**Android老司机收集的面试题集 包含答案**

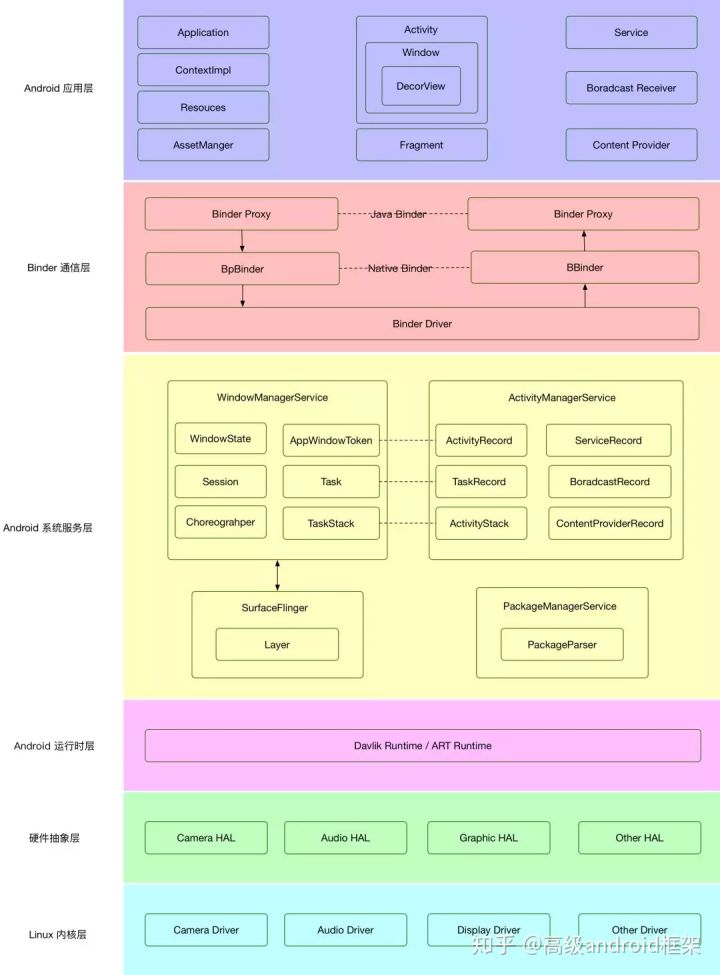
[高级android框架](https://www.zhihu.com/people/an-zhuo-an-zhuo-an-zhuo)

[**高级android框架**](https://www.zhihu.com/people/an-zhuo-an-zhuo-an-zhuo)

Android开发，安卓，性能优化、包括 OpenGLjava开发 NDK技术

8 人赞了该文章

**手画一下Android系统架构图，描述一下各个层次的作用？**



Android系统架构图

从上到下依次分为六层：

* 应用框架层
* 进程通信层
* 系统服务层
* Android运行时层
* 硬件抽象层
* Linux内核层

**Activity如与Service通信？**

可以通过bindService的方式，先在Activity里实现一个ServiceConnection接口，并将该接口传递给bindService()方法，在ServiceConnection接口的onServiceConnected()方法  
里执行相关操作。

**Service的生命周期与启动方法有什么区别？**

* startService()：开启Service，调用者退出后Service仍然存在。
* bindService()：开启Service，调用者退出后Service也随即退出。

Service生命周期：

* 只是用startService()启动服务：onCreate() -> onStartCommand() -> onDestory
* 只是用bindService()绑定服务：onCreate() -> onBind() -> onUnBind() -> onDestory
* 同时使用startService()启动服务与bindService()绑定服务：onCreate() -> onStartCommnad() -> onBind() -> onUnBind() -> onDestory

**广播分为哪几种，应用场景是什么？**

* 普通广播：调用sendBroadcast()发送，最常用的广播。
* 有序广播：调用sendOrderedBroadcast()，发出去的广播会被广播接受者按照顺序接收，广播接收者按照Priority属性值从大-小排序，Priority属性相同者，动态注册的广播优先，广播接收者还可以  
  选择对广播进行截断和修改。

**广播的两种注册方式有什么区别？**

* 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。

**广播发送和接收的原理了解吗？**

1. 继承BroadcastReceiver，重写onReceive()方法。
2. 通过Binder机制向ActivityManagerService注册广播。
3. 通过Binder机制向ActivityMangerService发送广播。
4. ActivityManagerService查找符合相应条件的广播（IntentFilter/Permission）的BroadcastReceiver，将广播发送到BroadcastReceiver所在的消息队列中。
5. BroadcastReceiver所在消息队列拿到此广播后，回调它的onReceive()方法。

**ContentProvider、ContentResolver与ContentObserver之间的关系是什么？**

* 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，ContentProvider为这些数据的访问提供了统一的接口，可以用来做进程间数据共享。
* ContentResolver：ContentResolver可以不同URI操作不同的ContentProvider中的数据，外部进程可以通过ContentResolver与ContentProvider进行交互。
* ContentObserver：观察ContentProvider中的数据变化，并将变化通知给外界。

**遇到过哪些关于Fragment的问题，如何处理的？**

* getActivity()空指针：这种情况一般发生在在异步任务里调用getActivity()，而Fragment已经onDetach()，此时就会有空指针，解决方案是在Fragment里使用  
  一个全局变量mActivity，在onAttach()方法里赋值，这样可能会引起内存泄漏，但是异步任务没有停止的情况下本身就已经可能内存泄漏，相比直接crash，这种方式  
  显得更妥当一些。
* Fragment视图重叠：在类onCreate()的方法加载Fragment，并且没有判断saveInstanceState==null或if(findFragmentByTag(mFragmentTag) == null)，导致重复加载了同一个Fragment导致重叠。（PS：replace情况下，如果没有加入回退栈，则不判断也不会造成重叠，但建议还是统一判断下）

@Override

protected void onCreate(@Nullable Bundle savedInstanceState) {

// 在页面重启时，Fragment会被保存恢复，而此时再加载Fragment会重复加载，导致重叠 ;

if(saveInstanceState == null){

// 或者 if(findFragmentByTag(mFragmentTag) == null)

