Java 开发规范

Java Development Specification

版本 2.0

当前版本	2.0alpha
制定日期	2004-07-18
文档编号	Java_Dev_Spec
文档作者	韩卿(l_walker)
电子邮件	walker@skyinn.org

Skyinn Group(http://www.skyinn.org)

修订历史记录

日期	版本	说明	作者
2002-9-7	1.0	创建文件	韩卿
2002-9-8	1.0	修改、增加	韩卿
2002-9-9	1.0	完成	韩卿
2003-1-19	1.0	更新	韩卿
2003-05-28	1.1	修改并完成	韩卿
2004-7-3	2.0	修改版式	韩卿
2004-7-11	2. 0	更新文档	韩卿
2004-7-18	2.0 alpha	更新并发布	韩卿

目录

第 1 章 绪论 5

1.1 目的 5
1.2 范围 5
1.3 版权声明 5
1.4 参考资料 5
1.5 概述 5

第 2 章 代码组织与风格 7

2.1 基本原则 7
2.2 缩进 7
2.3 长度 7
2.4 行宽 7
2.5 间隔 7
2.6 对齐 7
2.7 括号 8

第 3 章 注释 9

3.1 基本原则 9
3.2 JAVADOC 9
3.3 文件与包注释 9
3.4 类、接口注释 10
3.5 方法注释 10
3.6 其他注释 11
3.7 注释参考表 11

第 4 章 命名 13

4.1 基本原则 13
4.2 文件、包 13
4.3 类、接口 13
4.4 字段 14
4.5 方法 14
4.6 异常 15
4.7 命名约定表 15

第 5 章 声明 17

5.1 基本原则 17
5.2 包 17
5.3 类、接口 17
5.4 方法 17
5.5 字段 18
5.6 示例 18

第 6 章 类与接口 20

6.1 基本原则 20
6.2 抽象类与接口 20
6.3 继承与组合 20

6.4 构造函数和静态工厂方法 20
6.5 TOSTRING(),EQUALS(),HASHCODE()... 20
6.6 SINGLETON CLASS 22

第 7 章 方法 **23**

7.1 基本原则 23
7.2 参数和返回值 23

第 8 章 表达式与语句 **24**

8.1 基本原则 24
8.2 控制语句 24
8.3 循环语句 25

第 9 章 错误与异常 **26**

9.1 基本原则 26
9.2 已检查异常与运行时异常 26
9.3 异常的捕捉与处理 26

第 10 章 测试与BUG跟踪 **27**

10.1 基本原则 27
10.2 测试驱动开发 27
10.3 JUNIT单元测试 27
10.4 自动测试与持续集成 27
10.5 BUG跟踪和缺陷处理 27

第 11 章 性能与安全 **28**

11.1 基本原则 28
11.2 STRING与STRINGBUGFFER 28
11.3 集合 28
11.4 对象 28
11.5 同步 28
11.6 FINAL 28
11.7 垃圾收集和资源释放 29

第 12 章 其他 **30**

12.1 目录结构 30
12.2 CVS注释与标记 31
12.3 31

第 13 章 附录 **32**

13.1 CVS标识符 32
13.2 注释模板 32
13.3 常用缩写简表 33
13.4 版权声明模板 33
13.5 示例代码 34

Java 开发规范

第1章 绪论

1.1 目的

本规范的目的是使本组织能以标准的、规范的方式设计和编码。通过建立编码规范，以使每个开发人员养成良好的编码风格和习惯；并以此形成开发小组编码约定，提高程序的可靠性、可读性、可修改性、可维护性和一致性等，增进团队间的交流，并保证软件产品的质量。

1.2 范围

本规范适用于“Skyinn Group”及其下所有软件项目、产品等的设计、开发以及维护、升级等。

本规范使用于“Skyinn Group”的所有软件开发人员，在整个软件开发过程中必须遵循此规范。

1.3 版权声明

1.4 参考资料

《Java 编程指南》见 RUP（Rational Unified Process）中文版。
《Java 技术手册》(Java in a Nutshell)
《Sun Java 语言编码规范》(Java Code Conventions)
《Effictive Java》
《Java Pitfalls》
《Java Rules》

1.5 概述

对于代码，首要要求是它必须正确，能够按照设计预定功能去运行；第二是要求代码必须清晰易懂，使自己和其他的程序员能够很容易地理解代码所执行的功能等。然而，在实际开发中，每个程序员所写的代码却经常自成一套，很少统一，导致理解困难，影响团队的开发效率及系统的质量等。因此，一份完整并被严格执行的开发规范是非常必须的，特别是对软件公司的开发团队而言。此规范参考自业界标准编程规范并结合本人多年编程经验、习惯等而制定，在本人工作过的公司中都曾参考本文档而形成内部开发规范并执行。现在将本文档共享之，希望能对各位有所帮助，并做引玉之砖，希望各位朋友将自己的经验等增补进去，对我们所热爱的软件业有所裨益。

最根本的原则：

代码虽然是给机器运行的，但却是给人读的！

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。当出现该情况时，应当及时收集并提交，以便对本规范进行修改。

您可以在以下地址中找到本文档的最新版本：

http://www.skyinn.org/wiki/Wiki.jsp?page=JavaDevSpec

本文档尚未完善，我将不断更新之，如果您有任何问题、建议或意见等请联系我：

walker@skyinn.org

第2章　　代码组织与风格

2.1　基本原则

代码的组织和风格的基本原则是：便于自己的开发，易于与他人的交流。

因个人习惯和编辑器等可以设置和形成自己的风格，但必须前后一致，并符合本规范的基本要求和原则。

本章所涉及到的内容一般都可在 Java 集成编辑环境中进行相应设置，也可由 Ant 等调用 checkstyle 等来进行自动规整。

2.2　缩进

子功能块当在其父功能块后缩进。

当功能块过多而导致缩进过深时当将子功能块提取出来做为子函数。

代码中以 TAB（4 个字符）缩进，在编辑器中请将 TAB 设置为以空格替代，否则在不同编辑器或设置下会导致 TAB 长度不等而影响整个程序代码的格式。例如：

Table1. 缩进示例

```
public void methodName(){
    if(some condition){
        for(...){
            //some sentences
        }//end for
    }//end if
}
```

