**目录**

[1 目的 1](#_Toc23258654)

[2 范围 1](#_Toc23258655)

[3 职责 1](#_Toc23258656)

[4 定义 1](#_Toc23258657)

[5 流程图 2](#_Toc23258658)

[6 内容 2](#_Toc23258659)

[6.1 IDE使用规范 2](#_Toc23258660)

[6.1.1 版本统一规范 2](#_Toc23258661)

[6.1.2 Module的命名和版本 4](#_Toc23258662)

[6.1.3 动态权限处理 5](#_Toc23258663)

[6.1.4 代码混淆 5](#_Toc23258664)

[6.2 编码规范 6](#_Toc23258665)

[6.2.1 文件名 6](#_Toc23258666)

[6.2.2 文件编码：UTF-8 6](#_Toc23258667)

[6.2.3 特殊字符 6](#_Toc23258668)

[6.2.4 源码文件结构 7](#_Toc23258672)

[6.2.5 类成员顺序 7](#_Toc23258673)

[6.2.6 花括号 8](#_Toc23258674)

[6.2.7 语句块的缩进：4空格 9](#_Toc23258675)

[6.2.8 行长度限制：100 9](#_Toc23258676)

[6.2.9 长行断行 9](#_Toc23258677)

[6.2.10 Kotlin中的特殊格式化 11](#_Toc23258680)

[6.2.11 空白空间 14](#_Toc23258681)

[6.2.12 分组括号 15](#_Toc23258682)

[6.2.13 特殊结构 15](#_Toc23258683)

[6.2.14 命名 18](#_Toc23258684)

[6.2.15 资源文件规范 25](#_Toc23258685)

[6.3 文档注释 (Javadoc) 29](#_Toc23258686)

[6.4 Android日志记录 33](#_Toc23258688)

[6.5 编程实践 33](#_Toc23258689)

[6.5.1 异常捕获规约 33](#_Toc23258690)

[6.5.2 线程使用规约 36](#_Toc23258691)

[6.5.3 常规代码规约 37](#_Toc23258692)

[6.5.4 Android 中的代码规约 41](#_Toc23258693)

[6.5.5 UI与布局规约 46](#_Toc23258694)

[6.5.6 Kotlin特性的惯用法 48](#_Toc23258695)

[附: 参考资料 51](#_Toc23258696)

# 1 目的

为了有利于项目维护、增强代码可读性、提升 Code Review 效率以及规范团队安卓开发，确保每个安卓APP软件开发人员必须一致遵守开发规范，特制定本规范。

# 2 范围

本规定适用于指导基于Android系统的设备上的上层APP开发过程中的规范，以开发者为中心视角，分为：IDE使用规范，编码规范，文档注释规范，Android日志记录，编程实践等部分。包含Java、Kotlin、xml、gradle、proguard-rules.pro等与Android APP开发密切相关的内容。

根据语言描述，除包含“建议”、“尽量”词汇的条目外，其他都为必须遵守项，违反本约定或将会引起各种后果或不便。“推荐”和“尽量”修饰的条目长期遵守有助于系统稳定性和合作效率的提升。

# 3 职责

3.1 安卓APP开发工程师：按照本规范执行APP开发。

3.2 评审专家：负责进行代码规范评审。

# 4 定义

## 4.1 术语、缩略语

| **术语、缩略语** | **说明** |
| --- | --- |
| IDE | 集成开发环境Integrated Development Environment的缩写 |
| AS | Google推出的一个Android集成开发工具，全称Android Studio |
| Kotlin | 一种兼容Java的语言，目前是Android官方支持的一级开发语言 |
| gradle | Android Studio附带的自动化构建工具 |

## 4.2 其他说明

1. 因为Java与Kotlin 100%互通，在代码规范与编程实践中，未强调Java或Kotlin时为Java独有或Java与Kotlin通用，强调后为对应语言特定遵守内容。
2. 在Android Studio中，构建的项目为Project，项目中的组件为Module。

# 5 流程图

无

# 6 内容

## 6.1 IDE使用规范

遵循如下规则：

1. APP开发IDE使用Android Studio，尽量使用最新的稳定版进行开发；AS项目自动构建工具使用gradle，禁止使用maven。
2. 当AS需要进行大版本更新时不进行直接更新，需要下载完整新版本AS进行验证或询问部门其他人是否验证过后再进行更新，小版本可直接进行更新。大版本指AS版本号第二位，比如3.4->3.5，小版本号指AS版本号第三位，如3.5.1。
3. 保持AS中gradle插件为AS对应版本最新版本，Kotlin插件为最新版本。及时删除gralde中未使用到的依赖库和各种定义。
4. 编辑完 .java、.xml、.kt等文件后必须使用AS自带 **格式化 功能进行格式化，格式化模板**使用 AS 默认模板即可。
5. 删除多余的import，编码过程中根据IDE的提示尽量减少警告出现。

### 6.1.1 版本统一规范

在项目根目录下的 build.gradle 文件中添加如下内容:

ext {  
 minSdkVersion = 19  
 targetSdkVersion = 28  
 compileSdkVersion = 28  
 supportVersion = '28.0.0'  
 anko\_version = '0.10.4'  
 recyclerViewUtilsVersion = '3.2'  
 minifyEnabled = true  
  
 sign = [  
 keyAlias : "xxx",  
 keyPassword : "xxx",  
 storePassword : "xxx",  
 storeFile : "../../\_storeFile/xx",  
 systemSignFile : "../../\_storeFile/xxx",  
 serverPath : "../../apk/"  
 ]  
}

在Module的build.gradle中使用如下方式进行引用:

compileSdkVersion rootProject.ext.compileSdkVersion  
  
defaultConfig {  
 minSdkVersion rootProject.ext.minSdkVersion  
 targetSdkVersion rootProject.ext.targetSdkVersion  
}

signingConfigs {  
 debug {  
 keyAlias sign.keyAlias  
 keyPassword sign.keyPassword  
 storeFile file(sign.systemSignFile)  
 storePassword sign.storePassword  
 }  
 release {  
 keyAlias sign.keyAlias  
 keyPassword sign.keyPassword  
 storeFile file(sign.systemSignFile)  
 storePassword sign.storePassword  
 }  
}  
buildTypes {  
 debug {  
 minifyEnabled false  
 signingConfig signingConfigs.debug  
 }  
 release {  
 minifyEnabled rootProject.ext.minifyEnabled  
 shrinkResources rootProject.ext.minifyEnabled  
 proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'  
 }  
}

implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin\_version"

implementation "com.android.support:appcompat-v7:$supportVersion"

implementation "com.github.NessCurie:RecyclerViewUtils:$recyclerViewUtilsVersion"

gradle插件与类库的添加遵循:

1. 添加插件时添加到gradle文件顶部时，每个插件之间不添加空行。

例如:

apply plugin: 'com.android.application'  
apply plugin: 'kotlin-android'  
apply plugin: 'kotlin-android-extensions'

1. 添加类库依赖时， 将出现的freeTree与project以及android和Kotlin的依赖放置到最顶部，第三方类库建议将同一系列类库放到一起。

例如：

dependencies {  
 implementation fileTree(include: ['\*.jar'], dir: 'libs')  
 implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin\_version"  
 implementation project(':xxx)  
 implementation "com.github.NessCurie:RecyclerViewUtils:$recyclerViewUtilsVersion"  
 implementation 'com.github.bumptech.glide:glide:4.9.0'  
 implementation 'com.github.chrisbanes:PhotoView:2.0.0'  
 implementation 'com.facebook.fresco:fresco:2.0.0'  
 implementation 'com.facebook.fresco:animated-gif:2.0.0'  
 implementation 'com.facebook.fresco:animated-webp:2.0.0'  
 implementation 'com.facebook.fresco:webpsupport:2.0.0'  
}

注意:

1. 其中除特殊情况外禁止出现: buildToolsVersion，禁止使用过时的gradle api。如：compile。
2. 对于可能在多个module中出现的依赖和一些定义比如混淆开关也在根目录下的gradle文件中添加对应定义后在module中进行引用以方便统一版本更新。
3. 完整的Project根目录gradle文件参考附件:。
4. 完整的Module下的gradle文件参考附件:。

### 6.1.2 Module的命名和版本

对于library类型的Module：

library类型的Module命名应体现出该类库的功能，一般以lib开头或结尾。

例如：LibVLC，BluetoothLib。

对于application类型的Module：

可在Module的gradle文件中的android节点下添加如下内容以确保Module与生成的APK文件名称对应:

android.applicationVariants.all { variant ->  
 if ("release" == variant.buildType.name) {  
 assembleRelease.doLast {  
 variant.outputs.all { output ->  
 delete output.outputFile.parent + "/output.json"  
 copy {  
 from output.outputFile.parent  
 into output.outputFile.parentFile.parent  
 rename { filename ->  
 filename.replace("-release", "")  
 }  
 }  
 delete output.outputFile.parent  
 }  
 }  
 }  
}

Module版本的versionName采用四位小数点进行表示，如：versionName "2.6.1.0"。

任意高版本号升级，低版本号归0，如：1.2.3.4升级为 1.3.0.0。

1. 经过测试并发布过，重新发布必须升级第二位，视情况考虑修改第一位（例如较大界面变动、功能变动或代码重构）。
2. 经过测试但未发布，重新修改测试，视改动大小，升级第二/三/四位。
3. versionCode和versionName 同步升级，versionName 做过任何变动，versionCode增加１，避免单方面改动。
4. 软件更改履历中注明版本号“versionCode”和“versionName”。

### 6.1.3 动态权限处理

Android系统从6.0开始一些特殊权限需要在应用中动态获取，否则会对应用功能造成影响。对动态权限的获取，开发手机上的APP时正常在应用中进行获取，开发公司Android设备上的应用时采用下列规则:

1. 除特殊情况外不采用将targetSdkVersion设置为19以获取到权限， 设置为targetSdkVersion为19会引发一些复杂问题。
2. 尽量减少需要使用动态权限的情况。
3. 当必须使用到动态权限时，采用在AndroidManifest.xml中配置android:sharedUserId="android.uid.system"并添加系统签名进行签名以获取权限；

### 6.1.4 代码混淆

代码混淆需要遵守以下规则:

1. 手机上的app进行发布时，必须进行代码混淆。初步测试期可不进行混淆以便问题定位，最后通过测试后必须进行混淆后再次测试。
2. 在公司android屏设备上的app尽量在发布前开启混淆。
3. 目前不推荐开启IDE混淆优化，防止出现复杂问题。

混淆文件和使用参考: 。

## 6.2 编码规范

### 6.2.1 文件名

Java源码文件名由它所包含的顶级class的类名（大小写敏感），加上.java后缀组成。Java每个源码文件中只能有一个顶级class。

Kotlin 源文件应当与 Java 源文件位于同一源文件根目录下，并遵循相同的目录结构，每个文件应存储在与其 package 语句对应的目录中。

Kotlin中鼓励多个声明（类、顶级函数或者属性）放在同一个 Kotlin 源文件中， 只要这些声明在语义上彼此紧密关联并且文件保持合理大小 （不超过几百行）。特别是在为类定义与类的所有客户都相关的扩展函数时，请将它们放在与类自身定义相同的地方。而在定义仅对指定客户有意义的扩展函数时，请将它们放在紧挨该客户代码之后。例如不要只是为了保存 “Foo 的所有扩展函数”而创建文件。

