第 3 章 我编写过的最漂亮的代码

Jon Bentley

我曾经听一位大师级的程序员这样称赞到，“我通过删除代码来实现功能的提升。”而法国著名作家兼飞行家 Antoine de Saint-Exupéry 的说法则更具代表性，“只有在不仅没有任何功能可以添加，而且也没有任何功能可以删除的情况下，设计师才能够认为自己的工作已臻完美。” 某些时候，在软件中根本就不存在最漂亮的代码，最漂亮的函数，或者最漂亮的程序。

当然，我们很难对不存在的事物进行讨论。本章将对经典 Quicksort（快速排序）算法的运行时间进行全面的分析，并试图通过这个分析来说明上述观点。在第一节中，我将首先根据我自己的观点来回顾一下 Quicksort，并为后面的内容打下基础。第二节的内容将是本章的重点部分。我们将首先在程序中增加一个计数器，然后通过不断地修改，从而使程序的代码变得越来越短，但程序的功能却会变得越来越强，最终的结果是只需要几行代码就可以使算法的运行时间达到平均水平。在第三节将对前面的技术进行小结，并对二分搜索树的运行开销进行简单的分析。最后的两节将给出学完本章得到的一些启示，这将有助于你在今后写出更为优雅的程序。

3.1 我编写过的最漂亮代码

当 Greg Wilson 最初告诉我本书的编写计划时，我曾自问编写过的最漂亮的代码是什么。这个有趣的问题在我脑海里盘旋了大半天，然后我发现答案其实很简单：Quicksort 算法。但遗憾的是，根据不同的表达方式，这个问题有着三种不同的答案。

当我撰写关于分治（divide-and-conquer）算法的论文时，我发现 C.A.R. Hoare 的 Quicksort 算法（“Quicksort”，Computer Journal 5）无疑是各种 Quicksort 算法的鼻祖。这是一种解决基本问题的漂亮算法，可以用优雅的代码实现。我很喜欢这个算法，但我总是无法弄明白算法中最内层的循环。我曾经花两天的时间来调试一个使用了这个循环的复杂程序，并且几年以来，当我需要完成类似的任务时，我会很小心地复制这段代码。虽然这段代码能够解决我所遇到的问题，但我却并没有真正地理解它。

我后来从 Nico Lomuto 那里学到了一种优雅的划分（partitioning）模式，并且最终编写出了我能够理解，甚至能够证明的 Quicksort 算法。William Strunk Jr. 针对英语所提出的“良好的写作风格即为简练”这条经验同样适用于代码的编写，因此我遵循了他的建议，“省略不必要的字词”（来自《The Elements of Style》一书）。我最终将大约 40 行左右的代码缩减为十几行的代码。因此，如果要回答“你曾编写过的最漂亮代码是什么？”这个问题，那么我的答案就是：在我编写的《Programming Pearls, Second Edition》(Addison-Wesley)一书中给出的 Quichsort 算法。在示例 3-1 中给出了用 C 语言编写的 Quicksort 函数。我们在接下来的章节中将进一步地研究和改善这个函数。

【示例】 3-1 Quicksort 函数

```
void quicksort(int l, int u)
{   int i, m;
    if (l >= u) return;
```

```
    swap(l, randint(l, u));
    m = l;
    for (i = l+1; i <= u; i++)
        if (x[i] < x[l])
            swap(++m, i);
    swap(l, m);
    quicksort(l, m-1);
    quicksort(m+1, u);
}
```

如果函数的调用形式是 quicksort(0, n-1)，那么这段代码将对一个全局数组 x[n]进行排序。函数的两个参数分别是将要进行排序的子数组的下标：l 是较低的下标，而 u 是较高的下标。函数调用 swap(i,j)将会交换 x[i]与 x[j]这两个元素。第一次交换操作将会按照均匀分布的方式在 l 和 u 之间随机地选择一个划分元素。

在《Programming Pearls》一书中包含了对 Quicksort 算法的详细推导以及正确性证明。在本章的剩余内容中，我将假设读者熟悉在《Programming Pearls》中所给出的 Quicksort 算法以及在大多数初级算法教科书中所给出的 Quicksort 算法。