// 正常情况下去 加载根Fragment

}

}

**Android里的Intent传递的数据有大小限制吗，如何解决？**

Intent传递数据大小的限制大概在1M左右，超过这个限制就会静默崩溃。处理方式如下：

* 进程内：EventBus，文件缓存、磁盘缓存。
* 进程间：通过ContentProvider进行款进程数据共享和传递。

**描述一下Android的事件分发机制？**

Android事件分发机制的本质：事件从哪个对象发出，经过哪些对象，最终由哪个对象处理了该事件。此处对象指的是Activity、Window与View。

Android事件的分发顺序：Activity（Window） -> ViewGroup -> View

Android事件的分发主要由三个方法来完成，如下所示：

// 父View调用dispatchTouchEvent()开始分发事件

public boolean dispatchTouchEvent(MotionEvent event){

boolean consume = false;

// 父View决定是否拦截事件

if(onInterceptTouchEvent(event)){

// 父View调用onTouchEvent(event)消费事件，如果该方法返回true，表示

// 该View消费了该事件，后续该事件序列的事件（Down、Move、Up）将不会在传递

// 该其他View。

consume = onTouchEvent(event);

}else{

// 调用子View的dispatchTouchEvent(event)方法继续分发事件

consume = child.dispatchTouchEvent(event);

}

return consume;

}

**描述一下View的绘制原理？**

View的绘制流程主要分为三步：

1. onMeasure：测量视图的大小，从顶层父View到子View递归调用measure()方法，measure()调用onMeasure()方法，onMeasure()方法完成绘制工作。
2. onLayout：确定视图的位置，从顶层父View到子View递归调用layout()方法，父View将上一步measure()方法得到的子View的布局大小和布局参数，将子View放在合适的位置上。
3. onDraw：绘制最终的视图，首先ViewRoot创建一个Canvas对象，然后调用onDraw()方法进行绘制。onDraw()方法的绘制流程为：① 绘制视图背景。② 绘制画布的图层。 ③ 绘制View内容。  
   ④ 绘制子视图，如果有的话。⑤ 还原图层。⑥ 绘制滚动条。

**requestLayout()、invalidate()与postInvalidate()有什么区别？**

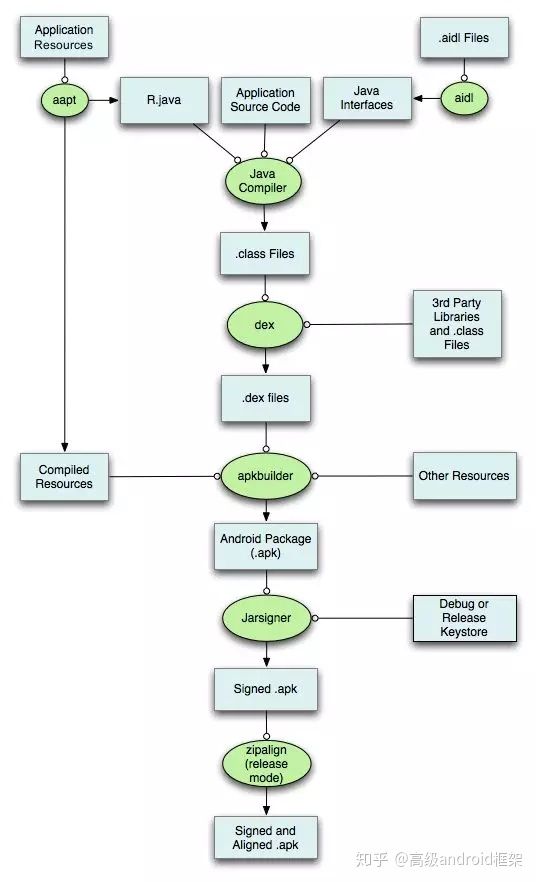
* requestLayout()：该方法会递归调用父窗口的requestLayout()方法，直到触发ViewRootImpl的performTraversals()方法，此时mLayoutRequestede为true，会触发onMesaure()与onLayout()方法，不一定  
  会触发onDraw()方法。
* invalidate()：该方法递归调用父View的invalidateChildInParent()方法，直到调用ViewRootImpl的invalidateChildInParent()方法，最终触发ViewRootImpl的performTraversals()方法，此时mLayoutRequestede为false，不会  
  触发onMesaure()与onLayout()方法，当时会触发onDraw()方法。
* postInvalidate()：该方法功能和invalidate()一样，只是它可以在非UI线程中调用。

