1 什么是重构

所谓重构(refactoring)是这样一个过程：在不改变代码外在行为的前提下，对代码做出修改，以改进程序的内部结构。重构是一种经千锤百炼形成的有条不紊的程序整理方法，可以最大限度地减少整理过程中引入错误的概率。本质上说，重构就是在代码写好之后改进它的设计。

“在代码写好之后改进它的设计”这种说法有点儿奇怪。在软件开发的大部分历史时期，大部分人相信应该先设计而后编码：首先得有一个良好得设计，然后才能开始编码。但是，随着时间流逝，人们不断修改代码，于是根据原先设计所得的系统，整体结构逐渐衰弱。代码质量慢慢沉沦，编码工作从严谨的工程堕落为胡砍乱劈的随性行为。

“重构”正好与此相反。哪怕手上有一个糟糕的设计，甚至是一堆混乱的代码，我们也可以借由重构将它加工成设计良好的代码。重构的每个步骤都很简单，甚至显得有些过于简单：只需要把某个字段从一个类移到另一个类，把某些代码从一个函数拉出来构成另一个函数，或是在继承体系中把某些代码推上推下就行了。但是聚沙成塔，这些小小的修改累积起来就可以根本上改善设计质量。这和一般常见的“软件会慢慢腐烂”的观点恰恰相反。

有了重构以后，工作的平衡点开始发生变化。我发现设计不是在一开始完成的，而是在整个开发过程中逐渐浮现出来。在系统构筑过程中，我学会了如何不断改进设计。这个“构筑 - 设计”的反复互动，可以让一个程序在开发过程中持续保有良好的设计。

2 为什么要重构

2.1 重构的必要

一个软件总是为解决某种特定的需求而产生，时代在发展，客户的业务也在发生变化。有的需求相对稳定一些，有的需求变化的比较剧烈，还有的需求已经消失了，或者转化成了别的需求。在这种情况下，软件必须相应的改变。

考虑到成本和时间等因素，当然不是所有的需求变化都要在软件系统中实现。但是总的说来，软件要适应需求的变化，以保持自己的生命力。

这就产生了一种糟糕的现象：软件产品最初制造出来，是经过精心的设计，具有良好架构的。但是随着时间的发展、需求的变化，必须不断的修改原有的功能、追加新的功能，还免不了有一些缺陷需要修改。为了实现变更，不可避免的要违反最初的设计构架。经过一段时间以后，软件的架构就千疮百孔了。bug越来越多，越来越难维护，新的需求越来越难实现，软件的架构对新的需求渐渐的失去支持能力，而是成为一种制约。最后新需求的开发成本会超过开发一个新的软件的成本，这就是这个软件系统的生命走到尽头的时候。

重构就能够最大限度的避免这样一种现象。系统发展到一定阶段后，使用重构的方式，不改变系统的外部功能，只对内部的结构进行重新的整理。通过重构，不断的调整系统的结构，使系统对于需求的变更始终具有较强的适应能力。

重构可以降低项目的耦合度，使项目更加模块化，有利于项目的开发效率和后期的维护。让项目主框架突出鲜明，给人一种思路清晰，一目了然的感觉，其实重构是对框架的一种维护。

如果没有重构，程序会逐渐腐败甚至变质。当我们只为了短期的目的或者在没有完全理解代码之前，就贸然修改代码，程序就会逐渐失去已有的结构，程序员则愈来愈难通过阅读源码来理解原来的设计。重构其实就像是在整理代码一样，你所做的就是让所有东西回到应处的位置上。就像我们收拾屋子一样，如果屋子天天都打扫，那么每天花几分钟就能保持干净；如果屋子一个月不打扫，你可以想想得花多久才能收拾完。代码结构的流失是具有累积性的，愈难看出代码所代表的设计意图，就愈难以保护其中的设计，于是设计就变腐败的愈快。如果能够经常地对代码进行重构，则可以帮助代码维持自己该有的状态。

