**Java编程规范**

**目录**

1       概述.... 5

1.1      目标：.... 5

1.2      原则：.... 5

2       Java编程命名规范.... 5

2.1      包的命名标准.... 6

**2.1.1命名包**.... 6

2.2      类、接口的命名标准.... 6

**2.2.1**         **命名类**.... 6

**2.2.2**         **命名接口**.... 6

**2.2.3**         **命名编译单元**.... 6

2.3      成员函数命名标准.... 7

**2.3.1**         **命名成员函数**.... 7

**2.3.2**         **获取函数**.... 7

**2.3.3**         **设置函数**.... 7

**2.3.4**         **命名构造函数**.... 8

**2.3.5**         **成员函数的可见性**.... 8

2.4      字段/属性 命名标准.... 9

**2.4.1**         **命名字段**.... 9

**2.4.2**         **命名组件（部件）**.... 9

2.5      局部变量命名标准.... 9

**2.5.1**         **命名流**.... 9

**2.5.2**         **命名循环计数器**.... 10

**2.5.3**         **命名异常对象**.... 10

2.6      成员函数参数命名标准.... 10

3      Java编程注释规范.... 10

3.1      注释约定：.... 10

**3.1.1**         **Java 注释语句类型**.... 10

**3.1.2**         **javadoc标志**.... 11

3.2      包的注释标准.... 12

3.3      类、接口的注释标准.... 12

**3.3.1**         **注释类**.... 12

**3.3.2**         **注释接口**.... 13

4      成员函数注释标准.... 13

4.1      成员函数的函数头.... 13

**4.1.1**         **内部注释**.... 14

4.2      字段/属性 注释标准.... 16

**4.2.1**         **注释一个字段**.... 16

4.3      局部变量注释标准.... 16

**4.3.1**         **声明和注释局部变量**.... 16

4.4      成员函数参数注释标准.... 16

5      Java编程格式规范.... 17

5.1      if-else-elseif. 17

5.2      switch. 17

5.3      while. 17

5.4      do … while. 18

5.5      for. 18

5.6      try-catch. 18

6       Java错误和异常处理规范.... 18

6.1      Java程序错误和异常处理原则.... 18

6.2      在底层的类中.... 18

6.3      在高层的类中.... 19

7      Java编程一般约定：.... 20

7.1      将公共和保护接口最小化.... 20

7.2      编程一般约定.... 20

8      其他.... 21

8.1      编写清晰整洁的代码的原则.... 21

**8.1.1**         **给代码加上注释**.... 22

**8.1.2**         **让代码分段/缩进**.... 22

**8.1.3**         **遵循 30 秒条法则**.... 22

**8.1.4**         **写短小单独的命令行**.... 22

**8.1.5**         **说明运行顺序**.... 22

8.2      为什么采用存取函数？.... 22

**1      概述**

**1.1 目标：**

建立一个可行可操作的编程标准、约定和指南，以规范我们的代码开发工作。提高代码的可读性，提高系统的健壮性、稳定性、可靠性。通过遵循这些程序设计标准，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。

**1.2 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。

**当你要违反一个标准时，注释你的原因**

当你要违反一个标准时，你这样做的时候，你必须注释你违反这个标准的原因，违反标准的潜在原因以及在何种条件（可能/必须）满足的情况下，标准可以被应用到这个情况中。

**代码的可维护性**

你的 Java 代码在你已离开并开始另一个项目之后，会保留相当长的一段时间。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。

**2      Java编程命名规范**

**一般约定：**

命名时应始终采用完整的英文描述符。此外，一般应采用小写字母，但类名、接口名以及任何非初始单词的第一个字母要大写。

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” mers)，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，但如果需要使用，可以选择单词的头4个字母作为缩写。如：DeptInfoValue，MapActi, MapActiInst 等等。
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。

**2.1 包的命名标准**

**2.1.1命名包**

关于包的命名有几条规则，这些规则是：

1、 包的命名都是使用小写的英文字母组成，每个包名称之间用点号分隔开来。如java.lang;javax.servlet.http 等。

2、 全局包的名字用所在机构的 Internet 保留域名开头。如：com.sun   com.oracle. 等等

3、 我们的命名约定：

com.excellence.common  excellence下的通用文件，可能被以后不同项目通用的类文件。

com.excellence.exoa4j exoa2004forjava项目的java分模块分类存放。

在com.excellence.exoa4j下面，我们在按照模块，分开各自的包。

**2.2 类、接口的命名标准**

**2.2.1命名类**

标准 Java类约定是使用完全的英文描述符，所有单词的第一个字母要大写，并且单词中大小写混合。

类名应是单数形式。

示例：

Customer   Employee        Order    OrderItem        FileStream        String

     相关的类如果使用了某些已经约定的开发模式，应该使用相关约定的命名：如：LoggerFactory , UserDAO 等。

     在同一个模块中的相关类，可以采用同样开头的名称以区分他们的相关性。如：UserValue, UserDAO 等。

**2.2.2命名接口**

采用完整的英文描述符说明接口封装，所有单词的第一个字母大写。习惯上，名字后面加上后缀 able,. ible 或者 er。

示例：

Runnable, Contactable,

Prompter, Singleton

**2.2.3命名编译单元**

编译单元，在这个情况下是一个源码文件，应被赋予文件内定义的主要的类或接口的名字。用与包或类相同的名字命名文件，大小写也应相同。扩展名 .java 应作为文件名的后缀。

