靠谱程序员必备技能——重构

为什么要重构

你可能正在面对一个遗留系统，增加一个需求要改动好几个文件，定位 Bug 经常要花掉一整天时间，修复一个 Bug 可能又制造了 3 个新的 Bug。你也可能会为了软件设计和同事争得面红耳赤，讨论如何应对未来可能出现的需求变化。

为了开发一个新需求，你打开一份源代码，完全不知所云嘛，你吐槽着谁能写出如此不堪入目的代码，于是决定查看版本记录，把这个家伙找出来鄙视一下。然后你在提交历史里看到了自己的名字… 恭喜你，你进步了。如果你是一个积极进取的程序员，通常在几个月甚至几个星期之后就认不出自己写的代码。你总能发现更好的实现方式，让代码更加优雅。
随着增加新特性或需求变更，代码会变得越来越难以维护。敏捷软件开发的十二条原则中有一条是：我们始终拥抱需求变化，哪怕是在软件开发的后期。为了达到这种状态，我们就要在开发过程中持续地优化代码。

而重构这项技术，为我们提供了一种更可控的方式来优化代码。

重构是什么

重构，通常指的是「代码重构」，起源于 Smalltalk 圈子。

在日常工作中，我们把重构既作为名词又作为动词来使用，作为名词时，它的意思是：

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。

所以我们会说，「这里需要做一个重构」，「这个重构有点问题」等。

而在其它时候，我们也会说：「我们来重构一下这段代码吧」，「我正在重构一个遗留系统」，这时就是把重构当做动词在用，它的意思是：

> 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。

重构本质上是一种代码整理技术，这项技术使得代码整理的效率更高，风险更小。

如何做

接下来从几个方面来说说如何做重构：

- 什么时候开始
- 什么时候停止
- 前提条件
- 重构的过程

什么时候开始

重构不应该是一个单独的环节，应该融入到开发软件编写代码的过程中，就像使用版本控制系统提交代码一样，是一个必须做的动作。你不会跟项目经理说，我需要申请一段时间来提交代码，所以也不用说服项目经理给你时间重构。你可以在开发新功能，修复 Bug 的过程中就把重构做了，除了你的程序员同伴，没有人知道你做了什么。而他们会认为你做了一件了不起的事情，因为你让代码结构更清晰了，以后添加新特性就会更容易，而 Bug 也无处藏身。

如果你采用 TDD 的方式（测试驱动开发），那重构已完全融入到了开发过程中。如果没有采用 TDD，通常有四个时机可以考虑要不要重构：

事不过三

如果有段代码让你修改起来很不舒服，前两次还可以忍耐，第三次就无需再忍了，果断操起 IDE 重构之。因为出现了三次修改，说明有很大概率以后还会修改，这是一笔划算的投资。

添加新功能

有时候我们发现要添加一个新功能很难，我们可以对代码做一些重构，让添加新功能变得容易。

修复缺陷

在修 Bug 时，我们大部分的时间会花在定位 Bug 上，为什么这么难以找到呢？多半是因为代码结构不清晰，如果代码在同一抽象层次上，每个方法都在 10 行以内，每个方法名和变量名都能清晰地表达意图，Bug 就再无藏身之处。所以，通过重构代码，可以让 Bug 自动浮现出来。

代码评审

Code Review 已是一个广泛采用的实践，在 Code Review 时，其他程序员会提出代码修改的意见，记录下来，等 Code Review 结束之后就可以开始重构了。

什么时候停止

重构到什么时候，我们就认为可以停止了呢？

有两个标准可以参考，一个是「简单设计」的四条原则：

- 通过所有测试
- 没有重复
- 表达意图
- 最少化程序元素（类，接口，变量，方法等）

另一个是满足《Clean Code》（整洁代码）的要求。

前提条件

现代 IDE，尤其是 JetBrains 公司的一系列产品，支持常用的重构手法，极大地降低了重构的风险。但为了保证不改变软件的可观察行为，还是需要完善的测试。我也做过一些没有测试代码保护的重构，通常会加一个端到端测试以保证不破坏最重要的功能。实在很难编写测试代码，至少也要手工测试来保证重构真的没有改变软件行为。

另一个重要前提是，使用版本控制系统，比如 Git。因为我们的重构并不一定总是令人满意，也有可能出现错误，导致软件变得不可用，所以最好是小步提交，以保证可以随时放弃变更，回到上一次满意的状态。

