China-pub.com
下载

第8章　可移植性

最后，标准化与常规相仿，能成为强有力秩序的另一种具体化形式。但是它又与常规不同，它已经被现代建筑学公认为是我们技术的浓缩产物，因此以其潜在的支配地位和蛮横特征而令人恐惧。

Robert Venturi，《建筑学中的复杂和矛盾》

写出能够正确而有效地运行的软件是很困难的。因此，如果某个程序能在一个环境里工作，当你需要把它移到另一个编译系统，或者处理器，或者操作系统上时，不会希望再重复做太多原来已经做过的工作。最理想的情况是什么都不用改。

这种理想就是程序的可移植性。实际上，“可移植性”常被用来指一个更弱的概念，其意思是说，与凭空写出这个程序相比，对它做些修改挪到另一个地方将更容易一些。这种修改越容易做，我们就说这个程序的可移植性越强。

你可能会奇怪，为什么我们还要为可移植性费心呢？如果软件都是准备在某些特定条件下，在一个特定环境里运行的，为什么还要在使它具有更广泛的可接受性方面白费精力呢？首先，任何成功的程序，几乎总是注定要被以原来不曾预料的方式，用到从未想到的地方去。把一个软件构造得比它的规范更一般些，结果就会是以后的更少维护和更好使用。第二，环境总是在变。当编译系统、操作系统或者硬件升级的时候，其特性可能就不同了。程序对特殊特征的依赖越少，它也就越少可能崩溃，也越容易适应改变以后的环境。最后，也是最重要的，可移植的程序总是更优秀的程序。为把程序构造得更具有可移植性的努力也会使它具有更好的设计，更好的结构，经过更彻底的测试。一般地说，可移植程序设计的技术与优良程序设计的技术是密切相关的。

当然，可移植性的程度也应当根据现实来考虑。不存在什么绝对的可移植程序，只有那种已经在足够多的环境里试验过的程序。但是，我们仍然可以把可移植性作为一个目标，力图去开发那种几乎不用修改就能够运行在任何环境上的软件。甚至当这个目的不能完全达到时，在程序构造过程中花在可移植性上的功夫也将会得到回报，例如在这个软件需要升级的时候。

我们的看法是：应该设法写这样的软件，它能工作在它必须活动于其中的各种标准、界面和环境的交集里。不要为纠正每个移植性问题写一段特殊代码，正相反，应该修改这个软件本身，使它能够在新增加的限制下工作。利用抽象和封装机制限制和控制那些无法避免的不可移植代码。通过将软件维持在各种限制的交集里面，局部化它的系统依赖性，这样你的代码在被移植后仍将更加清晰、更具通用性。

8.1　语言

盯紧标准。得到可移植代码的第一步当然是使用某种高级语言，应该按照语言标准（如果有的话)去写程序。二进制不可能很容易地移植，但是源代码可以。当然，即使这样做也还会有问

China-pub.com 下载

题，在编译系统如何将源程序翻译到机器指令的方式方面，也可能有些东西没有精确定义，对标准语言也是如此。广泛使用的语言只存在一个实现的情况是罕见的，通常都有多个编译系统提供商，对于不同操作系统，又有不同的版本，还有随着年月更替而不断出现的不同发行版本。它们对你的源代码可能做出不同解释。

为什么语言标准不是一个严格定义呢？有时标准是不完全的，对某些特性之间的相互作用没有给出定义。有时标准会有意地不对某些东西做出定义，例如， C和C++ 语言里的char类型可以是有符号的或是无符号的，而且不必正好是 8位。把这些事项留给写编译系统的人去解决，有可能产生出更有效的实现，或者避免语言对它能在其上运行的硬件提出太多限制。当然，这种做法可能给程序员带来困难。政治上和技术上的相容性问题也可能导致某种妥协，使标准对某些细节不做具体定义。最后，语言都是极端复杂的，编译系统也很复杂，理解中可能出错，实现里面也可能有毛病。

有时语言根本没有经过标准化。 C语言正式的ANSI/ISO标准在1988年颁布，而ISO的C++的标准直到1998年才被批准，在我们写这些的时候，还没有一个在用的编译系统支持这个正式标准。Java是更新一些的语言，与标准化的距离还有许多年。一个语言标准的开发通常总是要等到这个语言已经有了许多不同的、互相冲突的实现，有了进行统一的需求的时候；此外，它也必须已经被广泛使用，值得付出标准化的代价。在这期间，还是有许多程序需要写，有许多环境需要支持。

综上所述，虽然在给人的印象上，参考手册和标准是一种严格规范，但它们从来也不能完全地定义一个语言。这样，由不同实现给出的就可能都是合法的，但却又是互不相容的解释。有时甚至实现中还存在错误。在我们刚开始写这一章的时候，发现过一个很有意思的小问题。下面的外部说明在C或者C++ 里都是不合法的：

```
? *x[] = {"abc"};
```

我们测试了十来个编译系统，只有不多几个正确诊断出 x缺少char类型说明符；好几个系统给出类型不匹配的警告 (它们明显是采用了语言的老定义，错误地推论出 x是一个整型指针的数组)，还有几个在编译这段非法代码时一点牢骚也不发。

在主流中做程序设计。某些编译系统不能辨识上面的错误，这当然很不幸，也说明了与可移植性有关的一个重要问题。任何语言都有黑暗的角落，在那里实践会出现分歧。例如 C和C++ 的位域，回避它们是比较稳健的做法。我们应该只使用那些在语言的定义里毫无歧义、而且又很容易理解的东西。这类特性更可能是到处都能用的，也会在任何地方都具有同样的行为方式。我们称这种东西为语言的**主流**。

要想确定哪里是主流有时也非常困难，但我们很容易辨明哪些东西是在主流之外。一些全新的东西，例如C里面的 // 注释或者complex类型；或者那些特定的与某种体系结构有关的东西，如near或者far；它们一定会带来麻烦。如果某个特性是如此地不寻常、不清楚，为了理解它，你在阅读定义时必须去咨询一个“ 语言律师 ”，一个专家，那么请不要用它。

在下面的讨论里，我们要把注意力集中在 C和C++，它们是常被人用来写可移植程序的通用程序语言。C语言标准已经有了十几年的历史，这个语言也是很稳固的。人们正在为建立一个新标准而工作，所以，也可以说是喷发在即。在另一方面， C++ 标准则是全新的，各种实现还没有时间汇合到一起。

什么是C语言的主流？这个术语常被用来指那些已经建立起来的语言使用风格，但我们最

China-pub.com
下载

好还是为将来做点准备。例如，原来的 C 版本并不要求函数原型，说明sqrt是一个函数的方式是写：

```
?     double sqrt();
```

这里定义了函数的返回值类型，对参数则什么也没说。 ANSI C增加了函数原型，它把所有东西都刻画清楚了：

```
double sqrt(double);
```

ANSI C标准要求编译系统也要接受原来的语法，不过你无论如何也应该为自己的所有函数都写出原型。这样做能保证得到更安全的代码，保证所有的函数调用都得到完全的检查。此外，如果界面改变了，编译系统也能够捕捉到它们。如果你的代码里有调用：

```
func(7, PI);
```

如果函数func没有原型，那么编译系统就很可能不检验 func调用的正确性。如果后来有关的库改变了， func改变为有了3个参数，必须修改软件这件事很可能被忽略，因为 C语言的老语法关闭了对函数参数的类型检查。

C++ 是一个更庞大的语言，有最新的标准，所以它的主流就更难辨别清楚。例如，虽然我们希望STL能够变成主流，但这件事一时半会是不可能实现的。况且当前的一些实现根本就不支持它。

