Java 语言进阶与 Android 技术相关面试题

Android 专题讲解

专题讲解——Activity 组件

在 Activity 的生命周期中，可以将 Activity 表现为 3 种状态：

激活态：当 Acitivity 位于屏幕前端，并可以获得用户焦点、收用户输入时，这种状态称为激活态，也可以称为运行态；

暂停态：当 Activity 在运行时被另一个 Activity 所遮挡并获取焦点，此时 Activity 仍然可见，也就是说另一个 Activity 是部分遮挡的，或者另一个 Activity 是透明的或半透明的，此时 Activity 处于暂停状态；

停止态：当 Activity 被另一个 Activity 完全遮挡不可见时处于停止状态，这个 Activity 仍然存在，它保留在内存中并保持所有状态和成员信息，但是当该设备内存不足时，该 Activity 可能会被系统杀掉以释放其占用的内存空间，当再次打开该 Activity 时，它会被重新创建；

Activity **生命周期中的 7 个方法：**

1. onCreate( )：当 Activity 被创建时调用；

2 onStart( )：当 Activity 被创建后将可见时调用；

3 onResume( )：（继续开始）当 Activity 位于设备最前端，对用户可见时调用；

4 onPause( )：（暂停）当另一个 Activity 遮挡当前 Activity，当前 Activity 被切换到后台时调用；

5 onRestart( )：（重新启动）当另一个 Activity 执行完 onStop（）方法，又被用户打开时调用；

6　onStop( )：如果另一个 Activity 完全遮挡了当前 Activity 时，该方法被调用；

7.　onDestory( )：当Activity 被销毁时调用；

Activity **的四种启动模式：**standard、singleTop、singleTask 和 singleInstance，他们是在配置文件中通过 android：LauchMode 属性配置；

1. standard：默认的启动模式，每次启动会在任务栈中新建一个启动的Activity 的实例；

2. SingleTop：如果要启动的 Activity 实例已位于栈顶，则不会重新创建该Activity 的实例，否则会产生一个新的运行实例；

3. SingleTask：如果栈中有该 Activity 实例，则直接启动，中间的 Activity 实例将会被关闭，关闭的顺序与启动的顺序相同；

4. SingleInstance：该启动模式会在启动一个 Activity 时，启动一个新的任务栈，将该 Activity 实例放置在这个任务栈中，并且该任务栈中不会再保存其他的Activity 实例；

Activity **任务栈：**即存放 Activity 任务的栈，每打开一个 Activity 时就会往Activity 栈中压入一个 Activity 任务，每当销毁一个 Activity 的时候，就会从 Activity 任务栈中弹出一个 Activity 任务，由于安卓手机的限制，只能从手机屏幕获取当前一个 Activity 的焦点，即栈顶元素（最上面的 Activity），其余的 Activity 会暂居后台等待系统的调用；

关于任务栈的更多概念：

当程序打开时就创建了一个任务栈，用于存储当前程序的 Activity，当前程序（包括被当前程序所调用的）所有的 Activity 都属于一个任务栈；

一个任务栈包含了一个 Activity 的集合，可以有序的选择哪个 Activity 和用户进行交互，只有任务栈顶的 Activity 才可以与用户进行交互；

任务栈可以移动到后台，并保留每一个 Activity 的状态，并且有序的给用户

列出他们的任务，而且还不会丢失他们的状态信息；

退出应用程序时，当把所有的任务栈中所有的 Activity 清除出栈时，任务栈会被销毁，程序退出；

默认 Acctivity 启动方式的缺点：

每开启一次页面都会在任务栈中添加一个 Activity，而只有任务栈中的 Activity 全部清除出栈时，任务栈被销毁，程序才会退出，这样就造成了用户体验差，需要点击多次返回才可以把程序退出了。

每开启一次页面都会在任务栈中添加一个 Activity 还会造成数据冗余重复数据太多，会导致内存溢出的问题(OOM)。

专题讲解——Service 组件

Service 组件主要用于处理不干扰用户交互的后台操作，如更新应用、播放音乐等，Service 组件既不会开启新的进程，也不会开启新的线程，它运行在应用程序的主线程中，Service 实现的逻辑代码不能阻塞整个应用程序的运行，否则会引起应用程序抛出 ANR 异常。

Service 组件常被用于实现以下两种功能（分别对应两种启动模式）：

使用 startService（）方法启动 Service 组件，运行在系统的后台在不需要用户交互的前提下，实现某些功能；

使用 bindService（）方法启动 Service 组件，启动者与服务者之间建立“绑定关系”，应用程序可以与 Service 组件交互；

Service 中常用的方法：

1. onBind(Intent intent)：抽象方法绑定模式时执行的方法，通过 Ibinder 对

象访问 Service；

2　onCreate（）：Service 组件创建时执行；

3　onDestroy（）：Service 组件销毁时执行；

4　onStartCommand(Intent intent ， int flags ,int startId)：开始模式时执行的方法，每次执行 startService （）方法，该方法都会执行；

5　onUnBind( )：解除绑定时执行；

6　stop Self （）：停止Service 组件；

Service **组件的生命周期：**Service 有两种启动模式，startService 方法或 bindServce 方法，启动方法决定了 Service 组件的生命周期和他所执行的生命周期方法：

通过 statrtService 方法启动 Service 组件，如果已经存在目标组件，则会直接调用 onStartCommand 方法，否则回调哦那 Create （）方法，创建 Service 对象，启动后的 Service 组件会一直运行在后台，与“启动者”无关联，其状态不会受“启动者”的影响，即使“启动者被销毁，Service 组件还会继续运行，直到调用 StopService （）方法，或执行 stopSelf （）方法；

通过bindService （）方法，启动 Service 组件，如果组件对象已经存在，则与之绑定，否则创建性的 Service 对象后再与之绑定，这种启动方法把 Service 组件与“启动者“绑定，Service 返回 Ibinder 对象，启动者借助 ServiceConnection 对象实例实现与之交互，这种启动方式会将 Service 组件与” 启动者“相关联，Service 的状态会受到启动者的影响；

Service **的启动模式详解**

启动和停止 Service 组件的方法都位于 context 类中，再 Activity 中可以直接调用；

startService（Intent service）：以 Start 模式启动 Service 组件，参数 service
是目标组件；

stopService(Intent service)：停止 start 模式启动的 Service 组件，参数 service 是目标组件；

bindService（Intent service ,serviceConnection conn ,int flags）：以 Bind 模式启动 Service 组件，参数 service 是目标组件，conn 是与目标链接的对象，不可为 NULL，flags 是绑定模式；

unbindService（ServiceConnection conn）：解除绑定模式启动的 Service 组件，conn 是绑定时的链接对象；

专题讲解——BoradcastReceiver 组件

广播机制是安卓系统中的通信机制，可以实现在应用程序内部或应用程序之间传递消息的作用，发出广播（或称广播）和接收广播是两个不同的动作，Android 系统主动发出的广播称为系统广播，应用程序发出广播称为自定义广播，广播实质上是一个 Intent 对象，接收广播实质上是接收 Intent 对象。

