27-理论一：什么情况下要重构？到底重构什么？又该如何重构？

“重构”这个词对于大部分工程师来说都不陌生。不过，据我了解，大部分人都只是“听得多做得少”，真正进行过代码重构的人不多，而把持续重构作为开发的一部分的人，就更是少之又少了。

一方面，重构代码对一个工程师能力的要求，要比单纯写代码高得多。重构需要你能洞察出代码存在的坏味道或者设计上的不足，并且能合理、熟练地利用设计思想、原则、模式、编程规范等理论知识解决这些问题。

另一方面，很多工程师对为什么要重构、到底重构什么、什么时候重构、又该如何重构等相关问题理解不深，对重构没有系统性、全局性的认识，面对一堆烂代码，没有重构技巧的指导，只能想到哪改到哪，并不能全面地改善代码质量。

为了让你对重构有个清晰的认识，对于这部分知识的讲解，我安排了六节课的内容，主要包含以下几个方面：

- 对重构概括性的介绍，包括重构的目的（why）、对象（what）、时机（when）、方法（how）；
- 保证重构不出错的手段，这里我会重点讲解单元测试和代码的可测试性；
- 不同规模的重构，重点讲解大规模高层次重构（比如系统、模块、代码结构、类与类之间的交互等的重构）和小规模低层次重构（类、函数、变量等的重构）。

话不多说，现在就让我们来学习第一部分内容：重构的目的、对象、时机和方法。

重构的目的：为什么要重构（why）？

虽然对于你来说，重构这个词可能不需要过多解释，但我们还是简单来看一下，大师是怎么描述它的。软件设计大师Martin Fowler 是这样定义重构的：“重构是一种对软件内部结构的改善，目的是在不改变软件的可见行为的情况下，使其更易理解，修改成本更低。”

实际上，当讲到重构的时候，很多书籍都会引用这个定义。这个定义中有一个值得强调的点：“重构不改变外部的可见行为”。我们可以把重构理解为，在保持功能不变的前提下，利用设计思想、原则、模式、编程规范等理论来优化代码，修改设计上的不足，提高代码质量。

简单了解重构的定义之后，我们重点来看一下，为什么要进行代码重构？

首先，重构是时刻保证代码质量的一个极其有效的手段，不至于让代码腐化到无可救药的地步。项目在演进，代码不停地在堆砌。如果没有人为代码的质量负责任，代码总是会往越来越混乱的方向演进。当混乱到一定程度之后，量变引起质变，项目的维护成本已经高过重新开发一套新代码的成本，想要再去重构，已经没有人能做到了。

其次，优秀的代码或架构不是一开始就能完全设计好的，就像优秀的公司和产品也都是迭代出来的。我们无法100%遇见未来的需求，也没有足够的精力、时间、资源为遥远的未来买单，所以，随着系统的演进，重构代码也是不可避免的。

最后，重构是避免过度设计的有效手段。在我们维护代码的过程中，真正遇到问题的时候，再对代码进行重构，能有效避免前期投入太多时间做过度的设计，做到有的放矢。

除此之外，重构对一个工程师本身技术的成长也有重要的意义。

从前面我给出的重构的定义来看，重构实际上是对我们学习的经典设计思想、设计原则、设计模式、编程规范的一种应用。重构实际上就是将这些理论知识，应用到实践的一个很好的场景，能够锻炼我们熟练使用这些理论知识的能力。除此之外，平时堆砌业务逻辑，你可能总觉得没啥成长，而将一个比较烂的代码重构成一个比较好的代码，会让你很有成就感。

除此之外，重构能力也是衡量一个工程师代码能力的有效手段。所谓“初级工程师在维护代码，高级工程师在设计代码，资深工程师在重构代码”，这句话的意思是说，初级工程师在已有代码框架下修改bug、修改添加功能代码；高级工程师从零开始设计代码结构、搭建代码框架；而资深工程师为代码质量负责，需要发觉代码存在的问题，重构代码，时刻保证代码质量处于一个可控的状态（当然这里的初级、高级、资深只是一个相对概念，并不是一个确定的职级）。

