China-pub.com
下载

第3章　设计与实现

给我看你的流程图而藏起你的表，我将仍然是莫名其妙。如果给我你的表，那么我将不再要你的流程图，因为它们太明显了。

—— Frederick P. Brooks, Jr.，《神秘的人月》

以上从Brooks的经典书中摘录的内容想说的是，数据结构设计是程序构造过程的中心环节。一旦数据结构安排好了，算法就像是瓜熟蒂落，编码也比较容易。

这种观点虽然有点过于简单化，但也不是在哄骗人。在前一章里我们考察了各种基本数据结构，它们是许多程序的基本构件。在这一章中，我们将组合这些结构，要完成的工作是设计和实现一个中等规模的程序。我们将说明被处理的问题将如何影响数据结构，从这里还可以看到，一旦数据结构安排好之后，代码将会如何自然地随之而来。

我们的观点的另一个方面是：程序设计语言的选择在整个设计过程中，相对而言，并不是那么重要。我们将抽象地设计这个程序，然后用 C、C++、Awk和Perl把它写出来。由不同实现之间的比较，可以看出语言在这里能有什么帮助或者妨碍，以及它们并不那么重要的各种情况。程序的设计当然可以通过语言来装饰，但是通常不会为语言所左右。

我们要选择的问题并不是很常见的，但它在基本形式上又是非常典型的：一些数据进去，另一些数据出来，其处理过程并不依赖于多少独创性。

我们准备做的就是产生一些随机的可以读的英语文本。如果随便扔出来一些随机字母或随机的词，结果当然是毫无意义的。例如，一个随机选取字母 (以及空格，用作词之间的分隔) 的程序可能产生：

```
xptmxgn xusaja  afqnzgxl  lhidlwcd rjdjuvpydrlwnjy
```

没人会觉得这有什么意思。如果以字母在英语里出现的频率作为它们的权重，我们可能得到下面这样的内容：

```
idtefoae tcs trder jcii ofdslnqetacp t ola
```

这个也好不到哪儿去。采用从字典里随机选择词的方式也弄不出多少意思来：

```
polydactyl equatorial splashily jowl verandah circumscribe
```

为了得到更好一些的结果，我们需要一个带有更多内在结构，例如包含着各短语出现频率的统计模型。但是，我们怎么才能得到这种统计呢？

我们当然可以抓来一大堆英语材料，仔细地研究。但是，实际上有一种更简单也更有意思的方法。这里有一个关键性的认识：用任何一个现成的某种语言的文本，可以构造出由这个文本中的语言使用情况而形成的统计模型。通过该模型生成的随机文本将具有与原文本类似的统计性质。

3.1　马尔可夫链算法

完成这种处理有一种非常漂亮的方法，那就是使用一种称为**马尔可夫链算法**的技术。我

China-pub.com
下载

们可以把输入想像成由一些互相重叠的短语构成的序列，而该算法把每个短语分割为两个部分：一部分是由多个词构成的前缀，另一部分是只包含一个词的后缀。马尔可夫链算法能够生成输出短语的序列，其方法是依据(在我们的情况下)原文本的统计性质，随机性地选择跟在前缀后面的特定后缀。采用三个词的短语就能够工作得很好——利用连续两个词构成的前缀来选择作为后缀的一个词：

设置w_1和w_2为文本的前两个词
输出w_1和w_2
循环：
    随机地选出w_3，它是文本中w_1w_2的后缀中的一个
    打印w_3
    把w_1和w_2分别换成w_2和w_3
    重复循环

为了说明问题，假设我们要基于本章开头的引语里的几个句子生成一些随机文本。这里采用的是两词前缀：

```
Show your flowcharts and conceal your tables and I will be
mystified.  Show your tables and your flowcharts will be
obvious.  (end)
```

下面是一些输入的词对和跟随它们之后的词：

输入前缀	跟随的后缀词
Show your	flowcharts tables
your flowcharts	and will
flowcharts and	conceal
flowcharts will	be
your tables	and and
will be	mystified. obvious.
be mystified.	Show
be obvious.	(end)

处理这个文本的马尔可夫算法将首先打印出`Show your`，然后随机取出`flowcharts`或`table`。如果选中了前者，那么现在前缀就变成`your flowcharts`，而下一个词应该是`and`或`will`。如果它选取`tables`，下一个词就应该是`and`。这样继续下去，直到产生出足够多的输出，或者在找后缀时遇到了结束标志。

我们的程序将读入一段英语文本，并利用马尔可夫链算法，基于文本中固定长度的短语的出现频率，产生一段新文本。前缀中词的数目是个参数，上面用的是 2。如果将前缀缩短，产生出来的东西将趋向于无聊词语，更加缺乏内聚力；如果加长前缀，则趋向于产生原始输入的逐字拷贝。对于英语文本而言，用两个词的前缀选择第三个是一个很好的折衷方式。看起来它既能重现输入的风味，又能加上程序的古怪润饰。

什么是一个词？最明显的回答是字母表字符的一个序列。这里我们更愿意把标点符号也附着在词后，把“ words ”和“ words. ”看成是不同的词，这样做将有利于改进闲话生成的质量。加上标点符号，以及(间接的)语法将影响词的选择，虽然这种做法也可能会产生不配对的引语和括号。我们将把“ 词 ”定义为任何实际位于空格之间的内容，这对输入语言并没有造成任何限制，但却将标点符号附到了词上。许多语言里都有把文本分割成“ 空白界定的词 ”的功能，这个功能也很容易自己实现。

China-pub.com 下载

根据这里所采用的方法，输出中所有的词、所有的两词短语以及所有三个词的短语都必然是原来输入中出现过的，但是，也会有许多四个词或更多个词的短语将被组合产生出来。下面几个句子是由我们在本章里将要开发的程序生成的，给它提供的文本是艾尼思特·海明威的《太阳也升起来》的第4章：

> As I started up the underskin came his chest black, and his stomach muscles bulging under the light. "You see them?" Below the one where his ribs stopped were two raised white welts. "See on the forehead." "Oh, Brett, I love you." "Let's not talk. Talking's all bilge. I'm going away tomorrow." "Tomorrow?" "Yes. Didn't I say so? Isn." "Let's have a drink, then."

