链滴

《重构》读书笔记

作者：valarchie

原文链接：https://ld246.com/article/1577636831325

来源网站：链滴

<blockquote>
之前读完《重构，改善代码既有设计》一书，书中内容虽然简单，但却有效。一些小小的重构积起来便使得我们的代码开始变得更优雅。印象很深的是作者倡导的事不过三的原则，代码出现三次重即需要进行重构。我深有体会，经常觉得自己的某些设计不好，但是又跟自己说下次再一起重构吧？果日复一日需要重构的设计越来越多，导致重构的成本越高，越不敢重构。这篇文章再温习一下重构书中的内容。
</blockquote>
<pre><code class="highlight-chroma">尽信书不如无书。书中只是建议并非标准，有些时候我们需要具体情况具体分析。
</code></pre>
<h3 id="代码的坏味道">代码的坏味道</h3>
<h5 id="1--重复代码">1. 重复代码</h5>
重复代码意味着冗余，当重复的代码需要修改时要修改所有重复的地方，稍微疏忽便会出现 bug
<h5 id="2--过长函数">2. 过长函数</h5>
当函数过长时，人很难一下掌握太多的细节，使得修改这个函数的成本很高。
<h5 id="3--过大的类">3. 过大的类</h5>
单个类承担的责任过多，违反单一职责而形成过大的类。
<h5 id="4--过长的参数列">4. 过长的参数列</h5>
长的参数列简直是噩梦，因为稍有不慎将类型相同的参数填错顺序就引发 Bug 了。（好在 IDEA 编译器能对填入的每个参数提示参数名）
<h5 id="5--发散式变化">5. 发散式变化</h5>
就是类受多个变化的影响，是违反迪米特法则的结果。类应该尽可能少的与其他类打交道，避免必要的关联。
<h5 id="6--霰弹式修改">6. 霰弹式修改</h5>
单个变化引发多个类的相应修改。
<h5 id="7--依恋情节">7. 依恋情节</h5>
当一个类的行为严重依赖其他类的时候，我们需要思考这个行为真正的归属。
<h5 id="8--数据泥团">8. 数据泥团</h5>
两个类中相同的字段或者方法签名中相同的参数总是一起出现，可能这些字段可以自成一类。
<h5 id="9--基本类型偏执">9. 基本类型偏执</h5>
此基本类型并不是指 int、long 这类，而是指类中出现很多小字段，比如省、市、区、住址可以装成一个地址对象作为 User 类中的一个属性。
<h5 id="10--switch问题">10. Switch 问题</h5>
当出现 Switch 重复时，同样的 Switch 散布在不同的地方，增加一个新的 case，需要找到所有 witch 进行增加。
<h5 id="11--平行继承体系">11. 平行继承体系</h5>
每当你为某个类增加一个子类，必须也为另一个类相应增加一个子类。
<h5 id="12--夸夸奇谈的未来性">12. 夸夸奇谈的未来性</h5>
过度设计。
<h5 id="13--冗赘类">13. 冗赘类</h5>
用处微乎其微以致于不如不作为一个类。或者作为内部类。
<h5 id="14--令人迷惑的临时字段">14. 令人迷惑的临时字段</h5>
类中的某些字段仅为特殊情况而定，或者类中的字段仅为了某个函数的方便声明为成员变量而没其他用处。
<h5 id="15--过度耦合的调用链">15. 过度耦合的调用链</h5>
一个对象向另一个对象发起一个请求，再由另一个对象请求其他对象。
<h5 id="16--中间人">16. 中间人</h5>
无用的委托，过多中间层。
<h5 id="17--狎昵关系">17. 狎昵关系</h5>
两个类关系过于紧密。一个类过于关注另一个类的成员，使得高耦合。
<h5 id="18--异曲同工的类">18. 异曲同工的类</h5>
不同的类或者函数，做着相同的事。

