

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：

1. Activity

1.1. Activity的启动流程

Activity跨进程启动

<https://juejin.im/post/6844903959581163528#heading-1>

<http://gityuan.com/2016/03/12/start-activity/>

启动流程：

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

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

Zygote进程fork出新的子进程，即App进程；

App进程，通过Binder IPC向system_server进程发起attachApplication请求；

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；

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；

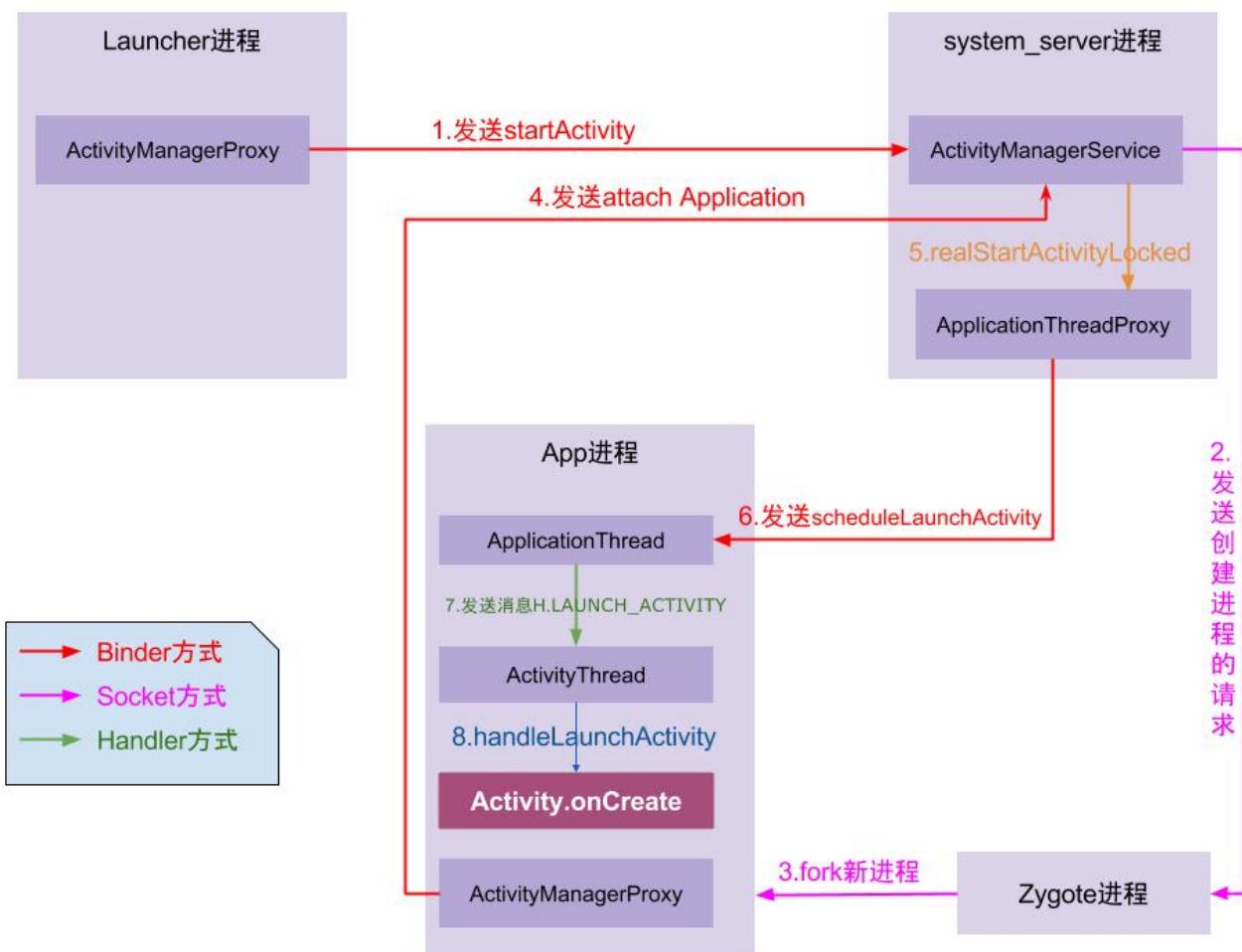
主线程在收到Message后，通过发射机制创建目标Activity，并回调Activity.onCreate()等方法。

Activity进程内启动

请求进程A：startActivity—(hook插入点1) (AMP, ActivityManager代理对象)——>
system_server进程：AMS(ActivityManagerService)

解析Activity信息、处理启动参数、scheduleLaunchActivity/mH中EXECUTE_TRANSACTION消息处理(Android P)-->

回到请求进程A：ApplicationThread --> ActivityThread -(hook插入点2)-> Activity生命周期



()

1.2.onSaveInstanceState(),onRestoreInstanceState的掉用时机

1.2.1.onSaveInstanceState(Bundle outState)会在以下情况被调用：

- 1.2.1.1、从最近应用中选择运行其他的程序时。
- 1.2.1.2、当用户按下HOME键时。
- 1.2.1.3、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)。
- 1.2.1.4、按下电源按键 (关闭屏幕显示) 时。
- 1.2.1.5、从当前activity启动一个新的activity时。

onPause -> onSaveInstanceState -> onStop.

1.2.2.onRestoreInstanceState(Bundle outState)会在以下情况被调用：

onRestoreInstanceState(Bundle savedInstanceState)只有在activity确实是被系统回收，重新创建activity的情况下才会被调用。

- 1.2.2.1.屏幕方向切换时，activity生命周期如下

onPause -> onSaveInstanceState -> onStop -> onDestroy -> onCreate -> onStart -> onRestoreInstanceState -> onResume

1.2.2.2.在后台被回收

1.2.2.3.按HOME键返回桌面,又马上点击应用图标回到原来页面时不会被回收

onStart -> onRestoreInstanceState -> onResume

1.2.3源码

系统会调用ActivityThread的performStopActivity方法中掉用onSaveInstanceState，将状态保存在mActivities中，mActivities维护了一个Activity的信息表，当Activity重启时候，会从mActivities中查询到对应的ActivityClientRecord。

如果有信息，则调用Activity的onResoreInstanceState方法，

在ActivityThread的performLaunchActivity方法中，统会判断ActivityClientRecord对象的state是否为空

不为空则通过Activity的onSaveInstanceState获取其UI状态信息，通过这些信息传递给Activity的onCreate方法，

1.3.activity的启动模式和使用场景

1.3.1 android任务栈

我们每次打开一个新的Activity或者退出当前Activity都会在一个称为任务栈的结构中添加或者减少一个Activity组件，一个任务栈包含了一个activity的集合。

android通过ActivityRecord、TaskRecord、ActivityStack，ActivityStackSupervisor，ProcessRecord有序地管理每个activity。

1.3.2 Standard

默认模式，每次启动Activity都会创建一个新的Activity实例。

1.3.3 SingleTop

通知消息打开的页面

如果要启动的Activity已经在栈顶，则不会重新创建Activity，只会调用该该Activity的onNewIntent()方法。

如果要启动的Activity不在栈顶，则会重新创建该Activity的实例。

1.3.4 SingleTask

主界面

如果要启动的Activity已经存在于它想要归属的栈中，那么不会创建该Activity实例，将栈中位于该Activity上的所有的Activity出栈，同时该Activity的onNewIntent()方法会被调用。

1.3.5SingleInstance

呼叫来电界面

要创建在一个新栈，然后创建该Activity实例并压入新栈中，新栈中只会存在这一个Activity实例。

1.4.Activity A跳转Activity B，再按返回键，生命周期执行的顺序

https://www.sohu.com/a/402329833_611601

<https://www.jianshu.com/p/6d9d830a758d>

在A跳转B会执行: A onPause -> B onCreate -> B onStart -> B onResume->A onStop

在B按下返回键会执行: B onPause -> A onRestart -> A onStart -> A onResume-> B onStop -> B onDestroy

当A跳转到B的时候, A先执行onPause, 然后居然是B再执行onCreate -> onStart -> onResume, 最后才执行A的onStop!!!

当B按下返回键, B先执行onPause, 然后居然是A再执行onRestart -> onStart -> onResume, 最后才是B执行onStop -> onDestroy!!!

当 B Activity 的 launchMode 为 singleInstance, singleTask 且对应的 B Activity 有可复用的实例时, 生命周期回调是这样的:

A.onPause -> B.onNewIntent -> B.onRestart -> B.onStart -> B.onResume -> A.onStop -> (如果 A 被移出栈的话还有一个 A.onDestroy)

当 B Activity 的 launchMode 为 singleTop 且 B Activity 已经在栈顶时 (一些特殊情况如通知栏点击、连点) , 此时只有 B 页面自己有生命周期变化:

B.onPause -> B.onNewIntent -> B.onResume

1.5. 横竖屏切换,按home键,按返回键,锁屏与解锁屏幕,跳转透明Activity界面,启动一个 Theme 为 Dialog 的 Activity, 弹出Dialog时Activity的生命周期

横竖屏切换:

从 Android 3.2 (API级别 13)开始

<https://www.jianshu.com/p/dbc7e81aead2>

1、不设置Activity的androidconfigChanges, 或设置Activity的androidconfigChanges="orientation", 或设置Activity的android:configChanges="orientation|keyboardHidden", 切屏会重新调用各个生命周期, 切横屏时会执行一次, 切竖屏时会执行一次。

2、配置 android:configChanges="orientation|keyboardHidden|screenSize", 才不会销毁 activity, 且只调用onConfigurationChanged方法。

竖屏:

启动: onCreate->onStart->onResume.

切换横屏时:

onPause-> onSaveInstanceState ->onStop->onDestroy

onCreate->onStart->onSaveInstanceState->onResume.

如果配置这个属性: androidconfigChanges="orientation | keyboardHidden | screenSize"

就不会在调用Activity的生命周期, 只会调用onConfigurationChanged方法

HOME键的执行顺序: onPause->onStop->onRestart->onStart->onResume

BACK键的顺序: onPause->onStop->onDestroy->onCreate->onStart->onResume

锁屏: 锁屏时只会调用onPause(), 而不会调用onStop方法, 开屏后则调用onResume()。

弹出 Dialog: 直接是通过 WindowManager.addView 显示的 (没有经过 AMS) , 所以不会对生命周期有任何影响。

启动theme为DialogActivity,跳转透明Activity

A.onPause -> B.onCreate -> B.onStart -> B.onResume

(Activity 不会回调 onStop, 因为只有在 Activity 切到后台不可见才会回调 onStop)

https://www.sohu.com/a/402329833_611601

1.6.onStart 和 onResume、 onPause 和 onStop 的区别

onStart 和 onResume 从 Activity 可见可交互区分

onStart 用户可以看到部分activity但不能与它交互 onResume()可以获得activity的焦点，能够与用户交互

onStop 和 onPause 从 Activity 是否位于前台，是否有焦点区分

onPause表示当前页面失去焦点。

onStop表示当前页面不可见。

dialog的主题页面，这个时候，打开着一个页面，就只会执行onPause，而不会执行onStop。

1.7.Activity之间传递数据的方式Intent是否有大小限制，如果传递的数据量偏大，有哪些方案

startActivity->startActivityForResult->Instrumentation.execStartActivity

->ActivityManager.getService().startActivity

intent中携带的数据要从APP进程传输到AMS进程，再由AMS进程传输到目标Activity所在进程

通过Binder来实现进程间通信

1.Binder 驱动在内核空间创建一个数据接收缓存区。

2.在内核空间开辟一块内核缓存区，建立内核缓存区和内核空间的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 到内核空间的内核缓存区，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。

为当使用Intent来传递数据时，用到了Binder机制，数据就存放在了Binder的事务缓冲区里面，而事务缓冲区是有大小限制的。普通的由Zygote孵化而来的用户进程，映射的Binder内存大小是不到1M的

Binder 本身就是为了进程间频繁-灵活的通信所设计的，并不是为了拷贝大量数据

如果非 ipc

单例, eventBus, Application, sqlite、 shared preference、 file 都可以;

如果是 ipc

1.共享内存性能还不错，通过 MemoryFile 开辟内存空间，获得 FileDescriptor；将 FileDescriptor 传递给其他进程；往共享内存写入数据；从共享内存读取数据。<https://www.jianshu.com/p/4a4bc36000fc>

2.Socket或者管道性能不太好，涉及到至少两次拷贝。

1.8.Activity的onNewIntent()方法什么时候执行

如果IntentActivity处于任务栈的顶端，也就是说之前打开过的Activity，现在处于onPause、onStop状态的话，其他应用再发送Intent的话，执行顺序为：onNewIntent, onRestart, onStart, onResume。

ActivityA已经启动过，处于当前应用的Activity堆栈中；

当ActivityA的LaunchMode为SingleTop时，如果ActivityA在栈顶，且现在要再启动ActivityA，这时会调用onNewIntent()方法；

当ActivityA的LaunchMode为SingleInstance,SingleTask时，如果已经ActivityA已经在堆栈中，那么此时再次启动会调用onNewIntent()方法；

1.9.显示启动和隐式启动

显示启动

1、构造方法传入Component，最常用的方式

2、setComponent(componentName)方法

3、setClass/setClassName方法

隐式启动

<https://www.jianshu.com/p/12c6253f1851>

隐式Intent是通过在AndroidManifest文件中设置action、data、category，让系统来筛选出合适的Activity

action的匹配规则

Intent-filter action可以设置多条

intent中的action只要与intent-filter其中的一条匹配成功即可，且intent中action最多只有一条

Intent-filter内必须至少包含一个action。

category的匹配规则

Intent-filter内必须至少包含一个category，android:name为android.intent.category.DEFAULT。

intent-filter中，category可以有多条

intent中，category也可以有多条

intent中所有的category都可以在intent-filter中找到一样的（包括大小写）才算匹配成功

data的匹配规则

intent-filter中可以设置多个data

intent中只能设置一个data

intent-filter中指定了data，intent中就要指定其中的一个data

1.10.scheme使用场景,协议格式,如何使用

scheme是一种页面内跳转协议，是一种非常好的实现机制，通过定义自己的scheme协议，可以非常方便跳转app中的各个页面

APP根据URL跳转到另外一个APP指定页面

可以通过h5页面跳转app原生页面

服务器可以定制化跳转app页面

Scheme链接格式样式

样式[scheme://host/path?query](#).

Uri.parse("hr://test:8080/goods?goodsId=8897&name=test")

hr代表Scheme协议名称

test代表Scheme作用的地址域

8080代表改路径的端口号

/goods代表的是指定页面(路径)

goodsId和name代表传递的两个参数

使用

```
<intent-filter>
    <!-- 协议部分配置 ,注意需要跟web配置相同-->
    <!--协议部分, 随便设置 hr://test:8080/goods?name=test -->
    <data android:scheme="hr"
        android:host="test"
        android:path="/goods"
        android:port="8080"/>
    <!--下面这几行也必须得设置-->
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <action android:name="android.intent.action.VIEW" />
</intent-filter>
```

掉用

```
Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse("hr://test:8080/goods?name=test"));
startActivity(intent);
```

1.11.ANR 的四种场景

ANR 的四种场景：

Service TimeOut: service 未在规定时间执行完成：前台服务 20s，后台 200s

BroadCastQueue TimeOut: 未在规定时间内未处理完广播：前台广播 10s 内, 后台 60s 内

ContentProvider TimeOut: publish 在 10s 内没有完成

Input Dispatching timeout: 5s 内未响应键盤输入、触摸屏幕等事件

我们可以看到， Activity 的生命周期回调的阻塞并不在触发 ANR 的场景里面，所以并不会直接触发 ANR。

只不过死循环阻塞了主线程，如果系统再有上述的四种事件发生，就无法在相应的时间内处理从而触发 ANR。

1.12.onCreate和onRestoreInstanceState方法中恢复数据时的区别

onSaveInstanceState 不一定会被调用，因为它只有在上次activity被回收了才会调用。

onCreate()里的Bundle参数可能为空，一定要做非空判断。而onRestoreInstanceState的Bundle参数一定不会是空值。

1.13.activity间传递数据的方式

通过 Intent 传递 (Intent.putExtra 的内部也是维护的一个 Bundle，因此，通过 putExtra 放入的数据，取出时也可以通过 Bundle 去取)

通过全局变量传递

通过 SharedPreferences 传递

通过数据库传递

通过文件传递

1.14.跨App启动Activity的方式,注意事项

<https://www.jianshu.com/p/ad01ac11b4f1>

<https://juejin.im/post/6844904056461197326#heading-0>

使用intentFilter(隐式跳转)

在Manifest的Activity标签中添加：

启动时：startActivity(new Intent("com.example.test.action.BActivity"))

如果有两个action属性值相同的Activity，那么在启动时手机系统会让你选择启动哪一个Activity

要解决这个问题，需要给被启动的Activity再加上一个属性，

然后再启动该Activity的Intent中加上一个URI，其中“app”必须与data属性的scheme的值一样，

```
intent=new Intent("com.zs.appb.intent.action.BeStartActivity", Uri.parse("app://hello"));
```

共享uid的App

android中uid用于标识一个应用程序，uid在应用安装时被分配，并且在应用存在于手机上期间，都不会改变。一个应用程序只能有一个uid，多个应用可以使用sharedUserId 方式共享同一个uid，前提是这些应用的签名要相同。

在AndroidManifest中：manifest标签中添加android:sharedUserId="xxxx"

启动时：startActivity(new Intent().setComponent(new ComponentName("com.example.test", "com.example.test.XxxActivity")));

使用exported

一旦设置了intentFilter之后，exported就默认被设置为true了

在Manifest中添加exported属性

启动时：startActivity(new Intent().setComponent(new ComponentName("com.example.zhu","com.example.zhu.XxxActivity")));

注意(如何防止自己的Activity被外部非正常启动):

如果AppB设置了android:permission="xxx.xxx.xx"那么，就必须在你的AppA的AndroidManifest.xml中uses-permission xxx.xxx.xx才能访问人家的东西。

给AppA的manifest中添加权限：

给AppB中需要启动的Activity添加permission属性：

android:permission="com.example.test"

1.15.Activity任务栈是什么

1.android任务栈又称为Task，它是一个栈结构，具有后进先出的特性，用于存放我们的Activity组件。

2.我们每次打开一个新的Activity或者退出当前Activity都会在一个称为任务栈的结构中添加或者减少一个Activity组件，一个任务栈包含了一个activity的集合，只有在任务栈栈顶的activity才可以跟用户进行交互。

3.在我们退出应用程序时，必须把所有的任务栈中所有的activity清除出栈时，任务栈才会被销毁。当然任务栈也可以移动到后台，并且保留了每一个activity的状态。可以有序的给用户列出它们的任务，同时也不会丢失Activity的状态信息。

4.对应AMS中的ActivityRecord、TaskRecord、ActivityStack(AMS中的总结)

1.16.有哪些Activity常用的标记位Flags

FLAG_ACTIVITY_NEW_TASK

此标记位作用是为Activity指定“singleTask”启动模式，其效果和在XML中指定相同
android:launchMode="singleTask"

FLAG_ACTIVITY_SINGLE_TOP

此标记位作用是为Activity指定“singleTop”启动模式，其效果和在XML中指定相同
android:launchMode="singleTop"

FLAG_ACTIVITY_CLEAR_TOP

具有此标记位的Activity，当它启动时，在同一个任务栈中位于它上面的Activity都要出栈。此标记位一般会和singleTask启动模式一起出现，此情况下，若被启动的Activity实例存在，则系统会调用它的onNewIntent。

1.17.Activity的数据是怎么保存的，进程被Kill后，保存的数据怎么恢复的

<https://www.wanandroid.com/wenda/show/12574>

在Activity的onSaveInstanceState方法回调时，put到参数outState (Bundle) 里面。outState就是ActivityClientRecord的state。

ActivityClientRecord实例，都存放在ActivityThread的mActivities里面。

Activity变得不可见时（onSaveInstanceState和onStop回调之后），在应用进程这边会通过ActivityTaskManagerService的activityStopped方法，把刚刚在onSaveInstanceState中满载了数据的Bundle对象，传到系统服务进程那边！然后（在系统服务进程这边），会进一步将这个Bundle对象，赋值到对应ActivityRecord的icicle上

ActivityRecord是用来记录对应Activity的各种信息的，如theme，启动模式、当前是否可见等等（为了排版更简洁，上图只列出来一个icicle），它里面还有很多管理Activity状态的相关方法；

TaskRecord就是大家耳熟能详的任务栈（从上图可以看出并不真的是栈）了，它的主要职责就是管理ActivityRecord。每当Activity启动时，会先找到合适的TaskRecord（或创建新实例），然后将该Activity所对应的ActivityRecord添加到TaskRecord的mActivities中；

ActivityStack管理着TaskRecord，当新TaskRecord被创建后，会被添加到它mTaskHistory里面。

2.Service

2.1.service 的生命周期，两种启动方式的区别

startService

onCreate() -> onStartCommand() -> onDestroy()

bindService

onCreate() -> onBind() -> onUnbind() -> onDestroy()

区别

启动

如果服务已经开启，多次执行startService不会重复的执行onCreate()，而是会调用onStart()和onStartCommand()。

如果服务已经开启，多次执行bindService时，onCreate和onBind方法并不会被多次调用

销毁

当执行stopService时，直接调用onDestroy方法

调用者调用unbindService方法或者调用者Context不存在了（如Activity被finish了），Service就会调用onUnbind->onDestroy

使用startService()方法启用服务，调用者与服务之间没有关系，即使调用者退出了，服务仍然运行。

使用bindService()方法启用服务，调用者与服务绑定在了一起，调用者一旦退出，服务也就终止。

1、单独使用startService & stopService

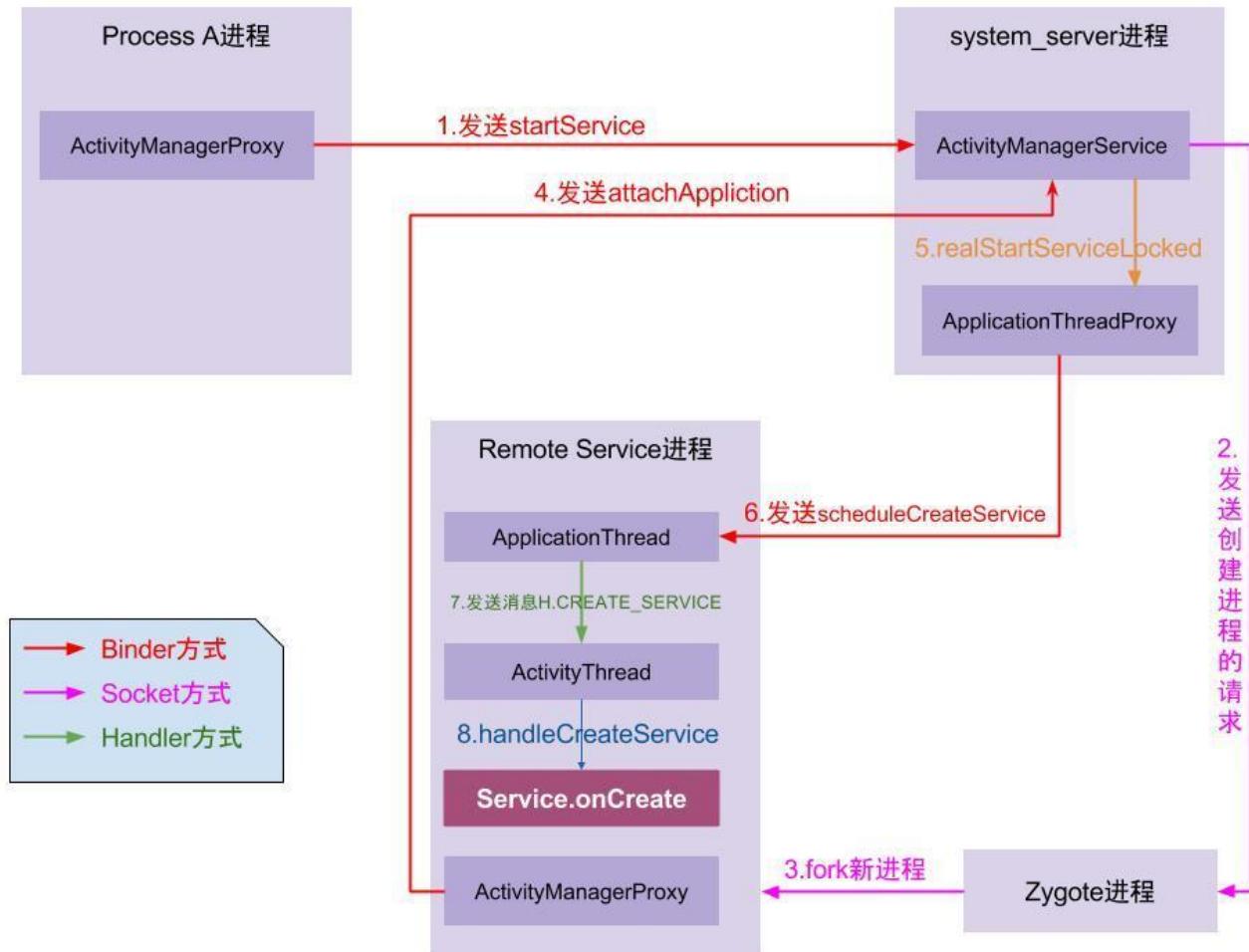
- (1) 第一次调用startService会执行onCreate、onStartCommand。
- (2) 之后再多次调用startService只执行onStartCommand，不再执行onCreate。
- (3) 调用stopService会执行onDestroy。

2、单独使用bindService & unbindService

- (1) 第一次调用bindService会执行onCreate、onBind。
- (2) 之后再多次调用bindService不会再执行onCreate和onBind。
- (3) 调用unbindService会执行onUnbind、onDestroy。

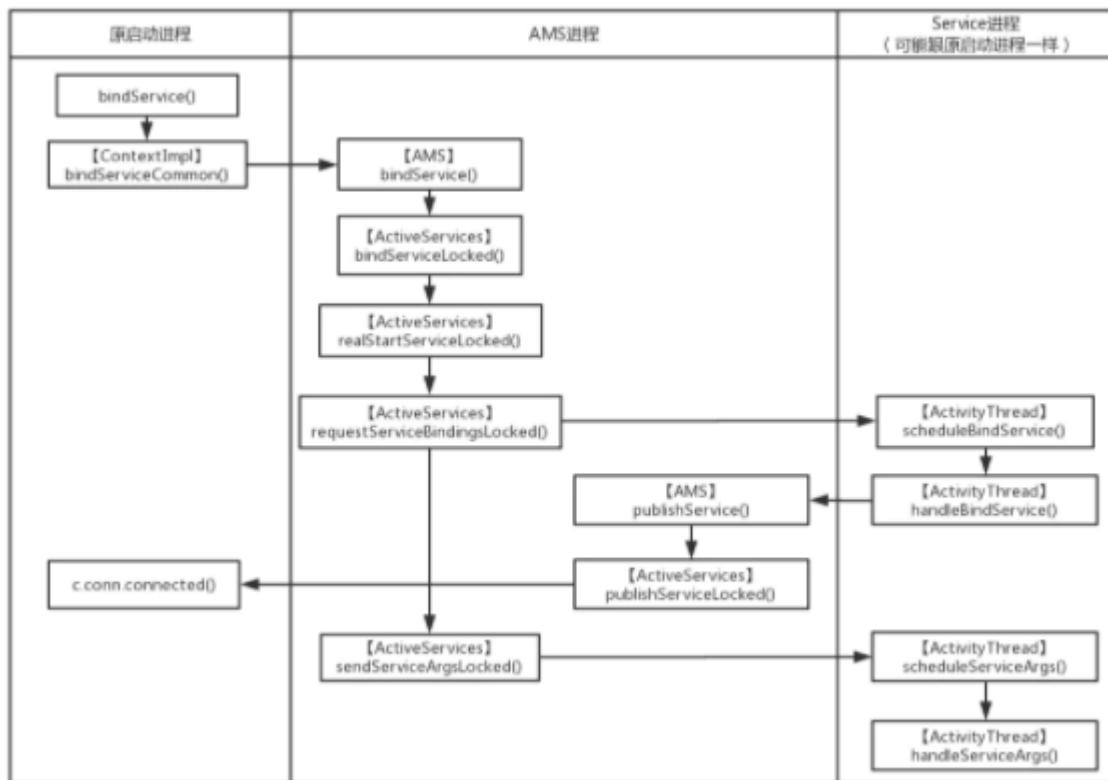
2.2.Service启动流程

<http://gityuan.com/2016/03/06/start-service/>

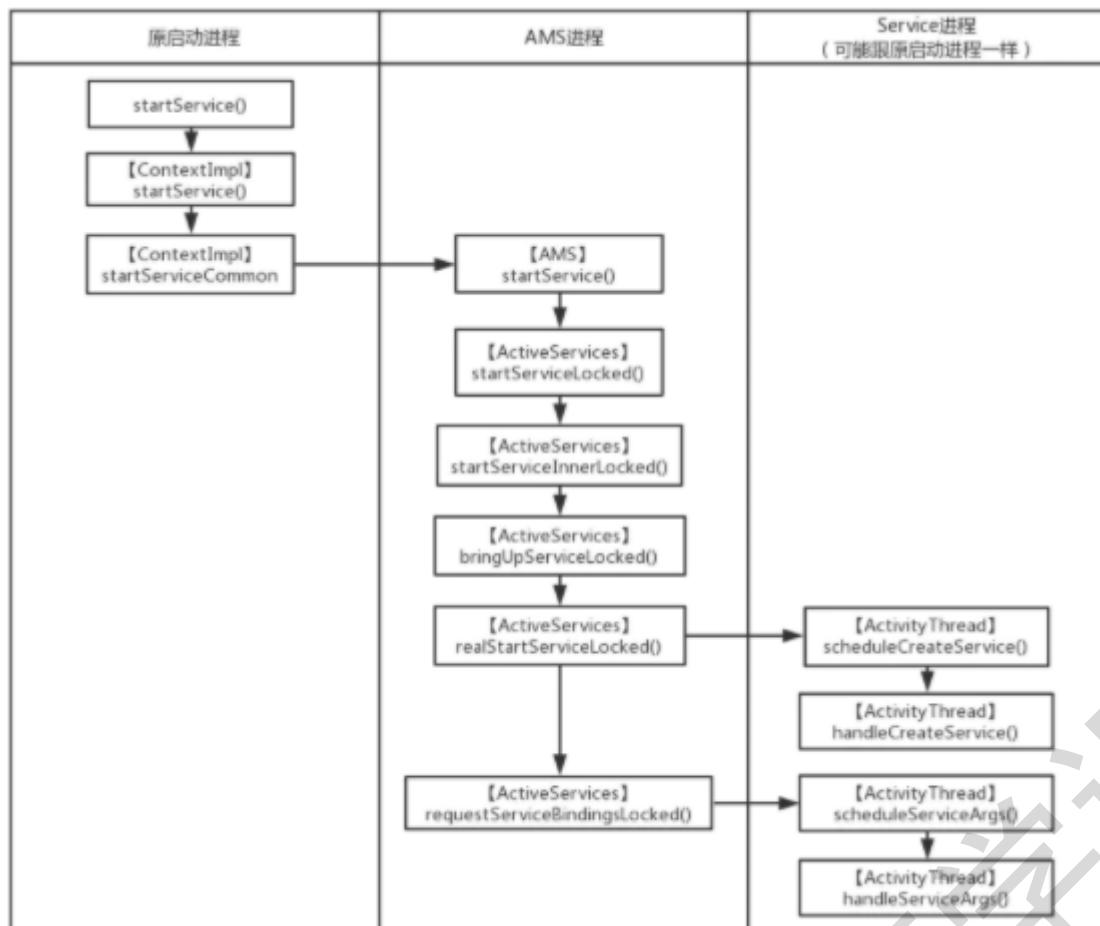


1. Process A进程采用Binder IPC向system_server进程发起startService请求;
 2. system_server进程接收到请求后，向zygote进程发送创建进程的请求；
 3. zygote进程fork出新的子进程Remote Service进程；
 4. Remote Service进程，通过Binder IPC向sytem_server进程发起attachApplication请求；
 5. system_server进程在收到请求后，进行一系列准备工作后，再通过binder IPC向remote Service进程发送scheduleCreateService请求；
 6. Remote Service进程的binder线程在收到请求后，通过handler向主线程发送CREATE_SERVICE消息；
 7. 主线程在收到Message后，通过发射机制创建目标Service，并回调Service.onCreate()方法。
- 到此，服务便正式启动完成。当创建的是本地服务或者服务所属进程已创建时，则无需经过上述步骤2、3，直接创建服务即可。

bindService



startService



2.3.Service与Activity怎么实现通信

通过Binder对象

- 1.Service中添加一个继承Binder的内部类，并添加相应的逻辑方法
- 2.Service中重写Service的onBind方法，返回我们刚刚定义的那个内部类实例
- 3.Activity中绑定服务，重写ServiceConnection，onServiceConnected时返回的IBinder (Service中的binder) 调用逻辑方法

Service通过BroadCast广播与Activity通信

2.4.IntentService是什么，IntentService原理，应用场景及其与Service的区别

what

IntentService 是 Service 的子类，默认开启了一个工作线程HandlerThread，使用这个工作线程逐一处理所有启动请求，在任务执行完毕后会自动停止服务。只要实现一个方法 onHandleIntent，该方法会接收每个启动请求的 Intent，能够执行后台工作和耗时操作。

可以启动 IntentService 多次，而每一个耗时操作会以队列的方式在 IntentService 的 onHandleIntent 回调方法中执行，并且，每一次只会执行一个工作线程，执行完第一个再执行第二个。并且等待所有消息都执行完后才终止服务。

how

1. 创建一个名叫 ServiceHandler 的内部 Handler
2. 把内部Handler与HandlerThread所对应的子线程进行绑定
3. HandlerThread开启线程 创建自己的looper
4. 通过 onStartCommand() intent，依次插入到工作队列中，并发送给 onHandleIntent()逐个处理

可以用作后台下载任务 静默上传

why IntentService会创建独立的worker线程来处理所有的Intent请求 Service主线程不能处理耗时操作,IntentService不会阻塞UI线程，而普通Service会导致ANR异常。

为Service的onBind()提供默认实现，返回null；onStartCommand提供默认实现，将请求Intent添加到队列中。

所有请求处理完成后，IntentService会自动停止，无需调用stopSelf()方法停止Service。

2.5.Service 的 onStartCommand 方法有几种返回值?各代表什么意思?

START_NOT_STICKY

在执行完 onStartCommand 后,服务被异常 kill 掉,系统不会自动重启该服务。

START_STICKY

重传 Intent。使用这个返回值时,如果在执行完 onStartCommand 后,服务被异常 kill 掉,系统会自动重启该服务，并且onStartCommand方法会执行, onStartCommand方法中的intent值为null。

适用于媒体播放器或类似服务。

START_REDELIVER_INTEN

使用这个返回值时,服务被异常 kill 掉,系统会自动重启该服务,并将 Intent 的值传入。

适用于主动执行应该立即恢复的作业(例如下载文件)的服务。

2.6.bindService和startService混合使用的生命周期以及怎么关闭

如果你只是想要启动一个后台服务长期进行某项任务,那么使用startService便可以了。如果你还想要与正在运行的Service取得联系,那么有两种方法:一种是使用broadcast,另一种是使用bindService。

<https://blog.csdn.net/u014520745/article/details/49669641>

如果先startService,再bindService

onCreate() -> onBind() -> onStartCommand()

如果先bindService,再startService

onCreate() -> onStartCommand() -> onBind()

如果只stopService

Service的OnDestroy()方法不会立即执行,在Activity退出的时候,会执行OnDestroy。

如果只unbindService

只有onUnbind方法会执行, onDestory不会执行

如果要完全退出Service,那么就得执行unbindService()以及stopService。

3.BroadcastReceiver

3.1.广播的分类和使用场景

Android 广播分为两个角色:广播发送者、广播接受者

广播接收器的注册分为两种:静态注册、动态注册。

静态广播接收者:通过AndroidManifest.xml的标签来申明的BroadcastReceiver。

动态广播接收者:通过AMS.registerReceiver()方式注册的BroadcastReceiver,动态注册更为灵活,可在不需要时通过unregisterReceiver()取消注册。

广播类型:根据广播的发送方式,

1.普通广播:通过Context.sendBroadcast()发送,可并行处理

2.系统广播:当使用系统广播时,只需在注册广播接收者时定义相关的action即可,不需要手动发送广播(网络变化,锁屏,飞行模式)

3.有序广播:指的是发送出去的广播被 BroadcastReceiver 按照先后顺序进行接收 发送方式变为:
sendOrderedBroadcast(intent);

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。

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，获取广播具体信息；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）

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。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。

具体使用2 - 使用封装好的LocalBroadcastManager类

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

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。

应用场景

同一 App 内部的不同组件之间的消息通信（单个进程）；

不同 App 之间的组件之间消息通信；

Android系统在特定情况下与App之间的消息通信，如：网络变化、电池电量、屏幕开关等。

3.2.广播的两种注册方式的区别

静态注册：常驻系统，不受组件生命周期影响，即便应用退出，广播还是可以被接收，耗电、占内存。

动态注册：非常驻，跟随组件的生命变化，组件结束，广播结束。在组件结束前，需要先移除广播，否则容易造成内存泄漏。

3.3.广播发送和接收的原理

<https://juejin.im/post/6844904057891471367#heading-0>

<http://gityuan.com/2016/06/04/broadcast-receiver/>

动态注册

1. 创建对象LoadedApk.ReceiverDispatcher.InnerReceiver的实例，该对象继承于IIntentReceiver.Stub（InnerReceiver实际是一个binder本地对象（BBinder：本地Binder，服务实现方的基类，提供了onTransact接口来接收请求））。

2. 将IIntentReceiver对象和注册所传的IntentFilter对象发送给AMS。AMS记录IIntentReceiver、IntentFilter和注册的进程ProcessRecord，并建立起它们的对应关系。

3. 当有广播发出时，AMS根据广播intent所携带的IntentFilter找到IIntentReceiver和ProcessRecord，然后回调App的ApplicationThread对象的scheduleRegisteredReceiver，将IIntentReceiver和广播的intent一并传给App，App直接调用IIntentReceiver的performReceive。

4. 因为广播是通过binder线程回调到接收进程的，接收进程通过ActivityThread里的H这个Handler将调用转到主线程，然后回调BroadcastReceiver的onReceive。

静态注册

静态注册是通过在Manifest文件中声明实现了BroadcastReceiver的自定义类和对应的IntentFilter，来告诉PMS(PackageManagerService)这个App所注册的广播。

当AMS接收到广播后，会查找所有动态注册的和静态注册的广播接收器，静态注册的广播接收器是通过PMS(PackageManagerService)发现的，PMS找到对应的App

对应进程已经创建，直接调用App的ApplicationThread对象的scheduleReceiver

对应进程尚未创建，先启动App进程，App进程启动后回调AMS的attachApplication，attachApplication则继续派发刚才的广播App这边收到调用后会先通过Handler转到主线程，然后根据AMS传过来的参数实例化广播接收器的类，接着调用广播接收器的onReceive。

3.4.本地广播和全局广播的区别

BroadcastReceiver是针对应用间、应用与系统间、应用内部进行通信的一种方式

LocalBroadcastReceiver仅在自己的应用内发送接收广播，也就是只有自己的应用能收到，数据更加安全广播只在这个程序里，而且效率更高。

BroadcastReceiver采用的binder方式实现跨进程间的通信；

LocalBroadcastManager使用Handler通信机制。

LocalBroadcastReceiver 使用

LocalBroadcastReceiver不能静态注册，只能采用动态注册的方式。

在发送和注册的时候采用，LocalBroadcastManager的sendBroadcast方法和registerReceiver方法

[http://gityuan.com/2017/04/23/local broadcast manager/](http://gityuan.com/2017/04/23/local_broadcast_manager/)

注册过程，主要是向mReceivers和mActions添加相应数据：

mReceivers：数据类型为HashMap<BroadcastReceiver, ArrayList>，记录广播接收者与IntentFilter列表的对应关系；

mActions：数据类型为HashMap<String, ArrayList>，记录action与广播接收者的对应关系

根据Intent的action来查询相应的广播接收者列表；

发送MSG_EXEC_PENDING_BROADCASTS消息，回调相应广播接收者的onReceive方法

4.ContentProvider

4.1.什么是ContentProvider及其使用

ContentProvider的作用是为不同的应用之间数据共享，提供统一的接口，我们知道安卓系统中应用内部的数据是对外隔离的，要想让其它应用能使用自己的数据（例如通讯录）这个时候就用到了ContentProvider。

ContentProvider（内容提供者）通过uri来标识其它应用要访问的数据。

通过ContentResolver（内容解析者）的增、删、改、查方法实现对共享数据的操作。

还可以通过注册 ContentObserver (内容观察者) 来监听数据是否发生了变化来对应的刷新页面

4.2.ContentProvider,ContentResolver,ContentObserver之间的关系

ContentProvider：管理数据，提供数据的增删改查操作，数据源可以是数据库、文件、XML、网络等。

ContentResolver：外部进程可以通过 ContentResolver 与 ContentProvider 进行交互。其他应用中 ContentResolver 可以不同 URI 操作不同的 ContentProvider 中的数据。

ContentObserver：观察 ContentProvider 中的数据变化，并将变化通知给外界。

4.3.ContentProvider的实现原理

<https://juejin.im/post/6844904062173839368#heading-0>

<http://gityuan.com/2016/07/30/content-provider/>

<https://blog.csdn.net/u011733869/article/details/83958712>

ContentProvider的安装(ActivityThread.installProvider)

当主线程收到H.BIND_APPLICATION消息后，会调用handleBindApplication方法。

handleBindApplication->installProvider

installProvider()

创建了provider对象

创建ProviderClientRecord，这是一个provider在client进程中对应的对象

放入mProviderMap(记录所有contentProvider)

总结：把provider启动起来并记录和发布给AMS

ContentResolver.query

调用端App在使用ContentProvider前首先要获取ContentProvider

1. 通过ContentResolver调用acquireProvider

2.ActivityThread首先通过一个map查找是否已经install过这个Provider，如果install过就直接将之返回给调用者，如果没有install过就调用AMS的getContentProvider,AMS首先查找这个Provider是否被publish过，如果publish过就直接返回，否则通过PMS找到Provider所在的App。

3.如果发现目标App进程未启动,就创建一个ContentProviderRecord对象然后调用其wait方法阻塞当前执行流程,启动目标App进程,AMS找到App的所有运行于当前进程的Provider,保存在map中,将要启动的所有Provider传给目标App进程,解除前面对获取Provider执行流程的阻塞.