我们很幸运，在这里标点符号没出问题。实际中却未必总能这样。

3.2 数据结构的选择

有多少输入需要处理？程序应该运行得多快？要求程序读完一整本书并不是不合理的，因此我们需要准备对付输入规模 n=100 000个词甚至更多的情况。输出将包括几百甚至几千个词，而程序的运行应该在若干秒内完成，而不是几分钟。假定输入文本有 100 000个词，n已经相当大了，因此，如果还要求程序运行得足够快，这个算法就不会太简单。

马尔可夫算法必须在看到了所有输入之后才能开始产生输出。所以它必须以某种形式存储整个输入。一个可能的方式是读完整个输入，将它存为一个长长的字符串。情况的另一方面也很明显，输入必须被分解成词。如果另用一个指向文本中各词的指针数组，输出的生成将很简单：产生一个词，扫描输入中的词，看看刚输出的前缀有哪些可能的后缀，然后随机选取一个。但是，这个方法意味着生成每个词都需要扫描 100 000个输入词。1000个输出就意味着上亿次字符串比较。这样做肯定快不了。

另一种可能性是存储单个的词，给每个词关联一个链表，指出该词在文本中的位置。这样就可以对词进行快速定位。在这里可以使用第 2章提出的散列表。但是，这种方式并没有直接触及到马尔可夫算法的需要。在这里最需要的是能够由前缀出发快速地确定对应的后缀。

我们需要有一种数据结构，它能较好地表示前缀以及与之相关联的后缀。程序将分两部分，第一部分是输入，它构造表示短语的整个数据结构；第二部分是随后的输出，它使用这个数据结构，生成随机的输出。这两部分都需要 (快速地)查询前缀：输入过程中需要更新与前缀相关的后缀；输出时需要对可能后缀做随机选择。这些分析提醒我们使用一种散列结构，其关键码是前缀，其值是对应于该前缀的所有可能后缀的集合。

为了描述的方便，我们将假定采用二词前缀，在这种情况下，每个输出词都是根据它前面的一对词得到的。前缀中词的个数对设计本身并没有影响，程序应该能对付任意的前缀长度，但给定一个数能使下面的讨论更具体些。我们把一个前缀和它所有可能后缀的集合放在一起，称其为一个状态，这是马尔可夫算法的标准术语。

对于一个特定前缀，我们需要存储所有能跟随它的后缀，以便将来取用。这些后缀是无序的，一次一个地加进去。我们不知道后缀将会有多少，因此，需要一种能容易且高效地增长的数据结构，例如链表或者动态数组。在产生输出的时候，我们要能从关联于特定前缀的后缀集合中随机地选出一个后缀。还有，数据项绝不会被删除。

如果一个短语出现多次，那么又该怎么办？例如，短语“ might appear twice ”可能在文

China-pub.com
下载

本里出现两次，而“ might appear once ”只出现了一次。这个情况有两种可能的表示方式：或者在“ might appear ”的后缀表里放两个“ twice ”；或者是只放一个，但还要给它附带一个计数值为2的计数器。我们对用或不用计数器的方式都做过试验。不用计数器的情况处理起来比较简单，因为在加入后缀时不必检查它是否已经存在。试验说明这两种方式在执行时间上的差别是微不足道的。

总结一下：每个状态由一个前缀和一个后缀链表组成。所有这些信息存在一个散列表里，以前缀作为关键码。每个前缀是一个固定大小的词集合。如果一个后缀在给定前缀下的出现多于一次，则每个出现都单独包含在有关链表里。

下面的问题是如何表示词本身。最简单的方法是把它们存储为独立的字符串。在一般文本里总是有许多反复出现的词，如果为单词另外建一个散列表有可能节约存储，因为在这种情况下每个词只需要存储一次。此外，这样做还能加快前缀散列的速度，因为这时每个词都只有一个惟一地址，可以直接比较指针而不是比较词里的各个字符。我们把这种做法留做练习。下面采用每个词都分开存放的方式。

3.3 在C中构造数据结构

现在开始考虑C语言中的实现。首先是定义一些常数：

```
enum {
    NPREF   = 2,    /* number of prefix words */
    NHASH   = 4093, /* size of state hash table array */
    MAXGEN  = 10000 /* maximum words generated */
};
```

这个声明定义了前缀中词的个数 (NPREF)，散列表数组的大小 (NHASH)，生成词数的上界 (MAXGEN)。如果NPREF是个编译时的常数而不是运行时的变量，程序里的存储管理将会更简单些。数组的规模设得相当大，因为我们预计程序可能处理很大的输入文件，或许是整本书。选择NHASH=4093，这样，即使输入里有10 000个不同前缀(词对)，平均链长仍然会很短，大约两个到三个前缀。数组越大，链的期望长度越短，查询进行得也越快。实际上，这个程序还仍然是个摆设，因此其性能并不那么关键。另一方面，如果选用的数组太小，程序将无法在合理时间里处理完可能的输入。而如果它太大，又可能无法放进计算机的存储器中。

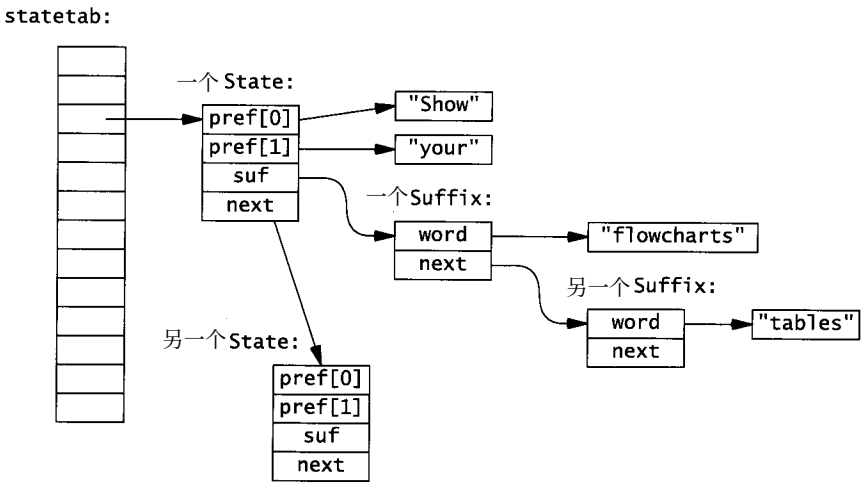
前缀可以用词的数组的方式存储。散列表的元素用 State(状态)数据类型表示，它是前缀与Suffix(后缀)链表的关联：

```
typedef struct State State;
typedef struct Suffix Suffix;
struct State {  /* prefix + suffix list */
    char    *pref[NPREF];   /* prefix words */
    Suffix  *suf;           /* list of suffixes */
    State   *next;          /* next in hash table */
};
struct Suffix { /* list of suffixes */
    char    *word;          /* suffix */
    Suffix  *next;          /* next in list of suffixes */
};

State   *statetab[NHASH];   /* hash table of states */
```

