Android编码规范

背景

Android的代码规范，Google有出过，阿里巴巴也出过，各大静态代码扫描工具也有。但都比较零散，有些规范甚至要求不同，无法统一。因此我们也需要一套统一的规范，尽量做到所有人的代码风格一致，相互快速理解代码。同时，我们会加上一些安全规范，从而可以让大家在安全意识上面有所增强，防患于未然。

目的

1. 统一规范，相互遵守，提升协作效率
2. 追求极致，鼓励精品
3. 提升质量意识，消除安全隐患

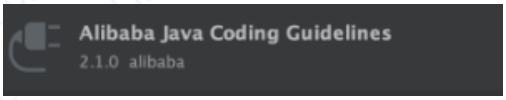
说明

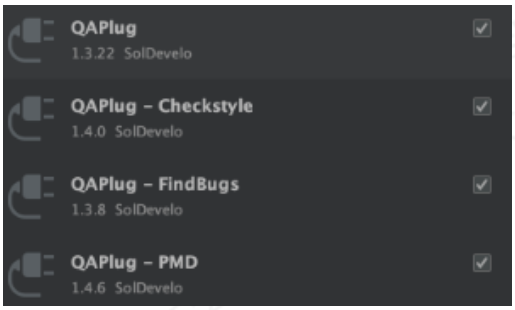
- 【强制】必须遵守，违反本约定或将会引起严重的后果；
- 【推荐】尽量遵守，长期遵守有助于系统稳定性和合作效率的提升；
- 【参考】充分理解，技术意识的引导，是个人学习、团队沟通、项目合作的方向

说明：规范中有涉及一些知识点的，会额外做解释，让开发同事不仅知道要这样做，更要知道为什么要这样做。

IDEA工具设置

【推荐】推荐安装QAPlug相关的四个插件及Alibaba Java Coding Guidelines，可以做到实时监测代码问题





说明：Codota是一款人工智能提示代码的插件，学习了当前工程的代码后，提示会更加智能，推荐安装

Kotlin编码规范

公司目前大部分项目采用Java语言开发，有部分项目组采用的开发语言是Kotlin，Kotlin编码规范见：Kotlin coding conventions (Kotlin编码规范)

编码风格

注释

基本原则：

1. 注释应该增加代码的清晰度。

2. 避免使用装饰性内容。
3. 保持注释的简洁。
4. 注释信息不仅要包括代码的功能，还应给出原因。
5. 不要为注释而注释。
6. 除变量定义等较短语句的注释可用行尾注释外，其他注释应当避免使用行尾注释。

【推荐】类、属性、方法、抽象方法、枚举应遵守javadoc规范，

【强制】接口类、接口方法、抽象方法、枚举应该提供注释。

说明：javadoc规范如下：

1. @author 标明开发该类模块的作者（对类的说明）
2. @version 标明该类模块的版本（对类的说明）
3. @date 类的创建日期或者修改日期（对类的说明）
4. @since 最早使用该方法/类/接口的JDK版本（对类的说明）
5. @see 参考转向，也就是相关主题（对类、方法、属性的说明）
6. @deprecated 引起不推荐使用的警告（对类、方法、属性的说明）
7. @modified 最后的修改作者与时间（对类、方法、属性的说明）
8. @param 对方法中某参数的说明（对方法的说明）
9. @return 对方法返回值的说明（对方法的说明）
10. @exception 对方法可能抛出的异常进行说明（对方法的说明）

正例

```
/**
 * {}
 * @author user.name
 * @date 2020-12-29
 * @modified author 2020-12-30()
 */
public class ActivityUtil {

    /**
     *
     */
    public static Object sGlobalData;

    /**
     * {/}
     * @param {} {}
     * @return {} {}
     * @exception {, }
     */
    public static Object doMethodCall(Object arg1, Object arg2) {
        return null;
    }
}
```

反例

【参考】代码应望文生义，避免过多无用的注释，应删除与代码逻辑不符的注释

正例

```
Order getOrder() {
    return mOrderData;
}
```

反例

```
/**
* Order
* void
*/
Order getOrder() {
    //
    return mOrderData;
}
```

