

Android 编码规范

版本	修订内容	作者	时间
0.1	初稿	魏铮铮	2011-08-15
0.2	添加 Eclipse 配置方法	魏铮铮	2011-08-16

介绍

1. 为什么需要编码规范?

编码规范对于程序员而言尤为重要，有以下几个原因：

- 一个软件的生命周期中，80%的花费在于维护
- 几乎没有任何一个软件，在其整个生命周期中，均由最初的开发人员来维护
- 编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码
- 如果你将源码作为产品发布，就需要确保它是否被很好的打包并且清晰无误，一如你已构建的其它任何产品

命名

2. 包命名

命名规则：一个唯一包名的前缀总是全部小写的 ASCII 字母并且是一个顶级域名，通常是 com, edu, gov, mil, net, org。包名的后续部分根据不同机构各自内部的命名规范而不尽相同。这类命名规范可能以特定目录名的组成来区分部门 (department)，项目(project)，机器(machine)，或注册名(login names)。

例如：*com.hymobile.nloc.activities*

规约：包命名必须以 *com.hymobile* 开始，后面跟有项目名称（或者缩写），再后面为模块名或层级名称。

如：*com.hymobile.项目缩写.模块名* → *com.hymobile.nloc.bookmark*

如：*com.hymobile.项目缩写.层级名* → *com.hymobile.nloc.activities*

3. 类和接口命名

命名规则：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，避免缩写词(除非该缩写词被更广泛使用，像 URL, HTML)

接口一般要使用 able、ible、er 等后缀

例如：*class Raster; class ImageSprite;*

规约：类名必须使用驼峰规则，即首字母必须大写，如果为词组，则每个单词的首字母也必须大写，类名必须使用名词，或名词词组。要求类名简单，不允许出现无意义的单词（如 *class XXXActivity*）。

如：*class BookMarkAdd* → 正确

如：*class AddBookReadPlanActivity* → 错误！ 应为 *class BookReadPlanAdd*

4. 方法的命名

命名规则：方法名是一个动词，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。

例如: `public void run(); public String getBookName();`

类中常用方法的命名:

1. 类的获取方法（一般具有返回值）一般要求在被访问的字段名前加上 `get`，如 `getFirstName()`，`getLastName()`。一般来说，`get` 前缀方法返回的是单个值，`find` 前缀的方法返回的是列表值。
2. 类的设置方法（一般返回类型为 `void`）：被访问字段名的前面加上前缀 `set`，如 `setFirstName()`，`setLastName()`。
3. 类的布尔型的判断方法一般要求方法名使用单词 `is` 或 `has` 做前缀，如 `isPersistent()`，`isString()`。或者使用具有逻辑意义的单词，例如 `equal` 或 `equals`。
4. 类的普通方法一般采用完整的英文描述说明成员方法功能，第一个单词尽可能采用动词，首字母小写，如 `openFile()`，`addCount()`。
5. 构造方法应该用递增的方式写。（参数多的写在后面）。
6. `toString()`方法：一般情况下，每个类都应该定义 `toString()`，其格式为：

5. 变量命名

命名规则：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。变量名应简短且富于描述。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。

例如: `String bookName;`

规约：变量命名也必须使用驼峰规则，但是首字母必须小写，变量名尽可能的使用名词或名词词组。同样要求简单易懂，不允许出现无意义的单词。

如: `String bookName;` → 正确

如: `String bookNameString;` → 错误!

6. 成员变量命名

同变量命名，但不要在私有变量前添加 `m` 字样！

7. 常量命名

命名规则：类常量的声明，应该全部大写，单词间用下划线隔开。

例如: `static final int MIN_WIDTH = 4;`

例如: `static final int MAX_WIDTH = 999;`

例如: `static final int GET_THE_CPU = 1;`

8. 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。

9. layout 命名

规约: layout xml 的命名必须以全部单词小写, 单词间以下划线分割, 并且使用名词或名词词组, 即使用 模块名_功能名称 来命名。

如: `knowledge_gained_main.xml` → 正确

如: `list_book.xml` → 错误!

