#### Activity生命周期



1.启动Activity：系统会先调用onCreate方法，然后调用onStart方法，最后调用onResume，Activity进入运行状态。

2.当前Activity被其他Activity覆盖其上或被锁屏：系统会调用onPause方法，暂停当前Activity的执行。

3.当前Activity由被覆盖状态回到前台或解锁屏：系统会调用onResume方法，再次进入运行状态。

4.当前Activity转到新的Activity界面或按Home键回到主屏，自身退居后台：系统会先调用onPause方法，然后调用onStop方法，进入停滞状态。

5.用户后退回到此Activity：系统会先调用onRestart方法，然后调用onStart方法，最后调用onResume方法，再次进入运行状态。

6.当前Activity处于被覆盖状态或者后台不可见状态，即第2步和第4步，系统内存不足，杀死当前Activity，而后用户退回当前Activity：再次调用onCreate方法、onStart方法、onResume方法，进入运行状态。

7.用户退出当前Activity：系统先调用onPause方法，然后调用onStop方法，最后调用onDestory方法，结束当前Activity。

除了几个常见的方法外，我们还添加了onWindowFocusChanged、onSaveInstanceState、onRestoreInstanceState方法：

1. onWindowFocusChanged方法：（1）在Activity窗口获得或失去焦点时被调用，例如创建时首次呈现在用户面前；（2）当前Activity被其他Activity覆盖；（3）当前Activity转到其他Activity或按Home键回到主屏，自身退居后台；（4）用户退出当前Activity。以上几种情况都会调用onWindowFocusChanged，并且当Activity被创建时是在onResume之后被调用，当Activity被覆盖或者退居后台或者当前Activity退出时，它是在onPause之后被调用。这个方法在某种场合下还是很有用的，例如程序启动时想要获取视特定视图组件的尺寸大小，在onCreate中可能无法取到，因为窗口Window对象还没创建完成，这个时候我们就需要onWindowFocusChanged里获取。
2. onSaveInstanceState：(1)在Activity被覆盖或退居后台之后，系统资源不足将其杀死，此方法会被调用；(2)在用户改变屏幕方向时，此方法会被调用；(3)在当前Activity跳转到其他Activity或者按Home键回到主屏，自身退居后台时，此方法会被调用。第一种情况我们无法保证什么时候发生，系统根据资源紧张程度去调度；第二种是屏幕翻转方向时，系统先销毁当前的Activity，然后再重建一个新的，调用此方法时，我们可以保存一些临时数据；第三种情况系统调用此方法是为了保存当前窗口各个View组件的状态。onSaveInstanceState的调用顺序是在onPause之前。
3. .onRestoreInstanceState：(1)在Activity被覆盖或退居后台之后，系统资源不足将其杀死，然后用户又回到了此Activity，此方法会被调用；(2)在用户改变屏幕方向时，重建的过程中，此方法会被调用。我们可以重写此方法，以便可以恢复一些临时数据。onRestoreInstanceState的调用顺序是在onStart之后。

#### 防止屏幕切换时activity的销毁->重建

1. 在AndroidManifest.xml中设置

android:screenOrientation="portrait"指定Activity的屏幕方向为竖屏；android:screenOrientation="landscape"指定Activity的屏幕方向为横屏

1. 设置即使切换屏幕也不执行销毁->重建：

不设置Activity的android:configChanges时，切屏会重新调用各个生命周期，切横屏时会执行一次，切竖屏时会执行两次

设置Activity的android:configChanges="orientation"时，切屏还是会重新调用各个生命周期，切横、竖屏时只会执行一次

设置Activity的android:configChanges="orientation|keyboardHidden"时，切屏不会重新调用各个生命周期，只会执行onConfigurationChanged方法

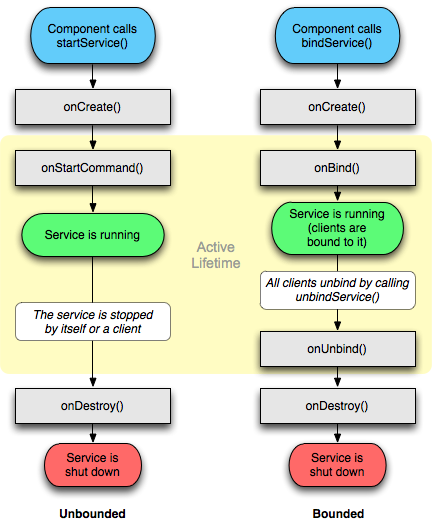
#### Activity的四种启动模式

注意：存放Activity实例的是一个具有栈结构的对象

1. standard:每次都会创建新的Activity实例
2. singleTop:只要这个Activity位于栈顶的话，就直接使用而不重新创建
3. singleTask:只要这个栈中有这个Activity实例，就会将这个实例上面的Activity弹出去，使这个Activity位于栈顶来使用
4. singleInstance:一旦使用这个Activity实例的话，就会启用一个新的栈来创建这个Activity实例，每次回退的第一步都是栈之间的跳转

备注：每个栈都会有一个affinity属性，这个属性就是决定栈中放置哪些Activity实例的重要原因。在这个栈中Activity实例肯定都是具有相同affinity属性的实例。默认的affinity属性就是menifest配置文件中的包名。

#### Service生命周期



Service生命周期可以从两种启动Service的模式开始讲起，分别是context.**startService()**和context.**bindService()**

**1）**.startService的启动模式下的生命周期：当我们首次使用startService启动一个服务时，系统会实例化一个Service实例，依次调用其onCreate和onStartCommand方法，然后进入运行状态，此后，如果再使用startService启动服务时，不再创建新的服务对象，系统会自动找到刚才创建的Service实例，调用其onStart方法；如果我们想要停掉一个服务，可使用stopService/stopSelf方法，此时onDestroy方法会被调用，需要注意的是，不管前面使用了多个次startService，只需一次stopService，即可停掉服务。

(2) .bindService启动模式下的生命周期：在这种模式下，当调用者首次使用bindService绑定一个服务时，系统会实例化一个Service实例，并一次调用其onCreate方法和onBind方法，然后调用者就可以和服务进行交互了，此后，如果再次使用bindService绑定服务，系统不会创建新的Service实例，也不会再调用onBind方法；如果我们需要解除与这个服务的绑定，可使用unbindService方法，此时onUnbind方法和onDestroy方法会被调用。

备注：service被销毁，执行onDestory的条件

1. 不用startService启动
2. Service中没有一个绑定者

Service是用于后台服务的，当应用程序被挂到后台的时候，为了保证应用某些组件仍然可以工作。Service不是独立的线程，是依赖与应用程序的主线程。所以service必须是在主线中运行的，且不能执行耗时操作。如果一定要执行耗时操作的话那么就要在service中开启线程。

