

关于烂代码的那些事（上）

作者: 秦迪 发布时间: 2015-08-12 14:05 阅读: 39943 次 推荐: 104 [原文链接](#) [\[收藏\]](#)

1. 摘要

最近写了不少代码，review了不少代码，也做了不少重构，总之是对着烂代码工作了几周。为了抒发一下这几周里好几次到达崩溃边缘的情绪，我决定写一篇文章谈一谈烂代码的那些事。这里是上篇，谈一谈烂代码产生的原因和现象。

2. 写烂代码很容易

刚入程序员这行的时候经常听到一个观点：你要把精力放在ABCD（需求文档/功能设计/架构设计/理解原理）上，写代码只是把想法翻译成编程语言而已，是一个没什么技术含量的事情。

当时的我在听到这种观点时会有一种近似于高冷的不屑：你们就是一群傻X，根本不懂代码质量的重要性，这么下去迟早有一天会踩坑，呸。

可是几个月之后，他们似乎也没怎么踩坑。而随着编程技术一直在不断发展，带来了更多的我以前认为是傻X的人加入到程序员这个行业中来。

语言越来越高级、封装越来越完善，各种技术都在帮助程序员提高生产代码的效率，依靠层层封装，程序员真的不需要了解一丁点技术细节，只要把需求里的内容逐行翻译出来就可以了？

很多程序员不知道要怎么组织代码、怎么提升运行效率、底层是基于什么原理，他们写出来的是在我心目中烂成一坨翔一样的代码。

但是那一坨翔一样代码竟然他妈的能正常工作。

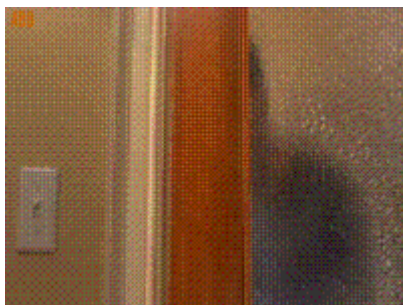
即使我认为他们写的代码是坨翔，但是从从不接触代码的人的视角来看（比如说你的boss），代码编译过了，测试过了，上线运行了一个月都没出问题，你还想要奢求什么？

所以，即使不情愿，也必须承认，时至今日，写代码这件事本身没有那么难了。

3. 烂代码终究是烂代码

但是偶尔有那么几次，写烂代码的人离职之后，事情似乎又变得不一样了。

想要修改功能时却发现程序里充斥着各种无法理解的逻辑，改完之后莫名其妙的bug一个接一个，接手这个项目的人开始漫无目的的加班，并且原本一个挺乐观开朗的人渐渐的开始喜欢问候别人祖宗了。



我总结了几类经常被“祖宗”的烂代码：

3.1. 意义不明

能力差的程序员容易写出意义不明的代码，他们不知道自己究竟在做什么。

就像这样：

```
public void save() {  
    for(int i=0;i<100;i++) {  
        //防止保存失败，重试100次  
        document.save();  
    }  
}
```

对于这类程序员，我一般建议他们转行。

3.2. 不说人话

不说人话是新手最经常出现的问题，直接的表现就是写了一段很简单的代码，其他人却看不懂。

比如下面这段：

```
public boolean getUrl(Long id) {  
    UserProfile up = us.getUser(ms.get(id).getMessage().aid);  
    if (up == null) {  
        return false;  
    }  
    if (up.type == 4 || ((up.id >> 2) & 1) == 1) {  
        return false;  
    }  
    if(Util.getUrl(up.description)) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

很多程序员喜欢简单的东西：简单的函数名、简单的变量名、代码里翻来覆去只用那么几个单词命名；能缩写就缩写、能省略就省略、能合并就合并。这类人写出来的代码里充斥着各种g/s/gos/of/mss之类的全世界没人懂的缩写，或者一长串不知道在做什么的连续调用。

还有很多程序员喜欢复杂，各种宏定义、位运算之类写的天花乱坠，生怕代码让别人一下子看懂了会显得自己水平不够。

简单的说，他们的代码是写给机器的，不是给人看的。

3.3. 不恰当的组织

不恰当的组织是高级一些的烂代码，程序员在写过一些代码之后，有了基本的代码风格，但是对于规模大一些的工程的掌控能力不够，不知道代码应该如何解耦、分层和组织。

这种反模式的现象是经常会看到一段代码在工程里拷来拷去；某个文件里放了一大坨堆砌起来的代码；一个函数堆了几百上千行；或者一个简单的功能七拐八绕的调了几十个函数，在某个难以发现的猥琐的小角

落里默默的调用了某些关键逻辑。



这类代码大多复杂度高，难以修改，经常一改就崩；而另一方面，创造了这些代码的人倾向于修改代码，畏惧创造代码，他们宁愿让原本复杂的代码一步步变得更复杂，也不愿意重新组织代码。当你面对一个几千行的类，问为什么不把某某逻辑提取出来的时候，他们会说：

“但是，那样就多了一个类了呀。”

3.4. 假设和缺少抽象

相对于前面的例子，假设这种反模式出现的场景更频繁，花样更多，始作俑者也更难以自己意识到问题。比如：

```
public String loadString() {  
    File file = new File("c:/config.txt");  
    // read something  
}
```

文件路径变更的时候，会把代码改成这样：

```
public String loadString(String name) {  
    File file = new File(name);  
    // read something  
}
```

需要加载的内容更丰富的时候，会再变成这样：

```
public String loadString(String name) {  
    File file = new File(name);  
    // read something  
}  
public Integer loadInt(String name) {  
    File file = new File(name);  
    // read something  
}
```

之后可能会再变成这样：

```
public String loadString(String name) {  
    File file = new File(name);  
    // read something  
}  
public String loadStringUtf8(String name) {  
    File file = new File(name);
```

