第 18 章

# 运行时环境

本章，我们将研究由某个特定的编译器为某个特定的计算机所产生的汇编语言代码，目的是学习一些关于这个编译器的运行时环境的几个有趣的内容。我们需要回答的几个问题是“我的运行时环境的限制是什么？”和“我如何使 C 程序和汇编语言程序一起工作？”

## 18.1 判断运行时环境

你的编译器或环境和我们在这里所看到的肯定不同，所以你将需要自己执行类似这样的试验以便找出在你的机器上它们是如何运作的。

第 1 个步骤是从你的编译器获得一个汇编语言代码列表。在 UNIX 系统中，编译器选项-S 使编译器把每个源文件的汇编代码写到一个具有.s 后缀的文件中。Borland 编译器也支持这种选项，不过它使用的是.asm 后缀。请参阅相关文档，获得其他系统的特定细节。

你还需要阅读你的机器上的汇编语言代码。你并不一定要成为一个熟练的汇编语言程序员，但你需要对每条指令的工作过程以及如何解释地址模型有一个基本的了解。一本描述你的计算机指令集的手册是完成这个任务的绝佳参考材料。

本章并不讲授汇编语言，因为这不是本书的要点。你的机器所产生的汇编语言很可能和本书的不一样。但是，如果你编译测试程序，我在这里对本书的汇编语言的解释可能有助于你分析你的机器上的汇编语言，因为这两种汇编程序实现了相同的源代码。

### 18.1.1 测试程序

让我们观察程序 18.1，也就是测试程序。它包含了各种不同的代码段，它们的实现颇有意思。这个程序并没有实现任何有用的功能，但它并不需要如此——我们需要的只是观察编译器为它所产生的汇编代码。如果你希望研究你的运行时环境的其他方面，你可以修改这个程序，包含这些方面的例子。

```
/*
** 判断 C 运行时环境的程序。
*/
```

```
/*
**  静态初始化
*/
int   static_variable = 5;

void
f()
{
      register int  i1, i2, i3, i4, i5,
                    i6, i7, i8, i9, i10;
      register char*c1, *c2, *c3, *c4, *c5,
                   *c6, *c7, *c8, *c9, *c10;
      extern   inta_very_long_name_to_see_how_long_they_can_be;
      double   dbl;
      intfunc_ret_int();
      double   func_ret_double();
      char     *func_ret_char_ptr();

/*
** 寄存器变量的最大数量。
*/
i1 = 1; i2 = 2; i3 = 3; i4 = 4; i5 = 5;
i6 = 6; i7 = 7; i8 = 8; i9 = 9; i10 = 10;
c1 = (char *)110; c2 = (char *)120;
c3 = (char *)130; c4 = (char *)140;
c5 = (char *)150; c6 = (char *)160;
c7 = (char *)170; c8 = (char *)180;
c9 = (char *)190; c10 = (char *)200;

/*
**  外部名字
*/
a_very_long_name_to_see_how_long_they_can_be = 1;

/*
**  函数调用/返回协议，堆栈帧（过程活动记录）
*/
i2 = func_ret_int( 10, i1, i10 );
dbl = func_ret_double();
      c1 = func_ret_char_ptr( c1 );
}

int
func_ret_int( int a, int b, register int c )
{
      intd;

      d = b - 6;
      return a + b + c;
}

double
func_ret_double()
{
      return 3.14;
}
```

```
char *
func_ret_char_ptr( char *cp )
{
        return cp + 1;
}
```

**程序 18.1　测试程序**　　runtime.c

程序 18.2 的汇编代码是由一台使用 Motorola 68000 处理器家族的计算机产生的。我对代码进行了编辑，使它看上去更清晰，我还去掉了一些不相关的声明。

这是一个很长的程序。和绝大部分的编译器输出一样，它没有包含帮助读者阅读的注释。但你不要被它吓倒！我将逐行解释绝大部分代码。我采用的方法是分段解释，先显示一小段 C 代码，后面是根据它产生的汇编代码。完整的代码列表只是作为参考而给出，这样你可以观察所有这些小段例子是如何组成一个整体的。

```
        .data
        .even
        .globl    _static_variable
_static_variable:
        .long     5
        .text

        .globl    _f
_f:     link a6,#-88
        moveml    #0x3cfc,sp@
        moveq     #1,d7
        moveq     #2,d6
        moveq     #3,d5
        moveq     #4,d4
        moveq     #5,d3
        moveq     #6,d2
        movl      #7,a6@(-4)
        movl      #8,a6@(-8)
        movl      #9,a6@(-12)
        movl      #10,a6@(-16)
        movl      #110,a5
        movl      #120,a4
        movl      #130,a3
        movl      #140,a2
        movl      #150,a6@(-20)
        movl      #160,a6@(-24)
        movl      #170,a6@(-28)
        movl      #180,a6@(-32)
        movl      #190,a6@(-36)
        movl      #200,a6@(-40)
        movl      #1,_a_very_long_name_to_see_how_long_they_can_be
        movl      a6@(-16),sp@-
        movl      d7,sp@-
        pea       10
        jbsr      _func_ret_int
        lea       sp@(12),sp
        movl      d0,d6
        jbsr      _func_ret_double
        movl      d0,a6@(-48)
```

```
        movl    d1,a6@(-44)
        pea     a5@
        jbsr    _func_ret_char_ptr
        addqw   #4,sp
        movl    d0,a5
        moveml  a6@(-88),#0x3cfc
        unlk    a6
        rts

        .globl  _func_ret_int
_func_ret_int:
        link    a6,#-8
        moveml  #0x80,sp@
        movl    a6@(16),d7
        movl    a6@(12),d0
        subql   #6,d0
        movl    d0,a6@(-4)
        movl    a6@(8),d0
        addl    a6@(12),d0
        addl    d7,d0
        moveml  a6@(-8),#0x80
        unlk    a6
        rts

        .globl  _func_ret_double
_func_ret_double:
        link    a6,#0
        moveml  #0,sp@
        movl    L2000000,d0
        movl    L2000000+4,d1
        unlk    a6
        rts
L2000000:.long      0x40091eb8,0x51eb851f

        .globl  _func_ret_char_ptr
_func_ret_char_ptr:
        link    a6,#0
        moveml  #0,sp@
        movl    a6@(8),d0
        addql   #1,d0
        unlk    a6
        rts
```

