处理啦。

- 触发 onSaveInstanceState(Bundle outState) 的条件 :
 - 1、当用户按下 HOME 键时。
 - 2、从最近应用中选择运行其他的程序时。
 - 3、按下电源按键 (关闭屏幕显示) 时。
 - 4、从当前 activity 启动一个新的 activity 时。
 - 5、屏幕方向切换时 (无论竖屏切横屏还是横屏切竖屏都会调用)。
- 在前 4 种情况下 , 当前 activity 的生命周期为 :
onPause -> onSaveInstanceState -> onStop.

2. 系统回收引起异常

在 Android 系统内存不足时 , 同时 Activity 失去焦点后被系统给回收后 , Activity 再次被创建时 , 通过 onSaveInstanceState 和 onRestoreInstanceState 来存储恢复数据再次显示在屏幕上之前的 View 等状态 ;

3.Activity 异常基础处理

1. 最常见的就是在 `onCreate(Bundle savedInstanceState)` 中通过判断 `savedInstanceState` 是否为空，恢复 Activity 中的数据体；
2. 另一种就是根据应用场景在 `mainfest` 中配置各种参数尽可能减少由于配置参数引起的 Activity 异常；
3. 使用 `onConfigurationChanged` 方法代替 `onRestoreInstanceState` 实现恢复数据逻辑，更高级的自然就是性能优化、实时监视的思路啦；

2、`onNewIntent()`和`onConfigurationChanged()`

背景

如果系统由于系统约束（而不是正常的应用程序行为）而破坏了 Activity，那么尽管实际 Activity 实例已经消失，但是系统还是会记住它已经存在，这样如果用户导航回到它，系统会创建一个新的实例的 Activity 使用一组保存的数据来描述 Activity 在被销毁时的状态。系统用于恢复以前状态的已保存数据称为“实例状态”，是存储在 Bundle 对象中的键值对的集合。

解决

`onSaveInstanceState()` 和 `onRestoreInstanceState()` 就是这样的背景下大展身手了。

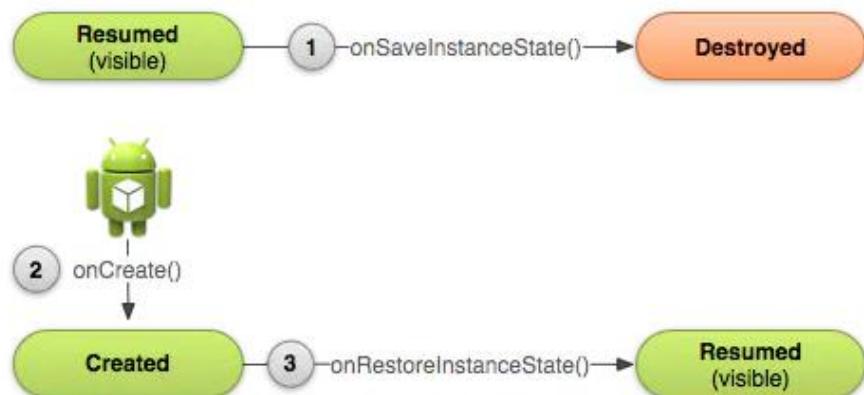
注意

- 1、如果是用户自动按下返回键，或程序调用 `finish()` 退出程序，是不会触发 `onSaveInstanceState()` 和 `onRestoreInstanceState()` 的。
- 2、每次用户旋转屏幕时，您的 Activity 将被破坏并重新创建。当屏幕改变方向时，系统会破坏并重新创建前台 Activity，因为屏幕配置已更改，您的 Activity 可能需要加载替代资源（例如布局）。即会执行 `onSaveInstanceState()` 和 `onRestoreInstanceState()` 的。

介绍

默认情况下，系统使用 Bundle 实例状态来保存有关 View 中 Activity 布局每个对象的信息（例如输入到 `EditText` 对象中的文本值）。因此，如果您的 Activity 实

例被销毁并重新创建，则布局状态会自动恢复到之前的状态。但是，您的 Activity 可能包含更多要恢复的状态信息，例如跟踪 Activity 中用户进度的成员变量。为了让您为 Activity 添加额外的数据到已保存的实例状态，Activity 生命周期中还有一个额外的回调方法，这些回调方法在前面的课程中没有显示。该方法是 `onSaveInstanceState()`，系统在用户离开 Activity 时调用它。当系统调用此方法时，它将传递 Bundle 将在您的 Activity 意外销毁的事件中保存的对象，以便您可以向其中添加其他信息。然后，如果系统在被销毁之后必须重新创建 Activity 实例，它会将相同的 Bundle 对象传递给您的 Activity 的 `onRestoreInstanceState()` 方法以及您的 `onCreate()` 方法。



这是一个简介图

如上图所示：

当系统开始停止您的Activity时，它会调用`onSaveInstanceState()`（1），以便您可以指定要保存的其他状态数据，以防Activity必须重新创建实例。如果Activity被破坏并且必须重新创建相同的实例，则系统将（1）中定义的状态数据传递给`onCreate()`方法（2）和`onRestoreInstanceState()`方法（3）。

保存你的 Activity 状态

当您的 Activity 开始停止时，系统会调用，`onSaveInstanceState()`以便您的 Activity 可以使用一组键值对来保存状态信息。此方法的默认实现保存有关 Activity 视图层次结构状态的信息，例如 `EditText` 小部件中的文本或 `ListView` 的滚动位置。为了保存 Activity 的附加状态信息，您必须实现 `onSaveInstanceState()` 并向对象添加键值对 `Bundle`。例如：

```
static final String STATE_SCORE = "playerScore"; static final String STATE_LEVEL = "playerLevel"; ...  
  
@Override
```

```
public void onSaveInstanceState(Bundle savedInstanceState) {  
  
    // 保存用户自定义的状态  
  
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);  
  
    savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);  
  
  
    // 调用父类交给系统处理，这样系统能保存视图层次结构状态  
  
    super.onSaveInstanceState(savedInstanceState);  
}
```

恢复您的 Activity 状态

当您的 Activity 在之前被破坏后重新创建时，您可以从 Bundle 系统通过您的 Activity 中恢复您的保存状态。这两个方法 `onCreate()` 和 `onRestoreInstanceState()` 回调方法都会收到 Bundle 包含实例状态信息的相同方法。

因为 `onCreate()` 调用该方法是否系统正在创建一个新的 Activity 实例或重新创建一个以前的实例，所以您必须 Bundle 在尝试读取之前检查该状态是否为空。如果它为空，那么系统正在创建一个 Activity 的新实例，而不是恢复之前被销毁的实例。

例如，下面是如何恢复一些状态数据 `onCreate()`:

```
@Overrideprotected void onCreate(Bundle savedInstanceState) {  
  
    super.onCreate(savedInstanceState); // 记得总是调用父类  
  
  
    // 检查是否正在重新创建一个以前销毁的实例  
  
    if (savedInstanceState != null) {  
  
        // 从已保存状态恢复成员的值  
  
        mCurrentScore = savedInstanceState.getInt(STATE_SCORE);  
  
        mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);  
  
    } else {  
  
        // 可能初始化一个新实例的默认值的成员  
  
    }  
}
```

```
...}
```

onCreate() 您可以选择执行 onRestoreInstanceState()，而不是在系统调用 onStart() 方法之后恢复状态。系统 onRestoreInstanceState() 只有在存在保存状态的情况下才会恢复，因此您不需要检查是否 Bundle 为空：

```
public void onRestoreInstanceState(Bundle savedInstanceState) {  
  
    // 总是调用超类，以便它可以恢复视图层次超级  
  
    super.onRestoreInstanceState(savedInstanceState);  
  
    // 从已保存的实例中恢复状态成员  
  
    mCurrentScore = savedInstanceState.getInt(STATE_SCORE);  
  
    mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);}
```

3、onSaveInstanceState()和onRestoreInstanceState()

newConfig: 新的设备配置信息

当系统的配置信息发生改变时，系统会调用此方法。注意，只有在配置文件 `AndroidManifest` 中处理了 `configChanges` 属性 对应的设备配置，该方法才会被调用。如果发生设备配置与在配置文件中设置的不一致，则 **Activity** 会被销毁并使用新的配置重建。

例如：当屏幕方向发生改变时，Activity 会被销毁重建，如果在 `AndroidManifest` 文件中处理屏幕方向配置信息如下：

```
<activity  
    android:name=".MainActivity"  
    android:label="@string/app_name"  
    android:configChanges="orientation/screenSize|"/>
```

则 Activity 不会被销毁重建，而是调用 `onConfigurationChanged` 方法。

如果 `configChanges` 只设置了 `orientation`，则当其他设备配置信息改变时，Activity 依然会销毁重建，且不会调用 `onConfigurationChanged`。

例如，在上面的配置的情况下，如果语言改变了，Activity 就会销毁重建，且不会调用 `onConfigurationChanged` 方法。

configChanges 设置取值

值	说明
"mcc"	IMSI 移动国家/地区代码 (MCC) 发生了变化 - 检测到了 SIM 并更新了 MCC。
"mnc"	IMSI 移动网络代码 (MNC) 发生了变化 - 检测到了 SIM 并更新了 MNC。
"locale"	语言区域发生了变化 - 用户为文本选择了新的显示语言。
"touchscreen"	触摸屏发生了变化。（这种情况通常永远不会发生。）
"keyboard"	键盘类型发生了变化 - 例如，用户插入了一个外置键盘。
"keyboardHidden"	键盘无障碍功能发生了变化 - 例如，用户显示了硬件键盘。
"navigation"	导航类型（轨迹球/方向键）发生了变化。（这种情况通常永远不会发生。）
"screenLayout"	屏幕布局发生了变化 - 这可能是由激活了其他显示方式所致。
"fontScale"	字体缩放系数发生了变化 - 用户选择了新的全局字号。
"uiMode"	用户界面模式发生了变化 - 这可能是因用户将设备放入桌面/车载基座或夜间模式发生变化所致。请参阅 UiModeManager 。 此项为 API 级别 8 中新增配置。
"orientation"	屏幕方向发生了变化 - 用户旋转了设备。 http://blog.csdn.net/q_27570955 注：如果您的应用面向 API 级别 13 或更高级别（按照 <code>minSdkVersion</code> 和 <code>targetSdkVersion</code> 属性所声明的级别），则还应声明 "screenSize" 配置，因为当设备在横向与纵向之间切换时，该配置也会发生变化。
"screenSize"	当前可用屏幕尺寸发生了变化。它表示当前可用尺寸相对于当前纵横比的变化，因此会在用户在横向与纵向之间切换时发生变化。不过，如果您的应用面向 API 级别 12 或更低级别，则 Activity 始终会自行处理此配置变更（即使是在 Android 3.2 或更高版本的设备上运行，此配置变更也不会重新启动 Activity）。 此项为 API 级别 13 中新增配置。
"smallestScreenSize"	物理屏幕尺寸发生了变化。它表示与方向无关的尺寸变化，因此只有在实际物理屏幕尺寸发生变化（如切换到外部显示器）时才会变化。对此配置的变更对应于 <code>smallestWidth</code> 配置的变化。不过，如果您的应用面向 API 级别 12 或更低级别，则 Activity 始终会自行处理此配置变更（即使是在 Android 3.2 或更高版本的设备上运行，此配置变更也不会重新启动 Activity）。 此项为 API 级别 13 中新增配置。
"layoutDirection"	布局方向发生了变化。例如，从左至右 (LTR) 更改为从右至左 (RTL)。 此项为 API 级别 17 中新增配置。

点击可查看清晰大图

注意：横竖屏切换的属性是 `orientation`。如果 `targetSdkVersion` 的值大于等于 13，则如下配置才会回调 `onConfigurationChanged` 方法：

`android:configChanges="orientation|screenSize"`

如果 `targetSdkVersion` 的值小于 13，则只要配置：

`android:configChanges="orientation"`

网上有很多文章写说横竖屏切换时 `onConfigurationChanged` 方法 没有调用，使用如下的配置：

`android:configChanges="orientation|keyboard|keyboardHidden"`

但是，其实查官方文档，只要配置 `android:configChanges="orientation|screenSize"` 就可以了。

扩展：当用户接入一个外设键盘时，默认软键盘会自动隐藏，系统自动使用外设键盘。这个过程 Activity 的销毁和隐藏执行了两次。并且 `onConfigurationChanged()` 不会调用。

但是在配置文件中设置 `android:configChanges="keyboardHidden|keyboard"`。当接入外设键盘或者拔出外设键盘时，调用的周期是先调用 `onConfigurationChanged()` 周期后销毁重建。

在这里有一个疑点，为什么有两次的销毁重建？

其中一次的销毁重建可以肯定是因为外设键盘的插入和拔出。当设置 android:configChanges="keyboardHidden|keyboard" 之后，就不会销毁重建，而是调用 onConfigurationChanged()方法。

但是还有一次销毁重建一直存在。

经过测试，当接入外设键盘时，除了键盘类型的改变，触摸屏也发生了变化。因为使用外设键盘，触摸屏不能使用了。（如果是接入触摸板，不知道会不会有这个问题？欢迎大家提供意见）。这里，我接入的是键盘，所以触摸屏不能使用了。

总结：如果是键盘类型发生了改变，则 configChanges 属性 配置如下 Activity 才不会销毁重建，且回调 onConfigurationChanged 方法：

```
<activity
    android:name=".MainActivity"
    android:label="@string/app_name"
    android:configChanges="keyboard|keyboardHidden|touchscreen" |
```

note:这里的外置物理键盘可以是游戏手柄、扫描枪、键盘等等。

官方文档：

<https://developer.android.com/guide/topics/manifest/activity-element.html>

小楠篇

在手机 APP 开发的时候，一般默认会适配竖屏，游戏开发除外。但是在 Android 平板电脑开发中，屏幕旋转的问题比较突出，可以说，平板电脑的最初用意就是横屏使用的，比较方便，用户会经常旋转我们设备的屏幕。

屏幕旋转的适配问题以及遇到的一些坑

<http://www.jianshu.com/p/19393bb08e4f>

上面我的文章中提到了一些坑，包括 View 的测量不准确，

onConfigurationChanged 的回调不确定，今天主要分析一下

onConfigurationChanged 调用的不确定性因素。

关于这个问题，笔者在网上搜索了一下关于为什么 onConfigurationChanged 的方法不会被调用，基本都是说清单文件里面没有正确配置，因为在 Android2.3 以后需要增加 screenSize 这个配置。

但是完全搜索不到关于我的问题的搜索结果，毕竟做 Android 平板的并不多，因此写下来记录自己的学习过程。

关于官方文档

我们知道，在 Activity、View（ViewGroup）、Fragment、Service、Content Provider 等等在设备的配置发生变化的时候，会回调 onConfigurationChanged 的方法。

实质上主要是 Activity 中收到 AMS 的通知，回调，然后把事件分发到 Window、Fragment、ActionBar 等。

下面我们可以从 Activity 的 onConfigurationChanged 方法 源码可以看到：

```

public void onConfigurationChanged(Configuration newConfig) {
    mCalled = true;

    // 分发到Activity中的所有Fragment
    mFragments.dispatchConfigurationChanged(newConfig);

    // 分发到Activity的Window对象
    if (mWindow != null) {
        // Pass the configuration changed event to the window
        mWindow.onConfigurationChanged(newConfig);
    }

    // 分发到Activity的ActionBar
    if (mActionBar != null) {
        mActionBar.onConfigurationChanged(newConfig);
    }
}

```

这里我们讨论的是为什么当我们的界面在设备配置发生变化的时候（屏幕旋转），有时候并不会回调 `onConfigurationChanged` 呢？

关于 `Activity` 的官方文档有下面一句话：

onConfigurationChanged Added in API level 1

```

void onConfigurationChanged (Configuration newConfig)

```

Called by the system when the device configuration changes while your activity is running. Note that this will only be called if you have selected configurations you would like to handle with the `configChanges` attribute in your manifest. If any configuration change occurs that is not selected to be reported by that attribute, then instead of reporting it the system will stop and restart the activity (to have it launched with the new configuration).

At the time that this function has been called, your Resources object will have been updated to return resource values matching the new configuration.

Parameters
<code>newConfig</code> Configuration: The new device configuration.

也就是说，在设备配置发生变化的时候，会回调 `onConfigurationChanged`，但是前提条件是当你的 `Activity`（组件）还在运行的时候。

这就很明显了，说明一旦你的界面暂停以后就不会回调这个方法了。但是这样会导致一个问题，就是你的界面跳转到其他界面的时候（当前界面暂停），然后发生了一次屏幕旋转，再返回的时候，你的界面虽然旋转了，但是并没有回调 `onConfigurationChanged` 方法，并没有执行你的 UI 适配代码。

源码分析

想到四大组件，我们第一时间应该会想到 `AMS(ActivityManagerService)`，没错，今天我们的始发站就是 `AMS`。

在 `AMS` 里面搜索了一下关键字 `Configuration`，发现了 `updateConfigurationLocked` 这个方法（没有说明的情况下，都是只给出省略版）：

相信眼尖的朋友一定会看出来，在这里由 AMS 创建了 Configuration 对象，然后通过进程间通信，通知我们的 app 进程。

```
private boolean updateConfigurationLocked(Configuration values, ActivityRecord starting,
    boolean initLocale, boolean persistent, int userId, boolean deferResume) {
    int changes = 0;

    if (m.WindowManager != null) {
        m.WindowManager.deferSurfaceLayout();
    }
    if (values != null) {
        //创建Configuration对象
        Configuration newConfig = new Configuration(m.Configuration);
        changes = newConfig.updateFrom(values);
        if (changes != 0) {
            for (int i=m.LruProcesses.size()-1; i>=0; i--) {
                ProcessRecord app = m.LruProcesses.get(i);
                try {
                    if (app.thread != null) {
                        //通过进程间通信，通知我们的app进程
                        app.thread.scheduleConfigurationChanged(configCopy);
                    }
                } catch (Exception e) {
                }
            }
        }
    }
}
```

thread 是一个 IApplicationThread 对象，继承了 IInterface 接口，也就是说是一个 AIDL 对象，实际上这个接口的实现类是 ActivityThread 里面的内部类 ApplicationThread。

public interface IApplicationThread extends IInterface {}

那么就是说这时候 AMS 通过 IApplicationThread 进行了进程间通信，实际上调用了我们 APP 所在的进程的 ActivityThread 里面的内部类 ApplicationThread 的 scheduleConfigurationChanged 方法：

```
public void scheduleConfigurationChanged(Configuration config) {
    updatePendingConfiguration(config);
    sendMessage(H.CONFIGURATION_CHANGED, config);
}
```

这个方法很简单，就是发送消息给我们的应用程序的系统 Handler，然后由它来处理消息，下面继续分析处理消息的过程：

```
case CONFIGURATION_CHANGED:
    mCurDefaultDisplayDpi = ((Configuration)msg.obj).densityDpi;
    mUpdatingSystemConfig = true;
    handleConfigurationChanged((Configuration)msg.obj, null);
    mUpdatingSystemConfig = false;
    break;
```

这里继续调用了 handleConfigurationChanged 方法:

```
final void handleConfigurationChanged(Configuration config, CompatibilityInfo compat) {
    //收集需要回调onConfigurationChanged的组件信息
    ArrayList<ComponentCallbacks2> callbacks = collectComponentCallbacks(false, config);
    if (callbacks != null) {
        final int N = callbacks.size();
        for (int i=0; i<N; i++) {
            ComponentCallbacks2 cb = callbacks.get(i);
            if (cb instanceof Activity) {
                //如果当前循环的组件是Activity，那么回调Activity的onConfigurationChanged
                Activity a = (Activity) cb;
                performConfigurationChangedForActivity(mActivities.get(a.getActivityToken()),
                    config, REPORT_TO_ACTIVITY);
            } else {
                //如果当前循环不是Activity，比如说是Service等，也需要回调
                performConfigurationChanged(cb, null, config, null, REPORT_TO_ACTIVITY);
            }
        }
    }
}
```

这个方法首先收集需要回调 onConfigurationChanged 的组件信息，如果当前循环的组件是 Activity，那么通过调用 performConfigurationChangedForActivity 方法回调 Activity 的 onConfigurationChanged。

如果当前循环不是 Activity，比如说是 Service 等，也需要 performConfigurationChanged 进行相应回调。

下面我们先看 performConfigurationChangedForActivity 这个方法:

```
private void performConfigurationChangedForActivity(ActivityClientRecord r,
    Configuration newBaseConfig, boolean reportToActivity) {
    r.tmpConfig.setTo(newBaseConfig);
    if (r.overrideConfig != null) {
        r.tmpConfig.updateFrom(r.overrideConfig);
    }
    performConfigurationChanged(r.activity, r.token, r.tmpConfig,
        r.overrideConfig, reportToActivity);
    freeTextLayoutCachesIfNeeded(r.activity.mCurrentConfig.diff(r.tmpConfig));
}
```

实际上也会调用 performConfigurationChanged 方法，这里最终会回调 Activity 的 onConfigurationChanged 方法:

```

private void performConfigurationChanged(ComponentCallbacks2 cb,
                                         IBinder activityToken,
                                         Configuration newConfig,
                                         Configuration amOverrideConfig,
                                         boolean reportToActivity) {
    Activity activity = (cb instanceof Activity) ? (Activity) cb : null;

    if (shouldChangeConfig) {
        if (reportToActivity) {
            final Configuration configToReport = createNewConfigAndUpdateIfNotNull(
                newConfig, contextThemeWrapperOverrideConfig);

            //回调Activity的onConfigurationChanged方法
            cb.onConfigurationChanged(configToReport);
        }

        //这里有个注意点，就是我们需要先调用super的onConfigurationChanged方法，
        //父类的方法中会把mCalled置为true。
        //因为上文提到，父类的方法需要进行一次分发。否则就会抛出SuperNotCalledException。
        if (activity != null) {
            if (reportToActivity && !activity.mCalled) {
                throw new SuperNotCalledException(
                    "Activity " + activity.getLocalClassName() +
                    " did not call through to super.onConfigurationChanged()");
            }
            activity.mConfigChangeFlags = 0;
            activity.mCurrentConfig = new Configuration(newConfig);
        }
    }
}
}

```

这里有个注意点，就是我们需要先调用 `super` 的 `onConfigurationChanged` 方法，父类的方法中会把 `mCalled` 置为 `true`。

因为上文提到，父类的方法需要进行一次分发。否则就会抛出

`SuperNotCalledException`。

我们的问题还没有解决，就是为什么我们的组件在暂停以后并不会回调呢？问题的核心代码就出在收集组件信息的时候，我们回到 `ActivityThread` 的系统 `Handler` 的 `handleConfigurationChanged` 方法中：

```

//收集需要回调onConfigurationChanged的组件信息
ArrayList<ComponentCallbacks2> callbacks = collectComponentCallbacks(false, config);

```

这里收集了组件的信息，下面我们点进去 `collectComponentCallbacks` 这个方法瞄一眼：

```

ArrayList<ComponentCallbacks2> collectComponentCallbacks(
    boolean allActivities, Configuration newConfig) {
    //初始化一个ArrayList用于存储需要回调的组件信息
    ArrayList<ComponentCallbacks2> callbacks
        = new ArrayList<ComponentCallbacks2>();

    synchronized (mResourcesManager) {
        //拿到所有Application对象
        final int NAPP = mAllApplications.size();
        for (int i=0; i<NAPP; i++) {
            callbacks.add(mAllApplications.get(i));
        }
        final int NACT = mActivities.size();
        for (int i=0; i<NACT; i++) {
            //拿到所有Activity
            ActivityClientRecord ar = mActivities.valueAt(i);
            Activity a = ar.activity;
            if (a != null) {
                Configuration thisConfig = applyConfigCompatMainThread(
                    mCurDefaultDisplayDpi, newConfig,
                    ar.packageInfo.getCompatibilityInfo());
                if (!ar.activity.mFinished && (allActivities || !ar.paused)) {
                    // If the activity is currently resumed, its configuration
                    // needs to change right now.
                    //如果当前的Activity是resumed状态的时候，需要马上回调
                    callbacks.add(a);
                } else if (thisConfig != null) {
                    ar.newConfig = thisConfig;
                }
            }
        }
        //收集所有的Service信息
        final int NSVC = mServices.size();
        for (int i=0; i<NSVC; i++) {
            callbacks.add(mServices.valueAt(i));
        }
    }
    //收集所有Content Provider
    synchronized (mProviderMap) {
        final int NPRV = mLocalProviders.size();
        for (int i=0; i<NPRV; i++) {
            callbacks.add(mLocalProviders.valueAt(i).mLocalProvider);
        }
    }
    return callbacks;
}

```

这个方法初始化一个 `ArrayList` 用于存储需要回调的组件信息，然后收集了当前应用的所有 `Application` 对象（多进程的时候可能就会有多个），`Activity`, `Service`, `Content Provider` 信息，然后进行下一步回调。

关键是在收集 Activity 的时候，进行了一次判断：

```
if (!ar.activity.mFinished && (allActivities || !ar.paused)) {  
    // If the activity is currently resumed, its configuration  
    // needs to change right now.  
    //如果当前的Activity是resumed状态的时候，需要马上回调  
    callbacks.add(a);
```

经过源码的分析，已经可以得出这个结论就是：

当 Activity 已经 Finish 掉 或者 已经暂停的时候，并不会把这个 Activity 添加进来，这样做是为了保证系统的效率，只去处理那些活跃（resume）的 Activity，其他的不处理。

解决办法

办法一

我们可以《屏幕旋转的适配问题以及遇到的一些坑》这篇文章中提到的，通过自定义广播的方式去接收 `android.intent.action.CONFIGURATION_CHANGED` 这个广播。注意这个广播只能在 Java 代码中注册才会有效果。

办法二

重写 Activity 的 `onRestart` 代替 `onConfigurationChanged` 方法，只不过需要判断一下当前的屏幕方向：

```
@Override  
protected void onRestart() {  
    super.onRestart();  
    if (isLandOrientation()) {  
        //横屏  
        ...  
    } else {  
        //竖屏  
        ...  
    }  
  
    public boolean isLandOrientation() {  
        if (getResources().getConfiguration().orientation  
            == Configuration.ORIENTATION_LANDSCAPE) {  
            return true;  
        } else {  
            return false;  
        }  
    }
```

自己手动判断一下横竖屏即可。

4、Activity 到底是如何启动的

zygote 是什么？有什么作用？

首先，你觉得这个单词眼熟不？当你的程序 Crash 的时候，打印的红色 log 下面通常带有这一个单词。

zygote 意为 “受精卵”。Android 是基于 Linux 系统的，而在 Linux 中，所有的进程都是由 init 进程直接或者是间接 fork 出来的，zygote 进程也不例外。

在 Android 系统里面，zygote 是一个进程的名字。Android 是基于 Linux System 的，当你的手机开机的时候，Linux 的内核加载完成之后就会启动一个叫 “init”的进程。在 Linux System 里面，所有的进程都是由 init 进程 fork 出来的，我们的 zygote 进程也不例外。

我们都知道，每一个 App 其实都是

一个单独的 dalvik 虚拟机

一个单独的进程

所以当系统里面的第一个 zygote 进程运行之后，在这之后再开启 App，就相当于开启一个新的进程。而为了实现资源共用和更快的启动速度，Android 系统开启新进程的方式，是通过 fork 第一个 zygote 进程实现的。所以说，除了第一个 zygote 进程，其他应用所在的进程都是 zygote 的子进程，这下你明白为什么这个进程叫 “受精卵” 了吧？因为就像是一个受精卵一样，它能快速的分裂，并且产生遗传物质一样的细胞！

SystemServer 是什么？有什么作用？它与 zygote 的关系是什么？

首先我要告诉你的是 ,SystemServer 也是一个进程 ,而且是由 zygote 进程 fork 出来的。

知道了 SystemServer 的本质 ,我们对它就不算太陌生了 ,这个进程是 Android Framework 里面两大非常重要的进程之一——另外一个进程就是上面的 zygote 进程。

为什么说 SystemServer 非常重要呢 ? 因为系统里面重要的服务都是在这个进程里面开启的 ,比如

ActivityManagerService、PackageManagerService、
WindowManagerService 等等 ,看着是不是都挺眼熟的 ?

那么这些系统服务是怎么开启起来的呢 ?

在 zygote 开启的时候 ,会调用 ZygoteInit.main() 进行初始化

```
public static void main(String argv[]) {  
  
    ...ignore some code...  
  
    //在加载首个 zygote 的时候, 会传入初始化参数, 使得 startSystemServer = true  
    boolean startSystemServer = false;  
    for (int i = 1; i < argv.length; i++) {  
        if ("start-system-server".equals(argv[i])) {  
            startSystemServer = true;  
        } else if (argv[i].startsWith(ABI_LIST_ARG)) {  
            abiList = argv[i].substring(ABI_LIST_ARG.length());  
        } else if (argv[i].startsWith(SOCKET_NAME_ARG)) {  
            socketName = argv[i].substring(SOCKET_NAME_ARG.length());  
        } else {  
            throw new RuntimeException("Unknown command line argument: " +  
                argv[i]);  
        }  
    }  
}
```

```
...ignore some code...

//开始 fork 我们的 SystemServer 进程
if (startSystemServer) {
    startSystemServer(abiList, socketName);
}

...ignore some code...

}
```

我们看下 startSystemServer()做了些什么

```
/**留着这个注释，就是为了说明 SystemServer 确实是被 fork 出来的

 * Prepare the arguments and fork for the system server process.

 */

private static boolean startSystemServer(String abiList, String
socketName)

throws MethodAndArgsCaller, RuntimeException {

...ignore some code...
```

```
//留着这段注释，就是为了说明上面 ZygoteInit.main(String argv[])

里面的 argv 就是通过这种方式传递进来的
```

```
/* Hardcoded command line to start the system server */

String args[] = {

    "--setuid=1000",

    "--setgid=1000",
```

```
--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,101  
8,1032,3001,3002,3003,3006,3007",  
"--capabilities=" + capabilities + "," + capabilities,  
"--runtime-init",  
"--nice-name=system_server",  
"com.android.server.SystemServer",  
};
```

```
int pid;  
try {  
    parsedArgs = new ZygoteConnection.Arguments(args);
```

```
ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);
```

```
ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);
```

```
//确实是 fuck 出来的吧，我没骗你吧~不对，是 fork 出来的 -_-|||
```

```
/* Request to fork the system server process */
```

```
pid = Zygote.forkSystemServer(
```

```
parsedArgs.uid, parsedArgs.gid,
```

```
parsedArgs.gids,
```

```
parsedArgs.debugFlags,
```

```
        null,  
        parsedArgs.permittedCapabilities,  
        parsedArgs.effectiveCapabilities);  
  
    } catch (IllegalArgumentException ex) {  
  
        throw new RuntimeException(ex);  
    }  
  
    /* For child process */  
  
    if (pid == 0) {  
  
        if (hasSecondZygote(abiList)) {  
  
            waitForSecondaryZygote(socketName);  
        }  
  
        handleSystemServerProcess(parsedArgs);  
    }  
  
    return true;  
}
```

ActivityManagerService 是什么？什么时候初始化的？有什么作用？

ActivityManagerService，简称 AMS，服务端对象，负责系统中所有 Activity 的生命周期。

ActivityManagerService 进行初始化的时机很明确，就是在 SystemServer 进程开启的时候，就会初始化 ActivityManagerService。从下面的代码中可以看到

```
public final class SystemServer {  
  
    //zygote 的主入口  
  
    public static void main(String[] args) {  
  
        new SystemServer().run();  
  
    }  
  
    public SystemServer() {  
  
        // Check for factory test mode.  
  
        mFactoryTestMode = FactoryTest.getMode();  
  
    }  
  
    private void run() {
```

...ignore some code...

//加载本地系统服务库，并进行初始化

```
System.loadLibrary("android_servers");
```

```
nativeInit();
```

// 创建系统上下文

```
createSystemContext();
```

//初始化 SystemServiceManager 对象，下面的系统服务开启都需要
调用 SystemServiceManager.startService(Class<T>)，这个方法通过反射来
启动对应的服务

```
mSystemServiceManager = new  
SystemServiceManager(mSystemContext);
```

//开启服务

```
try {
```

```
    startBootstrapServices();

    startCoreServices();

    startOtherServices();

} catch (Throwable ex) {

    Slog.e("System",
"***** Failure starting system
services", ex);

    throw ex;

}

...ignore some code...

}
```

//初始化系统上下文对象 mSystemContext，并设置默认的主题，mSystemContext 实际上是一个 ContextImpl 对象。调用

ActivityThread.systemMain()的时候，会调用 ActivityThread.attach(true)，而在 attach()里面，则创建了 Application 对象，并调用了 Application.onCreate()。

```
private void createSystemContext() {  
  
    ActivityThread activityThread = ActivityThread.systemMain();  
  
    mSystemContext = activityThread.getSystemContext();  
  
  
  
    mSystemContext.setTheme(android.R.style.Theme_DeviceDefault_Light_  
DarkActionBar);  
  
}
```

//在这里开启了几个核心的服务，因为这些服务之间相互依赖，所以都放在了这个方法里面。

```
private void startBootstrapServices() {  
  
    ...ignore some code...
```

```
//初始化 ActivityManagerService

mActivityManagerService =
mSystemServiceManager.startService(
    ActivityManagerService.Lifecycle.class).getService();

mActivityManagerService.setSystemServiceManager(mSystemServiceMa
nager);

//初始化 PowerManagerService , 因为其他服务需要依赖这个
Service , 因此需要尽快的初始化

mPowerManagerService =
mSystemServiceManager.startService(PowerManagerService.class);

// 现在电源管理已经开启 , ActivityManagerService 负责电源管理功
能

mActivityManagerService.initPowerManagement();

// 初始化 DisplayManagerService
```

```
mDisplayManagerService =  
mSystemServiceManager.startService(DisplayManagerService.class);  
  
//初始化 PackageManagerService  
  
mPackageManagerService =  
PackageManagerService.main(mSystemContext, mInstaller,  
  
mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF,  
mOnlyCore);  
  
...ignore some code...  
  
}  
  
}
```

经过上面这些步骤，我们的 ActivityManagerService 对象已经创建好了，并且完成了成员变量初始化。而且在这之前，调用 createSystemContext() 创建系统上下文的时候，也已经完成了 mSystemContext 和 ActivityThread 的创建。注

意，这是系统进程开启时的流程，在这之后，会开启系统的 Launcher 程序，完成系统界面的加载与显示。

你是否会好奇，我为什么说 AMS 是服务端对象？下面我给你介绍下 Android 系统里面的服务器和客户端的概念。

其实服务器客户端的概念不仅仅存在于 Web 开发中，在 Android 的框架设计中，使用的也是这一种模式。服务器端指的就是所有 App 共用的系统服务，比如我们这里提到的 ActivityManagerService，和前面提到的 PackageManagerService、WindowManagerService 等等，这些基础的系统服务是被所有的 App 公用的，当某个 App 想实现某个操作的时候，要告诉这些系统服务，比如你想打开一个 App，那么我们知道了包名和 MainActivity 类名之后就可以打开

```
Intent intent = new Intent(Intent.ACTION_MAIN);

intent.addCategory(Intent.CATEGORY_LAUNCHER);

ComponentName cn = new ComponentName(packageName,
className);

intent.setComponent(cn);

startActivity(intent);
```

但是，我们的 App 通过调用 startActivity() 并不能直接打开另外一个 App，这个方法会通过一系列的调用，最后还是告诉 AMS 说：“我要打开这个 App，我

知道他的住址和名字 ,你帮我打开吧 !” 所以是 AMS 来通知 zygote 进程来 fork 一个新进程 , 来开启我们的目标 App 的。这就像是浏览器想要打开一个超链接一样 , 浏览器把网页地址发送给服务器 , 然后还是服务器把需要的资源文件发送给客户端的。

知道了 Android Framework 的客户端服务器架构之后 , 我们还需要了解一件事情 , 那就是我们的 App 和 AMS(SystemServer 进程)还有 zygote 进程分属于三个独立的进程 , 他们之间如何通信呢 ?

App 与 AMS 通过 Binder 进行 IPC 通信 , AMS(SystemServer 进程)与 zygote 通过 Socket 进行 IPC 通信。

那么 AMS 有什么用呢 ? 在前面我们知道了 , 如果想打开一个 App 的话 , 需要 AMS 去通知 zygote 进程 , 除此之外 , 其实所有的 Activity 的开启、暂停、关闭都需要 AMS 来控制 , 所以我们说 , AMS 负责系统中所有 Activity 的生命周期。

在 Android 系统中 , 任何一个 Activity 的启动都是由 AMS 和应用程序进程 (主要是 ActivityThread) 相互配合来完成的。 AMS 服务统一调度系统中所有进程的 Activity 启动 , 而每个 Activity 的启动过程则由其所属的进程具体来完成。

这样说你可能还是觉得比较抽象 , 没关系 , 下面有一部分是专门来介绍 AMS 与 ActivityThread 如何一起合作控制 Activity 的生命周期的。

Launcher 是什么 ? 什么时候启动的 ?

当我们点击手机桌面上的图标的时候，App 就由 Launcher 开始启动了。但是，你有没有思考过 Launcher 到底是一个什么东西？

Launcher 本质上也是一个应用程序，和我们的 App 一样，也是继承自 Activity

packages/apps/Launcher2/src/com/android/launcher2/Launcher.java

```
public final class Launcher extends Activity
```

```
    implements View.OnClickListener, OnLongClickListener,  
    LauncherModel.Callbacks,
```

```
    View.OnTouchListener {
```

```
}
```

Launcher 实现了点击、长按等回调接口，来接收用户的输入。既然是普通的 App，那么我们的开发经验在这里就仍然适用，比如，我们点击图标的时候，是怎么开启的应用呢？如果让你，你怎么做这个功能呢？捕捉图标点击事件，然后 startActivity() 发送对应的 Intent 请求吧！是的，Launcher 也是这么做的，就是这么 easy！

那么到底是处理的哪个对象的点击事件呢？既然 Launcher 是 App，并且有界面，那么肯定有布局文件呀，是的，我找到了布局文件 launcher.xml

```
<FrameLayout
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
xmlns:launcher="http://schemas.android.com/apk/res/com.android.laun  
cher"
```

```
    android:id="@+id/launcher">
```

```
        <com.android.launcher2.DragLayer
```

```
            android:id="@+id/drag_layer"
```

```
            android:layout_width="match_parent"
```

```
            android:layout_height="match_parent"
```

```
            android:fitsSystemWindows="true">
```

```
        <!-- Keep these behind the workspace so that they are not  
            visible when
```

```
                we go into AllApps -->
```

```
        <include
```

```
            android:id="@+id/dock_divider"
```

```
            layout="@layout/workspace_divider"
```

```
    android:layout_marginBottom="@dimen/button_bar_height"

        android:layout_gravity="bottom" />

<include

    android:id="@+id/paged_view_indicator"

    layout="@layout/scroll_indicator"

    android:layout_gravity="bottom"

    android:layout_marginBottom="@dimen/button_bar_height" />

<!-- The workspace contains 5 screens of cells -->

<com.android.launcher2.Workspace

    android:id="@+id/workspace"

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:paddingStart="@dimen/workspace_left_padding"
```

```
    android:paddingEnd="@dimen/workspace_right_padding"

    android:paddingTop="@dimen/workspace_top_padding"

    android:paddingBottom="@dimen/workspace_bottom_padding"

    launcher:defaultScreen="2"

    launcher:cellCountX="@integer/cell_count_x"

    launcher:cellCountY="@integer/cell_count_y"

    launcher:pageSpacing="@dimen/workspace_page_spacing"

    launcher:scrollIndicatorPaddingLeft="@dimen/workspace_divider_padding_left"

    launcher:scrollIndicatorPaddingRight="@dimen/workspace_divider_padding_right">

        <include android:id="@+id/cell1"
layout="@layout/workspace_screen" />
```

```
<include android:id="@+id/cell2"
layout="@layout/workspace_screen" />

<include android:id="@+id/cell3"
layout="@layout/workspace_screen" />

<include android:id="@+id/cell4"
layout="@layout/workspace_screen" />

<include android:id="@+id/cell5"
layout="@layout/workspace_screen" />

</com.android.launcher2.Workspace>
```

...ignore some code...

```
</com.android.launcher2.DragLayer>

</FrameLayout>
```

为了方便查看，我删除了很多代码，从上面这些我们应该可以看出一些东西来：Launcher 大量使用标签来实现界面的复用，而且定义了很多的自定义控件实现界面效果，dock_divider 从布局的参数声明上可以猜出，是底部操作栏和上面图标布局的分割线，而 paged_view_indicator 则是页面指示器，和 App 首次

进入的引导页下面的界面引导是一样的道理。当然，我们最关心的是 Workspace 这个布局，因为注释里面说在这里面包含了 5 个屏幕的单元格，想必你也猜到了，这个就是在首页存放我们图标的那五个界面(不同的 ROM 会做不同的 DIY，数量不固定)。

接下来，我们应该打开 workspace_screen 布局，看看里面有什么东东。

workspace_screen.xml

```
<com.android.launcher2.CellLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
  
    xmlns:launcher="http://schemas.android.com/apk/res/com.android.laun  
cher"  
  
    android:layout_width="wrap_content"  
  
    android:layout_height="wrap_content"  
  
    android:paddingStart="@dimen/cell_layout_left_padding"  
  
    android:paddingEnd="@dimen/cell_layout_right_padding"  
  
    android:paddingTop="@dimen/cell_layout_top_padding"  
  
    android:paddingBottom="@dimen/cell_layout_bottom_padding"
```

```
        android:hapticFeedbackEnabled="false"

        launcher:cellWidth="@dimen/workspace_cell_width"

        launcher:cellHeight="@dimen/workspace_cell_height"

        launcher:widthGap="@dimen/workspace_width_gap"

        launcher:heightGap="@dimen/workspace_height_gap"

        launcher:maxGap="@dimen/workspace_max_gap" />
```

里面就一个 CellLayout，也是一个自定义布局，那么我们就可以猜到了，既然可以存放图标，那么这个自定义的布局很有可能是继承自 ViewGroup 或者是其子类，实际上，CellLayout 确实是继承自 ViewGroup。在 CellLayout 里面，只放了一个子 View，那就是 ShortcutAndWidgetContainer。从名字也可以看出来，ShortcutAndWidgetContainer 这个类就是用来存放**快捷图标**和**Widget 小部件**的，那么里面放的是什么对象呢？

在桌面上的图标，使用的是 BubbleTextView 对象，这个对象在 TextView 的基础之上，添加了一些特效，比如你长按移动图标的时候，图标位置会出现一个背景(不同版本的效果不同)，所以我们找到 BubbleTextView 对象的点击事件，就可以找到 Launcher 如何开启一个 App 了。

除了在桌面上有图标之外，在程序列表中点击图标，也可以开启对应的程序。这里的图标使用的不是 BubbleTextView 对象，而是 PagedViewIcon 对象，我们如果找到它的点击事件，就也可以找到 Launcher 如何开启一个 App。

其实说这么多，和今天的主题隔着十万八千里，上面这些东西，你有兴趣就看，没兴趣就直接跳过，不知道不影响这篇文章阅读。

BubbleTextView 的点击事件在哪里呢？我来告诉你：在 Launcher.onClick(View v) 里面。

```
/**
```

```
* Launches the intent referred by the clicked shortcut
```

```
*/
```

```
public void onClick(View v) {
```

```
...ignore some code...
```

```
Object tag = v.getTag();
```

```
if (tag instanceof ShortcutInfo) {
```

```
// Open shortcut
```

```
final Intent intent = ((ShortcutInfo) tag).intent;
```

```
int[] pos = new int[2];
```

```
v.getLocationOnScreen(pos);
```

```
    intent.setSourceBounds(new Rect(pos[0], pos[1],
        pos[0] + v.getWidth(), pos[1] + v.getHeight()));

    //开始开启 Activity 咯~

    boolean success = startActivitySafely(v, intent, tag);

    if (success && v instanceof BubbleTextView) {

        mWaitingForResume = (BubbleTextView) v;

        mWaitingForResume.setStayPressed(true);

    }

} else if (tag instanceof FolderInfo) {

    //如果点击的是图标文件夹，就打开文件夹

    if (v instanceof FolderIcon) {

        FolderIcon fi = (FolderIcon) v;

        handleFolderClick(fi);

    }

} else if (v == mAllAppsButton) {
```

```
...ignore some code...
```

```
}
```

```
}
```

从上面的代码我们可以看到，在桌面上点击快捷图标的时候，会调用

```
startActivitySafely(v, intent, tag);
```

那么从程序列表界面，点击图标的时候会发生什么呢？实际上，程序列表界面使用的是 AppsCustomizePagedView 对象，所以我在这个类里面找到了 onClick(View v)。

```
com.android.launcher2.AppsCustomizePagedView.java
```

```
/**
```

```
* The Apps/Customize page that displays all the applications,  
widgets, and shortcuts.
```

```
*/
```

```
public class AppsCustomizePagedView extends  
PagedViewWithDraggableItems implements  
View.OnClickListener, View.OnKeyListener, DragSource,
```

```
PagedViewIcon.PressedCallback,  
PagedViewWidget.ShortPressListener,  
  
    LauncherTransitionable {  
  
        @Override  
        public void onClick(View v) {  
  
            ...ignore some code...  
  
            if (v instanceof PagedViewIcon) {  
  
                mLauncher.updateWallpaperVisibility(true);  
  
                mLauncher.startActivitySafely(v, appInfo.intent,  
                    appInfo);  
  
            } else if (v instanceof PagedViewWidget) {  
  
                ...ignore some code..  
  
            }  
        }  
    }  
}
```

```
}
```

调用了 startActivity(v, intent, tag)

```
boolean startActivity(View v, Intent intent, Object tag) {  
  
    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);  
  
    try {  
  
        boolean useLaunchAnimation = (v != null) &&  
            !intent.hasExtra(INTENT_EXTRA_IGNORE_LAUN  
CH_ANIMATION);  
  
        if (useLaunchAnimation) {  
  
            if (user == null ||  
                user.equals(android.os.Process.myUserHandle())) {  
  
                startActivity(intent, opts.toBundle());  
  
            } else {  
  
                launcherApps.startActivity(intent.getComponent(), user,  
                    opts.toBundle());  
            }  
        } else {  
            launcherApps.startActivity(intent.getComponent(), user);  
        }  
    } catch (Exception e) {  
        Log.w("ActivityManager", "startActivity() failed", e);  
    }  
}
```

```
        intent.getSourceBounds(),  
  
        opts.toBundle());  
  
    }  
  
} else {  
  
    if (user == null ||  
  
user.equals(android.os.Process.myUserHandle())) {  
  
        startActivity(intent);  
  
    } else {  
  
        launcherApps.startActivity(intent.getComponent(), user,  
  
        intent.getSourceBounds(), null);  
  
    }  
  
}  
  
return true;  
  
} catch (SecurityException e) {  
  
    ...  
  
}
```

```
    return false;  
  
}
```

这里会调用 Activity.startActivity(intent, opts.toBundle()) ,这个方法熟悉吗？这就是我们经常用到的 Activity.startActivity(Intent)的重载函数。而且由于设置了

```
intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
```

所以这个 Activity 会添加到一个新的 Task 栈中，而且，startActivity()调用的其实是 startActivityForResult()这个方法。

```
@Override
```

```
public void startActivity(Intent intent, @Nullable Bundle options)  
{  
  
    if (options != null) {  
  
        startActivityForResult(intent, -1, options);  
  
    } else {  
  
        // Note we want to go through this call for compatibility  
        // with  
  
        // applications that may have overridden the method.  
    }  
}
```

```
        startActivityForResult(intent, -1);

    }

}
```

所以我们现在明确了，Launcher 中开启一个 App，其实和我们在 Activity 中直接 startActivity() 基本一样，都是调用了 Activity.startActivityForResult()。

Instrumentation 是什么？和 ActivityThread 是什么关系？

还记得前面说过的 Instrumentation 对象吗？每个 Activity 都持有 Instrumentation 对象的一个引用，但是整个进程只会存在一个 Instrumentation 对象。当 startActivityForResult() 调用之后，实际上还是调用了 mInstrumentation.execStartActivity()

```
public void startActivityForResult(Intent intent, int requestCode,
@Nullable Bundle options) {

    if (mParent == null) {

        Instrumentation.ActivityResult ar =
            mInstrumentation.execStartActivity(
                this, mMainThread.getApplicationThread(),
                mToken, this,
                intent, requestCode, options);
    }
}
```

```
    if (ar != null) {  
  
        mMainThread.sendActivityResult(  
            mToken, mEmbeddedID, requestCode,  
            ar.getResultCode(),  
  
            ar.getResultData());  
  
    }  
  
    ...ignore some code...  
  
} else {  
  
    if (options != null) {  
  
        //当现在的 Activity 有父 Activity 的时候会调用，但是  
        在 startActivityFromChild()内部实际还是调用的  
        mInstrumentation.execStartActivity()  
  
        mParent.startActivityFromChild(this, intent,  
            requestCode, options);  
  
    } else {  
  
        mParent.startActivityFromChild(this, intent,  
            requestCode);  
    }  
}
```

```
    }  
  
}  
  
...ignore some code...
```

```
}
```

下面是 mInstrumentation.execStartActivity() 的实现

```
public ActivityResult execStartActivity(  
    Context who, IBinder contextThread, IBinder token,  
    Activity target,  
    Intent intent, int requestCode, Bundle options) {  
  
    IApplicationThread whoThread = (IApplicationThread)  
        contextThread;  
  
    ...ignore some code...  
  
    try {  
        intent.migrateExtraStreamToClipData();  
  
        intent.prepareToLeaveProcess();  
  
        int result = ActivityManagerNative.getDefault()
```

```
        .startActivity(whoThread,  
        who.getPackageName(), intent,  
  
        intent.resolveTypeIfNeeded(who.getContentResolver()),  
  
        token, target != null ?  
        target.mEmbeddedID : null,  
  
        requestCode, 0, null, options);  
  
    checkStartActivityResult(result, intent);  
  
} catch (RemoteException e) {  
  
}  
  
return null;  
  
}
```

所以当我们在程序中调用 `startActivity()` 的时候，实际上调用的是 `Instrumentation` 的相关的方法。

`Instrumentation` 意为“仪器”，我们先看一下这个类里面包含哪些方法吧。我们可以看到，这个类里面的方法大多数和 `Application` 和 `Activity` 有关，是的，这个类就是完成对 `Application` 和 `Activity` 初始化和生命周期的工具类。比如说，我单独挑一个 `callActivityOnCreate()` 让你看看。

```
public void callActivityOnCreate(Activity activity, Bundle icicle) {  
  
    prePerformCreate(activity);  
  
    activity.performCreate(icicle);  
  
    postPerformCreate(activity);  
  
}
```

对 `activity.performCreate(icicle);` 这一行代码熟悉吗？这一行里面就调用了传说中的 Activity 的入口函数 `onCreate()`，不信？接着往下看

```
Activity.performCreate()  
  
final void performCreate(Bundle icicle) {  
  
    onCreate(icicle);  
  
    mActivityTransitionState.readState(icicle);  
  
    performCreateCommon();  
  
}
```

没骗你吧，`onCreate` 在这里调用了吧。但是有一件事情必须说清楚，那就是这个 `Instrumentation` 类这么重要，为啥我在开发的过程中，没有发现他的踪迹呢？

是的，Instrumentation 这个类很重要，对 Activity 生命周期方法的调用根本就离不开他，他可以说是一个大管家，但是，这个大管家比较害羞，是一个女的，管内不管外，是老板娘~

那么你可能要问了，老板是谁呀？

老板当然是大名鼎鼎的 ActivityThread 了！

ActivityThread 你都没听说过？那你肯定听说过传说中的 UI 线程吧？是的，这就是 UI 线程。我们前面说过，App 和 AMS 是通过 Binder 传递信息的，那么 ActivityThread 就是专门与 AMS 的外交工作的。

AMS 说：“ActivityThread，你给我暂停一个 Activity！”

ActivityThread 就说：“没问题！”然后转身和 Instrumentation 说：“老婆，AMS 让暂停一个 Activity，我这里忙着呢，你快去帮我把这事办了把~”于是，Instrumentation 就去把事儿搞定了。

所以说，AMS 是董事会，负责指挥和调度的，ActivityThread 是老板，虽然说家里的事自己说了算，但是需要听从 AMS 的指挥，而 Instrumentation 则是老板娘，负责家里的大事小事，但是一般不抛头露面，听一家之主 ActivityThread 的安排。

如何理解 AMS 和 ActivityThread 之间的 Binder 通信？

前面我们说到，在调用 startActivity() 的时候，实际上调用的是

`mInstrumentation.execStartActivity()`

但是到这里还没完呢！里面又调用了下面的方法

```
ActivityManagerNative.getDefault()
```

```
.startActivity
```

这里的 ActivityManagerNative.getDefault 返回的就是

ActivityManagerService 的远程接口，即 ActivityManagerProxy。

怎么知道的呢？往下看

```
public abstract class ActivityManagerNative extends Binder  
implements IActivityManager  
{
```

//从类声明上，我们可以看到 ActivityManagerNative 是 Binder 的一个子类，而且实现了 IActivityManager 接口

```
static public IActivityManager getDefault() {  
  
    return gDefault.get();  
  
}
```

//通过单例模式获取一个 IActivityManager 对象，这个对象通过
asInterface(b)获得

```
private static final Singleton<IActivityManager> gDefault = new  
Singleton<IActivityManager>() {  
  
    protected IActivityManager create() {  
  
        IBinder b = ServiceManager.getService("activity");  
  
        if (false) {  
  
            Log.v("ActivityManager", "default service binder = "  
+ b);  
  
        }  
  
        IActivityManager am = asInterface(b);  
  
        if (false) {  
  
            Log.v("ActivityManager", "default service = " + am);  
  
        }  
  
        return am;  
  
    }  
  
};
```

```
}
```

```
//最终返回的还是一个 ActivityManagerProxy 对象
```

```
static public IActivityManager asInterface(IBinder obj) {
```

```
    if (obj == null) {
```

```
        return null;
```

```
}
```

```
IActivityManager in =
```

```
(IActivityManager)obj.queryLocalInterface(descriptor);
```

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

```
        return in;
```

```
}
```

```
//这里面的 Binder 类型的 obj 参数会作为 ActivityManagerProxy 的  
成员变量保存为 mRemote 成员变量，负责进行 IPC 通信
```

```
        return new ActivityManagerProxy(obj);

    }

}
```

再看 `ActivityManagerProxy.startActivity()`，在这里面做的事情就是 IPC 通信，利用 Binder 对象，调用 `transact()`，把所有需要的参数封装成 Parcel 对象，向 AMS 发送数据进行通信。

```
public int startActivity(IApplicationThread caller, String
callingPackage, Intent intent,
String resolvedType, IBinder resultTo, String resultWho,
int requestCode,
int startFlags, ProfilerInfo profilerInfo, Bundle options)
throws RemoteException {

    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IActivityManager.descriptor);
```

```
    data.writeStrongBinder(caller != null ? caller.asBinder() :  
        null);  
  
    data.writeString(callingPackage);  
  
    intent.writeToParcel(data, 0);  
  
    data.writeString(resolvedType);  
  
    data.writeStrongBinder(resultTo);  
  
    data.writeString(resultWho);  
  
    data.writeInt(requestCode);  
  
    data.writeInt(startFlags);  
  
    if (profilerInfo != null) {  
  
        data.writeInt(1);  
  
        profilerInfo.writeToParcel(data,  
            Parcelable.PARCELABLE_WRITE_RETURN_VALUE);  
  
    } else {  
  
        data.writeInt(0);  
  
    }  
  
    if (options != null) {
```

```
        data.writeInt(1);

        options.writeToParcel(data, 0);

    } else {

        data.writeInt(0);

    }

    mRemote.transact(START_ACTIVITY_TRANSACTION, data,
reply, 0);

    reply.readException();

    int result = reply.readInt();

    reply.recycle();

    data.recycle();

    return result;

}
```

Binder 本质上只是一种底层通信方式，和具体服务没有关系。为了提供具体服务，Server 必须提供一套接口函数以便 Client 通过远程访问使用各种服务。这时通常采用 Proxy 设计模式：将接口函数定义在一个抽象类中，Server 和 Client

都会以该抽象类为基类实现所有接口函数，所不同的是 Server 端是真正的功能实现，而 Client 端是对这些函数远程调用请求的包装。

为了更方便的说明客户端和服务器之间的 Binder 通信，下面以 ActivityManagerServices 和他在客户端的代理类 ActivityManagerProxy 为例。

ActivityManagerServices 和 ActivityManagerProxy 都实现了同一个接口——IActivityManager。

```
class ActivityManagerProxy implements IActivityManager{  
  
    public final class ActivityManagerService extends  
        ActivityManagerNative{}  
  
    public abstract class ActivityManagerNative extends Binder  
        implements IActivityManager{  
  
    虽然都实现了同一个接口，但是代理对象 ActivityManagerProxy 并不会对这些  
    方法进行真正地实现，ActivityManagerProxy 只是通过这种方式对方法的参数  
    进行打包(因为都实现了相同接口，所以可以保证同一个方法有相同的参数，即  
    对要传输给服务器的数据进行打包)，真正实现的是 ActivityManagerService。
```

但是这个地方并不是直接由客户端传递给服务器，而是通过 Binder 驱动进行中转。其实我对 Binder 驱动并不熟悉，我们就把他当做一个中转站就 OK，客户端调用 ActivityManagerProxy 接口里面的方法，把数据传送给 Binder 驱动，然后 Binder 驱动就会把这些东西转发给服务器的 ActivityManagerServices，由 ActivityManagerServices 去真正的实施具体的操作。

但是 Binder 只能传递数据，并不知道是要调用 ActivityManagerServices 的哪个方法，所以在数据中会添加方法的唯一标识码，比如前面的 startActivity() 方法：

```
public int startActivity(IApplicationThread caller, String
callingPackage, Intent intent,
String resolvedType, IBinder resultTo, String resultWho,
int requestCode,
int startFlags, ProfilerInfo profilerInfo, Bundle options)
throws RemoteException {

    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();

    ...
    ...ignore some code...
}
```

```
mRemote.transact(START_ACTIVITY_TRANSACTION, data,  
    reply, 0);  
  
    reply.readException();  
  
    int result = reply.readInt();  
  
    reply.recycle();  
  
    data.recycle();  
  
    return result;  
}
```

上面的 START_ACTIVITY_TRANSACTION 就是方法标示 , data 是要传输给
Binder 驱动的数据 , reply 则接受操作的返回值。

即

客户端 : ActivityManagerProxy =====> Binder 驱动=====>
ActivityManagerService : 服务器

而且由于继承了同样的公共接口类 , ActivityManagerProxy 提供了与
ActivityManagerService 一样的函数原型 ,使用户感觉不出 Server 是运行在本
地还是远端 ,从而可以更加方便的调用这些重要的系统服务。

但是！这里 Binder 通信是单方向的，即从 ActivityManagerProxy 指向 ActivityManagerService 的。如果 AMS 想要通知 ActivityThread 做一些事情，应该咋办呢？

还是通过 Binder 通信，不过是换了另外一对，换成了 ApplicationThread 和 ApplicationThreadProxy。

客户端：ApplicationThread <=====Binder 驱动<=====

ApplicationThreadProxy：服务器

他们也都实现了相同的接口 IApplicationThread

```
private class ApplicationThread extends ApplicationThreadNative
{}
```

```
public abstract class ApplicationThreadNative extends Binder
implements IApplicationThread{}
```

```
class ApplicationThreadProxy implements IApplicationThread {}
```

剩下的就不必多说了吧，和前面一样。

AMS 接收到客户端的请求之后，会如何开启一个 Activity？

OK，至此，点击桌面图标调用 `startActivity()`，终于把数据和要开启 Activity 的请求发送到了 AMS 了。说了这么多，其实这些都在一瞬间完成了，下面咱们研究下 AMS 到底做了什么。

注：前方有高能的方法调用链，如果你现在累了，请先喝杯咖啡或者是上趟厕所休息下

AMS 收到 `startActivity` 的请求之后，会按照如下的方法链进行调用

调用 `startActivity()`

```
@Override  
  
    public final int startActivity(IApplicationThread caller, String  
callingPackage,  
  
        Intent intent, String resolvedType, IBinder resultTo,  
String resultWho, int requestCode,  
  
        int startFlags, ProfilerInfo profilerInfo, Bundle options) {  
  
        return startActivityAsUser(caller, callingPackage, intent,  
resolvedType, resultTo,  
  
        resultWho, requestCode, startFlags, profilerInfo,  
options,  
  
        UserHandle.getCallingUserId());
```

```
}
```

调用 startActivityAsUser()

```
@Override
```

```
    public final int startActivityAsUser(IAplicationThread caller,  
                                         String callingPackage,  
                                         Intent intent, String resolvedType, IBinder resultTo,  
                                         String resultWho, int requestCode,  
                                         int startFlags, ProfilerInfo profilerInfo, Bundle options,  
                                         int userId) {
```

...ignore some code...

```
        return mStackSupervisor.startActivityMayWait(caller, -1,  
                                                     callingPackage, intent,  
                                                     resolvedType, null, null, resultTo, resultWho,  
                                                     requestCode, startFlags,  
                                                     profilerInfo, null, null, options, userId, null, null);
```

```
}
```

在这里又出现了一个新对象 ActivityStackSupervisor，通过这个类可以实现对 ActivityStack 的部分操作。

```
final int startActivityMayWait(IApplicationThread caller, int
callingUid,
String callingPackage, Intent intent, String
resolvedType,
IVoiceInteractionSession voiceSession, IVoiceInteractor
voiceInteractor,
IBinder resultTo, String resultWho, int requestCode, int
startFlags,
ProfilerInfo profilerInfo, WaitResult outResult,
Configuration config,
Bundle options, int userId, IActivityContainer iContainer,
TaskRecord inTask) {
...ignore some code...
```

```
    int res = startActivityLocked(caller, intent,  
        resolvedType, aInfo,  
  
        voiceSession, voiceInteractor, resultTo,  
        resultWho,  
  
        requestCode, callingPid, callingUid,  
        callingPackage,  
  
        realCallingPid, realCallingUid, startFlags,  
        options,  
  
        componentSpecified, null, container, inTask);  
  
    ...ignore some code...  
}
```

继续调用 startActivityLocked()

```
final int startActivityLocked(IApplicationThread caller,  
    Intent intent, String resolvedType, ActivityInfo aInfo,
```

```
    IVoiceInteractionSession voiceSession, IVoiceInteractor  
    voiceInteractor,  
  
    IBinder resultTo, String resultWho, int requestCode,  
  
    int callingPid, int callingUid, String callingPackage,  
  
    int realCallingPid, int realCallingUid, int startFlags,  
  
    Bundle options,  
  
    boolean componentSpecified, ActivityRecord[]  
    outActivity, ActivityContainer container,  
  
    TaskRecord inTask) {  
  
    err = startActivityUncheckedLocked(r, sourceRecord,  
    voiceSession, voiceInteractor,  
  
    startFlags, true, options, inTask);  
  
    if (err < 0) {  
  
        notifyActivityDrawnForKeyguard();  
  
    }  
  
    return err;
```

```
}
```

调用 `startActivityUncheckedLocked()`, 此时要启动的 Activity 已经通过检验 ,
被认为是一个正当的启动请求。

终于 ,在这里调用到了 `ActivityStack` 的 `startActivityLocked(ActivityRecord r,`
`boolean newTask, boolean doResume, boolean keepCurTransition,`
`Bundle options)`。

`ActivityRecord` 代表的就是要开启的 Activity 对象 , 里面分装了很多信息 , 比
如所在的 `ActivityTask` 等 , 如果这是首次打开应用 , 那么这个 Activity 会被放
到 `ActivityTask` 的栈顶 ,

```
final int startActivityUncheckedLocked(ActivityRecord r,  
ActivityRecord sourceRecord,  
IVoiceInteractionSession voiceSession, IVoiceInteractor  
voiceInteractor, int startFlags,  
boolean doResume, Bundle options, TaskRecord inTask)  
{  
    ...ignore some code...
```

```
targetStack.startActivityLocked(r, newTask, doResume,  
keepCurTransition, options);
```

...ignore some code...

```
return ActivityManager.START_SUCCESS;
```

```
}
```

调用的是 ActivityStack.startActivityLocked()

```
final void startActivityLocked(ActivityRecord r, boolean newTask,  
boolean doResume, boolean keepCurTransition, Bundle  
options) {
```

//ActivityRecord 中存储的 TaskRecord 信息

```
TaskRecord rTask = r.task;
```

...ignore some code...

//如果不是在新的 ActivityTask(也就是 TaskRecord)中的话，就
找出要运行在的 TaskRecord 对象

```
TaskRecord task = null;

if (!newTask) {

    boolean startIt = true;

    for (int taskNdx = mTaskHistory.size() - 1; taskNdx >= 0;
--taskNdx) {

        task = mTaskHistory.get(taskNdx);

        if (task.getTopActivity() == null) {

            // task 中的所有 Activity 都结束了

            continue;

        }

        if (task == r.task) {

            // 找到了

            if (!startIt) {

                task.addActivityToTop(r);

                r.putInHistory();

            }

        }

    }

}

if (task == null) {

    return null;

}

return task;
```

```
mWindowManager.addAppToken(task.mActivities.indexOf(r),  
r.appToken,  
r.task.taskId, mStackId,  
r.info.screenOrientation, r.fullscreen,  
(r.info.flags &  
ActivityInfo.FLAG_SHOW_ON_LOCK_SCREEN) != 0,  
r.userId, r.info.configChanges,  
task.voiceSession != null,  
r.mLaunchTaskBehind);  
  
if (VALIDATE_TOKENS) {  
    validateAppTokensLocked();  
}  
  
ActivityOptions.abort(options);  
  
return;  
}  
  
break;
```

```
    } else if (task.numFullscreen > 0) {  
  
        startIt = false;  
  
    }  
  
}  
  
}
```

...ignore some code...

```
// Place a new activity at top of stack, so it is next to interact  
// with the user.
```

```
task = r.task;  
  
task.addActivityToTop(r);  
  
task.setFrontOfTask();
```

...ignore some code...

```
    if (doResume) {  
  
        mStackSupervisor.resumeTopActivitiesLocked(this, r,  
options);  
  
    }  
  
}
```

咱们一起看下 StackSupervisor.resumeTopActivitiesLocked(this, r, options)

```
boolean resumeTopActivitiesLocked(ActivityStack targetStack,  
ActivityRecord target,
```

```
    Bundle targetOptions) {
```

```
        if (targetStack == null) {  
  
            targetStack = getFocusedStack();  
  
        }  
  
    }
```

```
// Do targetStack first.
```

```
        boolean result = false;
```

```
        if (isFrontStack(targetStack)) {
```

```
    result = targetStack.resumeTopActivityLocked(target,  
targetOptions);  
  
}
```

...ignore some code...

```
return result;
```

```
}
```

咱们坚持住，看一下 `ActivityStack.resumeTopActivityInnerLocked()` 到底进行了什么操作

```
final boolean resumeTopActivityInnerLocked(ActivityRecord prev,  
Bundle options) {
```

...ignore some code...

```
//找出还没结束的首个 ActivityRecord
```

```
ActivityRecord next = topRunningActivityLocked(null);
```

```
//如果一个没结束的 Activity 都没有，就开启 Launcher 程序

if (next == null) {

    ActivityOptions.abort(options);

    if (DEBUG_STATES) Slog.d(TAG,
"resumeTopActivityLocked: No more activities go home");

    if (DEBUG_STACK)

mStackSupervisor.validateTopActivitiesLocked();

    // Only resume home if on home display

    final int returnTaskType = prevTask == null

|| !prevTask.isOverHomeStack() ?

        HOME_ACTIVITY_TYPE :

prevTask.getTaskToReturnTo();

    return isOnHomeDisplay() &&

mStackSupervisor.resumeHomeStackTask(returnTaskType, prev);

}
```

//先需要暂停当前的 Activity。因为我们是在 Launcher 中启动 mainActivity，所以当前 mResumedActivity != null，调用 startPausingLocked()使得 Launcher 进入 Pausing 状态

```
if (mResumedActivity != null) {  
  
    pausing |= startPausingLocked(userLeaving, false, true,  
        dontWaitForPause);  
  
    if (DEBUG_STATES) Slog.d(TAG,  
        "resumeTopActivityLocked: Pausing " + mResumedActivity);  
  
}  
  
}
```

在这个方法里，prev.app 为记录启动 Launcher 进程的 ProcessRecord， prev.app.thread 为 Launcher 进程的远程调用接口 IApplicationThread，所以可以调用 prev.app.thread.schedulePauseActivity，到 Launcher 进程暂停指定 Activity。

```
final boolean startPausingLocked(boolean userLeaving, boolean  
    uiSleeping, boolean resuming,  
    boolean dontWait) {
```

```
    if (mPausingActivity != null) {

        completePauseLocked(false);

    }

    ...ignore some code...

    if (prev.app != null && prev.app.thread != null)

        try {

            mService.updateUsageStats(prev, false);

        }

        prev.app.thread.schedulePauseActivity(prev.appToken, prev.finishing,
                                              userLeaving, prev.configChangeFlags,
                                              dontWait);

    } catch (Exception e) {

        mPausingActivity = null;

        mLastPausedActivity = null;

        mLastNoHistoryActivity = null;

    }

}
```

```
    }

} else {

    mPausingActivity = null;

    mLastPausedActivity = null;

    mLastNoHistoryActivity = null;

}

...ignore some code...
```

在 Launcher 进程中消息传递，调用 ActivityThread.handlePauseActivity()，最终调用 ActivityThread.performPauseActivity() 暂停指定 Activity。接着通过前面所说的 Binder 通信，通知 AMS 已经完成暂停的操作。

ActivityManagerNative.getDefault().activityPaused(token)。

上面这些调用过程非常复杂，源码中各种条件判断让人眼花缭乱，所以说如果你没记住也没关系，你只要记住这个流程，理解了 Android 在控制 Activity

生命周期时是如何操作，以及是通过哪几个关键的类进行操作的就可以了，以后遇到相关的问题之道从哪块下手即可

三、Fragment

1、Fragment 生命周期和 Activity 对比

2、Fragment 之间如何进行通信

3、Fragment 的 startActivityForResult

4、Fragment 重叠问题

文章、Android Fragment 完全解析，关于碎片你所需知道的一切

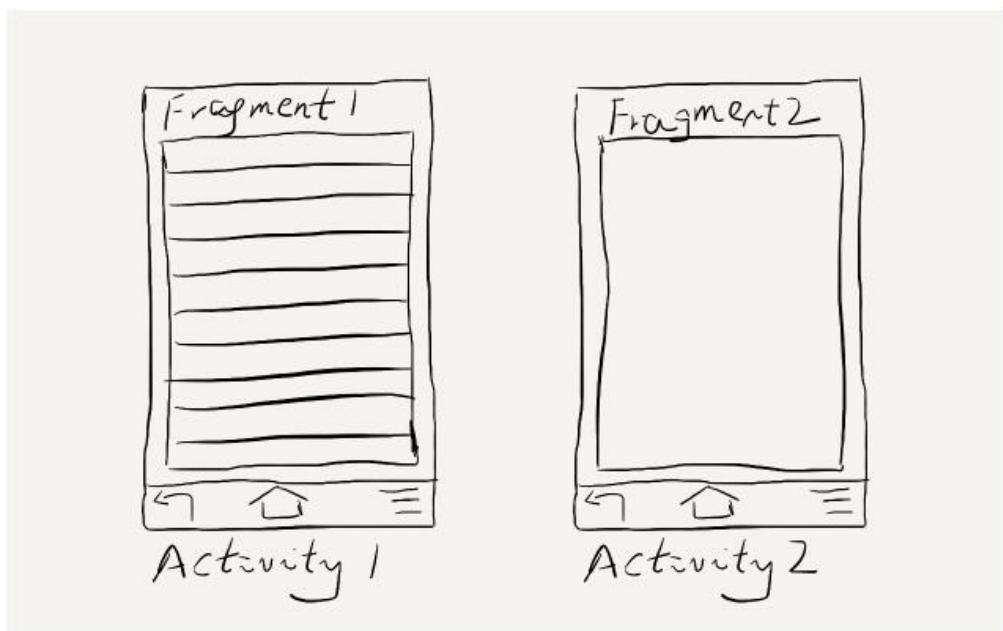
我们都知道，Android 上的界面展示都是通过 Activity 实现的，Activity 实在是太常用了，我相信大家都已经非常熟悉了，这里就不再赘述。

但是 Activity 也有它的局限性，同样的界面在手机上显示可能很好看，在平板上就未必了，因为平板的屏幕非常大，手机的界面放在平板上可能会有过分被拉长、控件间距过大等情况。这个时候更好的体验效果是在 Activity 中嵌入"小 Activity"，然后每个"小 Activity"又可以拥有自己的布局。因此，我们今天的主角 Fragment 登场了。

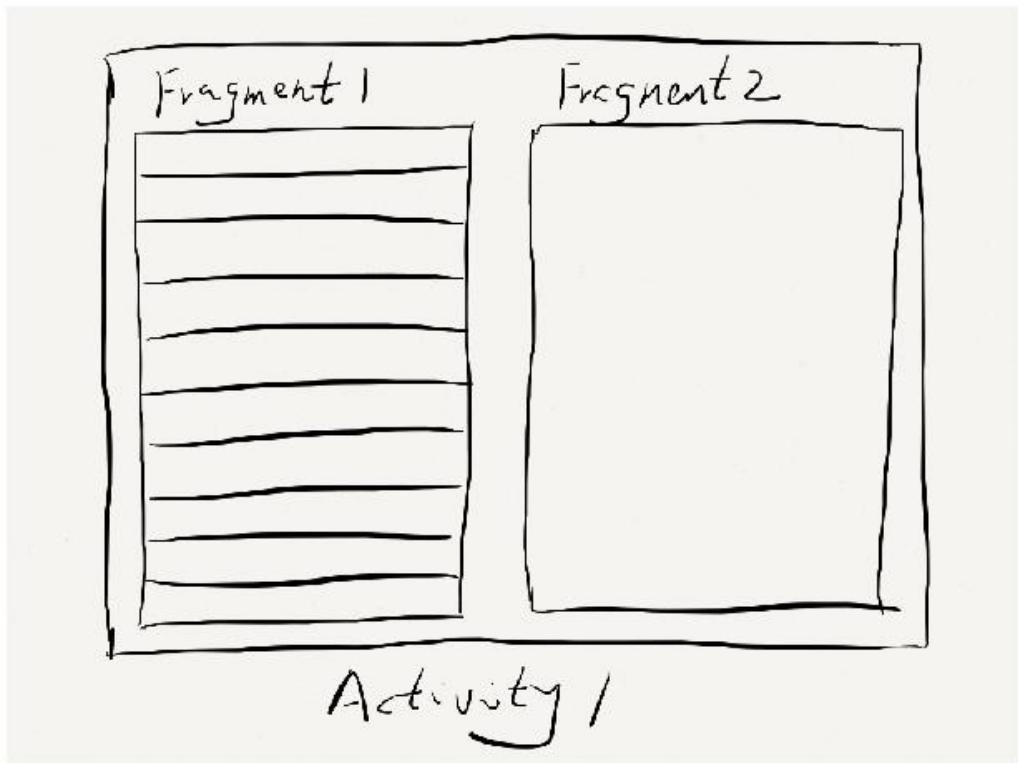
Fragment 初探

为了让界面可以在平板上更好地展示，Android 在 3.0 版本引入了 Fragment(碎片)功能，它非常类似于 Activity，可以像 Activity 一样包含布局。Fragment 通常是嵌套在 Activity 中使用的，现在想象这种场景：有两个 Fragment，Fragment 1 包含了一个 ListView，每行显示一本书的标题。Fragment 2 包含了 TextView 和 ImageView，来显示书的详细内容和图片。

如果现在程序运行竖屏模式的平板或手机上，Fragment 1 可能嵌入在一个 Activity 中，而 Fragment 2 可能嵌入在另一个 Activity 中，如下图所示：



而如果现在程序运行在横屏模式的平板上，两个 Fragment 就可以嵌入在同一个 Activity 中了，如下图所示：



由此可以看出，使用 Fragment 可以让我们更加充分地利用平板的屏幕空间，下面我们一起来探究下如何使用 Fragment。

首先需要注意，Fragment 是在 3.0 版本引入的，如果你使用的是 3.0 之前 的系统，需要先导入 android-support-v4 的 jar 包才能使用 Fragment 功能。

新建一个项目叫做 Fragments，然后在 layout 文件夹下新建一个名为 fragment1.xml 的布局文件：

```
1 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2     android:layout_width="match_parent"
3     android:layout_height="match_parent"
4     android:background="#00ff00" >
5
6     <TextView
7         android:layout_width="wrap_content"
8         android:layout_height="wrap_content"
9         android:text="This is fragment 1"
10        android:textColor="#000000"
11        android:textSize="25sp" />
12
13 </LinearLayout>
```

复制

可以看到，这个布局文件非常简单，只有一个 LinearLayout，里面加入了一个 TextView。我们如法炮制再新建一个 fragment2.xml：

```
1 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2     android:layout_width="match_parent"
3     android:layout_height="match_parent"
4     android:background="#ffff00" >
5
6     <TextView
7         android:layout_width="wrap_content"
8         android:layout_height="wrap_content"
9         android:text="This is fragment 2"
10        android:textColor="#000000"
11        android:textSize="25sp" />
12
13 </LinearLayout>
```

复制

然后新建一个类 Fragment1，这个类是继承自 Fragment 的：

```
public class Fragment1 extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
    container, Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment1, container, false);
    }
}
```

```
}
```

然后新建一个类 Fragment1，这个类是继承自 Fragment 的：

```
public class Fragment1 extends Fragment {  
  
    @Override  
  
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {  
  
        return inflater.inflate(R.layout.fragment1, container, false);  
  
    }  
  
}
```

我们可以看到，这个类也非常简单，主要就是加载了我们刚刚写好的 fragment1.xml 布局文件并返回。同样的方法，我们再写好 Fragment2：

```
public class Fragment2 extends Fragment {  
  
    @Override  
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {  
        return inflater.inflate(R.layout.fragment2, container, false);  
    }  
  
}
```

然后打开或新建 activity_main.xml 作为主 Activity 的布局文件，在里面加入两个 Fragment 的引用，使用 android:name 前缀来引用具体的 Fragment：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:baselineAligned="false" >  
  
    <fragment  
        android:id="@+id/fragment1"  
        android:name="com.example.fragmentdemo.Fragment1"  
        android:layout_width="0dp"
```

```
    android:layout_height="match_parent"
    android:layout_weight="1" />

<fragment
    android:id="@+id/fragment2"
    android:name="com.example.fragmentdemo.Fragment2"
    android:layout_width="0dip"
    android:layout_height="match_parent"
    android:layout_weight="1" />
```

```
</LinearLayout>
```

最后打开或新建 `MainActivity` 作为程序的主 `Activity`，里面的代码非常简单，都是自动生成的

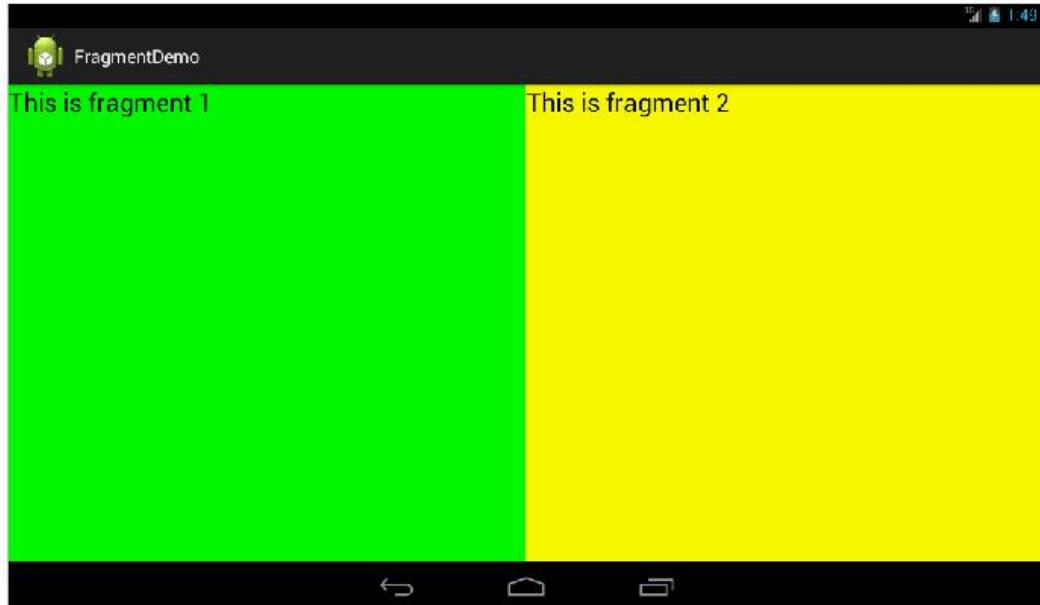
```
public class MainActivity extends Activity {
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

}
```

现在我们来运行一次程序，就会看到，一个 `Activity` 很融洽地包含了两个

`Fragment`，这两个 `Fragment` 平分了整个屏幕，效果图如下：



动态添加 Fragment

你已经学会了如何在 XML 中使用 Fragment , 但是这仅仅是 Fragment 最简单的功能而已。Fragment 真正的强大之处在于可以动态地添加到 Activity 当中 , 因此这也是你必须要掌握的东西。当你学会了在程序运行时向 Activity 添加 Fragment , 程序的界面就可以定制的更加多样化。下面我们立刻来看看 , 如何动态添加 Fragment。

还是在上一节代码的基础上修改 , 打开 activity_main.xml , 将其中对 Fragment 的引用都删除 , 只保留最外层的 LinearLayout , 并给它添加一个 id , 因为我们 要动态添加 Fragment , 不用在 XML 里添加了 , 删除后代码如下 :

```
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
  
    android:id="@+id/main_layout"  
  
    android:layout_width="match_parent"  
  
    android:layout_height="match_parent"  
  
    android:baselineAligned="false" >  
  
</LinearLayout>
```

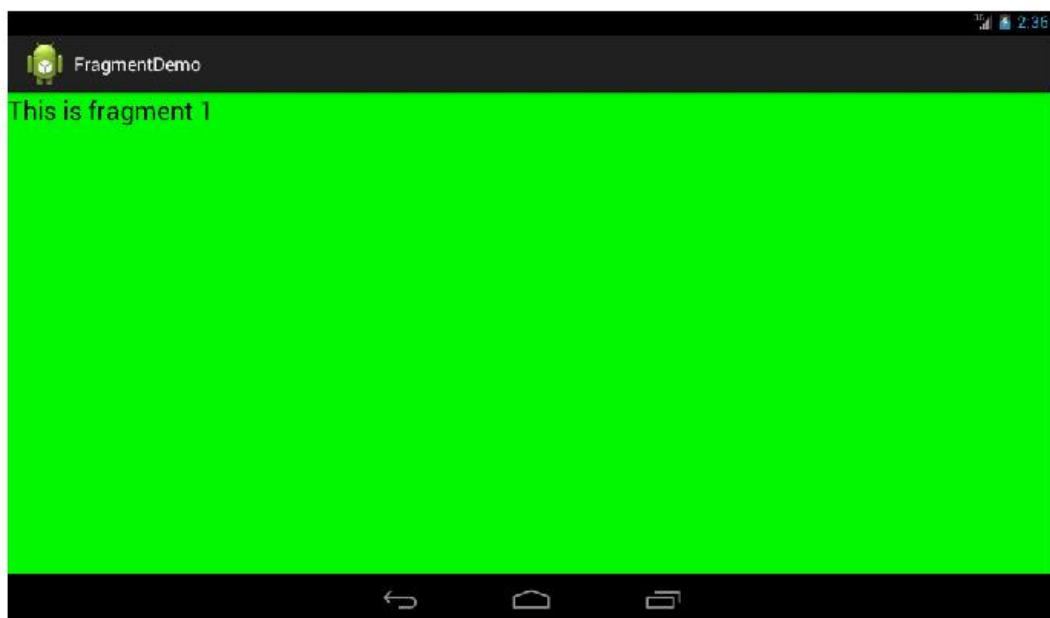
然后打开 MainActivity，修改其中的代码如下所示：

```
public class MainActivity extends Activity {  
  
    @Override  
  
    protected void onCreate(Bundle savedInstanceState) {  
  
        super.onCreate(savedInstanceState);  
  
        setContentView(R.layout.activity_main);  
  
        Display display = getWindowManager().getDefaultDisplay();  
  
        if (display.getWidth() > display.getHeight()) {  
  
            Fragment1 fragment1 = new Fragment1();  
  
            getFragmentManager().beginTransaction().replace(R.id.main_layout,  
fragment1).commit();  
  
        } else {  
  
            Fragment2 fragment2 = new Fragment2();  
  
            getFragmentManager().beginTransaction().replace(R.id.main_layout,  
fragment2).commit();  
  
        }  
  
    }  
  
}
```

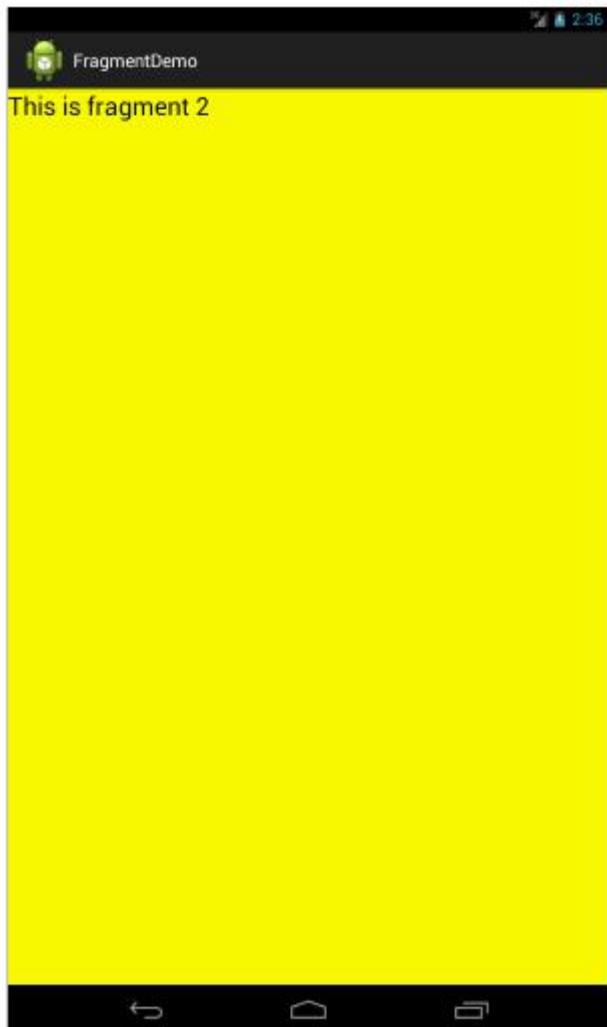
首先，我们要获取屏幕的宽度和高度，然后进行判断，如果屏幕宽度大于高度就添加 fragment1，如果高度大于宽度就添加 fragment2。动态添加 Fragment 主要分为 4 步：

1. 获取到 FragmentManager，在 Activity 中可以直接通过 getFragmentManager 得到。
2. 开启一个事务，通过调用 beginTransaction 方法开启。
3. 向容器内加入 Fragment，一般使用 replace 方法实现，需要传入容器的 id 和 Fragment 的实例。
4. 提交事务，调用 commit 方法提交。

现在运行一下程序，效果如下图所示：



如果你是在使用模拟器运行，按下 $ctrl + F11$ 切换到竖屏模式。效果如下图所示：



Fragment 的生命周期

和 Activity 一样，Fragment 也有自己的生命周期，理解 Fragment 的生命周期非常重要，我们通过代码的方式来瞧一瞧 Fragment 的生命周期是什么样的：

```
public class Fragment1 extends Fragment {  
  
    public static final String TAG = "Fragment1";
```

```
    @Override

    public View onCreateView(LayoutInflater inflater, ViewGroup
    container, Bundle savedInstanceState) {

        Log.d(TAG, "onCreateView");

        return inflater.inflate(R.layout.fragment1, container, false);

    }
```

```
    @Override

    public void onAttach(Activity activity) {

        super.onAttach(activity);

        Log.d(TAG, "onAttach");

    }
```

```
    @Override

    public void onCreate(Bundle savedInstanceState) {
```

```
super.onCreate(savedInstanceState);

Log.d(TAG, "onCreate");

}

@Override

public void onActivityCreated(Bundle savedInstanceState) {

    super.onActivityCreated(savedInstanceState);

    Log.d(TAG, "onActivityCreated");

}

@Override

public void onStart() {

    super.onStart();

    Log.d(TAG, "onStart");

}
```

```
@Override
```

```
public void onResume() {
```

```
    super.onResume();
```

```
    Log.d(TAG, "onResume");
```

```
}
```

```
@Override
```

```
public void onPause() {
```

```
    super.onPause();
```

```
    Log.d(TAG, "onPause");
```

```
}
```

```
@Override
```

```
public void onStop() {
```

```
    super.onStop();
```

```
    Log.d(TAG, "onStop");
```

```
}
```

```
@Override
```

```
public void onDestroyView() {
```

```
    super.onDestroyView();
```

```
    Log.d(TAG, "onDestroyView");
```

```
}
```

```
@Override
```

```
public void onDestroy() {
```

```
    super.onDestroy();
```

```
    Log.d(TAG, "onDestroy");
```

```
}
```

```
@Override
```

```
public void onDetach() {
```

```
super.onDetach();  
  
Log.d(TAG, "onDetach");  
  
}  
  
}
```

可以看到，上面的代码在每个生命周期的方法里都打印了日志，然后我们来运行一下程序，可以看到打印日志如下：

Application	Tag	Text
com.example.fragm...	Fragment1	onAttach
com.example.fragm...	Fragment1	onCreate
com.example.fragm...	Fragment1	onCreateView
com.example.fragm...	Fragment1	onActivityCreated
com.example.fragm...	Fragment1	onStart
com.example.fragm...	Fragment1	onResume

这时点击一下 home 键，打印日志如下：

Application	Tag	Text
com.example.fragment	Fragment1	onPause
com.example.fragment	Fragment1	onStop

如果你再重新进入进入程序，打印日志如下：

Application	Tag	Text
com.example.fragm...	Fragment1	onStart
com.example.fragm...	Fragment1	onResume

然后点击 back 键退出程序，打印日志如下：

Application	Tag	Text
com.example.fragm...	Fragment1	onPause
com.example.fragm...	Fragment1	onStop
com.example.fragm...	Fragment1	onDestroyView
com.example.fragm...	Fragment1	onDestroy
com.example.fragm...	Fragment1	onDetach

看到这里，我相信大多数朋友已经非常明白了，因为这和 Activity 的生命周期太相似了。只是有几个 Activity 中没有的新方法，这里需要重点介绍一下：

onAttach 方法：Fragment 和 Activity 建立关联的时候调用。

onCreateView 方法：为 Fragment 加载布局时调用。

onActivityCreated 方法 当 Activity 中的 onCreate 方法执行完后调用。

onDestroyView 方法：Fragment 中的布局被移除时调用。

onDetach 方法：Fragment 和 Activity 解除关联的时候调用。

Fragment 之间进行通信

通常情况下，Activity 都会包含多个 Fragment，这时多个 Fragment 之间如何进行通信就是个非常重要的问题了。我们通过一个例子来看一下，如何在一个 Fragment 中去访问另一个 Fragment 的视图。

还是在第一节代码的基础上修改，首先打开 fragment2.xml，在这个布局里面添加一个按钮：

```
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
  
    android:layout_width="match_parent"  
  
    android:layout_height="match_parent"  
  
    android:orientation="vertical"  
  
    android:background="#ffff00" >  
  
<TextView  
    android:layout_width="wrap_content"  
  
    android:layout_height="wrap_content"  
  
    android:text="This is fragment 2"
```

```
    android:textColor="#000000"

    android:textSize="25sp" />

<Button

    android:id="@+id/button"

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:text="Get fragment1 text"

/>

</LinearLayout>
```

然后打开 fragment1.xml，为 TextView 添加一个 id:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:background="#00ff00" >

    <TextView
```

```
    android:id="@+id/fragment1_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="This is fragment 1"
    android:textColor="#000000"
    android:textSize="25sp" />

```

</LinearLayout>

接着打开 Fragment2.java，添加 `onActivityCreated` 方法，并处理按钮的点击事件：

```
public class Fragment2 extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment2, container, false);
    }
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        Button button = (Button) getActivity().findViewById(R.id.button);
        button.setOnClickListener(new OnClickListener() {
```

```
    @Override

    public void onClick(View v) {

        TextView textView = (TextView)
getActivity().findViewById(R.id.fragment1_text);

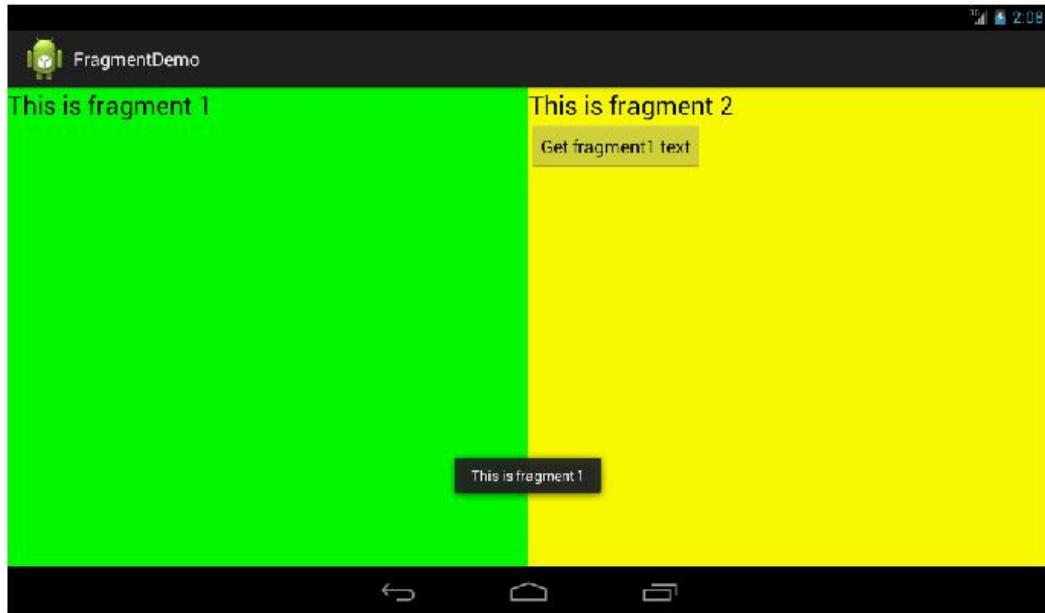
        Toast.makeText(getActivity(), textView.getText(),
Toast.LENGTH_LONG).show();

    }

});
```

}

现在运行一下程序，并点击一下 fragment2 上的按钮，效果如下图所示：



我们可以看到，在 fragment2 中成功获取到了 fragment1 中的视图，并弹出 Toast。这是怎么实现的呢？主要都是通过 getActivity 这个方法实现的。getActivity 方法可以让 Fragment 获取到关联的 Activity，然后再调用 Activity 的 findViewById 方法，就可以获取到和这个 Activity 关联的其它 Fragment 的视图了。

文章、Fragment 重叠，如何通信

一、概述

上一篇已经说明了 Fragment 的生命周期，以及基础的使用方法和一些 api 的作用。但是想要在项目中使用好 Fragment 必须能够清晰明白的管理好它的状态，以下会介绍实际开发会遇到的一些场景。

二、Fragment 回退栈管理

Activity 是由任务栈管理的，遵循先进后出的原则，Fragment 也可以实现类似的栈管理，从而实现多个 Fragment 先后添加后可以返回上一个 Fragment，当 activity 容器内没有 Fragment 时回退则退出 Activity。

具体方法：**FragmentTransaction.addToBackStack(String)** // 通常传入 null 即可

代码如下：

```
Fragment f = new Fragment();
```

```
FragmentManager fm = getSupportFragmentManager();
```

```
FragmentTransaction ftx = fm.beginTransaction();

ftx.replace(R.id.fragment_container, f, "ONE");

ftx.addToBackStack(null);

ftx.commit();
```

注：
1.activity的第一个Fragment(根Fragment)可以不添加回退栈，这样最后一个Fragment按返回时就不会空白而是直接退出activity。
2.调用addToBackStack(null)将当前的事务添加到了回退栈，调用replace方法后Fragment实例不会被销毁，但是视图层次会被销毁，即会调用onDestoryView和onCreateView。若需保存当前fragment视图状态，则可以使用hide后add新的Fragment

三、Fragment 与 Activity 通信

- a、如果 Activity 中包含自己管理的 Fragment 的引用，可以通过引用直接访问所有的 Fragment 的 public 方法
- b、如果 Activity 中未保存任何 Fragment 的引用，可以通过每个 Fragment 都有一个唯一的 TAG 或者 ID 使用 getFragmentManager.findFragmentByTag() 或者 findFragmentById() 获得任何 Fragment 实例，然后进行操作。
- c、在 Fragment 中可以通过 getActivity 得到当前绑定的 Activity 的实例，然后进行操作。

注：如果在Fragment中需要Context，可以通过调用getActivity()，如果该Context需要在Activity被销毁后还存在，则使用getActivity().getApplicationContext()。

推荐方式：

1. 接口 (Fragment 返回数据给 Activity)

Fragment 部分代码：

```
public class TestFragment extends Fragment {

    private OnSaveListener listener;

    public void setListener(OnSaveListener listener) {
        this.listener = listener;
    }

    public interface OnSaveListener {
        void onSaveFinished(boolean result);

        void onSaveStart();
    }

    @OnClick(R.id.btn_save)
    public void save() {
        ....
        listener.onSaveFinished(true);
    }
}
```

Activity 部分代码：

```
TestFragment f = new TestFragment();

f.setListener(new ShowCheckFragment.OnSaveListener() {
```

```
    @Override

    public void onSaveFinished(boolean result) {

        .....
    }
}
```

```
    @Override

    public void onSaveStart() {
```

.....

```
    }  
});  
  
FragmentTransaction fragmentTransaction =  
    getSupportFragmentManager().beginTransaction();  
  
fragmentTransaction.replace(R.id.fragment_container, f);  
  
fragmentTransaction.commit();
```

2.Fragment Arguments (传递数据到 Fragment 中)

Fragment 部分代码：

```
public class TestFragment extends Fragment  
{  
  
    private String mArgument;  
  
    public static final String ARGUMENT = "argument";  
  
    @Override  
    public void onCreate(Bundle savedInstanceState)  
    {  
        super.onCreate(savedInstanceState);  
  
        Bundle bundle = getArguments();
```

```
    if (bundle != null)

        mArgument = bundle.getString(ARGUMENT);

    }

/**

 * 传入需要的参数，设置给 arguments
 *
 * @param argument
 *
 * @return
 */

public static TestFragment newInstance(String argument)

{
    Bundle bundle = new Bundle();

    bundle.putString(ARGUMENT, argument);

    TestFragment f = new TestFragment();

    f.setArguments(bundle);

    return f;
}
```

Fragment 添加 newInstance 静态方法给实例化时调用，将需要的参数传入，设置到 bundle 中，然后 setArguments(bundle)，最后在 onCreate 中进行获取（Activity 之间的 Intent 调用也可以采取类似方式，详见郭霖大神的第一行代码 Activity 章节）；

注：

setArguments方法必须在fragment创建以后，添加给Activity前完成。千万不要先调用了add，然后设置arguments。

四、Fragment 重叠问题

当屏幕旋转或者内存重启 (Fragment 以及容器 activity 被系统回收后再打开时重新初始化)会导致 Fragment 重叠问题，是因为 activity 本身重启的时候会恢复 Fragment，然后创建 Fragment 的代码又会新建一个 Fragment 的原因。

解决方法：在 onCreate 方法中判断参数 Bundle savedInstanceState，为空时初始化 Fragment 实例，然后在 Fragment 中通过 onSaveInstanceState 的方法恢复数据

代码：

```
private TestFragment f;

protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.e(TAG, savedInstanceState+"");
```

```
if(savedInstanceState == null)

{

    f = new TestFragment();

    FragmentManager fm = getSupportFragmentManager();

    FragmentTransaction tx = fm.beginTransaction();

    tx.add(R.id.id_content, f, "ONE");

    tx.commit();

}

}
```

五、Fragment 与 ActionBar 和 MenuItem

Fragment 可以添加自己的 MenuItem 到 Activity 的 ActionBar 或者可选菜单中。

a、在 Fragment 的 onCreate 中调用 setHasOptionsMenu(true);

- b、然后在 Fragment 类中实现 onCreateOptionsMenu;
- c、如果希望在 Fragment 中处理 MenuItem 的点击，也可以实现 onOptionsItemSelected ; Activity 也可以直接处理该 MenuItem 的点击事件。

Fragment 部分代码:

```
public class TestFragment extends Fragment

{

    @Override

    public void onCreate(Bundle savedInstanceState)

    {

        super.onCreate(savedInstanceState);

        setHasOptionsMenu(true);

    }

}

....
```



```
    return true;  
}  
}
```

Activity 代码：

```
@Override  
  
public boolean onCreateOptionsMenu(Menu menu)  
  
{  
  
    super.onCreateOptionsMenu(menu);  
  
    getMenuInflater().inflate(R.menu.main, menu);  
  
    return true;  
  
}  
  
  
@Override  
  
public boolean onOptionsItemSelected(MenuItem item)
```

```
switch (item.getItemId())  
  
{  
  
    case R.id.action_settings:  
  
        .....  
  
        return true;  
  
    default:  
  
        //如果希望 Fragment 自己处理 MenuItem 点击事件 , 一定不要  
        //忘了调用 super.xxx  
  
        return super.onOptionsItemSelected(item);  
  
}  
  
}
```

注 : 如果要 Fragment 自己处理 MenuItem 点击事件 , 一定要调用 super.xxx

六、没有布局的 Fragment—保存大量数据

主要用于处理异步请求带来的数据保存问题 , 尤其是异步请求未完成时屏幕旋转这种现象。步骤如下 :

1、继承 Fragment , 声明引用指向你的有状态的对象

2、当 Fragment 创建时调用 setRetainInstance(boolean)

3、把 Fragment 实例添加到 Activity 中

4、当 Activity 重新启动后，使用 FragmentManager 对 Fragment 进行恢复

Fragment 部分代码：

```
public class TestFragment extends Fragment

{

    // data object we want to retain

    // 保存一个异步的任务

    private MyAsyncTask data;

    // this method is only called once for this fragment

    @Override

    public void onCreate(Bundle savedInstanceState)

    {

        super.onCreate(savedInstanceState);

        // retain this fragment
```

```
    setRetainInstance(true);

}

public void setData(MyAsyncTask data)

{
    this.data = data;

}

public MyAsyncTask getData()

{
    return data;

}

}
```

AsyncTask 部分代码 :

```
public class MyAsyncTask extends AsyncTask<Void, Void, Void>

{

    private FixProblemsActivity activity;

    /**
     * 是否完成
     */

    private boolean isCompleted;

    /**
     * 进度框
     */

    private LoadingDialog mLoadingDialog;

    private List<String> items;

    public MyAsyncTask(FixProblemsActivity activity)

    {

        this.activity = activity;
    }
}
```

```
    }

    /**
     * 开始时，显示加载框
     */
    @Override
    protected void onPreExecute()
    {
        // 使用 DialogFragment 创建对话框
        mLoadingDialog = new LoadingDialog();
        mLoadingDialog.show(activity.getFragmentManager(),
        "LOADING");
    }

    /**
     * 加载数据
    }
```

```
*/  
  
@Override  
  
protected Void doInBackground(Void... params)  
  
{  
  
    items = loadingData();  
  
    return null;  
  
}  
  
/**  
 * 加载完成回调当前的 Activity  
 */  
  
@Override  
  
protected void onPostExecute(Void unused)  
  
{  
  
    isCompleted = true;  
  
    notifyActivityTaskCompleted();
```

```
    if (mLoadingDialog != null)

        mLoadingDialog.dismiss();

    }

public List<String> getItems()

{

    return items;

}

private List<String> loadingData()

{

    try

    {

        Thread.sleep(5000);

    } catch (InterruptedException e)

    {


```

```
    }

    return new ArrayList<String>(Arrays.asList("通过 Fragment
保存大量数据",
"onSaveInstanceState 保存数据",
"getLastNonConfigurationInstance 已经被弃用",
"RabbitMQ", "Hadoop",
"Spark"));

}

/**
 * 设置 Activity ,因为 Activity 会一直变化 ,在 onDestroy 中 set null
 *
 * @param activity
 */
public void setActivity(FixProblemsActivity activity)
{
```

```
// 如果上一个 Activity 销毁，将与上一个 Activity 绑定的  
DialogFragment 销毁  
  
    if (activity == null)  
  
    {  
  
        mLoadingDialog.dismiss();  
  
    }  
  
// 设置为当前的 Activity  
  
this.activity = activity;  
  
// 开启一个与当前 Activity 绑定的等待框  
  
if (activity != null && !isCompleted)  
  
{  
  
    mLoadingDialog = new LoadingDialog();  
  
    mLoadingDialog.show(activity.getFragmentManager(),  
"LOADING");  
  
}  
  
// 如果完成，通知 Activity  
  
if (isCompleted)
```

```
{  
    notifyActivityTaskCompleted();  
  
}  
  
}  
  
private void notifyActivityTaskCompleted()  
{  
    if (null != activity)  
    {  
        activity.onTaskCompleted();  
  
    }  
  
}
```

Activity 部分代码：

```
public class FixProblemsActivity extends ListActivity
```

```
{\n\n    private static final String TAG = "MainActivity";\n\n    private ListAdapter mAdapter;\n\n    private List<String> mDatas;\n\n    private OtherRetainedFragment dataFragment;\n\n    private MyAsyncTask mMyTask;\n\n    @Override\n\n    public void onCreate(Bundle savedInstanceState)\n{\n    super.onCreate(savedInstanceState);\n\n    Log.e(TAG, "onCreate");\n\n    // find the retained fragment on activity restarts\n\n    FragmentManager fm = getFragmentManager();
```

```
    dataFragment = (OtherRetainedFragment)  
  
    fm.findFragmentByTag("data");  
  
    // create the fragment and data the first time  
  
    if (dataFragment == null)  
  
    {  
  
        // add the fragment  
  
        dataFragment = new OtherRetainedFragment();  
  
        fm.beginTransaction().add(dataFragment,  
"data").commit();  
  
    }  
  
    mMyTask = dataFragment.getData();  
  
    if (mMyTask != null)  
  
    {  
  
        mMyTask.setActivity(this);  
  
    } else  
  
    {
```

```
    mMyTask = new MyAsyncTask(this);

    dataFragment.setData(mMyTask);

    mMyTask.execute();

}

// the data is available in dataFragment.getData()

}

@Override

protected void onRestoreInstanceState(Bundle state)

{

    super.onRestoreInstanceState(state);

    Log.e(TAG, "onRestoreInstanceState");

}

}
```

```
    @Override  
  
    protected void onSaveInstanceState(Bundle outState)  
  
    {  
  
        mMyTask.setActivity(null);  
  
        super.onSaveInstanceState(outState);  
  
        Log.e(TAG, "onSaveInstanceState");  
  
    }  

```

```
    @Override  
  
    protected void onDestroy()  
  
    {  
  
        Log.e(TAG, "onDestroy");  
  
        super.onDestroy();  
  
    }  
  
    /**
```

```
* 回调  
*/  
  
public void onTaskCompleted()  
{  
  
    mDatas = mMyTask.getItems();  
  
    mAdapter = new  
        ArrayAdapter<String>(FixProblemsActivity.this,  
            android.R.layout.simple_list_item_1, mDatas);  
  
    setListAdapter(mAdapter);  
  
}  
  
}
```

七、DialogFragment 的使用

和 Fragment 有着基本一致的声明周期。且 DialogFragment 也允许开发者把 Dialog 作为内嵌的组件进行重用，类似 Fragment（可以在大屏幕和小屏幕显示出不同的效果）。使用 DialogFragment 至少需要实现 onCreateView 或者 onCreateDialog 方法。onCreateView 即使用定义的 xml 布局文件展示 Dialog。onCreateDialog 即利用 AlertDialog 或者 Dialog 创建出 Dialog。

1、重写 onCreateView 创建 Dialog

a.创建一个对话框布局文件

b.继承 DialogFragment，重写 onCreateView 方法:

```
public class TestFragment extends DialogFragment  
{  
  
    @Override  
  
    public View onCreateView(LayoutInflater inflater, ViewGroup  
    container,  
  
        Bundle savedInstanceState)  
  
    {  
  
        // 隐藏对话框标题栏  
  
        getDialog().requestWindowFeature(Window.FEATURE_NO_TITLE);  
  
        View view = inflater.inflate(R.layout.fragment_edit_name,  
        container);
```

```
    return view;  
  
}  
  
}
```

c.在 Activity 中调用：

```
public void showDialog(View view)  
  
{  
  
    TestDialogFragment dialog = new TestDialogFragment();  
  
    dialog.show(getFragmentManager(), "TestDialog");  
  
}
```

2、重写 onCreateDialog 创建 Dialog

a.新建对话框布局文件

b.继承 DialogFragment 重写 onCreateDialog 方法:

```
public class TestFragment extends DialogFragment
```

```
{
```

```
@Override
```

```
public Dialog onCreateDialog(Bundle savedInstanceState)  
  
{  
  
    AlertDialog.Builder builder = new  
    AlertDialog.Builder(getActivity());  
  
    // Get the layout inflater  
  
    LayoutInflater inflater = getActivity().getLayoutInflater();  
  
    View view = inflater.inflate(R.layout.fragment_test_dialog, null);  
  
    // Inflate and set the layout for the dialog  
  
    // Pass null as the parent view because its going in the dialog  
    layout  
  
    builder.setView(view)  
  
    // Add action buttons  
  
    .setPositiveButton("Test",  
        new DialogInterface.OnClickListener()  
  
    {  
  
        @Override
```

```
        public void onClick(DialogInterface dialog,
int id)

    {

}

}).setNegativeButton("Cancel", null);

return builder.create();

}

}
```

c.调用：

```
public void showDialog(View view)

{

TestFragment dialog = new TestFragment();

dialog.show(getFragmentManager(), "testDialog");

}
```

八、Fragment 的 startActivityForResult

在 Fragment 中存在 startActivityForResult() 以及 onActivityResult() 方法，需要通过调用 getActivity().setResult(、Fragment.REQUEST_CODE, intent)来设置返回。

部分代码：

```
// 传入数据
```

```
Intent intent = new Intent(getActivity(),ContentActivity.class);
```

```
intent.putExtra(ContentFragment.ARGUMENT, mTitles.get(position));
```

```
startActivityForResult(intent, REQUEST_DETAIL);
```

```
@Override
```

```
public void onActivityResult(int requestCode, int resultCode, Intent  
data)
```

```
{
```

```
Log.e("TAG", "onActivityResult");
```

```
super.onActivityResult(requestCode, resultCode, data);
```

```
if(requestCode == REQUEST_DETAIL)
```

```
{
```



```
{  
  
    mArgument = bundle.getString(ARGUMENT);  
  
    Intent intent = new Intent();  
  
    intent.putExtra(RESPONSE, "good");  
  
    getActivity().setResult(ListTitleFragment.REQUEST_DETAIL,  
    intent);  
  
}  
  
}
```

```
public static ContentFragment newInstance(String argument)  
  
{  
  
    Bundle bundle = new Bundle();  
  
    bundle.putString(ARGUMENT, argument);  
  
    ContentFragment contentFragment = new ContentFragment();  
  
    contentFragment.setArguments(bundle);  
  
}
```

```
    return contentFragment;  
  
}
```

九、FragmentPagerAdapter 与 FragmentStatePagerAdapter 的区别

使用 ViewPager 再结合上面任何一个实例的制作 APP 主页 ,主要区别就在与对于 fragment 是否销毁 :

FragmentPagerAdapter : 对于不再需要的 fragment , 选择调用 detach 方法 , 仅销毁视图 , 并不会销毁 fragment 实例。

FragmentStatePagerAdapter : 会销毁不再需要的 fragment , 当当前事务提交以后 , 会彻底的将 fragmeng 从当前 Activity 的 FragmentManager 中移除 , state 标明 , 销毁时 , 会将其 onSaveInstanceState(Bundle outState) 中的 bundle 信息保存下来 , 当用户切换回来 , 可以通过该 bundle 恢复生成新的 fragment , 也就是说 , 你可以在 onSaveInstanceState(Bundle outState) 方法中保存一些数据 , 在 onCreate 中进行恢复创建。

如上所说 , 使用 FragmentStatePagerAdapter 当然更省内存 , 但是销毁新建也是需要时间的。一般情况下 , 如果你是制作主页面 , 就 3、4 个 Tab , 那么可以选择使用 FragmentPagerAdapter , 如果你是用于 ViewPager 展示数量特别多的条目时 , 那么建议使用 FragmentStatePagerAdapter.

十、Fragment 间的数据传递

调用 Fragment.setTargetFragment , 这个方法一般用于当前 fragment 由其它 fragment 启动时。

部分代码：

```
EvaluateDialog dialog = new EvaluateDialog();

//注意 setTargetFragment

dialog.setTargetFragment(ContentFragment.this,
REQUEST_EVALUATE);

dialog.show(getFragmentManager(), EVALUATE_DIALOG);

//接收返回回来的数据

@Override

public void onActivityResult(int requestCode, int resultCode, Intent
data)

{

super.onActivityResult(requestCode, resultCode, data);

if (requestCode == REQUEST_EVALUATE)

{

String evaluate = data
```

```
        .getStringExtra(EvaluateDialog.RESPONSE_EVA  
LUATE);  
  
        Toast.makeText(getActivity(), evaluate,  
Toast.LENGTH_SHORT).show();  
  
        Intent intent = new Intent();  
  
        intent.putExtra(RESPONSE, evaluate);  
  
        getActivity().setResult(Activity.REQUEST_OK, intent);  
  
    }  
  
}
```

```
public class EvaluateDialog extends DialogFragment
```

```
{
```

```
.....
```

```
// 设置返回数据

protected void setResult(int which)

{

    // 判断是否设置了 targetFragment

    if (getTargetFragment() == null)

        return;

    Intent intent = new Intent();

    intent.putExtra(RESPONSE_EVALUATE,

mEvaluateVals[which]);

    getTargetFragment().onActivityResult(ContentFragment.REQUEST_EVALUATE, Activity.RESULT_OK, intent);

}

}
```

文章、Activity 与 Fragment 生命周期探讨

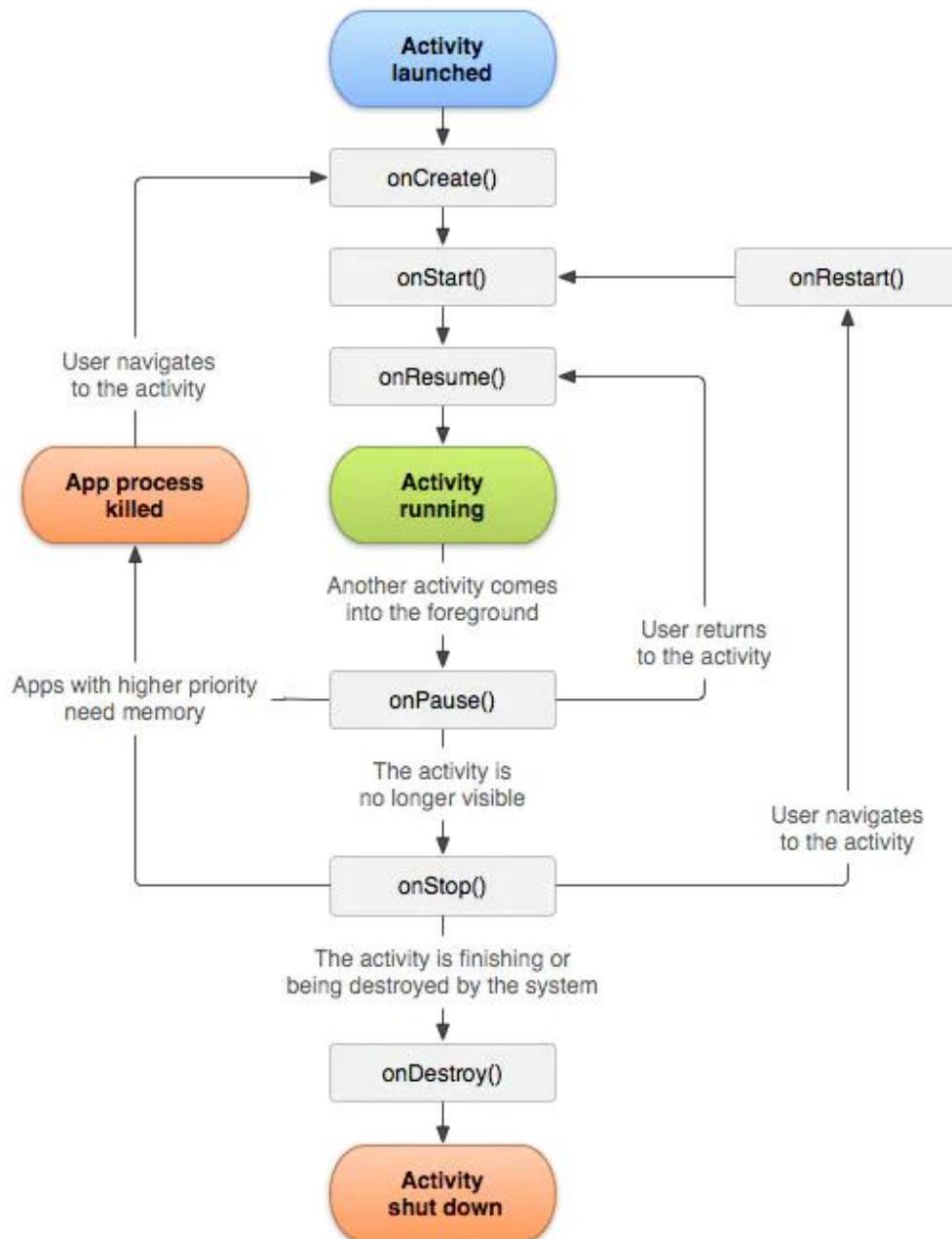


图1.Activity生命周期

其实这张图已经说明了 activity 的生命周期，但是在这里需要的注意的是，

(1) onCreate 是 activity 正在被创建，也就是说此时的 UI 操作不会更新 UI，比如 setText 操作，所以此时在子线程调用 setText 不会报线程错误。详解可见 Android 子线程更新 View 的探索，在这个方法内我们可以做一些初始化工作。

(2) onRestart 需要注意的是：activity 正在重新启动，一般情况下，activity 从不可见状态到可见状态，onRestart 才会被调用，但是一定要注意的一般来说这是用户行为导致 activity 不可见的时候，此时变为可见的时候才会调用 onRestart，

这里所说的用户行为就是用户按 home 键，或者进入“新”的 activity。这样的操作会使 activity 先执行 onPause, 后执行 onStop, 这样回到这个 activity 会调用 onRestart。为什么我这里强调说用户行为导致的不可见状态，等下我会说。。。

(3) onStart 的时候，activity 才可见，但是没有出现在前台，无法与用户交互

(4) onResume 的时候，activity 已经可见，并且出现在前台开始活动，与 onStart 相比，activity 都已经可见，但是 onStart 的时候 activity 还在后台，onResume 才显示在前台

(5) onPause 主要注意的是：此时的 activity 正在被停止，接下来马上调用 onStop。特殊情况下快速回到该 activity，onStop 不会执行，会去执行 onResume。

一般在这个生命周期内做存储数据、停止动画工作，但不能太耗时。

为什么特殊强调呢，因为该 activity 的 onPause 执行完了，才回去执行新的 activity 的 onResume，一旦耗时，必然会拖慢新的 activity 的显示。

(6) onStop：此时的 activity 即将停止。在这里可以做稍微重量级的操作，同样也不能耗时。

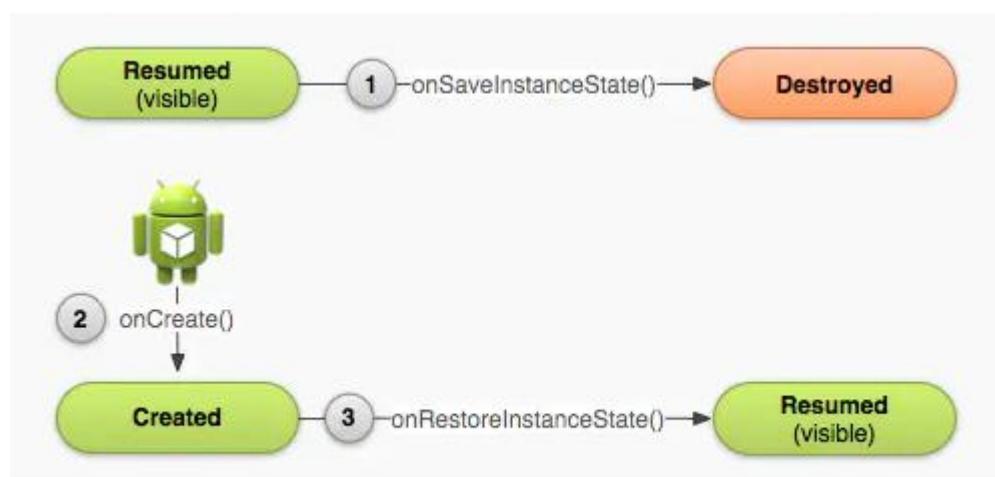
(7) onDestroy：此时的 activity 即将被回收，在这里会做一些回收工作和最终资源释放。

Activity 注意事项

Activity 中所有和状态相关的回调函数：

回调函数名称	使用场合
onCreate()	Activity实例被创建后第一次运行时
onNewIntent()	<p>有两种情况会执行该回调：</p> <ul style="list-style-type: none"> Intent的Flag中包含CLEAR_TOP，并且目标Activity已经存在当前任务队列中。 Intent的Flag中包括SINGLE_TOP，并且目标Activity已经存在当前任务队列中。 <p>前者和后者的区别在于，举例如下：当前任务队列为ABCD，此时目标Activity为B，那么前者会把队列改变为AB，而后者则会改变为ABCDB；但假设当前为ABCD，目标为D，前者则会改变为ABCD，后者则还是ABCD，而不会重新创建D对象，即SINGLE_TOP只对目标在top上时才有效</p>
onStart()	Activity从stop状态重新运行时
onRestoreInstanceState(Bundle savedInstanceState)	与onPostCreate()相同，只是先于onPostCreate()调用
onPostCreate()	如果Activity实例是第一次启动，则不调用，否则，以后的每次重新启动都会调用
onResume()	Activity继续运行时
onSave	与onPause()相同，只是会先于onPause()调用

在这里我会特别提出一个 point，就是异常情况下 activity 被杀死，而后被重新创建的情况。



异常情况下activity的重建过程

这张图非常重要，可以帮我们解决异常情况下 activity 如何正常回复的问题
当系统停止 activity 时，它会调用 onSaveInstanceState()（过程 1），如果 activity 被销毁了，但是需要创建同样的实例，系统会把过程 1 中的状态数据传给 onCreate() 和 onRestoreInstanceState()，所以我们要在 onSaveInstanceState() 内做保存参数的动作，在 onRestoreInstanceState() 做获取参数的动作。

```
Save Activity State

static final String STATE_SCORE = "playerScore";static final String STATE_LEVEL = "playerLevel";...

@Override

public void onSaveInstanceState(Bundle savedInstanceState) {

    // Save the user's current game state

    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);

    savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);

    // Always call the superclass so it can save the view hierarchy state

    super.onSaveInstanceState(savedInstanceState);}


```

获取参数操作：

```
onCreate() 方法

@Override protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState); // Always call the superclass first

    // Check whether we're recreating a previously destroyed instance

    if (savedInstanceState != null) {

        // Restore value of members from saved state

        mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
```

```
mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);

} else {

    // Probably initialize members with default values for a new instance

}

...}
```

也可以

`onRestoreInstanceState()`方法

```
public void onRestoreInstanceState(Bundle savedInstanceState) {

    // Always call the superclass so it can restore the view hierarchy
    super.onRestoreInstanceState(savedInstanceState);

    // Restore state members from saved instance

    mCurrentScore = savedInstanceState.getInt(STATE_SCORE);

    mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);}
```

Fragment 生命周期探讨

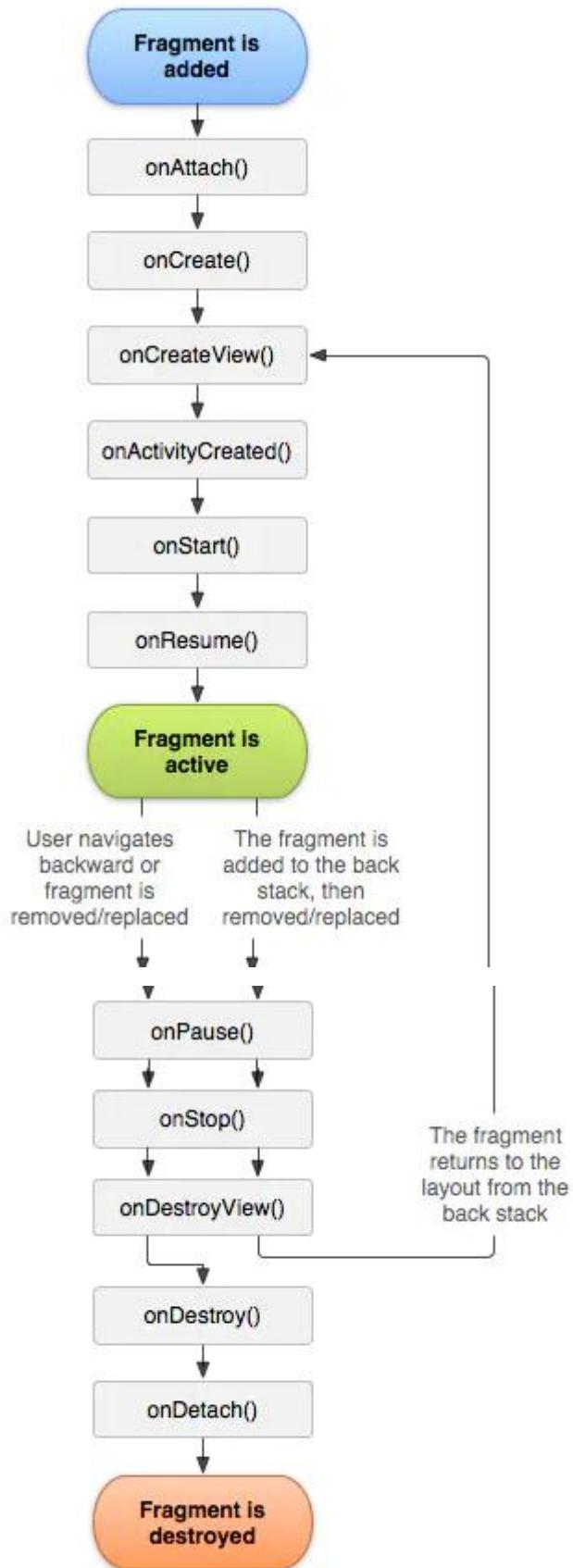


图2.Fragment生命周期

这张图很好的说明了 Fragment 与 Activity 生命周期的整合情况。通过对 Fragment 和 Activity 对比，将会发现许多不同之处，主要原因是因为 Activity 和 Fragment 之间需要交互。

在这里我首先需要提出的一些 points:

- Fragment 是直接从 Object 继承的，而 Activity 是 Context 的子类。因此我们可以得出结论：Fragment 不是 Activity 的扩展。但是与 Activity 一样，在我们使用 Fragment 的时候我们总会扩展 Fragment（或者是她的子类），并可以通过子类更改她的行为。
- 使用 Fragment 时，必要构建一个无参构造函数，系统会默认带。但一旦写有参构造函数，就必要构建无参构造函数。一般来说我们传参数给 Fragment，会通过 bundle，而不会用构造方法传，代码如下：

```
public static MyFragment newInstance(int index){  
  
    MyFragment mf = new MyFragment();  
  
    Bundle args = new Bundle();  
  
    args.putInt("index",index);  
  
    mf.setArguments(args);  
  
    return mf; }
```

下面来分析生命周期，然后根据这些我会强调一些 point。

(1) `onAttach`: `onAttach()`回调将在 Fragment 与其 Activity 关联之后调用。需要使用 Activity 的引用或者使用 Activity 作为其他操作的上下文，将在此回调方法中实现。

需要注意的是：将 Fragment 附加到 Activity 以后，就无法再次调用 `setArguments()`——除了在最开始，无法向初始化参数添加内容。

(2) `onCreate(Bundle savedInstanceState)`: 此时的 Fragment 的 `onCreate` 回调时，该 fragment 还没有获得 Activity 的 `onCreate()` 已完成的通知，所以不能将依赖于 Activity 视图层次结构存在性的代码放入此回调方法中。在 `onCreate()` 回调方法中，我们应该尽量避免耗时操作。此时的 `bundle` 就可以获取到 activity 传来的参数

```
@Override public void onCreate(Bundle savedInstanceState) {  
  
    super.onCreate(savedInstanceState);  
  
    Bundle args = getArguments();
```

```
if (args != null) {  
  
    mLabel = args.getCharSequence("label", mLabel);  
  
}  
  
}
```

(3) onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState): 其中的 Bundle 为状态包与上面的 bundle 不一样。

注意的是：不要将视图层次结构附加到传入的 ViewGroup 父元素中，该关联会自动完成。如果在此回调中将碎片的视图层次结构附加到父元素，很可能会出现异常。

这句话什么意思呢？就是不要把初始化的 view 视图主动添加到 container 里面，以为这会系统自带，所以 inflate 函数的第三个参数必须填 false，而且不能出现 container.addView(v) 的操作。

```
View v = inflater.inflate(R.layout.hello_world, container, false);
```

(4) onActivityCreated: onActivityCreated() 回调会在 Activity 完成其 onCreate() 回调之后调用。在调用 onActivityCreated() 之前，Activity 的视图层次结构已经准备好了，**这是在用户看到用户界面之前你可对用户界面执行的最后调整的地方。** 强调的 point: 如果 Activity 和她的 Fragment 是从保存的状态重新创建的，此回调尤其重要，也可以在这里确保此 Activity 的其他所有 Fragment 已经附加到该 Activity 中了

(5) Fragment 与 Activity 相同生命周期调用：接下来的 onStart()\onResume()\onPause()\onStop() 回调方法将和 Activity 的回调方法进行绑定，也就是说与 Activity 中对应的生命周期相同，因此不做过多介绍。

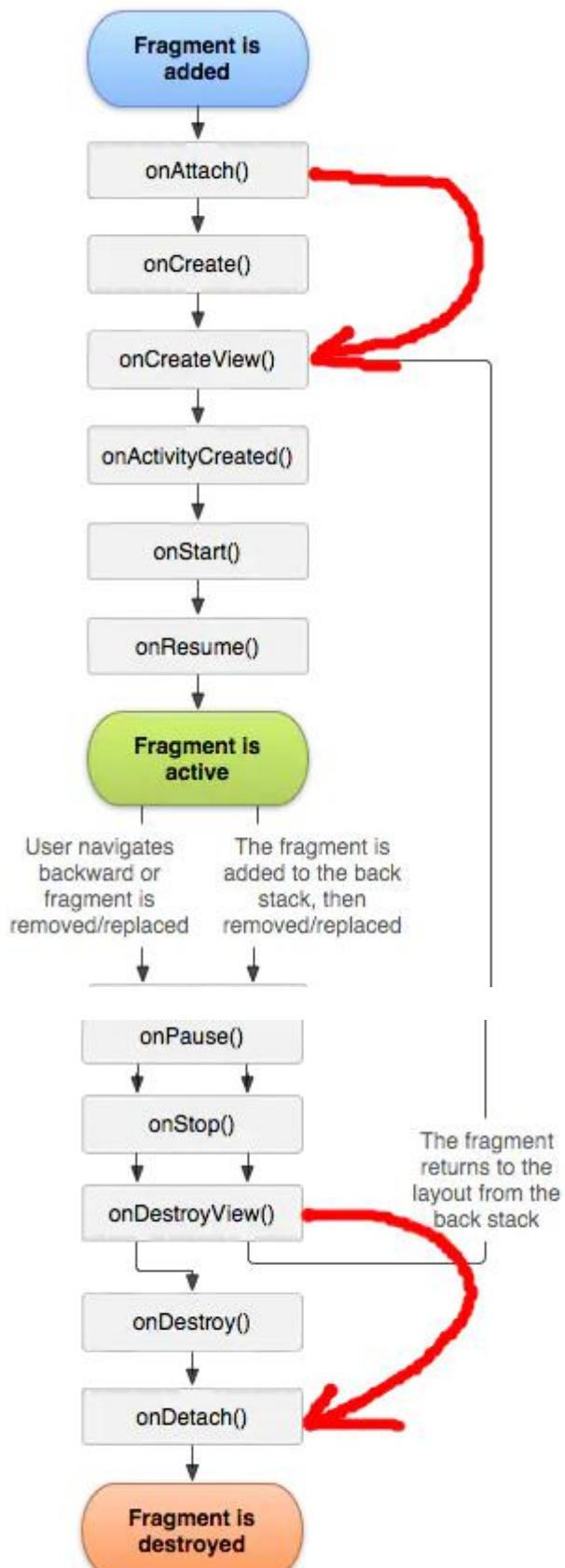
(6) onDestoryView: 该回调方法在视图层次结构与 Fragment 分离之后调用。

(7) onDestory: 不再使用 Fragment 时调用。(备注: Fragment 仍然附加到 Activity 并任然可以找到，但是不能执行其他操作)

(8) onDetach: Fragme 生命周期最后回调函数，调用后，Fragment 不再与 Activity 绑定，释放资源。

Fragment 注意事项

在使用 Fragment 时，我发现了一个金矿，那就是 setRetainInstance() 方法，此方法可以有效地提高系统的运行效率，对流畅性要求较高的应用可以适当采用此方法进行设置。



setRetainInstance(true)使用效果

Fragment 有一个非常强大的功能——就是可以在 Activity 重新创建时可以不完全销毁 Fragment，以便 Fragment 可以恢复。在 onCreate() 方法中调用 setRetainInstance(true/false) 方法是最佳位置。当 Fragment 恢复时的生命周期如上图所示，注意图中的红色箭头。当在 onCreate() 方法中调用了 setRetainInstance(true) 后，Fragment 恢复时会跳过 onCreate() 和 onDestroy() 方法，因此不能在 onCreate() 中放置一些初始化逻辑，切忌！

四、Service 相关

1、进程保活

2、Service 的运行线程（生命周期方法全部在主线程）

3、Service 启动方式以及如何停止

4、ServiceConnection 里面的回调方法运行在哪个线程？

文章、startService 和 bindService 区别

文章、Android 进程保活的一般套路

自己曾经也在这个问题上伤过脑经，前几日刚好有一个北京的哥们在 QQ 说在做 IM 类的项目，问我进程保活如何处理比较恰当，决定去总结一下，网上搜索一下进程常驻的方案好多好多，但是很多的方案都是不靠谱的或者不是最好的，结合很多资料，今天总结一下 Android 进程保活的一些方案，都附有完整的实现源码，有些可能你已经知道，但是有些你可能是第一次听说，（1 像素 Activity，前台服务，账号同步，Jobscheduler，相互唤醒，系统服务捆绑，如果你都了解了，请忽略）经过多方面的验证，Android 系统中在没有白名单的情况下做一个任何情况下都不被杀死的应用是基本不可能的，但是我们可以做到我们的应用基本不被杀死，如果杀死可以马上满血复活，原谅我讲的特别含蓄，毕竟现在的技术防不胜防啊，不死应用还是可能的。

有几个问题需要思考，系统为什么会杀掉进程，杀的为什么是我的进程，这是按照什么标准来选择的，是一次性干掉多个进程，还是一个接着一个杀，保活套路一堆，如何进行进程保活才是比较恰当……如果这些问题你还还存在，或许这篇文章可以解答。

一、进程初步了解

每一个 Android 应用启动后至少对应一个进程，有的是多个进程，而且主流应用中多个进程的应用比例较大

正在运行		
	今日头条 1 个进程和 2 个服务	12 MB 1:55:17
	今日头条 1 个进程和 1 个服务	12 MB 1:55:20
	今日头条 2 个进程和 2 个服务	126 MB 1:55:15
	微信 1 个进程和 1 个服务	27 MB 31:05:11
	微信 1 个进程和 1 个服务	179 MB 28:01:14
	QQ 2 个进程和 2 个服务	80 MB 07:40
	微博 1 个进程和 1 个服务	76 MB 24:46:12
	微博 1 个进程和 2 个服务	11 MB 24:46:10
	微博 2 个进程和 3 个服务	173 MB 3:24:25
	简书 2 个进程和 1 个服务	62 MB 13:00

Paste_Image.png

1、如何查看进程解基本信息

对于任何一个进程，我们都可以通过 adb shell ps|grep <package_name>的方式来查看它的基本信息

```
C:\Users\...>adb shell  
root@generic_x86:/ # ps|grep com.wangjing.processlive  
u0_a16 3881 1223 873024 37108 SyS_epoll_ b733cf15 S com.wangjing.processlive  
root@generic_x86:/ #
```

值	解释
u0_a16	USER 进程当前用户
3881	进程ID
1223	进程的父进程ID
873024	进程的虚拟内存大小
37108	实际驻留“在内存中”的内存大小
com.wangjing.processlive	进程名

2、进程划分

Android 中的进程跟封建社会一样，分了三流九等，Android 系统把进程划为了如下几种（重要性从高到低），网上多位大神都详细总结过（备注：严格来说是划分了 6 种）。

2.1、前台进程(Foreground process)

场景：

- 某个进程持有一个正在与用户交互的 Activity 并且该 Activity 正处于 resume 的状态。
- 某个进程持有一个 Service，并且该 Service 与用户正在交互的 Activity 绑定。
- 某个进程持有一个 Service，并且该 Service 调用 startForeground()方法使之位于前台运行。
- 某个进程持有一个 Service，并且该 Service 正在执行它的某个生命周期回调方法，比如 onCreate()、onStart()或 onDestroy()。
- 某个进程持有一个 BroadcastReceiver，并且该 BroadcastReceiver 正在执行其 onReceive()方法。

用户正在使用的程序，一般系统是不会杀死前台进程的，除非用户强制停止应用或者系统内存不足等极端情况会杀死。

2.2、可见进程(Visible process)

场景：

- 拥有不在前台、但仍对用户可见的 Activity (已调用 onPause())。
- 拥有绑定到可见 (或前台) Activity 的 Service

用户正在使用，看得到，但是摸不着，没有覆盖到整个屏幕，只有屏幕的一部分可见进程不包含任何前台组件，一般系统也是不会杀死可见进程的，除非要在资源吃紧的情况下，要保持某个或多个前台进程存活

2.3、服务进程(Service process)

场景

- 某个进程中运行着一个 Service 且该 Service 是通过 startService() 启动的，与用户看见的界面没有直接关联。

在内存不足以维持所有前台进程和可见进程同时运行的情况下，服务进程会被杀死

2.4、后台进程(Background process)

场景：

- 在用户按了"back"或者"home"后，程序本身看不到了，但是其实还在运行的程序，比如 Activity 调用了 onPause 方法

系统可能随时终止它们，回收内存

2.5、空进程(Empty process)

场景：

- 某个进程不包含任何活跃的组件时该进程就会被置为空进程，完全没用，杀了它只有好处没坏处，第一个干它！

3、内存阈值

上面是进程的分类，进程是怎么被杀的呢？系统出于体验和性能上的考虑，app 在退到后台时系统并不会真正的 kill 掉这个进程，而是将其缓存起来。打开的应用越多，后台

缓存的进程也越多。在系统内存不足的情况下，系统开始依据自身的一套进程回收机制来判断要 kill 掉哪些进程，以腾出内存来供给需要的 app，这套杀进程回收内存的机制就叫 Low Memory Killer。那这个不足怎么来规定呢，那就是内存阈值，我们可以使用 cat /sys/module/lowmemorykiller/parameters/minfree 来查看某个手机的内存阈值。

```
root@generic_x86:/ # cat /sys/module/lowmemorykiller/parameters/minfree  
18432, 23040, 27648, 32256, 36864, 46080  
root@generic_x86:/ #
```

注意这些数字的单位是 page. 1 page = 4 kb. 上面的六个数字对应的就是(MB): 72, 90, 108, 126, 144, 180，这些数字也就是对应的内存阈值，内存阈值在不同的手机上不一样，一旦低于该值，Android 便开始按顺序关闭进程。因此 Android 开始结束优先级最低的空进程，即当可用内存小于 180MB(46080*4/1024)。

读到这里，你或许有一个疑问，假设现在内存不足，空进程都被杀光了，现在要杀后台进程，但是手机中后台进程很多，难道要一次性全部都清理掉？当然不是的，进程是有它的优先级的，这个优先级通过进程的 adj 值来反映，它是 linux 内核分配给每个系统进程的一个值，代表进程的优先级，进程回收机制就是根据这个优先级来决定是否进行回收，adj 值定义在 com.android.server.am.ProcessList 类中，这个类路径是 \${android-sdk-path}\sources\android-23\com\android\server\am\ProcessList.java。oom_adj 的值越小，进程的优先级越高，普通进程 oom_adj 值是大于等于 0 的，而系统进程 oom_adj 的值是小于 0 的，我们可以通过 cat /proc/进程 id/oom_adj 可以看到当前进程的 adj 值。

```
C:\Users\...>adb shell  
root@generic_x86:/ # ps|grep com.wangjing.processlive  
u0_a16 3811 1223 873024 37108 SyS_epoll_b733cf15 S com.wangjing.processlive  
root@generic_x86:/ # cat /proc/3811/oom_adj  
0  
root@generic_x86:/ #
```

看到 adj 值是 0，0 就代表这个进程是属于前台进程，我们按下 Back 键，将应用至于后台，再次查看

```
C:\Users\...>adb shell  
root@generic_x86:/ # ps|grep com.wangjing.processlive  
u0_a16 3811 1223 873024 37108 SyS_epoll_b733cf15 S com.wangjing.processlive  
root@generic_x86:/ # cat /proc/3811/oom_adj  
0  
root@generic_x86:/ # cat /proc/3811/oom_adj  
8  
root@generic_x86:/ #
```

adj 值变成了 8, 8 代表这个进程是属于不活跃的进程，你可以尝试其他情况下，oom_adj 值是多少，但是每个手机的厂商可能不一样，oom_adj 值主要有这么几个，可以参考一下。

adj级别	值	解释
UNKNOWN_ADJ	16	预留的最低级别，一般对于缓存的进程才有可能设置成这个级别
CACHED_APP_MAX_ADJ	15	缓存进程，空进程，在内存不足的情况下就会优先被kill
CACHED_APP_MIN_ADJ	9	缓存进程，也就是空进程
SERVICE_B_ADJ	8	不活跃的进程
PREVIOUS_APP_ADJ	7	切换进程
HOME_APP_ADJ	6	与Home交互的进程
SERVICE_ADJ	5	有Service的进程
HEAVY_WEIGHT_APP_ADJ	4	高权重进程
BACKUP_APP_ADJ	3	正在备份的进程
PERCEPITBLE_APP_ADJ	2	可感知的进程，比如那种播放音乐
VISIBLE_APP_ADJ	1	可见进程
FOREGROUND_APP_ADJ	0	前台进程
PERSISTENT_SERVICE_ADJ	-11	重要进程
PERSISTENT_PROC_ADJ	-12	核心进程
SYSTEM_ADJ	-16	系统进程
NATIVE_ADJ	-17	系统起的Native进程

备注：（上表的数字可能在不同系统会有一定的出入）

根据上面的 adj 值，其实系统在进程回收跟内存回收类似也是有一套严格的策略，可以自己去了解，大概是这个样子的， **oom_adj 越大，占用物理内存越多会被最先 kill 掉**，

OK, 那么现在对于进程如何保活这个问题就转化成, 如何降低 **oom_adj** 的值, 以及如何使得我们应用占的内存最少。

二、进程保活方案

1、开启一个像素的 Activity

据说这个是手 Q 的进程保活方案, 基本思想, 系统一般是不会杀死前台进程的。所以要使得进程常驻, 我们只需要在锁屏的时候在本进程开启一个 Activity, 为了欺骗用户, 让这个 Activity 的大小是 1 像素, 并且透明无切换动画, 在开屏幕的时候, 把这个 Activity 关闭掉, 所以这个就需要监听系统锁屏广播, 我试过了, 的确好使, 如下。

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

如果直接启动一个 Activity, 当我们按下 back 键返回桌面的时候, **oom_adj** 的值是 8, 上面已经提到过, 这个进程在资源不够的情况下是容易被回收的。现在造一个一个像素的 Activity。

```
public class LiveActivity extends Activity {

    public static final String TAG = LiveActivity.class.getSimpleName();

    public static void actionToLiveActivity(Context pContext) {
        Intent intent = new Intent(pContext, LiveActivity.class);
        intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        pContext.startActivity(intent);
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, "onCreate");
        setContentView(R.layout.activity_live);

        Window window = getWindow();
        //放在左上角
        window.setGravity(Gravity.START | Gravity.TOP);
        WindowManager.LayoutParams attributes = window.getAttributes();
        //宽高设计为1个像素
        attributes.width = 1;
        attributes.height = 1;
        //起始坐标
        attributes.x = 0;
        attributes.y = 0;
        window.setAttributes(attributes);

        ScreenManager.getInstance(this).setActivity(this);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.d(TAG, "onDestroy");
    }
}
```

为了做的更隐藏，最好设置一下这个 Activity 的主题，当然也无所谓了

```
<style name="LiveStyle">
    <item name="android:windowIsTranslucent">true</item>
    <item name="android:windowBackground">@android:color/transparent</item>
    <item name="android:windowAnimationStyle">@null</item>
    <item name="android:windowNoTitle">true</item>
</style>
```

在屏幕关闭的时候把 LiveActivity 启动起来，在开屏的时候把 LiveActivity 关闭掉，所以要监听系统锁屏广播，以接口的形式通知 MainActivity 启动或者关闭 LiveActivity。

```
public class ScreenBroadcastListener {

    private Context mContext;

    private ScreenBroadcastReceiver mScreenReceiver;

    private ScreenStateListener mListener;

    public ScreenBroadcastListener(Context context) {
        mContext = context.getApplicationContext();
        mScreenReceiver = new ScreenBroadcastReceiver();
    }

    interface ScreenStateListener {

        void onScreenOn();

        void onScreenOff();
    }

    /**
     * screen状态广播接收者
     */
    private class ScreenBroadcastReceiver extends BroadcastReceiver {
        private String action = null;

        @Override
        public void onReceive(Context context, Intent intent) {
            action = intent.getAction();
            if (Intent.ACTION_SCREEN_ON.equals(action)) { // 开屏
                mListener.onScreenOn();
            } else if (Intent.ACTION_SCREEN_OFF.equals(action)) { // 锁屏
                mListener.onScreenOff();
            }
        }
    }
}
```

```
        }
    }

    public void registerListener(ScreenStateListener listener) {
        mListener = listener;
        registerListener();
    }

    private void registerListener() {
        IntentFilter filter = new IntentFilter();
        filter.addAction(Intent.ACTION_SCREEN_ON);
        filter.addAction(Intent.ACTION_SCREEN_OFF);
        mContext.registerReceiver(mScreenReceiver, filter);
    }
}
```

```
public class ScreenManager {

    private Context mContext;

    private WeakReference<Activity> mActivityWref;

    public static ScreenManager gDefualt;

    public static ScreenManager getInstance(Context pContext) {
        if (gDefualt == null) {
            gDefualt = new ScreenManager(pContext.getApplicationContext());
        }
        return gDefualt;
    }

    private ScreenManager(Context pContext) {
        this.mContext = pContext;
    }

    public void setActivity(Activity pActivity) {
        mActivityWref = new WeakReference<Activity>(pActivity);
    }

    public void startActivity() {
        LiveActivity.actionToLiveActivity(mContext);
    }

    public void finishActivity() {
        //结束掉LiveActivity
        if (mActivityWref != null) {
            Activity activity = mActivityWref.get();
            if (activity != null) {
                activity.finish();
            }
        }
    }
}
```

现在 MainActivity 改成如下

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        final ScreenManager screenManager = ScreenManager.getInstance(MainActivity.this);
        ScreenBroadcastListener listener = new ScreenBroadcastListener(this);
        listener.registerListener(new ScreenBroadcastListener.ScreenStateListener() {
            @Override
            public void onScreenOn() {
                screenManager.finishActivity();
            }

            @Override
            public void onScreenOff() {
                screenManager.startActivity();
            }
        });
    }
}
```

按下 back 之后，进行锁屏，现在测试一下 oom_adj 的值

```
root@generic_x86:/ # cat /proc/9566/oom_adj
0
```

果然将进程的优先级提高了。

但是还有一个问题，内存也是一个考虑的因素，内存越多会被最先 kill 掉，所以把上面的业务逻辑放到 Service 中，而 Service 是在另外一个 进程中，在 MainActivity 开启这个服务就行了，这样这个进程就更加的轻量，