4.如果目标App进程已启动，AMS在getContentProvider里会查找到要获取的Provider，就直接返回了.调用端App收到AMS的返回结果后(acquireProvider返回)，调用ActivityThread的installProvider将Provider记录到本地的一个map中，下次再调用acquireProvider就直接返回。

ContentProvider所提供的接口中只有query是基于共享内存的，其他都是直接使用binder的入参出参进行数据传递。

AMS作为一个中间管理员的身份，所有的provider会向它注册

向AMS请求到provider之后，就可以在client和server之间自行binder通信，不需要再经过systemserver

4.4.ContentProvider的优点

封装

采用ContentProvider方式，其解耦了底层数据的存储方式，使得无论底层数据存储采用何种方式，外界对数据的访问方式都是统一的，这使得访问简单 & 高效

如一开始数据存储方式采用 SQLite 数据库，后来把数据库换成 MongoDB，也不会对上层数据ContentProvider使用代码产生影响

提供一种跨进程数据共享的方式。

应用程序间的数据共享还有另外的一个重要话题，就是数据更新通知机制了。因为数据是在多个应用程序中共享的，当其中一个应用程序改变了这些共享数据的时候，它有责任通知其它应用程序，让它们知道共享数据被修改了，这样它们就可以作相应的处理。

4.5.Uri 是什么

定义：Uniform Resource Identifier，即统一资源标识符

作用：唯一标识 ContentProvider & 其中的数据，URI分为 系统预置 & 自定义，分别对应系统内置的数据（如通讯录、日程表等等）和自定义数据库

每一个 ContentProvider 都拥有一个公共的 URI，这个 URI 用于表示这个 ContentProvider 所提供的数据。

自定义URI = content:// com.carson.provider / User / 1

主题名 授权信息 表名 记录

- 主题 (Schema) : Content Provider的URI前缀 (Android 规定)
- 授权信息 (Authority) : Content Provider的唯一标识符
- 表名 (Path) : Content Provider 指向数据库中的某个表名
- 记录 (ID) : 表中的某个记录 (若无指定，则返回全部记录)

将其分为 A, B, C, D 4个部分：

A: 标准前缀，； "content://"；

B: URI 的标识，用于唯一标识这个 ContentProvider，外部调用者可以根据这个标识来找到它。

C: 路径 (path)，通俗的讲就是你要操作的数据库中表的名字，

D: 如果URI中包含表示需要获取的记录的 ID；则就返回该id对应的数据，如果没有 ID，就表示返回全部。

5.Handler

5.1.Handler的实现原理

从四个方面看Handler、Message、MessageQueue 和 Looper

Handler:负责消息的发送和处理

Message:消息对象，类似于链表的一个结点;

MessageQueue:消息队列，用于存放消息对象的数据结构;

Looper:消息队列的处理器 (用于轮询消息队列的消息对象)

Handler发送消息时调用MessageQueue的enqueueMessage插入一条信息到MessageQueue,Looper不断轮询调用MessageQueue的next方法 如果发现message就调用handler的dispatchMessage, dispatchMessage被成功调用, 接着调用handlerMessage()。

5.2.子线程中能不能直接new一个Handler,为什么主线程可以

主线程的Looper第一次调用loop方法,什么时候,哪个类

不能, 因为Handler 的构造方法中, 会通过Looper.myLooper()获取looper对象, 如果为空, 则抛出异常, 主线程则因为已在入口处ActivityThread的main方法中通过 Looper.prepareMainLooper()获取到这个对象, 并通过 Looper.loop()开启循环, 在子线程中若要使用handler, 可先通过Loop.prepare获取到looper对象, 并使用Looper.loop()开启循环

5.3.Handler导致的内存泄露原因及其解决方案

原因:1.Java中非静态内部类和匿名内部类都会隐式持有当前类的外部引用

2.我们在Activity中使用非静态内部类初始化了一个Handler,此Handler就会持有当前Activity的引用。

3.我们想要一个对象被回收, 那么前提它不被任何其它对象持有引用, 所以当我们Activity页面关闭之后, 存在引用关系: "未被处理 / 正处理的消息 -> Handler实例 -> 外部类", 如果在Handler消息队列 还有未处理的消息 / 正在处理消息时 导致Activity不会被回收, 从而造成内存泄漏 解决方案: 1.将Handler的子类设置成 静态内部类, 使用WeakReference弱引用持有Activity实例 2.当外部类结束生命周期时, 清空Handler内消息队列

5.4.一个线程可以有几个Handler,几个Looper,几个MessageQueue对象

一个线程可以有多个Handler,只有一个Looper对象,只有一个MessageQueue对象。Looper.prepare()函数中知道, 在Looper的prepare方法中创建了Looper对象, 并放入到ThreadLocal中, 并通过ThreadLocal来获取looper的对象, ThreadLocal的内部维护了一个ThreadLocalMap类, ThreadLocalMap是以当前thread做为key的, 因此可以得知, 一个线程最多只能有一个Looper对象, 在Looper的构造方法中创建了MessageQueue对象, 并赋值给mQueue字段。因为Looper对象只有一个, 那么Messagequeue对象肯定只有一个。

5.5.Message对象创建的方式有哪些 & 区别

Message.obtain()怎么维护消息池的

1.Message msg = new Message();

每次需要Message对象的时候都创建一个新的对象, 每次都要去堆内存开辟对象存储空间 2.Message msg = Message.obtain();

obtainMessage能避免重复Message创建对象。它先判断消息池是不是为空, 如果非空的话就从消息池表头的Message取走, 再把表头指向 next。

如果消息池为空的话说明还没有Message被放进去, 那么就new出来一个Message对象。消息池使用 Message 链表结构实现, 消息池默认最大值 50。消息在loop中被handler分发消费之后会执行回收的操作, 将该消息内部数据清空并添加到消息链表的表头。

3.Message msg = handler.obtainMessage(); 其内部也是调用的obtain()方法

5.6.Handler 有哪些发送消息的方法

sendMessage(Message msg)

sendMessageDelayed(Message msg, long uptimeMillis)

post(Runnable r)

postDelayed(Runnable r, long uptimeMillis)

sendMessageAtTime(Message msg, long when)

5.7.Handler的post与sendMessage的区别和应用场景

1.源码

sendMessage
sendMessage->sendMessageAtTime->enqueueMessage。

post

sendMessage->getPostMessage->sendMessageAtTime->enqueueMessage
getPostMessage会先生成一个Message，并且把Runnable赋值给message的callback

2Looper->dispatchMessage处理时

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

dispatchMessage方法中直接执行post中的Runnable方法。

而sendMessage中如果mCallback不为null就会调用mCallback.handleMessage(msg)方法，如果handler内的callback不为空，执行mCallback.handleMessage(msg)这个处理消息并判断返回是否为true，如果返回true，消息处理结束，如果返回false，handleMessage(msg)处理。否则会直接调用handleMessage方法。

post方法和handleMessage方法的不同在于，区别就是调用post方法的消息是在post传递的Runnable对象的run方法中处理，而调用sendMessage方法需要重写handleMessage方法或者给handler设置callback，在callback的handleMessage中处理并返回true

应用场景

post一般用于单个场景 比如单一的倒计时弹框功能 sendMessage的回调需要去实现handleMessage Message则做为参数 用于多判断条件的场景。

5.8.handler postDelay后消息队列有什么变化，假设先 postDelay 10s, 再 postDelay 1s, 怎么处理这2条消息sendMessageDelayed-sendMessageAtTime-sendMessage

postDelayed传入的时间，会和当前的时间SystemClock.uptimeMillis()做加和，而不是单纯的只是用延时时间。延时消息会和当前消息队列里的消息头的执行时间做对比，如果比头的时间靠前，则会成为新的消息头，不然则会从消息头开始向后遍历，找到合适的位置插入延时消息。

postDelay()一个10秒钟的Runnable A、消息进队，MessageQueue调用nativePollOnce()阻塞，Looper阻塞；紧接着post()一个Runnable B、消息进队，判断现在A时间还没到、正在阻塞，把B插入消息队列的头部（A的前面），然后调用nativeWake()方法唤醒线程；

MessageQueue.next()方法被唤醒后，重新开始读取消息链表，第一个消息B无延时，直接返回给Looper；

Looper处理完这个消息再次调用next()方法，MessageQueue继续读取消息链表，第二个消息A还没到时间，计算一下剩余时间（假如还剩9秒）继续调用nativePollOnce()阻塞；直到阻塞时间到或者下一次有Message进队；

5.9.MessageQueue是什么数据结构

内部存储结构并不是真正的队列，而是采用单链表的数据结构来存储消息列表

这点和传统的队列有点不一样，主要区别在于Android的这个队列中的消息是按照时间先后顺序来存储的，时间较早的消息，越靠近队头。当然，我们也可以理解成，它是先进先出的，只是这里的先依据的不是谁先入队，而是消息待发送的时间

5.10.Handler怎么做到的一个线程对应一个Looper，如何保证只有一个MessageQueue ThreadLocal在Handler机制中的作用

设计的初衷是为了解决多线程编程中的资源共享问题，

synchronized采取的是“以时间换空间”的策略，本质上是对关键资源上锁，让大家排队操作。

而ThreadLocal采取的是“以空间换时间”的思路，它一个线程内部的数据存储类，通过它可以在制定的线程中存储数据，数据存储以后，只有在指定线程中可以获取到存储的数据，对于其他线程就获取不到数据，可以保证本线程任何时间操纵的都是同一个对象。比如对于Handler，它要获取当前线程的Looper，很显然Looper的作用域就是线程，并且不同线程具有不同的Looper。ThreadLocal本质是操作线程中ThreadLocalMap来实现本地线程变量的存储的ThreadLocalMap是采用数组的方式来存储数据，其中key(弱引用)指向当前ThreadLocal对象，value为设的值通过ThreadLocal计算出Hash key，通过这个哈 ThreadLocal对象，value为设的值

5.11.HandlerThread是什么 & 好处 & 原理 & 使用场景

HHandlerThread本质上是一个线程类，它继承了Thread；HandlerThread有自己的内部Looper对象，通过Looper.loop()进行looper循环；

通过获取HandlerThread的looper对象传递给Handler对象，然后在handleMessage()方法中执行异步任务；

优势：

- 1.将loop运行在子线程中处理，减轻了主线程的压力，使主线程更流畅，有自己的消息队列，不会干扰UI线程
- 2.串行执行，开启一个线程起到多个线程的作用

劣势：

- 1.由于每一个任务队列逐步执行，一旦队列耗时过长，消息延时
- 2.对于IO等操作，线程等待，不能并发

我们可以使用HandlerThread处理本地IO读写操作（数据库，文件），因为本地IO操作大多数的耗时属于毫秒级别，对于单线程 + 异步队列的形式 不会产生较大的阻塞

5.12.IdleHandler及其使用场景

Handler 机制提供的一种，可以在 Looper 事件循环的过程中，当出现空闲的时候，允许我们执行任务的一种机制。IdleHandler在looper里面的message处理完了的时候去调用

怎么使用

IdleHandler 被定义在 MessageQueue 中，它是一个接口。定义时需要实现其 queueIdle() 方法。返回值为 true 表示是一个持久的 IdleHandler 会重复使用，返回 false 表示是一个一次性的 IdleHandler。

IdleHandler 被 MessageQueue 管理，对应的提供了 addIdleHandler() 和 removeIdleHandler() 方法。将其存入 mIdleHandle addIdleHandler() 和 removeIdleHandler() 方法。将其存入 mIdleHandlers 这个 ArrayList 中。

什么时候掉用

就在MessageQueue的next方法里面。MessageQueue 为空，没有 Message；MessageQueue 中最近待处理的 Message，是一个延迟消息（when>currentTime），需要滞后执行；

使用场景

1. Activity启动优化：onCreate, onStart, onResume中耗时较短但非必要的代码可以放到IdleHandler中执行，减少启动时间

2. 想要在一个View绘制完成之后添加其他依赖于这个View的View，当然这个用View#post()也能实现，区别就是前者会在消息队列空闲时执行

优化页面的启动，较复杂的view填充 填充里面的数据界面view绘制之前的话，就会出现以上的效果了，view先是白的，再出现。app的进程其实是ActivityThread,performResumeActivity先回调onResume，之后执行view绘制的measure, layout, draw,也就是说onResume的方法是在绘制之前，在onResume中做一些耗时操作都会影响启动时间 把在onResume以及其之前的调用的但非必须的事件（如某些界面View的绘制）挪出来找一个时机（即绘制完成以后）去调用即可。

5.13.消息屏障，同步屏障机制what

同步屏障只在Looper死循环获取待处理消息时才会起作用，也就是说同步屏障在MessageQueue.next函数中发挥着作用。

在next()方法中，有一个屏障的概念(message.target == null为屏障消息)，遇到target为null的Message，说明是同步屏障，循环遍历找出一条异步消息，然后处理。在同步屏障没移除前，只会处理异步消息，处理完所有的异步消息后，就会处于堵塞 当出现屏障的时候，会滤过同步消息，而是直接获取其中的异步消息并返回，就是这样来实现「异步消息优先执行」的功能

how

- 1、Handler构造方法中传入async参数，设置为true，使用此Handler添加的Message都是异步的；
- 2、创建Message对象时，直接调用setAsynchronous(true) 3.removeSyncBarrier() 移除同步屏障：

应用

在 View 更新时，draw、requestLayout、invalidate 等很多地方都调用了ViewRootImpl#scheduleTraversals(Android应用框架中为了更快的响应UI刷新事件在ViewRootImpl.scheduleTraversals中使用了同步屏障

5.14.子线程能不能更新UI

刷新UI，都会调用到ViewRootImpl.Android每次刷新UI的时候，最终根布局ViewRootImpl.checkThread()来检验线程是否是View的创建线程。ViewRootImpl创建的第一个地方，从Activity声明周期handleResumeActivity会被优先调用到，也就是说在OnResume后ViewRootImpl就被创建，这个时候无法在在子线程中访问UI了，上面子线程延迟了一会，handleResumeActivity已经被调用了，所以发生了崩溃 不延迟在create里直接设置不会崩溃 线程更新UI也行，但是只能更新自己创建的View

5.15.为什么Android系统不建议子线程访问UI

在android中子线程可以有好多个，但是如果每个线程都可以对ui进行访问，我们的界面可能就会变得混乱不堪，这样多个线程操作同一资源就会造成线程安全问题，当然，需要解决线程安全问题的时候，我们第一想到的可能就是加锁，但是加锁会降低运行效率，所以android出于性能的考虑，并没有使用加锁来进行ui操作的控制。

5.16.Android中为什么主线程不会因为Looper.loop()里的死循环卡死？

MessageQueue#next 在没有消息的时候会阻塞，如何恢复？

他不阻塞的原因是epoll机制，他是linux里面的，在native层会有一个读取端和一个写入端，当有消息发送过来的时候会去唤醒读取端，然后进行消息发送与处理，没消息的时候是处于休眠状态，所以他不会阻塞他。

5.17.Handler消息机制中，一个looper是如何区分多个Handler的当Activity有多个Handler的时候，怎么样区分当前消息由哪个Handler处理处理message的时候怎么知道是去哪个callback处理的

每个Handler会被添加到 Message 的target字段上面，Looper 通过调用 Message.target.handleMessage() 来让 Handler 处理消息。

5.18.Looper.quit/quitSafely的区别

当我们调用Looper的quit方法时，实际上执行了MessageQueue中的removeAllMessagesLocked方法，该方法的作用是把MessageQueue消息池中所有的消息全部清空，无论是延迟消息（延迟消息是指通过sendMessageDelayed或通过postDelayed等方法发送的需要延迟执行的消息）还是非延迟消息。

当我们调用Looper的quitSafely方法时，实际上执行了MessageQueue中的removeAllFutureMessagesLocked方法，通过名字就可以看出，该方法只会清空MessageQueue消息池中所有的延迟消息，并将消息池中所有的非延迟消息派发出去让Handler去处理，quitSafely相比于quit方法安全之处在于清空消息之前会派发所有的非延迟消息。

5.19.通过Handler如何实现线程的切换

当在A线程中创建handler的时候，同时创建了MessageQueue与Looper，Looper在A线程中调用loop进入一个无限的for循环从MessageQueue中取消息，当B线程调用handler发送一个message的时候，会通过msg.target.dispatchMessage(msg);将message插入到handler对应的MessageQueue中，Looper发现有message插入到MessageQueue中，便取出message执行相应的逻辑，因为Looper.loop()是在A线程中启动的，所以则回到了A线程，达到了从B线程切换到A线程的目的。

5.20.Handler 如何与 Looper 关联的

通过构造方法 mLooper = Looper.myLooper()->sThreadLocal.get()(sThreadLocal.set)

5.21.Looper 如何与 Thread 关联的

Looper 与 Thread 之间是通过 ThreadLocal 关联的，这个可以看 Looper#prepare() 方法 Looper 中有一个 ThreadLocal 类型的 sThreadLocal 静态字段，Looper 通过它的 get 和 set 方法来赋值和取值。由于 ThreadLocal 是与线程绑定的，所以我们只要把 Looper 与 ThreadLocal 绑定了，那 Looper 和 Thread 也就关联上了

5.22.Looper.loop()源码

for无限循环，阻塞于消息队列的next方法 取出消息后调用msg.target.dispatchMessage(msg)进行消息分发

5.23.MessageQueue的enqueueMessage()方法如何进行线程同步的

就是单链表的插入操作 如果消息队列被阻塞回调用nativeWake去唤醒。用synchronized代码块去进行同步。

5.24.MessageQueue的next()方法内部原理

next() 是如何处理一般消息的？

next() 是如何处理同步屏障的？

next() 是如何处理延迟消息的？

调用 MessageQueue.next() 方法的时候会调用 Native 层的 nativePollOnce() 方法进行精准时间的阻塞。在 Native 层，将进入 pullInner() 方法，使用 epoll_wait 阻塞等待以读取管道的通知。如果没有从 Native 层得到消息，那么这个方法就不会返回。此时主线程会释放 CPU 资源进入休眠状态。

5.25.子线程中是否可以用MainLooper去创建Handler，Looper和Handler是否一定处于一个线程

可以的。子线程中 Handler handler = new Handler(Looper.getMainLooper());，此时两者就不在一个线程中

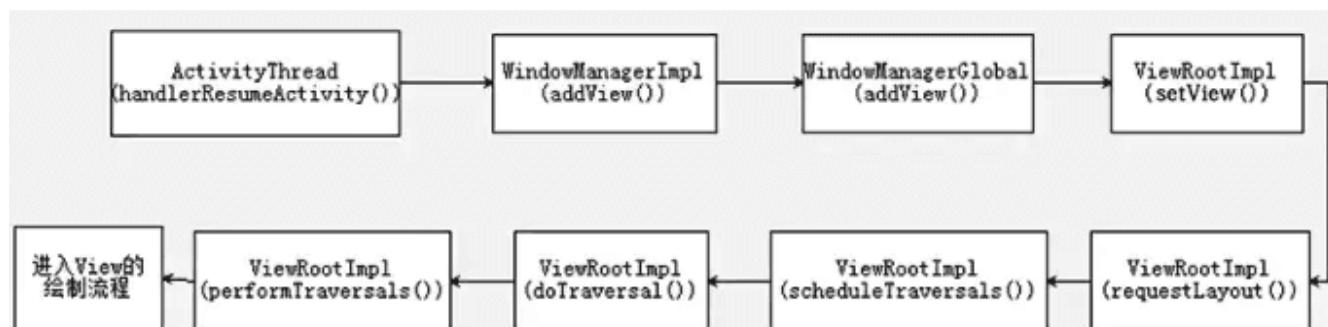
5.26.ANR和Handler的联系

Handler是线程间通讯的机制，Android中，网络访问、文件处理等耗时操作必须放到子线程中去执行，否则将会造成ANR异常。ANR异常：Application Not Response 应用程序无响应产生ANR异常的原因：在主线程执行了耗时操作，对Activity来说，主线程阻塞5秒将造成ANR异常，对BroadcastReceiver来说，主线程阻塞10秒将会造成ANR异常。解决ANR异常的方法：耗时操作都在子线程中去执行但是，Android不允许在子线程去修改UI，可我们又有在子线程去修改UI的需求，因此需要借助Handler。

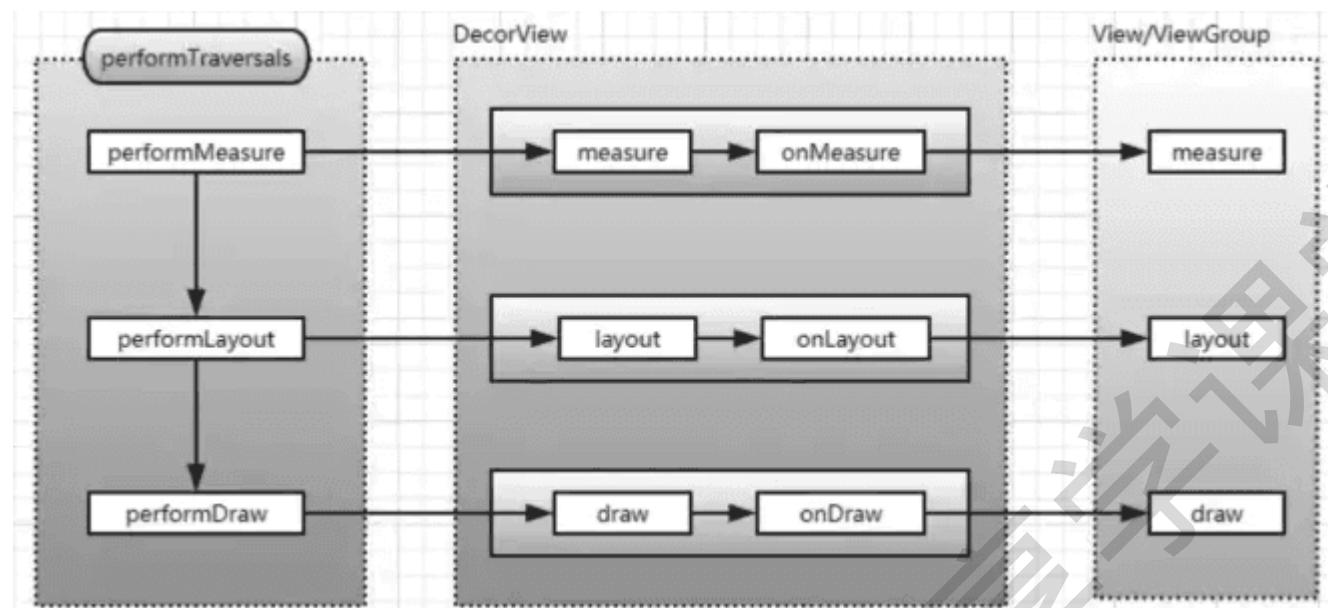
6.View绘制

6.1.View绘制流程

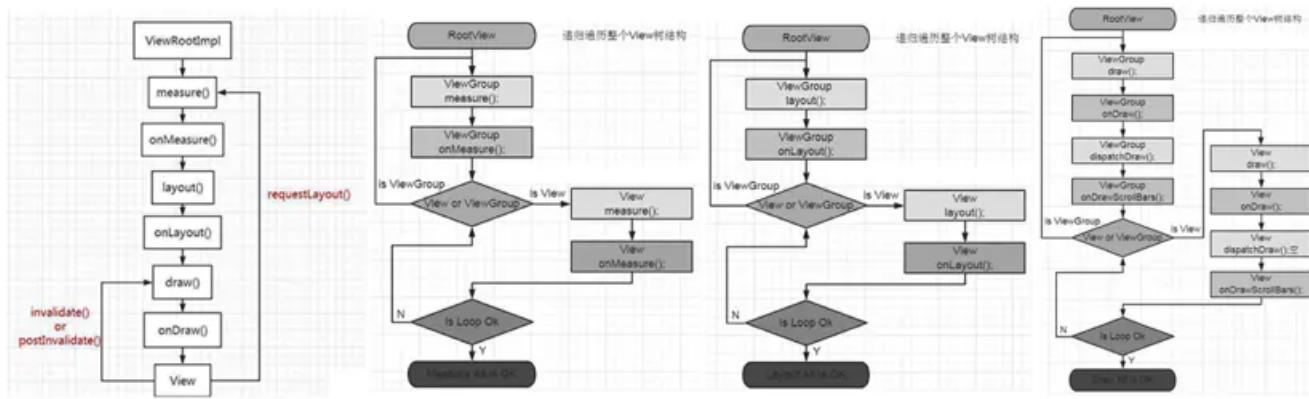
触发addView流程：



performTraversals流程：



measure、layout、draw流程：



`startActivity->ActivityThread.handleLaunchActivity->onCreate` ->完成DecorView和Activity的创建-
`>handleResumeActivity->onResume()->DecorView添加到WindowManager->ViewRootImpl.performTraversals()`方法，测量 (measure) ,布局 (layout) ,绘制 (draw) ,从DecorView自上而下遍历整个View树。

Measure：测量视图宽高。

单一View:measure() -> onMeasure() -> getDefaultSize() 计算View的宽/高值 -> setMeasuredDimension存储测量后的View宽 / 高

ViewGroup:

- > measure()
- > 需要重写onMeasure(ViewGroup)没有定义测量的具体过程，因为ViewGroup是一个抽象类，其测量过程的onMeasure方法需要各个子类去实现。如：LinearLayout、RelativeLayout、FrameLayout等等，这些控件的特性都是不一样的，测量规则自然也都不一样。)遍历测量ViewGroup中所有的View
- > 根据父容器的MeasureSpec和子View的LayoutParams等信息计算子View的MeasureSpec
- > 合并所有子View计算出ViewGroup的尺寸
- > setMeasuredDimension 存储测量后的宽 / 高 从顶层父View向子View的递归调用view.layout方法的过程，即父View根据上一步measure子View所得到的布局大小和布局参数，将子View放在合适的位置上。

Layout: 先通过 measure 测量出 ViewGroup 宽高， ViewGroup 再通过 layout 方法根据自身宽高来确定自身位置。当 ViewGroup 的位置被确定后，就开始在 onLayout 方法中调用子元素的 layout 方法确定子元素的位置。子元素如果是 ViewGroup 的子类，又开始执行 onLayout，如此循环往复，直到所有子元素的位置都被确定，整个 View 树的 layout 过程就执行完了。

Draw: 绘制视图。ViewRoot创建一个Canvas对象，然后调用OnDraw()。六个步骤：①、绘制视图的背景；②、保存画布的图层 (Layer) ；③、绘制View的内容；④、绘制View子视图，如果没有就不用；⑤、还原图层 (Layer) ；⑥、绘制View的装饰(例如滚动条等等)。

6.2.MeasureSpec是什么

MeasureSpec表示的是一个32位的整形值，它的高2位表示测量模式SpecMode，低30位表示某种测量模式下的规格大小SpecSize。MeasureSpec是View类的一个静态内部类，用来说明应该如何测量这个View。它由三种测量模式，如下：

EXACTLY: 精确测量模式，视图宽高指定为match_parent或具体数值时生效，表示父视图已经决定了子视图的精确大小，这种模式下View的测量值就是SpecSize的值。

AT_MOST: 最大值测量模式，当视图的宽高指定为wrap_content时生效，此时子视图的尺寸可以是不超过父视图允许的最大尺寸的任何尺寸。

UNSPECIFIED：不指定测量模式，父视图没有限制子视图的大小，子视图可以是想要的任何尺寸，通常用于系统内部，应用开发中很少用到。

MeasureSpec通过将SpecMode和SpecSize打包成一个int值来避免过多的对象内存分配，为了方便操作，其提供了打包和解包的方法，打包方法为makeMeasureSpec，解包方法为getMode和getSize。

6.3.子View创建MeasureSpec创建规则是什么

根据父容器的MeasureSpec和子View的LayoutParams等信息计算子View的MeasureSpec

规律前提	子View的MeasureSpec值
当子View采用具体数值 (dp / px) 时	<ul style="list-style-type: none">测量模式 = EXACTLY测量大小 = 其自身设置的具体数值
当子View采用match_parent时	<ul style="list-style-type: none">测量模式 = 父容器的测量模式测量大小：<ul style="list-style-type: none">a. 若父容器的测量模式为EXACTLY，那么测量大小 = 父容器的剩余空间b. 若父容器的测量模式为AT_MOST，那么测量大小 = 不超过父容器的剩余空间
当子View采用wrap_content时	<ul style="list-style-type: none">测量模式 = AT_MOST测量大小 = 不超过父容器的剩余空间

6.4.自定义Viewwrap_content不起作用的原因

1.因为onMeasure()->getDefaultSize()，当View的测量模式是AT_MOST或EXACTLY时，View的大小都会被设置成子View MeasureSpec的specSize。

```
public static int getDefaultSize(int size, int measureSpec) {  
    switch (specMode) {  
        case MeasureSpec.UNSPECIFIED:  
            result = size;  
            break;  
        case MeasureSpec.AT_MOST:  
        case MeasureSpec.EXACTLY:  
            result = specSize;  
            break;  
    }  
    return result;  
}
```

2.View的MeasureSpec值是根据子View的布局参数 (LayoutParams) 和父容器的MeasureSpec值计算得来，具体计算逻辑封装在getChildMeasureSpec()。当子View wrap_content或match_parent情况下，子View MeasureSpec的specSize被设置成parentSize = 父容器当前剩余空间大小

break;			
父视图测量模式 子视图布局参数 (mode) (LayoutParams)	EXACTLY	AT_MOST	UNSPECIFIED
具体数值 (dp / px)	EXACTLY + childSize	EXACTLY + childSize	EXACTLY + childSize
match_parent	EXACTLY + parentSize (父容器的剩余空间)	AT_MOST + parentSize (大小不超过父容器的剩余空间)	UNSPECIFIED + 0
wrap_content	AT_MOST + parentSize (大小不超过父容器的剩余空间)	AT_MOST + parentSize (大小不超过父容器的剩余空间)	UNSPECIFIED + 0

3. 所以当给一个View/ViewGroup设置宽高为具体数值或者match_parent，它都能正确的显示，但是如果你设置的是wrap_content->AT_MOST，则默认显示出来是其父容器的大小。如果你想要它正常的显示为wrap_content，所以需要自己重写onMeasure()来自己计算它的宽高度并设置。此时，可以在wrap_content的情况下（对应MeasureSpec.AT_MOST）指定内部宽/高(mWidth和mHeight)。

6.5. 在Activity中获取某个View的宽高有几种方法

- Activity/View#onWindowFocusChanged：此时View已经初始化完毕，当Activity的窗口得到焦点和失去焦点时均会被调用一次，如果频繁地进行onResume和onPause，那么onWindowFocusChanged也会被频繁地调用。
- view.post(runnable)：通过post将runnable放入ViewRootImpl的RunQueue中，RunQueue中runnable最后的执行时机，是在下一个performTraversals到来的时候，也就是view完成layout之后的第一时间获取宽高。
- ViewTreeObserver#addOnGlobalLayoutListener：当View树的状态发生改变或者View树内部的View的可见性发生改变时，onGlobalLayout方法将被回调。
- View.measure(int widthMeasureSpec, int heightMeasureSpec)：match_parent 直接放弃，无法measure出具体的宽/高。原因很简单，根据view的measure过程，构造此种MeasureSpec需要知道parentSize，即父容器的剩余空间，而这个时候我们无法知道parentSize的大小，所以理论上不可能测量处view的大小。

```
wrap_content int widthMeasureSpec = View.MeasureSpec.makeMeasureSpec((1<<30)-1,  
View.MeasureSpec.AT_MOST); int heightMeasureSpec = View.MeasureSpec.makeMeasureSpec((1<<30)-1,  
View.MeasureSpec.AT_MOST); v_view1.measure(widthMeasureSpec, heightMeasureSpec);
```

注意到 $(1<<30)-1$ ，我们知道MeasureSpec的前2位为mode，后面30位为size，所以说我们使用最大size值去匹配该最大化模式，让view自己去计算需要的大小。这个特殊的int值就是View理论上能支持的最大值。View的尺寸使用30位二进制来表示，也就是说最大是30个1（即 $2^{30}-1$ ），也就是 $(1<<30)-1$ 。

具体的数值(dp/px) 这种模式下，只需要使用具体数值去measure即可，比如宽/高都是100px：
int widthMeasureSpec = View.MeasureSpec.makeMeasureSpec(100, View.MeasureSpec.EXACTLY);
int heightMeasureSpec = View.MeasureSpec.makeMeasureSpec(100, View.MeasureSpec.EXACTLY);
v_view1.measure(widthMeasureSpec, heightMeasureSpec);

6.6. 为什么onCreate获取不到View的宽高

Activity在执行完onCreate, onResume之后才创建ViewRootImpl, ViewRootImpl进行View的绘制工作
调用链

startActivity->ActivityThread.handleLaunchActivity->onCreate ->完成DecorView和Activity的创建-
>handleResumeActivity->onResume()->DecorView添加到WindowManager->ViewRootImpl.performTraversals()
方法，测量(measure), 布局(layout), 绘制(draw)，从DecorView自上而下遍历整个View树。

6.7.View#post与Handler#post的区别

```
public boolean post(Runnable action) {  
    final AttachInfo attachInfo = mAttachInfo;  
    if (attachInfo != null) {  
        return attachInfo.mHandler.post(action);  
    }  
    getRunQueue().post(action);  
    return true;  
}
```

对于View#post当View已经attach到window，直接调用UI线程的Handler发送Runnable。如果View还未attach到window，将Runnable放入ViewRootImpl的RunQueue中，而不是通过MessageQueue。RunQueue的作用类似于MessageQueue，只不过这里面的所有Runnable最后的执行时机，是在下一个performTraversals到来的时候，也就是view完成layout之后的第一时间获取宽高，MessageQueue里的消息处理的则是下一次loop到来的时候。

6.8.Android绘制和屏幕刷新机制原理

绘制原理

<https://juejin.cn/post/6844904080989487118#heading-6>

<https://blog.csdn.net/freekiteyu/article/details/79483406>

<http://skyacer.github.io/2018/06/09/Android%E7%AA%97%E5%8F%A3%E7%AE%A1%E7%90%86%E5%88%86%E6%9E%90%EF%BC%88%E4%BA%8C%EF%BC%89%E2%80%94%E2%80%94%20WindowManagerService%E5%9B%BE%E5%B1%82%E7%AE%A1%E7%90%86%E4%B9%8B%E7%AA%97%E5%8F%A3%E7%9A%84%E6%B7%BB%E5%8A%A0/>

1.在App进程中创建PhoneWindow后会创建ViewRoot。ViewRoot的创建会创建一个Surface壳子，请求WMS填充Surface，WMS copyFrom()一个NativeSurface。

2.响应客户端事件，创建Layer(FrameBuffer)与客户端的Surface建立连接。

3.copyFrom()的同时创建匿名共享内存SharedClient(每一个应用和SurfaceFlinger之间都会创建一个SharedClient)

4.当客户端addView()或者需要更新View时，App进程的SharedBufferClient写入数据到共享内存ShareClient中，SurfaceFlinger中的SharedBufferServer接收到通知会将FrameBuffer中的数据传输到屏幕上。

绘制的过程 CPU准备数据，通过Driver层把数据交给GPU渲染，Display负责消费显示内容

1.CPU主要负责Measure、Layout、Record、Execute的数据计算工作

2.GPU负责Rasterization(栅格化(向量图形的格式表示的图像转换成位图用于显示器))、渲染，渲染好后放到buffer(图像缓冲区)里存起来。

3.Display(屏幕或显示器)屏幕会以一定的帧率刷新，每次刷新的时候，就会从缓存区将图像数据读取显示出来，如果缓存区没有新的数据，就一直用旧的数据，这样屏幕看起来就没有变

刷新机制 (<https://juejin.cn/post/6863756420380196877#heading-11>)

双缓存

屏幕刷新频是固定的，每16.6ms从buffer取数据显示完一帧，理想情况是帧率（GPU 在一秒内绘制操作的帧数，单位fps）和刷新频率保持一致，即每绘制完成一帧，显示器显示一帧，但是CPU/GPU写数据是不可控，所以会出现buffer里有些数据根本没显示出来就被重写了导致buffer抓取的帧并不是完整的一帧画面，即出现画面撕裂。

由于图像绘制和屏幕读取 使用的是同个buffer，所以屏幕刷新时可能读取到的是不完整的一帧画面。所以引入双缓存让绘制和显示器拥有各自的buffer：GPU 始终将完成的一帧图像数据写入到 Back Buffer，而显示器使用 Frame Buffer，当屏幕刷新时，Frame Buffer 并不会发生变化，当Back buffer准备就绪后，它们才进行交换。

什么时候进行交换 引入VSync

VSync (解决画面撕裂)

如果 Back buffer准备完成一帧数据以后就进行交换,时屏幕还没有完整显示上一帧内容的话，肯定是会出问题

如果 Frame buffer处理完一帧数据以后进行交换，可以。

vsync垂直同步利用 垂直同步脉冲（当扫描完一个屏幕后，设备需要重新回到第一行以进入下一次的循环，，此时屏幕没有在刷新，有一段时间空隙，这个时间点就是我们进行缓冲区交换的最佳时间。）保证双缓冲在最佳时间点才进行交换。

在Android4.1之前，屏幕刷新也遵循 上面介绍的 双缓存+VSync 机制

第2帧的CPU/GPU计算 没能在VSync信号到来前完成，屏幕平白无故地多显示了一次第1帧。

解决方式如果 Vsyn到来时 CPU/GPU就开始操作的话，是有完整的16.6ms的，这样应该会基本避免jank的出现了

为了优化显示性能，Google在Android 4.1系统中对Android Display系统进行了重构，实现了Project Butter (黄油工程)

1.drawing with VSync

一旦收到VSync通知（16ms触发一次），CPU和GPU 才立刻开始计算然后把数据写入buffer，可以让CPU/GPU有完整的16ms时间来处理数据，减少了jank。

2.三缓存

如果界面比较复杂，CPU/GPU的处理时间较长 超过了16.6ms, Back buffer正在被GPU用来处理B帧的数据， Frame buffer的内容用于Display的显示，这样两个buffer都被占用，CPU 则无法准备下一帧的数据,在Jank的阶段空空等待，存在CPU资源浪费。

三缓存就是在双缓冲机制基础上增加了一个 Graphic Buffer 缓冲区，这样可以最大限度的利用空闲时间，带来的坏处是多使用的一个 Graphic Buffer 所占用的内存。

让多增加一个Buffer给CPU用，让它提前忙起来，这样就能做到三方都有Buffer可用，CPU跟GPU不用争一个Buffer，真正实现并行处理

三缓冲有效利用了等待vysnc的时间，减少了jank，保证画面的连续性，提高柔韧性

3.Choreographer

Choreographer，编舞者。指对CPU/GPU绘制的指导，收到VSync信号才开始绘制，保证绘制拥有完整的16.6ms，避免绘制的随机性。控制只在vsync信号来时触发重绘呢

比如说绘制可能随时发起，封装一个Runnable丢给Choreography，下一个vsync信号来的时候，开始处理消息，然后真正的开始界面的重绘了。相当于UI绘制的节奏完全由Choreography来控制。

应用程序调用requestLayout发起重绘，通过Choreographer发送异步消息，请求同步vsync信号，即下一次vsync信号过来时，系统服务SurfaceFlinger在第一时间通知我们，触发UI绘制。虽然可以手动多次调用，但是在同一个vsync周期内，requestLayout只会执行一次。

6.9.Choreography原理

绘制是由应用端(任何时候都有可能)发起的，如果屏幕收到vsync信号，但是这一帧的还没有绘制完，就会显示上一帧的数据，这并不是因为绘制这一帧的时间过长(超过了信号发送周期)，只是信号快来的时候才开始绘制，如果频繁的出现的这种情况。一般调用requestLayout触发，这个函数随时都能调用，为了只控制在vsync信号来时触发重绘引入Choreography。ViewRoot.doTravle()->mChoreographer.postCallback

Choreographer对外提供了postCallback等方法，最终他们内部都是通过调用postCallbackDelayedInternal () 实现这个方法主要会做两件事情 1存储Action 请求垂直同步，垂直同步 2垂直同步回调立马执行 Action (CallBack/Runnable) 。

6.10.什么是双缓冲

通俗来讲就是有两个缓冲区，一个后台缓冲区和一个前台缓冲区，每次后台缓冲区接受数据，当填充完整后交换给前台缓冲，这样就保证了前台缓冲里的数据都是完整的。Surface 对应了一块屏幕缓冲区，是要显示到屏幕的内容的载体。每一个Window都对应了一个自己的Surface。这里说的window包括Dialog, Activity, Status Bar 等。SurfaceFlinger 最终会把这些Surface 在z轴方向上以正确的方式绘制出来(比如Dialog在Activity之上)。SurfaceView的每个Surface都包含两个缓冲区，而其他普通Window的对应的Surface则不是。

6.11.为什么使用SurfaceView

我们知道View是通过刷新来重绘视图，系统通过发出VSSYNC信号来进行屏幕的重绘，刷新的时间间隔是16ms,如果我们可以在16ms以内将绘制工作完成，则没有任何问题，如果我们绘制过程逻辑很复杂，并且我们的界面更新还非常频繁，这时候就会造成界面的卡顿，影响用户体验，为此Android提供了SurfaceView来解决这一问题。他们的UI不适合在主线程中绘制。对一些游戏画面，或者摄像头，视频播放等，UI都比较复杂，要求能够进行高效的绘制，因此，他们的UI不适合在主线程中绘制。这时候就必须给那些需要复杂而高效的UI视图生成一个独立的绘制表面Surface,并且使用独立的线程来绘制这些视图UI。

6.12.什么是SurfaceView

SurfaceView是View的子类，且实现了Parcelable接口且实现了Parcelable接口，其中内嵌了一个专门用于绘制的Surface，SurfaceView可以控制这个Surface的格式和尺寸，以及Surface的绘制位置。可以理解为Surface就是管理数据的地方，SurfaceView就是展示数据的地方。使用双缓冲机制，有自己的surface，在一个独立的线程里绘制。SurfaceView虽然具有独立的绘图表面，不过它仍然是宿主窗口的视图结构中的一个结点，因此，它仍然是可以参与到宿主窗口的绘制流程中去的。从SurfaceView类的成员函数draw和dispatchDraw的实现就可以看出，SurfaceView在其宿主窗口的绘图表面上面所做的操作就是将自己所占据的区域绘为黑色，除此之外，就没有其它更多的操作了，这是因为SurfaceView的UI是要展现在它自己的绘图表面上面的。优点：使用双缓冲机制，可以在一个独立的线程中进行绘制，不会影响主线程，播放视频时画面更流畅 缺点：Surface不在View hierarchy中，它的显示也不受View的属性控制，SurfaceView不能嵌套使用。在7.0版本之前不能进行平移，缩放等变换，也不能放在其它ViewGroup中，在7.0版本之后可以进行平移，缩放等变换。