**程序 18.2　测试程序的汇编语言代码**　　runtime.s

### 18.1.2　静态变量和初始化

测试程序所执行的第 1 项任务是在静态内存中声明并初始化一个变量。

```
/*
**  静态初始化
*/
int         static_variable = 5;
```

```
.data
.enen
.global  _static_variable
_static_variable:
.long    5
```

汇编代码的一开始是两个指令，分别表示进入程序的数据区以及确保变量开始于内存的偶数地址。68000 处理器要求边界对齐。然后变量被声明为全局类型。注意变量名以一个下划线开始。许多（但不是所有）C 编译器会在 C 代码所声明的外部名字前加一个下划线，以免与各个库函数所使用的名字冲突。最后，编译器为变量创建空间，并用适当的值对它进行初始化。

### 18.1.3　堆栈帧

接下来是函数 f。一个函数分成三个部分：**函数序**(prologue)、**函数体**(body)和**函数跋**(epilogue)。函数序用于执行函数启动需要的一些工作，例如为局部变量保留堆栈中的内存。函数跋用于在函数即将返回之前清理堆栈。当然，函数体是用于执行有用工作的地方。

```
void
f()
{
        register int    i1, i2, i3, i4, i5,
                        i6, i7, i8, i9, i10;
        register char   *c1, *c2, *c3, *c4, *c5,
                        *c6, *c7, *c8, *c9, *c10;
        extern  int     a_very_long_name_to_see_...
        double  dbl;
        int     func_ret_int( );
        double  func_ret_double();
        char    *func_ret_char_ptr( );
```

```
        .text
        .globl    _f
_f:     link      a6, #-88
        moveml    #0x3cfc,sp@
```

这些指令的第 1 条表示进入程序的代码（文本）段，紧随其后的是函数名的全局声明。注意在名字前面也有一条下划线。第 1 条可执行指令开始为函数创建**堆栈帧**(stack frame)。堆栈帧是堆栈中的一个区域，函数在那里存储变量和其他值。link 指令将在稍后详细解释，现在你只需要记住它在堆栈中保留了 88 个字节的空间，用于存储局部变量和其他值。

这个代码序列中的最后一条指令把选定寄存器中的值复制到堆栈中。68000 处理器有 8 个用于操纵数据的寄存器，它们的名字是从 d0 至 d7。还有 8 个寄存器用于操纵地址，它们的名字是从 a0 至 a7。值 0x3cfc 表示寄存器 d2 至 d7、a2 至 a5 中的值需要被存储，这些值就是前面提到的“其他值”。稍后你就会明白为什么这些寄存器的值需要进行保存。

局部变量声明和函数原型并不会产生任何汇编代码。但如果任何局部变量在声明时进行了初始化，那么这里也会出现指令用于执行赋值操作。

### 18.1.4　寄存器变量

接下来便是函数体。测试程序的这部分代码的目的是判断寄存器里可以存储多少个变量。它声明了许多寄存器变量，每个都用不同的值进行初始化。汇编代码通过显示每个值在何处存储来回答这个问题。

```
/*
**  寄存器变量的最大数量。
*/
i1 = 1; i2 = 2; i3 = 3; i4 = 4; i5 = 5;
i6 = 6; i7 = 7; i8 = 8; i9 = 9; i10 = 10;
c1 = (char *)110; c2 = (char *)120;
c3 = (char *)130; c4 = (char *)140;
c5 = (char *)150; c6 = (char *)160;
c7 = (char *)170; c8 = (char *)180;
c9 = (char *)190; c10 = (char *)200;
```

```
moveq    #1,d7
moveq    #2,d6
moveq    #3,d5
moveq    #4,d4
moveq    #5,d3
moveq    #6,d2
movl     #7,a6@(-4)
movl     #8,a6@(-8)
movl     #9,a6@(-12)
movl     #10,a6@(-16)
movl     #110,a5
movl     #120,a4
movl     #130,a3
movl     #140,a2
movl     #150,a6@(-20)
movl     #160,a6@(-24)
movl     #170,a6@(-28)
movl     #180,a6@(-32)
movl     #190,a6@(-36)
movl     #200,a6@(-40)
```

整型变量首先进行初始化。注意值 1 至 6 被存放在数据寄存器，但 7 至 10 却被存放在其他地方。这段代码显示了最多只能有 6 个整型值可以被存放在数据寄存器。那么其他不是整型的数据又如何呢？有些编译器不会把字符型变量存放在寄存器中。在有些机器上，double 的长度太长，无法存放在寄存器中。有些机器具有特殊的寄存器，用于存放浮点值。我们可以很容易地对测试程序进行修改来发现这些细节。

接下来的几条指令对指针变量进行初始化。前 4 个值被存放在寄存器，最后那个值被存放在其他地方。因此，这个编译器最多允许 4 个指针变量存放在寄存器中。那么其他类型的指针变量又是如何呢？同样，我们也需要进行试验。但是，在许多机器上，不管指针指向什么类型的东西，它的长度是固定的。所以你可能会发现任何类型的指针都可以存放在寄存器中。

那么其他变量存放在什么地方呢？机器使用的地址模型执行间接寻址和索引操作。这种组合工作颇似数组的下标引用。寄存器 a6 称为**帧指针**(frame pointer)，它指向堆栈帧内部的一个“引用”位置。堆栈帧中的所有值都是通过这个引用位置再加上一个偏移量进行访问的。a6@(-28)指定了一个偏移地址-28。注意偏移位置从-4 开始，每次增长 4。这台机器上的整型值和指针都占据 4 个字节的内存。使用这些偏移地址，你可以建立一张映射表，准确地显示堆栈中的每个值相对于帧指针 a6 的位置。

我们已经见到寄存器 d2 至 d7、a2 至 a5 用于存放寄存器变量，现在很清楚为什么这些寄存器需要在函数序中进行保存。函数必须对任何将用于存储寄存器变量的寄存器进行保存，这样它们原先的值可以在函数返回到调用函数前恢复，这样就能保留调用函数的寄存器变量。

