技 术 文 件

文件名称 ：Android编码规范

文件编号 ：

版 本 ： **Draft**

共 16 页

(包括封面)

拟 制 徐申龙

审 核

批 准

修改记录

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 文件编号 | 版本号 | 拟制人/  修改人 | 拟制/修改日期 | 更改理由 | 主要更改内容  （写要点即可） |
|  | Draft | 徐申龙 | 2017-05-5 | 初稿 |  |
|  |  | 徐申龙 | 2017-6-13 | 规范文档 | 重新排版，组织篇章 |
|  |  |  |  |  |  |
| 注1：每次更改归档文件（指归档到事业部或公司档案室的文件）时，需填写此表。  注2：文件第一次归档时，“更改理由”、“主要更改内容”栏写“无”。 | | | | | |

[1 命名规范 4](#_Toc485299264)

[1.1 目录和包的命名 4](#_Toc485299265)

[1.1.1 Java包的命名 4](#_Toc485299266)

[1.1.2 资源文件的目录结构 5](#_Toc485299267)

[1.2 文件命名 6](#_Toc485299268)

[1.2.1 类文件 6](#_Toc485299269)

[1.2.2 资源文件 7](#_Toc485299270)

[1.3 方法名 8](#_Toc485299271)

[1.4 变量命名 9](#_Toc485299272)

[1.4.1 全局变量定义及命名 9](#_Toc485299273)

[1.4.2 字符串常量，命名和取值 10](#_Toc485299274)

[1.4.3 普遍变量 10](#_Toc485299275)

[1.4.4 泛型 11](#_Toc485299276)

[2 Java排版 11](#_Toc485299277)

[2.1 使用空格 11](#_Toc485299278)

[2.1.1 缩进 11](#_Toc485299279)

[2.1.2 空格分割 12](#_Toc485299280)

[2.2 使用标准的大括号风格 12](#_Toc485299281)

[2.3 文件长度 12](#_Toc485299282)

[2.4 函数行数 13](#_Toc485299283)

[2.5 代码行长度限制 13](#_Toc485299284)

[2.5.1 换行策略 13](#_Toc485299285)

[2.6 排序import语句 14](#_Toc485299286)

[2.6.1 完全的import语句 15](#_Toc485299287)

[2.7 类成员排序 15](#_Toc485299288)

[2.8 方法的参数顺序 16](#_Toc485299289)

[2.9 Fragment和Activity的参数 16](#_Toc485299290)

[注意2: 如果提供上面方法，这些extras对应的键和参数应该private，因为他们不需要暴露到类以外。 17](#_Toc485299291)

[2.10 XML 风格规定 17](#_Toc485299292)

[2.10.1 使用自关闭的标签 17](#_Toc485299293)

[2.10.2 资源命名 17](#_Toc485299294)

[2.10.3 属性顺序 18](#_Toc485299295)

[3 注释 19](#_Toc485299296)

[3.1 Javadoc注释 19](#_Toc485299297)

[4 内存管理 20](#_Toc485299298)

[4.1 一般内存泄漏(traditional memory leak) 20](#_Toc485299299)

[4.1.1 Cursor忘记关闭 20](#_Toc485299300)

[4.1.2 Stream 21](#_Toc485299301)

[4.1.3 .Bitmap 没有及时回收 21](#_Toc485299302)

[4.1.4 服务未取消监听 21](#_Toc485299303)

[4.2 逻辑内存泄漏(logical memory leak) 22](#_Toc485299304)

[4.2.1 static对象 22](#_Toc485299305)

[4.2.2 非静态内部类和匿名内部类 22](#_Toc485299306)

[5 异常处理 24](#_Toc485299307)

[5.1 不要忽略了异常 24](#_Toc485299308)

[5.2 不捕获一般异常。 25](#_Toc485299309)

[5.3 不要用finalizer 25](#_Toc485299310)

[5.4 清理使用的资源 26](#_Toc485299311)

[5.5 重新抛出异常 27](#_Toc485299312)

[执行异常处理时，有时希望重新抛出已经处理过的异常，可以让客户也能处理这个异常。 27](#_Toc485299313)

[6 注解 27](#_Toc485299314)

[6.1 注解实践规范 27](#_Toc485299315)

[6.2 注解风格 28](#_Toc485299316)

[6.2.1 类，方法及构造器 28](#_Toc485299317)

[6.2.2 全局变量 28](#_Toc485299318)

[6.2.3 限制变量范围 28](#_Toc485299319)

[7 日志(Loging)指南 28](#_Toc485299320)

# 命名规范

主要的的命名规范有以下三种：

* Camel 标记法: 首字母是小写的,接下来的单词都以大写字母开头
* Pascal 标记法: 首字母是大写的,接下来的单词都以大写字母开头
* 匈牙利标记法: 在以Pascal标记法的变量前附加小写序列说明该变量的类型

