China-pub.com
下载

第4章 界 面

在造墙之前，我必须设法弄清
该把什么放在墙里，什么放在墙外，
最需要防御的又是什么。
确实有些东西不喜欢墙，
总希望它倒下来。

—— Robert Frost，《修墙》

设计的真谛，就是在一些互相冲突的需求和约束条件之间寻找平衡点。如果要写的是一个自给自足的小程序，那么常常可以找到许多折衷方式，所做出的特定选择将产生一些后果，会遗留在系统里，但其影响还只限于写程序的个人。如果写出的代码是为了别人使用的，这些选择决定就会产生更广泛的影响。

在进行设计的时候，必须考虑的问题包括：

- 界面：应提供哪些服务和访问？界面在效能上实际成为服务的提供者和使用者之间的一个约定。在这里要做的是提供一种统一而方便的服务，使用方便，有足够丰富的功能，而又不过多过滥以至无法控制。
- 信息隐藏：哪些信息应该是可见的，哪些应该是私有的？一个界面必须提供对有关部件的方便访问方式，而同时又隐蔽其实现的细节。这样，部件的修改才不会影响到使用者。
- 资源管理：谁负责管理内存或者其他有限的资源？这里的主要问题是存储的分配和释放，以及管理共享信息的拷贝等。
- 错误处理：谁检查错误？谁报告？如何报告？如果检查中发现了错误，那么应该设法做哪些恢复性操作？

在第2章里，我们关注的主要是孤立的程序片段 —— 数据结构，它们是构造各种系统的基础。在第3章我们考察了怎样把它们组合成一个小程序。现在我们的论题将转到部件之间的界面，这些部件的来源又可能不同。在这一章里，我们将针对一个日常任务构造出一个函数库和一些数据结构，通过这个工作展示界面设计中的问题。其间我们还要提出一些设计原则。在设计界面时，我们需要做出大量的决定，而其中绝大部分常常又是在无意识中做出的。如果在这个过程中不遵循某些原则，产生出来的可能就是某种非常随意的界面，它们将会妨害甚至挫败我们日常的程序设计工作。

4.1 逗号分隔的值

逗号分隔的值(comma-separated value)，或CSV，是个术语，指的是一种用于表示表格数据的自然形式，使用很广泛。这里表格的每行是个正文行，行中不同的数据域由逗号分隔。看前面一章最后的那个表格，其开始的一段用 CSV格式表示大概是：

China-pub.com 下载

```
,"250MHz","400MHz","Lines of"
,"R10000","Pentium II","source code"
C,0.36 sec,0.30 sec,150
Java,4.9,9.2,105
```

这种格式通常由某些程序(如电子报表程序等)进行读写。自然，这种形式也会出现在 Web 页里，作为一种服务项目，例如显示股票价格行情表等。一个 Web页上典型的股票行情表显示出来可能是这种样子：

Symbol	Last Trade		Change		Volume
LU	2:19PM	86-1/4	+4-1/16	+4.94%	5,804,800
T	2:19PM	60-11/16	−1-3/16	−1.92%	2,468,000
MSFT	2:24PM	106-9/16	+1-3/8	+1.31%	11,474,900

下载电子表格格式

以交互方式，用一个 Web浏览器提取数据是可行的，但是又很耗费时间，令人生厌：启动浏览器、等待、观看狂泻而来的广告，输入一个股票表、等待、等待、等待，观看又一轮倾泻，最后才得到几个数。要处理这些数还要做许多交互式操作：选择“下载电子表格格式”链接，取得一个包含着差不多同样数据的文件，其中有些像下面这样的 CSV数据行(经过编辑，以便放在这里)：

```
"LU",86.25,"11/4/1998","2:19PM",+4.0625,
        83.9375,86.875,83.625,5804800
"T",60.6875,"11/4/1998","2:19PM",-1.1875,
        62.375,62.625,60.4375,2468000
"MSFT",106.5625,"11/4/1998","2:24PM",+1.375,
        105.8125,107.3125,105.5625,11474900
```

在这个过程中，明显欠缺的是让机器帮助做的原理。浏览器使你的计算机可以访问位于远程服务器的数据，但更方便的是在无须人工交互的方式下提取数据。在所有按钮动作的后面实际上是一个纯粹的文本处理过程 —— 浏览器读HTML，你输入一些文本，浏览器把它送到服务器，再读一些回来。如果有合适的工具和语言，自动提取数据是不难做到的。下面写的是一段Tcl语言的程序，它能访问有关股票行情的 Web站点，提取上面说的那种形式的 CSV数据，数据中带有几个前导行：

```
# getquotes.tcl: stock prices for Lucent, AT&T, Microsoft

set so [socket quote.yahoo.com 80]  ;# connect to server
set q "/d/quotes.csv?s=LU+T+MSFT&f=sl1d1t1c1ohgv"

puts $so "GET $q HTTP/1.0\r\n\r\n"  ;# send request
flush $so
puts [read $so]                     ;# read & print reply
```

跟在&符号后面那段神秘的序列 f=…是一段不作为文档的控制串，其作用就像 printf的第一个参数，用来确定被提取数据值是什么。经过试验，我们确定其中 s表示股票符号，l1是最终价格，c1是昨天以来的变化，如此等等。这里重要的并不是细节，因为它们都很可能会改变，最重要的是自动化：在没有人工干预的情况下提取所需数据，并把它们转换为人们需要的形式。我们应该让机器做这些事。

在典型情况下，运行 getquotes程序只需要几分之一秒的时间，远快于用浏览器来做。一旦有了数据，我们将希望能对它做进一步处理。如果能有一个方便的库，完成数据格式的

China-pub.com
下载

转入转出，像CSV这样格式的数据是很容易使用的。库里最好还带有另一些辅助处理功能，例如数值转换等等。但是，我们不知道哪里有处理 CSV数据的库，因此只能自己写一个。

在下面几节里，我们要写出这个库的三个版本，其功能是读 CSV数据，将它转换为内部表示。库是需要与其他软件一起工作的软件，在这个工作中，我们要讨论一些在设计这类软件时经常遇到的问题。例如，由于没有 CSV的标准定义，它的设计不能是基于精确规范进行的，这也是界面设计时经常面临的情况。

4.2 一个原型库

我们不大可能在第一次设计函数库或者界面时就做得很好。正如 Fred Brooks有一次写的“要计划扔掉一个，你总会这样做，无论是以什么方式”。Brooks的话是针对大系统说的，但是，这种说法实际上适用于任何实质性的软件片段。事情往往是这样，只有在你已经构造和使用了程序的一个版本之后，才能对如何把系统设计正确有足够的认识。

基于这种理解，我们构造 CSV库时准备采用的途径就是：先搞出一个将要丢掉的，搞出一个原型。我们的第一个版本将忽略许多完备的工程库应该牵涉的难点，但却又必须足够完整和有用，以便能帮助我们熟悉问题。

我们的出发点是一个名为 csvgetline的函数，它由文件读入一个 CSV数据行，将它放入缓冲区，在一个数组里把该行分解为一些数据域，删除引号，最后返回数据域的个数。以前我们已经在自己熟悉的几乎所有语言里写过类似代码，所以这是一个熟悉的工作。下面是用C语言写的原型版本，这里用问号做标记，是因为它仅仅是个原型：

```
?   char buf[200];      /* input line buffer */
?   char *field[20];    /* fields */
?
?   /* csvgetline: read and parse line, return field count */
?   /* sample input: "LU",86.25,"11/4/1998","2:19PM",+4.0625 */
?   int csvgetline(FILE *fin)
?   {
?       int nfield;
?       char *p, *q;
?
?       if (fgets(buf, sizeof(buf), fin) == NULL)
?           return -1;
?       nfield = 0;
?       for (q = buf; (p=strtok(q, ",\n\r")) != NULL; q = NULL)
?           field[nfield++] = unquote(p);
?       return nfield;
?   }
```

函数前面的注释包含了本程序能接受的输入行的一个例子，对于理解那些剖析较复杂输入的程序而言，这种形式的注释是非常有帮助的。

由于CSV数据太复杂，不可能简单地用函数 scanf做输入剖析，我们使用了C标准库函数 strtok。对strtok(p, s)的调用将返回p中的一个标识符的指针，标识符完全由不在s中的字符构成。strtok将原串里跟在这个标识符之后的字符用空字符覆盖掉，用这种方式表示标识符的结束。在第一次调用时，strtok的第一个参数应该是原来的字符串，随后的调用都应该用NULL作为第一个参数，指明这次扫描应该从前次调用结束的地方继续下去。这是一个很糟糕的界面，在函数的不同调用之间，strtok需要在某个隐秘处所存放一个变量。这样，同

