Java编码规范

|  |  |
| --- | --- |
| 公司 | 美博雅（MEIBOYA） |
| 项目 | 美肤志（meifuzhi.com） |
| 编者 | 曲显平 |
| 审核 | 赵楠 |
| 发布日期 | 2012-10-16 |
| 版本 | V1r1 |

目录

[Java编码规范 1](#_Toc338174500)

[一、 基础篇 2](#_Toc338174501)

[1. 包 2](#_Toc338174502)

[2. 类和接口 2](#_Toc338174503)

[3. 成员变量 3](#_Toc338174504)

[4. 常量 3](#_Toc338174505)

[5. 局部变量 3](#_Toc338174506)

[6. 成员函数和静态变量 4](#_Toc338174507)

[7. 布局文件内部命名 4](#_Toc338174508)

[二、 进阶篇 5](#_Toc338174509)

[8. 不能忽视异常处理 5](#_Toc338174510)

[9. 不要捕获通用异常类（Exception） 8](#_Toc338174511)

[10. 不要使用Finalizers（终结）（这个不太懂） 9](#_Toc338174512)

[11. 完全限定imports 9](#_Toc338174513)

[12. 使用Javadoc标准写注释 10](#_Toc338174514)

[13. 写短方法 12](#_Toc338174515)

[14. 局部变量应该推迟至使用前声明并初始化 12](#_Toc338174516)

[15. 限制变量的作用域 13](#_Toc338174517)

[16. import的顺序 15](#_Toc338174518)

[17. 使用空格进行缩进 15](#_Toc338174519)

[18. 变量命名 16](#_Toc338174520)

[19. 使用标准的括号风格 17](#_Toc338174521)

[20. 行长度限制 18](#_Toc338174522)

[21. 使用标准Java Annotations（注解@） 18](#_Toc338174523)

[22. 对待字母缩写大小写 19](#_Toc338174524)

[23. 使用TODO注释 20](#_Toc338174525)

[24. 日志 20](#_Toc338174526)

[25. 保持一致性 23](#_Toc338174527)

[三、 Javatests Style Rules 23](#_Toc338174528)

[26. 按照测试方法命名 23](#_Toc338174529)

# 基础篇

## 包

1. 包名必须全部小写，单词数<=2，最好为1个单数名词

## 类和接口

1. 需每个单词首字母大写。
2. 接口，常以able或ible等结尾或以I开头
3. 抽象类，使用Abstract作为类名前缀
4. 异常类，使用Exception作为类名后缀
5. 类名，尽量少地使用其他人不熟悉的缩写，不要和 Java 的标准库中的类名冲突，比如：Class, Object, String 等

例外：以下情况可以允许写特殊类名（名称简短）

类特别常用

类非常特殊，难以归类

私有类或内部类

不推荐其他人调用的 共有、保护、默认类

1. 缺省接口实现应该使用 Default 名称前缀 。例 如 ： DefaultEntityMaker

## 成员变量

1. 最好为单数名词
2. 能private就不要default，能default就不要protected，最好不要public
3. 如果是集合或数组，用复数名词，Map pets， 比 Map petMap 要好
4. 不要用一个字母，尤其是 i，你可以用 index 或者 cursor 来代替

## 常量

1. 全部大写，下划线分隔
2. 命名常量（带有 final 修饰符的域）时需分隔。如 ： public final int MAX\_VALUE = 30 。

## 局部变量

1. 局域变量名可以短小精悍，可以用缩写，比如 StringBuilder sb（…）

比如下面两种情况可能后者要好于前者：

 public String abc(String str){  
        AbcObjectSet abcObjectSet = new AbcObjectSet();  
        abcObjectSet.setName(str);  
        return abcObjectSet.getBrief();  
 }

下面这个就很容易阅读了：

 public String abc(String str){  
        AbcObjectSet aos = new AbcObjectSet();  
        aos.setName(str);  
        return aos.getBrief();  
 }

