中软国际公司内部技术规范

Android平台Java语言编程规范

中软国际科技服务有限公司

修订声明

参考: 业界 Android 编码规范指南，华为 Android 编码规范等。

0.

日期	修订版本	修订描述	作者
2016-01-28	1.0	整理创建初稿	徐庆阳
2016-02-05	1.1	合入华为终端Android开发编程规范文档	徐庆阳
2016-02-29	1.2	根据业务部门评审意见修改	徐庆阳
2017-8-2	1.3	统一整理格式，并修正部分错误	陈丽佳

目录

1. 前言

建立 Android 平台 Java 语言编程规范的主要目的是：

统一编程风格

提高代码的可阅读性

减少错误产生

减少性能漏洞

提高代码可靠性

减少错误的编码设计

作为代码检查的依据

建立可维护的Android开发编程规范

本规范采用以下的术语描述：

★ 规则：编程时强制必须遵守的原则。

★ 建议：编程时必须加以考虑的原则。

★ 说明：对此规则或建议进行必要的解释、说明。

★ 示例：对此规则或建议给出脚本例子。 本规范没有涉及到的相关部分，请参见《Java 语言编程规范》。

2. 排版规范

2.1 规则

缩进风格

程序块要采用缩进风格编写，**缩进的空格数为 4 个**。说明：对于由开发工具自动生成的 代码可以有不一致。使用范围包括 java,xml 文件。

对齐尽量使用空格键，也可以使用TAB 键

说明：以免用不同的编辑器阅读程序时，因 TAB 键所设置的空格数目不同而造成程序布局不整齐。JBuilder、UltraEdit 等编辑环境，支持行首 TAB 替换成空格，应将该选项打开。

如果要用 tab，设置 eclipse 的 tab 为 4 个空格数。

较长的语句、表达式或参数

较长的语句、表达式或参数(**不超过80字符**)要分成多行书写,但不是只有超过80才分 行，多逻辑表达式建议添加(),不是靠本身的优先级判断，长表达式要在低优先级操作符处 划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。

示例：

```
if ( (filename != null && filename.length() < LogConfig.getFileSize())
      || list !=null)
{
... // program code
}
```

意义相同的考虑上下对齐，如下参数，不受缩进规则影响

```
public static LogIterator read(String logType, Date startTime, Date
                               endTime,
                               int logLevel, String userName, int
                               bufferNum)
```

方法里面的代码长度最大 80 行代码。

不允许把多个短语句写在一行中

不允许把多个短语句写在一行中，即一行只写一条语句示例：如下例子不符合规范：

```
LogFilename now = null;  LogFilename that = null;
```

应如下书写：

```
LogFilename now = null;
LogFilename that = null;
```

if, for, do, while, case, switch, default 等语句自占一行

if, for, do, while, case, switch, default 等语句自占一行，且 if, for, do, while 等语句的执行语句无论多少都要加括号{}。

示例：如下例子不符合规范。

```
if(writeToFile) writeFileThread.interrupt();
```

应 如 下 书 写 ：

```
if(writeToFile)  {
    writeFileThread.interrupt();
}
```

相对独立的程序块之间、变量说明之后必须加空行

示例，如下例子不符合规范：

```
if(log.getLevel() < LogConfig.getRecordLevel())
{
   return;
}
LogWriter writer;
```

应如下书写：

```
if(log.getLevel() < LogConfig.getRecordLevel())
{
   return;
}

LogWriter writer;
int index;
```

关键字、变量、常量进行对等操作

在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者 前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如.），后不应加空格。 说明：采用这种松散方式编写代码的目的是使代码更加清晰。由于留空格所产生的清晰性是相对的，

所以，在已经非常清晰的语句中没有必要再留空格，如果语句已足够清晰则括号内 侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在 Java 语言中 括号已经是最清晰的标志了。在长语句中，如果需要加的空格非常多，那么应该保持整体清 晰，而在局部不加空格。给操作符留空格时不要连续留两个以上空格。示例：

逗号、分号只在后面加空格。

```
int a, b, c;
```

比较操作符, 赋值操作符"="、 "+=", 算术操作符"+"、"%", 逻辑操作符"&&"、

"&", 位域操作符"<<"、"^"等双目操作符的前后加空格。 if (MAX_TIME_VALUE <= current_time)

```
a = b + c;
a *= 2;
a = b ^ 2;
```

"!"、"~"、"++"、"--"、"&"(地址运算符）等单目操作符前后不加空格。

```
flag = !isEmpty; // 非操作"!"与内容之间
i++; // "++","--"与内容之间
```

"."前后不加空格。

```
p.id = pid; // "."前后不加空格
```

if、for、while、switch 等与后面的括号间应加空格，使 if 等关键字更为突出、明显。

2.2 建议

交叉放置

类属性和类方法不要交叉放置，不同存取范围的属性或者方法也尽量不要交叉放置。 类格式定义：

```
{
    类的公有属性定义
    类的保护属性定义
    类的私有属性定义
    类的公有方法定义
    类的保护方法定义
    类的私有方法定义
}
```

3. 注释规范

3.1 规则

注释量

源程序有效注释量控制在一定比例，不追求绝对数字，按需确定。

注释的原则是有助于对程序的阅读理解，在该加的地方都加了，**注释不宜太多也不能太 少**，注释语言必须准确、易懂、简洁。

注释应解释代码的意图，而不是描述代码怎么做的。

注释风格

单行注释用//，块注释用/* */，JavaDoc 注释用/** */

类和接口注释

文件头的不添加版权注释，代码不开源。

类和接口注释放在 package 关键字之后，class 或者 interface 关键字之前。这样可 以方便 JavaDoc 收集。示例如下：

```
package com.company.android.mms.ui;

/**
 * 注释内容
 */
public class CommManager
```