【参考】单行注释应位于行首，同时不能有多余的todo

正例

```
public static Object doMethodCall(Object arg1, Object arg2) {
    //
    return null;
}
```

反例

```
public static Object doMethodCall(Object arg1, Object arg2) {
    // todo
    return null;
}
```

行限制

【推荐】一行代码不应该超过100个字符。

说明：以前是80个字符，但目前的显示器普遍是宽屏，也不会用于纸张打印，可以适当放宽。

【推荐】一个方法不应该超过80行。

说明：包含所有空行、注释。《Clean Code》提出过25行，但实际情况来看，有些业务逻辑繁琐，且加上匿名内部类，很容易超出，因此放宽上限。

【推荐】操作符前置：除赋值操作符外，需要把换行符放在操作符之前，如上一行未结束，下一行应空出8个空格（两个Tab）

正例1

```
int result =
        longVeriableName1
        + longVariableName2
        + longVariableName3;
```

正例2

```
if (testCondition1
        && testCondition2
        && testCondition3
        || testCondition4) {
    // code block
}
```

反例

```
int result = longVeriableName1 +
        longVariableName2 +
        longVariableName3;
```

【推荐】链式调用：当同一行中调用多个行数时，需要在"."之前换行

正例

```
Glide.with(fragment)
        .load(url)
        .placeholder(R.drawable.ic_banner_placeholder)
        .into(view.findViewById(R.id.image_banner));
```

反例

```
Glide.with(fragment).load(url).placeholder(R.drawable.ic_banner_placeholder)
        .into(view.findViewById(R.id.image_banner));
```

【推荐】多个参数：当一个方法有很多参数或者参数很长的时候，我们应该在每个","后面进行换行。

正例

```
loadPicture(context,
        "network_image.jpg",
        ivAvatar,
        "Avatar of the user",
        clickListener);
```

反例

```
loadPicture(context, "network_image.jpg", ivAvatar, "Avatar of the user", clickListener);
```

语法规范

【强制】重写方法必须添加@Override注解

正例

```
@Override
public void func() {
}
```

【强制】case必须有default，每个分支必须有break，相同逻辑可收归

正例

```
switch(type) {
    case TYPE_1:
        xxxx;
        break;
    case TYPE_2:
    case TYPE_3:
        xxxx;
        break;
    default:
        break;
}
```

【强制】if-else必须添加加{}，仅有一个return语句的可以省略，其他情况必须添加{}

正例1

```
if (test1) return;
```

正例2

```
if (test1) {
    return;
}
```

反例

```
if (test1) doSomething();
```

【推荐】避免判断条件复杂，超过3个条件的逻辑，应该抽出方法，或使用卫语句提前处理

正例1

```
private boolean testResult() {
    return a >= b
            && c >= d
            || c >= a;
}
if (testResult()) {
    doSomething();
}
```

正例2

```
if (a < b) {
    return;
}

if (c >=d || c >= a) {
    doSomething();
}
```

反例

```
if (a >= b && c >= d || c >= a) {
    doSomething();
}
```

【强制】finally中禁止执行return，容易混淆

正例

```
public int func() {
    try {
        return getResult();
    } finally {
        o.close();
    }
    return 1;
}
```

反例

```
public int func() {
    try {
        return getResult();
    } finally {
        o.close();
        return 1;
    }
}
```

【推荐】判断equals时，应该将常量/确定值在前，同时非基础类型都应该使用equals来判断是否相等

正例

```
FINAL_VALUE.equals(value)
```

反例

```
value != null && value.equals(FINAL_VALUE)
```

【强制】根据Java规范，重写了equals，必须重写hashcode，做到两个对象equal时，hashcode应该也相等。hashcode不同时，两个对象的equals应该返回false。
说明：此处采用IDE自动生成即可，高效无bug！

正例

```
public class Data {
    private String s1;
    private String s2;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Data data = (Data) o;

        if (s1 != null ? !s1.equals(data.s1) : data.s1 != null) return false;
        return s2 != null ? s2.equals(data.s2) : data.s2 == null;
    }

    @Override
    public int hashCode() {
        int result = s1 != null ? s1.hashCode() : 0;
        result = 31 * result + (s2 != null ? s2.hashCode() : 0);
    return result;
    }
}
```

