第 1 章

试水

你都知道一开始会来什么，那就直接点吧：

```
import std.stdio;

void main()

{

   writeln(“Hello,world!”);

}
```

源于对其它语言的了解，你可能有一种似曾相识的感觉，对其之简洁的一种淡淡的欣赏，又或是对 D 没有走脚本语言允许顶层语句的路线的一点失望。（顶层语句引入的全局变量在程序增长时很快就变不得利了；D 的确也提供了在 main 之外执行代码的选择，但设计为一种更结构化的方式。）如果你是一个非常细致的人，请放心，void main 和执行成功的话向操作系统返回值为“ success ”（代码 0）的 int main 完全一样。

慢着先，别超前。经典的“ Hello,world! ”程序并不是用来讨论一门语言的能力，而是引导你开始用这种语言写程序并运行它。如果你眼下没有一个集成开发环境可视化地编译程序，命令行也很容易。输入上面的代码，保存为 hello.d,打开命令行然后输入以下命令：

```
$  dmd hello.d

$  ./hello

Hello,world!

$ _
```

$ 代表命令提示符（Windows 下应是 C:\Path\To\Dir> ，Unix 下，比如 OSX，Linux，Cygwin 下，应是/path/to/dir%）。如果你用上一点该系统下的小技术，你也可以让这个程序自动编译自动运行，Windows 下，你可能会将命令行的“ 运行 ”命令与 rdmd.exe 关联系起来而达到这个目的，rdmd.exe 就在 D 的安装包里；Unix 族支持“ shebang 标记 ”载入脚本，而 D 也读得懂该标记；将下面这行

```
#!/usr/bin/rdmd
```

添加在 hello.d 第一行就可以让它自动执行。做了上面这个修改后，命令行下你只要简单地敲入：

```
$ chmod u+x hello.d
$ ./hello.d
Hello, world!
$ _
```

(你只需 chomd 一次。)

在所有系统上，rdmd 会将编译的可执行文件缓存起来，这样只有在源程序有改动时才会重新编译，而不是每次运行都要事先编译。这一点再加上编译器本身就超快，促成了 D 修改-运行周期短的特点，这对于写短小的脚本或是大程序都非常有利。

这个程序以下面的指示语句开始：

```
import std.stdio;
```

该指示句的作用是指示编译器查找名叫 std.stdio 的模块文件并使其可用。import 和 C++的预编译指示符#include 相近，但更接近 Python 的 import: 没有文本包含，只导入符号表。重复 import 同一文件不会将该文件导入两次。

继承了伟大的 C 的传统，一个 D 程序由一系列的声明组成， 这些声明可以传播到多个文件当中。它们可以引入如类型 ，函数，数据等很多东西。我们的第一个程序定义了 main 函数，它没有参数，没有返回值--void 就表示没有返回值。程序执行时，main 调用函数 writeln（它在 std.stdio 模块里定义）,并给它传入了一个字符串常量。ln 后缀表示 writeln 在被打印的文本后添加新行（换行）。

接下来的几个部分带领你快速领略 D。小的演示程序可以用来解释基本概念。眼下着重于对 D 语言的领略，而不是学究式定义。后续的章节会针对语言的每一部分进行详细解释。

1.1 数字与表达式

你有没有好奇过老外有多高？我们这就开始写一个程序，显示正常身高范围内英尺+英寸到厘米的转换。

```
/*
计算给定英尺+英寸表示的一定范围内的身高，其厘米表示的身高是多少。
*/
import std.stdio;
void main()
{
    // 不会很快改变的数值
    immutable inchesPerFoot = 12;
    immutable cmPerInch = 2.54;
    // 循环并打印
    foreach (feet; 5 .. 7)
    {
       foreach (inches; 0 inchesPerFoot)
       {
           writeln(feet, "'", inches, "\"\t",
              (feet * inchesPerFoot + inches) * cmPerInch);
       }
    }
}
```

程序执行后，将会打印一个可爱的两栏列表：

```
5'0"    152.4
5'1"   154.94
5'2"   157.48
...
6'10"  208.28
6'11"  210.82
```

foreach(feet;5..7){…} 是一个迭代语句，它定义了一个整型的变量 feet，使其由 5 递增至 6，但不到 7（左闭右开）。

和 Java，C++以及 C#一样，D 同时支持多行注释/*multiline comments */和单行注释//single-line comments（还有文档注释，稍后再讲）。还有一个有意思的地方是我们的小程序引入数据的方式。一开始，有两个常量：

```
immutable inchesPerFoot = 12;
immutable cmPerInch = 2.54;
```

由关键字 immutable 引入的常量其值永不会改变。做为常量，并不需要提供一个明确的类型，其真正的类型可以通过初始化值的类型推断得到。拿这个例子来说，字面量 12 告诉编译器 inchesPerFoot 是一个整型（在 D 里用 int 表示）；类似地，字面量 2.54 使得 cmPerInch 成为一个浮点常量（double 类型）。再往下看，对 feet 和 inches 使了同样的魔术。因为它们看起来完全象变量，但没有明显的类型修饰。但这并没有让程序变得比下面这个不安全：

```
immutable int inchesPerFoot = 12;
immutable double cmPerInch = 2.54;
foreach (int feet; 5 .. 7)
{
    ...
}
```

以此类推，相反却少了冗余。只有在通过上下文推断类型时不产生二歧性的情况下，编译器才允许省略类型声明。现在类型已通过推断得到，让我们先停一下，来看看语言都有哪些数值类型提供。

为了增长大小，有符号整型类型包括 byte，short，int 和 long，其大小分别为 8 位，16 位，32 位和 64 位。它们每一个都有相对应的相同大小的无符号类型，命名很简单：ubyte，ushort，uint 和 ulong（没有 C 的 unsigned 修饰符）。浮点类型包括 float（32 位 IEEE 754 单精度），double（64 位 IEEE 754），和 real（机器浮点寄存器可容纳的最大实数，最小不会小于 64 位；比如，在 Intel 机器上 real 被称为 IEEE 754 双精度扩展 79 位格式）。

现在回到健全的整型王国，象 42 这样的字面量可以赋给任一数字类型，但编译器会检查目标类型是否足够大以能容纳这个值。因此声明

```
immutable byte inchesPerFoot = 12;
```

或者干脆省略掉 byte 都没问题，因为 12 可以很好地存在 8 位或者 32 位当中。默认情况下，数字型如果省略了类型声明（比如上面的例子程序），整型常量拥有 int 类型，浮点常量拥有 double 类型。