2.3　长度

为便于阅读和理解，单个函数的有效代码长度当尽量控制在 100 行以内（不包括注释行），当一个功能模块过大时往往造成阅读困难，因此当使用子函数等将相应功能抽取出来，这也有利于提高代码的重用度。

单个类也不宜过大，当出现此类情况时当将相应功能的代码重构到其他类中，通过组合等方式来调用，建议单个类的长度包括注释行不超过 1500 行。

尽量避免使用大类和长方法。

2.4　行宽

页宽应该设置为 80 字符。一般不要超过这个宽度，这会导致在某些机器中无法以一屏来完整显示，但这一设置也可以灵活调整。在任何情况下，超长的语句应该在一个逗号后或一个操作符前折行。一条语句折行后，应该比原来的语句再缩进一个 TAB 或 4 个空格，以便于阅读。

2.5　间隔

类、方法及功能块间等应以空行相隔，以增加可读性，但不得有无规则的大片空行。

操作符两端应当各空一个字符以增加可读性。

相应独立的功能模块之间可使用注释行间隔，并标明相应内容，具体参看附录的代码示例

2.6　对齐

关系密切的行应对齐，对齐包括类型、修饰、名称、参数等各部分对齐。

连续赋值时当对齐操作符。

当方法参数过多时当在每个参数后（逗号后）换行并对齐。
当控制或循环中的条件比较长时当换行（操作符前）、对齐并注释各条件。
变量定义最好通过添加空格形成对齐，同一类型的变量应放在一起。如下例所示：
Table2. 对齐示例

```
//变量对齐-----------------------------------------------
int        count       = 100;
int        length      = 0;
String     strUserName = null;
Integer[]  porductCode = new Integer(2); //产品编码数组
//参数对齐-----------------------------------------------
public Connection getConnection(String url,
                                String userName,
                                String password)
    throws SQLException,IOException{
}
//换行对齐-----------------------------------------------
public final static String SQL_SELECT_PRODUCT = "SELECT * "
    + " FROM TProduct WHERE Prod_ID = "
    + prodID;
//条件对齐-----------------------------------------------
if( Condition1      //当条件一
    && Condition2   //并且条件二
    || Condition3){ //或者条件三
}
for(int i = 0;
    i < productCount.length;  //循环终止条件
    i++){
}
```

2.7 括号

{} 中的语句应该单独作为一行，左括号"{"当紧跟其语句后，右括号"}"永远单独作为一行且与其匹配行对齐，并尽量在其后说明其匹配的功能模块。

较长的方法以及类、接口等的右括号后应使用//end ...等标识其结束。如:

类的结束符：}//EOC ClassName，
方法结束符：}//end methodName()，
功能块结束：}//end if...userName is null?
循环快结束：}//end for...every user in userList

不要在程序中出现不必要的括号，但有时为了增加可读性和便于理解，当用括号限定相应项。

左括号是否换行等随个人习惯而定，若换行则当与其前导语句首字符对齐。

第3章 注释

3.1 基本原则

3.1.1 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其他开发人员等理解。

3.1.2 如果你的程序不值得注释，那么它很可能也不值得运行。

3.1.3 避免使用装饰性内容。

3.1.4 保持注释的简洁。

3.1.5 注释信息不仅要包括代码的功能，还应给出原因。

3.1.6 不要为注释而注释。

3.1.7 除变量定义等较短语句的注释可用行尾注释外，其他注释当避免使用行尾注释。

3.2 JavaDoc

对类、方法、变量等的注释需要符合 JavaDoc 规范，对每个类、方法都应详细说明其功能、条件、参数等，并使用良好的 HTML 标记格式化注释，以使生成的 JavaDoc 易阅读和理解。

类注释中当包含版本和作者信息，使用 CVS 标记自动跟踪版本变化和修改记录，具体内容参见《CVS 使用手册》及下面几节的相应内容等，CVS 标识符请参加附录中的《CVS 标识符》。

3.3 文件与包注释

在每个文件、包的头部都应该包含该文件的功能、作用、作者、版权以及创建、修改记录等。并在其中使用 CVS 标记自动跟踪版本变化及修改记录等信息。注意是/**/注释而不是/***/JavaDoc 注释。

文件注释模板见附件《注释模板》中的文件注释部分。

版权声明部分请参见附件《版权声明》并注意更新到最新版本。

Table3 文件注释示例：

```
/* ============================================================
 * $Id: User.java,v 1.1 2002/09/07 14:36:23 l_walker Exp $
 * Created: [2003-3-23 20:18:53] by l_walker
 * ============================================================
 *
 * ProjectName
 *
 * Description
 *
 * ============================================================
 *
 * Copyright Information.
 *
 * ============================================================*/
```

每个包当有包注释，在源码及 JavaDoc 相应包路径下建立 package.html 以描述包的功能、作用等。

3.4 类、接口注释

在类、接口定义之前当对其进行注释，包括类、接口的目的、作用、功能、继承于何种父类，实现的接口、实现的算法、使用方法、示例程序等，在作者和版本域中使用 CVS 标记自动跟踪版本变化等，具体参看附件《注释模板》中相关部分。

