重构--改善既有代码

# 什么是重构？

所谓重构是这样一个过程：在不改变代码外在行为的前提下，对代码作出修改，以改进程序的内部结构。本质上说，重构就是在代码写好之后改进它的设计。

**重构（名词）：**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（动词）：**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。

重构的目的是使软件更容易被理解和修改。重构不会改变软件可观察的行为——重构之后软件功能一如既往。重构技术就是以微小的步伐修改程序，如果你犯下错误，很容易便可以发现它。

# 为何重构？

（1） 重构改进软件设计。如果没有重构，程序的设计会逐渐腐败变质。重构很像是在整理代码，你所做的就是让所有东西回到应出的位置上。经常性的重构可以帮助维持自己该有的形态。

（2）重构使软件更容易理解。重构可以帮助我们让代码更易读。

（3）重构帮助找到bug。对代码进行重构，可以帮助我们深入理解代码，对代码理解的越深，就越能帮我们找到bug。重构能够帮我们更有效地写出强健的代码。

（4）重构提高变成速度。重构可以帮助我们更快速地开发软件，因为它阻止系统腐败变质，它甚至还可以提高设计质量。

# 何时重构？

重构不是一件应该特别拨出时间做的事情，重构应该随时随地进行。不应该为重构而重构，之所以重构，是因为我们想做别的什么事，而重构可以帮助我们把那些事做好。

三次法则：事不过三，三则重构。

（1）添加功能时重构。

（2）修补错误时重构。

（3）复审代码时重构。

# 何时不该重构？

代码根本无法工作或者太糟糕，重构还不如重写来的简单。

在项目的最后期限，应该避免重构。

# 代码的坏味道

|  |
| --- |
| 重复代码(Duplicated Code)  过长函数(Long Method)  过大的类(Large Class)  过长参数列(Long Parameter List)  发散式变化(Divergent Change)：一个类受多种变化的影响  霰弹式修改(Shotgun Surgery)：一种变化引发多个类相应修改  依恋情结(Feature Envy)：函数对某个类的兴趣高过自己所处类的兴趣  数据泥团(Data Clumps)：相同的若干项数据出现在不同地方，这些绑在一起出现的数据应该有属于它们自己的对象  基本类型偏执(Private Obsession)：很多人不愿意在小任务上运用小对象  switch惊悚现身(Switch Statements)：switch语句会在很多地方重复出现，一改则需全改  平行继承体系(Parallel Inheritance Hierarchies)：当你为某一个类增加子类时，也必须为另一个类相应增加一个类  冗赘类(Lazy Class)：如果一个类不值得存在，那就让它消失  夸夸其谈的未来星(Speculative Generality)：预留的无用的抽象类，无用的抽象参数  令人迷惑的暂时字段(Temporary Field)：类中某个字段只为某些特殊情况而设置  过度耦合的消息链(Message Chains)：用户向一个对象请求另一个对象，然后再向后者请求另一个对象......  中间人(Middle Man)：无用的委托，过多的中间层  狎昵关系(Inappropriate Intimacy)：两个类过于亲密，一个类过于关注另一个类的成员  异曲同工的类(Alternative Classes with Different Interfaces)：不同名字的类或函数，做相同的事  不完美的库类(Incomplete Library Class)：类库设计不可能完美  纯数据类(Data Class)：一个类拥有一些字段以及用于访问这些字段的函数，除此之外一无长物  被拒绝的遗赠(Refused Bequest)：子类不想继承超类所有的函数和数据，只想挑几样来玩  过多的注释(Comments) |

# 构筑测试体系

重构的首要前提是拥有一个可靠的测试环境。只要写好一点功能，就立即添加测试，并确保所有测试都完全自动化，让它们检查自己的测试结果。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。

