Android开发规范

1 介绍

1.1 目的

- 减少维护花费。
- 提高可读性。
- 加快工作交接。
- 减少名字增生。
- 降低缺陷引入的机会。

1.2 术语和定义

强制：编程时必须遵守的规定，含有强制字样或字体用加粗式样标注。

推荐：编程时推荐遵守的规定，字体用普通式样标注。

2 文件组织

避免超过 2000 行的源文件。

2.1 Java 包和源文件

每个 Java 源文件都包含一个单一的公共类或接口。若私有类和接口与一个公共类相关联，可以将它们和公共类放入同一个源文件。公共类必须是这个文件中的第一个类或接口。

Java 源文件还遵循以下规则：

- 开头注释
- 包和引入语句
- 类和接口声明

2.1.1 开头注释

所有的源文件都应该在开头有一个的注释，其中列出版本信息、日期和版权声明。

```
/*
 * Copyright (c) 2014 SIMCOM, Inc.
 * All Rights Reserved.
 * SIMCOM Proprietary and Confidential.
 */
```

2.1.2 包和引入语句

在多数 Java 源文件中，第一个非注释行是包语句，在它之后可以跟引入语句。

```
package com.android.sim;

import java.io.IOException;
```

在导入包时当完全限制代码所使用的类的名字，尽量少用通配符的方式，但导入一些通用包，或用到一个包下大部分类时，则可是使用通配符方式。同一包中的类在导入时应声明在一起，无效的未使用到的引用要即时删除。

2.1.3 类和接口声明

下表（强制）描述了类和接口声明的各个部分以及它们出现的先后次序。

类/接口声明的各部分	说明
类/接口文档注释(/**……*/)	类和接口应该有标准 Javadoc 注释。
类或接口的声明	类名和接口的第一个字符大写，每个单词的首字母大写，中间可以有下划线。
类/接口实现的长度限制	限制一个匿名内部类的长度不要超过 80 行。
类的(静态)变量	首先是类的公共变量，随后是保护变量，再后是包一级别的变量（没有访问修饰符，access modifier），最后是私有变量。静态 final 变量应全部大写，中间可以有下划线。
实例变量	
构造器	构造器的代码长度（不计空行），不应超过 200 行。
方法	方法名应为动词或动宾短语，首字母小写，其后每个单词首字母大写，方法的参数不要超过 8个，参数名字必须和变量的命名规范一致，public 的方法之前应该有 Javadoc 注释，方法之后的大括号位于行尾。方法应该保持简短和重点突出，对方法的代码长度并没有硬性的限制。如果方法代码超过了 40 行，就该考虑是否可以在不损害程序结构的前提下进行分拆。

3 缩进排版

4 个空格作为缩进排版的一个单位，不使用制表符 tab。

8 个空格作为换行后的缩进，包括函数调用和赋值。

```
Instrument i =
        someLongexpression_r(that, NotFit, on, one, line);    // 推荐

Instrument i =
    someLongexpression_r(that, NotFit, on, one, line);        // 避免
```

3.1 行长度

尽量避免一行的长度超过 100 个字符。

例外：如果注释行包含了超过 100 个字符的命令示例或者 url 文字，为了便于剪切和复制，其长度可以超过

100 个字符。

例外：import 行可以超过限制，因为很少有人会去阅读它。这也简化了编程工具的写入操作。

3.2 括号

大括号不单独占用一行，应紧接着上一行书写。

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

我们需要用大括号来包裹条件语句块。不过也有例外，如果整个条件语句块（条件和语句本身）都能容纳在一行内，也可以（但不是必须）把它们放入同一行中。也就是说，这是合法的：

```
if (condition) {
    body();
}                           // 推荐
if (condition) body();      // 避免
if (condition)
    body();                 // 错误
```

3.3 换行

当一个表达式无法容纳在一行内时，可以依据如下一般规则断开：

- 在一个逗号后面断开。
- 在一个操作符前面断开。
- 宁可选择较高级别的断开，而非较低级别的断开。
- 新的一行应该与上一行同一级别表达式的开头处对齐。
- 如果以上规则导致你的代码混乱或者使你的代码都堆挤在右边，那就代之以缩进 8个空格。

以下是断开方法调用的一些例子：

```
someMethod(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);

var = someMethod1(longExpression1,
                      someMethod2(longExpression2,
                                  longExpression3));
```

以下是两个断开算术表达式的例子。前者更好，因为断开处位于括号表达式的外边，这是个较高级别的断开。

```
long1 = long2 * (long3 + long4 - long5)
```

```
        + 4 * longname6;             // 推荐

long1 = long2 * (long3 + long4
                 - long5) + 4 * long6; // 避免
```

以下是两个缩进方法声明的例子。前者是常规情形。后者若使用常规的缩进方式将会使第二行和第三行移得很靠右，所以代之以缩进 8 个空格。

```
// 常规缩进
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
             Object andStillAnother) {
...
}
// 为避免太靠右，用8个空格缩进
private static synchronized     horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
...
}
```

if 语句的换行通常使用 8 个空格的规则，因为常规缩进（4 个空格）会使语句体看起来比较费劲。比如：

```
// 请不要使用这种缩进
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) {    // 不好的缩进
    doSomethingAboutIt();               // 该行和if条件处于同一级
}                                       // 避免

// 使用这种缩进
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}                                       // 推荐

// 或这种
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}                                       // 推荐
```