重构的过程

重构的基本步骤是：

-测试保护

-识别味道

-采用手法

-运行测试

-提交代码

测试保护

如果没有测试代码，就要先添加测试代码。如果有测试代码，先运行一下，保证在开始重构之前，测试是运行通过的。还要认真审查一下测试代码，看是否有遗漏一些场景，有遗漏的话要补充遗漏的测试场景。

识别味道

怎么知道哪些代码需要重构呢？首先，代码是可以工作的，我们并不能说它有问题，但它又不像我们期望的那样好。受 Kent Beck 刚出生的女儿的使用的尿布的启发，Martin Fowler 和 Kent Beck 决定用「味道」这个词来表示需要重构的代码。他们在《重构》一书中列举了 22 中常见的味道，如果你看《Clean Code》的话，会发现还有更多。不过，他们并没有给出一个具体的标准，而是需要我们的直觉来判断。比如多大的类算「过大的类」？多少行代码算「过长的方法」？这些需要自行判断，而直觉的形成有两种方法，一是随着编码经验的增多自然形成，另一种更快的方式是大量阅读优秀的开源代码，提高自己的代码审美。

《重构》一书中的味道可以分为五类：

-膨胀剂

-OO 使用不合理

-难以修改

-可有可无

-耦合

书中都有详细的解释，这里不再赘述。

膨胀剂

滥用 OO

Switch 语句

临时字段

被拒绝的馈赠

异曲同工的类

难以修改

发散式变化

散弹式修改

平行继承体系

发散式变化和散弹式修改是比较容易混淆的两个味道。前者指一个类的职责过多，有很多因素会引起它的变化，具体的表现就是，不同的需求都会修改同一个文件，导致经常冲突，不能顺利地并行开发。后者指的是改一个需求要修改很多个文件，说明没有把强内聚的代码归拢到一起。

可有可无

注释

重复代码

冗赘类

数据类

死代码

夸夸其谈未来性

大部分的注释都是没有必要的，注释应该描述「做了什么」和「为什么做」而不是「怎么做」，方法体内的注释基本都可以通过抽取方法并指定一个有意义的名字来解决。很多为了应对未来需求变化而写的代码基本永远不会被执行。