撰写测试代码的最有用时机是在开始编程之前。当你需要添加特性的时候，先写相应测试代码。编写测试代码其实就是在问自己：添加这个功能需要做些什么。编写测试代码还能使你把注意力集中于接口而非实现。预先写好的测试代码也为你的工作安上一个明确的结束标志：一旦测试代码正常运行，工作就可以结束了。多运用单元测试。测试你最担心出错的地方，考虑可能出错的边界条件。不要因为测试无法捕捉所有bug就不写测试，因为测试的确可以捕捉到大多数bug。"花合理时间抓出大多数bug"**要好过**"穷尽一生抓出所有bug"。

# 重新组织函数

**提炼函数**（Extract Method）。你有一段代码可以被组织在一起并独立出来。将这段代码放进一个独立函数中，并将函数名称解释该函数的用途。

|  |
| --- |
| public void PrintOwing(double amount) {  printBanner();  printDetails(amount);  }    private void printDetails(double amount) {  System.err.println("name: " + \_name);  System.err.println("amount: " + amount);  } |

**内联函数**（Inline Method）。一个函数的本体与名称同样清楚易懂。在函数调用点插入函数本体，然后移除该函数。

|  |
| --- |
| /**/重构前**  public int getRating(){  return (moreThanSixLateDeliveries()) ? 2 : 1;  }    boolean moreThanSixLateDeliveries(){  return \_numberOfLateDeliveries > 6;  }  **//重构后**  public int getRating(){  return (\_numberOfLateDeliveries > 6) ? 2 : 1;  } |

**内联临时变量**（Inline Temp）。你有一个临时变量，只被一个简单表达式赋值一次，而它妨碍了其他重构手法。将所有对该变量的引用动作，替换为对它赋值的那个表达式自身。

|  |
| --- |
| **//重构前**  double basePrice = singleOrder.basePrice();  return (basePrice > 1000)  **//重构后**  return (singleOrder.basePrice() > 1000)  #注意对临时变量设置final,找到所有的修改临时变量的地方。 |

**以查询取代临时变量**（Replace Temp with Query）。你的程序以一个临时变量保存某一表达式的运算结果。将这个表达式提炼到一个独立函数中。将这个临时变量的所有引用点替换为对新函数的调用。此后，新函数就可被其他函数使用。

|  |
| --- |
| **//重构前**  double basePrice = \_quantity \* \_itemPrice;  if(basePrice > 5000)  return basePrice \* 0.95;  else  return basePrice \* 0.98;  **//重构后**  if(basePrice() > 5000)  return basePrice() \* 0.95;  else  return basePrice() \* 0.98;  ...    double basePrice(){  return \_quantity \* \_itemPrice;  } |

**引入解释性变量**（Introduce Explaining Variable）。你有一个复杂的表达式。将该复杂表达式（或其中一部分）的结果放进一个临时变量，以此变量名称来解释表达式用途。

|  |
| --- |
| **//重构前**  if((platform.toUpperCase().indexOf("MAC") > -1)　&&  (browser.toUpperCase().indexOf("IE") > -1) &&  wasInitialized() && resize > 0)  {  //do something  }  **//重构后**  final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;  final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;  final boolean wasResize = resize > 0;    if(isMacOs &&　isIEBrowser　&& wasInitialized() && wasResize){  //do something  } |

**分解临时变量**（Split Temporary Variable）。你的程序有某个临时变量被赋值过一次，它既不是循环变量，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。

|  |
| --- |
| /**/重构前**  double temp = 2 \* (\_height + \_width);  System.out.println(temp);  temp = \_height + \_width;  System.out.println(temp);  **//重构后**  final double perimeter = 2 \* (\_height + \_width);  System.out.println(perimeter);  final double area = \_height + \_width;  System.out.println(area); |

**移除对参数的赋值**（Remove Assignments Parameters）。代码对一个参数进行赋值。以一个临时变量取代参数的位置。

|  |
| --- |
| **//重构前**  int dicount(int inputVal, int quantity, int yearToDate){  if(inputVal > 50) inputVal-=10;  }  **//重构后**  int dicount(final int inputVal, int quantity, int yearToDate){  int result = inputVal;  if(result > 50) result-=10;  } |