重构的对象：到底重构什么（what）？

根据重构的规模，我们可以笼统地分为大规模高层次重构（以下简称为“大型重构”）和小规模低层次的重构（以下简称为“小型重构”）。

大型重构指的是对顶层代码设计的重构，包括：系统、模块、代码结构、类与类之间的关系等的重构，重构的手段有：分层、模块化、解耦、抽象可复用组件等等。这类重构的工具就是我们学习过的那些设计思想、原则和模式。这类重构涉及的代码改动会比较多，影响面会比较大，所以难度也较大，耗时会比较长，引入bug的风险也会相对比较大。

小型重构指的是对代码细节的重构，主要是针对类、函数、变量等代码级别的重构，比如规范命名、规范注释、消除超大类或函数、提取重复代码等等。小型重构更多的是利用我们能后面要讲到的编码规范。这类重构要修改的地方比较集中，比较简单，可操作性较强，耗时会比较短，引入bug的风险相对来说也会比较小。你只需要熟练掌握各种编码规范，就可以做到得心应手。

关于具体如何来做大型重构和小型重构，我会在后面的课程中详细讲解。

重构的时机：什么时候重构（when）？

搞清楚了为什么重构，到底重构什么，我们再来看一下，什么时候重构？是代码烂到一定程度之后才去重构吗？当然不是。因为当代码真的烂到出现“开发效率低，招了很多人，天天加班，出活却不多，线上bug频发，领导发飙，中层束手无策，工程师抱怨不断，查找bug困难”的时候，基本上重构也无法解决问题了。

我个人比较反对，平时不注重代码质量，堆砌烂代码，实在维护不了了就大刀阔斧地重构、甚至重写的行为。有时候项目代码太多了，重构很难做得彻底，最后又搞出来一个“四不像的怪物”，这就更麻烦了！所以，寄希望于在代码烂到一定程度之后，集中重构解决所有问题是不现实的，我们必须探索一条**可持续、可演进**的方式。

所以，我特别提倡的重构策略是持续重构。这也是我在工作中特别喜欢干的事情。平时没有事情的时候，你可以看看项目中有哪些写得不够好的、可以优化的代码，主动去重构一下。或者，在修改、添加某个功能代码的时候，你也可以顺手把不符合编码规范、不好的设计重构一下。总之，就像把单元测试、Code Review作为开发的一部分，我们如果能把持续重构也作为开发的一部分，成为一种开发习惯，对项目、对自己都会很有好处。

尽管我们说重构能力很重要，但持续重构意识更重要。我们要正确地看待代码质量和重构这件事情。技术在更新、需求在变化、人员在流动，代码质量总会在下降，代码总会存在不完美，重构就会持续在进行。时刻具有持续重构意识，才能避免开发初期就过度设计，避免代码维护的过程中质量的下降。而那些看到别人代码有点瑕疵就一顿乱骂，或者花尽心思去构思一个完美设计的人，往往都是因为没有树立正确的代码质量观，没有持续重构意识。

重构的方法：又该如何重构（how）？

前面我们讲到，按照重构的规模，重构可以笼统地分为大型重构和小型重构。对于这两种不同规模的重构，我们要区别对待。

对于大型重构来说，因为涉及的模块、代码会比较多，如果项目代码质量又比较差，耦合比较严重，往往会牵一发而动全身，本来觉得一天就能完成的重构，你会发现越改越多、越改越乱，没一两个礼拜都搞不定。而新的业务开发又与重构相冲突，最后只能半途而废，revert掉所有的改动，很失落地又去堆砌烂代码了。

在进行大型重构的时候，我们要提前做好完善的重构计划，有条不紊地分阶段来进行。每个阶段完成一小部分代码的重构，然后提交、测试、运行，发现没有问题之后，再继续进行下一阶段的重构，保证代码仓库中的代码一直处于可运行、逻辑正确的状态。每个阶段，我们都要控制好重构影响到的代码范围，考虑好如何兼容老的代码逻辑，必要的时候还需要写一些兼容过渡代码。只有这样，我们才能让每一阶段的重构都不至于耗时太长（最好一天就能完成），不至于与新的功能开发相冲突。