类的注释主要是一句话功能描述，说明：可根据需要列出：生成日期、作者、内容、功 能、与其它类的关系等。

格式如下：

```
/**
 * @function: 功能描述
 * @author: [作者]
```

```
 * @version: [版本号, YYYY-MM-DD]
 */
```

说明：描述部分说明该类或者接口的功能、作用、使用方法和注意事项，每次修改后增加作者和更新版本号和日期。
示例：

```
/**
 * function: 集中控制对日志读写的操作。
 * @author: 张三，李四，王五
 * @version: 2001-03-25
*/
```

类属性、公有和保护方法注释

类属性、公有和保护方法注释写在类属性、公有和保护方法上面。示例：

```
/**
 * 注释内容
 */
private String logType;

/**
 * 注释内容
 */
public void write()
```

成员变量注释内容包括成员变量的意义、目的、功能，可能被用到的地方。 公有和保 护方法注释内容：列出方法的一句话功能简述、**功能详细描述、输入参数、输出参数、返回值**、违例等。

格式：

```
/**
 * 〈功能描述〉
 * @param [参数 1] [参数 1 说明]
 * @param [参数 2] [参数 2 说明]
 * @return [返回类型说明]
 * @exception/throws [违例类型] [违例说明]
 */
```

示例：

```
/**
```

```
 * 根据日志类型和时间读取日志。
 * 分配对应日志类型的 LogReader， 指定类型、查询时间段、条件和反复器缓冲数，
 *  读取日志记录。查询条件为 null 或 0 的表示没有限制，反复器缓冲数为 0 读不到日志。
 * 查询时间为左包含原则，即 [startTime, endTime) 。
 * @param logTypeName  日志类型名（在配置文件中定义的）
 * @param startTime    查询日志的开始时间
 * @param endTime      查询日志的结束时间
 * @param logLevel     查询日志的级别
 * @param userName     查询该用户的日志
 * @param bufferNum    日志反复器缓冲记录数
 * @return 结果集，日志反复器
*/

   public static LogIterator read(String logType, Date startTime,
      Date endTime, int logLevel, String userName, int bufferNum)
```

排除：对于实体类的 get，set 方法， 不用加注释， 其它类的 get，set 等简单方法也可以考虑不加注释。

异常抛出

对于方法内部用 throw 语句抛出的异常，必须在方法的注释中标明，对于所调用的其他方法所抛出的异常，选择主要的在注释中说明。对于非 RuntimeException ，即 throws 子句 声明会抛出的异常，必须在方法的注释中标明。

说明：异常注释用@exception 或@throws 表示，在 JavaDoc 中两者等价，但推荐用 @exception 标注 Runtime 异常，@throws 标注非 Runtime 异常。**异常的注释**必须说明该异常的含义及什么条件下抛出该异常。

注释与代码放置

注释应与其描述的代码相近，对代码的注释应放在其上方或右方（对单条语句的注释） 相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开。

注释与描述内容的缩排

注释与所描述内容进行同样的缩排。说明：可使程序排版整齐，并方便注释的阅读与理 解。示例：如下例子，排版不整齐，阅读稍感不方便。

```
public void example( ){

// 注释
   CodeBlock One
```

```
 // 注释
    CodeBlock Two
}
```

应改为如下布局。public
void example( )
{
    // 注释
    CodeBlock One

   // 注释
   CodeBlock Two
 }

对变量的定义和关键分支语句（条件分支、循环语句等）必须编写注释

说明：这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释帮助更好的理解程序，有时甚至优于看设计文档。

switch 语句下的 case 语句

对于 switch 语句下的 case 语句，如果因为特殊情况需要处理完一个 case 后进入下一个 case 处理，必须在该 case 语句处理完、下一个 case 语句前加上明确的注释。 如关键呼叫分支。

说明：这样比较清楚程序编写者的意图，有效防止无故遗漏 break 语句。**由 format.xml 文件限制，switch 下面注释写在 case 后面，如下。**

```
    //进行联系人搜索

    switch (range)
    {
        case SEARCH_RANGE_ALL:              // 搜索所有
            setSearchRangeIndicate(R.string.dialpad_all); break;
         case SEARCH_RANGE_FAVORITE:            //搜索爱好
            setSearchRangeIndicate(R.string.dialpad_favorete_number);
            break;
        default:
            break;

    }
```

不允许如下

```
switch (range)
{
    // 搜索所有
    case SEARCH_RANGE_ALL:
        setSearchRangeIndicate(R.string.dialpad_all);
        break;
    //搜索爱好
    case SEARCH_RANGE_FAVORITE:
        setSearchRangeIndicate(R.string.dialpad_favorete_number);
        break;
    default:
        break;
}
```

