# JAVA编程规范

# 简介

本文提供一整套编写高效可靠的Java代码的标准、约定和指南。它们以安全可靠的软件工程原则为基础，使代码易于理解、维护和增强。而且，通过遵循这些程序设计标准，你作为一个Java软件开发者的生产效率会有显著提高。经验证明，若从一开始就花时间编写高质量的代码，则在软件开发阶段，对代码的修改要容易很多。最后，遵循一套通用的程序设计标准将带来更大的一致性，使软件开发团队的效率明显提高。

## 最根本的原则：

运用常识。当找不到任何规则或指导方针，当规则明显不能适用，当所有的方法都失效的时侯：运用常识并核实这些基本原则。这条规则比其它所有规则都重要。常识是必不可少的。

# 程序设计标准

Java的程序设计标准很重要，原因在于它将提高开发团队各成员的代码的一致性。一致性的提高会使代码更易理解，这意味着它更易开发和维护。从而降低了应用程序的总开发成本。

你必须牢记的是：你的Java代码在你已离开并开始另一个项目之后，会保留相当长的一端时间。因此开发过程中一个很重要的目标就是要确保在开发成员或开发团队之间的工作可以顺利交接，不必花很大的力气便能理解已编写的代码，以便继续维护和改进以前的工作。如果代码难以理解，很有可能被废弃和重写。

### 命名约定

我们将在整个标准中讨论命名约定，所以让我们先讨论几个基本点：

1.使用可以准确说明变量/字段/类的完整的英文描述符。例如，采用类似firstName，grandTotal或CorporateCustomer这样的名字。虽然象x1，y1或fn这样的名字很简短，输入起来容易，但是我们难以知道它们代表什么、结果是什么含义，因而使代码难以理解、维护和改进。

2.采用该领域的术语。如果用户称他们的“客户”(clients)为“顾客”(customers)，那么就采用术语Customer来命名这个类，而不用Client。许多程序开发者会犯的一个错误是，不去使用工业或领域里已经存在着很完美的术语时，却生造出一些普通词汇。

3.采用大小写混合，提高名字的可读性。一般应该采用小写字母，但是类和接口的名字的首字母，以及任何中间单词的首字母应该大写。

4.尽量少用缩写，但如果一定要使用，就要谨慎地使用。这意味着应该保留一个标准缩写的列表，明智地从中选取，并且在使用时保持一致。例如，想对单词“number”采用缩写，那么可从nbr，no或者num中选取一个，说明一下采用了哪一个（具体是哪个倒无所谓），并且只使用这一种形式。

5.避免使用长名字（最好不超过15个字母）。虽然PhysicalOrVirtualProductOrService看起来似乎是个不错的类名，但是这个名字太长了，应该考虑重新给它起个短一点的名字，比如象Offering。

6.避免使用相似或者仅在大小写上有区别的名字。例如，不应同时使用变量名persistentObject和persistentObjects，以及anSqlDatabase和anSQLDatabase。

7.避免使用下划线作为名字的首末字母。以下划线为首末字母的名字通常为系统保留，除预处理定义之外，一般不用作用户命名。更重要的是，下划线经常造成麻烦而且难输入，所以尽量避免使用。

### 注释约定

我们还会对注释约定进行讨论，所以，我们先谈谈一些基本点：

1.注释应该增加代码的清晰度。代码注释的目的是要使代码更易于被同时参与程序设计的开发人员以及其他后继开发人员理解。

2.如果你的程序不值得注释，那么它也很可能也不值得运行[NAG95]。

3.避免使用装饰性内容，也就是说，不要使用象广告横幅那样的注释语句。二十世纪六十年代和七十年代，COBOL程序员们养成了画框的习惯，典型的是用星号将他们的内部注释圈起来。当然，这给他们的艺术创造欲一个发泄方式，但是坦白地说，这只是在大量浪费时间，并不能给最终的产品增加丝毫价值。要写的是清晰的代码，不是外表可爱的代码。此外，由于有些字体的显示和打印是成比例的，但有些又不是，所以无法将那些框排整齐。