警惕语言的麻烦特性。我们已经提过，标准里常常有意遗留下一些东西，不给以定义或者不加以清楚的说明，通常这是为了给写编译系统的人更大的自由度。这种东西的列表实在太长了。

数据类型的大小。在C和C++ 里，基本数据类型的大小并没有明确定义，给出的仅仅是下面这些规则：

```
sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
sizeof(float) ≤ sizeof(double)
```

此外，还规定char至少必须有8位， short和int至少是16位， long至少应该是32位。这里有许多不加保证的性质。甚至没要求一个指针值应该能够放进一个 int中。

很容易确定在一个特定编译系统里的各种类型的大小：

```
/* sizeof: display sizes of basic types */
int main(void)
{
    printf("char %d, short %d, int %d, long %d,",
        sizeof(char), sizeof(short),
        sizeof(int), sizeof(long));
    printf(" float %d, double %d, void* %d\n",
        sizeof(float), sizeof(double), sizeof(void *));
    return 0;
}
```

在我们正常使用的大部分机器上，输出都是一样的：

```
char 1, short 2, int 4, long 4, float 4, double 8, void* 4
```

但是完全可能有其他情况。例如，在某些 64位机器上产生的是：

```
char 1, short 2, int 4, long 8, float 4, double 8, void* 8
```

在早期的PC机上，典型的输出是：

```
char 1, short 2, int 2, long 4, float 4, double 8, void* 2
```

China-pub.com 下载

对于早期的PC机，硬件支持多种指针。为处理这些麻烦事，人们发明了一些指针修饰符，如far和near等，它们都不是标准的，但这些魔鬼保留字仍然在纠缠着当前的编译系统。如果在你的编译系统里基本类型的大小能够改变，或者你使用几种有着不同数据类型大小的机器，那么就应该在这些不同配置之下试试你的程序。

标准头文件sdtdef.h里定义了一些类型，它们对可移植性能有些帮助。这其中最常用的是size_t，它是一个无符号的整数类型，是sizeof运算符的返回类型。有些函数(例如strlen)返回这种类型的值；也有不少函数(如malloc)要求这种类型的参数。

根据从上面这些情况取得的经验，Java对所有基本数据类型都明确地定义了大小：byte为8位，char和short为16位，int是32位，而long是64位。

我们将忽略对浮点数计算方面各种问题的讨论，关于这个问题可以写出整本书。幸运的是，大多数现代机器都支持IEEE的浮点硬件标准，这也使浮点算术的有关特性都合理地定义清楚了。

求值顺序。在C和C++ 语言里，有关表达式中的运算对象、副作用产生以及函数参数的求值顺序都没给出明确定义。例如，赋值语句

```
n = (getchar() << 8) | getchar();
```

这里的第二个getchar也有可能率先执行，表达式的书写顺序不一定就是它们的执行顺序。在语句

```
ptr[count] = name[++count];
```

里，count的增值可能在它被用做ptr的下标之前或者之后完成。同样，在

```
printf("%c %c\n", getchar(), getchar());
```

里，第一个输入字符可能被打印在后面(未必是第一个打印)。在

```
printf("%f %s\n", log(-1.23), strerror(errno));
```

里，errno也可能在log调用之前就求了值。

对于各种表达式如何求值，实际也有些规则。按照定义，在每个分号处，或者到一个函数被实际调用的时刻，所有的副作用或者函数调用[⊖]都必须完成。运算符&&和||总是从左到右执行，而且只执行到表达式的真值能够确定时为止(包括有关副作用，也只到此时为止)。在运算符?:里，条件先被求值(包括副作用)，此后，后面两个表达式中只有一个被求值。

Java对求值的顺序有严格定义，它要求所有的表达式，包括副作用，都严格地从左向右进行。然而，有一本权威性手册中提出建议，不要写“过分”依赖这种行为的代码。这是一个合理建议，如果存在着把Java转换到C或者C++ 的可能性，情况就更是如此，因为C、C++ 都没有如上的保证。语言间的转换是对可移植性的一种极端性测试，虽然这种东西并不太有用。

char的符号问题。char数据类型到底是有符号还是无符号的，C和C++ 并没有对此给出明确规定。在结合了char和int的代码里，这个问题就有可能造成麻烦，例如getchar()函数得到int值，调用它的代码就可能出问题。假设你写了：

```
char c;   /* should be int */
c = getchar();
```

如果char是无符号的，c值将在0和255之间；而如果char是有符号的，对于2补码机器上8位字符的最一般配置情况，c的值将在-128与127之间，这种情况将造成一些影响。例如

⊖ 这个函数调用指的是作为外层函数或运算符的参数的那些函数调用。——译者

China-pub.com
下载

我们用字符作为数组的下标，或者用它去与 EOF做比较(EOF通常在stdio.h里定义为- 1)。在6.1节的一个例子里，我们改正了原来代码的一个边界条件，在那里遇到的就是这种代码。如果char是无符号类型，条件s[i]==EOF将总是假的：

```
?    int i;
?    char s[MAX];
?
?    for (i = 0; i < MAX-1; i++)
?        if ((s[i] = getchar()) == '\n' || s[i] == EOF)
?            break;
?    s[i] = '\0';
```

假设getchar返回EOF，存入s[i]的值将是255(即0xFF，这是把 -1转换到unsigned char所得到的结果)。如果char是无符号的，在与EOF做比较时这个值还是255，这必然导致比较的失败。

即使char是有符号的，上面的代码同样也不正确。在执行中遇到 EOF值时，这里的比较就会成功。但是，在这种情况下正常输入的字节 0xFF也会被当作EOF，从而导致这个循环不正确地结束。所以，无论char的符号情况如何，你都必须把getchar的返回值存入一个int，以便与EOF做比较。下面是按可移植方式写出的同一循环：

```
int c, i;
char s[MAX];

for (i = 0; i < MAX-1; i++) {
    if ((c = getchar()) == '\n' || c == EOF)
        break;
    s[i] = c;
}
s[i] = '\0';
```

Java没有unsigned修饰符。在这里所有的整型都是有符号的，只有 16位char类型是无符号的。

算术或者逻辑移位。在对有符号的量用运算符>>做右移时，这个移位可以是算术的 (符号位将在移位的过程中复制传播)，也可以是逻辑的 (移位中空出的位被自动补 0)。同样，根据从C和C++学到的经验，Java把>>保留作算术右移，为逻辑右移另外提供了一个 >>>。

字节顺序。在类型short、int和long里，字节的顺序并没有规定，具有最低地址的字节可能是最高位的字节，也可能是最低位的字节。这是一种依赖硬件的特性，在本章的后面部分我们将做详细讨论。

结构或类成员的对齐。在结构、类或者联合里，各个成分的对齐方式并没有规定。这里只规定各成分一定按说明的顺序排列。例如，在下面的结构里：

```
struct X {
    char c;
    int i;
};
```

成分i的地址与结构开始位置的距离可能是 2、4或者8个字节。很少有机器允许 int存储在奇数边界上，一般都要求占据 n个字节的基本数据类型存放在 n字节的边界上。例如， double一般是8个字节长，所以需要存储在 8的倍数的地址上。在这之上，写编译程序的人还可能再做进一步调整，例如可能为了执行性能做进一步的强制对齐。

你绝不能假定在一个结构里各成员占着连续的存储区。对齐限制实际上会造成结构中的

China-pub.com 下载

“空洞”，在上面的struct X里，至少存在一个字节的未用空间。空洞的存在说明了一个结构可能比它成员的大小之和更大一些，在不同的机器上又可能具有不同大小。如果你需要分配空间，用以存放上述结构，就必须去申请 sizeof(struct X) 个字节，而绝不应该是sizeof(char)+sizeof(int) 个。