2.2 重构的原因

1．函数逻辑结构混乱，或因为没注释原因，连原代码写作者都很难理清当中的逻辑。

2.函数无扩展性可言，遇到新的变化，不能灵活的处理。

3.因为对象强耦合或者业务逻辑的原因，导致业务逻辑的代码巨大，维护 的时候排查困难。

4.重复代码太多，没有复用性。

5.随着技术的发展，代码可能也需要使用新特性进行修改。

6.随着学习的深入，对于以前的代码，是否有着更好的一个解决方案。

7.因为代码的写法，虽然功能正常使用，但是性能消耗较多，需要换方案进行优化。

2.3 重构的时机

事不过三，三则重构。当我们第一次做一件事的时只管去做；第二次做类似的事就会产生反感，但无论如何还是可以去做；第三次再做类似的事，你就应该重构。

（1）当我们添加新功能时需要重构。

最常见的重构时机是给软件添加新特性的时候。此时进行重构能够帮助我们理解需要修改的代码——这些代码可能是别人写的，也可能使我们自己写的，无论何时，只要我们能够理解代码所做的事，我们就有必要问自己：是否能对这段代码进行重构，使我能更快地理解它。之所这么做，部分原因是为了在下次看到这段代码时容易理解，但最主要的原因是：如果在前进的过程中把代码结构理清，就可以从中理解更多的东西。

（2）当我们修补错误时需要重构。

调试过程中运用重构，大多是为了让代码更具有可读性。当我们看代码并努力去理解它的时候，我们使用重构帮助加深自己的理解。以这种程序来处理代码，常常能够帮助我们找出bug。可以这样想：如果收到一份错误报告，这就是需要重构的信号，因为代码还不够清晰——没有清晰到一眼就能看出bug。

（3）当我们复审代码时需要重构。

重构可以帮助我们复审别人的代码。在开始重构前，我们可以先阅读代码，有一定程度的理解后，提出一些建议。一旦想到一些点子我们就可以考虑是否可以通过重构立即轻松地实现它们。如果可以，我们应该动手。这样做几次之后，我们把代码看得更清楚，提出更多恰当的建议。我们不必想象代码应该是什么样，我可以“看见”它是什么样子。

2.3 重构的时机

代码并不是在开发完成后就像一潭死水一样，不去使之“流动”和“改变”，而是应该经常性地回过来审视已写代码，对其进行批判和思考，对其进行整理和重构。使之变得更加完美，使之质量有所提高。但是，大多数的程序员根本没有做到经常性地对代码进行审视和评判，以及整理和修改。在没有了解重构之前，我自己也经常那么做的。大体上的做法是，开发任务完成后，就可以拍屁股走人了，等后续有问题再说，没有问题就那样了。这样的做法其实是不好的，首先，体现在对自己所写代码不负责；其次，体现在没有一颗精益求精的上进心；最后，归根结底还是体现在一个“懒”字上 ，认为是在为公司干活，没有必要那么卖力，自己做完就行了，再说公司也没给涨工资......这些想法在我看来都是不理智的。在公司上班干活学技术都是为自己以后的发展做铺垫的，怎么能说和自己无关呢！再说每一次对自己的代码进行修改和优化，本身就是提高逻辑思维能力的过程，怎么能偷懒到不思进取呢！

3 如何重构

3.1 烂代码的味道

3.1.1**重复的代码**

重复的代码是坏味道中出现频率最高的情形非其莫属。如果在一个的以上地方看到相同的代码，那么就可以肯定：想办法将它们合而为一，代码会变得更好。最单纯的重复代码就是“同一个类的两个函数含有相同的表达式”，这时候可以采用抽取方法提炼出重复的代码，然后让这两个地点都调用被提炼出的那一段代码。

另一种常见情况就是“两个互为兄弟的子类内含相同的表达式”，这时候只需对两个类抽取方法，然后将提炼出的代码推入到超类中。如果代码之间只是类似而并非完全相同，那么就需要通过抽取方法将相似部分和差异部分分开，构成单独一个函数。如果有些函数以不同的算法做相同的事，可以使用比较清晰的一个替换掉其余的。

3.1.2**过长的函数**

程序员都喜欢简短的函数。拥有短函数的对象会活的比较好、比较长。不熟悉面向对象技术的人，常常觉得对象程序中只有无穷无尽的委托，根本没有进行任何计算。和此类程序共同生活数年后，你才会知道这些小小函数的价值。