关于寄存器变量最后还要提一点：为什么寄存器 d0-d1、a0-a1 以及 a6-a7 并未用于存放寄存器

变量呢？在这台机器上，a6 用作帧指针，而 a7 是堆栈指针（这个汇编语言给它取了个别名 sp）。后面有个例子将显示 d0 和 d1 用于从函数返回值，所以它们不能用于存放寄存器变量。

但是，在这个程序的代码里并没有明确显示 a0 或 a1 的用途。显而易见的结论是它们将用于某种目的，但这个测试程序并不包含这种类型的代码。要回答这个问题需要进行进一步的试验。

### 18.1.5　外部标识符的长度

接下来的测试用于确定外部标识符所允许的最大长度。这个测试看上去够简单了：用一个长名字声明并使用一个变量，看看会发生什么。

```
/*
**  外部名字
*/
a_very_long_name_to_see_how_long_they_can_be = 1
```

```
movl   #1, a_very_long_name_to_see_how_long_they_can_be
```

从这段代码似乎可以看出，名字的长度并没有限制。更精确地说，这个名字未超出限制。为了找出这个限制，你可以不断加长这个名字，直到发现汇编程序把这个名字截短。

**警告：**

事实上，这个测试是不够充分的。外部名字的最终限制是链接器施加的，它很可能愉快地接受任何长度的名字但忽略除前几个字符以外的其他字符。标准要求外部名字至少区分前 6 个字符（但并不要求区分大小写）。为了测试链接器做了些什么，我们只要简单地链接程序并检查一下结果的装入映像表(load map)和名字列表。

### 18.1.6　判断堆栈帧布局

运行时堆栈保存了每个函数运行时所需要的数据，包括它的自动变量和返回地址。接下来的几个测试将确定两个相关的内容：堆栈帧的组织形式，调用和从函数返回的协议。它们的结果显示了如何提供 C 和汇编程序的接口。

#### 一、传递函数参数

这个例子从调用一个函数开始。

```
/*
** 函数调用/返回协议、堆栈帧。
*/
i2=func_ret_int(10,i1,i10);
```

```
movl      a6@(-16), sp@-
movl      d7, sp@-
pea       10
jbsr      _func_ret_int
```

前 3 条指令把函数的参数压入到堆栈中。被压入的第 1 个参数存储于 a6@(-16)：这个我们原先

讨论过的偏移地址显示这个值就是变量 i10。然后被压入的是 d7，它包含了变量 i1。最后一个参数的压入方式和前两个不同。pea 指令简单地把它的操作数压入到堆栈中，这是一种高效的压入字面值常量的方法。为什么参数要以它们在参数列表中的相反次序逐个压到堆栈中？我们很快就能找到这个答案。

这些指令一开始创建属于即将被调用的函数的堆栈帧。通过跟踪指令并记住它们的效果，我们可以勾勒一幅关于堆栈帧的完整的图。如果你需要从汇编语言的层次追踪一个 C 程序的执行过程，这幅图可以向你提供一些有用的信息。图 18.1 显示了到目前为止所创建的内容。图中显示低内存地址位于顶部而高内存地址位于底部。当值压入堆栈时，堆栈向低地址方向生长（向上）。在原先的堆栈指针以下的堆栈内容是未知的，所以在图中以一个问号显示。

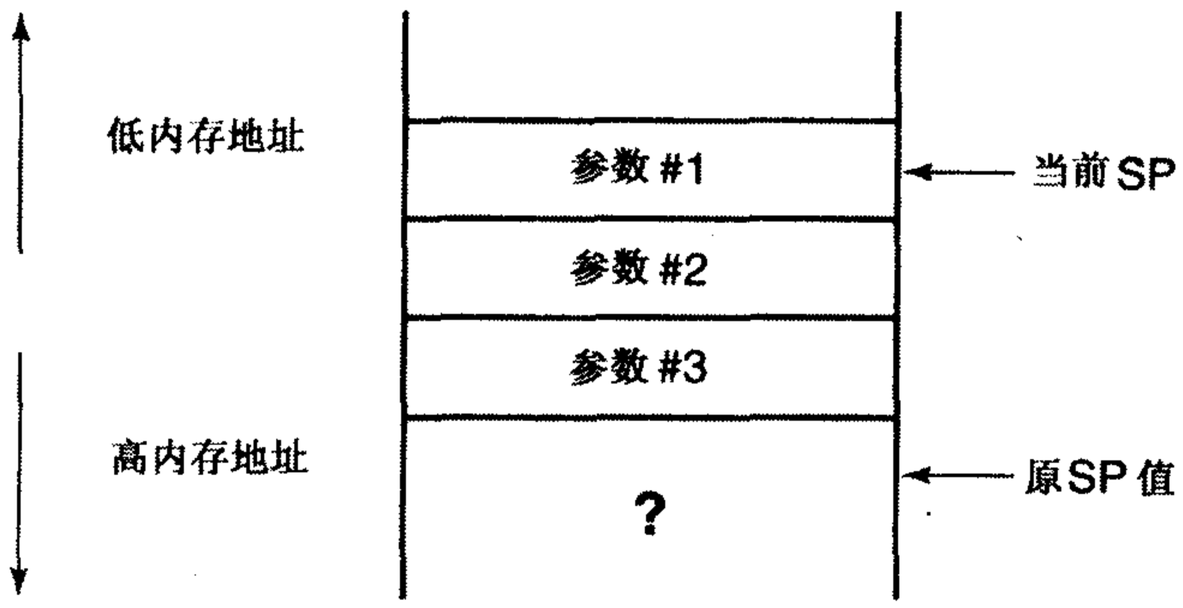


图 18.1　压入参数后的堆栈帧

接下来的指令是一个“跳转子程序(jump subroutine)”。它把返回地址压入到堆栈中，并跳转到 _func_ret_int 的起始位置。当被调用函数结束任务后需要返回到它的调用位置时，就需要使用这个压入到堆栈中的返回地址。现在，堆栈的情况如图 18.2 所示。