如果你把问题改为“在你编写那些广为应用的代码中，哪一段代码是最漂亮的？”我的答案还是 Quicksort 算法。在我和 M. D. McIlroy 一起编写的一篇文章("Engineering a sort function," Software-Practice and Experience, Vol. 23, No. 11）中指出了在原来 Unix qsort 函数中的一个严重的性能问题。随后，我们开始用 C 语言编写一个新排序函数库，并且考虑了许多不同的算法，包括合并排序（Merge Sort）和堆排序（Heap Sort）等算法。在比较了 Quicksort 的几种实现方案后，我们着手创建自己的 Quicksort 算法。在这篇文章中描述了我们如何设计出一个比这个算法的其他实现要更为清晰，速度更快以及更为健壮的新函数——部分原因是由于这个函数的代码更为短小。Gordon Bell 的名言被证明是正确的：“在计算机系统中，那些最廉价，速度最快以及最为可靠的组件是不存在的。”现在，这个函数已经被使用了 10 多年的时间，并且没有出现任何故障。

考虑到通过缩减代码量所得到的好处，我最后以第三种方式来问自己在本章之初提出的问题。“你没有编写过的最漂亮代码是什么？”。我如何使用非常少的代码来实现大量的功能？答案还是和 Quicksort 有关，特别是对这个算法的性能分析。我将在下一节给出详细介绍。

3.2 事倍功半

Quicksort 是一种优雅的算法，这一点有助于对这个算法进行细致的分析。大约在 1980 年左右，我与 Tony Hoare 曾经讨论过 Quicksort 算法的历史。他告诉我，当他最初开发出 Quicksort 时，他认为这种算法太简单了，不值得发表，而且直到能够分析出这种算法的预期运行时间之后，他才写出了经典的“Quicksoft”论文。

我们很容易看出，在最坏的情况下，Quicksort可能需要n^2的时间来对数组元素进行排序。而在最优的情况下，它将选择中值作为划分元素，因此只需nlgn次的比较就可以完成对数组的排序。那么，对于n个不同值的随机数组来说，这个算法平均将进行多少次比较？

Hoare 对于这个问题的分析非常漂亮，但不幸的是，其中所使用的数学知识超出了大多数程序员的理解范围。当我为本科生讲授 Quicksort 算法时，许多学生即使在费了很大的努力之后，还是无法理解其中的证明过程，这令我非常沮丧。下面，我们将从 Hoare 的程序开

始讨论，并且最后将给出一个与他的证明很接近的分析。

我们的任务是对示例 3-1 中的 Quicksort 代码进行修改，以分析在对元素值均不相同的数组进行排序时平均需要进行多少次比较。我们还将努力通过最短的代码、最短运行时间以及最小存储空间来得到最深的理解。

为了确定平均比较的次数，我们首先对程序进行修改以统计次数。因此，在内部循环进行比较之前，我们将增加变量 comps 的值（参见示例 3-2）。

【示例 3-2】 修改 Quicksort 的内部循环以统计比较次数。

```
for (i = l+1; i <= u; i++) {
    comps++;
    if (x[i] < x[l])
        swap(++m, i);
}
```

如果用一个值 n 来运行程序，我们将会看到在程序的运行过程中总共进行了多少次比较。如果重复用 n 来运行程序，并且用统计的方法来分析结果，我们将得到 Quicksort 在对 n 个元素进行排序时平均使用了 1.4 nlgn 次的比较。

在理解程序的行为上，这是一种不错的方法。通过十三行的代码和一些实验可以反应出许多问题。这里，我们引用作家 Blaise Pascal 和 T. S. Eliot 的话，“如果我有更多的时间，那么我给你写的信就会更短。”现在，我们有充足的时间，因此就让我们来对代码进行修改，并且努力编写出更短（同时更好）的程序。

我们要做的事情就是提高这个算法的速度，并且尽量增加统计的精确度以及对程序的理解。由于内部循环总是会执行 u-l 次比较，因此我们可以通过在循环外部增加一个简单的操作来统计比较次数，这就可以使程序运行得更快一些。在示例 3-3 的 Quicksort 算法中给出了这个修改。