接收广播需要自定义广播接收器类，继承自 BoradcastReceiver 类，BoradcastReceiver 的生命周期很短，BoradcastReceiver 对象在执行完 onReceiver
（）方法后就会被销毁，在 onReceiver 方法中不能执行比较耗时的操作，也不能在 onReceiver 方法中开启子线程，因为当父线程被杀死后，他的子线程也会被杀死，从而导致线程不安全。

广播分为有序广播和无序广播

无序广播：通过 sendBoradcast（）方法发送的广播，普通广播对于接收器来说是无序的，没有优先级，每个接收器都无需等待即可以接收到广播，接收器之间相互是没有影响的，这种广播无法被终止，即无法阻止其他接收器的

接

淘宝关注【闵课通商学院】，免费领取200G大礼包　淘宝搜《闵课通商学院》，小白轻松拿高薪offer

收动作。

有序广播：通过 sendOrderedBroadcast()方法发送的广播，有序广播对于接收器来说是有序的，有优先级区分的，接收者有权终止广播，使后续接收者无法接收到广播，并且接收者接收到广播后可以对结果对象进行操作。

注册广播接收器的方式：配置文件静态注册和在代码中动态注册。

配置文件中静态注册：BoradcastReceiver 组件使用<receiver/>标签配置，写在 application 标签内部，receiver 标签内的<intent-filter>标签用于设置广播接收器能够响应的广播动作。

使用代码动态注册：使用context 中的registerReceiver（BoradcastReceiver receiver ，IntentFileter filter）方法，参数receiver 是需要注册的广播接收器， 参数filter 是用于选择相匹配的广播接收器，使用这种方法注册广播接收器，最后必须解除注册，解除注册使用context 的unregisterReceiver(BoradcastReceiverreceiver)方法，参数receiver 为要解除注册的广播接收器。

配置文件静态注册和在代码中动态注册两种方式的区别

静态注册（也称常驻型注册）：这种广播接收器需要在 Androidmainfest.xml 中进行注册，这种方式注册的广播，不受页面生命周期的影响，即使退出了页面，也可以收到广播，这种广播一般用于开机自启动，由于这种注册方式的广播是常驻型广播，所以会占用 CPU 资源。

动态注册：是在代码中注册的，这种注册方式也叫非常驻型注册，会受到页面生命周期的影响，退出页面后就不会收到广播，我们通常将其运用在更新 UI 方面，这种注册方式优先级较高，最后需要解绑（解除注册），否则会导致内存泄漏）。

要点：使用广播机制更新 UI 的思路，在需要更新的 Activity 内定义一个继承自 BroadcastReceiver 的

内部类，在 Activty 中动态注册该广播接收器，通过广播接收器的的

onReceiver（）方法来更新 UI。

专题讲解——ContentProvider(内容提供者)组件

Android 系统将所有的数据都规定为私有，无法直接访问应用程序之外的数据，如果需要访问其他程序的数据或向其他程序提供数据，需要用到 ContentProvider 抽象类，该抽象类为存储和获取数据提供了统一的接口，ContentProvider 对数据进行了封装，也在使用时不必关心数据的存储方式。

URI（**统一资源标识符**）：一个Uri 一般有三部分组成，分别是访问协议、唯一性标识字符串（或访问权限字符串）、资源部份。

ContentProvider **实现对外部程序数据操作的思路：**

在一个程序中自定义 ContentProvider 类（继承自 ContentProvider），并实现其中的抽象方法，在配置文件中使用<provider/>标签对其进行注册（<provider/>标签中有 3 个很重要的属性：name 为指明自定义的 ContentProvider 的全路径，authories 为访问标识，exported 是否允许外部应用访问），在另一个需要操作该程序数据的程序中，通过 context 类的 getContentResolver 方法获得 ContentResolver 类的实例对象，ContentResolver 类执行增、删、改、查操作的方法与 ContentProvider 类的几乎一致，通过调用ContentResolver 类实现类的方法来实现数据操作（在这些方法中仍然会用到 Uri 参数，其应与被操作数据的所在程序中的定义的 Uri 一致）。

专题讲解——Fragment

Android3.0 引入Fragment 技术，译为“碎片、片段”，在Activity 中可以通

过FragmentManager 来添加、移除和管理所加入的 Fragment，每个 Fragment 都有自己的布局、生命周期、交互事件的处理，由于 Fragment 是嵌入到Activity 中的，所以Fragment 的生命周期又和 Activity 的生命周期有密切的关联。

Fragment **的生命周期的方法：**

1．onAttach()：当Fagment 和Activity 产生关联时被调用； 2．onCreate()： 当 Fragment 被创建时调用； 3．onCreateView（）：创建并返回与 Fragment 相关的view 界面； 4．onViewCreate（）：在onCreateView 执行完后立即执行；

5．onActivityCreated( )：通知Fragment，他所关联的 Activity 已经完成了 onCreate 的调用；

6．onStart( )：让Fragment 准备可以被用户所见，该方法和 Activity 的 onStart（）方法相关联；

7．onResume（）：Fragment 可见，可以和用户交互，该方法和 Activity 的 onResume 方法相关联；

8．onPause（）：当用户离开Fragment 时调用该方法，此操作是由于所在的Activity 被遮挡或者是在 Activity 中的另一个Fragment 操作所引起的；

9．onStop（）：对用户而言，Fragment 不可见时调用该方法，此操作是由于他所在的 Activity 不再可见或者是在 Activity 中的一个Fragment 操作所引起的；

10．onDestroyView（）：ment 清理和它的 view 相关的资源； 11．onDestroy（）：最终清理 Fragment 的状态；

12．onDetach（）：Fragment 与Activity 不再产生关联；

Fragment 加载布局文件是在 onCreateView（）方法中使用 LayoutInflater(布

局加载器)的 inflate( )方法加载布局文件。

Fragment **中传递数据：**当Activity 加载Fragment 的时候，往其内部传递参数，官方推荐使用 Fragment 的setArguments（Bundle bundle）方式传递参数， 具体方法为在 Activity 中，对要传递参数的 Fragment 调用 setArguments(Bundle bundle)方法，在接收参数的 Fragment 中调用 context 的 getArguments()方法获取Bundle 参数；