示例：

Customer.java

Singleton.java

DeptInfo.java

**2.3 成员函数命名标准**

**2.3.1命名成员函数**

成员函数的命名应采用完整的英文描述符，大小写混合使用：所有中间单词的第一个字母大写。**成员函数名称的第一个单词常常采用一个有强烈动作色彩的动词。**

**示例：**

openAccount()   printMailingLabel() createUser()   delete()

这种约定常常使人一看到成员函数的名称就能判断它的功能。虽然这种约定要使开发者多做一些输入的工作，因为函数名常常较长，但是回报是提高代码的可理解性。

**2.3.2获取函数**

获取函数作为一个成员函数，返回一个字段的值。除了布尔字段之外，应采用 get 作为字段的前缀；布尔字段采用 is 作为前缀。

**示例：**

public String getUserCode() {

    return userCode;

  }

  public String getUserName() {

    return userName;

  }

  getFirstName()

getAccountNumber()

isPersistent()

isAtEnd()

遵循这个命名约定，显然，成员函数将返回对象的字段，布尔型的获取函数将返回布尔值“真”或者“假”。这个标准的另一个优点是：它遵循 beans development kit (BDK) 对获取成员函数采用的命名约定 [[DES97](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#DES97#DES97)]。它的一个主要的缺点是 get 是多余的，需要额外的录入工作。

按照jbuilder中bean/properties工具生成的设置和获取函数。

**2.3.3设置函数**

设置函数，也叫变值函数，是可以修改一个字段值的成员函数。无论何种字段类型，都要在字段名的前面加上 set 前缀。

**示例：**

public void setUserCode(String userCode) {

    this.userCode = userCode;

}

public void setUserName(String userName) {

    this.userName = userName;

  }

setFirstName(String name)

setAccountNumber(int accountNumber)

setReasonableGoals(Vector goals)

setPersistent(boolean isPersistent)

setAtEnd(boolean isAtEnd)

按照这种命名约定，显然是一个成员函数设定一个对象的字段值。这个标准的另一个优点是：它遵循 beans development kit (BDK) 对设置函数采用的命名约定 [[DES97](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#DES97#DES97)]。 它的一个主要的缺点是 set 是多余的，需要额外的录入。

按照jbuilder中bean/properties工具生成的设置和获取函数。

**2.3.4命名构造函数**

构造函数是在一个对象初次生成时，完成所有必需的初始化的成员函数。构造函数与它所属类的名字总是相同的。例如，类 **Customer** 的构造函数是 **Customer()**。注意大小写一致。

**示例：**

Customer()

SavingsAccount()

PersistenceBroker()

这个命名约定由 Sun 公司设定，必须严格遵守。

**2.3.5成员函数的可见性**

良好的程序设计应该尽可能减小类与类之间耦合，所遵循的经验法则是：尽量限制成员函数的可见性。如果成员函数没必要公有 (public)，就定义为保护 (protected)；没必要保护 (protected)，就定义为私有 (private)。

|  |  |  |  |
| --- | --- | --- | --- |
| **可见性** | **Java关键字** | **说明** | **正确用法** |
| 公共 | **public** | 公有成员函数可被任何其它对象和类的成员函数调用。 | 当该成员函数必须被该函数所在的层次结构之外的其他对象和类在访问时。 |
| 受保护 | **protected** | 被保护的成员函数可被它所在的类或该类的子类的任何成员函数调用。 | 当该成员函数提供的行为被它所在类的层次结构内部而非外部需要时。 |
| 专用 | **private** | 私有成员函数只可以被该类所在的其它成员函数调用，该类的子类不可以调用。 | 当该成员函数所提供的行为明确针对定义它的类时。私有成员函数常常是重新分配要素的结果。重新分配要素又叫“重组”，指类内其它成员函数封装某一个特定行为的做法。 |
| 缺省 | 无关键字，简单地使其为空白 | 成员函数对于同一包中的其它所有类实际上都是公共的，但是对该包外部的类是专用的。 有时，它称为包可见性或友好的可见性。 | 这是一个有趣的功能，但要小心使用。 一般不建议使用。 |

**2.4 字段/属性 命名标准**

*field* 这个词在这里指的是字段，Beans Development Kit (BDK) 叫它“属性” [[DES97](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#DES97#DES97)]。字段是说明一个对象或者一个类的一段数据。字段可以是象字符串或者浮点数这样的基本数据类型，也可以是一个对象，例如一个消费者或者一个银行帐户。