有了这些类型，在 D 里利用算术运算符和函数就可以建造很多的表达式。这些运算符和优先级与你所见的 D 的兄弟语言基本一样：+，-，*，/，以及%用来做基本算术操作，==，!=，<，>，<=，>=用来做比较，fun(参数 1，参数 2)用来做函数调用，如此等等。

回到我们的厘米英寸转换程序。writeln 有两点值得一起。一是 writeln 有 5 个参数，与 Pascal 的 writeln，C 的 printf，C++的 cout 相同。D 的 writeln 函数接受一系列参数（称做“ 可变参数函数 ”）。D 允许程序员自定义可变参数函数（不象 Pascal），并且是类型安全的（不象 C），也无需捣弄操作符（不象 C＋＋）。另一个要说的是 writeln 把数据与格式混在一起，有点丑。我们经常期望把数据部分与显示部分分离开来，因此，接下来我们用带格式化的输出函数 writefln 来代替 writeln：

```
writefln("%s'%s"\t%s", feet, inches,
    (feet * inchesPerFoot + inches) * cmPerInch);
```

输出结果完全一样，不同的是 writefln 的第一个参数描述要显示的格式。%为占位符，类似于 C 的 printf，比如，%i 格式化整型，%f 格式化浮点型，而%s 格式化字符串。

如果你用过 C 的 printf，请相信你的疑惑，没错：我们要打印的明明是 int 和 double—占位符怎么是%s，而%s 应该只能格式化字符串才对啊？答案很简单。D 的可变参数的特性让 writefln 能够访问实参的真实类型，然后分两步处理：（1）%s 被扩展定义为“ 实参默认的字符串表示 ”；（2）如果占位符与实参类型不匹配，你会得到一个干脆直接的错误，而不是象 printf 一样在占位符匹配错误后有可能得到莫名其妙的结果（更不要说 printf 因非信任的格式化字符串导致的安全泄露问题）。

1.2 语句

D 和它的兄弟语言一样，以分号结尾的表达式构成一个完整的语句（例如“ Hello,world! ”程序 writeln 以分号结尾）。语句的作用就是对表达式求值。

D 属于“ 大括号界定块范围 ”大家庭的一员，意即你可以将数个语句用大括号｛和｝括起来组成一个语句--而｛和｝有时候是不能少的，比如，在一个 foreach 循环里你要做不止一几件事时。在只有一个语句时，你可以省略大括号。事实上，我们的身高转换程序里两个循环这一部分完全可以重写成下面这样：

```
foreach (feet; 5 .. 7)
  foreach (inches; 0 .. inchesPerFoot)
    writefln("%s'%s"\t%s", feet, inches,
      (feet * inchesPerFoot + inches) * cmPerInch);
```

为一行语句省略大括号有好也有坏。好处是减少代码，坏处是编辑变成了技术活（在维护阶段，你得因为不同块的代码混在一起为了修改而时不时地这里加上大括号，那里去掉大括号）。人们总是为代码缩进和在哪里放大括号分成不同流派。事实上，只要保持一致性，并不需要为此太过强调，举一个例证，本书选择的风格（为单语句加大括号，左括号与引入它的语句处于同一行，右括号单独占一行）主要出于印刷排版的考虑，和作者本人日常的代码风格大不相同。作者不需变成狼人都可以这样，大家都可以。

Python 使得由缩进构成表达式块的这种编码风格很快流行起来。空格影响程序这个决策对使用其它语言的程序员来说有点古怪，Python 却极其信赖这一点。D 一般情况下忽略空格，但它经过特别设计，以方便解析（例如，不需了解符号的意思即可解析），因而一个加载了简单预编译的玩具程序就能采用 Python 式的代码缩进风格，编译、运行、调试都不会有问题。

上面的例子也引入了 if 语句，大致形式你应该很熟悉：

```
if (<expression>) <statement1> else <statement2>
```

一个众人皆知的优秀的理论结果即结构定理[10]证明利用 if 测试条件，for 或 foreach 进行循环组成的复合语句可以实现任何算法。当然，任何现实中的语言都提供了更多，D 也不例外。不过眼下我们还是用刚接触到的语句来表达，随着学到的语句的增多，逐步前进。

1.3 函数基础

我们先走出必不可少的 main 函数的束缚，来看看如何在 D 里定义其它的函数。函数声明遵循 Algol 类语言：首先是返回类型，然后是函数名，最后是一对圆括号包括以逗号隔开的一系列形参[1]。例如，定义一个名叫 pow 的函数，它有一个 double 类型和一个 int 类型的参数，返回值为 double，你应该写成：

```
double pow(double base, int exponent)
{
    ...
}
```

函数的每个形参（上例中的 base 和 exponent）都除了拥有类型外，还有一个可选的存储类以决定该函数被调用时实参传入的方式。默认地，实参是按值传给形参 pow 的。如果在形参类型前加上存储类关键字 ref，实参就会捆绑在形参上，对形参的改变会直接立即反映到从外部接受到的实参上。例如：

```
import std.stdio;
void fun(ref uint x, double y)
{
  x = 42;
  y = 3.14;
}
void main()
{
  uint a = 1;
  double b = 2;
  fun(a, b);
  writeln(a, " ", b);
}
```

上面的程序结果打印 42 2。原因是 x 是一个 unit 类型的引用，意味着给 x 赋值就是给 a 赋值。另一方面，给 y 赋值并不会影响 b 的值，因为 y 只拥有函数 fun 作用域内的一份私有拷贝。

要介绍的最后两个存储类是 in 和 out。简单地说，in 恪守这样一个承诺：对于它所修饰的函数参数，保证眼看手不动。out 对函数参数的修饰作用与 ref 相似，唯一不同的是不需要在进入函数体时对参数赋初值。（每种类型 T 都定义了一个初始化值，用 T.init 表示。用户自定义类型可以由用户来定义它们自己的 init）。

关于函数要说的太多了。你可以将一个函数传给另一个函数，一个套一个，允许函数保存自己的本地环境（完整的语法闭包），创建并自如地操纵匿名函数（lambda 表达式），以及其它的一些丰富特产。我们将一一接触它们。

1.4 数组和关联数组

数组和关联数组（后者俗称哈希表或哈希）可以说是计算机史上用的最多的复合数据结构，紧随其后的是满眼艳羡的 lisp 的列表。许多有用的程序只用几样数组和关组数组就可以做到，因此是时候来看看 D 是如何实现的。