<h5 id="19--不完美的类库">19. 不完美的类库</h5>
类库不能满足实际开发需求。
<h5 id="20--纯数据类">20. 纯数据类</h5>
类似于 DDD 中所阐述的贫血模型，仅有数据没有行为的类。
<h5 id="21--被拒绝的遗赠">21. 被拒绝的遗赠</h5>
子类继承了父类不必要的函数或者数据。
<h5 id="22--过多的注释">22. 过多的注释</h5>
过多的注释说明代码的自解释能力很差。需要重构。
<hr>
<h3 id="重新组织函数">重新组织函数</h3>
<h5 id="1--提炼函数">1. 提炼函数</h5>
将代码放在独立函数里中，并让函数名解释该函数的用途。函数的粒度越小，被复用的可能性就大；并且粒度越小，函数的覆写也会更容易。
<h5 id="2--内联函数">2. 内联函数</h5>
对于一些函数，它的本体和函数名一样清楚易懂，那这个函数没有必要。在函数调用点插入函数体，然后移除该函数。
<h5 id="3--内联临时变量">3. 内联临时变量</h5>
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（下一个重构手法以查询代替临时变量）。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。
<h5 id="4--以查询代替临时变量">4. 以查询代替临时变量</h5>
如果你的程序中有一个临时变量，他的赋值操作是通过一个表达式来进行的，那我们可以把这个达式单独提炼出一个函数，在源程序中对变量的引用改为对这个函数的引用。
<h5 id="5--引入解释性变量">5. 引入解释性变量</h5>
你有一个复杂的表达式，该表达式可能非常复杂并难以阅读，将该复杂表达式（或其中的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，条件逻辑中，可以运用临时变量来解释一步运算的意义。
<h5 id="6--移除对参数的赋值">6. 移除对参数的赋值</h5>
代码对一个参数进行赋值，非常具体迷惑性，特别是 Java 这种值传递的方式。我们应该以一个时变量取代该参数的位置。
<h5 id="7--分解临时变量">7. 分解临时变量</h5>
你的程序有某个临时变量被赋值超过一次，它既不是循环变量，也不被用于收集计算结果。这种次赋值导致变量含义不清晰。我们应该针对每次赋值，创造一个独立、对应的临时变量。
<h5 id="8--以函数对象取代函数">8. 以函数对象取代函数</h5>
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独的对象里中，如此一来局部变量就成了对象内的字段，然后你就可以在同一个对象中将这个大型函分解为多个小型函数。局部变量的存在会增加函数分解难度。如果一个函数之中局部变量泛滥成灾，么想分解这个函数是非常困难的。
<h5 id="9--替换算法">9. 替换算法</h5>
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。
<hr>
<h3 id="在对象之间搬移特性">在对象之间搬移特性</h3>
<h5 id="1--搬移函数">1. 搬移函数</h5>
类中的一个函数使用另一个类的对象的次数比使用自己所在类的对象的次数还多，很有可能这个数被放在了错误的类中。
<h5 id="2--搬移字段">2. 搬移字段</h5>
你的程序中，某个字段被其所驻类之外的另一个类更多的用到。在目标类新建一个字段，修改源段的所有用户，令它们该用新字段。
<h5 id="3--提炼类">3. 提炼类</h5>
某个类做了看似应该两个类做的事，或者某些数据和函数总是一起出现，经常同时变化甚至彼此依，这就表示你应该将它们分离出去。子类化的时候你发现某些特性需要以一种方式来子类化，另外些特性需要以另一种方式子类化，这就意味着你需要分解原来的类。
<h5 id="4--内联类">4. 内联类</h5>
如果一个类不再承担足够责任、不再有单独存在的理由，就挑选这个“萎缩类”的最频繁的用户也是个类），以内联类的手法将“萎缩类”放进另一个类中。