#### 保证Service不死

1. onStartCommand() 方法的返回值设为 START\_STICKY ，服务就会在资源紧张的时候被杀掉，然后在资源足够的时候再恢复
2. 设置为前台服务，使其有高的优先级，在资源紧张的时候也不会被杀掉。
3. 在service的ondestory方法中发广播，销毁就让其重建

#### IntentService

它是service的子类，当你的service的操作里不得不执行耗时操作的时候。就得在service中开启工作线程。这个IntentService的作用就是在执行onCreate操作的时候，内部开了一个线程，去执行你的耗时操作。

我们知道，service中每次启动或者调用service的时候都会调用onStartCommand()。与之对应的是，IntentService在每次启动或者调用的时候会执行onHandleIntent()方法（这是一个抽象方法，具体的实现要子类实现）。

onCreate()方法中：开启了一个handlerThread,并且获取到了当前线程的Looper。

onStart()方法中：将消息放入了消息队列。

在消息被handler接受并且回调的时候，执行了onHandlerIntent方法

#### Android异步消息处理机制（Looper,Handler,Message）

异步消息处理线程启动后会进入一个无限的循环中，每次循环都会从消息队列中取出一个消息然后回调消息处理函数进行处理，处理完成之后就会继续执行，一旦消息队列之中没有消息就会阻塞。

1. Looper
   1. Looper.prepare()函数中包含一个ThreadLocal，它是一个线程变量，将Looper实例放入ThreadLocal中。这就保证一个线程里面就只有一个Looper对象。并且还对Looper实例进行了判空，如果为空就会抛出异常,同时也说明了prepare函数不能调用两次。
   2. Looper.Loop()不断从MessageQueue中去取消息，交给消息的target属性的dispatchMessage去处理
2. 执行流程

1、首先Looper.prepare()在本线程中保存一个Looper实例，然后该实例中保存一个MessageQueue对象；因为Looper.prepare()在一个线程中只能调用一次，所以MessageQueue在一个线程中只会存在一个。

2、Looper.loop()会让当前线程进入一个无限循环，不端从MessageQueue的实例中读取消息，然后回调msg.target.dispatchMessage(msg)方法。

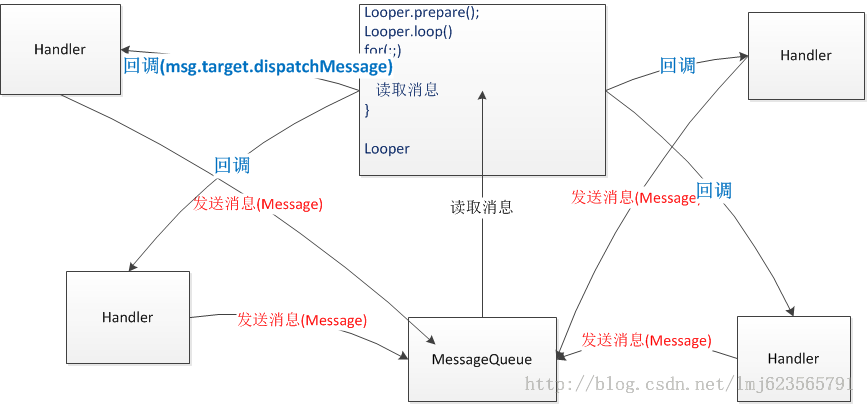
3、Handler的构造方法，会首先得到当前线程中保存的Looper实例，进而与Looper实例中的MessageQueue想关联。

4、Handler的sendMessage方法发消息，会给msg的target赋值为handler自身，然后加入MessageQueue中。

5、在构造Handler实例时，我们会重写handleMessage方法处理消息，也就是msg.target.dispatchMessage(msg)最终调用的方法。

备注：handler.post(Runnable r),这个也是在给创建Handler的线程发送消息。因为在post的方法中实际上调用了一个getPostMessage()方法，得到了一个Message对象，然后将我们创建的Runanable对象赋值给了Message对象。

注：虽然Message有两个方法都可以创建实例，但是Message.obtain()方法是从消息池中取对象用于复用，避免了使用New重新分配内存。



#### Android Binder框架

Android Binder框架分为服务器接口、Binder驱动、以及客户端接口；简单想一下，需要提供一个全局服务，那么全局服务那端即是服务器接口，任何程序即客户端接口，它们之间通过一个Binder驱动访问。

服务器端接口：实际上是Binder类的对象，该对象一旦创建，内部则会启动一个隐藏线程，会接收Binder驱动发送的消息，收到消息后，会执行Binder对象中的onTransact()函数，并按照该函数的参数执行不同的服务器端代码。

Binder驱动（连接的是Linux内核中的Binder事务中心）：该对象也为Binder类的实例，客户端通过该对象访问远程服务。

客户端接口：获得Binder驱动对象，调用其transact()发送消息至服务器

Service的作用其实就是为我们创建Binder驱动，即服务端与客户端连接的桥梁。

AIDL（android interface define lanuage）其实通过我们写的aidl文件，帮助我们生成了一个接口，一个Stub类用于服务端，一个Proxy类用于客户端调用。

#### BroadcastReceive

备注：广播一般也是在主线程中运行的，所以在onReceiver()方法中不能执行耗时操作。一般如果超过10s就会发生ANR.

**Android广播的使用场景？**

1.同一app内部的同一组件内的消息通信（单个或多个线程之间）；

2.同一app内部的不同组件之间的消息通信（单个进程）；

3.同一app具有多个进程的不同组件之间的消息通信；

4.不同app之间的组件之间消息通信；

5.Android系统在特定情况下与App之间的消息通信。

**Android广播的具体实现流程？**

1.广播接收者BroadcastReceiver通过Binder机制向AMS(Activity Manager Service)进行注册；

2.广播发送者通过binder机制向AMS发送广播；

3.AMS查找符合相应条件（IntentFilter/Permission等）的BroadcastReceiver，将广播发送到BroadcastReceiver（一般情况下是Activity）相应的消息循环队列中；

4.消息循环执行拿到此广播，回调BroadcastReceiver中的onReceive()方法。

**Android广播的分类？**

**普通广播：用户自己定义的广播。**完全异步的，可以在同一时刻（逻辑上）被所有广播接收者接收到，消息传递的效率比较高，但缺点是：接收者不能将处理结果传递给下一个接收者，并且无法终止广播Intent的传播

**系统广播：**系统广播在系统内部当特定事件发生时，有系统自动发出。

**有序广播：发送是无序的，但是接收是有序的。**先后顺序判定标准遵循为：将当前系统中所有有效的动态注册和静态注册的BroadcastReceiver按照priority属性值（-1000~1000）从大到小排序，对于具有相同的priority的动态广播和静态广播，动态广播会排在前面。先接收的BroadcastReceiver可以对此有序广播进行截断，使后面的BroadcastReceiver不再接收到此广播，也可以对广播进行修改，使后面的BroadcastReceiver接收到广播后解析得到错误的参数值。