现在看一看图示，整个数据结构将具有下面的样子：

China-pub.com 下载



我们需要一个作用于前缀的散列函数，前缀的形式是字符串数组，显然不难对第 2章的字符串散列函数做一点修改，使之可用于字符串的数组。下面的函数对数组里所有字符串的拼接做散列：

```
/* hash: compute hash value for array of NPREF strings */
unsigned int hash(char *s[NPREF])
{
    unsigned int h;
    unsigned char *p;
    int i;

    h = 0;
    for (i = 0; i < NPREF; i++)
        for (p = (unsigned char *) s[i]; *p != '\0'; p++)
            h = MULTIPLIER * h + *p;
    return h % NHASH;
}
```

再加上对检索函数的简单修改，散列表的实现就完成了：

```
/* lookup: search for prefix; create if requested. */
/*  returns pointer if present or created; NULL if not. */
/*  creation doesn't strdup so strings mustn't change later. */
State* lookup(char *prefix[NPREF], int create)
{
    int i, h;
    State *sp;

    h = hash(prefix);
    for (sp = statetab[h]; sp != NULL; sp = sp->next) {
        for (i = 0; i < NPREF; i++)
            if (strcmp(prefix[i], sp->pref[i]) != 0)
                break;
        if (i == NPREF)     /* found it */
            return sp;
    }
    if (create) {
        sp = (State *) emalloc(sizeof(State));
        for (i = 0; i < NPREF; i++)
            sp->pref[i] = prefix[i];
        sp->suf = NULL;
        sp->next = statetab[h];
```

China-pub.com
下载


```
            statetab[h] = sp;
        }
        return sp;
    }
```

注意，lookup在建立新状态时并不做输入字符串的拷贝。它只是向 sp->pref[]里存入一个指针。这实际上要求调用 lookup的程序必须保证这些数据不会被覆盖。例如，如果字符串原来存放在I/O缓冲区里，那么在调用 lookup前必须先做一个拷贝。否则后面的输入就会把散列表指针所指的数据覆盖掉。对于跨越某个界面的共享资源，常常需要确定它的拥有者到底是谁。在下一章里有对这个问题的详尽讨论。

作为下一步，我们需要在读入文件的同时构造散列表：

```
/* build: read input, build prefix table */
void build(char *prefix[NPREF], FILE *f)
{
    char buf[100], fmt[10];

    /* create a format string; %s could overflow buf */
    sprintf(fmt, "%%%ds", sizeof(buf)-1);
    while (fscanf(f, fmt, buf) != EOF)
        add(prefix, estrdup(buf));
}
```

对sprintf的调用有些奇怪，这完全是为了避免 fscanf的一个非常烦人的问题，而从其他方面看，使用fscanf都是很合适的。如果以%s作为格式符调用fscanf，那就是要求把文件里的下一个由空白界定的词读入缓冲区。但是，假如在这种情况下没有长度限制，特别长的词就可能导致输入缓冲区溢出，从而酿成大祸。假设缓冲区的长度为 100个字节(这远远超出正常文本中可能出现的词的长度)，我们可以用%99s(留一个字节给串的结束符‘\0’)，这是告诉fscanf读到99个字符就结束。这样做有可能把过长的词分成段，虽然是不幸的，但却是安全的。我们可以声明：

```
?   enum { BUFSIZE = 100 };
?   char    fmt[] = "%99s"; /* BUFSIZE-1 */
```

但是这里又出现了由一个带随意性的决定 (缓冲区大小)导出的两个常数，并要求维护它们之间的关系。这个问题可以一下子解决：只要利用 sprintf动态地建立格式串，也就是上面程序里采用的方式。

函数build有两个参数，一个是 prefix数组，用于保存前面的 NPREF个输入词；另一个是个FILE指针。函数把prefix和读入词的一个拷贝送给 add，该函数在散列表里加入一个新项，并更新前缀数组：

```
/* add: add word to suffix list, update prefix */
void add(char *prefix[NPREF], char *suffix)
{
    State *sp;

    sp = lookup(prefix, 1);  /* create if not found */
    addsuffix(sp, suffix);
    /* move the words down the prefix */
    memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
    prefix[NPREF-1] = suffix;
}
```

对memmove的调用是在数组里做删除的一个惯用法。该操作把前缀数组里从 1到NPREF-1的

China-pub.com 下载

元素向下搬，移到从0到NPREF-2的位置。这也就删去了第一个前缀词，并为新来的一个在后面腾出了位置。

函数addsuffix把一个新后缀加进去：

```
/* addsuffix: add to state. suffix must not change later */
void addsuffix(State *sp, char *suffix)
{
    Suffix *suf;

    suf = (Suffix *) emalloc(sizeof(Suffix));
    suf->word = suffix;
    suf->next = sp->suf;
    sp->suf = suf;
}
```

这里的状态更新操作分由两个函数实现： add完成给有关前缀加入一个后缀的一般性工作，addsuffix做的是由特定实现方式决定的动作，把一个词具体地加进后缀链表里。函数 add由build调用，而addsuffix只在add内部使用，因为这里牵涉到的是一个实现细节，这个细节今后也可能会变化。所以，虽然该操作只在这一个地方用，最好也将它建立为一个独立的函数。

3.4 生成输出

数据结构构造好之后，下一步就是产生输出。这里的基本思想与前面类似：给定一个前缀，随机地选出它的某个后缀，打印输出并更新前缀。当然，这里说的是处理过程的稳定状态，还需要弄清算法应该如何开始和结束。如果我们已经记录了文中第一个前缀，操作就非常简单了：直接从它们起头。结束也容易。我们需要一个标志字来结束算法，在所有正常输入完成之后，我们可以加进一个结束符，一个保证不会在任何输入里出现的“词”：

```
build(prefix, stdin);
add(prefix, NONWORD);
```

NONWORD应该是某个不可能在正规输入里遇到的值。由于输入词是由空白界定的，一个空白的“词”总能扮演这个角色，比如用一个换行符号：

```
char NONWORD[] = "\n";  /* cannot appear as real word */
```

还有一件事也需要考虑：如果输入根本就不够启动程序，那么又该怎么办呢？处理这类问题有两种常见方式：或是在遇到输入不足时立即退出执行；或是通过安排使得输入总是足够的，从而就完全不必再理会这个问题了。对这里的程序而言，采用后一种方式能够做得很好。