## 目录和包的命名

### Java包的命名

包名按照域名的范围从大到小逐步列出，恰好和Internet上的域名命名规则相反。包名限小写英文字母、数字，不含下划线（”\_”），且以小写字母开头。

由一组以“.”连接的标识符构成，通常第一个标识符为符合网络域名的两个或者三个英文小写字母；第二个为公司名，一般就是”jrd”；第三项为app名，或者库名；后面再是模块的名称，子模块名称。不要超过6个层级。

com.jrd.smartlinkv3.mediaplayer.video

com.jrd.widget

com.jrd.httpservice

### 资源文件的目录结构

这个按照Android的规范，都是放在src/main/res目录下

* 布局文件放在layout，强制竖屏的放在layout-port, 强制横屏的放在目录 layout-land。
* 图片放在drawable-mdpi, drawable-hdpi，drawable-xhdpi, drawable-xlarge等目录，但不要放在drawable目录。

分辨率高的图片要放在分辨率的目录。比如，一个Icon只有一张图像，而不是一套的（即有多种分辨率的图片可以放到对应分辨率目录），那么，把这种图片放在低分辨率目录（如drawable-mdpi）显式出来的会大一点，放到高分辨率的目录（如drawable-xlarge）显示出来会小一点。实际项目中，很多图标只有一张图片，我们要根据效果，调整图片的存放目录，而不是在代码里硬编码缩放图片尺寸。

* drawable目录放xml描述的图像，即以shape, layer-list, selector, rotate, transition为root 元素的xml文件。
* anim目录放动画效果的描述文件，即cycleInterpolator, translate, set, animated-rotate, animation-list, scale为根元素的xml文件。
* values 是各种资源文件，包括字符串、颜色、尺寸、属性、风格, 以及数值（integer, fraction, bool, plurals），他们都以resources为根元素，但各种资源分文件归类：
  + strings.xml, 字符串资源， 即string元素。
  + colors.xml, 定义颜色资源，即color元素。
  + dimens.xml, 定义跟UI相关的尺寸（像素），即dimen元素。
  + arrays.xml, 字符串数组以及整形数组等，即array, string-array, integer-array元素，这些数组单元以item声明，item不需要任何属性，只需指定元素值。
  + styles.xml, 样式和主题声明，即style元素，style的各个属性也已item元素指定，但是这些item元素要指定name属性和值。
  + attrs.xml，这个是一般给自定义的View使用的。以declare-styleable元素为单元，各个属性由旗下attr声明，attr元素只有name和format属性，没有值。format表示的属性的数据类型，如boolean, integer, float, string, dimen, color。
* values的variant，简单举例如下：
  + values 目录的strings.xml是英语字符串，各种语言翻译版本放在values-<locale>目录下，比如values-419, values-es, values-it。
  + values-v11是Android API 11特定的资源，如果App运行的是Android API 11，这个目录的资源会优先使用。
  + values-xlarge，适用大屏的资源。
* menu目录放的xml是描述菜单的。
* xml目录存放其他的未归类的xml文件。比如以appwidget-provider, searchable, PreferenceScreen等为根元素的xml文件。
* 原生的文件，即不需要加工文件，放raw 目录。

注意上述目录中，只能放文件，不能包含子目录。特别，如果某些原生的资源有子目录，请放在目录src/main/asset下。

## 文件命名

### 类文件

类名书写方式是 [大写的驼峰命名](http://en.wikipedia.org/wiki/CamelCase).

#### 继承于Android组件的类

名称应该以继承的组件命名，比如：SignInActivity, SignInFragment, ImageUploaderService, ChangePasswordDialog.

#### 当要区别接口和实现类的时

可以在类的后面加上“Impl”。

interface Container {  
}  
public class ContainerImpl {  
}

#### Exception类

最好能用“Exception”做为类命名的结尾。

class DataNotFoundException extends Exception {  
   
}  
class InvalidArgumentException extends Exception {  
   
}

#### 抽象类

最好能用“Abstract”做为类命名的开头

public abstract class AbstractBean {  
   
}  
   
public abstract class AbstractBeanFactory {  
   
}

#### Test类

以“Test”做为类命名的结尾，如ContainerTest。

#### 把首字母缩写当做一个单词来看

| **推荐方式** | **不推荐方式** |
| --- | --- |
| XmlHttpRequest | XMLHTTPRequest |
| getCustomerId | getCustomerID |
| String url | String URL |
| long id | long ID |

### 资源文件

资源文件书写方式：小写\_下划线。如：ic\_launcher.png

**1.1.2.1 Drawable 文件**

drawable类型资源命名：

| **资源类别** | **前缀** | **例子** |
| --- | --- | --- |
| Action bar | ab\_ | ab\_stacked.9.png |
| Button | btn\_ | btn\_send\_pressed.9.png |
| Dialog | dialog\_ | dialog\_top.9.png |
| Divider | divider\_ | divider\_horizontal.9.png |
| Icon | ic\_ | ic\_star.png |
| Menu | menu\_ | menu\_submenu\_bg.9.png |
| Notification | notification\_ | notification\_bg.9.png |
| Tabs | tab\_ | tab\_pressed.9.png |