**2.4.1命名字段**

应采用完整的英文描述符来命名字段 [[GOS96](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#GOS96#GOS96)]，[[AMB98](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#AMB98#AMB98)]，以便使字段所表达的意思一目了然。象数组或者矢量这样是集合的字段，命名时应使用复数来表示它们代表多值。

示例：

firstName   zipCode unitPrice  discountRate    orderItems（复数）

**2.4.2命名组件（部件）**

应采用完整的英文描述符命名组件（接口部件），名字的后缀是组件类型名。这让你容易区分一个组件的目的和它的类型，容易在一个表里找到各个组件（许多可视化的编程环境在一个 Applet 程序或者一个应用程序中提供了一个所有组件的列表。如果所有名字都是类似于 button1， button2, & 这样的话，很容易混淆）。

示例：

okButton        customerList        fileMenu        newFileMenuItem

**2.5 局部变量命名标准**

一般说来，命名局部变量遵循与命名字段一样的约定，即使用完整的英文描述符，任何非开头的单词的第一个字母要大写。

但是为方便起见，对于如下几个特殊的局部变量类型，这个约定可以放宽：

* 流
* 循环计数器
* 异常

**2.5.1命名流**

当有一个单输入和/或单输出流在一个成员函数中被打开、使用和关闭时，通常的约定是对这些流分别采用 in 和 out [[GOS96](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#GOS96#GOS96)] 来命名。对于既用于输入又用于输出的流，采用 inOut 来命名。

一个常用的取代这种约定的方法是分别采用 inputStream，outputStream 和 ioStream 这样的名字，而不是 in，out 和 inOut，虽然这与 Sun 公司的建议相抵触。

**2.5.2命名循环计数器**

因为局部变量常用作循环计数器，并且它为 C/C++ 所接受，所以在 Java 编程中，可以采用 i， j 或 k 作为循环计数器 [[GOS96](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#GOS96#GOS96)]。 若采用这些名字作为循环计数器，要始终使用它们。

概括起来说，i，j，k 作为计数器时， 它们可以很快被输入，它们被广泛的接受。

**2.5.3命名异常对象**

因为在 Java 代码中异常处理也非常普遍，所以字母 e 作为**一般的异常**符被广泛地接受。

如果有多个异常同时出现，应该使用该类型异常全称的缩写表示。

catch(SQLException sqlexception)

**2.6 成员函数参数命名标准**

命名参数

参数命名遵循与局部变量命名完全一样的约定。对于局部变量，名字隐藏是一个问题。

示例：

customer        inventoryItem       photonTorpedo       in      e

一个可行的取代方法，是采用局部变量的命名约定，但在名字之前加入“a”或“an”。加上“a”或“an”有助于让参数与局部变量和字段区分开来，避免名字隐藏的问题。这种方法较好。

示例：

aCustomer   anInventoryItem     aPhotonTorpedo  anInputStream       anException

但是，如果是设置/获取函数，由于采用了jbuilder 下的bean/properties工具生成了代码，可以不采用这种命名方法。而采用，this关键字将局部变量和函数参数区分开。

如：public void setUserName(String userName) {

    this.userName = userName;

  }

**3   Java编程注释规范**

**3.1 注释约定：**

**3.1.1Java 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。  @author S.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。 |

使用单行注释方式来说明业务逻辑，使用C 语言风格的注释屏蔽旧的代码。

**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。

**3.1.2javadoc标志**

Sun 公司的 Java Development Kit (JDK) 中有一个名为 *javadoc* 的程序。它可以处理 Java 的源代码文件，并且为 Java 程序产生 HTML 文件形式的外部注释文档。Javadoc 支持一定数目的标记，标识注释文档中各段起始位置的保留字。详情请参考 JDK *javadoc* 文档。

|  |  |  |
| --- | --- | --- |
| **标记** | **用于** | **目的** |
| @author name | 类、  接口 | 说明特定某一段程序代码的作者。每一个作者各有一个标记。 |
| @deprecated | 类、  成员函数 | 说明该类的应用程序编程接口 (API) 已被废弃，因此应不再使用。 |
| @exception name description | 成员函数 | 说明由成员函数发出的异常。一个异常采用一个标记，并要给出异常的完整类名。 |
| @param name description | 成员函数 | 用来说明传递给一个成员函数的参数，其中包括参数的类型/类和用法。每个参数各有一个标记。 |
| @return description | 成员函数 | 若成员函数有返回值，对该返回值进行说明。应说明返回值的类型/类和可能的用途。 |
| @since | 类、成员函数 | 说明自从有 JDK 1.1 以来，该项已存在了多长时间。 |
| @see ClassName | 类、接口、成员函数、字段 | 在文档中生成指向特定类的超文本链接。可以并且应该采用完全合法的类名。 |
| @see ClassName#member functionName | 类、接口、成员函数、字段 | 在文档中生成指向特定成员函数的超文本链接。可以并且应该采用完全合法的类名。 |
| @version text | 类、接口 | 说明特定一段代码的版本信息。 |

