代码重构新手教程：如何将烂代码变成好代码?

> 作为有几年工作经验的程序员，都会对 bad code 不满意。如何将烂代码变成好代码，本文将由浅入深、一步步带你理解重构的奥秘，让你对重构有个基本的了解。本文基于文章《The Simple Ways to Refactor Terrible Code》编译整理而成。

任何一个有几年工作经验的程序员都经历过这样的场景：回顾早期写的代码，会陷入深深的怀疑，这么烂的代码是我写的吗？相比起刚入行时候的你，这几年不管是自己发奋学习，如阅读《编程模式》、《重构：改善现有代码的设计》等圣经级著作，还是公司大神对你耳提面命，你对程序架构、编码规范的认识都有了很大的提升。还有一种情况是做项目的时候，或多或少存在赶工期的问题，很可能为了能在 deadline 之前交付，某个问题的解决方案并不够优雅。

以每一个有责任心（代码洁癖）的程序员都会去考虑重构的问题，重构代码有很多好处，正如作者在文章中提到的： 后期修正 bug 更加容易、提高程序的可读性、改进程序原有设计，更重要的是当你在小组讨论会上，向同事展示代码时不会觉得丢人。以上这些好处是彼此联系的，比如当你接手一个遗留工程，前开发人员早已不知去向，不管是要增加功能还是修正 bug，你都需要读懂代码，这就需要提高程序的可读性，你能依靠的除了你堪比福尔摩斯的推理能力，就只有重构这把杀猪刀了。

虽然重构有这么多好处，为什么当我们准备开始的时候，却会反复纠结？作者大致提到以下原因：

担心破坏已有代码。这种情况在核心业务系统尤为普遍，比如电商平台，企业的 ERP 平台等，系统需要 7*24 运行，你的一个修改可能导致 10000 元的商品被 1 元钱买走了。作为普通人的我们，自然会抱着多一事不如少一事的心理，毕竟出了问题，会吃不了兜着走。

不能立即看到产出。在每日的 stand-up 会议中，当研发经理问我，你今天干什么了，我说重构代码，如果连续三天都是这个答案，估计研发经理就要发飙了。

没有时间去做。面对繁重的研发任务和日益逼近的 deadline，重构，真的不是一场说走就走的旅行。

说了这么多，读者朋友可能会有一个想法，是否有一些方法，能让我享受重构的好处，又能避免上面提到的风险。

幸运的是还真有！

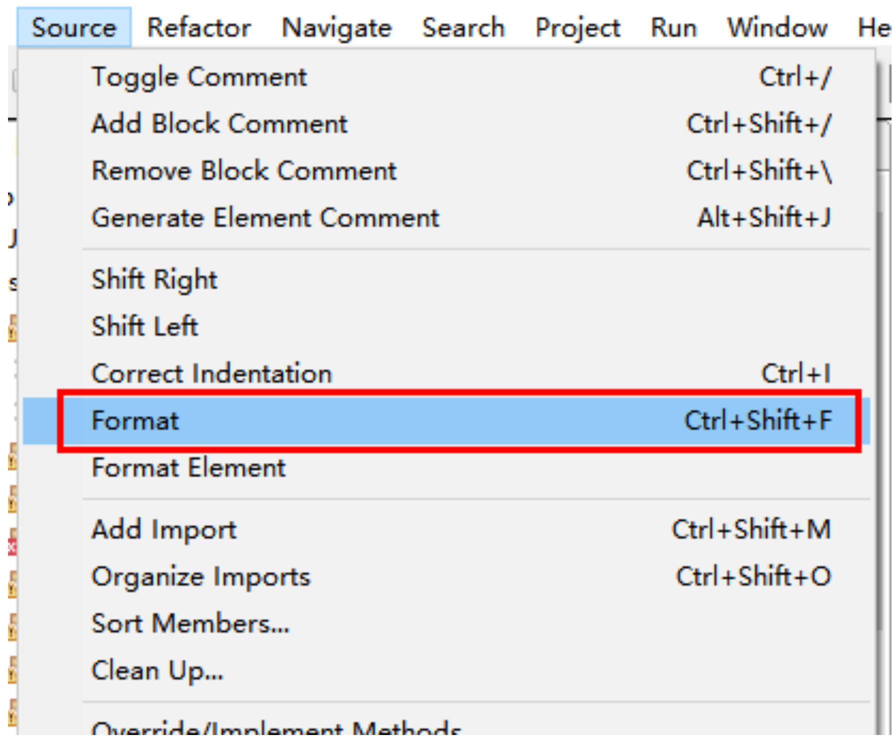
下面我将从最简单、基本不会破坏已有代码、花费很少时间的重构方法入手，逐步深入，让大家对重构有一个基本了解，在对方法的介绍中，我将按照《InfoQ编程语言2月排行榜结果出炉》中的调研情况，选取用户掌握最多的编程语言 Java，以及该语言使用最多的 IDE 环境 Eclipse，进行举例。