**以函数对象取代函数**（Replace Method with Method Object）。你有一个大型函数，其中对局部变量的使用使你无法采用Extract Method。将这个函数放进一个单独对象中，如此一来局部变量就成了对象内的字段。然后你可以在同一个对象中将这个大型函数分解为多个小型函数。

|  |
| --- |
| **//重构前**  class Order....  double price(){  double basePrice;  double secondaryPrice;  double thirdaryPrice;  //compute()  ......  }  **//重构后**  class Order...  double price(){  return new PriceCalculator(this).compute();  }    class PriceCalculator{  double basePrice;  double secondaryPrice;  double thirdaryPrice;    double compute(){  //...  }  } |

**替换算法**（Substitute Algorithm）。你想要把某个算法替换为另一个更清晰的算法。将函数本体替换为另一个算法。

# 在对象之间搬移特性

**搬移函数**（Move Method）。你的程序中，有个函数与其所驻之外的另一个类进行更多交流：调用后者，或被后者调用。在该函数最常引用的类中建立一个有着类似行为的新函数。将旧函数变成一个单纯的委托函数，或是将旧函数完全移除。

**搬移字段**（Move Field）。你的程序中，某个字段被其所驻类之外的另一个类更多地用到。在目标类新建一个字段，修改源字段的所有用户，令它们改用新字段。

**提炼类**（Extract Class）。某个类做了应该有两个类做的事。建立一个新类，将相关的字段和函数从旧类搬移到新类。

**将类内联化**（Inline Class）。某个类没有做太多事情。将这个类的所有特性搬移到另一个类中，然后移除原类。

**隐藏"委托关系"**（Hide Delegate）。客户通过一个委托来调用另一个对象。在服务类上建立客户所需的所有函数，用以隐藏委托关系。

**移除中间人**（Remove Middle Man）。某个类做了过多的简单委托动作。让客户直接调用受托类。

**引入外加函数**（Introduce Foreign Method）。你需要为提供服务的类增加一个函数，但你无法修改这个类。在客户类中建立一个函数，并以第一参数形式传入一个服务类实例。

|  |
| --- |
| **//before**  Date newStart = new Date(previous.getYear(),  previous.getYear(),previous.getYear() + 1);  **//after**  Date newStart = nextDay(previous);    private static Date nextDay(Date date){  //......  return new Date(date.getYear(),date.getMonth(),date.getDate() + 1)  } |