这里有三种可行的方法用于处理三元运算表达式：

```
alpha = (aLongBooleanExpression) ? beta : gamma;
alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

4 注释

Java 程序有两类注释：实现注释（使用/*...*/和//界定的注释）和文档注释（由/**...*/界定，可通过 javadoc 工具转换成 html 文件）。

实现注释用以注释代码或者实现细节。文档注释从实现自由的角度描述代码的规范。它可以被那些手头没有源码的开发人员读懂。

注释应被用来给出代码的总括，并提供代码自身没有提供的附加信息。注释应该仅包含与阅读和理解程序有关的信息。例如，相应的包如何被建立或位于哪个目录下之类的信息不应包括在注释中。

在注释里，对设计决策中重要的或者不是显而易见的地方进行说明是可以的，但应避免提供代码中已清晰表达出来的重复信息。

注释不应写在用星号或其他字符画出来的大框里。注释不应包括诸如制表符和回退符之类的特殊字符。

使用 Javadoc 标准注释，每个文件的开头都应该有一句版权说明。然后下面应该是package 包语句和 import 语句，每个语句块之间用空行分隔。然后是类或接口的定义。在Javadoc 注释中，应描述类或接口的用途。

```
/*
 * Copyright (c) 2014 SIMCOM, Inc.
 * All Rights Reserved.
 * SIMCOM Proprietary and Confidential.
 */

package com.android.internal.foo;

import android.os.Blah;
import android.view.Yada;

import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * 一句话功能描述
 * 功能详细描述
 * @see 相关类/方法
 * @deprecated
 */

public class Foo {
    ...
}
```

每个类和自建的 public 方法必须包含 **Javadoc** 注释，注释至少要包含描述该类或方法用途的语句。并且该语句应该用第三人称的动词形式来开头。

```
/** Returns the correctly rounded positive square root of a double
value. */
static double sqrt(double a) {
    ...
}

or

/**
 * Constructs a new String by converting the specified array of
 * bytes using the platform's default character encoding.
 */
public String(byte[] bytes) {
    ...
}
```

如果所有的 Javadoc 都会写成“sets Foo”，对于那些无关紧要的类似 setFoo()的 get和 set 语句是不必撰写

Javadoc 的。如果方法执行了比较复杂的操作（比如执行强制约束或者产生很重要的副作用），那就必须进行注释。如果"Foo"属性的意义不容易理解，也应该进行注释。

无论是 public 的还是其它类型的，所有自建的方法都将受益于 Javadoc。public 的方法是 API 的组成部分，因此更需要 Javadoc。

4.1 实现注释的格式

程序可以有 4 种实现注释的风格：块、单行、尾端和行末。

4.1.1 块注释

块注释通常用于提供对文件，方法，数据结构和算法的描述。块注释被置于每个文件的开始处以及每个方法之前。它们也可以用于其他地方，比如方法内部。在功能和方法内部的块注释应该和它们所描述的代码具有一样的缩进格式。

块注释之首应该有一个空行，用于块注释和代码分割开来，比如：

```
/*
 * Here is a block comment.
 */
```

块注释可以以/*-开头，这样indent(1)就可以将之识别为一个代码块的开始，而不是重排它。

```
/*-
  * Here is a block comment with some very special
  * formatting that I want indent(1) to ignore.
  *
  *    one
  *        two
  *            three
  */
```

4.1.2 单行注释

短注释可以显示在一行内，并与其后的代码具有一样的缩进层级。如果一个注释不能在一行内写完，就该采用块注释（参见"块注释"）。单行注释之前应该有一个空行。以下是一个Java代码中单行注释的例子：

```
        if (condition) {
            /* Handle the condition. */
            …
        }
```

4.1.3 尾端注释

极短的注释可以与它们所要描述的代码位于同一行，但是应该有足够的空白来分开代码和注释。若有多个短注释出现于大量代码中，它们应该具有相同的缩进。

以下是一个Java代码中尾端注释的例子：

```
        if (a == 2) {
            return true;                        /* special case */
        } else {
            return isPrime(a);  /* works only for odd a */
        }
```

4.1.4 行末注释

注释界定符"//"，可以注释掉整行或者一行中的一部分。它一般不用于连续多行的注释文本；然而，它可以用来注释掉连续多行的代码段。以下是所有三种风格的例子：

```
        if (foo > 1) {
            // Do a double-flip.
            ...
        } else {
            return false;               // Explain why here.
        }

        //if (bar > 1) {
        //
        //    // Do a triple-flip.
        //    ...
        //} else {
        //    return false;
        //}
```

4.2 文档注释

文档注释描述 Java 的类、接口、构造器，方法，以及字段。每个文档注释都会被置于注释定界符/**...*/之中，一个注释对应一个类、接口或成员。

该注释应位于声明之前：

