**如何编写高质量的代码——来自《代码大全（第2版）》的启示**

如何编写高质量的代码

——来自《代码大全（第2版）》的启示

陈硕 编译

说明：这篇文章大量文字直接取自《代码大全（第2版）》中译本。本文发表于《程序员》杂志2006年第3期（杂志上的文章去掉了参考文献和脚注，格式编排也有些小问题），转载请先获得杂志社的许可。

软件的首要技术使命是管理复杂度，计算先驱Edsger Dijkstra指出，只有在“计算(Computing)”这种职业中，人的思维需要从一个字节大幅跨越到几百兆字节——跨度为109比1，也就是9个数量级**[[1]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_ftn1" \o ")**。Dijkstra还指出，没有谁的大脑能容得下一个现代的计算机程序，也就是说，作为软件开发人员，我们不应该试着在同一时间把整个程序都塞进自己的大脑，而应该试着以某种方式去组织程序，以便能够在一个时刻可以专注于一个特定的部分。这么做的目的是尽量减少在任一时间所要考虑的程序量，你需要同时记住的东西越多，就越可能漏掉其中的某一个，从而导致设计或编码的错误。

尽管谁都希望成为英雄，自如地应对各种计算机问题，但没有人的大脑真正有能力同时掌握9个数量级的细节。计算机科学和软件工程已经开发了许多智力工具，来应对这种复杂度，《代码大全》围绕这一主题作了详尽的讨论。

■    在架构层将系统划分为多个子系统，以便让思绪在某段时间内能专注于系统的一小部分。（第5章）

■    仔细定义类接口，从而可以忽略类内部的工作机理。（第6.1节）

■    保持类接口的抽象性，从而不必记住不必要的细节。（第6.2节）

■    避免全局变量，因为它会大大增加总是需要兼顾的代码比例。（第13.3节）

■    避免深层次的继承，因为这样会耗费很大精力。（第6.3节）

■    避免深度嵌套的循环或条件判断，因为它们都能用简单的控制结构取代，后者占用较少的脑力资源。（第19.4节）

■    别用goto语句，因为它们引入了非顺序执行，多数人都不容易弄懂。（第17.3节）

■    小心定义错误处理的方法，不要滥用不同的错误处理技术。（第8.3节）

■    以系统的观点对待内置的异常机制，后者会成为非线性的控制结构。异常如果不受约束地使用，会和goto一样难以理解。（第8.4节）

■    不要让类过度膨胀，以至于占据整个程序。（第6章）

■    子程序应保持短小。（第7.4节）

■    使用清楚、不言自明的变量名，从而大脑不必费力记住诸如“i代表账号下标，j代表顾客下标，还是另有它意？”之类的细节。（第11章）

■    传递给子程序的参数数目应尽量少。更重要的是，只传递保持子程序接口抽象所必需的参数。（第7.5节）

■    用规范和约定来使大脑从记忆不同代码段的随意性、偶然性差异中解脱出来。（第4.2节，第31、32章）

■    只要有可能，一般情况下应避免“偶然性困难”**[[2]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_ftn2" \o ")**。（第5.2节）

如果将复杂的逻辑判断代码放入布尔函数，并将其意图概括出来，就可以降低代码的复杂程度（第19.1节）。用查表法代替繁琐的逻辑链，也能达到同样目的（第18章）。如果采用定义良好的一致的类接口，你就无须操心类的实现细节，从而整体上简化自己的工作。

采用编码规范主要也是为了降低复杂度。如果在格式编排、循环、变量命名、建模表示法等方面有统一的考虑，就能将精力集中于更具挑战性的编码问题上。规范最有用之处在于它们能免于你做出任意决定，省却了为之辩解的麻烦。

各种形式的抽象对于管理复杂度都是很强大的工具。通过增强程序组件的抽象性，编程领域已经取得了很大的进步。Fred Brooks指出，计算机的科学最了不起的成就，就是从机器语言跃进到高级语言，解放了程序员——我们不用再操心某种特定的硬件细节，而能够专心于编程。子程序的想法则是另一个巨大的进步，随后的重要进步是类和程序包。