一般说来需要重新布局就调用requestLayout()方法，需要重新绘制就调用invalidate()方法。

**了解APK的打包流程吗，描述一下？**

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

APK整体的的打包流程如下图所示：

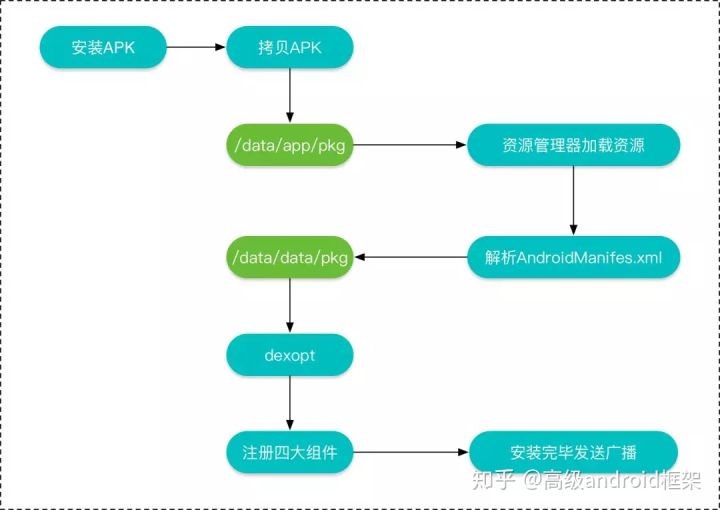


具体说来：

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

**了解APK的安装流程吗，描述一下？**

APK的安装流程如下所示：



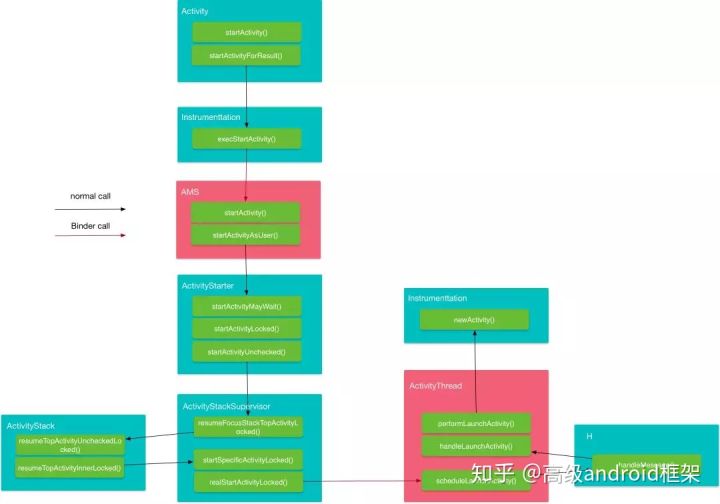
复制APK到/data/app目录下，解压并扫描安装包。

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

**当点击一个应用图标以后，都发生了什么，描述一下这个过程？**

点击应用图标后会去启动应用的LauncherActivity，如果LancerActivity所在的进程没有创建，还会创建新进程，整体的流程就是一个Activity的启动流程。

Activity的启动流程图（放大可查看）如下所示：



整个流程涉及的主要角色有：

* Instrumentation: 监控应用与系统相关的交互行为。
* AMS：组件管理调度中心，什么都不干，但是什么都管。
* ActivityStarter：Activity启动的控制器，处理Intent与Flag对Activity启动的影响，具体说来有：1 寻找符合启动条件的Activity，如果有多个，让用户选择；2 校验启动参数的合法性；3 返回int参数，代表Activity是否启动成功。
* ActivityStackSupervisior：这个类的作用你从它的名字就可以看出来，它用来管理任务栈。
* ActivityStack：用来管理任务栈里的Activity。
* ActivityThread：最终干活的人，是ActivityThread的内部类，Activity、Service、BroadcastReceiver的启动、切换、调度等各种操作都在这个类里完成。

注：这里单独提一下ActivityStackSupervisior，这是高版本才有的类，它用来管理多个ActivityStack，早期的版本只有一个ActivityStack对应着手机屏幕，后来高版本支持多屏以后，就有了多个ActivityStack，于是就引入了ActivityStackSupervisior用来管理多个ActivityStack。

整个流程主要涉及四个进程：

* 调用者进程，如果是在桌面启动应用就是Launcher应用进程。
* ActivityManagerService等所在的System Server进程，该进程主要运行着系统服务组件。
* Zygote进程，该进程主要用来fork新进程。
* 新启动的应用进程，该进程就是用来承载应用运行的进程了，它也是应用的主线程（新创建的进程就是主线程），处理组件生命周期、界面绘制等相关事情。

有了以上的理解，整个流程可以概括如下：

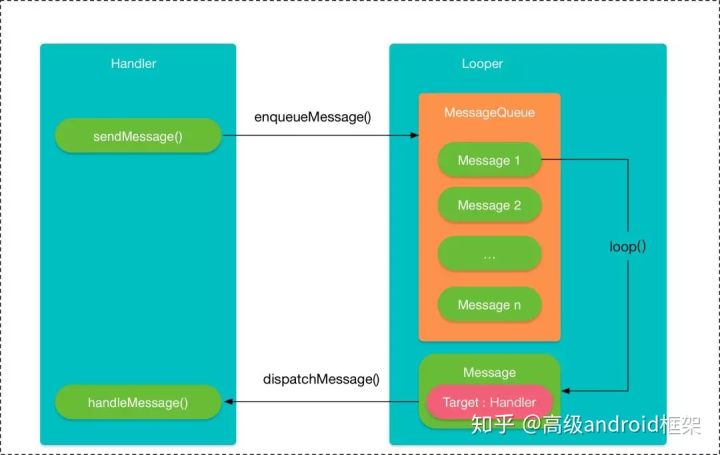
1. 点击桌面应用图标，Launcher进程将启动Activity（MainActivity）的请求以Binder的方式发送给了AMS。
2. AMS接收到启动请求后，交付ActivityStarter处理Intent和Flag等信息，然后再交给ActivityStackSupervisior/ActivityStack  
   处理Activity进栈相关流程。同时以Socket方式请求Zygote进程fork新进程。
3. Zygote接收到新进程创建请求后fork出新进程。
4. 在新进程里创建ActivityThread对象，新创建的进程就是应用的主线程，在主线程里开启Looper消息循环，开始处理创建Activity。
5. ActivityThread利用ClassLoader去加载Activity、创建Activity实例，并回调Activity的onCreate()方法。这样便完成了Activity的启动。

**BroadcastReceiver与LocalBroadcastReceiver有什么区别？**

* BroadcastReceiver 是跨应用广播，利用Binder机制实现。
* LocalBroadcastReceiver 是应用内广播，利用Handler实现，利用了IntentFilter的match功能，提供消息的发布与接收功能，实现应用内通信，效率比较高。

**Android Handler机制是做什么的，原理了解吗？**

Android消息循环流程图如下所示：



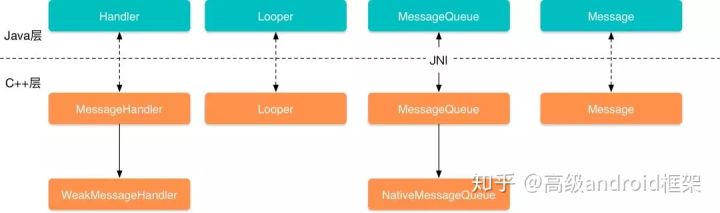
主要涉及的角色如下所示：

* Message：消息，分为硬件产生的消息（例如：按钮、触摸）和软件产生的消息。
* MessageQueue：消息队列，主要用来向消息池添加消息和取走消息。
* Looper：消息循环器，主要用来把消息分发给相应的处理者。
* Handler：消息处理器，主要向消息队列发送各种消息以及处理各种消息。

整个消息的循环流程还是比较清晰的，具体说来：

1. Handler通过sendMessage()发送消息Message到消息队列MessageQueue。
2. Looper通过loop()不断提取触发条件的Message，并将Message交给对应的target handler来处理。
3. target handler调用自身的handleMessage()方法来处理Message。

