China-pub.com
下载

第7章 性 能

他过去所做的许诺，是非凡的；
而他今天的执行[⊖]，则什么也不是。

莎士比亚，《亨利八世》

很久以前，程序员花费了他们的主要精力，想方设法把程序的效率弄得高一点。因为在那个时候计算机非常慢，而且非常昂贵。今天的机器便宜多了也快多了，因此，对绝对效率的需求也就大大地减弱了。我们难道还值得去考虑执行性能吗？

当然需要。但是，只有在问题确实是非常重要，程序真正是非常慢，而且确有预期能在保持正确性、坚固性和清晰性的同时把程序弄得更快的情况下，才应该做这件事情。一个快速的程序如果给出的是错误结果，那么它一点也没有节省时间。

优化的第一要义是不做。程序是不是已经足够好了？应该了解程序将如何使用以及它将要运行于其中的环境，把它搞得更快有什么益处吗？为大学课程作业而写的程序不会再去使用，速度一般是没有关系的。对于大部分个人程序、偶尔用用的工具、测试框架、各种试验和原型程序而言，通常也都没有速度问题。而在另一方面，对一个商业产品或者其中的核心部件，例如一个图形库，性能可能就是非常关键的。因此，我们还是需要理解如何去考虑性能问题。

什么时候我们应该试图去加速一个程序？我们该如何做，又能够期望得到些什么呢？本章要讨论如何能使程序运行得更快些，或者少用些存储。速度通常是人们最关心的，因此也是我们讨论的主题。空间(主存、磁盘)出问题的情况少一点，但也可能是至关重要的，所以我们也要在这里花一些时间和空间(篇幅)。

正如我们在第2章看到的，最好的策略就是使用那些能适应工作需要的、最简单最清晰的算法和数据结构。然后再度量其性能，看是否需要做什么改变；打开编译系统的选择项，生成尽可能快的代码；考虑对程序本身做什么改变才能有最大作用；一次做一个改变并重新进行评价。在这期间始终保留最简单的版本，用它作为测试版本的对照物。

测量是改进性能过程中最关键的一环，推断和直觉都很容易受骗，所以在这里必须使用各种工具，如计时命令或轮廓文件等等。性能改进与程序测试有许多共同之处，包括许多技术，如自动化，认真保存记录，用回归测试以保证所做的改变维持了正确性，而且没有损害以前的改进等。

如果算法选择是明智的，程序也写得很仔细，那么你就可能发现完全没有进一步加速的必要性。对于写得很好的代码，很小的改变往往就能解决它的性能问题，而对那些设计拙劣的代码，经常会需要大范围地重写。

7.1 瓶颈

让我们从消除一个瓶颈的事例开始，事情出在我们局部环境里的一个关键程序上。

⊖ 这里的原文是performence，在本章题目和正文中均译为“性能”或“执行性能”。——译者

China-pub.com
下载

外来的电子邮件汇集之后通过一台称为网关的机器，这台机器将内部网络与外部的Internet连接起来。外来的电子邮件消息——对于一个有数千人的团体，每天是数以万计的。这些邮件到达网关并传送到内部网。这个分隔把我们的私有网络和公共的 Internet隔离开，并使这个团体所有的人只需要公开一台机器的名字 (网关的名字)。

网关有一项工作是过滤掉所有的“垃圾邮件” (spam)，即那些不请自来的邮件，它们通常宣扬一些没什么价值的服务。通过早期的成功试验之后，这个垃圾邮件过滤程序被安装好，作为提供给网关用户的一项固定服务。这之后，一个问题就出现了。该网关机器原来已经很陈旧，已经非常忙，现在则完全被压垮了。造成这种情况的原因是过滤器程序占用了太多的机器时间——比对消息的所有其他处理所需时间加在一起还要多得多，这使消息队列被填满，在系统的哽噎挣扎之中，消息发送被成小时地拖延。

这是性能真有问题的一个例子：程序对于完成指定工作而言太慢，而人则由于其工作拖延感到非常不舒服。这种程序确实必须大大地加快运行速度。

对问题稍做简化，这个垃圾邮件过滤程序的工作方式大致是这样：每个外来消息被当作一个字符串看待，一个模式匹配程序检查这种串，看它是否包含了某些已知垃圾邮件中的短语，如“Make millions in your spare time”或“XXX-rated”等。这种消息总是翻来覆去的，所以这个技术应该很有效。如果发现一个垃圾邮件没被捉住，只要向表里加入一个短语，程序在下次就能逮住它。

系统里现成的串匹配工具，如grep等，都不能同时具有合适的执行性能和包装方式，因此我们只能设法写一个特殊的垃圾邮件过滤程序。原始代码非常简单，它检查一个消息里是否包含任一指定的短语(模式)：

```
/* isspam: test mesg for occurrence of any pat */
int isspam(char *mesg)
{
    int i;

    for (i = 0; i < npat; i++)
        if (strstr(mesg, pat[i]) != NULL) {
            printf("spam: match for '%s'\n", pat[i]);
            return 1;
        }
    return 0;
}
```

这种东西还怎么可能弄得更快呢？字符串必须被检索，而来自 C标准库的strstr函数是做这种检索的最好方式：它是标准的、高效的。

通过使用轮廓文件技术(将在下一节里讨论)，我们弄清问题就出在strstr的实现方式上。在用于垃圾邮件过滤程序时，它具有一些很不幸的性质。通过改变 strstr的工作方式，可以使它的效率大大提高(这是针对这个特定问题而言)。

函数strstr的现有实现大致是下面的样子：

```
/* simple strstr: use strchr to look for first character */
char *strstr(const char *s1, const char *s2)
{
    int n;

    n = strlen(s2);
    for (;;) {
        s1 = strchr(s1, s2[0]);
```

China-pub.com 下载

```
            if (s1 == NULL)
                return NULL;
            if (strncmp(s1, s2, n) == 0)
                return (char *) s1;
            s1++;
        }
    }
```

这个写法已经考虑了效率问题。实际上，在典型应用中它也是非常快的，因为工作中用的是经过高度优化的库代码。函数调用strchr去发现模式中第一个字符在串里的下一个出现位置，然后调用strcmp查看串的后面部分能否与模式的剩余字符匹配。这样，在寻找模式首字符的过程中，strstr能迅速地跳过消息的绝大部分，而后又能对其余部分做快速检查。这怎么会产生很糟糕的性能呢？

确实有许多原因：首先，strncmp带有一个模式长度参数，它必须通过strlen计算出来。由于执行中的模式是固定的，因此不应该对各个消息都重新计算模式的长度。

第二，strncmp用到一个复杂的内部循环。它不只是对两个串里的各个字节做比较，在对长度值向下计数的同时还必须关注两个串的结束\0字节。由于所有字符串的长度都为已知的(虽然strncmp并不知道)，这种复杂性完全是不必要的。知道计数值就足够了，检查\0纯粹是浪费时间。

第三，strchr本身也很复杂，因为它必须在寻找字符的同时也关注作为消息结束标志的\0字节。在对isspam的每次调用中，消息都是固定不变的，所以在寻找\0上花的时间完全是白费功夫，因为我们知道消息到那儿结束。

最后，虽然strstr、strchr和strncmp各自独立地看效率都很高，但反复调用这些函数的代价，与它们执行的计算工作相比，也非常可观。完成所有这些工作，一个更有效的方法是用一个特殊的、仔细写出的strstr版本，完全避免对其他函数的调用。

