《代码整洁之道》

【美】Robert C. Martin著

韩磊 译

人民邮电出版社

2017-07-22：开始~Page11

2017-07-26：Page12~Page28

2017-07-27：Page29~Page35

2017-07-31：Page36~Page38

2017-08-01：Page39~Page42

2017-08-08：Page43~Page44

2017-08-09：Page45~Page46

2017-08-28：Page47~Page50

2017-08-29：Page51~Page54

2017-08-30：Page55~Page55

2017-09-01：Page56~Page59

2017-09-03：Page60~Page62

影响代码质量的因素：架构、项目管理、代码质量等

大约在1951年，一种名为“全员生成维护”（Total Productive Maintenance， TPM）的质量保证手段在日本出现。TPM的主要支柱之一是所谓的5S原则体系。

5S哲学包括以下概念：

* 整理（Seiri），或谓组织：搞清楚事物之所在——通过切当地命名之类的手段——至关重要
* 整顿（Seiton），或谓整齐：物皆有其位，而后物尽归其位
* 清楚（Seiso），或谓清洁：对于那种四处遗弃的带注释的代码以及反映过往或期望的无注释代码，应除之而后快。
* 清洁（Seiketsu），或谓标准化：在开发组内使用一贯的代码风格和实践手段。
* 身美（Shitsuke），或谓纪律（自律）：在实践中贯彻规程，并时时体现于个人工作上，而且要乐于改进。

# 序

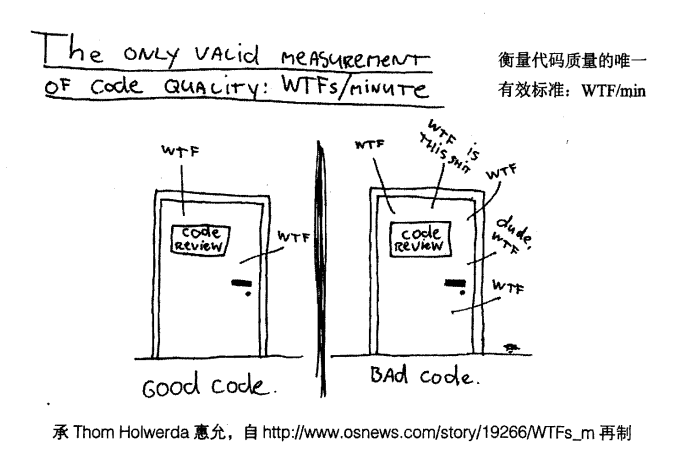
我们过早地放弃在代码上的工作，并不是因为它业已完成，而是因为我们的价值体系关注外在表现甚于关注要交付之物的本质。

# 代码猴子与童子军军规

代码猴子：低水平编码者

童子军军规：离开时要比发现时更整洁

# 前言



批注：好搞笑啊，哈哈哈哈~

如何保证我们在那道正确的门后做补救工作？答案是：技艺（craftsmanship）。

习艺之要有二：知和行。你应当习得有关原则、模式和实践的知识，穷尽应知之事，并且要对其了如指掌，通过刻苦实践掌握它。

学写整洁代码很难。它可不止于要求你掌握原则和模式。你得在这上面花功夫。你须自行实践，且体验自己的失败。你须观察他人的实践与失败。你须看看别人是怎样蹒跚学步，再转头研究他们的路数。你须看看别人是如何绞尽脑汁做出决策，又是如何为错误决策付出代价。

阅读本书要多用心思，要非常用功。如何用功？阅读代码——大量代码。而且你要去琢磨某段代码好在什么地方、坏在什么地方。

# 整洁代码

## 1.3 混乱的代价

### 1.3.5 什么是整洁代码

我喜欢优雅和高效的代码。代码逻辑应当直截了当，叫缺陷难以隐藏；尽量减少依赖关系，使之便于维护；依赖某种分层战略完善错误处理代码；性能调至最优，省得引诱别人做没规矩的优化，搞出一堆混乱来。整洁的代码只做好一件事。

