代码重构

引

任何一个傻瓜都能写出计算机可以理解的程序，只有写出人类容易理解的程序才是优秀的程序员

一.优点:

1.与其重构,不如尽可能设计好

2.设计并不能预料到所有的情况,代码重构是必须的一个过程,只不过,设计的合理之后需要重构的代码就会很少,很大程度上降低了工作量(其实未必,最大的功效只是其可读性可能更好,代码逻辑可能更为清晰).

3.重新审视自己的代码会有很大的进步,也会发现更好的逻辑,更清晰的思维,可以培养更好地思维习惯.

二.准则:(转自https://baike.baidu.com/item/%E4%BB%A3%E7%A0%81%E9%87%8D%E6%9E%84/7219667?fr=aladdin)

1.·代码中存在重复的代码

程序代码更是不能搞重复建设，如果同一个类中有相同的代码块，请把它提炼成类的一个独立方法，如果不同类中具有相同的代码，请把它提炼成一个新类，永远不要重复代码。

2.过大的类往往是类抽象不合理的结果，类抽象不合理将降低了代码的复用率。

方法是类王国中的诸侯国，诸侯国太大势必动摇中央集权。过长的方法由于包含的逻辑过于复杂，错误机率将直线上升，而可读性则直线下降，类的健壮性很容易被打破。当看到一个过长的方法时，需要想办法将其划分为多个小方法，以便于分而治之。

3.·牵一毛而需要动全身的修改

当你发现修改一个小功能，或增加一个小功能时，就引发一次代码地震，也许是你的设计抽象度不够理想，功能代码太过分散所引起的。

4·类之间需要过多的通讯

A类需要调用B类的过多方法访问B的内部数据，在关系上这两个类显得有点狎昵，可能这两个类本应该在一起，而不应该分家。

5·过度耦合的信息链

"计算机是这样一门科学，它相信可以通过添加一个中间层解决任何问题"，所以往往中间层会被过多地追加到程序中。如果你在代码中看到需要获取一个信息，需要一个类的方法调用另一个类的方法，层层挂接，就象输油管一样节节相连。这往往是因为衔接层太多造成的，需要查看就否有可移除的中间层，或是否可以提供更直接的调用方法。

6·各立山头干革命

如果你发现有两个类或两个方法虽然命名不同但却拥有相似或相同的功能，你会发现往往是因为开发团队协调不够造成的。笔者曾经写了一个颇好用的字符串处理类，但因为没有及时通告团队其他人员，后来发现项目中居然有三个字符串处理类。革命资源是珍贵的，我们不应各立山头干革命。

7·不完美的设计

在笔者刚完成的一个比对报警项目中，曾安排阿朱开发报警模块，即通过Socket向指定的短信平台、语音平台及客户端报警器插件发送报警报文信息，阿朱出色地完成了这项任务。后来用户又提出了实时比对的需求，即要求第三方系统以报文形式向比对报警系统发送请求，比对报警系统接收并响应这个请求。这又需要用到Socket报文通讯，由于原来的设计没有将报文通讯模块独立出来，所以无法复用阿朱开发的代码。后来我及时调整了这个设计，新增了一个报文收发模块，使系统所有的对外通讯都复用这个模块，系统的整体设计也显得更加合理。

每个系统都或多或少存在不完美的设计，刚开始可能注意不到，到后来才会慢慢凸显出来，此时唯有勇于更改才是最好的出路。

8·缺少必要的注释

虽然许多软件工程的书籍常提醒程序员需要防止过多注释，但这个担心好象并没有什么必要。往往程序员更感兴趣的是功能实现而非代码注释，因为前者更能带来成就感，所以代码注释往往不是过多而是过少，过于简单。人的记忆曲线下降的坡度是陡得吓人的，当过了一段时间后再回头补注释时，很容易发生"提笔忘字，愈言且止"的情形。

曾在网上看到过微软的代码注释，其详尽程度让人叹为观止，也从中体悟到了微软成功的一个经验。

重构（Refactoring）的难题

学习一种可以大幅提高生产力的新技术时，你总是难以察觉其不适用的场合。通常你在一个特定场景中学习它，这个场景往往是个项目。这种情况下你很难看出什么会造成这种新技术成效不彰或甚至形成危害。十年前，对象技术（object

tech.）的情况也是如此。那时如果有人问我「何时不要使用对象」，我很难回答。并非我认为对象十全十美、没有局限性 — 我最反对这种盲目态度，而是尽管我知道它的好处，但确实不知道其局限性在哪儿。