## 成员函数和静态变量

1. 除了 setter / getter，其他的函数采用动词或者动名短语
2. 以下情况可以允许写特殊函数名（名称简短）
3. 函数特别常用
4. 函数非常特殊
5. 私有函数或默认函数
6. 允许写成支持链式赋值的 setter，并且也可以使用同名 getter。（这个我其实不太懂优劣，看客可以自己捉摸）例如：

// Setter  
public Pet name(String name){  
        this.name = name;  
        return this;  
}  
  
// Getter  
public String name(){  
        return this.name;  
}

## 布局文件内部命名

1. 布局文件：布局文件名称的定义必须为小写字母，否者无法生成R类，尽量不要用缩写，以表达清楚该文件用途为本，通常情况下用下划线连接各语义单词。例如dialog\_title\_icons.xml 或者list\_menu\_item\_checkbox.xml.
2. 控件ID的定义，ID的定义一律为小写，例如：一个用户名的 TextView  可以定义为：@+id/username\_view。以“名词\_控件名称”这种形式定义。
3. 其次是图片的定义格式，图片的定义也是以解释清楚用途为准，参照这种定义格式：btn\_background\_ok.png。
4. string类的name定义，这里可以按照JAVA中变量的定义方式定义。首字母小写，驼峰命名法。例如：<string name="userName\_view">用户名：</string>
5. 最后普通类名与变量的定义：定义与用户交互的类：××Activity.java；自定义变量一律以小写m开头，例如：EditText mUserName =（EditText）findViewById（R.id.username\_edit）

# 进阶篇

## 不能忽视异常处理

先看下面一个例子，catch之后是一个空的处理：

void setServerPort(String value) {

try {

serverPort = Integer.parseInt(value);

} catch (NumberFormatException e) { }

}

不应该使用上面这种方式，你可能觉得这里我就不需要做什么事情呀！可是别人却不这样看，他会觉得疑惑疑虑，不知所以。

可以使用如下的方式替代：

1. Throw the exception up to the caller of your method.

void setServerPort(String value) throws NumberFormatException {

serverPort = Integer.parseInt(value);

}

1. Throw a new exception that's appropriate to your level of abstraction.

void setServerPort(String value) throws ConfigurationException {

try {

serverPort = Integer.parseInt(value);

} catch (NumberFormatException e) {

throw new ConfigurationException("Port " + value + " is not valid.");

}

}

1. Handle the error gracefully and substitute an appropriate value in the catch {} block.

/\*\* 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. Catch the Exception and throw a new RuntimeException. This is dangerous: only do it if you are positive that if this error occurs, the appropriate thing to do is crash.

/\*\* Set port. If value is not a valid number, die. \*/

void setServerPort(String value) {

try {

serverPort = Integer.parseInt(value);

} catch (NumberFormatException e) {

throw new RuntimeException("port " + value " is invalid, ", e);

}

}

Note that the original exception is passed to the constructor for RuntimeException. If your code must compile under Java 1.3, you will need to omit the exception that is the cause.

1. Last resort: if you are confident that actually ignoring the exception is appropriate then you may ignore it, but you must also comment why with a good reason.

/\*\* 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.

}

}

## 不要捕获通用异常类（Exception）

当向下面这样捕获异常是，你就是一个大懒人：

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!

}

这种代码非常危险，因为它会捕捉一切异常，任何你没有想到的异常都会被它捕获，你的handleError可能没法处理这么多异常，事情就会变得更糟糕。

当然也有例外，比如你在测试的时候，你真正需要把所有异常都捕获，然后分析它。

一般可以用如下方式替代上面不好的代码段：

1. 单独的try后面跟着每一个异常的catch代码块，分别捕捉每个异常，这可能很麻烦，但仍是最好的捕获异常方式。当然要注意catch多了之后，里面可能会有许多重复代码。
2. 用大量的try代码块，重构你的代码来做到更细粒度的错误处理。
3. 重新抛出异常。很多时候，在这层函数中不需要处理，那么继续将它抛出就好了。