**Android广播的两种注册方式和区别？**

**（1）在配置文件中注册：**当此App首次启动时，系统会自动实例化MyBroadcastReceiver，并注册到系统中。静态注册的广播接收器即使app已经退出，主要有相应的广播发出，依然可以接收到。

**（2）在代码中注册：**当此Activity实例化时，会动态将MyBroadcastReceiver注册到系统中。当此Activity销毁时，动态注册的MyBroadcastReceiver将不再接收到相应的广播。

#### ContentProvider

备注：是Android数据存储方式之一，它为存储和获取数据提供了统一的接口。ContentProvide对数据进行封装，不用关心数据存储的细节。使用表的形式来组织数据。提供者和使用者都是基于URI（"<content://hb.android.contentProvider/teacher>"）的。

**ContentProvider的实现过程？**

1. 定义一个CONTENT\_URI常量，提供了访问ContentProvider的标识符。
2. 定义一个类，继承ContentProvider。
3. 实现query,insert,update,delete,getType和onCreate方法。
4. 在AndroidManifest.xml当中进行声明。

#### Intent

备注：“意图，意向”，是不同组件之间通讯的媒介

**Intent七大属性？**

1. componentName（组件名称），指定Intent的目标组件的类名称。
2. action（动作）,指定Intent的执行动作，比如调用拨打电话组件。
3. data（数据），起到表示数据和数据MIME类型的作用。不同的action是和不同的data类型配套的，通过设置data的Uri来获得。
4. category（类别），被执行动作的附加信息。例如应用的启动Activity在intent-filter中设置category。

（备注：action只能有一个，category至少有一个）

1. extras（附加信息），为处理Intent组件提供附加的信息。可通过putXX()和getXX()方法存取信息；也可以通过创建Bundle对象，再通过putExtras()和getExtras()方法来存取。
2. flags（标记），指示Android如何启动目标Activity，设置方法为调用Intent的setFlags方法。

7.type（类型），基本和data类似。只是在一般的情况下。Data和type中只能有一个使用。Data有值的话，type就为null

第一类：启动，有ComponentName（显式）,Action（隐式），Category（隐式）。

第二类：传值，有Data（隐式），Type（隐式），Extra（隐式、显式）。

第三类：启动模式，有Flag。

**Intent的两种使用方法？**

1.显式方式。直接设置目标组件的ComponentName，用于一个应用内部的消息传递，比如启动另一个Activity或者一个services。  
通过Intent的setComponent和setClass来制定目标组件的ComponentName。

2.隐式方式。ComponentName为空，用于调用其他应用中的组件。需要包含足够的信息，这样系统才能根据这些信息使用intent filter在所有的组件中过滤action、data或者category来匹配目标组件。

**Intent可以传递的数据类型？**

1.简单或基本数据类型  
2.传递一个Bundle  
3.传递Serializable对象  
4.Parcelable对象  
5.Intent

**DVM进程，Linux进程，应用程序进程是否是同一个概念？**

答：DVM进程就是Dalivk虚拟机进程，而每一个应用程序进程就是一个虚拟机实例，同时每一个应用程序进程都会被系统分配一个LinuxId.所以这三者是同一个概念。

**同一个程序，但不同的Activity是否可以放在不同的Task任务栈中？**

可以放在不同的Task中。需要为不同的activity设置不同的affinity属性，启动activity的Intent需要包含FLAG\_ACTIVITY\_NEW\_TASK标记。

**程序之间的亲和性的理解。**

1、默认情况下一个应用的所有Activity都是具有相同的affinity，都是从application中继承，application的affinity默认就是manifest的包名。

2、affinity对Activity来说，就像是身份证一样，可以告诉所在的Task，自己属于其中的一员。

**AsyncTask使用在哪些场景？它的缺陷是什么？如何解决？**

运用场景:我们需要进行一些耗时的操作，耗时操作完成后更新主线程，或者在操作过程中对主线程的UI进行更新。

缺陷：AsyncTask中维护着一个长度为128的线程池，同时可以执行5个工作线程，还有一个缓冲队列，当线程池中已有128个线程，缓冲队列已满时，如果此时向线程提交任务，将会抛出RejectedExecutionException。

解决：由一个控制线程来处理AsyncTask的调用判断线程池是否满了，如果满了则线程睡眠否则请求AsyncTask继续处理

**内存溢出和内存泄漏有什么区别？何时会产生内存泄漏？内存优化有哪些方法？**

内存溢出通俗理解就是软件（应用）运行需要的内存，超出了它可用的最大内存。

内存泄漏就是我们对某一内存空间的使用，使用完成后没有释放。

内存优化：Android中容易内存溢出的部分，就是图片的加载，我们可以使用图片的压缩加上使用缓存的目的来控制图片所能够使用的内存。还有对于比较耗资源的对象及时的关闭，数据库的Conn及时关闭，listview的优化，取消注册广播接收器，及时关闭I/O流,Bitmap的recyle等

**Java/Android基于多线程的文件下载的实现？**

1. 对于网络上的一个资源，首先发送一个请求，从返回的Content-Length中得到需要下载文件的大小，然后根据文件大小创建一个文件。
2. 根据线程数和文件大小，为每个线程分配下载的字节区间，然后每个线程向服务器发送请求，获取这段字节区间的文件内容。
3. 利用RandomAccessFile的seek方法，多线程同时往一个文件中写入字节。

**Thread,AsyncTask和IntentService的使用场景和特点？**

**Thread:执行耗时或者可能发生阻塞的操作，而且基本不和UI线程发生交互的。它的特点就是操作方便。**

**AsyncTask:** 需要进行一些耗时的操作，耗时操作完成后更新主线程，或者在操作过程中对主线程的UI进行更新。**它的特点是线程可控，操作简单，缺点是代码量大。**

**IntentService:需要在后台持续运行，而且得起工作线程的操作。要知道service的优先级要比Activity和Thread的高，也就是这两者被销毁的时候service是不会被销毁的。Thrad在Actiivty中使用的话，一旦activity不可见的话这个线程就有可能在内存不足的情况下被杀死。Intentservice会创建独立的线程黑醋栗intent的请求，而且他会在任务完成之后自动停止。**

**五大布局的特点和绘制效率？**

**FrameLayout(框架布局):** 框架布局是最简单的布局形式。所有添加到这个布局中的视图都以层叠的方式显示。第一个添加的控件被放在最底层，最后一个添加到框架布局中的视图显示在最顶层，上一层的控件会覆盖下一层的控件。这种显示方式有些类似于堆栈。