<h5 id="5--隐藏委托关系">5. 隐藏委托关系</h5>
如果某个客户先通过服务对象的字段得到另一个对象，然后调用后者的函数，那么客户就必须知这一层委托关系。万一委托关系发生变化，客户也得相应变化。你可以在服务对象上放置一个简单的托函数，将委托关系隐藏起来，从而去除这种依赖。这么一来，即便将来发生委托关系上的变化，变也将被限制在服务对象中，不会波及客户。
<h5 id="6--移除中间人">6. 移除中间人</h5>
某个类做了过多的简单委托动作，让客户直接调用受托类。
<h5 id="7--引入外加函数">7. 引入外加函数</h5>
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；在客户端建立一个函数，并以参数式传入服务类。
<h5 id="8--引入本地扩展">8. 引入本地扩展</h5>
当一个类需要太多外加函数时，新建一个类容纳这些外加函数，并让这个类成为源类的包装类或子类。
<hr>
<h3 id="重新组织数据">重新组织数据</h3>
<h5 id="1--自封装数据">1. 自封装数据</h5>
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。比如在某些情况下，需要对值的获取或者设置做特殊操作时，get\set方法就显得很灵活。
<h5 id="2--以对象取代数据值">2. 以对象取代数据值</h5>
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；将数据项与其他数据以及行为封装对象。
<h5 id="3--将值对象改为引用对象">3. 将值对象改为引用对象</h5>
如果你有多个类，这些类都使用相同的类的不同实例作为成员变量。现在你希望这些类都引用相的类的相同的实例作为成员变量，就可以使用本方法进行重构。
<h5 id="4--将引用对象改为值对象">4. 将引用对象改为值对象</h5>
如果你有一个引用对象,很小且不可改变，而且不易管理，你就需要考虑将他改为一个值对象。这能也是在领域驱动设计实现一书当中推荐使用值对象的原因之一。
<h5 id="5--复制被监视数据">5. 复制被监视数据</h5>
针对 MVC 中 V 的数据，复制到 M 中处理好之后再通过 Listener 机制或者 Obsever 模式同步新 V。
<h5 id="6--将单向关联改为双向关联">6. 将单向关联改为双向关联</h5>
两个类都需要互相使用对象的属性，但之间只有一条单向连接。添加一个反向连接，并使修改函能够同时更新两条连接。
<h5 id="7--将双向关联改为单向关联">7. 将双向关联改为单向关联</h5>
两个类互相引用，但其中一个类并不需要引用另一个类。
<h5 id="8--以字面常量取代魔法数">8. 以字面常量取代魔法数</h5>
魔法数经常难以捉摸含义，应该使用一个字面常量去取代它。
<h5 id="9--封装字段">9. 封装字段</h5>
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。简而言之就是不直接暴露对象内的字段。
<h5 id="10--封装集合">10. 封装集合</h5>
有个函数返回集合：让这个函数返回集合的只读副本，并在这个类中提供添加/移除集合元素的数。主要目的是函数无副作用，比如 BigDecimal 等类返回的就是副本。
<h5 id="11--以数据类取代记录">11. 以数据类取代记录</h5>
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。哑即无行为，这种法常见于数据库的映射对象。
<h5 id="12--以类取代类型码">12. 以类取代类型码</h5>
善用枚举。
<h5 id="13--以子类取代类型码">13. 以子类取代类型码</h5>
如果对象需要依据类型码来进行不同的行为，此时利用子类的多态来取代。
<h5 id="14--以状态-策略模式取代类型码">14. 以状态、策略模式取代类型码</h5>
当类不便于利用继承产生多态时，可利用状态或者策略模式来取代类型码。
<h5 id="15--以字段取代子类">15. 以字段取代子类</h5>

你的各个子类的唯一差别只在“返回常量数据”的函数上。修改这些函数，使他们返回超类的某新增字段，然后销毁子类。
<hr>
<h3 id="简化条件表达式">简化条件表达式</h3>
<h5 id="1--分解条件表达式">1. 分解条件表达式</h5>
你有一个复杂的条件语句；从 if,then,else 三个段落中分别提炼出独立函数。
<h5 id="2--合并条件表达式">2. 合并条件表达式</h5>
你有一系列条件表达式，都会返回相同的结果，将合并后的表达式提炼成一个独立函数。前提是些表达式都是相关的。
<h5 id="3--合并重复的条件片段">3. 合并重复的条件片段</h5>
在条件表达式的每个分支上有着一段相同的代码，将这段代码移出条件表达式之外。
<h5 id="4--移除控制标记">4. 移除控制标记</h5>
不必严格遵守单一出口原则（函数内仅有一个 return）,不用通过控制标记来决定是否退出循环者跳过，直接 break 或者 return。
<h5 id="5--以卫语句取代嵌套条件表达式">5. 以卫语句取代嵌套条件表达式</h5>
条件表达式有两种表现形式。一：所有分支都是正常情况。二：只有一种是正常情况，其他都是正常情况。对于一的情况，按照 if-else 的条件表达式。对于二的情况，如果某个条件表达式不常见应该独立检查该条件，并及时返回。这样的单独检查称为卫语句。
<h5 id="6--以多态取代条件表达式">6. 以多态取代条件表达式</h5>
在条件表达式中通过类型来决定不同的行为，此时应利用多态来替换条件表达式。
<h5 id="7--引入Null对象">7. 引入 Null 对象</h5>
当执行一些操作时，需要再三检查某对象是否为 null，可以专门新建一个 Null 对象。可以参考 Java 8 中的 Optional 类。
<h5 id="8--引入断言">8. 引入断言</h5>
断言是一个条件表达式，应该总是为真。如果它失败了，表示程序员犯了错误。因此断言的失败该导致一个非受控异常。某一段代码需要对程序状态作出某种假设；以断言明确表现这种假设。
<hr>
<h3 id="简化函数调用">简化函数调用</h3>
<h5 id="1--函数改名">1. 函数改名</h5>
给函数取一个见名知其意的名字，使得代码的可读性提高。
<h5 id="2--添加参数">2. 添加参数</h5>
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，让该对象带进函数所需信息。
<h5 id="3--消除参数">3. 消除参数</h5>
当函数不再需要某个参数时，果断移除，不要为了一些未知的需求预留参数，导致过度设计。参多的函数给调用者造成困扰。
<h5 id="4--将查询函数和修改函数分离">4. 将查询函数和修改函数分离</h5>
如果某个函数既返回对象，又修改对象。此时应该创建两个函数，一个负责查询，一个负责修改做到函数的无副作用。
<h5 id="5--令函数携带参数">5. 令函数携带参数</h5>
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，以参数表达那些同的值。例如 getTenPersentIncome 方法和 getFivePercentIncome 应合并为 getIncome(percent 的形式。
<h5 id="6--以明确函数取代参数">6. 以明确函数取代参数</h5>
有一个函数，完全取决于参数值而采取不同的行为。针对该参数设计不同的独立函数。
<h5 id="7--保持对象完整">7. 保持对象完整</h5>
从某个对象取出若干值作为函数的参数改成直接传入这个对象。到时候参数列表变化的话就不必改了。（但这会破坏迪米特法则）。
<h5 id="8--以函数取代参数">8. 以函数取代参数</h5>
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。如果函数可以通过其他途径得参数值，那么它就不应该通过参数取得该值。过长的参数列会增加程序阅读者的理解难度，因此应尽可能缩短参数列的长度。
<h5 id="9--引入参数对象">9. 引入参数对象</h5>