注释的内容

注释的内容要清楚、明了，含义准确，防止注释二义性。说明：错误的注释不但无益反 而有害。

注释语言

注释应该**使用中文(英语看项目组能力)和半角标点符号**，否则编译中会出现大量乱 码，影响编译问题定位。

Eclipse 工程用 utf-8 编码。

删除代码的注释

不要用注释保留废弃代码，现代的配置管理工具能恢复任意历史时刻的代码。可以通过注释说明修改信息。

3.2 建议

注释不应该出现在代码行中

避免在一行代码或表达式的中间插入注释。说明：除非必要，不应在代码或表达中间插 入注释，否则容易使代码可理解性变差。

简洁明了

说明：注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助 读者理解代码，防止没必要的重复注释信息。示例：如下注释意义不大。

```
// 如果 receiveFlag 为真
if (receiveFlag)
```

而如下的注释则给出了额外有用的信息。

```
// 如果从连结收到消息
if (receiveFlag)
```

在程序块的结束行右方加注释标记，以表明某程序块的结束

说明：当代码段较长，特别是多重嵌套时，这样做可以使代码更清晰，更便于阅读。示 例：参见如下例子。

```
if (...)
{
    program code1
   while   (index   <   MAX_INDEX)
      { program code2
     } // end of while (index < MAX_INDEX) // 指明该条 while 语句结束

} // end of if (...) //指明是哪条 if 语句结束
```

一些复杂的代码需要说明

示例：这里主要是对闰年算法的说明。

```
//1. 如果能被 4 整除，是闰年；
//2. 如果能被 100 整除，不是闰年；
//3. 如果能被 400 整除，是闰年。
```

4. 命名规范

4.1 规则

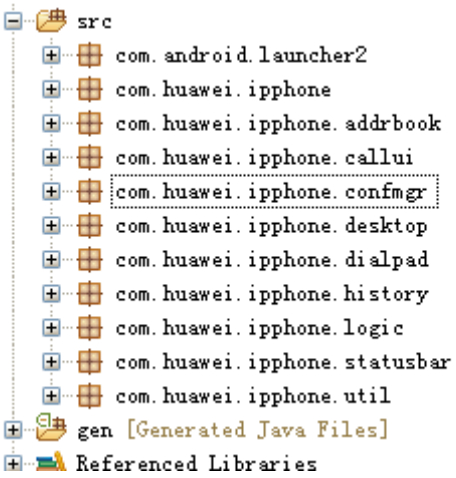
包名

完全自己开发的包，包名采用域后缀倒置的加上自定义的包名，采用小写字母。

包命名格式需要采取如下格式：com.company.ipphone.模块名

com.company.ipphone.模块名.小模块名

对于修改的代码，一律保持原先的包名不变化。



Setting - > apk

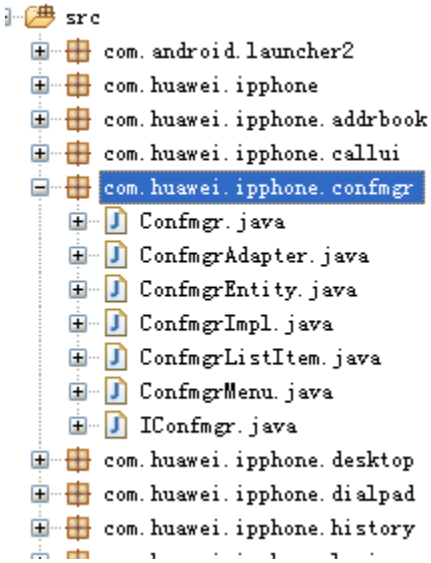
类名和接口名

使用类意义完整的英文描述，每个英文单词的首字母使用大写、其余字母使用小写的大小写混合法，尽量按模块名开头。

示例：OrderInformation, CustomerList, LogManager, LogConfig, SmpTransaction。

对于具体业务类，有以下约定：

类型	命名规定
模块的主体类，包含事件定义的类	&模块名相同
接口	I＋类名
实现类	Impl 结尾
适配类	Adapter 结尾
实体类	Entity 结尾
抽象类	Abs 开始



方法名

使用意义完整的英文描述：第一个单词的字母使用小写、剩余单词首字母大写其余字母小写的大小写混合法。函数的命名需要采用“动宾”结构，动词说明需要做的操作如 Add，Delete。宾语表示动作作用的对象。如 addContact，deleteMmsItem。

示例：

```
private void calculateRate();
public void addNewOrder();
```

方法get 和set

存取属性的方法采用 setter 和 getter 方法，动作方法采用动词和动宾结构。 取消 android 默认的 m 前缀变量命名。

格式：get + 非布尔属性名() is + 布尔属性名() set + 属性名() 动词() 动词 + 宾

语()

示例：

```
public String getType();
public boolean isFinished();
public void setVisible(boolean);
public void show();
public void addKeyListener(Listener);
```

属性名

属性名使用意义完整的英文描述：第一个单词必须是小写字母、剩余单词首字母大写其余字母小写的大小写混合法。属性名不能与方法名相同。命名规范示例：

```
private String customerName;
private String orderNumber;
```

循环变量统一用 i ， j ， k 三个。一重循环用 i，二重循环用 j，三重循环用 k。

常量名

使用**全大写的英文**描述，英文单词之间用下划线分隔开，并且使用 final static 修饰 示例：

```
public final static int MENU_MAX_VALUE = 1000;
public final static String DEFAULT_START_DATE = "2001-12-08";
```

属性名约定

属性名不可以和公有方法参数相同，如果相同，需要使用 this 来表示成员变量。属性名不能和局部变量相同。引用静态成员变量时使用类名引用。即使是在类内部使用，静态成员变量也需要使用类名进行引用。

