如何写可维护代码

细节之中自有天地，整洁成就卓越代码

1.可维护代码的重要性

- 个人-个人被糟糕的代码困扰
- 团队-团队扩大，业务线扩大，功能增加，会导致系统无法继续扩展。（架构师需要考虑这些问题）
- 公司-公司会被糟糕的代码毁掉。（大版本迭代）

原因：

- 写糟糕代码的原因：或许时间不够，或许不耐烦等等，而且也总是想稍后清理一下。
- 勒布朗法则：稍后等于永不 （Later equals never）
- 写整洁代码就如同画画，好的坏的回看不一定会画，代码也一样，写整洁代码是一种艺术的提升。（一定要有一定的强迫症，才能写出艺术级别的代码）
- 为了编写显得高深莫测的代码。

2.怎么写出可维护性代码

- 1.命名
- 2.函数
- 3.注释

2.1 有意义的命名

- 1.名副其实
- 2.避免误导
- 3.做有意义的分区
- 4.使用读得出来的名称
- 5.使用可搜索的名称
- 6.避免使用编码
- 7.避免思维映射
- 8.类名与方法名
- 9.其他

2.1.1 名副其实

- 名副其实说起来很简单，但是这个事很严肃，选个好名字要花时间，但省下来的时间比花掉的多。注意命名，而且一旦发现好的名称，就替换掉旧的。这么做读你代码的人也会很开心。
- 变量、函数或者类的名称应该答复了所有的大问题。它该告诉你，它为什么会存在，它做什么事情，应该怎么用。如果名称需要注释来补充，那就不算是名副其实了。（eg）

- 名称命名网站：http://unbug.github.io/codelf/

2.2.2 避免误导

- 程序员必须避免留下掩藏代码本意的错误线索。
- 应当避免使用与本意相悖的词。
- 提防使用不同之处较小的名称。（eg）

2.2.3做有意义的分区

- 如果程序员只是以满足编译器或者解释器的需要而写代码，就会造成麻烦。

（例如：因为同一作用域内两样不同的东西不能重名，你可能会随手改掉其中一个的名称，有时候干脆以错误的拼写充数，结果就是出现在更正拼写错误后会导致编译器出错的情况）(eg)

2.2.4 使用读的出来的名称

- 开发工作跟社区一样，经常会大家一起讨论，所以命名时一定要用读的出来的名称，否则就会时长听见一边讨论问题一边读拼音的感觉。(eg)

2.2.5 使用可搜索的名称

- 当我们从事的开发项目规模逐渐增加，代码量以及文件量逐渐增加的情况下，对于搜索的依赖性会越来越强，如果采用的命名不能搜索，会增加很大的工作量，而且容易导致代码重复。(eg)

2.2.6 避免使用编码

- 避免把类型或者作用域编进名称里
- 避免使用前缀来表明成员变量。应当把类做的足够小，消除对成员前缀的需求。

（人们很快学会无视前缀或者后缀，只看到名称中有意义的部分，代码读的越多，眼中就越没有前缀。最终,前缀变作了不入法眼的废料，变作了旧代码的标志物）(eg-)

2.2.7 避免思维映射

- 单字母变量首先就是个问题，在作用域比较小的时候，也没有命名冲突时，循环计数器自然有可能被命名为i,j,k（但千万别用l），然后在很多情况下单字母不是一个好选择，读者看到变量时会在脑海中将它映射成真实概念。仅仅因为看到了a和b，就要取c，实在不是像样的理由。
- 聪明程序员和专业程序员之间区别在于：专业程序员了解，明确是王道，专业程序要能编写出其他人能理解的代码。

2.2.8 类名与方法名

- 类名和对象名应该是名词或者名词短语，不应该是动词。(eg+)

2.2.9 其他

- 别扮可爱：名称别搞得太花哨而不实用
- 每个概念对应一个词：给每个抽象的概念选一个词，并且一直如此。
- 别用双关语。如插入数组，用insert比用add好。
- 使用解决方案领域名称.（记住只有程序员才会读你的代码，所以尽管用那些计算机术语）
- 使用源自所涉及问题领域的名称。（业务层术语）
- 添加有意义的语境。如state，单独出现不明白意思，如果在有关地址的函数作用域中，就会很清楚意思了
- 不要添加没用的语境