事实上，在整个消息循环的流程中，并不只有Java层参与，很多重要的工作都是在C++层来完成的。我们来看下这些类的调用关系。



注：虚线表示关联关系，实线表示调用关系。

在这些类中MessageQueue是Java层与C++层维系的桥梁，MessageQueue与Looper相关功能都通过MessageQueue的Native方法来完成，而其他虚线连接的类只有关联关系，并没有直接调用的关系，它们发生关联的桥梁是MessageQueue。

**Android Binder机制是做什么的，为什么选用Binder，原理了解吗？**

Android Binder是用来做进程通信的，Android的各个应用以及系统服务都运行在独立的进程中，它们的通信都依赖于Binder。

为什么选用Binder，在讨论这个问题之前，我们知道Android也是基于Linux内核，Linux现有的进程通信手段有以下几种：

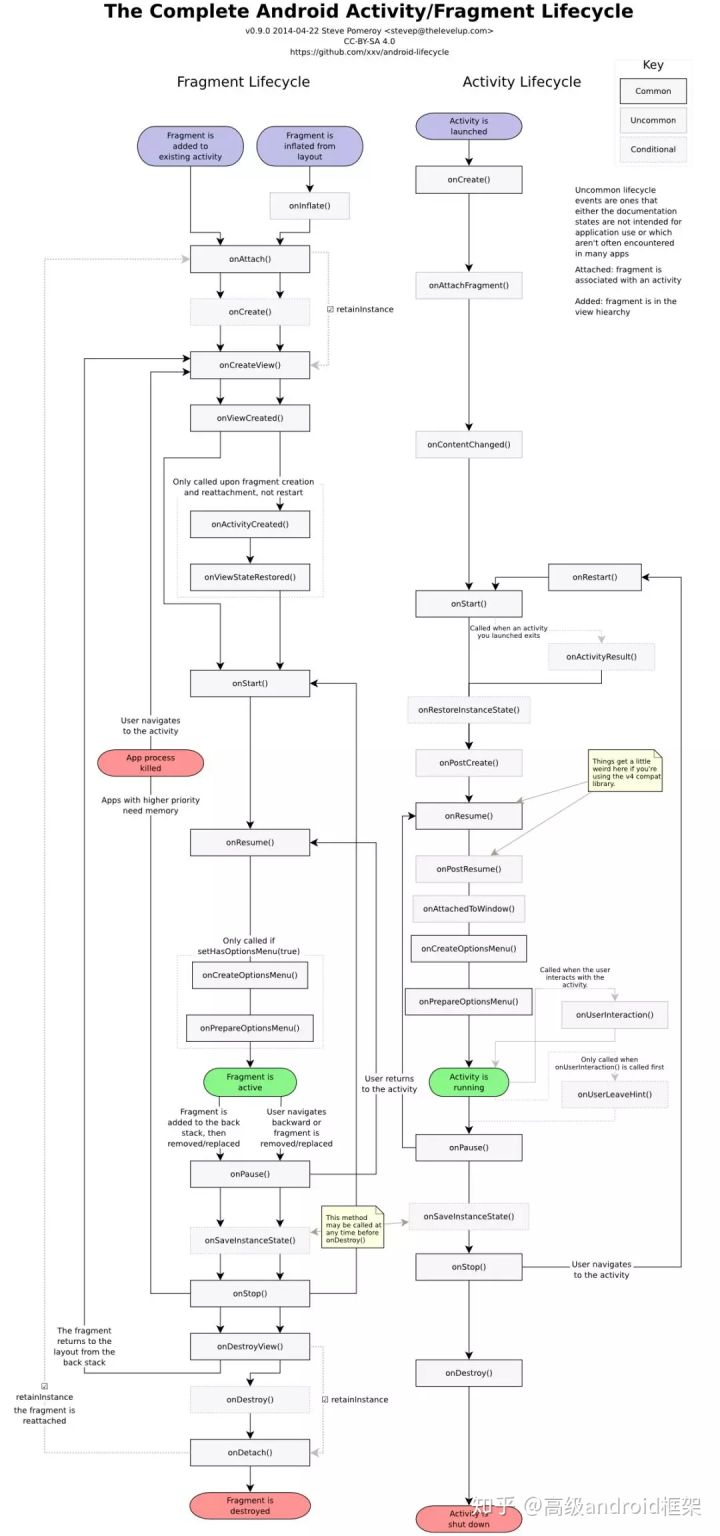
1. 管道：在创建时分配一个page大小的内存，缓存区大小比较有限；
2. 消息队列：信息复制两次，额外的CPU消耗；不合适频繁或信息量大的通信；
3. 共享内存：无须复制，共享缓冲区直接付附加到进程虚拟地址空间，速度快；但进程间的同步问题操作系统无法实现，必须各进程利用同步工具解决；
4. 套接字：作为更通用的接口，传输效率低，主要用于不通机器或跨网络的通信；
5. 信号量：常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。6. 信号: 不适用于信息交换，更适用于进程中断控制，比如非法内存访问，杀死某个进程等；

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：

* 高性能：从数据拷贝次数来看Binder只需要进行一次内存拷贝，而管道、消息队列、Socket都需要两次，共享内存不需要拷贝，Binder的性能仅次于共享内存。
* 稳定性：上面说到共享内存的性能优于Binder，那为什么不适用共享内存呢，因为共享内存需要处理并发同步问题，控制负责，容易出现死锁和资源竞争，稳定性较差。而Binder基于C/S架构，客户端与服务端彼此独立，稳定性较好。
* 安全性：我们知道Android为每个应用分配了UID，用来作为鉴别进程的重要标志，Android内部也依赖这个UID进行权限管理，包括6.0以前的固定权限和6.0以后的动态权限，传荣IPC只能由用户在数据包里填入UID/PID，这个标记完全  
  是在用户空间控制的，没有放在内核空间，因此有被恶意篡改的可能，因此Binder的安全性更高。