【强制】异常只是对出现未知错误的场景的处理，不能用来做流程控制。

正例

```
public void doSomething(Object obj) {
    if (obj == null) {
        doOtherThings();
        return;
    }

    try {
        obj.calculateData();
    } catch (IOException e) {
        e.printStacktrace();
    }
}
```

反例

```
public void doSomething(Object obj) {
    try {
        obj.calculateData();
    } catch (NullPointerException e) {
        doOtherThings(); //
    } catch (IOException e) {
        e.printStacktrace();
    }
}
```

【推荐】异常的捕获需要尽量详细，如无特殊情况，不能直接捕获Exception。

正例

```
try {
    doSomething();
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

命名

Java代码

【强制】包名应全部采用小写字母

【推荐】类名不能以"_"或“$"开始或结束

【强制】类名不能采用拼音、中文

【强制】类名应采用UpperCamelCase，

【推荐】类名该类中如果用使用到设计模式，类名中需要有所体现

正例

```
public class OrderProxy
public class MessageFacade
```

【参考】private变量前应该加‘m'前缀，如：mOrderData; public变量无需前缀

【强制】Long型初始值数字末尾应该用大写L，小写l容易混淆。如 long numValue = 100L;

【强制】静态常量所有字母应大写

【推荐】Intent传递参数名应以EXTRA_KEY开始，如EXTRA_KEY_MODULE_PARAM_NAME，注释应该遵循javadoc规范

【强制】方法名应采用lowerCamelCase。

【推荐】方法表示一种行为，它代表一种动作，最好是一个动词后者动词词组或者一个单词为一个动词。如sendMessage

【推荐】全局变量（static变量）应该以s作为前缀明确表示，如：

正例

```
private static Object sInstance;
```

【推荐】方法参数应清晰表示其含义，一般为名词，如：function(String key, String defaultValue)

Android资源

【推荐】Android的所有资源文件需带模块前缀。

【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 的行 layout 以 module_grid_item 开头

【推荐】style 资源采用小写单词+下划线方式命名，写入 module_styles.xml 文件中，
采用以下规则：
父 style 名称.当前 style 名称

正例

```
<style name="ParentTheme.ThisActivityTheme">
…
</style>
```

【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

正例

```
<dimen name="module_horizontal_line_height">1dp</dimen>
```

【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 动画资源：尽可能以模 块+功能命名+序号。如：module_loading_grey_001

【强制】禁止使用app_name来指定App名字，应该改用其他key

说明：由于Android Studio创建的Demo工程会使用app_name，而很多第三方库也未做修改，如果直接使用，会导致部分语言下出现异常。

【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：module_login_tips,module_homepage_notice_desc

说明：如使用的是全局的翻译文件，则可根据翻译平台来命名即可，但组件内禁止使用PRD提供的key，应该根据命名规范做映射，如：

```
<string name="module_login_tips">@string/apc_tips</string>
```

其中apc_tips是翻译平台中指明的翻译key

【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>
```

说明：可以参考一下flutter对主题的字段定义：theme_data.dart

【强制】禁止使用app_luncher作为App的Luncher Icon

说明：由于Android Studio创建的Demo工程会使用app_luncher，而很多第三方库也未做修改，如果直接使用，而我们的App未提供更多的分辨率下的icon，会导致部分屏幕分辨率下icon异常（可能会出现绿色小机器人）。

【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，建议只使用一套,例如 drawable-xhdpi。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

【参考】UI组件缩写规范

常见使用单词缩写表

*UI*控件		*Android*中常见英文单词缩写			
全称	缩写	全称	缩写	全称	缩写
LinearLayout	ll	图标icon	ic	密码password	pwd
RelativeLayout	rl	颜色color	cl	位置position	pos
FrameLayout	fl	分割线divider	di	服务器server	ser
TableLayout	tl	选择器selector	sl	字符串string	str
Button	btn	背景background	bg	窗口window	win
ImageButton	ibtn	删除delete	del	临时temp	tmp
TextView	tv	错误error	err	图片image	img
EditText	et	信息infomation	info	长度length	len
ListView	lv	初始化initial	init	库library	lib
ImageView	iv			信息message	mes
GridView	gv				