```
/**
 * The Example class providers ...
 */
public class Example {
    ...
}
```

注意顶层的类和接口是不缩进的，而其成员是缩进的。描述类和接口的文档注释的第一行（/** ）不需缩进，随后的文档注释每行都缩进 1 格（使星号纵向对齐）。成员，包括构造函数在内，其文档注释的第一行缩进 4 格，随后每行都缩进 5 格。

若你想给出有关类、接口、变量或方法的信息，而这些信息又不适合写在文档中，则可使用实现块注释或紧跟在声明后面的单行注释。例如，有关一个类实现的细节，应放入紧跟在类声明后面的实现块注释中，而不是放在文档注释中。

文档注释不能放在一个方法或构造器的定义块中，因为 Java 会将位于文档注释之后的第一个声明与其相关联。

4.2.1 类注释

在类、接口定义之前当对其进行注释，包括类、接口的目的、作用、功能、继承于何种 父类，实现的接口、实

现的算法、使用方法、示例程序等。

```
/**
 * 一句话功能描述
 * 功能详细描述
 * @see 相关类/方法
 * @deprecated
 */
```

4.2.2 方法注释

据标准Javadoc规范对方法进行注释，以明确该方法功能、作用、各参数含义以及返回值等。复杂的算法用/**/在方法内注解出。

- 参数注释时当注明其取值范围等。
- 返回值当注释出失败、错误、异常时的返回情况。
- 异常当注释出什么情况、什么时候、什么条件下会引发什么样的异常。

```
/**
 * 一句话方法描述
 * 方法详细描述
 * @param 参数名 参数描述
 * @param 参数名2 参数描述
 * @return 返回值类型说明
 * @throws Exception 异常说明
 * @see 类/方法/成员
*/
```

4.2.3 类成员变量和常量注释

成员变量和常量需要使用javadoc形式的注释，以说明当前变量或常量的含义。

```
/**
 * 成员变量描述
 */
private String test;

/** 成员变量描述 */
private int hello;
```

5 声明

5.1 每行声明变量的数量

推荐一行一个声明，因为这样以利于写注释。

```
int level;       // indentation level
int size;        // size of table
```

注意：上面的例子中，在类型和标识符之间放了一个空格，另一种被允许的替代方式是使用制表符。

```
int             level;          // indentation level
int             size;           // size of table
char            username;       // username
```

5.2 初始化

尽量在声明局部变量的同时初始化。唯一不这么做的理由是变量的初始值依赖于某些先前发生的计算。

5.3 布局

只在代码块的开始处声明变量（一个块是指任何被包含在大括号"{"和"}"中间的代码）。不要在首次用到该变量时才声明之。这会把注意力不集中的程序员搞糊涂，同时会妨碍代码在该作用域内的可移植性。

```
void myMethod() {
int int1 = 0;           // beginning of method block

    if (condition) {
        int int2 = 0;   // beginning of “if” block
        ...
    }
}
```

避免声明的局部变量覆盖上一级声明的变量。例如，不要在内部代码块中声明相同的变量名：

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;     // 避免
        ...
   }
   ...
}
```

5.4 类和接口的声明

当编写类和接口是，应该（强制）遵守以下格式规则：

- 在方法名与其参数列表之前的左括号"("间不要有空格。
- 左大括号"{"位于声明语句同行的末尾。
- 右大括号"}"另起一行，与相应的声明语句对齐。除非是一个空语句，"}"应紧跟在"{"之后。
- 方法与方法之间以空行分隔。

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i,       int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}
```

```
    ...
}
```

6 语句

6.1 简单语句

每行之多包含一条语句，例如：

```
    argv++;             // 推荐
    argc--;             // 推荐
    argv++; argc--;     // 避免
```

6.2 复合语句

复合语句是包含在大括号中的语句序列，形如"{ 语句 }"。例如下面各段。

- 被括其中的语句应该较之复合语句缩进一个层次。
- 左大括号"{"应位于复合语句起始行的行尾，右大括号"}"应另起一行并与复合语句首行对齐。
- 大括号可以被用于所有语句，包括单个语句，只要这些语句是诸如 if-else 或 for 控制结构的一部分。这样便于添加语句而无需担心由于忘了加括号而引入 bug。

6.3 返回语句

一个带返回值的return语句不使用小括号"()"，除非它们以某种方式使返回值更为显 见。 例如：

```
return;

return myDisk.size();                   // 避免

return (size ? size : defaultSize);     // 避免
```

6.4 if, if-else, if else-if else 语句

if-else语句应该具有如下格式：

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
```

```
    statements;
} else{
    statements;
}
```

注意：if语句总是用”{“和”}“括起来，避免使用如下容易引起错误的格式：

```
if (condition)  // 避免
    statement;
```

6.5 for 语句

一个for语句应该具有如下格式：

```
for (initialization; condition; update) {
    statements;
}
```

当在for语句的初始化或更新子句中使用逗号时，避免因使用三个以上变量，而导致复杂度提高。若需要，可以在for循环之前(为初始化子句)或for循环末尾(为更新子句) 使用单独的语句。

6.6 while 语句

一个while语句应该具有如下格式：

```
while (condition) {
    statements;
}
```

6.7 do-while 语句

```
do {
    statements;
} while (condition);
```

6.8 switch 语句

一个switch语句应该具有如下格式：

```
switch (condition) {
    case ABC:
        statements;
        /* falls through */
    case DEF:
        statements;
        break;

    case XYZ:
        statements;
        break;
```

```
    default:
        statements;
        break;
}
```