示例如下：

```
public class Person
{
      private String Name;
      private static List PROPERTIES;

      public void setName (String name)
      {
          Name = name;
      }
```

```
        public void setProperties (List properties)
        {
            Person. PROPERTIES = properties;
        }
    }
```

图片&xml 文件的命名

统一按模块名字开头，进行分类，其中图片格式必须为 png 格式，图片， XML 的存放路径分别放在唯一路径,如下。

图片： drawable-sw800dp-mdpi

XML: layout-sw800dp

Drawable: 按钮的点击文件定义以 pressed 结尾，默认是 default 结尾。

控件名

控件 id	常用控件 ID 命名	前缀+对应的英文字
TextView	txt	txtUsername
Button	btn	btnLogin
Imagebuton	ibtn	
ListView	lv	
EditText	edt	
ScrollView	sv	
Imageview	iv	
Gallery	gallery	
LinearLayout	lt	
RelativeLayout	rl	
AbsoluteLayout	al	
Tablelayout	tl	
framelayout	fl	

layout 命名

layout 的 xml 命名是以名词或者名词词组的方式，单词间以下划线分割，全部单词采 用

小写。

1. Activity 布局命名：activity_功能模块_描述.xml

   例如：activity_clound_store.xml

2. Dialog 布局命名：dialog_描述.xml

   例如：dialog_error_msg.xml

3. PopupWindow 布局命名：ppw_描述.xml

   例如：popupwindow _info.xml

4. 列表项布局命名 listitem_描述.xml

   例如：listitem_city.xml

5. 包含项：include_模块.xml

   例如：include_warning_prompt.xml

4.2 建议

常用组件类的命名以组件名加上组件类型名结尾

示例：

Application 类型的，命名以 App 结尾——MainApp

Frame 类型的，命名以 Frame 结尾——TopoFrame

Panel 类型的，建议命名以 Panel 结尾——CreateCircuitPanel

Bean 类型的，建议命名以 Bean 结尾——DataAccessBean

EJB 类型的，建议命名以 EJB 结尾——DBProxyEJB

Applet 类型的，建议命名以 Applet 结尾——PictureShowApplet。

函数名过长的处理

如果函数名**超过 15 个字母**，可采用以去掉元音字母的方法或者以行业内约定俗成的缩写方式缩写函数名。示例：getCustomerInformation() 改为 getCustomerInfo()。

准确地确定成员函数的存取控制符号

不是必须使用 public 属性的，请使用 protected，不是必须使用 protected, 请使用 private。示例：protected void setUserName(),默认，private void calculateRate()。

含有集合意义的属性命名，尽量包含其复数的意义

示例：customers, orderItems。

5. 编码规范

5.1 规则

消除告警

所有告警尽量清除。

比较语句常量在前面

对于 if 之类判断语句，==, != 运算符，常量放在左边，如

```
if( MIN_VALUE == value)
```

其它运算符， 变量放左边， 如

```
 if( value >= MIN_VALUE)
```

函数参数合法性检查

应明确规定对接口方法参数的合法性检查应由方法的调用者负责还是由接口方法本身负责，缺省是由方法调用者负责。

说明：对于模块间接口方法的参数的合法性检查这一问题，往往有两个极端现象，即：要么是调用者和被调用者对参数均不作合法性检查，结果就遗漏了合法性检查这一必要的处理过程，造成问题隐患；要么就是调用者和被调用者均对参数进行合法性检查，这种情况虽不会造成问题，但产生了冗余代码，降低了效率。

Xml 文件使用规定

Layout_marginleft,layout_marginright 用 Layout_marginstart, Layout_marginend 代替，目的是解决阿拉伯语的对齐。

Paddingleft,paddingright 也用 paddingstart, paddingend 代替。

类功能明确，类实现精确

明确类的功能，精确（而不是近似）地实现类的设计。一个类仅实现一组相近的功能。说明：划分类的时候，应该尽量把逻辑处理、数据和显示分离，实现类功能的单一性。示例： 数

据类不能包含数据处理的逻辑。通信类不能包含显示处理的逻辑。

数据库操作、IO 操作等需要使用结束 close()的对象必须在 try-catch-finally 的 finally 中 close()

如果有多个 I/O 对象需要关闭,则需要分别对每个对象的 close()方法进行 try-catch，放置一个 I/O 对象关闭失败使其他对象都未关闭。

示例：

```
try
{
   statement1;
}
catch (IOException ioe)
{
   statement2;
}
finally
 {
   try
   {
       out.close();
   }
   catch(IOException ioe)
   {
       statement3;
    }
   try
   {
         in.close()
    }
    catch(IOException ioe)
    {
       statement4;
    }
}
```

Android 数据库Sqlite 实践

说明：通过 DBHelper 建立一个数据库和一个数据库连接（建议使用单例模式），多线程对这条连接进行 Write 操作时需要锁管理，对连接进行 Read 操作时不需要锁管理。当程序退 出时关闭数据库连接。通过 DBHelper 建立的数据库连接本身是支持多线程操作的。

数据库批处理

说明：Android 数据库如果需要进行批量处理，请使用数据库事务处理，以提高效率，

但必须保证在处理结束或异常情况下，事务能够关闭。

示例：

```
try
{
    db.beginTransaction();
    for()
    {
        db.action();
    }
}
catch(Exception e)
{

    e.printStacekTrace();
}
finally
{
    db.endTransaction(); db.close();
}
```

异常捕获后，对不处理的该异常进纪录日志或者ex.printStackTrace()