Table4 类注释示例

```
/**
 * <p>字符串实用类。</p>
 *
 * 定义字符串操作时所需要用到的方法，如转换中文、HTML 标记处理等。
 *
 * @author $Author: l_walker$
 * @version $Revision: 1.2 $ $Date: 2003/05/15 02:10:27 $
 */
public class StringUtil {
  …
}
```

3.5 方法注释

依据标准 JavaDoc 规范对方法进行注释，以明确该方法功能、作用、各参数含义以及返回值等。复杂的算法用/**/在方法内注解出。

参数注释时当注明其取值范围等

返回值当注释出失败、错误、异常时的返回情况。

异常当注释出什么情况、什么时候、什么条件下会引发什么样的异常

Table5 方法注释示例

```
/**
  * 执行查询。
  *
  * 该方法调用 Statement 的 executeQuery(sql)方法并返回 ResultSet
  *结果集。
  *
  * @param sql 标准的 SQL 语句
  * @return ResultSet 结果集，若查询失败则返回 null
  * @throws SQLException 当查询数据库时可能引发此异常
  */
 public ResultSet executeQuery(String sql) throws SQLException
 {
     //Statement 和 SQL 语句都不能为空
     if(null != stmt && !StringUtil.isEmpty(sql)){
         //返回查询执行结果
         return stmt.executeQuery(sql);
     }
     return null;
 }//end executeQuery()
```

3.6 其他注释

应对重要的变量加以注释，以说明其含义等。

应对不易理解的分支条件表达式加注释。不易理解的循环，应说明出口条件。过长的方法实现，应将其语句按实现的功能分段加以概括性说明。

对于异常处理当注明正常情况及异常情况或者条件，并说明当异常发生时程序当如何处理。

3.7 注释参考表

Table6 注释参考表

项目	注释内容
参数	参数类型 参数用来做什么 约束或前提条件 示例
字段/属性	字段描述 注释所有使用的不变量 示例 并行事件 可见性决策
类	类的目的 已知的问题 类的开发/维护历史、版本 注释出采用的不变量 并行策略
编译单元 （文件）	每一个类/类内定义的接口，含简单的说明 文件名和/或标识信息 修改/维护记录 版权信息
获取成员函数	若可能，说明为什么使用滞后初始化
接口	目的 它应如何被使用以及如何不被使用
局部变量	用处/目的
成员函数注释	成员函数做什么以及它为什么做这个 哪些参数必须传递给一个成员函数 成员函数返回什么 已知的问题 任何由某个成员函数抛出的异常 可见性决策 成员函数是如何改变对象的 包含任何修改代码的历史 如何在适当情况下调用成员函数的例子 适用的前提条件和 后置条件
成员函数内部注释	控制结构 代码做了些什么以及为什么这样做 局部变量

	难或复杂的代码 处理顺序
包	包的功能和用途

第4章 命名

4.1 基本原则

规范的命名能使程序更易阅读，从而更易于理解。它们也可以提供一些标识功能方面的信息，有助于更好的理解代码和应用。

4.1.1 使用可以准确说明变量/字段/类/接口/包等的完整的英文描述符。例如，采用类似 firstName，listAllUsers 或 CorporateCustomer 这样的名字，严禁使用汉语拼音及不相关单词命名，虽然 Java 支持 Unicode 命名，但本规范规定对包、类、接口、方法、变量、字段等不得使用汉字等进行命名。

4.1.2 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。

4.1.3 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。包名全部小写。

4.1.4 尽量少用缩写，但如果一定要使用，当使用公共缩写和习惯缩写等，如实现（implement）可缩写成 impl，经理（manager）可缩写成 mgr 等，具体参看附录之《常用缩写简表》，严禁滥用缩写。

4.1.5 避免使用长名字（最好不超过 25 个字母）。

4.1.6 避免使用相似或者仅在大小写上有区别的名字。

4.1.7 避免使用数字，但可用 2 代替 to，用 4 代替 for 等，如：go2Jsp。

4.2 文件、包

4.2.1 文件名当与其类严格相同，所有单词首字母大写。

4.2.2 包名一般以项目或模块名命名，少用缩写和长名，一律小写。

4.2.3 基本包：org.skyinn，所有包、文件都从属于此包。

4.2.4 包名按如下规则组成：

[基本包].[项目名].[模块名].[子模块名]...

如：

org.skyinn.quasar

org.skyinn.skyhome.dao.hibernate

4.2.5 不得将类直接定义在基本包下，所有项目中的类、接口等都当定义在各自的项目和模块包中。

4.3 类、接口

所有单词首字母大写。使用能确切反应该类、接口含义、功能等的词。一般采用名词。接口可带 I 前缀或 able、ible、er 等后缀。

4.4 字段

4.4.1 常量

采用完整的英文大写单词，在词与词之间用下划线连接，如：DEFAULT_VALUE

4.4.2 变量和参数

对不易清楚识别出该变量类型的变量应使用类型缩写作其前缀，如字符串使用 strXXX,boolean 使用 isXXX，hasXXX 等等。除第一各个单词外其余单词首字母大写。

对私有实例变量可使用_前缀，但在其存取方法中则应该将其前缀去掉。

4.4.3 组件/部件

应采用完整的英文描述符命名组件（接口部件），遵循匈牙利命名法则

如：btnOK，lblName。

4.4.4 集合

一个集合，例如数组和矢量，应采用复数命名来表示队列中存放的对象类型。命名应采用完整的英文描述符，名字中所有非开头的单词的第一个字母应大写，适当使用集合缩写前缀。如：

```
Vector vProducts = new Vector(); //产品向量
Array aryUsers = new Array();    //用户列表
```

4.4.5 神秘的数

我们在程序里经常会用到一些量，它是有特定的含义的。例如，现在我们写一个薪金统计程序，公司员工有 50 人，我们在程序里就会用 50 这个数去进行各种各样的运算。在这里，50 就是"神秘的数"。当别的程序员在程序里看到 50 这个数，将很难知道它的含义，造成理解上的困难。

