# Android 日常使用技巧合集

* 1. 自定义View

自定义View属性引入 xmlns:hyman=<http://schemas.android.com/apk/res-auto>

**生命周期：**

1、onFinishInflate() 当View中所有的子控件均被映射成xml后触发 。

2、onMeasure( int , int ) 确定所有子元素的大小 。

3、onLayout( boolean , int , int , int , int ) 当View分配所有的子元素的大小和位置时触发 。

4、onSizeChanged( int , int , int , int ) 当view的大小发生变化时触发 。

5、onDraw(Canvas) view渲染内容的细节。

6、onKeyDown( int , KeyEvent) 有按键按下后触发 。

7、onKeyUp( int , KeyEvent) 有按键按下后弹起时触发 。

8、onTrackballEvent(MotionEvent) 轨迹球事件 。

9、onTouchEvent(MotionEvent) 触屏事件 。

10、onFocusChanged( boolean , int , Rect) 当View获取或失去焦点时触发 。

11、onWindowFocusChanged( boolean ) 当窗口包含的view获取或失去焦点时触发 。

12、onAttachedToWindow() 当view被附着到一个窗口时触发 。

13、onDetachedFromWindow() 当view离开附着的窗口时触发，Android123提示该方法和 onAttachedToWindow() 是相反的。

14、onWindowVisibilityChanged( int ) 当窗口中包含的可见的view发生变化时触发。

如果要设计到大量频繁的绘制操作，请继承surfaceview：

Android系统中，有一种特殊的视图，称为SurfaceView，它拥有独立的绘图表面，即它不与其宿主窗口共享同一个绘图表面。由于拥有独立的绘图表面，因此SurfaceView的UI就可以在一个独立的线程中进行绘制。又由于不会占用主线程资源，SurfaceView一方面可以实现复杂而高效的UI，另一方面又不会导致用户输入得不到及时响应。

* 1. **Fragment**

### 1.2.1、Fragment介绍以及生命周期

Android运行在各种各样的设备中，有小屏幕的手机，超大屏的平板甚至电视。针对屏幕尺寸的差距，很多情况下，都是先针对手机开发一套App，然后拷贝一份，修改布局以适应平板神马超级大屏的。难道无法做到一个App可以同时适应手机和平板么，当然了，必须有啊。Fragment的出现就是为了解决这样的问题。你可以把Fragment当成Activity的一个界面的一个组成部分，甚至Activity的界面可以完全有不同的Fragment组成，更帅气的是Fragment拥有自己的生命周期和接收、处理用户的事件，这样就不必在Activity写一堆控件的事件处理的代码了。更为重要的是，你可以动态的添加、替换和移除某个Fragment。

Fragment的生命周期

Fragment必须是依存与Activity而存在的，因此Activity的生命周期会直接影响到Fragment的生命周期。官网这张图很好的说明了两者生命周期的关系：



可以看到Fragment比Activity多了几个额外的生命周期回调方法：

onAttach(Activity)

当Fragment与Activity发生关联时调用。

onCreateView(LayoutInflater, ViewGroup,Bundle)

创建该Fragment的视图

onActivityCreated(Bundle)

当Activity的onCreate方法返回时调用

onDestoryView()

与onCreateView想对应，当该Fragment的视图被移除时调用

onDetach()

与onAttach相对应，当Fragment与Activity关联被取消时调用

注意：除了onCreateView，其他的所有方法如果你重写了，必须调用父类对于该方法的实现，

### 1.2.2 ragment和v4包中的fragment的使用的区别

1、尽量不要用app包中的fragment，因为这个是在3.0之后才有的，支持的版本太高，在低版本中是是用不了的

2、android.support.v4.app.Fragment：可以兼容到1.6的版本，

3、关于这两个fragment使用<fragment>标签的问题

（1）app.fragment和v4.fragment都是可以使用<fragment>标签的

只是在在使用的时候如果是app.fragment则没有什么特殊的地方继承Activity即可。

（2）当v4.fragment使用<fragment>标签的时候就要特别注意了：

当这个Activity的布局中有<fragment>标签的时候，这个Activity必须继承

FragmentActivity，否则就会报错

此时如果不继承FragmentActivity的话 编译系统会把<fragment>认为是app包中的Fragment来处理。但是此时我们导入的是v4包中的Fragment

Android官方文档中的Fragment的例子就是以app包中的Fragment来讲解的。

（3）app包中关于Fragment的类和方法在V4包中都是有相应的对应的

使用v4包时：需要把getFragmentManager( )改成getSupportFragmentManager( )

### 1.2.3 Fragment的使用

静态的使用Fragment

这是使用Fragment最简单的一种方式，把Fragment当成普通的控件，直接写在Activity的布局文件中。步骤：

1、继承Fragment，重写onCreateView决定Fragemnt的布局

2、在Activity中声明此Fragment，就当和普通的View一样

<**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="fill\_parent"  
 android:layout\_height="fill\_parent"  
 android:orientation="horizontal"**>  
  
<**fragment  
 android:id="@+id/left\_fragment"  
 android:name="com.sunflower.LeftFragment"  
 android:layout\_width="match\_parent"  
 android:layout\_height="fill\_parent"  
 android:layout\_weight="3"**/>  
  
<**fragment  
 android:id="@+id/right\_fragment"  
 android:name="com.sunflower.RightFragment"  
 android:layout\_width="match\_parent"  
 android:layout\_height="fill\_parent"  
 android:layout\_weight="1"**/>  
  
</**LinearLayout**>

动态的使用Fragment

1. FragmentManager fm = getFragmentManager();//v4 getSupportFragmentManager( )
2. FragmentTransaction transaction = fm.beginTransaction();
3. mWeixin = **new** ContentFragment();
4. transaction.replace(R.id.id\_content, mWeixin);
5. transaction.commit();

### 1.2.4 Fragment 通信

**与activity通讯**

尽管fragment的实现是独立于activity的，可以被用于多个activity，但是每个activity所包含的是同一个fragment的不同的实例。

Fragment可以调用getActivity()方法很容易的得到它所在的activity的对象，然后就可以查找activity中的控件们（findViewById()）。例如：ViewlistView =getActivity().findViewById(R.id.list);同样的，activity也可以通过FragmentManager的方法查找它所包含的frament们。例如：

ExampleFragment fragment =(ExampleFragment)getFragmentManager().findFragmentById(R.id.example\_fragment

**activity响应fragment的事件**

有时，你可能需要fragment与activity共享事件。一个好办法是在fragment中定义一个回调接口，然后在activity中实现之。例如，还是那个新闻程序的例子，它有一个activity，activity中含有两个fragment。fragmentA显示新闻标题，fragmentB显示标题对应的内容。fragmentA必须在用户选择了某个标题时告诉activity，然后activity再告诉fragmentB，fragmentB就显示出对应的内容（为什么这么麻烦？直接fragmentA告诉fragmentB不就行了？也可以啊，但是你的fragment就减少了可重用的能力。现在我只需把我的事件告诉宿主，由宿主决定如何处置，这样是不是重用性更好呢？）。如下例，OnArticleSelectedListener接口在fragmentA中定义：

1. **public** **static** **class** FragmentA **extends** ListFragment{
2. ...
3. //Container Activity must implement this interface
4. **public** **interface** OnArticleSelectedListener{
5. **public** **void** onArticleSelected(Uri articleUri);
6. }
7. ...

然后activity实现接口OnArticleSelectedListener，在方法onArticleSelected()中通知fragmentB。当fragment添加到activity中时，会调用fragment的方法onAttach()，这个方法中适合检查activity是否实现了OnArticleSelectedListener接口，检查方法就是对传入的activity的实例进行类型转换，如下所示：

1. **public** **static** **class** FragmentA **extends** ListFragment{
2. OnArticleSelectedListener mListener;
3. ...
4. @Override
5. **public** **void** onAttach(Activity activity){
6. **super**.onAttach(activity);
7. **try**{
8. mListener =(OnArticleSelectedListener)activity;
9. }**catch**(ClassCastException e){
10. **throw** **new** ClassCastException(activity.toString()+"must implement OnArticleSelectedListener");
11. }
12. }
13. ...

### 1.2.5 fragment界面重叠问题

原因：为什么会出现界面重叠呢？因为每当我们离开Activity的时候，切换到别的APP的时候，当内存不够用，Fragment所在Activity被销毁，会调用onSaveInstanceState()方法，Fragment都会被保存起来，当我再次回到这个app的时候，通过onCreate中的参数savedInstanceState恢复了之前的fragment，就导致了界面重叠。

在进入onCreate函数时，先去判断savedInstanceState是否为null，如果不为null,则表示里面有保存这个fragment。则不再重新去add这个fragment，而是通过Tag从前保存的数据中直接去读取，看一下代码：

**protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.activity);  
  
 TargetFragment targetFragment;  
 HideFragment hideFragment;  
  
 **if** (savedInstanceState != **null**) { *// “内存重启”时调用* List<Fragment> fragmentList = getSupportFragmentManager().getFragments();  
 **for** (Fragment fragment : fragmentList) {  
 **if**(fragment **instanceof** TartgetFragment){  
 targetFragment = (TargetFragment)fragment;  
 }**else if**(fragment **instanceof** HideFragment){  
 hideFragment = (HideFragment)fragment;  
 }  
 }  
 *// 解决重叠问题* getFragmentManager().beginTransaction()  
 .show(targetFragment)  
 .hide(hideFragment)  
 .commit();  
 }**else**{ *// 正常时* targetFragment = TargetFragment.newInstance();  
 hideFragment = HideFragment.newInstance();  
  
 *// 这里add时，tag可传可不传* getFragmentManager().beginTransaction()  
 .add(R.id.container)  
 .add(R.id,container,hideFragment)  
 .hide(hideFragment)  
 .commit();  
 }  
 }

### 1.2.6 Fragment中getActivity()的空指针问题

这个问题的原因大部分在于Fragment已经和所在activity解除了关联，也就是调用了onDetach()方法。

解决方法：

我们可以在Fragment中定义Activity全局变量，当Activity和Fragment关联的时候，给Activity赋值。

**public void** onAttach(Context context) {  
 **super**.onAttach(context);  
 **this**.mActivity = (Activity)context;  
}

### 1.2.7 Fragment监听返回键事件

项目要求用户注册成功后进入修改个人资料的页面，且不允许返回到上一个页面，资料修改完成后结束当前页面，进入APP主页。

**代码实现**

xxx\_Fragment

@Override

public void onResume() {

super.onResume();

getFocus();

}

//主界面获取焦点

private void getFocus() {

getView().setFocusableInTouchMode(true);

getView().requestFocus();

getView().setOnKeyListener(new View.OnKeyListener() {

@Override

public boolean onKey(View v, int keyCode, KeyEvent event) {

if (event.getAction() == KeyEvent.ACTION\_UP && keyCode == KeyEvent.KEYCODE\_BACK) {

// 监听到返回按钮点击事件

......

return true;

}

return false;

}

});

}

但是在使用时，由于Fragment页面里可能有其他能获取焦点的View（例如EditText），会导致监听失效，点击返回键会返回到上个页面。

更完善的解决方案：

除了上面的代码，我们需要对可以获取焦点的View的setOnKeyListener进行处理，这里以一个EditText为例：

//private EditText nickname;

nickname.setOnKeyListener(new View.OnKeyListener() {

@Override

public boolean onKey(View v, int keyCode, KeyEvent event) {

if (keyCode == KeyEvent.KEYCODE\_BACK

&& event.getAction() == KeyEvent.ACTION\_UP) {

//关闭软键盘

InputMethodManager imm = (InputMethodManager) getActivity().getSystemService(Context.INPUT\_METHOD\_SERVICE);

imm.hideSoftInputFromWindow(nickname.getWindowToken(), 0);

//使得根View重新获取焦点，以监听返回键

getFocus();

}

return false;

}

});

### 1.2.8 一些建议

1、对Fragment传递数据，建议使用setArguments(Bundle args)，而后在onCreate中使用getArguments()取出，在 “内存重启”前，系统会帮你保存数据，不会造成数据的丢失。和Activity的Intent原理一致。

2、使用newInstance(参数) 创建Fragment对象，优点是调用者只需要关系传递的哪些数据，而无需关心传递数据的Key是什么。

3、如果你需要在Fragment中用到宿主Activity对象，建议在你的基类Fragment定义一个Activity的全局变量，在onAttach中初始化。原因参考第一篇的“getActivity()空指针”部分，在onCreateView() 内出现getActivity()的代码 很可能是危险的。

**protected** Activity **mActivity**;  
@Override  
**public void** onAttach(Activity activity) {  
 **super**.onAttach(activity);  
 **this**.**mActivity** = activity;  
}

### 1.2.9 add(), show(), hide(), replace()的那点事

1、区别

show()，hide()最终是让Fragment的View setVisibility(true还是false)，不会调用生命周期；

replace()的话会销毁视图，即调用onDestoryView、onCreateView等一系列生命周期；

add()和 replace()不要在同一个阶级的FragmentManager里混搭使用。

2、使用场景

如果你有一个很高的概率会再次使用当前的Fragment，建议使用show()，hide()，可以提高性能。

在我使用Fragment过程中，大部分情况下都是用show()，hide()，而不是replace()。

3、onHiddenChanged的回调时机

当使用add()+show()，hide()跳转新的Fragment时，旧的Fragment回调onHiddenChanged()，不会回调onStop()等生命周期方法，而新的Fragment在创建时是不会回调onHiddenChanged()，这点要切记。

4、Fragment重叠问题

使用show()，hide()带来的一个问题就是，如果你不做任何处理，在“内存重启”后，Fragment会重叠；

### 1.2.10 恢复Fragment时（同时防止Fragment重叠），选择getFragments()还是findFragmentByTag()

（1）选择getFragments()

对于一个Activity内的多个Fragment，如果Fragment的关系是“流程”，比如登录->注册/忘记密码->填写信息->跳转到主页Activity。这种情况下，用getFragments()的方式是最合适的，在你的Activity内（更好的方式是在你的所有"流程"基类Activity里），写下如下代码：

**protected void** onCreate(@Nullable Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
  
 **if** (savedInstanceState != **null**) {  
 List<Fragment> fragments = getSupportFragmentManager().getFragments();  
  
 **if** (fragments != **null** && fragments.size() > 0) {  
 **boolean** showFlag = **false**;  
  
 FragmentTransaction ft = getSupportFragmentManager().beginTransaction();  
 **for** (**int** i = fragments.size() - 1; i >= 0; i--) {  
 Fragment fragment = fragments.get(i);  
 **if** (fragment != **null**) {  
 **if** (!showFlag) {  
 ft.show(fragments.get(i));  
 showFlag = **true**;  
 } **else** {  
 ft.hide(fragments.get(i));  
 }  
 }  
 }  
 ft.commit();  
 }  
 }  
}

上面恢复Fragment的方式，不仅提高性能，同时避免了Fragment重叠现象，最重要的事，你根本不用关系Activity容器里都有哪些Fragment。

（2）选择findFragmentByTag()恢复

如果你的Activity的Fragments，不是“流程”关系，而是“同级”关系，比如QQ的主界面，“消息”、“联系人”、“动态”，这3个Fragment属于同级关系，用上面的代码就不合适了，恢复的时候总会恢复最后一个，即“动态Fragment”。

正确的做法是在onSaveInstanceState()内保存当前所在Fragment的tag或者下标，在onCreate()是恢复的时候，隐藏其它2个Fragment。

**protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.activity);  
  
 MsgFragment msgFragment;  
 ContactFragment contactFragment;  
 MeFragment meFragment;  
  
 **if** (savedInstanceState != **null**) { *// “内存重启”时调用* msgFragment = getSupportFragmentManager().findFragmentByTag(msgFragment.getClass().getName);  
 contactFragment = getSupportFragmentManager().findFragmentByTag(contactFragment.getClass().getName);  
 meFragment = getSupportFragmentManager().findFragmentByTag(meFragment.getClass().getName);  
  
 index = saveInstanceState.getInt(KEY\_INDEX);  
 *// 根据下标判断离开前是显示哪个Fragment，  
 // 这里省略判断代码，假设离开前是ConactFragment  
 // 解决重叠问题* getFragmentManager().beginTransaction()  
 .show(contactFragment)  
 .hide(msgFragment)  
 .hide(meFragment)  
 .commit();  
 }**else**{ *// 正常时* msgFragment = MsgFragment.newInstance();  
 contactFragment = ContactFragment.newInstance();  
 meFragment = MeFragment.newInstance();  
  
 getFragmentManager().beginTransaction()  
 .add(R.id.container, msgFragment, msgFragment.getClass().getName())  
 .add(R.id.container, contactFragment, contactFragment.getClass().getName())  
 .add(R.id,container,meFragment,meFragment.getClass().getName())  
 .hide(contactFragment)  
 .hide(meFragment)  
 .commit();  
 }  
}  
  
@Override  
**public void** onSaveInstanceState(Bundle outState) {  
 **super**.onSaveInstanceState(outState);  
 *// 保存当前Fragment的下标* outState.putInt(KEY\_INDEX, index);  
}

当然在“同级”关系中，使用getFragments()恢复也是可以的。

### 1.2.11 使用ViewPager+Fragment的注意事项

1、使用ViewPager＋Fragment时，切换不同ViewPager页面，不会回调任何生命周期方法以及onHiddenChanged()，只有setUserVisibleHint(boolean isVisibleToUser)会被回调，所以如果你想进行一些懒加载，需要在这里处理。

2、在给ViewPager绑定FragmentPagerAdapter时，new FragmentPagerAdapter(fragmentManager)的FragmentManager，一定要保证正确，如果ViewPager是Activity内的控件，则传递getSupportFragmentManager()，如果是Fragment的控件中，则应该传递getChildFragmentManager()。只要记住ViewPager内的Fragments是当前组件的子Fragment这个原则即可。

3、如果使用ViewPager+Fragment，不需要在“内存重启”的情况下，去恢复的Fragments，有FragmentPagerAdapter的存在，不需要你去做恢复工作。

### 1.2.12 Fragment事务，你可能不知道的坑

1、如果你在使用popBackStackImmdiate()方法后，紧接着直接调用类似如下事务的方法，因为他们运行在消息队列的问题，还没来得及出栈就运行事务的方法了，这可能会导致不正常现象。

getSupportFragmentManager().popBackStackImmdiate();  
getSupportFragmentManager().beginTransaction()  
.add(R.id.container, fragment , tag)  
.hide(currentFragment)  
.commit;

正确的做法是使用主线程的Handler，将事务放到Runnable里运行。

getSupportFragmentManager().popBackStackImmdiate();  
**new** Handler().post(**new** Runnable(){  
 @Override  
 **public void** run() {  
 *// 在这里执行Fragment事务* }  
});

2、给Fragment设定Fragment转场动画时，如果你没有一整套解决方案，应避免使用.setTransition(transit)以及.setCustomAnimations(enter, exit, popEnter, popExit)，而只使用.setCustomAnimations(enter, exit)这个方法。

### 1.2.13 是使用单Activity＋多Fragment的架构，还是多模块Activity＋多Fragment的架构？

单Activity＋多Fragment：

一个app仅有一个Activity，界面皆是Frament，Activity作为app容器使用。

优点：性能高，速度最快。参考：新版知乎 、google系app

缺点：逻辑比较复杂，尤其当Fragment之间联动较多或者嵌套较深时，比较复杂。

多模块Activity＋多Fragment：

一个模块用一个Activity，比如

1、登录注册流程：

LoginActivity + 登录Fragment + 注册Fragment + 填写信息Fragment ＋ 忘记密码Fragment

2、或者常见的数据展示流程：

DataActivity + 数据列表Fragment + 数据详情Fragment ＋ ...