【示例 3-3】 Quicksort 的内部循环，将递增操作移到循环的外部

```
comps += u-l;
for (i = l+1; i <= u; i++)
    if (x[i] < x[l])
        swap(++m, i);
```

这个程序会对一个数组进行排序，同时统计比较的次数。不过，如果我们的目标只是统计比较的次数，那么就不需要对数组进行实际地排序。在示例 3-4 中去掉了对元素进行排序的“实际操作”，而只是保留了程序中各种函数调用的“框架”。

【示例 3-4】将 Quicksort 算法的框架缩减为只进行统计

```
void quickcount(int l, int u)
{   int m;
    if (l >= u) return;
    m = randint(l, u);
    comps += u-l;
    quickcount(l, m-1);
    quickcount(m+1, u);
}
```

这个程序能够实现我们的需求，因为 Quichsort 在选择划分元素时采用的是“随机”方式，并且我们假设所有的元素都是不相等的。现在，这个新程序的运行时间与 n 成正比，并且相对于示例 3-3 需要的存储空间与 n 成正比来说，现在所需的存储空间缩减为递归堆栈的大小，即存储空间的平均大小与 lgn 成正比。

虽然在实际的程序中，数组的下标（l 和 u）是非常重要的，但在这个框架版本中并不重要。因此，我们可以用一个表示子数组大小的整数(n)来替代这两个下标（参见示例 3-5）

【示例 3-5】 在 Quicksort 代码框架中使用一个表示子数组大小的参数

```
void qc(int n)
{   int m;
    if (n <= 1) return;
    m = randint(1, n);
    comps += n-1;
    qc(m-1);
    qc(n-m);
}
```

现在，我们可以很自然地把这个过程整理为一个统计比较次数的函数，这个函数将返回在随机 Quicksort 算法中的比较次数。在示例 3-6 中给出了这个函数。

【示例 3-6】 将 Quicksort 框架实现为一个函数

```
int cc(int n)
{   int m;
    if (n <= 1) return 0;
    m = randint(1, n);
    return n-1 + cc(m-1) + cc(n-m);
}
```

在示例 3-4、示例 3-5 和示例 3-6 中解决的都是相同的基本问题，并且所需的都是相同的运行时间和存储空间。在后面的每个示例都对这些函数的形式进行了改进，从而比这些函数更为清晰和简洁。

在定义发明家的矛盾（inventor's paradox）(How To Solve It, Princeton University Press)时，George Póllya 指出“计划越宏大，成功的可能性就越大。”现在，我们就来研究在分析 Quicksort 时的矛盾。到目前为止，我们遇到的问题是，“当 Quicksort 对大小为 n 的数组进行一次排序时，需要进行多少次比较？”我们现在将对这个问题进行扩展，“对于大小为 n 的随机数组来说，Quichsort 算法平均需要进行多少次的比较？”我们通过对示例 3-6 进行扩展以引出示例 3-7。

【示例 3-7】 伪码：Quicksort 的平均比较次数

```
float c(int n)
    if (n <= 1) return 0
    sum = 0
    for (m = 1; m <= n; m++)
        sum += n-1 + c(m-1) + c(n-m)
    return sum/n
```

如果在输入的数组中最多只有一个元素，那么 Quichsort 将不会进行比较，如示例 3-6

中所示。对于更大的 n，这段代码将考虑每个划分值 m（从第一个元素到最后一个，每个都是等可能的）并且确定在这个元素的位置上进行划分的运行开销。然后，这段代码将统计这些开销的总和（这样就递归地解决了一个大小为 m-1 的问题和一个大小为 n-m 的问题），然后将总和除以 n 得到平均值并返回这个结果。

如果我们能够计算这个数值，那么将使我们实验的功能更加强大。我们现在无需对一个 n 值运行多次来估计平均值，而只需一个简单的实验便可以得到真实的平均值。不幸的是，实现这个功能是要付出代价的：这个程序的运行时间正比于 3n（如果是自行参考（self-referential）的，那么用本章中给出的技术来分析运行时间将是一个很有趣的练习）。

