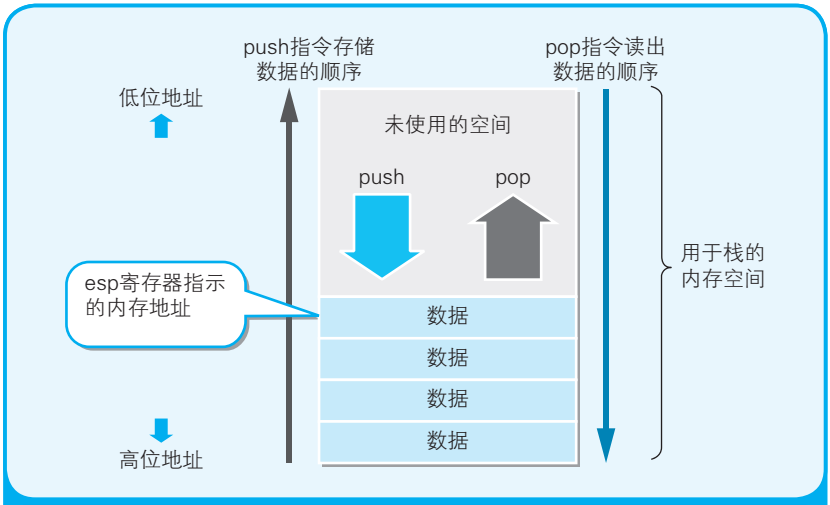

图 10-3　栈的模型

栈是存储临时数据的区域，它的特点是通过 push 指令和 pop 指令进行数据的存储和读出。往栈中存储数据称为“入栈”，从栈中读出数据称为“出栈”。32 位 x86 系列的 CPU 中，进行 1 次 push 或 pop，即可处理 32 位（4 字节）的数据。

push 指令和 pop 指令中只有一个操作数。该操作数表示的是“push 的是什么及 pop 的是什么”，而不需要指定“对哪一个地址编号的内存进行 push 或 pop”。这是因为，对栈进行读写的内存地址是由 esp 寄存器（栈指针）进行管理的。push 指令和 pop 指令运行后，esp 寄存器的值会自动进行更新（push 指令是 -4，pop 命令是 +4），因而程序员就没有必要指定内存地址了。

代码清单 10-2 中多次用到了 push 指令和 pop 指令。push 指令运行后，操作数中指定的值就会被自动 push 入栈，pop 指令运行后，最后存储在栈中的值就会被 pop 到指定的操作数中出栈。就如第 4 章中所

介绍的那样，这种数据的存储顺序称为 LIFO（Last In First Out）方式。

10.7 函数调用机制

前面说了这么多，至此我们终于把阅读汇编语言源代码的准备工作完成了。让我们再来回顾一下代码清单 10-2 的内容。首先，让我们从 MyFunc 函数调用 AddNum 函数的汇编语言部分开始，来对函数的调用机制进行说明。函数调用是栈发挥大作用的场合。把代码清单 10-2 中的 C 语言源代码部分去除，然后再在各行追加注释，这时汇编语言的源代码就如代码清单 10-4 所示。这也就是 MyFunc 函数的处理内容。

代码清单 10-4　函数调用的汇编语言代码 ①

```
_MyFunc    proc    near
   push     ebp      ; 将 ebp 寄存器的值存入栈中 ——————— (1)
   mov      ebp,esp  ; 将 ebp 寄存器的值存入 ebp 寄存器 ——— (2)
   push     456      ; 456 入栈 ——————————————— (3)
   push     123      ; 123 入栈 ——————————————— (4)
   call     _AddNum  ; 调用 AddNum 函数 ——————— (5)
    add     ebp,8    ; esp 寄存器的值加 8 ————— (6)
    pop     ebp      ; 读出栈中的数值存入 esp 寄存器 ——— (7)
    ret              ; 结束 MyFunc 函数，返回到调用源 ——— (8)
_MyFunc    endp
```

（1）、（2）、（7）、（8）的处理适用于 C 语言中所有的函数，我们会在后面展示 AddNum 函数处理内容时进行说明。这里希望大家先关注一下（3）～（6）部分，这对了解函数调用的机制至关重要。