6.13.View和SurfaceView的区别

View适用于主动更新的情况，而SurfaceView则适用于被被动更新的情况，比如频繁刷新界面。 View在主线程中对页面进行刷新，而SurfaceView则开启一个子线程来对页面进行刷新。 View在绘图时没有实现双缓冲机制，SurfaceView在底层机制中就实现了双缓冲机制。

6.14.SurfaceView为什么可以直接子线程绘制

通常View更新的时候都会调用ViewRootImpl中的performXXX()方法，在该方法中会首先使用checkThread()检查是否当前更新位于主线程，SurfaceView提供了专门用于绘制的Surface，可以通过SurfaceView来控制Surface的格式和尺寸，SurfaceView更新就不需要考虑线程的问题，它既可以在子线程更新，也可以在主线程更新。

6.15.SurfaceView、 TextureView、 SurfaceTexture、 GLSurfaceView

<https://zhooker.github.io/2018/03/24/SurfaceTexture%E7%9A%84%E5%8C%BA%E5%88%AB/>

SurfaceView：使用双缓冲机制，有自己的surface，在一个独立的线程里绘制，Android7.0之前不能平移、缩放

TextureView：它不会在WMS中单独创建窗口，而是作为一个普通View，可以和其它普通View一样进行移动，旋转，缩放，动画等变化。值得注意的是TextureView必须在硬件加速的窗口中。

SurfaceTexture：SurfaceTexture和SurfaceView不同的是，它对图像流的处理并不直接显示，而是转为OpenGL外部纹理，因此可用于图像流数据的二次处理（如Camera滤镜，桌面特效等）。 GLSurfaceView：SurfaceView不同的是，它加入了EGL的管理，并自带了渲染线程。

6.16.getWidth()方法和getMeasuredWidth()区别

①getMeasuredWidth方法获得的值是setMeasuredDimension方法设置的值，它的值在measure方法运行后就会确定

②getWidth方法获得是layout方法中传递的四个参数中的mRight-mLeft，它的值是在layout方法运行后确定的

③一般情况下在onLayout方法中使用getMeasuredWidth方法，而在除onLayout方法之外的地方用getWidth方法。

6.17.invalidate() 和 postInvalidate() 方法的区别

requestLayout：会触发三大流程。 invalidate：触发 onDraw 流程，在 UI 线程调用。 postInvalidate：触发 onDraw 流程，在非 UI 线程中调用。

1. view的invalidate递归调用父view的invalidateChildInParent，直到ViewRootImpl的invalidateChildInParent，然后触发performTraversals，会导致当前view被重绘，由于mLayoutRequested为false，不会导致onMeasure和onLayout被调用，而OnDraw会被调用
2. postInvalidate(),它可以在UI线程调用，也可以在子线程中调用，postInvalidate()方法内部通过Handler发送了一个消息将线程切回到UI线程通知重新绘制。最终还是调用了子View的invalidate()

6.18.Requestlayout, onlayout, onDraw, DrawChild区别与联系

requestLayout()方法：会导致调用 measure()过程 和 layout()过程, 不一定会触发OnDraw。 requestLayout会直接递归调用父窗口的requestLayout，直到ViewRootImpl,然后触发performTraversals，由于mLayoutRequested为true，会导致onMeasure和onLayout被调用。不一定会触发OnDraw，将会根据标志位判断是否需要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。

6.19.LinearLayout、FrameLayout 和 RelativeLayout 哪个效率高

简单布局 FrameLayout>LinearLayout>RelativeLayout 复杂布局 RelativeLayout>LinearLayout>FrameLayout

(1) Fragment是从上到下的一个堆叠的方式布局的，那当然是绘制速度最快，只需要将本身绘制出来即可，但是由于它的绘制方式导致在复杂场景中直接是不能使用的，所以工作效率来说Fragment仅使用于单一场景

(2) RelativeLayout会让子View调用2次onMeasure，LinearLayout 在有weight时，也会调用子View 2次onMeasure。由于RelativeLayout需要在横向和纵向分别进行一次measure过程。而LinearLayout只进行纵向或横向的测量，所以measure的时间会比RelativeLayout少很多。但是如果设置了 weight，在测量的过程中，LinearLayout会将设置过weight的和没设置的分别测量一次，这样就导致measure两次。

(3) 在不影响层级深度的情况下，使用LinearLayout和FrameLayout而不是RelativeLayout，复杂布局使用RelativeLayout

简单布局：在DecorView自己是FrameLayout但是它只有一个子元素是属于LinearLayout。因为DecorView的层级深度是已知而且固定的，上面一个标题栏，下面一个内容栏。采用RelativeLayout并不会降低层级深度，所以此时在根节点上用LinearLayout是效率最高的。

复杂布局：RelativeLayout 在性能上更好，使用 LinearLayout 容易产生多层嵌套的布局结构，这在性能上是不好的。而 RelativeLayout 通常层级结构都比较扁平，很多使用LinearLayout 的情况都可以用一个 RelativeLayout 来替代，以降低布局的嵌套层级，优化性能。

6.20.LinearLayout的绘制流程

onMeasure():1>：把 ViewRootImpl 的测量模式 传递给 DecorView，然后 DecorView 把测量模式 传递给 LinearLayout，遍历子元素并对每个子元素执行 measureChildBeforeLayout 方法，这个方法内部会调用子元素的 measure 方法，这样各个子元素就开始依次进入 measure 过程。2. LinearLayout类的 measureVertical 方法会遍历每一个子元素并且执行 LinearLayout类的 measureChildBeforeLayout 方法对子元素进行测量， LinearLayout类的 measureChildBeforeLayout 方法内部会执行子元素的 measure 方法。在代码中，变量 mTotalLength 会是用来存放 LinearLayout 在竖直方向上的当前高度，每遍历一个子元素， mTotalLength 就会增加 onLayout(): onLayout(): 其中会遍历调用每个子View的 setChildFrame 方法为子元素确定对应的位置 其中会遍历调用每个子View的 setChildFrame 方法为子元素确定对应的位置。其中的 childTop 会逐渐增大，意味着后面的子元素会被放置在靠下的位置。

6.21.自定义 View 的流程和注意事项

大多数自定义View要么是在onDraw方法中画点东西，和在onTouchEvent中处理触摸事件。

自定义View步骤：onMeasure，可以不重写，不重写的话就要在外面指定宽高，建议重写；onDraw，看情况重写，如果需要画东西就要重写；onTouchEvent，也是看情况，如果要做能跟手指交互的View，就重写；自定义View注意事项：

如果有自定义布局属性的，在构造方法中取得属性后应及时调用 recycle 方法回收资源；
onDraw 和 onTouchEvent 方法中都应尽量避免创建对象，过多操作可能会造成卡顿；

自定义ViewGroup步骤：

onMeasure（必须），在这里测量每一个子View，还有处理自己的尺寸；

onLayout（必须），在这里对子View进行布局；

如有自己的触摸事件，需要重写 onInterceptTouchEvent 或 onTouchEvent；

自定义ViewGroup注意事项：

如果想在 ViewGroup 中画点东西，又没有在布局中设置 background 的话，会画不出来，这时候需要调用 setWillNotDraw 方法，并设置为 false；

如果有自定义布局属性的，在构造方法中取得属性后应及时调用 recycle 方法回收资源；
onDraw 和 onTouchEvent 方法中都应尽量避免创建对象，过多操作可能会造成卡顿；

6.22.自定义View如何考虑机型适配

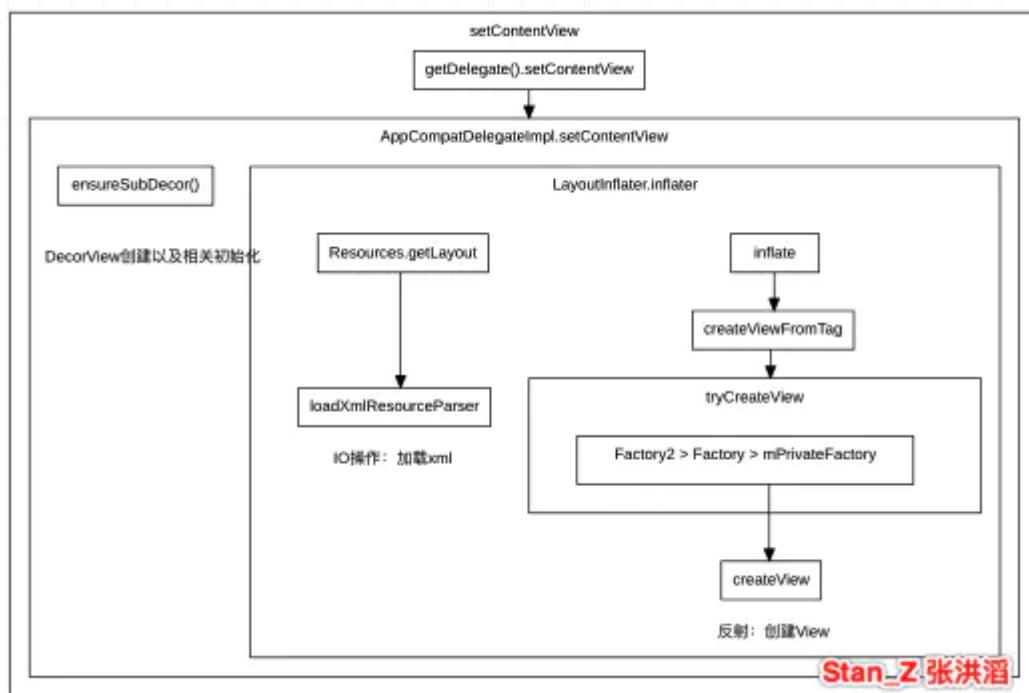
合理使用warp_content, match_parent。尽可能地使用RelativeLayout。
针对不同的机型，使用不同的布局文件放在对应的目录下，android会自动匹配。
尽量使用点9图片。
使用与密度无关的像素单位dp, sp。
引入android的百分比布局。
切图的时候切大分辨率的图，应用到布局当中，在小分辨率的手机上也会有很好的显示效果。

6.23.自定义控件优化方案

1.降低View.onDraw () 的复杂度 onDraw不要创建新的局部对象 onDraw不执行耗时操作
2.避免过度绘制 (Overdraw) 过度绘制会导致屏幕显示的色块不同，尽可能避免过度绘制的粉色 & 红色情况 移除默认的 Window 背景 @null 若不移除，则导致所有界面都多 1 次绘制 移除 控件中不必要的背景 对于1个ViewPager + 多个 Fragment 组成的首页界面，若每个 Fragment 都设有背景色，即 ViewPager 则无必要设置，可移除 减少布局文件的层级（嵌套）减少不必要的嵌套 ->> UI层级少 ->> 过度绘制的可能性低 自定义控件View优化：使用 clipRect()、quickReject() 给 Canvas 设置一个裁剪区域，只有在该区域内才会被绘制，区域之外的都不绘制

6.24.invalidate怎么局部刷新

6.25.View加载流程 (setContentView)



- 1.DecorView初始化
- 2.通过LayoutInflate对象去加载View，主要步骤是
 - (1) 通过xml的Pull方式去解析xml布局文件，获取xml信息，并保存缓存信息。
 - (2) 根据xml的tag标签通过反射创建View逐层构建View
 - (3) 递归构建其中的子View，并将子View添加到父ViewGroup中

7.View事件分发

7.1.View事件分发机制

<https://www.jianshu.com/p/e99b5e8bd67b>

三个角色

1、Activity：只有分发dispatchTouchEvent和消费onTouchEvent两个方法。事件由ViewRootImpl中DecorView dispatchTouchEvent分发Touch事件->Activity的dispatchTouchEvent()- DecorView。superDispatchTouchEvent-> ViewGroup的dispatchTouchEvent()。如果返回false直接掉用onTouchEvent, true表示被消费

2、ViewGroup：拥有分发、拦截和消费三个方法。：对应一个根ViewGroup来说，点击事件产生后，首先会传递给它，dispatchTouchEvent就会被调用，如果这个ViewGroup的onInterceptTouchEvent方法返回true就表示它要拦截当前事件，事件就会交给这个ViewGroup的onTouchEvent处理。如果这个ViewGroup的onInterceptTouchEvent方法返回false就表示它不拦截当前事件，这时当前事件就会继续传递给它的子元素，接着子元素的 dispatchTouchEvent方法就会被调用。

3、View：只有分发和消费两个方法。方法返回值为true表示当前视图可以处理对应的事件；返回值为false表示当前视图不处理这个事件，会被传递给父视图的

三个核心事件

1、dispatchTouchEvent(): 方法返回值为true表示事件被当前视图消费掉；返回为false表示停止往子View传递和分发,交给父类的onTouchEvent处理

2、onInterceptTouchEvent(): return false 表示不拦截，需要继续传递给子视图。return true 拦截这个事件并交由自身的onTouchEvent方法进行消费。

3、onTouchEvent(): return false 是不消费事件，会被传递给父视图的onTouchEvent方法进行处理。return true 是消费事件。

7.2.view的onTouchEvent, OnClickListerner和OnTouchListener的 onTouch方法 三者优先级

dispatchTouchEvent->onTouch->onInterceptTouchEvent->onTouchEvent。

1.dispatchTouchEvent中限制性mOnTouchListener.onTouch() onTouchListener的onTouch方法优先级比onTouchEvent高，会先触发。2.假如onTouch方法返回false会接着触发onTouchEvent，返回true,onTouchEvent方法不会被调用。3.onClick事件是在onTouchEvent的MotionEvent.ACTION_UP事件通过performClick() 触发的。OnTouchListener中onTouch方法如果返回true，则不会执行view的onTouchEvent方法，也就更不会执行view的onClickListener的onClick方法,返回false，则两个都会执行。

7.3.onTouch 和onTouchEvent 的区别

onTouch方法是View的 OnTouchListener借口中定义的方法。当一个View绑定了OnTouchListener后，当有touch事件触发时，就会调用onTouch方法处理点击事件在dispatchTouchEvent中掉用

onTouchListener的onTouch方法优先级比onTouchEvent高，会先触发。假如onTouch方法返回false，会接着触发onTouchEvent，反之onTouchEvent方法不会被调用。内置诸如click事件的实现等等都基于onTouchEvent，假如onTouch返回true，这些事件将不会被触发

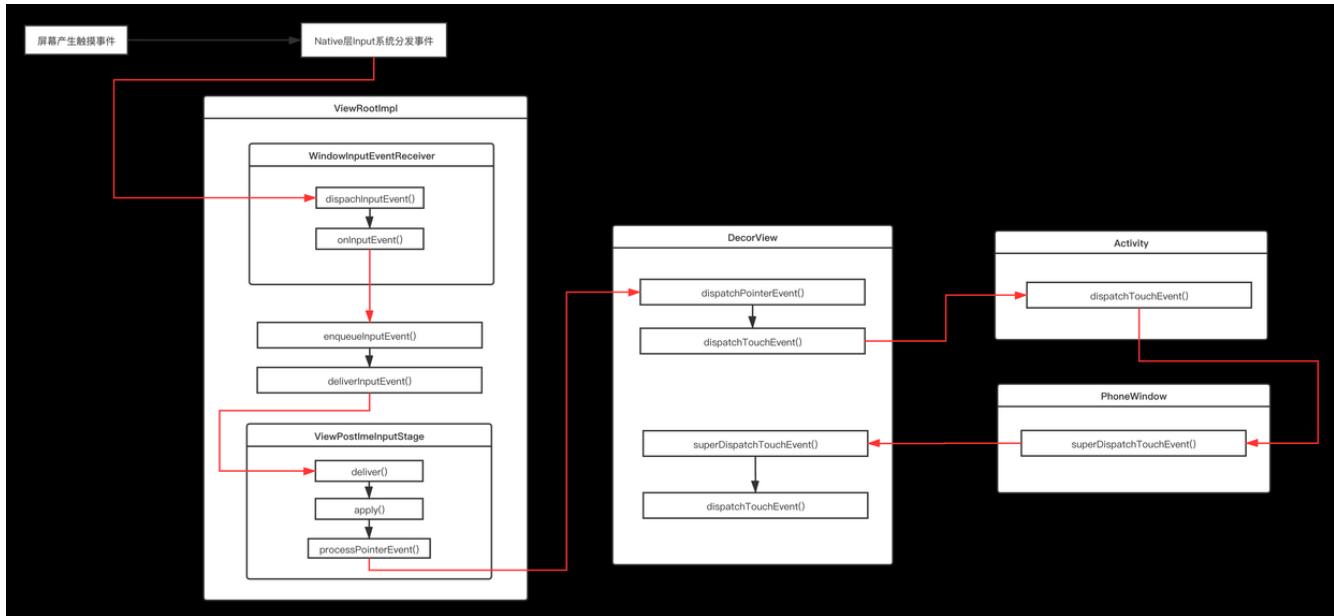
7.4.ACTION_CANCEL什么时候触发

- 1.如果在父View中拦截ACTION_UP或ACTION_MOVE，在第一次父视图拦截消息的瞬间，父视图指定子视图不接受后续消息了，同时子视图会收到ACTION_CANCEL事件。
- 2.如果触摸某个控件，但是又不是在这个控件的区域上抬起（移动到别的地方了），就会出现action_cancel。

7.5.事件是先到DecorView还是先到Window

DecorView -> Activity -> PhoneWindow -> DecorView

当屏幕被触摸input系统事件从Native层分发Framework层的InputEventReceiver.dispatchInputEvent()调用了ViewRootImpl.WindowInputEventReceiver.dispatchInputEvent()->ViewRootImpl中的DecorView.dispatchInputEvent()->Activity.dispatchInputEvent()->window.superDispatchTouchEvent()->DecorView.superDispatchTouchEvent()->Viewgroup.superDispatchTouchEvent()



7.6.点击事件被拦截，但是想传到下面的View，如何操作

重写子类的requestDisallowInterceptTouchEvent()方法返回true就不会执行父类的onInterceptTouchEvent(), 可将点击事件传到下面的View, 剥夺了父view 对除了ACTION_DOWN以外的事件的处理权。

7.7.如何解决View的事件冲突

常见开发中事件冲突的有ScrollView与RecyclerView的滑动冲突、 RecyclerView内嵌同时滑动同一方向

滑动冲突的实现方法：

外部拦截法：指点击事件都先经过父容器的拦截处理，如果父容器需要此事件就拦截，否则就不拦截。具体方法：需要重写父容器的onInterceptTouchEvent方法，在内部做出相应的拦截。 内部拦截法：指父容器不拦截任何事件，而将所有的事件都传递给子容器，如果子容器需要此事件就直接消耗，否则就交由父容器进行处理。具体方法：需要配合requestDisallowInterceptTouchEvent方法。

<https://www.jianshu.com/p/982a83271327>

外部拦截法：

父View在ACTION_MOVE中开始拦截事件，那么后续ACTION_UP也将默认交给父View处理！

内部拦截法：

即父View不拦截任何事件，所有事件都传递给子View，子View根据需要决定是自己消费事件还是给父View处理

如果父容器需要获取点击事件则调用 parent.requestDisallowInterceptTouchEvent(false)方法，让父容器去拦截事件

7.8.在 ViewGroup 中的 onTouchEvent 中消费 ACTION_DOWN 事件，ACTION_UP 事件是怎么传递

一个事件序列只能被一个View拦截且消耗。因为一旦一个元素拦截了此事件，那么同一个事件序列内的所有事件都会直接交给它处理（即不会再调用这个View的拦截方法去询问它是否要拦截了，而是把剩余的ACTION_MOVE、ACTION_DOWN等事件直接交给它来处理）。

Activity.dispatchTouchEvent() -> ViewGroup1.dispatchTouchEvent() -> ViewGroup1.onInterceptTouchEvent() ->
view1.dispatchTouchEvent() -> view1.onTouchEvent() -> ViewGroup1.onTouchEvent()

-> Activity.dispatchTouchEvent() -> ViewGroup1.dispatchTouchEvent() -> ViewGroup1.onTouchEvent()

7.9. Activity ViewGroup 和 View 都不消费 ACTION_DOWN，那么 ACTION_UP 事件是怎么传递的

ACTION_DOWN:-> Activity.dispatchTouchEvent() -> ViewGroup1.dispatchTouchEvent() ->
ViewGroup1.onInterceptTouchEvent() -> view1.dispatchTouchEvent() -> view1.onTouchEvent() ->
ViewGroup1.onTouchEvent() -> Activity.onTouchEvent();

**ACTION_MOVE > Activity.dispatchTouchEvent() ->
Activity.onTouchEvent(); -> 消费**

7.10. 同时对父 View 和子 View 设置点击方法，优先响应哪个

优先响应子 view，，如果先响应父 view，那么子 view 将永远无法响应，父 view 要优先响应事件，必须先调用 onInterceptTouchEvent 对事件进行拦截，那么事件不会再往下传递，直接交给父 view 的 onTouchEvent 处理。

7.11. requestDisallowInterceptTouchEvent 的调用时机

事件分发例子

<https://blog.csdn.net/lmj623565791/article/details/39102591>

8. RecycleView

8.1. RecyclerView 的多级缓存机制，每一级缓存具体作用是什么，分别在什么场景下会用到哪些缓存

<https://zhooker.github.io/2017/08/14/%E5%85%B3%E4%BA%8ERecyclerview%E7%9A%84%E7%BC%93%E5%AD%98%E6%9C%BA%E5%88%B6%E7%9A%84%E7%90%86%E8%A7%A3/>

<https://www.wanandroid.com/wenda/show/14222>

<https://blog.csdn.net/u013700502/article/details/105058771>

<https://juejin.cn/post/6854573221702795277#heading-9>

Scrap、Cache、ViewCacheExtension、RecycledViewPool

Scrap缓存用在RecyclerView布局时，布局完成之后就会清空

添加到Cache缓存和RecyclerViewPool缓存的item，他们的View必须已经从RecyclerView中detached或removed

一级缓存：mAttachedScrap 和 mChangedScrap 二级缓存：mCachedViews 三级缓存：ViewCacheExtension 四级缓存：RecycledViewPool 然后说怎么用，就是先从 1 级找，然后 2 级...然后4 级，找不到 create ViewHolder。

<https://www.jianshu.com/p/467ae8a7ca6e>

mAttachedScrap/mChangedScrap

屏幕内缓存

RecyclerView 的滑动场景来说，新卡位的复用以及旧卡位的回收机制，不会涉及到 mChangedScrap 和 mAttachedScrap

notifyItemChanged/rangeChange，此时如果Holder发生了改变那么就放入changeScrap中，反之放入到 AttachScrap。

mCachedViews

当列表滑动出了屏幕时，ViewHolder会被缓存在 mCachedViews，其大小由mViewCacheMax决定，默认 DEFAULT_CACHE_SIZE为2，可通过Recyclerview.setItemViewCacheSize()动态设置。

ViewCacheExtension

可以自己实现ViewCacheExtension类实现自定义缓存，可通过Recyclerview.setViewCacheExtension()设置。

缓存池

ViewHolder在首先会缓存在 mCachedViews 中，当超过了数（比如默认为2），就会添加到 RecycledViewPool 中。RecycledViewPool 会根据每个ViewType把ViewHolder分别存储在不同的列表中，每个ViewType最多缓存 DEFAULT_MAX_SCRAP = 5 个ViewHolder

8.2.RecyclerView的滑动回收复用机制

<https://www.jianshu.com/p/467ae8a7ca6e>

RecyclerView 滑动的场景触发的回收复用机制工作时，并不需要四级缓存都参与的。

1.RecyclerView 向下滑动操作的日志，第三行5个卡位的显示都是重新创建的 ViewHolder

新一行5个卡位和复用不可能会用到刚移出屏幕的5个卡位，因为先复用再回收，新一行的5个卡位先去目前的 mCachedViews 和 ViewPool 的缓存中寻找复用，没有就重新创建，然后移出屏幕的那行的5个卡位再回收缓存到 mCachedViews 和 ViewPool 里面，

2.RecyclerView 再次向上滑动重新显示第一行的5个卡位时，只有后面3个卡位触发了 onBindViewHolder() 方法，重新绑定数据。

滑动场景下涉及到的回收和复用的结构体是 mCachedViews 和 ViewPool，前者默认大小为2，后者为5。所以，当第三行显示出来后，第一行的5个卡位被回收，回收时先缓存在 mCachedViews，满了再移出旧的到 ViewPool 里，所有5个卡位有2个缓存在 mCachedViews 里，3个缓存在 ViewPool，所以最新的两个卡位是0、1，会放在 mCachedViews 里，而2、3、4的卡位则放在 ViewPool 里。

3.而至于为什么会创建了17个 ViewHolder，那是因为再第四行的卡位要显示出来时，ViewPool 里只有3个缓存，而第四行的卡位又用不了 mCachedViews 里的2个缓存，因为这两个缓存的是6、7卡位的 ViewHolder，所以就需要再重新创建2个 ViewHolder 来给第四行最后的两个卡位使用。

8.3.RecyclerView的刷新回收复用机制

notifyXxx后会RecyclerView会进行两次布局，一次预布局，一次实际布局，然后执行动画操作

dispatchLayoutStep1

查找改变holder，并保存在mChangedScrap中；其他未改变的保存到mAttachedScrap中（mChangedScrap保存的holder信息只有预布局时才会被复用）

dispatchLayoutStep2

此步骤会创建一个新的holder并执行绑定数据，充当改变位置的holder，其他位置holder从mAttachedScrap中获取

8.4.RecyclerView 为什么要预布局

<https://juejin.cn/post/6890288761783975950#heading-0>

why

这种负责执行动画的View在原布局或新布局中不存在的动画，就是预测动画。

因为RecyclerView 要执行预测动画。比如有A,B,C三个itemView，其中A和B被加载到屏幕上，这时候删除B后，按照最终效果我们会看到C移动到B的位置；因为我们只知道 C 最终的位置，但是不知道 C 的起始位置在哪里(即C还未被加载)。

用户有 A、B、C 三个 item，A，B 刚好显示在屏幕中，这个时候，用户把 B 删除了，那么最终 C 会显示在 B 原来的位置

因为我们只知道 C 最终的位置，但是不知道 C 的起始位置在哪里，无法确定 C 应该从哪里滑动过来。

在其他 LayoutManager 中，它可能是从侧面或者是其他地方滑动过来的。

what

当 Adapter 发生变化的时候，RecyclerView 会让 LayoutManager 进行两次布局。

第一次，预布局，为动画前的表项先执行一次pre-layout，根据 Adapter 的 notify 信息，我们知道哪些 item 即将变化了，将不可见的表项 3 也加载到布局中，形成一张布局快照（1、2、3）。

第二次，实际布局，也就是变化完成之后的布局同样形成一张布局快照（1、3）。

这样只要比较前后布局的变化，就能得出应该执行什么动画了，就称为预测动画。

8.5.ListView 与 RecyclerView区别

1.布局效果

ListView 的布局比较单一，只有一个纵向效果； RecyclerView 的布局效果丰富，可以在 LayoutManager 中设置：线性布局（纵向，横向），表格布局，瀑布流布局

2.局部刷新

RecyclerView中可以实现局部刷新，例如：notifyItemChanged();

如果要在ListView实现局部刷新，依然是可以实现的，当一个item数据刷新时，我们可以在Adapter中，实现一个notifyItemChanged()方法，在方法里面通过这个 item 的 position，刷新这个item的数据

3.缓存区别

ListView有两级缓存，在屏幕与非屏幕内。 RecyclerView比ListView多两级缓存 ListView缓存View。 RecyclerView缓存RecyclerView.ViewHolder

8.6.RecyclerView性能优化

1.数据处理与视图加载分离

简单来说就是在onBindViewHolder()只设置UI显示，不做任何逻辑判断，需要的业务逻辑在得到javabean之前处理好，

2.布局优化

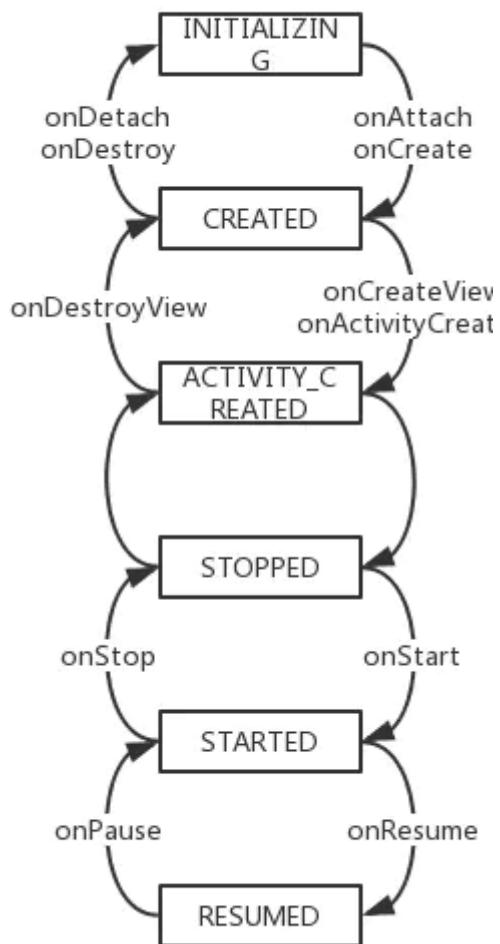
减少过渡绘制 减少布局层级

3.设置RecyclerView.addOnScrollListener()来在滑动过程中停止加载的操作。

9.Viewpager&Fragment

9.1.Fragment的生命周期 & 结合Activity的生命周期

<https://juejin.cn/post/6844903752114126855#heading-0>



9.2.Activity和Fragment的通信方式， Fragment之间如何进行通信

Activity和Fragment

1.采用Bundle的方式 在activity中建一个bundle， 把要传的值存入bundle， 然后通过fragment的setArguments (bundle) 传到fragment，在fragment中，用getArguments接收。

2.采用接口回调的方式

3.EventBus的方式

4.viewModel 做数据管理， activity 和 fragment 公用同个viewModel 实现数据传递

Fragment之间

1.EventBus的方式

2.采用接口回调的方式

3.Fragment 通过 getActivity 获取到Activity， Activity通过findFragmentByTag || findFragmentById获取 Fragment,Fragment 实现接口.

9.3.为什么使用Fragment.setArguments(Bundle)传递参数

<https://www.jianshu.com/p/c06efe090589>

Activity.onCreate(Bundle savedInstanceState)->Fragment.instantiate()

当再次重建时会通过空参构造方法反射出新的fragment。并且给mArguments初始化为原先的值，而原来的Fragment实例的数据都丢失了。

Activity重新创建时，会重新构建它所管理的Fragment，原先的Fragment的字段值将会全部丢失，但是通过Fragment.setArguments(Bundle bundle)方法设置的bundle会保留下来，并在重建时恢复。所以，尽量使用Fragment.setArguments(Bundle bundle)方式来进行参数传递。

9.4.FragmentPagerAdapter和FragmentStatePagerAdapter区别及使用场景

使用FragmentPagerAdapter时 页面切换，只是调用detach，而不是remove，所以只执行onDestroyView，而不是onDestroy，不会摧毁Fragment实例，只会摧毁Fragment 的View；

使用FragmentStatePagerAdapter时 页面切换，调用remove，执行onDestroy。直接摧毁Fragment。

FragmentPagerAdapter最好用在少数静态Fragments的场景，用户访问过的Fragment都会缓存在内存中，即使其视图层次不可见而被释放(onDestroyView)。因为Fragment可能保存大量状态，因此这可能会导致使用大量内存。

页面很多时，可以考虑FragmentStatePagerAdapter

9.5.fragment懒加载

判断当前 Fragment 是否对用户可见，只是 onHiddenChanged() 是在 add+show+hide 模式下使用， setUserVisibleHint 是在 ViewPager+Fragment 模式下使用。

<https://juejin.cn/post/6844904050698223624#heading-0>

<https://www.jianshu.com/p/bef74a4b6d5e>

老的懒加载处理方案

对 ViewPager 中的 Fragment 懒加载

方法1：继承模式

通过继承懒加载Fragment基类，在 setUserVisibleHint 中判断可见并且当 onViewCreated() 表明 View 已经加载完毕后再掉用加载方法。

方法2：代理+反射模式

1.adapter 中 getItem 时 new 一个代理 fragment

2.setUserVisibleHint 中根据反射得到真正的 fragment

3. 通过 add commit 把真正的 fragment 添加到代理 fragment 中

方法1：不可见的 Fragment 执行了 onResume() 方法。因为 setUserVisibleHint 位于 onCreateView 之前，此时为 false，onResume 之后为 true，在 true 后加载相当于 onResume() 等方法在真实的 onCreateView 之前调用，不可见的 Fragment 执行了 onResume() 方法。

Androidx 下的懒加载

在 FragmentPagerAdapter 与 FragmentStatePagerAdapter 新增了含有 behavior 字段的构造函数

如果 behavior 的值为 BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT，那么当前选中的 Fragment 在 Lifecycle.State#RESUMED 状态，其他不可见的 Fragment 会被限制在 Lifecycle.State#STARTED 状态

原因：FragmentPagerAdapter 在其 setPrimaryItem 方法中调用了 setMaxLifecycle，所以说 onResume 中执行懒加载

9.6. ViewPager2 与 ViewPager 区别

<https://juejin.cn/post/6844904020553760782#heading-0>

1. FragmentStateAdapter 替代 FragmentStatePagerAdapter，PagerAdapter 被 RecyclerView.Adapter 替代
2. 支持竖直滑动，禁止滑动
3. PageTransformer 用来设置页面动画，设置页面间距
4. 预加载当 setOffscreenPageLimit 被设置为 OFFSCREEN_PAGE_LIMIT_DEFAULT 时候会使用 RecyclerView 的缓存机制。

9.7. fragment 嵌套问题

10. WebView

10.1. 如何提高 WebView 加载速度

<https://tech.meituan.com/2017/06/09/webviewperf.html>

WebView 启动过程大概分为以下几个阶段：



App 中打开 WebView 的第一步并不是建立连接，而是启动浏览器内核。

1. 优化手段围绕着以下两个点进行

预加载WebView。

加载WebView的同时，请求H5页面数据。

2.常见的方法是

全局WebView

在客户端刚启动时，就初始化一个全局的WebView待用，并隐藏；

这种方法可以比较有效的减少WebView在App中的首次打开时间。当用户访问页面时，不需要初始化WebView的时间。

当然这也带来了一些问题，包括：

额外的内存消耗。

页面间跳转需要清空上一个页面的痕迹，更容易内存泄露。

客户端代理页面请求WebView初始化完成后向客户端请求数据

在客户端初始化WebView的同时，直接由native开始网络请求数据；

当页面初始化完成后，向native获取其代理请求的数据。

asset存放离线包。

3.除此之外还有一些其他的优化手段：

DNS和链接慢

想办法复用客户端使用的域名和链接，可以让客户端复用使用的域名与链接。

DNS采用和客户端API相同的域名

DNS会在系统级别进行缓存，对于WebView的地址，如果使用的域名与native的API相同，则可以直接使用缓存的DNS而不用再发起请求图片。

脚本执行慢

可以把框架代码拆分出来，在请求页面之前就执行好。

后端处理慢

可以让服务器分trunk输出，在后端计算的同时前端也加载网络静态资源。

10.2.WebView与js的交互

https://blog.csdn.net/carson_ho/article/details/64904691

Android去调用JS的代码

1.通过WebView的loadUrl ()

2.通过WebView的evaluateJavascript ()

JS调用Android代码的方法

1.通过WebView的addJavascriptInterface () 进行对象映射

2.通过 WebViewClient 的shouldOverrideUrlLoading ()方法回调拦截 url

3.Android通过 WebChromeClient 的onJsAlert()、onJsConfirm()、onJsPrompt (方法回调分别拦截JS对话框 (即上述三个方法) , 得到他们的消息内容, 然后解析即可。

10.3.WebView的漏洞

https://blog.csdn.net/carson_ho/article/details/64904635

任意代码执行漏洞

JS调用Android的可以通过addJavascriptInterface接口进行对象映射

当JS拿到Android这个对象后, 就可以调用这个Android对象中所有的方法, 包括系统类 (java.lang.Runtime 类) , 从而进行任意代码执行。

java.lang.Runtime 类, 可以执行本地命令的

解决

对于Android 4.2以前, 需要采用**拦截prompt ()** 的方式进行漏洞修复

原理

每次当 WebView 加载页面前加载一段本地的 JS 代码,

让JS调用一Javasctipt方法: 该方法是通过调用prompt () 把JS中的信息 (含特定标识, 方法名称等) 传递到 Android端;

在Android的onJsPrompt () 中, 解析传递过来的信息, 再通过反射机制调用Java对象的方法, 这样实现安全的JS调用Android代码。

对于Android 4.2以后, 则只需要对被调用的函数以 @JavascriptInterface进行注解

密码明文存储漏洞

WebView默认开启密码保存功能

原因 WebView默认开启密码保存功能: mWebView.setSavePassword(true) 开启后, 在用户输入密码时, 会弹出提示框: 询问用户是否保存密码; 如果选择“是”, 密码会被明文保到 /data/data/com.package.name/databases/webview.db 中, 这样就有被盗取密码的危险 解决 关闭密码保存提醒: WebSettings.setSavePassword(false)

域控制不严格漏洞

当其他应用启动可以允许外部调用的Activity 时, intent 中的 data 直接被当作 url 来加载 (假定传进来的 url 为 file:///data/local/tmp/attack.html) , 其他 APP 通过使用显式 ComponentName 或者其他类似方式就可以很轻松的启动该 WebViewActivity 并加载恶意url。

对于不需要使用 file 协议的应用, 禁用 file 协议; // 禁用 file 协议; setAllowFileAccess(false);
setAllowFileAccessFromFileURLs(false); setAllowUniversalAccessFromFileURLs(false);

对于需要使用 file 协议的应用, 禁止 file 协议加载 JavaScript。

10.4.JsBridge原理

<https://juejin.cn/post/6844903585268891662#heading-0>

<https://www.jianshu.com/p/910e058a1d63>

1.优点

1.Javascript 端可以确定 JSBridge 的存在，直接调用即可

2.H5同时适配Android和iOS两个平台

3.java与js的交互存在一些安全漏洞

2.原理

JavaScript 调用 Native

注入 API 和 拦截 URL SCHEME。

注入API

在 4.2 之前，Android 注入 JavaScript 对象的接口是 addJavascriptInterface，但是这个接口有漏洞，可以被不法分子利用，危害用户的安全，因此在 4.2 中引入新的接口 @JavascriptInterface（上面代码中使用的）来替代这个接口，解决安全问题。所以 Android 注入对对象的方式是有兼容性问题的。

拦截 URL SCHEME

```
mWebView.registerHandler("startload", (data, function) -> { function.onCallBack("aaaaa");});
```

1. 初始化 webview 时，将 WebViewJavascriptBridge.js 文件注入页面。向 body 中添加一个不可见的 iframe 元素。通过改变一个不可见的 iframe 的 src 就可以让 webview 拦截到 url，而用户是无感知的。

2. Web 端通过某种方式（例如 iframe.src）发送 URL Scheme 请求，通过 shouldOverrideUrlLoading 来拦截约定规则的 Ur

Native 调用 JavaScript

```
mWebView.callHandler("test", mJson, data -> LogUtil.d(回调:" + data));
```

```
webView.loadUrl("javascript:" + javaScriptString);
```

dosend 发送消息时，带上 callId，如果 Js 在调用 Handler 的时候设置了回调方法，会调用 queueMessage 的方法，然后往下就是走 Native 给 Js 发送消息的步骤。

11.动画

https://blog.csdn.net/carson_ho/article/details/79860980

<https://anriku.top/2018/09/01/Android%E5%B1%9E%E6%80%A7%E5%8A%A8%E7%94%BB%E6%8E%A2%E7%B4%A2/>

https://blog.csdn.net/carson_ho/article/details/72827747

<https://juejin.cn/post/6846687601118691341#heading-12>

11.1. 动画的类型

11.1.1. 视图动画

https://blog.csdn.net/carson_ho/article/details/72827747 // 以下参数是 4 种动画效果的公共属性，即都有的属性
android:duration="3000" // 动画持续时间 (ms)，必须设置，动画才有效果 android:startOffset ="1000" // 动画延迟开始时间 (ms) android:fillBefore = "true" // 动画播放完后，视图是否会停留在动画开始的状态，默认为 true
android:fillAfter = "false" // 动画播放完后，视图是否会停留在动画结束的状态，优先于 fillBefore 值，默认为 false

android:fillEnabled= "true" // 是否应用fillBefore值，对fillAfter值无影响，默认为true android:repeatMode= "restart" // 选择重复播放动画模式，restart代表正序重放，reverse代表倒序回放，默认为restart | android:repeatCount = "0" // 重放次数（所以动画的播放次数=重放次数+1），为infinite时无限重复 android:interpolator = @nim/interpolator_resource // 插值器，即影响动画的播放速度，下面会详细讲

11.1.2 补间动画

平移动画 (Translate) // 1. fromXDelta : 视图在水平方向x 移动的起始值 // 2. toXDelta : 视图在水平方向x 移动的结束值 // 3. fromYDelta : 视图在竖直方向y 移动的起始值 // 4. toYDelta: 视图在竖直方向y 移动的结束值 缩放动画 (scale) android:fromXScale="0.0" // 动画在水平方向X的起始缩放倍数 // 0.0表示收缩到没有；1.0表示正常无伸缩 // 值小于1.0表示收缩；值大于1.0表示放大 android:toXScale="2" //动画在水平方向X的结束缩放倍数 android:fromYScale="0.0" //动画开始前在竖直方向Y的起始缩放倍数 android:toYScale="2" //动画在竖直方向Y的结束缩放倍数 android:pivotX="50%" // 缩放轴点的x坐标 android:pivotY="50%" // 缩放轴点的y坐标 旋转动画 (rotate) android:fromDegrees="0" // 动画开始时 视图的旋转角度(正数 = 顺时针，负数 = 逆时针) android:toDegrees="270" // 动画结束时 视图的旋转角度(正数 = 顺时针，负数 = 逆时针) android:pivotX="50%" // 旋转轴点的x坐标 android:pivotY="0" // 旋转轴点的y坐标 透明度动画 (alpha) android:fromAlpha="1.0" // 动画开始时视图的透明度(取值范围: -1 ~ 1) android:toAlpha="0.0"// 动画结束时视图的透明度(取值范围: -1 ~ 1)

11.1.3 逐帧动画

https://blog.csdn.net/carson_ho/article/details/73087488

按序播放一组预先定义好的图片

xml:animation-list

// item = 动画图片资源； duration = 设置一帧持续时间(ms)