图标（icons）命名方式 (取自 [Android图标官方指南](http://developer.android.com/design/style/iconography.html)):

| **资源类别** | **前缀** | **例子** |
| --- | --- | --- |
| Icons | ic\_ | ic\_star.png |
| Launcher icons | ic\_launcher | ic\_launcher\_calendar.png |
| Menu icons and Action Bar icons | ic\_menu | ic\_menu\_archive.png |
| Status bar icons | ic\_stat\_notify | ic\_stat\_notify\_msg.png |
| Tab icons | ic\_tab | ic\_tab\_recent.png |
| Dialog icons | ic\_dialog | ic\_dialog\_info.png |

selector 状态命名方式:

| **状态** | **后缀** | **例子** |
| --- | --- | --- |
| Normal (正常) | \_normal | btn\_order\_normal.9.png |
| Pressed（按下） | \_pressed | btn\_order\_pressed.9.png |
| Focused （获得焦点） | \_focused | btn\_order\_focused.9.png |
| Disabled （不可用） | \_disabled | btn\_order\_disabled.9.png |
| Selected （选中） | \_selected | btn\_order\_selected.9.png |

**1.1.2.2 布局文件**

布局文件应该跟对应的Android组件名称匹配，但是把最顶级的组件名称放在前面。比如，当我们创建一个用于 SignInActivity的布局，那布局文件名称应该是activity\_sign\_in.xml.

| **组件** | **类名** | **布局文件名** |
| --- | --- | --- |
| Activity | UserProfileActivity | activity\_user\_profile.xml |
| Fragment | SignUpFragment | fragment\_sign\_up.xml |
| Dialog | ChangePasswordDialog | dialog\_change\_password.xml |
| AdapterView item | — | item\_person.xml |
| Partial layout | — | partial\_stats\_bar.xml |

一个稍微不同的地方是当我们创建用于Adapter渲染的布局,如填充一个 ListView， 在这种情况下，布局文件命名应该是用 item\_开始。  
注意有些地方这些规则没法应用，如，当我们创建一个布局文件，用于渲染另一个布局的一部分，这样的情况下，我们应该使用partial\_前缀。

**1.1.2.3 菜单文件**

跟布局文件相似，按钮文件应该跟对应的组件匹配。比如：当我们定义一个菜单，用于 UserActivity,那么文件名应该是 activity\_user.xml

文件名不含有menu是个良好的习惯做法，因为这些文件已经位于 menu 目录了。

**1.1.2.4 取值文件(Values files)**

在values文件夹的资源文件应该是复数形式（plural），比如，strings.xml, styles.xml, colors.xml, dimens.xml, attrs.xml。

## 方法名

方法的名字的第一个单词应以小写字母作为开头，后面的单词则用大写字母开头。可以为动词或动词+名词组合。

一些通用的约定：

* 设置/获取某个值的方法，应该遵循JavaBean规范（set/get）
* 返回长度的方法，应该命名为length
* 测试某个布尔值的方法，命名以is开头
* 将对象转换为某个特定类型的方法名以to开始

Date getDate() {  
 return new Date(System.currentTimeMillis())  
}  
int length() {  
 return 0;  
}  
boolean isReady(){  
 //...  
 return false;  
}  
String toString() {  
 //Do something  
 return "Hell World";  
}

## 变量命名

### 全局变量定义及命名

全局变量应该定义在**文件的头部**,遵循下面的命名规则。

* 私有的非静态全局变量以m开头
* 私有的静态全局变量以s开头
* 其他的全局变量以小写单词开头
* 常量使用全部大写，单词间用下划线间隔（ALL\_CAPS\_WITH\_UNDERSCORES）.

例子:

public class MyClass {  
 public static final int SOME\_CONSTANT = 42;  
 public int publicField;  
 private static MyClass sSingleton;  
 int mPackagePrivate;  
 private int mPrivate;  
 protected int mProtected;  
}

### 字符串常量，命名和取值

很多AndroidSDK的元素如SharedPreferences, Bundle, 或 Intent使用键值对实现，所以即使是一个小应用，使用一堆字符串常量也是很常见的。

当使用这些组件，你**必须**定义这些键是 static final 变量，并且它们应该像下面这样使用前缀

| **元素** | **变量名前缀** |
| --- | --- |
| SharedPreferences | PREF\_ |
| Bundle | BUNDLE\_ |
| Fragment Arguments | ARGUMENT\_ |
| Intent Extra | EXTRA\_ |
| Intent Action | ACTION\_ |

注意Fragment的参数——Fragment.getArguments()也是一个Bundle。然而，因为这是Bundle非常常见的用法，我们为它定义一个不同的前缀。

例子:

|  |
| --- |
| //注意这些变量值应该和名称一致来避免问题 static final String PREF\_EMAIL = "PREF\_EMAIL"; static final String BUNDLE\_AGE = "BUNDLE\_AGE"; static final String ARGUMENT\_USER\_ID = "ARGUMENT\_USER\_ID"; //Intent相关的使用完全包名作为值 static final String EXTRA\_SURNAME = "com.myapp.extras.EXTRA\_SURNAME"; static final String ACTION\_OPEN\_USER = "com.myapp.action.ACTION\_OPEN\_USER"; 普遍变量  * 普通变量命名应该采用首字母小写，其他字母首字母大写的方式。 * i、j、k等只作为小型循环的循环索引变量。 * 数组或者容器推荐命名方式为 “名词+s”的方式   List<Person> persons = getPerson(); for(Person person : persons){  dosomeworks; }   * 用“is+形容词”来命名逻辑变量   if(isClosed){  //do some works;  ...  return; } 泛型 应该尽量简明扼要（最好是一个字母），以利于与普通的class或interface区分。   * Container中的Element应该用E表示； * Map里的key用K表示，value用V； * Type用T表示； * 异常用X表示   如果需要接收多个Type类型的参数，应该用邻接T的大写字母——例如S——来依次表示，当然也可以用T1, T2这样的方式  public class HashSet<E> extends AbstractSet<E> {…}  public class HashMap<K, V> extends AbstractMap<K, V> {…}  public class ThreadLocal<T> {…}  public interface Functor<T, X extends Throwable> {  T val() throws X; } |