**Bjarne Stroustrup，C++语言发明者，C++ Programming Language（中译版《C++程序设计语言》）一书作者**

破窗理论：窗户破损了的建筑让人觉得似乎无人照管。于是别人也再不关心。他们放任窗户继续破损。最终自己也参与破坏活动，在外墙上涂鸦，任垃圾堆积。一扇破损的窗户开辟了大厦走向倾颓的道路。

整洁的代码简单直接。整洁的代码如同优美的散文。整洁的代码从不隐藏设计者的意图，充满了干净利落的抽象和直截了当的控制语句。

**Grady Booch，Object Oriented Analysis and Design with Applications（中译版《面向对象分析与设计》）一书作者**

整洁的代码应可由作者之外的开发者阅读和增补。它应该有单元测试和验收测试。它使用有意义的命名。它只提供一种而非多种做一件事的途径。它只有尽量少的依赖关系，而且要明确地定义和提供清晰、尽量少的API。代码应通过其字面表达含义，因为不同的语言导致并非所有必需信息均可通过代码自身清晰表达。

**“老大” Dave Thomas，OTI公司创始人，Eclipse战略**

代码应在字面上表达其含义。这一观点源自Knuth的“字面编程”（literate programming）。结论就是应当用人类可读的方式来写代码。

……

## 1.4 思想流派

要通过多元思维来思考问题，取各家之长。

## 1.5 我们是作者

读与写花费时间的比例超过10:1。写新代码时，我们一直在读旧代码。不读周边代码的话就没法写代码。编写代码的难度，取决于读周边代码的难度。要想干得快，要想早点做完，要想轻松写代码，先让代码易读吧。

## 1.6 童子军军规

让营地比你来时更干净。

## 1.7 前传与原则

本书可认为是Agile Software Development: Principles, Patterns, and Practices（中译版《敏捷软件开发：原则、模式与实践》，简称PPP）的“前传”。

本书引用的设计原则有：单一权责原则（Single Responsibility Principle, SRP）、开放闭合原则（Open Closed Principle, OCP）和依赖倒置原则（Dependency Inversion Principle, DIP）等。

## 1.8 小结

本书展示了号程序员的思维过程，还有他们使用的技巧、技术和工具。

# 有意义的命名

## 2.1 介绍

## 2.2 名副其实

变量、函数或类的名称应该已经答复了所有的大问题。它该告诉你，它为什么会存在，它做什么事，应该怎么用。如果名称需要注释来补充，那就不算是名副其实。

反例：

int d; //消逝的时间，以日计

正例：

int elapsedTimeInDays;

int daysSinceCreation;

int daysSinceModificaiton;

int fileAgeInDays;

反例：

public List<int[]> getThem(){

List<int[]> list1 = new ArrayList<int[]>();

for (int[] x : theList)

if(x[0] == 4)

list1.add(x);

return list1;

}

正例：

public List<int[]> getFlaggedCells(){

List<int[]> flaggedCells = new ArrayList<int[]>();

for (int[] cell : gameBoard)

if (cell[STATUS\_VALUE] == FLAGGED)

flaggedCells.add(cell);

return flaggedCells;

}

【批注】反例中并不清楚代码具体在做什么事情，而正例中讲的是在开发一种扫雷游戏。盘面名称由theList改为gameBoard，0下标改为STATUS\_VALUE，4改为FLAGGED表示“已标记”。

更进一步：

public List<Cell> getFlaggedCells(){

List<Cell> flaggedCells = new ArrayList<Cell>();

for (Cell cell : gameBoard)

if (cell.isFlagged())

flaggedCells.add(cell);

return flaggedCells;

}

【批注】int数组表示单元格，改为用一个类来表示。

## 2.3 避免误导

**程序员必须避免留下掩藏代码本意的错误线索。应当避免使用与本意相悖的词。**

例如，别用accountList来指称一组账号，除非它真的是List类型，使用accountGroup，bunchOfAccounts或accounts会好一些。