China-pub.com 下载

时激活的调用序列就只能有一个，如果有多个无关的调用交替进行，它们之间必定会互相干扰。

函数unquote的功能是去除像前面例子的数据行里那些表示开头和结束的引号。它并不处理嵌套引号的问题，对于原型而言这样做已足够了。当然这种做法还不够一般。

```
?    /* unquote: remove leading and trailing quote */
?    char *unquote(char *p)
?    {
?        if (p[0] == '"') {
?            if (p[strlen(p)-1] == '"')
?                p[strlen(p)-1] = '\0';
?            p++;
?        }
?        return p;
?    }
```

下面的简单测试程序可以帮我们确认csvgetline能够工作：

```
?    /* csvtest main: test csvgetline function */
?    int main(void)
?    {
?        int i, nf;
?
?        while ((nf = csvgetline(stdin)) != -1)
?            for (i = 0; i < nf; i++)
?                printf("field[%d] = '%s'\n", i, field[i]);
?        return 0;
?    }
```

在printf里用一对单引号括起数据域，这起着划清界限的作用，还能帮助我们发现空格处理不正确一类的错误。

我们可以用这个函数处理getquotes.tcl生成的输出：

```
% getquotes.tcl | csvtest
...
field[0] = 'LU'
field[1] = '86.375'
field[2] = '11/5/1998'
field[3] = '1:01PM'
field[4] = '-0.125'
field[5] = '86'
field[6] = '86.375'
field[7] = '85.0625'
field[8] = '2888600'
field[0] = 'T'
field[1] = '61.0625'
...
```

(我们已经通过编辑方式删去了HTTP头部的行。)

现在我们有了一个原型，看起来它能对付上面给出的那种数据形式。不过，再用另外一些东西试试它可能是更明智的，特别是如果打算把它提供给别人使用。我们发现了另外的Web站点，从那里下载了一些股票行情表，得到包含类似信息的文件，但文件的格式有些不同：分隔数据记录用的是回车符号 (\r)而不是换行符号，文件最后也没有表示文件结束的回车字符。我们对它做了些格式编辑，以便能放在这里：

```
"Ticker","Price","Change","Open","Prev Close","Day High",
    "Day Low","52 Week High","52 Week Low","Dividend",
    "Yield","Volume","Average Volume","P/E"
"LU",86.313,-0.188,86.000,86.500,86.438,85.063,108.50,
```

China-pub.com
下载


```
    36.18,0.16,0.1,2946700,9675000,N/A
"T",61.125,0.938,60.375,60.188,61.125,60.000,68.50,
    46.50,1.32,2.1,3061000,4777000,17.0
"MSFT",107.000,1.500,105.313,105.500,107.188,105.250,
    119.62,59.00,N/A,N/A,7977300,16965000,51.0
```

对于这样的输入，我们的原型败得很惨。

我们在查看了一个数据来源之后设计了这个原型，并且只用同样来源的数据做过一些测试。因此，如果在使用其他来源的数据时发现了程序里的大错误，我们一点都不应该对此感到惊奇。长的输入行、很多的数据域以及未预料到的或者欠缺的分隔符都可能造成大麻烦。这个脆弱的原型作为个人使用而言可能还勉强，或者可以用来说明这种方法的可行性，但绝不可能有更多的意义。在着手开始下一个实现之前，我们需要重新认真地想一想，到底应该如何做这个设计。

现在这个原型里包含着我们的许多决定，有些是明显的，也有些是隐含的。下面列出的是前面做过的一些选择，对一个通用库而言，它们并不都是最好的选择。实际上，每个选择都提出了一个问题，需要进一步仔细考虑。

- 原型没有处理特别长的行、很多的域。遇到这些情况时它可能给出错误结果甚至垮台，因为它没有检查溢出，在出现错误时也没有返回某种合理的值。
- 这里假定输入是由换行字符结尾的行组成。
- 数据域由逗号分隔，数据域前后的引号将被去除，但没有考虑嵌套的引号或逗号。
- 输入行没有保留，在构造数据域的过程中将它覆盖掉了。
- 在从一行输入转到另一行时没有保留任何数据。如果需要记录什么东西，那么就必须做一个拷贝。
- 对数据域的访问是通过全局变量 (数组`field`)进行的。这个数组由`csvgetline`与调用它的函数共享。这里对数据域内容或指针的访问都没有任何控制。对于超出最后一个域的访问也没有任何防御措施。
- 使用了全局变量，这就使得这个设计不能适合多线程环境，甚至也不允许两个交替进行的调用序列。
- 调用库的程序必须显式地打开和关闭文件， `csvgetline`做的只是从已经打开的文件读入数据。
- 输入和划分操作纠缠在一起：每个调用读入一行并把它切分为一些域，不管实际应用中是否真的需要后一个服务。
- 函数返回值表示一个输入行中的数据域个数，每行都被切分，以便得到这个数值。这里也没有办法把出现错误和文件结束区分开。
- 除了更改代码外，没有任何办法来改变这些特性。

以上所列的并不完全，它说明了许多可能的设计选择，各个决定都已经被编织到代码里。对于一个急迫的工作，这样做还说得过去，例如要剖析从一个已知信息源来的固定格式的东西。但是，如果格式可能有变化，例如逗号出现在引号括起的串内，或者服务器产生很长的行或很多的域，那么又会怎么样呢？

这些具体东西看起来都好对付，因为这个“库”很小，而且只是一个原型。但是，如果设想一下，当这个库出台几个月或者几年以后，它不可能变成某些大系统的一部分，而系统的规范又在随着时间的推移不断变化， `csvgetline`能怎样适应这些情况吗？如果这个程序

China-pub.com 下载

是提供给别人用的，在原始设计中的这些仓促选择引起的麻烦就可能到许多年后才浮现出来。这正是许多不良界面的历史画卷。事实确实非常令人沮丧，许多仓促而就的肮脏代码最后变成广泛使用的软件，在那里它们仍然是肮脏的，而且常常也达不到它们应有的速度。

4.3 为别人用的库

借助于从原型中学到的东西，我们现在希望能构造出一个值得普遍使用的库。一个最明显的需求是，必须使csvgetline更健壮，使它能够处理很长的行和很多的域，它也必须能更仔细地剖析数据域。

要建立一个其他人能用的界面，我们必须考虑在本章开始处列出的那些问题：界面、信息隐藏、资源管理和错误处理。它们的相互作用对设计有极强的影响。我们把这些问题分割开是有点太随意了，实际上它们是密切相关的。

界面。我们决定提供3个操作：

char *csvgetline(FILE *)：读一个新CSV行

char *csvfield(int n)：返回当前行的第n个数据域

int csvnfield(void)：返回当前行中数据域的个数

函数csvgetline的返回值应该是什么？它应该返回尽可能方便使用的有用信息。正是这种想法导致我们在原型里让它返回域的个数。而如果要这样做，就必须先计算出域的个数，即使还没有用它们。另一个可能性是令函数返回行的长度，但这又牵涉到是否把换行符计算在内的问题。经过一些试验，我们决定让 csvgetline返回指向输入行本身的指针。如果遇到文件结束就返回NULL。

我们令csvgetline在返回输入行前去掉最后的换行符，因为在必要时很容易恢复。

域的定义是很复杂的，我们试图使该定义符合见到过的各种实际的电子表格或者其他程序。一个域是0个或多个字符的序列。域由逗号分隔，开头和尾随的空格应该保留。一个域可能由一对双引号括起，在这种情况下它还可能包含逗号。一个引号括起来的数据域里可能包含双引号本身，这需要用连续两个双引号表示，例如 CSV域 "x" "y" 定义的实际上是串x" y。域可以为空，两个连续的引号 "" 表示空，连续出现的逗号也表示有一个空域。

域从 0开始编号。如果用户要求一个不存在的域，例如 csvfield(-1)或者csvfield(100000)，我们又该怎么办？我们可以返回 "" (一个空串)，因为这样可以做打印或比较。这里的程序要准备处理数目不定的域，不应该让它们再肩负处理根本不存在的东西的特殊使命。不过这种选择无法区分空域和不存在。另外两种选择是输出错误消息，或者是终止执行。我们很快就会说明为什么这些方式也很不理想。最后，我们决定让函数返回NULL，这是C中表示不存在的常用方式。

信息隐藏。这个库应该对输入行长或域的个数没有限制。为了达到这个目的，或者是让调用程序为它提供存储，或者是被调用者 (库)自己需要做分配。在调用库函数 fgets时，传给它一个数组和一个最大长度，如果输入行比缓冲区长，那么就将它切分成片段。这种方式对CSV界面不太合适，我们的库在发现需要更多存储时应该能自动做存储分配。

