# Yaheen 代码规范

**文档版本：V1.1.0**

**目 录**

Yaheen 代码规范 1

1. 概述 4
   1. 编写目的 4
   2. 适用范围 4
   3. 术语和缩写 5
   4. 参考资料 5
2. 字符集 5
3. 开发工具 5
4. 代码组织与风格 6
   1. 基本原则 6
   2. 缩进 6
   3. 长度 6
   4. 行宽 6
   5. 间隔 7
   6. 对齐 7
   7. 括号 7
5. 注释 7
   1. 基本原则 7
   2. JavaDoc 7
   3. 注释的格式 8
   4. 方法注释 8
   5. 失效代码注释 8
   6. 代码细节注释 9
   7. 特殊代码的注释 9
   8. 注释参考表 9
6. 命名 10
   1. 基本原则 10
   2. 文件、包 11
   3. 类、接口 11
   4. 属性 11
      1. 常量 11
      2. 变量和参数 12
      3. 集合 12
      4. 其他 12
   5. 方法 12
   6. SpringMVC命名规范 12
      1. Action的命名 12
      2. JSP的命名 12
   7. DAO命名规范 13
      1. 接口命名 13
      2. 接口实现命名 13
   8. Service命名规范 13
      1. 接口命名 13
      2. 接口实现命名 14
   9. 配置文件命名规范 14
      1. Hibernate配置文件 14
      2. Spring配置文件 14
      3. 查询配置文件 14
   10. 命名约定表 14
7. 类与接口 16
   1. 基本原则 16
   2. 抽象类与接口 16
   3. 继承与组合 16
8. 方法 17
   1. 基本原则 17
   2. 参数与返回值 17
9. 表达式与语句 17
   1. 基本原则 17
   2. 控制语句 18
   3. 循环语句 19
10. 错误与异常 19
    1. 基本原则 19
    2. 已检查异常和运行时异常 20
    3. 异常的捕捉与处理 20
11. 测试和Bug跟踪 20
    1. 基本原则 20
    2. 测试驱动开发 21
    3. Junit单元测试 21
    4. 自动测试与持续集成 21
    5. Bug跟踪与缺陷处理 21
12. 性能与安全 22
    1. 基本原则 22
    2. String与StringBuffer 22
    3. 集合 23
    4. 对象 23
    5. 同步 23
    6. final 23
    7. 垃圾回收与资源释放 23

# 概述

## 编写目的

本规范的目的是使本项目组成员以标准的、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；并以此形成项目组的编码约定，以提高程序的可靠性、可读性、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。

## 适用范围

本规范适用于中国电信集团级CRM项目组的设计、开发以及维护、升级等。

本规范适用于中国电信集团级CRM项目组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。

## 术语和缩写

|  |  |  |
| --- | --- | --- |
| 术语和缩写 | 解释 | 备注 |
|  |  |  |

## 参考资料

|  |  |
| --- | --- |
| 参考文件 | 备注 |
| 《Java编程指南》 |  |
| 《Sun Java语言编码规范》 |  |
| 《Java Rules》 |  |
| 《Effictive Java 》 |  |
| 《Java技术手册》 |  |

# 字符集

在整个系统中，项目默认编码、新建代码文件、配置文件、资源文件、页面默认编码都要统一采用UTF-8的编码格式！

# 开发工具

本项目统一采用Genuitec的Eclipse Luna工具进行开发。

# 代码组织与风格

## 基本原则

代码的组织和风格的基本原则是：便于自己开发，易于与他人交流。因为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。本节所涉及的内容一般都可以在Java集成编辑环境中进行相应设置，也可由Ant等调用checkstyle等来进行自动调整。

## 缩进

子功能块应该在其父功能块后缩进，当功能块过多而导致缩进过深的时候应当将子功能模块提取成子函数。代码中以TAB（4个字符）缩进，在编辑器中请将TAB设置成空格替代，否则在不同的编辑器或者设置下会导致TAB的长度不等而影响整个程序代码的格式。

缩进示例：

**public** **void** methodName(){

**if**(some condition){

**for**(...){

//same sentence

}//end for

}//end if

}

## 长度

为了便于阅读和理解，单个函数的有效代码长度应尽量控制在**150**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。

单个类也不宜于过大，当出现此类情况时当将相应的功能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。

## 行宽

页宽应该设置为**120**字符。一般不要超过超过这个宽度，这会导致在某些机器中无法用一屏来完整显示，但这一设置也可灵活调整，在任何情况下，超长的语句应该在一个逗号或一个操作符前折行。一个语句折行后应该比原来的语句再缩进一个TAB或者4个空格，以便于阅读。

## 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。

## 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；当控制或者循环中的条件比较长时换行则对齐并注释各条件；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。