避免使用不同之处较小的名称。

例如，

XYZControllerForEfficientHandlingOfStrings和XYZControllerForEfficientStorageOfStrings。

**以同样的方式写出同样的概念。拼写前后不一致就是误导。**

**避免使用小写字母l和大写字母O作为变量。**

## 2.4 做有意义的区分

**以数字系列命名（a1，a2，……aN）是依义命名的对立面。**

反例：

public static void copyChars(char a1[], char a2[]){

for(int i = 0; I < a1.length; i++){

a2[i] = a1[i];

}

}

上述代码的参数名改为source和destination会好很多。

**废话是另一种没意义的区分。**

例如：

Product类、ProductInfo或ProductData类，它们的名称虽然不同，意思却无区别。

Variable一词永远不应当出现在变量名中。Table一词永远不应当出现在表名中。

## 2.5 使用读得出来的名称

反例：genymdhms（生成日期，年、月、日、时、分、秒）

## 2.6 使用可搜索的名称

单字母名称和数字常量有个问题，就是很难再一大篇文字中找出来。

单字母的名称仅用于短方法中的本地变量。名称长短应用与其作用域大小相对应。若变量或常量可能在代码中多处使用，则应赋其以便于搜索的名称。

反例：

for(int j = 0; j < 34; j++){

s += (t[j]\*4)/5;

}

正例：

int realDaysPerIdealDay = 4;

cons tint WORK\_DAYS\_PER\_WEEK = 5;

int sum = 0;

for(int j = 0; j < NUMBER\_OF\_TASKS; j++){

int realTaskDays = taskEstimate[j] \* realDaysPerIdealDay;

int realTaskWeeks = (realdays / WORK\_DAYS\_PER\_WEEK);

sum += realTaskWeeks;

}

## 2.7 避免使用编码

把类型或作用域编进名称里面，徒然增加解码的负担。带编码的名称通常也不便发音，容易打错。

### 2.7.1 匈牙利语标记法（Hungarian Notation, HN）

在Windows的C语言API时代，所有名称要么是个整数句柄，要么是个长指针或者void指针，要不然就是string的几种实现，那时编译器不做类型检查，程序员需要使用匈牙利语标记法帮助自己记住类型。

**Java程序员不需要类型编码。**

PhoneNumber phoneString;

// 类型变化时，名称并不变化！

### 2.7.2 成员前缀

也不必用m\_前缀来标明成员变量。应当使用某种可以高亮或用颜色标出成员的编辑环境。

private String m\_dsc;// The textual description

void setName(String name){

m\_dsc = name;

}

public class Part {

String description;

void setDescription(String description){

this.description = description;

}

}

### 2.7.3 接口和实现

避免使用前导字母I，如果接口和实现必须选一个编码的话，选择实现会好一些，如ShapeFactory和ShapeFactoryImp。

## 2.8 避免思维映射

在作用域较小、也没有名称冲突时，循环计数器自然有可能被命名为i或j或k。（但千万别用字母l）这是因为传统上管用单字母名称做循环计数器。然而，在多数其他情况下，单字母名称不是个好选择；读者必须在脑中将它映射为真实概念。仅仅是因为有了a和b，就要取名为c，实在并非像样的理由。

**名称要明确。**

## 2.9 类名

类名和对象名应该是名词或名词短语，如Customer、WikiPage、Account和AddressParser。避免使用Manager、Processor、Data或Info这样的类名。类名不应该是动词。

## 2.10 方法名

方法名应当是动词或动词短语，如postPayment、deletePage或save。属性访问器、修改器和断言应该根据其值命名，并依JavaBean标准加上get、set和is前缀。

string name = employee.getName();

customer.setName(“mike”);

if(paycheck.isPosted())…

重载构造器时，使用描述了参数的静态工厂方法名。例如，

Complex fulcrumPoint = Complex.FromRealNumber(23.0);

