# Refactoring \_学习笔记

## 第二章：重构原则

### 什么是重构？

重构是对软件内部结构的一种调整，目的是在不改变软件可观察行为的前提下，提高其可理解性，降低其修改成本。

重构的要点是：

重构之后的软件功能一如既往。

### 为什么要重构？？？

#### 消除重复代码

消除重复代码

“

完成同样一件事，设计不良的程序往往需要更多的代码，这常常是因为代码在不同的地方使用相同的语句做同样的事。

因此改进设计的一个重要方向就是消除重复代码。这个动作的重要性在于方便未来的修改。

然而代码量减少将使未来可能的程序修改动作容易的多。代码愈多，正确的修改就愈困难，因为有更多代码需要理解。

你在这儿做了点修改，系统却不如预期那样工作，是因为有更多代码需要理解，你在这儿做了点修改，系统却不能如预期那样工作，是因为

你没有修改另一处-----那儿的代码做着几乎完全一样的事情，只是所处环境略有不同。如果消除重复代码，你就可以确定所有事物和行为在代码中只表述一次，这正是优秀设计的根本。

”

#### 重构让代码更易读

“准确地说出你要做什么”

#### 良好的设计

“拥有良好的设计才能做到快速开发。良好设计师维持软件开发速度的根本”

“事不过三，三则重构”

#### “重构让添加新功能更简单”

“

重构的另一个原动力是：代码的设计无法帮助我轻松添加我所需要的特性。然后我对自己说：“如果用某种方式来设计，添加特性会简单得多。”

这种情况下我不会因为自己过去的错误而懊恼----我用重构来弥补它。

之所以这么做，部分原因是为了让未来增加新特性时能够更轻松一些，但最主要的原因还是：

我发现这是最快捷的途径。重构是一个快速流畅的过程，一旦完成重构，新特性的添加就会更迅速、更流畅。

”

#### 程序的两面性价值

程序的两面性价值： “今天可以为你做什么”和“明天可以为你做什么”。

#### 增加间接层

#### 重构接口

该如何面对那些必须修改“已发布接口”的重构手法？？？

：

“

简言之，如果重构手法改变了已发布接口，你必须同时维护新旧两个接口，

知道所有用户都有时间对这个变化做出反应。幸运的是，这不太困难，你通常都有办法把事情组织好，让旧接口继续工作。

请尽量这么做：让旧接口调用新接口。当你要修改某个函数名称时，请留下旧函数，让它调用新函数。千万不要复制函数实现。

那会让你陷入重复代码的泥潭中难以自拔。你还应该使用Java提供的deprecation(不建议使用)设施，将旧接口标记为deprecated.

这样一来，你的调用者就会注意到它了。

”

#### 把“大块头软件”重构为封装良好的小型组件

“

将“大块头软件”重构为封装良好的小型组件，然后你就可以逐一对组件做出“重构或重建”的决定。

”

#### 重构与性能

如果你想优化性能，那么你首先需要准确找出性能热点。

“

关于性能，一件很有趣的事情是：如果你对大多数程序进行分析，就会发现它把大半时间都耗费在一小段代码身上。

如果你一视同仁地优化所有代码，90%的优化工作都是白费劲的，因为被你优化的代码大多很少被执行。

你花时间做优化是为了让程序运行的更快，但如果因为缺乏对程序的清楚认识而花费时间，那些时间就都是被浪费掉了。

”

“

在性能优化阶段，你首先应该用一个度量工具来监控程序的运行，让它告诉你程序中哪些地方大量消耗时间和空间。

这样你就可以找出性能热点所在的一小段代码。然后你应该集中关注这些性能热点，并使用持续关注法中的优化手段来、

优化它们。由于你把注意力都集中在热点上，较少的工作量便可显现较好的成果。即便如此你还是必须保持谨慎。

和重构一样，你应该小幅度进行修改。每走一步都需要编译、测试、再次度量。如果没能提高性能，就应该撤销此次修改。

你应该继续这个“发现热点、去除热点”的过程，直到获得客户满意的性能为止。

”

## 第三章：代码的坏味道Bad Smells in Code

“没有任何量度规矩比得上一个见识广博者的直觉”

1.判断一个类内有多少实例变量算是太大？？？

2.一个函数内有多少行代码才算是太长？？？

### 一、Duplicated Code重复代码

"坏味道行列中首当其冲的就是Duplicated Code。如果你在一个以上的地点看到相同的程序结构，

那么可以肯定：设法将它们合而为一，程序会变得更好。"

1.Extract Method

2.Pull Up Method //把两个互为兄弟的子类含有的相同表达式，抽取成方法并把它推入到超类中

3.Extract Class //将重复代码提炼到一个独立类中

### 二、Large Class 过大的类

“和太多实例变量”一样，类内如果有太多代码，也是代码重复、混乱并最终走向死亡的源头。最简单的解决方案是把

多余的东西消弭于类内部。如果有五个"百行函数"，它们之中有很多代码相同，那么或许你可以把它们变成五个“十行函数”和十个提炼出来的“双行函数”。

解决方案： Extract Class

### 三、Shotgun Surgery(散弹式修改)

“

Shotgun Surgery 类似于Divergent Change ,但恰恰相反。如果每遇到某种变化，你都必须在许多不同的类内做出许多小修改，

你所面临的坏味道就是Shotgun Surgery.

如果需要修改的代码散步四处，你不但很难找到它们，也很容易忘记某个重要的修改。

这种情况下你应该使用Move Method 和 Move Field 把所有需要修改的代码放进同一个类。如果眼下没有合适的类可以安置这些代码，就创造一个。

通常可以运用Inline Class 把一系列相关行为放进同一个类。这可能会造成少量Divergent Change ,但你可以轻易处理它。

”

Divergent Change是指“一个类受多种变化的影响”，Shotgun Surgery则是指“一种变化引发多个类相应修改”。这两种情况下你都会希望整理代码，使“外界变化”

与“需要修改的类”趋于一一对应。

#### 最根本的原则是： “将总是一起变化的东西放在一块儿”。

### 四、Data Clump 数据泥团

#### 概要

整个的意思就是： 把那些喜欢一起出现的字段或参数提炼到一个独立的对象中。

#### 原文

“

数据项就像小孩子，喜欢成群结队地待着一块儿。你常常可以在很多地方看到相同的三四项数据：两个类中相同的字段、许多函数签名中相同的参数。

这些总是绑在一起出现的数据真应该又有属于他们自己的对象。首先请找出这些数据以字段形式出现的地方，运用Extract Class将它们提炼到一个独立对象中。

然后将注意力转移到函数签名上，运用Introduce Parameter Object 或Preserve Whole Object 为它减肥。

这么做的直接好处是可以将很多参数列缩短，简化函数调用。

是的，不必在意Data Clumps只用上新对象的一部分字段，只要以新对象取代两个(或更多)字段，你就值回票价啦。

”

“

一个好的评判办法是：删掉众多数据中的一项，。这么做，其他数据有没有因而失去意义？如果它们不再有意义，这就是个明确信号：你应该为它们产生一个新对象。

减少字段和参数的个数，当然可以去除一些坏味道，但更重要的是：一旦拥有新对象，你就有机会让程序散发出一种芳香。得到新对象后，你就可以着手寻找Feature Envy ,这可以帮你

指出能够移至新类中的种种程序行为。不必太久，所有的类都将在它们的小小社会中发挥价值。

”