这些问题都是性能方面出麻烦的常见原因——某个例程或者界面对于各种典型情况都工作得非常好，但是对某些特殊情况却很糟糕，特别是如果这些情况又恰好出现在程序所处理工作的中心位置时。现有的strstr在模式和字符串都比较短、每次调用的处理对象又都不同的情况下，都能工作得很好；对于那些长而固定的字符串，它的开销就太高了。

有了这些考虑后，我们重写了strstr，同时处理模式和消息串，在其中寻找匹配，不调用任何子程序。这个实现确实具有我们所期望的行为：在某些情况下它稍微慢一点，但是做垃圾邮件过滤器就快得多，更重要的是它不再出现极坏的情况。为了验证新实现的正确性和执行性能，我们构造了一个性能测试集。这个测试集里不仅包括简单的例子，例如在一个句子里查找一个词；还包括一些病态情况，如在有一千个e的串里查找只有单个字母x的模式，在一个字母e的串里查找有一千个x的模式等。简单实现对这两种情况的处理都非常糟糕。这种极端性例子也是性能评价的关键。

用新的strstr更新了函数库之后，垃圾邮件过滤程序的运行速度提高了大约30%。对于只是重写一个子程序而言，这已是很好的回报了。

不幸的是，现在的程序仍然太慢。

在需要解决问题的时候，最重要的是给出正确的提问。到目前为止，我们提出的问题一直是，寻找在一个字符串里查找一个正文模式的最快方法。而实际问题却是要在一种很长的变动的串里，查找一个很大的固定的正文模式集合。对于这种提问，strstr是不是正确的解

China-pub.com
下载

决办法，这件事并不是一目了然的。

要使程序运行得快，最有效的方法就是使用更好的算法。对问题有了更清楚的看法之后，现在是认真想一想什么算法有可能工作得更好的时候了。

基本循环：

```
for (i = 0; i < npat; i++)
    if (strstr(mesg, pat[i]) != NULL)
        return 1;
```

对消息做npat次独立的扫描。即使它没有发现任何匹配，也需要对消息串里的每个字节做npat次检查，总共要做strlen(mesg)*npat次比较。

一种更好的方法是把循环翻转过来，在外层循环中只对消息串扫描一次，其间在内层循环里并行地查找各个模式：

```
for (j = 0; mesg[j] != '\0'; j++)
    if (some pattern matches starting at mesg[j])
        return 1;
```

通过简单观察，就可以看出性能改善的可能性。要知道是否有模式能在位置 j与消息串匹配，我们并不需要检查所有的模式，只要检查那些开始字符与 mesg[j]相同的模式就足够了。粗略估计，由于有52个大写和小写字母，我们只需要做大约strlen(mesg)*npat/52次比较。由于字母的分布不均匀 —— 例如，以s开头的词远多于以x开头的 —— 我们不能期望性能会提高52倍，但总应该提高一些。为此我们构造了一个散列表，用模式串的第一个字母作为关键字。

通过预先计算，构造出一个表，其中的项是以各个字符开始的模式，此后的 isspam仍然非常短：

```
int patlen[NPAT];                   /* length of pattern */
int starting[UCHAR_MAX+1][NSTART]; /* pats starting with char */
int nstarting[UCHAR_MAX+1];         /* number of such patterns */
...
/* isspam: test mesg for occurrence of any pat */
int isspam(char *mesg)
{
    int i, j, k;
    unsigned char c;

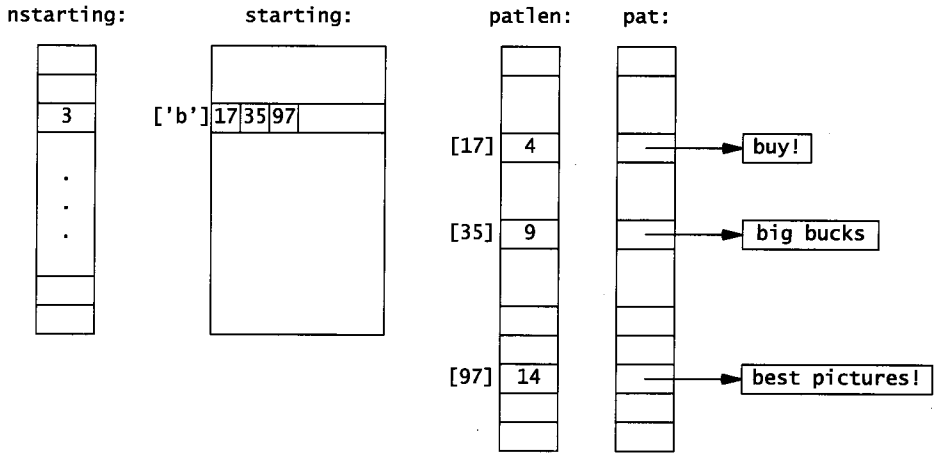
    for (j = 0; (c = mesg[j]) != '\0'; j++) {
        for (i = 0; i < nstarting[c]; i++) {
            k = starting[c][i];
            if (memcmp(mesg+j, pat[k], patlen[k]) == 0) {
                printf("spam: match for '%s'\n", pat[k]);
                return 1;
            }
        }
    }
    return 0;
}
```

这里用了一个两维数组starting[c][]。对每个字符c，在数组starting[c]里存储着以c开头的所有模式的指标。在与之相关的另一个数组 nstarting[c]里，记录着以c开头的模式的个数。如果没有这些表格，内部循环就必须从 0运行到npat，大概需要上千次。现在它只要从0循环到大约20。最后，在数组patlen[k]里存储着预先算出的strlen (pat[k])

China-pub.com 下载

的值。

下面的图中显示了这些数据结构的样子，其中只画出由字母 b开头的三个模式：



构造这些表格的代码很简单：

```
int i;
unsigned char c;

for (i = 0; i < npat; i++) {
    c = pat[i][0];
    if (nstarting[c] >= NSTART)
        eprintf("too many patterns (>=%d) begin '%c'",
            NSTART, c);
    starting[c][nstarting[c]++] = i;
    patlen[i] = strlen(pat[i]);
}
```

根据输入不同，这个垃圾邮件过滤程序比采用改进的 strstr程序快5~10倍，比初始程序快7~15倍。我们没有得到52倍的改进，其中一个原因是字母分布的不一致性，还由于新程序的循环比原来复杂许多。此外，在这里仍要做许多失败的串比较。但是垃圾邮件过滤程序已经不再是电子邮件发送的瓶颈，性能问题已经解决了。

本章的后面部分将详细讨论用于发现性能问题、隔离出过慢的代码并使其加速的各种技术。在讨论这些内容之前，重新审视垃圾邮件过滤程序，看看它给我们上了一堂什么课，也是非常有益的。在这里最重要的，就是必须弄清性能是不是一个实实在在的问题。如果垃圾邮件过滤程序不是瓶颈，那么所有努力就都毫无价值了。一旦弄清楚性能确实是问题，我们就可以用轮廓程序或者其他技术去研究程序的行为，搞清楚问题究竟出在哪里。我们必须保证确实是找准了出问题的地方。这可能需要检查整个程序，而不是仅仅把精力集中到 strstr上，虽然它是明显的，但却又是误认的嫌犯。最后，我们用一个更好的算法解决了应该解决的问题，并通过测试知道它确实快了许多。一旦程序已经足够快了，我们就停止，干嘛要没事找事呢？