通常好于

Complex fulcrumPoint = new Complex(23.0);

可以考虑将相应的构造器设置为private，强制使用这种命名手段。

## 2.11 别扮可爱

扮可爱的做法在代码中经常体现为使用俗语或俚语。例如，别用whack()表示kill()。别用eatMyShorts()这类与文化紧密相关的笑话来表示abort()。

## 2.12 每个概念对应一个词

给每个抽象概念选一个词，并且一以贯之。例如，使用fetch、retrieve和get来给在多个类中的同种方法命名。你怎么记得住哪个类中是哪个方法呢？

## 2.13 别用双关语

避免将同一个单词用于不同目的。同一术语用于不同概念，基本上就是双关语了。

## 2.14 使用解决方案领域名称

尽量使用计算机科学（Computer Science，CS）术语、算法名、模式名、数学术语，少用依据问题所涉领域来命名。

## 2.15 使用源自所涉问题领域的名称

如果不能用程序员熟悉的术语来给手头的工作命名， 就采用从所涉问题领域而来的名称命名。

## 2.16 添加有意义的语境

需要用有良好命名的类、函数或名称空间来放置名称，给读者提供语境。如果没这么做，请给名称添加前缀。

反例：

firstName, lastName, street, houseName, city, state, zipcode这些变量在一起时，构成了一个地址，如果在某个方法中单独出现一个state变量，就很难推测那是地址的一部分。

正例：

addFirstName, addLastName, addrState，增加前缀以提供语境，让读者知道这些变量隶属某个更大的概念。

更好地方案是创建名为Address的类。

代码清单2-1 语境不明确的变量

private void printGuessStatistics(char candidate, int count){

String number;

String verb;

String pluralModifier;

if(count == 0){

number = “0”;

verb = “are”;

pluralModifier = “s”;

}else if(count == 1){

number = “1”;

verb = “is”;

pluralModifier = “s”;

}else {

number = Integer.toString(count);

verb = “are”;

pluralModifier = “s”;

}

String guessMessage = String.format(“there %s %s %s%s”, verb, number, candidate, pluralModifier);

print(guessMessage);

}

代码清单2-2 有语境的变量

public class GuessStatisticsMessage{

private String number;

private String verb;

private String pluralModifier;

public String make(char candidate, int count){

createPluralDependentMessageParts(count);

return String.format(“there %s %s %s%s”, verb, number, candidate, pluralModifier);

}

private void createPluralDependentMessageParts(int count){

if(count == 0){

thereAreNoLetters();

}else if(count == 1){

thereIsOneLetter();

} else {

thereAreManyLetters(count);

}

}

private void thereAreManyLetters(int count){

number = Integer.toString(count);

verb = “are”;

pluralModifier = “s”;

}

private void thereIsOneLetter(){

number = “1”;

verb = “is”;

pluralModifier = “”;

}

private void thereAreNoLetters(){

number = “no”;

verb = “are”;

pluralModifier = “s”;

}

}

## 2.17 不要添加没用的语境

只要短名称足够清楚，就要比长名称好。别给名称添加不必要的语境。

反例：

假如有一个名为“加油站豪华版”（Gas Station Deluxe）的应用，在其中给每个类添加GSD前缀。

## 最后的话

取好名字最难的地方在于需要良好的描述技巧和共有文化背景。

# 函数

## 3.1 短小

通常来说，应该是以下代码清单中的样子。

public static String renderPageWithSetupAndTeardowns(PageData pageData, boolean isSuite){

if(isTestPage(pageData))

includeSetupAndTeardownPages(pageData, isSuite);

return pageData.getHtml();

}

**代码块和缩进**

if语句、else语句、while语句等，其中的代码块应该只有一行。该行大抵应该是一个函数调用语句。这样不但能保持函数短小，而且，因为块内调用的函数拥有较具说明性的名称，从而增加了文档上的价值。

