2020 阿里钉钉、美团、今日头条、腾讯、Bilibili、京东 Android 面试真题解析

知识点总结

Android 基本知识点

1 基本概念

1.1 Android 类加载器

我们知道不管是插件化还是组件化，都是基于系统的 ClassLoader 来设计的。只不过 Android 平台上虚拟机运行的是 Dex 字节码,一种对 class 文件优化的产物,传统 Class 文件是一个 Java 源码文件会生成一个.class 文件，而 Android 是把所有 Class 文件进行合并，优化，然后生成一个最终的 class.dex,目的是把不同 class 文件重复的东西只需保留一份,如果我们的 Android 应用不进行分 dex 处理,最后一个应用的 apk 只会有一个 dex 文件。

Android 中常用的有两种类加载器，DexClassLoader 和 PathClassLoader，它们都继承于 BaseDexClassLoader。区别在于调用父类构造器时，DexClassLoader 多传了一个 optimizedDirectory 参数，这个目录必须是内部存储路径，用来缓存系统创建的 Dex 文件。而 PathClassLoader 该参数为 null，只能加载内部存储目录的 Dex 文件。所以我们可以用 DexClassLoader 去加载外部的 apk

```
public class DexClassLoader extends BaseDexClassLoader {
    public DexClassLoader(String dexPath, String optimizedDirectory, String librarySearchPath, ClassLoader parent) {
        super((String)null, (File)null, (String)null, (ClassLoader)null);
        throw new RuntimeException("Stub!");
    }
}
```

1.2 Service

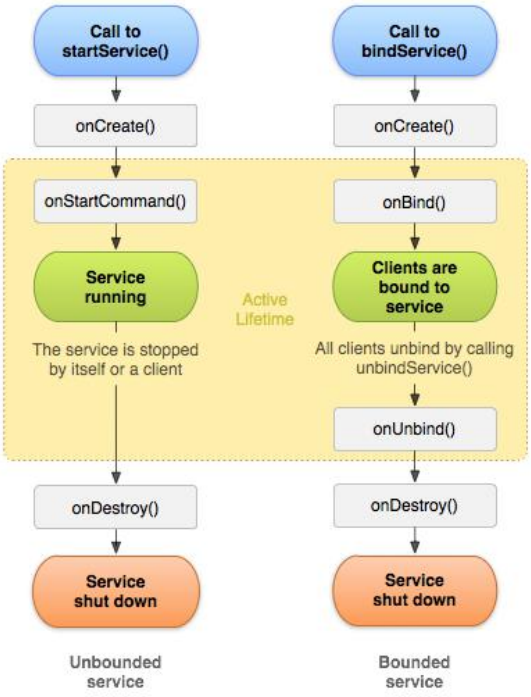
- Service 是在 main Thread 中执行，Service 中不能执行耗时操作（网络请求，拷贝数据库，大文件）。

- 可以在 Androidmanifest.xml 中设置 Service 所在的进程,让 Service 在另外的进程中执行。
- Service 执行的操作最多是 20s，BroadcastReceiver 是 10s，Activity 是 5s。
- Activity 通过 bindService（Intent，ServiceConnection，flag）与 Service 绑定。
- Activity 可以通过 startService 和 bindService 启动 Service。

1.2.1 IntentService

IntentService 是一个抽象类，继承自 Service，内部存在一个 ServiceHandler（Handler）和 HandlerThread（Thread）。IntentService 是处理异步请求的一个类，在 IntentService 中有一个工作线程（HandlerThread）来处理耗时操作，启动 IntentService 的方式和普通的一样，不过当执行完任务之后，IntentService 会自动停止。另外可以多次启动 IntentService，每一个耗时操作都会以工作队列的形式在 IntentService 的 onHandleIntent 回调中执行，并且每次执行一个工作线程。它的本质是：**封装了一个 HandlerThread 和 Handler 的异步框架。**

1.2.2 生命周期



1.2.3 startService 生命周期

当我们通过调用了 Context 的 startService 方法后，我们便启动了 Service，通过 startService 方法启动的 Service 会一直无限期地运行下去，只有在外部调用 Context 的 **stopService** 或 Service 内部调用 Service 的 **stopSelf** 方法时，该 Service 才会停止运行并销毁。

- onCreate

onCreate：执行 startService 方法时，如果 Service 没有运行的时候会创建该 Service 并执行 Service 的 onCreate 回调方法;如果 Service 已经处于运行中,那么执行 startService 方法不会执行Service 的onCreate 方法。也就是说如果多次执行了 Context 的 startService 方法启动 Service,Service 方法的 onCreate 方法只会在第一次创建 Service 的时候调用一次，以后均不会再次调用。我们可以在 onCreate 方法中完成一些 Service 初始化相关的操作。

- onStartCommand

onStartCommand：在执行了 startService 方法之后，有可能会调用 Service 的 onCreate 方法，在这之后一定会执行 Service 的 onStartCommand 回调方法。也就是说，**如果多次执行了 Context 的 startService 方法，那么 Service 的 onStartCommand 方法也会相应的多次调用**。onStartCommand 方法很重要，我们在该方法中根据传入的 Intent 参数进行实际的操作，比如会在此处创建一个线程用于下载数据或播放音乐等。

```
public @StartResult int onStartCommand(Intent intent, @StartArgFlags int flags,
int startId) {}
```

当 Android 面临内存匮乏的时候，可能会销毁掉你当前运行的 Service，然后待内存充足的时候可以重新创建 Service，Service 被 Android 系统强制销毁并再次重建的行为依赖于 Service 中 onStartCommand 方法的返回值。我们常用的返回值有三种值,START_NOT_STICKY、START_STICKY 和 START_REDELIVER_INTENT，这三个值都是 Service 中的静态常量。

- START_NOT_STICKY

如果返回 START_NOT_STICKY，表示当 Service 运行的进程被 Android 系统强制杀掉之后，不会重新创建该 Service，当然如果在其被杀掉之后一段时间又调用了 startService，那么该 Service 又将被实例化。那什么情境下返回该值比较恰当呢？如果我们某个 Service 执行的工作被中断几次无关紧要或者对 Android 内存紧张的情况下需要被杀掉且不会立即重新创建这种行为也可接受，那么我们便可将 onStartCommand 的返回值设置为 START_NOT_STICKY。举个例子，某个 Service 需要定时从服务器获取最新数据：通过一个定时器每隔指定的 N 分钟让定时器启动 Service 去获取服务端的最新数据。当执行到 Service 的 onStartCommand 时，在该方法内再规划一个 N 分钟后的定时器用于再次启动该 Service 并开辟一个新的线程去执行网络操作。假设 Service 在从服务器获取最新数据的过程中被 Android 系统强制杀掉，Service 不会再重新创建，这也没关系，因为再过 N 分钟定时器就会再次启动该 Service 并重新获取数据。

- START_STICKY

如果返回 START_STICKY，表示 Service 运行的进程被 Android 系统强制杀掉之后，Android 系统会将该 Service 依然设置为 started 状态(即运行状态),**但是不再保存 onStartCommand 方法传入的 intent 对象，**然后 Android 系统会尝试再次重新创建该 Service，并执行 onStartCommand 回调方法，但是 onStartCommand 回调方法的 Intent 参数为 null，也就是 onStartCommand 方法虽然会执行但是获取不到 intent 信息。如果你的 Service 可以在任意时刻运行或结束都没什么问题，而且不需要 intent 信息，那么就可以在 onStartCommand 方法中返回 START_STICKY，比如一个用来播放背景音乐功能的 Service 就适合返回该值。

- START_REDELIVER_INTENT

如果返回 START_REDELIVER_INTENT，表示 Service 运行的进程被 Android 系统强制杀掉之后，与返回 START_STICKY 的情况类似，Android 系统会将再次重新创建该 Service，并执行 onStartCommand 回调方法，但是不同的是，**Android 系统会再次将 Service 在被杀掉之前最后一次传入 onStartCommand 方法中的 Intent 再次保留下来并再次传入到重新创建后的 Service 的 onStartCommand 方法中，这样我们就能读取到 intent 参数。**只要返回 START_REDELIVER_INTENT，那么 onStartCommand 重的 intent 一定不是 null。如果我们的 Service 需要依赖具体的 Intent 才能运行（需要从 Intent 中读取相关数据信息等），并且在强制销毁后有必要重新创建运行，那么这样的 Service 就适合返回 START_REDELIVER_INTENT。

- onBind

Service 中的 onBind 方法是抽象方法，所以 Service 类本身就是抽象类，也就是 onBind 方法是必须重写的，即使我们用不到。在通过 startService 使用 Service 时，我们在重写 onBind 方法时，只需要将其返回 null 即可。onBind 方法主要是用于给 bindService 方法调用 Service 时才会使用到。

- onDestroy

onDestroy：通过 startService 方法启动的 Service 会无限期运行，只有当调用了 Context 的 stopService 或在 Service 内部调用 stopSelf 方法时，Service 才会停止运行并销毁，在销毁的时候会执行 Service 回调函数。

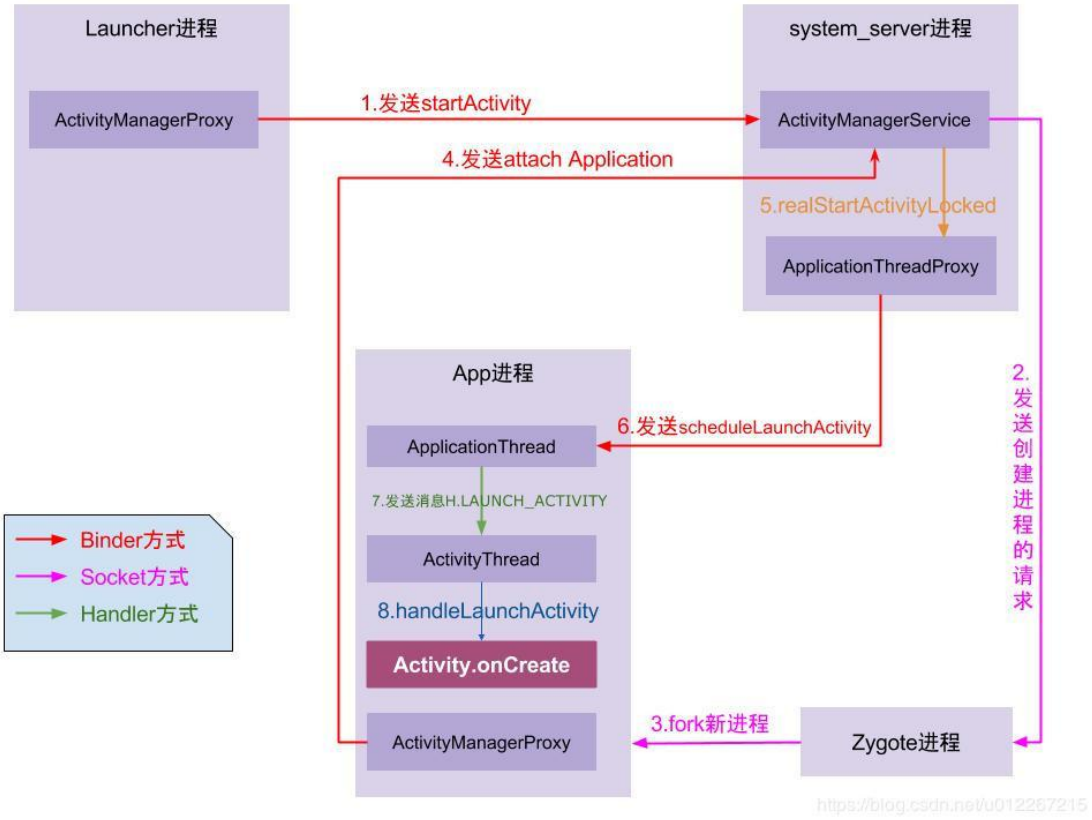
1.4 Activity

1.4.1 Activity 启动流程

用户点击屏幕上面的 app icon，进入响应的 app，背后经历了 Activity 和 AMS 的反反复复的通信过程。手机屏幕他就是一个 activity，而这个 activity 所在的 app 被称之为 Launcher，这是各个手机厂商提供的。

我们在开发 app 时，会在 AndroidManifest.xml 文件中定义默认启动的 activity，设置 activity 的 action 和 category 属性标签。Launcher 位每个 app 的 icon 图标提供了启动这个 app 时所需要的 Intent 信息。这些信息在 app 安装时由 PackageManagerService 从 app 的 AndroidManifest.xml 文件中读取。

1.4.2 点击 Launcher 图标来启动 Activity



启动流程进程间简单分析：

Zygote 进程 -> SystemServer 进程 -> 各种系统服务 -> 应用进程

在 Actvity 启动过程中,其实是应用进程与 SystemServer 进程相互配合启动 Activity 的过程，其中应用进程主要用于执行具体的 Activity 的启动过程，回调生命周期方法等操作，而 SystemServer 进程则主要是调用其中的各种服务，将 Activity 保存在栈中，协调各种系统资源等操作。

通过 ActivityManagerNative -> ActivityManagerService 实现了应用进程与 SystemServer 进程的通讯

通过 AppicationThread <- IApplicationThread 实现了 SystemServer 进程与应用进程的通讯

ActivityManagerProxy 相当于 Proxy

ActivityManagerNative 就相当于 Stub

ActivityManagerService 是 ActivityManagerNative 的具体实现，换句话说，就是 AMS 才是服务端的具体实现！

ApplicationThreadProxy 相当于 Proxy

ApplicationThreadNative 相当于 Stub

ApplicationThread 相当于服务器端，代码真正的实现者！

（1）点击桌面 App 图标，Launcher 进程采用 Binder IPC 向 system_server 进程发起 startActivity 请求；

（2）system_server 进程接收到请求后，向 zygote 进程发送创建进程的请求；

（3）Zygote 进程 fork 出新的子进程，即 App 进程；

（4）App 进程，通过 Binder IPC 向 sytem_server 进程发起 attachApplication 请求；

（5）system_server 进程在收到请求后，进行一系列准备工作后，再通过 binder IPC 向 App 进程发送 scheduleLaunchActivity 请求；

（6）App 进程的 binder 线程（ApplicationThread）在收到请求后，通过 handler 向主线程发送 LAUNCH_ACTIVITY 消息；

（7）主线程在收到 Message 后，通过反射机制创建目标 Activity，并回调 Activity.onCreate()等方法。

（8）到此，App 便正式启动，开始进入 Activity 生命周期，执行完 onCreate/onStart/onResume 方法，UI 渲染结束后便可以看到 App 的主界面。

涉及到的类有：

（1）Instrumentation

（2）ActivityThread

（3）H

（4）LoadedApk

（5）AMS

（6）ActivityManagerNative 和 ActivityManagerProxy

（7）ApplicationThread 和 ApplicationThreadProxy

第一阶段：Launcher 通知 AMS

（1）启动 activity 调用 startActivity 方法，最后会调用 startActivityForResult 方法并传入 code=-1 表示不关心是否启动成功。

（2）startActivityForResult 内部会调用 Instrumentation 的 execStartActivity 方法，这个方法有几个参数需要注意：

```
execStartActivity(
            Context who, IBinder contextThread, IBinder token, Activity target,
            Intent intent, int requestCode, Bundle options)
```

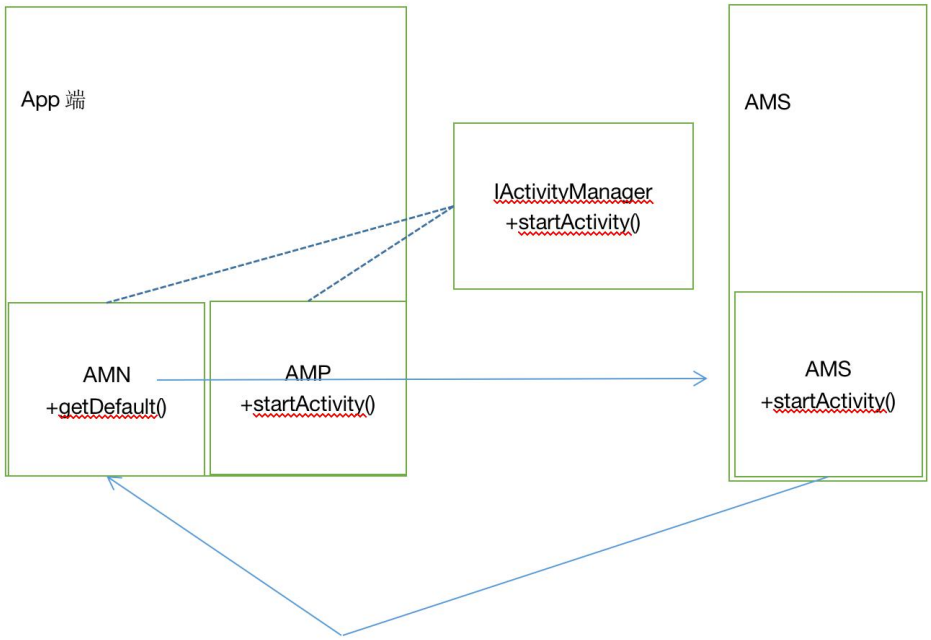
contextThread： 位 ActivityThread 的 getApplicationThread 获取到一个 IBinder 对象，这个对象类型是 ApplicationThread，代表 Launcher 所在的 app 进程。

Token： 也是一个 IBinder 类型，代表 Launcher 这个 activity 也通过 Instrumentation 传给 AMS，AMS 就知道是谁向 AMS 发起了请求。

（3）接着，通过 Instrumentation，Activity 将数据传递给 ActivityManagerNativ。

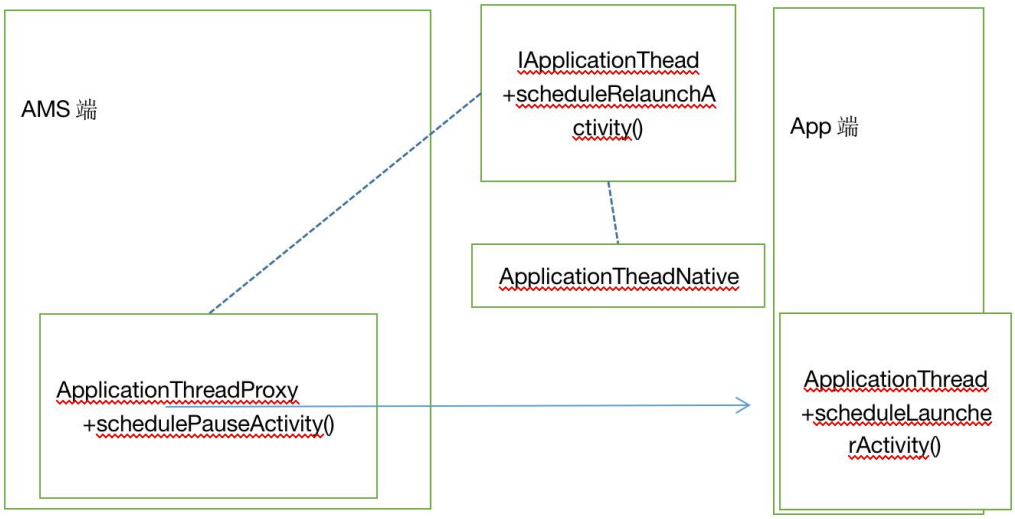
（4）AMN 通过 getDefault 方法，从 ServiceManager 中取得一个 activity 对象将其包装成 ActivityManagerProxy(AMP)对象，他就是 AMS 的代理对象。

（5）AMP 调用 startActivity 方法就会将数据写入到另一个进程 AMS 然后等待 AMS 返回结果。



第二阶段：AMS 处理 Launcher 传过来的信息

AMS 给 Launcher 发送消息，它在 AMS 这边保存一个 ActivityRecord 对象，这个对象里面有个 ApplicationThreadProxy，是一个 Binder 代理对象，就是 ApplicationThread。



AMS 通过 ApplicationThreadProxy 发送消息，在 App 端则通过 ApplicationThread 来接收消息。

第三阶段：

ApplicationThread 接收到来自 AMS 的消息后，调用 ActivityThread（UI 主线程）的 sendMessage 方法，向 Launcher 的主线程消息队列发送一个 PAUSE_ACTIVITY 的消息。

这里面涉及到一个 H 类，AMS 给 activity 发送的所有消息，以及给三大组件发送的消息都是从 H 类经过，因此可以从这里做插件化。

第四阶段：AMS 启动新的进程

新的进程以 ActivityThread 的 main 方法作为入口，启动新进程就是启动一个新的 app。在这个方法中，创建一个主线程 Looper 也就是 MainLooper，其次创建 Application，其次在创建 LoadedApk 对象,然后创建 ContextImpl 通过反射创建目标 Application 调用其 attach 的方法最后再调用 onCreate 方法。