在程序里出现"神秘的数"会降低程序的可读性、可维护性和可扩展性，故规定不得出现此类"神秘的数"。避免的方法是把神秘的数定义为一个常量。注意这个常量的命名应该能表达该数的意义，并且应该全部大写，以与对应于变量的标识符区别开来。例如上面 50 这个数，我们可以定义为一个名为 NUM_OF_EMPLOYEES 的常量来代替。这样，别的程序员在读程序的时候就可以容易理解了。

4.4.6 其他

命名时应使用复数来表示它们代表多值。如：orderItems。

4.5 方法

方法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。

方法名称的第一个单词常常采用一个有强烈动作色彩的动词。

取值类使用 get 前缀，设值类使用 set 前缀，判断类使用 is(has)前缀。

例： getName()
setSarry()
isLogon()

方法参数建议顺序：(被操作者，操作内容，操作标志，其他…)

```
例：public void replace(String sourceStr,  //源字串
                        String oldStr,     //被替换字串
                        String newStr){    //替换为字串
```

```
}
```

4.6 异常

异常类名由表示该异常类型的单词和 Exception 组成，如 ActionException。

异常实例一般使用 e、ex 等，在多个异常时使用该异常名或简写加 E，Ex 等组成，如：

```
SQLEx
ActionEx
```

4.7 命名约定表

Table7 命名约定表

操作项	命名约定	示例
参数	使用传递值/对象的完整的英文描述符。	userID
字段/属性	字段采用完整的英文描述，第一个字母小写，任何中间单词的首字母大写。	firstName
布尔型的获取成员函数	所有的布尔型获取函数必须用单词 is（has）做前缀。	isString() hasMoney()
类	采用完整的英文描述符，所有单词的第一个字母大写。	Customer
编译单元文件	使用类或接口的名字，或者如果文件中除了主类之外还有多个类时，加上前缀 java 来说明它是一个源码文件。	Customer.java
部件/组件	使用完整的英文描述来说明组件的用途，将组件类型使用匈牙利命名法则作其前缀。	btnOK,cboTypeList
构造函数	使用类名	Customer()
析构函数	Java 没有析构函数，但一个对象在垃圾收集时，调用成员函数 finalize() 。	finalize()
异常类名	由表示该异常类型等的单词和 Exception 组成	SQLException ActionException
异常实例名	通常采用字母 e 、ex 表示异常。多个异常时使用异常名或其简写加 E、Ex 等构成	e SQLEx
静态常量字段（常量）	全部采用大写字母，单词之间用下划线分隔。采用静态常量获取成员函数。	DEFAULT_NAME
获取成员函数	被访问字段名的前面加上前缀 get。	getUserName()
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。使用 I 前缀，其后使用 able,.	IRunnable IPrompter

	ible 或者 er 等后缀，但这不是必需的。	
局部变量	采用完整的英文描述符，第一个字母小写，但不要隐藏已有字段。例如，如果有一个字段叫 firstName，不要让一个局部变量叫 firstName。	strName,totalMoney
循环计数器	通常采用字母 i，j，k 或者 counter，index	i,j,k,count,index
包	采用完整的英文描述符，所有单词都小写，多个单词以下划线相连。所有包都属于 org.skyinn	org.skyinn.quasar org.skyinn.skyhome.dao
成员函数	采用完整的英文描述说明成员函数功能，第一个单词尽可能采用一个生动的动词，除第一个单词外，每个单词第一个字母小写。	openFile() addUser()
设置成员函数	被访问字段名的前面加上前缀 set。	setName() setPower()

第5章 声明

5.1 基本原则

声明的基本原则是遵守 Java 语言规范，并遵从习惯用法。

5.2 包

在导入包时当完全限制代码所使用的类的名字,尽量少用通配符的方式，但导入一些通用包，或用到一个包下大部分类时，则可是使用通配符方式，如：

```
import org.skyinn.quasar.services.Service;
import java.util.*;
```

同一包中的类导入时当声明在一起，可由编辑器自动完成此功能。

重要的包当添加注释。

5.3 类、接口

类、接口定义语法规范：

```
[可见性][('abstract'|'final')] [Class|Interface] class_name
     [('extends'|'implements')][父类或接口名]{
}
```

如：

```
public class LoginAction extends BaseAction implemnets ActionListener {
}
```

5.4 方法

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)。

方法定义语法规范：

```
[可见性][('abstract'|'final')] ['synchronized'][返回值类型] method_name(参数列表)[('throws')][异常列表]{
    // 功能模块
}
```

如：

```
public List listAllUsers() throws DAOException{
}
```

若有 toString(),equals(),hashCode(),colone()等重载自 Object 的方法，应将其放在类的最后。

声明顺序：

构造方法
静态公共方法
静态私有方法
受保护方法

私有方法
继承自 Object 的方法

5.5 字段

字段定义语法规范：

```
[('public'|'private'|'protected')]
    [('final'|'volatile')]['static']['transient']
        data_type field_name [ '=' expression] ';'
```

若没有足够理由，不要把实例或类变量声明为公有。公共和保护的可见性应当尽量避免，所有的字段都建议置为私有，由获取和设置成员函数（Getter、Setter）访问。

不允许“隐藏”字段，即给局部变量所取的名字，不可与另一个更大范围内定义的字段的名字相同（或相似）。例如，如果把一个字段叫做 firstName ，就不要再生成一个局部变量叫做 firstName，或者任何易混肴的名字，如 fistName。

数组声明时当将"[]"跟再类型后，而不时字段名后，如：

```
Integer[] ai   = new Integer (2);   //一个整数对象数组，用于...
Integer   aj[] = new Integer (3);   //不要用这种数组声明方式
```