管理 Fragment：通过调用context 的getFragmentManager( )方法或者 getsupportFragmentManager( )方法（这两个方法需要导入的包不同）来获取 FragmentManager，使用FragmentManager 对象，主要可以调用如下方法：

findFragmentById/findFragmentByTag：获取 Activity 中已经存在的Fragment；

getFragments( )：获取所有加入 Activity 中的 Fragment；

begainTransaction( )：获取一个 FragmentTransaction 对象，用来执行 Fragment 的事物；

popBackStack（）：从 Activity 的后退栈中弹出 Fragment；

addOnBackChagedListerner：注册一个侦听器以监视后退栈的变化； FragmentTransaction：

在Activity 中对Fragment 进行添加、删除、替换以及执行其他的动作将引起Fragment 的变化，叫做一个事务，**事务**是通过 FragmentTransaction 来执行的，可以用 add、remove、replace、show、hide( )等方法构成事务，最后使用 commit 提交事务，参考代码：

```
getSupportFragmentManager( ).beginTransact
ion()
        .add(R.id.main_container,tFragment)
```

```
.add(R.id.main_container,eFragment)
```


淘宝关注【闵课通商学院】，免费领取200G大礼包　淘宝搜《闵课通商学院》，小白轻松拿高薪offer

```
.hide(eFragment).commit( );
```

其中 R.id.main_container 为布局中容纳 Fragment 的 view 的 ID。

专题讲解——ViewPager

ViewPager 是 google SDK 自带的一个附加包（android.support.V4）中的一个类，可视为一个可以实现一种卡片式左右滑动的 View 容器，使用该类类似于ListView，需要用到自定义的适配器 PageAdapter，区别在于每次要获取一个view 的方式，ViewPager 准确的说应该是一个 ViewGroup。

PagerAdapter 是 ViewPager 的支持者，ViewPager 调用它来获取所需显示的页面，而 PagerAdapter 也会在数据变化时，通知 ViewPager，这个类也是FragmentPagerAdapter 和 FragmentStatePagerAdapter 的基类。

FragmentPageAdapter 和 FragmentStatePagerAdapter 的区别

FragmentPageAdapter：和 PagerAdapter 一样，只会缓存当前 Fragment 以及左边一个和右边一个，如果左边或右边不存在 Fragment 则不缓存；

FragmentStatePagerAdapter：当 Fragment 对用户不可见时，整个 Fragment 会被销毁，只会保存 Fragment 的状态，而在页面需要重新显示的时候，会生成新的页面；

综上，FragmentPagerAdapter 适合页面较少的场合，而 FragmentStatePagerAdapter 则适合页面较多或页面的内容非常复杂（需要占用大量内存）的情况。

当实现一个 PagerAdapter 时，需要重写相关方法：

1. getCount( )：获得 viewPager 中有多少个 view ；

2 destroyItem（viewGroup，interesting，object）：移除一个给定位置的页面；

3 instantiateItem（ViewGroup ，int ）：将给定位置的 view 添加到

viewgroup（容器中）,创建并显示出来，返回一个代表新增页面的 Object（key），key 和每个 view 要有一一对应的关系；

4　isviewFromObject( )：判断 instantiateItem（ViewGroup，int ）函数所返回的key 和每一个页面视图是否是代表的同一个视图；

综合使用 ViewPager、Fragment 和 FragmentPagerAdapter：

自定义一个继承 FragmentPagerAdapter 的子类，重写其相关方法，当在实例化该子类时，可以传入 Fragmnt 集合（即要使用到的全部 Fragmnet 的集合），将ViewPager 与该自定义的适配器实例绑定，为 ViewPager 设置 OnPagerListener( )监听事件，重写 OnPagerChangeListener 的 onPageSelected( )方法，实现页面的翻转。

关于 Fragment 中的控件的事件的监听：

在Fragment 中的onActivityCreated(　　)生命周期方法中通过context 的 getActivity（）方法获取到Fragment 类相关联的Activity，并就此Activity 通过 findViewById( )方法来获取相关组件，再为组件添加监听事件。

专题讲解——Android 的事件传递（分发）机制

基础概念：

事件分发的对象：点击事件（Touch 事件），当用户触摸屏幕时（view 或 viewGroup 派生的控件），将产生点击事件（Touch 事件），Touch 事件的相关细节（发生触摸的位置、时间等）被封装程 MotionEvent 对象；

事件的类型：

MotionEvent . ACTION_DOWN　按下view（所有事件的开始）

MotionEvent .ACTION_UP　抬起view（与DOWN 对应）

MotionEvent .ACTION_MOVE　滑动view

MotionEvent .ACTION_CANCEL　结束事件（非人为的原因）

事件列：从手指触摸至离开屏幕，这个过程中产生的一系列事件为事件列，一般情况下，事件列都是以 DOWN 事件开始，UP 事件结束，中间有无数的 MOVE 事件；

事件分发的本质：将点击事件（MotionEvent）传递到某个具体的 View& 处理的整个过程，即事件的传递过程=分发过程；

事件在哪些对象之间传递：Activity、viewGroup、view 等对象；**事件分发过程中协作完成的方法：**

dispatchTouchEvent() 、onInterceptTouchEvent（）和onTouchEvent（）方法；

Android 事件传递机制跟布局的层次密切相关，一个典型的布局包括根节点ViewGroup、子 ViewGroup、以及最末端的 view(如下图)：

在这种结构中最上层的是 RootViewGroup，下层是子 view，当我们点击子 view 的时候，点击事件从上层往下层依次传递（即下沉传递 Activity——ViewGroup——View），传递的过程中调用 dispatchTouchEvent 和 onInterceptTouchEvent 函数，当事件传递到被点击的子 view 之后，停止事件的传递，开始改变方向（即冒泡响应 View ——ViewGroup——Activity），依次向上层响应，响应的过程调用 onTouch 或onTouchEvent 方法。

传递过程中的协作方法：

public boolean dispatchTouchEvent（MotionEvent ev）：事件分发处理函

数，通常会在 Activity 层根据 UI 的情况，把事件传递给相应的 ViewGroup；

public boolean onInterceptTouchEvent(MotionEvent ev)：对分发的事件进行拦截（注意拦截 ACTION_DOWN 和其他 ACTION 的差异），有两种情况：

第一种情况：如果 ACTION_DOWN 的事件没有被拦截，顺利找到了 TargetView，那么后续的 MOVE 和UP 都能够下发，如果后续的 MOVE 与UP 下发时还有继续拦截的话，事件只能传递到拦截层，并且发出 ACTION_CANCEL;