**描述一下Activity的生命周期，这些生命周期是如何管理的？**

Activity与Fragment生命周期如下所示：

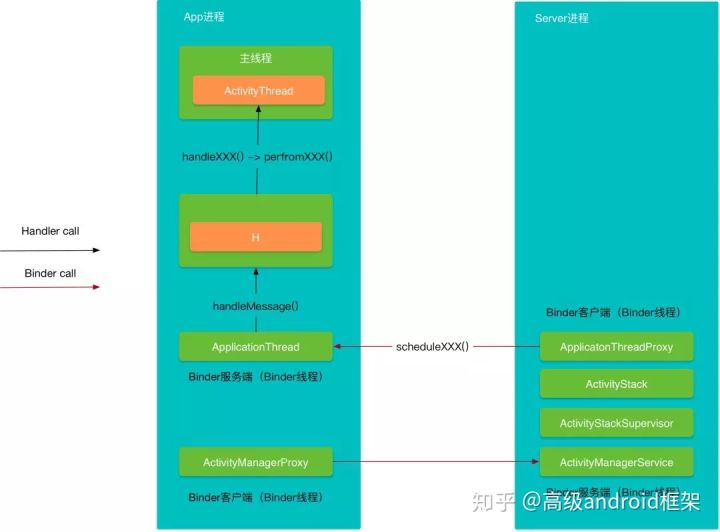


读者可以从上图看出，Activity有很多种状态，状态之间的变化也比较复杂，在众多状态中，只有三种是常驻状态：

* Resumed（运行状态）：Activity处于前台，用户可以与其交互。
* Paused（暂停状态）：Activity被其他Activity部分遮挡，无法接受用户的输入。
* Stopped（停止状态）：Activity被完全隐藏，对用户不可见，进入后台。

其他的状态都是中间状态。

我们再来看看生命周期变化时的整个调度流程，生命周期调度流程图如下所示：



所以你可以看到，整个流程是这样的：

1. 比方说我们点击跳转一个新Activity，这个时候Activity会入栈，同时它的生命周期也会从onCreate()到onResume()开始变换，这个过程是在ActivityStack里完成的，ActivityStack  
   是运行在Server进程里的，这个时候Server进程就通过ApplicationThread的代理对象ApplicationThreadProxy向运行在app进程ApplicationThread发起操作请求。
2. ApplicationThread接收到操作请求后，因为它是运行在app进程里的其他线程里，所以ApplicationThread需要通过Handler向主线程ActivityThread发送操作消息。
3. 主线程接收到ApplicationThread发出的消息后，调用主线程ActivityThread执行响应的操作，并回调Activity相应的周期方法。

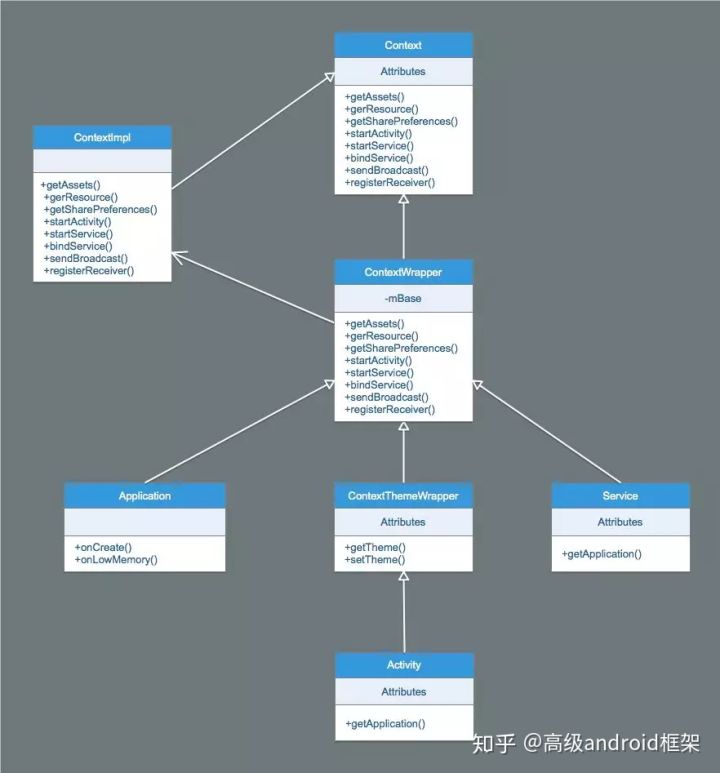
注：这里提到了主线程ActivityThread，更准确来说ActivityThread不是线程，因为它没有继承Thread类或者实现Runnable接口，它是运行在应用主线程里的对象，那么应用的主线程  
到底是什么呢？从本质上来讲启动启动时创建的进程就是主线程，线程和进程处理是否共享资源外，没有其他的区别，对于Linux来说，它们都只是一个struct结构体。

**Activity的通信方式有哪些？**

* startActivityForResult
* EventBus
* LocalBroadcastReceiver

**Android应用里有几种Context对象?**

Context类图如下所示：



可以发现Context是个抽象类，它的具体实现类是ContextImpl，ContextWrapper是个包装类，内部的成员变量mBase指向的也是个ContextImpl对象，ContextImpl完成了  
实际的功能，Activity、Service与Application都直接或者间接的继承ContextWrapper。

**描述一下进程和Application的生命周期？**

一个安装的应用对应一个LoadedApk对象，对应一个Application对象，对于四大组件，Application的创建和获取方式也是不尽相同的，具体说来：

* Activity：通过LoadedApk的makeApplication()方法创建。
* Service：通过LoadedApk的makeApplication()方法创建。
* 静态广播：通过其回调方法onReceive()方法的第一个参数指向Application。
* ContentProvider：无法获取Application，因此此时Application不一定已经初始化。

**Android哪些情况会导致内存泄漏，如何分析内存泄漏？**

常见的产生内存泄漏的情况如下所示：

* 持有静态的Context（Activity）引用。
* 持有静态的View引用，
* 内部类&匿名内部类实例无法释放（有延迟时间等等），而内部类又持有外部类的强引用，导致外部类无法释放，这种匿名内部类常见于监听器、Handler、Thread、TimerTask
* 资源使用完成后没有关闭，例如：BraodcastReceiver，ContentObserver，File，Cursor，Stream，Bitmap。
* 不正确的单例模式，比如单例持有Activity。
* 集合类内存泄漏，如果一个集合类是静态的（缓存HashMap），只有添加方法，没有对应的删除方法，会导致引用无法被释放，引发内存泄漏。
* 错误的覆写了finalize()方法，finalize()方法执行执行不确定，可能会导致引用无法被释放。