```
        // read something
    }
    public Integer loadInt(String name) {
        File file = new File(name);
        // read something
    }
    public String loadStringFromNet(String url) {
        HttpClient ...
    }
    public Integer loadIntFromNet(String url) {
        HttpClient ...
    }
}
```

这类程序员往往是项目组里开发效率比较高的人，但是大量的业务开发工作导致他们不会做多余的思考，他们的口头禅是：“我每天要做XX个需求”或者“先做完需求再考虑其他的吧”。

这种反模式表现出来的后果往往是代码很难复用，面对deadline的时候，程序员迫切的想要把需求落实成代码，而这往往也会是个循环：写代码的时候来不及考虑复用，代码难复用导致之后的需求还要继续写大量的代码。

一点点积累起来的大量的代码又带来了组织和风格一致性问题，最后形成了一个新功能基本靠拷的遗留系统。

3.5. 还有吗

烂代码还有很多种类型，沿着功能-性能-可读-可测试-可扩展这条路线走下去，还能看到很多匪夷所思的例子。

那么什么是烂代码？个人认为，烂代码包含了几个层次：

- 如果只是一个人维护的代码，满足功能和性能要求倒也足够了。
- 如果在一个团队里工作，那就必须易于理解和测试，让其他人员有能力修改各自的代码。
- 同时，越是处于系统底层的代码，扩展性也越重要。

所以，当一个团队里的底层代码难以阅读、耦合了上层的逻辑导致难以测试、或者对使用场景做了过多的假设导致难以复用时，虽然完成了功能，它依然是坨翔一样的代码。

3.6. 够用的代码

而相对的，如果一个工程的代码难以阅读，能不能说这个是烂代码？很难下定义，可能算不上好，但是能说它烂吗？如果这个工程自始至终只有一个人维护，那个人也维护的很好，那它似乎就成了“够用的代码”。

很多工程刚开始可能只是一个人负责的小项目，大家关心的重点只是代码能不能顺利的实现功能、按时完工。

过上一段时间，其他人参与时才发现代码写的有问题，看不懂，不敢动。需求方又开始催着上线了，怎么办？只好小心翼翼的只改逻辑而不动结构，然后在注释里写上这么实现很ugly，以后明白内部逻辑了再重构。

再过上一段时间，有个相似的需求，想要复用里面的逻辑，这时才意识到代码里做了各种特定场景的专用逻辑，复用非常麻烦。为了赶进度只好拷代码然后改一改。问题解决了，问题也加倍了。

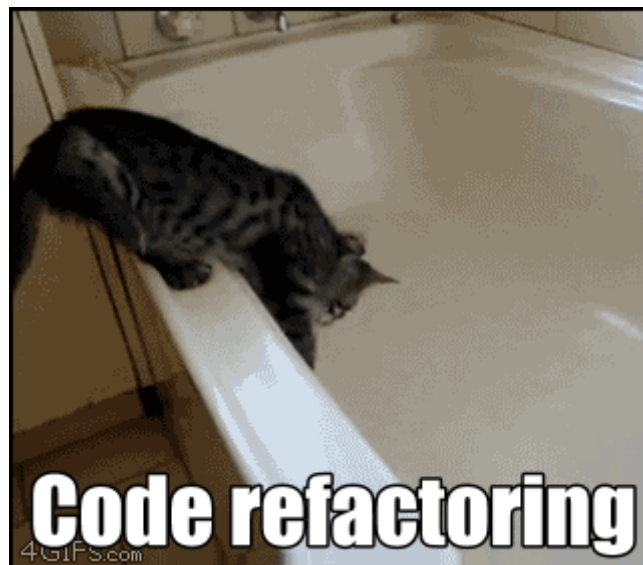
几乎所有的烂代码都是从“够用的代码”演化来的，代码没变，使用代码的场景发生变了，原本够用的代码不符合新的场景，那么它就成了烂代码。

4. 重构不是万能药

程序员最喜欢跟程序员说的谎话之一就是：现在进度比较紧，等X个月之后项目进度宽松一些再去做重构。

不能否认在某些（极其有限的）场景下重构是解决问题的手段之一，但是写了不少代码之后发现，重构往往是程序开发过程中最复杂的工作。花一个月写的烂代码，要花更长的时间、更高的风险去重构。

曾经经历过几次忍无可忍的大规模重构，每一次重构之前都是找齐了组里的高手，开了无数次分析会，把组内需求全部暂停之后才敢开工，而重构过程中往往哀嚎遍野，几乎每天都会出上很多意料之外的问题，上线时也几乎必然会出几个问题。



从技术上来说，重构复杂代码时，要做三件事：理解旧代码、分解旧代码、构建新代码。而待重构的旧代码往往难以理解；模块之间过度耦合导致牵一发而动全身，不易控制影响范围；旧代码不易测试导致无法保证新代码的正确性。

这里还有一个核心问题，重构的复杂度跟代码的复杂度不是线性相关的。比如有1000行烂代码，重构要花1个小时，那么5000行烂代码的重构可能要花2、3天。要对一个失去控制的工程做重构，往往还不如重写更有效率。

而抛开具体的重构方式，从受益上来说，重构也是一件很麻烦的事情：它很难带来直接受益，也很难量化。这里有个很有意思的现象，基本关于重构的书籍无一例外的都会有独立的章节介绍“如何向boss说明重构的必要性”。

重构之后能提升多少效率？能降低多少风险？很难答上来，烂代码本身就不是一个可以简单的标准化的东西。

举个例子，一个工程的代码可读性很差，那么它会影响多少开发效率？

你可以说：之前改一个模块要3天，重构之后1天就可以了。但是怎么应对“不就是做个数据库操作吗，为什么要3天”这类问题？烂代码“烂”的因素有不确定性，开发效率也因人而异，想要证明这个东西“确实”会增加2天开发时间，往往反而会变成“我看了3天才看懂这个函数是做什么的”或者“我做这么简单的修改要花3天”这种神经病才会去证明的命题。

而另一面，许多技术负责人也意识到了代码质量和重构的必要性，“那就重构嘛”，或者“如果看到问题了，那就重构”。上一个问题解决了，但实际上关于重构的代价和收益仍然是一笔糊涂账，在没有分配给你更多资源、没有明确的目标、没有具体方法的情况下，很难想象除了有代码洁癖的人还有谁会去执行这种莫名其妙的任务。

于是往往就会形成这种局面：

- 不写代码的人认为应该重构，重构很简单，无论新人还是老人都有责任做重构。
- 写代码老手认为应该迟早应该重构，重构很难，现在凑合用，这事别落在我头上。
- 写代码的新手认为不出bug就谢天谢地了，我也不知道怎么重构。

5. 写好代码很难

与写出烂代码不同的是，想写出好代码有很多前提：

- 理解要开发的功能需求
- 了解程序的运行原理
- 做出合理的抽象
- 组织复杂的逻辑
- 对自己开发效率的正确估算
- 持续不断的练习

写出好代码的方法论很多，但我认为写出好代码的核心反而是听起来非常low的“持续不断的练习”。这里就不展开了，留到下篇再说。

很多程序员在写了几年代码之后并没有什么长进，代码仍然烂的让人不忍直视，原因有两个主要方面：

- 环境是很重要的因素之一，在烂代码的熏陶下很难理解什么是好代码，知道的人大部分也会选择随波逐流。
- 还有个人性格之类的说不清道不明的主观因素，写出烂代码的程序员反而都是一些很好相处的人，他们往往热爱公司团结同事平易近人工作任劳任怨-只是代码很烂而已。

而工作几年之后的人很难再说服他们去提高代码质量，你只会反复不断的听到：“那又有什么用呢？”或者“以前就是这么做的啊？”之类的说法。

那么从源头入手，提高招人时对代码的质量的要求怎么样？

前一阵面试的时候增加了白板编程，最近又增加了上机编程的题目。发现了一个现象：一个人工作了几年、做过很多项目、带过团队、发了一些文章，不一定能代表他代码写的好；反之，一个人代码写的好，其它方面的能力一般不会太差。

举个例子，最近喜欢用“写一个代码行数统计工具”作为面试的上机编程题目。很多人看到题目之后第一反映是，这道题太简单了，这不就是写写代码嘛。

从实际效果来看，这道题识别度却还不错。