以其功能对变量命名，说明问题是什么，而非其怎样实现，能提升其抽象层次。如果你说：“这是弹出栈，意味着我在取最近雇员的信息”，那么抽象使你可以省掉记住“弹出栈”的脑力步骤，你只需简单地说“我在取最近雇员的信息。”这一长进是微不足道的，但当你要减少从1到109这么大范围的复杂度时，任何改进措施都是值得的，勿以善小而不为。采用具名常量而非文字量（神秘数值）也能提高抽象级别。面向对象的编程方法提供同时适用于算法和数据的抽象，单靠功能分解做不到这一点。

总而言之，软件设计与构建的主要目标就是征服复杂度。许多编程实践背后的动机正是为了降低程序的复杂度。降低复杂度几乎是衡量程序员成果的最重要依据。这是《代码大全》体现的最主要的编程思想。（虽然这本书从头至尾没有正式提到过“编程思想”这个词。）

以上讨论“抽象”的文字本身也够抽象的，下面谈谈具体的、看得见摸得着的代码。

**何谓“高质量的代码”**

就“高质量的代码”而言，正确性、简单性、清晰性是首要的**[SA04, Item 6]**，可测试性也同样重要。清晰性（可读性）是“易于维护、易于重构的程序”最有价值的特性。若无法读懂代码，你就不能有信心地修改，也无法调试和修正错误。阅读代码的次数要比编写代码多得多，即使在开发的初期也是如此。因此，为了让编写代码更方便而降低代码的可读性是非常不经济的。一项可读性原则是应该把修改你代码的人记在心上。编程首先是与人交流，其次才是与计算机交流。代码的维护者会感激你使代码容易理解——而且将来的维护者很可能就是你自己，到时候你得尝试记起自己六个月以前在想什么。

可测试性指的是能很方便地用自动化的手段来测试你的程序，把代码应有的功能用另一种形式（测试用例）描述一遍，等于给代码再加一道保险，降级出错的可能。下面两句话经常被引用来说明代码可读性的重要。

程序必须是写给人看的，仅仅偶尔才在机器上执行。——Harold Abelson等人**[SICP]**

编写程序首先为人，其次为计算机。——Steve McConnell**[M04, Section 34.3]**

与之相比，复用性、高效率就显得不那么重要，“使正确的程序变快”远远比“使快的程序变正确”容易得多**[SA04, Item 8]**。清晰的代码更容易写正确，更容易理解，更易于重构——因此更易于性能优化。

至于富于技巧(tricky)、聪明(clever)更可算是代码的恶劣品质，编程不是为了炫耀自己的聪明程度，这样写程序简直是歪门邪道。不能手里拿个铁锤，就把满世界都看成钉子。（比如在重载操作符的时候应该保持其自然语义，否则宁可用具名子程序来实现相同的操作。**[SA04, Item 26]**）Dijkstra在1972年的图灵奖演讲会上宣读了一篇名为《The Humble Programmer》（谦卑的程序员）的文章。他认为大部分编程工作都旨在弥补我们有限的智力。精通编程的人是那些了解自己头脑有多大局限性的人，都很谦虚。而那些编程糟糕的人，总是拒绝接受自己脑瓜不能胜任工作的事实，自负使得他们无法成为优秀的程序员。研究表明，谦虚的程序员善于弥补其不足之处，使用能奏效的最简单的技术，所编写的代码让自己和他人都易看懂，其中的错误也较少。

调试代码的难度是首次编写这些代码的两倍。因此，如果你在编写代码的时候就已经发挥了全部聪明才智，那么按照常理，你将无法凭借自己的智慧去调试这些代码。——Brian Kernighan**[KP78]**

那么代码的可读性具体体现在哪些方面呢？大致有以下几点。一是名字，最常见的有类名、子程序名、变量名；二是长度，比如类的长度、数据成员的数目、子程序的长度、子程序的参数数目、语句的嵌套层数；三是复杂度，包括表达式的复杂度、语句逻辑的复杂度等；四是耦合度，包括由于共享数据（含全局数据）导致的耦合、类之间耦合（继承、组合、友元）、子程序之间的耦合等；五是格式，包括缩进、空格、注释等。