练习7-1 通过一个表格，实现从一个字符到以这个字符开头的一组模式的映射，能得到大约一个数量级的性能改进。实现 isspam的另一个版本，它采用两个字符作为下标变量，这样做可能得到怎样的性能改进？这些做法都是一种称为字符树 (trie)的数据结构的特例。许多这类数据结构都是用空间换时间。

China-pub.com
下载

7.2 计时和轮廓

自动计时测量。许多系统里都有这种命令，它们可用于测量一个程序到底用了多少时间。Unix系统里有关命令的名字是time：

```
% time slowprogram

real        7.0
user        6.2
sys         0.1
%
```

这执行了一个命令，报告出三个数值，都以秒数计："real(实际)"时间，程序从开始到完成的全部时间；"user"CPU时间，执行用户程序本身所花费的时间；以及"system"(系统)CPU时间，也就是花费在操作系统内部程序行为方面的时间。如果你的系统里有类似命令，请使用它。与用秒表计时相比，这样做得到的信息更多，更可靠，也更容易追踪。应该留下很好的记录。在对一个程序做工作，修改并测量的过程中，你可能积累了大量数据，过一两天就可能把人搞糊涂了(哪一个版本的速度快20%？)。我们在关于测试的一章里讨论过许多技术，它们都可以移植到性能的测量和改进方面来。用机器去运行并测量你的测试集，更重要的是使用回归测试来保证所做的修改并没有使程序崩溃。

如果在你的系统里没有time命令，或者你需要孤立地对一个函数进行计时，构造一个计时平台也很容易，与构造一个测试台差不多。C和C++提供了一个标准函数，clock，它报告程序到某个时刻总共消耗的CPU时间。可以在一个函数的执行前和执行后调用clock，测量CPU的使用情况：

```
#include <time.h>
#include <stdio.h>
    ...
    clock_t before;
    double elapsed;

    before = clock();
    long_running_function();
    elapsed = clock() - before;
    printf("function used %.3f seconds\n",
        elapsed/CLOCKS_PER_SEC);
```

CLOCKS_PER_SEC是个尺度项，表示由clock报告的计时器的分辨率。如果一个函数消耗的时间远远不到一秒，那么就应该把它放在一个循环里运行，但这时就需要考虑循环本身的开销，如果这个开销所占的比例很大的话：

```
before = clock();
for (i = 0; i < 1000; i++)
    short_running_function();
elapsed = (clock()-before)/(double)i;
```

在Java中，Date类里的函数给出的是挂钟时间，它是CPU时间的近似值：

```
Date before = new Date();
long_running_function();
Date after = new Date();
long elapsed = after.getTime() - before.getTime();
```

函数getTime的返回值以毫秒计。

使用轮廓程序。除了可靠的计时方法外，在性能分析中最重要的工具就是一种能产生轮廓文

China-pub.com 下载

件的系统。轮廓文件是对程序在哪些地方消耗了时间的一种度量。在有些轮廓文件中列出了执行中调用的各个函数、各函数被调用的次数以及它们消耗的时间在整个执行中的百分比。另一些轮廓文件计算每个语句执行的次数。执行非常频繁的语句通常对总运行时间的贡献比较大，根本没执行的语句所指明的可能是些无用代码，或者是没有合理测试到的代码。

轮廓文件是一种发现程序中执行热点的有效手段，所谓热点就是那些消耗了大部分计算时间的函数或者代码段。当然，对轮廓文件的解释也应该慎重。由于编译程序本身的复杂性、缓冲存储器和主存的复杂影响，还有做程序的轮廓文件对其本身执行所造成的影响等，轮廓文件的统计信息只能看作是近似的。

在1971年引进轮廓文件这个术语的文章中， Don Knuth写到：“一个程序中少于百分之四的部分通常占了程序一半以上的执行时间。”这也指明了轮廓文件的使用方法：弄清程序中最消耗时间的部分，尽可能地改进这些部分，然后再重新测量，看看是否有新的热点浮现出来。通常在重复一次两次以后，程序里就再也没有明显的热点了。

轮廓文件常常可以通过编译系统的一个标志或选择项打开，然后运行程序，最后用一个分析工具显示结果。在Unix上，这个标志一般是`-p`，对应的工具是`prof`：

```
% cc -p spamtest.c -o spamtest
% spamtest
% prof spamtest
```

在下面的表格里，列出的是对垃圾邮件过滤程序某个特定版本产生的轮廓文件，我们建立这个文件，以便理解程序的行为。这里用的是一个固定的消息，一个也是固定的、有 217个短语的测试集合，用它与该消息匹配 10 000次。程序运行在250MHz的MIPS R10 000上，使用`strstr`的原始实现并调用其他基本函数。程序输出经过编辑和重新编排，以便能够放在这里。请注意输入的大小(217个短语)和运行的次数(10 000)怎样出现在表的“调用”列里，该列中显示的是各个函数的调用次数，可以作为一种一致性检查。

12234768552: 总的执行的指令数
13961810001: 总的计算循环数
55.847: 总的执行时间(秒)
1.141: 平均每指令的循环数

秒	%	累计%	循 环	指 令	调 用	函 数
45.260	81.0%	81.0%	11314990000	9440110000	48350000	strchr
6.081	10.9%	91.9%	1520280000	1566460000	46180000	strncmp
2.592	4.6%	96.6%	648080000	854500000	2170000	strstr
1.825	3.3%	99.8%	456225559	344882213	2170435	strlen
0.088	0.2%	100.0%	21950000	28510000	10000	isspam
0.000	0.0%	100.0%	100025	100028	1	main
0.000	0.0%	100.0%	53677	70268	219	_memccpy
0.000	0.0%	100.0%	48888	46403	217	strcpy
0.000	0.0%	100.0%	17989	19894	219	fgets
0.000	0.0%	100.0%	16798	17547	230	__malloc
0.000	0.0%	100.0%	10305	10900	204	realfree
0.000	0.0%	100.0%	6293	7161	217	estrdup
0.000	0.0%	100.0%	6032	8575	231	cleanfree
0.000	0.0%	100.0%	5932	5729	1	readpat
0.000	0.0%	100.0%	5899	6339	219	getline
0.000	0.0%	100.0%	5500	5720	220	_malloc

China-pub.com
下载

很明显，strchr和strncmp(两个都是strstr调用的)完全主导了整个执行过程。Knuth的教诲是正确的，程序里很小的一部分消耗了大部分执行时间。当对某个程序首次做轮廓时，经常能看到一个运行最多的函数使用了 50%或更多的时间，就像在这里一样，因此很容易决定应该把注意力集中于何处。

集中注意热点。在重写了strstr之后，我们重新做spamtest的轮廓文件。发现当时99.8%的时间都消耗在strstr一个函数上，虽然整个程序确实得到了明显的加速。当某一个函数单独成为一个瓶颈的主导因素时，那么就只有两条路可以走了：或是采用一种更好的算法来改进这个函数；或者直接删去这个函数，重写环绕它的程序部分。