第五阶段：启动相关 Activity

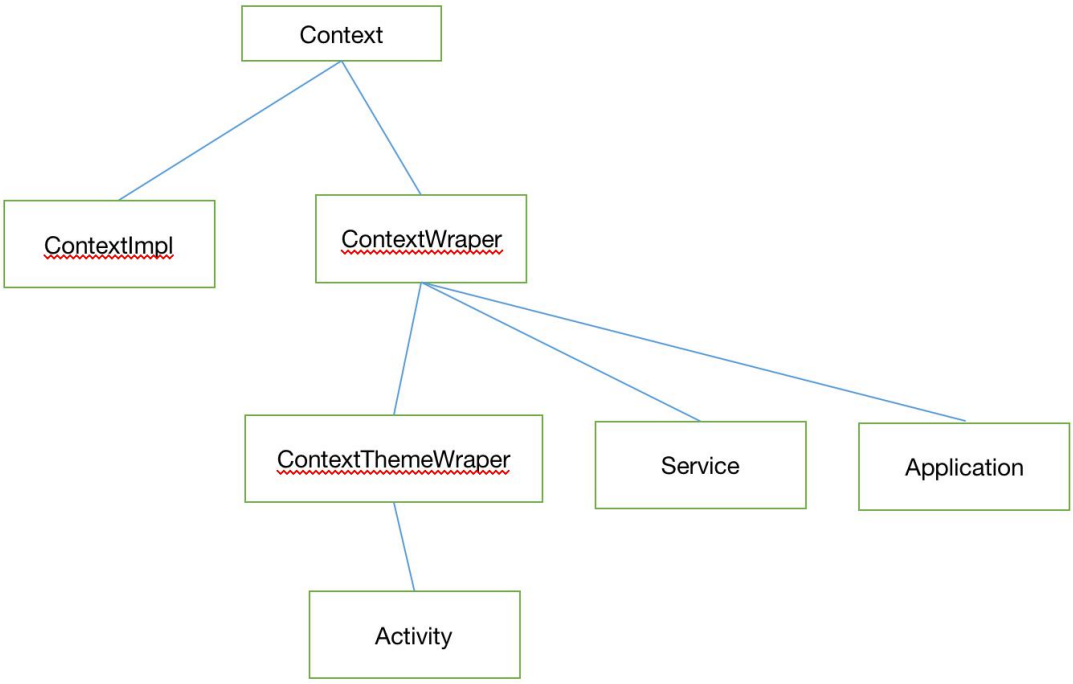
在 Binder 的另一端，App 通过 APT 接收到 AMS 的消息，仍然在 H 的 handleMessage 中处理。在这个 handleMessage 中会调用 ActivityThread 的 handleLaunchActivity 方法。

流程梳理：
每次 ApplicationThread 执行 ActivityThread 的 sendMessage 方法，在这里拼装消息，丢给 H 的 switch 方法去执行，决定执行 ActivityThread 的哪个方法，每次都是这样。

handleLaunchActivity 方法内容：
（1）通过 Instrumentation 的 newActivity 方法，创建要启动的 Activity 实例
（2）为这个 Activity 创建一个上下文 Context，并与 Activity 关联
（3）通过 Instrumentation 的 callActivityOnCreate 方法执行 Activity 的 onCreate 方法从而启动 Activity。

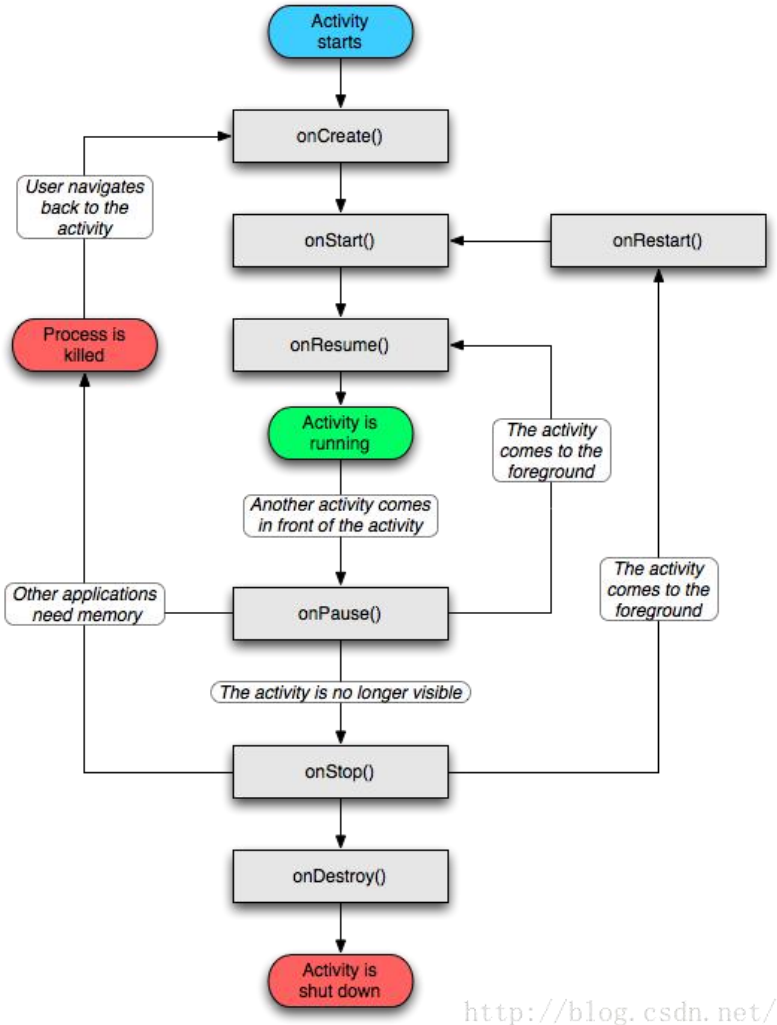
1.4.2.1 ActivityThread.main

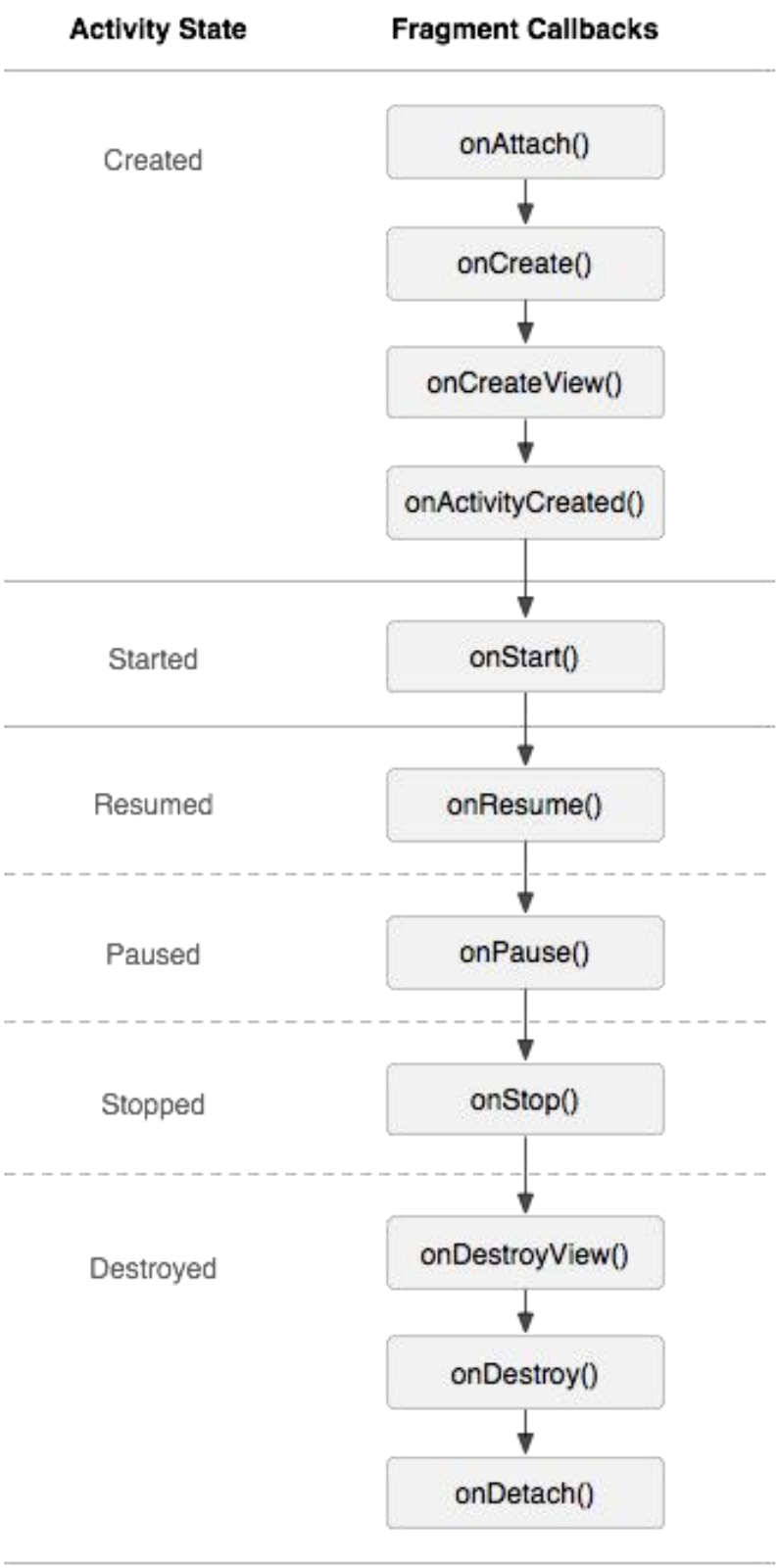
```
public static void main(String[] args) {
//创建 Looper 和 MessageQueue 对象，用于处理主线程的消息
 Looper.prepareMainLooper();


 //创建 ActivityThread 对象
 ActivityThread thread = new ActivityThread();


 //建立 Binder 通道 (创建新线程)
 thread.attach(false);


 Looper.loop(); //消息循环运行
 throw new RuntimeException("Main thread loop unexpectedly exited");
```


Context
ContextImpl
ContextWraper
ContextThemeWraper
Service
Application
Activity

context 关系图

1.4.3 Activity 生命周期

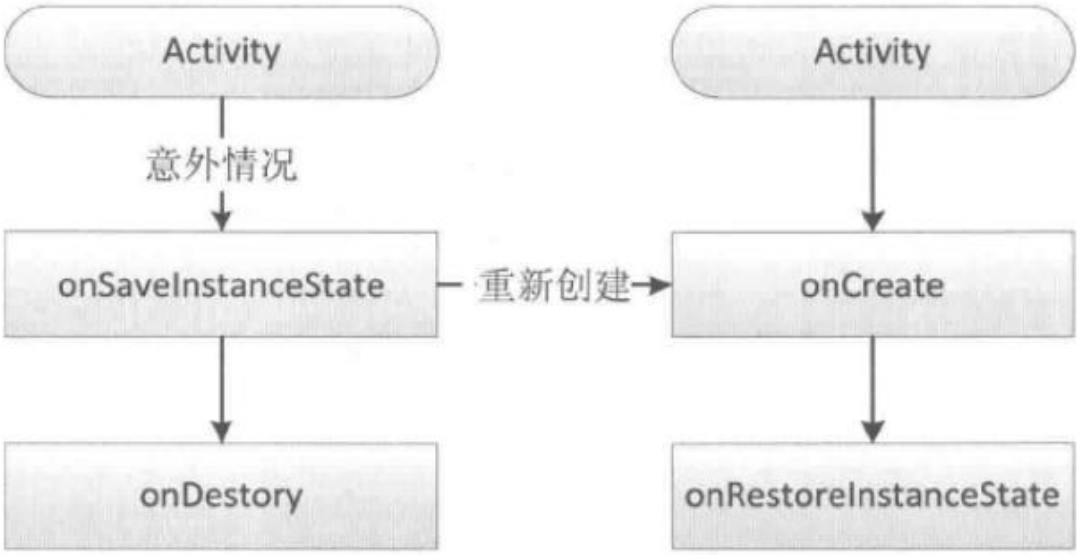




onSaveInstanceState 和 onRestoreInstanceState 调用的过程和时机

（1）调用时机：Activity 的异常情况下（例如转动屏幕或者被系统回收）的情况下，会调用到 onSaveInstanceState 和 onRestoreInstanceState。其他情况不会触发这个过程。但是按 Home 键或者启动新 Activity 仍然会单独触发 onSaveInstanceState 的调用。

（2）调用过程：旧的 Activity 要被销毁时，由于是**异常情况**下的，所以除了正常调用 onPause，onStop，onDestroy 方法外，还会在**调用 onStop 方法前，调用 onSaveInstanceState 方法**。新的 Activity 重建时，我们就可以通过 onRestoreInstanceState 方法取出之前保存的数据并恢复，**onRestoreInstanceState 的调用时机在 onCreate 之后**。



1.4.4 启动模式

（1）*Standard* 模式：

Activity 可以有多个实例，每次启动 Activity，无论任务栈中是否已经有这个 Activity 的实例，系统都会创建一个新的 Activity 实例

（2）*SingleTop* 模式：

当一个 singleTop 模式的 Activity 已经位于任务栈的栈顶，再去启动它时，不会再创建新的实例, 如果不位于栈顶，就会创建新的实例，会调用 onNewIntent 方法

（3）*SingleTask* 模式:

如果 Activity 已经位于栈顶，系统不会创建新的 Activity 实例，和 singleTop 模式一样。但 Activity 已经存在但不位于栈顶时，系统就会把该 Activity 移到栈顶，并把它上面的 activity 出栈，onNewIntent 方法

（4）*SingleInstance* 模式:

singleInstance 模式也是单例的，但和 singleTask 不同，singleTask 只是任务栈内单例，系统里是可以有多个 singleTaskActivity 实例的，而 singleInstance Activity 在整个系统里只有一个实例，启动一 singleInstanceActivity 时，系统会创建一个新的任务栈，并且这个任务栈只有他一个 Activity 生命周期

onNewIntent 的作用和调用时机？

- **调用时机**：如果 Activity 的启动模式是：singleTop, singleTask, singleInstance，在复用这些 Acitivity 时就会在调用 onStart 方法前调用 onNewIntent 方法
- **作用**：让已经创建的 Activity 处理新的 Intent。

1.4.5 两个 Activity 跳转的生命周期

1.启动 *A*

onCreate - onStart - onResume

2. 在 *A* 中启动 *B*

ActivityA onPause

ActivityB onCreate

ActivityB onStart

ActivityB onResume

ActivityA onStop

3.从 *B* 中返回 *A*（按物理硬件返回键）

ActivityB onPause

ActivityA onRestart

ActivityA onStart

ActivityA onResume

ActivityB onStop

ActivityB onDestroy

*4.*继续返回

ActivityA onPause

ActivityA onStop

ActivityA onDestroy

1.4.6 onRestart 的调用场景

（1） 按下 home 键之后，然后切换回来，会调用 onRestart()。

（2） 从本 Activity 跳转到另一个 Activity 之后，按 back 键返回原来 Activity，会调用 onRestart()；

（3） 从本 Activity 切换到其他的应用,然后再从其他应用切换回来,会调用 onRestart()；

1.4.7 横竖屏切换生命周期

说下 Activity 的横竖屏的切换的生命周期，用那个方法来保存数据，两者的区别。触发在什么时候在那个方法里可以获取数据等。

1，默认情况下横竖屏切换，

a，3.2 之前的版本先执行 onPause()，再执行 onSaveInstanceState(Bundle outState)

b，3.2 版本开始先执行 onSaveInstanceState(Bundle outState)，再执行 onPause()

2，配置 android:configChanges="orientation"，防止横竖屏切换重新创建 Activity

a，如果 targetSdkVersion<=12，所有版本都有效

b，如果 targetSdkVersion>12，3.2 版本开始需要多加个 screenSize

一，默认情况

android3.2之前的版本，生命周期方法执行顺序如下

```
onSaveInstanceState(Bundle outState)
onPause()
onStop()
onDestroy()
onCreate()
onStart()
onResume()
```

android3.2版本开始，生命周期方法执行顺序如下

```
onPause()
onSaveInstanceState(Bundle outState)
onStop()
onDestroy()
onCreate()
onStart()
onResume()
```

1.5 Handler 原理

Handler，Message，looper 和 MessageQueue 构成了安卓的消息机制，handler 创建后可以通过 sendMessage 将消息加入消息队列,然后 looper 不断的将消息从 MessageQueue 中取出来，回调到 Hander 的 handleMessage 方法，从而实现线程的通信。IO

主线程：

从两种情况来说，第一在 UI 线程创建 Handler,此时我们不需要手动开启 looper，因为在应用启动时，在 ActivityThread 的 main 方法中就创建了一个当前主线程的 looper，并开启了消息队列，消息队列是一个无限循环，为什么无限循环不会 ANR?因为可以说，应用的整个生命周期就是运行在这个消息循环中的，安卓是由事件驱动的，Looper.loop 不断的接收处理事件，每一个点击触摸或者 Activity 每一个生命周期都是在 Looper.loop 的控制之下的，looper.loop 一旦结束，应用程序的生命周期也就结束了。我们可以想想什么情况下会发生 ANR，第一，事件没有得到处理，第二，事件正在处理，但是没有及时完成，而对事件进行处理的就是 looper，所以只能说事件的处理如果阻塞会导致 ANR，而不能说 looper 的无限循环会 ANR

子线程：

另一种情况就是在子线程创建 Handler,此时由于这个线程中没有默认开启的消息队列，所以我们需要手动调用 looper.prepare(),并通过 looper.loop 开启消息

主线程 Looper 从消息队列读取消息，当读完所有消息时，主线程阻塞。子线程往消息队列发送消息，并且往管道文件写数据，主线程即被唤醒，从管道文件读取数据，主线程被唤醒只是为了读取消息，当消息读取完毕，再次睡眠。因此 loop 的循环并不会对 CPU 性能有过多的消耗。

1.5.1 主线程的消息循环机制是什么

Activity 的生命周期都是依靠主线程的 Looper.loop，当收到不同 Message 时

则采用相应措施：一旦退出消息循环，那么你的程序也就可以退出了。 从消息队列中取消息可能会阻塞,取到消息会做出相应的处理。如果某个消息处理时间 过长,就可能会影响 UI 线程的刷新速率，造成卡顿的现象。

thread.attach(false)方法函数中便会创建一个 Binder 线程（具体是指 ApplicationThread，Binder 的服务端，用于接收系统服务 AMS 发送来的事件），该 Binder 线程通过 Handler 将 Message 发送给主线程。「Activity 启动过程」比如收到 msg=H.LAUNCH_ACTIVITY，则调用 ActivityThread.handleLaunchActivity()方法,最终会通过反射机制,创建 Activity 实例,然后再执行 Activity.onCreate()等方法;再比如收到 msg=H.PAUSE_ACTIVITY，则调用 ActivityThread.handlePauseActivity()方法，最终会执行 Activity.onPause()等方法。主线程的消息又是哪来的呢？当然是 app 进程中的其他线程通过 Handler 发送给主线程

1.6 Requestlayout，onlayout，onDraw，DrawChild

requestLayout()方法 :会导致调用 measure()过程 和 layout()过程 。说明:只是对 View 树重新布局 layout 过程包括 measure()和 layout()过程，如果 view 的 l,t,r,b 没有必变，那就不会触发 onDraw；但是如果这次刷新是在动画里，mDirty 非空，就会导致 onDraw。

onLayout()方法(如果该 View 是 ViewGroup 对象,需要实现该方法,对每个子视图进行布局)

onDraw()方法绘制视图本身 (每个 View 都需要重载该方法，ViewGroup 不需要实现该方法)

drawChild()去重新回调每个子视图的 draw()方法

1.7 invalidate 和 postInvalidate 的区别及使用

View.invalidate()：层层上传到父级，直到传递到 ViewRootImpl 后触发了 scheduleTraversals()，然后整个 View 树开始重新按照 View 绘制流程进行重绘任务。

invalidate:在 ui 线程刷新 view

postInvalidate：在工作线程刷新 view（底层还是 handler）其实它的原理就是 invalidate+handler

View.postInvalidate 最终会调用 ViewRootImpl.dispatchInvalidateDelayed()方法

```
public void dispatchInvalidateDelayed(View view, long delayMilliseconds) {
        Message msg = mHandler.obtainMessage(MSG_INVALIDATE, view);
        mHandler.sendMessageDelayed(msg, delayMilliseconds);
    }
```

这里的 mHandler 是 ViewRootHandler 实例，在该 Handler 的 handleMessage 方法中调用了 view.invalidate()方法。

```
case MSG_INVALIDATE:
                ((View) msg.obj).invalidate();
break;
```

1.8 如何优化自定义 view

1）不要在 onDraw 或是 onLayout()中去创建对象，因为 onDraw()方法可能会被频繁调用，可以在 view 的构造函数中进行创建对象；

2）降低 view 的刷新频率，尽可能减少不必要的调用 invalidate()方法。或是调用带四种参数不同类型的 invalidate()，而不是调用无参的方法。无参变量需要刷新整个 view，而带参数的方法只需刷新指定部分的 view。在 onDraw()方法中减少冗余代码。

3）使用硬件加速，GPU 硬件加速可以带来性能增加。

4）状态保存与恢复，如果因内存不足，Activity 置于后台被杀重启时，View 应尽可能保存自己属性，可以重写 onSaveInstanceState 和 onRestoreInstanceState 方法，状态保存。