```
public class LiveService extends Service {

    public static void toLiveService(Context pContext){
        Intent intent=new Intent(pContext,LiveService.class);
        pContext.startService(intent);
    }

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        //屏幕关闭的时候启动一个1像素的Activity，开屏的时候关闭Activity
        final ScreenManager screenManager = ScreenManager.getInstance(LiveService.this);
        ScreenBroadcastListener listener = new ScreenBroadcastListener(this);
        listener.registerListener(new ScreenBroadcastListener.ScreenStateListener() {
            @Override
            public void onScreenOn() {
                screenManager.finishActivity();
            }
            @Override
            public void onScreenOff() {
                screenManager.startActivity();
            }
        });
        return START_REDELIVER_INTENT;
    }
}
```

```
<service android:name=".LiveService"
        android:process=":live_service"/>
```

OK，通过上面的操作，我们的应用就始终和前台进程是一样的优先级了，为了省电，系统检测到锁屏事件后一段时间内会杀死后台进程，如果采取这种方案，就可以避免了这个问题。但是还是有被杀掉的可能，所以我们还需要做双进程守护，关于双进程守护，比较适合的就是 aidl 的那种方式，但是这个不是完全的靠谱，原理是 A 进程死的时候，B 还在活着，B 可以将 A 进程拉起来，反之，B 进程死的时候，A 还活着，A 可以将 B 拉起来。所以双进程守护的前提是，系统杀进程只能一个个的去杀，如果一次性杀两个，这种方法也是不 OK 的。

事实上

那么我们先来看看 Android5.0 以下的源码，ActivityManagerService 是如何关闭在应用退出后清理内存的

```
Process.killProcessQuiet(pid);
```

应用退出后，ActivityManagerService 就把主进程给杀死了，但是，在 Android5.0 以后，ActivityManagerService 却是这样处理的：

```
Process.killProcessQuiet(app.pid);
Process.killProcessGroup(app.info.uid, app.pid);
```

在应用退出后，ActivityManagerService 不仅把主进程给杀死，另外把主进程所属的进程组一并杀死，这样一来，由于子进程和主进程在同一进程组，子进程在做的事情，也就停止了。所以在 Android5.0 以后的手机应用在进程被杀死后，要采用其他方案。

2、前台服务

这种大部分人都了解，据说这个微信也用过的进程保活方案，移步[微信 Android 客户端后台保活经验分享](#)，这方案实际利用了 Android 前台 service 的漏洞。

原理如下

对于 API level < 18：调用 startForeground(ID, new Notification())，发送空的 Notification，图标则不会显示。

对于 API level >= 18：在需要提优先级的 service A 启动一个 InnerService，两个服务同时 startForeground，且绑定同样的 ID。Stop 掉 InnerService，这样通知栏图标即被移除。

```
public class KeepLiveService extends Service {
    public static final int NOTIFICATION_ID = 0x11;
    public KeepLiveService() { }
    @Override public IBinder onBind(Intent intent) {
        throw new UnsupportedOperationException("Not yet implemented");
    }
    @Override public void onCreate() {
        super.onCreate();
        // API 18 以下，直接发送 Notification 并将其置为前台
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.JELLY_BEAN_MR2) {
            startForeground(NOTIFICATION_ID, new Notification());
        } else { // API 18 以上，发送 Notification 并将其置为前台后，启动 InnerService
            Notification.Builder builder = new Notification.Builder(this);
            builder.setSmallIcon(R.mipmap.ic_launcher);
            startForeground(NOTIFICATION_ID, builder.build());
            startService(new Intent(this, InnerService.class));
        }
    }
    public class InnerService extends Service {
        @Override public IBinder onBind(Intent intent) {
            return null;
        }
        @Override public void onCreate() {
            super.onCreate();
            // 发送与 KeepLiveService 中 ID 相同的 Notification，然后将其取消并取消自己的前台显示
            Notification.Builder builder = new Notification.Builder(this);
            builder.setSmallIcon(R.mipmap.ic_launcher);
        }
    }
}
```

```
startForeground(NOTIFICATION_ID, builder.build()); new Handler().postDelayed(new Runnable() { @Override public void run() { stopForeground(true); NotificationManager manager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE); manager.cancel(NOTIFICATION_ID); stopSelf(); } }, 100); } }
```

在没有采取前台服务之前，启动应用，oom_adj 值是 0，按下返回键之后，变成 9（不同 ROM 可能不一样）

```
1|root@android:/ # ps|grep zhangwan.wj.com.processlive  
u0_a59 6827 163 904668 34668 ffffffff b7551a27 S zhangwan.wj.com.processlive  
root@android:/ # cat /proc/6827/oom_adj  
0  
root@android:/ # cat /proc/6827/oom_adj  
9
```

在采取前台服务之后，启动应用，oom_adj 值是 0，按下返回键之后，变成 2（不同 ROM 可能不一样），确实进程的优先级有所提高。

```
root@android:/ # ps|grep zhangwan.wj.com.processlive  
u0_a59 7188 163 904680 34492 ffffffff b7551a27 S zhangwan.wj.com.processlive  
root@android:/ # cat /proc/7188/oom_adj  
0  
root@android:/ # cat /proc/7188/oom_adj  
2
```

3、相互唤醒

相互唤醒的意思就是，假如你手机里装了支付宝、淘宝、天猫、UC 等阿里系的 app，那么你打开任意一个阿里系的 app 后，有可能就顺便把其他阿里系的 app 给唤醒了。这个完全有可能的。此外，开机，网络切换、拍照、拍视频时候，利用系统产生的广播也能唤醒 app，不过 Android N 已经将这三种广播取消了。





如果应用想保活，要是 QQ，微信愿意救你也行，有多少手机上没有 QQ，微信呢？或者像友盟，信鸽这种推送 SDK，也存在唤醒 app 的功能。

拉活方法

4、JobScheduler

JobScheduler 是作为进程死后复活的一种手段，native 进程方式最大缺点是费电，Native 进程费电的原因是感知主进程是否存活有两种实现方式，在 Native 进程中通过死循环或定时器，轮训判断主进程是否存活，当主进程不存活时进行拉活。其次 5.0 以上系统不支持。但是 JobScheduler 可以替代在 Android5.0 以上 native 进程方式，这种方式即使用户强制关闭，也能被拉起来，亲测可行。

```
JobScheduler@TargetApi(Build.VERSION_CODES.LOLLIPOP) public class MyJobService extends JobService { @Override public void onCreate() { super.onCreate(); startJobScheduler(); } public void startJobScheduler() { try { JobInfo.Builder builder = new JobInfo.Builder(1, new ComponentName(getApplicationContext(), MyJobService.class.getName())); builder.setPeriodic(5); builder.setPersisted(true); JobScheduler jobScheduler = (JobScheduler) this.getSystemService(Context.JOB_SCHEDULER_SERVICE); jobScheduler.schedule(builder.build()); } catch (Exception ex) { ex.printStackTrace(); } } @Override public boolean onStartJob(JobParameters jobParameters) { return false; } @Override public boolean onStopJob(JobParameters jobParameters) { return false; } }
```

5、粘性服务&与系统服务捆绑

这个是系统自带的，`onStartCommand` 方法必须具有一个整形的返回值，这个整形的返回值用来告诉系统在服务启动完毕后，如果被 Kill，系统将如何操作，这种方案虽然可以，但是在某些情况 or 某些定制 ROM 上可能失效，我认为可以多做一种保保守方案。

```
@Override  
public int onStartCommand(Intent intent, int flags, int startId) {  
    return START_REDELIVER_INTENT;  
}
```

START_STICKY

如果系统在 `onStartCommand` 返回后被销毁，系统将会重新创建服务并依次调用 `onCreate` 和 `onStartCommand`（注意：根据测试 Android2.3.3 以下版本只会调用 `onCreate` 根本不会调用 `onStartCommand`，Android4.0 可以办到），这种相当于服务又重新启动恢复到之前的状态了）。

-
-

START_NOT_STICKY

如果系统在 `onStartCommand` 返回后被销毁，如果返回该值，则在执行完 `onStartCommand` 方法后如果 Service 被杀掉系统将不会重启该服务。

-
-

START_REDELIVER_INTENT

`START_STICKY` 的兼容版本，不同的是其不保证服务被杀后一定能重启。

相比与粘性服务与系统服务捆绑更厉害一点，这个来自[爱哥](#)的研究，这里说的系统服务很好理解，比如 NotificationListenerService，NotificationListenerService 就是一个监听通知的服务，只要手机收到了通知，NotificationListenerService 都能监听到，即时用户把进程杀死，也能重启，所以说要是把这个服务放到我们的进程之中，那么就可以呵呵了

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
public class LiveService extends NotificationListenerService {

    public LiveService() {
    }

    @Override
    public void onNotificationPosted(StatusBarNotification sbn) {
    }

    @Override
    public void onNotificationRemoved(StatusBarNotification sbn) {
    }
}
```

但是这种方式需要权限

```
<service
    android:name=".LiveService"
    android:permission="android.permission.BIND_NOTIFICATION_LISTENER_SERVICE">
    <intent-filter>
        <action android:name="android.service.notification.NotificationListenerService" />
    </intent-filter>
</service>
```

所以你的应用要是有消息推送的话，那么可以用这种方式去欺骗用户。

文章、关于 Android 进程保活，你所需要知道的一切

保活手段

当前业界的 Android 进程保活手段主要分为** 黑、白、灰 **三种，其大致的实现思路如下：

黑色保活：不同的 app 进程，用广播相互唤醒（包括利用系统提供的广播进行唤醒）

白色保活: 启动前台 Service

灰色保活: 利用**系统的漏洞**启动前台 Service

黑色保活

所谓黑色保活，就是利用不同的 app 进程使用广播来进行相互唤醒。举个 3 个比较常见的场景：

场景 1: 开机，网络切换、拍照、拍视频时候，利用系统产生的广播唤醒 app

场景 2: 接入第三方 SDK 也会唤醒相应的 app 进程，如微信 sdk 会唤醒微信，支付宝 sdk 会唤醒支付宝。由此发散开去，就会直接触发了下面的 **场景 3**

场景 3: 假如你手机里装了支付宝、淘宝、天猫、UC 等阿里系的 app，那么你打开任意一个阿里系的 app 后，有可能就顺便把其他阿里系的 app 给唤醒了。（只是拿阿里打个比方，其实 BAT 系都差不多）

没错，我们的 Android 手机就是一步一步的被上面这些场景给拖卡机的。

针对**场景 1**，估计 Google 已经开始意识到这些问题，所以在最新的 Android N 取消了 ACTION_NEW_PICTURE（拍照），ACTION_NEW_VIDEO（拍视频），CONNECTIVITY_ACTION（网络切换）等三种广播，无疑给了很多 app 沉重的打击。我猜他们的心情是下面这样的



而开机广播的话，记得有一些定制 ROM 的厂商早已经将其去掉。

针对**场景 2** 和 **场景 3**，因为调用 SDK 唤醒 app 进程属于正常行为，此处不讨论。但是在借助 LBE 分析 app 之间的唤醒路径的时候，发现了两个问题：

1. 很多推送 SDK 也存在唤醒 app 的功能
2. app 之间的唤醒路径真是多，且错综复杂

我把自己使用的手机测试结果给大家围观一下（**我的手机是小米 4C，刷了原生的 Android5.1 系统，且已经获得 Root 权限才能查看这些唤醒路径**）





我们直接点开 简书 的唤醒路径进行查看



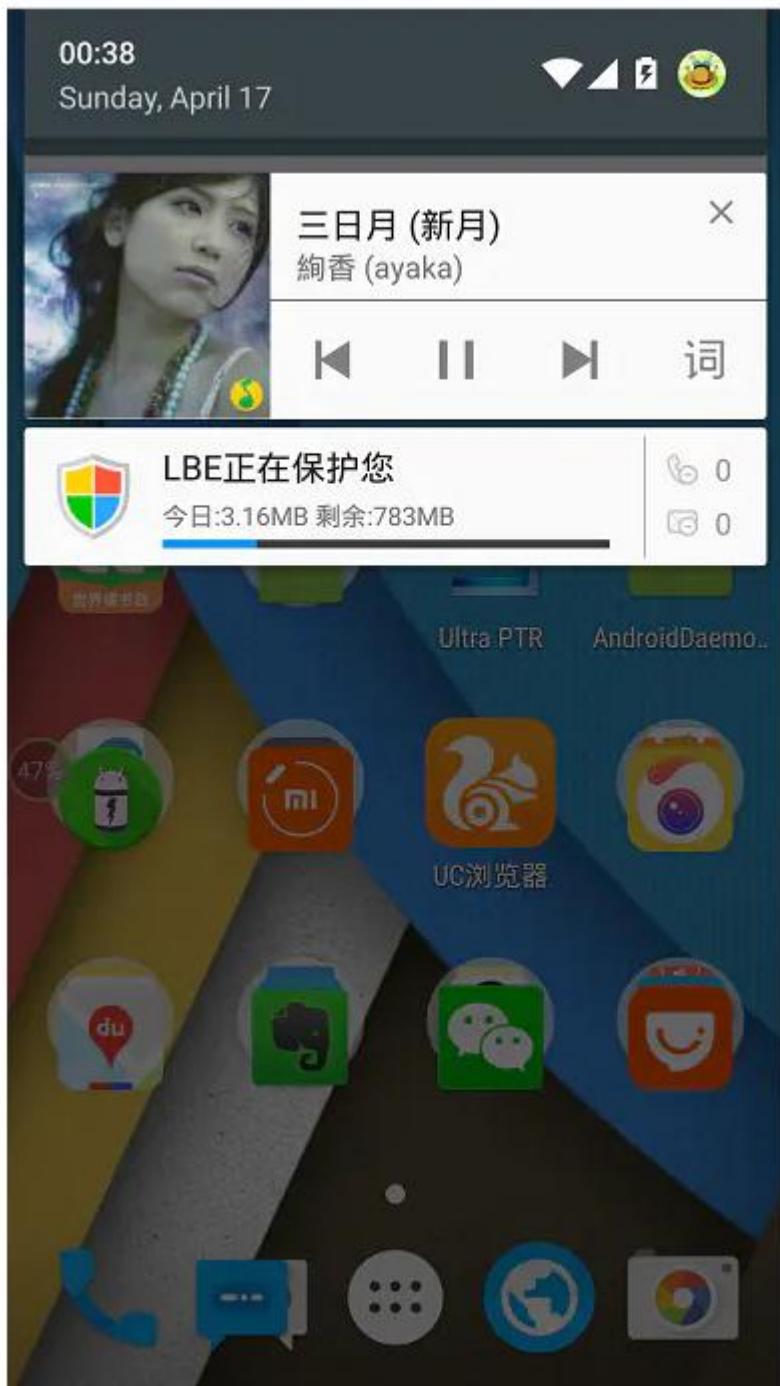
可以看到以上 3 条唤醒路径，但是涵盖的唤醒应用总数却达到了 $23+43+28$ 款，数目真心惊人。请注意，这只是我手机上一款 app 的唤醒路径而已，到了这里是不是有点细思极恐。

当然，这里依然存在一个疑问，就是 LBE 分析这些唤醒路径和互相唤醒的应用是基于什么思路，我们不得而知。所以我们也无法确定其分析结果是否准确，如果有 LBE 的童鞋看到此文章，不知可否告知一下思路呢？但是，手机打开一个 app 就唤醒一大批，我自己可是亲身体验到这种酸爽的……



白色保活

白色保活手段非常简单，就是调用系统 api 启动一个前台的 Service 进程，这样会在系统的通知栏生成一个 Notification，用来让用户知道有这样一个 app 在运行着，哪怕当前的 app 退到了后台。如下方的 LBE 和 QQ 音乐这样：



灰色保活

灰色保活，这种保活手段是应用范围最广泛。它是利用系统的漏洞来启动一个前台的 Service 进程，与普通的启动方式区别在于，它不会在系统通知栏处出现一个 Notification，看起来就如同运行着一个后台 Service 进程一样。这样做带来的好处就是，用户无法察觉到你运行着一个前台进程（因为看不到 Notification），但你的进程优先级又是高于普通后台进程的。那么如何利用系统的漏洞呢，大致的实现思路和代码如下：

- 思路一： API < 18，启动前台 Service 时直接传入 new Notification();
- 思路二： API >= 18，同时启动两个 id 相同的前台 Service，然后再将后启动的 Service 做 stop 处理；

```
public class GrayService extends Service {  
  
    private final static int GRAY_SERVICE_ID = 1001;  
  
    @Override  
  
    public int onStartCommand(Intent intent, int flags, int startId) {  
  
        if (Build.VERSION.SDK_INT < 18) {  
  
            startForeground(GRAY_SERVICE_ID, new Notification());//API <  
18 ，此方法能有效隐藏 Notification 上的图标  
  
        } else {  
  
            Intent innerIntent = new Intent(this, GrayInnerService.clas  
s);  
  
            startService(innerIntent);  
  
            startForeground(GRAY_SERVICE_ID, new Notification());  
  
        }  
  
        return super.onStartCommand(intent, flags, startId);  
    }  
  
    ...  
    ...
```

```
/**  
 * 给 API >= 18 的平台上用的灰色保活手段  
 */  
  
public static class GrayInnerService extends Service {  
  
    @Override  
  
    public int onStartCommand(Intent intent, int flags, int startId)  
{  
  
        startForeground(GRAY_SERVICE_ID, new Notification());  
  
        stopForeground(true);  
  
        stopSelf();  
  
        return super.onStartCommand(intent, flags, startId);  
    }  
  
}  
}}}
```

代码大致就是这样，能让你神不知鬼不觉的启动着一个前台 Service。其实市面上很多 app 都用着这种灰色保活的手段，什么？你不信？好吧，我们来验证一下。流程很简单，打开一个 app，看下系统通知栏有没有一个 Notification，如果没有，我们就进入手机的 adb shell 模式，然后输入下面的 shell 命令

```
dumpsys activity services PackageName
```

打印出指定包名的所有进程中的 Service 信息，看下有没有 **isForeground=true** 的关键信息。如果通知栏没有看到属于 app 的 Notification 且又看到 **isForeground=true** 则说明了，此 app 利用了这种灰色保活的手段。

下面分别是我手机上微信、qq、支付宝、陌陌的测试结果，大家有兴趣也可以自己验证一下。

```
* ServiceRecord{372af99c u0 com.tencent.mm/.booter.CoreService}
intent={cmp=com.tencent.mm/.booter.CoreService}
packageName=com.tencent.mm
processName=com.tencent.mm:push
baseDir=/data/app/com.tencent.mm-1/base.apk
dataDir=/data/data/com.tencent.mm
app=ProcessRecord[389896f 11800:com.tencent.mm:push/u0a64]
isForeground=true foregroundId=-1213 foregroundNoti=Notification(pri=0 contentView=com.tencent.mm/0x1090077 vibrate=null sound=null defaults=0x0 flags=0x62 color=0xffff07d8b vis=PRIVATE)
createTime=-49ms963ms startingBgTimeout=-42m14s002ms
lastActivity=-5m11s338ms restartTime=-42m29s812ms createdFromFg=false
startRequested=true delayedStop=false stopIfKilled=false callStart=true lastStartId=11
Bindings:
* IntentBindRecord[e302ccc CREATE]:
  intent={cmp=com.tencent.mm/.booter.CoreService}
  binder=android.os.BinderProxy@7f6b15
  requested=true received=true hasBound=true doRebind=false
* Client AppBindRecord{301f2e2a ProcessRecord[106s7714 11638:com.tencent.mm/u0a64]}
  Per-process Connections:
    ConnectionRecord[425f9d4 u0 CR com.tencent.mm/.booter.CoreService:@01dfcd327]
```

```
* ServiceRecord{207e66f7 u0 com.tencent.mobileqq/.app.CoreService$KernelService}
intent={cmp=com.tencent.mobileqq/.app.CoreService$KernelService}
packageName=com.tencent.mobileqq
processName=com.tencent.mobileqq
baseDir=/data/app/com.tencent.mobileqq-2/base.apk
dataDir=/data/data/com.tencent.mobileqq
app=ProcessRecord[33e202bf 26530:com.tencent.mobileqq/u0a65]
isForeground=true foregroundId=537045978 foregroundNoti=Notification(pri=0 contentView=com.tencent.mobileqq/0x1090077 vibrate=null sound=null defaults=0x0 flags=0x62 color=0xffff07d8b vis=PRIVATE)
createTime=-40s775ms startingBgTimeout=-
lastActivity=-40s775ms restartTime=-40s775ms createdFromFg=true
startRequested=true delayedStop=false stopIfKilled=true callStart=true lastStartId=1
```

```
* ServiceRecord{28193c6f u0 com.eg.android.AlipayGphone/com.alipay.android.launcher.service.LauncherService$InnerService}
intent={cmp=com.eg.android.AlipayGphone/com.alipay.android.launcher.service.LauncherService$InnerService}
packageName=com.eg.android.AlipayGphone
processName=com.eg.android.AlipayGphone
baseDir=/data/app/com.eg.android.AlipayGphone-2/base.apk
dataDir=/data/data/com.eg.android.AlipayGphone
app=ProcessRecord[1ad3039b 15556:com.eg.android.AlipayGphone/u0a73]
isForeground=true foregroundId=168810881 foregroundNoti=Notification(pri=0 contentView=com.eg.android.AlipayGphone/0x1090077 vibrate=null sound=null defaults=0x0 flags=0x62 color=0xffff07d8b vis=PRIVATE)
createTime=-2m56s270ms startingBgTimeout=-
lastActivity=-2m56s270ms restartTime=-2m56s270ms createdFromFg=true
startRequested=true delayedStop=false stopIfKilled=true callStart=true lastStartId=1
```

支付宝

```
* ServiceRecord{36569b0b u0 com.innomo.mono/.android.service.KService}
intent={cmp=com.innomo.mono/.android.service.KService}
packageName=com.innomo.mono
processName=com.innomo.mono:in
baseDir=/data/app/com.innomo.mono-2/base.apk
dataDir=/data/data/com.innomo.mono
app=ProcessRecord[369bb5c 13252:com.innomo.mono/in/u0a06]
isForeground=true foregroundId=9998 foregroundNoti=Notification(pri=0 contentView=com.innomo.mono/0x1090077 vibrate=null sound=null defaults=0x0 flags=0x72 color=0x00000000 vis=PRIVATE)
createTime=-39m12s718ms startingBgTimeout=-38m57s003ms
lastActivity=-34s2ms restartTime=-39m12s570ms createdFromFg=false
startRequested=true delayedStop=false stopIfKilled=false callStart=true lastStartId=11
Bindings:
* IntentBindRecord{10647021}:
  intent={cmp=com.innomo.mono/.android.service.KService}
```

陌陌

其实 Google 察觉到了此漏洞的存在，并逐步进行封堵。这就是为什么这种保活方式分 API >= 18 和 API < 18 两种情况，从 Android5.0 的 ServiceRecord 类的 postNotification 函数源代码中可以看到这样的一行注释

```

public void postNotification() {
    final int appUid = appInfo.uid;
    final int appPid = app.pid;
    if (foregroundId != 0 && foregroundNoti != null) {
        // Do asynchronous communication with notification manager to
        // avoid deadlocks.
        final String localPackageName = packageName;
        final int localForegroundId = foregroundId;
        final Notification localForegroundNoti = foregroundNoti;
        ams.mHandler.post(new Runnable() {
            public void run() {
                NotificationManagerInternal nm = LocalServices.getService(
                    NotificationManagerInternal.class);
                if (nm == null) {
                    return;
                }
                try {
                    if (localForegroundNoti.icon == 0) {
                        // It is not correct for the caller to supply a notification
                        // icon, but this used to be able to slip through, so for
                        // those dirty apps give it the app's icon.
                        localForegroundNoti.icon = appInfo.icon;
                    }
                    // Do not allow apps to present a sneaky invisible content view either.
                    localForegroundNoti.contentView = null;
                    localForegroundNoti.bigContentView = null;
                    CharSequence appName = appInfo.loadLabel(
                        ams.mContext.getPackageManager());
                    if (appName == null) {
                        appName = appInfo.packageName;
                    }
                }
            }
        });
    }
}

```

当某一天 API ≥ 18 的方案也失效的时候，我们就又要另谋出路了。需要注意的是，**使用灰色保活并不代表着你的 Service 就永生不死了，只能说是提高了进程的优先级。如果你的 app 进程占用了大量的内存，按照回收进程的策略，同样会干掉你的 app。**感兴趣于灰色保活是如何利用系统漏洞不显示 Notification 的童鞋，可以研究一下系统的 ServiceRecord、NotificationManagerService 等相关源代码，因为不是本文的重点，所以不做详述。

唠叨的分割线

到这里基本就介绍完了** 黑、白、灰 **三种实现方式，仅仅从代码层面去讲保活是不够的，我希望能够通过系统的进程回收机制来理解保活，这样能够让我们更好的避免踩到进程被杀的坑。

进程回收机制

熟悉 Android 系统的童鞋都知道，系统出于体验和性能上的考虑，app 在退到后台时系统并不会真正的 kill 掉这个进程，而是将其缓存起来。打开的应用越多，后台缓存的进程也越多。在系统内存不足的情况下，系统开始依据自身的一套进程回收机制来判断要 kill 掉哪些进程，以腾出内存来供给需要的 app。这套杀进程回收内存的机制叫 **Low Memory Killer**，它是基于 Linux 内核的 **OOM Killer (Out-Of-Memory killer)** 机制诞生。

了解完 **Low Memory Killer**, 再科普一下 **oom_adj**。什么是 **oom_adj**? 它是 linux 内核分配给每个系统进程的一个值, 代表进程的优先级, 进程回收机制就是根据这个优先级来决定是否进行回收。对于 **oom_adj** 的作用, 你只需要记住以下几点即可:

- 进程的 **oom_adj** 越大, 表示此进程优先级越低, 越容易被杀回收; 越小, 表示进程优先级越高, 越不容易被杀回收
- 普通 app 进程的 **oom_adj>=0**, 系统进程的 **oom_adj <0**

那么我们如何查看进程的 **oom_adj** 呀, 需要用到下面的两个 shell 命令

```
ps | grep PackageName //获取你指定的进程信息
```

```
shell@cancro:/ $ ps | grep com.clock.daemon
u0_a263 18937 271 1491564 49004 ffffffff 00000000 S com.clock.daemon:bg
u0_a263 19016 271 1549640 66664 ffffffff 00000000 S com.clock.daemon
u0_a263 27785 271 1490508 50936 ffffffff 00000000 S com.clock.daemon:gray
l 11@
```

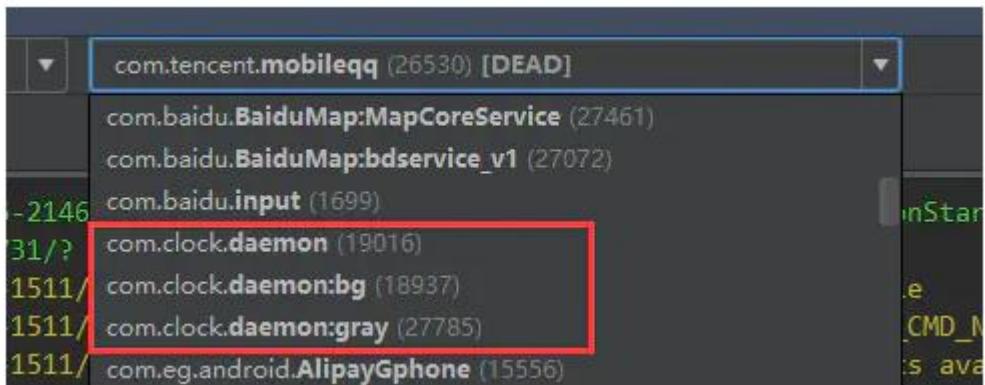
这里是以我写的 demo 代码为例子, 红色圈中部分别为下面三个进程的 ID

UI 进程: **com.clock.daemon**

普通后台进程: **com.clock.daemon:bg**

灰色保活进程: **com.clock.daemon:gray**

当然, 这些进程的 id 也可以通过 AndroidStudio 获得



接着我们来再来获取三个进程的 **oom_adj**

```
cat /proc/进程 ID/oom_adj
```

```
shell@cancro:/ $ ps | grep com.clock.daemon
u0_a263 18937 271 1491564 49004 ffffffff 00000000 S com.clock.daemon:bg
u0_a263 19016 271 1549640 66664 ffffffff 00000000 S com.clock.daemon
u0_a263 27785 271 1490508 50936 ffffffff 00000000 S com.clock.daemon:gray
shell@cancro:/ $ cat /proc/27785/oom_adj
0
shell@cancro:/ $ cat /proc/18937/oom_adj
15
shell@cancro:/ $ cat /proc/19016/oom_adj
0
```

从上图可以看到 UI 进程和灰色保活 Service 进程的 `oom_adj=0`, 而普通后台进程 `oom_adj=15`。到这里估计你也能明白，**为什么普通的后台进程容易被回收，而前台进程则不容易被回收了吧。**但明白这个还不够，接着看下图

```
shell@cancro:/ $ ps | grep com.clock.daemon
u0_a263 18937 271 1491564 49004 ffffffff 00000000 S com.clock.daemon:bg
u0_a263 19016 271 1520460 63188 ffffffff 00000000 S com.clock.daemon
u0_a263 27785 271 1490508 50936 ffffffff 00000000 S com.clock.daemon:gray
shell@cancro:/ $ cat /proc/19016/oom_adj
6
shell@cancro:/ $ cat /proc/27785/oom_adj
1
shell@cancro:/ $
```

上面是我把 app 切换到后台，再进行一次 `oom_adj` 的检验，你会发现 UI 进程的值从 0 变成了 6, 而灰色保活的 Service 进程则从 0 变成了 1。这里可以观察到，**app 退到后台时，其所有的进程优先级都会降低。但是 UI 进程是降低最为明显的，因为它占用的内存资源最多，系统内存不足的时候肯定优先杀这些占用内存高的进程来腾出资源。所以，为了尽量避免后台 UI 进程被杀，需要尽可能的释放一些不用的资源，尤其是图片、音视频之类的。**

从 Android 官方文档中，我们也能看到优先级从高到低列出了这些不同类型的进程：**Foreground process、Visible process、Service process、Background process、Empty process**。

五、Android 布局优化之 ViewStub、include、merge

1、什么情况下使用 ViewStub、include、merge？

2、他们的原理是什么？

文章、布局优化神器 include 、 merge、 ViewStub 标签详解

Tips

使用Android Studio 的同学,可以直接在布局文件对应控件:
右键 -> Refactor -> Extract -> Style 抽取样式
右键 -> Refactor -> Extract -> Layout 抽取布局 include标签

include 标签使用：

一、定义要实现(抽取)的 layout 布局：

```
//include_test.xml
```

```
<?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="wrap_content"
    android:gravity="center_horizontal"
    android:orientation="vertical">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        android:text="@string/textview"
        android:textSize="24sp"/>

    <EditText
        android:id="@+id/editText"
        android:hint="@string/divide"
        android:layout_width="300dp"
        android:layout_height="wrap_content"/>

</LinearLayout>
```

```
-----  
//include_text_relative  
  
<?xml version="1.0" encoding="utf-8"?>  
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:gravity="center_horizontal"  
>  
  
<TextView  
    android:id="@+id/textView"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginTop="20dp"  
    android:text="TextView_Relative"  
    android:textSize="24sp"/>  
  
<EditText  
    android:id="@+id/editText"  
    android:layout_width="300dp"  
    android:layout_height="wrap_content"  
    android:layout_below="@+id/textView"  
    android:hint="@string/divide"/>  
  
</RelativeLayout>
```

```
-----  
//include_toolbar.xml  
  
<?xml version="1.0" encoding="utf-8"?>  
<android.support.v7.widget.Toolbar  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:id="@+id/tb_toolbar"  
    android:layout_width="match_parent"  
    android:layout_height="?attr actionBarSize"  
    android:background="#00f"  
    app:theme="@style/AppTheme"  
    app:title="这是一个ToolBar"
```

```
    app:titleTextColor="@android:color/white"/>
```

二、Activity 的 XML 布局文件调用 include 标签：

```
<?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"
    >
    <!--测试 layout 和<include>都设置 ID 的情况-->
    <include
        android:id="@+id/tb_toolbar"
        layout="@layout/include_toolbar"/>

    <!--如果只有单个 include 这样写就可以,加载的布局的子 View,直接 findViewById 就能找到-->
    <include layout="@layout/include_text"/>

    <!--如果有多个 include,需要添加 ID 属性-->
    <include
        android:id="@+id/include_text1"
        layout="@layout/include_text"/>

    <!--这个 layout 用 RelativeLayout 实现-->
    <!--如果要使用 layout_margin 这样的属性,要同时加上 layout_w/h 属性,不然没反应-->
    <include
        android:id="@+id/include_text2"
        layout="@layout/include_text_relative"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="50dp"/>

</LinearLayout>
```

三、Activity 中调用 include 标签 layout 中的子 View：

```
private void initView() {
    //如果 include 布局根容器和 include 标签中的 id 设置的是不同的值, 这里获取的
    mToolbar 值将为 null
```

```
Toolbar mToolbar = (Toolbar) findViewById(R.id.tb_toolbar);
setSupportActionBar(mToolbar);

//普通 include 标签用法,直接拿子 View 属性实现
TextView textView = (TextView) findViewById(R.id.textView);
textView.setText("不加 ID 实现的 include 标签");

//多个 include 标签用法,添加 ID,findViewById 找到 layout,再找子控件
View view_include = findViewById(R.id.include_text1);
TextView          view_include_textView           =           (TextView)
view_include.findViewById(R.id.textView);
view_include_textView.setText("加了 ID 实现的 include 标签");

//多个 include 标签用法,添加 ID,findViewById 找到 layout,再找子控件
View view_include_Relative = findViewById(R.id.include_text2);
TextView          view_textView_relative           =           (TextView)
view_include_Relative.findViewById(R.id.textView);
view_textView_relative.setText("加了 ID 实现的 include 标签(RelaviteLayout)");

}

include 标签 Demo 效果图
```



==include 使用注意==

1. 一个xml布局文件有多个include标签需要设置ID,才能找到相应子View的控件,否则只能找到第一个include的layout布局,以及该布局的控件
2. include标签如果使用layout_xx属性,会覆盖被include的xml文件根节点对应的layout_xx属性,建议在include标签调用的布局设置好宽高位置,防止不必要的bug
3. include 添加id,会覆盖被include的xml文件根节点ID,这里建议include和被include覆盖的xml文件根节点设置同名的ID,不然有可能会报空指针异常
4. 如果要在include标签下使用RelativeLayout,如layout_margin等其他属性,记得要同时设置layout_width和layout_height,不然其它属性会没反应

merge 标签

merge 标签主要用于辅助 include 标签,在使用 include 后可能导致布局嵌套过多,多余的 layout 节点或导致解析变慢(可通过 hierarchy viewer 工具查看布局的嵌套情况)

官方文档说明:merge 用于消除视图层次结构中的冗余视图,例如根布局是 LinearLayout,那么我们又 include 一个 LinearLayout 布局就没意义了,反而会减慢 UI 加载速度

[merge 官方文档](#)

merge 标签常用场景:

根布局是 FrameLayout 且不需要设置 background 或 padding 等属性,可以用 merge 代替,因为 Activity 的 ContentView 父元素就是 FrameLayout,所以可以用 merge 消除只剩一个.

某布局作为子布局被其他布局 include 时,使用 merge 当作该布局的顶节点,这样在被引入时顶结点会自动被忽略,而将其子节点全部合并到主布局中.

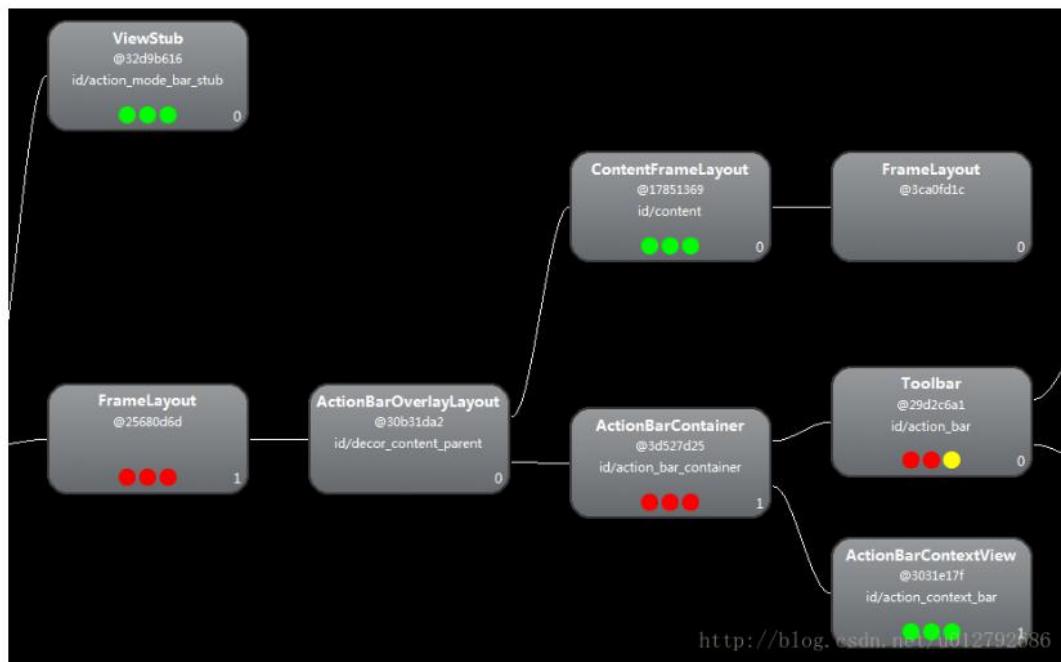
自定义 View 如果继承 LinearLayout(ViewGroup),建议让自定义 View 的布局文件根布局设置成 merge,这样能少一层结点.

merge 标签使用:

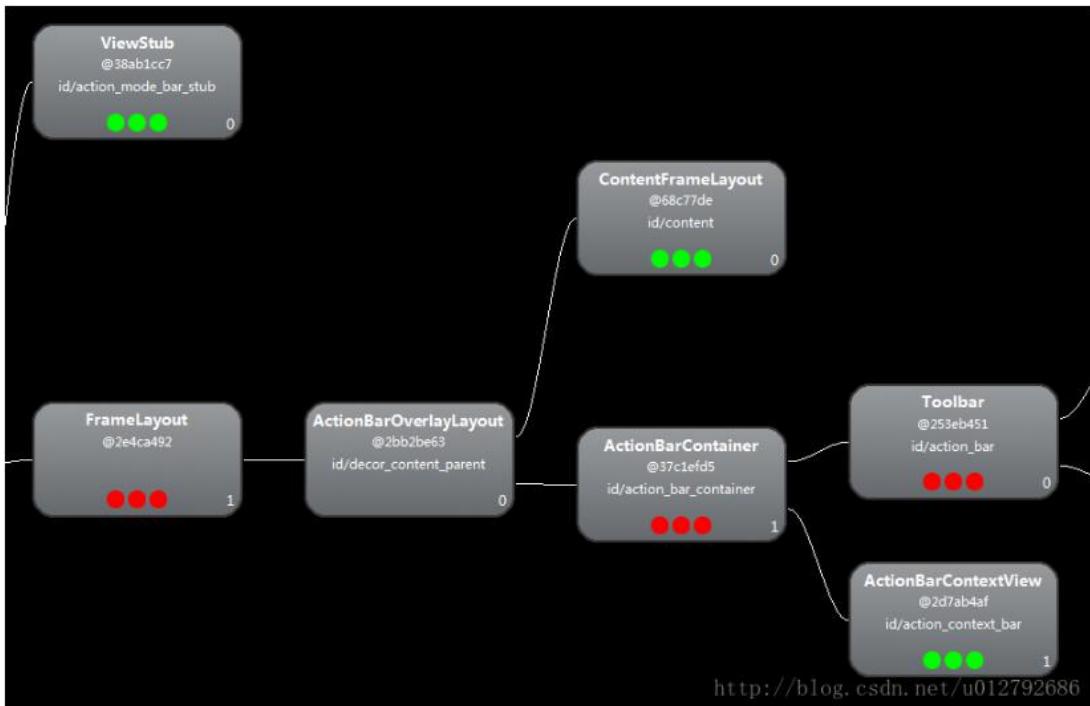
在 XML 布局文件的根布局如 RelativeLayout 直接改成 merge 即可

merge 标签使用前后 Hierarchy Viewer 截图

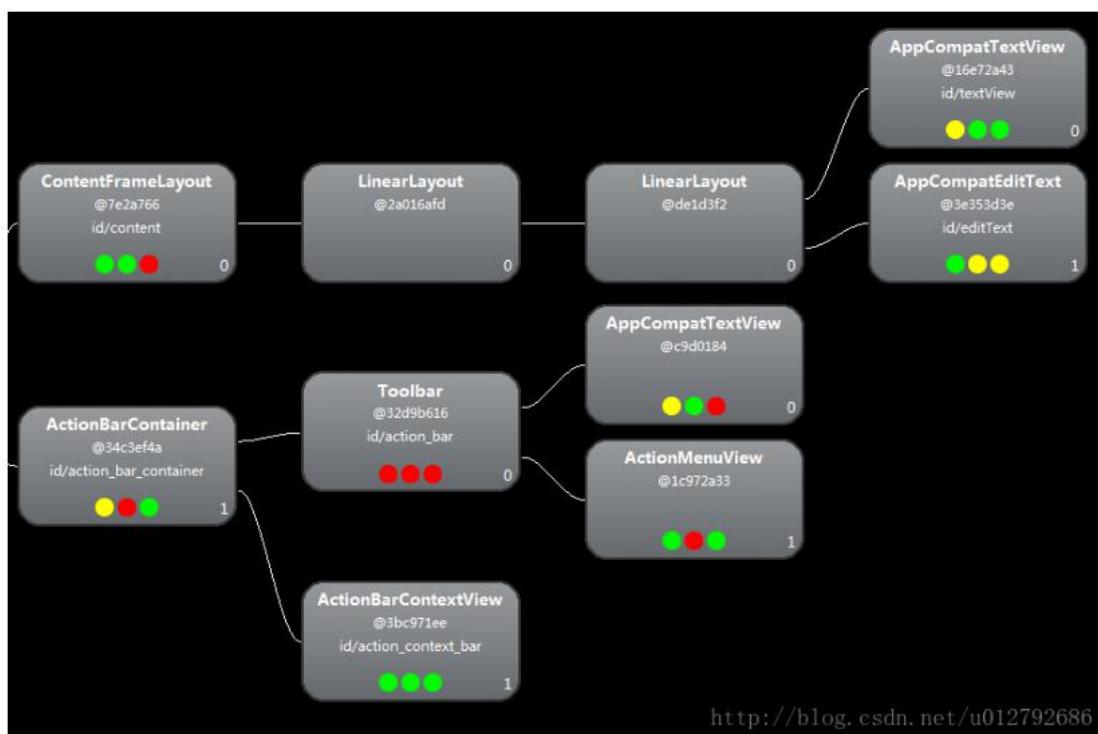
FrameLayout 替换 merge 前



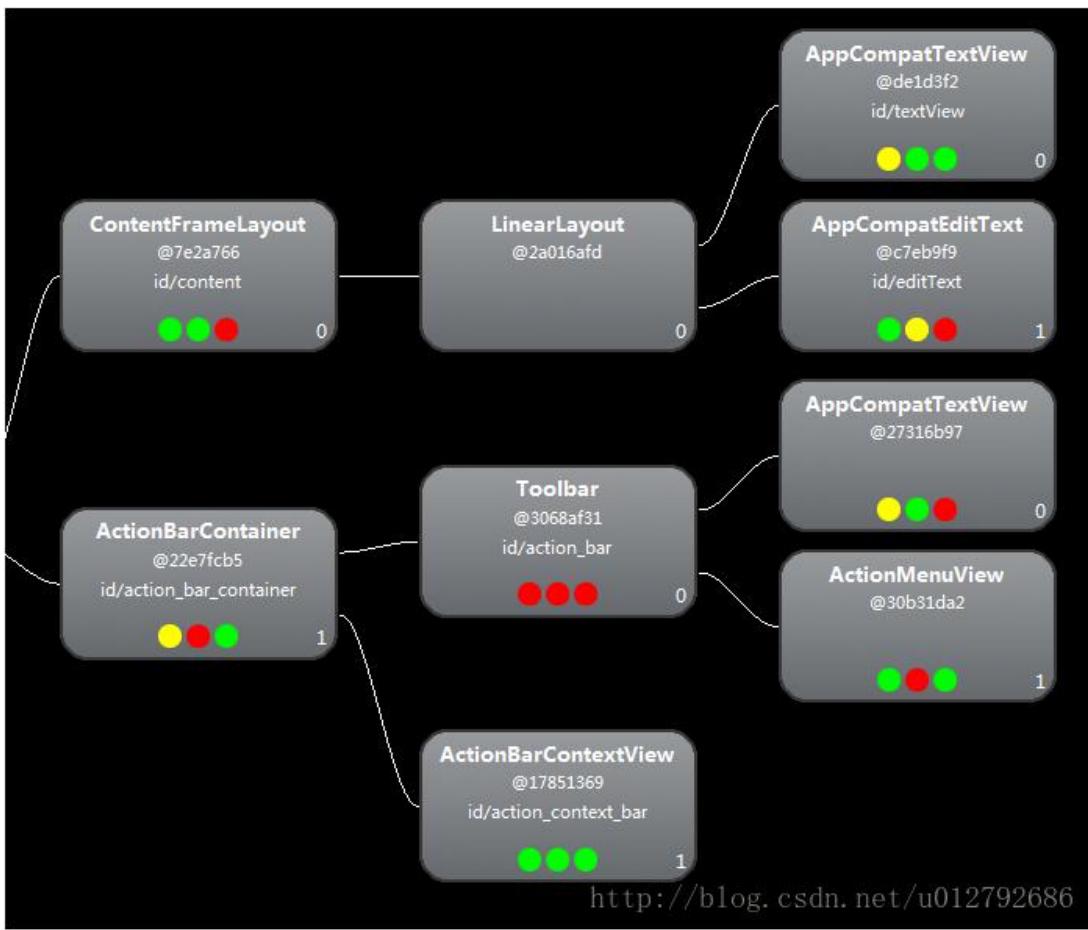
FrameLayout 替换 merge 后



include 标签对应 layout 根布局 替换 merge 前



include 标签对应 layout 根布局 替换 merge 后



==merge 使用注意==

1. 因为merge标签并不是View,所以在通过LayoutInflate.inflate()方法渲染的时候,第二个参数必须指定一个父容器,且第三个参数必须为true,也就是必须为merge下的视图指定一个父亲节点.
2. 因为merge不是View,所以对merge标签设置的所有属性都是无效的.
3. 注意如果include的layout用了merge,调用include的根布局也使用了merge标签,那么就失去布局的属性了
4. merge标签必须使用在根布局
5. ViewStub标签中的layout布局不能使用merge标签

ViewStub 标签

ViewStub 标签最大的优点是当你需要时才会加载,使用它并不会影响 UI 初始化时的性能.各种不常用的布局像进度条、显示错误消息等可以使用 **ViewStub** 标签,以减少内存使用量,加快渲染速度.**ViewStub** 是一个不可见的,实际上是把宽高设置为 0 的 View.效果有点类似普通的 `view.setVisible()`,但性能体验提高不少

第一次初始化时,初始化的是 ViewStub View,当我们调用 inflate() 或
setVisibility() 后会被 remove 掉,然后在将其中的 layout 加到当前 view
hierarchy 中

[ViewStub 官方文档链接](#)

//官方例子

```
<ViewStub
```

```
    android:id="@+id/stub_import"
```

```
    <!--android:inflateId : 重写 ViewStub 的父布局控件的 Id-->
```

```
    android:inflatedId="@+id/panel_import"
```

```
    <<!--android:layout : 设置 ViewStub 被 inflate 的布局-->
```

```
        android:layout="@layout/progress_overlay"
```

```
        android:layout_width="fill_parent"
```

```
        android:layout_height="wrap_content"
```

```
        android:layout_gravity="bottom" />
```

//当你想加载布局时，可以使用下面其中一种方法：

((ViewStub)

```
findViewById(R.id.stub_import).setVisibility(View.VISIBLE);
```

// or

```
View importPanel = ((ViewStub)  
findViewById(R.id.stub_import)).inflate();
```

判断 ViewStub(做单例)是否已经加载过:

1. 如果通过 setVisibility 来加载,那么通过判断可见性即可;
2. 如果通过 inflate()来加载,判断 ViewStub 的 ID 是否为 null 来判断
(findViewById(...))

```
public class ViewStubActivity extends AppCompatActivity  
implements View.OnClickListener, ViewStub.OnInflateListener {
```

@Override

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

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

    }

private View networkErrorView;

private void showNetError() {

    // not repeated infalte

    if (networkErrorView != null) {

        //setVisibility()方式加载布局,加载次数不限

        networkErrorView.setVisibility(View.VISIBLE);

        return;
    }

    //inflate()方式加载布局,只能加载一次

    ViewStub stub = (ViewStub)

    findViewById(R.id.network_error_layout);

    stub.setOnInflateListener(this);
}
```

```
    networkErrorView = stub.inflate();

    Button networkSetting = (Button)

networkErrorView.findViewById(R.id.network_miss);

    networkSetting.setOnClickListener(this);

    Button refresh = (Button)

findViewById(R.id.network_refresh);

    refresh.setOnClickListener(this);

}
```

```
private void showNormal() {

    if (networkErrorView != null) {

        networkErrorView.setVisibility(View.GONE);

    }
}
```

```
public void show(View view) {
```



```
        break;

    }

}

/**

 *
 * @param stub 当前待 inflate 的 ViewStub 控件
 *
 * @param inflated 当前被 inflate 的 View 视图
 */

@Override

public void onInflate(ViewStub stub, View inflated) {

}

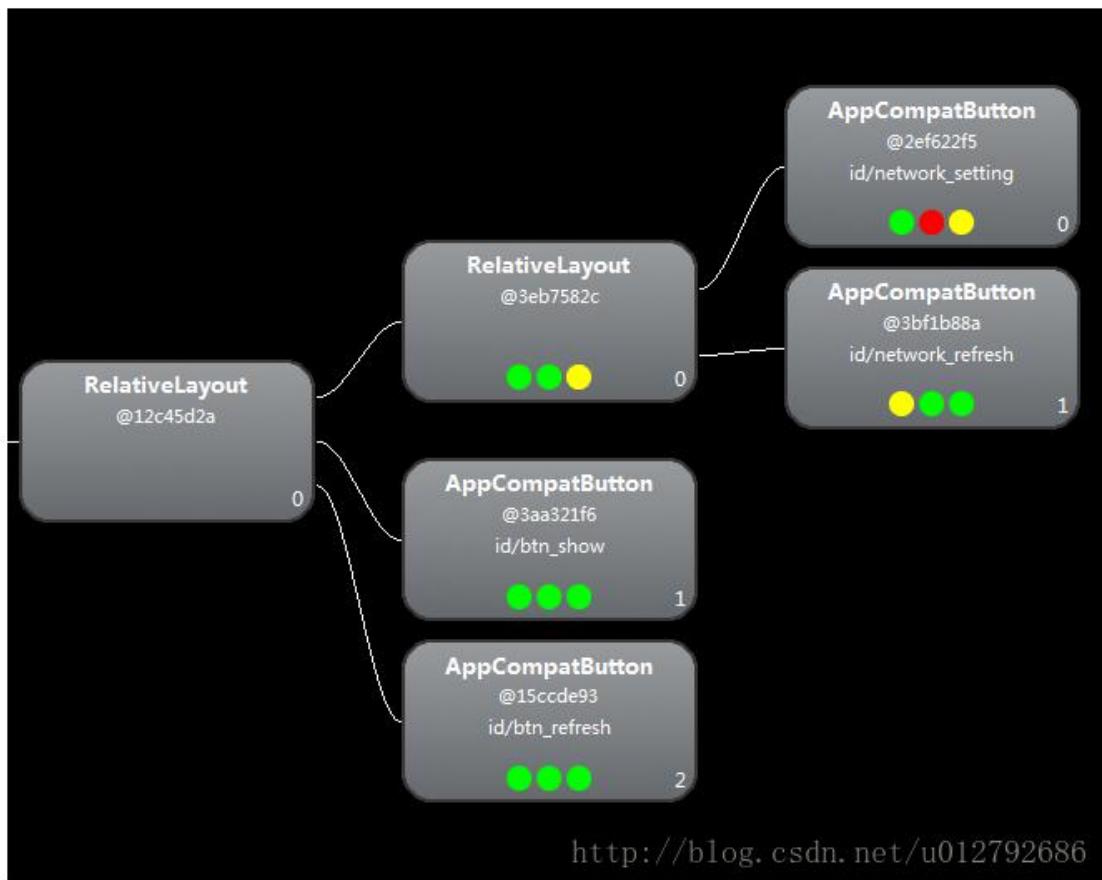
}
```

ViewStub 标签 Demo 效果图：

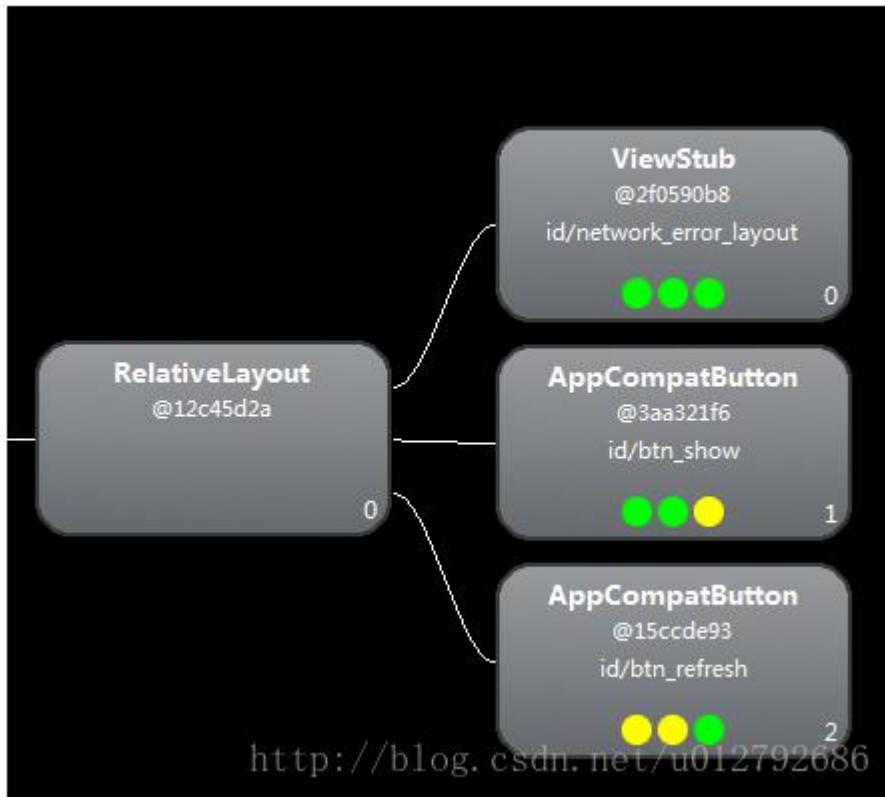


ViewStub 标签使用前后 Hierarchy Viewer 截图

ViewStub 标签使用前



ViewStub 标签使用后



==ViewStub 标签使用注意==

- 1. ViewStub标签不支持merge标签
- 2. ViewStub的inflate只能被调用一次,第二次调用会抛出异常,setVisibility可以被调用多次,但不建议这么做(ViewStub 调用过后,可能被GC掉,再调用setVisibility()会报异常)
- 3. 为ViewStub赋值的android:layout_XX属性会替换待加载布局文件的根节点对应的属性

扩展:

Space 组件

在 ConstraintLayout 出来前,我们写布局都会使用到大量的 margin 或 padding,但是这种方式可读性会很差,加一个布局嵌套又会损耗性能

鉴于这种情况,我们可以使用 space, 使用方式和 View 一样,不过主要用来占位置,不会有任何显示效果

文章、Android 布局优化之 ViewStub、include、merge 使用与源码分析

一、include

首先用得最多的应该是 include，按照官方的意思，include 就是为了解决重复定义相同布局的问题。例如你有五个界面，这五个界面的顶部都有布局一模一样的一个返回按钮和一个文本控件，在不使用 include 的情况下你在每个界面都需要重新在 xml 里面写同样的返回按钮和文本控件的顶部栏，这样的重复工作会相当的恶心。使用 include 标签，我们只需要把这个会被多次使用的顶部栏独立成一个 xml 文件，然后在需要使用的地方通过 include 标签引入即可。其实就相当于 C 语言、C++ 中的 include 头文件一样，我们把一些常用的、底层的 API 封装起来，然后复用，需要的时候引入它即可，而不必每次都自己写一遍。示例如下：

my_title_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:id="@+id/my_title_parent_id"
    android:layout_height="wrap_content" >

    <ImageButton
        android:id="@+id/back_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_launcher" />

    <TextView
        android:id="@+id/title_tv"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
    android:layout_centerVertical="true"
    android:layout_marginLeft="20dp"
    android:layout_toRightOf="@+id/back_btn"
    android:gravity="center"
    android:text="我的 title"
    android:textSize="18sp" />

</RelativeLayout>
```

include 布局文件：

```
<?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" >

    <include
        android:id="@+id/my_title_ly"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        layout="@layout/my_title_layout" />

    <!-- 代码省略 -->
</LinearLayout>
```

这样我们就可以使用 my_title_layout 了。

注意事项

使用 include 最常见的问题就是 findViewById 查找不到目标控件，这个问题出现的前提是在 include 时设置了 id ,而在 findViewById 时却用了被 include 进来的布局的根元素 id。例如上述例子中，include 时设置了该布局的 id 为 my_title_ly , 而 my_title_layout.xml 中的根视图的 id 为 my_title_parent_id。此时如果通过 findViewById 来找 my_title_parent_id 这个控件，然后再查找 my_title_parent_id 下的子控件则会**抛出空指针**。

代码如下：

```
View titleView = findViewById(R.id.my_title_parent_id) ;  
// 此时 titleView 为空，找不到。此时空指针  
  
TextView titleTextView = (TextView)titleView.findViewById(R.id.title_tv) ;  
titleTextView.setText("new Title");
```

其正确的使用形式应该如下:

```
// 使用 include 时设置的 id,即 R.id.my_title_ly  
  
View titleView = findViewById(R.id.my_title_ly) ;  
// 通过 titleView 找子控件  
  
TextView titleTextView = (TextView)titleView.findViewById(R.id.title_tv) ;  
titleTextView.setText("new Title");
```

或者更简单的直接查找它的子控件:

```
TextView titleTextView = (TextView)findViewById(R.id.title_tv) ;  
titleTextView.setText("new Title");
```

那么使用 findViewById(R.id.my_title_parent_id)为什么报空指针呢？ 我们来分析它的源码看看吧。对于布局文件的解析，最终都会调用到 LayoutInflater 的 inflate 方法，该方法最终又会调用 rInflate 方法，我们看看这个方法。

```
/*
 * Recursive method used to descend down the xml hierarchy and
 * instantiate
 *
 * views, instantiate their children, and then call onFinishInflate().
 */

void rInflate(XmlPullParser parser, View parent, final AttributeSet
attrs,
              boolean finishInflate) throws XmlPullParserException,
IOException {

    final int depth = parser.getDepth();
    int type;
    // 迭代 xml 中的所有元素，挨个解析
    while (((type = parser.next()) != XmlPullParser.END_TAG ||
            parser.getDepth() > depth) && type !=

            XmlPullParser.END_DOCUMENT) {

        if (type != XmlPullParser.START_TAG) {

            continue;
        }
    }
}
```

```
    }

    final String name = parser.getName();

    if (TAG_REQUEST_FOCUS.equals(name)) {
        parseRequestFocus(parser, parent);
    } else if (TAG_INCLUDE.equals(name)) {// 如果 xml 中的节点是
        include 节点，则调用 parseInclude 方法
        if (parser.getDepth() == 0) {
            throw new InflateException("<include /> cannot be
the root element");
        }
        parseInclude(parser, parent, attrs);
    } else if (TAG_MERGE.equals(name)) {
        throw new InflateException("<merge /> must be the
root element");
    } else if (TAG_1995.equals(name)) {
        final View view = new BlinkLayout(mContext, attrs);
        final ViewGroup viewGroup = (ViewGroup) parent;
        final ViewGroup.LayoutParams params =
viewGroup.generateLayoutParams(attrs);
        rInflate(parser, view, attrs, true);
    }
}
```

```
        viewGroup.addView(view, params);

    } else {

        final View view = createViewFromTag(parent, name,
attrs);

        final ViewGroup viewGroup = (ViewGroup) parent;
        final ViewGroup.LayoutParams params =
viewGroup.generateLayoutParams(attrs);

        rInflate(parser, view, attrs, true);

        viewGroup.addView(view, params);

    }

}

if (finishInflate) parent.onFinishInflate();

}
```

这个方法其实也就是遍历 xml 中的所有元素，然后挨个进行解析。例如解析到一个标签，那么就根据用户设置的一些 layout_width、layout_height、id 等属性来构造一个 TextView 对象，然后添加到父控件(ViewGroup 类型)中。标签也是一样的，我们看到遇到 include 标签时，会调用 parseInclude 函数，这就是对标签的解析，我们看看吧。

```
private void parseInclude(XmlPullParser parser, View parent, AttributeSet
attrs)
throws XmlPullParserException, IOException {
```

```
int type;

if (parent instanceof ViewGroup) {

    final int layout = attrs.getAttributeResourceValue(null,
"layout", 0);

    if (layout == 0) {// include 标签中没有设置 layout 属性，会抛出
        异常

    final String value = attrs.getAttributeValue(null, "layout");

    if (value == null) {

        throw new InflateException("You must specifiy a
layout in the"
                                + "      "      include      tag:      <include
layout="@layout/layoutID\" />");

    } else {

        throw new InflateException("You must specifiy a
valid layout "
                                + "reference. The layout ID " + value + " is
not valid.");
    }
} else {

    final XmlResourceParser childParser =
```

```
        getContext().getResources().getLayout(layout);

try {// 获取属性集，即在 include 标签中设置的属性
    final AttributeSet childAttrs =
        Xml.asAttributeSet(childParser);

    while ((type = childParser.next()) !=

XmlPullParser.START_TAG &&

        type != XmlPullParser.END_DOCUMENT) {
        // Empty.

    }

    if (type != XmlPullParser.START_TAG) {
        throw new InflateException(childParser.getPositionDescription() +
                ": No start tag found!");

    }

    // 1、解析 include 中的第一个元素
    final String childName = childParser.getName();

    // 如果第一个元素是 merge 标签，那么调用 rInflate 函
数解析

    if (TAG_MERGE.equals(childName)) {
```

```
// Inflate all children.  
  
        rInflate(childParser, parent, childAttrs, false);  
  
    } else { // 2、我们例子中的情况会走到这一步,首先根据  
        include 的属性集创建被 include 进来的 xml 布局的根 view  
  
        // 这里的根 view 对应为 my_title_layout.xml 中的  
        RelativeLayout  
  
        final View view = createViewFromTag(parent,  
        childName, childAttrs);  
  
        final ViewGroup group = (ViewGroup) parent; //  
        include 标签的 parent view  
  
        ViewGroup.LayoutParams params = null;  
  
        try { // 获 3、取布局属性  
            params =  
                group.generateLayoutParams(attrs);  
  
        } catch (RuntimeException e) {  
            params =  
                group.generateLayoutParams(childAttrs);  
  
        } finally {  
            if (params != null) { // 被 include 进来的根  
                view.setLayoutParams(params);  
            }  
        }  
    }  
}
```

```
        }

    }

// 4、 Inflate all children. 解析所有子控件
rInflate(childParser, view, childAttrs, true);

// Attempt to override the included layout's
android:id with the
// one set on the <include /> tag itself.

TypedArray a =
mContext.obtainStyledAttributes(attrs,
                                com.android.internal.R.styleable.View, 0, 0);

int id =
a.getResourceId(com.android.internal.R.styleable.View_id, View.NO_ID);

// While we're at it, let's try to override
android:visibility.

int visibility =
a.getInt(com.android.internal.R.styleable.View_visibility, -1);

a.recycle();

// 5、将 include 中设置的 id 设置给根 view,因此实
际上 my_title_layout.xml 中的 RelativeLayout 的 id 会变成 include 标签中的
id , include 不设置 id , 那么也可以通过 relative 的找到.
```

```
if (id != View.NO_ID) {  
    view.setId(id);  
}  
  
switch (visibility) {  
    case 0:  
        view.setVisibility(View.VISIBLE);  
        break;  
    case 1:  
        view.setVisibility(View.INVISIBLE);  
        break;  
    case 2:  
        view.setVisibility(View.GONE);  
        break;  
}  
  
// 6、将根 view 添加到父控件中  
group.addView(view);  
}  
}  
} finally {  
    childParser.close();  
}  
}
```

```
    } else {
        throw new InflateException("<include /> can only be used
inside of a ViewGroup");
    }

final int currentDepth = parser.getDepth();
while (((type = parser.next()) != XmlPullParser.END_TAG ||
        parser.getDepth() > currentDepth) && type !=
XmlPullParser.END_DOCUMENT) {
    // Empty
}
}
```

整个过程就是根据不同的标签解析不同的元素，首先会解析 include 元素，然后再解析被 include 进来的布局的 root view 元素。在我们的例子中对应的 root view 就是 id 为 my_title_parent_id 的 RelativeLayout，然后再解析 root view 下面的所有元素，这个过程是从上面注释的 2~4 的过程，然后是设置布局参数。我们注意看注释 5 处，这里就解释了为什么 include 标签和被引入的布局的根元素都设置了 id 的情况下，通过被引入的根元素的 id 来查找子控件会找不到的情况。我们看到，注释 5 处的会判断 include 标签的 id 如果不是 View.NO_ID 的话会把该 id 设置给被引入的布局根元素的 id，即此时在我们的例子中被引入的 id 为 my_title_parent_id 的根元素 RelativeLayout 的 id 被设置成了 include 标签中的 id，即 RelativeLayout 的 id 被动态修改成了“my_title_ly”。因此此

时我们再通过 “my_title_parent_id” 这个 id 来查找根元素就会找不到了！

所以结论就是：如果 include 中设置了 id ,那么就通过 include 的 id 来查找被 include 布局根元素的 View ;如果 include 中没有设置 Id, 而被 include 的布局的根元素设置了 id ,那么通过该根元素的 id 来查找该 view 即可。拿到根元素后查找其子控件都是一样的。

二、ViewStub

我们先看看官方的说明:

ViewStub is a lightweight view with no dimension and doesn' t draw anything or participate in the layout. As such, it' s cheap to inflate and cheap to leave in a view hierarchy. Each ViewStub simply needs to include the android:layout attribute to specify the layout to inflate.

其实 ViewStub 就是一个宽高都为 0 的一个 View ,它默认是不可见的，只有通过调用 setVisibility 函数或者 Inflate 函数才会将其要装载的目标布局给加载出来，从而达到延迟加载的效果，这个要被加载的布局通过 android:layout 属性来设置。例如我们通过一个 ViewStub 来惰性加载一个消息流的评论列表，因为一个帖子可能并没有评论，此时我可以不加载这个评论的 ListView ，只有当有评论时我才把它加载出来，这样就去除了加载 ListView 带来的资源消耗以及延时，示例如下：

```
<ViewStub  
    android:id="@+id/stub_import"  
    android:inflatedId="@+id/stub_comm_lv"
```

```
    android:layout="@layout/my_comment_layout"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:layout_gravity="bottom" /
```

my_comment_layout.xml 如下：

```
<?xml version="1.0" encoding="utf-8"?>  
<ListView  
    xmlns:android="http://schemas.android.com/apk/res/android"  
        android:layout_width="match_parent"  
        android:id="@+id/my_comm_lv"  
        android:layout_height="match_parent" >  
  
</ListView>
```

在运行时，我们只需要控制 id 为 stub_import 的 ViewStub 的可见性或者调用 inflate() 函数来控制是否加载这个评论列表即可。示例如下：

```
public class MainActivity extends Activity {
```

```
    public void onCreate(Bundle b){  
        // main.xml 中包含上面的 ViewStub  
        setContentView(R.layout.main);
```

```
// 方式 1 , 获取 ViewStub,  
ViewStub          listStub          =          (ViewStub)  
findViewById(R.id.stub_import);  
  
// 加载评论列表布局  
listStub.setVisibility(View.VISIBLE);  
  
// 获取到评论 ListView ,注意这里是通过 ViewStub 的 inflatedId  
来获取
```

```
ListView commLv = findViewById(R.id.stub_comm_lv);  
  
if ( listStub.getVisibility() == View.VISIBLE ) {  
  
    // 已经加载, 否则还没有加载  
  
}  
  
}
```

通过 setVisibility(View.VISIBLE) 来加载评论列表，此时你要获取到评论 ListView 对象的话，则需要通过 findViewById 来查找，而这个 id 并不是就是 ViewStub 的 id。

这是为什么呢？

我们先看 ViewStub 的部分代码吧：

```
@SuppressWarnings({"UnusedDeclaration"})  
public ViewStub(Context context, AttributeSet attrs, int defStyle)
```

```
{  
    TypedArray a = context.obtainStyledAttributes(attrs,  
        com.android.internal.R.styleable.ViewStub,  
        defStyle, 0);  
  
    // 获取 inflatedId 属性  
    mInflatedId =  
        a.getResourceId(R.styleable.ViewStub_inflatedId, NO_ID);  
    mLayoutResource =  
        a.getResourceId(R.styleable.ViewStub_layout, 0);  
  
    a.recycle();  
  
    a = context.obtainStyledAttributes(attrs,  
        com.android.internal.R.styleable.View, defStyle, 0);  
    mID = a.getResourceId(R.styleable.View_id, NO_ID);  
    a.recycle();  
  
    initialize(context);  
}  
  
private void initialize(Context context) {  
    mContext = context;
```

```
        setVisibility(GONE);// 设置不可教案

        setWillNotDraw(true);// 设置不绘制

    }

@Override

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

    setMeasuredDimension(0, 0);// 宽高都为 0

}

@Override

public void setVisibility(int visibility) {

    if (mInflatedViewRef != null) {// 如果已经加载过则只设置

        Visibility 属性

        View view = mInflatedViewRef.get();

        if (view != null) {

            view.setVisibility(visibility);

        } else {

            throw new IllegalStateException("setVisibility called

on un-referenced view");

        }

    }

}
```

```
    } else {// 如果未加载,这加载目标布局
        super.setVisibility(visibility);
        if (visibility == VISIBLE || visibility == INVISIBLE) {
            inflate();// 调用 inflate 来加载目标布局
        }
    }

}

/***
 * Inflates the layout resource identified by {@link
 #getLayoutResource()}
 * and replaces this StubbedView in its parent by the inflated
 layout resource.
 *
 * @return The inflated layout resource.
 *
 */
public View inflate() {
    final ViewParent viewParent = getParent();
}

if (viewParent != null && viewParent instanceof ViewGroup)
{
```

```
    if (mLayoutResource != 0) {  
  
        final ViewGroup parent = (ViewGroup)  
viewParent;// 获取 ViewStub 的 parent view ,也是目标布局根元素的 parent  
view  
  
        final LayoutInflater factory =  
LayoutInflater.from(mContext);  
  
        final View view = factory.inflate(mLayoutResource,  
parent,  
false);// 1、加载目标布局  
  
        // 2、如果 ViewStub 的 inflatedId 不是 NO_ID 则把  
inflatedId 设置为目标布局根元素的 id , 即评论 ListView 的 id  
  
        if (mInflatedId != NO_ID) {  
  
            view.setId(mInflatedId);  
  
        }  
  
        final int index = parent.indexOfChild(this);  
  
        parent.removeViewInLayout(this);// 3、将 ViewStub  
自身从 parent 中移除
```

```
    final ViewGroup.LayoutParams layoutParams =  
getLayoutParams();  
  
    if (layoutParams != null) {
```

```
parent.addView(view, index, layoutParams); //
```

4、将目标布局的根元素添加到 parent 中，有参数

```
} else {
```

```
parent.addView(view, index); // 4、将目标布局的
```

根元素添加到 parent 中

```
}
```

```
mInflatedViewRef = new
```

```
WeakReference<View>(view);
```

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

```
    mInflateListener.onInflate(this, view);
```

```
}
```

```
return view;
```

```
} else {
```

```
    throw new IllegalArgumentException("ViewStub
```

```
must have a valid layoutResource");
```

```
}
```

```
} else {
```

```
    throw new IllegalStateException("ViewStub must have a
```

```
non-null ViewGroup viewParent");
```

```
    }  
}
```

可以看到，其实最终加载目标布局的还是 inflate() 函数，在该函数中将加载目标布局，获取到根元素后，如果 mInflatedId 不为 NO_ID 则把 mInflatedId 设置为根元素的 id，这也是为什么我们在获取评论 ListView 时会使用 findViewById(R.id.stub_comm_lv) 来获取，其中的 stub_comm_lv 就是 ViewStub 的 inflatedId。当然如果你没有设置 inflatedId 的话还是可以通过评论列表的 id 来获取的，例如 findViewById(R.id.my_comm_lv)。然后就是 ViewStub 从 parent 中移除、把目标布局的根元素添加到 parent 中。最后会把目标布局的根元素返回，因此我们在调用 inflate() 函数时可以直接获得根元素，省掉了 findViewById 的过程。

还有一种方式加载目标布局的就是直接调用 ViewStub 的 inflate() 方法，示例如下：

```
public class MainActivity extends Activity {  
  
    // 把 commLv2 设置为类的成员变量  
    ListView commLv2 = null;  
  
    //  
  
    public void onCreate(Bundle b){  
        // main.xml 中包含上面的 ViewStub  
        setContentView(R.layout.main);
```

```
// 方式二

ViewStub           listStub2           =           (ViewStub)

findViewById(R.id.stub_import) ;

// 成员变量 commLv2 为空则代表未加载

if ( commLv2 == null ) {

// 加载评论列表布局， 并且获取评论 ListView,inflate 函数直接返回

ListView 对象

commLv2 = (ListView)listStub2.inflate();

} else {

// ViewStub 已经加载

}

}

}

注意事项
```

1. 判断是否已经加载过， 如果通过 setVisibility 来加载，那么通过判断可见性即可；如果通过 inflate()来加载是不可以通过判断可见性来处理的，而需要使用方式 2 来进行判断。
2. findViewById 的问题，注意 ViewStub 中是否设置了 inflatedId，如果设置了则需要通过 inflatedId 来查找目标布局的根元素。

三、Merge

首先我们看官方的说明:

The tag helps eliminate redundant view groups in your view hierarchy when including one layout within another. For example, if your main layout is a vertical LinearLayout in which two consecutive views can be re-used in multiple layouts, then the re-usable layout in which you place the two views requires its own root view. However, using another LinearLayout as the root for the re-usable layout would result in a vertical LinearLayout inside a vertical LinearLayout. The nested LinearLayout serves no real purpose other than to slow down your UI performance.

其实就是减少在 include 布局文件时的层级。标签是这几个标签中最让我费解的，大家可能想不到，标签竟然会是一个 Activity，里面有一个 LinearLayout 对象。

```
/**  
 * Exercise <merge /> tag in XML files.  
 */  
  
public class Merge extends Activity {  
  
    private LinearLayout mLayout;  
  
    @Override  
    protected void onCreate(Bundle icicle) {  
        super.onCreate(icicle);
```

```
mLayout = new LinearLayout(this);
mLayout.setOrientation(LinearLayout.VERTICAL);

LayoutInflater.from(this).inflate(R.layout.merge_tag, mLayout);

setContentView(mLayout);

}

public ViewGroup getLayout() {
    return mLayout;
}

}
```

使用 merge 来组织子元素可以减少布局的层级。例如我们在复用一个含有多个子控件的布局时，肯定需要一个 ViewGroup 来管理，例如这样：

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <ImageView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
```

```
    android:scaleType="center"
    android:src="@drawable/golden_gate" />

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="20dip"
    android:layout_gravity="center_horizontal|bottom"
    android:padding="12dip"

    android:background="#AA000000"
    android:textColor="#ffffffff"

    android:text="Golden Gate" />

</FrameLayout>
```

使用 merge 标签就会消除上图中蓝色的 FrameLayout 层级。示例如下：

```
<merge
    xmlns:android="http://schemas.android.com/apk/res/android">
```

```
<ImageView  
    android:layout_width="fill_parent"  
    android:layout_height="fill_parent"  
  
    android:scaleType="center"  
    android:src="@drawable/golden_gate" />
```

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginBottom="20dip"  
    android:layout_gravity="center_horizontal|bottom"  
  
    android:padding="12dip"  
  
    android:background="#AA000000"  
    android:textColor="#ffffffff"  
  
    android:text="Golden Gate" />
```

```
</merge>
```

那么它是如何实现的呢，我们还是看源码吧。相关的源码也是在 LayoutInflator

的 inflate() 函数中。

```
public View inflate(XmlPullParser parser, ViewGroup root, boolean
attachToRoot) {

    synchronized (mConstructorArgs) {

        final AttributeSet attrs = Xml.asAttributeSet(parser);

        Context lastContext = (Context)mConstructorArgs[0];
        mConstructorArgs[0] = mContext;

        View result = root;

        try {
            // Look for the root node.

            int type;
            while ((type = parser.next()) != XmlPullParser.START_TAG
&&

                    type != XmlPullParser.END_DOCUMENT) {

                // Empty

            }

            if (type != XmlPullParser.START_TAG) {
                throw new
InflateException(parser.getPositionDescription()
                    + ": No start tag found!");
            }
        }
    }
}
```

```
    }

final String name = parser.getName();

// m 如果是 merge 标签，那么调用 rInflate 进行解析

if (TAG_MERGE.equals(name)) {

    if (root == null || !attachToRoot) {

        throw new InflateException("<merge /> can be
used only with a valid "
+ "ViewGroup" + " root" + " and
attachToRoot=true");

    }

    // 解析 merge 标签
    rInflate(parser, root, attrs, false);

} else {

    // 代码省略
}

} catch (XmlPullParserException e) {

    // 代码省略
}
```

```
        return result;

    }

}

void rInflate(XmlPullParser parser, View parent, final AttributeSet
attrs,
              boolean finishInflate) throws XmlPullParserException,
IOException {

    final int depth = parser.getDepth();
    int type;

    while (((type = parser.next()) != XmlPullParser.END_TAG ||
            parser.getDepth() > depth) && type !=
XmlPullParser.END_DOCUMENT) {

        if (type != XmlPullParser.START_TAG) {
            continue;
        }

        final String name = parser.getName();
```

```
if (TAG_REQUEST_FOCUS.equals(name)) {  
    parseRequestFocus(parser, parent);  
}  
} else if (TAG_INCLUDE.equals(name)) {  
    // 代码省略  
    parseInclude(parser, parent, attrs);  
}  
} else if (TAG_MERGE.equals(name)) {  
    throw new InflateException("<merge /> must be the root  
element");  
}  
} else if (TAG_1995.equals(name)) {  
    final View view = new BlinkLayout(mContext, attrs);  
    final ViewGroup viewGroup = (ViewGroup) parent;  
    final ViewGroup.LayoutParams params =  
    viewGroup.generateLayoutParams(attrs);  
    rInflate(parser, view, attrs, true);  
    viewGroup.addView(view, params);  
}  
} else { // 我们的例子会进入这里  
    final View view = createViewFromTag(parent, name,  
    attrs);  
    // 获取 merge 标签的 parent  
    final ViewGroup viewGroup = (ViewGroup) parent;  
    // 获取布局参数
```

```
final ViewGroup.LayoutParams params =  
viewGroup.generateLayoutParams(attrs);  
  
        // 递归解析每个子元素  
  
        rInflate(parser, view, attrs, true);  
  
        // 将子元素直接添加到 merge 标签的 parent view 中  
  
        viewGroup.addView(view, params);  
  
    }  
  
}  
  
if (finishInflate) parent.onFinishInflate();  
}
```

其实就是如果是 merge 标签，那么直接将其中的子元素添加到 merge 标签 parent 中，这样就保证了不会引入额外的层级。

六、 BroadcastReceiver 相关

1、注册方式，优先级

2、广播类型，区别

3、广播的使用场景，原理

文章、Android 广播 Broadcast 的两种注册方式静态和动态
定义

`BroadcastReceiver` , “广播接收者” 的意思 , 顾名思义 , 它就是用来接收来自系统和应用中的广播。在 `Android` 系统中 , 广播体现在方方面面 , 例如当开机完成后系统会产生一条广播 , 接收到这条广播就能实现开机启动服务的功能 ; 当网络状态改变时系统会产生一条广播 , 接收到这条广播就能及时地做出提示和保存数据等操作 ; 当电池电量改变时 , 系统会产生一条广播 , 接收到这条广播就能在电量低时告知用户及时保存进度等等。 `Android` 中的广播机制设计的非常出色 , 很多事情原本需要开发者亲自操作的 , 现在只需等待广播告知自己就可以了 , 大大减少了开发的工作量和开发周期。而作为应用开发者 , 就需要熟练掌握 `Android` 系统提供的一个开发利器 , 那就是 `BroadcastReceiver`。

在我们详细分析创建 `BroadcastReceiver` 的两种注册方式前 , 我们先罗列本次分析的大纲 :

- (1) 对静态和动态两种注册方式进行概念阐述以及演示实现步骤
- (2) 简述两种 `BroadcastReceiver` 的类型 (为后续注册方式的对比做准备)
- (3) 在默认广播类型下设置优先级和无优先级情况下两种注册方式的比较
- (4) 在有序广播类型下两种注册方式的比较
- (5) 通过接受打电话的广播 , 在程序 (`Activity`) 运行时和终止运行时 , 对两种注册方式的比较
- (6) 总结两种方式的特点

一、静态和动态注册方式

构建 Intent , 使用 sendBroadcast 方法发出广播定义一个广播接收器 , 该广播接收器继承 BroadcastReceiver , 并且覆盖 onReceive() 方法来响应事件注册该广播接收器 , 我们可以在代码中注册(动态注册) , 也可以 AndroidManifest.xml 配置文件中注册 (静态注册) 。

动态注册 :

效果如下图 :



这里就不演示点击按钮布局的实现了 , MainActivity.java 中实现代码如下 :

[java] view plain copy

1. **import** android.content.BroadcastReceiver;
2. **import** android.content.Context;
3. **import** android.content.Intent;
4. **import** android.content.IntentFilter;
5. **import** android.support.v7.app.AppCompatActivity;
6. **import** android.os.Bundle;

```
7. import android.view.Gravity;  
8. import android.view.View;  
9. import android.widget.Toast;  
10.  
11. public class MainActivity extends AppCompatActivity {  
12.     DynamicReceiver dynamicReceiver;  
13.     @Override  
14.     protected void onCreate(Bundle savedInstanceState) {  
15.         super.onCreate(savedInstanceState);  
16.         setContentView(R.layout.activity_main);  
17.         //实例化 IntentFilter 对象  
18.         IntentFilter filter = new IntentFilter();  
19.         filter.addAction("panhouye");  
20.         dynamicReceiver = new DynamicReceiver();  
21.         //注册广播接收  
22.         registerReceiver(dynamicReceiver,filter);  
23.     }  
24.     //按钮点击事件  
25.     public void send2(View v){
```

```
26.     Intent intent = new Intent();  
  
27.     intent.setAction("panhouye");  
  
28.     intent.putExtra("sele", "潘侯爷");  
  
29.     sendBroadcast(intent);  
  
30. }  
  
31. /*动态注册需在 Activity 生命周期 onPause 通过  
32. *unregisterReceiver()方法移除广播接收器，  
33. * 优化内存空间，避免内存溢出  
34. */  
  
35. @Override  
  
36. protected void onPause() {  
  
37.     super.onPause();  
  
38.     unregisterReceiver(new MyReceiver());  
  
39. }  
  
40. //通过继承 BroadcastReceiver 建立动态广播接收器  
  
41. class DynamicReceiver extends BroadcastReceiver{  
  
42.     @Override  
  
43.     public void onReceive(Context context, Intent intent) {  
  
44.         //通过吐司验证接收到广播
```

```
45.         Toast t = Toast.makeText(context,"动态广播 :  
        "+ intent.getStringExtra("sele"), Toast.LENGTH_SHORT);  
  
46.         t.setGravity(Gravity.TOP,0,0);//方便录屏 ,将吐司设置在屏幕顶端  
  
47.         t.show();  
  
48.     }  
  
49. }  
  
50.}
```

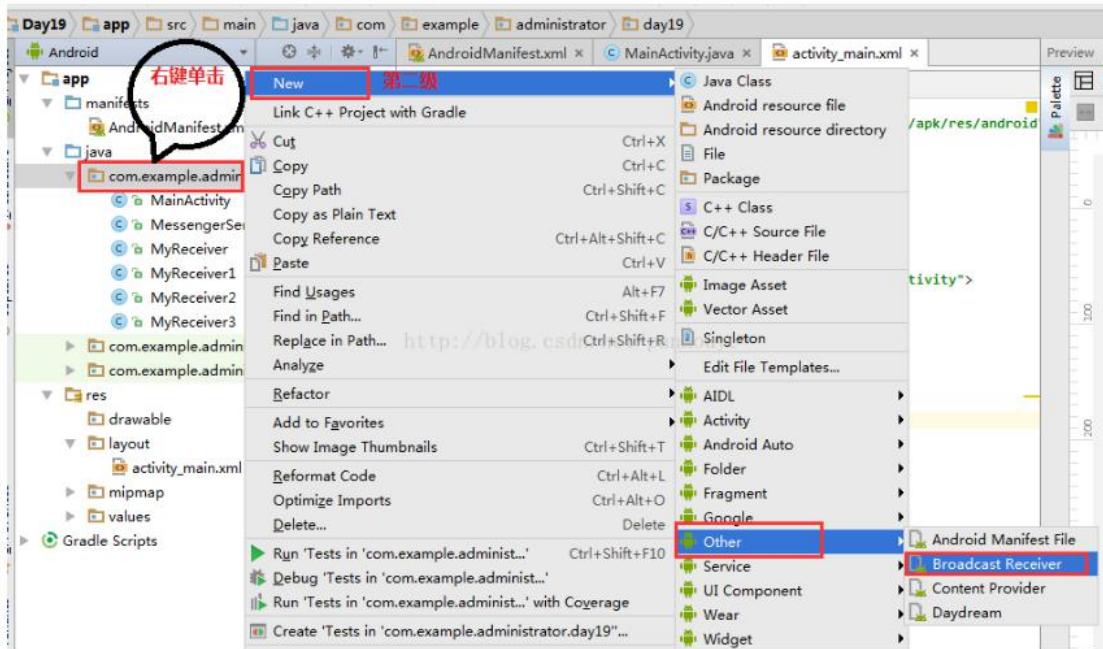
建立方法代码中做了详细注释，有不明白的地方请留言讨论。

静态注册：

效果如下：



静态注册建立第一步，新建 BroadcastReceiver，见下图：



通过以上步骤，生成 MyReceiver.java 文件：

[java] [view plain](#) [copy](#)

1. **import** android.content.BroadcastReceiver;
2. **import** android.content.Context;
3. **import** android.content.Intent;
4. **import** android.view.Gravity;
5. **import** android.widget.Toast;
- 6.
7. **public class** MyReceiver **extends** BroadcastReceiver {
8. **public** MyReceiver() {
9. }
10. **@Override**

```
11. public void onReceive(Context context, Intent intent) {  
12.     Toast t = Toast.makeText(context, "静态广播 :  
13.         "+intent.getStringExtra("info"), Toast.LENGTH_SHORT);  
14.         t.setGravity(Gravity.TOP,0,0);  
15.     }  
16.}
```

生成 MyReceiver.java 的同时 ,修改 AndroidManifest.xml 配置文件中的代码 :

[java] [view](#) [plain](#) [copy](#)

```
1. <?xml version="1.0" encoding="utf-8"?>  
2. <manifest xmlns:android="http://schemas.android.com/apk/  
    res/android"  
3.     package="com.example.administrator.day19">  
4.     <uses-permission android:name="android.permission.PROCE  
        SS_OUTGOING_CALLS"/>  
5.     <application  
6.         android:allowBackup="true"  
7.         android:icon="@mipmap/ic_launcher"
```

```
8.      android:label="@string/app_name"  
9.      android:supportsRtl="true"  
10.     android:theme="@style/AppTheme">  
11.     <activity android:name=".MainActivity">  
12.         <intent-filter>  
13.             <action android:name="android.intent.action.MAIN" />  
14.  
15.             <category android:name="android.intent.category.LAUNCHER" />  
16.         </intent-filter>  
17.     </activity>  
18. //生成的 receiver 配置文件  
19.     <receiver  
20.         android:name=".MyReceiver"  
21.         android:enabled="true"  
22.         android:exported="true">  
23.         <intent-filter>  
24.             //自定义 Action
```

```
25.      <action android:name="MLY" />  
  
26.      </intent-filter>  
  
27.      </receiver>  
  
28.  </application>  
  
29.</manifest>
```

最后在 MainActivity.java 文件中添加按钮点击事件，如下：

[java] [view](#) [plain](#) [copy](#)

```
1. import android.content.BroadcastReceiver;  
  
2. import android.content.Context;  
  
3. import android.content.Intent;  
  
4. import android.content.IntentFilter;  
  
5. import android.support.v7.app.AppCompatActivity;  
  
6. import android.os.Bundle;  
  
7. import android.view.Gravity;  
  
8. import android.view.View;  
  
9. import android.widget.Toast;  
  
10.  
  
11. public class MainActivity extends AppCompatActivity {
```

```
12.    DynamicReceiver dynamicReceiver;  
  
13.    @Override  
  
14.    protected void onCreate(Bundle savedInstanceState) {  
  
15.        super.onCreate(savedInstanceState);  
  
16.        setContentView(R.layout.activity_main);  
  
17.    }  
  
18.    //静态广播点击  
  
19.    public void send(View v){  
  
20.        Intent intent = new Intent();  
  
21.        intent.setAction("MLY");  
  
22.        intent.putExtra("info","panhouye");  
  
23.        sendBroadcast(intent);  
  
24.    }  
  
25.}
```

至此，两种注册方式的实现代码演示完毕，欢迎探讨。

二、插入 BroadcastReceiver 的两种常用类型

(1) Normalbroadcasts : 默认广播

发送一个默认广播使用 Context.sendBroadcast() 方法，普通广播对于多个接收者来说是完全异步的，通常每个接收者都无需等待即可以接收到广播，接收者相互之间不会有影响。对于这种广播，接收者无法终止广播，即无法阻止其他接收者的接收动作。

(2) orderedbroadcasts : 有序广播

发送一个有序广播使用 Context.sendOrderedBroadcast() 方法，有序广播比较特殊，它每次只发送到优先级较高的接收者那里，然后由优先级高的接受者再传播到优先级低的接收者那里，优先级高的接收者有能力终止这个广播。

发送有序广播：sendOrderedBroadcast()

在注册广播中的<intent-filter>中使用 android:priority 属性。这个属性的范围在 -1000 到 1000，数值越大，优先级越高。在广播接收器中使用 setResultExtras 方法将一个 Bundle 对象设置为结果集对象，传递到下一个接收者那里，这样优先级低的接收者可以用 getResultExtras 获取到最新的经过处理的信息集合。使用 sendOrderedBroadcast 方法发送有序广播时，需要一个权限参数，如果为 null 则表示不要求接收者声明指定的权限，如果不为 null 则表示接收者若要接收此广播，需声明指定权限。这样做是从安全角度考虑的，例如系统的短信就是有序广播的形式，一个应用可能是具有拦截垃圾短信的功能，当短信到来时它可以先接受到短信广播，必要时终止广播传递，这样的软件就必须声明接收短信的权限。

三、默认广播下两种注册方式的比较

(1) 两种注册方式均不设置优先级

这里将动态与静态两种注册的广播触发集中在一个按钮上，显示效果如下(未设置优先级的情况下，先动态后静态)：



这里同样不演示按钮布局文件，以及静态注册涉及 AndroidManifest.xml 和 MyReceiver.java 文件。直接展示 MainActicity.java 的实现代码：

[java] [view](#) [plain](#) [copy](#)

1. **import** android.content.BroadcastReceiver;
2. **import** android.content.Context;
3. **import** android.content.Intent;
4. **import** android.content.IntentFilter;
5. **import** android.support.v7.app.AppCompatActivity;
6. **import** android.os.Bundle;
7. **import** android.view.Gravity;
8. **import** android.view.View;
9. **import** android.widget.Toast;

```
10.

11.public class MainActivity extends AppCompatActivity {

12.    DynamicReceiver dynamicReceiver;

13.    @Override

14.    protected void onCreate(Bundle savedInstanceState) {

15.        super.onCreate(savedInstanceState);

16.        setContentView(R.layout.activity_main);

17.        IntentFilter filter = new IntentFilter();

18.        filter.addAction("panhouye");

19.        dynamicReceiver = new DynamicReceiver();

20.        registerReceiver(dynamicReceiver,filter);

21.    }

22.    //静态广播点击

23.    public void send(View v){

24.        Intent intent = new Intent();

25.        //设置与动态相同的 Action , 方便同时触发静态与动态

26.        intent.setAction("panhouye");

27.        intent.putExtra("info","潘侯爷");

28.        sendBroadcast(intent);//默认广播
```

```
29.    }

30.    @Override

31.    protected void onPause() {

32.        super.onPause();

33.        unregisterReceiver(new MyReceiver());

34.    }

35.    class DynamicReceiver extends BroadcastReceiver{

36.        @Override

37.        public void onReceive(Context context, Intent intent) {

38.            Toast t = Toast.makeText(context,"动态广播 :

39.                "+ intent.getStringExtra("info"), Toast.LENGTH_SHORT);

40.            t.setGravity(Gravity.TOP,0,0);

41.            t.show();

42.        }

43.    }
```

(2) 将动态优先级设置为最低-1000 , 静态优先级设置为最高 1000

显示效果如下 (动态仍先于静态被接收到) :



MainActivity 中动态优先级设置如下：

[java] view plain copy

```
1. protected void onCreate(Bundle savedInstanceState) {  
2.     super.onCreate(savedInstanceState);  
3.     setContentView(R.layout.activity_main);  
4.     IntentFilter filter = new IntentFilter();  
5.     filter.addAction("panhouye");  
6.     filter.setPriority(-1000);//设置动态优先级  
7.     dynamicReceiver = new DynamicReceiver();  
8.     registerReceiver(dynamicReceiver,filter);  
9. }
```

AndroidManifest.xml 中静态优先级设置如下：

[java] view plain copy

```
1. <receiver
```

```
2.    android:name=".MyReceiver"

3.    android:enabled="true"

4.    android:exported="true">

5.    //设置静态优先级

6.    <intent-filter android:priority="1000">

7.        <action android:name="panhouye" />

8.    </intent-filter>

9. </receiver>
```

四、在有序广播下两种注册方式比较

静态广播 1 (优先级为 200) , 静态广播 2 (优先级为 300) , 静态广播 3 (优先级为 400) , 静态广播优先级为 (-100) , 动态广播优先级为 0。显示效果如下：



出现顺序由优先级决定，由高到低分别为静态 3-静态 2-静态 1-动-静态。 (这里参照前文代码)

五、接受打电话的广播，程序运行与结束时比较两种注册方式

本次比较采用比对 Log 的方式对两种注册方式进行比较，在 MainActivity.java 中会插入 Activity 全部生命周期用于检测 Log 分析。

AndroidManifest.xml 配置文件代码如下：

[java] [view](#) [plain](#) [copy](#)

```
1. <manifest xmlns:android="http://schemas.android.com/apk/  
    res/android"  
  
2.     package="com.example.administrator.test19">  
  
3.     //添加拨打电话权限  
  
4.     <uses-permission android:name="android.permission.PROCE  
        SS_OUTGOING_CALLS"/>  
  
5.     <application  
  
6.         android:allowBackup="true"  
  
7.         android:icon="@mipmap/ic_launcher"  
  
8.         android:label="@string/app_name"  
  
9.         android:supportsRtl="true"
```

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

11.    <activity android:name=".MainActivity" >

12.        <intent-filter>

13.            <action android:name="android.intent.action.MAIN" />

14.

15.            <category android:name="android.intent.category.LAUNCHER" />

16.        </intent-filter>

17.    </activity>

18.    <receiver

19.        android:name=".StaticReceiver"

20.        android:enabled="true"

21.        android:exported="true">

22.        <intent-filter>

23.            //设置打电话对应的 action

24.            <action android:name="android.intent.action.NEW_OUTGOING_CALL" />

25.        </intent-filter>
```

26. </receiver>

27. </application>

28.</manifest>

MainActivity.java 中实现代码(动态注册将解除注册放在 `onDestory` 方法内是因为在真机测试过程中拨打电话 , 需要返回主页面 , 而此操作会造成 Activity 处于 `onStop` 状态 , 若放在 `onPause` 中 , 将无法在程序运行时启用动态注册接受广播。真实环境下建议在 `onpause` 下解除注册 , 尽早释放内存 , 避免内存溢出) :

[java] view plain copy

1. **import** android.content.BroadcastReceiver;
2. **import** android.content.Context;
3. **import** android.content.Intent;
4. **import** android.content.IntentFilter;
5. **import** android.support.v7.app.AppCompatActivity;
6. **import** android.os.Bundle;
7. **import** android.util.Log;
- 8.
9. **public class** MainActivity **extends** AppCompatActivity {

```
10.    DynamicReceiver dynamicReceiver;//声明动态注册广播接收  
11.    @Override  
12.    protected void onCreate(Bundle savedInstanceState) {  
13.        super.onCreate(savedInstanceState);  
14.        setContentView(R.layout.activity_main);  
15.        IntentFilter filter = new IntentFilter();  
16.        filter.addAction("android.intent.action.NEW_OUTGOING  
_CALL");  
17.        dynamicReceiver = new DynamicReceiver();  
18.        registerReceiver(dynamicReceiver,filter);  
19.        Log.i("Tag","Activity-onCreate");  
20.    }  
21.    @Override  
22.    protected void onStart() {  
23.        super.onStart();  
24.        Log.i("Tag","Activity-onStart");  
25.    }  
26.    @Override
```

```
27. protected void onResume() {  
  
28.     super.onResume();  
  
29.     Log.i("Tag","Activity-onResume");  
  
30. }  
  
31. @Override  
  
32. protected void onPause() {  
  
33.     super.onPause();  
  
34.     Log.i("Tag","Activity-onPause");  
  
35. }  
  
36. @Override  
  
37. protected void onStop() {  
  
38.     super.onPause();  
  
39.     Log.i("Tag","Activity-onStop");  
  
40. }  
  
41. @Override  
  
42. protected void onDestroy() {  
  
43.     super.onDestroy();  
  
44.     Log.i("Tag","Activity-onDestroy");  
  
45.     unregisterReceiver(dynamicReceiver);
```

```
46. }

47. class DynamicReceiver extends BroadcastReceiver{

48.     @Override

49.     public void onReceive(Context context, Intent intent) {

50.         Log.i("Tag", "动态注册广播接收到您正在拨打电话

51.         "+getResultData());

52.     }

53.}
```

StaticReceiver.java 中实现代码：

[\[java\]](#) [view](#) [plain](#) [copy](#)

1. import android.content.BroadcastReceiver;
2. import android.content.Context;
3. import android.content.Intent;
4. import android.util.Log;
5. public class StaticReceiver extends BroadcastReceiver {
6. public StaticReceiver() {
7. }

```

8.    @Override

9.    public void onReceive(Context context, Intent intent) {

10.       Log.i("Tag", "静态注册广播接收到您正在拨打电话

" + getResultData());

11.   }

12.}

```

(1) 在未退出 Activity 时 , 拨打电话 , Log 如下 :



由 Log 可知在未退出 Activity 是 , 两种方式均可接受到广播。

(2) 在退出 Activity 时 , 拨打电话 , Log 如下 (即便不解除注册 , 动态仍无法接受到广播) :



在退出程序 (Activity) 时 , 只有静态注册方式可以接受到广播。

六、总结两种注册方式特点

广播接收器注册一共有两种形式：静态注册和动态注册。

两者及其接收广播的区别：

(1) 动态注册广播不是常驻型广播，也就是说广播跟随 Activity 的生命周期。

注意在 Activity 结束前，移除广播接收器。

静态注册是常驻型，也就是说当应用程序关闭后，如果有信息广播来，程序也会被系统调用自动运行。

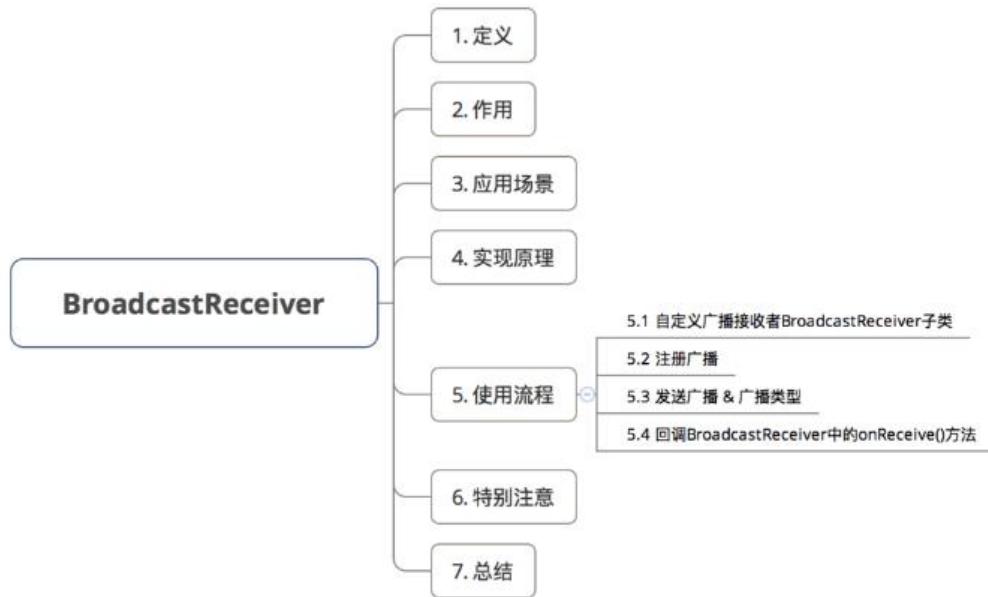
(2) 当广播为有序广播时：优先级高的先接收（不分静态和动态）。同优先级的广播接收器，动态优先于静态

(3) 同优先级的同类广播接收器，静态：先扫描的优先于后扫描的，动态：先注册的优先于后注册的。

(4) 当广播为默认广播时：无视优先级，动态广播接收器优先于静态广播接收器。同优先级的同类广播接收器，静态：先扫描的优先于后扫描的，动态：先注册的优先于后注册的。

文章、Android 四大组件：**BroadcastReceiver** 史上最全面解析

目录



1. 定义

即 广播，是一个全局的监听器，属于 四大组件之一

广播分为两个角色：广播发送者、广播接收者

2. 作用

监听 / 接收 应用 发出的广播消息，并 做出响应

3. 应用场景

不同组件间的通信（含：应用内 / 不同应用之间）

多线程通信

与 系统在特定情况下的通信

如：电话呼入时、网络可用时

4. 实现原理

4.1 采用的模型

- 中的广播使用了设计模式中的**观察者模式**：基于消息的发布 / 订阅事件模型

因此，Android 将广播的**发送者 和 接收者 解耦**，使得系统方便集成，更容易扩展

4.2 模型讲解

-

模型中有 3 个角色：

- 消息订阅者（广播接收者）
- 消息发布者（广播发布者）
- 消息中心（**AMS**，即 **处理中心**）

示意图 & 原理如下



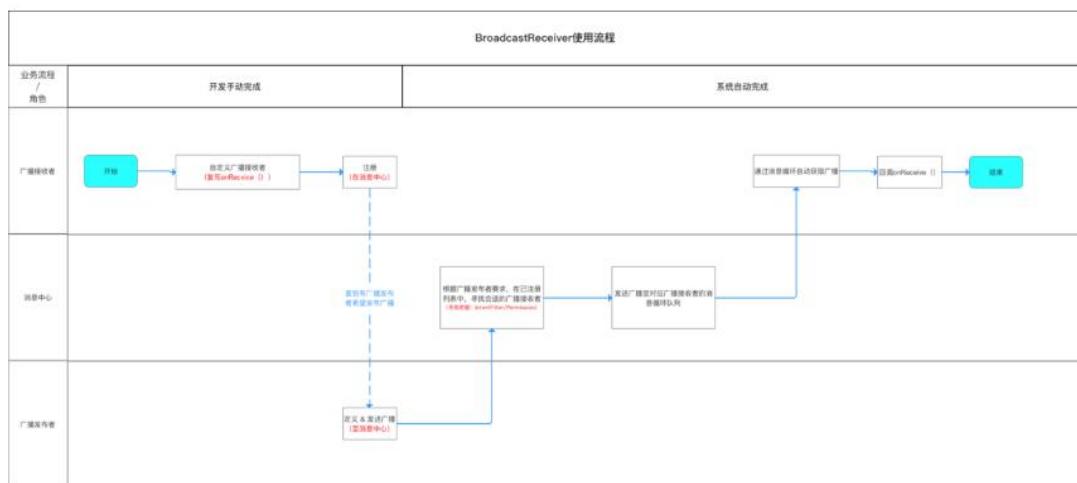
原理描述

- 广播接收者 通过 Binder机制在 AMS 注册
- 广播发送者 通过 Binder 机制向 AMS 发送广播
- AMS 根据 广播发送者 要求, 在已注册列表中, 寻找合适的广播接收者
(寻找依据: IntentFilter / Permission)
- AMS将广播发送到合适的广播接收者相应的消息循环队列中
- 广播接收者通过 消息循环 拿到此广播, 并回调 onReceive()

特别注意: 广播发送者 和 广播接收者的执行 是 *异步的, 即 广播发送者 不会关心有无接收者接收 & 也不确定接收者何时才能接收到

5. 使用流程

使用流程如下:



- 下面，我将一步步介绍如何使用

即上图中的 **开发者手动完成部分**

5.1 自定义广播接收者 BroadcastReceiver

- 继承 **基类**
- 必须复写抽象方法 **方法**

- 广播接收器接收到相应广播后，会自动回调 `onReceive()` 方法
- 一般情况下，`onReceive` 方法会涉及 与 其他组件之间的交互，如发送 `Notification`、启动 `Service` 等
- 默认情况下，广播接收器运行在 `UI` 线程，因此，`onReceive()` 方法不能执行耗时操作，否则将导致 `ANR`

代码范例

mBroadcastReceiver.java

// 继承 BroadcastReceivre 基类

public class mBroadcastReceiver extends BroadcastReceiver {

// 复写 onReceive()方法

// 接收到广播后，则自动调用该方法

@Override

public void onReceive(Context context, Intent intent) {

//写入接收广播后的操作

}

}

5.2 广播接收器注册

注册的方式分为两种：静态注册、动态注册

5.2.1 静态注册

注册方式：在 AndroidManifest.xml 里通过****标签声明

属性说明：

<receiver

android:enabled=["true" | "false"]

//此 broadcastReceiver 能否接收其他 App 的发出的广播

//默认值是由 receiver 中有无 intent-filter 决定的：如果有 intent-filter， 默认值为 true，否则为 false

android:exported=["true" | "false"]

android:icon="drawable resource"

android:label="string resource"

//继承 BroadcastReceiver 子类的类名

android:name=".mBroadcastReceiver"

//具有相应权限的广播发送者发送的广播才能被此 BroadcastReceiver 所接收；

android:permission="string"

//BroadcastReceiver 运行所处的进程

//默认为 app 的进程，可以指定独立的进程

//注：Android 四大基本组件都可以通过此属性指定自己的独立进程

android:process="string" >

```
//用于指定此广播接收器将接收的广播类型  
  
//本示例中给出的是用于接收网络状态改变时发出的广播  
  
<intent-filter>  
  
<action android:name="android.net.conn.CONNECTIVITY_CHANGE" />  
  
</intent-filter>  
  
</receiver>
```

注册示例

```
<receiver  
  
    //此广播接收者类是 mBroadcastReceiver  
  
    android:name=".mBroadcastReceiver" >  
  
    //用于接收网络状态改变时发出的广播  
  
    <intent-filter>  
  
        <action  
  
            android:name="android.net.conn.CONNECTIVITY_CHANGE" />  
  
    </intent-filter>
```

```
</receiver>
```

当此 首次启动时 ,系统会**自动**实例化 类 ,并注册到系统中。

5.2.2 动态注册

注册方式 : 在代码中调用 **()** 方法

具体代码如下 :

```
// 选择在 Activity 生命周期方法中的 onResume() 中注册
```

```
@Override
```

```
protected void onResume(){
```

```
    super.onResume();
```

```
// 1. 实例化 BroadcastReceiver 子类 & IntentFilter
```

```
    mBroadcastReceiver mBroadcastReceiver = new  
    mBroadcastReceiver();
```

```
    IntentFilter intentFilter = new IntentFilter();
```

```
// 2. 设置接收广播的类型
```

```
intentFilter.addAction(android.net.conn.CONNECTIVITY_CHANGE);

// 3. 动态注册：调用 Context 的 registerReceiver( ) 方法

registerReceiver(mBroadcastReceiver, intentFilter);

}

// 注册广播后，要在相应位置记得销毁广播

// 即在 onPause() 中 unregisterReceiver(mBroadcastReceiver)

// 当此 Activity 实例化时，会动态将 MyBroadcastReceiver 注册到系统中

// 当此 Activity 销毁时，动态注册的 MyBroadcastReceiver 将不再接收到相
应的广播。

@Override

protected void onPause() {

    super.onPause();

    // 销毁在 onResume() 方法中的广播
```

```
unregisterReceiver(mBroadcastReceiver);  
  
}  
  
}
```

特别注意

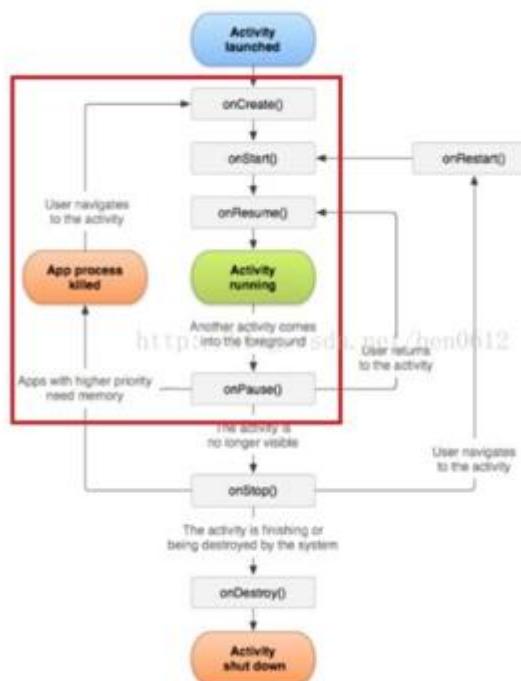
动态广播最好在 **onStart()** 的 **onReceive()** 注册、**onStop()** 注销。

原因：

- 对于动态广播，有注册就必然得有注销，否则会导致**内存泄露**

重复注册、重复注销也不允许

1. 生命周期如下



Activity 生命周期的方法是成对出现的：

`onCreate()` & `onDestory()`

`onStart() & onStop()`

`onResume() & onPause()`

在 `onResume()` 注册、`onPause()` 注销是因为 `onPause()` 在 App 死亡前一定会被执行，从而保证广播在 App 死亡前一定会被注销，从而防止内存泄露。

1. 不在 `onCreate()` & `onDestory()` 或 `onStart()` & `onStop()` 注册、注销是因为：

当系统因为内存不足（优先级更高的应用需要内存，请看上图红框）

要回收 Activity 占用的资源时，Activity 在执行完 `onPause()` 方法后

就会被销毁，有些生命周期方法 `onStop()`, `onDestory()` 就不会执行。

当再回到此 Activity 时，是从 `onCreate` 方法开始执行。

2. 假设我们将广播的注销放在 `onStop()`, `onDestory()` 方法里的话，有

可能在 Activity 被销毁后还未执行 `onStop()`, `onDestory()` 方法，即

广播仍还未注销，从而导致内存泄露。

3. 但是，`onPause()` 一定会被执行，从而保证了广播在 App 死亡前一定

会被注销，从而防止内存泄露。

5.2.3 两种注册方式的区别

注册方式	区别		
	使用方式	特点	应用场景
静态注册 <small>(常驻广播)</small>	在 <code>AndroidManifest.xml</code> 里通过 <code><receive></code> 标签声明	<ul style="list-style-type: none">常驻，不受任何组件的生命周期影响 (应用程序关闭后，如果有信息广播来，程序依旧会被系统调用)缺点：耗电、占内存	需要时刻监听广播
动态注册 <small>(非常驻广播)</small>	在代码中调用 <code>Context.registerReceiver()</code> 方法	非常驻，灵活，跟随组件的生命周期变化 (组件结束=广播结束，在组件结束前，必须移除广播接收器)	需要特定时刻监听广播

5.3 广播发送者向 AMS 发送广播

5.3.1 广播的发送

广播是用“意图 ()”标识

定义广播的本质 = 定义广播所具备的“意图 ()”

广播发送 = 广播发送者 将此广播的“意图 ()”通过

sendBroadcast()方法发送出去

5.3.2 广播的类型

广播的类型主要分为 5 类：

- 普通广播 ()
- 系统广播 ()
- 有序广播 ()
- 粘性广播 ()
- App 应用内广播 ()

具体说明如下：

1. 普通广播 (Normal Broadcast)

即 开发者自身定义 的广播 (最常用)。发送广播使用如下：

```
Intent intent = new Intent();
```

```
//对应 BroadcastReceiver 中 intentFilter 的 action
```

```
intent.setAction(BROADCAST_ACTION);
```

```
//发送广播
```

```
sendBroadcast(intent);
```

若被注册了的广播接收者中注册时 `IntentFilter.addAction(action)` 的 `IntentFilter` 与上述匹配，则会接收此广播（即进行回调 `onReceive(Context context, Intent intent)`）。如下 `<receiver>` 则会接收上述广播

```
<receiver  
    //此广播接收者类是 mBroadcastReceiver  
    android:name=".mBroadcastReceiver" >  
  
    //用于接收网络状态改变时发出的广播  
  
    <intent-filter>  
        <action android:name="BROADCAST_ACTION" />  
  
    </intent-filter>  
  
</receiver>
```

- 若发送广播有相应权限，那么广播接收者也需要相应权限

2. 系统广播 (System Broadcast)

- Android 中内置了多个系统广播：只要涉及到手机的基本操作（如开机、网络状态变化、拍照等等），都会发出相应的广播
- 每个广播都有特定的 Intent - Filter (包括具体的 action)，Android 系统广播 action 如下：

系统操作	action
监听网络变化	android.net.conn.CONNECTIVITY_CHANGE
关闭或打开飞行模式	Intent.ACTION_AIRPLANE_MODE_CHANGED
充电时或电量发生变化	Intent.ACTION_BATTERY_CHANGED
电池电量低	Intent.ACTION_BATTERY_LOW
电池电量充足 (即从电量低变化到饱满时会发出广播)	Intent.ACTION_BATTERY_OKAY
系统启动完成后(仅广播一次)	Intent.ACTION_BOOT_COMPLETED
按下照相时的拍照按键(硬件按键)时	Intent.ACTION_CAMERA_BUTTON
屏幕锁屏	Intent.ACTION_CLOSE_SYSTEM_DIALOGS
设备当前设置被改变时(界面语言、设备方向等)	Intent.ACTION_CONFIGURATION_CHANGED
插入耳机时	Intent.ACTION_HEADSET_PLUG
未正确移除SD卡但已取出来时(正确移除方法:设置-SD卡和设备内存-卸载SD卡)	Intent.ACTION_MEDIA_BAD_REMOVAL
插入外部储存装置 (如SD卡)	Intent.ACTION_MEDIA_CHECKING
成功安装APK	Intent.ACTION_PACKAGE_ADDED
成功删除APK	Intent.ACTION_PACKAGE_REMOVED
重启设备	Intent.ACTION_REBOOT
屏幕被关闭	Intent.ACTION_SCREEN_OFF
屏幕被打开	Intent.ACTION_SCREEN_ON
关闭系统时	Intent.ACTION_SHUTDOWN
重启设备	Intent.ACTION_REBOOT

注：当使用系统广播时，只需要在注册广播接收者时定义相关的 action 即可，
并不需要手动发送广播，当系统有相关操作时会自动进行系统广播

3. 有序广播 (Ordered Broadcast)

定义

发送出去的广播被广播接收者按照先后顺序接收

有序是针对广播接收者而言的

-

广播接受者接收广播的顺序规则（同时面向静态和动态注册的广播接受者）

- - 按照 Priority 属性值从大-小排序；
 - Priority 属性相同者，动态注册的广播优先；
-

特点

- - 接收广播按顺序接收
 - 先接收的广播接收者可以对广播进行截断，即后接收的广播接收者不再接收到此广播；
 - 先接收的广播接收者可以对广播进行修改，那么后接收的广播接收者将接收到被修改后的广播
-

具体使用

有序广播的使用过程与普通广播非常类似，差异仅在于广播的发送方式：

```
sendOrderedBroadcast(intent);
```

4. App 应用内广播 (Local Broadcast)

背景

Android 中的广播可以跨 App 直接通信 (exported 对于有 intent-filter 情况下默认值为 true)

冲突

可能出现的问题：

- 其他 App 针对性发出与当前 App intent-filter 相匹配的广播，由此导致当前 App 不断接收广播并处理；
- 其他 App 注册与当前 App 一致的 intent-filter 用于接收广播，即会出现安全性 & 效率性的问题。

解决方案

使用 App 应用内广播 (Local Broadcast)

1. App 应用内广播可理解为一种局部广播，广播的发送者和接收者都同属于一个App。
2. 相比于全局广播（普通广播），App 应用内广播优势体现在：安全性高 & 效率高

-

具体使用 1 - 将全局广播设置成局部广播

- . 注册广播时将 `exported` 属性设置为 `false` , 使得非本 App 内部发出的此广播不被接收 ;
- . 在广播发送和接收时 , 增设相应权限 `permission` , 用于权限验证 ;
- . 发送广播时指定该广播接收器所在的包名 , 此广播将只会发送到此包中的 App 内与之相匹配的有效广播接收器中。

通过`intent.setPackage(packageName)`指定报名

具体使用 2 - 使用封装好的 LocalBroadcastManager 类

使用方式上与全局广播几乎相同 , 只是注册/取消注册广播接收器和发送广播时将参数的 `context` 变成了 `LocalBroadcastManager` 的单一实例

注 : 对于 `LocalBroadcastManager` 方式发送的应用内广播 , 只能通过

`LocalBroadcastManager` 动态注册 , 不能静态注册

```
//注册应用内广播接收器
```

```
//步骤 1 : 实例化 BroadcastReceiver 子类 & IntentFilter
```

```
mBroadcastReceiver
```

```
mBroadcastReceiver = new mBroadcastReceiver();
```

```
IntentFilter intentFilter = new IntentFilter();
```

```
//步骤2：实例化 LocalBroadcastManager 的实例
```

```
localBroadcastManager = LocalBroadcastManager.getInstance(this);
```

//步骤3：设置接收广播的类型

```
intentFilter.addAction(android.net.conn.CONNECTIVITY_CHANGE);
```

```
//步骤4：调用LocalBroadcastManager单一实例的registerReceiver()方法进行动态注册
```

```
localBroadcastManager.registerReceiver(mBroadcastReceiver,  
intentFilter);
```

//取消注册应用内广播接收器

```
localBroadcastManager.unregisterReceiver(mBroadcastReceiver);
```

//发送应用内广播

```
Intent intent = new Intent();
```

```
intent.setAction(BROADCAST_ACTION);  
  
localBroadcastManager.sendBroadcast(intent);
```

5. 粘性广播 (Sticky Broadcast)

由于在 Android5.0 & API 21 中已经失效，所以不建议使用，在这里也不作过多的总结。

6. 特别注意

对于不同注册方式的广播接收器回调 OnReceive (Context context , Intent intent) 中的 context 返回值是不一样的：

对于静态注册（全局+应用内广播），回调 onReceive(context, intent) 中的 context 返回值是：ReceiverRestrictedContext；

对于全局广播的动态注册，回调 onReceive(context, intent) 中的 context 返回值是：Activity Context；

对于应用内广播的动态注册（ LocalBroadcastManager 方式），回调 onReceive(context, intent) 中的 context 返回值是：Application Context。

对于应用内广播的动态注册（非 LocalBroadcastManager 方式），回调 onReceive(context, intent) 中的 context 返回值是：Activity Context；

文章、安卓广播的底层实现原理

静态广播的注册

静态广播是通过 PackageManagerService 在启动的时候扫描已安装的应用去注册的。

在 PackageManagerService 的构造方法中,会去扫描应用安装目录,顺序是先扫描系统应用安装目录再扫描第三方应用安装目录。

PackageManagerService.scanDirLI 就是用于扫描目录的方法,由于代码比较少,这里我们直接把它贴了出来:

```
private void scanDirLI(File dir, int flags, int scanMode, long currentTi  
me) {  
  
    String[] files = dir.list();  
  
    if (files == null) {  
  
        return;  
  
    }  
  
    int i;  
  
    for (i=0; i<files.length; i++) {  
  
        File file = new File(dir, files[i]);  
  
        if (!isPackageFilename(files[i])) {  
  
            continue;  
  
        }  
  
        PackageParser.Package pkg = scanPackageLI(file,  
  
            flags|PackageParser.PARSE_MUST_BE_APK, scanMode, currentTi  
me, null);  
  
        if (pkg == null && (flags & PackageParser.PARSE_IS_SYSTEM) == 0 &&  
  
            mLastScanError == PackageManager.INSTALL_FAILED_INVALID_AP  
K) {
```

```
        file.delete();

    }

}

private static final boolean isPackageFilename(String name) {

    return name != null && name.endsWith(".apk");
}
```

可以看到, 它通过 File.list 方法列出目录下的所有后缀为". apk"的文件传给 scanPackageLI 去处理.

而 scanPackageLI(File scanFile, int parseFlags, int scanMode, long currentTime, UserHandle user) 内部会调用它的重载方法
scanPackageLI(PackageParser.Package pkg, int parseFlags, int scanMode, long currentTime, UserHandle user):

```
private PackageParser.Package scanPackageLI(File scanFile,int parseFlags,
int scanMode, long currentTime, UserHandle user) {

    ...

    final PackageParser.Package pkg = pp.parsePackage(scanFile,scanPath, m
Metrics, parseFlags);

    ...

    PackageParser.Package scannedPkg = scanPackageLI(pkg, parseFlags, scan
Mode | SCAN_UPDATE_SIGNATURE, currentTime, user);

    ...
}
```

在这个 scanPackageLII 里面会解析 Package 并且将 AndroidManifest.xml 中注册的 BroadcastReceiver 保存下来:

```
...

N = pkg.receivers.size();

r = null;for (i=0; i<N; i++) {

    PackageParser.Activity a = pkg.receivers.get(i);

    a.info.processName = fixProcessName(pkg.applicationInfo.processName,
```

```
a.info.processName, pkg.applicationInfo.uid);  
  
mReceivers.addActivity(a, "receiver");  
  
...}...
```

所以从上面获取静态广播的流程可以看出来:系统应用的广播先于第三方应用的广播注册,而安装在同一个目录下的应用的静态广播的注册顺序是按照 File.list 列出来的 apk 的顺序注册的.他们的注册顺序就决定了它们接收广播的顺序.

通过静态广播的注册流程,我们已经将静态广播注册到了 PackageManagerService 的 mReceivers 中,而我们可以使用 PackageManagerService.queryIntentReceivers 方法查询 intent 对应的静态广播

```
public List<ResolveInfo> queryIntentReceivers(Intent intent, String resolvedType, int flags, int userId) {  
  
    if (!sUserManager.exists(userId)) return Collections.emptyList();  
  
    ComponentName comp = intent.getComponent();  
  
    if (comp == null) {  
  
        if (intent.getSelector() != null) {  
  
            intent = intent.getSelector();  
  
            comp = intent.getComponent();  
  
        }  
  
    }  
  
    if (comp != null) {  
  
        List<ResolveInfo> list = new ArrayList<ResolveInfo>(1);  
  
        ActivityInfo ai = getReceiverInfo(comp, flags, userId);  
  
        if (ai != null) {  
  
            ResolveInfo ri = new ResolveInfo();  
  
            ri.activityInfo = ai;  
  
            list.add(ri);  
        }  
    }  
}
```

```
    }

    return list;
}

}

synchronized (mPackages) {

    String pkgName = intent.getPackage();

    if (pkgName == null) {

        return mReceivers.queryIntent(intent, resolvedType, flags, userId);
    }

    final PackageParser.Package pkg = mPackages.get(pkgName);

    if (pkg != null) {

        return mReceivers.queryIntentForPackage(intent, resolvedType,
flags, pkg.receivers,
userId);
    }

    return null;
}
}
```

动态广播的注册

我们调用 Context.registerReceiver 最后会调到

ActivityManagerService.registerReceiver:

```
public Intent registerReceiver(IApplicationThread caller, String callerP
ackage, IIIntentReceiver receiver, IntentFilter filter, String permission,
int userId) {
```

```
...
ReceiverList rl = (ReceiverList)mRegisteredReceivers.get(receiver.asBinder());
...
BroadcastFilter bf = new BroadcastFilter(filter, rl, callerPackage, permission, callingUid, userId);
...
mReceiverResolver.addFilter(bf);
...}
```

所以通过 mReceiverResolver.queryIntent 就能获得 intent 对应的动态广播了.

发送广播

```
ContextImpl.sendBroadcast 中会调用
ActivityManagerNative.getDefault().broadcastIntent()
public void sendBroadcast(Intent intent) {

    warnIfCallingFromSystemProcess();

    String resolvedType = intent.resolveTypeIfNeeded(getApplicationContext());
    try {

        intent.prepareToLeaveProcess();

        ActivityManagerNative.getDefault().broadcastIntent(
            mMainThread.getApplicationThread(), intent, resolvedType, null,
            Activity.RESULT_OK, null, null, null, AppOpsManager.OP_NONE,
            false, false, getUserId());

    } catch (RemoteException e) {

    }
}
```

实际是调用 ActivityManagerService.broadcastIntent:

```
public final int broadcastIntent(IAplicationThread caller,
                                 Intent intent, String resolvedType, IIntentReceiver resultTo,
                                 int resultCode, String resultData, Bundle map,
                                 String requiredPermission, int appOp, boolean serialized, boolean
                                 sticky, int userId) {
    enforceNotIsolatedCaller("broadcastIntent");
    synchronized(this) {
        intent = verifyBroadcastLocked(intent);
        final ProcessRecord callerApp = getRecordForAppLocked(caller);
        final int callingPid = Binder.getCallingPid();
        final int callingUid = Binder.getCallingUid();
        final long origId = Binder.clearCallingIdentity();
        int res = broadcastIntentLocked(callerApp,
                                         callerApp != null ? callerApp.info.packageName : null,
                                         intent, resolvedType, resultTo,
                                         resultCode, resultData, map, requiredPermission, appOp, s
                                         erialized, sticky,
                                         callingPid, callingUid, userId);
        Binder.restoreCallingIdentity(origId);
        return res;
    }
}
```

ActivityManagerService.broadcastIntent 中又会调用
ActivityManagerService.broadcastIntentLocked, 而 broadcastIntentLocked
中的关键代码如下:

```
// 静态广播 List<BroadcastFilter> registeredReceivers = null; // 动态广播 List<BroadcastFilter> registeredReceivers = null; if ((intent.getFlags() & Intent.FLAG_RECEIVER_REGISTERED_ONLY) == 0) { // 查询静态广播 registeredReceivers = collectReceiverComponents(intent, resolvedType, users); } if (intent.getComponent() == null) { // 查询动态广播 registeredReceivers = mReceiverResolver.queryIntent(intent, resolvedType, false, userId); } final boolean replacePending = (intent.getFlags() & Intent.FLAG_RECEIVER_REPLACE_PENDING) != 0; int NR = registeredReceivers != null ? registeredReceivers.size() : 0; if (!ordered && NR > 0) { final BroadcastQueue queue = broadcastQueueForIntent(intent); BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp, callerPackage, callingPid, callingUid, resolvedType, requiredPermission, appOp, registeredReceivers, resultTo, resultCode, resultData, map, ordered, sticky, false, userId); final boolean replaced = replacePending && queue.replaceParallelBroadcastLocked(r); if (!replaced) { // 发送动态广播 queue.enqueueParallelBroadcastLocked(r); }}
```

```
queue.scheduleBroadcastsLocked();

}

registeredReceivers = null;

NR = 0;}...if ((receivers != null && receivers.size() > 0)

|| resultTo != null) {

BroadcastQueue queue = broadcastQueueForIntent(intent);

BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp,

callerPackage, callingPid, callingUid, resolvedType,

requiredPermission, appOp, receivers, resultTo, resultCode,

resultData, map, ordered, sticky, false, userId);

boolean replaced = replacePending && queue.replaceOrderedBroadcastLocked(r);

if (!replaced) {

// 发送静态广播

queue.enqueueOrderedBroadcastLocked(r);

queue.scheduleBroadcastsLocked();

}}}
```

大家应该都有听说过动态广播会优先于静态广播,从上面的代码我们可以看到,这实际是因为安卓的源代码就是按这个顺序写的...

最后我们来看一下 ActivityManagerService.collectReceiverComponents 方法,实际上静态广播静态就是从 PackageManagerService 中查询的:

```
private List<ResolveInfo> collectReceiverComponents(Intent intent, String resolvedType,

int[] users) {

...

List<ResolveInfo> newReceivers = AppGlobals.getPackageManager()
```

```
        .queryIntentReceivers(intent, resolvedType, STOCK_
PM_FLAGS, user);
```

```
    ...
```

粘性广播的实现原理

ActivityManagerService.broadcastIntentLocked 有下面这样一段代码, 它将粘性广播存到了 mStickyBroadcasts 中。

```
if (sticky) {

    ...

    ArrayMap<String, ArrayList<Intent>> stickies = mStickyBroadcasts.get
(userId);

    if (stickies == null) {

        stickies = new ArrayMap<String, ArrayList<Intent>>();

        mStickyBroadcasts.put(userId, stickies);
    }

    ArrayList<Intent> list = stickies.get(intent.getAction());

    if (list == null) {

        list = new ArrayList<Intent>();

        stickies.put(intent.getAction(), list);
    }

    int N = list.size();

    int i;

    for (i=0; i<N; i++) {

        if (intent.filterEquals(list.get(i))) {
```

```
// This sticky already exists, replace it.

list.set(i, new Intent(intent));

break;

}

}

if (i >= N) {

list.add(new Intent(intent));

}}
```

而 ManagerService.registerReceiver 会获取之前发送的粘性广播,再次发送给刚刚注册的 receiver:

```
...

List allSticky = null;

// 获取符合的粘性广播

Iterator actions = filter.actionsIterator();if (actions != null) {

while (actions.hasNext()) {

String action = (String)actions.next();

allSticky = getStickiesLocked(action, filter, allSticky,

UserHandle.USER_ALL);

allSticky = getStickiesLocked(action, filter, allSticky,

UserHandle.getUserId(callingUid));

}} else {

allSticky = getStickiesLocked(null, filter, allSticky,

UserHandle.USER_ALL);
```

```
    allSticky = getStickiesLocked(null, filter, allSticky,
```

```
        UserHandle.getUserId(callingUid));}...//向新注册的 receiver 发
```

```
送粘性广播 if (allSticky != null) {
```

```
    ArrayList receivers = new ArrayList();
```

```
    receivers.add(bf);
```



```
    int N = allSticky.size();
```

```
    for (int i=0; i<N; i++) {
```

```
        Intent intent = (Intent)allSticky.get(i);
```

```
        BroadcastQueue queue = broadcastQueueForIntent(intent);
```

```
        BroadcastRecord r = new BroadcastRecord(queue, intent, null,
```

```
            null, -1, -1, null, null, AppOpsManager.OP_NONE, receivers,
```

```
            null, 0,
```

```
            null, null, false, true, true, -1);
```

```
        queue.enqueueParallelBroadcastLocked(r);
```

```
        queue.scheduleBroadcastsLocked();
```

```
    }...}
```

getStickiesLocked 即从 mStickyBroadcasts 中查询之前发送过的粘性广播

```
private final List getStickiesLocked(String action, IntentFilter filter,
```

```
        List cur, int userId) {
```

```
    final ContentResolver resolver = mContext.getContentResolver();
```

```
    ArrayMap<String, ArrayList<Intent>> stickies = mStickyBroadcasts.get
```

```
(userId);
```

```
    if (stickies == null) {
```

```
        return cur;

    }

    final ArrayList<Intent> list = stickies.get(action);

    if (list == null) {

        return cur;

    }

    int N = list.size();

    for (int i=0; i<N; i++) {

        Intent intent = list.get(i);

        if (filter.match(resolver, intent, true, TAG) >= 0) {

            if (cur == null) {

                cur = new ArrayList<Intent>();

            }

            cur.add(intent);

        }

    }

    return cur;
}
```

广播队列

从 ActivityManagerService.broadcastIntentLocked 中我们可以看到,实际上它不是直接将广播发送到 BroadcastReceiver 中的.

而是将他包装到 BroadcastRecord 中,再放进 BroadcastQueue:

```
BroadcastQueue queue = broadcastQueueForIntent(intent);BroadcastRecord r
= new BroadcastRecord(queue, intent, null,
                      null, -1, -1, null, null, AppOpsManager.OP_NONE, receivers, null,
                      0,
```

```
    null, null, false, true, true, -1);

queue.enqueueParallelBroadcastLocked(r);

queue.scheduleBroadcastsLocked();
```

enqueueParallelBroadcastLocked 方法用于并发执行广播的发送.它很简单,就是将 BroadcastRecord 放到了 mParallelBroadcasts 中:

```
public void enqueueParallelBroadcastLocked(BroadcastRecord r) {

    mParallelBroadcasts.add(r);}
```

scheduleBroadcastsLocked 方法同样很简单,就是向 mHandler 发送了个 BROADCAST_INTENT_MSG 消息:

```
public void scheduleBroadcastsLocked() {

    if (mBroadcastsScheduled) {

        return;

    }

    mHandler.sendMessage(mHandler.obtainMessage(BROADCAST_INTENT_MSG, this));

    mBroadcastsScheduled = true;}
```

这个时候我们就需要再去看看 mHandler 在接收到 BROADCAST_INTENT_MSG 消息的时候会做些什么:

```
final Handler mHandler = new Handler() {

    public void handleMessage(Message msg) {

        switch (msg.what) {

            case BROADCAST_INTENT_MSG: {

                processNextBroadcast(true);
```

```
        } break;

    case BROADCAST_TIMEOUT_MSG: {

        synchronized (mService) {

            broadcastTimeoutLocked(true);

        }

    } break;

}

}};
```

processNextBroadcast 方法用于从队列中获取广播消息并发送给 BroadcastReceiver,它内部有两个分支,并行处理和串行处理.

并行处理

例如动态注册的非有序广播等就是使用并行处理,我们先看看并行处理的分支:

```
final void processNextBroadcast(boolean fromMsg) {

    synchronized(mService) {

        BroadcastRecord r;

        mService.updateCpuStats();

        if (fromMsg) {

            mBroadcastsScheduled = false;

        }

        while (mParallelBroadcasts.size() > 0) {

            r = mParallelBroadcasts.remove(0);

            r.dispatchTime = SystemClock.uptimeMillis();

            r.dispatchClockTime = System.currentTimeMillis();

        }

    }

}
```

```
final int N = r.receivers.size();

for (int i=0; i<N; i++) {

    Object target = r.receivers.get(i);

    // 发送消息给 Receiver

    deliverToRegisteredReceiverLocked(r, (BroadcastFilter)target,
false);

}

addBroadcastToHistoryLocked(r);

}

...

}

}

private final void deliverToRegisteredReceiverLocked(BroadcastRecord r,
BroadcastFilter filter, boolean ordered) {

...

// 获取 BroadcastReceiver 的 Binder

r.receiver = filter.receiverList.receiver.asBinder();

...

