[**http://click.aliyun.com/m/42909/**](http://click.aliyun.com/m/42909/)

# Android 资源文件命名与使用

1. 【推荐】资源文件需带模块前缀。

2. 【推荐】layout 文件的命名方式。

Activity 的layout 以module\_activity 开头

Fragment 的layout 以module\_fragment 开头

Dialog 的layout 以module\_dialog 开头

include 的layout 以module\_include 开头

ListView 的行layout 以module\_list\_item 开头

RecyclerView 的item layout 以module\_recycle\_item 开头

GridView 的item layout 以module\_grid\_item 开头

3. 【推荐】drawable 资源名称以小写单词+下划线的方式命名，根据分辨率不同存放在

不同的drawable 目录下，如果介意包大小建议只使用一套，系统去进行缩放。采用

规则如下：

模块名\_业务功能描述\_控件描述\_控件状态限定词

如：module\_login\_btn\_pressed,module\_tabs\_icon\_home\_normal

4. 【推荐】anim 资源名称以小写单词+下划线的方式命名，采用以下规则：

模块名\_逻辑名称\_[方向|序号]

Tween 动画（使用简单图像变换的动画，例如缩放、平移）资源：尽可能以通用的

动画名称命名，如module\_fade\_in , module\_fade\_out , module\_push\_down\_in (动

画+方向)。

Frame 动画（按帧顺序播放图像的动画）资源：尽可能以模块+功能命名+序号。如

module\_loading\_grey\_001。

5. 【推荐】color 资源使用#AARRGGBB 格式，写入module\_colors.xml 文件中，命名

格式采用以下规则：

模块名\_逻辑名称\_颜色

**如：**

**<color name="module\_btn\_bg\_color">#33b5e5e5</color>**

**6. 【推荐】dimen 资源以小写单词+下划线方式命名，写入module\_dimens.xml 文件中，**

**采用以下规则：**

模块名\_描述信息

如：

<dimen name="module\_horizontal\_line\_height">1dp</dimen>

7.【推荐】style 资源采用“ 父style 名称.当前style 名称”方式命名，写入

module\_styles.xml 文件中，首字母大写。如：

<style name="ParentTheme.ThisActivityTheme">

…

</style>

8. 【推荐】string资源文件或者文本用到字符需要全部写入module\_strings.xml 文件中，

字符串以小写单词+下划线的方式命名，采用以下规则：

模块名\_逻辑名称

如：moudule\_login\_tips,module\_homepage\_notice\_desc

9. 【推荐】Id 资源原则上以驼峰法命名，View 组件的资源id 建议以View 的缩写作为

前缀。常用缩写表如下：

控件 缩写

LinearLayout ll

RelativeLayout rl

ConstraintLayout cl

ListView lv

ScollView sv

TextView tv

Button btn

ImageView iv

CheckBox cb

RadioButton rb

EditText et

其它控件的缩写推荐使用小写字母并用下划线进行分割，例如：ProgressBar 对应

的缩写为progress\_bar；DatePicker 对应的缩写为date\_picker。

10.【推荐】图片根据其分辨率，放在不同屏幕密度的drawable 目录下管理，否则可能

在低密度设备上导致内存占用增加，又可能在高密度设备上导致图片显示不够清晰。

说明：

ldpi - 120dpi

mdpi - 160dpi

hdpi - 240dpi

xhdpi - 320dpi

xxhdpi - 480dpi

xxxhdpi - 640dpi

Android 的屏幕分辨率和密度并不存在严格的对应关系，应尽量避免直接基于分辨

率来开发，而是通过适配不同的屏幕密度来保证控件和图片的显示效果。不同密度

drawable 目录中的图片分辨率设置，参考不同密度的dpi 比例关系。

正例：

为显示某个图标，将48 x 48 的图标文件放在drawable-mdpi 目录（160dpi）下；

将72 x 72 的图标文件放在drawable-hdpi 目录（240dpi）下；将96 x 96 的图标

文件放在drawable-xhdpi 目录（320dpi）下；将144 x 144 的图标文件放在

drawable-xxhdpi 目录（480dpi）下。

反例：

上述图标，只有一个144 x 144 的图标文件放在drawable 目录下。

# Android 基本组件

Android 基本组件指Activity 、Fragment 、Service 、BroadcastReceiver 、

ContentProvider 等等。

1. 【强制】Activity 间的数据通信，对于数据量比较大的，避免使用Intent + Parcelable

的方式，可以考虑EventBus 等替代方案，以免造成TransactionTooLargeException。

2. 【推荐】Activity#onSaveInstanceState()方法不是Activity 生命周期方法，也不保证

一定会被调用。它是用来在Activity 被意外销毁时保存UI 状态的，只能用于保存临

时性数据，例如UI 控件的属性等，不能跟数据的持久化存储混为一谈。持久化存储

应该在Activity#onPause()/onStop()中实行。

3. 【强制】Activity 间通过隐式Intent 的跳转，在发出Intent 之前必须通过resolveActivity

检查，避免找不到合适的调用组件，造成ActivityNotFoundException 的异常。

正例：

正例：

public void viewUrl(String url, String mimeType) {

Intent intent = new Intent(Intent.ACTION\_VIEW);

intent.setDataAndType(Uri.parse(url), mimeType);

if (getPackageManager().resolveActivity(intent, PackageManager.MATCH\_DEFAULT\_

ONLY) != null) {

startActivity(intent);

}else {

// 找不到指定的Activity

}

}

反例：

Intent intent = new Intent();

intent.setAction("com.example.DemoIntent ");

try {

startActivity(intent);

} catch (ActivityNotFoundException e) {

e.printStackTrace();

}

4. 【强制】避免在Service#onStartCommand()/onBind()方法中执行耗时操作，如果确

实有需求，应改用IntentService 或采用其他异步机制完成。

正例：

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **class** MainActivity **extends** Activity {
2. @Override
3. **public** **void** onCreate(Bundle savedInstanceState) {
4. **super**.onCreate(savedInstanceState);
5. setContentView(R.layout.main);
6. }
8. **public** **void** startIntentService(View source) {
9. Intent intent = **new** Intent(**this**, MyIntentService.**class**);
10. startService(intent);
11. }
12. }
14. **public** **class** MyIntentService **extends** IntentService {
15. **public** MyIntentService() {
16. **super**("MyIntentService");
17. }
19. @Override
20. **protected** **void** onHandleIntent(Intent intent) {
21. **synchronized** (**this**) {
22. **try** {
23. ......
24. } **catch** (Exception e) {
25. }
26. }
27. }
28. }