优点：速度快，相比较单Activity+多Fragment，更易维护。

我的观点：

权衡利弊，我认为多模块Activity＋多Fragment是最合适的架构，开发起来不是很复杂，app的性能又很高效。

当然。Fragment只是官方提供的灵活组件，请优先遵从你的项目设计！真的特别复杂的界面，或者单个Activity就可以完成一个流程的界面，使用Activity可能是更好的方案。

## view onTouch onClick onLongClick

**onClick、onLongClick与onTouchEvent**

从实现中可以看到onClick()和onLongClick()方法是由ACTION\_DOWN和ACTION\_UP事件捕捉后根据各种情况最终确定是否触发的，也就是说如果我们在一个Activity或者View中同时监听或者覆写了onClick(),onLongClick()和onTouchEvent()方法，并不意味着只会发生其中一种。

下面是一个onClick被触发的基本时序的Log：

04-05 05:57:47.123: DEBUG/TSActivity(209): onTouch ACTION\_DOWN

04-05 05:57:47.263: DEBUG/TSActivity(209): onTouch ACTION\_UP

04-05 05:57:47.323: DEBUG/TSActivity(209): onClick

可以看出是按ACTION\_DOWN -> ACTION\_UP -> onClick的次序发生的。

下面是一个onLongClick被触发的基本时序的Log：

04-05 06:00:04.133: DEBUG/TSActivity(248): onTouch ACTION\_DOWN

04-05 06:00:04.642: DEBUG/TSActivity(248): onLongClick

04-05 06:00:05.083: DEBUG/TSActivity(248): onTouch ACTION\_UP

可以看到，在保持按下的状态一定时间后会触发onLongClick,之后抬起手才会发生ACTION\_UP。

**onClick和onLongClick能同时发生吗？**

要弄清楚这个问题只要理解Android对事件处理的所谓消费(consume)概念即可，一个用户的操作会被传递到不同的View控件和同一个控件的不同监听方法处理，任何一个接收并处理了该次事件的方法如果在处理完后返回了true，那么该次event就算被完全处理了，其他的View或者监听方法就不会再有机会处理该event了。

onLongClick的发生是由单独的线程完成的，并且在ACTION\_UP之前，而onClick的发生是在ACTION\_UP后，因此同一次用户touch操作就有可能既发生onLongClick又发生onClick。这样是不是不可思议？所以及时向系统表示“我已经完全处理（消费）了用户的此次操作”，是很重要的事情。例如，我们如果在onLongClick()方法的最后**return true**，那么onClick事件就没有机会被触发了。

下面的Log是在onLongClick()方法return false的情况下，一次触碰操作的基本时序：

04-05 06:00:53.023: DEBUG/TSActivity(277): onTouch ACTION\_DOWN

04-05 06:00:53.533: DEBUG/TSActivity(277): onLongClick

04-05 06:00:55.603: DEBUG/TSActivity(277): onTouch ACTION\_UP

04-05 06:00:55.663: DEBUG/TSActivity(277): onClick

可以看到，在ACTION\_UP后仍然触发了onClick()方法。

## 1.4 Touch 事件的分发和消费机制

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、View、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 | View | Activity |
| public boolean dispatchTouchEvent(MotionEvent ev) | 事件分发 | Yes | Yes | Yes |
| public boolean onInterceptTouchEvent(MotionEvent ev) | 事件拦截 | Yes | Yes | No |
| public boolean onTouchEvent(MotionEvent ev) | 事件响应 | Yes | Yes | Yes |

**dispatchTouchEvent**

如果dispatchTouchEvent 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dispatchTouchEvent return false，事件分发分为三种情况：

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。

3 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。

**onInterceptTouchEvent**

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。

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。

从这张表中我们可以看到 ViewGroup 和 View 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onInterceptTouchEvent(MotionEvent ev)，只有dispatchTouchEvent(MotionEvent ev) 和 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。

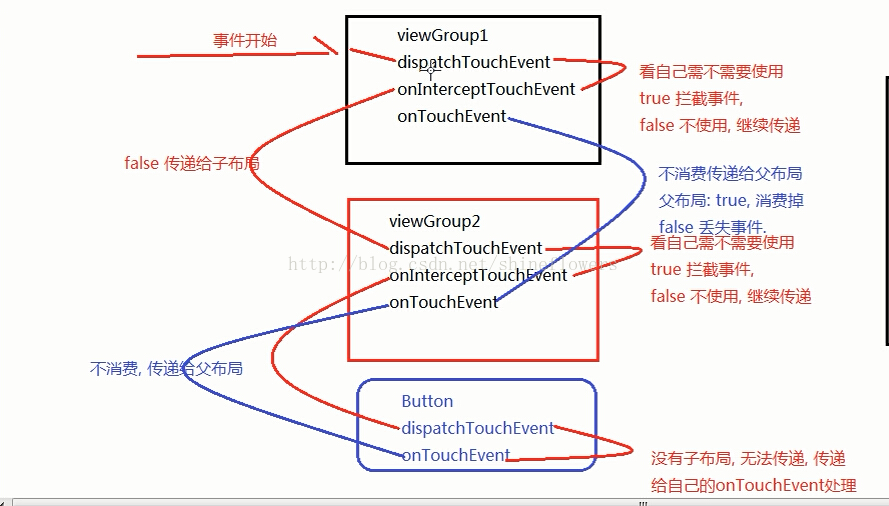
事件拦截：public boolean onInterceptTouchEvent(MotionEvent ev)

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。

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：

通过下面的图可以清楚的看到事件整个传递的过程（布局为viewGroup1在最外层，里面嵌套一个viewGroup2，viewGroup2里面添加了一个Button）。从下图也可以看出，事件会最先被最外层的布局（viewGroup1）给响应，然后依次向内层（viewGroup2）传递，当传递到最内层的View （Button），如果View已经是一个最小单元时，那么就会交给View的onTouchEvent处理事件，如果还不消费该事件，那么事件就会向上回传，直到没有人处理这次事件，事件就会丢失。



## 1.5 WebView

**webview 与JS交互**

webview -> js

mWebView.loadUrl("javascript:setUserName('zhengxiaoyong');");

js –> webview 两种方式：

第一种通过addjavascriptInterface

java:

mWebView.addJavascriptInterface(new JavaScriptInterface(), "ncp");

final class JavaScriptInterface {

public int callOnJs() {

return 1000;

}

public void callOnJs2(String mode) {

//TODO

}

}

JS调用：

window.ncp.callOnJs2("click");

缺陷：4.2版本前存在js注入bug(解决方案是在4.2之后加@JavascriptInterface，在4.2之前通过调用prompt方法实现)

第二种方法通过prompt方法实现

通过js的window.prompt(msg,defaultText) = android->WebViewChromeClient->onJsPrompt(WebView view,

String url, String message, String defaultValue, JsPromptResult result)

其中window.prompt -> msg = onJsPrompt->message

所以在js调用java方法步骤如下：

1 在js中配置好需要的参数信息msg(此消息格式可以自定义)。

2 在js中通过window.prompt(msg,"")方法将信息传到java中的WebViewChromeClient->onJsPrompt方法中。

3 在onJsPrompt中将js传来的信息进行解析处理，传到供js调用的native方法中，之后做相关native方法处理。

4 获得处理结果 拼接callbackJs参数 jscallback()->webView.loadUrl(callbackJs);

**webview的其他准备**

1. 自定义webviewclient实现如下方法：

**public void** onPageFinished(WebView view, String url) {  
 **super**.onPageFinished(view, url);  
 *//4.4以前 先让WebView加载网页的其它静态资源：js、css、文本等等，之后加载图片* **if** (!view.getSettings().getLoadsImagesAutomatically()) {  
 view.getSettings().setLoadsImagesAutomatically(**true**);  
 }  
}  
  
*//无法加载不信任网页SSL错误的处理*@Override  
**public void** onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {  
 *//继续加载* handler.proceed();  
}  
  
@Override  
**public void** onReceivedError(WebView view, WebResourceRequest request, WebResourceError error) {  
 **super**.onReceivedError(view, request, error);  
 *//view.loadUrl("xxxx/404.html");*}

2．自定义webview实现白名单：

*//建立白名单规则*@Override  
**public void** postUrl(String url, **byte**[] postData) {  
 **if** (JsBridgeUrlCheckUtil.*isTrustUrl*(url)) {  
 **super**.postUrl(url, postData);  
 }**else** {  
 System.***out***.println(**"url not on the list"**);  
 **super**.postUrl(url,**null**);  
 }  
}  
  
@Override  
**public void** loadUrl(String url) {  
 **if** (JsBridgeUrlCheckUtil.*isTrustUrl*(url)) {  
 **super**.loadUrl(url);  
 }**else** {  
 System.***out***.println(**"url not on the list"**);  
 }  
}

WebView的混淆问题：

#webview与js交互 class <包名>.类名$\* 这里的类名是你写js交互方法的类 内部类

#-keepclassmembers class 包名.类名$内部类名 {# public \*;#}

#webview与js交互 class <包名>.类名$\* 这里的类名是你写js交互方法的类-keepclassmembers class ccom.pingan.paces.ccms.html.TCJavaScriptInterface { public \*;}

#保护注解

-keepattributes \*Annotation\*

-keepattributes \*JavascriptInterface\*

#当对WebViewClient进行复杂处理时调用

-keepclassmembers class \* extends android.webkit.WebViewClient {\*;}

-keepclassmembers class \* extends android.webkit.WebChromeClient {\*;}

-keep class android.webkit.\*\*{\*;}

-keep class android.net.http.SslError{\*;}

**原生和H5的比较：**

原生页面

优势：

（1）运行速度比较快

（2）能使用设备的底层功能，如摄像头、方向传感器、重力传感器、拨号、GPS、语音、短信、蓝牙等

（3）在界面设计、功能模块、操作逻辑等层面相较web更易做到App的便捷性和舒适性，功能更加强大

（4）节省流量

劣势：

（1）不同的操作系统（如Android和iOS）需要独立的进行开发，使用其各自的开发包、开发工具和控件

（2）每次有更新，都需要重新打包一次发布到应用平台上，且每次要向各个应用商店进行提交审核。之后用户需要手动进行点击更新安装（安装成本较高）

（3）开发成本比较高，尤其需要适配各种机型时（如Android应用，需要适配各种Android手机）

H5页面

优势：

（1）由于是运行在浏览器上，所以只需要开发一次便可以在不同的操作系统上显示

（2）迭代版本时，不需要打包便可以发布（实时更新、快速迭代），与云端实现实时数据交互

（3）开发成本相对较低，对浏览器的适配较简单，且发布门槛相对较低

劣势：

（1）每次打开页面，都得重新加载，获取数据...

（2）过于依赖网络，速度无法保证。特别在弱网环境下，不仅耗费流量而且加载缓慢，就算是WiFi情况下也不容乐观

（3）只能使用有限的设备底层功能（无法使用摄像头、方向传感器、重力传感器、拨号、GPS、语音、短信、蓝牙等功能）

（4）仍处于发展阶段，部分功能无法在基于现有技术的浏览器基础上实现，且无法全面的显示最完美的用户体验，只能用现有技术去弥去找最佳解决方案

**区分混合APP哪部分是H5实现的：**

设置-开发者选项-显示布局边界

H5中使用了webview控件，其作为一个控件，只有一个边界框，所以通过这一点，就比较容易区分出一个界面是webview实现的还是原生布局控件实现的，当然也不排除用一堆webview来拼成一个界面的实现方法。

1. 网络相关
   1. **Thread+handler**

### 2.1.1Handler的定义:

主要接受子线程发送的数据, 并用此数据配合主线程更新UI.

解释: 当应用程序启动时，Android首先会开启一个主线程 (也就是UI线程) , 主线程为管理界面中的UI控件，进行事件分发, 比如说, 你要是点击一个 Button, Android会分发事件到Button上，来响应你的操作。 如果此时需要一个耗时的操作，例如: 联网读取数据，或者读取本地较大的一个文件的时候，你不能把这些操作放在主线程中，如果你放在主线程中的话，界面会出现假死现象, 如果5秒钟还没有完成的话，会收到Android系统的一个错误提示 "强制关闭". 这个时候我们需要把这些耗时的操作，放在一个子线程中,因为子线程涉及到UI更新，Android主线程是线程不安全的，也就是说，更新UI只能在主线程中更新，子线程中操作是危险的. 这个时候，Handler就出现了来解决这个复杂的问题，由于Handler运行在主线程中(UI线程中)，它与子线程可以通过Message对象来传递数据，这个时候，Handler就承担着接受子线程传过来的(子线程用sedMessage()方法传弟)Message对象，(里面包含数据) , 把这些消息放入主线程队列中，配合主线程进行更新UI。

### 2.1.2使用过程

1定义Handler对象并初始化，重写handleMessage（）函数

2定义Thread线程对象，通常写成一个类形式（如class ThreadTest implements Runnable），在run()方法中操作数据，并把数据handler.sendMessage（）方法传输 到handler对象中，并开启线程。（注意：该步骤不一定用Thread实现，也可以利用TimeTask实现，具体的操作同样放在run()方法中）

3在handleMessage（）函数中根据不同的数据形式实现不同的方法。

### 2.1.3 Handler内存泄露问题

Handler 的使用造成的内存泄漏问题应该说是最为常见了，很多时候我们为了避免 ANR 而不在主线程进行耗时操作，在处理网络任务或者封装一些请求回调等api都借助Handler来处理，但 Handler 不是万能的，对于 Handler 的使用代码编写一不规范即有可能造成内存泄漏。另外，我们知道 Handler、Message 和 MessageQueue 都是相互关联在一起的，万一 Handler 发送的 Message 尚未被处理，则该 Message 及发送它的 Handler 对象将被线程 MessageQueue 一直持有。

修复方法：在 Activity 中避免使用非静态内部类，比如上面我们将 Handler 声明为静态的，则其存活期跟 Activity 的生命周期就无关了。同时通过弱引用的方式引入 Activity，避免直接将 Activity 作为 context 传进去，见下面代码：

public class SampleActivity extends Activity {

/\*\*

\* Instances of static inner classes do not hold an implicit

\* reference to their outer class.

\*/

private static class MyHandler extends Handler {

private final WeakReference<SampleActivity> mActivity;

public MyHandler(SampleActivity activity) {

mActivity = new WeakReference<SampleActivity>(activity);

}

@Override

public void handleMessage(Message msg) {

SampleActivity activity = mActivity.get();

if (activity != null) {

// ...

}

to do something

}

}

private final MyHandler mHandler = new MyHandler(this);

/\*\*

\* Instances of anonymous classes do not hold an implicit

\* reference to their outer class when they are "static".

\*/

private static final Runnable sRunnable = new Runnable() {

@Override

public void run() { /\* ... \*/ }

};

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

// Post a message and delay its execution for 10 minutes.

mHandler.postDelayed(sRunnable, 1000 \* 60 \* 10);

// Go back to the previous Activity.

finish();

}

}

综述，即推荐使用静态内部类 + WeakReference 这种方式。每次使用前注意判空。

前面提到了 WeakReference，所以这里就简单的说一下 Java 对象的几种引用类型。

Java对引用的分类有 Strong reference, SoftReference, WeakReference, PhatomReference 四种。



1. public void onDestroy() {
2. //  If null, all callbacks and messages will be removed.
3. mHandler.removeCallbacksAndMessages(null);
4. }

### 2.1.4 handler+looper+messagequeue

1） 接下来，我们开始这部分的内容，首先了解一下各自的职责及相互之间的关系。

职责

Message：消息，其中包含了消息ID，消息处理对象以及处理的数据等，由MessageQueue统一列队，终由Handler处理。

Handler：处理者，负责Message的发送及处理。使用Handler时，需要实现handleMessage(Message msg)方法来对特定的Message进行处理，例如更新UI等。

MessageQueue：消息队列，用来存放Handler发送过来的消息，并按照FIFO规则执行。当然，存放Message并非实际意义的保存，而是将Message以链表的方式串联起来的，等待Looper的抽取。

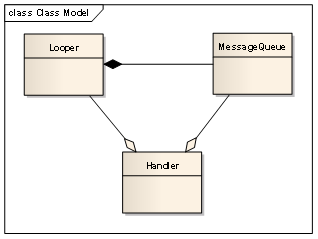
Looper：消息泵，不断地从MessageQueue中抽取Message执行。因此，一个MessageQueue需要一个Looper。

Thread：线程，负责调度整个消息循环，即消息循环的执行场所。

关系

Handler，Looper和MessageQueue就是简单的三角关系。Looper和MessageQueue一一对应，创建一个Looper的同时，会创建一个MessageQueue。而Handler与它们的关系，只是简单的聚集关系，即Handler里会引用当前线程里的特定Looper和MessageQueue。

这样说来，多个Handler都可以共享同一Looper和MessageQueue了。当然，这些Handler也就运行在同一个线程里。



2） 接下来，我们简单地看一下消息的循环过程：

**生成**

       Message msg = mHandler.obtainMessage();

       msg.what = what;

       msg.sendToTarget();

**发送**

       MessageQueue queue = mQueue;

        if (queue != null) {

            msg.target = this;

            sent = queue.enqueueMessage(msg, uptimeMillis);

        }

在Handler.java的sendMessageAtTime(Message msg, long uptimeMillis)方法中，我们看到，它找到它所引用的MessageQueue，然后将Message的target设定成自己（目的是为了在处理消息环节，Message能找到正确的Handler），再将这个Message纳入到消息队列中。

**抽取**

        Looper me = myLooper();

        MessageQueue queue = me.mQueue;

        while (true) {

            Message msg = queue.next(); // might block

            if (msg != null) {

                if (msg.target == null) {

                    // No target is a magic identifier for the quit message.

                    return;

                }

                msg.target.dispatchMessage(msg);

                msg.recycle();

            }

        }

在Looper.java的loop()函数里，我们看到，这里有一个死循环，不断地从MessageQueue中获取下一个（next方法）Message，然后通过Message中携带的target信息，交由正确的Handler处理（dispatchMessage方法）。

**处理**

        if (msg.callback != null) {

            handleCallback(msg);

        } else {

            if (mCallback != null) {

                if (mCallback.handleMessage(msg)) {

                    return;

                }

            }

            handleMessage(msg);

        }

在Handler.java的dispatchMessage(Message msg)方法里，其中的一个分支就是调用handleMessage方法来处理这条Message，而这也正是我们在**职责**处描述使用Handler时需要实现handleMessage(Message msg)的原因。

至于dispatchMessage方法中的另外一个分支，我将会在后面的内容中说明。

至此，我们看到，一个Message经由Handler的发送，MessageQueue的入队，Looper的抽取，又再一次地回到Handler的怀抱。而绕的这一圈，也正好帮助我们将同步操作变成了异步操作。

3）剩下的部分，我们将讨论一下Handler所处的线程及更新UI的方式。

在主线程（UI线程）里，如果创建Handler时不传入Looper对象，那么将直接使用主线程（UI线程）的Looper对象（系统已经帮我们创建了）；在其它线程里，如果创建Handler时不传入Looper对象，那么，这个Handler将不能接收处理消息。在这种情况下，通用的作法是：

                class LooperThread extends Thread {

                               public Handler mHandler;

                               public void run() {

                                               Looper.prepare();

                                               mHandler = new Handler() {

                                                               public void handleMessage(Message msg) {

                                                                              // process incoming messages here

                                                               }

                                               };

                                               Looper.loop();

                               }

                }

在创建Handler之前，为该线程准备好一个Looper（Looper.prepare），然后让这个Looper跑起来（Looper.loop），抽取Message，这样，Handler才能正常工作。

因此，Handler处理消息总是在创建Handler的线程里运行。而我们的消息处理中，不乏更新UI的操作，不正确的线程直接更新UI将引发异常。因此，需要时刻关心Handler在哪个线程里创建的。

如何更新UI才能不出异常呢？SDK告诉我们，有以下4种方式可以从其它线程访问UI线程：

·      [Activity.runOnUiThread(Runnable)](file:///C:/DEV/eclipse/SDK/docs/reference/android/app/Activity.html#runOnUiThread(java.lang.Runnable))

·      [View.post(Runnable)](file:///C:/DEV/eclipse/SDK/docs/reference/android/view/View.html#post(java.lang.Runnable))

·      [View.postDelayed(Runnable, long)](file:///C:/DEV/eclipse/SDK/docs/reference/android/view/View.html#postDelayed(java.lang.Runnable, long))

·      [Handler](file:///C:/DEV/eclipse/SDK/docs/reference/android/os/Handler.html)

其中，重点说一下的是View.post(Runnable)方法。在post(Runnable action)方法里，View获得当前线程（即UI线程）的Handler，然后将action对象post到Handler里。在Handler里，它将传递过来的action对象包装成一个Message（Message的callback为action），然后将其投入UI线程的消息循环中。在Handler再次处理该Message时，有一条分支（未解释的那条）就是为它所设，直接调用runnable的run方法。而此时，已经路由到UI线程里，因此，我们可以毫无顾虑的来更新UI。

4） 几点小结

·      Handler的处理过程运行在创建Handler的线程里

·      一个Looper对应一个MessageQueue

·      一个线程对应一个Looper

·      一个Looper可以对应多个Handler

·      不确定当前线程时，更新UI时尽量调用post方法

* 1. **Asynctask**

### 2.2.1 基本信息

先来看看AsyncTask的定义：

**public** **abstract** **class** AsyncTask<Params, Progress, Result>

三种泛型类型分别代表“启动任务执行的输入参数”、“后台任务执行的进度”、“后台计算结果的类型”。在特定场合下，并不是所有类型都被使用，如果没有被使用，可以用java.lang.Void类型代替。

一个异步任务的执行一般包括以下几个步骤：

1.execute(Params... params)，执行一个异步任务，需要我们在代码中调用此方法，触发异步任务的执行。

2.onPreExecute()，在execute(Params... params)被调用后立即执行，一般用来在执行后台任务前对UI做一些标记。

3.doInBackground(Params... params)，在onPreExecute()完成后立即执行，用于执行较为费时的操作，此方法将接收输入参数和返回计算结果。在执行过程中可以调用publishProgress(Progress... values)来更新进度信息。

4.onProgressUpdate(Progress... values)，在调用publishProgress(Progress... values)时，此方法被执行，直接将进度信息更新到UI组件上。

5.onPostExecute(Result result)，当后台操作结束时，此方法将会被调用，计算结果将做为参数传递到此方法中，直接将结果显示到UI组件上。

在使用的时候，有几点需要格外注意：

1.异步任务的实例必须在UI线程中创建。

2.execute(Params... params)方法必须在UI线程中调用。

3.不要手动调用onPreExecute()，doInBackground(Params... params)，onProgressUpdate(Progress... values)，onPostExecute(Result result)这几个方法。

4.不能在doInBackground(Params... params)中更改UI组件的信息。

5.一个任务实例只能执行一次，如果执行第二次将会抛出异常。

### 2.2.2 AsyncTast对象多次使用以及线程池大小

如果我们创建了一个AsyncTask的对象，多次执行execute()会报错，一个对象只能运行一次worker线程。

因为每个AsyncTask对象在执行前都会判断一下当前对象的状态值，只有在特定值的时候在可以执行，而这个对象一旦执行当前状态就变化了。

AsyncTask线程池的队列大小是128。

* 1. **线程池**
     1. **ExecutorService**

通过上述分析，我们知道了通过new Thread().start()方式创建线程去处理任务的弊端，而为了解决这些问题，Java为我们提供了ExecutorService线程池来优化和管理线程的使用