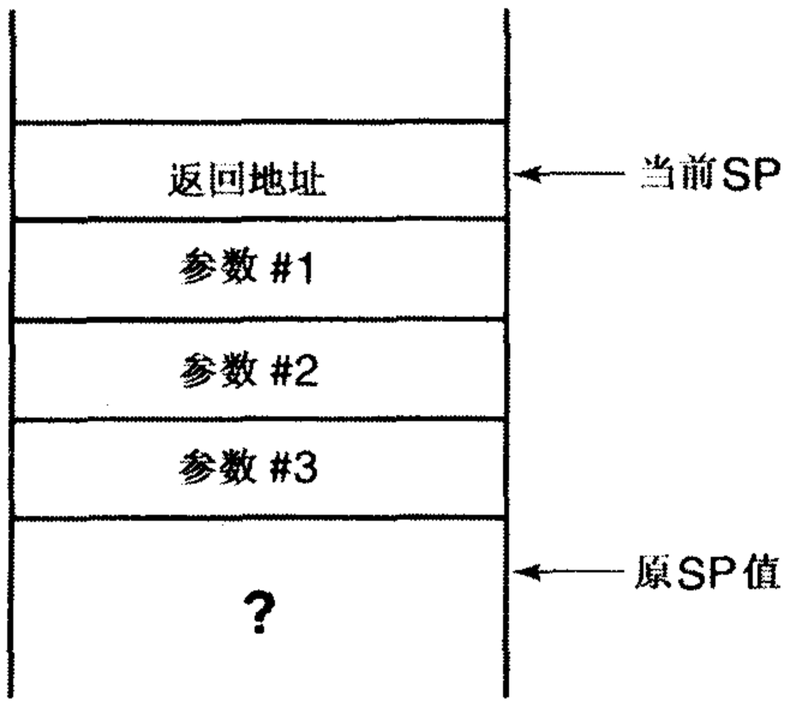


图 18.2　在跳转子程序指令之后的堆栈帧

## 二、函数序

接下来，执行流来到被调用函数的函数序：

```
int
func_ret_int( int a, int b, register int c )
{
        int     d;
```

```
        .globl  _func_ret_int
_func_ret_int:
        link    a6,#-8
        moveml  #0x80,sp@
        movl    a6@(16),d7
```

这个函数序类似于我们前面观察的那个。我们对指令必须进行更详细的研究以便完整地弄清整个堆栈帧的映像。link 指令分成几个步骤。首先，a6 的内容被压入到堆栈中。其次，堆栈指针的当前值被复制到 a6。图 18.3 显示了这个结果。