大规模高层次的重构一定是有组织、有计划，并且非常谨慎的，需要有经验、熟悉业务的资深同事来主导。而小规模低层次的重构，因为影响范围小，改动耗时短，所以，只要你愿意并且有时间，随时都可以去做。实际上，除了人工去发现低层次的质量问题，我们还可以借助很多成熟的静态代码分析工具（比如CheckStyle、FindBugs、PMD），来自动发现代码中的问题，然后针对性地进行重构优化。

对于重构这件事情，资深的工程师、项目leader要负起责任来，没事就重构一下代码，时刻保证代码质量处在一个良好的状态。否则，一旦出现“破窗效应”，一个人往里堆了一些烂代码，之后就会有更多的人往里堆更烂的代码。毕竟往项目里堆砌烂代码的成本太低了。不过，保持代码质量最好的方法还是打造一种好的技术氛围，以此来驱动大家主动去关注代码质量，持续重构代码。

重点回顾

今天的讲解比较偏理论、偏思想教育，主要还是让你对重构有个正确的、全局性的认知，建立持续重构意识。我觉得，这可能比教会你一些重构技巧更重要，因为很多技术问题本身就不是单纯靠技术来解决的，更重要的是要有这种认知和意识。

好了，下面我们还是来总结一下。对于今天的内容，你需要重点理解并且掌握如下知识点。

1.重构的目的：为什么重构（why）？

对于项目来言，重构可以保持代码质量持续处于一个可控状态，不至于腐化到无可救药的地步。对于个人而言，重构非常锻炼一个人的代码能力，并且是一件非常有成就感的事情。它是我们学习的经典设计思想、原则、模式、编程规范等理论知识的练兵场。

2.重构的对象：重构什么（what）？

按照重构的规模，我们可以将重构大致分为大规模高层次的重构和小规模低层次的重构。大规模高层次重构包括对代码分层、模块化、解耦、梳理类之间的交互关系、抽象复用组件等等。这部分工作利用的更多的是比较抽象、比较顶层的设计思想、原则、模式。小规模低层次的重构包括规范命名、注释、修正函数参数过多、消除超大类、提取重复代码等等编程细节问题，主要是针对类、函数级别的重构。小规模低层次的重构更多的是利用编码规范这一理论知识。

3.重构的时机：什么时候重构（when）？

我反复强调，我们一定要建立持续重构意识，把重构作为开发必不可少的部分，融入到日常开发中，而不是等到代码出现很大问题的时候，再大刀阔斧地重构。

4.重构的方法：如何重构（how）？

大规模高层次的重构难度比较大，需要组织、有计划地进行，分阶段地小步快跑，时刻让代码处于一个可运行的状态。而小规模低层次的重构，因为影响范围小，改动耗时短，所以，只要你愿意并且有时间，随时随地都可以去做。

课堂讨论

今天课堂讨论的话题是：关于代码重构，你有什么心得体会、经验教训？或者，你也可以说说，在重构过往项目的时候，你遇到过哪些问题？

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言：

- brianhuuu 2020-01-03 06:24:11
  重构最难的还是领导不支持 [22赞]

  作者回复2020-01-03 08:03:06
  😂

- 刘大明 2020-01-03 08:38:56
  前段时间刚重构了一个功能模块。该模块可以说是祖传代码。里面堆砌着各种判断条件，就是所谓的箭头型代码。我接手这个功能重构的
  1.把代码读一遍和跑一遍，理解里面的需求。尽量画一个流程图。
  2.建立防护网。将需求拆分之后，针对每个拆分的业务点写单元测试。
  4.开始重构，解耦逻辑，设计方法的时候尽量让职业单一，类与类之间尽量符合迪米特原则，有依赖关系的类尽量只依赖类的特定方法。我觉得比较基础也是比较重要一点。不要有重复代码。命名要规范，类的各个职业要清晰。重构过程中，其实也要时不时的识别代码的坏味道。尽然是重构，那么肯定要比不重构之前肯定要更好。
  5.重构完成之后，通过防护网的测试。

