的英语单词缩写。例如，在加法运算的本地代码中加上 add（addition 的缩写）、在比较运算的本地代码中加上 cmp（compare 的缩写）等。这些缩写称为**助记符**，使用助记符的编程语言称为**汇编语言**。这样，通过查看汇编语言编写的源代码，就可以了解程序的本质了。因为这和查看本地代码的源代码，是同一级别的。

不过，即使是用汇编语言编写的源代码，最终也必须要转换成本地代码才能运行。负责转换工作的程序称为**汇编器**，转换这一处理本身称为**汇编**。在将源代码转换成本地代码这个功能方面，汇编器和编译器是同样的。

用汇编语言编写的源代码，和本地代码是一一对应的。因而，本地代码也可以反过来转换成汇编语言的源代码。持有该功能的逆变换程序称为**反汇编程序**，逆变换这一处理本身称为**反汇编**（图 10-1）。

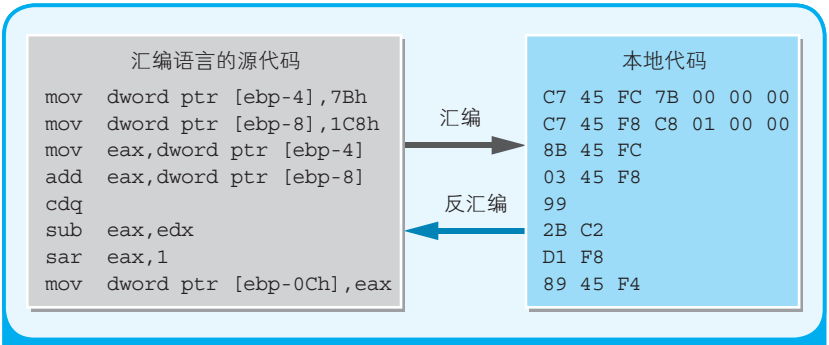

图 10-1　汇编语言的源代码和本地代码是一一对应的

哪怕是用 C 语言编写的源代码，编译后也会转换成特定 CPU 用的本地代码。而将其反汇编的话，就可以得到汇编语言的源代码，并对其内容进行调查。不过，本地代码变换成 C 语言源代码的反编译，则要比反汇编困难。这是因为，C 语言的源代码同本地代码不是一一对应

的，因此完全还原到原始的源代码是不太可能的[①]。

10.2 通过编译器输出汇编语言的源代码

除了将本地代码进行反汇编这一方法外，通过其他方式也可以获取汇编语言的源代码。大部分 C 语言编译器，都可以把利用 C 语言编写的源代码转换成汇编语言的源代码，而不是本地代码。利用该功能，就可以对 C 语言的源代码和汇编语言的源代码进行比较研究。笔者在学生时代的报告中，使用的便是该功能。Borland C++ 中，通过在编译器的选项中指定“-S”，就可以生成汇编语言的源代码了。大家也可以实际尝试一下。

用 Windows 的记事本等文本编辑器编写如代码清单 10-1 所示的 C 语言源代码，并将其命名为 Sample4.c 进行保存。C 语言源文件的扩展名，通常用“.c”来表示。该程序是由返回参数的两个整数值之和的 AddNum 函数[②]和调用 AddNum 函数的 MyFunc 函数构成的。因为没有包含程序运行起始位置[③]的 main 函数部分，这种情况下直接编译是无法运行的。大家只需把它看成是学习汇编语言的一个示例即可。

代码清单 10-1　由两个函数构成的 C 语言的源代码

```
// 返回两个参数值之和的函数
int AddNum(int a, int b)
```

① 通过解析可执行文件得到源代码的方式称为“反汇编”或“反编译”，也称为“反向工程”。市场上销售的软件程序等，有时会在其使用说明书中明确表明禁止反汇编及反编译。

② AddNum 函数仅仅返回两个参数值的相加结果。在实际的编程中，这种函数是不需要的。为了说明函数调用的机制，这里特意使用了这种简单的函数。

③ 在命令提示符上运行的程序中，main 函数位于程序运行起始位置。而在 Windows 上运行的程序中，WinMain 函数位于程序运行起始位置。程序运行起始位置也称为“入口点”。

```
{
    return a + b;
}

// 调用 AddNum 函数的函数
void MyFunc()
{
    int c;
    c = AddNum(123, 456);
}
```