4.保持注释的简洁。最好的注释应该是简单明了的注释。注释不必洋洋洒洒，只需提供足够的信息，使别人能够理解你的代码。

5.先写注释，后写代码。写代码注释的最好方法是在写代码之前就写注释。这使你在写代码之前可以想想代码的功能和运行。而且这样确保不会遗漏注释。另一种方法是边写代码边写注释。因为注释可以使代码更易理解，所以在程序开发的过程中，也可以利用这一点。如果打算花些时间写注释，那么至少你应从这个过程中获得些什么[AMB98]。

6.注释信息不仅要包括代码的功能，还应给出原因。例如，下面例1中的代码显示金额在$1,000以上（包括$1,000）的定单可给予5%的折扣。为什么要这样做呢？难道有一个商业法则规定大额定单可以得到折扣吗？这种给大额定单的特殊是有时限的呢，还是一直都这样？最初的程序设计者是否只是由于慷慨大度才这样做呢？除非它们在某个地方（或者是在源代码本身，或者是在一个外部文档里）被注释出来，否则你不可能知道这些。

例1.1

if(grandTotal>=1000.00)

{

grandTotal=grandTotal\*0.95;

}

#### Java注释语句类型

Java有三种注释语句风格：以/\*\*开始，\*/结束的文档注释，以/\*开始，以\*/结束的C语言风格注释，以及以//开始，代码行末尾结束的单行注释。下表是对各类注释语句建议用法的一个概括，也给出了几个例子。

|  |  |  |
| --- | --- | --- |
| 注释语句类型 | 用法 | 示例 |
| 文档注释 | 在接口、类、成员函数和字段声明之前紧靠它们的位置用文档注释进行说明。文档注释由javadoc处理，为一个类生成外部注释文档，如下所示。 | /\*\*  Customer（顾客）.顾客是指作为我们的服务及产品的销售对象的任何个人或组织。  @authorS.W.Ambler  \*/ |
| C语言风格注释 | 采用C语言风格的注释语句将无用的代码注释掉。保留这些代码是因为用户可能改变想法，或者只是想在调试中暂时不执行这些代码。 | /\*  这部分代码已被它前面的代码替代，所以于1999年6月4日被B.Gustafsson注释掉。如果两年之后仍未用这些代码，将其删除。  ...（源代码）  \*/ |
| 单行注释 | 在成员函数内部采用单行注释语句对业务逻辑、代码片段和临时变量声明进行说明。 | //因为让利活动  //从1995年2月开始，  //所以给所有超过$1000的  //发货单5%的折扣。 |

一件很重要的事情是，你的机构应该制订一套如何使用C语言风格注释和单行注释的标准，并始终严格遵守。使用一种注释方式来说明业务逻辑，使用另一种方式注释掉旧的代码。业务逻辑采用单行注释，因为这样可以将注释和代码放在同一行（这又叫做“内联”）。采用C语言风格的注释屏蔽掉旧的代码，因为这样可以同时注释掉数行。C语言风格注释看起来很象文档注释，所以为了防止混淆，不应在别处使用。

注意行末注释。[MCO93]强烈反对采用行内注释，即在一行的末尾加上注释。他指出，这种注释必须在代码的右端对齐，这样才能避免代码结构看起来混乱。结果，这些注释的格式难以划一。“如果你使用了很多这样的注释，则要花时间去将它们排列整齐。这些时间并没有花在更多地了解代码上，而完全花在了敲击空格键和制表符这种冗长乏味的工作上。”他又指出，行末注释也难以维护。因为当该行程序的代码加长时，它们会将这些注释挤出该行，如果你将它们排齐了，你不得不对余下的注释做同样的工作。

## ****编码规则****

1、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IO对象需要close()，需要分别对每个对象的close()方法进行try-catch,防止一个IO对象关闭失败其他IO对象都未关闭。

手动控制事务提交也要进行关闭，对大对象进行关闭操作

示例：

|  |
| --- |
| try  {      // ... ...  }  catch(IOException ioe)  {       //... ...  }  finally  {      try      {            out.close();      }      catch (IOException ioe)      {           //... ...  }      try      {            in.close();      }      catch (IOException ioe)      {           //... ...      }  } |