**使用线程池管理线程的优点**

1、线程的创建和销毁由线程池维护，一个线程在完成任务后并不会立即销毁，而是由后续的任务复用这个线程，从而减少线程的创建和销毁，节约系统的开销

2、线程池旨在线程的复用，这就可以节约我们用以往的方式创建线程和销毁所消耗的时间，减少线程频繁调度的开销，从而节约系统资源，提高系统吞吐量

3、在执行大量异步任务时提高了性能

4、Java内置的一套ExecutorService线程池相关的api，可以更方便的控制线程的最大并发数、线程的定时任务、单线程的顺序执行等

**ExecutorService简介**

通常来说我们说到线程池第一时间想到的就是它：ExecutorService，它是一个接口，其实如果要从真正意义上来说，它可以叫做线程池的服务，因为它提供了众多接口api来控制线程池中的线程，而真正意义上的线程池就是：ThreadPoolExecutor，它实现了ExecutorService接口，并封装了一系列的api使得它具有线程池的特性，其中包括工作队列、核心线程数、最大线程数等。

**线程池：ThreadPoolExecutor**

既然线程池就是ThreadPoolExecutor，所以我们要创建一个线程池只需要new ThreadPoolExecutor(…);就可以创建一个线程池，而如果这样创建线程池的话，我们需要配置一堆东西，非常麻烦，我们可以看一下它的构造方法就知道了：

public ThreadPoolExecutor(int corePoolSize,

                              int maximumPoolSize,

                              long keepAliveTime,

                              TimeUnit unit,

                              BlockingQueue workQueue,

                              ThreadFactory threadFactory,

                              RejectedExecutionHandler handler) {...}

* + 1. **线程池种类**

所以，官方也不推荐使用这种方法来创建线程池，而是推荐使用Executors的工厂方法来创建线程池，Executors类是官方提供的一个工厂类，它里面封装好了众多功能不一样的线程池，从而使得我们创建线程池非常的简便，主要提供了如下五种功能不一样的线程池：

1固定线程池newFixedThreadPool() ：

作用：该方法返回一个固定线程数量的线程池，该线程池中的线程数量始终不变，即不会再创建新的线程，也不会销毁已经创建好的线程，自始自终都是那几个固定的线程在工作，所以该线程池可以控制线程的最大并发数。

栗子：假如有一个新任务提交时，线程池中如果有空闲的线程则立即使用空闲线程来处理任务，如果没有，则会把这个新任务存在一个任务队列中，一旦有线程空闲了，则按FIFO方式处理任务队列中的任务。

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(5);

2 newCachedThreadPool() ：

作用：该方法返回一个可以根据实际情况调整线程池中线程的数量的线程池。即该线程池中的线程数量不确定，是根据实际情况动态调整的。

栗子：假如该线程池中的所有线程都正在工作，而此时有新任务提交，那么将会创建新的线程去处理该任务，而此时假如之前有一些线程完成了任务，现在又有新任务提交，那么将不会创建新线程去处理，而是复用空闲的线程去处理新任务。那么此时有人有疑问了，那这样来说该线程池的线程岂不是会越集越多？其实并不会，因为线程池中的线程都有一个“保持活动时间”的参数，通过配置它，如果线程池中的空闲线程的空闲时间超过该“保存活动时间”则立刻停止该线程，而该线程池默认的“保持活动时间”为60s。

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

3 newSingleThreadExecutor() ：

作用：该方法返回一个只有一个线程的线程池，即每次只能执行一个线程任务，多余的任务会保存到一个任务队列中，等待这一个线程空闲，当这个线程空闲了再按FIFO方式顺序执行任务队列中的任务。

ExecutorService singleThreadPool = Executors.newSingleThreadExecutor();

4 newScheduledThreadPool() ：

作用：该方法返回一个可以控制线程池内线程定时或周期性执行某任务的线程池。

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);

5 newSingleThreadScheduledExecutor() ：

作用：该方法返回一个可以控制线程池内线程定时或周期性执行某任务的线程池。只不过和上面的区别是该线程池大小为1，而上面的可以指定线程池的大小。

ScheduledExecutorService singleThreadScheduledPool = Executors.newSingleThreadScheduledExecutor();

线程池ThreadPoolExecutor的使用

使用线程池，其中涉及到一个极其重要的方法，即：

execute(Runnable command)

该方法意为执行给定的任务，该任务处理可能在新的线程、已入池的线程或者正调用的线程，这由ThreadPoolExecutor的实现决定。

1. 四大组件
   1. **Activity**

**3.1.1生命周期**

**Activity生命周期**

**在系统中的Activity被一个Activity栈所管理。当一个新的Activity启动时，将被放置到栈顶，成为运行中的Activity，前一个Activity保留在栈中，不再放到前台，直到新的Activity退出为止。**

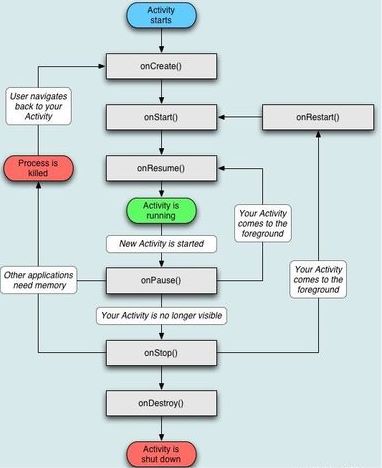
**Activity有四种本质区别的状态：**

**1.在屏幕的前台（Activity栈顶），叫做活动状态或者运行状态（active or running）**

**2.如果一个Activity失去焦点，但是依然可见（一个新的非全屏的Activity 或者一个透明的Activity 被放置在栈顶），叫做暂停状态（Paused）。一个暂停状态的Activity依然保持活力（保持所有的状态，成员信息，和窗口管理器保持连接），但是在系统内存极端低下的时候将被杀掉。**

**3.如果一个Activity被另外的Activity完全覆盖掉，叫做停止状态（Stopped）。它依然保持所有状态和成员信息，但是它不再可见，所以它的窗口被隐藏，当系统内存需要被用在其他地方的时候，Stopped的Activity将被杀掉。**

**4.如果一个Activity是Paused或者Stopped状态，系统可以将该Activity从内存中删除，Android系统采用两种方式进行删除，要么要求该Activity结束，要么直接杀掉它的进程。当该Activity再次显示给用户时，它必须重新开始和重置前面的状态。**

****

**在上图中，Activity有三个关键的循环：**

1. **整个的生命周期，从onCreate(Bundle)开始到onDestroy()结束。Activity在onCreate()设置所有的“全局”状态，在onDestory()释放所有的资源。例如：某个Activity有一个在后台运行的线程，用于从网络下载数据，则该Activity可以在onCreate()中创建线程,在onDestory()中停止线程。**
2. **可见的生命周期，从onStart()开始到onStop()结束。在这段时间，可以看到Activity在屏幕上，尽管有可能不在前台，不能和用户交互。在这两个接口之间，需要保持显示给用户的UI数据和资源等，例如：可以在onStart中注册一个IntentReceiver来监听数据变化导致UI的变动，当不再需要显示时候，可以在onStop()中注销它。onStart()，onStop()都可以被多次调用，因为Activity随时可以在可见和隐藏之间转换。**
3. **前台的生命周期，从onResume()开始到onPause()结束。在这段时间里，该Activity处于所有 Activity的最前面，和用户进行交互。Activity可以经常性地在resumed和paused状态之间切换，例如：当设备准备休眠时，当一个 Activity处理结果被分发时，当一个新的Intent被分发时。所以在这些接口方法中的代码应该属于非常轻量级的。**

### 3.1.2 Activity启动模式

启动模式简单地说就是Activity启动时的策略，在AndroidManifest.xml中的标签的android:launchMode属性设置；

　　启动模式有4种，分别为standard、singleTop、singleTask、singleInstance；

讲解启动模式之前，有必要先讲解一下“任务栈”的概念;

　　任务栈

每个应用都有一个任务栈，是用来存放Activity的，功能类似于函数调用的栈，先后顺序代表了Activity的出现顺序；比如Activity1-->Activity2-->Activity3,则任务栈为：

(1)standard：每次激活Activity时(startActivity)，都创建Activity实例，并放入任务栈；

(2)singleTop：如果某个Activity自己激活自己，即任务栈栈顶就是该Activity，则不需要创建，其余情况都要创建Activity实例；

(3)singleTask：如果要激活的那个Activity在任务栈中存在该实例，则不需要创建，只需要把此Activity放入栈顶，并把该Activity以上的Activity实例都pop；

(4)singleInstance：如果应用1的任务栈中创建了MainActivity实例，如果应用2也要激活MainActivity，则不需要创建，两应用共享该Activity实例；

### 3.1.3 Activity跳转动画

startActivity(new Intent(this, OtherActivity.class));

overridePendingTransition(R.anim.move\_in\_right, R.anim.move\_out\_right);

### 3.1.4 Activity访问权限

一定要为Activity中的属性android:exported设置值为true，以表示可以被其他应用程序打开。或者，也可以进行如下配置：

<activity android:name="com.example.testandroid.BActivity" >

<intent-filter>

<action android:name="" />

</intent-filter>

</activity>

1.既然是自定义权限，那么首先得申明此权限：

在B中的AndroidManifest.xml中，一般是紧跟uses-sdk标签后，通过permission标签进行申明。

<permission android:description="string resource"

android:icon="drawable resource"

android:label="string resource"

android:name="string"

android:permissionGroup="string"

android:protectionLevel=["normal" | "dangerous" | "signature" | "signatureOrSystem"] />

2.当B中BActivity进行权限限定时，需要对BActivity进行如下声明：

<activity

android:name="com.example.testandroid.BActivity"

android:exported="true"

android:label="B"

android:permission="corn.permission.CORN\_OWN" >

</activity>

3.此时外部应用A中的Activity想直接打开B中BActivity，则需要添加上相应权限：

<uses-permission android:name="corn.permission.CORN\_OWN" >

* 1. **Service**
     1. **service生命周期**

****

**3.2.2service启动方式**

**通过startService()启动的服务处于“启动的”状态，一旦启动，service就在后台运行，即使启动它的应用组件已经被销毁了。通常started状态的service执行单任务并且不返回任何结果给启动者。比如当下载或上传一个文件，当这项操作完成时，service应该停止它本身。**

**还有一种“绑定”状态的service，通过调用bindService()来启动，一个绑定的service提供一个允许组件与service交互的接口，可以发送请求、获取返回结果，还可以通过夸进程通信来交互（IPC）。绑定的service只有当应用组件绑定后才能运行，多个组件可以绑定一个service，当调用unbind()方法时，这个service就会被销毁了。**

**另外，在官方的说明文档中还有一个警告：**

**意思是service与activity一样都存在与当前进程的主线程中，所以，一些阻塞UI的操作，比如耗时操作不能放在service里进行，比如另外开启一个线程来处理诸如网络请求的耗时操作。如果在service里进行一些耗CPU和耗时操作，可能会引发ANR警告，这时应用会弹出是强制关闭还是等待的对话框。所以，对service的理解就是和activity平级的，只不过是看不见的，在后台运行的一个组件，这也是为什么和activity同被说为Android的基本组件。**

**两种启动service的方式以及他们的生命周期，bind service的不同之处在于当绑定的组件销毁后，对应的service也就被kill了。service的声明周期相比与activity的简单了许多，只要好好理解两种启动service方式的异同就行。**

**多次绑定服务时，只有所有的服务都解绑后，系统才会销毁当前服务。**

* + 1. **IntentService**

**IntentService:异步处理服务，新开一个线程：handlerThread在线程中发消息，然后接受处理完成后，会清理线程，并且关掉服务。**

**IntentService有以下特点：**

**（1）它创建了一个独立的工作线程来处理所有的通过onStartCommand()传递给服务的intents。**

**（2）创建了一个工作队列，来逐个发送intent给onHandleIntent()。**

**（3）不需要主动调用stopSelft()来结束服务。因为，在所有的intent被处理完后，系统会自动关闭服务。**

**（4）默认实现的onBind()返回null**

**（5）默认实现的onStartCommand()的目的是将intent插入到工作队列中**

**继承IntentService的类至少要实现两个函数：构造函数和onHandleIntent()函数。要覆盖 IntentService的其它函数时，注意要通过super调用父类的对应的函数。**

* 1. **BroadcastProvider**

### 3.3.1广播的两种注册方式

**注册方式有两种：**

**一种是静态注册，就是在AndroidManifest.xml文件中定义，注册的广播接收器必须要继承BroadcastReceiver；**

**另一种是动态注册，是在程序中使用Context.registerReceiver注册，注册的广播接收器相当于一个匿名类。两种方式都需要IntentFIlter。**

**发送广播事件：通过Context.sendBroadcast来发送，由Intent来传递注册时用到的Action。**

**接收广播事件：当发送的广播被接收器监听到后，会调用它的onReceive()方法，并将包含消息的Intent对象传给它。onReceive中代码的执行时间不要超过5s，否则Android会弹出超时dialog。**

### 3.3.2 自定义广播权限

**1. 一些敏感的广播并不想让第三方的应用收到 ；**

**2. 要限制自己的Receiver接收某广播来源，避免被恶意的同样的ACTION的广播所干扰。**

**第一种场景： 谁有权收我的广播？**

**在这种情况下，可以在自己应用发广播时添加参数声明Receiver所需的权限。**

**首先，在Androidmanifest.xml中定义新的权限RECV\_XXX，例如：**

**<permission android:name = "com.android.permission.RECV\_XXX"/>**

**然后，在Sender app发送广播时将此权限作为参数传入，如下：**

**sendBroadcast("com.android.XXX\_ACTION", "com.android.permission.RECV\_XXX");**

**这样做之后就使得只有具有permission权限的Receiver才能接收此广播要接收该广播，在Receiver应用的AndroidManifest.xml中要添加对应的RECV\_XXX权限。**

**<uses-permission android:name="com.android.permission.RECV\_XXX"></uses-permission>**

**第二种场景： 谁有权给我发广播？**

**在这种情况下，需要在Receiver app的<receiver> tag中声明一下Sender app应该具有的权限。**

**首先同上，在AndroidManifest.xml中定义新的权限SEND\_XXX，例如：**

**<permission android:name="com.android.SEND\_XXX"/>**

**然后，在Receiver app的Androidmanifest.xml中的<receiver>tag里添加权限SEND\_XXX的声明，如下：**

**<receiver android:name=".XXXReceiver"**

**android:permission="com.android.permission.SEND\_XXX">**

**<intent-filter>**

**<action android:name="com.android.XXX\_ACTION" />**

**</intent-filter>**

**</receiver>**

**这样一来，该Receiver便只能接收来自具有该send\_permission权限的应用发出的广播。**

**要发送这种广播，需要在Sender app的AndroidManifest.xml中也声明使用该权限即可，如下：**

**<uses-permission android:name="com.android.permission.SEND\_XXX"></uses-permission>**

* 1. **ContentProvider**

**ContentProvider（内容提供者）是Android中的四大组件之一。主要用于对外共享数据，也就是通过ContentProvider把应用中的数据共享给其他应用访问，其他应用可以通过ContentProvider对指定应用中的数据进行操作。ContentProvider分为系统的和自定义的，系统的也就是例如联系人，图片等数据。**

**以下这段是Google Doc中对ContentProvider的大致概述。**

**内容提供者将一些特定的应用程序数据供给其它应用程序使用。数据可以存储于文件系统、SQLite数据库或其它方式。内容提供者继承于ContentProvider 基类，为其它应用程序取用和存储它管理的数据实现了一套标准方法。然而，应用程序并不直接调用这些方法，而是使用一个 ContentResolver 对象，调用它的方法作为替代。ContentResolver可以与任意内容提供者进行会话，与其合作来对所有相关交互通讯进行管理。**

### 3.4.1.ContentProvider

**Android提供了一些主要数据类型的ContentProvider，比如音频、视频、图片和私人通讯录等。可在android.provider包下面找到一些Android提供的ContentProvider。通过获得这些ContentProvider可以查询它们包含的数据，当然前提是已获得适当的读取权限。**

**主要方法：**

**public boolean onCreate() 在创建ContentProvider时调用**

**public Cursor query(Uri, String[], String, String[], String) 用于查询指定Uri的ContentProvider，返回一个Cursor**

**public Uri insert(Uri, ContentValues) 用于添加数据到指定Uri的ContentProvider中**

**public int update(Uri, ContentValues, String, String[]) 用于更新指定Uri的ContentProvider中的数据**

**public int delete(Uri, String, String[]) 用于从指定Uri的ContentProvider中删除数据**

**public String getType(Uri) 用于返回指定的Uri中的数据的MIME类型**

**\*如果操作的数据属于集合类型，那么MIME类型字符串应该以vnd.android.cursor.dir/开头。**

**例如：要得到所有person记录的Uri为content://contacts/person，那么返回的MIME类型字符串为"vnd.android.cursor.dir/person"。**

**\*如果要操作的数据属于非集合类型数据，那么MIME类型字符串应该以vnd.android.cursor.item/开头。**

**例如：要得到id为10的person记录的Uri为content://contacts/person/10，那么返回的MIME类型字符串应为"vnd.android.cursor.item/person"。**

### 3.4.2.ContentResolver

**当外部应用需要对ContentProvider中的数据进行添加、删除、修改和查询操作时，可以使用ContentResolver类来完成，要获取ContentResolver对象，可以使用Context提供的getContentResolver()方法。**

**ContentResolver提供的方法和ContentProvider提供的方法对应的有以下几个方法。**

**public Uri insert(Uri uri, ContentValues values) 用于添加数据到指定Uri的ContentProvider中。**

**public int delete(Uri uri, String selection, String[] selectionArgs) 用于从指定Uri的ContentProvider中删除数据。**

**public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) 用于更新指定Uri的ContentProvider中的数据。**

**public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) 用于查询指定Uri的ContentProvider。**

### 3.4.3 Uri

**Uri指定了将要操作的ContentProvider，其实可以把一个Uri看作是一个网址，我们把Uri分为三部分。**

**第一部分是"content://"。可以看作是网址中的"http://"。**

**第二部分是主机名或authority，用于唯一标识这个ContentProvider，外部应用需要根据这个标识来找到它。可以看作是网址中的主机名，比如"blog.csdn.net"。**

**第三部分是路径名，用来表示将要操作的数据。可以看作网址中细分的内容路径。**

1. 数据存储
   1. **Sqlite**
      1. **数据库升级问题**

1.1准备目录

在Android项目工程res下新建raw文件夹，在这个文件夹里面的文件不会被编译压缩。

1.2准备数据库

代码实现

/\*\*

\* 是否完成初始化

\*/

private static boolean isInit = false;

/\*\*

\* 初始化数据库

\* @param context

\*/

synchronized public static void init(Context context)

{

if(isInit)

return;

// 输出路径

String outFileName = DATABASE\_PATH + DATABASE\_NAME;

//检测是否已经创建

File dir = new File(outFileName);

if(dir.exists())

return;

// 检测/创建数据库的文件夹

dir = new File(DATABASE\_PATH);

if (!dir.exists())

dir.mkdir();

InputStream input = null;

OutputStream output = null;

// 从资源中读取数据库流

input = context.getResources().openRawResource(R.raw.db);

try {

output = new FileOutputStream(outFileName);

// 拷贝到输出流

byte[] buffer = new byte[2048];

int length;

while ((length = input.read(buffer)) > 0) {

output.write(buffer, 0, length);

}

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

} finally {

// 关闭输出流

try {

output.flush();

output.close();

} catch (IOException e) {

}

// 关闭输入流

try {

input.close();

} catch (IOException e) {

}

}

isInit = true;

}

### 4.1.2 数据库升级表结构变化

做Android应用，不可避免的会与SQLite打交道。随着应用的不断升级，原有的数据库结构可能已经不再适应新的功能，这时候，就需要对SQLite数据库的结构进行升级了。 SQLite提供了ALTER TABLE命令，允许用户重命名或添加新的字段到已有表中，但是不能从表中删除字段。

并且只能在表的末尾添加字段，比如,为 Subscription添加两个字段：

1 ALTER TABLE Subscription ADD COLUMN Activation BLOB;

2 ALTER TABLE Subscription ADD COLUMN Key BLOB;

另外，如果遇到复杂的修改操作，比如在修改的同时，需要进行数据的转移，那么可以采取在一个事务中执行如下语句来实现修改表的需求。

1. 将表名改为临时表

ALTER TABLE Subscription RENAME TO \_\_temp\_\_Subscription;

2. 创建新表

CREATE TABLE Subscription (OrderId VARCHAR(32) PRIMARY KEY ,UserName VARCHAR(32) NOT NULL ,ProductId VARCHAR(16) NOT NULL);

3. 导入数据

INSERT INTO Subscription SELECT OrderId, “”, ProductId FROM \_\_temp\_\_Subscription;

　　或者

INSERT INTO Subscription() SELECT OrderId, “”, ProductId FROM \_\_temp\_\_Subscription;

　　\* 注意 双引号”” 是用来补充原来不存在的数据的

4. 删除临时表

DROP TABLE \_\_temp\_\_Subscription;

　　通过以上四个步骤，就可以完成旧数据库结构向新数据库结构的迁移，并且其中还可以保证数据不会应为升级而流失。

当然，如果遇到减少字段的情况，也可以通过创建临时表的方式来实现。

### 4.1.3 SQLCipher对SQLite数据库进行加密

我们都知道，Android系统内置了SQLite数据库，并且提供了一整套的API用于对数据库进行增删改查操作。数据库存储是我们经常会使用到的一种存储方式，相信大多数朋友对它的使用方法都已经比较熟悉了吧。在Android中，我们既可以使用原生的SQL语句来对数据进行操作，也可以使用Android API提供的CRUD方法来对数据库进行操作，两种方式各有特点，选择使用哪一种就全凭个人喜好了。