// 使用 Binder 机制将消息传递给 BroadcastReceiver

performReceiveLocked(filter.receiverList.app, filter.receiverList.re
ceiver,

new Intent(r.intent), r.resultCode, r.resultData,
r.resultExtras, r.ordered, r.initialSticky, r.userId);

...}
```

```
void performReceiveLocked(ProcessRecord app, IIntentReceiver receiver,
    Intent intent, int resultCode, String data, Bundle extras,
    boolean ordered, boolean sticky, int sendingUser) throws RemoteException {
    .....
    //通过 Binder 将消息处理传到应用进程,应用进程内部再使用 Handler 机制,将
    //消息处理放到主线程中
    app.thread.scheduleRegisteredReceiver(receiver, intent, resultCode,
        data, extras, ordered, sticky, sendingUser, app.re
        pProcState);
    .....
}
```

串行处理

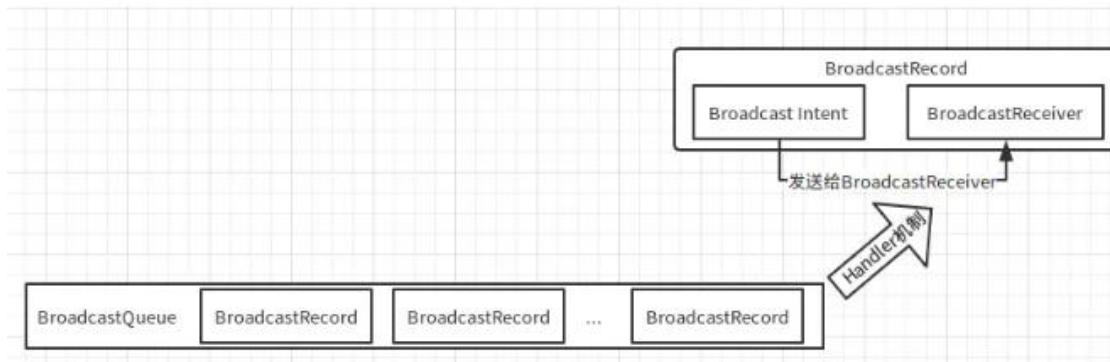
例如有序广播和静态广播等,会通过 enqueueOrderedBroadcastLocked 传给 BroadcastQueue:

```
public void enqueueOrderedBroadcastLocked(BroadcastRecord r) {
    mOrderedBroadcasts.add(r);}
```

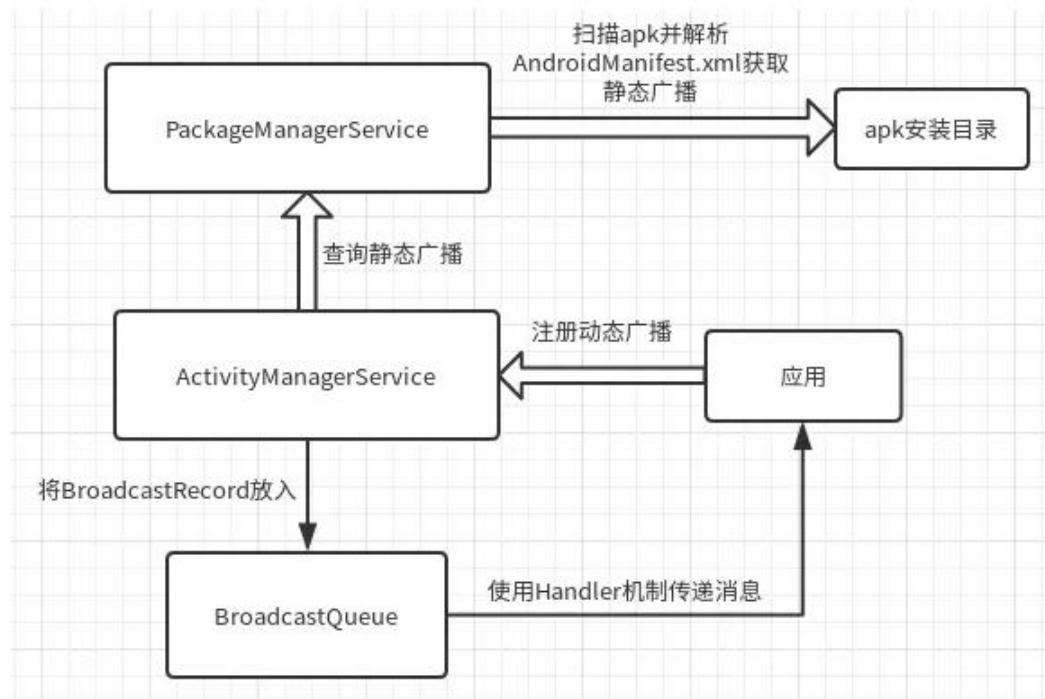
然后在 processNextBroadcast 里面会对 mOrderedBroadcasts 进行特殊处理,但是恕我愚钝,这部分代码比较复杂,我现在还没有搞懂它实际的怎么运行的.这块就留下来之后再讲了.

总结

广播队列传递广播给 Receiver 的原理其实就是将 BroadcastReceiver 和消息都放到 BroadcastRecord 里面,然后通过 Handler 机制遍历 BroadcastQueue 里面的 BroadcastRecord,将消息发送给 BroadcastReceiver:



所以整个广播的机制可以总结成下面这张图:



2.png

七、 AsyncTask 相关

1、**AsyncTask** 是串行还是并行执行？

2、**AsyncTask** 随着安卓版本的变迁

文章、**Android AsyncTask** 完全解析，带你从源码的角度彻底理解

AsyncTask 的基本用法

首先来看一下 **AsyncTask** 的基本用法，由于 **AsyncTask** 是一个抽象类，所以如果我们要想使用它，就必须要创建一个子类去继承它。在继承时我们可以为 **AsyncTask** 类指定三个泛型参数，这三个参数的用途如下：

1. Params

在执行 **AsyncTask** 时需要传入的参数，可用于在后台任务中使用。

2. Progress

后台任务执行时，如果需要在界面上显示当前的进度，则使用这里指定的泛型作为进度单位。

3. Result

当任务执行完毕后，如果需要对结果进行返回，则使用这里指定的泛型作为返回值类型。

因此，一个最简单的自定义 AsyncTask 就可以写成如下方式：

```
1 | class DownloadTask extends AsyncTask<Void, Integer, Boolean> {  
2 |     ....  
3 | }
```

复制

这里我们把 AsyncTask 的第一个泛型参数指定为 Void ,表示在执行 AsyncTask 的时候不需要传入参数给后台任务。第二个泛型参数指定为 Integer , 表示使用整型数据来作为进度显示单位。第三个泛型参数指定为 Boolean , 则表示使用布尔型数据来反馈执行结果。

当然，目前我们自定义的 DownloadTask 还是一个空任务，并不能进行任何实际的操作，我们还需要去重写 AsyncTask 中的几个方法才能完成对任务的定制。经常需要去重写的方法有以下四个：

1. onPreExecute()

这个方法会在后台任务开始执行之间调用，用于进行一些界面上的初始化操作，比如显示一个进度条对话框等。

2. doInBackground(Params...)

这个方法中的所有代码都会在子线程中运行，我们应该在这里去处理所有的耗时任务。任务一旦完成就可以通过 return 语句来将任务的执行结果进行返回，如果 AsyncTask 的第三个泛型参数指定的是 Void，就可以不返回任务执行结果。注意，在这个方法中是不可以进行 UI 操作的，如果需要更新 UI 元素，比如说反馈当前任务的执行进度，可以调用 publishProgress(Progress...) 方法来完成。

3. onProgressUpdate(Progress...)

当在后台任务中调用了 publishProgress(Progress...) 方法后，这个方法就很快会被调用，方法中携带的参数就是在后台任务中传递过来的。在这个方法中可以对 UI 进行操作，利用参数中的数值就可以对界面元素进行相应的更新。

4. onPostExecute(Result)

当后台任务执行完毕并通过 return 语句进行返回时，这个方法就很快会被调用。返回的数据会作为参数传递到此方法中，可以利用返回的数据来进行一些 UI 操作，比如说提醒任务执行的结果，以及关闭掉进度条对话框等。

因此，一个比较完整的自定义 AsyncTask 就可以写成如下方式：

```
class DownloadTask extends AsyncTask<Void, Integer, Boolean> {
```

```
    @Override
```

```
protected void onPreExecute() {  
    progressDialog.show();  
}  
  
@Override  
  
protected Boolean doInBackground(Void... params) {  
  
    try {  
  
        while (true) {  
  
            int downloadPercent = doDownload();  
  
            publishProgress(downloadPercent);  
  
            if (downloadPercent >= 100) {  
  
                break;  
            }  
        }  
    } catch (Exception e) {  
        return false;  
    }  
}
```

```
    }

    return true;

}

@Override

protected void onProgressUpdate(Integer... values) {

    progressDialog.setMessage("当前下载进度 : " + values[0] + "%");

}

@Override

protected void onPostExecute(Boolean result) {

    progressDialog.dismiss();

    if (result) {

        Toast.makeText(context, "下载成功",

Toast.LENGTH_SHORT).show();

    } else {


```

```
        Toast.makeText(context, "下载失败",  
        Toast.LENGTH_SHORT).show();  
  
    }  
  
}  
  
}
```

这里我们模拟了一个下载任务，在 `doInBackground()`方法中去执行具体的下载逻辑，在 `onProgressUpdate()`方法中显示当前的下载进度，在 `onPostExecute()`方法中来提示任务的执行结果。如果想要启动这个任务，只需要简单地调用以下代码即可：

```
new MyAsyncTask().execute();
```

以上就是 `AsyncTask` 的基本用法，怎么样，是不是感觉在子线程和 UI 线程之间进行切换变得灵活了很多？我们并不需求去考虑什么异步消息处理机制，也不需要专门使用一个 `Handler` 来发送和接收消息，只需要调用一下 `publishProgress()`方法就可以轻松地从子线程切换到 UI 线程了。

分析 `AsyncTask` 的源码

虽然 `AsyncTask` 这么简单好用，但你知道它是怎样实现的吗？那么接下来，我们就来分析一下 `AsyncTask` 的源码，对它的实现原理一探究竟。注意这里我选用的是 Android 4.0 的源码，如果你查看的是其它版本的源码，可能会有一些出入。

从之前 DownloadTask 的代码就可以看出，在启动某一个任务之前，要先 new 出它的实例，因此，我们就先来看一看 AsyncTask 构造函数中的源码，如下所示：

```
public AsyncTask() {  
  
    mWorker = new WorkerRunnable<Params, Result>() {  
  
        public Result call() throws Exception {  
  
            mTaskInvoked.set(true);  
  
            Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND)  
;  
  
            return postResult(doInBackground(mParams));  
  
        }  
  
    };  
  
    mFuture = new FutureTask<Result>(mWorker) {  
  
        @Override  
  
        protected void done() {  
  
            try {  
                ...  
            } catch (Exception e) {  
                ...  
            }  
        }  
    };  
}
```

```
    final Result result = get();

    postResultIfNotInvoked(result);

} catch (InterruptedException e) {

    android.util.Log.w(LOG_TAG, e);

} catch (ExecutionException e) {

    throw new RuntimeException("An error occurred while
executing doInBackground(),

e.getCause());

} catch (CancellationException e) {

    postResultIfNotInvoked(null);

} catch (Throwable t) {

    throw new RuntimeException("An error occurred while
executing "

+ "doInBackground()", t);

}

}

};
```

```
}
```

这段代码虽然看起来有点长，但实际上并没有任何具体的逻辑会得到执行，只是初始化了两个变量，`mWorker` 和 `mFuture`，并在初始化 `mFuture` 的时候将 `mWorker` 作为参数传入。`mWorker` 是一个 `Callable` 对象，`mFuture` 是一个 `FutureTask` 对象，这两个变量会暂时保存在内存中，稍后才会用到它们。

接着如果想要启动某一个任务，就需要调用该任务的 `execute()`方法，因此现在我们来看一看 `execute()`方法的源码，如下所示：

```
public final AsyncTask<Params, Progress, Result> execute(Params...
    params) {

    return executeOnExecutor(sDefaultExecutor, params);
}

}
```

简单的有点过分了，只有一行代码，仅是调用了 `executeOnExecutor()`方法，那么具体的逻辑就应该写在这个方法里了，快跟进去瞧一瞧：

```
public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor exec,
    Params... params) {

    if (mStatus != Status.PENDING) {

        switch (mStatus) {

            case RUNNING:

                throw new IllegalStateException("Cannot execute task:"
                    + " the task is already running.");

            case FINISHED:
        }
    }
}
```

```

        throw new IllegalStateException("Cannot execute task:"
                + " the task has already been executed "
                + "(a task can be executed only once)");

    }

}

mStatus = Status.RUNNING;

onPreExecute();

mWorker.mParams = params;

exec.execute(mFuture);

return this;

}

```

果然，这里的代码看上去才正常点。可以看到，在第 15 行调用了 `onPreExecute()`方法，因此证明了 `onPreExecute()`方法会第一个得到执行。可是接下来的代码就看不明白了，怎么没见到哪里有调用 `doInBackground()`方法呢？别着急，慢慢找总会找到的，我们看到，在第 17 行调用了 `Executor` 的 `execute()`方法，并将前面初始化的 `mFuture` 对象传了进去，那么这个 `Executor` 对象又是什么呢？查看上面的 `execute()`方法，原来是传入了一个 `sDefaultExecutor` 变量，接着找一下这个 `sDefaultExecutor` 变量是在哪里定义的，源码如下所示：

```

public static final Executor SERIAL_EXECUTOR = new SerialExecutor();

.....
private static volatile Executor sDefaultExecutor = SERIAL_EXECUTOR;

```

可以看到，这里先 `new` 出了一个 `SERIAL_EXECUTOR` 常量，然后将 `sDefaultExecutor` 的值赋值为这个常量，也就是说明，刚才在 `executeOnExecutor()`方法中调用的 `execute()`方法，其实也就是调用的 `SerialExecutor` 类中的 `execute()`方法。那么我们自然要去看看 `SerialExecutor` 的源码了，如下所示：

```
private static class SerialExecutor implements Executor {  
  
    final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();  
  
    Runnable mActive;  
  
    public synchronized void execute(final Runnable r) {  
  
        mTasks.offer(new Runnable() {  
  
            public void run() {  
  
                try {  
  
                    r.run();  
  
                } finally {  
  
                    scheduleNext();  
  
                }  
            }  
        });  
  
        if (mActive == null) {  
  
            scheduleNext();  
  
        }  
    }  
  
    protected synchronized void scheduleNext() {  
  
        if ((mActive = mTasks.poll()) != null) {  
  
            THREAD_POOL_EXECUTOR.execute(mActive);  
        }  
    }  
}
```

```
    }

}

}
```

SerialExecutor 类中也有一个 `execute()`方法，这个方法里的所有逻辑就是在子线程中执行的了，注意这个方法有一个 `Runnable` 参数，那么目前这个参数的值是什么呢？当然就是 `mFuture` 对象了，也就是说在第 9 行我们要调用的是 `FutureTask` 类的 `run()`方法，而在这个方法里又会去调用 `Sync` 内部类的 `innerRun()`方法，因此我们直接来看 `innerRun()`方法的源码：

```
void innerRun() {
    if (!compareAndSetState(READY, RUNNING))
        return;

    runner = Thread.currentThread();

    if (getState() == RUNNING) { // recheck after setting thread
        V result;
        try {
            result = callable.call();
        } catch (Throwable ex) {
            setException(ex);
            return;
        }
        set(result);
    } else {
        releaseShared(0); // cancel
    }
}
```

```
}
```

可以看到，在第 8 行调用了 `callable` 的 `call()`方法，那么这个 `callable` 对象是什么呢？其实就是在初始化 `mFuture` 对象时传入的 `mWorker` 对象了，此时调用的 `call()`方法，也就是一开始在 `AsyncTask` 的构造函数中指定的，我们把它单独拿出来看一下，代码如下所示：

```
public Result call() throws Exception {  
  
    mTaskInvoked.set(true);  
  
    Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);  
  
    return postResult(dolnBackground(mParams));  
  
}
```

在 `postResult()`方法的参数里面，我们终于找到了 `dolnBackground()`方法的调用处，虽然经过了很多周转，但目前的代码仍然是运行在子线程当中的，所以这也就是为什么我们可以在 `dolnBackground()`方法中去处理耗时的逻辑。接着将 `dolnBackground()`方法返回的结果传递给了 `postResult()`方法，这个方法的源码如下所示：

```
private Result postResult(Result result) {  
  
    Message message = sHandler.obtainMessage(MESSAGE_POST_RESULT,  
  
        new AsyncTaskResult<Result>(this, result);  
  
    message.sendToTarget();  
  
    return result;  
  
}
```

如果你已经熟悉了异步消息处理机制，这段代码对你来说一定非常简单吧。这里使用 `sHandler` 对象发出了一条消息，消息中携带了 `MESSAGE_POST_RESULT` 常量和一个表示任务执行结果的 `AsyncTaskResult` 对象。这个 `sHandler` 对象是 `InternalHandler` 类的一个实例，那么稍后这条消息肯定会在 `InternalHandler` 的 `handleMessage()`方法中被处理。`InternalHandler` 的源码如下所示：

```
private static class InternalHandler extends Handler {
```

```
@SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})  
  
@Override  
  
public void handleMessage(Message msg) {  
  
    AsyncTaskResult result = (AsyncTaskResult) msg.obj;  
  
    switch (msg.what) {  
  
        case MESSAGE_POST_RESULT:  
  
            // There is only one result  
  
            result.mTask.finish(result mData[0]);  
  
            break;  
  
        case MESSAGE_POST_PROGRESS:  
  
            result.mTask.onProgressUpdate(result mData);  
  
            break;  
  
    }  
  
}  
  
}
```

这里对消息的类型进行了判断，如果这是一条 MESSAGE_POST_RESULT 消息，就会去执行 `finish()`方法，如果这是一条 MESSAGE_POST_PROGRESS 消息，就会去执行 `onProgressUpdate()`方法。那么 `finish()`方法的源码如下所示：

可以看到，如果当前任务被取消掉了，就会调用 `onCancelled()`方法，如果没有被取消，则调用 `onPostExecute()`方法，这样当前任务的执行就全部结束了。

我们注意到，在刚才 InternalHandler 的 `handleMessage()`方法里，还有一种 MESSAGE_POST_PROGRESS 的消息类型，这种消息是用于当前进度的，调用

的正是 `onProgressUpdate()` 方法 , 那么什么时候才会发出这样一条消息呢 ? 相信你已经猜到了 , 查看 `publishProgress()` 方法的源码 , 如下所示 :

```
protected final void publishProgress(Progress... values) {  
    if (!isCancelled()) {  
        sHandler.obtainMessage(MESSAGE_POST_PROGRESS,  
            new AsyncTaskResult<Progress>(this, values)).sendToTarget();  
    }  
}
```

非常清晰了吧 ! 正因如此 , 在 `doInBackground()` 方法中调用 `publishProgress()` 方法才可以从子线程切换到 UI 线程 , 从而完成对 UI 元素的更新操作。其实也没有什么神秘的 , 因为说到底 , `AsyncTask` 也是使用的异步消息处理机制 , 只是做了非常好的封装而已。

读到这里 , 相信你对 `AsyncTask` 中的每个回调方法的作用、原理、以及何时会被调用都已经搞明白了吧。

关于 `AsyncTask` 你所不知道的秘密

不得不说 , 刚才我们在分析 `SerialExecutor` 的时候 , 其实并没有分析的很仔细 , 仅仅只是关注了它会调用 `mFuture` 中的 `run()` 方法 , 但是至于什么时候会调用我们并没有进一步地研究。其实 `SerialExecutor` 也是 `AsyncTask` 在 3.0 版本以后做了最主要修改的地方 , 它在 `AsyncTask` 中是以常量的形式被使用的 , 因此

在整个应用程序中的所有 `AsyncTask` 实例都会共用同一个 `SerialExecutor`。下面我们就来对这个类进行更加详细的分析，为了方便阅读，我把它的代码再贴出来一遍：

```
private static class SerialExecutor implements Executor {  
  
    final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();  
  
    Runnable mActive;  
  
    public synchronized void execute(final Runnable r) {  
  
        mTasks.offer(new Runnable() {  
  
            public void run() {  
  
                try {  
  
                    r.run();  
  
                } finally {  
  
                    scheduleNext();  
  
                }  
            }  
        });  
  
        if (mActive == null) {  
  
            scheduleNext();  
  
        }  
    }  
}
```

```
protected synchronized void scheduleNext() {  
  
    if ((mActive = mTasks.poll()) != null) {  
  
        THREAD_POOL_EXECUTOR.execute(mActive);  
  
    }  
  
}  
  
}
```

可以看到，SerialExecutor 是使用 ArrayDeque 这个队列来管理 Runnable 对象的，如果我们一次性启动了很多个任务，首先在第一次运行 execute()方法的时候，会调用 ArrayDeque 的 offer()方法将传入的 Runnable 对象添加到队列的尾部，然后判断 mActive 对象是不是等于 null，第一次运行当然是等于 null 了，于是会调用 scheduleNext()方法。在这个方法中会从队列的头部取值，并赋值给 mActive 对象，然后调用 THREAD_POOL_EXECUTOR 去执行取出的取出的 Runnable 对象。之后如何又有新的任务被执行，同样还会调用 offer()方法将传入的 Runnable 添加到队列的尾部，但是再去给 mActive 对象做非空检查的时候就会发现 mActive 对象已经不再是 null 了，于是就不再调用 scheduleNext()方法。

那么后面添加的任务岂不是永远得不到处理了？当然不是，看一看 offer()方法里传入的 Runnable 匿名类，这里使用了一个 try finally 代码块，并在 finally 中调用了 scheduleNext()方法，保证无论发生什么情况，这个方法都会被调用。也就是说，每次当一个任务执行完毕后，下一个任务才会得到执行，

SerialExecutor 模仿的是单一线程池的效果，如果我们快速地启动了很多任务，同一时刻只会有一个线程正在执行，其余的均处于等待状态。[Android 照片墙应用实现，再多的图片也不怕崩溃](#) 这篇文章中例子的运行结果也证实了这个结论。

不过你可能还不知道，在 Android 3.0 之前是并没有 SerialExecutor 这个类的，那个时候是直接在 AsyncTask 中构建了一个 sExecutor 常量，并对线程池总大小，同一时刻能够运行的线程数做了规定，代码如下所示：

```
private static final int CORE_POOL_SIZE = 5;  
  
private static final int MAXIMUM_POOL_SIZE = 128;  
  
private static final int KEEP_ALIVE = 10;  
  
.....  
  
private static final ThreadPoolExecutor sExecutor = new  
ThreadPoolExecutor(CORE_POOL_SIZE,  
  
MAXIMUM_POOL_SIZE, KEEP_ALIVE, TimeUnit.SECONDS, sWorkQueue,  
sThreadFactory);
```

可以看到，这里规定同一时刻能够运行的线程数为 5 个，线程池总大小为 128。也就是说当我们启动了 10 个任务时，只有 5 个任务能够立刻执行，另外的 5 个任务则需要等待，当有一个任务执行完毕后，第 6 个任务才会启动，以此类推。而线程池中最大能存放的线程数是 128 个，当我们尝试去添加第 129 个任务时，程序就会崩溃。

因此在 3.0 版本中 AsyncTask 的改动还是挺大的，在 3.0 之前的 AsyncTask 可以同时有 5 个任务在执行，而 3.0 之后的 AsyncTask 同时只能有 1 个任务在执行。为什么升级之后可以同时执行的任务数反而变少了呢？这是因为更新后的

AsyncTask 已变得更加灵活，如果不想使用默认的线程池，还可以自由地进行配置。比如使用如下的代码来启动任务：

```
Executor exec = new ThreadPoolExecutor(15, 200, 10,  
    TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());  
  
new DownloadTask().executeOnExecutor(exec);
```

这样就可以使用我们自定义的一个 Executor 来执行任务，而不是使用 SerialExecutor。上述代码的效果允许在同一时刻有 15 个任务正在执行，并且最多能够存储 200 个任务。

文章、Android 源码分析——带你认识不一样的 AsyncTask

前言

什么是 AsyncTask，相信搞过 android 开发的朋友们都不陌生。AsyncTask 内部封装了 Thread 和 Handler，可以让我们在后台进行计算并且把计算的结果及时更新到 UI 上，而这些正是 Thread+Handler 所做的事情，没错，AsyncTask 的作用就是简化 Thread+Handler，让我们能够通过更少的代码来完成一样的功能，这里，我要说明的是：AsyncTask 只是简化 Thread+Handler 而不是替代，实际上它也替代不了。同时，AsyncTask 从最开始到现在已经经过了几次代码修改，任务的执行逻辑慢慢地发生了改变，并不是大家所想象的那样：

AsyncTask 是完全并行执行的就像多个线程一样，其实不是的，所以用 AsyncTask 的时候还是要注意，下面会一一说明。另外本文主要是分析 AsyncTask 的源代码以及使用时候的一些注意事项，如果你还不熟悉 AsyncTask，请先阅读 [android 之 AsyncTask](#) 来了解其基本用法。

这里先给出 AsyncTask 的一个例子：

```
1 | private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
2 |     protected Long doInBackground(URL... urls) {
3 |         int count = urls.length;
4 |         long totalSize = 0;
5 |         for (int i = 0; i < count; i++) {
6 |             totalSize += Downloader.downloadFile(urls[i]);
7 |             publishProgress((int) ((i / (float) count) * 100));
8 |             // Escape early if cancel() is called
9 |             if (isCancelled()) break;
10 |         }
11 |         return totalSize;
12 |     }
13 |
14 |     protected void onProgressUpdate(Integer... progress) {
15 |         setProgressPercent(progress[0]);
16 |     }
17 |
18 |     protected void onPostExecute(Long result) {
19 |         showDialog("Downloaded " + result + " bytes");
20 |     }
21 | }
```

复制

使用 AsyncTask 的规则

AsyncTask 的类必须在 UI 线程加载(从 4.1 开始系统会帮我们自动完成)

AsyncTask 对象必须在 UI 线程创建

execute 方法必须在 UI 线程调用

不要在你的程序中去直接调用 onPreExecute(), onPostExecute,
doInBackground, onProgressUpdate 方法

一个 AsyncTask 对象只能执行一次 , 即只能调用一次 execute 方法 , 否
则会报运行时异常

AsyncTask 不是被设计为处理耗时操作的 , 耗时上限为几秒钟 , 如果要做
长耗时操作 , 强烈建议你使用 Executor , ThreadPoolExecutor 以及
FutureTask

在 1.6 之前，`AsyncTask` 是串行执行任务的，1.6 的时候 `AsyncTask` 开始采用线程池里处理并行任务，但是从 3.0 开始，为了避免 `AsyncTask` 所带来的并发错误，`AsyncTask` 又采用一个线程来串行执行任务

`AsyncTask` 到底是串行还是并行？

给大家做一下实验，请看如下实验代码：代码很简单，就是点击按钮的时候同时执行 5 个 `AsyncTask`，每个 `AsyncTask` 休眠 3s，同时把每个 `AsyncTask` 执行结束的时间打印出来，这样我们就能观察出到底是串行执行还是并行执行。

```
@Override  
  
public void onClick(View v) {  
  
    if (v == mButton) {  
  
        new MyAsyncTask("AsyncTask#1").execute("");  
  
        new MyAsyncTask("AsyncTask#2").execute("");  
  
        new MyAsyncTask("AsyncTask#3").execute("");  
  
        new MyAsyncTask("AsyncTask#4").execute("");  
  
        new MyAsyncTask("AsyncTask#5").execute("");  
  
    }  
}
```

```
}

private static class MyAsyncTask extends AsyncTask<String, Integer,
String> {

    private String mName = "AsyncTask";

    public MyAsyncTask(String name) {
        super();
        mName = name;
    }

    @Override
    protected String doInBackground(String... params) {
        try {
            Thread.sleep(3000);
        }
    }
}
```

```
        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        return mName;

    }

    @Override

    protected void onPostExecute(String result) {

        super.onPostExecute(result);

        SimpleDateFormat df = new

        SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

        Log.e(TAG, result + "execute finish at " + df.format(new

        Date()));

    }

}
```

我找了 2 个手机，系统分别是 4.1.1 和 2.3.3，按照我前面的描述，**AsyncTask** 在 4.1.1 应该是串行的，在 2.3.3 应该是并行的，到底是不是这样呢？请看 Log

Android 4.1.1 上执行：从下面 Log 可以看出，5 个 AsyncTask 共耗时 15s 且时间间隔为 3s，很显然是串行执行的

Application	Tag	Text
com.example.test	AsyncTaskTest	AsyncTask#1execute finish at 2013-12-27 01:44:47
com.example.test	AsyncTaskTest	AsyncTask#2execute finish at 2013-12-27 01:44:50
com.example.test	AsyncTaskTest	AsyncTask#3execute finish at 2013-12-27 01:44:53
com.example.test	AsyncTaskTest	AsyncTask#4execute finish at 2013-12-27 01:44:56
com.example.test	AsyncTaskTest	AsyncTask#5execute finish at 2013-12-27 01:44:59

Android 2.3.3 上执行：从下面 Log 可以看出，5 个 AsyncTask 的结束时间是一样的，很显然是并行执行

Application	Tag	Text
com.example.test	AsyncTaskTest	AsyncTask#1execute finish at 2013-12-27 01:45:39
com.example.test	AsyncTaskTest	AsyncTask#2execute finish at 2013-12-27 01:45:39
com.example.test	AsyncTaskTest	AsyncTask#3execute finish at 2013-12-27 01:45:39
com.example.test	AsyncTaskTest	AsyncTask#4execute finish at 2013-12-27 01:45:39
com.example.test	AsyncTaskTest	AsyncTask#5execute finish at 2013-12-27 01:45:39

结论：从上面的两个 Log 可以看出，我前面的描述是完全正确的。下面请看源码，让我们去了解下其中的原理。

源码分析

```
/*
```

```
* Copyright (C) 2008 The Android Open Source Project
```

```
*
```

```
* Licensed under the Apache License, Version 2.0 (the "License");
```

```
* you may not use this file except in compliance with the License.
```

- * You may obtain a copy of the License at
 - *
 - * <http://www.apache.org/licenses/LICENSE-2.0>
 - *
- * Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS,
 - * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
- * See the License for the specific language governing permissions and limitations under the License.

*/

package android.os;

import java.util.ArrayDeque;

import java.util.concurrent.BlockingQueue;

```
import java.util.concurrent.Callable;

import java.util.concurrent.CancellationException;

import java.util.concurrent.Executor;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.FutureTask;

import java.util.concurrent.LinkedBlockingQueue;

import java.util.concurrent.ThreadFactory;

import java.util.concurrent.ThreadPoolExecutor;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.TimeoutException;

import java.util.concurrent.atomic.AtomicBoolean;

import java.util.concurrent.atomic.AtomicInteger;

public abstract class AsyncTask<Params, Progress, Result> {

    private static final String LOG_TAG = "AsyncTask";
```

```
//获取当前的 cpu 核心数

private static final int CPU_COUNT =
    Runtime.getRuntime().availableProcessors();

//线程池核心容量

private static final int CORE_POOL_SIZE = CPU_COUNT + 1;

//线程池最大容量

private static final int MAXIMUM_POOL_SIZE = CPU_COUNT * 2 + 1;

//过剩的空闲线程的存活时间

private static final int KEEP_ALIVE = 1;

//ThreadFactory 线程工厂，通过工厂方法 newThread 来获取新线程

private static final ThreadFactory sThreadFactory = new
    ThreadFactory() {

    //原子整数，可以在超高并发下正常工作

    private final AtomicInteger mCount = new AtomicInteger(1);

    public Thread newThread(Runnable r) {
```

```
        return new Thread(r, "AsyncTask #" +  
mCount.getAndIncrement());  
  
    }  
  
};  
  
//静态阻塞式队列，用来存放待执行的任务，初始容量：128 个  
  
private static final BlockingQueue<Runnable> sPoolWorkQueue =  
    new LinkedBlockingQueue<Runnable>(128);  
  
/**  
 * 静态并发线程池，可以用来并行执行任务，尽管从 3.0 开始，AsyncTask  
默认是串行执行任务  
 * 但是我们仍然能构造出并行的 AsyncTask  
 */  
  
public static final Executor THREAD_POOL_EXECUTOR  
    = new ThreadPoolExecutor(CORE_POOL_SIZE,  
MAXIMUM_POOL_SIZE, KEEP_ALIVE,
```

```
    TimeUnit.SECONDS, sPoolWorkQueue,  
    sThreadFactory);  
  
/**  
 * 静态串行任务执行器，其内部实现了串行控制，  
 * 循环的取出一个个任务交给上述的并发线程池去执行  
 */  
  
public static final Executor SERIAL_EXECUTOR = new  
SerialExecutor();  
  
//消息类型：发送结果  
  
private static final int MESSAGE_POST_RESULT = 0x1;  
  
//消息类型：更新进度  
  
private static final int MESSAGE_POST_PROGRESS = 0x2;  
  
/**静态 Handler，用来发送上述两种通知，采用 UI 线程的 Looper 来处理  
消息  
* 这就是为什么 AsyncTask 必须在 UI 线程调用，因为子线程  
* 默认没有 Looper 无法创建下面的 Handler，程序会直接 Crash
```

```
*/  
  
private static final InternalHandler sHandler = new InternalHandler();  
  
//默认任务执行器，被赋值为串行任务执行器，就是它，AsyncTask 变成串  
行的了  
  
private static volatile Executor sDefaultExecutor =  
SERIAL_EXECUTOR;  
  
//如下两个变量我们先不要深究，不影响我们对整体逻辑的理解  
  
private final WorkerRunnable<Params, Result> mWorker;  
  
private final FutureTask<Result> mFuture;  
  
//任务的状态 默认为挂起，即等待执行，其类型标识为易变的（ volatile ）  
  
private volatile Status mStatus = Status.PENDING;  
  
//原子布尔型，支持高并发访问，标识任务是否被取消  
  
private final AtomicBoolean mCancelled = new AtomicBoolean();  
  
//原子布尔型，支持高并发访问，标识任务是否被执行过  
  
private final AtomicBoolean mTaskInvoked = new AtomicBoolean();  
  
/*串行执行器的实现，我们要好好看看，它是怎么把并行转为串行的
```

*目前我们需要知道，`asyncTask.execute(Params ...)`实际上会调用
*`SerialExecutor` 的 `execute` 方法，这一点后面再说明。也就是说：当你的
`asyncTask` 执行的时候，
*首先你的 task 会被加入到任务队列，然后排队，一个个执行
*/

```
private static class SerialExecutor implements Executor {  
  
    //线性双向队列，用来存储所有的 AsyncTask 任务  
  
    final ArrayDeque<Runnable> mTasks = new  
    ArrayDeque<Runnable>();  
  
    //当前正在执行的 AsyncTask 任务  
  
    Runnable mActive;  
  
    public synchronized void execute(final Runnable r) {  
  
        //将新的 AsyncTask 任务加入到双向队列中  
  
        mTasks.offer(new Runnable() {  
  
            public void run() {  
  
                try {  
                    //  
                } catch (Exception e) {  
                    //  
                }  
            }  
        });  
        if (mActive == null) {  
            mActive = r;  
        } else {  
            //  
        }  
    }  
}
```

```
//执行 AsyncTask 任务  
  
        r.run();  
  
    } finally {  
  
        //当前 AsyncTask 任务执行完毕后，进行下一轮执行，  
        //如果还有未执行任务的话  
  
        //这一点很明显体现了 AsyncTask 是串行执行任务的，  
        //总是一个任务执行完毕才会执行下一个任务  
  
        scheduleNext();  
  
    }  
  
});  
  
//如果当前没有任务在执行，直接进入执行逻辑  
  
if (mActive == null) {  
  
    scheduleNext();  
  
}  
  
}
```

```
protected synchronized void scheduleNext() {  
  
    //从任务队列中取出队列头部的任务 如果有就交给并发线程池去执行  
  
    if ((mActive = mTasks.poll()) != null) {  
  
        THREAD_POOL_EXECUTOR.execute(mActive);  
  
    }  
  
}  
  
}  
  
/**  
 * 任务的三种状态  
 */  
  
public enum Status {  
  
    /**  
     * 任务等待执行  
     */
```

```
PENDING,  
  
    /**  
     * 任务正在执行  
     */  
  
    RUNNING,  
  
    /**  
     * 任务已经执行结束  
     */  
  
    FINISHED,  
  
}  
  
/** 隐藏 API : 在 UI 线程中调用 , 用来初始化 Handler */  
  
public static void init() {  
  
    sHandler.getLooper();  
  
}
```

```
/** 隐藏 API : 为 AsyncTask 设置默认执行器 */

public static void setDefaultExecutor(Executor exec) {

    sDefaultExecutor = exec;

}

/**
 * Creates a new asynchronous task. This constructor must be
invoked on the UI thread.

 */

public AsyncTask() {

    mWorker = new WorkerRunnable<Params, Result>() {

        public Result call() throws Exception {

            mTaskInvoked.set(true);

            Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);

            //noinspection unchecked

```

```
        return postResult(doInBackground(mParams));

    }

};

mFuture = new FutureTask<Result>(mWorker) {

    @Override

    protected void done() {

        try {

            postResultIfNotInvoked(get());

        } catch (InterruptedException e) {

            android.util.Log.w(LOG_TAG, e);

        } catch (ExecutionException e) {

            throw new RuntimeException("An error occurred

while executing doInBackground()",

e.getCause());

        } catch (CancellationException e) {
```

```
    postResultIfNotInvoked(null);

}

};

}

}
```

```
private void postResultIfNotInvoked(Result result) {

    final boolean wasTaskInvoked = mTaskInvoked.get();

    if (!wasTaskInvoked) {

        postResult(result);

    }

}

//doInBackground 执行完毕，发送消息

private Result postResult(Result result) {

    @SuppressWarnings("unchecked")
```

```
    Message message =  
  
    sHandler.obtainMessage(MESSAGE_POST_RESULT,  
  
        new AsyncTaskResult<Result>(this, result));  
  
    message.sendToTarget();  
  
    return result;  
  
}  
  
/**  
 * 返回任务的状态  
 */  
  
public final Status getStatus() {  
  
    return mStatus;  
  
}  
  
/**  
 * 这个方法是我们必须要重写的，用来做后台计算
```

* 所在线程：后台线程

*/

```
protected abstract Result doInBackground(Params... params);
```

/**

* 在 doInBackground 之前调用，用来做初始化工作

* 所在线程：UI 线程

*/

```
protected void onPreExecute() {
```

}

/**

* 在 doInBackground 之后调用，用来接受后台计算结果更新 UI

* 所在线程：UI 线程

*/

```
protected void onPostExecute(Result result) {
```

```
}
```

```
/**
```

```
* Runs on the UI thread after {@link #publishProgress} is invoked.
```

```
/**
```

```
* 在 publishProgress 之后调用，用来更新计算进度
```

```
* 所在线程：UI 线程
```

```
*/
```

```
protected void onProgressUpdate(Progress... values) {
```

```
}
```

```
/**
```

```
* cancel 被调用并且 doInBackground 执行结束，会调用 onCancelled，  
表示任务被取消
```

```
* 这个时候 onPostExecute 不会再被调用，二者是互斥的，分别表示任务  
取消和任务执行完成
```

```
* 所在线程：UI 线程
```

```
 */
```

```
@SuppressWarnings({"UnusedParameters"})
```

```
protected void onCancelled(Result result) {
```

```
    onCancelled();
```

```
}
```

```
protected void onCancelled() {
```

```
}
```

```
public final boolean isCancelled() {
```

```
    return mCancelled.get();
```

```
}
```

```
public final boolean cancel(boolean mayInterruptIfRunning) {
```

```
    mCancelled.set(true);
```

```
    return mFuture.cancel(mayInterruptIfRunning);
```

```
    }

    public final Result get() throws InterruptedException,
        ExecutionException {
        return mFuture.get();
    }

    public final Result get(long timeout, TimeUnit unit) throws
        InterruptedException,
        ExecutionException, TimeoutException {
        return mFuture.get(timeout, unit);
    }

    /**
     * 这个方法如何执行和系统版本有关，在 AsyncTask 的使用规则里已经说
     * 明，如果你真的想使用并行 AsyncTask ，
     * 也是可以的，只要稍作修改
    
```

* 必须在 UI 线程调用此方法

*/

```
public final AsyncTask<Params, Progress, Result> execute(Params...
params) {

    //串行执行

    return executeOnExecutor(sDefaultExecutor, params);

    //如果我们想并行执行，这样改就行了，当然这个方法我们没法改

    //return executeOnExecutor(THREAD_POOL_EXECUTOR, params);

}
```

/**

* 通过这个方法我们可以自定义 AsyncTask 的执行方式，串行 or 并行，
甚至可以采用自己的 Executor

* 为了实现并行，我们可以在外部这么用 AsyncTask：

*

```
asyncTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR,
Params... params);
```

* 必须在 UI 线程调用此方法

*/

```
public final AsyncTask<Params, Progress, Result>
executeOnExecutor(Executor exec,
                  Params... params) {
    if (mStatus != Status.PENDING) {
        switch (mStatus) {
            case RUNNING:
                throw new IllegalStateException("Cannot execute task:" +
                    + " the task is already running.");
            case FINISHED:
                throw new IllegalStateException("Cannot execute task:" +
                    + " the task has already been executed " +
                    + "(a task can be executed only once)");
        }
    }
}
```

```
    }

    mStatus = Status.RUNNING;

    //这里#onPreExecute 会最先执行

    onPreExecute();

    mWorker.mParams = params;

    //然后后台计算#doInBackground 才真正开始

    exec.execute(mFuture);

    //接着会有#onProgressUpdate 被调用，最后是#onPostExecute

    return this;

}

/**
 * 这是 AsyncTask 提供的一个静态方法，方便我们直接执行一个 runnable

```

```
*/\n\n    public static void execute(Runnable runnable) {\n\n        sDefaultExecutor.execute(runnable);\n\n    }\n\n    /**\n     * 打印后台计算进度，onProgressUpdate 会被调用\n     */\n\n    protected final void publishProgress(Progress... values) {\n\n        if (!isCancelled()) {\n\n            sHandler.obtainMessage(MESSAGE_POST_PROGRESS,\n\n                new AsyncTaskResult<Progress>(this,\n\n                    values)).sendToTarget();\n\n        }\n\n    }\n}
```

//任务结束的时候会进行判断，如果任务没有被取消，则 onPostExecute
会被调用

```
private void finish(Result result) {  
  
    if (isCancelled()) {  
  
        onCancelled(result);  
  
    } else {  
  
        onPostExecute(result);  
  
    }  
  
    mStatus = Status.FINISHED;  
  
}
```

// AsyncTask 内部 Handler ,用来发送后台计算进度更新消息和计算完成消
息

```
private static class InternalHandler extends Handler {  
  
    @SuppressWarnings({"unchecked",  
    "RawUseOfParameterizedType"})  
  
    @Override
```

```
public void handleMessage(Message msg) {  
  
    AsyncTaskResult result = (AsyncTaskResult) msg.obj;  
  
    switch (msg.what) {  
  
        case MESSAGE_POST_RESULT:  
  
            // There is only one result  
  
            result.mTask.finish(result mData[0]);  
  
            break;  
  
        case MESSAGE_POST_PROGRESS:  
  
            result.mTask.onProgressUpdate(result mData);  
  
            break;  
  
    }  
  
}  
  
}  
  
private static abstract class WorkerRunnable<Params, Result>  
implements Callable<Result> {
```

```
Params[] mParams;  
}  
  
@SuppressWarnings({"RawUseOfParameterizedType"})  
private static class AsyncTaskResult<Data> {  
  
    final AsyncTask mTask;  
  
    final Data[] mData;  
  
    AsyncTaskResult(AsyncTask task, Data... data) {  
  
        mTask = task;  
  
        mData = data;  
  
    }  
  
}
```

让你的 AsyncTask 在 3.0 以上的系统中并行起来

通过上面的源码分析，我已经给出了在 3.0 以上系统中让 AsyncTask 并行执行的方法，现在，让我们来试一试，代码还是之前采用的测试代码，我们要稍作修改，调用 AsyncTask 的 executeOnExecutor 方法而不是 execute，请看：

```
@TargetApi(Build.VERSION_CODES.HONEYCOMB)

@Override

public void onClick(View v) {

    if (v == mButton) {

        new

MyAsyncTask("AsyncTask#1").executeOnExecutor(AsyncTask.THREAD_P

OOL_EXECUTOR,"");

        new

MyAsyncTask("AsyncTask#2").executeOnExecutor(AsyncTask.THREAD_P

OOL_EXECUTOR,"");

        new

MyAsyncTask("AsyncTask#3").executeOnExecutor(AsyncTask.THREAD_P

OOL_EXECUTOR,"");

        new

MyAsyncTask("AsyncTask#4").executeOnExecutor(AsyncTask.THREAD_P

OOL_EXECUTOR,"");
```

```
    new  
    MyAsyncTask("AsyncTask#5").executeOnExecutor(AsyncTask.THREAD_P  
    OOL_EXECUTOR,"");
```

```
}
```

```
}
```

```
private static class MyAsyncTask extends AsyncTask<String, Integer,  
String> {
```

```
    private String mName = "AsyncTask";
```

```
    public MyAsyncTask(String name) {
```

```
        super();
```

```
        mName = name;
```

```
}
```

```
    @Override

    protected String doInBackground(String... params) {

        try {

            Thread.sleep(3000);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        return mName;

    }

    @Override

    protected void onPostExecute(String result) {

        super.onPostExecute(result);

        SimpleDateFormat df = new

        SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

        Log.e(TAG, result + "execute finish at " + df.format(new

Date()));
```

```
    }  
  
}
```

下面是系统为 4.1.1 手机打印出的 Log: 很显然，我们的目的达到了，成功的让 AsyncTask 在 4.1.1 的手机上并行起来了

Application	Tag	Text
com.example.test	AsyncTaskTest	AsyncTask#1execute finish at 2013-12-27 01:52:40
com.example.test	AsyncTaskTest	AsyncTask#2execute finish at 2013-12-27 01:52:40
com.example.test	AsyncTaskTest	AsyncTask#4execute finish at 2013-12-27 01:52:40
com.example.test	AsyncTaskTest	AsyncTask#3execute finish at 2013-12-27 01:52:40
com.example.test	AsyncTaskTest	AsyncTask#5execute finish at 2013-12-27 01:52:40

八、Android 事件分发机制

1、**onTouch** 和 **onTouchEvent** 区别，调用顺序

2、**dispatchTouchEvent** , **onTouchEvent** ,
onInterceptTouchEvent 方法顺序以及使用场景

3、滑动冲突，如何解决

文章、Android 事件分发机制完全解析，带你从源码的角度彻底理解

只有一个 Activity，并且 Activity 中只有一个按钮。你可能已经知道，如果想要给这个按钮注册一个点击事件，只需要调用：

```
1 button.setOnClickListener(new OnClickListener() {  
2     @Override  
3     public void onClick(View v) {  
4         Log.d("TAG", "onClick execute");  
5     }  
6 });
```

这样在 onClick 方法里面写实现，就可以在按钮被点击的时候执行。你可能也已经知道，如果想给这个按钮再添加一个 touch 事件，只需要调用：

```
1 button.setOnTouchListener(new OnTouchListener() {  
2     @Override  
3     public boolean onTouch(View v, MotionEvent event) {  
4         Log.d("TAG", "onTouch execute, action " + event.getAction());  
5         return false;  
6     }  
7});
```

复制

onTouch 方法里能做的事情比 onClick 要多一些，比如判断手指按下、抬起、移动等事件。那么如果我两个事件都注册了，哪一个会先执行呢？我们来试一下就知道了，运行程序点击按钮，打印结果如下：

Application	Tag	Text
com.example.viewt...	TAG	onTouch execute, action 0
com.example.viewt...	TAG	onTouch execute, action 1
com.example.viewt...	TAG	onClick execute

可以看到，onTouch 是优先于 onClick 执行的，并且 onTouch 执行了两次，一次是 ACTION_DOWN，一次是 ACTION_UP(你还可能会有多个 ACTION_MOVE 的执行，如果你手抖了一下)。因此事件传递的顺序是先经过 onTouch，再传递到 onClick。

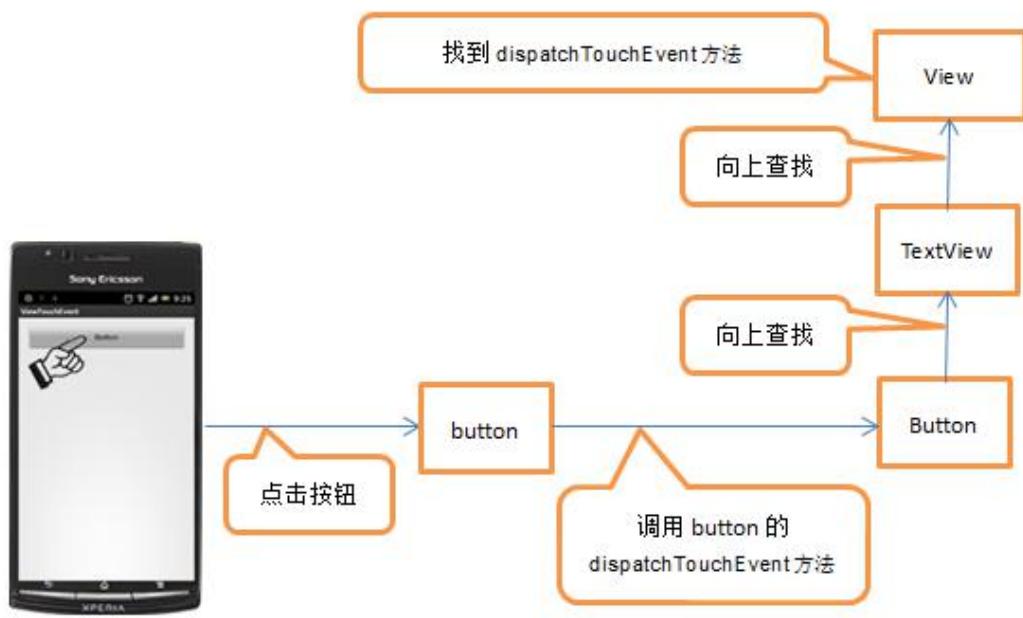
细心的朋友应该可以注意到，onTouch 方法是有返回值的，这里我们返回的是 false，如果我们尝试把 onTouch 方法里的返回值改成 true，再运行一次，结果如下：

Application	Tag	Text
com.example.viewt...	TAG	onTouch execute, action 0
com.example.viewt...	TAG	onTouch execute, action 1

我们发现，onClick 方法不再执行了！为什么会这样呢？你可以先理解成 onTouch 方法返回 true 就认为这个事件被 onTouch 消费掉了，因而不会再继续向下传递。

如果到现在为止，以上的所有知识点你都是清楚的，那么说明你对 Android 事件传递的基本用法应该是掌握了。不过别满足于现状，让我们从源码的角度分析一下，出现上述现象的原理是什么。

首先你需要知道一点，只要你触摸到了任何一个控件，就一定会调用该控件的 dispatchTouchEvent 方法。那当我们去点击按钮的时候，就会去调用 Button 类里的 dispatchTouchEvent 方法，可是你会发现 Button 类里并没有这个方法，那么就到它的父类 TextView 里去找一找，你会发现 TextView 里也没有这个方法，那没办法了，只好继续在 TextView 的父类 View 里找一找，这个时候你终于在 View 里找到了这个方法，示意图如下：



然后我们来看一下 View 中 dispatchTouchEvent 方法的源码：

```

1 public boolean dispatchTouchEvent(MotionEvent event) {
2     if (mOnTouchListener != null && (mViewFlags & ENABLED_MASK) == ENABLED &&
3         mOnTouchListener.onTouch(this, event)) {
4         return true;
5     }
6     return onTouchEvent(event);
7 }
```

复制

这个方法非常的简洁，只有短短几行代码！我们可以看到，在这个方法内，首先是进行了一个判断，如果 `mOnTouchListener != null`，`(mViewFlags & ENABLED_MASK) == ENABLED` 和 `mOnTouchListener.onTouch(this, event)` 这三个条件都为真，就返回 `true`，否则就去执行 `onTouchEvent(event)` 方法并返回。

先看一下第一个条件，`mOnTouchListener` 这个变量是在哪里赋值的呢？我们寻找之后在 `View` 里发现了如下方法：

```
1 | public void setOnTouchListener(OnTouchListener l) {  
2 |     mOnTouchListener = l;  
3 | }
```

复制

Bingo！找到了，`mOnTouchListener` 正是在 `setOnTouchListener` 方法里赋值的，也就是说只要我们给控件注册了 `touch` 事件，`mOnTouchListener` 就一定被赋值了。

第二个条件(`mViewFlags & ENABLED_MASK`) == `ENABLED` 是判断当前点击的控件是否是 `enable` 的，按钮默认都是 `enable` 的，因此这个条件恒定为 `true`。

第三个条件就比较关键了，`mOnTouchListener.onTouch(this, event)`，其实也就是去回调控件注册 `touch` 事件时的 `onTouch` 方法。也就是说如果我们在 `onTouch` 方法里返回 `true`，就会让这三个条件全部成立，从而整个方法直接返回 `true`。如果我们在 `onTouch` 方法里返回 `false`，就会再去执行 `onTouchEvent(event)` 方法。

现在我们可以结合前面的例子来分析一下了，首先在 `dispatchTouchEvent` 中最先执行的就是 `onTouch` 方法，因此 `onTouch` 肯定是要优先于 `onClick` 执行

的，也是印证了刚刚的打印结果。而如果在 onTouch 方法里返回了 true，就会让 dispatchTouchEvent 方法直接返回 true，不会再继续往下执行。而打印结果也证实了如果 onTouch 返回 true，onClick 就不会再执行了。

根据以上源码的分析，从原理上解释了我们前面例子的运行结果。而上面的分析还遗漏出了一个重要的信息，那就是 onClick 的调用肯定是在 onTouchEvent(event)方法中的！那我们马上来看下 onTouchEvent 的源码，如下所示：

```
public boolean onTouchEvent(MotionEvent event) {  
  
    final int viewFlags = mViewFlags;  
  
    if ((viewFlags & ENABLED_MASK) == DISABLED) {  
  
        // A disabled view that is clickable still consumes the touch  
        // events, it just doesn't respond to them.  
  
        return (((viewFlags & CLICKABLE) == CLICKABLE ||  
                (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE));  
  
    }  
  
    if (mTouchDelegate != null) {
```

```
    if (mTouchDelegate.onTouchEvent(event)) {  
  
        return true;  
  
    }  
  
}  
  
if (((viewFlags & CLICKABLE) == CLICKABLE ||  
  
(viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)) {  
  
    switch (event.getAction()) {  
  
        case MotionEvent.ACTION_UP:  
  
            boolean prepressed = (mPrivateFlags &  
PREPRESSED) != 0;  
  
            if ((mPrivateFlags & PRESSED) != 0 || prepressed) {  
  
                // take focus if we don't have it already and we  
should in  
  
                // touch mode.  
  
                boolean focusTaken = false;  
  
                if (isFocusable() && isFocusableInTouchMode()  
&& !isFocused()) {
```

```
    focusTaken = requestFocus();  
  
}  
  
if (!mHasPerformedLongPress) {  
  
    // This is a tap, so remove the longpress check  
  
    removeLongPressCallback();  
  
    // Only perform take click actions if we were in  
    the pressed state  
  
    if (!focusTaken) {  
  
        // Use a Runnable and post this rather than  
        calling  
  
        // performClick directly. This lets other  
        visual state  
  
        // of the view update before click actions  
        start.  
  
        if (mPerformClick == null) {  
  
            mPerformClick = new PerformClick();  
  
        }  
    }
```

```
        if (!post(mPerformClick)) {  
  
            performClick();  
  
        }  
  
    }  
  
    if (mUnsetPressedState == null) {  
  
        mUnsetPressedState = new  
        UnsetPressedState();  
  
    }  
  
    if (prepressed) {  
  
        mPrivateFlags |= PRESSED;  
  
        refreshDrawableState();  
  
        postDelayed(mUnsetPressedState,  
  
                    ViewConfiguration.getPressedStateDuration());  
  
    } else if (!post(mUnsetPressedState)) {  
  
        // If the post failed, unpress right now
```

```
        mUnsetPressedState.run();

    }

    removeTapCallback();

}

break;

case MotionEvent.ACTION_DOWN:

    if (mPendingCheckForTap == null) {

        mPendingCheckForTap = new CheckForTap();

    }

    mPrivateFlags |= PREPRESSED;

    mHasPerformedLongPress = false;

    postDelayed(mPendingCheckForTap,

ViewConfiguration.getTapTimeout());



break;

case MotionEvent.ACTION_CANCEL:

    mPrivateFlags &= ~PRESSED;
```

```
refreshDrawableState();

removeTapCallback();

break;

case MotionEvent.ACTION_MOVE:

    final int x = (int) event.getX();

    final int y = (int) event.getY();

    // Be lenient about moving outside of buttons

    int slop = mTouchSlop;

    if ((x < 0 - slop) || (x >= getWidth() + slop) ||
        (y < 0 - slop) || (y >= getHeight() + slop)) {

        // Outside button

        removeTapCallback();

        if ((mPrivateFlags & PRESSED) != 0) {

            // Remove any future long press/tap checks

            removeLongPressCallback();

            // Need to switch from pressed to not pressed
        }
    }
}
```

```
    mPrivateFlags &= ~PRESSED;

    refreshDrawableState();

}

break;

}

return true;

}

return false;

}
```

相较于刚才的 dispatchTouchEvent 方法，onTouchEvent 方法复杂了很多，不过没关系，我们只挑重点看就可以了。

首先在第 14 行我们可以看出，如果该控件是可以点击的就会进入到第 16 行的 switch 判断中去，而如果当前的事件是抬起手指，则会进入到 MotionEvent.ACTION_UP 这个 case 当中。在经过种种判断之后，会执行到第 38 行的 performClick()方法，那我们进入到这个方法里瞧一瞧：

```
1 public boolean performClick() {  
2     sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);  
3     if (mOnClickListener != null) {  
4         playSoundEffect(SoundEffectConstants.CLICK);  
5         mOnClickListener.onClick(this);  
6         return true;  
7     }  
8     return false;  
9 }
```

复制

可以看到，只要 `mOnClickListener` 不是 `null`，就会去调用它的 `onClick` 方法，那 `mOnClickListener` 又是在哪里赋值的呢？经过寻找后找到如下方法：

```
1 public void setOnClickListener(OnClickListener l) {  
2     if (!isClickable()) {  
3         setClickable(true);  
4     }  
5     mOnClickListener = l;  
6 }
```

复制

一切都是那么清楚了！当我们通过调用 `setOnClickListener` 方法来给控件注册一个点击事件时，就会给 `mOnClickListener` 赋值。然后每当控件被点击时，都会在 `performClick()` 方法里回调被点击控件的 `onClick` 方法。

这样 View 的整个事件分发的流程就让我们搞清楚了！不过别高兴的太早，现在还没结束，还有一个很重要的知识点需要说明，就是 touch 事件的层级传递。我们都应该知道如果给一个控件注册了 touch 事件，每次点击它的时候都会触发一系列的 `ACTION_DOWN`，`ACTION_MOVE`，`ACTION_UP` 等事件。这里需要注意，如果你在执行 `ACTION_DOWN` 的时候返回了 `false`，后面一系列其它的 action 就不会再得到执行了。简单的说，就是当 `dispatchTouchEvent` 在进行事件分发的时候，只有前一个 action 返回 `true`，才会触发后一个 action。

说到这里，很多的朋友肯定要有巨大的疑问了。这不是在自相矛盾吗？前面的例子中，明明在 onTouch 事件里面返回了 false，ACTION_DOWN 和 ACTION_UP 不是都得到执行了吗？其实你只是被假象所迷惑了，让我们仔细分析一下，在前面的例子当中，我们到底返回的是什么。

参考着我们前面分析的源码，首先在 onTouch 事件里返回了 false，就一定会进入到 onTouchEvent 方法中，然后我们来看一下 onTouchEvent 方法的细节。由于我们点击了按钮，就会进入到第 14 行这个 if 判断的内部，然后你会发现，不管当前的 action 是什么，最终都一定会走到第 89 行，返回一个 true。

是不是有一种被欺骗的感觉？明明在 onTouch 事件里返回了 false，系统还是在 onTouchEvent 方法中帮你返回了 true。就因为这个原因，才使得前面的例子中 ACTION_UP 可以得到执行。

那我们可以换一个控件，将按钮替换成 ImageView，然后给它也注册一个 touch 事件，并返回 false。如下所示：

```
1 imageView.setOnTouchListener(new OnTouchListener() {  
2     @Override  
3     public boolean onTouch(View v, MotionEvent event) {  
4         Log.d("TAG", "onTouch execute, action " + event.getAction());  
5         return false;  
6     }  
7 });
```

运行一下程序，点击 ImageView，你会发现结果如下：

Application	Tag	Text
com.example.viewt...	TAG	onTouch execute, action 0

在 ACTION_DOWN 执行完后，后面的一系列 action 都不会得到执行了。这又是为什么呢？因为 ImageView 和按钮不同，它是默认不可点击的，因此在 onTouchEvent 的第 14 行判断时无法进入到 if 的内部，直接跳到第 91 行返回了 false，也就导致后面其它的 action 都无法执行了。

好了，关于 View 的事件分发，我想讲的东西全都在这里了。现在我们再来看看一下开篇时提到的那三个问题，相信每个人都会有更深一层的理解。

1. onTouch 和 onTouchEvent 有什么区别，又该如何使用？

从源码中可以看出，这两个方法都是在 View 的 dispatchTouchEvent 中调用的，onTouch 优先于 onTouchEvent 执行。如果在 onTouch 方法中通过返回 true 将事件消费掉，onTouchEvent 将不会再执行。

另外需要注意的是，onTouch 能够得到执行需要两个前提条件，第一 mOnTouchListener 的值不能为空，第二当前点击的控件必须是 enable 的。因

此如果你有一个控件是非 enable 的，那么给它注册 onTouch 事件将永远得不到执行。对于这一类控件，如果我们想要监听它的 touch 事件，就必须通过在该控件中重写 onTouchEvent 方法来实现。

2. 为什么给 ListView 引入了一个滑动菜单的功能，ListView 就不能滚动了？

如果你阅读了 [Android 滑动框架完全解析，教你如何一分钟实现滑动菜单特效](#) 这篇文章，你应该会知道滑动菜单的功能是通过给 ListView 注册了一个 touch 事件来实现的。如果你在 onTouch 方法里处理完了滑动逻辑后返回 true，那么 ListView 本身的滚动事件就被屏蔽了，自然也就无法滑动(原理同前面例子中按钮不能点击)，因此解决办法就是在 onTouch 方法里返回 false。

3. 为什么图片轮播器里的图片使用 Button 而不用 ImageView？

提这个问题的朋友是看过了 [Android 实现图片滚动控件，含页签功能，让你的应用像淘宝一样炫起来](#) 这篇文章。当时我在图片轮播器里使用 Button，主要原因是因为 Button 是可点击的，而 ImageView 是不可点击的。如果想要使用 ImageView，可以有两种改法。第一，在 ImageView 的 onTouch 方法里返回 true，这样可以保证 ACTION_DOWN 之后的其它 action 都能得到执行，才能实现图片滚动的效果。第二，在布局文件里面给 ImageView 增加一个 android:clickable="true" 的属性，这样 ImageView 变成可点击的之后，即使

在 onTouch 里返回了 false , ACTION_DOWN 之后的其它 action 也是可以得到执行的。

文章、Android ViewGroup 事件分发机制

1、案例

首先我们接着上一篇的代码，在代码中添加一个自定义的 LinearLayout :

```
package com.example.zhy_event03;

import android.content.Context;
import android.util.AttributeSet;
import android.util.Log;
import android.view.MotionEvent;
import android.widget.LinearLayout;

public class MyLinearLayout extends LinearLayout
{
    private static final String TAG = MyLinearLayout.class.getSimpleName();

    public MyLinearLayout(Context context, AttributeSet attrs)
    {
        super(context, attrs);
    }

    @Override
    public boolean dispatchTouchEvent(MotionEvent ev)
    {
        int action = ev.getAction();
        switch (action)
        {
            case MotionEvent.ACTION_DOWN:
                Log.e(TAG, "dispatchTouchEvent ACTION_DOWN");
                break;
            case MotionEvent.ACTION_MOVE:
                Log.e(TAG, "dispatchTouchEvent ACTION_MOVE");
                break;
            case MotionEvent.ACTION_UP:
                Log.e(TAG, "dispatchTouchEvent ACTION_UP");
                break;
        }
        return true;
    }
}
```

```
    default:
        break;
    }
    return super.dispatchTouchEvent(ev);
}

@Override
public boolean onTouchEvent(MotionEvent event)
{

    int action = event.getAction();

    switch (action)
    {
        case MotionEvent.ACTION_DOWN:
            Log.e(TAG, "onTouchEvent ACTION_DOWN");
            break;
        case MotionEvent.ACTION_MOVE:
            Log.e(TAG, "onTouchEvent ACTION_MOVE");
            break;
        case MotionEvent.ACTION_UP:
            Log.e(TAG, "onTouchEvent ACTION_UP");
            break;

        default:
            break;
    }

    return super.onTouchEvent(event);
}

@Override
public boolean onInterceptTouchEvent(MotionEvent ev)
{

    int action = ev.getAction();
    switch (action)
    {
        case MotionEvent.ACTION_DOWN:
            Log.e(TAG, "onInterceptTouchEvent ACTION_DOWN");
            break;
        case MotionEvent.ACTION_MOVE:
            Log.e(TAG, "onInterceptTouchEvent ACTION_MOVE");
            break;
    }
}
```

```
        break;
    case MotionEvent.ACTION_UP:
        Log.e(TAG, "onInterceptTouchEvent ACTION_UP");
        break;

    default:
        break;
    }

    return super.onInterceptTouchEvent(ev);
}

@Override
public void requestDisallowInterceptTouchEvent(boolean disallowIntercept)
{
    Log.e(TAG, "requestDisallowInterceptTouchEvent ");
    super.requestDisallowInterceptTouchEvent(disallowIntercept);
}

}
```

继承 `LinearLayout`, 然后复写了与事件分发机制有关的代码, 添加上了日志的打印~

然后看我们的布局文件 :

```
<com.example.zhy_event03.MyLinearLayout
    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"
    tools:context=".MainActivity" >

    <com.example.zhy_event03.MyButton
        android:id="@+id/id_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="click me" />

</com.example.zhy_event03.MyLinearLayout>
```

`MyLinearLayout` 中包含一个 `MyButton`, `MyButton` 都上篇博客中已经出现过, 这里就不再贴代码了, 不清楚可以去查看~

然后 `MainActivity` 就是直接加载布局, 没有任何代码~~~

直接运行我们的代码，然后点击我们的 Button，依然是有意的 MOVE 一下，不然不会触发 MOVE 事件，看一下日志的输出：

```
09-06 09:57:27.287: E/MyLinearLayout(959): dispatchTouchEvent ACTION_DOWN  
09-06 09:57:27.287: E/MyLinearLayout(959): onInterceptTouchEvent ACTION_DOWN  
09-06 09:57:27.287: E/MyButton(959): dispatchTouchEvent ACTION_DOWN  
09-06 09:57:27.297: E/MyButton(959): onTouchEvent ACTION_DOWN  
09-06 09:57:27.297: E/MyButton(959): onTouchEvent ACTION_MOVE  
09-06 09:57:27.327: E/MyLinearLayout(959): dispatchTouchEvent ACTION_MOVE  
09-06 09:57:27.327: E/MyLinearLayout(959): onInterceptTouchEvent ACTION_MOVE  
09-06 09:57:27.337: E/MyButton(959): dispatchTouchEvent ACTION_MOVE  
09-06 09:57:27.337: E/MyButton(959): onTouchEvent ACTION_MOVE  
09-06 09:57:27.457: E/MyLinearLayout(959): dispatchTouchEvent ACTION_UP  
09-06 09:57:27.457: E/MyLinearLayout(959): onInterceptTouchEvent ACTION_UP  
09-06 09:57:27.457: E/MyButton(959): dispatchTouchEvent ACTION_UP  
09-06 09:57:27.457: E/MyButton(959): onTouchEvent ACTION_UP
```

可以看到大体的事件流程为：

MyLinearLayout 的 dispatchTouchEvent -> MyLinearLayout 的
onInterceptTouchEvent -> MyButton 的 dispatchTouchEvent
-> Mybutton 的 onTouchEvent

可以看出，在 View 上触发事件，最先捕获到事件的为 View 所在的 ViewGroup，
然后才会到 View 自身~

下面我们按照日志的输出，进入源码~

2、源码分析

ViewGroup - dispatchTouchEvent

1、ViewGroup - dispatchTouchEvent - ACTION_DOWN

首先是 ViewGroup 的 dispatchTouchEvent 方法：

```
@Override  
public boolean dispatchTouchEvent(MotionEvent ev) {
```

```

if (!onFilterTouchEventForSecurity(ev)) {
    return false;
}

final int action = ev.getAction();
final float xf = ev.getX();
final float yf = ev.getY();
final float scrolledXFloat = xf + mScrollIX;
final float scrolledYFloat = yf + mScrollIY;
final Rect frame = mTempRect;

boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;

if (action == MotionEvent.ACTION_DOWN) {
    if (mMotionTarget != null) {
        // this is weird, we got a pen down, but we thought it was
        // already down!
        // XXX: We should probably send an ACTION_UP to the current
        // target.
        mMotionTarget = null;
    }
    // If we're disallowing intercept or if we're allowing and we didn't
    // intercept
    if (disallowIntercept || !onInterceptTouchEvent(ev)) {
        // reset this event's action (just to protect ourselves)
        ev.setAction(MotionEvent.ACTION_DOWN);
        // We know we want to dispatch the event down, find a child
        // who can handle it, start with the front-most child.
        final int scrolledXInt = (int) scrolledXFloat;
        final int scrolledYInt = (int) scrolledYFloat;
        final View[] children = mChildren;
        final int count = mChildrenCount;

        for (int i = count - 1; i >= 0; i--) {
            final View child = children[i];
            if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE
                || child.getAnimation() != null) {
                child.getHitRect(frame);
                if (frame.contains(scrolledXInt, scrolledYInt)) {
                    // offset the event to the view's coordinate system
                    final float xc = scrolledXFloat - child.mLeft;
                    final float yc = scrolledYFloat - child.mTop;
                    ev.setLocation(xc, yc);
                    child.mPrivateFlags &= ~CANCEL_NEXT_UP_EVENT;
                }
            }
        }
    }
}

```

```
        if (child.dispatchTouchEvent(ev)) {
            // Event handled, we have a target now.
            mMotionTarget = child;
            return true;
        }
        // The event didn't get handled, try the next view.
        // Don't reset the event's location, it's not
        // necessary here.
    }
}
}
}
}
}
..../
/other code omitted
```

代码比较长，决定分段贴出，首先贴出的是 ACTION_DOWN 事件相关的代码。

16 行：进入 ACTION_DOWN 的处理

17-23 行：将 mMotionTarget 置为 null

26 行：进行判断：if(disallowIntercept || !onInterceptTouchEvent(ev))

两种可能会进入 IF 代码段

1、当前不允许拦截，即 disallowIntercept =true ,

2、当前允许拦截但是不拦截，即 disallowIntercept =false,但是

onInterceptTouchEvent(ev)返回 false ;

注：disallowIntercept 可以通过

viewGroup.requestDisallowInterceptTouchEvent(boolean);进行设置，后面会详细说；而 onInterceptTouchEvent(ev)可以进行复写。

36-57 行：开始遍历所有的子 View

41 行：进行判断当前的 x,y 坐标是否落在子 View 身上，如果在，47 行，执行 child.dispatchTouchEvent(ev)，就进入了 View 的 dispatchTouchEvent 代码中了，如果不了解请参考：[Android View 的事件分发机制](#)，当 child.dispatchTouchEvent(ev) 返回 true，则为 mMotionTarget=child；然后 return true；

ViewGroup 的 ACTION_DOWN 分析结束，总结一下：

ViewGroup 实现捕获到 DOWN 事件，如果代码中不做 TOUCH 事件拦截，则开始查找当前 x,y 是否在某个子 View 的区域内，如果在，则把事件分发下去。

按照日志，接下来到达 ACTION_MOVE

2、 ViewGroup – dispatchTouchEvent – ACTION_MOVE

首先我们源码进行删减，只留下 MOVE 相关的代码：

```
@Override
public boolean dispatchTouchEvent(MotionEvent ev) {
    final int action = ev.getAction();
    final float xf = ev.getX();
    final float yf = ev.getY();
    final float scrolledXFloat = xf + mScrollIX;
    final float scrolledYFloat = yf + mScrollIY;
    final Rect frame = mTempRect;

    boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;

    //...ACTION_DOWN

    //...ACTION_UP or ACTION_CANCEL

    // The event wasn't an ACTION_DOWN, dispatch it to our target if
    // we have one.
    final View target = mMotionTarget;
```

```
// if have a target, see if we're allowed to and want to intercept its
// events
if (!disallowIntercept && onInterceptTouchEvent(ev)) {
    //....
}

// finally offset the event to the target's coordinate system and
// dispatch the event.
final float xc = scrolledXFloat - (float) target.mLeft;
final float yc = scrolledYFloat - (float) target.mTop;
ev.setLocation(xc, yc);

return target.dispatchTouchEvent(ev);
}
```

18 行 : 把 ACTION_DOWN 时赋值的 mMotionTarget , 付给 target ;

23 行 : if (!disallowIntercept && onInterceptTouchEvent(ev)) 当前允许拦截且拦截了 , 才进入 IF 体 , 当然了默认是不会拦截的~这里执行了 onInterceptTouchEvent(ev)

28-30 行 : 把坐标系统转化为子 View 的坐标系统

32 行 : 直接 return target.dispatchTouchEvent(ev);

可以看到 , 正常流程下 , ACTION_MOVE 在检测完是否拦截以后 , 直接调用了子 View.dispatchTouchEvent , 事件分发下去;

最后就是 ACTION_UP 了

3、 ViewGroup - dispatchTouchEvent - ACTION_UP

```
public boolean dispatchTouchEvent(MotionEvent ev) {
    if (!onFilterTouchEventForSecurity(ev)) {
        return false;
    }

    final int action = ev.getAction();
    final float xf = ev.getX();
```

```
final float yf = ev.getY();
final float scrolledXFloat = xf + mScrollIX;
final float scrolledYFloat = yf + mScrollIY;
final Rect frame = mTempRect;

boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;

if (action == MotionEvent.ACTION_DOWN) {...}

boolean isUpOrCancel = (action == MotionEvent.ACTION_UP) ||
    (action == MotionEvent.ACTION_CANCEL);

if (isUpOrCancel) {
    mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
}
final View target = mMotionTarget;
if(target ==null ){...}
if (!disallowIntercept && onInterceptTouchEvent(ev)) {...}

if (isUpOrCancel) {
    mMotionTarget = null;
}

// finally offset the event to the target's coordinate system and
// dispatch the event.
final float xc = scrolledXFloat - (float) target.mLeft;
final float yc = scrolledYFloat - (float) target.mTop;
ev.setLocation(xc, yc);

return target.dispatchTouchEvent(ev);
}
```

17 行：判断当前是否是 ACTION_UP

21 , 28 行 : 分别重置拦截标志位以及将 DOWN 赋值的 mMotionTarget 置为 null , 都 UP 了 , 当然置为 null , 下一次 DOWN 还会再赋值的~

最后 , 修改坐标系统 , 然后调用 target.dispatchTouchEvent(ev);

正常情况下，即我们上例整个代码的流程我们已经走完了：

- 1、ACTION_DOWN 中，ViewGroup 捕获到事件，然后判断是否拦截，如果没有拦截，则找到包含当前 x,y 坐标的子 View，赋值给 mMotionTarget，然后调用 mMotionTarget.dispatchTouchEvent
- 2、ACTION_MOVE 中，ViewGroup 捕获到事件，然后判断是否拦截，如果没有拦截，则直接调用 mMotionTarget.dispatchTouchEvent(ev)
- 3、ACTION_UP 中，ViewGroup 捕获到事件，然后判断是否拦截，如果没有拦截，则直接调用 mMotionTarget.dispatchTouchEvent(ev)

当然了在分发之前都会修改下坐标系统，把当前的 x, y 分别减去 child.left 和 child.top，然后传给 child;

3、关于拦截

1、如何拦截

上面的总结都是基于：如果没有拦截；那么如何拦截呢？

复写 ViewGroup 的 onInterceptTouchEvent 方法：

```
@Override  
public boolean onInterceptTouchEvent(MotionEvent ev)  
{  
    int action = ev.getAction();  
    switch (action)  
    {  
        case MotionEvent.ACTION_DOWN:  
            //如果你觉得需要拦截
```

```
        return true ;
    case MotionEvent.ACTION_MOVE:
        //如果你觉得需要拦截
        return true ;
    case MotionEvent.ACTION_UP:
        //如果你觉得需要拦截
        return true ;
    }

    return false;
}
```

默认是不拦截的，即返回 `false`；如果你需要拦截，只要 `return true` 就行了，这要该事件就不会往子 View 传递了，并且如果你在 `DOWN` return `true`，则 `DOWN,MOVE,UP` 子 View 都不会捕获事件；如果你在 `MOVE` return `true`，则子 View 在 `MOVE` 和 `UP` 都不会捕获事件。

原因很简单，当 `onInterceptTouchEvent(ev)` return `true` 的时候，会把 `mMotionTarget` 置为 `null`；

2、如何不被拦截

如果 `ViewGroup` 的 `onInterceptTouchEvent(ev)` 当 `ACTION_MOVE` 时 `return true`，即拦截了子 View 的 `MOVE` 以及 `UP` 事件；

此时子 View 希望依然能够响应 `MOVE` 和 `UP` 时该咋办呢？

Android 给我们提供了一个方法：

`requestDisallowInterceptTouchEvent(boolean)` 用于设置是否允许拦截，我们在子 View 的 `dispatchTouchEvent` 中直接这么写：

```
@Override
public boolean dispatchTouchEvent(MotionEvent event)
{
    getParent().requestDisallowInterceptTouchEvent(true);
    int action = event.getAction();

    switch (action)
{
```

```

        case MotionEvent.ACTION_DOWN:
            Log.e(TAG, "dispatchTouchEvent ACTION_DOWN");
            break;
        case MotionEvent.ACTION_MOVE:
            Log.e(TAG, "dispatchTouchEvent ACTION_MOVE");
            break;
        case MotionEvent.ACTION_UP:
            Log.e(TAG, "dispatchTouchEvent ACTION_UP");
            break;

        default:
            break;
    }
    return super.dispatchTouchEvent(event);
}

```

getParent().requestDisallowInterceptTouchEvent(true); 这样即使 ViewGroup 在 MOVE 的时候 return true，子 View 依然可以捕获到 MOVE 以及 UP 事件。

从源码也可以解释：

ViewGroup MOVE 和 UP 拦截的源码是这样的：

```

if (!disallowIntercept && onInterceptTouchEvent(ev)) {
    final float xc = scrolledXFloat - (float) target.mLeft;
    final float yc = scrolledYFloat - (float) target.mTop;
    mPrivateFlags &= ~CANCEL_NEXT_UP_EVENT;
    ev.setAction(MotionEvent.ACTION_CANCEL);
    ev.setLocation(xc, yc);
    if (!target.dispatchTouchEvent(ev)) {
        // target didn't handle ACTION_CANCEL. not much we can do
        // but they should have.
    }
    // clear the target
    mMotionTarget = null;
    // Don't dispatch this event to our own view, because we already
    // saw it when intercepting; we just want to give the following
    // event to the normal onTouchEvent().
    return true;
}

```

当我们把 disallowIntercept 设置为 true 时，!disallowIntercept 直接为 false，于是拦截的方法体就被跳过了~

注 : 如果 ViewGroup 在 onInterceptTouchEvent(ev) ACTION_DOWN 里面直接 return true 了 , 那么子 View 是木有办法的捕获事件的~~~

4、如果没有找到合适的子 View

我们的实例 , 直接点击 ViewGroup 内的按钮 , 当然直接很顺利的走完整个流程 ;

但是有两种特殊情况

1、ACTION_DOWN 的时候 , 子 View.dispatchTouchEvent(ev) 返回的为 false ;

如果你仔细看了 , 你会注意到 ViewGroup 的 dispatchTouchEvent(ev) 的 ACTION_DOWN 代码是这样的

```
if (child.dispatchTouchEvent(ev)) {  
    // Event handled, we have a target now.  
    mMotionTarget = child;  
    return true;  
}
```

只有在 child.dispatchTouchEvent(ev) 返回 true 了 , 才会认为找到了能够处理当前事件的 View , 即 mMotionTarget = child;

但是如果返回 false, 那么 mMotionTarget 依然是 null

mMotionTarget 为 null 会咋样呢 ?

其实 ViewGroup 也是 View 的子类 , 如果没有找到能够处理该事件的子 View , 或者干脆就没有子 View ;

那么，它作为一个 View，就相当于 View 的事件转发了~~直接
super.dispatchTouchEvent(ev);

源码是这样的：

```
final View target = mMotionTarget;
if (target == null) {
    // We don't have a target, this means we're handling the
    // event as a regular view.
    ev.setLocation(xf, yf);
    if ((mPrivateFlags & CANCEL_NEXT_UP_EVENT) != 0) {
        ev.setAction(MotionEvent.ACTION_CANCEL);
        mPrivateFlags &= ~CANCEL_NEXT_UP_EVENT;
    }
    return super.dispatchTouchEvent(ev);
}
```

我们没有一个能够处理该事件的目标元素，意味着我们需要自己处理~~~就相当于传统的 View~

2、那么什么时候子 View.dispatchTouchEvent(ev)返回的为 true

如果你仔细看了上篇博客，你会发现只要子 View 支持点击或者长按事件一定返
回 true~~

源码是这样的：

```
if (((viewFlags & CLICKABLE) == CLICKABLE ||
     (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)) {
    return
true
;
}
```

5、总结

关于代码流程上面已经总结过了~

1、如果 `ViewGroup` 找到了能够处理该事件的 `View` , 则直接交给子 `View` 处理 ,

自己的 `onTouchEvent` 不会被触发 ;

2、可以通过复写 `onInterceptTouchEvent(ev)`方法 , 拦截子 `View` 的事件 (即

`return true`) , 把事件交给自己处理 , 则会执行自己对应的 `onTouchEvent` 方
法

3、子 `View` 可以通过调用

`getParent().requestDisallowInterceptTouchEvent(true);` 阻止
`ViewGroup` 对其 `MOVE` 或者 `UP` 事件进行拦截;

好了，那么实际应用中能解决哪些问题呢？

比如你需要写一个类似 `slidingmenu` 的左侧隐藏 `menu`，主 `Activity` 上有个 `Button`、`ListView` 或者任何可以响应点击的 `View`，你在当前 `View` 上死命的滑动，菜单栏也出不来；因为 `MOVE` 事件被子 `View` 处理了~ 你需要这么做：在 `ViewGroup` 的 `dispatchTouchEvent` 中判断用户是不是想显示菜单，如果是，则在 `onInterceptTouchEvent(ev)`拦截子 `View` 的事件；自己进行处理，这样自己的 `onTouchEvent` 就可以顺利展现出菜单栏了~~

文章、面试：讲讲 `Android` 的事件分发机制

面试场景

讲讲 `Android` 的事件分发机制？

基本会遵从 `Activity => ViewGroup => View` 的顺序进行事件分发，然后通过调用 `onTouchEvent()` 方法进行事件的处理。我们在项目中一般会对 `MotionEvent.ACTION_DOWN`, `MotionEvent.ACTION_UP`, `MotionEvent.ACTION_MOVE`, `MotionEvent.ACTION_CANCEL` 分情况进行操作。

有去查看源码中的事件拦截方法吗？或者说在进行事件分发的时候如何让正常的分发方式进行拦截？

我知道有个拦截事件的方法叫...叫，`onInterceptEvent()`？应该是，不过由于平时项目较多，确实没时间去关注太多源码。

厄，那你觉得在一个列表中，同时对父 View 和子 View 设置点击方法，优先响应哪个？为什么会这样？

肯定是优先响应子 View 的，至于为什么这样，平时知道这个结论，所以没去太深入研究，但我相信我简单看一下源码是肯定知道的。

先发表点扯淡

我们可能经常会遇到上面的这种情况，面试官希望了解我们知识的深入情况，或者说是平时学习欲望到底怎样。可很不幸的是，我搞 模拟面试 以来，80% 的小伙伴都属于开发能力不错，可对类似事件分发这样的基础问题一概不知。究其原因，除去忙以外，大多数小伙伴还是觉得平时开发也用不上什么，即使用到了，直接 Google 一下便能得到正确答案。

这大概就是很多人不会自定义 View 的原因吧，大多数效果在 GitHub 上都是现成的了，即使不太一样，也可以简单改改完事。

可很遗憾的是，我模拟面试那额外的 20% 的人，总拿到了令大多数人羡慕嫉妒恨的 offer，这不是没有原因的。可能别人就平时的开发中保持了更多的一点求知欲，就学到了很多至关重要的细节知识。

正文

还是不能偏题，其实这样的一个面试问题，确实是一个较为普遍的问题，我相信同类型的文章，网上一搜也是比比皆是，而且简单看一下关注度就能知道有多少人倒在了这种源码类型的面试上。

一般情况下，事件列都是从用户按下（ACTION_DOWN）的那一刻产生的，不得不提到，三个非常重要的与事件相关的方法。

- `dispatchTouchEvent()`
- `onTouchEvent()`
- `onInterceptTouchEvent()`

Activity 的事件分发机制

从英文单词中已经很明显的知道，`dispatchTouchEvent()` 是负责事件分发的。当点击事件产生后，事件首先会传递给当前的 Activity，这会调用 Activity 的 `dispatchTouchEvent()` 方法，我们来看看源码中是怎么处理的。

```
/**  
 * Called to process touch screen events. You can override this to  
 * intercept all touch screen events before they are dispatched to the  
 * window. Be sure to call this implementation for touch screen events  
 * that should be handled normally.  
 *  
 * @param ev The touch screen event.  
 *  
 * @return boolean Return true if this event was consumed.  
 */  
public boolean dispatchTouchEvent(MotionEvent ev) {  
    if (ev.getAction() == MotionEvent.ACTION_DOWN) {  
        // 由于事件开始一般都为 down 事件（按下）  
        // 所以一般都会调用该方法。  
        onUserInteraction();  
    }  
    if (getWindow().superDispatchTouchEvent(ev)) {  
        return true;  
        // 若 getWindow().superDispatchTouchEvent(ev) 返回 true  
        // 则 Activity.dispatchTouchEvent() 也返回 true, 停止事件传递  
    }  
    // 否则直接调用 onTouchEvent(ev)  
    return onTouchEvent(ev);  
}
```

注意截图中，我增加了一些注释，便于我们更加方便的理解，由于我们一般产生点击事件都是 MotionEvent.ACTION_DOWN，所以一般都会调用到 onUserInteraction() 这个方法。我们不妨来看看都做了什么。

```
/**  
 * Called whenever a key, touch, or trackball event is dispatched to the  
 * activity. Implement this method if you wish to know that the user has  
 * interacted with the device in some way while your activity is running.  
 * This callback and {@link #onUserLeaveHint} are intended to help  
 * activities manage status bar notifications intelligently; specifically,  
 * for helping activities determine the proper time to cancel a notification.  
 *  
 * <p>All calls to your activity's {@link #onUserLeaveHint} callback will  
 * be accompanied by calls to {@link #onUserInteraction}. This  
 * ensures that your activity will be told of relevant user activity such  
 * as pulling down the notification pane and touching an item there.  
 *  
 * <p>Note that this callback will be invoked for the touch down action  
 * that begins a touch gesture, but may not be invoked for the touch-moved  
 * and touch-up actions that follow.  
 *  
 * @see #onUserLeaveHint()  
 */  
public void onUserInteraction() {
```

很遗憾，这个方法实现是空的，不过我们可以从注释和其他途径可以了解到，该方法主要的作用是实现**屏保功能**，并且当此 Activity 在栈顶的时候，触屏点击 Home、Back、Recent 键等都会触发这个方法。

再来看看第二个 if 语句，`getWindow().superDispatchTouchEvent()`，`getWindow()` 明显是获取 Window，由于 Window 是一个抽象类，所以我们能拿到其子类 PhoneWindow，我们直接看看 `PhoneWindow.superDispatchTouchEvent()` 到底做了什么操作。

```
@Override  
public boolean superDispatchTouchEvent(MotionEvent event) {  
    return mDecor.superDispatchTouchEvent(event);  
}
```

直接调用了 `DecorView` 的 `superDispatchTouchEvent()` 方法。`DecorView` 继承于 `FrameLayout`，作为顶层 View，是所有界面的父类。而 `FrameLayout` 作为 `ViewGroup` 的子类，所以直接调用了 `ViewGroup` 的 `dispatchTouchEvent()`。

ViewGroup 的事件分发机制

我们通过查看 `ViewGroup` 的 `dispatchTouchEvent()` 可以发现。

```
final int actionMasked = action & MotionEvent.ACTION_MASK;  
  
// Handle an initial down.  
if (actionMasked == MotionEvent.ACTION_DOWN) {  
    // Throw away all previous state when starting a new touch gesture.  
    // The framework may have dropped the up or cancel event for the previous gesture  
    // due to an app switch, ANR, or some other state change.  
    cancelAndClearTouchTargets(ev);  
    resetTouchState();  
}  
  
// Check for interception.  
final boolean intercepted;  
if (actionMasked == MotionEvent.ACTION_DOWN  
    || mFirstTouchTarget != null) {  
    final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;  
    if (!disallowIntercept) {  
        intercepted = onInterceptTouchEvent(ev);  
        ev.setAction(action); // restore action in case it was changed  
    } else {  
        intercepted = false;  
    }  
} else {  
    // There are no touch targets and this action is not an initial down  
    // so this view group continues to intercept touches.  
    intercepted = true;  
}  
  
// If intercepted, start normal event dispatch. Also if there is already  
// an event being dispatched, then don't do anything.
```

注意其中红框里面的代码，看注释也能知道，定义了一个 `boolean` 值变量 `intercept` 来表示是否要拦截事件。

其中采用到了 `onInterceptTouchEvent(ev)` 对 `intercept` 进行赋值。大多数情况下，`onInterceptTouchEvent()` 返回值为 `false`，但我们完全可以通过重写

`onInterceptTouchEvent(ev)` 来改变它的返回值，不妨继续往下看，我们后面对这个 `intercept` 做了什么处理。

```
boolean alreadyDispatchedToNewTouchTarget = false;
if (!canceled && !intercepted) {

    // If the event is targeting accessibility focus we give it to the
    // view that has accessibility focus and if it does not handle it
    // we clear the flag and dispatch the event to all children as usual.
    // We are looking up the accessibility focused host to avoid keeping
    // state since these events are very rare.
    View childWithAccessibilityFocus = ev.isTargetAccessibilityFocus()
        ? findChildWithAccessibilityFocus() : null;

    if (actionMasked == MotionEvent.ACTION_DOWN
        || (split && actionMasked == MotionEvent.ACTION_POINTER_DOWN)
        || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
        final int actionBarIndex = ev.getActionIndex(); // always 0 for down
        final int idBitsToAssign = split ? 1 << ev.getPointerId(actionIndex)
            : TouchTarget.ALL_POINTER_IDS;

        // Clean up earlier touch targets for this pointer id in case they
        // have become out of sync.
        removePointersFromTouchTargets(idBitsToAssign);

        final int childrenCount = mChildrenCount;
        if (newTouchTarget == null && childrenCount != 0) {
            final float x = ev.getX(actionIndex);
            final float y = ev.getY(actionIndex);
            // Find a child that can receive the event.
            // Scan children from front to back.
            final ArrayList<View> preorderedList = buildTouchDispatchChildList();
            final boolean customOrder = preorderedList == null
                && isChildrenDrawingOrderEnabled();
        }
    }
}
```

暂时忽略 判断的 `canceled`，该值同样大多数时候都返回 `false`，所以当我们没有重写 `onInterceptTouchEvent()` 并使它的返回值为 `true` 时，一般情况下都是可以进入到该方法的。

继续阅读源码可以发现，里面做了一个 `For` 循环，通过倒序遍历 `ViewGroup` 下面的所有子 `View`，然后一个一个判断点击位置是否是该子 `View` 的布局区域，当然还有一些其他的，由于篇幅原因，这里就不细讲了。

View 的事件分发机制

`ViewGroup` 说到底还是一个 `View`，所以我们不得不继续看看 `View` 的 `dispatchTouchEvent()`。

```
/*
 * Pass the touch screen motion event down to the target view, or this
 * view if it is the target.
 *
 * @param event The motion event to be dispatched.
 * @return True if the event was handled by the view, false otherwise.
 */
public boolean dispatchTouchEvent(MotionEvent event) {
    // If the event should be handled by accessibility focus first.
    // ...
    boolean result = false;
    //...
    if (onFilterTouchEventForSecurity(event)) {
        // ...
        //noinspection SimplifiableIfStatement
        ListenerInfo li = mListenerInfo;

        // 必须满足三个条件都为真，才会返回 true
        // 1、mOnTouchListener 不为 null，即调用了 setOnTouchListener()
        // 2、(mViewFlags & ENABLED_MASK) == ENABLED
        // 3、li.mOnTouchListener.onTouch(this, event)
        if (li != null && li.mOnTouchListener != null
            && (mViewFlags & ENABLED_MASK) == ENABLED
            && li.mOnTouchListener.onTouch(this, event)) {
            result = true;
        }
        //...
    }
    //...
    return result;
}
```

截图中的代码是有删减的，我们重点看看没有删减的代码。

红框中的三个条件，第一个我就不用说了。

•

(mViewFlags & ENABLED_MASK) == ENABLED

该条件是判断当前点击的控件是否为 enable，但由于基本 View 都是 enable 的，所以这个条件基本都返回 true。

•

•

mOnTouchListener.onTouch(this, event)

即我们调用 `setOnTouchListener()` 时必须覆盖的方法 `onTouch()` 的返回值。

•

从上述的分析，终于知道「`onTouch()` 方法优先级高于 `onTouchEvent(event)` 方法」是怎么来的了吧。

再来看看 `onTouchEvent()`

```
* If this method is used to detect click actions, it is recommended that
* the actions be performed by implementing and calling
* {@link #performClick()}. This will ensure consistent system behavior,
* including:
* <ul>
* <li>obeying click sound preferences
* <li>dispatching OnClickListener calls
* <li>handling {@link AccessibilityNodeInfo#ACTION_CLICK ACTION_CLICK} when
* accessibility features are enabled
* </ul>
*
* @param event The motion event.
* @return True if the event was handled, false otherwise.
*/
public boolean onTouchEvent(MotionEvent event) {
    final float x = event.getX();
    final float y = event.getY();
    final int viewFlags = mViewFlags;
    final int action = event.getAction();

    final boolean clickable = ((viewFlags & CLICKABLE) == CLICKABLE
        || (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)
        || (viewFlags & CONTEXT_CLICKABLE) == CONTEXT_CLICKABLE;
    // ...

    if (clickable || (viewFlags & TOOLTIP) == TOOLTIP) {
        switch (action) {
            case MotionEvent.ACTION_UP:
                //...
                boolean prepressed = (mPrivateFlags & PFLAG_PREPRESSED) != 0;
                if ((mPrivateFlags & PFLAG_PRESSED) != 0 || prepressed) {
                    // take focus if we don't have it already and we should in
                    // touch mode.
                    boolean focusTaken = false;
                    //...
                    if (!mHasPerformedLongPress && !mIgnoreNextUpEvent) {

```

```
        if ((mHasPress) || mIsLongPress && !mIgnoreNextUpEvent) {
            // This is a tap, so remove the longpress check
            removeLongPressCallback();

            // Only perform take click actions if we were in the press
            if (!focusTaken) {
                // Use a Runnable and post this rather than calling
                // performClick directly. This lets other visual state
                // of the view update before click actions start.
                if (mPerformClick == null) {
                    mPerformClick = new PerformClick();
                }
                if (!post(mPerformClick)) {
                    performClick();
                }
            }
        }
        // ...
    }
    mIgnoreNextUpEvent = false;
    break;
}
//...
}
return true;
}
return false;
```

从上面的代码可以明显地看到，只要 View 的 CLICKABLE 和 LONG_CLICKABLE 有一个为 true，那么 onTouchEvent() 就会返回 true 消耗这个事件。CLICKABLE 和 LONG_CLICKABLE 代表 View 可以被点击和长按点击，我们通常都会采用 setOnClickListener() 和 setOnLongClickListener() 做设置。接着在 ACTION_UP 事件中会调用 performClick() 方法，我们看看都做了什么。

```
/*
 * Call this view's OnClickListener, if it is defined. Performs all normal
 * actions associated with clicking: reporting accessibility event, playing
 * a sound, etc.
 *
 * @return True there was an assigned OnClickListener that was called, false
 *         otherwise is returned.
 */
public boolean performClick() {
    final boolean result;
    final ListenerInfo li = mListenerInfo;
    if (li != null && li.mOnClickListener != null) {
        playSoundEffect(SoundEffectConstants.CLICK);
        li.mOnClickListener.onClick(this);
        result = true;
    } else {
        result = false;
    }

    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);

    notifyEnterOrExitForAutoFillIfNeeded(true);

    return result;
}
```

从截图中可以看到，如果 `mOnClickListener` 不为空，那么它的 `onClick()` 方法就会调用。

总结

本来写到这就结束了，但回顾一遍还是打算给大家稍微总结一下。

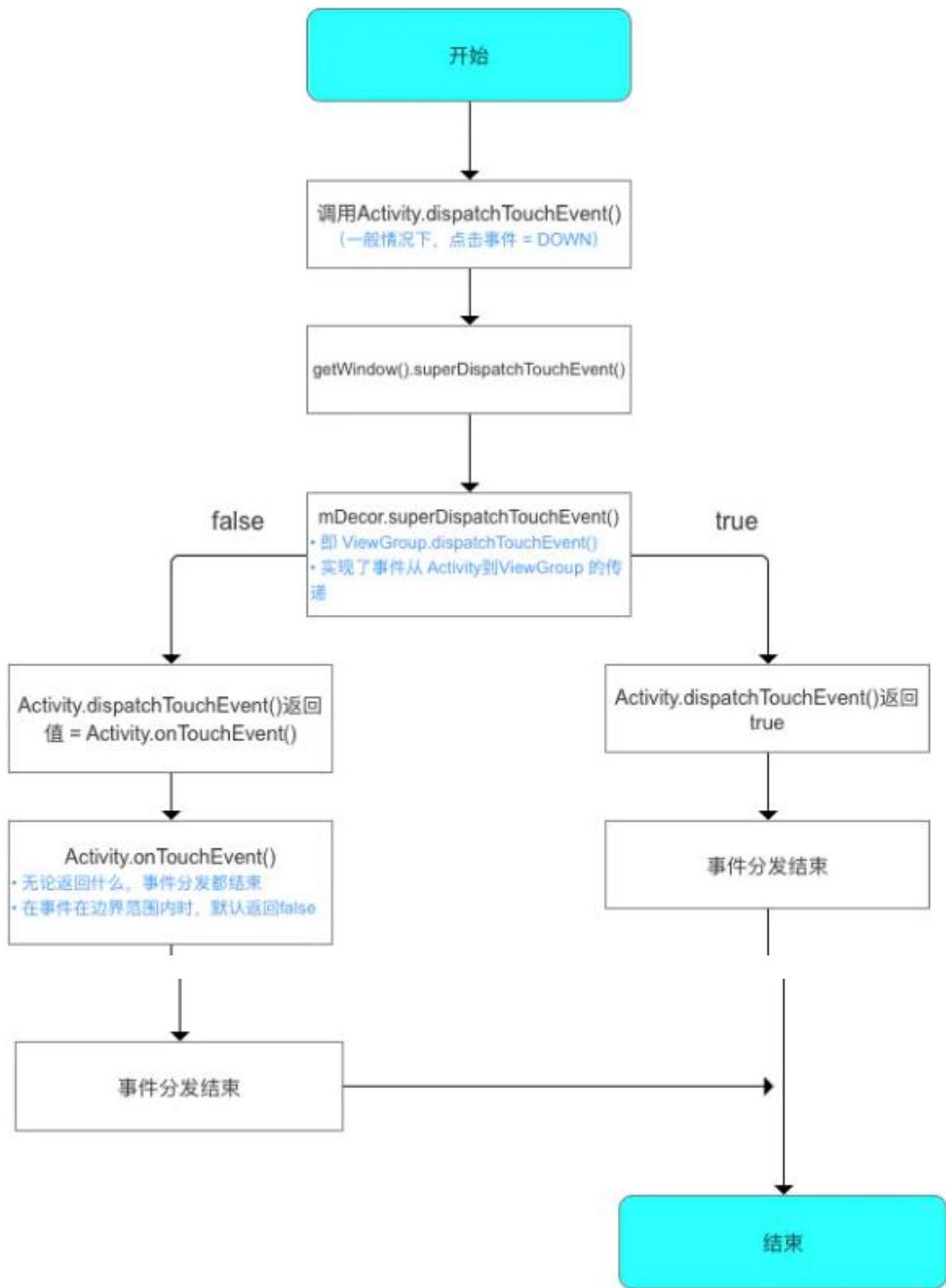
需要总结的小点：

- 1、Android 事件分发总是遵循 `Activity => ViewGroup => View` 的传递顺序；
- 2、`onTouch()` 执行总优先于 `onClick()`

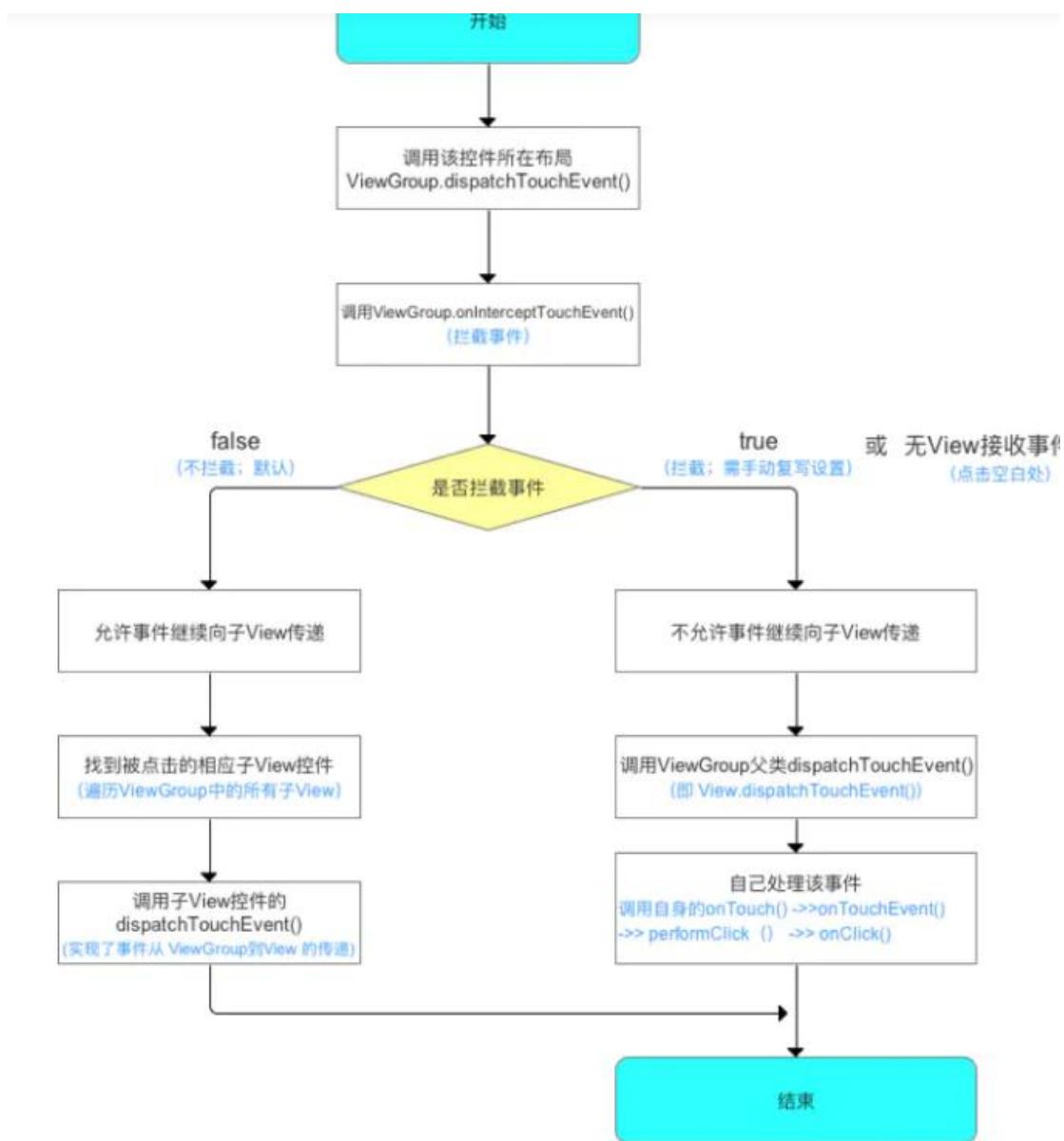
原本想用文字总结的，结果发现简书上还有这样一篇神文：Android 事件分发机制详解：史上最全面、最易懂，所以直接引用一下其中的图片。

•

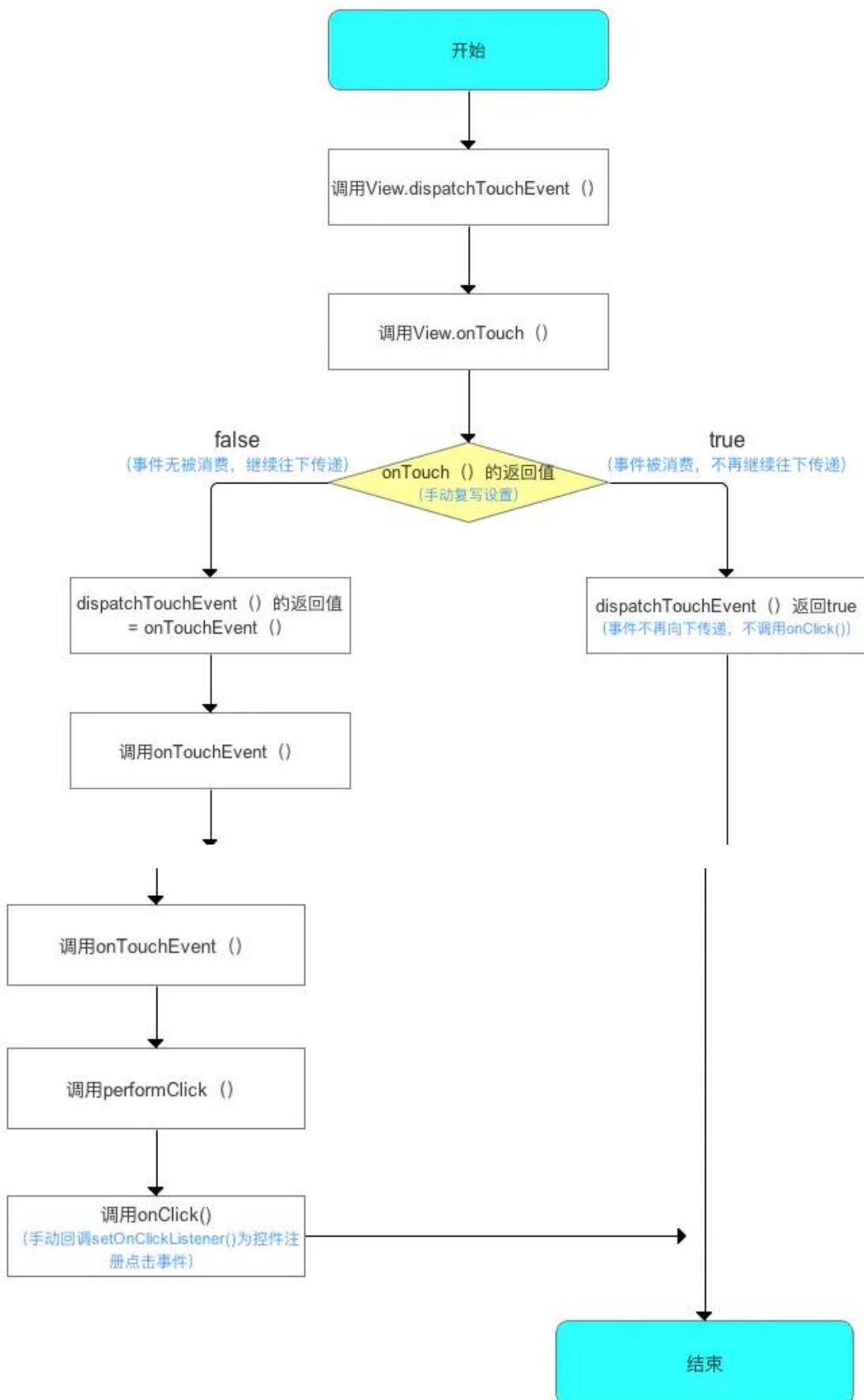
Activity 的事件分发示意图



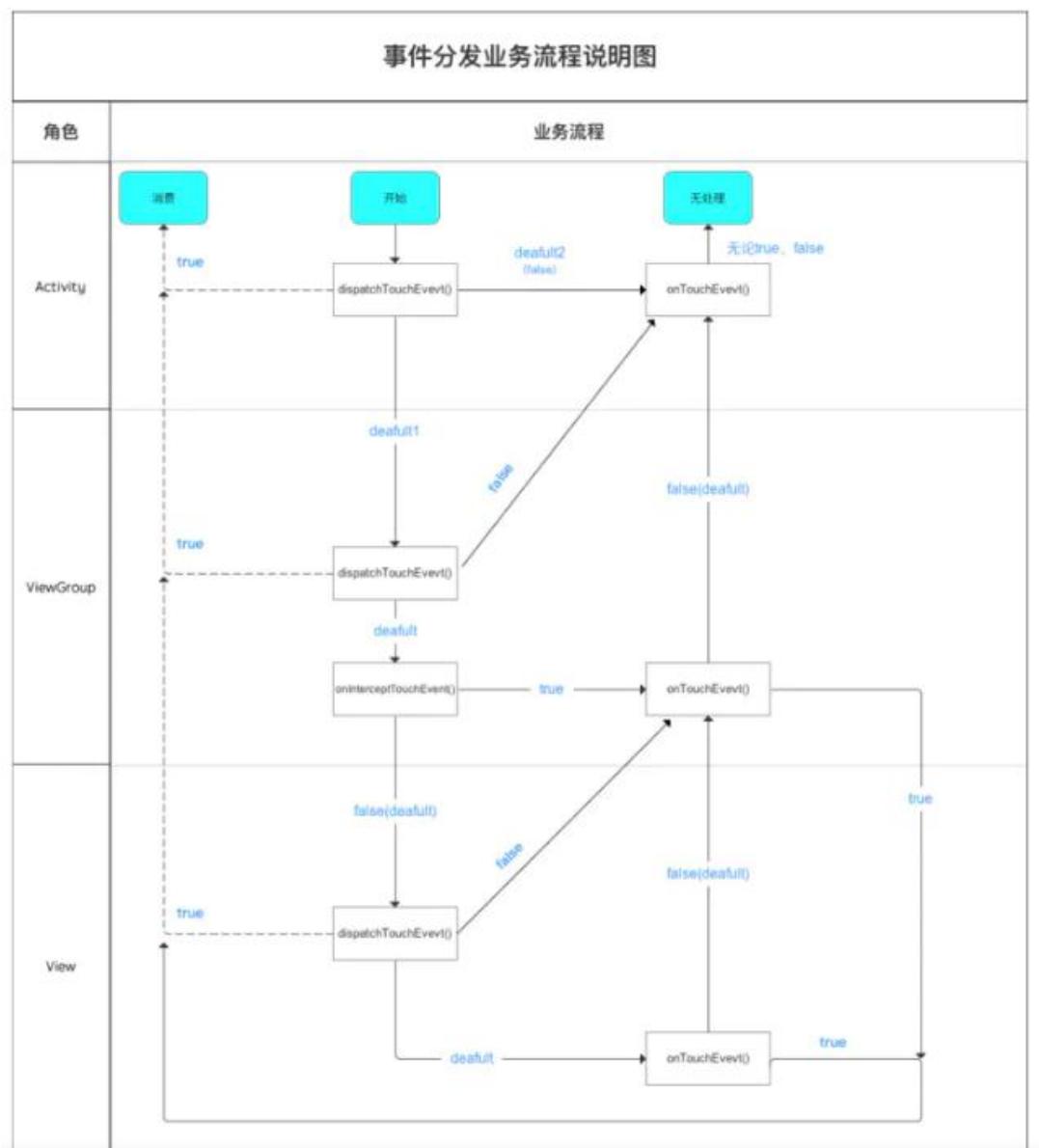
ViewGroup 事件分发示意图



View 的事件分发示意图



事件分发工作流程总结



九、Android View 绘制流程

1、简述 View 绘制流程

2、onMeasure, onlayout, ondraw 方法中需要注意的点

3、如何进行自定义 View

4、view 重绘机制

文章、Android LayoutInflater 原理分析，带你一步步深入了解 View(一)

先来看一下 LayoutInflater 的基本用法吧，它的用法非常简单，首先需要获取到 LayoutInflater 的实例，有两种方法可以获取到，第一种写法如下：

```
LayoutInflater layoutInflater = LayoutInflater.from(context);
```

当然，还有另外一种写法也可以完成同样的效果：

```
1 LayoutInflater layoutInflater = (LayoutInflater) context  
2         .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
```

其实第一种就是第二种的简单写法，只是 Android 给我们做了一下封装而已。得到了 LayoutInflater 的实例之后就可以调用它的 inflate()方法来加载布局了，如下所示：

```
layoutInflater.inflate(resourceId, root);
```

inflate()方法一般接收两个参数，第一个参数就是要加载的布局 id，第二个参数是指给该布局的外部再嵌套一层父布局，如果不需要就直接传 null。这样就成功成功创建了一个布局的实例，之后再将它添加到指定的位置就可以显示出来了。

下面我们就通过一个非常简单的小例子，来更加直观地看一下 LayoutInflater 的用法。比如说当前有一个项目，其中 MainActivity 对应的布局文件叫做 activity_main.xml，代码如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:id="@+id/main_layout"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent" >  
  
</LinearLayout>
```

这个布局文件的内容非常简单，只有一个空的 LinearLayout，里面什么控件都没有，因此界面上应该不会显示任何东西。

那么接下来我们再定义一个布局文件，给它取名为 button_layout.xml，代码如下所示：

```
<Button xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Button" >  
  
</Button>
```

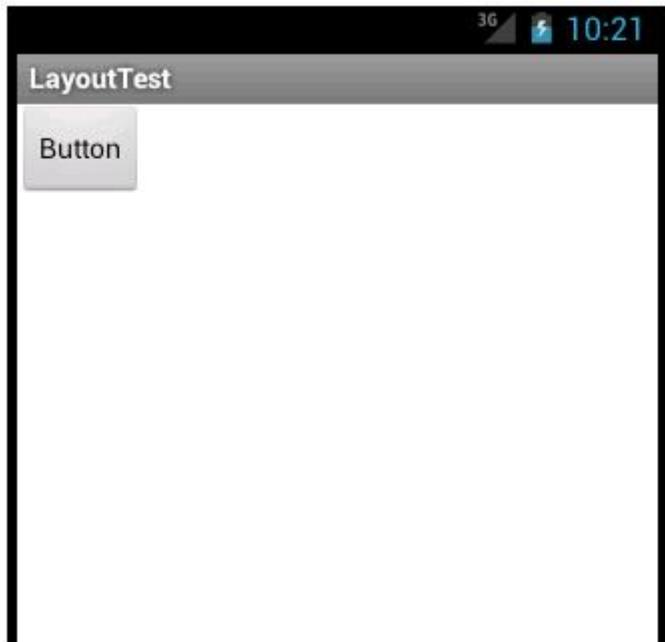
这个布局文件也非常简单，只有一个 Button 按钮而已。现在我们要想办法，如何通过 LayoutInflater 来将 button_layout 这个布局添加到主布局文件的 LinearLayout 中。根据刚刚介绍的用法，修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {  
  
    private LinearLayout mainLayout;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        mainLayout = (LinearLayout) findViewById(R.id.main_layout);  
        LayoutInflator layoutInflater = LayoutInflator.from(this);  
        View buttonLayout = layoutInflater.inflate(R.layout.button_layout, null);  
        mainLayout.addView(buttonLayout);  
    }  
}
```

```
}
```

可以看到，这里先是获取到了 LayoutInflator 的实例，然后调用它的 inflate() 方法来加载 button_layout 这个布局，最后调用 LinearLayout 的 addView() 方法将它添加到 LinearLayout 中。

现在可以运行一下程序，结果如下图所示：



Button 在界面上显示出来了！说明我们确实是借助 LayoutInflator 成功将 button_layout 这个布局添加到 LinearLayout 中了。LayoutInflater 技术广泛应用于需要动态添加 View 的时候，比如在 ScrollView 和 ListView 中，经常都可以看到 LayoutInflater 的身影。

当然，仅仅只是介绍了如何使用 LayoutInflater 显然是远远无法满足大家的求知欲的，知其然也要知其所以然，接下来我们就从源码的角度上看一看 LayoutInflater 到底是如何工作的。

不管你是使用的哪个 inflate()方法的重载，最终都会辗转调用到 LayoutInflator 的如下代码中：

```
public View inflate(XmlPullParser parser, ViewGroup root, boolean  
attachToRoot) {  
  
    synchronized (mConstructorArgs) {  
  
        final AttributeSet attrs = Xml.asAttributeSet(parser);  
  
        mConstructorArgs[0] = mContext;  
  
        View result = root;  
  
        try {  
  
            int type;  
  
            while ((type = parser.next()) != XmlPullParser.START_T  
                &&  
                type != XmlPullParser.END_DOCUMENT) {  
  
                if (type == XmlPullParser.TEXT) {  
                    String text = parser.getText();  
                    if (text != null && !text.equals("")) {  
                        result.setText(text);  
                    }  
                }  
            }  
        } catch (Exception e) {  
            Log.e("Inflate", "Error inflating view", e);  
        }  
    }  
    return result;  
}
```

```
    }

    if (type != XmlPullParser.START_TAG) {

        throw new
InflateException(parser.getPositionDescription()

                + ": No start tag found!");

    }

    final String name = parser.getName();

    if (TAG_MERGE.equals(name)) {

        if (root == null || !attachToRoot) {

            throw new InflateException("merge can be used
only with a valid "

                + "ViewGroup root and

attachToRoot=true");

        }

        rInflate(parser, root, attrs);

    } else {

        View temp = createViewFromTag(name, attrs);

    }

}
```

```
ViewGroup.LayoutParams params = null;

if (root != null) {

    params = root.generateLayoutParams(attrs);

    if (!attachToRoot) {

        temp.setLayoutParams(params);

    }

}

rInflate(parser, temp, attrs);

if (root != null && attachToRoot) {

    root.addView(temp, params);

}

if (root == null || !attachToRoot) {

    result = temp;

}

}

} catch (XmlPullParserException e) {
```

```
InflateException ex = new InflateException(e.getMessage());  
  
ex.initCause(e);  
  
throw ex;  
  
} catch (IOException e) {  
  
    InflateException ex = new InflateException(  
  
        parser.getPositionDescription()  
  
        + ": " + e.getMessage());  
  
    ex.initCause(e);  
  
    throw ex;  
  
}  
  
return result;  
  
}  
  
}
```

从这里我们就可以清楚地看出，`LayoutInflater` 其实就是使用 Android 提供的 `pull` 解析方式来解析布局文件的。不熟悉 `pull` 解析方式的朋友可以网上搜一下，教程很多，我就不细讲了，这里我们注意看下第 23 行，调用了 `createViewFromTag()` 这个方法，并把节点名和参数传了进去。看到这个方法名，我们就应该能猜到，它是用于根据节点名来创建 `View` 对象的。确实如此，在 `createViewFromTag()` 方法的内部又会去调用 `createView()` 方法，然后使用反射的方式创建出 `View` 的实例并返回。

当然，这里只是创建出了一个根布局的实例而已，接下来会在第 31 行调用 `rInflate()` 方法来循环遍历这个根布局下的子元素，代码如下所示：

```
private void rInflate(XmlPullParser parser, View parent, final AttributeSet attrs)
throws XmlPullParserException, IOException {

    final int depth = parser.getDepth();

    int type;

    while (((type = parser.next()) != XmlPullParser.END_TAG ||
            parser.getDepth() > depth) && type !=
            XmlPullParser.END_DOCUMENT) {

        if (type != XmlPullParser.START_TAG) {

            continue;
        }

        final String name = parser.getName();

        if (TAG_REQUEST_FOCUS.equals(name)) {

            parseRequestFocus(parser, parent);
        } else if (TAG_INCLUDE.equals(name)) {
```

```
        if (parser.getDepth() == 0) {

            throw new InflateException("<include /> cannot be the
root element");

        }

        parseInclude(parser, parent, attrs);

    } else if (TAG_MERGE.equals(name)) {

        throw new InflateException("<merge /> must be the root
element");

    } else {

        final View view = createViewFromTag(name, attrs);

        final ViewGroup viewGroup = (ViewGroup) parent;

        final ViewGroup.LayoutParams params =
viewGroup.generateLayoutParams(attrs);

        rInflate(parser, view, attrs);

        viewGroup.addView(view, params);

    }

}
```

```
parent.onFinishInflate();  
}  
}
```

可以看到，在第 21 行同样是 `createViewFromTag()` 方法来创建 `View` 的实例，然后还会在第 24 行递归调用 `rInflate()` 方法来查找这个 `View` 下的子元素，每次递归完成后则将这个 `View` 添加到父布局当中。

这样的话，把整个布局文件都解析完成后就形成了一个完整的 DOM 结构，最终会把最顶层的根布局返回，至此 `inflate()` 过程全部结束。

比较细心的朋友也许会注意到，`inflate()` 方法还有个接收三个参数的方法重载，结构如下：

```
inflate(XmlPullParser parser, boolean attachToRoot, View root);
```

那么这第三个参数 `attachToRoot` 又是什么意思呢？其实如果你仔细去阅读上面的源码应该可以自己分析出答案，这里我先将结论说一下吧，感兴趣的朋友可以再阅读一下源码，校验我的结论是否正确。

1. 如果 `root` 为 `null`，`attachToRoot` 将失去作用，设置任何值都没有意义。
2. 如果 `root` 不为 `null`，`attachToRoot` 设为 `true`，则会给加载的布局文件的指定一个父布局，即 `root`。

3. 如果 `root` 不为 `null`, `attachToRoot` 设为 `false`, 则会将布局文件最外层的所有 `layout` 属性进行设置, 当该 `view` 被添加到父 `view` 当中时, 这些 `layout` 属性会自动生效。
4. 在不设置 `attachToRoot` 参数的情况下, 如果 `root` 不为 `null`, `attachToRoot` 参数默认为 `true`。

好了, 现在对 `LayoutInflater` 的工作原理和流程也搞清楚了, 你该满足了吧。额。。。还嫌这个例子中的按钮看起来有点小, 想要调大一些? 那简单的呀, 修改 `button_layout.xml` 中的代码, 如下所示:

```
<Button xmlns:android="http://schemas.android.com/apk/res/android"  
        android:layout_width="300dp"  
        android:layout_height="80dp"  
        android:text="Button" >  
  
</Button>
```

这里我们将按钮的宽度改成 `300dp`, 高度改成 `80dp`, 这样够大了吧? 现在重新运行一下程序来观察效果。咦? 怎么按钮还是原来的大小, 没有任何变化! 是不是按钮仍然不够大, 再改大一点呢? 还是没有用!

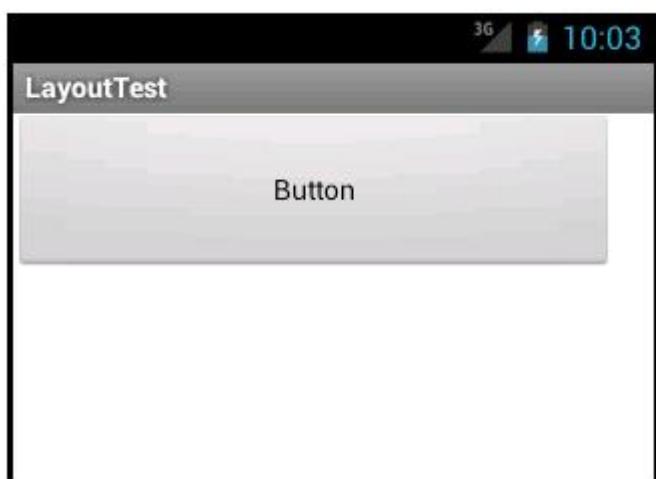
其实这里不管你将 **Button** 的 **layout_width** 和 **layout_height** 的值修改成多少，都不会有任何效果的，因为这两个值现在已经完全失去了作用。平时我们经常使用 **layout_width** 和 **layout_height** 来设置 **View** 的大小，并且一直都能正常工作，就好像这两个属性确实是用于设置 **View** 的大小的。而实际上则不然，它们其实是用于设置 **View** 在布局中的大小的，也就是说，首先 **View** 必须存在于一个布局中，之后如果将 **layout_width** 设置成 **match_parent** 表示让 **View** 的宽度填充满布局，如果设置成 **wrap_content** 表示让 **View** 的宽度刚好可以包含其内容，如果设置成具体的数值则 **View** 的宽度会变成相应的数值。这也是为什么这两个属性叫作 **layout_width** 和 **layout_height** 而不是 **width** 和 **height**。

再来看一下我们的 **button_layout.xml** 吧，很明显 **Button** 这个控件目前不存在于任何布局当中，所以 **layout_width** 和 **layout_height** 这两个属性理所当然没有任何作用。那么怎样修改才能让按钮的大小改变呢？解决方法其实有很多，最简单的方式就是在 **Button** 的外面再嵌套一层布局，如下所示：

```
<RelativeLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
  
        android:layout_width="match_parent"  
  
        android:layout_height="match_parent" >  
  
<Button  
    android:layout_width="300dp"  
  
    android:layout_height="80dp"
```

```
    android:text="Button" >  
  
  </Button>  
  
</RelativeLayout>
```

可以看到，这里我们又加入了一个 `RelativeLayout`，此时的 `Button` 存在与 `RelativeLayout` 之中，`layout_width` 和 `layout_height` 属性也就有作用了。当然，处于最外层的 `RelativeLayout`，它的 `layout_width` 和 `layout_height` 则会失去作用。现在重新运行一下程序，结果如下图所示：



OK！按钮的终于可以变大了，这下总算是满足大家的要求了吧。

看到这里，也许有些朋友心中会有一个巨大的疑惑。不对呀！平时在 `Activity` 中指定布局文件的时候，最外层的那个布局是可以指定大小的呀，`layout_width` 和 `layout_height` 都是有作用的。确实，这主要是因为，在 `setContentView()` 方法中，Android 会自动在布局文件的最外层再嵌套一个 `FrameLayout`，所以

layout_width 和 layout_height 属性才会有效果。那么我们来证实一下吧，修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {  
  
    private LinearLayout mainLayout;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        mainLayout = (LinearLayout)  
                findViewById(R.id.main_layout);  
        ViewParent viewParent = mainLayout.getParent();  
        Log.d("TAG", "the parent of mainLayout is " + viewParent);  
    }  
}
```

可以看到，这里通过 findViewById()方法，拿到了 activity_main 布局中最外层的 LinearLayout 对象，然后调用它的 getParent()方法获取它的父布局，再通过 Log 打印出来。现在重新运行一下程序，结果如下图所示：

Tag	Text
TAG	the parent of mainLayout is android.widget.FrameLayout@41069ff8 http://blog.csdn.net/guolin_blog

非常正确！`LinearLayout` 的父布局确实是一个 `FrameLayout`，而这个 `FrameLayout` 就是由系统自动帮我们添加上的。

说到这里，虽然 `setContentView()` 方法大家都会用，但实际上 Android 界面显示的原理要比我们所看到的东西复杂得多。任何一个 `Activity` 中显示的界面其实主要都由两部分组成，标题栏和内容布局。标题栏就是在很多界面顶部显示的那部分内容，比如刚刚我们的那个例子当中就有标题栏，可以在代码中控制让它是否显示。而内容布局就是一个 `FrameLayout`，这个布局的 `id` 叫作 `content`，我们调用 `setContentView()` 方法时所传入的布局其实就是放到这个 `FrameLayout` 中的，这也是为什么这个方法名叫作 `setContentView()`，而不是叫 `setView()`。

最后再附上一张 `Activity` 窗口的组成图吧，以便于大家更加直观地理解：



http://blog.csdn.net/guolin_blog

文章、Android 视图绘制流程完全解析，带你一步步深入了解 View(二)

一. onMeasure()

measure 是测量的意思，那么 onMeasure()方法顾名思义就是用于测量视图的大小的。View 系统的绘制流程会从 ViewRoot 的 performTraversals()方法中开始，在其内部调用 View 的 measure()方法。measure()方法接收两个参数，widthMeasureSpec 和 heightMeasureSpec ,这两个值分别用于确定视图的宽度和高度的规格和大小。

MeasureSpec 的值由 specSize 和 specMode 共同组成的，其中 specSize 记录的是大小，specMode 记录的是规格。specMode 一共有三种类型，如下所示：

1. EXACTLY

表示父视图希望子视图的大小应该是由 specSize 的值来决定的，系统默认会按照这个规则来设置子视图的大小，开发人员当然也可以按照自己的意愿设置成任意的大小。

2. AT_MOST

表示子视图最多只能是 specSize 中指定的大小，开发人员应该尽可能小得去设置这个视图，并且保证不会超过 specSize。系统默认会按照这个规则来设置子视图的大小，开发人员当然也可以按照自己的意愿设置成任意的大小。

3. UNSPECIFIED

表示开发人员可以将视图按照自己的意愿设置成任意的大小，没有任何限制。这种情况比较少见，不太会用到。

那么你可能会有疑问了，widthMeasureSpec 和 heightMeasureSpec 这两个值又是从哪里得到的呢？通常情况下，这两个值都是由父视图经过计算后传递给子视图的，说明父视图会在一定程度上决定子视图的大小。但是最外层的根视图，

它的 widthMeasureSpec 和 heightMeasureSpec 又是从哪里得到的呢？这就需要去分析 ViewRoot 中的源码了，观察 performTraversals()方法可以发现如下代码：

```
1 childWidthMeasureSpec = getRootMeasureSpec(desiredWindowWidth, lp.width);
2 childHeightMeasureSpec = getRootMeasureSpec(desiredWindowHeight, lp.height);
```

复制

可以看到，这里调用了 getRootMeasureSpec()方法去获取 widthMeasureSpec 和 heightMeasureSpec 的值，注意方法中传入的参数，其中 lp.width 和 lp.height 在创建 ViewGroup 实例的时候就被赋值了，它们都等于 MATCH_PARENT。然后看下 getRootMeasureSpec()方法中的代码，如下所示：

```
1 private int getRootMeasureSpec(int windowHeight, int rootDimension) {
2     int measureSpec;
3     switch (rootDimension) {
4         case ViewGroup.LayoutParams.MATCH_PARENT:
5             measureSpec = MeasureSpec.makeMeasureSpec(windowHeight, MeasureSpec.EXACTLY);
6             break;
7         case ViewGroup.LayoutParams.WRAP_CONTENT:
8             measureSpec = MeasureSpec.makeMeasureSpec(windowHeight, MeasureSpec.AT_MOST);
9             break;
10        default:
11            measureSpec = MeasureSpec.makeMeasureSpec(rootDimension, MeasureSpec.EXACTLY);
12            break;
13        }
14    return measureSpec;
15 }
```

复制

可以看到，这里使用了 MeasureSpec.makeMeasureSpec()方法来组装一个 MeasureSpec，当 rootDimension 参数等于 MATCH_PARENT 的时候，MeasureSpec 的 specMode 就等于 EXACTLY，当 rootDimension 等于 WRAP_CONTENT 的时候，MeasureSpec 的 specMode 就等于 AT_MOST。并且 MATCH_PARENT 和 WRAP_CONTENT 时的 specSize 都是等于 windowHeight 的，也就意味着根视图总是会充满全屏的。

介绍了这么多 MeasureSpec 相关的内容，接下来我们看下 View 的 measure() 方法里面的代码吧，如下所示：

```
public final void measure(int widthMeasureSpec, int heightMeasureSpec) {  
    if ((mPrivateFlags & FORCE_LAYOUT) == FORCE_LAYOUT ||  
        widthMeasureSpec != mOldWidthMeasureSpec ||  
        heightMeasureSpec != mOldHeightMeasureSpec) {  
  
        mPrivateFlags &= ~MEASURED_DIMENSION_SET;  
  
        if (ViewDebug.TRACE_HIERARCHY) {  
  
            ViewDebug.trace(this, ViewDebug.HierarchyTraceType.ON_MEASURE);  
  
        }  
  
        onMeasure(widthMeasureSpec, heightMeasureSpec);  
  
        if ((mPrivateFlags & MEASURED_DIMENSION_SET) !=  
            MEASURED_DIMENSION_SET) {  
  
            throw new IllegalStateException("onMeasure() did not set the"  
                + " measured dimension by calling"  
                + " setMeasuredDimension()");  
  
        }  
  
        mPrivateFlags |= LAYOUT_REQUIRED;  
  
    }  
  
    mOldWidthMeasureSpec = widthMeasureSpec;  
  
    mOldHeightMeasureSpec = heightMeasureSpec;  
}
```

注意观察，`measure()`这个方法是 `final` 的，因此我们无法在子类中去重写这个方法，说明 Android 是不允许我们改变 `View` 的 `measure` 框架的。然后在第 9 行调用了 `onMeasure()` 方法，这里才是真正去测量并设置 `View` 大小的地方，默认会调用 `getDefaultSize()` 方法来获取视图的大小，如下所示：

```
public static int getDefaultSize(int size, int measureSpec) {  
  
    int result = size;  
  
    int specMode = MeasureSpec.getMode(measureSpec);  
  
    int specSize = MeasureSpec.getSize(measureSpec);  
  
    switch (specMode) {  
  
        case MeasureSpec.UNSPECIFIED:  
  
            result = size;  
  
            break;  
  
        case MeasureSpec.AT_MOST:  
  
        case MeasureSpec.EXACTLY:  
  
            result = specSize;  
  
            break;  
  
    }  
  
    return result;  
}
```

这里传入的 `measureSpec` 是一直从 `measure()` 方法中传递过来的。然后调用 `MeasureSpec.getMode()` 方法可以解析出 `specMode`，调用 `MeasureSpec.getSize()` 方法可以解析出 `specSize`。接下来进行判断，如果 `specMode` 等于 `AT_MOST` 或 `EXACTLY` 就返回 `specSize`，这也是系统默认的

行为。之后会在 `onMeasure()`方法中调用 `setMeasuredDimension()`方法来设定测量出的大小，这样一次 `measure` 过程就结束了。

当然，一个界面的展示可能会涉及到很多次的 `measure`，因为一个布局中一般都会包含多个子视图，每个视图都需要经历一次 `measure` 过程。`ViewGroup` 中定义了一个 `measureChildren()`方法来去测量子视图的大小，如下所示：

```
protected void measureChildren(int widthMeasureSpec, int heightMeasureSpec) {  
    final int size = mChildrenCount;  
    final View[] children = mChildren;  
    for (int i = 0; i < size; ++i) {  
        final View child = children[i];  
        if ((child.mViewFlags & VISIBILITY_MASK) != GONE) {  
            measureChild(child, widthMeasureSpec, heightMeasureSpec);  
        }  
    }  
}
```

这里首先会去遍历当前布局下的所有子视图，然后逐个调用 `measureChild()`方法来测量相应子视图的大小，如下所示：

```
protected void measureChild(View child, int parentWidthMeasureSpec,  
    int parentHeightMeasureSpec) {  
    final LayoutParams lp = child.getLayoutParams();
```

```
final int childWidthMeasureSpec = getChildMeasureSpec(parentWidthMeasureSpec,
    mPaddingLeft + mPaddingRight, lp.width);

final int childHeightMeasureSpec =
getChildMeasureSpec(parentHeightMeasureSpec,
    mPaddingTop + mPaddingBottom, lp.height);

child.measure(childWidthMeasureSpec, childHeightMeasureSpec);

}
```

可以看到，在第 4 行和第 6 行分别调用了 `getChildMeasureSpec()` 方法来去计算子视图的 `MeasureSpec`，计算的依据就是布局文件中定义的 `MATCH_PARENT`、`WRAP_CONTENT` 等值，这个方法的内部细节就不再贴出。然后在第 8 行调用子视图的 `measure()` 方法，并把计算出的 `MeasureSpec` 传递进去，之后的流程就和前面所介绍的一样了。

当然，`onMeasure()` 方法是可以重写的，也就是说，如果你不想使用系统默认的测量方式，可以按照自己的意愿进行定制，比如：

```
public class MyView extends View {
    .....
    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
```

```
    setMeasuredDimension(200, 200);  
  
}  
  
}
```

这样的话就把 View 默认的测量流程覆盖掉了，不管在布局文件中定义 MyView 这个视图的大小是多少，最终在界面上显示的大小都将会是 200*200。

需要注意的是，在 setMeasuredDimension()方法调用之后，我们才能使用 getMeasuredWidth() 和 getMeasuredHeight() 来获取视图测量出的宽高，以此之前调用这两个方法得到的值都会是 0。

由此可见，视图大小的控制是由父视图、布局文件、以及视图本身共同完成的，父视图会提供给子视图参考的大小，而开发人员可以在 XML 文件中指定视图的大小，然后视图本身会对最终的大小进行拍板。

到此为止，我们就把视图绘制流程的第一阶段分析完了。

二. onLayout()

measure 过程结束后，视图的大小就已经测量好了，接下来就是 layout 的过程了。正如其名字所描述的一样，这个方法是用于给视图进行布局的，也就是确定视图的位置。ViewRoot 的 performTraversals()方法会在 measure 结束后继续执行，并调用 View 的 layout()方法来执行此过程，如下所示：

0 0

layout()方法接收四个参数，分别代表着左、上、右、下的坐标，当然这个坐标是相对于当前视图的父视图而言的。可以看到，这里还把刚才测量出的宽度和高度传到了 layout()方法中。那么我们来看下 layout()方法中的代码是什么样的吧，如下所示：

```
public void layout(int l, int t, int r, int b) {  
  
    int oldL = mLeft;  
  
    int oldT = mTop;  
  
    int oldB = mBottom;  
  
    int oldR = mRight;  
  
    boolean changed = setFrame(l, t, r, b);  
  
    if (changed || (mPrivateFlags & LAYOUT_REQUIRED) == LAYOUT_REQUIRED) {  
  
        if (ViewDebug.TRACE_HIERARCHY) {  
  
            ViewDebug.trace(this, ViewDebug.HierarchyTraceType.ON_LAYOUT);  
  
        }  
  
        onLayout(changed, l, t, r, b);  
  
        mPrivateFlags &= ~LAYOUT_REQUIRED;  
  
        if (mOnLayoutChangeListeners != null) {  
  
            ArrayList<OnLayoutChangeListener> listenersCopy =
```

```
(ArrayList<OnLayoutChangeListener>)
mOnLayoutChangeListeners.clone();

    int numListeners = listenersCopy.size();

    for (int i = 0; i < numListeners; ++i) {

        listenersCopy.get(i).onLayoutChange(this, l, t, r, b, oldL, oldT, oldR,
oldB);

    }

}

mPrivateFlags &= ~FORCE_LAYOUT;

}
```

在 layout()方法中，首先会调用 setFrame()方法来判断视图的大小是否发生过变化，以确定有没有必要对当前的视图进行重绘，同时还会在这里把传递过来的四个参数分别赋值给 mLeft、mTop、mRight 和 mBottom 这几个变量。接下来会在第 11 行调用 onLayout()方法，正如 onMeasure()方法中的默认行为一样，也许你已经迫不及待地想知道 onLayout()方法中的默认行为是什么样的了。进入 onLayout()方法，咦？怎么这是个空方法，一行代码都没有？！

没错，View 中的 onLayout()方法就是一个空方法，因为 onLayout()过程是为了确定视图在布局中所在的位置，而这个操作应该是由布局来完成的，即父视图决定子视图的显示位置。既然如此，我们来看下 ViewGroup 中的 onLayout()方法是怎么写的吧，代码如下：

可以看到，ViewGroup 中的 onLayout()方法竟然是一个抽象方法，这就意味着所有 ViewGroup 的子类都必须重写这个方法。没错，像 LinearLayout、RelativeLayout 等布局，都是重写了这个方法，然后在内部按照各自的规则对子视图进行布局的。由于 LinearLayout 和 RelativeLayout 的布局规则都比较复杂，就不单独拿出来进行分析了，这里我们尝试自定义一个布局，借此来更深刻地理解 onLayout()的过程。

自定义的这个布局目标很简单，只要能够包含一个子视图，并且让子视图正常显示出来就可以了。那么就给这个布局起名叫做 SimpleLayout 吧，代码如下所示：

```
public class SimpleLayout extends ViewGroup {  
  
    public SimpleLayout(Context context, AttributeSet attrs) {  
        super(context, attrs);  
    }  
  
    @Override
```

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  
  
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);  
  
    if (getChildCount() > 0) {  
  
        View childView = getChildAt(0);  
  
        measureChild(childView, widthMeasureSpec, heightMeasureSpec);  
  
    }  
  
}  
  
@Override  
  
protected void onLayout(boolean changed, int l, int t, int r, int b) {  
  
    if (getChildCount() > 0) {  
  
        View childView = getChildAt(0);  
  
        childView.layout(0, 0, childView.getMeasuredWidth(),  
childView.getMeasuredHeight());  
  
    }  
  
}  
  
}
```

代码非常的简单，我们来看下具体的逻辑吧。你已经知道，`onMeasure()`方法会在`onLayout()`方法之前调用，因此这里在`onMeasure()`方法中判断`SimpleLayout`中是否有包含一个子视图，如果说有的话就调用`measureChild()`方法来测量出子视图的大小。

接着在 `onLayout()` 方法中同样判断 `SimpleLayout` 是否有包含一个子视图，然后调用这个子视图的 `layout()` 方法来确定它在 `SimpleLayout` 布局中的位置，这里传入的四个参数依次是 0、0、`childView.getMeasuredWidth()` 和 `childView.getMeasuredHeight()`，分别代表着子视图在 `SimpleLayout` 中左上右下四个点的坐标。其中，调用 `childView.getMeasuredWidth()` 和 `childView.getMeasuredHeight()` 方法得到的值就是在 `onMeasure()` 方法中测量出的宽和高。

这样就已经把 `SimpleLayout` 这个布局定义好了，下面就是在 XML 文件中使用它了，如下所示：

```
<com.example.viewtest.SimpleLayout
    xmlns:android="http://schemas.android.com/apk/res/android"

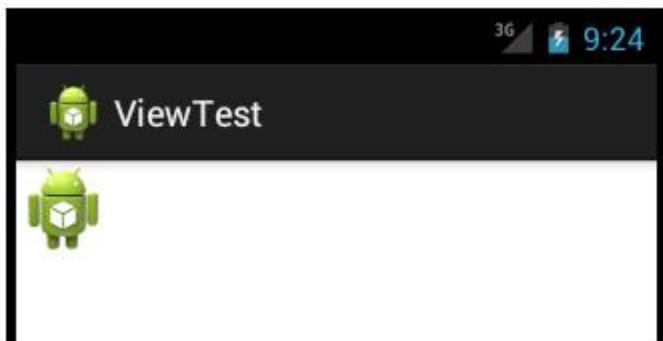
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_launcher"

    />
```

```
</com.example.viewtest.SimpleLayout>
```

可以看到，我们能够像使用普通的布局文件一样使用 SimpleLayout，只是注意它只能包含一个子视图，多余的子视图会被舍弃掉。这里 SimpleLayout 中包含了一个 ImageView，并且 ImageView 的宽高都是 wrap_content。现在运行一下程序，结果如下图所示：



OK！ImageView 成功已经显示出来了，并且显示的位置也正是我们所期望的。如果你想改变 ImageView 显示的位置，只需要改变 childView.layout()方法的四个参数就行了。

在 onLayout()过程结束后，我们就可以调用 getWidth()方法和 getHeight()方法来获取视图的宽高了。说到这里，我相信很多朋友长久以来都会有一个疑问，getWidth()方法和 getMeasuredWidth()方法到底有什么区别呢？它们的值好像永远都是相同的。其实它们的值之所以会相同基本都是因为布局设计者的编码习惯非常好，实际上它们之间的差别还是挺大的。

首先 `getMeasuredWidth()`方法在 `measure()`过程结束后就可以获取到了，而 `getWidth()`方法要在 `layout()`过程结束后才能获取到。另外，
`getMeasuredWidth()`方法中的值是通过 `setMeasuredDimension()`方法来进行设置的，而 `getWidth()`方法中的值则是通过视图右边的坐标减去左边的坐标计算出来的。

观察 `SimpleLayout` 中 `onLayout()`方法的代码，这里给子视图的 `layout()`方法传入的四个参数分别是 0、0、`childView.getMeasuredWidth()` 和 `childView.getMeasuredHeight()`，因此 `getWidth()`方法得到的值就是 `childView.getMeasuredWidth() - 0 = childView.getMeasuredWidth()`，所以此时 `getWidth()`方法和 `getMeasuredWidth()` 得到的值就是相同的，但如果将 `onLayout()`方法中的代码进行如下修改：

```
@Override
```

```
protected void onLayout(boolean changed, int l, int t, int r, int b) {  
  
    if (getChildCount() > 0) {  
  
        View childView = getChildAt(0);  
  
        childView.layout(0, 0, 200, 200);  
  
    }  
}
```

这样 `getWidth()` 方法得到的值就是 $200 - 0 = 200$ ，不会再和 `getMeasuredWidth()` 的值相同了。当然这种做法充分不尊重 `measure()` 过程计算出的结果，通常情况下是不推荐这么写的。`getHeight()` 与 `getMeasureHeight()` 方法之间的关系同上，就不再重复分析了。

到此为止，我们把视图绘制流程的第二阶段也分析完了。

三. `onDraw()`

`measure` 和 `layout` 的过程都结束后，接下来就进入到 `draw` 的过程了。同样，根据名字你就能够判断出，在这里才真正地开始对视图进行绘制。`ViewRoot` 中的代码会继续执行并创建出一个 `Canvas` 对象，然后调用 `View` 的 `draw()` 方法来执行具体的绘制工作。`draw()` 方法内部的绘制过程总共可以分为六步，其中第二步和第五步在一般情况下很少用到，因此这里我们只分析简化后的绘制过程。代码如下所示：

```
public void draw(Canvas canvas) {  
    if (ViewDebug.TRACE_HIERARCHY) {  
        ViewDebug.trace(this, ViewDebug.HierarchyTraceType.DRAW);  
    }  
}
```

```
final int privateFlags = mPrivateFlags;

final boolean dirtyOpaque = (privateFlags & DIRTY_MASK) ==
DIRTY_OPAQUE &&

(mAttachInfo == null || !mAttachInfo.mIgnoreDirtyState);

mPrivateFlags = (privateFlags & ~DIRTY_MASK) | DRAWN;

// Step 1, draw the background, if needed

int saveCount;

if (!dirtyOpaque) {

    final Drawable background = mBGDrawable;

    if (background != null) {

        final int scrollX = mScrollX;

        final int scrollY = mScrollY;

        if (mBackgroundSizeChanged) {

            background.setBounds(0, 0, mRight - mLeft, mBottom
- mTop);

            mBackgroundSizeChanged = false;
        }
    }
}
```

```
    if ((scrollX | scrollY) == 0) {  
  
        background.draw(canvas);  
  
    } else {  
  
        canvas.translate(scrollX, scrollY);  
  
        background.draw(canvas);  
  
        canvas.translate(-scrollX, -scrollY);  
  
    }  
  
}  
  
final int viewFlags = mViewFlags;  
  
boolean horizontalEdges = (viewFlags &  
FADING_EDGE_HORIZONTAL) != 0;  
  
boolean verticalEdges = (viewFlags & FADING_EDGE_VERTICAL) != 0;  
  
if (!verticalEdges && !horizontalEdges) {  
  
    // Step 3, draw the content  
  
    if (!dirtyOpaque) onDraw(canvas);
```

```
// Step 4, draw the children  
  
dispatchDraw(canvas);  
  
// Step 6, draw decorations (scrollbars)  
  
onDrawScrollBars(canvas);  
  
// we're done...  
  
return;  
  
}  
  
}
```

可以看到，第一步是从第 9 行代码开始的，这一步的作用是对视图的背景进行绘制。这里会先得到一个 mBGDrawable 对象，然后根据 layout 过程确定的视图位置来设置背景的绘制区域，之后再调用 Drawable 的 draw()方法来完成背景的绘制工作。那么这个 mBGDrawable 对象是从哪里来的呢？其实就是在 XML 中通过 android:background 属性设置的图片或颜色。当然你也可以在代码中通过 setBackgroundColor()、setBackgroundResource()等方法进行赋值。

接下来的第三步是在第 34 行执行的，这一步的作用是对视图的内容进行绘制。可以看到，这里去调用了一下 onDraw()方法，那么 onDraw()方法里又写了什么代码呢？进去一看你会发现，原来又是个空方法啊。其实也可以理解，因为每

个视图的内容部分肯定都是各不相同的，这部分的功能交给子类来实现也是理所当然的。

第三步完成之后紧接着会执行第四步，这一步的作用是对当前视图的所有子视图进行绘制。但如果当前的视图没有子视图，那么也就不需要进行绘制了。因此你会发现 View 中的 dispatchDraw()方法又是一个空方法，而 ViewGroup 的 dispatchDraw()方法中就会有具体的绘制代码。

以上都执行完后就会进入到第六步，也是最后一步，这一步的作用是对视图的滚动条进行绘制。那么你可能会奇怪，当前的视图又不一定是 ListView 或者 ScrollView，为什么要绘制滚动条呢？其实不管是 Button 也好，TextView 也好，任何一个视图都是有滚动条的，只是一般情况下我们都还没有让它显示出来而已。绘制滚动条的代码逻辑也比较复杂，这里就不再贴出来了，因为我们的重点是第三步过程。

通过以上流程分析，相信大家已经知道，View 是不会帮我们绘制内容部分的，因此需要每个视图根据想要展示的内容来自行绘制。如果你去观察 TextView、ImageView 等类的源码，你会发现它们都有重写 onDraw()这个方法，并且在里面执行了相当不少的绘制逻辑。绘制的方式主要是借助 Canvas 这个类，它会作为参数传入到 onDraw()方法中，供给每个视图使用。Canvas 这个类的用法

非常丰富，基本可以把它当成一块画布，在上面绘制任意的东西，那么我们就来尝试一下吧。

这里简单起见，我只是创建一个非常简单的视图，并且用 Canvas 随便绘制了一点东西，代码如下所示：

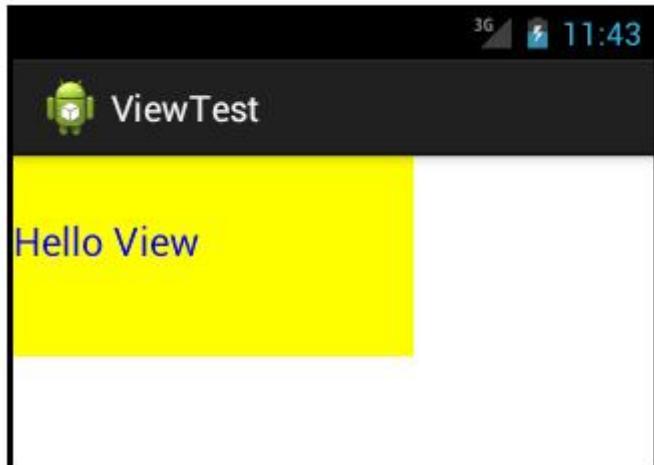
```
public class MyView extends View {  
  
    private Paint mPaint;  
  
    public MyView(Context context, AttributeSet attrs) {  
        super(context, attrs);  
        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);  
    }  
  
    @Override  
    protected void onDraw(Canvas canvas) {  
        mPaint.setColor(Color.YELLOW);  
        canvas.drawRect(0, 0, getWidth(), getHeight(), mPaint);  
        mPaint.setColor(Color.BLUE);  
        mPaint.setTextSize(20);  
        String text = "Hello View";  
        canvas.drawText(text, 0, getHeight() / 2, mPaint);  
    }  
}
```

可以看到，我们创建了一个自定义的 MyView 继承自 View，并在 MyView 的构造函数中创建了一个 Paint 对象。Paint 就像是一个画笔一样，配合着 Canvas 就可以进行绘制了。这里我们的绘制逻辑比较简单，在 onDraw()方法中先是把画笔设置成黄色，然后调用 Canvas 的 drawRect()方法绘制一个矩形。然后在把画笔设置成蓝色，并调整了一下文字的大小，然后调用 drawText()方法绘制了一段文字。

就这么简单，一个自定义的视图就已经写好了，现在可以在 XML 中加入这个视图，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent" >  
  
    <com.example.viewtest.MyView  
        android:layout_width="200dp"  
        android:layout_height="100dp"  
    />  
  
</LinearLayout>
```

将 MyView 的宽度设置成 200dp，高度设置成 100dp，然后运行一下程序，结果如下图所示：



图中显示的内容也正是 MyView 这个视图的内容部分了。由于我们没给 MyView 设置背景，因此这里看不出来 View 自动绘制的背景效果。

文章、Android 视图状态及重绘流程分析，带你一步步深入 了解 View(三)

一、视图状态

视图状态的种类非常多，一共有十几种类型，不过多数情况下我们只会使用到其中的几种，因此这里我们也就只去分析最常用的几种视图状态。

1. enabled

表示当前视图是否可用。可以调用 `setEnable()`方法来改变视图的可用状态，传入 `true` 表示可用，传入 `false` 表示不可用。它们之间最大的区别在于，不可用的视图是无法响应 `onTouch` 事件的。

2. focused

表示当前视图是否获得焦点。通常情况下有两种方法可以让视图获得焦点，即通过键盘的上下左右键切换视图，以及调用 `requestFocus()`方法。而现在的 Android 手机几乎没有键盘了，因此基本上只能使用 `requestFocus()`这个办法来让视图获得焦点了。而 `requestFocus()`方法也不能保证一定可以让视图获得焦点，它会有一个布尔值的返回值，如果返回 `true` 说明获得焦点成功，返回 `false` 说明获得焦点失败。一般只有视图在 `focusable` 和 `focusable in touch mode` 同时成立的情况下才能成功获取焦点，比如说 `EditText`。

3. window_focused

表示当前视图是否处于正在交互的窗口中，这个值由系统自动决定，应用程序不能进行改变。

4. selected

表示当前视图是否处于选中状态。一个界面当中可以有多个视图处于选中状态，调用 `setSelected()` 方法能够改变视图的选中状态，传入 `true` 表示选中，传入 `false` 表示未选中。

5. pressed

表示当前视图是否处于按下状态。可以调用 `setPressed()` 方法来对这一状态进行改变，传入 `true` 表示按下，传入 `false` 表示未按下。通常情况下这个状态都是由系统自动赋值的，但开发者也可以自己调用这个方法来进行改变。

我们可以在项目的 `drawable` 目录下创建一个 `selector` 文件，在这里配置每种状态下视图对应的背景图片。比如创建一个 `compose_bg.xml` 文件，在里面编写如下代码：

```
<selector

    xmlns:android="http://schemas.android.com/apk/res/android">

        <item          android:drawable="@drawable/compose_pressed"
            android:state_pressed="true"></item>

        <item          android:drawable="@drawable/compose_pressed"
            android:state_focused="true"></item>

        <item android:drawable="@drawable/compose_normal"></item>

</selector>
```

这段代码就表示，当视图处于正常状态的时候就显示 `compose_normal` 这张背景图，当视图获得到焦点或者被按下的时候就显示 `compose_pressed` 这张背景图。

创建好了这个 `selector` 文件后，我们就可以在布局或代码中使用它了，比如将它设置为某个按钮的背景图，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>

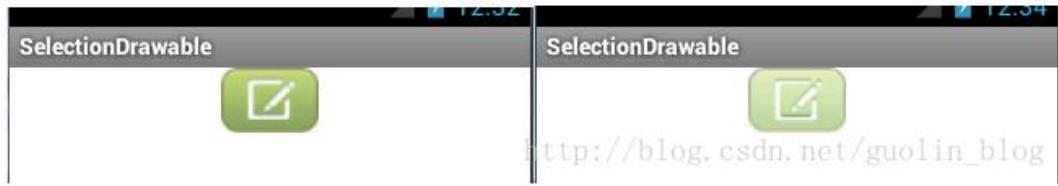
<LinearLayout

    xmlns:android="http://schemas.android.com/apk/res/android"

        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical" >
```

```
<Button  
    android:id="@+id/compose"  
    android:layout_width="60dp"  
    android:layout_height="40dp"  
    android:layout_gravity="center_horizontal"  
    android:background="@drawable/compose_bg"  
/>  
  
</LinearLayout>
```

现在运行一下程序 ,这个按钮在普通状态和按下状态的时候就会显示不同的背景图片 ,如下图所示 :



这样我们就用一个非常简单的方法实现了按钮按下的效果 ,但是它的背景原理到底 是怎样的呢 ?这就又要从源码的层次上进行分析了。

我们都知道 ,当手指按在视图上的时候 ,视图的状态就已经发生了变化 ,此时视图的 pressed 状态是 true。每当视图的状态有发生改变的时候 ,就会回调 View 的 drawableStateChanged()方法 ,代码如下所示 :

```
protected void drawableStateChanged() {  
    Drawable d = mBGDrawable;  
  
    if (d != null && d.isStateful()) {  
  
        d.setState(getDrawableState());  
  
    }  
}
```

在这里的第一步，首先是将 `mBGDrawable` 赋值给一个 `Drawable` 对象，那么这个 `mBGDrawable` 是什么呢？观察 `setBackgroundResource()` 方法中的代码，如下所示：

```
public void setBackgroundResource(int resid) {  
    if (resid != 0 && resid == mBackgroundResource) {  
        return;  
    }  
    Drawable d= null;  
    if (resid != 0) {  
        d = mResources.getDrawable(resid);  
    }  
    setBackgroundDrawable(d);  
    mBackgroundResource = resid;  
}
```

可以看到，在第 7 行调用了 `Resource` 的 `getDrawable()` 方法将 `resid` 转换成了一个 `Drawable` 对象，然后调用了 `setBackgroundDrawable()` 方法并将这个 `Drawable` 对象传入，在 `setBackgroundDrawable()` 方法中会将传入的 `Drawable` 对象赋值给 `mBGDrawable`。

而我们在布局文件中通过 `android:background` 属性指定的 `selector` 文件，效果等同于调用 `setBackgroundResource()` 方法。也就是说 `drawableStateChanged()` 方法中的 `mBGDrawable` 对象其实就是我们指定的 `selector` 文件。

接下来在 `drawableStateChanged()` 方法的第 4 行调用了 `getDrawableState()` 方法来获取视图状态，代码如下所示：

```
public final int[] getDrawableState() {
    if ((mDrawableState != null) && ((mPrivateFlags & DRAWABLE_STATE_DIRTY) == 0)) {
        return mDrawableState;
    } else {
        mDrawableState = onCreateDrawableState(0);
        mPrivateFlags &= ~DRAWABLE_STATE_DIRTY;
        return mDrawableState;
    }
}
```

在这里首先会判断当前视图的状态是否发生了改变，如果没有改变就直接返回当前的视图状态，如果发生了改变就调用 `onCreateDrawableState()` 方法来获取最新的视图状态。视图的所有状态会以一个整型数组的形式返回。

在得到了视图状态的数组之后，就会调用 `Drawable` 的 `setState()` 方法来对状态进行更新，代码如下所示：

```
public boolean setState(final int[] stateSet) {
    if (!Arrays.equals(mStateSet, stateSet)) {
        mStateSet = stateSet;
        return onStateChange(stateSet);
    }
    return false;
}
```

这里会调用 `Arrays.equals()` 方法来判断视图状态的数组是否发生了变化，如果发生了变化则调用 `onStateChange()` 方法，否则就直接返回 `false`。但你会发现，`Drawable` 的 `onStateChange()` 方法中其实就只是简单返回了一个 `false`，并没有任何的逻辑处理，这是为什么呢？这主要是因为 `mBGDrawable` 对象是通过一个 `selector` 文件创建出来的，而通过这种文件创建出来的 `Drawable` 对象其实都是一个 `StateListDrawable` 实例，因此这里调用的

`onStateChange()`方法实际上调用的是 `StateListDrawable` 中的 `onStateChange()`方法，那么我们赶快看一下吧：