查找内存泄漏可以使用Android Profiler工具或者利用LeakCanary工具。

**Android有哪几种进程，是如何管理的？**

Android的进程主要分为以下几种：

**前台进程**

用户当前操作所必需的进程。如果一个进程满足以下任一条件，即视为前台进程：

* 托管用户正在交互的 Activity（已调用 Activity 的 onResume() 方法）
* 托管某个 Service，后者绑定到用户正在交互的 Activity
* 托管正在“前台”运行的 Service（服务已调用 startForeground()）
* 托管正执行一个生命周期回调的 Service（onCreate()、onStart() 或 onDestroy()）
* 托管正执行其 onReceive() 方法的 BroadcastReceiver

通常，在任意给定时间前台进程都为数不多。只有在内存不足以支持它们同时继续运行这一万不得已的情况下，系统才会终止它们。 此时，设备往往已达到内存分页状态，因此需要终止一些前台进程来确保用户界面正常响应。

**可见进程**

没有任何前台组件、但仍会影响用户在屏幕上所见内容的进程。 如果一个进程满足以下任一条件，即视为可见进程：

* 托管不在前台、但仍对用户可见的 Activity（已调用其 onPause() 方法）。例如，如果前台 Activity 启动了一个对话框，允许在其后显示上一 Activity，则有可能会发生这种情况。
* 托管绑定到可见（或前台）Activity 的 Service。

可见进程被视为是极其重要的进程，除非为了维持所有前台进程同时运行而必须终止，否则系统不会终止这些进程。

**服务进程**

正在运行已使用 startService() 方法启动的服务且不属于上述两个更高类别进程的进程。尽管服务进程与用户所见内容没有直接关联，但是它们通常在执行一些用户关  
心的操作（例如，在后台播放音乐或从网络下载数据）。因此，除非内存不足以维持所有前台进程和可见进程同时运行，否则系统会让服务进程保持运行状态。

**后台进程**

包含目前对用户不可见的 Activity 的进程（已调用 Activity 的 onStop() 方法）。这些进程对用户体验没有直接影响，系统可能随时终止它们，以回收内存供前台进程、可见进程或服务进程使用。 通常会有很多后台进程在运行，因此它们会保存在 LRU （最近最少使用）列表中，以确保包含用户最近查看的 Activity 的进程最后一个被终止。如果某个 Activity 正确实现了生命周期方法，并保存了其当前状态，则终止其进程不会对用户体验产生明显影响，因为当用户导航回该 Activity 时，Activity 会恢复其所有可见状态。

**空进程**

不含任何活动应用组件的进程。保留这种进程的的唯一目的是用作缓存，以缩短下次在其中运行组件所需的启动时间。 为使总体系统资源在进程缓存和底层内核缓存之间保持平衡，系统往往会终止这些进程。

ActivityManagerService负责根据各种策略算法计算进程的adj值，然后交由系统内核进行进程的管理。

**SharePreference性能优化，可以做进程同步吗？**

在Android中, SharePreferences是一个轻量级的存储类，特别适合用于保存软件配置参数。使用SharedPreferences保存数据，其背后是用xml文件存放数据，文件  
存放在/data/data/ < package name > /shared\_prefs目录下.

之所以说SharedPreference是一种轻量级的存储方式，是因为它在创建的时候会把整个文件全部加载进内存，如果SharedPreference文件比较大，会带来以下问题：

1. 第一次从sp中获取值的时候，有可能阻塞主线程，使界面卡顿、掉帧。
2. 解析sp的时候会产生大量的临时对象，导致频繁GC，引起界面卡顿。
3. 这些key和value会永远存在于内存之中，占用大量内存。

优化建议

1. 不要存放大的key和value，会引起界面卡、频繁GC、占用内存等等。
2. 毫不相关的配置项就不要放在在一起，文件越大读取越慢。
3. 读取频繁的key和不易变动的key尽量不要放在一起，影响速度，如果整个文件很小，那么忽略吧，为了这点性能添加维护成本得不偿失。
4. 不要乱edit和apply，尽量批量修改一次提交，多次apply会阻塞主线程。
5. 尽量不要存放JSON和HTML，这种场景请直接使用JSON。
6. SharedPreference无法进行跨进程通信，MODE\_MULTI\_PROCESS只是保证了在API 11以前的系统上，如果sp已经被读取进内存，再次获取这个SharedPreference的时候，如果有这个flag，会重新读一遍文件，仅此而已。

**如何做SQLite升级？**

数据库升级增加表和删除表都不涉及数据迁移，但是修改表涉及到对原有数据进行迁移。升级的方法如下所示：

1. 将现有表命名为临时表。
2. 创建新表。
3. 将临时表的数据导入新表。
4. 删除临时表。

如果是跨版本数据库升级，可以由两种方式，如下所示：

1. 逐级升级，确定相邻版本与现在版本的差别，V1升级到V2,V2升级到V3，依次类推。
2. 跨级升级，确定每个版本与现在数据库的差别，为每个case编写专门升级大代码。

**进程保护如何做，如何唤醒其他进程？**

进程保活主要有两个思路：

1. 提升进程的优先级，降低进程被杀死的概率。
2. 拉活已经被杀死的进程。

如何提升优先级，如下所示：

监控手机锁屏事件，在屏幕锁屏时启动一个像素的Activity，在用户解锁时将Activity销毁掉，前台Activity可以将进程变成前台进程，优先级升级到最高。

如果拉活

利用广播拉活Activity。

**理解序列化吗，Android为什么引入Parcelable？**

所谓序列化就是将对象变成二进制流，便于存储和传输。

* Serializable是java实现的一套序列化方式，可能会触发频繁的IO操作，效率比较低，适合将对象存储到磁盘上的情况。
* Parcelable是Android提供一套序列化机制，它将序列化后的字节流写入到一个共性内存中，其他对象可以从这块共享内存中读出字节流，并反序列化成对象。因此效率比较高，适合在对象间或者进程间传递信息。

**如何计算一个Bitmap占用内存的大小，怎么保证加载Bitmap不产生内存溢出？**

Bitamp 占用内存大小 = 宽度像素 x （inTargetDensity / inDensity） x 高度像素 x （inTargetDensity / inDensity）x 一个像素所占的内存.