如果 Kotlin 文件包含单个类（以及可能相关的顶层声明），那么文件名应该与该类的名称相同，并追加 .kt 扩展名。

如果文件包含多个类或只包含顶层声明， 那么选择一个描述该文件所包含内容的名称，并以此命名该文件。 使用首字母大写的驼峰风格（例如 ProcessDeclarations.kt）。文件的名称应该描述文件中代码的作用。

### 6.2.2 文件编码：UTF-8

源码文件使用UTF-8编码。

IDE 中文件的换行符使用 Unix 格式，不要使用 Windows 格式。

### 6.2.3 特殊字符

#### 6.2.3.1空格字符

除了换行符外，ASCII水平空白字符（0x20）是源码文件中唯一支持的空格字符。

这意味着：

1. 其他空白字符将被转义；
2. Tab字符不被用作缩进控制。

#### 6.2.3.2 特殊转义字符串

任何需要转义字符串表示的字符（例如\b， \t， \n， \f， \r， \'， \\等），采用这种转义字符串的方式表示，而不采用对应字符的八进制数（例如 \012）或Unicode码（例如 \u000a）表示。

#### 6.2.3.3 非ASCII字符

对于其余非ASCII字符，直接使用Unicode字符（例如 ∞），或者使用对应的Unicode码（例如 \u221e）转义，都是允许的。唯一需要考虑的是，何种方式更能使代码容易阅读和理解。在使用unicode码转义，或者甚至是有时直接使用unicode字符的时候，添加一点说明注释对读懂代码很有帮助。

### 6.2.4 源码文件结构

源码文件按照先后顺序，由以下几部分组成：

1. License或者copyright声明信息。（如果需要声明）；
2. 包声明语句；
3. import语句；
4. class类声明。

注意:

1. 如果需要声明lincense或copyright信息，应该在文件开始时声明。 其后是 package 和 import 语句， 最后是类或接口声明。
2. 每个部分之间应该只有一行空行作为间隔。
3. import语句的行、包声明的行，没有行长度的限制。单行长度限制不适用于import语句所在行和包声明。

### 6.2.5 类成员顺序

类成员的顺序对代码的易读性有很大影响：

1. 不要按字母顺序或者可见性对方法声明排序，也不要将常规方法与扩展方法分开，而是要把相关的东西放在一起，这样从上到下阅读类的人就能够跟进所发生事情的逻辑。
2. 将嵌套类放在紧挨使用这些类的代码之后。如果打算在外部使用嵌套类，而且类中并没有引用这些类，那么把它们放到顶部或末尾，Kotlin中放在伴生对象之后。
3. 在实现一个接口时，实现成员的顺序应该与该接口的成员顺序相同（如果需要， 还要插入用于实现的额外的私有方法）。
4. 如果类继承于 Android 组件（例如 Activity 或 Fragment），需要把重写函数按照他们的生命周期进行排序。
5. 重载方法不应该分开。
6. 当一个类有多个构造函数，或者多个同名成员方法时，这些函数应该写在一起，不应该被其他成员分开。

**Java推荐使用如下排序：**

常量

字段

构造函数

重写函数和回调

公有函数

私有函数

内部类或接口

字段可以定义在类的顶部或者在使用它们的方法之前定义字段，内部类或接口也可以在使用它们的方法之前定义。

函数顺序需要根据实际逻辑进行调整，遵守把相关的东西放在一起的规则。

**Kotlin中，一个类的内容按以下顺序排列：**

属性声明与初始化块

次构造函数

方法声明

伴生对象

当伴生对象只用于定义一些常量和伴生变量时放置到类顶部。

### 6.2.6 花括号

花括号一般用在if， else， for， do， 和 while等语句。

对于非空语句块，花括号遵循K&R风格：

1. 左括号前不换行；
2. 左括号后换行；
3. 右括号前换行；
4. 如果右括号结束一个语句块或者函数体、构造函数体或者有命名的类体，则需要换行。例如，当右括号后面接else或者逗号时，不应该换行。

例如:

return () -> {  
 while (condition()) {  
 method();  
 }  
};  
  
return new MyClass() {  
 @Override public void method() **{** if (condition()) {  
 try {  
 something();  
 } catch (ProblemException e) {  
 recover();  
 }  
 } else if (otherCondition()) {  
 somethingElse();  
 } else {  
 lastThing();  
 }  
 **}**};

**注意:**

1. 如果整个条件语句（条件和主体）适合放在同一行，那么可以（但不是必须）将其全部放在一行上。

接受以下样式:

if (condition) {  
 body();  
}

同样也接受以下样式：

if (condition) body();

但不接受以下样式：

if (condition)  
 body(); // bad!

1. 一个空的语句块，可以在左花括号之后直接接右花括号，中间不需要空格或换行。

例如：void doNothing() {}

但是当一个由几个语句块联合组成的语句块时，则需要换行。

例如：if/else，try/catch/finally

### 6.2.7 语句块的缩进：4空格

1. 每当一个新的语句块产生，缩进就增加4个空格，当这个语句块结束时，缩进恢复到上一层级的缩进格数。
2. 缩进要求对整个语句块中的代码和注释都适用。

### 6.2.8 行长度限制：100

一行最多只有一句代码，每句代码的结束都需要换行。

代码中每一行文本的长度都应该不超过 100 个字符（AS 窗口右侧的竖线）。

除了以下几个特殊情况外，其他代码内容都需要遵守这个长度限制：

1. 如果备注行包含长度超过 100 个字符的示例命令或文字网址，那么为了便于剪切和粘贴，该行可以超过 100 个字符。
2. 导入语句行和包声明行可以超出此限制。
3. 注释中的命令行指令行，将被直接复制到shell中执行的。
4. 按照行长度限制，无法实现地方（例如：Javadoc中超长的URL地址， 或者一个超长的JSNI方法的引用）。

### 6.2.9 长行断行

当一行代码按照其他规范都合法，只是为了避免超出行长度限制而换行时，称为长行断行。

注意：将长行封装为函数，或者使用局部变量的方法，也可以解决一些超出行长度限制的情况，并非一定要断行。

断行的主要原则是：选择在更高一级的语法逻辑的地方断行。

其他一些原则如下：

1. 当一个非赋值运算的语句断行时，在运算符号之前断行。
2. 当一个赋值运算语句断行时，一般在赋值符号之后断行，但是也可以在之前断行。
3. 在调用函数或者构造函数需要断行时，与函数名相连的左括号要在一行，也就是在左括号之后断行。
4. 逗号断行时，要和逗号隔开的前面的语句断行，也就是在逗号之后断行。

使用八 (8) 个空格来缩进自动换行，包括函数调用和赋值：

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

错误示例：

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

#### 6.2.9.1 操作符的换行

除赋值操作符之外，把换行符放在操作符之前。

例如：

int longName = anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne  
 + theFinalOne;

赋值操作符的换行放在其后。

例如：

int longName =  
 anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne + theFinalOne;

#### 6.2.9.2 函数链的换行

当同一行中调用多个函数时（比如使用构建器时），对每个函数的调用应该在新的一行中，把换行符插入在 . 之前。

例如：

Picasso.with(context).load("http://xxx.com/xxx/xxx.jpg").into(iv);

添加换行后如下：

Picasso.with(context)  
 .load("http://xxx.com/xxx/xxx.jpg")  
 .into(iv);

#### 6.2.9.3 多参数的换行

当一个方法有很多参数或者参数很长的时候，应该在每个　，后面进行换行。

例如：

loadPicture(context, "http://xxx.com/xxx/xxx.jpg", iv, "xxxxxxxxxx", clickListener);

应该使用如下规则：

loadPicture(context,  
 "http://xxx.com/xxx/xxx.jpg",  
 iv,  
 "xxxxxxxxxx",  
 clickListener);

#### 6.2.9.4 RxJava链式的换行

RxJava 的每个操作符都需要换新行，并且把换行符插入在 . 之前。

例如：

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;  
 }  
 });  
}

### 6.2.10 Kotlin中的特殊格式化

#### 6.2.10.1 类头格式化

具有少数主构造函数参数的类可以写成一行：

class Person(id: Int, name: String)

具有较长类头的类应该格式化，以使每个主构造函数参数都在带有缩进的独立的行中。另外，右括号应该位于一个新行上。如果使用了继承，那么超类的构造函数调用或者所实现接口的列表应该与右括号位于同一行：

class Person(  
 id: Int,  
 name: String,  
 surname: String  
) : Human(id, name) { /\*……\*/ }

对于多个接口，应该将超类构造函数调用放在首位，然后将每个接口应放在不同的行中：

class Person(  
 id: Int,  
 name: String,  
 surname: String  
) : Human(id, name),  
 KotlinMaker { /\*……\*/ }

对于具有很长超类型列表的类，在冒号后面换行，并横向对齐所有超类型名：

class MyFavouriteVeryLongClassHolder :  
 MyLongHolder<MyFavouriteVeryLongClass>(),  
 SomeOtherInterface,  
 AndAnotherOne {  
  
 fun foo() { /\*...\*/ }  
}

#### 6.2.10.2 函数格式化

如果函数签名不适合单行，请使用以下语法：

fun longMethodName(  
 argument: ArgumentType = defaultValue,  
 argument2: AnotherArgumentType  
): ReturnType {  
 // 函数体  
}

对于由单个表达式构成的函数体，优先使用表达式形式：

fun foo(): Int { // 不良  
 return 1  
}

fun foo() = 1 // 良好

#### 6.2.10.3 表达式函数体格式化

如果函数的表达式函数体与函数声明不适合放在同一行，那么将 = 留在第一行，将表达式函数体缩进 8 个空格：

fun f(x: String) =   
 x.length

#### 6.2.10.4 属性格式化

对于非常简单的只读属性，使用单行格式：

val isEmpty: Boolean get() = size == 0

对于更复杂的属性，总是将 get 与 set 关键字放在不同的行上：

val *foo*: String  
 get() { /\*……\*/ }

#### 6.2.10.5 格式化控制流语句

在 when 语句中，如果一个分支不止一行，用空行将其与相邻的分支块分开：

private fun parsePropertyValue(propName: String, token: Token) {  
 when (token) {  
 is Token.ValueToken ->  
 callback.visitValue(propName, token.*value*)  
  
 Token.*LBRACE* -> { // ……  
 }  
 }  
}

将短分支放在与条件相同的行上，无需花括号：

when (foo) {  
 true -> bar() // 良好  
 false -> **{** baz() **}** // 不良  
}

#### 6.2.10.6 方法调用格式化

在较长参数列表的左括号后添加一个换行符，按 4 个空格缩进参数，将密切相关的多个参数分在同一行：

drawSquare(  
 x = 10, y = 10,  
 width = 100, height = 100,  
 fill = true  
)