每当一个case顺着往下执行时（因为没有break语句），通常应在break语句的位置添加注释。上面的示例代码中就包含注释/* falls through */。

6.9 try-catch 语句

一个try-catch语句应该具有如下格式：

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}

try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

7 空白

7.1 空行

空行将逻辑相关的代码段分隔开，以提高可读性。下列情况应该总是使用空行：

- 一个源文件的两个片段（section）之间。
- 类声明和接口声明之间。
- 两个方法之间。
- 方法内的局部变量和方法的第一条语句之间。
- 块注释或单行注释之前。
- 一个方法内的两个逻辑段之间，用以提高可读性。

7.2 空格

下列情况应该使用空格：

- 一个紧跟着括号的关键字应该被空格分开。例如：

```
while (true) {
    ...
}
```

- 空白应该位于参数列表中逗号的后面。
- 所有的二元运算符，除了“.”，都应该使用空格将之与操作数分开。一元操作符和操作数之间不应该加空格，比如：负号(“-”)、自增(“++”)和自减(“--”)。例如：

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
    n++;
}
```

- for语句中的表达式应该被空格分开。

```
for (expr1; expr2; expr3)
```

- 强制转型后应该跟一个空格。

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

8 命名规范

命名规范使程序更易读，从而更易于理解。它们也可以提供一些有关标识符功能的信息，以助于理解代码。

8.1 包命名

包名由全部小写字母组成，包名的前缀以com开头，包名后续部分的格式为：

[域名反转].[项目名].[模块名].[子模块名]...

例如：com.android.sim.message.sms

8.2 类和接口命名

类名是个一名词，采用大小写混合的方式，每个单词的首字母大写。尽量使你的类名简洁而富于描述。使用完整单词，或约定成俗并且使用广泛的缩写词，如url，html，接口和类名规则一至但要使用I前缀。

继承自系统组件类的命名，后缀必须明确表示出系统组件的类别，Activity类后缀使用Activity，Service类后缀使用Service，BroadcaseReceiver类后缀使用Receiver，ContentProvider使用Provider。

8.3 方法命名

方法名是一个动词或者动名词结构，采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写，即驼峰命名规则。

以它做什么来命名，而不是以它怎样做命名。如doUpdate()，isNumber()。

8.4 变量命名

第一个单词的首字母小写，其后单词的首字母大写。变量名不应以下划线或美元符号开头，尽管这在语法上是允许的。变量名的选用应该易于记忆，即，能够指出其用途。尽量避免单个字符的变量名，除非是一次性的临时变量。临时变量通常被取名为 i，j，k，m 和 n，它们一般用于整型；c，d，e，它们一般用于字符型。

其中系统控件中在后缀中体现控件类型，如下所示：

组件名称	简写	组件名称	简写
Button	Btn	RadioButton	Rbtn
ImageButton	Ibtn	TextView	Tv
ImageView	Iv	ListView	Lv
ProgressBar	Pbar	EditText	Et
ScrollView	Sv	CheckBox	Cb
RelativeLayout	Rly	LinearLayout	Lly
TableLayout	Tly	LinearLayout	Aly
FrameLayout	Fly		

非 public 的、非 static 的字段名称以 m 开头。

static 字段名称以 s 开头。

其它字段以小写字母开头。

```
public  class MyClass {
    public int publicField;
    private static MyClass sSingleton;
    int mPackagePrivate;
    private int mPrivate;
    protected int mProtected;
}
```

8.5 常量命名

类常量的声明，应该全部大写，单词间用下划线隔开。

```
static final int MIN_WIDTH = 4;
static final int MAX_WIDTH = 999;
static final int GET_THE_CPU = 1;
```

8.6 异常命名

自定义异常的命名必须以Exception为结尾，已明确标示为一个异常。

异常实例一般使用e、ex等，在多个异常时使用该异常名或简写加E，Ex等组成，如：SQLEx，ActionEx。

8.7 Layout 命名

命名必须以全部单词小写，单词间以下划线分割，并且尽可能的使用名词或名词组，即使用 模块名_功能名称来命名。

```
addressbook_list.xml    // 推荐
list_addressbook.xml    // 避免
```

8.8 资源 ID 命名

layout中所使用的id命名必须以全部单词小写，单词间以下划线分割，并且尽可能的使用名词或名词组，并且要求能够通过id直接理解当前组件要实现的功能。

```
EditText名 @+id/book_name_edit   // 推荐
EditText名 @+id/textbookname     // 避免
```

8.9 Activity 中 View 命名

采用大小写混合模式，第一个单词首字母小写，其余单词首字母大写最后一个单词为该View 类型的缩写,格式如下:

逻辑名+View 类型缩写（View 缩写参照 8.4 组件名称缩写表）。

Button homeBtn

8.10 strings.xml 中 ID 命名

命名必须以全部单词小写，单词间以下划线分割，并且尽可能的使用名词或名词组，格式如下:

- activity名称_功能模块名称_逻辑名称 或
- activity名称_逻辑名称 或
- common_逻辑名称

逻辑名称多个单词用下划线连接，同时使用activity名称注释。