注：这里inDensity表示目标图片的dpi（放在哪个资源文件夹下），inTargetDensity表示目标屏幕的dpi，所以你可以发现inDensity和inTargetDensity会对Bitmap的宽高  
进行拉伸，进而改变Bitmap占用内存的大小。

在Bitmap里有两个获取内存占用大小的方法。

* getByteCount()：API12 加入，代表存储 Bitmap 的像素需要的最少内存。
* getAllocationByteCount()：API19 加入，代表在内存中为 Bitmap 分配的内存大小，代替了 getByteCount() 方法。

在不复用 Bitmap 时，getByteCount() 和 getAllocationByteCount 返回的结果是一样的。在通过复用 Bitmap 来解码图片时，那么 getByteCount() 表示新解码图片占用内存的大  
小，getAllocationByteCount() 表示被复用 Bitmap真实占用的内存大小（即 mBuffer 的长度）。

为了保证在加载Bitmap的时候不产生内存溢出，可以受用BitmapFactory进行图片压缩，主要有以下几个参数：

* BitmapFactory.Options.inPreferredConfig：将ARGB\_8888改为RGB\_565，改变编码方式，节约内存。
* BitmapFactory.Options.inSampleSize：缩放比例，可以参考Luban那个库，根据图片宽高计算出合适的缩放比例。
* BitmapFactory.Options.inPurgeable：让系统可以内存不足时回收内存。

**Android如何在不压缩的情况下加载高清大图？**

使用BitmapRegionDecoder进行布局加载。

**Android里的内存缓存和磁盘缓存是怎么实现的?**

内存缓存基于LruCache实现，磁盘缓存基于DiskLruCache实现。这两个类都基于Lru算法和LinkedHashMap来实现。

LRU算法可以用一句话来描述，如下所示：

LRU是Least Recently Used的缩写，最近最久未使用算法，从它的名字就可以看出，它的核心原则是如果一个数据在最近一段时间没有使用到，那么它在将来被  
访问到的可能性也很小，则这类数据项会被优先淘汰掉。

LruCache的原理是利用LinkedHashMap持有对象的强引用，按照Lru算法进行对象淘汰。具体说来假设我们从表尾访问数据，在表头删除数据，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。

**为什么会选择LinkedHashMap呢？**

这跟LinkedHashMap的特性有关，LinkedHashMap的构造函数里有个布尔参数accessOrder，当它为true时，LinkedHashMap会以访问顺序为序排列元素，否则以插入顺序为序排序元素。

DiskLruCache与LruCache原理相似，只是多了一个journal文件来做磁盘文件的管理和迎神，如下所示：

libcore.io.DiskLruCache

1

1

1

DIRTY 1517126350519

CLEAN 1517126350519 5325928

REMOVE 1517126350519

注：这里的缓存目录是应用的缓存目录/data/data/pckagename/cache，未root的手机可以通过以下命令进入到该目录中或者将该目录整体拷贝出来：

//进入/data/data/pckagename/cache目录

adb shell

run-as com.your.packagename

cp /data/data/com.your.packagename/

//将/data/data/pckagename目录拷贝出来

adb backup -noapk com.your.packagename

我们来分析下这个文件的内容：

* 第一行：libcore.io.DiskLruCache，固定字符串。
* 第二行：1，DiskLruCache源码版本号。
* 第三行：1，App的版本号，通过open()方法传入进去的。
* 第四行：1，每个key对应几个文件，一般为1.
* 第五行：空行
* 第六行及后续行：缓存操作记录。

第六行及后续行表示缓存操作记录，关于操作记录，我们需要了解以下三点：

1. DIRTY 表示一个entry正在被写入。写入分两种情况，如果成功会紧接着写入一行CLEAN的记录；如果失败，会增加一行REMOVE记录。注意单独只有DIRTY状态的记录是非法的。
2. 当手动调用remove(key)方法的时候也会写入一条REMOVE记录。
3. READ就是说明有一次读取的记录。
4. CLEAN的后面还记录了文件的长度，注意可能会一个key对应多个文件，那么就会有多个数字。

**PathClassLoader与DexClassLoader有什么区别？**

* PathClassLoader：只能加载已经安装到Android系统的APK文件，即/data/app目录，Android默认的类加载器。
* DexClassLoader：可以加载任意目录下的dex、jar、apk、zip文件。

**WebView优化了解吗，如何提高WebView的加载速度？**

为什么WebView加载会慢呢？

这是因为在客户端中，加载H5页面之前，需要先初始化WebView，在WebView完全初始化完成之前，后续的界面加载过程都是被阻塞的。

优化手段围绕着以下两个点进行：

1. 预加载WebView。
2. 加载WebView的同时，请求H5页面数据。

因此常见的方法是：

1. 全局WebView。
2. 客户端代理页面请求。WebView初始化完成后向客户端请求数据。
3. asset存放离线包。

除此之外还有一些其他的优化手段：

* 脚本执行慢，可以让脚本最后运行，不阻塞页面解析。
* DNS与链接慢，可以让客户端复用使用的域名与链接。
* React框架代码执行慢，可以将这部分代码拆分出来，提前进行解析。

**Java和JS的相互调用怎么实现，有做过什么优化吗？**

jockeyjs：[https://github.com/tcoulter/jockeyjs](https://link.zhihu.com/?target=https%3A//github.com/tcoulter/jockeyjs)

对协议进行统一的封装和处理。

**JNI了解吗，Java与C++如何相互调用？**

Java调用C++

1. 在Java中声明Native方法（即需要调用的本地方法）
2. 编译上述 Java源文件javac（得到 .class文件）  
   3。 通过 javah 命令导出JNI的头文件（.h文件）
3. 使用 Java需要交互的本地代码 实现在 Java中声明的Native方法
4. 编译.so库文件
5. 通过Java命令执行 Java程序，最终实现Java调用本地代码