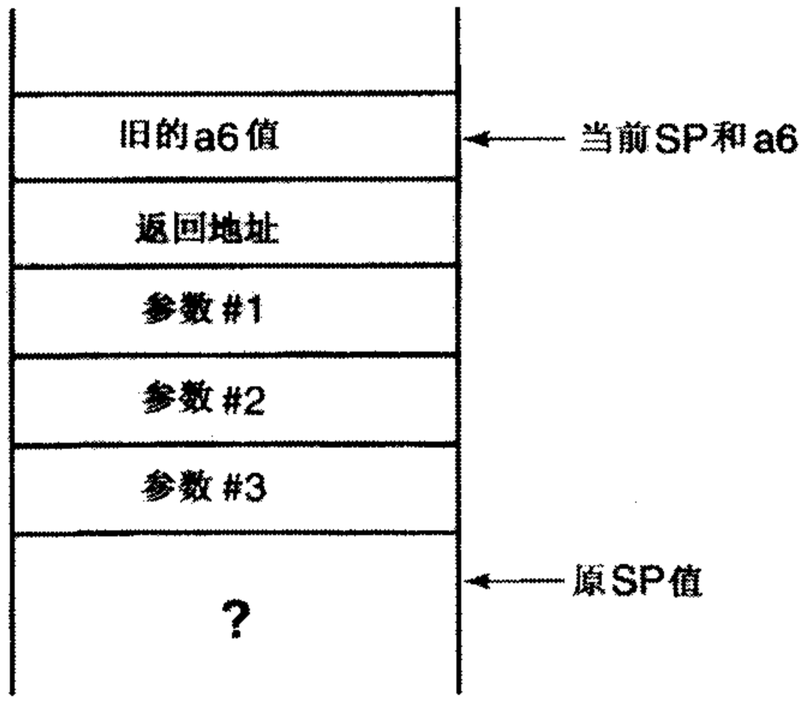


图 18.3　link 指令期间的堆栈帧

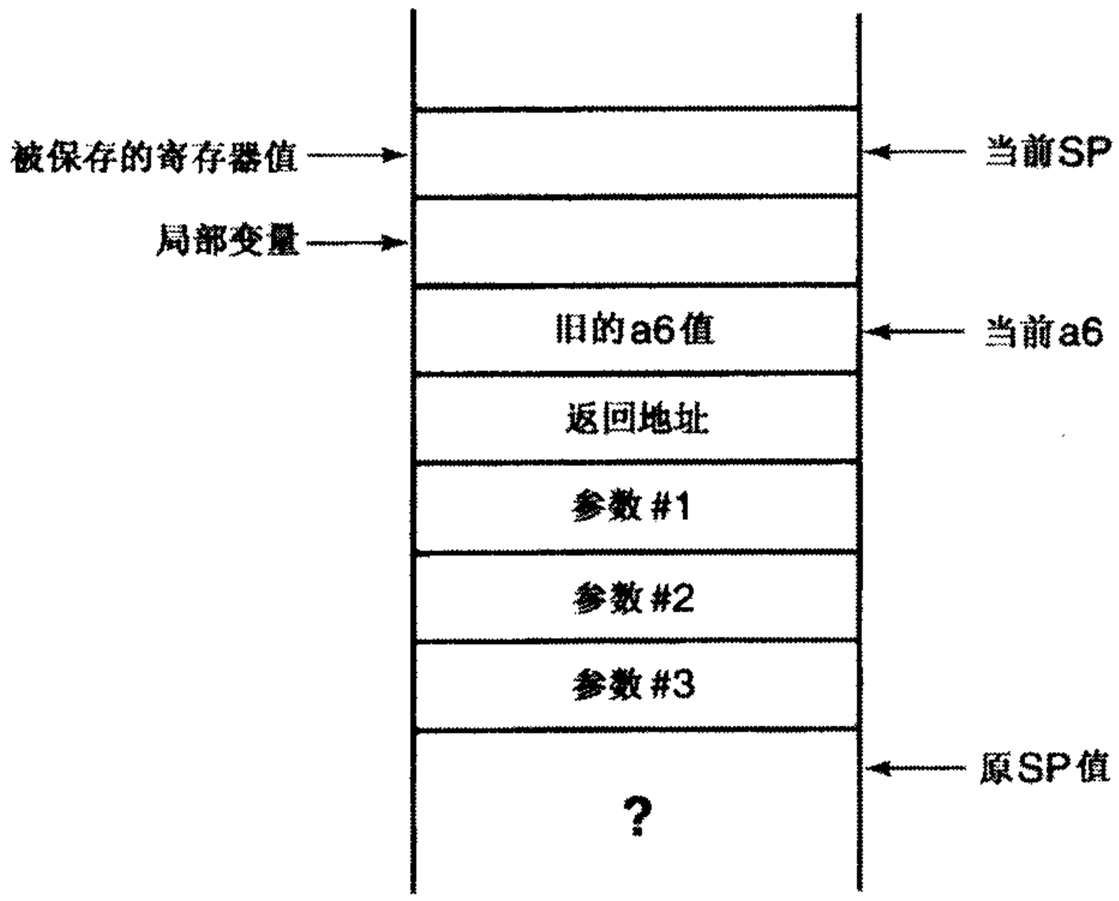


图 18.4　link 指令之后的堆栈帧

最后，link 指令从堆栈指针中减去 8。和以前一样，这将创建空间用于保存局部变量和被保存的寄存器值。下一条指令把一个单一的寄存器保存到堆栈帧。操作数 0x80 指定寄存器 d7。寄存器

存储在堆栈的顶部，它提示堆栈帧的顶部就是寄存器值保存的位置。堆栈帧剩余的部分必然是局部变量存储的地方。图 18.4 显示了到目前为止我们所知道的堆栈帧的情况。

函数序所执行的最后一个任务是从堆栈复制一个值到 d7。函数把第 3 个参数声明为寄存器变量，这第 3 个参数的位置是从帧指针往下 16 个字节。在这台机器上，寄存器变量在函数序中正常地通过堆栈传递并复制到一个寄存器。这条额外的指令带来了一些开销——如果函数中并没有很多指令使用这个参数，那么它在时间或空间上的节约将无法弥补把参数复制到寄存器而带来的开销。

### 三、堆栈中的参数次序

我们现在可以推断出为什么参数要按参数列表相反的顺序压入到堆栈中。被调用函数使用帧指针加一个偏移量来访问参数。当参数以反序压入到堆栈时，参数列表的**第 1 个**参数便位于堆栈中这堆参数的顶部，它距离帧指针的偏移量是一个常数。事实上，**任何**一个参数距离帧指针的偏移量都是一个常数，这和堆栈中压入多少个参数并无关系。

如果参数以相反的顺序压入到堆栈中又会怎样呢（也就是按照参数列表的顺序）？这样一来，第 1 个参数距离帧指针的偏移量就和压入到堆栈的参数数量有关。编译器可以计算出这个值，但还是存在一个问题——实际传递的参数数量和函数期望接受的参数数量可能并不相同。在这种情况下，这个偏移量就是不正确的，当函数试图访问一个参数时，它实际所访问的将不是它想要的那个。

那么在反序方案中，额外的参数是如何处理的呢？堆栈帧的图显示任何额外的参数都将位于前几个参数的下面，第 1 个参数距离帧指针的距离将保持不变。因此，函数可以正确地访问前三个参数，对于额外的参数可以简单地忽略。

**提示：**

如果函数知道存在额外的参数，在这台机器上，函数可以通过取最后一个参数的地址并增加堆栈指针的值来访问它们的值。但更好的方法是使用 stdarg.h 文件定义的宏，它们提供了一个可移植的接口来访问可变参数。

### 四、最终的堆栈帧布局

这个编译器所产生的堆栈帧的映像到此就完成了，它在图 18.5 中显示。

让我们继续观察这个函数：

```
        d = b - 6;
        return a + b + c;
}
```

---

```
movl    a6@(12), d0
subql   #6, d0
movl    d0, a6@(-4)
movl    a6@(8), d0
addl    a6@(12), d0
addl    d7, d0
moveml  a6@(-8), #0x80
unlk    a6
rts
```

通过堆栈帧映像，我们很容易判断第 1 条 movl 指令是把第 2 个参数复制到 d0。下一条指令将这个值减去 6，第 3 条指令把结果存储到局部变量 d。d0 的作用是计算过程中的“中间结果暂存器”