1.4.1 创建一个词典

例如，让我们依据下面的描述来写一个简单的程序：

读取一段由空格间隔的单词，为每一个不同的单词加上标号。每行输出的格式为 ***标号 单词。***

这个小脚本在你做文本处理时会很有用；一旦你创建了这样一个词典，以后只需对数字操作（更经济），而不是操作整个单词。创建这样一个词典的一个可能的尝试是把所读取的单词添加给由单词映射到整数的字典中。每当添加一个新的映射，只需确保映射的数字是唯一的（一个可靠的选择是用字典当前的长度来表示标号，所得到的标号即为 0，1，2，…）。现在让我们一起来看看在 D 里怎么做。

1.本书一致用形参表示函数体内接受到的参数值，用实参表示函数执行过程中从外部传给函数的参数值。

```
import std.stdio, std.string;
void main()
{
    uint[string] dictionary;
    foreach (line; stdin.byLine())
    {
        //把句子分割成单词
        //把句子中的单词添加到词典里
        foreach (word; splitter(strip(line)))
        {
           if (word in dictionary) continue; //什么也不做
           auto newlD = dictionary.length;
           dictionary[word] = newlD;
           writeln(newlD, word);
        }
    }
}
```

在 D 里，类型 K 映射至类型 V 的关联数组（哈希表）表示为 V[K]。因此变量 dictionary 表示的类型 uint[string]即将字符串映射到整型--正是我们想要的：存储单词—标识映射。如果在关组数组 dictionary 里 word 的键值已存在，则表达式 word in dictionary 为非零值。最后，插入操作由语句 dictionary[word]=newID 完成。

在上面的脚本里虽然没有明确地说，但事实上 string 类型是真正的字符数组。大体上，类型为 T 大小可变的数组表示为 T[]，有下面几种方法获得分配：

```
int[] a = new int[20]; // 20个元素初值为0的数组
int[] b = [ 1, 2, 3 ]; // 含有元素 1，2，3 的数组
```

不象 C 数组，D 数组知道自己的长度，任一数组 arr 的长度均可由 arr.length 得到。给 arr.length 赋值意味为重新分配数组。数组访问会先进行边界检查；喜欢冒缓冲区溢出风险的代码员最怕把指针拔出了数组边界然后进行无边界检查的指针算术运算。如果你真的想得到硅晶片可以给你的一切，编译器本身也提供了一个选项，可以关掉边界检查。编译器默认提供的边界检查为安全代码铺平了道路：默认情况下代码是安全的，你得做更多的工作以获得速度上的轻微提升。

这里用了早已熟悉的 foreach 的另一种形式来迭代数组：

```
int[] arr = new int[20];
foreach (elem; arr)
{
   /* 使用elem */
}
```

上面的循环依次把数组 arr 的每个元素绑在 elem 上。给 elem 赋值不会向 arr 的元素赋值。如果要改变数组，用关键字 ref 即可：

```
//把数组arr所有元素清零
foreach (ref elem; arr)
{
   elem = 0;
}
```

现在我们知道 foreach 怎样和数组打交道了，我们再来看另一种情况。如果在迭代数组时需要元素的索引，foreach 可以为你做到：

```
int[] months = new int[12];
foreach (i, ref e; months)
{
   e = i + 1;
}
```

上面的代码创建了一个包含元素 1，2，…，12 的数组。这个循环和下面这个稍显啰嗦的代码等效，后者用 foreach 迭代了一定区间的整数：

```
foreach (i; 0 .. months.length)
{
   months[i] = i + 1;
}
```

D 也提供了静态数组，表示为如 int[5]。除了一些特殊的应用，动态数组推荐优先使用，原因是大多数情况下你事先并不能确定数组的大小。

数组是浅拷贝的，意即拷贝一个数组并不会拷贝数组的整个内容，只是对底层存储的另一种新的解读。如果你不想获取一份拷贝，用数组的 dup 属性了就够了：

```
int[] a = new int[100];
int[] b = a;
// ++x 使x自增
++b[10];   // b[10]现在的值为1,和a[10]一样
b = a.dup; // 拷贝整个a给b
++b[10];   // b[10]现在的值为 2,a[10]保持 1 不变<
```

1.4.2 数组切片，泛型函数，单元测试

数组切片是一个非常强大的特性，它可以不需拷贝数据就能引用数组的一部分。做为证明，我们来写一个与二分查找齐名的 binarySearch 算法的实现：给定一已排序的数组和一数值，binarySearch 迅速返回一个布尔值表示该值是否存在于该数组中。D 的标准库已有这样一个函数，它以泛型的方式实现，且返回更多的信息，而不仅仅一个布尔值，但得等我们学会更多的语言特性才能弄懂。不过我们可以稍稍扩张一下野心，让我们的 binarySearch 不单单适用于整型，而是适用于任一能用<做比较的类型。事实证明这个野心并不大。这个泛型 binaraySearch 看起来象这样:

```
import std.array;
bool binarySearch(T)(T[] input, T value)
{
  while (!input.empty)
  {
    auto i = input.length / 2;
    auto mid = input[i];

    if (mid > value) input = input[0 .. i];
    else if (mid < value) input = input[i + 1 .. $];
    else return true;
  }
  return false;
}
unittest
{
    assert(binarySearch([ 1, 3, 6, 7, 9, 15 ], 6));
    assert(!binarySearch([ 1, 3, 6, 7, 9, 15 ], 5));
}
```

binarySearch 签名里的符号(T)引入了类型参数 T，之后类型参数就可以和普通参数一样在函数里引用。在调用时，binarySearch 由接受到的实参推断 T 的类型。如果你想显式指定 T 的类型（例如，为了再三确认的目的），你也可以写成：

```
assert(binarySearch!(int)([ 1, 3, 6, 7, 9, 15 6));
```

这段代码告诉我们泛型数可以用两对圆括号括起来的实参来调用。先是!(…)围起来的编译期实参，然后是(…)围起来的运行期实参。我们有想过把这两个不同时期的参数结合起来放在同一个地方，但实践表明这样做得不偿失。