我们可以用一个伪造的前缀来初始化数据结构构造和输出生成过程，这样就能保证程序的输入总是足够的。在做循环准备时，我们把前缀数组装满 NONWORD词。这样做有一个非常好的效果：输入文件里的第一个词将总是这个伪造前缀的第一个后缀[⊖]。这样，生成循环要打印的全都是它自己生成的后缀。

如果输出非常长，我们可以在产生了一定数目的词之后终止程序；另一种情况是程序遇到了后缀NONWORD。最终看哪个情况先出现。

在数据的最后加上几个 NONWORD，可以大大简化程序的主处理循环。这是一种常用技术的又一个实例：给数据加上哨卫，用以标记数据的界限。

⊖ 实际上也是惟一的一个。——译者

China-pub.com
下载

作为编程的一个规则，我们总应该设法处理数据中各种可能的非正常情况、意外情况和特殊情况。编出正确代码很不容易，因此应该尽量使控制流简单和规范。

函数generate采用的就是前面已经给出了轮廓的算法。它产生每行一个词的输出，用文字处理程序可以把它们汇成长的行。第 9章将给出一个能完成这个工作的简单格式化程序fmt。

借助于数据开始和结束的NONWORD串，generate也很容易开始和结束：

```
/* generate: produce output, one word per line */
void generate(int nwords)
{
    State *sp;
    Suffix *suf;
    char *prefix[NPREF], *w;
    int i, nmatch;

    for (i = 0; i < NPREF; i++) /* reset initial prefix */
        prefix[i] = NONWORD;

    for (i = 0; i < nwords; i++) {
        sp = lookup(prefix, 0);
        nmatch = 0;
        for (suf = sp->suf; suf != NULL; suf = suf->next)
            if (rand() % ++nmatch == 0) /* prob = 1/nmatch */
                w = suf->word;
        if (strcmp(w, NONWORD) == 0)
            break;
        printf("%s\n", w);
        memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
        prefix[NPREF-1] = w;
    }
}
```

注意，算法需要在不知道到底有多少个项的情况下随机地选取一个项。变量 nmatch用于在扫描后缀表的过程中记录匹配的个数。表达式：

```
rand() % ++nmatch == 0
```

增加nmatch的值，而且使它为真的概率总是 1/nmatch。这样，第一个匹配的项被选中的概率为1，第二个将有1/2的概率取代它，第3个将以1/3的概率取代前面选择后留下的项，依此类推。在任何时刻，前面匹配的k个项中的每一个都有$1/k$的被选概率。

在算法开始时，prefix已经被设为初始值，可以保证散列表中一定有它。这样得到的第一个suffix值就是文件里的第一个词，因为它是跟随初始前缀的惟一后缀。在此之后，算法随机地选择后缀：循环中调用 lookup，查出当前prefix对应的散列表项，然后随机地选出一个对应后缀，打印它并更新前缀。

如果选出的后缀是NONWORD，则工作完成，因为已经选到了对应输入结束的状态。如果后缀不是NONWORD，则打印它，然后调用memmove丢掉前缀的第一个词，把选出的后缀升格为前缀的最后一个词，并继续循环下去。

现在，我们可以把所有东西放到一起，装进一个 main函数里，它从标准输入流读入，生成至多有指定个数的词序列：

```
/* markov main: markov-chain random text generation */
int main(void)
{
```

China-pub.com 下载

```
    int i, nwords = MAXGEN;
    char *prefix[NPREF];        /* current input prefix */
    for (i = 0; i < NPREF; i++) /* set up initial prefix */
        prefix[i] = NONWORD;
    build(prefix, stdin);
    add(prefix, NONWORD);
    generate(nwords);
    return 0;
}
```

这就完成了我们的C实现。本章最后对不同语言的实现做比较时我们还要回到这里。 C语言最强的地方就是给了程序员对实现方式的完全控制，用它写出的程序趋向于快速。但是这也有代价，这就是C程序员必须做更多的工作，包括分配和释放存储，建立散列表和链接表，以及其他许多类似的事项。C语言像一把飞快的剃刀，使用它，你可以创造优雅和高效的程序，或者弄出些一塌糊涂的东西来。

练习3-1 算法从未知长度的链表里随机地选择项，这依赖于一个好的随机数生成器。设计并执行一些试验，确定这个方法在实践中到底工作得怎么样。

练习3-2 如果把所有输入词都存入另一个散列表，每个词只保存一个，那么可以节约许多存储。对一些文本做测试，估计可以节省多少。采用这种组织方式，我们对前缀的散列链表可以用比较指针而不是比较字符串的方式，这应该运行得更快。请实现这种方式，估计速度和存储消耗的改变情况。

练习3-3 去掉在数据开头和结束设置的作为哨卫的NONWORD的语句，修改generate，使程序在没有它们的情况下还能正常开始和结束。请确认程序能对 0、1、2、3和4个词的输入正确工作。将这个实现方式和使用哨卫的实现方式做些比较。

3.5 Java

我们在Java里做第二个马尔可夫链算法的实现。 Java是面向对象的语言，这种语言将促使人们对程序里各组件之间的界面给予特别关注。在这里，具有独立性的数据项和与之相关的函数(称为方法)被封装一起，形成称为对象或类的程序部件。

Java的库比C语言丰富得多，其中包含了一组能以各种方式把已有数据汇集起来的容器类。Vector是容器类的一个例子，它提供一种动态增长的数组，可以存储任何Object类型的东西。另一个例子是Hashtable类，它允许以任何类型的对象作为关键码，存储或提取某种类型的值。

在这个应用里，字符串的Vector是保存前缀和后缀的自然选择。用一个散列表，以前缀向量作为关键码，后缀向量作为值。按照术语，这种结构是从前缀到后缀的一个映射。在Java语言里不需要显式定义的State类型，因为Hashtable结构能将前缀隐含地连接(映射)到后缀。这种设计与前面C版本的情况不同，那里建立了State结构，其中包含前缀和后缀链表，通过对前缀的散列计算得到的是一个完整的State。

Hashtable类提供了一个put方法，用于存储关键码和值的对；另外还提供了一个 get方法，通过它可以从关键码出发得到对应的值：

```
Hashtable h = new Hashtable();
h.put(key, value);
Sometype v = (Sometype) h.get(key);
```

China-pub.com
下载

我们的实现总共使用了三个类。第一个类Prefix保存前缀向量的词：

```
class Prefix {
    public Vector pref; // NPREF adjacent words from input
    ...
```