或临时位置。这也是它不能用于存放寄存器变量的原因之一。

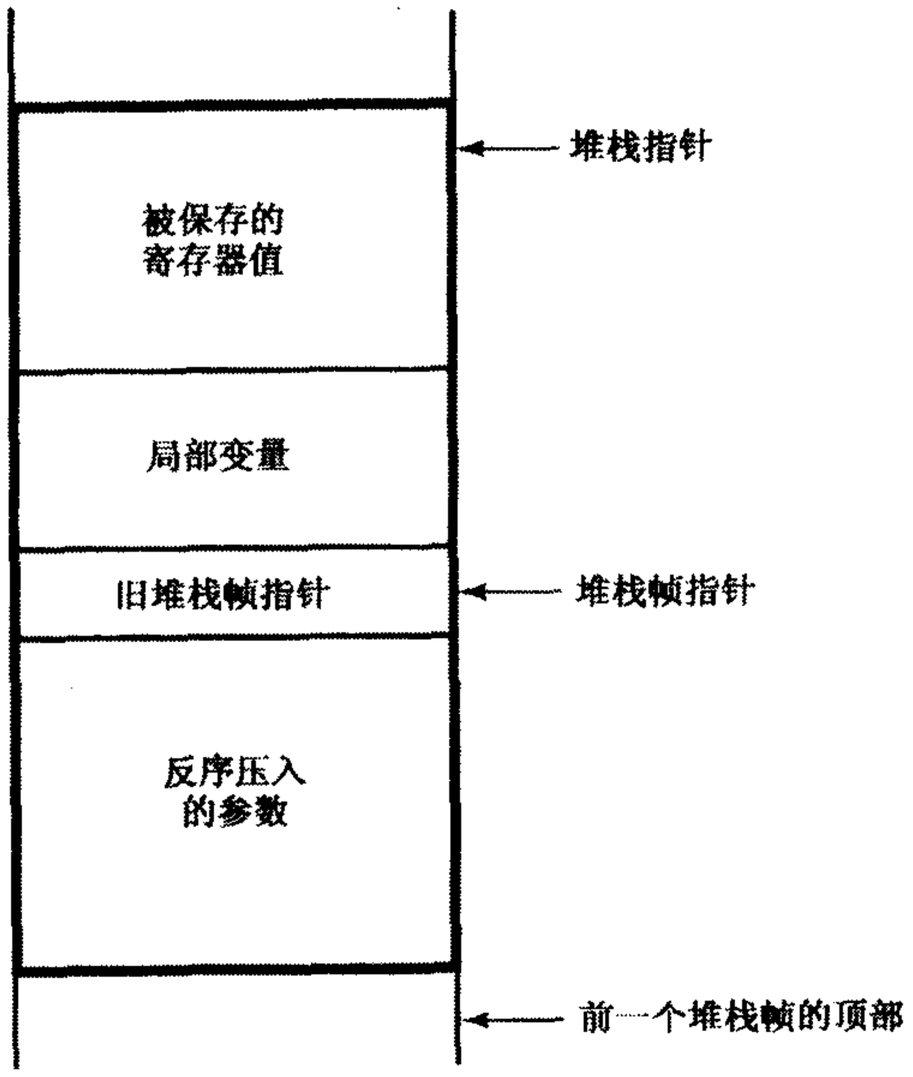


图 18.5　堆栈帧布局

接下来的三条指令对 return 语句进行求值。这个值就是我们希望返回给调用函数的值。但在这里，结果值存放在 d0 中。记住这个细节，以后会用到。

## 五、函数跋

这个函数的函数跋以一条 moveml 指令开始，它用于恢复以前被保存的寄存器值。然后 unkl(unlink)指令把 a6 的值复制给堆栈指针并把从堆栈中弹出的 a6 的旧值装入到 a6 中。这个动作的效果就是清除堆栈帧中返回地址以上的那部分内容。最后，rts 指令通过把返回地址从堆栈中弹出到程序计数器，从而从该函数返回。

现在，执行流从调用程序的地点继续。注意此时堆栈尚未被完全清理。

```
i2 = func_ret_int( 10, i1, i10 );
```

```
lea     sp@(12),sp
movl    d0,d6
```

当我们返回到调用程序之后执行的第 1 条指令就是把 12 加到堆栈指针。这个加法运算有效地把参数值从堆栈中弹出。现在，堆栈的状态就和调用函数前的状态完全一样了。

有趣的是，**被调用**函数并没有从堆栈中完全清除它的整个堆栈帧：参数还留在那里等待**调用**函数清除。同样，它的原因和可变参数列表有关。调用函数把参数压到堆栈上，所以只有它才知道堆栈中到底有多少个参数。因此，只有调用函数可以安全地清除它们。

## 六、返回值

函数跋并没有使用 d0，因此它依然保存着函数的返回值。第 2 条指令在从函数返回后执行，它

把 d0 的值复制到 d6，后者是变量 i2 的存放位置，也就是结果所在的位置。

在这个编译器中，函数返回一个值时把它存放在 d0，调用函数从被调用函数返回之后从 d0 获取这个值。这个协议是 d0 不能用于存放寄存器变量的另一个原因。

下一个被调用的函数返回一个 double 值。

```
dbl = func_ret_double();
c1 = func_ret_char_ptr( c1 );
```

```
jbsr    _func_ret_double
movl    d0,a6@(-48)
movl    d1,a6@(-44)

pea     a5@
jbsr    _func_ret_char_ptr
addqw   #4,sp
movl    d0,a5
```

这个函数并没有任何参数，所以没有什么东西压入到堆栈中。在这个函数返回之后，d0 和 d1 的值都被保存。在这台机器上，double 的长度是 8 个字节，无法放入一个寄存器中。因此，要返回这种类型的值，必须同时使用 d0 和 d1 寄存器。

最后那个函数调用说明了指针变量是如何从函数中返回的：它们也是通过 d0 进行传递的。不同的编译器可能通过 a0 或其他寄存器来传递它们。这个程序的剩余指令属于这个函数的函数序部分。

### 18.1.7 表达式的副作用

在第 4 章，我曾提到如果像下面这样的表达式

```
y + 3;
```

出现在程序中，它将会被求值但不会对程序产生影响，因为它的结果并未保存。接着我在一个脚注里说明它实际上**可以**以一种微妙的方式对程序的执行产生影响。

考虑程序 18.3，它被认为将返回 a+b 的值。这个函数计算一个结果但并不返回任何东西，因为这个表达式被错误地从 return 语句中省略。但使用这个编译器，这个函数实际上可以返回这个值！d0 被用于计算 x，并且由于这个表达式是最后进行求值的，所以当函数结束时 d0 仍然保存了这个结果值。所以这个函数很意外地向调用函数返回了正确的值。

```
/*
**  尽管存在一个巨大错误，但仍能在某些机器上正确运行的函数。
*/
int
erroneous( int a, int b )
{
        intx;

        /*
        ** 计算答案，并返回它
        */
        x = a + b;
        return;
}
```

**程序 18.3　一个意外地返回正确值的函数**　　no_ret.c

现在假定我们在 return 语句之前插入了这样一个表达式：

```
a + 3;
```

这个新表达式将修改 d0 的值。即使这个表达式的结果并未存储于任何变量中，但它还是影响了程序的执行，因为它修改了这个函数的返回值。

类似的问题也可以由于调试语句引起。如果你增加了一条语句

```
printf( "Function returns the value %d\n", x );
```

把它插入到 return 语句之前，函数也将不会返回正确的值。如果删除了这条语句，函数又能正确运行。当你发现一条调试语句也能改变程序的行为时，你心中的挫折感可想而知！

之所以可能出现这些效果，其罪魁祸首是原先存在的那个错误——return 语句省略了表达式。这种现象听上去好像不太可能，但令人吃惊的是，在一些老式的编译器里经常出现这种情况，这是因为当它们发现一个函数应该返回某个值但实际上并未返回任何值时并不会向程序员发出警告。

## 18.2　C 和汇编语言的接口

这个试验已经显示了编写能够调用 C 程序或者被 C 程序调用的汇编语言程序所需要的内容。与**这个**环境相关的结果总结如下——**你的环境肯定在某些方面与它不同！**