如果你熟悉 Java,C#或 C++语言里与之相当的特性，你肯定会发现 D 并没有用尖括号对<和>来表示编译期参数，而是用了一个完全不同的方式来定义。这是一个深思熟虑的的决定，极大的避免了 C++里所遇到的一些严重问题，比如解析难度增加，繁复的规则定义，为排除歧义而显得晦涩难懂的语法 [2]。这些麻烦起因于<和>说到底只是比较操作符 [3]，用它来做定界符当其内有表达式时就非常容易造成歧义性。象这样的定界符在配对时就变得很困难。Java 和 C#日子过得比较轻松是因为它们都不允许在定界符内有表达式存在，但这限制了语言在这些特性上的可扩展性。D 的确允许表达式作为编译期参数，它选择了一种对人和对编译器都比较简单的方式，就是把传统的单目运算符！扩展为双目运算符，这是通过使其后紧跟圆括号（我相信你总能正确匹配圆括号）来做到的。

另一个有关 binarySearch 实现的细节是用关键字 auto 做类型推断：i 和 mid 分别在它们各自的初始化表达式里获得推断的类型。

为保持良好的编程习惯，binarySearch 附带着单元测试。单元测试由关键字 unittest 打头的语句块引入（一个文件里面可以包含足够多的 unittest，但我想你懂得过犹不及的道理）。在进入 main 函数体之前要进行单元测试的话，只要将编译开关-unittest 传给编译器即可。单元测试初看起来象是一个微不足道的小特性，实际上它可以帮助你观察什么是良好的编程风格：是代码里很容易地插入这样的测试语句，还是很费事。另外，如果你是一个自顶而下的思想者，偏好于先设计 unittest 然后实现具体功能，你尽管将 unittest 移到 binarySearch 的前面；在 D 里，模块级的符号语法从不会依赖于代码出现的先后次序。

切片表达式 input[a..b]返回数组 input 的一个切片，从索引 a 到 b，不包括 b。如果 a==b，则返回一个空切片，如果 a>b，则引发异常。切片不会触发动态内存分配；它只是数组一部分的别名。在索引表达式内部或是切片表达式内部，$代表被访问数组的长度；比如，input[0..$]与 input 等价。

再讲一遍，虽然看起来 binarySearch 多次移动数组，实际上并没有新的数组获取分配；所有 input 的切片都和原 input 共享共一地址。这种技术和传统的基于索引维护索引的实现相比不会慢一丁点，但却更容易理解，因为它所需操作的状态更少。讲到状态，让我们一起再来写 binarySearch 算法，这次用递归实现，并且完全不需重复索引:

```
import std.array;
bool binarySearch(T)(T[] input, T value) {
   if (input.empty) return false;
   auto i = input.length / 2;
   auto mid = input[i];
   if (mid > value) return binarySearch(input[0 .. i]);
   if (mid < value) return binarySearch(input[i + 1 .. $]);
   return true;
}
```

---

2. 如果你的 C++程序员同事有超人般的自信，你问问他或她 object.template fun<arg>() 这个语法起什么作用，你将会看到 Kryptonite 在使坏了。【译注】Kryptonite ，超人唯一的弱点。

3.给伤口上再撒把盐，<<和>>也都是运算符。

递归实现的算法相比迭代实现要简练易懂，而且一点也不比迭代版本的低效，因为现代的编译器都会利用尾调用消除技术：简单地说，如果一个函数的 return 语句仅仅是简单靠传递不同的实参来调用自身，编译器会修改实参并跳转到函数开始的地方。

1.4.3 频率统计，Lambda 函数

从现在开始我们着手写另一个有用的程序：从一个文本中对每一不同的单词计数。想知道《哈姆雷特》里什么单词用的最频繁？你来对了。

下面的程序使用一个关联数组映射字符串到无符号整数，和前面的词典程序的结构有点象，再加上一句简单的循环打印就可以完成这个词频统计程序：

```
import std.stdio, std.string;
void main()
{
    // 计算出现的频率
    uint[string] freqs;
    foreach (line; stdin.byLine())
    {
        foreach (word; split(strip(line)))
        {
            ++freqs[word.idup];
        }
     }
   // 打印计算结果
   foreach (key, value; freqs)
   {
       writefln(" %6u\t %s", value, key);
   }
}
```

从网上下载一份 hamlet.txt（你可以在 http://erdani.com/tdpl/hamlet.txt,得到，该链接永久有效。），运行程序，输出：

```
1 outface
1 come?
1 blanket,
1 operant
1 reckon
2 liest
1 Unhand
1 dear,
1 parley.
1 share.
...
```

上面显示出打印的结果并没有排好序，使用最频繁的单词也没能排在最前面。这并不奇怪：为使关联数组尽可能的快，它被设计成随机存储。

为了达到最频繁使用的单词最先打印这个目的，你可以在运行程序时添加开关 sort –nr（按数字升序排列然后倒置），但是这样做有点取巧。要使程序本身支持排序，我们用下面的代码来替换原程序里的最后一个循环：

```
// 计算出现的频率
string[] words = freqs . keys;
sort! ((a, b) { return freqs[a] > freqs[b]; }) (words) ;
foreach (word; words)
{
    writefln("%6u\ns" , freqs [word] , word) ;
}
```

属性.keys 取出关联数组 freqs 的所有键值并生成一个字符串数组。这个数组是新申请的，这是必需的，因为等下我们要打乱数组的排列。现在让我们来看看下面的代码：

```
sort!((a, b) { return freqs[a] > freqs[b]; })(words);
```

它看起来有点象我们已接触过的下面这个结构：

```
sort!(<compile-time arguments>) (<runtime arguments>) ;
```

剥掉!(...)外壳，里面的部分看起来，有点象一个忘记了参数类型、返回类型和函数名的不完整的函数：

```
(a, b) { return freqs[a] > freqs[b] ; }
```

这就是 Lambda 函数—传递给其它函数的一个短小的匿名函数。Lambda 函数在很多场合非常有用，而 D 也尽力消除在声明 Lambda 时不必要的累赘：省略了参数类型和函数返回类型，这样是没问题的，原因是 Lambda 函数体仅在当下（场合）定义，不管是作者本人，读者或是编译器看到这个定义，都不会产生任何误解，也不会破坏模块化的规则。

该例中的 Lambda 函数还有一个比较细节的地方需要说一下。Lambda 函数访问的 freqs 变量是在 main 里定义的一个本地变量，并不是全局变量或静态变量。相比 C 语言，这一点更象 Lisp，使得 Lambda 更加强大。虽然传统上这样强大的特性往往是以运行时的效率损失为代价（需要间接函数调用），但 D 确保没有间接函数调用（充分的内联）。