不过，使用SQLite来存储数据却存在着一个问题。因为大多数的Android手机都是Root过的，而Root过的手机都可以进入到/data/data/<package\_name>/databases目录下面，在这里就可以查看到数据库中存储的所有数据。如果是一般的数据还好，但是当涉及到一些账号密码，或者聊天内容的时候，我们的程序就会面临严重的安全漏洞隐患。那么今天，就让我们一起研究一下如何借助SQLCipher来解决这个安全性问题。

SQLCipher是一个在SQLite基础之上进行扩展的开源数据库，它主要是在SQLite的基础之上增加了数据加密功能，如果我们在项目中使用它来存储数据的话，就可以大大提高程序的安全性。SQLCipher支持很多种不同的平台，这里我们要学习的自然是Android中SQLCipher的用法了。

导入jar包，引用jar包内的SQLiteOpenHelper

设置打开数据库的密码

mSqLiteDatabase = mSqlCipherOpenHelper.getWritableDatabase(SECRET\_KEY);

* 1. **Sharedpreferences**

1.概述。SharePreferences是用来存储一些简单配置信息的一种机制，使用Map数据结构来存储数据，以键值对的方式存储，采用了XML格式将数据存储到设备中。例如保存登录用户的用户名和密码。只能在同一个包内使用，不能在不同的包之间使用，其实也就是说只能在创建它的应用中使用，其他应用无法使用。

创建的存储文件保存在/data/data/<package name>/shares\_prefs文件夹下。

2.使用。

通过Context.getSharedPreferences方法获取SharedPreferences对象，参数分别为存储的文件名和存储模式。

// 获取SharedPreferences对象

            SharedPreferences sp = getSharedPreferences(DATABASE,

                    Activity.MODE\_PRIVATE);

            // 获取Editor对象

            Editor editor = sp.edit();

            // 　获取界面中的信息

            String key = keyET.getText().toString();

            String value = valueET.getText().toString();

editor.putString(key, value);

String result = sp.getString(key, "");

editor.remove(key);

           editor.commit();

# 性能优化

## 5.1图片相关

### 1 离线图片处理

压缩：https://tinypng.com/

图片正确位置的选择：

| **密度分类** | **密度值范围** | **代表分辨率** | **图标尺寸** | **图片比例** |
| --- | --- | --- | --- | --- |
| mdpi | 120~160dpi | 320x480px | 48x48px | 1 |
| hdpi | 160~240dpi | 480x800px | 72x72px | 1.5 |
| xhdpi | 240~320dpi | 720x1280px | 96x96px | 2 |
| xxhdpi | 320~480dpi | 1080x1920px | 144x144px | 3 |
| xxxhdpi | 480~640dpi | 1440x2560px | 192x192px | 4 |

能使用.9图片，尽量使用.9图片，能使用色值的就不要使用图片

图片缓存实现：

1. 内存缓存，LruCache实现
2. 磁盘缓存实现

磁盘缓存的优势在于能够缓存的图片数量比较多，不足就是磁盘IO的速度比较慢。磁盘缓存我们可以用DiskLruCache来实现，这个类不属于Android SDK，它的源码可以从这里获取：<http://developer.android.com/intl/zh-cn/samples/DisplayingBitmaps/src/com.example.android.displayingbitmaps/util/DiskLruCache.html>。无法访问的同学请看文末给出的本文示例代码的地址，其中包含了DiskLruCache。

## 5.2handler内存泄漏问题

Handler的使用造成的内存泄漏问题应该说最为常见了，平时在处理网络任务或者封装一些请求回调等api都应该会借助Handler来处理，对于Handler的使用代码编写一不规范即有可能造成内存泄漏，如下示例：

**public class** MainActivity **extends** Activity {  
 **private** Handler **mHandler** = **new** Handler() {  
 @Override  
 **public void** handleMessage(Message msg) {  
*//...* }  
 };  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 loadData();  
 }  
 **private void** loadData(){  
*//...request* Message message = Message.obtain();  
 **mHandler**.sendMessage(message);  
 }  
 }

这种创建Handler的方式会造成内存泄漏，由于mHandler是Handler的非静态匿名内部类的实例，所以它持有外部类Activity的引用，我们知道消息队列是在一个Looper线程中不断轮询处理消息，那么当这个Activity退出时消息队列中还有未处理的消息或者正在处理消息，而消息队列中的Message持有mHandler实例的引用，mHandler又持有Activity的引用，所以导致该Activity的内存资源无法及时回收，引发内存泄漏，所以另外一种做法为：

**public class** MainActivity **extends** Activity {  
 **private** MyHandler **mHandler** = **new** MyHandler(**this**);  
 **private** TextView **mTextView** ;  
 **private static class** MyHandler **extends** Handler {  
 **private** WeakReference<Context> **reference**;  
 **public** MyHandler(Context context) {  
 **reference** = **new** WeakReference<>(context);  
 }  
 @Override  
 **public void** handleMessage(Message msg) {  
 MainActivity activity = (MainActivity) **reference**.get();  
 **if**(activity != **null**){  
 activity.**mTextView**.setText(**""**);  
 }  
 }  
 }  
    
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 **mTextView** = (TextView)findViewById(R.id.textview);  
 loadData();  
 }  
    
 **private void** loadData() {  
 *//...request* Message message = Message.obtain();  
 **mHandler**.sendMessage(message);  
 }  
}

创建一个静态Handler内部类，然后对Handler持有的对象使用弱引用，这样在回收时也可以回收Handler持有的对象，这样虽然避免了Activity泄漏，不过Looper线程的消息队列中还是可能会有待处理的消息，所以我们在Activity的Destroy时或者Stop时应该移除消息队列中的消息，更准确的做法如下：

**public class** MainActivity **extends** AppCompatActivity {  
 **private** MyHandler **mHandler** = **new** MyHandler(**this**);  
 **private** TextView **mTextView** ;  
 **private static class** MyHandler **extends** Handler {  
 **private** WeakReference<Context> **reference**;  
 **public** MyHandler(Context context) {  
 **reference** = **new** WeakReference<>(context);  
 }  
 @Override  
 **public void** handleMessage(Message msg) {  
 MainActivity activity = (MainActivity) **reference**.get();  
 **if**(activity != **null**){  
 activity.**mTextView**.setText(**""**);  
 }  
 }  
 }  
  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 **mTextView** = (TextView)findViewById(R.id.textview);  
 loadData();  
 }  
  
 **private void** loadData() {  
*//...request* Message message = Message.obtain();  
 **mHandler**.sendMessage(message);  
 }  
  
 @Override  
 **protected void** onDestroy() {  
 **super**.onDestroy();  
 **mHandler**.removeCallbacksAndMessages(**null**);  
 }  
 }

使用mHandler.removeCallbacksAndMessages(null);是移除消息队列中所有消息和所有的Runnable。当然也可以在Activity的OnDestroy()中使用mHandler.removeCallbacks();或mHandler.removeMessages();来移除指定的Runnable和Message。

## 5.3UI性能优化

1 避免重绘

Overdraw就是过度绘制，是指在一帧的时间内（16.67ms）像素被绘制了多次，理论上一个像素每次只绘制一次是最优的，但是由于重叠的布局导致一些像素会被多次绘制，而每次绘制都会对应到CPU的一组绘图命令和GPU的一些操作，当这个操作耗时超过16.67ms时，就会出现掉帧现象，也就是我们所说的卡顿，所以对重叠不可见元素的重复绘制会产生额外的开销，需要尽量减少Overdraw的发生。

Android提供了测量Overdraw的选项，在开发者选项－调试GPU过度绘制（Show GPU Overdraw），打开选项就可以看到当前页面Overdraw的状态，就可以观察屏幕的绘制状态。该工具会使用三种不同的颜色绘制屏幕，来指示overdraw发生在哪里以及程度如何，其中：

没有颜色： 意味着没有overdraw。像素只画了一次。

蓝色： 意味着overdraw 1倍。像素绘制了两次。大片的蓝色还是可以接受的（若整个窗口是蓝色的，可以摆脱一层）。

绿色： 意味着overdraw 2倍。像素绘制了三次。中等大小的绿色区域是可以接受的但你应该尝试优化、减少它们。

浅红： 意味着overdraw 3倍。像素绘制了四次，小范围可以接受。

暗红： 意味着overdraw 4倍。像素绘制了五次或者更多。这是错误的，要修复它们。



**第一招：合理选择控件容器**

既然overdraw是因为重复绘制了同一片区域的像素点，那我们首先想到的是解决布局问题。Android提供的Layout控件主要包括LinearLayout、TableLayout、FrameLayout、RelativeLayout。俗话说条条大路通罗马，同一个界面我们可以使用不同的容器控件来表达，但是各个容器控件描述界面的复杂度是不一样的。一般来说LinearLayout最易，RelativeLayout较复杂。但是尺有所短，寸有所长，LinearLayout只能用来描述一个方向上连续排列的控件，而RelativeLayout几乎可以用于描述任意复杂度的界面。但是我又要说但是了，表达能力越强的容器控件，性能往往略低一些，因为系统需要将更多的时间花在计算子控件的位置上。综上所述：LinearLayout易用，效率高，表达能力有限。RelativeLayout复杂，表达能力强，效率稍逊。

那么对于同一界面而言，作为开发者考虑是使用尽量少的、表达能力强的RelativeLayout作为容器，还是选择多个、表达能力稍弱的LinearLayout来展示。从减少overdraw的角度来看，LinearLayout会增加控件数的层级，自然是RelativeLayout更优，但是当某一界面在使用LinearLayout并不会比RelativeLayout带来更多的控件数和控件层级时，LinearLayout则是首选。所以在表达界面的时候，作为一个有前瞻性的开发者要根据实际情况来选择合适容器控件，在保证性能的同时，尽量避免overdraw。

**第二招：去掉window的默认背景**

当我们使用了Android自带的一些主题时，window会被默认添加一个纯色的背景，这个背景是被DecorView持有的。当我们的自定义布局时又添加了一张背景图或者设置背景色，那么DecorView的background此时对我们来说是无用的，但是它会产生一次Overdraw，带来绘制性能损耗。

去掉window的背景可以在onCreate()中setContentView()之后调用

getWindow().setBackgroundDrawable(null);

或者在theme中添加

android:windowbackground="null"；

**第三招：去掉其他不必要的背景**

有时候为了方便会先给Layout设置一个整体的背景，再给子View设置背景，这里也会造成重叠，如果子View宽度mach\_parent，可以看到完全覆盖了Layout的一部分，这里就可以通过分别设置背景来减少重绘。再比如如果采用的是selector的背景，将normal状态的color设置为“@android:color/transparent”,也同样可以解决问题。这里只简单举两个例子，我们在开发过程中的一些习惯性思维定式会带来不经意的Overdraw，所以开发过程中我们为某个View或者ViewGroup设置背景的时候，先思考下是否真的有必要，或者思考下这个背景能不能分段设置在子View上，而不是图方便直接设置在根View上。

**第四招：ClipRect & QuickReject**

为了解决Overdraw的问题，Android系统会通过避免绘制那些完全不可见的组件来尽量减少消耗。但是不幸的是，对于那些过于复杂的自定义的View(通常重写了onDraw方法)，Android系统无法检测在onDraw里面具体会执行什么操作，系统无法监控并自动优化，也就无法避免Overdraw了。但是我们可以通过canvas.clipRect()来帮助系统识别那些可见的区域。这个方法可以指定一块矩形区域，只有在这个区域内才会被绘制，其他的区域会被忽视。这个API可以很好的帮助那些有多组重叠组件的自定义View来控制显示的区域。同时clipRect方法还可以帮助节约CPU与GPU资源，在clipRect区域之外的绘制指令都不会被执行，那些部分内容在矩形区域内的组件，仍然会得到绘制。除了clipRect方法之外，我们还可以使用canvas.quickreject()来判断是否没和某个矩形相交，从而跳过那些非矩形区域内的绘制操作。

**第五招：ViewStub**

ViewStub是个什么东西？一句话总结：高效占位符。

我们经常会遇到这样的情况，运行时动态根据条件来决定显示哪个View或布局。常用的做法是把View都写在上面，先把它们的可见性都设为View.GONE，然后在代码中动态的更改它的可见性。这样的做法的优点是逻辑简单而且控制起来比较灵活。但是它的缺点就是，耗费资源。虽然把View的初始可见View.GONE但是在Inflate布局的时候View仍然会被Inflate，也就是说仍然会创建对象，会被实例化，会被设置属性。也就是说，会耗费内存等资源。

推荐的做法是使用android.view.ViewStub，ViewStub是一个轻量级的View，它一个看不见的，不占布局位置，占用资源非常小的控件。可以为ViewStub指定一个布局，在Inflate布局的时候，只有ViewStub会被初始化，然后当ViewStub被设置为可见的时候，或是调用了ViewStub.inflate()的时候，ViewStub所向的布局就会被Inflate和实例化，然后ViewStub的布局属性都会传给它所指向的布局。这样，就可以使用ViewStub来方便的在运行时，要还是不要显示某个布局。

<ViewStub

    android:id="@+id/stub\_view"

    android:inflatedId="@+id/panel\_stub"

    android:layout="@layout/progress\_overlay"

    android:layout\_width="fill\_parent"

    android:layout\_height="wrap\_content"

android:layout\_gravity="bottom" />

当你想加载布局时，可以使用下面其中一种方法：

((ViewStub) findViewById(R.id.stub\_view)).setVisibility(View.VISIBLE);

View importPanel = ((ViewStub) findViewById(R.id.stub\_view)).inflate();

**第六招：Merge**

Merge标签有什么用呢？简单粗暴点回答：干掉一个view层级。

Merge的作用很明显，但是也有一些使用条件的限制。有两种情况下我们可以使用Merge标签来做容器控件。第一种子视图不需要指定任何针对父视图的布局属性，就是说父容器仅仅是个容器，子视图只需要直接添加到父视图上用于显示就行。另外一种是假如需要在LinearLayout里面嵌入一个布局（或者视图），而恰恰这个布局（或者视图）的根节点也是LinearLayout，这样就多了一层没有用的嵌套，无疑这样只会拖慢程序速度。而这个时候如果我们使用merge根标签就可以避免那样的问题。另外Merge只能作为XML布局的根标签使用，当Inflate以开头的布局文件时，必须指定一个父ViewGroup，并且必须设定attachToRoot为true。

举个简单的例子吧：

<RelativeLayout

xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent" >

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="merge标签使用" />

</RelativeLayout>

把上面这个XML加载到页面中，布局层级是RelativeLayout-TextView。但是采用下面的方式，把RelativeLayout提换成merge，RelativeLayout这一层级就被干掉了。

<merge

xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent" >

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="merge标签使用" />

</merge>

**第七招：善用draw9patch**

给ImageView加一个边框，你肯定遇到过这种需求，通常在ImageView后面设置一张背景图，露出边框便完美解决问题，此时这个ImageView，设置了两层drawable，底下一层仅仅是为了作为图片的边框而已。但是两层drawable的重叠区域去绘制了两次，导致overdraw。

优化方案： 将背景drawable制作成draw9patch，并且将和前景重叠的部分设置为透明。由于Android的2D渲染器会优化draw9patch中的透明区域，从而优化了这次overdraw。 但是背景图片必须制作成draw9patch才行，因为Android 2D渲染器只对draw9patch有这个优化，否则，一张普通的Png，就算你把中间的部分设置成透明，也不会减少这次overdraw。

**第八招：慎用Alpha**

假如对一个View做Alpha转化，需要先将View绘制出来，然后做Alpha转化，最后将转换后的效果绘制在界面上。通俗点说，做Alpha转化就需要对当前View绘制两遍，可想而知，绘制效率会大打折扣，耗时会翻倍，所以Alpha还是慎用。

如果一定做Alpha转化的话，可以采用缓存的方式。

view.setLayerType(LAYER\_TYPE\_HARDWARE);

doSmoeThing();

view.setLayerType(LAYER\_TYPE\_NONE);

通过setLayerType方式可以将当前界面缓存在GPU中，这样不需要每次绘制原始界面，但是GPU内存是相当宝贵的，所以用完要马上释放掉。

**第九招：避免“OverDesign”**

overdraw会给APP带来不好的体验，overdraw产生的原因无外乎：复杂的Layout层级，重叠的View，重叠的背景这几种。开发人员无节制的View堆砌，究其根本无非是产品无节制的需求设计。有道是“由俭入奢易，由奢入俭难”，很多APP披着过度设计的华丽外衣，却忘了简单易用才是王道的本质，纷繁复杂的设计并不会给用户带来好的体验，反而会让用户有压迫感，产品本身也有可能因此变得卡顿。当然，一切抛开业务谈优化都是空中楼阁，这就需要产品设计也要有一个权衡，在复杂的业务逻辑与简单易用的界面展现中做一个平衡，而不是一味的OverDesign。

# 规范

* 1. 客户端API接口规范

### 1、请求协议

所有与服务端交互的API，都走**HTTP(S)**协议。

### 2、请求参数

API的请求参数分两部分，一部分为**协议参数**，属于协议本身，与业务关务，如t,imei,appkey等；一部分为**业务参数**，这些参数会对系统中的系统流程、操作方式产生重要的影响。

**协议参数**

服务端约定，客户端需在每次请求的URL里面，加上下列协议参数：

| 参数名 | 是否必须 | 说明 |
| --- | --- | --- |
| imei | 必须 | INTERNATIONAL\_MOBILE\_EQUIPMENT\_IDENTITY,客户端设备标识，国际移动设备身份码 |
| imsi | 必须 | INTERNATIONAL\_MOBILE\_SUBSCRIBER\_IDENTITIFICATION\_NUMBER,客户端用户标识 |
| t | 必须 | TIMESTAMP,请求的时间戳，UTC 1970-1-1 0时到现在的时间差，精确到秒 |
| appkey | 必须 | 由服务端颁发的appkey |
| sign | 必须 | md5签名串 |
| lng | 可选 | LONGITUDE,手机上获取的经度 |
| lat | 可选 | LATITUDE,手机上获取的纬度 |
| ci | 可选 | CHANNEL\_IDENTITY,渠道标识，格式为：channelId@应用名平台客户端版本，例如：1001@nzaom\_android\_1.0，其中1001表示应用宝 |

**业务参数**

业务具体参数由服务端在接口文档中进行约定。

#### 2.1、参数传递

协议参数必须通过url parameter传递，业务参数通过body，path，url parameter等多样化的形式传递。

服务端提供标准的Restful API，支持**POST\DELETE\PUT\PATCH\GET**方式。

下面两个示例分别演示GET和POST的请求:

GET

GET http://localhost:80/api/testGet?appkey=123456&data=%7B%22name%22%3A%22%E5%A4%A7%E7%99%BD%22%2C%22sex%22%3A%22%E7%94%B7%22%7D&ci=1001\_nzaom\_android\_1.0&imei=imei11111&imsi=imsi22222&lat=23.1&lng=111.21&t=1432747514991&sign=760b2c16ddd8b47288b0e16871fbd5de HTTP/1.1

Host: localhost:80

Connection: Keep-Alive

User-Agent: Apache-HttpClient/4.4.1 (Java/1.7.0\_51)

Accept-Encoding: gzip,deflate

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Expires: Thu, 01 Jan 1970 00:00:00 GMT

Set-Cookie: SESSION=7a449d11-9a09-45b4-9c49-f43f0131ee51;Path=/

Content-Length: 49

Server: Jetty(6.1.22)

{"data": {"name":"大白","sex":"男"},message:"OK","status": 200}

POST

POST http://localhost:80/api/testPost?appkey=123456&data=%7B%22name%22%3A%22%E5%A4%A7%E7%99%BD%22%2C%22sex%22%3A%22%E7%94%B7%22%7D&hci=1001\_hehuyou\_android\_1.0&imei=imei11111&imsi=imsi22222&lat=23.1&lng=111.21&t=1432747714602&sign=2dfb020566c7d826e3ed7276c7c49fb8 HTTP/1.1

Content-Type: application/json;charset=UTF-8

Content-Length: 29

Host: localhost:80

Connection: Keep-Alive

User-Agent: Apache-HttpClient/4.4.1 (Java/1.7.0\_51)

Accept-Encoding: gzip,deflate

{"name":"大白","sex":"男"}

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Expires: Thu, 01 Jan 1970 00:00:00 GMT

Set-Cookie: SESSION=13e853c2-2f55-43c6-b4f0-cb016590141a;Path=/

Content-Length: 49

Server: Jetty(6.1.22)

{"message": "权限不够","status": 403}

#### 2.2 参数Sign

为了减轻非法恶意请求，每次来自APP的请求都需要对请求参数进行签名以实现安全认证。通过参数sign来验证参数传递是否合法，具体流程如下：

1. 根据参数名称（除签名sign）将所有请求参数按照字母先后顺序排序:key + value …. key + value，例如：将foo=1,bar=2,baz=3 排序为bar=2,baz=3,foo=1，参数名和参数值链接后，得到拼装字符串bar2baz3foo1
2. 将secret拼接到参数字符串尾进行md5加密后，再转化成大写，格式是：byte2hex(md5(secretkey1value1key2value2…secret))
3. 第2步生成的字符串即为参数sign

**参考代码（Java版本）：**

TreeMap<String, String> tree=new TreeMap<String, String>();

Enumeration<String> enums = request.getParameterNames();

String param="";

while(enums.hasMoreElements()){

String ele = enums.nextElement();

tree.put(ele, request.getParameter(ele));

}

Iterator<String> it = tree.keySet().iterator();

while( it.hasNext() ){

String v=it.next();