在目前情况下，我们采用了重写程序的方式。对于最后的 isspam的快速实现，下面是它的spamtest轮廓文件的前几行。在这里可以看到总的时间大大减少了，函数 memcmp变成了新热点，而isspam也消耗了时间中相当大的一部分。现在这个 isspam比直接调用strstr的那个版本复杂多了，但它的开销早已通过删去 strlen和strchr，并用memcmp取代strncmp得到了补偿，memcmp对每个字节做的工作少得多。

秒	%	累计%	循 环	指 令	调 用	函 数
3.524	56.9%	56.9%	880890000	1027590000	46180000	memcmp
2.662	43.0%	100.0%	665550000	902920000	10000	isspam
0.001	0.0%	100.0%	140304	106043	652	strlen
0.000	0.0%	100.0%	100025	100028	1	main

花一点时间，比较两个轮廓文件里机器循环计数和调用计数，也是很有教益的。注意这里对strlen的调用从数百万次下降到652，而对memcmp的调用则与原来对strncmp的调用次数相同。还应该注意 isspam，它现在包含了 strchr的功能，但占用周期比原来的strchr少得多，这是因为它在每步只检查相关的模式。通过检查这里的数字，可以看到执行情况的许多细节。

程序中的热点常常可以用比我们处理垃圾邮件过滤程序简单得多的方法除去，或者至少是将它冷却下来。很久以前，在一次回归测试中， Awk的一个轮廓文件指明有一个函数被调用了大约一百万次，下面是有关循环：

```
for (j = i; j < MAXFLD; j++)
    clear(j);
```

这个循环的作用是在每次读入新行之前清理各个域，它大概耗费了整个执行时间的 50%。在这里常数MAXFLD的值为200，是一个输入行里最多允许的域的个数。但是，对于大部分 Awk应用而言，实际上域的个数常常只是 2或3。这样，大量时间被白白浪费到清理那些根本没使用过的域上了。我们修改了方式，以曾经用到的最多的域个数值取代这个常数，这就使总的执行速度提高了25%。

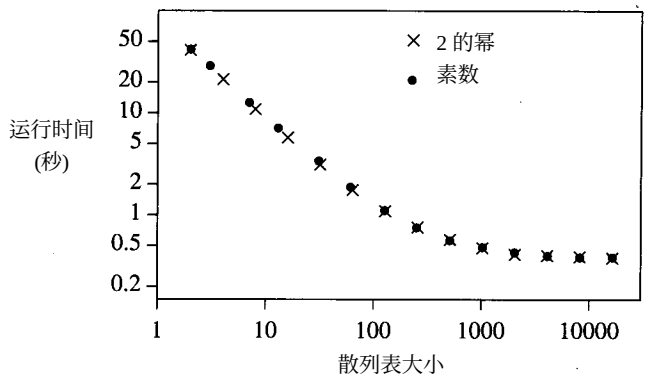
```
for (j = i; j < maxfld; j++)
    clear(j);
maxfld = i;
```

画一个图。图形特别适合用来表现性能测量的情况，它可以很好地传达信息，例如参数改变的影响、算法和数据结构的比较，有时也能指出某些没有预料到的情况。在第 5章给出过一些散列乘因子产生的链长度的图形，显示出某些乘因子比另一些好得多。

下面的图形显示的是散列数组大小对运行时间的影响，这里用的是 C版本的markov程序，输入数据是《诗篇》(共42 685个词，22 482个前缀)。我们做了两集试验，一集试验用 2的幂

China-pub.com 下载

作为数组大小，从2到16 384；另一集试验采用小于2各个幂次的最大素数作为数组大小。我们希望了解的是，采用素数大小的数组在性能方面能否产生什么可以测量到的差异。



这个图说明，当数组大小超过1000个元素之后，运行时间对数组大小的变化就不再敏感了。对于素数或者2的幂作为数组大小，我们根本看不到任何明显差异。

练习7-2 无论你的系统里有没有time命令，都请用clock或者getTime写一个为自己使用的计时程序。将它的时间与挂钟做比较，机器的其他活动对计时会有影响吗？

练习7-3 在第一个轮廓文件里可以看到，strchr被调用了48 350 000次，而strncmp只被调用了46 180 000次。请解释这个次数差。

7.3 加速策略

在改造一个程序，设法把它弄得更快之前，首先应该确定它确实太慢。然后应该通过计时工具和轮廓文件，弄清时间到底跑到哪里去了。在你知道发生了什么之后，有一些可以采用的策略。我们列出几个，按照获益递减的顺序。

使用更好的算法或数据结构。要使程序运行速度快，最重要的因素是算法与数据结构的选择，有效的算法和不那么有效的算法造成的差距是巨大的。在spam程序上，我们已经看到由于改变数据结构所产生的效率上十倍的变化。如果一个新算法降低了计算的数量级，例如从$O(n^2)$降低到$O(n\log n)$，那么速度的改善又会更大得多。第2章已经讨论过这个问题，这里就不再仔细谈它了。

应该弄清实际的复杂性正是我们所期望的，否则这里就可能隐藏着一个性能错误。下面的片段看起来是一个线性的字符串扫描算法：

```
?     for (i = 0; i < strlen(s); i++)
?         if (s[i] == c)
?             ...
```

但它实际上是平方的。如果s有n个字符，每次调用strlen都需要对这个串里的n个字符扫视一遍，而循环本身又要做n次。

让编译程序做优化。有一种毫不费力的改变就可能产生明显的加速效果，那就是打开编译系统的所有优化开关。现代编译程序已经做得非常好了，这实际上大大减小了程序员对程序做各种小改进的必要性。

在默认情况下，C和C++ 编译程序并不设法做很多优化工作，通过编译选项可以打开一个优化程序(说得更准确些，应该是“改进程序”)。按说这个选项应该是默认的，但是由于做优

China-pub.com
下载

化会妨碍源程序排错系统的工作，所以，程序员必须在确认程序已经排错完毕之后，自己来打开优化系统。

编译程序的优化通常可以在所有地方改进运行的时间性能，一般从几个百分点到两倍的样子。不过，有时优化也可能使程序变慢，所以你应该在交付产品之前重新测量一下。对垃圾邮件过滤程序的几个版本，我们比较了未优化的和优化编译的结果。对用来测试匹配算法最后版本的测试集，原来的运行时间是8.1秒，打开优化开关后时间降到5.9秒，性能改进了大约25%。另一方面，对于使用原始strstr的版本，优化后性能并没显示出什么改进，因为strstr在装入程序库之前已经进行过优化，而优化程序只对被编译的源代码工作，也不再去处理系统库。实际上，也有些编译系统能做全局优化，它们分析整个程序，寻找所有可以改进的地方。如果在你的系统里有这种编译程序，那么也不妨试试它，说不定它能够进一步挤出一些指令周期来。

还有一个问题应该引起警惕，那就是编译优化做得越多越深入，把错误引进编译结果（程序)的可能性也就越大。在打开了优化程序之后，应该重新运行回归测试集，就像你自己做了什么改动一样。

调整代码。只要数据有足够的规模，算法的正确选择就会显示它的作用。进一步说，算法方面的改进是跨机器、跨系统和跨语言的。但是，如果已经选择了正确的算法，程序的速度仍然是问题的话，下一步还能做的就是调整代码，整理循环和表达式的细节，设法使事情做得更快些。