重构入门

为了消除恐惧，让我们从最简单的重构方法入手

1. 格式化代码

当你发现代码缩进层次不齐，代码块中缺少{}等问题时，就需要考虑代码格式化了，现在的 IDE 工具已经对格式化提供了很好的支持，以 eclipse 为例，选中要格式化的代码，点击以下菜单项就能完成代码格式化。



此外很多源代码管理网站，也提供了格式化工具，如图所示：

在团队开发中，为了保证开发代码样式统一，需要建立编码规范。我们并不需要重头建立编码规范，可以在大厂的编码规范基础上进行定制，比如在 Java 领域可采用阿里、华为、Google Java Style Guide 等编码规范。该编码规范可以与 IDE 进行结合，如在 eclipse 中，打开 Window->Preferences->Java->Code Style 导入编码规范：

重要的是，不管选择何种规范，要坚持下去，并让每个团队成员都用起来。

2. 注释

在代码开发中，好的注释可以提高程序的可读性，坏的注释可能会

画蛇添足，甚至起反作用。作者提到好的注释要做到和代码相关、及时更新。很多时候，代码刚开始编写时，注释和代码是一致，后期因为间隔时间过长或其他人接手修改代码，没有对注释及时修改，就会造成注释和代码渐行渐远。

尽量减少不必要的注释。如很多函数或者类，如果设计架构清晰，通过命名就能知道他们做什么，注释不是必须的。还有一种情况是暂时不用的代码，很多人会觉得以后会用到，会加个注释，作者给出的建议是删掉它，如果你将来真的用到了，可以到 git（一种代码管理工具）的历史记录中查找。

对于逻辑混乱的代码，如在循环中随意使用 break，复杂的 if 语句嵌套等，你要做的是理清逻辑，重构代码，而不是让注释替你补锅。正如《重构》中提到的“当你感觉需要撰写注释，请先尝试重构，试着让所有注释都变得多余。”