//System.out.println(" "+v+" decode:"+java.net.URLDecoder.decode(request.getParameter(v),"utf-8"));

if(!"sign".equals(v)){

param+=v+java.net.URLDecoder.decode(request.getParameter(v),"utf-8");

}

}

param = param+SECRET;

String sign2=Md5.sign(param).toUpperCase();

System.out.println("sign:"+sign);

System.out.println("sign2:"+sign2);

if(!sign2.equalsIgnoreCase(sign)){

System.out.println("非法访问");

}

**注意事项：**

* 所有的参数皆为utf-8格式

#### 2.3 关于登录

服务端和客户端都是采用基于Header的用户会话状态解决方案。登录后需要将服务端返回的token设置在Header的参数X-Auth-Token中。

#### 2.3 关于Header

目前必须传递的两个Header参数分别是：

X-Auth-Token

X-Platform

其中X-Auth-Token为会话token，X-Platform为请求的终端系统，如iOS、Android。如果涉及到API版本，请将版本的参数设置在Header中，Header的key为：**X-Api-Version**

### 3、响应结果

服务端都以json格式返回，分成功、失败两种状态。

成功：

{"data": {"name":"大白","sex":"男"},message:"OK","status": 200}

{"data": ["a","b","c"],message:"OK","status": 200}

失败：

{"message": "权限不够","status": 403}

**4、关于错误码**

400：非法请求

401：未登陆

403：权限不够

500：系统错误

900：业务级别的错误

200：正常请求

## 编码规范

源码分类-按项目的模块进行分类，方便代码管理。

例如：

com.xxxx.xxx模块

com.xxxx.xxx模块

com.xxxx.xxx模块

com.xxxx.util………

### 6.2.1源文件基础

1 文件名

源文件以其最顶层的类名来命名，大小写敏感，文件扩展名为.java。

2 文件编码：UTF-8

源文件编码格式为 UTF-8。

3 特殊字符

3.1 空白字符

除了行结束符序列，ASCII水平空格字符(0x20，即空格)是源文件中唯一允许出现的空白字符，这意味着：

1. 所有其它字符串中的空白字符都要进行转义。
2. 制表符不用于缩进（可以在IDE中Tab键设置为若干个空格）。

3.2 特殊转义序列

对于具有特殊转义序列的任何字符(\b, \t, \n, \f, \r, \", \'及)，我们使用它的转义序列，而不是相应的八进制(比如\012)或Unicode(比如\u000a)转义。

3.3 非ASCII字符

对于剩余的非ASCII字符，是使用实际的Unicode字符(比如∞)，还是使用等价的Unicode转义符(比如\u221e)，取决于哪个能让代码更易于阅读和理解。

### 6.2.2 源文件结构

一个源文件包含(按顺序地)：

1许可证或版权信息(如有需要)

2 package语句

3 import语句

4一个顶级类(只有一个)以上每个部分之间用一个空行隔开。

**1 许可证或版权信息**

如果一个文件包含许可证或版权信息，那么它应当被放在文件最前面。

**2 package语句**

package 语句不换行，列限制(4.4节)并不适用于package语句。(即package语句写在一行里)

**3 import语句**

3.1 import不要使用通配符

即，不要出现类似这样的import语句：import java.util.\*;

3.2 不要换行

import语句不换行，列限制(4.4节)并不适用于import语句。(每个import语句独立成行)

3.3 顺序和间距

import语句可分为以下几组，按照这个顺序，每组由一个空行分隔：

所有的静态导入独立成组

com.google imports(仅当这个源文件是在com.google包下)

第三方的包。每个顶级包为一组，字典序。例如：android, com, junit, org, sun

java imports5.javax imports组内不空行，按字典序排列。

**4 类声明**

4.1 只有一个顶级

类声明每个顶级类都在一个与它同名的源文件中(当然，还包含.java后缀)。

例外：package-info.java，该文件中可没有package-info类。

4.2 类成员顺序

类的成员顺序对易学性有很大的影响，但这也不存在唯一的通用法则。不同的类对成员的排序可能是不同的。

最重要的一点，每个类应该以某种逻辑去排序它的成员，维护者应该要能解释这种排序逻辑。比如， 新的方法不能总是习惯性地添加到类的结尾，因为这样就是按时间顺序而非某种逻辑来排序的。

4.2.1 区块划分

建议使用注释将源文件分为明显的区块，区块划分如下

常量声明区

UI控件成员变量声明区

普通成员变量声明区

内部接口声明区

初始化相关方法区

事件响应方法区

普通逻辑方法区

重载的逻辑方法区

发起异步任务方法区

异步任务回调方法区

生命周期回调方法区（出去onCreate()方法）

内部类声明区

4.2.2 类成员排列通用规则

按照发生的先后顺序排列

常量按照使用先后排列

UI控件成员变量按照layout文件中的先后顺序排列

普通成员变量按照使用的先后顺序排列

方法基本上都按照调用的先后顺序在各自区块中排列

相关功能作为小区块放在一起（或者封装掉）

4.2.3 重载：永不分离

当一个类有多个构造函数，或是多个同名方法，这些函数/方法应该按顺序出现在一起，中间不要放进其它函数/方法。

### 6.2.3 格式术语

说明：块状结构(block-like construct)指的是一个类，方法或构造函数的主体。需要注意的是，数组初始化中的初始值可被选择性地视为块状结构(4.8.3.1节)。

1 大括号

1.1 使用大括号(即使是可选的)

大括号与if, else, for, do, while语句一起使用，即使只有一条语句(或是空)，也应该把大括号写上。

1.2 非空块：K & R 风格

对于非空块和块状结构，大括号遵循 Kernighan 和 Ritchie 风格 (Egyptian brackets):

左大括号前不换行

左大括号后换行

右大括号前换行

如果右大括号是一个语句、函数体或类的终止，则右大括号后换行; 否则不换行。

例如，如果右大括号后面是else或逗号，则不换行。

示例：

return new MyClass() {

@Override public void method() {

if (condition()) {

try {

something();

} catch (ProblemException e) {

recover();

}

}

}

};

1.3 空块：可以用简洁版本

一个空的块状结构里什么也不包含，大括号可以简洁地写成{}，不需要换行。

例外：如果它是一个多块语句的一部分(if/else 或 try/catch/finally) ，即使大括号内没内容，右大括号也要换行。

示例：

void doNothing() {}

2 块缩进：4个空格

每当开始一个新的块，缩进增加4个空格，当块结束时，缩进返回先前的缩进级别。缩进级别适用于代码和注释。(见4.1.2节中的代码示例)

3 一行一个语句

每个语句后要换行。

4 列限制：80或100

一个项目可以选择一行80个字符或100个字符的列限制，除了下述例外，任何一行如果超过这个字符数限制，必须自动换行。

例外：

不可能满足列限制的行(例如，Javadoc中的一个长URL，或是一个长的JSNI方法参考)。

package和import语句(见3.2节和3.3节)。

注释中那些可能被剪切并粘贴到shell中的命令行。

5 自动换行

术语说明：一般情况下，一行长代码为了避免超出列限制(80或100个字符)而被分为多行，我们称之为自动换行(line-wrapping)。我们并没有全面，确定性的准则来决定在每一种情况下如何自动换行。很多时候，对于同一段代码会有好几种有效的自动换行方式。

Tip:提取方法或局部变量可以在不换行的情况下解决代码过长的问题(是合理缩短命名长度吧)

5.1 从哪里断开

自动换行的基本准则是：更倾向于在更高的语法级别处断开。

如果在非赋值运算符处断开，那么在该符号前断开(比如+，它将位于下一行)。注意：这一点与 Google 其它语言的编程风格不同(如 C++ 和 JavaScript )。

这条规则也适用于以下"类运算符"符号：点分隔符(.)，类型界限中的 &（)，catch 块中的管道符号(catch (FooException | BarException e)

如果在赋值运算符处断开，通常的做法是在该符号后断开(比如=，它与前面的内容留在同一行)。这条规则也适用于foreach语句中的分号。

方法名或构造函数名与左括号留在同一行。

逗号(,)与其前面的内容留在同一行。

5.2 自动换行时缩进至少+8个空格

自动换行时，第一行后的每一行至少比第一行多缩进8个空格(注意：制表符不用于缩进。见2.3.1节)。当存在连续自动换行时，缩进可能会多缩进不只8个空格(语法元素存在多级时)。一般而言，两个连续行使用相同的缩进当且仅当它们开始于同级语法元素。

第4.6.3水平对齐一节中指出，不鼓励使用可变数目的空格来对齐前面行的符号。

6 空白

6.1 垂直空白

以下情况需要使用一个空行：

类内连续的成员之间：字段，构造函数，方法，嵌套类，静态初始化块，实例初始化块。 例外： 两个连续字段之间的空行是可选的，用于字段的空行主要用来对字段进行逻辑分组。

在函数体内，语句的逻辑分组间使用空行。

类内的第一个成员前或最后一个成员后的空行是可选的(既不鼓励也不反对这样做，视个人喜好而定)。

要满足本文档中其他节的空行要求(比如3.3节：import语句)

多个连续的空行是允许的，但没有必要这样做(我们也不鼓励这样做)。

6.2 水平空白

除了语言需求和其它规则，并且除了文字，注释和Javadoc用到单个空格，单个ASCII空格也出现在以下几个地方：

分隔任何保留字与紧随其后的左括号(()(如if, for catch等)。

分隔任何保留字与其前面的右大括号(})(如else, catch)。

在任何左大括号前({)，两个例外：

o @SomeAnnotation({a, b})(不使用空格)。

o String[][] x = foo;(大括号间没有空格，见下面的Note)。

在任何二元或三元运算符的两侧。这也适用于以下"类运算符"符号：

o 类型界限中的&()。

o catch块中的管道符号(catch (FooException | BarException e)。

o foreach语句中的分号。

在, : ;及右括号())后

如果在一条语句后做注释，则双斜杠(//)两边都要空格。这里可以允许多个空格，但没有必要。

类型和变量之间：List list。

数组初始化中，大括号内的空格是可选的，即new int[] {5, 6}和new int[] { 5, 6 }都是可以的。

Note：这个规则并不要求或禁止一行的开关或结尾需要额外的空格，只对内部空格做要求。

6.3 水平对齐：不做要求

术语说明：水平对齐指的是通过增加可变数量的空格来使某一行的字符与上一行的相应字符对齐。

这是允许的(而且在不少地方可以看到这样的代码)，但Google编程风格对此不做要求。即使对于已经使用水平对齐的代码，我们也不需要去保持这种风格。

以下示例先展示未对齐的代码，然后是对齐的代码：

private int x; // this is fine

private Color color; // this too

private int x; // permitted, but future edits

private Color color; // may leave it unaligned

Tip：对齐可增加代码可读性，但它为日后的维护带来问题。考虑未来某个时候，我们需要修改一堆对齐的代码中的一行。

这可能导致原本很漂亮的对齐代码变得错位。很可能它会提示你调整周围代码的空白来使这一堆代码重新水平对齐(比如程序员想保持这种水平对齐的风格)。

这就会让你做许多的无用功，增加了reviewer的工作并且可能导致更多的合并冲突。

7 用小括号来限定组：推荐

除非作者和reviewer都认为去掉小括号也不会使代码被误解，或是去掉小括号能让代码更易于阅读，否则我们不应该去掉小括号。

我们没有理由假设读者能记住整个Java运算符优先级表。

8 具体结构

8.1 枚举类

枚举常量间用逗号隔开，换行可选。

没有方法和文档的枚举类可写成数组初始化的格式：

private enum Suit {

CLUBS,

HEARTS,

SPADES,

DIAMONDS

}

由于枚举类也是一个类，因此所有适用于其它类的格式规则也适用于枚举类。

8.2 变量声明

8.2.1 每次只声明一个变量

不要使用组合声明，比如int a, b;。

8.2.2 需要时才声明，并尽快进行初始化

不要在一个代码块的开头把局部变量一次性都声明了(这是c语言的做法)，而是在第一次需要使用它时才声明。 局部变量在声明时最好就进行初始化，或者声明后尽快进行初始化。

8.3 数组

8.3.1 数组初始化：可写成块状结构

数组初始化可以写成块状结构，比如，下面的写法都是OK的：

new int[] {

0, 1, 2, 3

}

new int[] {

0,

1,

2,

3

}

new int[] {

0, 1,

2, 3

}

new int[]

{0, 1, 2, 3}

8.3.2 非C风格的数组声明

中括号是类型的一部分：String[] args， 而非 String args[]。

8.4 switch语句

术语说明：switch块的大括号内是一个或多个语句组。

每个语句组包含一个或多个switch标签(case FOO:或default:)，后面跟着一条或多条语句。

8.4.1 缩进

与其它块状结构一致，switch块中的内容缩进为2个空格。每个switch标签后新起一行，再缩进2个空格，写下一条或多条语句。

8.4.2 Fall-through：注释

在一个switch块内，每个语句组要么通过break, continue, return或抛出异常来终止，要么通过一条注释来说明程序将继续执行到下一个语句组， 任何能表达这个意思的注释都是OK的(典型的是用// fall through)。这个特殊的注释并不需要在最后一个语句组(一般是default)中出现。

示例：

switch (input) {

case 1:

case 2:

prepareOneOrTwo(); // fall through

case 3:

handleOneTwoOrThree();

break;

default:

handleLargeNumber(input);

}

8.4.3 default的情况要写出来

每个switch语句都包含一个default语句组，即使它什么代码也不包含。

8.5 注解(Annotations)

注解紧跟在文档块后面，应用于类、方法和构造函数，一个注解独占一行。这些换行不属于自动换行(第4.5节，自动换行)，因此缩进级别不变。

例如：

@Nullable public String getNameIfPresent() { ... }

例外：单个的注解可以和签名的第一行出现在同一行。

例如：

@Override public int hashCode() { ... }应用于字段的注解紧随文档块出现，应用于字段的多个注解允许与字段出现在同一行。

例如：

@Partial @Mock DataLoader loader;

参数和局部变量注解没有特定规则。

8.6 注释

8.6.1 块注释风格

块注释与其周围的代码在同一缩进级别。它们可以是/ ... /风格，也可以是// ...风格。对于多行的/ ... /注释，后续行必须从开始， 并且与前一行的对齐。

以下示例注释都是OK的。

/\*\* This is // And so /\* Or you can

\* okay. // is this. \* even do this. \*/

\*/

注释不要封闭在由星号或其它字符绘制的框架里。

Tip：在写多行注释时，如果你希望在必要时能重新换行(即注释像段落风格一样)，那么使用/ ... /。

8.7 Modifiers

类和成员的modifiers如果存在，则按Java语言规范中推荐的顺序出现。

public protected private abstract static final transient volatile synchronized native strictfp

### 6.2.4 命名约定

1 对所有标识符都通用的规则

标识符只能使用ASCII字母和数字，因此每个有效的标识符名称都能匹配正则表达式\w+。

2 标识符类型的规则

2.1 包名

包名全部小写，连续的单词只是简单地连接起来，不使用下划线。

采用反域名命名规则，全部使用小写字母。一级包名为com，二级包名为xx（可以是公司或则个人的随便），三级包名根据应用进行命名，四级包名为模块名或层级名。

例如：com.jiashuangkuaizi.kitchen

包名 此包中包含

com.xx.应用名称缩写.activity 页面用到的Activity类 (activitie层级名用户界面层)

com.xx.应用名称缩写.base 基础共享的类

com.xx.应用名称缩写.adapter 页面用到的Adapter类 (适配器的类)

com.xx.应用名称缩写.util 此包中包含：公共工具方法类（util模块名）

com.xx.应用名称缩写.bean 下面可分：vo、po、dto 此包中包含：JavaBean类

com.xx.应用名称缩写.model 此包中包含：模型类

com.xx.应用名称缩写.db 数据库操作类

com.xx.应用名称缩写.view

(或者 com.xx.应用名称缩写.widget ) 自定义的View类等

com.xx.应用名称缩写.service Service 服务

com.xx.应用名称缩写.receiver BroadcastReceiver服务

注意：

如果项目采用MVP，所有M、V、P抽取出来的接口都放置在相应模块的i包下，所有的实现都放置在相应模块的impl下

2.2 类名

类名都以UpperCamelCase风格编写。

UpperCamelCase，第一个词Upper使得它与lowerCamelCase 名字容易地能很容易的区别出来。

类名通常是名词或名词短语，接口名称有时可能是形容词或形容词短语。现在还没有特定的规则或行之有效的约定来命名注解类型。

名词，采用大驼峰命名法，尽量避免缩写，除非该缩写是众所周知的， 比如HTML,URL，如果类名称中包含单词缩写，则单词缩写的每个字母均应大写。

类 描述 例如

Activity 类 Activity为后缀标识 欢迎页面类WelcomeActivity

Adapter类 Adapter 为后缀标识 新闻详情适配器 NewDetailAdapter

解析类 Parser为后缀标识 首页解析类HomePosterParser

工具方法类 Util或Manager为后缀标识（与系统或第三方的Utils区分）或功能+Util 线程池管理类：ThreadPoolManager

日志工具类：LogUtil（Logger也可）

打印工具类：PrinterUtil

数据库类 以DBHelper后缀标识 新闻数据库：NewDBHelper

Service类 以Service为后缀标识 时间服务TimeServiceBroadcast

Receiver类 以Receiver为后缀标识 推送接收JPushReceiver

ContentProvider 以Provider为后缀标识

自定义的共享基础类 以Base开头 BaseActivity,BaseFragment

测试类的命名以它要测试的类的名称开始，以Test结束。

例如：HashTest 或 HashIntegrationTest。

接口（interface）：命名规则与类一样采用大驼峰命名法，多以able或ible结尾，如

interface Runnable ;

interface Accessible。

注意：

如果项目采用MVP，所有Model、View、Presenter的接口都以I为前缀，不加后缀，其他的接口采用上述命名规则。

2.3 方法名

方法名都以 LowerCamelCase 风格编写。

LowerCamelCase是一种命名惯例，它的名字是由多个和在一起的词组合成的一个单词且它们的每个词（除了第一个词以外）的第一个字母大写。

方法名通常是动词或动词短语。

方法 说明

initXX() 初始化相关方法,使用init为前缀标识，如初始化布局initView()

isXX() checkXX() 方法返回值为boolean型的请使用is或check为前缀标识

getXX() 返回某个值的方法，使用get为前缀标识

handleXX() 对数据进行处理的方法，尽量使用handle为前缀标识

displayXX()/showXX() 弹出提示框和提示信息，使用display/show为前缀标识

saveXX() 与保存数据相关的，使用save为前缀标识

resetXX() 对数据重组的，使用reset前缀标识

clearXX() 清除数据相关的

removeXXX() 清除数据相关的

drawXXX() 绘制数据或效果相关的，使用draw前缀标识

下划线可能出现在JUnit测试方法名称中用以分隔名称的逻辑组件。一个典型的模式是：test\_，例如testPop\_emptyStack。

并不存在唯一正确的方式来命名测试方法。

2.4 常量名

常量名命名模式为CONSTANT\_CASE，全部字母大写，用下划线分隔单词。那，到底什么算是一个常量？

每个常量都是一个静态final字段，但不是所有静态final字段都是常量。在决定一个字段是否是一个常量时，考虑它是否真的感觉像是一个常量。

例如，如果任何一个该实例的观测状态是可变的，则它几乎肯定不会是一个常量。只是永远不打算改变对象一般是不够的，它要真的一直不变才能将它示为常量。

// Constants

static final int NUMBER = 5;

static final ImmutableListNAMES = ImmutableList.of("Ed", "Ann");

static final Joiner COMMA\_JOINER = Joiner.on(','); // because Joiner is immutable

static final SomeMutableType[] EMPTY\_ARRAY = {};

enum SomeEnum { ENUM\_CONSTANT }

// Not constants

static String nonFinal = "non-final";

final String nonStatic = "non-static";

static final SetmutableCollection = new HashSet();

static final ImmutableSetmutableElements = ImmutableSet.of(mutable);

static final Logger logger = Logger.getLogger(MyClass.getName());

static final String[] nonEmptyArray = {"these", "can", "change"};

这些名字通常是名词或名词短语。

2.5 非常量字段名

非常量字段名以LowerCamelCase风格的基础上改造为如下风格：

基本结构为scopeVariableNameType，

scope：范围

非公有，非静态字段命名以m开头。

静态字段命名以s开头。

公有非静态字段命名以p开头。

公有静态字段（全局变量）命名以g开头。

public static final 字段(常量) 全部大写，并用下划线连起来。

例子：

public class MyClass {

public static final int SOME\_CONSTANT = 42;

public int pField;

private static MyClass sSingleton;

int mPackagePrivate;

private int mPrivate;

protected int mProtected;

public static int gField;

}

使用1个字符前缀来表示作用范围，1个字符的前缀必须小写，前缀后面是由表意性强的一个单词或多个单词组成的名字，而且每个单词的首写字母大写，其它字母小写，这样保证了对变量名能够进行正确的断句。

Type：类型

考虑到Android中使用很多UI控件，为避免控件和普通成员变量混淆以及更好达意，所有用来表示控件的成员变量统一加上控件缩写作为后缀（文末附有缩写表）。

对于普通变量一般不添加类型后缀，如果统一添加类型后缀，请参考文末的缩写表。

用统一的量词通过在结尾处放置一个量词，就可创建更加统一的变量，它们更容易理解，也更容易搜索。

注意：如果项目中使用ButterKnife，则不添加m前缀，以LowerCamelCase风格命名。

例如，请使用 mCustomerStrFirst 和 mCustomerStrLast，而不要使用mFirstCustomerStr和mLastCustomerStr。