一行代码只声明一个变量，仅将一个变量用于一件事。

声明顺序：

常量
类变量
实例变量
    公有字段
    受保护字段
    私有字段

可以将私有变量声明在类或接口的最后。

注意受保护变量、私有变量、“包”变量间的区别。

5.6 示例

Table8

```
//常量-------------------------------------------------
public final static double PI = 3.141592653589793;

// ------------------------------------------------类变量
protected static String key = "Love";

// ------------------------------------------------实例变量
//共有字段---------------------------------------------
public String userName = "Tom";
//受保护字段-------------------------------------------
protected float price = 0.0;
//友元字段---------------------------------------------
Vector vPorducts = null;
//私有字段---------------------------------------------
private int count;
```

```
//构造方法-----------------------------------------------
public Constructor(){
}

//公共方法-----------------------------------------------
public String getUserName(){
}

//友元方法-----------------------------------------------
void createProduct(){
}

//受保护方法---------------------------------------------
protected void convert(){
}

//私有方法-----------------------------------------------
private void init(){
}

//重载 Object 方法---------------------------------------
public String toString(){
}

//main 方法----------------------------------------------
public static void main(String[] args){
}
```

第6章　　类与接口

6.1 基本原则

类的划分粒度，不可太大，造成过于庞大的单个类，也不可太细，从而使类的继承太深。一般而言，一个类只做一件事；另一个原则是根据每个类的职责进行划分，比如用 User 来存放用户信息，而用 UserDAO 来对用户信息进行数据访问操作（比如存取数据库），用 UserBroker 来封装用户信息的业务操作等等。

多使用设计模式，随时重构。

多个类中使用相同方法时将其方法提到一个接口中或使用抽象类，尽量提高重用度。

将不希望再被继承的类声明成 final，例如某些实用类，但不要滥用 final，否则会对系统的可扩展性造成影响。

将不希望被实例化的类的缺省构造方法声明成 private。

6.2 抽象类与接口

一般而言：接口定义行为，而抽象类定义属性和公有行为，注意两者间的取舍，在设计中，可由接口定义公用的行为，由一个抽象类来实现其部分或全部方法，以给子类提供统一的行为定义，可参考 Java 集合等实现。

多使用接口，尽量做到面向接口的设计，以提高系统的可扩展性。

6.3 继承与组合

尽量使用组合来代替继承，一则可以使类的层次不至于过深，而且会使类与类，包与包之间的耦合度更小，更具可扩展性。

6.4 构造函数和静态工厂方法

当需要使用多个构造函数创建类时，建议使用静态工厂方法替代这些构造方法(参考《Effictive Java》 Item1)，例如：

```
public class User{
    public User(){
        super();
        //do somethings to create user instance
    }

    public static User getInstance(String name,String password){
        User u = new User();
        u.setName(name);
        u.setPassword(password);
        return u;
    }
```

参看 String，Boolean 的实现等：String.valueOf(Long l),Boolean.valueOf(String)...

6.5 toString(),equals(),hashCode()...

每个类都应该重载 toString()方法，以使该类能输出必要和有用的信息等。

```
/**
 * @see java.lang.Object#toString()
 */
public String toString() {
    final StringBuffer sb = new StringBuffer("Actor:[");
    sb.append("ID = ").append(_id)
            .append(",Name = ").append(_name)
            .append(']');
    return sb.toString();
}
```

若一个类需要重载 equals()方法，则必须同时重载 hashCode()方法实现方式参考《Effictive Java》Item7,Item8

```
/**
 * @see java.lang.Object#equals(java.lang.Object)
 */
public boolean equals (Object obj) {
    //空值
    if( null == obj){
        return false;
    }
    //引用相等
    if (obj == this) {
        return true;
    }

    //判断是否为当前类实例
    if (!(obj instanceof Actor)) {
        return false;
    }
    //若 ID 相等则认为该对象相等
    return this._id == ((Actor) obj).getId ();
}

/**
 * @see java.lang.Object#hashCode()
 */
public int hashCode () {
    int result = 17;//init result
    //String 对象 hashCode
    result = (37 * result) + _name.hashCode ();
    //数值
    result = (37 * result) + (int) (_id ^ (_id >>> 32));
    //String 对象 hashCode
    result = (37 * result) + _description.hashCode ();
```

```
    return result;
}
```

这些方法的重载实现也可参考如下实现（使用 Jakarta commons-lang）：

```
```

6.6 Singleton Class

单例类使用如下方式声明，并将其缺省构造方法声明成 private：

```
public class Singleton{
    private static Singleton instance = new Singleton();
    // 私有缺省构造方法，避免被其他类实例化
    private Singleton(){
        //do something
    }

    public static Singleton getInstance(){
          if(null == instance){
              instance = new Singleton;
          }
          return instance;
    }
}//EOC Singleton
```

单例类若需要实现序列化，则必须提供 readResolve()方法，以使反序列化出来的类仍然是唯一的实例，参加《Effictive Java》 Item57。

第7章　　方法

7.1　基本原则

一个方法只完成一项功能，在定义系统的公用接口方法外的方法应尽可能的缩小其可见性。

避免用一个类是实例去访问其静态变量和方法。

避免在一个较长的方法里提供多个出口：

```
//不要使用这钟方式，当处理程序段很长时将很难找到出口点
if(condition){
    return A;
}else{
    return B;
}

//建议使用如下方式
String result = null;
if(condition){
    result = A;
}else{
    result = B;
}
return result;
```

7.2　参数和返回值

避免过多的参数列表，尽量控制在 5 个以内，若需要传递多个参数时，当使用一个容纳这些参数的对象进行传递，以提高程序的可读性和可扩展性。

参数类型和返回值尽量接口化，以屏蔽具体的实现细节，提高系统的可扩展性，例如：

```
public void joinGroup(List userList){}
public List listAllUsers(){}
```