## 括号

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

较长的方法以及类、接口等右括号使用//end…等标识其结束。

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，若换行当与其前导语句手字符对齐。

# 注释

## 基本原则

注释应该增加代码的清晰度，代码注释的目的要使代码更易于被其他开发人员理解；如果你的程序不值得注释，那么它很可能也不值得运行；应避免使用装饰性内容；应保持注释的简介；注释信息不仅要包括代码的功能还应给出原因；不要为了注释而注释；除变量定义等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。

## JavaDoc

对类、方法、变量等的注释需要符合JavaDoc规范，对于每个类，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，以便生成的JavaDoc易阅读和理解，类注释中当包含版本和作者信息。

## 注释的格式

* 注释中的第一个句子要以（英文）句号、问号或者感叹号结束。Javadoc生成工具会将注释中的第一个句子放在方法汇总表和索引中。
* 为了在JavaDoc和IDE中能快速链接跳转到相关联的类与方法，尽量多的使用@see xxx.MyClass，@see xx.MyClass#find(String)。
* Class必须以@author 作者名声明作者，不需要声明@version与@date，由版本管理系统保留此信息。
* 如果注释中有超过一个段落，用<p>分隔，示例代码以<pre></pre>包裹。
* 标识(java keyword, class/method/field/argument名，Constants) 以<code></code>包裹。
* 标识在第一次出现时以{@linkxxx.Myclass}注解以便JavaDoc与IDE中可以链接。

## 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。

方法注释示例：

/\*\*\*

\* 执行查询。

\* 该方法调用Statement的excuteQuery（sql）方法并返回ResultSet结果集

\* **@param** sql 标准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 excuteQuery(String sql)**throws** SQLException{

//Statement和sql语句都不能为空

**if**(**null** != stmt && !StringUtil.Empty(sql)){

//返回查询执行结果

**return** stmt.executeQuery(sql);

}

**return** **null** ;

}

## 失效代码注释

由/\*...\*/界定，标准的C-Style的注释。专用于注释已失效的代码。

/\*

 \* Comment out the code

 \* String s = "hello";

\* System.out.println(s);

 \*/

## 代码细节注释

由//界定，专用于注释代码细节，即使有多行注释也仍然使用//，以便与用/\*\*/注释的失效代码分开，除了私有变量外，不推荐使用行末注释。

class MyClass {

private int myField; // An end-line comment.

public void myMethod {

//a very very long

//comment.

if (condition1) {

//condition1 comment

...

} else {

//elses condition comment

...

}

}

}

## 特殊代码的注释

* 代码质量不好但能正常运行，或者还没有实现的代码用//TODO: 或 //XXX:声明
* 存在错误隐患的代码用//FIXME:声明

## 注释参考表

注释参考表：

|  |  |
| --- | --- |
| **项目** | **注释内容** |
| 参数 | 参数类型  参数用来做什么  约束或者前提条件  示例 |
| 字段/属性 | 字段属性  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  采用的不变量  并行策略 |
| 编译单元（文件） | 每一类/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 |
| 或许成员函数 | 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  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  适当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 |
| 包 | 包的功能和用途 |

# 命名

## 基本原则

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

使用可以准确说明变量/字段/类/接口/包等完整的英文描述符。例如，采用类似firstName，listAllUsers 或者CorpoateCustomer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，但本规范对包、类、接口、方法、变量、字段等**不得使用汉字**等进行命名。

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，而不是Client。

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。

尽量少用缩写，但如果一定要使用，当使用公共缩写或者习惯缩写等，如实现（implement）可缩写成**impl**，经理（manager）可缩写成**mgr**等，**严禁滥用缩写**。

避免使用长名字（最好不超过**25**个字母）；避免使用相似或者仅在大小写上有区别的名字；避免使用数字，但可用**2代替to**，用**4代替for**等，如：go2Jsp。

## 文件、包

文件名当与其类严格相同，所有单词首字母大写；

**包名**一般以**项目**或者**模块名**命名，少用缩写和长名，**一律小写**；

基本包：net.yasion.common.cthq.crm,所有包、文件都从属于此包；

包名按照如下：

[基本包].[公司名缩写].[核心或者子系统名].[模块名].[子模块名].…

如：

net.yasion.tlep.crm.core

net.yasion.tlep.crm.jdbc

net.yasion.tlep.crm.util

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

## 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。一把采用名词。接口可带**I**前缀或**able**、**ible**、**er**等后缀。

## 属性

### 常量

采用完整的英文大单词，在词与词之间用**下划线连接**，如DEFAULE\_VALUE。

### 变量和参数

对于不能清楚地识别改变量类型的变量应使用**类型缩写**作为其前缀，如字符串使用strXXX，boolean使用isXXX，hasXXX等。除第**一个单词**外其余单词首字母大写。

对私有变量可以使用\_作为前缀，但是在其存取方法中将其前缀去掉。

### 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，名字由所有非开头的单词的第一个字母大写，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