第二个类是Chain，它读取输入、构造散列表并产生输出。下面是它的类定义：

```
class Chain {
    static final int NPREF = 2; // size of prefix
    static final String NONWORD = "\n";
                    // "word" that can't appear
    Hashtable statetab = new Hashtable();
                    // key = Prefix, value = suffix Vector
    Prefix prefix = new Prefix(NPREF, NONWORD);
                    // initial prefix
    Random rand = new Random();
    ...
```

第三个类是公共界面，其中包含一个main函数和一个Chain实例：

```
class Markov {
    static final int MAXGEN = 10000; // maximum words generated
    public static void main(String[] args) throws IOException
    {
        Chain chain = new Chain();
        int nwords = MAXGEN;

        chain.build(System.in);
        chain.generate(nwords);
    }
}
```

Chain类实例在创建时构造起一个散列表，其中的前缀数组用 NPREF个NONWORD设置了初始值。函数build利用库函数StreamTokenizer进行输入剖析，将输入分解为空白符分隔的一系列单词。进入循环前有三个函数调用，这是为了对完成单词分解的库函数做一些设置，使它的工作方式能符合我们关于“词”的定义。

```
// Chain build: build State table from input stream
void build(InputStream in) throws IOException
{
    StreamTokenizer st = new StreamTokenizer(in);

    st.resetSyntax();                       // remove default rules
    st.wordChars(0, Character.MAX_VALUE); // turn on all chars
    st.whitespaceChars(0, ' ');             // except up to blank
    while (st.nextToken() != st.TT_EOF)
        add(st.sval);
    add(NONWORD);
}
```

函数add根据给定的前缀参数从散列表里提取对应的后缀向量，找不到 (向量为空)时就创建一个新向量，并把这个新向量和新前缀一起存入散列表。在任何情况下，它都向后缀向量里加入一个新词，然后更新前缀，丢掉第一个词并在最后加一个新词。

```
// Chain add: add word to suffix list, update prefix
void add(String word)
{
    Vector suf = (Vector) statetab.get(prefix);
    if (suf == null) {
        suf = new Vector();
```

China-pub.com 下载

```
            statetab.put(new Prefix(prefix), suf);
        }
        suf.addElement(word);
        prefix.pref.removeElementAt(0);
        prefix.pref.addElement(word);
    }
```

请注意，如果suf为空，add将在散列表里建立一个新的Prefix，而不是直接把prefix本身存进去。这里必须采用这种做法，因为Hashtable类是以引用方式存储数据项的，如果不明确地建立新拷贝，随后的操作就会把已经存入表里的数据覆盖掉。在前面写 C程序时，我们也特别注意到对这个问题的处理。

生成函数与C程序里对应的函数差不多，只是稍微紧凑一些。在这里可以通过下标随机地直接访问向量元素，不必写一个循环去扫描整个链表。

```
// Chain generate: generate output words
void generate(int nwords)
{
    prefix = new Prefix(NPREF, NONWORD);
    for (int i = 0; i < nwords; i++) {
        Vector s = (Vector) statetab.get(prefix);
        int r = Math.abs(rand.nextInt()) % s.size();
        String suf = (String) s.elementAt(r);
        if (suf.equals(NONWORD))
            break;
        System.out.println(suf);
        prefix.pref.removeElementAt(0);
        prefix.pref.addElement(suf);
    }
}
```

Prefix有两个建构函数，它们根据由参数提供的数据创建新实例。第一个建构函数复制已有的Prefix，第二个函数建立一个新前缀，其中存放某字符串的 n个拷贝。在程序初始化时，我们要用第二个建构函数建立包含NPREF个NONWORD的前缀：

```
// Prefix constructor: duplicate existing prefix
Prefix(Prefix p)
{
    pref = (Vector) p.pref.clone();
}

// Prefix constructor: n copies of str
Prefix(int n, String str)
{
    pref = new Vector();
    for (int i = 0; i < n; i++)
        pref.addElement(str);
}
```

Prefix类里也有两个方法，hashCode和equals，它们在Hashtable的实现中隐含地被调用，产生下标和对表进行检索。Hashtable类要求必须为这两个方法专门建立一个类，这迫使我们把Prefix建成一个完整的新类，而不是像后缀那样只建立一个 Vector。

hashCode对向量各元素使用hashCode方法，将得到的值组合成一个散列值：

```
static final int MULTIPLIER = 31;    // for hashCode()

// Prefix hashCode: generate hash from all prefix words
public int hashCode()
```

China-pub.com
下载


```
{
    int h = 0;
    for (int i = 0; i < pref.size(); i++)
        h = MULTIPLIER * h + pref.elementAt(i).hashCode();
    return h;
}
```

函数equals对两个前缀做比较，采用逐个比较元素的方式。

```
// Prefix equals: compare two prefixes for equal words
public boolean equals(Object o)
{
    Prefix p = (Prefix) o;
    for (int i = 0; i < pref.size(); i++)
        if (!pref.elementAt(i).equals(p.pref.elementAt(i)))
            return false;
    return true;
}
```

这个Java程序比前面的C程序小了不少，并照顾到更多的细节，Vector和Hashtable是其中最明显的例子。一般地说，这里的存储管理比较简单，因为向量本身能够自动增长，废料收集将管理回收那些不再引用的存储。但是，为了能使用 Hashtable类，我们还必须自己写函数hashCode和equals。可见Java并没有照顾好一切细节。

对C和Java程序里针对相同基本数据结构的表示和操作做一个比较，可以看到，在 Java程序里的功能划分做得更好。例如，在这里想把 Vector换成数组是非常简单的。在C程序里就不同，每个东西都知道其他东西在做什么。例如：散列表在数组上进行操作，因此，在许多地方它都要做数组的维护工作；lookup知道State和Suffix的内部结构；任何东西都知道前缀数组的大小。

```
% java Markov <jr_chemistry.txt | fmt
Wash the blackboard.  Watch it dry.  The water goes
into the air.  When water goes into the air it
evaporates.  Tie a damp cloth to one end of a solid or
liquid.  Look around.  What are the solid things?
Chemical changes take place when something burns.  If
the burning material has liquids, they are stable and
the sponge rise.  It looked like dough, but it is
burning.  Break up the lump of sugar into small pieces
and put them together again in the bottom of a liquid.
```

