HZ BOOKS

PEARSON Addison Wesley

名著名译　双语重温

程序设计实践

The Practice of Programming

双语版

（美）Brian W. Kernighan　Rob Pike 著

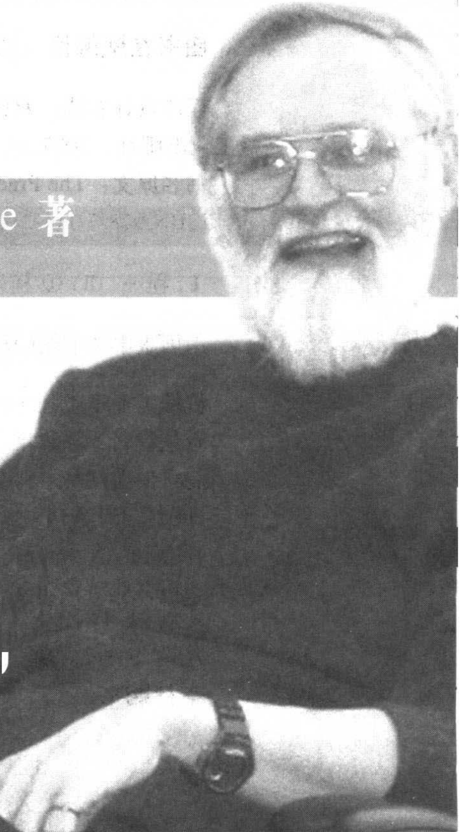
裘宗燕　译

机械工业出版社
China Machine Press

程序设计实践

The Practice of Programming

双语版



(美) Brian W. Kernighan Rob Pike 著

裘宗燕 译

机械工业出版社
China Machine Press

本书是著名计算机专家的畅销作品，曾在国内外受到广泛赞誉。本书从排错、测试、性能、可移植性、设计、界面、风格和记法等方面，讨论了程序设计中实际的、又是非常深刻和具有广泛意义的思想、技术和方法。本书值得每个梦想并努力使自己成为优秀程序员的人参考，值得每个计算机专业的学生和计算机工作者阅读，也可作为程序设计高级课程的教材或参考书。

本书版权登记号：图字：01-2007-1756

图书在版编目（CIP）数据

程序设计实践：双语版 /（美）柯奈汉（Kernighan, B. W.）著；裘宗燕译. —北京：机械工业出版社，2007.3
书名原文：The Practice of Programming
ISBN 978-7-111-21127-3

Ⅰ. 程…　Ⅱ. ① 柯…　② 裘…　Ⅲ. 程序设计—英、汉　Ⅳ. TP311.1

中国版本图书馆CIP数据核字（2007）第033486号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：陈冀康
北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行
2007年3月第1版第1次印刷
186mm × 240mm · 30印张
定价：59.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：(010) 68326294

Preface

Have you ever...

wasted a lot of time coding the wrong algorithm?
used a data structure that was much too complicated?
tested a program but missed an obvious problem?
spent a day looking for a bug you should have found in five minutes?
needed to make a program run three times faster and use less memory?
struggled to move a program from a workstation to a PC or vice versa?
tried to make a modest change in someone else's program?
rewritten a program because you couldn't understand it?

Was it fun?

These things happen to programmers all the time. But dealing with such problems is often harder than it should be because topics like testing, debugging, portability, performance, design alternatives, and style—the *practice* of programming—are not usually the focus of computer science or programming courses. Most programmers learn them haphazardly as their experience grows, and a few never learn them at all.

In a world of enormous and intricate interfaces, constantly changing tools and languages and systems, and relentless pressure for more of everything, one can lose sight of the basic principles—simplicity, clarity, generality—that form the bedrock of good software. One can also overlook the value of tools and notations that mechanize some of software creation and thus enlist the computer in its own programming.

Our approach in this book is based on these underlying, interrelated principles, which apply at all levels of computing. These include *simplicity*, which keeps programs short and manageable; *clarity*, which makes sure they are easy to understand, for people as well as machines; *generality*, which means they work well in a broad range of situations and adapt well as new situations arise; and *automation*, which lets the machine do the work for us, freeing us from mundane tasks. By looking at computer programming in a variety of languages, from algorithms and data structures through design, debugging, testing, and performance improvement, we can illustrate universal engineering concepts that are independent of language, operating system, or programming paradigm.

This book comes from many years of experience writing and maintaining a lot of software, teaching programming courses, and working with a wide variety of programmers. We want to share lessons about practical issues, to pass on insights from our experience, and to suggest ways for programmers of all levels to be more proficient and productive.