### 其他

命名时采用复数来表示他们代表多值，例如orderItems。

## 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……)。

## SpringMVC命名规范

### Action的命名

Action的命名：**XxxAct**。例如UserAct。

Action中的方法使用add、edit、save 、view、list 、delete等含义清晰的名称来表达方法，

### JSP的命名

这里主要针对jsp页面命名。Jsp页面我们是分模块进行管理的，不同模块的jsp页面应该放到不同的文件夹里。Jsp页面的命名应该是与Action中的方法命名对应，方便管理。

* 与Action中add，edit方法对应的页面命名为：edit.jsp，例如：edit.jsp 。
* 与Action中list方法对应的页面命名为：list.jsp，例如：list.jsp 。
* 与Action中viewt方法对应的页面命名为：view.jsp，例如：view.jsp 。

## DAO命名规范

### 接口命名

接口命名：**IXxxDAO**。例如：**IUserDAO** 。

接口对应的方法命名可以对照Action命名规则。

saveXxx/save：此方法用来新增记录，例如 ：saveUser(User user);/save(User user);

updateXxx/update：此方法用来修改记录，例如 ：updateUser(User user)/update(User user);

deleteXxx：此方法用来删除某条或者多条记录，例如 ：deleteByRoleId（int id）；

listXxx：此方法用来显示某条记录，例如 ：listStatisticsByQuestionnaire（int id）；

selectXxx：此方法用显示符合条件的列表，例如 ：selectUSerByDepartId（int departId）；

### 接口实现命名

接口实现命名：**XxxDAOImpl**，例如**UserDAOImpl**

## Service命名规范

### 接口命名

Service接口命名：**IXxxService**，例如**IUserServce**，里面的方法可以参照DAO里面的方法命名

saveXxx/save：此方法用来新增记录，例如 ：saveUser(User user);/save(User user);

updateXxx/update：此方法用来修改记录，例如 ：updateUser(User user)/update(User user);

removeXxx：此方法用来删除某条或者多条记录，例如 ：removeById（int id）；

findXxx：此方法用来显示某条记录，例如 ：findByDTO（int id）；

selectXxx：此方法用显示符合条件的列表，例如 ：selectUSerByDepartId（int departId）；

### 接口实现命名

接口实现命名：**XxxServiceImpl**，例如**UserServiceImpl**

## 配置文件命名规范

### Hibernate配置文件

统一存放hibernate配置文件，配置文件统一存放在/WEB-INF/config/hibernate/mapping/子系统或者核心系统名称/\*.hbm.xml；按照表来划分hbm.xml文件。

hbm.xml文件采用**union-subclass**方式编写，因为里面使用了**继承映射**。具体查考开发文档。

### Spring配置文件

BeanId的名称采用模块名+功能名+bean的形式命名；(如果有)

### 查询配置文件

Query文件name名称采用模块名+功能名+query的形式命名；(如果有)

## 命名约定表

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 |
| 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, ible 或者 er 等后缀，但这不是必需的 | IRunnable  IPrompter |
| 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com.telecom |  |
| 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() |

# 类与接口

## 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 UserManager 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。

## 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。

## 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。

# 方法

## 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;

## 参数与返回值

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(){}

# 表达式与语句

## 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。

## 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())...

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

## 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){

}

# 错误与异常

## 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。

## 已检查异常和运行时异常

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。

## 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

}

# 测试和Bug跟踪

## 基本原则

测试不通过的代码不得提交到版本控制库库或者发布。

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。

## 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。

## 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。

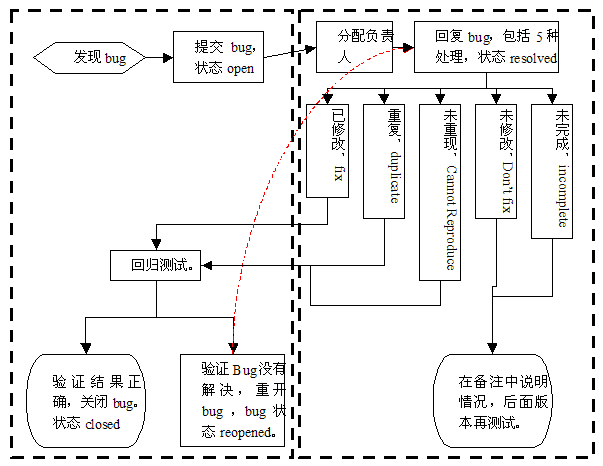
## 自动测试与持续集成

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。

## Bug跟踪与缺陷处理

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，禁止隐瞒 Bug。

测试人员 开发人员



# 性能与安全

## 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！

## String与StringBuffer

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；

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。

## 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。

## 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。

## 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。

## 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；

## 垃圾回收与资源释放

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