# TODO

重构的原则：流程方法论

阅读Confluence，完善部分

# 泛读

## 目录

第 1 章　重构，第 一个示例 1

1.1　起点 1

1.2　对此起始程序的评价 3

1.3　重构的第 一步 5

1.4　分解statement 函数 6

1.5　进展：大量嵌套函数 22

1.6　拆分计算阶段与格式化阶段 24

1.7　进展：分离到两个文件（和两个阶段） 31

1.8　按类型重组计算过程 34

1.9　进展：使用多态计算器来提供数据 41

1.10　结语 43

第 2 章　重构的原则 45

2.1　何谓重构 45

2.2　两顶帽子 46

2.3　为何重构 47

2.4　何时重构 50

2.5　重构的挑战 55

2.6　重构、架构和YAGNI 62

2.7　重构与软件开发过程 63

2.8　重构与性能 64

2.9　重构起源何处 67

2.10　自动化重构 68

2.11　延展阅读 70

第3 章　代码的坏味道 71

3.1　神秘命名（Mysterious Name） 72

3.2　重复代码（Duplicated Code） 72

3.3　过长函数（Long Function） 73

3.4　过长参数列表（Long Parameter List） 74

3.5　全局数据（Global Data） 74

3.6　可变数据（Mutable Data） 75

3.7　发散式变化（Divergent Change） 76

3.8　霰弹式修改（Shotgun Surgery） 76

3.9　依恋情结（Feature Envy） 77

3.10　数据泥团（Data Clumps） 78

3.11　基本类型偏执（Primitive Obsession） 78

3.12　重复的switch（Repeated Switches） 79

3.13　循环语句（Loops） 79

3.14　冗赘的元素（Lazy Element） 80

3.15　夸夸其谈通用性（Speculative Generality） 80

3.16　临时字段（Temporary Field） 80

3.17　过长的消息链（Message Chains） 81

3.18　中间人（Middle Man） 81

3.19　内幕交易（Insider Trading） 82

3.20　过大的类（Large Class） 82

3.21　异曲同工的类（Alternative Classes with Different Interfaces） 83

3.22　纯数据类（Data Class） 83

3.23　被拒绝的遗赠（Refused Bequest） 83

3.24　注释（Comments） 84

第4 章　构筑测试体系 85

4.1　自测试代码的价值 85

4.2　待测试的示例代码 87

4.3　第 一个测试 90

4.4　再添加一个测试 93

4.5　修改测试夹具 95

4.6　探测边界条件 96

4.7　测试远不止如此 99

第5 章　介绍重构名录 101

5.1　重构的记录格式 101

5.2　挑选重构的依据 102

第6 章　第 一组重构 105

6.1　提炼函数（Extract Function） 106

6.2　内联函数（Inline Function） 115

6.3　提炼变量（Extract Variable） 119

6.4　内联变量（Inline Variable） 123

6.5　改变函数声明（Change Function Declaration） 124

6.6　封装变量（Encapsulate Variable） 132

6.7　变量改名（Rename Variable） 137

6.8　引入参数对象（Introduce Parameter Object） 140

6.9　函数组合成类（Combine Functions into Class） 144

6.10　函数组合成变换（Combine Functions into Transform） 149

6.11　拆分阶段（Split Phase） 154

第7 章　封装 161

7.1　封装记录（Encapsulate Record） 162

7.2　封装集合（Encapsulate Collection） 170

7.3　以对象取代基本类型（Replace Primitive with Object） 174

7.4　以查询取代临时变量（Replace Temp with Query） 178

7.5　提炼类（Extract Class） 182

7.6　内联类（Inline Class） 186

7.7　隐藏委托关系（Hide Delegate） 189

7.8　移除中间人（Remove Middle Man） 192

7.9　替换算法（Substitute Algorithm） 195

第8 章　搬移特性 197

8.1　搬移函数（Move Function） 198

8.2　搬移字段（Move Field） 207