2、 系统非正常运行产生的异常捕获后，如果不对该异常进行处理，则应该记录日志。

说明：此规则指通常的系统非正常运行产生的异常，不包括一些基于异常的设计。若有特殊原因必须用注释加以说明。

logger.error(ioe，“[类.方法]描述”,参数);

示例：

|  |
| --- |
| try  {      //.... ...  }  catch (IOException ioe)  {      logger.error(ioe);  } |

3、自己抛出的异常必须要填写详细的描述信息。

说明：便于问题定位。

示例：

|  |
| --- |
| throw new  IOException("Writing data error! Data: " + data.toString()); |

4(删除)、在程序中使用异常处理还是使用错误返回码处理，根据是否有利于程序结构来确定，并且异常和错误码不应该混合使用，推荐使用异常。

说明：

一个系统或者模块应该统一规划异常类型和返回码的含义。

但是不能用异常来做一般流程处理的方式，不要过多地使用异常，异常的处理效率比条件分支低，而且异常的跳转流程难以预测。

注意：JDK1.5 程序内部的错误码可以使用枚举来表示。

5、 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)  if ((a | b) && (a & c))      (2)  if ((a | b) < (c & d))       (3) |

如果书写为

|  |
| --- |
| high << 8 | low  a | b && a & c  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。

6、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，不应直接使用数字，必须用有意义的静态变量或者枚举来代替。

示例：如下的程序可读性差。

|  |
| --- |
| if (state == 0)  {      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  } |

注意：JDK 1.5 下建议使用枚举来表示。

7(删除)、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，int [] index 表示声明了一个int数组(int [])叫做index

示例：

如下程序可读性差：

|  |
| --- |
| public int getIndex()[]  {      ....  } |

如下程序可读性好：

|  |
| --- |
| public int[] getIndex()  {      ....  } |

8、 不要使用 System.out 与 System.err 进行控制台打印，应该使用工具类(如：日志工具)进行统一记录或者打印。

说明：代码发布的时候可以统一关闭控制台打印，代码调试的时候又可以打开控制台打印，方便调试。

9、集合必须指定模板类型

说明：方便程序阅读，除去强制转换代码

示例：

|  |
| --- |
| Map<String,MyObject> map = new HashMap<String,MyObject>(); |

10、一个文件不要定义两个类(并非指内部类)。

说明：方便程序的阅读与代码的维护

注意：一个文件定义两个类，在play框架的生产模式会报错，因为play重写了class的加载方法，生产模式启动时进行预编译，这时会保存；但是在开发模式则不会报错，因为开发模式不会进行预编写，而是在运行时编译，jvm运行时编译则不会出错。

11(抽取成一个日志公用类)、 对Debug，Info级别日志输出前必须对当前的调试等级先进行判断。  
说明：日志一般都会有不少字符串的处理，如果不是Debug级别就没有必要进行处理  
示例：

|  |
| --- |
| if (logger.debugEnable())  {  logger.debug(“request : ” + request.getMethod());  } |

12、不要使用循环将集合转为数组，可以使用集合的toArray()方法。

说明：更好的性能，代码更加简洁

(1、数据量多的情况下，ArrayList跟LinkedList是否应该区分循环访问方式，即（for跟foreach）的区别；

(2、当需要使用Map进行拼接数据的时候，HashMap跟LinkedHashMap使用循环打印出来的数据顺序跟添加的顺序是不一样的，是否也需要区分开；  
示例：

|  |
| --- |
| ArrayList list = new ArrayList();  list.add....  String [] array = new String[list.size()];  list.toArray(array); |

13、JDK 1.4中大量字符串的“相加”操作应该使用StringBuffer。JDK 1.5以上版本大量字符串“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。  
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”); |

JDK 1.4使用 StringBuffer。JDK1.5如果不涉及线程安全用StringBuilder替代以上示例。

14、对类中日志工具对象logger应声明为static.   
说明：防止重复new 出logger对象(logger指各种日志工具类，可以是log4j，jdk logger，内部API等，尽管一些logger对LogFactory工厂有一些优化，但是我们也必须防止代码没有必要的运行)。