```
main_menu_about
main_title
common_exit
common_app_name
```

8.11 资源命名

layout中使用的所有资源（如drawable，style等）命名必须以全部单词小写，单词间以下划线分割，并且尽可能的使用名词或名词组，即使用模块名_用途来命名。如果为公共资源，如分割线等，则直接用用途来命名。如：menu_icon_navigate.png

9 编程规范

9.1 单位规范

在使用单位时，如果没有特殊情况，一律使用 sp 作为文字大小的单位,将 dip 作为其他元素的单位。因为这两个单位是与设备分辨率无关的，能够解决在不同分辨率的设备上显示效果不同的问题。另外，在编码中定义控件的 margin 或 padding 属性时，SDK 里面并没有提供 dip 单位的 api 设置接口，而是提供了默认的 px 设置。

```
Button btn = new Button(context);
LayoutParams lp = new
        LayoutParams(LayoutParams.FILL_PARENT,LayoutParams.FILL_PARENT);
lp.setMargins(0, 0, 0, 0);
btn.setTextSize(12);
btn.setPadding(0, 0, 0, 0);
```

这个时候，一般在设置 margin 和 padding 时，应该对要设置的 dip 值转换为 px 单位，而字体的大小设置中，系统默认给出了 sp 的单位，所以可以不用进行转换。转换的方法参考下面的代码：

```
/**
 * 把dip单位转成px单位
 * @param context context对象
 * @param dip dip数值
 * @return dip对应的px值
 */
public static int formatDipToPx(Context context, int dip) {
    DisplayMetrics dm = new DisplayMetrics();
    ((Activity)context).getWindowManager().getDefaultDisplay().getMetrics(dm);
    int dip = (int) Math.ceil(dip * dm.density);
    return dip;
}
```

9.2 引用类变量和类方法

避免用一个对象访问一个类的静态变量和方法。应该用类名替代。

```
classMethod();               // 推荐
AClass.classMethod();        // 推荐
anObject.classMethod();      // 避免
```

9.3 常量

位于 for 循环中作为计数器值的数字常量，除了-1,0 和 1 之外，不应被直接写入代码。

9.4 变量赋值

避免在一个语句中给多个变量赋相同的值，它很难读懂。

9.5 信令类

如果类只是用来作为信息传递的中间变量，则应该声明为信令类，即所有的全局变量都是 final 类型，在初始化

时赋值。

```
private final String name;
public Foo(String str) {
    name = str;
}
public Foo(String str ) {
    this.str = str;                    // 避免在构造函数中出现this引用
}
```

9.6 不要忽略异常

有时，完全忽略异常是非常诱人的。

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

绝对不要这么做。也许你会认为：你的代码永远不会碰到这种出错的情况，或者处理异常并不重要，可类似上述忽略异常的代码将会在代码中埋下一颗地雷，说不定哪天它就会炸到某个人了。你必须在代码中以某种规矩来处理所有的异常。根据情况的不同，处理的方式也会不一样。 可接受的替代方案包括（按照推荐顺序）： 向方法的调用者抛出异常。

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

根据抽象级别抛出新的异常。

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

默默地处理错误并在 catch {} 语句块中替换为合适的值。

```
/**     设置端口。假如值不是数字则用80代替 */
void setServerPort(String value) {
    try {
        serverPort = Integer.parseInt(value);
    }   catch (NumberFormatException e) {
        serverPort = 80;        // 服务默认端口
    }
}
```

捕获异常并抛出一个新的 RuntimeException。这种做法比较危险：只有确信发生该错误时最合适的做法就是崩溃，才会这么做。

```
/**     设置端口，假如值不是数字则程序终止。 */
```

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

请记住，最初的异常是传递给构造方法的 RuntimeException。如果代码必须在 Java 1.3 版本下编译，需要忽略该异常。 最后一招：如果确信忽略异常比较合适，那就忽略吧，但必须把理想的原因注释出来。

```
/**     假如值不是数字则使用原来的端口号。 */
void setServerPort(String value) {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) {
        // 方法记录：无视无效的用户输入。
        // 服务端口不会被改变。
    }
}
```

9.7 不要捕获顶级的 Exception

有时在捕获 Exception 时偷懒也是很吸引人的，类似如下的处理方式：