We are writing for several kinds of readers. If you are a student who has taken a programming course or two and would like to be a better programmer, this book will expand on some of the topics for which there wasn't enough time in school. If you write programs as part of your work, but in support of other activities rather than as the goal in itself, the information will help you to program more effectively. If you are a professional programmer who didn't get enough exposure to such topics in school or who would like a refresher, or if you are a software manager who wants to guide your staff in the right direction, the material here should be of value.

We hope that the advice will help you to write better programs. The only prerequisite is that you have done some programming, preferably in C, C++ or Java. Of course the more experience you have, the easier it will be; nothing can take you from neophyte to expert in 21 days. Unix and Linux programmers will find some of the examples more familiar than will those who have used only Windows and Macintosh systems, but programmers from any environment should discover things to make their lives easier.

The presentation is organized into nine chapters, each focusing on one major aspect of programming practice.

Chapter 1 discusses programming style. Good style is so important to good programming that we have chosen to cover it first. Well-written programs are better than badly-written ones—they have fewer errors and are easier to debug and to modify—so it is important to think about style from the beginning. This chapter also introduces an important theme in good programming, the use of idioms appropriate to the language being used.

Algorithms and data structures, the topics of Chapter 2, are the core of the computer science curriculum and a major part of programming courses. Since most readers will already be familiar with this material, our treatment is intended as a brief review of the handful of algorithms and data structures that show up in almost every program. More complex algorithms and data structures usually evolve from these building blocks, so one should master the basics.

Chapter 3 describes the design and implementation of a small program that illustrates algorithm and data structure issues in a realistic setting. The program is implemented in five languages; comparing the versions shows how the same data structures are handled in each, and how expressiveness and performance vary across a spectrum of languages.

Interfaces between users, programs, and parts of programs are fundamental in programming and much of the success of software is determined by how well interfaces are designed and implemented. Chapter 4 shows the evolution of a small library for parsing a widely used data format. Even though the example is small, it illustrates many of the concerns of interface design: abstraction, information hiding, resource management, and error handling.

Much as we try to write programs correctly the first time, bugs, and therefore debugging, are inevitable. Chapter 5 gives strategies and tactics for systematic and effective debugging. Among the topics are the signatures of common bugs and the importance of ``numerology,'' where patterns in debugging output often indicate where a problem lies.

Testing is an attempt to develop a reasonable assurance that a program is working correctly and that it stays correct as it evolves. The emphasis in Chapter 6 is on systematic testing by hand and machine. Boundary condition tests probe at potential weak spots. Mechanization and test scaffolds make it easy to do extensive testing with modest effort. Stress tests provide a different kind of testing than typical users do and ferret out a different class of bugs.

Computers are so fast and compilers are so good that many programs are fast enough the day they are written. But others are too slow, or they use too much memory, or both. Chapter 7 presents an orderly way to approach the task of making a program use resources efficiently, so that the program remains correct and sound as it is made more efficient.

Chapter 8 covers portability. Successful programs live long enough that their environment changes, or they must be moved to new systems or new hardware or new countries. The goal of portability is to reduce the maintenance of a program by minimizing the amount of change necessary to adapt it to a new environment.

Computing is rich in languages, not just the general-purpose ones that we use for the bulk of programming, but also many specialized languages that focus on narrow domains. Chapter 9 presents several examples of the importance of notation in computing, and shows how we can use it to simplify programs, to guide implementations, and even to help us write programs that write programs.

To talk about programming, we have to show a lot of code. Most of the examples were written expressly for the book, although some small ones were adapted from other sources. We've tried hard to write our own code well, and have tested it on half a dozen systems directly from the machine-readable text. More information is available at the web site for *The Practice of Programming*:

```
http://tpop.awl.com
```

The majority of the programs are in C, with a number of examples in C++ and Java and some brief excursions into scripting languages. At the lowest level, C and C++ are almost identical and our C programs are valid C++ programs as well. C++ and Java are lineal descendants of C, sharing more than a little of its syntax and much of its efficiency and expressiveness, while adding richer type systems and libraries.

In our own work, we routinely use all three of these languages, and many others. The choice of language depends on the problem: operating systems are best written in an efficient and unrestrictive language like C or C++; quick prototypes are often easiest in a command interpreter or a scripting language like Awk or Perl; for user interfaces, Visual Basic and Tcl/Tk are strong contenders, along with Java.

There is an important pedagogical issue in choosing a language for our examples. Just as no language solves all problems equally well, no single language is best for presenting all topics. Higher-level languages preempt some design decisions. If we use a lower-level language, we get to consider alternative answers to the questions; by exposing more of the details, we can talk about them better. Experience shows that even when we use the facilities of high-level languages, it's invaluable to know how they relate to lower-level issues; without that insight, it's easy to run into performance problems and mysterious behavior. So we will often use C for our examples, even though in practice we might choose something else.