（3）和（4）表示的是将传递给 AddNum 函数的参数通过 push 入栈。在 C 语言的源代码中，虽然记述为函数 AddNum（123，456），但

① 在函数的入口处把寄存器 ebp 的值入栈保存（代码清单 10-4（1）），在函数的出口处出栈（代码清单 10-4（7）），这是 C 语言编译器的规定。这样做是为了确保函数调用前后 ebp 寄存器的值不发生变化。

入栈时则会按照 456、123 这样的顺序，也就是位于后面的数值先入栈。这是 C 语言的规定。(5) 的 call 指令，把程序流程跳转到了操作数中指定的 AddNum 函数所在的内存地址处。在汇编语言中，函数名表示的是函数所在的内存地址。AddNum 函数处理完毕后，程序流程必须要返回到编号 (6) 这一行。call 指令运行后，call 指令的下一行 ((6) 这一行) 的内存地址 (调用函数完毕后要返回的内存地址) 会自动地 push 入栈。该值会在 AddNum 函数处理的最后通过 ret 指令 pop 出栈，然后程序流程就会返回到 (6) 这一行。

(6) 部分会把栈中存储的两个参数 (456 和 123) 进行销毁处理，也就是在第 5 章提到的栈清理处理。虽然通过使用两次 pop 指令也可以实现，不过采用 esp 寄存器加 8 的方式会更有效率 (处理 1 次即可)。对栈进行数值的输入输出时，数值的单位是 4 字节。因此，通过在负责栈地址管理的 esp 寄存器中加上 4 的 2 倍 8，就可以达到和运行两次 pop 命令同样的效果。虽然内存中的数据实际上还残留着，但只要把 esp 寄存器的值更新为数据存储地址前面的数据位置，该数据也就相当于被销毁了。

前面已经提到，push 指令和 pop 指令必须以 4 字节为单位对数据进行入栈和出栈处理。因此，AddNum 函数调用前和调用后栈的状态变化就如图 10-4 所示。长度小与 4 字节的 123 和 456 这些值在存储时，也占用了 4 字节的栈区域。

代码清单 10-1 中列出的 C 语言源代码中，有一个处理是在变量 *c* 中存储 AddNum 函数的返回值，不过在汇编语言的源代码中，并没有与此对应的处理。这是因为编译器有最优化功能。**最优化功能**是编译器在本地代码上费尽功夫实现的，其目的是让编译后的程序运行速度更快、文件更小。在代码清单 10-1 中，由于存储着 AddNum 函数返回值的变量 *c* 在后面没有被用到，因此编译器就会认为“该处理没有意

义”，进而也就没有生成与之对应的汇编语言代码。在编译代码清单 10-1 的代码时，应该会出现“警告 W8004 Sample4.c 11: 'c' 的赋值未被使用（函数 MyFunc）”这样的警告消息。

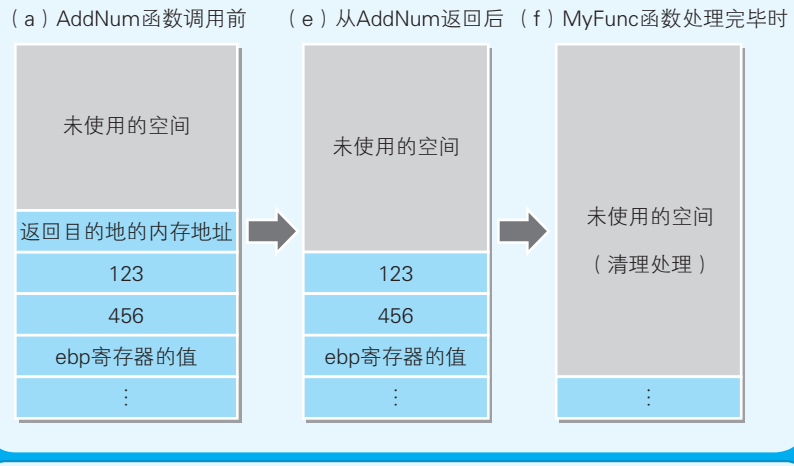

图 10-4　AddNum 函数调用前后栈的状态变化

10.8　函数内部的处理

接下来，让我们透过执行 AddNum 函数的源代码部分，来看一下参数的接收、返回值的返回等机制（代码清单 10-5）。

代码清单 10-5　函数内部的处理

```
_AddNum    proc    near
    push   ebp                          (1)
    mov    ebp,esp                      (2)
    mov    eax,dword ptr [ebp+8]        (3)
    add    eax,dword ptr [ebp+12]       (4)
    pop    ebp                          (5)
    ret                                 (6)
_AddNum    endp
```