第7.1节最后给出的isspam版本并没有经过仔细调整。这里我们要说明，通过翻转有关循环，可以进一步改进程序的性能。为帮助回忆，我们把原来的东西抄在这里：

```
for (j = 0; (c = mesg[j]) != '\0'; j++) {
    for (i = 0; i < nstarting[c]; i++) {
        k = starting[c][i];
        if (memcmp(mesg+j, pat[k], patlen[k]) == 0) {
            printf("spam: match for `%s'\n", pat[k]);
            return 1;
        }
    }
}
```

这个初始版本在经过优化编译后，运行我们的测试集需要6.6秒。在内部循环的循环条件里有一个数组下标(nstarting[c])，对于外层循环的各次重复执行，这个值都是固定的。我们可以把这个值存储在一个局部变量里，以避免反复重新计算：

```
for (j = 0; (c = mesg[j]) != '\0'; j++) {
    n = nstarting[c];
    for (i = 0; i < n; i++) {
        k = starting[c][i];
        ...
```

这使程序运行时间减少到5.9秒，大约提高了10%，这是调整代码能得到的典型加速情况。在这里还有一个可以揪出来的变量，starting[c]也是固定的，把它拉到循环之外可能也有所帮助。通过测试之后，我们发现这个改动并没有产生显著影响。这个情况在代码调整中也是很典型的，调整某些东西有作用，而调整另一些东西则没有作用，人必须自己设法确定各种情况。此外，对于不同的机器或者编译系统，情况可能又会不同。

对于垃圾邮件过滤程序，还有一个改造也是可以做的。在内层循环里，程序用整个模式

China-pub.com 下载

与字符串做比较，而算法实际上保证了第一个字符已经做过匹配。代码可以调整，让 memcmp 从一个字节之后开始做。我们试验了这个改进，得到 3%的加速。这个收获确实不大，但只需要修改程序中的三行，其中有一行在预先计算中。

不要优化那些无关紧要的东西。有时所做的代码调整毫无作用，这就是因为用到了一些不能产生差异的地方。首先应该确认你优化的代码正是那些真正耗费时间的东西。下面的故事可能是虚构的，但我们还是要说一说。有人对一个现已倒闭的公司的一种早期机器做分析，安装了一个硬件执行监控器，发现系统把 50%的时间花在执行同一个包含几条指令的序列上。工程人员构造了一条特殊指令来封装这个指令序列的功能，重新构造好系统之后，却没有看到任何变化。原来他们优化的是操作系统的空转循环。

我们到底应该在加快程序速度方面花多少时间？最重要的标准就是看所做的改造能否带来足够的效益。作为一个准则，在加快程序速度方面所花的时间不应该超过这种加速在程序的整个生存期间中获得的时间。按照这个规则，对 isspam的算法改造是值得做的：这个工作用掉大约一天时间，但在以后的每一天里都节约了 (还将继续节约)几个小时。从内层循环里去掉一个数组下标的收获没那么激动人心，但也是值得做的，因为这个程序是在为一个大机构提供一种服务。对于像垃圾邮件过滤程序或者程序库这样的公共服务程序，优化通常都是很值得的，而加快一个测试程序的速度几乎就没有什么价值了。如果有一个程序要运行一年，那么就应该从中挤压出你所能做到的一切。甚至在这个程序已经运行了一个月以后，如果你发现了一种能使它改进百分之十的方法，可能也值得从头再开始一次。

存在竞争对象的程序 —— 游戏程序、编译系统、字处理系统、电子表格系统和数据库系统等 —— 都应该纳入这个类别，因为商业上的成功常常就在这细微的差别上，至少是在公开的标准测试结果方面。

一件最重要的工作就是在做了修改之后对程序重新计时，以确认情况真正得到了改善。实际中也有这样的情况，两个修改分开来看对程序都有改进，但它们却会互相影响，对另一方的改进起负面作用。也存在这种情况，由于计时机制的误差太大，以至无法对修改程序的影响做出准确评价。即使是在单用户系统里，时间也可能有无法预计的起伏波动。如果内部计时器的变数是 10%(或至少报告给你的时间是如此)，那么对那些只产生了 10%改进的修改，我们就很难把它们与噪声区分开来。

7.4 代码调整

在发现热点之后，存在许多可以使用的能缩短运行时间的技术。这里提出一些建议，不过使用时都应该小心，使用之后应该做回归测试，确认结果代码还能工作。请记住，好的编译程序能为我们做其中的许多优化工作，而且你所做的事情有时也可能使程序变得更加复杂，从而可能妨碍编译程序的工作。无论你做了些什么，都应该通过测量，确定其作用，弄清它是否真的有所帮助。

收集公共表达式。如果一个代价昂贵的计算多次出现，那么就只在一个地方做它，并记录计算的结果。例如在第 1章里我们给出过一个宏，它在一行里用同样的值两次调用 sqrt，以这种方式计算距离。这个计算的实际作用是：

```
?     sqrt(dx*dx + dy*dy) + ((sqrt(dx*dx + dy*dy) > 0) ? ...)
```

应该只算一次平方根，而在两个地方使用得到的值。

China-pub.com
下载

如果一个计算出现在循环里，而它又不依赖任何在循环过程中改变的东西，那么就应该把它移到循环之外。就像我们把：

```
for (i = 0; i < nstarting[c]; i++) {
```

改造为：

```
n = nstarting[c];
for (i = 0; i < n; i++) {
```

用低代价操作代替高代价的。术语降低强度指的是用低代价操作代替高代价操作的那些优化。在过去，这个说法特别用来指用加法和移位操作取代乘法。今天再这样做，恐怕不会有太大收获了。除法和求余数比乘法慢得多，如果可以用乘倒数的方法来代替除法，或者在除数是 2 的幂时用掩码操作代替求余，则确实可能得到性能改进。在 C和C++ 里，用指针代替数组下标有可能提高速度，但是今天的大部分编译程序都已经能自动做这种优化了。用简单计算代替函数调用仍然是有价值的。平面上的距离由公式 `sqrt(dx*dx+dy*dy)`确定，因此，要确定两个点中哪个离得更远，按正规方式需要算两个平方根。但是，同样的判断也可以通过比较两个距离的平方做出来。

```
if (dx1*dx1+dy1*dy1 < dx2*dx2+dy2*dy2)
    ...
```

这样给出的结果与比较表达式的平方根完全一样。

另一个例子与文本模式匹配有关，我们在垃圾邮件过滤程序和 `grep`里都用到这种东西。如果模式的开始是个文字字符，我们可以用这个字符在输入文本中快速地查找下去。如果这样做也找不到匹配，更昂贵的查找机制根本就不需要调用。

铺开或者删除代码。循环的准备和运行都需要一定的开销。如果循环体本身非常小，循环次数很少，有一个更有效的方法，就是把它重写为一个重复进行的计算序列。例如，把下面的循环：

```
for (i = 0; i < 3; i++)
    a[i] = b[i] + c[i];
```

改造为：

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
```

这样可以去掉循环的开销和特殊的分支操作，而这些都会造成执行流的中断，降低现代处理器的速度。

如果循环的次数很多，也可以做一些类似形式的变换，通过较少的重复来减少开销。例如把：