```
@Override  
protected boolean onStateChange(int[] stateSet) {  
    int idx = mStateListState.indexOfStateSet(stateSet);  
    if (DEBUG) android.util.Log.i(TAG, "onStateChange " + this + " states "  
        + Arrays.toString(stateSet) + " found " + idx);  
    if (idx < 0) {  
        idx = mStateListState.indexOfStateSet(StateSet.WILD_CARD);  
    }  
    if (selectDrawable(idx)) {  
        return true;  
    }  
    return super.onStateChange(stateSet);  
}
```

可以看到，这里会先调用 `indexOfStateSet()`方法来找到当前视图状态所对应的 `Drawable` 资源下标，然后在第 9 行调用 `selectDrawable()`方法并将下标传入，在这个方法中就会将视图的背景图设置为当前视图状态所对应的那张图片了。

那你可能会有疑问，在前面一篇文章中我们说到，任何一个视图的显示都要经过非常科学的绘制流程的，很显然，背景图的绘制是在 `draw()`方法中完成的，那么为什么 `selectDrawable()`方法能够控制背景图的改变呢？这就要研究一下视图重绘的流程了。

二、视图重绘

虽然视图会在 `Activity` 加载完成之后自动绘制到屏幕上，但是我们完全有理由在与 `Activity` 进行交互的时候要求动态更新视图，比如改变视图的状态、以及显示或隐藏某个控件等。那在这个时候，之前绘制出的视图其实就已经过期了，此时我们就应该对视图进行重绘。

调用视图的 setVisibility()、setEnabled()、setSelected()等方法时都会导致视图重绘，而如果我们想要手动地强制让视图进行重绘，可以调用 invalidate()方法来实现。当然了，setVisibility()、setEnabled()、setSelected()等方法的内部其实也是通过调用 invalidate()方法来实现的，那么就让我们来看一看 invalidate()方法的代码是什么样的吧。

View 的源码中会有数个 invalidate()方法的重载和一个 invalidateDrawable()方法，当然它们的原理都是相同的，因此我们只分析其中一种，代码如下所示：

```
void invalidate(boolean invalidateCache) {
    if (ViewDebug.TRACE_HIERARCHY) {
        ViewDebug.trace(this, ViewDebug.HierarchyTraceType.INVALIDATE);
    }
    if (skipInvalidate()) {
        return;
    }
    if ((mPrivateFlags & (DRAWN | HAS_BOUNDS)) == (DRAWN | HAS_BOUNDS) ||
        (invalidateCache && (mPrivateFlags & DRAWING_CACHE_VALID) ==
DRAWING_CACHE_VALID) ||
        (mPrivateFlags & INVALIDATED) != INVALIDATED || isOpaque() !=
mLastIsOpaque) {
        mLastIsOpaque = isOpaque();
        mPrivateFlags &= ~DRAWN;
        mPrivateFlags |= DIRTY;
        if (invalidateCache) {
            mPrivateFlags |= INVALIDATED;
            mPrivateFlags &= ~DRAWING_CACHE_VALID;
        }
        final AttachInfo ai = mAttachInfo;
        final ViewParent p = mParent;
        if (!HardwareRenderer.RENDER_DIRTY_REGIONS) {
            if (p != null && ai != null && ai.mHardwareAccelerated) {
```

```

        p.invalidateChild(this, null);
        return;
    }
}
if (p != null && ai != null) {
    final Rect r = ai.mTmpInvalRect;
    r.set(0, 0, mRight - mLeft, mBottom - mTop);
    p.invalidateChild(this, r);
}
}
}

```

在这个方法中首先会调用 `skipInvalidate()` 方法来判断当前 View 是否需要重绘，判断的逻辑也比较简单，如果 View 是不可见的且没有执行任何动画，就认为不需要重绘了。之后会进行透明度的判断，并给 View 添加一些标记位，然后在第 22 和 29 行调用 `ViewParent` 的 `invalidateChild()` 方法，这里的 `ViewParent` 其实就是当前视图的父视图，因此会调用到 `ViewGroup` 的 `invalidateChild()` 方法中，代码如下所示：

```

public final void invalidateChild(View child, final Rect dirty) {
    ViewParent parent = this;
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        final boolean drawAnimation = (child.mPrivateFlags & DRAW_ANIMATION)
== DRAW_ANIMATION;
        if (dirty == null) {
            .....
        } else {
            .....
            do {
                View view = null;
                if (parent instanceof View) {
                    view = (View) parent;
                    if (view.mLayerType != LAYER_TYPE_NONE &&
                        view.getParent() instanceof View) {
                        final View grandParent = (View) view.getParent();
                        grandParent.mPrivateFlags |= INVALIDATED;
                        grandParent.mPrivateFlags &= ~DRAWING_CACHE_VALID;
                    }
                }
                if (drawAnimation) {
                    if (view != null) {
                        view.mPrivateFlags |= DRAW_ANIMATION;
                    } else if (parent instanceof ViewRootImpl) {
                        ((ViewRootImpl) parent).mIsAnimating = true;
                    }
                }
            }
        }
    }
}

```

```

        if (view != null) {
            if ((view.mViewFlags & FADING_EDGE_MASK) != 0 &&
                view.getSolidColor() == 0) {
                opaqueFlag = DIRTY;
            }
            if ((view.mPrivateFlags & DIRTY_MASK) != DIRTY) {
                view.mPrivateFlags     =      (view.mPrivateFlags     &
~DIRTY_MASK) | opaqueFlag;
            }
        }
        parent = parent.invalidateChildInParent(location, dirty);
        if (view != null) {
            Matrix m = view.getMatrix();
            if (!m.isIdentity()) {
                RectF boundingRect = attachInfo.mTmpTransformRect;
                boundingRect.set(dirty);
                m.mapRect(boundingRect);
                dirty.set((int) boundingRect.left, (int)
boundingRect.top,
                           (int) (boundingRect.right + 0.5f),
                           (int) (boundingRect.bottom + 0.5f));
            }
        }
    } while (parent != null);
}
}

```

可以看到，这里在第 10 行进入了一个 `while` 循环，当 `ViewParent` 不等于空的时候就会一直循环下去。在这个 `while` 循环当中会不断地获取当前布局的父布局，并调用它的 `invalidateChildInParent()` 方法，在 `ViewGroup` 的 `invalidateChildInParent()` 方法中主要是来计算需要重绘的矩形区域，这里我们先不管它，当循环到最外层的根布局后，就会调用 `ViewRoot` 的 `invalidateChildInParent()` 方法了，代码如下所示：

```

public ViewParent invalidateChildInParent(final int[] location, final
Rect dirty) {
    invalidateChild(null, dirty);
    return null;
}

```

这里的代码非常简单，仅仅是去调用了 `invalidateChild()` 方法而已，那我们再跟进去瞧一瞧吧：

```

public void invalidateChild(View child, Rect dirty) {
    checkThread();
    if (LOCAL_LOGV) Log.v(TAG, "Invalidate child: " + dirty);
    mDirty.union(dirty);
    if (!mWillDrawSoon) {

```

```
        scheduleTraversals();
    }
}
```

这个方法也不长，它在第 6 行又调用了 `scheduleTraversals()` 这个方法，那么我们继续跟进：

```
public void scheduleTraversals() {
    if (!mTraversalScheduled) {
        mTraversalScheduled = true;
        sendEmptyMessage(DO_TRAVERSAL);
    }
}
```

可以看到，这里调用了 `sendEmptyMessage()` 方法，并传入了一个 `DO_TRAVERSAL` 参数。了解 Android 异步消息处理机制的朋友们都会知道，任何一个 `Handler` 都可以调用 `sendEmptyMessage()` 方法来发送消息，并且在 `handleMessage()` 方法中接收消息，而如果你看一下 `ViewRoot` 的类定义就会发现，它是继承自 `Handler` 的，也就是说这里调用 `sendEmptyMessage()` 方法出的消息，会在 `ViewRoot` 的 `handleMessage()` 方法中接收到。那么赶快看一下 `handleMessage()` 方法的代码吧，如下所示：

```
public void handleMessage(Message msg) {
    switch (msg.what) {
        case DO_TRAVERSAL:
            if (mProfile) {
                Debug.startMethodTracing("ViewRoot");
            }
            performTraversals();
            if (mProfile) {
                Debug.stopMethodTracing();
                mProfile = false;
            }
            break;
        .....
    }
}
```

熟悉的代码出现了！这里在第 7 行调用了 `performTraversals()` 方法，这不就是我们在前面一篇文章中学到的视图绘制的入口吗？虽然经过了很多辗转的调用，但是可以确定的是，调用视图的 `invalidate()` 方法后确实会走到 `performTraversals()` 方法中，然后重新执行绘制流程。之后的流程就不再进行描述了吧，可以参考 [Android 视图绘制流程完全解析，带你一步步深入了解 View\(二\)](#) 这篇文章。

了解了这些之后，我们再回过头来看看刚才的 `selectDrawable()` 方法中到底做了什么才能够控制背景图的改变，代码如下所示：

```
public boolean selectDrawable(int idx) {  
    if (idx == mCurIndex) {  
        return false;  
    }  
  
    final long now = SystemClock.uptimeMillis();  
  
    if (mDrawableContainerState.mExitFadeDuration > 0) {  
        if (mLastDrawable != null) {  
            mLastDrawable.setVisible(false, false);  
        }  
  
        if (mCurrDrawable != null) {  
            mLastDrawable = mCurrDrawable;  
            mExitAnimationEnd = now +  
                mDrawableContainerState.mExitFadeDuration;  
        }  
    }  
}
```

```
    } else {

        mLastDrawable = null;

        mExitAnimationEnd = 0;

    }

} else if (mCurrDrawable != null) {

    mCurrDrawable.setVisible(false, false);

}

if (idx >= 0 && idx < mDrawableContainerState.mNumChildren) {

    Drawable d = mDrawableContainerState.mDrawables[idx];

    mCurrDrawable = d;

    mCurIndex = idx;

    if (d != null) {

        if (mDrawableContainerState.mEnterFadeDuration > 0) {

            mEnterAnimationEnd = now + mDrawableContainerState.mEnterFadeDuration;

        } else {

            d.setAlpha(mAlpha);

        }

        d.setVisible(isVisible(), true);

        d.setDither(mDrawableContainerState.mDither);

        d.setColorFilter(mColorFilter);

        d.setState(getState());

    }

}
```

```
    d.setLevel(getLevel());

    d.setBounds(getBounds());

}

} else {

    mCurrDrawable = null;

    mCurIndex = -1;

}

if (mEnterAnimationEnd != 0 || mExitAnimationEnd != 0) {

    if (mAnimationRunnable == null) {

        mAnimationRunnable = new Runnable() {

            @Override public void run() {

                animate(true);

                invalidateSelf();

            }

        };

    } else {

        unscheduleSelf(mAnimationRunnable);

    }

    animate(true);

}

invalidateSelf();

return true;
```

```
}
```

这里前面的代码我们可以都不管，关键是要看到在第 54 行一定会调用 `invalidateSelf()`方法，这个方法中的代码如下所示：

```
public void invalidateSelf() {  
    final Callback callback = getCallback();  
    if (callback != null) {  
        callback.invalidateDrawable(this);  
    }  
}
```

可以看到，这里会先调用 `getCallback()`方法获取 `Callback` 接口的回调实例，然后再去调用回调实例的 `invalidateDrawable()`方法。那么这里的回调实例又是什么呢？观察一下 `View` 的类定义其实你就知道了，如下所示：

```
1 public class View implements Drawable.Callback, Drawable.Callback2, KeyEvent.Callback,  
2 AccessibilityEventSource {  
3     .....  
4 }
```

`View` 类正是实现了 `Callback` 接口，所以刚才其实调用的就是 `View` 中的 `invalidateDrawable()`方法，之后就会按照我们前面分析的流程执行重绘逻辑，所以视图的背景图才能够得到改变的。

另外需要注意的是，`invalidate()`方法虽然最终会调用到 `performTraversals()`方法中，但这时 `measure` 和 `layout` 流程是不会重新执行的，因为视图没有强制重新测量的标志位，而且大小也没有发生过变化，所以这时只有 `draw` 流程可以得到执行。而如果你希望视图的绘制流程可以完完整整地重新走一遍，就不能使用 `invalidate()`方法，而应该调用 `requestLayout()` 了。这个方法中的流程比 `invalidate()`方法要简单一些，但中心思想是差不多的，这里也就不再详细进行分析了。

这样的话，我们就将视图状态以及重绘的工作原理都搞清楚了，相信大家对 `View` 的理解变得更加深刻了。

文章、Android 自定义 View 的实现方法，带你一步步深入了解 View(四)

一、自绘控件

自绘控件的意思就是，这个 View 上所展现的内容全部都是我们自己绘制出来的。绘制的代码是写在 `onDraw()` 方法中的，而这部分内容我们已经在 [Android 视图绘制流程完全解析，带你一步步深入了解 View\(二\)](#) 中学习过了。

下面我们准备来自定义一个计数器 View，这个 View 可以响应用户的点击事件，并自动记录一共点击了多少次。新建一个 CounterView 继承自 View，代码如下所示：

```
public class CounterView extends View implements OnClickListener {  
  
    private Paint mPaint;  
  
    private Rect mBounds;  
  
    private int mCount;  
  
    public CounterView(Context context, AttributeSet attrs) {
```

```
super(context, attrs);

mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);

mBounds = new Rect();

setOnClickListener(this);

}

@Override

protected void onDraw(Canvas canvas) {

    super.onDraw(canvas);

    mPaint.setColor(Color.BLUE);

    canvas.drawRect(0, 0, getWidth(), getHeight(), mPaint);

    mPaint.setColor(Color.YELLOW);

    mPaint.setTextSize(30);

    String text = String.valueOf(mCount);

    mPaint.getTextBounds(text, 0, text.length(), mBounds);

    float textWidth = mBounds.width();

    float textHeight = mBounds.height();

    canvas.drawText(text, getWidth() / 2 - textWidth / 2,
        getHeight() / 2

        + textHeight / 2, mPaint);

}
```

```
    @Override  
  
    public void onClick(View v) {  
  
        mCount++;  
  
        invalidate();  
  
    }  
  
}
```

可以看到，首先我们在 CounterView 的构造函数中初始化了一些数据，并给这个 View 的本身注册了点击事件，这样当 CounterView 被点击的时候，onClick()方法就会得到调用。而 onClick()方法中的逻辑就更加简单了，只是对 mCount 这个计数器加 1，然后调用 invalidate()方法。通过 [Android 视图状态及重绘流程分析](#)，带你一步步深入了解 View(三) 这篇文章的学习我们都已经知道，调用 invalidate()方法会导致视图进行重绘，因此 onDraw()方法在稍后就将会得到调用。

既然 CounterView 是一个自绘视图，那么最主要的逻辑当然就是写在 onDraw() 方法里的了，下面我们就来仔细看一下。这里首先是将 Paint 画笔设置为蓝色，然后调用 Canvas 的 drawRect()方法绘制了一个矩形，这个矩形也就可以当作是 CounterView 的背景图吧。接着将画笔设置为黄色，准备在背景上面绘制当前的计数，注意这里先是调用了 getTextBounds()方法来获取到文字的宽度和高度，然后调用了 drawText()方法去进行绘制就可以了。

这样，一个自定义的 View 就已经完成了，并且目前这个 CounterView 是具备自动计数功能的。那么剩下的问题就是如何让这个 View 在界面上显示出来了，其实这也非常简单，我们只需要像使用普通的控件一样来使用 CounterView 就可以了。比如在布局文件中加入如下代码：

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"

    android:layout_width="match_parent"
    android:layout_height="match_parent" >

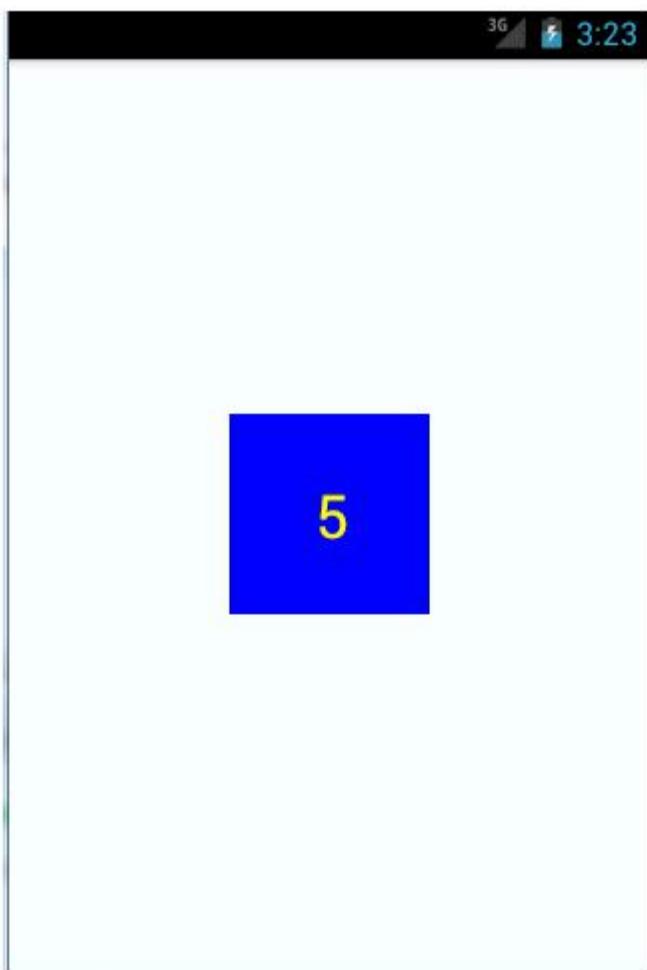
    <com.example.customview.CounterView
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_centerInParent="true" />

</RelativeLayout>
```

可以看到，这里我们将 CounterView 放入了一个 RelativeLayout 中，然后可以像使用普通控件来给 CounterView 指定各种属性，比如通过 layout_width 和 layout_height 来指定 CounterView 的宽高，通过 android:layout_centerInParent 来指定它在布局里居中显示。只不过需要注意，自定义的 View 在使用的时候一定要写出完整的包名，不然系统将无法找到这个 View。

好了，就是这么简单，接下来我们可以运行一下程序，并不停地点击

CounterView，效果如下图所示。



怎么样？是不是感觉自定义 View 也并不是什么高级的技术，简单几行代码就可以实现了。当然了，这个 CounterView 功能非常简陋，只有一个计数功能，因此只需几行代码就足够了，当你需要绘制比较复杂的 View 时，还是需要很多技巧的。

二、组合控件

组合控件的意思就是，我们并不需要自己去绘制视图上显示的内容，而只是用系统原生的控件就好了，但我们可以将几个系统原生的控件组合到一起，这样创建出的控件就被称为组合控件。

举个例子来说，标题栏就是个很常见的组合控件，很多界面的头部都会放置一个标题栏，标题栏上会有个返回按钮和标题，点击按钮后就可以返回到上一个界面。那么下面我们就来尝试去实现这样一个标题栏控件。

新建一个 title.xml 布局文件，代码如下所示

```
<?xml version="1.0" encoding="utf-8"?>

<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"

    android:layout_width="match_parent"
    android:layout_height="50dp"
    android:background="#ffcb05" >

    <Button
        android:id="@+id/button_left"
        android:layout_width="60dp"
```

```
    android:layout_height="40dp"
    android:layout_centerVertical="true"
    android:layout_marginLeft="5dp"
    android:background="@drawable/back_button"
    android:text="Back"
    android:textColor="#fff" />

<TextView
    android:id="@+id/title_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:text="This is Title"
    android:textColor="#fff"
    android:textSize="20sp" />

</RelativeLayout>
```

在这个布局文件中，我们首先定义了一个 RelativeLayout 作为背景布局，然后在这个布局里定义了一个 Button 和一个 TextView , Button 就是标题栏中的返回按钮， TextView 就是标题栏中的显示的文字。

接下来创建一个 TitleView 继承自 FrameLayout，代码如下所示：

```
public class TitleView extends FrameLayout {  
  
    private Button leftButton;  
  
    private TextView titleText;  
  
    public TitleView(Context context, AttributeSet attrs) {  
        super(context, attrs);  
  
        LayoutInflater.from(context).inflate(R.layout.title,  
this);  
  
        titleText = (TextView) findViewById(R.id.title_text);  
  
        leftButton = (Button) findViewById(R.id.button_left);  
  
        leftButton.setOnClickListener(new OnClickListener() {  
  
            @Override  
  
            public void onClick(View v) {  
  
                ((Activity) getContext()).finish();  
  
            }  
  
        });  
  
    }  
  
    public void setTitleText(String text) {
```

```
        titleText.setText(text);

    }

    public void setLeftButtonText(String text) {

        leftButton.setText(text);

    }

    public void setLeftButtonListener(OnClickListener l) {

        leftButton.setOnClickListener(l);

    }

}
```

TitleView 中的代码非常简单，在 TitleView 的构建方法中，我们调用了 LayoutInflator 的 inflate() 方法来加载刚刚定义的 title.xml 布局，这部分内容我们已经在 [Android LayoutInflator 原理分析，带你一步步深入了解 View\(一\)](#) 这篇文章中学习过了。

接下来调用 findViewById() 方法获取到了返回按钮的实例，然后在它的 onClick 事件中调用 finish() 方法来关闭当前的 Activity，也就相当于实现返回功能了。

另外，为了让 TitleView 有更强地扩展性，我们还提供了 setTitleText()、setLeftButtonText()、setLeftButtonListener()等方法，分别用于设置标题栏上的文字、返回按钮上的文字、以及返回按钮的点击事件。

到了这里，一个自定义的标题栏就完成了，那么下面又到了如何引用这个自定义 View 的部分，其实方法基本都是相同的，在布局文件中添加如下代码：

```
<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" >  
  
    <com.example.customview.TitleView  
        android:id="@+id/title_view"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content" >  
    </com.example.customview.TitleView>  
  
</RelativeLayout>
```

这样就成功将一个标题栏控件引入到布局文件中了，运行一下程序，效果如下图所示：



现在点击一下 Back 按钮，就可以关闭当前的 Activity 了。如果你想要修改标题栏上显示的内容，或者返回按钮的默认事件，只需要在 Activity 中通过

`findViewById()`方法得到 `TitleView` 的实例，然后调用 `setTitleText()`、
`setLeftButtonText()`、`setLeftButtonListener()` 等方法进行设置就 OK 了。

三、继承控件

继承控件的意思就是，我们并不需要自己重头去实现一个控件，只需要去继承一个现有的控件，然后在这个控件上增加一些新的功能，就可以形成一个自定义的控件了。这种自定义控件的特点就是不仅能够按照我们的需求加入相应功能，还可以保留原生控件的所有功能，比如 [Android PowerImageView 实现，可以播放动画的强大 ImageView](#) 这篇文章中介绍的 `PowerImageView` 就是一个典型的继承控件。

为了能够加深大家对这种自定义 View 方式的理解，下面我们再来编写一个新的继承控件。`ListView` 相信每一个 Android 程序员都一定使用过，这次我们准备对 `ListView` 进行扩展，加入在 `ListView` 上滑动就可以显示出一个删除按钮，点击按钮就会删除相应数据的功能。

首先需要准备一个删除按钮的布局，新建 `delete_button.xml` 文件，代码如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<Button xmlns:android="http://schemas.android.com/apk/res/android"  
    android:id="@+id/delete_button"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:background="@drawable/delete_button" >  
  
</Button>
```

这个布局文件很简单，只有一个按钮而已，并且我们给这个按钮指定了一张删除背景图。

接着创建 `MyListView` 继承自 `ListView`，这就是我们自定义的 View 了，代码如下所示：

```
public class MyListView extends ListView implements OnTouchListener,  
    OnGestureListener {  
  
    private GestureDetector gestureDetector;  
  
    private OnDeleteListener listener;  
  
    private View deleteButton;
```

```
private ViewGroup itemLayout;

private int selectedItem;

private boolean isDeleteShown;

public MyListView(Context context, AttributeSet attrs) {

    super(context, attrs);

    gestureDetector = new GestureDetector(getContext(), this);

    setOnTouchListener(this);

}

public void setonDeleteListener(OnDeleteListener l) {

    listener = l;

}

@Override

public boolean onTouch(View v, MotionEvent event) {

    if (isDeleteShown) {

        itemLayout.removeView(deleteButton);

        deleteButton = null;

        isDeleteShown = false;

    }

}
```

```
        return false;

    } else {
        return gestureDetector.onTouchEvent(event);
    }
}

@Override
public boolean onDown(MotionEvent e) {
    if (!isDeleteShown) {
        selectedItem = pointToPosition((int) e.getX(), (int) e.getY());
    }
    return false;
}

@Override
public boolean onFling(MotionEvent e1, MotionEvent e2, float
velocityX,
    float velocityY) {
    if (!isDeleteShown && Math.abs(velocityX) > Math.abs(velocityY))
    {
        deleteButton = LayoutInflater.from(getContext()).inflate(
            R.layout.delete_button, null);
    }
}
```

```
deleteButton.setOnClickListener(new OnClickListener() {

    @Override

    public void onClick(View v) {

        itemLayout.removeView(deleteButton);

        deleteButton = null;

        isDeleteShown = false;

        listener.onDelete(selectedItem);

    }

});

itemLayout = (ViewGroup) getChildAt(selectedItem

    - getFirstVisiblePosition());

    RelativeLayout.LayoutParams     params      =      new

RelativeLayout.LayoutParams(

    LayoutParams.WRAP_CONTENT,

    LayoutParams.WRAP_CONTENT);

    params.addRule(RelativeLayout.ALIGN_PARENT_RIGHT);

    params.addRule(RelativeLayout.CENTER_VERTICAL);

    itemLayout.addView(deleteButton, params);

    isDeleteShown = true;

}

return false;

}
```

```
@Override  
public boolean onSingleTapUp(MotionEvent e) {  
    return false;  
}
```

```
@Override  
public void onShowPress(MotionEvent e) {  
}  
}
```

```
@Override  
public boolean onScroll(MotionEvent e1, MotionEvent e2, float  
distanceX,  
    float distanceY) {  
    return false;  
}
```

```
@Override  
public void onLongPress(MotionEvent e) {  
}
```

```
public interface OnDeleteListener {  
  
    void onDelete(int index);  
  
}  
  
}
```

由于代码逻辑比较简单，我就没有加注释。这里在 `MyListView` 的构造方法中创建了一个 `GestureDetector` 的实例用于监听手势，然后给 `MyListView` 注册了 `touch` 监听事件。然后在 `onTouch()` 方法中进行判断，如果删除按钮已经显示了，就将它移除掉，如果删除按钮没有显示，就使用 `GestureDetector` 来处理当前手势。

当手指按下时，会调用 `OnGestureListener` 的 `onDown()` 方法，在这里通过 `pointToPosition()` 方法来判断出当前选中的是 `ListView` 的哪一行。当手指快速滑动时，会调用 `onFling()` 方法，在这里会去加载 `delete_button.xml` 这个布局，然后将删除按钮添加到当前选中的那一行 `item` 上。注意，我们还给删除按钮添加了一个点击事件，当点击了删除按钮时就会回调 `onDeleteListener` 的 `onDelete()` 方法，在回调方法中应该去处理具体的删除操作。

好了，自定义 View 的功能到此就完成了，接下来我们需要看一下如何才能使用这个自定义 View。首先需要创建一个 ListView 子项的布局文件，新建 my_list_view_item.xml，代码如下所示：

```
<?xml version="1.0" encoding="utf-8"?>

<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:descendantFocusability="blocksDescendants"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/text_view"
        android:layout_width="wrap_content"
        android:layout_height="50dp"
        android:layout_centerVertical="true"
        android:gravity="left|center_vertical"
        android:textColor="#000" />

</RelativeLayout>
```

然后创建一个适配器 MyAdapter，在这个适配器中去加载 my_list_view_item 布局，代码如下所示：

```
public class MyAdapter extends ArrayAdapter<String> {
```

```
public MyAdapter(Context context, int textViewResourceId, List<String> objects) {  
    super(context, textViewResourceId, objects);  
}  
  
@Override  
public View getView(int position, View convertView, ViewGroup parent) {  
    View view;  
    if (convertView == null) {  
        view = LayoutInflater.from(getContext()).inflate(R.layout.my_list_view_item,  
null);  
    } else {  
        view = convertView;  
    }  
    TextView textView = (TextView) view.findViewById(R.id.text_view);  
    textView.setText(getItem(position));  
    return view;  
}  
  
}
```

到这里就基本已经完工了，下面在程序的主布局文件里面引入 **MyListView** 这个控件，如下所示：

```
<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" >  
  
    <com.example.customview.MyListView  
        android:id="@+id/my_list_view"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content" >  
    </com.example.customview.MyListView>  
  
</RelativeLayout>
```

最后在 **Activity** 中初始化 **MyListView** 中的数据，并处理了 **onDelete()**方法的删除逻辑，代码如下所示：

```
public class MainActivity extends Activity {  
  
    private MyListView myListview;  
  
    private MyAdapter adapter;  
  
    private List<String> contentList = new ArrayList<String>();  
  
    @Override
```

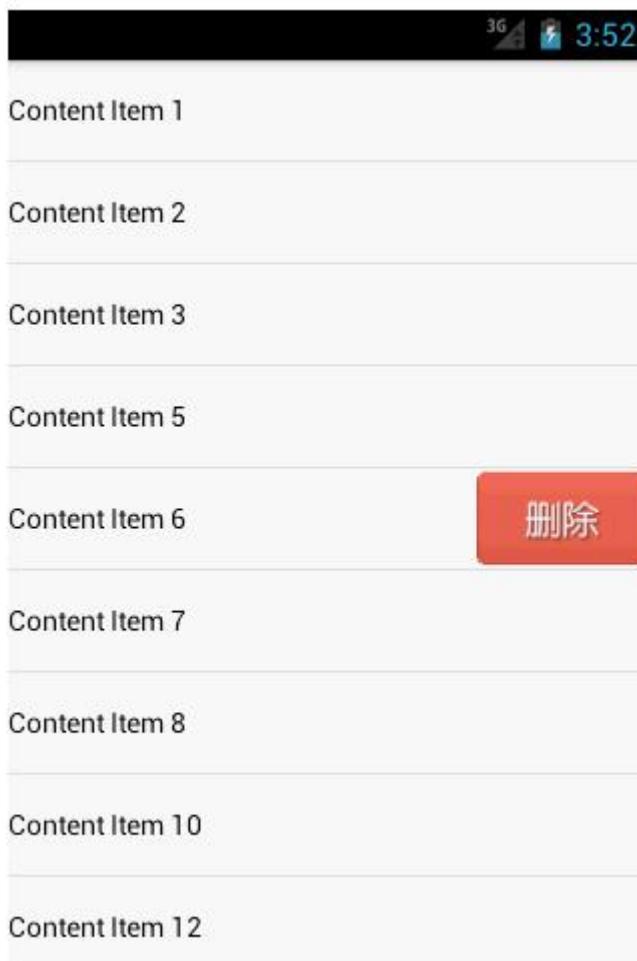
```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    setContentView(R.layout.activity_main);
    initList();
    myListview = (MyListView) findViewById(R.id.my_list_view);
    myListview.setOnDeleteListener(new OnDeleteListener() {
        @Override
        public void onDelete(int index) {
            contentList.remove(index);
            adapter.notifyDataSetChanged();
        }
    });
    adapter = new MyAdapter(this, 0, contentList);
    myListview.setAdapter(adapter);
}

private void initList() {
    contentList.add("Content Item 1");
    contentList.add("Content Item 2");
    contentList.add("Content Item 3");
    contentList.add("Content Item 4");
    contentList.add("Content Item 5");
    contentList.add("Content Item 6");
    contentList.add("Content Item 7");
    contentList.add("Content Item 8");
    contentList.add("Content Item 9");
    contentList.add("Content Item 10");
    contentList.add("Content Item 11");
    contentList.add("Content Item 12");
    contentList.add("Content Item 13");
    contentList.add("Content Item 14");
    contentList.add("Content Item 15");
    contentList.add("Content Item 16");
    contentList.add("Content Item 17");
    contentList.add("Content Item 18");
    contentList.add("Content Item 19");
    contentList.add("Content Item 20");
}

}
```

这样就把整个例子的代码都完成了，现在运行一下程序，会看到 MyListView 可以像 ListView 一样，正常显示所有的数据，但是当你用手指在 MyListView 的

某一行上快速滑动时，就会有一个删除按钮显示出来，如下图所示：



点击一下删除按钮就可以将第 6 行的数据删除了。此时的 `MyListView` 不仅保留了 `ListView` 原生的所有功能，还增加了一个滑动进行删除的功能，确实是一个不折不扣的继承控件。

十、Android Window、Activity、 DecorView 以及 ViewRoot

一、职能简介

Activity

Activity 并不负责视图控制，它只是控制生命周期和处理事件。真正控制视图的是 Window。一个 Activity 包含了一个 Window，Window 才是真正代表一个窗口。**Activity 就像一个控制器，统筹视图的添加与显示，以及通过其他回调方法，来与 Window、以及 View 进行交互。**

Window

Window 是视图的承载器，内部持有一个 DecorView，而这个 DecorView 才是 view 的根布局。Window 是一个抽象类，实际在 Activity 中持有的是其子类 PhoneWindow。PhoneWindow 中有个内部类 DecorView，通过创建 DecorView 来加载 Activity 中设置的布局 R.layout.activity_main。Window 通过 WindowManager 将 DecorView 加载其中，并将 DecorView 交给 ViewRoot，进行视图绘制以及其他交互。

DecorView

DecorView 是 FrameLayout 的子类，它可以被认为是 Android 视图树的根节点视图。DecorView 作为顶级 View，一般情况下它内部包含一个竖直方向的 LinearLayout，在这个 LinearLayout 里面有上下三个部分，上面是个 ViewStub，延迟加载的视图（应该是设置 ActionBar，根据 Theme 设置），中间的是标题栏（根据 Theme 设置，有的布局没有），下面的是内容栏。具体情况和 Android 版本及主体有关，以其中一个布局为例，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

    android:orientation="vertical"

    android:fitsSystemWindows="true">

    <!-- Popout bar for action modes -->

    <ViewStub android:id="@+id/action_mode_bar_stub"

        android:inflatedId="@+id/action_mode_bar"

        android:layout="@layout/action_mode_bar"

        android:layout_width="match_parent"

        android:layout_height="wrap_content"

        android:theme="?attr/actionBarTheme" />

    <FrameLayout

        android:layout_width="match_parent"

        android:layout_height="?android:attr/windowTitleSize"

        style="?android:attr/windowTitleBackgroundStyle">
```

```
<TextView android:id="@+id/title"
    style="?android:attr/windowTitleStyle"
    android:background="@null"
    android:fadingEdge="horizontal"
    android:gravity="center_vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

</FrameLayout>

<FrameLayout android:id="@+id/content"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
    android:foregroundGravity="fill_horizontal|top"
    android:foreground="?android:attr/windowContentOverlay" /></LinearLayout>
```

在Activity中通过setContentView所设置的布局文件其实就是在内容栏之中的，成为其唯一子View，就是上面的id为content的FrameLayout中，在代码中可以通过content来得到对应加载的布局。

```
 ViewGroup content = (ViewGroup) findViewById(android.R.id.content);

 ViewGroup rootView = (ViewGroup) content.getChildAt(0);
```

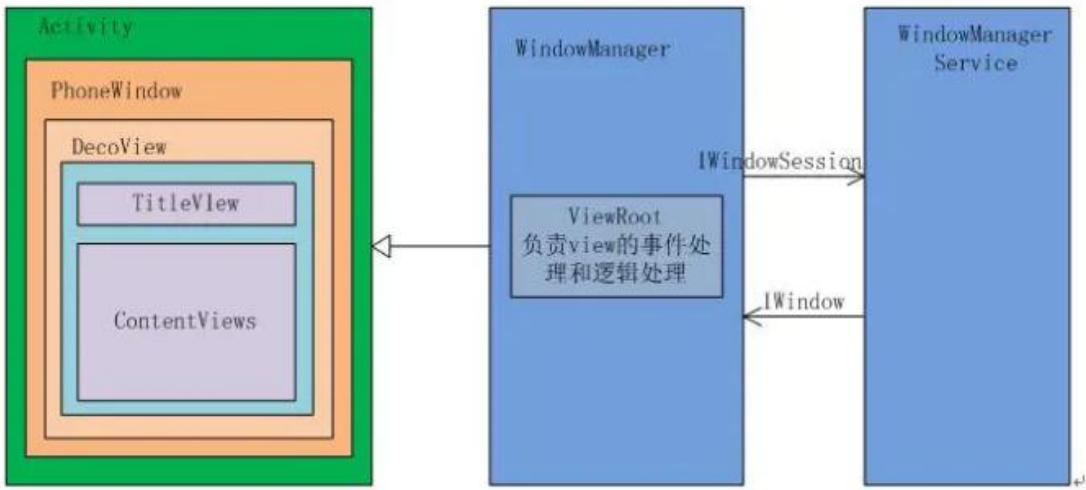
ViewRoot

ViewRoot可能比较陌生，但是其作用非常重大。所有View的绘制以及事件分发等交互都是通过它来执行或传递的。

ViewRoot对应**ViewRootImpl类**，它是连接**WindowManagerService**和**DecorView**的纽带，View的三大流程(measure)，布局(layout)，绘制(draw)均通过ViewRoot来完成。

ViewRoot并不属于View树的一份子。从源码实现上来看，它既非View的子类，也非View的父类，但是，它实现了ViewParent接口，这让它可以作为View的

名义上的父视图。RootView 继承了 Handler 类，可以接收事件并分发，Android 的所有触屏事件、按键事件、界面刷新等事件都是通过 ViewRoot 进行分发的。下面结构图可以清晰的揭示四者之间的关系：



二、DecorView 的创建

这部分内容主要讲 DecorView 是怎么一层层嵌套在 Activity, PhoneWindow 中的，以及 DecorView 如何加载内部布局。

`setContentView`

先是从 Activity 中的 `setContentView()` 开始。

```
public void setContentView(@LayoutRes int layoutResID) {  
    getWindow().setContentView(layoutResID);  
    initWindowDecorActionBar();  
}
```

可以看到实际是交给 Window 装载视图。下面来看看 Activity 是怎么获得 Window 对象的？

```
final void attach(Context context, ActivityThread aThread,  
    Instrumentation instr, IBinder token, int ident,  
    Application application, Intent intent, ActivityInfo info,  
    CharSequence title, Activity parent, String id,
```

```
NonConfigurationInstances lastNonConfigurationInstances,  
  
        Configuration config, String referrer, IVoiceInteractor voice  
Interactor,  
  
        Window window) {  
  
        .....  
  
        mWindow = new PhoneWindow(this, window); // 创建一个 Window 对象  
  
        mWindow.setWindowControllerCallback(this);  
  
        mWindow.setCallback(this); // 设置回调，向 Activity 分发点击或状态改变  
等事件  
  
        mWindow.setOnWindowDismissedCallback(this);  
  
        .....  
  
        mWindow.set WindowManager(  
  
                (WindowManager) context.getSystemService(Context.WINDOW_SE  
RVICE),  
  
                mToken, mComponent.flattenToString(),  
  
                (info.flags & ActivityInfo.FLAG_HARDWARE_ACCELERATED) !=  
0); // 给 Window 设置 WindowManager 对象  
  
        .....  
    }  
}
```

在 Activity 中的 attach() 方法中，生成了 PhoneWindow 实例。既然有了 Window 对象，那么我们就可以设置 DecorView 给 Window 对象了。

```
public void setContentView(int layoutResID) {  
  
    if (mContentParent == null) { // mContentParent 为空，创建一个 DecroV  
iew  
  
        installDecor();
```

```
    } else {

        mContentParent.removeAllViews(); //mContentParent 不为空，删除其中的 View

    }

    mLayoutInflater.inflate(layoutResID, mContentParent); //为 mContentParent 添加子 View, 即 Activity 中设置的布局文件

    final Callback cb = getCallback();

    if (cb != null && !isDestroyed()) {

        cb.onContentChanged(); //回调通知，内容改变

    }

}
```

看了下来，可能有一个疑惑：**mContentParent** 到底是什么？
就是前面布局中@**android:id/content** 所对应的 FrameLayout。
通过上面的流程我们大致可以了解先在 PhoneWindow 中创建了一个 DecroView，其中创建的过程中可能根据 Theme 不同，加载不同的布局格式，例如有没有 Title，或有没有 ActionBar 等，然后再向 mContentParent 中加入子 View，即 Activity 中设置的布局。到此位置，视图一层层嵌套添加上了。
下面具体来看看 **installDecor()** 方法，怎么创建的 DecroView，并设置其整体布局？

```
private void installDecor() {

    if (mDecor == null) {

        mDecor = generateDecor(); //生成 DecroView

        mDecor.setDescendantFocusability(ViewGroup.FOCUS_AFTER_DESCENDANTS);

        mDecor.setIsRootNamespace(true);

        if (!mInvalidatePanelMenuPosted && mInvalidatePanelMenuFeatures != 0) {

            mDecor.postOnAnimation(mInvalidatePanelMenuRunnable);

        }

    }

}
```

```
}

if (mContentParent == null) {

    mContentParent = generateLayout(mDecor); // 为 DecorView 设置布局格
式，并返回 mContentParent

    ...

}

}}
```

再来看看 generateDecor()

```
protected DecorView generateDecor() {

    return new DecorView(getContext(), -1);

}
```

很简单，创建了一个 DecorView。

再看 generateLayout

```
protected ViewGroup generateLayout(DecorView decor) {

    // 从主题文件中获取样式信息

    TypedArray a = getWindowStyle();

    .....

    if (a.getBoolean(R.styleable.Window_noTitle, false)) {

        requestFeature(FEATURE_NO_TITLE);

    } else if (a.getBoolean(R.styleable.Window_windowActionBar, fals
e)) {

        // Don't allow an action bar if there is no title.

        requestFeature(FEATURE_ACTION_BAR);

    }

}
```

```
    }

    .....

    // 根据主题样式，加载窗口布局

    int layoutResource;

    int features = getLocalFeatures();

    // System.out.println("Features: 0x" + Integer.toHexString(features));

    if ((features & (1 << FEATURE_SWIPE_TO_DISMISS)) != 0) {

        layoutResource = R.layout.screen_swipe_dismiss;

    } else if(...){

        ...

    }

    View in = mLayoutInflater.inflate(layoutResource, null);      //加载 layoutResource

    //往 DecorView 中添加子 View，即文章开头介绍 DecorView 时提到的布局格式，那只是一个例子，根据主题样式不同，加载不同的布局。

    decor.addView(in, new ViewGroup.LayoutParams(MATCH_PARENT, MATCH_PARENT));

    mContentRoot = (ViewGroup) in;

    ViewGroup contentParent = (ViewGroup) findViewById(ID_ANDROID_CONTENT); // 这里获取的就是 mContentParent

    if (contentParent == null) {
```

```
        throw new RuntimeException("Window couldn't find content container view");

    }

    if ((features & (1 << FEATURE_INDETERMINATE_PROGRESS)) != 0) {

        ProgressBar progress = getCircularProgressBar(false);

        if (progress != null) {

            progress.setIndeterminate(true);

        }

    }

    if ((features & (1 << FEATURE_SWIPE_TO_DISMISS)) != 0) {

        registerSwipeCallbacks();

    }

    // Remaining setup -- of background and title -- that only applies
    // to top-level windows.

    ...

    return contentParent;
}
```

虽然比较复杂，但是逻辑还是很清楚的。先从主题中获取样式，然后根据样式，加载对应的布局到 `DecorView` 中，然后从中获取 `mContentParent`。获得到之后，可以回到上面的代码，为 `mContentParent` 添加 `View`，即 `Activity` 中的布局。

以上就是 DecorView 的创建过程，其实到 `installDecor()` 就已经介绍了，后面只是具体介绍其中的逻辑。

三、DecorView 的显示

以上仅仅是将 DecorView 建立起来。通过 `setContentView()` 设置的界面，为什么在 `onResume()` 之后才对用户可见呢？

这就要从 ActivityThread 开始说起了，由于个人也是一知半解，仅仅阐述下个人理解吧。

```
private void handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {  
    // 就是在这里调用了 Activity.attach() 呀，接着调用了 Activity.onCreate() 和  
    // Activity.onStart() 生命周期，  
  
    // 但是由于只是初始化了 mDecor，添加了布局文件，还没有把  
  
    // mDecor 添加到负责 UI 显示的 PhoneWindow 中，所以这时候对用户来说，是不可见的  
  
    Activity a = performLaunchActivity(r, customIntent);  
  
    .....  
  
    if (a != null) {  
        // 这里面执行了 Activity.onResume()  
  
        handleResumeActivity(r.token, false, r.isForward,  
            !r.activity.mFinished && !r.startsNotResumed);  
  
        if (!r.activity.mFinished && r.startsNotResumed) {  
            try {  
                ....  
            } catch (Exception e) {  
                Log.w("ActivityThread", "handleLaunchActivity: " + e);  
            }  
        }  
    }  
}
```

```
    r.activity.mCalled = false;

    //执行 Activity.onPause()

    mInstrumentation.callActivityOnPause(r.activity);

}

}

}}
```

重点看下 handleResumeActivity(), 在这其中，DecorView 将会显示出来，同时重要的一个角色：ViewRoot 也将登场。

```
final void handleResumeActivity(IBinder token,
                                boolean clearHide, boolean isForward, boolean reallyResume) {

    //这个时候，Activity.onResume()已经调用了，但是现在界面还是不可见的

    ActivityClientRecord r = performResumeActivity(token, clearHi
de);

    if (r != null) {

        final Activity a = r.activity;

        if (r.window == null && !a.mFinished && willBeVisible) {

            r.window = r.activity.getWindow();

            View decor = r.window.getDecorView();

            //decor 对用户不可见

            decor.setVisibility(View.INVISIBLE);

            ViewManager wm = a.getWindowManager();
```

```
WindowManager.LayoutParams l = r.window.getAttributes();

a.mDecor = decor;

l.type = WindowManager.LayoutParams.TYPE_BASE_APPLICATION;

if (a.mVisibleFromClient) {

    a.mWindowAdded = true;

    //被添加进 WindowManager 了，但是这个时候，还是不可见的

    wm.addView(decor, l);

}

if (!r.activity.mFinished && willBeVisible

&& r.activity.mDecor != null && !r.hideForNow) {

    //在这里，执行了重要的操作，使得 DecorView 可见

    if (r.activity.mVisibleFromClient) {

        r.activity.makeVisible();

    }

}

}
```

当我们执行了 Activity.makeVisible()方法之后，界面才对我们是可见的。

```
void makeVisible() {

    if (!mWindowAdded) {
```

```
        ViewManager wm = getWindowManager();

        wm.addView(mDecor, getWindow().getAttributes());//将DecorView
添加到 WindowManager

        mWindowAdded = true;

    }

    mDecor.setVisibility(View.VISIBLE);//DecorView 可见

}
```

到此 DecorView 便可见，显示在屏幕中。但是在这其中,wm.addView(mDecor, getWindow().getAttributes());起到了重要的作用，因为其内部创建了一个 ViewRootImpl 对象，负责绘制显示各个子 View。

具体来看 addView()方法，因为 WindowManager 是个接口，具体是交给 WindowManagerImpl 来实现的。

```
public final class WindowManagerImpl implements WindowManager {

    private final WindowManagerGlobal mGlobal = WindowManagerGlobal.getInstance();

    ...

    @Override

    public void addView(View view, ViewGroup.LayoutParams params) {

        mGlobal.addView(view, params, mDisplay, mParentWindow);

    }
}
```

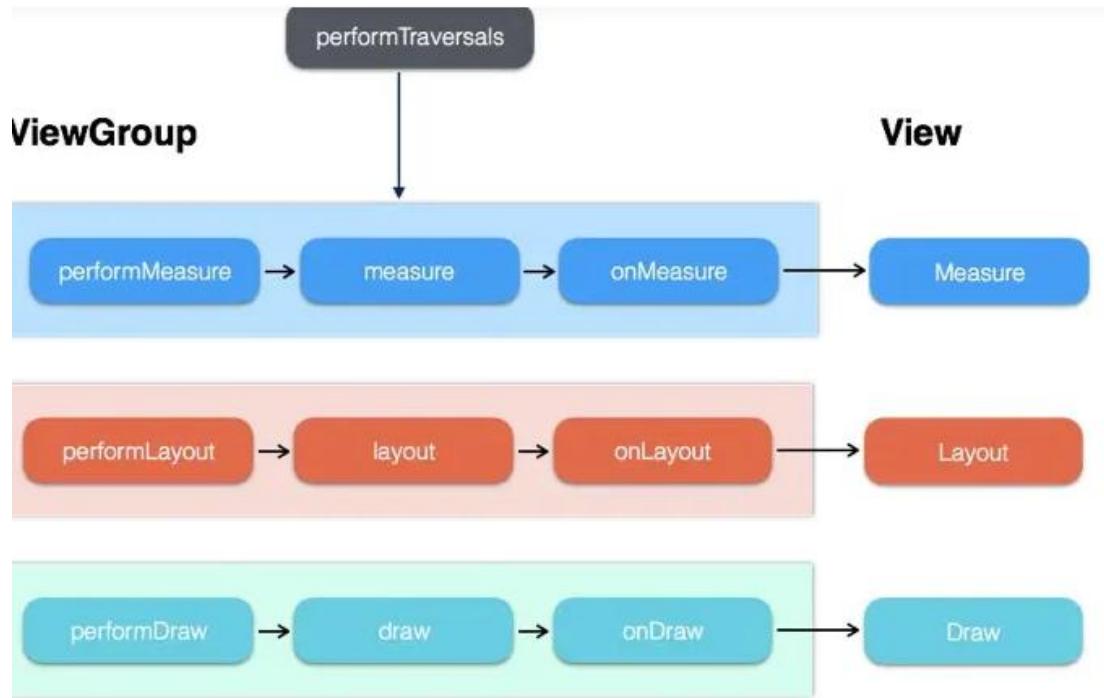
交给 WindowManagerGlobal 的 addView()方法去实现

```
public void addView(View view, ViewGroup.LayoutParams params,
                     Display display, Window parentWindow) {

    final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams)params;
    .....
}
```

```
synchronized (mLock) {  
  
    ViewRootImpl root;  
  
    //实例化一个 ViewRootImpl 对象  
  
    root = new ViewRootImpl(view.getContext(), display);  
  
    view.setLayoutParams(wparams);  
  
    .....  
  
    mViews.add(view);  
  
    mRoots.add(root);  
  
    mParams.add(wparams);  
  
}  
  
try {  
  
    //将 DecorView 交给 ViewRootImpl  
  
    root.setView(view, wparams, panelParentView);  
  
} catch (RuntimeException e) {  
  
}  
  
}  
}
```

看到其中实例化了 ViewRootImpl 对象，然后调用其 setView() 方法。其中 setView() 方法经过一些列折腾，最终调用了 performTraversals() 方法，**然后依照下图流程层层调用，完成绘制，最终界面才显示出来。**



其实 `ViewRootImpl` 的作用不止如此，还有许多功能，如事件分发。

要知道，当用户点击屏幕产生一个触摸行为，这个触摸行为则是通过底层硬件来传递捕获，然后交给 `ViewRootImpl`，接着将事件传递给 `DecorView`，而 `DecorView` 再交给 `PhoneWindow`，`PhoneWindow` 再交给 `Activity`，然后接下来就是我们常见的 `View` 事件分发了。

`硬件 -> ViewRootImpl -> DecorView -> PhoneWindow -> Activity`

十一、Android 的核心 Binder 多进程 AIDL

1、常见的 IPC 机制以及使用场景

2、为什么安卓要用 `binder` 进行跨进程传输

3、多进程带来的问题

文章、Android aidl Binder 框架浅析

1、概述

Binder 能干什么？Binder 可以提供系统中任何程序都可以访问的全局服务。这个功能当然是任何系统都应该提供的，下面我们简单看一下 Android 的 Binder 的框架

Android Binder 框架分为服务器接口、Binder 驱动、以及客户端接口；简单想一下，需要提供一个全局服务，那么全局服务那端即是服务器接口，任何程序即客户端接口，它们之间通过一个 Binder 驱动访问。

服务器端接口：实际上是 Binder 类的对象，该对象一旦创建，内部则会启动一个隐藏线程，会接收 Binder 驱动发送的消息，收到消息后，会执行 Binder 对象中的 `onTransact()` 函数，并按照该函数的参数执行不同的服务器端代码。

Binder 驱动：该对象也为 Binder 类的实例，客户端通过该对象访问远程服务。

客户端接口：获得 Binder 驱动，调用其 `transact()` 发送消息至服务器

如果大家对上述不了解，没关系，下面会通过例子来更好的说明，实践是检验真理的唯一标准嘛

2、AIDL 的使用

如果对 Android 比较熟悉，那么一定使用过 AIDL，如果你还不了解，那么也没关系，下面会使用一个例子展示 AIDL 的用法。

我们使用 AIDL 实现一个跨进程的加减法调用

1、服务端

新建一个项目，创建一个包名：com.zhy.calc.aidl，在包内创建一个 ICalcAIDL

文件：

```
1 package com.zhy.calc.aidl;
2 interface ICalcAIDL
3 {
4     int add(int x , int y);
5     int min(int x , int y );
6 }
```

注意，文件名为 ICalcAIDL.aidl

然后在项目的 gen 目录下会生成一个 ICalcAIDL.java 文件，暂时不贴这个文件

的代码了，后面会详细说明

然后我们在项目中新建一个 Service，代码如下：

```
package com.example.zhy_binder;

import com.zhy.calc.aidl.ILocalAIDL;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.RemoteException;
import android.util.Log;

public class CalcService extends Service
{
    private static final String TAG = "server";

    public void onCreate()
    {
        Log.e(TAG, "onCreate");
    }

    public IBinder onBind(Intent t)
    {
        Log.e(TAG, "onBind");
        return mBinder;
    }

    public void onDestroy()
```

```
{  
    Log.e(TAG, "onDestroy");  
    super.onDestroy();  
}  
  
public boolean onUnbind(Intent intent)  
{  
    Log.e(TAG, "onUnbind");  
    return super.onUnbind(intent);  
}  
  
public void onRebind(Intent intent)  
{  
    Log.e(TAG, "onRebind");  
    super.onRebind(intent);  
}  
  
private final ICalcAIDL.Stub mBinder = new ICalcAIDL.Stub()  
{  
  
    @Override  
    public int add(int x, int y) throws RemoteException  
    {  
        return x + y;  
    }  
  
    @Override  
    public int min(int x, int y) throws RemoteException  
    {  
        return x - y;  
    }  
};  
}
```

在此 Service 中, 使用生成的 ICalcAIDL 创建了一个 mBinder 的对象, 并在 Service 的 onBind 方法中返回

最后记得在 AndroidManifest 中注册

```
1 <service android:name="com.example.zhy_binder.CalcService" >
2     <intent-filter>
3         <action android:name="com.zhy.aidl.calc" />
4
5         <category android:name="android.intent.category.DEFAULT" />
6     </intent-filter>
7 </service>
```

复制

在此 Service 中, 使用生成的 ICalcAIDL 创建了一个 mBinder 的对象, 并在 Service 的 onBind 方法中返回

最后记得在 AndroidManifest 中注册

```
<LinearLayout 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:orientation="vertical" >

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:onClick="bindService"
        android:text="BindService" />

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:onClick="unbindService"
        android:text="UnbindService" />

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:onClick="addInvoked"
        android:text="12+12" />

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:onClick="minInvoked"
        android:text="50-12" />

</LinearLayout>
```

主 Activity

```
package com.example.zhy_binder_client;
```

```
import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.util.Log;
import android.view.View;
import android.widget.Toast;

import com.zhy.calc.aidl.ILocalAIDL;

public class MainActivity extends Activity
{
    private ILocalAIDL mCalcAidl;

    private ServiceConnection mServiceConn = new ServiceConnection()
    {
        @Override
        public void onServiceDisconnected(ComponentName name)
        {
            Log.e("client", "onServiceDisconnected");
            mCalcAidl = null;
        }

        @Override
        public void onServiceConnected(ComponentName name, IBinder service)
        {
            Log.e("client", "onServiceConnected");
            mCalcAidl = ILocalAIDL.Stub.asInterface(service);
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

    }

    /**

```

```
* 点击 BindService 按钮时调用
* @param view
*/
public void bindService(View view)
{
    Intent intent = new Intent();
    intent.setAction("com.zhy.aidl.calc");
    bindService(intent, mServiceConn, Context.BIND_AUTO_CREATE);
}

/**
 * 点击 unBindService 按钮时调用
 * @param view
*/
public void unbindService(View view)
{
    unbindService(mServiceConn);
}

/**
 * 点击 12+12 按钮时调用
 * @param view
*/
public void addInvoked(View view) throws Exception
{

    if (mCalcAidl != null)
    {
        int addRes = mCalcAidl.add(12, 12);
        Toast.makeText(this, addRes + "", Toast.LENGTH_SHORT).show();
    } else
    {
        Toast.makeText(this, "服务器被异常杀死, 请重新绑定服务端",
Toast.LENGTH_SHORT)
        .show();
    }
}

/**
 * 点击 50-12 按钮时调用
 * @param view
*/
public void minInvoked(View view) throws Exception
{
```

```
if (mCalcAidl != null)
{
    int addRes = mCalcAidl.min(58, 12);
    Toast.makeText(this, addRes + "", Toast.LENGTH_SHORT).show();
} else
{
    Toast.makeText(this, "服务端未绑定或被异常杀死, 请重新绑定服务端",
Toast.LENGTH_SHORT)
    .show();
}

}
```

很标准的绑定服务的代码。

直接看运行结果：



我们首先点击 BindService 按钮，查看 log

```
1 | 08-09 22:56:38.959: E/server(29692): onCreate  
2 | 08-09 22:56:38.959: E/server(29692): onBind  
3 | 08-09 22:56:38.959: E/client(29477): onServiceConnected
```

复制

可以看到，点击 BindService 之后，服务端执行了 `onCreate` 和 `onBind` 的方法，并且客户端执行了 `onServiceConnected` 方法，标明服务器与客户端已经联通

然后点击 $12+12$ ， $50-12$ 可以成功的调用服务端的代码并返回正确的结果

下面我们再点击 unBindService

08-09 22:59:25.567: E/server(29692): onUnbind

08-09 22:59:25.567: E/server(29692): onDestroy

由于我们当前只有一个客户端绑定了此 Service，所以 Service 调用了 onUnbind 和 onDestroy

然后我们继续点击 12+12 , 50-12 , 通过上图可以看到，依然可以正确执行，

也就是说即使 onUnbind 被调用，连接也是不会断开的，那么什么时候会端口呢？

即当服务端被异常终止的时候，比如我们现在在手机的正在执行的程序中找到该服务：



08-09 23:04:21 433: / 30146 :

可以看到调用了 onServiceDisconnected 方法，此时连接被断开，现在点击 12+12, 50-12 的按钮，则会弹出 Toast 服务端断开的提示。

说了这么多，似乎和 Binder 框架没什么关系，下面我们来具体看一看 AIDL 为什么做了些什么。

3、分析 AIDL 生成的代码

1、服务端

先看服务端的代码，可以看到我们服务端提供的服务是由

```
private final ICalcAIDL.Stub mBinder = new ICalcAIDL.Stub()
```

```
{
```

```
@Override
```

```
public int add(int x, int y) throws RemoteException
```

```
{
```

```
    return x + y;
```

```
}
```

```
@Override
```

```
public int min(int x, int y) throws RemoteException
```

```
{
```

```
    return x - y;
```

```
}
```

```
};
```

ICalcAIDL.Stub 来执行的，让我们来看看 Stub 这个类的声明：

```
public static abstract class Stub extends android.os.Binder implements  
com.zhy.calc.aidl.ICalcAIDL
```

清楚的看到这个类是 **Binder** 的子类，是不是符合我们文章开通所说的服务端其实是一个 **Binder** 类的实例

接下来看它的 **onTransact()**方法：

```
@Override public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel  
reply, int flags) throws android.os.RemoteException  
{  
    switch (code)  
    {  
        case INTERFACE_TRANSACTION:  
        {  
            reply.writeString(DESCRIPTOR);  
            return true;  
        }  
        case TRANSACTION_add:  
        {  
            data.enforceInterface(DESCRIPTOR);  
            int _arg0;  
            _arg0 = data.readInt();  
            int _arg1;  
            _arg1 = data.readInt();  
            int _result = this.add(_arg0, _arg1);  
            reply.writeNoException();  
            reply.writeInt(_result);  
            return true;  
        }  
        case TRANSACTION_min:  
        {  
            data.enforceInterface(DESCRIPTOR);  
            int _arg0;  
            _arg0 = data.readInt();  
            int _arg1;  
            _arg1 = data.readInt();  
            int _result = this.min(_arg0, _arg1);  
            reply.writeNoException();  
            reply.writeInt(_result);  
            return true;  
        }  
    }  
}
```

```
        return super.onTransact(code, data, reply, flags);
    }
```

文章开头也说到服务端的 **Binder** 实例会根据客户端依靠 **Binder** 驱动发来的消息，执行 **onTransact** 方法，然后由其参数决定执行服务端的代码。

可以看到 **onTransact** 有四个参数

code , **data** , **replay** , **flags**

code 是一个整形的唯一标识，用于区分执行哪个方法，客户端会传递此参数，告诉服务端执行哪个方法

data 客户端传递过来的参数

replay 服务器返回回去的值

flags 标明是否有返回值，0 为有（双向），1 为没有（单向）

我们仔细看 **case TRANSACTION_min** 中的代码

data.enforceInterface(DESCRIPTOR);与客户端的 **writeInterfaceToken** 对用，标识远程服务的名称

```
int _arg0;  
  
_arg0 = data.readInt();  
  
int _arg1;  
  
_arg1 = data.readInt();
```

接下来分别读取了客户端传入的两个参数

```
int _result = this.min(_arg0, _arg1);

reply.writeNoException();

reply.writeInt(_result);
```

然后执行 this.min , 即我们实现的 min 方法 ; 返回 result 由 reply 写回。

add 同理 , 可以看到服务端通过 AIDL 生成 Stub 的类 , 封装了服务端本来需要写的代码。

2、客户端

客户端主要通过 ServiceConnected 与服务端连接

```
private ServiceConnection mServiceConn = new ServiceConnection()
{
    @Override
    public void onServiceDisconnected(ComponentName name)
    {
        Log.e("client", "onServiceDisconnected");
        mCalcAidl = null;
    }

    @Override
    public void onServiceConnected(ComponentName name, IBinder service)
    {
        Log.e("client", "onServiceConnected");
        mCalcAidl = ICalcAIDL.Stub.asInterface(service);
    }
};
```

如果你比较敏锐，应该会猜到这个 onServiceConnected 中的 IBinder 实例，其实这就是我们文章开通所说的 Binder 驱动，也是一个 Binder 实例

在 ICalcAIDL.Stub.asInterface 中最终调用了：

这个 `Proxy` 实例传入了我们的 `Binder` 驱动，并且封装了我们调用服务端的代码，文章开头说，客户端会通过 `Binder` 驱动的 `transact()`方法调用服务端代码

直接看 `Proxy` 中的 `add` 方法

```
@Override public int add(int x, int y) throws android.os.RemoteException
{
    android.os.Parcel _data = android.os.Parcel.obtain();
    android.os.Parcel _reply = android.os.Parcel.obtain();
    int _result;
    try {
        _data.writeInterfaceToken(DESCRIPTOR);
        _data.writeInt(x);
        _data.writeInt(y);
        mRemote.transact(Stub.TRANSACTION_add, _data, _reply, 0);
        _reply.readException();
        _result = _reply.readInt();
    }
    finally {
        _reply.recycle();
        _data.recycle();
    }
    return _result;
}
```

首先声明两个 `Parcel` 对象，一个用于传递数据，一个用户接收返回的数据

`_data.writeInterfaceToken(DESCRIPTOR);`与服务器端的 `enforceInterface` 对应

```
_data.writeInt(x);
_data.writeInt(y);写入需要传递的参数
```

```
mRemote.transact(Stub.TRANSACTION_add, _data, _reply, 0);
```

终于看到了我们的 `transact` 方法，第一个对应服务端的 `code`,`_data`,`_repay` 分别对应服务端的 `data` , `reply` , 0 表示是双向的

```
_reply.readException();

_result = _reply.readInt();
```

最后读出我们服务端返回的数据 然后 return。可以看到和服务端的 onTransact 基本是一行一行对应的。

到此 ,我们已经通过 AIDL 生成的代码解释了 Android Binder 框架的工作原理。 Service 的作用其实就是为我们创建 Binder 驱动 , 即服务端与客户端连接的桥梁。

AIDL 其实通过我们写的 aidl 文件 , 帮助我们生成了一个接口 , 一个 Stub 类用于服务端 , 一个 Proxy 类用于客户端调用。那么我们是否可以不通过写 AIDL 来实现远程的通信呢 ? 下面向大家展示如何完全不依赖 AIDL 来实现客户端与服务端的通信。

4、不依赖 AIDL 实现程序间通讯

1、服务端代码

我们新建一个 CalcPlusService.java 用于实现两个数的乘和除

```
package com.example.zhy_binder;

import android.app.Service;
import android.content.Intent;
import android.os.Binder;
```

```
import android.os.IBinder;
import android.os.Parcel;
import android.os.RemoteException;
import android.util.Log;

public class CalcPlusService extends Service
{
    private static final String DESCRIPTOR = "CalcPlusService";
    private static final String TAG = "CalcPlusService";

    public void onCreate()
    {
        Log.e(TAG, "onCreate");
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId)
    {
        Log.e(TAG, "onStartCommand");

        return super.onStartCommand(intent, flags, startId);
    }
}
```

```
public IBinder onBind(Intent t)

{
    Log.e(TAG, "onBind");

    return mBinder;

}
```

```
public void onDestroy()

{
    Log.e(TAG, "onDestroy");

    super.onDestroy();

}
```

```
public boolean onUnbind(Intent intent)

{
    Log.e(TAG, "onUnbind");

    return super.onUnbind(intent);

}
```

```
public void onRebind(Intent intent)

{
    Log.e(TAG, "onRebind");

    super.onRebind(intent);

}
```

```
}
```

```
private MyBinder mBinder = new MyBinder();
```

```
private class MyBinder extends Binder
```

```
{
```

```
    @Override
```

```
    protected boolean onTransact(int code, Parcel data, Parcel reply,
```

```
        int flags) throws RemoteException
```

```
{
```

```
    switch (code)
```

```
{
```

```
    case 0x110:
```

```
{
```

```
        data.enforceInterface(DESCRIPTOR);
```

```
        int _arg0;
```

```
        _arg0 = data.readInt();
```

```
        int _arg1;
```

```
        _arg1 = data.readInt();
```

```
        int _result = _arg0 * _arg1;
```

```
        reply.writeNoException();
```

```
        reply.writeInt(_result);
```

```
        return true;

    }

    case 0x111:
    {
        data.enforceInterface(DESCRIPTOR);

        int _arg0;
        _arg0 = data.readInt();

        int _arg1;
        _arg1 = data.readInt();

        int _result = _arg0 / _arg1;

        reply.writeNoException();
        reply.writeInt(_result);

        return true;

    }

    return super.onTransact(code, data, reply, flags);
}

};

}

我们自己实现服务端，所以我们自定义了一个 Binder 子类，然后复写了其 onTransact 方法，我们指定服务的标识为 CalcPlusService，然后 0x110 为乘，0x111 为除；
```

记得在 AndroidManifest 中注册

```
1 <service android:name="com.example.zhy_binder.CalcPlusService" >
2     <intent-filter>
3         <action android:name="com.zhy.aidl.calcplus" />
4         <category android:name="android.intent.category.DEFAULT" />
5     </intent-filter>
6 </service>
```

复制

服务端代码结束。

2、客户端代码

单独新建了一个项目，代码和上例很类似

首先布局文件：

```
<LinearLayout
    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:orientation="vertical" >

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:onClick="bindService"
        android:text="BindService" />
```

```
<Button  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:onClick="unbindService"  
    android:text="UnbindService" />
```

```
<Button  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:onClick="mulInvoked"  
    android:text="50*12" />
```

```
<Button  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:onClick="divInvoked"  
    android:text="36/12" />
```

```
</LinearLayout>
```

可以看到加入了乘和除

然后是 Activity 的代码

```
package com.example.zhy_binder_client03;
```

```
import android.app.Activity;  
import android.content.ComponentName;  
import android.content.Context;  
import android.content.Intent;  
import android.content.ServiceConnection;  
import android.os.Bundle;  
import android.os.IBinder;  
import android.os.RemoteException;  
import android.util.Log;  
import android.view.View;  
import android.widget.Toast;  
  
public class MainActivity extends Activity  
{  
  
    private IBinder mPlusBinder;  
    private ServiceConnection mServiceConnPlus = new  
    ServiceConnection()  
    {  
        @Override  
        public void onServiceDisconnected(ComponentName name)  
    }  
}
```

```
    Log.e("client", "mServiceConnPlus onServiceDisconnected");

}

@Override
public void onServiceConnected(ComponentName name, IBinder
service)
{
    Log.e("client", " mServiceConnPlus onServiceConnected");
    mPlusBinder = service;
}

};

@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

}
```

```
public void bindService(View view)
{
    Intent intentPlus = new Intent();
    intentPlus.setAction("com.zhy.aidl.calcplus");
    boolean plus = bindService(intentPlus, mServiceConnPlus,
        Context.BIND_AUTO_CREATE);
    Log.e("plus", plus + "");
}
```

```
public void unbindService(View view)
{
    unbindService(mServiceConnPlus);
}
```

```
public void mulInvoked(View view)
```

```
{
```

```
    if (mPlusBinder == null)
    {
        Toast.makeText(this, "未连接服务端或服务端被异常杀死",
        Toast.LENGTH_SHORT).show();
    } else
```

```
{\n\n    android.os.Parcel _data = android.os.Parcel.obtain();\n\n    android.os.Parcel _reply = android.os.Parcel.obtain();\n\n    int _result;\n\n    try\n\n    {\n\n        _data.writeInterfaceToken("CalcPlusService");\n\n        _data.writeInt(50);\n\n        _data.writeInt(12);\n\n        mPlusBinder.transact(0x110, _data, _reply, 0);\n\n        _reply.readException();\n\n        _result = _reply.readInt();\n\n        Toast.makeText(this, _result + "",\n\n        Toast.LENGTH_SHORT).show();\n\n    } catch (RemoteException e)\n\n    {\n\n        e.printStackTrace();\n\n    } finally\n\n    {\n\n        _reply.recycle();\n\n        _data.recycle();\n\n    }\n\n}
```

```
        }

    }

}

public void divInvoked(View view)
{
    if (mPlusBinder == null)
    {
        Toast.makeText(this, "未连接服务端或服务端被异常杀死",
Toast.LENGTH_SHORT).show();
    } else
    {
        android.os.Parcel _data = android.os.Parcel.obtain();
        android.os.Parcel _reply = android.os.Parcel.obtain();
        int _result;
        try
        {
            _data.writeInterfaceToken("CalcPlusService");
            _data.writeInt(36);
            _data.writeInt(12);
        }
    }
}
```

```
mPlusBinder.transact(0x111, _data, _reply, 0);

_reply.readException();

_result = _reply.readInt();

Toast.makeText(this, _result + "",

Toast.LENGTH_SHORT).show();

}

} catch (RemoteException e)

{

e.printStackTrace();

}

} finally

{

_reply.recycle();

_data.recycle();

}

}

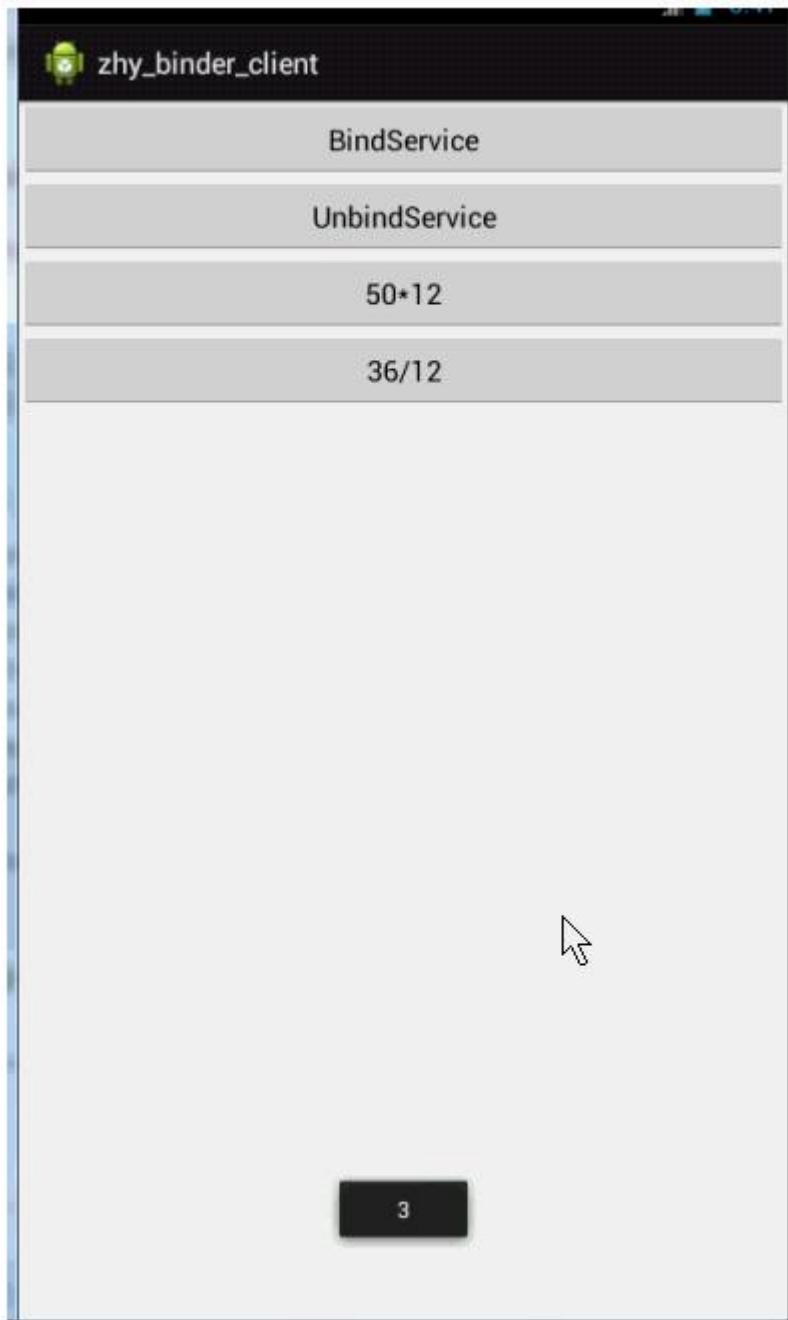
}

}
```

为了明了，我直接在 `mullInvoked` 里面写了代码，和服务端都没有抽象出一个接口。首先绑定服务时，通过 `onServiceConnected` 得到 Binder 驱动即 `mPlusBinder`:

然后准备数据，调用 `transact` 方法，通过 `code` 指定执行服务端哪个方法，代码和上面的分析一致。

下面看运行结果：



是不是很好的实现了我们两个应用程序间的通讯，并没有使用 aidl 文件，也从侧面分析了我们上述分析是正确的。

文章、Android 应用工程师的 Binder 原理剖析

一. 前言

这篇文章我酝酿了很久，参考了很多资料，读了很多源码，却依旧不敢下笔。生怕自己理解上还有偏差，对大家造成误解，贻笑大方。又怕自己理解不够透彻，无法用清晰直白的文字准确的表达出 Binder 的设计精髓。直到今天提笔写作时还依旧战战兢兢。

Binder 之复杂远远不是一篇文章就能说清楚的，本文想站在一个更高的维度来俯瞰 Binder 的设计，最终帮助大家形成一个完整的概念。对于应用层开发的同学来说，理解到本文这个程度也就差不多了。希望更加深入理解 Binder 实现机制的，可以阅读文末的参考资料以及相关源码。

二. Binder 概述

简单介绍下什么是 Binder。Binder 是一种进程间通信机制，基于开源的 OpenBinder 实现；OpenBinder 起初由 Be Inc. 开发，后由 Plam Inc. 接手。从字面上来解释 Binder 有胶水、粘合剂的意思，顾名思义就是粘和不同的进程，使之实现通信。对于 Binder 更全面的定义，等我们介绍完 Binder 通信原理后再做详细说明。

2.1 为什么必须理解 Binder ?

作为 Android 工程师的你，是不是常常会有这样的疑问：

- 为什么 Activity 间传递对象需要序列化？
- Activity 的启动流程是什么样的？
- 四大组件底层的通信机制是怎样的？
- AIDL 内部的实现原理是什么？
- 插件化编程技术应该从何学起？等等...

这些问题的背后都与 Binder 有莫大的关系，要弄懂上面这些问题理解 Binder 通信机制是必须的。

我们知道 Android 应用程序是由 Activity、Service、Broadcast Receiver 和 Content Provide 四大组件中的一个或者多个组成的。有时这些组件运行在同一进程，有时运行在不同的进程。这些进程间的通信就依赖于 Binder IPC 机制。不仅如此，Android 系统对应用层提供的各种服务如：ActivityManagerService、PackageManagerService 等都是基于 Binder IPC 机制来实现的。Binder 机制在 Android 中的位置非常重要，毫不夸张的说理解 Binder 是迈向 Android 高级工程的第一步。

2.2 为什么是 Binder ?

Android 系统是基于 Linux 内核的，Linux 已经提供了管道、消息队列、共享内存和 Socket 等 IPC 机制。那为什么 Android 还要提供 Binder 来实现 IPC 呢？主要是基于**性能、稳定性和安全性**几方面的原因。

性能

首先说说性能上的优势。Socket 作为一款通用接口，其传输效率低，开销大，主要用在跨网络的进程间通信和本机上进程间的低速通信。消息队列和管道采用存储-转发方式，即数据先从发送方缓存区拷贝到内核开辟的缓存区中，然后再从内核缓存区拷贝到接收方缓存区，至少有两次拷贝过程。共享内存虽然无需拷贝，但控制复杂，难以使用。Binder 只需要一次数据拷贝，性能上仅次于共享内存。

注：各种 IPC 方式数据拷贝次数，此表来源于 [Android Binder 设计与实现 - 设计篇](#)

IPC方式	数据拷贝次数
共享内存	0
Binder	1
Socket/管道/消息队列	2

稳定性

再说说稳定性，Binder 基于 C/S 架构，客户端（Client）有什么需求就丢给服务端（Server）去完成，架构清晰、职责明确又相互独立，自然稳定性更好。共享内存虽然无需拷贝，但是控制负责，难以使用。从稳定性的角度讲，Binder 机制是优于内存共享的。

安全性

另一方面就是安全性。Android 作为一个开放性的平台，市场上有各类海量的应用供用户选择安装，因此安全性对于 Android 平台而言极

其重要。作为用户当然不希望我们下载的 APP 偷偷读取我的通信录，上传我的隐私数据，后台偷跑流量、消耗手机电量。传统的 IPC 没有任何安全措施，完全依赖上层协议来确保。首先传统的 IPC 接收方无法获得对方可靠的进程用户 ID/进程 ID (UID/PID)，从而无法鉴别对方身份。Android 为每个安装好的 APP 分配了自己的 UID，故而进程的 UID 是鉴别进程身份的重要标志。传统的 IPC 只能由用户在数据包中填入 UID/PID，但这样不可靠，容易被恶意程序利用。可靠的身份标识只有由 IPC 机制在内核中添加。其次传统的 IPC 访问接入点是开放的，只要知道这些接入点的程序都可以和对端建立连接，不管怎样都无法阻止恶意程序通过猜测接收方地址获得连接。同时 Binder 既支持实名 Binder，又支持匿名 Binder，安全性高。

基于上述原因，Android 需要建立一套新的 IPC 机制来满足系统对稳定性、传输性能和安全性方面的要求，这就是 Binder。

最后用一张表格来总结下 Binder 的优势：

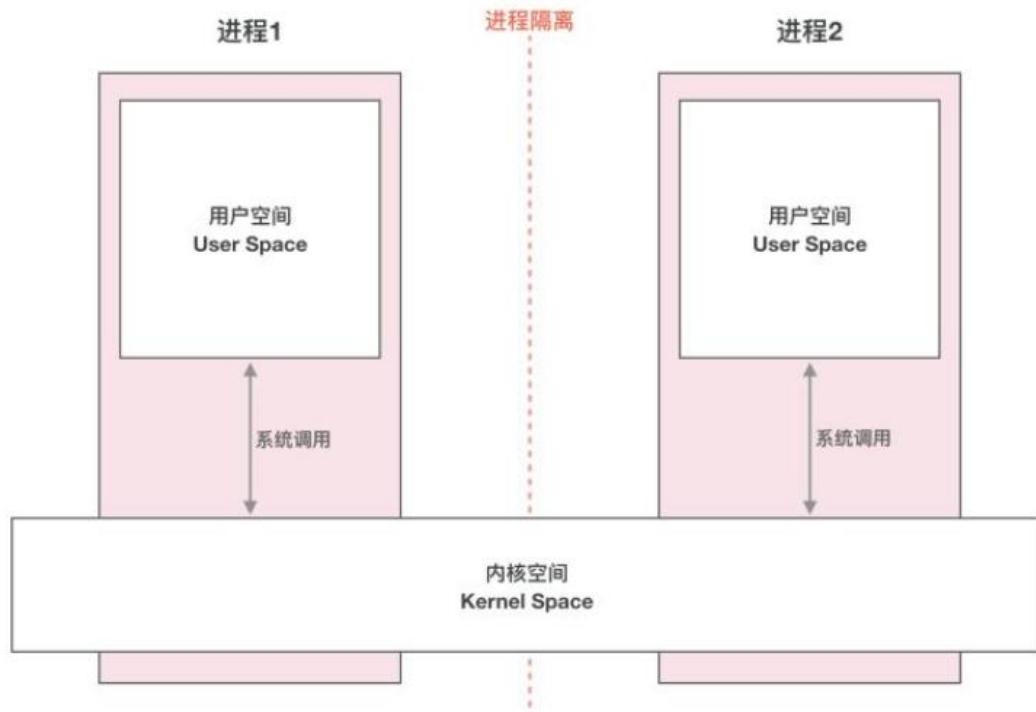
优势	描述
性能	只需要一次数据拷贝，性能上仅次于共享内存
稳定性	基于 C/S 架构，职责明确、架构清晰，因此稳定性好
安全性	为每个 APP 分配 UID，进程的 UID 是鉴别进程身份的重要标志

三. Linux 下传统的进程间通信原理

了解 Linux IPC 相关的概念和原理有助于我们理解 Binder 通信原理。因此，在介绍 Binder 跨进程通信原理之前，我们先聊聊 Linux 系统下传统的进程间通信是如何实现。

3.1 基本概念介绍

这里我们先从 Linux 中进程间通信涉及的一些基本概念开始介绍，然后逐步展开，向大家说明传统的进程间通信的原理。



上图展示了 Linux 中跨进程通信涉及到的一些基本概念：

- **进程隔离**
- **进程空间划分：用户空间(User Space)/内核空间(Kernel Space)**
- **系统调用：用户态/内核态**

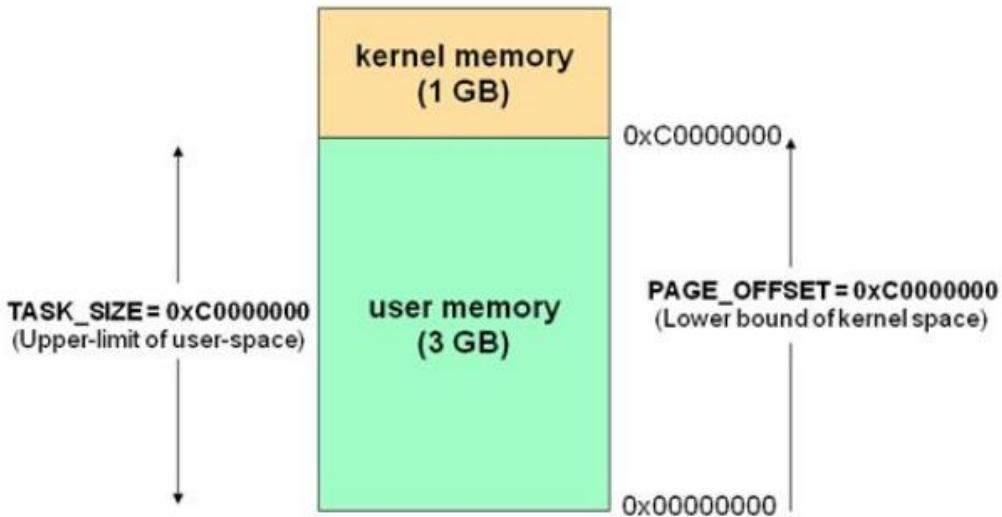
进程隔离

简单的说就是操作系统中，进程与进程间内存是不共享的。两个进程就像两个平行的世界，A 进程没法直接访问 B 进程的数据，这就是进程隔离的通俗解释。A 进程和 B 进程之间要进行数据交互就得采用特殊的通信机制：进程间通信（IPC）。

进程空间划分：用户空间(User Space)/内核空间(Kernel Space)

现在操作系统都是采用的虚拟存储器，对于 32 位系统而言，它的寻址空间（虚拟存储空间）就是 2 的 32 次方，也就是 4GB。操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也可以访问底层硬件设备的权限。为了保护用户进程不能直接操作内核，保证内核的安全，操作系统从逻辑上将虚拟空间划分为用户空间（User Space）和内核空间（Kernel Space）。针对 Linux 操作系统而言，将最高的 1GB 字节供内核使用，称为内核空间；较低的 3GB 字节供各进程使用，称为用户空间。

简单的说就是，内核空间（Kernel）是系统内核运行的空间，用户空间（User Space）是用户程序运行的空间。为了保证安全性，它们之间是隔离的。



系统调用：用户态与内核态

虽然从逻辑上进行了用户空间和内核空间的划分，但不可避免的用户空间需要访问内核资源，比如文件操作、访问网络等等。为了突破隔离限制，就需要借助**系统调用**来实现。系统调用是用户空间访问内核空间的唯一方式，保证了所有的资源访问都是在内核的控制下进行的，避免了用户程序对系统资源的越权访问，提升了系统安全性和稳定性。

Linux 使用两级保护机制 :0 级供系统内核使用 ,3 级供用户程序使用。

当一个任务（进程）执行系统调用而陷入内核代码中执行时，称进程处于**内核运行态（内核态）**。此时处理器处于特权级最高的（0 级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。

当进程在执行用户自己的代码的时候，我们称其处于**用户运行态（用户态）**。此时处理器在特权级最低的（3 级）用户代码中运行。

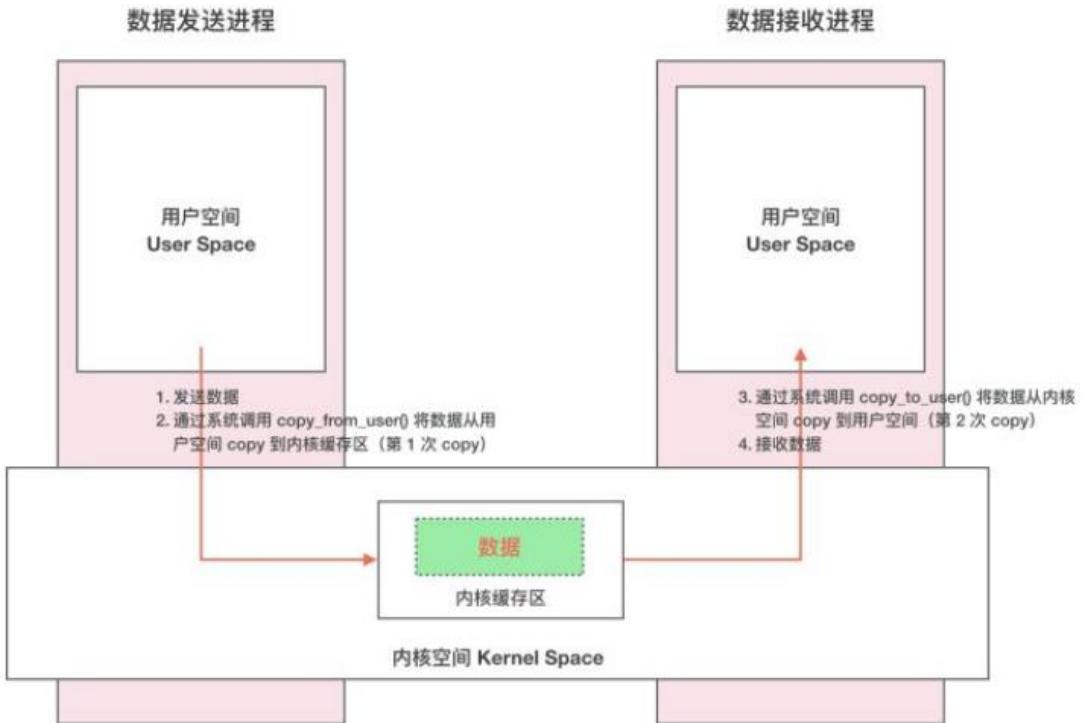
系统调用主要通过如下两个函数来实现：

```
copy_from_user() //将数据从用户空间拷贝到内核空间  
copy_to_user() //将数据从内核空间拷贝到用户空间
```

3.2 Linux 下的传统 IPC 通信原理

理解了上面的几个概念，我们再来看看传统的 IPC 方式中，进程之间是如何实现通信的。

通常的做法是消息发送方将要发送的数据存放在内存缓存区中，通过系统调用进入内核态。然后内核程序在内核空间分配内存，开辟一块内核缓存区，调用 `copyfromuser()` 函数将数据从用户空间的内存缓存区拷贝到内核空间的内核缓存区中。同样的，接收方进程在接收数据时在自己的用户空间开辟一块内存缓存区，然后内核程序调用 `copytouser()` 函数将数据从内核缓存区拷贝到接收进程的内存缓存区。这样数据发送方进程和数据接收方进程就完成了一次数据传输，我们称完成了一次进程间通信。如下图：



这种传统的 IPC 通信方式有两个问题：

- 性能低下，一次数据传递需要经历：内存缓存区 --> 内核缓存区 --> 内存缓存区，需要 2 次数据拷贝；
- 接收数据的缓存区由数据接收进程提供，但是接收进程并不知道需要多大的空间来存放将要传递过来的数据，因此只能开辟尽可能大的内存空间或者先调用 API 接收消息头来获取消息体的大小，这两种做法不是浪费空间就是浪费时间。

四. Binder 跨进程通信原理

理解了 Linux IPC 相关概念和通信原理，接下来我们正式介绍下 Binder IPC 的原理。

4.1 动态内核可加载模块 && 内存映射

正如前面所说，跨进程通信是需要内核空间做支持的。传统的 IPC 机制如管道、Socket 都是内核的一部分，因此通过内核支持来实现进程间通信自然是没问题的。但是 Binder 并不是 Linux 系统内核的一部分，那怎么办呢？这就得益于 Linux 的**动态内核可加载模块**（Loadable Kernel Module，LKM）的机制；模块是具有独立功能的程序，它可以被单独编译，但是不能独立运行。它在运行时被链接到内核作为内核的一部分运行。这样，Android 系统就可以通过动态添加一个内核模块运行在内核空间，用户进程之间通过这个内核模块作为桥梁来实现通信。

在 Android 系统中，这个运行在内核空间，负责各个用户进程通过 Binder 实现通信的内核模块就叫 **Binder 驱动**（Binder Driver）。

那么在 Android 系统中用户进程之间是如何通过这个内核模块（Binder 驱动）来实现通信的呢？难道是和前面说的传统 IPC 机制一样，先将数据从发送方进程拷贝到内核缓存区，然后再将数据从内核缓存区拷贝到接收方进程，通过两次拷贝来实现吗？显然不是，否则也不会有开篇所说的 Binder 在性能方面的优势了。

这就不得不通道 Linux 下的另一个概念：**内存映射**。

Binder IPC 机制中涉及到的内存映射通过 `mmap()` 来实现，`mmap()` 是操作系统中一种内存映射的方法。内存映射简单的讲就是将用户空间的一块内存区域映射到内核空间。映射关系建立后，用户对这块内存区域的修改可以直接反应到内核空间；反之内核空间对这段区域的修改也能直接反应到用户空间。

内存映射能减少数据拷贝次数，实现用户空间和内核空间的高效互动。两个空间各自的修改能直接反映在映射的内存区域，从而被对方空间及时感知。也正因为如此，内存映射能够提供对进程间通信的支持。

4.2 Binder IPC 实现原理

Binder IPC 正是基于内存映射（`mmap`）来实现的，但是 `mmap()` 通常是在有物理介质的文件系统上的。

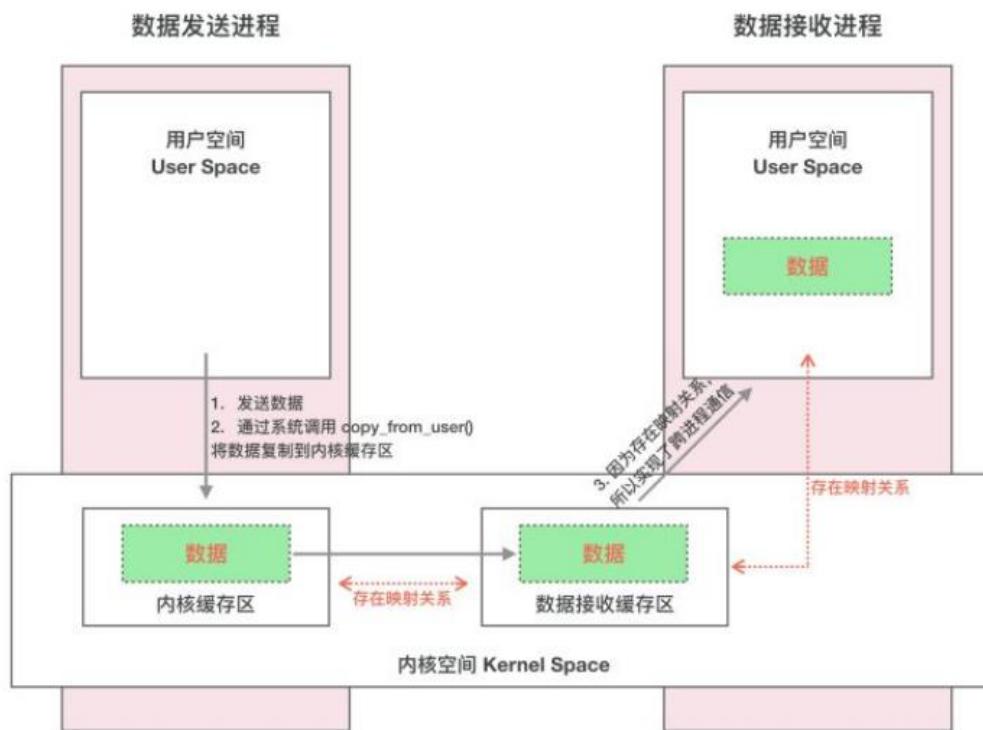
比如进程中的用户区域是不能直接和物理设备打交道的，如果想要把磁盘上的数据读取到进程的用户区域，需要两次拷贝（磁盘-->内核空间-->用户空间）；通常在这种场景下 `mmap()` 就能发挥作用，通过在物理介质和用户空间之间建立映射，减少数据的拷贝次数，用内存读写取代 I/O 读写，提高文件读取效率。

而 Binder 并不存在物理介质，因此 Binder 驱动使用 `mmap()` 并不是为了在物理介质和用户空间之间建立映射，而是用来在内核空间创建数据接收的缓存空间。

一次完整的 Binder IPC 通信过程通常就是这样：

- 首先 Binder 驱动在内核空间创建一个数据接收缓存区；
- 接着在内核空间开辟一块内核缓存区，建立**内核缓存区**和**内核中数据接收缓存区**之间的映射关系，以及**内核中数据接收缓存区**和**接收进程用户空间地址**的映射关系；
- 发送方进程通过系统调用 `copyfromuser()` 将数据 copy 到内核中的**内核缓存区**，由于内核缓存区和接收进程的用户空间存在内存映射，因此也就相当于把数据发送到了接收进程的用户空间，这样便完成了一次进程间的通信。

如下图：



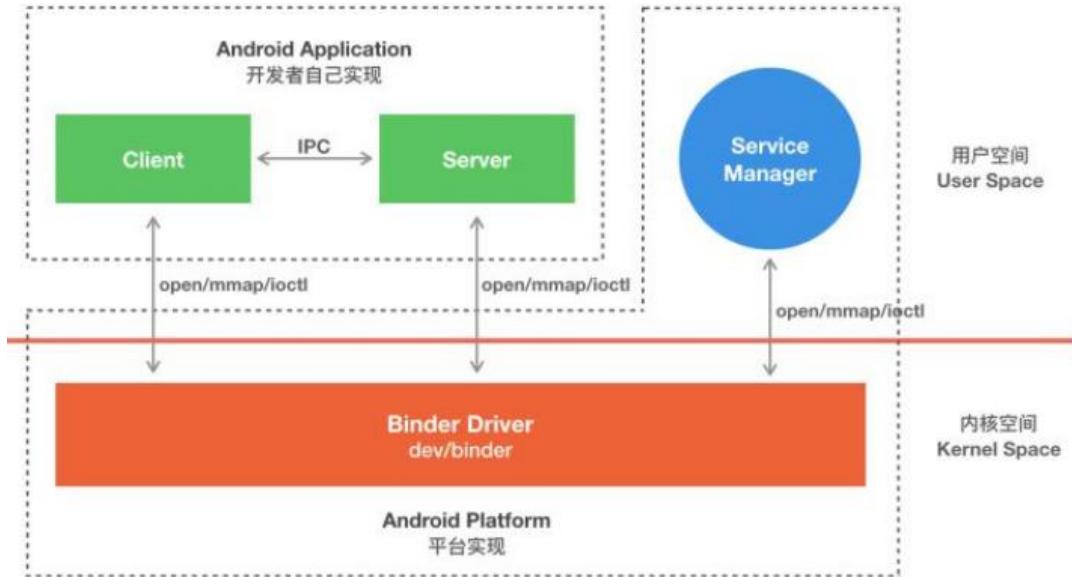
五. Binder 通信模型

介绍完 Binder IPC 的底层通信原理，接下来我们看看实现层面是如何设计的。

一次完整的进程间通信必然至少包含两个进程，通常我们称通信的双方分别为客户端进程（Client）和服务端进程（Server），由于进程隔离机制的存在，通信双方必然需要借助 Binder 来实现。

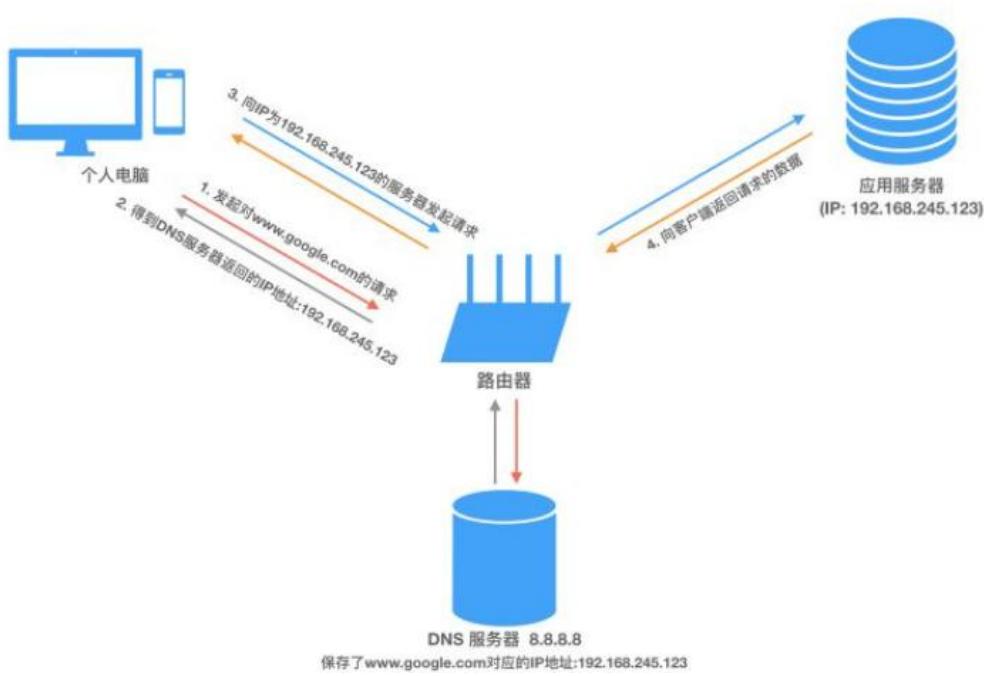
5.1 Client/Server/ServiceManager/驱动

前面我们介绍过，Binder 是基于 C/S 架构的。由一系列的组件组成，包括 Client、Server、ServiceManager、Binder 驱动。其中 Client、Server、Service Manager 运行在用户空间，Binder 驱动运行在内核空间。其中 Service Manager 和 Binder 驱动由系统提供，而 Client、Server 由应用程序来实现。Client、Server 和 ServiceManager 均是通过系统调用 open、mmap 和 ioctl 来访问设备文件 /dev/binder，从而实现与 Binder 驱动的交互来间接的实现跨进程通信。



Client、Server、ServiceManager、Binder 驱动这几个组件在通信过程中扮演的角色就如同互联网中服务器（Server）、客户端（Client）、DNS 域名服务器（ServiceManager）以及路由器（Binder 驱动）之前的关系。

通常我们访问一个网页的步骤是这样的：首先在浏览器输入一个地址，如 `google.com` 然后按下回车键。但是并没有办法通过域名地址直接找到我们要访问的服务器，因此需要首先访问 DNS 域名服务器，域名服务器中保存了 `google.com` 对应的 ip 地址 `10.249.23.13`，然后通过这个 ip 地址才能放到到 `google.com` 对应的服务器。



Binder 驱动

Binder 驱动就如同路由器一样，是整个通信的核心；驱动负责进程之间 Binder 通信的建立，Binder 在进程之间的传递，Binder 引用计数管理，数据包在进程之间的传递和交互等一系列底层支持。

ServiceManager 与实名 Binder

ServiceManager 和 DNS 类似，作用是将字符形式的 Binder 名字转化成 Client 中对该 Binder 的引用，使得 Client 能够通过 Binder 的名字获得对 Binder 实体的引用。注册了名字的 Binder 叫实名 Binder，就像网站一样除了有 IP 地址意外还有自己的网址。Server 创建了 Binder，并为它起一个字符形式，可读易记得名字，

将这个 Binder 实体连同名字一起以数据包的形式通过 Binder 驱动发送给 ServiceManager , 通知 ServiceManager 注册一个名为 “张三” 的 Binder , 它位于某个 Server 中。驱动为这个穿越进程边界的 Binder 创建位于内核中的实体节点以及 ServiceManager 对实体的引用 , 将名字以及新建的引用打包传给 ServiceManager 。 ServiceManger 收到数据后从中取出名字和引用填入查找表。

细心的读者可能会发现 , ServierManager 是一个进程 , Server 是另一个进程 , Server 向 ServiceManager 中注册 Binder 必然涉及到进程间通信。当前实现进程间通信又要用到进程间通信 , 这就好像蛋可以孵出鸡的前提却是要先找只鸡下蛋 ! Binder 的实现比较巧妙 , 就是预先创造一只鸡来下蛋。 ServiceManager 和其他进程同样采用 Bidner 通信 , ServiceManager 是 Server 端 , 有自己的 Binder 实体 , 其他进程都是 Client , 需要通过这个 Binder 的引用来实现 Binder 的注册 , 查询和获取。 ServiceManager 提供的 Binder 比较特殊 , 它没有名字也不需要注册。当一个进程使用 BINDER_SET_CONTEXT_MGR 命令将自己注册成 ServiceManager 时 Binder 驱动会自动为它创建

Binder 实体（这就是那只预先造好的那只鸡）。其次这个 Binder 实体的引用在所有 Client 中都固定为 0 而无需通过其它手段获得。也就是说，一个 Server 想要向 ServiceManager 注册自己的 Binder 就必须通过这个 0 号引用和 ServiceManager 的 Binder 通信。类比互联网，0 号引用就好比是域名服务器的地址，你必须预先动态或者手工配置好。要注意的是，这里说的 Client 是相对于 ServiceManager 而言的，一个进程或者应用程序可能是提供服务的 Server，但对于 ServiceManager 来说它仍然是个 Client。

Client 获得实名 Binder 的引用

Server 向 ServiceManager 中注册了 Binder 以后，Client 就能通过名字获得 Binder 的引用了。Client 也利用保留的 0 号引用向 ServiceManager 请求访问某个 Binder：我申请访问名字叫张三的 Binder 引用。ServiceManager 收到这个请求后从请求数据包中取出 Binder 名称，在查找表里找到对应的条目，取出对应的 Binder 引用作为回复发送给发起请求的 Client。从面向对象的角度看，Server 中的 Binder 实体现在有两个引用：一个位于 ServiceManager 中，一个位于发起请求的 Client 中。如果接下来有更多的 Client 请求该

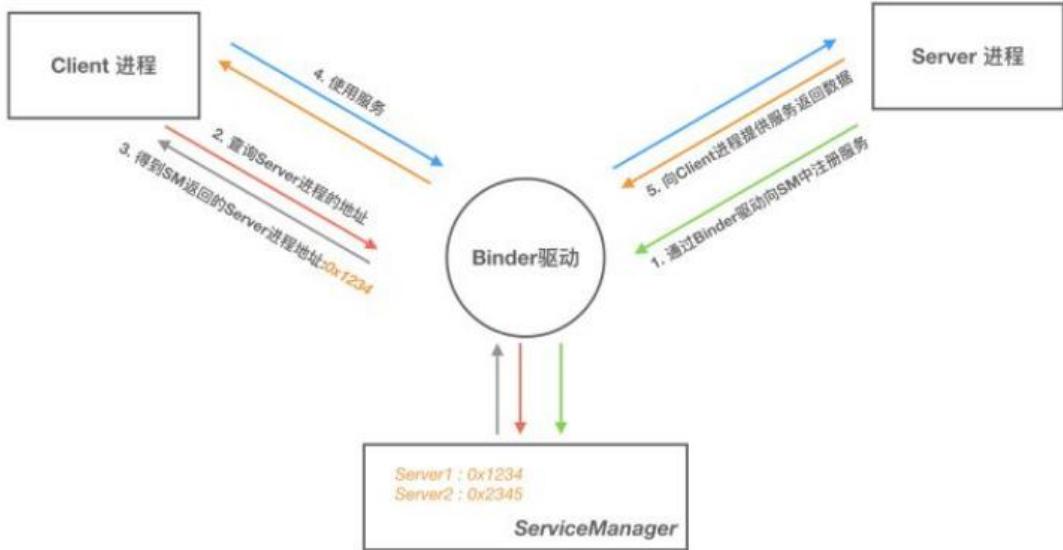
Binder , 系统中就会有更多的引用指向该 Binder , 就像 Java 中一个对象有多个引用一样。

5.2 Binder 通信过程

至此 , 我们大致能总结出 Binder 通信过程 :

- 首先 , 一个进程使用 BINDERSETCONTEXT_MGR 命令通过 Binder 驱动将自己注册成为 ServiceManager ;
- Server 通过驱动向 ServiceManager 中注册 Binder (Server 中的 Binder 实体) , 表明可以对外提供服务。驱动为这个 Binder 创建位于内核中的实体节点以及 ServiceManager 对实体的引用 , 将名字以及新建的引用打包传给 ServiceManager , ServiceManger 将其填入查找表。
- Client 通过名字 在 Binder 驱动的帮助下从 ServiceManager 中获取到对 Binder 实体的引用 , 通过这个引用就能实现和 Server 进程的通信。

我们看到整个通信过程都需要 Binder 驱动的接入。下图能更加直观的展现整个通信过程(为了进一步抽象通信过程以及呈现上的方便 , 下图我们忽略了 Binder 实体及其引用的概念) :

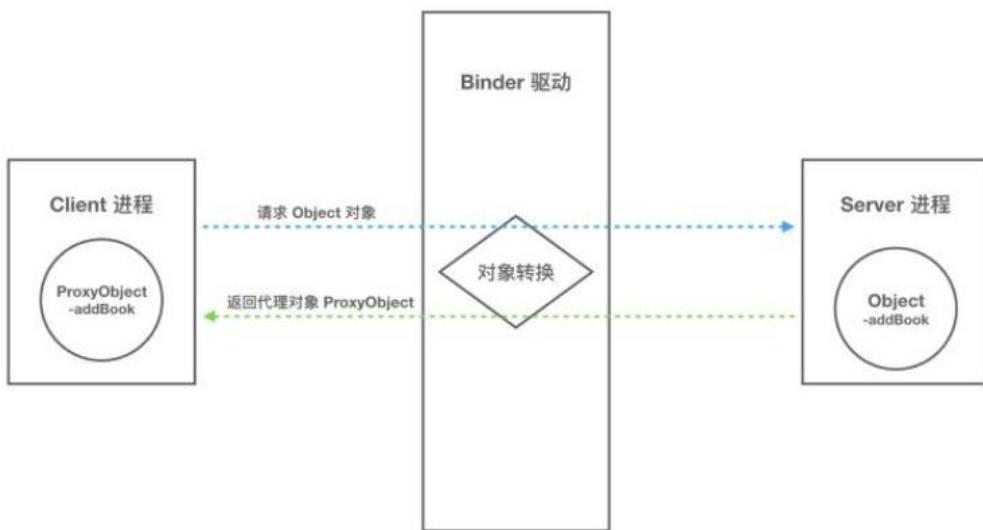


5.3 Binder 通信中的代理模式

我们已经解释清楚 Client、Server 借助 Binder 驱动完成跨进程通信的实现机制了，但是还有个问题会让我们困惑。A 进程想要 B 进程中某个对象 (object) 是如何实现的呢？毕竟它们分属不同的进程，A 进程 没法直接使用 B 进程中的 object。

前面我们介绍过跨进程通信的过程都有 Binder 驱动的参与，因此在数据流经 Binder 驱动的时候驱动会对数据做一层转换。当 A 进程想要获取 B 进程中的 object 时，驱动并不会真的把 object 返回给 A，而是返回了一个跟 object 看起来一模一样的代理对象 objectProxy，这个 objectProxy 具有和 object 一摸一样的方法，但是这些方法并没有 B 进程中 object 对象那些方法的能力，这些方法只需要把请求参数交给驱动即可。对于 A 进程来说和直接调用 object 中的方法是一样的。

当 Binder 驱动接收到 A 进程的消息后，发现这是个 objectProxy 就去查询自己维护的表单，一查发现这是 B 进程 object 的代理对象。于是就会去通知 B 进程调用 object 的方法，并要求 B 进程把返回结果发给自己。当驱动拿到 B 进程的返回结果后就会转发给 A 进程，一次通信就完成了。



5.4 Binder 的完整定义

现在我们可以对 Binder 做个更加全面的定义了：

- 从进程间通信的角度看，Binder 是一种进程间通信的机制；
- 从 Server 进程的角度看，Binder 指的是 Server 中的 Binder 实体对象；
- 从 Client 进程的角度看，Binder 指的是对 Binder 代理对象，是 Binder 实体对象的一个远程代理

- 从传输过程的角度看，Binder 是一个可以跨进程传输的对象；
Binder 驱动会对这个跨越进程边界的对象对一点点特殊处理，自动完成代理对象和本地对象之间的转换。
-

六. 手动编码实现跨进程调用

通常我们在做开发时，实现进程间通信用的最多的就是 AIDL。当我们定义好 AIDL 文件，在编译时编译器会帮我们生成代码实现 IPC 通信。借助 AIDL 编译以后的代码能帮助我们进一步理解 Binder IPC 的通信原理。

但是无论是从可读性还是可理解性上来看，编译器生成的代码对开发者并不友好。比如一个 BookManager.aidl 文件对应会生成一个 BookManager.java 文件，这个 java 文件包含了一个 BookManager 接口、一个 Stub 静态的抽象类和一个 Proxy 静态类。Proxy 是 Stub 的静态内部类，Stub 又是 BookManager 的静态内部类，这就造成了可读性和可理解性的问题。

Android 之所以这样设计其实是有道理的，因为当有多个 AIDL 文件的时候把 BookManager、Stub、Proxy 放在同一个文件里能有效避免 Stub 和 Proxy 重名的问题。

因此便于大家理解，下面我们来手动编写代码来实现跨进程调用。

6.1 各 Java 类职责描述

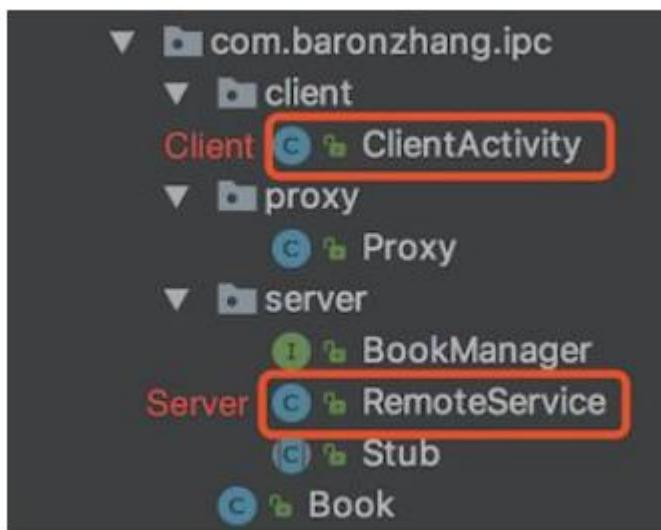
在正式编码实现跨进程调用之前，先介绍下实现过程中用到的一些类。

了解了这些类的职责，有助于我们更好的理解和实现跨进程通信。

- **IBinder** : IBinder 是一个接口，代表了一种跨进程通信的能力。
只要实现了这个借口，这个对象就能跨进程传输。
- **IInterface** : IInterface 代表的就是 Server 进程对象具备什么样的能力（能提供哪些方法，其实对应的就是 AIDL 文件中定义的接口）
- **Binder** : Java 层的 Binder 类，代表的其实就是 Binder 本地对象。BinderProxy 类是 Binder 类的一个内部类，它代表远程进程的 Binder 对象的本地代理；这两个类都继承自 IBinder, 因而都具有跨进程传输的能力；实际上，在跨越进程的时候，Binder 驱动会自动完成这两个对象的转换。
- **Stub** : AIDL 的时候，编译工具会给我们生成一个名为 Stub 的静态内部类；这个类继承了 Binder, 说明它是一个 Binder 本地对象，它实现了 IInterface 接口，表明它具有 Server 承诺给 Client 的能力；Stub 是一个抽象类，具体的 IInterface 的相关实现需要开发者自己实现。

6.2 实现过程讲解

一次跨进程通信必然会涉及到两个进程，在这个例子中
RemoteService 作为服务端进程，提供服务；ClientActivity 作为客
户端进程，使用 RemoteService 提供的服务。如下图：



那么服务端进程具备什么样的能力？能为客户端提供什么样的服务
呢？还记得我们前面介绍过的 IInterface 吗，它代表的就是服务端进
程具体什么样的能力。因此我们需要定义一个 BookManager 接口，
BookManager 继承自 IInterface，表明服务端具备什么样的能力。

```
/**  
 * 这个类用来定义服务端 RemoteService 具备什么样的能力  
 */  
public interface BookManager extends IInterface {  
  
    void addBook(Book book) throws RemoteException;  
}
```

只定义服务端具备什么样的能力是不够的，既然是跨进程调用，那么接
下来我们得实现一个跨进程调用对象 Stub。Stub 继承 Binder，说明

它是一个 Binder 本地对象；实现 IInterface 接口，表明具有 Server 承诺给 Client 的能力；Stub 是一个抽象类，具体的 IInterface 的相关实现需要调用方自己实现。

```
public abstract class Stub extends Binder implements BookManager {  
  
    ...  
  
    public static BookManager asInterface(IBinder binder) {  
        if (binder == null)  
            return null;  
        IInterface iin = binder.queryLocalInterface(DESCRIPTOR);  
        if (iin != null && iin instanceof BookManager)  
            return (BookManager) iin;  
        return new Proxy(binder);  
    }  
  
    ...  
  
    @Override  
    protected boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws  
RemoteException {  
    switch (code) {  
  
        case INTERFACE_TRANSACTION:  
            reply.writeString(DESCRIPTOR);  
            return true;  
  
        case TRANSACTION_addBook:  
            data.enforceInterface(DESCRIPTOR);  
            Book arg0 = null;  
            if (data.readInt() != 0) {  
                arg0 = Book.CREATOR.createFromParcel(data);  
            }  
            this.addBook(arg0);  
            reply.writeNoException();  
            return true;  
  
    }  
    return super.onTransact(code, data, reply, flags);  
}
```

...

Stub 类中我们重点介绍下 `asInterface` 和 `onTransact`。

先说说 `asInterface`，当 Client 端在创建和服务端的连接，调用 `bindService` 时需要创建一个 `ServiceConnection` 对象作为入参。在 `ServiceConnection` 的回调方法 `onServiceConnected` 中 会通过这个 `asInterface(IBinder binder)` 拿到 `BookManager` 对象，这个 `IBinder` 类型的入参 `binder` 是驱动传给我们的，正如你在代码中看到的一样，方法中会去调用 `binder.queryLocalInterface()` 去查找 `Binder` 本地对象，如果找到了就说明 Client 和 Server 在同一进程，那么这个 `binder` 本身就是 `Binder` 本地对象，可以直接使用。否则说明是 `binder` 是个远程对象，也就是 `BinderProxy`。因此需要我们创建一个代理对象 `Proxy`，通过这个代理对象来实现远程访问。

接下来我们就要实现这个代理类 `Proxy` 了，既然是代理类自然需要实现 `BookManager` 接口。

```
public class Proxy implements BookManager {  
    ...  
  
    public Proxy(IBinder remote) {  
        this.remote = remote;  
    }  
  
    @Override  
    public void addBook(Book book) throws RemoteException {  
  
        Parcel data = Parcel.obtain();  
        Parcel replay = Parcel.obtain();  
        try {  
            data.writeInterfaceToken(DESCRIPTOR);  
            if (book != null) {  
                data.writeInt(1);  
                book.writeToParcel(data, 0);  
            } else {  
                data.writeInt(0);  
            }  
            remote.transact(Stub.TRANSAVTION_addBook, data, replay, 0);  
            replay.readException();  
        } finally {  
            replay.recycle();  
            data.recycle();  
        }  
    }  
    ...  
}
```

我们看看 addBook() 的实现 ;在 Stub 类中 ,addBook(Book book) 是一个抽象方法 , Server 端需要去实现它。

- 如果 Client 和 Server 在同一个进程 , 那么直接就是调用这个方法。
- 如果是远程调用 , Client 想要调用 Server 的方法就需要通过 Binder 代理来完成 , 也就是上面的 Proxy。

在 Proxy 中的 addBook() 方法中首先通过 Parcel 将数据序列化，然后调用 remote.transact()。正如前文所述 Proxy 是在 Stub 的 asInterface 中创建，能走到创建 Proxy 这一步就说明 Proxy 构造函数的入参是 BinderProxy，即这里的 remote 是个 BinderProxy 对象。最终通过一系列的函数调用，Client 进程通过系统调用陷入内核态，Client 进程中执行 addBook() 的线程挂起等待返回 驱动完成一系列的操作之后唤醒 Server 进程，调用 Server 进程本地对象的 onTransact()。最终又走到了 Stub 中的 onTransact() 中，onTransact() 根据函数编号调用相关函数（在 Stub 类中为 BookManager 接口中的每个函数中定义了一个编号，只不过上面的源码中我们简化掉了；在跨进程调用的时候，不会传递函数而是传递编号来指明要调用哪个函数）；我们这个例子里面，调用了 Binder 本地对象的 addBook() 并将结果返回给驱动，驱动唤醒 Client 进程里刚刚挂起的线程并将结果返回。

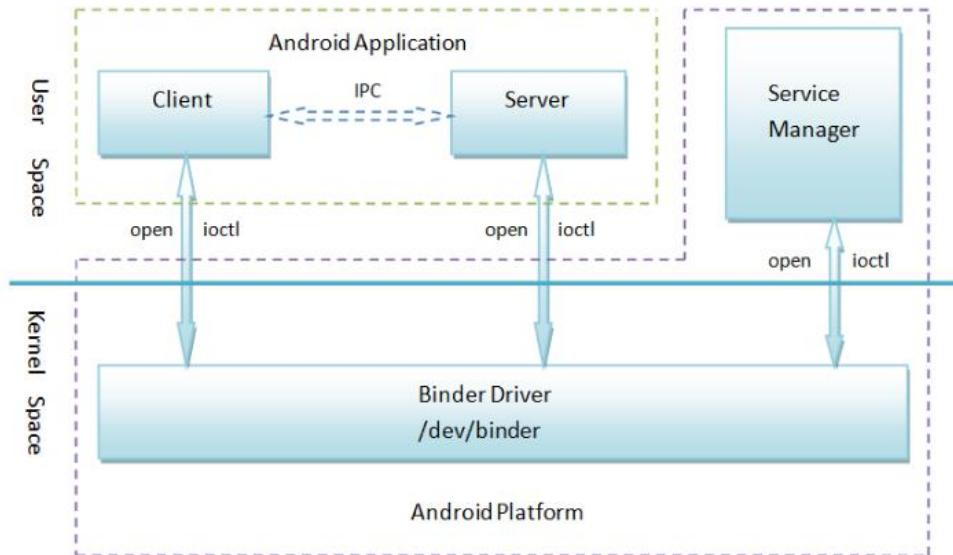
这样一次跨进程调用就完成了。

完整的代码我放到 GitHub 上了，有兴趣的小伙伴可以去看看。源码地址：github.com/BaronZ88/Hello-Binder

最后建议大家在不借助 AIDL 的情况下手写实现 Client 和 Server 进程的通信，加深对 Binder 通信过程的理解。

文章、Android 进程间通信（IPC）机制 Binder 简要介绍和学习计划

Android 深入浅出之 Binder 机制一文从情景出发，深入地介绍了 Binder 在用户空间的三个组件 Client、Server 和 Service Manager 的相互关系，Android Binder 设计与实现一文则是详细地介绍了内核空间的 Binder 驱动程序的数据结构和设计原理。非常感谢这两位作者给我们带来这么好的 Binder 学习资料。总结一下，Android 系统 Binder 机制中的四个组件 Client、Server、Service Manager 和 Binder 驱动程序的关系如下图所示：



1. Client、Server 和 Service Manager 实现在用户空间中，Binder 驱动程序实现在内核空间中

2. Binder 驱动程序和 Service Manager 在 Android 平台中已经实现，开发者只需要在用户空间实现自己的 Client 和 Server
 3. Binder 驱动程序提供设备文件/dev/binder 与用户空间交互，Client、Server 和 Service Manager 通过 open 和 ioctl 文件操作函数与 Binder 驱动程序进行通信
 4. Client 和 Server 之间的进程间通信通过 Binder 驱动程序间接实现
 5. Service Manager 是一个守护进程，用来管理 Server，并向 Client 提供查询 Server 接口的能力
- 至此，对 Binder 机制总算是有了一个感性的认识，但仍然感到不能很好地从上到下贯穿整个 IPC 通信过程，于是，打算通过下面四个情景来分析 Binder 源代码，以进一步理解 Binder 机制：
1. Service Manager 是如何成为一个守护进程的？即 Service Manager 是如何告知 Binder 驱动程序它是 Binder 机制的上下文管理者。
 2. Server 和 Client 是如何获得 Service Manager 接口的？即 defaultServiceManager 接口是如何实现的。
 3. Server 是如何把自己的服务启动起来的？Service Manager 在 Server 启动的过程中是如何为 Server 提供服务的？即 IServiceManager::addService 接口是如何实现的。

4 Service Manager 是如何为 Client 提供服务的？即
IServiceManager::getService 接口是如何实现的。

在接下来的四篇文章中，将按照这四个情景来分析 Binder 源代码，都将会涉及到用户空间到内核空间的 Binder 相关源代码。这里为什么没有 Client 和 Server 是如何进行进程间通信的情景呢？这是因为 Service Manager 在作为守护进程的同时，它也充当 Server 角色。因此，只要我们能够理解第三和第四个情景，也就理解了 Binder 机制中 Client 和 Server 是如何通过 Binder 驱动程序进行进程间通信的了。

为了方便描述 Android 系统进程间通信 Binder 机制的原理和实现，在接下来的四篇文章中，我们都是基于 C/C++ 语言来介绍 Binder 机制的实现的，但是，我们在 Android 系统开发应用程序时，都是基于 Java 语言的，因此，我们会在最后一篇文章中，详细介绍 Android 系统进程间通信 Binder 机制在应用程序框架层的 Java 接口实现：

5、Android 系统进程间通信 Binder 机制在应用程序框架层的 Java 接口源代码分析。

文章、Android 进程间通信

什么是进程

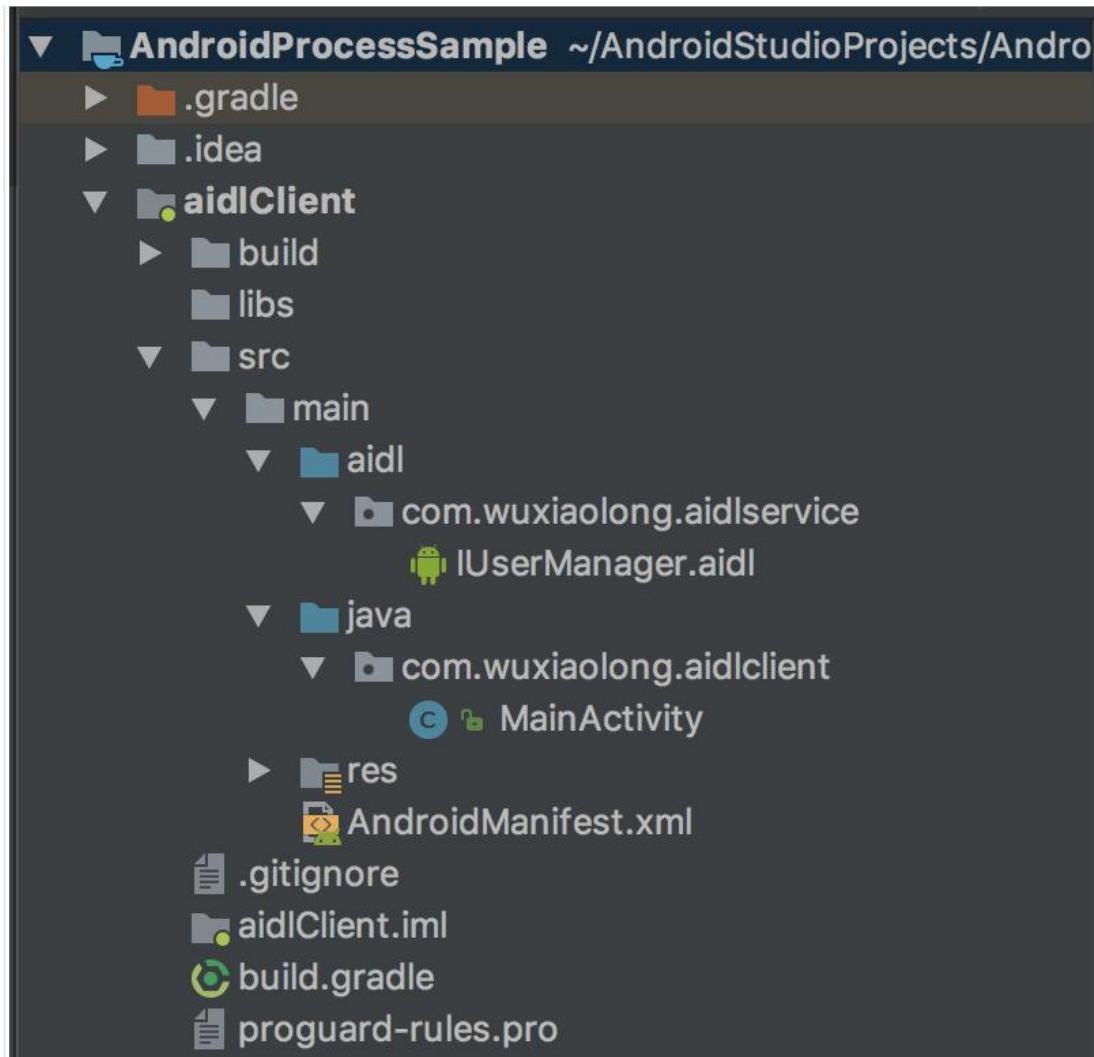
按照操作系统中的描述：进程一般指一个执行单元，在 PC 和移动设备上指一个程序或者一个应用。

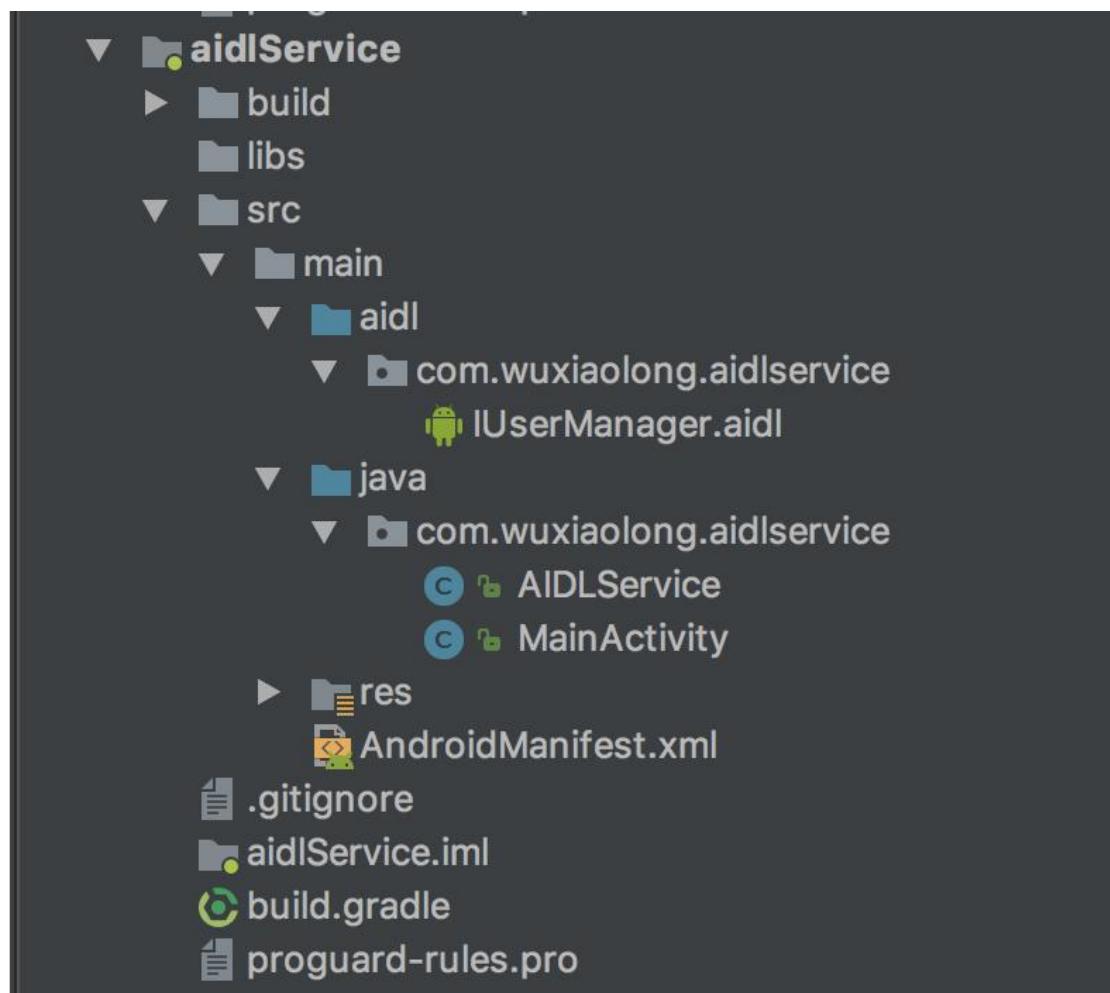
为什么要使用多进程

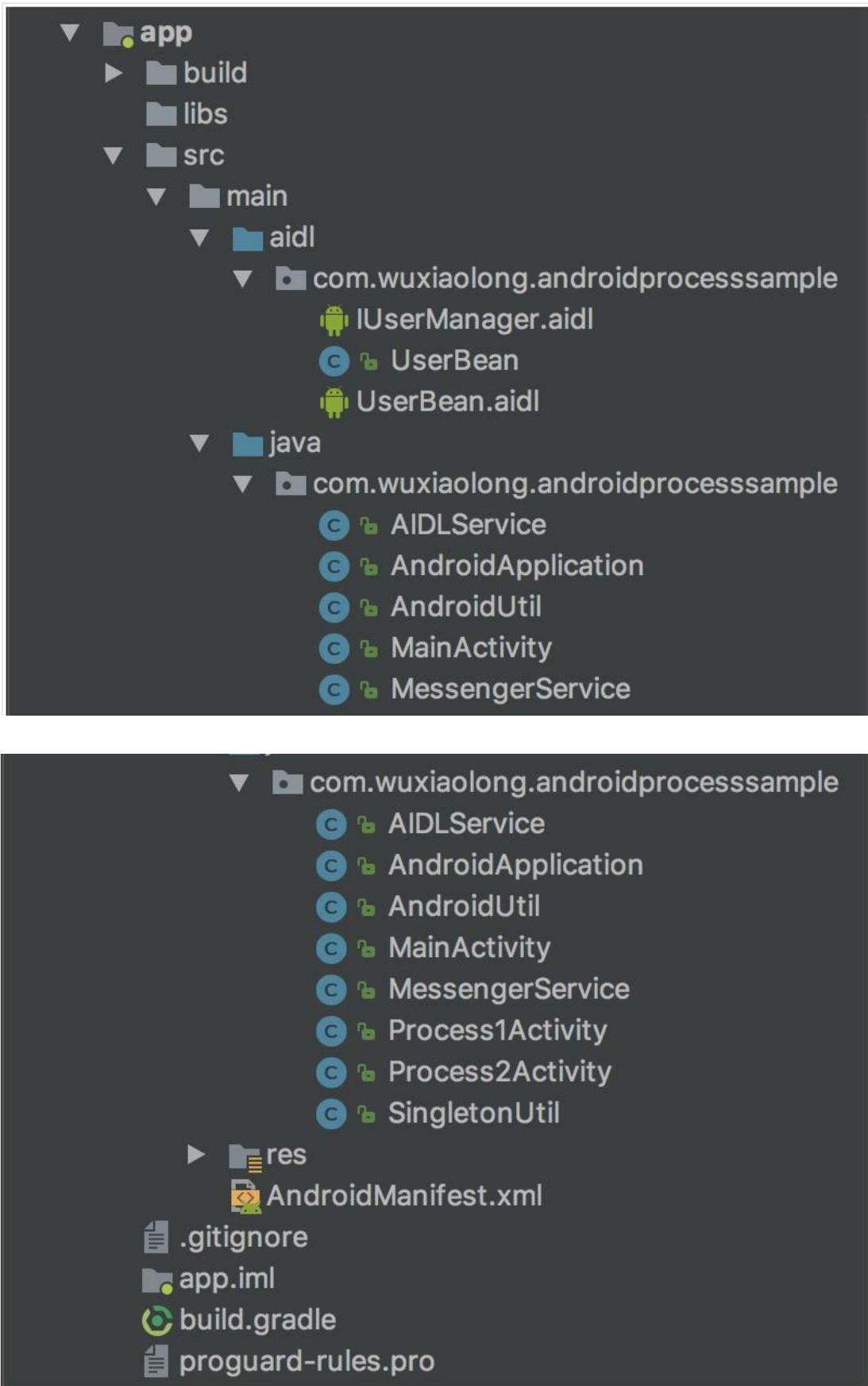
我们都知道，系统为 APP 每个进程分配的内存是有限的，如果想获取更多内存分配，可以使用多进程，将一些看不见的服务、比较独立而又相当占用内存的功能运行在另外一个进程当中。

目录结构预览

先放出最终实践后的目录结构，有个大概印象，后面一一介绍。



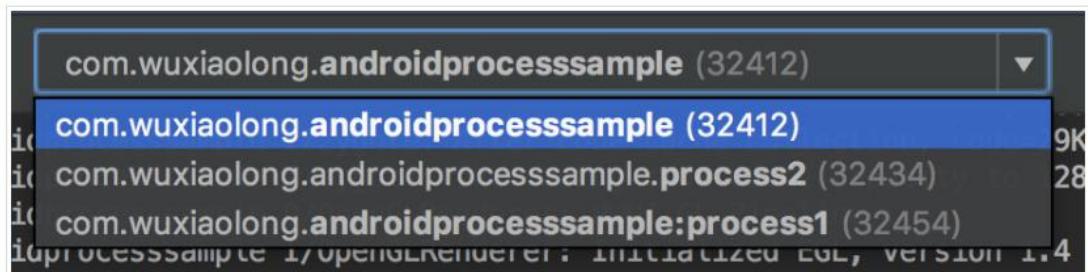




如何使用多进程

AndroidManifest.xml 清单文件中注册 Activity、Service 等四大组件时，指定 android:process 属性即可开启多进程，如：

```
1 <activity
2     android:name=".Process1Activity"
3     android:process=":process1" />
4 <activity
5     android:name=".Process2Activity"
6     android:process="com.wuxiaolong.androidprocesssample.process2" />
```



说明：

- 1、`com.wuxiaolong.androidprocesssample`，主进程，默认的是应用包名；
- 2、`android:process=":process1"`，“`:`”开头，是简写，完整进程名包名 + `:process1`；
- 3、`android:process="com.wuxiaolong.androidprocesssample.process2"`，以小写字母开头的，属于全局进程，其他应用可以通过 ShareUID 进行数据共享；
- 4、进程命名跟包名的命名规范一样。

进程弊端

Application 多次创建

我们自定义一个 Application 类，`onCreate` 方法进行打印 `Log.d("wx1", "AndroidApplication onCreate");`，然后启动 Process1Activity：

```
1 com.wuxiaolong.androidprocesssample D/wxl: AndroidApplication onCreate  
2 com.wuxiaolong.androidprocesssample:process1 D/wxl: AndroidApplication onCreate
```

看到确实被创建两次，原因见：android:process 的坑，你懂吗？多数情况下，我们都会在工程中自定义一个 Application 类，做一些全局性的初始化工作，因为我们要区分出来，让其在主进程进行初始化，网上解决方案：

```
1 @Override  
2 public void onCreate() {  
3     super.onCreate();  
4     String processName = AndroidUtil.getProcessName();  
5     if (getPackageName().equals(processName)) {  
6         //初始化操作  
7         Log.d("wxl", "AndroidApplication onCreate=" + processName);  
8     }  
9 }
```

AndroidUtil：

```
1 public static String getProcessName() {  
2     try {  
3         File file = new File("/proc/" + android.os.Process.myPid() + "/" + "cmdline");  
4         BufferedReader mBufferedReader = new BufferedReader(new FileReader(file));  
5         String processName = mBufferedReader.readLine().trim();  
6         mBufferedReader.close();  
7         return processName;  
8     } catch (Exception e) {  
9         e.printStackTrace();  
10    return null;  
11 }  
12 }
```

静态成员和单例模式失效

创建一个类 SingletonUtil：

```
1 public class SingletonUtil {  
2     private static SingletonUtil singletonUtil;  
3     private String userId = "0";  
4  
5     public static SingletonUtil getInstance() {  
6         if (singletonUtil == null) {  
7             singletonUtil = new SingletonUtil();  
8         }  
9         return singletonUtil;  
10    }  
11  
12    public String getUserId() {  
13        return userId;  
14    }  
15  
16    public void setUserId(String userId) {  
17        this.userId = userId;  
18    }  
19 }
```

在 MainActivity 进行设置：

```
1 SingletonUtil.getInstance().setUserId("007");
```

Process1Activity 取值，打印：

```
1 Log.d("wxl", "userId=" + SingletonUtil.getInstance().getUserId());
```

发现打印 `userId=0`，单例模式失效了，因为这两个进程不在同一内存了，自然无法共享。

进程间通信

文件共享

既然内存不能共享，是不是可以找个共同地方，是的，可以把要共享的数据保存 SD 卡，实现共享。首先将 SingletonUtil 实现 Serializable 序列化，将对象存入 SD 卡，然后需要用的地方，反序列化，从 SD 卡取出对象，完整代码如下：

SingletonUtil

```
public class SingletonUtil implements Serializable{  
    public static String ROOT_FILE_DIR =  
        Environment.getExternalStorageDirectory() + File.separator + "User" +  
        File.separator;  
    public static String USER_STATE_FILE_NAME_DIR = "UserState";  
    private static SingletonUtil singletonUtil;  
    private String userId = "0";  
    public static SingletonUtil getInstance() {
```

```

if (singletonUtil == null) {
    singletonUtil = new SingletonUtil();
}
return singletonUtil;
}
public String getUserId() {
    return userId;
}
public void setId(String userId) {
    this.userId = userId;
}
}

```

序列化和反序列化

```

public class AndroidUtil {
    public static boolean createOrExistsDir(final File file) {
        // 如果存在，是目录则返回 true，是文件则返回 false，不存在则返回是否创建成功
        return file != null && (file.exists() ? file.isDirectory() : file.mkdirs());
    }
    /**
     * 删除目录
     *
     * @param dir 目录
     * @return {@code true}: 删除成功<br>{@code false}: 删除失败
     */
    public static boolean deleteDir(final File dir) {
        if (dir == null) return false;
        // 目录不存在返回 true
        if (!dir.exists()) return true;
        // 不是目录返回 false
        if (!dir.isDirectory()) return false;
        // 现在文件存在且是文件夹
        File[] files = dir.listFiles();
        if (files != null && files.length != 0) {
            for (File file : files) {
                if (file.isFile()) {
                    if (!file.delete()) return false;
                } else if (file.isDirectory()) {
                    if (!deleteDir(file)) return false;
                }
            }
        }
        return dir.delete();
    }
}

```

```
}

/**
 * 序列化，对象存入 SD 卡
 *
 * @param obj 存储对象
 * @param destFileDir SD 卡目标路径
 * @param destFileName SD 卡文件名
 */
public static void writeObjectToSDCard(Object obj, String destFileDir,
String destFileName) {
createOrExistsDir(new File(destFileDir));
deleteDir(new File(destFileDir + destFileName));
FileOutputStream fileOutputStream = null;
ObjectOutputStream objectOutputStream = null;
try {
fileOutputStream = new FileOutputStream(new File(destFileDir,
destFileName));
objectOutputStream = new ObjectOutputStream(fileOutputStream);
objectOutputStream.writeObject(obj);
} catch (Exception e) {
e.printStackTrace();
} finally {
try {
if (objectOutputStream != null) {
objectOutputStream.close();
objectOutputStream = null;
}
if (fileOutputStream != null) {
fileOutputStream.close();
fileOutputStream = null;
}
} catch (IOException e) {
e.printStackTrace();
}
}
}

/**
 * 反序列化，从 SD 卡取出对象
 *
 * @param destFileDir SD 卡目标路径
 * @param destFileName SD 卡文件名
 */
public static Object readObjectFromSDCard(String destFileDir, String
destFileName) {
```

```
FileInputStream fileInputStream = null;
Object object = null;
ObjectInputStream objectInputStream = null;
try {
    fileInputStream = new FileInputStream(new File(destFileDir,
destFileName));
    objectInputStream = new ObjectInputStream(fileInputStream);
    object = objectInputStream.readObject();
} catch (Exception e) {
    e.printStackTrace();
} finally {
try {
    if (objectInputStream != null) {
        objectInputStream.close();
        objectInputStream = null;
    }
    if (fileInputStream != null) {
        fileInputStream.close();
        fileInputStream = null;
    }
} catch (IOException e) {
    e.printStackTrace();
}
}
return object;
}
}
```

十二、Android 高级必备：

AMS,WMS,PMS

1、**AMS,WMS,PMS 全解析**

开机 SystemServer 到 ActivityManagerService 启动过程

一 从 Systemserver 到 AMS

zygote-> systemserver: java 入层口:

```
/**
```

```
* The main entry point from zygote.  
*/  
public static void main(String[] args) {  
    new SystemServer().run();  
}  
接下来继续看 SystemServer run 函数执行过程:  
private void run() {  
  
    // 准备 SystemServer 运行环境: 设置线程优先级, 创建主线层 Looper,  
    ActivityThread 和 SystemContext  
    android.os.Process.setThreadPriority();  
    Looper.prepareMainLooper();  
  
    // 创建 systemserver 上进程的 ActivityThread 和 SystemContext  
    createSystemContext();  
  
    // 增加SystemServiceManager: 统一管理 system services 的创建, 启动和生命  
    周期, 多用户切换  
    mSystemServiceManager = new  
    SystemServiceManager(mSystemContext);  
  
    // Start services.  
  
    // 1. 创建 AMS  
    mActivityManagerService = mSystemServiceManager.startService(  
        ActivityManagerService.Lifecycle.class).getService();  
  
    // Start the Power Manager service  
    mPowerManagerService =  
    mSystemServiceManager.startService(PowerManagerService.class);  
  
    // Start the package manager service  
    mPackageManagerService = PackageManagerService.main();  
  
    // 2. 将 SystemServer 进程可加到 AMS 中调度管理  
    mActivityManagerService.setSystemProcess();  
  
    // 3. 将相关 provider 运行在 systemserver 进程中: SettingsProvider  
    mActivityManagerService.installSystemProviders();  
  
    //
```

```

final Watchdog watchdog = Watchdog.getInstance();
watchdog.init(context, mActivityManagerService);

// Start Window Manager
wm = WindowManagerService.main();

// 4. 直接保存 wms 对象，与 WMS 交互
mActivityManagerService.setWindowManager(wm);

// 5. 通过 WMS 弹出“正在启动应用”框
// R.string.android_upgrading_starting_apps
ActivityManagerNative.getDefault().showBootMessage();

// 6. AMS 作为 Framework 核心，做好准备就绪后就开始启动应用层，和对 AMS 有
依赖的服务
mActivityManagerService.systemReady(new Runnable(){
    // 启动 SystemUI
    startSystemUi(context);

    // 启动 WatchDog 监控核心服务状态
    Watchdog.getInstance().start();

    //
    mmsServiceF.systemRunning();
});

// Loop forever.
Looper.loop();
}

```

以上 6 个步骤是 `SystemServer` 中关于 **AMS** 的调用，完成 **AMS** 的创建和系统的初始化，下面按照这步骤继续升入分析。

- 这里有个疑问：

AMS 保存对象，本身就在同一个进程，**WMS** 与 **WMS** 之间的交互式直接调用速度会更快，其他服务为何不这样，是因为耦合太强，还是实时性要求更高？

弹出“正在启动应用”框，这里为何不直接调用 **AMS** 的 `showBootMessage` 而是通过 `binder` 方式调用，其他接口都是直接调用，为何？直接调用有何不可吗？

二 ActivityManagerService 创建过程

接上面 `SystemServer.run` 中：

```
mActivityManagerService = mSystemServiceManager.startService(
```

```
ActivityManagerService.Lifecycle.class).getService();
```

这是通过 `SystemServiceManager` 这样一个模板类来创建运行在 `SystemServer` 中的 `Framework` 服务；

并将创建的服务统一保存在队列管理，会涉及到多用户切换。

```
// Note: This method is invoked on the main thread but may need to attach various  
// handlers to other threads. So take care to be explicit about the looper.  
public ActivityManagerService(Context systemContext) {  
    // 1. 系统 Context 和 ActivityThread (将 systemserver 进程作为  
    // 应用进程管理)  
    mContext = systemContext;  
    mFactoryTest = FactoryTest.getMode();  
    mSystemThread = ActivityThread.currentActivityThread();  
  
    // 2. AMS 工作的线程和 Handler，处理显示相关的 UiHandler ---》 知识点  
    // HandlerThread 和 Handler  
    mHandlerThread = new ServiceThread(TAG,  
        android.os.Process.THREAD_PRIORITY_FOREGROUND, false  
    /*allowIo*/);  
    mHandlerThread.start();  
    mHandler = new MainHandler(mHandlerThread.getLooper());  
    mUiHandler = new UiHandler();  
  
    // 3. 广播队列 BroadcastQueue 初始化：前台广播队列和后台广播队列  
    mFgBroadcastQueue = new BroadcastQueue(this,  
    mHandler, "foreground", BROADCAST_FG_TIMEOUT, false);  
    mBgBroadcastQueue = new BroadcastQueue(this,  
    mHandler, "background", BROADCAST_BG_TIMEOUT, true);  
    mBroadcastQueues[0] = mFgBroadcastQueue;  
    mBroadcastQueues[1] = mBgBroadcastQueue;  
  
    // 4. Service 和 Provider 管理  
    mServices = new ActiveServices(this);  
    mProviderMap = new ProviderMap(this);  
  
    // 5. 系统数据存放目录：/data/system/  
    File dataDir = Environment.getDataDirectory();  
    File systemDir = new File(dataDir, "system");  
    systemDir.mkdirs();  
  
    // 电池状态信息，进程状态 和 应用权限管理  
    mBatteryStatsService = new BatteryStatsService(systemDir,
```

```

mHandler);
        mProcessStats = new ProcessStatsService(this, new File(systemDir,
"procstats"));
        mAppOpsService = new AppOpsService(new File(systemDir,
"appops.xml"), mHandler);

        // 6.多用户管理
        mStartedUsers.put(UserHandle.USER_OWNER, new
UserState(UserHandle.OWNER, true));
        mUserLru.add(UserHandle.USER_OWNER);
        updateStartedUserArrayLocked();

        // 7.最近任务, Activity, Task 管理
        mRecentTasks = new RecentTasks(this);
        mStackSupervisor = new ActivityStackSupervisor(this,
mRecentTasks);
        mTaskPersister = new TaskPersister(systemDir, mStackSupervisor,
mRecentTasks);

        // 创建一个新线程, 用于监控和定时更新系统 CPU 信息, 30 分钟更新一次 CPU
        和电池信息
        mProcessCpuTracker.init();
        mProcessCpuThread = new Thread("CpuTracker") {}

        // 加入 Watchdog 监控起来
        Watchdog.getInstance().addMonitor(this);
        Watchdog.getInstance().addThread(mHandler);
    }

```

- 以上 **AMS** 创建过程 涉及到 **Android** 四大组件管理的初始化:

```

Broadcast --> BroadcastQueue
Provider --> ProviderMap
Service --> ActiveServices
Activity --> ActivityStackSupervisor

```

备注 1: **Android6.0** 上加入多用户功能, 增加了一些涉及多用户的管理。

拓展知识点: HandlerThread, Handle, Looper

三 将 SystemServer 进程可加到 AMS 中调度管理

接上面 systemserver.run 中:

```
mActivityManagerService.setSystemProcess();
```

```

public void setSystemProcess() {
    // 将服务加入到 ServiceManager 中
    ServiceManager.addService(Context.ACTIVITY_SERVICE,      this,
true);
    ServiceManager.addService(ProcessStats.SERVICE_NAME,
mProcessStats);
    ServiceManager.addService("meminfo", new MemBinder(this));
    ServiceManager.addService("gfxinfo", new GraphicsBinder(this));
    ServiceManager.addService("dbinfo", new DbBinder(this));

    // 设置 application info LoadedApkinfo 有关 framework-res.apk
    ApplicationInfo           info          =
mContext.getPackageManager().getApplicationInfo(
        "android", STOCK_PM_FLAGS);
    mSystemThread.installSystemApplicationInfo(info,
getClass().getClassLoader());

    // 给 SystemServer 进程创建 ProcessRecord , adj 值 , 就是将
SystemServer 进程加入到 AMS 进程管理机制中，跟应用进程一致
    synchronized (this) {
        ProcessRecord   app     =   newProcessRecordLocked(info,
info.processName, false, 0);
        app.persistent = true;
        app.pid = MY_PID;
        app.maxAdj = ProcessList.SYSTEM_ADJ;
        app.makeActive(mSystemThread.getApplicationThread(),
mProcessStats);
        synchronized (mPidSelfLocked) {
            mPidSelfLocked.put(app.pid, app);
        }
        updateLruProcessLocked(app, false, null);
        updateOomAdjLocked();
    }
}

```

- `setSystemProcess` 意义：

这一步就是给 `SystemServer` 进程创建 `ProcessRecord`, `adj` 值, 就是将 `SystemServer` 进程加入到 AMS 进程管理机制中, 跟应用进程一致;

进程调度更新优先级 `oomadj` 值, 个人感觉 `SystemServer` 进程跟应用进程就不一样, 却加入 AMS 来调度管理, 这样做的意义何在?