你注释代码的方式很大地影响着你的工作效率以及所有维护改进代码的后继开发者的工作效率。在软件开发过程中及早注释代码，会促使你在开始撰写代码之前仔细考虑这些代码，从而带来更高的工作效率。而且，当你重新阅读数天前或者数星期前所写的代码时，你可以很容易地判断出当时你是怎么想的，因为这一切都有记录。

**3.2 包的注释标准**

**注释包**

应保有一个或者多个外部文档以说明你的机构所创建的包的用途。对于每个包，应说明：

1. 包的基本原理。其他开发者需要了解一个包是用来做什么的，这样他们才能判断是否可以用它，如果是一个共享包，他们可以判断是否需要改进或是扩展它。
2. 包中的类。在包中要包含一个类和接口的列表，每个类或接口用简短的一行文字来说明，以便让其他的开发者了解这个包中含有什么。

技巧：生成一个以包名命名的 HTML 文件，将它放到包的适当的目录中去。这个文件应具有后缀 *.htm*。

**3.3 类、接口的注释标准**

**3.3.1注释类**

以下的信息应写在文档注释中紧靠类的定义的前面：

1. 类的目的和作用。开发者需要了解一个类的一般目的，以判断这个类是否满足他们的需求。养成一个注释与类有关的任何好东西的习惯。
2. 已知的问题。如果一个类有任何突出的问题，应说明出来，让其他的开发者了解这个类的缺点/难点。此外，还应注明为什么不解决问题的原因。
3. 类的开发/维护历史。通常要包含一个历史记录表，列出日期、类的作者和修改概要。这样做的目的是让进行维护的程序员了解过去曾对一个类所做的修改，是谁做了什么样的修改。
4. 版权信息。

对于以上几点：其中类的目的和作用这项是必须要填写的。采用Javadoc形式标志的一个示例如下：

/\*\*

 \* <p>类说明: LoggerFactory,创建Logger的工厂类</p>

 \* <p>Copyright 2003: Excellence Network Co.,LTD</p>

 \* @author fjy

 \* @version 1.0

 \* @since 2003-06-10

 \* @modified fjy 2003-06-13  修改了XXXX

\* @modified fjy 2003-06-23  修改了XXXX

 \*/

**3.3.2注释接口**

以下的信息应写在文档注释中紧靠接口定义的前面：

1. 目的。在其他开发者应用一个接口之前，他们需要理解接口封装的概念。换句话说，他们需要了解接口的目的。一个好的检测是否有必要定义一个接口的方法是你是否可以很容易地说明它的目的。 如果说明起来有困难，很可能从一开始起就不需要这个接口。在 Java 中接口的概念较新，所以人们对如何正确使用它们还不是很有经验，常常滥用它们。
2. 它应如何被使用以及如何不被使用。开发者需要了解接口应如何使用以及如何不被使用。

因为成员函数的标识在接口中定义，所以对于每个成员函数的标识应遵循第二章中所讨论的注释约定。

示例如下：

/\*\*

 \* <p>接口说明: Runnable，作为线程的接口。实现run()方法，可以实现线程的效果。</p>

 \* <p>Copyright 2003: Excellence Network Co.,LTD</p>

 \* @author fjy

 \* @version 1.0

 \* @since 2003-06-10

 \* @modified fjy 2003-06-13  修改了XXXX

\*/

**4   成员函数注释标准**

注释一个成员函数是为了函数更加可被理解，进而可维护和可扩展。

**4.1 成员函数的函数头**

每一个 Java 成员函数都应包含某种称之为“成员函数文档”的函数头。这些函数头在源代码的前面，用来记录所有重要的有助于理解函数的信息。 这些信息包含但不仅仅局限于以下内容：

1. **成员函数做什么以及它为什么做这个。**通过给一个成员函数加注释，让别人更加容易判断他们是否可以复用代码。注释出函数为什么做这个可以让其他人更容易将你的代码放到程序的上下文中去。也使其他人更容易判断是否应该对你的某一段代码加以修改（有可能他要做的修改与你最初为什么要写这一段代码是相互冲突的）。
2. **哪些参数必须传递给一个成员函数。**还必须说明，如果带参数，那么什么样的参数必须传给成员函数，以及成员函数将怎样使用它们。这个信息使其他程序员了解应将怎样的信息传递给一个成员函数。*javadoc* *@param* 标识便用于该目的。
3. **成员函数返回什么。**如果成员函数有返回值，则应注释出来，这样可以使其他程序员正确地使用返回值/对象。*javadoc* *@return* 标识便用于此目的。
4. **已知的问题。**成员函数中的任何突出的问题都应说明，以便让其他程序开发者了解该成员函数的弱点和难点。如果在一个类的多个成员函数中都存在着同样的问题，那么这个问题应该写在类的说明里。
5. **任何由某个成员函数抛出的异常。**应说明成员函数抛出的所有异常，以便使其他程序员明白他们的代码应该捕获些什么。*javadoc* *@exception* 标识便用于此目的。
6. 如何在适当情况下调用成员函数的例子。最简单的确定一段代码如何工作的方法是看一个例子。考虑包含一到两个如何调用成员函数的例子。
7. 可用的前提条件和后置条件。 前提条件是指一个成员函数可正确运行的限制条件；后置条件是指一个成员函数执行完以后的属性或声明 [[MEY88](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#MEY88#MEY88)]。前提条件和后置条件以多种方式说明了在编写成员函数过程中所做的假设 [[AMB98](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#AMB98#AMB98)]，精确定义了一个成员函数的应用范围。
8. 成员函数的开发/维护历史。通常要包含一个历史记录表，列出日期、修改的作者和修改概要。这样做的目的是让进行维护的程序员了解过去曾对一个成员函数所做的修改，是谁做了什么样的修改。