```
try {
    someComplicatedIOFunction();        // 可能抛出IOException
    someComplicatedParsingFunction();   // 可能抛出ParsingException
    someComplicatedSecurityFunction();  // 可能抛出SecurityException
    // 其他可以抛出Exception的代码
} catch (Exception e) {                 // 一次性捕获所有exceptions
    handleError();                      // 只有一个通用的处理方法!
}
```

不要这么做。绝大部分情况下，捕获顶级的 Exception 或 Throwable 都是不合适的，Throwable 更不合适，因为它还包含了 Error 异常。这种捕获非常危险。这意味着本来不必考虑的 Exception（包括类似 ClassCastException 的 RuntimeException）被卷入到应用程序级的错误处理中来。这会让代码运行的错误变得模糊不清。这意味着，假如别人在你调用的代码中加入了新的异常，编译器将无法帮助你识别出各种不同的错误类型。绝大部分情况下，无论如何你都不应该用同一种方式来处理各种不同类型的异常。

本规则也有极少数例外情况：期望捕获所有类型错误的特定的测试代码和顶层代码（为了阻止这些错误在用户界面上显示出来，或者保持批量工作的运行）。这种情况下可以捕获顶级的 Exception（或 Throwable）并进行相应的错误处理。在开始之前，你应该非常仔细地考虑一下，并在注释中解释清楚为什么这么做是安全的。

比捕获顶级 Exception 更好的方案：

- 分开捕获每一种异常，在一条 try 语句后面跟随多个 catch 语句块。这样可能会有点别扭，但总比捕获所有 Exception 要好些。请小心别在 catch 语句块中重复执行大量的代码。
- 重新组织一下代码，使用多个 try 块，使错误处理的粒度更细一些。把 IO 从解析内容的代码中分离出来，根据各自的情况进行单独的错误处理。
- 再次抛出异常。很多时候在你这个级别根本就没必要捕获这个异常，只要让方法抛出该异常即可。

请记住：异常是你的朋友！当编译器指出你没有捕获某个异常时，请不要皱眉头。而应该微笑：编译器帮助你找

到了代码中的运行时（runtime）问题。

9.8 不要使用 Finalizer

Finalizer 提供了一个机会，可以让对象被垃圾回收器回收时执行一些代码。

优点：便于执行清理工作，特别是针对外部资源。

缺点：调用 finalizer 的时机并不确定，甚至根本就不会调用。

结论：我们不要使用 finalizers。大多数情况下，可以用优秀的异常处理代码来执行那些要放入 finalizer 的工作。如果确实是需要使用 finalizer，那就定义一个 close() 方法（或类似的方法），并且在文档中准确地记录下需要调用该方法的时机。相关例程可以参见InputStream。这种情况下还是适合使用 finalizer 的，但不需要在 finalizer 中输出日志信息，因为日志不能因为这个而被撑爆。

9.9 使用完全限定 Import

当需要使用 foo 包中的 Bar 类时，存在两种可能的 import 方式：

1. import foo.*;

优点：可能会减少 import 语句。

2. import foo.Bar;

优点：实际用到的类一清二楚。代码的可读性更好，便于维护。

结论： 用后一种写法来 import 所有的 Android 代码。不过导入 java 标准库 (java.util.*、java.io.*等) 和单元测试代码 (junit.framework.*) 时可以例外。

9.10 对 Import 语句排序

import 语句的次序应该如下：

1. Android imports

2. 第三方库（com、junit、net、org）

3. java 和 javax

为了精确匹配 IDE 的配置，import 顺序应该是：

1. 在每组内部按字母排序，大写字母排在小写字母的前面。

2. 每个大组之间应该空一行（android、com、junit、net、org、java、javax）。

3. 原先次序是不作为规范性要求的。这意味着要么允许 IDE 改变顺序，要么使用 IDE的开发者不得不禁用 import 自动管理功能并且人工维护 import。这看起来比较糟糕。每当说起 java 规范，推荐的规范到处都是。符合我们要求的差不多就是"选择一个次序并坚持下去。"于是，我们就选择一个规范，更新规范手册，并让 IDE

去遵守它。我们期望：不必耗费更多的精力，用 IDE 编码的用户就按照这种规则去 import 所有的 package。

基于以下原因，选定了本项规则：

- 导入人员期望最先看到的放在最开始位置（android）。
- 导入人员期望最后才看到的放在最后（java）。
- 风格让人容易遵守。
- IDE 可以遵守。

静态 import 的使用和位置已经成为略带争议的话题。有些人愿意让静态 import 和其它 import 混在一起，另一些人则期望让它们位于其它 import 之上或者之下。另外，我们还未提到让所有 IDE 都遵守同一个次序的方法。

因为大多数人都认为这部分内容并不要紧，只要遵守你的决定并坚持下去即可。

9.11 限制变量的作用范围

局部变量的作用范围应该是限制为最小的（Effective Java 第 29 条）。使用局部变量，可以增加代码的可读性和可维护性，并且降低发生错误的可能性。每个变量都应该在最小范围的代码块中进行声明，该代码块的大小只要能够包含所有对该变量的使用即可。

应该在第一次用到局部变量的地方对其进行声明。几乎所有局部变量声明都应该进行初始化。如果还缺少足够的信息来正确地初始化变量，那就应该推迟声明，直至可以初始化为止。

本规则存在一个例外，就是涉及 try-catch 语句的情况。如果变量是用方法的返回值来初始化的，而该方法可能会抛出一个 checked 异常，那么必须在 try 块中进行变量声明。如果需在 try 块之外使用该变量，那它就必须在 try 块之前就进行声明了，这时它是不可能进行正确的初始化的。

```
//      实例化类cl，表示有序集合
Set s = null;
try {
    s = (Set) cl.newInstance();
} catch(IllegalAccessException e) {
    throw new IllegalArgumentException(cl + " not accessible");
} catch(InstantiationException e) {
    throw new IllegalArgumentException(cl + " not instantiable");
}

//      集合练习
s.addAll(Arrays.asList(args));
```

但即便是这种情况也是可以避免的，把try-catch 块封装在一个方法内即可。