#### 6.2.10.7 Lambda 表达式格式化

如果一个调用接受单个 Lambda 表达式，应该尽可能将其放在圆括号外边传入。

如果为 Lambda 表达式分配一个标签，那么不要在该标签与左花括号之间留空格：

fun foo() {  
 ints.forEach lit@**{** // ……  
 **}**}

在多行的 Lambda 表达式中声明参数名时，将参数名放在第一行，后跟箭头与换行符：

appendCommaSeparated(properties) { prop ->  
 val propertyValue = prop.get(obj) // ……  
}

### 6.2.11 空白空间

#### 6.2.11.1垂直空白

单行空行在以下情况使用：

1. 类成员间需要空行隔开。

例如：成员变量、构造函数、成员函数、内部类、静态初始化语句块（static initializers）、实例初始化语句块（instance initializers）。

例外：成员变量之间的空白行不是必需的，一般多个成员变量中间的空行，是为了对成员变量做逻辑上的分组。

1. 在函数内部，根据代码逻辑分组的需要，设置空白行作为间隔。
2. 类的第一个成员之前，最后一个成员结束之后，用空行间隔。

#### 6.2.11.2 水平空白

除了语法、其他规则、词语分隔、注释外，水平的ASCII空格只在以下情况出现：

1. 所有保留的关键字与紧接它之后的位于同一行的左括号之间需要用空格隔开。

例如：if、for、catch 和Kotlin的when。

1. 所有保留的关键字与在它之前的右花括号之间需要空格隔开。

例如：else、catch。

1. 在左花括号之前都需要空格隔开。只有两种嵌套时的例外：

@SomeAnnotation({a, b}) String[][] x = {{"foo"}};

1. 所有的二元运算符和三元运算符的两边，都需要空格隔开。如:（a + b）.

例外：Kotlin不要在“range to”操作符（0..i）左右留空格

1. 逗号、冒号、分号和右括号之后，需要空格隔开。
2. // 双斜线开始一行注释时。双斜线两边都应该用空格隔开，但不作为强制要求。
3. 变量声明时，变量类型和变量名之间需要用空格隔开。
4. 初始化一个数组时，花括号之间可以用空格隔开，也可以不使用。

例如：new int[] {5, 6}和 new int[] { 5, 6 }都可以。

Kotlin的冒号水平空白需要额外注意：在 : 之后总要留一个空格，以下场景中的 : 之前留一个空格：

1. 当它用于分隔类型与超类型时，而当分隔声明与其类型时不要在 : 之前留空格。
2. 当委托给一个超类的构造函数或者同一类的另一个构造函数时。
3. 在 object 关键字之后。
4. 在 Lambda 表达式中，应该在花括号左右以及分隔参数与代码体的箭头左右留空格。

例如：

abstract class Foo<out T : Any> : IFoo {  
 abstract fun foo(a: Int): T  
}  
  
class FooImpl : Foo() {  
 constructor(x: String) : this(x) { /\*……\*/ }  
  
 val x = object : IFoo { /\*……\*/ }  
}

注意下面不留空格的情况:

1. 绝不在 (、 [ 之后或者 ]、 ) 之前留空格。
2. 绝不在. 或者 ?. 左右留空格。

例如：foo.bar().filter { it > 2 }.joinToString(), foo?.bar()

1. 不要在用于指定类型参数的尖括号前后留空格。

例如：class Map<K, V> { …… }

1. 不要在 :: 前后留空格。

例如：Foo::class、 String::length

1. 不要在用于标记可空类型的 ? 前留空格。

例如：String?

1. 不要在一元运算符左右留空格。

例如：a++

1. Kotlin不要在主构造函数声明、方法声明或者方法调用的左括号之前留空格。

例如: class A(val x: Int) fun foo(x: Int) { …… } foo(1)

### 6.2.12 分组括号

非必须的分组括号只有在编写代码者和代码审核者都认为大家不会因为没有它而导致代码理解错误的时候，或者它不会使代码更易理解的时候才能省略。因为没有理由认为所有阅读代码的人都能记住所有运算符的优先级。

### 6.2.13 特殊结构

#### 6.2.13.1 枚举类型

如果没有函数和Javadoc，处理格式可以按照数组初始化来处理：每个逗号后接一个枚举变量，不要求换行。

例如：private enum Suit { *CLUBS*, *HEARTS*, *SPADES*, *DIAMONDS* }

枚举类型也是一种类（class），因此类的其他格式要求，也适用于枚举类型。

#### 6.2.13.2 变量声明

每次声明一个变量，不要采用一个声明声明多个变量。

例如：不允许 int a, b;

#### 6.2.13.3 数组

不能像C风格一样声明数组，方括号应该是变量类型的一部分，因此不应该和变量名放在一起。

例如：应该是String[] args，而不是 String args[].

数组的初始化，可以采用和块代码相同的格式处理，以下格式都是允许的：

new int[]{  
 0, 1, 2, 3  
};

new int[]  
 {0, 1, 2, 3};

#### 6.2.13.4 switch语句块

每个case之后紧接的非标签新行，按照花括号相同的处理方式缩进4个字符，在标签结束后，恢复到之前的缩进，类似花括号结束。

每个标签对应的代码执行完后，都应该通过语句结束（例如：break、continue、return 或抛出异常），否则应该通过注释说明，代码需要继续向下执行下一个标签的代码。这个注释在最后一个标签之后不需要注释。

每个switch语句中，都需要显式声明default标签，即使没有任何代码也需要显示声明。

例如：

switch (input) {  
 case 1:  
 case 2:  
 prepareOneOrTwo();  
 // fall through  
 case 3:  
 handleOneTwoOrThree();  
 break;  
 default:  
 handleLargeNumber(input);  
}

#### 6.2.13.5 注解

注解应用到类、函数或者构造函数时，应紧接Javadoc之后，放在修饰符前。

注解通常放在单独的行上，在它们所依附的声明之前，并使用相同的缩进，每一行只有一个。注解所在行不受行长度限制，也不需要增加缩进。

例如：

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

@Named("Foo")  
private val foo: Foo

应用到成员变量时，也是紧接Javadoc之后，不同的是，多个注解可以放在同一行。

例如：@Partial @Mock DataLoader loader;

Kotlin文件注解位于文件注释（如果有的话）之后、package 语句之前，并且用一个空白行与 package 分开（为了强调其针对文件而不是包）。

例如:

*/\*\* 授权许可、版权以及任何其他内容 \*/*@file:JvmName("FooBar")  
  
package foo.bar

例外情况：

1. 如果注解只有一个，并且不带参数，则它可以和类或方法名放在同一行。

例如：@Override public int hashCode() { ... }

1. Kotlin无参数的注解可以放在同一行。

@JsonExclude @JvmField  
var x: String

#### 6.2.13.6 注释

注释的缩进与它所注释的代码缩进相同。

可以采用 /\* \*/ 进行注释，也可以用 // 进行注释。

当使用 /\*\*/ 进行多行注释时，每一行都应该以 \* 开始， 并且 \* 应该上下对齐。

例如：

/\*  
 \* This is // And so /\* Or you can  
 \* okay. // is this. \* even do this. \*/  
 \*/

#### 6.2.13.7 修饰符

Java多个类和成员变量的修饰符，按Java Lauguage Specification中介绍的先后顺序排序，具体是：

public / protected / private

abstract

static

final

transient

volatile

synchronized

native

strictfp

Kotlin如果一个声明有多个修饰符，请始终按照以下顺序安放

public / protected / private / internal

expect / actual

final / open / abstract / sealed / const

external

override

lateinit

tailrec

vararg

suspend

inner

enum / annotation

companion

inline

infix

operator

data

注意: 接口中的方法和属性不要加任何修饰符号，保持代码的简洁性。

#### 6.2.13.8 在接口中定义常量的情况

1. 如果需要在接口中定义常量，必须是与接口方法相关，并去掉其修饰符。
2. 当一个类中定义的字段全部为公开的常量(被public static final修饰)时，为了使代码简洁，将这个类转化为接口进行表示，但必须去掉所有修饰符，并为每个常量添加Javadoc。
3. 不要使用一个常量类维护所有常量，要按常量功能进行归类，分开维护。
4. 注意常量类所放置的位置，需要根据常量的复用层次放置在合理的位置。

跨应用共享常量需要放置在公共库中，应用内共享常量需要放置在该应用对应目录中， 子模块内部共享常量在该子模块下，只在某个类内或少量几个类中使用不要放到常量类中，直接在类中定义。

### 6.2.14 命名

#### 6.2.14.1 标识符的通用规范

1. 标识符只应该使用ASCII字母、数字和下划线，字母大小写敏感。因此所有的标识符，都应该能匹配正则表达式 \w+ 。
2. 代码中的命名均不能以下划线或美元符号开始，也不能以下划线或美元符号结束。

反例：\_name / \_\_name / $name / name\_ / name$ / name\_

1. 代码中的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式。

注意：即使纯拼音命名方式也要避免采用。但 Beijing，Hangzhou 等国际通用的名称可视同英文。

#### 6.2.14.2 Camel case的定义

一些短语被写成Camel case的时候可以有多种写法，例如一些缩写词汇，或者一些组合词：IPv6 或者 iOS 等。

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， 杜绝完全不规范的缩写。当使用缩写时应遵循以下规则：

1. 将字符全部转换为ASCII字符，并且去掉 ' 等符号。

例如："Müller's algorithm" 被转换为 "Muellers algorithm" 。

1. 将上一步转换的结果拆分成一个一个的词语，从空格处和从其他剩下的标点符号处划分。

注意：一些已经是Camel case的词语，也应该在这个时候被拆分。（例如 AdWords 被拆分为 ad words）。但是例如iOS之类的词语，它其实不是一个Camel case的词语，而是人们惯例使用的一个词语，因此不用做拆分。

1. 经过上面两部后，先将所有的字母转换为小写，再把每个词语的第一个字母转换为大写。
2. 最后，将所有词语连在一起，形成一个标示符。

注意：

1. 词语原来的大小写规则，应该被完全忽略。
2. 有些词语在英文中，可以用 - 连接使用，也可以不使用 - 直接使用。例如 “nonempty”和 “non-empty”都是可以的。因此，方法名字为checkNonempty 或者checkNonEmpty 都是可以的。
3. JDK 和 Android 代码库在首字母缩写词上非常不一致，几乎也不可能与周围的代码保持一致。因此，请务必将首字母缩写词视为字词，使名称更具可读性

例：

| 良好 | 不佳 |
| --- | --- |
| XmlHttpRequest | XMLHTTPRequest |
| getCustomerId | getCustomerID |
| class Html | class HTML |
| String url | String URL |
| long id | long ID |