示例：

/\*\*

   \* 执行SELECT函数，可以分页。按照输入的页码、每页的行数，返回当前页的结果集

   \* 又多行记录：结果放入VECTOR，VECTOR里面的每一项代表一行记录。

   \*          VECTOR里面包含的String[]就是真正的结果

   \* @param pageNo 返回SELECT结果中的第几页数据

   \* @param pageCount 每页的最大行数

   \* @throws java.lang.Exception

   \*  modified by fjy 2003-07-03 增加了pageCount参数的检查

   \*/

public void ExecuteSelect(int pageNo,int pageCount)  throws Exception  {

}

**4.1.1内部注释**

除成员函数注释以外，在成员函数内部还需加上注释语句来说明你的工作。 目的是使成员函数更易理解、维护和增强。

内部注释应采用两种方式：C 语言风格的注释 (/\* 和 \*/) 和单行注释 (//)。正如上述所讨论的，应认真考虑给代码的业务逻辑采用一种风格的注释，给要注释掉的无用代码采用另外一种风格的注释。**对业务逻辑采用单行注释，**因为它可用于整行注释和行末注释。**采用 C 语言风格的注释语句去掉无用的代码，**因为这样仅用一个语句就可以容易地去掉几行代码。此外，因为 C 语言风格的注释语句很象文档注释符。它们之间的用法易混淆，这样会使代码的可理解性降低。所以，应尽量减少使用它们。

在函数内，需要注释说明的地方：

1. 控制结构。说明每个控制结构，例如比较语句和循环。你无须读完整个控制结构内的代码才判断它的功能，而仅需看看紧靠它之前的一到两行注释即可。
2. 代码做了些什么以及为什么这样做。通常你常能看懂一段代码做了什么，但对于那些不明显的代码，你很少能判断出它为什么要那样做。例如，看完一行代码，你很容易地就可以断定它是在定单总额上打了 5% 的折扣。这很容易。不容易的是为什么要打这个折扣。显然，肯定有一条商业法则说应打折扣，那么在代码中至少应该提到那条商业法则，这样才能使其他开发者理解你的代码为什么会是这样。
3. 局部变量。虽然我们在第 4 章将仔细讨论这一点，在一个成员函数内定义的每一个局部变量都应在它代码的所在行声明，并且应采用一个行内注释说明它的用法。
4. 难或复杂的代码。若发现不能或者没有时间重写代码，那么应将成员函数中的复杂代码详细地注释出来。一般性的经验法则是，如果代码并非显而易见的，则应说明。
5. 处理顺序。如果代码中有的语句必须在一个特定的顺序下执行，则应保证将这一点注释出来 [[AMB98](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#AMB98#AMB98)]。没有比下面更糟糕的事了：你对一段代码做一点简单的改动，却发现它不工作，于是花了几个小时查找问题，最后发现原来是搞错了代码的执行顺序。

**在闭括号后加上注释。**常常会发现你的控制结构内套了一个控制结构，而在这个控制结构内还套了一个控制结构。虽然应该尽量避免写出这样的代码，但有时你发现最好还是要这样写。问题是闭括号 } 应该属于哪一个控制结构这一点就变得混淆了。一个好消息是，有一些编辑器支持一种特性：当选用了一个开括号后，它会自动地使相应得闭括号高亮显示；一个坏消息是，并非所有的编辑器都支持这种属性。我发现通过将类似 **//end if**，**//end for**，**//end switch**，&这样的注释加在闭括号所在行的行后，可以使代码更易理解。

实例：

注释一段代码的作用

//sql语句是否为空

    if(strSQL == null){

        throw(new Exception ("查询语句为空！"));

    }

注释一段不使用的代码

/\*以下代码已经暂停使用  by fjy 2003-07-02

for (int i = 0, l = ve.size(); i < l; i++) {

      String[] sre = (String[]) (ve.elementAt(i));

      user = new User();

      user.setUserCode(sre[0]);

      user.setUserName(sre[1]);

      user.setDeptCode(sre[2]);

      user.setUserTitle(sre[3]);

      user.setUserOffi(sre[4]);

      user.setUserPhone(sre[5]);

      user.setUserAddr(sre[6]);

      alist.add(user);