# Java排版

## 使用空格

### 缩进

使用四个空格做代码块缩进:

if (x == 1) {  
 x++;  
}

换行时使用8个空格缩进:

Instrument i =  
 someLongExpression(that, wouldNotFit, on, one, line);

### 空格分割

* 逗号之后紧跟一个空格。
* 括号前后不需要空格
* 二元操作符前后跟空格。
* 一元操作符不需要空格,for语句分号后有空格

Person p = new Person(“Jack”, 16, “China”);

int i = a + b – c \* d;  
  
for(int i = 0; I < 10; i++){  
 //dosomework;  
}

## 使用标准的大括号风格

|  |
| --- |
| 大括号的开始跟代码同一行 |

|  |
| --- |
| class MyClass {  int func() {  if (something) {  // ...  } else if (somethingElse) {  // ...  } else {  // ...  }  } } |

条件语句使用大括号包住，除非条件体只有一行。  
如果条件及只有一行条件体，而且没被换行，大括号是不必的。

if (condition) body();

|  |
| --- |
| 下面的风格**不推荐的**: |

if (condition)  
 body(); // 不好的，没有大括号包围!

## 文件长度

Java文件行数请控制在2000行以内。

## 函数行数

一般函数不要超过100行为宜。如果函数体超长，请拆分几个函数。

## 代码行长度限制

代码行不应该超过**78 个字符**。如果代码行超过这个限制长度，通常有两种方式来降低长度：

* 提取一个局部变量或方法（推荐方式）.
* 把单行换行成多行。

有两种**例外**可以让一行超过78字符：

* 这行不可分割，如长URL。
* package 和 import 语句。

### 换行策略

没有精确的公式解释怎么换行，很多不同的方案是有效的。然而，这里有几个规则可以应用到常见情况。

#### 在操作符前断开

当一行被操作符断开，断开处应该在操作符**之前**，例子如下：

int longName = anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne  
 + theFinalOne;

#### 赋值符号（=）例外

在操作符前断开有一个例外，那就是赋值符号 =，应该在赋值符号**后面**断开。

int longName =  
 anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne + theFinalOne;

#### 方法链的情况

当多个方法被链接在同一行的时候——如使用Builder时，每一个方法的调用应该在独立一行，在 ‘.’之前断开。

Picasso.with(context)  
 .load("http://ribot.co.uk/images/sexyjoe.jpg")  
 .into(imageView);

#### 长参数的情况

当一个方法有很多参数或它的参数非常长，我们应该在每个逗号 ‘,’后面断开。

loadPicture(context,  
 "http://ribot.co.uk/images/sexyjoe.jpg",  
 mImageViewProfilePicture,  
 clickListener,  
 "Title of the picture");

#### RxJava 链式风格

Rx链式操作符要求换行。每一个操作符必须在新的一行，断行应该在‘.’之前。

public Observable<Location> syncLocations() {  
 return mDatabaseHelper.getAllLocations()  
 .concatMap(new Func1<Location, Observable<? extends Location>>() {  
 @Override  
 public Observable<? extends Location> call(Location location) {  
 return mRetrofitService.getLocation(location.id);  
 }  
 })  
 .retry(new Func2<Integer, Throwable, Boolean>() {  
 @Override  
 public Boolean call(Integer numRetries, Throwable throwable) {  
 return throwable instanceof RetrofitError;  
 }  
 });  
}

## 排序import语句

如果你使用IDE,如Android Studio，你不必担心这些规则，这些IDE已经遵循这些规则。如果你不用IDE，往下看这些规则:

1. import Android框架的
2. Import 来自第三方的(com, junit, net, org)
3. java 以及 javax
4. 同一个项目的 imports

为了保持跟IDE的设置一致，这些imports应该是这样的：

* 首字母分组排序，大写的字母在小写的字母前面（如Z在a前面）。
* 每个主要分组 (android, com, junit, net, org, java, javax)间应该有一行空行。

更多信息查看[这里](https://source.android.com/source/code-style.html#limit-variable-scope)

### 完全的import语句

反例: import foo.\*;

正解: import foo.Bar;

点击 [这里](https://source.android.com/source/code-style.html#fully-qualify-imports) 查看更多说明

## 类成员排序

这虽然没有简单正确的方案，但是使用一个 **逻辑的** 及 **一致的** 顺序可以提高代码的可读性和可学习性。下面是推荐的顺序：

1. 常量
2. 全局变量
3. 构造函数
4. 重载的函数和回调 (public 或 private)
5. Public 方法
6. Private 方法
7. 内部类或内部接口