ebp 寄存器的值在（1）中入栈，在（5）中出栈。这主要是为了把函数中用到的 ebp 寄存器的内容，恢复到函数调用前的状态。在进入函数处理之前，无法确定 ebp 寄存器用到了什么地方，但由于函数内部也会用到 ebp 寄存器，所以就暂时将该值保存了起来。CPU 拥有的寄存器是有数量限制的。在函数调用前，调用源有可能已经在使用 ebp 寄存器了。因而，在函数内部利用的寄存器，要尽量返回到函数调用前的状态。为此，我们就需要将其暂时保存在栈中，然后再在函数处理完毕之前出栈，使其返回到原来的状态。

（2）中把负责管理栈地址的 esp 寄存器的值赋值到了 ebp 寄存器中。这是因为，在 mov 指令中方括号内的参数，是不允许指定 esp 寄存器的。因此，这里就采用了不直接通过 esp，而是用 ebp 寄存器来读写栈内容的方法。

（3）是用 [ebp+8] 指定栈中存储的第 1 个参数 123，并将其读出到 eax 寄存器中。像这样，不使用 pop 指令，也可以参照栈的内容。而之所以从多个寄存器中选择了 eax 寄存器，是因为 eax 寄存器是负责运算的累加寄存器。

通过（4）的 add 指令，把当前 eax 寄存器的值同第 2 个参数相加后的结果存储在 eax 寄存器中。[ebp+12] 是用来指定第 2 个参数 456 的。在 C 语言中，函数的返回值必须通过 eax 寄存器返回，这也是规定。不过，和 ebp 寄存器不同的是，eax 寄存器的值不用还原到原始状态。至此，我们进行了很多细节的说明，其实就是希望大家了解“**函数的参数是通过栈来传递，返回值是通过寄存器来返回的**”这一点。

（6）中 ret 指令运行后，函数返回目的地的内存地址会自动出栈，据此，程序流程就会跳转返回到代码清单 10-4 的（6）（Call _AddNum 的下一行）。这时，AddNum 函数入口和出口处栈的状态变化，就如图

10-5 所示。将图 10-4 和图 10-5 按照（a）（b）（c）（d）（e）（f）的顺序来看的话，函数调用处理时栈的状态变化就会很清楚了。由于（a）状态时处理跳转到 AddNum 函数，因此（a）和（b）是同样的。同理，在（d）状态时，处理跳转到了调用源，因此（d）和（e）是同样的。在（f）状态时则进行了清理处理。栈的最高位的数据地址，是一直存储在 esp 寄存器中的。