首先，题目足够简单，即使没有看过《面试宝典》之类书的人也不会吃亏。而题目的扩展性很好，即使提前知道题目，配合不同的条件，可以变成不同的题目。比如要求按文件类型统计行数、或者要求提高统计效率、或者统计的同时输出某些单词出现的次数，等等。

从考察点来看，首先是基本的树的遍历算法；其次有一定代码量，可以看出程序员对代码的组织能力、对问题的抽象能力；上机编码可以很简单的看出应聘者是不是很久没写程序了；还包括对于程序易用性和性能的理解。

最重要的是，最后的结果是一个完整的程序，我可以按照日常工作的标准去评价程序员的能力，而不是从十几行的函数里意淫这个人在日常工作中大概会有什么表现。

但即使这样，也很难拍着胸脯说，这个人写的代码质量没问题。毕竟面试只是代表他有写出好代码的能力，而不是他将来会写出好代码。

6. 悲观的结语

说了那么多，结论其实只有两条，作为程序员：

- 不要奢望其他人会写出高质量的代码
- 不要以为自己写出来的是高质量的代码

如果你看到了这里还没有丧失希望，那么可以期待一下这篇文章的第二部分，关于如何提高代码质量的一些建议和方法。

关于烂代码的那些事（中）

作者: 秦迪 发布时间: 2015-08-13 22:36 阅读: 33911 次 推荐: 83 [原文链接](#) [\[收藏\]](#)

1. 摘要

这是烂代码系列的第二篇，在文章中我会跟大家讨论一下如何尽可能高效和客观的评价代码的优劣。

在发布了关于[烂代码的那些事（上）](#)之后，发现这篇文章竟然意外的很受欢迎，很多人也描(tu)述(cao)了各自代码中这样或者那样的问题。

最近部门在组织bootcamp，正好我负责培训代码质量部分，在培训课程中让大家花了不少时间去讨论、改进、完善自己的代码。虽然刚毕业的同学对于代码质量都很用心，但最终呈现出来的质量仍然没能达到“十分优秀”的程度。究其原因，主要是不了解好的代码“应该”是什么样的。

2. 什么是好代码

写代码的第一步是理解什么是好代码。在准备bootcamp课程的时候，我就为这个问题犯了难，我尝试着用一些精确的定义区分出“优等品”、“良品”、“不良品”，但是在总结的过程中，关于“什么是好代码”的描述却大多没有可操作性

2.1. 好代码的定义

随便从网上搜索了一下“优雅的代码”，找到了下面这样的定义：

Bjarne Stroustrup, C++之父：

- 逻辑应该是清晰的，bug难以隐藏；
- 依赖最少，易于维护；
- 错误处理完全根据一个明确的策略；
- 性能接近最佳化，避免代码混乱和无原则的优化；
- 整洁的代码只做一件事。

Grady Booch, 《面向对象分析与设计》作者：

- 整洁的代码是简单、直接的；
- 整洁的代码，读起来像是一篇写得很好的散文；
- 整洁的代码永远不会掩盖设计者的意图，而是具有少量的抽象和清晰的控制行。

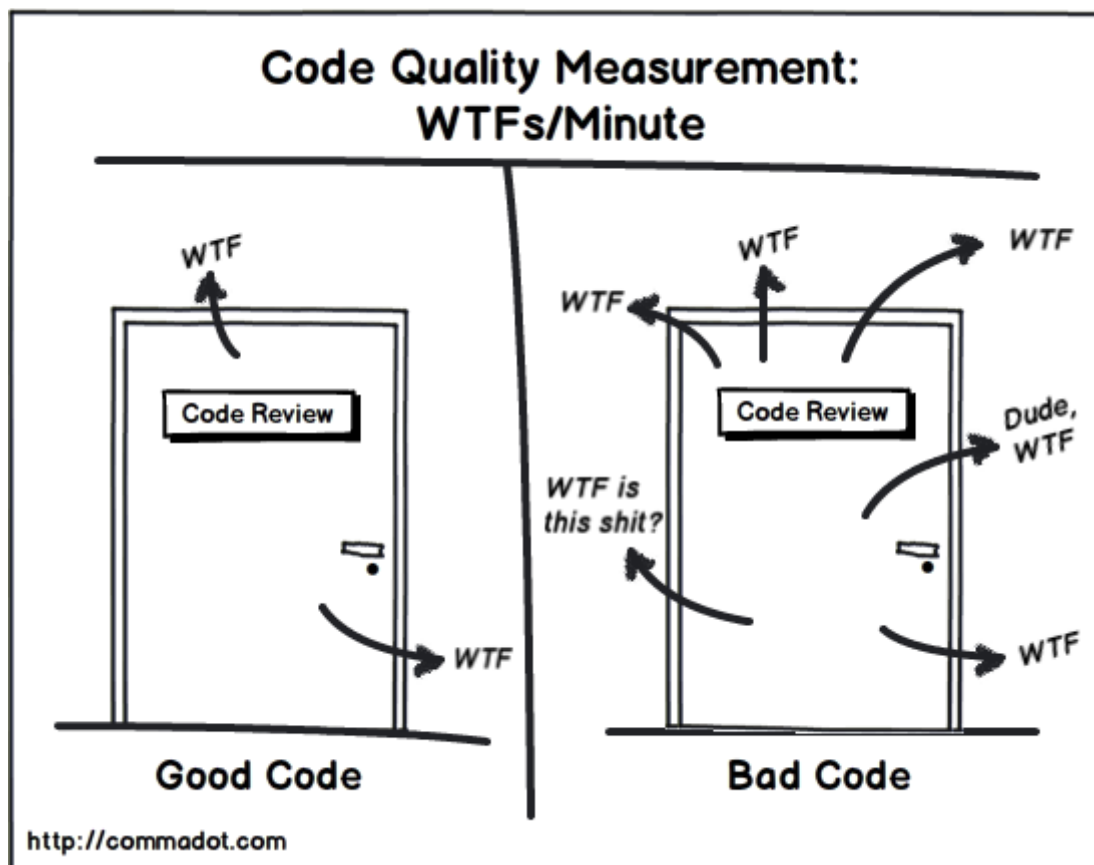
Michael Feathers, 《修改代码的艺术》作者：

- 整洁的代码看起来总是像很在乎代码质量的人写的；
- 没有明显的需要改善的地方；
- 代码的作者似乎考虑到了所有的事情。

看起来似乎说的都很有道理，可是实际评判的时候却难以参考，尤其是对于新人来说，如何理解“简单的、直接的代码”或者“没有明显的需要改善的地方”？

而实践过程中，很多同学也确实面对这种问题：对自己的代码总是处在一种心里不踏实的状态，或者是自己觉得很好了，但是却被其他人认为很烂，甚至有几次我和新同学因为代码质量的标准一连讨论好几天，却谁也说服不了谁：我们都坚持自己对于好代码的标准才是正确的。

在经历了无数次code review之后，我觉得这张图似乎总结的更好一些：



代码质量的评价标准某种意义上有点类似于文学作品，比如对小说的质量的评价主要来自于它的读者，由个体主观评价形成一个相对客观的评价。并不是依靠字数，或者作者使用了哪些修辞手法之类的看似完全客观但实际没有什么意义的评价手段。

但代码和小说还有些不一样，它实际存在两个读者：计算机和程序员。就像上篇文章里说的，即使所有程序员都看不懂这段代码，它也是可以被计算机理解并运行的。

所以对于代码质量的定义我需要于从两个维度分析：主观的，被人类理解的部分；还有客观的，在计算机里运行的状况。

既然存在主观部分，那么就会存在个体差异，对于同一段代码评价会因为看代码的人的水平不同而得出不一样的结论，这也是大多数新人面对的问题：他们没有一个可以执行的评价标准，所以写出来的代码质量也很难提高。

有些介绍代码质量的文章讲述的都是倾向或者原则，虽然说的很对，但是实际指导作用不大。所以在这篇文章里我希望尽可能把评价代码的标准用（我自认为）与实际水平无关的评价方式表示出来。

2.2. 可读的代码

在权衡很久之后，我决定把可读性的优先级排在前面：一个程序员更希望接手一个有bug但是看得懂的工程，还是一个没bug但是看不懂的工程？如果是后者，可以直接关掉这个网页，去做些对你来说更有意义的事情。

2.2.1. 逐字翻译

在很多跟代码质量有关的书里都强调了一个观点：程序首先是给人看的，其次才是能被机器执行，我也比较认同这个观点。在评价一段代码能不能让人看懂的时候，我习惯让作者把这段代码逐字翻译成中文，试着组成句子，之后把中文句子读给另一个人没有看过这段代码的人听，如果另一个人能听懂，那么这段代码的可读性基本就合格了。

用这种判断方式的原因很简单：其他人在理解一段代码的时候就是这么做的。阅读代码的人会一个词一个词的阅读，推断这句话的意思，如果仅靠句子无法理解，那么就需要联系上下文理解这句代码，如果简单的联系上下文也理解不了，可能还要掌握更多其它部分的细节来帮助推断。大部分情况下，理解一句代码在做什么需要联系的上下文越多，意味着代码的质量越差。

逐字翻译的好处是能让作者能轻易的发现那些只有自己知道的、没有体现在代码里的假设和可读性陷阱。无法从字面意义上翻译出原本意思的代码大多都是烂代码，比如“ms代表messageService”，或者“ms.proc()是发消息”，或者“tmp代表当前的文件”。

2.2.2. 遵循约定

约定包括代码和文档如何组织，注释如何编写，编码风格的约定等等，这对于代码未来的维护很重要。对于遵循何种约定没有一个强制的标准，不过我更倾向于遵守更多人的约定。

与开源项目保持风格一致一般来说比较靠谱，其次也可以遵守公司内部的编码风格。但是如果公司内部的编码风格和当前开源项目的风格冲突比较严重，往往代表着这个公司的技术倾向于封闭，或者已经有些跟不上节奏了。

但是无论如何，遵守一个约定总比自己创造出一些规则要好很多，这降低了理解、沟通和维护的成本。如果一个项目自己创造出了一些奇怪的规则，可能意味着作者看过的代码不够多。

一个工程是否遵循了约定，往往需要代码阅读者有一定经验，或者需要借助checkstyle这样的静态检查工具。如果感觉无处下手，那么大部分情况下跟着google做应该不会有什麼大问题：可以参考[google code style](#)，其中一部分有对应的[中文版](#)。

另外，没有必要纠结于遵循了约定到底有什么收益，就好像走路是靠左好还是靠右好一样，即使得出了结论也没有什麼意义，大部分约定只要遵守就可以了。

2.2.3. 文档和注释