5. 【强制】避免在BroadcastReceiver#onReceive()中执行耗时操作，如果有耗时工作，

应该创建IntentService 完成，而不应该在BroadcastReceiver 内创建子线程去做

说明：

由于该方法是在主线程执行，如果执行耗时操作会导致UI 不流畅。可以使用

IntentService 、创建HandlerThread 或者调用Context#registerReceiver

(BroadcastReceiver, IntentFilter, String, Handler)方法等方式，在其他Wroker 线程

执行onReceive 方法。BroadcastReceiver#onReceive()方法耗时超过10 秒钟，可

能会被系统杀死。

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. IntentFilter filter=**new** IntentFilter();
2. filter.addAction(LOGIN\_SUCCESS);
3. **this**.registerReceiver(mBroadcastReceiver,filter);
4. mBroadcastReceiver=**new** BroadcastReceiver(){
5. @Override
6. **public** **void** onReceive(Context context,Intent intent){
7. Intent userHomeIntent=**new** Intent();
8. userHomeIntent.setClass(**this**,UserHomeService.**class**);
9. **this**.startService(userHomeIntent);
10. }
11. };

反例：

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. mBroadcastReceiver=**new** BroadcastReceiver(){
2. @Override
3. **public** **void** onReceive(Context context,Intent intent){
4. @Override
5. **public** **void** onReceive(Context context,Intent intent){
6. MyDatabaseHelper myDB=**new** MyDatabaseHelper(context);
7. myDB.initData();
8. // have more database operation here
9. }
10. }
11. };

6. 【强制】避免使用隐式Intent 广播敏感信息，信息可能被其他注册了对应

BroadcastReceiver 的App 接收。

说明：

通过Context#sendBroadcast()发送的隐式广播会被所有感兴趣的receiver 接收，恶

意应用注册监听该广播的receiver 可能会获取到Intent 中传递的敏感信息，并进行

其他危险操作。如果发送的广播为使用Context#sendOrderedBroadcast()方法发送

的有序广播，优先级较高的恶意receiver 可能直接丢弃该广播，造成服务不可用，

或者向广播结果塞入恶意数据。

如果广播仅限于应用内，则可以使用LocalBroadcastManager#**sendBroadcast**()实

现，避免敏感信息外泄和Intent 拦截的风险。

正例：

Intent intent = new Intent("my-sensitive-event");

intent.putExtra("event", "this is a test event");

LocalBroadcastManager.getInstance(this).sendBroadcast(intent);

反例：

Intent intent = new Intent();

v1.setAction("com.sample.action.server\_running");

v1.putExtra("local\_ip", v0.h);

v1.putExtra("port", v0.i);

v1.putExtra("code", v0.g);

v1.putExtra("connected", v0.s);

v1.putExtra("pwd\_predefined", v0.r);

if (!TextUtils.isEmpty(v0.t)) {

v1.putExtra("connected\_usr", v0.t);

}

context.sendBroadcast(v1);

以上广播可能被其他应用的如下receiver 接收导致敏感信息泄漏

final class MyReceiver extends BroadcastReceiver {

public final void onReceive(Context context, Intent intent) {

if (intent != null && intent.getAction() != null) {

String s = intent.getAction();

if (s.equals("com.sample.action.server\_running") {

String ip = intent.getStringExtra("local\_ip");

String pwd = intent.getStringExtra("code");

String port = intent.getIntExtra("port", 8888);

boolean status = intent.getBooleanExtra("connected", false);

}

}

}

}

7. 【推荐】添加Fragment 时， 确保FragmentTransaction#commit() 在

ActivityActivity#onPostResume()或者FragmentActivity#onResumeFragments()内调用。

不要随意使用FragmentTransaction#commitAllowingStateLoss() 来代替，任何

commitAllowingStateLoss()的使用必须经过code review，确保无负面影响。

说明：

Activity 可能因为各种原因被销毁， Android 支持页面被销毁前通过

Activity#onSaveInstanceState() 保存自己的状态。但如果

FragmentTransaction.commit()发生在Activity 状态保存之后，就会导致Activity 重

建、恢复状态时无法还原页面状态，从而可能出错。为了避免给用户造成不好的体验，系统会抛出IllegalStateExceptionStateLoss 异常。推荐的做法是在Activity 的

onPostResume() 或onResumeFragments() （ 对FragmentActivity ） 里执行

FragmentTransaction.commit()，如有必要也可在onCreate()里执行。不要随意改用

FragmentTransaction.commitAllowingStateLoss() 或者直接使用try-catch 避免

crash，这不是问题的根本解决之道，当且仅当你确认Activity 重建、恢复状态时，

本次commit 丢失不会造成影响时才可这么做。

正例：

。。。

8. 【推荐】不要在Activity#onDestroy()内执行释放资源的工作，例如一些工作线程的

销毁和停止，因为onDestroy()执行的时机可能较晚。可根据实际需要，在

Activity#onPause()/onStop()中结合isFinishing()的判断来执行。

9. 【推荐】如非必须，避免使用嵌套的Fragment。

说明：

嵌套Fragment 是在Android API 17添加到SDK以及Support 库中的功能，Fragment

嵌套使用会有一些坑，容易出现bug，比较常见的问题有如下几种：

1) onActivityResult()方法的处理错乱，内嵌的Fragment 可能收不到该方法的回调，

需要由宿主Fragment 进行转发处理；

2) 突变动画效果；

3) 被继承的setRetainInstance()，导致在Fragment 重建时多次触发不必要的逻

辑。

非必须的场景尽可能避免使用嵌套Fragment，如需使用请注意上述问题。

正例：

FragmentManager fragmentManager = getFragmentManager();