耦合

特性依恋

狎昵关系

消息链

中间人

不完整的类库

你可能发现了，有些味道是比较容易识别的，比如重复代码，注释等。而有些就比较高

级，比如特性依恋，中间人等，要识别高级味道，需要理解面向对象的特性和设计原则。

采用手法

识别到味道之后，就要知道有什么对应的手法可以消除这个味道，执行完这个手法之后代码会变成什么样子。

在《重构》一书中，列举了 66 个常用手法，可以分为六大类：

-重组函数

-搬移特性

-组织数据

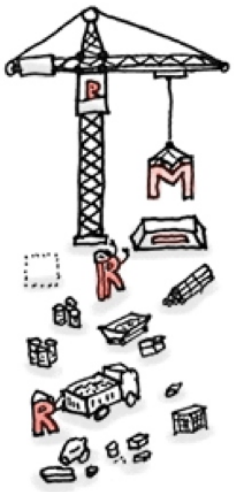
-简化条件

-简化调用

-处理概括

这些手法在书中都有详细的讲解，我就不在这里重复了。只整理出来，给大家一个宏观的印象：

重组函数



抽取方法　　分解临时变量

内联方法　　移除对参数赋值

抽取变量　　以函数对象取代函数

内联临时变量　　替换算法

查询取代临时变量

搬移特性

搬移函数

搬移字段

提炼类

内联类

隐藏委托关系

移除中间人

引入外加函数

引入本地扩展

组织数据

自封装字段

以对象取代数据值

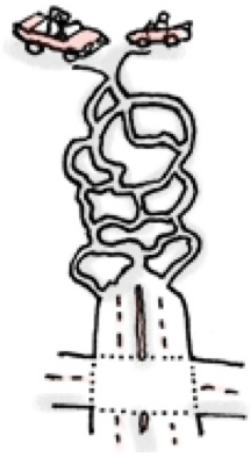
将值对象改为引用对象

以对象取代数组

复制被监视数据

单向关联到双向关联

简化条件



分解条件表达式　　卫语句取代嵌套条件

合并条件表达式　　多态取代条件

合并重复条件片段　引入NULL对象

移除控制标记　　引入断言

简化函数调用

函数改名　　零函数携带参数

添加参数　　用函数取代参数

移除参数　　保留完整对象

引入参数对象　　以函数取代参数

移除设值函数　　隐藏函数

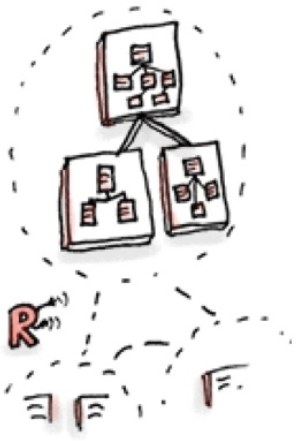
以测试取代异常　　封装向下转型

以异常取代错误码

以工厂函数取代构造函数

将查询函数和修改函数分离

处理概括关系



字段上移	字段下移
函数上移	抽取子类
构造函数本体上移	抽取超类
函数下移	抽取接口
折叠继承体系	塑造函数模板
以委托取代继承	以继承取代委托

运行测试

在采用了手法修改代码之后，就要执行测试以确保真的没有改变软件的行为。可能有时会发现，做了重构之后测试会失败，但实现并没有问题，我们需要修改测试代码让它成功。这就说明测试写的不合理，给重构带来了负担，所以我们测试的粒度要把握好，太细的粒度就会增加维护成本。比如，有些人会给每个私有方法都写单元测试，那有可能采用「内联函数」这个手法之后这个方法就不存在了，就需要修改测试。这里说起来话就长了，以后再写一篇如何写有效的测试的文章吧。重点是重构之后，一定要执行测试，不管是手工测试或自动化测试。

提交代码

最后，如果你采用了一个比较复杂的手法，或者即将采用一个复杂的手法，最好先提交一下代码，以保证出现意外后能快速回滚，避免浪费时间。

重构要采取「小步快跑」的原则，尽量采用安全的手法，让测试一直处于通过的状态。从低级的坏味道开始，消除低级味道之后，高级味道才会浮现出来。

进阶

重构与设计的关系

在没有重构这个技术之前，广泛采用的是 Big Front Design，在开始编码之前要进行非常详细的设计，考虑应对未来出现的各种变化。而有了重构技术之后，前期设计的压力就小了，毕竟可以随时通过重构来改善设计，应对变化。所以你大可不必一上来就应用《设计模式》把代码搞复杂，先用简单的实现满足当前需求即可。等变化真正来临时，再通过重构技术调整设计，模式给我们提供了一个方向，但并不是最终目标。还记得简单设计的四条原则吗？通过测试，没有重复，表达意图，最少元素。除了这四条原则，还有 SOLID，DRY，KISS 等设计原则。只要最终的代码符合好的原则，干净整洁没有坏味道，管它符不符合某个模式呢？！

大型遗留系统的重构

对于代码上百万，千万行的遗留系统，怎么重构呢？满地都是坏味道，一点点去重构，什么时候是个头？

这时，选择哪些代码来重构就非常重要，影响到投资回报。如果对代码进行分类，将会得出几种类型：

- 不会被执行的烂代码
- 运行稳定，基本不会改动的烂代码
- 经常发现 Bug 的烂代码
- 经常需要变更的烂代码

不会被执行的代码，直接删除就好了。运行稳定的又不需要改动的，动它反而可能引入风险，当然，在时间充裕的情况下，还是可以重构的。真正有价值，值得重构的，投入产出比最高的，是经常出问题和经常会有需求变更的烂代码。优化了这部分代码，可以减少 Bug 和进行需求变更的时间。

总结

好了。关于重构我想分享的就是这些，我们来回顾一下：

为什么要重构？

为了让软件始终可以维护，保证开发效率。

什么是重构？

一种以可控的方式整理代码的技术，在不改变软件可观察行为的前提下改善其内部结构。

什么时候开始？

事不过三，添加功能，修复 Bug，代码评审时。

什么时候停止？

重构到符合简单设计四条原则的 Clean Code。

前提条件

测试保护，版本控制。

重构的过程

运行测试，识别味道（常见的 22 种），采用手法（66 个），运行测试，提交代码。

重构与设计的关系

有了重构技术，我们不用在前期做非常详细的设计，做适当的设计，然后通过重构让设计浮现出来。不用在乎软件是否符合模式，只要符合原则即可。

大型遗留系统的重构

在经常需要修改的烂代码上做重构才有最大收益。

最后推荐一些学习资源：

- 《代码整洁之道》
- 《编写可读代码的艺术》
- 《重构》
- 《设计模式 - 可复用面向对象软件的基础》
- 《重构与模式》
- Transformation Priority Premise
- 用 IntelliJ IDEA 重构
- 重构十六字心法
- 练习重构的 Kata