文档和注释是程序很重要的部分，他们是理解一个工程或项目的途径之一。两者在某些场景下定位会有些重合或者交叉（比如javadoc实际可以算是文档）。

对于文档的标准很简单，能找到、能读懂就可以了，一般来说我比较关心这几类文档：

- 对于项目的介绍，包括项目功能、作者、目录结构等，读者应该能3分钟内大致理解这个工程是做什么的。

- 针对新人的QuickStart，读者按照文档说明应该能在1小时内完成代码构建和简单使用。
- 针对使用者的详细说明文档，比如接口定义、参数含义、设计等，读者能通过文档了解这些功能（或接口）的使用方法。

有一部分注释实际是文档，比如之前提到的javadoc。这样能把源码和注释放在一起，对于读者更清晰，也能简化不少文档的维护的工作。

还有一类注释并不作为文档的一部分，比如函数内部的注释，这类注释的职责是说明一些代码本身无法表达的作者在编码时的思考，比如“为什么这里没有做XXX”，或者“这里要注意XXX问题”。

一般来说我首先会关心注释的数量：函数内部注释的数量应该不会有很多，也不会完全没有，个人的经验值是滚动几屏幕看到一两处左右比较正常。过多的话可能意味着代码本身的可读性有问题，而如果一点都没有可能意味着有些隐藏的逻辑没有说明，需要考虑适当的增加一点注释了。

其次也需要考虑注释的质量：在代码可读性合格的基础上，注释应该提供比代码更多的信息。文档和注释并不是越多越好，它们可能会导致维护成本增加。关于这部分的讨论可以参考简洁部分的内容。

2.2.4. 推荐阅读

《代码整洁之道》

2.3. 可发布的代码

新人的代码有一个比较典型的特征，由于缺少维护项目的经验，写的代码总会有很多考虑不到的地方。比如说测试的时候似乎没什么异常，项目发布之后才发现有很多意料之外的状况；而出了问题之后不知道从哪下手排查，或者仅能让系统处于一个并不稳定的状态，依靠一些巧合勉强运行。

2.3.1. 处理异常

新手程序员普遍没有处理异常的意识，但代码的实际运行环境中充满了异常：服务器会死机，网络会超时，用户会胡乱操作，不怀好意的人会恶意攻击你的系统。

我对一段代码异常处理能力的第一印象来自于单元测试的覆盖率。大部分异常难以在开发或者测试环境里复现，即使有专业的测试团队也很难在集成测试环境中模拟所有的异常情况。

而单元测试可以比较简单的模拟各种异常情况，如果一个模块的单元测试覆盖率连50%都不到，很难想象这些代码考虑了异常情况下的处理，即使考虑了，这些异常处理的分支都没有被验证过，怎么指望实际运行环境中出现问题时表现良好呢？

2.3.2. 处理并发

我收到的很多简历里都写着：精通并发编程/熟悉多线程机制，诸如此类，跟他们聊的时候也说的头头是道，什么锁啊互斥啊线程池啊同步啊信号量啊一堆一堆的名词滔滔不绝。而给应聘者一个实际场景，让应聘者写一段很简单的并发编程的小程序，能写好的却不多。

实际上并发编程也确实很难，如果说写好同步代码的难度为5，那么并发编程的难度可以达到100。这并不是危言耸听，很多看似稳定的程序，在面对并发场景的时候仍然可能出现问题：比如最近我们就碰到了一个linux kernel在调用某个系统函数时由于同步问题而出现crash的情况。

而是否高质量的实现并发编程的关键并不是是否应用了某种同步策略，而是看代码中是否保护了共享资源：

- 局部变量之外的内存访问都有并发风险（比如访问对象的属性，访问静态变量等）
- 访问共享资源也会有并发风险（比如缓存、数据库等）。
- 被调用方如果不是声明为线程安全的，那么很有可能存在并发问题（比如java的hashmap）。
- 所有依赖时序的操作，即使每一步操作都是线程安全的，还是存在并发问题（比如先删除一条记录，然后把记录数减一）。

前三种情况能够比较简单的通过代码本身分辨出来，只要简单培养一下自己对于共享资源调用的敏感度就可以了。

但是对于最后一种情况，往往很难简单的通过看代码的方式看出来，甚至出现并发问题的两处调用并不是在同一个程序里（比如两个系统同时读写一个数据库，或者并发的调用了一个程序的不同模块等）。但是，只要是代码里出现了不加锁的，访问共享资源的“先做A，再做B”之类的逻辑，可能就需要提高警惕了。

2.3.3. 优化性能

性能是评价程序员能力的一个重要指标，很多程序员也对程序的性能津津乐道。但程序的性能很难直接通过代码看出来，往往要借助于一些性能测试工具，或者在实际环境中执行才能有结果。

如果仅从代码的角度考虑，有两个评价执行效率的办法：

- 算法的时间复杂度，时间复杂度高的程序运行效率必然会低。
- 单步操作耗时，单步耗时高的操作尽量少做，比如访问数据库，访问IO等。

而实际工作中，也会见到一些程序员过于热衷优化效率，相对的会带来程序易读性的降低、复杂度提高、或者增加工期等等。对于这类情况，简单的办法是让作者说出这段程序的瓶颈在哪里，为什么会有这个瓶颈，以及优化带来的收益。

当然，无论是优化不足还是优化过度，判断性能指标最好的办法是用数据说话，而不是单纯看代码，性能测试这部分内容有些超出这篇文章的范围，就不详细展开了。

2.3.4. 日志

日志代表了程序在出现问题时排查的难易程度，经(jing)验(chang)丰(cai)富(keng)的程序员大概都会遇到过这个场景：排查问题时就少一句日志，查不到某个变量的值不知道是什么，导致死活分析不出来问题到底出在哪。

对于日志的评价标准有三个：

- 日志是否足够，所有异常、外部调用都需要有日志，而一条调用链路上的入口、出口和路径关键点上也需要有日志。
- 日志的表达是否清晰，包括是否能读懂，风格是否统一等。这个的评价标准跟代码的可读性一样，不重复了。
- 日志是否包含了足够的信息，这里包括了调用的上下文、外部的返回值，用于查询的关键字等，便于分析信息。

对于线上系统来说，一般可以通过调整日志级别来控制日志的数量，所以打印日志的代码只要不对阅读造成障碍，基本上都是可以接受的。

2.3.5. 扩展阅读

- 《Release It!: Design and Deploy Production-Ready Software》（不要看中文版，翻译的实在是太烂了）
- [Numbers Everyone Should Know](#)

2.4. 可维护的代码

相对于前两类代码来说，可维护的代码评价标准更模糊一些，因为它要对应的是未来的情况，一般新人很难想象现在的一些做法会对未来造成什么影响。不过根据我的经验，一般来说，只要反复的提问两个问题就可以了：

- 他离职了怎么办？
- 他没这么做怎么办？

2.4.1. 避免重复

几乎所有程序员都知道要避免拷代码，但是拷代码这个现象还是不可避免的成为了程序可维护性的杀手。

代码重复分为两种：模块内重复和模块间重复。无论何种重复，都在一定程度上说明了程序员的水平有问题，模块内重复的问题更大一些，如果在同一个文件里都能出现大片重复的代码，那表示他什么不可思议的代码都有可能写出来。

对于重复的判断并不需要反复阅读代码，一般来说现代的IDE都提供了检查重复代码的工具，只需点几下鼠标就可以了。

除了代码重复之外，很多热衷于维护代码质量的程序员新人很容易出现另一类重复：信息重复。

我见过一些新人喜欢在每行代码前面写一句注释，比如：