四 创建运行在 `SystemServer` 进程中 Provider

接上面 `SystemServer.run` 中:

```
mActivityManagerService.installSystemProviders();
```

备注 2： 将相关 provider 运行在 systemserver 进程中： SettingsProvider
具体安装过程这里暂不详述，在应用启动过程中具体分析。

五 AMS systemReady 过程

接上面 SystemServer.run 中：

```
mActivityManagerService.systemReady();  
  
public void systemReady(final Runnable goingCallback) {  
    synchronized(this) {  
        if (mSystemReady) {  
            goingCallback.run();  
        }  
        .....  
  
        // 1.升级相关处理：发送 PRE_BOOT_COMPLETED 广播 等待升级处  
理完成才能继续  
        // Check to see if there are any update receivers to run.  
        if (!mDidUpdate) {  
            // 等待升级完成，否则直接返回  
            if (mWaitingUpdate) {  
                return;  
            }  
            // 发送 PRE_BOOT_COMPLETED 广播  
            final ArrayList<ComponentName> doneReceivers = new  
ArrayList<ComponentName>();  
            mWaitingUpdate      =      deliverPreBootCompleted(new  
Runnable() {  
            // 等待所有接收 PRE_BOOT_COMPLETED 广播者处理完毕  
            public void run() {  
                synchronized (ActivityManagerService.this) {  
                    mDidUpdate = true;  
                }  
                showBootMessage(mContext.getText(  
                    R.string.android_upgrading_complete),  
                    false);  
  
                // 将系统版本号和处理过的广播写入文件：  
                // /data/system/called_pre_boots.dat 文件  
                writeLastDonePreBootReceivers(doneReceivers);  
  
                // 继续 systemReady 流程  
                systemReady(goingCallback);  
            }  
        }, doneReceivers, UserHandle.USER_OWNER);
```

```

        if (mWaitingUpdate) {
            return;
        }
        mDidUpdate = true;
    }

    mSystemReady = true;
}

// 2. 收集已经启动的进程并杀死，除过 persistent 常驻进程
ArrayList<ProcessRecord> procsToKill = null;
synchronized(mPidsSelfLocked) {
    for (int i=mPidsSelfLocked.size()-1; i>=0; i--) {
        ProcessRecord proc = mpidsSelfLocked.valueAt(i);
        if (!isAllowedWhileBooting(proc.info)){
            if (procsToKill == null) {
                procsToKill = new ArrayList<ProcessRecord>();
            }
            procsToKill.add(proc);
        }
    }
}

synchronized(this) {
    if (procsToKill != null) {
        for (int i=procsToKill.size()-1; i>=0; i--) {
            ProcessRecord proc = procsToKill.get(i);
            Slog.i(TAG, "Removing system update proc: " + proc);
            removeProcessLocked(proc, true, false, "system
update done");
        }
    }
}

// Now that we have cleaned up any update processes, we
// are ready to start launching real processes and know that
// we won't trample on them any more.
mProcessesReady = true;
}

// 3.系统准备好后回调传入的 Runnable:
if (goingCallback != null) goingCallback.run();

// 4. 发送账户启动的广播，涉及多用户          long ident =

```

```

        Binder.clearCallingIdentity();
        Intent intent = new Intent(Intent.ACTION_USER_STARTED);
        broadcastIntentLocked(intent);                                intent = new
        Intent(Intent.ACTION_USER_STARTING);
        broadcastIntentLocked(intent);
        Binder.restoreCallingIdentity(ident);
        // 5. 启动桌面 Home Activity
        mBooting = true;
        startHomeActivityLocked(mCurrentUserId, "systemReady");
        mStackSupervisor.resumeTopActivitiesLocked();
    }

```

- 下面看下 AMS systemReady 的过程:

1. 升级相关处理: 发送 PRE_BOOT_COMPLETED 广播

顾名思义: 只有系统做 OTA 升级 和 手机初次开机的时候, 应当才会走此广播, 下面看看这个函数具体的处理。

接上面:

```

deliverPreBootCompleted(new Runnable() {
    // 向 PMS 查询, 所有接收 ACTION_PRE_BOOT_COMPLETED 广播的
    Receiver
    Intent intent = new
    Intent(Intent.ACTION_PRE_BOOT_COMPLETED);
    List<ResolveInfo> ris = null;
    ris = AppGlobals.getPackageManager().queryIntentReceivers(
        intent, null, 0, userId);

    // 只有系统广播才能接收该广播, 去掉非系统应用
    for (int i=ris.size()-1; i>=0; i--) {
        if ((ris.get(i).activityInfo.applicationInfo.flags
            &ApplicationInfo.FLAG_SYSTEM) == 0) {
            ris.remove(i);
        }
    }

    // 给 Intent 设置 flag: FLAG_RECEIVER_BOOT_UPGRADE, 很关键这
    // 个看看 flag 的作用:
    // 只有设置这个标志, 才能让应用在系统没有 ready 的情况下启动, 见下文
    // 原始注释
    intent.addFlags(Intent.FLAG_RECEIVER_BOOT_UPGRADE);

    // 将已经处理过 ACTION_PRE_BOOT_COMPLETED 广播的 Receiver 去
    // 掉
    // 已经处理该广播的 Receiver 记录 和 对应的系统版本号 都记录在:
}

```

```
/data/system/called_pre_boots.dat 文件中,
        // 通过与系统当前版本号比对，确认是否已处理过。考虑处理过程异常中断
        的情况：比如断电
        ArrayList<ComponentName>           lastDoneReceivers      =
readLastDonePreBootReceivers();

        // 将已经处理过的广播去除，同时记录已处理过保存在 doneReceivers 数
组中
        for (int i=0; i<ris.size(); i++) {
            ActivityInfo ai = ris.get(i).activityInfo;
            ComponentName          comp          =           new
ComponentName(ai.packageName, ai.name);
            if (lastDoneReceivers.contains(comp)) {
                // We already did the pre boot receiver for this app with the
current
                // platform version, so don't do it again...
                ris.remove(i);
                i--;
                // ...however, do keep it as one that has been done, so we
don't
                // forget about it when rewriting the file of last done
receivers.
                doneReceivers.add(comp);
            }
        }

        // 内部类专门用来 ACTION_PRE_BOOT_COMPLETED 广播的发送，要看看这个 PreBootContinuation 类
        // 这块逻辑一直在变，Android5.0, 6.0，以及看到在 7.0 上又变了，基本
思路不变，本文代码基于 Android6.0
        PreBootContinuation cont = new PreBootContinuation(intent,
onFinishCallback, doneReceivers,
                    ris, users);
        cont.go();
        return true;
    }

    给 intent 设置的广播意义：
    /**
     * Set when this broadcast is for a boot upgrade, a special mode that
     * allows the broadcast to be sent before the system is ready and
     launches
     * the app process with no providers running in it.
     * @hide
     */

```

```
public static final int FLAG_RECEIVER_BOOT_UPGRADE = 0x02000000;
```

- 继续接着上面 PreBootContinuation 类：从继承关系看到可以跨进程的

```
final class PreBootContinuation extends IIntentReceiver.Stub {  
    void go() {  
        // 判断是不是最后一个接收者  
        if (lastRi != curRi) {  
  
            // 疑问：如果不是最后一个接收者，则发给一个指定接收者  
            ComponentName  
            // 为什么要在这里指定接收者，一个个发送，而不是交给广播自己  
            // 去处理？  
            ActivityInfo ai = ris.get(curRi).activityInfo;  
            ComponentName comp = new  
            ComponentName(ai.packageName, ai.name);  
            intent.setComponent(comp);  
            doneReceivers.add(comp);  
            lastRi = curRi;  
  
            // 界面显示正在处理的广播，上面的指定接收者，就是为了这里能  
            // 显示正在处理的广播名称？  
            CharSequence label =  
            ai.loadLabel(mContext.getPackageManager());  
  
            showBootMessage(mContext.getString(R.string.android_preparing_apk, label),  
            false);  
        }  
  
        // 发送广播，指定接收者处理完毕，会 resultTo 回来--> this  
        Slog.i(TAG, "Pre-boot of " +  
        intent.getComponent().toShortString()  
        + " for user " + users[curUser]);  
        broadcastIntentLocked(null, null, intent, null, this,  
        0, null, null, null, AppOpsManager.OP_NONE,  
        null, true, false, MY_PID, Process.SYSTEM_UID,  
        users[curUser]);  
    }  
  
    public void performReceive(Intent intent, int resultCode,  
    String data, Bundle extras, boolean ordered,  
    boolean sticky, int sendingUser) {
```

```

        // 指定接收者广播处理完毕回调 resultTo 回来，继续处理下一个，如果
所有处理完，则 post 消息执行 onFinishCallback
        curUser++;
        if (curUser >= users.length) {
            curUser = 0;
            curRi++;
            if (curRi >= ris.size()) {
                // All done sending broadcasts!
                if (onFinishCallback != null) {
                    // The raw IIntentReceiver interface is called
                    // with the AM lock held, so redispatch to
                    // execute our code without the lock.
                    mHandler.post(onFinishCallback);
                }
            }
            return;
        }
    }
    go();
}
}

```

备注 1:

在 Android L 版本上：是直接发送广播，通过 action: PRE_BOOT_COMPLETED，AMS 会去发给各个接收者。处理完毕回调 resultTo 回来；

在 Android M 版本上：这里就直接指定接收者，一个个发送出去，处理完毕回调 resultTo 回来，继续下一个。界面上可以看到在变化：显示正在处理的广播；

这样做的好处界面体验更好，木有看出有什么其他特别的用意。

备注 2:

系统都有哪些地方接收 PRE_BOOT_COMPLETED，以及什么情况下应该接收该广播？

从目前看到的主要应用在数据库应用升级方面，数据库升级涉及到数据字段变化，数据的增加等会比较耗时，

为了加快应用启动和提供数据，需要在开机过程中做升级操作，避免使用时耗时。

备注 3:

这里其实存在一个隐患：从上面的流程看到，系统发送广播给接收者处理，只有等所有接收者处理完毕，才会继续系统的启动流程。

试想：如果某个接收者的操作处理耗时较长，甚至被阻塞 或其他异常导致广播处理无法完成，不能回调回来怎么办？

结果：开机时间需要的更长了，或无法开机，一直就卡在这里无法开机。

很不幸，这种情况被我遇到过，大概是这样的：

某次 Hota 升级某应用 A 注册 PRE_BOOT_COMPLETED 广播，处理该广播时，由于某种情况需要访问应用 B 的数据库，等待应用 B 启动，

由于系统没有 ready 和应用 B 非 persistit 进程，系统不让启动 B，结果系统就被阻塞在这里，始终无法开机。

这其实是系统不合理的地方，没有相应的超时控制的安全机制，所幸这里只允许系统应用接收该广播，如果允许第三方接收，后果可想而知。

2. 收集已经启动的进程并杀死除过 persistent 进程

比如接收 PRE_BOOT_COMPLETED 启动的应用

到此系统准备完毕，可以开始启动应用进程，并置变量：mProcessesReady = true；

疑问：系统还没准备之前不允许启动非 persistent 进程，这之前的接收 PRE_BOOT_COMPLETED 广播的应用是如何启动的？

--》见应用启动部分分析。

3. 系统准备好后回调传入的 Runnable

启动应用和服务：{

```
startSystemUi(context);
connectivityF.systemReady();
.....
Watchdog.getInstance().start();
mmsServiceF.systemRunning();
}
```

4. 发送账户启动的广播，涉及多用户

多用户的问题这里不讨论。

注意发送该广播前有如下操作：成对出现

```
//操作前 clear
long ident = Binder.clearCallingIdentity();

//do 相关操作
.....
//操作后 restore
Binder.restoreCallingIdentity(ident);
```

通常这两者都是成对出现，具体的作用简单说下：这涉及到权限管理后面会讨论。

Binder.clearCallingIdentity():

通过 IPC binder 调用来远端进程，当前进程会记录调用者的 PID 和 UID，即通常使用的 getCallingPid 和 getCallingUid，

而会 **clearCallingIdentity** 把调用者 PID 和 UID 清除，将其设置为当前进程的 PID 和 UID，并将原来的 PID 和 UID 作为返回值；

PID 和 UID 是保存在一个 long 型数中，通过移位计算。

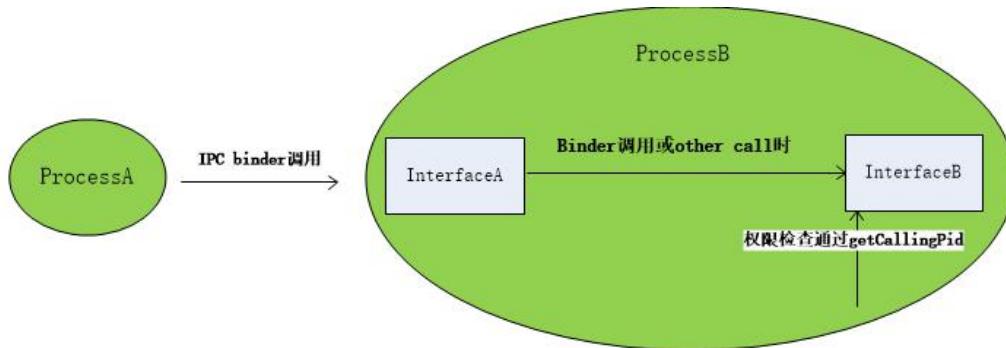
Binder.restoreCallingIdentity(ident): 恢复刚才清除的远端调用者的 PID 和 UID。

这样做有什么作用？

这涉及到权限管理，**clearCallingIdentity** 接口注释，举了 incoming call 例子，看下原注释：

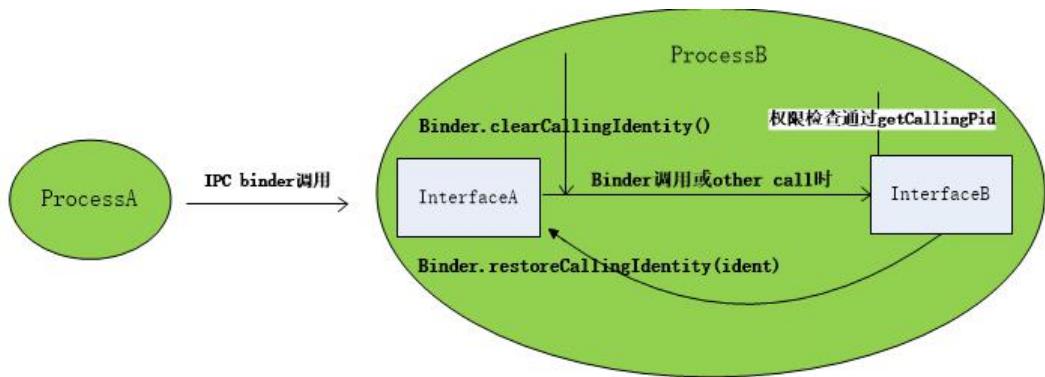
```
/**  
 * Reset the identity of the incoming IPC on the current thread. This can  
 * be useful if, while handling an incoming call, you will be calling  
 * on interfaces of other objects that may be local to your process and  
 * need to do permission checks on the calls coming into them (so they  
 * will check the permission of your own local process, and not whatever  
 * process originally called you).  
 *  
 * @return Returns an opaque token that can be used to restore the  
 * original calling identity by passing it to  
 * {@link #restoreCallingIdentity(long)}.  
 *  
 * @see #getCallingPid()  
 * @see #getCallingUid()  
 * @see #restoreCallingIdentity(long)  
 */
```

大概的意思可以理解成这样：



在 **ProcessB** 中，**InterfaceA** 调用 **InterfaceB** 时，**InterfaceB** 中要做权限检查，通过 **getCallingPid**，

这时拿到的 **PID** 是 **ProcessA** 的，权限不够肿么办。**ProcessB** 的权限是够可以的：就可以如下面



代码里面很多这样的例子，具体原因请自行体会，贴一段源代码看看

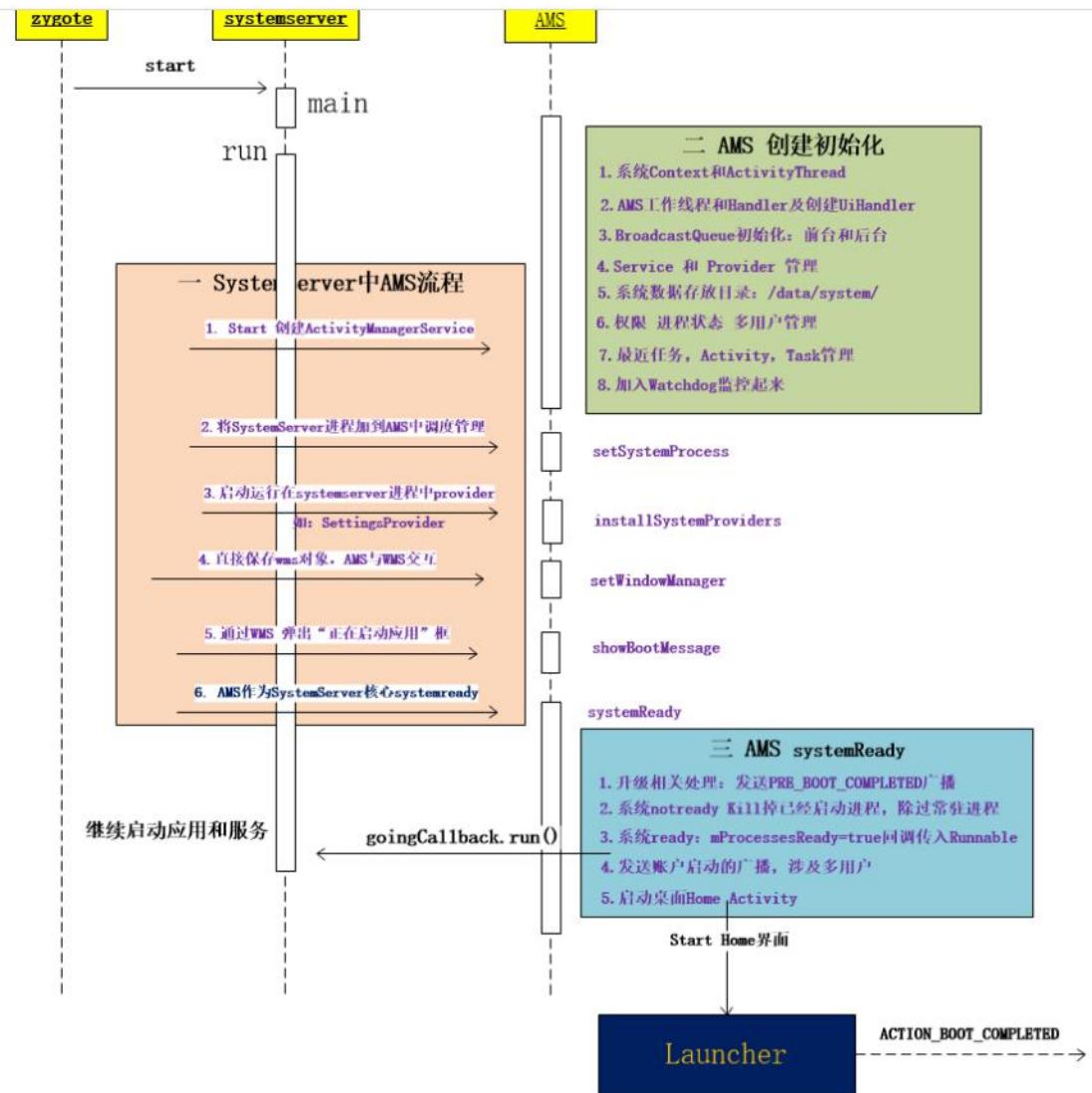
```
xref: /packages/services/Mms/src/com/android/mms/service/MmsService.java
Home | History | Annotate | Line# | Navigate | Download | Search | only in MmsService.java
238
239     @Override
240     public boolean deleteStoredMessage(String callingPkg, Uri messageUri)
241             throws RemoteException {
242         LogUtil.d("deleteStoredMessage " + messageUri);
243         enforceSystemUid();
244         if (!isSameMsgContentUci(messageUri)) {
245             LogUtil.e("deleteStoredMessage: invalid message URI: " + messageUri.toString());
246             return false;
247         }
248         // Clear the calling identity and query the database using the phone user id
249         // Otherwise the AppOps check in TelephonyProvider would complain about mismatch
250         // between the calling uid and the package uid
251         final long identity = Binder.clearCallingIdentity();
252         try {
253             if (getContentResolver().delete(
254                 messageUri, null/*where*/, null/*selectionArgs*/) != 1) {
255                 LogUtil.e("deleteStoredMessage: failed to delete");
256                 return false;
257             }
258         } catch (SQLException e) {
259             LogUtil.e("deleteStoredMessage: failed to delete", e);
260         } finally {
261             Binder.restoreCallingIdentity(identity);
262         }
263     }
}

```

5. 启动桌面 Home Activity

```
接上面systemReady最后部分：
// Start up initial activity.
mBooting = true;
startHomeActivityLocked(mCurrentUserId, "systemReady");
mStackSupervisor.resumeTopActivitiesLocked();
```

一张图说明 AMS 启动如上整个过程：



2、Android 解析 ActivityManagerService (一) AMS 启动流程和 AMS 家族

1. 概述

AMS 是系统的引导服务，应用进程的启动、切换和调度、四大组件的启动和管理都需要 AMS 的支持。从这里可以看出 AMS 的功能会十分的繁多，当然它并不是一个类承担这个重责，它有一些关联类，这在文章后面会讲到。AMS 的涉及的知识点非常多，这篇文章主要会讲解 AMS 的以下几个知识点：

AMS 的启动流程。

AMS 与进程启动。

AMS 家族。

2. AMS 的启动流程

AMS 的启动是在 SystemServer 进程中启动的，在 [Android 系统启动流程\(三\)](#) 解析 SystemServer 进程启动过程这篇文章中提及过，这里从 SystemServer 的 main 方法开始讲起：

frameworks/base/services/java/com/android/server/SystemServer.java

```
public static void main(String[] args) {  
    new SystemServer().run();  
}
```

main 方法中只调用了 SystemServer 的 run 方法，如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
private void run() {  
  
    ...  
  
    System.loadLibrary("android_servers");//1  
  
    ...  
  
    mSystemServiceManager = new  
        SystemServiceManager(mSystemContext);//2  
  
    LocalServices.addService(SystemServiceManager.class,  
        ...);  
}
```

```
mSystemServiceManager);  
  
...  
  
try {  
  
    Trace.traceBegin(Trace.TRACE_TAG_SYSTEM_SERVER,  
    "StartServices");  
  
    startBootstrapServices()//3  
  
    startCoreServices()//4  
  
    startOtherServices()//5  
  
} catch (Throwable ex) {  
  
    Slog.e("System",  
    "*****");  
  
    Slog.e("System", "***** Failure starting system  
services", ex);  
  
    throw ex;  
  
} finally {  
  
    Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);  
  
}  
  
...  
}
```

在注释 1 处加载了动态库 libandroid_servers.so。接下来在注释 2 处创建 SystemServiceManager，它会对系统的服务进行创建、启动和生命周期管理。在注释 3 中的 startBootstrapServices 方法中用 SystemServiceManager 启动

了 ActivityManagerService、PowerManagerService、PackageManagerService 等服务。在注释 4 处的 startCoreServices 方法中则启动了 BatteryService、UsageStatsService 和 WebViewUpdateService。注释 5 处的 startOtherServices 方法中启动了 CameraService、AlarmManagerService、VrManagerService 等服务。这些服务的父类均为 SystemService。从注释 3、4、5 的方法可以看出，官方把系统服务分为了三种类型，分别是引导服务、核心服务和其他服务，其中其他服务是一些非紧要和一些不需要立即启动的服务。系统服务总共大约有 80 多个，我们主要来查看引导服务 AMS 是如何启动的，注释 3 处的 startBootstrapServices 方法如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

ava

```
private void startBootstrapServices() {  
    Installer installer = null;  
    mSystemServiceManager.startService(Installer.class);  
    // Activity manager runs the show.  
    mActivityManagerService = null;  
    mSystemServiceManager.startService(  
        ActivityManagerService.Lifecycle.class).getService(); //1  
    mActivityManagerService.setSystemServiceManager(mSystemServiceManager);  
}
```

```
eManager);  
  
    mActivityManagerService.setInstaller(installer);  
  
    ...  
  
}
```

在注释 1 处调用了 SystemServiceManager 的 startService 方法，方法的参数是 ActivityManagerService.Lifecycle.class :

```
frameworks/base/services/core/java/com/android/server/SystemSe  
rviceManager.java  
  
@SuppressWarnings("unchecked")  
  
public <T extends SystemService> T startService(Class<T>  
serviceClass) {  
  
    try {  
  
        ...  
  
        final T service;  
  
        try {  
  
            Constructor<T> constructor  
            =  
serviceClass.getConstructor(Context.class);//1  
  
            service = constructor.newInstance(mContext);//2  
  
        } catch (InstantiationException ex) {  
  
            ...  
  
        }  
  
        // Register it.
```

```
mServices.add(service);//3

// Start it.

try {

    service.onStart();//4

} catch (RuntimeException ex) {

    throw new RuntimeException("Failed to start service

" + name

        + ": onStart threw an exception", ex);

}

return service;

} finally {

    Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);

}

}
```

startService 方法传入的参数是 Lifecycle.class，Lifecycle 继承自 SystemService。首先，通过反射来创建 Lifecycle 实例，在注释 1 处得到传进来的 Lifecycle 的构造器 constructor，在注释 2 处调用 constructor 的 newInstance 方法来创建 Lifecycle 类型的 service 对象。接着在注释 3 处将刚创建的 service 添加到 ArrayList 类型的 mServices 对象中来完成注册。最后在注释 4 处调用 service 的 onStart 方法来启动 service，并返回该 service。 Lifecycle 是 AMS 的内部类，代码如下所示。

[frameworks/base/services/core/java/com/android/server/am/Activi](#)

tyManagerService.java

```
public static final class Lifecycle extends SystemService {  
    private final ActivityManagerService mService;  
  
    public Lifecycle(Context context) {  
        super(context);  
  
        mService = new ActivityManagerService(context);//1  
    }  
  
    @Override  
    public void onStart() {  
        mService.start();//2  
    }  
  
    public ActivityManagerService getService() {  
        return mService;//3  
    }  
}
```

上面的代码结合 SystemServiceManager 的 startService 方法来分析，当通过反射来创建 Lifecycle 实例时，会调用注释 1 处的方法创建 AMS 实例，当调用 Lifecycle 类型的 service 的 onStart 方法时，实际上是调用了注释 2 处 AMS 的 start 方法。在 SystemServer 的 startBootstrapServices 方法的注释 1 处，调用了如下代码

```
mActivityManagerService = mSystemServiceManager.startService(  
    ActivityManagerService.Lifecycle.class).getService();
```

我们知道SystemServiceManager 的 startService 方法最终会返回 Lifecycle 类型的对象，紧接着又调用了 Lifecycle 的 getService 方法，这个方法会返回 AMS 类型的 mService 对象，见注释 3 处，这样 AMS 实例就会被创建并且返回。

3.AMS 与进程启动

在 [Android 系统启动流程（二）解析 Zygote 进程启动过程](#)这篇文章中，我提到了 Zygote 的 Java 框架层中，会创建一个 Server 端的 Socket，这个 Socket 用来等待 AMS 来请求 Zygote 来创建新的应用程序进程。要启动一个应用程序，首先要保证这个应用程序所需要的应用程序进程已经被启动。AMS 在启动应用程序时会检查这个应用程序所需要的应用程序进程是否存在，不存在就会请求 Zygote 进程将所需要的应用程序进程启动。Service 的启动过程中会调用 ActiveServices 的 bringUpServiceLocked 方法，如下所示。

frameworks/base/services/core/java/com/android/server/am/ActiveServices.java

```
private String bringUpServiceLocked(ServiceRecord r, int intentFlags,  
        boolean execInFg,  
        boolean whileRestarting, boolean  
        permissionsReviewRequired)  
        throws TransactionTooLargeException {  
  
    ...  
  
    final String procName = r.processName;//1
```

```
ProcessRecord app;

if (!isolated) {

    app      =      mAm.getProcessRecordLocked(procName,
r.appInfo.uid, false); //2

    if (DEBUG_MU) Slog.v(TAG_MU, "bringUpServiceLocked:
appInfo.uid=" + r.appInfo.uid

        + " app=" + app);

    if (app != null && app.thread != null) { //3

        try {

            app.addPackage(r.appInfo.packageName,
r.appInfo.versionCode,

            mAm.mProcessStats);

            realStartServiceLocked(r, app, execInFg); //4

            return null;
        }

        } catch (TransactionTooLargeException e) {

            ...

        }
    }

} else {

    app = r.isolatedProc;

}

if (app == null && !permissionsReviewRequired) { //5

    if ((app=mAm.startProcessLocked(procName, r.appInfo, true,
```

```
intentFlags,  
        "service", r.name, false, isolated, false)) == null) { //6  
  
        ...  
  
    }  
  
    if (isolated) {  
  
        r.isolatedProc = app;  
  
    }  
  
}  
  
...  
  
}
```

在注释 1 处得到 ServiceRecord 的 processName 的值赋值给 procName ，其中 ServiceRecord 用来描述 Service 的 android:process 属性。注释 2 处将 procName 和 Service 的 uid 传入到 AMS 的 getProcessRecordLocked 方法中，来查询是否存在一个与 Service 对应的 ProcessRecord 类型的对象 app ， ProcessRecord 主要用来记录运行的应用程序进程的信息。注释 5 处判断 Service 对应的 app 为 null 则说明用来运行 Service 的应用程序进程不存在，则调用注释 6 处的 AMS 的 startProcessLocked 方法来创建对应的应用程序进程，具体的过程请查看 [Android 应用程序进程启动过程（前篇）](#)。

4.AMS 家族

ActivityManager 是一个和 AMS 相关联的类，它主要对运行中的 Activity 进行管理，这些管理工作并不是由 ActivityManager 来处理的，而是交由 AMS 来处理，ActivityManager 中的方法会通过 ActivityManagerNative (以下简称 AMN) 的 getDefault 方法来得到 ActivityManagerProxy(以下简称 AMP)，通过 AMP 就可以和 AMN 进行通信，而 AMN 是一个抽象类，它会将功能交由它的子类 AMS 来处理，因此，AMP 就是 AMS 的代理类。AMS 作为系统核心服务，很多 API 是不会暴露给 ActivityManager 的，因此 ActivityManager 并不算是 AMS 家族一份子。

为了讲解 AMS 家族，这里拿 Activity 的启动过程举例，Activity 的启动过程中会调用 Instrumentation 的 execStartActivity 方法，如下所示。

frameworks/base/core/java/android/app/Instrumentation.java

```
public ActivityResult execStartActivity(  
    Context who, IBinder contextThread, IBinder token, Activity  
    target,  
    Intent intent, int requestCode, Bundle options) {  
  
    ...  
  
    try {  
        intent.migrateExtraStreamToClipData();  
        intent.prepareToLeaveProcess(who);  
        int result = ActivityManagerNative.getDefault()  
            .startActivity(whoThread, who.getPackageName(),  
            intent,
```

```
        intent.resolveTypeIfNeeded(who.getContentResolver()),  
        token, target != null ? target.mEmbeddedID :  
        null,  
        requestCode, 0, null, options);  
  
        checkStartActivityResult(result, intent);  
    } catch (RemoteException e) {  
        throw new RuntimeException("Failure from system", e);  
    }  
    return null;  
}
```

execStartActivity 方法中会调用 AMN 的 getDefault 来获取 AMS 的代理类 AMP。接着调用了 AMP 的 startActivity 方法，先来查看 AMN 的 getDefault 方法做了什么，如下所示。

frameworks/base/core/java/android/app/ActivityManagerNative.jav

```
a  
  
static public IActivityManager getDefault() {  
    return gDefault.get();  
}  
  
private static final Singleton<IActivityManager> gDefault = new  
Singleton<IActivityManager>() {  
    protected IActivityManager create() {
```

```
IBinder b = ServiceManager.getService("activity");//1

if (false) {

    Log.v("ActivityManager", "default service binder = "
+ b);

}

IActivityManager am = asInterface(b);//2

if (false) {

    Log.v("ActivityManager", "default service = " + am);

}

return am;

}+

};

}
```

getDefault 方法调用了 gDefault 的 get 方法，我们接着往下看，gDefault 是一个 Singleton 类。注释 1 处得到名为“ activity” 的 Service 引用，也就是 IBinder 类型的 AMS 的引用。接着在注释 2 处将它封装成 AMP 类型对象，并将它保存到 gDefault 中 此后调用 AMN 的 getDefault 方法就会直接获得 AMS 的代理对象 AMP。注释 2 处的 asInterface 方法如下所示。

```
frameworks/base/core/java/android/app/ActivityManagerNative.jav

a

static public IActivityManager asInterface(IBinder obj) {

    if (obj == null) {
```

```
    return null;

}

IActivityManager in =
(IActivityManager)obj.queryLocalInterface(descriptor);

if (in != null) {

    return in;

}

return new ActivityManagerProxy(obj);

}
```

asInterface 方法的主要作用就是将 IBinder 类型的 AMS 引用封装成 AMP ,
AMP 的构造方法如下所示。

```
frameworks/base/core/java/android/app/ActivityManagerNative.jav
a

class ActivityManagerProxy implements IActivityManager
{
    public ActivityManagerProxy(IBinder remote)
    {
        mRemote = remote;
    }

    ...
}
```

AMP 的构造方法中将 AMS 的引用赋值给变量 mRemote , 这样在 AMP 中就可以使用 AMS 了。

其中 IActivityManager 是一个接口 , AMN 和 AMP 都实现了这个接口 , 用于实现代理模式和 Binder 通信。

再回到 Instrumentation 的 execStartActivity 方法 , 来查看 AMP 的 startActivity 方法 , AMP 是 AMN 的内部类 , 代码如下所示。

```
frameworks/base/core/java/android/app/ActivityManagerNative.java

public int startActivity(IApplicationThread caller, String
callingPackage, Intent intent,
String resolvedType, IBinder resultTo, String resultWho,
int requestCode,
int startFlags, ProfilerInfo profilerInfo, Bundle options)
throws RemoteException {

    ...
    data.writeInt(requestCode);
    data.writeInt(startFlags);
    ...

    mRemote.transact(START_ACTIVITY_TRANSACTION, data,
    reply, 0); //1
    reply.readException();+
    int result = reply.readInt();
```

```
    reply.recycle();

    data.recycle();

    return result;

}
```

首先会将传入的参数写入到 Parcel 类型的 data 中。在注释 1 处，通过 IBinder 类型对象 mRemote (AMS 的引用) 向服务端的 AMS 发送一个 START_ACTIVITY_TRANSACTION 类型的进程间通信请求。那么服务端 AMS 就会从 Binder 线程池中读取我们客户端发来的数据，最终会调用 AMN 的 onTransact 方法，如下所示。

```
frameworks/base/core/java/android/app/ActivityManagerNative.jav
a

@Override
public boolean onTransact(int code, Parcel data, Parcel reply, int
flags)
throws RemoteException {
switch (code) {
case START_ACTIVITY_TRANSACTION:
{
...
int result = startActivity(app, callingPackage, intent,
resolvedType,
resultTo, resultWho, requestCode, startFlags,
```

```
profilerInfo, options);

    reply.writeNoException();

    reply.writeInt(result);

    return true;

}

}
```

onTransact 中会调用 AMS 的 startActivity 方法，如下所示。

```
frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

@Override

public final int startActivity(IApplicationThread caller, String callingPackage,

    Intent intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode,

    int startFlags, ProfilerInfo profilerInfo, Bundle bOptions) {

    return    startActivityAsUser(caller,    callingPackage,    intent,

resolvedType, resultTo,

    resultWho,    requestCode,    startFlags,    profilerInfo,

bOptions,

    UserHandle.getCallingUserId());

}
```

startActivity 方法会最后 return startActivityAsUser 方法，如下所示。

```
    @Override

    public final int startActivityAsUser(IApplicationThread caller, String
callingPackage,

        Intent intent, String resolvedType, IBinder resultTo, String
resultWho, int requestCode,
        int startFlags, ProfilerInfo profilerInfo, Bundle bOptions, int
userId) {

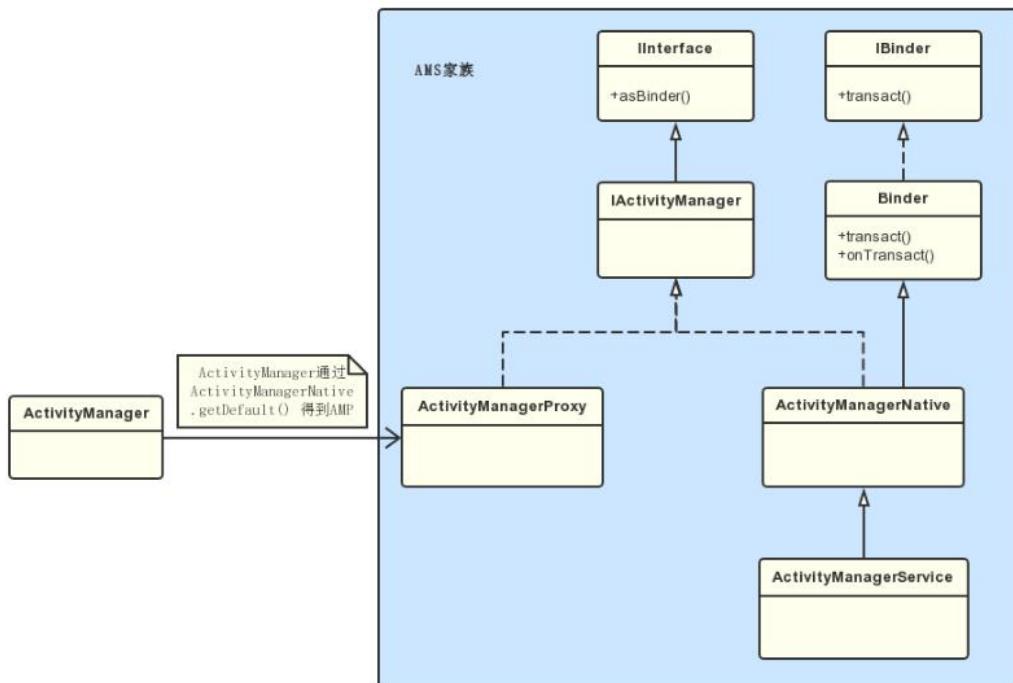
    enforceNotIsolatedCaller("startActivity");

    userId = mUserController.handleIncomingUser(Binder.getCallingPid(),
    Binder.getCallingUid(),
    userId, false, ALLOW_FULL_ONLY, "startActivity", null);

    return mActivityStarter.startActivityMayWait(caller, -1,
callingPackage, intent,
    resolvedType, null, null, resultTo, resultWho, requestCode,
startFlags,
    profilerInfo, null, null, bOptions, false, userId, null, null);
}
```

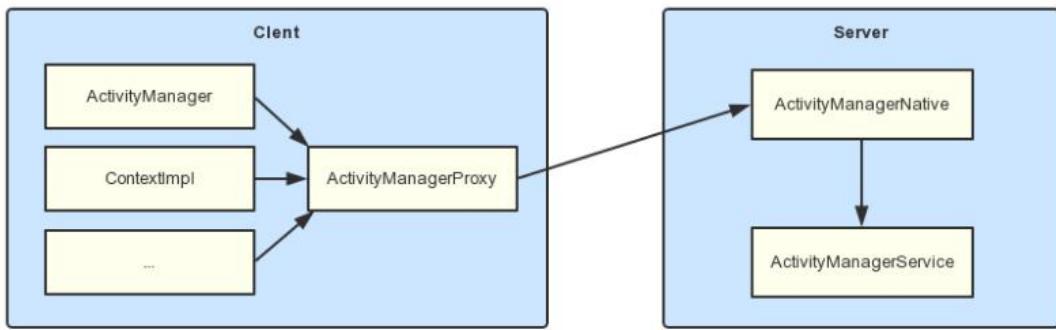
startActivityAsUser 方法最后会 return ActivityStarter 的
startActivityMayWait 方法，这一调用过程已经脱离了本节要讲的 AMS 家族，
因此这里不做介绍了，具体的调用过程可以查看 [Android 深入四大组件（一）](#)
[应用程序启动过程（后篇）](#)这篇文章。

在 Activity 的启动过程中提到了 AMP、AMN 和 AMS，它们共同组成了 AMS 家族的主要部分，如下图所示。



AMP 是 AMN 的内部类，它们都实现了 IActivityManager 接口，这样它们就可以实现代理模式，具体来讲是远程代理：AMP 和 AMN 是运行在两个进程的，AMP 是 Client 端，AMN 则是 Server 端，而 Server 端中具体的功能都是由 AMN 的子类 AMS 来实现的，因此，AMP 就是 AMS 在 Client 端的代理类。AMN 又实现了 Binder 类，这样 AMP 可以和 AMS 就可以通过 Binder 来进行进程间通信。

ActivityManager 通过 AMN 的 getDefault 方法得到 AMP，通过 AMP 就可以和 AMN 进行通信，也就是间接的与 AMS 进行通信。除了 ActivityManager，其他想要与 AMS 进行通信的类都需要通过 AMP，如下图所示。



3、WindowManagerService 启动过程解析

1.WMS 概述

WMS 是系统的其他服务，无论对于应用开发还是 Framework 开发都是重点的知识，它的职责有很多，主要有以下几点：

窗口管理

WMS 是窗口的管理者，它负责窗口的启动、添加和删除，另外窗口的大小和层级也是由 WMS 进行管理的。窗口管理的核心成员有 DisplayContent、WindowToken 和 WindowState。

窗口动画

窗口间进行切换时，使用窗口动画可以显得更炫一些，窗口动画由 WMS 的动画子系统来负责，动画子系统的管理者为 WindowAnimator。

输入系统的中转站

通过对窗口的触摸从而产生触摸事件，InputManagerService (IMS)会对触摸事件进行处理，它会寻找一个最合适的窗口来处理触摸反馈信息，WMS 是窗口的管理者，因此，WMS “理所应当” 的成为了输入系统的中转站。

Surface 管理

窗口并不具备有绘制的功能，因此每个窗口都需要有一块 Surface 来供自己绘制。为每个窗口分配 Surface 是由 WMS 来完成的。

WMS 的职责可以简单总结为下图。

[外链图片转存失败

(img-dQZmmKDp-1563381456222)(<https://s2.ax1x.com/2019/05/28/VexIGd.png>)]

2.WMS 的诞生

WMS 的知识点非常多，在了解这些知识点前，我们十分有必要知道 WMS 是如何产生的。WMS 是在 SystemServer 进程中启动的，不了解 SystemServer 进程的可以查看在 [Android 系统启动流程（三）解析 SystemServer 进程启动过程这篇文章](#)。

先来查看 SystemServer 的 main 方法：

```
public static void main(String[] args) {  
    new SystemServer().run();  
}
```

main 方法中只调用了 SystemServer 的 run 方法，如下所示。

```
private void run() {
```

```
try {

    System.loadLibrary("android_servers");//1

    ...

    mSystemServiceManager = new
SystemServiceManager(mSystemContext);//2

mSystemServiceManager.setRuntimeRestarted(mRuntimeRestart);

    LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);

    // Prepare the thread pool for init tasks that can be
parallelized

    SystemServerInitThreadPool.get();

} finally {

    traceEnd(); // InitBeforeStartServices

}

try {

    traceBeginAndSlog("StartServices");

    startBootstrapServices();//3

    startCoreServices();//4

    startOtherServices();//5

    SystemServerInitThreadPool.shutdown();

} catch (Throwable ex) {
```

```
Slog.e("System", "*****");
Slog.e("System", "***** Failure starting system
services", ex);
throw ex;
} finally {
    traceEnd();
}
...
}
```

run 方法代码很多，这里截取了关键的部分，在注释 1 处加载了 libandroid_servers.so。在注释 2 处创建 SystemServiceManager，它会对系统的服务进行创建、启动和生命周期管理。接下来的代码会启动系统的各种服务，在注释 3 中的 startBootstrapServices 方法中用 SystemServiceManager 启动了 ActivityManagerService、PowerManagerService、PackageManagerService 等服务。在注释 4 处的方法中则启动了 BatteryService、UsageStatsService 和 WebViewUpdateService。注释 5 处的 startOtherServices 方法中则启动了 CameraService、AlarmManagerService、VrManagerService 等服务，这些服务的父类为 SystemService。从注释 3、4、5 的方法名称可以看出，官方把大概 100 多个系统服务分为了三种类型，分别是引导服务、核心服务和其他服务，其中其他服务为一些非紧要和一些不需要立即启动的服务，WMS 就是其他服务的一种。

我们来查看 startOtherServices 方法是如何启动 WMS 的：

```
private void startOtherServices() {  
    ...  
    traceBeginAndSlog("InitWatchdog");  
    final Watchdog watchdog = Watchdog.getInstance(); //1  
    watchdog.init(context, mActivityManagerService); //2  
    traceEnd();  
    traceBeginAndSlog("StartInputManagerService");  
    inputManager = new InputManagerService(context); //3  
    traceEnd();  
    traceBeginAndSlog("StartWindowManagerService");  
  
    ConcurrentUtils.waitForFutureNoInterrupt(mSensorServiceStart,  
    START_SENSOR_SERVICE);  
    mSensorServiceStart = null;  
    wm = WindowManagerService.main(context, inputManager,  
    mFactoryTestMode !=  
    FactoryTest.FACTORY_TEST_LOW_LEVEL,  
    !mFirstBoot, mOnlyCore, new  
    PhoneWindowManager()); //4  
    ServiceManager.addService(Context.WINDOW_SERVICE,  
    wm); //5  
    ServiceManager.addService(Context.INPUT_SERVICE,
```

```
    inputManager); //6

    traceEnd();

    ...

    try {

        wm.displayReady(); //7

    } catch (Throwable e) {

        reportWtf("making display ready", e);

    }

    ...

    try {

        wm.systemReady(); //8

    } catch (Throwable e) {

        reportWtf("making Window Manager Service ready", e);

    }

    ...

}

startOtherServices 方法用于启动其他服务，其他服务大概有 70 多个，上面的代码只列出了 WMS 以及和它相关的 IMS 的启动逻辑，剩余的其他服务的启动逻辑也都大同小异。
```

在注释 1、2 处分别得到 Watchdog 实例并对它进行初始化，Watchdog 用来监控系统的一些关键服务的运行状况，后文会再次提到它。在注释 3 处创建了 IMS，并赋值给 IMS 类型的 inputManager 对象。注释 4 处执行了 WMS 的

main 方法，其内部会创建 WMS，需要注意的是 main 方法其中一个传入的参数就是注释 1 处创建的 IMS，WMS 是输入事件的中转站，其内部包含了 IMS 引用并不意外。结合上文，我们可以得知 WMS 的 main 方法是运行在 SystemServer 的 run 方法中，换句话说就是运行在"system_server"线程" 中，后面会再次提到"system_server"线程。

注释 5 和注释 6 处分别将 WMS 和 IMS 注册到 ServiceManager 中，这样如果某个客户端想要使用 WMS，就需要先去 ServiceManager 中查询信息，然后根据信息与 WMS 所在的进程建立通信通路，客户端就可以使用 WMS 了。注释 7 处用来初始化显示信息，注释 8 处则用来通知 WMS，系统的初始化工作已经完成，其内部调用了 WindowManagerPolicy 的 systemReady 方法。

我们来查看注释 4 处 WMS 的 main 方法，如下所示。

```
public static WindowManagerService main(final Context context, final
InputManagerService im,
final boolean haveInputMethods, final boolean
showBootMsgs, final boolean onlyCore,
WindowManagerPolicy policy) {
    DisplayThread.getHandler().runWithScissors(() -> //1
        sInstance = new WindowManagerService(context, im,
        haveInputMethods, showBootMsgs,
        onlyCore, policy), 0);
    return sInstance;
}
```

在注释 1 处调用了 DisplayThread 的 getHandler 方法，用来得到 DisplayThread 的 Handler 实例。DisplayThread 是一个单例的前台线程，这个线程用来处理需要低延时显示的相关操作，并只能由 WindowManager、DisplayManager 和 InputManager 实时执行快速操作。注释 1 处的 runWithScissors 方法中使用了 Java8 中的 Lambda 表达式，它等价于如下代码：

```
DisplayThread.getHandler().runWithScissors(new Runnable {  
    @Override  
    public void run() {  
        sInstance = new WindowManagerService(context, im,  
                haveInputMethods, showBootMsgs,  
                onlyCore, policy); //2  
    }  
}, 0);
```

在注释 2 处创建了 WMS 的实例，这个过程运行在 Runnable 的 run 方法中，而 Runnable 则传入到了 DisplayThread 对应 Handler 的 runWithScissors 方法中，说明 WMS 的创建是运行在 “android.display” 线程中。需要注意的是，runWithScissors 方法的第二个参数传入的是 0，后面会提到。来查看 Handler 的 runWithScissors 方法里做了什么：

```
public final boolean runWithScissors(final Runnable r, long timeout) {  
    if (r == null) {  
        throw new IllegalArgumentException("Runnable must not be null");  
    }  
    synchronized (this) {  
        if (mHandler != null) {  
            mHandler.removeCallbacks(r);  
        }  
        mRunnable = r;  
        mTimeout = timeout;  
        mHandler.post(this);  
    }  
}
```

```
    null");
}

if (timeout < 0) {

    throw new IllegalArgumentException("timeout must be
non-negative");

}

if (Looper.myLooper() == mLooper) { //1

    r.run();

    return true;

}

BlockingRunnable br = new BlockingRunnable(r);

return br.postAndWait(this, timeout);

}
```

开头对传入的 Runnable 和 timeout 进行了判断，如果 Runnable 为 null 或者 timeout 小于 0 则抛出异常。注释 1 处根据每个线程只有一个 Looper 的原理来判断当前的线程（"system_server" 线程）是否是 Handler 所指向的线程（"android.display" 线程），如果是则直接执行 Runnable 的 run 方法，如果不是则调用 BlockingRunnable 的 postAndWait 方法，并将当前线程的 Runnable 作为参数传进去，BlockingRunnable 是 Handler 的内部类，代码如下所示。

```
private static final class BlockingRunnable implements Runnable {

    private final Runnable mTask;
```

```
private boolean mDone;

public BlockingRunnable(Runnable task) {

    mTask = task;

}

@Override

public void run() {

    try {

        mTask.run(); //1

    } finally {

        synchronized (this) {

            mDone = true;

            notifyAll();

        }

    }

}

public boolean postAndWait(Handler handler, long timeout) {

    if (!handler.post(this)) //2

        return false;

}

synchronized (this) {

    if (timeout > 0) {

        final long expirationTime =
```

```
SystemClock.uptimeMillis() + timeout;

        while (!mDone) {

            long      delay      =      expirationTime      -
SystemClock.uptimeMillis();

            if (delay <= 0) {

                return false; // timeout

            }

            try {

                wait(delay);

            } catch (InterruptedException ex) {

                }

            }

        } else {

            while (!mDone) {

                try {

                    wait();//3

                } catch (InterruptedException ex) {

                }

            }

        }

    }

    return true;
}
```

```
    }  
  
}
```

注释 2 处将当前的 BlockingRunnable 添加到 Handler 的任务队列中。前面 runWithScissors 方法的第二个参数为 0 , 因此 timeout 等于 0 , 这样如果 mDone 为 false 的话会一直调用注释 3 处的 wait 方法使得当前线程 ("system_server" 线程) 进入等待状态 , 那么等待的是哪个线程呢 ? 我们往上看 , 注释 1 处 , 执行了传入的 Runnable 的 run 方法(运行在 "android.display" 线程) , 执行完毕后在 finally 代码块中将 mDone 设置为 true , 并调用 notifyAll 方法唤醒处于等待状态的线程 , 这样就不会继续调用注释 3 处的 wait 方法。因此得出结论 , "system_server" 线程线程等待的就是 "android.display" 线程 , 一直到 "android.display" 线程执行完毕再执行 "system_server" 线程 , 这是因为 "android.display" 线程内部执行了 WMS 的创建 , 显然 WMS 的创建优先级更高些。

WMS 的创建就讲到这 , 最后我们来查看 WMS 的构造方法 :

```
private WindowManagerService(Context context,  
InputManagerService inputManager,  
boolean haveInputMethods, boolean showBootMsgs,  
boolean onlyCore, WindowManagerPolicy policy) {  
  
    ...  
  
    mInputManager = inputManager;//1  
  
    ...  
  
    mDisplayManager = null; //2
```

```
(DisplayManager)context.getSystemService(Context.DISPLAY_SERVICE);

    mDisplays = mDisplayManager.getDisplays();//2

    for (Display display : mDisplays) {

        createDisplayContentLocked(display);//3

    }

    ...

    mActivityManager = ActivityManager.getService();//4

    ...

    mAnimator = new WindowAnimator(this);//5

    mAllowTheaterModeWakeFromLayout = context.getResources().getBoolean(
        com.android.internal.R.bool.config_allowTheaterModeWakeFromWind
        wLayout);

    LocalServices.addService(WindowManagerInternal.class, new
        LocalService());

    initPolicy();//6

    // Add ourself to the Watchdog monitors.

    Watchdog.getInstance().addMonitor(this);//7

    ...

}
```

注释 1 处用来保存传进来的 IMS , 这样 WMS 就持有了 IMS 的引用。注释 2 处通过 DisplayManager 的 getDisplays 方法得到 Display 数组 (每个显示设备都有一个 Display 实例) , 接着遍历 Display 数组 , 在注释 3 处的 createDisplayContentLocked 方法会将 Display 封装成 DisplayContent , DisplayContent 用来描述一块屏幕。

注释 4 处得到 AMS 实例 , 并赋值给 mActivityManager , 这样 WMS 就持有了 AMS 的引用。注释 5 处创建了 WindowAnimator , 它用于管理所有的窗口动画。注释 6 处初始化了窗口管理策略的接口类 WindowManagerPolicy (WMP) , 它用来定义一个窗口策略所要遵循的通用规范。注释 7 处将自身也就是 WMS 通过 addMonitor 方法添加到 Watchdog 中 , Watchdog 用来监控系统的一些关键服务的运行状况 (比如传入的 WMS 的运行状况) , 这些被监控的服务都会实现 Watchdog.Monitor 接口。 Watchdog 每分钟都会对被监控的系统服务进行检查 , 如果被监控的系统服务出现了死锁 , 则会杀死 Watchdog 所在的进程 , 也就是 SystemServer 进程。

查看注释 6 处的 initPolicy 方法 , 如下所示。

```
private void initPolicy() {  
    UiThread.getHandler().runWithScissors(new Runnable() {  
        @Override  
        public void run() {  
            WindowManagerPolicyThread.set(Thread.currentThread(),
```

```
Looper.myLooper());  
mPolicy.init(mContext, WindowManagerService.this,  
WindowManagerService.this); //1  
}  
, 0);  
}  
}
```

initPolicy 方法和此前讲的 WMS 的 main 方法的实现类似，注释 1 处执行了 WMP 的 init 方法，WMP 是一个接口，init 方法的具体实现现在 PhoneWindowManager (PWM) 中。PWM 的 init 方法运行在"android.ui" 线程中，它的优先级要高于 initPolicy 方法所在的"android.display"线程，因此 "android.display"线程要等 PWM 的 init 方法执行完毕后，处于等待状态的 "android.display"线程才会被唤醒从而继续执行下面的代码。

4、PMS 启动流程解析

我们来分析 Android 的 PackageManagerService ,后面简称 PMS。PMS 用来管理所有的 package 信息，包括安装、卸载、更新以及解析 AndroidManifest.xml 以组织相应的数据结构，这些数据结构将会被 PMS、ActivityMangerService 等等 service 和 application 使用到。PMS 有几个比较重要的命令可以用于我们 debug 中：

adb shell dumpsys package (dump 出系统中所有的 application 信息)

```
adb shell dumpsys package "com.android.contacts" p (dump 出系统中  
特定包名的 application 信息)
```

首先来看 SystemServer 中 PMS 的构造以及注册：

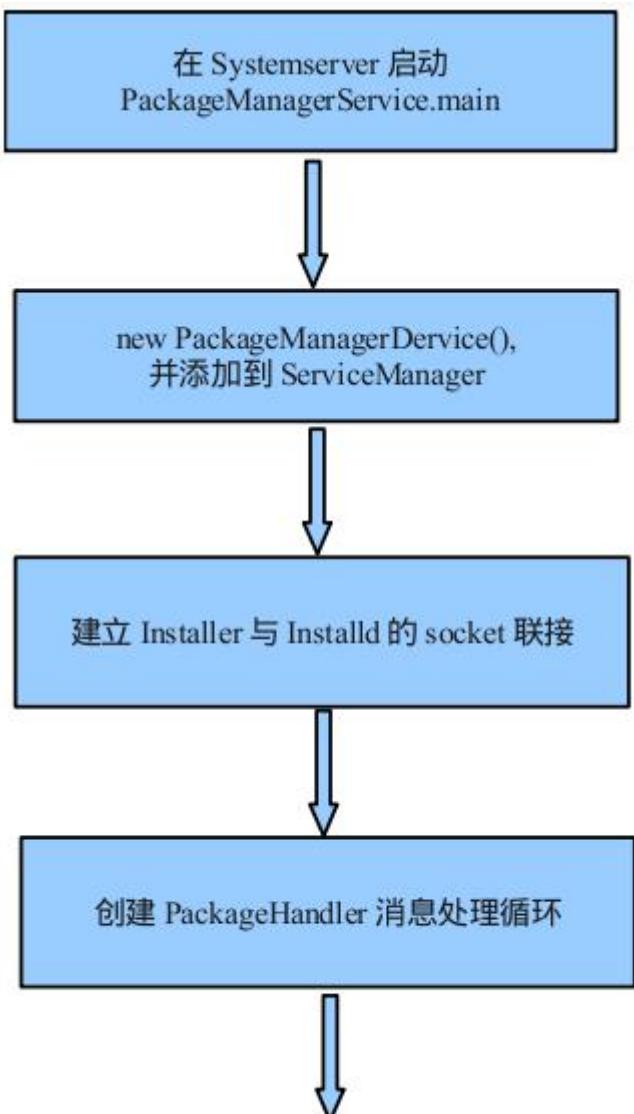
```
pm = PackageManagerService.main(context, installer,  
        factoryTest != SystemServer.FACTORY_TEST_OFF,  
        onlyCore);  
  
try {  
    firstBoot = pm.isFirstBoot();  
}  
catch (RemoteException e) {  
}  
  
try {  
    pm.performBootDexOpt();  
}  
catch (Throwable e) {  
    reportWtf("performing boot dexopt", e);  
}  
  
try {  
    pm.systemReady();
```

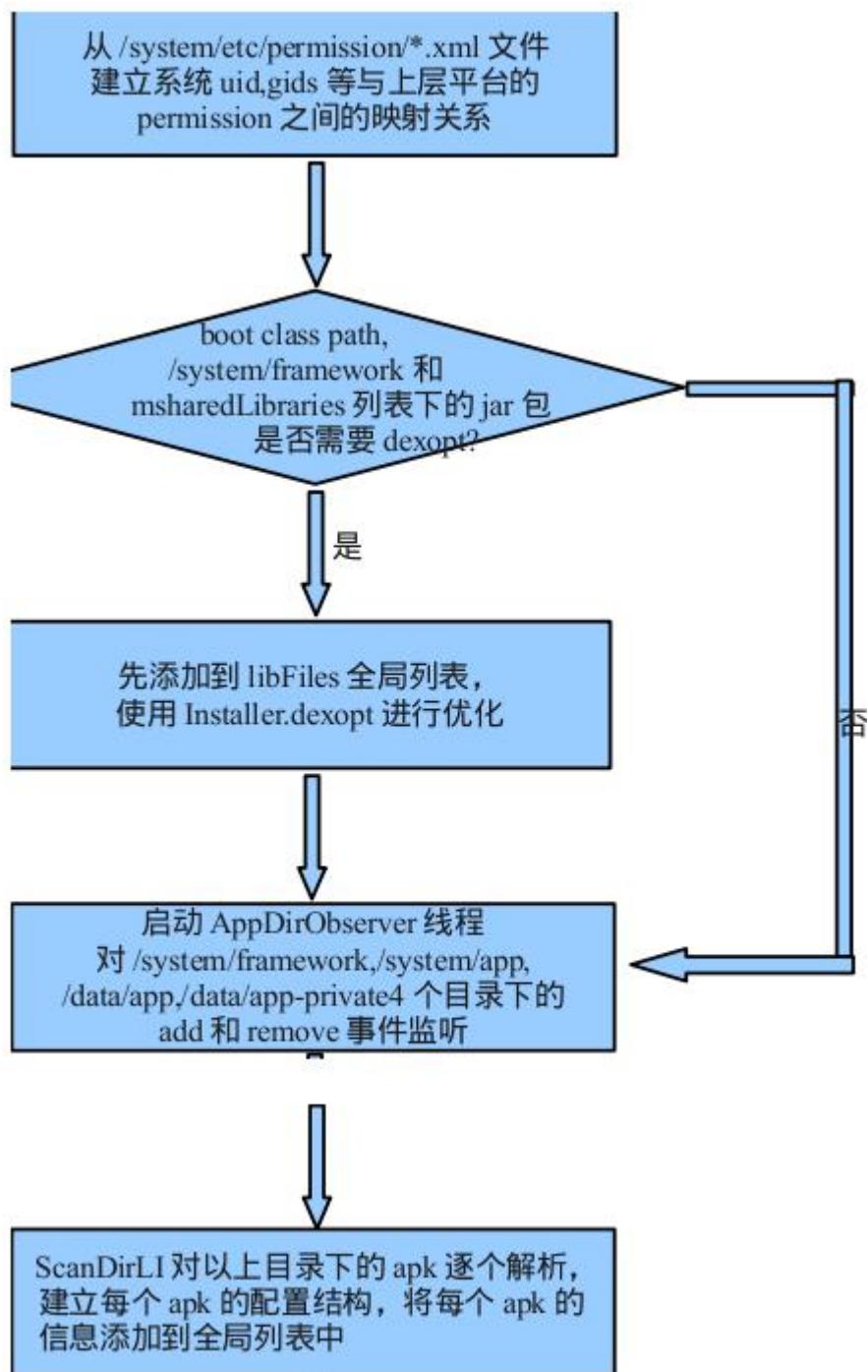
```
        } catch (Throwable e) {  
  
            reportWtf("making Package Manager Service ready", e);  
  
        }  
    }
```

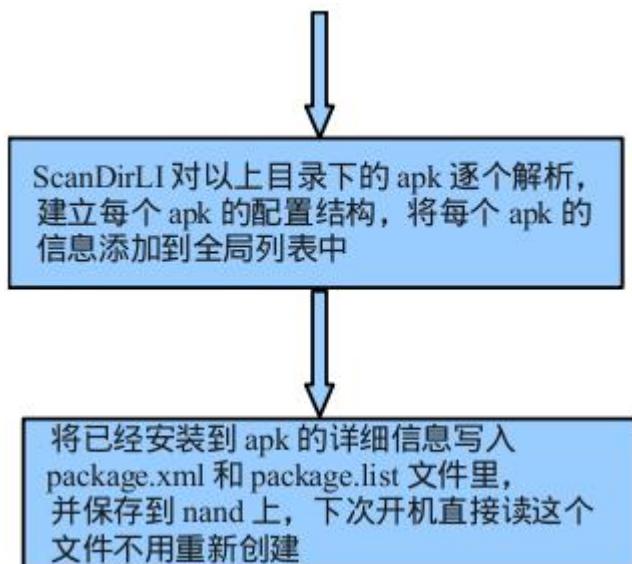
首先来看 PMS 的 main 方法：

```
public static final IPackageManager main(Context context, Installer  
installer,  
  
        boolean factoryTest, boolean onlyCore) {  
  
    PackageManagerService m = new  
PackageManagerService(context, installer,  
  
        factoryTest, onlyCore);  
  
    ServiceManager.addService("package", m);  
  
    return m;  
}
```

首先构造一个 PMS 对象，然后调用 ServiceManager 的 addService 注册这个服务。构造函数的第二个参数是一个 Installer 对象，用于和 Installd 通信使用，我们后面分析 Installd 再来介绍；第三个参数 factoryTest 为出厂测试，默認為 false；第四个参数 onlyCore 与 vold 相关，我们以后再分析，这里也为 false。PMS 的构造函数比较长，我们首先来看一下大概的流程图，然后我们分段来分析代码：







```
public PackageManagerService(Context context, Installer  
installer,  
boolean factoryTest, boolean onlyCore) {
```

```
    mContext = context;  
  
    mFactoryTest = factoryTest;  
  
    mOnlyCore = onlyCore;  
  
    mNoDexOpt =  
    "eng".equals(SystemProperties.get("ro.build.type"));  
  
    mMetrics = new DisplayMetrics();  
  
    mSettings = new Settings(context);  
  
    mSettings.addSharedUserLPw("android.uid.system",  
    Process.SYSTEM_UID,  
  
    ApplicationInfo.FLAG_SYSTEM|ApplicationInfo.FLAG_PRIVILEGED);
```

```
mSettings.addSharedUserLPw("android.uid.phone",
RADIO_UID,
ApplicationInfo.FLAG_SYSTEM|ApplicationInfo.FLAG_PRIVILEGED);

mSettings.addSharedUserLPw("android.uid.log", LOG_UID,
ApplicationInfo.FLAG_SYSTEM|ApplicationInfo.FLAG_PRIVILEGED);

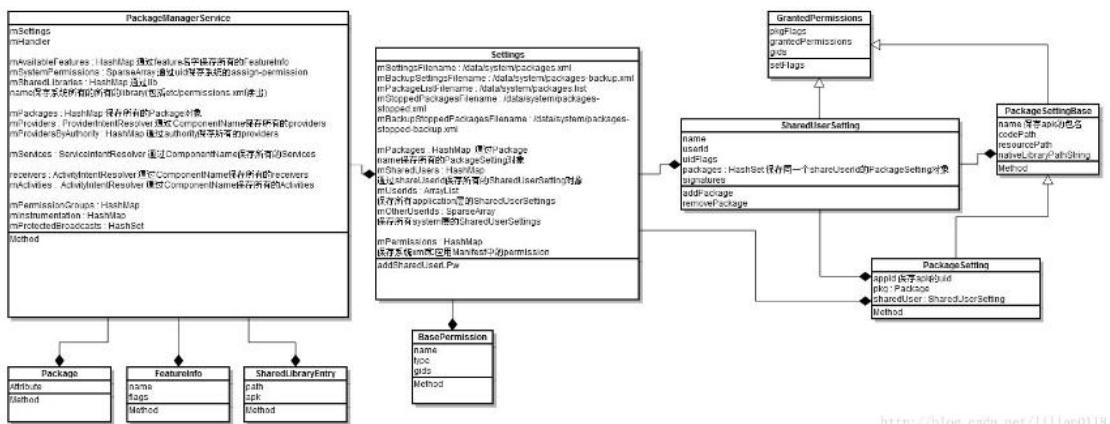
mSettings.addSharedUserLPw("android.uid.nfc", NFC_UID,
ApplicationInfo.FLAG_SYSTEM|ApplicationInfo.FLAG_PRIVILEGED);

mSettings.addSharedUserLPw("android.uid.bluetooth",
BLUETOOTH_UID,
ApplicationInfo.FLAG_SYSTEM|ApplicationInfo.FLAG_PRIVILEGED);

mSettings.addSharedUserLPw("android.uid.shell",
SHELL_UID,
ApplicationInfo.FLAG_SYSTEM|ApplicationInfo.FLAG_PRIVILEGED);
```

上面首先做一个变量的赋值，然后取出"ro.build.type"属性值，build 版本分为 user 和 eng 两种，一种是面向 user，一种是用于 engineer debug 版，这里假设 mNoDexOpt 为 false。然后构造一个 Settings 对象，Settings 是

Android 的全局管理者，用于协助 PMS 保存所有的安装包信息，PMS 和 Settings 之间的类图关系如下：



来看一下 `Settings` 的构造函数：

```
Settings(Context context) {  
    this(context, Environment.getDataDirectory());  
}  
}
```

```
Settings(Context context, File dataDir) {  
    mContext = context;  
  
    mSystemDir = new File(dataDir, "system");  
  
    mSystemDir.mkdirs();  
  
    FileUtils.setPermissions(mSystemDir.toString(),  
        FileUtils.S_IRWXU | FileUtils.S_IRWXG  
        | FileUtils.S_IROTH | FileUtils.S_IXOTH,  
        -1, -1);  
  
    mSettingsFilename = new File(mSystemDir, "pa
```

```
mBackupSettingsFilename = new File(mSystemDir,  
"packages-backup.xml");  
  
mPackageListFilename = new File(mSystemDir, "packages.list");  
  
FileUtils.setPermissions(mPackageListFilename, 0660,  
SYSTEM_UID, PACKAGE_INFO_GID);  
  
  
  
mStoppedPackagesFilename = new File(mSystemDir,  
"packages-stopped.xml");  
  
mBackupStoppedPackagesFilename = new File(mSystemDir,  
"packages-stopped-backup.xml");  
  
}
```

Environment.getDataDirectory()返回/data 目录，然后创建 /data/system/目录，并设置它的权限，并在/data/system 目录中创建 mSettingsFilename、mBackupSettingsFilename、mPackageListFilename、 mStoppedPackagesFilename 和 mBackupStoppedPackagesFilename 几个文件。 packages.xml 就是保存了系统所有的 Package 信息， packages-backup.xml 是 packages.xml 的备份，防止在写 packages.xml 突然断电等问题。回到 PMS 的构造函数，调用 addSharedUserLPw 将几种 SharedUserId 的名字和它对应的 UID 对应写到 Settings 当中。关于 SharedUserId 的使用，我们在后面介绍 APK 的安装过程中再来分析。这里先简单看一下 Process 中提供的 UID 列表：

```
1 public static final int SYSTEM_UID = 1000;
2 public static final int PHONE_UID = 1001;
3 public static final int SHELL_UID = 2000;
4 public static final int LOG_UID = 1007;
5 public static final int WIFI_UID = 1010;
6 public static final int MEDIA_UID = 1013;
7 public static final int DRM_UID = 1019;
8 public static final int VPN_UID = 1016;
9 public static final int NFC_UID = 1027;
10 public static final int BLUETOOTH_UID = 1002;
11 public static final int MEDIA_RW_GID = 1023;
12 public static final int PACKAGE_INFO_GID = 1032;
13 public static final int FIRST_APPLICATION_UID = 10000;
14 public static final int LAST_APPLICATION_UID = 19999;
```

复制

上面定义了一系列的 UID，其中 application 的 uid 从 10000 开始到 19999 结束。来看 addSharedUserLPw 函数的实现：

```
SharedUserSetting addSharedUserLPw(String name, int uid, int pkgFlags) {
    SharedUserSetting s = mSharedUsers.get(name);
    if (s != null) {
        if (s.userId == uid) {
            return s;
        }
        PackageManagerService.reportSettingsProblem(Log.ERROR,
                "Adding duplicate shared user, keeping first: " + name);
        return null;
    }
    s = new SharedUserSetting(name, pkgFlags);
    s.userId = uid;
    if (addUserIdLPw(uid, s, name)) {
        mSharedUsers.put(name, s);
        return s;
    }
    return null;
}
```

```
private boolean addUserIdLPw(int uid, Object obj, Object name) {
    if (uid > Process.LAST_APPLICATION_UID) {
        return false;
    }

    if (uid >= Process.FIRST_APPLICATION_UID) {
        int N = mUserIds.size();
        final int index = uid - Process.FIRST_APPLICATION_UID;
        while (index >= N) {
            mUserIds.add(null);
            N++;
        }
    }
}
```

```

        if (mUserIds.get(index) != null) {
            PackageManagerService.reportSettingsProblem(Log.ERROR,
                "Adding duplicate user id: " + uid
                + " name=" + name);
            return false;
        }
        mUserIds.set(index, obj);
    } else {
        if (mOtherUserIds.get(uid) != null) {
            PackageManagerService.reportSettingsProblem(Log.ERROR,
                "Adding duplicate shared id: " + uid
                + " name=" + name);
            return false;
        }
        mOtherUserIds.put(uid, obj);
    }
    return true;
}

```

mSharedUsers 是一个 HashMap，保存着所有的 name 和 SharedUserSetting 的映射关系。这里先调用 addUserIdLPw 将 uid 和 SharedUserSetting 添加到 mOtherUserIds 中，然后将 name 和 SharedUserSetting 添加到 mSharedUsers 中方便以后查找。接着来看 PMS 的构造函数：

```

mInstaller = installer;

WindowManager wm =
(WindowManager)context.getSystemService(Context.WINDOW_SERVICE);
Display d = wm.getDefaultDisplay();
d.getMetrics(mMetrics);

synchronized (mInstallLock) {
// writer
synchronized (mPackages) {
    mHandlerThread.start();
    mHandler = new PackageHandler(mHandlerThread.getLooper());
    Watchdog.getInstance().addThread(mHandler,
mHandlerThread.getName(),
    WATCHDOG_TIMEOUT);

    File dataDir = Environment.getDataDirectory();
    mAppDataDir = new File(dataDir, "data");
    mAppInstallDir = new File(dataDir, "app");
    mAppLibInstallDir = new File(dataDir, "app-lib");
    mAsecInternalPath = new File(dataDir, "app-asec").getPath();
    mUserAppDataDir = new File(dataDir, "user");
}
}

```

```
mDrmAppPrivateInstallDir = new File(dataDir, "app-private");

sUserManager = new UserManagerService(context, this,
        mInstallLock, mPackages);

readPermissions();
```

上面首先获得显示屏的相关信息并保存在 mMetrics 中。然后启动“PackageManager”的HandleThread 并绑定到 PackageHandler 上，这就是最后处理所有的跨进程消息的 handler。接着调用 readPermissions() 来处理系统的 permissions 相关的文件。在/etc/permissions 的文件大多来源于代码中的 frameworks/native/data/etc，这些文件的作用是表明系统支持的 feature 有哪些，例如是否支持蓝牙、wifi、P2P 等。文件目录如下：

1. android.hardware.bluetooth.xml
 2. android.hardware.bluetooth_le.xml
 3. android.hardware.camera.autofocus.xml
 4. android.hardware.camera.flash-autofocus.xml
 5. android.hardware.camera.front.xml
 6. android.hardware.camera.xml
-
1. android.software.sip.xml
 2. com.android.nfc_extras.xml
 3. com.nxp.mifare.xml
 4. handheld_core_hardware.xml

这里的文件内容很简单，例如 android.hardware.bluetooth.xml 的内容如下：

```
1 <permissions>
2     <feature name="android.hardware.bluetooth" />
3 </permissions>
```

复制

在/etc/permissions 中有一个 platform.xml, 它是来源于 frameworks/base/data/etc/中, 其中的内容大致如下:

```
<permissions>

    <permission name="android.permission.BLUETOOTH_ADMIN" >
        <group gid="net_bt_admin" />
    </permission>

    <permission name="android.permission.BLUETOOTH" >
        <group gid="net_bt" />
    </permission>

    <permission name="android.permission.BLUETOOTH_STACK" >
        <group gid="net_bt_stack" />
    </permission>

    <permission name="android.permission.NET_TUNNELING" >
        <group gid="vpn" />
    </permission>

    <permission name="android.permission.INTERNET" >
        <group gid="inet" />
    </permission>

    <assign-permission name="android.permission.MODIFY_AUDIO_SETTINGS"
uid="media" />
    <assign-permission name="android.permission.ACCESS_SURFACE_FLINGER"
uid="media" />
        <assign-permission name="android.permission.WAKE_LOCK" uid="media" />
        <assign-permission name="android.permission.UPDATE_DEVICE_STATS"
uid="media" />
        <assign-permission name="android.permission.UPDATE_APP_OPS_STATS"
uid="media" />

    <assign-permission name="android.permission.ACCESS_SURFACE_FLINGER"
uid="graphics" />

    <library name="android.test.runner"
        file="/system/framework/android.test.runner.jar" />
    <library name="javax.obex"
```

```
    file="/system/framework/javax.obex.jar"/>
```

现在来看 `readPermissions()` 的实现：

```
void readPermissions() {
    File libraryDir = new File(Environment.getRootDirectory(), "etc/permissions");
    if (!libraryDir.exists() || !libraryDir.isDirectory()) {
        Slog.w(TAG, "No directory " + libraryDir + ", skipping");
        return;
    }
    if (!libraryDir.canRead()) {
        Slog.w(TAG, "Directory " + libraryDir + " cannot be read");
        return;
    }

    for (File f : libraryDir.listFiles()) {
        if (f.getPath().endsWith("etc/permissions/platform.xml")) {
            continue;
        }

        if (!f.getPath().endsWith(".xml")) {
            Slog.i(TAG, "Non-xml file " + f + " in " + libraryDir + " directory,
ignoring");
            continue;
        }
        if (!f.canRead()) {
            Slog.w(TAG, "Permissions library file " + f + " cannot be read");
            continue;
        }

        readPermissionsFromXml(f);
    }

    // Read permissions from .../etc/permissions/platform.xml last so it will take
precedence
    final File permFile = new File(Environment.getRootDirectory(),
        "etc/permissions/platform.xml");
    readPermissionsFromXml(permFile);
}
```

首先不断的读出 `/etc/permissions` 下面的文件，并依此处理除了 `platform.xml` 以外的其它 `xml` 文件，并最后处理 `platform.xml` 文件，来看 `readPermissionsFromXml()` 的实现，这个函数比较长，我们主要看处理 `feature`、`permission`、`assign-permission` 和 `library` 的代码：

```
private void readPermissionsFromXml(File permFile) {
    FileReader permReader = null;
    try {
        permReader = new FileReader(permFile);
```

```

} catch (FileNotFoundException e) {
}

try {
    XmlPullParser parser = Xml.newPullParser();
    parser.setInput(permReader);

    XmlUtils.beginDocument(parser, "permissions");

    while (true) {
        XmlUtils.nextElement(parser);
        if (parser.getEventType() == XmlPullParser.END_DOCUMENT) {
            break;
        }

        String name = parser.getName();
        if ("group".equals(name)) {

        } else if ("permission".equals(name)) {
            String perm = parser.getAttributeValue(null, "name");
            if (perm == null) {
                Slog.w(TAG, "<permission> without name at "
                    + parser.getPositionDescription());
                XmlUtils.skipCurrentTag(parser);
                continue;
            }
            perm = perm.intern();
            readPermission(parser, perm);

        } else if ("assign-permission".equals(name)) {
            String perm = parser.getAttributeValue(null, "name");
            String uidStr = parser.getAttributeValue(null, "uid");
            if (uidStr == null) {
                Slog.w(TAG, "<assign-permission> without uid at "
                    + parser.getPositionDescription());
                XmlUtils.skipCurrentTag(parser);
                continue;
            }
            int uid = Process.getUidForName(uidStr);
            perm = perm.intern();
            HashSet<String> perms = mSystemPermissions.get(uid);
            if (perms == null) {
                perms = new HashSet<String>();
                mSystemPermissions.put(uid, perms);
            }
            perms.add(perm);
        }
    }
}

```

```
        }
        perms.add(perm);
        XmlUtils.skipCurrentTag(parser);

    } else if ("library".equals(name)) {
        String lname = parser.getAttributeValue(null, "name");
        String lfile = parser.getAttributeValue(null, "file");
        if (lname == null) {

            } else if (lfile == null) {

            } else {
                mSharedLibraries.put(lname, new
SharedLibraryEntry(lfile, null));
            }
            XmlUtils.skipCurrentTag(parser);
            continue;

        } else if ("feature".equals(name)) {
            String fname = parser.getAttributeValue(null, "name");
            if (fname == null) {

            } else {
                FeatureInfo fi = new FeatureInfo();
                fi.name = fname;
                mAvailableFeatures.put(fname, fi);
            }
            XmlUtils.skipCurrentTag(parser);
            continue;

        } else {
            XmlUtils.skipCurrentTag(parser);
            continue;
        }

    }
    permReader.close();
} catch (XmlPullParserException e) {
    Slog.w(TAG, "Got execption parsing permissions.", e);
} catch (IOException e) {
    Slog.w(TAG, "Got execption parsing permissions.", e);
}
}
```

首先来看处理 feature 这个 tag 的代码，在 fname 中保存 feature 的名字，然后创建一个 FeatureInfo，并把 fname 和 FeatureInfo 保存到 mAvailableFeatures 这个 HashMap 中。接着来看处理 permission tag，首先读出 permission 的 name，然后调用 readPermission 去处理后面的 group 信息：

```
void readPermission(XmlPullParser parser, String name)
    throws IOException, XmlPullParserException {

    name = name.intern();

    BasePermission bp = mSettings.mPermissions.get(name);
    if (bp == null) {
        bp = new BasePermission(name, null, BasePermission.TYPE_BUILTIN);
        mSettings.mPermissions.put(name, bp);
    }
    int outerDepth = parser.getDepth();
    int type;
    while ((type=parser.next()) != XmlPullParser.END_DOCUMENT
            && (type != XmlPullParser.END_TAG
                  || parser.getDepth() > outerDepth)) {
        if (type == XmlPullParser.END_TAG
            || type == XmlPullParser.TEXT) {
            continue;
        }

        String tagName = parser.getName();
        if ("group".equals(tagName)) {
            String gidStr = parser.getAttributeValue(null, "gid");
            if (gidStr != null) {
                int gid = Process.getGidByName(gidStr);
                bp.gids = appendInt(bp.gids, gid);
            } else {
                Slog.w(TAG, "<group> without gid at "
                      + parser.getPositionDescription());
            }
        }
        XmlUtils.skipCurrentTag(parser);
    }
}
```

在 readPermission 中首先构造 BasePermission 对象，并把 name 和 BasePermission 一起添加到 Settings 的 mPermissions 这个 HashMap 中。Android 管理权限的机制其实就是对应相应的 permission，用一个 gid 号来描述，当一个应用程序请求这个 permission 的时候，就把这个 gid 号添加到对应的 application 中去。Process.getGidByName 方法通过 JNI 调用 getgrnam 系统函数去获取相应的组名称所对应的 gid 号，并把它添加到 BasePermission 对象的 gids 数组中。再来看处理 assign-permission 这个 tag 的代码，首

先读出 permission 的名字和 uid，保存在 perm 和 uidStr 中，Process.getUidForName 方法通过 JNI 调用 getpwnam 系统函数获取相应的用户名所对应的 uid 号，并把刚解析的 permission 名添加到 HashSet 当中，最后把上面的 uid 和 hashset 添加到 mSystemPermissions 这个数组中。最后来看处理 library 这个 tag 的代码，这里把解析处理的 library 名字和路径保存在 mSharedLibraries 这个 hashMap 中。再回到 PMS 的构造函数中，接着往下来看：

```
mRestoredSettings = mSettings.readLPw(this,  
sUserManager.getUsers(false),  
        mSdkVersion, mOnlyCore);  
  
String customResolverActivity = Resources.getSystem().getString(  
        R.string.config_customResolverActivity);  
if (TextUtils.isEmpty(customResolverActivity)) {  
    customResolverActivity = null;  
} else {  
    mCustomResolverComponentName =  
ComponentName.unflattenFromString(  
        customResolverActivity);  
}  
  
long startTime = SystemClock.uptimeMillis();  
  
int scanMode = SCAN_MONITOR | SCAN_NO_PATHS |  
SCAN_DEFER_DEX | SCAN_BOOTING;  
if (mNoDexOpt) {  
    Slog.w(TAG, "Running ENG build: no pre-dexopt!");  
    scanMode |= SCAN_NO_DEX;  
}  
  
final HashSet<String> alreadyDexOpted = new HashSet<String>();  
  
String bootClassPath = System.getProperty("java.boot.class.path");  
if (bootClassPath != null) {  
    String[] paths = splitString(bootClassPath, ':');  
    for (int i=0; i<paths.length; i++) {  
        alreadyDexOpted.add(paths[i]);  
    }  
} else {  
    Slog.w(TAG, "No BOOTCLASSPATH found!");  
}  
  
boolean didDexOpt = false;  
  
if (mSharedLibraries.size() > 0) {
```

```

Iterator<SharedLibraryEntry> libs =
mSharedLibraries.values().iterator();
    while (libs.hasNext()) {
        String lib = libs.next().path;
        if (lib == null) {
            continue;
        }
        try {
            if (dalvik.system.DexFile.isDexOptNeeded(lib)) {
                alreadyDexOpted.add(lib);
                mInstaller.dexopt(lib, Process.SYSTEM_UID, true);
                didDexOpt = true;
            }
        } catch (FileNotFoundException e) {
            Slog.w(TAG, "Library not found: " + lib);
        } catch (IOException e) {
            Slog.w(TAG, "Cannot dexopt " + lib + "; is it an APK or JAR? "
+ e.getMessage());
        }
    }
}

```

这里首先调用 Settings 的 readLPw 函数去解析 packages.xml 和 packages-backup.xml 保存的安装列表信息，并把解析的 pakcages 信息添加到相应的数据结构中，这里我们先假设这是第一次开机，所有 packages.xml 和 packages-backup.xml 文件都还不存在。所以 Settings 的 readLPw 函数会直接返回。接着把 boot class path 里面的文件添加到 alreadyDexOpted 这个 HashSet 中，因为它们在 zygote 启动时已经进过 Dex 优化了。接着扫描 mSharedLibraries 中的文件，这些文件是在解析 platfrom.xml 中的 library tag 添加进来的，如果它们需要做 dex 优化，则调用 Installd 的的 dexopt 方法，关于 installd 的调用流程，我们后面在安装 apk 的时候再来分析。接着来看 PMS 的构造函数：

```

File frameworkDir = new
File(Environment.getRootDirectory(), "framework");

```

```
alreadyDexOpted.add(frameworkDir.getPath() +  
"/framework-res.apk");  
  
alreadyDexOpted.add(frameworkDir.getPath() +  
"/core-libart.jar");  
  
String[] frameworkFiles = frameworkDir.list();  
  
if (frameworkFiles != null) {  
  
    for (int i=0; i<frameworkFiles.length; i++) {  
  
        File libPath = new File(frameworkDir,  
frameworkFiles[i]);  
  
        String path = libPath.getPath();  
  
        if (alreadyDexOpted.contains(path)) {  
  
            continue;  
  
        }  
  
        if (!path.endsWith(".apk") && !path.endsWith(".jar"))  
    {  
  
        continue;  
  
    }  
  
    try {  
  
        if (dalvik.system.DexFile.isDexOptNeeded(path))  
    {
```

```
        mInstaller.dexopt(path,  
Process.SYSTEM_UID, true);  
  
        didDexOpt = true;  
  
    }  
  
} catch (FileNotFoundException e) {  
  
    Slog.w(TAG, "Jar not found: " + path);  
  
} catch (IOException e) {  
  
    Slog.w(TAG, "Exception reading jar: " + path, e);  
  
}  
  
}  
  
}  
  
if (didDexOpt) {  
  
    File dalvikCacheDir = new File(dataDir, "dalvik-cache");  
  
    String[] files = dalvikCacheDir.list();  
  
    if (files != null) {  
  
        for (int i=0; i<files.length; i++) {  
  
            String fn = files[i];  
  
            if (fn.startsWith("data@app@"))  
                || fn.startsWith("data@app-private@"))  
  
    {
```

```

        Slog.i(TAG, "Pruning dalvik file: " + fn);

        (new File(dalvikCacheDir, fn)).delete();

    }

}

}

```

这里扫描所有的/system/framework 下面除 framework-res 以外的 apk 和 jar 包(因为 framework-res 只有 resource 文件), 然后依次对它们做 Dex 优化。在上面如果有对文件做过 Dex 优化, 就要去删除 dalvik-cache 下面所有的 dex 文件, 以防止 cache 文件和现在的文件不相符。接着来看 PMS 的构造函数:

```

mFrameworkInstallObserver = new AppDirObserver(
    frameworkDir.getPath(), OBSERVER_EVENTS, true, false);
mFrameworkInstallObserver.startWatching();
scanDirLI(frameworkDir, PackageParserPARSE_IS_SYSTEM
    | PackageParserPARSE_IS_SYSTEM_DIR
    | PackageParserPARSE_IS_PRIVILEGED,
    scanMode | SCAN_NO_DEX, 0);

// Collected privileged system packages.
File privilegedAppDir = new File(Environment.getRootDirectory(),
"priv-app");
mPrivilegedInstallObserver = new AppDirObserver(
    privilegedAppDir.getPath(), OBSERVER_EVENTS, true, true);
mPrivilegedInstallObserver.startWatching();
scanDirLI(privilegedAppDir, PackageParserPARSE_IS_SYSTEM
    | PackageParserPARSE_IS_SYSTEM_DIR
    | PackageParserPARSE_IS_PRIVILEGED, scanMode, 0);

// Collect ordinary system packages.
File systemAppDir = new File(Environment.getRootDirectory(), "app");
mSystemInstallObserver = new AppDirObserver(
    systemAppDir.getPath(), OBSERVER_EVENTS, true, false);
mSystemInstallObserver.startWatching();
scanDirLI(systemAppDir, PackageParserPARSE_IS_SYSTEM
    | PackageParserPARSE_IS_SYSTEM_DIR, scanMode, 0);

// Collect all vendor packages.
File vendorAppDir = new File("/vendor/app");

```

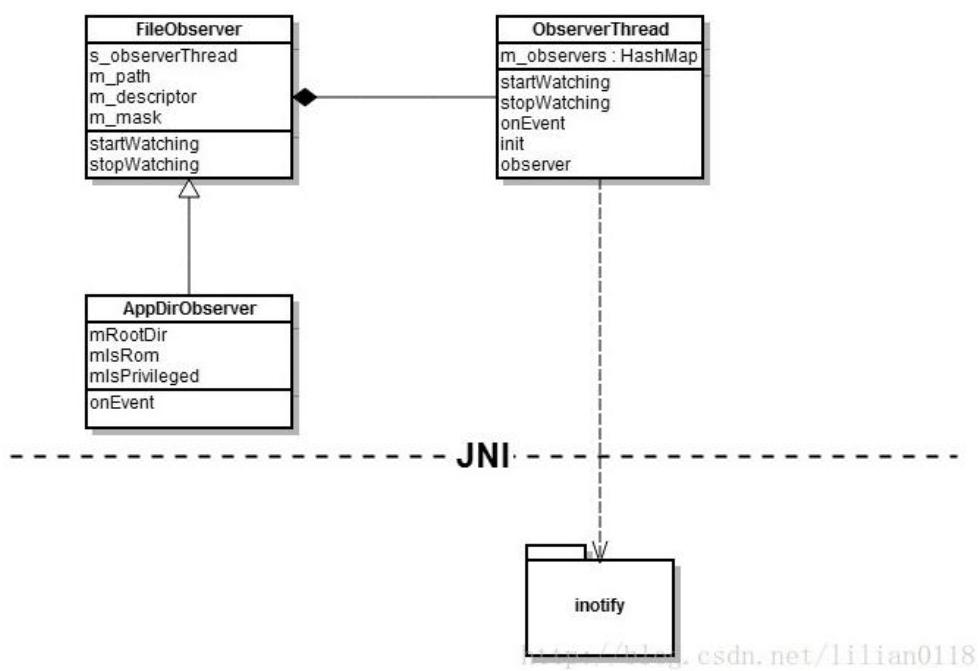
```

mVendorInstallObserver = new AppDirObserver(
    vendorAppDir.getPath(), OBSERVER_EVENTS, true, false);
mVendorInstallObserver.startWatching();
scanDirLI(vendorAppDir, PackageParserPARSE_IS_SYSTEM
    | PackageParserPARSE_IS_SYSTEM_DIR, scanMode, 0);

if (DEBUG_UPGRADE) Log.v(TAG, "Running installd update commands");
mInstaller.moveFiles();

```

这里首先会/system/framework、/system/priv-app、/system/app、/vendor/app 四个目录建立 AppDirObserver 去监听它们的 add、delete 等操作，AppDirObserver 是继承于 FileObserver，它的底层是通过 linux 内核的 inotify 机制实现的。接着调用 scanDirLI 去扫描上面的四个目录。我们来看一下 AppDirObserver 的架构：



接着来看 `scanDirLI` 的代码：

```

private void scanDirLI(File dir, int flags, int scanMode, long currentTime) {
    String[] files = dir.list();

    int i;
    for (i=0; i<files.length; i++) {
        File file = new File(dir, files[i]);
        if (!isPackageFilename(files[i])) {

            continue;
        }
        PackageParser.Package pkg = scanPackageLI(file,

```

```

        flags|PackageParserPARSE_MUST_BE_APK, scanMode,
currentTime, null);

        if (pkg == null && (flags & PackageParserPARSE_IS_SYSTEM) == 0 &&
            mLastScanError ==
PackageManager.INSTALL_FAILED_INVALID_APK) {

            Slog.w(TAG, "Cleaning up failed install of " + file);
            file.delete();
        }
    }
}

scanDirLI 调用 scanPackageLI 依次扫描并解析上面四个目录的目录下所有的 apk 文件:
private PackageParser.Package scanPackageLI(File scanFile,
    int parseFlags, int scanMode, long currentTime, UserHandle user) {
    mLastScanError = PackageManager.INSTALL_SUCCEEDED;
    String scanPath = scanFile.getPath();

    parseFlags |= mDefParseFlags;
    PackageParser pp = new PackageParser(scanPath);

    //首先解析出一个 Package 对象
    final PackageParser.Package pkg = pp.parsePackage(scanFile,
        scanPath, mMetrics, parseFlags);

    PackageSetting ps = null;
    PackageSetting updatedPkg;

    synchronized (mPackages) {

        String oldName = mSettings.mRenamedPackages.get(pkg.packageName);
        if (pkg.mOriginalPackages != null &&
pkg.mOriginalPackages.contains(oldName)) {

            ps = mSettings.peekPackageLPr(oldName);
        }
        if (ps == null) {
            ps = mSettings.peekPackageLPr(pkg.packageName);
        }

        updatedPkg = mSettings.getDisabledSystemPkgLPr(ps != null ? ps.name :
pkg.packageName);
    }
}

```

```
        if (DEBUG_INSTALL && updatedPkg != null) Slog.d(TAG, "updatedPkg = " +  
updatedPkg);  
    }  
  
    if (updatedPkg != null && (parseFlags&PackageParserPARSE_IS_SYSTEM) !=  
0) {  
        //与 update app 相关的  
    }  
  
    if (updatedPkg != null) {  
        parseFlags |= PackageParserPARSE_IS_SYSTEM;  
    }  
  
    if (!collectCertificatesLI(pp, ps, pkg, scanFile, parseFlags)) {  
        Slog.w(TAG, "Failed verifying certificates for package:" +  
pkg.packageName);  
        return null;  
    }  
  
    //处理 system 与非 system 的 app 同名的问题  
    boolean shouldHideSystemApp = false;  
    if (updatedPkg == null && ps != null  
        && (parseFlags & PackageParserPARSE_IS_SYSTEM_DIR) != 0  
&& !isSystemApp(ps)) {  
  
        if (compareSignatures(ps.signatures.mSignatures, pkg.mSignatures)  
            != PackageManager.SIGNATURE_MATCH) {  
            if (DEBUG_INSTALL) Slog.d(TAG, "Signature mismatch!");  
            deletePackageLI(pkg.packageName, null, true, null, null, 0, null, false);  
            ps = null;  
        } else {  
  
            if (pkg.mVersionCode < ps.versionCode) {  
                shouldHideSystemApp = true;  
            } else {  
                InstallArgs args =  
createInstallArgs(packageFlagsToInstallFlags(ps),  
                           ps.codePathString, ps.resourcePathString,  
                           ps.nativeLibraryPathString);  
                synchronized (mInstallLock) {  
                    args.cleanUpResourcesLI();  
                }  
            }  
        }  
    }  
}
```

```

    }

    if ((parseFlags & PackageParserPARSE_IS_SYSTEM_DIR) == 0) {
        if (ps != null && !ps.codePath.equals(ps.resourcePath)) {
            parseFlags |= PackageParserPARSE_FORWARD_LOCK;
        }
    }

    String codePath = null;
    String resPath = null;
    if ((parseFlags & PackageParserPARSE_FORWARD_LOCK) != 0) {
        if (ps != null && ps.resourcePathString != null) {
            resPath = ps.resourcePathString;
        } else {

        }
    } else {
        resPath = pkg.mScanPath;
    }

    codePath = pkg.mScanPath;
    setApplicationInfoPaths(pkg, codePath, resPath);

    //
    PackageParser.Package scannedPkg = scanPackageLI(pkg, parseFlags,
scanMode
        | SCAN_UPDATE_SIGNATURE, currentTime, user);

    if (shouldHideSystemApp) {
        synchronized (mPackages) {
            grantPermissionsLPw(pkg, true);
            mSettings.disableSystemPackageLPw(pkg.packageName);
        }
    }

    return scannedPkg;
}

```

scanPackageLI 首先调用 PackageParser 的 parsePackage 去解析扫描的文件，注意这里有两个 parsePackage 函数，但它们的参数不同，我们来看以 File 为第一个参数的 parsePackage 方法：

```
public Package parsePackage(File sourceFile, String  
destCodePath,  
    DisplayMetrics metrics, int flags) {  
    mParseError = PackageManager.INSTALL_SUCCEEDED;  
  
    mArchiveSourcePath = sourceFile.getPath();  
  
    XmlResourceParser parser = null;  
    AssetManager assmgr = null;  
    Resources res = null;  
    boolean assetError = true;  
    try {  
        assmgr = new AssetManager();  
        int cookie = assmgr.addAssetPath(mArchiveSourcePath);  
        if (cookie != 0) {  
            res = new Resources(assmgr, metrics, null);  
            assmgr.setConfiguration(0, 0, null, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0,  
                Build.VERSION.RESOURCES_SDK_INT);  
            parser = assmgr.openXmlResourceParser(cookie,  
ANDROID_MANIFEST_FILENAME);  
            assetError = false;  
        }  
    } catch (Exception e) {  
        Log.e("ManifestParser", "Error parsing manifest: " +  
e.getMessage());  
        assetError = true;  
    }  
    if (assetError) {  
        mParseError = PackageManagerPARSE_FAILED;  
    }  
}
```

```
        } else {

            Slog.w(TAG, "Failed adding asset
path:" +mArchiveSourcePath);

        }

    } catch (Exception e) {

        Slog.w(TAG, "Unable to read AndroidManifest.xml of "
+ mArchiveSourcePath, e);

    }

    if (assetError) {

        if (assmgr != null) assmgr.close();

        mParseError =
PackageManager.INSTALL_PARSE_FAILED_BAD_MANIFEST;

        return null;

    }

    String[] errorText = new String[1];

    Package pkg = null;

    Exception errorException = null;

    try {

        // XXXX todo: need to figure out correct configuration.

        pkg = parsePackage(res, parser, flags, errorText);

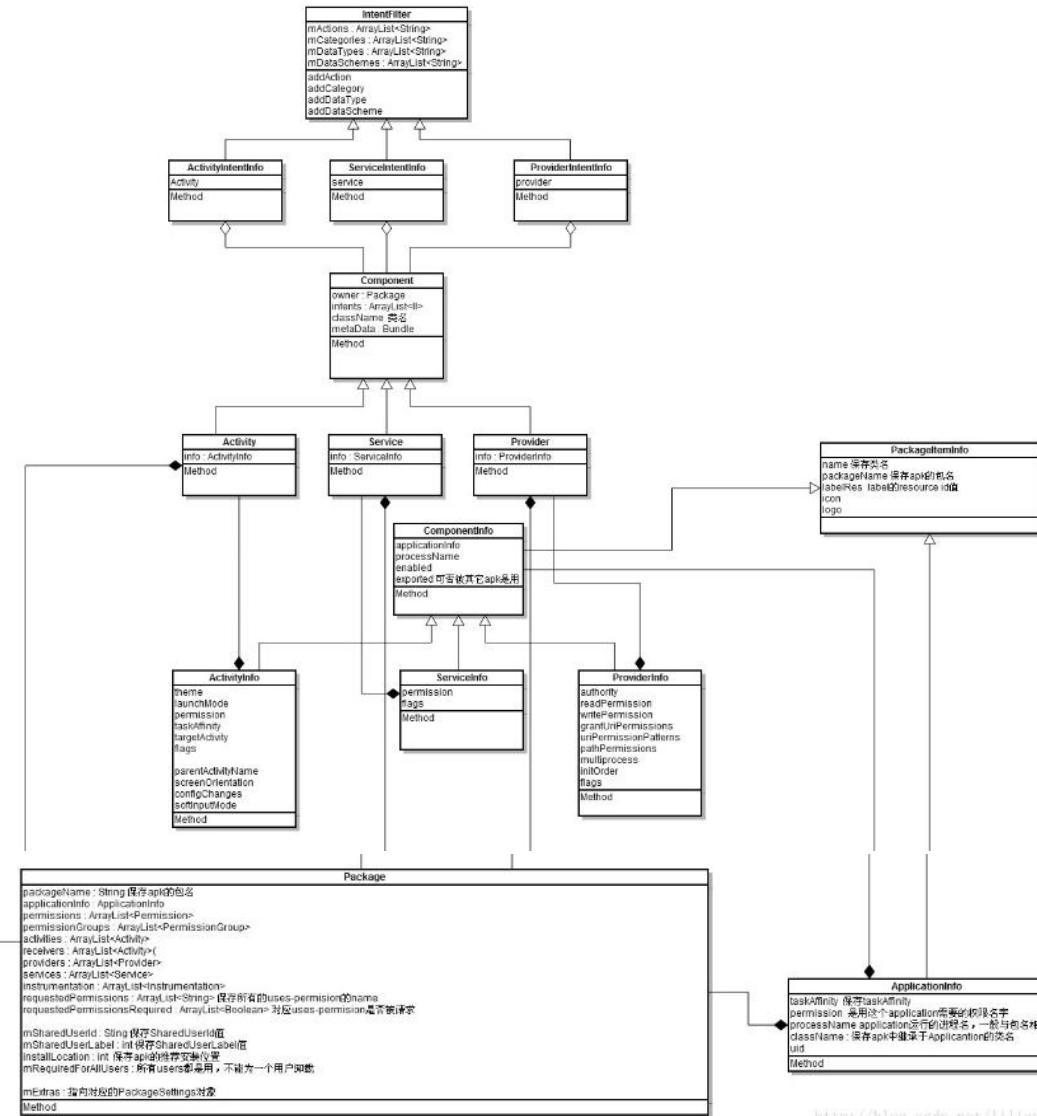
    } catch (Exception e) {

        errorException = e;

    }
```

```
    mParseError =  
  
PackageManager.INSTALL_PARSE_FAILED_UNEXPECTED_EXCEPTION;  
  
}  
  
  
    parser.close();  
  
    assmgr.close();  
  
  
    pkg.mPath = destCodePath;  
  
    pkg.mScanPath = mArchiveSourcePath;  
  
    pkg.mSignatures = null;  
  
  
    return pkg;  
  
}
```

首先从 apk 文件中打开 AndroidManifest.xml 文件 然后调用以 Resources 为第一个参数的 parsePackage 方法，这个函数比较长，主要就是解析 AndroidManifest.xml 文件，建立一个 Package 对象，大概类图如下。最后设置 Package 对象的 mPath 和 mScanPath 为当前 APK 所在的全路径名。



我们以 Mms 这个应用的 Manifest 文件来看分析解析后的结果 ,首先来看 Mms 的 AndroidManifest.xml 文件(这里只截取了一部分)

```
<manifest
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
        package="com.android.mms">
```

```
            <original-package android:name="com.android.mms" />
```

```
            <uses-permission
```

```
                android:name="android.permission.RECEIVE_MMS" />
```

```
<uses-permission android:name="android.permission.SEND_SMS"

/>

<!-- System apps can access the receiver through intent-->

<permission

    android:name="android.permission.MMS_SEND_OUTBOX_MSG"

        android:protectionLevel="signatureOrSystem"

        android:label="@string/label_mms_send_outbox_msg"

        android:description="@string/desc_mms_send_outbox_msg"/>

<application android:name="MmsApp"

    android:label="@string/app_label"

    android:icon="@mipmap/ic_launcher_smsmms"

    android:taskAffinity="android.task.mms"

    android:allowTaskReparenting="true">

    <activity android:name=".ui.ConversationList"

        android:label="@string/app_label"

        android:configChanges="orientation|screenSize|keyboardHidden"

        android:theme="@style/MmsHoloTheme"

        android:uiOptions="splitActionBarWhenNarrow"

        android:launchMode="singleTop">

        <intent-filter>
```

```
<action android:name="android.intent.action.MAIN" />

<category
    android:name="android.intent.category.LAUNCHER" />

    <category
        android:name="android.intent.category.DEFAULT" />

        <category
            android:name="android.intent.category.APP_MESSAGING" />

            </intent-filter>

            <intent-filter>

                <action android:name="android.intent.action.MAIN" />

                <category
                    android:name="android.intent.category.DEFAULT" />

                    <data
                        android:mimeType="vnd.android.cursor.dir/mms" />

                        </intent-filter>

                        <intent-filter>

                            <action android:name="android.intent.action.MAIN" />

                            <category
                                android:name="android.intent.category.DEFAULT" />

                                <data android:mimeType="vnd.android-dir/mms-sms"
                                />

                                </intent-filter>
```

```
</activity>

<activity android:name=".ui.ComposeMessageActivity"

        android:configChanges="orientation|screenSize|keyboardHidden"
        android:windowSoftInputMode="stateHidden|adjustResize"
        android:theme="@style/MmsHoloTheme"
        android:parentActivityName=".ui.ConversationList"
        android:launchMode="singleTop" >

    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category
            android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android-dir/mms-sms"
    />

    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <action android:name="android.intent.action.SENDTO"
    />

    <category
        android:name="android.intent.category.DEFAULT" />
```

```
<category  
    android:name="android.intent.category.BROWSABLE" />  
  
    <data android:scheme="sms" />  
  
    <data android:scheme="smsto" />  
  
  </intent-filter>  
  
  <intent-filter>  
  
    <action android:name="android.intent.action.VIEW" />  
  
    <action android:name="android.intent.action.SENDTO"  
/>  
  
    <category  
        android:name="android.intent.category.DEFAULT" />  
  
      <category  
          android:name="android.intent.category.BROWSABLE" />  
  
          <data android:scheme="mms" />  
  
          <data android:scheme="mmsto" />  
  
        </intent-filter>  
  
      <intent-filter>  
  
        <action android:name="android.intent.action.SEND" />  
  
        <category  
            android:name="android.intent.category.DEFAULT" />  
  
          <data android:mimeType="image/*" />  
  
        </intent-filter>
```

```
<intent-filter>

    <action android:name="android.intent.action.SEND" />

    <category
        android:name="android.intent.category.DEFAULT" />

    <data android:mimeType="text/plain" />

</intent-filter>

<intent-filter>

    <action
        android:name="android.intent.action.SEND_MULTIPLE" />

    <category
        android:name="android.intent.category.DEFAULT" />

    <data android:mimeType="image/*" />

</intent-filter>

</activity>

<receiver android:name=".transaction.PushReceiver"
          android:permission="android.permission.BROADCAST_WAP_PUSH">

    <intent-filter>

        <action
            android:name="android.provider.Telephony.WAP_PUSH_DELIVER" />

        <data
            android:mimeType="application/vnd.wap.mms-message" />

    </intent-filter>

```

```
</intent-filter>

</receiver>

<receiver android:name=".transaction.SmsReceiver">

    <intent-filter>

        <action

            android:name="android.intent.action.BOOT_COMPLETED" />

        </intent-filter>

        <intent-filter>

            <action

                android:name="com.android.mms.transaction.MESSAGE_SENT" />

            <!-- TODO Do a better data match here. -->

            <data android:scheme="content" />

        </intent-filter>

        <intent-filter>

            <action

                android:name="android.intent.action.SEND_MESSAGE" />

            </intent-filter>

        </receiver>

        <provider android:name="SuggestionsProvider"

            android:exported="true"

            android:readPermission="android.permission.READ_SMS"
```

```
    android:authorities="com.android.mms.SuggestionsProvider" >

        <path-permission

            android:pathPrefix="/search_suggest_query"

    android:readPermission="android.permission.GLOBAL_SEARCH" />

        <path-permission

            android:pathPrefix="/search_suggest_shortcut"

    android:readPermission="android.permission.GLOBAL_SEARCH" />

    </provider>

    <service android:name=".ui.NoConfirmationSendService"

        android:permission="android.permission.SEND_RESPOND_VIA_MESSAGE"
        android:exported="true" >

        <intent-filter>

            <action
                android:name="android.intent.action.RESPOND_VIA_MESSAGE" />

            <category
                android:name="android.intent.category.DEFAULT" />

            <data android:scheme="sms" />
```

```

<data android:scheme="smsto" />

</intent-filter>

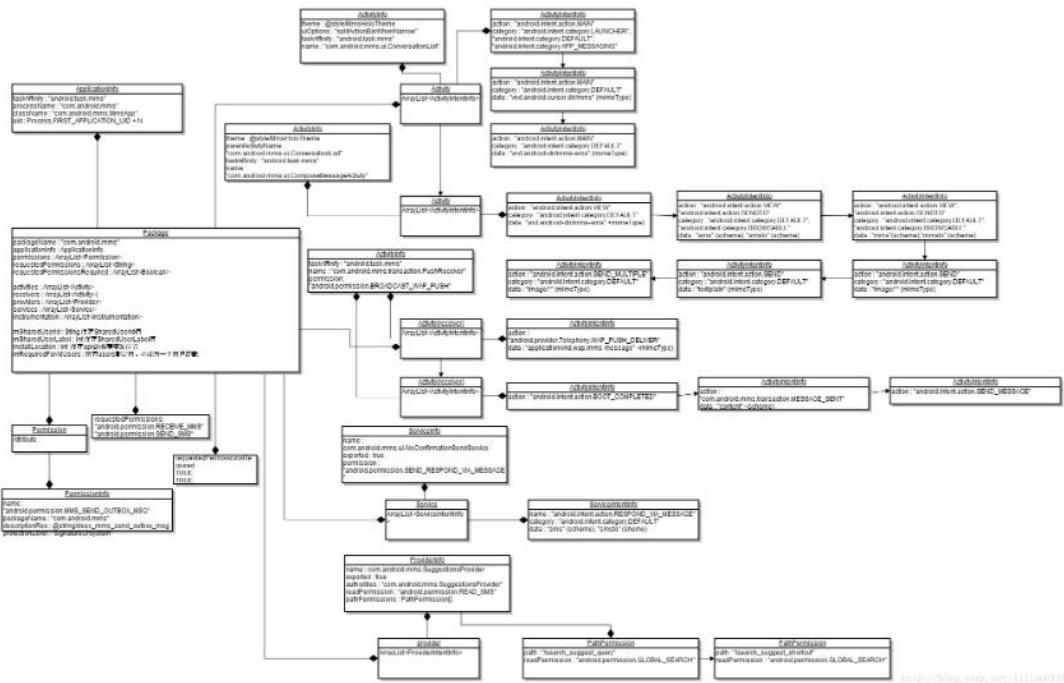
</service>

</application>

</manifest>

```

上面的 AndroidManifest.xml 文件中定义了 2 个 Activity , 2 个 receiver , 1 个 service 和 1 个 provider , 我们来看进过 parsePackage 得到的 Package 对象如下 :



接着回到 scanPackageLI 方法 , 解析完 AndroidManifest.xml 文件后 , 再来检查是否是更新的 APK , 如果更新的 APK 版本比以前的版本还有低 , 则直接返回 ; 如果更新的 APK 版本比以前的版本高 , 则去删除之前的 APK 以及 resource 文件。若不是更新 APK , 并且当前 package 是系统 app , 但之前安装了非系统的 app , 这里首先比较签名 , 如果签名不一致 , 则直接删除当前 package ; 若签名文件一致 , 则首先比较当前 package 和之前的版本号 , 如果当前版本号比较新 ,

则直接删除之前的 APK 以及 resource 文件。最后调用 scanPackageLI 方法让把当前 package 的信息归入到 PMS 中的数据结构：

```
private PackageParser.Package  
scanPackageLI(PackageParser.Package pkg,  
    int parseFlags, int scanMode, long currentTime, UserHandle  
user) {  
  
    File scanFile = new File(pkg.mScanPath);  
    mScanningPath = scanFile;  
  
    if ((parseFlags&PackageParserPARSE_IS_SYSTEM) != 0) {  
        pkg.applicationInfo.flags |= ApplicationInfo.FLAG_SYSTEM;  
    }  
  
    if ((parseFlags&PackageParserPARSE_IS_PRIVILEGED) != 0) {  
        pkg.applicationInfo.flags |=  
ApplicationInfo.FLAG_PRIVILEGED;  
    }  
  
    if (mCustomResolverComponentName != null &&  
mCustomResolverComponentName.getPackageName().equals(pkg.pack  
ageName)) {
```

```
        setUpCustomResolverActivity(pkg);

    }

    if (pkg.packageName.equals("android")) {

        synchronized (mPackages) {

            if (mAndroidApplication != null) {

                mLastScanError =

PackageManager.INSTALL_FAILED_DUPLICATE_PACKAGE;

                return null;

            }

            mPlatformPackage = pkg;

            pkg.mVersionCode = mSdkVersion;

            mAndroidApplication = pkg.applicationInfo;

            if (!mResolverReplaced) {

                mResolveActivity.applicationInfo =

mAndroidApplication;

                mResolveActivity.name =

ResolverActivity.class.getName();

                mResolveActivity.packageName =

mAndroidApplication.packageName;
```

```
        mResolveActivity.processName = "system:ui";  
  
        mResolveActivity.launchMode =  
  
ActivityInfo.LAUNCH_MULTIPLE;  
  
        mResolveActivity.flags =  
  
ActivityInfo.FLAG_EXCLUDE_FROM_RECENTS;  
  
        mResolveActivity.theme =  
  
com.android.internal.R.style.Theme_Holo_Dialog_Alert;  
  
        mResolveActivity.exported = true;  
  
        mResolveActivity.enabled = true;  
  
        mResolveInfo.activityInfo = mResolveActivity;  
  
        mResolveInfo.priority = 0;  
  
        mResolveInfo.preferredOrder = 0;  
  
        mResolveInfo.match = 0;  
  
        mResolveComponentName = new  
ComponentName(  
  
                mAndroidApplication.packageName,  
  
mResolveActivity.name);  
  
        }  
  
    }  
  
}
```

```
File destCodeFile = new File(pkg.applicationInfo.sourceDir);
```

```
File destResourceFile = new  
File(pkg.applicationInfo.publicSourceDir);  
  
  
SharedUserSetting suid = null;  
PackageSetting pkgSetting = null;  
  
  
if (!isSystemApp(pkg)) {  
    // Only system apps can use these features.  
    pkg.mOriginalPackages = null;  
    pkg.mRealPackage = null;  
    pkg.mAdoptPermissions = null;  
}  
  
  
// writer  
synchronized (mPackages) {  
    if ((parseFlags&PackageParserPARSE_IS_SYSTEM_DIR) == 0)  
    {  
        if (!updateSharedLibrariesLPw(pkg, null)) {  
            return null;  
        }  
    }  
}
```

```
    if (pkg.mSharedUserId != null) {

        uid =
mSettings.getSharedUserLPw(pkg.mSharedUserId, 0, true);

        if (uid == null) {

            Slog.w(TAG, "Creating application package " +
pkg.packageName

                + " for shared user failed");

            mLastScanError =
PackageManager.INSTALL_FAILED_INSUFFICIENT_STORAGE;

            return null;
        }
    }

    if (DEBUG_PACKAGE_SCANNING) {

        if ((parseFlags & PackageParserPARSE_CHATTY) !=

0)

            Log.d(TAG, "Shared UserID " +

pkg.mSharedUserId + " (uid=" + uid.userId

                + "): packages=" + uid.packages);

    }
}
```

这里的 mCustomResolverComponentName 默认是空，采用 framework 是本身的 ResolverActivity 去解析 intent。mAndroidApplication 在 Android 系统中只有一个这样的 application，就是 framework-res.apk，它的

packageName 是"android"。然后在 mResolveActivity 和 mResolveInfo 保存 ResolverActivity 的信息，ResolverActivity 用于在启动 Activity 的时候，如果有多个 activity 符合条件，弹出对话框给用户选择，这部分我们在以后分析 AcitivityManagerService 的时候再来分析。如果在 Manifest 中指定了 ShareUserId，则首先获取一个关联的 SharedUserSetting 对象：

```
SharedUserSetting getSharedUserLPw(String name,
        int pkgFlags, boolean create) {
    SharedUserSetting s = mSharedUsers.get(name);
    if (s == null) {
        if (!create) {
            return null;
        }
        s = new SharedUserSetting(name, pkgFlags);
        s.userId = newUserIdLPw(s);
        Log.i(PackageManagerService.TAG, "New shared user " + name + ": id=" + s.userId);
        // < 0 means we couldn't assign a userid; fall out and return
        // s, which is currently null
        if (s.userId >= 0) {
            mSharedUsers.put(name, s);
        }
    }
    return s;
}
```

在开始 PMS 的构造函数里面我们知道，系统会首先添加一系列的 system 的 user id 到 mSharedUsers，所以如果能够从 mSharedUsers 获得到就直接返回；如果不能，则首先构造一个 SharedUserSetting，并指派一个没有使用的 APPLICATION UID，当然 APPLICATION UID 的值是在 FIRST_APPLICATION_UID 到 LAST_APPLICATION_UID 之间。最后把创建的 SharedUserSetting 添加到 mSharedUsers 和 mUserIds 数组当中。接着来看 scanPackageLI 函数：

```
PackageSetting origPackage = null;
String realName = null;
if (pkg.mOriginalPackages != null) {
    //关于应用命名和更新的代码
}

pkgSetting = mSettings.getPackageLPw(pkg, origPackage, realName, suid,
destCodeFile,
        destResourceFile, pkg.applicationInfo.nativeLibraryDir,
        pkg.applicationInfo.flags, user, false);
if (pkgSetting == null) {
    Slog.w(TAG, "Creating application package " + pkg.packageName + "
failed");
```

```

        mLastScanError =
PackageManager.INSTALL_FAILED_INSUFFICIENT_STORAGE;
        return null;
    }

    if (pkgSetting.origPackage != null) {
        pkg.setPackageName(origPackage.name);

        String msg = "New package " + pkgSetting.realName
            + " renamed to replace old package " + pkgSetting.name;
        reportSettingsProblem(Log.WARN, msg);

        mTransferredPackages.add(origPackage.name);

        pkgSetting.origPackage = null;
    }

    if (realName != null) {
        mTransferredPackages.add(pkg.packageName);
    }

    if (mSettings.isDisabledSystemPackageLPr(pkg.packageName)) {
        pkg.applicationInfo.flags |=
ApplicationInfo.FLAG_UPDATED_SYSTEM_APP;
    }

    if (mFoundPolicyFile) {
        SELinuxMMAC.assignSeinfoValue(pkg);
    }

    pkg.applicationInfo.uid = pkgSetting.appId;
    pkg.mExtras = pkgSetting;

    if (!verifySignaturesLP(pkgSetting, pkg)) {
        if ((parseFlags&PackageParser.PARSE_IS_SYSTEM_DIR) == 0) {
            return null;
        }
        pkgSetting.signatures.mSignatures = pkg.mSignatures;
        if (pkgSetting.sharedUser != null) {
            if
(compareSignatures(pkgSetting.sharedUser.signatures.mSignatures,
                    pkg.mSignatures) !=
PackageManager.SIGNATURE_MATCH) {

```

```
        Log.w(TAG, "Signature mismatch for shared user : " +
pkgSetting.sharedUser);
        mLastScanError =
PackageManager.INSTALL_PARSE_FAILED_INCONSISTENT_CERTIFICATES;
        return null;
    }
}
String msg = "System package " + pkg.packageName
+ " signature changed; retaining data.";
reportSettingsProblem(Log.WARN, msg);
}
```

关于应用程序改名和更新的代码我们这里先忽略，首先来看构造

PackageSetting 的方法：

```
PackageSetting getPackageLPw(PackageParser.Package pkg,
PackageSetting origPackage,
String realName, SharedUserSetting sharedUser, File
codePath, File resourcePath,
String nativeLibraryPathString, int pkgFlags, UserHandle
user, boolean add) {

    final String name = pkg.packageName;

    PackageSetting p = getPackageLPw(name, origPackage,
realName, sharedUser, codePath,
resourcePath, nativeLibraryPathString,
pkg.mVersionCode, pkgFlags,
user, add, true /* allowInstall */);

    return p;
}
```

```
private PackageSetting getPackageLPw(String name, PackageSetting origPackage,  
        String realName, SharedUserSetting sharedUser, File codePath, File resourcePath,  
        String nativeLibraryPathString, int vc, int pkgFlags,  
        UserHandle installUser, boolean add, boolean allowInstall) {  
  
    PackageSetting p = mPackages.get(name);  
  
    if (p != null) {  
        //更新 apk 相关  
    }  
  
    if (p == null) {  
        if (origPackage != null) {  
            //更新 apk 相关  
        } else {  
            p = new PackageSetting(name, realName, codePath,  
                    resourcePath,  
                    nativeLibraryPathString, vc, pkgFlags);  
            p.setTimeStamp(codePath.lastModified());  
            p.sharedUser = sharedUser;  
            if ((pkgFlags&ApplicationInfo.FLAG_SYSTEM) == 0) {  
                List<UserInfo> users = getAllUsers();  
                if (users != null && allowInstall) {  
                    UserHandle userHandle = UserHandle.of(users.get(0).uid);  
                    if (userHandle != null) {  
                        UserHandle userHandle2 = UserHandle.of(origPackage.uid);  
                        if (userHandle2 != null) {  
                            UserHandle userHandle3 = UserHandle.of(installUser.getIdentifier());  
                            if (userHandle3 != null) {  
                                if (userHandle.equals(userHandle2)) {  
                                    userHandle.setInstallUser(installUser);  
                                }  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```
//多用户的部分

}

}

if (sharedUser != null) {

    p.appId = sharedUser.userId;

} else {

    //更新系统 apk 相关

}

if (add) {

    // Finish adding new package by adding it and updating
shared

    // user preferences

    addPackageSettingLPw(p, name, sharedUser);

}

} else {

    //多用户的部分

}

return p;
}
```

关于 PackageSetting 结构可以看文章最开始的类图。在新建一个 PackageSetting 对象后，首先将得到的系统的 uid 值赋给 applicationInfo.uid ，这就是当前 APK 以后运行时

的 UID 了。然后就做数字签名验证，这里主要是对于更新 APK 来做验证。在做完数字签名验证后，还需要检查当前 APK 是否提供 providers 与系统已有的 providers 冲突，如果冲突，则提示安装失败。接着来看 scanPackageLI 函数：

```
synchronized (mPackages) {
    // We don't expect installation to fail beyond this point,
    if ((scanMode&SCAN_MONITOR) != 0) {
        mAppDirs.put(pkg.mPath, pkg);
    }
    // Add the new setting to mSettings
    mSettings.insertPackageSettingLPw(pkgSetting, pkg);
    // Add the new setting to mPackages
    mPackages.put(pkg.applicationInfo.packageName, pkg);
    // Make sure we don't accidentally delete its data.
    final Iterator<PackageCleanItem> iter =
mSettings.mPackagesToBeCleaned.iterator();
    while (iter.hasNext()) {
        PackageCleanItem item = iter.next();
        if (pkgName.equals(item.packageName)) {
            iter.remove();
        }
    }

    // Take care of first install / last update times.
    if (currentTime != 0) {
        if (pkgSetting.firstInstallTime == 0) {
            pkgSetting.firstInstallTime = pkgSetting.lastUpdateTime =
currentTime;
        } else if ((scanMode&SCAN_UPDATE_TIME) != 0) {
            pkgSetting.lastUpdateTime = currentTime;
        }
    } else if (pkgSetting.firstInstallTime == 0) {
        // We need *something*. Take time stamp of the file.
        pkgSetting.firstInstallTime = pkgSetting.lastUpdateTime =
scanFileTime;
    } else if ((parseFlags&PackageParserPARSE_IS_SYSTEM_DIR) != 0)
{
    if (scanFileTime != pkgSetting.timeStamp) {
        // A package on the system image has changed; consider this
        // to be an update.
        pkgSetting.lastUpdateTime = scanFileTime;
    }
}

// Add the package's KeySets to the global KeySetManager
```

```

KeySetManager ksm = mSettings.mKeySetManager;
try {
    ksm.addSigningKeySetToPackage(pkg.packageName,
pkg.mSigningKeys);
    if (pkg.mKeySetMapping != null) {
        for (Map.Entry<String, Set<PublicKey>> entry :
pkg.mKeySetMapping.entrySet()) {
            if (entry.getValue() != null) {

ksm.addDefinedKeySetToPackage(pkg.packageName,
                                entry.getValue(), entry.getKey());
        }
    }
}
} catch (NullPointerException e) {
    Slog.e(TAG, "Could not add KeySet to " + pkg.packageName, e);
} catch (IllegalArgumentException e) {
    Slog.e(TAG, "Could not add KeySet to malformed package" +
pkg.packageName, e);
}
}

```

首先调用 `Settings` 的 `insertPackageSettingLPw` 将 `pkgSetting` 对象加入到 `Settings` 中的 `mPackages` 这个 `HashMap` 中。在 `insertPackageSettingLPw` 方法中，首先将 `Package` 中的一些信息赋予给 `PackageSetting`，然后调用 `addPackageSettingLPw` 方法将 `PackageSetting` 对象添加到 `mPackages` 中，并将 `PackageSetting` 加入到 `SharedUserSetting` 中的 `packages` 这个 `HashSet` 中。接着将 `pkg` 对象加入到 `PMS` 的 `mPackages` 这个 `HashMap` 中，保存在 `mPackages` 中信息会被后面很多地方使用到。最后对 `apk` 的安装或者更新时间做相应的更新。接着来看 `scanPackageLI` 函数：

```

int N = pkg.providers.size();
StringBuilder r = null;
int i;
for (i=0; i<N; i++) {
    PackageParser.Provider p = pkg.providers.get(i);
    p.info.processName =
fixProcessName(pkg.applicationInfo.processName,
                p.info.processName, pkg.applicationInfo.uid);
    mProviders.addProvider(p);
    p.syncable = p.info.isSyncable;
    if (p.info.authority != null) {
        String names[] = p.info.authority.split(";");
        p.info.authority = null;
        for (int j = 0; j < names.length; j++) {
            if (j == 1 && p.syncable) {
                p = new PackageParser.Provider(p);
                p.syncable = false;

```

```

        }
        if (!mProvidersByAuthority.containsKey(names[j])) {
            mProvidersByAuthority.put(names[j], p);
            if (p.info.authority == null) {
                p.info.authority = names[j];
            } else {
                p.info.authority = p.info.authority + ";" + names[j];
            }
        } else {
            PackageParser.Provider other =
            mProvidersByAuthority.get(names[j]);
            }
        }
    }
    if ((parseFlags&PackageParserPARSE_CHATTY) != 0) {
        if (r == null) {
            r = new StringBuilder(256);
        } else {
            r.append(' ');
        }
        r.append(p.info.name);
    }
}
if (r != null) {
    if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, " Providers: " + r);
}

N = pkg.services.size();
r = null;
for (i=0; i<N; i++) {
    PackageParser.Service s = pkg.services.get(i);
    s.info.processName =
fixProcessName(pkg.applicationInfo.processName,
               s.info.processName, pkg.applicationInfo.uid);
mServices.addService(s);
if ((parseFlags&PackageParserPARSE_CHATTY) != 0) {
    if (r == null) {
        r = new StringBuilder(256);
    } else {
        r.append(' ');
    }
    r.append(s.info.name);
}
}

```

```

        if (r != null) {
            if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, " Services: " + r);
        }

        N = pkg.receivers.size();
        r = null;
        for (i=0; i<N; i++) {
            PackageParser.Activity a = pkg.receivers.get(i);
            a.info.processName =
fixProcessName(pkg.applicationInfo.processName,
                a.info.processName, pkg.applicationInfo.uid);
            mReceivers.addActivity(a, "receiver");
            if ((parseFlags&PackageParserPARSE_CHATTY) != 0) {
                if (r == null) {
                    r = new StringBuilder(256);
                } else {
                    r.append(' ');
                }
                r.append(a.info.name);
            }
        }
        if (r != null) {
            if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, " Receivers: " + r);
        }

        N = pkg.activities.size();
        r = null;
        for (i=0; i<N; i++) {
            PackageParser.Activity a = pkg.activities.get(i);
            a.info.processName =
fixProcessName(pkg.applicationInfo.processName,
                a.info.processName, pkg.applicationInfo.uid);
            mActivities.addActivity(a, "activity");
            if ((parseFlags&PackageParserPARSE_CHATTY) != 0) {
                if (r == null) {
                    r = new StringBuilder(256);
                } else {
                    r.append(' ');
                }
                r.append(a.info.name);
            }
        }
        if (r != null) {
            if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, " Activities: " + r);
        }
    }
}

```

```

    }

    N = pkg.permissionGroups.size();
    r = null;
    for (i=0; i<N; i++) {
        PackageParser.PermissionGroup pg = pkg.permissionGroups.get(i);
        PackageParser.PermissionGroup cur =
mPermissionGroups.get(pg.info.name);
        if (cur == null) {
            mPermissionGroups.put(pg.info.name, pg);
            if ((parseFlags&PackageParserPARSE_CHATTY) != 0) {
                if (r == null) {
                    r = new StringBuilder(256);
                } else {
                    r.append(' ');
                }
                r.append(pg.info.name);
            }
        } else {
            Slog.w(TAG, "Permission group " + pg.info.name + " from package
"
+ pg.info.packageName + " ignored: original from "
+ cur.info.packageName);
            if ((parseFlags&PackageParserPARSE_CHATTY) != 0) {
                if (r == null) {
                    r = new StringBuilder(256);
                } else {
                    r.append(' ');
                }
                r.append("DUP:");
                r.append(pg.info.name);
            }
        }
    }
    if (r != null) {
        if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, " Permission Groups:
" + r);
    }

    N = pkg.permissions.size();
    r = null;
    for (i=0; i<N; i++) {
        PackageParser.Permission p = pkg.permissions.get(i);
        HashMap<String, BasePermission> permissionMap =

```

```

    p.tree ? mSettings.mPermissionTrees
        : mSettings.mPermissions;
p.group = mPermissionGroups.get(p.info.group);
if (p.info.group == null || p.group != null) {
    BasePermission bp = permissionMap.get(p.info.name);
    if (bp == null) {
        bp = new BasePermission(p.info.name, p.info.packageName,
            BasePermission.TYPE_NORMAL);
        permissionMap.put(p.info.name, bp);
    }
    if (bp.perm == null) {
        if (bp.sourcePackage != null
            && !bp.sourcePackage.equals(p.info.packageName))
    {
        if (isSystemApp(p.owner)) {
            Slog.i(TAG, "New decl " + p.owner + " of permission
"
                + p.info.name + " is system");
            bp.sourcePackage = null;
        }
    }
    if (bp.sourcePackage == null
        || bp.sourcePackage.equals(p.info.packageName)) {
        BasePermission tree =
findPermissionTreeLP(p.info.name);
        if (tree == null
            ||
tree.sourcePackage.equals(p.info.packageName)) {
            bp.packageSetting = pkgSetting;
            bp.perm = p;
            bp.uid = pkg.applicationInfo.uid;
            if ((parseFlags&PackageParserPARSE_CHATTY) !=
0) {
                if (r == null) {
                    r = new StringBuilder(256);
                } else {
                    r.append(' ');
                }
                r.append(p.info.name);
            }
        } else {
            }
    } else {
        }
    }
}

```

```

        }

    } else if ((parseFlags&PackageParserPARSE_CHATTY) != 0) {
        if (r == null) {
            r = new StringBuilder(256);
        } else {
            r.append(' ');
        }
        r.append("DUP:");
        r.append(p.info.name);
    }

    if (bp.perm == p) {
        bp.protectionLevel = p.info.protectionLevel;
    }
} else {

}

if (r != null) {
    if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, " Permissions: " + r);
}

N = pkg.instrumentation.size();
r = null;
for (i=0; i<N; i++) {
    PackageParser.Instrumentation a = pkg.instrumentation.get(i);
    a.info.packageName = pkg.applicationInfo.packageName;
    a.info.sourceDir = pkg.applicationInfo.sourceDir;
    a.info.publicSourceDir = pkg.applicationInfo.publicSourceDir;
    a.info.dataDir = pkg.applicationInfo.dataDir;
    a.info.nativeLibraryDir = pkg.applicationInfo.nativeLibraryDir;
    mInstrumentation.put(a.getComponentName(), a);
    if ((parseFlags&PackageParserPARSE_CHATTY) != 0) {
        if (r == null) {
            r = new StringBuilder(256);
        } else {
            r.append(' ');
        }
        r.append(a.info.name);
    }
}
if (r != null) {
    if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, " Instrumentation: "
+ r);
}

```

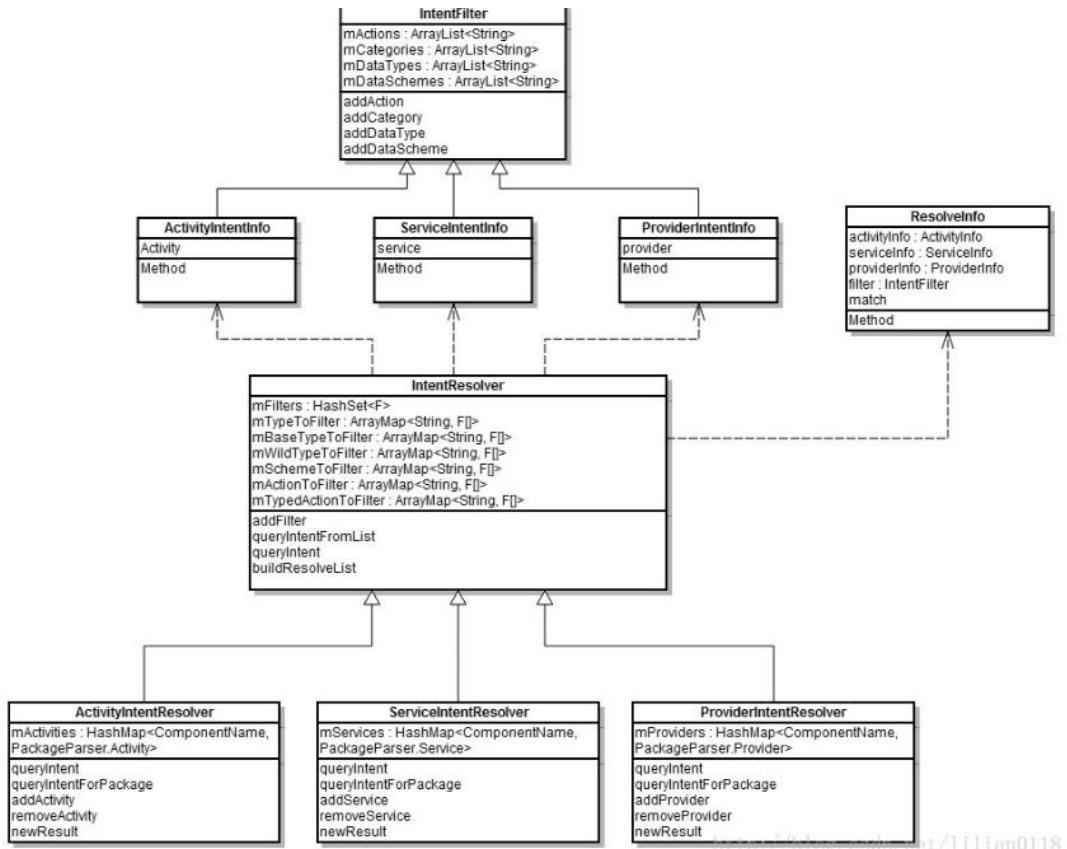
```
        }

        if (pkg.protectedBroadcasts != null) {
            N = pkg.protectedBroadcasts.size();
            for (i=0; i<N; i++) {
                mProtectedBroadcasts.add(pkg.protectedBroadcasts.get(i));
            }
        }

        pkgSetting.setTimeStamp(scanFileTime);
    }

    return pkg;
}
```

上面的代码比较长，但功能却比较简单，就是将前面从 AndroidManifest 里面 Parse 出来的 providers、services、receivers、activities、permissionGroups、permissions 和 instrumentation 添加到 PMS 的相应数据结构中。providers 保存在 ProviderIntentResolver 对象中；services 保存在 ServiceIntentResolver 对象中；receivers 和 activities 保存在 ActivityIntentResolver 中；permissionGroups、permissions 和 permissions 保存在 HashMap 中。ProviderIntentResolver、ServiceIntentResolver 和 ActivityIntentResolver 都是继承于 IntentResolver，它们的类图关系如下：



到这里，我们就把 scanDirLI 介绍完了，依次扫描完/system/framework、/system/priv-app、/system/app、/vendor/app 这四个目录下面所有的 APK 文件，并解析成一个个 Package 对象，并把他们加入到 PMS 和 Settings 中的一些数据结构中。接着回到 PMS 的构造函数来分析：

```

final List<String> possiblyDeletedUpdatedSystemApps =
new ArrayList<String>();
if (!mOnlyCore) {
    Iterator<PackageSetting> psit =
mSettings.mPackages.values().iterator();
    while (psit.hasNext()) {
        PackageSetting ps = psit.next();

```

```
        if ((ps.pkgFlags & ApplicationInfo.FLAG_SYSTEM)
== 0) {

        continue;

    }

final PackageParser.Package scannedPkg =
mPackages.get(ps.name);

if (scannedPkg != null) {

    if
(mSettings.isDisabledSystemPackageLPr(ps.name)) {

        Slog.i(TAG, "Expecting better updatd
system app for " + ps.name
+ "; removing system app");

        removePackageLI(ps, true);

    }

    continue;

}

if
(!mSettings.isDisabledSystemPackageLPr(ps.name)) {

    psit.remove();

}
```

```

        String msg = "System package " + ps.name
                + " no longer exists; wiping its data";
        reportSettingsProblem(Log.WARN, msg);
        removeDataDirsLI(ps.name);

    } else {
        final PackageSetting disabledPs =
mSettings.getDisabledSystemPkgLPr(ps.name);
        if (disabledPs.codePath == null
|| !disabledPs.codePath.exists()) {

possiblyDeletedUpdatedSystemApps.add(ps.name);

    }
}

}
}

```

上面的代码主要是处理在 `mSettings` 中保存了的 `System app`, 但在刚刚的 `scanDirLI` 并没有找到对应的 `APK`, 这里分三种情况: 一是在 `mSettings` 中有保存, 刚刚的 `scanDirLI` 也有找到, 这里判断它是否被 `disable` 的 `APK`, 如果是, 说明这个 `apk` 是通过 OTA 升级后更新了的, 所以清除相应的数据结构; 二是在 `mSettings` 中有保存, 但刚刚的 `scanDirLI` 没有找到对应的 `APK`, 并且不是被 `disable` 的 `APK` 就直接删除相应的数据结构; 三是在 `mSettings` 中有保存, 但刚刚的 `scanDirLI` 没有找到对应的 `APK`, 并且是被 `disable` 的 `APK`, 就通过查看它的 `codePath` 是否存在来判断它是否有可能被更新或者删除, 并添加到 `possiblyDeletedUpdatedSystemApps` 链表里。接着来看 PMS 的构造函数:

```

ArrayList<PackageSetting> deletePkgsList =
mSettings.getListOfIncompleteInstallPackagesLPr();
for(int i = 0; i < deletePkgsList.size(); i++) {
    //clean up here
    cleanupInstallFailedPackage(deletePkgsList.get(i));
}

```

```

        deleteTempPackageFiles();
        mSettings.pruneSharedUsersLPw();

        if (!mOnlyCore) {

EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_DATA_SCAN_START,
                    SystemClock.uptimeMillis());
        mAppInstallObserver = new AppDirObserver(
            mAppInstallDir.getPath(), SERVER_EVENTS, false, false);
        mAppInstallObserver.startWatching();
        scanDirLI(mAppInstallDir, 0, scanMode, 0);

        mDrmAppInstallObserver = new AppDirObserver(
            mDrmAppPrivateInstallDir.getPath(), SERVER_EVENTS, false,
false);
        mDrmAppInstallObserver.startWatching();
        scanDirLI(mDrmAppPrivateInstallDir,
PackageParserPARSE_FORWARD_LOCK,
            scanMode, 0);

        for (String deletedAppName : possiblyDeletedUpdatedSystemApps) {
            PackageParser.Package deletedPkg =
mPackages.get(deletedAppName);

mSettings.removeDisabledSystemPackageLPw(deletedAppName);

            String msg;
            if (deletedPkg == null) {
                msg = "Updated system package " + deletedAppName
                    + " no longer exists; wiping its data";
                removeDataDirsLI(deletedAppName);
            } else {
                msg = "Updated system app + " + deletedAppName
                    + " no longer present; removing system privileges for
"
                    + deletedAppName;

                deletedPkg.applicationInfo.flags &=
~ApplicationInfo.FLAG_SYSTEM;

                PackageSetting deletedPs =
mSettings.mPackages.get(deletedAppName);
                deletedPs.pkgFlags &= ~ApplicationInfo.FLAG_SYSTEM;

```

```
        }
        reportSettingsProblem(Log.WARN, msg);
    }
} else {
    mAppInstallObserver = null;
    mDrmAppInstallObserver = null;
}
```

首先去删除安装不完整的 APK 文件及其数据，然后清除安装的 temp 文件，并清除在 mSharedUsers 没有被使用的 ShareUserSettins。接着去扫描 /data/app 和 /data/app-private 两个目录中的所有 APK 文件，并建立对应的 FileObserver。最后处理通过 OTA 更新或者删除的 APK 文件。来看 PMS 构造函数的最后一部分：

```
updateAllSharedLibrariesLPw();
```

```
final boolean regrantPermissions =
mSettings.mInternalSdkPlatform
!= mSdkVersion;

if (regrantPermissions) Slog.i(TAG, "Platform changed from "
+ mSettings.mInternalSdkPlatform + " to " +
mSdkVersion
+ "; regranting permissions for internal storage");

mSettings.mInternalSdkPlatform = mSdkVersion;

updatePermissionsLPw(null, null,
UPDATE_PERMISSIONS_ALL
| (regrantPermissions
```

```

    ?