例子:

public class MainActivity extends Activity {  
 private String mTitle;  
 private TextView mTextViewTitle;  
 public void setTitle(String title) {  
 mTitle = title;  
 }  
 @Override  
 public void onCreate() {  
 ...  
 }  
 private void setUpView() {  
 ...  
 }  
 static class AnInnerClass {  
 }  
}

如果你的类继承一个**Android组件**如Activity或Fragment，对重载的方法进行排序以便**跟组件的生命周期匹配**是一个最佳实践。比如，当你的Activity实现onCreate(), onDestroy(), onPause() 和 onResume()，正确的顺序是：

public class MainActivity extends Activity {  
 //顺序跟Activity的生命周期匹配  
 @Override  
 public void onCreate() {}  
 @Override  
 public void onResume() {}  
 @Override  
 public void onPause() {}  
 @Override  
 public void onDestroy() {}  
}

## 方法的参数顺序

在Android编码时候，定义一个方法拥有一个Context参数是非常常见的，如果你写一个这样的方法，**Context**应该是第一个参数。

相反的例子是**回调**接口，它应该是**最后**一个参数  
例子:

// Context 总是在第一个位  
public User loadUser(Context context, int userId);  
// 回调总是在最后一位  
public void loadUserAsync(Context context, int userId, UserCallback callback);

## Fragment和Activity的参数

当数据通过Intent 或Bundle传递到一个Activity或 Fragment，这些不同值的键**必须**遵循上面描述的规则。

当一个 Activity 或 Fragment 想要参数，它应该提供一个public static方法来简化对应的Intent 或 Fragment创建  
在Activity中这个方法通常叫 getStartIntent():

public static Intent getStartIntent(Context context, User user) {  
 Intent intent = new Intent(context, ThisActivity.class);  
 intent.putParcelableExtra(EXTRA\_USER, user);  
 return intent;  
}

对于Fragment它的名称是newInstance()，处理使用合适的参数创建Fragment。

public static UserFragment newInstance(User user) {  
 UserFragment fragment = new UserFragment;  
 Bundle args = new Bundle();  
 args.putParcelable(ARGUMENT\_USER, user);  
 fragment.setArguments(args)  
 return fragment;  
}

**注意1**: 这些方法应该在类的 onCreate()前面

# 注意2: 如果提供上面方法，这些extras对应的键和参数应该private，因为他们不需要暴露到类以外。

## XML 风格规定

### 使用自关闭的标签

当一个XML元素没有任何内容时，你**必须**使用自关闭标签。这是好例子:

<TextView  
android:id="@+id/text\_view\_profile"  
android:layout\_width="wrap\_content"  
android:layout\_height="wrap\_content" />

**不好的**例子 :

<!-- 不要这样做! -->  
<TextView  
 android:id="@+id/text\_view\_profile"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content" >  
</TextView>

### 资源命名

资源 ID 和名称使用小写加下划线方式 **lowercase\_underscore**.

#### ID 命名

ID应该使用小写加下划线命名，加上元素名称为前缀，如：

| **元素** | **前缀** |
| --- | --- |
| TextView | text\_ |
| ImageView | image\_ |
| Button | button\_ |
| Menu | menu\_ |

ImageView 例子:

<ImageView  
android:id="@+id/image\_profile"  
android:layout\_width="wrap\_content"  
android:layout\_height="wrap\_content" />

Menu 的例子

<menu xmlns:android="http://schemas.android.com/apk/res/android">  
 <item android:id="@+id/menu\_done"  
 android:title="Done" />  
</menu>

#### 字符串

字符串名称以它们所属的部分为前缀命名，比如registration\_email\_hint 或 registration\_name\_hint。如果一个字符串**不属于**任何部分，那么你应该遵循下面的规则：

| **前缀** | **描述** |
| --- | --- |
| error\_ | An error message |
| msg\_ | A regular information message |
| title\_ | A title, i.e. a dialog title |
| action\_ | An action such as “Save” or “Create” |

#### Styles 和 Themes

不像其他的资源，style命名方式是大写的驼峰是命名**UpperCamelCase**

### 属性顺序

作为通用的规则，你应该把相似的属性放在一起分组，一个对常见属性良好的排序方式是

1. View Id
2. Style
3. 布局的宽和高
4. 其他的布局属性，按字母排序
5. 剩下的属性，按字母排序

# 注释

## Javadoc注释

在每个程序的最开始部分，一般都用Javadoc注释对程序的总体描述以及版权信息，之后在主程序中可以为每个类、接口、方法、字段添加 Javadoc注释，每个注释的开头部分先用一句话概括该类、接口、方法、字段所完成的功能，这句话应单独占据一行以突出其概括作用，在这句话后面可以跟随更加详细的描述段落。在描述性段落之后还可以跟随一些以Javadoc注释标签开头的特殊段落，例如上面例子中的@auther和@version，这些段落将在生成文档中以特定方式显示

import java.net.InetAddress;  
import java.util.Arrays;  
  