For the most part, however, the lessons are independent of any particular programming language. The choice of data structure is affected by the language at hand; there may be few options in some languages while others might support a variety of alternatives. But the way to approach making the choice will be the same. The details of how to test and debug are different in different languages, but strategies and tactics are similar in all. Most of the techniques for making a program efficient can be applied in any language.

Whatever language you write in, your task as a programmer is to do the best you can with the tools at hand. A good programmer can overcome a poor language or a clumsy operating system, but even a great programming environment will not rescue a bad programmer. We hope that, no matter what your current experience and skill, this book will help you to program better and enjoy it more.

We are deeply grateful to friends and colleagues who read drafts of the manuscript and gave us many helpful comments. Jon Bentley, Russ Cox, John Lakos, John Linderman, Peter Memishian, Ian Lance Taylor, Howard Trickey, and Chris Van Wyk read the manuscript, some more than once, with exceptional care and thoroughness. We are indebted to Tom Cargill, Chris Cleeland, Steve Dewhurst, Eric Grosse, Andrew Herron, Gerard Holzmann, Doug McIlroy, Paul McNamee, Peter Nelson, Dennis Ritchie, Rich Stevens, Tom Szymanski, Kentaro Toyama, John Wait, Daniel C. Wang, Peter Weinberger, Margaret Wright, and Cliff Young for invaluable comments on drafts at various stages. We also appreciate good advice and thoughtful suggestions from Al Aho, Ken Arnold, Chuck Bigelow, Joshua Bloch, Bill Coughran, Bob Flandrena, Renée French, Mark Kernighan, Andy Koenig, Sape Mullender, Evi Nemeth, Marty Rabinowitz, Mark V. Shaney, Bjarne Stroustrup, Ken Thompson, and Phil Wadler. Thank you all.

Brian W. Kernighan

Rob Pike

Contents/目录

注：斜体页码为中文部分页码

1 Style

It is an old observation that the best writers sometimes disregard the rules of rhetoric. When they do so, however, the reader will usually find in the sentence some compensating merit, attained at the cost of the violation. Unless he is certain of doing as well, he will probably do best to follow the rules.

William Strunk and E. B. White, *The Elements of Style*

This fragment of code comes from a large program written many years ago:

```
if ( (country == SING) || (country == BRNI) ||
     (country == POL) || (country == ITALY) )
{
        /*
         * If the country is Singapore, Brunei or Poland
         * then the current time is the answer time
         * rather than the off hook time.
         * Reset answer time and set day of week.
         */
        ...
```

It's carefully written, formatted, and commented, and the program it comes from works extremely well; the programmers who created this system are rightly proud of what they built. But this excerpt is puzzling to the casual reader. What relationship links Singapore, Brunei, Poland and Italy? Why isn't Italy mentioned in the comment? Since the comment and the code differ, one of them must be wrong. Maybe both are. The code is what gets executed and tested, so it's more likely to be right; probably the comment didn't get updated when the code did. The comment doesn't say enough about the relationship among the three countries it does mention; if you had to maintain this code, you would need to know more.

The few lines above are typical of much real code: mostly well done, but with some things that could be improved.

This book is about the practice of programming—how to write programs for real. Our purpose is to help you to write software that works at least as well as the program this example was taken from, while avoiding trouble spots and weaknesses. We will talk about writing better code from the beginning and improving it as it evolves.

We are going to start in an unusual place, however, by discussing programming style. The purpose of style is to make the code easy to read for yourself and others, and good style is crucial to good programming. We want to talk about it first so you will be sensitive to it as you read the code in the rest of the book.

There is more to writing a program than getting the syntax right, fixing the bugs, and making it run fast enough. Programs are read not only by computers but also by programmers. A well-written program is easier to understand and to modify than a poorly-written one. The discipline of writing well leads to code that is more likely to be correct. Fortunately, this discipline is not hard.

The principles of programming style are based on common sense guided by experience, not on arbitrary rules and prescriptions. Code should be clear and simple—straightforward logic, natural expression, conventional language use, meaningful names, neat formatting, helpful comments—and it should avoid clever tricks and unusual constructions. Consistency is important because others will find it easier to read your code, and you theirs, if you all stick to the same style. Details may be imposed by local conventions, management edict, or a program, but even if not, it is best to obey a set of widely shared conventions. We follow the style used in the book *The C Programming Language*, with minor adjustments for C++ and Java.