说明：若有特殊原因必须用注释加以说明。示例：

```
try
{

    //.... ...
}

//asldjljkasdklsdf

catch (IOException ioe) //asldjljkasdklsdf
{

    //asldjljkasdklsdf ioe.printStackTrace ();
}
catch(Throwable e)
{


}
```

运行期异常RuntimeException

运行期异常使用 RuntimeException 的子类来表示，不用在可能抛出异常的方法声明上加 throws 子句。非运行期异常是从 Exception 继承而来的，必须在方法声明上加 throws 子句。

说明：非运行期异常是由外界运行环境决定异常抛出条件的异常，例如文件操作，可能 受权限、磁盘空间大小的影响而失败，这种异常是程序本身无法避免的，需要调用者明确考虑该异常出现时该如何处理方法，因此非运行期异常必须有 throws 子句标出，不标出或者调 用者不捕获该类型异常都会导致编译失败，从而防止程序员本身疏忽。运行期异常是程序在运行过程中本身考虑不周导致的异常，例如传入错误的参数等。抛出运行期异常的目的是防 止异常扩散，导致定位困难。因此在做异常体系设计时要根据错误的性质合理选择自定义异常的继承关系。

所有异常都需要捕获，不能将原始信息直接抛给客户端用户呈现，以免信息安全。

运算符的优先级

注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。说明：防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。示例：

下列语句中的表达式

```
word = (high << 8) | low    (1)
boolean changingPages = (which Page != mCurrentPage); (2)
if ((a | b) < (c & d)) (3)
```

如果书写为

```
high << 8 | low
boolean changingPages = whichPage != mCurrentPage; a | b < c & d
```

（1）（2）虽然不会出错，但语句不易理解；（3）造成了判断条件出错。

避免使用不易理解的数字，用有意义的标识来替代

涉及物理状态或者含有物理意义的常量，**不应直接使用魔鬼数字**，必须用有意义的静态变量来代替。

示例：如下的程序可读性差。

```
if (state == 1)
{
   state = 1;
   ... // program code
```

```
}
```

应改为如下形式：

```
private final static int TRUNK_IDLE = 0;
private final static int TRUNK_BUSY = 1;
private final static int TRUNK_UNKNOWN = -1;
if (state == TRUNK_IDLE)
{
    state = TRUNK_BUSY;
    ... // program code
}
```

数组声明的时候使用 int[] index ，而不要使用 int index[]

说明：使用 int index[] 格式使程序的可读性较差示例：如下程序可读性差：

```
public int getIndex()[]{
    ....
}
```

如下程序可读性好：

```
public int[] getIndex()
{
    ....
}
```

代码的调试Log

调试代码的时候，不要使用 System.out 和 System.err 进行打印，应该使用一个包含 统一开关的测试类进行统一打印。

说明：代码发布的时候可以统一关闭调试代码，定位问题的时候又可以打开开关。

Log.error	出错日志，影响后面方法运行
Log.info	正常日志
Log.warn	告警，但不影响方法运行
Log.debug	调试日志
Log.operation	操作日志

5.2 建议

记录异常

记录异常不要保存 exception.getMessage()，而要记录 exception.toString()。示例：

NullPointException 抛出时常常描述为空，这样往往看不出是出了什么错。

一个方法不应抛出太多类型的异常

编码规范说明：如果程序中需要分类处理，则将异常根据分类组织成继承关系。如果确 实有很多异常类型首先考虑用异常描述来区别，throws/exception 子句标明的异常最好不要超过三个。

异常捕获尽量不要直接 catch (Exception ex) ，应该把异常细分处理。对于比较复杂 的情况，建议捕获全部细分的异常后在末尾增加 catch (Exception)，避免一些不易发生的 运行时异常未被捕获。

集合数据管理应用及时

集合中的数据如果不使用了应该及时释放，尤其是可重复使用的集合。说明：由于集合 保存了对象的句柄，虚拟机的垃圾收集器就不会回收。

源程序中关系较为紧密的代码应尽可能相邻

说明：便于程序阅读和查找。 示例：矩形的长与宽关系较密切，放在一起。

```
rect.length = 10;
rect.width = 5;
```

不要使用难懂的技巧性很高的语句，除非很有必要时

说明：高技巧语句不等于高效率的程序，实际上程序的效率关键在于算法。

6. 性能考虑

6.1 避免内存泄露

规则 1：使用数据库查询后的 Cursor 需要关闭。 示例：

```
Cursor c = null;
try
{
   c = db.query();
   if(c.moveToFirst())
   {
      while(!c.isAfterLast())
      {
          parserXXX(c);
          c.moveToNext();
      }
   }
}
catch(Exception e)
{
    e.printStackTrace();
}
finally
{
   if(c != null)
   {
      c.close();
   }
}
```

规则 2：ListView 使用构造 adapter 方式，并且使用缓存 contentView

示例：

```
@Override
public View getView(int position,View convertView, ViewGroup parent)
{
  View v = null;
  if (convertView == null)
  {
      v = newView(mContext,parent);
  }
  else
  {
```

```
        v = convertView;
    }
    bindView(v,mContext,position);
    return v;
}
```

规则 3：Bitmap 不使用时采用 recycle()回收资源 示例：

```
InputStream is = null;
Bitmap bitmap = null;
is = new FileInputStream(new File(filepath));
bitmap = BitmapFactory.decodeStream(is,null,opt);
//获取到 bitmap 使用完成后，调用 bitmap 的回收资源方法 bitmap.recycle();
```