**引入本地扩展**（Introduce Local Extension）。你需要为服务类提供一些额外函数，但你无法修改这个类。建立一个新类，使它包含这些额外函数。让这个扩展品成为源类的子类或包装类。

# 重新组织数据

**自封装字段**（Self Encapsulate Field）。你直接访问一个字段，但与字段之间的耦合关系逐渐变得笨拙。为这个字段建立取值/设值函数，并且只以这些函数来访问字段。

**以对象取代数据值**（Replace Data Value with Object）。你有一个数据项，需要与其他数据和行为一起使用才有意义。将数据项变成对象。

**将值对象改为引用对象**（Change Value to Reference）。你从一个类衍生出许多彼此相等的实例，希望将它们替换为同一个对象。将这个值对象变成引用对象。

**将引用对象改为值对象**（Change Reference to Value）。你有一个引用对象，很小且不可变，而且不易管理。将它变成一个值对象。

**以对象取代数组**（Replace Array with Object）。你有一个数组，其中的元素各自代表不同的东西。以对象替换数组，对于数组中的每个元素，以一个字段来表示。

**复制"被监视数据**"（Duplicate Observed Data）。你有一些领域数据置身GUI控件中，而领域函数需要访问这些数据。将该数据复制到一个领域对象中。建立一个Observe模式，用以同步领域对象和GUI对象内的重复数据。

**将单向关联改为双向关联**（Change Unidirectional Association to Bidirectional）。两个类都需要使用对方特性，但其间只有一条单向链接。添加一个反向指针，并使修改函数能够同时更新两条链接。

**将双向关联改为单向关联**（Change Bidirectional Association to Unidirectional）。两个类之间有双向关联，但其中一个类如今不再需要另一个类的特性。去除不必要的关联。

**以字面常量取代魔法数**（Replace Magic Number with Symbolic Constant）。你有一个字面数值，带有特别含义。创造一个常量，根据其意义为它命名，并将上述的字面数值替换为这个常量。

|  |
| --- |
| //**重构前**  double potentialEnergy(double mass, double height) {  return mass \* 9.81 \* height;  }  //**重构后**  static final double GRAVITATIONAL\_CONSTANT = 9.81;  double potentialEnergy(double mass, double height) {  return mass \* GRAVITATIONAL\_CONSTANT \* height;  } |

**封装字段**（Encapsulate Field）。你的类中存在一个public字段。将它声明为private，并提供相应的访问函数。

**封装集合**（Encapsulate Collection）。有个函数返回一个集合。让这个函数返回该集合的一个只读副本，并在这个类中提供添加/移除集合元素的函数。

**以数据类取代记录**（Replace Record with Data Class）。你需要面对传统编程环境中的记录结构。为该记录创建一个"哑"数据对象。

**以类取代类型码**（Replace Type Code with Class）。类之中有一个数值类行码，但它并不影响类的行为。以一个新的类替换该数值类型码。(**现在应该使用枚举**)

**以子类取代类型码**（Replace Type Code with Subclass）。你又一个不可变的类型码，它会影响类的行为。以子类取代这个类型码。

**以State/Strategy取代类型码**（Replace Type Code with State/Strategy）。你有一个类型码，它会影响类的行为，但你无法通过继承手法消除它。以状态对象取代类型码。

**以字段取代子类**（Replace Subclass with Fields）。你的各个子类的唯一差别只在"返回常量数据"的函数身上。修改这些函数，使他么返回超类中的某个（新增）字段，然后销毁子类。（**行为基本不同则需要泛化**）

**注意**：如果类型码对应的对象拥有很多不同的属性和行为，则应该使用泛化；否则，直接定义为类型属性，并创建相应枚举。

# 简化条件表达式

**分解条件表达式**（Decompose Conditional）。你有一个复杂的条件（if-then-else）语句。从if、then、else三分段落中分别提炼出独立函数。

|  |
| --- |
| //**重构前**  if (date.before (SUMMER\_START) || date.after(SUMMER\_END)){  charge = quantity \* \_winterRate + \_winterServiceCharge;  } else{  charge = quantity \* \_summerRate;  }  //**重构后**  if (notSummer(date)){  charge = winterCharge(quantity);  }else{  charge = summerCharge (quantity);  }  private boolean notSummer(Date date) {  return date.before (SUMMER\_START) || date.after(SUMMER\_END);  }  private double summerCharge(int quantity) {  return quantity \* \_summerRate;  }  private double winterCharge(int quantity) {  return quantity \* \_winterRate + \_winterServiceCharge;  } |

**合并条件表达式**（Consolidate Conditional Expression）。你有一系列条件测试，都得到相同结果。将这些测试合并为一个条件表达式，并将这个条件表达式提炼成为一个独立函数。

**合并重复的条件片段**（Consolidate Duplicate Conditional Fragments）。在条件表达式的每个分支上有着相同的一段代码。将这段重复的代码搬移到条件表达式之外。

**移除控制标记**（Remove Control Flag）。在一系列布尔表达式中，某个变量带有"控制标记"的作用。以break语句或return语句取代控制标记。

**以卫语句取代嵌套条件表达式**（Replace nested Conditional with Guard Clauses）。函数中的条件逻辑使人难以看清正常的执行路径。使用卫语句表现所有的特殊情况。

|  |
| --- |
| //**重构前**  double getPayAmount() {  double result;  if (\_isDead) result = deadAmount();  else {  if (\_isSeparated) result = separatedAmount();  else {  if (\_isRetired) result = retiredAmount();  else result = normalPayAmount();  };  }  return result;  };      //**重构后**  double getPayAmount() {  if (\_isDead) return deadAmount();  if (\_isSeparated) return separatedAmount();  if (\_isRetired) return retiredAmount();  return normalPayAmount();  }; |