练习3-4 重写Java的markov程序，用数组(而不是Vector)表示Prefix类里的前缀。

3.6 C++

我们的第三个实现将在C++ 中完成。因为C++ 语言几乎是C的一个超集，它也能以C的形式使用，只要注意某些写法规则。实际上，前面 C版本的markov也是一个完全合法的C++ 程序。对C++而言，更合适的用法应该是定义一些类，建立起程序中需要的各种对象，或多或少像我们写Java程序时所做的那样，这样可以隐蔽起许多实现细节。我们在这里希望更前进一步，使用C++的Standard Template Library(标准模板库)，即STL。因为STL提供了许多内部机制，能完成我们需要做的许多事情。ISO的C++ 标准已经把STL作为语言定义的一部分。

STL提供了许多容器类，例如向量、链表、集合等，还包括了许多检索、排序、插入和删

China-pub.com 下载

除的基本算法。通过利用 C++ 的模板特性，每个 STL 算法都能用到很多不同的容器类上，容器类的元素可以是用户定义类型的或者是内部类型的 (如整数)。这里的容器都被描述为 C++ 模板，可以对特定类型进行实例化。例如， STL里有一个vector容器类，由它可以导出各种具体类型，比如vector<int>或vector<string>等。所有的vector操作，包括排序的标准算法等，都可以直接用于这些数据类型。

在STL里，除了有vector容器(它与Java的vector类差不多)，还提供了一个deque容器类。deque(念为deck)是一种双端队列，它正好能符合我们对前缀操作的需要：可以用它存放NPREF个元素，丢掉最前面的元素并在后面添一个新的，这都是 $O(1)$操作。实际上， STL的deque比我们需要的东西更一般，它允许在两端进行压入和弹出，而执行性能方面的保证是我们选择它的原因。

STL还提供了一个map容器，其内部实现基于平衡的树。在map中可以存储关键码—值对。map的实现方式保证，从任何关键码出发提取相关值的操作都是 $O(\log n)$。虽然map可能不如 $O(1)$的散列表效率高，但是，直接使用它就可以不必写任何代码，这样也很不错 (某些非标准的C++库提供了hash或hash_map容器，它们的性能可能更好些)。

我们也可以使用内部提供的比较函数，用于对前缀中各字符串做比较。

手头有了这些组件，有关代码可以流畅地做出来了。下面是有关声明：

```
typedef deque<string> Prefix;
map<Prefix, vector<string> > statetab; // prefix -> suffixes
```

STL提供了deque的模板，记法deque<string>将它指定为以字符串为元素的 deque。由于这个类型将在程序里多次出现，在这里用一个 typedef声明，将它另外命名为 Prefix。映射类型中将存储前缀和后缀，因为它在程序里只出现一次，我们就没有给它命名。这里声明了一个map类型的变量statetab，它是从前缀到后缀向量的映射。在这里工作比在 C或Java中更方便，根本不需要提供散列函数或者 equal方法。

main函数对前缀做初始化，读输入 (对于标准输入，调用 C++ iostream里的cin)，在输入最后加一个尾巴，然后产生输出。和前面各个版本完全一样。

```
// markov main: markov-chain random text generation
int main(void)
{
    int nwords = MAXGEN;
    Prefix prefix;                  // current input prefix

    for (int i = 0; i < NPREF; i++) // set up initial prefix
        add(prefix, NONWORD);
    build(prefix, cin);
    add(prefix, NONWORD);
    generate(nwords);
    return 0;
}
```

函数build使用iostream库，一次读入一个词：

```
// build: read input words, build state table
void build(Prefix& prefix, istream& in)
{
    string buf;

    while (in >> buf)
```

China-pub.com
下载


```
            add(prefix, buf);
    }
```

字符串buf能够根据输入词的长度需要自动增长。

函数add能够进一步说明使用STL 的优越性：

```
// add: add word to suffix list, update prefix
void add(Prefix& prefix, const string& s)
{
    if (prefix.size() == NPREF) {
        statetab[prefix].push_back(s);
        prefix.pop_front();
    }
    prefix.push_back(s);
}
```

这几个非常简单的语句确实做了不少事情。 map容器类重载了下标运算符 ([]运算符)，使它在这里成为一种查询运算。表达式 statetab[prefix]在statetab里完成一次查询，以prefix作为查询的关键码，返回对于所找到的项的一个引用。如果对应的向量不存在，这个操作将建立一个新向量。 vector和deque类的push_back函数分别把一个新字符串加到向量或deque的最后；pop_front从deque里弹出头一个元素。

输出生成与前面的版本类似：

```
// generate: produce output, one word per line
void generate(int nwords)
{
    Prefix prefix;
    int i;

    for (i = 0; i < NPREF; i++) // reset initial prefix
        add(prefix, NONWORD);

    for (i = 0; i < nwords; i++) {
        vector<string>& suf = statetab[prefix];
        const string& w = suf[rand() % suf.size()];
        if (w == NONWORD)
            break;
        cout << w << "\n";
        prefix.pop_front();     // advance
        prefix.push_back(w);
    }
}
```

总的来说，这个版本看起来特别清楚和优雅 —— 代码很紧凑，数据结构清晰，算法完全一目了然。可惜的是，在这里也要付出一些代价：这个版本比原来的 C版本慢得多，虽然它还不是最慢的。不久我们将回头来讨论性能测试问题。

练习3-5 STL的强项是很容易在其中做不同数据结构的试验。修改这里C++ 版本的马尔可夫程序，用不同的结构表示前缀、后缀表以及状态表。对于不同结构，程序的执行性能有什么变化？

练习3-6 另写一个C++ 程序，只使用类和string数据类型，不使用其他高级的库功能。从风格和速度方面将它与采用 STL的版本做各种比较。

3.7 Awk和Perl

为使这个演习比较圆满，我们也用两个应用广泛的脚本语言 (Awk和Perl)写了这个程序。

China-pub.com 下载

这些语言都提供了本应用所需要的各种特征，包括关联数组和字符串处理等。

关联数组实际上是散列表的一种包装，使用起来非常方便。它看起来像数组，但可以用任意的字符串或数，或者字符串和数的由逗号分隔的表作为下标。关联数组也是一种从一个数据类型到另一类型的映射。在Awk里，所有数组都是关联数组；Perl则不同，这里既有以整数作为下标的普通数组，也有关联数组——称为“散列”，这实际上也说明了它们的实现方法。