第二种情况：如果 ACTION_DOWN 的事件下发时被拦截，导致没有找到 TargetView，那么后续的 MOVE 和UP 都无法向下派发了，在 Activity 层就终止了传递。

public boolean onTouchEvent（MotionEvent ev）：响应处理函数，如果设置对应的 Listener 的话，这里还会与 OnTouch、onClick、onLongClick 相关联， 具体的执行顺序是：

onTouch（）——>onTouchEvent（）——>onClick（）——>onLongClick

（），是否能够顺序执行，取决于每个方法返回值是 true 还是false。

专题讲解——Bitmap 的使用及内存优化

位图是相对于矢量图而言的，也称为点阵图，位图由像素组成，图像的清晰度由单位长度内的像素的多少来决定的，在 Android 系统中，位图使用 Bitmap 类来表示，该类位于 android.graphics 包中，被 final 所修饰，不能被继承，创建 Bitmap 对象可使用该类的静态方法 createBitmap，也可以借助 BitmapFactory 类来实现。Bitmap 可以获取图像文件信息，如宽高尺寸等，可

以进行图像剪切、旋转、缩放等操作.

淘宝关注【闵课通商学院】，免费领取200G大礼包　淘宝搜《闵课通商学院》，小白轻松拿高薪offer

BitmapFactory 是创建 Bitmap 的工具类，能够以文件、字节数组、输入流的形式创建位图对象，BitmapFactory 类提供的都是静态方法，可以直接调用， BitmapFactory. Options 类是 BitmapFactory 的静态内部类，主要用于设定位图的解析参数。在解析位图时，将位图进行相应的缩放，当位图资源不再使用时， 强制资源回收，可以有效避免内存溢出。

缩略图：不加载位图的原有尺寸，而是根据控件的大小呈现图像的缩小尺寸，就是缩略图。

将大尺寸图片解析为控件所指的尺寸的思路：

实例化BitmapFactory . Options 对象来获取解析属性对象，设置BitmapFactory. Options 的属性inJustDecodeBounds 为true 后，再解析位图时并不分配存储空间，但可以计算出原始图片的宽度和高度，即outWidth 和outHeight，将这两个数值与控件的宽高尺寸相除，就可以得到缩放比例，即inSampleSize 的值，然后重新设置inJustDecodeBounds 为false，inSampleSize 为计算所得的缩放比例，重新解析位图文件，即可得到原图的缩略图。

获取控件宽高属性的方法：可以利用控件的getLayoutParams()方法获得控件的LayoutParams 对象，通过LayoutParams 的Width 和Height 属性来得到控件的宽高，同样可以利用控件的setLayoutParams( )方法来动态的设置其宽高， 其 中 LayoutParams 是 继 承 于 Android.view.viewGroup.LayoutParams， LayoutParams 类是用于child view(子视图)向parent view(父视图)传递布局（Layout）信息包，它封装了 Layout 的位置、宽、高等信息。

Bitmap **的内存优化：**

及时回收Bitmap 的内存：Bitmap 类有一个方法recycle( )，用于回收该Bitmap 所占用的内存，当保证某个Bitmap 不会再被使用（因为Bitamap 被强制释放后，再次使用它会抛出异常）后，能够在Activity 的onStop （）方法或onDestory （）方法中将其回收，回收方法：

```
if ( bitmap !=null && !bitmap.isRecycle ( ) ){
```

```
            bitmap.recycle( )
            ; bitmap=null ;
            }
        System.gc( ) ;
```

System.gc()方法可以加快系统回收内存的到来；

捕获异常：为了避免应用在分配 Bitmap 内存时出现 OOM 异常以后 Crash 掉，需在对 Bitmap 实例化的过程中进行 OutOfMemory 异常的捕获；

缓存通用的 Bitmap 对象：缓存分为硬盘缓存和内存缓存，将常用的 Bitmap
对象放到内存中缓存起来，或将从网络上获取到的数据保存到 SD 卡中；

压缩图片：即以缩略图的形式显示图片。

专题讲解——使用 View 绘制视图

View 类是Android 平台中各种控件的父类，是 UI（用户界面）的基础构件，View 相当于屏幕上的一块矩形区域，其重复绘制这个区域和处理事件，View 是所有 Weight 类（组件类）的父类，其子类 ViewGroup 是所有 Layout 类的父类，如果需要自定义控件，也要继承 View，实现 onDraw 方法和事件处理方法。

View **绘制的流程：**OnMeasure()——>OnLayout（）——>OnDraw（）

OnMeasure（）：测量视图大小，从顶层父 View 到子View 递归调用 measure 方法，measure 方法又回调 OnMeasure。

OnLayout（）：确定 View 的位置，进行页面的布局，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 放到合适的位置上。

OnDraw（）：绘制视图，ViewGroup 创建一个 Canvas 对象，调用 OnDraw（）方法，但OnDraw 方法是个空方法，需要自己根据 OnMeasure 和OnLayout 获取得到参数进行自定义绘制。

注意：在 Activity 的生命周期中没有办法获取 View 的宽高，原因就是 View 没有测量完。

Canvas **类：**Canvas 类位于 android.graphics 包中，它表示一块画布，可以使用该类提供的方法，绘制各种图形图像，Canvas 对象的获取有两种方式：一种是重写 View.onDraw 方法时，在该方法中 Canvas 对象会被当作参数传递过来， 在该 Canvas 上绘制的图形图像会一直显示在 View 中，另一种是借助位图创建Canvas，参考代码：

Bitmap bitmap = Bitmap.CreatBitmap(100，100，Bitmap.Config，

ARGB_888); Canvas canvas = new Canvas（bitmap）；

Android **中页面的横屏与竖屏操作：**

在配置文件中为Activity 设置属性 android：screenOrientation = “string ”；其中string 为属性值，landscape（横屏）或portrait（竖屏）；

在代码中设置：setRequestedOrientation（ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE）；

setRequestedOrientation（ActivityInfo.SCREEN_ORIENTATION_PORTRAIT）;为防止切换后重新启动当前 Activity，应在配置文件中添加 android：configChanges=“keyboardHidden|orientation”属性，并在 Activity 中重写 onConfigurationChanged 方法。

获取手机中屏幕的宽和高的方法：

通过Context 的getWindowManager（）方法获得 WindowManager 对象， 再从WindowManager 对象中通过 getDefaultDisplay 获得Display 对象，

再从

Display 对象中获得屏幕的宽和高。

专题讲解——Android 内存泄漏及管理

内存溢出（out of memory**）：**是指程序在申请内存时，没有足够的内存空间供其使用，出现 out of Memory 的错误，比如申请了一个 integer，但给它存了 long 才能存下的数，这就是内存溢出，通俗来讲，内存溢出就是内存不够用。