**以多态取代条件表达式**（Replace Conditional with Polymorphism）。你手上有个条件表达式，它根据对象类型的不同选择不同的行为。将这个条件表达式的每个分支放进一个子类内的覆写函数中，然后将原始函数声明为抽象函数。

**引入Null对象**（Introduce Null Object）。你需要再三检查某对象是否为null。将null值替换为null对象。

**引入断言**（Introduce Assertion）。某一段代码需要对程序状态做出某种假设。以断言明确表现这种假设。

# 简化函数调用

**函数改名**（Rename Method）。函数的名称未能揭示函数的用途。修改函数的名称。

**添加参数**（Add Parameter）。某个函数需要从调用端得到更多信息。为此函数添加一个对象参数，让该对象带进函数所需信息。

**移除参数**（Remove Parameter）。函数本体不再需要某个参数。将该参数去除。

**将查询函数和修改函数分离**（Separate Query from Modifier）。某个函数既返回对象状态值，又修改对象状态。建立两个不同的函数，其中一个负责查询，另一个负责修改。

*ps*：函数按功能进行拆分。

**令函数携带参数**（Parameterize Method）。若干函数做了类似的工作，但在函数本体中却包含了不同的值。建立单一函数，以参数表达那些不同的值。

*ps*：如果参数对函数影响很多，使用**以明确函数取代参数**重构

**以明确函数取代参数**（Replace Parameter with Explicit Methods）。你有一个函数，其中完全取决于参数值而采取不同行为。针对该参数的每一个可能值，建立一个独立函数。

|  |
| --- |
| **//重构前**  static final int ENGINEER = 0;  static final int SALESMAN = 1;  static final int MANAGER = 2;  static Employee create(int type){  switch(type){  case ENGINEER:  return new Engineer();  case SALESMAN:  return new Salesman();  case MANAGER:  return new Manager();  default:  throw new IllegalArgumentException();  }  }  **//重构后**  static employee createEngineer(){  return new Engineer();  }  static employee createSalesman(){  return new Salesman();  }  static employee createManager(){  return new Manager();  } |

**保持对象完整**（Preserve Whole Object）。你从某个对象中取出若干值，将它们作为某一次函数调用时的参数。改为传递整个对象。

**以函数取代参数**（Replace Parameter with Methods）。对象调用某个函数，并将所得结果作为参数，传递给另一个函数。而接受该参数的函数本身也能够调用前一个函数。让参数接受者去除该项参数，并直接调用前一个函数。

|  |
| --- |
| **//重构前**  public double getPrice(){  int basePrice = \_quantity \* \_itemPrice;  int discountLevel;  if(\_quantity > 100){  discountLevel = 2;  }else{  discountLevel = 1;  }  double final Price = discountedPrice(basePrice, discountLevel);  return finalPrice;  }  private double discountedPrice(int basePrice, int discountLevel){  if(discountLevel == 2){  return basePrice \* 0.1;  }else{  return basePrice \* 0.05;  }  }  **//重构后**  public double getPrice(){  return discountedPrice();  }  private int getDiscountLevel(){  if(\_quantity > 100){  return 2;  }else{  return 1;  }  }  private doulbe discountedPrice(){  if(getDiscountLevle() == 2){  return getBasePrice() \* 0.1;  }else{  return getBasePrice() \* 0.05;  }  }  private double getBasePrice(){  return \_quantity \* \_itemPrice;  } |