位域。位域对机器的依赖太强，无论如何都不应该用它。

从上面关于危险特征的长表里可以总结出下面的规则，不要使用副作用，除了在很少的几个惯用结构里，例如：

```
a[i++] = 0;
c = *p++;
*s++ = *t++;
```

不要用char与EOF做比较。总使用sizeof计算类型和对象的值。决不右移带符号的值。你所用的数据类型应该足够大，足够存储你希望放在里面的值。

用多个编译系统试验。人们很容易认为自己已经理解了可移植性，但是编译系统能看出某些你没有看到的问题。进一步说，不同编译程序有时会对你的程序有不同的看法，因此，你应该尽量利用这些帮助。打开编译程序所有的警告开关，在同一机器或者不同机器上试用多种编译系统，试用一个C++ 编译系统处理你的C程序。

由于不同编译系统在接受的语言方面可能有差异，所以，即使你的程序能用一个编译系统完成编译，你甚至都无法保证它在语法上是正确的。如果几个编译系统都能接受你的代码，那么你的胜算就大得多。对于这本书里的每个 C程序，我们都在三个互不相干的操作系统(Unix，Plan 9和Windows)上用三个C编译系统和若干C++ 编译系统处理过。这是一个清醒的试验，它确实挖出了数十个移植性错误，而这些问题通过人工的大量仔细检查也没有发现。所有这些错误的更正都是非常简单的。

当然，编译系统本身也会引起可移植性问题，因为它们可能对语言中未加规定的行为做出各自不同的选择处理。即使如此，我们的途径仍然是有希望的。我们应该努力去构造这样的软件，使它们的行为能够独立于具体的系统、环境或者编译之间的差异，而不应该去写那种以某种方式展现这些差异情况的代码。简而言之，我们应该设法回避那些很有可能变动的性质和特征。

8.2 头文件和库

头文件和库提供了许多服务，它们是基本语言的扩充。有关例子包括 C里通过stdio、C++里通过iostream、Java里通过java.io完成的输入和输出等。严格地说，这些都不是语言的组成部分，但它们又是和语言本身一起定义，并被期望作为支持有关语言的环境的一个组成部分。在另一方面，由于库通常都覆盖了范围相当广泛的一组操作，常要处理与操作系统有关的问题，因此也就很容易成为不可移植问题的避风港。

使用标准库。在这里，应该提出与核心语言同样的建议：盯紧标准，特别是其中比较成熟的、构造良好的成分。C语言定义了标准库，其中包括许多函数，它们处理输入输出、字符串操作、字符类检测、存储分配以及另外的许多工作。如果你把与操作系统的交互限制在这些函数的范围内，那么如果要从一个系统搬到另一个，你的代码很有希望还能具有同样的行为方式，执行得很好。不过你也要当心，因为存在许多标准库的实现，其中有些包含了标准里未定义的行为。

China-pub.com
下载

ANSI C没有定义串复制函数strdup，然而许多系统里都提供了它，甚至在那些声明自己完全符合标准的系统里。一个有经验的程序员可能会根据习惯去使用 strdup，完全没意识到它并不在标准之中。而后，当这个系统被移植到某个未提供这个功能的系统上时，程序在编译时就会出问题。这种问题是库引起的移植麻烦中最主要的一类。要解决这类问题，只能靠严格按照标准行事，并要在多种不同环境里测试你的程序。

在头文件或者包定义里声明了标准函数的界面。与头文件有关的一个问题是它们往往非常杂乱，因为它们必须设法在一个文件里同时服侍多种不同的语言。例如，我们常常看到，像stdio.h这样的头文件需要同时为老的C语言、ANSI C和C++ 编译程序服务。在这种情况下，文件里到处都散布着#if和#ifdef一类的条件编译指示符号。由于语言预处理程序并不很灵活，这些文件常常都非常复杂，有时可能还包含着错误。

这里是从我们的某系统里摘录的一段，它比其他许多类似的东西还好一些，因为它具有很好的格式：

```
#ifdef _OLD_C
   extern int fread();
   extern int fwrite();
#else
#  if defined(__STDC__) || defined(__cplusplus)
     extern size_t fread(void*, size_t, size_t, FILE*);
     extern size_t fwrite(const void*, size_t, size_t, FILE*);
#  else /* not __STDC__ || __cplusplus */
     extern size_t fread();
     extern size_t fwrite();
#  endif /* else not __STDC__ || __cplusplus */
#endif
```

虽然这个例子相对而言是清晰的，它也确实印证了我们前面的说法，像这样的头文件 (和程序)的结构过于复杂，很难进行维护。针对每个编译系统或者环境建立一个独立的头文件，事情可能更容易些。这样就要求维护一组文件，但其中的每一个都是自足的，适应一个特定系统。这样做也减少了像在严格的ANSI C环境里包括strdup这一类的错误。

头文件还可能“污染”名字空间，因为它里面的某个函数可能正好与程序里的函数同名。例如，我们的weprintf原来被称为wprintf，但是后来发现，在一些环境里根据新的 C标准在stdio.h里定义了一个函数，用的也是这个名字。我们只好修改自己函数的名字，以便能在这种系统里完成编译，同时也是为未来做点准备。如果遇到的问题源于一个错误的实现，而不是规范的合法变化，我们就会想办法绕过它，可以采用的方法是在引入头文件时对有关名字重新做定义：

```
/* some versions of stdio use wprintf so define it away: */
#define wprintf stdio_wprintf
#include <stdio.h>
#undef wprintf
/* code using our wprintf() follows... */
```

这样做的效果，是把头文件里所有的 wprintf都映射到stdio_wprintf，使它们不会再与我们的函数发生冲突。在此之后，我们就可以用原来的 wprintf，不必再改名了。这种写法有些臃肿，而且还付出了额外代价，与程序连接的库将会调用我们的 wprintf函数，而不是调用原来的那个。对于一个函数而言，这可能不必特别担心。但是，确实有些系统给出的环境是非常混乱的，我们必须尽可能地保持代码的清晰性。应该用注释说明这个结构到底

China-pub.com 下载

做了些什么，绝不能再用条件编译把它弄得更糟糕了。如果发现在有的环境里定义了wprintf，那么就应该假定所有的环境里都有这种定义。这样，这个修改就是永久性的，你完全不需要再去维护那些#ifdef语句。当然，更简单的方式是绕道而行，而不是去做斗争，这样做也更安全些。这也就是我们做的事，把函数名字改成weprintf。

即使你总能严格地按规矩办事，环境本身也非常干净，仍然很容易走出限定的范围，例如无意识地假定某些自己喜欢的性质在所有地方都对等等。这方面的例子如，ANSI C定义了6个信号，函数signal能够捕捉到它们。而在POSIX里定义了19个，大部分Unix系统支持32个或者更多的信号。如果你想要用一个非ANSI信号，这很明显就牵涉到功能和可移植性之间的权衡问题，你必须决定哪方面对自己更重要。

目前还存在许多其他标准，它们又不是程序语言定义的组成部分。这方面的例子包括操作系统和网络界面、图形界面，以及许多其他类似的东西。这其中有些东西试图跨越多个系统，例如POSIX；另一些则是为某个特定系统度身打造的，例如各种不同的Microsoft Windows API。与上面类似的建议也适用于这些方面。如果你选择广泛适用的具有良好构造的标准，如果你能盯住最核心的使用最广泛的特性，你的程序就能更具可移植性。