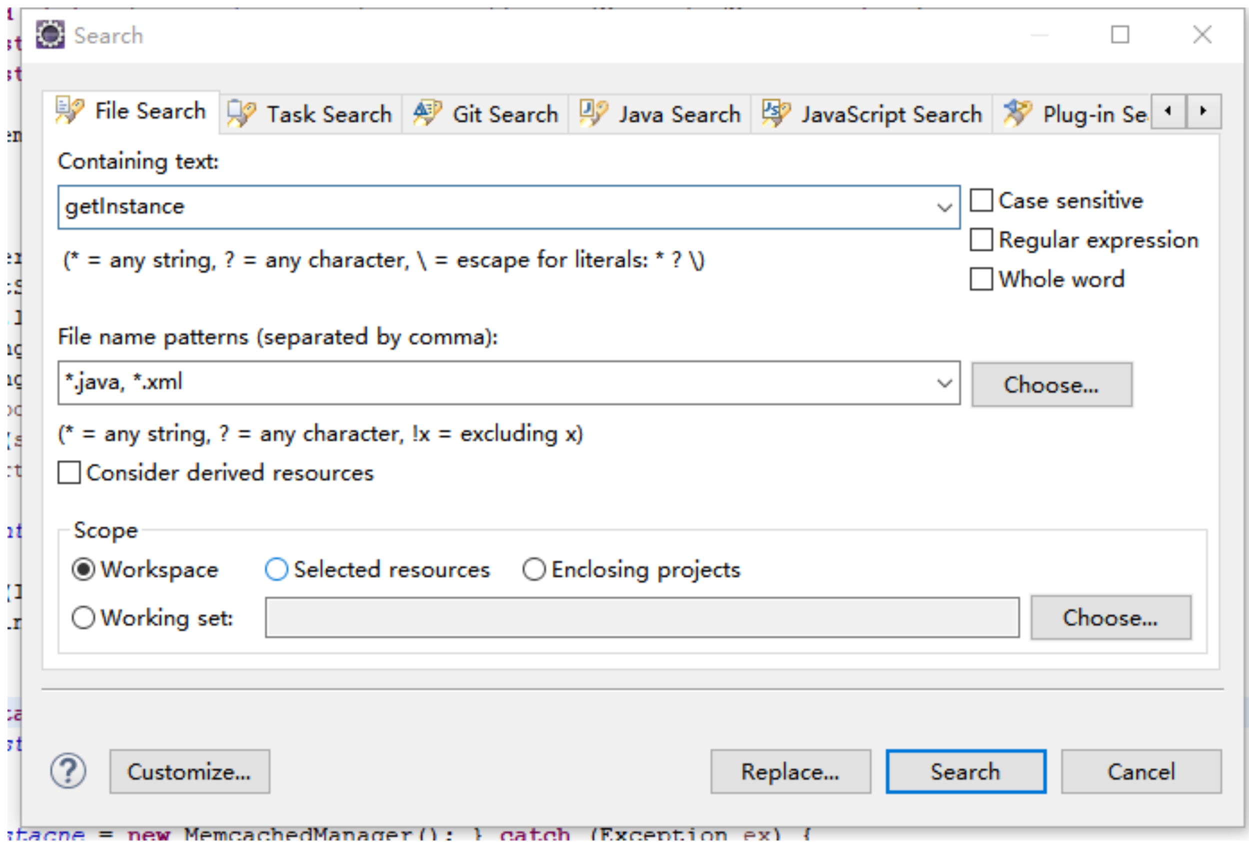
3. 废弃的代码

随着系统版本的不断迭代，有一些函数或类不再使用后，我们应该将它及时删除，否则随着时间流逝，会造成代码库臃肿，程序可读性变差。而且如果还发生人员的变动，慢慢会成为谁也不敢动的代码，因为都不知道有啥用和在哪用到。

多先进的 IDE 工具都对查找代码的调用提供了支持，以 eclipse 为例，查找函数是否被调用，可以使用调用层次图功能，或者直接使用高级搜索功能，如图所示：

在调用层次图（Call Hierarchy）中可以看到 getInstance()函数被什么地方调用。

如果使用类似于 spring 的自动装配功能，在 xml 中定义了调用关系，可以使用高级搜索功能查看 xml 或 properties 文件中定义的同名函数进行筛选。



4. 变量命名

就像我们人一样，一个好名字对变量、常量、函数和类都很重要，一个好的名字会让其他开发人员很容易明白其功能是什么。以下是命名的一些注意事项：

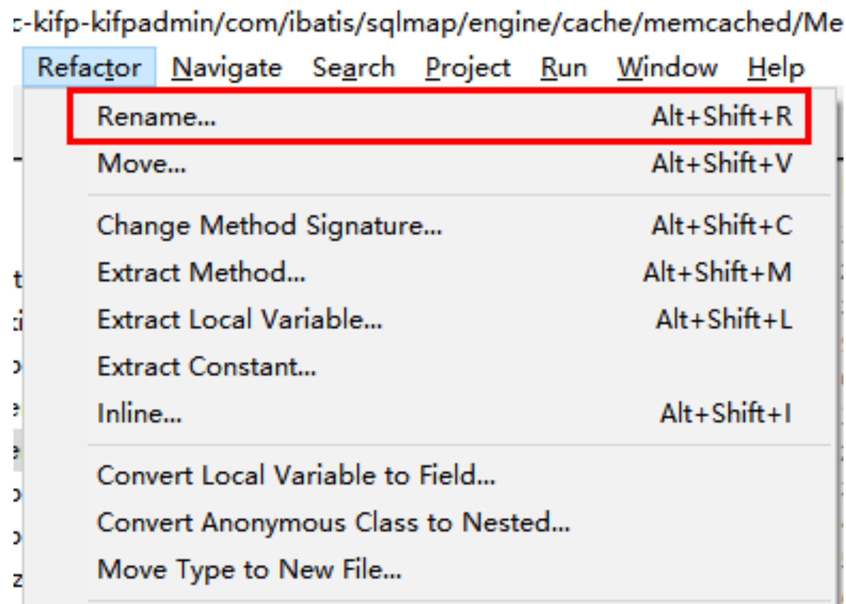
类和文件名使用名词，但这个名词要有意义，比如 Data、Information 就意义不明显，不是好名字。

函数使用动词或短语命名，比如 isReady hasName。

长名字 vs 无意义名字：在长名字和无意义名字中选择时，请选择长且有意义的名字，比如 java 语言中使用最广的类库 spring 中的一个命名是：SimpleBeanFactoryAwareAspectInstanceFactory。