15、不能用“==”比较两个字符串(Long对象等)内容相等,能用基本类型就是基本类型

说明：两个字符串在比较内容是否相等的时候，如果使用“==”，当两个字符串不是指向内存中同一地址，那么即使这两个字符串内容一样，但是用“==”比较出来的结果也是false。所以两个字符串在比较内容是否相等的时候一定要使用“equals”方法。

示例：

|  |
| --- |
| public class Test {  public static void main(String[] args)  {  String a = new String("a");  String a2 = "a";  if(a == a2)  {  System.out.println("a == a2 return true.");  }  else  {  System.out.println("a == a2 return false.");  }  if(a.equals(a2))  {  System.out.println("a.equals(a2) return true.");  }  else  {  System.out.println("a.equals(a2) return false.");  }  }  } |

最终输出的结果为：  
a == a2 return false.  
a.equals(a2) return true.

16、对list做foreach循环时，循环代码中不能修改list的结构

说明：在jdk1.5版以上的foreach循环写法中，不能在循环代码中对正在循环的list的结构进行修改，即对list做add、remove等操作，如果做了这些操作，必须立即退出循环，否则会抛出异常。

17、避免空指针异常,增加静态代码检查

 说明：空指针异常是编码过程中最常见的异常，在使用一个对象的时候，如果对象可能为空，并且使用次对象可能会造成空指针异常，那么需要先判断对象是否为空，再使用这个对象。

18、避免下标越界,增加静态代码检查

说明：访问数组、List等容器内的元素时，必须首先检查下标是否越界，杜绝下标越界异常的发生。

19、将字符串转数字时没有捕获NumberFormatException异常,写工具类

说明：调用Java方法将字符串转换为数字时，如果字符串的格式非法，会抛出运行时异常NumberFormatException。

示例：

错误例子：

|  |
| --- |
| public Integer getInteger1(String number)  {  // 如果number格式非法，会抛出NumberFormatException  return Integer.valueOf(number);  } |

正确的处理方法如下：

|  |
| --- |
| public Integer getInteger2(String number)  {  try  {  return Integer.valueOf(number);  }  catch (NumberFormatException e)  {  ...  //记录日志异常信息  return null;  }  } |

注意：在捕获异常后一定要记录日志。

20、循环体编码时不考虑性能，循环体中包含不需要的重复逻辑。(考虑循环体的性能)

说明：循环体是软件中最容易造成性能问题的地方，所以在进行循环体编码时务必考虑性能问题。在循环体内重复使用且不会变化的资源（如变量、文件对象、数据库连接等），应该在循环体开始前构造并初始化，避免在循环体内重复和构造初始化造成CPU资源的浪费。除非业务场景需要，避免在循环体内构造try...catch块，因为每次进入、退出try...catch块都会消耗一定的CPU资源，将try...catch块放在循环体之外可以节省大量的执行时间。

21、equals操作时没有将常量放在equals操作符的左边,增加静态代码检查

说明：字符串变量与常量比较时，先写常量，这样可以避免空指针异常。

示例：

|  |
| --- |
| public static final String SP\_NAME = "SPNAME";  public void equalsConst()  {      String temp = null;      //错误;//在temp为null的情况下，此时会导致意想不到的异常抛出，如果将常量放在左边，就不会有这种问题。      if(temp.equals(SP\_NAME))      {          temp = SP\_NAME + "AA";      }  } |