10. id 命名

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

如: 某 `TextView` `@+id/textbookname` → 错误! 应为 `@+id/book_name_show`

如: 某 `EditText` `@+id/textbookname` → 错误! 应为 `@+id/book_name_edit`

11. 资源命名

规约: layout 中所使用的所有资源(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` → 正确

如: 某分割线: `line.png` 或 `separator.png` → 正确

注释

Java 程序有两类注释: 实现注释 (implementation comments) 和文档注释 (document comments)。实现注释是使用 `/*...*/` 和 `//` 界定的注释。文档注释 (被称为 “doc comments”) 由 `/**...*/` 界定。文档注释可以通过 `javadoc` 工具转换成 HTML 文件。

1. 文件注释

所有的源文件都应该在开头有一个注释, 其中列出类名、版本信息、日期和版权声明。如下:

```
/*
 * 文件名
 * 包含类名列表
 * 版本信息, 版本号
 * 创建日期。
 * 版权声明
 */
```

2. 类注释

每一个类都要包含如下格式的注释, 以说明当前类的功能等。

```
/**
 * 类名
 * @author 作者 <br/>
```

```
* 实现的主要功能。
* 创建日期
* 修改者，修改日期，修改内容。
*/
```

3. 方法注释

每一个方法都要包含 如下格式的注释 包括当前方法的用途，当前方法参数的含义，当前方法返回值的内容和抛出异常的列表。

```
/**
 *
 * 方法的一句话概述
 * <p>方法详述（简单方法可不必详述）</p>
 * @param s 说明参数含义
 * @return 说明返回值含义
 * @throws IOException 说明发生此异常的条件
 * @throws NullPointerException 说明发生此异常的条件
 */
```

4. 类成员变量和常量注释

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

```
/**
 * XXXX含义
 */
```

5. 其他注释

方法内部的注释 如果需要多行 使用/*…… */形式，如果为单行是用//……形式的注释。不要再方法内部使用 java doc 形式的注释 “/**……*/”，简单的区分方法是，java doc 形式的注释在 eclipse 中为蓝色，普通注释为绿色。

6. XML 注释

规约: 如果当前 layout 或资源需要被多处调用, 或为公共使用的 layout(若 list_item), 则需要在 xml 写明注释。要求注释清晰易懂。

代码风格

1. 缩进

规约: 不允许使用 Tab 进行缩进，使用空格进行缩进，推荐缩进为 2 空格。

2. 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
- 一个方法内的两个逻辑段之间, 用以提高可读性

规约: 通常在 变量声明区域之后要用空行分隔, 常量声明区域之后要有空行 分隔, 方法声明之前要有空行分隔。

3. 行宽

无特别规定, 因为现在的显示器都比较大, 所以推荐使用 120 进行设置。

规约

1. 方法

- 一个方法尽量不要超过 15 行, 如果方法太长, 说明当前方法业务逻辑已经非常复杂, 那么就需要进行方法拆分, 保证每个方法只作一件事。
- 不要使用 try catch 处理业务逻辑!!!!

2. 参数和返回值

- 一个方法的参数尽可能的不要超过 4 个!
- 如果一个方法返回的是一个错误码, 请使用异常!!
- 尽可能不要使用 null, 替代为异常 或者使用空变量 如返回 List 则可以使用 Collections.emptyList()

3. 神秘的数

代码中不允许出现单独的数字, 字符! 如果需要使用数字或字符, 则将它们按照含义封装为静态常量! (for 语句中除外)

4. 控制语句

判断中如有常量, 则应将常量置于判断式的右侧。如:

```
if ( true == isAdmin())...
```

尽量不使用三目条件的嵌套。

所有 if 语句必须用{}包括起来,即便是只有一句:

```
if (true){  
//do something.....  
}
```

```
if (true)  
    i = 0; //不要使用这种
```