1.9 动画

1.9.1 动画分类

（1）frame 帧动画：AnimationDrawable 控制 animation-list.xml 布局。

（2）tween 补间动画：通过指定 View 的初末状态和变化方式，对 View 的内容完成一系列的图形变换来实现动画效果， Alpha, Scale ,Translate, Rotate。

（3）PropertyAnimation 属性动画：3.0 引入，属性动画核心思想是对值的变化。

1.9.2 属性动画&&补间动画的性能差异：

（1）属性动画操作的是对象的实例属性，例如 translationX,然后反射调用 set,geView 动画:t 方法，多个属性动画同时执行，会频繁反射调用类方法，降低性能。

（2）补间动画只产生了一个动画效果，其真实的坐标并没有发生改变，是效果一直在发生变化，没有频繁反射调用方法的耗费性能操作。

1.9.3 原理及特点

（1）帧动画：

是在 xml 中定义好一系列图片之后，使用 AnimatonDrawable 来播放的动画。

（2）View 动画：

只是影像变化，view 的实际位置还在原来地方。

（3）属性动画：

插值器：作用是根据时间流逝的百分比来计算属性变化的百分比。

估值器：在 1 的基础上由这个东西来计算出属性到底变化了多少数值的类。

其实就是利用插值器和估值器，来计出各个时刻 View 的属性，然后通过改变 View 的属性来实现 View 的动画效果。

1.9.4 区别

属性动画才是真正的实现了 view 的移动,补间动画对 view 的移动更像是在不同地方绘制了一个影子，实际对象还是处于原来的地方。 当动画的 repeatCount 设置为无限循环时，如果在 Activity 退出时没有及时将动画停止，属性动画会导致 Activity 无法释放而导致内存泄漏，而补间动画却没问题。 xml 文件实现的补间动画，复用率极高。在 Activity 切换，窗口弹出时等情景中有着很好的效果。 使用帧动画时需要注意，不要使用过多特别大的图，容导致内存不足。

1.9.5 为什么属性动画移动后仍可点击？

播放补间动画的时候，我们所看到的变化，都只是临时的。而属性动画呢，它所改变的东西，却会更新到这个 View 所对应的矩阵中，所以当 ViewGroup 分派事件的时候，会正确的将当前触摸坐标，转换成矩阵变化后的坐标，这就是为什么播放补间动画不会改变触摸区域的原因了。

1.10 RecyclerView 复用机制

（1）在 RecyclerView **中，并不是每次绘制表项，都会重新创建** ViewHolder **对象，也不是每次都会重新绑定** ViewHolder **数据。**

（2）RecyclerView **通过** Recycler **获得下一个待绘制表项。**

（3） Recycler **有 4 个层次用于缓存** ViewHolder **对象，优先级从高到底依次为** ArrayList<ViewHolder> **mAttachedScrap、**ArrayList<ViewHolder> **mCachedViews、**ViewCacheExtension **mViewCacheExtension、RecycledViewPool** mRecyclerPool**。如果四层缓存都未命中，则重新创建并绑定** ViewHolder **对象**

（4） RecycledViewPool 对 ViewHolder 按 viewType 分类存储（通过 SparseArray），同类 ViewHolder 存储在默认大小为 5 的 ArrayList 中

（5） 从 mRecyclerPool 中复用的 ViewHolder 需要重新绑定数据，从 mAttachedScrap 中复用的 ViewHolder 不要重新出创建也不需要重新绑定数据。

https://www.cnblogs.com/dasusu/p/7746946.html

2 Serializable 与 Parcable 的区别

什么是序列化 —— 序列化，表示将一个对象转换成可存储或可传输的状态。序列化后的对象可以在网络上进行传输，也可以存储到本地。

由于在系统底层，数据的传输形式是简单的字节序列形式传递，即在底层，系统不认识对象，只认识字节序列，而为了达到进程通讯的目的，需要先将数据序列化，而**序列化就是将对象转化字节序列的过程**。相反地，当字节序列被运到相应的进程的时候，进程为了识别这些数据，就要将其**反序列化，即把字节序列转化为对象**。

怎么通过序列化传输对象？

Android 中 Intent 如果要传递类对象，可以通过两种方式实现。

方式一：Serializable，要传递的类实现 Serializable 接口传递对象，

方式二：Parcelable，要传递的类实现 Parcelable 接口传递对象。

Serializable（Java 自带）：

Serializable 是序列化的意思，表示将一个对象转换成可存储或可传输的状态。序列化后的对象可以在网络上进行传输，也可以存储到本地。

Parcelable（android 专用）：

除了 Serializable 之外，使用 Parcelable 也可以实现相同的效果，

不过不同于将对象进行序列化，Parcelable 方式的实现原理是将一个完整的对象进行分解，而分解后的每一部分都是 Intent 所支持的数据类型，这样也就实现传递对象的功能了。

实现序列化的作用

1）永久性保存对象，保存对象的字节序列到本地文件中；

2）通过序列化对象在网络中传递对象；

3）通过序列化在进程间传递对象。

选择序列化方法的原则

1）在使用内存的时候，Parcelable 比 Serializable 性能高，所以推荐使用 Parcelable。

2）Serializable 在序列化的时候会产生大量的临时变量，从而引起频繁的 GC。

3）Parcelable 不能使用在要将数据存储在磁盘上的情况，因为 Parcelable 不能很好的保证数据的持续性在外界有变化的情况下。尽管 Serializable 效率低点，但此时还是建议使用 Serializable 。

android 上应该尽量采用 Parcelable，效率至上

编码上：

Serializable 代码量少，写起来方便

Parcelable 代码多一些

效率上：

Parcelable 的速度比高十倍以上

serializable 的迷人之处在于你只需要对某个类以及它的属性实现 Serializable 接口即可。Serializable 接口是一种标识接口（marker interface），这意味着无需实现方法，Java 便会对这个对象进行高效的序列化操作。

这种方法的缺点是使用了反射，序列化的过程较慢。这种机制会在序列化的时候创建许多的临时对象，容易触发垃圾回收。

Parcelable 方式的实现原理是将一个完整的对象进行分解，而分解后的每一部分都是 Intent 所支持的数据类型，这样也就实现传递对象的功能了

3 三级缓存原理

3.1 三级缓存原理

当 Android 端需要获得数据时比如获取网络中的图片，首先从内存中查找（按键查找），内存中没有的再从磁盘文件或 sqlite 中去查找，若磁盘中也没有才通过网络获取

3.2 LruCache 底层实现原理：

LruCache（Least Recently Used）中 Lru 算法的实现就是通过 LinkedHashMap 来实现的。LinkedHashMap 继承于 HashMap，它使用了一个双向链表来存储 Map 中的 Entry 顺序关系，对于 get、put、remove 等操作，LinkedHashMap 除了要做 HashMap 做的事情，还做些调整 Entry 顺序链表的工作。

LruCache 中将 LinkedHashMap 的顺序设置为 LRU 顺序来实现 LRU 缓存，每次调用 get(也就是从内存缓存中取图片)，则将该对象移到链表的尾端。
调用 put 插入新的对象也是存储在链表尾端，这样当内存缓存达到设定的最大值时，将链表头部的对象（近期最少用到的）移除。

4 进程保活

（1）Service 设置成 START_STICKY kill 后会被重启(等待 5 秒左右)，重传 Intent，保持与重启前一样

（2）通过 startForeground 将进程设置为前台进程， 做前台服务，优先级和前台应用一个级别，除非在系统内存非常缺，否则此进程不会被 kill

（3）双进程 Service： 让 2 个进程互相保护对方，其中一个 Service 被清理后，另外没被清理的进程可以立即重启进程

（4）用 C 编写守护进程(即子进程）：Android 系统中当前进程(Process)fork 出来的子进程，被系统认为是两个不同的进程。当父进程被杀死的时候，子进程仍然可以存活，并不受影响(Android5.0 以上的版本不可行）联系厂商，加入白名单

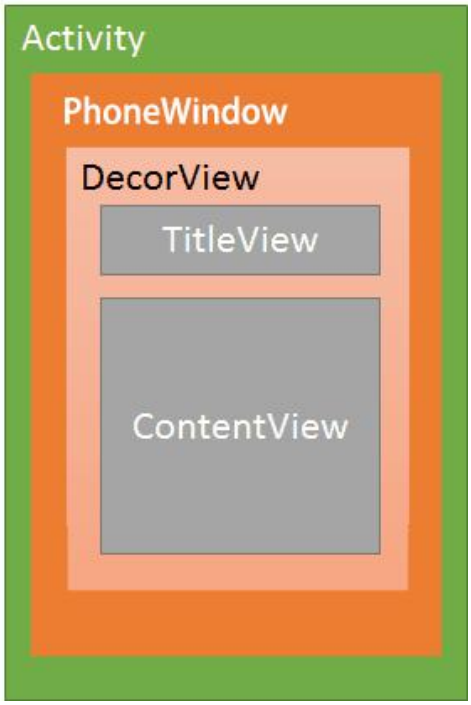
（5）锁屏状态下，开启一个一像素 Activity

5. view 部分知识点

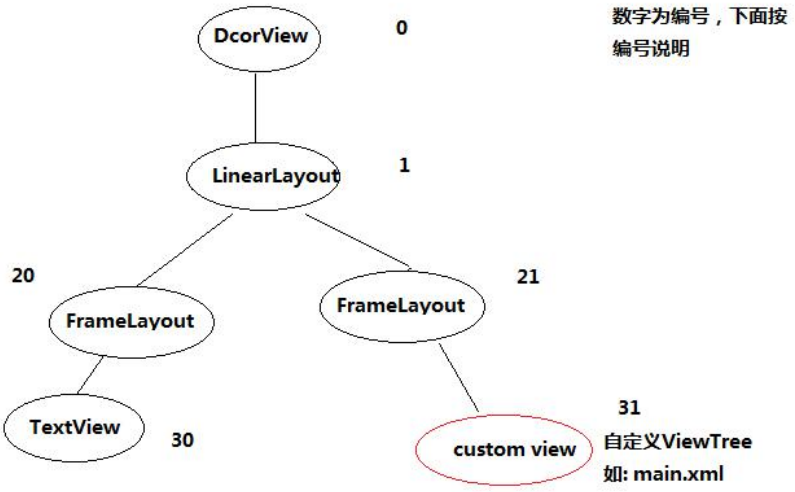
5.1 DecorView 浅析

Android View 源码解读：浅谈 DecorView 与 ViewRootImpl

DecorView 为整个 Window 界面的最顶层 View，它只有一个子元素 LinearLayout。代表整个 Window 界面，包含通知栏、标题栏、内容显示栏三块区域。其中 LinearLayout 中有两个 FrameLayout 子元素。



1



- （20）标题栏 FrameLayout

其中（20）为标题栏显示界面，只有一个 TextView 显示应用的名称。

- （21）内容栏 FrameLayout

其中（21）位内容栏显示界面，就是 setContentView()方法载入的布局界面。

5.1.1 DecorView 的作用

1、 DecorView 是顶级 View，本质是一个 FrameLayout

2、 它包含两部分，标题栏和内容栏，都是 FrameLayout

3、 内容栏 id 是 content，也就是 activity 中设置 setContentView 的部分，最终将布局添加到 id 为 content 的 FrameLayout 中。

4、 获取 content：ViewGroup content=findViewById（R.android.id.content）

5、 获取设置的 View：content.getChildAt(0).

5.1.2 使用总结

1、 每个 Activity 都包含一个 Window 对象，Window 对象通常是由 PhoneWindow 实现的。

2、 PhoneWindow：将 DecorView 设置为整个应用窗口的根 View，是 Window 的实现类。它是 Android 中的最基本的窗口系统，每个 Activity 均会创建一个 PhoneWindow 对象，是 Activity 和整个 View 系统交互的接口。

3、 DecorView：是顶层视图，将要显示的具体内容呈现在 PhoneWindow 上，DecorView 是当前 Activity 所有 View 的祖先，它并不会向用户呈现任何东西。

5.2 View 的事件分发

图解 Android 事件分发机制

5.2.1 ViewGroup 事件分发

```
onInterceptTouchEvent(MotionEvent ev)
```

```
//该方法用来进行事件的分发，即无论ViewGroup或者View的事件，都是从这个方法开始的。
public boolean dispatchTouchEvent(MotionEvent ev)
```

和

```
//这个方法表示对事件进行处理，在dispatchTouchEvent方法内部调用，如果返回true表示消耗当前事件，如果返回false表示不消耗当前事件。
public boolean onTouchEvent(MotionEvent ev)
```

当一个点击事件产生后，它的传递过程将遵循如下顺序：

Activity -> Window -> View

事件总是会传递给 Activity，之后 Activity 再传递给 Window，最后 Window 再传递给顶级的 View，顶级的 View 在接收到事件后就会按照事件分发机制去分发事件。如果一个 View 的 onTouchEvent 返回了 FALSE，那么它的父容器的 onTouchEvent 将会被调用，依次类推，如果所有都不处理这个事件的话，那么 Activity 将会处理这个事件。

对于 ViewGroup 的事件分发过程，大概是这样的：如果顶级的 ViewGroup 拦截事件即 onInterceptTouchEvent 返回 true 的话，则事件会交给 ViewGroup 处理，如果 ViewGroup 的 onTouchListener 被设置的话，则 onTouch 将会被调用，否则的话 onTouchEvent 将会被调用，也就是说：两者都设置的话，onTouch 将会屏蔽掉 onTouchEvent，在 onTouchEvent

中，如果设置了 onClickerListener 的话，那么 onClick 将会被调用。如果顶级 ViewGroup 不拦截的话，那么事件将会被传递给它所在的点击事件的子 view，这时候子 view 的 dispatchTouchEvent 将会被调用

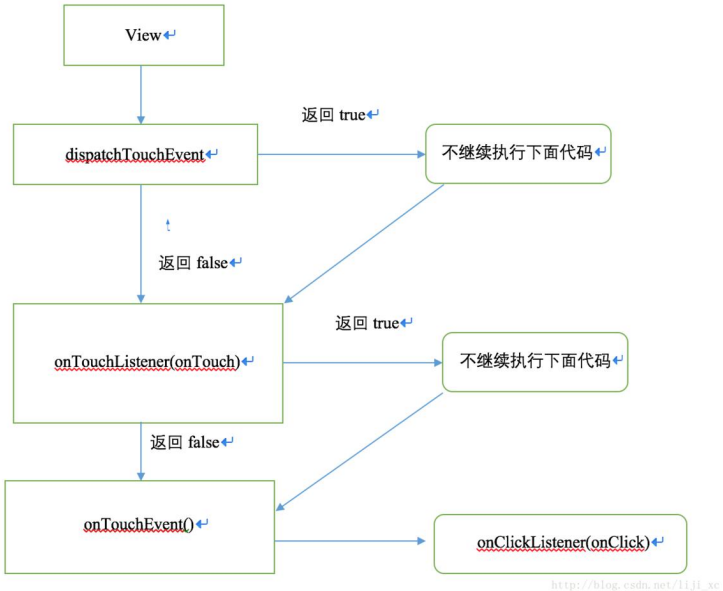
```
public boolean dispatchTouchEvent(MotionEvent event){
    boolean consume = false;
    // 父View决定是否拦截事件
    if(onInterceptTouchEvent(event)){
        // 父View调用onTouchEvent(event)消费事件，如果该方法返回true，表示
        // 该View消费了该事件，后续该事件序列的事件（Down、Move、Up）将不会在传递
        // 该其他View。
        consume = onTouchEvent(event);
    }else{
        // 调用子View的dispatchTouchEvent(event)方法继续分发事件
        consume = child.dispatchTouchEvent(event);
    }
    return consume;
}
```

5.2.2 View 的事件分发

dispatchTouchEvent -> onTouch(setOnTouchListener) -> onTouchEvent -> onClick

5.2.3 onTouch 和 onTouchEvent 的区别

两者都是在 dispatchTouchEvent 中调用的，onTouch 优先于 onTouchEvent，如果 onTouch 返回 true，那么 onTouchEvent 则不执行，及 onClick 也不执行。



5.2.4 滑动冲突解决

外部拦截法：

(onInterceptTouchEvent)，如果父容器需要则拦截，如果不需要则不拦截，称为外部拦截法

内部拦截法：

父容器不拦截任何事件，将所有事件传递给子元素，如果子元素需要则消耗掉，如果不需要则通过 requestDisallowInterceptTouchEvent 方法(请求父类不要拦截,返回值为 true 时不拦截,返回值为 false 时为拦截)交给父容器处理,称为内部拦截法，

5.3 View 的绘制

5.3.1 onMeasure(int widthMeasureSpec, int heightMeasureSpec)

在 xml 布局文件中，我们的 layout_width 和 layout_height 参数可以不用写具体的尺寸，而是 wrap_content 或者是 match_parent。这两个设置并没有指定真正的大小，可是我们绘制到屏幕上的 View 必须是要有具体的宽高的，正是因为这个原因，我们必须自己去处理和设置尺寸。当然了，View 类给了默认的处理，但是如果 View 类的默认处理不满足我们的要求，我们就得重写 onMeasure 函数啦~。

一个 int 整数，里面放了测量模式和尺寸大小。int 型数据占用 32 个 bit，而 google 实现的是，将 int 数据的前面 2 个 bit 用于区分不同的布局模式，后面 30 个 bit 存放的是尺寸的数据。

```
int widthMode = MeasureSpec.getMode(widthMeasureSpec);
int widthSize = MeasureSpec.getSize(widthMeasureSpec);
```

测量模式	表示意思
UNSPECIFIED	父容器没有对当前View有任何限制，当前View可以任意取尺寸
EXACTLY	当前的尺寸就是当前View应该取的尺寸
AT_MOST	当前尺寸是当前View能取的最大尺寸

match_parent—>EXACTLY。怎么理解呢？match_parent 就是要利用父 View 给我们提供的所有剩余空间，而父 View 剩余空间是确定的，也就是这个测量模式的整数里面存放的尺寸。

wrap_content—>AT_MOST。怎么理解：就是我们想要将大小设置为包裹我们的 view 内容，那么尺寸大小就是父 View 给我们作为参考的尺寸，只要不超过这个尺寸就可以啦，具体尺寸就根据我们的需求去设定。

固定尺寸（如 100dp）—>EXACTLY。用户自己指定了尺寸大小，我们就不用再去干涉了，当然是以指定的大小为主啦。

```
private int getMySize(int defaultSize, int measureSpec) {
       int mySize = defaultSize;

       int mode = MeasureSpec.getMode(measureSpec);
       int size = MeasureSpec.getSize(measureSpec);

       switch (mode) {
           case MeasureSpec.UNSPECIFIED: {//如果没有指定大小，就设置为默认大小
               mySize = defaultSize;
               break;
           }
           case MeasureSpec.AT_MOST: {//如果测量模式是最大取值为size
               //我们将大小取最大值,你也可以取其他值
               mySize = size;
               break;
           }
           case MeasureSpec.EXACTLY: {//如果是固定的大小，那就不要去改变它
               mySize = size;
               break;
           }
       }
       return mySize;
}
```

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        super.onMeasure(widthMeasureSpec, heightMeasureSpec);
        int width = getMySize(100, widthMeasureSpec);
        int height = getMySize(100, heightMeasureSpec);

        if (width < height) {
            height = width;
        } else {
            width = height;
        }

        setMeasuredDimension(width, height);
}
```

5.3.2 onDraw

自定义属性

首先我们需要在`res/values/styles.xml`文件（如果没有请自己新建）里面声明一个我们自定义的属性：

```
<resources>

    <!--name为声明的"属性集合"名，可以随便取，但是最好是设置为跟我们的View一样的名称-->
    <declare-styleable name="MyView">
        <!--声明我们的属性，名称为default_size,取值类型为尺寸类型（dp,px等）-->
        <attr name="default_size" format="dimension" />
    </declare-styleable>
</resources>
```

命名空间："http://schemas.android.com/apk/res-auto"

```
public MyView(Context context, AttributeSet attrs) {
    super(context, attrs);
    //第二个参数就是我们在styles.xml文件中的<declare-styleable>标签
      //即属性集合的标签，在R文件中名称为R.styleable+name
      TypedArray a = context.obtainStyledAttributes(attrs, R.styleable.MyView);

      //第一个参数为属性集合里面的属性，R文件名称：R.styleable+属性集合名称+下划线+属性名称
      //第二个参数为，如果没有设置这个属性，则设置的默认的值
      defalutSize = a.getDimensionPixelSize(R.styleable.MyView_default_size, 100);