记住：异常是你的朋友，要善待哦！

## 不要使用Finalizers（终结）（这个不太懂）

Finalizers是当一个对象被垃圾回收时的一种代码块执行方式。

优点：可以很方便的进行清理，特别是外部资源。

缺点：不能保证它何时会被调用，甚至是否最终真的会被调用。

结论：我们不使用它。在大多数情况下，你可以用优秀的异常处理来替代它。如果你确实需要，定义一个close（）方法（或其它类似的方法），并需要在注释中清楚记录什么时候需要调用该方法。可以把InputStream作为一个例子，在这种情况下，它是合适的，但并不需要从finalizer输出任何日志消息。

## 完全限定imports

当你想使用package foo中的class Bar时，有两种方式，将其import：

import foo.\*;

第一种方式的优点：潜在的import语句的数量减少。

import foo.Bar;

第二种方式的优点：明确指出了实际使用了哪些类，使代码对维护者更可读。

结论： 使用后者来进行所有代码的import。不过，有明确的例外，它是在使用Java标准库（java.util.\*，java.io.\*等）和单元测试代码（junit.framework.\*）时。

## 使用Javadoc标准写注释

1. 文件顶部应该有版权声明
2. 紧跟着是package和import的相关语句，每个代码块由一个空行分隔
3. 接下来就是类或接口的声明。
4. 在Javadoc注释中，描述类或接口是做什么的。

* **类/接口注释的内容** （第1条必须保证）

1. 用途。开发人员使用某个类/接口之前，需要知道采用该类/接口的用途。

2. 如何使用。开发人员需要知道该类/接口应该如何使用，如果必要的话还需要注明不应该如何使用。

3. 开发维护的日志。一个有关于该类/接口的维护记录：时间、作者、摘要。

* **方法注释的内容** （第1、5、6、7项必须保证）

1. 类该方法是做什么的 。

2. 该方法如何工作。

3. 代码修改历史纪录。

4. 方法调用代码示范。

5. 必须传入什么样的参数给这个方法。@param

6. 异常处理。@throws

7. 这个方法返回什么。@return

/\*

\* Copyright (C) 2010 The Android Open Source Project

\*

\* Licensed under the Apache License, Version 2.0 (the "License");

\* you may not use this file except in compliance with the License.

\* You may obtain a copy of the License at

\*

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

\*/

package com.android.internal.foo;

import android.os.Blah;

import android.view.Yada;

import java.sql.ResultSet;

import java.sql.SQLException;

/\*\*

\* Does X and Y and provides an abstraction for Z.

\*/

public class Foo {

...

}

每个类以及重要的公共方方法必须包含Javadoc注释。这句话应该开始用第三人称描述动词。

/\*\* Returns the correctly rounded positive square root of a double value. \*/

static double sqrt(double a) {

...

}

或

/\*\*

\* Constructs a new String by converting the specified array of

\* bytes using the platform's default character encoding.

\*/

public String(byte[] bytes) {

...

}

推荐使用后者

## 写短方法

1. 如果超过40行，想想是否可以拆分。

## 局部变量应该推迟至使用前声明并初始化

1. 变量应该被声明（定义）在文件的顶部或者在使用它们的方法之前。

## 限制变量的作用域

1. 局部变量的范围应保持在最小区间（《Effective Java》的第29项）。通过这样做，你可以增加你的代码的可读性和可维护性，并降低出错的可能性。每个变量都声明在靠近使用的最内部的块。
2. 局部变量应在它被第一次使用的时候声明。每一个局部变量声明应包含一个初始值。如果你还没有足够的信息来初始化一个变量，你应该推迟声明直到你准备好了。
3. 在使用try-catch语句时会有例外，如果一个变量初始化的值是一个方法的返回值，而这个方法抛出一个可检测的异常，那它必须初始化一个try块内。如果该变量必须在try块外部使用，那么它必须在try块之前声明，显然这是它尚未被初始化之前：