这样就只有csvgetline知道存储管理情况，外边程序对其存储组织方式完全不能触及。提供这种隔离的最好途径是通过函数界面： csvgetline读入一行，无论它有多大；csvfiled(n)返回到当前行的第n个域那些字节的指针；csvnfield返回当前行中域的个数。

China-pub.com
下载

当程序遇到更长的行或者更多的域时，它使用的存储就必须增加。我们把如何做这件事的细节都隐藏在csv函数中，其他程序部分完全不知晓这方面的情况。例如，库函数是先使用小数组而后再扩大？还是使用了一个非常大的数组？或者用的是某种与此完全不同的办法？从界面上也看不到存储在何时释放。

如果用户只调用了csvgetline，那么就没有必要做域的切分，这件事可以随后根据命令再做。关于这种切分是立即做 (读到行的时候立刻就做)，或是延后再做(直到有了对数据域或者域个数的要求时再做)，还是以更延后的方式做 (只有需要用的域才做切分)，这是另一个实现细节，对用户也是隐藏的。

资源管理。必须确定谁负责共享的信息。函数csvgetline是直接返回原始数据，还是做一个拷贝？我们确定的方式是：csvgetline的返回值是到原始输入的一个指针，在读下一行时这里将被复写。数据域将在输入行的一个拷贝上构造，csvfield返回指向这个拷贝里的域指针。按照这种安排，如果需要保存或修改某个特殊的行或者域，用户就必须自己做一个拷贝。当这种拷贝不再需要时，释放存储也是用户的责任。

谁来打开和关闭文件？做文件打开的部分也应负责关闭：互相匹配的操作应该在同一个层次或位置完成。我们将假定csvgetline调用时得到的是一个打开了的文件的FILE指针，它的调用者也要负责在处理完毕后关闭文件。

在一个库与它的调用程序之间共享的、或者传递通过它们之间的界限的资源，管理起来总是很困难的，这里经常出现一些合理而又互相矛盾的道理，要求我们做这种或者那种选择。在共享资源的责任方面常常出现错误或误解，这是程序错误最常见的根源之一。

错误处理。由于确定了csvgetline返回NULL，在这里就没有办法区分文件结束和真正的错误，例如存储耗尽等情况。类似地，用户访问不存在的域也不产生错误。这里我们也可以参考ferror的方法，给界面增加一个csvgeterror函数，它报告最近发生的一个错误。为了简单起见，这个版本里将不包括这种功能。

这里有一个基本原则：在错误发生的时候，库函数绝不能简单地死掉，而是应该把错误状态返回给调用程序，以便那里能采取适当的措施。另一方面，库函数也不应该输出错误信息，或者弹出一个会话框，因为这个程序将来可能运行在某种环境里，在那里这种信息可能干扰其他东西。错误处理本身就是一个值得仔细研究的题目，本章后面还有进一步的讨论。

规范。把上面做出的这些决定汇集到一起，就形成一个规范，它说明了csvgetline能提供什么服务，应该如何使用等等。对于一个大项目，规范是在实现之前做出来的，因为写规范和做实现的通常是不同的人，他们甚至来自不同的机构。但是，在实践中这两件事常常是一起做的，规范和实现一起发展。甚至在有些时候，“规范”不过就是为了写明已经做好的代码大概是干了些什么。

最好的方式当然是及早写出规范，而后，随着从正在进行的实现中学到的新情况，对规范进行必要的修改。规范写得越精确越细心，程序工作得很好的可能性也就越大。即使对于个人使用的程序，先准备一个具有合理完整性的规范也是很有价值的，因为它促使人们考虑各种可能性，并记录自己做过的选择。

就我们的目的而言，有关规范应该包含函数的原型、函数行为的细节描述、各种责任和假设等：

域由逗号分隔。

China-pub.com 下载

一个域可能由一对双引号"..."括起。
一个括起的域中可以包含逗号，但不能有换行。
一个括起的域中可以包含双引号 " 本身，表示为 ""。
域可以为空； "" 和一个空串都表示空的域。
引导的和尾随的空格将预留。

```
char *csvgetline(FILE *f);
```

从打开的输入文件f读入一行；
假定输入行的结束标志是\r、\n、\r\n、或EOF。
返回指向行的指针，行中结束符去掉；如果遇到 EOF则返回NULL[⊖]。
行可以任意长，如果超出存储的限度也返回 NULL。
行必须当作只读存储看待；
如果需要保存或修改，调用者必须自己建立拷贝。

```
char *csvfield(int n);
```

域从0开始编号。
返回由csvgetline最近读入的行的第n个域；
如果n<0或超出最后一个域，则返回NULL。
域由逗号分隔。
域可以用 "..." 括起来，这些引号将被去除；
在 "..." 内部的 " " 用"取代，内部的逗号不再作为分隔符。
在没有引号括起来的域里，引号当作普通字符。
允许有任意个数和任意长度的域；
如果超出存储的限度，返回NULL。
域必须当作只读存储看待；
如果需要保存或修改，调用者必须自己建立拷贝。
在调用csvgetline之前调用本函数，其行为没有定义。

```
int csvnfield(void);
```

返回由csvgetline最近读入的行的长度。
在调用csvgetline之前调用本函数，其行为没有定义。

这个规范还是遗留下一些未尽的问题。例如，如果 csvfield或csvnfiled在csvgetline遇到EOF之后被调用，它们的返回值是什么？具有错误形式的域将如何处理？要想把所有这些难题都解决，即使对一个很小的系统也是很困难的；对大系统而言那就是真正的挑战，当然我们还是应该试一试。通常的情况就是这样，直到实现已经在进行时，还可能发现有些遗漏的东西。

这节剩下的部分就是csvgetline的一个符合上面的规范的新实现。我们把这个库分成两个文件，头文件csv.h包含了函数声明，表示的是界面的公共部分；实现文件 csv.c是程序代码。使用者应该在他们的代码中包括这里的 csv.h，并把csv.c编译后的代码连接到他们的代码上。源代码从来都不必是可见的。

⊖ 这个说法应该准确理解。作者的意思是“直接遇到 EOF时返回NULL”。遇到一个行后跟EOF(行可能由EOF结束)时，仍然正常返回这个行。按照C语言的规定，随后再读输入将直接得到EOF。——译者

China-pub.com
下载

这里是头文件：

```
/* csv.h: interface for csv library */

extern char *csvgetline(FILE *f); /* read next input line */
extern char *csvfield(int n);     /* return field n */
extern int csvnfield(void);       /* return number of fields */
```

用于存储正文行的内部变量，以及 split等内部函数都被声明为 static，这样就使它们只在自己的定义文件里是可见的。这是C语言里最简单的信息隐蔽方法。

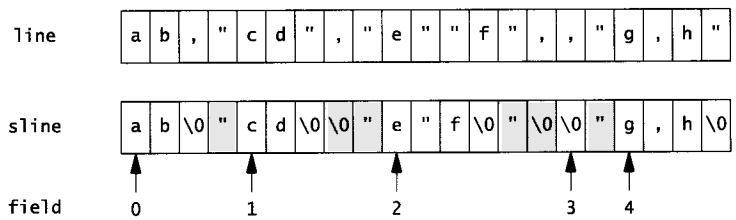
```
enum { NOMEM = -2 };          /* out of memory signal */

static char *line     = NULL;  /* input chars */
static char *sline    = NULL;  /* line copy used by split */
static int  maxline   = 0;     /* size of line[] and sline[] */
static char **field   = NULL;  /* field pointers */
static int  maxfield = 0;      /* size of field[] */
static int  nfield    = 0;     /* number of fields in field[] */

static char fieldsep[] = ","; /* field separator chars */
```

所有这些变量都静态地进行初始化，这些初始值将被用来检测是否需要建立或增大数组。

上述声明描述了一种很简单的数据结构。数组 line存放输入行，sline用于存放由line复制而来的字符行，并用于给每个域添加结束符号。数组 field指向sline的各个项。下面的图显示出这三个数组的状态，表示在输入行 ab,“cd”,“e”“f”,,“g,h”被处理完之后的情况。加阴影的字符不属于任何一个域。



这里是csvgetline的定义：

```
/* csvgetline:  get one line, grow as needed */
/* sample input: "LU",86.25,"11/4/1998","2:19PM",+4.0625 */
char *csvgetline(FILE *fin)
{
    int i, c;
    char *newl, *news;

    if (line == NULL) {          /* allocate on first call */
        maxline = maxfield = 1;
        line = (char *) malloc(maxline);
        sline = (char *) malloc(maxline);
        field = (char **) malloc(maxfield*sizeof(field[0]));
        if (line == NULL || sline == NULL || field == NULL) {
            reset();
            return NULL;         /* out of memory */
        }
    }
    for (i=0; (c=getc(fin))!=EOF && !endofline(fin,c); i++) {
        if (i >= maxline-1) {    /* grow line */
            maxline *= 2;        /* double current size */
            newl = (char *) realloc(line, maxline);
```