#### 6.2.14.3 包名

包名全部使用小写字母，通过 . 将各级连在一起，连续的单词只是简单地连接起来，不使用下划线。

名称采用反域名命名规则，一级包名是顶级域名，通常为 com、edu、gov、net、org 等，二级包名为公司名，三级包名根据应用进行命名，后面是对包名的划分。

例如：com.xxx.launcher

#### 6.2.14.4 类名

类的命名，采用以大写字母开头的大小写字符间隔的方式（UpperCamelCase）

1. 如果模块、接口、类、方法使用了设计模式，在命名时需体现出具体模式。

例如: OrderFactory LoginProxy ResourceObserver

1. 对于 Service 和 DAO 类，基于 SOA 的理念，暴露出来的服务一定是接口，内部的实现类用 Impl 的后缀与接口区别。

例如: CacheServiceImpl 实现 CacheService 接口

1. 如果是形容能力的接口名称，取对应的形容词为接口名（通常是–able 的形式）。

例如: AbstractTranslator 实现 Translatable 接口。

1. 异常类命名使用 Exception 结尾。
2. 测试类的命名，应该以它所测试的类的名字为开头，并在最后加上Test结尾。

例如：HashTest 、 HashIntegrationTest。

1. Android中特殊的类比如四大组件，需要以对应后缀结尾。

Android中常用类名举例：

| **类** | **描述** | **例如** |
| --- | --- | --- |
| Activity | Activity为后缀标识 | WelcomeActivity |
| Adapter | Adapter为后缀标识 | NewsDetailAdapter |
| 解析类 | Parser为后缀标识 | HomePosterParser |
| 工具方法类 | Utils或Manager为后缀标识 | 线程池管理类：ThreadPoolManager 日志工具类：LogUtils 打印工具类：PrinterUtils |
| 数据库类 | 以DBHelper后缀标识 | 新闻数据库：NewsDBHelper |
| Service | 以Service为后缀标识 | 时间服务 TimeService |
| BroadcastReceiver | 以Receiver为后缀标识 | 推送接收 JPushReceiver |
| ContentProvider | 以Provider为后缀标识 | ShareProvider |
| 自定义的共享基础类 | 以Base开头 | BaseActivity,BaseFragment |

#### 6.2.14.5 方法名

方法命名采用以小写字母开头的大小写字符间隔的方式（lowerCamelCase）。一般使用动词或者动词短语。

在JUnit的测试方法中，可以使用下划线，用来区分测试逻辑的名字，经常使用如下的结构：test<MethodUnderTest>\_<state> 。例如：testPop\_emptyStack

Android中常用方法名举例：

| **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/check 为前缀标识 |
| handleXX(), processXX() | 对数据进行处理的方法 |
| displayXX(), showXX() | 弹出提示框和提示信息，使用 display/show 为前缀标识 |
| updateXX() | 更新数据 |
| saveXX(), insertXX() | 保存或插入数据 |
| resetXX() | 重置数据 |
| clearXX() | 清除数据 |
| removeXX(), deleteXX() | 移除数据或者视图等，如 removeView() |
| drawXX() | 绘制数据或效果相关的，使用 draw 前缀标识 |

###### 工厂函数命名

如果为一个类声明一个工厂函数，那么不要让它与类自身同名。优先使用独特的名称，该名称能表明为何该工厂函数的行为与众不同。只有当确实没有特殊的语义时，才可以使用与该类相同的名称。

例如：

class Point(val x: Double, val y: Double) {  
 companion object {  
 fun fromPolar(angle: Double, radius: Double) = Point(...)  
 }  
}

#### 6.2.14.6 常量名

常量命名全部使用大写字符，词与词之间用下划线隔开。（CONSTANCE\_CASE）。常量一般使用名词或者名词短语命名。

不允许任何魔法值（即未经预先定义的常量）直接出现在代码中。

Java中常量都是一个 static final 字段，但不是所有 static final 字段都是常量。在决定一个字段是否是一个常量时，得考虑它是否真的感觉像是一个常量。例如，如果观测任何一个该实例的状态是可变的，则它几乎肯定不会是一个常量。只是永远不打算改变的对象一般是不够的，它要真的一直不变才能将它示为常量。下面是常量和非常量的例子：

**常量：**  
static final int *NUMBER* = 5;  
static final ImmutableList<String> *NAMES* = ImmutableList.*of*("Ed", "Ann");  
static final ImmutableMap<String, Integer> *AGES* =

ImmutableMap.*of*("Ed", 35, "Ann", 32);  
static final Joiner *COMMA\_JOINER* = Joiner.on(','); // because Joiner is immutable  
static final SomeMutableType[] *EMPTY\_ARRAY* = {};  
enum SomeEnum { *ENUM\_CONSTANT* }  
  
**非常量：**static String *nonFinal* = "non-final";  
final String nonStatic = "non-static";  
static final Set<String> *mutableCollection* = new HashSet<String>();  
static final ImmutableSet<SomeMutableType> *mutableElements* = ImmutableSet.*of*(mutable);  
static final ImmutableMap<String, SomeMutableType> *mutableValues* =  
 ImmutableMap.*of*("Ed", mutableInstance, "Ann", mutableInstance2);  
static final Logger *logger* = Logger.*getLogger*(MyClass.getName());  
static final String[] *nonEmptyArray* = {"these", "can", "change"};

Kotlin中常量是指标有关键字 const 的属性，或者保存不可变数据的没有自定义 get 函数的顶层/对象 val 属性。

保存带有行为的对象或者可变数据的顶层/对象属性的名称应该使用变量风格名称。

保存单例对象引用的属性的名称可以使用与 object 声明相同的命名风格。

例如：

const val MAX\_COUNT = 8  
val USER\_NAME\_FIELD = "UserName"  
val mutableCollection: MutableSet<String> = HashSet()  
val PersonComparator: Comparator<Person> =

Android SDK 中的很多类都用到了键值对函数，比如 SharedPreferences、Bundle、Intent. 当时用到这些类的时候，必须将它们的键定义为 static final 字段，并遵循以下指示作为前缀：

| **类** | **字段名前缀** |
| --- | --- |
| SharedPreferences | PREF\_ / SP\_ |
| Bundle | BUNDLE\_ |
| Fragment Arguments | ARGUMENT\_ / ARG\_ |
| Intent Extra | EXTRA\_ |
| Intent Action | ACTION\_ |

说明：虽然 Fragment.getArguments() 得到的也是 Bundle ，但因为这是 Bundle 的常用用法，所以特意为此定义一个不同的前缀。

例如：

// 注意：字段的值与名称相同以避免重复问题  
static final String *PREF\_EMAIL* = "PREF\_EMAIL";  
static final String *BUNDLE\_AGE* = "BUNDLE\_AGE";  
static final String *ARGUMENT\_USER\_ID* = "ARGUMENT\_USER\_ID";  
// 与意图相关的项使用完整的包名作为值的前缀  
static final String *EXTRA\_SURNAME* = "com.myapp.extras.EXTRA\_SURNAME";  
static final String *ACTION\_OPEN\_USER* = "com.myapp.action.ACTION\_OPEN\_USER";

#### 6.2.14.7 非常量的成员变量名

非常量的成员变量命名（包括静态变量和非静态变量），采用lowerCamelCase命名。一般使用名词或名词短语。还需要遵守以下规则：

1. 对于表示集合或者数组的非常量字段名，可以添加后缀来增强字段的可读性。

比如：集合添加如下后缀：List、Map、Set，数组添加如下后缀：Arr。

例如：mIvAvatarList、userArr、firstNameSet。

如果数据类型不确定的话，那么使用其复数形式来表示也可，例如 books。

1. Android 众多的 UI 控件，为避免控件和普通成员变量混淆以及更好地表达意思，所有用来表示控件的成员变量统一加上控件缩写作为前缀。

例如:rvBooks，具体前缀参考6.2.15.6的id命名。

例外：一般json解析库需要bean类各成员变量名称与json中的字段名称对应，可能由于json中字段的名称造成成员变量名不符合规范，当出现这种情况时，尽可能去修改json协议符合规范，无法修改json协议时不作命名上的强行要求，与json中的字段保持一致即可。

###### Android platform成员变量

Android platform中成员变量的命名具有特定的规则.若需要对Android platform代码进行修改，也必须遵守这些规则，但**不建议在app开发中使用这些规则，若使用这些规则则需要完整遵循，禁止一份代码中部分遵循但部分未寻准的情况出现。**

规则:

1. 非公开且非静态字段的名称以 m 开头。
2. 静态字段的名称以 s 开头。
3. 其他字段以小写字母开头。
4. 公开静态 final 字段（常量）为全部大写并用下划线连接。

例:

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;  
}

###### Kotlin中幕后属性的名称

如果一个类有两个概念上相同的属性，一个是公共 API 的一部分，另一个是实现细节，那么使用下划线作为私有属性名称的前缀：

例如：

class C {  
 private val \_elementList = *mutableListOf*<Element>()  
  
 val elementList: List<Element>  
 get() = \_elementList  
}

#### 6.2.14.8 参数名

参数命名采用lowerCamelCase命名，避免使用一个字符作为参数的命名方式。

###### 特殊的参数顺序

在 Android 开发过程中，当出现Context作为参数时，应将 Context 作为其第一个参数，当出现回调接口作为参数时，应将回调接口应该作为其最后一个参数。

例如：

// Context always goes first  
public User loadUser(Context context, int userId);  
  
// Callbacks always go last  
public void loadUserAsync(Context context,   
 int userId, UserCallback callback);

#### 6.2.14.9 局部变量名

1. 局部变量采用lowerCamelCase命名。它相对于其他类型的命名，可以采用更简短宽松的方式。
2. 尽量避免采用单个字母进行命名的情况，除了在循环体内使用的临时变量。
3. 即使局部变量是final、不可改变的，它也不能被认为是常量，也不应该采用常量的命名方式去命名。
4. 禁止在局部变量前添加m等前缀，即不要将Android platform成员变量规则混用到成员变量中。

#### 6.2.14.10 数据库相关命名

表名、字段名必须使用小写字母或数字，使用下划线进行分割，不允许出现任何大写字母。禁止出现数字开头，禁止两个下划线中间只出现数字。

表名不使用复数名词，禁用保留字，如： desc、range、match、delayed 等。

说明：Windows与Linux大小写敏感不同，避免大小写造成的意外。

### 6.2.15 Android资源文件规范

资源文件命名为全部小写，采用下划线命名法。

如果是三方库开发，其使用到的资源文件相关的名称都应该使用库名作为前缀，避免三方库资源和实际应用资源重名的冲突。

#### 6.2.15.1 动画资源文件

属性动画文件需要放在 res/animator/ 目录下；

视图动画文件需放在 res/anim/ 目录下。

命名规则：{模块名\_}逻辑名称。{} 中的内容为可选，逻辑名称可由多个单词加下划线组成。

例如：refresh\_progress.xml、market\_cart\_add.xml、market\_cart\_remove.xml。

如果是普通的补间动画或属性动画，可采用：动画类型\_方向 的命名方式。例如：

| **名称** | **说明** |
| --- | --- |
| 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 |