命名法则： 常见的有驼峰命名法（camelCase）和蛇形命名法（snake_case）,比如文件名使用蛇形是 file_name，驼峰式 fileName。选择一种，所有的命名都按照这个规则，并将其作为编码规范的一部分，让团队成员都要遵守。

如果你要对已有代码中错误的命名方式进行修改，eclipse 提供了很好地支持：选择要修改的类、函数或变量，选择 Refactor——》Rename 可以同时修改该变量在声明和使用处的名称，如下图所示：



5. 常量命名

常量的命名除了要遵守上一小节提到的通用方法外，还有一类魔法数字（magical numbers）的情况，如使用 0,1 来代表男女。更恰当的做法是定义常量名来代替魔法数字，如在 Java 中：

```
final int static FEMALE=0,MALE=1;
```

复制代码

6. 负值条件的重构

在条件或循环语句中，使用负值条件，会让代码难以理解、容易出错，比如判断是否为男性，条件写成了 "! isNotFemale(gender)"

重构方法是将条件改成正值，并调换 if/else 语句代码块的顺序。

7. {} 作为单独的一行

正确的{}格式已经在 1 部分中提到，这里再强调下，如果你没有将括号作为单独的一行，如下所示：

```
catch (IOException e) {  e.printstackTrace(); }
```

复制代码

你得到的好处只是减少了一行代码，但是当你设置断点调试时，断点将不能精确定位到你想调试的部分。

8. 变量定义和使用距离太远

变量的定义和使用不要离得太远，一般不要超过 20 行，函数也类似。如果你意识到这个问题，但并不能缩短定义和使用的距离，那代表这是个大函数（big function），你需要对函数做拆分。

以上是对入门级重构方法的介绍，在进行重构时，最重要的规则是：每次只做微小修改，并保证测试能正确运行（小步快跑）。

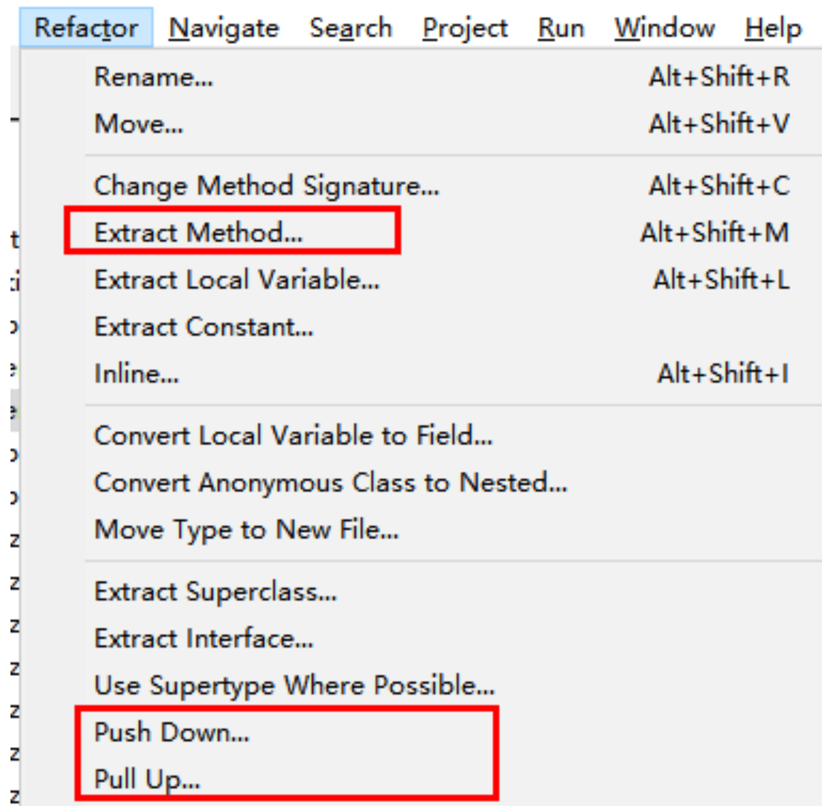
重构进阶

现在我们对重构已经有了基本的了解，并建立了初步的信心。让我们下面关注一些稍微复杂的重构内容。

1. 重复代码

当你发现相同的代码块在三个地方都出现时，你就需要考虑重构代码了。对于同一个类中重复的代码块，可使用提取方法（extract method：将重复代码提取出单独的函数）来完成；对于一组相关类如父类、子类 A、子类 B 中的重复函数，通过上移方法（pull method：将子类中的方法移入父类中）和模板方法（template method：父类方法定义模板，子类编写不同实现）来完成。