对于循环:

```
//不推荐方式  


---

while (index < products.getCount()) {  
//每此都会执行一次 getCount () 方法,
```

```
//若此方法耗时则会影响到执行效率
//而且可能带来同步问题，若有同步需求，请使用同步块或同步方法
}
//推荐方式_____
//将操作结构保存在临时变量里，减少方法调用次数
final int count = products.getCount();
while(index < count){
}
```

5. 异常的捕捉处理

- 通常的思想是只对错误采用异常处理：逻辑和编程错误，设置错误，被破坏的数据，资源耗尽，等等。
- 通常的法则是系统在正常状态下以及无重载和硬件失效状态下，不应产生任何异常。
- 最小化从一个给定的抽象类中导出的异常的个数。对于经常发生的可预计事件不要采用异常。不要使用异常实现控制结构。
- 若有 finally 子句，则不要在 try 块中直接返回，亦不要在 finally 中直接返回。

6. 访问控制

若没有足够理由，不要把实例或类变量声明为公有。通常，实例变量无需显式的设置(set)和获取(gotten)，通常这作为方法调用的边缘效应(side effect)而产生。

一个具有公有实例变量的恰当例子，是类仅作为数据结构，没有行为。亦即，若你要使用一个结构(struct)而非一个类(如果 java 支持结构的话)，那么把类的实例变量声明为公有是合适的。

约定俗成

1. 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。例如：

```
fooBar.fChar = barFoo.lChar = 'c';
```

不要将赋值运算符用在容易与相等关系运算符混淆的地方。例如：

```
if (c++ = d++) {           // AVOID! (Java disallows)
    ...
}
```

应该写成

```
if ((c++ = d++) != 0) {
    ...
}
```

不要使用内嵌(embedded)赋值运算符试图提高运行时的效率，这是编译器的工作。例如：

```
d = (a = b + c) + r;       // AVOID!
```

应该写成

```
a = b + c;
d = a + r;
```

2. 圆括号

一般而言，在含有多种运算符的表达式中使用圆括号来避免运算符优先级问题，是个好方法。

即使运算符的优先级对你而言可能很清楚，但对其他人未必如此。你不能假设别的程序员和你一样清楚运算符的优先级。

```
if (a == b && c == d)    // AVOID!  
if ((a == b) && (c == d)) // RIGHT
```

3. 返回值

设法让你的程序结构符合目的。例如：

```
if (booleanExpression) {  
    return true;  
} else {  
    return false;  
}
```

应该代之以如下方法：

```
return booleanExpression
```

类似地：

```
if (condition) {  
    return x;  
}  
return y;
```

应该写做：

```
return (condition ? x : y);
```

4. 条件运算符"?"前的表达式

如果一个包含二元运算符的表达式出现在三元运算符"?:"的"?"之前，那么应该给表达式添上一对圆括号。例如：

```
(x >= 0) ? x : -x
```

21 种代码的坏味道

应该在编程中尽量避免这 21 种“坏味道”。

1. Duplicated Code

代码重复几乎是最常见的异味了。他也是 Refactoring 的主要目标之一。代码重复往往来自于 copy-and-paste 的编程风格。

2. Long method

它是传统结构化的“遗毒”。一个方法应当具有自我独立的意图，不要把几个意图放在一起。

3. Large Class

大类就是你把太多的责任交给了一个类。这里的规则是 **One Class One Responsibility**。

4. Divergent Change

一个类里面的内容变化率不同。某些状态一个小时变一次，某些则几个月一年才变一次；某些状态因为这方面的原因发生变化，而另一些则因为其他方面的原因变一次。面向对象的抽象就是把相对不变的和相对变化相隔离。把问题变化的一方面和另一方面相隔离。这使得这些相对不变的可以重用。问题变化的每个方面都可以单独重用。这种相异变化的共存使得重用非常困难。

5. Shotgun Surgery

这正好和上面相反。对系统一个地方的改变涉及到其他许多地方的相关改变。这些变化率和变化内容相似的状态和行为通常应当放在同一个类中。

6. Feature Envy

对象的目的是封装状态以及与这些状态紧密相关的行为。如果一个类的方法频繁用 `get` 方法存取其他类的状态进行计算，那么你要考虑把行为移到涉及状态数目最多的那个类。