We will often illustrate rules of style by small examples of bad and good programming, since the contrast between two ways of saying the same thing is instructive. These examples are not artificial. The "bad" ones are all adapted from real code, written by ordinary programmers (occasionally ourselves) working under the common pressures of too much work and too little time. Some will be distilled for brevity, but they will not be misrepresented. Then we will rewrite the bad excerpts to show how they could be improved. Since they are real code, however, they may exhibit multiple problems. Addressing every shortcoming would take us too far off topic, so some of the good examples will still harbor other, unremarked flaws.

To distinguish bad examples from good, throughout the book we will place question marks in the margins of questionable code, as in this real excerpt:

```
?    #define ONE 1
?    #define TEN 10
?    #define TWENTY 20
```

Why are these `#define`s questionable? Consider the modifications that will be necessary if an array of `TWENTY` elements must be made larger. At the very least, each name should be replaced by one that indicates the role of the specific value in the program:

```
     #define INPUT_MODE 1
     #define INPUT_BUFSIZE 10
     #define OUTPUT_BUFSIZE 20
```

1.1 Names

What's in a name? A variable or function name labels an object and conveys information about its purpose. A name should be informative, concise, memorable, and pronounceable if possible. Much information comes from context and scope; the broader the scope of a variable, the more information should be conveyed by its name.

Use descriptive names for globals, short names for locals. Global variables, by definition, can crop up anywhere in a program, so they need names long enough and descriptive enough to remind the reader of their meaning. It's also helpful to include a brief comment with the declaration of each global:

```
    int npending = 0;  // current length of input queue
```

Global functions, classes, and structures should also have descriptive names that suggest their role in a program.

By contrast, shorter names suffice for local variables; within a function, `n` may be sufficient, `npoints` is fine, and `numberOfPoints` is overkill.

Local variables used in conventional ways can have very short names. The use of `i` and `j` for loop indices, `p` and `q` for pointers, and `s` and `t` for strings is so frequent that there is little profit and perhaps some loss in longer names. Compare

```
?   for (theElementIndex = 0; theElementIndex < numberOfElements;
?           theElementIndex++)
?       elementArray[theElementIndex] = theElementIndex;
```

to

```
    for (i = 0; i < nelems; i++)
        elem[i] = i;
```

Programmers are often encouraged to use long variable names regardless of context. That is a mistake: clarity is often achieved through brevity.

There are many naming conventions and local customs. Common ones include using names that begin or end with `p`, such as `nodep`, for pointers; initial capital letters for `Globals`; and all capitals for `CONSTANTS`. Some programming shops use more sweeping rules, such as notation to encode type and usage information in the variable, perhaps `pch` to mean a pointer to a character and `strTo` and `strFrom` to mean strings that will be written to and read from. As for the spelling of the names themselves, whether to use `npending` or `numPending` or `num_pending` is a matter of taste; specific rules are much less important than consistent adherence to a sensible convention.

Naming conventions make it easier to understand your own code, as well as code written by others. They also make it easier to invent new names as the code is being written. The longer the program, the more important is the choice of good, descriptive, systematic names.

Namespaces in C++ and packages in Java provide ways to manage the scope of names and help to keep meanings clear without unduly long names.

Be consistent. Give related things related names that show their relationship and highlight their difference.

Besides being much too long, the member names in this Java class are wildly inconsistent:

```
?   class UserQueue {
?       int noOfItemsInQ, frontOfTheQueue, queueCapacity;
?       public int noOfUsersInQueue() {...}
?   }
```

The word "queue" appears as Q, Queue and queue. But since queues can only be accessed from a variable of type UserQueue, member names do not need to mention "queue" at all; context suffices, so

```
?   queue.queueCapacity
```

is redundant. This version is better:

```
    class UserQueue {
        int nitems, front, capacity;
        public int nusers() {...}
    }
```

since it leads to statements like

```
    queue.capacity++;
    n = queue.nusers();
```

No clarity is lost. This example still needs work, however: "items" and "users" are the same thing, so only one term should be used for a single concept.

Use active names for functions. Function names should be based on active verbs, perhaps followed by nouns:

```
    now = date.getTime();
    putchar('\n');
```

Functions that return a boolean (true or false) value should be named so that the return value is unambiguous. Thus

```
?   if (checkoctal(c)) ...
```

does not indicate which value is true and which is false, while

```
    if (isoctal(c)) ...
```

makes it clear that the function returns true if the argument is octal and false if not.

Be accurate. A name not only labels, it conveys information to the reader. A misleading name can result in mystifying bugs.