内存泄漏（Memory Leak**）**:是指程序使用 new 关键字向系统申请内存后，无法释放已经申请的内存空间，一次内存泄漏可以忽略，但多次内存泄漏堆积后会造成很严重的后果，会占用光内存空间。

以发生的方式来看，内存泄漏可以分为以下几类：

常发性内存泄漏：发生内存泄漏的代码会被执行多次，每次执行都会导致一块内存泄漏；

偶发性内存泄漏：发生内存泄露的代码只有在某些特定的环境或情况下才会发生；

一次性内存泄漏：发生内存泄漏的代码只会被执行一次，或是由于算法上的缺陷，导致总会有一块且仅有一块内存发生泄漏；

隐式内存泄漏：程序在运行过程中不停的分配内存，但是直到程序结束的时候才会释放内存；

常见造成内存泄漏的原因：

单例模式引起的内存泄漏：由于单例的静态特性使得单例的生命周期和程序的生命周期一样长，如果一个对象（如 Context）已经不再使用，而单例对象

还持有对象的引用就会造成这个对象不能正常回收；

Handler 引起的内存泄漏：子线程执行网络任务，使用 Handler 处理子线程发送的消息，由于 Hnadler 对象是非静态匿名内部类的实例对象，持有外部类
（Activity）的引用，在Handler　Message 中，Looper 线程不断轮询处理消息，当Activity 退出时还有未处理或正在处理的消息时，消息队列中的消息持有Handler 对象引用，Handler 又持有Activity，导致Activity 的内存和资源不能及时回收；

线程造成内存泄漏：匿名内部类 Runnable 和AsyncTask 对象执行异步任务，当前 Activity 隐式引用，当前 Activity 销毁之前，任务还没有执行完，将导致Activity 的内存和资源不能及时回收；

资源对象未关闭造成的内存泄漏：对于使用了 BroadcastReceiver、File、Bitmap 等资源，应该在 Activity 销毁之前及时关闭或注销它们，否则这些资源将不会回收，造成内存泄漏；

内存泄漏的检测工具：LeakCanary

LeakCanary 是 Square 公司开源的一个检测内存泄漏的函数库，可以在开发的项目中集成，在 Debug 版本中监控 Activity、Fragment 等的内存泄漏，LeakCanaty 集成到项目之后，在检测内存泄漏时，会发送消息到系统通知栏，点击后会打开名称 DisplayLeakActivity 的页面，并显示泄露的跟踪信息，Logcat 上面也会有对应的日志输出。

专题讲解——Android 设计模式之 MVC

MVC 即Model-View-Controller，M 是模型，V 是视图，C 是控制器，MVC 模式下系统框架的类库被划分为模型（Model）、视图（View）、控制器（Controller），模型对象负责建立数据结构和相应的行为操作处理，视图负责在屏幕上渲染出相应的图形信息，展示给用户看，控制器对象负责截获用户的按键和屏幕触摸事件，协调 Model 对象和View 对象。

用户与视图交互，视图接收并反馈用户的动作，视图把用户的请求传给相应的控制器，由控制器决定调用哪个模型，然后由模型调用相应的业务逻辑对用户请求进行加工处理，如果需要返回数据，模型会把相应的数据返回给控制器，由控制器调用相应的视图，最终由视图格式化和渲染返回的数据，一个模型可以有多个视图，一个视图可以有多个控制器，一个控制器可以有多个模型。

Model(**模型**)：Model 是一个应用系统的核心部分，代表了该系统实际要实现的所有功能处理，比如在视频播放器中，模型代表了一个视频数据库及播放视频的程序和函数代码，Model 在values 目录下通过 xml 文件格式生成，也可以由代码动态生成，View 和Model 是通过桥梁 Adapter 来连接起来的；

View(**视图**)：View 是软件应用传给用户的一个反馈结果，它代表了软件应用中的图形展示，声音播放、触觉反馈等，视图的根节点是应用程序的自身窗口，View 在Layout 目录中通过 xml 文件格式生成，用 findViewByld （）获取， 也可以通过代码动态生成；

Controller(**控制器**)：Controller 在软件应用中负责对外部事件的响应，包括：键盘敲击、屏幕触摸、电话呼入等，Controller 实现了一个事件队列，每一个外部事件均在事件队列中被唯一标识，框架依次将事件从队列中移出并派发出去；

专题讲解——JVM 运行原理详解

JVM **简析：**说起 Java，我们首先想到的是 Java 编程语言，然而事实上， Java 是一种技术，它由四方面组成：Java 编程语言、Java 类文件格式、Java 虚拟机和 Java 应用程序接口(Java API)。它们的关系如下图所示：

Java 平台由Java 虚拟机和Java 应用程序接口搭建，Java 语言则是进入这个平台的通道，用Java 语言编写并编译的程序可以运行在这个平台上。这个平台的结构如下图所示：　运行期环境代表着Java 平台，开发人员编写Java 代码(.java 文件)，然后将之编译成字节码(.class 文件)，再然后字节码被装入内存， 一旦字节码进入虚拟机，它就会被解释器解释执行，或者是被即时代码发生器有选择的转换成机器码执行。

JVM 在它的生存周期中有一个明确的任务，那就是运行Java 程序，因此当Java 程序启动的时候，就产生JVM 的一个实例；当程序运行结束的时候，该实例也跟着消失了。在Java 平台的结构中，　可以看出，Java 虚拟机(JVM)　处在核心的位置，是程序与底层操作系统和硬件无关的关键。它的下方是移植接口， 移植接口由两部分组成：适配器和Java 操作系统，其中依赖于平台的部分称为适配器；JVM　通过移植接口在具体的平台和操作系统上实现；在JVM　的上方是Java 的基本类库和扩展类库以及它们的API， 利用Java API 编写的应用程序(application) 和小程序(Java applet) 可以在任何Java 平台上运行而无需考虑底层平台，就是因为有Java 虚拟机(JVM)实现了程序与操作系统的分离，从而实现了Java 的平台无关性。

下面我们从 JVM 的基本概念和运过程程这两个方面入手来对它进行深入的研究。

2.JVM **基本概念**

(1) 基本概念：JVM 是可运行Java 代码的假想计算机 ，包括一套字节码指令集、一组寄存器、一个栈、一个垃圾回收堆和一个存储方法域。JVM 是运行在操作系统之上的，它与硬件没有直接的交互。

(2) 运行过程：我们都知道 Java 源文件，通过编译器，能够生产相应的.Class 文件，也就是字节码文件，而字节码文件又通过 Java 虚拟机中的解释器，编译成特定机器上的机器码 。也就是如下： ① Java 源文件—->编译器—
->字节码文件；② 字节码文件—->JVM—->机器码