由 Windows 开始菜单启动命令提示符，把当前目录[①]变更到 Sample4.c 保存的文件夹后，输入下面的命令并按下 Enter 键。bcc32 是启动 Borland C++ 编译器的命令。“-c”选项指的是，仅进行编译而不进行链接[②]。“-S”选项被用来指定生成汇编语言的源代码。

```
bcc32 -c -S Sample4.c
```

作为编译的结果，当前目录下会生成一个名为 Sample4.asm 的汇编语言源代码。汇编语言源文件的扩展名，通常用“.asm”来表示。下面就让我们使用记事本来看一下 Sample4.asm 的内容。可以发现，C 语言的源代码和转换成汇编语言的源代码是交叉显示的。而这也为我们对两者进行比较学习提供了绝好的教材。在该汇编语言代码中，分号（;）以后是注释。由于 C 语言的源代码变成了注释，因此就可以直接对 Sample4.asm 进行汇编并将其转换成本地代码了（代码清单 10-2）。

① 当前目录指的是当前正在打开的目录（文件夹）。在命令提示符下对 C 语言的源文件进行编译时，该文件所在的目录必须是当前目录，所以有时候就需要变换当前目录。变换当前目录时，只需在命令提示符中的“CD”后面空上一个半角空格，然后加上需要跳转的目录，再按下回车即可。例如，如果要将 \Test 指定为当前目录的话，只需输入 CD\Test 然后按下回车键即可。CD 是 Change Dirctory 的略称。

② 链接是指把多个目标文件结合成 1 个可执行文件。详情请参考第 8 章。

代码清单 10-2 编译器生成的汇编语言源代码（一部分做了省略，彩色部分是转换成注释的 C 语言源代码）

```
_TEXT  segment dword public use32 'CODE'
_TEXT  ends
_DATA  segment dword public use32 'DATA'
_DATA  ends
_BSS   segment dword public use32 'BSS'
_BSS   ends
DGROUP group   _BSS,_DATA

_TEXT  segment dword public use32 'CODE'

_AddNum        proc    near
   ;
   ;   int AddNum(int a, int b)
   ;
       push      ebp
       mov       ebp,esp
   ;
   ;   {
   ;       return a + b;
   ;
       mov       eax,dword ptr [ebp+8]
       add       eax,dword ptr [ebp+12]
   ;
   ;   }
   ;
       pop       ebp
       ret
_AddNum        endp

_MyFunc        proc    near
   ;
   ;   void MyFunc()
   ;
       push      ebp
       mov       ebp,esp
   ;
   ;   {
   ;       int c;
   ;       c = AddNum(123, 456);
   ;
       push      456
       push      123
       call      _AddNum
       add       esp,8
```

```
    ;
    ;   }
    ;
        pop         ebp
        ret
_MyFunc         endp

_TEXT   ends
        end
```

10.3　不会转换成本地代码的伪指令

第一次看到汇编语言源代码的读者可能会感到有些难，不过实际上很简单。而且毫不夸张地说它比 C 语言还要简单。为了便于阅读汇编语言编写的源代码，在开始源代码内容的讲解前，让我们先来看一下下面几个要点。

汇编语言的源代码，是由转换成本地代码的指令（后面讲述的操作码）和针对汇编器的**伪指令**构成的。伪指令负责把程序的构造及汇编的方法指示给汇编器（转换程序）。不过伪指令本身是无法汇编转换成本地代码的。这里我们把代码清单 10-2 中用到的伪指令部分摘出，如代码清单 10-3 所示。

代码清单 10-3　从代码清单 10-2 中摘出的伪指令部分（彩色部分是伪指令）

```
_TEXT   segment dword public use32 'CODE'
_TEXT   ends
_DATA   segment dword public use32 'DATA'
_DATA   ends
_BSS    segment dword public use32 'BSS'
_BSS    ends
DGROUP group    _BSS,_DATA

_TEXT   segment dword public use32 'CODE'

_AddNum         proc    near
_AddNum         endp
```

```
_MyFunc         proc    near
_MyFunc         endp

_TEXT   ends
        end
```

由伪指令 segment 和 ends 围起来的部分，是给构成程序的命令和数据的集合体加上一个名字而得到的，称为**段定义**[①]。段定义的英文表达 segment 具有“区域”的意思。在程序中，段定义指的是命令和数据等程序的集合体的意思。一个程序由多个段定义构成。