属性动画

what

顾名思义，通过控制对象的属性，来实现动画效果。官方定义：定义一个随着时间（注：停个顿）更改任何对象属性的动画，无论其是否绘制到屏幕上

链接：<https://www.jianshu.com/p/821ef6d1e1c9>

Duration：定义动画时长，默认是300 ms。

Time interpolation: 时间插值器，它可以指定属性值如何随时间变化的，反应了动画的运动速率。

Repeat count and behavior: 指定当动画结束时是否重复动画以及动画重复多少次，还可以设置反向播放动画，播放到达指定次数后动画结束。

Animator sets：把一组动画聚在一起，顺序播放或者同时播放或者延迟播放。

Frame refresh delay：指定刷新动画帧的频率，默认时间是10ms，但是刷新频率最终取决于系统是否繁忙以及系统服务底层计时器的快慢。

why

11.2. 补间动画和属性动画的区别

a. 作用对象局限：View

补间动画只能够作用在视图View上，即只可以对一个Button、TextView、甚至是LinearLayout、或者其它继承自View的组件进行动画操作，但无法对非View的对象进行动画操作。

有些情况下的动画效果只是视图的某个属性 & 对象而不是整个视图；如，现需要实现视图的颜色动态变化，那么就需要操作视图的颜色属性从而实现动画效果，而不是针对整个视图进行动画操作

b. 没有改变View的属性，只是改变视觉效果

补间动画只是改变了View的视觉效果，而不会真正去改变View的属性。

如，将屏幕左上角的按钮通过补间动画移动到屏幕的右下角

点击当前按钮位置（屏幕右下角）是没有效果的，因为实际上按钮还是停留在屏幕左上角，补间动画只是将这个按钮绘制到屏幕右下角，改变了视觉效果而已。

c. 动画效果单一

补间动画只能实现平移、旋转、缩放 & 透明度这些简单的动画需求

11.3. ObjectAnimator, ValueAnimator及其区别

ObjectAnimator

位移

val objectAnimation = ObjectAnimator.ofFloat(llAddAccount, "translationX", 0f, -70f) 第三参数为可变长参数，第一个值为动画开始的位置，第二个值为结束值得位置，如果数组大于3位数，那么前者将是后者的起始位置

旋转 val objectAnimation = ObjectAnimator.ofFloat(tvText, "rotation", 0f, 180f, 0f)

ofFloat()方法的可变长参数，如果后者的值大于前者，那么顺时针旋转，小于前者，则逆时针旋转。

缩放

```
val objectAnimation = ObjectAnimator.ofFloat(tvText, "scaleX", 1f, 2f)
```

ofFloat()方法传入参数属性为scaleX和scaleY时，动态参数表示缩放的倍数

透明 val objectAnimation = ObjectAnimator.ofFloat(tvText, "alpha", 1f, 0f, 1f)

ValueAnimator

val valueAnimator = ValueAnimator.ofFloat(0f, 180f) valueAnimator.addUpdateListener { tvText.rotationY = it.animatedValue as Float //手动赋值 } valueAnimator.start()

等价于

```
ObjectAnimator.ofFloat(tvText, "rotationY", 0f, 180f).apply { start() }
```

ValueAnimator作为ObjectAnimator的父类，主要动态计算目标对象属性的值，然后设置给对象属性，达到动画效果

使用ValueAnimator实现动画，需要手动赋值给目标对象tvText的rotationY，而ObjectAnimator则是自动赋值，不需要手动赋值就可以达到效果

ObjectAnimator和ValueAnimator区别

其实二者都是属于属性动画，本质上是一样的，都是先改变值，然后赋值给对象属性，从而实现动画操作。

但二者区别就在于，ValueAnimator类是 手动 赋值给对象的属性，从而实现动画，

而ObjectAnimator类，是自动赋值给对象的属性，从

AnimatorSet

一个动画结束后播放另外一个动画，或者同时播放

```
val aAnimator=ObjectAnimator.ofInt(1) val bAnimator=ObjectAnimator.ofInt(1) val  
cAnimator=ObjectAnimator.ofInt(1) val dAnimator=ObjectAnimator.ofInt(1)
```

```
AnimatorSet().apply { play(aAnimator).before(bAnimator)//a 在b之前播放 play(bAnimator).with(cAnimator)//b  
和c同时播放动画效果 play(dAnimator).after(cAnimator)//d 在c播放结束之后播放 start() }
```

11.4. TimeInterpolator插值器，自定义插值器

TimeInterpolator

它的作用是根据时间流逝的百分比来计算出当前属性值改变的百分比

插值器，Interpolator负责控制动画变化的速率，使得基本的动画效果能够以匀速、加速、减速、抛物线速率等各种速率变化

匀速插值器 <https://blog.csdn.net/qinxiandiqi/article/details/51719926> 图1假设了一个对象需要对它的x属性设定动画，这个x属性代表这个对象在屏幕上面的横坐标。这个动画的时间设定为40毫秒，以及需要移动的距离是40个像素。每过10毫秒（默认的帧频率），这个对象就会在水平方向上移动10个像素。等到40毫秒之后，这个动画停止，对象在水平方向上总共移动了40个像素。这个例子中的动画使用了一个线性插值器，意味着这个对象以匀速移动。

例子: http://static.kancloud.cn/alex_wsc/android_art/1828610

，当时间t=20ms的时候，时间流逝的百分比是0.5 ($20/40=0.5$) 意味着现在时间过了一半，那x应该改变多少呢这个就由插值器和估值算法来确定

拿线性插值器来说，当时间流逝一半的时候，x的变换也应该是一半，即x的改变是0.5，为什么呢？因为它是线性插值器，是实现匀速动画的

估值器evaluate的三个参数分别表示估值小数、开始值和结束值，对应于我们的例子就分别是0.5、0、40。根据上述算法，整型估值返回给我们的结果是20，这就是 ($x=20, t=20\text{ms}$) 的由来。

系统已有的插值器：①LinearInterpolator（线性插值器）：匀速动画。②AccelerateDecelerateInterpolator（加速减速插值器）：动画两头慢，中间快。③DecelerateInterpolator（减速插值器）：动画越来越慢。

自定义插值器

写一个自定义Interpolator：先减速后加速

```
public class DecelerateAccelerateInterpolator implements TimeInterpolator {
```

```
@Override  
public float getInterpolation(float input) {  
    float result;  
    if (input <= 0.5) {
```

```
result = (float) (Math.sin(Math.PI * input)) / 2;
// 使用正弦函数来实现先减速后加速的功能，逻辑如下：
// 因为正弦函数初始弧度变化值非常大，刚好和余弦函数是相反的
// 随着弧度的增加，正弦函数的变化值也会逐渐变小，这样也就实现了减速的效果。
// 当弧度大于π/2之后，整个过程相反了过来，现在正弦函数的弧度变化值非常小，渐渐随着弧度继续增加，变化
值越来越大，弧度到π时结束，这样从0过度到π，也就实现了先减速后加速的效果
} else {
    result = (float) (2 - Math.sin(Math.PI * input)) / 2;
}
return result;
// 返回的result值 = 随着动画进度呈先减速后加速的变化趋势
}
```

https://blog.csdn.net/carson_ho/article/details/72863901

11.5. TypeEvaluator估值器

TypeEvaluator

估值器 据当前属性改变的百分比来计算改变后的属性值

①IntEvaluator：针对整型属性 ②FloatEvaluator：针对浮点型属性 ③ArgbEvaluator：针对Color属性

```
public class FloatEvaluator implements TypeEvaluator {
// FloatEvaluator实现了TypeEvaluator接口

// 重写evaluate() public Object evaluate(float fraction, Object startValue, Object endValue) {
// 参数说明 // fraction：表示动画完成度（根据它来计算当前动画的值） // startValue、endValue：动画的初始值和
结束值 float startFloat = ((Number) startValue).floatValue();
```

```
return startFloat + fraction * (((Number) endValue).floatValue() - startFloat);
// 初始值 过渡 到结束值 的算法是：
// 1\ . 用结束值减去初始值，算出它们之间的差值
// 2\ . 用上述差值乘以fraction系数
// 3\ . 再加上初始值，就得到当前动画的值
}
```

实例

实现的动画效果：一个圆从一个点 移动到 另外一个点

https://blog.csdn.net/carson_ho/article/details/72863901

1.onDraw画圆 canvas.drawCircle(x, y, RADIUS, mPaint)

2:创建动画对象 & 设置初始值 和 结束值 ValueAnimator anim = ValueAnimator.ofObject(new PointEvaluator(),
startPoint, endPoint);

3.自定义PointEvaluator,evaluate根据插值器的速率计算出Point 的x,y 4.通过 值 的更新监听器，将改变的对象手动
赋值给当前对象

5.每次赋值后就重新绘制，从而实现动画效果

12.Bitmap

<https://my.oschina.net/zbj1618/blog/2961367>

12.1.Bitmap 内存占用的计算

占用的内存大小 = 像素总数量 (图片宽x高) × 每个像素的字节大小

每个像素的字节大小与Bitmap的色彩模式有关

public static final Bitmap.Config ALPHA_8 //代表8位Alpha位图 每个像素占用1byte内存

public static final Bitmap.Config ARGB_4444 //代表16位ARGB位图 每个像素占用2byte内存

public static final Bitmap.Config ARGB_8888 //代表32位ARGB位图 每个像素占用4byte内存

public static final Bitmap.Config RGB_565 //代表8位RGB位图 每个像素占用2byte内存

假设这张图片是ARGB_8888的，那这张图片占的内存就是 $\text{width} * \text{height} * 4$ 个字节或者 $\text{width} * \text{height} * \text{inTargetDensity} / \text{inDensity} * 4$

加载一张本地Res、Raw资源图片，得到的是图片的原始尺寸 * 缩放系数(inDensity)

inTargetDensity 为当前屏幕像素密度(宽平方+高平方)/尺寸。

inDensity默认为图片所在文件夹对应的密度()

densityDpi 160 240 320 480 640

资源目录dpi mdpi hdpi xhdpi xxhdpi xxxhdpi;

像素密度 5.0英寸的手机的屏幕分辨率为1280x720，那么像素密度为192dpi

占用的内存 = $\text{width} * \text{height} * \text{targetDensity} / \text{inDensity} * 4$

读取SD卡上的图,得到的是图片的原始尺寸

占用的内存 = $\text{width} * \text{height} * 4$

12.2.getByteCount() & getAllocationByteCount()的区别

如果被复用的Bitmap的内存比待分配内存的Bitmap大

getByteCount()获取到的是当前图片应当所占内存大小

getAllocationByteCount()获取到的是被复用Bitmap真实占用内存大小

在复用Bitmap的情况下，getAllocationByteCount()可能会比getByteCount()大。

12.3.Bitmap的压缩方式

质量压缩，不会对内存产生影响；

采样率压缩，比例压缩，会对内存产生影响；

质量压缩

不会减少图片的像素，它是在保持像素的前提下改变图片的位深及透明度等

图片的长，宽，像素都不变，那么bitmap所占内存大小是不会变的

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
src.compress(Bitmap.CompressFormat.JPEG, quality, baos);
byte[] bytes = baos.toByteArray();
Bitmap mBitmap = BitmapFactory.decodeByteArray(bytes, 0, bytes.length);
```

RGB_565法

改变一个像素所占的内存

```
BitmapFactory.Options options2 = new BitmapFactory.Options();
options2.inPreferredConfig = Bitmap.Config.RGB_565;
bm = BitmapFactory.decodeFile(Environment.getExternalStorageDirectory().getAbsolutePath()
+ "/DCIM/Camera/test.jpg", options2);
```

采样率压缩

设置inSampleSize的值(int类型)后，假如设为n，则宽和高都为原来的1/n，宽高都减少，内存降低

```
BitmapFactory.Options options = new BitmapFactory.Options();
options.inSampleSize = 2;
bm = BitmapFactory.decodeFile(Environment.getExternalStorageDirectory().getAbsolutePath()
+ "/DCIM/Camera/test.jpg", options);
```

比例压缩

根据图片的缩放比例进行等比大小的缩小尺寸，从而达到压缩的效果

压缩的图片文件尺寸变小，但是解码成bitmap后占得内存变小

Android中使用Matrix对图像进行缩放、旋转、平移、斜切等变换的。

```
Matrix matrix = new Matrix();
matrix.setScale(0.5f, 0.5f);
bm = Bitmap.createBitmap(bit, 0, 0, bit.getWidth(), bit.getHeight(), matrix, true);
```

12.4.LruCache & DiskLruCache原理

LRU (Least Recently Used)

<https://blog.csdn.net/u010983881/article/details/79050209>

LruCache的核心思想就是维护一个缓存对象列表，从表尾访问数据，在表头删除数据。对象列表的排列方式是按照访问顺序实现，就是当访问的数据项在链表中存在时，则将该数据项移动到表尾，否则在表尾新建一个数据项。当链表容量超过一定阈值，则移除表头的数据。

利用LinkedHashMap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accessOrder设置为true则为访问顺序，为false，则为插入顺序。

写入缓存

1插入元素，并相应增加当前缓存的容量。

2调用trimToSize()开启一个死循环，不断的从表头删除元素，直到当前缓存的容量小于最大容量为止。

读取缓存

调用LinkedHashMap的get()方法，注意如果该元素存在，这个方法会将该元素移动到表尾。

DiskLruCache

<https://juejin.cn/post/6844903556705681421#heading-6>

<https://nich.work/2017/DiskLruCache/>

<https://www.jianshu.com/p/400bda3e37ed>

利用 LinkedHashMap实现算法LRU

DiskLruCache 有三个内部类，分别为 Entry、Snapshot 和 Editor。

Entry是 DiskLruCache 内 LinkedHashMap Value 的基本结构。

Journal 文件

DiskLruCache 通过在磁盘中创建并维护一个简单的Journal文件来记录各种缓存操作，。记录的类型有4种，分别为 READ、 REMOVE、 CLEAN和DIRTY。

写入缓存的时候会向journal文件写入一条以DIRTY开头的数据表示正在进行写操作，当写入完毕时，分两种情况：

1、写入成功，会向journal文件写入一条以CLEAN开头的文件，其中包括该文件的大小。 2、写入失败，会向journal文件写入一条以REMOVE开头的文件,表示删除了该条缓存。也就是说每次写入缓存总是写入两条操作记录。

读取的时候，会向journal文件写入一条以READ开头的文件,表示进行了读操作

删除的时候，会向journal文件写入一条以REMOVE开头的文件,表示删除了该条缓存

通过journal就记录了所有对缓存的操作。并且按照从上到下的读取顺序记录了对所有缓存的操作频繁度和时间顺序。这样当退出程序再次进来调用缓存时，就可以读取这个文件来知道哪些缓存用的比较频繁了。然后把这些操作记录读取到集合中，操作的时候就可以直接从集合中去对应的数据了。

在缓存记录之外，Journal 文件在初始化创建的时候还有一些固定的头部信息，包括了文件名、版本号和 valueCount(决定每一个 key 能匹配的 Entry 数量)。

读取

Snapshot是Entry的快照(snapshot)。当调用 diskLruCache.get(key)时，便能获得一个Snapshot对象，该对象可用于获取或更新存于磁盘的缓存

```
String key = util.hashKeyForDisk(util.IMG_URL);
DiskLruCache.Snapshot snapshot = diskLruCache.get(key);
if (snapshot != null) {
    InputStream in = snapshot.getInputStream(0);
    return BitmapFactory.decodeStream(in);
}
```

1获取到缓存文件的输入流，等待被读取。

2向journal写入一行READ开头的记录，表示执行了一次读取操作。

3如果缓存总大小已经超过了设定的最大缓存大小或者操作次数超过了2000次，就开一个线程将集合中的数据删除到小于最大缓存大小为止并重新写journal文件。

4返回缓存文件快照，包含缓存文件大小，输入流等信息。

写入

```
DiskLruCache.Editor editor = diskLruCache.edit(key);
if (editor != null) {
    OutputStream outputStream = editor.newOutputStream(0);
    if (downloadUrlToStream(Util.IMG_URL, outputStream)) {
        publishProgress("");
        //写入缓存
        editor.commit();
    } else {
        //写入失败
        editor.abort();
    }
}
diskLruCache.flush();
```

1从集合中找到对应的实例（如果没有创建一个放到集合中），然后创建一个editor，将editor和entry关联起来。

2向journal中写入一行操作数据（DITTY 空格 和key拼接的文字），表示这个key当前正处于编辑状态。

newOutputStream

1向journal文件写入一行CLEAN开头的字符（包括key和文件的大小，文件大小可能存在多个 使用空格分开的）

2重新比较当前缓存和最大缓存的大小，如果超过最大缓存或者journal文件的操作大于2000条，就把集合中的缓存删除一部分，直到小于最大缓存，重新建立新的journal文件

12.5.如何设计一个图片加载库

<https://juejin.cn/post/6844904099297624077#heading-0>

- 1.对图片进行内存压缩；
- 2.高分辨率的图片放入对应文件夹；
- 3.缓存
- 4.及时回收

12.6.有一张非常大的图片,如何去加载这张大图片

12.7.如果把drawable-xxhdpi下的图片移动到drawable-xhdpi下，图片内存是如何变的。

验证原图：1000宽X447高，位于drawable-xxhdpi (480dpi) 文件包，设备Pixel-XL (560dpi)。使用默认Bitmap.Config=ARGB_8888,设置inSampleSize=2

缩放比=主动设置×被动设置=1/2×(560/480)=0.5×1.166=0.5833

一个像素所占的内存=ARGB_8888=32bit=4byte

原始大小=1000×447

width * height * *

内存占用=(原始宽×缩放比)×(原始高×缩放比)×targetDensity/inDensity×一个像素所占的内存
=1000×0.5833×447×0.5833×4 =583×260×4 =606320byte ≈0.578MB

12.8.如果在hdpi、xxhdpi下放置了图片，加载的优先级。如果是400800，10801920，加载的优先级。

<https://cloud.tencent.com/developer/article/1015960>

优先会去更高密度的文件夹下找这张图片，我们当前的场景就是drawable-xxxhdpi文件夹，然后发现这里也没有android_logo这张图，接下来会尝试再找更高密度的文件夹，发现没有更高密度的了，这个时候会去drawable-nodpi文件夹找这张图，发现也没有，那么就会去更低密度的文件夹下面找，依次是drawable-xhdpi -> drawable-hdpi -> drawable-mdpi -> drawable-ldp

0dpi ~ 120dpi ldpi

120dpi ~ 160dpi mdpi

160dpi ~ 240dpi hdpi

240dpi ~ 320dpi xhdpi

320dpi ~ 480dpi xxhdpi

480dpi ~ 640dpi xxxhdpi

13.mvc&mvp&mvvm

1.MVC及其优缺点

原理

视图层(View)

一般采用XML文件进行界面的描述，这些XML可以理解为AndroidApp的View。

控制层(Controller)

Android的控制层的重任通常落在了众多的Activity的肩上。

模型层(Model)

我们针对业务模型，建立的数据结构和相关的类，就可以理解为AndroidApp的Model，Model是与View无关，而与业务相关的。对数据库的操作、对网络等的操作都应该在Model里面处理，当然对业务计算等操作也是必须放在该层的。

缺点

随着界面及其逻辑的复杂度不断提升，Activity类的职责不断增加，以致变得庞大臃肿。

2.MVP及其优缺点

原理

MVP框架由3部分组成：View负责显示，Presenter负责逻辑处理，Model提供数据。

View: 显示数据，并向Presenter报告用户行为。与用户进行交互(在Android中体现为Activity)。

Presenter: 逻辑处理，从Model拿数据，回显到UI层，响应用户的行为。

Model:负责存储、检索、操纵数据(有时也实现一个Model interface用来降低耦合)。

google todo-mvp加入契约类来统一管理view与presenter的所有的接口，这种方式使得view与presenter中有哪些功能，一目了然

优点

1.分离视图逻辑和业务逻辑，降低了耦合，修改视图而不影响模型，不需要改变Presenter的逻辑 模型与视图完全分离，我们可以修改视图而不影响模型；

2.视图逻辑和业务逻辑分别抽象到了View和Presenter的接口中，Activity只负责显示，代码变得更加简洁，提高代码的阅读性。

3.Presenter被抽象成接口，可以有多种具体的实现，所以方便进行单元测试。

Presenter是通过interface与View(Activity)进行交互的，这说明我们可以通过自定义类实现这个interface来模拟Activity的行为对Presenter进行单元测试，省去了大量的部署及测试的时间（不需要将应用部署到Android模拟器或真机上，然后通过模拟用户操作进行测试）

缺点

1.那就是对 UI 的操作必须在 Activity 与 Fragment 的生命周期之内，更细致一点，最好在 onStart() 之后 onPause() 之前，否则极其容易出现各种异常，内存泄漏。

2.Presenter与View之间的耦合度高，app中很多界面都使用了同一个Presenter。一旦需要变更，那么视图需要变更了。

MVP如何设计避免内存泄漏？

Mvp模式在封装的时候会造成内存泄漏，因为presenter层，需要做网络请求，所以就需要考虑到网络请求的取消操作，如果不处理，activity销毁了，presenter层还在请求网络，就会造成内存泄漏。**如何解决Mvp模式造成内存泄漏？**只要presenter层能感知activity生命周期的变化，在activity销毁的时候，取消网络请求，就能解决这个问题。下面开始封装activity和presenter。

定义IPresenter 声明activity (Fragment) 生命周期中的各个回调方法

```
<U extends IUI> void init(BaseActivity activity, U ui);  
  
/**  
 * onUICreate:UI被创建的时候应该invoke这个method. <br/>  
 * <p/>  
 * 比如Activity的onCreate()、Fragment的onCreateView()的方法应该调用Presenter的这个方法  
 *  
 * @param savedInstanceState 保存了的状态  
 */  
void onUICreate(Bundle savedInstanceState);  
  
/**  
 * onUIStart:在UI被创建和被显示到屏幕之间应该回调这个方法. <br/>
```

```
* <p/>
* 比如Activity的onStart()方法应该调用Presenter的这个方法
*/
void onUIStart();

/**
* onUIResume:在UI被显示到屏幕的时候应该回调这个方法. <br/>
* <p/>
* 比如Activity的onResume()方法应该调用Presenter的这个方法
*/
void onUIResume();

/**
* onUIPause:在UI从屏幕上消失的时候应该回调这个方法. <br/>
* <p/>
* 比如Activity的onPause()方法应该调用Presenter的这个方法
*/
void onUIPause();

/**
* onUIStop:在UI从屏幕完全隐藏应该回调这个方法. <br/>
* <p/>
* 比如Activity的onStop()方法应该调用Presenter的这个方法
*/
void onUIStop();

/**
* onUIDestroy:当UI被Destory的时候应该回调这个方法. <br/>
*/
void onUIDestroy();

/**
* onSaveInstanceState:保存数据. <br/>
* <p/>
* 一般是因为内存不足UI的状态被回收的时候调用
*
* @param outState 待保存的状态
*/
void onSaveInstanceState(Bundle outState);

/**
* onRestoreInstanceState:当UI被恢复的时候被调用. <br/>
*
* @param savedInstanceState 保存了的状态
*/
void onRestoreInstanceState(Bundle savedInstanceState);
```

封装 BaseActivity 拥有一个 protected P mPresenter实例，类型写成泛型，protected修饰，子类实现构造方法中，对mPresenter进行实例化：采用反射的形式（避免让子类进行实例化，繁琐）

```
public abstract class BaseMVPActivity<P extends IPresenter> extends BaseActivity {

    protected P mPresenter;
```

```
public BaseMVPActivity(){
    this.mPresenter = createPresenter();
}

protected P createPresenter(){
    ParameterizedType type = (ParameterizedType)(getClass().getGenericSuperclass());
    if(type == null){
        return null;
    }
    Type[] typeArray = type.getActualTypeArguments();
    if(typeArray.length == 0){
        return null;
    }
    Class<P> clazz = (Class<P>) typeArray[0];
    try {
        return clazz.newInstance();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InstantiationException e) {
        e.printStackTrace();
    }
    return null;
}

@Override
protected void onDestroy() {
    mPresenter.onDestroy();
    super.onDestroy();
}
```

封装BasePresenter 3.定义BasePresenter，页面销毁的回调里面。处理网络请求 定义一个集合，把每次网络请求装载到集合里面，在页面销毁的时候，取消所有的网络请求。防止内存泄漏。

```
public abstract class BasePresenter<U extends IUI> implements IPresenter {

    @Override
    public void onDestroy() {
        // 清空call列表，并放弃call请求
        clearAndCancelCallList();

    }
}
```

3.MVVM及其优缺点

View层包含布局,以及布局生命周期控制器(Activity/Fragment)

ViewModel与Presenter大致相同,都是负责处理数据和实现业务逻辑,但 ViewModel层不应该直接或者间接地持有 View层的任何引用

model层是数据层

Model, 模型层, 即数据模型, 用于获取和存储数据。

View, 视图, 即Activity/Fragment

ViewModel, 视图模型, 负责业务逻辑。

MVVM 的本质是 数据驱动, 把解耦做的更彻底, viewModel不持有view。

View 产生事件, 使用 ViewModel进行逻辑处理后, 通知Model更新数据, Model把更新的数据给ViewModel, ViewModel自动通知View更新界面, 而不是主动调用View的方法

LiveData是具有生命周期的可观察的数据持有类。理解它需要注意这几个关键字, 生命周期, 可观察, 数据持有类。

DataBinding用来实现View层与ViewModel数据的双向绑定, Data Binding 減輕原本 MVP 中 Presenter 要與 p 和 View 互動的職責

MVVM的优点

核心思想是观察者模式, 它通过事件和转移View层数据持有权来实现View层与ViewModel层的解耦.

1.耦合度更低, 复用性更强, 没有内存泄漏

2.结合jetpack, 写出更优雅的代码

缺点

ViewModel与View层的通信变得更加困难了,所以在一些极其简单的页面中请酌情使用,否则就会有一种脱裤子放屁的感觉,在使用MVP这个道理也依然适用.

4.MVC与MVP区别

View与Model并不直接交互, 而是通过与Presenter交互来与Model间接交互。而在MVC中View可以与Model直接交互。MVP 隔离了MVC中的 M 与 V 的直接联系后, 靠 Presenter 来中转, 所以使用 MVP 时 P 是直接调用 View 的接口来实现对视图的操作的,

5.MVP如何管理Presenter的生命周期, 何时取消网络请求

<https://blog.csdn.net/mq2553299/article/details/78927617>

<https://www.jianshu.com/p/0d07fba84cb8>

使用RxLifecycle, 通过监听Activity、Fragment的生命周期, 来自动断开subscription以防止内存泄漏。

RxLifecycle原理

操作符

1.takeUntil 发射来自原始Observable的数据, 如果第二个Observable发射了一项数据或者发射了一个终止通知, 原始Observable会停止发射并终止。

2.CombineLatest 当两个Observables中的任何一个发射了数据时, 使用一个函数结合每个Observable发射的最近数据项, 并且基于这个函数的结果发射数据。

流程

1.BehaviorSubject 在ActivityActivity不同的生命周期, BehaviorSubject对象会发射对应的ActivityEvent, 比如在onCreate()生命周期发射ActivityEvent.CREATE, 在onStop()发射ActivityEvent.STOP。

BehaviorSubject, 实际上也还是一个Observable

BehaviorSubject (释放订阅前最后一个数据和订阅后接收到的所有数据)

2.LifecycleTransformer 在自己的请求中bindActivity返回的是LifecycleTransformer

LifecycleTransformer中使用upstream.takeUntil(observable),

upstream就是原始的Observable

observable是Activity中的BehaviorSubject

指定生命周期断开

原始的Observable通过takeUntil和BehaviorSubject绑定，当BehaviorSubject发出数据即Activity的生命周期走到和你指定销毁的事件一样时,BehaviorSubject才会把事件传递出去,原始的Observable就终止发射数据了

不指定生命周期断开

combineLatest() 当两个Observables中的任何一个发射了数据时，使用一个函数结合每个Observable发射的最近数据项，并且基于这个函数的结果发射数据。不指定生命周期时bindToLifecycle中通过combineLatest组合

比如说我们在onCreate()中执行了bindToLifecycle，那么lifecycle.take(1)指的就是ActivityEvent.CREATE，经过map(correspondingEvents)，这个map中传的函数就是1中的ACTIVITY_LIFECYCLE

lifecycle.skip(1)就简单了，除去第一个保留剩下的，以ActivityEvent.Create为例，这里就剩下：

ActivityEvent.START ActivityEvent.RESUME ActivityEvent.PAUSE ActivityEvent.STOP ActivityEvent.DESTROY

三个参数 意味着，lifecycle.take(1).map(correspondingEvents)的序列和lifecycle.skip(1)进行combine，形成一个新的序列：

false,false,fasle,false,true

这意味着，当Activity走到onStart生命周期时，为false,这次订阅不会取消，直到onDestroy，为true，订阅取消。

最终还是和lifecycle (BehaviorSubject) 发射的数据比较，如果两个一样说明Activity走到了该断开的生命周期了，upstream.takeUntil(observable)中的observable就要发通知告诉upstream (原始的Observable) 该断开了。

14.Binder

<https://juejin.im/post/6844903589635162126#heading-1>

https://blog.csdn.net/carson_ho/article/details/73560642

<https://www.colabug.com/2019/0421/6041679/>

<https://juejin.im/post/6844903469971685390#heading-0>

14.1.Android中进程和线程的关系,区别

1、进程是什么？

它是系统进行资源分配和调度的一个独立单位,也就是说进程是可以独立运行的一段程序。

2、线程又是什么？

线程进程的一个实体，是CPU调度和分派的基本单位，他是比进程更小的能独立运行的基本单位,线程自己基本上不拥有系统资源。在运行时，只是暂用一些计数器、寄存器和栈。

1、进程有不同的代码和数据空间，而多个线程则共享数据空间，每个线程有自己的执行堆栈和程序计数器为其执行上下文。

2、进程间相互独立，同一进程的各线程间共享。

3、进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。

14.2.为何需要进行IPC,多进程通信可能会出现什么问题

为了保证进程空间不被其他进程破坏或干扰，Linux中的进程是相互独立或相互隔离的。在Android系统中一个应用默认只有一个进程，每个进程都有自己独立的资源和内存空间，其它进程不能任意访问当前进程的内存和资源。这样导致在不同进程的四大组件没法进行通信，线程间没法做同步，静态变量和单例也会失效。所以需要有一套IPC机制来解决进程间通信、数据传输的问题。

开启多进程虽简单，但会引发如下问题，必须引起注意。

- 1.静态成员和单例模式失效
- 2.线程同步机制失效
- 3.SharedPreferences 可靠性降低
- 4.Application 被多次创建

对于前两个问题，可以这么理解，在Android中，系统会为每个应用或进程分配独立的虚拟机，不同的虚拟机自然占有不同的内存地址空间，所以同一个类的对象会产生不同的副本，导致共享数据失败，必然也不能实现线程的同步。由于SharedPreferences底层采用读写XML的文件的方式实现，多进程并发的的读写很可能导致数据异常。

Application被多次创建和前两个问题类似，系统在分配多个虚拟机时相当于把同一个应用重新启动多次，必然会导致 Application 多次被创建，为了防止在 Application 中出现无用的重复初始化，可使用进程名来做过滤，只让指定进程的才进行全局初始：

```
public class MyApplication extends Application{  
    @Override  
    public void onCreate() {  
        super.onCreate();  
        String processName = "com.shh.ipctest";  
        if (getPackageName().equals(processName)){  
            // do some init  
        }  
    }  
}
```

14.3.Android中IPC方式有几种、各种方式优缺点

进程间通信的方式-对比

名称	优点	缺点	适用场景
Intent	简单易用	只能传输Bundle所支持的数据类型	四大组件间的进程间通信
文件共享	简单易用	不适合高并发	简单的数据共享，无高并发场景
AIDL	功能强大，支持一对多并发实时通信	使用稍微复杂，需要注意线程同步	复杂的进程间调用，Android中最常用
Messenger	比AIDL稍微简单易用些	比AIDL功能弱，只支持一对多串行实时通信	简单的进程间通信
ContentProvider	强大的数据共享能力，可通过call方法扩展	受约束的AIDL，主要对外提供数据线的CRUD操作	进程间的大量数据共享
RemoteViews	在跨进程访问UI方面有奇效	比较小众的通信方式	某些特殊的场景
Socket	跨主机，通信范围广	只能传输原始的字节流	常用于网络通信中

14.4.为何新增Binder来作为主要的IPC方式

Android也是基于Linux内核，Linux现有的进程通信手段有管道/消息队列/共享内存/套接字/信号量。

既然有现有的IPC方式，为什么重新设计一套Binder机制呢

主要是出于以上三个方面的考量：

1、效率：传输效率主要影响因素是内存拷贝的次数，拷贝次数越少，传输速率越高。从Android进程架构角度分析：对于消息队列、Socket和管道来说，数据先从发送方的缓存区拷贝到内核开辟的缓存区中，再从内核缓存区拷贝到接收方的缓存区，一共两次拷贝。

一次数据传递需要经历：用户空间 -> 内核缓存区 -> 用户空间，需要2次数据拷贝，这样效率不高。

而对于Binder来说，数据从发送方的缓存区拷贝到内核的缓存区，而接收方的缓存区与内核的缓存区是映射到同一块物理地址的，节省了一次数据拷贝的过程：共享内存不需要拷贝，Binder的性能仅次于共享内存。

2、稳定性：上面说到共享内存的性能优于Binder，那为什么不采用共享内存呢，因为共享内存需要处理并发同步问题，容易出现死锁和资源竞争，稳定性较差。Binder基于C/S架构，Server端与Client端相对独立，稳定性较好。

3、安全性：传统Linux IPC的接收方无法获得对方进程可靠的UID/PID，从而无法鉴别对方身份；而Binder机制为每个进程分配了UID/PID，且在Binder通信时会根据UID/PID进行有效性检测。

14.5.什么是Binder

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(Binder类 IBinder)；

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 驱动会自动完成代理对象和本地对象之间的转换。

从Android Framework角度来说，Binder是ServiceManager连接各种Manager和相应ManagerService的桥梁
Binder跨进程通信机制：基于C/S架构，由Client、Server、ServerManager和Binder驱动组成。

进程空间分为用户空间和内核空间。用户空间不可以进行数据交互；内核空间可以进行数据交互，所有进程共用一个内核空间

Client、Server、ServiceManager均在用户空间中实现，而Binder驱动程序则是在内核空间中实现的；

14.6.Binder的原理

Binder Driver 如何在内核空间中做到一次拷贝的

进程空间分为用户空间和内核空间。用户空间不可以进行数据交互；内核空间可以进行数据交互，所有进程共用一个内核空间。

应用程序不能直接操作设备硬件地址,如果用户空间需要读取磁盘的文件，如果不采用内存映射，需要两次拷贝（磁盘-->内核空间-->用户空间）；

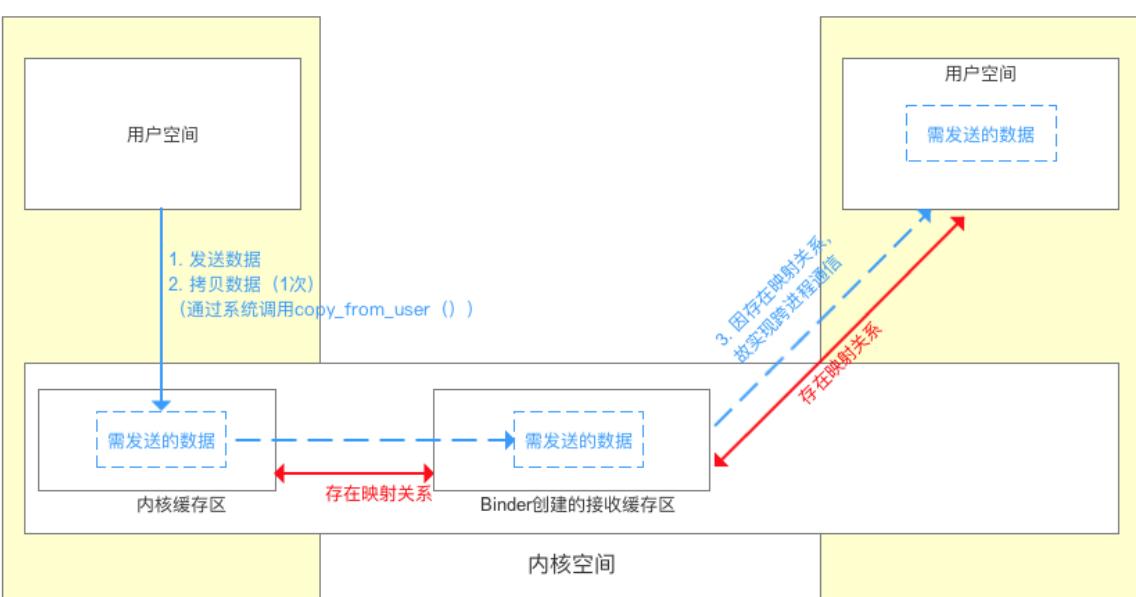
内存映射将用户空间的一块内存区域映射到内核空间。映射关系建立后，内核空间对这段区域的修改也能直接反应到用户空间,少了一次拷贝。

Binder 驱动使用 mmap() 在内核空间创建数据接收的缓存空间。 mmap(NULL, MAP_SIZE, PROT_READ, MAP_PRIVATE, fd, 0)的返回值是内核空间映射在用户空间的地址

1.Binder 驱动在内核空间创建一个数据接收缓存区。

2.在内核空间开辟一块内核缓存区，建立内核缓存区和内核空间的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 到内核空间的内核缓存区，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。

工作流程	<ol style="list-style-type: none">1. Binder驱动 创建一块 接收缓存区2. 实现地址映射关系：即 根据需映射的接收进程信息，实现 内核缓存区 和 接收进程用户空间地址 同时映射到 同1个共享接收缓存区中 (注：前2个阶段仅创建了虚拟区间 & 映射关系，但并无将传输数据；真正的数据传输时刻：当进程发起读 / 写操作时)3. 发送进程 通过 系统调用copy_from_user () 发送数据到虚拟内存区域（数据拷贝1次）4. 由于 内核缓存区 & 接收进程的用户空间地址 存在映射关系（同时映射Binder创建的接收缓存区中），故相当于也发送到了接收进程的用户空间地址，即实现了跨进程通信
示意图	
优点	<ul style="list-style-type: none">• 传输效率高：数据拷贝次数少（1次）、用户空间 & 内核空间可直接通过共享对象直接交互• 为接收进程 分配了不确定大小的接收缓存区

14.7. 使用Binder进行数据传输的具体过程

系统层面：

注册服务

服务进程向Binder进程发起服务注册

Binder驱动将注册请求转发给ServiceManager进程

ServiceManager进程添加这个服务进程

获取服务

用户进程向Binder驱动发起获取服务的请求，传递要获取的服务名称Binder驱动

将该请求转发给ServiceManager进程

ServiceManager进程查到到用户进程需要的服务进程信息最后

通过Binder驱动将上述服务信息返回个用户进程

使用服务

1.Binder通过内存映射建立数据缓存区

2. 根据ServiceManager查到的服务的进程和数据缓存区，数据缓存区和client进程的内存缓存区建立映射

3. client掉用copy_from_user数据到内存缓存区

4. 收到binder启动后服务进程根据用户进程要求调用目标方法

5. 服务进程将目标方法的结果返回给用户进程

步骤	过程描述	
1. 注册服务	1. Server进程 向 Binder驱动 发起服务注册请求 2. Binder驱动 将注册请求转发给Service Manager进程 3. Service Manager进程 添加该Server进程（即已注册服务）	此时，ServiceManager进程拥有了Server进程的信息
2. 获取服务	1. Client 向 Binder 驱动发起获取服务的请求，传递要获取的服务名称 2. Binder 驱动将该请求转发给 ServiceManager 进程 3. ServiceManager 查找到 Client 需要的 Server 对应的服务信息 4. 通过 Binder 驱动将上述服务信息返回给 Client进程	此时，Client进程与 Server进程已经建立了连接
	步骤1： Binder驱动为跨进程通信作准备：实现内存映射 (调用mmap () 系统函数)	1. Binder驱动 创建一块 接收缓存区 2. 实现地址映射关系：即 根据 ServiceManager进程里的Server信息找到对应的Server 进程，实现 内核缓存区 和 Server 进程用户空间地址 同时映射到 同1个接收缓存区中 <i>(注：此时仅创建了虚拟区间 & 映射关系，但并无将传输数据)</i>
	步骤2： Client进程 将参数数据发送到Server进程	1. Client进程 通过 系统调用copy_from_user () 发送数据到内核空间中的缓存区；(当前线程被挂起) <i>(由于 内核缓存区 & 接收进程的用户空间地址 存在映射关系 (同时映射Binder创建的接收缓存区中)，故相当于也发送到了Server进程的用户空间地址，即Binder驱动实现了跨进程通信)</i> 3. Binder驱动 通知Server 进程执行 解包
	步骤3： Server进程 根据Client进程要求 调用目标方法	1. 收到Binder驱动通知后，Server 进程从线程池中取出 线程，进行数据解包 & 调用目标方法 2. 将最终执行结果写入到自己的共享内存中
	步骤4： Server进程 将目标方法的结果 返回给Client进程	1. 将最终执行结果写入存在映射的用户空间的内存区域中 <i>(由于 内核缓存区 & 接收进程的用户空间地址 存在映射关系 (同时映射Binder创建的接收缓存区中)，故相当于也发送到了内核缓存区中)</i> 2. Binder驱动通知Client进程获得返回结果 (此时Client进程之前被挂起的线程被重新唤醒) 3. Client进程 通过 系统调用copy_to_user () 从内核缓存区接收Server进程返回的数据
3. 使用服务	<p>示意图</p> <p>The diagram shows two processes: a '发送进程' (left) and a '接收进程' (right). Both have '用户空间' (User Space) and '内核空间' (Kernel Space). A '内核缓存区' (Kernel Cache) and a 'Binder创建的接收缓存区' (Binder-created receiver cache) are shown in the kernel space. In the sending process, a box labeled '需发送的数据' (Data to be sent) is copied from user space to the kernel cache using 'copy_from_user()' (step 1). This creates a '存在映射关系' (Mapping relationship) between the user space data and the kernel cache. In the receiving process, the data is copied from the kernel cache back to user space using 'copy_to_user()' (step 6). A pink dashed arrow labeled '3. 因存在映射关系，故实现跨进程通信' (As there is a mapping relationship, cross-process communication is implemented) indicates the flow from the kernel cache to user space. Another pink dashed arrow labeled '5. 当于写入到内核缓存区中' (Equivalent to writing into the kernel cache) points from the kernel cache back to user space.</p>	
优点	<ul style="list-style-type: none"> 传输效率高：每次单向通信数据拷贝次数少（1次）、用户空间 & 内核空间可直接通过共享对象直接交互 为接收进程 分配了不确定大小的接收缓存区 	

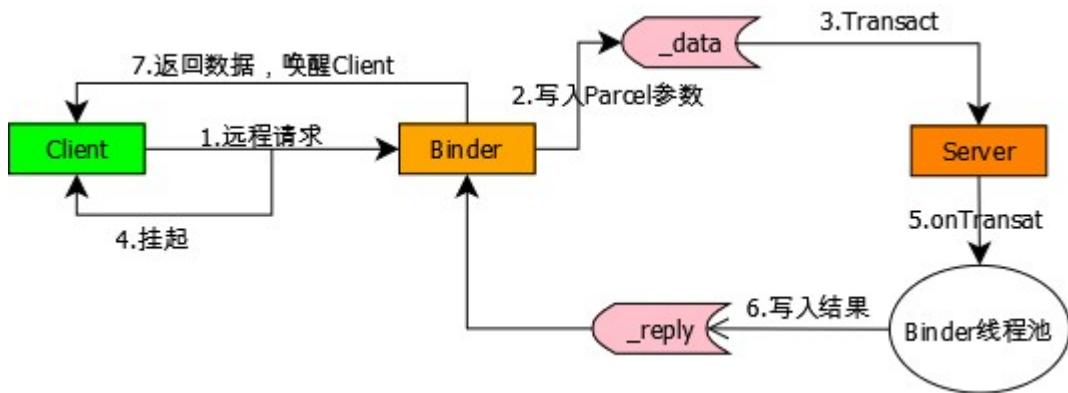
具体代码层面：

1、服务端中的Service给客户端提供Binder对象

2、客户端通过AIDL接口中的asInterface()将这个Binder对象转换为代理Proxy并通过它发起RPC请求

3、client进程的请求数据data通过代理binder对象的transact方法，发送到内核空间，当前线程被挂起

- 4、server进程收到binder驱动通知，onTransact(在线程池中进行数据反序列化&调用目标方法)处理客户端请求，并将结果写入reply
- 5、Binder驱动将server进程的目标方法执行结果，拷贝到client进程的内核空间
- 6、Binder驱动通知client进程，之前挂起的线程被唤醒，并收到返回结果



14.8.Binder框架中ServiceManager的作用

ServiceManager使得客户端可以获取服务端binder实例对象的引用

14.9.什么是AIDL

AIDL是android提供的接口定义语言，简化Binder的使用，轻松地实现IPC进程间通信机制。AIDL会生成一个服务端对象的代理类，通过它客户端可以实现间接调用服务端对象的方法。

14.10.AIDL使用的步骤

书写 AIDL

创建要操作的实体类，实现 Parcelable 接口，以便序列化/反序列化

新建 aidl 文件夹，在其中创建接口 aidl 文件以及实体类的映射 aidl 文件

Make project，生成 Binder 的 Java 文件

编写服务端

创建 Service，在Service中创建生成的Stub实例，实现接口定义的方法

在 onBind() 中返回Binder实例

编写客户端

实现 ServiceConnection 接口，在其中通过asInterface拿到 AIDL 类

bindService()

调用 AIDL 类中定义好的操作请求

14.11.AIDL支持哪些数据类型

Java八种基本数据类型(int、char、boolean、double、float、byte、long、string)但不支持short
String、CharSequence

List和Map，List接收方必须是ArrayList，Map接收方必须是HashMap

实现Parcelable的类

14.12.AIDL的关键类，方法和工作流程

Client和Server都使用同一个AIDL文件，在AIDL编译后会生成java文件，其中有Stub服务实体和Proxy服务代理两个类

AIDL接口：编译完生成的接口继承IInterface。

Stub类：服务实体，Binder的实现类，服务端一般会实例化一个Binder对象，在服务端onBind中绑定，客户端asInterface获取到Stub。

这个类在编译aidl文件后自动生成，它继承自Binder，表示它是一个Binder本地对象；它是一个抽象类，实现了IInterface接口，表明它的子类需要实现Server将要提供的具体能力（即aidl文件中声明的方法）。

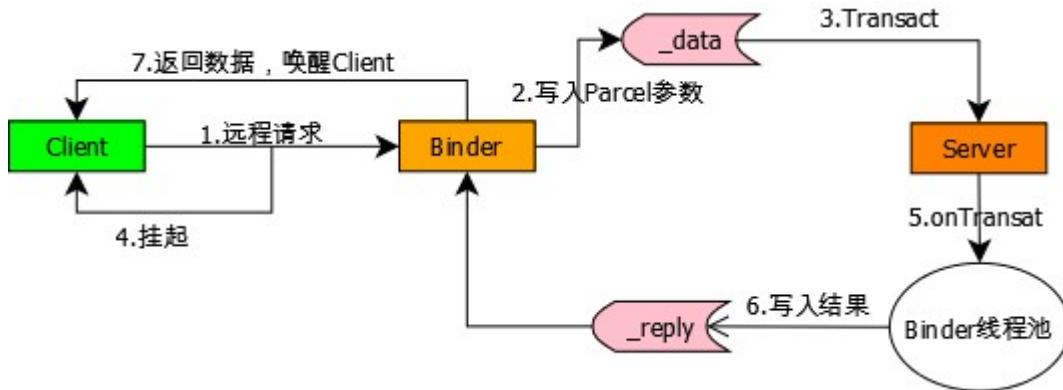
Stub.Proxy类：服务的代理，客户端asInterface获取到Stub.Proxy。

它实现了IInterface接口，说明它是Binder通信过程的一部分；它实现了aidl中声明的方法，但最终还是交由其中的mRemote成员来处理，说明它是一个代理对象，mRemote成员实际上就是BinderProxy。

asInterface()：客户端在ServiceConnection通过Person.Stub.asInterface(IBinder)，会根据是同一进行通信，还是不同进程通信，返回Stub()实体，或者Stub.Proxy()代理对象

transact()：运行在客户端，当客户端发起远程请求时，内部会把信息包装好，通过transact()向服务端发送。并将当前线程挂起，Binder驱动完成一系列的操作唤醒Server进程，调用Server进程本地对象的onTransact()来调用相关函数。到远程请求返回，当前线程继续执行。

onTransact()：运行在服务端的Binder线程池中，当客户端发起跨进程请求时，onTransact()根据Client传来的code调用相关函数。调用完成后把数据写入Parcel，通过reply发送给Client。驱动唤醒Client进程里刚刚挂起的线程并将结果返回。



14.13.如何优化多模块都使用AIDL的情况

每个业务模块创建自己的AIDL接口并创建Stub的实现类，向服务端提供自己的唯一标识和实现类。

服务端只需要一个Service，创建Binder连接池接口，根据业务模块的特征来返回相应的Binder对象。

客户端使用时通过Binder连接池，即将每个业务模块的Binder请求统一转发到一个远程Service中去执行，从而避免重复创建Service。

https://blog.csdn.net/it_yangkun/article/details/79888900

14.14.使用 Binder 传输数据的最大限制是多少，被占满后会导致什么问题

因为Binder本身就是为了进程间频繁而灵活的通信所设计的，并不是为了拷贝大数据而使用的。比如在Activity之间传输BitMap的时候，如果Bitmap过大，就会引起问题，比如崩溃等，这其实就跟Binder传输数据大小的限制有关系mmap函数会为Binder数据传递映射一块连续的虚拟地址，这块虚拟内存空间其实是有大小限制。

普通的由Zygote孵化而来的用户进程，所映射的Binder内存大小是不到1M的，准确说是 110241024 - (4096 *2)