(UPDATE_PERMISSIONS_REPLACE_PKG|UPDATE_PERMISSIONS_REPLACE
_ALL)

        : 0));

if (!mRestoredSettings && !onlyCore) {

    mSettings.readDefaultPreferredAppsLPw(this, 0);

}

mSettings.writeLPr();

Runtime.getRuntime().gc();

} // synchronized (mPackages)

} // synchronized (mInstallLock)

}

```

这一部分的代码量虽然很少，却做了比较多的事情，首先调用 `updateAllSharedLibrariesLPw` 为所有需要使用 `ShareLibrary` 的 package 找到对应的 `ShareLibrary` 路径，并把这些路径保存在 `package` 的 `usesLibraryFiles` 数组中。接着调用 `updatePermissionsLPw` 为需要使用权限的 APK 分配对应的权限：

```

private void updatePermissionsLPw(String changingPkg,
    PackageParser.Package pkgInfo, int flags) {
    Iterator<BasePermission> it = mSettings.mPermissionTrees.values().iterator();
    while (it.hasNext()) {
        final BasePermission bp = it.next();
        if (bp.packageSetting == null) {
            bp.packageSetting = mSettings.mPackages.get(bp.sourcePackage);
        }
    }
}

```

```

it = mSettings.mPermissions.values().iterator();
while (it.hasNext()) {
    final BasePermission bp = it.next();
    if (bp.type == BasePermission.TYPE_DYNAMIC) {
        if (bp.packageSetting == null && bp.pendingInfo != null) {
            final BasePermission tree = findPermissionTreeLP(bp.name);
            if (tree != null && tree.perm != null) {
                bp.packageSetting = tree.packageSetting;
                bp.perm = new PackageParser.Permission(tree.perm.owner,
                    new PermissionInfo(bp.pendingInfo));
                bp.perm.info.packageName = tree.perm.info.packageName;
                bp.perm.info.name = bp.name;
                bp.uid = tree.uid;
            }
        }
    }
    if (bp.packageSetting == null) {
        bp.packageSetting = mSettings.mPackages.get(bp.sourcePackage);
    }
    if (bp.packageSetting == null) {
        Slog.w(TAG, "Removing dangling permission: " + bp.name
            + " from package " + bp.sourcePackage);
        it.remove();
    } else if (changingPkg != null && changingPkg.equals(bp.sourcePackage)) {
        if (pkgInfo == null || !hasPermission(pkgInfo, bp.name)) {
            Slog.i(TAG, "Removing old permission: " + bp.name
                + " from package " + bp.sourcePackage);
            flags |= UPDATE_PERMISSIONS_ALL;
            it.remove();
        }
    }
}

if ((flags&UPDATE_PERMISSIONS_ALL) != 0) {
    for (PackageParser.Package pkg : mPackages.values()) {
        if (pkg != pkgInfo) {
            grantPermissionsLPw(pkg,
(flags&UPDATE_PERMISSIONS_REPLACE_ALL) != 0);
        }
    }
}
}

```

updatePermissionsLPw 中首先对 mPermissionTrees 和 mPermissions 两个 Map 中的 permission 的一些信息进行赋值，然后调用 grantPermissionsLPw 为每个 package 分配权

限，Android 分配权限其实就是分配对应的 `gid` 号。在 `grantPermissionsLPw` 有一系列的判断条件，如果请求分配的权限被允许，就会将对应的 `gid` 号码加入到 `GrantedPermissions` 的 `gids` 数组当中。回到 PMS 的构造函数函数，最后调用 `readDefaultPreferredAppsLPw` 去设置对于处理 Intent 默认的 Activity 信息，关于这部分，我们在 ActivityMangerService 中再来介绍。在 PMS 构造函数的最后调用 `mSettings.writeLPr` 将所有的 package、permission、ShareUserID 等信息全部写到 `/data/system/packages.xml` 中。到这里 PMS 的构造函数就介绍完了。

十三、Android ANR

1、为什么会发生 ANR？

2、如何定位 ANR？

3、如何避免 ANR？

文章、什么是 ANR

ANR 全称 `Application Not Responding`，意思是程序未响应。如果一个应用无法响应用户的输入，系统就会放入一个 ANR

出现场景

- 主线程被 IO 操作（从 4.0 之后网络 IO 中断在主线程中）中断。
- 主线程中存在耗时的计算
- 主线程中错误的操作，线程.wait 或者线程.sleep 等

Android 系统会监控程序的响应状况，一旦出现以下两种情况，则点击 ANR 依次

- 应用在 **5** 秒内未响应用户的输入事件（如按键或触摸）

- 广播接收器未在 **10** 秒内完成相关的处理

如何避免

基本的思路就是将 IO 操作在工作线程来处理，减少其他耗时操作和错误操作

- 使用 **AsyncTask** 处理耗时 IO 操作。
- 使用 Thread 或 HandlerThread 时，调用 `Process.setThreadPriority` (`Process.THREAD_PRIORITY_BACKGROUND`) 设置优先级，否则仍会降低程序响应，因为线程的优先级和主线程相同。
- 使用 **Handler** 处理工作线程结果，而不是使用 `Thread.wait()` 或者 `Thread.sleep()` 来双重主线程。
- Activity 的在 Create 和 on Resume 上中尝试避免耗时的代码
- BroadcastReceiver 中的 `onReceive` 代码也要尽量减少耗时，建议使用 IntentService 处理。

画龙点睛

通常 **100** 到 **200** 分钟就会有人察觉程序反应慢，为了更加增强响应，可以使用下面的几种方法

- 如果程序正在执行后台处理用户的输入，建议使用让用户知道进度，例如使用 `ProgressBar` 控件。
- 程序启动时可以选择加上欢迎界面，避免让用户察觉卡顿。
- 使用 `Systrace` 和 `TraceView` 发现影响响应的问题。

如何定位

如果开发机器上出现问题，我们可以通过查看 `/data/anr/traces.txt` 立即，最新的 ANR 信息在最开始部分。我们从 stacktrace 中可以找到出问题的具体行数。本例中问题出现在 `MainActivity.java` 27 行，因为这里调用了 `Thread.sleep` 方法。

```
1 root@htc_m8t1:/ # cat /data/anr/traces.txt | more
2
3
4 ----- pid 30307 at 2015-05-30 14:51:14 -----
5 Cmd line: com.example.androidyue.bitmapdemo
6
7 JNI: CheckJNI is off; workarounds are off; pins=0; globals=272
8
9 DALVIK THREADS:
10 (mutexes: t1l=0 ts1=0 tscl=0 ghl=0)
11
12 "main" prio=5 tid=1 TIMED_WAIT
13   | group="main" sCount=1 dsCount=0 obj=0x416eaf18 self=0x416d8650
14   | sysTid=30307 nice=0 sched=0/0 cgrp=apps handle=1074565528
15   | state=S schedstat=( 0 0 0 ) utm=5 stm=4 core=3
16   at java.lang.VMThread.sleep(Native Method)
17   at java.lang.Thread.sleep(Thread.java:1044)
18   at java.lang.Thread.sleep(Thread.java:1026)
19   at com.example.androidyue.bitmapdemo.MainActivity$1.run(MainActivity.java:27)
20   at android.app.Activity.runOnUiThread(Activity.java:4794)
21   at com.example.androidyue.bitmapdemo.MainActivity.onResume(MainActivity.java:33)
22   at android.app.Instrumentation.callActivityOnResume(Instrumentation.java:1282)
23   at android.app.Activity.performResume(Activity.java:5405)
```

细致分析

提问：BroadcastReceiver 过了 60 秒居然没有 ANR？现场代码如下

```
1 public class NetworkReceiver extends BroadcastReceiver{
2     private static final String LOGTAG = "NetworkReceiver";
3
4     @Override
5     public void onReceive(Context context, Intent intent) {
6         Log.i(LOGTAG, "onReceive intent=" + intent);
7         try {
8             Thread.sleep(60000);
9         } catch (InterruptedException e) {
10             e.printStackTrace();
11         }
12         Log.i(LOGTAG, "onReceive end");
13     }
14 }
```

回答:实际上已经发生了 ANR,只是没有进行对话框弹出而已这种 ANR

就是背景 ANR, 即后台程序的 ANR, 我们可以通过过滤日志验证

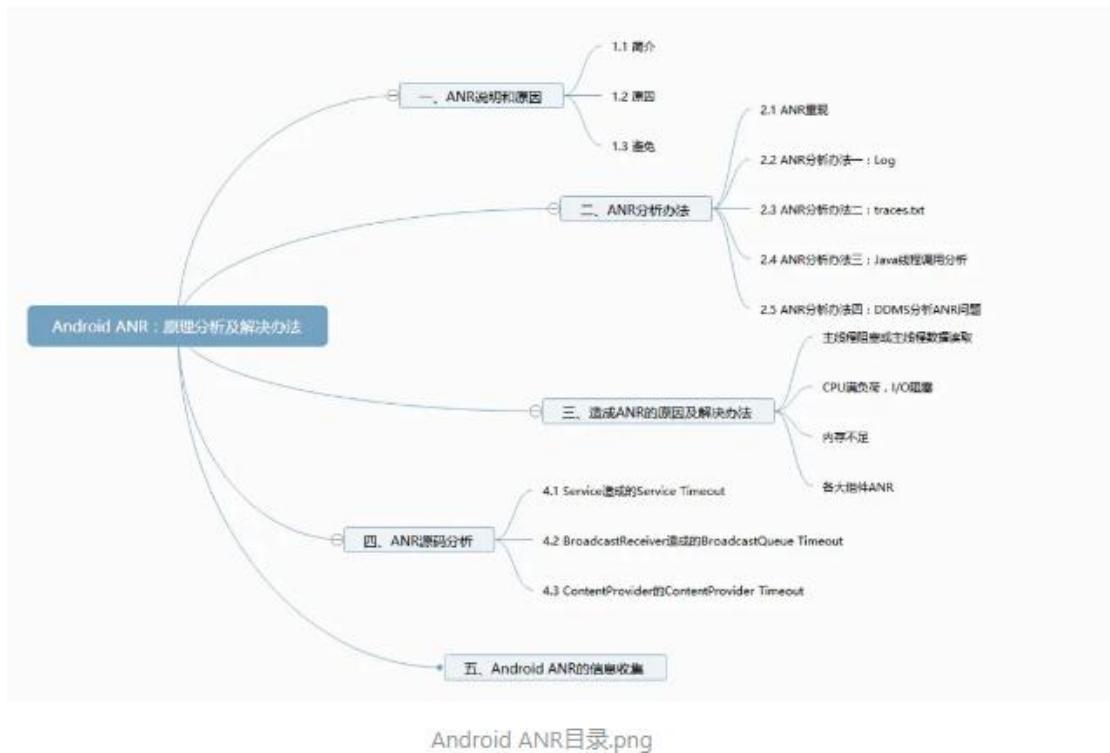
```
1  adb logcat | grep "NetworkReceiver|ActivityManager|WindowManager"
2  I/NetworkReceiver( 4109): onReceive intent=Intent { act=android.net.conn.CONNECTIVITY_CHANGE
3  I/ActivityManager( 462): No longer want com.android.exchange (pid 1054): empty #17
4  I/NetworkReceiver( 4109): onReceive end
5  W/BroadcastQueue( 462): Receiver during timeout: ResolveInfo{5342dde4 com.example.androidyue
6  E/ActivityManager( 462): ANR in com.example.androidyue.bitmapdemo
7  E/ActivityManager( 462): Reason: Broadcast of Intent { act=android.net.conn.CONNECTIVITY_CHA
8  E/ActivityManager( 462): Load: 0.37 / 0.2 / 0.14
9  E/ActivityManager( 462): CPU usage from 26047ms to 0ms ago:
10 E/ActivityManager( 462):   0.4% 58/adbd: 0% user + 0.4% kernel / faults: 1501 minor
11 E/ActivityManager( 462):   0.3% 462/system_server: 0.1% user + 0.1% kernel
12 E/ActivityManager( 462):   0% 4109/com.example.androidyue.bitmapdemo: 0% user + 0% kernel /
13 E/ActivityManager( 462): 1.5% TOTAL: 0.5% user + 0.9% kernel + 0% softirq
14 E/ActivityManager( 462): CPU usage from 87ms to 589ms later:
15 E/ActivityManager( 462):   1.8% 58/adbd: 0% user + 1.8% kernel / faults: 30 minor
16 E/ActivityManager( 462):   1.8% 58/adbd: 0% user + 1.8% kernel
17 E/ActivityManager( 462): 4% TOTAL: 0% user + 4% kernel
18 W/ActivityManager( 462): Killing ProcessRecord{5326d418 4109:com.example.androidyue.bitmapde
19 I/ActivityManager( 462): Process com.example.androidyue.bitmapdemo (pid 4109) has died.
```

除了日志，我们还可以根据前面提到的查看 traces.txt 文件。

提问: 可以更容易了解背景 ANR 么？

答案:当然可以，在 Android 开发者选项—>高级—>显示所有”应用程序无响应“可以直接对后台 ANR 也进行弹窗显示，方便查看了解程序运行情况。

文章、如何避免以及分析方法



Android ANR目录.png

一、ANR 说明和原因

1.1 简介

ANR 全称：**Application Not Responding**，也就是应用程序无响应。

1.2 原因

Android 系统中，**ActivityManagerService**(简称 AMS) 和 **WindowManagerService**(简称 WMS) 会检测 App 的响应时间，如果 App 在特定时间无法相应屏幕触摸或键盘输入时间，或者特定事件没有处理完毕，就会出现 ANR。

以下四个条件都可以造成 ANR 发生：

- **InputDispatching Timeout**: 5 秒内无法响应屏幕触摸事件或键盘输入事件
- **BroadcastQueue Timeout** : 在执行前台广播（BroadcastReceiver）的 `onReceive()` 函数时 10 秒没有处理完成，后台为 60 秒。
- **Service Timeout** : 前台服务 20 秒内，后台服务在 200 秒内没有执行完毕。
- **ContentProvider Timeout** : ContentProvider 的 publish 在 10s 内没进行完。

1.3 避免

尽量避免在主线程（UI 线程）中作耗时操作。
那么耗时操作就放在子线程中。
关于多线程可以参考：Android 多线程：理解和简单使用总结

二、ANR 分析办法

2.1 ANR 重现

这里使用的是号称 Google 亲儿子的 Google Pixel xl (Android 8.0 系统) 做的测试，生成一个按钮跳转到 `ANRTestActivity`，在后者的 `onCreate()` 中主线程休眠 20 秒：

```
@Overrideprotected void onCreate(@Nullable Bundle savedInstanceState) {  
  
    super.onCreate(savedInstanceState);  
  
    setContentView(R.layout.activity_anr_test);  
  
    // 这是 Android 提供线程休眠函数，与 Thread.sleep() 最大的区别是  
  
    // 该使用该函数不会抛出 InterruptedException 异常。  
  
    SystemClock.sleep(20 * 1000);}
```

在进入 `ANRTestActivity` 后黑屏一段时间，大概有七八秒，终于弹出了 ANR 异常。



2.2 ANR 分析办法一：Log

刚才产生 ANR 后，看下 Log:

```
13-07 11:31:57.534 25104-23364/com.sky.myjavatest I/zygote64: Thread[2,tid=23364,WaitingForMicroSignalCatcherLoop,Thread*=0x4d4b7490,peer=0x13340020,"Signal Catcher"]: reacting to signal 3
13-07 11:31:57.534 25104-23364/com.sky.myjavatest I/zygote64: Write stack traces to '/data/smc/traces.txt'
13-07 11:34:11.283 25104-23364/com.sky.myjavatest I/Choreographer: Skipped 1108 frames!  The application may be doing too much work on its main thread.
```

可以看到 logcat 清晰地记录了 ANR 发生的时间，以及线程的 tid 和一句话概括原因：WaitingInMainSignalCatcherLoop，大概意思为主线程等待异常。

最后一句 The application may be doing too much work on its main thread. 告知可能在主线程做了太多的工作。

2.3 ANR 分析办法二：traces.txt

刚才的 log 有第二句 Wrote stack traces to '/data/anr/traces.txt'，说明 ANR 异常已经输出到 traces.txt 文件，使用 adb 命令把这个文件从手机里导出来：

1. cd 到 adb.exe 所在的目录，也就是 **Android SDK** 的 platform-tools 目录，例如：

```
cd D:\Android\AndroidSdk\platform-tools
```

此外，除了 **Windows** 的 cmd 以外，还可以使用 **AndroidStudio** 的 Terminal 来输入 adb 命令。

1. 到指定目录后执行以下 adb 命令导出 traces.txt 文件：

```
adb pull /data/anr/traces.txt
```

traces.txt 默认会被导出到 **Android SDK** 的 \platform-tools 目录。一般来说 traces.txt 文件记录的东西会比较多，分析的时候需要有针对性地去找相关记录。

```
----- pid 23346 at 2017-11-07 11:33:57 ----- ----> 进程 id 和 ANR 产生时间  
cmd line: com.sky.myjavatestBuild fingerprint: 'google/marlin/marlin:8.0.  
0/OPR3.170623.007/4286350:user/release-keys'
```

```
ABI: 'arm64' Build type: optimizedZygote loaded classes=4681 post zygote  
classes=106Intern table: 42675 strong; 137 weak
```

```
JNI: CheckJNI is on; globals=526 (plus 22 weak)
```

```
Libraries: /system/lib64/libandroid.so /system/lib64/libcompiler_rt.so /  
system/lib64/libjavacrypto.so /system/lib64/libjnigraphics.so /system/lib  
64/libmedia_jni.so /system/lib64/libsoundpool.so /system/lib64/libwebview  
chromium_loader.so libjavacore.so libopenjdk.so (9)
```

```
Heap: 22% free, 1478KB/1896KB; 21881 objects -----> 内存使用情况
```

```
...
```

```
"main" prio=5 tid=1 Sleeping    ----> 原因为 Sleeping
| group="main" sCount=1 dsCount=0 flags=1 obj=0x733d0670 self=0x74a4ab
ea00
| sysTid=23346 nice=-10 cgrp=default sched=0/0 handle=0x74a91ab9b0
| state=S schedstat=( 391462128 82838177 354 ) utm=33 stm=4 core=3 HZ=1
00
| stack=0x7fe6fac000-0x7fe6fae000 stackSize=8MB
| held mutexes=
at java.lang.Thread.sleep(Native method)
- sleeping on <0x053fd2c2> (a java.lang.Object)

at java.lang.Thread.sleep(Thread.java:373)
- locked <0x053fd2c2> (a java.lang.Object)

at java.lang.Thread.sleep(Thread.java:314)

at android.os.SystemClock.sleep(SystemClock.java:122)

at com.sky.myjavatest.AnRTestActivity.onCreate(AnRTestActivity.java:20)
----> 产生 ANR 的包名以及具体行数

at android.app.Activity.performCreate(Activity.java:6975)

at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1213)

at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2770)

at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2892)

at android.app.ActivityThread.-wrap11(ActivityThread.java:-1)

at android.app.ActivityThread$H.handleMessage(ActivityThread.java:159
3)
```

```
at android.os.Handler.dispatchMessage(Handler.java:105)  
at android.os.Looper.loop(Looper.java:164)  
at android.app.ActivityThread.main(ActivityThread.java:6541)  
at java.lang.reflect.Method.invoke(Native method)  
at com.android.internal.os.Zygote$MethodAndArgsCaller.run(Zygote.java:  
240)  
at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:767)
```

在文件中使用 `ctrl + F` 查找包名可以快速定位相关代码。

通过上方 log 可以看出相关问题：

- 进程 id 和包名: `pid 23346 com.sky.myjavatest`
- 造成 ANR 的原因: `Sleeping`
- 造成 ANR 的具体行数: `ANRTestActivity.java:20` 类的第 20 行

特别注意：产生新的 ANR，原来的 `traces.txt` 文件会被覆盖。

2.4 ANR 分析办法三：Java 线程调用分析

通过 JDK 提供的命令可以帮助分析和调试 Java 应用，命令为：

```
jstack {pid}
```

```
7266 Test
```

```
7267 Jps
```

具体分析参考：Android 应用 ANR 分析 四. 1 节

2.5 ANR 分析办法四：DDMS 分析 ANR 问题

- 使用 DDMS——Update Threads 工具
- 阅读 Update Threads 的输出

具体分析参考：Android 应用 ANR 分析 四. 2 节

三、造成 ANR 的原因及解决办法

上面例子只是由于简单的主线程耗时操作造成的 ANR，造成 ANR 的原因还有很多：

- 主线程阻塞或主线程数据读取

解决办法：避免死锁的出现，使用子线程来处理耗时操作或阻塞任务。尽量避免在主线程 query provider、不要滥用 SharePreferenceS

- CPU 满负荷，I/O 阻塞

解决办法：文件读写或数据库操作放在子线程异步操作。

- 内存不足

解决办法：AndroidManifest.xml 文件<application>中可以设置 android:largeHeap="true"，以此增大 App 使用内存。不过**不建议使用此法**，从根本上防止内存泄漏，优化内存使用才是正道。

- 各大组件 ANR

各大组件生命周期中也应避免耗时操作，注意 BroadcastReceiver 的 onReceive()、后台 Service 和 ContentProvider 也不要执行太长时间的任务。

四、ANR 源码分析

特别声明：文章 理解 Android ANR 的触发原理 分别记录了由 Service、BroadcastReceiver 和 ContentProvider 造成的 ANR。下文引用该文代码，并依据自己的简单理解作总结。

4.1 Service 造成的 Service Timeout

Service Timeout 是位于“ActivityManager”线程中的 AMS.MainHandler 收到 SERVICE_TIMEOUT_MSG 消息时触发。

4.1.1 发送延时消息

Service 进程 attach 到 system_server 进程的过程中会调用 realStartServiceLocked，紧接着 mAm.mHandler.sendMessageAtTime() 来发送一个延时消息，延时的时常是定义好的，如前台 Service 的 20 秒。ActivityManager 线程中的 AMS.MainHandler 收到 SERVICE_TIMEOUT_MSG 消息时会触发。

AS.realStartServiceLocked

```
ActiveServices.java
private final void realStartServiceLocked(ServiceRecord r,
                                           ProcessRecord app, boolean execInFg) throws RemoteException {
    ...
    //发送 delay 消息(SERVICE_TIMEOUT_MSG)
```

```
bumpServiceExecutingLocked(r, execInFg, "create");

try {

    ...

    //最终执行服务的 onCreate()方法

    app.thread.scheduleCreateService(r, r.serviceInfo,

        mAm.compatibilityInfoForPackageLocked(r.serviceInfo.appli
        cationInfo),

        app.repProcState);

} catch (DeadObjectException e) {

    mAm.appDiedLocked(app);

    throw e;

} finally {

    ...

}

AS.bumpServiceExecutingLocked private final void bumpServiceExecutingLoc
ked(ServiceRecord r, boolean fg, String why) {

    ...

    scheduleServiceTimeoutLocked(r.app);}

void scheduleServiceTimeoutLocked(ProcessRecord proc) {

    if (proc.executingServices.size() == 0 || proc.thread == null) {

        return;

    }

    long now = SystemClock.uptimeMillis();
```

```
Message msg = mAm.mHandler.obtainMessage(  
        ActivityManagerService.SERVICE_TIMEOUT_MSG);  
  
msg.obj = proc;  
  
//当超时后仍没有 remove 该 SERVICE_TIMEOUT_MSG 消息，则执行 service Timeout  
t 流程  
  
mAm.mHandler.sendMessageAtTime(msg,  
        proc.execServicesFg ? (now+SERVICE_TIMEOUT) : (now+ SERVICE_BACKG  
ROUND_TIMEOUT));}
```

4.1.2 进入目标进程的主线程创建 Service

经过 Binder 等层层调用进入目标进程的主线程
handleCreateService(CreateServiceData data)。

```
ActivityThread.java  
private void handleCreateService(CreateServiceData data) {  
  
    ...  
  
    java.lang.ClassLoader cl = packageInfo.getClassLoader();  
  
    Service service = (Service) cl.loadClass(data.info.name).newInstance();  
  
    ...  
  
    try {  
  
        //创建 ContextImpl 对象  
  
        ContextImpl context = ContextImpl.createAppContext(this, pack  
ageInfo);  
  
        context.setOuterContext(service);  
  
        //创建 Application 对象
```

```
        Application app = packageInfo.makeApplication(false, mInstrumentation);

        service.attach(context, this, data.info.name, data.token, app,
p,

            ActivityManagerNative.getDefault());

        //调用服务 onCreate()方法

        service.onCreate();

        ...

        //取消 AMS.MainHandler 的延时消息

        ActivityManagerNative.getDefault().serviceDoneExecuting(
            data.token, SERVICE_DONE_EXECUTING_ANON, 0, 0);

    } catch (Exception e) {

        ...

    }

}
```

这个方法中会创建目标服务对象，以及回调常用的 **Service** 的 `onCreate()`方法，紧接着通过 `serviceDoneExecuting()`回到 `system_server` 执行取消 `AMS.MainHandler` 的延时消息。

4.1.3 回到 `system_server` 执行取消 `AMS.MainHandler` 的延时消息

AS.serviceDoneExecutingLocked

```
private void serviceDoneExecutingLocked(ServiceRecord r, boolean inDestroying,
                                         boolean finishing) {

    ...

    if (r.executeNesting <= 0) {

        if (r.app != null) {
```

```
r.app.execServicesFg = false;

r.app.executingServices.remove(r);

if (r.app.executingServices.size() == 0) {

    //当前服务所在进程中没有正在执行的 service

    mAm.mHandler.removeMessages(ActivityManagerService.SERVICE_TIMEOUT_MSG, r.app);

    ...

}

...}
```

此方法中 Service 逻辑处理完成则移除之前延时的消息 SERVICE_TIMEOUT_MSG。如果没有执行完毕不调用这个方法，则超时后会发出 SERVICE_TIMEOUT_MSG 来告知 ANR 发生。

4.2 BroadcastReceiver 造成的 BroadcastQueue Timeout

BroadcastReceiver Timeout 是位于“ActivityManager”线程中的 BroadcastQueue. BroadcastHandler 收到 BROADCAST_TIMEOUT_MSG 消息时触发。

4.2.1 处理广播函数 processNextBroadcast() 中 broadcastTimeoutLocked(false) 发送延时消息

广播处理顺序为先处理并行广播，再处理当前有序广播。

```
final void processNextBroadcast(boolean fromMsg) {

    synchronized(mService) {

        ...

        // 处理当前有序广播

        do {

            r = mOrderedBroadcasts.get(0);

            //获取所有该广播所有的接收者
```

```
        int numReceivers = (r.receivers != null) ? r.receivers.size()
        : 0;

        if (mService.mProcessesReady && r.dispatchTime > 0) {

            long now = SystemClock.uptimeMillis();

            if ((numReceivers > 0) &&
                (now > r.dispatchTime + (2*mTimeoutPeriod*numReceivers))) {

                //step 1. 发送延时消息，这个函数处理了很多事情，比如广播处
                理超时结束广播

                broadcastTimeoutLocked(false);

                ...

            }

        }

        if (r.receivers == null || r.nextReceiver >= numReceivers
            || r.resultAbort || forceReceive) {

            if (r.resultTo != null) {

                //2. 处理广播消息消息

                performReceiveLocked(r.callerApp, r.resultTo,
                    new Intent(r.intent), r.resultCode,
                    r.resultData, r.resultExtras, false, false, r.user
                    Id);

                r.resultTo = null;

            }

            //3. 取消广播超时 ANR 消息

            cancelBroadcastTimeoutLocked();
        }
    }
}
```

```
        }

    } while (r == null);

    ...

// 获取下条有序广播

r.receiverTime = SystemClock.uptimeMillis();

if (!mPendingBroadcastTimeoutMessage) {

    long timeoutTime = r.receiverTime + mTimeoutPeriod;

    // 设置广播超时

    setBroadcastTimeoutLocked(timeoutTime);

}

...

}}
```

上文的 step 1. broadcastTimeoutLocked(false) 函数：记录时间信息并调用函数设置发送延时消息

```
final void broadcastTimeoutLocked(boolean fromMsg) {

    ...

    long now = SystemClock.uptimeMillis();

    if (fromMsg) {

        if (mService.mDidDexOpt) {

            // Delay timeouts until dexopt finishes.

            mService.mDidDexOpt = false;

            long timeoutTime = SystemClock.uptimeMillis() + mTimeoutP
eriod;
```

```
        setBroadcastTimeoutLocked(timeoutTime);

        return;

    }

    if (!mService.mProcessesReady) {

        return;

    }

    long timeoutTime = r.receiverTime + mTimeoutPeriod;

    if (timeoutTime > now) {

        // step 2

        setBroadcastTimeoutLocked(timeoutTime);

        return;

    }

}
```

上文的 step 2.setBroadcastTimeoutLocked 函数： 设置广播超时具体操作， 同样是发送延时消息

```
final void setBroadcastTimeoutLocked(long timeoutTime) {

    if (!mPendingBroadcastTimeoutMessage) {

        Message msg = mHandler.obtainMessage(BROADCAST_TIMEOUT_MSG, thi
s);

        mHandler.sendMessageAtTime(msg, timeoutTime);

        mPendingBroadcastTimeoutMessage = true;

    }

}
```

4. 2. 2 setBroadcastTimeoutLocked(long timeoutTime) 函数的参数 timeoutTime 是当前时间加上设定好的超时时间。

也就是上文的

```
long timeoutTime = SystemClock.uptimeMillis() + mTimeoutPeriod;
```

mTimeoutPeriod 也就是前台队列的 10s 和后台队列的 60s。

```
public ActivityManagerService(Context systemContext) {  
  
    ...  
  
    static final int BROADCAST_FG_TIMEOUT = 10 * 1000;  
  
    static final int BROADCAST_BG_TIMEOUT = 60 * 1000;  
  
    ...  
  
    mFgBroadcastQueue = new BroadcastQueue(this, mHandler,  
        "foreground", BROADCAST_FG_TIMEOUT, false);  
  
    mBgBroadcastQueue = new BroadcastQueue(this, mHandler,  
        "background", BROADCAST_BG_TIMEOUT, true);  
  
    ...}
```

4.2.3 在 processNextBroadcast() 过程，执行完 performReceiveLocked 后调用 cancelBroadcastTimeoutLocked

cancelBroadcastTimeoutLocked : 处理广播消息函数 **processNextBroadcast()** 中
performReceiveLocked() 处理广播消息完毕则调用
cancelBroadcastTimeoutLocked() 取消超时消息。

```
final void cancelBroadcastTimeoutLocked() {  
  
    if (mPendingBroadcastTimeoutMessage) {  
  
        mHandler.removeMessages(BROADCAST_TIMEOUT_MSG, this);  
  
        mPendingBroadcastTimeoutMessage = false;  
    }  
}
```

4.3 ContentProvider 的 ContentProvider Timeout

ContentProvider Timeout 是位于”ActivityManager”线程中的 AMS.MainHandler 收到 CONTENT_PROVIDER_PUBLISH_TIMEOUT_MSG 消息时触发。

五、Android ANR 的信息收集

无论是四大组件或者进程等只要发生 ANR，最终都会调用 AMS.appNotResponding()方法。

文章、Android 性能优化之 ANR 详解

1，ANR 分析

1.1 ANR 的分类：

1.

KeyDispatchTimeout –按键或触摸事件在特定时间无响应；

2.

3.

BroadcastTimeout –BroadcastReceiver 在特定时间无法处理完成；

4.

5.

ServiceTimeout –Service 在特定的短期无法处理完成；

6.

1.2 ANR 的发生原因：

1. 应用本身引起，例如：

1. 主线程双向，IOWait 等；

2. 其他进展间接引起，例如：

1. 当前应用进程进行进程间通信请求其他进程，其他进程的操作连续没有反馈；

2. 其他进程的 CPU 占用率高，导致当前应用进程无法抢占到 CPU 时间片；

1.3 ANR 日志分析:

当发生 ANR 的时候 Logcat 中会出现提示;

```
04-06 15:58:46.215 23480-23483/com.example.testanr I/art:  
Thread[2,tid=23483,WaitingInMainSignalCatcherLoop,Thread*=0x7fa2307000,peer=0x12cb40a0,  
"Signal Catcher"]: reacting to signal 3  
04-06 15:58:46.364 23480-23483/com.example.testanr I/art: Wrote stack traces to  
'/data/anr/traces.txt'
```

ANR 的日志信息保存在: **/data/anr/traces.txt**, 每一次新的 **ANR** 发生, 会把之前的 **ANR** 信息覆盖掉。

例如:

```
04-01 13:12:11.572 I/InputDispatcher( 220): Application is not  
responding:Window{2b263310com.android.email/com.android.email.activity.SplitScreenActivity  
paused=false}.  
5009.8ms since event, 5009.5ms since waitstarted  
04-0113:12:11.572 I/WindowManager( 220): Input event  
dispatching timedout sending  
to com.android.email/com.android.email.activity.SplitScreenActivity
```

04-01 13:12:14.123 I/Process(220): Sending signal. PID: 21404 SIG:3---发生 ANR 的时间和生成 trace.txt 的时间

```
04-01 13:12:14.123 I/dalvikvm(21404):threadid=4: reacting to signal 3 .....  
04-0113:12:15.872 E/ActivityManager( 220): ANR in  
com.android.email(com.android.email/.activity.SplitScreenActivity)  
04-0113:12:15.872 E/ActivityManager( 220): Reason:keyDispatchingTimedOut ----ANR 的类型  
04-0113:12:15.872 E/ActivityManager( 220): Load: 8.68 / 8.37 / 8.53 --CPU 的负载情况  
04-0113:12:15.872 E/ActivityManager( 220): CPUUsage from 4361ms to 699ms ago ----CPU 在  
ANR 发生前的使用情况; 备注: 这个 ago, 是发生前一段时间的使用情况, 不是当前时间点的使用情况;
```

04-0113:12:15.872 E/ActivityManager(220): 5.5%21404/com.android.email: 1.3% user + 4.1%
kernel / faults:

10 minor

04-0113:12:15.872 E/ActivityManager(220): 4.3%220/system_server: 2.7% user + 1.5% kernel /
faults: 11

minor 2 major

04-0113:12:15.872 E/ActivityManager(220): 0.9%52/spi_qsd.0: 0% user + 0.9% kernel

04-0113:12:15.872 E/ActivityManager(220): 0.5%65/irq/170-cyttsp: 0% user + 0.5% kernel

04-0113:12:15.872 E/ActivityManager(220): 0.5%296/com.android.systemui: 0.5% user + 0%
kernel

04-0113:12:15.872 E/ActivityManager(220): 100%TOTAL: 4.8% user + 7.6% kernel + 87%

iowait---注意这行：注意 87% 的 iowait

04-0113:12:15.872 E/ActivityManager(220): CPUUsage from 3697ms to 4223ms later:-- ANR 后
CPU 的使用量

04-0113:12:15.872 E/ActivityManager(220): 25% 21404/com.android.email: 25% user + 0%
kernel / faults: 191 minor

04-0113:12:15.872 E/ActivityManager(220): 16% 21603/_eas(par.hakan: 16% user + 0% kernel

04-0113:12:15.872 E/ActivityManager(220): 7.2% 21406/GC: 7.2% user + 0% kernel

04-0113:12:15.872 E/ActivityManager(220): 1.8% 21409/Compiler: 1.8% user + 0% kernel

04-0113:12:15.872 E/ActivityManager(220): 5.5% 220/system_server: 0% user + 5.5% kernel /
faults: 1 minor

04-0113:12:15.872 E/ActivityManager(220): 5.5% 263/InputDispatcher: 0% user + 5.5% kernel

04-0113:12:15.872 E/ActivityManager(220): 32% TOTAL: 28% user + 3.7% kernel

从 Logcat 中可以得到以下信息：

1. 导致 ANR 的包名（com.android.emai），类名
(com.android.email.activity.SplitScreenActivity)，进度 PID (21404)
2. 导致 ANR 的原因：keyDispatchingTimedOut
3. 系统中活跃进程的 CPU 占用率，关键的一句：100%：4.8% 用户 + 7.6% 内核 + 87%
iowait；表示 CPU 占用满负荷了，其中绝大部分是被 iowait 即 I/O 操作占用了。我们
就可以大致得出是 io 操作导致的 ANR。

同时需要注意：并非所有的 ANR 类型都有章可循，很多偶发的 ANR 取代于当时
发生的环境或系统 Bug；因此对 ANR，更应该注意预防而不是分析。

2. ANR 触发场景

1.

InputDispatching Timeout：输入事件分发超时 5s 未响应完成；

2.

3.

BroadcastQueue 超时：前台广播在 10s 内，后台广播在 20 秒内未执行完
成；

4.

5.

服务超时：前台服务在 20s 内，后台服务在 200 秒内未执行完成；

6.

7.

ContentProvider 超时：内容提供者，在发布过超时 10s；

8.

3, ANR 触发场景源码分析

系统的 **ANR** 机制是如何运行的，**ANR** 是如何被发出的呢？

我们先来说说 **Service**，主要用于在后台处理一些耗时的逻辑，或者去执行某些需要长期运行的任务。但很多同志认为服务就可以执行耗时任务，这是一种误解，**Service** 本身也运行于主线程，执行耗时任务同样会发生 **ANR**。此处仅分析 **Service Timeout** 的启动场景。

3.1 ANR 的起源

Service 的启动过程由 **ContextWrapper** 开始，我们直接步入重点位置

ActiveServices.java 中 **realStartServiceLocked** 方法

```
// 此处仅列出精简之后的代码;
```

```
private final void realStartServiceLocked(ServiceRecord r, ProcessRecord app, boolean execInFg) throws RemoteException {
    /**
     * 关键代码：发送延时消息，SERVICE_TIMEOUT_MSG，此处是 Service ANR 的源头
     */
    bumpServiceExecutingLocked(r, execInFg, "create");
    try {
        /**
         * 接下来真正的创建 Service 对象，并执行 Service 的 onCreate()方法
         */
        app.thread.scheduleCreateService(r, r.serviceInfo,
                                         mAm.compatibilityInfoForPackageLocked(r.serviceInfo.applicationInfo),
                                         app.repProcState);
    } catch (DeadObjectException e) {
        Slog.w(TAG, "Application dead when creating service " + r);
        mAm.appDiedLocked(app);
        throw e;
    }
}
```

其中 **bumpServiceExecutingLocked**（）方法又会调用到 **scheduleServiceTimeoutLocked**（）方法重新发送 **SERVICE_TIMEOUT_MSG** 消息。

```
/**
 * 延时发送 SERVICE_TIMEOUT_MSG 消息，前、后台 Service 时间不一样。
 * @param proc
 */
void scheduleServiceTimeoutLocked(ProcessRecord proc) {
    if (proc.executingServices.size() == 0 || proc.thread == null) {
        return;
    }
    long now = SystemClock.uptimeMillis();
```

```

        Message msg = mAm.mHandler.obtainMessage(
                ActivityManagerService.SERVICE_TIMEOUT_MSG);
        msg.obj = proc;
        mAm.mHandler.sendMessageAtTime(msg,
                proc.execServicesFg ? (now+SERVICE_TIMEOUT) : (now+
SERVICE_BACKGROUND_TIMEOUT));
    }

```

接下来调用到了 ActivityThread.java 的 scheduleCreateService () 方法，实际上使用处理程序发送了一条 msg，最终调用到 ActivityThread.java 的 handleCreateService () 方法

```

private void handleCreateService(CreateServiceData data) {
    try {
        java.lang.ClassLoader cl = packageInfo.getClassLoader();
        // 创建 Service 对象
        service = (Service) cl.loadClass(data.info.name).newInstance();
    } catch (Exception e) {
    }
    try {
        // 调用 Service 的 onCreate 方法
        service.onCreate();
        mServices.put(data.token, service);
        // 移除 SERVICE_TIMEOUT_MSG 的消息
        ActivityManagerNative.getDefault().serviceDoneExecuting(
                data.token, SERVICE_DONE_EXECUTING_ANON, 0, 0);
    } catch (Exception e) {
    }
}

```

3. 2 规定时间之内完成了方法的调用

ActivityManagerNative.getDefault () 。serviceDoneExecuting 会执行到 ActivityManagerService.java 中的 serviceDoneExecuting () 方法，长袍执行到 ActiveService.java 中的 serviceDoneExecutingLocked () 方法。

```

private void serviceDoneExecutingLocked(ServiceRecord r, boolean
inDestroying,boolean finishing) {
    if (r.app.executingServices.size() == 0) {
        if ((DEBUG_SERVICE || DEBUG_SERVICE_EXECUTING)
Slog.v(TAG_SERVICE_EXECUTING,
                "No more executingServices of " + r.shortName);
        // 关键代码：remove 掉刚刚延时发送的 Message，否则 Message 被执行，ANR 就发生了：
    }
}

```

```

mAm.mHandler.removeMessages(ActivityManagerService.SERVICE_TIMEOUT_MSG,
r.app);

```

```

        } else if (r.executeFg) {
            // Need to re-evaluate whether the app still needs to be in the
            foreground.
            for (int i=r.app.executingServices.size()-1; i>=0; i--) {
                if (r.app.executingServices.valueAt(i).executeFg) {
                    r.app.execServicesFg = true;
                    break;
                }
            }
        }
    }
}

```

3. 3 规定时间之内未完成方法的调用，出现了 ANR

而如果消息没有被 `mAm.mHandler`（也就是 `ActivityManagerService` 中的 `MainHandler`）及时删除，被执行的话就会开始 ANR 的发生；执行到 `ActivityManagerService` 中 `MainHandler` 的 `SERVICE_TIMEOUT_MSG` 然后调用到 `ActiveServices` 的 `serviceTimeout()` 方法，最终执行到 `ActivityManagerService` 的 `appNotResponding()` 方法。

```

//非常重要的方法，代码多留了点。
final void appNotResponding(ProcessRecord app, ActivityRecord activity,
                             ActivityRecord parent, boolean aboveSystem,
final String annotation) {
    ArrayList<Integer> firstPids = new ArrayList<Integer>(5);
    SparseArray<Boolean> lastPids = new SparseArray<Boolean>(20);

    synchronized (this) {
        // Dump thread traces as quickly as we can, starting with
        "interesting" processes.
        firstPids.add(app.pid);

        int parentPid = app.pid;
        if (parent != null && parent.app != null && parent.app.pid > 0)
parentPid = parent.app.pid;
        if (parentPid != app.pid) firstPids.add(parentPid);
        if (MY_PID != app.pid && MY_PID != parentPid)
firstPids.add(MY_PID);

        for (int i = mLruProcesses.size() - 1; i >= 0; i--) {
            ProcessRecord r = mLruProcesses.get(i);
            if (r != null && r.thread != null) {
                int pid = r.pid;
                if (pid > 0 && pid != app.pid && pid != parentPid &&
pid != MY_PID) {

```

```
        if (r.persistent) {
            firstPids.add(pid);
        } else {
            lastPids.put(pid, Boolean.TRUE);
        }
    }
}

// 获取 ANR 日志信息
StringBuilder info = new StringBuilder();
info.setLength(0);
info.append("ANR in ").append(app.processName);
if (activity != null && activity.shortComponentName != null) {
    info.append(
(").append(activity.shortComponentName).append(")");
}
info.append("\n");
info.append("PID: ").append(app.pid).append("\n");
if (annotation != null) {
    info.append("Reason: ").append(annotation).append("\n");
}
if (parent != null && parent != activity) {
    info.append("Parent:
").append(parent.shortComponentName).append("\n");
}

final ProcessCpuTracker processCpuTracker = new
ProcessCpuTracker(true);

File tracesFile = dumpStackTraces(true, firstPids, processCpuTracker,
lastPids,
NATIVE_STACKS_OF_INTEREST);

String cpuInfo = null;
if (MONITOR_CPU_USAGE) {
    updateCpuStatsNow();
    synchronized (mProcessCpuTracker) {
        cpuInfo = mProcessCpuTracker.printCurrentState(anrTime);
    }
    info.append(processCpuTracker.printCurrentLoad());
    info.append(cpuInfo);
}
```

```
        info.append(processCpuTracker.printCurrentState(anrTime));

        Slog.e(TAG, info.toString());
        if (tracesFile == null) {
            // There is no trace file, so dump (only) the alleged culprit's threads
            to the log
            Process.sendSignal(app.pid, Process.SIGNAL_QUIT);
        }

        // 添加到 DropBox, 2.3 之后出的功能, 解决 traces.txt 被覆盖的问题
        addErrorToDropBox("anr", app, app.processName, activity, parent,
annotation,
                           cpuInfo, tracesFile, null);

        // 获取设置, 确认是否需要弹出 ANR 提示框, 需要的话弹出, 不需要的
        // 话直接 kill。
        boolean showBackground =
Settings.Secure.getInt(mContext.getContentResolver(),
                           Settings.Secure.ANR_SHOW_BACKGROUND, 0) != 0;

        synchronized (this) {
            mBatteryStatsService.noteProcessAnr(app.processName, app.uid);

            if (!showBackground && !app.isInterestingToUserLocked() &&
app.pid != MY_PID) {
                // kill 进程
                app.kill("bg_anr", true);
                return;
            }

            // Set the app's notResponding state, and look up the
            errorReportReceiver
            makeAppNotRespondingLocked(app,
                           activity != null ? activity.shortComponentName : null,
                           annotation != null ? "ANR " + annotation : "ANR",
                           info.toString());

            // Bring up the infamous App Not Responding dialog
            Message msg = Message.obtain();
            HashMap<String, Object> map = new HashMap<String, Object>();
            msg.what = SHOW_NOT RESPONDING_MSG;
            msg.obj = map;
            msg.arg1 = aboveSystem ? 1 : 0;
        }
    }
}
```

```
        map.put("app", app);
        if (activity != null) {
            map.put("activity", activity);
        }
        // 去弹出 ANR 提示框。
        mUiHandler.sendMessage(msg);
    }
}
```

流程总结：

- 1.服务创建之前会延迟发送一个消息，而这个消息就是 **ANR** 的起源；
- 2.服务创建完毕，在规定的时间之内执行完成 **onCreate()** 方法就可删除这个消息，就不会产生 **ANR** 了；
- 3.在规定的时间之内没有完成 **onCreate()** 的调用，消息被执行，**ANR** 发生。

十四、Android 内存相关

1、什么情况下会内存泄漏？

Java 是垃圾回收语言的一种，其优点是开发者无需特意管理**内存分配**，降低了应用由于**局部故障(segmentation fault)**导致崩溃，同时防止未释放的内存把**堆栈(heap)**挤爆的可能，所以写出来的代码更为安全。

不幸的是，在 Java 中仍存在很多容易导致内存泄漏的**逻辑可能(logical leak)**。如果不小心，你的 Android 应用很容易浪费掉未释放的内存，最终导致内存用光的错误抛出(**out-of-memory, OOM**)。

一般内存泄漏(traditional memory leak)的原因是：由忘记释放分配的内存导致的。
(译者注：**Cursor** 忘记关闭等)

逻辑内存泄漏(logical memory leak)的原因是：当应用不再需要这个对象，当仍未释放该对象的所有引用。

如果持有对象的强引用，垃圾回收器是无法在内存中回收这个对象。

在 Android 开发中，最容易引发的内存泄漏问题是 Context。比如 Activity 的 Context，就包含大量的内存引用，例如 **View Hierarchies** 和其他资源。一旦泄漏了 Context，也意味泄漏它指向的所有对象。Android 机器内存有限，太多的内存泄漏容易导致 OOM。

检测逻辑内存泄漏需要主观判断，特别是对象的生命周期并不清晰。幸运的是，Activity 有着明确的生命周期，很容易发现泄漏的原因。Activity.onDestroy()被视为 Activity 生命的结束，程序上来看，它应该被销毁了，或者 Android 系统需要回收这些内存(译者注：当内存不够时，Android 会回收看不见的 Activity)。如果这个方法执行完，在堆栈中仍存在持有该 Activity 的强引用，垃圾回收器就无法把它标记成已回收的内存，而我们本来目的就是要回收它！

结果就是 Activity 存活在它的生命周期之外。

Activity 是重量级对象，应该让 Android 系统来处理它。然而，逻辑内存泄漏总是在不经意间发生。(译者注：曾经试过一个 Activity 导致 20M 内存泄漏)。在 Android 中，导致潜在内存泄漏的陷阱不外乎两种：

- 全局进程(process-global)的 static 变量。这个无视应用的状态，持有 **Activity** 的强引用的怪物。
- 活在 **Activity** 生命周期之外的线程。没有清空对 **Activity** 的强引用。

检查一下你有没有遇到下列的情况。

Static Activities

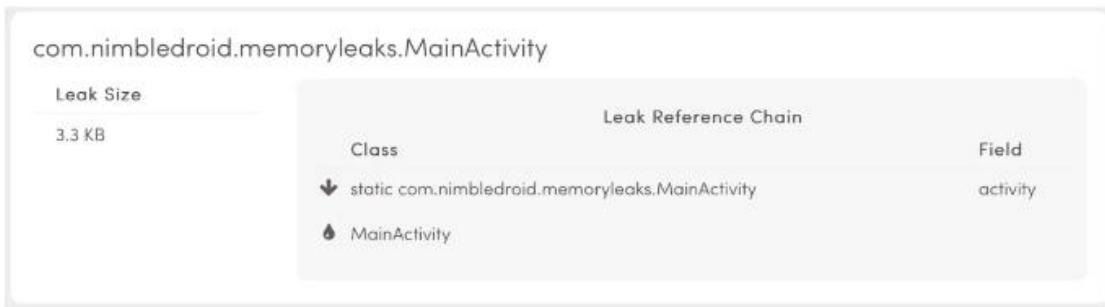
在类中定义了静态 **Activity** 变量，把当前运行的 **Activity** 实例赋值于这个静态变量。

如果这个静态变量在 **Activity** 生命周期结束后没有清空，就导致内存泄漏。因为 static 变量是贯穿这个应用的生命周期的，所以被泄漏的 **Activity** 就会一直存在于应用的进程中，不会被垃圾回收器回收。

```
static Activity activity;

void setStaticActivity() {
    activity = this;
}

View saButton = findViewById(R.id.sa_button);
saButton.setOnClickListener(new View.OnClickListener() {
    @Override public void onClick(View v) {
        setStaticActivity();
        nextActivity();
    }
});
```



Memory Leak 1 - Static Activity

Static Views

类似的情况会发生在单例模式中，如果 `Activity` 经常被用到，那么在内存中保存一个实例是很实用的。正如之前所述，强制延长 `Activity` 的生命周期是相当危险而且不必要的，无论如何都不能这样做。

特殊情况：如果一个 `View` 初始化耗费大量资源，而且在一个 `Activity` 生命周期内保持不变，那可以把它变成 `static`，加载到视图树上(View Hierachy)，像这样，当 `Activity` 被销毁时，应当释放资源。（译者注：示例代码中并没有释放内存，把这个 `static view` 置 `null` 即可，但是还是不建议用这个 `static view` 的方法）

```
static view;

void setStaticView() {

    view = findViewById(R.id.sv_button);
}

View svButton = findViewById(R.id.sv_button);

svButton.setOnClickListener(new View.OnClickListener() {

    @Override public void onClick(View v) {

        setStaticView();

        nextActivity();
    }
});
```

com.nimbledroid.memoryleaks.MainActivity

Leak Size	Leak Reference Chain	Field
3.3 KB	Class	
	↓ static com.nimbledroid.memoryleaks.MainActivity	view
	↓ android.support.v7.widget.AppCompatButton	mContext
	↓ android.support.v7.widget.TintContextWrapper	mBase
	>MainActivity	

Inner Classes

继续，假设 `Activity` 中有个内部类，这样做可以提高可读性和封装性。将如我们创建一个内部类，而且持有一个静态变量的引用，恭喜，内存泄漏就离你不远了（译者注：销毁的时候置空，嗯）。

```
private static Object inner;

void createInnerClass() {

    class InnerClass {

    }

    inner = new InnerClass();
}

View icButton = findViewById(R.id.ic_button);

icButton.setOnClickListener(new View.OnClickListener() {

    @Override public void onClick(View v) {

        createInnerClass();

        nextActivity();
    }
})
```

```
});
```

com.nimbledroid.memoryleaks.MainActivity

Leak Size

3.3 KB

Leak Reference Chain

Class

↓ static com.nimbledroid.memoryleaks.MainActivity

Field

inner

↓ com.nimbledroid.memoryleaks.MainActivity\$1InnerClass

this\$0

↓ MainActivity

内部类的优势之一就是可以访问外部类，不幸的是，导致内存泄漏的原因，就是内部类持有外部类实例的强引用。

Anonymous Classes

相似地，匿名类也维护了外部类的引用。所以内存泄漏很容易发生，当你在 Activity 中定义了匿名的 AsyncTask。当异步任务在后台执行耗时任务期间，Activity 不幸被销毁了（译者注：用户退出，系统回收），这个被 AsyncTask 持有的 Activity 实例就不会被垃圾回收器回收，直到异步任务结束。

```
void startAsyncTask() {  
  
    new AsyncTask<Void, Void, Void>() {  
  
        @Override protected Void doInBackground(Void... params) {  
  
            while(true);  
  
        }  
  
        }.execute();  
  
    }  
  
    super.onCreate(savedInstanceState);  
  
    setContentView(R.layout.activity_main);  
  
    View aicButton = findViewById(R.id.at_button);  
  
    aicButton.setOnClickListener(new View.OnClickListener() {
```

```
@Override public void onClick(View v) {  
  
    startAsyncTask();  
  
    nextActivity();  
  
}  
  
});
```

com.nimbledroid.memoryleaks.MainActivity

Leak Size	Leak Reference Chain	Field
113 KB		
Class		
	↓ com.nimbledroid.memoryleaks.MainActivity\$9	this\$0
	>MainActivity	

Handler

同样道理，定义匿名的 Runnable，用匿名类 Handler 执行。Runnable 内部类会持有外部类的隐式引用，被传递到 Handler 的消息队列 MessageQueue 中，在 Message 消息没有被处理之前，Activity 实例不会被销毁了，于是导致内存泄漏。

```
void createHandler() {  
  
    new Handler() {  
  
        @Override public void handleMessage(Message message) {  
  
            super.handleMessage(message);  
  
        }  
  
        }.postDelayed(new Runnable() {  
  
            @Override public void run() {  
  
                while(true);  
  
            }  
  
            }, Long.MAX_VALUE >> 1);  
  
    }
```

```
View hButton = findViewById(R.id.h_button);

hButton.setOnClickListener(new View.OnClickListener() {

    @Override public void onClick(View v) {

        createHandler();

        nextActivity();

    }

});
```

com.nimbledroid.memoryleaks.MainActivity

Leak Size

3.3 KB

Leak Reference Chain

Class

↓ static com.nimbledroid.memoryleaks.MainActivity

Field

activity

↓ MainActivity

Threads

我们再次通过 `Thread` 和 `TimerTask` 来展现内存泄漏。

```
void spawnThread() {

    new Thread() {

        @Override public void run() {

            while(true);

        }

    }.start();

}
```

```
View tButton = findViewById(R.id.t_button);

tButton.setOnClickListener(new View.OnClickListener() {

    @Override public void onClick(View v) {

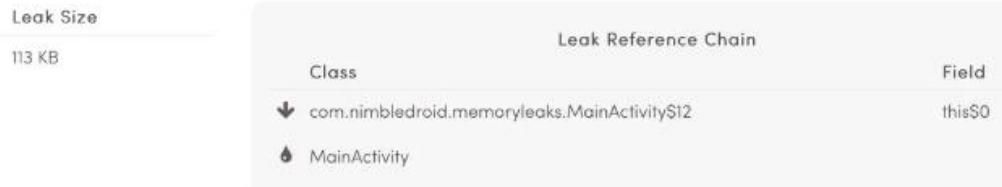
        spawnThread();

        nextActivity();

    }

});
```

com.nimbledroid.memoryleaks.MainActivity



TimerTask

只要是匿名类的实例，不管是不在工作线程，都会持有 `Activity` 的引用，导致内存泄漏。

```
void scheduleTimer() {

    new Timer().schedule(new TimerTask() {

        @Override

        public void run() {

            while(true);

        }

    }, Long.MAX_VALUE >> 1);

}
```

```
View ttButton = findViewById(R.id.tt_button);

ttButton.setOnClickListener(new View.OnClickListener() {

    @Override public void onClick(View v) {

        scheduleTimer();

        nextActivity();

    }

});
```

com.nimbledroid.memoryleaks.MainActivity

Leak Size	Leak Reference Chain	Field
113 KB		
Class		
	↳ com.nimbledroid.memoryleaks.MainActivity\$12	this\$0
	↳ MainActivity	

Sensor Manager

最后，通过 `Context.getSystemService(int name)` 可以获取系统服务。这些服务工作在各自的进程中，帮助应用处理后台任务，处理硬件交互。如果需要使用这些服务，可以注册监听器，这会导致服务持有了 `Context` 的引用，如果在 `Activity` 销毁的时候没有注销这些监听器，会导致内存泄漏。

```
void registerListener() {

    SensorManager sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);

    Sensor sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ALL);

    sensorManager.registerListener(this, sensor, SensorManager.SENSOR_DELAY_FASTEST);

}
```

```
View smButton = findViewById(R.id.sm_button);

smButton.setOnClickListener(new View.OnClickListener() {

    @Override public void onClick(View v) {

        registerListener();

        nextActivity();

    }

});
```



2、如何防止内存泄漏？

我们讨论了八种容易发生内存泄漏的代码。其中，尤其严重的是泄漏 `Activity` 对象，因为它占用了大量系统内存。不管内存泄漏的代码表现形式如何，其核心问题在于：

在 `Activity` 生命周期之外仍持有其引用。
幸运的是，一旦泄漏发生且被定位到了，修复方法是相当简单的。

Static Activities

这种泄漏

```
private static MainActivity activity;

void setStaticActivity() {

    activity = this;
}
```

构造静态变量持有 `Activity` 对象很容易造成内存泄漏，因为静态变量是全局存在的，所以当 `MainActivity` 生命周期结束时，引用仍被持有。这种写法开发者是有理由来使用的，所以我们需要正确的释放引用让垃圾回收机制在它被销毁的同时将其回收。

Android 提供了特殊的 Set 集合

<https://developer.android.com/reference/java/lang/ref/package-summary.html#classes>

允许开发者控制引用的“强度”。Activity 对象泄漏是由于需要被销毁时，仍然被强引用着，只要强引用存在就无法被回收。

可以用弱引用代替强引用。

<https://developer.android.com/reference/java/lang/ref/WeakReference.html>.

弱引用不会阻止对象的内存释放，所以即使有弱引用的存在，该对象也可以被回收。

```
private static WeakReference<MainActivity> activityReference;

void setStaticActivity() {

    activityReference = new WeakReference<MainActivity>(this);

}
```

Static Views

静态变量持有 View

```
private static View view;

void setStaticView() {

    view = findViewById(R.id.sv_button);}
```

由于 View 持有其宿主 Activity 的引用，导致的问题与 Activity 一样严重。弱引用是个有效的解决方法，然而还有另一种方法是在生命周期结束时清除引用，Activity#onDestory() 方法就很适合把引用置空。

```
private static View view;

@Override public void onDestroy() {

    super.onDestroy();

    if (view != null) {

        unsetStaticView();
    }
}
```

```
    }

void unsetStaticView() {
    view = null;
}
```

Inner Class

这种泄漏

```
private static Object inner;

void createInnerClass() {

    class InnerClass {

    }

    inner = new InnerClass();
}
```

与上述两种情况相似，开发者必须注意少用非静态内部类，因为非静态内部类持有外部类的隐式引用，容易导致意料之外的泄漏。然而内部类可以访问外部类的私有变量，只要我们注意引用的生命周期，就可以避免意外的发生。

避免静态变量

这样持有内部类的成员变量是可以的。

```
private Object inner;

void createInnerClass() {

    class InnerClass {

    }

    inner = new InnerClass();
}
```

Anonymous Classes

前面我们看到的都是持有全局生命周期的静态成员变量引起的，直接或间接通过链式引用 `Activity` 导致的泄漏。这次我们用 `AsyncTask`

```
void startAsyncTask() {

    new AsyncTask<Void, Void, Void>() {
```

```
@Override protected Void doInBackground(Void... params) {  
  
    while(true);  
  
}  
  
.execute();}
```

Handler

```
void createHandler() {  
  
    new Handler() {  
  
        @Override public void handleMessage(Message message) {  
  
            super.handleMessage(message);  
  
        }  
  
.postDelayed(new Runnable() {  
  
        @Override public void run() {  
  
            while(true);  
  
        }  
  
}, Long.MAX_VALUE >> 1);}
```

Thread

```
void scheduleTimer() {  
  
    new Timer().schedule(new TimerTask() {  
  
        @Override  
  
        public void run() {  
  
            while(true);  
  
        }  
  
}, Long.MAX_VALUE >> 1);}
```

全部都是因为匿名类导致的。匿名类是特殊的内部类——写法更为简洁。当需要一次性特殊的子类时，Java 提供的语法糖能让表达式最少化。这种很赞很偷懒的写法容易导致泄漏。正如使用内部类一样，只要不跨越生命周期，内部类是完全没问题的。但是，这些类是用于产生后台线程的，这些 Java 线程是全局的，而且持有创建者的引用（即匿名类的引用），而匿名类又持有外部类的引用。线程是可能长时间运行的，所以一直持有 Activity 的引用导致当销毁时无法回收。这次我们不能通过移除静态成员变量解决，因为线程是于应用生命周期相关的。为了避免泄漏，我们必须舍弃简洁偷懒的写法，把子类声明为静态内部类。

静态内部类不持有外部类的引用，打破了链式引用。

所以对于 AsyncTask

```
private static class NimbleTask extends AsyncTask<Void, Void, Void> {

    @Override protected Void doInBackground(Void... params) {

        while(true);

    }

    void startAsyncTask() {

        new NimbleTask().execute();
    }
}
```

Handler

```
private static class NimbleHandler extends Handler {

    @Override public void handleMessage(Message message) {

        super.handleMessage(message);

    }

    private static class NimbleRunnable implements Runnable {

        @Override public void run() {

            while(true);

        }
    }

    void createHandler() {

        new NimbleHandler().postDelayed(new NimbleRunnable(), Long.MAX_VALUE
            >> 1);
    }
}
```

TimerTask

```
private static class NimbleTimerTask extends TimerTask {

    @Override public void run() {

        while(true);

    }

}

void scheduleTimer() {

    new Timer().schedule(new NimbleTimerTask(), Long.MAX_VALUE >> 1);
}
```

但是，如果你坚持使用匿名类，只要在生命周期结束时中断线程就可以。

```
private Thread thread;

@Override public void onDestroy() {

    super.onDestroy();

    if (thread != null) {

        thread.interrupt();

    }
}

void spawnThread() {

    thread = new Thread() {

        @Override public void run() {

            while (!isInterrupted()) {

            }

        }
    }

    thread.start();
}
```

Sensor Manager

这种泄漏

```
void registerListener() {  
  
    SensorManager sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);  
  
    Sensor sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ALL);  
  
    sensorManager.registerListener(this, sensor, SensorManager.SENSOR_DELAY_FASTEST);}
```

使用 Android 系统服务不当容易导致泄漏，为了 `Activity` 与服务交互，我们把 `Activity` 作为监听器，引用链在传递事件和回调中形成了。只要 `Activity` 维持注册监听状态，引用就会一直持有，内存就不会被释放。

在 `Activity` 结束时注销监听器

```
private SensorManager sensorManager; private Sensor sensor;  
  
@Override public void onDestroy() {  
  
    super.onDestroy();  
  
    if (sensor != null) {  
  
        unregisterListener();  
  
    }  
  
}  
  
void unregisterListener() {  
  
    sensorManager.unregisterListener(this, sensor);}
```

十五、Android 屏幕适配

1、屏幕适配相关名词解析

屏幕尺寸

屏幕尺寸指屏幕的对角线的长度，单位是英寸，1 英寸=2.54 厘米

比如常见的屏幕尺寸有 2.4、2.8、3.5、3.7、4.2、5.0、5.5、6.0 等

屏幕分辨率

屏幕分辨率是指在横纵向上的像素点数，单位是 px， $1px=1$ 个像素点。一般以纵向像素*横向像素，如 $1960*1080$ 。

屏幕像素密度

屏幕像素密度是指每英寸上的像素点数，单位是 dpi，即 “dot per inch”的缩写。屏幕像素密度与屏幕尺寸和屏幕分辨率有关，在单一变化条件下，屏幕尺寸越小、分辨率越高，像素密度越大，反之越小。

dp、dip、dpi、sp、px

px 我们应该是比较熟悉的，前面的分辨率就是用的像素为单位，大多数情况下，比如 UI 设计、Android 原生 API 都会以 px 作为统一的计量单位，像是获取屏幕宽高等。

dip 和 dp 是一个意思，都是 Density Independent Pixels 的缩写，即密度无关像素，上面我们说过，dpi 是屏幕像素密度，假如一英寸里面有 160 个像素，这个屏幕的像素密度就是 $160dpi$ ，那么在这种情况下，dp 和 px 如何换算呢？在 Android 中，规定以 $160dpi$ 为基准， $1dip=1px$ ，如果密度是 $320dpi$ ，则 $1dip=2px$ ，以此类推。

假如同样都是画一条 $320px$ 的线，在 $480*800$ 分辨率手机上显示为 $2/3$ 屏幕宽度，在 $320*480$ 的手机上则占满了全屏，如果使用 dp 为单位，在这两种分辨率下， $160dp$ 都显示为屏幕一半的长度。这也是为什么在 Android 开发中，写布局的时候要尽量使用 dp 而不是 px 的原因。

而 sp , 即 scale-independent pixels , 与 dp 类似 , 但是可以根据文字大小首选项进行放缩 , 是设置字体大小的御用单位。

mdpi、hdpi、xdpi、xxdpi

其实之前还有个 ldpi , 但是随着移动设备配置的不断升级 , 这个像素密度的设备已经很罕见了 , 所在现在适配时不需考虑。

mdpi、hdpi、xdpi、xxdpi 用来修饰 Android 中的 drawable 文件夹及 values 文件夹 , 用来区分不同像素密度下的图片和 dimen 值。

那么如何区分呢 ? Google 官方指定按照下列标准进行区分 :

名称	像素密度范围
mdpi	120dpi~160dpi
hdpi	160dpi~240dpi
xdpi	240dpi~320dpi
xxdpi	320dpi~480dpi
xxxdpi	480dpi~640dpi

在进行开发的时候 , 我们需要把合适大小的图片放在合适的文件夹里面。下面以图标设计为例进行介绍。

在设计图标时 , 对于五种主流的像素密度 (MDPI 、 HDPI 、 XHDPI 、 XXHDPI 和 XXXHDPI) 应按照 2:3:4:6:8 的比例进行缩放。例如 , 一个启动图标的尺寸为 48x48 dp , 这表示在 MDPI 的屏幕上其实际尺寸应为 48x48 px , 在 HDPI 的

屏幕上其实际大小是 MDPI 的 1.5 倍 (72x72 px) , 在 XDPI 的屏幕上其实际大小是 MDPI 的 2 倍 (96x96 px) , 依此类推。

虽然 Android 也支持低像素密度 (LDPI) 的屏幕 , 但无需为此费神 , 系统会自动将 HDPI 尺寸的图标缩小到 1/2 进行匹配。

下图为图标的各个屏幕密度的对应尺寸

屏幕密度	图标尺寸
mdpi	48x48px
hdpi	72x72px
xhdpi	96x96px
xxhdpi	144x144px
xxxhdpi	192x192px

2、现在流行的屏幕适配方式

传统 dp 适配方式的缺点

android 中的 dp 在渲染前会将 dp 转为 px , 计算公式 :

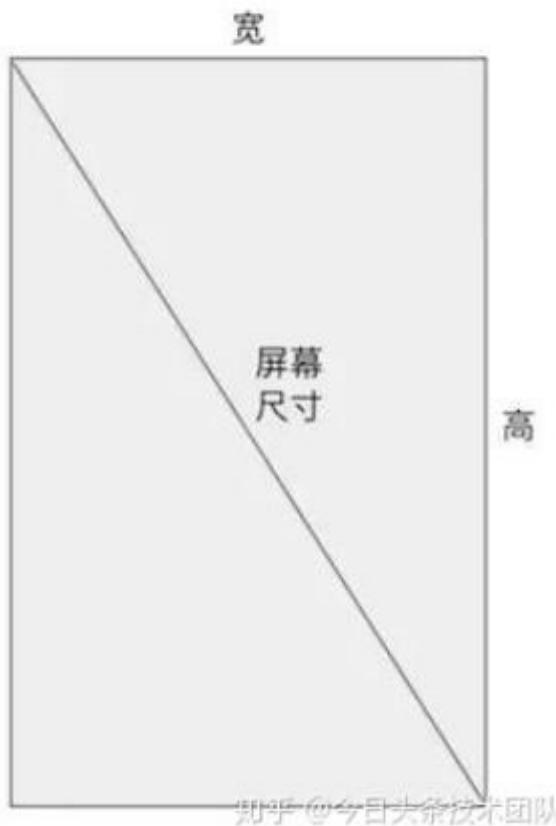
- $\text{px} = \text{density} * \text{dp};$
- $\text{density} = \text{dpi} / 160;$
- $\text{px} = \text{dp} * (\text{dpi} / 160);$

而 dpi 是根据屏幕真实的分辨率和尺寸来计算的 , 每个设备都可能不一样的。

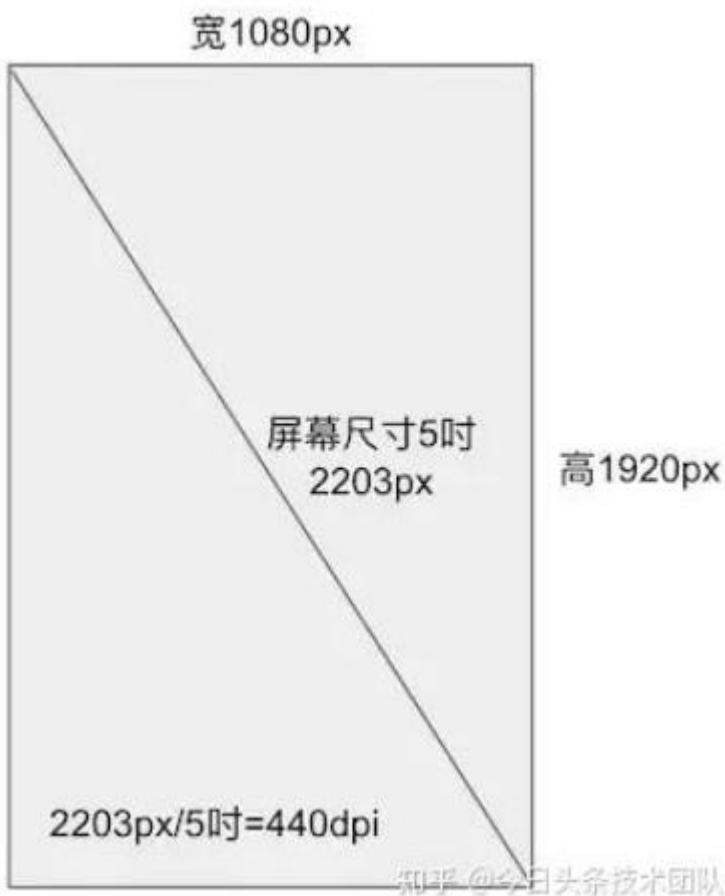
屏幕尺寸、分辨率、像素密度三者关系

通常情况下，一部手机的分辨率是宽 x 高，屏幕大小是以寸为单位，那么三者的关系是：

$$dpi = \frac{\sqrt{(宽^2 + 高^2) \text{ (单位 px)}}}{\text{屏幕尺寸} \text{ (单位 inch)}}$$



举个例子：屏幕分辨率为：1920*1080，屏幕尺寸为 5 吋的话，那么 dpi 为 440。



这样会存在什么问题呢？

假设我们 UI 设计图是按屏幕宽度为 360dp 来设计的，那么在上述设备上，屏幕宽度其实为 $1080/(440/160)=392.7\text{dp}$ ，也就是屏幕是比设计图要宽的。这种情况下，即使使用 dp 也是无法在不同设备上显示为同样效果的。同时还存在部分设备屏幕宽度不足 360dp，这时就会导致按 360dp 宽度来开发实际显示不全的情况。

而且上述屏幕尺寸、分辨率和像素密度的关系，很多设备并没有按此规则来实现，因此 dpi 的值非常乱，没有规律可循，从而导致使用 dp 适配效果差强人意。

探索新的适配方式

梳理需求

首先来梳理下我们的需求，一般我们设计图都是以固定的尺寸来设计的。比如以分辨率 1920px * 1080px 来设计，以 density 为 3 来标注，也就是屏幕其实是 640dp * 360dp。如果我们想在所有设备上显示完全一致，其实是不现实的，因为屏幕高宽比不是固定的，16:9、4:3 甚至其他宽高比层出不穷，宽高比不同，显示完全一致就不可能了。但是通常下，我们只需要以宽或高一个维度去适配，比如我们 Feed 是上下滑动的，只需要保证在所有设备中宽的维度上显示一致即可，再比如一个不支持上下滑动的页面，那么需要保证在高这个维度上都显示一致，尤其不能存在某些设备上显示不全的情况。同时考虑到现在基本都是以 dp 为单位去做的适配，如果新的方案不支持 dp，那么迁移成本也非常高。

因此，总结下大致需求如下：

- 支持以宽或者高一个维度去适配，保持该维度上和设计图一致；
- 支持 dp 和 sp 单位，控制迁移成本到最小。

找兼容突破口

从 dp 和 px 的转换公式 : $px = dp * density$

可以看出，如果设计图宽为 360dp，想要保证在所有设备计算得出的 px 值都正好是屏幕宽度的话，我们只能修改 density 的值。

通过阅读源码，我们可以得知，density 是 DisplayMetrics 中的成员变量，而 DisplayMetrics 实例通过 Resources#getDisplayMetrics 可以获得，而 Resources 通过 Activity 或者 Application 的 Context 获得。

先来熟悉下 DisplayMetrics 中和适配相关的几个变量：

- DisplayMetrics#density 就是上述的 density
- DisplayMetrics#densityDpi 就是上述的 dpi
- DisplayMetrics#scaledDensity 字体的缩放因子，正常情况下和 density 相等，但是调节系统字体大小后会改变这个值

那么是不是所有的 dp 和 px 的转换都是通过 DisplayMetrics 中相关的值来计算的呢？

首先来看看布局文件中 dp 的转换，最终都是调用 TypedValue#applyDimension(int unit, float value, DisplayMetrics metrics) 来进行转换：

```
public static float applyDimension(int unit, float value, DisplayMetrics metrics) {
    switch (unit) {
        case COMPLEX_UNIT_PX:
            return value;
        case COMPLEX_UNIT_DIP:
            return value * metrics.density;
        case COMPLEX_UNIT_SP:
            return value * metrics.scaledDensity;
        // ... 省略不相关的
    }
    return 0;
}
```

知乎 @今日头条技术团队

这里用到的 DisplayMetrics 正是从 Resources 中获得的。

再看看图片的 decode , BitmapFactory#decodeResourceStream 方法:

```
public static Bitmap decodeResourceStream(Resources res, TypedValue value,
    InputStream is, Rect pad, Options opts) {
    // ... 省略不相关的
    if (opts.inTargetDensity == 0 && res != null) {
        opts.inTargetDensity = res.getDisplayMetrics().densityDpi;
    }
    return decodeStream(is, pad, opts);
}
```

知乎 @今日头条技术团队

可见也是通过 DisplayMetrics 中的值来计算的。

当然还有些其他 dp 转换的场景，基本都是通过 DisplayMetrics 来计算的，这里不再详述。因此，想要满足上述需求，我们只需要修改 DisplayMetrics 中和 dp 转换相关的变量即可。

最终方案

下面假设设计图宽度是 360dp，以宽维度来适配。

那么适配后的 `density` = 设备真实宽(单位 px) / 360 , 接下来只需要把我们计算好的 `density` 在系统中修改下即可 , 代码实现如下 :

```
private static void setCustomDensity(@NonNull Activity activity, @NonNull Application application) {
    final DisplayMetrics appDisplayMetrics = application.getResources().getDisplayMetrics();
    final float targetDensity = appDisplayMetrics.widthPixels / 360;
    final int targetDensityDpi = (int) (160 * targetDensity);

    appDisplayMetrics.density = appDisplayMetrics.scaledDensity = targetDensity;
    appDisplayMetrics.densityDpi = targetDensityDpi;

    final DisplayMetrics activityDisplayMetrics = activity.getResources().getDisplayMetrics();
    activityDisplayMetrics.density = activityDisplayMetrics.scaledDensity = targetDensity;
    activityDisplayMetrics.densityDpi = targetDensityDpi;
}
```

知乎 @今日头条技术团队

同时在 `Activity#onCreate` 方法中调用下。代码比较简单 , 也没有涉及到系统非公开 api 的调用 , 因此理论上不会影响 app 稳定性。

于是修改后上线灰度测试了一版 , 稳定性符合预期 , 没有收到由此带来的 crash , 但是收到了很多字体过小的反馈 :

系统版本	设备	备注	反馈内容 (点击查看详细)
6.0.1	MI 4C	字体小	更新完后字体怎么这么大 , 跟老人机一样 , 看的很不舒服。
7	SM-C5010	字体小	升级后板面字变小啦!
5.1	HUAWEI TAG-TL	字体小	我想把评论的字调大点
7	WAS-AL00	字体小	字的尺寸。增加到最大 , 这样 , 我看着才真正舒服
5.1.1	OPPO A33	字体小	为什么没有字体大小及颜色设置呢? ? ?
7	WAS-AL00	字体小	各项目里 , 字的尺寸都太小 , 如果还不把字的尺寸变大 , 这就很反理性了
8.0.0	MHA-AL00	字体小	怎么不禁弯? 标题字号又都一样大小了 , 长标题显示不全!
7	PIC-AL00	字体小	字突然变小了 , 没法看了
8.0.0	MHA-AL00	字体小	这个就对了 , 上边视频标题字数多于比左边的字少 , 就显示不全了
5.1	VOTO GT11	字体小	西爪视频字体和图片一回大 , 一回小怎么回事

原因是在上面的适配中 , 我们忽略了 `DisplayMetrics#scaledDensity` 的 特殊性 , 将 `DisplayMetrics#scaledDensity` 和 `DisplayMetrics#density` 设置为同样的值 , 从而某些用户在系统中修改了字体大小失效了 , 但是我们还不能直接用原始的 `scaledDensity` , 直接用的话可能导致某些文字超过显示区域 , 因此我们可以通过计算之

前 scaledDensity 和 density 的比获得现在的 scaledDensity，方式如下：

```
final float targetScaledDensity = targetDensity * (appDisplayMetrics.scaledDensity / appDisplayMetrics.density);
```

但是测试后发现另外一个问题，就是如果在系统设置中切换字体，再返回应用，字体并没有变化。于是还得监听下字体切换，调用 Application#registerComponentCallbacks 注册下 onConfigurationChanged 监听即可。

因此最终方案如下：

```
private static float sNoncompatDensity;
private static float sNoncompatScaledDensity;

private static void setCustomDensity(@NotNull Activity activity, @NotNull final Application application) {
    final DisplayMetrics appDisplayMetrics = application.getResources().getDisplayMetrics();

    if (sNoncompatDensity == 0) {
        sNoncompatDensity = appDisplayMetrics.density;
        sNoncompatScaledDensity = appDisplayMetrics.scaledDensity;
        application.registerComponentCallbacks(new ComponentCallbacks() {
            @Override
            public void onConfigurationChanged(Configuration newConfig) {
                if (newConfig != null && newConfig.fontScale > 0) {
                    sNoncompatScaledDensity = application.getResources().getDisplayMetrics().scaledDensity;
                }
            }
            @Override
            public void onLowMemory() {
            }
        });
    }

    final float targetDensity = appDisplayMetrics.widthPixels / 360;
    final float targetScaledDensity = targetDensity * (sNoncompatScaledDensity / sNoncompatDensity);
    final int targetDensityDpi = (int) (160 * targetDensity);

    appDisplayMetrics.density = targetDensity;
    appDisplayMetrics.scaledDensity = targetScaledDensity;
    appDisplayMetrics.densityDpi = targetDensityDpi;

    final DisplayMetrics activityDisplayMetrics = activity.getResources().getDisplayMetrics();
    activityDisplayMetrics.density = targetDensity;
    activityDisplayMetrics.scaledDensity = targetScaledDensity;
    activityDisplayMetrics.densityDpi = targetDensityDpi;
}
```

知乎 @今日头条技术团队

当然以上代码只是以设计图宽 360dp 去适配的，如果要以高维度适配，可以再扩展下代码即可。

适配前后和设计图对比：



知乎 @今日头条技术团队

适配后各机型的显示效果：



知乎 @今日头条技术团队

十六、Android 缓存机制

LruCache 使用极其原理

一、Android 中的缓存策略

一般来说，缓存策略主要包含缓存的添加、获取和删除这三类操作。如何添加和获取缓存这个比较好理解，那么为什么还要删除缓存呢？这是因为不管是内存缓存还是硬盘缓存，它们的缓存大小都是有限的。当缓存满了之后，再想其添加缓存，这个时候就需要删除一些旧的缓存并添加新的缓存。

因此 LRU(Least Recently Used)缓存算法便应运而生，LRU 是近期最少使用的算法，它的核心思想是当缓存满时，会优先淘汰那些近期最少使用的缓存对象。采用

LRU 算法的缓存有两种：`LrhCache` 和 `DisLruCache`，分别用于实现内存缓存和硬盘缓存，其核心思想都是 LRU 缓存算法。

二、`LruCache` 的使用

`LruCache` 是 Android 3.1 所提供的一个缓存类，所以在 Android 中可以直接使用 `LruCache` 实现内存缓存。而 `DisLruCache` 目前在 Android 还不是 Android SDK 的一部分，但 Android 官方文档推荐使用该算法来实现硬盘缓存。

1. `LruCache` 的介绍

`LruCache` 是个泛型类，主要算法原理是把最近使用的对象用强引用（即我们平常使用的对象引用方式）存储在 `LinkedHashMap` 中。当缓存满时，把最近最少使用的对象从内存中移除，并提供了 `get` 和 `put` 方法来完成缓存的获取和添加操作。

2. `LruCache` 的使用

`LruCache` 的使用非常简单，我们就以图片缓存为例。

```
int maxMemory = (int) (Runtime.getRuntime().totalMemory()/1024);

int cacheSize = maxMemory/8;

mMemoryCache = new LruCache<String,Bitmap>(cacheSize){

    @Override

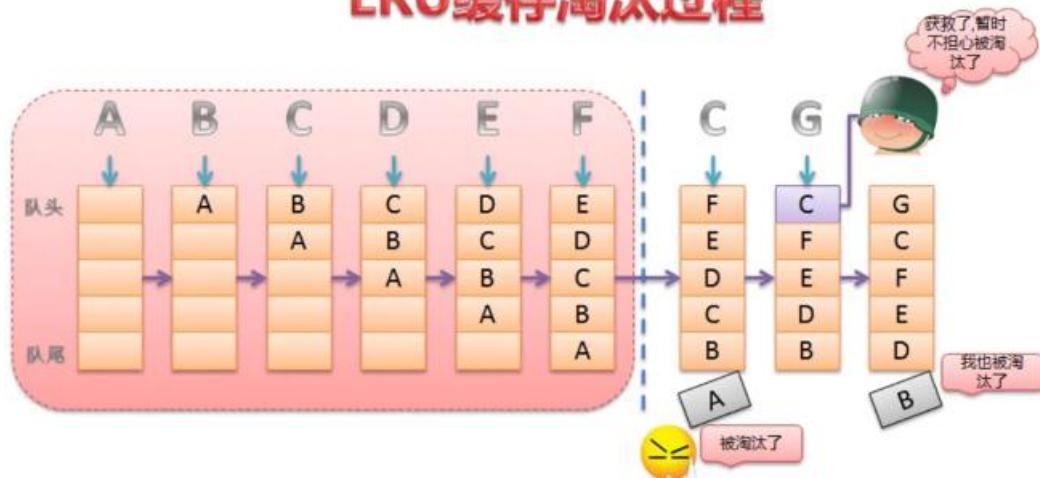
    protected int sizeOf(String key, Bitmap value) {

        return value.getRowBytes()*value.getHeight()/1024;

    }

};
```

LRU缓存淘汰过程



那么这个队列到底是由谁来维护的，前面已经介绍了是由 LinkedHashMap 来维护。

而 LinkedHashMap 是由数组+双向链表的数据结构来实现的。其中双向链表的结构可以实现访问顺序和插入顺序，使得 LinkedHashMap 中的<key,value>对按照一定顺序排列起来。

通过下面构造函数来指定 LinkedHashMap 中双向链表的结构是访问顺序还是插入顺序

```
public LinkedHashMap(int initialCapacity,  
                      float loadFactor,  
                      boolean accessOrder) {  
  
    super(initialCapacity, loadFactor);  
  
    this.accessOrder = accessOrder;  
}
```

其中 accessOrder 设置为 true 则为访问顺序，为 false，则为插入顺序。

以具体例子解释：

当设置为 true 时

```
public static final void main(String[] args) {  
  
    LinkedHashMap<Integer, Integer> map = new LinkedHashMap<>(0, 0.75  
f, true);
```

```
map.put(0, 0);

map.put(1, 1);

map.put(2, 2);

map.put(3, 3);

map.put(4, 4);

map.put(5, 5);

map.put(6, 6);

map.get(1);

map.get(2);

for (Map.Entry<Integer, Integer> entry : map.entrySet()) {

    System.out.println(entry.getKey() + ":" + entry.getValue());

}

}
```

输出结果

```
0:0
3:3
4:4
5:5
6:6
1:1
2:2
```

即最近访问的最后输出，那么这就正好满足的 LRU 缓存算法的思想。[可见 LruCache 巧妙实现，就是利用了 LinkedHashMap 的这种数据结构。](#)

下面我们在 LruCache 源码中具体看看，怎么应用 LinkedHashMap 来实现缓存的添加，获得和删除的。

```
public LruCache(int maxSize) {  
  
    if (maxSize <= 0) {  
  
        throw new IllegalArgumentException("maxSize <= 0");  
  
    }  
  
    this.maxSize = maxSize;  
  
    this.map = new LinkedHashMap<K, V>(0, 0.75f, true);  
  
}
```

从 LruCache 的构造函数中可以看到正是用了 LinkedHashMap 的访问顺序。

put()方法

```
public final V put(K key, V value) {  
  
    //不可为空，否则抛出异常  
  
    if (key == null || value == null) {  
  
        throw new NullPointerException("key == null || value == null");  
  
    }  
  
    V previous;  
  
    synchronized (this) {  
  
        //插入的缓存对象值加 1  
  
        putCount++;  
  
        //增加已有缓存的大小  
  
        size += safeSizeOf(key, value);  
  
    }  
  
    return previous;  
}
```

```
//向 map 中加入缓存对象

previous = map.put(key, value);

//如果已有缓存对象，则缓存大小恢复到之前

if (previous != null) {

    size -= safeSizeOf(key, previous);
}

//entryRemoved()是个空方法，可以自行实现

if (previous != null) {

    entryRemoved(false, key, previous, value);
}

//调整缓存大小(关键方法)

trimToSize(maxSize);

return previous;
}
```

可以看到 put()方法并没有什么难点，重要的就是在添加过缓存对象后，调用 trimToSize()方法，来判断缓存是否已满，如果满了就要删除近期最少使用的算法。

trimToSize()方法

```
public void trimToSize(int maxSize) {

    //死循环

    while (true) {

        K key;
```

```
V value;

synchronized (this) {

    //如果 map 为空并且缓存 size 不等于 0 或者缓存 size 小于 0, 抛出异常

    if (size < 0 || (map.isEmpty() && size != 0)) {

        throw new IllegalStateException(getClass().getName()

            + ".sizeOf() is reporting inconsistent results!");

    }

    //如果缓存大小 size 小于最大缓存, 或者 map 为空, 不需要再删除缓存对象, 跳出循环

    if (size <= maxSize || map.isEmpty()) {

        break;

    }

    //迭代器获取第一个对象, 即队尾的元素, 近期最少访问的元素

    Map.Entry<K, V> toEvict = map.entrySet().iterator().next

();

    key = toEvict.getKey();

    value = toEvict.getValue();

    //删除该对象, 并更新缓存大小

    map.remove(key);

    size -= safeSizeOf(key, value);

    evictionCount++;

}

entryRemoved(true, key, value, null);
```

```
}
```

```
}
```

trimToSize()方法不断地删除 LinkedHashMap 中队尾的元素，即近期最少访问的，直到缓存大小小于最大值。

当调用 LruCache 的 get()方法获取集合中的缓存对象时，就代表访问了一次该元素，将会更新队列，保持整个队列是按照访问顺序排序。这个更新过程就是在 LinkedHashMap 中的 get()方法中完成的。

先看 LruCache 的 get()方法

get()方法

```
public final V get(K key) {  
  
    //key 为空抛出异常  
  
    if (key == null) {  
  
        throw new NullPointerException("key == null");  
  
    }  
  
    V mapValue;  
  
    synchronized (this) {  
  
        //获取对应的缓存对象  
  
        //get()方法会实现将访问的元素更新到队列头部的功能  
  
        mapValue = map.get(key);  
  
        if (mapValue != null) {  
  
            hitCount++;  
  
            return mapValue;  
  
        }  
  
        missCount++;  
  
    }  
}
```

其中 LinkedHashMap 的 get()方法如下：

```
public V get(Object key) {  
  
    LinkedHashMapEntry<K,V> e = (LinkedHashMapEntry<K,V>)getEntry(key);  
  
    if (e == null)  
  
        return null;  
  
    //实现排序的关键方法  
  
    e.recordAccess(this);  
  
    return e.value;  
  
}
```

调用 recordAccess()方法如下：

```
void recordAccess(HashMap<K,V> m) {  
  
    LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;  
  
    //判断是否是访问排序  
  
    if (lm.accessOrder) {  
  
        lm.modCount++;  
  
        //删除此元素  
  
        remove();  
  
        //将此元素移动到队列的头部  
  
        addBefore(lm.header);  
  
    }  
  
}
```

由此可见 LruCache 中维护了一个集合 LinkedHashMap，该 LinkedHashMap 是以访问顺序排序的。当调用 put()方法时，就会在集合中添加元素，并调用

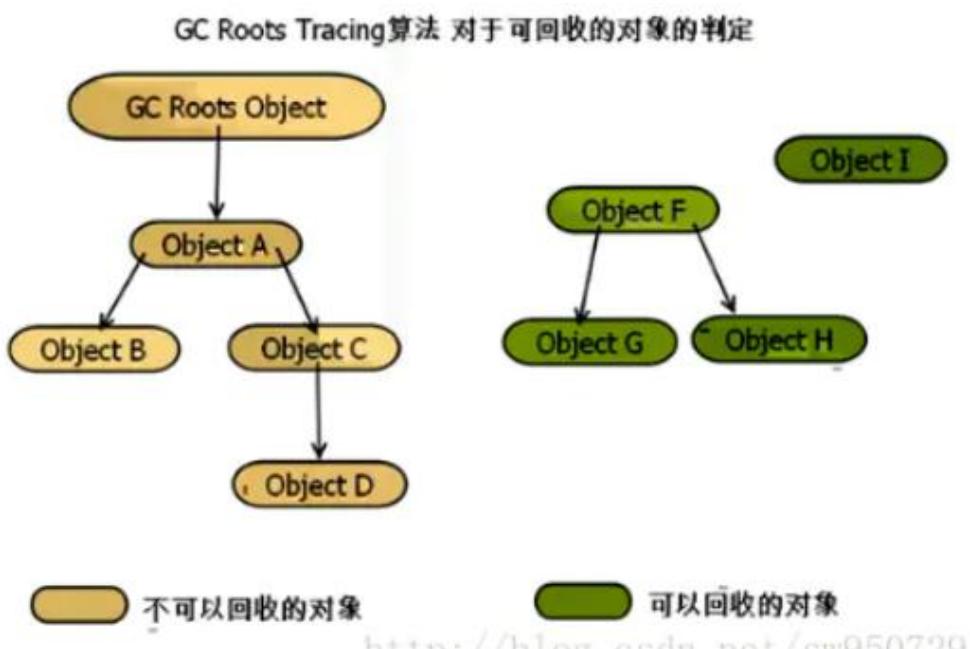
`trimToSize()` 判断缓存是否已满，如果满了就用 `LinkedHashMap` 的迭代器删除队尾元素，即近期最少访问的元素。当调用 `get()` 方法访问缓存对象时，就会调用 `LinkedHashMap` 的 `get()` 方法获得对应集合元素，同时会更新该元素到队头。

十七、Android 性能优化

1、如何进行 内存 cpu 耗电 的定位以及优化

内存优化

关于性能优化我们可以不知道其他的，但一定要知道内存优化。因为内存泄漏可以是 Android 的常客。那么什么是内存泄漏呢？内存不在 GC 的掌控范围之内了。那么 Java 的 GC 内存回收机制是什么？某对象不在有任何引用的时候才会进行回收。那么 GC 回收机制的原理是什么？又或者说可以作为 GC Root 引用点的是啥？或许有人听不懂我在讲啥。我们先来看张图。



当我们向上寻找，一直寻找到 GC Root 的时候，此对象不会进行回收，例如，一个 Activity。那么如果我们向上寻找，直到找到 GC Root 对象的时候，就说明它是不可以回收的，例如，我定义了一个 int a；但是这个数据，我整个页面或者说整个项目都没有用到，则这个对象会被 GC 掉。

GC 的引用点

1.

java 栈中引用的对象

2.

3.

方法静态引用的对象

4.

5.

方法常量引用的对象

6.

7.

Native 中 JNI 引用的对象

8.

9.

Thread——“活着的”线程

10.

如何判断

那么我们如何判断一个对象是一个垃圾对象，可以讲他进行回收呢？举了小例子教你们如何区分：

一般在学校吃饭，我们有两种情况，第一：吃完饭就直接走人，碗筷留给阿姨来收拾处理。
第二：吃完之后把碗筷放到收盘处直接进行回收。

但我们是个有素质的人，一般采用第二种情况，但根据想法，我们更倾向于第一种。

那么一般在饭店或者 KFC 中，都是第一种情况。

那么此时，问题来了，如果我已经吃完饭，然后我并没有离开饭店，坐在位置上和朋友吹牛逼，谈谈理想，聊聊人生。

那么桌上那一堆碗筷是收还是不收？讲道理是不能收的。虽然实际也是不能收的。因为顾客是上帝~~~

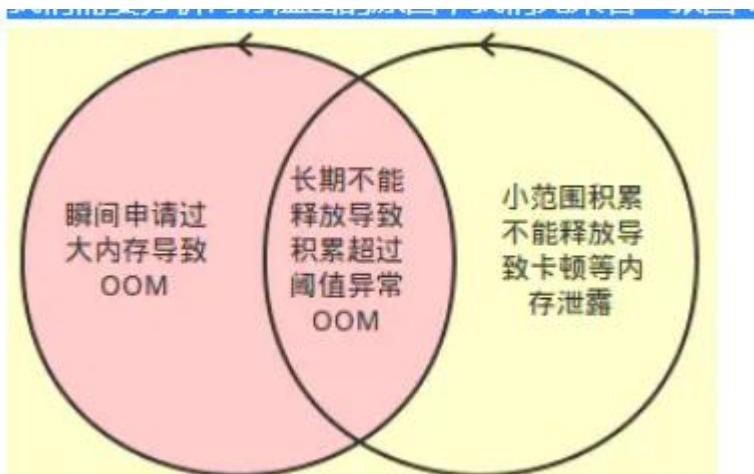
一般在学校吃饭，我们有两种情况，第一：吃完饭就直接走人，碗筷留给阿姨来收拾处理。
第二：吃完之后把碗筷放到收盘处直接进行回收。

但我们是个有素质的人，一般采用第二种情况，但根据想法，我们更倾向于第一种。

那么一般在饭店或者 KFC 中，都是第一种情况。

那么此时，问题来了，如果我已经吃完饭，然后我并没有离开饭店，坐在位置上和朋友吹牛逼，谈谈理想，聊聊人生。

那么桌上那一堆碗筷是收还是不收？讲道理是不能收的。虽然实际也是不能收的。因为顾客是上帝~~~



<http://blog.csdn.net/sw950729>

内存泄漏一般导致应用卡顿，极端情况会导致项目 boom。Boom 的原因是因为超过内存的阈值。原因主要有两方面：

•

代码存在泄漏，内存无法及时释放导致 oom（这个我们后面说）

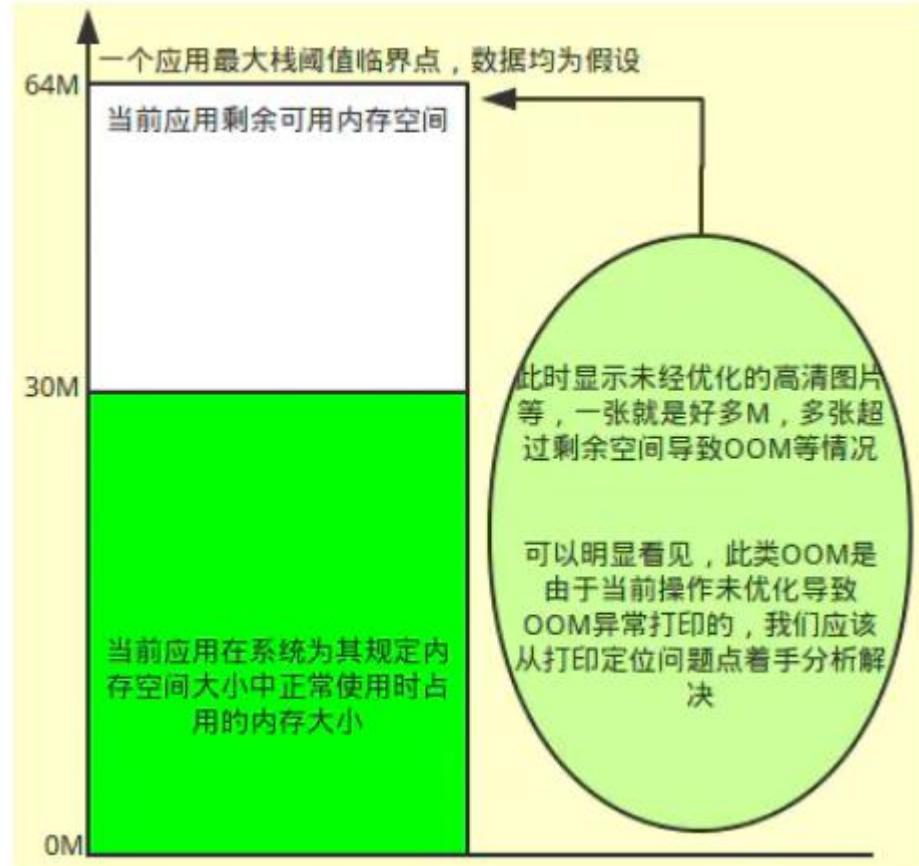
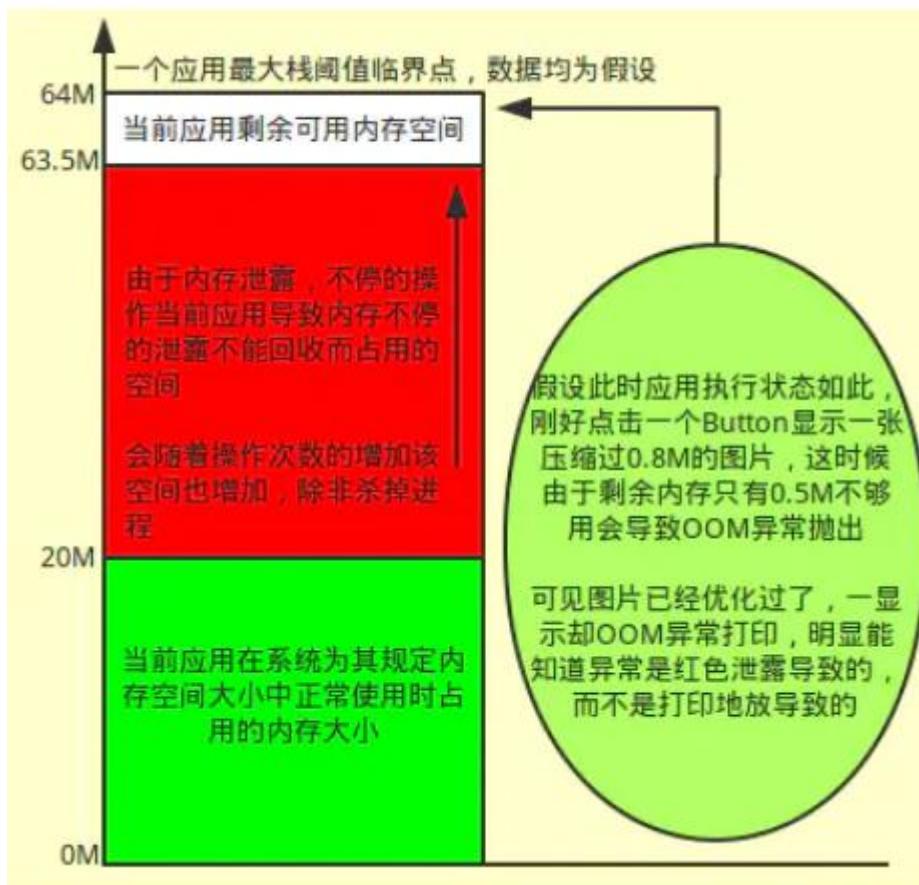
•

•

一些逻辑消耗了大量内存，无法及时释放或者超过导致 oom

•

所谓消耗大量的内存的，绝大多数是因为图片加载。这是我们 oom 出现最频繁的地方。我前面有写过图片加载的方法，一个是控制每次加载的数量，第二，保证每次滑动的时候不进行加载，滑动完进行加载。一般情况使用先进后出，而不是先进先出。不过一般我们图片加载都是使用 fresco 或者 Glide 等开源库。我们来看下下面两张图：



对比两张图，我们可以在第一张的情况出现了 oom 情况，我们通过 log 打印发现，处理的好像没什么问题，换句话说，如果我不放那 0.8M 的图片。然后继续不停的操作同样会出现 OOM，然而我们就蒙了。没什么图片加载怎么就这么崩掉了。

如何查看

首先，我们确定我们项目或者某几个类里面是否存在内存溢出的问题。我们可以通过如下方法：

•

Android-->System Information-->MemoryUsage 查看 Object 里面是否有没有被释放的 Views 和 Activity

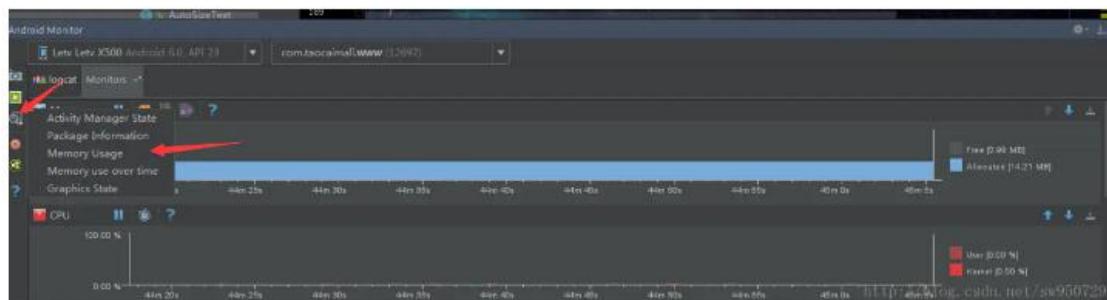
•

•

命令行模式：adb shell dumpsys meminfo 包名 -d

•

就那我公司的项目举例把。首先，我们在这边可以看到 memory。CPU 和 net 的使用情况。



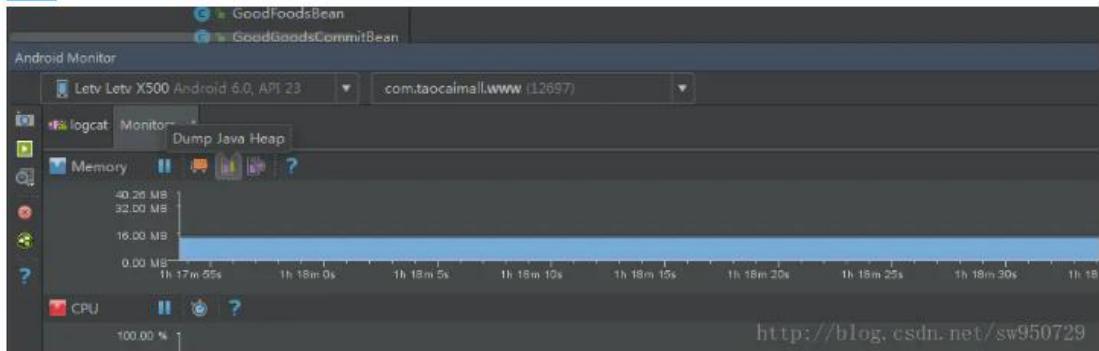
我们找到 Object。看看我们内存的消耗情况。



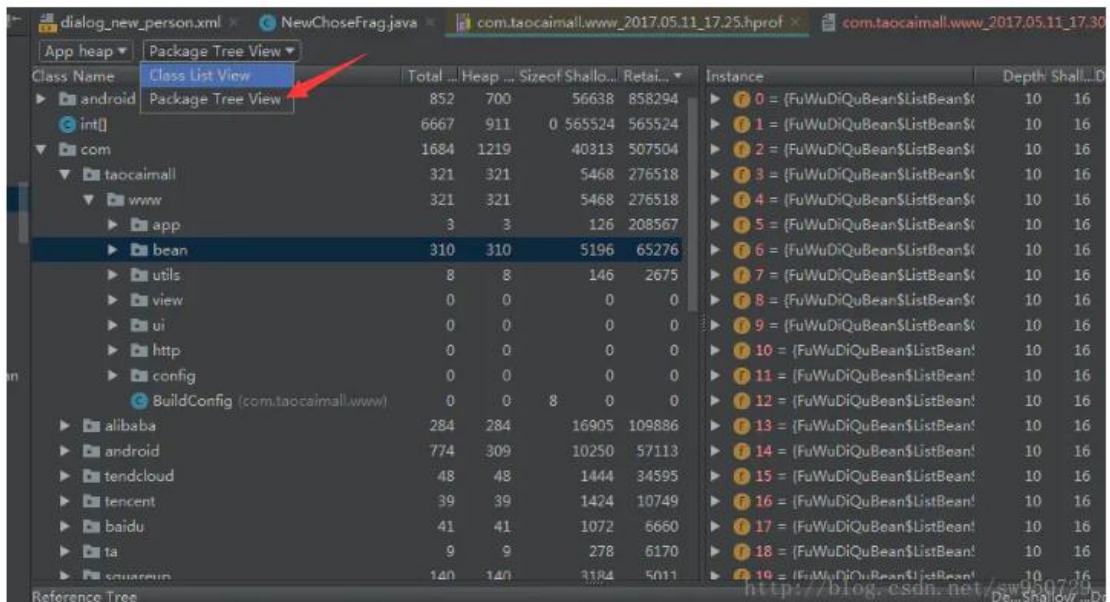
随便这么一看，尼玛蛋，1300 左右的 view 和一个 Activity。还有 3 个 context。可怕。。。可以理解为一个 Activity 里面使用了将近 1300 个 view。。。想都不敢想。。。

我们可以通过看 Memory Monitor 工具。 检查一个一个的动作。（比如 Activity 的跳转）。反复多次执行某一个操作，不断的通过这个工具查看内存的大概变化情况。 前后两个内存变化增加了不少。

我们可以更仔细的查找泄漏的位置，在 AS 里面使用 Heap SnapShot 工具（堆栈快照）。如图所示：



我们点击后，他会进行一段时间的监控，然后会生成一个文件。我们点击我们 package tree view。



我们找到自己项目的包名。然后进行进一步的分析。首先看一下 2 个列表的列名到底指的什么。

名称	意义
Total Count	内存中该类的对象个数
Heap Count	堆内存中该类的对象个数
Sizeof	物理大小
Shallow size	该对象本身占有内存大小
Retained Size	释放该对象后，节省的内存大小

我们来随便的看一下内存中的数量

bean	310	310	5196	65276
FuWuDiQuBean\$ListBean (com.ti	6	6	20	120 23744
FuWuDiQuBean\$ListBean\$ChildB	31	31	20	620 22876
FuWuDiQuBean\$ListBean\$ChildB	272	272	16	4352 18512

内存分析工具

性能优化工具：

•

Heap SnapShot 工具

•

•

Heap Viewer 工具

•

•

LeakCanary 工具

•

•

MAT 工具

•

•

TraceView 工具 (Device Monitor)

•

第三方分析工具：

•

MemoryAnalyzer

•

•

GT Home

-
-

iTest

-

因为我没有这些工具，无法进行演示。

注意事项

-

我们尽量不要使用 `Activity` 的上下文，而是使用 `application` 的上下文，因为 `application` 的生命周期长，进程退出时才会被销毁。所以，单例模式是最容易造成内存溢出的原本所在，因为单例模式的生命周期的应该和 `application` 的生命周期一样长，而不是和 `Activity` 的相同。

-
-

`Animation` 也会导致内存溢出，为什么？因为我们是通过 `view` 来进行演示的，导致 `view` 被 `Activity` 持有，而 `Activity` 又持有 `view`。最后因为 `Activity` 无法释放，导致内存泄漏。解决方法是在 `Activity` 的 `onDestory()` 方法中调用 `Animation.cancel()` 进行停止，当然一些简单的动画我们可以通过自定义 `view` 来解决。至少我现在已经很少使用 `Animation` 了。没有一个动画是自定义 `view` 解决不了的。如何有，那就是两个~~~。

UI 优化

UI 优化主要包括布局优化以及 `view` 的绘制优化。不急，我们接下来一个一个慢慢看~~。先说下 UI 的优化到底是什么？有些时候我们打开某个软件，会出现卡顿的情况。这就是 UI 的问题。那么我们想一下，什么情况会导致卡顿呢？一般是如下几种情况：

1.

人为在 UI 线程中做轻微耗时操作，导致 UI 线程卡顿；

2.

3.

布局 Layout 过于复杂，无法在 16ms 内完成渲染；

4.

5.

同一时间动画执行的次数过多，导致 CPU 或 GPU 负载过重；

6.

7.

View 过度绘制，导致某些像素在同一帧时间内被绘制多次，从而使 CPU 或 GPU 负载过重；

8.

9.

View 频繁的触发 measure、layout，导致 measure、layout 累计耗时过多及整个 View 频繁的重新渲染；

10.

11.

内存频繁触发 GC 过多（同一帧中频繁创建内存），导致暂时阻塞渲染操作；

12.

13.

冗余资源及逻辑等导致加载和执行缓慢；

14.

15.

臭名昭著的 ANR；

16.

可以看见，上面这些导致卡顿的原因都是我们平时开发中非常常见的。有些人可能会觉得自己的应用用着还蛮 OK 的，其实那是因为你没进行一些瞬时测试和压力测试，一旦在这种环境下运行你的 App 你就会发现很多性能问题。

布局优化

GPU 绘制

我们对于 UI 性能的优化还可以通过开发者选项中的 GPU 过度绘制工具来进行分析。在设置->开发者选项->调试 GPU 过度绘制（不同设备可能位置或者叫法不同）中打开调试后可以看见如下图（对 settings 当前界面过度绘制进行分析）：



调试 GPU 过度绘制

-
- 关闭**: Off, switch is off.
 - 显示过度绘制区域**: Show overdraw regions, switch is on (indicated by a checked checkbox).
 - 显示适合绿色弱视患者查看的区域**: Show regions suitable for colorblind users, switch is off.

<http://blog.csdn.net/sw950729> 取消



这图看着太乱，我们来一张简洁明了的图：



我们的目标就是尽量减少红色 Overdraw，看到更多的蓝色区域。

可以发现，开启后在我们想要调试的应用界面中可以看到各种颜色的区域，具体含义如下：

颜色	含义
无色	WebView等的渲染区域
蓝色	1x过度绘制
绿色	2x过度绘制
淡红色	3x过度绘制
红色	4x(+)过度绘制

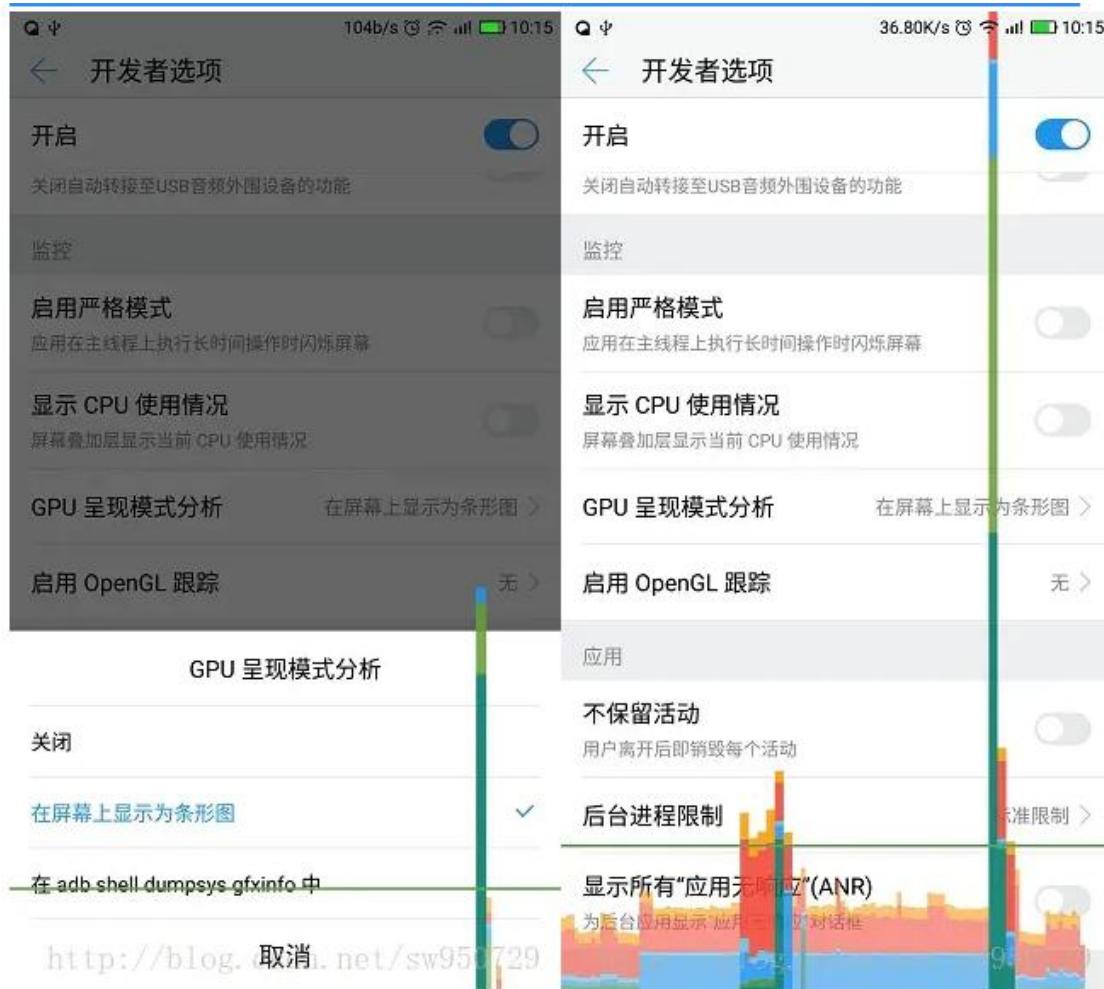
Overdraw 有时候是因为你的 UI 布局存在大量重叠的部分，还有的时候是因为非必须的重叠背景。例如某个 Activity 有一个背景，然后里面的 Layout 又有自己的背景，同时子 View 又分别有自己的背景。仅仅是通过移除非必须的背景图片，这就能够减少大量的红色 Overdraw 区域，增加蓝色区域的占比。这一措施能够显著提升程序性能。

如果布局中既能采用 RelativeLayout 和 LinearLayout，那么直接使用 LinearLayout，因为 RelativeLayout 的布局比较复杂，绘制的时候需要花费更多的 CPU 时间。如

果需要多个 LinearLayout 或者 FrameLayout 嵌套，那么可采用 Relativelayout。因为多层嵌套导致布局的绘制有大部分是重复的，这会减少程序的性能。

GPU 呈现模式分析

我们依旧打开设置-->开发者选项-->GPU 呈现模式分析-->在屏幕上显示为条形图，如图所示：



当然，也可以在执行完 UI 滑动操作后在命令行输入如下命令查看命令行打印的 GPU 渲染数据（分析依据：Draw + Process + Execute = 完整的显示一帧时间 < 16ms）：
adb shell dumpsys gfxinfo [应用包名]

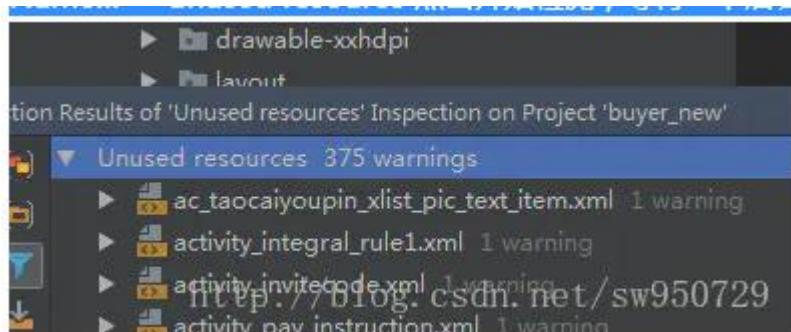
随着界面的刷新，界面上会以实时柱状图来显示每帧的渲染时间，柱状图越高表示渲染时间越长，每个柱状图偏上都有一根代表 16ms 基准的绿色横线，每一条竖着的柱状线都包含三部分（蓝色代表测量绘制 Display List 的时间，红色代表 OpenGL 渲染 Display List 所需要的时间，黄色代表 CPU 等待 GPU 处理的时间），只要我们每一帧的总时间低于基准线就不会发生 UI 卡顿问题（个别超出基准线其实也不算啥问题的）。就简单的看下我们公司项目刚启动的时候：



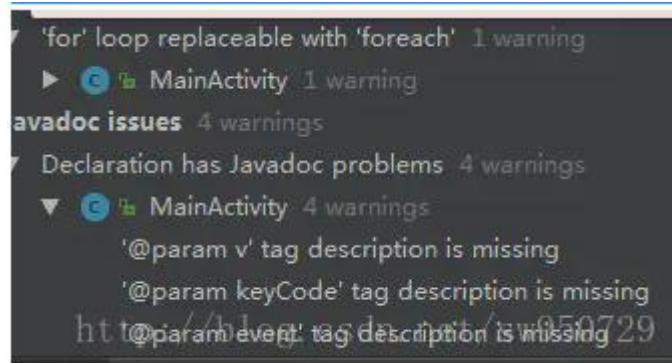
突然就有那么一种想吐槽的感觉.....我记得之前我做了瘦身的优化，但是要让我做性能优化，我觉得应该没那么简单.....

代码优化

Android Studio 和 IntelliJ idea 都有自带的代码检查工具。打开 Analyze->Run Inspection by Name... ->unused resource 点击开始检测，等待一下后会发现如下结果：



我们还可以这样，将鼠标放在代码区点击右键->Analyze->Inspect Code->界面选择你要检测的模块->点击确认开始检测，等待一下后会发现如下结果：



当然，我这只是截取了少一部分，我们看下下面那个提示：@param v tag description is missing 。意味着 v 的类型缺少了，要么补上介绍，要么直接删除。上面那两种方法是最容易找到代码缺陷以及无用代码的地方。所以尽情的入坑去填坑把~~~

绘制优化

那么什么是绘制优化？绘制优化主要是指 View 的 OnDraw 方法需要避免执行大量的操作。我将分为了 2 个方面。

•

OnDraw 方法不需要创建新的局部对象，这是因为 OnDraw 方法是实时执行的，这样会产品大量的临时对象，导致占用了更多内存，并且使系统不断的 GC。降低了执行效率。

•

•

OnDraw 方法不需要执行耗时操作，在 OnDraw 方法里少使用循环，因为循环会占用 CPU 的时间。导致绘制不流畅，卡顿等等。Google 官方指出，

`view` 的绘制帧率稳定在 `60dps`, 这要求每帧的绘制时间不超过 `16ms` ($1000/60$)。虽然很难保证, 但我们需要尽可能的降低。

•

`60dps` 是目前最合适的图像显示速度, 也是绝大部分 `Android` 设备设置的调试频率, 如果在 `16ms` 内顺利完成界面刷新操作可以展示出流畅的画面, 而由于任何原因导致接收到 `VSYNC` 信号的时候无法完成本次刷新操作, 就会产生掉帧的现象, 刷新帧率自然也就跟着下降(假定刷新帧率由正常的 `60fps` 降到 `30fps`, 用户就会明显感知到卡顿)。So, 前面我们说 `GPU` 的时候也谈到了这个。总的而言, 感觉还是蛮重要的.....

电量优化

有了 `UI` 优化、内存优化、代码优化、网络优化之后我们在来说说应用开发中很重要的一个优化模块——电量优化。

耗电概念

其实大多数开发者对电量优化的重视程度极低, 其实提到性能优化想到的就是内存优化, 但我们不能忽视其他的优化, 电量优化其实还是必要的, 例如爱奇艺、优酷等等的视频播放器以及音乐播放器。众所周知, 音乐和视频其实是耗电量最大的。如果用户一旦发现我们的应用非常耗电, 不好意思, 他们大多会选择卸载来解决此类问题。为此, 我们需要进行优化。

如何优化

其实我们把上面那四种优化解决了, 就是最好的电量优化。So, 对于电量优化, 我在此提一些建议:

需要进行网络请求时, 我们需先判断网络当前的状态。

在多网络请求的情况下, 最好进行批量处理, 尽量避免频繁的间隔网络请求。

在同时有 `wifi` 和移动数据的情况下, 我们应该直接屏幕移动数据的网络请求, 只有当 `wifi` 断开时在调用, 因为, `wifi` 请求的耗电量远比移动数据的耗电量低的低。

后台任务要尽可能少的唤醒 `CPU`。(比方说, 锁屏时, `QQ` 的消息提示行就是唤醒了 `CPU`。但是它的提示只有在你打开锁屏或者进行充电时才会进行提示。)

2、性能优化经常使用的方法

1、布局优化

和 UI 相关的首先就是布局，特别是在开发一些复杂界面的时候，通常我们都是采用布局嵌套的方法，每个人的布局思路不太一样，写出的也不太一样，，所以就可能造成嵌套的层级过多。

官方 屏幕上的某个像素在同一帧的时间内被绘制了多次。在多层次的 UI 结构里面，如果不可见的 UI 也在做绘制的操作，这就会导致某些像素区域被绘制了多次。这就浪费大量的 CPU 以及 GPU 资源。

白话 显示一个布局就好比我们盖一个房子，首先我们要测量房子的大小，还要测量房间里面各个家具的大小，和位置，然后进行摆放同时也要对房子进行装修，如果我们是一层，都在明面上，干起活来敞亮也轻松，可是有的人的房子，喜欢各种隔断，分成一个一个的大隔断间，每个大隔断间里还有小隔断间，小隔断间里有小小隔断间，还有小小小隔断间。。。N 层隔断间。

看到这些头皮发麻吧，而且是一个大隔断间里面所有的小隔断，小小隔断等等都测量完摆放好，才能换另外一个大隔断，天呢，太浪费时间了，不能都直接都放外面吗？也好摆放啊，这么搞我怎么摆，每个隔断间都要装修一遍，太浪费时间了啊。

我们的 Android 虚拟机也会这么抱怨，咱们家本来就不富裕，什么都要省着用，你这么搞，肯定运转有问题啊，那么多嵌套的小隔断间需要处理，都会占用 cpu 计算的时间和 GPU 渲染的时间。显示 GPU 过度绘制，分层如下如所示：



通过颜色我们可以知道我们应用是否有多余层次的绘制，如果一路飘红，那么我们就要相应的处理了。

所以我们有了第一个优化版本：

优化 1.0

1. 如果父控件有颜色，也是自己需要的颜色，那么就不必在子控件加背景颜色
2. 如果每个自控件的颜色不太一样，而且可以完全覆盖父控件，那么就不需要再父控件上加背景颜色
3. 尽量减少不必要的嵌套
4. 能用 `LinearLayout` 和 `FrameLayout`，就不要用 `RelativeLayout`，因为 `RelativeLayout` 控件相对比较复杂，测绘也想要耗时。

做到了以上 4 点只能说恭喜你，入门级优化已经实现了。

针对嵌套布局，谷歌也是陆续出了一些新的方案。对就是 `include`、`merge` 和 `ViewStub` 三兄弟。

`include` 可以提高布局的复用性，大大方便我们的开发，有人说这个没有减少布局的嵌套吧，对，`include` 确实没有，但是 `include` 和 `merge` 联手搭配，效果那是杠杠滴。

`merge` 的布局取决于父控件是哪个布局，使用 `merge` 相当于减少了自身的一层布局，直接采用父 `include` 的布局，当然直接在父布局里面使用意义不大，所以会和 `include` 配合使用，既增加了布局的复用性，用减少了一层布局嵌套。

`ViewStub` 它可以按需加载，什么意思？用到他的时候喊他一下，再来加载，不需要的时候像空气一样，在一边静静的呆着，不吃你的米，也不花你家的钱。等需要的时候 `ViewStub` 中的布局才加载到内存，多节俭持家啊。对于一些进度条，提示信息等等八百年才用一次的功能，使用 `ViewStub` 是极其合适的。这就是不用不知道，一用戒不了。

我们开始进化我们的优化

优化 1.1

1. 使用 `include` 和 `merge` 增加复用，减少层级
2. `ViewStub` 按需加载，更加轻便

可能又有人说了：背景复用了，嵌套已经很精简了，再精简就实现了不了复杂视图了，可是还是一路飘红，这个怎么办？面对这个问题谷歌给了我们一个新的布局 `ConstraintLayout`。

`ConstraintLayout` 可以有效地解决布局嵌套过多的问题。`ConstraintLayout` 使用约束的方式来指定各个控件的位置和关系的，它有点类似于 `RelativeLayout`，但远比 `RelativeLayout` 要更强大(照抄隔壁 iOS 的约束布局)。所以简单布局简单处理，复杂布局 `ConstraintLayout` 很好使，提升性能从布局做起。

再次进化：

优化 1.2

1. 复杂界面可选择 `ConstraintLayout`，可有效减少层级

2、绘制优化

我们把布局优化了，但是和布局息息相关的还有绘制，这是直接影响显示的两个根本因素。

其实布局优化了对于性能提升影响不算很大，但是是我们最容易下手，最直接接触的优化，所以不管能提升多少，哪怕只有百分之一的提升，我们也要做，因为影响性能的地方太多了，每个部分都提升一点，我们应用就可以提升很多了。

我们平时感觉的卡顿问题最主要的原因之一是因为渲染性能，因为越来越复杂的界面交互，其中可能添加了动画，或者图片等等。我们希望创造出越来越炫的交互界面，同时也希望他可以流畅显示，但是往往卡顿就发生在这里。

这个是 Android 的渲染机制造成的，Android 系统每隔 16ms 发出 VSYNC 信号，触发对 UI 进行渲染，但是渲染未必成功，如果成功了那么代表一切顺利，但是失败了可能就要延误时间，或者直接跳过去，给人视觉上的表现，就是要么卡了一会，要么跳帧。

View 的绘制频率保证 60fps 是最佳的，这就要求每帧绘制时间不超过 16ms($16ms = 1000/60$)，虽然程序很难保证 16ms 这个时间，但是尽量降低 `onDraw` 方法中的复杂度总是切实有效的。

这个正常情况下，每隔 16ms `draw()`一下，很整齐，很流畅，很完美。



往往会发生如下图的情况，有个便秘的家伙霸占着，一帧画面拉的时间那么长，这一下可不就卡顿了嘛。把后面的时间给占用了，后面只能延后，或者直接略过了。



既然问题找到了，那么我们肯定要有相应的解决办法，根本做法是 减轻 `onDraw()` 的负担。

所以

第一点： `onDraw` 方法中不要做耗时的任务，也不做过多的循环操作，特别是嵌套循环，虽然每次循环耗时很小，但是大量的循环势必霸占 CPU 的时间片，从而造成 View 的绘制过程不流畅。

第二点： 除了循环之外，`onDraw()` 中不要创建新的局部对象，因为 `onDraw()` 方法一般都会频繁大量调用，就意味着会产生大量的零时对象，不进占用过的内存，而且会导致系统更加频繁的 GC，大大降低程序的执行速度和效率。

其实这两点在 android 的 UI 线程中都适用。

升级进化：

优化 2.0

1. `onDraw` 中不要创建新的局部对象
2. `onDraw` 方法中不要做耗时的任务

其实从渲染优化里我们也牵扯出了另一个优化，那就是内存优化。

3、内存优化

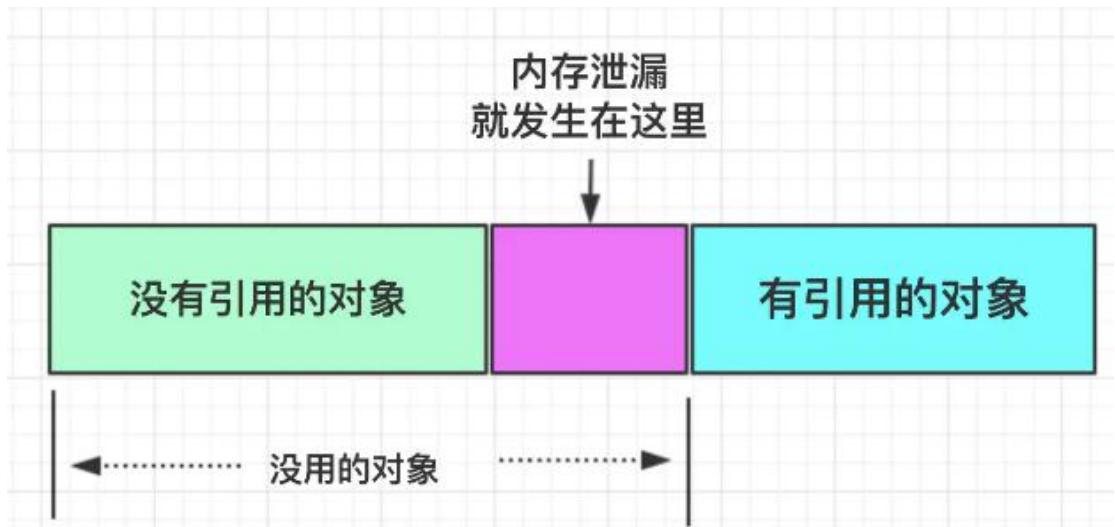
内存泄漏指的是那些程序不再使用的对象无法被 GC 识别，这样就导致这个对象一直留在内存当中，占用了没来就多的内存空间。

内存泄漏是一个缓慢积累的过程，一点一点的给你，温水煮青蛙一般，我们往往很难直观的看到，只能最后内存不够用了，程序奔溃了，才知道里面有大量的泄漏，但是到底是那些地方？估计是狼烟遍地，千疮百孔，都不知道如何下手。怎么办？最让人难受的是内存泄漏情况那么多，记不住，理解也不容易，关键是老会忘记。怎么办呢？老这么下去也不是事，总不能面试的时候突击，做项目的时候不知所措吧。所以一定要记住了解 GC 原理，这样才可以更准确的理解内存泄漏的场景和原因。不懂 GC 原理的可以先看一下这个 JVM 初探：内存分配、GC 原理与垃圾收集器

本来 GC 的诞生是为了让 java 程序员更加轻松（这一点隔壁 C++ 痛苦的一匹），java 虚拟机会自动帮助我们回收那些不再需要的内存空间。通过引用计数法，可达性分析法等等方法，确认该对象是否没有引用，是否可以被回收。

有人说真么强悍的功能看起来无懈可击啊，对，理论上可以达到消除内存泄漏，但是很多人不按常理出牌啊，往往很多时候，有的对象还保持着引用，但逻辑上已经不会再用到。就是这一类对象，游走于 GC 法律的边缘，我没用了，但是你又不知道我没用了，就是这么赖着不走，空耗内存。

因为有内存泄漏，所以内存被占用越来越多，那么 GC 会更容易被触发，GC 会越来越频发，但是当 GC 的时候所有的线程都是暂停状态的，需要处理的对象数量越多耗时越长，所以这也会造成卡顿。



那么什么情况下会出现这样的对象呢？基本可以分为以下四大类：1、集合类泄漏 2、单例/静态变量造成的内存泄漏 3、匿名内部类/非静态内部类 4、资源未关闭造成的内存泄漏

1、集合类泄漏

集合类添加元素后，仍引用着集合元素对象，导致该集合中的元素对象无法被回收，从而导致内存泄露。

举个栗子：

```
static List<Object> mList = new ArrayList<>();
for (int i = 0; i < 100; i++) {
    Object obj = new Object();
    mList.add(obj);
    obj = null;
}
```

那么什么情况下会出现这样的对象呢？基本可以分为以下四大类：1、集合类泄漏 2、单例/静态变量造成的内存泄漏 3、匿名内部类/非静态内部类 4、资源未关闭造成的内存泄漏

1、集合类泄漏

集合类添加元素后，仍引用着集合元素对象，导致该集合中的元素对象无法被回收，从而导致内存泄露。

2、单例/静态变量造成的内存泄漏

单例模式具有其 静态特性，它的生命周期 等于应用程序的生命周期，正是因为这一点，往往很容易造成内存泄漏。

3、匿名内部类/非静态内部类

这里有一张宝图：

class 对比	static inner class	non static inner class
与外部 class 引用关系	如果没有传入参数，就没有引用关系	自动获得强引用
被调用时需要外部实例	不需要	需要
能否调用外部 class 中的变量和方法	不能	能
生命周期	自主的生命周期	依赖于外部类，甚至比外部类更长

非静态内部类他会持有他外部类的引用，从图我们可以看到非静态内部类的生命周期可能比外部类更长，这就是二楼的情况一致了，如果非静态内部类的周明周期长于外部类，在加上自动持有外部类的强引用，我的乖乖，想不泄漏都难啊。

我们再来举个栗子：

```
public class TestActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test);
        new MyAscnTask().execute();
    }

    class MyAscnTask extends AsyncTask<Void, Integer, String>{
        @Override
        protected String doInBackground(Void... params) {
            try {
                Thread.sleep(100000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "";
        }
    }
}
```

我们经常会用这个方法去异步加载，然后更新数据。貌似很平常，我们开始学这个的时候就是这么写的，没发现有问题啊，但是你这么想一想，`MyAscnTask` 是一个非静态内部类，如果他处理数据的时间很长，极端点我们用 `sleep 100 秒`，在这期间 `Activity` 可能早就关闭了，本来 `Activity` 的内存应该被回收的，但是我们

知道非静态内部类会持有外部类的引用，所以 Activity 也需要陪着非静态内部类 MyAscnTask 一起天荒地老。好了，内存泄漏就形成了。

怎么办呢？

既然 MyAscnTask 的生命周期可能比较长，那就把它变成静态，和 Application 玩去吧，这样 MyAscnTask 就不会再持有外部类的引用了。两者也相互独立了。

```
public class TestActivity extends Activity {
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test);
        new MyAscnTask().execute();
    }
    //改了这里 注意一下 static
    static class MyAscnTask extends AsyncTask<Void, Integer, String>{
        @Override
        protected String doInBackground(Void... params) {
            try {
                Thread.sleep(100000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "";
        }
    }
}
```

说完非静态内部类，我再来看看匿名内部类，这个问题很常见，匿名内部类和非静态内部类有一个共同的地方，就是会只有外部类的强引用，所以这哥俩本质是一样的。但是处理方法有些不一样。但是思路绝对一样。换汤不换药。

举个灰常熟悉的栗子：

```
public class TestActivity extends Activity {
    private TextView mText;
    private Handler mHandler = new Handler(){
        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            //do something
            mText.setText(" do someThing");
        }
    };
}
```

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_test);
mText = findViewById(R.id.mText);
    // 匿名线程持有 Activity 的引用，进行耗时操作
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                Thread.sleep(100000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }).start();

    mHandler.sendMessageDelayed(0, 100000);
}

```

想必这两个方法是我们经常用的吧，很熟悉，也是这么学的，没感觉不对啊，老师就是这么教的，通过我们上面的分析，还这么想吗？关键是耗时时间过长，造成内部类的生命周期大于外部类，对非静态内部类，我们可以静态化，至于匿名内部类怎么办呢？一样把它变成静态内部类，也就是说尽量不要用匿名内部类。完事了吗？很多人不注意这么一件事，如果我们在 `handleMessage` 方法里进行 UI 的更新，这个 Handler 静态化了和 Activity 没啥关系了，但是比如这个 `mText`，怎么说？全写是 `activity.mText`，看到了吧，持有了 Activity 的引用，也就是说 Handler 费劲心思变成静态类，自认为不持有 Activity 的引用了，准确的说是不自动持有 Activity 的引用了，但是我们要做 UI 更新的时候势必会持有 Activity 的引用，静态类持有非静态类的引用，我们发现怎么又开始内存泄漏了呢？处处是坑啊，怎么办呢？我们这里就要引出弱引用的概念了。

引用分为强引用，软引用，弱引用，虚引用，强度依次递减。

强引用 我们平时不做特殊处理的一般都是强引用，如果一个对象具有强引用，GC 宁可 OOM 也绝不会回收它。看出多强硬了吧。

软引用(SoftReference) 如果内存空间足够，GC 就不会回收它，如果内存空间不足了，就会回收这些对象的内存。

弱引用(WeakReference) 弱引用要比软引用更弱一个级别，内存不够要回收他，GC 的时候不管内存够不够也要回收他，简直是弱的一匹。不过 GC 是一个优先级很低的线程，也不是太频繁进行，所以弱引用的生活还过得去，没那么提心吊胆。

虚引用 用的甚少，我没有用过，如果想了解的朋友，可以自行谷歌百度。

所以我们用弱引用来修饰 Activity，这样 GC 的时候，该回收的也就回收了，不会再有内存泄漏了。很完美。

```
public class TestActivity extends Activity {
```

```
private TextView mText;
private MyHandler myHandler = new MyHandler(TestActivity.this);
private MyThread myThread = new MyThread();

private static class MyHandler extends Handler {

    WeakReference<TestActivity> weakReference;

    MyHandler(TestActivity testActivity) {
        this.weakReference = new WeakReference<TestActivity>(testActivity);

    }

    @Override
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        weakReference.get().mText.setText("do someThing");

    }
}

private static class MyThread extends Thread {

    @Override
    public void run() {
        super.run();

        try {
            sleep(100000);

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_test);
    mText = findViewById(R.id.mText);
    myHandler.sendMessageDelayed(0, 100000);
    myThread.start();
}
```

```
//最后清空这些回调
@Override
protected void onDestroy() {
    super.onDestroy();
    myHandler.removeCallbacksAndMessages(null);
}
```

4、资源未关闭造成内存泄漏

- 网络、文件等流忘记关闭
- 手动注册广播时，退出时忘记 `unregisterReceiver()`
- Service 执行完后忘记 `stopSelf()`
- EventBus 等观察者模式的框架忘记手动解除注册

这些需要记住又开就有关，具体做法也很简单就不一一赘述了。给大家介绍几个很好用的工具： 1、leakcanary 傻瓜式操作，哪里有泄漏自动给你显示出来，很直接很暴力。 2、我们平时也要多使用 **Memory Monitor** 进行内存监控，这个分析就有些难度了，可以上网搜一下具体怎么使用。 3、Android Lint 它可以帮助我们发现代码机构 / 质量问题，同时提供一些解决方案，内存泄露的会飘黄，用起来很方便，具体使用方法上网学习，这里不多做说明了。

so

优化 3.0

1. 解决各个情况下的内存泄漏，注意平时代码的规范。

4、启动速度优化

不知道大家有没有细心发现，我们的应用启动要比别的大厂的要慢，要花费更多的时间，明明他们的包体更大，app 更复杂，怎么启动时间反而比我们的短呢？但是这块的优化关注的人很少，因为 App 常常伴有闪屏页，所以这个问题看起来就不是问题了，但是一款好的应用是绝对不允许这样的，我加闪屏页是我的事，启动速度慢绝对不可以。

app 启动分为冷启动（Cold start）、热启动（Hot start）和温启动（Warm start）三种。

优化 4.0

1. 利用提前展示出来的 Window，快速展示出来一个界面，给用户快速反馈的体验；
2. 避免在启动时做密集沉重的初始化（Heavy app initialization）；
3. 避免 I/O 操作、反序列化、网络操作、布局嵌套等。

5、包体优化

我做过两年的海外应用产品，深知包体大小对于产品新增的影响，包体小百分之五，可能新增就增加百分之五。如果产品基数很大，这个提升就更可怕了。不管怎么说，我们要减肥，要六块腹肌，不要九九归一的大肚子。

既然要瘦身，那么我们必须知道 APK 的文件构成，解压 apk：



assets 文件夹 存放一些配置文件、资源文件，**assets** 不会自动生成对应的 ID，而是通过 `AssetManager` 类的接口获取。

res 目录 **res** 是 `resource` 的缩写，这个目录存放资源文件，会自动生成对应的 ID 并映射到 `.R` 文件中，访问直接使用资源 ID。

META-INF 保存应用的签名信息，签名信息可以验证 APK 文件的完整性。

AndroidManifest.xml 这个文件用来描述 Android 应用的配置信息，一些组件的注册信息、可使用权限等。

classes.dex Dalvik 字节码程序，让 Dalvik 虚拟机可执行，一般情况下，Android 应用在打包时通过 Android SDK 中的 `dx` 工具将 Java 字节码转换为 Dalvik 字节码。

resources.arsc 记录着资源文件和资源 ID 之间的映射关系，用来根据资源 ID 寻找资源。

我们需要从代码和资源两个方面去减少响应的大小。

1、首先我们可以使用 `lint` 工具，如果有没有使用过的资源就会打印如下的信息(不会使用的朋友可以上网看一下)

`res/layout/preferences.xml: Warning: The resource R.layout.preferences appears`

`to be unused [UnusedResources]`

同时我们可以开启资源压缩，自动删除无用的资源

```
android {
```

```
    ...
```

```
    buildTypes {
```

```
        release {
```

```
            shrinkResources true
```

```
            minifyEnabled true
```

```
            proguardFiles getDefaultProguardFile('proguard-android.txt'),  
                'proguard-rules.pro'
```

```
        }
```

```
}
```

无用的资源已经被删除了，接下来哪里可以在瘦身呢？

2、我们可以使用可绘制对象，某些图像不需要静态图像资源；框架可以在运行时动态绘制图像。Drawable 对象（`<shape>`以 XML 格式）可以占用 APK 中的少量空间。此外，XML Drawable 对象产生符合材料设计准则的单色图像。

上面的话官方，简单说来就是，能自己用 XML 写 Drawable，就自己写，能不用公司的 UI 切图，就别和他们说话，咱们自己造，做自己的 UI，美滋滋。而且这种图片占用空间会很小。

3、重用资源，比如一个三角按钮，点击前三角朝上代表收起的意思，点击后三角朝下，代表展开，一般情况下，我们会用两张图来切换，我们完全可以用旋转的形式去改变

```
<?xml version="1.0" encoding="utf-8"?>
<rotate xmlns:android="http://schemas.android.com/apk/res/android"
    android:drawable="@drawable/ic_thumb_up"
    android:pivotX="50%"
    android:pivotY="50%"
    android:fromDegrees="180" />
```

比如同一图像的着色不同，我们可以用 `android:tint` 和 `tintMode` 属性，低版本（5.0 以下）可以使用 `ColorFilter`。

4、压缩 PNG 和 JPEG 文件 您可以减少 PNG 文件的大小，而不会丢失使用工具如图像质量 `pngcrush`, `pngquant`, 或 `zopflipng`。所有这些工具都可以减少 PNG 文件的大小，同时保持感知的图像质量。

5、使用 WebP 文件格式 可以使用图像的 WebP 文件格式，而不是使用 PNG 或 JPEG 文件。WebP 格式提供有损压缩（如 JPEG）以及透明度（如 PNG），但可以提供比 JPEG 或 PNG 更好的压缩。

可以使用 Android Studio 将现有的 BMP, JPG, PNG 或静态 GIF 图像转换为 WebP 格式。

6、使用矢量图形 可以使用矢量图形来创建与分辨率无关的图标和其他可伸缩 Image。使用这些图形可以大大减少 APK 大小。一个 100 字节的文件可以生成与屏幕大小相关的清晰图像。

但是，系统渲染每个 VectorDrawable 对象需要花费大量时间，而较大的图像需要更长的时间才能显示在屏幕上。因此，请考虑仅在显示小图像时使用这些矢量图形。

不要把 AnimationDrawable 用于创建逐帧动画，因为这样做需要为动画的每个帧包含一个单独的位图文件，这会大大增加 APK 的大小。

7、代码混淆 使用 `proGuard` 代码混淆器工具，它包括压缩、优化、混淆等功能。这个大家太熟悉了。不多说了。

```
android {
    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                'proguard-rules.pro'
        }
    }
}
```

}

8、插件化。 比如功能模块放在服务器上，按需下载，可以减少安装包大小。

so

优化 5.0

1. 代码混淆
2. 插件化
3. 资源优化

6、耗电优化

我们可能对耗电优化不怎么感冒，没事，谷歌这方面做得也不咋地，5.0 之后才有像样的方案，讲实话这个优化的优先级没有前面几个那么高，但是我们也要了解一些避免耗电的坑，至于更细的耗电分析可以使用这个 Battery Historian。

Battery Historian 是由 Google 提供的 Android 系统电量分析工具，从手机中导出 bugreport 文件上传至页面，在网页中生成详细的图表数据来展示手机上各模块电量消耗过程，最后通过 App 数据的分析制定出相关的电量优化的方法。

我们来谈一下怎么规避电老虎吧。

谷歌推荐使用 JobScheduler，来调整任务优先级等策略来达到降低损耗的目的。

JobScheduler 可以避免频繁的唤醒硬件模块，造成不必要的电量消耗。避免在不合适的时间(例如低电量情况下、弱网络或者移动网络情况下的)执行过多的任务消耗电量。

具体功能： 1、可以推迟的非面向用户的任务(如定期数据库数据更新); 2、当充电时才希望执行的工作(如备份数据); 3、需要访问网络或 Wi-Fi 连接的任务(如向服务器拉取配置数据); 4、零散任务合并到一个批次去定期运行; 5、当设备空闲时启动某些任务; 6、只有当条件得到满足，系统才会启动计划中的任务 (充电、WIFI...)；

同时谷歌针对耗电优化也提出了一个懒惰第一的法则：

减少 你的应用程序可以删除冗余操作吗？例如，它是否可以缓存下载的数据而不是重复唤醒无线电以重新下载数据？

推迟 应用是否需要立即执行操作？例如，它可以等到设备充电才能将数据备份到云端吗？

合并 可以批处理工作，而不是多次将设备置于活动状态吗？例如，几十个应用程序是否真的有必要在不同时间打开收音机发送邮件？在一次唤醒收音机期间，是否可以传输消息？

谷歌在耗电优化这方面确实显得有些无力，希望以后可以推出更好的工具和解决方案，不然这方面的优化优先级还是很低。付出和回报所差太大。

so

优化 6.0

1. 使用 JobScheduler 调度任务
2. 使用懒惰法则

6、ListView 和 Bitmap 优化

针对 ListView 优化，主要是合理使用 ViewHolder。创建一个内部类 ViewHolder，里面的成员变量和 view 中所包含的组件个数、类型相同，在 convertView 为 null 的时候，把 findViewById 找到的控件赋给 ViewHolder 中对应的变量，就相当于先把它们装进一个容器，下次要用的时候，直接从容器中获取。

现在我们现在一般使用 RecyclerView，自带这个优化，不过还是要理解一下原理的好。然后可以对接受来的数据进行分段或者分页加载，也可以优化性能。

对于 Bitmap，这个我们使用的就比较多了，很容易出现 OOM 的问题，图片内存的问题可以看一下我之前写的这篇文章一张图片占用多少内存。

Bitmap 的优化套路很简单，粗暴，就是让压缩。三种压缩方式： 1.对图片质量进行压缩 2.对图片尺寸进行压缩 3.使用 libjpeg.so 库进行压缩

优化 7.0

1. ListView 使用 ViewHolder，分段，分页加载
2. 压缩 Bitmap

8、响应速度优化

影响响应速度的主要因素是主线程有耗时操作，影响了响应速度。所以响应速度优化的核心思想是避免在主线程中做耗时操作，把耗时操作异步处理。

9、线程优化

线程优化的思想是采用线程池，避免在程序中存在大量的 Thread。线程池可以重用内部的线程，从而避免了现场的创建和销毁所带来的性能开销，同时线程池还能有效地控制线程池的最大并发数，避免大量的线程因互相抢占系统资源从而导致阻塞现象发生。

《Android 开发艺术探索》对线程池的讲解很详细，不熟悉线程池的可以去了解一下。

•

优点： 1、减少在创建和销毁线程上所花的时间以及系统资源的开销。 2、如不使用线程池，有可能造成系统创建大量线程而导致消耗完系统内存以及“过度切换”。

•

•

需要注意的是： 1、如果线程池中的数量为达到核心线程的数量，则直接会启动一个核心线程来执行任务。 2、如果线程池中的数量已经达到或超过核心线程的数量，则任务会被插入到任务队列中待执行。 3、如果(2)中的任务无法插入到任务队列中，由于任务队列已满，这时候如果线程数量未达到线程池规定最大值，则会启动一个非核心线程来执行任务。 4、如果(3)中线程数量已经达到线程池最大值，则会拒绝执行此任务， ThreadPoolExecutor 会调用 RejectedExecutionHandler 的 rejectedExecution 方法通知调用者。

10、微优化

这些微优化可以在组合时提高整体应用程序性能，但这些更改不太可能导致显着的性能影响。选择正确的算法和数据结构应始终是我们的首要任务，以提高代码效率。

- 编写高效代码有两个基本规则： 1、不要做你不需要做的工作 2、如果可以避免，请不要分配内存

- 1、避免创建不必要的对象 对象创建永远不是免费的，虽然每一个的代价不是很大，但是总归是代价的不是吗？能不创建何必要浪费资源呢？
- 2、首选静态（这里说的是特定情景） 如果您不需要访问对象的字段，请使您的方法保持静态。调用速度将提高约 15%-20%。这也是很好的做法，因为你可以从方法签名中看出，调用方法不能改变对象的状态
- 3、对常量使用 static final 此优化仅适用于基本类型和 String 常量，而不适用于任意引用类型。尽管如此， static final 尽可能声明常量是一种好习惯。
- 4、使用增强的 for 循环语法 增强 for 循环（for-each）可用于实现 Iterable 接口和数组的集合。对于集合，分配一个迭代器来对 hasNext() 和进行接口调用 next()。使用一个 ArrayList，手写计数循环快约 3 倍，但对于其他集合，增强的 for 循环语法将完全等效于显式迭代器用法。
- 5、避免使用浮点数 根据经验，浮点数比 Android 设备上的整数慢约 2 倍