现在，重构的处境也是如此。我们知道重构的好处，我们知道重构可以给我们的工作带来垂手可得的改变。但是我们还没有获得足够的经验，我们还看不到它的局限性。

这一小节比我希望的要短。暂且如此吧。随着更多人学会重构技巧，我们也将对??你应该尝试一下重构，获得它所提供的利益，但在此同时，你也应该时时监控其过程，注意寻找重构可能引入的问题。请让我们知道你所遭遇的问题。随着对重构的了解日益增多，我们将找出更多解决办法，并清楚知道哪些问题是真正难以解决的。

9 • 数据库（Databases）

「重构」经常出问题的一个领域就是数据库。绝大多数商用程序都与它们背后的database schema（数据库表格结构）紧密耦合（coupled）在一起，这也是database schema如此难以修改的原因之一。另一个原因是数据迁移（migration）。就算你非常小心地将系统分层（layered），将database schema和对象模型（object model）间的依赖降至最低，但database schema的改变还是让你不得不迁移所有数据，这可能是件漫长而烦琐的工作。

在「非对象数据库」（nonobject databases）中，解决这个问题的办法之一就是：在对象模型（object model）和数据库模型（database model）之间插入一个分隔层（separate layer），这就可以隔离两个模型各自的变化。升级某一模型时无需同时升级另一模型，只需升级上述的分隔层即可。这样的分隔层会增加系统复杂度，但可以给你很大的灵活度。如果你同时拥有多个数据库，或如果数据库模型较为复杂使你难以控制，那么即使不进行重构，这分隔层也是很重要的。

你无需一开始就插入分隔层，可以在发现对象模型变得不稳定时再产生它。这样你就可以为你的改变找到最好的杠杆效应。

对开发者而言，对象数据库既有帮助也有妨碍。某些面向对象数据库提供不同版本的对象之间的自动迁移功能，这减少了数据迁移时的工作量，但还是会损失一定时间。如果各数据库之间的数据迁移并非自动进行，你就必须自行完成迁移工作，这个工作量可是很大的。这种情况下你必须更加留神classes内的数据结构变化。你仍然可以放心将classes的行为转移过去，但转移值域（field）时就必须格外小心。数据尚未被转移前你就得先运用访问函数（accessors）造成「数据已经转移」的假象。一旦你确定知道「数据应该在何处」时，就可以一次性地将数据迁移过去。这时惟一需要修改的只有访问函数（accessors），这也降低了错误风险。

10 • 修改接口（Changing Interfaces）

关于对象，另一件重要事情是：它们允许你分开修改软件模块的实现（implementation）和接口（interface）。你可以安全地修改某对象内部而不影响他人，但对于接口要特别谨慎 — 如果接口被修改了，任何事情都有可能发生。

一直对重构带来困扰的一件事就是：许多重构手法的确会修改接口。像Rename Method（273）这么简单的重构手法所做的一切就是修改接口。这对极为珍贵的封装概念会带来什么影响呢？

如果某个函数的所有调用动作都在你的控制之下，那么即使修改函数名称也不会有任何问题。哪怕面对一个public函数，只要能取得并修改其所有调用者，你也可以安心地将这个函数易名。只有当需要修改的接口系被那些「找不到，即使找到也不能修改」的代码使用时，接口的修改才会成为问题。如果情况真是如此，我就会说：这个接口是个「已发布接口」（published interface）— 比公开接口（public interface）更进一步。接口一旦发行，你就再也无法仅仅修改调用者而能够安全地修改接口了。你需要一个略为复杂的程序。

这个想法改变了我们的问题。如今的问题是：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（published interface），你必须同时维护新旧两个接口，直到你的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。千万不要拷贝函数实现码，那会让你陷入「重复代码」（duplicated code）的泥淖中难以自拔。你还应该使用Java提供的 deprecation（反对）设施，将旧接口标记为 "deprecated"。这么一来你的调用者就会注意到它了。

这个过程的一个好例子就是Java容器类（collection classes）。Java 2的新容器取代了原先一些容器。当Java 2容器发布时，JavaSoft花了很大力气来为开发者提供一条顺利迁徙之路。

「保留旧接口」的办法通常可行，但很烦人。起码在一段时间里你必须建造（build）并维护一些额外的函数。它们会使接口变得复杂，使接口难以使用。还好我们有另一个选择：不要发布（publish）接口。当然我不是说要完全禁止，因为很明显你必得发布一些接口。如果你正在建造供外部使用的APIs，像Sun所做的那样，肯定你必得发布接口。我之所以说尽量不要发布，是因为我常常看到一些开发团队公开了太多接口。我曾经看到一支三人团队这么工作：每个人都向另外两人公开发布接口。这使他们不得不经常来回维护接口，而其实他们原本可以直接进入程序库，径行修改自己管理的那一部分，那会轻松许多。过度强调「代码拥有权」的团队常常会犯这种错误。发布接口很有用，但也有代价。所以除非

真有必要，别发布接口。这可能意味需要改变你的代码拥有权观念，让每个人都可以修改别人的代码，以运应接口的改动。以搭档（成对）编程（Pair Programming）完成这一切通常是个好主意。