```
Set createSet(Class cl) {
//      实例化类cl，表示有序集合
try {
    return (Set) cl.newInstance();
} catch(IllegalAccessException e) {
    throw new IllegalArgumentException(cl + " not accessible");
} catch(InstantiationException e) {
    throw new IllegalArgumentException(cl + " not instantiable");
    }
}
...
```

```
//      集合练习
Set s = createSet(cl);
s.addAll(Arrays.asList(args));
```

除非理由十分充分，否则循环变量都应该在for语句内进行声明。

```
for (int i = 0; i n; i++) {
    doSomething(i);
}
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomethingElse(i.next());
}
```

9.12 使用标准的 Java Annotation

Annotation 应该位于 Java 语言元素的其它修饰符之前。 简单的 marker annotation（@Override 等）可以和语言元素放在同一行。 如果存在多个 annotation，或者annotation 是参数化的，则应按字母顺序各占一行来列出。

对于 Java 内建的三种 annotation，Android 标准的实现如下：

- @Deprecated：只要某个语言元素已不再建议使用了，就必须使用@Deprecated annotation。如果使用了@Deprecated annotation，则必须同时进行@deprecated Javadoc 标记，并且给出一个替代的实现方式。此外请记住，被@Deprecated 的方法仍然是能正常执行的。如果看到以前的代码带有@deprecated Javadoc 标记，也请加上@Deprecated annotation。
- @Override: 只要某个方法覆盖了已过时的或继承自超类的方法，就必须使用@Override annotation。例如，如果方法使用了@inheritdocs Javadoc 标记，且继承自超类（而不是interface），则必须同时用@Override 标明覆盖了父类方法。
- @SuppressWarnings：@SuppressWarnings annotation 仅用于无法消除编译警告的场合。如果警告确实经过测试"不可能消除"，则必须使用@SuppressWarnings annotation，以确保所有的警告都能真实反映代码中的问题。

当需要使用@SuppressWarnings annotation时，必须在前面加上TODO注释行，用于解释"不可能消除"警告的条件。通常是标明某个令人讨厌的类用到了某个拙劣的接口。

```
// TODO:        第三方类 com.third.useful.Utility.rotate()       必须采用泛型
@SuppressWarnings("generic-cast")
List<String> blix = Utility.rotate(blax);
```

如果需要使用@SuppressWarnings annotation，应该重新组织一下代码，把需要应用 annotation 的语言元素独立出来。

9.13 简称等同于单词

简称和缩写都视为变量名、方法名和类名。以下名称可读性更强：

好	差
XmlHttpRequest	XMLHTTPRequest

好	差
getCustomerId	getCustomerID
class Html	class HTML
String url	String URL
long id	long ID

如何对待简称，JDK 和 Android 底层代码存在很大的差异。因此，你几乎不大可能与其它代码取得一致。别无选择，把简称当作完整的单词看待吧。

关于本条规则的进一步解释，请参阅 Effective Java 第 38 条和 Java Puzzlers 第 68条。

9.14 使用 TODO 注释

对那些临时性的、短期的、够棒但不完美的代码，请使用 **TODO** 注释。

TODO 注释应该包含全部大写的 TODO，后跟一个冒号：