每一种平台的解释器是不同的，但是实现的虚拟机是相同的，这也就是 Java 为什么能够跨平台的原因了，当一个程序从开始运行，这时虚拟机就开始实例化了，多个程序启动就会存在多个虚拟机实例。程序退出或者关闭，则虚拟机实例消亡，多个虚拟机实例之间数据不能共享。

(3) 三种JVM:① Sun 公司的HotSpot；② BEA 公司的JRockit；③ IBM 公司的J9JVM；

在JDK1.7 及其以前我们所使用的都是 Sun 公司的 HotSpot，但由于 Sun 公司和 BEA 公司都被 oracle 收购，jdk1.8 将采用 Sun 公司的 HotSpot 和BEA 公司的JRockit 两个JVM 中精华形成 jdk1.8 的JVM。

3.JVM **的体系结构**

（1） Class Loader 类加载器： 负责加载 .class 文件，class 文件在文件开头有特定的文件标示，并且 ClassLoader 负责class 文件的加载等，至于它是否可以运行，则由 Execution Engine 决定。①定位和导入二进制 class 文件；② 验证导入类的正确性；③ 为类分配初始化内存；④ 帮助解析符号引用.

（2） Native Interface 本地接口:本地接口的作用是融合不同的编程语言为Java 所用，它的初衷是融合 C/C++程序，Java 诞生的时候 C/C++横行的时候，要想立足，必须有调用 C/C++程序，于是就在内存中专门开辟了一块区域处理标记为native 的代码，它的具体作法是 Native Method Stack 中登记native 方法， 在Execution Engine 执行时加载 native libraies。

(3) Execution Engine 执行引擎：执行包在装载类的方法中的指令，也就是方法。

(4) Runtime data area 运行数据区：虚拟机内存或者 Jvm 内存，冲整个计

算

淘宝关注【闵课通商学院】，免费领取200G大礼包　淘宝搜《闵课通商学院》，小白轻松拿高薪offer

机内存中开辟一块内存存储 Jvm 需要用到的对象，变量等，运行区数据有分很多小区，分别为：方法区，虚拟机栈，本地方法栈，堆，程序计数器。

4.JVM **数据运行区详解（栈管运行，堆管存储）：**

说明：JVM 调优主要就是优化 Heap 堆 和 Method Area 方法区。

- Native Method Stack 本地方法栈：它的具体做法是 Native Method Stack

中登记 native 方法，在 Execution Engine 执行时加载 native libraies。

- PC Register 程序计数器：每个线程都有一个程序计算器，就是一个指针，指向方法区中的方法字节码（下一个将要执行的指令代码），由执行引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不记。

- Method Area 方法区：方法区是被所有线程共享，所有字段和方法字节码，以及一些特殊方法如构造函数，接口代码也在此定义。简单说，所有定义的方法的信息都保存在该区域，此区域属于共享区间。静态变量+常量+类信息+ 运行时常量池存在方法区中，实例变量存在堆内存中。

- Stack 栈：① 栈是什么，栈也叫栈内存，主管 Java 程序的运行，是在线程创建时创建，它的生命期是跟随线程的生命期，线程结束栈内存也就释放，对于栈来说不存在垃圾回收问题，只要线程一结束该栈就Over，生命周期和线程一致，是线程私有的。基本类型的变量和对象的引用变量都是在函数的栈内存中分配。② 栈存储什么？栈帧中主要保存3 类数据：本地变量（Local Variables）：输入参数和输出参数以及方法内的变量；栈操作（Operand Stack）：记录出栈、入栈的操作； 栈帧数据（Frame Data）：包括类文件、方法等等。③栈运行原理，栈中的数据都是以栈帧（Stack Frame）的格式存在， 栈帧是一个内存区块，是一个数据集，是一个有关方法和运行期数据的数据集，当一个方法 A 被调用时就产生了一个栈帧 F1，并被压入到栈中，A 方法又调用了 B 方法，于是产生栈帧 F2 也被压入栈，B 方法又调用了 C 方法，于是产生栈帧F3 也被压入栈……依次执行完毕后，先弹出后进 F3 栈帧，再弹出 F2 栈帧，再弹出 F1 栈帧。遵循“先进后出”/“后进先出”原则。

- Heap 堆：堆这块区域是 JVM 中最大的，应用的对象和数据都是存在这

个区域，这块区域也是线程共享的，也是 gc 主要的回收区，一个 JVM 实例只存在一个堆类存，堆内存的大小是可以调节的。类加载器读取了类文件后，需要把类、方法、常变量放到堆内存中，以方便执行器执行，堆内存分为三部分：

① 新生区:新生区是类的诞生、成长、消亡的区域，一个类在这里产生，应用，最后被垃圾回收器收集，结束生命。新生区又分为两部分：伊甸区（Eden space）和幸存者区（Survivor pace），所有的类都是在伊甸区被 new 出来的。幸存区有两个：0 区（Survivor0 space）和1 区（Survivor 1 space）。当伊甸园的空间用完时，程序又需要创建对象，JVM 的垃圾回收器将对伊甸园进行垃圾回收（Minor GC）,将伊甸园中的剩余对象移动到幸存0 区。若幸存0 区也满了，再对该区进行垃圾回收，然后移动到1 区。那如果1 去也满了呢？再移动到养老区。若养老区也满了，那么这个时候将产生Major GC（FullGCC），进行养老区的内存清理。若养老区执行Full GC 之后发现依然无法进行对象的保存，就会产生OOM 异常“OutOfMemoryError”。如果出现 java.lang.OutOfMemoryError: Java heap space 异常，说明Java 虚拟机的堆内存不够。原因有二： a.Java 虚拟机的堆内存设置不够，可以通过参数-Xms、-Xmx 来调整。b.代码中创建了大量大对象，并且长时间不能被垃圾收集器收集（存在被引用）。

养老区:养老区用于保存从新生区筛选出来的 JAVA 对象，一般池对象都在这个区域活跃。

③ 永久区：永久存储区是一个常驻内存区域，用于存放 JDK 自身所携带的 Class,Interface 的元数据，也就是说它存储的是运行环境必须的类信息，被装载进此区域的数据是不会被垃圾回收器回收掉的，关闭 JVM 才会释放此区域所占用的内存。 如果出现 java.lang.OutOfMemoryError:PermGen space，说明是Java 虚拟机对永久代 Perm 内存设置不够。原因有二：

a. 程序启动需要加载大量的第三方 jar 包。例如：在一个 Tomcat 下部署了太多的应用。

b. 大量动态反射生成的类不断被加载，最终导致 Perm 区被占满。

方法区和堆内存的异议： 实际而言，方法区和堆一样，是各个线程共享的内存区域，它用于存储虚拟机加载的：类信息+普通常量+静态常量+编译器编译后的代码等等，虽然 JVM 规范将方法区描述为堆的一个逻辑部分，但它却还有一个别名叫做 Non-Heap（非堆），目的就是要和堆分开。