China-pub.com 下载

```
            news = (char *) realloc(sline, maxline);
            if (newl == NULL || news == NULL) {
                reset();
                return NULL;    /* out of memory */
            }
            line = newl;
            sline = news;
        }
        line[i] = c;
    }
    line[i] = '\0';
    if (split() == NOMEM) {
        reset();
        return NULL;            /* out of memory */
    }
    return (c == EOF && i == 0) ? NULL : line;
}
```

一个输入行被积累存入line，必要时将调用realloc使有关数组增大，每次增大一倍，就像第2.6节中所讲的那样。在这里还需要保持数组sline与line的大小相同。csvgetline调用split，在数组field里建立域的指针，如果需要的话，这个数组也将自动增大。

按照我们的习惯，开始时总把数组定义得很小，当需要时再让它们增大，这种方法能保证增大数组的代码总会执行到。如果分配失败，我们就调用reset把所有全局变量恢复到开始的状态，以使随后对csvgetline的调用还有成功的机会。

```
/* reset: set variables back to starting values */
static void reset(void)
{
    free(line); /* free(NULL) permitted by ANSI C */
    free(sline);
    free(field);
    line = NULL;
    sline = NULL;
    field = NULL;
    maxline = maxfield = nfield = 0;
}
```

函数endofline处理输入行的各种可能结束情况，包括回车、换行或两者同时出现，或者EOF：

```
/* endofline: check for and consume \r, \n, \r\n, or EOF */
static int endofline(FILE *fin, int c)
{
    int eol;

    eol = (c=='\r' || c=='\n');
    if (c == '\r') {
        c = getc(fin);
        if (c != '\n' && c != EOF)
            ungetc(c, fin); /* read too far; put c back */
    }
    return eol;
}
```

在这里使用一个独立函数是很有必要的，因为标准输入函数不会处理实际输入中可能遇到的各种各样乖张古怪的格式。

前面的原型里用strtok找下一标识符，其方法是查找一个分隔符，一般是个逗号。可是

China-pub.com
下载

这种做法无法处理引号内部的逗号。在 split的实现里必须反映这个重要变化，虽然它的界面可以和strtok相同。考虑下面的输入行：

```
"",,""
,"",
,,
```

这里每行都包含三个空的域。为了保证 split能剖析这些输入以及其他的罕见输入，这个函数将会变得更复杂一些。这又是一个例子，说明一些特殊情况和边界条件会对程序起到某种支配作用。

```
/* split: split line into fields */
static int split(void)
{
    char *p, **newf;
    char *sepp; /* pointer to temporary separator character */
    int sepc;   /* temporary separator character */

    nfield = 0;
    if (line[0] == '\0')
        return 0;
    strcpy(sline, line);
    p = sline;

    do {
        if (nfield >= maxfield) {
            maxfield *= 2;          /* double current size */
            newf = (char **) realloc(field,
                        maxfield * sizeof(field[0]));
            if (newf == NULL)
                return NOMEM;
            field = newf;
        }
        if (*p == '"')
            sepp = advquoted(++p);  /* skip initial quote */
        else
            sepp = p + strcspn(p, fieldsep);
        sepc = sepp[0];
        sepp[0] = '\0';             /* terminate field */
        field[nfield++] = p;
        p = sepp + 1;
    } while (sepc == ',');

    return nfield;
}
```

在循环中，数组可能根据需要增大，随后它调用一个或两个函数，对下一个域进行定位和处理。如果一个域由引号开头，advquoted将找出这个域，并返回指向这个域后面的分隔符的指针值。如果域不是由引号开头，程序就调用标准库函数 strcspn(p,s)，找下一个逗号，该函数的功能是在串p里查找s中任意字符的下一次出现，返回查找中跳过的字符个数。

数据域里的引号由两个连续的引号表示，advquoted需要把它们缩成一个，它还将删除包围着数据域的那一对引号。由于要考虑处理某些不符合我们规范的可能输入，例如"abc"def，这又会给程序增加一些复杂性。对于这种情况，我们把所有跟在第二个引号后面的东西附在已有内容后面，把直到下一分隔符的所有东西作为这个域的内容。 Microsoft的

China-pub.com 下载

Excel使用的看来是类似算法。

```
/* advquoted: quoted field; return pointer to next separator */
static char *advquoted(char *p)
{
    int i, j;

    for (i = j = 0; p[j] != '\0'; i++, j++) {
        if (p[j] == '"' && p[++j] != '"') {
            /* copy up to next separator or \0 */
            int k = strcspn(p+j, fieldsep);
            memmove(p+i, p+j, k);
            i += k;
            j += k;
            break;
        }
        p[i] = p[j];
    }
    p[i] = '\0';
    return p + j;
}
```

由于输入行都已经切分好，csvfield和csvnfield的实现非常简单：

```
/* csvfield:  return pointer to n-th field */
char *csvfield(int n)
{
    if (n < 0 || n >= nfield)
        return NULL;
    return field[n];
}

/* csvnfield:  return number of fields */
int csvnfield(void)
{
    return nfield;
}
```

最后，我们可以对原有测试驱动程序稍微做点修改，再用它来检测这个新版本的库。由于新库保留了输入行的副本(前面原型没保留)，在这里可以先打印出原来的行，然后再打印各个域：

```
/* csvtest main: test CSV library */
int main(void)
{
    int i;
    char *line;

    while ((line = csvgetline(stdin)) != NULL) {
        printf("line = '%s'\n", line);
        for (i = 0; i < csvnfield(); i++)
            printf("field[%d] = '%s'\n", i, csvfield(i));
    }
    return 0;
}
```

这就完成了库的C语言版本，它能够处理任意长的输入，对某些格式乖张的数据也能够做出合理处理。其中也付出了一些代价：新库的大小超过第一个原型的四倍，而且包含一些很复杂的代码。在从原型转换到产品时，程序在大小和复杂性方面有所扩张是很典型的情况。

练习4-1 在数据域切分方面有多种延后再做的可能性。其中包括：一次完成所有切分，但是只在要求做的时候才做；只切分所要求的域；一直切分到所要求的域；等等。列举各种可能

China-pub.com
下载

性，评价不同做法的好处和潜在困难。然后把它们写出来，并测试其速度。

练习4-2 加上一个功能，使分隔符可以改变为：(a) 任意的字符类；(b) 不同域可以有不同分隔符；(c) 正则表达式(见第9章)。这时库的界面应该是什么样的？

练习4-3 在这里我们以C提供的静态初始化机制为基础，实现了一个仅动作一次的开关：当一个指针是NULL时就执行初始化。另一种可能性是要求用户调用一个显式的初始化函数，其中还可以包括建议数组初始大小的参数等。设法实现这个库的一个新版本，使之能同时具有两种方式的优点。在新版本中reset将起什么作用？

练习4-4 设计和实现一个建立CSV格式数据的库。最简单的方式可能是以一个字符串数组为参数，加上引号和逗号打印出来。复杂点的方式可以采用类似 printf的格式串，如果想这样做，可以参看第9章关于记法的一些建议。

4.4 C++ 实现

本节打算写一个C++ 版本的CSV库，它应该能克服C版本里遗留下的某些限制。为此我们需要对原规范做一些改变，其中最重要的是把函数处理的对象由 C语言的字符数组变成 C++的字符串。使用 C++的字符串功能将自动解决许多存储管理方面的问题，因为有关的库函数能帮我们管理存储。由于可以让域函数返回字符串，这就允许调用程序直接修改它们，比原来的版本更加灵活。

类Csv定义了库的公共界面，它明确地隐藏了实现中使用的一些变量和函数。由于在一个类对象里包含了所有的状态信息，建立多个 Csv的实例也不会有问题，不同实例之间是相互独立的，这就使我们的程序能够同时处理多个 CSV输入流。

```
class Csv { // read and parse comma-separated values
    // sample input: "LU",86.25,"11/4/1998","2:19PM",+4.0625

  public:
    Csv(istream& fin = cin, string sep = ",") :
        fin(fin), fieldsep(sep) {}

    int getline(string&);
    string getfield(int n);
    int getnfield() const { return nfield; }

  private:
    istream& fin;           // input file pointer
    string line;            // input line
    vector<string> field;   // field strings
    int nfield;             // number of fields
    string fieldsep;        // separator characters

    int split();
    int endofline(char);
    int advplain(const string& line, string& fld, int);
    int advquoted(const string& line, string& fld, int);
};
```