#### 6.2.15.2 图片资源文件

res/drawable/ 用于存放位图文件（.png、.9.png、.jpg、.gif）或编译为可绘制对象资源子类型的 XML 文件。

res/mipmap/ 用于存放不同密度的启动图标，并且只用于存放启动图标，其余图片资源文件都应该放到 res/drawable/ 目录下。

命名规则：类型{\_模块名}\_逻辑名称、类型{\_模块名}\_颜色。{} 中的内容为可选，类型 可以是可绘制对象资源类型，也可以是控件类型（具体见6.2.15.6中控件缩写表）；最后可加上后缀 \_small 表示小图，\_big 表示大图。例如：

| **名称** | **说明** |
| --- | --- |
| btn\_main\_about.png | 主页关于按键 类型\_模块名\_逻辑名称 |
| btn\_back.png | 返回按键 类型\_逻辑名称 |
| divider\_maket\_white.png | 商城白色分割线 类型\_模块名\_颜色 |
| ic\_edit.png | 编辑图标 类型\_逻辑名称 |
| bg\_main.png | 主页背景 类型\_逻辑名称 |
| btn\_red.png | 红色按键 类型\_颜色 |
| btn\_red\_big.png | 红色大按键 类型\_颜色 |
| ic\_head\_small.png | 小头像图标 类型\_逻辑名称 |
| bg\_input.png | 输入框背景 类型\_逻辑名称 |
| divider\_white.png | 白色分割线 类型\_颜色 |
| bg\_main\_head.png | 主页头部背景 类型\_模块名\_逻辑名称 |
| def\_search\_cell.png | 搜索页面默认单元图片 类型\_模块名\_逻辑名称 |
| ic\_more\_help.png | 更多帮助图标 类型\_逻辑名称 |
| divider\_list\_line.png | 列表分割线 类型\_逻辑名称 |
| sel\_search\_ok.xml | 搜索界面确认选择器 类型\_模块名\_逻辑名称 |
| shape\_music\_ring.xml | 音乐界面环形形状 类型\_模块名\_逻辑名称 |

如果有多种形态，如按钮选择器：sel\_btn\_xx.xml，采用如下命名：

| **名称** | **说明** |
| --- | --- |
| sel\_btn\_xx | 作用在 btn\_xx 上的 selector |
| btn\_xx\_normal | 默认状态效果 |
| btn\_xx\_pressed | state\_pressed 点击效果 |
| btn\_xx\_focused | state\_focused 聚焦效果 |
| btn\_xx\_disabled | state\_enabled 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 激活效果 |
| btn\_xx\_window\_focused | state\_window\_focused 窗口聚焦效果 |

#### 6.2.15.3 布局资源文件

命名规则：类型\_模块名、类型{\_模块名}\_逻辑名称。

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 的行 layout 以 module\_grid\_item 开头。

例如：

| **名称** | **说明** |
| --- | --- |
| activity\_main.xml | 主窗体 类型\_模块名 |
| activity\_main\_head.xml | 主窗体头部 类型\_模块名\_逻辑名称 |
| fragment\_music.xml | 音乐片段 类型\_模块名 |
| fragment\_music\_player.xml | 音乐片段的播放器 类型\_模块名\_逻辑名称 |
| dialog\_loading.xml | 加载对话框 类型\_逻辑名称 |
| ppw\_info.xml | 信息弹窗（PopupWindow） 类型\_逻辑名称 |
| item\_main\_song.xml | 主页歌曲列表项 类型\_模块名\_逻辑名称 |

#### 6.2.15.4 菜单资源文件

菜单相关的资源文件应放在（menu/）目录下。

命名规则：{模块名\_}逻辑名称

例如：main\_drawer.xml、navigation.xml。

#### 6.2.15.5 values 资源文件

values/ 资源文件下的文件都以 s 结尾，如 attrs.xml、colors.xml、dimens.xml。

可以将一个大的 xml 文件分割成多个小的文件，比如可以有多个 style 文件，如 styles.xml、styles\_home.xml、styles\_item\_details.xml、styles\_forms.xml。

###### colors.xml

<color> 的 name 命名一般使用小写下划线命名法，但官方IDE自动生成的颜色定义是采用lowerCamelCase方式，所以lowerCamelCase也可以接受。命名有2种方式:

1. 在colors.xml中根据使用处定义颜色值.

缺点是可能容易定义重复的值，优点是这种定义具有较高的活动性，当需要对某些控件进行统一变色时直接调整其颜色值即可。

1. 在color.xml中根据颜色分类进行定义.

可以防止颜色的重复，但是当需要调整界面中一部分控件的颜色值时，若要保证颜色值与定义的名称对应需要到每处使用处进行修改使用的颜色值名称。

建议根据实际情况对颜色定义进行调整，并应尽量防止出现重复定义。

###### dimens.xml

<dimen>的name使用小写下划线命名法.和color.xml一样，不同的定义方式有不同的优缺点，建议根据实际情况进行命名。

###### strings.xml

<string> 的 name 命名使用下划线命名法。

###### styles.xml

<style> 的 name 命名使用大驼峰命名法。

例如:

<style name="ContentText">  
 <item name="android:textSize">@dimen/font\_normal</item>  
 <item name="android:textColor">@color/basic\_black</item>  
</style>

#### 6.2.15.6 id 命名

id的命名规则一直以来一般采用小写下划线法：view 缩写{\_模块名}\_逻辑名。

例如：btn\_main\_search、btn\_back.

例外：Kotlin中View binding插件功能会将id直接作为控件变量名在类中使用，当id是用于Kotlin时，直接采用lowerCamelCase。

UI控件名称缩写表：

| **名称** | **缩写** |
| --- | --- |
| Button | btn / bt |
| CheckBox | cb |
| EditText | et |
| FrameLayout | fl |
| GridView | gv |
| ImageButton | ib |
| ImageView | iv |
| LinearLayout | ll |
| ListView | lv |
| ProgressBar | pb |
| RadioButtion | rb |
| RecyclerView | rv |
| RelativeLayout | rl |
| ScrollView | sv |
| SeekBar | sb |
| Spinner | spn |
| TextView | tv |
| ToggleButton | tb |
| VideoView | vv |
| WebView | wv |

## 6.3 文档注释（Javadoc）

1. Javadoc应该应用于所有的public类、public和protected的成员变量和方法，至少用一句话说明类或方法的用途。

在IDE中，调用对应类或方法时，可以不进入即可悬浮提示方法、参数、返回值的意义，提高阅读效率，对于AS可以生成模板，只需要补全参数即可。

1. 一些在包外不可见的class和成员变量或方法，根据需要，也可以使用Javadoc。
2. 不需要使用英文进行注释，用中文注释把问题说清楚。
3. 专有名词与关键字保持英文原文即可.比如TCP代替传输控制协议。
4. 描述一个方法时，应当只保留方法名字，不要附带方法的参数。

比如：有个方法是add(Object obj)，那么用add指代该方法即可，而不是add(Object obj)

例外**：**

1. 当方法本身很显而易见时，可以不需要Javadoc。例如：getFoo。没有必要加上Javadoc说明“Returns the foo”。

如果该方法执行更复杂的操作（例如强制实施约束条件或具有重大副作用），那么必须添加备注。如果名称的意思不明确，也应该添加备注。

1. 重载方法有时不需要再写Javadoc。

### 6.3.1 基本格式

1. 当Javadoc块只有一行时，可以使用单行格式来替代通用格式。
2. 如果注释包含多段内容，段与段之间需要用 <p> 分隔(AS格式化自动完成)，空行不会生效。
3. 为了避免一行过长影响阅读效果，务必将每行的长度限制在100个字符以内。

例如：

/\*\*  
 \* Multiple lines of Javadoc text are written here,  
 \* <p>  
 \* wrapped normally...  
 \*/  
public int method(String p1) {...}

或者为单行格式：/\*\* An especially short bit of Javadoc. \*/

### 6.3.2 标签

1. 在描述和标签列表之间插入空白注释行。
2. 所有标准的@从句，应该按照如下的顺序添加：@param、@return、@throws、@deprecated、@see。
3. 当@从句无法在一行写完时，应该断行，延续行在第一行的@字符的位置，缩进至少4个字符单位。
4. 这些@从句，不应该出现在一个没有描述的Javadoc块中。

常见的错误是以这种形式使用Javadoc：/\*\* @return the customer ID \*/。这是错误的，应该改为：/\*\* Returns the customer ID. \*/。

1. @author和@version目前不作强行要求，其余除@see以外建议在需要时尽量使用，@see根据情况进行使用外。

主要标签:

1. @author 作者，没有特殊格式要求，名字或组织名称都可以。

@version  软件版本号（注意不是Java版本号），没有特殊格式要求。

1. @param 方法参数，格式为：@param 参数名称 参数描述．

当有多个参数时添加多个@param，并应该按照参数定义的顺序排列。

1. @return 方法返回值，格式为：@return 返回值描述 ，若方法没有返回值可以省略。
2. @deprecated 方法已过时时使用，应该告诉用户这个API被哪个新方法替代了，随后用 @see 标记或 {@link} 标记指向新API，比如：

/\*\*  
 \* **@deprecated** As of JDK 1.1, replaced by  
 \* {**@link** #setBounds(int,int,int,int)}  
 \*/

1. @throws　包含方法显式抛出的检查异常(Checked Exception)，至于非显示抛出的其他异常(Unchecked Exception)，除非特别有必要，可以不进行编写。原则是：只记录可控的问题，对于不可控的或不可预测的问题，不要往上写。

多个@thrown应该按照异常的字母顺序排列。

1. 多个@see标记，应该按照注释的逻辑顺序排列，即从最近的到最远的，从最具体的到最一般的。

完整示例:

/\*\*  
 \* Returns an Image object that can then be painted on the screen.  
 \* The url argument must specify an absolute {**@link** URL}. The name  
 \* argument is a specifier that is relative to the url argument.  
 \* <p>  
 \* This method always returns immediately, whether or not the  
 \* image exists. When this applet attempts to draw the image on  
 \* the screen, the data will be loaded. The graphics primitives  
 \* that draw the image will incrementally paint on the screen.  
 \*  
 \* **@param** url an absolute URL giving the base location of the image  
 \* **@param** name the location of the image, relative to the url argument  
 \* **@return** the image at the specified URL  
 \* **@see** Image  
 \*/  
public Image getImage(URL url, String name) {  
 try {  
 return getImage(new URL(url, name));  
 } catch (MalformedURLException e) {  
 return null;  
 }  
}

### 6.3.3 内联标记

Javadoc中可以使用{@link xxx}链接到对应的类，内联标记具有以下特点和注意点：

1. {@link xxx} 标记会被转化成指向URL类的超链接。
2. 控制 {@link xxx} 的数量，太多的链接会使文档的可读性很差。
3. 不要出现相同的链接，同样的链接只保留第一个。
4. 不要为Java自带的内容或是常识性的内容提供链接。
5. {@link xxx}标记中类使用 . 进行连接。