对于 HotSpot 虚拟机，很多开发者习惯将方法区称之为“永久代（Parmanent Gen）”,但严格本质上说两者不同，或者说使用永久代来实现方法区而已，永久代是方法区的一个实现，jdk1.7 的版本中，已经将原本放在永久代的字符串常量池移走。

常量池（Constant Pool）是方法区的一部分，Class 文件除了有类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池，这部分内容将在类加载后进入方法区的运行时常量池中存放。

专题讲解——Android 平台的虚拟机 Dalvik

Dalvik **概述：**Dalvik 是Google 公司自己设计用于 Android 平台的 Java 虚拟机。它可以支持已转换为.dex（即Dalvik Executable）格式的 Java 应用程序的运行，.dex 格式是专为 Dalvik 设计的一种压缩格式，可以减少整体文件尺寸，提高I/o 操作的类查找速度所以适合内存和处理器速度有限的系统。

Dalvik **虚拟机**(DVM)**和** Java **虚拟机**(JVM)**首要差别：**Dalvik 基于寄存器，而JVM 基于栈。性能有很大的提升。基于寄存器的虚拟机对于更大的程序来说， 在它们编译的时候，花费的时间更短。

寄存器的概念：寄存器是中央处理器内的组成部分。寄存器是有限存贮容

量的高速存贮部件，它们可用来暂存指令、数据和位址。在中央处理器的控制部件中，包含的寄存器有指令寄存器(IR)和程序计数器(PC)，在中央处理器的算术及逻辑部件中，包含的寄存器有累加器(ACC)。

栈的概念：栈是线程独有的，保存其运行状态和局部自动变量的（所以多线程中局部变量都是相互独立的，不同于类变量）。栈在线程开始的时候初始化（线程的Start 方法，初始化分配栈），每个线程的栈互相独立。每个函数都有自己的栈，栈被用来在函数之间传递参数。操作系统在切换线程的时候会自动的切换栈，就是切换 SS/ESP 寄存器。栈空间不需要在高级语言里面显式的分配和释放。

DVM **进程的设计规则：**

每个应用程序都运行在它自己的 Linux 空间。在需要执行该应用程序时 Android 将启动该进程，当不再需要该应用程序，并且系统资源分配不够时，则系统终止该进程。

每个应用程序都有自己的（DVM），所以任一应用程序的代码与其他应用程序的代码是相互隔离的。

默认情况下，每个应用程序都给分配一个唯一的 Linux 用户ID。所以应用程序的文件只能对该应用程序可见。

所以说每个应用程序都拥有一个独立的 DVM，而每个DVM 在Linux 中又是一个进程，所以说 DVM 进程和Linux 进程可以说是一个概念。

Android **应用程序的编译：**Android 所有类都通过 JAVA 编译器编译，然后通过Android SDK 的“dex 文件转换工具”转换为“dex”的字节文件，再由 DVM 载入执行。

Android　ART **模式简介：**Android4.4 引入ART 模式来代替 Dalvik 虚拟机。ART 是AndroidRuntime 的缩写，它提供了以 AOT（Ahead-Of-Time）的方式运行Android 应用程序的机制。所谓 AOT 是指在运行前就把中间代码静态编译成本地代码，这就节省了 JIT 运行时的转换时间。因此，和采用 JIT 的Dalvik 相比，

ART 模式在总体性能有了很大的提升，应用程序不但运行效率更高，耗电量更低，而且占用的内存也更少；ART 和 dalvik 相比，系统的性能得到了显著提升，同时占用的内存更少，因此能支持配置更低的设备。但是 ART 模式下编译出来的文件会比以前增大 10%-20%，系统需要更多的存储空间，同时因为在安装时要执行编译，应用的安装时间也比以前更长了；ART 和dalvik 相比，系统的性能得到了显著提升，同时占用的内存更少，因此能支持配置更低的设备。但是ART 模式下编译出来的文件会比以前增大 10%-20%，系统需要更多的存储空间，同时因为在安装时要执行编译，应用的安装时间也比以前更长了。

专题讲解——Java 的内存分配

Java **内存分配主要包括以下几个区域**：1. 寄存器：我们在程序中无法控制；2. 栈：存放基本类型的数据和对象的引用，但对象本身不存放在栈中，而是存放在堆中；3. 堆：存放用 new 产生的数据；4. 静态域：存放在对象中用 static 定义的静态成员；5. 常量池：存放常量；6. 非RAM(随机存取存储器)存储：硬盘等永久存储空间。

Java **内存分配中的栈：**在函数中定义的一些基本类型的变量数据和对象的引用变量都在函数的栈内存中分配。当在一段代码块定义一个变量时，Java 就在栈中为这个变量分配内存空间，当该变量退出该作用域后，Java 会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作他用。

Java **内存分配中的堆：**堆内存用来存放由 new 创建的对象和数组。在堆中分配的内存，由 Java 虚拟机的自动垃圾回收器来管理。在堆中产生了一个数组或对象后，还可以 在栈中定义一个特殊的变量，让栈中这个变量的取值等于数组或对象在堆内存中的首地址，栈中的这个变量就成了数组或对象的引用变量。引用变量就相当于是为数组或对象起的一个名称，以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象。引用变量就相当于是为数组或者

对象起的一个名称。引用变量是普通的变量，定义时在栈中分配，引用变量在程序运行到其作用域之外后被释放。而数组和对象本身在堆中分配，即使程序运行到使用 new 产生数组或者对象的语句所在的代码块之外，数组和对象本身占据的内存不会被释放，数组和对象在没有引用变量指向它的时候，才变为垃圾，不能在被使用，但仍然占据内存空间不放，在随后的一个不确定的时间被垃圾回收器收走（释放掉）。这也是 Java 比较占内存的原因。实际上，栈中的变量指向堆内存中的变量，这就是Java 中的指针!