类的建构函数对参数提供了默认定义，按默认方式建立的 Csv对象将直接从标准输入读数据，使用正规的域分隔符。这些都可以显式地通过实际参数重新设定。

为了管理字符串，在这里使用了标准 C++的string和vector类，没有采用C风格的字

China-pub.com 下载

符串。对于string而言，它没有“不存在”这种状态，“空”串只意味着字符串的长度为0。在这里没有NULL的等价物，我们不能用它作为文件的结束信号。所以，Csv::getline中采用的方式是通过一个引用参数返回输入行，用函数返回值处理文件结束和错误报告。

```
// getline: get one line, grow as needed
int Csv::getline(string& str)
{
    char c;

    for (line = ""; fin.get(c) && !endofline(c); )
        line += c;
    split();
    str = line;
    return !fin.eof();
}
```

运算符+=已被重载，其作用是在字符串的后面加一个字符。

函数endofline需要做一点改造。在这里程序也需要一个一个地读入字符，因为不存在能处理输入中各种变化情况的标准输入函数。

```
// endofline: check for and consume \r, \n, \r\n, or EOF
int Csv::endofline(char c)
{
    int eol;

    eol = (c=='\r' || c=='\n');
    if (c == '\r') {
        fin.get(c);
        if (!fin.eof() && c != '\n')
            fin.putback(c); // read too far
    }
    return eol;
}
```

这里是新版本的split：

```
// split: split line into fields
int Csv::split()
{
    string fld;
    int i, j;

    nfield = 0;
    if (line.length() == 0)
        return 0;
    i = 0;

    do {
        if (i < line.length() && line[i] == '"')
            j = advquoted(line, fld, ++i);  // skip quote
        else
            j = advplain(line, fld, i);
        if (nfield >= field.size())
            field.push_back(fld);
        else
            field[nfield] = fld;
        nfield++;
        i = j + 1;
    } while (j < line.length());
```

China-pub.com
下载


```
    return nfield;
}
```

由于对 C++ 的串不能用 strcspn，我们必须改造 split和advquoted。新的advquoted用C++标准函数find_first_of确定分隔符的下一个出现位置。函数调用s.find_first_of(fieldsep,j)由字符串s的第j个位置开始查找，检查fieldsep里任何字符的第一个出现。如果无法找到这种位置，该函数将返回串尾后面一个位置的指标，在最后还必须把它改回范围之内。随后的一个内层 for循环把字符逐个附到在fld里积累的域后面，直到分隔符处为止。

```
// advquoted: quoted field; return index of next separator
int Csv::advquoted(const string& s, string& fld, int i)
{
    int j;

    fld = "";
    for (j = i; j < s.length(); j++) {
        if (s[j] == '"' && s[++j] != '"') {
            int k = s.find_first_of(fieldsep, j);
            if (k > s.length()) // no separator found
                k = s.length();
            for (k -= j; k-- > 0; )
                fld += s[j++];
            break;
        }
        fld += s[j];
    }
    return j;
}
```

函数find_first_of也被用在新的advplain函数里，这个函数扫过一个简单的无引号数据域。之所以需要做这种改动，其原因和前面一样，因为 strcspn无法用在C++串上，这是一种完全不同的数据类型。

```
// advplain: unquoted field; return index of next separator
int Csv::advplain(const string& s, string& fld, int i)
{
    int j;

    j = s.find_first_of(fieldsep, i); // look for separator
    if (j > s.length())               // none found
        j = s.length();
    fld = string(s, i, j-i);
    return j;
}
```

函数Csv::getfield仍然非常简单，它可以直接写在类的定义里。

```
// getfield: return n-th field
string Csv::getfield(int n)
{
    if (n < 0 || n >= nfield)
        return "";
    else
        return field[n];
}
```

现在的测试程序和前面的差不多，只有很少的改动：

China-pub.com 下载

```
// Csvtest main: test Csv class
int main(void)
{
    string line;
    Csv csv;
    while (csv.getline(line) != 0) {
        cout << "line = '" << line <<"'\n";
        for (int i = 0; i < csv.getnfield(); i++)
            cout << "field[" << i << "] = '"
                 << csv.getfield(i) << "'\n";
    }
    return 0;
}
```

函数的使用方式与C版本有微小的不同。对一个包括30 000行，每行25个域的大输入文件，根据编译程序的不同，这个C++版本比前面的C版本慢40%到4倍。正如我们对markov程序做比较时看到的，实际上，这种变化情况主要反映出标准库本身的不成熟。C++的源程序大约短20%左右。

练习4-5 改造C++的实现，重载下标操作operator[]，使域访问可以用csv[i]形式完成。

练习4-6 写一个Java版本的CSV库，然后从清晰性、坚固性和速度方面比较这三个实现。

练习4-7 把C++版本的CSV代码重新做成一个STL的迭代器。

练习4-8 C++版本允许建立多个独立的Csv实例，它们可以平行地操作而不会互相干扰。这是把一个对象的所有状态封装起来，允许多次实例化带来的收获。设法修改 C版本以得到同样效果，采用的方法是把全局数据结构都装进一个结构，并用一个显式的 csvnew函数做这种结构分配和初始化。

4.5 界面原则

在前面几节里，我们一直在努力做好界面的各方面细节，使它能成为在提供服务的代码和使用服务的代码间的一条清晰的分界线。界面定义了某个代码体为其用户提供的各种东西，定义了哪些功能或者数据元素可以为程序的其他部分使用。我们的 CSV界面提供了三个函数：读一行、取得一个域以及得到域的个数，这些就是能使用的全部操作。

一个界面要想成功，它就必须特别适合有关的工作 —— 必须简单、通用、规范、其行为可以预料及坚固等等，它还必须能很好地适应用户或者实现方式的变化。好的界面总是遵循着一组原则，这些原则不是互相独立的，互相之间甚至可能并不很协调，但它们能帮助我们刻画那些位于界限两边的两部分软件之间的问题。

隐藏实现细节。对于程序的其他部分而言，界面后面的实现应该是隐藏的，这样才能使它的修改不影响或破坏别的东西。人们用了许多术语来描述这种组织原则：信息隐蔽、封装、抽象和模块化，它们谈论的都是类似的思想。一个界面应该隐藏那些与界面的客户或者用户无关的实现细节。这些看不到的细节可以在不影响客户的情况下做修改。例如对界面进行扩充，提高执行效率，甚至把它的实现完全换掉。

各种各样程序设计语言的基本库是大家熟悉的例子，虽然它们并不是都是设计得很好的。C语言的标准I/O库是其中最好的：几十个函数可以执行打开、关闭文件，对文件做读、写以及其他操作。文件I/O的所有实现细节都隐蔽在一个数据类型 FILE*的后面，该类型的实现细节通常可以看到(因为它们通常直接写在<stdio.h>里面)，但却绝不应该使用。

China-pub.com
下载

如果在头文件里不包含实际结构的声明，而只有结构的名字，这种情况被称作是模糊类型，因为其特性本身无法看到，所有操作都通过指针方式进行，实际的数据对象则潜藏在指针后面。

应该避免全局变量。如果可能，最好是把所有需要引用的数据都通过函数参数传递给函数。

我们强烈反对任何形式的公有可见的数据，因为如果用户同样可以改变这些变量，要维护值的一致性就太困难了。通过函数界面，可以很容易提出要求遵守的访问规则。但是人们常常违反这个设计原则。例如，预定义的 I/O流，如sdtin和stdout，基本上都被定义为一个全局的FILE结构数组的元素：

```
extern FILE    __iob[_NFILE];
#define stdin  (&__iob[0])
#define stdout (&__iob[1])
#define stderr (&__iob[2])
```

这就使功能的实现完全暴露出来。这种方式也意味着人们无法给 sdtin、stdout或stderr重新赋值，虽然它们看起来像变量。这里写的特殊名字 —— __iob使用了ANSI C的惯例：采用由连续两个下划线开头的名字为必须暴露的私有对象命名，这种形式的名字不大可能与程序用的其他名字冲突。

C++和Java的类提供了更好的信息隐藏机制，这也是正确使用这些语言的核心。我们在第3章用过的标准模板库把这种想法更向前推进了一步，除了某些执行性能方面的保证之外，这里没有任何关于实现的信息，而库的构造者则可以使用自己喜欢的任何技术。

选择一小组正交的基本操作。一个界面应该提供外界所需要的全部功能，但是绝不要更多；函数在功能方面不应该有过度的重叠。虽然提供大量函数可能使库变得更容易使用，你需要什么就有什么。但是大的界面既难写又难维护，太大的规模也使它难以学习、难以被用好。许多“应用程序界面”(Application Program Interface)或称API有时是如此的庞大，以至没人能有指望完全把握它。