// 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));

但即使是这种情况，也可避免封装try-catch块在一个方法中：

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));

循环变量应该声明在for语句本身，除非有令人信服的理由不这样做：

for (int i = 0; i n; i++) {

doSomething(i);

}

和

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

doSomethingElse(i.next());

}

## import的顺序

1. Android imports
2. Imports from third parties (com, junit, net, org)
3. java and javax

为了与IDE的设置相匹配，还应该有：

1. 每个分组内按字母顺序排列，大写字母在小写字母之前（例如：Z在a之前）。
2. 各主要分组（android，com，junit，net，org，java，javax）之间应该有一个空行分隔。

如上做的好处：

1. The imports people want to look at first tend to be at the top (android)
2. The imports people want to look at least tend to be at the bottom (java)
3. Humans can easily follow the style
4. IDEs can follow the style

## 使用空格进行缩进

1. 使用4个空格做块缩进，不使用tab。保证代码看起来在任何编辑器上是一致的。
2. 我们使用8个空格来缩进换行，包括​​函数调用和分配。例如，下面是正确的：

Instrument i =

someLongExpression(that, wouldNotFit, on, one, line);

下面是不正确的：

Instrument i =

someLongExpression(that, wouldNotFit, on, one, line);

## 变量命名

1. 非公有的，非静态字段的变量，使用m开始。
2. 静态变量使用s开始。
3. 其他字段使用小写字母开始。
4. Public static final字段（或常量）ALL\_CAPS\_WITH\_UNDERSCORES。

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;

}

## 使用标准的括号风格

class MyClass {

int func() {

if (something) {

// ...

} else if (somethingElse) {

// ...

} else {

// ...

}

}

}

这是合法的：

if (condition) {

body();

}

下面也合法：

if (condition) body();

下面不合法：

if (condition)

body(); // bad!

## 行长度限制

代码行每行不超过100个字符。

例外：如果注释中含有url之类的东西，而url长度超过了100字符，那没问题

例外：import可以不受这个限制，因为大家很少看它，同时避免折行也简化了自动化工具的开发成本。

## 使用标准Java Annotations（注解@）

注解优先于其他相同的语言元素修饰符。有一些关键词可以在同一行上简单的标记注释（例如@override）。如果有多个@，或参数化@，按行列出、按字母顺序排列。

在Andr​​oid标准的做法是：

* @Deprecated：
  + 先来解释一下什么是@Deprecated：在写程序中，我们可能希望某个方法之后不能够再被调用，那么我们就可以把这个方法定义为过时的方法，过时的方法在调用时，会在方法名上打上删除线，告诉程序员该方法为过时方法，最后不要再用了。（通常是方法很危险或存在更好的选择）
  + 请记住，@Deprecated的方法*还是应该能够正常工作。*
  + 如果你看到有一个@Deprecated的Javadoc标记的旧代码，请添加@Deprecated的注释。
* @Override：
  + 必须被用在重载或继承自父类时。
  + 这个没什么好说的，很常用，谁用谁知道。
* @ SuppressWarnings：
  + @SuppressWarnings("deprecation") 指示应该在注释元素（以及包含在该注释元素中的所有程序元素）中取消显示指定的编译器警告。
  + 当一个@ SuppressWarnings注释是必要的，它必须在前面加上一个TODO注释，解释加该注释的原因，以及可能的解决方法。要注意这通常会产生一个不合法的类，它有一个比较尴尬的接口。例如：

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

@SuppressWarnings("generic-cast")

List<String> blix = Utility.rotate(blax);

当一个@ SuppressWarnings注释是必需的，代码应该是重构隔离的软件元素的注解适用的情况。

## 对待字母缩写大小写

| **Good** | **Bad** |
| --- | --- |
| XmlHttpRequest | XMLHTTPRequest |
| getCustomerId | getCustomerID |
| class Html | class HTML |
| String url | String URL |
| long id | long ID |