规则 4：对于 Activity 的 View 中默认 BitmapDrawable 在不使用的时候及时释放。 示例：

```
public void recycleBackgroundRes(View viewBackground)
{
    if (viewBackground != null)
    {
        BitmapDrawable bd = (BitmapDrawable) viewBackground
                            .getBackground();
        viewBackground.setBackgroundResource(0);
        if (bd != null)
        {
            bd.setCallback(null);
            bd.getBitmap().recycle();
        }
    }
}
```

规则 5：避免将 Activity 的 Context 传入到生命周期大于这个 Context 的生命周期。说明：通常出现的场景有如下：

创建一个数据库表的 DbHelper（Context）；

传给一个内部类作为参数 new ClassB（Context）；

把自身当一个回调对象进行传递 registerCallBack（this）；

在内部起动线程把自身作为参数传递进去；

……

规则 6：避免 Activity 里的静态内部变量导致内存泄露，谨慎使用静态内部变量，虽然 在结构上它对于内存是有节省的。

示例：

```
Private static Drawables Background;
@Override
Protected void onCreate(Bundle state)
{
```

```
        super.onCreate(state);
        TextView label = new TextView(this);
        label.setText("Leaks are bad");
        if(sBackground == null)
        {
            sBackground=getDrawable(R.drawable.large_bitmap);
        }
        label.setBackgroundDrawable(sBackground);
        setContentView(label);
    }
```

说明：当一个 Drawable 连接到一个视图上时，视图被设置为Drawable 上的一个回调，在上面的代码片段中，这就意味着 Drawable 引用了 TextView，而 TextView 又引用了ActivityContext，ActivityContext 又进一步引用了更多的东西（依赖与你的代码）。当Activity销毁时其静态内部对象不会被销毁，但是此时它持有ActivityContext 的引用，所以将导致内 存泄露。

规则7：及时释放对象的引用。

示例：

```
        public class DemoActivity extends Activity
        {
            private Object obj;
            public void operation()
            {
                obj = new Obj();
                final Object o= obj;
                //[Mark]
                mHandler.post(new Runnable()
                {
                    public void run()
                    {
                        useObj(o);
                    }
                });
            }
        }
```

说明：如果operation 方法在把 obj 对象推送到其他线程处理而不再需要时，需要将obj 引用释放掉，否则其他线程处理完 obj 将无法释放 obj，因为 DemoActivity 还是持有 obj 的引用，引起内存的局部泄露。

解决方法：

```
        public class DemoActivity extends Activity
        {
            private Object obj;
            public void operation()
            {
                obj = new Obj();
                final Object
                o = obj;
```

```
                obj = null;
                mHandler.post(new Runnable()
                {
                    public void run()
                    {
                       useObj(o);
                    });
                }
            }
        }
```

规则 8：在 Activity 的 onPause()，onStop()中要适当的释放暂时不需要的资源。 BroadcastReceiver，ContentObserver，FileObserver 在 Activity onDestory 或者某类生命周期结束之后一定 unregister 掉，否则这个 Activity 类会被 system 强引用，不会被内存回收。导致内存泄露！

规则9：避免循环引用，这会导致内存泄露。

说明：比如a 中包含了b，b 包含了c，c 又包含a，这样只要一个对象存在其他肯定会 一直常驻内存，这要从逻辑上来分析是否需要这样的设计。

规则 10：对于生命周期不明确或者常驻的线程，谨慎传入生命周期比此线程短的外部 对象，会导致内存泄露。

建议1：使用具体对象优于使用接口。

说明：

Map myMap1 = new HashMap();

HashMap myMap2 = new HashMap();

哪一个更好呢？

一般来说明智的做法是使用 Map，因为它能够允许你改变 Map 接口执行上面的任何东西，但是这种“明智”的方法只是适用于常规的编程，对于嵌入式系统并不适合。通过接口引用来调用会花费2 倍以上的时间，相对于通过具体的引用进行虚拟函数的调用。

6.2 优化应用性能

规则 1：数据库查询结果的数据结构优化，只查询和解析所需要的字段，谨慎使用select *这样的查询语句。可以多写几种查询结果的数据结构对象 bean，而节省了内存，查询时间，解析结果。

规则 2：合理设计数据库表结构，一张表对应一种功能，尽量避免复杂的数据库查询，关联查询等。对于大数据量的操作启用事务。

规则3：Activitiy 在关键生命周期方法中如onCreate()和onResume()应当做尽可能少

的操 作。

说明:潜在地的耗时长的操作(如网络或数据库操作,或高耗费数学计算如改变位图大 小)应该在子线程里面完成（或以数据库操作为例，可以通过异步请求)。你的主线程应该

为子线程提供一个 Handler，以便子线程完成时可以提交回给主线程。以这种方式来设计你 的应用，将会允许你的主线程一直可以响应输入，以避免由 5 秒钟的输入事件超时导致的 ANR 对话。这些做法同样应该被其它任何显示 UI 的线程所效仿，因为它们属于同样类型的 超时。

规则4:不要在BroadcastReceiver 的onReceiver()方法里执行一些耗时的,异步的操作,否则会出现不可预知的异常。

说明：如果 BroadcastReceiver 的 OnReceiver（）方法不能在 10 毫秒内执行完成，Android 会认为该程序无响应。所以不要在BroadcastReceiver 的onReceiver()方法里执行一些耗时的， 异步的操作，否则会弹出ANR（Application No Response）的对话框。