8.3　搬移语句到函数（Move Statements into Function） 213

8.4　搬移语句到调用者（Move Statements to Callers） 217

8.5　以函数调用取代内联代码（Replace Inline Code with Function Call） 222

8.6　移动语句（Slide Statements） 223

8.7　拆分循环（Split Loop） 227

8.8　以管道取代循环（Replace Loop with Pipeline） 231

8.9　移除死代码（Remove Dead Code） 237

第9 章　重新组织数据 239

9.1　拆分变量（Split Variable） 240

9.2　字段改名（Rename Field） 244

9.3　以查询取代派生变量（Replace Derived Variable with Query） 248

9.4　将引用对象改为值对象（Change Reference to Value） 252

9.5　将值对象改为引用对象（Change Value to Reference） 256

第 10 章　简化条件逻辑 259

10.1　分解条件表达式（Decompose Conditional） 260

10.2　合并条件表达式（Consolidate Conditional Expression） 263

10.3　以卫语句取代嵌套条件表达式（Replace Nested Conditional with Guard Clauses） 266

10.4　以多态取代条件表达式（Replace Conditional with Polymorphism） 272

10.5　引入特例（Introduce Special Case） 289

10.6　引入断言（Introduce Assertion） 302

第 11 章　重构API 305

11.1　将查询函数和修改函数分离（Separate Query from Modifier） 306

11.2　函数参数化（Parameterize Function） 310

11.3　移除标记参数（Remove Flag Argument） 314

11.4　保持对象完整（Preserve Whole Object） 319

11.5　以查询取代参数（Replace Parameter with Query） 324

11.6　以参数取代查询（Replace Query with Parameter） 327

11.7　移除设值函数（Remove Setting Method） 331

11.8　以工厂函数取代构造函数（Replace Constructor with Factory Function） 334

11.9　以命令取代函数（Replace Function with Command） 337

11.10　以函数取代命令（Replace Command with Function） 344

第 12 章　处理继承关系 349

12.1　函数上移（Pull Up Method） 350

12.2　字段上移（Pull Up Field） 353

12.3　构造函数本体上移（Pull Up Constructor Body） 355

12.4　函数下移（Push Down Method） 359

12.5　字段下移（Push Down Field） 361

12.6　以子类取代类型码（Replace Type Code with Subclasses） 362

12.7　移除子类（Remove Subclass） 369

12.8　提炼超类（Extract Superclass） 375

12.9　折叠继承体系（Collapse Hierarchy） 380

12.10　以委托取代子类（Replace Subclass with Delegate） 381

12.11　以委托取代超类（Replace Superclass with Delegate） 399

## 序言

不得不遗憾地说，尽管“重构”已经成了常用词汇，但重构技术井没有像我当初乐观认为的那样“变得像空气与水一样普通”。一方面，一种甚嚣尘的观点认为只要掌握重构的思想就足够了，没必要记住那些详细琐碎的重构手法;另一方面，倒是有很多人高擎“重构”大旗，刀劈斧砍进行着令人触目惊心的人胆修改—有些就是在重做整个系统。

## 他人笔记

https://www.cnblogs.com/lixianyuan-org/p/9664695.html

如果你发现自己需要为程序添加一个特性，而代码结构是你无法很方便地达成目的，那就先重构那个程序，是特性的添加比较容易进行，然后再添加特性。  
重构的原则：  
    重构：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。ACTION-为即将修改的代码建立一组可靠的测试环境  
    重构：使用一系列重构手法，在不改变软件可观察行为的前提下，调整其结构。

