**Java编程规范**

# 一：[Java语言规范](#Java语言规范)

# [不要忽略Exceptions](#不要忽略异常)

# [不要捕获顶级Exception](#不要捕获顶级的Exception)

# [不要使用Finalizer](#_不要使用Finalizer_1)

# [使用完全限定名称的Import](#使用完全限定Import)

# 二：[Java编码规范](#Java编程规范)

## 1. [使用Javadoc标准的注释](#使用Javadoc标准注释)

## 2. [编写简短的方法](#编写简短的方法)

## 3. [在标准的位置定义字段](#在标准的位置定义字段)

## 4. [限制变量的作用范围](#限制变量的作用范围)

## 5. [对Import语句排序](#对Import语句排序)

## 6. [使用空格进行缩进](#使用空格进行缩进)

## 7. [遵守字段命名惯例](#遵守字段命名惯例)

## 8. [使用标准的大括号风格](#使用标准的大括号风格)

## 9. [限制代码行长度](#限制代码行的长度)

## 10. [使用标准的Java Annotation](#使用标准的Java)

## 11. [简称等同于单词](#简称等同于单词)

## 12. [使用TODO注释](#使用TODO注释)

## 13. [慎用日志](#慎用Log)

## 14. [保持一致](#保持一致)

# 三：[Javatest风格规范](#Javatest风格规范)

## 1. [遵守测试方法命名规范](#遵守测试方法命名规范)

**Java语言规范**

我们遵循标准的Java编码规范，并加入了新的规则：

不要忽略异常

有时，完全忽略异常是非常诱人的，比如：

void setServerPort(String value) {

    try {

        serverPort = Integer.parseInt(value);

    } catch (NumberFormatException e) { }

}

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。

无论何时，空的*catch*语句都会让人感到不寒而栗。虽然很多情况下确实是一切正常，但至少你不得不去忧虑它。在*Java*中你无法逃离这种恐惧感。 -[James Gosling](http://www.artima.com/intv/solid4.html)

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。

        void setServerPort(String value) {

            try {

                serverPort = Integer.parseInt(value);

            } catch (NumberFormatException e) {

                serverPort = 80;  // default port for server

            }

        }

 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。

        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：

        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.

            }

        }

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();        // 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!

}

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。

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。

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。

1.   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。

**Java类库规范**

使用Android Java类库和工具存在一些惯例。有时这些惯例会作出重大变化，可之前的代码也许会用到过时的模板或类库。如果用到这部分过时的代码，沿用已有的风格就是了（参阅Consistency)。创建新的组件时就不要再使用过时的类库了。

**Java编程规范**

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。

package com.android.internal.foo;

import android.os.Blah;

import android.view.Yada;

import java.sql.ResultSet;

import java.sql.SQLException;

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。

例如：

static double sqrt(double a) {

    ...

}

或

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。

Android目前还没有规定自己的Javadoc注释撰写规范，但是应该遵守[Sun Javadoc约定](http://java.sun.com/j2se/javadoc/writingdoccomments/)。

编写简短的方法

为了把规模控制在合理范围内，方法应该保持简短和重点突出。不过，有时较长的方法也是合适的，所以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。

在标准的位置定义字段

字段应该定义在文件开头，或者紧挨着使用这些字段的方法之前。

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。

// Instantiate class cl, which represents some sort of Set

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");

}

// Exercise the set

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) {

    // Instantiate class cl, which represents some sort of Set

    try {

        return (Set) cl.newInstance();

    } catch(IllegalAccessException e) {

        throw new IllegalArgumentException(cl + " not accessible");

    } catch(InstantiationException e) {

        throw new IllegalArgumentException(cl + " not instantiable");

    }

}

...

// Exercise the set

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);

}

和

for (Iterator i = c.iterator(); i.hasNext(); ) {

    doSomethingElse(i.next());

}

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：

 在每组内部按字母排序，大写字母排在小写字母的前面。

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

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。

使用空格进行缩进

我们的代码块缩进使用4个空格。我们从不使用制表符tab。如果存在疑惑，与前后的其它代码保持一致即可。

我们用8个空格作为换行后的缩进，包括函数调用和赋值。例如这是正确的：

Instrument i =

        someLongexpression\_r(that, wouldNotFit, on, one, line);

而这是错误的：

Instrument i =

    someLongexpression\_r(that, wouldNotFit, on, one, line);

遵守字段命名惯例

 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 static final字段（常量）全部字母大写并用下划线分隔。

例如：

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;

}

使用标准的大括号风格

大括号不单独占用一行；它们紧接着上一行书写。就像这样：

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();

}

这也是合法的：

if (condition) body();

但这是非法的：

if (condition)

    body();  // bad!

限制代码行的长度

每行代码的长度应该不超过100个字符。

有关本规则的讨论有很多，最后的结论还是最多不超过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。

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。比如：

// TODO: The third-party class com.third.useful.Utility.rotate() needs generics

@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。

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 |
| 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。

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.

和

// 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2005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”）。

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。

保持一致

我们的最终想法是：保持一致。如果你正在编写代码，请花几分钟浏览一下前后的其它代码，以确定它们的风格。如果它们在if语句前后使用了空格，那你也应该遵循。如果它们的注释是用星号组成的框框围起来的，那也请你照办。

保持风格规范的重点是有一个公共的代码词汇表，这样大家就可以把注意力集中于你要说什么，而不是你如何说。我们在这里列出了全部的风格规范，于是大家也知道了这些词汇。不过本地化的风格也很重要。如果你要加入的代码和已存在的代码风格迥异，那就会突然打破阅读的节奏。请努力避免这种情况的发生。

**Javatest风格规范**

遵守测试方法命名规范

命名测试方法时，可以用下划线来分隔测试的条件。这种风格可以让测试的条件一目了然。

比如:

testMethod\_specificCase1 testMethod\_specificCase2

void testIsDistinguishable\_protanopia() {

    ColorMatcher colorMatcher = new ColorMatcher(PROTANOPIA)

    assertFalse(colorMatcher.isDistinguishable(Color.RED, Color.BLACK))

    assertTrue(colorMatcher.isDistinguishable(Color.X, Color.Y))

}