UI与基础组件

【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或ConstraintLayout，可以有效降低嵌套数。

【推荐】文本大小使用单位sp，可以适配系统字体大小，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。

【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、ConstraintLayout 次之。

【推荐】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：图片数量较少的AnimationDrawable还是可以接受的。如果确实有图片较多的动画，建议使用Lottie或自行实现Bitmap复用的动画组件

【推荐】自定义组件应实现onSaveInstanceState与onRestoreInstanceState来适配Activity被回收时的情况，同时应该注意保存的数据不能太大，防止出现TransactionTooLargeException

【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。如果适配了Android 11设备，跳转的目标并非系统App，并且无法预知App包名，则可以只添加try-catch，不做resolveActivity的判断（这个方法会返回空列表）。

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) {
        try {
            startActivity(intent);
        } catch (ActivityNotFoundException e) {
            if (Config.LOGD) {
                Log.d(LOGTAG, "activity not found for " + mimeType + " over " +
                Uri.parse(url). getScheme(), e);
            }
        }
    }
}
```

【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。

【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。

安全

WebView

【推荐】api < 17禁止使用addJavaScriptInterface

【推荐】对于不需要使用File协议的应用，禁用File协议，显式设置webView.getSettings().setAllowFileAccess(false)，对于需要使用File协议的应用，禁止File协议调用JavaScript，显式设置webView.getSettings().setJavaScriptEnabled(false)。

Manifest.xml

【强制】将android:allowbackup属性设置为false，防止adb backup导出app数据。

【强制】应用发布前确保android:debuggable属性设置为false。

【强制】Receiver/Provider不能在毫无权限控制的情况下，将android:export设置为 true。

【强制】阻止webview通过file:schema方式访问本地敏感数据。

【强制】不要广播敏感信息，只能在本应用使用LocalBroadcast，避免被别的应用收到，或者 setPackage 做限制。

【强制】对于内部使用的组件，显示设置组件的"android:exported"属性为false。如果为导出组件，在接收到数据后，需要检查数据的合法性，防止被第三方程序恶意制造crash，甚至伪造数据。

其它安全

【强制】在请求后端接口的URL中，需要校验主机名的合法性，否则会导致恶意程序利用中间人攻击绕过主机名效验，从而导致请求被监听甚至被篡改

【强制】zip解压缩漏洞：zip文件可以采用相对目录解压，从而造成路径穿越效应，使恶意代码访问到外部目录，应通过路径判断避免恶意文件写入app内部目录

【强制】密钥加密存储或者经过变形处理后用于加解密运算，切勿硬编码到代码中。
说明：可参考APC团队的加密做法：Android 本地数据加密

工具类使用

Log

【强制】每个App应该有自己的Log工具类库，而不是使用系统的Log库来打印Log

【强制】禁止使用Android Log库来打印log

【强制】不得打印敏感信息（Token、user info、IMEI等），并对仅Debug的log用DEBUG开关隔离，防止引入线上。

【推荐】Log应用条件判断包裹起来，可以有效避免无效的参数计算，提高性能

【强制】禁止在Log中调用具有写操作的表达式，应分开写。

正例

```
public void doSomething() {
    xxxxxxx;
    String data = computeData();
    // Log
    if (LogUtil.isLoggable()) {
        LogUtil.i(TAG, "log info: " + data);
    }
}
```

反例

```
public void doSomething() {
    xxxxxxx;
    // LogisLoggable
    LogUtil.i(TAG, "log info: " + computeData());
}
```

【推荐】TAG应直接用字符串，不能用获取类名的方式，防止类名被混淆。

说明：直接用getSimpleName的方式虽然有利于重构，但混淆后的Log无法查看。如果重构忘了改这里的TAG，其实通过搜索代码，还是可以定位到具体的代码的。

正例

```
public class SomeClass {
    public static final String TAG = "SomeClass";
}
```

反例

```
public class SomeClass {
    public static final String TAG = SomeClass.class.getSimpleName();
}
```

【强制】Exception应在出现问题的地方创建，不能在事后补充。

说明：创建Exception会在基类初始化好对应的行号，调用堆栈，如果创建后，在其他地方再抛出来，就无法定位问题了。

正例