```
// 成员列表的长度>0并且<200
if(memberList.size() > 0 && memberList.size() < 200) {
    // 返回当前成员列表
    return memberList;
}
```

看起来似乎很好懂，但是几年之后，这段代码就变成了：

```
// 成员列表的长度>0并且<200
if(memberList.size() > 0 && memberList.size() < 200 || (tmp.isOpen() && flag)) {
    // 返回当前成员列表
    return memberList;
}
```

再之后可能会改成这样：

```
// edit by axb 2015.07.30
// 成员列表的长度>0并且<200
//if(memberList.size() > 0 && memberList.size() < 200 || (tmp.isOpen() && flag)) {
//    返回当前成员列表
//    return memberList;
//}
if(tmp.isOpen() && flag) {
    return memberList;
}
```

随着项目的演进，无用的信息会越积越多，最终甚至让人无法分辨哪些信息是有效的，哪些是无效的。

如果在项目中发现好几个东西都在做同一件事情，比如通过注释描述代码在做什么，或者依靠注释替代版本管理的功能，那么这些代码也不能称为好代码。

2.4.2. 模块划分

模块内高内聚与模块间低耦合是大部分设计遵循的标准，通过合理的模块划分能够把复杂的功能拆分为更易于维护的更小的功能点。

一般来说可以从代码长度上初步评价一个模块划分的是否合理，一个类的长度大于2000行，或者一个函数的长度大于两屏幕都是比较危险的信号。

另一个能够体现模块划分水平的地方是依赖。如果一个模块依赖特别多，甚至出现了循环依赖，那么也可以反映出作者对模块的规划比较差，今后在维护这个工程的时候很有可能出现牵一发而动全身的情况。

一般来说有不少工具能提供依赖分析，比如IDEA中提供的[Dependencies Analysis](#)功能，学会这些工具的使用对于评价代码质量会有很大的帮助。

值得一提的是，绝大部分情况下，不恰当的模块划分也会伴随着极低的单元测试覆盖率：复杂模块的单元测试非常难写的，甚至是不可能完成的任务。所以直接查看单元测试覆盖率也是一个比较靠谱的评价方式。

2.4.3. 简洁与抽象

只要提到代码质量，必然会提到简洁、优雅之类的形容词。简洁这个词实际涵盖了很多东西，代码避免重复是简洁、设计足够抽象是简洁，一切对于提高可维护性的尝试实际都是在试图做减法。

编程经验不足的程序员往往不能意识到简洁的重要性，乐于捣鼓一些复杂的玩意并乐此不疲。但复杂是代码可维护性的天敌，也是程序员能力的一道门槛。

跨过门槛的程序员应该有能力控制逐渐增长的复杂度，总结和抽象出事物的本质，并体现到自己设计和编码中。一个程序的生命周期也是在由简入繁到化繁为简中不断迭代的过程。

对于这部分我难以总结出简单易行的评价标准，它更像是一种思维方式，除了要理解，还需要练习。多看、多想、多交流，很多的时候可以简化的东西会大大超出原先的预计。

2.2.4. 推荐阅读

- 《重构-改善既有代码的设计》
- 《设计模式-可复用面向对象软件的基础》

- 《Software Architecture Patterns-Understanding Common Architecture Patterns and When to Use Them》

3. 结语

这篇文章主要介绍了一些评价代码质量优劣的手段，这些手段中，有些比较客观，有些主观性更强。之前也说过，对代码质量的评价是一件主观的事情，这篇文章里虽然列举了很多评价手段。但是实际上，很多我认为没有问题的代码也会被其他人吐槽，所以这篇文章只能算是初稿，更多内容还需要今后继续补充和完善。

虽然每个人对于代码质量评价的倾向都不一样，但是总体来说评价代码质量的能力可以被比作程序员的“品味”，评价的准确度会随着自身经验的增加而增长。在这个过程中，需要随时保持思考、学习和批判的精神。

下篇文章里，会谈一谈具体如何提高自己的代码质量。

关于烂代码的那些事（下）

转载 isunlight001 发布于2017-08-07 18:00:18

[关于烂代码的那些事（上）](#)

[关于烂代码的那些事（中）](#)

[关于烂代码的那些事（下）](#)

假设你已经读过烂代码系列的前两篇：了解了什么是烂代码，什么是好代码，但是还是不可避免的接触到了烂代码（就像之前说的，几乎没有程序员可以完全避免写出烂代码！）接下来的问题便是：如何应对这些身边的烂代码。

1.改善可维护性

改善代码质量是项大工程，要开始这项工程，从可维护性入手往往是一个好的开始，但也仅仅是开始而已。

1.1.重构的悖论

很多人把重构当做一种一次性运动，代码实在是烂的没法改了，或者没什么新的需求了，就召集一帮人专门拿出来一段时间做重构。这在传统企业开发中多少能生效，但是对于互联网开发来说却很难适应，原因有两个：

1. 互联网开发讲究快速迭代，如果要做大型重构，往往需要暂停需求开发，这个基本上很难实现。
2. 对于没有什么新项目的项目，往往意味着项目本身已经过了发展期，即使做了重构也带来不了什么收益。

这就形成了一个悖论：一方面那些变更频繁的系统更需要重构；另一方面重构又会耽误开发进度，影响变更效率。

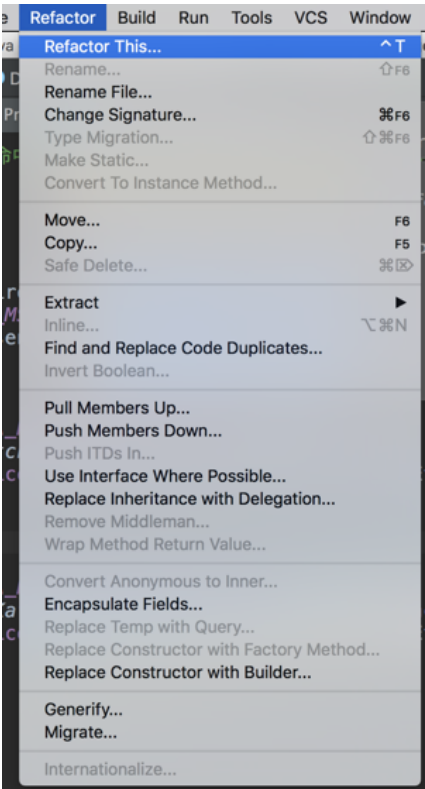
面对这种矛盾，一种方式是放弃重构，让代码质量自然下降，直到工程的生命周期结束，选择放弃或者重来。在某些场景下这种方式确实是有效的，但是我不喜欢：比起让工程师不得不把每天的精力都浪费在毫无意义的事情上，为什么不做些更有意义的事呢？

1.2.重构step by step

1.2.1.开始之前

开始改善代码的第一步是把IDE的重构快捷键设到一个顺手的键位上，这一步非常重要：决定重构成败的往往不是你的新设计有多么牛逼，而是重构本身会占用多少时间。

比如对于IDEA来说，我会把重构菜单设为快捷键：



这样在我想去重构的时候就可以随手打开菜单，而不是用鼠标慢慢去点，快捷键每次只能为重构节省几秒钟时间，但是却能明显减少工程师重构时的心理负担，后面会提到，小规模的重构应该跟敲代码一样属于日常开发的一部分。

我把重构分为三类：模块内部的重构、模块级别的重构、工程级别的重构。分为这三类并不是因为我是什么分类强迫症，后面会看到对重构的分类对于重构的意义。

1.2.2.随时进行模块内部的重构

模块内部重构的目的是把模块内部的逻辑梳理清楚，并且把一个巨大无比的函数拆分成可维护的小块代码。大部分IDE都提供了对这类重构的支持，类似于：

- 重命名变量
- 重命名函数
- 提取内部函数
- 提取内部常量
- 提取变量

这类重构的特点是修改基本集中在一个地方，对代码逻辑的修改很少并且基本可控，IDE的重构工具比较健壮，因而基本没有什么风险。

以下例子演示了如何通过IDE把一个冗长的函数做重构：