第8章　　表达式与语句

8.1　基本原则

表达式和语句当清晰、简洁，易于阅读和理解，避免使用晦涩难懂的语句。

每行至多包含一条执行语句，过长当换行。

避免在构造方法中执行大量耗时的初始化工作，应当将这中工作延迟到被使用时再创建相应资源，如果不可避免，则当使用对象池和 Cache 等技术提高系统性能。

避免在一个语句中给多个变量赋相同的值。它很难读懂。

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。

尽量在声明局部变量的同时初始化。唯一不这么做的理由是变量的初始值依赖于某些先前发生的计算。

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。

不要为了表现编程技巧而过分使用技巧，简单就好。

8.2　控制语句

判断中如有常量，则应将常量置与判断式的右侧。如：

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

尽量不使用三目条件判断。

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
```

当有多个 else 分句时当分别注明其条件，注意缩进并对齐，如：

```
//先判断 i 是否等于 1
if (i == 1){//if_1
    //.....
}//然后判断 i == 2
else if (i == 2){
    //i == 2 说明。。。。。。
    j = i;
}//如果都不是(i > 2 || i < 1)
else{
    //说明出错了
    //....
}//end if_1
```

过多的 else 分句请将其转成 switch 语句或使用子函数。

每当一个 case 顺着往下执行时(因为没有 break 语句)，通常应在 break 语句的位置添加注释。如：

```
switch (condition) {
  case ABC:
      //statements;
      //继续下一个 CASE
  case DEF:
      //statements;
      break;
  case XYZ:
      //statements;
      break;
  default:
      //statements;
      break;
}//end switch
```

8.3 循环语句

循环中必须有终止循环的条件或语句，避免死循环。

当在 for 语句的初始化或更新子句中使用逗号时，避免因使用三个以上变量，而导致复杂度提高。若需要，可以在 for 循环之前(为初始化子句)或 for 循环末尾(为更新子句)使用单独的语句。

因为循环条件在每次循环中多会执行一次，故尽量避免在其中调用耗时或费资源的操作，比较一下两种循环的差异：

```
//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){
}
```

第9章　　错误与异常

9.1　基本原则

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。

确保状态码有一个正确值。

在本地进行安全性检查，而不是让用户去做。

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。

9.2　已检查异常与运行时异常

已检查异常必须捕捉并做相应处理，不能将已检查异常抛到系统之外去处理。

对可预见的运行时异常当进行捕捉并处理，比如空指针等。通常，对空指针的判断不是使用捕捉 NullPointException 的方式，而是在调用该对象之前使用判断语句进行直接判断，如：

```
//若不对 list 是否为 null 进行检查，则在其为 null 时会抛出空指针异常
if(null != list && 0 < list.size()){
    for(int i = 0; i < list.size(); i++){
    }
}
```

建议使用运行时异常(RuntimeException)代替已检查异常（CheckedException），请参考网络资源以对此两种异常做更深入理解。

9.3　异常的捕捉与处理

捕捉异常是为了处理它，不要捕捉了却什么都不处理而抛弃之，最低限度当向控制台输出当前异常，如果你不想处理它，请将该异常抛给它的调用者，建议对每个捕捉到的异常都调用 printStackTrace()输出异常信息，避免因异常的湮没。

多个异常应分别捕捉并处理，避免使用一个单一的 catch 来处理。如：

```
try {
    //do something
}catch(IllegalStateException IllEx){
    IllEx.printStackTrace();
    //deal with IllEx
}catch(SQLException SQLEx){
    SQLEx.printStackTrace();
    throw SQLEx;  //抛给调用者处理
}finally{
    //释放资源
}
```

第10章　测试与 Bug 跟踪

10.1 基本原则

测试不通过的代码不得提交到 CVS 库或者发布。

不得隐瞒、忽略、绕过任何 Bug，有 Bug 不一定是你的错，但有了 Bug 不作为就是你的不对了。

多做测试，测试要完全，尽量将各种可能情况都测试通过，尽量将可能的 Bug 在开发中捕捉并处理掉。

测试要保证可再测试性。

测试应当对数据库等资源不留或少留痕迹，例如，当测试添加一个用户时，在其成功后当及时从数据库中删除该记录，以避免脏数据的产生（由此衍生的一个经验是将添加、获取、删除一起测试）。

对关键功能必须测试并通过，对辅助功能及非常简单之功能可不做测试。

10.2 测试驱动开发

测试驱动开发可很好的避免 Bug 的发生，并提升程序的质量，有助于提高个人的编程水平等，因此在开发中当逐步转向有测试驱动的开发，先写测试，再写代码。

具体请参考《测试驱动开发》。

10.3 Junit 单元测试

在 Java 应用中，单元测试使用 Junit 及其衍生工具。

在 TestCase 的 setUp()中初始化应用，在 tearDown()中释放资源。可由一个基础 TestCase 完成这些任务，其他 TestCase 继承之。

10.4 自动测试与持续集成

测试应当由系统自动完成并向相应人员发送测试报告。

由持续集成工具来完成测试的自动化。

10.5 Bug 跟踪和缺陷处理

当系统出现 Bug 时当由该 Bug 的负责人（代码负责人）尽快修改之。

Bug 的处理根据其优先级高低和级别高低先后处理。

Bug 级别和优先级别参见《测试手册》。

禁止隐瞒 Bug。

第11章　性能与安全

11.1 基本原则

性能的提升并不是一蹴而就的，而是由良好的编程积累的，虽然任何良好的习惯和经验所提升的性能都十分有限，甚至微乎其微，但良好的系统性能却是由这些习惯等积累而成，不积细流，无以成江海！

11.2 String 与 StringBugffer

不要使用如下 String 初始化方法：