下面谈谈其中三点影响代码可读性的因素。

**名不正则言不顺**

“为变量命名”恐怕是编程中最普通的一项活动，一般介绍编程风格的书都会用几页的篇幅给出一些好的建议**[KP99, Section 1.1]**，而《代码大全》用了整整一章30多页的篇幅（第11章）来讨论变量的命名，另外第7.3节专门讨论子程序的命名，第6.2节讨论了类的命名。如果变量、子程序和类型命名得当，代码本身就能用作程序的文档，可以减少注释和外部文档（第32.2节）。

**变量名**

为变量命名时最重要的考虑事项是，该名字要完全、准确地描述出该变量所代表的事物。currentDate和todaysDate都是很好的名字，因为它们都完全而且准确地描述出了“当前日期”这一概念。事实上，这两个名字都用了非常直白的词。程序员们有时候会忽视这些普通词语，而它们往往却是最明确的。cd和c是很糟的命名，因为它们太短，同时又不具有描述性。current也很糟，因为它并没有告诉你是当前的什么。date看上去不错，但经过最后推敲它也只是个坏名字，因为这里所说的日期并不是所有的日期均可，而只是特指当前日期；而date本身并未表达出这层含义。x、x1和x2永远是坏名字——传统上用x代表一个未知量；如果你不希望你的变量所代表的是一个未知量，那么请考虑取一个更好的名字吧。

名字应该尽可能地明确。像x、temp、i这些名字都泛泛得可以用于多种目的，它们并没有像应该的那样提供足够信息，因此通常都是命名上的败笔。有人也许会反驳说，把i用作循环下标是最正常不过的了，难道非得写成indexOfTheLoop这种又臭又长的名字才算好吗？

Steve McConnell认为，如果循环只有寥寥数行，而且只是单层循环，那么用i是也是可行的。不过试想一下，如果你一直习惯用i作循环下标，而你将来可能需要把这个循环放到另一个循环中去执行，即循环嵌套，那么内外层循环都用i作下标肯定是不行的。如果编译器提醒你说变量i重复定义，那还算走运；如果编译器默不作声，而你自己又忘了修改，呃，你听见虫子飞舞的声音了吗？由于代码会经常修改、扩充，或者复制到其他程序中去，因此很多有经验的程序员索性不使用类似于i这样的名字。

如果循环不是只有几行，那么代码阅读者会很容易忘记i本来具有的含义，因此最好给循环下标换一个更有意义的名字。导致循环变长的常见原因之一是出现循环的嵌套使用。如果你使用了多个嵌套的循环，那么就应该给循环变量赋予更长的名字以提高可读性：

for (int teamIndex = 0; teamIndex < teamCount; teamIndex++){

   for (int eventIndex = 0; eventIndex < eventCount[teamIndex]; eventIndex++){

     score[teamIndex][eventIndex] = 0;

   }

}

谨慎地为循环下标变量命名可以避免产生常见的下标串话（index cross-talk）问题：想用j的时候写了i，想用i的时候却写了j。同时这也使得数据访问变得更加清晰：score[teamIndex][eventIndex]要比score[i][j]给出的信息更多。

如果你一定要用i、j、k，那么不要把它们用于简单循环的循环下标之外的任何场合——这种传统已经太深入人心了，一旦违背该原则，将这些变量用于其他用途就可能造成误解。要想避免出现这样的问题，最简单的方法就是想出一个比i、j、k更具描述性的名字来。

变量名的最佳长度似乎应该介于x和maximum­­Number­­Of­­Points­­In­­Modern­­Olympics之间。太短的名字无法传达足够的信息。诸如x1和x2这样的名字所存在的问题是，即使你知道了x代表什么，也无法获知x1和x2之间的关系。太长的名字很难写，同时也会使得程序的视觉结构变得模糊不清。