Fragment fragment = fragmentManager.findFragmentByTag(FragmentB.TAG);

if (null == fragment) {

FragmentB fragmentB = new FragmentB();

FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

fragmentTransaction.add(R.id.fragment\_container, fragmentB, FragmentB.TAG).

commit();

}

反例：

Fragment videoFragment = new VideoPlayerFragment();

FragmentTransaction transaction = currentFragment.getChildFragmentManager().

beginTransaction();

transaction.add(R.id.video\_fragment, videoFragment).commit();

10. 【推荐】总是使用显式Intent 启动或者绑定Service，且不要为服务声明Intent Filter，

保证应用的安全性。如果确实需要使用隐式调用，则可为Service 提供Intent Filter

并从Intent 中排除相应的组件名称，但必须搭配使用Intent#setPackage()方法设置

Intent 的指定包名，这样可以充分消除目标服务的不确定性。

11.【推荐】Service 需要以多线程来并发处理多个启动请求，建议使用IntentService，

可避免各种复杂的设置。

说明：

Service 组件一般运行主线程，应当避免耗时操作，如果有耗时操作应该在Worker

线程执行。可以使用IntentService 执行后台任务。

正例：

12.【推荐】对于只用于应用内的广播，优先使用LocalBroadcastManager 来进行注册

和发送，LocalBroadcastManager 安全性更好，同时拥有更高的运行效率。

说明：

对于使用Context#sendBroadcast()等方法发送全局广播的代码进行提示。如果该广

播仅用于应用内，则可以使用LocalBroadcastManager 来避免广播泄漏以及广播被

拦截等安全问题，同时相对全局广播本地广播的更高效。

13.【推荐】当前Activity 的onPause 方法执行结束后才会创建（onCreate）或恢复

（onRestart）别的Activity，所以在onPause 方法中不适合做耗时较长的工作，这

会影响到页面之间的跳转效率。

14. 【强制】Activity 或者Fragment 中动态注册BroadCastReceiver 时，registerReceiver()

和unregisterReceiver()要成对出现。

说明：

如果registerReceiver()和unregisterReceiver()不成对出现，则可能导致已经注册的

receiver 没有在合适的时机注销，导致内存泄漏，占用内存空间，加重SystemService

负担。

部分华为的机型会对receiver 进行资源管控，单个应用注册过多receiver 会触发管

控模块抛出异常，应用直接崩溃。

15.【强制】Android 基础组件如果使用隐式调用，应在 AndroidManifest.xml 中使用

<intent-filter> 或在代码中使用 IntentFilter 增加过滤。

说明：

如果浏览器支持Intent Scheme Uri 语法，如果过滤不当，那么恶意用户可能通过浏

览器js 代码进行一些恶意行为，比如盗取cookie 等。如果使用了Intent.parseUri

函数，获取的intent 必须严格过滤。

正例：

正例：

// 将intent scheme URL 转换为intent 对象

Intent intent = Intent.parseUri(uri);

// 禁止没有BROWSABLE category 的情况下启动activity

intent.addCategory("android.intent.category.BROWSABLE");

intent.setComponent(null);

intent.setSelector(null);

// 使用intent 启动activity

context.startActivityIfNeeded(intent, -1)

反例：

Intent intent = Intent.parseUri(uri.toString().trim().substring(15), 0);

intent.addCategory("android.intent.category.BROWSABLE");

context.startActivity(intent);

# 、UI 与布局

## 不允许LinearLayout嵌套

【强制】布局中不得不使用ViewGroup 多重嵌套时，不要使用LinearLayout 嵌套，

改用RelativeLayout，可以有效降低嵌套数。

说明：

Android 应用页面上任何一个View 都需要经过 measure、layout、draw 三个步骤

才能被正确的渲染。从xml layout 的顶部节点开始进行measure，每个子节点都需

要向自己的父节点提供自己的尺寸来决定展示的位置，在此过程中可能还会重新

measure（由此可能导致measure 的时间消耗为原来的2-3 倍）。节点所处位置越深，

嵌套带来的measure 越多，计算就会越费时。这就是为什么扁平的View 结构会性

能更好。

同时，页面拥上的View 越多，measure、layout、draw 所花费的时间就越久。要缩

短这个时间，关键是保持View 的树形结构尽量扁平，而且要移除所有不需要渲染的

View。理想情况下，总共的measure，layout，draw 时间应该被很好的控制在16ms

以内，以保证滑动屏幕时UI 的流畅。

要找到那些多余的View（增加渲染延迟的view），可以用Android Studio Monitor

里的Hierarchy Viewer 工具，可视化的查看所有的view。

## DialogFragment替代Dialog【【推荐】】

在Activity 中显示对话框或弹出浮层时，尽量使用DialogFragment，而非

Dialog/AlertDialog，这样便于随Activity生命周期管理对话框/弹出浮层的生命周期。

### 正例：

G

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** showPromptDialog(String text) {
2. DialogFragment promptDialog = **new** DialogFragment() {
3. @Override
4. **public** View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
5. savedInstanceState) {
6. getDialog().requestWindowFeature(Window.FEATURE\_NO\_TITLE);
7. View view = inflater.inflate(R.layout.dji\_launcher\_qs\_hdmi\_rs\_ly, container);
8. **return** view;
9. }
10. };
11. promptDialog.show(((Activity)getContext()).getFragmentManager(), text);
12. }

## 【推荐】UTF-8

源文件统一采用UTF-8 的形式进行编码。

## 【强制】禁止在非UI 线程进行View 相关操作

## dp

【推荐】文本大小使用单位dp，View 大小使用单位dp。对于TextView，如果在文

字大小确定的情况下推荐使用wrap\_content 布局避免出现文字显示不全的适配问

题。

说明：

之所以文本大小也推荐使用dp 而非sp，因为sp 是Android 早期推荐使用的，但其

实sp 不仅和dp 一样受屏幕密度的影响，还受到系统设置里字体大小的影响，所以

使用dp 对于应用开发会更加保证UI 的一致性和还原度。

## 【强制】不允许过度绘制

【强制】禁止在设计布局时多次为子View 和父View 设置同样背景进而造成页面过

度绘制，推荐将不需要显示的布局进行及时隐藏。

正例：

## 【推荐】灵活使用布局

【推荐】灵活使用布局，推荐merge、ViewStub 来优化布局，尽可能多的减少UI

布局层级，推荐使用FrameLayout，LinearLayout、RelativeLayout 次之。

## 【推荐ROI

在需要时刻刷新某一区域的组件时，建议通过以下方式避免引发全局layout

刷新:

1) 设置固定的View 大小的宽高，如倒计时组件等；

2) 调用View 的layout 方法修改位置，如弹幕组件等；

3) 通过修改Canvas 位置并且调用invalidate(int l, int t, int r, int b)等方式限定刷新

区域；

4) 通过设置一个是否允许requestLayout 的变量，然后重写控件的requestlayout、

onSizeChanged 方法， 判断控件的大小没有改变的情况下， 当进入

requestLayout 的时候，直接返回而不调用super 的requestLayout 方法。

## 【推荐】onWindowFocusChanged() 之后）才创建对话框。

不能在Activity 没有完全显示时显示PopupWindow 和Dialog。

说明：

Android Activity 创建时的生命周期，按照 onCreate() -> onStart() -> onResume() ->

onAttachedToWindow() -> onWindowFocusChanged() 的顺序， 其中在

Activity#onAttachedToWindow() 时，Activity 会与它的 Window 关联，这时 UI 才

会开始绘制，在 Activity#onWindowFocusChanged() 时，UI 才变成可交互状态，

可以提示用户使用。如果在 Window 未关联时就创建对话框，UI 可能显示异常。

推荐的做法是在 Activity#onAttachedToWindow() 之后（ 其实最好是

Activity#onWindowFocusChanged() 之后）才创建对话框。

## 【推荐】尽量不要使用AnimationDrawable

尽量不要使用AnimationDrawable，它在初始化的时候就将所有图片加载

到内存中，特别占内存，并且还不能释放，释放之后下次进入再次加载时会报错。

说明：

Android 的帧动画可以使用AnimationDrawable 实现，但是如果你的帧动画中如果

包含过多帧图片，一次性加载所有帧图片所导致的内存消耗会使低端机发生OOM

异常。帧动画所使用的图片要注意降低内存消耗，当图片比较大时，容易出现OOM。

正例：

图片数量较少的AnimationDrawable 还是可以接受的。

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. <?xml version="1.0" encoding="utf-8"?>
2. <animation-list xmlns:android="http://schemas.android.com/apk/res/android" android:oneshot
3. ="true">
4. <item android:duration="500" android:drawable="@drawable/ic\_heart\_100"/>
5. <item android:duration="500" android:drawable="@drawable/ic\_heart\_75"/>
6. <item android:duration="500" android:drawable="@drawable/ic\_heart\_50"/>
7. <item android:duration="500" android:drawable="@drawable/ic\_heart\_25"/>
8. <item android:duration="500" android:drawable="@drawable/ic\_heart\_0"/>
9. </animation-list>

多了就不要了

## 【强制】不能使用ScrollView 包裹ListView

【强制】不能使用ScrollView 包裹ListView/GridView/ExpandableListVIew;因为这

样会把ListView 的所有Item 都加载到内存中，要消耗巨大的内存和cpu 去绘制图

面。

说明：

ScrollView 中嵌套List 或RecyclerView 的做法官方明确禁止。除了开发过程中遇到

的各种视觉和交互问题，这种做法对性能也有较大损耗。ListView 等UI 组件自身有

垂直滚动功能，也没有必要在嵌套一层ScrollView。目前为了较好的UI 体验，更贴

近Material Design 的设计，推荐使用NestedScrollView。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. <android.support.v4.widget.NestedScrollView>
2. <LinearLayout>
3. <ImageView/>
4. ...
5. <android.support.v7.widget.RecyclerView/>
6. </LinearLayout>
7. </android.support.v4.widget.NestedScrollView>

## 【强制】不要在Android 的Application 对象中缓存数据

】不要在Android 的Application 对象中缓存数据。基础组件之间的数据共享

请使用Intent 等机制，也可使用SharedPreferences 等数据持久化机制。

反例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **class** MyApplication **extends** Application {
2. String username;
4. String getUsername() {
5. **return** username;
6. }
8. **void** setUsername(String username) {
9. **this**.username = username;
10. }
11. }

## 【推荐】全局的Toast 对象

使用Toast 时，建议定义一个全局的Toast 对象，这样可以避免连续显示

Toast 时不能取消上一次Toast 消息的情况。即使需要连续弹出Toast，也应避免直

接调用Toast#makeText。

## 【强制】ViewHolder更新缓存

使用Adapter 的时候，如果你使用了ViewHolder 做缓存，在getView()的

方法中无论这项convertView 的每个子控件是否需要设置属性(比如某个TextView

设置的文本可能为null，某个按钮的背景色为透明，某控件的颜色为透明等)，都需

要为其显式设置属性(Textview 的文本为空也需要设置setText("")，背景透明也需要

设置)，否则在滑动的过程中，因为adapter item 复用的原因，会出现内容的显示错

乱。

# 进程、线程与消息通信

## 【强制】Binder 1M限制

不要通过Intent 在Android 基础组件之间传递大数据（binder transaction

缓存为1MB），可能导致OOM。

## 【强制Application按进程需要来初始化

在Application 的业务初始化代码加入进程判断，确保只在自己需要的进程

初始化。特别是后台进程减少不必要的业务初始化。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **class** MyApplication **extends** Application {
2. @Override
3. **public** **void** onCreate() {
4. //在所有进程中初始化
5. ....
6. //仅在主进程中初始化
7. **if** (mainProcess) {
8. ...
9. }
10. //仅在后台进程中初始化
11. **if** (bgProcess) {
12. ...
13. }
14. }
15. }