Eclipse 提供了相关功能，如图所示：



2. 函数参数

开关参数的滥用（boolean parameters）： 函数的形参中有一个是 boolean 类型，函数体根据该参数为 true 或者 false 执行不同的代码块。这种方式会导致重构的另一个坏味道——大函数（big function）的形成，从而增加代码的复杂性。重构方法是：去掉这个开关参数，将函数拆分成两个函数。

在函数内修改了参数： 参数用于外界向函数体传递信息，好的做法是参数对于函数是只读的。如果在函数内修改参数，会造成函数功能难以理解，如果函数内多次修改参数，这个函数会变成一座迷宫，重构方法是：将参数赋值给局部变量，对局部变量修改，如下代码所示：

原始的：

```
int fun(int val) { val=32; }
```

复制代码

修改后：

```
int fun(int val) {  int temp=val;  temp=32; }
```

复制代码

参数太多：函数的参数最多有三个是合理的，超过三个就需要提高警惕了。重构方法是：根据逻辑拆分函数；引入参数对象（parameter object：构造参数类，将原来传递的参数作为类的属性，调用方传入该类的一个对象）

3. 变量多余

当定义的变量没太多含义，而且没有赋值操作，如下代码：

```
double basePrice = anOrder.basePrice();    return (basePrice >
```

复制代码

其中的 basePrice 完全是多余的变量，完全可以用函数本身来替代它，如下代码：

```
return (anOrder.basePrice() > 1000);
```

复制代码

4. 缺少变量

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。如果它们被赋值超过一次，就意味着它们在函数中承担了一个以上的职责。如果临时变量承担多个责任，它就应该被替换为多个临时变量，每个变量只承担一个责任。

重构方法：针对每次赋值，创造一个独立、对应的临时变量

5. 复杂条件

我们都见过由 && || 构成的复杂的多行条件。复杂条件可读性很差，调试和修改也很麻烦。

一个简单的重构方式是：将这块代码抽取出来，变成一个单独的判断函数，如下代码：

```
double fetchSalary(double money, int day) {      if(money>10000 &&
```

复制代码

其中的条件 money>10000 && day>30 可重构为：

```
boolean isHigherSalary(double money,int day) {         return (mc
```

复制代码

老旧代码的重构

在进行代码重构时，需要考虑测试代码是否能覆盖重构的功能，如果没有，需要增加测试用例覆盖所做的修改，否则重构可能会破坏已有的功能。在前面的章节，作者假设已有足够的测试用例，并且重构完成后测试可以正确运行。

但是如何重构测试用例没有完全覆盖的代码呢，如老旧代码？作者的建议是只做必要的重构，如当需要修正 bug 或者增加新的功能，这种情况下，先为遗留代码编写测试用例，在理解的基础上重构代码，为代码修改做好准备，然后进行代码修改。

从这点上来说，你可以进行任何类型代码的重构：一次只做一步重构，从小的容易的重构做起，并频繁测试。

利用工具

重构代码需要花费时间，当项目工期很紧时，很难下定决心去做重构。为了让重构变得更容易，市面上提供了大量相关工具，如 pylint（ Python 代码分析工具）、Checkstyle（代码规范工具）、Sonarqube（代码质量管理的开源工具）

此外，你要保证你的测试用例跑的足够快，否则你会没有耐心等待测试运行结果，或者直接就不运行了。

理想情况下，程序在构建后部署到测试环境前，可以借助 CI/CD（持续集成/持续部署）工具实现代码质量检查、代码样式检查、潜在 bug 监测等模块的自动化运行。

总结

这篇文章并没有穷尽重构的所有内容，更多的重构清单和实例，请参考鲍勃大叔编写的《代码整洁之道》（Clean Code)，尤其是第 17 章的味道与启发（Smells and Heuristics）和马丁·福勒（Martin Fowler）编写的重构（Refactoring）一书 。不管你打算以哪本书为主，在实践过程中，都会殊途同归——沉淀出几条简单的规则。

不要专门花费大量的时间去进行重构，利用小块时间，每次只做一部分，只要保证代码质量比之前有进步就可以了。不要想着以后再做，这个以后很可能是永远不，最终你将面对一系列可怕的遗留代码，然后你就深刻理解了“出来混迟早是要还的”这句话的涵义。

参考文章：[The Simple Ways to Refactor Terrible Code]