      //最后记得将TypedArray对象回收
      a.recycle();
}
```

5.4 ViewGroup 的绘制

自定义 ViewGroup 可就没那么简单啦~，因为它不仅要管好自己的，还要兼顾它的子 View。我们都知道 ViewGroup 是个 View 容器，它装纳 child View 并且负责把 child View 放入指定的位置。

1、 首先，我们得知道各个子 View 的大小吧，只有先知道子 View 的大小，我们才知道当前的 ViewGroup 该设置为多大去容纳它们。
2、 根据子 View 的大小，以及我们的 ViewGroup 要实现的功能，决定出 ViewGroup 的大小
3、 ViewGroup 和子 View 的大小算出来了之后，接下来就是去摆放了吧，具体怎么去摆放呢？这得根据你定制的需求去摆放了，比如，你想让子 View 按照垂直顺序一个挨着一个放，或者是按照先后顺序一个叠一个去放，这是你自己决定的。
4、 已经知道怎么去摆放还不行啊，决定了怎么摆放就是相当于把已有的空间”分割”成大大小小的空间，每个空间对应一个子 View，我们接下来就是把子 View 对号入座了，把它们放进它们该放的地方去。

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    //将所有的子View进行测量，这会触发每个子View的onMeasure函数
    //注意要与measureChild区分，measureChild是对单个view进行测量
    measureChildren(widthMeasureSpec, heightMeasureSpec);
    int widthMode = MeasureSpec.getMode(widthMeasureSpec);
    int widthSize = MeasureSpec.getSize(widthMeasureSpec);
    int heightMode = MeasureSpec.getMode(heightMeasureSpec);
    int heightSize = MeasureSpec.getSize(heightMeasureSpec);
    int childCount = getChildCount();
    if (childCount == 0) {//如果没有子View,当前ViewGroup没有存在的意义，不用占用空间
        setMeasuredDimension( measuredWidth: 0,  measuredHeight: 0);
    } else {
        if (widthMode == MeasureSpec.AT_MOST && heightMode == MeasureSpec.AT_MOST) {
            //我们将高度设置为所有子View的高度相加，宽度设为子View中最大的宽度
            int height = getTotleHeight();
            int width = getMaxChildWidth();
            setMeasuredDimension(width, height);
        } else if (heightMode == MeasureSpec.AT_MOST) {//如果只有高度是包裹内容
            //宽度设置为ViewGroup自己的测量宽度，高度设置为所有子View的高度总和
            setMeasuredDimension(widthSize, getTotleHeight());
        } else if (widthMode == MeasureSpec.AT_MOST) {//如果只有宽度是包裹内容
            //宽度设置为子View中宽度最大的值，高度设置为ViewGroup自己的测量值
            setMeasuredDimension(getMaxChildWidth(), heightSize);

        }
    }
}
```

5.4.1 onLayout

```
@Override
    protected void onLayout(boolean changed, int l, int t, int r, int b) {
        int count = getChildCount();
        //记录当前的高度位置
        int curHeight = t;
        //将子View逐个摆放
        for (int i = 0; i < count; i++) {
            View child = getChildAt(i);
            int height = child.getMeasuredHeight();
            int width = child.getMeasuredWidth();
            //摆放子View，参数分别是子View矩形区域的左、上、右、下边
            child.layout(l, curHeight, l + width, curHeight + height);
            curHeight += height;
        }
    }
```

5.5 SurfaceView

SurfaceView 中采用了双缓冲机制，保证了 UI 界面的流畅性，同时 SurfaceView 不在主线程中绘制，而是另开辟一个线程去绘制，所以它不妨碍 UI 线程；

SurfaceView 继承于 View，他和 View 主要有以下三点区别：

(1)View 底层没有双缓冲机制，SurfaceView 有；

(2)view 主要适用于主动更新，而 SurfaceView 适用与被动的更新，如频繁的刷新

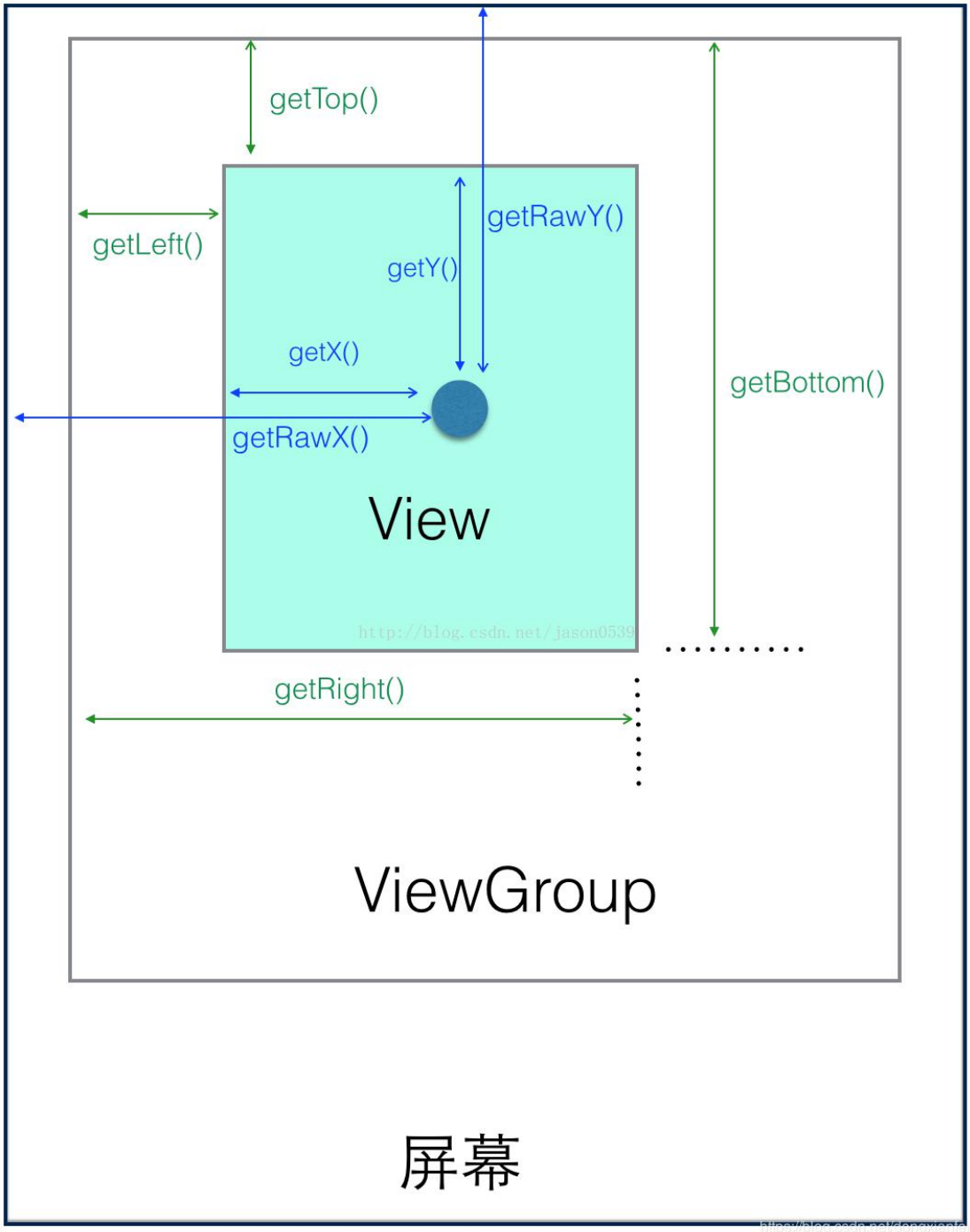
(3)view 会在主线程中去更新 UI，而 SurfaceView 则在子线程中刷新；SurfaceView 的内容不在应用窗口上，所以不能使用变换（平移、缩放、旋转等）。也难以放在 ListView 或者 ScrollView 中，不能使用 UI 控件的一些特性比如 View.setAlpha()

View：显示视图，内置画布，提供图形绘制函数、触屏事件、按键事件函数等；必须在 UI 主线程内更新画面，速度较慢。

SurfaceView：基于 view 视图进行拓展的视图类，更适合 2D 游戏的开发；是 view 的子类，类似使用双缓机制，在新的线程中更新画面所以刷新界面速度比 view 快，Camera 预览界面使用 SurfaceView。

GLSurfaceView：基于 SurfaceView 视图再次进行拓展的视图类，专用于 3D 游戏开发的视图；是 SurfaceView 的子类，openGL 专用。

5.6 坐标系

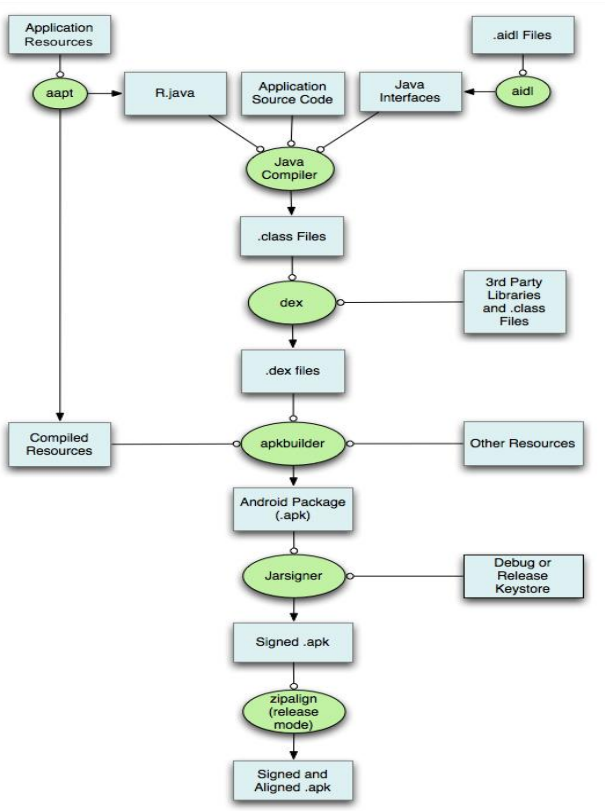

getTop()
getLeft()
getRawY()
getY()
getX()
getRawX()
getBottom()
View
http://blog.csdn.net/jason0539
getRight()
ViewGroup
屏幕

6 系统原理

6.1 打包原理

Android 的包文件 APK 分为两个部分：代码和资源，所以打包方面也分为资源打包和代码打包两个方面，这 篇文章就来分析资源和代码的编译打包原理。
具体说来：

1. 通过 AAPT 工具进行资源文件（包括 AndroidManifest.xml、布局文件、各种 xml 资源等）的打包，生成 R.java 文件。
2. 通过 AIDL 工具处理 AIDL 文件，生成相应的 Java 文件。
3. 通过 Javac 工具编译项目源码，生成 Class 文件。
4. 通过 DX 工具将所有的 Class 文件转换成 DEX 文件，该过程主要完成 Java 字节码转换成 Dalvik 字节码，压缩常量池以及清除冗余信息等工作。
5. 通过 ApkBuilder 工具将资源文件、DEX 文件打包生成 APK 文件。
6. 利用 KeyStore 对生成的 APK 文件进行签名。
7. 如果是正式版的 APK,还会利用 ZipAlign 工具进行对齐处理,对齐的过程就是将 APK 文件中所有的资源文件举例文件的起始距离都偏移 4 字节的整数倍，这样通过内存映射访问 APK 文件 的速度会更快。



6.1.1 问题复现

（1）插件资源和 host 资源产生冲突解决方案

问题引出：

每个资源都对应一个 R 中的 16 进制，由三部组成，packageId,TypeId,EntryId 组成

packageId：apk 包的 id，默认为 0x7f

TypeId:资源类型 id，如 layout，string，ID，drawable 等

EntryId：类型 TypeId 下面的资源 id，从 0 开始递增

如：0x7f0b006d,

packageId:0x7f

TypeId:0b

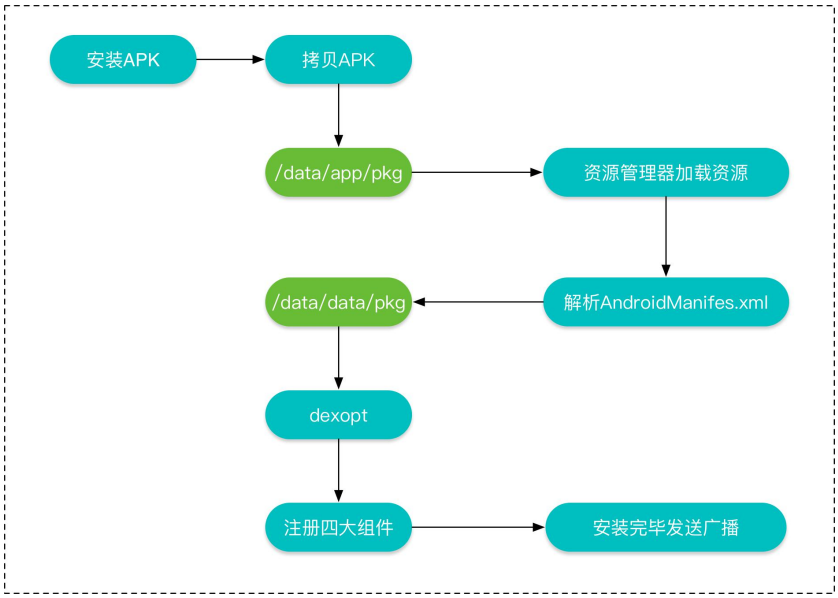
EntryId:006d

解决方法：

设置不同的 packageId,它们是由 aapt 这个工具进行打包的，修改它的源码工具

6.2 安装流程

1. 复制 APK 到/data/app 目录下，解压并扫描安装包。
2. 资源管理器解析 APK 里的资源文件。
3. 解析 AndroidManifest 文件，并在/data/data/目录下创建对应的应用数据目录。
4. 然后对 dex 文件进行优化，并保存在 dalvik-cache 目录下。
5. 将 AndroidManifest 文件解析出的四大组件信息注册到 PackageManagerService 中。
6. 安装完成后，发送广播。



6.3 混淆

好处：混淆将主项目及依赖库中未被使用的类，类成员，方法，属性移除，有助于规避 64K 方法数的瓶颈，会删除无用的资源，有效的减小 apk 包的大小，同时将类及其成员，方法重命名为无意义的简短名称，增加逆向工程的难度。

混淆操作：

1. 压缩（Shrinking）
2. 优化（Optimization）
3. 混淆（Obfuscation）
4. 预校验（Preverification）

6.3.1 正常 app 混淆规则：

1. 四大组件和 application 不能混淆，需要在 AndroidManifest 中声明
2. R 文件不能混淆，因为有时需要反射获取资源
3. support 的 v4 和 v7 包中的类不能混淆，系统的东西不要动
4. 实现了 Serializable 和 Parcelable 的类不混淆，否则反序列化会出现问题
5. 泛型不能混淆
6. 注解不混淆，反射时会使用
7. 不能混淆枚举中的 value 和 valueOf 方法，会被反射使用
8. 自定义 view 不能混淆，否则在 layout 中使用的自定义 view 会找不到。

6.3.2 插件 app 混淆规则

1. 遵守上述规则

6.3.3 插件 app 混淆方案

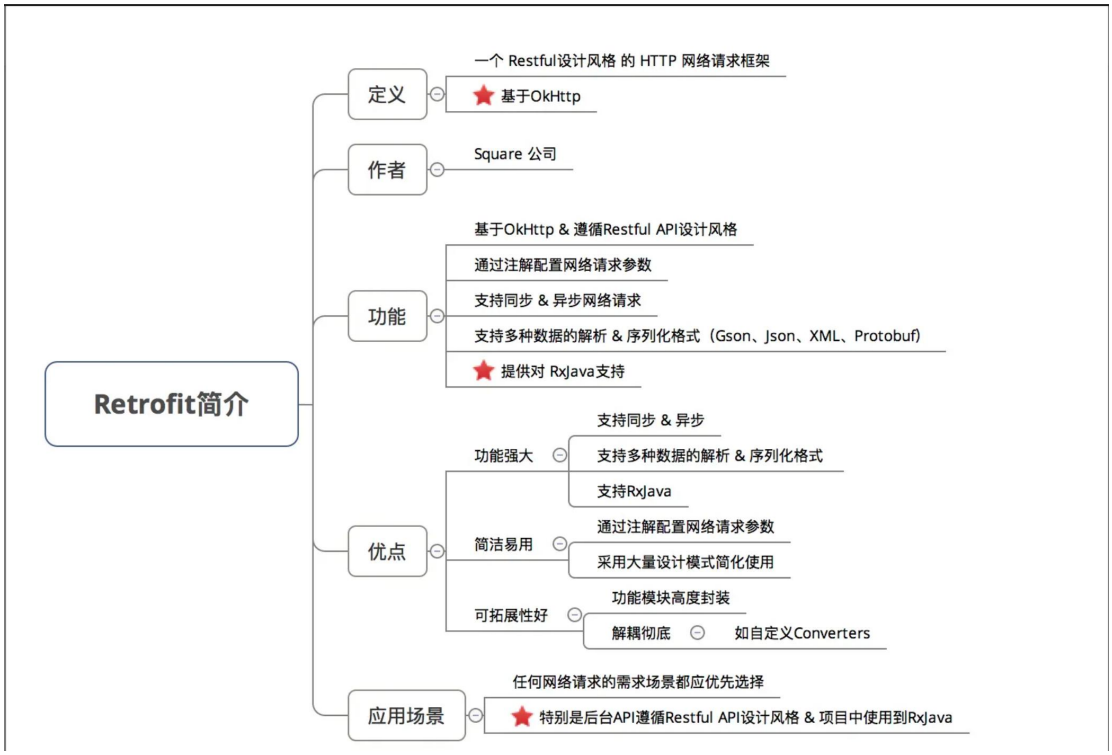
1. 不混淆公共库（provide 引用方式）
2.

7. 第三方库解析

7.1 网络请求框架

7.1.1 Retrofit

分析 retrofit 源码学习设计模式 https://www.jianshu.com/p/0c055ad46b6c



概念：Retrofit 是一个基于 RESTful 的 HTTP 网络请求框架的封装，其中网络请求的本质是由 OKHttp 完成的，而 Retrofit 仅仅负责网络请求接口的封装。

原理：App 应用程序通过 Retrofit 请求网络，实际上是使用 Retrofit 接口层封装请求参数，Header、URL 等信息，之后由 OKHttp 完成后续的请求，在服务器返回数据之后，OKHttp 将原始的结果交给 Retrofit，最后根据用户的需求对结果进行解析。

请求过程：

（1）通过解析网络请求接口的注解，配置网络请求参数

（2）通过动态代理生成网络请求对象

（3）通过网络请求适配器将网络请求对象进行平台（Android，RxJava，Guava，Java8）适配

（4）通过网络请求执行器发送网络请求

（5）通过数据转换器解析服务器返回的数据

（6）通过回调执行器切换线程（子线程到主线程）

（7）用户在主线程中返回结果

步骤	角色	流程	设计模式
1	接口-注解（BuildRequest）	配置网络请求参数（通过注解）	建造者模式，（**Retrofit 使用建造者模式通过 Builder 类创建一个实例**） 工厂方法模式 外观模式，代理模式（**Retrofit 是通过外观和代理模式使用 create 方法创建网络请求接口的实例,同时通过网络接口请求里设置的注解进行了网络请求参数的配置**） 单例模式() 策略模式 装饰器模式
2	网络请求执行器（Call）	创建网络请求对象	
3	网络请求适配器	适配到具体的 call	适配器模式

	（CallAdapter）		
4	网络请求执行器 （Call）	发送网络请求	代理模式
5	数据转换器 （Converter）	服务器返回结果 解析数据	
6	回调执行器 （Executor）	切换线程 （子线程到主线程）	适配器模式，装饰模式 **在 Android extends Platform 这个类中，有个 MainThreadExecutor 方法**
7		处理返回的数据	

总结：

Retrofit 采用了外观模式统一调用创建网络请求接口实例和网络请求参数配置的方法：

（1）动态创建网络请求接口的实例（动态代理模式）

（2）创建 ServiceMethod 对象（建造者模式&单例模式（缓存机制））

（3）对 ServiceMethod 对象进行网络请求参数配置，通过解析网络请求接口的参数，返回值和注解类型，从从 Retrofit 对象中获取对应的网络请求的 url 地址、网络请求执行器、网络请求适配器 & 数据转换器。**（策略模式）**