## 【强制】线程池提供创建线程

新建线程时，必须通过线程池提供（AsyncTask 或者ThreadPoolExecutor或者其他形式自定义的线程池），不允许在应用中自行显式创建线程。

说明：

使用线程池的好处是减少在创建和销毁线程上所花的时间以及系统资源的开销，解决资源不足的问题。如果不使用线程池，有可能造成系统创建大量同类线程而导致

消耗完内存或者“过度切换”的问题。另外创建匿名线程不便于后续的资源使用分析，

对性能分析等会造成困扰。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **int** NUMBER\_OF\_CORES = Runtime.getRuntime().availableProcessors();
2. **int** KEEP\_ALIVE\_TIME = 1;
3. TimeUnit KEEP\_ALIVE\_TIME\_UNIT = TimeUnit.SECONDS;
4. BlockingQueue<Runnable> taskQueue = **new** LinkedBlockingQueue<Runnable>();
5. ExecutorService executorService = **new** ThreadPoolExecutor(NUMBER\_OF\_CORES,
6. NUMBER\_OF\_CORES\*2, KEEP\_ALIVE\_TIME, KEEP\_ALIVE\_TIME\_UNIT, taskQueue,
7. **new** BackgroundThreadFactory(), **new** DefaultRejectedExecutionHandler());
8. //执行任务
9. executorService.execute(**new** Runnnable() {
10. ...
11. });

反例：

new Thread(new Runnable() {

@Override

public void run() {

//操作语句

...

}

}).start();

## 【强制】线程池不允许使用Executors 去创建

线程池不允许使用Executors 去创建，而是通过ThreadPoolExecutor 的方

式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明：

Executors 返回的线程池对象的弊端如下：

1) FixedThreadPool 和SingleThreadPool ： 允许的请求队列长度为Integer.MAX\_VALUE，可能会堆积大量的请求，从而导致OOM；

2) CachedThreadPool 和ScheduledThreadPool ： 允许的创建线程数量为

Integer.MAX\_VALUE，可能会创建大量的线程，从而导致OOM。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **int** NUMBER\_OF\_CORES = Runtime.getRuntime().availableProcessors();
2. **int** KEEP\_ALIVE\_TIME = 1;
3. TimeUnit KEEP\_ALIVE\_TIME\_UNIT = TimeUnit.SECONDS;
4. BlockingQueue<Runnable> taskQueue = **new** LinkedBlockingQueue<Runnable>();
5. ExecutorService executorService = **new** ThreadPoolExecutor(NUMBER\_OF\_CORES,
6. NUMBER\_OF\_CORES\*2, KEEP\_ALIVE\_TIME, KEEP\_ALIVE\_TIME\_UNIT,
7. taskQueue, **new** BackgroundThreadFactory(), **new** DefaultRejectedExecutionHandler());

反例：

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

## 【强制】子线程中不能更新界面，更新界面必须在主线程中进行，网络操作不能在

主线程中调用。

## 【推荐】尽量减少不同APP 之间的进程间通信及拉起行为。拉起导致占用系统资源，

**影响用户体验。**

## 【推荐】定义能识别自己业务的线程名称

新建线程时，定义能识别自己业务的线程名称，便于性能优化和问题排查。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **class** MyThread **extends** Thread {
2. **public** MyThread() {
3. **super**.setName("ThreadName");
4. …
5. }
6. }

## 【推荐】设置线程存活时间(setKeepAliveTime)

ThreadPoolExecutor 设置线程存活时间(setKeepAliveTime)，确保空闲时线程能被释放。

## 【推荐】禁止在多进程之间用SharedPreferences 共享数据

禁止在多进程之间用SharedPreferences 共享数据， 虽然可以

(MODE\_MULTI\_PROCESS)，但官方已不推荐。

## 【推荐】谨慎使用Android 的多进程

谨慎使用Android 的多进程，多进程虽然能够降低主进程的内存压力，但

会遇到如下问题：

1) 首次进入新启动进程的页面时会有延时的现象（有可能黑屏、白屏几秒，是白

屏还是黑屏和新Activity 的主题有关）；

2) 应用内多进程时，Application 实例化多次，需要考虑各个模块是否都需要在所

有进程中初始化。

# 文件与数据库

## 【强制】不要硬编码文件路径

任何时候不要硬编码文件路径，请使用Android 文件系统API 访问。

说明：

Android 应用提供内部和外部存储，分别用于存放应用自身数据以及应用产生的用

户数据。可以通过相关API 接口获取对应的目录，进行文件操作。

android.os.Environment#getExternalStorageDirectory()

android.os.Environment#getExternalStoragePublicDirectory()

android.content.Context#getFilesDir()

android.content.Context#getCacheDir

public File getDir(String alName) {

File file = new File(Environment.getExternalStoragePublicDirectory(Environment.

DIRECTORY\_PICTURES), alName);

if (!file.mkdirs()) {

Log.e(LOG\_TAG, "Directory not created");

}

return file;

}

反例：

public File getDir(String alName) {

// 任何时候都不要硬编码文件路径，这不仅存在安全隐患，也让app 更容易出现适配问题

File file = new File("/mnt/sdcard/Download/Album", alName);

if (!file.mkdirs()) {

Log.e(LOG\_TAG, "Directory not created");

}

return file;

## 【强制】当使用外部存储时，必须检查外部存储的可用性。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. // 读/写检查
2. **public** **boolean** isExternalStorageWritable() {
3. String state = Environment.getExternalStorageState();
4. **if** (Environment.MEDIA\_MOUNTED.equals(state)) {
5. **return** **true**;
6. }
7. **return** **false**;
8. }
10. // 只读检查
11. **public** **boolean** isExternalStorageReadable() {
12. String state = Environment.getExternalStorageState();
13. **if** (Environment.MEDIA\_MOUNTED\_READ\_ONLY.equals(state)) {
14. **return** **true**;
15. }
16. **return** **false**;
17. }