8.3 程序组织

达到可移植性的方式，最重要的有两种，我们将把它们称为联合的方式和取交集的方式。联合方式使用各个特殊途径的最佳特征，采用条件式的编译和安装，根据各个具体环境的特殊情况分别进行处理。这样，结果代码是所有方案的一种联合，它可以利用各系统在能力方面的优点。这种方式的缺点包括：安装过程的规模和复杂性，由代码中大量费解的编译条件造成的复杂性等等。

只使用到处都可用的特征。我们建议采用取交集的方式，即：只使用那些在所有目标系统里都存在的特性，绝不使用那些并不是到处都能用的特征。强求使用普遍可用特性也有危险性，这可能限制了目标系统的范围，或者限制了程序的功能。此外，也可能在某些系统里导致性能方面的损失。

为了比较这两种不同方式，我们来看一些使用联合方式的例子，以及采用交集方式对它们重新进行整理的情况。正如你将要看到的，联合方式的代码从设计上看根本就是不可移植的，虽然它们声称可移植性是自己的目标；而交集代码不仅是可移植的，通常也更加简单。

下面是个小例子，这里试图处理环境中因为某些原因而没有标准头文件stdlib.h的情况：

```
?    #if defined (STDC_HEADERS) || defined (_LIBC)
?    #include <stdlib.h>
?    #else
?    extern void *malloc(unsigned int);
?    extern void *realloc(void *, unsigned int);
?    #endif
```

如果偶然用用的话，这种防御式测试还是可以接受的，但频繁地这样做就很不好了。这里也提出了另一个问题：到底有多少stdlib函数最后出现在这种形式的或者其他类似形式的条件代码里。如果在程序里用到了malloc或者realloc，那么肯定也需要用其他的函数，例如free。如果unsigned int的大小与size_t(这是malloc和realloc参数的正确类型)不一样，那么又会出什么问题？进一步说，我们怎么知道STDC_HEADERS或_LIBC确实已经定

China-pub.com
下载

义了，而且定义正确？怎么保证绝不会有其他名字能在某种环境里启动这里的代换？任何像这样的条件代码都是不完全的、不可移植的，因为总会遇到某个系统不能与这里的条件协调，这时我们就必须重新编辑这些#ifdef。如果能不通过这类条件编译解决问题，我们就能够根除这些在程序维护中最令人头疼的事情。

然而，这个例子力图解决的问题确实是存在的，那么，怎样做才能一劳永逸地解决它呢？我们认为，宁可事先假设标准头文件是存在的。如果确实没有的话，那就是其他人的问题了。而如果实际情况就是没有，那么更简单的办法是与本软件一起发送一个头文件，在其中定义函数malloc、realloc和free，与ANSI C定义它们的形式完全相同。在程序里总包含这个文件，而不是在代码中到处打上面这样的绷带。这样，我们就能知道必要的界面总是可用的。

避免条件编译。使用#ifdef和其他类似预处理指示写的条件编译是很难管理的，因为在这种情况下有关信息趋向于散布在整个源文件里。

```
#ifdef NATIVE
    char *astring = "convert ASCII to native character set";
#else
#ifdef MAC
    char *astring = "convert to Mac textfile format";
#else
#ifdef DOS
    char *astring = "convert to DOS textfile format";
#else
    char *astring = "convert to Unix textfile format";
#endif /* ?DOS */
#endif /* ?MAC */
#endif /* ?NATIVE */
```

在这个摘录中，最好是在各定义之后用 #endif，而不是在最后堆积一批#endif。但是，实际问题是，无论写程序时的动机如何，这段代码都是高度不可移植的，因为它对每个系统的行为不同，对每个新环境必须再写一个新的 #ifdef。用一个串，其中使用一个一般性的词可能更方便，完全是可移植的，而且也提供了同样的信息：

```
char *astring = "convert to local text format";
```

这就不需要任何条件代码，因为它对所有系统都完全一样。

将编译时的控制流(由#ifdef语句确定的)和运行时的控制流混在一起，会使情况变得更坏，因为这种东西极其难读。

```
#ifndef DISKSYS
    for (i = 1; i <= msg->dbgmsg.msg_total; i++)
#endif
#ifdef DISKSYS
    i = dbgmsgno;
    if (i <= msg->dbgmsg.msg_total)
#endif
    {
        ...
        if (msg->dbgmsg.msg_total == i)
#ifndef DISKSYS
            break; /* no more messages to wait for */
        about 30 more lines, with further conditional compilation
#endif
    }
```

China-pub.com 下载

对于那些明显无害的应用，条件编译常常可以用更清晰的形式取代。例如 #ifdef常被用来控制排错代码的执行：

```
?     #ifdef DEBUG
?         printf(...);
?     #endif
```

用一个带常量条件的正规的条件语句也能把事情做得同样好：

```
enum { DEBUG = 0 };
...
if (DEBUG) {
    printf(...);
}
```

如果DEBUG的值是0，大部分编译系统对这段程序不会产生任何目标代码，不过它们会检查被排除代码的语法。与此相反， #ifdef里完全可能隐藏着语法错误，而以后一旦把 #ifdef打开，有关代码就会阻碍编译的执行。

有时人们用条件编译排除掉一大段代码：

```
#ifdef notdef   /* undefined symbol */
    ...
#endif
```

或

```
#if 0
    ...
#endif
```

在编译时采用有条件地替换文件的方式，可以完全避免这种条件代码。下一节里我们还要回到这个问题。

如果需要修改一个程序去适应某个新环境，你不应该以该程序的一个新副本作为出发点。相反，你应该设法调整现存的代码。你可能需要对代码的主体做一些修改。如果采用编辑程序副本的方式，慢慢地你就会做出许多发散的版本。对于一个程序，只要可能，应该只存在惟一的一套源文件。如果你发现某些东西需要改变，以便把程序移植到某个特定的环境去，那么请设法找到一种办法，使改造后的东西在所有地方都能用。如果认为需要，也可以修改内部的界面，但应该保持代码里不出现 #ifdef。这种做法每次都将使你的代码变得更具可移植性，而不是变得更特殊。应该缩小交集，而不是放宽联合。

我们已经说了许多反对条件编译的话，也展示了由它引起的一些问题。但我们还没有提到这其中最恶劣的问题：这种代码几乎是无法测试的。一个 #ifdef实际上把单个的程序变成了两个分别编译的程序，我们很难弄清程序的每个变形是否都已经编译过、测试过。如果在一个#ifdef块里做了修改，那么在其他地方也可能需要做这种修改。要想验证有关的修改，就要求环境条件能够打开对应的 #ifdef。虽然在某些其他配置方式下也需要类似修改，这些情况也不可能检测到。此外，如果程序里需要增加一个新 #ifdef块，我们很难把这种修改孤立出来，很难确定需要满足哪些额外条件才能到达这个地方，很难确定为解决这个问题还要修改哪些地方。最后，如果代码里确有某些东西，按照条件它们将被忽略，那么编译就根本看不到它。这里完全可能是些乱七八糟的东西，而我们却根本就不知道，直到某个不幸的用户试图在某个环境里编译程序，恰巧触发了有关条件。下面的程序当 _MAC有定义时是能够编译的，如果不是这样就会出毛病：

China-pub.com
下载


```
#ifdef _MAC
    printf("This is Macintosh\r");
#else
    This will give a syntax error on other systems
#endif
```

基于上述理由，我们更喜欢只使用那些对所有目标环境都是共同的特性。这样我们就能编译和测试所有代码。如果某些东西产生可移植性问题，我们不是增加条件性代码，而是设法重写代码，设法避免这些问题。沿着这条路走下去，程序的可移植性将逐步增强，其本身也会不断得到改进，而不是变得越来越复杂。