```
for (i = 0; i < 3*n; i++)
    a[i] = b[i] + c[i];
```

改为：

```
for (i = 0; i < 3*n; i += 3) {
    a[i+0] = b[i+0] + c[i+0];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
}
```

China-pub.com 下载

请注意，只有在循环长度是步数大小的整倍数时，这种方式才是适用的。若不是这样，在代码最后还需要增加一些补充代码，这常常会成为潜藏错误的地方，有些情况下也可能使效率的收获重新丧失。

高速缓存频繁使用的值。以缓冲方式保存的值无须重新计算。缓存的价值来自于局部性。程序和人都有这种倾向，那就是重复使用最近访问过的值，或者是近旁的值，而对老的值和远距离的值则用得比较少。计算机硬件领域里广泛使用了缓存技术，给计算机增加缓冲存储器后，确实能大大地改进机器在速度方面的表现。对于软件，情况也是一样的。例如， Web浏览器应用缓存技术，保存页面和图形，以减少慢速的 Internet数据传输。在前几年我们写的一个打印预览程序里，如½这样的非字母字符必须通过查表方式处理。我们统计了这些特殊字符的使用情况，发现大部分使用都是用这种字符排成一个行，用的是同一字符的长长的序列。将最近用过的一个字符缓存起来，就能大大提高程序对典型输入的处理速度。

最好是把缓存操作对外部隐蔽起来。这样，这种操作除了使程序运行得更快外，不会对程序其他部分有任何影响。在上面的打印预览程序的例子里，列出一个字符的函数并没有改变，它一直是：

```
drawchar(c);
```

函数drawchar原来的版本是直接调用show(lookup(c))。使用缓存技术的函数定义了一个内部的静态局部变量，用于记录前一个字符值。有关代码是：

```
if (c != lastc) { /* update cache */
    lastc = c;
    lastcode = lookup(c);
}
show(lastcode);
```

写专用的存储分配程序。经常可以看到这种情况，程序里的惟一热点就是存储分配，表现为对malloc和free的大量调用。如果程序中需要的经常是同样大小的存储块，采用一个特定用途的存储分配器取代一般的分配器，有可能使速度得到实质性提高。这种特定的存储分配器调用malloc一次，取得基本存储块的一个大数组，在随后需要时一次送出去一块，这是一个代价很低的操作。释放后的存储块接在一个自由表的最后，这使它们可以立即重新投入使用。

如果所需要的块在大小上差不多，你也可以用空间来交换时间，总是分配能够满足最大需求的块。对所有长度不超过某个特定值的字符串都使用同样大小的块，这是管理短字符串的一种有效方法。

有些算法可以采用栈方式的存储管理，先完成了一系列的存储分配，然后将整个集合一下子释放掉。在这种情况下，分配器可以先取得一个大块，而后像用一个栈似的使用它，需要的时候将分配的项目压入，结束时通过一个操作就把它们都弹出去。有些 C函数库里为这种分配方式提供了一个alloca函数，但它不是标准的。这个函数用一个局部栈作为存储资源，当调用alloca的函数返回时，自动释放掉所有的项目。

对输入输出做缓冲。缓冲方式使数据传输操作以成批的方式完成，这能使频繁操作所造成的负担减到最小，并使代价昂贵的操作只在不可避免时才进行，使这种操作的代价能分散到许多数据值上。例如，当C程序调用printf操作时，有关字符被存入一个缓冲区，而不是直接传给操作系统，直到缓冲区满，或者是显式地执行刷新操作。操作系统本身也可能推迟向磁

China-pub.com
下载

盘写数据的动作。这种方式也有缺点。为了使数据变成可见的，就必须对输出缓冲区做刷新。最糟糕的情况是，如果程序垮台，驻留在缓冲区里的数据就会丢失。

特殊情况特殊处理。通过使用特殊代码去操作同样大小的对象，特殊用途的分配器可以比通用分配器节约时间和空间开销，还可能减少碎片问题。在 Inferno系统的图形库里，基本的draw函数先用最简单、最直截了当的方式写出来。在这个东西能工作之后，再把对各种各样情况的优化(通过轮廓文件来选择)以一次一个的方式加进来。这样，每次都可以用简单版本与优化版本对照测试。到最后，也只有十来个特殊情况做了优化。产生这种情况的原因是，通过分析绘图函数调用的动态情况，我们发现其中最大量的还是做字符显示，因此没必要对所有情况都写出巧妙的代码。

预先算出某些值。有时可以让程序预先计算出一些值，需要时拿起来就用，这也可能使程序运行得更快些。我们已经在垃圾邮件过滤程序里看到过这种方式，在那里首先计算出所有strlen(pat[i])的值，将它们存入数组元素patlen[i]中。如果一个图形程序需要反复计算某个数学函数，例如正弦函数，但又只需要对某个离散集合里的某些值做计算，例如对所有整数角度做计算。那么预先算出这个 360项的表(甚至是把它作为数据提供给程序)，需要时直接通过下标取用，肯定能够快很多。这又是一个用空间交换时间的例子。用数据代替代码，或者在编译时预先完成某些计算的可能性非常多，这些都可以节约时间，有时甚至还能节约空间。例如，像isdigit这一类函数，通常是预先定义一个位标志的表，然后通过下标方式实现的，并不是通过一系列的测试来实现。

使用近似值。如果精度不太重要，那么就尽量使用具有较低精度的数据类型。在老的或者小的机器上，或者在那些采用软件模拟方式实现浮点数的机器上，单精度浮点运算通常比双精度运算更快一些，所以，用 float而不是double就有可能节省时间。一些新型的图形处理器也采用了类似技巧。IEEE的浮点数标准要求“ 适度下溢 ”，把它作为可表达的数值在最低端的计算方式，但是这种计算方式是昂贵的。对于图像而言，这种特性就完全没有必要，直接将它截断为0要快得多，也是完全可以接受的。这样做，不仅能在数值出现下溢时节省时间，还可以简化整个算术运算硬件。采用整数的 sin和cos函数是使用近似值的另一个例子。

在某个低级语言里重写代码。低级语言程序的效率可能更高，不过这样做也要付出代价，那就是程序员的时间。如果把 C++或Java程序里的某些关键部分用 C语言重写，或者用某种编译语言的程序取代一个解释性脚本程序，都可能使程序加快许多。

有时，使用依赖于机器的代码也可能得到显著的加速效果。但这是最后一招，是不该轻易采用的，因为它破坏了可移植性，也使将来的维护和修改都变得更加困难。实际中也常常出现这种需要，在这种情况下，用汇编语言描述的操作应该是些相对较小的函数，而且应该嵌入一个库里，memset、memmove以及一些图形操作都是这方面的典型例子。这时应该采用的方式是，首先在某种高级语言里以尽可能清晰的方式写出代码，并通过像我们在第 6章对memset描述的那样做好测试，确定它是正确的。这是你的可移植版本，它能在任何地方工作，可惜是慢了一点。当你遇到一个新环境时，就可以从这个已知能够工作的版本开始入手。如果你写好了一个汇编语言版本，就可以利用可移植版本对新版本做彻底的测试。当测试发现错误时，不可移植的版本总是最大的疑点。有一个可做比较的实现确实是非常舒服的。

练习7-4 要使memset一类函数运行得更快，一个方法就是让它一次写一个字，而不是一次写一个字节。这样做可能与硬件匹配得更好，也能以 4或者8的因子减少循环开销。不利的一

China-pub.com 下载

面是，这时需要处理各种端点情况，目标位置可能不是开始在与字对齐的边界点，长度也可能不是字大小的整数倍等等。写一个memset版本实现这种优化。将它的性能与现存的库版本，以及直截了当的一次一个字节的版本做些比较。

练习7-5 为C语言的字符串专门写一个存储分配器 smalloc，它对小字符串使用一个专用分配器，对大字符串则直接调用 malloc。在这种情况下，你可能需要定义一个 struct，以表示串的两种情况。你怎样确定何时从调用 smalloc转到malloc？

7.5 空间效率

存储空间过去一直是最贵重的计算资源，而且总是短缺的，想从这种没有多少油水的地方榨出许多东西来，就会形成特别坏的程序设计。声名狼藉的“两千年问题”是经常被人提起的典型例子。当存储器真正稀缺的时候，甚至为存储19所需要的那两个字节也被认为是太昂贵了。无论缺少空间是否是其中真正的原因(实际上，这种代码也直接反映了人们在日常生活中写日期的习惯，其中的世纪总是被省略的)，这件事总是说明了一个短视的优化所具有的内在危险性。

无论如何，时代已经不同了，主存和二级存储器都已经便宜得令人感到惊异。这样，优化空间的第一要义应该和改善速度完全一样：别为它费心。

实际中，确实还是存在某些情况，在那里空间效率是非常重要的。如果一个程序无法放进可用的存储里，对它的各部分必须反复做页面倒换，而这又会使性能降低到令人无法容忍的程度。我们都熟悉这种情况，新版本的软件大量地挥霍着存储器。一个可悲的现实是，软件升级通常都伴随着存储器的大量销售。

使用尽可能小的数据类型以节约存储。提高空间效率的一个步骤是做些小修改，使现有存储能使用得更好，例如使用能满足工作需要的最小数据类型。比如说，如果合适的话，可以用short取代int，这是2-D图形系统常用的一种技术，因为16位一般足以处理屏幕坐标的可能取值范围。或许是用 float代替double，这带来的潜在问题是精度损失，因为 float一般只能保持6到7位十进制数字。

对于这些修改，或者其他类似情况，程序也必须做一些改动，值得注意的是 printf和(特别是)scanf语句的格式描述。

这种方法的逻辑延伸是将信息编码到字节里面，甚至到若干个二进制位里，如果可能的话就只使用一个位。请不要使用 C或C++的位域，它们是高度不可移植的，而且倾向于产生大量的低效代码。你应该把所需要的操作都封装在函数里面，在这些操作中利用移位和掩码，取出或者设置字或字的数组里的位。下面的函数能够从一个字中间取出连续一段位值：