**AbsoluteLayout(绝对布局)**:绝对布局的子控件需要指定相对于此坐标布局的横纵坐标值，否则将会像框架布局那样被排在左上角。手机应用需要适应不同的屏幕大小，而这种布局模型不能自适应屏幕尺寸大小，所以应用的相对较少。

**RelativeLayout(相对布局)**:相对布局的子控件会根据它们所设置的参照控件和参数进行相对布局。参照控件可以是父控件，也可以是其它子控件，但是被参照的控件必须要在参照它的控件之前定义.。

**TableLayout(表格布局)**:表格布局模型以行列的形式管理子控件，每一行为一个TableRow的对象，当然也可以是一个View的对象。TableRow可以添加子控件，每添加一个为一列。

**LinearLayout(线性布局)**:在一个方向上(垂直或水平)对齐所有子元素，一个垂直列表每行将只有一个子元素(无论它们有多宽)，一个水平列表只是一列的高度（最高子元素的高度来填充）。

Android的布局优化？  
1，使用 style 主题来定义一个通用的属性，从而重复利用代码，减少代码量。

1. 减少布局嵌套的层次。
2. 使用 LinearLayoutCompat 组件来实现线性布局元素之间的分割线，从而减少了使用View来实现分割线效果。
3. 使用 include 标签加载底部菜单栏布局，include 标签的目的是重复利用布局，来减少代码。
4. 使用懒加载ViewStub,在需要额时候再加载。

Android五种数据存储格式

**Sharedpreference**:通常用于一些配置文件的保存，提供了android基本类型的保存。它的本质是基于XML的key-value键值对的存储。操作简单但是只能存储基本数据类型而且无法进行条件查询。SharedPreferences对象本身只能获取数据而不支持存储和修改，存储修改是通过Editor对象实现。缺点是只能在一个包里共享数据，多个应用程序间无法共享数据。

实现SharedPreferences存储的步骤如下：

一、根据Context获取SharedPreferences对象

二、利用edit()方法获取Editor对象。

三、通过Editor对象存储key-value键值对数据。

四、通过commit()方法提交数据。

**File：**可以实现大数量数据的存储，但是，只可以访问本地资源而且文件更新或者格式更新要再编程。

**网络存储**：

**SQLite：**轻量级数据库，操作简单，利用SQLiteDatabase的类就可以对数据进行CRUD

**ContentProvider：**提供了应用程序间数据存储和修改的统一接口。

Android的MVC和MVP？

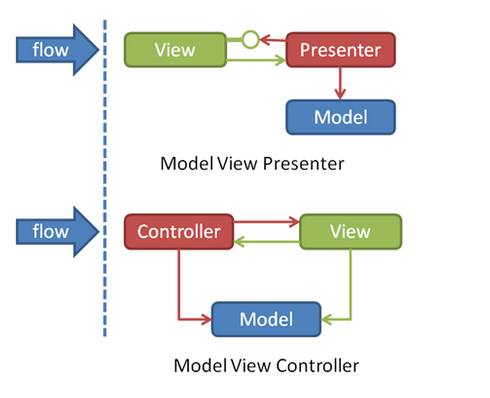
* View：对应于布局文件
* Model：业务逻辑和实体模型
* Controllor：对应于Activity

View层对应于各种布局文件，但是布局文件所能做的东西真的很有限。而实际上关于该布局文件的事件绑定和事件处理都是靠Activity来实现的，所以就会使Acitity既像View又像Controllor。

而当将架构改为MVP以后，Presenter的出现，将Actvity视为View层，Presenter负责完成View层与Model层的交互。现在是这样的：

* View 对应于Activity，负责View的绘制以及与用户交互
* Model 依然是业务逻辑和实体模型
* Presenter 负责完成View于Model间的交互

这次的跳跃是从并不标准的MVC到MVP的一个转变，减少了Activity的职责，简化了Activity中的代码，将复杂的逻辑代码提取到了Presenter中进行处理。与之对应的好处就是，耦合度更低，更方便的进行测试。



以上可以很明显的看出来：MVC中允许Model和View的交互，而在MVP中Model和View的交互完全是靠Presenter实现的，而且View与Presenter之间是靠接口实现的。

**MVP的优势**

1. 模型与视图完全分离，我们可以修改视图而不影响模型  
   2、可以更高效地使用模型，因为所以的交互都发生在一个地方——Presenter内部  
   3、我们可以将一个Presener用于多个视图，而不需要改变Presenter的逻辑。这个特性非常的有用，因为视图的变化总是比模型的变化频繁。  
   4、如果我们把逻辑放在Presenter中，那么我们就可以脱离用户接口来测试这些逻辑（单元测试）。

Include、Merge和ViewStub的作用？

1. 使用[include](http://www.111cn.net/tags.php/include/)标签将可复用的组件抽取出来(引用布局)  
   二、使用merge标签减少布局的嵌套层次（merge相当于framelayout）  
   场景1：布局根结点是FrameLayout且不需要设置background或padding等属性，可以用merge代替。  
   场景2：某布局作为子布局被其他布局include时，使用merge当作该布局的顶节点，这样在被引入时，顶结点会自动被忽略。  
   三、使用ViewStub标签来加载一些不常用的布局  
   作用：ViewStub标签同include标签一样可以用来引入一个外部布局，不同的是，ViewStub引入的布局默认不会扩张，既不会占用显示也不会占用位置，从而在解析Layout时节省cpu和内存。

JSON有什么优劣势？

**XML的优点**  
　　A.格式统一，符合标准；  
　　B.容易与其他系统进行远程交互，数据共享比较方便。

**XML的缺点**  
　　A.XML文件庞大，文件格式复杂，传输占带宽；  
　　B.服务器端和客户端都需要花费大量代码来解析XML，导致服务器端和客户端代码变得异常复杂且不易维护；  
　　C.客户端不同浏览器之间解析XML的方式不一致，需要重复编写很多代码；  
　　D.服务器端和客户端解析XML花费较多的资源和时间。

**JSON的优点**：

　　A.数据格式比较简单，易于读写，格式都是压缩的，占用带宽小；  
　　B.易于解析，客户端JavaScript可以简单的通过eval()进行JSON数据的读取；  
　　C.支持多种语言，包括ActionScript, C, C#, ColdFusion, Java, JavaScript, Perl, PHP, Python, Ruby等服务器端语言，便于服务器端的解析；  
　　D.在PHP世界，已经有PHP-JSON和JSON-PHP出现了，偏于PHP序列化后的程序直接调用，PHP服务器端的对象、数组等能直接生成JSON格式，便于客户端的访问提取；  
　　E.因为JSON格式能直接为服务器端代码使用，大大简化了服务器端和客户端的代码开发量，且完成任务不变，并且易于维护。