首先，汇编程序中的名字必须遵循外部标识符的规则。在这个系统中，它必须以一个下划线开始。

其次，汇编程序必须遵循正确的函数调用/返回协议。有两种情况：从一个汇编语言程序调用一个 C 程序和从一个 C 程序调用一个汇编程序。为了从汇编语言程序调用 C 程序：

1．如果寄存器 d0、d1、a0 或 a1 保存了重要的值，它们必须在调用 C 程序之前进行保存，因为 C 函数不会保存它们的值。

2．任何函数的参数必须以参数列表相反的顺序压入到堆栈中。

3．函数必须由一条“跳转子程序”类型的指令调用，它会把返回地址压入到堆栈中。

4．当 C 函数返回时，汇编程序必须清除堆栈中的任何参数。

5．如果汇编程序期望接受一个返回值，它将保存在 d0（如果返回值的类型为 double，它的另一半将位于 d1）。

6．任何在调用之前进行过保存的寄存器此时可以恢复。

为了编写一个由 C 程序调用的汇编程序：

1．保存任何你希望修改的寄存器（除 d0、d1、a0 和 a1 之外）。

2．参数值从堆栈中获得，因为调用它的 C 函数把参数压入在堆栈中。

3．如果函数应该返回一个值，它的值应保存在 d0 中（在这种情况下，d0 **不能**进行保存和恢复）。

4．在返回之前，函数必须清除任何它压入到堆栈中的内容。

在你的汇编程序中创建一个完全 C 风格的堆栈帧并无必要。你所要做的就是调用一个能够以正确的方式压入参数并当它返回时能够正确地执行清理任务的函数。在一个由 C 程序调用的汇编程序里，你必须访问 C 函数放置在那里的参数。

在你实际编写汇编函数之前，你需要知道你机器上的汇编语言。一些简陋的能够让我们明白一个现有的汇编程序是如何工作的知识对于编写新程序是远远不够的。

程序 18.4 和 18.5 是两个从 C 函数调用汇编函数以及从汇编函数调用 C 函数的例子。虽然它们

都是特定于这个环境的，但对于说明这方面的情况还是非常有用的。第 1 个例子是一个汇编语言程序，它返回 3 个整型参数的和。这个函数并没有费心完成堆栈帧，它只是计算参数的和并返回。我们将以下面的方式从一个 C 函数中调用这个函数：

```
sum = sum_three_values( 25, 14, -6 );
```

第 2 个例子显示了一段汇编语言程序，它需要打印 3 个值，它调用 printf 函数来完成这项工作。

```
|
| 对三个整数求和，并返回这个值。
|
        .text

        .globl    _sum_three_values
_sum_three_values:
        movl      sp@(4),d0       |Get 1st arg,
        addl      sp@(8),d0       |add 2nd arg,
        addl      sp@(12),d0      |add last arg.
        rts                       |Return.
```

**程序 18.4　对 3 个整数求和的汇编语言程序**　　sum.s

```
|
| 需要打印三个值，x,y 和 z。
|
        movl      z,sp@-                | Push args on the
        movl      y,sp@-                | stack in reverse
        movl      x,sp@-                | order: format, x,
        movl      #format,sp@-          | y, and z.
        jbsr      _printf               | Now call printf
        addl      #16,sp                | Clean up stack
        \&...
        .data
format:.ascii  "x = %d, y = %d,    and z = %d"
        .byte   012, 0                  | Newline and null
        .even
x:      .long   25
y:      .long   45
z:      .long   50
```

**程序 18.5　调用 printf 函数的汇编语言程序**　　printf.s

## 18.3　运行时效率

什么时候一个程序在老式的计算机上会“太大”呢？当程序增长后的容量超过了内存的数量时，它就无法运行，因此它就属于“太大”。即使在一些现代的机器上，一个必须存储于 ROM 的程序必须相当小才有可能装入到有限的内存空间中[1]。

但许多现代的计算机系统在这方面的限制大不如前，这是因为它们提供了**虚拟内存**(virtual memory)。虚拟内存是由操作系统实现的，它在需要时把程序的活动部分放入内存并把不活动的部分复制到磁盘中，这样就允许系统运行大型的程序。但程序越大，需要进行的复制就越多。所以大

[1] 只读内存(ROM, Read Only Memory)就是无法进行修改的内存。它通常用于存储那些在计算机上控制一些设备的程序。

型程序不是像以前那样根本无法运行，而是随着程序的增大，它的执行效率逐渐降低。所以，什么时候程序显得“太大”呢？就是当它运行得太慢的时候。

程序的执行速度显然与它的体积有关。程序执行的速度越慢，使用这个程序就会显得越不舒服。我们很难界定究竟在哪一点一个程序突然会被扣上一顶“太慢”的帽子。除非它必须对一些它自身无法控制的物理事件作出反应。例如，一个操作 CD 播放器的程序如果处理数据的速度无法赶上数据从 CD 传送过来的速度，它显然就太慢了。

## 提高效率

现代的经过优化的编译器在从一个 C 程序产生高效的目标代码方面做得非常好。因此，你把时间花在对代码进行一些小的修改以便使它效率更高常常并不是很合算。

**提示：**

如果一个程序太大或太慢，较之钻研每个变量，看看把它们声明为 register 能不能提高效率，选择一种效率更高的算法或数据结构往往效果要满意得多。然而，这并不是说你可以在代码中胡作非为，因为风格恶劣的代码总是会把事情弄得更糟。

如果一个程序太大，你很容易想到从哪里着手可以使程序变得更小：最大的函数和数据结构。但如果一个程序太慢，你该从何处着手提高它的速度呢？答案是对程序进行性能评测，简单地说就是测算程序的每个部分在执行时所花费的时间。花费时间最多的那部分程序显然是优化的目标。程序中使用最频繁的那部分代码运行速度如果能更快一些，将能够大大提高程序的整体运行速度。