C++调用Java

1. 从classpath路径下搜索ClassMethod这个类，并返回该类的Class对象。
2. 获取类的默认构造方法ID。
3. 查找实例方法的ID。
4. 创建该类的实例。
5. 调用对象的实例方法。

JNIEXPORT void JNICALL Java\_com\_study\_jnilearn\_AccessMethod\_callJavaInstaceMethod

(JNIEnv \*env, jclass cls)

{

jclass clazz = NULL;

jobject jobj = NULL;

jmethodID mid\_construct = NULL;

jmethodID mid\_instance = NULL;

jstring str\_arg = NULL;

// 1、从classpath路径下搜索ClassMethod这个类，并返回该类的Class对象

clazz = (\*env)->FindClass(env, "com/study/jnilearn/ClassMethod");

if (clazz == NULL) {

printf("找不到'com.study.jnilearn.ClassMethod'这个类");

return;

}

// 2、获取类的默认构造方法ID

mid\_construct = (\*env)->GetMethodID(env,clazz, "<init>","()V");

if (mid\_construct == NULL) {

printf("找不到默认的构造方法");

return;

}

// 3、查找实例方法的ID

mid\_instance = (\*env)->GetMethodID(env, clazz, "callInstanceMethod", "(Ljava/lang/String;I)V");

if (mid\_instance == NULL) {

return;

}

// 4、创建该类的实例

jobj = (\*env)->NewObject(env,clazz,mid\_construct);

if (jobj == NULL) {

printf("在com.study.jnilearn.ClassMethod类中找不到callInstanceMethod方法");

return;

}

// 5、调用对象的实例方法

str\_arg = (\*env)->NewStringUTF(env,"我是实例方法");

(\*env)->CallVoidMethod(env,jobj,mid\_instance,str\_arg,200);

// 删除局部引用

(\*env)->DeleteLocalRef(env,clazz);

(\*env)->DeleteLocalRef(env,jobj);

(\*env)->DeleteLocalRef(env,str\_arg);

}

**了解插件化和热修复吗，它们有什么区别，理解它们的原理吗？**

* 插件化：插件化是体现在功能拆分方面的，它将某个功能独立提取出来，独立开发，独立测试，再插入到主应用中。依次来较少主应用的规模。
* 热修复：热修复是体现在bug修复方面的，它实现的是不需要重新发版和重新安装，就可以去修复已知的bug。

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

**如何做性能优化？**

1. 节制的使用Service，当启动一个Service时，系统总是倾向于保留这个Service依赖的进程，这样会造成系统资源的浪费，可以使用IntentService，执行完成任务后会自动停止。
2. 当界面不可见时释放内存，可以重写Activity的onTrimMemory()方法，然后监听TRIM\_MEMORY\_UI\_HIDDEN这个级别，这个级别说明用户离开了页面，可以考虑释放内存和资源。
3. 避免在Bitmap浪费过多的内存，使用压缩过的图片，也可以使用Fresco等库来优化对Bitmap显示的管理。
4. 使用优化过的数据集合SparseArray代替HashMap，HashMap为每个键值都提供一个对象入口，使用SparseArray可以免去基本对象类型转换为引用数据类想的时间。

**如果防止过度绘制，如何做布局优化？**

1. 使用include复用布局文件。
2. 使用merge标签避免嵌套布局。
3. 使用stub标签仅在需要的时候在展示出来。

**如何提交代码质量？**

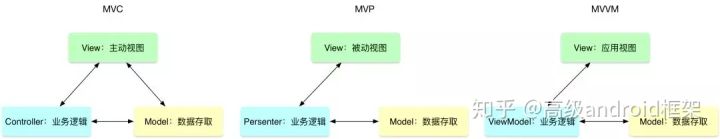
1. 避免创建不必要的对象，尽可能避免频繁的创建临时对象，例如在for循环内，减少GC的次数。
2. 尽量使用基本数据类型代替引用数据类型。
3. 静态方法调用效率高于动态方法，也可以避免创建额外对象。
4. 对于基本数据类型和String类型的常量要使用static final修饰，这样常量会在dex文件的初始化器中进行初始化，使用的时候可以直接使用。
5. 多使用系统API，例如数组拷贝System.arrayCopy()方法，要比我们用for循环效率快9倍以上，因为系统API很多都是通过底层的汇编模式执行的，效率比较高。

**有没有遇到64k问题，为什么，如何解决？**

* 在DEX文件中，method、field、class等的个数使用short类型来做索引，即两个字节（65535），method、field、class等均有此限制。
* APK在安装过程中会调用dexopt将DEX文件优化成ODEX文件，dexopt使用LinearAlloc来存储应用信息，关于LinearAlloc缓冲区大小，不同的版本经历了4M/8M/16M的限制，超出  
  缓冲区时就会抛出INSTALL\_FAILED\_DEXOPT错误。

解决方案是Google的MultiDex方案，具体参见：配置方法数超过 64K 的应用。

**MVC、MVP与MVVM之间的对比分析？**



* MVC：PC时代就有的架构方案，在Android上也是最早的方案，Activity/Fragment这些上帝角色既承担了V的角色，也承担了C的角色，小项目开发起来十分顺手，大项目就会遇到  
  耦合过重，Activity/Fragment类过大等问题。
* MVP：为了解决MVC耦合过重的问题，MVP的核心思想就是提供一个Presenter将视图逻辑I和业务逻辑相分离，达到解耦的目的。
* MVVM：使用ViewModel代替Presenter，实现数据与View的双向绑定，这套框架最早使用的data-binding将数据绑定到xml里，这么做在大规模应用的时候是不行的，不过数据绑定是
* 一个很有用的概念，后续Google又推出了ViewModel组件与LiveData组件。ViewModel组件规范了ViewModel所处的地位、生命周期、生产方式以及一个Activity下多个Fragment共享ViewModel数据的问题。LiveData组件则提供了在Java层面View订阅ViewModel数据源的实现方案。

本文仅是作者开源项目的一部分，整体包含：

* Java面试题集(含答案)
* Android面试题集(含答案)
* Android开源库面试题集(含答案)
* Android网络编程面试题集(含答案)
* 数据结构与算法面试题集(含答案)
* HR面试题集(含答案)

发布于 2018-06-22