```
String str = new String("abcdef");
```

这将产生两个对象，应当直接赋值：

```
String str = "abcdef";
```

在处理可变 String 的时候要尽量使用 StringBuffer 类，StringBuffer 类是构成 String 类的基础。String 类将 StringBuffer 类封装了起来，（以花费更多时间为代价）为开发人员提供了一个安全的接口。当我们在构造字符串的时候，我们应该用 StringBuffer 来实现大部分的工作，当工作完成后将 StringBuffer 对象再转换为需要的 String 对象。比如：如果有一个字符串必须不断地在其后添加许多字符来完成构造，那么我们应该使用 StringBuffer 对象和她的 append() 方法。如果我们用 String 对象代替 StringBuffer 对象的话，将会花费许多不必要的创建和释放对象的 CPU 时间。

11.3 集合

避免使用 Vector 和 HashTable 等旧的集合实现，这些实现的存在仅是为了与旧的系统兼容，而且由于这些实现是同步的，故而在大量操作时会带来不必要的性能损失。在新的系统设计中不当出现这些实现，使用 ArrayList 代替 Vector，使用 HashMap 代替 HashTable。

若却是需要使用同步集合类，当使用如下方式获得同步集合实例：

```
Map map = Collections.synchronizedMap(new HashMap());
```

由于数组、ArrayList 与 Vector 之间的性能差异巨大（具体参见《Java fitball》），故在能使用数组时不要使用 ArrayList，尽量避免使用 Vector。

11.4 对象

避免在循环中频繁构建和释放对象。

不再使用的对象应及时销毁。

如无必要，不要序列化对象。

11.5 同步

在不需要同步操作时避免使用同步操作类，如能使用 ArrayList 时不要使用 Vector。

尽量少用同步方法，避免使用太多的 synchronized 关键字。

尽量将同步最小化，即将同步作用到最需要的地方，避免大块的同步块或方法等。

11.6 final

将参数或方法声明成 final 可提高程序响应效率，故此：

注意绝对不要仅因为性能而将类、方法等声明成 final，声明成 final 的类、方法一定要确信不再被继承或重载！

不需要重新赋值的变量（包括类变量、实例变量、局部变量）声明成 final

所有方法参数声明成 final
私有(private)方法不需要声明成 final
若方法确定不会被继承，则声明成 final

11.7 垃圾收集和资源释放

不要过分依赖 JVM 的垃圾收集机制，因为你无法预测和知道 JVM 在什么时候运行 GC。
尽可能早的释放资源，不再使用的资源请立即释放。
可能有异常的操作时必须在 try 的 finally 块中释放资源，如数据库连接、IO 操作等：

```
Connection conn = null;
try{
    //do something
}catch(Exception e){  //异常捕捉和处理
    e.printStackTrack();
}finally{
    //判断 conn 等是否为 null
    if(null != conn){
        conn.close();
    }
}//end try...catch...finally
```

第12章　其他

12.1 目录结构

目录结构由用 AntColony 之 newapp 建立新项目时自动建立，参考结构如下：

目录			说明
ProjectName			项目目录，当和项目名一致
	/build		构建目录
		/classes	类构建目录
	/src		源码目录
		/java	Java 类文件
		/test	单元测试类文件
		/hibernate	Hibernate 之映射文件
	/config		配置目录
		/app	应用信息配置
		/log	日志配置信息
		/web	Web 配置信息
		/hibernate	Hibernate 配置信息
		/struts	Strutsp 配置信息
	/metadata		元数据目录
		/db	数据库脚本
	/docs		文档目录
		/api	JavaDoc 目录
		/j2h	Java2Html 目录
		/design	设计文档目录
	/lib		库文件目录
		/buildtime	构建时库目录
		/runtime	运行时库目录
	/web		Web 目录
		/module	模块目录
		/images	图片目录
		/web-inf	WEB-INF 目录
	/log		日志目录
		/buildtime	构建时日志
		/runtime	运行时日志
		/test	测试日志

12.2 CVS 注释与标记

每次向 CVS 服务器提交时都必须给出注释，以标明本次提交所作的改动以及目的等。

在开发周期中，除由系统定时给项目打上时间戳标记外，当在各个阶段对项目进行标记，如将第一次在 CVS 库中建立项目时命名为 baseline01，在第 4 个里程碑后命名为 M4。

项目结束时相关代码当即冻结。新阶段时当将相应代码版本提升，如从 1.0 提升到 2.0。

CVS 中尽量不放置大容量二进制文件，如某些 jar 运行库等。

具体参见《CVS 手册》。

12.3

第13章　附录

13.1 CVS 标识符

标识符	说明
Id	标识，包括文件名，版本，提交时间，提交者等信息 $Id: HibernateFilter.java,v 1.1 2003/11/17 13:20:07 l_walker Exp $
$Author$	最后提交者 $Author: l_walker $
$Reversion$	当前版本（服务器版本） $Revision: 1.1 $
$Date$	最后一次提交时间 $Date: 2003/11/17 13:20:07 $

其他标识符及详细说明请参见 CVS 文档。

13.2 注释模板

以下注释均针对 Eclipse 和 WSAD，其他开发工具可相应调整之。在 Eclipse 中，输入 CVS 标识符必须以双$开头，否则 Eclipse 会认为其为 Eclipse 内置参数而报错。

文件头注释：

```
/* ============================================
 * $$Id$$
 * Created [${date} ${time}] by ${user}
 * ============================================
 *
 * ${project_name}$
 *
 * ============================================
 * The Skyinn Software License v1.0 content.
 *  （详细内容参见 13.4 版权声明模板）
 * ============================================
 */
```

类注释：