例如：{@link Type.ScreenChildType}

1. 类中的元素(方法，属性)使用 # 进行连接。

例如：{@link System. PrintStream#println()}

### 6.3.4 包注释

需要在对应包中建立package-info.java文件与Java文件一起放置在源码目录中，进行包声明后在包声明上进行文档注释，包注释不作强行要求。

例如：

/\*\*  
 \* <b>package-info不是平常类</b>  
 \* 可以声明包的私有类和常量，可以提供包的整体注释说明，可以使用内联标记。  
 \*  
 \* **@see** {@link #print(boolean)}  
 \*/  
package java.io;

### 6.3.4 KDoc

Kotlin的文档注释与Java的有少许不同，生成时不使用Javadoc工具而使用 [Dokka](https://github.com/Kotlin/dokka)进行生成，同样能在IDE中直接进行悬浮提示效果。

这里只对不同部分进行说明，未说明部分表示与Javadoc相同。

额外的标签

@constructor 用于类的主构造函数。

@receiver 用于扩展函数的接收者。

@property <名称> 用于类中具有指定名称的属性，可用于在主构造函数中声明的属性。

@sample <标识符> 将具有指定限定的名称的函数的主体嵌入到当前元素的文档中， 以显示如何使用该元素的示例。

@suppress 从生成的文档中排除元素。

KDoc不支持 @deprecated 标签，使用 @Deprecated 注解: @Deprecated("xxx")

KDoc内联标记

KDoc 使用常规 [Markdown](http://daringfireball.net/projects/markdown/syntax) 语法

要链接到另一个元素（类、方法、属性或参数），只需将其名称放在方括号中。

例如：[foo]

如果要为链接指定自定义标签，请使用 Markdown 引用样式语法。

例如：[这个方法][foo]

限定的名称总是使用点字符来分隔组件，即使在方法名称之前。

例如： [kotlin.reflect.KClass.properties]

KDoc包文档

KDOC包文档使用单独的Markdown文件，使用Dokka生成时需要使用-include额外指定。Kotlin包文档也不做必须要求。

在包文档中，作为一个整体的模块和分开的软件包的文档由相应的一级标题引入；

标题的文本对于模块必须是“Module <模块名>”，对于包必须是“Package <限定的包名>”。

## 6.4 日志记录

虽然日志记录非常有必要，但对性能却有明显的负面影响，如果不能保持一定程度的简洁性，就会迅速失去其实用性。日志记录应遵守以下原则:

1. Log的 tag不能是空字符串，即：" "。
2. 请勿使用 System.out.println()。
3. 不要将敏感信息打印到 log 中。请务必谨慎，避免在日志中泄露安全方面的信息，避免提供个人信息和有关受保护内容的信息，特别是进行手机上的APP开发时。
4. 请注意，在对 String 使用 + 运算符时，它会隐式创建一个具有默认缓冲区（大小为 16 个字符）的 StringBuilder，还可能会创建其他临时 String 对象。因此，显式创建 StringBuilder 并不比依赖默认的“+”运算符成本更高（实际上可能更高效）。
5. 供其他人阅读并且在发布版本中提供的日志记录应当简洁明了、合理易懂。仅供自己阅读的日志应该在发布版本中尽量去除。
6. 在内容有意义的情况下尽可能使日志记录在一行之内。一行长度在 100 个字符内是可以接受的，应当尽可能避免长度超过 130 或 160 个字符。
7. 当事实证明某些日志记录可能会发生多次时，最好实施一种频率限制机制来防止出现具有相同（或非常相似）信息的大量重复日志副本。
8. 在发布前应尽量减少代码中不必要的日志输出，对于在发布中保留的输出语句确保不会频繁输出。

## 6.5 编程实践

### 6.5.1 异常捕获规约

可以通过预检查方式规避的 RuntimeException 异常不应该通过 catch 的方式来处理，比如：NullPointerException，IndexOutOfBoundsException 等；

无法通过预检查的异常除外，比如：在解析字符串形式的数字时，不得不通过 catch NumberFormatException 来实现。

一般情况下，catch住的异常不应该被忽略，而是都需要做适当的处理。例如将错误日志打印出来，或者如果认为这种异常完全不会发生，则应该作为断言异常重新抛出。可接受的替代方案（按优先顺序排列）包括:

1. 将异常抛给方法调用者。

void setServerPort(String value) throws NumberFormatException {  
 serverPort = Integer.*parseInt*(value);  
}

1. 抛出一个适合您的抽象级别的新异常。

void setServerPort(String value) throws ConfigurationException {  
 try {  
 serverPort = Integer.*parseInt*(value);  
 } catch (NumberFormatException e) {  
 throw new ConfigurationException("Port " + value + " is not valid.");  
 }  
}

1. 抛出一个适合您的抽象级别的新异常。

void setServerPort(String value) throws ConfigurationException {  
 try {  
 serverPort = Integer.*parseInt*(value);  
 } catch (NumberFormatException e) {  
 throw new ConfigurationException("Port " + value + " is not valid.");  
 }  
}

1. **妥善处理错误，并替换 catch {} 块中的相应值。**

*/\*\* Set port. If value is not a valid number, 80 is substituted. \*/*void setServerPort(String value) {  
 try {  
 serverPort = Integer.*parseInt*(value);  
 } catch (NumberFormatException e) {  
 serverPort = 80; // default port for server  
 }  
}

1. 捕获异常并抛出一个新的 RuntimeException。

这样做比较危险，仅用在你确定如果发生此错误，最适当的处理方式就是让应用崩溃时。

void setServerPort(String value) {  
 try {  
 serverPort = Integer.*parseInt*(value);  
 } catch (NumberFormatException e) {  
 throw new RuntimeException("port " + value " is invalid, ", e);  
 }  
}

1. 如果确信忽略异常是合适的处理方式，那么可以忽略异常，但必须添加备注以充分说明理由。

*/\*\* If value is not a valid number, original port number is used. \*/*void setServerPort(String value) {  
 try {  
 serverPort = Integer.*parseInt*(value);  
 } catch (NumberFormatException e) {  
 // Method is documented to just ignore invalid user input.  
 // serverPort will just be unchanged.  
 }  
}

例外：在测试类里，有时会针对方法是否会抛出指定的异常，这样的异常是可以被忽略的。但是这个异常通常需要命名为：expected。

例如：

try {  
 emptyStack.pop();  
 fail();  
} catch (NoSuchElementException expected) {  
}

关于在循环中使用try-catch语句

在一些代码规范中禁止在循环内部使用try-catch语句，经过进一步查证，当不发生异常时，在循环外和循环内性能基本完全没区别，但是重要的是循环外部的异常捕获会造成直接跳出循环，循环内部的异常捕获会让循环继续，性能的消耗主要体现在捕获异常后的处理。

所以：当针对需要在循环中捕获了异常还要继续循环的逻辑场景允许在循环中使用try-catch，但不需要时一律需要在循环外部进行异常捕获。

请勿直接捕获Exception

几乎所有情况下都不适合直接捕获Exception异常或 Throwable（Throwable包含 Error 异常）。这样做可能会捕获到你从未预料到的异常（比如ClassCastException 等 RuntimeException）。它会掩盖代码的故障处理属性，如果后续有人在你所调用的代码中添加了一种新类型的异常，编译器不会意识到你需要采取不同的方式来处理该错误。在大多数情况下，你不应以相同的方式处理不同类型的异常。不要为了方便而倾向于采用这种处理。

反例：

try {  
 someComplicatedIOFunction(); // may throw IOException  
 someComplicatedParsingFunction(); // may throw ParsingException  
 someComplicatedSecurityFunction(); // may throw SecurityException  
 // phew, made it all the way  
} catch (Exception e) { // I'll just catch all exceptions  
 handleError(); // with one generic handler!  
}

例外：在测试代码和顶级代码中，有时会希望捕获所有类型的错误（以防它们显示在界面中或者以便一直进行批处理作业）。在这些情况下，可以捕获Exception（或 Throwable）并适当地处理错误。但在这样做之前，请务必三思，并添加备注以说明为何在此处执行这类操作是安全之举。

直接捕获Exception的替代方案：

1. 在单个 try 之后将每个异常作为单独的 catch 块分别进行捕获。这样做可能显得比较笨拙，但仍比捕获所有异常更可取，注意不要在 catch 块中过多地重复使用代码。
2. 通过多个 try 块重构您的代码，使得错误处理过程更精细。从解析中分离出不同异常，然后分别处理每种情况下的错误。
3. 重新抛出异常。很多时候，无需在该级别捕获异常，只需让相应方法抛出异常即可。

### 6.5.2 线程使用规约

新建线程时，必须通过线程池提供（AsyncTask 或 ThreadPoolExecutor 或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。

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，创建线程或线程池时请指定有意义的线程名称，方便出错时回溯。

正例：

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

反例：

new Thread(new Runnable() {  
 @Override  
 public void run() {  
 //操作语句   
 ...  
 }   
}).start();

扩展参考：<https://blog.mindorks.com/threadpoolexecutor-in-android-8e9d22330ee3>

#### 6.5.2.1 线程池不允许使用 Executors 去创建

Executors 返回的线程池对象具有以下弊端：

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

通过 ThreadPoolExecutor等方式创建，可以更加明确线程池的运行规则，规避资源耗尽的风险。

正例：

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

反例：

ExecutorService cachedThreadPool = Executors.*newCachedThreadPool*();

扩展参考：<http://dev.bizo.com/2014/06/cached-thread-pool-considered-harmlful.html>

### 6.5.3 常规代码规约

#### 6.5.3.1 静态成员的访问

当一个静态成员被访问时，应该通过class名去访问，禁止使用这个class的具体实例对象进行访问。

例如：

Foo aFoo = ...;  
Foo.aStaticMethod(); // good  
aFoo.aStaticMethod(); // bad  
somethingThatYieldsAFoo().aStaticMethod(); // very bad

#### 6.5.3.2 long类型的表示使用L

在 long 或者 Long 赋值时，数值后使用大写的 L，不能是小写的 l，小写容易跟数字 1 混淆，造成误解。

#### 6.5.3.3 基本数据类型转为字符串

基本数据类型.toString() 是最快的方式，String.valueOf(数据)次之，数据 + ""最慢。

#### 6.5.3.4 确保关闭资源对象和流对象

如果可能有异常情况出现，使用try-catch-finally方式进行异常捕获和对资源对象流对象进行关闭。JDK7及以上，可以使用 try-with-resources 方式。

#### 6.5.3.5 使用常量或确定有值的对象来调用 equals

equals 方法容易抛出空指针异常，应使用常量或确定有值的对象来调用 equals。

当是在与常量或确定有值的对象进行比较时，必须使用常量或确定有值的对象来调用。

正例："test".equals(object);

反例：object.equals("test");

#### 6.5.3.6 尽量减少对变量的重复计算；

如下面的操作：

for (int i = 0; i < list.size(); i++) {

...

}

建议替换为：

for (int i = 0, len = list.size(); i < len; i++) {

...