应该积极地分解函数，将长长的函数变为多个短小的函数。一般会遵循这样的原则：每当感觉需要用注释来说明点什么的时候，就把需要说明的东西写进一个独立函数中，并以其用途命名。不要嫌麻烦。可以对一组甚至短短一行代码做这件事，哪怕替换后的函数调用动作比函数自身还长，只要函数名称能够解释其用途，也应毫不犹豫地那么做。关键不在于函数的长度，而在于函数“做什么”和“如何做”之间的语义距离。

3.1.3**过大的类**

如果想利用单个的类做太多的事情，其内往往会出现太多实例变量。一旦如此，重复的代码就接踵而来。

可以将几个变量一起提炼至新类内。提炼时应该选择类内彼此相关的变量，将它们放在一起。通常如果类内的数个变量有着相同的前缀或字尾，这就意味有机会把它们提炼到某个组件内。

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”，它们之中很多代码都相同，那么或许你可以把它们变成五个“十行函数”和十个提炼出的“双行函数”。

3.1.4**过长的参数列**

刚开始学编程的时候，或许都是“把函数所需的所有东西都以参数传递进去”。这样也是可以理解的，因为除此之外就只能选择全局数据，而全局数据是邪恶的东西。对象技术告诉我们，如果你手上没有所需的东西，总可以叫一个对象给你。有了对象，你就不必要把函数所需的所有东西都以参数传递给它，只需传给它足够的、让函数能从中获得自己的东西就行。

太长的的参数列难以理解，太多参数会造成前后不一致、不易使用，而且一旦需要更多数据，就不得不修改它。如果将对象传递给函数，大多数修改都将没有必要，因为很可能只需增加一两条请求，就能得到更多的数据。

3.1.5**发散式变化**

我们希望软件能够容易被修改——毕竟软件再怎么说本来就该是“软”的。一旦需要修改，我们希望能够跳到系统某一点，只在该处做修改。如果不能做到这点，你就会嗅出两种紧密相关的刺鼻味道中的一种。

如果某个类经常因为不同的原因在不同的方向上发生变化，发散式变化就出现了。其主要指“一个类受多种变化的影响”。当你看着一个类说：“呃，如果新加入一个数据库，就必须修改这三个函数；如果新出现一种工具，就必须修改这四个函数。”那么此时也许将这个对象分成两个会更好，这样对每个对象就可以只因一种变化而需要修改。

3.1.6**霾弹式修改**

如果每遇到变化，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霾弹式修改。其主要指“一种变化引发多个类相应修改”。如果需要修改的代码散布四周，不但很难找到它们，也很容易忘记某个重要的修改。

这种情况可以把所有需要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。通常可以运用内联类把一系列相关行为放进同一个类。

3.1.7**依恋情节**

众所周知，对象技术的全部要点在于：其是一种“将数据和对数据的操作行为包装在一起”的技术。有一种经典的气味：函数对于某个类的兴趣高过对自己所处类的兴趣。在很多情况下，都能够看到：某个函数为了计算某个值，从另一个对象那儿调用几乎半打的取值函数。疗法也显而易见：把这个函数移至另一个地点，移到它该去的地方。‘

有时候一个函数往往会用到几个类的功能，那么它究竟该被置于何处呢？处理原则通常为：判断哪个类拥有最多被此函数使用的数据，然后就把这个函数和那些数据摆在一起。

3.1.8**数据泥团**

如果用比较形象的事物来形容数据项，我想“小孩子”是一个不错的选择，数据项就像小孩子，喜欢成群结队地呆在一块儿。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拥有属于它们自己的对象。

这种情况可以先找出这些数据以字段形式出现的地方，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，运用参数对象为它减肥。这样做的直接好处是可以将很多参数列缩短，简化函数调用。一个比较好的评判方法是：删掉众多数据中的一项。这么做其它数据有没有因而失去意义？如果它们不再有意义，这就是一个明确的信号：应该为它们产生一个新对象。

3.1.9**基本类型偏执**

大多数编程环境都有两种数据：结构类型允许你将数据组织成有意义的形式；基本类型则是构成结构类型的积木块。但是请记住：结构总是会带来一定的额外开销。它们可能代表着数据库中的表，如果只为做一两件事而创建结构类型也可能显得很麻烦。