这也意味着函数不应该大到足以容纳嵌套结构。所以，函数的缩进层级不该多余一层或两层。当然，这样的函数易于阅读和理解。

## 3.2 只做一件事

如果函数只是做了该函数下同一抽象层级上的步骤，则函数还是只做了一件事。编写函数的目的是为了把大一些的概念（换言之，函数的名称）拆分为另一抽象层上的一系列步骤。

要判断函数是否不止做了一件事，还有一个方法，就是看是否能再拆出一个函数，该函数不仅只是单纯地重新诠释其实现。

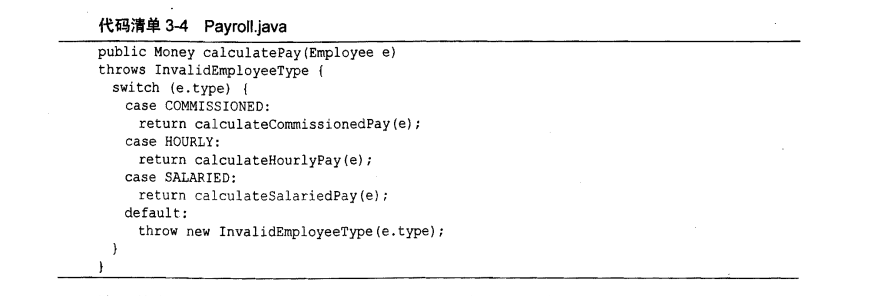
## 3.3 每个函数一个抽象层级

**自顶向下读代码：向下规则**

向下规则：每个函数后面都跟着位于下一抽象层级的函数，这样，在查看函数列表时，就能偱抽象层级向下阅读了。

## 3.4 switch语句

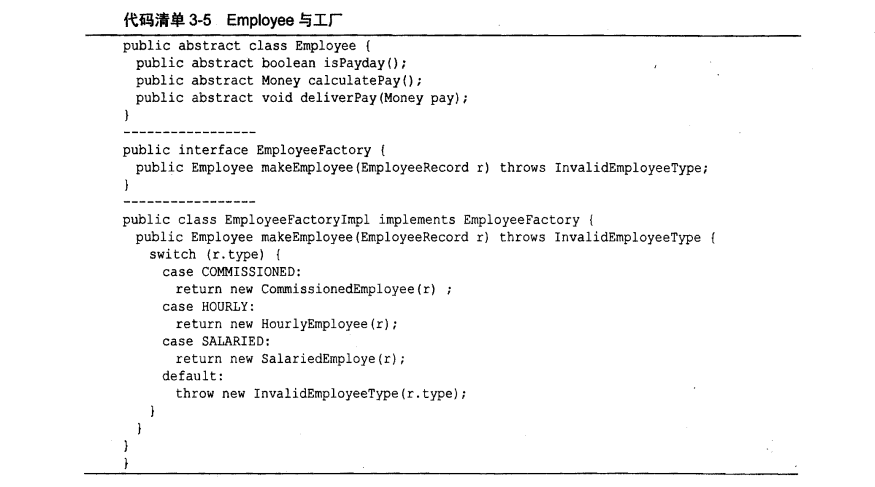
代码清单3-4呈现了可能依赖于雇员类型的仅仅一种操作。



该函数中的问题：

1. 太长，当有新的雇员类型时，还会更长。
2. 不止做了一件事。
3. 违反了单一权责原则。
4. 违反了开放闭合原则。每当添加新类型时，就必须修改它。

该问题的解决方案（如代码清单3-5所示）是将switch语句埋到抽象工厂底下，不让任何人看到。该工厂使用switch语句作为Employee的派生物创建适当的实体，而不同的函数，如calculatePay、isPayDay和deliverPay等，则藉由Employee接口多态地接收派遣。



## 3.5 使用描述性的名称

别害怕长名称。别害怕花时间取名字。

## 3.6 函数参数

最理想的参数数量是零（零参数函数），其次是一（单参数函数），再次是二（双参数函数），应尽量避免三（三参数函数）。有足够特殊的理由才能用三个以上参数（多参数函数）。