```
public class Demo {  
  
    private UserDao userDao;  
  
    void update(User u, String s) {  
        if (u.getType() == 1) {  
            if (s.length() > 0 && s.length() < 10) {  
                List<Group> list = u.getGroups();  
                for (Group g : list) {  
                    if ((g.getStatus() == 5 || g.getStatus() == 7)) {  
                        u.setName(s);  
                        userDao.save(u);  
                        break;  
                    }  
                }  
            }  
        } else {  
            if (s.length() > 0 && s.length() < 10) {  
                String url = "http://xxxxx/check.json?uid=" + u.getId();  
                HttpClient client = new HttpClient();  
                GetMethod method = new GetMethod(url);  
                try {  
                    client.executeMethod(method);  
                    String s1 = method.getResponseAsString();  
                    if ("1".equals(s1)) {  
                        u.setName(s);  
                        userDao.save(u);  
                    }  
                } catch (IOException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
}
```

上图的例子中，我们基本依靠IDE就把一个冗长的函数分成了两个子函数，接下来就可以针对子函数中的一些烂代码做进一步的小规模重构，而两个函数内部的重构也可以用同样的方法。每一次小规模重构的时间都不应该超过60s，否则将会严重影响开发的效率，进而导致重构被无尽的开发需求淹没。

在这个阶段需要对现有的模块补充一些单元测试，以保证重构的正确。不过以我的经验来看，一些简单的重构，例如修改局部变量名称，或者提取变量之类的重构，即使没有测试也是基本可靠的，如果要在快速完成模块内部重构和100%的单元测试覆盖率中选一个，我可能会选择快速完成重构。

而这类重构的收益主要是提高函数级别的可读性，以及消除超大函数，为未来进一步做模块级别的拆分打好基础。

1.2.3.一次只做一个较模块级别的的重构

之后的重构开始牵扯到多个模块，例如：

- 删除无用代码
- 移动函数到其它类
- 提取函数到新类

- 修改函数逻辑

IDE往往对这类重构的支持有限，并且偶尔会出一些莫名其妙的问题，（例如修改类名时一不小心把配置文件里的常量字符串也给修改了）。

这类重构主要在于优化代码的设计，剥离不相关的耦合代码，在这类重构期间你需要创建大量新的类和新的单元测试，而此时的单元测试则是**必须**的了。

为什么要创建单元测试？

- 一方面，这类重构因为涉及到具体代码逻辑的修改，靠集成测试很难覆盖所有情况，而单元测试可以验证修改的正确性。
- 更重要的意义在于，写不出单元测试的代码往往意味着糟糕的设计：模块依赖太多或者一个函数的职责太重，想象一下，想要执行一个函数却要模拟十几个输入对象，每个对象还要模拟自己依赖的对象.....如果一个模块无法被**单独**测试，那么从设计的角度来考虑，无疑是不合格的。

还需要啰嗦一下，这里说的单元测试只对一个模块进行测试，依赖多个模块共同完成的测试并不包含在内-例如在内存里模拟了一个数据库，并在上层代码中测试业务逻辑-这类测试并不能改善你的设计。

在这个期间还会写一些过渡用的临时逻辑，比如各种adapter、proxy或者wrapper，这些临时逻辑的生存期可能会有几个月到几年，这些看起来没什么必要的工作是为了控制重构范围，例如：

```
1      class Foo {
2          String foo() { ... }
3
4
5
6
```

如果要把函数声明改成

```
1      class Foo {
2          boolean foo() { ... }
3
4
5
6
```

那么最好通过加一个过渡模块来实现：