## 使用TODO注释

使用TODO注释的代码，代表暂时的、短期的解决方案，或者是还不错但不完美的。

// 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后面的意思是标记要在将来的某个特定时间完成一件事，那么应该注明一个很具体的日期（"Fix by November 2005"）或非常具体的事件（"Remove this code after all production mixers understand protocol V7."）

## 日志

日志是必要的，但它会影响一部分性能，如果它不能保持合理和简洁，就很快会失去它的用处。下面介绍不同级别的日志应该如何使用：

* ERROR：发生了致命错误（fatal）时，应使用这个级别的记录。
* WARNING：在某些严重的或未期望的事情，且未达到FATAL级别时，使用该级别。
* INFORMATIVE：当关心的某些事件发生，就可以使用该级别日志。例如：当检测到的情况可能产生广泛影响，尽管不一定是错误，这样的情况下应该只记录最相关的模块，以避免重复记录。
* DEBUG：这个级别的记录应被用来记录在设备上发生了什么，排查具体信息或意外行为。
  + 即使是发行版本（release build），该级别也会被记录。需使用 if (LOCAL\_LOG) or if (LOCAL\_LOGD)这种代码块来控制DEBUG-LOG的开启或关闭。
* VERBOSE：这个级别日志应用于记录上述情况以外的其他所有事件。
  + 该级别仅在调试版本（debug build）下记录。使用 if (LOCAL\_LOG) or if (LOCAL\_LOGD)这种代码块来控制DEBUG-LOG的开启或关闭，以防止编译到发行版中。
* 除了VERBOSE级别的日志外，在同一个模块中的相同错误应该只报告一次；在同一个模块内的一系列层次嵌套的函数中，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。
* 日志主要记录表面上调用者无法获取的信息，不需要记录已经抛出异常的情况（异常中应该包含了全部由价值的信息），也不必记录那些只包含错误代码的信息。
* 网络连接的丢失是完全可以预见的，不应该应该被记录。网络连接丢失在一个应用程序的后果，应记录在DEBUG或VERBOSE级别，编译之前判断是否造成严重后果的，再来决定是否在发行版中也记录日志。
* 来自任何未授权信远的非法数据，如果检测到非法数据不应该记录于大雨DEBUG级别的日志，即使记录也应该尽可能少。
* 请记住，在使用+运算符对字符串操作时，会隐式创建一个默认的缓冲区大小（16个字符）的StringBuilder，可能还会创建一些其他的临时String对象。换句话说，显式地创建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）。另外，即使没有再去读取这些日志，调用Log.v（）的代码也将编译进和发布版本并获得执行，包括建立字符串的代码。
* 所有要被其他人阅读并存在与发布版本的日志，应该是简洁、不隐晦、容易理解的，这包括所有的DEBUG以上级别的日志。
* 日志应该一句一行，长度最好在80至100个字符，不要超过130到160个字符（包括标示符）。
* 报告成功的日志，不应该出现在高于VERBOSE级别的日志中。
* 用于诊断很难重现的问题，用作临时记录应使用在DEBUG或VERBOSE水平的日志，应该用if封闭，这样在编译时就完全可以禁用它。
* 日志尝尝会泄露隐私造成安全漏洞。应避免私人信息的打印。受保护的内容不能出现在日志中。
* 不要使用System.out.println()（或本地代码中的printf（））。System.out和System.err重定向到/dev/null，所以打印出的信息没有任何可见效果。然而，这些调用中的所有字符串创建工作仍会执行。
* *日志的黄金法则：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。*

## 保持一致性

重要的就是新代码保持与模块原有代码风格的一致，千万不要在一个模块里有两种风格迥异的代码。只要有统一的风格，人们就能够专注于你在说什么，而不是你怎么说。

# Javatests Style Rules

## 按照测试方法命名

命名可以使用一个下划线分隔被测试的对象和具体的被测试case。这种风格可以更容易地看到究竟是什么case正在被测试。

例如：

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))

}