有些大系统在发布时带有一个配置脚本，以便能根据局部环境的情况对代码做一些剪裁。在编译的时候，这个脚本将检测局部环境的各方面特性——头文件和库的位置，字的字节顺序，各种类型的大小，实现者已知可能崩溃的情况 (这虽然出人意料，但却是很常见的)，如此等等，由此生成一套配置参数或者 make文件，以便对有关情况做出正确配置和设置。这些脚本可能很大、很复杂，是软件发布的重要组成部分，需要不断进行维护，以保证它们能完成任务。有的时候这种技术确实是必须的。但是从另一个角度说，代码的可移植性越强，`#ifdef`越少，它的配置和安装也就会越简单、越可靠。

练习8-1 研究你的编译系统如何处理括起来放在条件块里面的代码，例如：

```
const int DEBUG = 0;
/* or enum { DEBUG = 0 }; */
/* or final boolean DEBUG = false; */

if (DEBUG) {
    ...
}
```

在什么情况下它确实做了语法检查？什么时候它产生实际的代码？如果你能用的编译系统不止一个，它们互相比较的情况又如何？

8.4 隔离

我们宁愿能有这样一个源程序，它能在所有系统上编译，不需要做任何修改。不过这常常是不现实的。虽然如此，任由不可移植代码散布在程序的各处仍然是不对的，而这也正是条件编译造成的问题之一。

把系统依赖性局限在独立文件里。如果不同系统需要不同的代码，应该使这种差异局限在独立的文件里，一个文件对应一个系统。例如，文本编辑器 Sam能在Unix、Windows和许多其他系统上运行。这些环境的系统界面差别极大，但是 Sam的绝大部分代码在各处都是一样的。这里对每个特定环境提供一个独立文件，覆盖系统的变化情况。`unix.c`提供到Unix系统的界面代码，而`windows.c`用于Windows环境。这些文件实现了一个到操作系统的可移植界面，掩盖掉它们之间的差别。 Sam实际上是针对它自己的虚拟操作系统写的，这个虚拟操作系统可以移植到各种实际系统上，方法就是写出几百行 C代码，利用可用的系统调用实现十来个小的无法移植的操作。

各种操作系统的图形环境几乎是互不相干的， Sam处理这个问题的办法就是为自己的图形提供一个可移植库。与直接为某个给定系统修改代码相比，建立这种库要做更多的工作 (例如，关联X Window系统的界面代码大约有Sam所有其他部分的一半那么大)，但是从长远看，累计

China-pub.com 下载

起来的工作量则要小得多。作为这个工作的副产品，该图形库本身也很有价值，可以单独使用，并已经把几个其他程序弄得可移植了。

Sam是个很老的程序，今天，各种可移植的图形环境，例如 OpenGL、Tcl/Tk和Java等已经在许多不同平台上可以使用了，利用这些东西写你的代码，而不是用专有图形库，这将使你的程序具有更强的可用性。

把系统依赖性隐藏在界面后面。抽象是一种强有力的技术，应该通过它划清程序的可移植部分与不可移植部分之间的界限。大部分程序设计语言所附带的 I/O库就是一个很好的例子，它们使用可供打开/关闭、读和写的文件概念，从不提及任何物理位置或结构，为二级存储器提供了一种抽象。使用这些界面的程序将能在任何实现了它们的系统上运行。

Sam程序的实现是抽象的另一个例子。这里定义了与文件系统和图形操作相关的界面，程序里只使用界面所提供的特征，而界面本身则可以使用下层系统提供的任何功能。对不同系统而言，界面的实现可能差别很大。但是，只使用界面的程序则与这些情况完全无关，当它需要搬到另一处时根本不需要做任何修改。

Java的可移植途径也是个很好的例子，它也说明了沿着这条路上有可能走多么远。一个Java程序被翻译为一种“虚拟机器”上的一系列操作。所谓虚拟机就是一个模拟的计算机，它可以在任何现实的计算机上实现。 Java的库提供了一套一致的对下层系统特性进行访问的功能，包括图形、用户界面、网络以及其他类似的东西。库将被映射到局部系统提供的功能上。从理论上说，完全可能在任何地方，无须任何改变地运行同一个 Java程序(甚至是翻译之后的程序)。

8.5 数据交换

正文数据很容易从一个系统搬到另一个系统去，这是在不同系统间交换任意信息的最简单的方式。

用正文做数据交换。正文容易用各种工具操作，以原来未曾预计到的方式去处理。例如，如果一个程序的输出并不正好适合做另一个程序的输入，我们可以用一个 Awk或Perl脚本去矫正它；可以用`grep`选择或者删除其中的一些行；可以用你最喜欢的编辑器对它做各种更复杂的修改。正文文件很容易做文档，甚至可能再不需要做文档，因为人完全可以直接阅读它们。正文文件里可以写注释，指明处理这些数据需要什么版本的软件。例如在 PostScript文件里的第一行就说明了它的编码方式：

```
%!PS-Adobe-2.0
```

与此相反，处理二进制文件就需要有专门的工具，甚至在同一台机器上，这些工具常常也不能一起使用。存在许多使用广泛的工具，其作用就是把任意的二进制数据转换成正文，以便能以最不容易出毛病的形式发送出去。这其中包括 Macintosh系统里的`binhex`，Unix系统的`uuencode`和`uudecode`，以及各种各样的为在电子邮件里传递二进制数据而使用的MIME编码工具。在第9章，我们还要给出一组包装和解包例程，它们能用于对二进制数据进行编码，使其能够可移植地进行传递。存在这么多工具，这个情况就足以说明二进制格式存在某些问题。

正文数据交换中也存在一个不和谐音： PC系统使用一个回车符‘`\r`’和一个换行符‘`\n`’来结束一个行，而在Unix系统里只用一个换行符。回车是一种称为电传打字机的古老设备的产

China-pub.com
下载

物，这种设备需要用一个回车 (CR)操作使打字部件回到行的开始，再用另一个独立的换行操作(LF)将它推进到一个新行。

虽然今天的计算机已经根本没有什么车可以回了，大部分 PC软件仍然期望在每行的最后有这种组合(习惯上称为CRLF，读为“ curliff ”)。如果没有回车符，一个文件就可能被当作一个极长的行，行和字符计数都可能出错，或发生不能预期的更改。有些软件能很得体地适应这些情况，但大部分软件都不行。 PC并不是仅有的罪犯，由于一系列兼容性方面的考虑，某些现代的网络标准，例如HTTP，也用CRLF作为行限界符号。

我们建议只使用标准化的界面，这将在任何给定系统上一致地建立 CRLF，无论是(在PC上)输入时去掉\r，到输出时再把它加回去；还是 (在Unix上)什么也不做。对那些必须从这边搬到那边的文件，就需要使用能在两种格式间完成转换的程序。

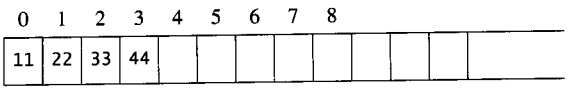
练习8-2 写一个程序从文件中去掉荒谬的回车。写第二个程序把它们加进去，方法是将每个换行符替换为一个回车和一个换行。你将如何测试这些程序？

8.6 字节序

虽然存在着上面讨论里提出的各种缺点，二进制数据有时仍然是必需的，因为它更紧凑，解码也更迅速。在计算机网络领域，二进制形式是许多工作的基础，紧凑和速度都是最根本的原因。但是，二进制数据确实存在许多移植性问题。