修改后的程序输出：

```
929   the
680   and
625   of
608   to
523   I
453   a
444   my
382   in
361   you
358   Ham.
```

正如所料，最经常使用的单词出现的频率最高，只有”Ham.”例外，这并不是为了显示这位戏剧主角爱好厨艺，而是所有哈姆雷特出现的行主角名字 Ham 后都有一个小圆点后缀。他前后开口 358 次，比其他任何角色都要多。如果你把打印结果往下翻，你会发现另一个发言人是国王，只有 116 行—不到哈姆雷特的三分之一。而仅仅的 58 行表明，Ophelia 可称得上惜言如金。

1.5 基本数据结构

现在我们已深入研究了《哈姆雷特》，让我们更深一步分析一下它的文字特点。例如，针对于每一个戏剧角色，我们来收集一些信息，比如说了总共多少个单词以及词汇量有多丰富。为完成这个任务，我们需要将几种数据与一个角色关联。为了将这数种信息收集到一起，我们定义一个如下所示的结构：

```
struct PersonaData
{
   uint totalWordsSpoken;
   uint [string] wordCount;
}
```

D 提供了 struct 和 class，它们拥有许多相似的特性，但有着本质的区别：struct 是值类型而 class 意味着动态、多态并且只能按引用访问。这种设计可以极大地避免混淆歧义、与数组切片相关的 bug，以及类似 “ //不！不要继承它！ ” 的注释。当你设计一个类型时，你从一开始就得决定要它成为单态的值类型还是多态的引用类型。C++的一个出名的特点是可以定义含混不清的类型，但是极为少见，而且容易滋生错误，我有理由反对使用它，因为你可以通过良好的设计而避免。

在上面的例子里，我们只需要收集一些数据，没有想过要多态，因此使用 struct 就好了。现在我们来定义一个映射角色到 PersonaData 结构的关联数组：

```
PersonaData[string] info;
```

我们要做的就是从文本文件 hamlet.txt 提取合适的信息然后写入到关联数组 info 里。这需要一定的工作，因为一个演员的一段台词可能有很多行，所以我们得处理一下这些多行台词以便正确衔接为段落。为了说明如何操作，我们来看看下面这段摘自《哈姆雷特》的片段，然后逐字剖析（为清楚起见把开始的空格也标了出来）：

```
__Pol. Marry, I will teach you! Think yourself a baby
____That you have ta'en these tenders for true pay,
____Which are not sterling. Tender yourself more dearly,
____Or (not to crack the wind of the poor phrase,
____Running it thus) you'll tender me a fool.
__Oph. My lord, he hath importun'd me with love
____In honourable fashion.
__Pol. Ay, fashion you may call it. Go to, go to!
```

Polonius 在说 go to 时迸发的过度热情与他的死亡是否互为因果，只到今日尤令人深思。先不说这个，我们已经留意到，每个演员的台词都是以两个空格开始，后随演员的名字（有的是缩写），然后一个句号，再一个空格，最后是真正的台词。如果一个逻辑行占了多行，后续各行均前导 4 个空格。我们可以用一个正则表达式引擎（可以在 std.regex 模块找到）来完成这个简单的匹配，但为了学习数组，我们还是徒手搞定。我们借助于 std.algorithm 模块下的一个返回值为布尔的函数 a.startsWith(b)，该函数可以判断 a 是否以 b 打头。

main 函数读取输入行，并把它们连接成逻辑行（忽略任何不合乎要求的匹配），然后把完整的段落传给一个累加函数，最后打印出期望的信息。

```
import std.algorithm, std.conv, std.ctype, std.regex,
std.range, std.stdio, std.string;
struct PersonaData
{
   uint totalWordsSpoken;
```

```
    uint [string] wordCount;
}
void main()
{
    //累加戏剧角色的有关信息
    PersonaData[string] info;
    // 填充info
    string currentParagraph;
    foreach (line; stdin.byLine())
    {
       if (line.startsWith("    ")
       && line.length > 4
       && isalpha(line[4]))
       {
          //处理跨行的角色台词
          currentParagraph ~= line[3 .. $];
       }
       else if (line.startsWith(" ")
       && line.length > 2
       && isalpha(line[2]))
       {
           //角色台词刚刚开始
           addParagraph(currentParagraph, info);
           currentParagraph = to!string(line[2 .. $]);
       }
    }
    // 完成，打印收集的信息
    printResults(info);
}
```

只要我们已弄明白了数组是如何工作的后，上面的代码都是自解释的了，只剩下 to!string（line[2..$]）没有提到。为什么需要它?如果忘记了写这行代码会怎样？

foreach 循环从 stdin 尝试读取每行文本，如果读取成功，则将之存入变量 line。为读入的每一行分配一个全新的缓冲区太浪费，所以 byLine 重新使用传给 loop 循环的 line 所包容的内容。line 本身是 char[]类型 -- 一个字符串数组。

如果只是简单的检查所读取的每一行然后随手丢弃，一切好象都很正常。但是，如果是要将每一行的内容存起来备用，最好还是复制一份。显然，currentParagraph 有意要储存文本，因而复制是必需的；因之 to!string 的出现就是将任一表达式转换为一 string。类型 string 本身就是不被修改的，并且提供必要时的复制操作以确保不可变这一承诺。

现在，如果我们忘了加上 to!string，后续的代码将仍旧能够通过编译，但的其结果有可能是没有任何意义的，并且也很难发现 bug 的所在。一部分被修改的数据散布于程序的另一部分，要想追踪是让人很不爽的，因为这并不单纯是本作用域造成的（在一个大型程序中有多少个地方调用了这部分数据你能全部记住没有遗漏？）。幸运的是，在这个例子里事实并非如此，原因是 line 和 currentParagraph 分别反映着各自的能力：line 拥有 char[]类型，亦即一个可被随时修改的字符串数组；而 currentParagraph 却拥有 string 类型，一个单个字符不能被修改的字符串数组。（如果你有点好奇，那么：string 的全名是 immutable(char)[],它的准确意思是 “ 一段连续区域的不可变字符 ”，我们将在第 4 章详细加以讨论）这两个类型不可能同时引用同一块内存，因为 line 会破坏 currentParagraph 的承诺。因而编译器将拒绝编译这样的错误代码，它要求 line 的一份拷贝。现在你通过 to!string 转换提供了这样一份拷贝，你看现在大家都开心了吧。