```
#define BINDER_VM_SIZE ((1*1024*1024) - (4096 *2))
```

特殊的进程ServiceManager进程，它为自己申请的Binder内核空间是128K，这个同ServiceManager的用途是分不开的，ServiceManager主要面向SystemService，只是简单的提供一些addService, getService的功能，不涉及多大的数据传输，因此不需要申请多大的内存：

```
bs = binder_open(128*1024);
```

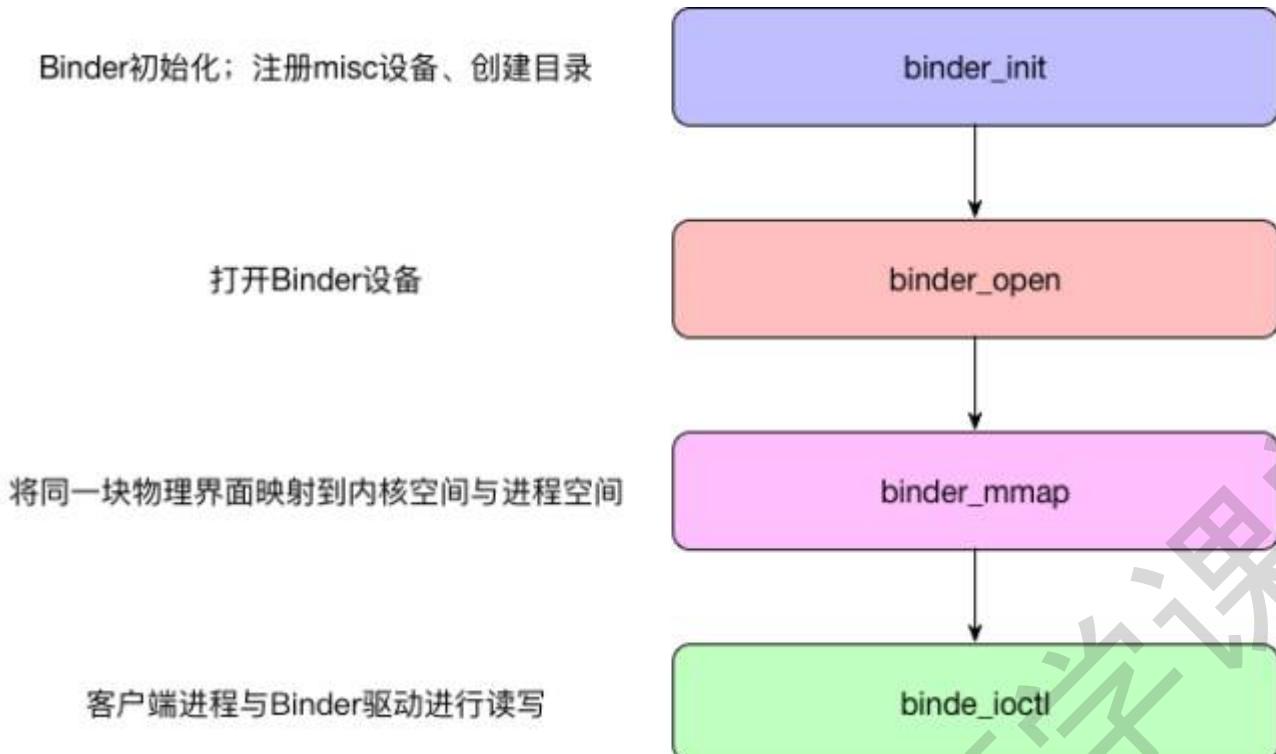
当服务端的内存缓冲区被Binder进程占用满后，Binder驱动不会再处理binder调用并在c++层抛出DeadObjectException到binder客户端

14.15.Binder 驱动加载过程中有哪些重要的步骤

从 Java 层来看就像访问本地接口一样，客户端基于 BinderProxy 服务端基于 IBinder 对象。

在Native层有一套完整的binder通信的C/S架构，Bpinder作为客户端，BBinder作为服务端。基于naive层的Binder框架，Java也有一套镜像功能的binder C/S架构，通过JNI技术，与native层的binder对应，Java层的binder功能最终都是交给native的binder来完成。

从内核看跨进程通信的原理最终是要基于内核的，所以最会会涉及到 binder_open、binder_mmap 和 binder_ioctl 这三种系统调用。



14.16.系统服务与bindService启动的服务的区别

服务可分为系统服务与普通服务，系统服务一般是在系统启动的时候，由SystemServer进程创建并注册到ServiceManager中例如AMS, WMS, PMS。而普通服务一般是通过ActivityManagerService启动的服务，或者说通过四大组件中的Service组件启动的服务。不同主要从以下几个方面：

服务的启动方式 系统服务这些服务本身其实实现了Binder接口，作为Binder实体注册到ServiceManager中，被ServiceManager管理。这些系统服务是位于SystemServer进程中

普通服务一般是通过Activity的startService或者其他context的startService启动的，这里的Service组件只是个封装，主要的是里面Binder服务实体类，这个启动过程不是ServiceManager管理的，而是通过ActivityManagerService进行管理的，同Activity管理类似

服务的注册与管理

系统服务一般都是通过ServiceManager的addService进行注册的，这些服务一般都是需要拥有特定的权限才能注册到ServiceManager，而bindService启动的服务可以算是注册到ActivityManagerService，只不过ActivityManagerService管理服务的方式同ServiceManager不一样，而是采用了Activity的管理模型

服务的请求使用方式

使用系统服务一般都是通过ServiceManager的getService得到服务的句柄，这个过程其实就是去ServiceManager中查询注册系统服务。而bindService启动的服务，主要是去ActivityManagerService中去查找相应的Service组件，最终会将Service内部Binder的句柄传给Client

14.17.Activity的bindService流程

- 1、Activity调用bindService：通过Binder通知ActivityManagerService，要启动哪个Service
- 2、ActivityManagerService创建ServiceRecord，并利用ApplicationThreadProxy回调，通知APP新建并启动Service启动起来
- 3、ActivityManagerService把Service启动起来后，继续通过ApplicationThreadProxy，通知APP，bindService，其实就是让Service返回一个Binder对象给ActivityManagerService，以便AMS传递给Client
- 4、ActivityManagerService把从Service处得到这个Binder对象传给Activity，这里是通过IServiceConnection binder实现。
- 5、Activity被唤醒后通过Binder Stub的asInterface函数将Binder转换为代理Proxy，完成业务代理的转换，之后就能利用Proxy进行通信了。

14.18.不通过AIDL，手动编码来实现Binder的通信

15.内存泄漏&内存溢出

15.1.什么是OOM & 什么是内存泄漏以及原因

内存泄漏

什么是

没有用的对象资源任与GC-Root保持可达路径，导致系统无法进行回收。

原因

- 1 非静态内部类默示持有外部类的引用，如非静态handler持有activity的引用
- 2.接收器、监听器注册没取消造成的内存泄漏，如广播，eventsbus

3. Activity 的 Context 造成的泄漏，可以使用 ApplicationContext
4. 单例中的 static 成员间接或直接持有了 activity 的引用
5. 资源对象没关闭造成的内存泄漏（如： Cursor、 File 等）
6. 全局集合类强引用没清理造成的内存泄漏（特别是 static 修饰的集合）

内存溢出

根据 java 的内存模型会出现内存溢出的内存有堆内存、 方法区内存、 虚拟机栈内存、 native 方法区内存， 一般说的 OOM 基本都是针对堆内存；

1、 对于堆内存溢出主的根本原因有两种

- (1) app 进程内存达到上限
- (2) 手机可用内存不足，这种情况并不是我们 app 消耗了很多内存，而是整个手机内存不足

2、 对于 app 内存达到上限只有两种情况

- (1) 申请内存的速度超出 gc 释放内存的速度. 往内存中加载超大文件，加载的文件或者图片过大造成或者循环创建大量对象
- (2) 内存出现泄漏， gc 无法回收泄漏的内存，导致可用内存越来越少

15.2. Thread 是如何造成内存泄露的，如何解决？

垃圾回收器不会回收 GC Roots 以及那些被它们间接引用的对象

非静态内部类会持有外部类的引用。 Thread 会长久地持有 Activity 的引用，使得系统无法回收 Activity 和它所关联的资源和视图。

使用静态内部类/匿名类，不要使用非静态内部类/匿名类。

该养成为 thread 设置退出逻辑条件的习惯。

15.3. Handler 导致的内存泄露的原因以及如何解决

原因：1. Java 中非静态内部类和匿名内部类都会隐式持有当前类的外部引用

2. 我们在 Activity 中使用非静态内部类初始化了一个 Handler，此 Handler 就会持有当前 Activity 的引用。

3. 我们想要一个对象被回收，那么前提它不被任何其它对象持有引用，所以当我们 Activity 页面关闭之后，存在引用关系：“未被处理 / 正处理的消息 -> Handler 实例 -> 外部类”，如果在 Handler 消息队列还有未处理的消息 / 正在处理消息时，导致 Activity 不会被回收，从而造成内存泄漏

解决方案：1. 将 Handler 的子类设置成静态内部类，使用 WeakReference 弱引用持有 Activity 实例

2. 当外部类结束生命周期时，清空 Handler 内消息队列

15.4. 如何加载 Bitmap 防止内存溢出

1. 对图片进行内存压缩；
2. 高分辨率的图片放入对应文件夹；
3. 内存复用

4.及时回收

15.5.MVP中如何处理Presenter层以防止内存泄漏的

首先 MVP 会出现内存泄漏是因为 Presenter 层持有 View 对象，一般我们会把 Activity 做为 View 传递到 Presenter，Presenter 持有 View对象，Activity 退出了但是没有回收出现内存泄漏。

解决办法： 1.Activity onDestroy() 方法中调用 Presenter 中的方法，把 View 置为 null

2.使用 Lifecycle

16.性能优化

<https://www.jianshu.com/u/7f26e9b13731> <https://github.com/liuyangbajin>

<https://www.jianshu.com/p/8bd39de7323f>

16.1.内存优化

<https://juejin.cn/post/6844903618642968590#heading-0>

首先你要确保你的应用里没有存在内存泄漏，然后再去做其他的内存优化。内存泄漏和图片优化相对来说比较容易定位问题，且优化后效果也非常明显，性价比非常高。

事实上很多优化都是这样，比如减包大小的优化，也是要先分析出主要大头祸首，比如可能你的包里包含了一张3M大小的无用图片，如果你没找到这种祸首，可能你做了大量的工作去想办法减少无用代码等，最终可能只有几百K的收益。

1、内存泄漏和图片优化

1.1 避免内存泄漏

1.1.1 注意代码中常见内存泄漏场景

Handler内存泄漏

在Activity中使用非静态内部类初始化了一个Handler,此Handler就会持有当前Activity的引用。Activity页面关闭之后，存在引用关系。非静态内部类和匿名内部类都会隐式持有当前类的外部引用。

单例造成的内存泄漏

单例持有 Context 对象，如果 Activity 中调用 getInstance 方法并传入 this 时，singleTon 就持有了此 Activity 的引用，当退出 Activity 时，Activity 就无法回收，造成内存泄漏

资源性对象未关闭，注册对象未注销

例如Bitmap等资源未关闭会造成内存泄漏，例如BroadcastReceiver、EventBus未注销造成的内存泄漏，我们应该在Activity销毁时及时注销。

MVP中的内存泄漏

Presenter 层持有 View 对象，一般我们会把 Activity 做为 View 传递到 Presenter，Presenter 持有 View对象，Activity 退出了但是没有回收出现内存泄漏。

ViewPager+fragment内存泄露

List里一直有Fragment的引用，Fragment无法回收造成内存泄漏 在重写的PagerAdapter的getItem()方法中，
return new yourFragment()解决此问题

在 MVP 的架构中，通常 Presenter 要同时持有 View 和 Model 的引用，如果在 Activity 退出的时候，Presenter 正在进行一个耗时操作，那么 Presenter 的生命周期会比 Activity 长，导致 Activity 无法回收，造成内存泄漏

避免在循环里创建对象，建立合适的缓存复用对象，避免在onDraw里创建对象

1.1.2 使用工具查找内存泄漏具体位置

LeakCanary

在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ReferenceQueue关联。这时再从ReferenceQueue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

Profiler

使用App后，Android Profiler中先触发GC，然后dump内存快照，之后点击按package分类，就可以迅速查看到你的App目前在内存中残留的class,点击class即可在右边查看到对应的实例以及引用对象。

1.2 图片优化

Bitmap 内存占用的计算

1.对图片进行内存压缩；

包括图片质量的压缩， RGB_565法，采样率压缩，Matrix比例压缩

2.高分辨率的图片放入对应文件夹；

不同dpi占用内存不一样,假设这张图片是ARGB_8888的，那这张图片占的内存就是 width * height * 4个字节或者 width * height * inTargetDensity /inDensity * 4

3.缓存

LruCache & DiskLruCache

4.及时回收

5.ARTHook非侵入式之图片检查

ARTHook 监控加载的图片是否过大的ImageView，可以在debug阶段发出警告，方便及早发现过大的图片。

1 使用Epic来进行Hook

2 DexposedBridge.hookAllConstructors(ImageView.class, new XC_MethodHook()

3 ImageHook中，图标宽高都大于view的2倍以上，则警告，当宽高度等于0时，说明ImageView还没有进行绘制，使用ViewTreeObserver进行大图检测的处理。

2、内存占用分析优化

2.1静态内存分析优化

确保打开每一个主要页面的主要功能，然后回到首页，进开发者选项去打开"不保留后台活动"。然后，将我们的app退到后台，GC，dump出内存快照。最后，我们就可以将对dump出的内存快照进行分析，看看有哪些地方是可以优化的，比如加载的图片、应用中全局的单例数据配置、静态内存与缓存、埋点数据、内存泄漏等等。

问题1：App首页的主图有两张(一张是保底图，一张是动态加载的图)，都比较大，而且动态加载的图回来后，保底图并没有及时被释放 优化：首先是对首页的主图进行颜色通道的改变以及压缩，可以大大降低这两张图所占的内存，然后在动态加载图回来后及时释放掉保底图 -5M

问题2：首页底部的轮播背景图占用内存1.6M，且在图片加载回来后，背景图一直没有置空 优化：首先一般来说对背景图的质量并没有很高的要求，所以这张背景图是可以被成倍压缩的，并且在图片加载回来后，背景图要及时的释放掉。同时首页的多张轮播图以及其他图片都可以进行颜色模式的改变以及质量压缩。 -1.6M -4M

3时间选择库

4：SharePreference在内存里占用了700K的内存 优化：由于SP中的东西是会一次性加载到内存里并且保存为静态的，直到App进程结束才会被销毁，所以SP中千万别放大的对象，别图一时方便把对象序列化成json后保存到SP里，优化点就是把已经保存在SP中的一些较大的json字符串或者对象迁移到文件或者数据库缓存。 -400K

2.2 动态内存分析优化

利用Android Profiler实时观察进入每个页面后的内存变化情况，对产生的内存较大波峰做分析，dump出2个页面的内存快照文件，然后利用MAT的对比功能，找出每个页面相对于上个页面内存里主要增加了哪些东西，做针对性优化。

问题1：在内存里发现两个极少概率出现的empty view，占用了接近2M的内存 优化：用ViewStub对empty view做了懒加载，对于这些没有马上用到的资源要做延迟加载，还有很多大概率不会出现的View更加要做懒加载。 -2M

问题2：发现详情页的轮播大图的Viewpager用的Adapter是FragmentPagerAdapter，导致了所有的page都会被保存，当图片页数多的时候，往后翻内存会不断上升。 优化：这种页数多的ViewPager使用FragmentStatePagerAdapter来替代，它只会保留前后page，在页数多的时候可以节省大量内存。

内存抖动：

没创建一个对象就会分配一个内存，可用内存就少用了一块，当程序占用的内存达到一定的临界值

就会出发GC 内存频繁分配和回收导致内存不稳定

瞬间产生大量的对象会严重占用Young Generation的内存区域，当达到阀值，剩余空间不够的时候，也会触发GC

解决

减少不合理的对象创建

onDraw、getView中对象的创建尽量进行复用。

避免在循环中不断创建局部变量。

实时监控

LeakCanary

hprof 文件裁剪

LeakCanary 用 shark 组件来实现裁剪 hprof 文件功能，在 shark-cli 工具中，我们可以通过添加 strip-hprof 选项来裁剪 hprof 文件，它的实现思路是：通过将所有基本类型数组替换为空数组（大小不变）。

实时监控

LeakCanary 主要是为测试环境开发，它会在 Activity 或者 Fragment 的 destroy 生命周期后，可以检测 Activity 和 Fragment 是否被回收，来判断它们是否存在泄露的情况。

Matrix-ResourceCanary

Matrix 主要也是在 hprof 文件裁剪 和 实时监控 这两方面做了一些优化。

hprof 文件裁剪

Matrix 的裁剪思路主要是将除了部分字符串和 Bitmap 以外实例对象中的 buffer 数组。之所以保留 Bitmap 是因为 Matrix 有个检测重复 Bitmap 的功能，会对 Bitmap 的 buffer 数组做一次 MD5 操作来判断是否重复。

监控原理

Matrix 是基于 LeakCanary 上进行二次开发，所以基本是一致的，主要增加了一些误报的优化，比如：

多次检测到相同的可疑对象，才认定为泄露对象，参数可配置。

增加一个哨兵对象，用于判断是否有 GC 操作，因为调用 Runtime.getRuntime().gc() 只是建议虚拟机进行 GC 操作，并不一定会进行。

避免重复检测相同的对象

Matrix 虽然是基于 LeakCanary，但额外增加了一些配置选项，可以用于生产环境，比如 dump 模式，支持手动触发 dump，自动 dump，和不进行 dump，可以根据不同的环境，使用不同的模式。

除此之前，还有检测时间间隔等等。

16.2.启动优化

<https://github.com/liuyangbajin/Performance----->

<https://my.oschina.net/u/660720/blog/3188893>

统计启动时间的方式

<https://www.jianshu.com/p/59a2ca7df681>

```
adb shell am start -S -R 10 -W  
com.ctg.and.mob.cuservice/com.ctg.and.mob.cuservice.mvp.ui.activity.MainActivity
```

TotalTime代表当前Activity启动时间，将多次TotalTime加起来求平均即可得到启动这个Activity的时间

缺点

应用的启动过程往往不只一个Activity，有可能是先进入一个启动页，然后再从启动页打开真正的首页。某些情况下还有可能中间经过更多的Activity，这个时候需要将多个Activity的时间加起来。

将多个Activity启动时间加起来并不完全等于用户感知的启动时间。例如在启动页可能是先等待某些初始化完成或者某些动画播放完毕后再进入首页。使用命令行统计的方式只是计算了Activity的启动以及初始化时间，并不能体现这种等待任务的时间。

解决

AMS会通过Zygote创建应用程序的进程后,执行Application构造方法 -> attachBaseContext() -> onCreate() -> Activity onCreate() ->-> onStart() -> onResume() -> 测量布局绘制显示在界面上->onWindowFocusChanged()绘制完毕 在Application的attachBaseContext()方法中开始计算冷启动计时，然后在真正首页Activity的onWindowFocusChanged()中停止冷启动计时。

代码

设置一个Util类专门做计时,Application的attachBaseContext()开始,首页Activity的 onWindowFocusChanged()结束

具体代码方法耗时分析工具

1. 手动记录

每个方法前记录System.currentTimeMillis(),之后System.currentTimeMillis()进行相减

2. AOP

Aspect统计 Application 中所有方法的耗时, AspectJ 其实就是一种 AOP 框架

2.1、连接点 (JoinPoint)

JPoint 是一个程序的关键执行点, 也是我们关注的重点。它就是指被拦截到的点 (如方法、字段、构造器等等)。

2.2、切入点 (PointCut)

对 JoinPoint 进行拦截的定义。PointCut 的目的就是提供一种方法使得开发者能够选择自己感兴趣的 JoinPoint。

2.3、通知 (Advice)

1)、Before: PointCut 之前执行。2)、After: PointCut 之后执行。3)、Around: PointCut 之前、之后分别执行。

3. Android Profiler 得到启动过程的 CPU 过程, 看到线程的具体代码耗时

<https://www.jianshu.com/p/e0d2b6347414>

Run -> Edit Configurations-> Sample Java Methods, 可以定位到 Java 代码大致定位到启动 CPU 耗时的原因

启动优化方案

<https://juejin.cn/post/6844903919580086280>

<https://www.jianshu.com/p/bef74a4b6d5e>

<https://juejin.cn/post/6844903459951476744#heading-6>

https://blog.csdn.net/qian520ao/article/details/81908505?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommentFromBaidu-1.control&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommentFromBaidu-1.control

<https://zhuanlan.zhihu.com/p/158683369>

1. 启动闪屏主题设置

<https://www.jianshu.com/p/436b91175826>

白屏原因:

1. 从 zygote 进程 fork 出一个新进程 2. 创建 Application 类并初始化, 主要是执行 Application.onCreate() 方法; 3. 创建并初始化入口 Activity 类, 并在窗口上绘制 UI;

设置 android:windowBackground 属性

windowBackground 属性可以配置任意 drawable, 与你启动页一样的图片。做到视觉上的无缝过渡。

2. application 初始化内存, 异步改造。

子线程来分担主线程的任务, 并减少运行时间

2. 线程池和启动器

线程缺乏统一管理, 可能无限制新建线程, 相互之间竞争, 及可能占用过多系统资源导致死机或 OOM, 所以我们使用线程池的方式去执行异步。

使用FixedThreadPool：创建一个定长的线程池，可控制线程最大的并发数，超出的部分任务，会在队列中等待。

利用CPU设置线程池数量

```
// 获得当前CPU的核心数 private static final int CPU_COUNT = Runtime.getRuntime().availableProcessors();  
// 设置线程池的核心线程数2-4之间,但是取决于CPU核数 private static final int CORE_POOL_SIZE = Math.max(2,  
Math.min(CPU_COUNT - 1, 4));  
  
// 创建线程池 ExecutorService executorService = Executors.newFixedThreadPool(CORE_POOL_SIZE);
```

2.线程池

通过线程池处理初始化任务的方式存在三个问题。

代码不够优雅

假如我们有 100 个初始化任务，那像上面这样的代码就要写 100 遍，提交 100 次任务。

无法限制在 onCreate 中完成

有的第三方库的初始化任务需要在 Application 的 onCreate 方法中执行完成，虽然可以用 CountDownLatch 实现等待，但是还是有点繁琐。

无法实现存在依赖关系

有的初始化任务之间存在依赖关系，比如极光推送需要设备 ID，而 initDeviceId() 这个方法也是一个初始化任务。

2.2.启动器

第一步是我们要对代码进行任务化，任务化是一个简称，比如把启动逻辑抽象成一个任务。

第二步是根据所有任务的依赖关系排序生成一个有向无环图，这个图是自动生成的，也就是对所有任务进行排序。

比如我们有个任务 A 和任务 B，任务 B 执行前需要任务 A 执行完，这样才能拿到特定的数据，比如上面提到的 initDeviceId。假如任务 B 依赖于任务 A，这时候生成的有向无环图就是 ACB，A 和 C 可以提前执行，B 一定要排在 A 之后执行。

第三步是多线程根据排序后的优先级依次执行。

4.延迟执行任务

延迟启动器利用了 IdleHandler 实现主线程空闲时才执行任务，IdleHandler 是 Android 提供的一个类，IdleHandler 会在当前消息队列空闲时才执行任务，这样就不会影响用户的操作了。假如现在 MessageQueue 中有两条消息，在这两条消息处理完成后，MessageQueue 会通知 IdleHandler 现在是空闲状态，然后 IdleHandler 就会开始处理它接收到的任务。

3.首页

首页部分接口合并为一，减少网络请求次数，降低频率；

首页布局优化

16.3.布局优化

布局加载流程

原理

<https://jsonchao.github.io/2020/01/13/%E6%B7%B1%E5%85%A5%E6%8E%A2%E7%B4%A2Android%E5%B8%83%E5%B1%80%E4%BC%98%E5%8C%96%EF%BC%88%E4%B8%8A%EF%BC%89/>

1、在setContentView方法中，会通过LayoutInflater的inflate方法去加载对应的布局到android.R.id.content中。

2、inflate方法中首先会调用Resources的getLayout方法去通过IO的方式去加载对应的Xml布局解析器到内存中。

调用链最终掉用的是AssetManager的openXmlAssetNative,一个Native方法，通过IO流的方式进行。

方法返回XmlResourceParser,XmlResourceParser继承XmlPullParser,根据提供的布局资源文件id可读取布局文件中view相关属性。

3、inflate, createViewFromTag来创建View的实例，在每次递归完成的时候将这个View添加到父布局中。

内部首先会判断mFactory2是否存在，存在就会使用mFactory2的onCreateView方法区创建视图，否则就会调用mFactory的onCreateView方法，接下来，如果此时的tag是一个Fragment，则会调用mPrivateFactory的onCreateView方法

createViewFromTag 将一些控件变成兼容性控件（例如将 TextView 变成 AppCompatTextView）以便于向下兼容新版本中的效果，createView，使用类加载器创建了对应的Class实例，根据Class实例获取到了对应的构造器实例，通过构造器实例constructor的newInstance方法创建了对应的View对象

<https://segmentfault.com/a/1190000019101554>

LayoutInflater.Factory的意义：

通过 LayoutInflater 创建 View 时候的一个回调，可以通过 LayoutInflater.Factory 来改造或定制创建 View 的过程。

AppCompatActivity 为什么 setFactory

向下兼容新版本中的效果。

Factory与Factory2的区别：

1、Factory2继承与Factory。 2、Factory2比Factory的onCreateView方法多一个parent的参数，即当前createView的父View。

优化

性能瓶颈在于LayoutInflater.inflate过程，主要包括如下两点：

1 xmlPullParser IO操作，布局越复杂，IO耗时越长。

2 createView 反射，View越多，反射调用次数越多，耗时越长

AsyncLayoutInflater

<https://www.jianshu.com/p/d61b513e6814>

1将构造好的 InflateRequest 请求放入到队列ArrayBlockingQueue中。

2.异步线程死循环轮训这个队列，当队列中有数据，取出一个 InflateRequest。

3.通过获取 InflateRequest.LayoutInflater 真正地加载 resid 对应的布局文件，最终得到一个 View 对象，并赋值给 InflateRequest.view。

4.通过 UIHandler 将 InflateRequest 回调到主线程中 (ps:这时加载完成的 View 就传到了主线程了)

5.UIHandler 处理消息，通过 InflateRequest#callback 将加载得到的 View 对象回调给调用层。

因为是异步加载，所以需要注意在布局加载过程中不能有依赖于主线程的操作，`AsyncLayoutInflater`仅仅只能通过侧面缓解的方式去缓解布局加载的卡顿。

X2C

X2C，它原理是采用APT (Annotation Processor Tool) + JavaPoet技术来完成编译期间视图xml布局生成java代码，这样布局依然是用xml来写，编译期X2C会将xml转化为动态加载视图的java代码。

工具

全局整体耗时监控：AspectJ做面向aop的非侵入性的布局整体耗时监控。

单个视图创建耗时监控：Factory2、Factory本质上他俩就是创建View的一个hook，可以通过这个回调来监控单个View创建耗时情况。

布局绘制流程

工具

Layout Inspector 和 GPU rendering

原理

绘制的过程 CPU准备数据，通过Driver层把数据交给GPU渲染，Display负责消费显示内容

1.CPU主要负责Measure、Layout、Record、Execute的数据计算工作

2.GPU负责Rasterization（栅格化(向量图形的格式表示的图像转换成位图用于显示器)）、渲染，渲染好后放到buffer(图像缓冲区)里存起来。

3.Display（屏幕或显示器）屏幕会以一定的帧率刷新，每次刷新的时候，就会从缓存区将图像数据读取显示出来，如果缓存区没有新的数据，就一直用旧的数据，这样屏幕看起来就没有变

1.在App进程中创建PhoneWindow后会创建ViewRoot。ViewRoot的创建会创建一个Surface壳子，请求WMS填充Surface，WMS copyFrom()一个NativeSurface。

2.响应客户端事件，创建Layer(FrameBuffer)与客户端的Surface建立连接。

3.copyFrom()的同时创建匿名共享内存SharedClient（每一个应用和SurfaceFlinger之间都会创建一个SharedClient）

4.当客户端addView()或者需要更新View时，App进程的SharedBufferClient写入数据到共享内存ShareClient中，SurfaceFlinger中的SharedBufferServer接收到通知会将FrameBuffer中的数据传输到屏幕上。

startActivity->ActivityThread.handleLaunchActivity->onCreate ->完成DecorView和Activity的创建->handleResumeActivity->onResume()->DecorView添加到WindowManager->ViewRootImpl.performTraversals()方法，测量(measure)，布局(layout)，绘制(draw)，从DecorView自上而下遍历整个View树。

<https://www.jianshu.com/p/779b5ad22316>

优化

5.1 优化布局层级及其复杂度 measure、layout、draw这三个过程都包含的自顶向下的tree遍历耗时，它是由视图层级太深会造成耗时，另外也要避免类似RelativeLayout嵌套造成的多次触发measure、layout的问题。最后onDraw在频繁刷新时可能多次被触发，因此onDraw不能做耗时操作，同时不能有内存抖动隐患等。

优化思路：减少View树层级

布局尽量宽而浅，避免窄而深 ConstraintLayout 实现几乎完全扁平化布局，同时具备RelativeLayout和LinearLayout特性，在构建复杂布局时性能更高。

不嵌套使用RelativeLayout

不在嵌套LinearLayout中使用weight

merge标签使用：减少一个根ViewGroup层级

ViewStub 延迟化加载标签，当布局整体被inflater，ViewStub也会被解析但是其内存占用非常低，它在使用前是作为占位符存在，对ViewStub的inflater操作只能进行一次，也就是只能被替换1次。

5.2 避免过度绘制 一个像素最好只被绘制一次。

优化思路：去掉多余的background，减少复杂shape的使用

避免层级叠加

自定义View使用clipRect屏蔽被遮盖View绘制

5.3 视图与数据绑定耗时

由于网络请求或者复杂数据处理逻辑耗时导致与视图绑定不及时。这里可以从优化数据处理的维度来解决

监控

布局加载监控

使用AspectJ做面向aop的非侵入性的监控。

针对Activity.setContentView监控简单示例：

```
@Aspect public class PerformanceAop { public static final String TAG = "aop"; @Around("execution(* android.app.Activity.setContentView(..))") public void getSetContentViewTime(ProceedingJoinPoint joinPoint) { Signature signature = joinPoint.getSignature(); String name = signature.toShortString(); long time = System.currentTimeMillis(); try { joinPoint.proceed(); } catch (Throwable throwable) { throwable.printStackTrace(); } Log.i(TAG, name + " cost " + (System.currentTimeMillis() - time)); } }
```

布局绘制监控

fpsviewer 利用Choreographer.FrameCallback来监控卡顿和Fps的计算，异步线程进行周期采样，当前的帧耗时超过自定义的阈值时，将帧进行分析保存，不影响正常流程的进行，待需要的时候进行展示，定位

Choreographer

1.只有当 App 注册监听下一个 Vsync 信号后才能接收到 Vsync 到来的回调。如果界面一直保持不变，那么 App 不会去接收每隔 16.6ms 一次的 Vsync 事件，但底层依旧会以这个频率来切换每一帧的画面。即当界面不变时屏幕也会固定每 16.6ms 刷新，但 CPU/GPU 不走绘制流程。

2.当 View 请求刷新时，这个任务并不会马上开始，而是需要等到下一个 Vsync 信号到来时才开始 measure/layout/draw 流程；measure/layout/draw 流程运行完后，界面也不会立刻刷新，而会等到下一个 VSync 信号到来时才进行缓存交换和显示。

3.造成丢帧主要原因：一是遍历绘制 View 树以及计算屏幕数据超过了16.6ms 第2帧的CPU/GPU计算 没能在VSync信号到来前完成，屏幕平白无故地多显示了一次第1帧。 VSync信号还没绘制完毕

Choreographer，意为 舞蹈编导、编舞者。在这里就是指 对CPU/GPU绘制的指导—— 收到VSync信号 才开始绘制，保证绘制拥有完整的16.6ms，避免绘制的随机性。 VSyn

1. 创建

mChoreographer，是在ViewRootImpl的构造方法内使用Choreographer.getInstance()创建

1.