11.不要过早发布（published）接口。请修改你的代码拥有权政策，使重构更顺畅。

Java之中还有一个特别关于「修改接口」的问题：在throws子句中增加一个异常。这并不是对签名式（signature）的修改，所以你无法以delegation（委托手法）隐藏它。但如果用户代码不作出相应修改，编译器不会让它通过。这个问题很难解决。你可以为这个函数选择一个新名tion（可控式异常）转换成一个unchecked exception（不可控异常）。你也可以抛出一个unchecked异常，不过这样你就会失去检验能力。如果你那么做，你可以警告调用者：这个unchecked异常日后会变成一个checked异常。这样他们就有时间在自己的代码中加上对此异常的处理。出于这个原因，我总是喜欢为整个package定义一个superclass异常（就像java.sql的SQLException），并确保所有public函数只在自己的throws子句中声明这个异常。这样我就可以随心所欲地定义subclass异常，不会影响调用者，因为调用者永远只知道那个更具一般性的superclass异常。

12・难以通过重构手法完成的设计改动

通过重构，可以排除所有设计错误吗？是否存在某些核心设计决策，无法以重构手法修改？在这个领域里，我们的统计数据尚不完整。当然某些情况下我们可以很有效地重构，这常常令我们倍感惊讶，但的确也有难以重构的地方。比如说在一个项目中，我们很难（但还是有可能）将「无安全需求（no security requirements）情况下构造起来的系统」重构为「安全性良好的（good security）系统」。

这种情况下我的办法就是「先想象重构的情况」。考虑候选设计方案时，我会问自己：将某个设计重构为另一个设计的难度有多大？如果看上去很简单，我就不必太担心选择是否得当，于是我就会选最简单的设计，哪怕它不能覆盖所有潜在需求也没关系。但如果预先看不到简单的重构办法，我就会在设计上投入更多力气。不过我发现，这种情况很少出现。

12・何时不该重构？

有时候你根本不应该重构 — 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，重构它还不如从新写一个来得简单。作出这种决定很困难，我承认我也没有什么好准则可以判断何时应该放弃重构。

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。

一个折衷办法就是：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作出「重构或重建」的决定。这是一个颇具希望的办法，但我还没有足够数据，所以也无法写出优秀的指导原则。对于一个重要的古老系统，这肯定会是一个很好的方向。

另外，如果项目已近最后期限，你也应该避免重构。在此时机，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，而那个时候已经时不我予。Ward Cunningham对此有一个很好的看法。他把未完成的重构工作形容为「债务」。很多公司都需要借债来使自己更有效地运转。但是借债就得付利息，过于复杂的代码所造成的「维护和扩展的额外开销」就是利息。你可以承受一定程度的利息，但如果利息太高你就会被压垮。把债务管理好是很重要的，你应该随时通过重构来偿还一部分债务。

如果项目已经非常接近最后期限，你不应该再分心于重构，因为已经没有时间了。不过多个项目经验显示：重构的确能够提高生产力。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（Refactoring）与设计

「重构」肩负一项特别任务：它和设计彼此互补。初学编程的时候，我埋头就写程序，浑浑噩噩地进行开发。然而很快我便发现，「事先设计」（upfront design）可以助我节省回头工的高昂成本。于是我很快加强这种「预先设计」风格。许多人都把设计看作软件开发的关键环节，而把编程（programming）看作只是机械式的低级劳动。他们认为设计就像画工程图而编码就像施工。但是你要知道，软件和真实器械有着很大的差异。软件的可塑性更强，而且完全是思想产品。正如Alistair Cockburn所说：『有了设计，我可以思考更快，但是其中充满小漏洞。』

有一种观点认为：重构可以成为「预先设计」的替代品。这意思是你根本不必做任何设计，只管按照最初想法开始编码，让代码有效运作，然后再将它重构成型。事实上这种办法真的可行。我的确看过有人这么做，最后获得设计良好的软件。极限编程（Extreme Programming）【Beck, XP】 的支持者极力提倡这种办法。

尽管如上所言，只运用重构也能收到效果，但这并不是最有效的途径。是的，即使极限编程（Extreme Programming）爱好者也会进行预先设计。他们会使用CRC卡或类似的东西来检验各种不同想法，然后才得到第一个可被接受的解决方案，然后才能开始编码，然后才能重构。关键在于：重构改变了「预先设计」的角色。如果没有重构，你就必须保证「预先设计」正确无误，这个压力太大了。这意味如果将来需要对原始设计做任何修改，代价都将非常高昂。因此你需要把更