```
// TODO: Remove this code after the UrlTable2 has been checked in.
// TODO: Change this to use a flag instead of a constant.
```

如果 TODO 注释是"将来要做某事"的格式，则请确保包含一个很明确的日期（"在2013 年 11 月会修正"），或是一个很明确的事件（"在所有代码整合人员理解了 V7 协议之后删除本段代码"）。

9.15 慎用 Log

记录日志会对性能产生显著的负面影响。如果日志内容不够简炼的话，很快会丧失可用性。日志功能支持五种不同的级别。以下列出了各个级别及其使用场合和方式。

- ERROR: 该级别日志应该在致命错误发生时使用，也就是说，错误的后果能被用户看到，但是不明确删除部分数据、卸装程序、清除数据区或重新刷机（或更糟糕）就无法恢复。该级别总是记录日志。需要记录 ERROR 级别日志的事件一般都应该向统计信息收集（statistics-gathering ）服务器报告。
- WARNING：该级别日志应该用于那些重大的、意外的事件，也就是说，错误的后果能被用户看到，但是不采取明确的动作可能就无法无损恢复，从等待或重启应用开始，直至重新下载新版程序或重启设备。该级别总是记录日志。需记录WARNING 级别日志的事件也可以考虑向统计信息收集服务器报告。
- INFORMATIVE：该级别的日志应该用于记录大部分人都会感兴趣的事件，也就是说，如果检测到事件的影响面可能很广，但不一定是错误。应该只有那些拥有本区域内最高级别身份认证的模块才能记录这些日志（为了避免级别不足的模块重复记录日志）。该级别总是记录日志。
- DEBUG：该级别的日志应该用于进一步记录有关调查、调试意外现象的设备事件。应该只记录那些有关控件运行所必需的信息。如果 debug 日志占用了太多的日志空间，那就应该使用详细级别日志（verbose）才更为合适。即使是发行版本（release build），该级别也会被记录，并且需用 if (LOCAL_LOG) 或 if (LOCAL_LOGD) 语句块包裹，这里的LOCAL_LOG[D] 在你的类或子控件中定义。这样就能够一次性关闭所有的调试日志。因此在 if (LOCAL_LOG) 语句块中不允许存在逻辑判断语句。所有日志所需的文字组织工作也应在 if (LOCAL_LOG) 语句块内完成。如果对记录日志的调用会导致在 if (LOCAL_LOG) 语句块之外完成文字组织工作，那该调用就必须控制在一个方法内完成。还存在一些代码仍然在使用 if (localLOGV)。这也是可以接受的，虽然名称不是标准的。

- VERBOSE：该级别日志应用于所有其余的事件。该级别仅会在调试版本（debug build）下记录日志，并且需用 if (LOCAL_LOGV) 语句块（或等效语句）包裹，这样该部分代码默认就不会编译进发行版本中去了。所有构建日志文字的代码将会在发行版本中剥离出去，并且需包含在 if (LOCAL_LOGV) 语句块中。

注意：

- 除了 VERBOSE 级别外，在同一个模块中同一个错误应该尽可能只报告一次：在同一个模块内的一系列层层嵌套的函数调用中，只有最内层的函数才返回错误；并且只有能为解决问题提供明显帮助的时候，同一模块中的调用方才写入一些日志。
- 除了 VERBOSE 级别外，在一系列嵌套的模块中，当较低级别的模块对来自较高级别模块的非法数据进行检测时，应该只把检测情况记录在 DEBUG 日志中，并且只记录那些调用者无法获取的信息。特别是不需要记录已经抛出异常的情况（异常中应该包含了全部有价值的信息），也不必记录那些只包含错误代码的信息。当应用程序与系统框架间进行交互时，这一点尤为重要。系统框架已能正确处理的第三方应用程序，也不应该记录大于 DEBUG 级别的日志。仅当一个模块或应用程序检测到自身或来自更低级别模块的错误时，才应该记录 INFORMATIVE 及以上级别的日志。
- 如果一个通常要记录日志的事件可能会多次发生，则采取一些频次限制措施或许是个好主意，以防日志被很多重复（或类似）的信息给撑爆了。
- 网络连接的丢失可被视为常见现象，也是完全可以预见的，不应该无缘无故就记录进日志。影响范围限于应用程序内部的网络中断应该记录在 DEBUG 或 VERBOSE级别的日志中（根据影响的严重程度及意外程度，再来确定是否在发行版本中也记录日志）。
- 有权访问的文件系统或第三方应用程序发起的系统空间满，应该记录大于INFORMATIVE 级别的日志。
- 来自任何未授信源的非法数据（包括共享存储上的任何文件，或来自任何网络连接的数据）可被视为可预见的，如果检测到非法数据也不应该记录大于 DEBUG 级别的日志（即使记录也应尽可能少）。
- 请记住，对字符串使用+操作符时，会在后台以默认大小（16 个字符）缓冲区创建一个 StringBuilder 对象，并且可能还会创建一些其它的临时 String 对象。换句话说，显式创建 StringBuilders 对象的代价并不会比用'+'操作符更高（事实上效率还将会提高很多）。还要记住，即使不会再去读取这些日志，调用 Log.v() 的代码也将编译进发行版中并获得执行，包括创建字符串的代码。
- 所有要被人阅读并存在于发行版本中的日志，都应该简洁明了、没有秘密、容易理解。这里包括所有 DEBUG 以上级别的日志。
- 只要有可能，日志就应该一句一行。行长最好不超过 80 或 100 个字符，尽可能避免超过 130 或 160 个字符（包括标识符）的行。
- 报告成功的日志记录绝不应该出现在大于 VERBOSE 级别的日志中。
- 用于诊断难以重现事件的临时日志应该限于 DEBUG 或 VERBOSE 级别，并且应该用 if 语句块包裹，以便在编译时能够一次全部关闭。
- 小心日志会泄漏隐私。应该避免将私人信息记入日志，受保护的内容肯定也不允许记录。这在编写系统框架级代码时尤为重要，因为很难预知哪些是私人信息和受保护信息。
- 绝对不要使用 System.out.println()（或本地代码中的 printf()）。System.out 和System.err 会重定向到/dev/null，因此 print 语句不会产生任何可见的效果。可是，这些调用中的所有字符串创建工作都仍然会执行。

日志的黄金法则是：你的日志记录不会导致其它日志的缓冲区溢出，正如其他人的日志也不会让你的溢出一样。

10 在Eclipse中使用模板

10.1 导入android编码规范模板

在eclipse的preferences中，选择java → code style → formatter中选择Import，选择工程根目录下的development/ide/eclipse/目录下的android-formatting.xml。

在eclipse的preferences中，选择java → code style → Organize Imports中选择Import，选择工程根目录下的development/ide/eclipse/目录下的android.importorder。

说明：导入这两个文件，是为了与源码中的Android程序保持一致的编码规范。android-formatting.xml用来配置eclipse编译器的代码风格；android.importorder用来配置eclipse的import的顺序和结构。

E:/xampp/backup/dokuwiki/data/pages/工作规范/android开发规范.txt · 最后更改: 2014/03/13 11:44 由 yuhua.wang