}

#### 6.5.3.7 尽可能使用局部变量以及注意

在Java类内部方法中定义的局部变量或对象是存储在stack（堆栈）中的，且JVM是一种stack-based的，因此访问和操纵stack中的数据时性能最佳。

而Java类的instance变量（这个类的field）和static变量是在constant pool（常量池）中存储和得到访问的。constant pool中保存了所有的符号引用（symbolic references），指向所有型别(types)、值域(field)，以及每个型别所使用的所有函数（mothods）。访问instance和static变量时，由于它们存放于constant pool中，所以JVM需要使用更多更耗时的操作码来访问它们。

例如:

private int x; // instance变量   
private static int *staticX*; //static 变量   
  
public void stackAccess(int val) { //访问和操作stack变量j   
 int j = 0;  
 for (int i = 0; i < val; i++) {  
 j += 1;  
 }  
}  
  
public void instanceAccess(int val) {//访问和操作instance变量x   
 for (int i = 0; i < val; i++) {  
 x += 1;  
 }  
}  
  
public void staticAccess(int val) {//访问和操作static变量staticX   
 for (int i = 0; i < val; i++) {  
 *staticX* += 1;  
 }  
}

运行instanceAccess（）和staticAccess（）方法的时间大约相同，但却比运行stackAccess（）方法慢了2～3倍。

应该将instance和static变量放到循环之外，而用一个stack变量来完成多次局部运算，最后再将这个stack变量的值传回instance或static变量：

public void instanceAccess(int val) {//访问和操作instance变量x   
 int tempX = x;  
 for (int i = 0; i < val; i++) {  
 tempX += 1;  
 }  
 x = tempX;  
}  
  
public void staticAccess(int val) { //访问和操作static变量staticX   
 int tempStaticX = staticX;  
 for (int i = 0; i < val; i++) {  
 tempStaticX += 1;  
 }  
 staticX = tempStaticX;  
}

注意：

1. 局部变量不应该习惯性地放在语句块的开始处声明，而应该尽量离它第一次使用的地方最近的地方声明，以减小它们的使用范围。
2. 尽可能缩小局部变量的作用域，这样做有助于提高代码的可读性和可维护性，并降低出错的可能性。
3. 每个变量应该在包含变量所有使用场合的最内层的块中进行声明。
4. 局部变量应该在声明的时候就进行初始化，如果不能在声明时初始化，也应该尽快完成初始化。

例如：

String str = "aaa";  
if (i == 1) {  
 list.add(str);  
}

建议替换为：

if (i == 1) {  
 String str = "aaa";  
 list.add(str);  
}

#### 6.5.3.8 @override都应该使用

@override 注解只要是符合语法的，都应该使用。即可简单验证是否覆盖成功.

#### 6.5.3.9 不能使用过时的类或方法

当被@Deprecated注解修饰时，应该进入到代码中查看注释中@Deprecated的内容，一般会标注有替代的新方法，请勿直接使用过时的方法。

#### 6.5.3.10 尽量在合适的场合使用单例；

使用单例可以减轻加载的负担、缩短加载的时间、提高加载的效率，但并不是所有地方都适用于单例，在实现单例时，需要保证线程安全，尽量使用静态内部类和枚举的方式实现单例。

简单来说，单例主要适用于以下三个方面：

1. 控制资源的使用，通过线程同步来控制资源的并发访问。
2. 控制实例的产生，以达到节约资源的目的。
3. 控制数据的共享，在不建立直接关联的条件下，让多个不相关的线程之间实现通信。

#### 6.5.3.11 善用封装

数据提供统一的入口，无论是在 MVP、MVC 还是 MVVM 中，提供一个统一的数据入口，都可以让代码变得更加易于维护。

比如：可使用一个 DataManager，把 http、preference、eventpost、database 都放在 DataManager 里面进行操作，只需要与 DataManager 进行交互。

#### 6.5.3.12 编写简短方法

在可行的情况下，尽量编写短小精炼的方法。有些情况下较长的方法是恰当的，因此对方法的代码长度没有做出硬性限制。如果某个方法的代码超出 40 行，请考虑是否可以在不破坏程序结构的前提下对其拆解。

#### 6.5.3.13 避免出现重复的代码

随意复制和粘贴代码，必然会导致代码的重复，在需要修改时，需要修改所有的副本，容易遗漏。

必要时抽取共性方法，或者抽象公共类，甚至是组件化，尽量提取方法，去除重复代码。

对于必要的工具类抽取也很重要，在以后的项目中是可能重用的。

#### 6.5.3.14 清理注释掉的代码或添加注释

谨慎注释掉代码，对于注释掉的代码需要在上方详细说明，而不是简单地注释掉。

如果无用，则删除。

对于代码被注释掉只有两种可能性：

1. 后续会恢复此段代码逻辑。
2. 永久不用。前者如果没有备注信息，难以知晓注释动机。后者建议直接删掉（但需要确保代码仓库保存了历史代码）。

#### 6.5.3.15 获取当前毫秒数使用System.currentTimeMillis();

获取当前毫秒数 System.currentTimeMillis()，而不是 new Date().getTime()。

#### 6.5.3.16 在循环中使用 StringBuilder拼接字符串

循环体内，字符串的连接方式使用 StringBuilder 的 append 方法进行扩展。

使用 + 进行字符串拼接每次循环都会新建一个 StringBuilder 对象，造成内存资源浪费。

#### 6.5.3.17 关于 hashCode 和 equals 的处理

1. 只要重写 equals，就必须重写 hashCode。
2. 因为 Set 存储的是不重复的对象，依据 hashCode 和 equals 进行判断，所以 Set 存储的对象必须重写这两个方法。
3. 如果自定义对象作为 Map的键，那么必须重写 hashCode 和 equals。

String 重写了 hashCode 和 equals 方法，所以可以使用String对象作为key来使用

#### 6.5.3.18 不使用Finalizers 方法

重载Object的finalize方法是非常非常罕见的。如果你认为你必须使用，请先仔细阅读并理解 Effective Java 第七条 “Avoid Finalizers”。

#### 6.5.3.19 建议在Java中尽量使用非空情况以适配Kotlin

例如：方法的返回值永远不会返回null，并为方法添加@NonNull注解，当与Kotlin进行调用时Kotlin确定了这属于返回了不可空类型，让Kotlin后续的编码更简洁清晰。

### 6.5.4 Android 中的代码规约

#### 6.5.4.1 抽取基类

当Activity 和 Fragment 里面有许多重复的操作以及操作步骤，建议提供一个 BaseActivity 和 BaseFragment，让所有的 Activity 和 Fragment 都继承这个基类，而不是在多个类中出现了类似的代码。

#### 6.5.4.2 Intent传递数据限制

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。

#### 6.5.4.3 广播的使用

尽量少使用广播，因为广播的接收可能的时机可能较晚，以及可能出现泄漏和被拦截。

进程间进行通信时尽量使用其他更即时和安全的方式，进程内进行通信尽量使用回调等方式。

当必须在进程内使用广播通信时，建议使用LocalBroadcastManager。

若使用了广播， registerReceiver() 和 unregisterReceiver()必须要成对出现。

#### 6.5.4.4 Android文件系统访问

不要硬编码文件路径，可以使用Android文件系统API访问的部分请使用下面列出的 Android 文件系统 API 进行访问。

android.os.Environment#getExternalStorageDirectory()

android.os.Environment#getExternalStoragePublicDirectory()

android.content.Context#getFilesDir()

android.content.Context#getCacheDir

正例：

new File(Environment.*getExternalStoragePublicDirectory*(  
 Environment. *DIRECTORY\_PICTURES*), alName)

反例：

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

###### 当使用外部存储时，必须检查外部存储的可用性

//读/写检查   
public boolean isExternalStorageWritable() {  
 String state = Environment.*getExternalStorageState*();  
 if (Environment.*MEDIA\_MOUNTED*.equals(state)) {  
 return true;  
 }  
 return false;  
}  
// 只读检查   
public boolean isExternalStorageReadable() {  
 String state = Environment.*getExternalStorageState*();  
 if (Environment.*MEDIA\_MOUNTED*.equals(state) ||  
 Environment.*MEDIA\_MOUNTED\_READ\_ONLY*.equals(state)) {  
 return true;  
 }  
 return false;  
}

扩展参考：

1. https://developer.android.com/training/data-storage/files.html
2. https://developer.android.com/reference/android/os/Environment.html#getExt ernalStorageDirectory()

#### 6.5.4.5 ArrayMap和SparseArray

当数据量小于1000时尽量使用ArrayMap和SparseArray；

当数据量大于1000时尽量使用 HashMap、ArrayList、StringBuilder；

除非线程安全需要，否则不推荐使用 HashTable、Vector、StringBuffer，其由于使用同步机制而导致了性能开销。

#### 6.5.4.6 SharedPreference 提交数据

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()。

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，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。

#### 6.5.4.7 数据库 Cursor 的使用

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。但是当结果集较大和长时间重复操作会导致内存消耗过大，需要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。

#### 6.5.4.8 多线程操作写入数据库时需要使用事务

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，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。

例：

public void insertUserPhoto(SQLiteDatabase db,

String userId, String content) {  
 ContentValues cv = new ContentValues();  
 cv.put("userId", userId);  
 cv.put("content", content);  
 db.beginTransaction();  
 try {  
 db.insert(TUserPhoto, null, cv);  
 // 其他操作   
 db.setTransactionSuccessful();  
 } catch (Exception e) {  
 // *TODO* } finally {  
 db.endTransaction();  
 }  
}

#### 6.5.4.9 保证数据库执行速度

大数据写入数据库时，使用事务或其他方案提高 I/O 效率的机制，保证执行速度。

public void insertBulk(SQLiteDatabase db,

ArrayList<UserInfo> users) {  
 db.beginTransaction();  
 try {  
 for (int i = 0; i < users.size; i++) {  
 ContentValues cv = new ContentValues();  
 cv.put("userId", users[i].userId);  
 cv.put("content", users[i].content);  
 db.insert(TUserPhoto, null, cv);  
 }  
 // 其他操作   
 db.setTransactionSuccessful();  
 } catch (Exception e) {  
 // *TODO* } finally {  
 db.endTransaction();  
 }  
}

#### 6.5.4.10 执行SQL语句时使用被封装方法

应使用SQLiteDatabase#insert()、update()、delete()，不要使用 execSQL()，以免 SQL 注入风险。

#### 6.5.4.11 图片加载注意事项

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。

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泄露，也避免重复创建图片，引起性能问题。

使用BitmapDrawable时，考虑是否可以替换为使用纯色（ColorDrawable）、 渐变色（GradientDrawable）、 StateSelector（StateListDrawable）等与 Shape 结合的形式构建绘图.