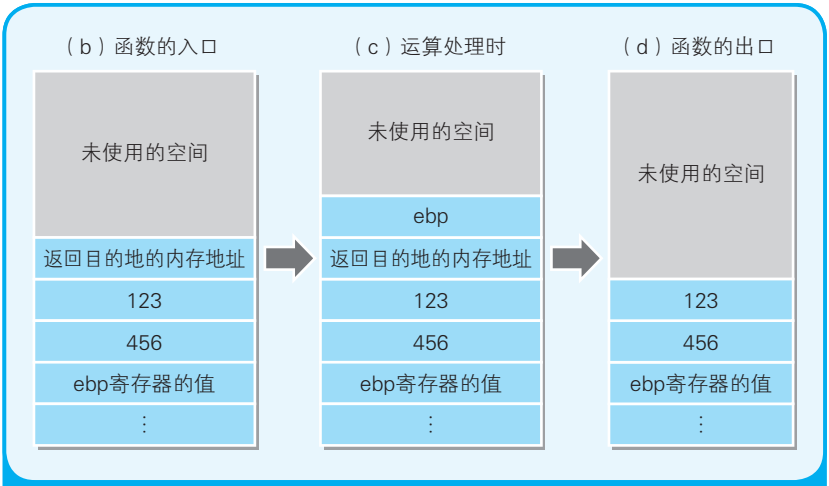

图 10-5　AddNum 函数内部的栈状态变化

10.9　始终确保全局变量用的内存空间

熟悉了汇编语言后，接下来将进入到本章的后半部分。C 语言中，在函数外部定义的变量称为**全局变量**，在函数内部定义的变量称为**局部变量**。全局变量可以参阅源代码的任意部分，而局部变量只能在定义该变量的函数内进行参阅。例如，在 MyFuncA 函数内部定义的 i 这个局部变量就无法通过 MyFuncB 函数进行参阅。与此相反，如果是在函数外部定义的全局变量，MyFuncA 函数和 MyFuncB 函数都可以参

阅。下面，就让我们通过汇编语言的源代码，来看一下全局变量和局部变量的不同。

代码清单 10-6 的 C 语言源代码中定义了初始化（设定了初始值）的 *a*1～*a*5 这 5 个全局变量，以及没有初始化（没有设定初始值）的 *b*1～*b*5 这 5 个全局变量，此外还定义了 *c*1～*c*10 这 10 个局部变量，且分别给各变量赋了值。程序的内容没有什么特别的意思，这里主要是为了向大家演示。

代码清单 10-6　使用全局变量和局部变量的 C 语言源代码

```
// 定义被初始化的全局变量
int a1 = 1;
int a2 = 2;
int a3 = 3;
int a4 = 4;
int a5 = 5;
// 定义没有初始化的全局变量
int b1, b2, b3, b4, b5;

// 定义函数
void MyFunc()
{
      // 定义局部变量
      int c1, c2, c3, c4, c5, c6, c7, c8, c9, c10;

      // 给局部变量赋值
      c1 = 1;
      c2 = 2;
      c3 = 3;
      c4 = 4;
      c5 = 5;
      c6 = 6;
      c7 = 7;
      c8 = 8;
      c9 = 9;
      c10 = 10;

      // 把局部变量的值赋给全局变量
      a1 = c1;
      a2 = c2;
      a3 = c3;
```

```
    a4 = c4;
    a5 = c5;
    b1 = c6;
    b2 = c7;
    b3 = c8;
    b4 = c9;
    b5 = c10;
}
```

将代码清单 10-6 变换成汇编语言的源代码后，结果就如代码清单 10-7 所示。这里为了方便说明，我们省略了一部分汇编语言源代码，并改变了一下段定义的配置顺序，删除了注释。关于代码清单 10-7 中出现的汇编语言的指令，请参考表 10-3。

代码清单 10-7　代码清单 10-6 转换成汇编语言后的结果

```
_DATA   segment dword public use32 'DATA'
_a1     label   dword                          (4)
        dd      1                              (5)
_a2     label   dword
        dd      2
_a3     label   dword                                  (1)
        dd      3
_a4     label   dword
        dd      4
_a5     label   dword
        dd      5
_DATA   ends

_BSS    segment dword public use32 'BSS'
_b1     label   dword
        db      4 dup(?)                       (6)
_b2     label   dword
        db      4 dup(?)
_b3     label   dword                                  (2)
        db      4 dup(?)
_b4     label   dword
        db      4 dup(?)
_b5     label   dword
        db      4 dup(?)
_BSS    ends
```