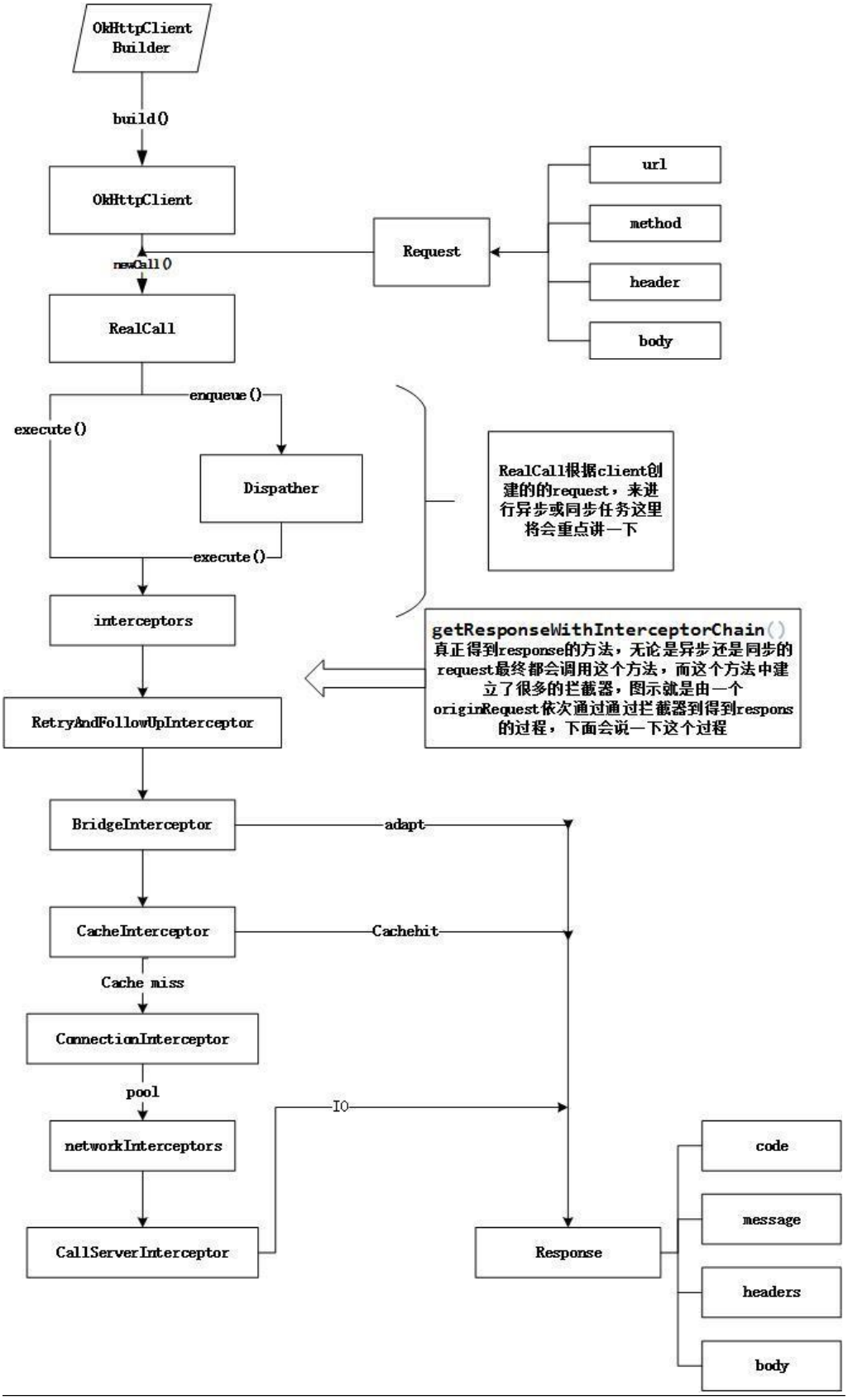
（4）对 serviceMethod 对象加入线程切换的操作，便于接收数据后通过 Handler 从子线程切换到主线程从而对返回数据结果进行处理**（装饰模式）**

（5）最终创建并返回一个 OkHttpCall 类型的网络请求对象

步骤		具体过程		使用的设计模式
步骤1	创建Retrofit实例	通过内部类Builder类建立一个Retrofit实例（建造者模式），具体创建过程是配置了： • 平台类型对象（Platform – Android） • 网络请求的url地址（baseUrl） • 网络请求工厂（callFactory）– 默认使用OkHttpCall（工厂方法模式） • 网络请求适配器工厂的集合（adapterFactories）– 默认是ExecutorCallAdapterFactory• • 数据转换器工厂的集合（converterFactories）– 本质是配置了数据转换器工厂 • 回调方法执行器（callbackExecutor）– 默认回调方法执行器作用是：切换线程（子线程 – 主线程）		建造者模式、工厂方法模式
步骤2	创建网络请求接口的实例（通过解析注解配置网络请求参数）	Retrofit采用了外观模式统一调用创建网络请求接口实例和网络请求参数配置的方法，具体过程： 1. 动态创建网络请求接口的实例（代理模式 – 动态代理# InnovationHandler对象# invoke（）） 2. 创建 serviceMethod 对象（建造者模式 & 单例模式（缓存机制）） 3. 对 serviceMethod 对象进行网络请求参数配置：通过解析网络请求接口方法的参数、返回值和注解类型，从Retrofit对象中获取对应的网络请求的url地址、网络请求执行器、网络请求适配器 & 数据转换器。（策略模式） 4. 对 serviceMethod 对象加入线程切换的操作，便于接收数据后通过Handler从子线程切换到主线程从而对返回数据结果进行处理（装饰模式） 5. 最终创建并返回一个OkHttpCall类型的网络请求对象		外观模式、代理模式、建造者模式、单例模式、策略模式、装饰模式
步骤3	发送网络请求	异步方式	同步方式	代理模式、适配器模式、装饰模式
		1. 通过静态delegate代理对网络请求接口的方法中的每个参数利用对应ParameterHandler进行解析，再根据ServiceMethod对象创建一个OkHttp的Request对象，并封装成OkHttp.call对象（代理模式） 2. 通过静态delegate通过OkHttp.call对象发送网络请求（代理模式）	1. 对网络请求接口的方法中的每个参数利用对应ParameterHandler进行解析，再根据ServiceMethod对象创建一个OkHttp的Request对象，并封装成OkHttp.call对象 2. 通过OkHttp.call对象发送网络请求	
步骤4	解析数据	对返回的数据使用之前设置的数据转换器（GsonConverterFactory）解析返回的数据，最终得到一个Response<T>对象	对返回的数据使用之前设置的数据转换器（GsonConverterFactory）解析返回的数据，最终得到一个Response<T>对象	
步骤5	切换线程	使用回调执行器进行线程切换（子线程 – 主线程）（适配器模式、装饰模式）	无线程切换	
步骤6	处理结果	在主线程处理返回的数据结果	通过Response<T>对象处理返回的数据结果	

7.1.2 Okhttp

https://www.jianshu.com/p/196f7b2a703c


OkHttpClient Builder
build()
OkHttpClient
newCall()
RealCall
Request
url
method
header
body
enqueue()
execute()
Dispather
execute()
RealCall根据client创建的的request，来进行异步或同步任务这里将会重点讲一下
interceptors
getResponseWithInterceptorChain()
真正得到response的方法，无论是异步还是同步的request最终都会调用这个方法，而这个方法中建立了很多的拦截器，图示就是由一个originRequest依次通过通过拦截器到得到respons的过程，下面会说一下这个过程
RetryAndFollowUpInterceptor
BridgeInterceptor
adapt
CacheInterceptor
Cachehit
Cache miss
ConnectionInterceptor
pool
networkInterceptors
IO
CallServerInterceptor
Response
code
message
headers
body

OKHttp 优点

支持 HTTP2/SPDY（SPDY 是 Google 开发的基于 TCP 的传输层协议，用以最小化网络延迟，提升网络速度，优化用户的网络使用体验。）

socket 自动选择最好路线，并支持自动重连，拥有自动维护的 socket 连接池，减少握手次数，减少了请求延迟，共享 Socket,减少对服务器的请求次数。

基于 Headers 的缓存策略减少重复的网络请求。

拥有 Interceptors 轻松处理请求与响应（自动处理 GZip 压缩）。

keepalive connections

当然大量的连接每次连接关闭都要三次握手四次分手的很显然会造成性能低下，因此 http 有一种叫做 keepalive connections 的机制，它可以在传输数据后仍然保持连接，当客户端需要再次获取数据时，直接使用刚刚空闲下来的连接而不需要再次握手。

Okhttp 支持 5 个并发 KeepAlive，默认链路生命为 5 分钟(链路空闲后，保持存活的时间)。

缓存策略

-

1、如果网络不可用并且无可用的有效缓存，则返回 504 错误；

2、继续，如果不需要网络请求，则直接使用缓存；

3、继续，如果需要网络可用，则进行网络请求；

4、继续，如果有缓存，并且网络请求返回 HTTP_NOT_MODIFIED，说明缓存还是有效的，则合并网络响应和缓存结果。同时更新缓存；

5、继续，如果没有缓存，则写入新的缓存；

五种拦截器

看到这里，不禁会问，上面看到的那么多种拦截器到底分别是用来干啥的呢，在这里来总结一下（来自网络，只是做个归纳）：

RetryAndFollowUpInterceptor

用来实现连接失败的重试和重定向

BridgeInterceptor

用来修改请求和响应的 header 信息

CacheInterceptor

用来实现响应缓存。比如获取到的 Response 带有 Date,Expires,Last-Modified,Etag 等 header，表示该 Response 可以缓存一定的时间，下次请求就可以不需要发往服务端，直接拿缓存的

ConnectInterceptor

用来打开到服务端的连接。其实是调用了 StreamAllocation 的 newStream 方法来打开连接的。建联的 TCP 握手，TLS 握手都发生该阶段。过了这个阶段，和服务端的 socket 连接打通

CallServerInterceptor

用来发起请求并且得到响应。上一个阶段已经握手成功，HttpStream 流已经打开，所以这个阶段把 Request 的请求信息传入流中，并且从流中读取数据封装成 Response 返回

总结：

OkHttp 的底层是通过 Java 的 Socket 发送 HTTP 请求与接受响应的(这也好理解，HTTP 就是基于 TCP 协议的)，但是 OkHttp 实现了连接池的概念，即对于同一主机的多个请求，其实可以公用一个 Socket 连接，而不是每次发送完 HTTP 请求就关闭底层的 Socket，这样就实现了连接池的概念。而 OkHttp 对 Socket 的读写操作使用的 Okio 库进行了一层封装。

7.2 图片加载库对比

Picasso：120K

Glide：475K

Fresco：3.4M

Android-Universal-Image-Loader：162K

图片函数库的选择需要根据 APP 的具体情况而定，对于严重依赖图片缓存的 APP，例如壁纸类，图片社交类 APP 来说，可以选择最专业的 Fresco。对于一般的 APP，选择 Fresco 会显得比较重，毕竟 Fresco3.4M 的体量摆在这。根据 APP 对图片的显示和缓存的需求从低到高，我们可以对以上函数库做一个排序。

Picasso < Android-Universal-Image-Loader < Glide < Fresco

7.2.1 介绍:

Picasso ：和 Square 的网络库一起能发挥最大作用，因为 Picasso 可以选择将网络请求的缓存部分交给了 okhttp 实现。

Glide：模仿了 Picasso 的 API，而且在他的基础上加了很多的扩展(比如 gif 等支持)，Glide 默认的 Bitmap 格式是 RGB_565，比 Picasso 默认的 ARGB_8888 格式的内存开销要小一半；Picasso 缓存的是全尺寸的(只缓存一种)，而 Glide 缓存的是跟 ImageView 尺寸相同的(即 56*56 和 128*128 是两个缓存) 。

FB 的图片加载框架 Fresco:最大的优势在于 5.0 以下(最低 2.3)的 bitmap 加载。在 5.0 以下系统，Fresco 将图片放到一个特别的内存区域(Ashmem 区)。当然，在图片不显示的时候，占用的内存会自动被释放。这会使得 APP 更加流畅，减少因图片内存占用而引发的 OOM。为什么说是 5.0 以下，因为在 5.0 以后系统默认就是存储在 Ashmem 区了。

7.2.3 总结:

Picasso 所能实现的功能，Glide 都能做，无非是所需的设置不同。但是 Picasso 体积比起 Glide 小太多如果项目中网络请求本身用的就是 okhttp 或者 retrofit(本质还是 okhttp)，那么建议用 Picasso，体积会小很多(Square 全家桶的干活)。Glide 的好处是大型的图片流，比如 gif、Video，如果你们是做美拍、爱拍这种视频类应用，建议使用。

Fresco 在 5.0 以下的内存优化非常好，代价就是体积也非常的大，按体积算

Fresco>Glide>Picasso

不过在使用起来也有些不便（小建议：他只能用内置的一个 ImageView 来实现这些功能，用起来比较麻烦，我们通常是根据 Fresco 自己改改，直接使用他的 Bitmap 层）

7.3 各种 json 解析库使用

参考链接：https://www.cnblogs.com/kunpengit/p/4001680.html

7.3.1 Google 的 Gson

Gson 是目前功能最全的 Json 解析神器,Gson 当初是为因应 Google 公司内部需求而由 Google 自行研发而来，但自从在 2008 年五月公开发布第一版后已被许多公司或用户应用。Gson 的应用主要为 toJson 与 fromJson 两个转换函数，无依赖，不需要例外额外的 jar，能够直接跑在 JDK 上。而在使用这种对象转换之前需先创建好对象的类型以及其成员才能成功的将 JSON 字符串成功转换成相对应的对象。类里面只要有 get 和 set 方法，Gson 完全可以将复杂类型的 json 到 bean 或 bean 到 json 的转换，是 JSON 解析的神器。Gson 在功能上面无可挑剔，但是性能上面比 FastJson 有所差距。

7.3.2 阿里巴巴的 FastJson

Fastjson 是一个 Java 语言编写的高性能的 JSON 处理器,由阿里巴巴公司开发。
无依赖，不需要例外额外的 jar，能够直接跑在 JDK 上。FastJson 在复杂类型的 Bean 转换 Json 上会出现一些问题，可能会出现引用的类型，导致 Json 转换出错，需要制定引用。
FastJson 采用独创的算法，将 parse 的速度提升到极致，超过所有 json 库。

综上 Json 技术的比较,在项目选型的时候可以使用 Google 的 Gson 和阿里巴巴的 FastJson 两种并行使用，如果只是功能要求，没有性能要求，可以使用 google 的 Gson，如果有性能上面的要求可以使用 Gson 将 bean 转换 json 确保数据的正确，使用 FastJson 将 Json 转换 Bean

8 热点技术

参考链接- Android 组件化方案

8.1 组件化

8.1.2 概念：

组件化：是将一个 APP 分成多个 module，每个 module 都是一个组件，也可以是一个基础库供组件依赖，开发中可以单独调试部分组件，组件中不需要相互依赖但是可以相互调用，最终发布的时候所有组件以 lib 的形式被主 APP 工程依赖打包成一个 apk。

8.1.3 由来：

1、 APP 版本迭代，新功能不断增加，业务变得复杂，维护成本高
2、 业务耦合度高，代码臃肿，团队内部多人协作开发困难
3、 Android 编译代码卡顿，单一工程下代码耦合严重，修改一处需要重新编译打包，耗时耗力。
4、 方便单元测试，单独改一个业务模块，不需要着重关注其他模块。

8.1.4 优势：

1、 组件化将通用模块独立出来，统一管理，以提高复用，将页面拆分为粒度更小的组件，组件内部出了包含 UI 实现，还可以包含数据层和逻辑层
2、 每个组件度可以独立编译、加快编译速度、独立打包。
3、 每个工程内部的修改，不会影响其他工程。

4、 业务库工程可以快速拆分出来，集成到其他 App 中。

5、 迭代频繁的业务模块采用组件方式，业务线研发可以互不干扰、提升协作效率，并控制产品质量，加强稳定性。

6、 并行开发，团队成员只关注自己的开发的小模块，降低耦合性，后期维护方便等。

8.1.5 考虑问题：

模式切换：如何使得 APP 在单独调试跟整体调试自由切换

组件化后的每一个业务的 module 都可以是一个单独的 APP（isModuleRun=false），release 包的时候各个业务 module 作为 lib 依赖，这里完全由一个变量控制，在根项目 gradle.properties 里面 isModuleRun=true。isModuleRun 状态不同，加载 application 和 AndroidManifest 都不一样，以此来区分是独立的 APK 还是 lib。

在 build.grade 里面配置：

```
if (isModuleRun.toBoolean()) {
    apply plugin: 'com.android.application'
} else {
    apply plugin: 'com.android.library'
}

##单Module运行需要配置
sourceSets {
        main {
            if (isModuleRun.toBoolean()) {
                manifest.srcFile 'src/main/debug/AndroidManifest.xml'
            } else {
                manifest.srcFile 'src/main/AndroidManifest.xml'
                java {
                    //全部Module一起编译的时候剔除debug目录
                    exclude '**/debug/**'
                }
            }
        }
    }
```

资源冲突：当我们创建了多个 Module 的时候，如何解决相同资源文件名合并的冲突

业务 Module 和 BaseModule 资源文件名称重复会产生冲突，解决方案在

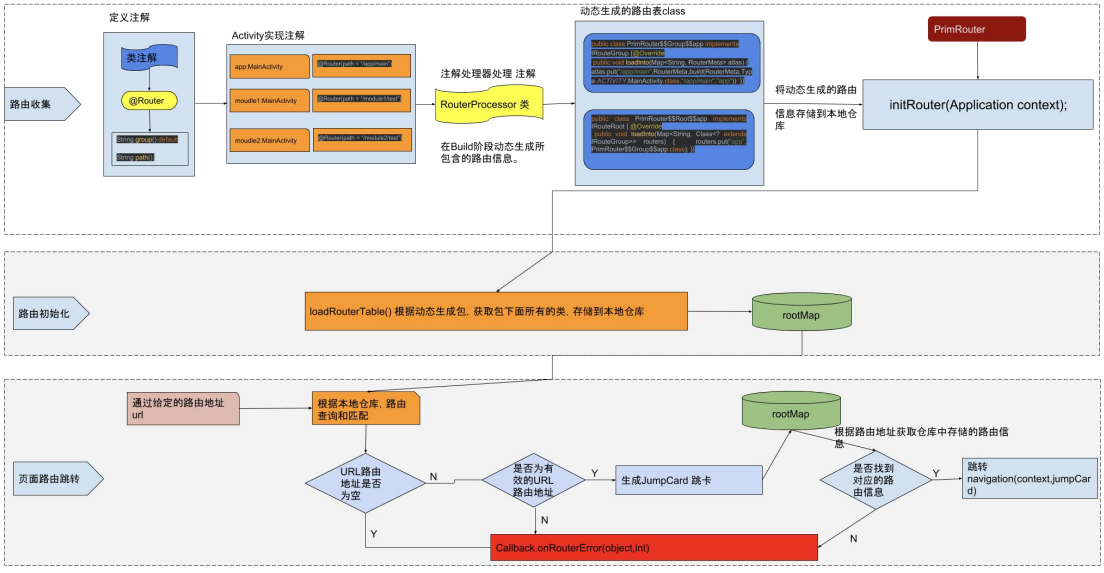
每个 module 都有 app_name，为了不让资源名重名，在每个组件的 build.gradle 中增加 resourcePrefix “xxx_强行检查资源名称前缀。固定每个组件的资源前缀。但是 resourcePrefix 这个值只能限定 xml 里面的资源，并不能限定图片资源。

依赖关系：多个 Module 之间如何引用一些共同的 library 以及工具类

组件通信：组件化之后，Module 之间是相互隔离的，如何进行 UI 跳转以及方法调用

阿里巴巴 ARouter

ARouter 核心实现思路是，我们在代码里加入的@Route 注解，会在编译时期通过 apt 生成一些存储 path 和 activityClass 映射关系的类文件，然后 app 进程启动的时候会拿到这些类文件，把保存这些映射关系的数据读到内存里(保存在 map 里)，然后在进行路由跳转的时候，通过 build()方法传入要到达页面的路由地址，ARouter 会通过它自己存储的路由表找到路由地址对应的 Activity.class(activity.class = map.get(path))，然后 new Intent()，当调用 ARouter 的 withString()方法它的内部会调用 intent.putExtra(String name, String value)，调用 navigation()方法，它的内部会调用 startActivity(intent)进行跳转，这样便可以实现两个相互没有依赖的 module 顺利的启动对方的 Activity 了。



各业务 Module 之前不需要任何依赖可以通过路由跳转，完美解决业务之间耦合。

入口参数：我们知道组件之间是有联系的，所以在单独调试的时候如何拿到其它的 Module 传递过来的参数

Application：

当组件单独运行的时候，每个 Module 自成一个 APK，那么就意味着会有多个 Application，很显然我们不愿意重复写这么多代码，所以我们只需要定义一个 BaseApplication 即可，其它的 Application 直接继承此 BaseApplication 就 OK 了，BaseApplication 里面还可定义公用的参数。

得到 APP 组件化

8.2 插件化

参考链接- 插件化入门

https://blog.csdn.net/github_37130188/article/details/89762543

8.2.1 概述

提到插件化，就不得不提起方法数超过 65535 的问题，我们可以通过 Dex 分包来解决，同时也可以通过使用插件化开发来解决。插件化的概念就是由宿主 APP 去加载以及运行插件 APP。

8.2.2 优点

1、 在一个大的项目里面，为了明确的分工，往往不同的团队负责不同的插件 APP，这样分工更加明确。各个模块封装成不同的插件 APK，不同模块可以单独编译，提高了开发效率。
2、 解决了上述的方法数超过限制的问题。
3、 可以通过上线新的插件来解决线上的 BUG，达到“热修复”的效果。
4、 减小了宿主 APK 的体积。

8.2.3 缺点

插件化开发的 APP 不能在 Google Play 上线，也就是没有海外市场。

8.2.4 插件化总结

HostApp：壳 app
PluginApp：插件 app

（1）插件中类的加载

HostApp 想要加载 Pluginapp 中的类，使用 HostApp 中的 classloader 是不行的，解决方案有以下三种：
a. 在反射插件中的类时可以使用插件的 classloader
b. 宿主和插件它们各自的 classloader 都对应一个 dex 数组，把这些 dex 数组都合并到宿主的 dex 数组中，那么宿主 app 就可以通过反射加载任何插件中的类，
c. 自定义一个 classloader，取代原先的宿主 classloader，同时在自定义的 classloader 中放一个集合，承载所有插件的 classloader，那么这个自定义 classloader 在加载任何一个类时都会在宿主中查找，如果没有的话再遍历内部的 classloader 集合，看哪个插件的 classloader 可以加载这个类。