示例 3-7 中的代码需要一定的时间开销，因为它重复计算了中间结果。当在程序中出现这种情况时，我们通常会使用动态编程来存储中间结果，从而避免重复计算。因此，我们将定义一个表 t[N+1]，其中在 t[n]中存储 c[n]，并且按照升序来计算它的值。我们将用 N 来表示 n 的最大值，也就是进行排序的数组的大小。在示例 3-8 中给出了修改后的代码。

【示例 3-8】 在 Quicksort 中使用动态编程来计算

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += n-1 + t[i-1] + t[n-i]
    t[n] = sum/n
```

这个程序只对示例 3-7 进行了细微的修改，即用 t[n]来替换 c(n)。它的运行时间将正比于 N2，并且所需的存储空间正比于 N。这个程序的优点之一就是：在程序执行结束时，数组 t 中将包含数组中从元素 0 到元素 N 的真实平均值（而不是样本均值的估计）。我们可以对这些值进行分析，从而生成在 Quichsort 算法中统计比较次数的计算公式。

我们现在来对程序做进一步的简化。第一步就是把 n-1 移到循环的外面，如示例 3-9 所示。

【示例 3-9】 在 Quicksort 中把代码移到循环外面来计算

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += t[i-1] + t[n-i]
    t[n] = n-1 +  sum/n
```

现在将利用对称性来对循环做进一步的调整。例如，当 n 为 4 时，内部循环计算总和为：

```
t[0]+t[3] + t[1]+t[2] + t[2]+t[1] + t[3]+t[0]
```

在上面这些组对中，第一个元素增加而第二个元素减少。因此，我们可以把总和改写为：

```
2 * (t[0] + t[1] + t[2] + t[3])
```

我们可以利用这种对称性来得到示例 3-10 中的 Quicksort。

【示例 3-10】 在 Quichsort 中利用了对称性来计算

```
t[0] = 0
```

```
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 0; i < n; i++)
        sum += 2 * t[i]
    t[n] = n-1 + sum/n
```

然而，在这段代码的运行时间中同样存在着浪费，因为它重复地计算了相同的总和。此时，我们不是把前面所有的元素加在一起，而是在循环外部初始化总和并且加上下一个元素，如示例 3-11 所示。

【示例 3-11】 在 Quicksort 中删除了内部循环来计算

```
sum = 0; t[0] = 0
for (n = 1; n <= N; n++)
    sum += 2*t[n-1]
    t[n] = n-1 + sum/n
```

这个小程序确实很有用。程序的运行时间与 N 成正比，对于每个从 1 到 N 的整数，程序将生成一张 Quicksort 的估计运行时间表。

我们可以很容易地把示例 3-11 用表格来实现，其中的值可以立即用于进一步的分析。在 3-1 给出了最初的结果行。

表 3-1 示例 3-11 中实现的表格输出

N	Sum	t[n]
0	0	0
1	0	0
2	0	1
3	2	2.667
4	7.333	4.833
5	17	7.4
6	31.8	10.3
7	52.4	13.486
8	79.371	16.921

这张表中的第一行数字是用代码中的三个常量来进行初始化的。下一行（输出的第三行）的数值是通过以下公式来计算的：

```
A3 = A2+1      B3 = B2 + 2*C2      C3 = A3-1 + B3/A3
```

把这些（相应的）公式记录下来就使得这张表格变得完整了。这张表格是“我曾经编写的最漂亮代码”的很好的证据，即使用少量的代码完成大量的工作。

但是，如果我们不需要所有的值，那么情况将会是什么样？如果我们更希望通过这种来方式分析一部分数值（例如，在 20 到 232 之间所有 2 的指数值）呢？虽然在示例 3-11 中构建了完整的表格 t，但它只需要使用表格中的最新值。因此，我们可以用变量 t 的定长空间来替代 table t[]的线性空间，如示例 3-12 所示。

【示例 3-12】 Quicksoft 计算——最终版本

```
sum = 0; t = 0
```

```
for (n = 1; n <= N; n++)
    sum += 2*t
    t = n-1 + sum/n
```

然后，我们可以插入一行代码来测试 n 的适应性，并且在必要时输出这些结果。

这个程序是我们漫长学习旅途的终点。通过本章所采用的方式，我们可以证明 Alan Perlis 的经验是正确的：“简单性并不是在复杂性之前，而是在复杂性之后”（"Epigrams on Programming," Sigplan Notices, Vol. 17, Issue 9）。