在程序的Awk和Perl实现里，前缀长度都固定为2。

```
# markov.awk: markov chain algorithm for 2-word prefixes
BEGIN { MAXGEN = 10000; NONWORD = "\n"; w1 = w2 = NONWORD }
{   for (i = 1; i <= NF; i++) {     # read all words
        statetab[w1,w2,++nsuffix[w1,w2]] = $i
        w1 = w2
        w2 = $i
    }
}
END {
    statetab[w1,w2,++nsuffix[w1,w2]] = NONWORD  # add tail
    w1 = w2 = NONWORD
    for (i = 0; i < MAXGEN; i++) {  # generate
        r = int(rand()*nsuffix[w1,w2]) + 1  # nsuffix >= 1
        p = statetab[w1,w2,r]
        if (p == NONWORD)
            exit
        print p
        w1 = w2         # advance chain
        w2 = p
    }
}
```

Awk是一个模式—操作对的语言：输入总以一次一行的方式读入，每个读入行都拿来与程序里的模式做匹配，与此同时，对各个成功匹配执行有关动作。这里存在着两个特殊的模式，BEGIN和END，它们分别能在输入的第一行之前和最后一行之后匹配成功。

动作是由花括号括起的一个语句块。在上面的Awk版本的马尔可夫程序里，BEGIN动作块对前缀和若干其他变量做初始化。

随后的一个语句块没有模式部分，这是一种默认方式，意味着这个块将对每个输入行执行一次。Awk自动把每个读入的行分割成一些域(由空白分隔的词)，它们将分别成为$1到$NF。变量NF的值是域的个数。语句：

```
statetab[w1,w2,++nsuffix[w1,w2]] = $i
```

建立从前缀到后缀的映射。数组nsuffix记录后缀个数，其元素nsuffix[w1,w2]记录与前缀对应的后缀的个数。而后缀本身则被做为数组statetab的元素，如statetab[w1, w2, 1]，statetab[w1, w2, ,2算等。

当END块执行时，所有内容都已经输入完毕。到这个时刻，对于每个前缀，nsuffix里都有一个元素记录着它对应的后缀个数，而在statetab里则存有相应个数的后缀。

用Perl语言写出的东西与此类似，但是需要另外用一个匿名数组(而不是用第三个下标变量)来保存后缀的有关情况。这里还需要用一个多重赋值完成对前缀的更新。Perl采用特殊字符表示变量的类型，$表示标量，@表示有下标的数组。这里的[ ]用于下标数组，而{}表示散列。

China-pub.com
下载


```
# markov.pl: markov chain algorithm for 2-word prefixes

$MAXGEN = 10000;
$NONWORD = "\n";
$w1 = $w2 = $NONWORD;              # initial state
while (<>) {                       # read each line of input
    foreach (split) {
        push(@{$statetab{$w1}{$w2}}, $_);
        ($w1, $w2) = ($w2, $_); # multiple assignment
    }
}
push(@{$statetab{$w1}{$w2}}, $NONWORD);     # add tail

$w1 = $w2 = $NONWORD;
for ($i = 0; $i < $MAXGEN; $i++) {
    $suf = $statetab{$w1}{$w2}; # array reference
    $r = int(rand @$suf);          # @$suf is number of elems
    exit if (($t = $suf->[$r]) eq $NONWORD);
    print "$t\n";
    ($w1, $w2) = ($w2, $t);        # advance chain
}
```

和前面程序一样，映射本身保存在变量statetab里。程序中最关键的行是：

```
push(@{$statetab{$w1}{$w2}}, $_);
```

它把一个新后缀加入到存储在statetab{$w1}{$w2}的匿名数组的最后。在随后的生成阶段里，$statetab{$w1}{$w2}是对后缀数组的一个引用，而$suf->[$r]指向其中的第r个后缀。

与前三个程序相比，用Awk和Perl写的程序都短得多，但要把它们修改为能处理前缀词不是两个的情况却很困难。采用C++ 的STL实现，其核心部分(函数add和generate)看起来长一点，但是却更清晰。无论如何，脚本语言对有些情况是很好的选择，例如做试验性的程序设计，构造系统原型，以及做那些对运行时间要求不高的实际产品。

练习3-7 修改上面的Awk和Perl程序，设法使它们能处理任意长度的前缀。通过试验确定这种修改对性能的影响。

3.8 性能

我们对上面的几个实现做了些比较。对程序做计时用的是《圣经·雅各书》中的《诗篇》，其中共有42 685个词(5 238个不同的词，22 482个前缀)。这个文本里有足够多的重复出现的短语(如“Blessed is the ...”)，其中有一个后缀包含多于400个元素，另外还有几百个链包含几十个后缀。所以这是个很好的测试数据集。

> Blessed is the man of the net. Turn thee unto me, and raise me up, that I may tell all my fears. They looked unto him, he heard. My praise shall be blessed. Wealth and riches shall be saved. Thou hast dealt well with thy hid treasure: they are cast into a standing water, the flint into a standing water, and dry ground into watersprings.

下面的表格列出的是生成10 000个输出词所用的秒数。这里用的机器一台是250 MHz的MIPS R10000，运行Irix 6.4系统，另一台是400 MHz的Pentium II，带有128 M内存，运行Windows NT系统。运行时间几乎完全由输入数据的规模决定，相对而言，输出生成是非常快的。在表格里，还以源程序行数的方式给出了程序的大致规模。

China-pub.com 下载

	250MHz R10000	400MHz Pentium II	源代码行数
C	0.36 sec	0.30 sec	150
Java	4.9	9.2	105
C++/STL/deque	2.6	11.2	70
C++/STL/list	1.7	1.5	70
Awk	2.2	2.1	20
Perl	1.8	1.0	18

C和C++ 程序都用带优化的编译器完成编译， Java程序运行时打开了即时编译功能。 Irix的C和C++编译是从三个不同编译器里选出的最快的一个，由 Sun SPARC和DEC Alpha机器得到的数据也差不多。C版本的程序是最快的，比别的程序都快得多。 Perl程序的速度次之。表格里的时间是我们试验的一个剪影，用的是特定的编译器和库。在你自己的环境里做，得到的结果也可能与此有很大差别。

Windows上的STL `deque`版本肯定存在什么问题。试验说明表示前缀的 `deque`用掉了大部分的运行时间，虽然在它里面从来都不超过两个元素。按说作为中心数据结构的映射所消耗的时间应该是最多的。把`deque`改成链表(在STL里是双链表)能使程序性质大大改善。另一方面，在`Irix`环境中把映射改为 (非标准的)`hash`容器则并没有产生多大影响。在我们的Windows机器上没有散列可以用。要完成上面说的这些修改，需要做的只是把 `deque`换成`list`，或者把`map`换成`hash`，然后重新做一下编译，这也是对 STL基本设计思想的有效性的一个认证。我们也认为， STL作为C++ 的一个新部分，仍然受到不成熟实现的损害。在使用STL的不同实现或使用不同数据结构时，导致的性能变化是不可预测的。Java也存在这个问题，那里的实现也变得很快。