量词列表：量词后缀说明

First 一组变量中的第一个

Last 一组变量中的最后一个

Next 一组变量中的下一个变量

Prev 一组变量中的上一个

Cur 一组变量中的当前变量。

说明：

集合添加如下后缀：List、Map、Set

数组添加如下后缀：Arr

注意：所有的VO（值对象）统一采用标准的lowerCamelCase风格编写，所有的DTO（数据传输对象）就按照接口文档中定义的字段名编写。

2.6 参数名

参数名以LowerCamelCase风格编写

2.7 局部变量名

局部变量名以LowerCamelCase风格编写，比起其它类型的名称，局部变量名可以有更为宽松的缩写。

虽然缩写更宽松，但还是要避免用单字符进行命名，除了临时变量和循环变量。

即使局部变量是final和不可改变的，也不应该把它示为常量，自然也不能用常量的规则去命名它。

临时变量

临时变量通常被取名为i，j，k，m和n，它们一般用于整型；c，d，e，它们一般用于字符型。 如： for (int i = 0; i < len ; i++)，并且它和第一个单词间没有空格。

2.8 类型变量名

类型变量可用以下两种风格之一进行命名：

单个的大写字母，后面可以跟一个数字(如：E, T, X, T2)。

以类命名方式(2.2)，后面加个大写的T(如：RequestT, FooBarT)。

2.9 资源文件命名规范

1. 资源布局文件（XML文件（layout布局文件））：

全部小写，采用下划线命名法

1) contentview 命名

必须以全部单词小写，单词间以下划线分割，使用名词或名词词组。

所有Activity或Fragment的contentView必须与其类名对应，对应规则为：

将所有字母都转为小写，将类型和功能调换（也就是后缀变前缀）。

例如：activity\_main.xml

2) Dialog命名：dialog\_描述.xml

例如：dialog\_hint.xml

3) PopupWindow命名：ppw\_描述.xml

例如：ppw\_info.xml

4) 列表项命名：item\_描述.xml

例如：item\_city.xml

5) 包含项命名：模块\_(位置)描述.xml

例如：activity\_main\_head.xml、activity\_main\_bottom.xml

注意：通用的包含项命名采用：项目名称缩写\_描述.xml

例如：xxxx\_title.xml

2. 资源文件（图片drawable文件夹下）：

全部小写，采用下划线命名法，加前缀区分

命名模式：可加后缀 \_small 表示小图, \_big 表示大图，逻辑名称可由多个单词加下划线组成，采用以下规则：

用途\_模块名\_逻辑名称

用途\_模块名\_颜色

用途\_逻辑名称

用途\_颜色

说明：用途也指控件类型（具体见UI控件缩写表）

例如：

btn\_main\_home.png 按键

divider\_maket\_white.png 分割线

ic\_edit.png 图标

bg\_main.png 背景

btn\_red.png 红色按键

btn\_red\_big.png 红色大按键

ic\_head\_small.png 小头像

bg\_input.png 输入框背景

divider\_white.png 白色分割线

如果有多种形态如按钮等除外如 btn\_xx.xml（selector）

名称 功能

btn\_xx 按钮图片使用btn\_整体效果（selector）

btn\_xx\_normal 按钮图片使用btn\_正常情况效果

btn\_xx\_pressed 按钮图片使用btn\_点击时候效果

btn\_xx\_focused state\_focused聚焦效果

btn\_xx\_disabled state\_enabled (false)不可用效果

btn\_xx\_checked state\_checked选中效果

btn\_xx\_selected state\_selected选中效果

btn\_xx\_hovered state\_hovered悬停效果

btn\_xx\_checkable state\_checkable 可选效果

btn\_xx\_activated state\_activated 激活的

btn\_xx\_windowfocused state\_window\_focused

bg\_head 背景图片使用bg\_功能\_说明

def\_search\_cell 默认图片使用def\_功能\_说明

ic\_more\_help 图标图片使用ic\_功能\_说明

seg\_list\_line 具有分隔特征的图片使用seg\_功能\_说明

sel\_ok 选择图标使用sel\_功能\_说明

注意：

使用AndroidStudio的插件SelectorChapek可以快速生成selector，前提是命名要规范。

3. 动画文件（anim文件夹下）：

全部小写，采用下划线命名法，加前缀区分。

具体动画采用以下规则：

模块名\_逻辑名称

逻辑名称

refresh\_progress.xml

market\_cart\_add.xml

market\_cart\_remove.xml

普通的tween动画采用如下表格中的命名方式

// 前面为动画的类型，后面为方向

动画命名例子 规范写法

fade\_in 淡入

fade\_out 淡出

push\_down\_in 从下方推入

push\_down\_out 从下方推出

push\_left 推向左方

slide\_in\_from\_top 从头部滑动进入

zoom\_enter 变形进入

slide\_in 滑动进入

shrink\_to\_middle 中间缩小

4. values中name命名

类别 命名 示例

strings strings的name命名使用下划线命名法，采用以下规则：

模块名+逻辑名称

main\_menu\_about 主菜单按键文字

friend\_title 好友模块标题栏

friend\_dialog\_del 好友删除提示

login\_check\_email 登录验证

dialog\_title 弹出框标题

button\_ok 确认键 loading 加载文字

colors colors的name命名使用下划线命名法，采用以下规则：

模块名+逻辑名称 颜色

friend\_info\_bg friend\_bg transparent gray

styles styles的name命名使用 Camel命名法，采用以下规则：模块名+逻辑名称 main\_tabBottom

5. layout中的id命名

命名模式为：view缩写\_view的逻辑名称

使用 AndroidStudio 的插件 ButterKnife Zelezny，生成注解非常方便。

如果不使用 ButterKnife Zelezny，则建议使用 view 缩写做后缀，如：username\_tv（展示用户名的TextView）

### 6.2.5 编程实践

1 @Override：能用则用

只要是合法的，就把@Override注解给用上。

2 捕获的异常：不能忽视

除了下面的例子，对捕获的异常不做响应是极少正确的。(典型的响应方式是打印日志，或者如果它被认为是不可能的，则把它当作一个 AssertionError 重新抛出。)

如果它确实是不需要在catch块中做任何响应，需要做注释加以说明(如下面的例子)。

try {

int i = Integer.parseInt(response);

return handleNumericResponse();

} catch (NumberFormatException ok) {

// it's not numeric; that's fine, just continue

}

return handleTextResponse(response);

例外：在测试中，如果一个捕获的异常被命名为expected，则它可以被不加注释地忽略。下面是一种非常常见的情形，用以确保所测试的方法会抛出一个期望中的异常，因此在这里就没有必要加注释。

try {

emptyStack.pop();

fail();

} catch (NoSuchElementException expected) {

}

3 静态成员：使用类进行调用

使用类名调用静态的类成员，而不是具体某个对象或表达式。

Foo aFoo = ...;

Foo.aStaticMethod(); // good

aFoo.aStaticMethod(); // bad

somethingThatYieldsAFoo().aStaticMethod(); // very bad

4 Finalizers: 禁用

极少会去重载Object.finalize。

不要使用finalize。如果你非要使用它，请先仔细阅读和理解Effective Java 第7条款："Avoid Finalizers"，然后不要使用它。

### 6.2.6 JavaDoc

1.格式

1.1 一般形式

Javadoc块的基本格式如下所示：

/\*\*

\* Multiple lines of Javadoc text are written here,

\* wrapped normally...

\*/

public int method(String p1) { ... }

或者是以下单行形式：

/\*\* An especially short bit of Javadoc. \*/

基本格式总是OK的。当整个Javadoc块能容纳于一行时(且没有Javadoc标记@XXX)，可以使用单行形式。

1.2 段落

空行(即，只包含最左侧星号的行)会出现在段落之间和Javadoc标记(@XXX)之前(如果有的话)。

除了第一个段落，每个段落第一个单词前都有标签<p>，并且它和第一个单词间没有空格。

1.3 JavaDoc标记

标准的Javadoc标记按以下顺序出现：@param, @return, @throws, @deprecated,

前面这4种标记如果出现，描述都不能为空。 当描述无法在一行中容纳，连续行需要至少再缩进4个空格。

2 摘要片段

每个类或成员的Javadoc以一个简短的摘要片段开始。这个片段是非常重要的，在某些情况下，它是唯一出现的文本，比如在类和方法索引中。

这只是一个小片段，可以是一个名词短语或动词短语，但不是一个完整的句子。它不会以A {@code Foo} is a...或This method returns...开头,

它也不会是一个完整的祈使句，如Save the record...。然而，由于开头大写及被加了标点，它看起来就像是个完整的句子。

一个常见的错误是把简单的Javadoc写成 /\*\* @return the customer ID \*/，这是不正确的。它应该写成/\*\* Returns the customer ID. \*/。

3 哪里需要使用Javadoc

至少在每个public类及它的每个public和protected成员处使用Javadoc，以下是一些例外：

3.1 例外：不言自明的方法

对于简单明显的方法如getFoo，Javadoc是可选的(即，是可以不写的)。这种情况下除了写"Returns the foo"，确实也没有什么值得写了。  
单元测试类中的测试方法可能是不言自明的最常见例子了，我们通常可以从这些方法的描述性命名中知道它是干什么的，因此不需要额外的文档说明。

Tip：  
如果有一些相关信息是需要读者了解的，那么以上的例外不应作为忽视这些信息的理由。例如，对于方法名getCanonicalName，

就不应该忽视文档说明，因为读者很可能不知道词语canonical name指的是什么。

3.2 例外：重载

如果一个方法重载了超类中的方法，那么Javadoc并非必需的。

3.3 可选的Javadoc

对于包外不可见的类和方法，如有需要，也是要使用Javadoc的。如果一个注释是用来定义一个类，方法，字段的整体目的或行为，  
那么这个注释应该写成Javadoc，这样更统一更友好。

表1 UI控件缩写表

| **控件** | **缩写** | **例子** |
| --- | --- | --- |
| LinearLayout | ll | llFriend或者mFriendLL |
| RelativeLayout | rl | rlMessage或mMessageRL |
| FrameLayout | fl | flCart或mCartFL |
| TableLayout | tl | tlTab或mTabTL |
| Button | btn | btnHome或mHomeBtn |
| ImageButton | ibtn | btnPlay或mPlayIBtn |
| TextView | tv | tvName或mNameTV |
| EditText | et | etName或mNameET |
| ListView | lv | lvCart或mCartLV |
| ImageView | iv | ivHead或mHeadIV |
| GridView | gv | gvPhoto或mPhotoGV |

表2 常见的英文单词缩写:

| **名称** | **缩写** |
| --- | --- |
| icon | ic （主要用在app的图标） |
| color | cl（主要用于颜色值） |
| divider | di（主要用于分隔线，不仅包括Listview中的divider，还包括普通布局中的线） |
| selector | sl（主要用于某一view多种状态，不仅包括Listview中的selector，还包括按钮的selector） |
| average | avg |
| background | bg（主要用于布局和子布局的背景） |
| buffer | buf |
| control | ctrl |
| delete | del |
| document | doc |
| error | err |
| escape | esc |
| increment | inc |
| infomation | info |
| initial | init |
| image | img |
| Internationalization | I18N |
| length | len |
| library | lib |
| message | msg |
| password | pwd |
| position | pos |
| server | srv |
| string | str |
| temp | tmp |
| window | wnd(win) |

程序中使用单词缩写原则：不要用缩写，除非该缩写是约定俗成的。

# 网络传输

* 1. **Socket**

Socket,又称为套接字，Socket是计算机网络通信的基本的技术之一。如今大多数基于网络的软件，如浏览器，即时通讯工具甚至是P2P下载都是基于Socket实现的。本文会介绍一下基于TCP/IP的Socket编程，并且如何写一个客户端/服务器程序。

Unix的输入输出(IO)系统遵循Open-Read-Write-Close这样的操作范本。当一个用户进程进行IO操作之前，它需要调用Open来指定并获取待操作文件或设备读取或写入的权限。一旦IO操作对象被打开，那么这个用户进程可以对这个对象进行一次或多次的读取或写入操作。Read操作用来从IO操作对象读取数据，并将数据传递给用户进程。Write操作用来将用户进程中的数据传递（写入）到IO操作对象。 当所有的Read和Write操作结束之后，用户进程需要调用Close来通知系统其完成对IO对象的使用。

在Unix开始支持进程间通信（InterProcess Communication，简称IPC）时，IPC的接口就设计得类似文件IO操作接口。在Unix中，一个进程会有一套可以进行读取写入的IO描述符。IO描述符可以是文件，设备或者是通信通道（socket套接字）。一个文件描述符由三部分组成：创建（打开socket），读取写入数据（接受和发送到socket）还有销毁（关闭socket）。

在Unix系统中，类BSD版本的IPC接口是作为TCP和UDP协议之上的一层进行实现的。消息的目的地使用socket地址来表示。一个socket地址是由网络地址和端口号组成的通信标识符。

进程间通信操作需要一对儿socket。进程间通信通过在一个进程中的一个socket与另一个进程中得另一个socket进行数据传输来完成。当一个消息执行发出后，这个消息在发送端的socket中处于排队状态，直到下层的网络协议将这些消息发送出去。当消息到达接收端的socket后，其也会处于排队状态，直到接收端的进程对这条消息进行了接收处理。

TCP和UDP通信

关于socket编程我们有两种通信协议可以进行选择。一种是数据报通信，另一种就是流通信。

数据报通信

数据报通信协议，就是我们常说的UDP（User Data Protocol 用户数据报协议）。UDP是一种无连接的协议，这就意味着我们每次发送数据报时，需要同时发送本机的socket描述符和接收端的socket描述符。因此，我们在每次通信时都需要发送额外的数据。

流通信

流通信协议，也叫做TCP(Transfer Control Protocol，传输控制协议)。和UDP不同，TCP是一种基于连接的协议。在使用流通信之前，我们必须在通信的一对儿socket之间建立连接。其中一个socket作为服务器进行监听连接请求。另一个则作为客户端进行连接请求。一旦两个socket建立好了连接，他们可以单向或双向进行数据传输。

读到这里，我们多少有这样的疑问，我们进行socket编程使用UDP还是TCP呢。选择基于何种协议的socket编程取决于你的具体的客户端-服务器端程序的应用场景。下面我们简单分析一下TCP和UDP协议的区别，或许可以帮助你更好地选择使用哪种。

在UDP中，每次发送数据报时，需要附带上本机的socket描述符和接收端的socket描述符。而由于TCP是基于连接的协议，在通信的socket对之间需要在通信之前建立连接，因此会有建立连接这一耗时存在于TCP协议的socket编程。

在UDP中，数据报数据在大小上有64KB的限制。而TCP中也不存在这样的限制。一旦TCP通信的socket对建立了连接，他们之间的通信就类似IO流，所有的数据会按照接受时的顺序读取。

UDP是一种不可靠的协议，发送的数据报不一定会按照其发送顺序被接收端的socket接受。然后TCP是一种可靠的协议。接收端收到的包的顺序和包在发送端的顺序是一致的。

简而言之，TCP适合于诸如远程登录(rlogin,telnet)和文件传输（FTP）这类的网络服务。因为这些需要传输的数据的大小不确定。而UDP相比TCP更加简单轻量一些。UDP用来实现实时性较高或者丢包不重要的一些服务。在局域网中UDP的丢包率都相对比较低。

* 1. **Http/Https**

**1 方法**

| **方法** | **版本** | **说明** |
| --- | --- | --- |
| GET | 1.0、1.1 | 获取资源 |
| POST | 1.0、1.1 | 传输实体主体 |
| PUT | 1.0、1.1 | 传输文件 |
| DELETE | 1.0、1.1 | 删除文件 |
| HEAD | 1.1、1.1 | 获取报文首部 |
| OPTIONS | 1.1 | 用来查询服务器支持的方法 |
| TRACE | 1.1 | 追踪路径 |
| CONNECT | 1.1 | 要求用隧道协议链接代理 |
| LINK | 1.0 | 建立和资源直接的联系 |
| UNLINE | 1.0 | 断开连接关系 |

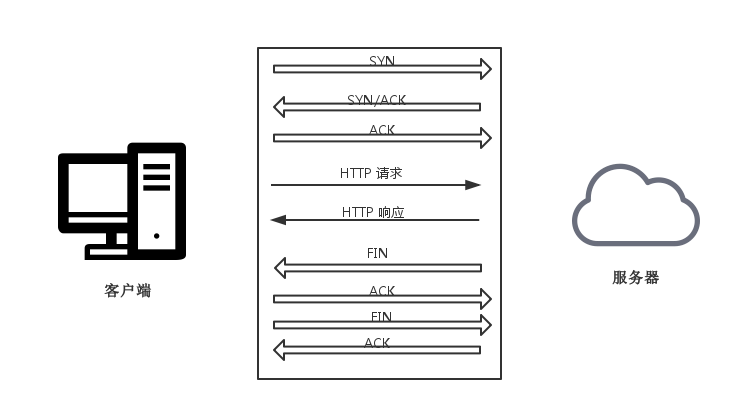
其中 HEAD 和 GET 方法类似，但是响应不包含实体部分，用于确定 URL 的有效性和过期时间;

TRACE 让 Web 服务器端将之前的请求环回路径返回给客户端的方法。发送请求时，在 Max-Forwards 首部字段中填入数值，每经过一个服务器端就减 1，当数值正好减到 0 时，停止传输，由最后接收到请求的服务器返回 200 OK 的响应。

CONNECT 要求在与代理服务器通信时建立隧道，实现用隧道协议实现 TCP 通信。主要使用 SSL 和 TLS 把通信内容加密后经网络隧道传输。

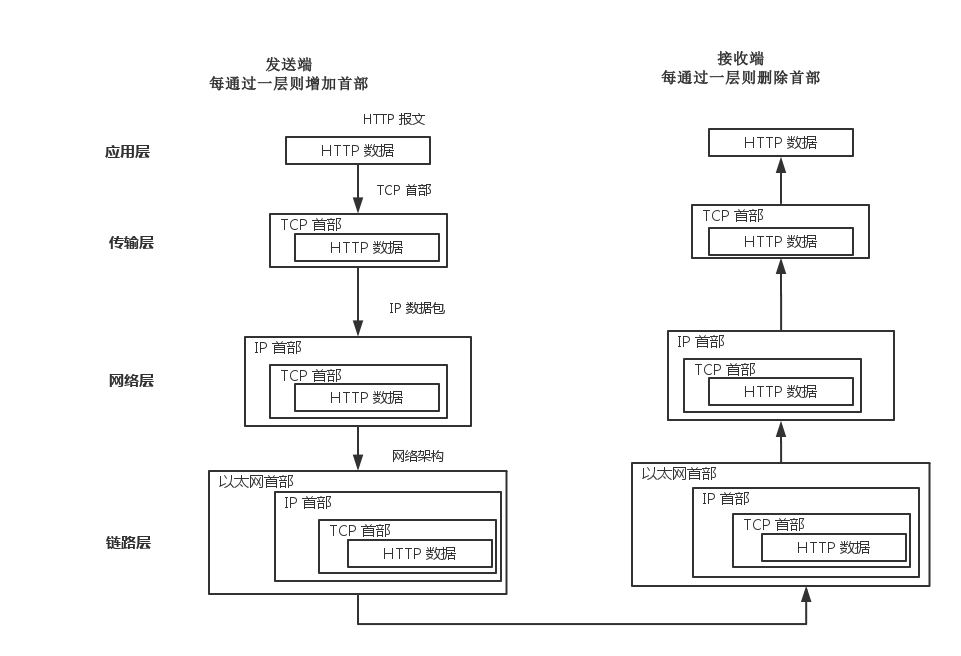
**2 TCP连接**

TCP 建立连接需要三次握手，断开连接时需要四次挥手。



**3 传输**

数据的传输过程中，从应用层到传输层，网络层，再到链路层，发送端每经过一层，就会增加相应的首部；接收端则每往上传播一层，则会剔除相应的首部，直到应用层，获取到的是原始的 HTTP 数据。



**4 状态码**

|  | **类别** | **原因短语** |
| --- | --- | --- |
| 1XX | Informational（信息性状态码） | 接收的请求正在处理 |
| 2XX | Success （成功状态码） | 请求正常，处理完毕 |
| 3XX | Redirection （重定向状态码） | 要求进行附加操作以完成请求 |
| 4XX | Client Error （客户端错误状态码） | 服务器无法处理请求 |
| 5XX | Server Error （服务器错误状态码） | 服务器处理请求出错 |

2XX：正确响应

200：正确处理；

204：No content，服务器已正确处理，但是响应实体中没有主体；

206：Partial content，对于请求头中包含 content-range 字段的，响应一部分数据；

3XX：重定向

301：Moved permanently，永久重定向，响应头中包含的 location 字段为重定向后的结果；

302：Found，临时重定向，请求的资源已被分配到新的URI，希望用户本次使用新URI（临时的意思是指以后有可能再变回来）。

303：See other, 请求的资源存在另一个URI，请以 GET 方式去获取。

304：Not modified，客户端发送附带条件（if-modified-since, if-range 等条件）的请求时，服务端资源已经找到，但是不符合条件。（304 响应中没有响应实体，和重定向没有关系）