**JSON的缺点**

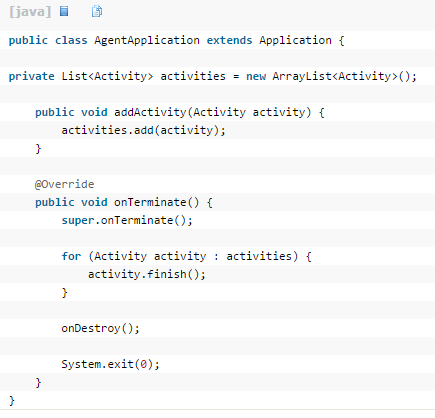
A.没有XML格式这么推广的深入人心和喜用广泛，没有XML那么通用性；  
　　B.JSON格式目前在Web Service中推广还属于初级阶段。

两种动画的特点？

逐帧动画：用一组图片轮流绘制，产生动画的感觉。类似于gif图片。

补间动画：过改变view的大小、旋转的角度、透明度、位置来产生动画，可以控制动画的播放快慢以及加速度。对同一个物体（绘制的一个view或者图片）进行操作产生动画。缺点是我们只能设置动画开始和结束两帧，而中间的过程则是由系统帮我们完成的。所以游戏中帧数比较多的时候是不实用的。（Scale,Alpha,Rotate,Translste）

退出终止APP的运行？

将每个Acivtity都添加到一个Application维护的List集合中，每个Activty创建之后都添加到这个集合中。在退出终止的时候就调用Application的onTerminate方法。Finish掉集合中的每个Activity，然后调用System.exit(0);例如：

Android中的Asset和Res文件夹的区别？

**\*res/raw和assets的相同点：**

1.两者目录下的文件在打包后会原封不动的保存在apk包中，不会被编译成二进制。  
**\*res/raw和assets的不同点：**  
1.res/raw中的文件会被映射到R.java文件中，访问的时候直接使用资源ID即R.id.filename；assets文件夹下的文件不会被映射到R.java中，访问的时候需要AssetManager类。  
2.res/raw不可以有目录结构，而assets则可以有目录结构，也就是assets目录下可以再建立文件夹

Android怎么加速启动Activity？

减少在onCreate中的代码，将其一部分在onResume()中写出来。还有对组件的初始化要按需加载。即最好的是用上include,merge和ViewStub.

#### Android的内存优化方法？

**OOM引发的原因**

1. **有限的堆内存，原始只有16M**
2. **内存消耗的大小根据多方面的因素：操作系统，屏幕尺寸等**
3. **后台的多任务**
4. **运行在虚拟机上**

**OOM产生的影响？**

1. **程序卡顿，响应速度慢**
2. **莫名消失（当你的程序所占用的内存越大在后台就越容易被回收。反之，内存越小就会生存的时间越长）**
3. **直接崩溃（OOM）**

**OOM的解决办法？（5R）**

**Reckon（计算内存大小）**

**系统内存和堆内存（MAT内存分析工具）**

**Reduce（减少内存的使用）**

**主要是对Bitmap的使用**

1. **显示：按需加载图片大小。在列表中加载缩略图，在用户单击条目的时候显示整个图片。**
2. **大小：直接使用原来的图片会耗用大的内存。所以最好的就是利用BitmapFactory.Options来减少对系统资源的需求。属性值inSampleSize表示的是缩略图是原来图片尺寸的几分之几。**

**( 3) 像素：Android中的图片有四种属性，分别是：**

**ALPHA\_8：**每个像素占用1byte内存   
**ARGB\_4444：**每个像素占用2byte内存   
**ARGB\_8888：**每个像素占用4byte内存 （默认）  
**RGB\_565：**每个像素占用2byte内存

Android默认的颜色模式为ARGB\_8888，这个颜色模式色彩最细腻，显示质量最高。但同样的，占用的内存也最大。 所以在对图片效果不是特别高的情况下使用RGB\_565（565没有透明度属性）

**（4）回收：使用Bitmap过后，就需要及时的调用Bitmap.recycle()方法来释放Bitmap占用的内存空间，而不要等Android系统来进行释放。**

**（5）捕捉异常**

**修改对象引用类型**

**四大引用类型**

**强引用（strong reference）**  
如：Object object=new Object（），object就是一个强引用了。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

**软引用（SoftReference）**  
只有内存不够时才回收,常用于缓存；当内存达到一个阀值，GC就会去回收它；

**弱引用（WeakReference）**

弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

**虚引用（PhantomReference）**

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

**软引用和弱引用的应用实例：**

注意：对于SoftReference(软引用)或者WeakReference(弱引用)的Bitmap缓存方案，现在已经不推荐使用了。自Android2.3版本(API Level 9)开始，垃圾回收器更着重于对软/弱引用的回收。

**其他的小tips**

1. **对常量使用static final修饰**

因为在成员变量初始化的时候，会将常量直接保存到类文件中。用到intVal的代码被直接替换成42，而使用strVal的会指向一个字符串常量，而不是使用成员变量。

1. **多使用实体类（减少使用接口）**HashMap map = new HashMap();

事实证明调用一个接口的引用会比调用实体类的引用多花费一倍的时间。

1. **少用浮点数**

在Android设备中，浮点数比整形慢两倍。

1. 减少不必要的全局变量
2. 避免使用枚举
3. for循环中避免方法的调用（for-each比for循环慢）
4. 了解并使用高效的类库

处理字串的时候，不要吝惜使用**String.indexOf()**，**String.lastIndexOf()**等特殊实现的方法。这些方法都是使用C/C++实现的，比起Java循环快10到100倍。**System.arraycopy**方法在有JIT的Nexus One上，自行编码的循环快9倍。

1. **Reuse（重用）**

**bitmap缓存**

Bitmap缓存分为两种：

一种是内存缓存，一种是硬盘缓存。

**内存缓存（LruCache）：**

以牺牲宝贵的应用内存为代价，内存缓存提供了快速的Bitmap访问方式。系统提供的LruCache类是非常适合用作缓存Bitmap任务的，它将最近被引用到的对象存储在一个强引用的LinkedHashMap中，并且在缓存超过了指定大小之后将最近不常使用的对象释放掉。

注意：以前有一个非常流行的内存缓存实现是SoftReference(软引用)或WeakReference(弱引用)的Bitmap缓存方案，然而现在已经不推荐使用了。自Android2.3版本(API Level 9)开始，垃圾回收器更着重于对软/弱引用的回收，这使得上述的方案相当无效。

**硬盘缓存（DiskLruCache）：**

一个内存缓存对加速访问最近浏览过的Bitmap非常有帮助，但是你不能局限于内存中的可用图片。GridView这样有着更大的数据集的组件可以很轻易消耗掉内存缓存。你的应用有可能在执行其他任务(如打电话)的时候被打断，并且在后台的任务有可能被杀死或者缓存被释放。一旦用户重新聚焦(resume)到你的应用，你得再次处理每一张图片。

在这种情况下，硬盘缓存可以用来存储Bitmap并在图片被内存缓存释放后减小图片加载的时间(次数)。当然，从硬盘加载图片比内存要慢，并且应该在后台线程进行，因为硬盘读取的时间是不可预知的。