如果确实需要 Broadcast 来完成一项比较耗时的操作，则可以考虑通过 Intent 启动一个Service来完成该操作。不应考虑使用新线程去完成耗时的操作,因为BroadcastReceiver 本身的周期很短,可能出现的情况是子线程还没有结束,BroadcastReceiver 就已经退出了。还有一种情况。如果BroadcastReceiver 所在的线程结束，虽然该进程内还有用户启动的新线程，但由于该进程内不包含任何活动组件，因此系统可能在内存紧张时优先结束该进程。这

样就可能导致 BroadcastReceiver 启动的子线程不能执行完
成。

规则 5：ListView & GridView 等控件使用时，必须要使用 Adapter 机制，并且在getView()的回调方法里，不要做耗时的操作。否则界面拖动会卡（虽然不会 ANR，但是不流畅的用户体验会非常差)

说明：

(1)耗时的操作务必要在 Activity 的子线程里完成后，再通知 Adapter 来处理数据，在 getView()的回调方法里避免耗时操作。

(2)优化拖动的数据加载触发点。比如要加载联系人的头像(或者从服务器拉取头像)，需要判断当前拖动是否结束。在拖动结束后再进行数据的加载。

规则 6：如果不可避免的需要用户等待数据的加载，为了好的用户体验，务必报告进度状态，或者提示用户正在处理。

说明：

(1)应该理解 UI 线程阻塞，和用户等待是两个不同的概念。

(2)这种场景一般出现在 Activity 需要为 ListView 的 Adapter 准备大量数据时，比如数据 库查询，数据运算等情况时，在 Activity 的独立线程还没有运行结束时，提示用户进度。或 者在进行网络请求，等待服务器响应数据来填充 UI 的情况等。

规则 7：避免在大量集中操作中做不必要的重复计算，能使用缓存的请使用缓存。避免 申请不必要的内存。

说明：

(1)在 for 循环，while 循环中重复的 new 对象，重复的计算可以缓存的数据。

(2)在常用的正则表达式匹配中，共用方法对正则表示重复的 compile，而对于指

定的正 则表达只需要compile 一次。

(3) 在常用的打印日志里，为了打印更多的信息，调用系统的调用堆栈信息。

……

规则8：使用对象自身的方法。

说明：比如当处理字符串的时候，尽可能多的使用诸如 String.indexOf(), String.lastIndexOf() 这样对象自身带有的方法。因为这些方法使用 C/C++来实现的，要比在一个 java 循环中做同样的事情快10-100 倍。

除了通常的那些原因外，考虑到系统空闲时会用汇编代码调用来替代 library 方法，这 可能比JIT（运行时编译执行技术）中生成的等价的最好的Java 代码还要好。典型的例子就 是String.indexOf，Dalvik 用内部内联来替代。同样的，System.arraycopy 方法在有JIT 的Nexus One 上，自行编码的循环快9 倍。

规则 9：缓冲属性调用：当你不止一次的调用某个实例时，直接本地化这个实例，把这个实例中的某些值赋给一个本地变量。因为访问对象属性要比访问本地变量慢得多。

示例：你不应该这样写你的代码：

```
for (int i = 0; i < this.mCount; i++)
dumpItem(this.mItems[i]);
```

而是应该这样写：

```
int count = this.mCount;
Item[] items = this.mItems;
for (int i = 0; i < count; i++)
dumpItems(items[i]);
```

规则10：尽可能避免使用内在的Get、Set 方法。

说明：C++编程语言，通常会使用 Get 方法(例如 i = getCount())去取代直接访问这个属 性(i = mCount)。这在C++编程里面是一个很好的习惯，因为编译器会把访问方式设置为 Inline，并且如果想约束或调试属性访问，你只需要在任何时候添加一些代码。

在Android 编程中，这不是一个很不好的主意。虚方法的调用会产生很多代价，比实例属性查询的代价还要多。我们应该在外部调用时使用 Get 和 Set 函数，但是在内部调用时， 我们应该直接调用。

规则 11：声明 Final 常量 说明：我们可以看看下面一个类的声明：

```
static int intVal = 42;
static String strVal = "Hello, world!";
```

当一个类第一次使用时，编译器会调用一个类初始化方法<clinit>，这个方法将42 存入 变量 intVal，并且为strVal 在类文件字符串常量表中提取一个引用，当这些值在后面引用时，就会直接属性调用。

我们可以用关键字“final”来改进代码：

```
static final int intVal = 42;

static final String strVal = "Hello, world!";
```

这个类将不会调用<clinit>方法，因为这些常量直接写入了类文件静态属性初始化中，这 个初始化直接由虚拟机来处理。代码访问intVal 将会使用Integer 类型的42，访问strVal 将 使用相对节省的“字符串常量”来替代一个属性调用。

规则12：使用改进的 for 循环语法

说明：改进for 循环(有时被称为“for-each”循环)能够用于实现了iterable 接口的集合 类及数组中。在集合类中，迭代器让接口调用 hasNext()和 next()方法。在ArrayList 中，手写的计数循环迭代要快3 倍(无论有没有JIT)，但其他集合类中，改进的for 循环语法和迭代器 具有相同的效率。

所以，优先采用改进 for 循环，但在性能要求苛刻的 ArrayList 迭代中，可以考虑采用手 写计数循环（不建议）。

规则 13：避免枚举类型 说明：列举类型非常好用，当考虑到尺寸和速度的时候，就会显得代价很高，例如：

```
public class Foo
{
    public enum Shrubbery
    { GROUND, CRAWLING, HANGING }
}
```