至少有一个情况可以确定：所有现代机器都使用 8位字节。但是，不同机器在如何表示大于一个字节的对象时就有许多不同方式，特别依赖某种特定方式就是一个错误。一个短整数(通常是16位，含两个字节)的低位字节可能存储在比其高位字节低的存储地址 (低尾端方式)，或者存在较高的存储地址(高尾端方式)。这方面的决定确实带有随意性，有的机器甚至同时支持两种方式。

这样一来，虽然对采用高尾端或低尾端方式的机器而言，存储器都被看成是某种顺序下的字序列，它们对一个字里各字节的解释却采用了相反的顺序。在下面的图中，从地址 0开始的4个字节表示某个十六进制整数。对高尾端机器而言，它是 0x11223344，而对低尾端机器就是0x44332211。



要想知道实际机器中的字节顺序问题，可以看下面的程序：

```
/* byteorder: display bytes of a long */
int main(void)
{
    unsigned long x;
    unsigned char *p;
    int i;

    /* 11 22 33 44 => big-endian */
    /* 44 33 22 11 => little-endian */
    /* x = 0x1122334455667788UL; for 64-bit long */

    x = 0x11223344UL;
    p = (unsigned char *) &x;
    for (i = 0; i < sizeof(long); i++)
        printf("%x ", *p++);
```

China-pub.com 下载

```
    printf("\n");
    return 0;
}
```

在32位的高尾端机器上，其输出是：

```
11 22 33 44
```

在低尾端机器上，输出就会是：

```
44 33 22 11
```

但是，在PDP-11机器(这是某个时期的一种卓越的机器，现在还可以在许多嵌入式系统中看到)上是：

```
22 11 44 33
```

在具有64位long类型的机器上，只要改变常数的大小，就可以看到类似现象。

这看起来像是一个无聊的问题，但是，如果我们需要把整数通过具有一个字节宽度的界面(例如通过网络连接)传输时，就必须选定首先传输哪个字节，而这个选择必然要牵涉到低尾端/高尾端的问题。换句话说，该程序应该以明显方式做某些事，而这些事在：

```
fwrite(&x, sizeof(x), 1, stdout);
```

中是以隐含方式处理的。如果在一台计算机里写出一个 int(或者short、long)，而要另一台计算机读它，这将是件很不安全的事情。

例如，假设源计算机用下面方式写

```
unsigned short x;
fwrite(&x, sizeof(x), 1, stdout);
```

而接收的计算机用下面方式读

```
unsigned short x;
fread(&x, sizeof(x), 1, stdin);
```

如果两台机器具有不同的字节顺序， x的值将无法保持原样。如果 x在开始时是0x1000，它在到达时可能就是0x0010。

人们通常用条件编译和字节交换来解决这种问题，写的东西大概是：

```
?    short x;
?    fread(&x, sizeof(x), 1, stdin);
?    #ifdef BIG_ENDIAN
?    /* swap bytes */
?    x = ((x&0xFF) << 8) | ((x>>8) & 0xFF);
?    #endif
```

当存在许多二字节和四字节整数需要交换时，这种方式就显得非常笨拙。在实践中，在把这些字节从一个地方传到另一个地方的过程中，可能需要多次完成这类交换。

对short而言，事情已经很不妙了，对于更长的数据类型，情况当然会更糟糕，因为这时存在着更多的排列其中字节的方式。如果再加上结构成员之间的大小可能不同的填充、对齐方面的限制，以及各种老式机器千奇百怪的字节顺序，问题看起来很难对付。

数据交换时用固定的字节序。这里提出一种解决办法：写可移植的代码，总按照正规的顺序写出各个字节：

```
unsigned short x;
putchar(x >> 8);       /* write high-order byte */
putchar(x & 0xFF);     /* write low-order byte */
```

China-pub.com
下载

一次一个字节地读回，把数据重装起来：

```
unsigned short x;
x = getchar() << 8;      /* read high-order byte */
x |= getchar() & 0xFF;   /* read low-order byte */
```

这个方法也可以推广到结构，你只要把结构成员按照一种规定顺序写出去，一次一个字节，丢掉填充。无论你采用什么字节顺序，所有事情都将能够一致地完成。这里惟一的要求是：对于传送中所采用的字节顺序和各种对象的大小，发送方和接收方都应该有同样的认识。在下一章里，我们要给出一对例行程序，它们的功能就是对一般性的数据进行打包和解包。

一次一个字节的处理方式看起来代价很高，但相对于需要打包和解包的 I/O而言，这个代价是微不足道的。考虑X Window系统，在其中客户总按照它自己的字节顺序去写数据，而服务器就必须设法解开客户送来的任何东西。从客户端的角度看，这样做确实能节约几条指令；而服务器就被弄得更大更复杂了，因为它必须同时处理各种可能的字节顺序 —— 它可能要同时应付低尾端和高尾端的客户 —— 在复杂性和代码方面付出的代价是很明显的。另一方面，这又是一个图形环境，在这里包装字节的代价会完全被浸没在它所编码的图形操作的执行之中。

X Window系统在客户方面完全忽略了字节顺序问题，而要求服务器方面能同时处理两种情况。与此相反，Plan 9操作系统则为送到文件服务器(或图形服务器)的信息定义了一种字节顺序，数据采用上面说的那种可移植代码做打包和解包。在实践中，这种操作对运行的影响根本无法察觉，与I/O操作相比，包装数据的代价完全可以忽略不计。

Java是一种比C或C++ 更高级的语言，它完全隐蔽了字节顺序的问题。 Java程序库提供了一个`Serializable`界面，它定义了在交换时数据项的包装方式。

如果你在C或C++ 里工作，那么你就得自己做这件事。采用一次处理一个字节的方式，最关键的收获就是既不需要使用 `#ifdef`，又能对所有8位字节的机器解决问题。我们将在下一章里继续这方面的讨论。

当然，解决问题的最好办法常常还是把信息转换到正文形式，这种形式 (除了CRLF问题外)完全是可移植的，这里根本不存在表达的歧义性问题。当然，这并不一定就是正确的解，因为时间和空间都是极其重要的。有些数据，特别是浮点数据，在传过 `printf`和`scanf`时有可能丢失精度，主要是由于截断问题[⊖]。如果你必须完全精确地交换浮点数据，那么就必须弄清你确有很好的格式化 I/O库。这种库是存在的，不过它可能不是你工作环境的一部分。想要以可移植的方式用二进制形式表示浮点数据非常困难，另一方面，如果你足够细心，通过正文可以做好这件事。

在使用标准库函数处理二进制文件时，还有一个很微妙的可移植问题 —— 必须以二进制方式打开文件：

```
FILE *fin;

fin = fopen(binary_file, "rb");
c = getc(fin);
```

如果忽略了这里的‘ b ’，在所有Unix系统上都不会出任何问题。但是在 Windows系统里，程

⊖ 实际上，更根本的问题是不同数制之间的转换，例如十进制的小数在转换到二进制表示时常常变成无限循环小数，不一定能有精确的对应物。 —— 译者

China-pub.com 下载

序输入中遇到的第一个 control-Z字节(八进制的032，十六进制的1A)将结束有关的读入动作(我们在第5章的strings程序里已经看到这种情况的发生)。在另一方面，如果按二进制模式读入正文文件，会导致\r字节被留在了输入里面，而在输出时又不会自动地产生它。

8.7 可移植性和升级

可移植性问题中还有一个最让人头疼的因素，那就是，系统软件总是在随着时间推移而不断变化，这种变化有可能发生在系统的任何层面上，导致程序的现存版本之间无缘无故地发生互不相容的情况。