3.3 观点

在表 3-2 中总结了本章中对 Quicksort 进行分析的程序。

表 3-2 对 Quicksort 比较次数的统计算法的评价

示例编号	代码行数	答案类型	答案数量	运行时间	空间
2	13	Sample	1	$n \lg n$	N
3	13	〃	〃	〃	〃
4	8	〃	〃	n	$\lg n$
5	8	〃	〃	〃	〃
6	6	〃	〃	〃	〃
7	6	Exact	〃	$3N$	N
8	6	〃	N	N^2	N
9	6	〃	〃	〃	〃
10	6	〃	〃	〃	〃
11	4	〃	〃	N	〃
12	4	Exact	N	N	1

在我们对代码的每次修改中，每个步骤都是很直接的；不过，从示例 3-6 中样本值到示例 3-7 中准确值的过渡过程可能是最微妙的。随着这种方式进行下去，代码变得更快和更有用，而代码量同样得到了缩减。在 19 世纪中期，Robert Browning 指出“少即是多（less is more）”，而这张表格正是一个证明这种极少主义哲学（minimalist philosophy）的实例。

我们已经看到了三种截然不同的类型的程序。示例 3-2 和示例 3-3 是能够实际使用的 Quicksort，可以用来在对真实数组进行排序时统计比较次数。示例 3-4 到示例 3-6 都实现了 Quicksort 的一种简单模型：它们模拟算法的运行，而实际上却没有做任何排序工作。从示例 3-7 到示例 3-12 则实现了一种更为复杂的模型：它们计算了比较次数的真实平均值而没有跟踪任何单次的运行。

我们在下面总结了实现每个程序所使用的技术：

* 示例 3-2，示例 3-4，3-7:对问题的定义进行根本的修改。
* 示例 3-5，示例 3-6，3-12：对函数的定义进行轻微的修改
* 示例 3-8：实现动态编程的新数据结构

这些技术都是非常典型的。我们在简化程序时经常要发出这样的疑问，“我们真正要解

决的问题是什么？”或者是，“有没有更好的函数来解决这个问题？”

当我把这个分析过程讲授给本科生时，这个程序最终被缩减成零行代码，化为一阵数学的轻烟消失了。我们可以把示例 3-7 重新解释为以下的循环关系：

$$C_0 = 0 \quad C_n = (n-1) + (1/n)\sum_{1 \le i \le n} C_{i-1} + C_{n-i}$$

这正是 Hoare 所采用的方法，并且后来由 D.E.Knuth 在他经典的《The Art of Computer Programming》（Addison-Wesley）一书的第三卷：排序与查找中给出的方法中给出了描述。通过重新表达编程思想的技巧和在示例 3-10 中使用的对称性，使我们可以把递归部分简化为：

$$C_n = n - 1 + (2/n)\sum_{0 \le i \le n-1} C_i$$

Knuth 删除了求和符号，从而引出了示例 3-11，这可以被重新表达为一个在两个未知量之间有着两种循环关系的系统：

$$C_0 = 0 \quad S_0 = 0 \quad S_n = S_{n-1} + 2C_{n-1} \quad C_n = n - 1 + S_n / n$$

Knuth 使用了“求和因子”的数学方法来实现这种解决方案：

$$C_n = (n+1)(2H_{n+1} - 2) - 2_n \sim 1.386 n \lg n$$

其中 H_n 表示第 n 个调和数（harmonic number），即 1 + 1/2 + 1/3 + … 1/n。这样，我们就从对程序不断进行修改以得到实验数据顺利地过渡到了对程序行为进行完全的数学分析。