我们惯于认为信息通过参数输入函数，通过返回值从函数中输出。因此不推荐使用输出参数。

### 3.6.1 一元函数的普遍形式

Boolean fileExists(“MyFile”)

InputStream fileOpen(“MyFile”)

事件：void passwordAttemptFailedNTimes(int attempts)

避免使用输出参数：void includeSetupPageInfo(StringBuffer pageText)

### 3.6.2 标识参数

避免向函数传入布尔值。违背了函数只做一件事的原则。

### 3.6.3 二元函数

应尽量利用一些机制将二元函数转换为一元函数。

### 3.6.4 三元函数

### 3.6.5 参数对象

当函数需要两个、三个或者三个以上的参数时，应将其中的一些参数封装成类。

### 3.6.6 参数列表

可变参数：public String format(String format, Object… args)

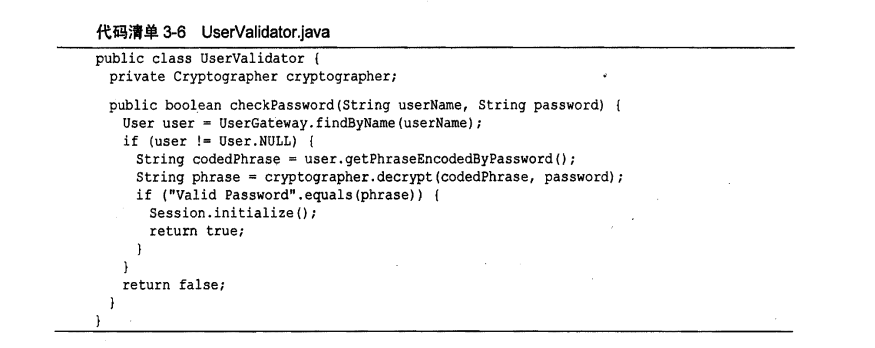
### 3.6.7 动词与关键字

给函数取个好名字，能较好地解释函数的意图，以及参数的顺序和意图。

例如，assertEqual改成assertExpectedEqualsActual(expected, actual)可能会好些。

## 3.7 无副作用

有时，函数会对自己类中的变量做出未能预期的改动。有时，函数会把变量搞成向函数传递的参数或是系统全局变量。



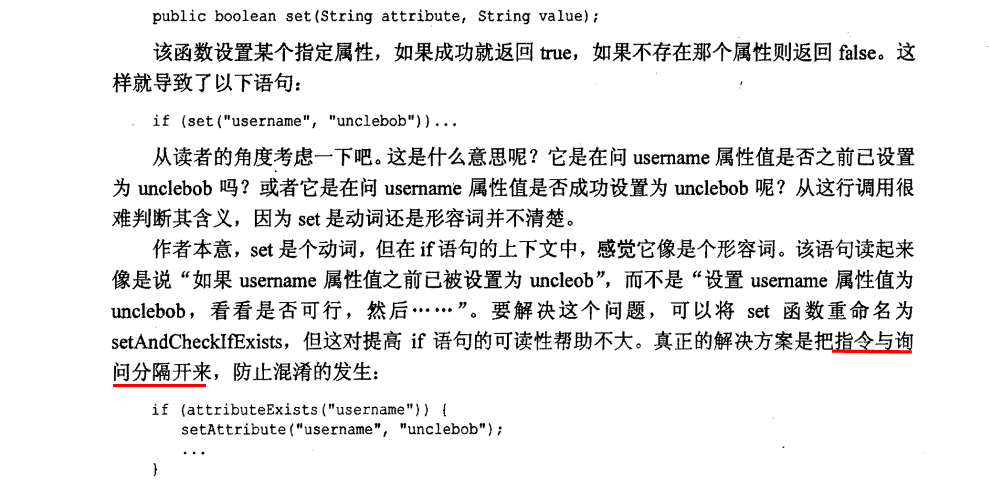
该函数的副作用就在于对Session.initialize()的调用。所以，当某个误信了函数名的调用者想要检查用户有效性时，就得冒抹除现有会话数据的风险。