4XX：客户端错误

400：Bad request，请求报文中有语法错误。

401：Unauthorized，请求需要 HTTP 认证（BASIC / DIGEST 认证，请求头中应包含 www-authenticate 字段）。

403：Forbidden，请求的资源被服务器拒绝了；

404：Not found，服务器上没有找到请求的资源。

5XX：服务器错误

500：Internal server error，服务器在执行请求时发生了错误。

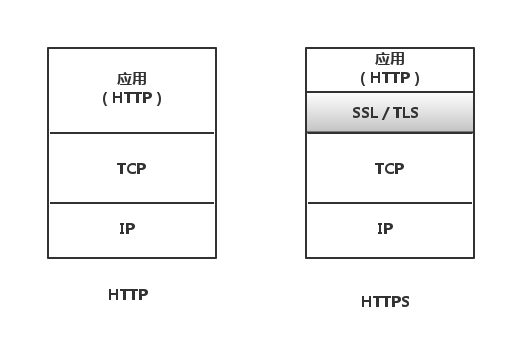
503：Service unavailable，服务器处于超负载状态或者正在进行停机维护，现在无法处理请求。

注：状态码和情况不一致，如服务器发生错误，但是仍然返回 200 的状态码，这种情况也很常见。

**5. 安全**

HTTPS

HTTPS 相对于 HTTP 就是在应用层和传输层之间添加了 SSL / TLS 。



数字签名

生成 Keypair，即密钥对，包括公钥和私钥；

对需要进行数字签名的内容进行 HASH 后得到该内容的摘要 Digest；

用私钥对 Digest 加密，得到 Signature，即数字签名；

证书

生成密钥对，服务器把自己的公钥登录至 CA 机构；

CA 机构用自己的私钥向服务器的公钥签署数字签名，并颁发公钥证书；

客户端拿到服务器的公钥证书后，使用 CA 机构的公钥，向 CA 机构验证公钥证书的数字签名，以确认服务器公钥的真实性（CA 机构的公钥已事先植入到浏览器中）；

客户端验证证书通过后，使用服务器的公钥对数据加密后，发送给服务器；

服务器接收到客户端加密的内容后，用私钥解密，获取真实的数据内容；

# UI效果相关

* 1. **背景图片切换**

<**selector xmlns:android="http://schemas.android.com/apk/res/android"**>  
 <**item  
 android:state\_pressed="true"  
 android:drawable="@drawable/btnout\_selectbg"**/>  
 <**item  
 android:state\_pressed="false"  
 android:drawable="@drawable/btnout\_normalbg"**/>  
</**selector**>

* 1. **圆角黑边效果**

<**shape xmlns:android="http://schemas.android.com/apk/res/android"**>  
 <**solid android:color="#ffffff"** /> //填充颜色  
 <**corners android:radius="10dp"** /> //圆角直径  
 <**stroke android:width="1dp" android:color="#cccccc"** /> //设置黑边  
</**shape**>

* 1. **基本的动画效果**

**淡出淡入**

*<!--   
 fromAlpha:开始时透明度   
 toAlpha： 结束时透明度   
 duration：动画持续时间 -->*

<**set xmlns:android="http://schemas.android.com/apk/res/android"**>  
<**alpha  
 android:fromAlpha="1.0"  
 android:toAlpha="0.0"  
 android:duration="500"** />  
</**set**>

**旋转效果**

*<!--   
 fromDegrees 动画开始时的角度   
 toDegrees 动画结束时物件的旋转角度,正代表顺时针   
 pivotX 属性为动画相对于物件的X坐标的开始位置   
 pivotY 属性为动画相对于物件的Y坐标的开始位置 -->*

<**set xmlns:android="http://schemas.android.com/apk/res/android"**>  
<**rotate  
 android:interpolator="@android:anim/accelerate\_decelerate\_interpolator"  
 android:fromDegrees="300"  
 android:toDegrees="-360"  
 android:pivotX="10%"  
 android:pivotY="100%"  
 android:duration="10000"** />  
</**set**>

**缩放动画**

*<!--   
fromXDelta,fromYDelta 起始时X，Y座标,屏幕右下角的座标是X:320,Y:480   
toXDelta， toYDelta 动画结束时X,Y的座标 --> <!--   
interpolator 指定动画插入器   
常见的有加速减速插入器 accelerate\_decelerate\_interpolator   
加速插入器 accelerate\_interpolator，   
减速插入器 decelerate\_interpolator。   
fromXScale,fromYScale， 动画开始前X,Y的缩放，0.0为不显示， 1.0为正常大小   
toXScale，toYScale， 动画最终缩放的倍数， 1.0为正常大小，大于1.0放大   
pivotX， pivotY 动画起始位置，相对于屏幕的百分比,两个都为50%表示动画从屏幕中间开始   
startOffset， 动画多次执行的间隔时间，如果只执行一次，执行前会暂停这段时间，   
 单位毫秒 duration，一次动画效果消耗的时间，单位毫秒，   
 值越小动画速度越快 repeatCount，动画重复的计数，动画将会执行该值+1次   
 repeatMode，动画重复的模式，reverse为反向，当第偶次执行时，动画方向会相反。   
 restart为重新执行，方向不变 -->*

<**set xmlns:android="http://schemas.android.com/apk/res/android"**>  
<**scale  
 android:interpolator= "@android:anim/decelerate\_interpolator"  
 android:fromXScale="0.0"  
 android:toXScale="1.5"  
 android:fromYScale="0.0"  
 android:toYScale="1.5"  
 android:pivotX="50%"  
 android:pivotY="50%"  
 android:startOffset="0"  
 android:duration="10000"  
 android:repeatCount="1"  
 android:repeatMode="reverse"** />  
</**set**>

**移动效果**

*<!--   
 fromXDelta,fromYDelta 起始时X，Y座标,屏幕右下角的座标是X:320,Y:480   
 toXDelta， toYDelta 动画结束时X,Y的座标 -->*

<**set xmlns:android="http://schemas.android.com/apk/res/android"**>  
<**translate  
 android:fromXDelta="320"  
 android:toXDelta="0"  
 android:fromYDelta="480"  
 android:toYDelta="0"  
 android:duration="10000"** />  
</**set**>

**8.3.1 动画事件监听**

ImageView imageView=(ImageView)findViewById(R.id.imageView);

ObjectAnimator animator=new ObjectAnimator().ofFloat(imageView, "translationX", 0f,200f);

animator.setDuration(1000);

/\*\*

\*添加所有监听事件

\*/

// animator.addListener(new AnimatorListener() {

//

// @Override

// public void onAnimationStart(Animator arg0) {

// }

//

// @Override

// public void onAnimationRepeat(Animator arg0) {

// }

//

// @Override

// public void onAnimationEnd(Animator arg0) {

// Toast.makeText(MainActivity.this, "animator is end!",Toast.LENGTH\_SHORT).show();

// }

//

// @Override

// public void onAnimationCancel(Animator arg0) {

// }

// });

/\*\*

\* 按需添加

\*/

animator.addListener(new AnimatorListenerAdapter() {

@Override

public void onAnimationEnd(Animator animation) {

// TODO Auto-generated method stub

super.onAnimationEnd(animation);

Toast.makeText(MainActivity.this, "animator is end!",Toast.LENGTH\_SHORT).show();

}

});

animator.start();

}

**8.3.2 android帧动画**

1. **<?xml** version="1.0" encoding="utf-8"**?>**
2. <!--
3. 根标签为animation-list，其中oneshot代表着是否只展示一遍，设置为false会不停的循环播放动画
4. 根标签下，通过item标签对动画中的每一个图片进行声明
5. android:duration 表示展示所用的该图片的时间长度
6. --**>**
7. **<animation-list**
8. xmlns:android="http://schemas.android.com/apk/res/android"
9. android:oneshot="true"
10. **>**
11. **<item** android:drawable="@drawable/icon1" android:duration="150"**></item>**
12. **<item** android:drawable="@drawable/icon2" android:duration="150"**></item>**
13. **<item** android:drawable="@drawable/icon3" android:duration="150"**></item>**
14. **<item** android:drawable="@drawable/icon4" android:duration="150"**></item>**
15. **<item** android:drawable="@drawable/icon5" android:duration="150"**></item>**
16. **<item** android:drawable="@drawable/icon6" android:duration="150"**></item>**
17. **</animation-list>**
18. **private** AnimationDrawable animationDrawable;
19. animationIV.setImageResource(R.drawable.animation1);
20. animationDrawable = (AnimationDrawable) animationIV.getDrawable();
21. animationDrawable.start();

# 跨进程通信

* 1. **访问其他程序的Activity**

Activity的跨进程访问与进程内访问略有不同。虽然它们都需要Intent对象，但跨进程访问并不需要指定Context对象和Activity的 Class对象，而需要指定的是要访问的Activity所对应的Action（一个字符串）。有些Activity还需要指定一个Uri（通过 Intent构造方法的第2个参数指定）。

android:exported="true" 只要不设置成false就可以访问

第一个参数包名，第二个参数类名的全路径

注意一定要设置Act的action标签才能访问

Intent intent = **new** Intent();  
intent.setClassName(**"com.app1.yongheshen.app1"**, **"com.app1.yongheshen.app1.ActTwo"**);  
startActivity(intent);

这个常量是一个字符串常量，也是我们在这节要介绍的跨进程调用Activity的关键。如果在应用程序中要共享某个Activity，需要为这个 Activity指定一个字符串ID，也就是Action。也可以将这个Action看做这个Activity的key。在其他的应用程序中只要通过这个 Action就可以找到与Action对应的Activity，并通过startActivity方法来启动这个Activity。

下面先来看一下如何将应用程序的Activity共享出来，读者可按如下几步来共享Activity：

1. 在AndroidManifest.xml文件中指定Action。指定Action要使用<action>标签，并在该标签的android:name属性中指定Action

2. 在AndroidManifest.xml文件中指定访问协议。在指定Uri（Intent类的第2个参数）时需要访问协议。访问协议需要使 用<data>标签的android:scheme属性来指定。如果该属性的值是“abc”，那么Uri就应该是“abc://Uri的主体 部分”，也就是说，访问协议是Uri的开头部分。

3. 通过getIntent().getData().getHost()方法获得协议后的Uri的主体部分。这个Host只是个称谓，并不一定是主机名。读者可以将其看成是任意的字符串。

4. 从Bundle对象中获得其他应用程序传递过来的数据。

5. 这一步当然是获得数据后做进一步的处理了。至于如何处理这些数据，就得根据具体的需求决定了。

下面来根据这些步骤共享一个Activity。首先建立一个android工程（ActionActivity），工程的主Activity是Main。在 本例中我们会共享这个Main类。首先打开AndroidManifest.xml文件，添加一个<activity>标签，并重新定义了 Main的相应属性。AndroidManifest.xml文件的内容如下：

1. <!--  重新配置Main  -->
2. <activity android:name=".Main" android:label="@string/app\_name">
3. <intent-filter>
4. <action android:name="net.blogjava.mobile.MYACTION" />
5. <data android:scheme="info" />
6. <category android:name="android.intent.category.DEFAULT" />
7. </intent-filter>
8. </activity>

在配置AndroidManifest.xml时要注意，不能在同一个<activity>中配置多个动作，否则会覆盖MAIN动作以使该程序无法正常启动（虽然其他应用程序调用Main是正常的）。

从上面的代码可以看出，<action>标签的android:name属性值是 net.blogjava.mobile.MYACTION，这就是Main自定义的动作。<data>标签指定了Url的协议。如果指定 了<data>标签的android:scheme属性值（info），则在调用Main时需要使用如下的URL:

1. **public**  **void**  onCreate(Bundle savedInstanceState)
2. {
3. **super** .onCreate(savedInstanceState);
4. setContentView(R.layout.main);
5. Button button = (Button) findViewById(R.id.button);
6. button.setOnClickListener(**this** );
7. editText = (EditText) findViewById(R.id.edittext);
8. //  获得其他应用程序传递过来的数据
9. **if**  (getIntent().getData() != **null** )
10. {
11. //  获得Host，也就是info://后面的内容
12. String host = getIntent().getData().getHost();
13. Bundle bundle = getIntent().getExtras();
14. //  其他的应用程序会传递过来一个value值，在该应用程序中需要获得这个值
15. String value = bundle.getString("value" );
16. //  将Host和Value组合在一下显示在EditText组件中
17. editText.setText(host + ":"  + value);
18. //  调用了按钮的单击事件，显示Toast信息提示框
19. onClick(button);
20. }
21. }
    1. **Content Provider**

Android应用程序可以使用文件或SqlLite数据库来存储数据。Content Provider提供了一种在多个应用程序之间数据共享的方式（跨进程共享数据）。应用程序可以利用Content Provider完成下面的工作

1. 查询数据

2. 修改数据

3. 添加数据

4. 删除数据

虽然Content Provider也可以在同一个应用程序中被访问，但这么做并没有什么意义。Content Provider存在的目的向其他应用程序共享数据和允许其他应用程序对数据进行增、删、改操作。

Android系统本身提供了很多Content Provider，例如，音频、视频、联系人信息等等。我们可以通过这些Content Provider获得相关信息的列表。这些列表数据将以Cursor对象返回。因此，从Content Provider返回的数据是二维表的形式。

android:exported="true" 默认时无法访问，必须设置成true

## 9.3 全局广播

广播是一种被动跨进程通讯的方式。当某个程序向系统发送广播时，其他的应用程序只能被动地接收广播数据。这就象电台进行广播一样，听众只能被动地收听，而不能主动与电台进行沟通。

在应用程序中发送广播比较简单。只需要调用sendBroadcast方法即可。该方法需要一个Intent对象。通过Intent对象可以发送需要广播的数据。

先建一个android工程：sendbroadcast。在XML布局文件中放两个组件：EditText和Button，当单击按钮后，会弹出显示 EditText组件中文本的对话框，关闭对话框后， 会使用sendBroadcast方法发送消息，并将EditText组件的文本通过Intent对象发送出去。

* 1. **AIDL服务**

实现：

① Bind Service就像是C/S架构中的服务端，其他组件（比如 Activity）绑定到它（通过 bindService()），可以向它发送请求，可以接受从它返回的响应，它甚至还提供了进程间通信（IPC）功能。

② 一个service要想能够被其他组件绑定，那么它的 onBind() 方法必须被实现，且必须返回一个 IBinder 对象，然后其他组件可以通过这个 IBinder 对象与该 service 进行通讯。

③ 多个client可以绑定至同一个service，但该 service 的onBind() 方法只会在第一个 client 绑定至其的时候被调用，当其他 client 再次绑定到它的时候，并不会调用 onBind() 方法，而是直接返回第一次被调用时产生的那个 IBinder 对象。也就是说，在其生命周期内，onBind() 只会被调用一次。

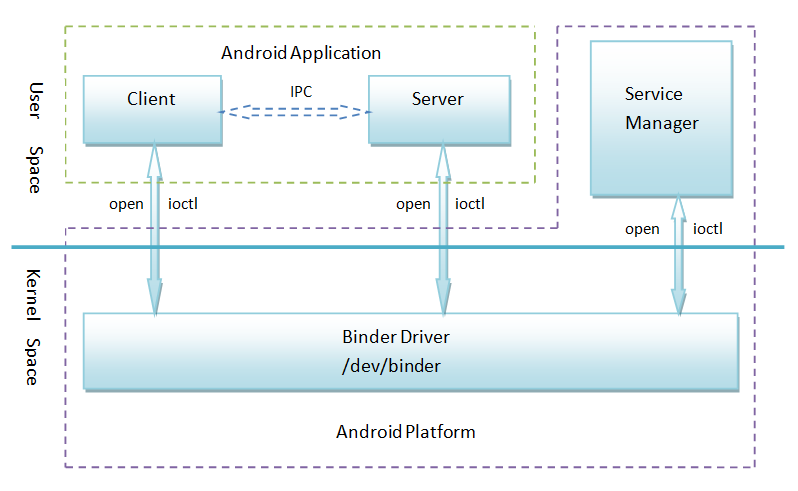
a. 在你的 service 类中声明一个内部类来继承 Binder 类。在该内部类中，最好提供一个公共方法来返回你的 service 实例。

b. 在你的 service 类中需要声明一个这个内部类的实例，以供在 onBind() 方法中返回

c. 在 client 端，在 onServiceConnected() 方法中得到从 onBind() 方法中返回的 IBinder 对象，然后可以通过该 对象中的公共方法得到相应的 service 实例，正如 第一个步骤 所说的那样。

d. 在 service 中提供公共方法， 这样就可以在组件（如 Activity 中调用这些公共方法了）

原理：



1. Client、Server和Service Manager实现在用户空间中，Binder驱动程序实现在内核空间中

2. Binder驱动程序和Service Manager在Android平台中已经实现，开发者只需要在用户空间实现自己的Client和Server

3. Binder驱动程序提供设备文件/dev/binder与用户空间交互，Client、Server和Service Manager通过open和ioctl文件操作函数与Binder驱动程序进行通信

4. Client和Server之间的进程间通信通过Binder驱动程序间接实现

5. Service Manager是一个守护进程，用来管理Server，并向Client提供查询Server接口的能力

binder通信是一种client-server的通信结构，

1.从表面上来看，是client通过获得一个server的代理接口，对server进行直接调用；

2.实际上，代理接口中定义的方法与server中定义的方法是一一对应的；

3.client调用某个代理接口中的方法时，代理接口的方法会将client传递的参数打包成为Parcel对象；

4.代理接口将该Parcel发送给内核中的binder driver.

5.server会读取binder driver中的请求数据，如果是发送给自己的，解包Parcel对象，处理并将结果返回；

6.整个的调用过程是一个同步过程，在server处理的时候，client会block住。

# Android 各个版本更新内容

版本：

| **版本** | **备注** |
| --- | --- |
| Android1.1 | 2008 年9月发布的Android第一版 |
| Android 1.5 Cupcake（纸杯蛋糕） | 2009年4月30日，官方1.5版本（Cupcake 纸杯蛋糕）的Android发布。主要的更新如下： 1、拍摄/播放影片，并支持上传到Youtube 2、支持立体声蓝牙耳机，同时改善自动配对性能 3、最新的采用WebKit技术的浏览器，支持复制/贴上和页面中搜索 4、GPS性能大大提高 5、提供屏幕虚拟键盘 6、主屏幕增加音乐播放器和相框widgets 7、应用程序自动随着手机旋转 8、短信、Gmail、日暦，浏览器的用户接口大幅改进，如Gmail可以批量删除邮件 9、相机启动速度加快，拍摄图片可以直接上传到Picasa 10、来电照片显示 |
| Android 1.6 Donut （甜甜圈） | 2009年9月15日，1.6（Donut 甜甜圈）版本软件开发工具包发布。主要的更新如下： 1、重新设计的Android Market手势  2、支持支持CDMA网络  3、文字转语音系统（Text-to-Speech）  4、快速搜索框  5、全新的拍照接口  6、查看应用程序耗电  7、支持虚拟私人网络（VPN） 8、支持更多的屏幕分辨率。 9、支持OpenCore2媒体引擎  10、新增面向视觉或听觉困难人群的易用性插件 |
| Android2.0/2.0.1/2.1 Eclair （松饼） | 2009年10月26日，2.0（Eclair 松饼）版本软件开发工具包发布。主要的更新如下： 1、优化硬件速度  2、”Car Home”程序  3、支持更多的屏幕分辨率  4、改良的用户界面  5、新的浏览器的用户接口和支持HTML5  6、新的联系人名单  7、更好的白色/黑色背景比率  8、改进Google Maps3.1.2  9、支持Microsoft Exchange  10、支持内置相机闪光灯  11、支持数码变焦  12、改进的虚拟键盘  13、支持蓝牙2.1 14、支持动态桌面的设计 |
| Android 2.2/2.2.1 Froyo （冻酸奶） | 2010年5月20日，2.2（Froyo 冻酸奶）版本软件开发工具包发布。主要的更新如下： 1、整体性能大幅度的提升  2、3G网络共享功能。  3、Flash的支持。 4、App2sd功能。 5、全新的软件商店。 6、更多的Web应用API接口的开发。 |
| Android 2.3 Gingerbread（姜饼） | 2010年12月7日，2.3（Gingerbread 姜饼）版本软件开发工具包发布。主要的更新如下： 1、增加了新的垃圾回收和优化处理事件  2、原生代码可直接存取输入和感应器事件、EGL/OpenGL ES、OpenSL ES。  3、新的管理窗口和生命周期的框架。  4、支持VP8和WebM视频格式，提供AAC和AMR宽频编码，提供了新的音频效果器  5、支持前置摄像头、SIP/VOIP和NFC（近场通讯）  6、简化界面、速度提升： 7、更快更直观的文字输入：  8、一键文字选择和复制/粘帖： 9、改进的电源管理系统：  10、新的应用管理方式： |
| Android 3.0 Honeycomb （蜂巢） | 2011年2月2日， 3.0（Honeycomb 蜂巢）版本主要更新如下：  ·优化针对平板  ·全新设计的UI增强网页浏览功能  ·n-app purchases功能 |
| Android 3.1 Honeycomb （蜂巢） | 2011年5月11日在Google I/O开发者大会宣布发布，版本主要更新如下：  Honeycomb 蜂巢（改进3.0BUG）  经过优化的Gmail电子邮箱；  全面支持GoogleMaps  将Android手机系统跟平板系统再次合并从而方便开发者。 任务管理器可滚动，支持USB 输入设备（键盘、鼠标等）。 支持 Google TV.可以支持XBOX 360无线手柄  widget支持的变化，能更加容易的定制屏幕widget插件。 |
| Android 3.2 Honeycomb （蜂巢） | 2011年7月13日发布，版本更新如下： 支持7英寸设备  引入了应用显示缩放功能 |
| Android 4.0 Ice Cream “冰激凌三明治” | 2011年10月19日 在香港发布， 4.0（Ice Cream Sandwich 冰激凌三明治）版本主要更新如下： 1、全新的ui  2、全新的Chrome Lite浏览器，有离线阅读，16标签页，隐身浏览模式等。  3、截图功能  4、更强大的图片编辑功能  5、自带照片应用堪比Instagram，可以加滤镜、加相框，进行360度全景拍摄，照片还能根据地点来排序  6、Gmail加入手势、离线搜索功能，UI更强大。 7、新功能People：以联系人照片为核心，界面偏重滑动而非点击，集成了Twitter、Linkedin、Google+等通讯工具。有望支持用户自定义添加第三方服务。 8、新增流量管理工具，可具体查看每个应用产生的流量。 9、正在运行的程序可以像电脑一样的互相切换 10、人脸识别功能  11、系统优化、速度更快  12、支持虚拟按键，手机可以不再拥有任何按键  13、更直观的程序文件夹：  14、平板电脑和智能手机通用  15、支持更大的分辨率  16、专为双核处理器编写的优化驱动  17、全新的Linux内核  18、增强的复制粘贴功能  19、语音功能  20、全新通知栏： 21、更加丰富的数据传输功能  22、更多的感应器支持  23、语音识别的键盘： 24、全新的3D驱动， 游戏支持能力提升  25、全新的谷歌电子市场 26、增强的桌面插件自定义 |
| Android 4.1 Jelly Bean（果冻豆） | 2012年6月28日发布 1.更快、更流畅、更灵敏 2.增强通知栏 3.全新搜索  4.桌面插件自动调整大小 5.加强无障碍操作 6.语言和输入法扩展 7.新的输入类型和功能 8.新的连接类型 9.新的媒体功能  10.浏览器增强 11.Google服务 |
| Android 4.2 Jelly Bean（果冻豆） | 2012年10月30日发布 1、完整的Chrome浏览器 2、全新的手机风景模式 3、全新的文件管理器 4、文本输入选项的改进 5、一个明确的升级方法 6、Android Key Lime Pie精简版  7、具有开关切换的用户界面 8、全新的电源管理系统 9、更为轻便的主题模式 10、全新的锁屏页面  11、全新的时钟界面 |
| Android 4.3 Jelly Bean（果冻豆） | 2013年7月25日发布 1、用户账户配制 2、拨号盘联系人自动补全 3、OpenGL 3.0 4、蓝牙低耗电技术 5、WIFI关闭后保持位置功能 其它特性 新的相机应用UI  新的开发者工具 通过邮件分享截屏时，日期和时间会自动加入进去。 |
| Android 4.4 KitKat（奇巧） | 2013年9月4日发布 1、优化了RenderScript计算和图像显示,取代OpenCL 2、支持两种编译模式 3、Android 4.4 KitKat针对RAM占用进行了优化，甚至可以在一些仅有512MB RAM的老款手机上流畅运行。 4、新图标、锁屏、启动动画和配色方案 5、新的拨号和智能来电显示 6、加强主动式语音功能  7、集成Hangouts IM软件  8、全屏模式 9、支持Emoji键盘 10、轻松访问在线存储  11、无线打印  12、屏幕录像功能 13、内置字幕管理功能  14、计步器应用  15、低功耗音频和定位模式  16、新的接触式支付系统  17、新的蓝牙配置文件和红外兼容性 |
| Android 5.0 Lollipop(棒棒糖) | 2014年6月26日发布 1、谷歌将为Android的语音服务Google Now加入一个名为OK Google Everywhere的全新功能。 2、Android 5.0可能还会加入更多的健身功能，考虑到谷歌在发布了Android Wear，后者与智能手表及谷歌眼镜等可穿戴设备的协作应该会成为下个版本的重点功能。 3、整合碎片化  4、传言Google将在Android5.0中，禁止厂商进行深度定制。 5、数据迁移  6、独立平板  7、功能按键  8、接口风格 |
| Android 6.0 Marshmallow(棉花糖) | 10月6日发布 1.App Permissions（软件权限管理）。  2.Chrome Custom Tabs（网页体验提升）。 3.App Links（APP关联）。  4.Android Pay（安卓支付）。 5.Fingerprint Support（指纹支持）。  6.Power & Change（电量管理 ）。 |