对象的一个极大价值在于：它们模糊甚至打破横亘于基本数据和体积较大的类之间的界限。如果你有一组应该总是被放在一起的字段，可以将其抽取为一个独立的类。如果你在参数列中看到基本型数据，可以引入参数对象进行处理。如果你发现自己正从数组中挑选数据，可以运用以对象取代数组进行处理。

3.1.10 **Switch惊悚现身**

面向对象程序的一个较明显特征是：少用switch语句。从本质上说，switch语句的问题在于重复。你常会发现同样的switch语句散布于不同的地方。如果要为它添加一个新的case语句，就必须找到所用switch语句并修改它们。面向对象中的多态概念可为此带来优雅的解决办法。

大多数时候，一看到switch语句，那就应该考虑以多态来替换它。switch语句常常根据类型码进行选择，你要的是“与该类型码相关的函数或类”，所以应该将switch语句提炼到一个独立函数中，再将它搬移到需要多态性的那个类里。

3.1.11 **平行继承体系**

平行继承体系其实是霾弹式修改的特殊情况。在这种情况下，每当为某个类增加一个子类，必须也为另一个类增加一个子类。如果发现某个继承体系的类名称前缀和另一个继承体系的类名称前缀完全相同，这种坏味道就会被嗅出。

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。

3.1.12 **冗赘类**

你所创建的每一个类，都得有人去理解它、维护它，这些工作都是需要花钱的。如果一个类的所得并不值其身价，他就应该消失。项目中经常会出现这样的情况：某个类原本对得起自己的价值，但重构使它身形缩水，不再做那么多工作；或开发者事先规划了某些变化，并添加一个类来应付这些变化，但变化实际没有发生。

不管是哪种原因，都应该让这个类庄严赴义吧。如果某些子类并没有做足够的工作，我们可以尝试“折叠继承体系”，将超类和子类合为一体，那样就会减少维护时间。对于那些几乎没用的组件，就应该将这个类的所有特性搬移到另一个类中，然后移除原类。

3.1.13 **夸夸其谈未来性**

我们经常会说：“我想总有一天需要做这事”，并因而企图以各样的钩子和特殊情况来处理一些非必要的事情。一旦这样，坏味道就浮现出来了。夸夸其谈未来的结果往往会造成系统更加难以理解和维护。如果所有的装置都被用到了，那就值得那么做；如果用不到，就不值得。用不上的装置只会阻挡你的路，给你添乱，那就搬开它吧。

如果某个抽象类其实没有太大作用，可以将超类和子类合为一体。将不必要的委托转移到另一个类中，并消除原先的类。如果函数的某些参数未被用上，那么就将参数移走。如果函数名称带有多余的抽象意味，就应该对它重命名，让它现实一些。

3.1.14 **令人迷惑的暂时字段**

有时候你会发现：类中的某个实例变量仅为某种特定情况而设。这样的代码让人难以理解，因为你通常认为对象在所有时候都需要它的所有变量。当变量在未被使用的情况下去猜测其当初设置的目的，会让你发疯的。

可以使用提炼新类为这个可怜的孤儿创造一个家，然后把所有和这个变量相关的代码都放进这个新家。

也许还可以使用“将Null值替换为Null对象”在“变量不合法”的情况下创建一个Null对象，从而避免写出条件式代码。

3.1.15 **过度耦合的消息链**

如果你看到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，然后再请求另一个对象.....这就是消息链。这种方式意味着客户代码将与某些功能函数中的导航结构紧密耦合。一旦对象间的关系发生任何变化，客户端就不得不做出相应修改。

这时候我们可以隐藏“委托关系”，并在服务类上建立客户所需要的所有函数。你可以在消息链的不同位置进行这种重构手法。理论上是可以重构消息链上的任何一个对象，但是这样做往往会把一系列对象都变成“中间人”。通常更好的选择是：先观察消息链最终得到的对象是用来干什么的，再看看能否通过抽取方法把使用该对象的代码提炼到一个独立函数中，然后再将这个函数推入消息链。

3.1.16 **中间人**

我们都知道对象的基本特征之一就是封装——对外部世界隐藏其内部细节。封装往往伴随着委托。比如你对Boss说是否有时间参加一个会议，他把这个消息“委托”给他的记事本，然后才能回答你。但是，你没有必要知道Boss到底使用传统记事本或电子记事本亦或秘书来记录自己的约会。