One of us wrote and distributed for years a macro called isoctal with this incorrect implementation:

```
?      #define isoctal(c) ((c) >= '0' && (c) <= '8')
```

instead of the proper

```
       #define isoctal(c) ((c) >= '0' && (c) <= '7')
```

In this case, the name conveyed the correct intent but the implementation was wrong; it's easy for a sensible name to disguise a broken implementation.

Here's an example in which the name and the code are in complete contradiction:

```
?      public boolean inTable(Object obj) {
?          int j = this.getIndex(obj);
?          return (j == nTable);
?      }
```

The function `getIndex` returns a value between zero and `nTable-1` if it finds the object, and returns `nTable` if not. The boolean value returned by `inTable` is thus the opposite of what the name implies. At the time the code is written, this might not cause trouble, but if the program is modified later, perhaps by a different programmer, the name is sure to confuse.

Exercise 1-1. Comment on the choice of names and values in the following code.

```
?      #define TRUE 0
?      #define FALSE 1
?
?      if ((ch = getchar()) == EOF)
?          not_eof = FALSE;
```

□

Exercise 1-2. Improve this function:

```
?      int smaller(char *s, char *t) {
?          if (strcmp(s, t) < 1)
?              return 1;
?          else
?              return 0;
?      }
```

□

Exercise 1-3. Read this code aloud:

```
?      if ((falloc(SMRHSHSCRTCH, S_IFEXT|0644, MAXRODDHSH)) < 0)
?          ...
```

□

1.2 Expressions and Statements

By analogy with choosing names to aid the reader's understanding, write expressions and statements in a way that makes their meaning as transparent as possible. Write the clearest code that does the job. Use spaces around operators to suggest grouping; more generally, format to help readability. This is trivial but valuable, like keeping a neat desk so you can find things. Unlike your desk, your programs are likely to be examined by others.

Indent to show structure. A consistent indentation style is the lowest-energy way to make a program's structure self-evident. This example is badly formatted:

```
?       for(n++;n<100;field[n++]='\0');
?       *i = '\0'; return('\n');
```

Reformatting improves it somewhat:

```
?       for (n++; n < 100; field[n++] = '\0')
?           ;
?       *i = '\0';
?       return('\n');
```

Even better is to put the assignment in the body and separate the increment, so the loop takes a more conventional form and is thus easier to grasp:

```
        for (n++; n < 100; n++)
            field[n] = '\0';
        *i = '\0';
        return '\n';
```

Use the natural form for expressions. Write expressions as you might speak them aloud. Conditional expressions that include negations are always hard to understand:

```
?       if (!(block_id < actblks) || !(block_id >= unblocks))
?           ...
```

Each test is stated negatively, though there is no need for either to be. Turning the relations around lets us state the tests positively:

```
        if ((block_id >= actblks) || (block_id < unblocks))
            ...
```

Now the code reads naturally.

Parenthesize to resolve ambiguity. Parentheses specify grouping and can be used to make the intent clear even when they are not required. The inner parentheses in the previous example are not necessary, but they don't hurt, either. Seasoned programmers might omit them, because the relational operators (< <= == != >= >) have higher precedence than the logical operators (&& and ||).

When mixing unrelated operators, though, it's a good idea to parenthesize. C and its friends present pernicious precedence problems, and it's easy to make a mistake.

Because the logical operators bind tighter than assignment, parentheses are mandatory for most expressions that combine them:

```
        while ((c = getchar()) != EOF)
             ...
```

The bitwise operators & and | have lower precedence than relational operators like ==, so despite its appearance,

```
?       if (x&MASK == BITS)
?            ...
```

actually means

```
?       if (x & (MASK==BITS))
?            ...
```

which is certainly not the programmer's intent. Because it combines bitwise and relational operators, the expression needs parentheses:

```
        if ((x&MASK) == BITS)
             ...
```

Even if parentheses aren't necessary, they can help if the grouping is hard to grasp at first glance. This code doesn't need parentheses:

```
?       leap_year = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
```

but they make it easier to understand:

```
        leap_year = ((y%4 == 0) && (y%100 != 0)) || (y%400 == 0);
```

We also removed some of the blanks: grouping the operands of higher-precedence operators helps the reader to see the structure more quickly.

Break up complex expressions. C, C++, and Java have rich expression syntax and operators, and it's easy to get carried away by cramming everything into one construction. An expression like the following is compact but it packs too many operations into a single statement:

```
?       *x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

It's easier to grasp when broken into several pieces:

```
        if (2*k < n-m)
            *xp = c[k+1];
        else
            *xp = d[k--];
        *x += *xp;
```

Be clear. Programmers' endless creative energy is sometimes used to write the most concise code possible, or to find clever ways to achieve a result. Sometimes these skills are misapplied, though, since the goal is to write clear code, not clever code.