```
//
1      class FooAdaptor {
2          private Foo foo;
3          boolean foo() {
4              return foo.foo().isEmpty();
5          }
6      }
7
```

这样做的好处是修改函数时不需要改动所有调用方，烂代码的特征之一就是模块间的耦合比较高，往往一个函数有几十处调用，牵一发而动全身。而一旦开始全面改造，往往就会把一次看起来很简单重构演变成几周的大工程，这种大规模重构往往是不可靠的。

每次模块级别的重构都需要精心设计，提前划分好哪些是需要修改的，哪些是需要用兼容逻辑做过渡的。但实际动手修改的时间都不应该超过一天，如果超过一天就意味着这次重构改动太多，需要控制一下修改节奏了。

1.2.4.工程级别的重构不能和任何其他任务并行

不安全重构相对而言影响范围比较大，比如：

- 修改工程结构
- 修改多个模块

我更建议这类操作不要用IDE，如果使用IDE，也只使用最简单的“移动”操作。这类重构单元测试已经完全没有作用，需要集成测试的覆盖。不过也不必紧张，如果只做“移动”的话，大部分情况下基本的冒烟测试就可以保证重构的正确性。

这类重构的目的是根据代码的层次或者类型进行拆分，切断循环依赖和结构上不合理的地方。如果不知道如何拆分，可以依照如下思路：

1. 优先按部署场景进行拆分，比如一部分代码是公用的，一部分代码是自己用的，可以考虑拆成两个部分。换句话说，A服务的修改能不能影响B服务。
2. 其次按照业务类型拆分，两个无关的功能可以拆分成两个部分。换句话说，A功能的修改能不能影响B功能。
3. 除此之外，尽量控制自己的代码洁癖，不要把代码切成一大堆豆腐块，会给日后的维护工作带来很多不必要的成本。
4. 案可以提前review几次，多参考一线工程师的意见，避免实际动手时才冒出新的问题。

而这类重构绝对不能跟正常的需求开发并行执行：代码冲突几乎无法避免，并且会让所有人崩溃。我的做法一般是在这类重构前先演练一次：把模块按大致的想法拖来拖去，通过编译器找到依赖问题，在日常上线中把容易处理的依赖问题解决掉；然后集中团队里的精英，通知所有人暂停开发，花最多2、3天时间把所有问题集中突击掉，新的需求都在新代码的基础上进行开发。

如果历史包袱实在太重，可以把这类重构也拆成几次做：先大体拆分成几块，再分别拆分。无论如何，这类重构务必控制好变更范围，一次严重的合并冲突有可能让团队中的所有几个周缓不过劲来。

1.3.重构的周期

典型的重构周期类似下面的过程：

1. 在正常需求开发的同时进行模块内部的重构，同时理解工程原有代码。
2. 在需求间隙进行模块级别的重构，把大模块拆分为多个小模块，增加脚手架类，补充单元测试，等等。
3. （如果有必要，比如工程过于巨大导致经常出现相互影响问题）进行一次工程级别的拆分，期间需要暂停所有开发工作，并且这次重构除了移动模块和移动模块带来的修改之外不做任何其他变更。
4. 重复1、2步骤

1.3.1.一些重构的tips

1. 只重构经常修改的部分，如果代码一两年都没有修改过，那么说明改动的收益很小，重构能改善的只是可维护性，重构不维护的代码不会带来收益。
2. 抑制住自己想要多改一点的冲动，一次失败的重构对代码质量改进的影响可能是毁灭性的。
3. 重构需要不断的练习，相比于写代码来说，重构或许更难一些。
4. 重构可能需要很长时间，有可能甚至达到几年的程度（我之前用断断续续两年多的时间重构了一个项目），主要取决于团队对于风险的容忍程度。
5. 删除无用代码是提高代码可维护性最有效的方式，切记，切记。
6. 单元测试是重构的基础，如果对单元测试的概念还不是很清晰，可以参考使用Spock框架进行单元测试。

2.改善性能与健壮性

2.1.改善性能的80%

性能这个话题越来越多的被人提起，随便收到一份简历不写上点什么熟悉高并发、做过性能优化之类的似乎都不好意思跟人打招呼。

说个真事，几年前在我做某公司的ERP项目，里面有个功能是生成一个报表。而使用我们系统的公司里有一个人，他每天要在下班前点一下报表，导出到excel，再发一封邮件出去。

问题是，那个报表每次都要2，3分钟才能生成。

我当时正年轻气盛，看到有个两分钟才能生成的报表一下就来兴趣，翻出了那段不知道谁写的代码，发现里面用了3层循环，每次都会去数据库查一次数据，再把一堆数据拼起来，一股脑塞进一个tableview里。

面对这种代码，我还能做什么呢？

- 我立刻把那个三层循环干掉了，通过一个存储过程直接输出数据。
- sql数据计算的逻辑也被我精简了，一些没必要做的外联操作被我干掉了。
- 我还发现很多ctrl+v生成的无用的控件（那时还是用的delphi），那些控件密密麻麻的贴在显示界面上，只是被前面的大table挡住了，我当然也把这些玩意都删掉了；

- 打开界面的时候还做了一些杂七杂八的工作（比如去数据库里更新点击数之类的），我把这些放到了异步任务里。
- 后面我又觉得没必要每次打开界面都要加载所有数据（那个tableview有几千行，几百列！），于是我hack了默认的tableview，每次打开的时候先计算当前实际显示了多少内容，把参数发给存储过程，初始化只加载这些数据，剩下的再通过线程异步加载。

做了这些之后，界面只需要不到1s就能展示出来了，不过我要说的不是这个。

后来我去客户公司给那个操作员演示新的模块的时候，点一下，刷，数据出来了。那个人很惊恐的看着我，然后问我，是不是数据不准了。

再后来，我又加了一个功能，那个模块每次打开之后都会显示一个进度条，上面的标题是“正在校验数据……”，进度条走完大概要1分钟左右，我跟那人说校验数据计算量很大，会比较慢。当然，实际上那60秒里程序毛事都没做，只是在一点点的更新那个进度条（我还做了个彩蛋，在读进度的时候按上上下下左右左右BABA的话就可以加速10倍读条...）。客户很开心，说感觉数据准确多了，当然，他没发现彩蛋。

我写了这么多，是想让你明白一个事实：大部分程序对性能并不敏感。而少数对性能敏感的程序里，一大半可以靠调节参数解决性能问题；最后那一小撮需要修改代码优化性能的程序里，性价比高的工作又是少数。

什么是性价比？回到刚才的例子里，我做了那么多事，每件事的收益是多少？

- 把三层循环sql改成了存储过程，大概让我花了一天时间，让加载时间从3分钟变成了2秒，模块加载变成了“唰”的一下。
- 后面的一坨事情大概花了我一周多时间，尤其是hack那个tableview，让我连周末都搭进去了。而所有的优化加起来，大概优化了1秒左右，这个数据是通过日志查到的：即使是我自己，打开模块也没感觉出有什么明显区别。

我现在遇到的很多面试者说程序优化时总是喜欢说一些玄乎的东西：调用栈、尾递归、内联函数、GC调优……但是当我问他们：把一个普通函数改成内联函数是把原来运行速度是多少的程序优化成多少了，却很少有人答出来；或者是扭扭捏捏的说，应该很多，因为这个函数会被调用很多遍。我再问会被调用多少遍，每遍是多长时间，就答不上来了。

所以关于性能优化，我有两个观点：

1. 优化主要部分，把一次网络IO改为内存计算带来的收益远大于折腾编译器优化之类的东西。这部分内容可以参考Numbers you should know；或者自己写一个for循环，做一个无限i++的程序，看看一秒钟i能累加多少次，感受一下cpu和内存的性能。