（2）哪些地方可以 Hook

a. app 中可以使用的类，可以 Hook，系统源码中标记了 hide 的类和方法不可以 hook，但是可以通过反射去调用它们。但是在 app 中可以使用的类可以 Hook，如 Instrumentation 和 Callback

b. 实现了接口的类，可以通过动态代理的方式

（3）Activity 的插件化解决方案

Activity 插件化解决方案主要有动态替换和静态代理两种。

a. 动态替换。占位思想，在 HostApp 中申明一个 StubActivity，启动插件的 ActivityA，但是告诉 AMS 启动的是 StubActivity，欺骗成功后在即将启动 Activity 时再将 StubActivity 改为 ActivityA，因此需要 Hook 一些系统方法。

b. 静态代理。在 HostApp 中设计一个 ProxyActivity，插件中的 Activity 都是没有生命周期的，在 ProxyActivity 的生命周期中，调用插件 Activity 相应的生命周期函数。

（4）资源的插件化解决方案

app 是通过 AssetManager 来加载资源的，它通过 addAssetPath 方法加载指定位置的资源，默认是加载 App 自身的资源。从 HostApp 跳转到 plugin 只能夹杂 HostApp 下面的资源而不能加载插件的资源，解决方案如下：

a.将 HostApp 和插件的资源都通过 AssetManager 的 addAssetPath 方法添加到一起，在插件 Activity 的基类中重写 getResource 方法。不过会产生资源 ID 冲突的问题：解决方法有：

1.修改 aapt，为每个插件的 id 指定不能的前缀，默认的 0x7f，可以改为 0x71 等

2.修改 resource.arsc，在 aapt 执行后生成了 R.java 和 resource.arsc ，可以把插件中 R.java 中的所有资源前缀都改为 0x71，把 resource.arsc 中的 0x7f 也改为 0x71 即可。

（5）Service 的插件化解决方案

1.动态代理，事先在 HostApp 中的 AndroidManifest.xml 中申明多个 StubService，分别对应不同的插件 Service

2.静态代理，创建一个 ProxyService，有 ProxyService 来启动插件的 Service，缺点是插件中有几个 Service，HostApp 中 AndroidManifest.xml 也要申明相同数量的 ProxyService

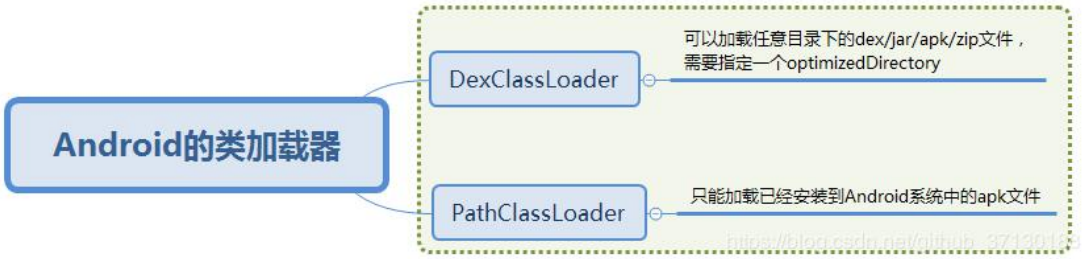
（6）BroadCastReceiver 的插件化解决方案

同上

8.3 热修复

（1）PathClassLoader：只能加载已经安装到 Android 系统中的 apk 文件（/data/app 目录），是 Android 默认使用的类加载器。

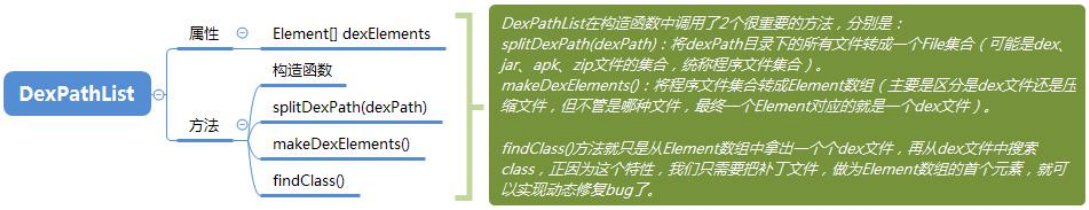
（2）DexClassLoader：可以加载任意目录下的 dex/jar/apk/zip 文件，比 PathClassLoader 更灵活，是实现热修复的重点。



类加载器肯定会提供有一个方法来供外界找到它所加载到的 class，该方法就是 findClass()，不过在 PathClassLoader 和 DexClassLoader 源码中都没有重写父类的 findClass()方法，但它们的父类 BaseDexClassLoader 就有重写 findClass()

BaseDexClassLoader 的 findClass()方法实际上是通过 DexPathList 对象（pathList）的 findClass()方法来获取 class 的，而这个 DexPathList 对象恰好在之前的 BaseDexClassLoader 构造函数中就已经被创建好了。

DexPathList 的构造函数是将一个个的程序文件（可能是 dex、apk、jar、zip）封装成一个个 Element 对象，最后添加到 Element 集合中。

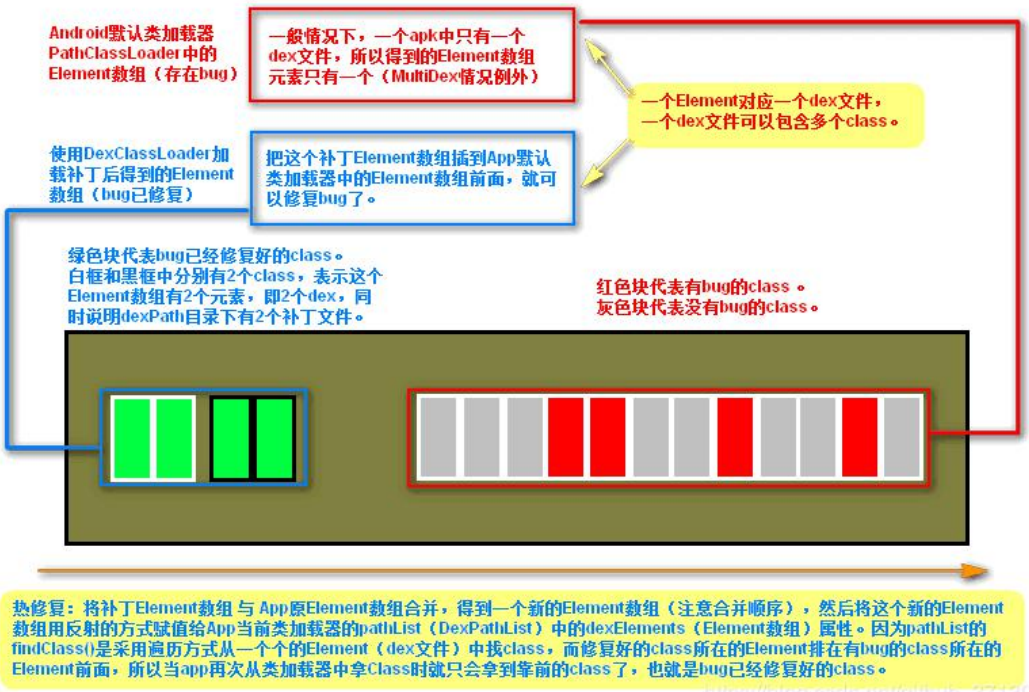


热修复的实现原理：

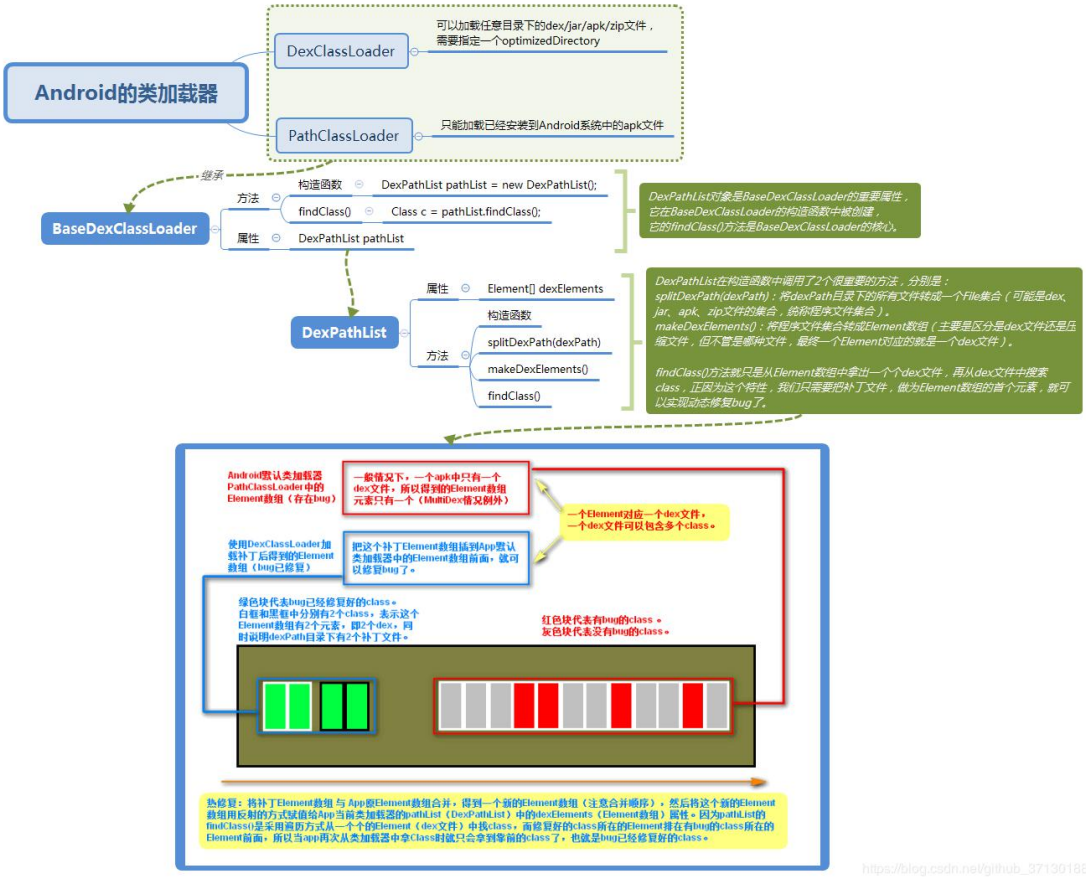
经过对 PathClassLoader、DexClassLoader、BaseDexClassLoader、DexPathList 的分析，我们知道，安卓的类加载器在加载一个类时会先从自身 DexPathList 对象中的 Element 数组中获取（Element[] dexElements）到对应的类，之后再加载。采用的是数组遍历的方式，不过注意，遍历出来的是一个个的 dex 文件。

在 for 循环中，首先遍历出来的是 dex 文件，然后再是从 dex 文件中获取 class，所以，我们只要让修复好的 class 打包成一个 dex 文件，放于 Element 数组的第一个元素，这样就能保证获取到的 class 是最新修复好的 class 了（当然，有 bug 的 class 也是存在的，不过是放在了 Element 数组的最后一个元素中，所以没有机会被拿到而已）。

利用 PathClassLoader 和 DexClassLoader 去加载与 bug 类同名的类，替换掉 bug 类，进而达到修复 bug 的目的，原理是在 app 打包的时候阻止类打上 **CLASS_ISPREVERIFIED** 标志，然后在热修复的时候动态改变 BaseDexClassLoader 对象间接引用的 dexElements，替换掉旧的类。


Android默认类加载器PathClassLoader中的Element数组（存在bug）
一般情况下，一个apk中只有一个dex文件，所以得到的Element数组元素只有一个（MultiDex情况例外）
一个Element对应一个dex文件，一个dex文件可以包含多个class。
使用DexClassLoader加载补丁后得到的Element数组（bug已修复）
把这个补丁Element数组插到App默认类加载器中的Element数组前面，就可以修复bug了。
绿色块代表bug已经修复好的class。
白框和黑框中分别有2个class，表示这个Element数组有2个元素，即2个dex，同时说明dexPath目录下有2个补丁文件。
红色块代表有bug的class。
灰色块代表没有bug的class。
热修复：将补丁Element数组与App原Element数组合并，得到一个新的Element数组（注意合并顺序），然后将这个新的Element数组用反射的方式赋值给App当前类加载器的pathList（DexPathList）中的dexElements（Element数组）属性。因为pathList的findClass()是采用遍历方式从一个个的Element（dex文件）中找class，而修复好的class所在的Element排在有bug的class所在的Element前面，所以当app再次从类加载器中拿Class时就只会拿到靠前的class了，也就是bug已经修复好的class。

总结：


Android的类加载器
DexClassLoader
可以加载任意目录下的dex/jar/apk/zip文件，需要指定一个optimizedDirectory
PathClassLoader
只能加载已经安装到Android系统中的apk文件
继承
BaseDexClassLoader
方法
构造函数
DexPathList pathList = new DexPathList();
findClass()
Class c = pathList.findClass();
属性
DexPathList pathList
DexPathList对象是BaseDexClassLoader的重要属性，它在BaseDexClassLoader的构造函数中被创建，它的findClass()方法是BaseDexClassLoader的核心。
DexPathList
属性
Element[] dexElements
构造函数
方法
splitDexPath(dexPath)
makeDexElements()
findClass()
DexPathList在构造函数中调用了2个很重要的方法，分别是：splitDexPath(dexPath)：将dexPath目录下的所有文件转成一个File集合（可能是dex、jar、apk、zip文件的集合，统称程序文件集合）。makeDexElements()：将程序文件集合转成Element数组（主要是区分是dex文件还是压缩文件，但不管是哪种文件，最终一个Element对应的就是一个dex文件）。
findClass()方法就只是从Element数组中拿出一个个dex文件，再从dex文件中搜索class，正因为这个特性，我们只需要把补丁文件，做为Element数组的首个元素，就可以实现动态修复bug了。

8.4 多渠道打包

多渠道打包关注两件事：

1. 将渠道信息写入 apk 中，
2. 将 apk 中的渠道信息传输到后台

8.4.1 签名方式

（1）Android7.0 之前使用 v1 签名方式，是 jar signature，源自于 JDK。7.0 和 7.0 以下的版本安装没有问题

（2）Android7.0 之后使用 v2 签名方式，是 Android 独有的 apk signature。7.0 以下的版本安装失败

9 屏幕适配

9.1 基本概念

屏幕尺寸

含义：手机对角线的物理尺寸 单位：英寸（inch），1 英寸=2.54cm

Android 手机常见的尺寸有 5 寸、5.5 寸、6 寸等等

屏幕分辨率

含义：手机在横向、纵向上的像素点数总和

一般描述成屏幕的”宽 x 高”=AxB 含义：屏幕在横向方向（宽度）上有 A 个像素点，在纵向方向

（高）有 B 个像素点 例子：1080x1920，即宽度方向上有 1080 个像素点，在高度方向上有 1920 个像素点

单位：px（pixel），1px=1 像素点

UI 设计师的设计图会以 px 作为统一的计量单位

Android 手机常见的分辨率：320x480、480x800、720x1280、1080x1920

屏幕像素密度

含义：每英寸的像素点数 单位：dpi（dots per ich）

假设设备内每英寸有 160 个像素，那么该设备的屏幕像素密度=160dpi

9.2 适配方法

1.支持各种屏幕尺寸： 使用 wrap_content, match_parent, weight.要确保布局的灵活性并适应各种尺寸的屏幕，应使用 “wrap_content”、“match_parent” 控制某些视图组件的宽度和高度。

2.使用相对布局，禁用绝对布局。

3.使用 LinearLayout 的 weight 属性

假如我们的宽度不是 0dp(**wrap_content 和 0dp 的效果相同**)，则是 match_parent 呢？

android:layout_weight 的真实含义是:如果 View 设置了该属性并且有效，那么该 View 的宽度等于原有宽度(android:layout_width)加上剩余空间的占比。

从这个角度我们来解释一下上面的现象。在上面的代码中，我们设置每个 Button 的宽度都是 match_parent，假设屏幕宽度为 L，那么每个 Button 的宽度也应该都为 L，剩余宽度就等于 L-（L+L）= -L。

Button1 的 weight=1，剩余宽度占比为 1/(1+2)= 1/3，所以最终宽度为 L+1/3*(-L)=2/3L，Button2 的计算类似，最终宽度为 L+2/3(-L)=1/3L。

4.使用.9 图片

10 性能优化

参考链接：Android 性能监测工具，优化内存、卡顿、耗电、APK 大小的方法

参考博客：http://mtancode.com/

1、 稳定（内存溢出、崩溃）

2、 流畅（卡顿）

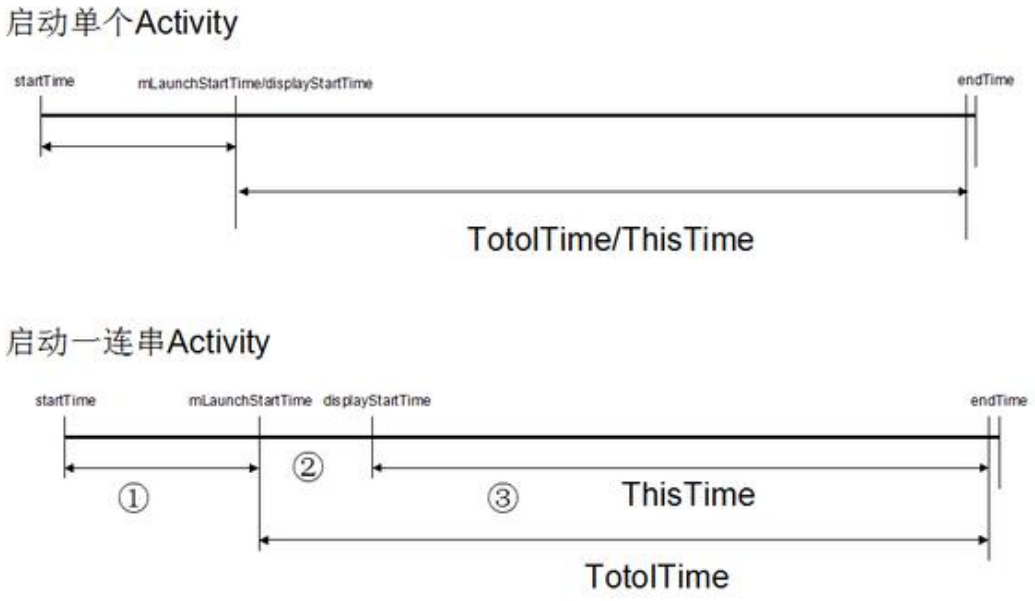
3、 耗损（耗电、流量）

4、 安装包（APK 瘦身）

影响稳定性的原因很多，比如内存使用不合理、代码异常场景考虑不周全、代码逻辑不合理等，都会对应用的稳定性造成影响。其中最常见的两个场景是：Crash 和 ANR，这两个错误将会使得程序无法使用。所以做好 Crash 全局监控，处理闪退同时把崩溃信息、异常信息收集记录起来，以便后续分析;合理使用主线程处理业务，不要在主线程中做耗时操作，防止 ANR 程序无响应发生。

10.0 启动 app 黑白屏优化

app 启动时间计算：**adb shell am start -w packagename/activity,**



WaitTime 就是总的耗时，包括前一个应用 Activity pause 的时间和新应用启动的时间；
ThisTime 表示一连串启动 Activity 的最后一个 Activity 的启动耗时；
TotalTime 表示新应用启动的耗时，包括新进程的启动和 Activity 的启动，但不包括前一个应用 Activity pause 的耗时。也就是说，开发者一般只要关心 TotalTime 即可，这个时间才是自己应用真正启动的耗时。

（1）窗口优化

Application 的启动优化

Application#attachBaseContext()

Application 启动会经过 attachBaseContext()->onCreate()；这时大家从 attachBaseContext 的生命周期联想到什么？没错就是 MultiDex 分包机制。想必大家都会发现，自从我们方法数超出了 65535 处理了分包之后，启动白屏/黑屏的问题就出现了，分包机制是导致冷启动缓慢的重要原因，而现在部分应用采用插件化的方式来避免 MultiDex 带来的白屏问题，这虽然是一种方法，但是开发成本实在高，对于不少应用来说是不必要的。我们来聊一下 MultiDex 优化，首先 MultiDex 可分成运行时和编译时两个部分：

编译期：将 App 中的 class 以某种策略拆分在多个 dex 中，为了减少第一个 dex 也就主 dex 中包含的 class 数；
运行期： App 启动时，虚拟机只加载主 dex 中的 class。app 启动以后，使用 Multidex.install，通过反射机制修改 ClassLoader 中的 dexElements 来加载其他 dex；