2.2.10

- 总结：取好的名字最难的地方在于需要良好的描述技巧和共有文化背景。我们应该致力于把代码写的就像词句篇章，像一种艺术品，让自己看着舒服，让别人看着也舒服。大家不妨试试上面的一些规则，看你的代码可读性是否有所提升。
- 当你开发代码的时候会注意以上命名的问题时，维护你代码的同事就会很容易的看懂你的代码。

3.1 函数

- 在编程的早年岁月，系统由程序和子程序组成。后来，在Fortran和PL/1的年代，系统由程序、子程序和函数组成。如今，只有函数存活了下来。函数是所有程序中的第一组代码，所以写好函数是很重要的一件事。主要分为以下几个部门来阐述如何能把函数写的可维护：
- 1. 短小。
- 2. 只做一件事。
- 3. 每个函数一个抽象层级
- 4.使用描述性名称
- 5.函数参数
- 6.无副作用
- 7.别重复自己
- 8.结构化编程

3.2.1 短小

- 写函数的第一准则就是要短小，第二条规则是还要更短小。
- 1.当一个函数过于庞大时，开发他的人最后会很乏力。
- 2.对于后续的维护人员来说，庞大的函数是很让人发怵的。

3.2.2 只做一件事

- 过去30年以来，以下建议以不同的形式一再出现：函数应该做一件事。做好这件事。只做一件事。
- 随着大家的项目代码规模越来越大，代码的冗余也会越来越多，面临的情况也会越来越多，当多个功能混杂在一个函数中时，最终会造成拆东墙补西墙的情况频繁出现。
- 业务增加，需求改变时。

3.2.3 每个函数一个抽象层级

- 函数中混杂不同抽象层级，让人迷惑。
- 编写函数尽量遵循向下规则。(eg)

3.2.4 使用描述性名称

- 沃德原则：如果每个例程都让你感到深合己意，那就是整洁代码。
- 大半工作都在于为一件小事的小函数取个好名字，函数越短小、功能越集中。就越便于取个好名字。
- 别害怕名字长，长而具有描述性的名称，要比短而令人费解的名称好。
- 选择描述性的名称能帮你理清关于模块的设计思路，并帮你改进，追求好名字。
- 命名方式要保持一致。

3.2.5 函数参数

- 最理想的参数数量是零，其实是一，再次是二，应尽量避免三。
- 一元函数，如果函数要对入参进行转换，应当在返回值中体现出转换结果。
- 标识参数，不要向函数中传如布尔值。这样就是大声宣布了函数不是做一件事。
- 二元函数，尽量将二元函数转换成一元函数。
- 三元函数，如果必须出现三元函数，则最好把参数封装成类，大多数情况下，多个参数都是某个概念的一部分。(eg)
- 给函数取个好名字，能较好的解释函数的意图，以及参数的顺序与意图。对于一元函数和参数应当行程一种非常良好的动词、名词对形式，如write(name)。

3.2.6 无副作用

- 当函数编写时如果内部嵌套或者调用函数时，命名时一定要表示出来，否则就可能会产生副作用。(eg)

3.2.7 别重复自己

- 写代码时，不要写重复的代码。否则会让错误产生的概率上升N倍。

3.2.8 结构化编程

- 有些程序员遵循艾兹格·迪科斯彻（关系数据库之父、结构程序设计之父）的结构化编程规范，他认为每个函数、函数中的每个代码块都应该有一个入口一个出口。这意味着函数中只能有一个return，不能有break，continue。
- 这种结构化的理念其实比较适合较大 的函数。所以我们的函数写的一定要小，函数足够小，偶尔出现一次break，continue语句也没有坏处。

3.2.8 小结

- 在程序开发中，函数写的最多，在开发中一定要切记最重要的两点：函数要小和只做一件事。

4 注释

- 注释并一定是好的，若编程语言有足够的表达力，那么就不那么需要注释了。（中国人语言差异化）
- 注释的恰当使用是为了弥补我们在用代码表达意图的失败。
- 如果你发现自己需要写注释，那么再想想看是不是能用代码来表达，每次用代码表达你都改表扬下自己，每次写注释你都在批评一下自己表达能力的不足。
- 下面从以下几点阐述以下注释。