Java **内存分配中的常量池** (constant pool)：常量池指的是在编译期被确定，并被保存在已编译的.class 文件中的一些数据。除了包含代码中所定义的各种基本类型（如int、long 等等）和对象型（如String 及数组）的常量值(final)还包含一些以文本形式出现的符号引用，比如： 1.类和接口的全限定名；2.字段的名称和描述符； 3.方法和名称和描述符。虚拟机必须为每个被装载的类型维护一个常量池。常量池就是该类型所用到常量的一个有序集和，包括直接常量

（string,integer 和 floating point 常量）和对其他类型，字段和方法的符号引用。对于String 常量，它的值是在常量池中的。而JVM 中的常量池在内存当中是以表的形式存在的，对于String 类型，有一张固定长度的

CONSTANT_String_info 表用来存储文字字符串值，注意：该表只存储文字字符串值，不存储符号引用。说到这里，对常量池中的字符串值的存储位置应该有一个比较明了的理解了。在程序执行的时候,常量池会储存在Method Area,而不是堆中。

堆与栈：Java 的堆是一个运行时数据区,类的(对象从中分配空间。这些对象通过new、newarray、anewarray 和multianewarray 等指令建立，它们不需要程序代码来显式的释放。堆是由垃圾回收来负责的，堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，因为它是在运行时动态分配内存的，Java 的垃圾收集器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。 栈的优势是，存取速度比堆要快，仅次于寄存器，栈数据可以共享。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。栈中主要存放一些基本类型的变量数据（int，short,

long, byte, float, double, boolean, char）和对象句柄(引用)。栈有一个很重要的特殊

性，就是存在栈中的数据可以共享

专题讲解——Android 中的 Binder 机制

Binder 是Android 系统进程间通信（IPC）方式之一。Linux 已经拥有的进程间通信IPC 手段包括(Internet Process Connection)：管道（Pipe）、信号（Signal）和跟踪（Trace）、插口（Socket）、报文队列（Message）、共享内存（Share Memory）和信号量（Semaphore）。Binder 可以提供系统中任何程序都可以访问的全局服务。Android 的Binder 的框架如下：

上图中涉及到Binder 模型的4 类角色：Binder 驱动，ServiceManager，Server 和Client。　因为后面章节讲解Binder 时，都是以MediaPlayerService 和MediaPlayer 为代表进行讲解的；这里就使用MediaPlayerService 代表了Server，而MediaPlayer 则代表了Client。

Binder 机制的目的是实现 IPC(Inter-ProcessCommunication)，即实现进程间通信。在上图中，由于 MediaPlayerService 是Server 的代表，而 MediaPlayer 是Client 的代表；因此，对于上图而言，Binder 机制则表现为"实现MediaPlayerService 和MediaPlayer 之间的通信"。

专题讲解——Android 中的缓存机制

移动开发本质上就是手机和服务器之间进行通信，需要从服务端获取数据。反复通过网络获取数据是比较耗时的，特别是访问比较多的时候，会极大影响了性能，Android 中可通过缓存机制来减少频繁的网络操作，减少流量、提升性能。

实现原理：把不需要实时更新的数据缓存下来，通过时间或者其他因素来判别是读缓存还是网络请求，这样可以缓解服务器压力，一定程度上提高应用响应速度，并且支持离线阅读。

Bitmap **的缓存：**在许多的情况下(像 ListView, GridView 或 ViewPager 之类的组件)我们需要一次性加载大量的图片，在屏幕上显示的图片和所有待显示的图片有可能需要马上就在屏幕上无限制的进行滚动、切换。像ListView, GridView 这类组件，它们的子项当不可见时，所占用的内存会被回收以供正在前台显示子项使用。垃圾回收器也会释放你已经加载了的图片占用的内存。如果你想让你的UI 运行流畅的话，就不应该每次显示时都去重新加载图片。保持一些内存和文件缓存就变得很有必要了。

使用内存缓存：通过预先消耗应用的一点内存来存储数据，便可快速的为应用中的组件提供数据，是一种典型的以空间换时间的策略。LruCache 类（Android v4 Support Library 类库中开始提供）非常适合来做图片缓存任务，它可以使用一个 LinkedHashMap 的强引用来保存最近使用的对象，并且当它保存的对象占用的内存总和超出了为它设计的最大内存时会把不经常使用的对象成员踢出以供垃圾回收器回收。给 LruCache 设置一个合适的内存大小，需考虑如下因素：

还剩余多少内存给你的 activity 或应用使用屏幕上需要一次性显示多少张图片和多少图片在等待显示 手机的大小和密度是多少（密度越高的设备需要越大的 缓存） 图片的尺寸（决定了所占用的内存大小） 图片的访问频率（频率高的在内存中一直保存）保存图片的质量（不同像素的在不同情况下显示）；

使用磁盘缓存：内存缓存能够快速的获取到最近显示的图片，但不一定就能够获取到。当数据集过大时很容易把内存缓存填满（如GridView　）。你的应用也有可能被其它的任务（比如来电）中断进入到后台，后台应用有可能会被杀死，那么相应的内存缓存对象也会被销毁。当你的应用重新回到前台显示时，你的应用又需要一张一张的去加载图片了。磁盘文件缓存能够用来处理这些情况，保存处理好的图片，当内存缓存不可用的时候，直接读取在硬盘中保存好的图片，这样可以有效的减少图片加载的次数。读取磁盘文件要比直接从内存缓存中读取要慢一些，而且需要在一个 UI 主线程外的线程中进行，因为磁盘的读取速度是不能够保证的，磁盘文件缓存显然也是一种以空间换时间的策略。

使用 SQLite 进行缓存：网络请求数据完成后，把文件的相关信息（如 url（一般作为唯一标示），下载时间，过期时间）等存放到数据库。下次加载的时候根据url 先从数据库中查询，如果查询到并且时间未过期，就根据路径读取本地文件，从而实现缓存的效果。

文件缓存：思路和一般缓存一样，把需要的数据存储在文件中，下次加载时判断文件是否存在和过期（使用File.lastModified()方法得到文件的最后修改时间，与当前时间判断），存在并未过期就加载文件中的数据，否则请求服务器重新下载。

专题讲解——Android 中图片的三级缓存策略

三级缓存：　•内存缓存，优先加载，速度最快；　•本地缓存，次优先加载，速度快；•网络缓存，最后加载，速度慢，浪费流量；

三级缓存策略，最实在的意义就是　减少不必要的流量消耗，增加加载速度。

三级缓存的原理：

•首次加载的时候通过网络加载，获取图片，然后保存到内存和 SD 卡中。

•之后运行 APP 时，优先访问内存中的图片缓存。

•如果内存没有，则加载本地 SD 卡中的图片。

具体的缓存策略可以是这样的：内存作为一级缓存，本地作为二级缓存，网络加载为最后。其中，内存使用 LruCache，其内部通过 LinkedhCache。加载图片的时候，首先使用 LRU 方式进行寻找，找不到指定内容，按照三级缓存的方式，进行本地搜索，还没有就网络加载。ashMap来持有外界缓存对象的强引用；对于本地缓存，使用 DiskLru