7. Data Clumps

某些数据通常像孩子一样成群玩耍：一起出现在很多类的成员变量中，一起出现在许多方法的参数中……，这些数据或许应该自己独立形成对象。

8. Primitive Obsession

面向对象的新手通常习惯使用几个原始类型的数据来表示一个概念。譬如对于范围，他们会使用两个数字。对于 `Money`，他们会用一个浮点数来表示。因为你没有使用对象来表达问题中存在的概念，这使得代码变的难以理解，解决问题的难度大大增加。好的习惯是扩充语言所能提供原始类型，用小对象来表示范围、金额、转化率、邮政编码等等。

9. Switch Statement

基于常量的开关语句是 **OO** 的大敌，你应当把他变为子类、`state` 或 `strategy`。

10. Parallel Inheritance Hierarchies

并行的继承层次是 `shotgun surgery` 的特殊情况。因为当你改变一个层次中的某一个类时，你必须同时改变另外一个层次的并行子类。

11. Lazy Class

一个干活不多的类。类的维护需要额外的开销，如果一个类承担了太少的责任，应当消除它。

12. Speculative Generality

一个类实现了从未用到的功能和通用性。通常这样的类或方法唯一的用户是 `testcase`。不要犹豫，删除它。

13. Temporary Field

一个对象的属性可能只在某些情况下才有意义。这样的代码将难以理解。专门建立一个对象来持有这样的孤儿属性，把只和他相关的行为移到该类。最常见的是一个特定的算法需要某些只有该算法才有用的变量。

14. Message Chain

消息链发生于当一个客户向一个对象要求另一个对象，然后客户又向这另一对象要求另一个对象，再向这另一个对象要求另一个对象，如此如此。这时，你需要隐藏分派。

15. Middle Man

对象的基本特性之一就是封装，而你经常会通过分派去实现封装。但是这一步不能走得太远，如果你发现一个类接口的一大半方法都在做分派，你可能需要移去这个中间人。

16. Inappropriate Intimacy

某些类相互之间太亲密，它们花费了太多的时间去钻研别人的私有部分。对人类而言，我们也许不应该太假正经，但我们应当让自己的类严格遵守禁欲主义。

17. Alternative Classes with Different Interfaces

做相同事情的方法有不同的函数 signature，一致把它们往类层次上移，直至协议一致。

18. Incomplete Library Class

要建立一个好的类库非常困难。我们大量的程序工作都基于类库实现。然而，如此广泛而又相异的目标对库构建者提出了苛刻的要求。库构建者也不是万能的。有时候我们会发现库类无法实现我们需要的功能。而直接对库类的修改有非常困难。这时候就需要用各种手段进行 Refactoring。

19. Data Class

对象包括状态和行为。如果一个类只有状态没有行为，那么肯定有什么地方出问题了。

20. Refused Bequest

超类传下来很多行为和状态，而子类只是用了其中的很小一部分。这通常意味着你的类层次有问题。

21. Comments

经常觉得要写很多注释表示你的代码难以理解。如果这种感觉太多，表示你需要 Refactoring。

Eclipse 配置方法

1. 注释模板

在 eclipse 的 preferences 中，选择 java → code style → code Template

1. 添加文件创建日志模板

Comments

- Files
- Types
- Fields
- Constructors
- Methods
- Overriding methods
- Delegate methods
- Getters
- Setters