4.1 注释不能美化的代码

- 写注释的常见原因是糟糕代码的存在，与其花时间编写解释糟糕代码的注释，不如花时间来清洁一下那堆糟糕的代码。

4.2 用代码来阐述

- //判断员工是否有拿全额养老金的资格

  If((employee.flags & HOURLY.FLAG) && (employee.age > 65)){}

用代码来阐述

If(employee.isEligibleForFullBenefits()){}

4.3 好注释（有一些注释是必须的）

- 1.法律信息
- 2.提供信息的注释
- 3.对意图的解释等
- 4.阐释
- 5.警示
- 6.TODO注释
- 7.放大

4.4 坏注释

- 1.喃喃自语
- 2.多余的注释
- 3.误导性注释
- 4.循轨式注释
- 5.日志式注释
- 6.废话注释
- 7.可怕的废话
- 8.能用函数或变量时就别用注释

9.位置标记
10.括号后面的注释
11.归属和署名
12.注释掉的代码
13.HTML注释
14.非本地信息
15.信息过多
16.不明显的联系
17.函数头

4.5 总结

- 对于我们来说，语言差异比较大，有一些注释其实是必须得，但是在有一些注释能用代码来阐述尽量别用注释，除非真的表达不清楚函数的意图。

借用美国童子军的军规

- 让营地比你来时更干净

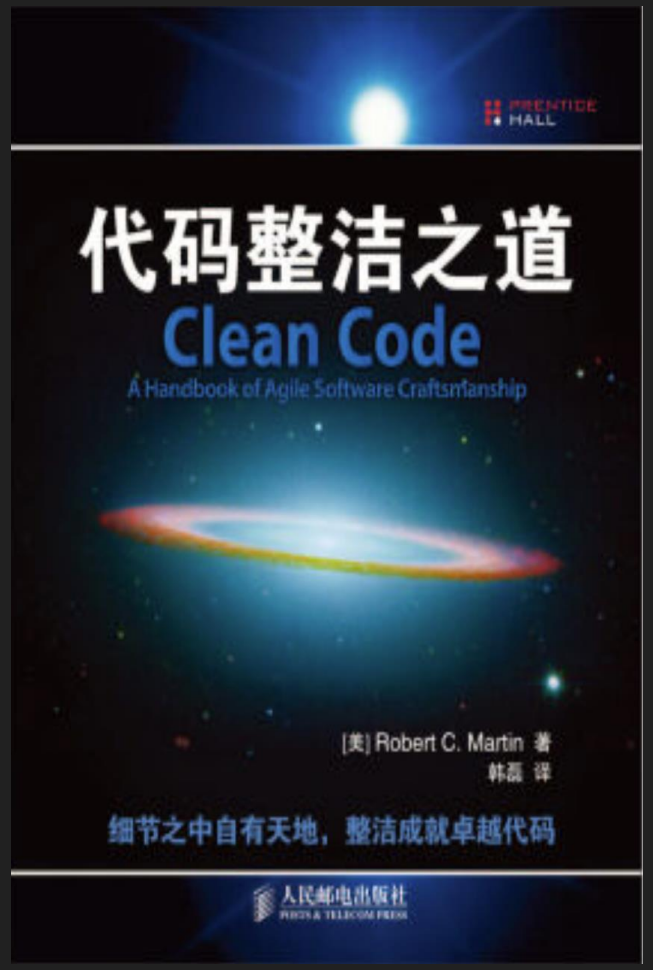
总结

- 以上所涵盖的内容，不一定所有情况下都必须遵循才是好的，一定要按照实际情况来编写，但是在开发的时候，可以去思考以下能否按照以上的一些方式，来让代码清晰的展示自己的意图，让其他人也能很容易的读懂这些内容，能否在团队开发时，交叉开发的过程中让大家都能高效的合作。

- 其实有很多人认为规范这个是每个人都有一套不同的形式。但是我不太认可，如果按照自己规范写出来的代码，大家看起来都很吃力的话，那就不是一个好的规范。真正的高手应该是把复杂的代码写的简单易懂，一个把复杂问题写的让新手都能看懂的高手，我认为才是真正的高手。

总结

- 给大家推荐一本书



京东链接：
https://item.jd.com/10064006.html