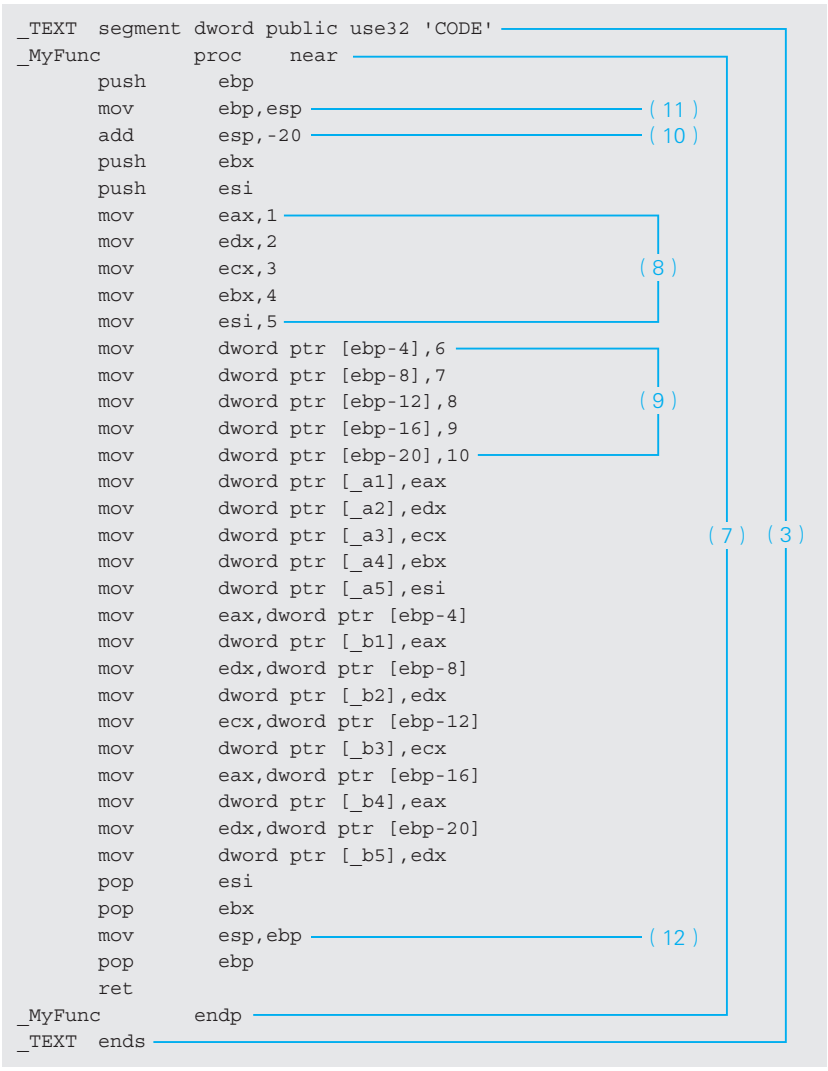

表 10-3　代码清单 10-7、10-9、10-12、10-14 中用到的汇编语言指令的功能

操作码	操作数	功　能
add	A,B	把 A 的值和 B 的值相加，并把结果存入 A
call	A	调用函数 A

（续）

操作码	操作数	功　能
cmp	A,B	对 A 和 B 的值进行比较，比较结果会自动存入标志寄存器中
inc	A	A 的值加 1
jge	标签名	和 cmp 命令组合使用。跳转到标签行
jl	标签名	和 cmp 命令组合使用。跳转到标签行
jle	标签名	和 cmp 命令组合使用。跳转到标签行
jmp	标签名	将控制无条件跳转到指定标签行
mov	A,B	把 B 的值赋值给 A
pop	A	从栈中读取出数值并存入 A 中
push	A	把 A 的值存入栈中
ret	无	将处理返回到调用源
xor	A,B	A 和 B 的位进行异或比较，并将结果存入 A 中

正如本章前半部分所讲的那样，编译后的程序，会被归类到名为段定义的组。初始化的全局变量，会像代码清单 10-7 的（1）那样被汇总到名为 _DATA 的段定义中，没有初始化的全局变量，会像（2）那样被汇总到名为 _BSS 的段定义中。指令则会像（3）那样被汇总到名为 _TEXT 的段定义中。这些段定义的名称是由 Borland C++ 的使用规范来决定的。_DATA segment 和 _DATA ends、_BSS segment 和 _BSS ends、_TEXT segment 和 _TEXT ends，这些都是表示各段定义范围的伪指令。

首先让我们来看一下 _DATA 段定义的内容。（4）中的 _a1 label dword 定义了 _a1 这个标签。**标签**表示的是相对于段定义起始位置的位置。由于 _a1 在 _DATA 段定义的开头位置，所以相对位置是 0。_a1 就相当于全局变量 *a*1。编译后的函数名和变量名前会附加一个下划线（_），这也是 Borland C++ 的规定。（5）中的 dd 1 指的是，申请分配了 4 字节的内存空间，存储着 1 这个初始值。dd（define double word）表示的是有两个长度为 2 的字节领域（word），也就是 4 字节的意思。