```
/**
 * <p>${file_name}</p>
 *
 * @author $$Author$$
 * @version $$Revision$$ $$Date$$
 */
```

```
${date}          创建日期
${time}          创建时间
${user}          创建者
${project_name}  项目名称
${file_name}     文件名
```

请在您的编辑器中使用相应标识符替代，若无，则可使用相应 CVS 标识符替代，但当使用 CVS 标识符替代后，当创建文件后当删除相应标记，避免因 CVS 的自动更新标识符功能而导致创建信息的丢失等。

13.3 常用缩写简表

缩写	全字	说明
app	application	应用程序
ctx	context	上下文信息
mgr	manager	管理者，经理
conn	connection	连接（数据库连接）
sys	system	系统

13.4 版权声明模板

```
/* ============================================================
 * $$Id$$
 *
```

```
 * Created: [${date} ${time}] by ${user}
 * ======================================================================
 *
 * ${project_name}
 *
 * ======================================================================
 * Skyinn Apache Style License v1.0
 *
 * Copyright (c) Skyinn Group, 2002-2004
 * ======================================================================
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions, and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above
 *    copyright notice, this list of conditions, and the disclaimer
 *    that follows these conditions in the documentation and/or other
 *    materials provided with the distribution.
 *
 * 3. The name "Skyinn" must not be used to endorse or promote
 *    products derived from this software without prior written
 *    permission.For written permission, please contact
 *    walker@skyinn.org.
 *
 * 4. Products derived from this software may not be called "Skyinn",
 *    nor may "Skyinn" appear in their name, without prior written
 *    permission from Han Qing(walker@skyinn.org).
 *
 * 5. Redistributions of any form whatsoever must retain the
 *    following acknowledgment:
 *    "This product includes software developed by Skyinn Group."
 *
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR(S) BE LIABLE FOR ANY
 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING
 * IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * For more information on Skyinn, please
 * see <http://www.skyinn.org>.
 *
 * ======================================================================*/
```

13.5 示例代码

```
/* ============================================
 * $Id: DAOFactory.java,v 1.4 2003/10/20 14:18:44 l_walker Exp $
 * Created on [2003-10-8 22:48:12] by l_walker
 * ============================================
 * The Skyinn Software License v1.0
 *
* Content...
 * ============================================
 */
package org.skyinn.quasar.dao;

import org.apache.commons.collections.FastHashMap;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.skyinn.quasar.config.ApplicationConfiguration;
import org.skyinn.quasar.util.StringUtil;


/**
 * <p>DAO 工厂类。</p>
 *
 * @author $Author: l_walker $
 * @version $Revision: 1.4 $ $Date: 2003/10/20 14:18:44 $
 */
public class DAOFactory {
    //~ Static fields/initializers =================================

    /**DAOFactory singleton instance.*/
    private static DAOFactory instance = new DAOFactory();

    //~ Instance fields ============================================

    /** Logging */
    private Log log = LogFactory.getLog (this.getClass ());

    /**DAO pool.*/
    protected FastHashMap daos = new FastHashMap();

    /**DAO configuration.*/
    protected DAOConfig daoConfig = null;

    //~ Constructors ===============================================

    /**
```

```
     * Default Construtor.
     */
    private DAOFactory () {
        super();
        daoConfig =
ApplicationConfiguration.getInstance()
.getDAOConfiguration();
    }

    //~ Methods ================================================

    /**
     * 取 DAO 工厂,Singleton 模式以确保系统中只有一个 DAOFactory 实例。
     *
     * @return DAO 工厂实例
     */
    public static synchronized DAOFactory getInstance () {
        if (null == instance) {
            instance = new DAOFactory();
        }
        return instance;
    }

    /**
     * 从 DAO 池中取对应 KEY 的 DAO 实例。
     * 若该实例未存在，则从 DAO 配置中取该 DAO 的 Mapping，并从中取该 DAO 的实现
类的类名，
     * 初始化之并将其置入池中缓存。
     *
     * @param key DAO Key，默认为该类（接口）全限定名
     * @return DAO 实例
     * @throws DAOException 若
     */
    public synchronized DAO findDAOByKey (final String key)
        throws DAOException {

        //get dao instance from dao pool
        DAO daoInstance = (DAO) daos.get (key);

        //get dao mapping
        final DAOMapping daoMapping = daoConfig.findDAOMapping (key);

        //if null or different type bewteen the current dao and it's mapping
        if ((null == daoInstance) ||
                !daoInstance.validateType (daoMapping.getCurrentType ())) {
            try {
```

```
                final String daoImplClass =
                    daoMapping.findDAOImplClass (daoMapping
.getCurrentType ());

                if (StringUtil.isNullOrEmpty (daoImplClass)) {
                    throw new DAOException(
"Not found DAO implement class of:["
+ daoMapping + "]");
                }

                //new instance
                Class clazz = Class.forName (daoImplClass);
                daoInstance = (DAO) clazz.newInstance ();

                //set current type
                daoInstance.setCurrentType (daoMapping.getCurrentType ());

                //add to dao pool
                daos.setFast (false);
                daos.put (key, daoInstance);
                daos.setFast (true);

                if (log.isDebugEnabled ()) {
                    log.debug ("A DAO instance created:[" + key + "]");
                }
            } catch (ClassNotFoundException e) {
                log.error ("ClassNotFoundException:" + e.getMessage());
                throw new DAOException(e);
            } catch (InstantiationException e) {
                log.error ("InstantiationException:" + e.getMessage());
                throw new DAOException(e);
            } catch (IllegalAccessException e) {
                log.error ("IllegalAccessException:" + e.getMessage());
                throw new DAOException(e);
            }//end try...catch...
        }//end if...daoInstance is null?
        return daoInstance;
    }//end findDAOByKey
}//EOC DAOFactory
```