}

\*/

**4.2 字段/属性 注释标准**

**4.2.1注释一个字段**

字段都应很好地加以注释，以便其他开发者理解它。要想有效地注释，以下的部分需要说明：

1. 字段的说明。需说明一个字段，才能使人了解如何使用它。
2. 注释出所有采用的不变量。字段中的不变量是指永远为“真”的条件。例如，字段 dayOfMonth 的不变量可能是它的值只能在 1 到 31 之间（显然，可以用基于某一年里的某个月份来限制这个字段值，使其变的更加复杂）。通过说明字段值的限制条件，有助于定义重要的业务规则，使代码更易理解。

  示例：

//最终执行的SQL 语句

  protected String     strSQL = null;

  //需要连接数据源的名称

  private   String     dsName;

**4.3 局部变量注释标准**

**4.3.1声明和注释局部变量**

在 Java 中声明和注释局部变量有几种约定。这些约定是：

1. 一行代码只声明一个局部变量。这与一行代码应只有一个语句相一致，并使得对每个变量采用一个行内注释成为可能。
2. 用一个行内注释语句说明局部变量。行内注释是一种紧接在同一行的命令代码后，用符号 // 标注出来的单行注释风格（它也叫“行末注释”）。应注释出一个局部变量用于做什么、在哪里适用、为什么要用等等，使代码易读。

如：

int count = 1;  //记录循环次数

int pangeSize = 5;  //页面的大小

**4.4 成员函数参数注释标准**

**注释参数**

成员函数的参数在采用 *javadoc* *@param* 标识的头文件中注释。应说明：

1. 参数用来做什么。需要注释出参数用来做什么，以便其他开发者了解使用参数的上下文。
2. 任何约束或 前提条件。 如果一个参数的值域不能被成员函数接收，则应让调用者知道。可能一个成员函数只接收正数，或者字符数小于五的字符串。
3. 示例。如果应传递什么样的参数不明显，那么应该在注释中给出一个或多个例子。

**5   Java编程格式规范**

         为了对应jbuilder工具的一些默认设置，我们定义大括号“{”一般放在一行语句的结尾。常见的格式示例如下：

**5.1 if-else-elseif**

如果是if…elseif…else的情况，超过一行的执行体语句要分行，如果其中有任何一个的执行体使用了花括号，就应该都使用花括号。

if ( a > b )

                   a = a + 1 ;

         else

                   a = a - 1 ;

         if ( ( a > b ) && ( a > c ) ){

                   b = b - 1 ;

                   c = c - 1 ;

         }else if ( a > b ){

                   b = b - 1 ;

         }else if ( a > c ){

                   c = c - 1 ;

         }else{

                   a = a + 1 ;

         }

**5.2 switch**

Switch语句应该都带有一个default，当出现了意料之外的情况时可以在default中抛出异常。

switch(n)

{

          case 0:

                   do3();

      break;

          case 1:

                   do1();

                   break;

          case 2:

                   do2();

                   break;

          case 3:

                   do3();

                   break;

          default:

                   dodefault();

                   break;

}

**5.3  while**

while语句也是循环性质，所以也要使用花括号

while (n < 6) {

         System.out.println("Current value of n is {0}", n);

         n++;

}

**5.4 do … while**

do语句因为使用的情况较少，最好是尽量使用while来代替do

do{

         System.out.println("Current value of n is {0}", n);

         n++;

}while (n < 6)

**5.5 for**

for语句因为具有循环的性质，应该使用花括号明确的标出语句块的范围

for (int i = 0; i < aa.length; i++) {

    System.out.println(aa[i]);

}

**5.6 try-catch**

try-catch是异常捕捉的语句，做为语言本身的规定，就必须使用花括号。

try {

}catch (Exception ex) {

          ex.printStackTrace();

}catch (ClassNotFoundException classnotfoundexception) {

          classnotfoundexception.printStackTrace();

}finally {

    Logger.log("ddd");

}

**6     Java错误和异常处理规范**

**6.1 Java程序错误和异常处理原则**

为了系统的稳定性，应该保证代码的健壮性。时时刻刻检查是否有异常产生，并按以下的原则处理。在底层的类中，如果没有显示的界面，应该在相关的log日志文件中记录错误；如果在有显示界面的层中，应该在记录log日志文件的同时，显示相关的出错信息。

在成员函数中应该检查参数的合法性，运行状态中参数的合法性，执行结果是否为null等。

不同模块可以定义自己的异常类,按照实际运行过程可能出现的

**6.2 在底层的类中**

在底层的类中出现的异常可以在log的同时，再往外抛出异常。如：

/\*\*

   \* 通过数据源名称取数据源

   \* @param dataSourceName

   \* @return

   \* @throws java.lang.Exception

   \*/

 public static Connection getConnection(String dataSourceName) throws Exception

 {

          DataSource ds = null;

          try{

                    //先取服务器的命名空间，再通过数据源的jndi名称取得数据源

                    Context initCtx = new InitialContext();