```
//使用当前线程looper创建 mHandler
```

```
// 创建一个链表类型CallbackQueue的数组，大小为5，
```

```
创建了一个mHandler、VSync事件接收器mDisplayEventReceiver、任务链表数组mCallbackQueues
```

2. 当有绘制请求时通过 postCallback 方法请求下一次 Vsync 信号

首先取对应类型的CallbackQueue添加任务，action就是mTraversalRunnable

```
ViewRootImpl.scheduleTraversals中
```

```
//添加同步屏障，屏蔽同步消息，保证VSync到来立即执行绘制 mTraversalBarrier =  
mHandler.getLooper().getQueue().postSyncBarrier(); mChoreographer.postCallback(  
Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null); mTraversalRunnable 移除同步屏障，开始三  
大绘制流程
```

```
postCallback
```

Android 4.1 之后系统默认开启 VSYNC，在 Choreographer 的构造方法会创建一个 FrameDisplayEventReceiver，scheduleVsyncLocked 方法将会通过它申请 VSYNC 信号。在 DisplayEventReceiver 的构造方法会通过 JNI 创建一个 IDisplayEventConnection 的 VSYNC 的监听者。

VSYNC信号的接受回调是onVsync()，使用mHandler发送消息到MessageQueue中 执行本次的doFrame(),这个 message的Runnable就是队列中的所有任务。

3. 申请VSync信号接收到后是走 doFrame()方 执行任务是 CallbackRecord的 run 方法：

Choreographer 注册一个 FrameCallback 回调，那么系统在每一帧开始绘制的时候，会通过 FrameCallback#doFrame(...) 回调出来。我们一般画面的 fps 一般是 60fps，在这个回调中计算对应的 fps 即可。

如果绘制时间超过16.6ms，计算丢掉的帧数

<https://juejin.cn/post/6863756420380196877#heading-17>

<https://ljd1996.github.io/2020/09/07/Android-Choreographer%E5%8E%9F%E7%90%86/>

16.4. 卡顿优化

<https://www.jianshu.com/p/03dd61816051>

<https://juejin.cn/post/6844904066259091469#heading-31>

https://blog.yorek.xyz/android/paid/master/stuck_1/#systrace

<https://www.jianshu.com/p/75aa88d1b575>

<https://blog.csdn.net/JArchie520/article/details/106710663#1.1%E3%80%81%E5%8D%A1%E9%A1%BF%E9%97%AE%E9%A2%98%E4%BB%8B%E7%BB%8D>

<https://www.jianshu.com/p/03dd61816051>

卡顿原因

FPS (帧率)：每秒显示帧数 (Frames per Second)。表示图形处理器每秒钟能够更新的次数。高的帧率可以得到更流畅、更逼真的动画。一般来说12fps大概类似手动快速翻动书籍的帧率，这明显是可以感知到不够顺滑的。提升至60fps则可以明显提升交互感和逼真感。

开发app的性能目标就是保持60fps，这意味着每一帧你只有 $16\text{ms} \approx 1000/60$ 的时间来处理所有的任务。Android系统每隔16ms发出VSYNC信号，触发对UI进行渲染，如果每次渲染都成功，这样就能够达到流畅的画面所需要的60fps。

android中某个操作花费时间是24ms，系统在得到VSYNC信号的时候就无法进行正常渲染，这样就发生了丢帧现象。那么用户在32ms内看到的会是同一帧画面。就会感觉到界面不流畅了（卡了一下）。丢帧导致卡顿产生。

卡顿优化就是监控和分析由于哪些因素的影响导致绘制渲染任务没有在一个vsync的时间内完成。尤其关注连续丢帧点。

卡顿产生的原因是错综复杂的，它涉及到代码、内存、绘制、IO、CPU等等

卡顿监控

1.BlockCanary: 动态检测消息执行耗时。基于消息机制，向Looper中设置Printer，监控dispatcher到finish之间的操作，满足耗时阈值dump堆栈、设备信息，以通知形式弹出卡顿信息以供分析 非侵入式的性能监控组件，通知形式弹出卡顿信息 `AndroidPerformanceMonitor implementation 'com.github.markzhai:blockcanary-android:1.5.0'` <https://github.com/markzhai/AndroidPerformanceMonitor>

2.ANR

ANR-WatchDog

3.单点问题监控

界面打开页面从onCreate到onWindowFocusChanged耗时统计 Activity/Fragement 生命周期耗时统计 我们也需要去监控生命周期的一个耗时，如onCreate、onStart、onResume等 我们也需要去做生命周期间隔的耗时监控用AOP方式进行非侵入式打点通过这三个方面的监控纬度，我们就能够非常细粒度地去检测页面秒开各个方面的情况。自己代码：<https://github.com/eleme/lancet> 非one way的binder IPC调用耗时统计（hook BinderProxy.transact），这类统计可以采用AOP方式进行非侵入式打点。hook art: Epic hook

卡顿分析

卡顿工具 AndroidPerformanceMonitor ANR-WatchDog

原因 <https://blog.csdn.net/axi295309066/article/details/72675365>

解决 <https://juejin.cn/post/6844904066259091469>

16.5.网络优化

<http://www.odev.top/2020/06/29/Top%E5%9B%A2%E9%98%9F%E5%A4%A7%E7%89%9B%E5%B8%A6%E4%BD%A0%E7%8EA9%E8%BD%ACAndroid%E6%80%A7%E8%83%BD%E5%88%86%E6%9E%90%E4%B8%8E%E4%BC%98%E5%8C%96/#Java%E5%86%85%E5%AD%98%E5%9B%9E%E6%94%B6%E7%AE%97%E6%B3%95>

<https://juejin.cn/post/6861856444032466952#heading-0>

1. 流量维度

流量消耗过多

3.1.1 自定义事件监听器

请求次数

4.1 数据缓存 4.2 数据压缩 4.3 图片压缩

2.质量维度

模拟数据

弱网模拟

17.Window&WindowManager

https://blog.csdn.net/my_csdnboke/article/details/106685736

17.1.什么是Window

视图承载器，是一个视图的顶层窗口，包含了View并对View进行管理，是一个抽象类，具体的实现类为PhoneWindow,内部持有DecorView。通过WindowManager创建，并通过WindowManger将DecorView添加进来。

17.2.什么是WindowManager

WindowManager是一个接口，继承自只有添加、删除、更新三个方法的ViewManager接口。它的实现类为WindowManagerImpl，WindowManagerImpl通过WindowManagerGlobal代理实现addView，最后调用到ViewRootImpl的setView使ViewRoot和Decorview相关联。如果要对Window进行添加和删除就需要使用WindowManager，具体的工作则由WMS来处理，WindowManager和WMS通过Binder来进行跨进程通信。

3.什么是ViewRootImpl

ViewRoot是View和WindowManager的桥梁，View通过WindowManager来转接调用ViewRootImpl View的三大流程(测量 (measure) , 布局 (layout) , 绘制 (draw))均通过ViewRoot来完成。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。

17.4.什么是DecorView

DecorView是FrameLayout的子类，它可以被认为是Android视图树的根节点视图，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。
setContentView就是把需要添加的View的结构添加保存在DecorView中。

17.5.Activity, View, Window三者之间的关系

Activity并不负责视图控制，它只是控制生命周期和处理事件，Activity中持有的是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
View就是视图,在setContentView中将R.layout.activity_main添加到DecorView。

17.6.DecorView什么时候被WindowManager添加到Window中

即使Activity的布局已经成功添加到DecorView中，DecorView此时还没有添加到Window中
ActivityThread的handleResumeActivity方法中，首先会调用Activity的onResume方法，接着调用Activity的makeVisible()方法
makeVisible()中完成了DecorView的添加和显示两个过程

18.AMS

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

ActivityManagerService 主要负责系统中四大组件的启动、切换、调度及应用进程的管理和调度等工作，其职责与操作系统的进程管理和调度模块类似。

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。（系统启动流程）

如果打开一个App的话，需要AMS去通知zygote进程，所有的Activity的生命周期AMS来控制

18.2.ActivityThread是什么？ApplicationThread是什么？他们的区别

ActivityThread

在Android中它就代表了Android的主线程，它是创建完新进程之后，main函数被加载，然后执行一个loop的循环使当前线程进入消息循环，并且作为主线程。

ApplicationThread

ApplicationThread是ActivityThread的内部类，是一个Binder对象。在此处它是作为IApplicationThread对象的server端等待client端的请求然后进行处理，最大的client就是AMS。

18.3.Instrumentation是什么？和ActivityThread是什么关系？

AMS与ActivityThread之间诸如Activity的创建、暂停等的交互工作实际上是由Instrumentation具体操作的。每个Activity都持有一个Instrumentation对象的一个引用，整个进程中是只有一个Instrumentation。
mInstrumentation的初始化在ActivityThread::handleBindApplication函数。

可以用来独立地控制某个组件的生命周期。

Activity`的`startActivity`方法。`startActivity`会调用`mInstrumentation.execStartActivity();`

mInstrumentation掉用AMS，AMS通过socket通信告知Zygote进程fork子进程。

18.4.ActivityManagerService和zygote进程通信是如何实现的。

应用启动时，Launcher进程请求AMS。AMS发送创建应用进程请求，Zygote进程接受请求并fork应用进程
客户端发送请求

调用Process.start()方法新建进程

连接调用的是ZygoteState.connect()方法，ZygoteState是ZygoteProcess的内部类。ZygoteState里用的LocalSocket

```
public static ZygoteState connect(LocalSocketAddress address) throws IOException {
    DataInputStream zygoteInputStream = null;
    BufferedWriter zygotewriter = null;
    final LocalSocket zygoteSocket = new LocalSocket();
    try {
        zygoteSocket.connect(address);
        zygoteInputStream = new DataInputStream(zygoteSocket.getInputStream());
        zygotewriter = new BufferedWriter(new OutputStreamWriter(
            zygoteSocket.getOutputStream()), 256);
    } catch (IOException ex) {
        try {
            zygoteSocket.close();
        } catch (IOException ignore) {
        }
        throw ex;
    }
    return new ZygoteState(zygoteSocket, zygoteInputStream, zygotewriter,
        Arrays.asList(abiListString.split(",")));
}
```

Zygote 处理客户端请求

Zygote 服务端接收到参数之后调用 ZygoteConnection.processOneCommand() 处理参数，并 fork 进程

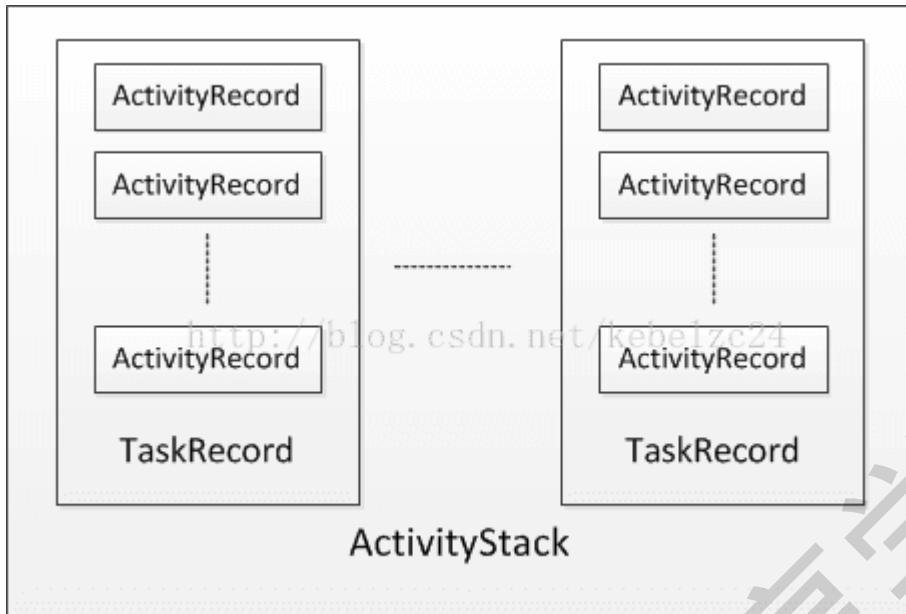
最后通过 findStaticMain() 找到 ActivityThread 类的 main() 方法并执行，子进程就启动了

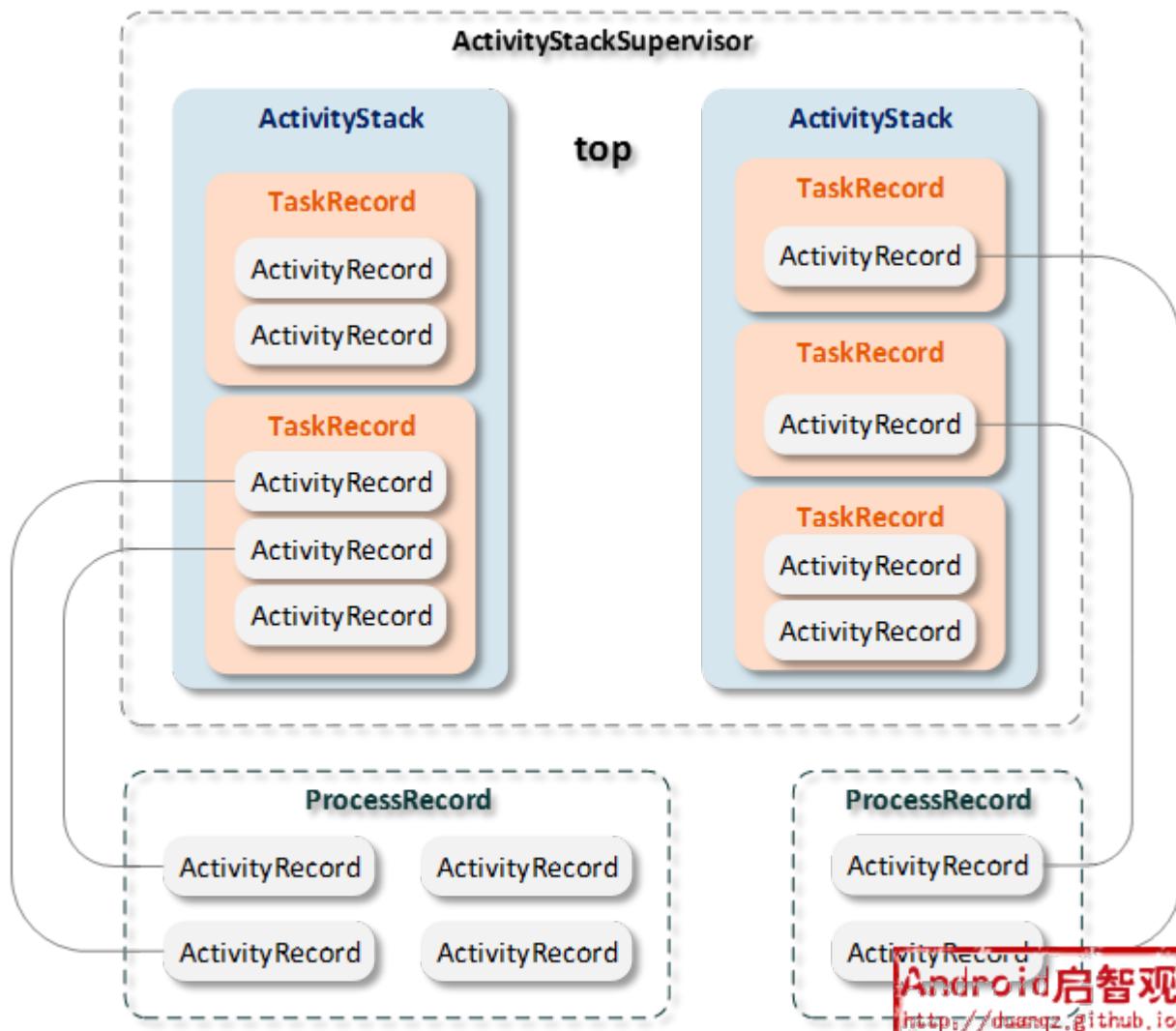
18.5. ActivityRecord、TaskRecord、ActivityStack, ActivityStackSupervisor, ProcessRecord

<https://duanqz.github.io/2016-02-01-Activity-Maintenance#activityrecord>

<https://www.jianshu.com/p/94816e52cd77>

<https://juejin.im/post/6856298463119409165#heading-10>





ActivityRecord

Activity管理的最小单位，它对应着一个用户界面

ActivityRecord是应用层Activity组件在AMS中的代表，每一个在应用中启动的Activity，在AMS中都有一个ActivityRecord实例来与之对应，这个ActivityRecord伴随着Activity的启动而创建，也伴随着Activity的终止而销毁。

TaskRecord

TaskRecord即任务栈，每一个TaskRecord都可能存在一个或多个ActivityRecord，栈顶的ActivityRecord表示当前可见的界面。

一个App是可能有多个TaskRecord存在的

一般情况下,启动App的第一个activity时，AMS为其创建一个TaskRecord任务栈

特殊情况,启动singleTask的Activity，而且为该Activity指定了和包名不同的taskAffinity，也会为该activity创建一个新的TaskRecord

ActivityStack

ActivityStack,ActivityStack是系统中用于管理TaskRecord的,内部维护了一个ArrayList。

ActivityStackSupervisor内部有两个不同的ActivityStack对象：mHomeStack、mFocusedStack，用来管理不同的任务。

我们启动的App对应的TaskRecord由非Launcher ActivityStack管理，它是在系统启动第一个app时创建的。

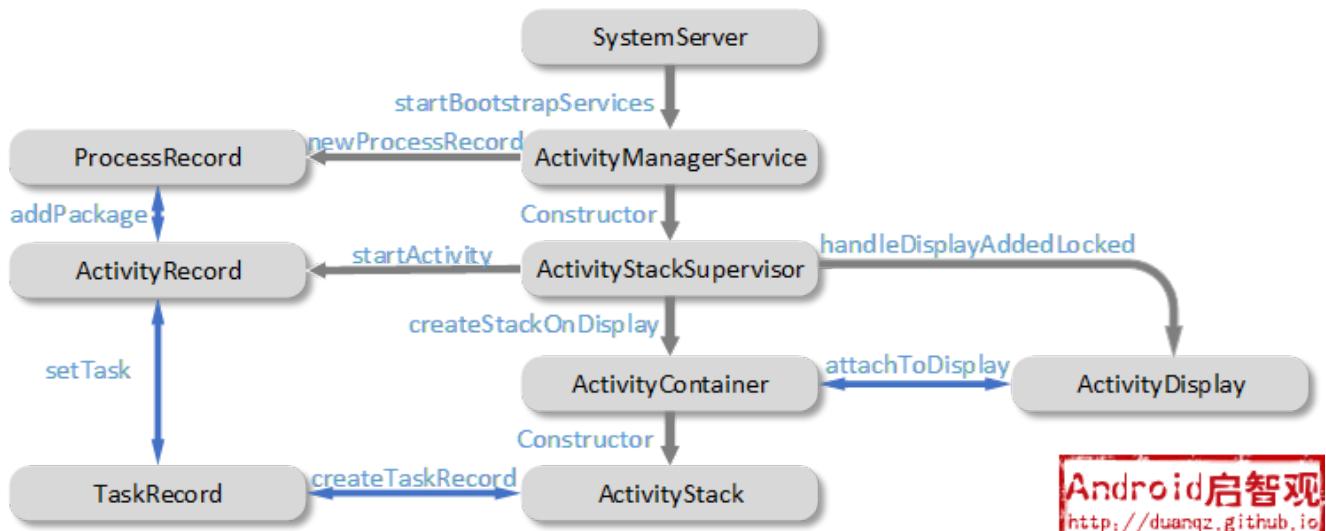
ActivityStackSupervisor

ActivityStackSupervisor管理着多个ActivityStack，但当前只会有一个获取焦点(Focused)的ActivityStack；
AMS对象只会存在一个，在初始化的时候，会创建一个唯一的ActivityStackSupervisor对象

ProcessRecord

ProcessRecord记录着属于一个进程的所有ActivityRecord，运行在不同TaskRecord中的ActivityRecord可能是属于同一个ProcessRecord。

关系



AMS运行在SystemServer进程中。SystemServer进程启动时，会通过SystemServer.startBootstrapServices()来创建一个AMS的对象；

AMS通过ActivityStackSupervisor来管理Activity。AMS对象只会存在一个，在初始化的时候，会创建一个唯一的ActivityStackSupervisor对象；

ActivityStackSupervisor中维护了显示设备的信息。当有新的显示设备添加时，会创建一个新的ActivityDisplay对象；

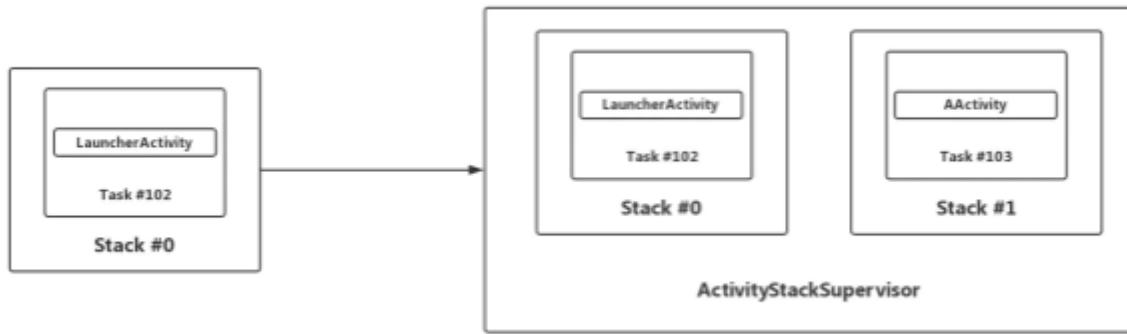
ActivityStack与显示设备的绑定。ActivityStack的创建时在Launcher启动时候进行的，AMS还未有非Launcher的ActivityStack。后面的App启动时就会创建Launcher的ActivityStack，

通过ActivityStackSupervisor来创建ActivityRecord

在ActivityStack上创建TaskRecord

每一个ActivityRecord都需要找到自己的宿主TaskRecord

从桌面启动

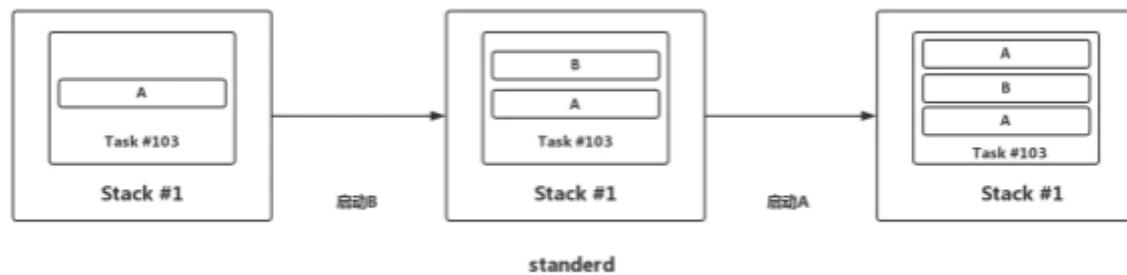


从桌面点击图标启动一个Activity，会启动ActivityStackSupervisor中的mFocusedStack，mFocusedStack负责管理的是非Launcher相关的任务。同时也会创建一个新的ActivityRecord和TaskRecord，ActivityRecord放到TaskRecord中，TaskRecord则放进mFocusedStack中。

四种启动模式

standerd

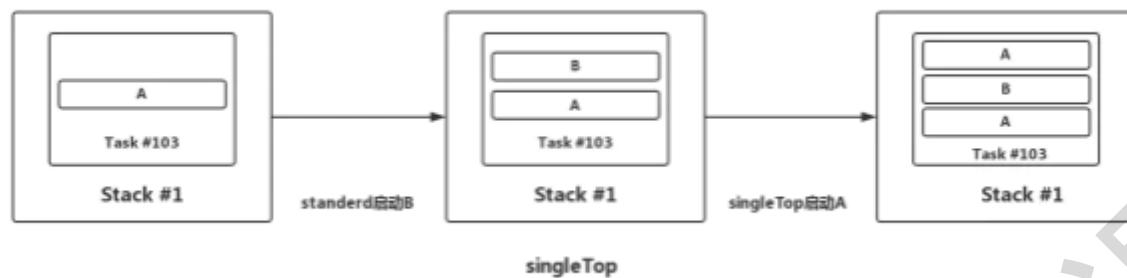
默认模式，每次启动Activity都会创建一个新的Activity实例。

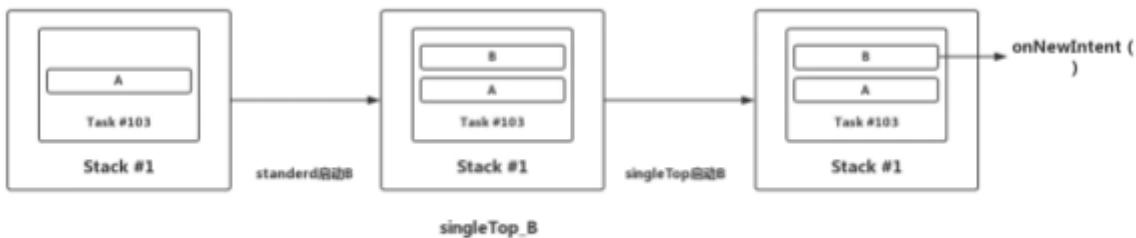


singleTop

如果要启动的Activity已经在栈顶，则不会重新创建Activity，只会调用该Activity的onNewIntent()方法。

如果要启动的Activity不在栈顶，则会重新创建该Activity的实例。

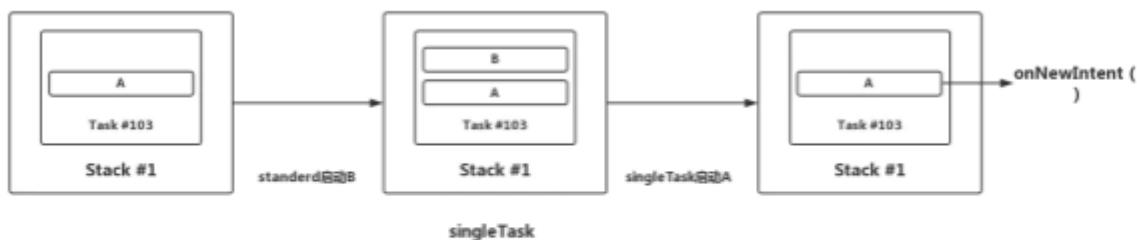




singleTop_B.png

singleTask

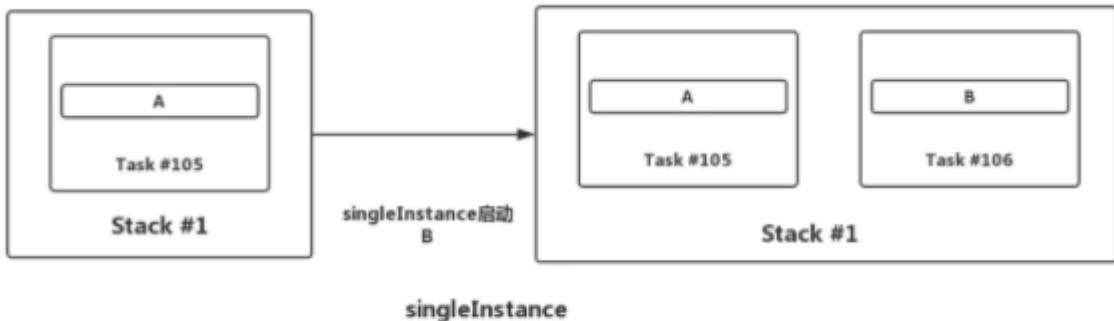
如果要启动的Activity已经存在于它想要归属的栈中，那么不会创建该Activity实例，将栈中位于该Activity上的所有的Activity出栈，同时该Activity的onNewIntent()方法会被调用。



singleTask.png

singleInstance

要创建在一个新栈，然后创建该Activity实例并压入新栈中，新栈中只会存在这一个Activity实例。



singleInstance.png

18.6.ActivityManager、ActivityManagerService、ActivityManagerNative、ActivityManagerProxy的关系

<https://www.cnblogs.com/mingfeng002/p/10650364.html>

Activity 的 `startActivity` 方法。 `startActivity` 会调用 `mInstrumentation.execStartActivity()`；
`execStartActivity` 通过 `ActivityManager` 的 `getService`。

代码层面：

ActivityManager.getRunningServices里通过ActivityManagerNative.getDefault得到此代理对象ActivityManagerProxy，ActivityManagerProxy代理类是ActivityManagerNative的内部类。ActivityManagerNative是个抽象类，真正发挥作用的是它的子类ActivityManagerService。

介绍：

ActivityManager

ActivityManager官方介绍：是与系统所有正在运行着的Acitvity进行交互，对系统所有运行中的Activity相关信息（Task, Memory, Service, App）进行管理和维护。

ActivityManagerNative、ActivityManagerProxy

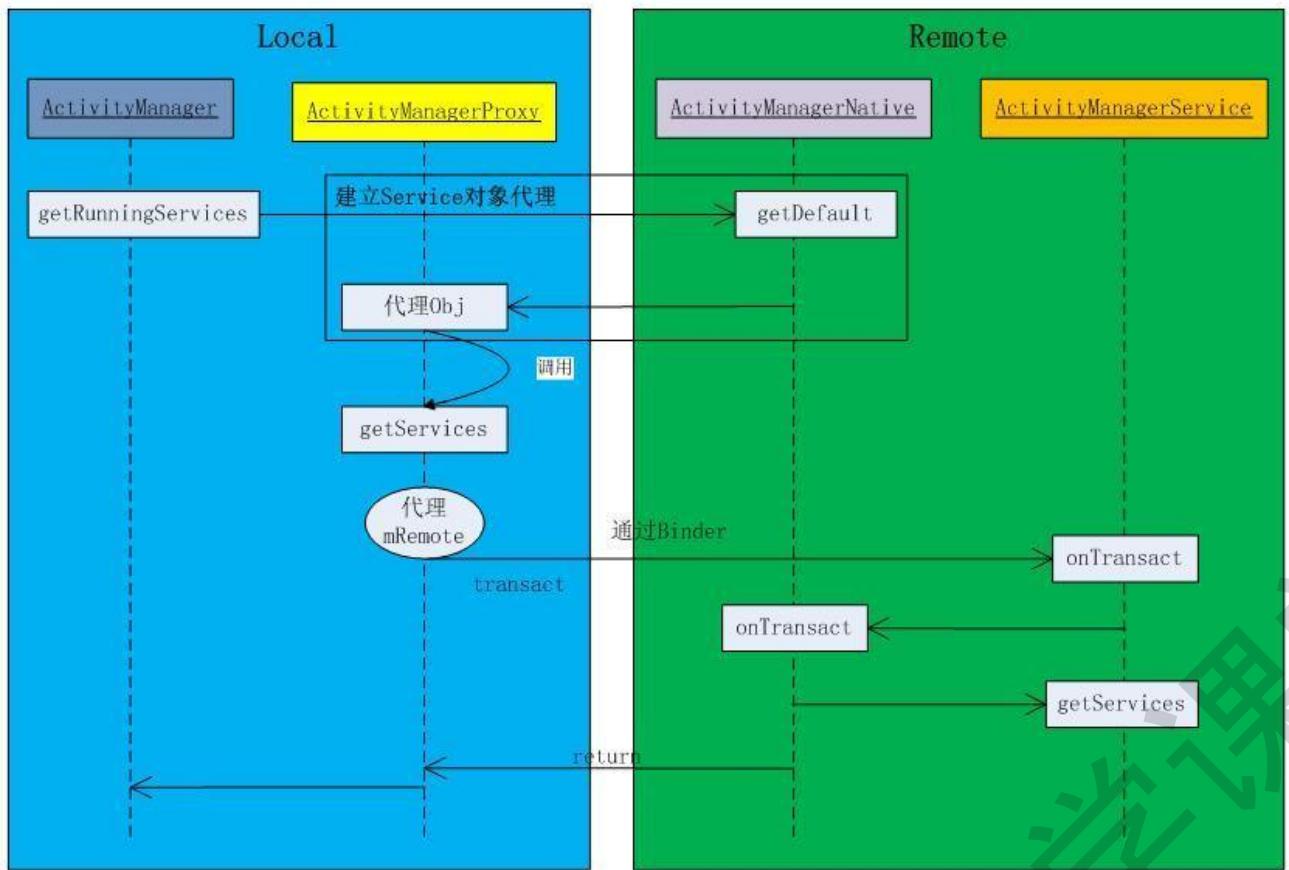
IActivityManager继承了Interface接口。而ActivityManagerNative和ActivityManagerPorxy实现了这个IActivityManager接口

ActivityManagerProxy代理类是ActivityManagerNative的内部类；

ActivityManagerNative是个抽象类，真正发挥作用的是它的子类ActivityManagerService

ActivityManager持有的是这个ActivityManagerPorxy代理对象，这样，只需要操作这个代理对象就能操作其业务实现的方法。那么真正实现其它业务的则是ActivityManagerService。

ActivityManagerNative这个类，他继承了Binder而Binder实现了IBinder接口。其子类则是ActivityManagerService。



18.7.手写实现简化版AMS

AMS与Binder相关，其中要明白下面几个类的职责：

IBinder：跨进程通信的Base接口，它声明了跨进程通信需要实现的一系列抽象方法，实现了这个接口就说明可以进行跨进程通信，所有的Binder实体都必须实现IBinder接口。

IInterface：这也是一个Base接口，用来表示Server提供了哪些能力，是Client和Server通信的协议，Client和Server都要实现此接口。

Binder：IBinder的子类，Java层提供服务的Server进程持有一个Binder对象从而完成跨进程间通信。

BinderProxy：在Binder.java这个文件中还定义了一个BinderProxy类，这个类表示Binder代理对象它同样实现了IBinder接口。Client中拿到的实际上是这个代理对象。

1、首先定义IActivityManager接口(继承IInterface)：

```
public interface IActivityManager extends IInterface {  
    //binder描述符  
    String DESCRIPTOR = "android.app.IActivityManager";  
    //方法编号  
    int TRANSACTION_startActivity = IBinder.FIRST_CALL_TRANSACTION + 0;  
    //声明一个启动activity的方法，为了简化，这里只传入intent参数  
    int startActivity(Intent intent) throws RemoteException;  
}
```

2、实现ActivityManagerService侧的本地Binder对象基类：

```
public abstract class ActivityManagerNative extends Binder implements IActivityManager {  
  
    public static IActivityManager asInterface(IBinder obj) {  
        if (obj == null) {  
            return null;  
        }  
        IActivityManager in = (IActivityManager)  
obj.queryLocalInterface(IActivityManager.DESCRIPTOR);  
        if (in != null) {  
            return in;  
        }  
        //代理对象，见下面的代码  
        return new ActivityManagerProxy(obj);  
    }  
  
    @Override  
    public IBinder asBinder() {  
        return this;  
    }  
  
    @Override  
    protected boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws  
RemoteException {  
        switch (code) {  
            // 获取binder描述符  
            case INTERFACE_TRANSACTION:  
                reply.writeString(IActivityManager.DESCRIPTOR);  
                return true;  
        }  
    }  
}
```

```
// 启动activity，从data中反序列化出intent参数后，直接调用子类startActivity方法启动
activity.

    case IActivityManager.TRANSACTION_startActivity:
        data.enforceInterface(IActivityManager.DESCRIPTOR);
        Intent intent = Intent.CREATOR.createFromParcel(data);
        int result = this.startActivity(intent);
        reply.writeNoException();
        reply.writeInt(result);
        return true;
    }

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

}
```

3、实现Client侧的代理对象：

```
public class ActivityManagerProxy implements IActivityManager {
    private IBinder mRemote;

    @Override
    public int startActivity(Intent intent) throws RemoteException {
        Parcel data = Parcel.obtain();
        Parcel reply = Parcel.obtain();
        int result;
        try {
            // 将intent参数序列化，写入data中
            intent.writeToParcel(data, 0);
            // 调用BinderProxy对象的transact方法，交由Binder驱动处理。
            mRemote.transact(IActivityManager.TRANSACTION_startActivity, data, reply, 0);
            reply.readException();
            // 等待server执行结束后，读取执行结果
            result = reply.readInt();
        } finally {
            data.recycle();
            reply.recycle();
        }
        return result;
    }
}
```

4、实现Binder本地对象（IActivityManager接口）：