2. 性能优化之后要有量化数据，明确的说出优化后哪个指标提升了多少。如果有人因为“提升性能”之类的理由写了一堆让人无法理解的代码，请务必让他给出性能数据：这很有可能是一坨没有什么收益的烂代码。

至于具体的优化措施，无外乎几类：

1. 让计算靠近存储
2. 优化算法的时间复杂度
3. 减少无用的操作
4. 并行计算

关于性能优化的话题还可以讲很多内容，不过对于这篇文章来说有点跑题，这里就不再详细展开了。

2.2.决定健壮性的20%

前一阵听一个技术分享，说是他们在编程的时候要考虑太阳黑子对cpu计算的影响，或者是农民伯伯的猪把基站拱塌了之类的特殊场景。如果要优化程序的健壮性，那么有时候就不得不去考虑这些极端情况对程序的影响。

大部分的人应该不用考虑太阳黑子之类的高深的问题，但是我们需要考虑一些常见的特殊场景，大部分程序员的代码对于一些特殊场景都会有或多或少考虑不周全的地方，例如：

- 用户输入
- 并发
- 网络IO

常规的方法确实能够发现代码中的一些bug，但是到了复杂的生产环境中时，总会出现一些完全没有想到的问题。虽然我也想了很久，遗憾的是，对于健壮性来说，我并没有找到什么立竿见影的解决方案，因此，我只能谨慎的提出一点点建议：

- 更多的测试测试的目的是保证代码质量，但测试并不等于质量，你做覆盖80%场景的测试，在20%测试不到的地方还是有可能出问题。关于测试又是一个巨大的话题，这里就先不展开了。
- 谨慎发明轮子。例如UI库、并发库、IO client等等，在能满足要求的情况下尽量采用成熟的解决方案，所谓的“成熟”也就意味着经历了更多实际使用环境下的测试，大部分情况下这种测试的效果是更好的。

3.改善生存环境

看了上面的那么多东西之后，你可以想一下这么个场景：

在你做了很多事情之后，代码质量似乎有了质的飞跃。正当你以为终于可以摆脱天天踩屎的日子了的时候，某次不小心瞥见某个类又长到几千行了。

你愤怒的翻看提交日志，想找出罪魁祸首是谁，结果却发现每天都会有人往文件里提交那么十几二十行代码，每次的改动看起来都没什么问题，但是日积月累，一年年过去，当初花了九牛二虎之力重构的工程又成了一坨烂代码.....

任何一个对代码有追求的程序员都有可能遇到这种问题，技术在更新，需求在变化，公司人员会流动，而代码质量总会在不经意间偷偷的变差.....

想要改善代码质量，最后往往就会变成改善生存环境。

3.1.1.统一环境

团队需要一套统一的编码规范、统一的语言版本、统一的编辑器配置、统一的文件编码，如果有条件最好能使用统一的操作系统，这能避免很多无意义的工作。

就好像最近渣浪给开发全部换成了统一的macbook，一夜之间以前的很多问题都变得不是问题了：字符集、换行符、IDE之类的问题只要一个配置文件就解决了，不再有各种稀奇古怪的代码冲突或者不兼容的问题，也不会有人突然提交上来一些编码格式稀奇古怪的文件了。

3.1.2.代码仓库

代码仓库基本上已经是每个公司的标配，而现在的代码仓库除了储存代码，还可以承担一些团队沟通、代码review甚至工作流程方面的任务，如今这类开源的系统很多，像gitlab(github)、Phabricator这类优秀的工具都能让代码管理变得简单很多。我这里无意讨论svn、git、hg还是什么其它的代码管理工具更好，就算最近火热的git在复杂性和集中化管理上也有一些问题，其实我是比较期待能有替代git的工具产生的，扯远了。

代码仓库的意义在于让更多的人能够获得和修改代码，从而提高代码的生命周期，而代码本身的生命周期足够持久，对代码质量做的优化才有意义。

3.1.3.持续反馈

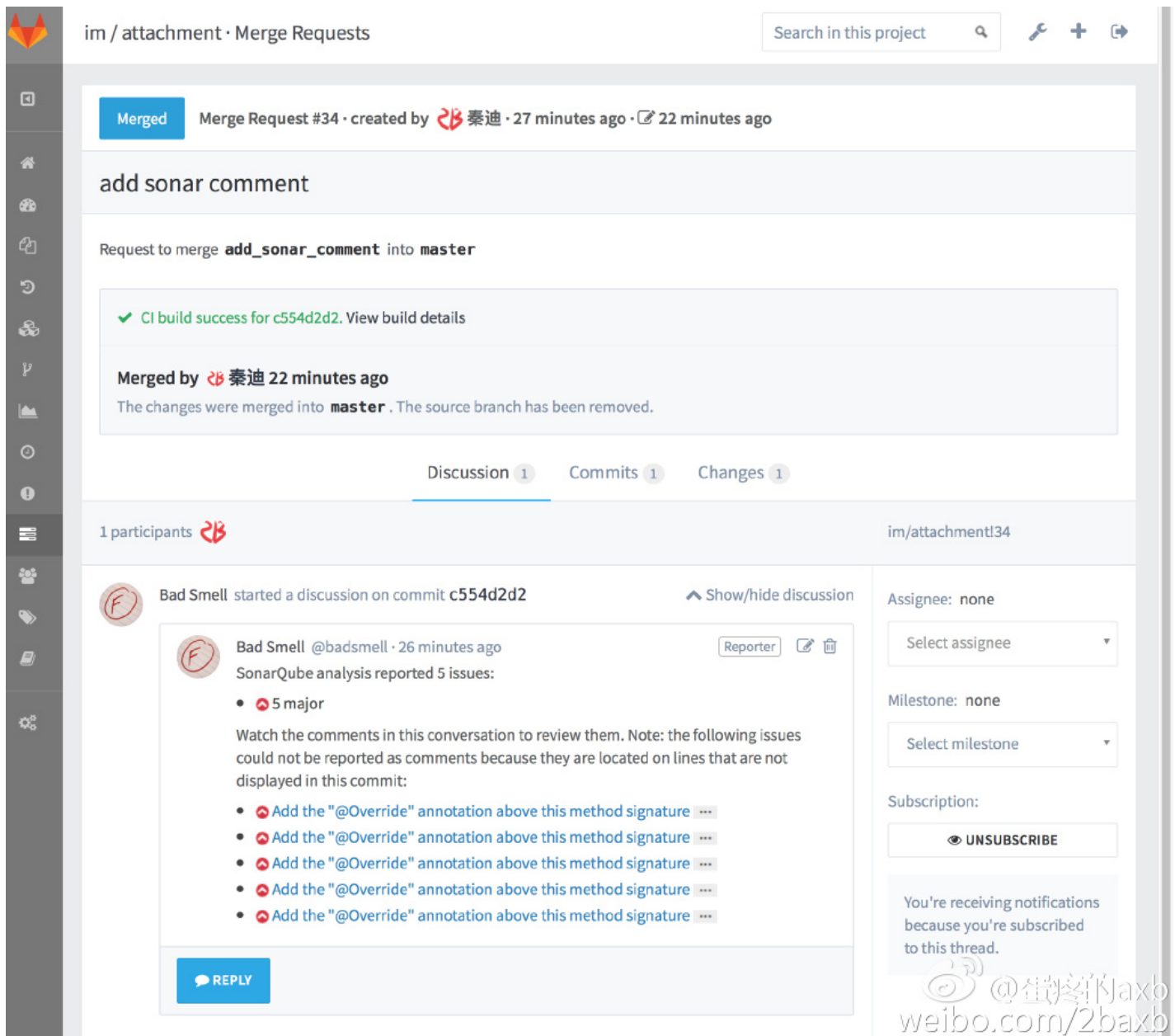
大多数烂代码就像癌症一样，当烂代码已经产生了可以感觉到的影响时，基本已经是晚期，很难治好了。

因此提前发现代码变烂的趋势很重要，这类工作可以依赖类似于checkstyle，findbug之类的静态检查工具，及时发现代码质量下滑的趋势，例如：

1. 每天都在产生大量的新代码
2. 测试覆盖率下降
3. 静态检查的问题增多

有了代码仓库之后，就可以把这种工具与仓库的触发机制结合起来，每次提交的时候做覆盖率、静态代码检查等工作，jenkins+sonarqube或者类似的工具就可以完成基本的流程：伴随着代码提交进行各种静态检查、运行各种测试、生成报告并供人参考。

在实践中会发现，关于持续反馈的五花八门的工具很多，但是真正有用的往往只有那么一两个，大部分人并不会去在每次提交代码之后再打开一个网页点击“生成报告”，或者去登陆什么系统看一下测试的覆盖率是不是变低了，因此一个一站式的系统大多数情况下会表现的更好。与其追求更多的功能，不如把有限的几个功能整合起来，例如我们把代码管理、回归测试、代码检查、和code review集成起来，就是这个样子：



当然，关于持续集成还可以做的更多，篇幅所限，就不多说了。

3.1.4. 质量文化

不同的团队文化会对技术产生微妙的影响，关于代码质量没有什么共同的文化，每个公司都有自己的一套观点，并且似乎都能说得通。

对于我自己来说，关于代码质量是这样的观点：

1. 烂代码无法避免
2. 烂代码无法接受
3. 烂代码可以改进
4. 好的代码能让工作更开心一些

如何让大多数人认同关于代码质量的观点实际上是有一些难度的，大部分技术人员对代码质量的观点是既不赞成、也不反对的中立态度，而代码质量就像是熵值一样，放着不管总是会像更加混乱的方向演进，并且写烂代码的成本实在是太低了，以至于一个实习生花上一个礼拜就可以毁了你花了半年精心设计的工程。

所以在提高代码质量时，务必想办法拉上团队里的其他人一起。虽然“引导团队提高代码质量”这件事情一开始会很辛苦，但是一旦有了一些支持者，并且有了可以参考的模板之后，剩下的工作就简单多了。

这里推荐《布道之道：引领团队拥抱技术创新》这本书，里面大部分的观点对于代码质量也是可以借鉴的。仅靠喊口号很难让其他人写出高质量的代码，让团队中的其他人体到高质量代码的收益，比喊口号更有说服力。

4. 最后再说两句

优化代码质量是一件很有意思，也很有挑战性的事情，而挑战不光来自于代码原本有多烂，要改进的也并不只是代码本身，还有工具、习惯、练习、开发流程、甚至团队文化这些方方面面的事情。

写这一系列文章前前后后花了半年多时间，一直处在写一点删一点的状态：我自身关于代码质量的想法和实践也在经历着不断变化。我更希望能写出一些能够实践落地的东西，而不是喊喊口号，忽悠悠“敏捷开发”、“测试驱动”之类的几个名词就结束了。

但是在写文章的过程中就会慢慢发现，很多问题的改进方法确实不是一两篇文章可以说明白的，问题之间往往又相互关联，全都展开说甚至超出了一本书的信息量，所以这篇文章也只能删去了很多内容。

我参与过很多代码质量很好的项目，也参与过一些质量很烂的项目，改进了很多项目，也放弃了一些项目，从最初的单打独斗自己改代码，到后来带领团队优化工作流程，经历了很多。无论如何，关于烂代码，我决定引用一下《布道之道》这本书里的一句话：

“‘更好’，其实不是一个目的地，而是一个方向...在当前的位置和将来的目标之间，可能有很多相当不错的地方。你只需关注离开现在的位置，而不要关心去向何方。”