另一方面，当你到处拷贝 string 时，底层数据不会被复制—它们共同引用同一块内存，因为任一份拷贝都不可能被修改，这一特性使得 string 在复制时既安全又高效。更妙的是，sting 可以毫无问题地在不同线程间共享，原因是，再重复一遍，它们从不吵架。看到了吧，不可变性超酷。另外，如果你非得要修改单个字符，你可能需要用 char[]（而不是 string）,那怕是临时的。

上面声明的 PersonaData 非常简单，事实上，struct 不单单能声明数据，也能声明其它的实体，比如：私有块，成员方法，单元测试，操作符，构造函数和析构函数。默认地，结构所拥有的每一种数据成员都会用初始值来初始化（整型初始化为 0，浮点数被初始化为 NaN[4]，数组及其它间接访问类型初始化为 null）。现在我们来实现 addParagraph，它切取一行文本然后将之存入关联数组。

供 main 所用的行具有这样的形式：“ Ham. To be,or not to be- that is the question. ”。我们得找到第一个“ . ”以区别角色的名字与实际的台词。我们用 find 函数来做到这一点。haystack.find(needle) 返回 haystack 中以 needle 首次出现的地方开始的右半部分（如果没有找到，find 返回一个空的 string）。在收集词典的同时，我们也要做一点清理工作。首先，我们得把整个句子转换为小写，这样大写和小写单词才会被当作同一个单词对待。这个任务用 tolower 函数就可轻松搞定。第二，我们得消除一个极大的噪音：标点符号使得“ him. ”和“ him ”被当做两个不同的单词。为达到这个清理目的，我们要做的就是给 split 传递一个附加的参数 regex(“[\t,.;:?]+”)，它是一个正则表达式，用以剥掉这层壳。有了这个参数，split 函数将视所有出现在正则表达式里[和]之间 的字符为单词分割符。就是说，我们已做好了准备，用很少的代码做很多有趣的事了：

```
void addParagraph(string line, ref PersonaData[string] info)
{
    //分开角色和台词
    line = strip(line);
    auto sentence = std.algorithm.find(line, ". " );
    if (sentence.empty)
    {
        return;
    }
    auto persona = line[0 $ - sentence.length];
    sentence = tolower(strip(sentence[2 .. $]));
    //得到所有说过的单词
    auto words = split(sentence, regex("[ \t,.;:?]+"));
    // 添加或更新信息
    if (!(persona in info))
    {
        // 该角色第一次说话
        info[persona] = PersonaData();
    }
    info[persona].totalWordsSpoken += words.length;
    foreach (word; words) ++info[persona].wordCount[word];
}
```

这一大段 addParagraph 主要完成关联数组的更新。如果这个角色尚未听说，则向关联数组里插入一个空的，默认构造的 PersonaData 对象。由于默认构造的 uint 和关联数组分别拥有初值 0 和为一个空数组，这个新添的槽已准备好吸收有意义的信息了。

最后我们来实现 printResult 来为每个角色打印简要总结：

4.对浮点数来说 NaN 是一个很好的初始化值，但不遗憾的是，整型没有类似的初始化值。

```
void printResults(PersonaData[string] info)
{
    foreach (persona, data; info)
    {
        writefln("%20s %6u %6u", persona, data.totalWordsSpoken,
        data.wordCount.length);
    }
}
```

准备好测试了吗？保存，运行！

```
Queen       1104    500
Ros          738    338
For           55     45
Fort          74     61
Gentlemen      4      3
Other        105     75
Guil         349    176
Mar          423    231
Capt          92     66
Lord          70     49
Both          44     24
Oph          998    401
Ghost        683    350
All           20     17
Player        16     14
Laer        1507    606
Pol         2626    870
Priest        92     66
Hor         2129    763
King        4153   1251
Cor., Volt    11     11
Both [Mar      8      8
Osr          379    179
Mess         110     79
Sailor        42     36
Servant       11     10
Ambassador    41     34
Fran          64     47
Clown        665    298
Gent         101     77
Ham        11901   2822
Ber          220    135
Volt         150    112
Rey           80     37
```

有趣的东西来了。毫不惊讶，我们的朋友“ Ham ”占有最大份额。Voltemand（Volt）的却很有意思：他说的不多，但在这不多的话语里他却展示出了非常丰富的词汇，在这点更不用说 Sailor 了，他几乎没有重复一个单词。再比比多才多艺的女王和 Ophelia:女王比 Ophelia 多说大概 10%的单词，但词汇量却多出不少于 25%。

输出结果也存在一些噪音（比如“ Both [Mar ”），一个勤劳的程序员很容易修改好它而且几乎不影响统计结果。还是把它留做（而且推荐）一个启发练习吧。

1.6 接口和类

面向对象的特性对大的项目来说很重要；尝试用很小的例子来解说显得有点笨。如果忍住不用那些如 shapes,animals,employees 等被用烂了的例子，好象陷入了没有好例子可用的困境，更有甚然，小例子通常缺失了多态对象的创建，这可是大事情。作者的路障！幸运的是，现实世界提供了一个有用的例子，它可是一个活生生的问题，相对比较小，但过程式语言还没有一个比较满意的解决方案。以下我们要讨论的代码是对一个小而有用的 awk 脚本的重写，而且在原有方案所隐藏的限制的基础上有很大的扩展。我们一起来迈向这个面向对象的、短小、完整而优雅的实现。

假设现在要写一个叫做 stats 的短小程序，它有一个简单的接口：stats 接受一系列的统计函数做为命令行参数，并且从标准输入接受空格隔开的一系列数字，最后分行输出每种统计结果。下面是一例程会话：

```
$ echo 3 5 1.3 4 10 4.5 1 5 | stats Min Max Average
1
10
4.225
$ _
```

一个快速应急的脚本可以毫无疑问地完成这个任务，但当传入的统计函数越来越多时，“ 应急 ”还行，“ 快速 ”就靠不住了。我们还是一起找一个好点的方案吧。眼下，我们从最简单的统计函数开始：最小值 Min，最大值 Max，平均值 Average。一旦我们找到一个可扩展的方案，实现更多更复杂的统计函数之门也就随之为我们打开了。

一个简单的实现就是迭代输入的数据然后计算所有需要的统计。这不是一个可扩展的设计，原因是每次当我们需要添加一个新统计函数时，都得对现有代码动手术。那么，如果我们只是根据命令行传入的统计要求进行计算，这样的修改应该不必太繁琐。理论上，我们最好把每个统计函数限制在同一段代码内。这样我们就可以通过简单地添加代码来扩充新功能--闭合原则【39】得以最恰当的运用。