研究发现，当变量名的平均长度在10到16个字符的时候，调试程序所需花费的气力是最小的。平均名字长度在8到20个字符的程序也几乎同样容易调试。这项原则并不意味着你应该尽量把变量名的长度控制在9到15或者10到16个字符。它强调的是，如果你查看自己写的代码时发现了很多更短的名字，那么需要认真检查，确保这些名字含义足够清晰。

**子程序名**

好的子程序名字能清晰地描述子程序所做的一切。《代码大全》第7.3节列举并详细说明了若干条指导原则：描述子程序所做的所有事情，避免使用无意义的、模糊或表述不清的动词，不要仅通过数字来形成不同的子程序名字，根据需要确定子程序名字的长度，给函数命名时要对返回值有所描述，给过程起名时使用语气强烈的动词加宾语**[[3]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_ftn3" \o ")**的形式，准确使用对仗词，为常用操作确立命名规则等。

**类名**

类的名称应该表达了其中心目的，准确的描述该类的接口所模塑的抽象概念（第5.3节），一般用名词。

无论如何，命名不是一锤子买卖，一旦发现有更好的名称，借助现代的IDE工具（Eclipse或Visual Studio 2005），我们很容易对变量、常量、类、子程序进行重命名（rename），这恐怕也是用得最多的一项重构操作了。

**宜短不宜长**

很多编程书籍都告诉我们不要写过长的子程序**[SA04, Item 20]**，那么子程序写多长才合适呢？《代码大全》第7.4节专门讨论了这个问题。与一般书不同的是，McConnell不是以一个先知的口吻说“汝当如何如何”，而是列出了学术界的十多项研究成果，然后分析作结论。

有研究表明，子程序的长度与错误量成反比，即：随着子程序长度的增加（上至200行代码），每行代码所包含的错误数量就会减少。另一项研究则发现，子程序的长度与错误量没有关联，而结构复杂度以及数据量却与错误量有关。还有研究发现，短小的子程序（含有32行或更少代码）与更低的成本或错误率无关。有证据表明，较长的子程序（含有65行或更多代码）使得每行代码的成本更低。……IBM所做的一项研究发现，最容易出错的是那些超过500行代码的子程序。超过500行之后，子程序的出错率就会与其长度成正比。

这似乎与我们平时接受的“子程序越短越好”的教导相违背。McConnell认为，在任何时候，复杂的算法总会导致更长的子程序。在这种情况下，可以允许子程序的长度有序地增长到100至200行（不算源代码中的注释行和空行）。数十年的证据表明，这么长的子程序也和短小的子程序一样不易出错。与其对子程序的长度强加限制，还不如让其他因素——如子程序的内聚性、嵌套的层次、变量的数量、决策点的数量、解释子程序用意所需的注释数量以及其他一些跟复杂度相关的考虑事项等——来决定子程度的长度。

这就是说，如果要编写一段超过200行代码的子程序，那就要小心了。对于超过200行代码的子程序来说，没有哪项研究发现它能降低成本和/或降低出错率，而且在超过200行后，迟早会在可读性方面遇到问题。

不过，话说回来，在一开始写程序的时候，可以不必在意这个限制。在写完一个子程序，实现了应有的功能，并通过单元测试之后，如果它过长，我们可以很容易地用Extract Method重构法对它进行改进，使之符合项目编码标准中规定的子程序长度。

McConnell似乎对数字7情有独钟，他建议把子程序的参数个数限制在大约7个以内（第7.4节），告诉我们要警惕拥有超过约7个数据成员的类，并把基类的派生类总数（注意不是继承体系的层数）限制在7±2等（第6.3节）。当然，书中都给出了理由，这里就不赘述了。

**格式与规范**

《代码大全》第31章专门介绍代码的布局与风格，前面提到过，编码规范最有用之处在于让你避免做出武断决定，避免把时间花在无谓的争执上（第34.5节）。McConnell并不像一位“家具警察”那样对待代码的格式，他认为好的代码布局应凸现程序的逻辑结构，使代码易于阅读、理解、检查及修改。至于循环体应该缩进几个空格，大括号的摆放位置这些问题，正确答案不止一种。每次回答同样内容比起只是回答正确更重要。