## 【强制】应用间共享文件时，不要通过放宽文件系统权限的方式去实现，而应使用

FileProvider。

**正例：**

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. - 37 -
2. <!-- AndroidManifest.xml -->
3. <manifest>
4. ...
5. <application>
6. ...<provider
7. android:name="android.support.v4.content.FileProvider"
8. android:authorities="com.example.fileprovider"
9. android:exported="false"
10. android:grantUriPermissions="true">
11. <meta-data
12. android:name="android.support.FILE\_PROVIDER\_PATHS"
13. android:resource="@xml/provider\_paths" />
14. </provider>
15. ...
16. </application>
17. </manifest>
18. <!-- res/xml/provider\_paths.xml -->
19. <paths>
20. <files-path path="album/" name="myimages" />
21. </paths>
22. **void** getAlbumImage(String imagePath) {
23. File image = **new** File(imagePath);
24. Intent getAlbumImageIntent = **new** Intent(MediaStore.ACTION\_IMAGE\_CAPTURE);
25. Uri imageUri = FileProvider.getUriForFile(
26. **this**,
27. "com.example.provider",
28. image);
29. getAlbumImageIntent.putExtra(MediaStore.EXTRA\_OUTPUT, imageUri);
30. startActivityForResult(takePhotoIntent, REQUEST\_GET\_ALBUMIMAGE);
31. }

反例

void getAlbumImage(String imagePath) {

File image = new File(imagePath);

Intent getAlbumImageIntent = new Intent(MediaStore.ACTION\_IMAGE\_CAPTURE);

//不要使用file://的URI 分享文件给别的应用，包括但不限于Intent

getAlbumImageIntent.putExtra(MediaStore.EXTRA\_OUTPUT, Uri.fromFile(image));

startActivityForResult(takePhotoIntent, REQUEST\_GET\_ALBUMIMAGE);

}...

## 【推荐】SharedPreference 中只能存储简单数据类型（int、boolean、String 等），

**复杂数据类型建议使用文件、数据库等其他方式存储。**

## 【推荐】尽量使用Editor#apply()

SharedPreference 提交数据时， 尽量使用Editor#apply() ， 而非Editor#commit()。一般来讲，仅当需要确定提交结果，并据此有后续操作时，才使用Editor#commit()。

说明：

SharedPreference 相关修改使用apply 方法进行提交会先写入内存，然后异步写入

磁盘， commit 方法是直接写入磁盘。如果频繁操作的话apply 的性能会优于commit，

apply 会将最后修改内容写入磁盘。但是如果希望立刻获取存储操作的结果，并据此

做相应的其他操作，应当使用commit。

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** updateSettingsAsync() {
2. SharedPreferences mySharedPreferences = getSharedPreferences("settings",
3. Activity.MODE\_PRIVATE);
4. SharedPreferences.Editor editor = mySharedPreferences.edit();
5. editor.putString("id", "foo");
6. editor.apply();
7. }
9. **public** **void** updateSettings() {
10. SharedPreferences mySharedPreferences = getSharedPreferences("settings",
11. Activity.MODE\_PRIVATE);
12. SharedPreferences.Editor editor = mySharedPreferences.edit();
13. editor.putString("id", "foo");
14. **if** (!editor.commit()) {
15. Log.e(LOG\_TAG, "Failed to commit setting changes");
16. }
17. }

反例：

editor.putLong("key\_name", "long value");

editor.commit();

## 【强制】数据库Cursor 必须确保使用完后关闭

说明：

Cursor 是对数据库查询结果集管理的一个类，当查询的结果集较小时，消耗内存不

易察觉。但是当结果集较大，长时间重复操作会导致内存消耗过大，需要开发者在

操作完成后手动关闭Cursor。

数据库Cursor 在创建及使用时，可能发生各种异常，无论程序是否正常结束，必须

在最后确保Cursor 正确关闭，以避免内存泄漏。同时，如果Cursor 的使用还牵涉

多线程场景，那么需要自行保证操作同步。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** handlePhotos(SQLiteDatabase db, String userId) {
2. Cursor cursor;
3. **try** {
4. cursor = db.query(TUserPhoto, **new** String[]{"userId", "content"}, "userId=?", **new**
5. String[]{userId}, **null**, **null**, **null**);
6. **while** (cursor.moveToNext()) {
7. // TODO
8. }
9. } **catch** (Exception e) {
10. // TODO
11. } **finally** {
12. **if** (cursor != **null**) {
13. cursor.close();
14. }
15. }
16. }

## 【强制】多线程操作写入数据库时，需要使用事务，以免出现同步问题。

说明：

通过SQLiteOpenHelper 获取数据库SQLiteDatabase 实例，Helper 中会自动缓存

已经打开的SQLiteDatabase 实例，单个App 中应使用SQLiteOpenHelper 的单例

模式确保数据库连接唯一。由于SQLite 自身是数据库级锁，单个数据库操作是保证

线程安全的（不能同时写入），transaction 是一次原子操作，因此处于事务中的操作

是线程安全的。

若同时打开多个数据库连接，并通过多线程写入数据库，会导致数据库异常，提示

数据库已被锁住。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** insertUserPhoto(SQLiteDatabase db, String userId, String content) {
2. ContentValues cv = **new** ContentValues();
3. cv.put("userId", userId);
4. cv.put("content", content);
5. db.beginTransaction();
6. **try** {
7. db.insert(TUserPhoto, **null**, cv);
8. // 其他操作
9. db.setTransactionSuccessful();
10. } **catch** (Exception e) {
11. // TODO
12. } **finally** {
13. db.endTransaction();
14. }
15. }

反例：

public void insertUserPhoto(SQLiteDatabase db, String userId, String content) {

ContentValues cv = new ContentValues();

cv.put("userId", userId);

cv.put("content", content);

db.insert(TUserPhoto, null, cv);

## 【推荐】大数据写入数据库时，请使用事务或其他能够提高I/O 效率的机制，保证执

行速度。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** insertBulk(SQLiteDatabase db, ArrayList<UserInfo> users) {
2. db.beginTransaction();
3. **try** {
4. **for** (**int** i = 0; i < users.size; i++) {
5. ContentValues cv = **new** ContentValues();
6. cv.put("userId", users[i].userId);
7. cv.put("content", users[i].content);
8. db.insert(TUserPhoto, **null**, cv);
9. }
10. // 其他操作
11. db.setTransactionSuccessful();
12. } **catch** (Exception e) {
13. // TODO
14. } **finally** {
15. db.endTransaction();
16. }
17. }

## 【强制】执行SQL 语句时，应使用SQLiteDatabase#insert()、update()、delete()，不要使用SQLiteDatabase#execSQL()，以免SQL 注入风险。

正例：

public int updateUserPhoto(SQLiteDatabase db, String userId, String content) {

ContentValues cv = new ContentValues();

cv.put("content", content);