绝大多数 UNIX 系统都具有性能评测工具，这些工具在许多其他操作系统中也有。图 18.6 是其中一个这类工具的输出的一部分。它显示了在某个特定程序的执行期间每个函数所耗费时间的名次

```
 Seconds    #Calls   Function Name
-------------------------------------
    4.94    293423   malloc
    3.21    272593   free
    2.85    658973   _nextch_from_chrlst
    2.82    272593   _insert
    2.69    791309   _check_traverse
    2.57      9664   _lookup_macro
    1.35    372915   _append_to_chrlst
    1.23    254501   _interpolate
    1.10    302714   _next_input_char
    1.09    285031   _input_filter
    0.91    197235   demote
    0.90    272419   putfreehdr
    0.82    285031   _nextchar
    0.79      7620   _lookup_number_register
    0.77     63946   _new_character
    0.65    292822   allocate
    0.57    272594   _getfreehdr
    0.51     34374   _next_text_char
    0.46    151006   _duplicate_char
    0.41      6473   _expression
    0.37      8843   _sub_expression
    0.35     23774   _skip_white_space
    0.34    203535   _copy_interpolate
    0.32     10984   _copy_function
    0.31    133032   _duplicate_ascii_char
    0.31       604   _process_filled_text
    0.31     52627   _next_ascii_char
```

图 18.6　性能评测样例信息

以及它所耗费的时间（以秒为单位）。这个程序的总共执行时间是 32.95 秒。我们可以从这个列表中发现三个有趣的地方。

1．在耗费时间最多的函数中，有些是库函数。在这个例子里，malloc 和 free 占据了前两位。你无法修改它们的实现方式，但在重新设计程序时，如果能够不用或少用动态内存分配，程序的执行速度在最多情况下可以提高 25%。

2．有些函数之所以耗费了大量的时间是因为它们被调用的次数非常多。即使每次单独调用时它的速度很快，由于调用次数多，所以总的时间不少。_nextch_from_chrlst 就是其中一例。这个函数每次调用所耗费的时间只有 4.3 微秒。由于它是如此之短，所以你通过对函数进行改进大幅度提高它的执行速度的可能性非常之小。但是，就是因为它的调用次数非常多，所以它还是值得加以关注。加上几个明智的 register 声明稍微提高函数的效率，对程序的总体性能可能还是会有较大的改善。

3．有些函数调用的次数并不多，但每次调用所花费的时间却很长。例如，_loopup_macro 平均每次调用要花费 265 微秒的时间。为这个函数寻找一种更快的算法最多可以使程序的速度提高 7.75%。[1]

**作为最后一招**，你可以对单个函数用汇编语言重新编码，函数越小，重新编码就越容易。这种方法的效果可能很好，因为在小型函数中，C 的函数序和函数跋所耗费的固定开销在执行时间中所占的比例不小。对较大的函数进行重新编码要困难得多，因此把你的时间花在这个地方效率不是很高。

性能评测常常并不能告诉你原先不知道的东西，但有时候它的结果可能相当出人意料。性能评测的优点在于你可能弄清你正在花时间研究的那部分程序可能会带来最大程度的性能提高。

## 18.4 总结

我们在这台机器上研究的有些任务在许多其他环境中也是以这些方式实现的。例如，绝大多数环境都创建某种类型的堆栈帧，函数用它来保存它们的数据。堆栈帧的细节可能各不相同，但它们的基本思路是相当一致的。

其他一些任务在不同的环境中可能差异较大。有些计算机具有特殊的硬件用于保存函数的参数，所以它们的处理方式和我们所看到的可能大不一样。其他机器在传递函数值时也可能采用不同的方式。

**警告：**

事实上，不同的编译器可能在相同的机器上产生不同的代码。另一种在我们的测试机器上使用的编译器能够使用 9 至 14 个寄存器变量（具体数目取决于一些其他情况）。不同的编译器可能具有不同的堆栈帧约定或者在函数的调用和返回上使用不兼容的协议。因此，在通常情况下，你不能使用不同的编译器编译同一个程序的不同片段。

提高程序效率的最好方法是为它选择一种更好的算法。接下来的一种提高程序执行速度的最佳手段是对程序进行性能评测，看看程序的哪个地方花费的时间最多。你把优化措施集中在程序的这部分将产生最好的结果。

---

[1] 事实上我们还需要注意第 4 点。malloc 的调用次数比 free 多了 20 833 次，所以有些内存被泄漏了。

**提示：**

学习机器的运行时环境既有益处又存在危险——说它有用是因为你获得的知识允许你做一些其他方法无法完成的事情，说它危险是因为程序中如果存在任何依赖于这方面知识的东西，可能会损害程序的可移植性。现在这个时代，计算机发展的速度很快，许多机器还没有摆到货架上就已经过时。因此，程序从一台机器转换到另一台机器的可能性是非常现实的，所以我们非常希望代码具有良好的可移植性。

## 18.5　警告的总结

1. 是链接器而不是编译器决定外部标识符的最大长度。
2. 你无法链接由不同编译器产生的程序。

## 18.6　编程提示的总结

1. 使用 stdarg 实现可变参数列表。
2. 改进算法比优化代码更有效率。
3. 使用某种环境特有的技巧会导致程序不可移植。

## 18.7　问题

1. 在你的环境中，堆栈帧的样子是什么样的？
2. 在你的系统中，有意义的外部标识符最长可以有多少个字符？
3. 在你的环境中，寄存器可以存储多少个变量？对于指针和非指针值，它是不是进行了任何区分？
4. 在你的环境中，参数是如何传递给函数的？值是如何从函数返回的？



5. 在本章我们所使用的这台机器上，如果一个函数把它的一个或多个参数声明为寄存器变量，那么这个函数的参数在函数序中和平常一样被压入到堆栈中，然后再复制到正确的寄存器中。如果这些参数能够直接保存到寄存器，函数的效率会更高一些。这种参数传递技巧能够实现吗？如果能，怎么实现呢？



6. 在我们所讨论的环境中，调用函数负责清除它压入到堆栈中的参数。那么，能不能由被调用函数来完成这项任务呢？如果不能，那么在满足什么条件下它才可能呢？
7. 如果说汇编语言程序比 C 程序效率更高，那么为什么不用汇编语言来编写所有程序呢？

## 18.8　编程练习

★ 1. 为你的系统编写一个汇编语言函数，它接受 3 个整型参数并返回它们的和。

★ 2. 编写一个汇编语言程序，创建 3 个整型值并调用 printf 函数把它们打印出来。



★★ 3. 假定 stdarg.h 文件被意外地从你的系统中删除。请编写一组第 7 章所描述的 stdarg 宏。