*/\*\*  
 \* 该类的整体性描述。  
 \*  
 \** ***@author*** *作者  
 \** ***@version*** *1.0, 05/22/07  
 \** ***@since*** *1.0  
 \*/*public class ChannelBinding {  
 */\*\*  
 \* 对该变量的备注信息  
 \*/* private InetAddress initiator;  
  
 */\*\*  
 \* 对该变量的备注信息  
 \*/* private InetAddress acceptor;  
  
 */\*\*  
 \* 对该变量的备注信息  
 \*/* private byte[] appData;  
  
 */\*\*  
 \* 对该类的构造函数的备注信息。  
 \*  
 \** ***@param*** *initAddr 对参数的备注。  
 \** ***@param*** *acceptAddr对参数的备注。  
 \** ***@param*** *appData对参数的备注。  
 \*/* public ChannelBinding(InetAddress initAddr, InetAddress acceptAddr, byte[] appData) {  
 initiator = initAddr;  
 acceptor = acceptAddr;  
  
 if (appData != null) {  
 this.appData = new byte[appData.length];  
 java.lang.System.arraycopy(appData, 0, this.appData, 0, appData.length);  
 }  
 }  
  
 */\*\*  
 \* 对该方法备注信息  
 \*  
 \** ***@param*** *obj 参数的备注信息  
 \** ***@return*** *返回值的备注信息  
 \*/* public boolean equals(Object obj) {  
 if (this == obj) return true;  
  
 if (! (obj instanceof ChannelBinding)) return false;  
  
 ChannelBinding cb = (ChannelBinding) obj;  
 return Arrays.equals(appData, cb.appData);  
 }  
}

# 内存管理

## 一般内存泄漏(traditional memory leak)

由忘记释放分配的内存导致的。

### Cursor忘记关闭

Cursor是Android管理查询数据类。如果查询得到的数据量较小时不会有内存问题，而且虚拟机能够保证Cusor最终会被释放掉。

然而如果Cursor的数据量特别大，里面含有Blob信息，应该及时释放Cursor占用内存，而不是等待GC来处理。Android倾向于手动关闭Cursor。而且android对Cursor限制个数的。

### Stream

文件或者IO流在不使用时也一定要关闭。

### .Bitmap 没有及时回收

Bitmap占用的内存比较多，当显示大量的高分辨率图片，容易出现OOM。因此，在用完Bitmap时，要及时的recycle掉。recycle并不能确定立即就会将Bitmap释放掉，但是会给虚拟机一个暗示：“该图片可以释放了”。

现在像Glide、Picaso等库在加载图片，可以避免上述问题。所以，尽量使用这些图片加载库去处理图片加载。

### 服务未取消监听

通过[Context.getSystemService(int name)](http://developer.android.com/reference/android/content/Context.html" \l "getSystemService%28java.lang.String%29" \t "_blank)可以获取系统服务。这些服务工作在各自的进程中，帮助应用处理后台任务，处理硬件交互。如果需要使用这些服务，可以注册[监听器](https://github.com/NimbleDroid/Memory-Leaks/blob/master/app/src/main/java/com/nimbledroid/memoryleaks/MainActivity.java" \l "L136" \t "_blank)，这会导致服务持有了Context的引用，如果在Activity销毁的时候没有注销这些监听器，会导致内存泄漏。

在onResume使用registerReceiver 注册接收BroadcastReceiver ，在onPause要去注册unregisterReceiver。

同样，在使用subscribe RxJava 实现的服务后，退出Activity时要unsubscribe。

void registerListener() {  
 SensorManager sensorManager = (SensorManager) getSystemService(SENSOR\_SERVICE);  
 Sensor sensor = sensorManager.getDefaultSensor(Sensor.TYPE\_ALL);  
 sensorManager.registerListener(this, sensor, SensorManager.SENSOR\_DELAY\_FASTEST);  
}

@Override  
protected void onResume() {  
 super.onResume();

registerListener();  
 registerReceiver(mReceiver, new IntentFilter(MessageUti.USER\_LOGOUT\_REQUEST));   
}

## 逻辑内存泄漏(logical memory leak)

当应用不再需要这个对象，当仍未释放该对象的所有引用。如果持有对象的强引用，垃圾回收器是无法在内存中回收这个对象。

在Android中，导致潜在内存泄漏的陷阱不外乎两种：

* 全局进程(process-global)的static变量。这个无视应用的状态，持有Activity的强引用的怪物。
* 活在Activity生命周期之外的线程。没有清空对Activity的强引用。

### static对象

* 应该尽量避免static成员变量引用资源耗费过多的实例，比如Context。
* Context尽量使用Application Context，因为Application的Context的生命周期比较长，引用它不会出现内存泄露的问题。
* 使用WeakReference代替强引用。比如可以使用WeakReference<Context> mContextRef。

比如在类中定义了静态Activity变量，把当前运行的Activity实例赋值于这个静态变量。如果这个静态变量在Activity生命周期结束后没有清空，就导致内存泄漏。因为static变量是贯穿这个应用的生命周期的，所以被泄漏的Activity就会一直存在于应用的进程中，不会被垃圾回收器回收。