#### 6.5.4.12 使用 ARGB\_565 代替 ARGB\_888，

ARGB\_565在不太降低视觉效果的前提下，可以减少内存占用。

但注意 RGB\_565 没有透明度，如果图片本身需要保留透明度，就不能使用 RGB\_565。

#### 6.5.4.13 使用完毕的图片应该及时回收

例如：

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* <= 10) {  
 bitmap.recycle();  
}  
bitmap = null;

#### 6.5.4.14 界面后台或关闭时的释放操作

建议不要在 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。

在Activity.onPause()或Activity.onStop()回调中，若有正在执行的动画需要进行关闭。

#### 6.5.4.15 没必要总是为bean类添加setter/getter方法

因为Java没有原生的getter / setter 语法，getter / setter 语法只是一个旧的Java Beans的命名规范，而在web开发中getter、setter方法是可能会起到一些实际作用以及一些框架会使用该特性进行一些特殊处理，它作为一个中间层起到了一定的实际作用，比如：

1. 这两个方法可以方便增加额外功能（比如验证）。
2. 内部存储和外部表现不同。
3. 可以保持外部接口不变的情况下，修改内部存储方式和逻辑。
4. 任意管理变量的生命周期和内存存储方式。
5. 提供一个debug接口。
6. 能够和模拟对象、序列化乃至WPF库等融合。
7. 允许继承者改变语义。
8. 可以将getter、setter用于Lambda表达式。
9. getter和setter可以有不同的访问级别。

但是Android开发中，Bean类的getter/setter方法很多情况下并没有起到实际意义，而没有逻辑意义的getter/setter是不应该存在的。

在Android源码中也符合这一标准，并没有进行无意义的getter/setter方法的添加，所以没必要总是为bean类添加无意义的getter/setter方法。

在出现包括但不限于这些情况时可以使用setter/getter方法：

1. 有效性验证：可以在setter里对设置的值进行检查。
2. 惰性求值：比如一个成员计算过于耗时，而这个类的使用处不一定全部需要使用到时，可以在getter方法调用的时候再计算。
3. 缓存额外的操作，比如用户调用setter方法时，可以把这个值更新到配置文件中。
4. 通知：比如其它模块可能需要在某个值发生变化的时候进行一些操作，那么就可以在setter里实现通知。
5. 调试：可以方便的打印设置日志，从而追踪错误。
6. 同步：如果多线程访问需要进行同步处理，可以在setter里实现。
7. 更精细的权限访问：比如只读或只可写。
8. 维护不变式关系：比如一个类内部要维持变量a和b有a = b \* 2的关系，那么在a和b的setter里维持这样的关系。
9. 不对外暴露内部的数据组织方式。
10. 即使类数据的组织结构发生变化也不需要修改外部用户的代码。

#### 6.5.4.16 使用FileProvider在应用间共享文件

应用间共享文件时，不要直接通过放宽文件系统权限的方式，比如使用intent直接传递 文件的URI分享文件到别的应用，而应使用 FileProvider。

#### 6.5.4.17 注意检查是否已经提供了想添加的方法

当准备进行某个方法的添加时，应确认是否已经存在了相同功能的方法。

例如：有些开发者会自定义检测String字符串是否为null并且不为null时是否为空串的方法，而实际上Android中已经提供了TextUtils.isEmpty(String)，类似的工具类还有DateUtils。

### 6.5.5 UI与布局规约

#### 6.5.5.1 合理布局

有效运用 <merge>、<ViewStub>、<include> 标签来优化布局，尽可能多的减少 UI 布局层级。

#### 6.5.5.2 尽量不要使用帧动画

帧动画AnimationDrawable在初始化时就将所有图片加载到内存中，非常占内存，并且不能释放，释放之后再次加载时会抛出异常。

如果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在机器配置过低时发生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。

图片数量较少的 AnimationDrawable 可以接受。

#### 6.5.5.3 必须多重嵌套时用 RelativeLayout降低嵌套数

节点所处位置越深，套嵌带来的 measure 越多，计算就会越费时。页面拥上的 View越多，measure、layout、draw所花费的时间就越久。

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。

当布局中不得不使用 ViewGroup 多重嵌套时，不要使用 LinearLayout 嵌套，使用 RelativeLayout，可以有效降低嵌套数。

尝试进行优化可以使用 Android Studio Monitor 里的 Hierarachy Viewer 工具，可视化的查看所有的 view。

#### 6.5.5.4 不能使用 ScrollView 包裹列表类控件

Android官方明确禁止了ScrollView中嵌套 List 或 RecyclerView的做法。使用ScrollView包裹ListView/GridView/ExpandableListVIew会把列表控件的所有 Item 都加载到内存中，会消耗巨大的内存和 cpu 去绘制图面。开发过程中会遇到各种视觉和交互问题，并对性能也有较大损耗。对于类似效果推荐使用 NestedScrollView。

正例：

<LinearLayout>  
 <android.support.v4.widget.NestedScrollView>  
 <LinearLayout>  
 <ImageView/>  
 ...  
 <android.support.v7.widget.RecyclerView/>  
 </LinearLayout>  
 </android.support.v4.widget.NestedScrollView>  
</LinearLayout>

反例：

<ScrollView>  
 <LinearLayout>  
 <TextView/>  
 ...  
 <ListView/>  
 <TextView />  
 </LinearLayout>  
</ScrollView>

扩展参考：

1. https://developer.android.com/reference/android/widget/ScrollView.html
2. https://developer.android.com/reference/android/support/v4/widget/NestedSc rollView.html

### 6.5.6 Kotlin特性的惯用法

#### 6.5.6.1 优先使用不可变数据

优先使用不可变而不是可变数据，初始化后未修改的局部变量与属性，总是将其声明为 val 而不是 var。

#### 6.5.6.2 使用不可变集合接口声明无需改变的集合

总是使用不可变集合接口（Collection, List, Set, Map）来声明无需改变的集合。

使用工厂函数创建集合实例时，尽可能选用返回不可变集合类型的函数。

// 不良：使用可变集合类型作为无需改变的值

fun validateValue(actualValue: String, allowedValues: HashSet<String>) { …… }

// 良好：使用不可变集合类型

fun validateValue(actualValue: String, allowedValues: Set<String>) { …… }

// 不良：arrayListOf() 返回 ArrayList<T>，这是一个可变集合类型

val allowedValues = *arrayListOf*("a", "b", "c")

// 良好：listOf() 返回 List<T>

val allowedValues = *listOf*("a", "b", "c")

#### 6.5.6.3 优先声明带有默认参数的函数而不是重载函数

// 不良

fun foo() = foo("a")  
fun foo(a: String) { /\*……\*/ }

// 良好

fun foo(a: String = "a") { /\*……\*/ }

#### 6.5.6.4 为多次用到的类型定义类型别名

如果有一个在代码库中多次用到的函数类型或者带有类型参数的类型，那么建议为它定义一个类型别名

例如：

typealias MouseClickHandler = (Any, MouseEvent) -> Unit  
typealias PersonIndex = Map<String, Person>

#### 6.5.6.5 尽量将it用法作为Lambda 表达式参数

在简短、非嵌套的 Lambda 表达式中建议使用 it 用法而不是显式声明参数。

而在有参数的嵌套 Lambda 表达式中，始终应该显式声明参数。

#### 6.5.6.6 在Lambda 表达式中返回

避免在 Lambda 表达式中使用多个返回到标签。应重新组织这样的 Lambda 表达式使其只有单一退出点。如果这无法做到或者不够清晰，请考虑将 Lambda 表达式转换为匿名函数。在 Lambda 表达式的最后一条语句中的返回标签应直接省略。

#### 6.5.6.7 优先使用try、if 与 when 的表达形式

例如：

应优先使用:

return if (x) foo() else bar()  
  
return when(x) {  
 0 -> "zero"  
 else -> "nonzero"  
}

而不是：

if (x) {  
 return foo()  
} else {  
 return bar()  
}  
  
when (x) {  
 0 -> return "zero"  
 else -> return "nonzero"  
}

#### 6.5.6.8 if 和 when的选择

二元条件优先使用 if 而不是 when;如果有三个或多个选项时优先使用 when。

#### 6.5.6.9 可空的 Boolean 值使用注意

如果需要在条件语句中用到可空的 Boolean，注意使用 if (value == true) 或 if (value == false) 检测。

#### 6.5.6.10 优先使用高阶函数而不是循环

优先使用高阶函数（filter、map 等）而不是循环。

当在使用多个高阶函数的复杂表达式与循环之间进行选择时，请了解每种情况下所执行操作的开销并且考虑性能因素。

例外：forEach优先使用常规的 for 循环. 除非 forEach 的接收者是可空的或者 forEach 用做长调用链的一部分。

#### 6.5.6.11 用until表示在区间上循环

for (i in 0..n - 1) { /\*……\*/ } // 不良  
for (i in 0 *until* n) { /\*……\*/ } // 良好

#### 6.5.6.12 字符串使用

1. 优先使用字符串模板而不是字符串拼接。
2. 优先使用多行字符串而不是将 \n 转义序列嵌入到常规字符串字面值中。
3. 如需在多行字符串中维护缩进，当生成的字符串不需要任何内部缩进时使用 trimIndent，而需要内部缩进时使用 trimMargin。

例如：

assertEquals(  
 """  
 Foo  
 Bar  
 """.*trimIndent*(),  
 value  
)  
  
val a = """if(a > 1) {  
 | return a  
 |}""".*trimMargin*()

#### 6.5.6.13 函数和属性的选择

在某些情况下，不带参数的函数可与只读属性互换，虽然语义相似，但是在底层算法中应优先使用属性而不是函数。因为属性具有如下特点：

1. 不会抛异常；
2. 计算开销小（或者在首次运行时缓存）；
3. 如果对象状态没有改变，那么多次调用都会返回相同结果。

#### 6.5.6.14 使用扩展函数减少API污染

每当你有一个主要用于某个对象的函数时，应考虑使其成为一个以该对象为接收者的扩展函数。另为了尽量减少 API 污染，应尽可能地限制扩展函数的可见性，应根据需要，使用局部扩展函数、成员扩展函数或者具有私有可视性的顶层扩展函数。

#### 6.5.6.15 中缀函数使用注意

一个函数只有用于两个角色类似的对象时才将其声明为中缀函数。如果一个方法会改动其接收者，那么不要声明为中缀形式。

良好示例如：and、 to、zip。 不良示例如：add。

#### 6.5.6.16 使用by lazy实现懒加载

by lazy() 是一种是一种属性委托，要求属性声明为val，可以使用于类属性或者局部变量。lazy()可以接受一个Lambda表达式作为参数，第一次调用时会执行 Lambda 表达式，以后调用该属性会返回之前的结果。

# 附: 参考资料

1. Google Java Style Guide

<https://google.github.io/styleguide/javaguide.html>

1. AOSP Java Code Style for Contributors

<https://source.android.google.cn/setup/contribute/code-style>

1. Kotlin Coding Conventions

<https://kotlinlang.org/docs/reference/coding-conventions.html>

1. AndroidStandardDevelop

<https://github.com/Blankj/AndroidStandardDevelop>

1. How to Write Doc Comments for the Javadoc Tool

<https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

1. Documenting Kotlin Code

<https://kotlinlang.org/docs/reference/kotlin-doc.html>

1. 阿里巴巴Java开发手册v1.4.0



1. 阿里巴巴Android开发手册