**输出参数**

普遍而言，应避免使用输出参数。如果函数必须要修改某种状态，就修改所属对象的状态吧。

## 3.8 分割指令与询问

函数要么做什么事，要么回答什么事，但二者不可兼得。函数应该修改某对象的状态，或是返回该对象的有关信息。两样都干常会导致混乱。



## 3.9 使用异常替代返回错误码

从指令式函数返回错误码轻微违反了指令与询问分割的规则。它鼓励了在if语句判断中把指令当做表达式使用。

if(deletePage(page) ==E\_OK)

这不会引起动词/形容词混淆，但却导致更深层次的嵌套结构。当返回错误码时，就是在要求调用者立刻处理错误。

if(deletePage(page) == E\_OK){

if(registry.deleteReference(page.name) == E\_OK){

if(configKeys.deleteKey(page.name.makeKey()) == E\_OK){

logger.log(“page deleted”);

} else {

logger.log(“configKey not deleted”);

}

} else {

logger.log(“deleteReference from registry failed”);

}

} else {

logger.log(“delete failed”);

return E\_ERROR;

}

另一方面，如果使用异常替代返回错误码，错误处理代码就能从主路径代码中分离出来，得到简化：

try {

deletePage(page);

registry.deleteReference(page.name);

configKeys.deleteKey(page.name.makeKey());

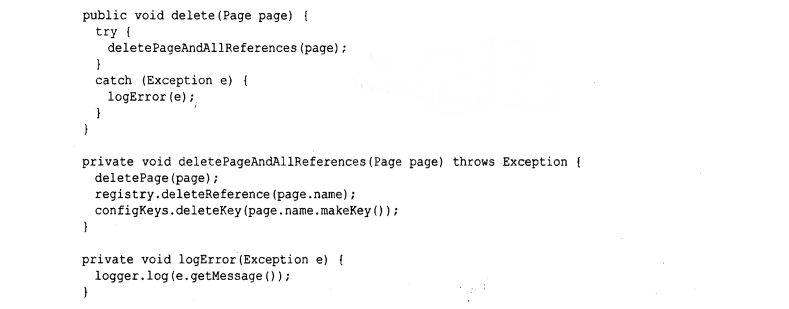
} catch (Exception e) {

logger.log(e.getMessage());

}

### 3.9.1 抽离Try/Catch代码块

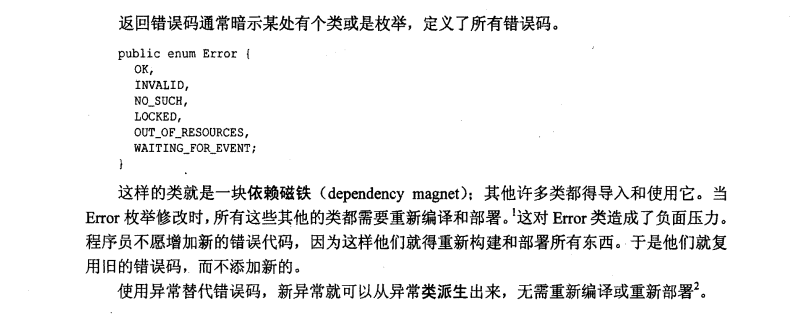
Try/Catch代码块搞乱了代码结构，把错误处理与正常流程混为一谈。最好把try和catch代码块的主体部分抽离出来，另外形成函数。



### 3.9.2 错误处理就是一件事

函数应该只做一件事。错误处理就是一件事。

### 3.9.3 Error.java依赖磁铁



## 3.10 别重复自己

重复会导致代码臃肿，而且当算法改变时需要修改4处地方，这样就会增加4次放过错误的可能性。

## 3.11 结构化编程