### 非静态内部类和匿名内部类

在Java中，非静态内部类和匿名内部类会持有外部类的隐式引用，而静态内部类不会。内部类的优势之一就是可以访问外部类，不幸的是，导致内存泄漏的原因，就是内部类持有外部类实例的强引用。

以下几个处理异步任务的类及其派生类就有类似的风险：

* Handler
* Threads
* AsyncTask
* TimerTask

下面的代码中，[定义匿名的Runnable类，用匿名类Handler执行](https://github.com/NimbleDroid/Memory-Leaks/blob/master/app/src/main/java/com/nimbledroid/memoryleaks/MainActivity.java" \l "L114" \t "_blank)。Runnable内部类会持有外部类的隐式引用，被传递到Handler的消息队列MessageQueue中，在Message消息没有被处理之前，Activity实例不会被销毁了，于是导致内存泄漏。

public class SampleActivity extends Activity {  
  
 private final Handler mLeakyHandler = new Handler() {  
 @Override  
 public void handleMessage(Message msg) {  
 // ...  
 }  
 }  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
  
 // Post a message and delay its execution for 10 minutes.  
 mLeakyHandler.postDelayed(new Runnable() {  
 @Override  
 public void run() { /\* ... \*/ }  
 }, 1000 \* 60 \* 10);  
  
 // Go back to the previous Activity.  
 finish();  
 }  
}

[当你在Activity中定义了匿名的AsyncTsk](https://github.com/NimbleDroid/Memory-Leaks/blob/master/app/src/main/java/com/nimbledroid/memoryleaks/MainActivity.java#L102)。当异步任务在后台执行耗时任务期间，Activity不幸被销毁了（如用户退出，系统回收），这个被AsyncTask持有的Activity实例就不会被垃圾回收器回收，直到异步任务结束。

而只要是匿名类TimerTask的实例，不管是不是在工作线程，都会持有Activity的引用，导致内存泄漏。

void scheduleTimer() {  
 new Timer().schedule(new TimerTask() {  
 @Override  
 public void run() {  
 while(true);  
 }  
 }, Long.MAX\_VALUE >> 1);  
}  
  
void spawnThread() {  
 new Thread() {  
 @Override public void run() {  
 while(true);  
 }  
 }.start();  
}

void startAsyncTask() {  
 new AsyncTask<Void, Void, Void>() {  
 @Override protected Void doInBackground(Void... params) {  
 while(true);  
 }  
 }.execute();  
}

View ttButton = findViewById(R.id.tt\_button);  
ttButton.setOnClickListener(new View.OnClickListener() {  
 @Override public void onClick(View v) {  
 scheduleTimer();  
// spawnThread();  
// startAsyncTask();  
 nextActivity();  
 }  
 });

1. 将线程的内部类，改为静态内部类。
2. 第二、如果需要引用Acitivity、Context，使用弱引用。
3. 在使用handler 的时候， 尤其用到循环调用的时候，在Activity 退出的时候注意移除。否则也会导致泄露

# 异常处理

Java可抛出三种异常：

* 受检的异常(checked exception)
* 运行时异常(run-time exception)
* 错误(error)

## 不要忽略了异常

避免如下处理:

void setServerPort(String value) {  
 try {  
 serverPort = Integer.parseInt(value);  
 } catch (NumberFormatException e) { }  
}

虽然你可能觉得代码不会抛出这个异常或者这异常不重要，但忽略异常会留下雷区，将来其他人会踩到。

空的catch块等于把整个异常丢进黑洞，能够说明何时何处为何出错的所有信息都会永远丢失。必须在代码中以某种原则处理每个异常。具体处理视情况而定 - ([Android官方代码指南](https://source.android.com/source/code-style.html))。

指南中关于不要忽略异常的说明在 [这里](https://source.android.com/source/code-style.html#dont-ignore-exceptions).

## 不捕获一般异常。

不应该这样做:

try {  
 someComplicatedIOFunction(); // 可能抛出 IOException  
 someComplicatedParsingFunction(); // 可能抛出 ParsingException  
 someComplicatedSecurityFunction(); // 可能抛出 SecurityException  
 //啊哈，统一处理掉  
} catch (Exception e) { //我将捕获所有的异常  
 handleError(); // 使用一个通用的处理手段!  
}

查看不这样做的原因以及要怎么做： [Android官方指南之不要捕获一般的异常](https://source.android.com/source/code-style.html#dont-catch-generic-exception)

如果两种不同的异常共享相同的异常处理代码，可以使用Java 7管道语法：(ExceptionType| ...| ExceptionType 变量）

try {  
 URL url = new URL("http://www.yoursimpledate.server/");  
 BufferedReader reader = new BufferedReader(  
 new InputStreamReader(url.openStream()));  
 String line = reader.readLine();  
 SimpleDateFormat format = new SimpleDateFormat("MM/DD/YY");  
 Date date = format.parse(line);  
} catch(ParseException | IOException exception) {  
// handle our problems here.  
}

## 不要用finalizer

它什么时候被调用是没有保证的，甚至它不被调用。大部分情况下，良好的异常处理来满足需要在finalizer中做的工作。如果确实需要它，定义一个 close() 方法（或类似的）然后在文档中明确的指出什么时候这个方法需要被调用。可以查看 InputStream 做为例子.这种情况下，在finalizer中打印一个简短的log信息是恰当的但不是必须的，因为我们不希望log泛滥 .([Android官方指南之不要用finalizer](https://source.android.com/source/code-style.html#dont-use-finalizers))