第28.5节谈到了程序员的信仰问题，缩进风格、大括号的摆放位置、注释风格、命名习惯、对goto的使用、对全局变量的使用等等都是十分敏感的话题。关于这种问题，我觉得Herb Sutter和Andrei Alexandrescu的观点更贴近程序员的想法**[SA04, Item 0]**。

那些“仅仅是个人品味、而不影响正确性或可读性的”议题不应出现在编码标准中。任何一个专业的程序员都应该能轻易地阅读并编写“那种格式与自己的习惯略有不同的”代码。

每个源文件（甚至每个项目）内确保采用一致的编排格式，因为在同一块代码中切换若干种风格是很不和谐的。但是不要试图对多个项目（甚至对整个公司）强制使用相同的编排格式。

……重要的不是设定格式规则，而仅仅是与“你维护的文件中已经采用的”格式保持一致。

　　■    不要指明缩进多少字符，但缩进要显出结构：你愿意用多少个空格来缩排都行，但至少每个文件保持一致。

　　■    不要规定每行的长度，但确保可读性：你愿意每行多长就多长，只要别太过分就行。研究表明，最适合人眼阅读的情况是每行不超过10个单词。

　　■    不要规定注释的风格（某些工具将特定风格的注释提取为文档的情况除外），但一定编写有用的注释：只要有可能，尽量以代码代替注解。不要编写与代码重复的注解；这些注解会逐渐变得与代码不同步。一定编写说明性的注解，以解释所用的方法和基本原理。

……

关于大括号的摆放，以下数种做法在可读性上没有区别：

void using\_k\_and\_r\_style() {

    // ...

}

void putting\_each\_brace\_on\_its\_own\_line()

{

    // ...

}

void or\_putting\_each\_brace\_on\_its\_own\_line\_indented()

    {

    // ...

    }

每个专业的程序员都能毫无困难地阅读并编写以上任何一种风格的代码。但一定要保持一致：不要随意放置大括号，不要让作用域的嵌套关系变得混淆，并尽量遵循各个文件已有的风格。

如果你已经知道什么样的代码才是高质量的，那么怎样才能编写出这种代码呢？McConnell认为，好习惯很重要，因为程序员做的大部分事情都是无意识完成的（第33.9节）。例如，你曾想过该如何格式化缩进的循环体，但现在每当写新的循环体时就不再去想了，而以习惯的方式来做。对程序格式的方方面面几乎都是如此。你上次质疑编排风格是什么时候？如果你有五年编程经验，最后一次提出这个问题多半是在四年半之前，其余时间都是按习惯编程的。Bill Gates说过，任何日后出色的程序员在入行的前几年就做得很好，从那以后，程序员的优劣就定型了。其实任何行当都是如此，因此在初涉编程时，就应端正态度来学，尽快培养良好的习惯。

衷心祝愿你在阅读《代码大全》之后，成为一名具有编写高质量代码的习惯的优秀程序员。

**参考文献**

[M04]     Steve McConnell. *Code Complete*, 2d ed, 2004. 中译本《代码大全 第2版》，电子工业出版社，2006年。

[SA04]    Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*, 2004. 中译本《C++编程规范：101条规则、准则与最佳实践》，人民邮电出版社，2006年。

[SICP]    Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*, 2d ed, 1996. 中译本《计算机程序的构造和解释（原书第2版）》，机械工业出版社，2004年。

[KP99]    Brian W. Kernighan and Rob Pike. *The Practice of Programming*, 1999. 中译本《程序设计实践》，机械工业出版社，2001年。

[KP78]    Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*, 2d ed, 1978.

**[[1]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_ftnref1" \o ")** Dijkstra这番话是在1989年说的，现在可能已经涨到了15个数量级。

**[[2]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_ftnref2" \o ")** 关于软件开发的偶然性问题和本质性问题，见Fred Brooks那篇具有里程碑意义的文章：《没有银弹：软件工程中本质性和偶然性》，收录于《人月神话（二十周年纪念版）》。《代码大全》第5.2节复述了其主要内容。

**[[3]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_ftnref3" \o ")** 宾语一词在英语中也是object，这里当然不能把object翻译为“目标”或“对象”。