如果改变规范就应改变名字。我们最喜欢(如果可以用这个词的话)用的例子是Unix系统中echo命令的性质变化。在开始时，它的设计仅仅是想做对参数的回应：

```
% echo hello, world
hello, world
%
```

但是，到了后来，echo变成了许多shell脚本里的关键部分，希望产生格式化输出的需求变得越来越强烈，所以echo被改造为解释它的参数，从某种意义上看更像printf了：

```
% echo 'hello\nworld'
hello
world
%
```

这种新特性确实很有用，但它也使许多 shell脚本产生了移植性问题，因为它们可能就依赖于echo命令除了回应之外什么也不做。而

```
% echo $PATH
```

的行为方式则依赖于现在所用的 echo是什么版本的。如果不巧在变量值里包含了一个反斜线符号(在DOS或Windows系统里很容易遇到这种情况)，echo可能对它做出某种特殊解释。这个问题类似于用printf(str)或printf("%s",str)产生输出时的差别。请看看，如果在str里正好有一个百分号，会发生什么事情。

我们说的还仅仅是echo故事中的一部分，但是它已经阐明了根本的问题：对系统的修改可能产生不同版本的软件，有时我们有意地使它们在行为方面有些变化，但同时也无意地造成了许多移植性问题，而这些问题常常是很难绕过去的。如果给 echo的新版本另起一个不同的名字，造成的麻烦可能要少得多。

现在再看一个更具启发性的例子。考虑Unix命令sum，它打印出一个文件的大小和它的检验和。下面一串命令是想验证一次信息传递是否完全成功：

```
% sum file
52313 2 file
%
% copy file to other machine
%
% telnet othermachine
$
$ sum file
52313 2 file
$
```

由于传递之后的检验和相同，我们可以合理地推断，新副本和老文件是一样的。

China-pub.com
下载

而后系统被改进了，版本发生了变化，有人发现检验和的算法并不完美，因此就修改了sum，使用了更好的算法。另外的什么人也做了同样研究，给出sum的另一个更好的算法。这样发展下来，到了今天，实际中已经存在着许多不同版本的 sum，每个版本给出的回答都不同。我们把一个文件复制到附近另一台机器上，想看看sum算出的是什么：

```
% sum file
52313 2 file
%
% copy file to machine 2
% copy file to machine 3
% telnet machine2
$
$ sum file
eaa0d468    713    file
$ telnet machine3
>
> sum file
62992   1 file
>
```

是文件破坏了吗？还是仅仅因为我们恰好用到不同版本的sum？或许两者都是？

这样的sum就形成了一个完美的可移植性灾难：一个程序，其目的是为了帮助人们在从一台机器到另一台复制软件时做检查。但是，由于存在许多互不兼容的版本，从原来的意图看，它已经变成毫无意义的东西了。

对于原本要应付的简单工作而言，最早的 sum是很好的，其低技术的检验和算法也是适宜的。虽然“修改”它有可能造就出一个更好的程序，但从中得到的并不多，至少完全不足以抵消这种不相容的代价。问题不在于性能提升，而在于互不相容的程序又偏偏用了同样名字。这种变化带来的版本问题还会折磨我们许多年。

维护现存程序与数据的相容性。当一个软件(例如一个字处理系统)的新版本发布时，新版本通常都能读入由较早版本产生的文件。我们也可以断定，由于增加了新的原来没有的特征，文件格式也可能有变化。但是，新版本常常没有提供一种方式，使之能写出原来格式的文件。这样，新版本的用户，即使他们没有使用新版本中的任何新特征，也无法与那些使用旧版本的人们共享文件。这将迫使每个人都去升级。无论是作为一个工程观点，还是作为一种市场策略，这种设计都是特别令人遗憾的。

向后兼容是使程序符合其过去规范的一种能力。如果你打算修改一个程序，那就应该保证你没有破坏老的程序和依赖于它的数据。应该正确地修改文档，提供一些办法去恢复原来的行为方式。最重要的是，应该仔细考虑你计划做的改变是不是真正的改进，与你将引进的不可移植性的代价相比，是不是真正值得去做。

8.8 国际化

生活在美国的人很容易忘记英语并不是惟一的语言， ASCII不是惟一的字符集， $也不是仅有的钱币符号，写日期时可以把日子写在前面，时间可以采用 24小时的钟点，如此等等。所以，可移植性的另一方面，更广泛地说，是要处理程序在跨越语言和文化边界时的可移植性问题。这是一个内涵极其丰富的题目，我们只能用很有限的篇幅指出其中的一些基本考虑。

国际化是个术语，意指设法使一个程序并不对其运行的文化环境有任何假定。这方面的

China-pub.com 下载

问题非常多，从所用的字符集，到界面上图标的解释信息等。

不要假定是ASCII。在世界上许多地方，字符集都比ASCII丰富得多。在ctype.h里提供的标准字符检测函数通常能够掩盖这些差异：

```
if (isalpha(c)) ...
```

这样就能独立于字符的特定编码方式。进而，在那些字母比从 a到z更多几个或者少几个的地方，如果程序在那里编译，它也应该能正常工作。当然，甚至 isalpha这个名字也表达了它自己的来历，因为在有些语言里根本没有字母表。

ASCII编码只定义了直到0x7F的值(7位)，大多数欧洲国家都对此做了扩充，增加了另外一些字符，以便表示他们语言中的字母。 Latin-1编码集在西欧使用很广泛，它就是ASCII的一个超集，其中把80到FF的字节值定义为一些符号和带调字母。例如编码 E7表示ç。英语单词boy在ASCII(和Latin-1)里用三个字节表示，按十六进制写是 62 6F 79，而法文词garçon在Latin-1里用字节67 61 72 E7 6F 6E表示。其他语言还定义了另外的符号。但是这些东西不可能全都放进 ASCII没用的那128个值里。因此，现实中存在许多互相冲突的标准，它们给80到FF的字节指定了不同字符。

有些语言根本就无法放进8位的字节里。各种主要的亚洲语言都有成千上万的字符，中国、日本和韩国用的编码都采用每个字符 16位的方式。由此产生的结果是，要在一台某种语言的计算机系统上阅读以另一种语言书写的文档，就会出现一个大的移植性问题。假定到来的字符本身并未受到任何损害，在一台美国计算机上读一个中文文档，至少也要涉及到特殊的软件和字型。如果我们希望能同时使用中文、英文和俄文，面临的障碍将是十分明显的。

Unicode字符集是人们希望改善这种状况的一个尝试，它为全世界所有的语言提供了一套统一编码。Unicode与ISO 10646标准的16位子集相兼容，这里对每个字符用 16位做编码，其中的00FF及比它小的值对应于 Latin-1。这样，法文单词 garçon用16位值0067 0061 0072 00E7 006F 006E表示。西里尔字符集的编码占的位置是0401到04FF，所有表意符号语言的字符占据从3000开始的大片位置。在Unicode里包含了所有重要的语言，还有许多不那么重要的语言。因此，如果要在不同国家之间传递文件，或者需要存储多语言的文本，Unicode是一种很合适的编码选择。 Unicode已经在Internet上逐渐流行起来，有些系统甚至把它作为标准加以支持，例如 Java语言就用 Unicode作为它字符串的基本字符集。 Plan 9和Inferno操作系统都完全使用了Unicode，甚至对文件名和用户名也是这样。 Microsoft Windows支持Unicode字符集，但却不是强制性的要求，大部分 Windows应用仍然是在ASCII里工作得最好，不过实际情况正在迅速向Unicode方面发展。