为使用方便，有些界面为做同一件事提供了多种方式，这种冗余其实是应该反对的。 C语言标准I/O库提供了至少四个不同函数，它们都能把一个字符写进一个输出流：

```
char c;
putc(c, fp);
fputc(c, fp);
fprintf(fp, "%c", c);
fwrite(&c, sizeof(char), 1, fp);
```

如果输出流是stdout，我们还可以找到另外一些方式。这样做确实很方便，但是却完全没有必要。

一般地说，窄的界面比宽的界面更受人欢迎，至少是在有了强有力的证据，说明确实需要给界面增加一些新功能之前。我们应该做一件事并且把它做好。不要因为一个界面可能做某些事就给它增加这些东西。如果是实现方面出了毛病那么就不要去修改界面。再比如，我们不应该为了速度就使用 memcpy，为了安全又去用memmove，最好是始终用一个总是安全的，而又比较快的函数。

不要在用户背后做小动作。一个库函数不应该写某个秘密文件、修改某个秘密变量，或者改变某些全局性数据，在改变其调用者的数据时也要特别谨慎。函数 strtok在这方面有几个毛

China-pub.com 下载

病：它在输入串里面写进空字节，这常常使人感到一点意外；它用空指针作为信号，回到上次离开的位置，这意味着它在不同调用之间保留了一些秘密数据，这个情况很可能成为程序错误的原因，也排除了并行使用这个函数的可能性。一种更好的设计是提供一个独立函数从输入字符串提取标识符。由于类似的原因，我们库的第二个 C版本也不能同时用于两个流 (参看练习4-8)。

一个界面在使用时不应该强求另外的东西，如果这样做仅仅是为了设计者或实现者的某些方便。相反，我们应该使界面成为自给自足的。如果确实无法做到这一点，那么也应该把需要哪些外部服务的问题做成明显的。不这样做就会给用户增加很大的负担。这方面有一个非常明显的例子，那就是在 C和C++ 源程序中经常需要费力地管理长长的头文件列表。有些头文件可能有数千行长，它还可能包含几十个其他头文件。

在各处都用同样方式做同样的事。一致性和规范性是非常重要的。相关的事物应该具有相关的意义。例如，C函数库里的`str...`函数没有文档也很容易使用，因为它们的行为具有一致性：数据总是从右向左流动，与赋值语句的流向相同；它们总是返回结果串。而另一方面，在C标准I/O库里，要预期函数的参数顺序就非常困难。有些函数把 `FILE*`作为第一个参数，有些则放在最后；另一些函数的元素大小参数和个数参数又有各种不同顺序。 STL容器类的算法提供了非常一致的界面，这样，即使面对一个不熟悉的函数，预计应该如何使用它也会变得很容易。

外部一致性，与其他东西的行为类似也是非常重要的。例如， C函数库里的`mem...`函数是在`str...`函数之后设计的，完全借用了它们前驱的风格。如果标准 I/O函数`fread`和`fwrite`看起来更像`read`和`write`(这是它们的由来)一些，它们也会更容易记忆。 Unix系统的命令行参数都由一个负号引导，但是同一个选项字母可能意味着完全不同的东西，甚至对一些相互有关的程序也常常是这样。

如果*这样的通配符(例如在`*.exe`里)都由命令解释器展开，其行为是一致的。但如果它们是由独立程序展开的，多半就会出现不一致的行为。 Web浏览器对一次鼠标点击的反应是跟踪一个链接，有些应用程序遇到两次鼠标点击才执行一个程序或跟踪一个链接。这种情况造成的后果是许多人每次都自动地点击两下。

要遵循这些原则，在有些环境里常常比在另一些环境里更容易。但是，无论如何，这些原则都是有效的。比如，在C语言里要隐藏实现细节就非常困难，但是，好的程序员就不会去探察这种细节，因为这种做法将牵连上界面的某些细节部分，违反了信息隐蔽原则。头文件里的注释，特殊形式的名字 (例如`__iob`)，以及其他一些类似东西，都是为了在不能强制规定的情况下，鼓励好行为而采用的方式。

当然，不管我们今天在设计界面时做得如何好，事情也总有一个限度。今天最好的界面最终也会变成明天的问题。但是，一个好的界面设计将把这个明天推到更远的将来。

4.6 资源管理

在设计库(或者类、包)的界面时，一个最困难的问题就是管理某些资源，这些资源是库所拥有的，而又在库和它的调用程序之间共享。这里最明显的资源是存储 —— 谁负责存储的分配和释放？其他的共享资源包括那些打开的文件以及共同关心的变量状态等。粗略地说，有关的问题大致涉及初始化、状态维护、共享和复制以及清除等等。

China-pub.com
下载

在我们的CSV原型中，采用静态初始化方式为各种指针、计数器等设置初始值。这种方式的功能很有限，因为一旦函数被调用过之后，就无法使这部分程序再从它们的初始状态重新开始。另一种方式是提供一个初始化函数，为所有内部变量正确设置初始值。这种方式能允许重新启动，但要依靠用户显式地对这个函数做调用。我们可以把第二个版本里的 reset 函数提供出来，服务于这个目的。

在C++和Java语言里，建构函数专用于给类的数据成分做初始化，设计正确的建构函数能保证所有数据成员都做了初始化，不可能建立未初始化的类对象。一组建构函数能支持各种各样的初始化方式。例如，我们可以为 Csv提供一个以文件名为参数的建构函数，另一个函数则以输入流作为参数，如此等等。

如果需要拷贝由库管理的数据，例如输入行或数据域，那么又会怎么样？在我们 C语言版本的csvgetline程序里，是通过返回指针的方式提供了对输入串 (行和数据域)的直接访问。这种不加限制的访问方式有几个缺点：它使用户有可能复写有关的存储区，以至可能使其他信息变成非法的。例如用户写了下面的表达式：

```
strcpy(csvfield(1), csvfield(2));
```

就可能造成多种不同后果。最可能的是复写掉第二个数据域的开头 (当第二个域比第一个长的时候)。此外，库用户如果想把任何信息保留到下一次 csvgetline调用之后，就必须自己做一个拷贝。在下面的例子里，有关的指针值在第二次 csvgetline调用后就可能变成非法的，因为这个调用有可能导致行缓冲区存储的重新分配。

```
char *p;

csvgetline(fin);
p = csvfield(1);
csvgetline(fin);
/* p could be invalid here */
```

C++版本要安全得多，由于在那里的串都是拷贝，所以可以随意修改。

Java采用的是对象引用，对所有基本类型 (如int)之外的东西都采用这种做法。这种做法比建立拷贝更有效，但是也很容易使人误解，误把引用当作拷贝。在用 Java写markov程序时，我们在一个早期版本里就犯过这样一个的错误。另外，C语言里与字符串有关的错误层出不穷，其根源也在这个地方。Java语言的克隆方法使人可以在需要时建立拷贝。

初始化和建构的另一面是终止和析构 —— 当某些实体不再需要时，就必须对它们进行清理并回收有关资源。对于存储而言这件事特别重要，因为如果一个程序不能把不再使用的存储收回，最终它就会把存储用光。许多很时髦的软件都存在这种毛病，实在令人羞愧。类似问题还出现在打开的文件需要关闭时，如果数据被做了缓冲，在关闭文件时应该做缓冲区的刷新(以及存储回收)。C语言的标准库函数在程序正常终止时将会自动做刷新，其他情况下我们就需要自己做了。C和C++ 的标准函数atexit提供了一个在程序正常终止前取得控制的方法，界面的实现者可以利用这个机制安排一些清理工作。

释放资源与分配资源应该在同一个层次进行。控制资源分配和回收有一种基本方式，那就是令完成资源分配的同一个库、程序包或界面也负责完成它的释放工作。这种处理原则的另一种说法是：资源的分配状态在跨过界面时不应该改变。例如，我们的 CSV库从一个已经打开的文件读数据，那么它在完成工作时还应该使文件保持在打开状态，由库的调用程序关闭文件。

China-pub.com 下载

C++的建构函数和析构函数对实施这个规则很有帮助。当一个类实例离开了作用域或者被显式地清除时，析构函数就会被自动调用，它可以刷新缓冲区、恢复存储、重新设置值以及完成其他一些应该做的事情。Java没有提供与此等价的机制，虽然可以为类定义一个终结方法，但是却不能保证它一定会执行，更不用说要求在某个特定时刻做了。因此，在这里不能保证清理操作一定会发生，虽然通常人们都合理地假定它们会。