String[] args = {String.valueOf(userId)};

return db.update(TUserPhoto, cv, "userId=?", args);

}

反例：

public void updateUserPhoto(SQLiteDatabase db, String userId, String content) {

String sqlStmt = String.format("UPDATE %s SET content=%s WHERE userId=%s",

TUserPhoto, userId, content);

//请提高安全意识，不要直接执行字符串作为SQL 语句

db.execSQL(sqlStmt);

}

## 【强制】如果ContentProvider 管理的数据存储在SQL 数据库中，应该避免将不受信任的外部数据直接拼接在原始SQL 语句中。

正例：

// 使用一个可替换参数

String mSelectionClause = "var = ?";

String[] selectionArgs = {""};

selectionArgs[0] = mUserInput;

反例：

// 拼接用户输入内容和列名

String mSelectionClause = "var = " + mUserInput;

# Bitmap、Drawable 与动画

## 【强制】加载大图片或者一次性加载多张图片，应该在异步线程中进行。图片的加

载，涉及到IO 操作，以及CPU 密集操作，很可能引起卡顿。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **class** BitmapWorkerTask **extends** AsyncTask<Integer, Void, Bitmap> {
2. // 在后台进行图片解码
3. @Override
4. **protected** Bitmap doInBackground(Integer... params) {
5. **final** Bitmap bitmap = BitmapFactory.decodeFile("some path");
6. **return** bitmap;
7. }
8. }

反例：

Button btnLoadImage = (Button) findViewById(R.id.btn);

btnLoadImage.setOnClickListener(new OnClickListener(){

public void onClick(View v) {

Bitmap bitmap = BitmapFactory.decodeFile("some path");

}

});

## 【强制】在ListView，ViewPager，RecyclerView，GirdView 等组件中使用图片时，

应做好图片的缓存，避免始终持有图片导致内存溢出，也避免重复创建图片，引起

性能问题。建议使用Fresco （ https://github.com/facebook/fresco ）、Glide

（https://github.com/bumptech/glide）等图片库。

正例：

例如使用系统LruCache 缓存，参考：

<https://developer.android.com/topic/performance/graphics/cache-bitmap.html>

反例：

没有存储，每次都需要解码，或者有缓存但是没有合适的淘汰机制，导致缓存效果

很差，依然经常需要重新解码。

## 【强制】png 图片使用TinyPNG 或者类似工具压缩处理，减少包体积。

## 【推荐】应根据实际展示需要，压缩图片

【推荐】应根据实际展示需要，压缩图片，而不是直接显示原图。手机屏幕比较小，

直接显示原图，并不会增加视觉上的收益，但是却会耗费大量宝贵的内存。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **static** Bitmap decodeSampledBitmapFromResource(Resources res, **int** resId,
2. **int** reqWidth, **int** reqHeight) {
3. // 首先通过inJustDecodeBounds=true 获得图片的尺寸
4. **final** BitmapFactory.Options options = **new** BitmapFactory.Options();
5. options.inJustDecodeBounds = **true**;
6. BitmapFactory.decodeResource(res, resId, options);
7. // 然后根据图片分辨率以及我们实际需要展示的大小，计算压缩率
8. options.inSampleSize = calculateInSampleSize(options, reqWidth, reqHeight);
9. // 设置压缩率，并解码
10. options.inJustDecodeBounds = **false**;
11. **return** BitmapFactory.decodeResource(res, resId, options);
12. }

反例：

不经压缩显示原图。

## 【强制】使用完毕的图片，应该及时回收，释放宝贵的内存。

正例：

Bitmap bitmap = null;

loadBitmapAsync(new OnResult(result){

bitmap = result;

});

...使用该bitmap...

// 使用结束，在2.3.3 及以下需要调用recycle()函数，在2.3.3 以上GC 会自动管理，除非你明

确不需要再用if (Build.VERSION.SDK\_INT <= Build.VERSION\_CODES.GINGERBREAD\_MR1) {

bitmap.recycle();

}

bitmap = null;

反例：

## 【强制】在Activity#onPause()或Activity#onStop()回调中，关闭当前activity 正在执

行的的动画。

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **class** MyActivity **extends** Activity {
2. ImageView mImageView;
3. Animation mAnimation;
4. Button mBtn;
6. @Override
7. **public** **void** onCreate(Bundle savedInstanceState) {
8. **super**.onCreate(savedInstanceState);
9. setContentView(R.layout.main);
10. mImageView = (ImageView) findViewById(R.id.ImageView01);
11. mAnimation = AnimationUtils.loadAnimation(**this**, R.anim.anim);
12. mBtn = (Button) findViewById(R.id.Button01);
13. mBtn.setOnClickListener(**new** View.OnClickListener() {
14. @Override
15. **public** **void** onClick(View v) {
16. mImageView.startAnimation(mAnimation);
17. }
18. });
19. }
21. @Override
22. **public** **void** onPause() {
23. //页面退出，及时清理动画资源
24. mImageView.clearAnimation()
25. }
26. }

反例：

## 【推荐】页面退出时，不关闭该页面相关的动画。

在动画或者其他异步任务结束时，应该考虑回调时刻的环境是否还支持业

务处理。例如Activity 的onStop()函数已经执行，且在该函数中主动释放了资源，

此时回调中如果不做判断就会空指针崩溃。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **class** MyActivity **extends** Activity {
2. **private** ImageView mImageView;
3. **private** Animation mAnimation;
5. @Override
6. **public** **void** onCreate(Bundle savedInstanceState) {
7. **super**.onCreate(savedInstanceState);
8. setContentView(R.layout.main);
9. mImageView = (ImageView) findViewById(R.id.ImageView01);
10. mAnimation = AnimationUtils.loadAnimation(**this**, R.anim.anim);
11. mAnimation.setAnimationListener(**new** AnimationListener() {
12. @Override
13. **public** **void** onAnimationEnd(Animation arg0) {
14. //判断一下资源是否被释放了
15. **if** (mImageView != **null**) {
16. mImageView.clearAnimation();
17. }
18. }
19. });
20. mImageView.startAnimation(mAnimation);
21. }
22. }