**图片缓存的开源项目：**

对于图片的缓存现在都倾向于使用开源项目，这里我列出几个我搜到的：

**1. Android-Universal-Image-Loader 图片缓存**

目前使用最广泛的图片缓存，支持主流图片缓存的绝大多数特性。  
项目地址：https://github.com/nostra13/Android-Universal-Image-Loader

**2. picasso square开源的图片缓存**  
项目地址：https://github.com/square/picasso  
特点：(1)可以自动检测adapter的重用并取消之前的下载  
(2)图片变换  
(3)可以加载本地资源  
(4)可以设置占位资源  
(5)支持debug模式

**3. ImageCache 图片缓存，包含内存和Sdcard缓存**  
项目地址：https://github.com/Trinea/AndroidCommon  
特点：

(1)支持预取新图片，支持等待队列  
(2)包含二级缓存，可自定义文件名保存规则  
(3)可选择多种缓存算法(FIFO、LIFO、LRU、MRU、LFU、MFU等13种)或自定义缓存算法  
(4)可方便的保存及初始化恢复数据  
(5)支持不同类型网络处理  
(6)可根据系统配置初始化缓存等

**4. Android 网络通信框架Volley**

项目地址：https://android.googlesource.com/platform/frameworks/volley

我们在程序中需要和网络通信的时候，大体使用的东西莫过于AsyncTaskLoader，HttpURLConnection，AsyncTask，HTTPClient（Apache）等，在2013年的Google I/O发布了Volley。Volley是Android平台上的网络通信库，能使网络通信更快，更简单，更健壮。

特点：

(1)JSON，图像等的异步下载；

(2)网络请求的排序（scheduling）

(3)网络请求的优先级处理

(4)缓存

(5)多级别取消请求

(6)和Activity和生命周期的联动（Activity结束时同时取消所有网络请求）

**Adapter适配器**

在Android中Adapter使用十分广泛，特别是在list中。所以adapter是数据的**“集散地”** ，所以对其进行内存优化是很有必要的。

下面算是一个标准的使用模版：

主要使用convertView和ViewHolder来进行缓存处理

**[java]** [view plaincopy](http://blog.csdn.net/a396901990/article/details/38707007)

1. @Override
2. **public** View getView(**int** position, View convertView, ViewGroup parent) {
3. ViewHolder vHolder = **null**;
4. //如果convertView对象为空则创建新对象，不为空则复用
5. **if** (convertView == **null**) {
6. convertView = inflater.inflate(..., **null**);
7. // 创建 ViewHodler 对象
8. vHolder = **new** ViewHolder();
9. vHolder.img= (ImageView) convertView.findViewById(...);
10. vHolder.tv= (TextView) convertView.findViewById(...);
11. // 将ViewHodler保存到Tag中(Tag可以接收Object类型对象，所以任何东西都可以保存在其中)
12. convertView.setTag(vHolder);
13. } **else** {
14. //当convertView不为空时，通过getTag()得到View
15. vHolder = (ViewHolder) convertView.getTag();
16. }
17. // 给对象赋值，修改显示的值
18. vHolder.img.setImageBitmap(...);
19. vHolder.tv.setText(...);
20. **return** convertView;
21. }
22. //将显示的View 包装成类
23. **static** **class** ViewHolder {
24. TextView tv;
25. ImageView img;
26. }

**池（PooL）**

**对象池：**

对象池使用的基本思路是：将用过的对象保存起来，等下一次需要这种对象的时候，再拿出来重复使用，从而在一定程度上减少频繁创建对象所造成的开销。 并非所有对象都适合拿来池化――因为维护对象池也要造成一定开销。对生成时开销不大的对象进行池化，反而可能会出现“维护对象池的开销”大于“生成新对象的开销”，从而使性能降低的情况。但是对于生成时开销可观的对象，池化技术就是提高性能的有效策略了。

**线程池：**

线程池的基本思想还是一种对象池的思想，开辟一块内存空间，里面存放了众多(未死亡)的线程，池中线程执行调度由池管理器来处理。当有线程任务时，从池中取一个，执行完成后线程对象归池，这样可以避免反复创建线程对象所带来的性能开销，节省了系统的资源。

比如：一个应用要和网络打交道，有很多步骤需要访问网络，为了不阻塞主线程，每个步骤都创建个线程，在线程中和网络交互，用线程池就变的简单，线程池是对线程的一种封装，让线程用起来更加简便，只需要创一个线程池，把这些步骤像任务一样放进线程池，在程序销毁时只要调用线程池的销毁函数即可。

java提供了**ExecutorService**和**Executors**类，我们可以应用它去建立线程池。

**（4）Recycle（回收）**

**（5）Review（检查）**

**软引用和弱引用的应用场景**

**软引用：如果一个对象具有软引用的话，只要内存空间足够的话就不会被回收；如果内存空间不够的情况下才会收回这些对象的空间。只要GC没有回收它就不会影响它的使用。如引用可以用来实现内存敏感的高速缓存。**

**使用流程：**

1. **首先定义一个HashMap来保存软引用对象**

Private HashMap<String,softReference<Bitmap>> hashmap = new HashMap<String,softReference<Bitmsp>>;

1. **定义一个方法保存bitmap的软引用到hashmap中**

public void addBitmapToCache(String path) {

// 强引用的Bitmap对象

        Bitmap bitmap = BitmapFactory.decodeFile(path);

        // 软引用的Bitmap对象

        SoftReference<Bitmap> softBitmap = new SoftReference<Bitmap>(bitmap);

        // 添加该对象到Map中使其缓存

        imageCache.put(path, softBitmap);

    }

1. **获取的时候通过softReference.get()方法获取bitmap对象（注意：有可能bitmap的软引用在内存不足的情况下被回收，所以就要先判空再取）**

public Bitmap getBitmapByPath(String path) {

        // 从缓存中取软引用的Bitmap对象

        SoftReference<Bitmap> softBitmap = imageCache.get(path);

        // 判断是否存在软引用

        if (softBitmap == null) {

            return null; }

        // 取出Bitmap对象，如果由于内存不足Bitmap被回收，将取得空

        Bitmap bitmap = softBitmap.get();

        return bitmap;

    }

**弱引用：**

如果一个对象只具有弱引用的话，那么垃圾回收器在扫描的过程中一旦发现某个对象只具有弱引用的话就会回收这块对象的内存。但是一般垃圾回收器的优先级是比较低的。所以就不会那么那么快的发现这个弱引用。

备注：弱引用与软引用的根本区别在于：只具有弱引用的对象拥有更短暂的生命周期，可能随时被回收。而只具有软引用的对象只有当内存不够的时候才被回收，在内存足够的时候，通常不被回收。