                    Context envCtx = (Context) initCtx.lookup("java:comp/env");

                    ds = (DataSource)envCtx.lookup(dataSourceName);

          } catch (Exception e) {

                      //使用log

                      LoggerFactory.getInstance().error("Error looking up the DataSource object: " + e);

                      //在底层先不处理异常

                      throw e;

          }

          return ds.getConnection();

  }

**6.3 在高层的类中**

在高层的类中，在页面中出现的异常，则应该引导到出错页面，并显示相关的错误信息。

Jsp中的处理：

LogWriterDemo<br>

<%

try{

//LogWriter

    LogWriter.logInfo("modue","source","working","user1",request,"memory");

}catch(Exception ee){

response.sendRedirect(/error.jsp);

}

%>

servlet中的处理：

try{

               performTask(request,response);

            }

            catch (Exception e){

                    //使用默认的exoa4j的实例打印错误信息

                    LoggerFactory.getInstance().error(this, e);

                    try{

                            request.setAttribute( "msg", e.getMessage() );

                            toNextPage( "/error.jsp", request, response );

                            return;

                    }catch ( Exception ex ){

                            //使用默认的exoa4j的实例打印错误信息

                            LoggerFactory.getInstance().error(this, ex);

                    }

            }

**7   Java编程一般约定：**

**7.1 将公共和保护接口最小化**

面向对象程序设计的基本点之一是最小化一个类的公共接口。这样做有几个理由：

1. 可学习性。要了解如何使用一个类，只需了解它的公共接口即可。公共接口越小，类越容易学习。
2. 减少耦合。当一个类的实例向另一个类的实例或者直接向这个类发送一条消息时，这两个类变得耦合起来。最小化公共接口意味着将耦合的可能降到最低。
3. 更大的灵活性。这直接与耦合相联系。一旦想改变一个公共接口的成员函数的实现方法，如你可能想修改成员函数的返回值，那么你很可能不得不修改所有调用了该成员函数的代码。公共接口越小，封装性就越大，代码的灵活性也越大。

尽力使公共接口最小化这一点明显地很值得你的努力，但通常不明显的是也应使被保护接口最小化。基本思想是，从一个子类的角度来看，它所有超类的被保护接口是公共的。任何在被保护接口内的成员函数可被一个子类调用。所以，出于与最小化公共接口同样的理由，应最小化类的被保护接口。

首先定义公共接口。大多数有经验的开发者在开始编写类的代码之前就先定义类的公共接口。 第一，如果你不知道一个类要完成怎样的服务/行为，你仍有一些设计工作要做。第二，这样做使这个类很快地初具雏形，以便其他有赖于该类的开发者在“真正的”类被开发出来以前至少可以用这个雏形开始工作。 第三，这种方法给你提供了一个初始框架，围绕着这个框架你构造类。

**7.2 编程一般约定**

有许多的有关 Java 代码可维护性和可改进性的重要约定和标准。99.9% 的时间里，面向他人，面向你的开发同事编程要比面向机器编程重要得多。使你的代码为别人所理解是最重要的。

|  |  |
| --- | --- |
| 约定目标 | 约定 |
| 存取成员函数 | 考虑对数据库中的字段使用滞后初始化  使用存取函数获得和修改所有字段  对常量采用存取函数  对于集合，加入成员函数来插入和删除项  一旦可能，将存取函数置为被保护类型，不是公共类型 |
| 字段 | 字段永远为私有类型  不要直接访问字段，应使用存取成员函数  不要使用静态常量字段（常量），应使用存取成员函数  不要隐藏名字  一定要初始化静态字段 |
| 类 | 最小化公共和保护接口  在开始写代码之前定义一个类的公共接口  按以下顺序声明一个类的字段和成员函数：   * 构造函数 * 公共成员函数 * 被保护成员函数 * 私有成员函数 * 私有字段 |
| 局部变量 | 不要隐藏名字  一行代码只声明一个局部变量  用一个行内注释说明局部变量  在使用局部变量之前声明它  仅将局部变量用于一件事 |
| 成员函数 | 给代码加上注释  给代码分段  使用空白，控制结构之前用一个空行，成员函数之前用两个空行  一个成员函数应能在 30 秒内让人理解  写短小单独的命令行  尽量限制成员函数的可见性  说明操作的顺序 |

**8   其他**

**8.1 编写清晰整洁的代码的原则**

这一部分讲述几个技巧，这些技巧有助于区分专业软件开发者和蹩脚代码编写者。这些技巧是：

* 给代码加上注释
* 给代码分段
* 使用空白
* 遵循 30 秒条规则
* 说明消息发送的顺序
* 写短小单独的命令行

**8.1.1给代码加上注释**

记住：如果你的代码不值得注释，那么它就不值得保留 [[NAG95](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#NAG95#NAG95)]。当正确地使用了本文提到的注释标准和方针，就可以大幅度地提高代码质量。