这个方案的关键在于找到所有的或者最低程度上，大多数的统计函数具有什么共同点，这样才能从同一个地方用同一方式操作它们。我们来边说边做。留意到 Min 和 Max 每次接受一个数字，一旦输入完成，就立即计算结果，该结果为一数字。另外，Average 还得做一个收尾动作（用输入数字的总和除以输入数字的个数）。还有，每个算法各自维护自身的状态。象这样各种不同的算法共同遵守同一约定并需维护各自的状态，最好的办法就是使它们成为对象，再为所有的对象定义同一个的接口。

```
interface Stat
{
    void accumulate(double x);
    void postprocess();
    double result();
}
```

接口用一组方法来定义所要求的行为，所以不论是谁要实现这个接口就得依据这些方法的声明逐一定义这些接口。讲到实现，我们来看看怎样定义 Min 以遵从 Stat 的铁律：

```
class Min : Stat
{
   private double min = double max;
```

```
   void accumulate(double x)
   {
      if (x < min)
      {
        min = x;
      }
   }
   void postprocess() {} // 什么也不做
   double result()
   {
      return min;
   }
}
```

Min 是一个类—一种用户自定类型，为 D 引入了很多面向对象的好东西。Min 通过 class Min:Stat 这样的语法清清楚楚地实现了 Stat 接口，并且老老实实地用相同的参数和返回值实现了 Stat 接口的三个方法（不然就过不了编译器这一关）。Min 仅有一个私有变量 min，它保存到目前为止的最小值，并在 accumulate 内更新。min 的初值是 double 类型的最大值，因而程序里输入的第一个数字将会取代它。

在定义更多的统计函数前，我们来为 stats 程序写一个测试驱动，它读取命令行参数，创建运算操作所需的对象（例如：如果命令行传入 Min 参数就需创建 Min 对象），并通过接口 Stat 来使用这些对象。

```
import std.contracts, std.stdio;
void main(string[] args)
{
    Stat[] stats;
    foreach (arg; args[1 $])
    {
        auto newStat = cast(Stat) Object.factory("stats." ~ arg);
        enforce(newStat, "Invalid statistics function: " ~ arg);
        stats ~= newStat;
    }
    for (double x; readf(" %s ", &x) == 1; )
    {
        foreach (s; stats)
       {
          s.accumulate(x);
       }
    }
    foreach (s; stats)
    {
        s.postprocess();
        writeln(s.result());
    }
}
```

这个程序做了很多事情，但实际上一两句话就可以讲清楚。首先，main 拥有的签名与我们之前所看到的有所不同—一个 string 数组做为参数。D 运行时支持用命令行参数来初始化数组。第一个循环用 args 来初始化 stats 数组。已知在 D 里（和其它语言）命令行的第一个参数是该程序的名字，我们通过 args[1..$] 跳过它。接着我们遇到：

```
auto newStat = cast(Stat) Object.factory("stats." ~ arg);
```

它有点长，但是，套用一句台词，“ 我能解释一切 ”。首先，~做为双目操作符时，用来连接字符串，所以如果命令行传入 Max，连接后的字符串即为”stats.Max”，它被传给函数 Object.factory。Object 是所有类对象的根，它定义了一个静态方法 factory，该方法接受一个字符串参数，在编译时通过从一个小小的数据库里查找信息来魔术般地创建一个与传入的参数同名的类对象，并返回这个对象。如果该类不存在，Object.factory 返回 null。为了能让这个调用成功，你唯一要做的就是在同一源文件里已经定义好一个名叫 Max 的类。通过给定的名字来创建对象是一项非常重要的特性---它是如此重要，以至一些动态语言使之成为语言的核心特性；拥有更多静态设计特征的语言要么需要依赖于运行时支持（象 D 和 Java），要么把它交给程序员去设计一个手工注册和检索的机制。

那么为什么是 stats.Max 而不是 Max?D 对待对待模块非常慎审，它没有一个任谁都可以把任合东西扔进去的全局命名空间。每种标识都栖身于一个命名模块下，默认情况下，模块名就是源文件的基名称。所以对一个名叫 stats.d 的文件，所有在该文件里定义的名字都归属于 stats 模块。

还剩一个问题没交待。这个返回 Min 对象的静态方法的返回类型并不是 Min。这听起来有点傻傻的，但是基于这样一个事实，调用 Object.factory(“whatever”)应该能够创建任何类型的对象，因此该方法的返回类型应该更通用—Object，这就是其中的原因。为了能对新创建的对象施以合适的调用，我们得让他为成一个 Stat 对象，类型地转换操作 cast 就是干这活的。在 D 里，表达式 cast(T)expr 将表达式 expr 转换为类型 T。涉及到类和接口的转换都会进行类型转换检查，所以我们的代码安全无比。

倒回去看，我们留意到在 main 的前 5 行我们已做了不少实际的工作。这也是最难的部分，因为余下的代码都是自解释的。第二个循环每次读取一个数字（由 readf 搞定），然后为每一种统计实例调用 accmulate。feadf 函数读取给定格式的数字并返回成功读取的个数。在本例，格式是“ %s ”，它的意思是前后均有若干空格的一项输入（输入的类型取决于所读取的元素本身，在本例 x 的类型为 double）。最后，程序打印所有结果。

1.6.1 更多统计，继承

实现 Max 和实现 Min 一样简单；除了 accumulate 有一点改变，其它都一模一样。当一个新任务看起来有点象旧的时，“ 有趣 ”和不“ 麻烦 ”这两个念头应该首先涌上心头。重复性任务给了我们再次利用代码的机会，能更好的应对这种相似性的合适的的语言在扩展性上应表现出更大的优越性。我们要找出的是 Min 和 Max(希望还有更多的统计函数)所具有的某种共性。如果我们仔细想一想，就能看出它们都是这样一组统计函数：根据输入的增多来计算结果，并且只用一个数字来描述这个结果。我们姑且把这种统计函数称做递增函数（incremental function）。

```
class IncrementalStat : Stat
{
    protected double _result;
    abstract void accumulate(double x);
    void postprocess(){}
    double result()
    {
        return _result ;
    }
}
```