源代码的开始位置，定义了 3 个名称分别为 _TEXT、_DATA、_BSS 的段定义。_TEXT 是指令的段定义，_DATA 是被初始化（有初始值）的数据的段定义，_BSS 是尚未初始化的数据的段定义。类似于这种段定义的名称及划分方法是 Borland C++ 的规定，是由 Borland C++ 的编译器自动分配的。因而程序段定义的配置顺序就成了 _TEXT、_DATA、_BSS，这样也确保了内存的连续性。group[②] 这一伪指令，表示的是把 _BSS 和 _DATA 这两个段定义汇总为名为 DGROUP 的组。此外，栈和堆的内存空间会在程序运行时生成，这一点已经在第 8 章中做过介绍。

围起 _AddNum 和 _MyFun 的 _TEXT segment 和 _TEXT ends，表示 _AddNum 和 _MyFunc 是属于 _TEXT 这一段定义的。因此，即使在源代码中指令和数据是混杂编写的，经过编译或者汇编后，也会转换成段定义划分整齐的本地代码。

_AddNum proc 和 _AddNum endp 围起来的部分，以及 _MyFunc

① 段定义（segment）是用来区分或者划定范围区域的意思。汇编语言的 segment 伪指令表示段定义的起始，ends 伪指令表示段定义的结束。段定义是一个连续的内存空间。

② group 指的是将源代码中不同的段定义在本地代码程序中整合为一个。

proc 和 MyFunc endp 围起来的部分，分别表示 AddNum 函数和 MyFunc 函数的范围。编译后在函数名前附带上下划线（_），是 Borland C++ 的规定。在 C 语言中编写的 AddNum 函数，在内部是以 _AddNum 这个名称被处理的。伪指令 proc 和 endp 围起来的部分，表示的是**过程**（procedure）的范围。在汇编语言中，这种相当于 C 语言的函数的形式称为过程。

末尾的 end 伪指令，表示的是源代码的结束。而至于其他伪指令的具体意思，大家不了解也没有关系。因为该章的主要目的并不是用汇编语言来编写程序。大家只需要能读懂汇编语言的源代码就足够了。

10.4　汇编语言的语法是“操作码 + 操作数”

在汇编语言中，1 行表示对 CPU 的一个指令。汇编语言指令的语法结构是**操作码 + 操作数**[①]（也存在只有操作码没有操作数的指令）。

操作码表示的是指令动作，操作数表示的是指令对象。操作码和操作数罗列在一起的语法，就是一个英文的指令文本。操作码是动词，操作数相当于宾语。例如，用汇编语言来分析“Give me money”这个英文指令的话，Give 就是操作码，me 和 money 就是操作数。汇编语言中存在多个操作数的情况下，要用逗号把它们分割开来，就像 Give me, money 这样。

能够使用何种形式的操作码，是由 CPU 的种类决定的。表 10-1 对代码清单 10-2 中用到的操作码的功能进行了整理，大家可以看一下。这些都是 32 位 x86 系列 CPU 用的操作码。操作数中指定了寄存器名、

① 在汇编语言中，类似于 mov 这样的指令称为“操作码”（opcode），作为指令对象的内存地址及寄存器称为“操作数”（operand）。被转换成 CPU 可以直接解析运行的二进制的操作码和操作数，就是本地代码。

内存地址、常数等。在表 10-1 中，操作数是用 A 和 B 来表示的。

表 10-1 代码清单 10-2 中用到的操作码的功能

操作码	操作数	功 能
mov	A，B	把 B 的值赋给 A
and	A，B	把 A 同 B 的值相加，并将结果赋给 A
push	A	把 A 的值存储在栈中
pop	A	从栈中读取出值，并将其赋给 A
call	A	调用函数 A
ret	无	将处理返回到函数的调用源

本地代码加载到内存后才能运行。内存中存储着构成本地代码的指令和数据。程序运行时，CPU 会从内存中把指令和数据读出，然后再将其存储在 CPU 内部的寄存器中进行处理（图 10-2）。