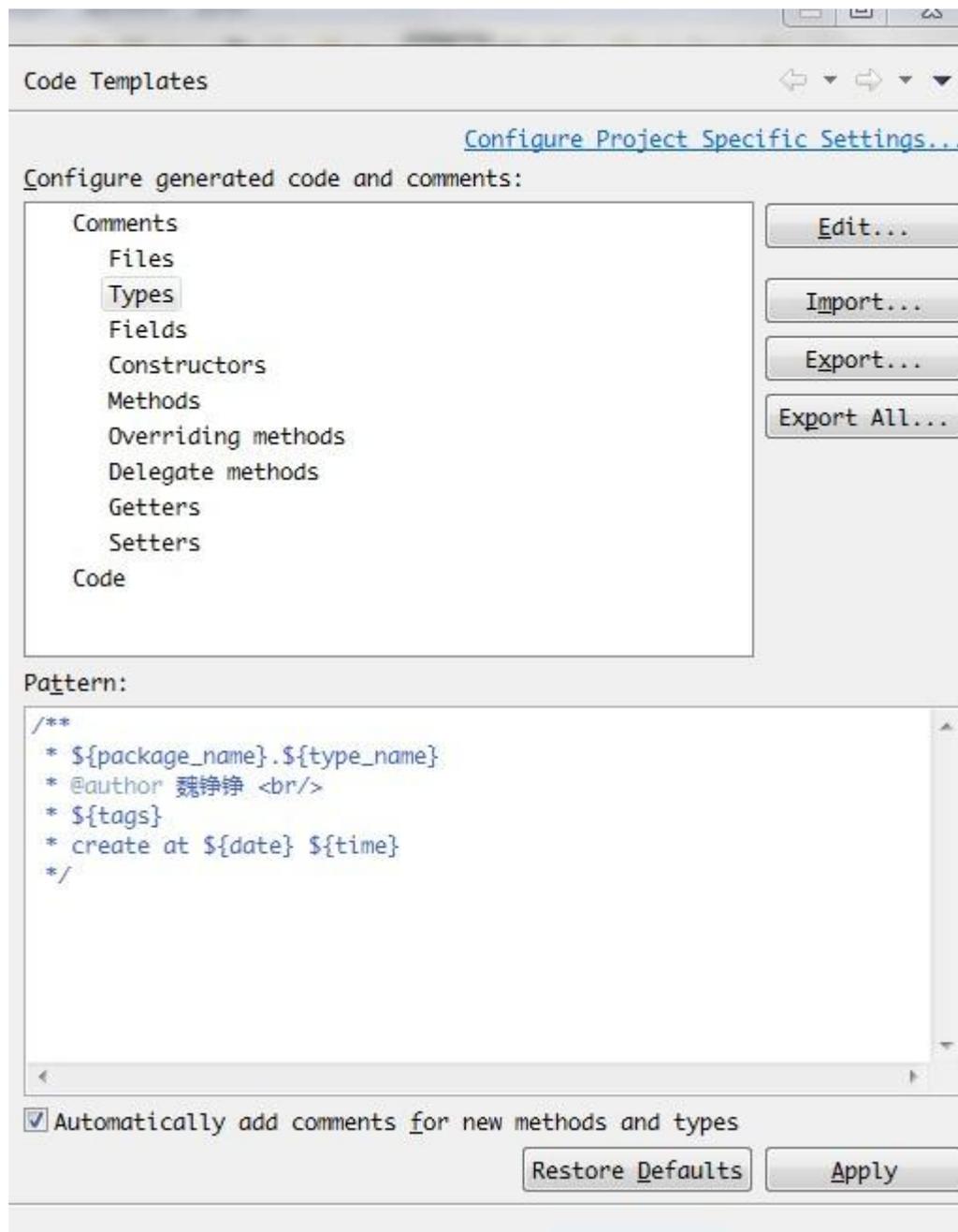
Code

Pattern:

```
/*
 * ${file_name} [V 1.0.0]
 * classes : ${package_name}.${type_name}
 * 魏铮铮 Create at ${date} ${time}
 */
```

Automatically add comments for new methods and types

2. 设置类注释模板



2. 导入方法

在 eclipse 的 preferences 中，选择 java → code style → code Template 中选择 Import，选择附件中的文件。



codetemplates.xml

但是注意修改 类注释 和 文件注释 的作者名称为自己的！

3. 格式化模板

在 eclipse 的 preferences 中，选择 java → code style → formatter 中选择 Import，选择附件中的文件。



formatter.xml

4. XML 格式化

在 eclipse 的 preferences 中，选择 xml → xml files → xml editor 中 做如下设置