# 注解

在Java中，注解(Annotation)引入始于Java5，用来描述Java代码的元信息，通常情况下注解不会直接影响代码的执行，尽管有些注解可以用来做到影响代码执行。

Annotation是一种应用于类、方法、参数、变量、构造器及包声明中的特殊修饰符。它是一种由JSR-175标准选择用来描述元数据的一种工具。

原理：根据反射得到字段，然后判断这个字段上面是否标识了这个注解,如果标识了，就如何如何。。。。

\*注解的使用：

1 自定义注解并添加到对应的位置。

2 解析注解并处理-，通过被添加类反射获取到所有的属性和方法，之后判断属性或者方法或者类是否添加了某个注解，要是添加了，

通过注解名称获取到对应的注解属性，之后将注解中的值取出来，处理。

**注解可以做什么**

Java中的注解通常扮演以下角色

编译器指令

构建时指令

运行时指令

Java内置了三种编译器指令，本文后面部分会重点介绍

Java注解可以应用在构建时，即当你构建你的项目时。构建过程包括生成源码，编译源码，生成xml文件，打包编译的源码和文件到JAR包等。软件的构建通常使用诸如Apache Ant和Maven这种工具自动完成。这些构建工具会依照特定的注解扫描Java代码，然后根据这些注解生成源码或文件。

通常情况下，注解并不会出现在编译后的代码中，但是如果想要出现也是可以的。Java支持运行时的注解，使用Java的反射我们可以访问到这些注解，运行时的注解的目的通常是提供给程序和第三方API一些指令。

**元注解：**

　　元注解的作用就是负责注解其他注解。Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。Java5.0定义的元注解：

　　　　1.@Target,

　　　　2.@Retention,

　　　　3.@Documented,

　　　　4.@Inherited

这些类型和它们所支持的类在java.lang.annotation包中可以找到。下面我们看一下每个元注解的作用和相应分参数的使用说明。

**@Target：**

　　　@Target说明了Annotation所修饰的对象范围：Annotation可被用于 packages、types（类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数）。在Annotation类型的声明中使用了target可更加明晰其修饰的目标。

　　作用：用于描述注解的使用范围（即：被描述的注解可以用在什么地方）

　　取值(ElementType)有：

　　　　1.CONSTRUCTOR:用于描述构造器

　　　　2.FIELD:用于描述域

　　　　3.LOCAL\_VARIABLE:用于描述局部变量

　　　　4.METHOD:用于描述方法

　　　　5.PACKAGE:用于描述包

　　　　6.PARAMETER:用于描述参数

　　　　7.TYPE:用于描述类、接口(包括注解类型) 或enum声明

使用实例：

@Target(ElementType.***TYPE***)  
 **public** @**interface** Table {  
 */\*\*  
 \* 数据表名称注解，默认值为类名称  
 \** ***@return*** *\*/* **public** String tableName() **default "className"**;  
 }  
 @Target(ElementType.***FIELD***)  
 **public** @**interface** NoDBColumn {  
  
 }

注解Table 可以用于注解类、接口(包括注解类型) 或enum声明,而注解NoDBColumn仅可用于注解类的成员变量。

**@Retention：**

@Retention定义了该Annotation被保留的时间长短：某些Annotation仅出现在源代码中，而被编译器丢弃；而另一些却被编译在class文件中；编译在class文件中的Annotation可能会被虚拟机忽略，而另一些在class被装载时将被读取（请注意并不影响class的执行，因为Annotation与class在使用上是被分离的）。使用这个meta-Annotation可以对 Annotation的“生命周期”限制。

作用：表示需要在什么级别保存该注释信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效）

取值（RetentionPoicy）有：

　　　　1.SOURCE:在源文件中有效（即源文件保留）

　　　　2.CLASS:在class文件中有效（即class保留）

　　　　3.RUNTIME:在运行时有效（即运行时保留）

Retention meta-annotation类型有唯一的value作为成员，它的取值来自java.lang.annotation.RetentionPolicy的枚举类型值。具体实例如下：

@Target(ElementType.***FIELD***)  
@Retention(RetentionPolicy.***RUNTIME***)  
**public** @**interface** Column {  
 **public** String name() **default "fieldName"**;  
 **public** String setFuncName() **default "setField"**;  
 **public** String getFuncName() **default "getField"**;  
 **public boolean** defaultDBValue() **default false**;  
}

Column注解的的RetentionPolicy的属性值是RUTIME,这样注解处理器可以通过反射，获取到该注解的属性值，从而去做一些运行时的逻辑处理

**@Documented:**

@Documented用于描述其它类型的annotation应该被作为被标注的程序成员的公共API，因此可以被例如javadoc此类的工具文档化。Documented是一个标记注解，没有成员。

@Target(ElementType.***FIELD***)  
@Retention(RetentionPolicy.***RUNTIME***)  
@Documented  
**public** @**interface** Column {  
 **public** String name() **default "fieldName"**;  
 **public** String setFuncName() **default "setField"**;  
 **public** String getFuncName() **default "getField"**;  
 **public boolean** defaultDBValue() **default false**;  
}

**@Inherited：**

　　@Inherited 元注解是一个标记注解，@Inherited阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited修饰的annotation类型被用于一个class，则这个annotation将被用于该class的子类。

　　注意：@Inherited annotation类型是被标注过的class的子类所继承。类并不从它所实现的接口继承annotation，方法并不从它所重载的方法继承annotation。

　　当@Inherited annotation类型标注的annotation的Retention是RetentionPolicy.RUNTIME，则反射API增强了这种继承性。如果我们使用java.lang.reflect去查询一个@Inherited annotation类型的annotation时，反射代码检查将展开工作：检查class和其父类，直到发现指定的annotation类型被发现，或者到达类继承结构的顶层。

实例代码：

@Inherited  
**public** @**interface** Greeting {  
 **public enum** FontColor{ ***BULE***,***RED***,***GREEN***};  
 String name();  
 FontColor fontColor() **default** FontColor.***GREEN***;  
}

# MVC MVP

**MVC模式**

MVC模式的结构分为三部分，实体层的Model，视图层的View，以及控制层的Controller。

其中M层处理数据，业务逻辑等；V层处理界面的显示结果；C层起到桥梁的作用，来控制V层和M层通信以此来达到分离视图显示和业务逻辑层。

**Android中界面部分也采用了当前比较流行的MVC框架，在Android中：**

**视图层(View)**

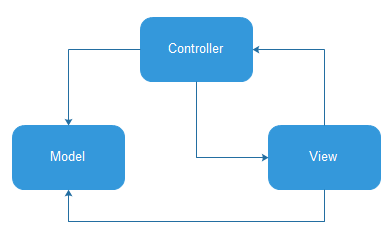
一般采用XML文件进行界面的描述，这些XML可以理解为AndroidApp的View。使用的时候可以非常方便的引入。同时便于后期界面的修改。逻辑中与界面对应的id不变化则代码不用修改，大大增强了代码的可维护性。

**控制层(Controller)**

Android的控制层的重任通常落在了众多的Activity的肩上。这句话也就暗含了不要在Activity中写代码，要通过Activity交割Model业务逻辑层处理，这样做的另外一个原因是Android中的Actiivity的响应时间是5s，如果耗时的操作放在这里，程序就很容易被回收掉。

**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（感谢@Xander的讲解）。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的该层的。就是应用程序中二进制的数据。



**MVP模式**

View:负责绘制UI元素、与用户进行交互(在Android中体现为Activity)

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)

Presenter:作为View与Model交互的中间纽带，处理与用户交互的负责逻辑。

\*View interface:需要View实现的接口，View通过View interface与Presenter进行交互，降低耦合，方便进行单元测试

**MVP流程**



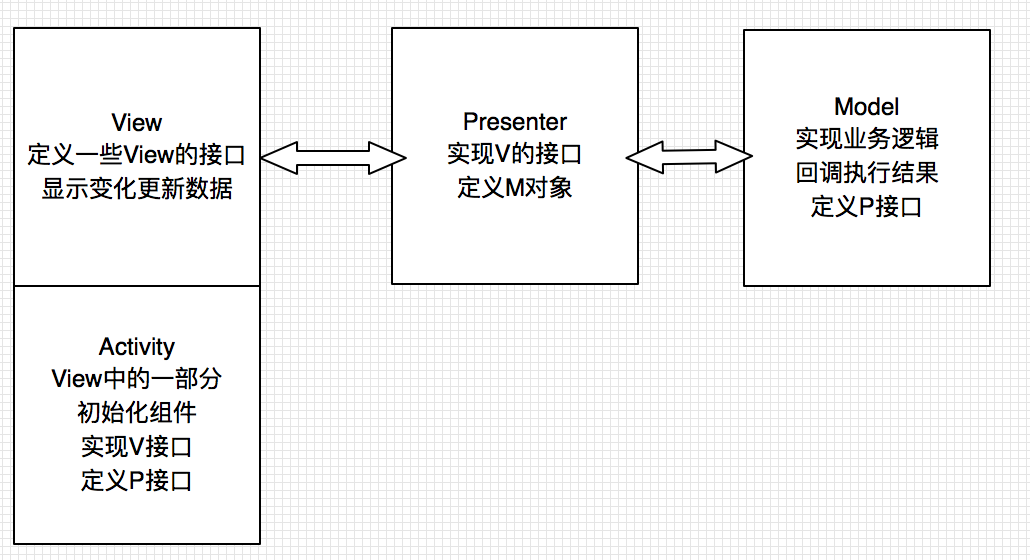
说明：

步骤1：UI实现View方法，引用Presenter

步骤2：Presenter调用Model，走Model具体逻辑

步骤3：Model逻辑实现，回调Presenter方法

步骤4：Presenter回调View，即回到UI，回调View方法



**MVP模式的核心思想：**

MVP把Activity中的UI逻辑抽象成View接口，把业务逻辑抽象成Presenter接口，Model类还是原来的Model。

这就是MVP模式，现在这样的话，Activity的工作的简单了，只用来响应生命周期，其他工作都丢到Presenter中去完成。从上图可以看出，Presenter是Model和View之间的桥梁，为了让结构变得更加简单，View并不能直接对Model进行操作，这也是MVP与MVC最大的不同之处。

**MVP的优势**

1.降低耦合度，隐藏数据，Activity中代码更简洁

2.模块职责划分明显

3.方便测试驱动开发

4.代码复用度较高

5.代码灵活性

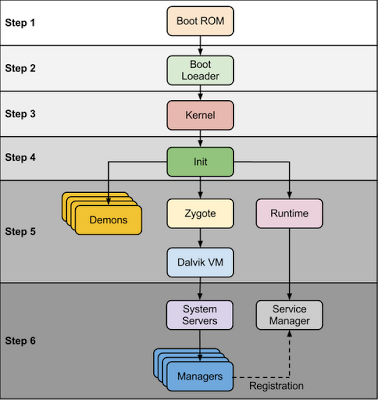
**MVP的问题**

由于对视图的渲染放在了Presenter中，所以视图和Persenter的交互会过于频繁。

还有一点你需要明白，如果Presenter过多地渲染了视图，往往会使得它与特定的视图的联系过于紧密。一旦视图需要变更，那么Presenter也需要变更了。比如说，原本用来呈现Html的Presenter现在也需要用于呈现Pdf了，那么视图很有可能也需要变更。

# Android 启动过程

当你按下电源开关后Android设备执行了以下步骤。



### 第一步：启动电源以及系统启动

当电源按下，引导芯片代码开始从预定义的地方（固化在ROM）开始执行。加载引导程序到RAM，然后执行。

### 第二步：引导程序

引导程序是在Android操作系统开始运行前的一个小程序。引导程序是运行的第一个程序，因此它是针对特定的主板与芯片的。设备制造商要么使用很受欢迎的引导程序比如[redboot](http://ecos.sourceware.org/redboot/)、[uboot](http://www.denx.de/wiki/U-Boot)、[qi bootloader](http://wiki.openmoko.org/wiki/Qi)或者开发自己的引导程序，它不是Android操作系统的一部分。引导程序是OEM厂商或者运营商加锁和限制的地方。

引导程序分两个阶段执行。第一个阶段，检测外部的RAM以及加载对第二阶段有用的程序；第二阶段，引导程序设置网络、内存等等。这些对于运行内核是必要的，为了达到特殊的目标，引导程序可以根据配置参数或者输入数据设置内核。

Android引导程序可以在\bootable\bootloader\legacy\usbloader找到。  
传统的加载器包含的个文件，需要在这里说明：

1. init.s初始化堆栈，清零BBS段，调用main.c的\_main()函数；
2. main.c初始化硬件（闹钟、主板、键盘、控制台），创建linux标签。

更多关于Android引导程序的可以在[这里](https://motorola-global-portal.custhelp.com/app/answers/detail/a_id/86208/~/bootloader-frequently-asked-questions)了解。

### 第三步：内核

Android内核与桌面linux内核启动的方式差不多。内核启动时，设置缓存、被保护存储器、计划列表，加载驱动。当内核完成系统设置，它首先在系统文件中寻找”init”文件，然后启动root进程或者系统的第一个进程。

### 第四步：init进程

init是第一个进程，我们可以说它是root进程或者说有进程的父进程。init进程有两个责任，一是挂载目录，比如/sys、/dev、/proc，二是运行init.rc脚本。

* init进程可以在/system/core/init找到。
* init.rc文件可以在/system/core/rootdir/init.rc找到。
* readme.txt可以在/system/core/init/readme.txt找到。

对于init.rc文件，Android中有特定的格式以及规则。在Android中，我们叫做Android初始化语言。  
Android初始化语言由四大类型的声明组成，即Actions（动作）、Commands（命令）、Services（服务）、以及Options（选项）。  
Action（动作）：动作是以命令流程命名的，有一个触发器决定动作是否发生。  
语法

|  |  |
| --- | --- |
| 1  2  3  4  5 | ; html-script: false ]      on &lt;trigger&gt;          &lt;command&gt;          &lt;command&gt;          &lt;command&gt; |

Service（服务）：服务是init进程启动的程序、当服务退出时init进程会视情况重启服务。  
语法

; html-script: false ]

service &lt;name&gt; &lt;pathname&gt; [&lt;argument&gt;]\*

    &lt;option&gt;

    &lt;option&gt;

    ...

Options（选项）  
选项是对服务的描述。它们影响init进程如何以及何时启动服务。  
咱们来看看默认的init.rc文件。这里我只列出了主要的事件以及服务。  
Table

|  |  |
| --- | --- |
| Action/Service | 描述 |
| on early-init | 设置init进程以及它创建的子进程的优先级，设置init进程的安全环境 |
| on init | 设置全局环境，为cpu accounting创建cgroup(资源控制)挂载点 |
| on fs | 挂载mtd分区 |
| on post-fs | 改变系统目录的访问权限 |
| on post-fs-data | 改变/data目录以及它的子目录的访问权限 |
| on boot | 基本网络的初始化，内存管理等等 |
| service servicemanager | 启动系统管理器管理所有的本地服务，比如位置、音频、Shared preference等等… |
| service zygote | 启动zygote作为应用进程 |

在这个阶段你可以在设备的屏幕上看到“Android”logo了。

### 第五步

在Java中，我们知道不同的虚拟机实例会为不同的应用分配不同的内存。假如Android应用应该尽可能快地启动，但如果Android系统为每一个应用启动不同的Dalvik虚拟机实例，就会消耗大量的内存以及时间。因此，为了克服这个问题，Android系统创造了”Zygote”。Zygote让Dalvik虚拟机共享代码、低内存占用以及最小的启动时间成为可能。Zygote是一个虚拟器进程，正如我们在前一个步骤所说的在系统引导的时候启动。Zygote预加载以及初始化核心库类。通常，这些核心类一般是只读的，也是Android SDK或者核心框架的一部分。在Java虚拟机中，每一个实例都有它自己的核心库类文件和堆对象的拷贝。

Zygote加载进程

1. 加载[ZygoteInit](http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/2.2_r1.1/com/android/internal/os/ZygoteInit.java)类，源代码：/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
2. registerZygoteSocket()为zygote命令连接注册一个服务器套接字。
3. preloadClassed “preloaded-classes”是一个简单的包含一系列需要预加载类的文本文件，你可以在<Android Source>/frameworks/base找到“preloaded-classes”文件。
4. preloadResources() preloadResources也意味着本地主题、布局以及android.R文件中包含的所有东西都会用这个方法加载。

在这个阶段，你可以看到启动动画。

### 第六步：系统服务或服务

完成了上面几步之后，运行环境请求Zygote运行系统服务。系统服务同时使用native以及java编写，系统服务可以认为是一个进程。同一个系统服务在Android SDK可以以System Services形式获得。系统服务包含了所有的System Services。

Zygote创建新的进程去启动系统服务。你可以在ZygoteInit类的”startSystemServer”方法中找到源代码。

核心服务：

1. 启动电源管理器；
2. 创建Activity管理器；
3. 启动电话注册；
4. 启动包管理器；
5. 设置Activity管理服务为系统进程；
6. 启动上下文管理器；
7. 启动系统Context Providers；
8. 启动电池服务；
9. 启动定时管理器；
10. 启动传感服务；
11. 启动窗口管理器；
12. 启动蓝牙服务；
13. 启动挂载服务。

其他服务：

1. 启动状态栏服务；
2. 启动硬件服务；
3. 启动网络状态服务；
4. 启动网络连接服务；
5. 启动通知管理器；
6. 启动设备存储监视服务；
7. 启动定位管理器；
8. 启动搜索服务；
9. 启动剪切板服务；
10. 启动登记服务；
11. 启动壁纸服务；
12. 启动音频服务；
13. 启动耳机监听；
14. 启动AdbSettingsObserver（处理adb命令）。

### 第七步：引导完成

一旦系统服务在内存中跑起来了，Android就完成了引导过程。在这个时候“ACTION\_BOOT\_COMPLETED”开机启动广播就会发出去。