## 清理使用的资源

如果在处理socket，File等资源时发送异常，需要在异常处理中，关闭这些资源。

public static void main(String[] args){  
 BufferedReader reader = null;  
  
 try {  
 URL url = new URL("http://www.yoursimpledate.server/");  
 reader = new BufferedReader(new  
 InputStreamReader(url.openStream()));  
 String line = reader.readLine();  
 SimpleDateFormat format = new SimpleDateFormat("MM/DD/YY");  
 Date date = format.parse(line);  
 } catch (MalformedURLException exception) {  
 // handle passing in the wrong type of URL.  
 } catch (IOException exception) {  
 // handle I/O problems.  
 } catch (ParseException exception) {  
 // handle date parse problems.  
 } finally {  
 if (reader != null) {  
 try {  
 reader.close();  
 } catch (IOException ex) {  
 ex.printStackTrace();  
 }  
 }  
 }  
}

上述代码中会分配 BufferedReader，所以必须添加一个将其关闭的最终代码块。

在Java 7，扩充 try的语法，可以关闭资源。请注意，此特性仅对实现了 [AutoCloseable 接口](http://download.java.net/jdk7/docs/api/java/lang/AutoCloseable.html)的类起作用。

public static void main(String[] args) {  
 try (BufferedReader reader = new BufferedReader(  
 new InputStreamReader(  
 new URL("http://www.yoursimpledate.server/").openStream())))  
 {  
 String line = reader.readLine();  
 SimpleDateFormat format = new SimpleDateFormat("MM/DD/YY");  
 Date date = format.parse(line);  
 } catch (ParseException | IOException exception) {  
 // handle I/O problems.  
 }  
}

## [重新抛出异常](http://www.cnblogs.com/fxyfirst/p/3805037.html)

执行异常处理时，有时希望重新抛出已经处理过的异常，可以让客户也能处理这个异常。

try {  
 throw new IOException("Error");  
 } catch(Exception exception) {  
/\*  
 \* Do some handling and then rethrow.  
 \*/  
 throw exception;  
 }

# 注解

## 注解实践规范

根据Android代码风格指南，一些Java内置的注解标准的实践规范有：

* @Override: 无论一个方法是重载父类还是实现某个接口的都**必须使用**@Override 注解。比如，当你使用 @inheritdocs Javadoc 标签,然后 从一个类(不是一个接口)中导出，你也必须声明那个方法 @Overrides 父类的方法。
* @SuppressWarnings: @SuppressWarnings 注解应该仅当不可能消除一个警告的时候才使用。如果一个警告 通过这个 “不可能消除” 的测试, @SuppressWarnings 注解必不可少, 以便保证所有的警告在代码中反映实际的问题。

关于注解的更多指南可以查看[这里](http://source.android.com/source/code-style.html#use-standard-java-annotations).

## 注解风格

### 类，方法及构造器

当注解被应用到一个类，方法或者构造函数时候，注解在代码注释之后，**一行一个**注解

|  |
| --- |
|  |

|  |
| --- |
| /\* 这是这个类的注解 \*/  @AnnotationA  @AnnotationB  public class MyAnnotatedClass { } |

### 全局变量

全局变量的注解应该在**同一行**列出。除非这一行超过了最大字数了。

|  |
| --- |
|  |

|  |
| --- |
| @Nullable @Mock DataManager mDataManager; |

### 限制变量范围

变量的使用范围应该限制到最小(Effective Java 条目 29)。这样做，你可以为你的代码增加可读性和可维护性，减少可能的错误。每个变量应该在最内层的使用到的它代码块中定义

局部变量应该在第一次用到他们的时候定义。几乎每个局部变量都应该初始化。如果你没有足够的信息去初始化一个变量，你应该延迟声明这个变量，直到你有足够的初始化信息。.\_ - ([Android代码风格指南](https://source.android.com/source/code-style.html#limit-variable-scope))

# 日志(Loging)指南

使用 Log 类提供的日志方法来打印错误信息或对开发者分辨问题有用的其他信息：

* Log.v(String tag, String msg) (verbose)
* Log.d(String tag, String msg) (debug)
* Log.i(String tag, String msg) (information)
* Log.w(String tag, String msg) (warning)
* Log.e(String tag, String msg) (error)

通常来说，我们使用一个类的名称作为一个TAG，在文件的开头定义成一个 static final 变量，如：

public class MyClass {  
 private static final String TAG = MyClass.class.getSimpleName();  
 public myMethod() {  
 Log.e(TAG, "My error message");  
 }  
}

|  |
| --- |
|  |

VERBOSE 和 DEBUG 日志 **必须**在release版本中禁用。对于INFORMATION, WARNING 和 ERROR 日志，在release版本中有助于定位问题，并要保证这些日志信息不会泄漏email地址，用户id之类的隐私信息。  
仅限debug版本显示Log的配置：

if (BuildConfig.DEBUG) Log.d(TAG, "The value of x is " + x);