```
public void doSomething(String params) {
    xxxxxx;
    xxxxxx;
    if (params == null) {
        throw new NullPointerException("params is null");
    }
}
```

反例

```
public void doSomething(String params) {
    Exception exception = new NullPointerException("params is null");
    xxxxxx;
    xxxxxx;
    if (params == null) {
        throw exception;
    }
}
```

SharedPreference

【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。

【强制】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。

【强制】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。可以采用ContentProvider来实现。

【参考】SharedPreference会有ANR问题，如果可以，推荐使用MMKV或自研SP框架。

多线程

【强制】App应有自己统一的多线程管理库，应统一使用自己的线程池库，禁止直接创建线程池。

【强制】禁止直接创建线程。

【强制】CountDownLatch需要在finally中执行countDown，防止前面的代码出现异常，导致countDown未执行，发生死锁问题。

【强制】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

```
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

【强制】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

```
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());
//
executorService.execute(new Runnnable() {
    ...
});
```

反例

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

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

日期

【强制】SimpleDateFormat非线程安全，禁止使用static修饰，需要ThreadLocal包裹

【推荐】日期格式化应统一使用App内部封装好的DateUtils工具类，而不是直接使用系统工具类，会有时区不统一的问题。

【强制】任务耗时的计算，如与日期无关，应使用SystemClock类。

说明：如果使用System.currentTimeMillis来计算耗时，如果用户有调整过系统时间，就会造成计算错误，出现极大值或负数。使用SystemClock相关方法就不会出现这种情况。

正例

```
public long calTimeCost() {
    long start = SystemClock.elapsedRealtime();
    xxxxxx;
    xxxxxx;
    long end = SystemClock.elapsedRealtime();
    return end - start;
}
```

反例

```
public long calTimeCost() {
    long start = System.currentTimeMillis();
    xxxxxx;
    xxxxxx;
    long end = System.currentTimeMillis();
    return end - start;
}
```

【强制】使用日期格式化时，如无特殊需求，格式化年份应使用’yyyy‘，yyyy表示所在年份，YYYY表示当前周所在年份（如果当前周跨年，会算在下一年里面），应谨慎使用。

正例

```
private String formatTime(long time) {
    // 2020-12-302020-12-30
    String result = new SimpleDateFormat("yyyy-MM-dd").format(new Date(time * 1000));
    return result;
}
```

反例

```
private String formatTime(long time) {
    // 2020-12-302021-12-30
    String result = new SimpleDateFormat("YYYY-MM-dd").format(new Date(time * 1000));
    return result;
}
```

数据库

【强制】数据库Cursor必须确保使用完后应在finally中关闭，以免内存泄漏。

【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。如果有多进程操作，应该使用ContentProvider转入同一个进程来操作

说明：Android的通过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时一次原子操作，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。

【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);
    // SQL
    db.execSQL(sqlStmt);
}
```

数值计算

【强制】需要精度较高的数据计算，比如货币金额的计算，应该使用BigDecimal类，并且初始化时应该传入String，而不是数值。

说明：BigDecimal是精准的数值计算，但如果传入的是数值（在内存中已经不精准），会影响精准度，应该用字符串传递参数。

正例

```
new BigDecimal("0.000001");
```

反例

```
new BigDecimal(0.000001);
```

Parcelable序列化

【强制】需Parcelable序列化强制使用插件自动化生成，不可手写。(除非Parcelable插件不满足)

说明：手写序列化读写代码容易出错，且有可能不会崩溃，难以排除。

其他

【推荐】Android自带的Toast工具类会有BadToken问题，8.0系统后有修复，但我们需要做兼容性适配，因此尽量使用App中已经封装好的Toast库。

【推荐】Android权限检测会有机型兼容问题，应使用我们内部久经验证的权限库做权限申请、判断，不能使用系统库。

说明：Android很多机型如Vivo、Oppo等，之前使用一些第三方的权限管理App，有时候会给App返回权限，但实际使用的时候却没有权限。

【参考】推荐使用Camera2的相机库，但需要做API兼容性判断。

【强制】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。

正例