3、如何避免 UI 卡顿

3. 1. 外部引起的

比如：Activity 里面直接进行网络访问/大文件的 IO 操作

内存这一块有些什么要注意的。

- 1) 内存抖动的问题。



- 2) 一个方法太耗时了。

3. 2. View 本身的卡顿

自定义 View 要注意的，能否优化：

- 1) 可以使用 Allocation Tracing 来定位大致的情况
- 2) 可以使用 TraceView 来确定详细的问题所在。

十八、Android MVC、MVP、MVVM

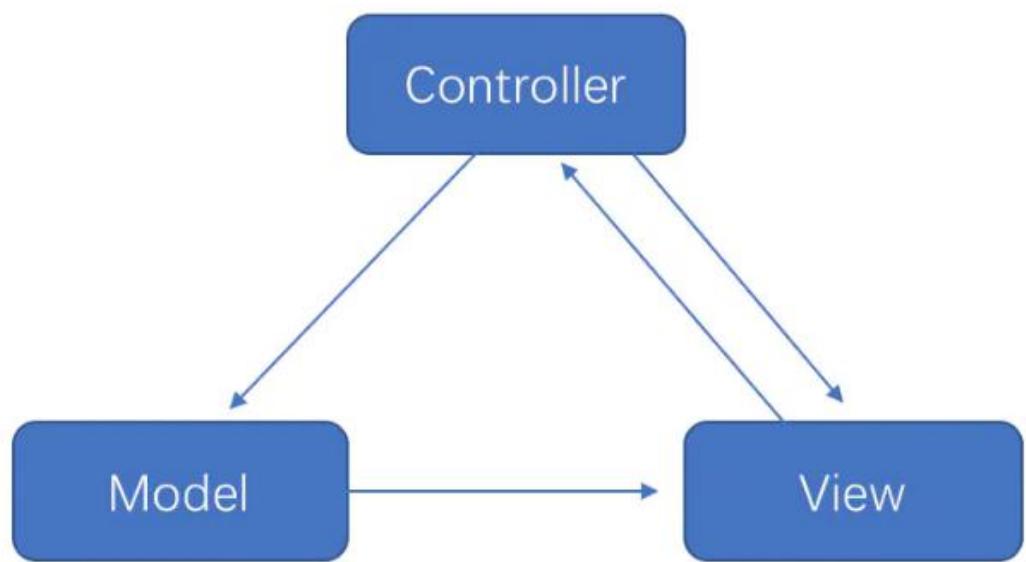
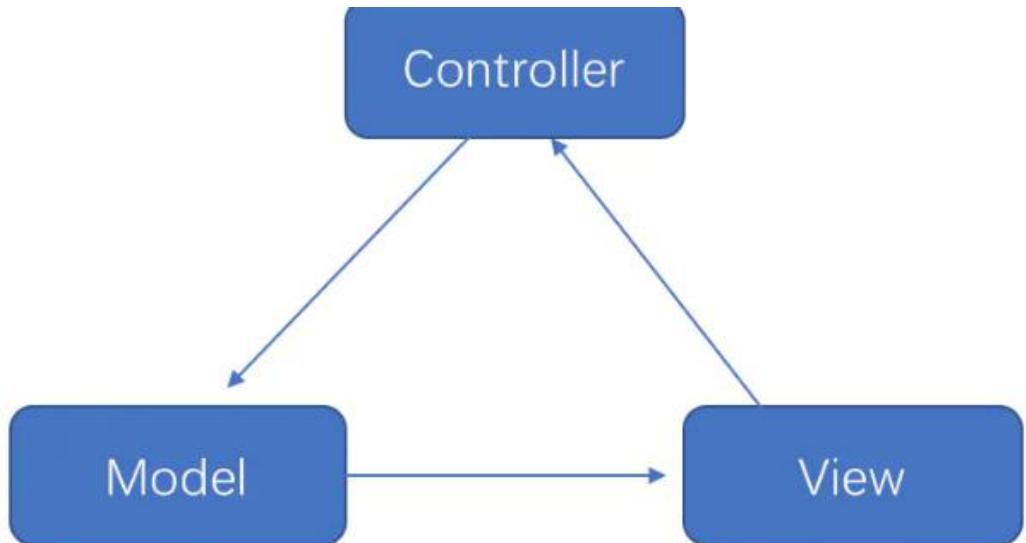
设计模式选择

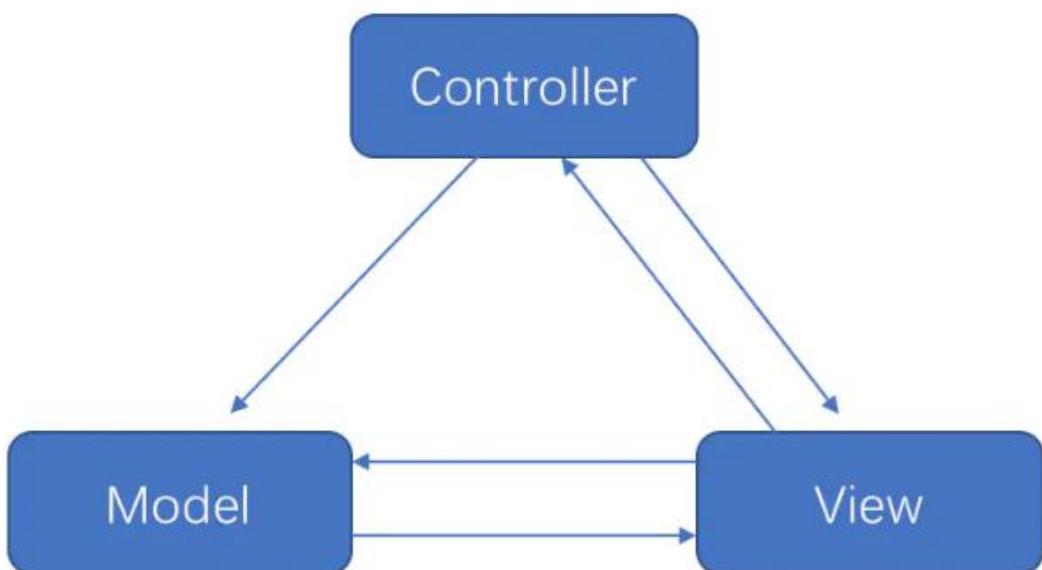
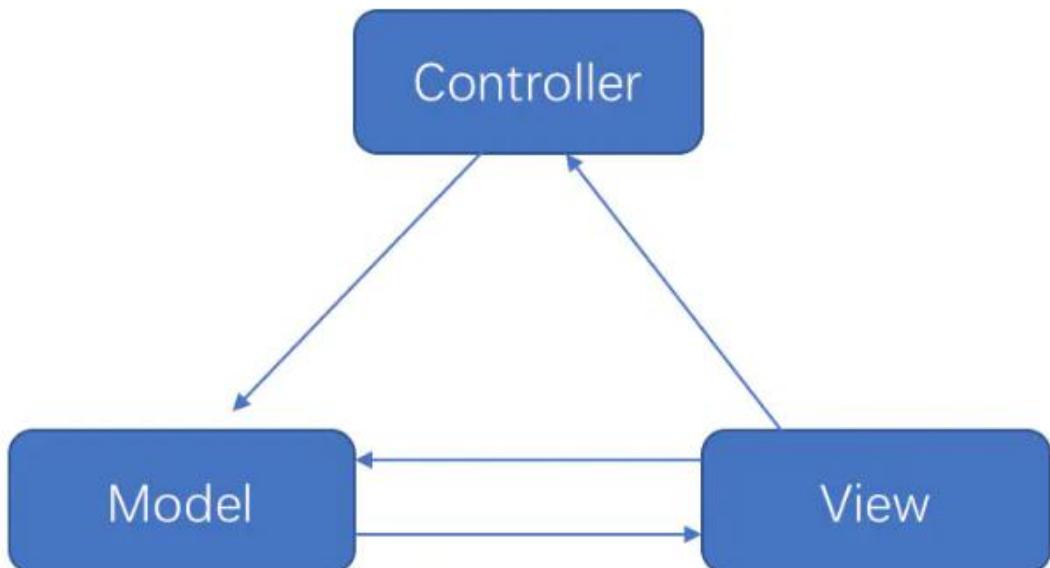
MVC

可能由于 MVP、MVVM 的兴起，MVC 在 android 中的应用变得越来越少了，但 MVC 是基础，理解好 MVC 才能更好的理解 MVP,MVVM。因为后两种都是基于 MVC 发展而来的。

1、MVC 眼花缭乱设计图

我们从网上搜索 mvc 相关资料时，如果你多看几篇文章的话可能会发现，好像他们介绍的设计图都不太一样，这里罗列了大部分的设计图



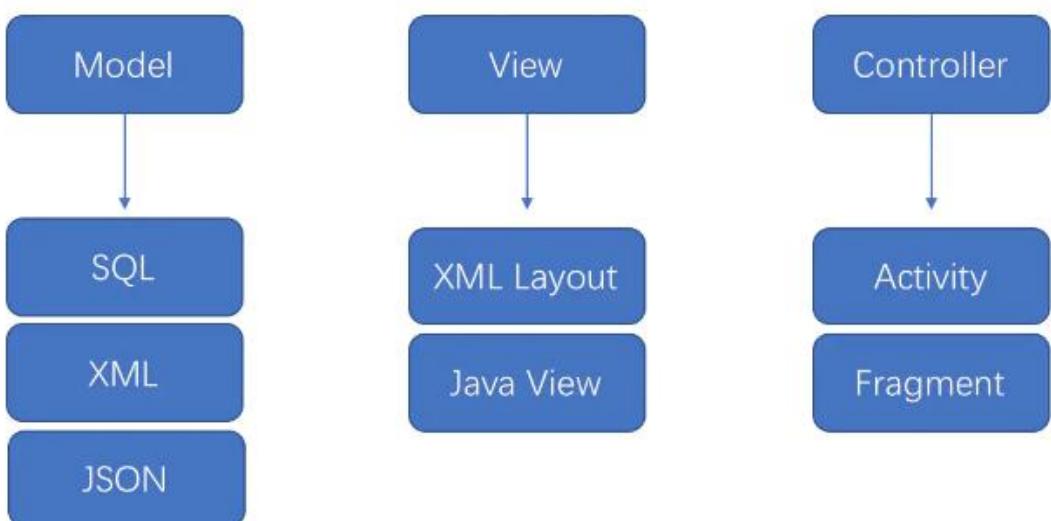


2、MVC 设计图解释

到底上面列出的设计图哪个才是对的。其实都是对的。为什么这么说呢，这得从 mvc 的发展说起。MVC 框架模式最早由 Trygve Reenskaug 于 1978 年在 Smalltalk-80 系统上首次提出。经过了这么多年的发展，当然会演变出不同的版本，但核心没变依旧还是三层模型 Model-View-Controller。

3、MVC 三层之间的关系

箭头→代表的是一种事件流向，并不一定要持有对方，比如上图中 model→view 的事件流向，view 可以通过注册监听器的形式得到 model 发来的事件。在设计中 model view controller 之间如果要通讯，尽量设计成不直接持有，这样方便复用。也符合 mvc 的设计初衷 在 android 中三者对应的关系如下：



视图层(View) 对应于 xml 布局文件和 java 代码动态 view 部分
 控制层(Controller) MVC 中 Android 的控制层是由 Activity 来承担的，Activity 本来主要是作为初始化页面，展示数据的操作，但是因为 XML 视图功能太弱，所以 Activity 既要负责视图的显示又要加入控制逻辑，承担的功能过多。

模型层(Model) 针对业务模型，建立的数据结构和相关的类，它主要负责网络请求，数据库处理，I/O 的操作。

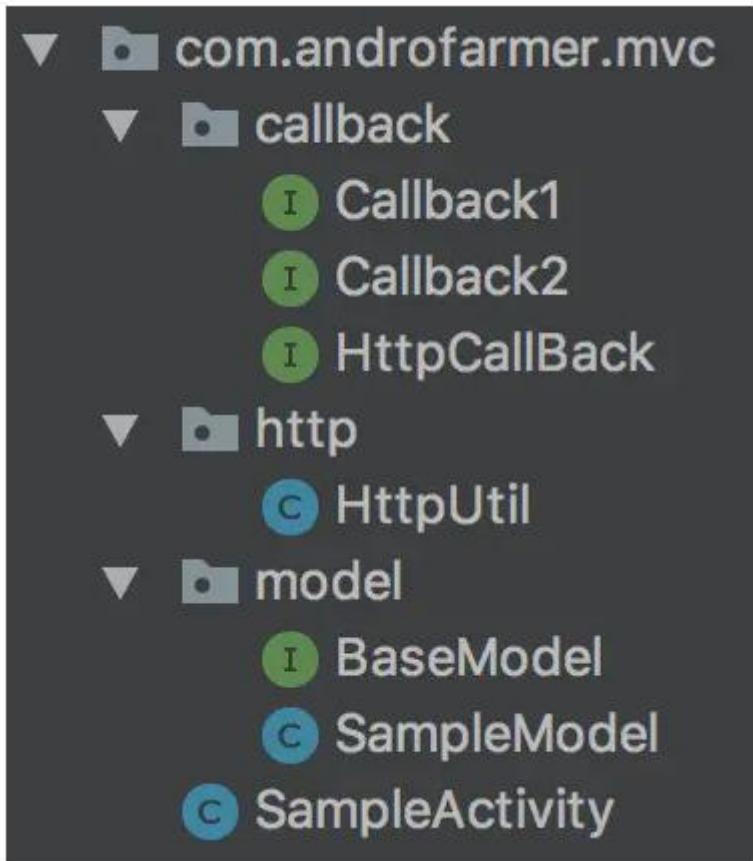
由于 android 中有个 god object 的存在 activity，再加上 android 中 xml 布局的功能性太弱，所以 activity 承担了绝大部分的工作。所以在 android 中 mvc 更像是这种形式：



因为 activity 扮演了 controller 和 view 的工作，所以 controller 和 view 不太好彻底解耦，但是在一定程度上我们还是可以解耦的。Talk is cheap. Show me the code. 扯了这么多，我们来看点代码。

4、MVC sample

通过代码来看下，mvc 在 android 中的实现



结构很简单，这里介绍下其中的关键代码

```
public interface BaseModel {  
    void onDestroy();  
}
```

[复制代码](#)

BaseModel 顾名思义就是所有业务逻辑 model 的父类，这里的 `onDestroy()`方法用于跟 activity 或者 fragment 生命周期同步，在 `destroy` 做一些销毁操作

```
public interface Callback1<T> {  
    void onCallBack(T t);  
}  
public interface Callback2<T,P> {  
    void onCallBack(T t,P p);  
}
```

[复制代码](#)

Callback 是根据 View 或者 Controller 调用 Model 时回调的参数个数选择使用
`public class SampleModel implements BaseModel{`

```
public void getUserInfo(String uid,Callback1<UserInfo> callback)  
{
```

```
        UserInfo userInfo= new HttpUtil<UserInfo>().get(uid);
        callback.onCallBack(userInfo);

    }

    @Override
    public void onDestroy() {

    }

    public class UserInfo
    {
        private int age;
        private String name;

        public int getAge() {
            return age;
        }

        public void setAge(int age) {
            this.age = age;
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }
    }
}
```

SampleModel 是我们业务逻辑的具体实现

```
public class SampleActivity extends AppCompatActivity {
    private SampleModel sampleModel;
    Button button;
    EditText textView;
    TextView tvAge,tvName;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_sample);
        sampleModel=new SampleModel();
        button.setOnClickListener(new View.OnClickListener() {
```

```

        @Override
        public void onClick(View view) {
            getUserInfo(textView.getText().toString());
        }
    });

}

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

/**
 * 获取用户信息
 * @param uid
 */
private void getUserInfo(String uid)
{
    sampleModel.getUserInfo(uid, new Callback1<SampleModel.UserInfo>() {
        @Override
        public void onCallBack(SampleModel.UserInfo userInfo) {
            setDataToView(userInfo);
        }
    });
}

/**
 * 设置用户信息到 view
 */
private void setDataToView(SampleModel.UserInfo userInfo)
{
    tvAge.setText(userInfo.getAge());
    tvName.setText(userInfo.getName());
}
}

```

前面说了 Activity 充当 View 和 Controller，但是我们依然要区分到底哪一部分是 View 的操作，哪一部分是 Controller 的操作。 我们分析下事件的流向
button 点击事件的触发： View→Controller 获得用户信息事件的触发：
Controller→Model 绑定用户信息到 View： Controller→View 至此 MVC 就讲完了

5、MVC 总结

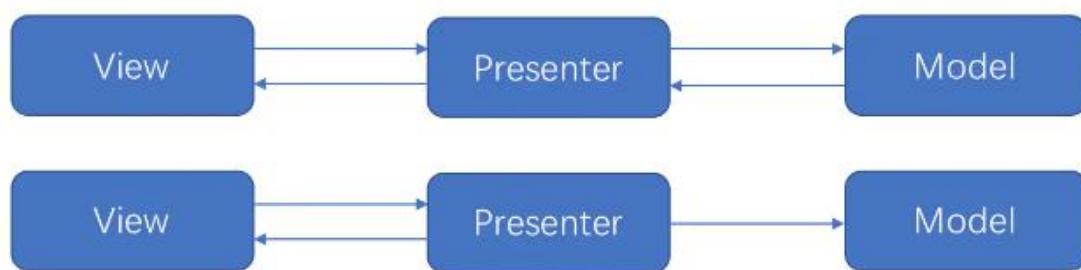
我们这里根据 sample 来总结下：

- 具有一定的分层，model 彻底解耦，controller 和 view 并没有解耦
- 层与层之间的交互尽量使用回调或者去使用消息机制去完成，尽量避免直接持有
- controller 和 view 在 android 中无法做到彻底分离，但在代码逻辑层面一定要分清
- 业务逻辑被放置在 model 层，能够更好的复用和修改增加业务

MVP

1、MVP 说明

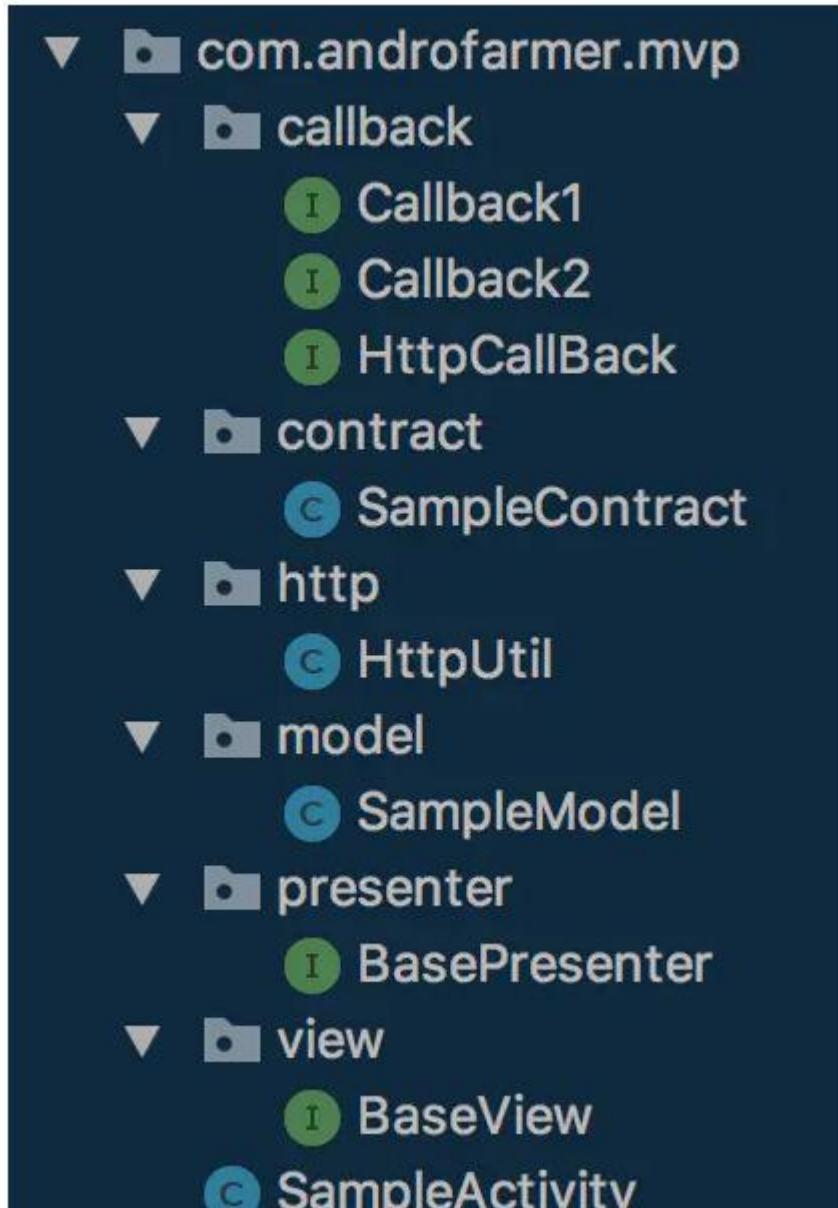
MVP 跟 MVC 很相像，文章开头列出了很多种 MVC 的设计图，所以根据 MVC 的发展来看，我们把 MVP 当成 MVC 来看也不为过，因为 MVP 也是三层，唯一的差别是 Model 和 View 之间不进行通讯，都是通过 Presenter 完成。前面介绍 MVC 的时候提到了算是致命缺点吧，在 android 中由于 activity (god object) 的存在，Controller 和 View 很难做到完全解耦。但在 MVP 中就可以很好的解决这个问题 看下 MVP 的设计图：



一般情况下就这两种

2、MVP Sample

依然延续 MVC 的例子，修改下结构通过 MVP 去实现，看下项目代码结构：



callback, http 包下内容基本一致，主要看下不同的地方

```
public interface BasePresenter {  
    void onDestroy();  
}
```

[复制代码](#)

BasePresenter 类似于 MVC 中的 BaseModel，主要负责业务逻辑的实现。我们这里没有把业务逻辑放在 Model 里去实现，当然把主要业务逻辑放在 Model 中去实现也是可以的。google 的 MVP 实现方案是把业务逻辑放在 presenter 中，弱化 Model，我们这里也是这样做的。

```
public interface BaseView<P extends BasePresenter> {  
    void setPresenter(P presenter);  
}
```

BaseView 是所有 View 的父类，将 android 中的 view 抽象话出来，只有跟 view 相关的

操作都由 baseView 的实现类去完成。

```
public class SampleContract {  
    public static class Presenter implements BasePresenter {  
        public void getUserInfo(String uid, Callback1<SampleModel.UserInfo> callback) {  
            SampleModel.UserInfo userInfo = new  
            HttpUtil<SampleModel.UserInfo>().get(uid);  
            callback.onCallBack(userInfo);  
        }  
  
        @Override  
        public void onDestroy() {  
        }  
    }  
    public interface View extends BaseView<Presenter> {  
        void setDataToView(SampleModel.UserInfo userInfo);  
    }  
}
```

Contract 契约类这是 Google MVP 与其他实现方式的又一个不同，契约类用于定义同一个界面的 view 的接口和 presenter 的具体实现。好处是通过规范的方法命名和注释可以清晰的看到整个页面的逻辑。

```
public class SampleActivity extends AppCompatActivity implements  
SampleContract.View {  
    private SampleContract.Presenter mPresenter;  
    Button button;  
    EditText textView;  
    TextView tvAge, tvName;  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_sample);  
        setPresenter(new SampleContract.Presenter());  
  
        button.setOnClickListener(new View.OnClickListener() {  
            @Override  
            public void onClick(View view) {  
                mPresenter.getUserInfo(textView.getText().toString(),  
                new  
                Callback1<SampleModel.UserInfo>() {  
                    @Override  
                    public void onCallBack(SampleModel.UserInfo userInfo) {  
                }  
            }  
        }  
    }  
}
```

```

        setDataToView(userInfo);
    }
});
}
});

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

@Override
public void setDataToView(SampleModel.UserInfo userInfo) {
    tvAge.setText(userInfo.getAge());
    tvName.setText(userInfo.getName());
}

@Override
public void setPresenter(SampleContract.Presenter presenter) {
    mPresenter=presenter;
}
}

```

这里的 `SampleActivity` 实现了 `SampleContract.View` 只是作为 `View` 存在的。虽然看起来，跟 `MVC` 中的实现很相似，但却有本质的区别。`mPresenter` 为 `Model` 和 `View` 之间交互的桥梁。`Presenter` 跟 `View` 相互持有，这里 `SampleActivity` 实现了 `SampleContract.View`，`mPresenter` 作为 `SampleActivity` 的成员变量，`SampleActivity` 当然持有 `mPresenter`，由于 `mPresenter` 是非静态的成员变量，因此默认持有 `SampleActivity` 的引用。

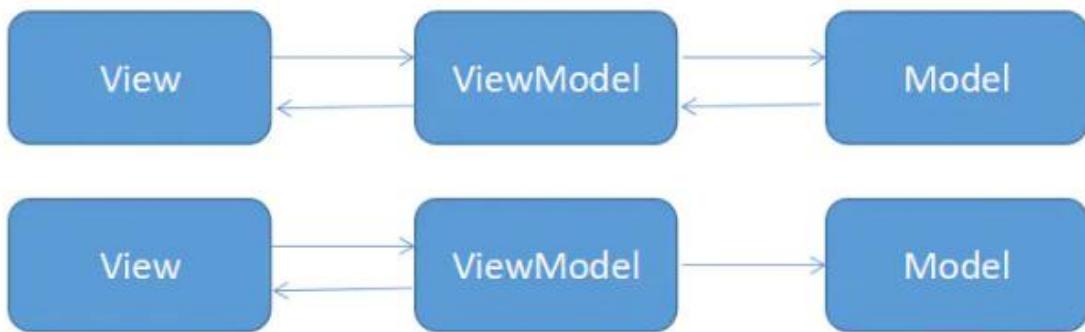
3、MVP 总结

通过引入接口 `BaseView`，让相应的视图组件如 `Activity`, `Fragment` 去实现 `BaseView`，实现了视图层的独立，通过中间层 `Preseter` 实现了 `Model` 和 `View` 的完全解耦。`MVP` 彻底解决了 `MVC` 中 `View` 和 `Controller` 傻傻分不清楚的问题，但是随着业务逻辑的增加，一个页面可能会非常复杂，UI 的改变是非常多，会有非常多的 case，这样就会造成 `View` 的接口会很庞大。

MVVM

1、MVVM 说明

MVP 中我们说过随着业务逻辑的增加，UI 的改变多的情况下，会有非常多的跟 UI 相关的 case，这样就会造成 View 的接口会很庞大。而 MVVM 就解决了这个问题，通过双向绑定的机制，实现数据和 UI 内容，只要想改其中一方，另一方都能够及时更新的一种设计理念，这样就省去了很多在 View 层中写很多 case 的情况，只需要改变数据就行。先看下 MVVM 设计图：



一般情况下就这两种情况，这看起来跟 MVP 好像没啥差别，其实区别还是挺大的，在 MVP 中 View 和 presenter 要相互持有，方便调用对方，而在 MVVM 中 View 和 ViewModel 通过 Binding 进行关联，他们之前的关联处理通过 DataBinding 完成。

2、MVVM 与 DataBinding 的关系

一句话表述就是，MVVM 是一种思想，DataBinding 是谷歌推出的方便实现 MVVM 的工具。在 google 推出 DataBinding 之前，因为 xml layout 功能较弱，想实现 MVVM 非常困难。而 DataBinding 的出现可以让我们很方便的实现 MVVM。

3、DataBinding 简介

DataBinding 是实现视图和数据双向绑定的工具，这里简单介绍下基本用法，详细用法可以参照官方:<https://developer.android.com/topic/libraries/data-binding/> 启用 DataBinding，只需要在 gradle 文件中添加如下代码：

```
android {  
    dataBinding{  
        enabled true  
    }  
}
```

通过 DataBindingUtil 可以动态生成一个 ViewDataBinding 的子类，类名以 layout 文件名大写加 Binding 组成，如：

```
ActivitySampleMvvmBinding binding = DataBindingUtil.setContentView(this,  
    R.layout.activity_sample_mvvm);
```

在 layout 中需要我们配置，每个控件绑定的实体对象，以 layout 进行包裹，data 中配置变量名和类型，通过 @{} 或 @= {} 的方式进行引用，其中 @= {} 的方式表示双向绑定。目前支持双向绑定的控件如下

```
AbsListView android:selectedItemPosition CalendarView android:date  
CompoundButton android:checked DatePicker android:year, android:month,  
android:day NumberPicker android:value RadioGroup android:checkedButton  
RatingBar android:rating SeekBar android:progress TabHost android:currentTab  
TextView android:text TimePicker android:hour, android:minute
```

```
<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android">  
    <data>  
        <variable  
            name="user"  
            type="com.androfarmer.mvvm.model.SampleModel.UserInfo">  
        </variable>  
    </data>  
<LinearLayout  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:orientation="vertical">  
  
    <TextView  
        android:id="@+id/tv_name"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="@={user.name}"  
    />  
    <TextView  
        android:id="@+id/tv_age"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="@={user.age+``}"  
    />  
  
</LinearLayout>  
</layout>
```

以上为具体在 xml 的用法展示

```
public static class UserInfo extends BaseObservable  
{  
    private int age;  
    private String name;  
  
    @Bindable  
    public int getAge() {
```

```

        return age;
    }

    public void setAge(int age) {
        this.age = age;
        notifyPropertyChanged(BR.age);
    }

    @Bindable
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
        notifyPropertyChanged(BR.name);
    }
}

```

为了实现双向绑定还需要对数据实体类做处理，继承 `BaseObservable`，对读写方法做`@Bindable` 和 `notifyPropertyChanged` 处理。还可以直接使用官方提供的泛型可观察对象 `ObservableField`，比如：`private ObservableField name=new ObservableField<>();`

4、MVVM Sample

MVVM 中跟 MVP 中一样，将三层划分的很清楚，Activity 和 xml layout 充当 View，ViewModel 处理业务逻辑以及获取数据，弱化 Model。很多代码跟前面类似，这里只列出核心代码，ViewModel 层的

```

public interface BaseViewModel {
    void onDestroy();
}

public abstract class AbstractViewModel<T extends ViewDataBinding> implements
BaseViewModel {
    public T mViewDataBinding;
    public AbstractViewModel(T viewDataBinding)
    {
        this.mViewDataBinding=viewDataBinding;
    }

    @Override
    public void onDestroy() {
        mViewDataBinding.unbind();
    }
}

```

```

}

public class SampleViewModel extends AbstractViewModel<ActivitySampleMvvmBinding> {

    public SampleViewModel(ActivitySampleMvvmBinding viewDataBinding) {
        super(viewDataBinding);
    }

    public void getUserInfo(String uid, Callback1<SampleModel.UserInfo> callback) {
        //从网络或缓存获取信息
        SampleModel.UserInfo userInfo=new SampleModel.UserInfo();
        userInfo.setName("tom");
        userInfo.setAge(18);
        callback.onCallBack(userInfo);
    }
}

ViewModel 层主要处理业务逻辑和获取数据，mViewDataBinding 是通过 View 层传递过来

private SampleViewModel mSampleViewModel;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ActivitySampleMvvmBinding binding = DataBindingUtil.setContentView(this,
R.layout.activity_sample_mvvm);
    mSampleViewModel=new SampleViewModel(binding);
    setDataToView();
}
private void setDataToView()
{
    mSampleViewModel.getUserInfo("uid", new
Callback1<SampleModel.UserInfo>() {
        @Override
        public void onCallBack(SampleModel.UserInfo userInfo) {
            mSampleViewModel.mViewDataBinding.setUser(userInfo);
        }
    });
}

xml layout 代码在上面介绍 databing 的用法时已给出，通过上面代码我们就将数据 UserInfo 跟 View 进行绑定了。比如我们更新用户信息，可以直接对 View 上的属性进行修改： mSampleViewModel.mViewDataBinding.tvName.setText("rose"); 也可以通过修改 UserInfo 实体类的字段信息：
mSampleViewModel.mViewDataBinding.setUser(userInfo);

```

从此告别 MVP 中 View 层好多接口的问题，让 View 变得更简洁，修改任何一方，两者都会保持数据同步。

5、MVVM 总结

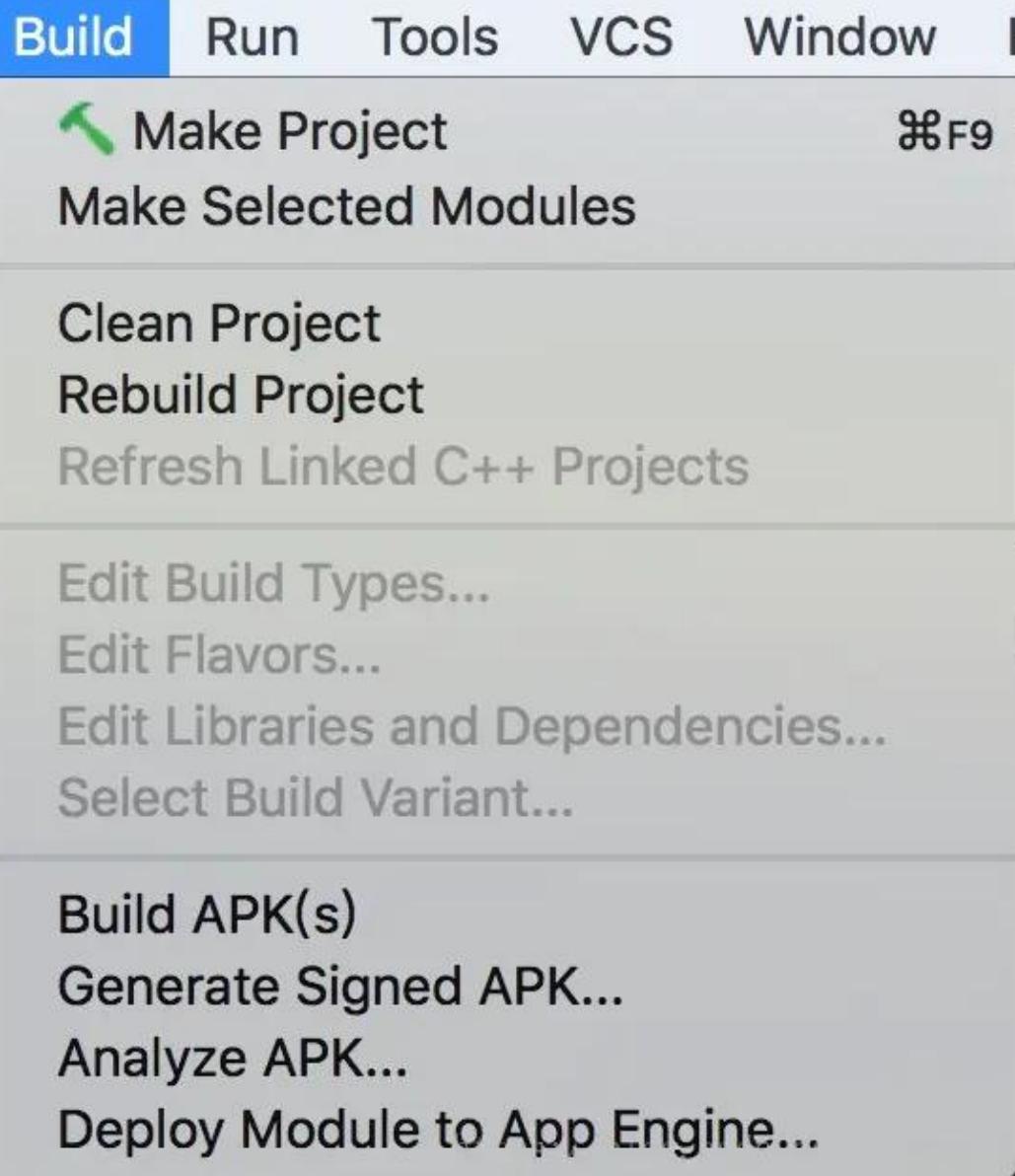
看起来 MVVM 很好的解决了 MVC 和 MVP 的不足，但是由于数据和视图的双向绑定，导致出现问题时不太好定位来源，有可能数据问题导致，也有可能业务逻辑中对视图属性的修改导致。如果项目中打算用 MVVM 的话可以考虑使用官方的架构组件 ViewModel、LiveData、DataBinding 去实现 MVVM

十九、Android Gradle 知识

Gradle 系列一

Gradle task

日常开发中，上一节提到的四点中，Gradle task 与大部分的开发者开发是最为紧密的。日常开发中开发者难免会进行 build/clean project、build apk 等操作 ——



实际上这些按钮的底层实现都是通过 Gradle task 来完成的，只不过 IDE 使用 GUI 降低开发者们的使用门槛。当然，如果各位读者足够细心观察的话，是能够看到点击相应按钮后在 Build 输出栏中会输出相应的信息 ——

```
:app:prebuiltd UP-TO-DATE
:app:preDebugBuild
:app:compileDebugAidl
:app:compileDebugRenderscript
:app:checkDebugManifest
:app:generateDebugBuildConfig
:app:prepareLintJar UP-TO-DATE
:app:generateDebugResValues
https://blog.csdn.net/ziwang_
:app:generateDebugResources
```

其实说到这里各位读者理应对 task 有一个基本的概念，一个 task 就是一个函数，没有什么神秘的地方。想要知道当前 Android 项目中共有哪些 task，可以在项目目录下输入：

```
./gradlew tasks (Mac、Linux)
gradlew tasks (Windows, 后文同)
```

将会输出类似以下信息：

```
> Task :tasks

-----
All tasks runnable from root project
-----

Android tasks
-----
app:androidDependencies - Displays the Android dependencies of the project.
app:signingReport - Displays the signing info for each variant.
app:sourceSets - Prints out all the source sets defined in this project.

Build tasks
-----
app:assemble - Assembles all variants of all applications and secondary packages
.
lib:assemble - Assembles the outputs of this project.
app:assembleAndroidTest - Assembles all the Test applications.
app:assembleDebug - Assembles all Debug builds.
app:assembleRelease - Assembles all Release builds.
app:build - Assembles and tests this project.
lib:build - Assembles and tests this project.
app:buildDependents - Assembles and tests this project and all projects that dep
```

这仅是部分 task，如果想要查看全部的 task 可以添加 `--all` 参数：

```
./gradlew tasks --all
```

拉到最后的 Other task 部分：

Other tasks

```
app:assembleDebugAndroidTest  
app:assembleDebugUnitTest  
app:assembleReleaseUnitTest  
app:bundleAppClassesDebug  
app:bundleAppClassesDebugAndroidTest  
app:bundleAppClassesDebugUnitTest  
app:bundleAppClassesRelease  
app:bundleAppClassesReleaseUnitTest  
app:checkDebugManifest  
app:checkReleaseManifest  
clean  
app:compileDebugAidl  
app:compileDebugAndroidTestAidl  
app:compileDebugAndroidTestJavaWithJavac  
app:compileDebugAndroidTestNdk  
app:compileDebugAndroidTestRenderscript  
app:compileDebugAndroidTestShaders  
app:compileDebugJavaWithJavac  
app:compileDebugNdk  
app:compileDebugRenderscript  
app:compileDebugShaders  
app:compileDebugUnitTestJavaWithJavac https://blog.csdn.net/ziwang_
```

可以看到所有的 task，可能有经验的读者更是感觉到亲切，提取部分 task 说明下：

- `compileDebugJavaWithJavac`: 编译 java 文件
- `processDebugManifest`: 生成最终 AndroidManifest 文件
- `compileDebugAidl`: 编译 AIDL 文件
- `packageDebug`: 打包成 apk

而如果为 release 包开启了混淆，可看到还有 `transformClassesAndResourcesWithProguardForRelease` task，即为 release 包混淆。

这些 task 实际上贯穿了开发者们的日常开发流程，只是 IDE 在上层封装了一层，开发者们点点按钮就完成了这些操作。当然也有读者会疑惑，这些 task 从哪里来的？为什么没有在 `build.gradle` 中看到蛛丝马迹？这些 task 都是 plugin 中的，在 `build.gradle` 顶部 `apply` 的 plugin 中的（`com.android.application/com.android.library`）。在后续的文章中笔者将会介绍如何找到这些 task 的源码、如何阅读它们、如何写 task、如何写 plugin 以及解析较为知名的 gradle 插件源码，这些也正是 Android 开发者学习 gradle 的目的所在。

Gradle 构建周期

根据 Build phases 文档 和 Settings file 文档可以知道，Gradle 构建周期分为：

•

Initialization: Gradle 支持单个和多项目构建。在 Initialization 阶段，Gradle 将会确定哪些项目将参与构建，并为每个项目创建一个 Project 对象实例。对于 Android 项目来说即为执行 `setting.gradle` 文件。那么日常开发中 `setting.gradle` 文件是如何书写的呢，假设当前应用中除 `app` 以外还有一个 `a module` 和 `b module`，那么 `setting.gradle` 文件应类似如下：

```
include ':app', ':a', ':b'
```

复制代码

所以 Gradle 将会为它们三个分别创建一个 Project 对象实例。

•

Configuration: 在这一阶段项目配置对象，所有项目的构建脚本将会被「执行」，这样才能够知道各个 task 之间的依赖关系。需要说的一点是，这里提到的「执行」可能会稍有一些歧义：

```
task a {  
}  
  
task testBoth {  
    // 依赖 a task 先执行  
    dependsOn("a")  
    println '我会在 Configuration 和 Execution 阶段都会执行'  
    doFirst {  
        println '我仅会在 testBoth 的 Execution 阶段执行'  
    }  
    doLast {  
        println '我仅会在 testBoth 的 Execution 阶段执行'  
    }  
}
```

复制代码

写在 task 闭包中的内容是会在 Configuration 中就执行的，例如上面的 dependsOn("a") 和 println 内容；而 doFirst {}/doLast {} 闭包中的内容是在 Execution 阶段才会执行到（doFirst {}/ doLast {} 实际上就是给当前 task 添加 Listener，这些 Listeners 只会在当前 task Execution 阶段才会执行）。

建议各位读者将这个 task 复制黏贴到 app 的 build.gradle 中，然后观察点击 Android Studio 的 Sync 按钮(含 Configuration 阶段)和在命令行输入 ./gradlew app:testBoth 时命令行 (testBoth 的 Execution 阶段) 输出的结果。便可理解上述闭包内容的执行过程。当然你也可以查看 Settings file 中的案例来理解。

对于 Android 项目来说即为执行各个 module 下的 build.gradle 文件，这样各个 build.gradle 文件中的 task 的依赖关系就被确认下来了（例如 assembleDebug task 的执行依赖于其他 tasks 先执行，而这个依赖关系的确定就是在 Configuration 阶段）。

Execution: task 的执行阶段。首先执行 doFirst {} 闭包中的内容，最后执行 doLast {} 闭包中的内容。

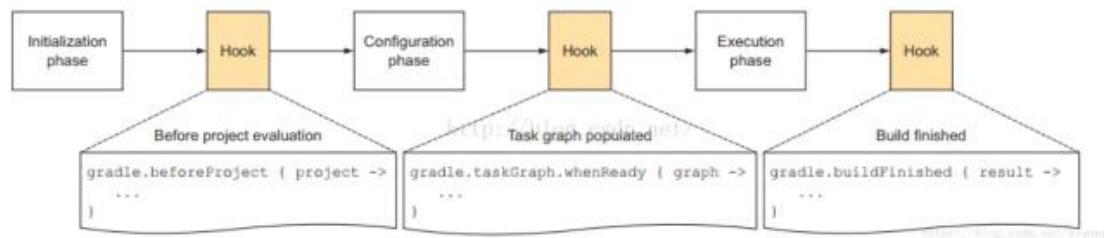
在命令行中执行某一个 task，是可以清晰地看见每一个执行流程：

```
→ ObfuscatedTest ./gradlew assembleD
Starting a Gradle Daemon (subsequent builds will be faster)
<-----> 0% INITIALIZING [0s] https://blog.csdn.net/ziwang_
```

```
→ ObfuscatedTest ./gradlew assembleD
Starting a Gradle Daemon (subsequent builds will be faster)
<-----> 50% CONFIGURING [4s] https://blog.csdn.net/ziwang_
```

```
<-----> 3% EXECUTING [7s]
```

在命令行输入 `./gradlew xxx` 并按下回车之后的执行过程中，就会执行上面的流程，再通过一张深入理解 Android 之 Gradle 中的图加深印象——



然而在笔者目前为止碰到的 Gradle 开发中，很少提及到上述的 hook 系列。用的比较多的一个是前文提到的 task 中的 `doFirst {} / doLast {}`，再一个就是 Project 在 Configuration 阶段结束的 hook，前文也提到**Configuration 阶段将会执行每一个 Project 的 build.gradle 文件**，那么如何监听这每一个 Project 的引入后的点呢？通过 `Project.afterEvaluate` ——

```
//afterEvaluate { Project project ->
//  println 'hook afterEvaluate'
//}

task hook { Project project ->
    afterEvaluate {
        println 'hook afterEvaluate'
    }
}
```

[复制代码](#)

那么为什么要 hook Project 的 `afterEvaluate` 阶段呢？因为在 `afterEvaluate` 阶段的时候，当前 Project 内的 task 信息才能被掌握，例如想在 `assembleDebug` task 前输出一段信息，那么在 `app/build.gradle` 中撰写以下代码：

```
// task badHook {
//   tasks.findByName("assembleDebug").doFirst {
//     println 'hook afterEvaluate from BadHook'
//   }
// }

task assDHook {
    afterEvaluate {
        tasks.findByName("assembleDebug").doFirst {
            println 'hook afterEvaluate from assDHook'
        }
    }
}
```

[复制代码](#)

此时如果调用 assembleDebug task 的话，首先是 Initialization 阶段，再是 Configuration 阶段，该阶段中当 appProject 到达 afterEvaluate 阶段的时候 appProject 中的 tasks 信息将会全部被获取，当然这其中也包括 assembleDebug task，所以此时再给它添加一个 doFirst {} 闭包便可以达到目的。而像 badHook 闭包的内容将会在 Initialization 阶段执行，此时 appProject 并没有获取全部的 task 信息，将会导致压根找不到 assembleDebug 的错误。

Gradle 系列二

task 撰写

task 声明

根据[官方文档](#)和 Task#create() 可以知道，task 的基本写法可以是如下四种：

```
task myTask  
task myTask { configure closure }  
task (myTask) { configure closure }  
task (name: myTask) { configure closure }
```

复制代码

每一个 task 都有自己的名字，这样开发者才能调用它，例如调用上面的 task:

```
./gradlew myTask
```

但是有一个问题，倘若当前项目的 app module 和 a module 都含有一个名为 myTask 的 task，那么会不会起冲突，该如何调用它们？答案是不会冲突，调用方式如下

```
./gradlew app:myTask (调用 app module 的 myTask )  
./gradlew a:myTask ( 调用 a module 的 myTask )
```

通过 ProjectName:taskName 的形式便可以指定唯一绝对路径去调用指定 Project 的指定 task 了。

扩展

根据 Task#create() 可以知道，task 的创建是可以声明参数的，除了上述的 name 参数之外，还有如下几种：

•

type: 默认为 DefaultTask。类似于父类。在后文中将会提及该参数。

•

•

dependsOn: 默认为`[]`。希望依赖的 `tasks`, 等同于 `Task.dependsOn(Object... path)` 中的 `path`。在后文中将会提及该参数。

-
-

action: 默认为 `null`。等同于 `Task.doFirst { Action }` 中的 `Action`。

```
task (name: actionTest, action: new Action<Task>() {  
    @Override  
    void execute(Task task) {  
        println 'hello'  
    }  
}) {  
}
```

[复制代码](#)

等同于

```
task (name: actionTest) {  
    doFirst {  
        println 'hello'  
    }  
}
```

[复制代码](#)

-

override: 默认为 `false`。是否替换已存在的 `task`。

-
-

group: 默认为 `null`。`task` 的分组类型。

-
-

description: 默认为 `null`。`task` 描述。

-
-

constructorArgs: 默认为 `null`。传给 `task` 构造函数的参数。

-

后面四种大部分开发过程中应该不怎么会用到，有需要的读者自行查阅文档。

task 内容格式

1.

根据官方文档以及前一篇文章中可以知道，如果想给 `task` 添加操作，可以添加在 `doLast {}/doFirst {}` 等闭包中，例如：

```
task myTask {  
    doFirst {  
        println 'myTask 最先执行的内容'  
    }  
    doLast {  
        println 'myTask 最后执行的内容'  
    }  
    // warning  
    // println 'Configuration 阶段和 Execution 阶段皆会执行'  
}
```

复制代码

切记大部分的内容是写在 `doLast{}` 或 `doFirst{}` 闭包中，因为在如果写在 `task` 闭包中的话，会在 Configuration 阶段也被执行。

1. 根据官方文档可知，为了提高 `task` 复用性，Gradle 还支持 `Task` 类的书写——

2.1 将下述代码写在 `build.gradle` 中，并用 `@TaskAction` 标记想要执行的方法。

```
class GreetingTask extends DefaultTask {  
    String greeting = 'hello from GreetingTask'  
  
    @TaskAction  
    def greet() {  
        println greeting  
    }  
}
```

复制代码

2.2 在 `build.gradle` 中撰写 `task` 调用 `GreetingTask` 类：

```
// Use the default greeting
task (name: hello , type: GreetingTask)

// Customize the greeting
task (name: greeting , type: GreetingTask) {
    greeting = 'greetings from GreetingTask'
}
```

复制代码

2.3 调用该 task——

```
./gradlew hello

> Task :app:hello
hello from GreetingTask

./gradlew greeting

> Task :app:greeting
greetings from GreetingTask
```

复制代码

所以看到这里应该不仅能够理解 Task 类的书写，并且应该能够大致明白 type 这个参数的含义了。

不知道会不会和笔者一样事儿逼的读者此时会疑惑 @TaskAction 修饰的方法和 doLast {} 以及 doFirst {} 闭包的执行顺序是怎样的？

```
task (name: hello, type: GreetingTask) {
    doFirst {
        def list = getActions()
        for (int i = 0; i < list.size(); i++) {
            println list.get(i).displayName
        }
    }

    doLast {
    }
}
```

复制代码

首先声明 doFirst {} 和 doLast {} 闭包；然后戳进 DefaultTask 源码并追踪到顶级父类 AbstractTask 中可以看到内部通过使用 actions 存储所有执行的 Action，并通过 getAction() 暴露；actions 是 List 类型，内部的元素类型是 ContextAwareTaskAction，该接口又实现了 Describable，Describable 仅声明了一个 getDisplayName() 方法，所以可以直接通过 displayName 获取该 Action 的名称。

理解以上三步即可完成上述 task 撰写，在命令行中试试——

```
./gradlew hello
```

```
> Task :app:hello
Execute doFirst {} action
Execute greet
Execute doLast {} action
```

Gradle 内部将会自动为变量设置 setter、getter 方法，所以当一个 Gradle 有 getXxx() 方法时，可以直接使用 xxx 变量。如果不清楚这个细节，建议回顾上一篇文章的附录。

task 依赖关系

开发者常使用 `dependsOn` 来指定依赖关系（另外两种是指定 task 执行顺序，详见文档 [Task Dependencies and Task Ordering](#)），如下：

```
task a {
    doLast {
        println 'a'
    }
}

task b {
    dependsOn('a')
    doFirst {
        println 'b'
    }
}
```

复制代码

不妨将以上代码写在 app `build.gradle` 文件下，当执行 task b 的时候，会输出如下信息

```
./gradlew task app:b
> Task :app:a
a
> Task :app:b
b
```

可以看到，由于 task b 需要依赖 task a，所以 task b 执行的时候会先执行 task a。

有经验的开发者如果在命令行中试过 `assembleDebug` 等命令会发现，它们的执行将会依赖于许多其他 task。所以不妨在命令行中试试 `./gradlew assembleDebug` 观察输出结果。

task 实战

```
install && launch apk
```

`com.android.application` 自带 `installDebug` task，开发者可以使用 `installDebug` 安装当前项目 apk：

```
./gradlew installDebug  
> Task :app:installDebug  
Installing APK 'app-debug.apk' on 'xxxxx' for app:debug Installed on 1 device.
```

但是似乎看起来有些不尽人意的地方，例如开发者希望安装的时候能够顺带能够启动该 app。那么该如何做呢？

首先从问题的可行性上来进行分析，开发者的直觉告诉我们是可以通过 gradle 实现的——命令行可以安装、启动 apk——`adb install -r app-debug.apk` 和 `adb shell am start -n 包名/首 Activity`。所以关键点就是如何通过 gradle 调用命令行代码以及如何获取到 包名/首 Activity 信息。

1.

开发者的直觉同样告诉我们 Gradle 开发文档中有关于命令行调用的信息，只需要使用 `exec {}` 闭包就好了。

2.

3.

如何获取 包名/首 Activity 信息？可以通过 `AndroidManifest.xml` 来获取。部分经验丰富的开发者知道——打入 apk 中的 `AndroidManifest.xml` 文件并不是我们平常写的 `AndroidManifest.xml`，而是 apk 编译后位于 `Project/app/build/intermediates/manifests/full/debug/` 包下的 `AndroidManifest.xml`（当然，如果是 Release 包的话，应该是 `Project/app/build/intermediates/manifests/full/release/` 包下）。

4.

1.

包名就是 `android` 闭包下的 `defaultConfig` 闭包下的 `applicationId`。

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 27
    defaultConfig {
        applicationId "com.joker.testapk"
        minSdkVersion 21
        targetSdkVersion 27
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}
```

<https://blog.csdn.net/xiwang>

目标 Activity 则是包含 action 为 `android.intent.action.MAIN` 的 Activity。

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="...ule"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>

            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>
</application>

</manifest>
```

<https://blog.csdn.net/xiwang>

理解了以上内容，便不难理解下面的内容：

[复制代码](#)

```
task installAndRun(dependsOn: 'assembleDebug') {
    doFirst {
        exec {
            workingDir "${buildDir}/outputs/apk/debug"
            commandLine 'adb', 'install', '-r', 'app-debug.apk'
        }
        exec {
            def path = "${buildDir}/intermediates/manifests/full/debug/AndroidManifest.xml"
            // xml 解析
            def parser = new XmlParser(false, false).parse(new File(path))
            // application 下的每一个 activity 结点
            parser.application.activity.each { activity ->
                // activity 下的每一个 intent-filter 结点
                activity.'intent-filter'.each { filter ->
                    // intent-filter 下的 action 结点中的 @android:name 包含 android.intent.action.MAIN
                    if (filter.action.@"android:name".contains("android.intent.action.MAIN")) {
                        def targetActivity = activity.@"android:name"
                        commandLine 'adb', 'shell', 'am', 'start', '-n',
                            "${android.defaultConfig.applicationId}/${targetActivity}"
                    }
                }
            }
        }
    }
}
```

1. install apk 的前提必须是得有一个 apk , 所以势必需要依赖 `assembleDebug` task。

实际上 `installDebug` task 也是依赖 `assembleDebug` task 的，不妨可以试试——

[复制代码](#)

```
task showInstallDepends {
    doFirst {
        println project.tasks.findByName("installDebug").dependsOn
    }
}
```

```
./gradlew showInstallDepends
> Configure project :app
[task 'installDebug' input files, assembleDebug]
```

2. `exec` 闭包中的几个参数提及下——

2.1 `workingDir` : 工作环境，参数为 `File` 格式。默认为当前 project 目录。

2.2 `commandLine` : 需要命令行执行的命令，参数为 `List` 格式。

3. 前一篇文章中提到——

说白了它们其实就是一些闭包、一些固定格式，正是因为它们的格式是固定的，task 才能够读取到相应的数据完成相应的事情。

在第二个 `exec` 闭包的第八行就很好的体现了这一点，通过

`{android.defaultConfig.applicationId}` 直接获取到 Gradle 文件中 `android` 闭包下的 `defaultConfig` 闭包下的 `applicationId` 的值。由此就获得了当前应用的包名。

当然，除了 Gradle 能够调用命令行以外，实际上 groovy 也是可以调用命令行的，但在此就不做扩展了。

1.

至于最先启动的 `Activity`, 肯定是 `action` 为 `android.intent.action.MAIN` 的 `Activity`, 那么问题就是变成如何在 `AndroidManifest.xml` 中寻找到该 `Activity` 的事了——作为一个合格的老司机，应该能够想到 `groovy` 一定会提供相应的 `xml` 解析 API 的，至于具体的使用笔者就不在此扩展了，留给各位读者去源码中探索成长。

2.

3.

除去上面的信息以外，还需要什么？还需要知道一些 `gradle` 构建的信息——例如 `debug` 包会最终出现在 `${buildDir}/outputs/apk/debug`; 例如 `debug` 包中的 `AndroidManifest.xml` 并不是日常开发中写的那个 `AndroidManifest.xml`（虽然可能它俩基本没什么差异），而是 `${buildDir}/intermediates/manifests/full/debug` 下的 `AndroidManifest.xml`。所以一是希望各位读者日常多去翻翻 `build` 文件夹，二是要知道 `${buildDir}(build` 文件夹)有多么重要，因为 `Gradle` 构建 `apk` 的过程中，但凡有输出文件那么基本都会存在这个文件夹中，所以多去翻一翻。

4.

由此之后，可以在命令行输入以下命令：

```
./gradlew installAndRun  
> Task :app:installAndRun  
[ 4%] /data/local/tmp/app-debug.apk  
[ 8%] /data/local/tmp/app-debug.apk  
[ 12%] /data/local/tmp/app-debug.apk ...  
Success  
Starting: Intent {com.test.Test/TestActivity}
```

至此便完成了一个安装并启动 apk 的 task 撰写了。

hook assets

上面的 task 看起来似乎和 Android 的构建过程并无多大关系，没错，那么接下来不妨深层次接触试试——通过 hook 原生 task 实现更改打包中的文件——在打包过程中向 assets 插入一张图片。

尽管这看起来丝毫没卵用

在打包流程中，有一个 task 名为 packageDebug，该 task 是打包文件生成 apk 的——

```
:app:transformDexArchiveWithExternalLibsDexMergerForD  
:app:transformDexArchiveWithDexMergerForDebug UP-T0-D  
:app:mergeDebugJniLibFolders UP-T0-DATE  
:app:transformNativeLibsWithMergeJniLibsForDebug UP-T  
:app:processDebugJavaRes NO-SOURCE  
:app:transformResourcesWithMergeJavaResForDebug UP-T0-  
:app:validateSigningDebug UP-T0-DATE  
:app:packageDebug ←  
:app:assembleDebug https://blog.csdn.net/ziwang_
```

接着，不妨在命令行键入以下命令：

```
./gradlew help --task "packageDebug"
```

Type

```
    PackageApplication  
(com.android.build.gradle.tasks.PackageApplic  
ation)
```

可以看到，该 task 的 type 是 PackageApplication——

```
/💡 Task to package an Android application (APK). */  
public class PackageApplication extends PackageAndroidArtifact {
```

不妨再看看它的父类 PackageAndroidArtifact：

```
protected FileCollection assets; https://blog.csdn.net/ziwang_
```

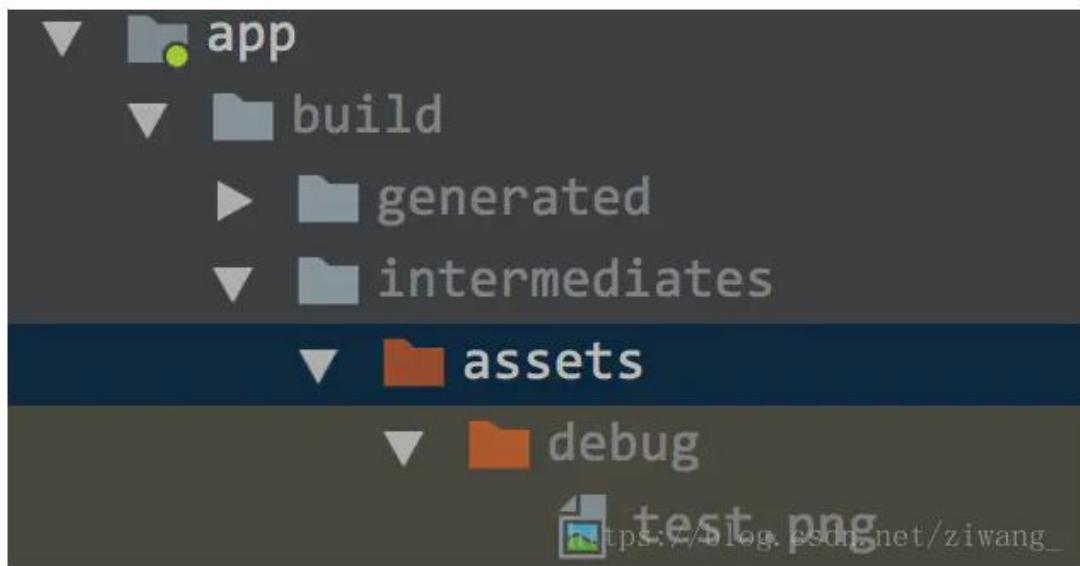
看到关键信息，该 task 中有一个类型为 `FileCollection assets` 字段，这便是最终打入 apk 中的那个 assets 了。所以不难写出以下代码——

```
task hookAssets {
    afterEvaluate {
        tasks.findByName("packageDebug").doFirst { task -
            copy {
                from "${projectDir.absolutePath}/test.png"
                into "${task.assets.asList()}"
            }
        }
    }
}
```

复制代码

1. 在 project `afterEvaluate` 之后找到 `packageDebug` task
2. 不妨在 app 目录下放入一个 `test.png`，使用 `copy {}` 闭包，`from` 填入的参数为 `test.png` 的路径，`into` 填入的参数为输出的路径，也就是 `assets` 的路径。

可以看到 `/app/build/intermediates/assets/debug/` 下存有 `test.png`



同样地，解压 apk 文件也可以看到——



一个实打实的 Gradle task hook 流程就这么操作完了。

Gradle 系列三

撰写 plugin

准备工作

1. 新建一个 Android 项目。
2. 新建一个 java library module，该 module 必须命名为 buildSrc。
3. 将 src/main/java 改成 src/main/groovy

基本实现

1.

新建一个 xxxPlugin.groovy 并实现 Plugin 接口，例如：

```
import org.gradle.api.Plugin
import org.gradle.api.Project

class TestPlugin implements Plugin<Project> {
    @Override
    void apply(Project project) {
        project.task('pluginTest') {
            doLast {
                println 'Hello World'
            }
        }
    }
}
```

复制代码

可以看到，上述 `plugin` 仅是在 `apply()` 方法内部创建了一个名为 `pluginTest` 的 `task`。

由于 Kotlin/Java 与 groovy 的兼容，所以并非一定要创建 groovy 文件，也可以是 `xxxPlugin.java`/`xxxPlugin.kotlin`。

- 既然 `plugin` 已经就这么简单的实现了，那么如何应用到实际项目中呢？在 `build.gradle` 文件中添加如下信息：

```
apply plugin: TestPlugin
```

至此之后，不妨在命令行调用 `pluginTest` task 看看是否有效果——

```
./gradlew pluginTest  
> Task :app:testPlugin  
Hello from the TestPlugin
```

扩展

随着项目的急速发展，有朝一日发现有时候不想输出 `Hello World` 而是希望这个 `pluginTest` task 可以根据开发者的需求进行配置。

1.

创建一个 `xxxExtension.groovy` 文件（当然，也可以用 Java/Kotlin 来写），实际上就是和 JavaBean 差不多的类，类似如下：

```
class TestPluginExtension {  
    String message = 'Hello World'  
}
```

复制代码

在 `Plugin` 类中获取闭包信息，并输出：

```
class TestPlugin implements Plugin<Project> {  
    void apply(Project project) {  
        // Add the 'testExtension' extension object  
        def extension = project.extensions.create('testExtension', TestPluginExtension)  
        project.task('pluginTest') {  
            doLast {  
                println extension.message  
            }  
        }  
    }  
}
```

复制代码

- 第四行通过 `project.extensions.create(String name, Class<T> type, Object... constructionArguments)` 来获取 `testExtension` 闭包中的内容并通过反射将闭包的内容转换成一个 `TestPluginExtension` 对象。
- 在 `build.gradle` 中添加一个 `testExtension` 闭包：

```
testExtension {  
    message 'Hello Gradle'  
}  
在命令行键入以下信息
```

```
./gradlew pluginTest
```

将会看到输出结果——

```
> Task :app:pluginTest  
Hello Gradle
```

项目化

到目前为止谈到的东西都还是一个普通的、不可以发布到仓库的插件，如果想要将插件发布出去供他人和自己在项目中 apply，需要进行以下步骤将插件变成一个 Project——

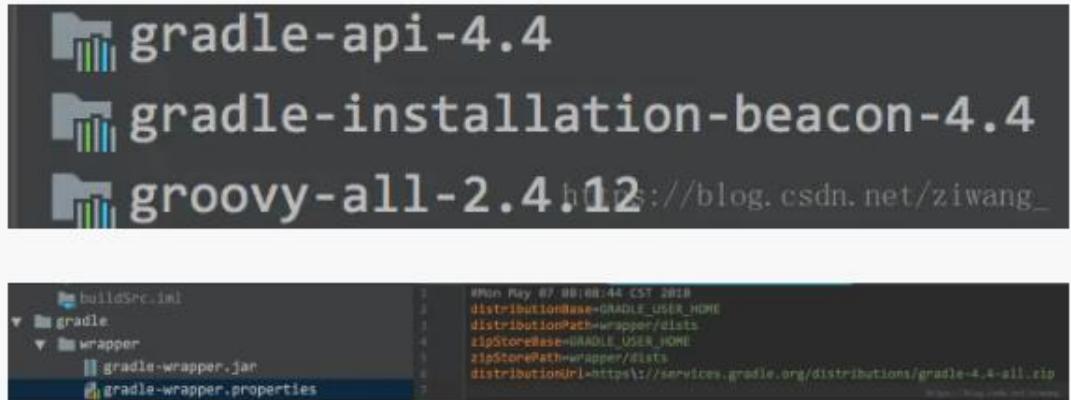
1.

更改 `build.gradle` 文件内容：

```
apply plugin: 'groovy'
```

```
dependencies {  
  
    compile gradleApi()  
  
    compile localGroovy()  
  
}
```

此时可以观察到 External Libraries 中多出了 gradle-api/gradle-installation-beacon/groovy 库。其中，gradle 的版本是基于项目下 gradle wrapper 中配置的版本——



- 创建 `src/main/resources/META-INF/gradle-plugins/插件名.properties`，例如

`src/main/resources/META-INF/gradle-plugins/com.sample.test.properties`，然后将 properties 文件内容改为 `implementation-class=Plugin` 路径，例如 `implementation-class=com.sample.test.TestPlugin`。

- 在 `build.gradle` 文件中通过 `apply plugin: '插件名'` 引入插件 ——
`apply plugin: 'com.sample.test'`。
- 在命令行键入以下信息：

```
./gradlew pluginTest
```

将会看到输出结果——

```
> Task :app:pluginTest
```

```
Hello Gradle
```

当然，以上仅是告诉各位读者如何将 plugin 项目化，并未涉及到如何将 plugin 提交到仓库中，关于 jcenter 仓库提交方式可借鉴手摸手教你如何把项目提交到 jcenter，其他仓库提交方式读者可自行搜索。

实战

Android 打包过程中，一个 task 接着一个 task 的执行，每个 task 都会执行一段特定的事情（例如第一篇文章中提到的几个 task），所以在 Gradle 插件的开发中，如果是针对打包流程的更改，实际上大部分都是 hook 某一个 task 来达

到目的——例如我司的 mess 通过 hook `transformClassesAndResourcesWithProguardForDebug` task (Gradle v2.0+ task) 来实现对四大组件以及 View 的混淆的；美丽说的 ThinRPlugin 是通过 hook `transformClassesWithDexForDebug` (Gradle v2.0+ task) 来实现精简 R.class/R2.class 的。

因为 Android 现有的 task 已经很完善了，所以如果想要达到目的，只需要了解相应的 task 并在其之前或之后做一些操作即可。

为了示例而示例的简单例子实在不多，笔者只能拿起上篇文章中的示例——在 app 目录下创建 pic 文件夹，并添加一个名为 test 的 png 图片，hook apk 打包流程将该图片添加入 apk 的 assets 文件夹。

尽管这看起来真的很没有卵用。

这次为了符合实际开发要求，不妨提升一定的难度——仅在 release 包中向 assets 添加图片，而 debug 包不向 assets 中添加图片。在实际开发中有很多这样的需求，例如前文提到的 mess 是对 apk 源码进行混淆的，那么日常开发者运行的 debug 包有必要执行该 task 么？显然并不需要，应该仅在发布的时候打 release 包的时候执行该 task 就好了。

那么如何知道当前 task 是为 release 服务的呢？简单的寻找到 name 为 `packageRelease` 的 task 是肯定不行的，日常开发中项目时常有很多种变体，例如在 `app/build.gradle` 中输入以下代码：

```
android {  
    ...  
    flavorDimensions "api", "mode"  
  
    productFlavors {  
        demo {  
            dimension "mode"  
        }  
  
        full {  
            dimension "mode"  
        }  
  
        minApi23 {  
            dimension "api"  
            minSdkVersion '23'  
        }  
  
        minApi21 {  
            dimension "api"  
            minSdkVersion '21'  
        }  
    }  
}
```

此时的变种共有 $3 (\text{debug, release, androidTest}) * 2 (\text{demo, full}) * 2 (\text{minApi23, minApi21})$ 共计 12 种，截图如下：

```
packageMinApi21DemoDebug
packageMinApi21DemoDebugAndroidTest
packageMinApi21DemoRelease
packageMinApi21FullDebug
packageMinApi21FullDebugAndroidTest
packageMinApi21FullRelease
packageMinApi23DemoDebug
packageMinApi23DemoDebugAndroidTest
packageMinApi23DemoRelease
packageMinApi23FullDebug
packageMinApi23FullDebugAndroidTest
packageMinApi23FullRelease
```

那么如何为以上所有的 release 变种包的 assets 中都填入图片呢？

根据官方文档可以知道开发者可以通过 `android.applicationVariants.all` 获取到当前所有的 apk 变体，该变体的类型为 `ApplicationVariant`，其父类 `BaseVariantOutput` 中含 `name` 字段，该字段实际上就是当前变体的名字，那么其实只需要判断该 `name` 字段是否包含 `release` 关键字即可。
创建 plugin 的基本流程已经在前文中阐述过了，直接进行核心 plugin 的撰写，`HookAssetsPlugin` 源码如下：

```
import com.android.build.gradle.api.ApkVariantOutput
import com.android.build.gradle.api.ApplicationVariant
import com.android.build.gradle.tasks.PackageApplication
import org.gradle.api.Plugin
import org.gradle.api.Project

class HookAssetsPlugin implements Plugin<Project> {
    @Override
    void apply(Project project) {
        project.afterEvaluate {
            project.plugins.withId('com.android.application') {
                project.android.applicationVariants.all { ApplicationVariant variant ->
                    variant.outputs.each { ApkVariantOutput variantOutput ->
                        if (variantOutput.name.equalsIgnoreCase("release")) {
```

```
variantOutput.packageApplication.doFirst { PackageApplication task  
->  
    project.copy {  
        from "${project.projectDir.absolutePath}/pic/test.png"  
        into "${task.assets.asPath}"  
    }  
}  
}  
}  
}  
}  
}  
}  
}  
}  
}  
}  
}
```

- 在第一篇文章中就阐述过，只能在 `project.afterEvaluate` 闭包中才能获取到当前 `project` 中的所有 `task`。
 - • 通过 `project.plugins.withId('com.android.application')` 确保当前 `project` 是 Android app project 而不是 Android library project，以此来避免无效操作，毕竟 `package task` 是 `com.android.application` 中的 `task`。
 - • 通过 `project.android.applicationVariants.all` 获取所有变体信息。
 - • 通过观察 `ApplicationVariant` 类的父类 `BaseVariant` 中 `outputs` 字段可知该字段代表着当前变体的输出信息（`DomainObjectCollection` 类型），`BaseVariantOutput` 的子类 `ApkVariantOutput` 中的 `packageApplication` 即为上一篇文章中所说的 `PackageAndroidArtifact task` 了。
 - • 判断当前变体是否是 `release` 的变体。（通过 `variantOutput.name.equalsIgnoreCase("release")/variant.name.equalsIgnoreCase("release")` 都是可以的。）
 - • hook 步骤 4 中所说的 `PackageAndroidArtifact task`，将图片复制到 `assets` 中。

三十一、RxJava

RxJava 名词以及如何使用

1. 整体思路

拆轮子这也是第四回了，套路也算得到了很好的验证，顺着常用的场景/用例出发，理解整个过程、结构、原理，不要沉迷于细节，先对常用的内容有一个全局的概览，每一块的细节再按需深入。入手新项目也是这个思路。

对 RxJava 来说，基于目前已有的认识，我觉得主要应该抓住四个方面：

- 事件流源头 (**observable**) 怎么发出数据
- 响应者 (**subscriber**) 怎么收到数据
- 怎么对事件流进行操作 (**operator/transformer**)
- 以及整个过程的调度 (**scheduler**)

另外还有三点也值得一提：

- **backpressure**
- **hook**
- 测试

2, Hello world

我们先看一个最简单的 Hello world 例子：

```
Observable.just("Hello world")
    .subscribe(word -> {
        System.out.println("got " + word + " @ "
            + Thread.currentThread().getName());
    });
}
```

2.1, just

逐行往下看显然是最自然的方式，那我们先看看 **just()**：

```
// Observable.java
public static <T> Observable<T> just(final T value) {
    return ScalarSynchronousObservable.create(value);
}

// ScalarSynchronousObservable.java
public static <T> ScalarSynchronousObservable<T> create(T t) {
    return new ScalarSynchronousObservable<T>(t);           // 1
}

protected ScalarSynchronousObservable(final T t) {
    super(RxJavaHooks.onCreate(new JustOnSubscribe<T>(t))); // 2
    this.t = t;
}
```

这里一定要注意，不要沉迷于细节，否则上万行的代码绝不是一两天能看出个大概来的。

1. 我们创建的是 `ScalarSynchronousObservable`, 一个 `Observable` 的子类。
2. 我们先跳过 `RxJavaHooks`, 从名字可以得知它是用来做一些 `hook` 的工作的, 那我们就先认为它什么也不做。所以我们传给父类构造函数的就是 `JustOnSubscribe`, 一个 `OnSubscribe` 的实现类。

`Observable` 的构造函数接受一个 `OnSubscribe`, 它是一个回调, 会在 `Observable#subscribe` 中使用, 用于通知 `observable` 自己被订阅, 它是怎么使用的, 我们马上就能看到。

2.2, subscribe

我们接着看 `subscribe()`:

```

public final Subscription subscribe(final Action1<? super T> onNext) {
    // 省略参数检查代码
    Action1<Throwable> onError =
        InternalObservableUtils.ERROR_NOT_IMPLEMENTED;
    Action0 onCompleted = Actions.empty();
    return subscribe(new ActionSubscriber<T>(onNext,
        onError, onCompleted));                                // 1
}

public final Subscription subscribe(Subscriber<? super T> subscriber) {
    return Observable.subscribe(subscriber, this);
}

static <T> Subscription subscribe(Subscriber<? super T> subscriber,
    Observable<T> observable) {
    // 省略参数检查代码
    subscriber.onStart();                                    // 2

    if (!(subscriber instanceof SafeSub
        subscriber = new SafeSubscriber<T>());           // 3
    }

    try {
        RxJavaHooks.onObservableStart(observable,
            observable.onSubscribe).call(subscriber);      // 4
        return RxJavaHooks.onObservableReturn(subscriber); // 5
    } catch (Throwable e) {
        // 省略错误处理代码
    }
}

```

我们抓住主要逻辑:

1. 我们首先对传入的 `Action` 进行包装, 包装为 `ActionSubscriber`, 一个 `Subscriber` 的实现类。

2. 调用 `subscriber.onStart()` 通知 `subscriber` 它已经和 `observable` 连接起来了。这里我们就知道，`onStart()` 就是在我们调用 `subscribe()` 的线程执行的。
3. 如果传入的 `subscriber` 不是 `SafeSubscriber`，那就把它包装为一个 `SafeSubscriber`。
4. 我们再次跳过 `hook`，认为它什么也没做，那这里我们调用的其实就是 `observable.onSubscribe.call(subscriber)`。这里我们就看到了前面提到的 `onSubscribe` 的使用代码，在我们调用 `subscribe()` 的线程执行这个回调。
5. 跳过 `hook`，那么这里就是直接返回了 `subscriber`。`Subscriber` 继承了 `Subscription`，用于取消订阅。

关于 `SafeSubscriber`，我们跳过源码，直接看它的文档，其中说明了这个类的作用：保证 `Subscriber` 实例遵循 `Observable contract`。至于具体怎么保证的，以及 `contract` 的内容，大家直接看文档即可，这里不再赘述。

好了，我们已经看完了例子中两个调用的代码，但是 `Hello world` 是怎么被传递到打印的代码里的呢？别急，就在 `observable.onSubscribe.call(subscriber)` 中。

2.3, OnSubscribe

还记得 `just()` 的实现中，我们创建了一个 `JustOnSubscribe` 吗？这里我们执行的就是它实现的 `call()` 函数：

```
// ScalarSynchronousObservable.java
static final class JustOnSubscribe<T> implements OnSubscribe<T> {
    // ...

    @Override
    public void call(Subscriber<? super T> s) {
        s.setProducer(createProducer(s, value));
    }
}

static <T> Producer createProducer(Subscriber<? super T> s, T v) {
    // ...
    return new WeakSingleProducer<T>(s, v);
}
```

这里我们就是为 `subscriber` 设置了一个 `WeakSingleProducer`。

在 RxJava 1.x 中，数据都是从 observable push 到 subscriber 的，但要是 observable 发得太快，subscriber 处理不过来，该怎么办？一种办法是，把数据保存起来，但这显然可能导致内存耗尽；另一种办法是，多余的数据来了之后就丢掉，至于丢掉和保留的策略可以按需制定；还有一种办法就是让 subscriber 向 observable 主动请求数据，subscriber 不请求，observable 就不发出数据。它俩相互协调，避免出现过多的数据，而协调的桥梁，就是 producer。producer 的内容这里不展开，大家可以看 ReactiveIO 的文档，或者 Advanced RxJava 这个系列博客。

2.4, setProducer

我们接着看 `setProducer()` 的实现：

```
// Subscriber.java
public void setProducer(Producer p) {
    long toRequest;
    boolean passToSubscriber = false;
    synchronized (this) {
        toRequest = requested;
        producer = p;
        if (subscriber != null) { // 1
            if (toRequest == NOT_SET) {
                passToSubscriber = true;
            }
        }
        if (passToSubscriber) { // 2
            subscriber.setProducer(producer);
        } else {
            if (toRequest == NOT_SET) { // 3
                producer.request(Long.MAX_VALUE);
            } else {
                producer.request(toRequest);
            }
        }
    }
}
```

这里逻辑比较复杂，但是我们理清我们当前所处的状态，就简单了：

1. 我们这里确实有一层包装，`ActionSubscriber -> SafeSubscriber`。
2. 所以这里我们会发生一次 `pass through`，然后我们会进入 `else` 代码块。
3. 这里所有的 `requested` 初始值都是 `NOT_SET`，所以我们会请求 `Long.MAX_VALUE`，即无限个数据。

2.5, request

那我们再看 `WeakSingleProducer#request()` 的实现:

```
// ScalarSynchronousObservable.java
static final class WeakSingleProducer<T> implements Producer {
    // ...

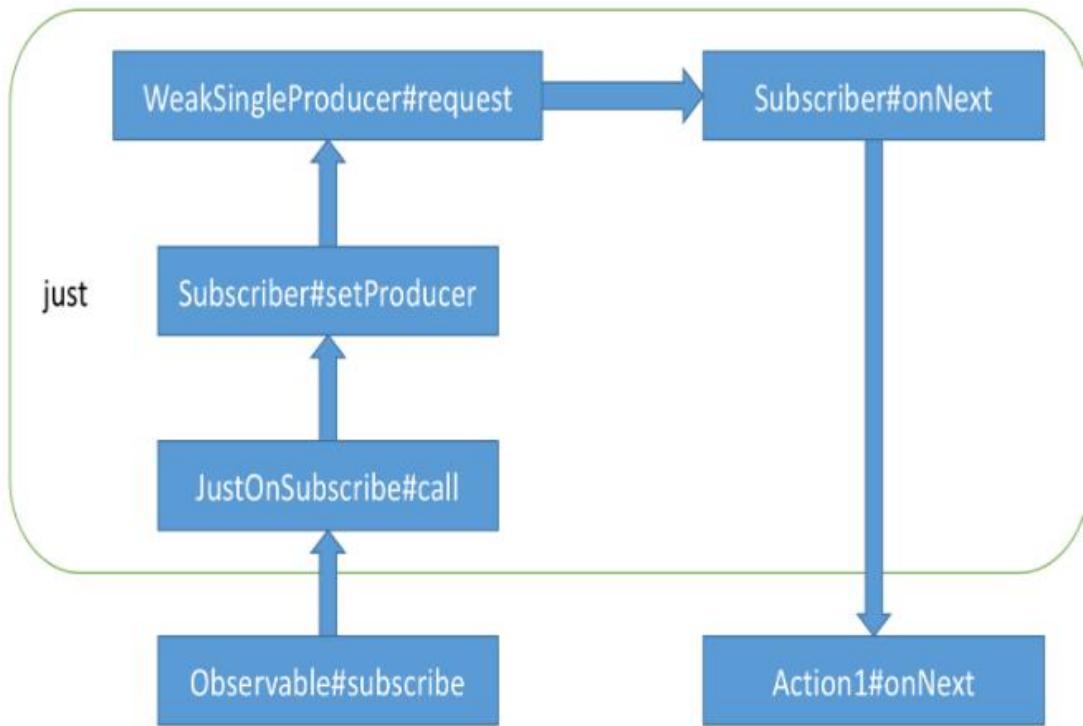
    @Override
    public void request(long n) {
        // 省略状态检查代码
        Subscriber<? super T> a = actual;
        if (a.isUnsubscribed()) {
            return;
        }
        T v = value;
        try {
            a.onNext(v);
        } catch (Throwable e) {
            Exceptions.throwOrReport(e, a, v);
            return;
        }

        if (a.isUnsubscribed()) {
            return;
        }
        a.onCompleted();
    }
}
```

我们看到，在 `request()` 中，终于调用了 `subscriber` 的 `onNext()` 和 `onCompleted()`，那么，`Hello world` 就传递到了我们的 `Action` 中，并被打印出来了。

2.6，完整的过程

到这里我们就已经梳理出完整的调用过程了：



一切行为都由 `subscribe` 触发，而且都是直接的函数调用，所以都在调用 `subscribe` 的线程执行。

下面我们看一下调试时的调用栈：

The screenshot shows the Android Studio debugger's call stack for the `MainActivity.onCreate()` method. The stack trace is as follows:

- `lambda$onCreate$0:71, MainActivity (com.github.piasy.testunderstand.rx)`
- `call:-1, MainActivity$$Lambda$1 (com.github.piasy.testunderstand.rx)`
- `onNext:39, ActionSubscriber (rx.internal.util)`
- `onNext:134, SafeSubscriber (rx.observers)`
- `request:276, ScalarSynchronousObservable$WeakSingleProducer (rx.internal.util)`
- `setProducer:209, Subscriber (rx)`
- `setProducer:205, Subscriber (rx)`
- `call:138, ScalarSynchronousObservable$JustOnSubscribe (rx.internal.util)`
- `call:129, ScalarSynchronousObservable$JustOnSubscribe (rx.internal.util)`
- `subscribe:10307, Observable (rx)`
- `subscribe:10274, Observable (rx)`
- `subscribe:10079, Observable (rx)`
- `onCreate:70, MainActivity (com.github.piasy.testunderstand.rx)`

Annotations highlight several points of interest:

- `onNext:39, ActionSubscriber (rx.internal.util)` and `onNext:134, SafeSubscriber (rx.observers)` are highlighted with red boxes.
- `setProducer:209, Subscriber (rx)` and `setProducer:205, Subscriber (rx)` are highlighted with red boxes.
- `subscribe:10307, Observable (rx)`, `subscribe:10274, Observable (rx)`, and `subscribe:10079, Observable (rx)` are highlighted with red boxes.

确实和我们的分析结果一致。

3，操作符

我们把 Hello world 稍微变复杂一点，使用一个操作符：

```
Observable.just("Hello world")
    .map(String::length)
    .subscribe(word -> {
        System.out.println("got " + word + " @ "
            + Thread.currentThread().getName());
    });
});
```

我们使用了一个 map 操作符，把字符串转换为它的长度。

3.1, map

```
// Observable.java
public final <R> Observable<R> map(Func1<? super T, ? extends R> func) {
    return create(new OnSubscribeMap<T, R>(this, func));
}

public static <T> Observable<T> create(OnSubscribe<T> f) {
    return new Observable<T>(RxJavaHooks.onCreate(f));
}
```

这里有两个小插曲，一是 map 的实现本来是利用 lift + Operator 实现的，但是后来改成了 create + OnSubscribe（RxJava #4097）；二是 lift 的实现本来是直接调用 observable 构造函数，后来改成了调用 create（RxJava #4007）。后者先发生，引入了新的 hook 机制，前者则是为了提升一点性能。

所以这里实际上是 OnSubscribeMap 干活了。

3.2, OnSubscribeMap

那我们看看 OnSubscribeMap：

```
public final class OnSubscribeMap<T, R> implements OnSubscribe<R> {

    final Observable<T> source;

    final Func1<? super T, ? extends R> transformer;

    public OnSubscribeMap(Observable<T> source,
        Func1<? super T, ? extends R> transformer) {
        this.source = source;
        this.transformer = transformer;
    }

    @Override
    public void call(final Subscriber<? super R> o) {
        MapSubscriber<T, R> parent =
            new MapSubscriber<T, R>(o, transformer); // 1
        o.add(parent); // 2
        source.unsafeSubscribe(parent); // 3
    }
}
```

它的实现很直观：

1. 利用传入的 `subscriber` 以及我们进行转换的 `Func1` 构造一个 `MapSubscriber`。
2. 把一个 `subscriber` 加入到另一个 `subscriber` 中，是为了让它们可以一起取消订阅。
3. `unsafeSubscribe` 相较于前面的 `subscribe`，可想而知就是少了一层 `SafeSubscriber` 的包装。为什么不要包装？因为我们在最后调用 `Observable#subscribe` 时进行包装，只需要包装一次即可。

转换的代码依然没有出现，它在 `MapSubscriber` 中。

3.3, MapSubscriber

```

static final class MapSubscriber<T, R> extends Subscriber<T> {

    final Subscriber<? super R> actual;

    final Func1<? super T, ? extends R> mapper;

    boolean done;

    public MapSubscriber(Subscriber<? super R> actual,
                         Func1<? super T, ? extends R> mapper) {
        this.actual = actual;
        this.mapper = mapper;
    }

    @Override
    public void onNext(T t) {
        R result;

        try {
            result = mapper.call(t);           // 1
        } catch (Throwable ex) {
            Exceptions.throwIfFatal(ex);
            unsubscribe();
            onError(OnErrorThrowable.addValueAsLastCause(ex, t));
            return;
        }

        actual.onNext(result);             // 2
    }

    // 省略 onError, onCompleted 和 setProducer
}

```

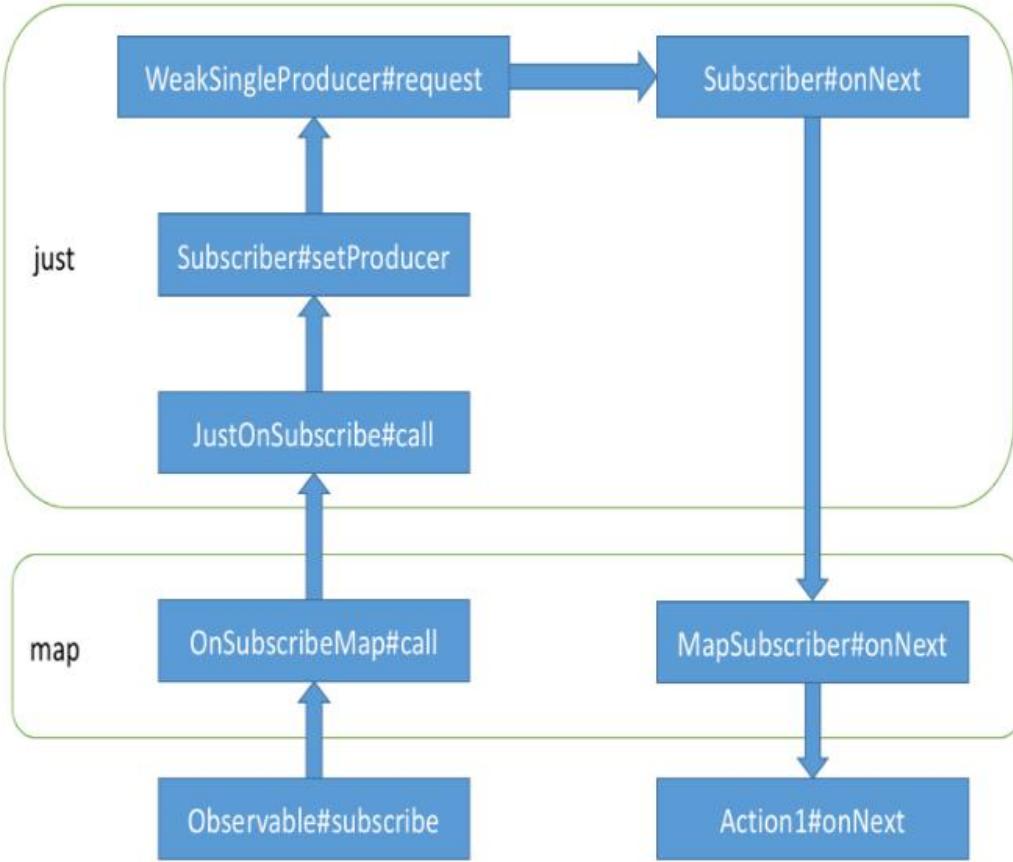
`MapSubscriber` 依然很直观：

1. 上游每新来一个数据，就用我们给的 `mapper` 进行数据转换。
2. 再把转换之后的数据发送给下游。

这里要解释一下“上游”和“下游”的概念：按照我们写的代码顺序，`just` 在 `map` 的上面，`Action1` 在 `map` 的下面，数据从 `just` 传递到 `map` 再传递到 `Action1`，所以对于 `map` 来说，`just` 就是上游，`Action1` 就是下游。数据是从上游（`Observable`）一路传递到下游（`Subscriber`）的，请求则相反，从下游传递到上游。

3.4，完整的过程

引入一个 `map` 操作符新增的内容并不多，而且由于责任分拆得好，每个部分的实现都很简单。结合第 2 部分基础的内容，这个例子的完整调用过程可以用下图表示：



上面的过程依然由 `subscribe` 触发，而且都是直接的函数调用，所以都在调用 `subscribe` 的线程执行。

4，线程调度

前面我们所有的过程都是通过函数调用完成的，都在 `subscribe` 所在线程执行。`RxJava` 进行异步非常简单，只需要使用 `subscribeOn` 和 `observeOn` 这两个操作符即可。既然它俩都是操作符，那流程上就是和 `map` 差不多的，这里我们主要关注线程调度的实现原理。

我们先看看例子：

```
Observable.just("Hello world")
    .map(String::length)
    .subscribeOn(Schedulers.computation())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(len -> {
        System.out.println("got " + len + " @ "
            + Thread.currentThread().getName());
    });
});
```

4.1, subscribeOn

我们看看 `subscribeOn` 的实现:

```
public final Observable<T> subscribeOn(Scheduler scheduler) {
    if (this instanceof ScalarSynchronousObservable) {
        return ((ScalarSynchronousObservable<T>)this)
            .scalarScheduleOn(scheduler);
    }
    return create(new OperatorSubscribeOn<T>(this, scheduler));
}
```

还记得上面的 `just` 吗? 它创建的就是 `ScalarSynchronousObservable`, 但是这个特殊情况我们先跳过, 我们看普通的情况: 通过 `create + OperatorSubscribeOn` 实现。

4.2, OperatorSubscribeOn

```
public final class OperatorSubscribeOn<T>
    implements OnSubscribe<T> {

    final Scheduler scheduler;
    final Observable<T> source;

    public OperatorSubscribeOn(Observable<T> source,
        Scheduler scheduler) {
        this.scheduler = scheduler;
        this.source = source;
    }

    @Override
    public void call(final Subscriber<? super T> subscriber) {
        final Worker inner = scheduler.createWorker();
        subscriber.add(inner); // 1

        inner.schedule(new Action0() {
            @Override
            public void call() {
                final Thread t = Thread.currentThread();

                Subscriber<T> s = new Subscriber<T>(subscriber) {
                    @Override
                    public void onNext(T t) {
                        subscriber.onNext(t); // 2
                    }
                };

                // 省略 onError 和 onCompleted

                @Override
                public void setProducer(final Producer p) {
                    subscriber.setProducer(new Producer() {
                        @Override
                        public void request(final long n) {
                            if (t == Thread.currentThread())
                                p.request(n); // 3
                            } else {
                                inner.schedule(new Action0() {
                                    @Override
                                    public void call() {
                                        p.request(n); // 4
                                    }
                                });
                            }
                        }
                    });
                }
            });
        });

        source.unsafeSubscribe(s); // 5
    }
}
```

1. `Worker` 也实现了 `Subscription`, 所以可以加入到 `Subscriber` 中, 用于集体取消订阅。
 2. 在匿名 `Subscriber` 中, 收到上游的数据后, 转发给下游。

3. `Producer#request` 被调用时，如果调用线程就是 `worker` 的线程（`t`），就直接把请求转发给上游。
4. 否则还需要进行一次调度，确保调用上游的 `request` 一定是在 `worker` 的线程。
5. 在 `worker` 线程中，把自己（匿名 `Subscriber`）和上游连接起来。

这里我们就看到，连接上游（可能会触发请求）、向上游发请求，都是在 `worker` 的线程上执行的，所以如果上游处理请求的代码没有进行异步操作，那上游的代码就是在 `subscribeOn` 指定的线程上执行的。这就解释了网上随处可见的一个结论：

`subscribeOn` 影响它上面的调用执行时所在的线程。

但如果仅仅是记住这么一句话，情况稍微一复杂，就必然蒙圈，所以一定要理解它的工作原理。

另外关于使用多次调用 `subscribeOn` 的效果，我们这里也就很清楚了，后面的 `subscribeOn` 只会改变前面的 `subscribeOn` 调度操作所在的线程，并不能改变最终被调度的代码执行的线程，但对于中途的代码执行的线程，还是会影响到的。

在上面的代码中，收到上游发来的数据之后，我们直接发给了下游，并没有进行线程切换，所以 `subscribeOn` 并不会改变数据向下游传递时的线程，这一工作由它的搭档 `observeOn` 完成。

4.3, `observeOn`

```

public final Observable<T> observeOn(Scheduler scheduler) {
    return observeOn(scheduler, RxRingBuffer.SIZE);
}

public final Observable<T> observeOn(Scheduler scheduler,
        int bufferSize) {
    return observeOn(scheduler, false, bufferSize);
}

public final Observable<T> observeOn(Scheduler scheduler,
        boolean delayError, int bufferSize) {
    if (this instanceof ScalarSynchronousObservable) {
        return ((ScalarSynchronousObservable<T>)this)
            .scalarScheduleOn(scheduler);
    }
    return lift(new OperatorObserveOn<T>(scheduler,
        delayError, bufferSize));
}

public final <R> Observable<R> lift(
        final Operator<? extends R, ? super T> operator) {
    return create(new OnSubscribeLift<T, R>(onSubscribe,
        operator));
}

```

`observeOn` 有好几个重载版本，支持指定 `buffer` 大小、是否延迟 `Error` 事件，这个 `delayError` 是从 `v1.1.1` 引入的，关于它还有一个小插曲。

之前和 [Ryan Hoo](#) 碰到过一个问题，`concat` 两个 `observable`，第一个没有 `error`，第二个有 `error`，结果居然收不到第一个里面的数据，而是直接收到了第二个的 `error`。最后发现就是没有用 `delayError` 参数。

这里我们依然关注普遍情况，即利用 `lift + operator` 实现的情况。

所以我们先看 `OnSubscribeLift`。

4.4, OnSubscribeLift

```

public final class OnSubscribeLift<T, R>
    implements OnSubscribe<R> {

    final OnSubscribe<T> parent;

    final Operator<? extends R, ? super T> operator;

    public OnSubscribeLift(OnSubscribe<T> parent,
        Operator<? extends R, ? super T> operator) {
        this.parent = parent;
        this.operator = operator;
    }

    @Override
    public void call(Subscriber<? super R> o) {
        Subscriber<? super T> st = RxJavaHooks
            .onObservableLift(operator).call(o);
        st.onStart();
        parent.call(st);
        // 省略了异常处理代码
    }
}

```

我们先对下游 subscriber 用操作符进行处理（跳过 hook），然后通知处理后的 subscriber，它将要和 observable 连接起来了，最后把它和上游连接起来。

这里并没有线程调度的逻辑，所以我们看 `OperatorObserveOn`。

4.5, `OperatorObserveOn`

```

public final class OperatorObserveOn<T> implements Operator<T, T> {
    // ...

    @Override
    public Subscriber<? super T> call(Subscriber<? super T> child) {
        if (scheduler instanceof ImmediateScheduler) {
            // avoid overhead, execute directly
            return child;
        } else if (scheduler instanceof TrampolineScheduler) {
            // avoid overhead, execute directly
            return child;
        } else {
            ObserveOnSubscriber<T> parent = new ObserveOnSubscriber<T>(
                scheduler, child, delayError, bufferSize);
            parent.init();
            return parent;
        }
    }
    // ...
}

```

作为操作符的逻辑，还是很简单的，如果 `scheduler` 是 `ImmediateScheduler/TrampolineScheduler`，就什么也不做，否则就把 `subscriber` 包装为 `ObserveOnSubscriber`，看来脏活累活都是 `ObserveOnSubscriber` 干的了。

4.6, ObserveOnSubscriber

`ObserveOnSubscriber` 除了负责把向下游发送数据的操作调度到指定的线程，还负责 `backpressure` 支持，这导致它的实现比较复杂，所以这里只展示和分析最简单的调度功能。完整代码的分析大家可以自行阅读源码，其中还会涉及到串行访问相关的内容，建议大家先看看下面几篇 Advanced RxJava 译文：

- Operator 并发原语：串行访问（serialized access）（一），`emitter-loop`
- Operator 并发原语：串行访问（serialized access）（二），`queue-drain`
- `SubscribeOn` 和 `ObserveOn`

我们来看看简化版的调度实现（摘自上面的译文）：

```
Observable.create(subscriber -> {
    Worker worker = scheduler.createWorker();
    subscriber.add(worker);

    source.unsafeSubscribe(new Subscriber<T>(subscriber) {
        @Override
        public void onNext(T t) {
            worker.schedule(() -> subscriber.onNext(t));
        }

        @Override
        public void onError(Throwable e) {
            worker.schedule(() -> subscriber.onError(e));
        }

        @Override
        public void onCompleted() {
            worker.schedule(() -> subscriber.onCompleted());
        }
    });
});
```

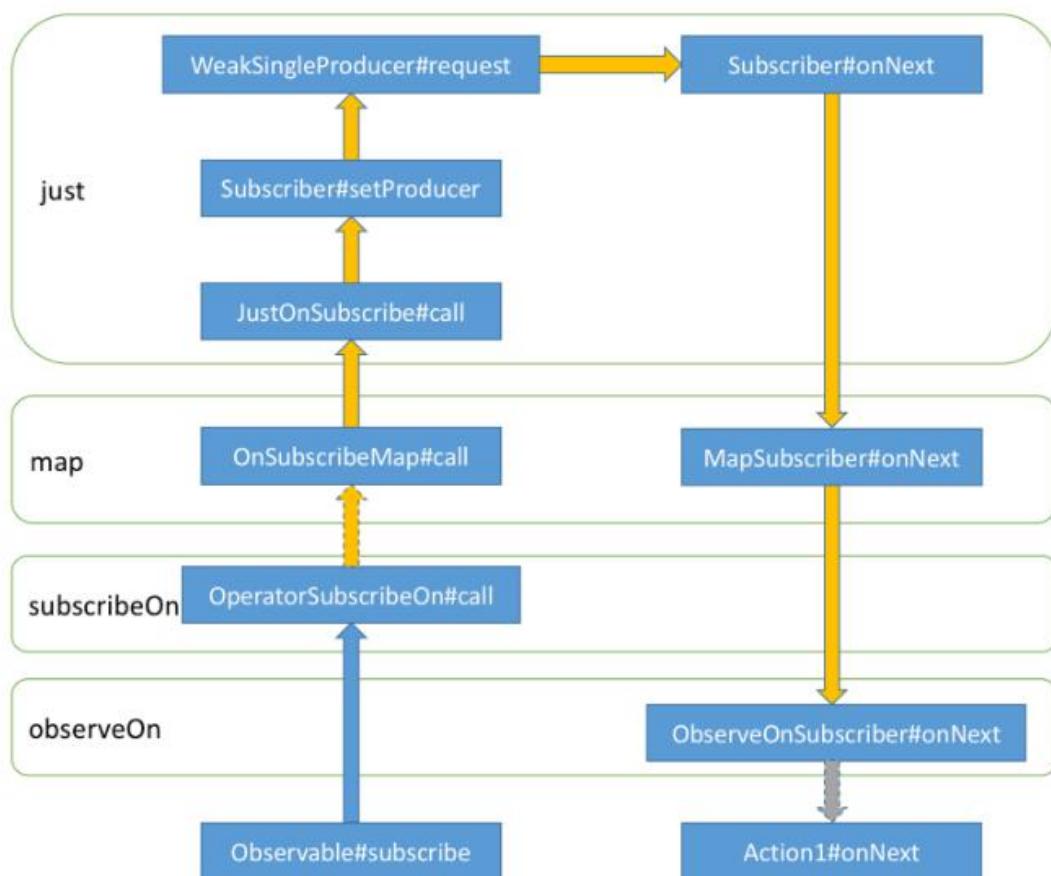
这里 `observeOn` 调度了每个单独的 `subscriber.onXXX()` 调用，使得数据向下游传递的时候可以切换到指定的线程。这也同样解释了网上随处可见的另一个结论：

`observeOn` 影响它下面的调用执行时所在的线程。

这时我们也就清楚了多次调用 `observeOn` 的效果，每次调用都会改变数据向下传递时所在的线程。

4.7, 完整的过程

最后我们看一下整个例子的调用过程：



5, backpressure

其实在前面我们讲 `just` 时，就已经讲过了 backpressure：

在 RxJava 1.x 中，数据都是从 `observable` push 到 `subscriber` 的，但要是 `observable` 发得太快，`subscriber` 处理不过来，该怎么办？一种办法是，把数据保存起来，但这显然可能导致内存耗尽；另一种办法是，多余的数据来了之后就丢掉，至于丢掉和保留的策略可以按需制定；还有一种办法就是让 `subscriber` 向 `observable` 主动请求数据，`subscriber` 不请求，`observable` 就

不发出数据。它俩相互协调，避免出现过多的数据，而协调的桥梁，就是 *producer*。

为了章节完整性，这里保留一节，更多细节的内容，可以查看 [stackoverflow 上的这篇 wiki](#)，由 Advanced RxJava 系列博客原文作者编写，非常不错。不过其内容总结来说，也就是上面这一段 :)

6, hook

在上面的内容中，我们多次遇见了 `hook`，为了简化逻辑，也多次跳过了 `hook`，这里我们就看看 `hook` 有什么用，工作原理是什么。

利用 `hook` 我们可以站在“上帝视角”，多种重要的节点上，都有 `hook`。例如创建 `Observable` (`create`) 时，有 `onCreate`，我们可以进行任意想要的操作，记录、修饰、甚至抛出异常；以及和 `scheduler` 相关的内容，获取 `scheduler` 时，我们都可以进行想要的操作，例如让 `Scheduler.io()` 返回立即执行的 `scheduler`。

这些内容让我们可以执行高度自定义的操作，其中就包括便于测试。

其实 `hook` 的原理并不复杂，在关心的节点（`hook point`）插桩，让我们可以操控（`manipulate`）程序在这些节点的行为，至于操控的策略，有一系列函数进行设置、以及清理。

目前和 `hook` 相关的内容主要在 `RxJavaPlugins` 和 `RxJavaHooks` 这两个类中，后者在 `v1.1.7` 引入，功能更加强大，使用更加方便。

7, 测试

RxJava 项目本身测试覆盖率高达 84%，为了便于我们对使用 RxJava 的代码进行测试，它还专门提供了 `TestSubscriber`，我们可以用它来获取我们的事件流中的事件、进行验证、进行等待，使用起来非常简便。

Rxjava 观察者模式原理解析

传统观察者模式

观察者模式面向的需求是：A 对象（观察者）对 B 对象（被观察者）的某种变化高度敏感，需要在 B 变化的一瞬间做出反应。举个例子，新闻里喜闻乐见的警察抓小偷，警察需要在小偷伸手作案的时候实施抓捕。在这个例子里，警察是观察者，小偷是被观察者，警察需要时刻盯着小偷的一举一动，才能保证不会漏过任何瞬间。程序的观察者模式和这种真正的『观察』略有不同，观察者不需要时刻盯着被观察者（例如 A 不需要每过 2ms 就检查一次 B 的状态），而是采用注册(Register)或者称为订阅(Subscribe)的方式，告诉被观察者：我需要你的某某状态，你要在它变化的时候通知我。Android 开发中一个比较典型的例子是点击监听器 OnClickListener 。对设置 OnClickListener 来说， View 是被观察者， OnClickListener 是观察者，二者通过 setOnClickListener() 方法达成订阅关系。订阅之后用户点击按钮的瞬间，Android Framework 就会将点击事件发送给已经注册的 OnClickListener 。采取这样被动的观察方式，既省去了反复检索状态的资源消耗，也能够得到最高的反馈速度。当然，这也得益于我们可以随意定制自己程序中的观察者和被观察者，而警察叔叔明显无法要求小偷『你在作案的时候务必通知我』。

OnClickListener 的模式大致如下图：



如图所示，通过 setOnClickListener() 方法， Button 持有 OnClickListener 的引用（这一过程没有在图上画出）；当用户点击时， Button 自动调用 OnClickListener 的 onClick() 方法。另外，如果把这张图中的概念抽象出来（Button -> 被观察者、OnClickListener -> 观察者、setOnClickListener() -> 订阅， onClick() -> 事件），就由专用的观察者模式（例如只用于监听控件点击）转变成了通用的观察者模式。如下图：



而 RxJava 作为一个工具库，使用的就是通用形式的观察者模式。

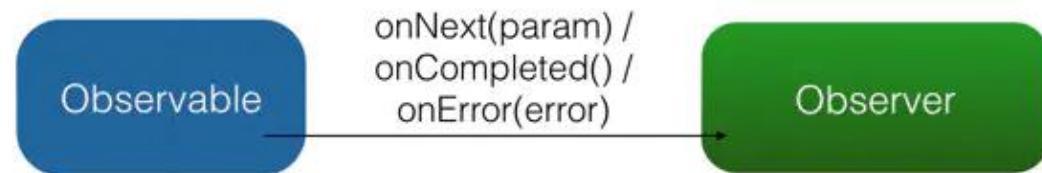
RxJava 中观察者模式

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 。

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。**并且只要 `onCompleted()` 和 `onError()` 中有一个调用了，都会中止 `onNext()` 的调用。**

RxJava 的观察者模式大致如下图：



基本实现

基于以上观点， RxJava 的基本实现主要有三点：

创建 `Observer`

`Observer` 即观察者，它决定事件触发的时候将有怎样的行为。 RxJava 中的 `Observer` 接口的实现方式：

[复制代码](#)

```
Observer<String> observer = new Observer<String>() {
    @Override
    public void onNext(String s) {
        Log.d(tag, "Item: " + s);
    }

    @Override
    public void onCompleted() {
        Log.d(tag, "Completed!");
    }

    @Override
    public void onError(Throwable e) {
        Log.d(tag, "Error!");
    }
};
```

除了 `Observer` 接口之外，RxJava 还内置了一个实现了 `Observer` 的抽象类：`Subscriber`。`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：

```
Subscriber<String> subscriber = new Subscriber<String>() {
    @Override
    public void onNext(String s) {
        Log.d(tag, "Item: " + s);
    }

    @Override
    public void onCompleted() {
        Log.d(tag, "Completed!");
    }

    @Override
    public void onError(Throwable e) {
        Log.d(tag, "Error!");
    }
};
```

不仅基本使用方式一样，实质上，在 RxJava 的 `subscribe` 过程中，`Observer` 也总是会先被转换成一个 `Subscriber` 再使用。

// Observable.java 源码

```
public final Subscription subscribe(final Observer<? super T> observer) {
    if (observer instanceof Subscriber) { // 如果是 Subscriber 的子类，直接转化
        return ((Subscriber<? super T>)observer).onSubscribe(this);
    }
}
```

```

    if (observer == null) {
        throw new NullPointerException("observer is null");
    }

    return subscribe(new ObserverSubscriber<T>(observer));
}
// ObserverSubscriber.java

public final class ObserverSubscriber<T> extends Subscriber<T> {
    ...
}

```

通过源码可以看到，传入的 `Observer` 最终还是会转化为 `Subscriber` 来使用。

所以如果你只想使用基本功能，选择 `Observer` 和 `Subscriber` 是完全一样的。它们的区别对于使用者来说主要有两点：

- `onStart()`: 这是 `Subscriber` 增加的方法。它会在 `subscribe` 刚开始，而事件还未发送之前被调用，可以用于做一些准备工作，例如数据的清零或重置。这是一个可选方法，默认情况下它的实现为空。需要注意的是，如果对准备工作的线程有要求（例如弹出一个显示进度的对话框，这必须在主线程执行），`onStart()` 就不适用了，因为它总是在 `subscribe` 所发生的线程被调用，而不能指定线程。要在指定的线程来做准备工作，可以使用 `doOnSubscribe()` 方法。

// Subscriber.java

```

public void onStart() {
    // do nothing by default
}

```

`unsubscribe()`: 这是 `Subscriber` 所实现的另一个接口 `Subscription` 的方法，用于取消订阅。在这个方法被调用后，`Subscriber` 将不再接收事件。一般在这个方法调用前，可以使用 `isUnsubscribed()` 先判断一下状态。

`unsubscribe()` 这个方法很重要，因为在 `subscribe()` 之后，`Observable` 会持有 `Subscriber` 的引用，这个引用如果不能及时被释放，将有内存泄露的风险。所以最好保持一个原则：要在不再使用的时候尽快在合适的地方（例如 `onPause()`、`onStop()` 等方法中）调用 `unsubscribe()` 来解除引用关系，以避免内存泄露的发生。

// Subscriber.java

```

@Override
public final void unsubscribe() {
    subscriptions.unsubscribe();
}

```

```
@Override  
public final boolean isUnsubscribed() {  
    return subscriptions.isUnsubscribed();  
}
```

创建 Observable

Observable 即被观察者，它决定什么时候触发事件以及触发怎样的事件。例如 `create()` 方法

```
Observable observable = Observable.create(new Observable.OnSubscribe<String>() {  
    @Override  
    public void call(Subscriber<? super String> subscriber) {  
        subscriber.onNext("Hello");  
        subscriber.onNext("Hi");  
        subscriber.onNext("Aloha");  
        subscriber.onCompleted();  
    }  
});
```

复制代码

可以看到，这里传入了一个 `OnSubscribe` 对象作为参数。`OnSubscribe` 会被存储在返回的 `Observable` 对象中，它的作用相当于一个计划表，当 `Observable` 被订阅的时候，`OnSubscribe` 的 `call()` 方法会自动被调用，事件序列就会依照设定依次触发（对于上面的代码，就是观察者 `Subscriber` 将会被调用三次 `onNext()` 和一次 `onCompleted()`）。这样，由被观察者调用了观察者的回调方法，就实现了由被观察者向观察者的事件传递，即观察者模式。

`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()`, `from()`

订阅 Subscribe

创建了 `Observable` 和 `Observer` 之后，再用 `subscribe()` 方法将它们联结起来，整条链子就可以工作了。代码形式很简单：

```
observable.subscribe(observer);
```

// 或者：

```
observable.subscribe(subscriber);
```

有人可能会注意到，`subscribe()` 这个方法有点怪：它看起来是『`observable` 订阅了 `observer / subscriber`』而不是『`observer / subscriber` 订阅了 `observable`』，这看起来就像『杂志订阅了读者』一样颠倒了对象关系。这让人读起来有点别扭，不过如果把 API 设计成 `observer.subscribe(observable) / subscriber.subscribe(observable)`，虽然更加符合思维逻辑，但对流式 API 的设计就造成影响了，比较起来明显是得不偿失的。

整个过程中对象间的关系如下图：



源码层解析

基础原理

// 例子

```
Observable.create(new Observable.OnSubscribe<String>() {
    @Override
    public void call(Subscriber<? super String> subscriber) {
        subscriber.onNext("Hello");
        subscriber.onNext("Hi");
        subscriber.onNext("Aloha");
        subscriber.onCompleted();
    }
}).subscribe(new Subscriber<String>() {
    @Override
    public void onCompleted() {
        System.out.println("onCompleted");
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onNext(String s) {
        System.out.println("value: " + s);
    }
});
```

log 信息

```
value: Hello  
value: Hi  
value: Aloha  
onCompleted
```

复制代码

看到上面代码，可能会有人跟我一样不明白，`create()` 中的 `OnSubscribe` 中 `call()` 的 `Subscriber` 是怎么样最终就变成了 `subscribe()` 中的 `Subscriber`。下面来一下 `Observable.subscribe(Subscriber)` 的内部实现是这样的（仅核心代码）：

```
// 注意：这不是 subscribe() 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。  
  
static <T> Subscription subscribe(Subscriber<? super T> subscriber, Observable<T> observable) {  
  
    ...  
  
    // 可以用于做一些准备工作，例如数据的清零或重置，默认情况下它的实现为空  
    subscriber.onStart();  
  
    if (!(subscriber instanceof SafeSubscriber)) {  
        // 强制转化为 SafeSubscriber 是为了保证 onCompleted 或 onError 调用的时候会中止 onNext 的调用  
        subscriber = new SafeSubscriber<T>(subscriber);  
    }  
  
    ...  
    // // onObservableStart() 默认返回的就是 observable.onSubscribe  
    RxJavaHooks.onObservableStart(observable, observable.onSubscribe).call(subscriber);  
  
    // onObservableReturn() 默认也是返回 subscriber  
    return RxJavaHooks.onObservableReturn(subscriber);  
  
    ...  
}
```

通过源码可以看到，`subscribe()` 实际就做了 4 件事情

- 调用 `Subscriber.onStart()`。这个方法在前面已经介绍过，是一个可选的准备方法。
- 将传入的 `Subscriber` 转化为 `SafeSubscriber`，为了保证 `onCompleted` 或 `onError` 调用的时候会中止 `onNext` 的调用。

```
// 注意：这不是 SafeSubscriber 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。
```

```
public class SafeSubscriber<T> extends Subscriber<T> {

    private final Subscriber<? super T> actual;

    boolean done; // 通过改标志来保证 onCompleted 或 onError 调用的时候会中
    止 onNext 的调用

    public SafeSubscriber(Subscriber<? super T> actual) {
        super(actual);
        this.actual = actual;
    }

    @Override
    public void onCompleted() {
        if (!done) {
            done = true;

            ...

            actual.onCompleted();

            ...

            unsubscribe(); // 取消订阅，结束事务
        }
    }

    @Override
    public void onError(Throwable e) {

        ...

        if (!done) {
            done = true;
            _onError(e);
        }
    }

    @Override
    public void onNext(T t) {

        if (!done) { // done 为 true 时，中止传递
            actual.onNext(t);
        }
    }
}
```

```

    }

    @SuppressWarnings("deprecation")
    protected void _onError(Throwable e) {
        ...
        actual.onError(e);
        ...
        unsubscribe();
        ...
    }
}

```

通过代码可以看出来，通过 `SafeSubscriber` 中的布尔变量 `done` 来做标记保证上文提到的 `onCompleted()` 和 `onError()` 二者的互斥性，即在队列中调用了其中一个，就不应该再调用另一个。**并且只要 `onCompleted()` 和 `onError()` 中有一个调用了，都会中止 `onNext()` 的调用。**

- 调用 `Observable` 中的 `OnSubscribe.call(Subscriber)`。在这里，事件发送的逻辑开始运行。从这也可以看出，在 RxJava 中，`Observable` 并不是在创建的时候就立即开始发送事件，而是在它被订阅的时候，即当 `subscribe()` 方法执行的时候。
- 将传入的 `Subscriber` 作为 `Subscription` 返回。这是为了方便 `unsubscribe()`。

以上就是 RxJava 最基本的一个通过观察者模式，来响应事件的原理。下面来看看 RxJava 中一些基本操作符的实现原理又是怎样的。

进阶

```

Observable.interval(1, TimeUnit.SECONDS)
    .map(new Func1<Long, Long>() {
        @Override
        public Long call(Long aLong) {
            return aLong * 5;
        }
    })
    .subscribe(new Action1<Long>() {
        @Override
        public void call(Long aLong) {
            System.out.println("value: " + aLong);
        }
    });

```

log 信息

[复制代码](#)

```
value: 0  
value: 5  
value: 10  
...
```

第一次看到该例子时，就喜欢上了 RxJava，这种链式函数的交互模式真的很简洁，终于可以从回调地狱里逃出来了。喜欢的同时不免也会想 RxJava 是如何实现的。这种链式的函数流可以算是建造者模式的一种变形，只不过省去了中间 `Builder` 而直接返回当前对象来实现。更让我兴奋的是内部这些操作符的实现原理。

上文也已经说过了在 RxJava 中，`Observable` 并不是在创建的时候就立即开始发送事件，而是在它被订阅的时候，即当 `subscribe()` 方法执行的时候。所以对于上面一段的代码我们要从 `subscribe()` 往前屡，首先看一下 `map()` 这个函数的内部实现。

```
public final <R> Observable<R> map(Func1<? super T, ? extends R> func) {  
    // 新建了一个 Observable 并使用新的 OnSubscribeMap 来封装传入的数据  
    return unsafeCreate(new OnSubscribeMap<T, R>(this, func));  
}
```

不用说大家也猜到了 `OnSubscribeMap` 是 `OnSubscribe` 的子类

// 注意：这不是 `OnSubscribeMap` 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。

```
public final class OnSubscribeMap<T, R> implements OnSubscribe<R> {  
  
    final Observable<T> source;  
  
    final Func1<? super T, ? extends R> transformer;  
  
    public OnSubscribeMap(Observable<T> source, Func1<? super T, ? extends R> transformer) {  
        this.source = source; // 列子中经过 Observable.interval() 函数生成的 Observable  
        this.transformer = transformer;  
    }  
  
    // 传入的 o 就是例子中 `subscribe()` 出入的 Subscriber  
    // 具体结合 Observable.subscribe() 源码来理解  
    @Override  
    public void call(final Subscriber<? super R> o) {  
  
        // 对传入的 Subscriber 进行再次封装成 MapSubscriber  
        // 具体 Observable.map() 的逻辑是在 MapSubscriber 中
```

```
MapSubscriber<T, R> parent = new MapSubscriber<T, R>(o, transformer);
o.add(parent); // 加入到 SubscriptionList 中，为之后取消订阅

// Observable.interval() 返回的 Observable 进行订阅(关键点)
source.unsafeSubscribe(parent);
}

...

}

可以看到 call() 方法的逻辑很简单，只是将例子中 observable.subscribe() 传入的 Subscriber 进行封装后，再将上流传入的 Observable 进行订阅
// 注意：这不是 MapSubscriber 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。
static final class MapSubscriber<T, R> extends Subscriber<T> {

    final Subscriber<? super R> actual;

    final Func1<? super T, ? extends R> mapper;

    public MapSubscriber(Subscriber<? super R> actual, Func1<? super T, ? extends R> mapper) {
        this.actual = actual; // Observable.subscribe() 传入的 Subscriber
        this.mapper = mapper;
    }

    @Override
    public void onNext(T t) {
        R result;

        ...

        result = mapper.call(t); // 数据进行了变换

        ...

        actual.onNext(result); // 往下流传
    }

    ...
}
```

通过以上就完成了 `map()` 对数据的变换，这里最终的就是理解 `OnSubscribeMap` 的 `call()` 中 `source.unsafeSubscribe(parent);` `source` 指的是例子中 `Observable.interval()` 生成的对象。

再来看一下 RxJava 中对 `Observable.interval()` 的实现

```
public static Observable<Long> interval(long initialDelay, long period, TimeUnit unit, Scheduler scheduler) {
```

```
    return unsafeCreate(new OnSubscribeTimerPeriodically(initialDelay, period, unit, scheduler));
```

```
}
```

可以看出 `interval()` 和 `map()` 一样都是通过生成新的 `Observable` 并向 `Observable` 中传入与之对应的 `OnSubscribe` 的子类来完成具体操作。

// 注意：这不是 `OnSubscribeTimerPeriodically` 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。

```
public final class OnSubscribeTimerPeriodically implements OnSubscribe<Long> {
```

```
    final long initialDelay;
```

```
    final long period;
```

```
    final TimeUnit unit;
```

```
    final Scheduler scheduler;
```

```
    public OnSubscribeTimerPeriodically(long initialDelay, long period, TimeUnit unit, Scheduler scheduler) {
```

```
        this.initialDelay = initialDelay;
```

```
        this.period = period;
```

```
        this.unit = unit;
```

```
        this.scheduler = scheduler;
```

```
}
```

```
    // 传入的 Subscriber 为上文提到的 OnSubscribeMap.call() 方法中 source.unsafeSubscribe(parent);
```

```
    @Override
```

```
    public void call(final Subscriber<? super Long> child) {
```

```
        final Worker worker = scheduler.createWorker();
```

```
        child.add(worker);
```

```
        worker.schedulePeriodically(new Action0() {
```

```
            long counter;
```

```
            @Override
```

```
            public void call() {
```

```
            ...
```

```
        child.onNext(counter++);

        ...
    }

}, initialDelay, period, unit);
}
}
```

以上就是 RxJava 整体的逻辑结构，可以看到 RxJava 将观察者模式发挥的淋漓尽致。整体逻辑的处理有点像递归函数的原理。而 `map()` 则像一种代理机制，通过事件拦截和处理实现事件序列的变换

Rxjava 订阅流程，线程切换，源码分析 系列

[RxJava2.x 源码解析（一）基本订阅流程](#)

[RxJava2.x 源码解析（二）线程切换](#)

[RxJava2.x 源码解析（三）zip 源码分析](#)

二十一、OKHTTP 和 Retrofit

OKHTTP 完整解析

一、概述

最近在群里听到各种讨论 okhttp 的话题，可见 okhttp 的口碑相当好了。再加上 Google 貌似在 6.0 版本里面删除了 HttpClient 相关 API，对于这个行为不做评价。为了更好的在应对网络访问，学习下 okhttp 还是蛮必要的，本篇博客首先介绍 okhttp 的简单使用，主要包含：

一般的 get 请求

一般的 post 请求

基于 Http 的文件上传

文件下载

加载图片

支持请求回调，直接返回对象、对象集合

支持 session 的保持

最后会对上述几个功能进行封装，完整的封装类的地址见：

<https://github.com/hongyangAndroid/okhttp-utils>

使用前，对于 Android Studio 的用户，可以选择添加：

compile 'com.squareup.okhttp:okhttp:2.4.0'

或者 Eclipse 的用户，可以下载最新的 jar [okhttp he latest JAR](#)，添加依赖就可以用了。

注意：okhttp 内部依赖 okio，别忘了同时导入 okio：

gradle: `com.squareup.okhttp:okhttp:2.4.0`

最新的 jar 地址：[okio the latest JAR](#)

二、使用教程

(一) Http Get

对了网络加载库，那么最常见的肯定就是 http get 请求了，比如获取一个网页的内容。

```
1 //创建okHttpClient对象
2 OkHttpClient mOkHttpClient = new OkHttpClient();
3 //创建一个Request
4 final Request request = new Request.Builder()
5         .url("https://github.com/hongyangAndroid")
6         .build();
7 //new call
8 Call call = mOkHttpClient.newCall(request);
9 //请求加入调度
10 call.enqueue(new Callback()
11 {
12     @Override
13     public void onFailure(Request request, IOException e)
14     {
15     }
16
17     @Override
18     public void onResponse(final Response response) throws IOException
19     {
20         //String htmlStr = response.body().string();
21     }
22 });
23
```

复制

1.

以上就是发送一个 get 请求的步骤，首先构造一个 Request 对象，参数最起码有个 url ,当然你可以通过 Request.Builder 设置更多的参数比如： 、 等。

2.

3.

然后通过 request 的对象去构造得到一个 Call 对象，类似于将你的请求封装成了任务，既然是任务，就会有 和 等方法。

4.

5.

最后，我们希望以异步的方式去执行请求，所以我们调用的是 `call.enqueue`，将 `call` 加入调度队列，然后等待任务执行完成，我们在 `Callback` 中即可得到结果。

6.

看到这，你会发现，整体的写法还是比较长的，所以封装肯定是要做的，不然每个请求这么写，得累死。

ok，需要注意几点：

`onResponse` 回调的参数是 `response`，一般情况下，比如我们希望获得返回的字符串，可以通过 `getResponseBody` 获取；如果希望获得返回的二进制字节数组，则调用 `getRawBody`；如果你想拿到返回的 `InputStream`，则调用 `getInputStream`。

看到这，你可能会奇怪，竟然还能拿到返回的 `InputStream`，看到这个最起码能意识到一点，这里支持大文件下载，有 `InputStream` 我们就可以通过 IO 的方式写文件。不过也说明一个问题，这个 `getInputStream` 执行的线程并不是 UI 线程。的确是的，如果你希望操作控件，还是需要使用 `handler` 等，例如：

```
1  @Override  
2  public void onResponse(final Response response) throws IOException  
3  {  
4      final String res = response.body().string();  
5      runOnUiThread(new Runnable()  
6      {  
7          @Override  
8          public void run()  
9          {  
10             mTv.setText(res);  
11         }  
12     });  
13 }  
14 }
```

复制

- 我们这里是异步的方式去执行，当然也支持阻塞的方式，上面我们也说了 Call 有一个 方法，你也可以直接调用 通过返回一个 。

(二) Http Post 携带参数

看来上面的简单的 get 请求，基本上整个的用法也就掌握了，比如 post 携带参数，也仅仅是 Request 的构造的不同。

```
1 Request request = buildMultipartFormRequest(  
2         url, new File[]{file}, new String[]{fileKey}, null);  
3 FormEncodingBuilder builder = new FormEncodingBuilder();  
4 builder.add("username", "张鸿洋");  
5  
6 Request request = new Request.Builder()  
7             .url(url)  
8             .post(builder.build())  
9             .build();  
10    mOkHttpClient.newCall(request).enqueue(new Callback(){});
```

复制

- 我们这里是异步的方式去执行，当然也支持阻塞的方式，上面我们也说了
Call 有一个 `execute` 方法，你也可以直接调用 `execute` 通过返回一个
`Response`。

(二) Http Post 携带参数

看来上面的简单的 get 请求，基本上整个的用法也就掌握了，比如 post 携带参数，也仅仅是 Request 的构造的不同。

```
File file = new File(Environment.getExternalStorageDirectory(), "balabala.mp4");

RequestBody fileBody = RequestBody.create(MediaType.parse("application/octet-stream"), file);

RequestBody requestBody = new MultipartBuilder()
    .type(MultipartBuilder.FORM)
    .addPart(Headers.of(
        "Content-Disposition",
        "form-data; name=\"username\"");
        RequestBody.create(null, "张鸿洋")))
    .addPart(Headers.of(
        "Content-Disposition",
        "form-data; name=\"mFile\"";
        filename="wjd.mp4"), fileBody)
    .build();

Request request = new Request.Builder()
    .url("http://192.168.1.103:8080/okHttpServer/fileUpload")
    .post(requestBody)
    .build();

Call call = mOkHttpClient.newCall(request);
call.enqueue(new Callback()
{
    //...
});
```

上述代码向服务器传递了一个键值对 `:张鸿洋` 和一个文件。我们通过 `MultipartBuilder` 的 `addPart` 方法可以添加键值对或者文件。

其实类似于我们拼接模拟浏览器行为的方式，如果你对这块不了解，可以参考：

[从原理角度解析 Android \(Java \) http 文件上传](#)

ok，对于我们最开始的目录还剩下图片下载，文件下载；这两个一个是通过回调的 `Response` 拿到 `byte[]` 然后 `decode` 成图片；文件下载，就是拿到 `InputStream` 做写文件操作，我们这里就不赘述了。

关于用法，也可以参考[泡网 OkHttp 使用教程](#)

接下来我们主要看如何封装上述的代码。

三、封装

由于按照上述的代码，写多个请求肯定包含大量的重复代码，所以我希望封装后的代码调用是这样的：

(一) 使用

1.

一般的 get 请求

```
OkHttpClientManager.getAsyn("https://www.baidu.com",  
                           new  
                           OkHttpClientManager.ResultCallback<String>()  
                           {
```

```
    @Override  
  
    public void onError(Request request, Exception e)  
  
    {  
  
        e.printStackTrace();  
  
    }
```

```
    @Override  
  
    public void onResponse(String u)  
  
    {  
  
        mTv.setText(u);//注意这里是 UI 线程  
  
    }  
  
});
```

对于一般的请求，我们希望给 url，然后 CallBack 里面直接操作控件。

文件上传且携带参数

我们希望提供一个方法，传入 url,params,file,callback 即可。

```
OkHttpClientManager.postAsyn("http://192.168.1.103:8080/okHttpServ  
er/fileUpload",//
```

```
new OkHttpClientManager.ResultCallback<String>()

{

    @Override

    public void onError(Request request, IOException e)

    {

        e.printStackTrace();

    }

    @Override

    public void onResponse(String result)

    {

    }

},//file,//

"mFile",//
```

```
new OkHttpClientManager.Param[]{
    new OkHttpClientManager.Param("username", "zhy"),
    new OkHttpClientManager.Param("password", "123")
};
```

键值对没什么说的，参数 3 为 file，参数 4 为 file 对应的 name，这个 name 不是文件的名字；
对应于 http 中的

```
<          >
```

对应的是 name 后面的值，即 mFile.

文件下载

对于文件下载，提供 url，目标 dir，callback 即可。

```
OkHttpClientManager.downloadAsyn(
    "http://192.168.1.103:8080/okHttpServer/files/messenger_01.png",
    Environment.getExternalStorageDirectory().getAbsolutePath(),
    new OkHttpClientManager.ResultCallback<String>()
{
```

```
    @Override  
  
    public void onError(Request request, IOException e)  
  
    {  
  
    }  
  
    @Override  
  
    public void onResponse(String response)  
  
    {  
  
        //文件下载成功，这里回调的 reponse 为文件的 absolutePath  
  
    }  
  
});
```

展示图片

展示图片，我们希望提供一个 url 和一个 imageview，如果下载成功，直接帮我们设置上即可。

```
1 OkHttpClientManager.displayImage(mImageView,  
2     "http://images.csdn.net/20150817/1.jpg");
```

复制

1.

内部会自动根据 imageview 的大小自动对图片进行合适的压缩。虽然，这里可能不适合一次性加载大量图片的场景，但是对于 app 中偶尔有几个图片的加载，还是可用的。

2.

四、整合 Gson

很多人提出项目中使用时，服务端返回的是 Json 字符串，希望客户端回调可以直接受到对象，于是整合进入了 Gson，完善该功能。

(一) 直接回调对象

例如现在有个 User 实体类：

```
package com.zhy.utils.http.okhttp;
```

```
public class User {
```

```
    public String username ;
```

```
    public String password ;
```

```
public User() {  
  
    // TODO Auto-generated constructor stub  
  
}  
  
  
  
  
public User(String username, String password) {  
  
    this.username = username;  
  
    this.password = password;  
  
}  
  
  
  
  
  
@Override  
  
public String toString()  
  
{  
  
    return "User{" +  
  
           "username='" + username + '\'' +  
  
           ", password='" + password + '\'' +  
  
           '}';
```

```
    }  
  
}
```

服务端返回：

```
1 {"username":"zhy","password":"123"}
```

复制

客户端可以如下方式调用：

```
1 OkHttpClientManager.getAsyn("http://192.168.56.1:8080/okHttpServer/user GetUser",  
2 new OkHttpClientManager.ResultCallback<User>()  
3 {  
4     @Override  
5     public void onError(Request request, Exception e)  
6     {  
7         e.printStackTrace();  
8     }  
9  
10    @Override  
11    public void onResponse(User user)  
12    {  
13        mTv.setText(u.toString());//UI线程  
14    }  
15});
```

我们传入泛型 User，在 onResponse 里面直接回调 User 对象。

这里特别要注意的事，如果在 **字符串->实体对象** 过程中发生错误，程序不会崩

溃，**方法会被回调**。

注意：这里做了少许的更新，接口命名从 **get User** 修改为 **GetUser**。接
口中的 **onResponse** 方法修改为 **onResponse**。

(二) 回调对象集合

依然是上述的 User 类，服务端返回

```
1 [{"username":"zhy","password":"123"}, {"username":"lmj","password":"12345"}]
```

则客户端可以如下调用：

```
1 OkHttpClientManager.getAsyn("http://192.168.56.1:8080/okHttpServer/user!getUsers" 复制
2 new OkHttpClientManager.ResultCallback<List<User>>()
3 {
4     @Override
5     public void onError(Request request, Exception e)
6     {
7         e.printStackTrace();
8     }
9     @Override
10    public void onResponse(List<User> us)
11    {
12        Log.e("TAG", us.size() + "");
13        mTv.setText(us.get(1).toString());
14    }
15});
```

唯一的区别，就是泛型变为 `< >`，`ok`，如果发现 bug 或者有任何意见欢迎留言。

源码

`ok`，基本介绍完了，对于封装的代码其实也很简单，我就直接贴出来了，因为也没什么好介绍的，如果你看完上面的用法，肯定可以看懂：

```
package com.zhy.utils.http.okhttp;

import android.graphics.Bitmap;

import android.graphics.BitmapFactory;
```

```
import android.os.Handler;  
  
import android.os.Looper;  
  
import android.widget.ImageView;  
  
  
  
  
import com.google.gson.Gson;  
  
import com.google.gson.internal.$Gson$Types;  
  
import com.squareup.okhttp.Call;  
  
import com.squareup.okhttp.Callback;  
  
import com.squareup.okhttp.FormEncodingBuilder;  
  
import com.squareup.okhttp.Headers;  
  
import com.squareup.okhttp.MediaType;  
  
import com.squareup.okhttp.MultipartBuilder;  
  
import com.squareup.okhttp.OkHttpClient;  
  
import com.squareup.okhttp.Request;  
  
import com.squareup.okhttp.RequestBody;  
  
import com.squareup.okhttp.Response;
```

```
import java.io.File;  
  
import java.io.FileOutputStream;  
  
import java.io.IOException;  
  
import java.io.InputStream;  
  
import java.lang.reflect.ParameterizedType;  
  
import java.lang.reflect.Type;  
  
import java.net.CookieManager;  
  
import java.net.CookiePolicy;  
  
import java.net.FileNameMap;  
  
import java.net.URLConnection;  
  
import java.util.HashMap;  
  
import java.util.Map;  
  
import java.util.Set;  
  
/**
```

* Created by zhy on 15/8/17.

*/

```
public class OkHttpClientManager
```

```
{
```

```
    private static OkHttpClientManager mInstance;
```

```
    private OkHttpClient mOkHttpClient;
```

```
    private Handler mDelivery;
```

```
    private Gson mGson;
```

```
    private static final String TAG = "OkHttpClientManager";
```

```
    private OkHttpClientManager()
```

```
{
```

```
        mOkHttpClient = new OkHttpClient();
```

```
        //cookie enabled
```

```
        mOkHttpClient.setCookieHandler(new CookieManager(null,  
CookiePolicy.ACCEPT_ORIGINAL_SERVER));  
  
        mDelivery = new Handler(Looper.getMainLooper());  
  
        mGson = new Gson();  
  
    }  
  
}
```

```
public static OkHttpClientManager getInstance()  
  
{  
  
    if (mInstance == null)  
  
    {  
  
        synchronized (OkHttpClientManager.class)  
  
        {  
  
            if (mInstance == null)  
  
            {  
  
                mInstance = new OkHttpClientManager();  
  
            }  
  
        }  
  
    }  
  
}
```

```
    }

}

return mInstance;

}

/**  

 * 同步的 Get 请求  

 *  

 * @param url  

 * @return Response  

 */

private Response _getAsyn(String url) throws IOException

{
    final Request request = new Request.Builder()  

        .url(url)  

        .build();
}
```

```
    Call call = mOkHttpClient.newCall(request);

    Response execute = call.execute();

    return execute;

}

/**

 * 同步的 Get 请求

 *

 * @param url

 *

 * @return 字符串

 */

private String _getAsString(String url) throws IOException

{

    Response execute = _getAsyn(url);

    return execute.body().string();

}
```

```
/**
```

```
* 异步的 get 请求
```

```
*
```

```
* @param url
```

```
* @param callback
```

```
*/
```

```
private void _getAsyn(String url, final ResultCallback callback)
```

```
{
```

```
    final Request request = new Request.Builder()
```

```
        .url(url)
```

```
        .build();
```

```
    deliveryResult(callback, request);
```

```
}
```

```
/**  
 * 同步的 Post 请求  
 *  
 * @param url  
 * @param params post 的参数  
 * @return  
 */  
  
private Response _post(String url, Param... params) throws  
IOException  
{  
    Request request = buildPostRequest(url, params);  
  
    Response response =  
        mOkHttpClient.newCall(request).execute();  
  
    return response;  
}
```

```
/**  
 * 同步的 Post 请求  
 *  
 * @param url  
 * @param params post 的参数  
 * @return 字符串  
 */  
  
private String _postAsString(String url, Param... params) throws  
IOException  
{  
    Response response = _post(url, params);  
  
    return response.body().string();  
}  
  
/**
```

```
* 异步的 post 请求

*

* @param url

* @param callback

* @param params

*/



private void _postAsyn(String url, final ResultCallback callback,
Param... params)

{



    Request request = buildPostRequest(url, params);

    deliveryResult(callback, request);





}





/**



* 异步的 post 请求

*




```

```
* @param url

* @param callback

* @param params

*/
private void _postAsyn(String url, final ResultCallback callback,
Map<String, String> params)

{
    Param[] paramsArr = map2Params(params);

    Request request = buildPostRequest(url, paramsArr);

    deliveryResult(callback, request);

}

/**
 * 同步基于 post 的文件上传

* @param params
```

```
* @return

*/
private Response _post(String url, File[] files, String[] fileKeys, Param...
params) throws IOException

{
    Request request = buildMultipartFormRequest(url, files, fileKeys,
params);

    return mOkHttpClient.newCall(request).execute();

}

private Response _post(String url, File file, String fileKey) throws
IOException
{
    Request request = buildMultipartFormRequest(url, new
File[]{file}, new String[]{fileKey}, null);

    return mOkHttpClient.newCall(request).execute();

}
```

```
private Response _post(String url, File file, String fileKey, Param...
params) throws IOException

{

    Request request = buildMultipartFormRequest(url, new
File[]{file}, new String[]{fileKey}, params);

    return mOkHttpClient.newCall(request).execute();

}

/**
 * 异步基于 post 的文件上传
 *
 * @param url
 *
 * @param callback
 *
 * @param files
 *
 * @param fileKeys
 *
 * @throws IOException

```

```
*/\n\nprivate void _postAsyn(String url, ResultCallback callback, File[] files,\nString[] fileKeys, Param... params) throws IOException\n{\n    Request request = buildMultipartFormRequest(url, files, fileKeys,\nparams);\n\n    deliveryResult(callback, request);\n}\n\n/**\n * 异步基于 post 的文件上传，单文件不带参数上传\n *\n * @param url\n *\n * @param callback\n *\n * @param file\n *\n * @param fileKey\n *\n * @throws IOException
```

```
*/\n\nprivate void _postAsyn(String url, ResultCallback callback, File file,\nString fileKey) throws IOException\n{\n    Request request = buildMultipartFormRequest(url, new\n    File[]{file}, new String[]{fileKey}, null);\n\n    deliveryResult(callback, request);\n}\n\n/**\n * 异步基于 post 的文件上传，单文件且携带其他 form 参数上传\n *\n * @param url\n *\n * @param callback\n *\n * @param file\n *\n * @param fileKey\n *\n * @param params
```

```
* @throws IOException  
  
*/  
  
private void _postAsyn(String url, ResultCallback callback, File file,  
String fileKey, Param... params) throws IOException  
  
{  
  
    Request request = buildMultipartFormRequest(url, new  
File[]{file}, new String[]{fileKey}, params);  
  
    deliveryResult(callback, request);  
  
}  
  
/**  
  
 * 异步下载文件  
  
 *  
  
 * @param url  
  
 * @param destFileDir 本地文件存储的文件夹  
  
 * @param callback  
  
*/
```

```
private void _downloadAsyn(final String url, final String destFileDir,
final ResultCallback callback)

{

    final Request request = new Request.Builder()

        .url(url)

        .build();

    final Call call = mOkHttpClient.newCall(request);

    call.enqueue(new Callback()

    {

        @Override

        public void onFailure(final Request request, final

IOException e)

        {

            sendFailedStringCallback(request, e, callback);

        }

        @Override
```

```
public void onResponse(Response response)

{

    InputStream is = null;

    byte[] buf = new byte[2048];

    int len = 0;

    FileOutputStream fos = null;

    try

    {

        is = response.body().byteStream();

        File file = new File(destFileDir, getFileName(url));

        fos = new FileOutputStream(file);

        while ((len = is.read(buf)) != -1)

        {

            fos.write(buf, 0, len);

        }

        fos.flush();

    }

}
```

```
//如果下载文件成功，第一个参数为文件的绝对路径

sendSuccessResultCallback(file.getAbsolutePath(),

callback);

} catch (IOException e)

{

    sendFailedStringCallback(response.request(), e,

callback);

}

} finally

{

try

{

if (is != null) is.close();

} catch (IOException e)

{

}

try

{
```

```
        if (fos != null) fos.close();

    } catch (IOException e)

    {

}

}

};

}

}

private String getFileName(String path)

{

    int separatorIndex = path.lastIndexOf("/");

    return (separatorIndex < 0) ? path :

path.substring(separatorIndex + 1, path.length());

}
```

```
/**  
  
 * 加载图片  
  
 *  
  
 * @param view  
  
 * @param url  
  
 * @throws IOException  
  
 */  
  
private void _displayImage(final ImageView view, final String url,  
final int errorResId)  
  
{  
  
    final Request request = new Request.Builder()  
  
        .url(url)  
  
        .build();  
  
    Call call = mOkHttpClient.newCall(request);  
  
    call.enqueue(new Callback())
```

```
{  
  
    @Override  
  
    public void onFailure(Request request, IOException e)  
  
    {  
  
        setErrorResId(view, errorResId);  
  
    }  
  
  
  
  
    @Override  
  
    public void onResponse(Response response)  
  
    {  
  
        InputStream is = null;  
  
        try  
  
        {  
  
            is = response.body().byteStream();  
  
            ImageUtils.ImageSize actualImageSize =  
            ImageUtils.getImageSize(is);  
        }  
    }  
}
```

```
    ImageUtils.ImageSize imageViewSize =  
  
    ImageUtils.getImageViewSize(view);  
  
    int inSampleSize =  
    ImageUtils.calculateInSampleSize(actualImageSize, imageViewSize);  
  
    try  
  
    {  
  
        is.reset();  
  
    } catch (IOException e)  
  
    {  
  
        response = _getAsyn(url);  
  
        is = response.body().byteStream();  
  
    }  
  
    BitmapFactory.Options ops = new  
    BitmapFactory.Options();  
  
    ops.inJustDecodeBounds = false;  
  
    ops.inSampleSize = inSampleSize;
```

```
final Bitmap bm = BitmapFactory.decodeStream(is,
null, ops);

mDelivery.post(new Runnable()

{

@Override

public void run()

{

view.setImageBitmap(bm);

}

});

} catch (Exception e)

{

setErrorResId(view, errorResId);

}

} finally

{
```

```
        if (is != null) try

        {

            is.close();

        } catch (IOException e)

        {

            e.printStackTrace();

        }

    }

});
```

```
    }
```

```
private void setErrorResId(final ImageView view, final int errorResId)

{
```

```
mDelivery.post(new Runnable()

{

    @Override

    public void run()

    {

        view.setImageResource(errorResId);

    }

});

}

//*****对外公布的方法*****



public static Response getAsyn(String url) throws IOException

{
```

```
        return getInstance()._getAsyn(url);

    }

public static String getAsString(String url) throws IOException

{

    return getInstance()._getAsString(url);

}

public static void getAsyn(String url, ResultCallback callback)

{

    getInstance()._getAsyn(url, callback);

}

public static Response post(String url, Param... params) throws

IOException
```



```
    public static void postAsyn(String url, final ResultCallback callback,  
        Map<String, String> params)
```

```
{
```

```
    getInstance()._postAsyn(url, callback, params);
```

```
}
```

```
    public static Response post(String url, File[] files, String[] fileKeys,  
        Param... params) throws IOException
```

```
{
```

```
    return getInstance()._post(url, files, fileKeys, params);
```

```
}
```

```
    public static Response post(String url, File file, String fileKey) throws  
        IOException
```

```
{
```

```
    return getInstance()._post(url, file, fileKey);
```

```
    }

    public static Response post(String url, File file, String fileKey, Param...
params) throws IOException

    {

        return getInstance()._post(url, file, fileKey, params);

    }

    public static void postAsyn(String url, ResultCallback callback, File[]
files, String[] fileKeys, Param... params) throws IOException

    {

        getInstance()._postAsyn(url, callback, files, fileKeys, params);

    }

    public static void postAsyn(String url, ResultCallback callback, File
file, String fileKey) throws IOException
```



```
public static void displayImage(final ImageView view, String url)  
{
```

```
    getInstance()._displayImage(view, url, -1);
```

```
}
```

```
public static void downloadAsyn(String url, String destDir,  
ResultCallback callback)
```

```
{
```

```
    getInstance()._downloadAsyn(url, destDir, callback);
```

```
}
```

```
////////////////////////////////////////////////////////////////////////
```

```
private Request buildMultipartFormRequest(String url, File[] files,
```

```
String[] fileKeys, Param[]

params)

{

params = validateParam(params);

MultipartBuilder builder = new MultipartBuilder()

.type(MultipartBuilder.FORM);

for (Param param : params)

{

builder.addPart(Headers.of("Content-Disposition",

"form-data; name=\"" + param.key + "\""),

RequestBody.create(null, param.value));

}

if (files != null)

{

RequestBody fileBody = null;
```

```
        for (int i = 0; i < files.length; i++)  
  
        {  
  
            File file = files[i];  
  
            String fileName = file.getName();  
  
            fileBody =  
                RequestBody.create(MediaType.parse(guessMimeType(fileName)), file);  
  
            //TODO 根据文件名设置 contentType  
  
            builder.addPart(Headers.of("Content-Disposition",  
                "form-data; name=\"" + fileKeys[i] +  
                "\"; filename=\"" + fileName + "\"),  
                fileBody);  
  
        }  
  
    }  
  
    RequestBody requestBody = builder.build();  
  
    return new Request.Builder()  
        .url(url)
```

```
        .post(requestBody)

        .build();

    }

private String guessMimeType(String path)

{
    FileNameMap fileNameMap =
URLConnection.getFileNameMap();

    String contentTypeFor =
fileNameMap.getContentTypeFor(path);

    if (contentTypeFor == null)

    {
        contentTypeFor = "application/octet-stream";
    }

    return contentTypeFor;
}
```

```
private Param[] validateParam(Param[] params)

{
    if (params == null)
        return new Param[0];

    else return params;
}
```

```
private Param[] map2Params(Map<String, String> params)

{
    if (params == null) return new Param[0];

    int size = params.size();

    Param[] res = new Param[size];

    Set<Map.Entry<String, String>> entries = params.entrySet();

    int i = 0;

    for (Map.Entry<String, String> entry : entries)
```



```
    @Override

        public void onFailure(final Request request, final
IOException e)

    {

        sendFailedStringCallback(request, e, callback);

    }

    @Override

        public void onResponse(final Response response)

    {

        try

    {

        final String string = response.body().string();

        if (callback.mType == String.class)

    {

        sendSuccessResultCallback(string, callback);

    }


```

```
        } else

        {

            Object o = mGson.fromJson(string,
callback.mType);

            sendSuccessResultCallback(o, callback);

        }

    }

} catch (IOException e)

{

    sendFailedStringCallback(response.request(), e,
callback);

} catch (com.google.gson.JsonParseException e)//Json
解析的错误

{

    sendFailedStringCallback(response.request(), e,
callback);
```

```
    }

}

});

}

private void sendFailedStringCallback(final Request request, final
Exception e, final ResultCallback callback)

{

    mDelivery.post(new Runnable()

    {

        @Override

        public void run()

        {

            if (callback != null)

                callback.onError(request, e);

        }

    });

}
```

```
    }

    });

}

private void sendSuccessResultCallback(final Object object, final
ResultCallback callback)

{
    mDelivery.post(new Runnable()

{
    @Override

    public void run()

{
    if (callback != null)

{
    callback.onResponse(object);

}
}
```

```
    }

});
```

private Request buildPostRequest(String url, Param[] params)

```
if (params == null)
```

```
{
```

```
    params = new Param[0];
```

```
}
```

FormEncodingBuilder builder = new FormEncodingBuilder();

```
for (Param param : params)
```

```
{
```

```
    builder.add(param.key, param.value);
```

```
}
```

RequestBody requestBody = builder.build();

```
return new Request.Builder()
```

.url(url)

.post(requestBody)

.build();

}

```
public static abstract class ResultCallback<T>
```

{

Type mType;

```
public ResultCallback()
```

{

```
mType = getSuperclassTypeParameter(getClass());
```

}

```
static Type getSuperclassTypeParameter(Class<?> subclass)

{

    Type superclass = subclass.getGenericSuperclass();

    if (superclass instanceof Class)

    {

        throw new RuntimeException("Missing type

parameter.");

    }

    ParameterizedType parameterized = (ParameterizedType)

superclass;

    return

$Gson$Types.canonicalize(parameterized.getActualTypeArguments()[0])

;

}

public abstract void onError(Request request, Exception e);
```

```
    public abstract void onResponse(T response);  
  
}
```

```
public static class Param
```

```
{  
  
    public Param()  
  
{
```

```
    public Param(String key, String value)
```

```
{  
  
    this.key = key;
```

```
    this.value = value;
```

```
}
```

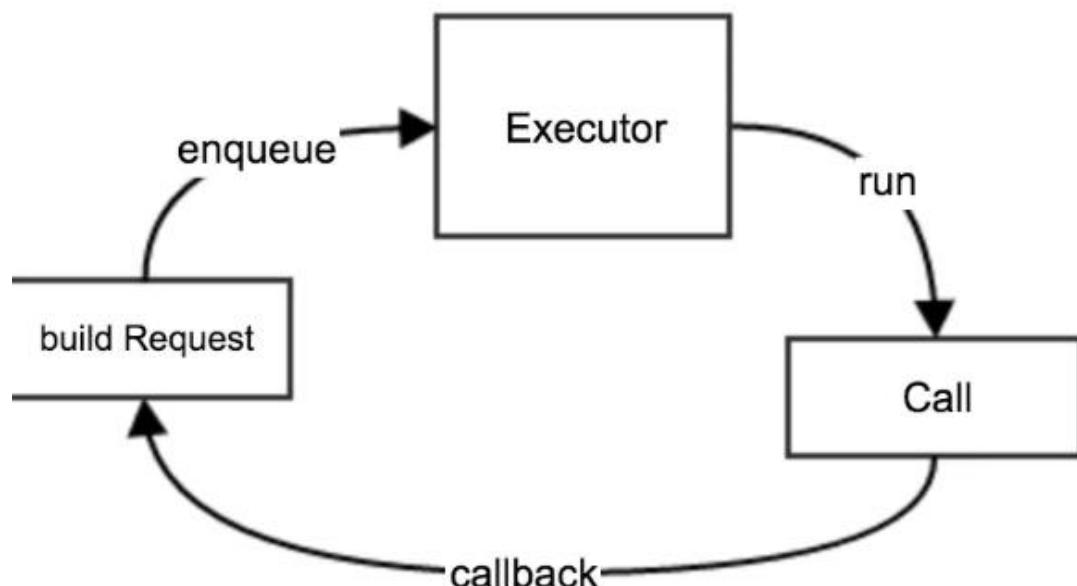
```
    String key;
```

```
        String value;  
    }  
}
```

从 HTTP 到 Retrofit

没有 HTTP 框架的日子

我们先来看一下没有 HTTP 框架以前，我们是如何做请求的。



1. 首先 build request 参数
2. 因为不能在主线程请求 HTTP，所以你得有个 Executer 或者线程
3. enqueue 后，通过线程去 run 你的请求
4. 得到服务器数据后，callback 回调给你的上层。

大概是以上 4 大步骤，在没有框架的年代，想要做一次请求，是万分痛苦的，你需要自己管理线程切换，需要自己解析读取数据，解析数据成对象，切换回主线程，回调给上层。

这段空白的时间持续了很久。从我 10 年工作起到 12 年，因为写烦了重复的代码，所以就得想办法，把那些变化的地方封装起来，也只是简单的封装。好在官方出了 AsyncTask，虽然坑很多，但如果再自己维护一个队列，基本不会出现问题。

更好的地方是数据格式从 xml 变成 json 了。gson 解放了双手，再也不用解析 dom 了。

早些时期的 HTTP 框架

后来慢慢出了不少真正的 HTTP 框架。Stay 也借鉴了很多文章，封装了一套适用于自身业务需求的框架。

这个时期的框架有个特点，就是拼了命去支持所有类型。比方说 **Volley** 支持直接返回 **Bitmap**。**xUtils** 不仅大而全，而且连多线程下载也要支持。在资源匮乏的时代，它们的存在有它们的道理。但如果说现在还用 **Volley** 做图片请求，还在用 **xUtils** 或 **Afinal** 里的各个模块。那就说不过去了。术业有专攻，百家争鸣的时期，难道不该选择最好的那一个吗？(Stay 没真的用过 **xUtils** 和 **Afinal** 这种组合框架，潜意识告诉我，它们有毒，一旦某个环节出问题或者需要扩展，那代价就太大了)

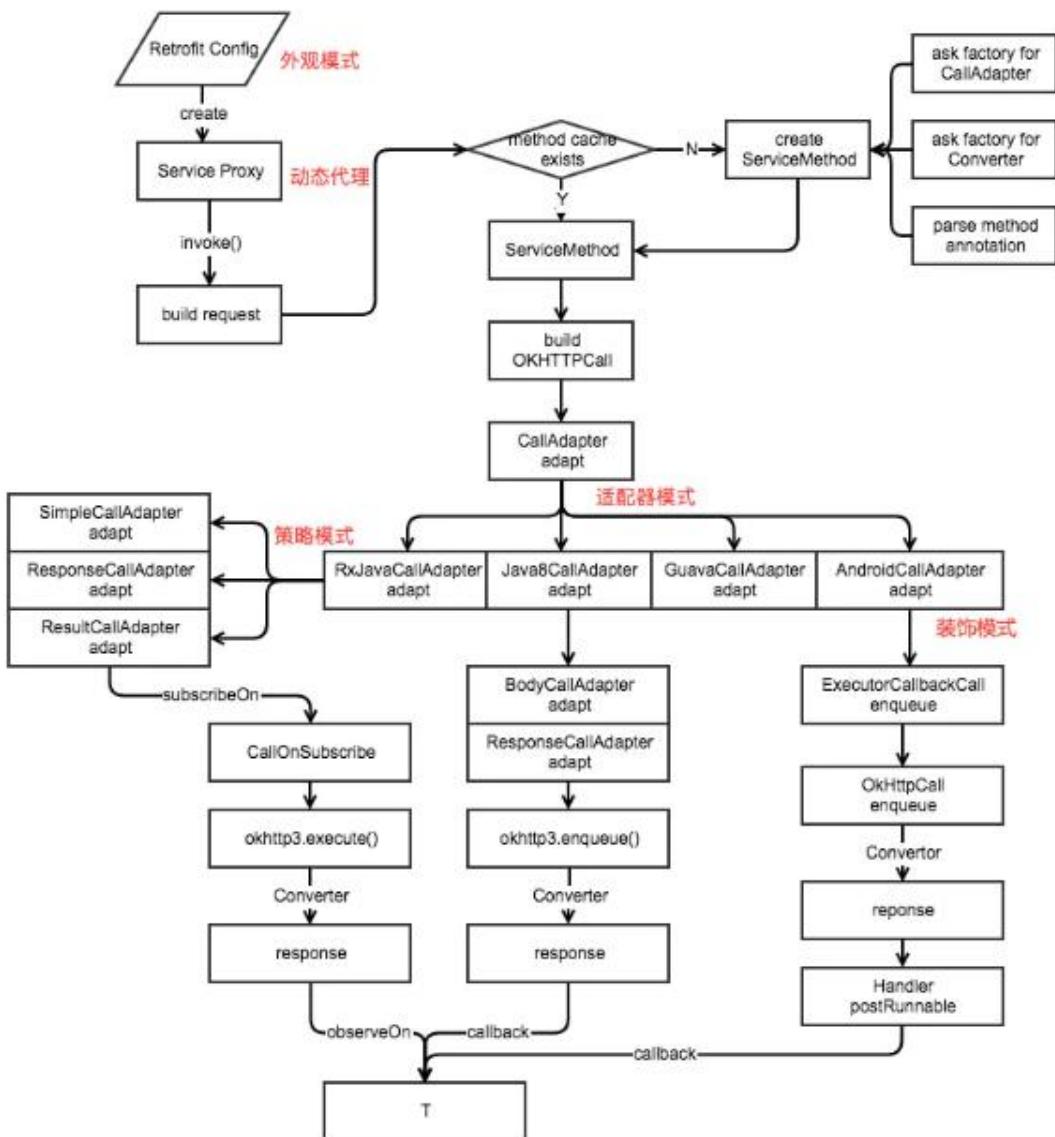
Retrofit

好吧，介绍完 HTTP 框架的发展，让我们单纯的说说 **Retrofit** 吧。

tips: 本文以 **retrofit 最新版本 2.0.1** 为例，大家也可以去 **github** 下源码，找 **tag** 为'parent-2.0.1'就可以。目前代码变动比较大。2.0.1 已经使用 **okhttp3** 了，而我项目中 2.0.0-beta2 还是 **okhttp2.5**。

retrofit 的最大特点就是解耦，要解耦就需要大量的设计模式，假如一点设计模式都不懂的人，可能很难看懂 **retrofit**。

先来看一张 Stay 画的精简流程图(如有错误，请斧正)，类图就不画了。



Stay 在一些设计模式很明确的地方做了标记。

外观模式，动态代理，策略模式，观察者模式。当然还有 Builder 模式，工厂等这些简单的我就没标。

先简述下流程吧：

1.