这会转变成为一个 900 字节的 class 文件(Foo$Shrubbery.class)。第一次使用时，类的 初始化要在调用方法去描述列举的每一项，每一个对象都要有它自身的静态空间，整个被储存在一个数组里面（一个叫做“$VALUE”的静态数组）。那是一大堆的代码和数据，仅仅是为了三个整数值。

```
Shrubbery shrub = Shrubbery.GROUND;
```

这会引起一个静态属性的调用，如果GROUND 是一个静态的Final 变量，编译器会把它 当做一个常数嵌套在代码里面。

规则14：避免浮点类型的使用

说明：在奔腾CPU 发布之前，游戏作者尽可能的使用Integer 类型的数学函数是很正常 的。在奔腾处理器里面，浮点数的处理变为它一个突出的特点，并且浮点数与整数的交互使用相比单独使用整数来说，前者会使你的游戏运行的更快，一般的在桌面电脑上面我们可以自由的使用浮点数。

不幸的是，嵌入式的处理器通常并不支持浮点数的处理，因此所有的“float”和“double”操作都是通过软件进行的，一些基本的浮点数的操作就需要花费毫秒级的时间。 即使是整数，一些芯片也只有乘法而没有除法。在这些情况下，整数的除法和取模操作都是通过软件实现。当你创建一个Hash 表或者进行大量的数学运算时，这都是你要考虑的。

规则 15：大量字符串的“相加等于”操作如果不涉及线程安全应该使用 StringBuilder， 如果有线程安全的要求应该使用 StringBuffer。

说明：大量的 String 相加等于处理性能消耗较多，大量一般指 5 次“+=”以上或者在循 环中进行字符串“+=”操作，StringBuilder 的性能优于 StringBuffer，但是是非线程安全的。

一般在网络请求的组包时使用。 示例：

不推荐：

```
String str = “”;
str += ”a”;
str += ”b”;
str += ”c”;
str += ”d”;
str += ”e”;
```

推荐：

```
StringBuilder sb = new StringBuilder();
sb.append(“aa”);
sb.append(“bb”);
sb.append(“cc”);
```

6.3 优化应用省电

规则 1：大数据量的传输实施流量压缩。当前 HTTP 可以采用 gzip 压缩方法，XMPP 可以采用 zlib 压缩方法。

规则 2：在需要网络连接的程序中，首先检查网络连接是否正常，如果没有网络连接，那么就不需要执行相应的程序。同时对于多次相同的网络请求应该要互斥，及不应该存在重复操作和线程。

规则 3：使用效率高的解析方法。通过测试发现，目前主流的数据格式，使用树形解析（如DOM）低于流的方式解析（SAX Simple API for XML）。

规则 4：不使用的服务要及时停止。程序后台如果有一个 service 不停的去服务器上更 新数据，在不更新数据的时候就让它 sleep，这种方式是非常耗电的，通常情况下，我们可 以使用 AlarmManager 来定时启动服务。

建议 1：使用效率高的数据结构。当前测试发现，JSON 性能优于 XML，针对移动设备，最好能使用 JSON 数据格式为佳。

建议 2：对定位要求不高尽量不要使用 GPS 定位，可以使用 wifi 和移动网络定位即可。GPS 定位消耗的电量远远高于移动网络定位。

6.4 优化应用安全

规则 1：密码等重要数据在数据库存储必须使用密文。说明：对密码明文进行加密，可以参考采用 AES128 位加密，或者其他的加密算法也是可以的。

规则 2：配置文件的读写权限必须为私有。

说明：xml 配置文件的权限设置为 Context.MODE_PRIVATE

规则 3：使用安全的 HTTP，XMPP 的连接。 解决方案：

(1)对 HTTP 和 XMPP 进行 TLS 加密

(2)对 HTTPS 和 XMPP 的加密进行证书验证（权威机构证书是收费的，根据局点的需求）

规则 4：不允许在 AndroidManifest.xml 申明额外的权限。 说明：因为一个应用的功能是一定的，使用到的权限也应该是一定的，如果在代码中申

明了额外的权限，用户会担心应用还在偷偷做其他事情。

规则 5：对于不需要跨进程启动的对象，不要在 AndroidManifest.xml 中申明<intent-filter>， 否则可能会导致其他应用的攻击

规则 6：SQL 注入的安全问题。 解决方法：将涉及到动态参数（服务器传输的数据或者用户输入的可能被 SQL 注入的数据）的 SQL 语句修改为预编译方式，程序中定义的常量参数（例如固定参数类型等不可 被修改数据）不涉及。

如将以下 SQL 语句，msgId，user_id 都可能被注入：

```
String sql = "SELECT " + ID + " FROM " + TABLE_NAME + " WHERE "
 + EMAIL_ID + "='" + msgId + "'" + " AND " + USER_ID + "='"
 + AASManager.getInstence().getUserInfo().getUserid() + "'";
```

修改为以下形式，通过系统预编译方式可杜绝注入：

```
String sql = "SELECT " + ID + " FROM " + TABLE_NAME + " WHERE " + EMAIL_ID + "=?" +
" AND " + USER_ID + "=?";
```

规则7：日志不允许打印重要信息，如密码等。

说明：发布版本关闭日志，如果支持日志开关，必须保证重要信息不能够打印。