从网上的多篇实践分析中，他们主要采用的是异步方式。因为 App 起始会先加载主 dex 包，那么我们可以自主去处理分包的工作，我们将启动页和首页需要的库、组件等主要 class 分在主 dex 中,从而达到精分主 dex 包的大小,具体的操作写法,大家可以参考网上 MultiDex 启动优化文章，但是大家要注意在主 dex 的分包过程中，主 dex 经过我们一系列的优化操作减少了主 dex 的大小，因此也增大了 NoClassDefFoundError 的异常的可能，此时会导致我们的应用启动失败的风险，所以在优化后我们一定做好测试工作。

10.1 稳定--内存优化

内存抖动

内存抖动（代码注意事项）：

内存抖动是由于短时间内有大量对象进出新生区导致的，它伴随着频繁的 GC，gc 会大量占用 ui 线程和 cpu 资源，会导致 app 整体卡顿。

避免发生内存抖动的几点建议：

（1）尽量避免在循环体内创建对象，应该把对象创建移到循环体外。

（2）注意自定义 View 的 onDraw()方法会被频繁调用，所以在这里面不应该频繁的创建对象。

（3）当需要大量使用 Bitmap 的时候，试着把它们缓存在数组或容器中实现复用。

（4）对于能够复用的对象，同理可以使用对象池将它们缓存起来。

（1）Memory Monitor 工具：

它是 Android Studio 自带的一个内存监视工具，它可以很好地帮助我们进行内存实时分析。通过点击 Android Studio 右下角的 Memory Monitor 标签，打开工具可以看见较浅蓝色代表 free 的内存，而深色的部分代表使用的内存从内存变换的走势图变换，可以判断关于内存的使用状态，例如当内存持续增高时，可能发生内存泄漏；当内存突然减少时，可能发生 GC 等，如下图所示。

（2）LeakCanary 工具：

这个工具是 Square 公司在 Github 开源的。

首先，笔者仔细查看了 Leakcanary 官方的 github 仓库，最重要的便是对 **Leakcanary 是如何起作用的**（即原理）这一问题进行了阐述，我自己把它翻译成了易于理解的文字，主要分为如下 7 个步骤：

- 1、RefWatcher.watch()创建了一个 KeyedWeakReference 用于去观察对象。
- 2、然后，在后台线程中，它会检测引用是否被清除了，并且是否没有触发 GC。
- 3、如果引用仍然没有被清除，那么它将会把堆栈信息保存在文件系统中的.hprof 文件里。
- 4、HeapAnalyzerService 被开启在一个独立的进程中，并且 HeapAnalyzer 使用了 HAHA 开源库解析了指定时刻的堆栈快照文件 heap dump。
- 5、从 heap dump 中，HeapAnalyzer 根据一个独特的引用 key 找到了 KeyedWeakReference，并且定位了泄露的引用。
- 6、HeapAnalyzer 为了确定是否有泄露，计算了到 GC Roots 的最短强引用路径，然后建立了导致泄露的链式引用。
- 7、这个结果被传回到 app 进程中的 DisplayLeakService，然后一个泄露通知便展现出来了。

官方的原理简单来解释就是这样的：**在一个 Activity 执行完 onDestroy()之后，将它放入 WeakReference 中，然后将这个 WeakReference 类型的 Activity 对象与 ReferenceQueque 关联。这时再从 ReferenceQueque 中查看是否有没有该对象，如果没有，执行 gc，再次查看，还是没有的话则判断发生内存泄露了。最后用 HAHA 这个开源库去分析 dump 之后的 heap 内存。**

（3）Android Lint 工具：

10.2 流畅——卡顿优化

View 的绘制帧数保持 60fps 最佳，这要求没帧绘制时间不超过 16ms，如果不能在 16ms 内完成界面的渲染，那么就会出现卡顿的现象。

- UI 线程中做了耗时操作，导致 UI 线程卡顿
- 布局层次嵌套过多，过于复杂，无法在 16ms 内完成渲染
- 同一时间动画执行的次数过多，导致 CPU 和 GPU 负载过重
- overDraw，导致像素在同一帧的时间内被绘制多次
- view 频繁的触发 measure、layout，导致 measure、layout 类似耗时过多和整个 view 频繁重新渲染
- 频繁触发 GC，使得 16ms 无法完成绘制
- ANR

（1）布局优化

在 Android 种系统对 View 进行测量、布局和绘制时，都是通过对 View 数的遍历来进行操作的。如果一个 View 数的高度太高就会严重影响测量、布局和绘制的速度。Google 也

在其 API 文档中建议 View 高度不宜哦过 10 层。现在版本种 Google 使用 RelativeLayout 替代 LineraLayout 作为默认根布局，目的就是降低 LineraLayout 嵌套产生布局树的高度，从而提高 UI 渲染的效率。

- 布局复用，使用<include>标签重用 layout；
- 提高显示速度，使用<ViewStub>延迟 View 加载；
- 减少层级，使用<merge>标签替换父级布局；
- 注意使用 wrap_content，会增加 measure 计算成本；
- 删除控件中无用属性；

（3）启动优化

应用一般都有闪屏页 SplashActivity，优化闪屏页的 UI 布局，可以通过 Profile GPU Rendering 检测丢帧情况。

（4）优化工具

CPU Profile

10.3 节省——耗电优化

在 Android5.0 以前，关于应用电量消耗的测试即麻烦又不准确，而 5.0 之后 Google 专门引入了一个获取设备上电量消耗信息的 API—— Battery Historian。**Battery Historian** 是一款由 Google 提供的 Android 系统电量分析工具，直观地展示出手机的电量消耗过程，通过输入电量分析文件，显示消耗情况。

最后提供一些可供参考耗电优化的方法：

（1）计算优化。算法、for 循环优化、Switch..case 替代 if..else、避开浮点运算。

浮点运算：计算机里整数和小数形式就是按普通格式进行存储，例如 1024、3.1415926 等等，这个没什么特点，但是这样的数精度不高，表达也不够全面，为了能够有一种数的通用表示法，就发明了浮点数。浮点数的表示形式有点像科学计数法（*.*****×10^***），它的表示形式是 0.*****×10^***，在计算机中的形式为 .***** e ±***），其中前面的星号代表定点小数，也就是整数部分为 0 的纯小数，后面的指数部分是定点整数。利用这样的形式就能表示出任意一个整数和小数，例如 1024 就能表示成 0.1024×10^4，也就是 .1024e+004，3.1415926 就能表示成 0.31415926×10^1，也就是 .31415926e+001，这就是浮点数。浮点数进行的运算就是浮点运算。浮点运算比常规运算更复杂，因此计算机进行浮点运算速度要比进行常规运算慢得多。

（2）避免 Wake Lock 使用不当。

Wake Lock 是一种锁的机制，主要是相对系统的休眠而言的，，只要有人拿着这个锁，系统就无法进入休眠意思就是我的程序给 CPU 加了这个锁那系统就不会休眠了，这样做的目的是为了全力配合我们程序的运行。有的情况如果不这么做就会出现一些问题，比如微信等及时通讯的心跳包会在熄屏不久后停止网络访问等问题。所以微信里面是有大量使用到了 Wake_Lock 锁。系统为了节省电量，CPU 在没有任务忙的时候就会自动进入休眠。有任务需要唤醒 CPU 高效执行的时候，就会给 CPU 加 Wake_Lock 锁。大家经常犯的错误，我们很容易去唤醒 CPU 来工作，但是很容易忘记释放 Wake_Lock。

（3）使用 Job Scheduler 管理后台任务。

在 Android 5.0 API 21 中，google 提供了一个叫做 JobScheduler API 的组件，来处理当某个时间点或者当满足某个特定的条件时执行一个任务的场景，例如当用户在夜间休息时或设备接通电源适配器连接 WiFi 启动下载更新的任务。这样可以在减少资源消耗的同时提升应用的效率。

10.4 安装包——APK 瘦身

（1）安装包的组成结构

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。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。

（2）减少安装包大小

- 代码混淆。使用 IDE 自带的 proGuard 代码混淆器工具 ，它包括压缩、优化、混淆等功能。
- 资源优化。比如使用 Android Lint 删除冗余资源，资源文件最少化等。
- 图片优化。比如利用 PNG 优化工具 对图片做压缩处理。推荐目前最先进的压缩工具 Googlek 开源库 zopfli。如果应用在 4.0 版本以上，推荐使用 WebP 图片格式。

- 避免重复或无用功能的第三方库。例如，百度地图接入基础地图即可、讯飞语音无需接入离线、图片库 Glide\Picasso 等。
- 插件化开发。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。
- 可以使用微信开源资源文件混淆工具——AndResGuard 。一般可以压缩 apk 的 1M 左右大。

10.5 冷启动与热启动

app 冷启动： 当应用启动时，后台没有该应用的进程，这时系统会重新创建一个新的进程分配给该应用， 这个启动方式就叫做冷启动（后台不存在该应用进程）。冷启动因为系统会重新创建一个新的进程分配给它，所以会先创建和初始化 Application 类，再创建和初始化 MainActivity 类（包括一系列的测量、布局、绘制），最后显示在界面上。

app 热启动： 当应用已经被打开， 但是被按下返回键、Home 键等按键时回到桌面或者是其他程序的时候，再重新打开该 app 时， 这个方式叫做热启动（后台已经存在该应用进程）。热启动因为会从已有的进程中来启动，所以热启动就不会走 Application 这步了，而是直接走 MainActivity（包括一系列的测量、布局、绘制），所以热启动的过程只需要创建和初始化一个 MainActivity 就行了，而不必创建和初始化 Application

冷启动的流程

当点击 app 的启动图标时,安卓系统会从 Zygote 进程中 fork 创建出一个新的进程分配给该应用,之后会依次创建和初始化 Application 类、创建 MainActivity 类、加载主题样式 Theme 中的 windowBackground 等属性设置给 MainActivity 以及配置 Activity 层级上的一些属性、再 inflate 布局、当 onCreate/onStart/onResume 方法都走完了后最后才进行 contentView 的 measure/layout/draw 显示在界面上冷启动的生命周期简要流程：

Application 构造方法 -> attachBaseContext()->onCreate ->Activity 构造方法 -> onCreate() -> 配置主体中的背景等操作 ->onStart() -> onResume() -> 测量、布局、绘制显示

冷启动的优化主要是视觉上的优化，解决白屏问题，提高用户体验，所以通过上面冷启动的过程。能做的优化如下：

1、减少 onCreate()方法的工作量

2、不要让 Application 参与业务的操作

3、不要在 Application 进行耗时操作

4、不要以静态变量的方式在 Application 保存数据

5、减少布局的复杂度和层级

6、减少主线程耗时

10.6 内存泄漏的场景和解决办法

https://blog.csdn.net/carson_ho/article/details/79407707

10.6.1 非静态内部类的静态实例

非静态内部类会持有外部类的引用，如果非静态内部类的实例是静态的，就会长期的维持着外部类的引用，组织被系统回收，解决办法是使用静态内部类

10.6.2 多线程相关的匿名内部类和非静态内部类

匿名内部类同样会持有外部类的引用，如果在线程中执行耗时操作就有可能发生内存泄漏，导致外部类无法被回收，直到耗时任务结束，解决办法是在页面退出时结束线程中的任务

10.6.3 Handler 内存泄漏

Handler 导致的内存泄漏也可以被归纳为非静态内部类导致的，*Handler* 内部 *message* 是被存储在 *MessageQueue* 中的，有些 *message* 不能马上被处理，存在的时间会很长，导致 *handler* 无法被回收，如果 *handler* 是非静态的，就会导致它的外部类无法被回收，解决办法是 *1.*使用静态 *handler*，外部类引用使用弱引用处理 *2.*在退出页面时移除消息队列中的消息

10.6.4 Context 导致内存泄漏

根据场景确定使用 *Activity* 的 *Context* 还是 *Application* 的 *Context*,因为二者生命周期不同，对于不必须使用 *Activity* 的 *Context* 的场景（*Dialog*），一律采用 *Application* 的 *Context*，单例模式是最常见的发生此泄漏的场景，比如传入一个 *Activity* 的 *Context* 被静态类引用，导致无法回收

10.6.5 静态 View 导致泄漏

使用静态 *View* 可以避免每次启动 *Activity* 都去读取并渲染 *View*，但是静态 *View* 会持有 *Activity* 的引用，导致无法回收，解决办法是在 *Activity* 销毁的时候将静态 *View* 设置为 *null*（*View* 一旦被加载到界面中将会持有一个 *Context* 对象的引用，在这个例子中，这个 *context* 对象是我们的 *Activity*，声明一个静态变量引用这个 *View*，也就引用了 *activity*）

10.6.6 WebView 导致的内存泄漏

WebView 只要使用一次，内存就不会被释放，所以 *WebView* 都存在内存泄漏的问题，通常的解决办法是为 *WebView* 单开一个进程，使用 *AIDL* 进行通信，根据业务需求在合适的时机释放掉

10.6.7 资源对象未关闭导致

如 *Cursor*，*File* 等，内部往往都使用了缓冲，会造成内存泄漏，一定要确保关闭它并将引用置为 *null*

10.6.8 集合中的对象未清理

集合用于保存对象，如果集合越来越大，不进行合理的清理，尤其是入股集合是静态的

10.6.9 Bitmap 导致内存泄漏

bitmap 是比较占内存的，所以一定要在不使用的时候及时进行清理，避免静态变量持有大的 *bitmap* 对象

10.6.10 监听器未关闭

很多需要 *register* 和 *unregister* 的系统服务要在合适的时候进行 *unregister*,手动添加的 *listener* 也需要及时移除

10.7 Bitmap 优化

https://www.cnblogs.com/dasusu/p/9789389.html

（1）要选择合适的图片规格（bitmap 类型）

通常我们优化 Bitmap 时，当需要做性能优化或者防止 OOM，我们通常会使用 RGB_565，因为 ALPHA_8 只有透明度，显示一般图片没有意义，Bitmap.Config.ARGB_4444 显示图片不清楚， Bitmap.Config.ARGB_8888 占用内存最多。：

ALPHA_8 每个像素占用 1byte 内存

ARGB_4444 每个像素占用 2byte 内存

ARGB_8888 每个像素占用 4byte 内存（默认）

RGB_565 每个像素占用 2byte 内存

（2）降低采样率

BitmapFactory.Options 参数 inSampleSize 的使用，先把 options.inJustDecodeBounds 设为 true，只是去读取图片的大小，在拿到图片的大小之后和要显示的大小做比较通过 calculateInSampleSize()函数计算 inSampleSize 的具体值，得到值之后。

options.inJustDecodeBounds 设为 false 读图片资源。

（3）复用内存：

即通过软引用(内存不够的时候才会回收掉)，复用内

存块，不需要再重新给这个 bitmap 申请一块新的内存，避免了

一次内存的分配和回收，从而改善了运行效率。

（4）使用 recycle()方法及时回收内存。

（5）压缩图片。

Bitmap.recycle()会立即回收么？什么时候会回收？如果没有地方使用这个 Bitmap，为什么垃圾回收不会直接回收？

通过源码可以了解到，加载 Bitmap 到内存里以后，是包含**两部分内存区域**的。简单的说，一部分是 **Java 部分**的，一部分是 **C 部分**的。这个 Bitmap 对象是由 Java 部分分配的，不用的时候系统就会自动回收了但是那个对应的 **C 可用**的内存区域，虚拟机是不能直接回收的，这个只能调用底层的功能释放。所以需要调用 recycle()方法来释放 C 部分的内存 bitmap.recycle()方法用于回收该 Bitmap 所占用的内存，接着将 bitmap 置空，最后使用 System.gc()调用一下系统的垃圾回收器进行回收，调用 System.gc()并不能保证立即开始进行回收过程，而只是为了加快回收的到来。

10.8 LRU 的原理

为减少流量消耗，可采用缓存策略。常用的缓存算法是 LRU(Least Recently Used)：当缓存满时，会优先淘汰那些近期最少使用的缓存对象。主要是两种方式：

（1） LruCache(内存缓存)：

LruCache 类是一个线程安全的泛型类：内部采用一个 LinkedHashMap 以强引用的方式存储外界的缓存对象，并提供 get 和 put 方法来完成缓存的获取和添加操作，当缓存满时会移除较早使用的缓存对象，再添加新的缓存对象。

（2）DiskLruCache(磁盘缓存)：

通过将缓存对象写入文件系统从

而实现缓存效果

10.9 webview 优化

加载一个 webview 经历的过程如下：

1. webview 的初始化
2. 从服务器下载 html，css，js，image 等文件
3. 解析 html，构建 dom 树，布局绘制
4. 显示一个 webview 页面

优化点：

1.将 html，css，js，image 等资源文件放到本地，设计增量更新策略，对需要更新的部分进行 webpack 和 gzip 压缩和 cdn 加速处理，提神拉取速度。并且在进行网络链接时可以让前端请求的域名跟 API 域名保持一致，减少 DNS 解析时间。

2.通过 setBlockNetworkImage(boolean)来设置预先加载非图片部分的内容，延迟图片加载，在 onPageStart 屏蔽图片加载，在 onPageFinished 开启图片加载

3.

11 如何避免 OOM?

11.1 使用更加轻量的数据结构：

如使用 ArrayMap/SparseArray 替代 HashMap,HashMap 更耗内存，因为它需要额外的实例对象来记录 Mapping 操作，SparseArray 更加高效，因为它避免了 Key Value 的自动装箱，和装箱后的解箱操作

11.2 便面枚举的使用，可以用静态常量或者注解@IntDef 替代

11.3 Bitmap 优化:

（）1 尺寸压缩：通过 InSampleSize 设置合适的缩放

（2）颜色质量：设置合适的 format，ARGB_6666/RBG_545/ARGB_4444/ALPHA_6，存在很大差异

（3）inBitmap ：使用 inBitmap 属性可以告知 Bitmap 解码器去尝试使用已经存在的内存区域，新解码的 Bitmap 会尝试去使用之前那张 Bitmap 在 Heap 中所占据的 pixel data 内存区域，而不是去问内存重新申请一块区域来存放 Bitmap。利用这种特性，即使是上千张的图片，也只会仅仅只需要占用屏幕所能够显示的图片数量的内存大小，但复用存在一些限制，具体体现在：在 Android 4.4 之前只能重用相同大小的 Bitmap 的内存，而 Android 4.4 及以后版本则只要后来的 Bitmap 比之前的小即可。使用 inBitmap 参数前，每创建一个 Bitmap 对象都会分配一块内存供其使用，而使用了 inBitmap 参数后，多个 Bitmap 可以复用一块内存，这样可以提高性能

11.4 StringBuilder 替代 String:

在有些时候，代码中会需要使用到大量的字符串拼接的操作，这种时候有必要考虑使用 StringBuilder 来替代频繁的“+”

11.5 避免在类似 onDraw 这样的方法中创建对象，因为它会迅速占用大量内存，引起频繁的 GC 甚至内存抖动

11.6 减少内存泄漏也是一种避免 OOM 的方法

11 模式架构

11.1 MVP 模式

MVP 架构由 MVC 发展而来。在 MVP 中，M 代表 Model，V 代表 View，P 代表 Presenter。

模型层（Model）:主要是获取数据功能，业务逻辑和实体模型。

视图层（View）：对应于 Activity 或 Fragment，负责视图的部分展示和业务逻辑用户交互

控制层（Presenter）:负责完成 View 层与 Model 层间的交互，通过 P 层来获取 M 层中数据后返回给 V 层，使得 V 层与 M 层间没有耦合。

在 MVP 中 ，Presenter 层完全将 View 层和 Model 层进行了分离，把主要程序逻辑放在 Presenter 层实现，Presenter 与具体的 View 层（Activity）是没有直接的关联，是通过定义接口来进行交互的，从而使得当 View 层（Activity）发生改变时，Persenter 依然可以保持不变。View 层接口类只应该只有 set/get 方法，及一些界面显示内容和用户输入，除此之外不应该有多余的内容。绝不允许 View 层直接访问 Model 层，这是与 MVC 最大区别之处，也是 MVP 核心优点。

11.2 MVVM 模式

• Model

Model 层就是职责数据的存储、读取网络数据、操作数据库数据以及 I/O，一般会有一个 ViewModel 对象来调用获取这一部分的数据。

• View

我感觉这里的 View 才是真正的 View，为什么这么说？View 层做的仅仅和 UI 相关的工作，我们只在 XML、Activity、Fragment 写 View 层的代码，View 层不做和业务相关的事，也就是我们的 Activity 不写和业务逻辑相关代码，一般 Activity 不写更新 UI 的代码，如果非得要写，那更新的 UI 必须和业务逻辑和数据是没有关系的，只是单纯 UI 逻辑来更新 UI，比如：滑动时头部颜色渐变、editttext 根据输入内容显示隐藏等，简单的说：View 层不做任何业务逻辑、不涉及操作数据、不处理数据、UI 和数据严格的分开。

• ViewModel

ViewModel 只做和业务逻辑和业务数据相关的事,不做任何和 UI、控件相关的事,ViewModel 层不会持有任何控件的引用,更不会在 ViewModel 中通过 UI 控件的引用去做更新 UI 的事情。ViewModel 就是专注于业务的逻辑处理，操作的也都是对数据进行操作，这些个数据源绑定在相应的控件上会自动去更改 UI，开发者不需要关心更新 UI 的事情。