当一个方法的参数超过 3 个以上，就可以考虑将参数封装成一个对象。将参数封装成对象后提高代码的可读性，并且该参数对象也可以供多个方法调用，以后如果增加删除参数，方法本身不需要修，只需要修改参数对象就可以。（比如函数的参数有 10 个，可以将相关联参数的合并成参数对象，得参数数量减少）。
<h5 id="10--移除设值函数">10. 移除设值函数</h5>
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。
<h5 id="11--隐藏函数">11. 隐藏函数</h5>
有一个函数，从来没有被其它任何类用到。将这个函数改为 private。
<h5 id="12--以工厂函数取代构造函数">12. 以工厂函数取代构造函数</h5>
你希望在创建对象时不仅仅是做简单的构造动作；将构造函数替换为工厂函数。可以参考设计模中的不同工厂模式。
<h5 id="13--封装向下转型">13. 封装向下转型</h5>
某个函数返回的对象，需要由函数调用者执行向下转型（downcast）。将向下转型动作移到函中。
<h5 id="14--以异常取代错误码">14. 以异常取代错误码</h5>
某个函数返回一个特定的错误码，用以表示某种错误情况。改用异常。有的文章会说异常的性能好，但牺牲一点微乎其微的性能（因为异常都是非正常情况），来实现代码流程的流畅是值得的。</
>
<h5 id="15--以测试取代异常">15. 以测试取代异常</h5>
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检查。在函数调用点之前，放置一个测试语句。
<hr>
<h3 id="处理概况关系">处理概况关系</h3>
<h5 id="1--字段上移">1. 字段上移</h5>
两个子类拥有相同的字段，将字段移至父类。
<h5 id="2--函数上移">2. 函数上移</h5>
有些函数在各个子类中产生完全相同的结果，上移至父类中消除重复并方便修改。
<h5 id="3--构造函数上移-你在各个子类中拥有一些构造方法-它们的本体几乎完全一致-在超类中新一个构造函数-并在子类构造函数中调用它-">3. 构造函数上移：你在各个子类中拥有一些构造方法，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。</h5>
<h5 id="4--函数下移">4. 函数下移</h5>
父类中的函数只与部分子类相关，将函数移至相关的子类中。
<h5 id="5--字段下移">5. 字段下移</h5>
父类中的某些字段只被部分子类用到，将这个字段移到相关的子类中去。
<h5 id="6--提炼子类">6. 提炼子类</h5>
类中的某些特性只被某些实例用到；新建一个子类，将上面所说的那一部分特性移到子类中。</
>
<h5 id="7--提炼父类">7. 提炼父类</h5>
两个类有相似的特性，为这两个类建立一个父类，将相同特性移至父类中。
<h5 id="8--提炼接口">8. 提炼接口</h5>
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；将相同的子类提炼到一个独接口中。
<h5 id="9--折叠继承体系">9. 折叠继承体系</h5>
父类与子类没有太大的区别，可能是因为之前 的过度设计。将它们合为一体。
<h5 id="10--塑造模板函数">10. 塑造模板函数</h5>
可以参考设计模式中的模板方法模式。
<h5 id="11--以委托取代继承">11. 以委托取代继承</h5>
某个子类只使用父类接口中的一部分，或是根本不需要继承而来的数据，将父类作为子类的一个性，将需要用到的函数委托给这个属性来执行。（以组合取代继承的例子）
<h5 id="12--以继承取代委托">12. 以继承取代委托</h5>
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。让委托类直接继受托类。