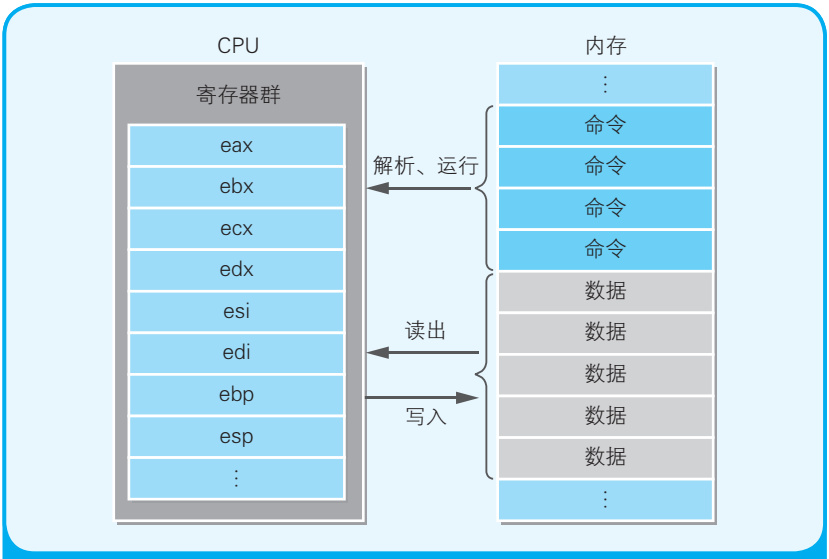

图 10-2 CPU 和内存的关系

寄存器是CPU中的存储区域。不过，寄存器并不仅仅具有存储指令和数据的功能，也有运算功能。x86系列CPU的寄存器的主要种类和角色如表10-2所示。寄存器的名称会通过汇编语言的源代码指定给操作数。内存中的存储区域是用地址编号来区分的。CPU内的寄存器是用eax及ebx这些名称来区分的。此外，CPU内部也有程序员无法直接操作的寄存器。例如，表示运算结果正负及溢出状态的标志寄存器及操作系统专用的寄存器等，都无法通过程序员编写的程序直接进行操作。

表10-2　x86系列CPU的主要寄存器[①]

寄存器名[②]	名　称	主要功能
eax	累加寄存器	运算
ebx	基址寄存器	存储内存地址
ecx	计数寄存器	计算循环次数
edx	数据计数器	存储数据
esi	源基址寄存器	存储数据发送源的内存地址
edi	目标基址寄存器	存储数据发送目标的内存地址
ebp	扩展基址指针寄存器	存储数据存储领域基点的内存地址
esp	扩展栈指针寄存器	存储栈中最高位数据的内存地址

① 表10-2中表示的寄存器名称是x86自带的寄存器名称。在第1章中表1-1列出的寄存器名称是一般叫法。两者有些不同，例如，x86的扩展基址指针寄存器就相当于第1章中介绍的基址寄存器。

② x86系列32位CPU的寄存器名称中，开头都带了一个字母e，例如eax、ebx、ecx、edx等。这是因为16位CPU的寄存器名称是ax、bx、cx、dx等。32位CPU寄存器的名称中的e，有扩展（extended）的意思。我们也可以仅利用32位寄存器的低16位，此时只需把要指定的寄存器名开头的字母e去掉即可。

10.5 最常用的 mov 指令

指令中最常使用的是对寄存器和内存进行数据存储的 **mov 指令**。mov 指令的两个操作数，分别用来指定数据的存储地和读出源。操作数中可以指定寄存器、常数、标签（附加在地址前），以及用方括号（[]）围起来的这些内容。如果指定了没有用方括号围起来的内容，就表示对该值进行处理；如果指定了用方括号围起来的内容，方括号中的值则会被解释为内存地址，然后就会对该内存地址对应的值进行读写操作。接下来就让我们来看一下代码清单 10-2 中用到的 mov 指令部分。

```
mov ebp,esp
mov eax,dword ptr [ebp+8]
```

mov ebp,esp 中，esp 寄存器中的值被直接存储在了 ebp 寄存器中。esp 寄存器的值是 100 时 ebp 寄存器的值也是 100。而在 mov eax,dword ptr [ebp+8] 的情况下，ebp 寄存器的值加 8 后得到的值会被解释为内存地址。如果 ebp 寄存器的值是 100 的话，那么 eax 寄存器中存储的就是 100 + 8 = 108 地址的数据。dword ptr（double word pointer）表示的是从指定内存地址读出 4 字节的数据。像这样，有时也会在汇编语言的操作数前附带 dword ptr 这样的修饰语。

10.6 对栈进行 push 和 pop

程序运行时，会在内存上申请分配一个称为栈的数据空间。栈（stack）有“干草堆积如山”的意思。就如该名称所表示的那样，数据在存储时是从内存的下层（大的地址编号）逐渐往上层（小的地址编号）累积，读出时则是按照从上往下的顺利进行（图 10-3）的。