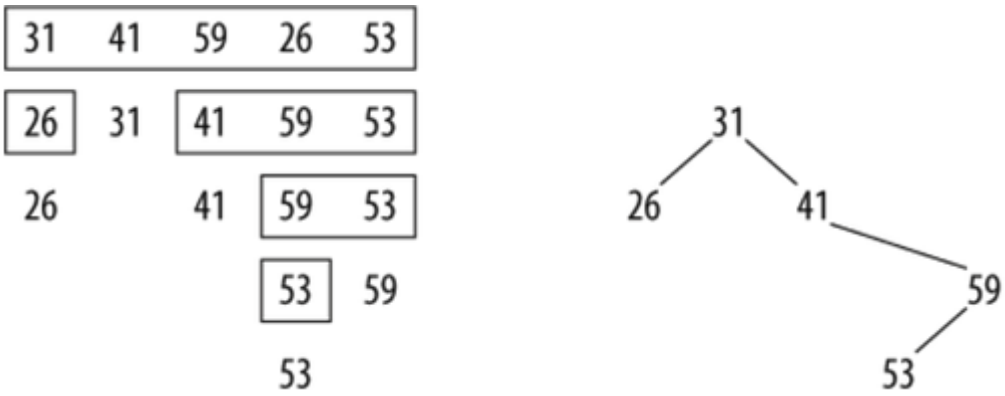
在得到这个公式之后，我们就可以结束我们的讨论。我们已经遵循了 Einstein 的著名建议：“尽量使每件事情变得简单，并且直到不可能再简单为止。”

附加分析

Goethe 的著名格言是：“建筑是静止的音乐”。按照这种说法，我可以说“数据结构是静止的算法。”如果我们固定了 Quichsort 算法，那么就将得到了一个二分搜索树的数据结构。在 Knuth 发表的文章中给出了这个结构并且采用类似于在 Quichsort 中的循环关系来分析它的运行时间。

如果要分析把一个元素插入到二分搜索树中的平均开销，那么我们可以以这段代码作为起点，并且对这段代码进行扩展来统计比较次数，然后在我们收集的数据上进行实验。接下来，我们可以仿照前面章节中的方式来简化代码。一个更为简单的解决方案就是定义一个新的 Quichsort，在这个算法中使用理想的划分算法把有着相同关联顺序的元素划分到两边。Quichsort 和二分搜索树是同构的，如图 3-1 所示。

图 3-1　实现理想划分的 Quicksort 以及相应的二分搜索树



左边的方框给出了正在进行中的理想划分的 Quicksort，右边的图则给出了相应的从相同输入中构建起来的二分搜索树。这两个过程不仅需要进行相同次数的比较，而且还将生成相同的比较集合。通过在前面对于在一组不同元素上进行 Quicksort 实验的平均性能分析，我们就可以得到将不同的元素随机插入到二分搜索树中的平均比较次数。

3.4 本章的中心思想是什么？

表面上看来，我“所写的”内容就是从示例 3-2 到示例 3-12 的程序。我最初是漫不经心地编写这些程序，然后将这些程序写在给本科生讲课的黑板上，并且最终写到本章中。我有条不紊地进行着这些程序的修改，并且花了大量的时间来分析这些程序，从而确信它们都是正确的。然而，除了在示例 3-11 中实现的表格外，我从来没有把任何一个示例作为计算机程序运行过。

我在贝尔实验室呆了将近二十年，我从许多教师（尤其是 Brian Kernighan，他所编写的编程内容作为本书的第 1 章）那里学到了：要“编写”一个在大众面前展示的程序，所涉及到的东西比键入这个程序要多得多。有人用代码实现了这个程序，最初运行在一些测试示例中，然后构建了完整的系统框架、驱动程序以及一个案例库来支撑这段代码。理想的情况是，人们可以手动地把编译后的代码包含到文本中，不加入任何的人为干涉。基于这种想法，我编写了示例 3-1（以及在《Programming Pearls》中的所有代码）。

为了维护面子，我希望永远都不要实现从示例 3-2 到示例 3-12 的代码，从而使我保持诚实的名声。然而，在计算机编程中的近四十年的实践使我对这个任务的困难性有着深深的敬畏（好吧，更准确地说，是对于错误的害怕）。我妥协了，把示例 3-11 用表格方式实现出来，并且无意中得到了一个完备的解答。当这两个东西完美地匹配在一起时，你可以想象一下我当时的喜悦吧！因此，我向世界提供了这些漂亮的并且未曾实现的程序，虽然在这些程序中可能会有一些还未发现的错误，但我对这些程序的正确性还是有一定信心的。我希望一些细微的错误不会掩盖我在这些程序中所展示的那些漂亮思想。