到底什么时候使用软引用，什么时候使用弱引用呢？

1）如果只是想避免OutOfMemory异常的发生，则可以使用软引用。

如果对于应用的性能更在意，想尽快回收一些占用内存比较大的对象，则可以使用弱引用。

1. 还有就是可以根据对象是否经常使用来判断。如果该对象可能会经常使用的，就尽量用软引用。如果该对象不被使用的可能性更大些，就可以用弱引用。

#### 有关View与ViewGroup

1. View与ViewGroup的关系

View是所有UI组件的基类；而ViewGroup是所有组件(View的组件和ViewGroup)的容器，其本身也是从View派生出来的。ViewGroup是“layout”的基类。

View 派生出来的直接或间接子类：ImageView, Button, CheckBox, SurfaceView, TextView, ViewGroup, AbsListView

ViewGourp 派生出来的直接或间接子类：AbsoluteLayout, FrameLayout, RelativeLayout, LinearLayout

1. 自定义View的步骤

1，绘图，通过重写onDraw方法控制View在屏幕上的渲染效果

2，交互，通过重写onTouchEvent方法或者使用手势来控制用户的交互   
3，测量，通过重写onMeasure方法来对控件进行测量   
4，属性，可以通过xml自定义控件的属性，然后通过TypedArray来进行使用   
5，状态的保存，为了避免配置改变时丢失View状态，通过重写onSaveInstanceState，onRestoreInstanceState方法来保存和恢复状态

1. View的内部过程（自上而下的过程，不断的循环）
2. onMessure():用于测量视图的大小
3. onLayout():用于测量视图的位置
4. onDraw():具体的绘制过程：
   1. View的背景绘制 ：根据layout过程确定的视图位置来设置背景的绘制区域，之后再调用Drawable的draw()方法来完成背景的绘制工作。 主要是根据XML中通过android:background属性
   2. View自身内容的绘制：onDraw()
   3. View的子视图的绘制：dispatchDraw()
   4. View的渐变框和滚动条的绘制：onDrawScrollBars()

重点关注View本身内容的绘制和子视图的绘制，即onDraw()和dispatchDraw()函数。

对于View.java和ViewGroup.java，onDraw()默认都是空实现，也就是View的具体形状是由我们开发者自己去实现的。View.java中dispatchDraw()默认为空实现，因为其不包含子视图，而ViewGroup重载了dispatchDraw()来对其子视图进行绘制，通常应用程序不应该对dispatchDraw()进行重载，其默认实现体现了View系统绘制的流程。dispatchDraw()的核心代码就是通过for循环调用drawChild()对ViewGroup的每个子视图进行绘制。

#### Touch事件分发

# [Android 编程下 Touch 事件的分发和消费机制](http://www.cnblogs.com/sunzn/archive/2013/05/10/3064129.html)

Android 中与 Touch 事件相关的方法包括：dispatchTouchEvent(MotionEvent ev)、onInterceptTouchEvent(MotionEvent ev)、onTouchEvent(MotionEvent ev)；能够响应这些方法的控件包括：ViewGroup 及其子类、Activity。方法与控件的对应关系如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| **Touch 事件相关方法** | **方法功能** | **ViewGroup** | **Activity** |
| public boolean dispatchTouchEvent(MotionEvent ev) | 事件分发 | Yes | Yes |
| public boolean onInterceptTouchEvent(MotionEvent ev) | 事件拦截 | Yes | No |
| public boolean onTouchEvent(MotionEvent ev) | 事件响应 | Yes | Yes |

从这张表中我们可以看到 ViewGroup 及其子类对与 Touch 事件相关的三个方法均能响应，而 Activity 对 onInterceptTouchEvent(MotionEvent ev) 也就是事件拦截不进行响应。另外需要注意的是 View 对 dispatchTouchEvent(MotionEvent ev) 和 onInterceptTouchEvent(MotionEvent ev) 的响应的前提是可以向该 View 中添加子 View，如果当前的 View 已经是一个最小的单元 View（比如 TextView），那么就无法向这个最小 View 中添加子 View，也就无法向子 View 进行事件的分发和拦截，所以它没有dispatchTouchEvent(MotionEvent ev) 和 onInterceptTouchEvent(MotionEvent ev)，只有 onTouchEvent(MotionEvent ev)。

**一、Touch 事件分析**

**▐ 事件分发：public boolean dispatchTouchEvent(MotionEvent ev)**

Touch 事件发生时 Activity 的 dispatchTouchEvent(MotionEvent ev) 方法会以隧道方式（从根元素依次往下传递直到最内层子元素或在中间某一元素中由于某一条件停止传递）将事件传递给最外层 View 的 dispatchTouchEvent(MotionEvent ev) 方法，并由该 View 的 dispatchTouchEvent(MotionEvent ev) 方法对事件进行分发。dispatchTouchEvent 的事件分发逻辑如下：

* 如果 **return true**，事件会分发给当前 View 并由 dispatchTouchEvent 方法进行消费，同时事件会停止向下传递；
* 如果 **return false**，事件分发分为两种情况：

1. 如果当前 View 获取的事件直接来自 Activity，则会将事件返回给 Activity 的 onTouchEvent 进行消费；
2. 如果当前 View 获取的事件来自外层父控件，则会将事件返回给父 View 的  onTouchEvent 进行消费。

* 如果返回系统默认的 **super**.dispatchTouchEvent(ev)，事件会自动的分发给当前 View 的 onInterceptTouchEvent 方法。

**▐ 事件拦截：public boolean onInterceptTouchEvent(MotionEvent ev)**

在外层 View 的 dispatchTouchEvent(MotionEvent ev) 方法返回系统默认的 super.dispatchTouchEvent(ev) 情况下，事件会自动的分发给当前 View 的 onInterceptTouchEvent 方法。onInterceptTouchEvent 的事件拦截逻辑如下：

* 如果 onInterceptTouchEvent 返回 **true**，则表示将事件进行拦截，并将拦截到的事件交由当前 View 的 onTouchEvent 进行处理；
* 如果 onInterceptTouchEvent 返回 **false**，则表示将事件放行，当前 View 上的事件会被传递到子 View 上，再由子 View 的 dispatchTouchEvent 来开始这个事件的分发；
* 如果 onInterceptTouchEvent 返回 **super**.onInterceptTouchEvent(ev)，事件默认会被拦截，并将拦截到的事件交由当前 View 的 onTouchEvent 进行处理。

**▐ 事件响应：public boolean onTouchEvent(MotionEvent ev)**

在 dispatchTouchEvent 返回 super.dispatchTouchEvent(ev) 并且 onInterceptTouchEvent 返回 true 或返回 super.onInterceptTouchEvent(ev) 的情况下 onTouchEvent 会被调用。onTouchEvent 的事件响应逻辑如下：