12 虚拟机

Android Dalvik 虚拟机和 ART 虚拟机对比

12.1 Dalvik

Android4.4 及以前使用的都是 Dalvik 虚拟机，我们知道 Apk 在打包的过程中会先将 java 等源码通过 javac 编译成.class 文件，但是我们的 Dalvik 虚拟机只会执行.dex 文件，这个时候 dx 会将.class 文件转换成 Dalvik 虚拟机执行的.dex 文件。Dalvik 虚拟机在启动的时候会先将.dex 文件转换成快速运行的机器码，又因为 65535 这个问题，导致我们在应用冷启动的时候有一个合包的过程，最后导致的一个结果就是我们的 app 启动慢，这就是 Dalvik 虚拟机的 JIT 特性（Just In Time）。

12.2 ART

ART 虚拟机是在 Android5.0 才开始使用的 Android 虚拟机，ART 虚拟机必须要兼容 Dalvik 虚拟机的特性，但是 ART 有一个很好的特性 AOT（ahead of time），这个特性就是我们在安装 APK 的时候就将 dex 直接处理成可直接供 ART 虚拟机使用的机器码，ART 虚拟机将.dex 文件转换成可直接运行的.oat 文件，ART 虚拟机天生支持多 dex，所以也不会有一个合包的过程，所以 ART 虚拟机会很大的提升 APP 冷启动速度。

ART 优点：

加快 APP 冷启动速度

提升 GC 速度

提供功能全面的 Debug 特性

ART 缺点：

APP 安装速度慢，因为在 APK 安装的时候要生成可运行.oat 文件

APK 占用空间大，因为在 APK 安装的时候要生成可运行.oat 文件

12.3 处理器

Android 支持三种处理器类型：

1. x86 体系，目前很少
2. arm 体系，分为 32 位和 64 位，向下兼容，32 位的可以运行在 64 位上面
3. mips 体系，目前很少

（1）armeabi/armeabi-v7a，主要用于 Android4.0 之后，cpu 是 32 位的，缺少对浮点数计算的硬件支持，基本被淘汰。

（2）Arm64-v8a，主要用于 Android5.0 之后，cpu 是 64 位的。

兼容性：

Android 上启动一个 app，都会为它创建一个虚拟机。Android64 位的系统加载 32 位的 so 文件或者 app 时，会在创建一个 64 位的虚拟机的同时还会创建一个 32 位的虚拟机，这样就能兼容 32 位的 app 应用。

12.3.1 so 加载流程

手机支持的 cpu 型号和种类存放在一个 abiList 集合中，有先后顺序，按照这个顺序遍历 jniLib 目录，如果这个目录下面有 arm64-v8a 子目录且里面有 so 文件，那么手机将加载这个目录下面的 so 文件就不会再去加载其他目录下面的 so 文件。

12.3.2 so 加载算法

假如 jniLib 目录下面存在 arm64-v8a 目录下有 a.so 文件,armeabi-v7a 下面有 a.so 和 b.so 文件，手机只会加载 arm64-v8a 下的 a.so 文件，永远不会加载 b.so 文件，所以就会报找不到 so 文件的异常。

方案：

为了节省 apk 的体积，可以只保存一份 so 文件，那就是 armeabi-v7a 下的 so 文件：

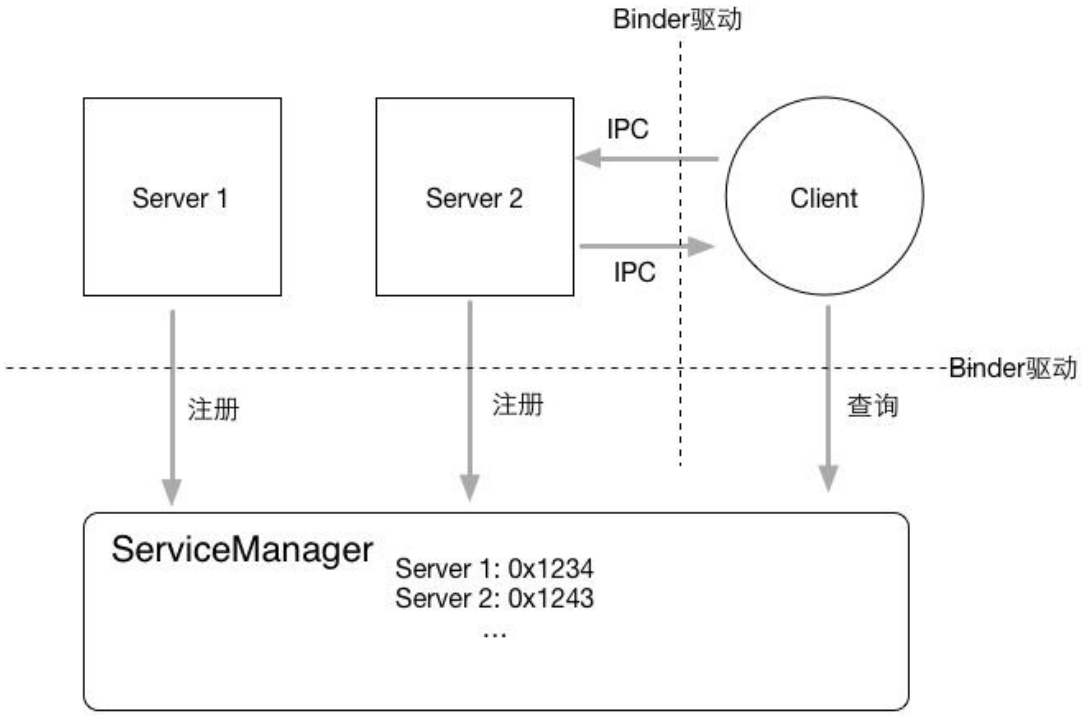
（1）32 位的 arm 手机肯定能加载到 armeabi-v7a 下的 so 文件

（2）64 位的 arm 手机要想加载 32 位的 so 文件，千万不要在 arm64-v8a 目录下面放置任何 so 文件。都放在 armeabi-v7a 下面即可。

13 Binder

13.1 Binder 组成

Binder 分为 client 进程和 server 进程。



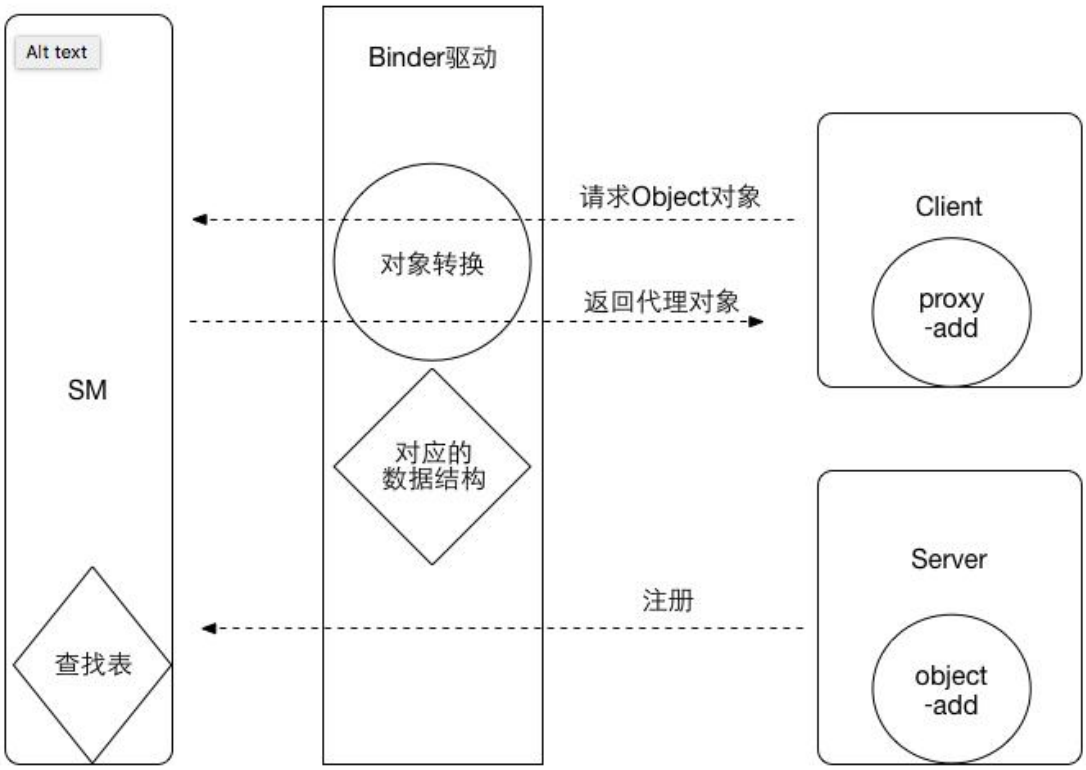
图中的 IPC 就是进程间通信的意思。

图中的 ServiceManager，负责把 Binder Server 注册到一个容器中。

有人把 ServiceManager 比喻成电话局，存储着每个住宅的座机电话，还是很恰当的。张三给李四打电话，拨打电话号码，会先转接到电话局，电话局的接线员查到这个电话号码的地址，因为李四的电话号码之前在电话局注册过，所以就能拨通；没注册，就会提示该号码不存在。

对照着 Android Binder 机制，对着上面这图，张三就是 Binder Client，李四就是 Binder Server，电话局就是 ServiceManager，电话局的接线员在这个过程中做了很多事情，对应着图中的 Binder 驱动

13.2 Binder 通信过程



通信流程：

（1）首先 Server 需要在 SM 容器中注册

（2）其次，Client 调用 Server 的 add 方法时，需要获取 Server 的对象，SM 不会把 Server 的对象返回给 Client，只是返回其一个代理对象 Proxy

（3）最后，Client 调用 Proxy 对象的 add 方法，SM 会帮它调用 Client 的 add 方法并把结果返回给 Client。

14 AIDL

AIDL 即 Android Interface Definition Language，接口定义语言，用于定义服务器和客户端通信接口的一种描述语言。

（1）IBinder

（2）IInterface

（3）Binder

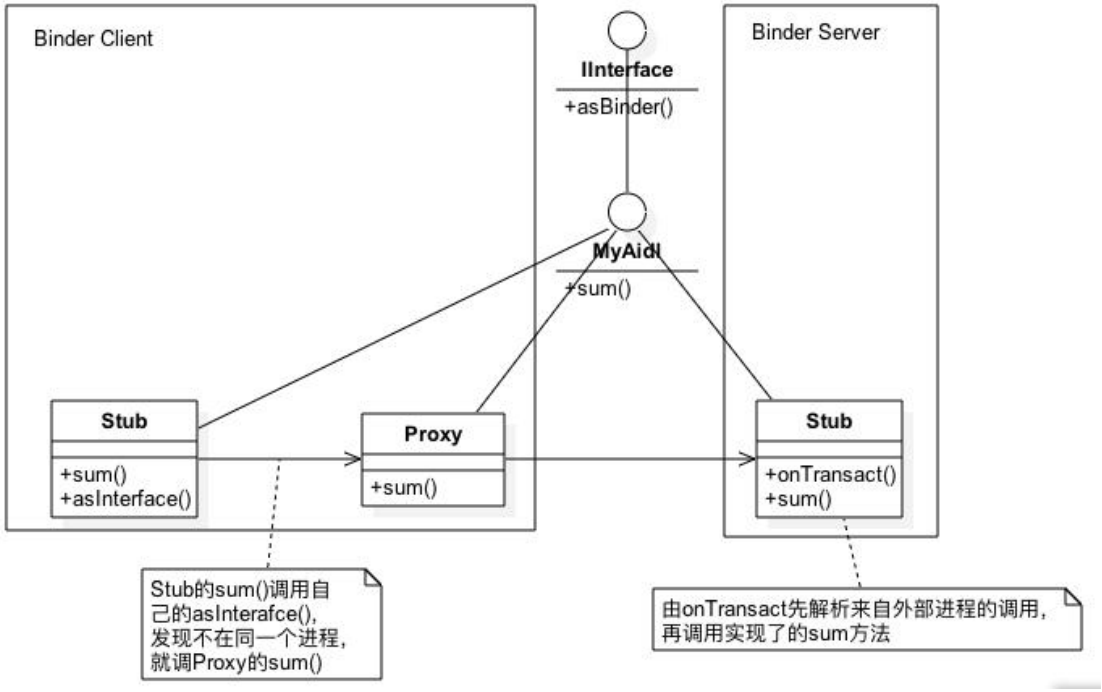
（4）Proxy

（5）Stub

```java
public interface MyAidl extends android.os.IInterface {
    public int sum(int a,int b);
}
```

```java
public abstract  class Stub extends android.os.Binder implements MyAidl{

    public Stub(){

    }

    public MyAidl asInterface(IBinder obj){
        //如果是在同一个进程，则直接返回
        return (MyAidl)obj;

        //不在同一个进程，则调用
        return new MyAidl.Stub.Proxy(obj);
    }

    @Override
    public boolean onTransact(int code,Parcel data,Parcel reply,int flags){
        switch (code){
            case TRANSACTION_sum:

                int _arg0=data.readInt();
                int _arg1=data.readInt();

                int _result=this.sum(_arg0,_arg1);
                reply.writeInt(_result);
                return true;
        }

        return super.onTransact(code,data,reply,flags);
    }
}
```

```java
public class Proxy implements MyAidl{

    private IBinder mRemote;
    public Proxy(IBinder remote){
        this.mRemote=remote;
    }

    @Override
    public int sum(int a,int b){
        Parcel _data=Parcel.obtain();
        Parcel _reply=Parcel.obtain();

        int _result;

        _data.writeInt(a);
        _data.writeInt(b);
        mRemote.transact(Stub.TRANSACTION_sum,_data,_reply,0);
        _result=_reply.readInt();

        return _result;
    }
}
```



调用方式：

MyAidl.Stub.asInterface(IBinder 对象).sum(1,2);

（1）判断上面的 IBinder 对象和自己是否在同一个进程，在的话直接使用，不在的话则需要把 IBinder 参数包装成一个 Proxy 对象，这时就相当于调用 Proxy 的 sum 方法。

（2）在 Proxy 的 sum 方法中，会使用 Parcelable 来准备数据将函数名称和函数参数写入_data，让_reply 接收函数返回值，最后使用 IBinder 的 transact 方法把数据转给 Binder 的 Server 端。

（3）Server 则通过 onTransact 方法接收 Client 进程传过来的数据，包括函数名称，函数参数，然后调用相关方法，得到结果返回。

15 Linux 进程间通信方式

15.1 管道

管道是由内核管理的一个缓冲区，它是一种环形的数据结构。大小有限，所以不适合 Android 大量的进程间通信。

15.2 消息队列

消息队列提供了一个进程向另外一个经常发送一个数据块的方法。

15.3 socket

15.4 共享内存

15.5 信号量

15.6 Android 为何不采用 Linux 的通信方式而用 binder

15.6.1 安全性

传统的 LinuxIPC 通信方式无法获取进程的 UID 和 PID，从而无法鉴别对方身份，Android 为每个安装好的应用程序分配了自己的 UID，故进程的 UID 是鉴别进程 身份的重要标志。前面提到 C/S 架构，Android 系统中对外只暴露 Client 端， Client 端将任务发送给

Server 端，Server 端会根据权限控制策略，判断 UID/PID 是否满足访问权限，目前权限控制很多时候是通过弹出权限询问对话框，让用 户选择是否运行。

Android 系统中也依然采用了大量 Linux 现有的 IPC 机制，根据每类 IPC 的原理特性，因时制宜，不同场景特性往往会采用其下 最适宜的。比如在 Android OS 中的 Zygote 进程的 IPC 采用的是 Socket（套 接字）机制，Android 中的 **Kill Process 采用的 signa（l 信号）机制**等等。而 **Binder 更多则用在 system_server 进程与上层 App 层的 IPC 交互**。

15.6.2 稳定性

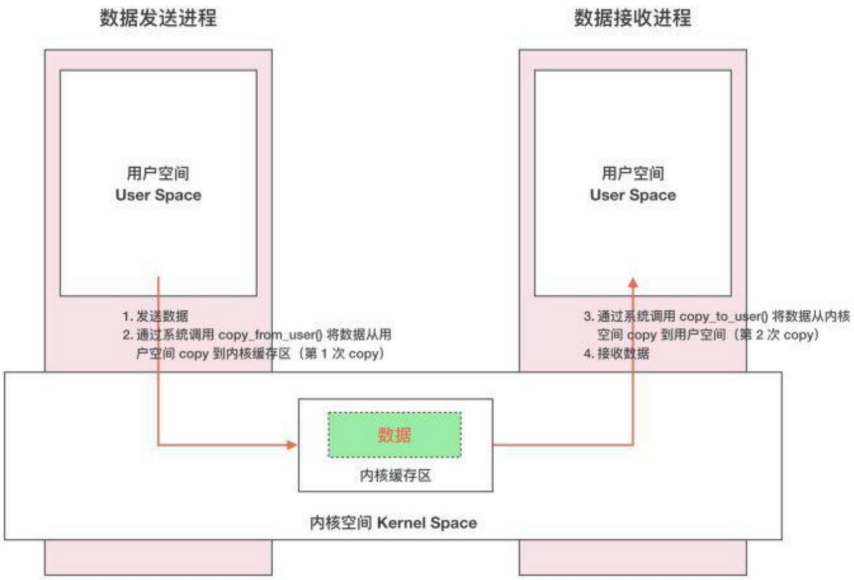
Binder 是基于 C/S 架构的,简单解释下 C/S 架构,是指客 户端(Client)和服务端(Server)组成的架构,Client 端有什么需求，直接发送给 Server 端去完成，架构清晰明朗，Server 端与 Client 端相对独立，稳定性较好；

而共享内存实现方式复杂，没有客户与服务端之别， 需要充分考虑到访问临界资源的并发同步问题，否则可能会出现死锁等问题；从这稳定性角度看，Binder 架构优越于共享内存。

15.6.3 性能

binder 性能仅次于共享内存，共享内存不需要拷贝，而管道，消息队列，socket 都需要拷贝两次

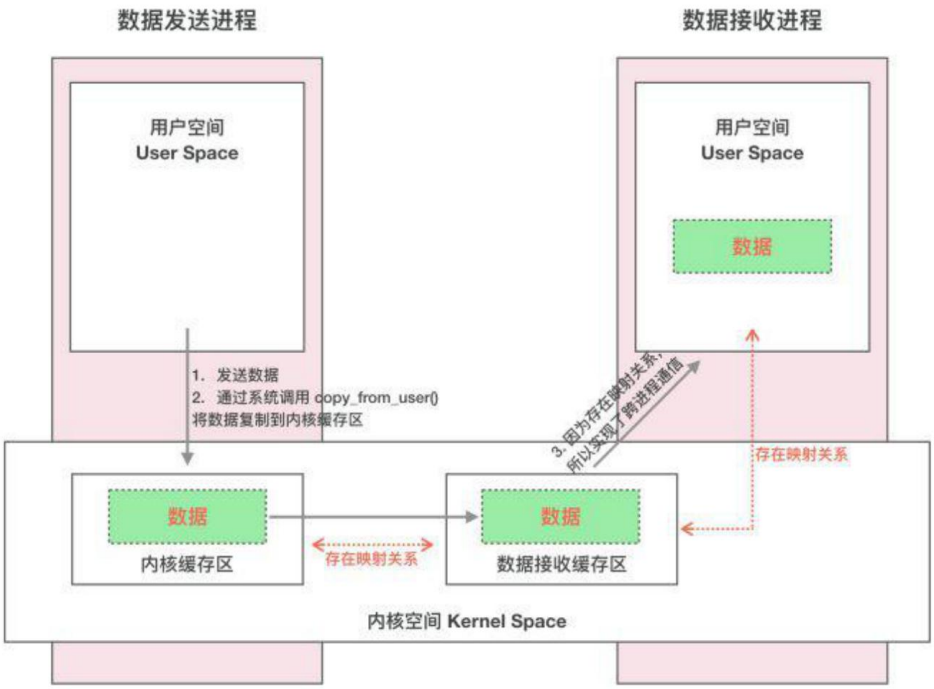
15.7 linux 数据拷贝



15.7.1 内存映射 mmap

一次完整的 Binder IPC 通信过程通常是这样：

1. 首先 Binder 驱动在内核空间创建一个数据接收缓存区；
2. 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；
3. 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。



16 API 版本

android3.0 代号 Honeycomb，引入 Fragments，ActionBar,属性动画，硬件加速

android4.0 代号 Ice Cream，API14：截图功能，人脸识别，虚拟按键，3D 优化驱动

android5.0 代号 Lollipop API21：调整桌面图标及部件透明度等

android6.0 代号 M Marshmallow API23，软件权限管理，安卓支付，指纹支持，App 关联，

android7.0 代号 N Preview API24，多窗口支持(不影响 Activity 生命周期)，增加了 JIT 编译器，引入了新的应用签名方案 APK Signature Scheme v2（缩短应用安装时间和更多未授权 APK 文件更改保护），严格了权限访问

android8.0 代号 O API26，取消静态广播注册，限制后台进程调用手机资源，桌面图标自适应

android9.0 代号 P API27,加强电池管理,系统界面添加了 Home 虚拟键,提供人工智能 API，支持免打扰模式