多时间和精力放在预先设计上，以避免日后修改。

如果你选择重构，问题的重点就转变了。你仍然做预先设计，但是不必一定找出正确的解决方案。此刻的你只需要得到一个足够合理的解决方案就够了。你很肯定地知道，在实现这个初始解决方案的时候，你对问题的理解也会逐渐加深，你可能会察觉最佳解决方案和你当初设想的有些不同。只要有重构这项武器在手，就不成问题，因为重构让日后的修改成本不再高昂。

这种转变导致一个重要结果：软件设计朝向简化前进了一大步。过去未曾运用重构时，我总是力求得到灵活的解决方案。任何一个需求都让我提心吊胆地猜疑：在系统寿命期间，这个需求会导致怎样的变化？由于变更设计的代价非常高昂，所以我希望建造一个足够灵活、足够强固的解决方案，希望它能承受我所能预见的所有需求变化。问题在于：要建造一个灵活的解决方案，所需的成本难以估算。灵活的解决方案比简单的解决方案复杂许多，所以最终得到的软件通常也会更难维护 — 虽然它在我预先设想的??方向上，你也必须理解如何修改设计。如果变化只出现在一两个地方，那不算大问题。然而变化其实可能出现在系统各处。如果在所有可能的变化出现地点都建立起灵活性，整个系统的复杂度和维护难度都会大大提高。当然，如果最后发现所有这些灵活性都毫无必要，这才是最大的失败。你知道，这其中肯定有些灵活性的确派不上用场，但你却无法预测到底是哪些派不上用场。为了获得自己想要的灵活性，你不得不加入比实际需要更多的灵活性。

有了重构，你就可以通过一条不同的途径来应付变化带来的风险。你仍旧需要思考潜在的变化，仍旧需要考虑灵活的解决方案。但是你不必再逐一实现这些解决方案，而是应该问问自己：『把一个简单的解决方案重构成这个灵活的方案有多大难度？』如果答案是「相当容易」（大多数时候都如此），那么你就只需实现目前的简单方案就行了。

重构可以带来更简单的设计，同时又不损失灵活性，这也降低了设计过程的难度，减轻了设计压力。一旦对重构带来的简单性有更多感受，你甚至可以不必再预先思考前述所谓的灵活方案 — 一旦需要它，你总有足够的信心去重构。是的，当下只管建造可运行的最简化系统，至于灵活而复杂的设计，唔，多数时候你都不会需要它。

劳而无获— Ron Jeffries

Chrysler Comprehensive Compensation（克莱斯勒综合薪资系统）的支付过程太慢了。虽然我们的开发还没结束，这个问题却已经开始困扰我们，因为它已经拖累了测试速度。

Kent Beck、Martin Fowler和我决定解决这个问题。等待大伙儿会合的时间里，凭着我对这个系统的全盘了解，我开始推测：到底是什么让系统变慢了？我想到数种可能，然后和伙伴们谈了几种可能的修改方案。最后，关于「如何让这个系统运行更快」，我们提出了一些真正的好点子。

然后，我们拿Kent的量测工具度量了系统性能。我一开始所想的可能性竟然全都不是问题肇因。我们发现：系统把一半时间用来创建「日期」实体（instance）。更有趣的是，所有这些实体都有相同的值。

于是我们观察日期的创建逻辑，发现有机会将它优化。日期原本是由字符串转换而生，即使无外部输入也是如此。之所以使用字符串转换方式，完全是为了方便键盘输入。好，也许我们可以将它优化。

于是我们观察日期怎样被这个程序运用。我们发现，很多日期对象都被用来产生「日期区间」实体（instance）。「日期区间」是个对象，由一个起始日期和一个结束日期组成。仔细追踪下去，我们发现绝大多数日期区间是空的！

处理日期区间时我们遵循这样一个规则：如果结束日期在起始日期之前，这个日期区间就该是空的。这是一条很好的规则，完全符合这个class的需要。采用此一规则后不久，我们意识到，创建一个「起始日期在结束日期之后」的日期区间，仍然不算是清晰的代码，于是我们把这个行为提炼到一个factory method（译注：一个著名的设计模式，见《Design Patterns》），由它专门创建「空的日期区间」。

我们做了上述修改，使代码更加清晰，却意外得到了一个惊喜。我们创建一个固定不变的「空日期区间」对象，并让上述调整后的factory method每次都返回该对象，而不再每次都创建新对象。这一修改把系统速度提升了几乎一倍，足以让测试速度达到可接受程度。这只花了我们大约五分钟。

我和团队成员（Kent和Martin谢绝参加）认真推测过：我们了若指掌的这个程序中可能有什么错误？我们甚至凭空做了些改进设计，却没有先对系统的真实情况进行量测。

我们完全错了。除了一场很有趣的交谈，我们什么好事都没做。

教训：哪怕你完全了解系统，也请实际量测它的性能，不要臆测。臆测会让你学到一些东西，但十有八九你是错的。