```
public class ActivityManagerService extends ActivityManagerNative {
    @Override
    public int startActivity(Intent intent) throws RemoteException {
        // 启动activity
        return 0;
    }
}
```

19.系统启动

19.1.Android系统启动流程

Android系统架分为应用层, framework层, 系统运行库层 (Native), Linux内核层 启动按照一个流程: Loader->kernel->framework->Application来进行的

1.Bootloader引导

- 当电源按下时, 引导芯片代码从 ROM (4G)开始执行。Bootloader引导程序把操作系统映像文件拷贝到RAM中去, 然后跳转到它的入口处去执行, 启动Linux内核。
- Linux kernel 内核启动, 会做设置缓存, 加载驱动等一些列操作
- 当内核启动完成之后, 启动 init 进程, 作为第一个系统进程, init 进程从内核态转换成用户态。

2.init进程启动

- fork 出 ServerManager 子进程。ServerManager主要用于管理我们的系统服务, 他内部存在一个server服务列表, 这个列表中存储的就是那些已经注册的系统服务。
- 解析 init.rc 配置文件并启动 zygote 进程

3.Zygote进程启动

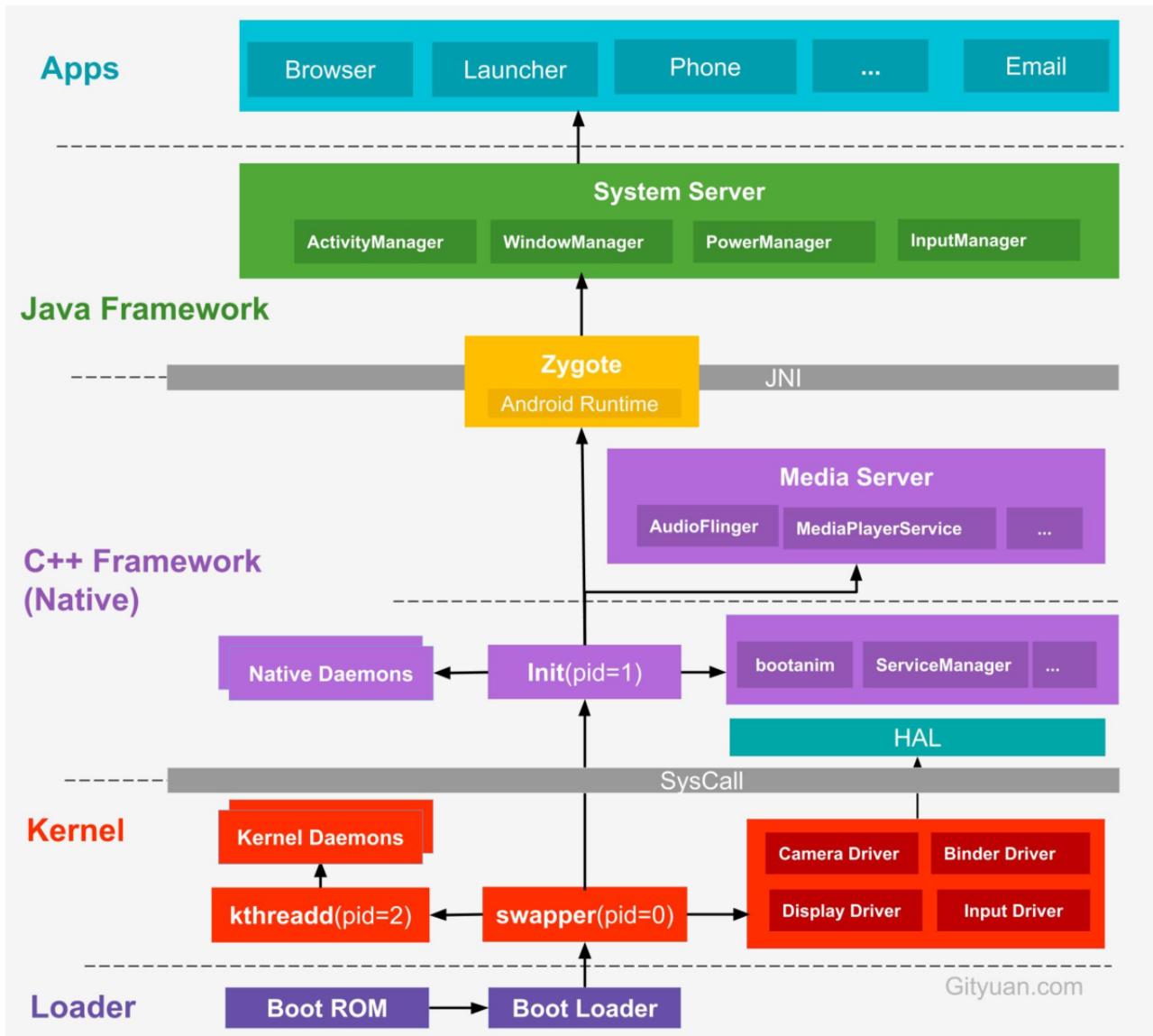
- 孵化其他应用程序进程, 所有的应用的进程都是由zygote进程fork出来的。通过创建服务端Socket, 等待AMS的请求来创建新的应用程序进程。
- 创建SystemServer进程, 在Zygote进程启动之后, 会通过ZygoteInit的main方法fork出SystemServer进程

4.SystemServer进程启动

- 创建SystemServiceManager, 它用来对系统服务进行创建、启动和生命周期管理。
- ServerManager.startService启动各种系统服务: WMS/PMS/AMS等, 调用ServerManager的addService方法, 将这些Service服务注册到ServerManager里面
- 启动桌面进程, 这样才能让用户见到手机的界面。

5.Launcher进程启动

- 开启系统Launcher程序来完成系统界面的加载与显示。



图片名称

19.2.SystemServer, ServiceManager, SystemServiceManager的关系

在SystemServer进程中创建SystemServiceManager, ServiceManager是系统服务管理者,SystemServiceManager启动一些继承自SystemService的服务，并将这些服务的Binder注册到ServiceManager中，对于其他的一些继承于IBinder的服务,通过ServiceMaanger的addService方法添加

SystemServer:

SystemServer是一个由zygote孵化出来的进程，名字为system_server。SystemServer叫做系统服务进程，大部分Android提供的一些系统服务都运行在该进程中,包括AMS, WMS, PMS, 这些系统的服务都是以一个线程的方式存在在SystemServer进程中。

SystemServiceManager:

管理一些系统的服务，在SystemServer中初始化。启动各种系统服务：WMS/PMS/AMS等,调用ServerManager的addService方，将这些Service服务注册到ServerManager里面

ServiceManager:

ServiceManager像是一个路由，Service把自己注册在ServiceManager中,客户端 通过ServiceManager查询服务

- 1、维护一个svclist列表来存储service信息。
- 2、向客户端提供Service的代理，也就是BinderProxy。
- 3、维护一个死循环，不断的查看是否有service的操作请求，如果有就读取相应的内核binder driver。

19.3. 孵化应用进程这种事为什么不交给SystemServer来做，而专门设计一个Zygote

- Zygote进程是所有Android进程的母体，包括system_server和各个App进程。zygote利用fork()方法生成新进程，对于新进程A复用Zygote进程本身的资源，再加上新进程A相关的资源，构成新的应用进程A。应用在启动的时候需要做很多准备工作，包括启动虚拟机，加载各类系统资源等等，这些都是非常耗时的，如果能在 zygote里就给这些必要的初始化工作做好，子进程在fork的时候就能直接共享，那么这样的话效率就会非常高
- SystemServer里跑了一堆系统服务，这些不能继承到应用进程

19.4. Zygote的IPC通信机制为什么使用socket而不采用binder

- Zygote是通过fork生成进程的
- 因为fork只能拷贝当前线程，不支持多线程的fork，fork的原理是copy-on-write机制，当父子进程任一方修改内存数据时（这是on-write时机），才发生缺页中断，从而分配新的物理内存（这是copy操作）。zygote进程中已经启动了虚拟机、进行资源和类的预加载以及各种初始化操作，App进程用时拷贝即可。Zygote fork出来的进程A只有一个线程，如果Zygote有多个线程，那么A会丢失其他线程。这时可能造成死锁。
- Binder通信需要使用Binder线程池，binder维护了一个16个线程的线程池，fork()出的App进程的binder通讯无法用

20.App启动&打包&安装

20.1. 应用启动流程

1.Launcher进程请求AMS

点击图标发生在 Launcher 应用的进程,实际上执行的是 Launcher 的 onclick 方法，在 onclick 里面会执行到 Activity 的 startActivity 方法。`startActivity`会调用 `mInstrumentation.execStartActivity()`;
`execStartActivity`通过 ActivityManager 的 `getService` 方法来得到 AMS 的代理对象(Launcher 进程作为客户端与服务端 AMS 不在同一个进程, ActivityManager.getService 返回的是 `IActivityManager.Stub` 的代理对象,此时如果要实现客户端与服务端进程间的通信，需要 AMS 继承 `IActivityManager.Stub` 类并实现相应的方法,这样 Launcher 进程作为客户端就拥有了服务端AMS的代理对象，然后就可以调用AMS的方法来实现具体功能了)

2. AMS发送创建应用进程请求，Zygote进程接受请求并fork应用进程

AMS 通过 socket 通信告知 zygote 进程 fork 子进程。

应用进程启动 `ActivityThread`,执行 `ActivityThread` 的 `main` 方法。

`main` 方法中创建 `ApplicationThread`，`Looper`，`Handler` 对象，并开启主线程消息循环 `Looper.loop()`。

3.App进程通过Binder向AMS(system_server)发起attachApplication请求,AMS绑定ApplicationThread

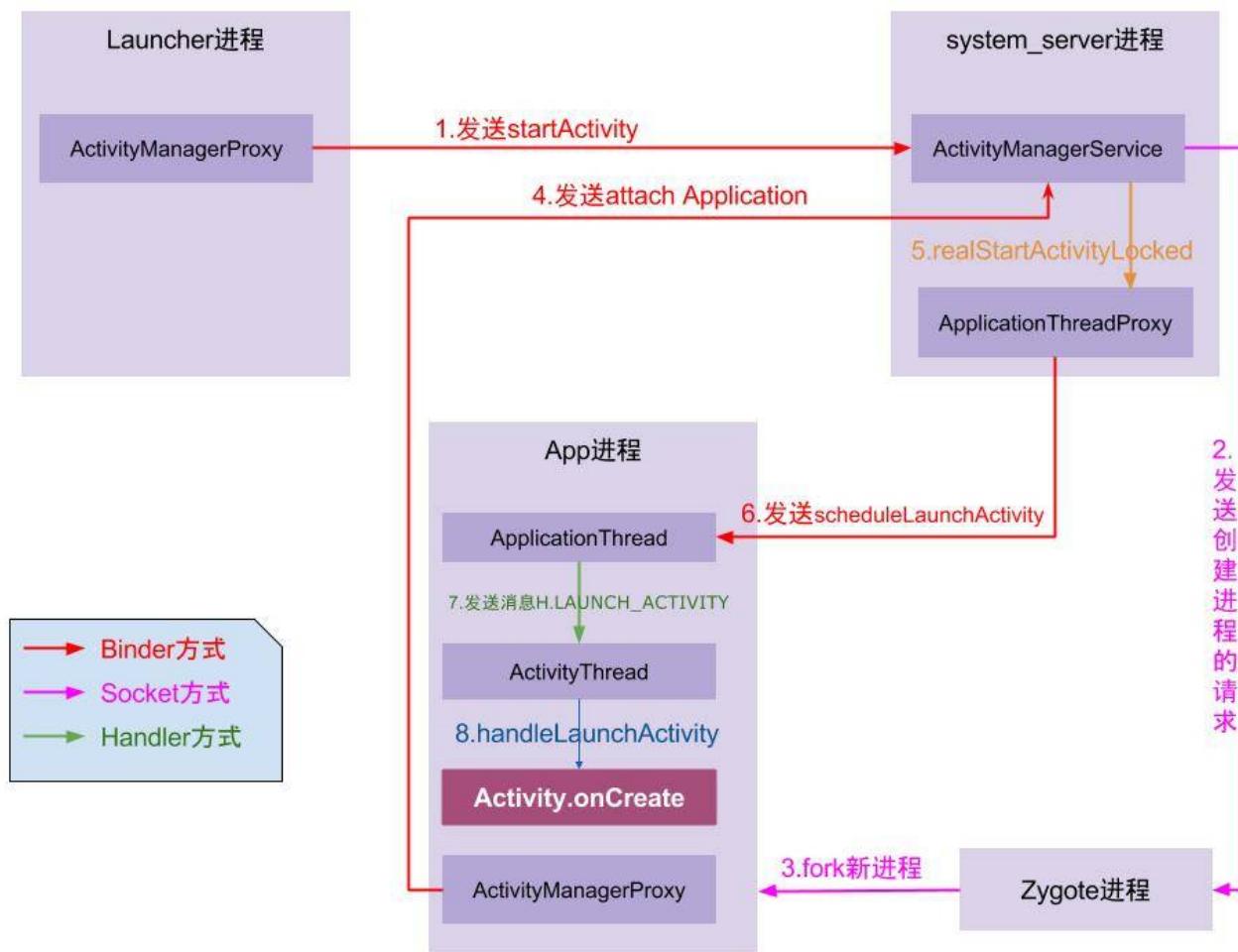
在 `ActivityThread` 的 `main` 中,通过 `ApplicationThread.attach(false, startSeq)`,将 AMS 绑定 `ApplicationThread` 对象,这样 AMS 就可以通过这个代理对象 来控制应用进程。

4.AMS发送启动Activity的请求

system_server 进程在收到请求后，进行一系列准备工作后，再通过 binder 向App进程发送 scheduleLaunchActivity 请求；AMS 将启动 Activity 的请求发送给 ActivityThread 的 Handler。

5.ActivityThread的Handler处理启动Activity的请求

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



20.2.apk组成和Android的打包流程?

名称	修改日期	类型	大小
META-INF	2016/10/6 14:04	文件夹	
res	2016/10/6 14:04	文件夹	
AndroidManifest.xml	2016/10/6 14:04	XML 文件	2 KB
classes.dex	2016/10/6 14:04	DEX 文件	2,548 KB
resources.arsc	2016/10/6 14:04	ARSC 文件	203 KB

resources.arsc 编译后的二进制资源文件。

classes.dex 是.dex文件。最终生成的Dalvik字节码

AndroidManifest.xml 程序的全局清单配置文件

res是uncompiled resources。存放资源文件的目录。

META-INF是签名文件夹。存放签名信息

MANIFEST.MF (清单文件)：其中每一个资源文件都有一个SHA-256-Digest签名，MANIFEST.MF文件的SHA256 (SHA1) 并base64编码的结果即为CERT.SF中的SHA256-Digest-Manifest值。

CERT.SF (待签名文件)：除了开头处定义的SHA256 (SHA1) -Digest-Manifest值，后面几项的值是对MANIFEST.MF文件中的每项再次SHA256并base64编码后的值。

CERT.RSA (签名结果文件)：其中包含了公钥、加密算法等信息。首先对前一步生成的MANIFEST.MF使用了SHA256 (SHA1) -RSA算法，用开发者私钥签名，然后在安装时使用公钥解密。最后，将其与未加密的摘要信息(MANIFEST.MF文件)进行对比，如果相符，则表明内容没有被修改。

具体打包过程

1.aapt 打包资源文件生成 R.java 文件；aidl 生成 java 文件 2.将 java 文件编译为 class 文件 3.将工程及第三方的 class 文件转换成 dex 文件 4.将 dex 文件、so、编译过的资源、原始资源等打包成 apk 文件 5.签名 6.资源文件对齐，减少运行时内存

通过AAPT工具进行资源文件 打包，生成R.java、resources.arsc和res文件

通过AIDL工具处理AIDL文件，生成对应的Java接口文件。

通过Java Compiler编译R.java、Java接口文件、Java源文件，生成.class文件。

通过dex命令，将.class文件和第三方库中的.class文件处理生成classes.dex，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。

通过ApkBuilder工具将资源文件、DEX文件打包生成APK文件。

通过Jarsigner工具，利用KeyStore对生成的APK文件进行签名。

assets和res/raw

这两个文件目录里的文件都会直接在打包apk的时候直接打包到apk中，携带在应用里面供应用访问，而且不会被编译成二进制；

他们的不同点在于：

1、assets中的文件资源不会映射到R中，而res中的文件都会映射到R中，所以raw文件夹下的资源都有对应的ID；

2、assets可以有更深的目录结构，而res/raw里面只能有一层目录；

3、资源存取方式不同，assets中利用AssetsManager，而res/raw直接利用getResource()，
openRawResource(R.raw.fileName)，很多人认为是R.id.filename，其实正确的是R.raw.filename，就像
R.drawable.filename一样，整体表示一个ID值，并非是R.id.filename；

20.3.Android的签名机制，签名如何实现的,v2相比于v1签名机制的改变

<https://blog.csdn.net/freekiteyu/article/details/84849651>

签名工具

Android 应用的签名工具有两种：jarsigner 和 signAPK。它们的签名算法没什么区别，主要是签名使用的文件不同

jarsigner: jdk 自带的签名工具，可以对 jar 进行签名。使用 keystore 文件进行签名。生成的签名文件默认使用 keystore 的别名命名。

signAPK: Android sdk 提供的专门用于 Android 应用的签名工具。signapk.jar 是 Android 源码包中的一个签名工具。代码位于 Android 源码目录下，signapk.jar 可以编译 build/tools/signapk/ 得到。使用 pk8、x509.pem 文件进行签名。其中 pk8 是私钥文件，x509.pem 是含有公钥的文件。生成的签名文件统一使用“CERT”命名。

jarsigner和apksigner的区别

Android 提供了两种对 Apk 的签名方式，一种是基于 JAR 的签名方式，另一种是基于 Apk 的签名方式，它们的主要区别在于使用的签名文件不一样：jarsigner 使用 keystore 文件进行签名；apksigner 除了支持使用 keystore 文件进行签名外，还支持直接指定 pem 证书文件和私钥进行签名。

签名过程

APK 是先摘要，再签名

要了解如何实现签名，需要了解两个基本概念：数字摘要和数字证书。

数字摘要

就是对消息数据，通过一个 Hash 算法计算后，都可以得到一个固定长度的 Hash 值，这个值就是消息摘要。

特征：唯一性 固定长度：比较常用的 Hash 算法有 MD5 和 SHA1，MD5 的长度是 128 拉，SHA1 的长度是 160 位。不可逆性

消息摘要只能保证消息的完整性，并不能保证消息的不可篡改性

数字签名

在摘要的基础上再进行一次加密，对摘要加密后的数据就可以当作数字签名。利用非对称加密技术，通过私钥对摘要进行加密，产生一个字符串，如 RSA 就是常用的非对称加密算法。在没有私钥的前提下，非对称加密算法能确保别人无法伪造签名，因此数字签名也是对发送者信息真实性的一个有效证明。不过由于 Android 的 keystore 证书是自签名的，没有第三方权威机构认证，用户可以自行生成 keystore，Android 签名方案无法保证 APK 不被二次签名。

签名和校验的主要过程

选取一个签名后的 APK (Sample-release.apk) 解压，在 META-INF 文件夹下有三个文件：MANIFEST.MF、CERT.SF、CERT.RSA。它们就是签名过程中生成的文件

MANIFEST.MF

逐一遍历 APK 中的所有条目，如果是目录就跳过，如果是一个文件，就用 SHA1（或者 SHA256）消息摘要算法提取出该文件的摘要然后进行 BASE64 编码。分别用 Name 和 SHA1-Digest 记录

```
MANIFEST.MF
1 Manifest-Version: 1.0
2 Built-By: Generated-by-ADT
3 Created-By: Android Gradle 3.2.1
4
5 Name: AndroidManifest.xml
6 SHA1-Digest: tu6500oYAigLcHHlBBMNWR0SwQo=
7
8 Name: META-INF/android.arch.core_runtime.version
9 SHA1-Digest: BeF7ZGqBckDCBhhvlPj0xwl01dw=
10
```

CERT.SF

SHA1-Digest: 对 MANIFEST.MF 的各个条目做 SHA1 (或者 SHA256) 后再用 Base64 编码

```
CERT.SF
1 Signature-Version: 1.0
2 Created-By: 1.0 (Android)
3 SHA1-Digest-Manifest: hcqSIH7TInM/kaPi/qMyDQ0kS2A=
4
5 Name: AndroidManifest.xml
6 SHA1-Digest: dawhJgHMQ7qFjTQquIoUi/q9e08=
7
8 Name: META-INF/android.arch.core_runtime.version
9 SHA1-Digest: OPQCkzMXJVPQryHeMowVNZmfRMw=
10
```

[app4](#)

CERT.RSA

之前生成的 CERT.SF 文件，用私钥计算出签名，然后将签名以及包含公钥信息的数字证书一同写入 CERT.RSA 中保存

签名过程：

1、计算摘要：

通过Hash算法提取出原始数据的摘要。

2、计算签名：

再通过基于密钥（私钥）的非对称加密算法对提取出的摘要进行加密，加密后的数据就是签名信息。

3、写入签名：

将签名信息写入原始数据的签名区块内。

校验过程：

签名验证是发生在APK的安装过程中

1、计算摘要

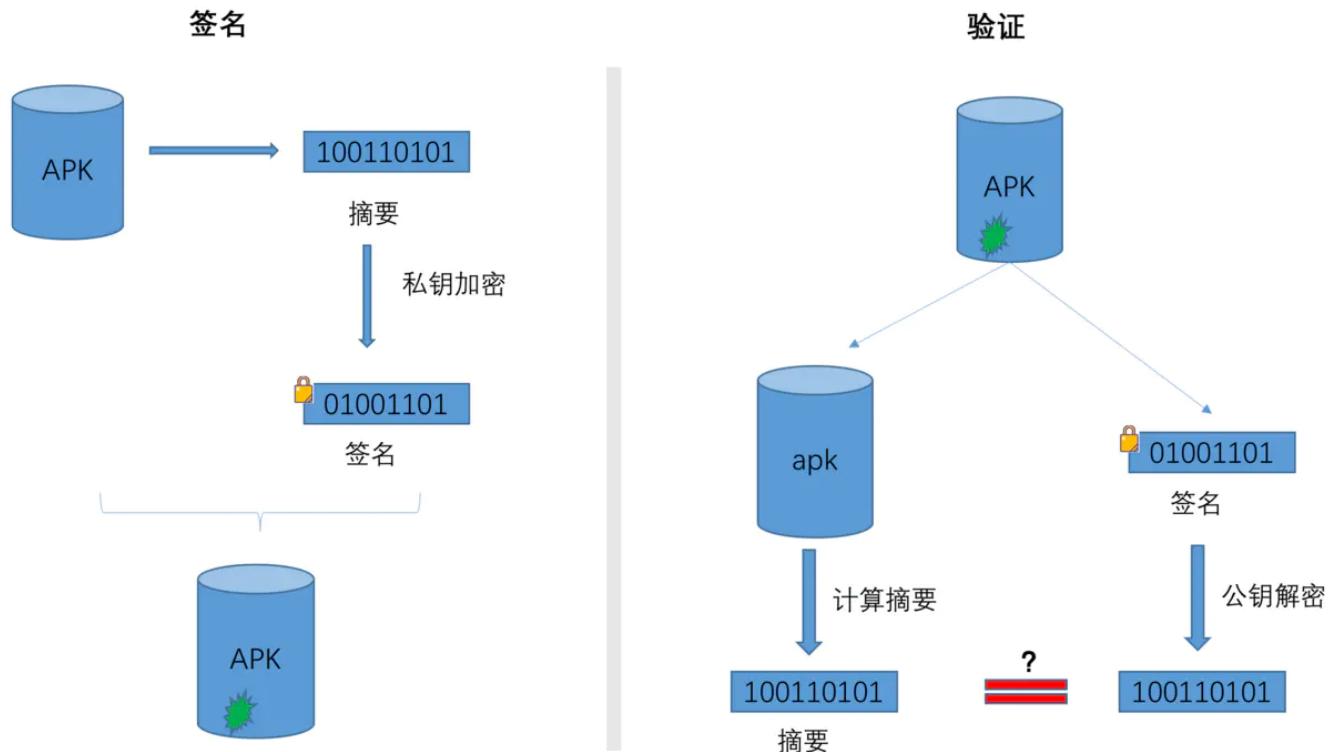
首先用同样的Hash算法从接收到的数据中提取出摘要。

2、解密签名：

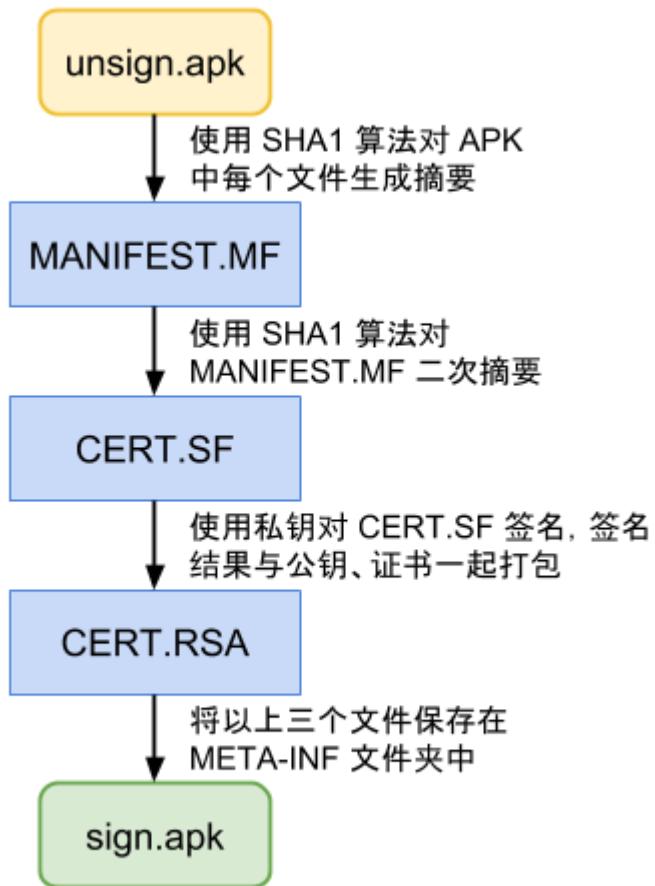
使用发送方的公钥对数字签名进行解密，解密出原始摘要。

3、比较摘要：

如果解密后的数据和提取的摘要一致，则校验通过；如果数据被第三方篡改过，解密后的数据和摘要将会不一致，则校验不通过。



Android Apk V1 签名过程



Android Apk V1 校验过程

1、解析出 CERT.RSA 文件中的证书、公钥，解密 CERT.RSA 中的加密数据。 2、解密结果和 CERT.SF 的指纹进行对比，保证 CERT.SF 没有被篡改。 3、而 CERT.SF 中的内容再和 MANIFEST.MF 指纹对比，保证 MANIFEST.MF 文件没有被篡改。 4、MANIFEST.MF 中的内容和 APK 所有文件指纹逐一对比，保证 APK 没有被篡改。

v2相比于v1签名机制的改变

v1 签名机制的劣势

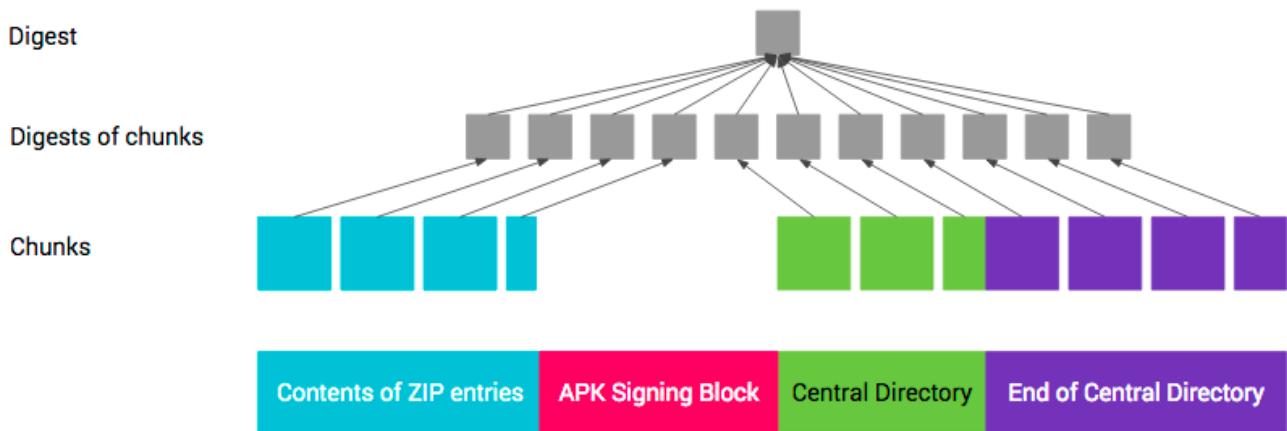
签名校验速度慢

校验过程中需要对apk中所有文件进行摘要计算，在 APK 资源很多、性能较差的机器上签名校验会花费较长时间，导致安装速度慢。

完整性保障不够

META-INF 目录用来存放签名，自然此目录本身是不计入签名校验过程的，可以随意在这个目录中添加文件

1.V2计算加快签名校验速度



就是把 APK 按照 1M 大小分割，分别计算这些分段的摘要，最后把这些分段的摘要在进行计算得到最终的摘要也就是 APK 的摘要。然后将 APK 的摘要 + 数字证书 + 其他属性生成签名数据写入到 APK Signing Block 区块

2.V2保证META-INFO目录不会被篡改

v2 签名模式在原先 APK 块中增加了一个新的块（签名块），新的块存储了签名，摘要，签名算法，证书链，额外属性等信息

为了保护 APK 内容，整个 APK (ZIP文件格式) 被分为以下 4 个区块：

头文件区、V2签名块、中央目录、尾部。

应用签名方案的签名信息会被保存在 区块 2 (APK Signing Block) 中，而区块 1 (Contents of ZIP entries)、区块 3 (ZIP Central Directory)、区块 4 (ZIP End of Central Directory) 是受保护的，在签名后任何对区块 1、3、4 的修改都逃不过新的应用签名方案的检查。

数字证书

如果数字签名和公钥一起被篡改，接收方无法得知，还是会校验通过。

如何保证公钥的可靠性呢？

证书授权机构——CA，小明去CA机构申请证书，将小明的个人信息、公钥生成一个证书，然后把这个证书发送给 jack，jack拿这个证书去证书授权机构查询，如果能匹配上小明的信息，就说明这个证书是小明的，就可以使用证书中的公钥来解密小明的消息。

接收方收到消息后，先向CA验证证书的合法性，再进行签名校验。

注意：Apk的证书通常是自签名的，也就是由开发者自己制作，没有向CA机构申请。Android在安装Apk时并没有校验证书本身的合法性，只是从证书中提取公钥和加密算法，这也正是对第三方Apk重新签名后，还能够继续在没有安装这个Apk的系统中继续安装的原因。

keystore

keystore文件中包含了私钥、公钥和数字证书。

除了要指定keystore文件和密码外，也要指定alias和key的密码，这是为什么呢？

keystore是一个密钥库，也就是说它可以存储多对密钥和证书，keystore的密码是用于保护keystore本身的，一对密钥和证书是通过alias来区分的

20.4 APK的安装流程

复制APK到/data/app目录下，解压并扫描安装包。

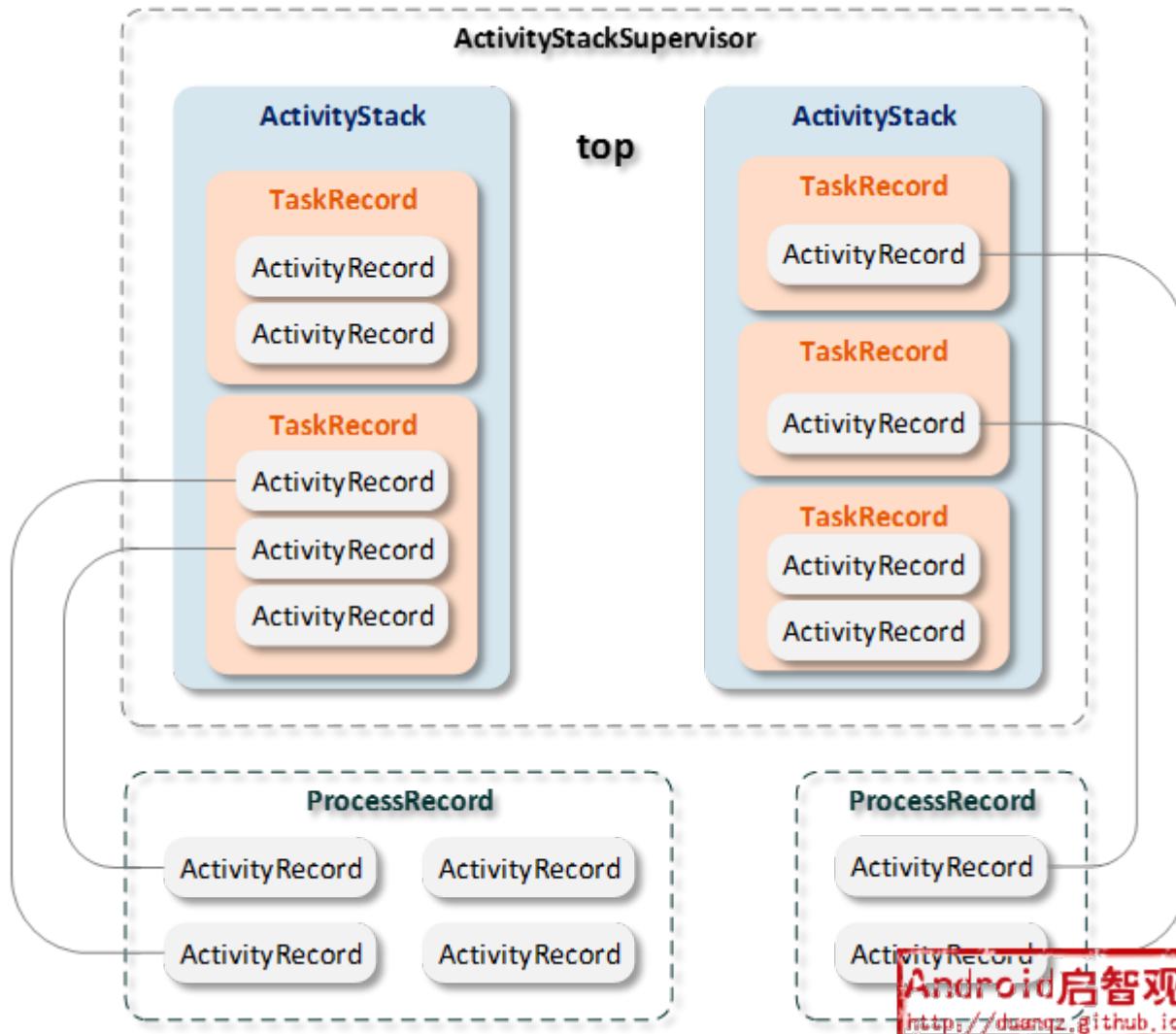
资源管理器解析APK里的资源文件。

解析AndroidManifest文件，并在/data/data/目录下创建对应的应用数据目录。

然后对dex文件进行优化，并保存在dalvik-cache目录下。

将AndroidManifest文件解析出的四大组件信息注册到PackageManagerService中。

安装完成后，发送广播。



21.序列化

21.1.什么是序列化,

Java序列化是指把Java对象转换为字节序列的过程

Java反序列化是指把字节序列恢复为Java对象的过程；

21.2.为什么需要使用序列化和反序列化

不同进程/程序间进行远程通信时，可以相互发送各种类型的数据，包括文本、图片、音频、视频等，而这些数据都会以二进制序列的形式在网络上传送。

当两个Java进程进行通信时，对于进程间的对象传送需要使用Java序列化与反序列化了。发送方需要把这个Java对象转换为字节序列，接收方则需要将字节序列中恢复出Java对象。

21.3. 序列化的有哪些好处

实现了数据的持久化，通过序列化可以把数据永久地保存到硬盘上（如：存储在文件里），实现永久保存对象。

利用序列化实现远程通信，即：能够在网络上传输对象。

21.4. Serializable 和 Parcelable 的区别

Serializable原理(<https://juejin.im/post/6844904049997774856>)

Serializable接口没有方法和属性，只是一个识别类可被序列化的标志。

Serializable是通过I/O读写存储在磁盘上的，通过反射解析出对象描述、属性的描述以HandleTable来缓存解析信息，之后解析成二进制，存储、传输。

Parcel原理(<https://www.wanandroid.com/wenda/show/9002>)

Parcel翻译过来是打包的意思，其实就是包装了我们需要传输的数据，然后在Binder中传输，也就是用于跨进程传输数据，将序列化之后的数据写入到一个共享内存中，其他进程通过Parcel可以从这块共享内存中读出字节流，并反序列化成对象。

它的各种writeXXX方法，在native层都是会调用Parcel.cpp的write方法

Serializable 和 Parcelable 的区别

存储媒介的不同(<https://www.jianshu.com/p/1b362e374354>)

Serializable 使用 I/O 读写存储在硬盘上，而 Parcelable 是直接 在内存中读写。很明显，内存的读写速度通常大于 IO 读写，所以在 Android 中传递数据优先选择 Parcelable。

效率不同

Serializable 会使用反射，序列化和反序列化过程需要大量 I/O 操作，

Parcelable 自己实现封送和解封（marshalled & unmarshalled）操作不需要用反射，数据也存放在 Native 内存中，效率要快很多。

21.5. 什么是serialVersionUID

<https://cloud.tencent.com/developer/article/1524781>

序列化是将对象的状态信息转换为可存储或传输的形式的过程。我们都知道，Java对象是保存在JVM的堆内存中的，也就是说，如果JVM堆不存在了，那么对象也就跟着消失了。

而序列化提供了一种方案，可以让你在即使JVM停机的情况下也能把对象保存下来的方案。就像我们平时用的U盘一样。把Java对象序列化成可存储或传输的形式（如二进制流），比如保存在文件中。这样，当再次需要这个对象的时候，从文件中读取出二进制流，再从二进制流中反序列化出对象。

虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化 ID 是否一致，这个所谓的序列化ID，就是我们在代码中定义的serialVersionUID。

21.6.为什么还要显示指定serialVersionUID的值?

如果不显示指定serialVersionUID, JVM在序列化时会根据属性自动生成一个serialVersionUID, 然后与属性一起序列化, 再进行持久化或网络传输. 在反序列化时, JVM会再根据属性自动生成一个新版serialVersionUID, 然后将这个新版serialVersionUID与序列化时生成的旧版serialVersionUID进行比较, 如果相同则反序列化成功, 否则报错.

如果显示指定了serialVersionUID, JVM在序列化和反序列化时仍然都会生成一个serialVersionUID, 但值为我们显示指定的值, 这样在反序列化时新旧版本的serialVersionUID就一致了.

在实际开发中, 不显示指定serialVersionUID的情况会导致什么问题? 如果我们的类写完后不再修改, 那当然不会有问题是, 但这在实际开发中是不可能的, 我们的类会不断迭代, 一旦类被修改了, 那旧对象反序列化就会报错. 所以在实际开发中, 我们都会显示指定一个serialVersionUID, 值是多少无所谓, 只要不变就行.

22.Art & Dalvik 及其区别

22.1Art & Dalvik 及其区别

<https://paul.pub/android-dalvik-vm/?spm=a2c6h.12873639.0.0.35ec6884Lt7nzq>

<https://paul.pub/android-art-vm/?spm=a2c6h.12873639.0.0.35ec6884Lt7nzq>

Dalvik, ART是Android的两种运行环境, 也可以叫做Android虚拟机JIT, AOT是Android虚拟机采用的两种不同的编译策略

在Dalvik虚拟机上, APK中的Dex文件在安装时会被优化成odex文件, 在运行时, 会被JIT编译器编译成native代码。

在ART虚拟机上安装时, Dex文件会直接由dex2oat工具翻译成oat格式的文件, oat文件中既包含了dex文件中原先的内容, 也包含了已经编译好的native代码。

22.1.1.Dalvik

1.原理

一个应用首先经过DX工具将class文件转换成Dalvik虚拟机可以执行的dex文件, 然后由类加载器加载原生类和Java类。 Dalvik虚拟机负责解释器根据指令集对Dalvik字节码进行释dex文件为机器码。

JIT编译器

Dalvik负责将dex翻译为机器码交由系统调用, 有一个缺陷, 每次执行代码, 都需要Dalvik将操作码代码翻译为机器对应的微处理器指令, 然后交给底层系统处理, 运行效率很低。

JIT编译器, 当App运行时, 每当遇到一个新类, JIT编译器就会对这个类进行即时编译, 经过编译后的代码, 会被优化成相当精简的原生型指令码(即native code), 这样在下次执行到相同逻辑的时候, 速度就会更快。

2.Dalvik的启动流程

Dalvik进程管理是依赖于linux的进程体系结构的, 如要为应用程序创建一个进程, 它会使用linux的fork机制来复制一个进程。

22.1.2.ART

1.原理

JIT是运行时编译，这样可以对执行次数频繁的dex代码进行编译和优化，减少以后使用时的翻译时间，但将dex翻译为本地机器码也要占用时间，所以Google在4.4之后推出了ART，用来替换Dalvik。

ART的策略与Dalvik不同，在ART环境中，应用在第一次安装的时候，字节码就会预先编译成机器码，使其成为真正的本地应用。之后打开App的时候，不需要额外的翻译工作，直接使用本地机器码运行，因此运行速度提高。

AOT

AOT是静态编译，应用在安装的时候会启动dex2oat过程把dex预编译成ELF文件，每次运行程序的时候不用重新编译。

23.模块化&组件化

23.1.什么是模块化

原本一个App模块承载了所有的功能，而模块化就是拆分成多个模块放在不同的Module里面，每个功能的代码都在自己所属的module中添加

通常还会有一个通用基础模块module_common，提供 BaseActivity/BaseFragment、图片加载、网络请求等基础能力，然后每个业务模块都会依赖这个基础模块。业务模块之间有有依赖

但多个模块中肯定会有页面跳转、数据传递、方法调用等情况，所以必然存在以上这种依赖关系，即模块间有着高耦合度。高耦合度加上代码量大，就极易出现上面提到的那些问题了，严重影响了团队的开发效率及质量。

23.2.什么是组件化

组件化，去除模块间的耦合，使得每个业务模块可以独立当做App存在，对于其他模块没有直接的依赖关系。此时业务模块就成为了业务组件。

23.3.组件化优点和方案

加快编译速度：每个业务功能都是一个单独的工程，可独立编译运行，拆分后代码量较少，编译自然变快。

提高协作效率：解耦使得组件之间彼此互不打扰，组件内部代码相关性极高。团队中每个人有自己的责任组件，不会影响其他组件；降低团队成员熟悉项目的成本，只需熟悉责任组件即可；对测试来说，只需重点测试改动的组件，而不是全盘回归测试。

组件化方案

<https://juejin.cn/post/6844904147641171981>

宿主app在组件化中，app可以认为是一个入口，一个宿主空壳，负责生成app和加载初始化操作。

业务层 每个模块代表了一个业务，模块之间相互隔离解耦，方便维护和复用。

公共层

既然是base，顾名思义，这里面包含了公共的类库。如Basexxx、toast,logutil, glide工具类，资源文件等

网络层

提供网络加载

三方库层

提供网络加载

组件化开发的问题点

<https://juejin.cn/post/6881116198889586701#heading-26>

<https://www.jianshu.com/p/8b6e6a50e21e>

<https://juejin.cn/post/6844904147641171981#heading-5>

23.4.组件独立调试

1.1 gradle.properties 中定义一个常量值 isModule
1.2 apply plugin 根据 boolean 值判断 if (isModule.toBoolean()) {
apply plugin: 'com.android.application' } else { apply plugin: 'com.android.library' }
1.3 配置 applicationId 和 manifest

23.5.组件间通信

跳转

ARouter 1.@Route 2.ARouter.getInstance().build("/xx/xx").navigation()

为彼此提供服务

既然首页组件可以访问购物车组件接口了，那就需要依赖购物车组件啊，这两组件还是耦合

1.首先在 commonlib 模块里创建一个暴露方法的接口，并定义接口签名，同时继承 Iprovide
1.首先在 commonlib 模块里创建一个暴露方法的接口，并定义接口签名，同时继承 Iprovider 接口

2.然后在 home 模块中继承 commonlib 里定义的接口，并实现签名方法。

3.Arouter 的 @Router 注解调用

r 接口

2.然后在 home 模块中继承 commonlib 里定义的接口，并实现签名方法。

3.Arouter 的 @Router 注解调用

23.6. Application 动态加载

组件有时候也需要获取应用的 Application，也需要在应用启动时进行初始化。这就涉及到组件的生命周期管理问题。

假设我们有组件 ModuleA、ModuleB、ModuleC，这3个组件内分别有 ModuleAAppLike、ModuleBAppLike、ModuleCAppLike，那么我们在壳工程集成时，怎么去组装他们呢。最简单的办法是，在壳工程的 Application.onCreate() 方法里执行

问题1：组件初始化的先后顺序，上层业务组件是依赖下层业务组件的，那么我们在加载组件时，必然要先加载下层组件，否则加载上层组件时可能会出现问题。

问题2：新增加一个组件，去修改壳工程代码，不利于代码维护。

解决

1. 定义一个注解来标识实现了 BaseAppLike 的类。

2. 通过 APT 技术，在组件编译时扫描和处理前面定义的注解，生成一个 BaseAppLike 的代理类

3. 组件集成后在应用的 Application.onCreate() 方法里，调用组件生命周期管理类的初始化方法。

4.组件生命周期管理类的内部，扫描到所有的BaseAppLikeProxy类名之后，通过反射进行类实例化。

难点

需要了解APT技术，怎么在编译时动态生成java代码；

1创建一个注解类 2定义baseapplication接口,设置优先级 3继承AbstractProcessor process中，生成代理类，并写入到文件里 (StringBuilder.append)

应用在运行时，怎么能扫描到某个包名下有多少个class，以及他们的名称呢；

如果有十多个组件里都有实现IAppLike接口的类，最终我们也会生成10多个代理类，这些代理类都是在同一个包下面运行时读取手机里的dex文件，从中读取出所有的class文件名，根据我们前面定义的代理类包名，来判断是不是我们的目标类，这样扫描一遍之后，就得到了固定包名下面所有类的类名了通常一个安装包里，加上第三方库，class文件可能数以千计、数以万计，这让人有点杀鸡用牛刀的感觉。每次应用冷启动时，都要读取一次dex文件并扫描全部class，这个性能损耗是很大的，我们可以做点优化，在扫描成功后将结果缓存下来，下次进来时直接读取缓存文件

在应用编译成apk时，就已经全量扫描过一次所有的class，并提取出所有实现了IAppLike接口的代理类呢，这样在应用运行时，效率就大大提升了。答案是肯定的，这就是gradle插件、动态插入java字节码技术。

采用gradle插件技术，在应用打包编译时，动态插入字节码来实现

<https://www.jianshu.com/p/3ec8e9574aaf>

1Gradle Transform在打包前去扫描所有的class文件

Gradle Transform技术，简单来说就是能够让开发者在项目构建阶段即由class到dex转换期间修改class文件

nputs就是所有扫描到的class文件或者是jar包，一共2种类型 遍历查找所有的jar包

2通过ASM动态修改字节码

init方法里找到 AppLifeCycleManager里的addClassFile()方法，我们在这个方法里插入字节码 通过反射创建的实例

23.7.ARouter原理

<https://juejin.cn/post/6885932290615509000#heading-1>

<https://juejin.cn/post/6844903648690962446#heading-0>

1.ARouter 路由表生成原理

@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

APT是Annotation Processing Tool的简称,即注解处理工具。它是在编译期对代码中指定的注解进行解析，然后做一些其他处理（如通过javapoet生成新的Java文件）

第一步：定义注解处理器，用来在编译期扫描加入@Route注解的类，然后做处理。这也是apt最核心的一步，新建RouterProcessor 继承自 AbstractProcessor,然后实现process方法。在项目编译期会执行RouterProcessor的process()方法，我们便可以在这个方法里处理Route注解了

第二步，在process()方法里开始生成EaseRouter_Route_moduleName类文件和EaseRouter_Group_moduleName文件。这里在process()里生成文件用javapoet生成java文件，就会用 JavaPoet 生成 Group、Provider 和 Root 路由文件，路由表就是由这些文件组成的，内容loadInto方法通过传入一个特定类型的map就能把分组信息放入map里为，一个map(其实是两个map，一个保存group列表，一个保存group下的路由地址和activityClass关系)保存了路由地址和ActivityClass的映射关系，然后通过map.get("router address") 拿到ActivityClass，通过startActivity()调用就好了

2.ARouter 路由表加载原理

<https://github.com/Xiasm/EasyRouter/wiki/%E6%A1%86%E6%9E%B6%E7%9A%84%E5%88%9D%E5%A7%8B%E5%8C%96>

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)到这些类文件便可以得到所有的routerAddress---activityClass映射关系 去扫描apk中所有的dex，遍历找到所有包名为packageName的类名，然后将类名再保存到classNames集合里

1. 读取apk中所有的dex文件 2.然后判断类的包名是否为 “com.alibaba.android.arouter.routes”，获取到注解处理器生成的类名时，就会把这些类名保存 SharedPreferences 中，下次就根据 App 版本判断，如果不是新版本，就从本地中加载类名，否则就用 ClassUtils 读取类名。 3.就会根据类名的后缀判断类是 IRouteRoot、IInterceptorGroup 还是 IProviderGroup，然后根据不同的类把类文件的内容加载到索引中。获取到映射关系

3.ARouter 跳转原理

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()

1.在build的时候，传入要跳转的路由地址，build()方法会返回一个Postcard对象，我们称之为跳卡。然后调用Postcard的navigation()方法完成跳转

2.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()

24.热修复&插件化

<https://zhuanlan.zhihu.com/p/33017826>

<https://juejin.cn/post/6844903613865672718#heading-1>

<https://fashare2015.github.io/2018/01/24/dynamic-load-learning-load-activity/>

24.1.插件化的定义

所谓插件化，就是让我们的应用不必再像原来一样把所有的内容都放在一个apk中，可以把一些功能和逻辑单独抽出来放在插件apk中，然后主apk做到 [按需调用] 。

24.2.插件化的优势

- 1.减少主apk的体积、65535 问题。让应用更轻便
- 2.让用户不用重新安装 APK 就能升级应用功能，减少发版本频率，增加用户体验。
- 3.模块化、解耦合

应用程序的工程和功能模块数量会越来越多，如果功能模块间的耦合度较高，修改一个模块会影响其它功能模块，势必会极大地增加成本。

24.3.插件化框架对比

- 1.最早的插件化框架：2012年大众点评的屠毅敏就推出了AndroidDynamicLoader框架。
 - (1) 首先通过 DLPluginManager 的 loadApk 函数加载插件，这步每个插件只需调用一次。
 - (2) 通过 DLPluginManager 的 startPluginActivity 函数启动代理 Activity。

(3) 代理 Activity 启动过程中构建、启动插件 Activity

2. 目前主流的插件化方案有滴滴任玉刚的VirtualApk、Wequick的Small框架。

Hook IActivityManager和Hook Instrumentation。主要方案就是先用一个在AndroidManifest.xml中注册的Activity来进行占坑，用来通过AMS的校验，接着在合适的时机用插件Activity替换占坑的Activity。

24.4. 插件化流程

在Android中应用插件化技术，其实也就是动态加载的过程，分为以下几步：

把可执行文件（.so/dex/jar/apk等）拷贝到应用APP内部。

加载可执行文件，更换静态资源

调用具体的方法执行业务逻辑

24.5. 插件化原理

一. 类加载原理

1.1 apk被安装之后，APK文件的代码以及资源会被系统存放在固定的目录（比如/data/app/package_name/base-1.apk）系统在进行类加载的时候，会自动去这一个或者几个特定的路径来寻找这个类；

系统无法加载我们插件中的类。需要我们自己处理这个类加载的过程。

1.2 这里用的DexClassLoader Android中的类加载器中主要包括三类BootClassLoader（继承ClassLoader），PathClassLoader和DexClassLoader，后两个继承于BaseDexClassLoader。

BootClassLoader：主要用于加载系统的类，包括java和android系统的类库。（比如TextView, Context，只要是系统的类都是由BootClassLoader加载完成）。

PathClassLoader：主要用于加载我们应用程序内的类。路径是固定的，只能加载 /data/app，无法指定解压释放dex的路径，无法动态加载。对于我们的应用默认为PathClassLoader

DexClassLoader：可以用来加载任意路径的zip,jar或者apk文件。

DexClassLoader重载findClass方法，在加载类时会调用其内部的DexPathList去加载。而DexPathList的loadClass会去遍历DexFile直到找到需要加载的类。

1.3 插件化中有单DexClassLoader和多DexClassLoader两种结构。

插件化要解决的是

主工程调用插件

有单DexClassLoader

对于每个插件都会生成一个DexClassLoader，

当加载该插件中的类时需要通过对DexClassLoader加载，需要先通过插件的ClassLoader加载该类再通过反射调用其方法

多DexClassLoader

将插件的DexClassLoader中的pathList合并到主工程的DexClassLoader中。主工程则可以直接通过类名去访问插件中的类

插件调用主工程

在构造插件的ClassLoader时会传入主工程的ClassLoader作为父加载器，所以插件是可以直接可以通过类名引用主工程的类。

双亲委派机制：ClassLoader在加载一个字节码时，首先会询问当前的ClassLoader是否已经加载过此类，如果已经加载过就直接返回，不在重复的去加载，如果没有的话，会查询它的parent是否已经加载过此类，如果加载过那么就直接返回parent加载过的字节码文件，如果整个继承线路上都没有加载过此类，最后由子ClassLoader执行真正的加载。

二. 资源加载原理

2.1 Android系统通过Resource对象加载资源，Resource对象的生成只要将插件apk的路径加入到AssetManager中，便能够实现对插件资源的访问。

和代码加载相似，插件和主工程的资源关系也有两种处理方式：

2.2 合并式：addAssetPath时加入所有插件和主工程的路径；

独立式：各个插件只添加自己apk路径

三. Activity加载原理

代理：dynamic-load-apk采用。Hook：主流。

Hook实现方式有两种：Hook IActivityManager和Hook Instrumentation。

主要方案就是先用一个在AndroidManifest.xml中注册的Activity来进行占坑，用来通过AMS的校验，接着在合适的时机用插件Activity替换占坑的Activity。

3.1 Hook IActivityManager：

3.1.1 占坑、通过校验：

hook点

IActivityManager

Activity启动过程

startActivity-Instrumentation.execStartActivity()->ActivityManager.getService().startActivity()->
IActivityManager.Stub.asInterface->AMS.startActivity()

ActivityManager中getService()借助Singleton类实现单例，而且该单例是静态的，IActivityManager是一个比较好的Hook点。由于Hook点IActivityManager是一个接口(源码中IActivityManager.aidl文件)，建议这里采用动态代理。

过程

1.1 AndroidManifest.xml中注册SubActivity

1.2 拦截startActivity方法，获取参数args中保存的Intent对象，它是原本要启动插件TargetActivity的Intent。

1.3 新建一个subIntent用来启动StubActivity，并将前面得到的TargetActivity的Intent保存到subIntent中，便于以后还原TargetActivity。

代码

1、反射获取ActivityManager类中的静态实例IActivityManagerSingleton

2.反射获取Singleton中的mInstance实例

3、获取此IActivityManagerSingleton内部的mInstance

4. 动态代理创建代理IActivityManager

5、将重写的代理IActivityManager设置给mInstance

3.1.2 还原插件Activity：

hook点

Handler中的mCallback

activity启动过程中,AMS会远程掉用applicationThread的scheduleLaunchActivity。

ActivityThread中的Handler-> h的handleLaunchActivity处理LAUNCH_ACTIVITY类型的消息-> ActivityThread#handleLaunchActivity-> instmentiong启动activity->Activity的onCreate方法。

在Handler的dispatchMessage处理消息的这个方法中，看到如果Handler的Callback类型的mCallBack不为null，就会执行mCallback的handleMessage方法，因此mCallback可以作为Hook点。我们可以用自定义的Callback来替换mCallback。

过程

重写callback，当收到消息的类型为LAUNCH_ACTIVITY时，将启动SubActivity的Intent替换为启动TargetActivity的Intent。

反射获取ActivityThread，反射获取mH

替换callback

使用时则在application的attachBaseContext方法中进行hook即可。

3.2 Hook Instrumentation：

与Hook IActivity实现不同的是，用占坑Activity替换插件Activity以及还原插件Activity的地方不同。

分析：

在Activity通过AMS校验前，会调用Activity的startActivityForResult方法

并会调用了Instrumentation的execStartActivity方法来激活Activity的生命周期。并且在ActivityThread的performLaunchActivity中使用了mInstrumentation的新newActivity方法，其内部会用类加载器来创建Activity的实例。

方案：

1. 在Instrumentation的execStartActivity方法中用占坑SubActivity来通过AMS的验证

首先检查TargetActivity是否已经注册，如果没有则将TargetActivity的ClassName保存起来用于后面还原。接着把要启动的TargetActivity替换为StubActivity，最后通过反射调用execStartActivity方法，这样就可以用StubActivity通过AMS的验证。

2. 在Instrumentation的新newActivity方法中还原TargetActivity

在newActivity方法中创建了此前保存的TargetActivity，完成了还原TargetActivit。

用InstrumentationProxy替换mInstrumentation。

3、插件Activity的生命周期：

AMS和ActivityThread之间的通信采用了token来对Activity进行标识，并且此后的Activity的生命周期处理也是根据token来对Activity进行标识的，因为我们在Activity启动时用插件TargetActivity替换占坑SubActivity，这一过程在performLaunchActivity之前，因此performLaunchActivity的r.token就是TargetActivity。所以TargetActivity具有生命周期。

25.AOP

<https://blog.csdn.net/u010289802?t=1>

<https://blog.csdn.net/u010289802/article/details/80183142>

<https://juejin.cn/user/4318537403878167/posts>

25.1.AOP是什么

AOP: 即面向切面编程。和OOP一样，是一种程序设计思想。实现是通过预编译方式和运行期动态代理实现程序功能的统一维护。

OOP (Object Oriented Programming) 面向对象设计就是一种典型的纵向编程方式。OOP更多关注的是对象Object本身的功能，对象之间的功能的联系往往不会考虑的那么详细。

1、继承模式比如之类和父类之间的这种关系就是一个很好的展示。

2、分层架构如MVP，每一层之间也是纵向关联在一起的。

切面，也叫横向，横切关注点是一个抽象的概念，它是指那些在项目中贯穿多个模块的业务。AOP在将横切关注点与业务主体进行分类，从而提高程序代码的模块化程度，则是将涉及到众多模块的某一类功能进行统一管理。

1 我们要为方法添加调用日志，那就必须为所有类的所有方法添加日志调用，尽管它们都是相同的。

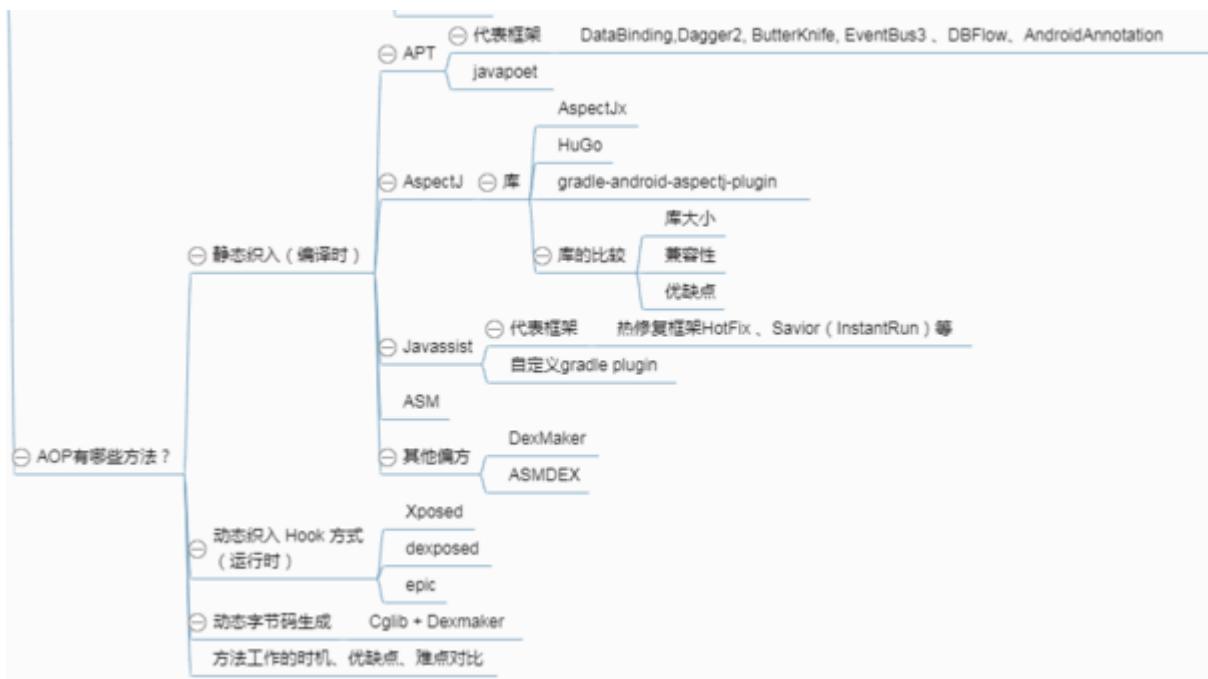
25.2.AOP的好处

使用切面的好处：

1.核心代码中不在包含记录日志的代码，类更专注于它的职责。2.如果想修改日志的记录方式，不需要修改Account类，只需要修改相应的配置文件和日志的实现，修改工作量小。3.解耦。为什么说解耦呢？用打印Log举例，app中可能存在不同的Log框架来实现。如果同一使用Log标签切入，那么在处理Log标签的地方可以统一Log框架。实现AOP的同时需要依赖注入，这又会减少模块间的依赖。

25.3.AOP的实现方式

<https://juejin.cn/post/6844903741808705544#heading-0>



<https://juejin.cn/post/6844903728525361165#heading-30>

Android中AOP的实现方式分两类：

运行时切入

(android hook机制)

集成Dexposed, epic框架 (运行时hook某些关键方法)

Java API实现动态代理机制 (基于反射, 性能不佳)

编译时切入

生成代理类(APT)

集成AspectJ框架(特殊的插件或编译器来生成特殊的class文件)

使用ASM,Javassist等字节码工具类来修改字节码(编译打包APK文件前修改class文件)

1.APT

APT (Annotation Processing Tool) 即注解处理器，是一种处理注解的工具，在编译时扫描和处理注解。注解处理器在编译期，通过注解生成 .java 文件。使用的 Annotation 类型是 SOURCE。

它在编译时扫描、解析、处理注解。它会对源代码文件进行检测，找出用户自定义的注解，根据注解、注解处理器和相应的apt工具自动生成代码。这段代码是根据用户编写的注解处理逻辑去生成的。最终将生成的新的源文件与原来的源文件共同编译（注意：APT并不能对源文件进行修改操作，只能生成新的文件，例如往原来的类中添加方法）

代表框架：DataBinding、Dagger2、ButterKnife、EventBus3、DBFlow、AndroidAnnotation

为什么这些框架注解实现 AOP 要使用 APT？

目前 Android 注解解析框架主要有两种实现方法，

一种是运行期通过反射去解析当前类，注入相应要运行的方法。

另一种是在编译期生成类的代理类，在运行期直接调用代理类的代理方法，APT 指的是后者。

如果不使用APT基于注解动态生成 java 代码，那么就需要在运行时使用反射或者动态代理，比如大名鼎鼎的 butterknife 之前就是在运行时反射处理注解，为我们实例化控件并添加事件，然而这种方法很大的一个缺点就是用了反射，导致 app 性能下降。所以后面 butterknife 改为 apt 的方式，可以留意到，butterknife 会在编译期间生成一个 XXX_ViewBinding.java。虽然 APT 增加了代码量，但是不再需要用反射，也就无损性能。

2.AspectJ

(AspectJ) 是编译期插入字节码，所以对性能也没什么影响。

通过Gradle Transform API，在class文件生成后至dex文件生成前，遍历并匹配所有符合AspectJ文件中声明的切点，然后将AspectJ的代码织入到目标.class。织入代码后的新.class会加入多个JoinPoint,这个JoinPoint会建立目标.class与AspectJ代码的连接，比如获得执行的对象、方法、参数等。

整个过程发生在编译期，是一种静态织入方式，所以会增加一定的编译时长，但几乎不会影响程序的运行时效率。

AspectJ 就是一个代码生成工具；编写一段通用的代码，然后根据 AspectJ 语法定义一套代码生成规则，AspectJ 就会帮你把这段代码插入到对应的位置去。

AspectJ 语法就是用来定义代码生成规则的语法。扩展编译器，引入特定的语法来创建 Advise，从而在编译期间就织入了Advise 的代码。如果使用过 Java Compiler Compiler (JavaCC)，你会发现两者的代码生成规则的理念惊人相似。JavaCC

3.ASM

ASM提供的API完全是面向Java字节码编程

ASM 是一个 Java 字节码层面的代码分析及修改工具，它有一套非常易用的 API，通过它可以实现对现有 class 文件的操纵，从而实现动态生成类，或者基于现有的类进行功能扩展。

Android 的编译过程中，首先会将 java 文件编译为 class 文件，之后会将编译后的 class 文件打包为 dex 文件，我们可以利用 class 被打包为 dex 前的间隙，插入 ASM 相关的逻辑对 class 文件进 ASM 是一个 Java 字节码层面的代码分析及修改工具，它有一套非常易用的 API，通过它可以实现对现有 class 文件的操纵，从而实现动态生成类，或者基于现有的类进行功能扩展。

Android 的编译过程中，首先会将 java 文件编译为 class 文件，之后会将编译后的 class 文件打包为 dex 文件，我们可以利用 class 被打包为 dex 前的间隙，插入 ASM 相关的逻辑对 class 文件进行操纵

4.epic

ART中的函数的调用约定,去修改函数的内容，将函数的前两条指令修改为跳转到自己自定义的逻辑，从而实现对任意方法的 Hook。

5.hook

<https://zhuanlan.zhihu.com/p/109157321>

反射/动态代理 如图中A点，作用于Java层。反射/动态代理是虚拟机提供的标准编程接口，可靠性较高。反射API可以帮助我们访问到private属性并修改，动态代理可以直接从Interface中动态的构造出代理对象，并去监控这个对象。

常见的用法是，用动态代理构造出一个代理对象，然后用反射API去替换进程中的对象，从而达到hook的目的。如：对Java Framework API的修改常用这种方法，修改ActivityThread、修当前进程的系统调用等。

缺点：只在java层，只能通过替换对象达到目的，适用范围较小

优点：稳定性好，调用反射和动态代理并不存在适配问题，技术门槛低

JNI Hook 如图中B点，java代码和native之间的调用是通过JNI接口调用的，所有JNI接口的函数指针都会被保存在虚拟机的一张表中。所以，java和native之间调用可以通过修改函数指针达到。

优点：稳定性高

缺点：只能hook Java和Native之间的native接口函数

ClassLoader 如图中C点，java代码的执行都是靠虚拟机的类加载器ClassLoader去加载，ClassLoader默认的双亲委派机制保证了ClassLoader总是从父类优先去加载java class。所以一类hook方案就是通过修改ClassLoader加载java class的Path路径达到目的。常见的应用场景有一些热修复技术。

优点：稳定性高

缺点：需要提前编译好修改后的class去替换，灵活性降低了

Xposed相关 如图中D点，这类hook技术的原理都是去修改ART/Dalvik虚拟机，虚拟机为java提供运行时环境，所有的java method都保存在虚拟机一张Map维护，每个Java Method都有个是否是JNI函数的标志位，如果是JNI函数则去查找对应的native函数。所以，一个hook方案是通过把要hook的函数修改为JNI函数，然后实现一个对应的native函数从而达到hook。

大量的一些自动化测试、动态调试都采用这个方法

优点：java层所有的class都可以修改，Activity等都可以注入。灵活性极高。

缺点：ART/Dalvik每次Android系统发布大版本都会被大改，导致每个Android版本都要去适配。稳定性变差。

前面hook技术都是去修改虚拟机中的java层，如果一个应用还包含Native code话，则得使用不同hook技术

26.jectpack

26.1. Navigation

<https://mp.weixin.qq.com/s/1URoDU0zgoYISQM8zYqx9w>

what

<https://juejin.cn/post/6844904131577004039#heading-0>

<https://www.jianshu.com/p/5c1763b0c9eb>

<https://zhuanlan.zhihu.com/p/69562454>

对于单个Activity嵌套多个Fragment的UI架构方式，对Fragment的管理一直是一个比较麻烦的事情。

需要通过FragmentManager和FragmentTransaction来管理Fragment之间的切换 对应用程序的App bar的管理，Fragment间的切换动画 Fragment间的参数传递 总之，使用起来不是特别友好。

Navigation是用来管理Fragment的切换，并且可以通过可视化的方式，看见App的交互流程

why

<https://www.jianshu.com/p/66b93df4b7a6> <https://zhuanlan.zhihu.com/p/69562454>

可以可视化的编辑各个组件之间的跳转关系

优雅的支持fragment之间的转场动画

通过第三方的插件支持fragment之间安全的参数传递

通过NavigationUI类，对菜单，底部导航，抽屉菜单导航进行方便统一的管理

支持通过deeplink直接定位到fragment

how

<https://juejin.cn/post/6844904068897308679#heading-0>

1 创建导航图

在 res 目录内创建一个 navigation 资源目录

根标签是navigation，它需要有一个属性startDestination，表示默认第一个显示的界面

每个fragment标签代表一个fragment类

每个action标签就相当于上图中的每条线，代表了执行切换的时候的目的地，切换动画等信息。

2 添加NavHostFragment

标签为fragment， android:name就是NavHost的实现类，这里是NavHostFragment app:navGraph 属性就是我们前面在res文件夹下创建的文件

Activity的布局中

1. 开启导航

通过Navigation#findNavController方法找到NavController，调用它的navigate方法开始导航。