反例：

动画结束回调中，直接使用资源不加判断，导致异常。

## 【推荐】使用 inBitmap 重复利用内存空间，避免重复开辟新内存。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **static** Bitmap decodeSampledBitmapFromFile(String filename, **int** reqWidth, **int**
2. reqHeight, ImageCache cache) {
3. **final** BitmapFactory.Options options = **new** BitmapFactory.Options();
4. ...
5. BitmapFactory.decodeFile(filename, options);
6. ...
7. // 如果在Honeycomb 或更新版本系统中运行，尝试使用inBitmap
8. **if** (Utils.hasHoneycomb()) {
9. addInBitmapOptions(options, cache);
10. }
11. ...
12. **return** BitmapFactory.decodeFile(filename, options);
13. }
15. **private** **static** **void** addInBitmapOptions(BitmapFactory.Options options,
16. ImageCache cache) {
17. // inBitmap 只处理可变的位图，所以强制返回可变的位图
18. options.inMutable = **true**;
19. **if** (cache != **null**) {
20. Bitmap inBitmap = cache.getBitmapFromReusableSet(options);
21. **if** (inBitmap != **null**) {
22. options.inBitmap = inBitmap;
23. }
24. }
25. }

## 【推荐】使用RGB\_565 代替RGB\_888，

使用RGB\_565 代替RGB\_888，在不怎么降低视觉效果的前提下，减少内

存占用。

说明：

android.graphics.Bitmap.Config 类中关于图片颜色的存储方式定义：

1) ALPHA\_8 代表8 位Alpha 位图；

2) ARGB\_4444 代表16 位ARGB 位图；

3) ARGB\_8888 代表32 位ARGB 位图；

4) RGB\_565 代表8 位RGB 位图。

位图位数越高，存储的颜色信息越多，图像也就越逼真。大多数场景使用的是

ARGB\_8888 和RGB\_565，RGB\_565 能够在保证图片质量的情况下大大减少内存

的开销，是解决OOM 的一种方法。

但是一定要注意RGB\_565 是没有透明度的，如果图片本身需要保留透明度，那么

就不能使用RGB\_565。

正例：

Config config = drawableSave.getOpacity() != PixelFormat.OPAQUE ? Config.ARGB\_8565 :

Config.RGB\_565;

Bitmap bitmap = Bitmap.createBitmap(w, h, config);

反例：

Bitmap newb = Bitmap.createBitmap(width, height, Config.ARGB\_8888);

## 【. 推荐】尽量减少 Bitmap（BitmapDrawable）的使用

推荐】尽量减少 Bitmap（BitmapDrawable）的使用，尽量使用纯色（ColorDrawable）、

渐变色（GradientDrawable）、StateSelector（StateListDrawable）等与Shape 结

合的形式构建绘图。

## 【推荐】谨慎使用gif 图片

，注意限制每个页面允许同时播放的gif 图片，以及单个

gif 图片的大小。

## 【参考】大图片资源不要直接打包到apk

可以考虑通过文件仓库远程下载，减小包

体积。

## 【推荐】根据设备性能，选择性开启复杂动画，以实现一个整体较优的性能和体验；

## 【推荐】postDelay 替代onAnimationEnd

在有强依赖 onAnimationEnd 回调的交互时，如动画播放完毕才能操作页

面， onAnimationEnd 可能会因各种异常没被回调（ 参考：

https://stackoverflow.com/questions/5474923/onanimationend-is-not-getting-calle

d-onanimationstart-works-fine ）， 建议加上超时保护或通过 postDelay 替代

onAnimationEnd。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. View v = findViewById(R.id.xxxViewID);
2. **final** FadeUpAnimation anim = **new** FadeUpAnimation(v);
3. anim.setInterpolator(**new**    AccelerateInterpolator());
4. anim.setDuration(1000);
5. anim.setFillAfter(**true**);
6. **new**   Handler().postDelayed(**new** Runnable() {
7. **public** **void** run () {
8. **if** (v != **null**) {
9. v.clearAnimation();
10. }
11. }
12. },anim.getDuration());
13. v.startAnimation(anim);

## 【推荐】当View Animation 执行结束时，调用View.clearAnimation()释放相关资源。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. View v = findViewById(R.id.xxxViewID);
2. **final** FadeUpAnimation anim = **new** FadeUpAnimation(v);
3. anim.setInterpolator(**new** AccelerateInterpolator());
4. anim.setDuration(1000);
5. anim.setFillAfter(**true**);
6. anim.setAnimationListener(**new** AnimationListener() {
7. @Override
8. **public** **void** onAnimationEnd(Animation arg0) {
9. //判断一下资源是否被释放了
10. **if** (v != **null**) {
11. v.clearAnimation();
12. }
13. }
14. });
15. v.startAnimation(anim);
16. }

# 安全

## 【强制】禁止使用常量初始化矢量参数构建IvParameterSpec，建议IV 通过随机方式产生。

说明：

使用常量初始化向量，密码文本的可预测性会高得多，容易受到字典式攻击。iv 的

作用主要是用于产生密文的第一个block，以使最终生成的密文产生差异（明文相同

的情况下），使密码攻击变得更为困难。

正例：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **byte**[] rand = **new** **byte**[16];
2. SecureRandom r = **new** SecureRandom();
3. r.nextBytes(rand);
4. IvParameterSpec iv = **new** IvParameterSpec(rand);

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. 反例：
2. IvParameterSpec iv\_ = **new** IvParameterSpec("1234567890".getBytes());
3. System.out.println(iv.getIV());

## 【强制】将android:allowbackup 属性必须设置为false，阻止应用数据被导出。

说明：

android:allowBackup 原本是 Android 提供的 adb 调试功能，如果设置为 true，

可以导出应用数据备份并在任意设备上恢复。这对应用安全性和用户数据隐私构成

极大威胁，所以必须设置为 false，防止数据泄露。

<application

android:allowBackup="false"

android:largeHeap="true"

android:icon="@drawable/test\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

## 【强制】如果使用自定义HostnameVerifier 实现类，必须在verify()方法中校验服务

器主机名的合法性，否则可能受到中间人攻击。