```
/* getbits:  get n bits from position p */
/*  bits are numbered from 0 (least significant) up */
unsigned int getbits(unsigned int x, int p, int n)
{
        return (x >> (p+1-n)) & ~(~0 << n);
}
```

如果这种函数太慢了，你可以用在本章前面描述的技术改进。 C++ 语言的操作符可以重载，这样就可以把对二进制位的访问做成正规的下标形式。

不存储容易重算的东西。与代码调整类似，这方面的修改也不太重要。最重要的改进应该是来自好的数据结构，或许还要伴随着算法的修改。这里有一个例子：许多年前一位同事来找

China-pub.com
下载

作者之一，该同事正试图做一个矩阵计算，但是这个矩阵实在太大了，他必须关掉计算机，重新装入一个简化的操作系统之后才能进行处理。这种操作方式实在是个噩梦，该同事想知道是不是有其他办法。我们问矩阵是什么样的，了解到的情况是，矩阵里存储着整数值，而其中大部分都是零。实际上，非零矩阵元素还不到5%。这立即提醒我们采用另一种表示方式，其中只存储矩阵的所有非零元素，`m[i][j]`一类形式的矩阵元素访问用函数调用`m(i,j)`取代。有许多存储这种数据的方法，最简单的就是用指针数组，一个指针对应矩阵的一行，它指向一个压缩数组，其元素是列号和对应的数据元素值。对于每个非零元素而言，采用这种方法耗费了较大空间，但就整个矩阵来说，所需要的空间却小多了。虽然每次元素访问都会慢一些，但从整体看则远快于重装操作系统。故事的结尾是，该同事采纳了这个建议，满意地离开了。

我们采用同样技术解决过另一个现代版本的同样问题。在一个无线电通信的设计系统里，需要表示一个很大的地理区域(边长100到200公里，解析度是100米)中的地形数据和无线电信号强度。存储这样巨大的矩形数组已经超出了目标机器的存储能力，必然会导致无法接受的页倒换开销。但是，在一些大片的部位中，地形和信号强度又几乎是一样的，因此我们采用一种层次性的表示方法，把具有同样值的区域结合成一个单元，这就使问题变得可以把握了。

这种问题有许许多多不同的变形，由此带来各种各样的特殊表示方式。这些方式有一个共同的基本思想：采用隐含的方式或者一种共同的形式存储公共值，在其他值上花更多的空间和时间。如果其中共性最强的值真正是共有的，这个招数就奏效了。

在组织程序时，复杂类型的特殊数据表示方式总是应该隐蔽在一个类里，或者隐藏在一集对私有数据类型操作的函数里。采取了这种预防措施，我们就能保证，当某个表示方法改变时，程序其他部分完全不会受影响。

空间效率方面的考虑有时也会表现在信息的外部表示方面，与转换和存储都有关系。一般地说，如果可能，最好以文本形式存储信息，而不要采用某种二进制表示形式。文本是可移植的，容易读，可以任由各类工具处理。二进制表示则完全没有这些优点。倾向二进制的论据通常都基于“速度”，实际上这种说法是很值得怀疑的，因为在文本形式和二进制形式之间的差异可能没那么大。

空间效率的获得往往要付出时间代价。在某个应用里，需要把大的图像从一个程序传给另一个程序。有一种简单的图像格式叫PPM，其典型大小是一兆字节左右。由此我们设想，如果把图像编码到压缩的GIF格式，其典型大小是50K字节，这样传输工作一定进行得更快。但是，完成到GIF的编码以及对应的解码要花费时间，与传送一个短文件节约的时间差不多，这样我们就不会有什么收获。处理GIF格式的代码大约有500行那么长，而处理PPM的源程序大约是10行。为了维护的方便，GIF编码方式被放弃了，在这个应用里继续使用PPM方式。当然，如果该文件是通过一个速度很慢的网络传输的，权衡结果就可能不同，GIF编码方式很可能更为有效。

7.6 估计

要预先估计一个程序能运行得多么快，通常是非常困难的，而要估计某个特殊程序语句或者机器指令的时间代价，那就是双倍的困难了。然而，为一个语言或者系统做一次代价模拟却是比较简单的，它至少能使你对各种重要操作花费的时间有一个粗略的概念。

China-pub.com 下载

对于常规的程序设计语言，可以用一个为有代表性的代码序列做计时的程序。在这里实际上存在着许多困难，比如很难取得可以重现的结果、难以消除无关开销等等。不过，这至少是一种不费多少事就能得到一些有用信息的途径。例如，我们用一个 C和C++ 代价模拟程序估计了一些独立语句的代价，采用的方法就是把它们放在循环里，运行成百万次，然后计算出平均时间。我们用一台250 MHz的MIPS 10 000产生这些数据，操作代价以纳秒计算。

```
Int Operations
  i1++                                  8
  i1 = i2 + i3                         12
  i1 = i2 - i3                         12
  i1 = i2 * i3                         12
  i1 = i2 / i3                        114
  i1 = i2 % i3                        114