当我为给出这些没有被实现过的程序感到不安时，Alan Perlis 的话安慰了我，他说“软件是不是不像任何一个事物，它就是意味着被抛弃：软件的所有意义就是把它看作为一个肥皂泡？”

3.5 结论

漂亮的含义有着许多来源。本章通过简化、优雅以及精简来刻画了漂亮的含义。下面这

些名言表达的是同样的意思：

* 通过删除代码来实现功能的提升。
* 只有在不仅没有任何功能可以添加，而且也没有任何功能可以删除的情况下，设计师才能够认为自己的工作已臻完美。
* 有时候，在软件中根本就不存在最漂亮的代码，最漂亮的函数，或者最漂亮的程序。
* 良好的写作风格即为简练。省略不必要的字词。 (Strunk and White)
* 在计算机系统中，那些最廉价、速度最快以及最为可靠的组件是不存在的（Bell）
* 努力做到事倍功半。
* 如果我有更多的时间，那么我给你写的信就会越短（Pascal）
* 发明家的矛盾：计划越宏大，成功的可能性就越大。（Pólya）
* 简单性并不是在复杂性之前，而是在复杂性之后（Perlis）
* 少即是多。（Browning）
* 尽量使每件事情变得简单，并且直到不可能再简单为止（Einstein）
* 软件有时候应该被视作为一个肥皂泡（Perlis）
* 在简单中寻找漂亮。

本章的内容到此结束。读者可以复习所学到的内容并进行模拟实验。

对于那些想要得到更具体信息的人们，我在下面给出了一些观点，这些观点分为三类

程序分析

深入理解程序行为的方式之一就是修改这个程序，然后在具有代表性的数据上运行这个程序，就像示例 3-2 那样。不过，我们通常会更关心程序的某个方面而不是程序的整体。例如，我们只是考虑Quichsort所使用的平均比较次数，而忽略了其他的方面。Sedgewick ("The analysis of Quicksort programs," Acta Informatica, Vol. 7)研究了 Quichsort 的其他特性，例如算法所需的存储空间以及各种 Quicksort 运行时间的其他方面。我们可以关注这些关键问题，而暂时）忽略了程序其他不太重要的方面。在我的一篇文章"A Case Study in Applied Algorithm Design" (IEEE Computer, Vol. 17, No. 2)中指出了我曾经遇到过的一个问题：对在单元空间中找出货郎行走路线的 strip 启发式算法的性能进行评价。我估计完成这个任务所要的程序大概在 100 行代码左右。在经历了一系列类似于本章前面看到的分析步骤之后，我只使用了十几行代码的模拟算法就实现了更为精确的效果（在我写完了这个模拟算法后，我发现 Beardwood 等人["The Shortest Path Through Many Points," Proc. Cambridge Philosophical Soc., Vol. 55]已经更完整地表述了我的模拟算法，因此已经在二十几年前就从数学上解决了这个问题）。

小段代码

我相信计算机编程是一项实践性的技术，并且我也同意这个观点：“任何技术都必须通过模仿和实践来掌握。” 因此，想要编写漂亮代码的程序员应该阅读一些漂亮的程序以及在编写程序时模仿所学到的技术。我发现在实践时有个非常有用的东西就是小段代码，也就是一二十行的代码。编写《Programming Pearls》这本书是一件艰苦的工作，但同时也有着极大的乐趣。我实现了每一小段代码，并且亲自把每段代码都分解为基本的知识。我希望其他人在阅读这些代码时与我在编写这些代码时有着同样的享受过程。

软件系统

为了有针对性，我极其详尽地描述了一个小型任务。我相信其中的这些准则不仅存在于小型程序中，它们同样也适用于大型的程序以及计算机系统。Parnas（"Designing software for ease of extension and contraction," IEEE T. Software Engineering, Vol. 5, No. 2）给出了把一个系统拆分为基本构件的技术。为了得用快速的应用性，不要忘了 Tom Duff 的名言："在尽可能的情况下，利用现有的代码。"

3.6 致谢

非常感谢 Dan Bentley, Brian Kernighan, Andy Oram 和 David Weiss 卓有见识的评语。