* 如果事件传递到当前 View 的 onTouchEvent 方法，而该方法返回了 false，那么这个事件会从当前 View 向上传递，并且都是由上层 View 的 onTouchEvent 来接收，如果传递到上面的 onTouchEvent 也返回 false，这个事件就会“消失”，而且接收不到下一次事件。
* 如果返回了 true 则会接收并消费该事件。
* 如果返回 super.onTouchEvent(ev) 默认处理事件的逻辑和返回 false 时相同。

**互联网推送服务原理：长连接+心跳机制**

1. **维护任何一个长连接都需要心跳机制**，客户端发送一个心跳给服务器，服务器给客户端一个心跳应答，这样就形成客户端服务器的一次完整的握手，这个握手是让双方都知道他们之间的连接是没有断开，客户端是在线的。如果超过一个时间的阈值，客户端没有收到服务器的应答，或者服务器没有收到客户端的心跳，那么对客户端来说则断开与服务器的连接重新建立一个连接，对服务器来说只要断开这个连接即可。
2. **IOS和Android系统的推送有什么区别？**

首先，所有的推送都是建立在长连接的基础上的。‘

 IOS长连接是由系统来维护的，也就是说苹果的IOS系统在系统级别维护了一个客户端和苹果服务器的长链接，IOS上的所有应用上的推送都是先将消息推送到苹果的服务器然后将苹果服务器通过这个系统级别的长链接推送到手机终端上，这样的的几个好处为：

**1.**在手机终端始终只要维护一个长连接即可，而且由于这个长链接是系统级别的不会出现被杀死而无法推送的情况。

**2.**省电，不会出现每个应用都各自维护一个自己的长连接。

**3.**安全，只有在苹果注册的开发者才能够进行推送

**Android的长连接是每个应用自己维护的。虽然google也提出了自己的系统级别的长连接（C2DM和云端推送）但是由于中国无法访问，所以还是于事无补。这样就导致了每个android开发者如果想要实现自己的应用24小时在线的话就不得不去自己维护一个长连接。浪费的电量和流量是可想而知的。**

**3，推送的实现方式：**

一：客户端不断的查询服务器，检索新内容，也就是所谓的pull 或者轮询方式  
二：客户端和服务器之间维持一个TCP/IP长连接，服务器向客户端push  
三：服务器有新内容时，发送一条类似短信的信令给客户端，客户端收到后从服务器中下载新内容，也就是SMS的推送方式。

#### Zoygote的启动过程

Android的系统是基于Linux的，android中所有进程都是init进程的子进程。那么理所当然的Zygote进程也是由init进程fork出来的。下面是启动Zygote的命令参数：

service zygote /system/bin/app\_process -Xzygote /system/bin --zygote --start-system-server

这个命令就是执行的是app\_main.cpp中main函数。所以

app\_main.cpp中main()｛创建runtime实例，调用了runtime.start()｝🡪

{1,调用startVM创建虚拟机；2，调用startReg注册android Native方法；3，让虚拟机执行Zygote的main()方法}

🡪

{1，创建Zygote套接字用于监听AndroidManegerService发过来的fork请求；2，预加载classes和resouse；3，创建SystemServer进程；4，进入循环监听模式，监听fork请求}

[Android 之ActivityThead、ActivityManagerService 与activity的管理和创建](http://blog.csdn.net/xieqibao/article/details/6570080)

**ActivityThread:**

ActivityThread主要用来启动应用程序的主线程，并且管理在应用端跟用户打交道的activity。在应用端的activity信息全部被存储在ActivityThread的成员变量mActivities中。

**[java]** [view plain](http://blog.csdn.net/xieqibao/article/details/6570080) [copy](http://blog.csdn.net/xieqibao/article/details/6570080)

1. **final** HashMap<IBinder, ActivityRecord> mActivities= **new**  HashMap<IBinder, ActivityRecord>();

也就是说，在mActivities中，记录了应用程序创建的所有activity实例记录，对应的是ActivityRecord。

ActivityThread是怎么启动应用程序的呢？ActivityThread中有一个main函数，在这个里面，将启动应用程序并建立消息循环。在之前也介绍过，系统会为主线程自动创建消息循环。

在建立消息循环之前，会通过thread.attach(false)来初始化应用程序的运行环境，并建立activityThread和ActivityManagerService之间的桥mAppThread， mAppThread是IApplicationThread的一个实例。在建立消息循环之前，会通过thread.attach(false)来初始化应用程序的运行环境，并建立activityThread和ActivityManagerService之间的桥mAppThread， mAppThread是IApplicationThread的一个实例。每个应用程序对应着一个ActivityThread实例，应用程序由ActivityThread.main打开消息循环。每个应用程序同时也对应着一个ApplicationThread对象。该对象是activityThread和ActivityManagerService之间的桥梁。在attach中还做了一件事情，就是通过代理调用attachApplication，并利用binder的transact机制，在ActivityManagerService中建立了ProcessRecord信息。之后通过该ProcessRecord就可以获得该ActivityThread中的所有ActivityRecord记录。

**ActivityManagerService:**

在ActivityManagerService中，也有一个用来管理activity的地方：mHistory栈，这个mHistory栈里存放的是服务端的activity记录HistoryActivity（class HistoryRecord extendsIApplicationToken.Stub）。处于栈顶的就是当前running状态的activity。Activity.startActivity()方法最终是通过代理类和Binder机制，在ActivityManagerService.startActivity方法中执行的。

那么在ActivityManagerService的startActivity中主要做的事情就是：

* 1. 根据activity、ProcessRecord等信息创建HistoryRecord实例HistoryRecord  r；
  2. 把r加入到mHistory中。
  3. activity被加入到mHistory之后，只是说明在服务端可以找到该activity记录了，但是在客户端目前还没有该activity记录。还需要通过ProcessRecord中的thread（IApplication）变量，调用它的scheduleLaunchActivity方法在ActivityThread中创建新的ActivityRecord记录（之前我们说过，客户端的activity是用ActivityRecord记录的，并放在mActivities中）。
  4. 客户端根据服务端返回回来的信息创建客户端activity记录ActivityRecord. 并通过Handler发送消息到消息队列，进入消息循环。在ActivityThread.handleMessage()中处理消息。最终在handleLaunchActivity方法中把ActivityRecord记录加入到mActivities（mActivities.put(r.token,r)）中，并启动activity

**总结：**

1. 在客户端和服务端分别有一个管理activity的地方，服务端是在mHistory中，处于mHistory栈顶的就是当前处于running状态的activity，客户端是在mActivities中。
2. 在startActivity时，首先会在ActivityManagerService中建立HistoryRecord，并加入到mHistory中，然后通过scheduleLaunchActivity在客户端创建ActivityRecord记录并加入到mActivities中。最终在ActivityThread.main()发起请求，进入消息循环，完成activity的启动和窗口的管理等。