  当天重构代码上线之后，基本上没有问题。运行了几天之后有一小段逻辑隐藏的比较深没有写这个逻辑测试，后面补上了一直都没有出过问题了。还是比较稳定的。
  我这里只是做了中小规模的重构，后面跟着小争哥继续系统的学习大规模重构，以及重构的技巧和思想。[5赞]

- 失火的夏天 2020-01-03 08:01:23

重构自然是要用的我们牛逼的设计模式和数据结构了。。。啊-_-||开个玩笑哈。

重构这玩意嘛，其实在第一版提上去的时候就应该要重构了，也就是我们常说的，边写边重构。

一个方法写的时候发现分支判断太多，工厂模式就要登场了。

如果大部分代码都比较重复，这个时候就需要往底层的抽象，甚至用上策略模式。

需要做一个非功能性需求，每一个接口调用都要记录的东西，我们为了避免业务侵入性，就要考虑代理模式重构之前的业务侵入性强的代码，将功能与非功能分离。

说到底，重构不要等，而是马上动手，只有行动了，才不会害怕。第一版稍微辛苦一些，以后就不会那么恶心了，功在当代，利在千秋。 [5赞]

- 峰 2020-01-03 07:49:58
  1，无单测的条件下，别说重构了，我不想以前任何代码，对，我是一个怕事的程序员～
  2，小步快走的重构方式很重要，毕其功于一役的重构总是构完了，发现和主干代码相差哈哈哈，我还是不合入了吧，留给自己欣赏自己精细雕琢的玩具。。。 [2赞]

- 辣么大 2020-01-03 08:40:29
  代码中的坏味道，好比人的头疼脑热。“小病”不管的话，迟早会发展成大病，需要动大手术，甚至病入膏肓。
  实际中的一些体会：
  一、在完成一个新需求时，在时间允许的情况下，会经常改进代码，使代码更优雅。
  二、“重构不改变外部的可见行为“，引入自动化测试非常重要，国内有些团队可能做的不好。因为改动代码可能引入bug，如果没有自动化测试，测起来就会非常费劲，改动的结果不确定。如果测试不方便，谁会愿意修改之前work的代码呢?
  三、持续集成、自动化测试、持续重构都是很好的工程实践。即使工作的项目中暂时没有使用，也应该有所了解。 [1赞]

- 黄林晴 2020-01-03 00:27:23
  打卡✔
  心得体会吧，哈哈哈哈哈
  我被频繁改动的需求压的喘不过气，再牛逼的架构怕是也抵不住😐 [1赞]

- Rain 2020-01-03 10:03:27
  最近一段时间都是在做重构. 其中包括了大规模重构, 也包括了中型的重构. 小范围重构比如一个类内, 两三个类之间, 这种一般都随手做了, 但大型和中型重构确实不好做.
  1. 我重构的目的, 改善既有代码, 使其内聚解耦方便扩展, 以适应新的业务发展需要;
  2. 我重构的对象, 一系列相关的模块, 一个大模块套着几个小模块, 各个小模块之间互相关联;
  3. 我重构的时机, (恰到好处) 为适应新的业务发展需要, 动态添加和组合模块;
  4. 我重构的方法, 小步快跑是一个很好的方式, 但我用不了, 因为既有代码耦合度有点高, 改一个小模块就要触动到大模块中的各个小模块, 所以不得不一下子迈个大步子; 单测和自动化测试肯定是保证重构结果准确性的好方法, 但不是每个团队或项目都有; 但但是并不是没有他们就不能重构, 只是会对重构的程序猿以及协助测试的测试媛有较高的要求, 我重构完之后还是出错不少, 我分析主要原因有两个, 第一个是我本身对业务没有完全吃透某些地方只能按照既有代码按照对代码的理解来重构; 第二是既有代码不是很熟悉, 以至于某些地方做错了 或 少做了. 这些问题包括如何引入合适的设计模式来满足最新的业务需要都不是最令我印象深刻的, 我在做重构之前跟领导商量了如何尽可能的小步快跑的方式, 结果是没有, 所以只能在重构分支做完完再合并到开发主干. 而主干正被若干其他功能分支依赖并且开发, 这样就造成了变重构变开发变处理冲突, 再重构再开发再处理冲突的过程. 有点痛苦但算是经历过了吧, 在这儿感谢下我们领导对我工作的

支持. 嘿嘿谢谢领导.