一个抽象类（abstract class）可以看做不完全地提供了这样一个承诺：它实现了一系列的方法，但不是全部，这样的类不能直接工作。使抽象类实质化的方法就是由之继承并实现所有的接口。IncrementalStat 承传了 Stat 几乎所有代码，只留下 accumulate 有待继承它的子类来实现。新 Min 类看起来象下面这样：

```
class Min : IncrementalStat
{
    this()
    {
```

```
        _result = double.max;
    }
    void accumulate(double x)
    {
        if (x < _result)
        {
            _result = x;
        }
    }
}
```

Min 类声明了一个构造函数 this() ----一种特殊的函数，它对 result 进行适当的初始化。就这样一个构造函数，只要构造的合适，已为后续的代码节省的大量的时间，特别是当我们想到许多统计函数（比如总和，方差，平均数，均方差）。我们先来看看平均数的实现，因为这是一个介绍几个新概念的好机会：

```
class Average : IncrementalStat
{
    private uint items = 0;
    this()
    {
        _result = 0;
    }
    void accumulate(double x)
    {
        _result += x;
        ++items;
    }
    override void postprocess()
    {
        if (items)
        {
            _result 1= items;
        }
    }
}
```

首先，Average 引入了一个新的成员变量 items，通过=0 这样的语法来初始化（在这里是纯属为了介始成员变量初始化的语法。在眼下这个地方这样来初始化是多余的，正如第 13 页已讨论过的那样，整型类型都拥有初始化值 0）。其次，Average 的构造函数将 result 设为 0，这是因为，不同于最小值和最大值，0 的平均值就是 0。虽然看起来用 NaN 来初始化 result，以后再用 0 改写它也不费功，但是任何尝试对这种被称作 “ 死赋值 ” 的优化都将收效甚微。最后，虽然 IncrementStat 已经定义了 postprocess，Average 还是重写了它。在 D 里，默认情况下，你可以对所有的类方法进行重写（继承和重定义），但你必须用上 override 关键字，以避免各种意外情况的发生（比如，因拼写错误或基类实现的改变而没有重写必须重写的方法，或者重写了不该重写的方法等）。如果你在一个类的方法成员前加下 final 关键字，它将禁止派生类对该方法进行重写，有效地阻止了动态查找机制。

1.7 值类型和引用类型

我们来进行一个简单的实验:

```
import std.stdio;
struct MyStruct
{
    int data;
}
class MyClass
{
    int data;
}
void main()
{
    //测试MyStruct对象
    MyStruct s1;
    MyStruct s2 = s1;
    ++s2.data;
    writeln(s1.data); //打印 0
    //测试MyClass对象
    MyClass c1 = new MyClass;
    MyClass c2 = c1;
    ++c2.data;
    writeln(c1.data); //打印 1

}
```

看起来玩 MyStruct 和玩 MyClass 是两个完全不同的游戏。两种情况我们都创建了一个变量，然后复制给另一个变量，随后我们修改了复制品的值（回忆++是一个单目运算符，它使参数的值自增）。这个实验揭露了这样一个事实：复制后，c1 和 c2 引用的是同一块内存，而 s1 和 s2 则相反，它们各自拥有独立的生活。

MyStruct 的行为遵从值语义：每个变量指向一个独立的值，把一个变量赋值给另一个变量意味着把一个变量的状态复制给另一个变量从而取代另一个变量的原有状态。复制源不会改变，两个变量互不影响。MyClass 的行为遵从引用语义：值被显式创建（本例中调用 new MyClass），把一个类变量赋值给另一个变量仅仅意味着两个变量指向同一个值。

值语义比较容易应付，道理也比较直白，允许有效地设计一些尺寸较小的类型。另一方面，如果没有“ 不用复制就可以引用一个值 ”这样的手段，一些不太简单的程序在实现起来就很困难。值语义先天就排除了这样一些应用，比如，构建一个自引用类型（list 和 tree），或者一些互相引用的结构，如可以获知父窗口状态的子窗口。任何严肃的语言都会实现某种引用语义；大家有可能的争议无非在于默认的语义是什么。C 排它性地拥有值语义，而允许用指针的形式来构造引用。除了指针外，C++还定义了引用类型。有趣的是，纯函数式语言不加区别地随意使用值语义或引用语义，因为用户代码无法区分这一点。这是因为纯函数式语言不允许值的可变性，所以你没法分辨它到底是攫取了一份拷贝还是只是一个引用---反正它是被冻结的，你没法通过试着改变它的值来检验它是否被共享。相反的是，纯面向对象的语言本来就是可变性密集型的语言，它拥有排它性的引用语义，其中一些甚至到了这样的程序，比如支持允许动态改变系统级的常量这样一种让人困惑的特性。

D 选择了一种杂交性的体系。你需要引用类型，就使用类，你需要值类型或杂交类型就用结构。第 6 章和第 7 章（分别）进行详细描述。每种类型的构造都因这种根本性的设计的选择被赋予了恰当的便利与特性。例如，struct 不支持动态继承和多态（如上面 stats 例子所展示的），因为这种行为与值语义不兼容。对象的动态多态需要引用语义，任何引起混淆的做法只会带来大麻烦。（比如，C++里一个常见的危险就是切片，比如，当你不小心把一个对象当做值类型来使用时，一下子就夺去了它的多态能力；而这在 D 里根本不会发生。）

一个比较接近的想法是可以认为 struct 在设计上更有弹性。你声明了一个 struct 后，你可以指定任何你想要的语义，要么值类型，要么懒拷贝或称写时复制，要么引用计数，甚或介于几者之间。你甚至也可以在 struct 内用 class 或指针来定义引用语义。另一方面，这其中的一些特技可能需要通晓更高级的技术；相反，你也可以在各种场合下都利用 class 所提供的简单性和一致性。

1.8 总结

因为本章只是介绍篇，部分概念和例子忽略了一些细节，部分假定你在其它语言里有一定的熟悉。另外要说的是，一个有经验的程序员应该很容易完善和改进这些例子。

幸运的是，这章毕竟包含有大家都需要的一些东西。如果你是一个注重实际，非无喱头派的程序员，你一定会为数组和关联数组的洁净感到眼前一亮。这两个概念一起营造了一个完全不同的世界，不论是大项目还是小项目，都极大地简化了日常的编程。如果你乐于面向对象这一特色，毫无疑问接口和类对你来说应该再熟悉不过，相信本章也为需要扩展性的大型程序提供了极好的启发。如果你也想用 D 来写小脚本，本章也已展示了文件操作的脚本，一样的容易编写，运行良好。

和任何别的一样，整个故事总是有点长。然而，回到最基本的地方，让简单的东西保持简单总是最有帮助的。