```
view.findViewById(R.id.button).setOnClickListener(v -> { Bundle bundle = new Bundle();
bundle.putString("title","我是前面传过来的");
Navigation.findNavController(v).navigate(R.id.action_firstFragment_to_secondFragment,bundle);});
```

<https://mp.weixin.qq.com/s/1URoDU0zgoYISQM8zYqx9w>

注意点：

1. 页面跳转和参数传递

页面间的跳转是通过action来实现

1、Bundle方式

第一种方式是通过Bundle的方式。NavController 的navigate方法提供了传入参数是Bundle的方法

```
2. 安全参数(SafeArg) build.gradle添加n:navigation-safe-args引用 添加 编译
DetailFragmentArgs.Builder().setProductName("苹果").setPrice(10.5f).build().toBundle(); NavController
contorller = Navigation.findNavController(view);
contorller.navigate(R.id.action_homeFragment_to_detailFragment, bundle);
```

接收 Bundle bundle = getArguments(); if(bundle != null){ mProductName =
DetailFragmentArgs.fromBundle(bundle).getProductName(); mPrice =
DetailFragmentArgs.fromBundle(bundle).getPrice(); }

2 动画 action中 enterAnim: 配置进场时目标页面动画 exitAnim: 配置进场时原页面动画 popEnterAnim: 配置回退时目标页面动画 popExitAnim: 配置回退时原页面动画 配置完后

3. 导航堆栈管理

Navigation 有自己的任务栈，每次调用navigate()函数，都是一个入栈操作，出栈操作有以下几种方式，下面详细介绍几种出栈方式和使用场景。

1、系统返回键

首先需要在xml中配置app:defaultNavHost="true"，才能让导航容器拦截系统返回键，点击系统返回键，是默认的出栈操作，回退到上一个导航页面。如果当栈中只剩一个页面的时候，系统返回键将由当前Activity处理。

2.popBackStack()或者navigateUp() 如果页面上有返回按钮，那么我们可以调用popBackStack()或者navigateUp()返回到上一个页面。

3.popUpTo 和 popUpToInclusive

我们看下面这个例子。假设有A,B,C 3个页面，跳转顺序是 A to B, B to C, C to A。依次执行几次跳转后，栈中的顺序是A>B>C>A>B>C>A。此时如果用户按返回键，会发现反复出现重复的页面，此时用户的预期应该是在A页面点击返回，应该退出应用。此时就需要在C到A的action中设置popUpTo="@+id/a"。这样在C跳转A的过程中会把B,C出栈。但是还会保留上一个A的实例，加上新创建的这个A的实例，就会出现2个A的实例。此时就需要设置popUpToInclusive=true。这个配置会把上一个页面的实例也弹出栈，只保留新建的实例。下面再分析一下设置成false的场景。还是上面3个页面，跳转顺序A to B, B to C. 此时在B跳C的action中设置 popUpTo="@+id/a"，popUpToInclusive=false. 跳到C后，此时栈中的顺序是AC。B被出栈了。如果设置popUpToInclusive=true. 此时栈中的保留的就是C。AB都被出栈了。

4 DeepLink

Navigation组件提供了对深层链接（DeepLink）的支持。通过该特性，我们可以利用PendingIntent或者一个真实的URL链接，直接跳转到应用程序的某个destination

```
1、 PendingIntent rivate PendingIntent getPendingIntent() { Bundle bundle = new Bundle();  
bundle.putString("productName", "香蕉"); bundle.putFloat("price", 6.66f); return Navigation  
.findNavController(this, R.id.fragment) .createDeepLink() .setGraph(R.navigation.nav_graph)  
.setDestination(R.id.detailFragment) .setArguments(bundle) .createPendingIntent(); }  
2、 URL连接
```

源码理解

<https://mp.weixin.qq.com/s/1URoDU0zgoYISQM8zYqx9w>

NavHostFragment

这是一个特殊的布局文件，Navigation Graph中的页面通过该Fragment展示

NavHostFragment

1.onInflate解析在xml配置的两个参数defaultNavHost， 和navGraph

2、onCreate 创建NavController

3、onCreateView 创建一个FrameLayout

4、onViewCreated 在这个函数中，把NavController设置给了父布局的view的中的ViewTag中
Navigation.findNavController(View)中 递归遍历view的父布局，查找是否有view含有id为
R.id.nav_controller_view_tag的tag, tag有值就找到了NavController。如果tag没有值.说明当前父容器没有
NavController

NavController

导航的主要工作都在NavController中，涉及xml解析，导航堆栈管理，导航跳转等方面

1.NavHostFragment把导航文件的资源id传给了NavController

2NavController把导航xml文件传递给了NavInflater解析导航xml文件

3生成NavGraph保存着xml中配置的导航目标NavDestination

NavController的navigate函

把所有Navigator的实例保存在了NavigatorProvider

navigator.navigate

Fragment实例是通过instantiateFragment创建的，这个函数中是通过反射的方式创建的Fragment实例，Fragment还是通过FragmentManager进行管理，是用replace方法替换新的Fragment，这就是说每次导航产生的Fragment都是一个新的实例，不会保存之前Fragment的状态

NavGraph

里面包含了一组NavDestination，每个NavDestination就是一个一个的页面，也就是导航目的地

NavigatorProvider

内部有个HashMap，用来存放Navigator，Navigator它是个抽象类，有三个比较重要的子类FragmentNavigator，ActivityNavigator，DialogFragmentNavigator

使用 <https://juejin.cn/post/6844904131577004039>

1.注解处理器的目标是，扫描出所有带FragmentDestination或者ActivityDestination的类，拿到注解中的参数和类的全类名，封装成对象放到map中，使用fastjson将map生成json字符串，保存在src/main/assets目录下面

https://blog.csdn.net/weixin_42575043/article/details/108709467

2.可以自定义FragmentNavigator解决Fragment重复创建的问题

26.2. DataBinding

<https://blog.csdn.net/LucasXu01/article/details/103807451>

<https://juejin.cn/post/6844903494831308814#heading-6>

<https://www.jianshu.com/p/c56a987347ff>

原理

1.APT预编译方式生成ActivityMainBinding和ActivityMainBindingImpl

2.处理布局的时候生成了两个xml文件

activity_main-layout.xml (DataBinding需要的布局控件信息)

activity_main.xml (Android OS 渲染的布局文件)

Model是如何刷新View

1.DataBindingUtil.setContentView方法将xml中的各个View赋值给 ViewDataBinding，完成findviewbyid的任务

2.当VM层调用notifyPropertyChanged方法时，最终在ViewDataBindingImpl的executeBindings方法中处理逻辑

View是如何刷新Model

ViewDataBindingImpl的executeBindings方法中在设置了双向绑定的控件上，为其添加对应的监听器，监听其变动，如：EditText上设置TextWatcher，具体的设置逻辑放置到了TextViewBindingAdapter.setTextWatcher里

当数据发生变化的时候，TextWatcher在回调onTextChanged()的最后，会通过textAttrChanged.onChange()回调到传入的mboundView2androidTextAttrChanged的onChange()。

使用

<https://juejin.cn/post/6844903872520011784#heading-0>

26.3. ViewModel

<https://blog.csdn.net/c10wtiybq1ye3/article/details/89934891>

https://www.jianshu.com/p/41c56570a266?utm_campaign=haruki&utm_content=note&utm_medium=seo_notes&utm_source=recommendation

<https://www.jianshu.com/p/ebdf656b6dd4>

https://blog.csdn.net/qq_15988951/article/details/105106867

how

viewmodel = ViewModelProvider(this).get(MyViewModel::class.java)

what

ViewModel 类旨在以注重生命周期的方式存储和管理界面相关的数据

why

Activity配置更改重建时(比如屏幕旋转)保留数据

问题

<https://blog.csdn.net/u014093134/article/details/104082453>

例如你的 APP 某个 Activity 中包含一个列表，因为配置更改而重新创建 Activity 后（例如众所周知的屏幕旋转发生后需手动保存数据在旋转后进行恢复），新 Activity 必须重新提取列表数据，对于简单数据，Activity 可以使用 onSaveInstanceState() 方法从 onCreate() 中的捆绑包恢复数据，但这种方法仅适合可以序列化再反序列化但少量数据，不适合数量可能较大但数据，如用户列表或位图

因为ViewModel的生命周期是比Activity还要长，所以ViewModel可以持久保存UI数据。

通常在系统首次调用 Activity 对象的 onCreate() 方法时请求 ViewModel。系统可能会在 Activity 的整个生命周期内多次调用 onCreate()，如在旋转设备屏幕时。所以当前Activity的生命周期不断变化，经历了被销毁重新创建，而 ViewModel 的生命周期没有发生变化，Activity因为配置更改或者被系统意外回收的时候，会自动保存数据。在 Activity 重建的时候就可以继续使用销毁之前保存的数据。

源码 <https://www.jianshu.com/p/ebdf656b6dd4>

ComponentActivity 中

onRetainNonConfigurationInstance是在onStop() 和 onDestroy()之间被调用，它内部会保存ViewModel数据；

它会被ActivityThread中performDestroyActivity方法调用，它执行在onDestroy生命周期之前

Activity的attach时会调用getLastNonConfigurationInstance来恢复数据

ViewModel将一直留在内存中，直到限定其存在时间范围的Lifecycle(activity destroy掉用clear) 永久消失：

UI组件(Activity与Fragment、Fragment与Fragment)间实现数据共享

当这两个 Fragment 各自获取 ViewModelProvider 时，它们会收到相同的 ViewModel 实例 ViewModelProvider 通过 ViewModelStore 获取 ViewModel，FragmentActivity 自身是持有 ViewModelStore

避免内存泄漏的发生。

<https://www.jianshu.com/p/41c56570a266>

引入了 ViewModel 和 LiveData 之后，可以实现 vm 和 view 的解耦，只是 view 引用 vm，而 vm 是不持有 view 的引用的。在 activity 退出之后即是还有网络在继续也不会引发内存泄漏和空指针异常

源码解析

<https://blog.csdn.net/c10wtiybq1ye3/article/details/89934891>

1. Factory 是 ViewModelProvider 的一个内部接口，它的实现类是拿来构建 ViewModel 实例

3. get mViewModelStore.get(key) create 通过 newInstance(application) 去实例化

ViewModelStore：和名字一样，就是存储 ViewModel 的，它里面定义了一个 HashMap 来存储 ViewModel，key 值是 ViewModel 全路径 + 一个默认的前缀

26.4. vm+LiveData

Viewmodel

<https://juejin.cn/post/6844904079265644551#heading-0>

<https://www.jianshu.com/p/35d143e84d42>

<https://www.jianshu.com/p/109644858928>

1.how

1. 通过 ViewModelProviders.of() 方法创建 ViewModel 对象

2. 在 Activity 或者 Fragment 中，是由 Activity 和 Fragment 来提供 ViewModelStore 类对象，每个 Activity 或者 Fragment 都有一个，目的是用于保存该页面的 ViewModel 对象

2. why

1. 管理 UI 界面数据，数据持久化（将加载数据与数据恢复从 Activity or Fragment 中解耦）

在 Android 系统中，需要数据恢复有如下两种场景：

场景1：资源相关的配置发生改变导致 Activity 被杀死并重新创建。场景2：资源内存不足导致低优先级的 Activity 被杀死。

使用 onSaveInstanceState 与 onRestoreInstanceState

onSaveInstanceState 只适合保存少量的可以被序列化、反序列化的数据

onRetainNonConfigurationInstance 方法，用于处理配置发生改变时数据的保存。随后在重新创建的 Activity 中调用 getLastNonConfigurationInstance 获取上次保存的数据

官方最终采用了 onRetainNonConfigurationInstance 的方式来恢复 ViewModel。

其实就是在屏幕旋转的时候，AMS通过Binder回调Activity的retainNonConfigurationInstances()方法，数据保存就是通过retainNonConfigurationInstances()方法保存在NonConfigurationInstances对象，而再一次使用取出ViewModel的数据的时候，就是从nc对象中取出ViewModelStore对象，而ViewModelStore对象保存有ViewModel集合，官方重写了onRetainNonConfigurationInstance方法，在该方法中保存了ViewModelStor

监听Activity声明周期，在onDestory方法被调用时，判断配置是否改变。如果没有发送改变，则调用Activity中的ViewModelStore的clear()方法，清除所有的ViewModel

2.Fragments 间共享数据

获取到了Activity的ViewModelStore对象，从而实现了Fragment之间共享ViewModel

为什么不同的Fragment使用相同的Activity对象来获取ViewModel，可以轻易的实现ViewModel共享？

讲道理，如果同学们仔细看了ViewModel的创建流程，这个问题自然迎刃而解。

因为不同的Fragment使用相同的Activity对象来获取ViewModel，在创建ViewModel之前都会先从Activity提供的ViewModelStore中先查询一遍是否已经存在该ViewModel对象。所以我们只需要先在Activity中同样调用一遍ViewModel的获取代码，即可让ViewModel存在于ViewModelStore中，从而不同的Fragment可以共享一份ViewModel了。

<https://juejin.cn/post/6844903919064186888#heading-2> (vm总结)

livedata

<https://zhuanlan.zhihu.com/p/76747541>

what LiveData是一个可被观察的数据容器类

它将数据包装起来，使得数据成为“被观察者”，页面成为“观察者”。这样，当该数据发生变化时，页面能够获得通知，进而更新UI。

可以看到它接收的第一个参数是一个LifecycleOwner对象，在我们的示例中即Activity对象。第二个参数是一个Observer对象。通过最后一行代码将Observer与Activity的生命周期关联在一起。

只有在页面处于激活状态（Lifecycle.State.ON_STARTED或Lifecycle.State.ON_RESUME）时，页面才会收到来自LiveData的通知，如果页面被销毁（Lifecycle.State.ON_DESTROY）

how

在页面中，我们通过LiveData.observe()方法对LiveData包装的数据进行观察，反过来，当我们想要修改LiveData包装的数据时，可通过LiveData.postValue()/LiveData.setValue()来完成。postValue()是在非UI线程中使用，如果在UI线程中，则使用setValue()方法。

why

不会发生内存泄漏

观察者会绑定到Lifecycle对象，并在其关联的生命周期遭到销毁后进行自我清理。

不再需要手动处理生命周期

如果观察者的生命周期处于非活跃状态（如返回栈中的Activity），则它不会接收任何LiveData事件。

getLifecycle().addObserver进行观察

activity实现LifecycleOwner，reprotofrgment注册

livedata继承LifecycleObserver, destroy销毁其他

26.5.liferecycle

<https://liuwangshu.cn/application/jetpack/3-lifecycle-theory.html>

<https://juejin.cn/post/6844903784166998023>

Lifecycle使用

LifecycleObserver:是一个空方法接口，用于标识观察者，对这个 Lifecycle 对象进行监听

LifecycleOwner: 是一个接口，持有方法Lifecycle getLifecycle()。

LifecycleRegistry 类用于注册和反注册需要观察当前组件生命周期的 LifecycleObserver

1.实现LifecycleOwner重写getLifecycle 返回mLifecycleRegistry, mLifecycleRegistry不同生命周期markState

2.继承LifecycleObserver

3.getLifecycle.addObserver注册LifecycleObserver

27.开源框架

27.1.Okhttp

1.execute执行

1.1.真正的请求交给了 RealCall 类

execute()方法执行RealCall的execute方法

```
client.dispatcher().enqueue(new AsyncCall(responseCallback));
```

利用dispatcher调度器enqueueAsyncCall，并通过回调（Callback）获取服务器返回的结果

1.2 Dispatcher

Dispatcher将call 加入到队列中，然后通过线程池来执行call

Dispatcher是一个任务调度器，它内部维护了三个双端队列： readyAsyncCalls: 准备运行的异步请求

runningAsyncCalls: 正在运行的异步请求 runningSyncCalls: 正在运行的同步请求

新来的请求放队尾，执行请求从对头部取。

1.3 线程池

这不是一个newCachedThreadPool吗？没错，除了最后一个threadFactory参数之外与newCachedThreadPool一毛一样，只不过是设置了线程名字而已，用于排查问题。

阻塞队列用的SynchronousQueue，它的特点是不存储数据，当添加一个元素时，必须等待一个消费线程取出它，否则一直阻塞。

通常用于需要快速响应任务的场景，在网络请求要求低延迟的大背景下比较合适，

采用责任链的模式来使每个功能分开，每个Interceptor自行完成自己的任务

2.拦截器

利用Builder模式配置各种参数，例如：超时时间、拦截器等

retryAndFollowUpInterceptor

失败和重定向拦截器

当请求内部抛出异常时，判定是否需要重试

当响应结果是3xx重定向时，构建新的请求并发送请求

BridgeInterceptor

封装request和response拦截

负责把用户构造的请求转换为发送到服务器的请求

把服务器返回的响应转换为用户友好的响应

CacheInterceptor

当在OkHttpClient中配置了缓存，则将这个Response缓存起来

ConnectInterceptor——连接服务，负责和服务器建立连接 这

负责了Dns解析和Socket连接

CallServerInterceptor

传输http的头部和body数据

3.addInterceptor 和 addNetworkInterceptor区别

在OkHttpClient.Builder的构造方法有两个参数，使用者可以通过addInterceptor 和 addNetworkInterceptor 添加自定义的拦截器

加拦截器的顺序可以知道 Interceptors 和 networkInterceptors 刚好一个在 RetryAndFollowUpInterceptor 的前面，一个在后面

责任链调用图可以分析出来，假如一个请求在 RetryAndFollowUpInterceptor 这个拦截器内部重试或者重定向了 N 次，那么其内部嵌套的所有拦截器也会被调用N次，同样 networkInterceptors 自定义的拦截器也会被调用 N 次。而相对的 Interceptors 则一个请求只会调用一次，所以在OkHttp的内部也将其称之为 Application Interceptor。

4.责任链模式

<https://juejin.cn/post/6844903792073261063#heading-12>

将处理者和请求者进行解耦

多个对象都有机会处理请求，将这些对象连成一个链，将请求沿着这条链传递。

在请求到达时，拦截器会做一些处理（比如添加参数等），然后传递给下一个拦截器进行处理。

5.缓存怎么处理

<https://juejin.cn/post/6844903552339410958#heading-4>

使用OkHttp的缓存

定义一个网络拦截器

Http协议 缓存的控制是通过首部的Cache-Control来控制

only-if-cache: 表示直接获取缓存数据，若没有数据返回，则返回504

有网络时访问服务器

无网络时返回缓存数据

1.自定义Interceptor,重写intercept设置header *2.OkHttpClient .cache// 设置缓存路径和缓存容量
*addNetworkInterceptor设置自定义缓存

不使用OkHttp的缓存

```
if (NetworkUtil.isConnected(mContext)) { response = chain.proceed(newRequest); saveCacheData(response);  
// 保存缓存数据 } else { // 不执行chain.proceed会打断责任链，即后面的拦截器不会被执行 response =  
getCacheData(chain.request().url()); // 获取缓存数据 }
```

6.Okhttp连接池

连接池是为了解决频繁的进行建立Sokcet连接 (TCP三次握手) 和断开Socket (TCP四次分手)

socket复用有何标准

get

1.http协议

1.在http 1.x协议下，所有的请求的都是顺序的，正在写入数据的socket无法被另一个请求复用 2.http2.0协议使用了多路复用技术，允许同一个socket在同一个时候写入多个流数据

http1.x协议下当前socket没有其他流正在读写时可以复用，否则不行，http2.0对流数量没有限制。

2.域名和http和ssl协议配置需要匹配

put

在连接池中找连接的时候会对比连接池中相同host的连接。

如果在连接池中找不到连接的话，会创建连接，创建完后会存储到连接池中。

27.2.Glide

2.1 Glide怎么绑定生命周期

<https://juejin.cn/post/6844903647877267463#heading-1>

Glide.with(Activity activity)的方式传入页面引用

1.创建无UI的Fragment，并绑定到当前activity

2.builder模式创建RequestManager，将fragment的lifecycle传入，这样Fragment和RequestManager就建立了联系

3.RequestManager实现LifecycleListener 是一个接口,回调中处理请求

2.1 Glide缓存机制内存缓存，磁盘缓存

<https://juejin.cn/post/6844904002551808013#comment>

Glide的缓存机制，主要分为2种缓存，一种是内存缓存，一种是磁盘缓存。

之所以使用内存缓存的原因是：防止应用重复将图片读入到内存，造成内存资源浪费。

之所以使用磁盘缓存的原因是：防止应用重复的从网络或者其他地方下载和读取数据

具体来讲，缓存分为加载和存储：

内存缓存分为弱引用和lru缓存

弱引用是缓存正在使用的图片，避免内存泄漏

将缓存图片的时候，写入顺序

弱引用缓存-》Lru算法缓存-》磁盘缓存中

当加载一张图片的时候，获取顺序

弱引用缓存-》Lru算法缓存-》磁盘缓存

2.3关于LruCache

LruCache 内部用LinkHashMap存取数据

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，这种顺序有两种，一种是LRU顺序，一种是插入顺序

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。

2.4 Glide与Picasso的区别

27.3.LruCache的原理是什么？

LruCache的实现需要两个数据结构：双向链表和哈希表。

双向链表用于记录元素被塞进cache的顺序，然后淘汰最久未使用的元素。

哈希表用于直接记录元素的位置，即用O(1)的时间复杂度拿到链表的元素。

get的操作逻辑：根据传入的key(图片url的MD5值)去哈希表里拿到对应的元素，如果元素存在，就把元素挪到链表的尾部。
put的操作逻辑：首先判断key是否在哈希表里面，如果在的话就去更新值，并把元素挪到链表的尾部。

如果不在哈希表里，说明是一个新的元素。这时候需要去判断此时cache的容量了，

如果超过了最大的容量，就淘汰链表头部的元素，再将新的元素插入链表的尾部，如果没有超过最大容量，直接在链表尾部追加新的元素。

为啥要用linkedHashMap的数据结构？？ HashMap是无序的，当我们希望有顺序地去存储key-value时，就需要使用LinkedHashMap了。

27.4.Glide如何绑定生命周期

27.5. Retrofit

1.通过建造者模式构建一个Retrofit实例

2.通过Retrofit对象的create方法返回一个Service的动态代理对象

3.调用service的方法的时候解析接口注解

4. 调用Okhttp的网络请求方法,通过 回调执行器 切换线程 (子线程 ->主线程)

动态代理

运行时创建的代理类，在委托类的方法前后去做一些事情

在运行过程中，会在虚拟机内部创建一个Proxy的类。通过实现InvocationHandler的接口，来代理委托类的函数。

使用动态代理来对接口中的注释进行解析，解析后完成OkHttp的参数构建。

优点

代理类原始类脱离联系,在原始类和接口未知的时候 就确定代理类的行为

27.6 LeakCanary

1. 四种引用

JVM通过垃圾回收器对这四种引用做不同的处理

1. 强引用

指向的对象任何时候都不会被回收,垃圾回收器宁愿抛出OOM也不会对该对象进行回收

2. 软引用

但是如果内存空间不足，才回去回收软引用中的对象.

3. 弱引用

当发生垃圾回收时，不管当前内存是否足够，都会将弱引用关联的对象进行回收。

4. 虚引用

虚引用必须和引用队列一同使用

ReferenceQueue

如果软/弱/虚引用中的对象被回收，那么软/弱/虚引用就会被JVM加入关联的引用队列ReferenceQueue中

是说我们可以通过监控引用队列来判断Reference引用的对象是否被回收，从而执行相应的方法。

1. 了Application类提供的registerActivityLifecycleCallback(ActivityLifecycleCallbacks callback)方法来注册ActivityLifecycleCallbacks回调，这样就能对当前应用程序中所有的Activity的生命周期事件进行集中处理，当监听到Activity或Fragment onDestroy()时，把他们放到一个弱引用WeakReference中。

2. 把弱引用WeakReference 关联到一个引用队列ReferenceQueue。（如果弱引用关联的对象被回收，则会把这个弱引用加入到ReferenceQueue中）。

3. 延时5秒检测ReferenceQueue中是否存在当前弱引用对象。

4. 如果检测不到说明可能发生泄露，通过gcTrigger.runGc()手动掉用GC。遍历ReferenceQueue中所有的记录，当未回收对象个数大于5个时,dump heap获取内存快照hprof文件。

6. 使用Shark解析hprof文件,Hprof.open()把heapDumpFile转换成Hprof对象,

7. 根据heap中的对象关系图HprofHeapGraph获取泄露对象的objectIds

8. 找出内存泄漏对象到GC roots的最短路径

9. 输出分析结果展示到页面。

享学课堂