对于测试一个有意产生大量随机输出的程序，实际上存在着一些具有挑战性的问题。我们怎么能知道它确实是能工作的？怎样知道它在所有情况下都能工作？在第 6章讨论测试时将提出一些建议，并描述如何测试马尔可夫程序。

3.9 经验教训

马尔可夫程序已经有了很长历史。其第一个版本是 Don P. Mitchell写的，后来由 Bruce Ellis做了些修改，它在80年代被大量应用到各种文献分析 (deconstructionist)活动中。这个程序一直潜伏在那里，直到我们想到把它用在一个大学课程中，作为讨论程序设计的一个例子。我们并不是简单地捡起已有的东西，而是用 C语言重新把它写出来，通过遇到的各种问题重新唤醒我们的记忆。而后我们又用几种不同语言重新写它，用各语言独特的习惯用法表述同样的基本想法。在这个课程之后，我们又多次重写这个程序，设法改进其清晰性和表现形式。

在这整个过程中，基本的设计并没有改变。最早的版本使用的也正是我们在这里给出的方式，虽然其中确实用了第二个散列表来表示各个词。如果我们还要重写它，基本情况大概也不会有多大变动。一个程序的设计根源于它的数据的形式。数据结构并没有定义所有的细节，但它们确实规定了整个解的基本构造。

有些数据结构选择造成的差异不太大，例如，是用链接表还是用可增长数组。有些实现方式比别的方式更具普遍性 —— 例如，很容易把Perl和Awk程序改造成使用一个词或三个词前缀的程序，但要想使这个选择能够参数化，就会遇到很多麻烦。面向对象的语言有它们的优

China-pub.com
下载

越之处，对C++或Java的实现做点小修改，就可能使它们的数据结构适合英语之外的其他文本，例如程序(在那里空格可能是重要的东西)，或者乐谱，甚至产生测试序列的鼠标点击和菜单选择。

当然，即使数据结构本身差不多，在程序的一般表现方面，在源代码的大小方面，在程序执行性能方面也可以存在很大差异。粗略地说，使用较高级的语言比更低级的语言写出的程序速度更慢，但这种说法只是定性的，把它随意推广也是不明智的。大型构件，如 C++的STL或脚本语言里的关联数组、字符串处理，能使代码更紧凑，开发时间也更短。当然这些东西不是没有代价的，但是，性能方面的损失对于大部分程序而言可能并不那么重要。比如马尔可夫这样的程序，它不过只运行几秒钟。

当系统内部提供的代码太多时，人们将无法知道程序在其表面之下到底做了什么。我们应该如何评价这种对控制和洞察力的丧失，这是更不清楚的事情。这也就是 STL版本中遇到的情况，它的性能无法预料，也没有很容易的办法去解决问题。我们使用过一个很不成熟的实现，必须对它做一些修正后才能够运行我们的程序。但是，很少有人能有资源和精力去弄清其中的问题并且修复它们。

目前存在着一种对软件的广泛的不断增长的关注：当程序库、界面和工具变得越来越复杂时，它们也变得更难以理解和控制了。当所有东西都正常运转时，功能丰富的程序设计环境可以是非常有生产效率的，但是如果它们出了毛病，那就没什么东西可以依靠了。如果问题牵涉到的是性能或者某些难于捉摸的逻辑错误时，我们很可能根本没有意识到有什么东西出了毛病。

在这个程序的设计和实现过程中，我们看到了许多东西，这些对于更大的程序是很有教益的。首先是选择简单算法和数据结构的重要性，应该选择那些能在合理时间内解决具有预期规模的问题的最简单的东西。如果有人已经把它写好并放在库里，那是最好了。我们的C++ 程序由此获益匪浅。

按照Brooks的建议，我们发现最好是从数据结构开始，在关于可以使用哪些算法的知识的指导下进行详细设计。当数据结构安置好后，代码就比较容易组织了。

要想先把一个程序完全设计好，然后再构造它，这是非常困难的。构造现实的程序总需要重复和试验。构造过程逼迫人们去把前面粗略做出的决定弄清楚。这正是我们在写这些程序中遇到的情况。这个程序经历了许多细节方面的变化。应该尽可能从简单的东西开始，根据获得的经验逐步发展。如果我们的目标只是为个人兴趣而写一个马尔可夫链算法程序，很可能就在Awk或者Perl里写一个，可能也会不像这里这样仔细地打磨它，而是随其自然。

做产品代码要花费的精力比做原型多得多。例如可以把这里给出的程序看作是产品代码(因为它们已经被仔细打磨过，并经过了彻底的测试)。产品质量要求我们付出的努力要比个人使用的程序高一两个数量级。

练习3-8 我们已经看到马尔可夫程序在许多语言里的版本，包括 Scheme、Tcl、Prolog、Python、Generic Java、ML以及Haskell。从其中每一个，都可以看到其特殊的挑战性和优越性。在你最喜欢的语言里实现这个程序，并考察它的一般风格和性能。

补充阅读

标准模板库在许多书籍里都有介绍，包括 Matthew Austern的《类属程序设计与 STL》

China-pub.com 下载

(Generic Programming and the STL，Addison-Wesley，1998)。对C++ 语言本身的参考文献当然是Bjarne Stroustrup的《C++程序设计语言》(C++ Programming Language第3版，Addison-Wesley，1997)。对于Java，我们参考了Ken Arnold和James Gosling的《Java程序设计语言》(The Java Programming Language第2版，Addison-Wesley，1998)，对Perl语言的最好描述是Larry Wall、Tom Christiansen和Randal Schwartz的《Perl程序设计》(Programming Perl 第2版，O'Reilly，1996)。

隐藏在设计模式后面的基本思想是：大部分程序所采用的不过是很少几种不同的设计结构，与此类似，实际上也只有不多的几种基本数据结构。说的远一点，这与我们在第 1章讨论过的编码习惯用法也是很相像的。这方面的经典参考文献是 Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides的《设计模式：可重用面向对象软件的要素》(Design Pattern: Elements of Reusable Object-Oriented Software，Addison-Wesley， 1995)。

Markov程序原来称为shaney，其传奇经历刊登在1989年6月号《科学美国人》杂志的“趣味计算”专栏里，该文重载于 A. K. Dewdney的《神奇的机器》(The Magic Machine，W. H. Freeman，1990)。