**引入参数对象**（Introduce Parameter Object）。某些参数总是很自然地同时出现。以一个对象取代这些参数。

**移除设值函数**（Remove Setting Method）。类中的某个字段应该在对象创建时被设值，然后就不再改变。去掉该字段的所有设值函数。

**隐藏函数**（Hide Method）。有一个函数，从来没有被其他任何类用到。将这个函数修改为private。

**以工厂函数取代构造函数**（Replace Constructor with Factory Method）。你希望在创建对象时不仅仅是做简单的构建动作。将构建函数替换为工厂函数。

**封装向下转型**（Encapsulate Downcast）。某个函数返回的对象，需要由函数调用者执行向下转型。将向下转型动作移到函数中。

*ps*：更改函数返回类型为原类型的子类型的时候，不需要更改客户端调用代码。

**以异常取代错误码**（Replace Error Code with Exception）。某个函数返回一个特定的代码，用以表示某种错误情况。改用异常。

*ps*：异常要尽早抛出。

**以测试取代异常**（Replace Exception with Test）。面对一个调用者可以预先检查的条件，你抛出了一个异常。修改调用者，使它在调用函数之前先做检查。

*ps*：异常不要乱用，性能不较低。

# 处理概括关系

**字段上移**（Pull Up Field）。两个子类拥有相同的字段。将该字段移至超类。

**函数上移**（Pull Up Method）。有些函数，在各个子类中产生完全相同的结果。将该函数移至超类。

**构造函数本体上移**（Pull Up Constructor Body）。你在各个子类中拥有一些构造函数，他们的本体几乎完全一致。在超类中新建一个构造函数，并在子类构造函数中调用它。

**函数下移**（Push Down Method）。超类中的某个函数只与部分（而非全部）子类有关。将这个函数移到相关的那些子类去。

**字段下移**（Push Down Field）。超类中的某个字段只被部分（而非全部）子类用到。将这个字段移到需要它的那些子类去。

**提炼子类**（Extract Subclass）。类中的某些特性只被某些（而非全部）实例用到。新建一个子类，将上面所说的那一部分特性移到子类中。

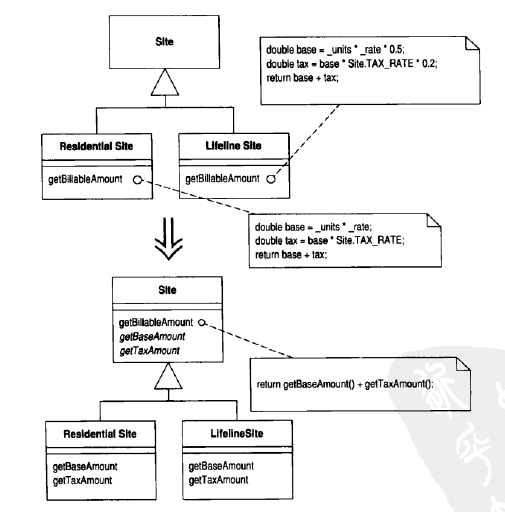
**提炼超类**（Extract Superclass）。两个类有相似特性。为这两个类建立一个超类，将相同特性移至超类。

**提炼接口**（Extract Interface）。若干客户使用类接口中的同一子集，或者两个类的接口有部分相同。将相同的子集提炼到一个独立接口中。

**折叠继承体系**（Collapse Hierarchy）。超类和子类之间无太大差别。将它们合为一体。

**总结**：把公共的属性和方法抽取到父类中，把子类独有的字段和方法定义在子类中。对于超类和子类之间无太大差别，将它们合为一体。

**塑造模板函数**（Form TemPlate Method）。你有一些子类，其中相应的某些函数以相同顺序执行类似的操作，但各个操作的细节上所有不同。将这些操作分别放进独立函数中，并保持它们都有相同的签名，于是原函数也就变得相同了。然后将原函数上移至超类。



ps：子类中具有相同的函数，其中大部分的工作相同。我们可以将公共的部分抽象到父类中，然后子类做具体不同实现。

**以委托取代继承**（Replace Inheritance with Delegation）。**某个子类只使用超类接口中的一部分，或是根本不需要继承而来的数据**。在子类中新建一个字段用以保存超类；调整子类函数令它改而委托超类；然后去掉两者之间的继承关系。

**以继承取代委托**（Replace Delegation with Inheritance）。你在两个类之间使用委托关系，并经常为整个接口编写许多极简单的委托函数。让委托类来继承受托类。

ps：有一个（**委托**）要优于是一个（**继承**）。