Float Operations
  f1 = f2                               8
  f1 = f2 + f3                         12
  f1 = f2 - f3                         12
  f1 = f2 * f3                         11
  f1 = f2 / f3                         28

Double Operations
  d1 = d2                               8
  d1 = d2 + d3                         12
  d1 = d2 - d3                         12
  d1 = d2 * d3                         11
  d1 = d2 / d3                         58

Numeric Conversions
  i1 = f1                               8
  f1 = i1                               8
```

整数计算是极快的，但除法和取模除外。浮点运算也很快，甚至可能更快。对于在浮点运算比整数运算昂贵得多的年代里成长起来的人们而言，看到这些一定很吃惊。

其他基本操作也都相当快，包括函数调用，至少在下面一组的最后三行里：

```
Integer Vector Operations
  v[i] = i                             49
  v[v[i]] = i                          81
  v[v[v[i]]] = i                      100

Control Structures
  if (i == 5) i1++                      4
  if (i != 5) i1++                     12
  while (i < 0) i1++                    3
  i1 = sum1(i2)                        57
  i1 = sum2(i2, i3)                    58
  i1 = sum3(i2, i3, i4)                54
```

但是，输入输出操作就不那么便宜了，大部分库函数也是如此：

```
Input/Output
  fputs(s, fp)                        270
  fgets(s, 9, fp)                     222
  fprintf(fp, "%d\n", i)             1820
  fscanf(fp, "%d", &i1)              2070

Malloc
  free(malloc(8))                     342
```

China-pub.com
下载


```
String Functions
  strcpy(s, "0123456789")              157
  i1 = strcmp(s, s)                    176
  i1 = strcmp(s, "a123456789")          64

String/Number Conversions
  i1 = atoi("12345")                   402
  sscanf("12345", "%d", &i1)          2376
  sprintf(s, "%d", i)                 1492
  f1 = atof("123.45")                 4098
  sscanf("123.45", "%f", &f1)         6438
  sprintf(s, "%6.2f", 123.45)         3902
```

这里有关于malloc和free的时间，它可能对真实计算没有多少指导意义，因为刚刚分配后立即释放并不是一种典型模式。

最后是数学函数：

```
Math Functions
  i1 = rand()                          135
  f1 = log(f2)                         418
  f1 = exp(f2)                         462
  f1 = sin(f2)                         514
  f1 = sqrt(f2)                        112
```

在不同的硬件上，有关的具体数据应该是不同的。但是，作为一种大致走向，这些数据还是可以用于粗略估计某些事情可能需要多长时间，或者比较I/O和基本操作之间的相对代价，或者确定是否有必要重写一个表达式、使用一个在线的函数，等等。

在这里也还有很多变数。一个是编译系统优化的级别。现代编译系统完全可能发现一些能难倒大部分程序员的优化。进一步的问题是，目前的CPU极端复杂，只有好的编译程序才能充分利用CPU能力方面的优势，同时发送多条指令，将它们的执行过程流水线化，在需要之前提取出有关的指令和数据，以及完成许多其他的类似事项。

对有关执行情况难以做出预计，另一个重要的原因是计算机的体系结构。缓冲存储器极大地改变了机器速度，灵巧的硬件设计几乎能够掩盖这样的事实：主存储器实际上比缓冲存储器慢很多很多。处理器主频，例如“400 MHz”，只有提示的意义，并没有讲出全部故事。我们的一台老的200 MHz奔腾机比另一台更老的100 MHz奔腾机明显慢了许多，就是因为后者有一个很大的二级缓存而前者没有。另外，不同代的处理器，即使它们的指令系统相同，为完成一个特定指令所用的时钟周期也可能是不同的。

练习7-6 为你身边的各种计算机或编译系统建立一个估计基本操作代价的测试集，并研究它们在性能方面的相似性和差异。

练习7-7 为C++语言的一些高级操作建立一个代价模型，有关的特征可能包括：类成员的构造、复制和删除，成员函数调用，虚函数，在线函数，iostream库，STL库。这个练习完全是开放性的，请集中于一小批有代表性的操作。

练习7-8 对Java重做上述练习。

7.7 小结

如果你已经选择了正确的算法，那么只有到了写程序的最后，才有可能要关心执行的优

China-pub.com 下载

化问题。如果你必须做这件事情，进行工作的基本循环应该是：测量，把注意力集中到若干个做一点修改就能产生最大改进的地方，验证你所做修改的正确性，然后再测量。在能停下的时候立刻停下来。在这里还应该保留那个最简单的版本，作为计时和验证正确性的基础。

当你要着手改进一个程序的速度或者空间耗费时，一个很好的做法是先建立一些基准测试和问题，这样你就很容易做出估计，并可以为自己保存性能的变化轨迹。如果对你的工作已经有某些标准的基准测试，那么就应该使用它们。如果程序是相对自足的，可以采用的一种办法是找到或者建立一集典型输入，这些也可以成为测试集的一部分。实际上，这也正是那些为编译系统、计算机或者其他类似东西而使用的基准测试集的起源。例如， Awk带着大约20个小程序，它们的全体覆盖了大部分常用的语言特征。这些程序运行时要处理很大的输入文件，以保证能够计算出同样结果，又没有引进性能缺陷。我们还有一集标准的大数据文件，它们用于做计时测试。在某些情况下，如果这些文件有特别容易辨识的特征也很有帮助，例如其大小是10或者2的幂。

对于基准测试，也可以用我们在第 6章讨论测试时提出的测试台来管理。计时测试可以自动执行，其输出中应该包含足够多的标志，使它们能够被理解和复制。还应该保存测试记录，以便于研究其中重要的趋势和变化。

当然，要想做出好的基准测试也是很困难的。大家都知道，某些公司设法调整它们的产品，使之在基准测试中表现良好。因此，对所有基准测试带着点疑问是很明智的。

补充阅读

我们关于垃圾邮件过滤程序的讨论是基于 Bob Flandrena和Ken Thompson的工作。他们的过滤程序包括正则表达式，以便于描述复杂的匹配，还包括根据与串匹配的情况自动对消息进行分类(肯定是垃圾邮件、可能是垃圾或不是垃圾)。

Knuth关于轮廓文件的文章“ FORTRAN程序的实证研究 ” (An Empirical Study of FORTRAN Programs)发表在《软件：实践和经验》(Software: Practice and Experience第1卷第2期pp105~133，1971)。该文的核心是对一些从废纸篓里和某计算机中心机器上公开可见的目录里翻出来的程序的统计分析。

在Jon Bentley的《程序设计精萃》和《更多的程序设计精萃》 (Programming Pearls，More Programming Pearls，Addison-Wesley，1986和1988)里，有一些关于算法和代码调整改进的极好实例，对使用测试台改善性能和使用轮廓文件也有很好的讨论。

Rick Booth的《内部循环》(Inner Loops，Addison-Wesley，1997)是关于调整PC程序的一本很好的参考书。但是，由于处理器更新得太快，书中的某些细节很快就过时了。

John Hennessy和David Patterson关于计算机体系结构的一套书，例如《计算机组织和设计：硬件/软件接口》，(Computer Organization and Design: The Hardware/Software Interface，Morgan Kaufman，1997)深入论述了当前计算机性能方面的许多问题。