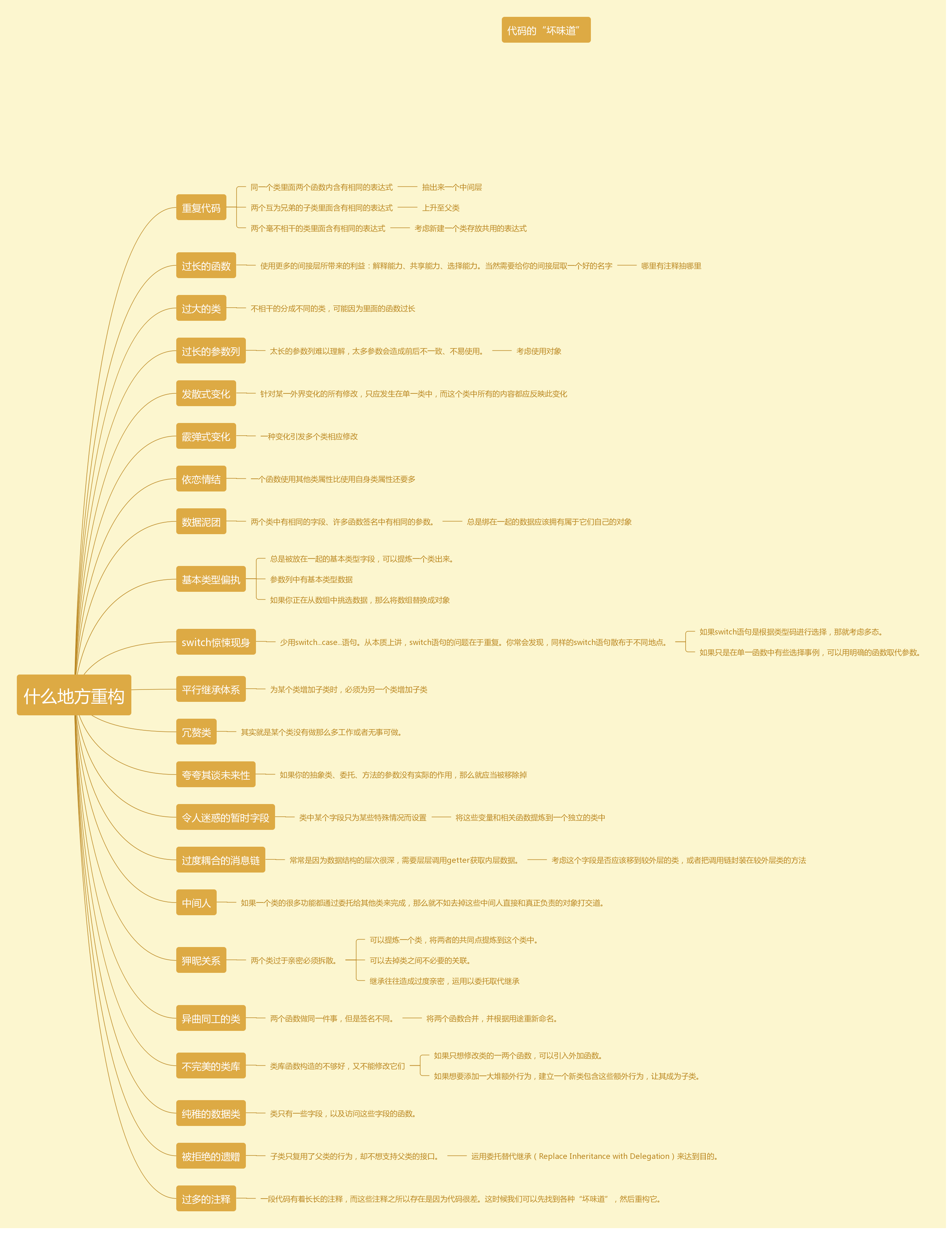
重构的关键在于运用大量微小且保持软件行为的步骤，一步一步达成大规模的修改。

每一次的重构要么很小，要么包含了若干个小步骤，即使重构没有完成，也应当可以在任何时刻停下来，所以如果有人说它们的代码在重构过程中有一两天时间不可用，基本上可以确定，他们做的事不是重构。  
  
与性能优化的区别  
  
重构与性能优化有很多相似的地方：两者都需要修改代码，并且两者都不会改变程序的整体功能。  
  
两者的差别在于起目的：  
\* 重构是为了让代码 “更容易理解，更容易修改”。这可能使程序运行得更快，也可能使程序运行的更慢。   
\* 性能优化则只关心程序是否运行的更快。对于最终得到的代码是否容易理解和维护就不知道了。   
为什么重构？  
重构不是包治百病的灵丹妙药，也绝对不是所谓的“银弹”。重构只是一种工具，能够帮助你始终良好的控制代码而已。使用它，可能基于下面的几个目的。

重构时你就不能再添加功能，只管改进程序结构。此时你不应该添加任何测试(除非发现有先前遗漏的东西)，只在绝对必要(用以处理接口变化)时才修改测试。

添加新功能时，你不应该修改既有代码，只管添加新功能(开-闭原则)。  
    为何重构？  
        重构改进软件设计-设计模式  
        重构增强软件可读性

   重构提高编程速度，提升维护效率      
    何时重构？  
        三次法则：第一次做某件事时只管去做；第二次做类似的事会产生反感，担误落如何还是可以去做；第三次再做类似的事，你就应该重构。也就是事不过三，三则重构。  
     何时不该重构：  
     例如当你应该重新编写所有代码的时候，有时候既有代码实在太混乱，重构它还不如重新写一个来得简单。  
        重写(而非重构)的一个清楚讯号就是：现有代码根本不能正常运作。你可能只是试着做点测试，然后就发现代码中满是错误，根本无法稳定运作。记住，重构之前，代码必须起码能够在大部分情况下正常运作。          
      另外，如果项目已近最后期限，你也应该避免重构。在此事件，从重构过程赢得的生产力只有在最后期限过后才能体现出来，而那个时候已经为时晚矣。 重构确实能够提高生产力，如果最后你没有足够时间，通常就表示你其实早该进行重构。  
    重构肩负一项特殊使命它和设计彼此互补。  
  
代码的坏味道：  
    Duplicated Code(重复代码)  
    Long Method(过长函数)  
        条件表达式和循环常常也是提炼的信号。  
    Large Class(过大的类)  
    Long Parameter List(过长参数列)  
    Divergent Change(发散式变化)  
        一个类受多种变化的影响  
    Shotgun Surgery  
        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。  
      