**8.1.2让代码分段/缩进**

一种提高代码可读性的方法是给代码分段，换句话说，就是在代码块内让代码缩进。所有在括号 **{** 和 **}** 之内的代码，构成一个块。基本思想是，块内的代码都应统一地缩进去一个单位。

Java 的约定似乎是开括号放在块的所有者所在行的后面，闭括号应缩进一级。在 [[LAF97](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#LAF97#LAF97)] 指出的很重要的一点是，你所在的机构应选取一个缩进风格并始终使用这种风格。采用与你的 Java 开发环境所生成的代码一样的缩进风格。

**在代码中使用空白。**在 Java 代码中加入几个空行，也叫空白，将代码分为一些小的、容易理解的部分，可以使它更加可读。[[VIS96](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#VIS96#VIS96)] 建议采用一个空行来分隔代码的逻辑组，例如控制结构，采用两个空行来分隔成员函数定义。没有空白的代码很难读，很难理解。

**8.1.3遵循 30 秒条法则**

其他的程序员应能在少于 30 秒钟的时间内完全理解你的成员函数，理解它做什么，为什么这样做，它是怎样做的。如果他们做不到，说明你的代码太难维护，应加以改进。30 秒钟，明明白白。 一个好的经验法则是：如果一个成员函数一个屏幕装不下，那么它就很可能太长了。

**8.1.4写短小单独的命令行**

每一行代码只做一件事情。在依赖于穿孔卡片的计算机发展的早期，想让一行代码完成尽量多的功能的想法是可以理解的。若想在一行里做多件事情，就会使代码难于理解。为什么要这样呢？我们应使代码尽量容易理解，从而更容易维护和改进。正如同一个成员函数应该并且只能做一件事一样，一行代码也只应做一件事情。

此外，应让代码在一个屏幕内可见 [[VIS96](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#VIS96#VIS96)]。 也不应向右滚动编辑窗口来读取一整行代码，包括含有行内注释语句的代码。

**8.1.5说明运行顺序**

提高代码可读性的一个相当简单的方法是使用圆括号 (parenthesis，又叫“round brackets”) 来说明 Java 代码运行的准确顺序 [[NAG95](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#NAG95#NAG95)]；[[AMB98](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#AMB98#AMB98)]。如果为了理解你的源码而必须了解编程语言的操作顺序，那么这说明源码中一定有什么重要的东西做的不对。 这大多是在 AND 或者 OR 其它几个比较关系处产生的逻辑比较上的问题。注意：如果你象前文所建议的那样，采用短小单独的命令行，那么就不会产生这个问题。

**8.2 为什么采用存取函数？**

“好的程序设计试图将程序部件与不必要、未计划或者不需的外部影响分隔开来。访问修饰语句（存取函数）给编程语言控制这一类的接触提供了一种清晰并可检验的方法。”

存取成员函数通过以下方法提高类的可维护性：

1. 更新字段。.每个字段只有几个单点要更新，这使得修改和检测都很容易。换句话说，字段已被封装。
2. 获得字段的值。你完全控制着字段应怎样被访问以及被谁访问。
3. 获取常量名和类名。在获取函数中封装常量值和类名，当这些值或名字改变时，只需更新获取函数内的值，而并非常量或者名字被使用处的每一行代码。
4. 初始化字段。采用滞后初始化 (lazy initialization) 保证字段总能被初始化，并且只在需要时才初始化。
5. 减少类与子类之间的耦合。 当子类通过它们相应的存取成员函数访问被继承的字段时，它可以不影响它的任何子类，而只修改超类字段的实现方式，这样有效地减少了超类与子类之间的耦合。存取函数减少了那种一旦超类被修改就会波及子类的“脆弱基类”的风险。
6. 将变化封装到字段中。如果一个或者多个字段的业务规则变化了，可以只潜在地修改存取函数，就同样可以提供规则变化之前的功能。这一点使你很容易响应新的业务规则。
7. 简化并行事件。[[LEA97](http://adionline.myrice.com/Java/java-ProgrammingModel.htm#LEA97#LEA97)] 指出，如果采用了基于字段值的 waits 语句，那么设置成员函数提供了一个位置可包含 *notifyAll*。这让转向并行解决方案更加容易。
8. 名字隐藏不再是一个大问题。虽然应该避免名字隐藏，即避免赋给局部变量一个与字段相同的名字，但是始终通过存取函数访问字段意味着可以给局部变量任何你想取的名字。不必担心字段名的隐藏，因为你无论如何都不会直接访问它们。

当不用存取函数时：唯一可能你不想用存取函数的时候是当执行时间最重要时。但是，这实际上很少见，因为应用程序耦合性的增加会平衡掉这样做之后赢得的好处。