人们可能会过度使用委托。你也许会看到某个类接口中有一半的函数都委托给其它类，这样就是过度委托。这时候就应该移除中间人，直接和真正的负责人打交道。如果这样“不干实事”的函数只有少数几个，可以将它们放进调用端。如果中间人还有其它行为，可以把它变成实责对象的子类，这样你既可以扩展原对象的行为，又不必负担那么多的委托动作。

3.1.17 **狎昵关系**

有时候你会看到两个类过于亲密，花费太多时间去探究彼此的private成分。如果这发生在两个“人”之间，我们无比做卫道士；但对于类，我们就希望它们严守清规。

也许就像古代的恋人一样，过分狎昵的类必须拆散。可以通过“移动方法”和“移动字段”帮它们划清界限，从而减少狎昵行径。如果两个类实在是情投意合，可以把两者共同点提炼到一个安全地点，让它们坦荡地使用这个新类。或者通过隐藏“委托关系”让另一个类来为它们传递相思情。

3.1.18 **异曲同工的类**

如果两个函数做同一件事，却有着不同的签名，可以根据它们的用途重新命名。

但这往往不够，可以反复将某些行为移入类中，直到两者的协议一致为止。

3.1.19 **不完美的库类**

复用常被视为对象的终极目的。不过我们认为复用的意义经常被高估——大多数对象只要够用就好。但是无可否认，许多编程技术都建立在程序库的基础上。库类构建者没有未卜先知的能力，我们不能因此责怪它们。

幸好我们有两个专门应付这种情况的工具。如果你只想修改库类的一两个函数，可以使用“引入外加参数”来进行修改。如果想要添加一大堆额外行为，就得运用“引入本地扩展（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）”来进行修改。

3.1.20 **纯稚的数据类**

纯稚的数据类是指：它们拥有一些字段，以及用于访问（读写）这些字段的函数，除此之外一无长物。这样的类只是一种不会说话的数据容器，它们几乎一定被其它类过分细琐地操控着。

这些类早期可能拥有public字段，果真如此就应该在别人注意到它们之前将它们封装起来。如果这些类内含容器类的字段，就应该检查它们是不是得到了恰当的封装；如果没有，就把它们封装起来。对于那些不该被其它类修改的字段，就应该去掉该字段的所有设值函数。

3.1.21 **被拒绝的遗赠**

子类应该继承超类的函数和数据。但是如果它们不想或者不需要继承，又该怎么办呢？它们得到了所有的礼物，但却只从中挑选几样来玩！

这可能意味着继承体系设计错误。你需要为这个子类新建一个兄弟类，再把所有用不到的函数下推给那个兄弟。这样一来，超类就只持有所有子类共享的东西。你常常会听到这样的建议：所有超类都应该是抽象的。

3.1.22 **过多的注释**

不要担心，并不是说不应该写注释。从嗅觉上说，注释不是一种坏味道，事实上它还是一种香味呢。常常还有这样的情况：你看到一段代码有着长长的注释，然后发现，这些注释之所以存在乃是因为代码很糟糕。这种情况出现次数多的实在令人吃惊。

注释可以带我们找到本文先前提到的各种坏味道。找到坏味道后，我们首先应该以各种重构手法把坏味道去除。完成之后我们发现注释已经变得多余了，因为代码已经清楚说明了一切。

如果你需要注释来解释一块代码做了什么，试试将该代码抽取为一个单独的函数；如果函数已经被提炼出来，但还是需要注释来解释其行为，试着对其重新命名。当你感觉需要撰写注释时，请先尝试重构，试着让所有注释都变得多余。

1.1 概述

用GPS卫星定位技术建立的测量控制网称为GPS控制网。······

1.2 GPS控制网的技术设计

**1.2.1 概述**

徐州市位于······，该市现有稳定的GPS控制网。中国矿业大学位于······

**1.2.2 作业依据**

·······

2 系统相关技术分析

2.1 XML技术

网络技术发展到今天，已经渗透到社会生活的每一个角落。

2 .2 XML概述

XML是一种类似于HTML的标记语言······

参考文献：

[1] 华罗庚，王元.论一致分布与近似分析.中国科学.1973（4）：339-357

[2] 李明.物理学.北京：科学出版社，1977，58-62