```
android.os.Environment#getExternalStorageDirectory()

android.os.Environment#getExternalStoragePublicDirectory()

android.content.Context#getFilesDir()

android.content.Context#getCacheDir()
```

【强制】当使用外部存储时，必须检查外部存储的可用性。

正例

```
// /
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

//
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)
            || Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

内存

【强制】IO、Stream、数据库等应在finally代码块中关闭，或使用try-with-resource方式（高版本api生效）。

【推荐】初始化集合应指定预期的大小。

正例

```
public void doSomething(int size) {
    Map<String, String> map = new Hashmap<String, String>(size); //
}
```

【强制】图片展示大小不应该大于屏幕分辨率，推荐用Glide加载。如必须自行处理，请计算好simple_size，再解码加载，并判断OOM。

【强制】无透明度的图片禁止使用ARGB8888方式来解码，应该用RGB_565方式。

性能

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

说明：下例中，反编译出的字节码文件显示每次循环都会 new 出一个 StringBuilder 对象， 然后进行 append 操作，最后通过 toString 方法返回 String 对象，造成内存资源浪费。

反例

```
String str = "start";
for (int i = 0; i < 100; i++) {
    str = str + "hello";
}
```

【强制】在使用正则表达式时，利用好其预编译功能，可以有效加快正则匹配速度。

说明：不要在方法体内定义：Pattern pattern = Pattern.compile(“规则”); 应直接缓存这个结果。

【推荐】对文件的读写，应当在异步线程进行，并加上BufferedStream包裹，提升读写性能。

依赖库

第三方库

【强制】引入第三方库应经过Manager/Leader讨论审批，禁止未经允许私自引入第三方库。

说明：引入第三方库很容易随之而来带入其他依赖库，导致代码不可控，需要仔细评估一下该库的依赖库有哪些，版本是否与App中的版本兼容。而且这个库带来的bug也可能引入线上。

【强制】使用第三方库，应指明具体的版本，禁止使用SNAPSHOT结尾的第三方库。

说明：对于maven仓库的引用，版本号是唯一的，无法被二次覆盖，可以避免我们的依赖库被静默升级。如果使用了SNAPSHOT的版本号，默认是可以被覆盖的，可能会出现代码被覆盖的情况，导致线上问题。

【推荐】我们所有依赖的第三方库，应该在统一的配置文件中指定版本号，做到整个App都依赖的同一个版本

说明：如果各个模块都使用自己的版本，则最终打包使用的版本虽然说是最新的版本，但是总是存在不确定性，最好能统一指定版本。

正例

```
// config.gradle
Versions.protobuf_java = 3.11.0


// build.gradle
implementation "com.google.protobuf:protobuf-javalite:${Versions.protobuf_java}"
```

内部依赖库

【推荐】内部库应以aar的形式在自建的maven仓库提供，并提供源码。

【推荐】内部库的版本升级可能会比较频繁，推荐使用与发布日期有关的版本号，如 1.21.0105，方便定位问题，也可以使用AndroidPods做版本控制

Git提交规范

【推荐】代码提交的message应该带上jira单号与清晰的描述，介绍主要的改动点与提交类型。可以参考Git提交规范。

例：

feat: [JIRA-Ticket] KYC优化

fix: [JIRA-Ticket] 修复订单错误的bug

Android静态代码扫描SDK

EEye SDK: EEye SDK是一款静态代码扫描工具集合，属于我们的CI检查模块，提供代码静态扫描，结果展示，Gitlab MR动态添加评论的能力，CI & CD的整体逻辑架构如下：

目前EEye集成了以下工具：

在MR中智能添加评论的效果如下：

扫描结果展示：

EEye SDK接入文档：文档完善中，后续提供完整的配置文档、原理文档

参考文档

1. 《阿里巴巴Android开发手册》
2. Android开发规范
3. Google代码规范
4. Git提交规范
5. Java https://github.com/ribot/android-g_and_code_guidelines.md
6. Kotlin http://kotlinlang.org/docs/reference/coding-conventions.html
7. https://github.com/ribot/android-guidelines/blob/master/project_and_code_guidelines.md