- 郡鸿 2020-01-03 09:42:00
  最近正好在重构一个银行类业务的代码，之前的代码是经过很多人的堆砌，很多不同的功能都写在一个类里面，导致那个类有几千行代码，而且大量的重复代码。我接手之后，首先是按照不同的功能，划分了不同的模块。其次是把大量的重复代码单独抽离了出去，进行了公共的调用。这样一来，结构上更独立了，耦合度降低了很多，当然还有很多其他可以重构的地方。

- Jackey 2020-01-03 09:28:22
  最近打算重构一波前同事留下的代码，有点无从下手。怎么说呢，不知道为什么把所有类都用一个工厂来生成，登录、注册、退出…四五个功能对外只暴露了一个接口。打算先把大工厂拆开吧😲

- liu_liu 2020-01-03 09:12:17
  1. 功能点的梳理。大型重构中会涉及到很多功能模块，整理出完整的功能点，从而避免重构中的遗漏。
  2. 熟悉理清各模块的业务逻辑，不要一上手就开始改动。
  3. 重构后各个模块的测试。
  4. 新功能的同步。
  5. 容灾开关，老旧代码的切换，降低上线风险。

- whistleman 2020-01-03 09:02:41
  打开
  持续重构很重要，想等到最后大重构，结果基本就是无从下手而放弃。

- 安静的boy 2020-01-03 08:56:49
  我现在负责的项目是我从零就参与的，到现在大大小小已经迭代了十几个版本。我发现随着版本的迭代，会出现很多相似的重复代码，这个时候我就会去想办法重构代码，做一些抽象啊，利用一些设计模式，不过我现在只用到了模板设计模式。如果不重构我觉着以后需求再变化改动的地方太多了，而且还很有可能出错。另外，我发现重构了以后代码的可读性也会比较好。

- Kang 2020-01-03 08:52:44
  打卡

- Frode 2020-01-03 08:44:38
  我也是像改以前的垃圾代码，进行重构，但现在我们项目中没有单元测试，有点不敢做太多的改动，怕影响功能，是不是重构后需要测试进行一下测试？我只能在修改功能的时候忍不住小重构一下，这样我也不怕引来bug，本来也要测试←_←

- Geek1173 2020-01-03 08:35:15
  重构这件事情难度还是比较大的。由于项目处于运营状态，一处改动都会牵涉到测试，同事也担心重构带来的风险无法意料，虽然都是本着不改变功能的目的。

- 小晏子 2020-01-03 08:28:09
  持续重构确实是个非常值得做的事情，而且我感觉不光资深工程师，技术leader要做，初级工程师也要建立这样的意识。我们很多时候在平时的开发过程中被各种业务需求追着跑，所以代码质量下降很厉害，甚至单元测试都很少写，所以不定期的随时小重构是非常必要的。还有一种情况是最开始会有组内大佬关注代码质量架构问题，会去code review，可是等大佬离职，就没人关注代码质量了，所以全员建立重构意识以及关注代码质量是很必要的。

- 醉比 2020-01-03 08:26:36

需求上线初期是经过测试同学测试的，后续重构也就不安排资源来测试了，这时候心里就会犯嘀咕别改出什么问题，慢慢重构也就搁置了，可能是没有把单测这些事情做好导致重构没底。

- 知行合一 2020-01-03 08:24:39
  重构的意义:防止代码腐化到无可救药并且避难过度设计。建立持续重构的意识，同时也要有足够的单元测试作为前提，才能保证重构的正确性。

- 梦醒十分 2020-01-03 08:14:06
  做重构要眼高手低。只有眼光到了，才知道如何修改。

- Chen 2020-01-03 07:54:36
  我们项目大刀阔斧的重构了几次。可能是像老师说的缺少规划，导致没有一个产品敢把重构的代码上线，最后不了了之。