Unicode也带来了一个问题：字符不再能放进字节里。因此， Unicode文本也会遇到字节顺序问题的滋扰。为了避免这种状况， Unicode文档在程序间或者通过网络进行传递之前，通常都先行转换为一种称为UTF-8的字节流编码形式。每个16位字符被编码为1个、2个或3个字节的一个序列，专门用于传输。 ASCII字符集仍然用从00到7F的值表示，在使用UTF-8时，所有这些字符都被放在一个字节里，所以 UTF-8对ASCII具有向后兼容性。位于80和7FF之间的值用两个字节表示， 800以及比它更大的值用三个字节表示。单词 garçon在UTF-8里表示为字节67 61 72 C3 A7 6F 6E，其中的Unicode值E7，即字符ç，在UTF-8里用两个字节C3 A7表示。

UTF-8对ASCII的向后兼容性是极其重要的。因为这就能保证，把正文当作不加解释的字节流的程序能在任何语言里对 Unicode工作。我们在UTF-8编码的多种语言正文上试验了第 3

China-pub.com
下载

章的Markov程序，包括俄文、希腊文、日文和中文。程序运行都没有问题。对于欧洲语言，它们的词都是由ASCII的空格、制表符或换行符分隔，程序输出的是合理的无意义句子。对于其他语言，我们就必须对词的分割规则做必要的修改，以设法得到在精神上接近于程序意图的输出。

C和C++语言支持“宽字符”，这是16位或者更大的整数。另外有几个伴随而来的函数，可用于处理Unicode或者其他大字符集的字符。宽字符串文字应写成 L"..."，它们也带来了进一步的移植性问题：一个用宽字符串的程序只能在那些能够使用该字符集的显示器上使用，只有这样人们才能够理解。为了在机器间可移植地进行传输，这种字符必须首先转换到字节流，例如UTF-8之类的东西，C提供了把宽字符转换到字节和反向转换的函数。但是我们应该用什么转换呢？对字符集的解释、对字节流编码的定义都隐蔽在程序库里，很难弄清楚。这种情况至少不是很令人满意的。或许在美好未来的某个时候，人们在使用的字符集上取得了一致，但是，随后出现的场面很可能又会使人回想起今天仍在折磨着我们的字节序问题。

不要假定是英语。界面的构建者必须记住，不同的语言在谈论同一事情时所使用的字符数目经常有很大差异。所以，在屏幕上和数组里都必须留有足够的空间。

错误信息该怎么办？至少说，它们必须不包含任何仅仅在某个特殊人群中才能理解的暗语和行话，用简明的语言写出它们不过是个开始。人们广泛采用的一种技术是把所有信息正文汇集到同一个地方，这样，在需要翻译到其他语言时就能方便地替换它们。

现实中存在着许多与文化密切相关的东西，例如， mm/dd/yy的日期格式表示只在北美使用。如果一个软件有一点用到其他国家去的可能性，就应该设法避免这类文化依赖性，或应设法将其减到最少。图形界面上的图标常常也带有文化依赖性，许多图标把预期使用环境中的本地人都弄得莫名其妙，更不用说具有其他文化背景的人们了。

8.9 小结

可移植代码是一个非常值得去追求的理想，因为有如此多的时间被浪费在修改程序方面，无论是把程序从一个系统移到另一个系统，还是为了它本身演化的需要，或是因为它运行的系统发生了变化，在这些情况下需要设法维持程序的继续运行。当然，可移植性不是随便就能得到的，它要求实现中的特别注意，也需要开发者具有对所有的潜在目标系统在可移植问题方面的知识。

我们已经指出了追求可移植性的两种途径，即联合和交集。联合途径相当于为在每个目标系统上工作而写一个版本，利用条件编译一类的机制，把这些代码尽可能地汇集在一起。这种途径的缺点很多，它造成过多的代码，而且常常是很多非常复杂的代码。它很难更新，也很难测试。

交集途径是设法以一种形式写出尽量多的代码，使它能在每种系统上运行而不需要做任何修改。把无法逃避的系统依赖性封装在独立的源文件里，其作用就像是程序与基础系统之间的界面。交集方法也有缺点，包括可能存在性能方面，甚至特征方面的损失。但是从长远的观点看，这种途径的利大于弊。

补充阅读

对于程序设计语言有许多描述，但是，在这之中只有很少是足够精确的，能够作为定义

China-pub.com 下载

性的参考手册。作为作者个人的偏爱，倾向于提出由 Brian Kernighan和Dennis Ritchie写的《C程序设计语言》(The C Programming Language，Prentice Hall，1988)。但它并不是语言标准的替代品。由 Sam Harbison和Guy Steele写的《C：参考手册》(C: A Reference Manual，Prentice Hall，1994)现在已经是第4版，在其中包括了许多关于可移植性的很好建议。官方的C和C++ 标准可以由 ISO(国际标准化组织)得到。与 Java的官方标准最接近的东西是 James Gosling、Bill Joy和Guy Steele的《Java语言规范》(Java Language Specification，Addison Wesley，1996)。

Rich Stevens的《Unix环境高级编程》[⊖](Advanced Programming in the Unix Evironment，Addison Wesley，1992)对于Unix程序员而言是绝好的资源，它提供了有关在不同 Unix变形间的可移植性问题的丰富材料。

POSIX(可移植操作系统界面， Portable Operating System Interface)是一个基于Unix的命令和程序库的国际标准定义。它提供了标准化的环境、有关应用的源代码可移植性以及一个到I/O、文件系统和进程的统一界面。 IEEE出版了描述它的丛书。

术语“ 高尾端 ”可以追溯到 1726年的Jonathan Swift。Danny Cohen的文章《论圣战以及对和平的祈祷》(On holy wars and a plea for peace，IEEE Computer，1981年10月)是一篇关于字节顺序的美妙寓言，它把“ 尾端 ”这个术语引进计算机界。

贝尔实验室开发的 Plan 9系统把可移植性作为其核心议题。这个系统可以从完全没有`#ifdef`的源程序编译到多种处理器，它从根本上使用了 Unicode字符集。Sam的第一个描述出现在“ 文本编辑器 sam ”(The Text Editor sam)，《软件：实践和经验》(Software：Practice and Experience)卷17，第11期，pp813~845页，其最新版本使用了 Unicode，而且可以运行在许多系统上。在Rob Pike和Ken Thompson的文章“ Hello World or Καλημέρακοσμε orこんにちは世界 ”(Proceedings of the Winter 1993 USENIX Conference，1993年冬季USENIX会议录，圣迭戈， 1993，pp43~50页)里讨论了处理16位字符集，例如Unicode的问题。UTF-8编码也是第一次出现在这篇文章里。这篇文章也可以在贝尔实验室的 Plan 9网络站点找到， Sam的当前版本也保存在那里。

Inferno系统与Java很像，是基于Plan 9的经验，在其中定义了一个虚拟机器，它可以在任何实际机器上实现。它还提供了一种语言 (Limbo)，该语言可以翻译到这种虚拟机上，并使用Unicode作为其基本字符集。在这里还包括一个虚拟操作系统，提供了到一些商品系统的可移植界面。有关工作发表在《Inferno操作系统》(The Inferno Operating System)上，作者是Sean Dorward、Rob Pike、David Leo Presotto、Dennis M. Ritchie、Howard W. Trickey和Philip Winterbottom，刊于《贝尔实验室技术杂志》 (Bell Labs Technical Journal)，卷2，第1期，1997年冬。

⊖ 此书已由机械工业出版社出版。 —— 编者