    Feature Envy(依恋情绪)  
    Data Clumps(数据泥团)  
        减少字段和参数的个数  
    Primitive Obsession(基本类型偏执)  
    Switch Statement  
        面向对象程序的一个最明显的特征就是:少用switch(或case)语句。  
    Parallel Inheritance Hierarchies(平行继承体系)  
        如果每当你为某个类增加一个子类，必须也为另一个类增加一个子类，如果你发现某个继承体系的类的名称前缀和另一个继承体系的类名称前缀完全相同，便是问到了这种坏味道。消除这种重复性的一般策略是：让一个继承体系的实例引用另一个继承体系的实例。  
    Lazy Class(冗赘类)  
            如果一个类的所得不值其身价，它就应该消失。  
    Speculative Generality(夸夸其谈未来性)  
    Temporary Field(令人迷惑的暂时字段)  
    Message Chains(过度耦合的消息链)  
    Middle Man(中间人)  
        不要过度运用委托  
    Inappropriate Intimacy  
    Alternative Classes With Different Interfaces(异曲同工的类)  
    Incomplete Library Class(不完美的类库)  
    Data Class(纯稚的数据类)  
        所谓Data Class是指：他们拥有一些字段，以及用于访问(读写)这些字段的函数，除此之外一无长物。这样的类知识一种不会说话的数据容器，他们几乎一定被其他类过分细琐地操控者，应该考虑把它们封装起来。  
    Refused Bequest(被拒绝的遗赠)  
        如果子类复用了超类的行为(实现)，却又不愿意支持超类的接口，Refused Bequest的坏味道就会变得浓烈。  
    Comments(过多的注释)  
        当你感觉需要写注释时，清闲尝试重构，试着让所有注释都变得多余。

[](https://www.processon.com/mindmap/5c203336e4b0e83682ecb557)  
  
构筑测试体系：  
        一套测试就是一个强大的bug侦测器，能够大大缩减查找bug所需要的时间。  
        Junit测试框架(用途是单元测试)  
        频繁地运行测试，每次编译请把测试页考虑进去-每天至少执行每个测试一次。  
        单元测试/功能测试。功能测试用来保证软件能够正常运作。  
        当你收到bug报告，请先写一个单元测试来暴露bug。  
        编写未臻完善的测试并实际运行，好过对完美测试的无尽等待。  
        考虑可能出错的边界条件，把测试火力集中在那儿。  
        当事情被认为应该会出错时，别忘了检查是否抛出了预期的异常。  
        不要因为测试无法捕捉到所有bug就不写测试，因为测试的确可以捕捉到大多数bug。

   重构列表：  
    每个重构手法都有如下五个部分：  
        首先是名称(name). 建造一个重构词汇表，名称是很重要的。  
        名称之后是一个简短概要。  
        动机为你介绍为什么需要这个重构和什么情况下不该使用这个重构。  
        做法简明扼要地一步一步介绍如何进行此一重构  
        范例以一个十分简单的例子说明此重构手法如何运作。  
          
重新组织函数：  
    Extract Method(提炼函数)  
        首先，如果每个函数的粒度都很小，那么函数被复用的机会就更大；其次，这会使高层函数读起来就像一系列注释；  
    函数的命名：以它“做什么”来命名，而不是以它“怎样做”命名。  
        即使你想要提炼的代码非常简单，例如只是一条消息或一个函数调用，只要心函数的名称能够以更好方式昭示代码意图，你也应该提炼它。但如果你想不出一个更有意义的名称，就别动。  
      
    分解临时变量：  
        如果程序有某个临时变量被赋值超过一次，他既不是循环变量，也不被用于收集计算结果。针对每次赋值，创造一个独立、对应的临时变量。同一个临时变量承担超过一件事情的时候，会令代码阅读者模糊。  
      
    移除对参数的赋值：  
            做法：  
                建立一个临时变量，把待处理的参数值赋予它；以“对参数的赋值”为界，将其后所有对此参数的引用点，全部替换为“对此临时变量的引用”；修改复制语句，使其改为对新建之临时变量赋值。(如果代码的语义是按引用传递的，在调用端检查调用后是否还使用了这个参数，页要检查有多少个按引用传递的参数被赋值后又被使用，请尽量以return方式返回一个值)。还可以为参数加上关键字final，从而强制他遵循“不对参数赋值”  
      
简化条件表达式  
    分解条件表达式 合并条件表达式  合并重复的条件片段

简化函数调用

# 重构流程

需要有对应的流程么？

重构是在已有的代码添加功能，但是要结合实际情况的，在已有的模式下修改功能

1. 梳理代码逻辑，需求分析
2. 概要设计：设计架构图，测试架构，小demo验证完善
3. 详细设计：
4. 编码实施：测试和开发架构，业务功能
5. 测试用例