Java也确实对存储管理提供了很大的帮助，它有一个内部的废料收集系统。Java程序在运行时必定要分配存储，但是它没办法显式地释放存储。运行系统跟踪所有在用对象和无用对象，周期性地把不再有用的对象收回到可用存储池里。

废料收集有许多不同的技术。有一类模式是维持着对每个对象的使用次数，即它们的引用计数值，一旦某个对象的引用计数值变成 0时就释放它。这种技术可以显式地应用到 C或C++里，用来管理共享的对象。另一种算法是周期性地工作，按照某种痕迹跟踪分配池到所有被引用的对象。所有能够以这种方式找到的对象都是正在被使用的，未被引用的对象是无用的，可以回收。

自动废料收集机制的存在并不意味着设计中不再有存储管理问题了。我们仍然需要确定界面是应该返回对共享对象的引用呢，还是返回它们的拷贝，而这个决定将影响到整个程序。此外，废料收集也不是白来的 —— 这里有维护信息和释放无用存储的代价，此外，这种收集动作发生的时间也是无法预期的。

如果一个库将被使用在有多个控制线程的环境里，其函数可能同时存在多个调用，例如用在多线程的Java程序里，上述所有问题都将变得更复杂。

为了避免出问题，我们必须把代码写成可重入的，也就是说，无论存在多少个同时的执行，它都应该能正常工作。可重入代码要求避免使用全局变量、静态局部变量以及其他可能在别的线程里改变的变量。对于好的多线程设计，最关键的就是做好各部件之间的隔离，使它们除了经过良好设计的界面外不再共享任何东西。那些随意地把变量暴露出来共享的库就破坏了这个模型(对于多线程的程序而言，`strtok`纯粹是个灾星，C函数库里其他采用在内部静态变量里存储值的函数也一样)。如果存在必需的共享变量，它们就必须放在某种锁定机制的保护之下，保证在每个时刻只有一个线程访问它们。类在这里有极大的帮助作用，因为它们能够成为共享和锁定模型的中心。Java的同步方法就是这类东西，它使线程可以锁定整个的类或者类的实例，防止其他线程同时进行修改。同步程序块在每个时刻只允许一个线程执行这个代码段。

多线程给程序设计增加了许多的新问题，这是一个太大的题目，我们无法在这里讨论更多的细节。

4.7 终止、重试或失败

在前面的章节里，我们用到了`eprintf`和`estrdup`等函数，用于处理结束程序执行之前的错误信息显示问题。例如，`eprintf`的功能与`fprintf(stderr, .`类似，它在出错状态下报告了错误后就结束程序。定义这个函数需要用 `<stdarg.h>`头文件，通过库函数`vprintf`打印原型中`...`表示的那些参数。使用 `stdarg`标准库的功能，必须先调用`va_start`做初始化，最后调用`va_end`做终止处理。在第9章里我们还要更多地使用这个界面。

China-pub.com
下载


```
#include <stdarg.h>
#include <string.h>
#include <errno.h>

/* eprintf: print error message and exit */
void eprintf(char *fmt, ...)
{
    va_list args;

    fflush(stdout);
    if (progname() != NULL)
        fprintf(stderr, "%s: ", progname());

    va_start(args, fmt);
    vfprintf(stderr, fmt, args);
    va_end(args);

    if (fmt[0] != '\0' && fmt[strlen(fmt)-1] == ':')
        fprintf(stderr, " %s", strerror(errno));
    fprintf(stderr, "\n");
    exit(2); /* conventional value for failed execution */
}
```

如果格式参数由一个冒号结束，eprintf就调用标准C函数strerror，该函数返回一个附加的系统错误信息串(如果存在的话)。我们也定义了weprintf，它的功能与eprintf类似，显示一个警告信息，但却不结束程序的执行。利用这种与printf类似的界面构造字符串非常方便，然后可以输出或者显示在对话框里。

与此类似，函数estrdup试图构造串的拷贝，如果存储用光了，它就通过eprintf输出一个错误信息并结束程序：

```
/* estrdup: duplicate a string, report if error */
char *estrdup(char *s)
{
    char *t;

    t = (char *) malloc(strlen(s)+1);
    if (t == NULL)
        eprintf("estrdup(\"%.20s\") failed:", s);
    strcpy(t, s);
    return t;
}
```

函数emalloc提供与调用malloc时类似的服务：

```
/* emalloc: malloc and report if error */
void *emalloc(size_t n)
{
    void *p;

    p = malloc(n);
    if (p == NULL)
        eprintf("malloc of %u bytes failed:", n);
    return p;
}
```

头文件eprintf.h里声明了这些函数：

```
/* eprintf.h: error wrapper functions */
extern  void    eprintf(char *, ...);
extern  void    weprintf(char *, ...);
extern  char    *estrdup(char *);
```

China-pub.com 下载

```
extern  void    *emalloc(size_t);
extern  void    *erealloc(void *, size_t);
extern  char    *progname(void);
extern  void    setprogname(char *);
```

这个头文件应该被包含到任何需要使用这些错误处理函数的文件里。输出的每个错误信息串中还可以包含一个程序名，这需要在调用程序里事先进行设置。完成这件事的是两个非常简单的函数setprogname和progname。它们也在头文件里声明，与eprintf在同一个源文件里定义：

```
static char *name = NULL;  /* program name for messages */

/* setprogname: set stored name of program */
void setprogname(char *str)
{
    name = estrdup(str);
}

/* progname: return stored name of program */
char *progname(void)
{
    return name;
}
```

这些函数的典型使用方式是：

```
int main(int argc, char *argv[])
{
    setprogname("markov");
    ...
    f = fopen(argv[i], "r");
    if (f == NULL)
        eprintf("can't open %s:", argv[i]);
    ...
}
```

输出的信息是这样的：

```
markov: can't open psalm.txt: No such file or directory
```

我们觉得这些包装函数给程序设计带来很大方便，因为它们能使错误得到统一处理，它们的存在也促使我们尽量去捕捉错误而不是简单地忽略错误。在我们的这种设计里并没有什么特殊的东西，但你在写自己的程序时也可能喜欢采用其他方式。

假定我们写的函数不是为了自己用，而是为别人写程序提供一个库。那么，如果库里的一个函数发现了某种不可恢复性的错误，它又该怎么办呢？在本章前面写的函数里，采取的做法是显示一段信息并令程序结束。这种方式对很多程序是可以接受的，特别是对那些小的独立工具或应用程序。而对另一些程序，终止就可能是错误的，因为这将完全排除程序其他部分进行恢复的可能性。例如，一个字处理系统必须由错误中恢复出来，这样它才能不丢掉你键入的文档内容。在某些情况下库函数甚至不应该显示信息，因为本程序段可能运行在某种特定的环境里，在这里显示信息有可能干扰其他显示数据，这种信息也可能被直接丢掉，没留下任何痕迹。在这种环境里，一种常用的方式是输出诊断情况，写入一个显式的记录文件，这个文件可以独立地进行监控和检查。

在低层检查错误，在高层处理它们。作为一条具有普遍意义的规则，错误应该在尽可能低的层次上检测和发现，但应该在某个高一些的层次上处理。一般情况下，应该由调用程序决定

China-pub.com
下载

对错误的处理方式，而不该由被调用程序决定。库函数应该以某种得体的失败方式在这方面起作用。在前面我们让函数对不存在的域返回NULL值，而不是立即退出执行，也就是由于这个原因。与此类似，csvgetline在遇到文件结束之后，无论再被调用多少次，也总是返回NULL值。

合适的返回值未必都是非常明显的，我们在前面讨论csvgetline应该返回什么时就遇到了这种情况。我们总希望能尽量返回某些有用信息，而且应该是以程序其他部分最容易使用的形式。在C、C++和Java里，这就意味着要求以函数值的方式返回某种东西，与此同时，其他的值就只能通过引用(或者指针)参数返回。许多库函数都区分了正常的值和错误值。像getchar一类的输入函数对正常数据返回一个char数据，而对错误或者文件结束则返回某种非char值，例如EOF等。

如果函数的合法返回值能取遍所有可能的返回值，上面这种方式就行不通了。例如一个数学函数，如log，它本来就可能返回任何浮点数值。在IEEE标准的浮点数里，有一个特殊值称为NaN(“不是数”，not a number)，就是专用于指明错误的，可以作为返回错误信号的值。

有些语言，例如Perl和Tcl等，提供了把两个或多个值组合起来形成元组的简便方法。对这类语言，我们很容易把函数值和可能的错误状态信息一起返回。C++的STL提供了一种pair数据类型，也可以用在这个地方。