## ****命名规范****

### 类名、变量名（非final）、方法名

### 驼峰式命名

### 不能使用没有任何含义的英文字母进行命名

### 不能使用拼音进行命名，统一使用准确的英文进行命名

不采用简写方式命名（除公认的常用简写）

### 包名

以cn.pp.ppbase.项目名.xxx 命名，都采用小写，连续的单词只是简单地连接起来，不使用下划线。

### 接口与类的命名

接口不要以 I 开头。如：IUserService，而直接采用具体的命名方式如UserService

### 抽象类命名

抽象类命名使用：Abstract+名词的方式进行命名，如：AbstractMessage。

### 实现类命名

对于只有一个实现类的情况，通常先不采用接口方式，直接采用实现类即可，如果有多个实现类，则采用名称+抽象名称，如UserMessage、AdminMessage。

注意：项目中的Service层，目前大多只有一个实现，所以不采用接口方式，直接使用具体类即可。

### 工具类命名

统一使用：名词+Util进行命名，如：UrlUtil、StringUtil……

### 变量命名

所有变量（描述状态的除外）统一以准确的名词性英文命名，名词性英文语法如下：

普通名词，如：user、clubMember、order等

组合名词，即名词+名词，如：clubMemberServiceOrder，paymentResult等

带修饰语（定语）名词，如：checkingResult，messageSendingTask（主动，用ing），closedOrder（已经关闭的订单，用ed）

描述状态的变量统一以准确的形容词性英文命名，形容词性英文语法如下：

普通形容词，如：active，inactive，valid，invalid等

动词转形容词，表示状态，如：close > closed，stop > stopped等

动词转形容词，标示能动，如：delete > deletable，use > useful等

对于反映状态的变量，不要在命名前面加“is”，因为自动生成的get方法，对于boolean值，方面名自动会变为is\*\*\*

常量用全大写，单词之间用“\_”分割，如：

public static final String QUERYCLUBMEMBER\_HQL=”…”;

对于成员变量，其名称可省略所属类的名称，如：

|  |
| --- |
| public class User {   private String clubMemberName; //NOT GOOD   private String name; //GOOD  } |

### 5、方法命名

方法名命名规则：

动词：init，validate，pay等

动词+介词+（宾语）：compareTo，getById等

动宾短语（表示动作）：createUser，publishPrivilege

谓表短语（表示状态）：isClosed，isUserExisted，canUserBeDeleted等

单复数的使用，对于一些动作需要操作多个对象，方法名要通过名词复数反映出来，例如：

deleteOrders（删除订单，可能多个）

如果一个方法设计两个动作或对象，用“And”连接，如：

createProductAndTag，但一般情况下，不推荐一个方法做多于一件事情

如非必要，或常见的简写，尽量不要使用简写，因为简写很难阅读，容易误解，宁愿方法名稍微长一些，也要让方法名用正确的短语来表述，由于一个方法一般不建议做多于一件事情，所以方法名一般不会太长。

一些Java业界约定俗成的方法命名：

与…进行比较：compareTo

获取单例对象：getInstance

是否相等：equals

初始化：init

### 系统的命名约定

#### 接口、类的命名约定

|  |  |
| --- | --- |
| 类名 | 示例 |
| Struts Action类 | \*\*\*Action OrderAction |
| Struts Interceptor类 | \*\*\*Interceptor AuthenticationInterceptor |
| 系统监控器类 | \*\*\*Monitor MemcachedMonitor |
| 逻辑层类 | \*\*\*Service UserService |
| VO/DTO（视图对象） | 实体名+VO UserVO |
| Entity（领域对象） | 实体名 User |
| DAO层 | \*\*\*Dao UserDao |

#### 常规方法的命名约定

|  |  |
| --- | --- |
| 方法名 | 示例 |
| 创建 | create\*\*\* create createUser |
| 更新 | update\*\*\* update updateUser |
| 删除 | delete\*\*\* delete deleteOrder |
| 批量删除 | delete\*\*\*(复数) deleteOrders |
| 获取单个对象 | get\*\*\* get getUser |
| 获取单个对象根据某个参数 | getBy\*\*\* getMarketingActivityByCode |
| 批量获取多个对象 | batchGet\*\*\* |
| 列表（不分页） | list\*\*\*（复数） listOrders |
| 列表（分页） | paged\*\*\*（复数） pagedOrders |
| 激活/启用 | activate\*\*\* activeMarketingActivity |
| 停用 | inactivate/stop\*\*\* inactivateMarketingActivity |

如果类名已经包含一些名称，方法和操作可以不写响应的名称，例如UserDao  的方法get  而不用getUser