通过门面 Retrofit 来 build 一个 Service Interface 的 proxy

```
public <T> T create(final Class<T> service) {
    Utils.validateServiceInterface(service);
    if (validateEagerly) {
        eagerlyValidateMethods(service);
    }
    return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>[] { service },
        new InvocationHandler() {
            private final Platform platform = Platform.get();

            @Override public Object invoke(Object proxy, Method method, Object... args)
                throws Throwable {
                // If the method is a method from Object then defer to normal invocation.
                if (method.getDeclaringClass() == Object.class) {
                    return method.invoke(this, args);
                }
                if (platform.isDefaultMethod(method)) {
                    return platform.invokeDefaultMethod(method, service, proxy, args);
                }
                ServiceMethod serviceMethod = loadServiceMethod(method);
                OkHttpCall okHttpCall = new OkHttpCall<?>(serviceMethod, args);
                return serviceMethod.callAdapter.adapt(okHttpCall);
            }
        });
}
```

当你调用这个 Service Interface 中的某个请求方法，会被 proxy 拦截

```
interface MyService {
    GET("user/me")
    Observable<User> getUser()
}
```

- 通过 **ServiceMethod** 来解析 `invoke` 的那个方法，通过**解析**注解，传参，将它们封装成我们所熟悉的 `request`。然后通过具体的返回值类型，让之前配置的工厂生成具体的 **CallAdapter**, **ResponseConverter**，这俩我们稍后再解释。
- • new 一个 `OkHttpCall`, 这个 `OkHttpCall` 算是 `OkHttp` 的包装类, 用它跟 `OkHttp` 对接，所有 `OkHttp` 需要的参数都可以看这个类。当然也还是可以扩展一个新的 `Call` 的，比如 `HttpURLConnectionCall`。但是有点耦合。看下图标注：

```

public <T> T create(final Class<T> service) {
    Utils.validateServiceInterface(service);
    if (validateEagerly) {
        eagerlyValidateMethods(service);
    }
    return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>[] { service },
        new InvocationHandler() {
            private final Platform platform = Platform.get();

            @Override public Object invoke(Object proxy, Method method, Object... args)
                throws Throwable {
                // If the method is a method from Object then defer to normal invocation.
                if (method.getDeclaringClass() == Object.class) {
                    return method.invoke(this, args);
                }
                if (platform.isDefaultMethod(method)) {
                    return platform.invokeDefaultMethod(method, service, proxy, args);
                }
                ServiceMethod serviceMethod = loadServiceMethod(method);
                OkHttpCall okHttpCall = new OkHttpCall<>(serviceMethod, args);
                return serviceMethod.callAdapter.adapt(okHttpCall);
            }
        });
}

```

- 红框中显式的指明了 `OkHttpCall`, 而不是通过工厂来生成 `Call`。所以如果你不想改源码, 重新编译, 那你就只能使用 `OkHttp` 了。不过这不碍事。(可能也是因为还在持续更新中, 所以这块可能后面会改进的)
- • 生成的 `CallAdapter` 有四个工厂, 分别对应不同的平台, RxJava, Java8, Guava 还有一个 Retrofit 默认的。这个 `CallAdapter` 不太好用中文解释。简单来说就是用来将 `Call` 转成 `T` 的一个策略。因为这里具体请求是耗时操作, 所以你需要 `CallAdapter` 去管理线程。怎么管理, 继续往下看。
- • 比如 RxJava 会根据调用方法的返回值, 如 `Response<'T>` |`Result<'T>`|`Observable<'T>` , 生成不同的 `CallAdapter`。实际上就是对 RxJava 的回调方式做封装。比如将 `response` 再拆解为 `success` 和 `error` 等。(这块还是需要在了解 RxJava 的基础上去理解, 以后有时间可以再详细做分析)
- • 在步骤 5 中, 我们说 `CallAdapter` 还管理线程。比方说 RxJava, 我们知道, 它最大的优点可以指定方法在什么线程下执行。如图

```

Observable<Response<HashMap>> request = getApiService().loginSSO(requestData);
request.subscribeOn(Schedulers.newThread())
    .observeOn(AndroidSchedulers.mainThread())

```

我们在子线程订阅(`subscribeOn`), 在主线程观察(`observeOn`)。具体它是如何做的呢。我们看下源码。

```

@Override public <R> Observable<Response<R>> adapt(Call<R> call) {
    Observable<Response<R>> observable = Observable.create(new CallOnSubscribe<>(call));
    if (scheduler != null) {
        return observable.subscribeOn(scheduler);
    }
    return observable;
}

```

- 在 `adapt Call` 时, `subscribeOn` 了, 所以就切换到子线程中了。
- • 在 `adapt Call` 中, 具体的调用了 `Call execute()`, `execute()` 是同步的, `enqueue()` 是异步的。因为 RxJava 已经切换了线程, 所以这里用同步方法 `execute()`。

```
@Override public void request(long n) {
    if (n < 0) throw new IllegalArgumentException("n < 0: " + n);
    if (n == 0) return; // Nothing to do when requesting 0.
    if (!compareAndSet(false, true)) return; // Request was already triggered.

    try {
        Response<T> response = call.execute();
        if (!subscriber.isUnsubscribed()) {
            subscriber.onNext(response);
        }
    } catch (Throwable t) {
        Exceptions.throwIfFatal(t);
        if (!subscriber.isUnsubscribed()) {
            subscriber.onError(t);
        }
        return;
    }

    if (!subscriber.isUnsubscribed()) {
        subscriber.onCompleted();
    }
}
```

1.

接下来的具体请求，就是 OkHttp 的事情了，retrofit 要做成的就是等待返回值。在步骤 4 中，我们说 OkHttpCall 是 OkHttp 的包装类，所以将 OkHttp 的 response 转换成我们要的 T，也是在 OkHttpCall 中执行的。

2.

3.

当然具体的解析转换操作也不是 OkHttpCall 来做的，因为它也不知道数据格式是什么样的。所以它只是将 response 包装成 retrofit 标准下的 response。

4.

5.

Converter->ResponseConverter，很明显，它是数据转换器。它将 response 转换成我们具体想要的 T。Retrofit 提供了很多 converter factory。比如 Gson，Jackson，xml，protobuf 等等。你需要什么，就配置什么工厂。在 Service 方法上声明泛型具体类型就可以了。

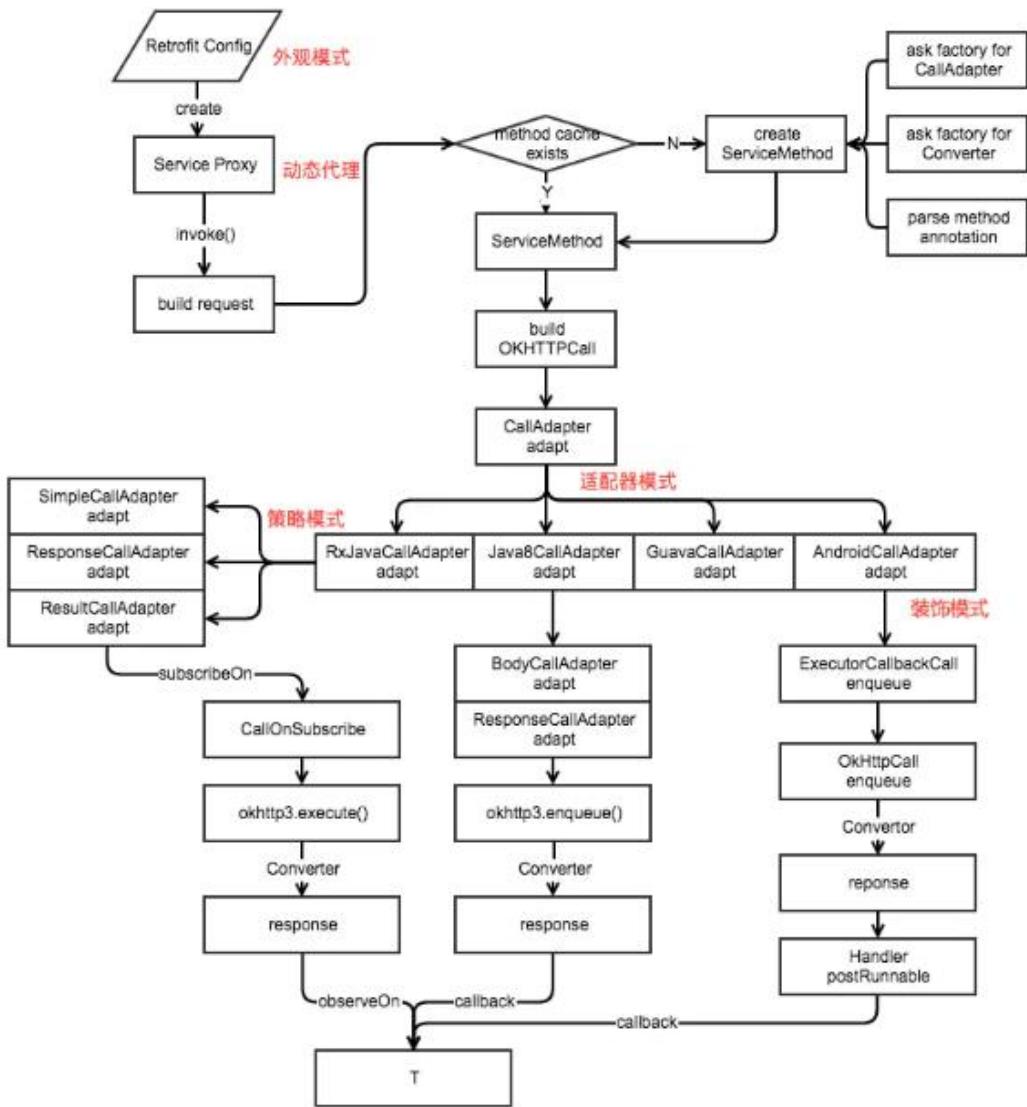
6.

7.

最后，通过声明的 observeOn 线程回调给上层。这样上层就拿到了最终结果。至于结果再如何处理，那就是上层的事了。

8.

再来回顾下 Stay 画的流程图：



这真是漫长的旅行，Stay 也是 debug 一个个单步调试才梳理出来的流程。当然其中还有很多巧妙的解耦方式，我这里就不赘述了。大家可以看看源码分析下，当真是设计模式的经典示例。

Retrofit 是如何工作的

疑惑

1. 我们调用接口的方法后是怎么发送请求的？这背后发生了什么？
2. Retrofit 与 OkHttp 是怎么合作的？
3. Retrofit 中的数据究竟是怎么处理的？它是怎么返回 RxJava.Observable 的？

Retrofit 的基本使用

```
public interface ApiService{  
  
    @GET("data/Android/" + GankConfig.PAGE_COUNT + "/{page}")  
  
    Observable<GankResponse> getAndroid(@Path("page") int page);}  
  
// Builder 模式来构建 retrofitRetrofit retrofit = new Retrofit.Builder()  
  
    .baseUrl(baseUrl)  
  
    .addConverterFactory(GsonConverterFactory.create(new Gson  
Builder().create()))  
  
    .addCallAdapterFactory(RxJavaCallAdapterFactory.create())  
  
    .client(okHttpClient)  
  
    .build(); // 通过 retrofit.create 方法来生成 service(非四大  
组件中的 Service) ApiService service = retrofit.create(ApiService.class);/  
// 发起请求 获取数据 Observable<GankResponse> observable = service.getAndroi  
d(1);  
  
observable....
```

Retrofit 就这样经过简单的配置后就可以向服务器请求数据了，超级简单。

Retrofit.create 方法分析

Retrofit 的 create 方法作为 Retrofit 的入口，当然得第一个分析。

```
public <T> T create(final Class<T> service) {  
  
    // 验证接口是否合理  
  
    Utils.validateServiceInterface(service);  
  
    // 默认 false  
  
    if (validateEagerly) {  
  
        eagerlyValidateMethods(service);  
  
    }  
  
    // 动态代理
```

```
    return (T) Proxy.newProxyInstance(service.getClassLoader(), new Clas
s<?>[] { service },

    new InvocationHandler() {

        // 平台的抽象，指定默认的 CallbackExecutor CallAdapterFactory 用,
        这里 Android 平台是 Android (Java8 iOS 咱不管)

        private final Platform platform = Platform.get();

        // ApiService 中的方法调用都会走到这里

        @Override public Object invoke(Object proxy, Method method, Obj
ect... args)

            throws Throwable {

            // If the method is a method from Object then defer to normal
            invocation.

            // 注释已经说明 Object 的方法不管

            if (method.getDeclaringClass() == Object.class) {

                return method.invoke(this, args);

            }

            // java8 的默认方法，Android 暂不支持默认方法，所以暂时也不需要管

            if (platform.isDefaultMethod(method)) {

                return platform.invokeDefaultMethod(method, service, proxy,
args);

            }

            // 重点了 后面分析

            // 为 Method 生成一个 ServiceMethod

            ServiceMethod serviceMethod = loadServiceMethod(method);

            // 再包装成 OkHttpCall
```

```
        OkHttpCall okHttpCall = new OkHttpCall<>(serviceMethod, args);
        // 请求

        return serviceMethod.callAdapter.adapt(okHttpCall);

    }

});
```

在上面的代码中可以看到，Retrofit 的主要原理是利用了 Java 的[动态代理技术](#)，把 ApiService 的方法调用集中到了 InvocationHandler.invoke，再构建了 ServiceMethod，OKHttpCall，返回 callAdapter.adapt 的结果。
要弄清楚，还需要分析那最后三行代码。
一步一步来。

ServiceMethod 的职责以及 loadServiceMethod 分析

我认为 ServiceMethod 是接口[具体方法的抽象](#)，它主要负责解析它对应的 method 的各种参数（它有各种如 parseHeaders 的方法），比如注解（@Get），入参，另外还负责获取 callAdapter, responseConverter 等 Retrofit 配置，好为后面的 okhttp3.Request 做好参数准备，它的 toRequest 为 OkHttp 提供 Request，可以说它承载了后续 Http 请求所需的一切参数。

再分析 loadServiceMethod，比较简单。

```
// serviceMethodCache 的定义 private final Map<Method, ServiceMethod> serviceMethodCache = new LinkedHashMap<>();

// 获取 method 对应的 ServiceMethod

ServiceMethod loadServiceMethod(Method method) {

    ServiceMethod result;

    synchronized (serviceMethodCache) {

        // 先从缓存去获取

        result = serviceMethodCache.get(method);

        if (result == null) {

            // 缓存中没有 则新建，并存入缓存

            result = new ServiceMethod.Builder(this, method).build();
        }
    }
}
```

```
        serviceMethodCache.put(method, result);

    }

}

return result;

}
```

loadServiceMethod 方法，负责 为 method 生成一个 ServiceMethod ， 并且给 ServiceMethod 做了缓存。

动态代理是有一定的性能损耗的，并且 ServiceMethod 的创建伴随着各种注解参数解析，这也是耗时间的，在加上一个 App 调用接口是非常频繁的，如果每次接口请求都需要重新生成那么有浪费资源损害性能的可能，所以这里做了一份缓存来提高效率。

OkHttpCall

再接下去往后看 OkHttpCall okHttpCall = new OkHttpCall<>(serviceMethod, args);，是再为 ServiceMethod 以及 args(参数)生成了一个 OkHttpCall。

从 OkHttpCall 这个名字来看就能猜到，它是对 OkHttp3.Call 的组合包装，事实上，它也确实是。(OkHttpCall 中有一个成员 okhttp3.Call rawCall)。

callAdapter.adapt 流程分析

最后 return serviceMethod.callAdapter.adapt(okHttpCall) 似乎是走到了最后一步。

如果说前面的都是准备的话，那么到这里就是真的要行动了。

来分析一下，这里涉及到的 callAdapter,是由我们配置 Retrofit 的 addCallAdapterFactory 方法中传入的 RxJavaCallAdapterFactory.create()生成，实例为 RxJavaCallAdapterFactory。

实例的生成大致流程为：

```
ServiceMethod.Bulider.Build()
->ServiceMethod.createCallAdapter()
->retrofit.callAdapter()
->adapterFactories 遍历
    ->最终到 RxJavaCallAdapterFactory.get()#getCallAdapter()
```

```
    ->return return new SimpleCallAdapter(observableType, scheduler);
```

由于使用了 RxJava ， 我们最终得到的 callAdapter 为 SimpleCallAdapter， 所以接下去分析 SimpleCallAdapter 的 adapt 方法：

这里涉及到的 CallOnSubscriber 后面有给出：

```
@Override public <R> Observable<R> adapt(Call<R> call) {
```

```
// 这里的 call 是 OkHttpCall okHttpCall = new OkHttpCall<>(serviceMethod, args) 生成的 okHttpCall

Observable<R> observable = Observable.create(new CallOnSubscribe<>(call)) //

.flatMap(new Func1<Response<R>, Observable<R>>() {

    @Override public Observable<R> call(Response<R> response) {

        if (response.isSuccessful()) {

            return Observable.just(response.body());

        }

        return Observable.error(new HttpException(response));

    }

});

if (scheduler != null) {

    return observable.subscribeOn(scheduler);

}

return observable;

}

}

static final class CallOnSubscribe<T> implements Observable.OnSubscribe<Response<T>> {

    private final Call<T> originalCall;

    CallOnSubscribe(Call<T> originalCall) {

        this.originalCall = originalCall;
```

```
}

@Override public void call(final Subscriber<? super Response<T>> subscriber) {

    // Since Call is a one-shot type, clone it for each new subscriber.

    final Call<T> call = originalCall.clone();

    // Attempt to cancel the call if it is still in-flight on unsubscription.

    // 当我们取消订阅的时候 会取消请求 棒棒哒

    subscriber.add(Subscriptions.create(new Action0() {

        @Override public void call() {

            call.cancel();

        }

    }));

}

try {

    // call 是 OkHttpCall 的实例

    Response<T> response = call.execute();

    if (!subscriber.isUnsubscribed()) {

        subscriber.onNext(response);

    }

} catch (Throwable t) {

    Exceptions.throwIfFatal(t);

    if (!subscriber.isUnsubscribed()) {
```

```
        subscriber.onError(t);  
  
    }  
  
    return;  
  
}  
  
if (!subscriber.isUnsubscribed()) {  
  
    subscriber.onCompleted();  
  
}  
  
}  
  
}  
  
}
```

SimpleCallAdapter.adapt 很简单，创建一个 Observable 获取 CallOnSubscribe 中的 Response<T> 通过 flatMap 转成 Observable<R> 后返回。这里去发送请求获取数据的任务在 CallOnSubscribe.call 方法之中。并且最后走到了 okhttpCall.execute 中去了。

```
// OkHttpCall.execute

@Override public Response<T> execute() throws IOException {
    okhttp3.Call call;

    synchronized (this) {
        //同一个请求 不能执行两次
        if (executed) throw new IllegalStateException("Already executed.");
    };

    executed = true;

    // ...省略 Execption 处理
}
```

```
call = rawCall;

if (call == null) {

    try {

        // 创建 okhttp3.call

        call = rawCall = createRawCall();

    } catch (IOException | RuntimeException e) {

        creationFailure = e;

        throw e;

    }

}

if (canceled) {

    call.cancel();

}

// 请求并解析 response 这个 call 是 okhttp3.call 是真交给 OkHttp 去发送
// 请求了

return parseResponse(call.execute());

}

// 解析 response

Response<T> parseResponse(okhttp3.Response rawResponse) throws IOException {

    //... 省略一些处理 只显示关键代码

    try {
```

```
T body = serviceMethod.toResponse(catchingBody);

return Response.success(body, rawResponse);

} catch (RuntimeException e) {

catchingBody.throwIfCaught();

throw e;

}

}

// serviceMethod.toResponse

T toResponse(ResponseBody body) throws IOException {

// 还记得吗？这就是我们配置 Retrofit 时候的 converter

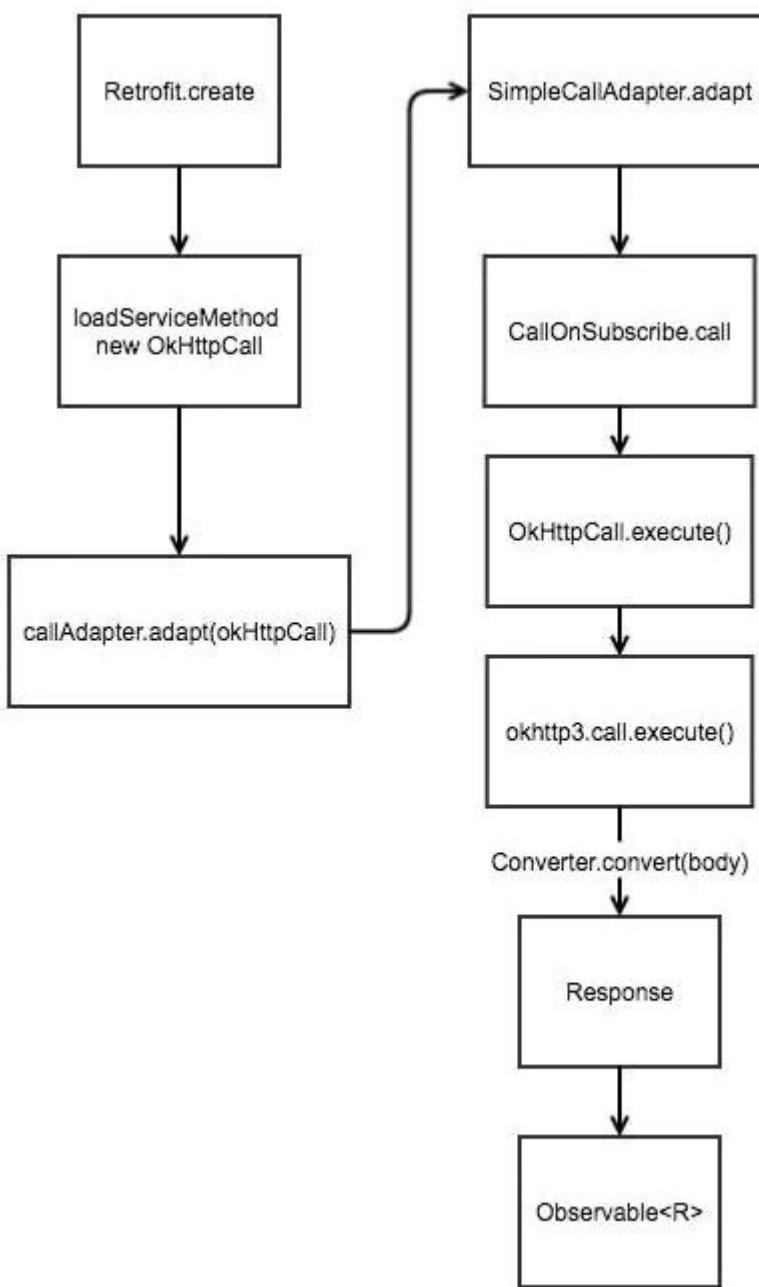
return responseConverter.convert(body);

}
```

经过一连串的处理，最终在 `OkHttpCall.execute()` 的方法中生成 `okhttp3.Call` 交给 `OkHttpClient` 去发送请求，再由我们配置的 `Converter`(本文为 `GsonConverterFactory`) 处理 `Response`, 返回给 `SimpleCallAdapter` 处理，返回我们最终所需要的 `Observable`。

流程分析流程图总结

总体的流程图整理如下：



二十二、最流行图片加载库：Glide

[Android 图片加载框架最全解析（一），Glide 的基本用法](#)

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：

```
1 dependencies {  
2     compile 'com.github.bumptech.glide:glide:3.7.0'  
3 }
```

如果你还在使用 Eclipse，可以点击 [这里](#) 下载 Glide 的 jar 包。

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：

```
1 <uses-permission android:name="android.permission.INTERNET" />
```

复制

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：

```
1 http://cn.bing.com/az/hprichbg/rb/Dongdaemun_ZH-CN10736487148_1920x1080
```

复制

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) {

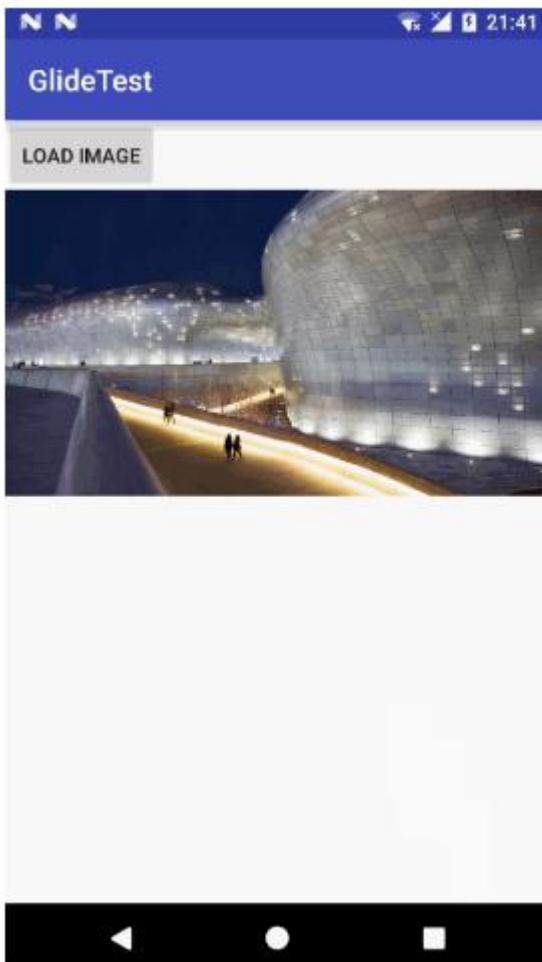
    String url =
"http://cn.bing.com/az/hprichbg/rb/Dongdaemun_ZH-CN10736487148
_1920x1080.jpg";

    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：

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

复制

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，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。

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：

```
1 Glide.with(this)
2     .load(url)
3     .placeholder(R.drawable.loading)
4     .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，不会再去找网络下载了，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：

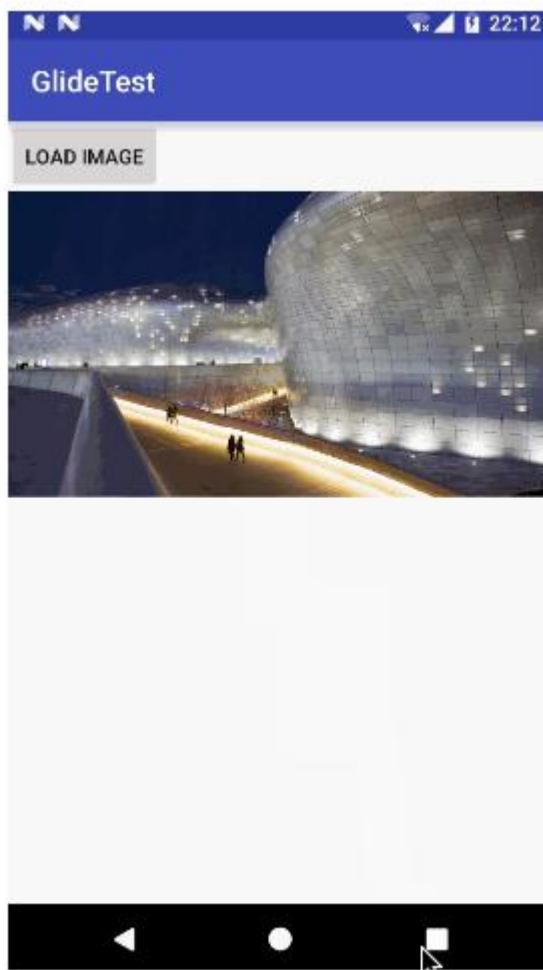
```
Glide.with(this)
    .load(url)
```

```
.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，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。

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。

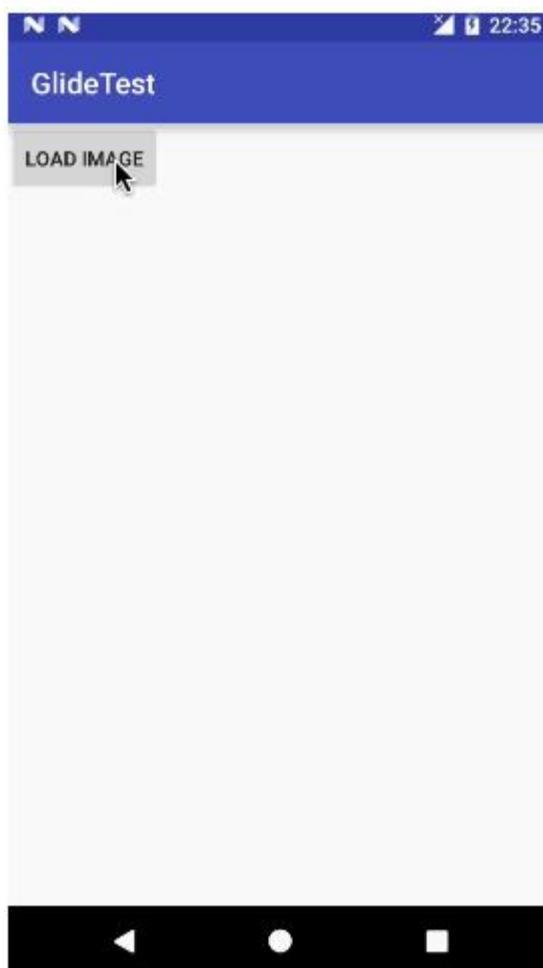
异常占位图的用法相信你已经可以猜到了，首先准备一张 error.jpg 图片，然后修改 Glide 加载部分的代码，如下所示：

```
1 Glide.with(this)
2     .load(url)
3     .placeholder(R.drawable.loading)
4     .error(R.drawable.error)
5     .diskCacheStrategy(DiskCacheStrategy.NONE)
6     .into(imageView);
```

复制

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：



这样我们就把 Glide 提供的占位图功能都掌握了。

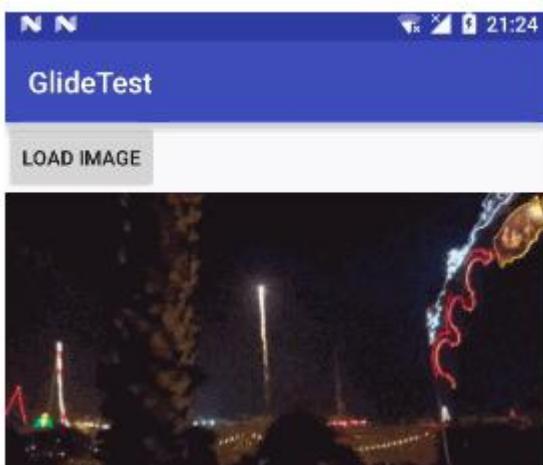
指定图片格式

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：

```
1 http://p1.pstatp.com/large/166200019850062839d3
```

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：



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。

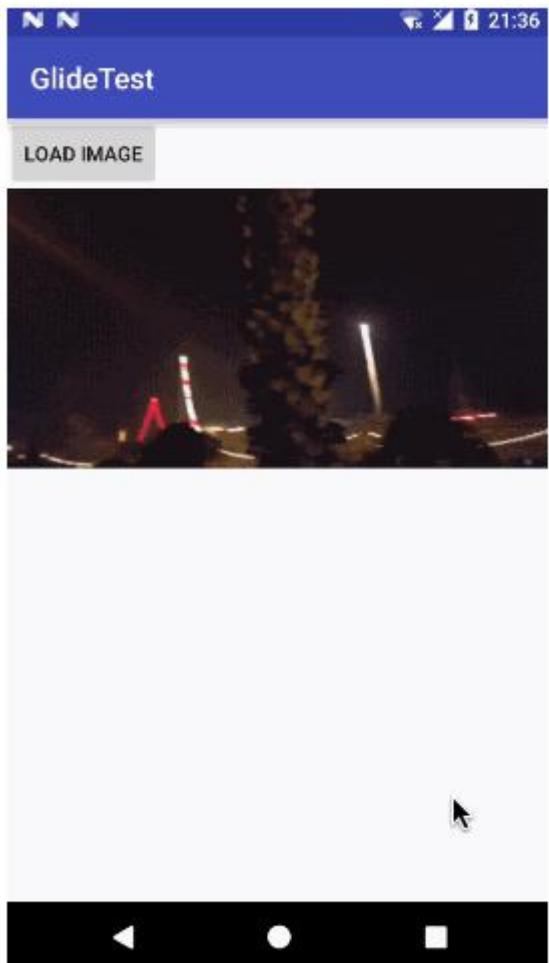
想实现这个功能仍然非常简单，我们只需要再串接一个新的方法就可以了，如下所示：

```
1 Glide.with(this)
2     .load(url)
3     .asBitmap()
4     .placeholder(R.drawable.loading)
5     .error(R.drawable.error)
6     .diskCacheStrategy(DiskCacheStrategy.NONE)
7     .into(imageView);
```

复制

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：



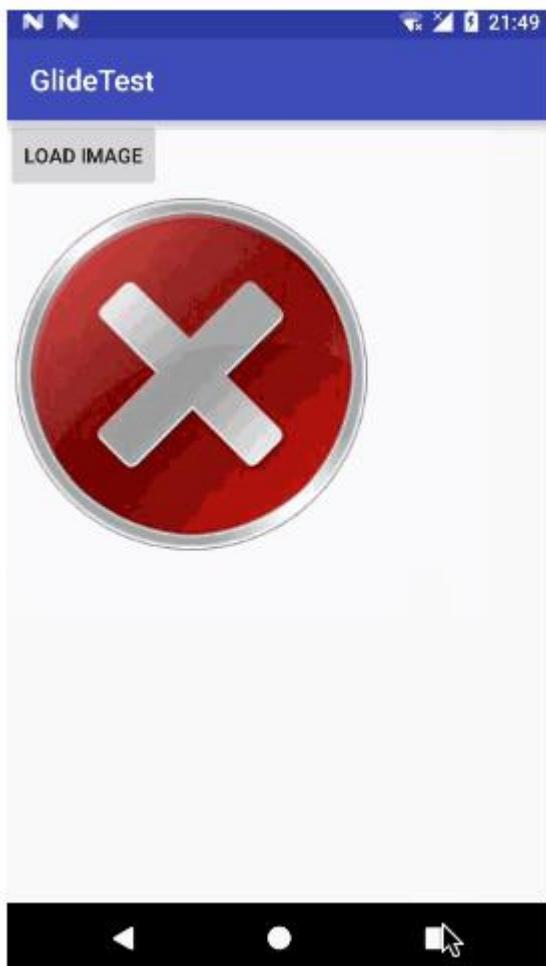
由于调用了 `asBitmap()`方法，现在 GIF 图就无法正常播放了，而是会在界面上显示第一帧的图片。

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

```
1 Glide.with(this)
2     .load(url)
3     .asGif()
4     .placeholder(R.drawable.loading)
5     .error(R.drawable.error)
6     .diskCacheStrategy(DiskCacheStrategy.NONE)
7     .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。



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：

```
1 Glide.with(this)
2     .load(url)
3     .placeholder(R.drawable.loading)
4     .error(R.drawable.error)
5     .diskCacheStrategy(DiskCacheStrategy.NONE)
6     .override(100, 100)
7     .into(imageView);
```

复制

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。

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

开始阅读

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

1. with()

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

```
public class Glide {
    ...
    public static RequestManager with(Context context) {
        RequestManagerRetriever retriever = RequestManagerRetriever.get();
```

```
        return retriever.get(context);
    }

    public static RequestManager with(Activity activity) {
        RequestManagerRetriever retriever = RequestManagerRetriever.get();
        return retriever.get(activity);
    }

    public static RequestManager with(FragmentActivity activity) {
        RequestManagerRetriever retriever = RequestManagerRetriever.get();
        return retriever.get(activity);
    }

    @TargetApi(Build.VERSION_CODES.HONEYCOMB)
    public static RequestManager with(android.app.Fragment fragment) {
        RequestManagerRetriever retriever = RequestManagerRetriever.get();
        return retriever.get(fragment);
    }

    public static RequestManager with(Fragment fragment) {
        RequestManagerRetriever retriever = RequestManagerRetriever.get();
        return retriever.get(fragment);
    }
}
```

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

而 RequestManagerRetriever 的实例 get()方法中的逻辑是什么样的呢？我们一起来看一看：

```
public class RequestManagerRetriever implements Handler.Callback {

    private static final RequestManagerRetriever INSTANCE = new RequestManagerRetriever();
```

```

private volatile RequestManager applicationManager;

...

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

private RequestManager getApplicationManager(Context context) {
    // Either an application context or we're on a background thread.
    if (applicationManager == null) {
        synchronized (this) {
            if (applicationManager == null) {
                // Normally pause/resume is taken care of by the fragment we add to
                // the fragment or activity.
                // However, in this case since the manager attached to the application
                // will not receive lifecycle
                // events, we must force the manager to start resumed using
                // ApplicationLifecycle.
                applicationManager = new
                    RequestManager(context.getApplicationContext(),
                        new ApplicationLifecycle(), new
                            EmptyRequestManagerTreeNode());
            }
        }
    }
    return applicationManager;
}

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

```

```

        }

        return getApplicationManager(context);
    }

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

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

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

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

```

```

activity");
    }
}

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

@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
RequestManagerFragment           getRequestManagerFragment(final
android.app.FragmentManager fm) {
    RequestManagerFragment     current     =     (RequestManagerFragment)
fm.findFragmentByTag(FRAGMENT_TAG);
    if (current == null) {
        current = pendingRequestManagerFragments.get(fm);
        if (current == null) {
            current = new RequestManagerFragment();
            pendingRequestManagerFragments.put(fm, current);
            fm.beginTransaction().add(current,
FRAGMENT_TAG).commitAllowingStateLoss();
            handler.obtainMessage(ID_REMOVE_FRAGMENT_MANAGER,
fm).sendToTarget();
        }
    }
    return current;
}

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

```

```

        current.getRequestManagerTreeNode());
        current.setRequestManager(requestManager);
    }
    return requestManager;
}

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

RequestManager supportFragmentGet(Context context, FragmentManager fm) {
    SupportRequestManagerFragment           current           =
getSupportRequestManagerFragment(fm);
    RequestManager requestManager = current.getRequestManager();
    if (requestManager == null) {
        requestManager = new RequestManager(context, current.getLifecycle(),
current.getRequestManagerTreeNode());
        current.setRequestManager(requestManager);
    }
    return requestManager;
}

...
}

```

上述代码虽然看上去逻辑有点复杂，但是将它们梳理清楚后还是很简单的。

RequestManagerRetriever 类中看似有很多个 get()方法的重载，什么 Context

参数，Activity 参数，Fragment 参数等等，实际上只有两种情况而已，即传入 Application 类型的参数，和传入非 Application 类型的参数。

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

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

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

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

2. load()

由于 with()方法返回的是一个 RequestManager 对象，那么很容易就能想到，load()方法是在 RequestManager 类当中的，所以说我们首先要看的就是 RequestManager 这个类。不过在上一篇文章中我们学过，Glide 是支持图片 URL 字符串、图片本地路径等等加载形式的，因此 RequestManager 中也有很多个 load()方法的重载。但是这里我们不可能把每个 load()方法的重载都看一遍，因此我们就只选其中一个加载图片 URL 字符串的 load()方法来进行研究吧。

RequestManager 类的简化代码如下所示：

```
public class RequestManager implements LifecycleListener {
```

```
...
```

```
/**  
 * Returns a request builder to load the given {@link String}.  
 * signature.
```

```

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

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

private <T> DrawableTypeRequest<T> loadGeneric(Class<T> modelClass) {
    ModelLoader<T, InputStream> streamModelLoader =
Glide.buildStreamModelLoader(modelClass, context);
    ModelLoader<T, ParcelFileDescriptor> fileDescriptorModelLoader =
        Glide.buildFileDescriptorModelLoader(modelClass, context);
    if (modelClass != null && streamModelLoader == null && fileDescriptorModelLoader ==
null) {
        throw new IllegalArgumentException("Unknown type " + modelClass + ". You must
provide a Model of a type for"
            + " which there is a registered ModelLoader, if you are using a custom

```

```
model, you must first call"
        + " Glide#register with a ModelLoaderFactory for your custom model
class");
    }
    return optionsApplier.apply(
        new      DrawableTypeRequest<T>(modelClass,      streamModelLoader,
fileDescriptorModelLoader, context,
        glide, requestTracker, lifecycle, optionsApplier));
}
...
}
```

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

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

其实 loadGeneric()方法也没几行代码，这里分别调用了 Glide.buildStreamModelLoader()方法和 Glide.buildFileDescriptorModelLoader()方法来获得 ModelLoader 对象。ModelLoader 对象是用于加载图片的，而我们给 load()方法传入不同类型的参数，这里也会得到不同的 ModelLoader 对象。不过 buildStreamModelLoader()方法内部的逻辑还是蛮复杂的，这里就不展开介绍了，要不然篇幅实在收不住，

感兴趣的话你可以自己研究。由于我们刚才传入的参数是 String.class，因此最终得到的是 StreamStringLoader 对象，它是实现了 ModelLoader 接口的。

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

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

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

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

        if (transcoder == null) {
            transcoder = glide.buildTranscoder(resourceClass, transcodedClass);
        }
        DataLoadProvider<ImageVideoWrapper, Z> dataLoadProvider =
        glide.buildDataProvider(ImageVideoWrapper.class,
            resourceClass);
        ImageVideoModelLoader<A> modelLoader =
            new
```

```

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

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

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

/**
 * Attempts to always load the resource as a {@link com.bumptech.glide.load.resource.gif.GifDrawable}.
 *
<p>
 * If the underlying data is not a GIF, this will fail. As a result, this should only be used
if the model
 *
 * represents an animated GIF and the caller wants to interact with the GifDrawable
directly. Normally using
 *
 * just an {@link DrawableTypeRequest} is sufficient because it will determine
whether or

```

```
*      not the given data represents an animated GIF and return the appropriate animated
or not animated
*      {@link android.graphics.drawable.Drawable} automatically.
* </p>
*
*      @return A new request builder for loading a {@link
com.bumptech.glide.load.resource.gif.GifDrawable}.
*/
public GifTypeRequest<ModelType> asGif() {
    return optionsApplier.apply(new GifTypeRequest<ModelType>(this,
streamModelLoader, optionsApplier));
}

...
}
```

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

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

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

```
public class DrawableRequestBuilder<ModelType>
    extends GenericRequestBuilder<ModelType, ImageVideoWrapper, GifBitmapWrapper,
GlideDrawable>
```

```
    implements BitmapOptions, DrawableOptions {

        DrawableRequestBuilder(Context context, Class<ModelType> modelClass,
            LoadProvider<ModelType>,           ImageVideoWrapper,           GifBitmapWrapper,
        GlideDrawable> loadProvider, Glide glide,
            RequestTracker requestTracker, Lifecycle lifecycle) {
        super(context, modelClass, loadProvider, GlideDrawable.class, glide, requestTracker,
lifecycle);
        // Default to animating.
        crossFade();
    }

    public DrawableRequestBuilder<ModelType> thumbnail(
        DrawableRequestBuilder<?> thumbnailRequest) {
    super.thumbnail(thumbnailRequest);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType> thumbnail(
    GenericRequestBuilder<?, ?, ?, GlideDrawable> thumbnailRequest) {
    super.thumbnail(thumbnailRequest);
    return this;
}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    @Override
```

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

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

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

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

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

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

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

@Override
public DrawableRequestBuilder<ModelType> listener(
    RequestListener<? super ModelType, GlideDrawable> requestListener) {
```

```
        super.listener(requestListener);
        return this;
    }

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

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

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

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

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

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

    @Override
    public DrawableRequestBuilder<ModelType> load(ModelType model) {
        super.load(model);
```

```
        return this;
    }

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

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

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

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

DrawableRequestBuilder 中有很多个方法 ,这些方法其实就是 Glide 绝大多数的 API 了。里面有不少我们在上篇文章中已经用过了 ,比如说 placeholder()方法、error()方法、diskCacheStrategy()方法、override()方法等。当然还有很多暂时还没用到的 API ,我们会在后面的文章当中学习。

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

3. into()

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

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

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

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

这里前面一大堆的判断逻辑我们都可以先不用管，等到后面文章讲 transform 的时候会再进行解释，现在我们只需要关注最后一行代码。最后一行代码先是调用了 glide.buildImageViewTarget()方法，这个方法会构建出一个 Target 对象，

Target 对象则是用来最终展示图片用的，如果我们跟进去的话会看到如下代码：

```
<R> Target<R> buildImageViewTarget(ImageView imageView, Class<R>
transcodedClass) {

    return imageViewTargetFactory.buildTarget(imageView,
transcodedClass);

}
```

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

```
public class ImageViewTargetFactory {

    @SuppressWarnings("unchecked")

    public <Z> Target<Z> buildTarget(ImageView view, Class<Z> clazz)

    {
        if (GlideDrawable.class.isAssignableFrom(clazz)) {

            return (Target<Z>) new
GlideDrawableImageViewTarget(view);

        } else if (Bitmap.class.equals(clazz)) {

            return (Target<Z>) new BitmapImageViewTarget(view);

        } else if (Drawable.class.isAssignableFrom(clazz)) {

            return (Target<Z>) new DrawableImageViewTarget(view);

        } else {

            throw new IllegalArgumentException("Unhandled class: " +

```

```
clazz

    + ", try .as*(Class).transcode(ResourceTranscoder)");

}

}

}
```

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

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

```
public <Y extends Target<TranscodeType>> Y into(Y target) {

    Util.assertMainThread();

    if (target == null) {

        throw new IllegalArgumentException("You must pass in a non
```

```
    null Target");
}

if (!isModelSet) {
    throw new IllegalArgumentException("You must first set a model
(try #load());
}

Request previous = target.getRequest();
if (previous != null) {
    previous.clear();
    requestTracker.removeRequest(previous);
    previous.recycle();
}

Request request = buildRequest(target);
target.setRequest(request);
lifecycle.addListener(target);
requestTracker.runRequest(request);

return target;
}
```

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

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

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

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

```
tory()) {  
    thumbnailRequestBuilder.animationFactory =  
    animationFactory;  
}  
  
if (thumbnailRequestBuilder.priority == null) {  
    thumbnailRequestBuilder.priority = getThumbnailPriority();  
}  
  
if (Util.isValidDimensions(overrideWidth, overrideHeight)  
&& !Util.isValidDimensions(thumbnailRequestBuilder.overrideWidth,  
    thumbnailRequestBuilder.overrideHeight)) {  
    thumbnailRequestBuilder.override(overrideWidth,  
    overrideHeight);  
}  
  
ThumbnailRequestCoordinator coordinator = new  
ThumbnailRequestCoordinator(parentCoordinator);  
Request fullRequest = obtainRequest(target, sizeMultiplier,  
priority, coordinator);  
// Guard against infinite recursion.
```

```
    isThumbnailBuilt = true;

    // Recursively generate thumbnail requests.

    Request thumbRequest = thumbnailRequestBuilder.buildRequestRecursive(target, coordinator);

    isThumbnailBuilt = false;

    coordinator.setRequests(fullRequest, thumbRequest);

    return coordinator;

} else if (thumbSizeMultiplier != null) {

    // Base case: thumbnail multiplier generates a thumbnail request,
    but cannot recurse.

    ThumbnailRequestCoordinator coordinator = new
    ThumbnailRequestCoordinator(parentCoordinator);

    Request fullRequest = obtainRequest(target, sizeMultiplier,
    priority, coordinator);

    Request thumbnailRequest = obtainRequest(target,
    thumbSizeMultiplier, getThumbnailPriority(), coordinator);

    coordinator.setRequests(fullRequest, thumbnailRequest);

    return coordinator;

} else {

    // Base case: no thumbnail.

    return obtainRequest(target, sizeMultiplier, priority,
    parentCoordinator);
```

```
    }

}

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

```
        glide.getEngine(),  
        transformation,  
        transcodeClass,  
        isCacheable,  
        animationFactory,  
        overrideWidth,  
        overrideHeight,  
        diskCacheStrategy);  
    }  
}
```

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

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

...

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

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

```
        fallbackDrawable,  
        fallbackResourceId,  
        requestListener,  
        requestCoordinator,  
        engine,  
        transformation,  
        transcodeClass,  
        isMemoryCacheable,  
        animationFactory,  
        overrideWidth,  
        overrideHeight,  
        diskCacheStrategy);  
  
    return request;  
  
}  
  
...  
}
```

可以看到，这里在第 33 行去 new 了一个 GenericRequest 对象，并在最后一行返回，也就是说，obtain()方法实际上获得的就是一个 GenericRequest 对象。另外这里又在第 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：

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

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

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

```
@Override

public void begin() {

    startTime = LogTime.getLogTime();

    if (model == null) {

        onException(null);

        return;

    }

    status = Status.WAITING_FOR_SIZE;

    if (Util.isValidDimensions(overrideWidth, overrideHeight)) {

        onSizeReady(overrideWidth, overrideHeight);

    } else {

        target.getSize(this);

    }

    if (!isComplete() && !isFailed() && canNotifyStatusChanged()) {

        target.onLoadStarted(getPlaceholderDrawable());

    }

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logV("finished      run      method      in      "
            + LogTime.getElapsedMillis(startTime));

    }

}
```

这里我们来注意几个细节，首先如果 model 等于 null，model 也就是我们在第

二步 load()方法中传入的图片 URL 地址，这个时候会调用 onException()方法。

如果你跟到 onException()方法里面去看看，你会发现它最终会调用到一个 setErrorPlaceholder()当中，如下所示：

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

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

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

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

很简单，其实就是将这张 error 占位图显示到 ImageView 上而已，因为现在出现了异常，没办法展示正常的图片了。而如果你仔细看下刚才 begin()方法的第 15 行，你会发现它又调用了一个 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：

```
@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 +
        "\'"));

    }

    ResourceTranscoder<Z, R> transcoder = loadProvider.getTranscoder();

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logV("finished setup for calling load in " +
        LogTime.getElapsedMillis(startTime));

    }

    loadedFromMemoryCache = true;

    loadStatus = engine.load(signature, width, height, dataFetcher,
    loadProvider, transformation, transcoder,
    priority, isMemoryCacheable, diskCacheStrategy, this);

    loadedFromMemoryCache = resource != null;

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logV("finished onSizeReady in " +
        LogTime.getElapsedMillis(startTime));

    }

}
```

从这里开始，真正复杂的地方来了，我们需要慢慢进行分析。先来看一下，在第12行调用了 loadProvider.getModelLoader()方法，那么我们第一个要搞清楚的就是，这个 loadProvider 是什么？要搞清楚这点，需要先回到第二步的 load()方法当中。还记得 load()方法是返回一个 DrawableTypeRequest 对象吗？刚才我们只是分析了 DrawableTypeRequest 当中的 asBitmap() 和 asGif() 方法，并没有仔细看它的构造函数，现在我们重新来看一下 DrawableTypeRequest 类的构造函数：

```
public class DrawableTypeRequest<ModelType> extends  
DrawableRequestBuilder<ModelType> implements DownloadOptions {  
  
    private final ModelLoader<ModelType, InputStream>  
    streamModelLoader;  
  
    private final ModelLoader<ModelType, ParcelFileDescriptor>  
    fileDescriptorModelLoader;  
  
    private final RequestManager.OptionsApplier optionsApplier;  
  
  
    private static <A, Z, R> FixedLoadProvider<A, ImageVideoWrapper,  
Z, R> buildProvider(Glide glide,  
    ModelLoader<A, InputStream> streamModelLoader,  
    ModelLoader<A, ParcelFileDescriptor>  
    fileDescriptorModelLoader, Class<Z> resourceClass,  
    Class<R> transcodedClass,
```

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

```
fileDescriptorModelLoader, Context context, Glide glide,
        RequestTracker requestTracker, Lifecycle lifecycle,
RequestManager.OptionsApplier optionsApplier) {
    super(context, modelClass,
        buildProvider(glide, streamModelLoader,
fileDescriptorModelLoader, GifBitmapWrapper.class,
        GlideDrawable.class, null),
        glide, requestTracker, lifecycle);
    this.streamModelLoader = streamModelLoader;
    this.fileDescriptorModelLoader = fileDescriptorModelLoader;
    this.optionsApplier = optionsApplier;
}
...
}
```

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

那么我们再来看一下 buildProvider()方法里面做了什么，在第 16 行调用了 glide.buildTranscoder()方法来构建一个 ResourceTranscoder，它是用于对图

片进行转码的，由于 ResourceTranscoder 是一个接口，这里实际会构建出一个 GifBitmapWrapperDrawableTranscoder 对象。

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

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

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

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

```
public class ImageVideoModelLoader<A> implements ModelLoader<A,
```

```
ImageVideoWrapper> {

    private static final String TAG = "IVML";

    private final ModelLoader<A, InputStream> streamLoader;

    private final ModelLoader<A, ParcelFileDescriptor>
    fileDescriptorLoader;

    public ImageVideoModelLoader(ModelLoader<A, InputStream>
        streamLoader,
        ModelLoader<A, ParcelFileDescriptor> fileDescriptorLoader)
    {

        if (streamLoader == null && fileDescriptorLoader == null) {
            throw new NullPointerException("At least one of
streamLoader and fileDescriptorLoader must be non null");
        }

        this.streamLoader = streamLoader;
        this.fileDescriptorLoader = fileDescriptorLoader;
    }

    @Override

    public DataFetcher<ImageVideoWrapper> getResourceFetcher(A
model, int width, int height) {
```

```
    DataFetcher<InputStream> streamFetcher = null;

    if (streamLoader != null) {

        streamFetcher = streamLoader.getResourceFetcher(model,
width, height);

    }

    DataFetcher<ParcelFileDescriptor> fileDescriptorFetcher = null;

    if (fileDescriptorLoader != null) {

        fileDescriptorFetcher = fileDescriptorLoader.getResourceFetcher(model, width, height);

    }

    if (streamFetcher != null || fileDescriptorFetcher != null) {

        return new ImageVideoFetcher(streamFetcher,
fileDescriptorFetcher);

    } else {

        return null;

    }

}

static class ImageVideoFetcher implements

DataFetcher<ImageVideoWrapper> {

    private final DataFetcher<InputStream> streamFetcher;
```

```
private final DataFetcher<ParcelFileDescriptor>
fileDescriptorFetcher;

public ImageVideoFetcher(DataFetcher<InputStream>
streamFetcher,
DataFetcher<ParcelFileDescriptor>
fileDescriptorFetcher) {

    this.streamFetcher = streamFetcher;
    this.fileDescriptorFetcher = fileDescriptorFetcher;
}

...
}

}
```

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

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

```
public class Engine implements EngineJobListener,  
    MemoryCache.ResourceRemovedListener,  
    EngineResource.ResourceListener {  
  
    ...  
  
    public <T, Z, R> LoadStatus load(Key signature, int width, int height,  
        DataFetcher<T> fetcher,  
        DataLoadProvider<T, Z> loadProvider, Transformation<Z>  
        transformation, ResourceTranscoder<Z, R> transcoder,  
        Priority priority, boolean isMemoryCacheable,  
        DiskCacheStrategy diskCacheStrategy, ResourceCallback cb) {  
        Util.assertMainThread();  
        long startTime = LogTime.getLogTime();  
  
        final String id = fetcher.getId();  
        EngineKey key = keyFactory.buildKey(id, signature, width, height,
```

```
loadProvider.getCacheDecoder(),  
        loadProvider.getSourceDecoder(),      transformation,  
loadProvider.getEncoder(),  
        transcoder, loadProvider.getSourceEncoder());
```

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

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

```
resources", startTime, key);

    }

    return null;
}

EngineJob current = jobs.get(key);

if (current != null) {

    current.addCallback(cb);

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logWithTimeAndKey("Added      to      existing      load",

startTime, key);

    }

    return new LoadStatus(cb, current);
}

EngineJob      engineJob      =      engineJobFactory.build(key,
isMemoryCacheable);

DecodeJob<T, Z, R> decodeJob = new DecodeJob<T, Z, R>(key,
width, height, fetcher, loadProvider, transformation,
transcoder,      diskCacheProvider,      diskCacheStrategy,
priority);

EngineRunnable runnable = new EngineRunnable(engineJob,
```

```
decodeJob, priority);

        jobs.put(key, engineJob);

        engineJob.addCallback(cb);

        engineJob.start(runnable);

    }

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logWithTimeAndKey("Started new load", startTime, key);

    }

    return new LoadStatus(cb, engineJob);

}

...

}
```

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

```
@Override

public void run() {

    if (isCancelled) {

        return;

    }

    Exception exception = null;

    Resource<?> resource = null;

    try {

        resource = decode();

    } catch (Exception e) {

        if (Log.isLoggable(TAG, Log.VERBOSE)) {

            Log.v(TAG, "Exception decoding", e);

        }

        exception = e;

    }

    if (isCancelled) {

        if (resource != null) {

            resource.recycle();

        }

        return;

    }

    if (resource == null) {
```

```
    onLoadFailed(exception);

} else {

    onLoadComplete(resource);

}

}
```

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

```
private Resource<?> decode() throws Exception {

    if (isDecodingFromCache()) {

        return decodeFromCache();

    } else {

        return decodeFromSource();

    }

}
```

decode() 方法中又分了两种情况，从缓存当中去 decode 图片的话就会执行 decodeFromCache()，否则的话就执行 decodeFromSource()。本篇文章中我们不讨论缓存的情况，那么就直接来看 decodeFromSource() 方法的代码吧，如下所示：

```
private Resource<?> decodeFromSource() throws Exception {

    return decodeJob.decodeFromSource();

}
```

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

```
class DecodeJob<A, T, Z> {
```

```
...
```

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

```
private Resource<T> decodeSource() throws Exception {
```

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

```
        decoded = decodeFromSourceData(data);

    } finally {
        fetcher.cleanup();
    }

    return decoded;
}

...
}
```

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

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

```
@Override
public ImageVideoWrapper loadData(Priority priority) throws Exception {
    InputStream is = null;
    if (streamFetcher != null) {
```

```
try {

    is = streamFetcher.loadData(priority);

} catch (Exception e) {

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        Log.v(TAG, "Exception fetching input stream, trying

ParcelFileDescriptor", e);

    }

    if (fileDescriptorFetcher == null) {

        throw e;

    }

}

ParcelFileDescriptor fileDescriptor = null;

if (fileDescriptorFetcher != null) {

    try {

        fileDescriptor = fileDescriptorFetcher.loadData(priority);

    } catch (Exception e) {

        if (Log.isLoggable(TAG, Log.VERBOSE)) {

            Log.v(TAG, "Exception fetching ParcelFileDescriptor", e);

        }

        if (is == null) {

            throw e;

        }

    }

}
```

```
        }

    }

}

return new ImageVideoWrapper(is, fileDescriptor);

}
```

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

```
public class HttpURLConnection implements DataFetcher<InputStream> {

    ...

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

    private InputStream loadDataWithRedirects(URL url, int redirects,
            URL lastUrl, Map<String, String> headers)
```

```
throws IOException {  
  
    if (redirects >= MAXIMUM_REDIRECTS) {  
  
        throw new IOException("Too many (> " +  
MAXIMUM_REDIRECTS + ") redirects!");  
  
    } else {  
  
        // Comparing the URLs using .equals performs additional  
network I/O and is generally broken.  
  
    }  

```

// See
[http://michaelscharf.blogspot.com/2006/11/javaneturlequals-and-hashc
ode-make.html](http://michaelscharf.blogspot.com/2006/11/javaneturlequals-and-hashcode-make.html).

```
try {  
  
    if (lastUrl != null && url.toURI().equals(lastUrl.toURI())) {  
  
        throw new IOException("In re-direct loop");  
  
    }  
  
} catch (URISyntaxException e) {  
  
    // Do nothing, this is best effort.  
  
}  
  
}  
  
urlConnection = connectionFactory.build(url);  
  
for (Map.Entry<String, String> headerEntry : headers.entrySet())  
  
{  
  
    urlConnection.addRequestProperty(headerEntry.getKey(),  
}
```

```
headerEntry.getValue());  
}  
  
urlConnection.setConnectTimeout(2500);  
  
urlConnection.setReadTimeout(2500);  
  
urlConnection.setUseCaches(false);  
  
urlConnection.setDoInput(true);  
  
  
  
// Connect explicitly to avoid errors in decoders if connection  
fails.  
  
urlConnection.connect();  
  
if (isCancelled) {  
  
    return null;  
}  
  
final int statusCode = urlConnection.getResponseCode();  
  
if (statusCode / 100 == 2) {  
  
    return getStreamForSuccessfulRequest(urlConnection);  
} else if (statusCode / 100 == 3) {  
  
    String redirect urlString =  
  
    urlConnection.getHeaderField("Location");  
  
    if (TextUtils.isEmpty	redirect urlString)) {  
  
        throw new IOException("Received empty or null redirect  
url");
```

```
        }

        URL redirectUrl = new URL(url, redirectUrlString);

        return loadDataWithRedirects(redirectUrl, redirects + 1, url,
headers);

    } else {

        if (statusCode == -1) {

            throw new IOException("Unable to retrieve response
code from HttpURLConnection.");

        }

        throw new IOException("Request failed " + statusCode + ": "
+ urlConnection.getResponseMessage());

    }

}
```

```
private InputStream
getStreamForSuccessfulRequest(HttpURLConnection urlConnection)
throws IOException {
    if (TextUtils.isEmpty(urlConnection.getContentEncoding())) {
        int contentLength = urlConnection.getContentLength();
        stream =
ContentLengthInputStream.obtain(urlConnection.getInputStream(),
contentLength);
    }
}
```

```
    } else {

        if (Log.isLoggable(TAG, Log.DEBUG)) {

            Log.d(TAG, "Got non empty content encoding: " +
urlConnection.getContentEncoding());

        }

        stream = urlConnection.getInputStream();

    }

    return stream;

}

...
}
```

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

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

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

decodeFromSourceData()方法的代码如下所示：

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

可以看到，这里在第 7 行调用了 loadProvider.getSourceDecoder().decode()方法来进行解码。loadProvider 就是刚才在 onSizeReady()方法中得到的 FixedLoadProvider，而 getSourceDecoder() 得到的则是一个

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

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

```
...
```

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

```
GifBitmapWrapperResource(wrapper) : null;  
}  
  
private GifBitmapWrapper decode(ImageVideoWrapper source, int  
width, int height, byte[] bytes) throws IOException {  
    final GifBitmapWrapper result;  
    if (source.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

        from the beginning of the stream, we need to

        // pass in a new source containing the buffered stream

        rather than the original stream.

        ImageVideoWrapper forBitmapDecoder = new

        ImageVideoWrapper(bis, source.getFileDescriptor());

        result = decodeBitmapWrapper(forBitmapDecoder, width,

        height);

    }

    return result;

}

private GifBitmapWrapper

decodeBitmapWrapper(ImageVideoWrapper toDecode, int width, int

height) throws IOException {

    GifBitmapWrapper result = null;
```

```
Resource<Bitmap> bitmapResource =  
    bitmapDecoder.decode(toDecode, width, height);  
  
    if (bitmapResource != null) {  
  
        result = new GifBitmapWrapper(bitmapResource, null);  
  
    }  
  
    return result;  
}  
  
...  
}
```

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

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

```
public class ImageVideoBitmapDecoder implements
```

```
ResourceDecoder<ImageVideoWrapper, Bitmap> {

    private final ResourceDecoder<InputStream, Bitmap>
    streamDecoder;

    private final ResourceDecoder<ParcelFileDescriptor, Bitmap>
    fileDescriptorDecoder;

    public ImageVideoBitmapDecoder(ResourceDecoder<InputStream,
        Bitmap> streamDecoder,
        ResourceDecoder<ParcelFileDescriptor, Bitmap>
        fileDescriptorDecoder) {

        this.streamDecoder = streamDecoder;
        this.fileDescriptorDecoder = fileDescriptorDecoder;
    }

    @Override
    public Resource<Bitmap> decode(ImageVideoWrapper source, int
width, int height) throws IOException {
        Resource<Bitmap> result = null;
        InputStream is = source.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 FileDescriptor", e);

            }

        }

    }

    if (result == null) {

        ParcelFileDescriptor fileDescriptor = source.getFileDescriptor();

        if (fileDescriptor != null) {

            result = fileDescriptorDecoder.decode(fileDescriptor,
width, height);

        }

    }

    return result;

}

...
}
```

代码并不复杂，在第 14 行先调用了 source.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 Downampler downampler;  
    private BitmapPool bitmapPool;  
    private DecodeFormat decodeFormat;  
  
    public StreamBitmapDecoder(Downampler downampler,  
        BitmapPool bitmapPool, DecodeFormat decodeFormat) {  
        this.downampler = downampler;  
        this.bitmapPool = bitmapPool;  
        this.decodeFormat = decodeFormat;  
    }  
  
    @Override  
    public Resource<Bitmap> decode(InputStream source, int width, int  
        height) {  
        Bitmap bitmap = downampler.decode(source, bitmapPool,
```

```
width, height, decodeFormat);

    return BitmapResource.obtain(bitmap, bitmapPool);

}

...
}
```

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

```
public      abstract      class      Downampler      implements

BitmapDecoder<InputStream> {

    ...
}

@Override

public Bitmap decode(InputStream is, BitmapPool pool, int outWidth,
int outHeight, DecodeFormat decodeFormat) {

    final ByteArrayPool byteArrayPool = ByteArrayPool.get();

    final byte[] bytesForOptions = byteArrayPool.getBytes();

    final byte[] bytesForStream = byteArrayPool.getBytes();

    final BitmapFactory.Options options = getDefaultOptions();

    // Use to fix the mark limit to avoid allocating buffers that fit
entire images.
```

```
    RecyclableBufferedInputStream bufferedStream = new
    RecyclableBufferedInputStream(
        is, bytesForStream);
    // Use to retrieve exceptions thrown while reading.
    // TODO(#126): when the framework no longer returns partially
    decoded Bitmaps or provides a way to determine
    // if a Bitmap is partially decoded, consider removing.
    ExceptionCatchingInputStream exceptionStream =
        ExceptionCatchingInputStream.obtain(bufferedStream);
    // Use to read data.
    // Ensures that we can always reset after reading an image
    header so that we can still attempt to decode the
    // full image even when the header decode fails and/or
    overflows our read buffer. See #283.
    MarkEnforcingInputStream invalidatingStream = new
    MarkEnforcingInputStream(exceptionStream);
    try {
        exceptionStream.mark(MARK_POSITION);
        int orientation = 0;
        try {
            orientation = new
            ImageHeaderParser(exceptionStream).getOrientation();
        } catch (IOException e) {
            Log.w("ImageHeaderParser", "Error reading image header", e);
        }
    } finally {
        exceptionStream.reset();
    }
}
```

```
        } catch (IOException e) {

            if (Log.isLoggable(TAG, Log.WARN)) {

                Log.w(TAG, "Cannot determine the image
orientation from header", e);

            }

        } finally {

            try {

                exceptionStream.reset();

            } catch (IOException e) {

                if (Log.isLoggable(TAG, Log.WARN)) {

                    Log.w(TAG, "Cannot reset the input stream", e);

                }

            }

        }

        options.inTempStorage = bytesForOptions;

        final int[] inDimens = getDimensions(invalidatingStream,
bufferedStream, options);

        final int inWidth = inDimens[0];

        final int inHeight = inDimens[1];

        final int degreesToRotate = TransformationUtils.getExifOrientationDegrees(orientation);

        final int sampleSize =
```

```
getRoundedSampleSize(degreesToRotate, inWidth, inHeight, outWidth,
outHeight);

    final Bitmap downsampled =
        downsampleWithSize(invalidatingStream,
bufferedStream, options, pool, inWidth, inHeight, sampleSize,
decodeFormat);

    // BitmapFactory swallows exceptions during decodes and
in some cases when inBitmap is non null, may catch
    // and log a stack trace but still return a non null bitmap. To
avoid displaying partially decoded bitmaps,
    // we catch exceptions reading from the stream in our
ExceptionCatchingInputStream and throw them here.
```

```
final           Exception           streamException      =
exceptionStream.getException();

if (streamException != null) {
    throw new RuntimeException(streamException);
}

Bitmap rotated = null;

if (downsampled != null) {
    rotated
        =
TransformationUtils.rotateImageExif(downscaled, pool, orientation);

    if
        (!downscaled.equals(rotated))
```

```
&& !pool.put(downscaled)) {  
    downsampled.recycle();  
}  
}  
  
return rotated;  
}  
finally {  
    byteArrayPool.releaseBytes(bytesForOptions);  
    byteArrayPool.releaseBytes(bytesForStream);  
    exceptionStream.release();  
    releaseOptions(options);  
}  
}  
  
}  
  
private Bitmap downsampleWithSize(MarkEnforcingInputStream is,  
RecyclableBufferedInputStream bufferedStream,  
BitmapFactory.Options options, BitmapPool pool, int  
inWidth, int inHeight, int sampleSize,  
DecodeFormat decodeFormat) {  
    // Prior to KitKat, the inBitmap size must exactly match the size  
    // of the bitmap we're decoding.  
    Bitmap.Config config = getConfig(is, decodeFormat);  
    options.inSampleSize = sampleSize;
```

```
    options.inPreferredConfig = config;

    if ((options.inSampleSize == 1 || Build.VERSION_CODES.KITKAT
<= Build.VERSION.SDK_INT) && shouldUsePool(is)) {

        int targetWidth = (int) Math.ceil(inWidth / (double)
sampleSize);

        int targetHeight = (int) Math.ceil(inHeight / (double)
sampleSize);

        // BitmapFactory will clear out the Bitmap before writing to
it, so getDirty is safe.
```

```
        setInBitmap(options, pool.getDirty(targetWidth,
targetHeight, config));

    }

    return decodeStream(is, bufferedStream, options);
}
```

```
/***
 * A method for getting the dimensions of an image from the given
InputStream.
```

```
*
```

```
* @param is The InputStream representing the image.
```

```
* @param options The options to pass to
```

```
* {@link BitmapFactory#decodeStream(InputStream,
```

```
    android.graphics.Rect,  
    *           BitmapFactory.Options}}.  
    * @return an array containing the dimensions of the image in the  
form {width, height}.  
*/  
  
public int[] getDimensions(MarkEnforcingInputStream is,  
RecyclableBufferedInputStream bufferedStream,  
    BitmapFactory.Options options) {  
    options.inJustDecodeBounds = true;  
    decodeStream(is, bufferedStream, options);  
    options.inJustDecodeBounds = false;  
    return new int[] { options.outWidth, options.outHeight };  
}  
  
  
private static Bitmap decodeStream(MarkEnforcingInputStream is,  
RecyclableBufferedInputStream bufferedStream,  
    BitmapFactory.Options options) {  
    if (options.inJustDecodeBounds) {  
        // This is large, but jpeg headers are not size bounded so  
we need something large enough to minimize  
        // the possibility of not being able to fit enough of the  
header in the buffer to get the image size so
```

```
// that we don't fail to load images. The  
BufferedInputStream will create a new buffer of 2x the  
// original size each time we use up the buffer space  
without passing the mark so this is a maximum  
// bound on the buffer size, not a default. Most of the time  
we won't go past our pre-allocated 16kb.  
  
    is.mark(MARK_POSITION);  
  
} else {  
  
    // Once we've read the image header, we no longer need to  
allow the buffer to expand in size. To avoid  
// unnecessary allocations reading image data, we fix the  
mark limit so that it is no larger than our  
// current buffer size here. See issue #225.  
  
    bufferedStream.fixMarkLimit();  
  
}  
  
final Bitmap result = BitmapFactory.decodeStream(is, null,  
options);  
  
try {  
  
    if (options.inJustDecodeBounds) {  
  
        is.reset();  
  
    }  
  
} catch (IOException e) {
```

```
        if (Log.isLoggable(TAG, Log.ERROR)) {  
            Log.e(TAG, "Exception loading inDecodeBounds=" +  
options.inJustDecodeBounds  
                + " sample=" + options.inSampleSize, e);  
        }  
  
    }  
  
    ...  
}
```

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 Downampler 中得到的是一个 Bitmap 对象，因此这里在第 18 行又调用了 BitmapResource.obtain()方法，将 Bitmap 对象包装成了 Resource<Bitmap> 对象。代码如下所示：

```
public class BitmapResource implements Resource<Bitmap> {

    private final Bitmap bitmap;

    private final BitmapPool bitmapPool;

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

    public BitmapResource(Bitmap bitmap, BitmapPool bitmapPool) {
```

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

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

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

```
@Override
```

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

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

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

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

```
    return result;  
}
```

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

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

```
    this.gifResource = gifResource;

}

/***
 * Returns the size of the wrapped resource.
 */

public int getSize() {

    if (bitmapResource != null) {

        return bitmapResource.getSize();

    } else {

        return gifResource.getSize();

    }

}

/***
 * Returns the wrapped {@link Bitmap} resource if it exists, or null.
 */

public Resource<Bitmap> getBitmapResource() {

    return bitmapResource;

}

/***
```

```
* Returns the wrapped {@link GifDrawable} resource if it exists, or
null.

*/
public Resource<GifDrawable> getGifResource() {
    return gifResource;
}

}
```

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

然后这个 GifBitmapWrapper 对象会一直向上返回，返回到 GifBitmapWrapperResourceDecoder 最外层的 decode()方法的时候，会对它再做一次封装，如下所示：

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

```
        pool.releaseBytes(tempBytes);

    }

    return wrapper != null ? new GifBitmapWrapperResource(wrapper) :
null;

}
```

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

```
public class GifBitmapWrapperResource implements
Resource<GifBitmapWrapper> {

    private final GifBitmapWrapper data;

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

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

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

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

```
@Override  
  
public void recycle() {  
  
    Resource<Bitmap> bitmapResource =  
    data.getBitmapResource();  
  
    if (bitmapResource != null) {  
  
        bitmapResource.recycle();  
  
    }  
  
    Resource<GifDrawable> gifDataResource =  
    data.getGifResource();  
  
    if (gifDataResource != null) {  
  
        gifDataResource.recycle();  
  
    }  
}
```

```
    }  
  
}
```

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

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

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

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

```
private Resource<Z> transformEncodeAndTranscode(Resource<T>  
decoded) {  
  
    long startTime = LogTime.getLogTime();  
  
    Resource<T> transformed = transform(decoded);
```

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

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

而 transcode()方法中又是调用了 transcoder 的 transcode()方法，那么这个 transcoder 是什么呢？其实这也是 Glide 源码特别难懂的原因之一，就是它用到的很多对象都是很早很早之前就初始化的，在初始化的时候你可能完全就没有留意过它，因为一时半会根本就用不着，但是真正需要用到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  
ResourceTranscoder<GifBitmapWrapper, GlideDrawable> {  
    private final ResourceTranscoder<Bitmap, GlideBitmapDrawable>  
        bitmapDrawableResourceTranscoder;  
  
    public GifBitmapWrapperDrawableTranscoder(
```

```
    ResourceTranscoder<Bitmap,           GlideBitmapDrawable>

    bitmapDrawableResourceTranscoder) {

        this.bitmapDrawableResourceTranscoder      =
    bitmapDrawableResourceTranscoder;

    }

@Override

public          Resource<GlideDrawable>

transcode(Resource<GifBitmapWrapper> toTranscode) {

    GifBitmapWrapper gifBitmap = toTranscode.get();

    Resource<Bitmap>           bitmapResource      =
    gifBitmap.getBitmapResource();

    final Resource<? extends GlideDrawable> result;

    if (bitmapResource != null) {

        result          =
    bitmapDrawableResourceTranscoder.transcode(bitmapResource);

    } else {

        result = gifBitmap.getGifResource();

    }

    return (Resource<GlideDrawable>) result;
}
```

```
...
```

```
}
```

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

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

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

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

```
public GlideBitmapDrawableTranscoder(Context context) {  
    this(context.getResources(), Glide.get(context).getBitmapPool());  
}  
  
public GlideBitmapDrawableTranscoder(Resources resources,  
BitmapPool bitmapPool) {  
    this.resources = resources;  
    this.bitmapPool = bitmapPool;  
}  
  
@Override  
public Resource<GlideBitmapDrawable>  
transcode(Resource<Bitmap> toTranscode) {  
    GlideBitmapDrawable drawable = new  
    GlideBitmapDrawable(resources, toTranscode.get());  
    return new GlideBitmapDrawableResource(drawable,  
    bitmapPool);  
}  
  
...  
}
```

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

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

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

```
@Override  
public void run() {  
    if (isCancelled) {  
        return;  
    }  
    Exception exception = null;
```

```
Resource<?> resource = null;

try {

    resource = decode();

} catch (Exception e) {

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        Log.v(TAG, "Exception decoding", e);

    }

    exception = e;

}

if (isCancelled) {

    if (resource != null) {

        resource.recycle();

    }

    return;

}

if (resource == null) {

    onLoadFailed(exception);

} else {

    onLoadComplete(resource);

}

}
```

也就是说，经过第 9 行 decode() 方法的执行，我们最终得到了这个

Resource<GlideDrawable>对象，那么接下来就是如何将它显示出来了。可以看到，这里在第 25 行调用了 onLoadComplete()方法，表示图片加载已经完成了，代码如下所示

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

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

```
class EngineJob implements EngineRunnable.EngineRunnableManager {  
  
    private static final Handler MAIN_THREAD_HANDLER = new  
Handler(Looper.getMainLooper(), new MainThreadCallback());  
  
    private final List<ResourceCallback> cbs = new  
ArrayList<ResourceCallback>();  
  
    ...  
  
    public void addCallback(ResourceCallback cb) {  
        Util.assertMainThread();  
        if (hasResource) {  
            cb.onResourceReady(engineResource);  
        }  
    }  
}
```

```
        } else if (hasException) {

            cb.onException(exception);

        } else {

            cbs.add(cb);

        }

    }

@Override

public void onResourceReady(final Resource<?> resource) {

    this.resource = resource;

    MAIN_THREAD_HANDLER.obtainMessage(MSG_COMPLETE,
this).sendToTarget();

}

private void handleResultOnMainThread() {

    if (isCancelled) {

        resource.recycle();

        return;

    } else if (cbs.isEmpty()) {

        throw new IllegalStateException("Received a resource
without any callbacks to notify");

    }

}
```

```
    engineResource      =      engineResourceFactory.build(resource,
isCacheable);

    hasResource = true;

    engineResource.acquire();

    listener.onEngineJobComplete(key, engineResource);

    for (ResourceCallback cb : cbs) {

        if (! 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).sendToTarget();

}
```

```
private void handleExceptionOnMainThread() {
```

```
    if (isCancelled) {

        return;

    } else if (cbs.isEmpty()) {

        throw new IllegalStateException("Received an exception

without any callbacks to notify");

    }

    hasException = true;

    listener.onEngineJobComplete(key, null);

    for (ResourceCallback cb : cbs) {

        if (! 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>  
        transformation, ResourceTranscoder<Z, R> transcoder, Priority priority,  
        boolean isMemoryCacheable, DiskCacheStrategy  
        diskCacheStrategy, ResourceCallback cb) {  
  
    ...  
  
    EngineJob engineJob = engineJobFactory.build(key,  
        isMemoryCacheable);  
  
    DecodeJob<T, Z, R> decodeJob = new DecodeJob<T, Z, R>(key,  
        width, height, fetcher, loadProvider, transformation,  
        transcoder, diskCacheProvider, diskCacheStrategy,
```

```
priority);

    EngineRunnable runnable = new EngineRunnable(engineJob,
decodeJob, priority);

    jobs.put(key, engineJob);

    engineJob.addCallback(cb);

    engineJob.start(runnable);

if (Log.isLoggable(TAG, Log.VERBOSE)) {

    logWithTimeAndKey("Started new load", startTime, key);

}

return new LoadStatus(cb, engineJob);

}

...
}
```

这次把目光放在第 18 行上面，看到了吗？就是在这里调用的 EngineJob 的 addCallback()方法来注册的一个 ResourceCallback。那么接下来的问题就是， Engine.load()方法的 ResourceCallback 参数又是谁传过来的呢？这就需要回到 GenericRequest 的 onSizeReady()方法当中了，我们看到 ResourceCallback 是 load()方法的最后一个参数，那么在 onSizeReady()方法中调用 load()方法时传入的最后一个参数是什么？代码如下所示：

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

```
SizeReadyCallback,  
    ResourceCallback {  
  
    ...  
  
    @Override  
    public void onSizeReady(int width, int height) {  
        if (Log.isLoggable(TAG, Log.VERBOSE)) {  
            logV("Got          onSizeReady          in          "  
                +  
                LogTime.getElapsedMillis(startTime));  
        }  
        if (status != Status.WAITING_FOR_SIZE) {  
            return;  
        }  
        status = Status.RUNNING;  
        width = Math.round(sizeMultiplier * width);  
        height = Math.round(sizeMultiplier * height);  
        ModelLoader<A,          T>          modelLoader          =  
        loadProvider.getModelLoader();  
        final          DataFetcher<T>          dataFetcher          =  
        modelLoader.getResourceFetcher(model, width, height);  
        if (dataFetcher == null) {
```

```
        onException(new Exception("Failed to load model: \'" +  
model + "\'"));  
  
        return;  
    }  
  
    ResourceTranscoder<Z, R> transcoder =  
loadProvider.getTranscoder();  
  
    if (Log.isLoggable(TAG, Log.VERBOSE)) {  
  
        logV("finished setup for calling load in " +  
LogTime.getElapsedMillis(startTime));  
  
    }  
  
    loadedFromMemoryCache = true;  
  
    loadStatus = engine.load(signature, width, height, dataFetcher,  
loadProvider, transformation,  
  
        transcoder, priority, isMemoryCacheable,  
diskCacheStrategy, this);  
  
    loadedFromMemoryCache = resource != null;  
  
    if (Log.isLoggable(TAG, Log.VERBOSE)) {  
  
        logV("finished onSizeReady in " +  
LogTime.getElapsedMillis(startTime));  
  
    }  
}
```

...

}

请将目光锁定在第 29 行的最后一个参数 ,this。没错 ,就是 this。GenericRequest 本身就实现了 ResourceCallback 的接口 ,因此 EngineJob 的回调最终其实就是回调到了 GenericRequest 的 onResourceReady()方法当中了 ,代码如下所示 :

```
public void onResourceReady(Resource<?> resource) {  
    if (resource == null) {  
        onException(new Exception("Expected to receive a Resource<R>  
with an object of " + transcodeClass  
                + " inside, but instead got null."));  
    }  
    Object received = resource.get();  
    if (received == null)  
        || !transcodeClass.isAssignableFrom(received.getClass())) {  
        releaseResource(resource);  
        onException(new Exception("Expected to receive an object of " +  
transcodeClass  
                + " but instead got " + (received != null ?  
received.getClass() : "") + "{" + received + "}"  
                + " inside Resource{" + resource + "}."))  
    }  
}
```

```
+ (received != null ? "" : " "
+ "To indicate failure return a null Resource object, "
+ "rather than a Resource object containing null
data.");
});

return;
}

if (!canSetResource()) {
    releaseResource(resource);
    // We can't set the status to complete before asking
    canSetResource();
    status = Status.COMPLETE;
    return;
}

onResourceReady(resource, (R) received);
}
```

```
private void onResourceReady(Resource<?> resource, R result) {
    // We must call isFirstReadyResource before setting status.
    boolean isFirstResource = isFirstReadyResource();
    status = Status.COMPLETE;
    this.resource = resource;
```

```
    if (requestListener == null || !requestListener.onResourceReady(result,
        model, target, loadedFromMemoryCache,
        isFirstResource)) {

        GlideAnimation<R>           animation      =
        animationFactory.build(loadedFromMemoryCache, isFirstResource);

        target.onResourceReady(result, animation);

    }

    notifyLoadSuccess();

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logV("Resource ready in " + LogTime.getElapsedMillis(startTime)
        + " size: "
        + (resource.getSize() * TO_MEGABYTE) + " fromCache: "
        + loadedFromMemoryCache);

    }

}
```

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

那么这个 `target` 又是什么呢 ? 这个又需要向上翻很久了 , 在第三步 `into()` 方法的一开始 , 我们就分析了在 `into()` 方法的最后一行 , 调用了

glide.buildImageViewTarget()方法来构建出一个 Target ,而这个 Target 就是一个 GlideDrawableImageViewTarget 对象。

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

```
public class GlideDrawableImageViewTarget extends ImageViewTarget<GlideDrawable> {

    private static final float SQUARE_RATIO_MARGIN = 0.05f;

    private int maxLoopCount;

    private GlideDrawable resource;

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

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

    @Override
    public void onResourceReady(GlideDrawable resource,
```

```

GlideAnimation<? super GlideDrawable> animation) {

    if (!resource.isAnimated()) {

        float viewRatio = view.getWidth() / (float) view.getHeight();

        float drawableRatio = resource.getIntrinsicWidth() / (float)
resource.getIntrinsicHeight();

        if (Math.abs(viewRatio - 1f) <= SQUARE_RATIO_MARGIN
&& Math.abs(drawableRatio - 1f) <=
SQUARE_RATIO_MARGIN) {

            resource = new SquaringDrawable(resource,
view.getWidth());

        }

    }

    super.onResourceReady(resource, animation);

    this.resource = resource;

    resource.setLoopCount(maxLoopCount);

    resource.start();

}

@Override

protected void setResource(GlideDrawable resource) {

    view.setImageDrawable(resource);

}

```

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

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

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

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

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

    ...

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

    protected abstract void setResource(Z resource);
}

}
```

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

那子类的 setResource()方法是怎么实现的呢？回头再来看一下 GlideDrawableImageViewTarget 的 setResource()方法，没错，调用的

`view.setImageDrawable()`方法，而这个 view 就是 ImageView。代码执行到这里，图片终于也就显示出来了。

那么，我们对 Glide 执行流程的源码分析，到这里也终于结束了。

Android 图片加载框架最全解析（三），深入探究 Glide 的缓存机制

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> transformation,  
        ResourceTranscoder<Z, R> transcoder,  
        Priority priority, boolean isMemoryCacheable, DiskCacheStrategy  
        diskCacheStrategy, ResourceCallback cb) {  
        Util.assertMainThread();  
        long startTime = LogTime.getLogTime();  
  
        final String id = fetcher.getId();  
        EngineKey key = keyFactory.buildKey(id, signature, width, height,  
        loadProvider.getCacheDecoder(),  
        loadProvider.getSourceDecoder(), transformation, loadProvider.getEncoder(),  
        transcoder, loadProvider.getSourceEncoder());  
  
        ...  
    }  
  
    ...  
}
```

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，我就不再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：

```
1 Glide.with(this)
2     .load(url)
3     .skipMemoryCache(true)
4     .into(imageView);
```

复制

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 {

    public static <T, Y> ModelLoader<T, Y> buildModelLoader(Class<T> modelClass, 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(modelClass,
resourceClass);
    }

    public static Glide get(Context context) {
        if (glide == null) {
            synchronized (Glide.class) {
                if (glide == null) {
                    Context applicationContext = context.getApplicationContext();
                    List<GlideModule> modules = new
ManifestParser(applicationContext).parse();
                    GlideBuilder builder = new GlideBuilder(applicationContext);
                    for (GlideModule module : modules) {
                        module.applyOptions(applicationContext, builder);
                    }
                    glide = builder.createGlide();
                    for (GlideModule module : modules) {
                        module.registerComponents(applicationContext, glide);
                    }
                }
            }
        }
        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().availableProcessors());  
            sourceService = new FifoPriorityThreadPoolExecutor(cores);  
        }  
  
        if (diskCacheService == null) {  
            diskCacheService = new FifoPriorityThreadPoolExecutor(1);  
        }  
  
        MemorySizeCalculator calculator = new  
            MemorySizeCalculator(context);  
  
        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.getMemoryCacheSize());

    }

    if (diskCacheFactory == null) {

        diskCacheFactory = new

InternalCacheDiskCacheFactory(context);

    }

    if (engine == null) {

        engine = new Engine(memoryCache, diskCacheFactory,

diskCacheService, sourceService);

    }

    if (decodeFormat == null) {

        decodeFormat = DecodeFormat.DEFAULT;

    }

    return new Glide(engine, memoryCache, bitmapPool, context,

decodeFormat);

}

}
```

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>  
        transformation, ResourceTranscoder<Z, R> transcoder,  
        Priority priority, boolean isMemoryCacheable,  
        DiskCacheStrategy diskCacheStrategy, ResourceCallback cb) {
```

```
Util.assertMainThread();

long startTime = LogTime.getLogTime();

final String id = fetcher.getId();

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

EngineResource<?> cached = loadFromCache(key,
isMemoryCacheable);

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

EngineResource<?> active = loadFromActiveResources(key,
```

```
isMemoryCacheable);

    if (active != null) {

        cb.onResourceReady(active);

        if (Log.isLoggable(TAG, Log.VERBOSE)) {

            logWithTimeAndKey("Loaded resource from active
resources", startTime, key);

        }

        return null;
    }
}

EngineJob current = jobs.get(key);

if (current != null) {

    current.addCallback(cb);

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logWithTimeAndKey("Added to existing load",
startTime, key);

    }

    return new LoadStatus(cb, current);
}

EngineJob engineJob = engineJobFactory.build(key,
isMemoryCacheable);
```

```
    DecodeJob<T, Z, R> decodeJob = new DecodeJob<T, Z, R>(key,  
    width, height, fetcher, loadProvider, transformation,  
    transcoder,    diskCacheProvider,    diskCacheStrategy,  
    priority);  
  
    EngineRunnable runnable = new EngineRunnable(engineJob,  
    decodeJob, priority);  
  
    jobs.put(key, engineJob);  
  
    engineJob.addCallback(cb);  
  
    engineJob.start(runnable);  
  
  
  
    if (Log.isLoggable(TAG, Log.VERBOSE)) {  
  
        logWithTimeAndKey("Started new load", startTime, key);  
  
    }  
  
    return new LoadStatus(cb, engineJob);  
  
}  
  
...  
}
```

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<?>>>  
    activeResources;  
  
    ...  
  
    private EngineResource<?> loadFromCache(Key key, boolean  
    isMemoryCacheable) {  
  
        if (!isMemoryCacheable) {  
  
            return null;  
        }  
  
        EngineResource<?> cached =  
        getEngineResourceFromCache(key);
```

```
    if (cached != null) {

        cached.acquire();

        activeResources.put(key, new ResourceWeakReference(key,
cached, 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 isMemoryCacheable) {  
  
    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,  
isCacheable);  
  
        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, resource, 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  
resource");  
        }  
        if (!Looper.getMainLooper().equals(Looper.myLooper())) {  
            throw new IllegalThreadStateException("Must call acquire  
on the 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
the 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<?>>>  
    activeResources;  
  
    ...  
  
    @Override  
    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：

```
1 Glide.with(this)
2     .load(url)
3     .diskCacheStrategy(DiskCacheStrategy.NONE)
4     .into(imageView);
```

复制

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：

```
1 private Resource<?> decode() throws Exception {
2     if (isDecodingFromCache()) {
3         return decodeFromCache();
4     } else {
5         return decodeFromSource();
6     }
7 }
```

复制

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：

```
1 private Resource<?> decodeFromCache() throws Exception {
2     Resource<?> result = null;
3     try {
4         result = decodeJob.decodeResultFromCache();
5     } catch (Exception e) {
6         if (Log.isLoggable(TAG, Log.DEBUG)) {
7             Log.d(TAG, "Exception decoding result from cache: " + e);
8         }
9     }
10    if (result == null) {
11        result = decodeJob.decodeSourceFromCache();
12    }
13    return result;
14 }
```

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：

```
1 public Key getOriginalKey() {  
2     if (originalKey == null) {  
3         originalKey = new OriginalKey(id, signature);  
4     }  
5     return originalKey;  
6 }
```

复制

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。

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：

```
1 public Resource<Z> decodeFromSource() throws Exception {  
2     Resource<T> decoded = decodeSource();  
3     return transformEncodeAndTranscode(decoded);  
4 }
```

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, height);

    }

    return decoded;

}

private Resource<T> cacheAndDecodeSourceData(A data) throws
IOException {

    long startTime = LogTime.getLogTime();

    SourceWriter<A> writer = new
SourceWriter<A>(loadProvider.getSourceEncoder(), data);

    diskCacheProvider.getDiskCache().put(resultKey.getOriginalKey(),
writer);

    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。而在的方法中同样调用了

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>>(loadProvider.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：

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>  
        transformation, ResourceTranscoder<Z, R> transcoder,  
        Priority priority, boolean isMemoryCacheable,
```

```
DiskCacheStrategy diskCacheStrategy, ResourceCallback cb) {  
    Util.assertMainThread();  
    long startTime = LogTime.getLogTime();  
  
    final String id = fetcher.getId();  
    EngineKey key = keyFactory.buildKey(id, signature, width, height,  
    loadProvider.getCacheDecoder(),  
    loadProvider.getSourceDecoder(), transformation,  
    loadProvider.getEncoder(),  
    transcoder, loadProvider.getSourceEncoder());  
  
    ...  
}  
  
...  
}
```

来看一下第 11 行，刚才已经说过了，这个 id 其实就是图片的 url 地址。那么，这里是通过调用 fetcher.getId()方法来获取的图片 url 地址，而我们在上一篇文章中已经知道了，fetcher 就是 HttpURLConnection 的实例，我们就来看一下它的 getId()方法的源码吧，如下所示：

```
public class HttpURLConnection implements DataFetcher<InputStream> {
```

```
private final GlideUrl glideUrl;  
  
...  
  
public HttpUrlFetcher(GlideUrl glideUrl) {  
    this(glideUrl, DEFAULT_CONNECTION_FACTORY);  
}  
  
  
HttpUrlFetcher(GlideUrl      glideUrl,      HttpURLConnectionFactory  
connectionFactory) {  
    this.glideUrl = glideUrl;  
    this.connectionFactory = connectionFactory;  
}  
  
  
@Override  
public String getId() {  
    return glideUrl.getCacheKey();  
}  
  
...  
}
```

可以看到，`getId()`方法中又调用了`GlideUrl`的`getCacheKey()`方法。那么这个`GlideUrl`对象是从哪里来的呢？其实就是在`load()`方法中传入的图片url地址，然后`Glide`在内部把这个url地址包装成了一个`GlideUrl`对象。

很明显，接下来我们就要看一下`GlideUrl`的`getCacheKey()`方法的源码了，如下所示：

```
public class GlideUrl {
```

```
    private final URL url;
```

```
    private final String urlString;
```

```
    ...
```

```
    public GlideUrl(URL url) {
```

```
        this(url, Headers.DEFAULT);
```

```
    }
```

```
    public GlideUrl(String url) {
```

```
        this(url, Headers.DEFAULT);
```

```
    }
```

```
    public GlideUrl(URL url, Headers headers) {
```

```
        ...
```

```
this.url = url;  
stringUrl = null;  
}  
  
public GlideUrl(String url, Headers headers) {  
    ...  
    this.stringUrl = url;  
    this.url = null;  
}  
  
public String getCacheKey() {  
    return stringUrl != null ? stringUrl : url.toString();  
}  
}  
...  
}  
  
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  
构造函数接收两种类型的参数，一种是 url 字符串，一种是 URL  
getCacheKey()方法中的判断逻辑非常简单，如果传入的是 url  
直接返回这个字符串本身，如果传入的是 URL 对象，那么就返  
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,  
nextAndIndex + 1);  
  
    } else {  
  
        tokenParam = mUrl.substring(tokenKeyIndex);  
  
    }  
  
}  
  
return tokenParam;  
  
}  
  
}
```

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：

```
1 Glide.with(this)
2     .load(new MyGlideUrl(url))
3     .into(imageView);
```

复制

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。

Android 图片加载框架最全解析（四），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：

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

复制

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();
    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：

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

```
        this.resource = resource;

        resource.setLoopCount(maxLoopCount);

        resource.start();

    }

    @Override

    protected void setResource(GlideDrawable resource) {

        view.setImageDrawable(resource);

    }

    ...
}
```

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。

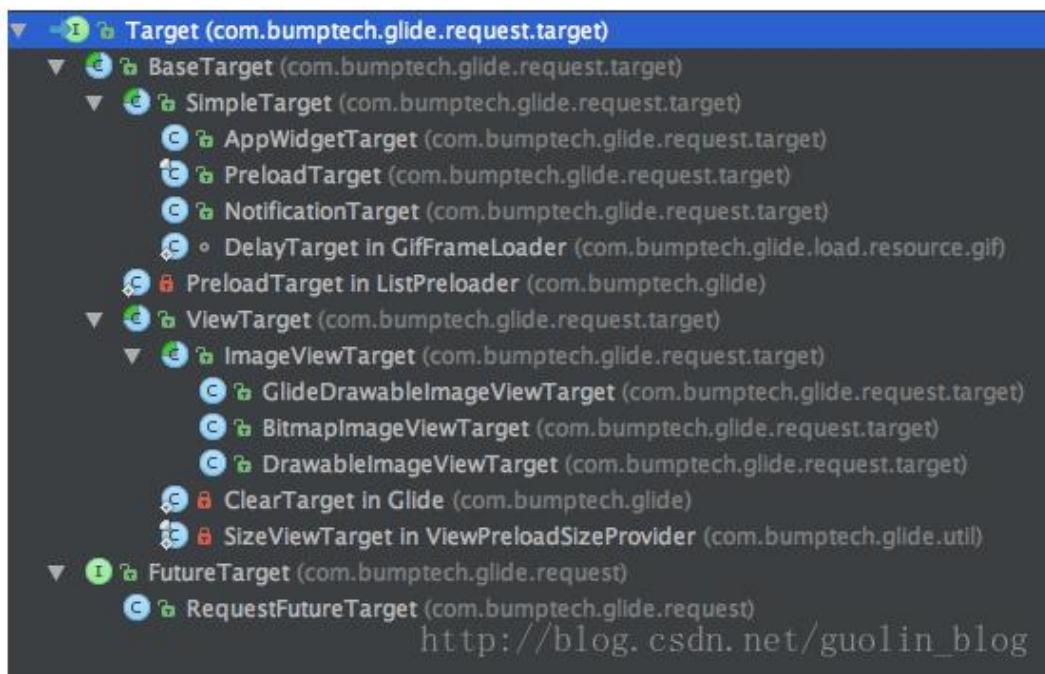
好的，那么原理就先分析到这儿，接下来我们就来看一下在回调和监听方面还有哪些知识是可以扩展的。

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。

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<GlideDrawable>() {
    @Override
    public void onResourceReady(GlideDrawable resource,
        GlideAnimation glideAnimation) {
        imageView.setImageDrawable(resource);
    }
};

public void loadImage(View view) {
    String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473_1920x10
80.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。



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  
glideAnimation) {  
  
        imageView.setImageBitmap(resource);  
  
    }  
  
};  
  
  
public void loadImage(View view) {  
  
    String url =  
"http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473_1920x10  
80.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,  
                ...  
            ) {  
                ...  
            }  
        };  
    }  
}
```

```
GlideAnimation 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

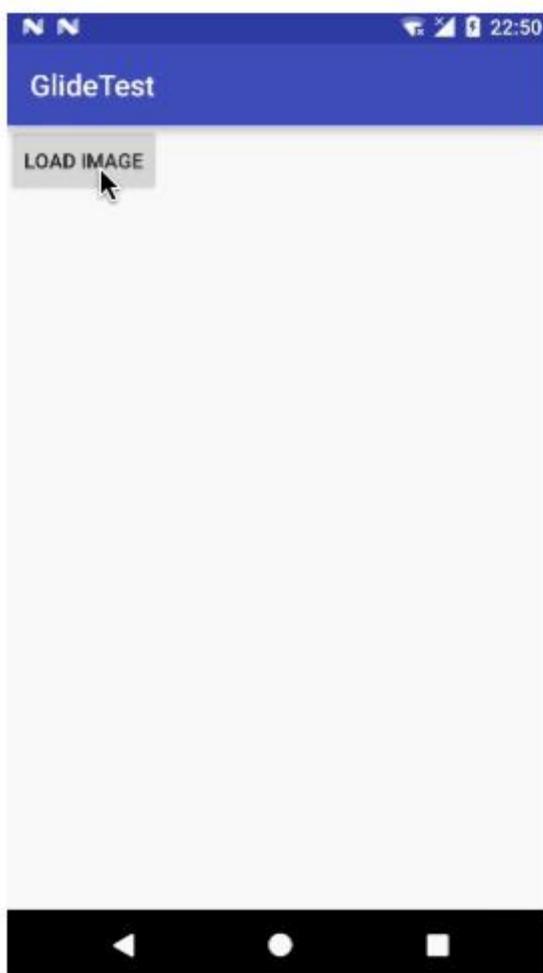
`setImageAsBackground()`方法来将加载出来的图片作为 `MyLayout` 布局的背景图。

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-CN7336795473_1920x10  
            80.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



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，不想再等待漫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？

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：

```
1 Glide.with(this)
2     .load(url)
3     .diskCacheStrategy(DiskCacheStrategy.SOURCE)
4     .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：

```
1 Glide.with(this)
2     .load(url)
3     .diskCacheStrategy(DiskCacheStrategy.SOURCE)
4     .into(imageView);
```

复用

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：

```
public class GenericRequestBuilder<ModelType, DataType,  
ResourceType, TranscodeType> 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);  
    }  
  
    ...  
}
```

正如刚才所说，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) {  
        return new PreloadTarget<Z>(width, height);  
    }  
  
    private PreloadTarget(int width, int height) {  
        super(width, height);  
    }  
  
    @Override  
    public void onResourceReady(Z resource, GlideAnimation<? super  
        Z> glideAnimation) {  
        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。

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-CN7336795473_1920x10
80.jpg";

    final Context context = getApplicationContext();

    FutureTarget<File> target = Glide.with(context)

        .load(url)

        .downloadOnly(Targ
et.SIZE_ORIGINAL, 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。

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。

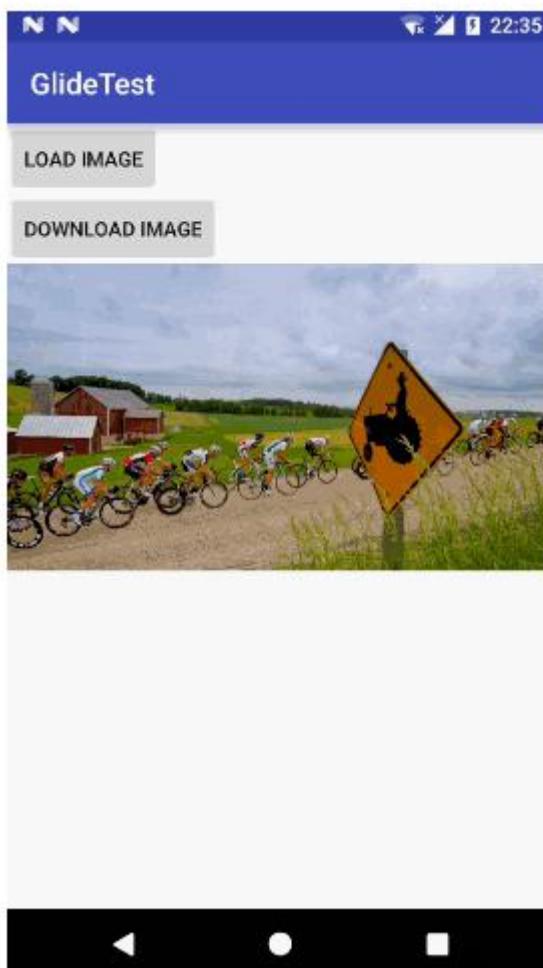
之后我们可以使用如下代码去加载这张图片，图片就会立即显示出来，而不用再去网络上请求了：

```
public void loadImage(View view) {  
    String url =  
        "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473_1920x10  
80.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  
DrawableRequestBuilder<ModelType>  
implements DownloadOptions {  
    ...  
  
    public FutureTarget<File> downloadOnly(int width, int height) {  
        return getDownloadOnlyRequest().downloadOnly(width,  
height);  
    }  
  
    private GenericTranscodeRequest<ModelType, InputStream, File>  
getDownloadOnlyRequest() {  
        return optionsApplier.apply(new  
GenericTranscodeRequest<ModelType, 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 < -> downloadOnly int
int return private
< > getDownloadOnlyRequest
< >
< >
< > new
< >
return new <
>
this
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 < -> int int final
< > new
< >
new @Override public void run if
return
```

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>, Runnable {  
  
    @Override public get throws  
        try return  
            null catch throw new @Override  
    public get long throws  
        return @Override  
    public void getSize @Override  
    public synchronized void onLoadFailed  
        true this this @Override public  
    synchronized void onResourceReady < super >  
        true this this  
    private synchronized doGet throws  
        if  
            if throw new else  
            if throw new else if  
        return if null this 0 else if
```

```
> 0           this       if
throw new      else if     throw new
else if        throw new   return static
public void waitForTimeout long throws
public void 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。

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  
    "DownloadImageTarget" @Override public void onStart @Override  
    public void onStop @Override public void onDestroy @Override public void  
    onLoadStarted @Override public void onLoadFailed @Override public void  
    onResourceReady < super >  
    @Override public void onLoadCleared  
    @Override public void getSize  
    - - - @Override public void  
    setRequest @Override public 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。这些方法大多数是 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：

```
1 public void downloadImage(View view) {  
2     String url = "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473_1920x1080  
3     Glide.with(this)  
4         .load(url)  
5         .downloadOnly(new DownloadImageTarget());  
6 }
```

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。

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  
    "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-CN7336795473_1920x1080.jpg"  
    this      new      <      >  
    @Override public boolean onException  
        <      >      boolean      return false      @Override  
    public boolean onResourceReady  
        <      >      boolean      boolean  
        return false
```

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：

```
1 public class GenericRequestBuilder<ModelType, DataType, ResourceType, TranscodeType> 复制
2
3     private RequestListener<? super ModelType, TranscodeType> requestListener;
4     ...
5
6     public GenericRequestBuilder<ModelType, DataType, ResourceType, TranscodeType> setRequestListener(
7             RequestListener<? super ModelType, TranscodeType> requestListener) {
8         this.requestListener = requestListener;
9         return this;
10    }
11
12    ...
13 }
```

Android 图片加载框架最全解析（五），Glide 强大的图片变换功能

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：

```
:// / _ 1
```

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：

```
<
    : // / _ 1
    :
    :
    >

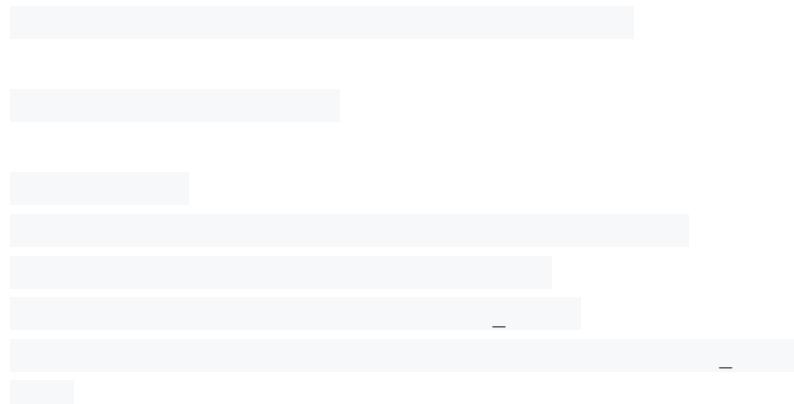
<
    :
    :
    :
    :
    />

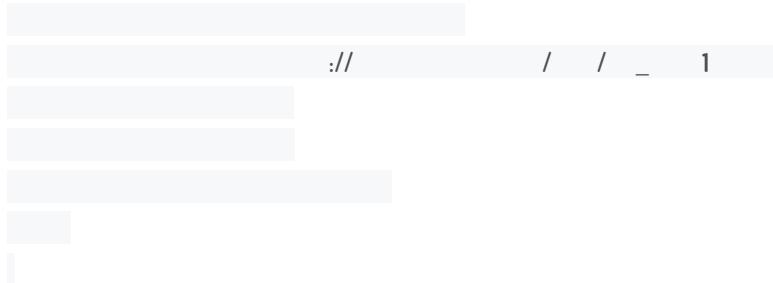
<
    :
    / _ 1
    :
    :
    />
</      >
```

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：





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。



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。

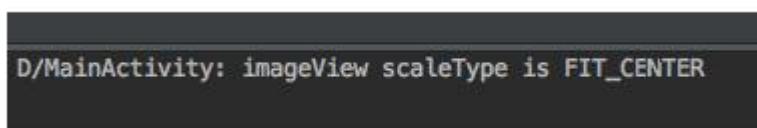
```
public class MainActivity extends AppCompatActivity {  
  
    private static final String TAG = "MainActivity";  
  
    ImageView imageView;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        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：



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](#)。

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 View");
}

if (!isTransformationSet && view.getScaleType() != null) {

    switch (view.getScaleType()) {

        case CENTER_CROP:

            applyCenterCrop();

            break;

        case FIT_CENTER:

        case FIT_START:

        case FIT_END:

            applyFitCenter();

            break;

        //$$CASES-OMITTED$$

        default:

            // Do nothing.

    }
}

return into(glide.buildImageViewTarget(view, transcodeClass));
}
```

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() 方法其实就是在 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：

```
1 Glide.with(this)
2     .load(url)
3     .dontTransform()
4     .into(imageView);
```

复制

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



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：

```
1 Glide.with(this)
2     .load(url)
3     .override(Target.SIZE_ORIGINAL, Target.SIZE_ORIGINAL)
4     .into(imageView);
```

复制

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。

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()  
    .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：

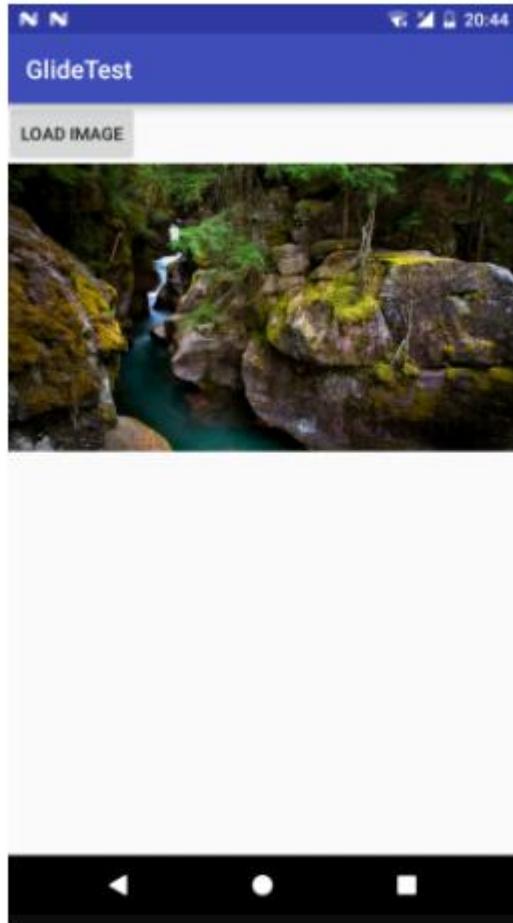
```
public class DrawableRequestBuilder<ModelType>  
    extends GenericRequestBuilder<ModelType,  
    ImageVideoWrapper, GifBitmapWrapper, GlideDrawable>
```

```
    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。

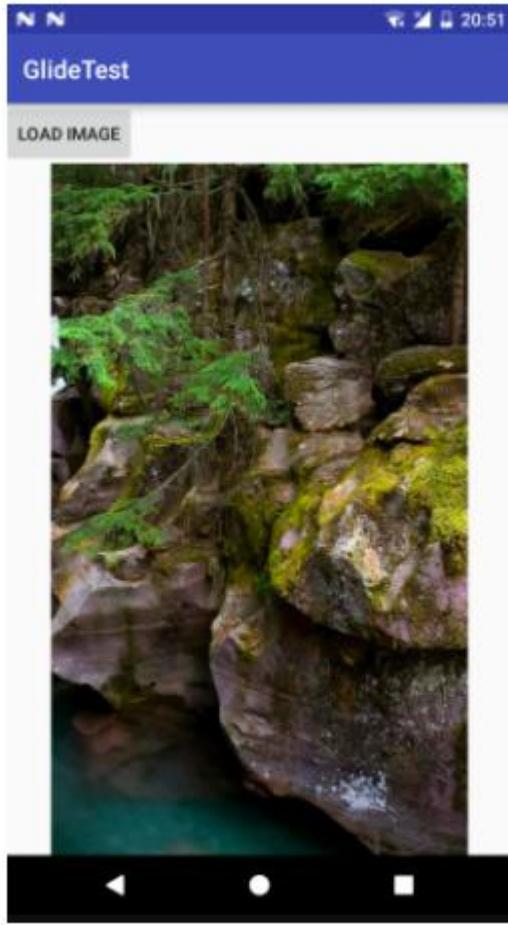
为了让效果更加明显，这里我就不使用百度首页的 Logo 图了，而是换成必应首页的一张美图。在不应用任何图片变换的情况下，使用 Glide 加载必应这张图片效果如下所示。



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);
```

重新运行一下程序并点击加载图片按钮，效果如下图所示



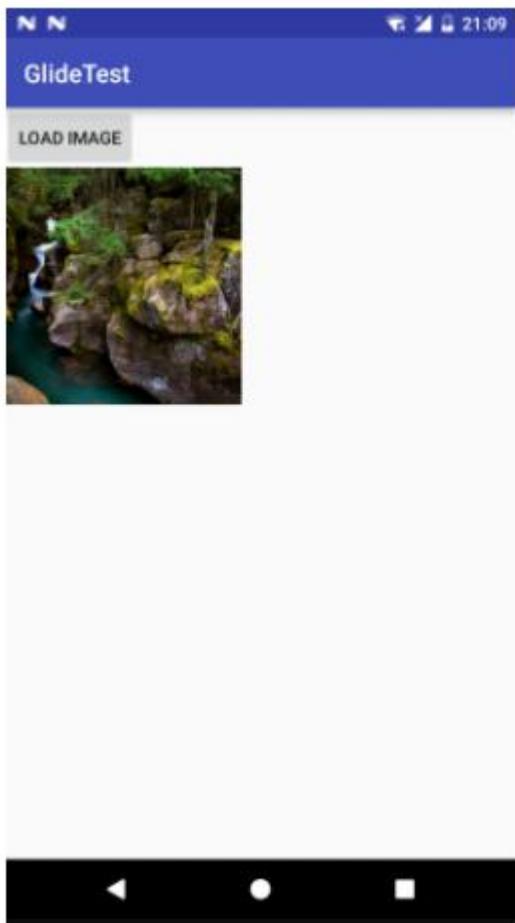
可以看到，现在展示的图片是对原图的中心区域进行裁剪后得到的图片。

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);
```

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 {  
  
    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,
```

```
toTransform, outWidth, outHeight);

    if (toReuse != null && toReuse != transformed
&& !pool.put(toReuse)) {

        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，这是原始图片的 Bitmap 对象，我们就是要对它来进行

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  
        width, int height) {  
        if (toCrop == null) {  
            return null;  
        } else if (toCrop.getWidth() == width && toCrop.getHeight() ==  
            height) {  
            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(toCrop));  
  
}  
  
  
  
// 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.glideCircleCrop.CircleCrop";  
}  
  
@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 接口就可以了。不过这个就非常复杂了，不在我今天的讨论范围。

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.glide.test.CircleCrop";  
    }
```

```
    @Override  
  
    protected Bitmap transform(BitmapPool pool, Bitmap toTransform,  
        int outWidth, int outHeight) {  
  
        int      diameter      =      Math.min(toTransform.getWidth(),  
            toTransform.getHeight());  
  
        final  Bitmap  toReuse   =   pool.get(outWidth,  outHeight,  
            Bitmap.Config.ARGB_8888);  
  
        final Bitmap result;  
  
        if (toReuse != null) {  
  
            result = toReuse;  
  
        } else {  
  
            result      =      Bitmap.createBitmap(diameter,      diameter,  
                Bitmap.Config.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

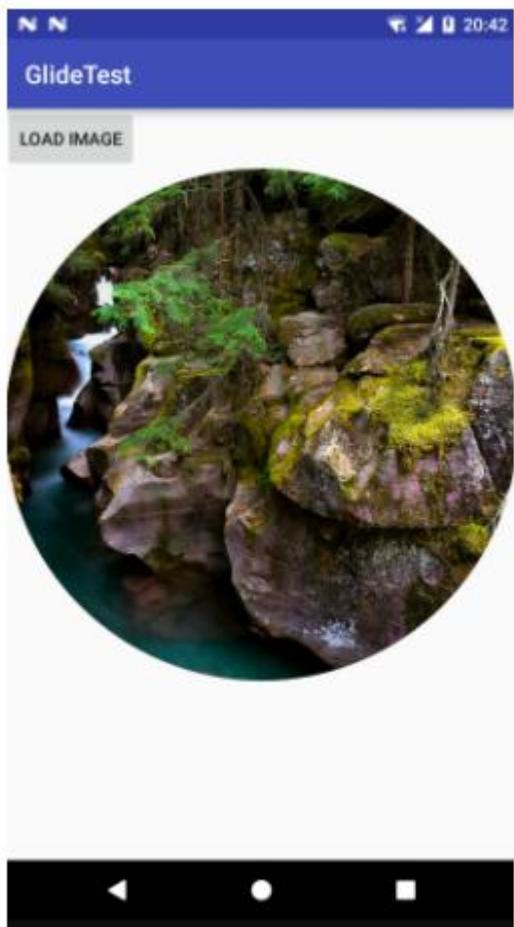
一个 Bitmap 对象来进行重用。如果没有可重用的 Bitmap 对象的话就创建一个。

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)  
.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：

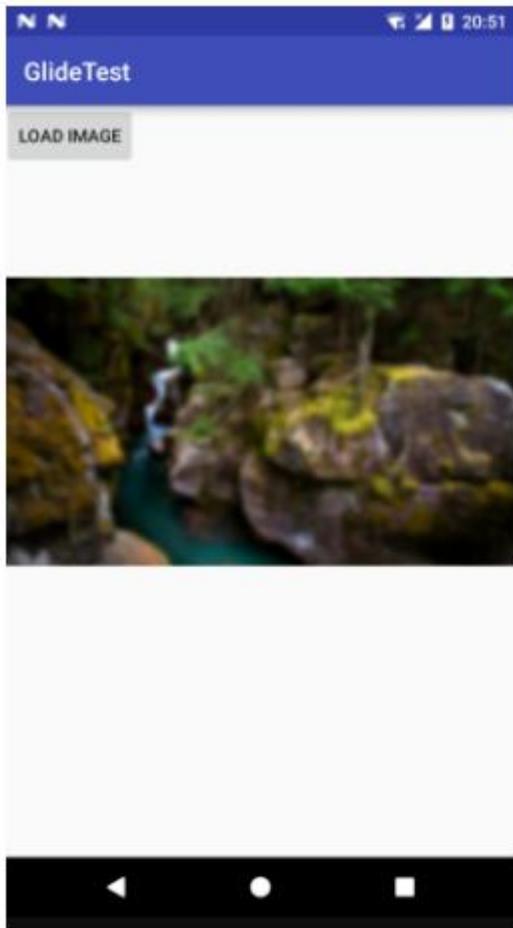
```
1 dependencies {  
2     compile 'jp.wasabeef:glide-transformations:2.0.2'  
3 }
```

复制

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：

```
1 Glide.with(this)  
2     .load(url)  
3     .bitmapTransform(new BlurTransformation(this))  
4     .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。



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。



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。



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。

Android 图片加载框架最全解析（六），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"
        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.getApplicationContext();  
                    List<GlideModule> modules = new  
ManifestParser(applicationContext).parse();  
                    GlideBuilder builder = new GlideBuilder(applicationContext);  
                    for (GlideModule module : modules) {  
                        module.applyOptions(applicationContext, builder);  
                    }  
                    glide = builder.createGlide();  
                    for (GlideModule module : modules) {  
                        module.registerComponents(applicationContext, glide);  
                    }  
                }  
            }  
        }  
        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().availableProcessors());
            sourceService = new FifoPriorityThreadPoolExecutor(cores);
        }
        if (diskCacheService == null) {
            diskCacheService = new FifoPriorityThreadPoolExecutor(1);
        }
        MemorySizeCalculator calculator = new MemorySizeCalculator(context);
```

```
    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.getMemoryCacheSize());
    }
    if (diskCacheFactory == null) {
        diskCacheFactory = new InternalCacheDiskCacheFactory(context);
    }
    if (engine == null) {
        engine = new Engine(memoryCache, diskCacheFactory, diskCacheService,
sourceService);
    }
    if (decodeFormat == null) {
        decodeFormat = DecodeFormat.DEFAULT;
    }
    return new Glide(engine, memoryCache, bitmapPool, context, decodeFormat);
}
}
```

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：

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 {

    @Override
    public void applyOptions(Context context, GlideBuilder builder) {
        builder.setDiskCache(new ExternalCacheDiskCacheFactory(context));
    }

    @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) {
        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



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 `/data/data/包名/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。



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。



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，我们都已经知道 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孰优孰劣，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。

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  
            FileDescriptorFileLoader.Factory());  
  
        register(File.class, InputStream.class, new  
            StreamFileLoader.Factory());
```

```
    register(int.class,          ParcelFileDescriptor.class,      new
FileDescriptorResourceLoader.Factory());

    register(int.class,          InputStream.class,            new
StreamResourceLoader.Factory());

    register(Integer.class,      ParcelFileDescriptor.class,  new
FileDescriptorResourceLoader.Factory());

    register(Integer.class,      InputStream.class,            new
StreamResourceLoader.Factory());

    register(String.class,       ParcelFileDescriptor.class, new
FileDescriptorStringLoader.Factory());

    register(String.class,       InputStream.class,            new
StreamStringLoader.Factory());

    register(Uri.class,          ParcelFileDescriptor.class, new
FileDescriptorUriLoader.Factory());

    register(Uri.class,          InputStream.class,            new
StreamUriLoader.Factory());

    register(URL.class,          InputStream.class,            new
StreamUrlLoader.Factory());

    register(GlideUrl.class,     InputStream.class,            new
HttpUrlGlideUrlLoader.Factory());

    register(byte[].class,        InputStream.class,            new
StreamByteArrayLoader.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.Factory());
```

这句代码就表示，我们可以使用

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。

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,  
InputStream> {  
  
    private final ModelCache<GlideUrl, GlideUrl> modelCache;  
  
    public static class Factory implements ModelLoaderFactory<GlideUrl,  
InputStream> {  
        private final ModelCache<GlideUrl, GlideUrl> modelCache =  
new ModelCache<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> modelCache) {
        this.modelCache = modelCache;
    }

    @Override
    public DataFetcher<InputStream> getResourceFetcher(GlideUrl model, int 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_FACTORY =
            new DefaultHttpUrlConnectionFactory();
```

```
private final GlideUrl glideUrl;  
private final HttpURLConnectionFactory connectionFactory;  
  
  
private HttpURLConnection urlConnection;  
private InputStream stream;  
private volatile boolean isCancelled;  
  
  
  
public HttpURLConnectionFactory(GlideUrl glideUrl) {  
    this(glideUrl, DEFAULT_CONNECTION_FACTORY);  
}  
  
  
  
HttpURLConnectionFactory(glideUrl, connectionFactory) {  
    this.glideUrl = glideUrl;  
    this.connectionFactory = connectionFactory;  
}  
  
  
  
@Override  
public InputStream loadData(Priority priority) throws Exception {  
    return loadDataWithRedirects(glideUrl.toURL(), 0, null,  
        glideUrl.getHeaders());
```

```
}

private InputStream loadDataWithRedirects(URL url, int redirects,
URL lastUrl, Map<String, String> headers)
throws IOException {
if (redirects >= MAXIMUM_REDIRECTS) {
throw new IOException("Too many (> " +
MAXIMUM_REDIRECTS + ") redirects!");
} else {
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(),
headerEntry.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 redirect urlString = urlConnection.getHeaderField("Location");

    if (TextUtils.isEmpty	redirect urlString)) {

        throw new IOException("Received empty or null redirect url");

    }

    URL redirectUrl = new URL(url, redirect urlString);

    return loadDataWithRedirects(redirectUrl, redirects + 1, url, headers);

} else {

    if (statusCode == -1) {
```



```
@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  
HttpURLConnectionFactory {  
    @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 HttpURLConnection 的代码来写，并且把 HTTP 的通讯组件替换成 OkHttpClient 就可以了。

现在新建一个 OkHttpClientFetcher 类，并且同样实现 DataFetcher<InputStream> 接口，代码如下所示：

```
public class OkHttpClientFetcher 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().entrySet()) {  
  
        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 `http://: 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,  
InputStream> {  
  
    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;  
        }  
    }  
}
```

```
        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, int 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  
OkHttpGlideUrlLoader.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络请求。

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。



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'  
}
```

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。

Android 图片加载框架最全解析(七) , 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。

我们都应该知道 , 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'  
}
```

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().entrySet()) {  
  
        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,  
InputStream> {
```

```
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, int 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  
OkHttpGlideUrlLoader.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.glideprogressstest.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) {  
  
    }  
  
    @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  
        OkHttpGlideUrlLoader.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 {

    static final Map<String, ProgressListener> LISTENER_MAP = new
    HashMap<>();

    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，而在这种情况下，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：

```
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 void writeTo(BufferedSink sink) throws IOException {  
        responseBody.writeTo(sink);  
        if (listener != null) {  
            ProgressInterceptor.setProgress(listener, url, bufferedSource);  
        }  
    }  
}
```

```
}
```

```
@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  
ProgressResponseBody(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>
```

很简单，这里使用了一个 Button 按钮来加载图片，使用了一个 ImageView 来展示图片。

然后修改 MainActivity 中的代码，如下所示

```
public class MainActivity extends AppCompatActivity {
```

```
String url = "http://guolin.tech/book.png";  
  
ImageView image;  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    image = (ImageView) findViewById(R.id.image);  
}  
  
public void loadImage(View view) {  
    Glide.with(this)  
        .load(url)  
        .diskCacheStrategy(DiskCacheStrategy.NONE)  
        .override(Target.SIZE_ORIGINAL, Target.SIZE_ORIGINAL)  
        .into(image);  
}  
}
```

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

```
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://40105.65qn.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_HORIZONTAL);
    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,  
GlideAnimation<? super GlideDrawable> animation) {  
  
        super.onResourceReady(resource, animation);  
  
        progressDialog.dismiss();  
  
        ProgressInterceptor.removeListener(url);  
  
    }  
  
});  
}
```

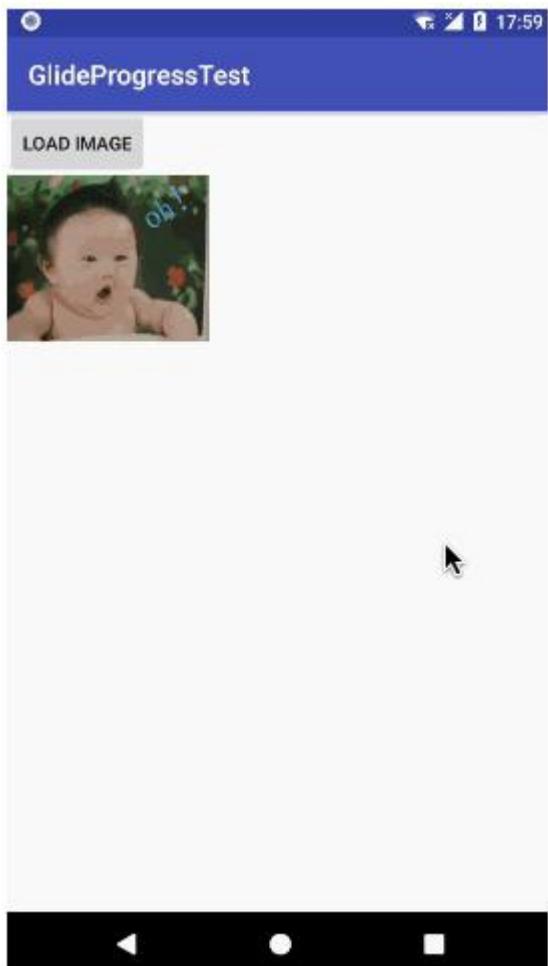
代码并不复杂。这里我们新增了一个 ProgressDialog 用来显示下载进度，然后在 loadImage()方法中，调用了 ProgressInterceptor.addListener()方法来去注册一个下载监听器，并在 onProgress()回调方法中更新当前的下载进度。

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。



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。



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

Android 图片加载框架最全解析(八),带你全面了解 Glide 4 的用法

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'
```

}

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) {
        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：

```
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，不会再在网络下载了，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：

```
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：



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);  
  
}  
  
}
```

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。

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，只要在它还没有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。

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。

[指定加载格式](#)

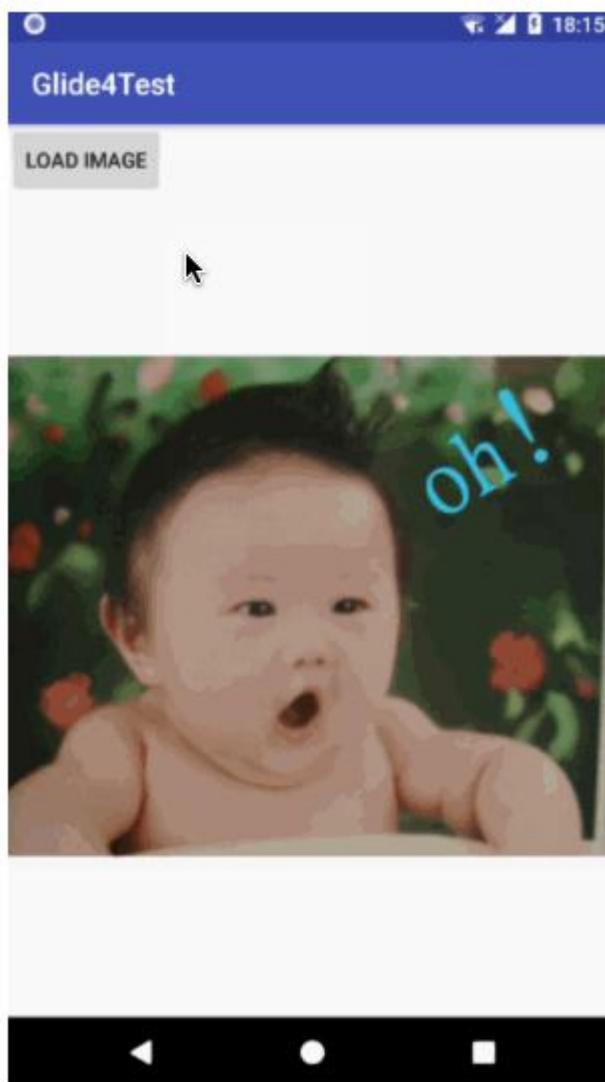
我们都知道，Glide 其中一个非常亮眼的功能就是可以加载 GIF 图片，而同样作为非常出色的图片加载框架的 Picasso 是不支持这个功能的。

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



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

我们都已经知道 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：

```
SimpleTarget<Drawable> simpleTarget = new SimpleTarget<Drawable>(){

    @Override
    public void onResourceReady(Drawable resource, Transition<? super
        Drawable> 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，不想再等待漫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：

```
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() {  
    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 ApplicationContext，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 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



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 model, 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：

```
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);
```

重新运行一下程序并点击加载图片按钮，效果如下图所示



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。

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 GrayscaleTransformation());  
  
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。



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。

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

```
@GlideModule  
public class MyAppGlideModule extends AppGlideModule {  
  
    @Override  
    public void applyOptions(Context context, GlideBuilder builder) {  
  
    }  
  
    @Override  
    public void registerComponents(Context context, Glide glide,  
        Registry 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 Glide 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。

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。

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：

```
@GlideExtension  
public 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()  
    .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。

二十三、Android 组件化与插件化

1、为什么要用组件化？

组件化的目标是要告别 APP 的臃肿 ,APP 的业务迭代不应该以牺牲 APP 的臃肿为代价。

各个业务组件相对独立，业务组件在组件模式下可以独立运行，自成一个 APP，而在集成模式下可以作为一个被依赖的 aar 库文件存在 集成进一个完整的 APP 当中

2、组件之间如何通信？

父组件传递数据给子组件
可以通过 `props` 属性来实现

- 父组件:

```
<parent>

    <child :child-msg="msg"></child>//这里必须要用 - 代替驼峰</parent>

data(){
    return {
        msg: [1,2,3]
    };
}
```

- 子组件

```
//方式一

props: ['childMsg']//方式二

props: {
    childMsg: Array //这样可以指定传入的类型，如果类型不对，会警告}//方式三
```

```
props: {

    childMsg: {

        type: Array,

        default: [0,0,0] //这样可以指定默认的值

    }
}
```

子组件向父组件通信

如果子组件想要改变数据呢？这在 vue 中是不允许的，因为 vue 只允许单向数据传递，这时候我们可以通过触发事件来通知父组件改变数据，从而达到改变子组件数据的目的

- 组件

```
<template>

<div @click="up"></div></template>

methods: {

    up() {

        this.$emit('upup', 'hehe') //主动触发 upup 方法，'hehe'为向父组件传递的数据

    }
}
```

- 父组件

```
<div>

    <child @upup="change" :msg="msg"></child> //监听子组件触发的 upup 事件，然后调用 change 方法</div>

methods: {

    change(msg) {

```

```
    this.msg = msg;  
}  
}
```

非父子组件通信

如果 2 个组件不是父子组件那么如何通信呢？这时可以通过 eventHub 来实现通信。

所谓 eventHub 就是创建一个事件中心，相当于中转站，可以用它来传递事件和接收事件

```
let Hub = new Vue(); // 创建事件中心
```

- 组件 1 触发事件

```
<div @click="eve"></div>  
  
methods: {  
  
  eve() {  
  
    Hub.$emit('change', 'hehe'); // Hub 触发事件  
  
  }  
}
```

- 组件 2 接收事件

```
<div></div>created() {  
  
  Hub.$on('change', () => { // Hub 接收事件  
  
    this.msg = 'hehe';  
  
  });  
}
```

这样就实现了非父子组件之间的通信了。原理就是把 Hub 当作一个中转站

3、组件之间如何跳转？

三种类型

1) 请求转发:(forward)

语法:

```
request 对 象 .getRequestDispatcher(String  
path).forward(request,response);
```

请求转发的特点:

- 1): 浏览器地址栏路径没变, 依然是 AServlet 的资源名称.
- 2): 只发送了一个请求.
- 3): 共享同一个请求, 在请求中共享数据.
- 4): 最终的响应输出由 BServlet 来决定.
- 5): 只能访问当前应用中的资源, 不能跨域跳转.
- 6): 可以访问 WEB-INF 中的资源

代码结构

```
▸ _01_forward  
  ▸ ForwardServlet.java  
  ▸ GoalServlet.java  
▸ _02_redirect  
  ▸ GoalServlet.java  
  ▸ RedirectServlet.java
```

代码演示

```
/**  
 * web 组件之间的跳转  
 * ① 请求转发(forward)
```

```
* @author AfricaYoung  
* @Date 2016-08-15  
*/  
  
@WebServlet("/forward/a")  
public class ForwardServlet extends HttpServlet{  
  
    private static final long serialVersionUID = 1L;  
  
    protected void service(HttpServletRequest req, HttpServletResponse resp)  
        throws ServletException, IOException {  
        //此处测试跳转后值可否有传递  
        String name = req.getParameter("name");  
        System.out.println("1"+name);  
        //请求转发(forward)  
        req.getRequestDispatcher("/forward/b").forward(req, resp);  
    }  
}  
  
/**  
 * web 组件之间的跳转
```

```
* ①请求转发(forward)

* @author AfricaYoung

* @Date 2016-08-15

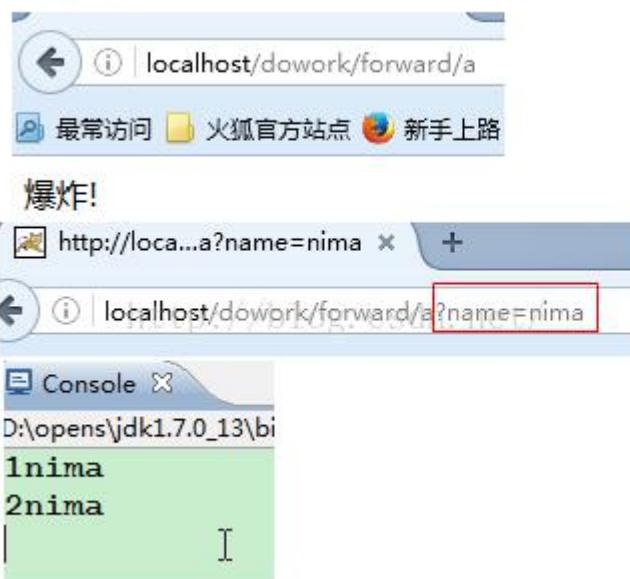
*/
@WebServlet("/forward/b")
public class GoalServlet extends HttpServlet{

    private static final long serialVersionUID = 1L;

    protected void service(HttpServletRequest req, HttpServletResponse
resp)
        throws ServletException, IOException {
        //接收数据
        String name = req.getParameter("name");
        System.out.println("2"+name);
        //设置响应格式
        resp.setContentType("text/html;charset=utf-8");
        //获取打印流
        PrintWriter out = resp.getWriter();
        out.print("爆炸!");
        //关闭流
        out.close();
    }
}
```

```
    }  
  
}
```

演示效果:



2):URL 重定向:(redirect)

AServlet 操作完毕之后,重定向到 BServlet,继续完成余下的功能.

语法:

```
response 对象.sendRedirect(String path);
```

参数:path,表示目标资源名称.

URL 重定向的特点:

- 1):浏览器地址栏路径发生变化,变成 Servlet2 的资源名称.
- 2):只发送了两个请求.
- 3):因为是不同的请求,所以不能共享请求中的数据.
- 4):最终的响应输出由 Servlet2 来决定.

5):可以跨域访问资源.

6):不可以访问 WEB-INF 中的资源.

代码演示:

```
/**  
 * web 组件之间的跳转  
 * ②URL 重定向(redirect)  
 * @author AfricaYoung  
 * @Date 2016-08-15  
 */  
  
@WebServlet("/redirect/a")  
public class RedirectServlet extends HttpServlet{  
  
  
  
    private static final long serialVersionUID = 1L;  
  
  
  
    protected void service(HttpServletRequest req, HttpServletResponse  
    resp)  
        throws ServletException, IOException {  
            //获取 http 传入的值  
            String name = req.getParameter("name");  
            System.out.println("1 "+name);  
            //URL 重定向  
        }  
}
```

```
        resp.sendRedirect("/dowork/redirect/b");

    }

}

/**



 * web 组件之间的跳转 ②URL 重定向(redirect)

 *

 * @author AfricaYoung

 * @Date 2016-08-15

 */

@WebServlet("/redirect/b")

public class GoalServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    protected void service(HttpServletRequest req, HttpServletResponse resp)

        throws ServletException, IOException {

        //测试下看是否能获取到值

        String name = req.getParameter("name");

        System.out.println("2"+name);

        //设置响应格式

        resp.setContentType("text/html;charset=utf-8");

        //获取打印流
```

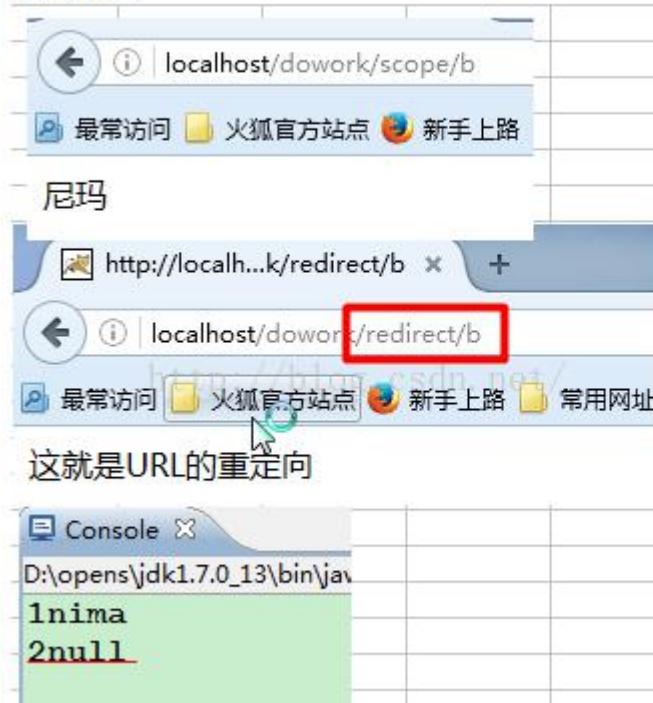
```
PrintWriter out = resp.getWriter();

out.print("这就是 URL 的重定向");

}

}
```

演示效果:



请求转发和 URL 重定向的选择?

- 1:若需要共享请求中的数据,只能使用请求转发.
- 2:若需要访问 WEB-INF 中的资源,只能使用请求转发.
- 3:若需要跨域访问,只能使用 URL 重定向.
- 4:请求转发可能造成表单的重复提交问题.
- 5:其他时候,任选.

3):请求包含:(**include**):到 jsp 说