如果能把各种各样的异常值(如文件结束、可能的错误状态)进一步区分开，而不是用单个返回值把它们堆在一起，那当然就更好了。如果无法立刻区分出这些值，另一个可能的方法是返回一个单一的“异常”值，但是另外提供一个函数，它能返回关于最近出现的错误的更详细信息。

这也是在Unix和C标准库里采用的方法。在那里许多系统调用或库函数都在返回-1(表示错误)的同时给名为errno的变量设一个针对特定错误的编码值，随后用strerror就可以返回与各个错误编码关联的字符串。在我们使用的系统里，程序：

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <math.h>

/* errno main: test errno */
int main(void)
{
    double f;

    errno = 0;  /* clear error state */
    f = log(-1.23);
    printf("%f %d %s\n", f, errno, strerror(errno));
    return 0;
}
```

输出：

```
nan0x10000000 33 Domain error
```

如上所示，errno必须事先予以清除。此后，如果发生错误，errno就会被置上非0值。

只把异常用在异常的情况。有些语言提供了异常机制，以帮助捕捉不正常状态并设法做恢复，这种机制实际上是提供了在某些坏事情发生时的另一条控制流。异常机制不应该用于处理可

China-pub.com 下载

预期的返回值。读一个文件最终总要遇到文件结束，这个情况就应该以返回值的方式处理，而不是通过异常机制。

在Java里可以写：

```
String fname = "someFileName";
try {
    FileInputStream in = new FileInputStream(fname);
    int c;
    while ((c = in.read()) != -1)
        System.out.print((char) c);
    in.close();
} catch (FileNotFoundException e) {
    System.err.println(fname + " not found");
} catch (IOException e) {
    System.err.println("IOException: " + e);
    e.printStackTrace();
}
```

这里的循环读字符，一直读到文件结束。这是一个可预期的事件，由 read的返回值为-1表示。如果文件打不开，就会引发一个异常，而不是把输入流设置为 null，后者是C语言或C++ 中的常用做法。如果在try块执行中出现了其他I/O错误，也会引发异常，这个异常将被最下面的IOException处理段捕获。

异常机制常常被人过度使用。由于异常是对控制流的一种旁路，它们可能使结构变得非常复杂，以至成为错误的根源。文件无法打开很难说是什么异常，在这种情况下产生一个异常有点过分。异常最好是保留给那些真正无法预期的事件，例如文件系统满或者浮点错误等等。

对于C程序，存在着一对库函数setjmp和longjmp，它们提供了一种很低层的服务，借助它们可以实现一套异常处理机制。不过这些机制太神秘，我们不打算再进一步讨论它们。

在发生错误时应该如何恢复有关的资源？如果发生了错误，库函数应该设法做这种恢复吗？通常它们不做这些事，但也可以在这方面提供一些帮助：提供尽可能清楚的信息和以尽可能无害的方式退出。当然，不再使用的存储应该释放，如果有些变量还可能被使用，那么就应该把它们设置成某种明显的值。有一种错误很常见，那就是通过指针使用那些已经释放的存储。如果错误处理代码在释放存储之后把指向它们的指针都置为空，这种错误就一定能发现。CSV库的第二个版本里的reset函数就考虑了这些问题。一般说来，我们的目标应该是保证在发生了错误之后，库仍然是可用的。

4.8 用户界面

至此我们谈论的主要是一个程序里不同部件之间的界面，或者是不同程序之间的界面问题。实际中还有另一类，也是非常重要的界面，那就是程序与作为程序使用者的人之间的界面。

本书里大部分程序例子都是基于文本方式的，因此它们的用户界面非常简单。正如我们在前一节谈到的，应该对各种错误进行检查并报告，还应该在有意义的情况下设法恢复到某种常态。错误信息输出应该包含所有可用信息，在可能情况下应给出有意义的上下文信息。一个诊断信息不应该这样简单：

```
estrdup failed
```

China-pub.com
下载

如果它应该提供的信息是：

```
markov: estrdup("Derrida") failed: Memory limit reached
```

像我们在estrdup里所做的那样并没有多费什么事，但却能帮助用户发现问题，或者告诉他们如何提供合法的输入。

程序在使用方式出现错误时应该显示有关正确方式的信息，就像下面的函数所做的：

```
/* usage: print usage message and exit */
void usage(void)
{
    fprintf(stderr, "usage: %s [-d] [-n nwords]"
        " [-s seed] [files ...]\n", progname());
    exit(2);
}
```

程序名标识了信息的来源，尤其是当它实际上是一个更大过程的一部分时，这个信息更是特别重要。如果一个程序提供的错误信息仅仅是说 syntax error(语法错)或者estrdup failed(estrdup失败)，用户将根本无法知道是谁在说话。

错误信息、提示符或对话框中的文本应该对合法输入给出说明。不要简单地说一个参数太大，而应说明参数值的合法范围。如果可能的话，给出的这段文本本身最好就是一段合法的输入，比如提供一个带合适参数的完整命令行。除了指导用户如何正确使用外，这种输出还应该能导入到一个文件里，或者通过鼠标器选择拷贝，以便用于运行某个另外的程序。由此也可以看到对话框的一个弱点：它们的内容通常是不能使用的。

建立有效用户界面的另一种途径是设计一个为设置参数、控制各种操作等等而用的特殊语言。一种好的记法常常能使程序的使用更加简单，也能够帮人组织程序的实现。以语言为基础的界面问题是第9章要讨论的题目。

防御性程序设计，或者说是保证程序在遇到坏的输入时本身不会受到损害，无论是从防止用户破坏的角度看，还是从安全性的角度看都是非常重要的。第 6章还要更多地讨论这个问题，那里将讨论程序测试问题。

对于大部分人而言，图形用户界面就是他们计算机的用户界面。图形用户界面是一个巨大的题目，对此我们将只想说一点与本书切题的内容。首先，图形界面很难做“正确”，因为它们的适用性或者成功非常强烈地依赖于个人的行为和期望。其次，作为另一个实际情况，如果一个系统有一个用户界面，通常这个程序里处理用户交互的代码将比实现所完成工作的算法的代码更多。

无论如何，我们熟悉的各种原则也适用于用户界面软件的外部设计及其内部实现。从用户的观点看，风格问题，如简单性、清晰性、规范性、统一性、熟悉性和严谨性等，对于保证一个界面容易使用都是非常重要的，不具有这些性质的界面必定是令人讨厌的难对付的界面。

统一和规范是非常必要的，这方面的问题包括术语、单位、格式、排布方式、字体、颜色、大小，以及其他一切图形系统可能选择的方面，在使用上都应该有一致性。对退出一个程序或者关闭一个窗口，有多少可以使用的英文词？从“ Abandon”(放弃)到“control-Z”，这种词至少有一打之多。这种不一致把说英语的人都搞糊涂了，更不用说是其他人了。

对图形程序而言，界面更加重要，因为这些系统都很大，很复杂，采用的是与文本顺序扫描很不相同的方式驱动的。对实现图形用户界面而言，面向对象的程序设计远胜于其他方

China-pub.com 下载

法，因为它提供了一种途径，能够封装各种窗口的行为和状态细节，通过继承来取得基类的相似性，通过导出类区分出相互的差异。

补充阅读

Frederick P. Brooks, Jr. 的《神秘的人月》(The Mythical Man Month，Addison-Wesley，1975，1995的周年版)包含了许多关于软件开发的真知灼见，虽然其中讨论的有些技术细节已经过时了，但在今天它仍然像当年出版的时候一样值得读一读。

对界面设计而言，几乎所有关于程序设计的书都可能有些用处。其中一本根据从艰难工作中的取胜经验写的实践性书籍是 John Lakos的《大规模C++软件设计》(Large-Scale C++ Software Design，Addison-Wesley，1996)，该书讨论如何构建和管理真正大规模的 C++程序。David Hanson的《C界面和实现》(C Interfaces and Implementation，Addison-Wesley，1997)是一本关于C程序的好书。

Steve McConnell的《快速开发》(Rapid Development，Microsoft Press，1996)是一本极好的关于怎样以软件队(组)方式构造软件的书，其中特别强调了原型的作用。

关于图形用户界面设计有许多有趣的书，它们提出了各种各样的看法。我们推荐 Kevin Mullet和Darrell Sano的《设计可视的界面：基于交流的技术》(Designing Visual Interfaces: Communication Oriented Techniques，Prentice Hall，1995)，Ben Shneiderman的《设计用户界面：有效人机交互的策略》(Designing the User Interface: Strategies for Effective Human-Computer Interaction，第3版，Addison-Wesley，1997)，Alan Cooper的《表面：用户界面设计要素》(About Face: The Essentials of User Interface Design，IDG，1995)，以及Harold Thimbleby的《用户界面设计》(User Interface Design，Addison-Wesley，1990)。