

目录

第 11 章 多模块程序设计.....	198
11.1 多汇编语言模块化程序设计.....	198
11.2 C 程序和汇编语言程序的混合	201
11.2.1 函数的申明和调用	202
11.2.1 变量的申明和调用	203
11.3 内嵌汇编.....	204
11.4 模块程序设计中的注意事项.....	205
11.5 可执行文件的格式.....	206
习题 11.....	212
上机实践 11.....	213

第 11 章 多模块程序设计

面对较大型的程序设计任务时，将所有的模块（函数）放在一个文件中是不合适的，采用多模块的编程方法能更有效的管理程序，也有利于实施多人的分工合作。在一个解决实际问题的工程中，所有的模块可以都是用汇编语言编写的，也可以是多种不同语言编写的。本章介绍多模块程序设计方法，包括只用汇编语言编写各个模块，以及 C 语言程序和汇编语言程序混合的多模块。

11.1 多汇编语言模块化程序设计

在 C 语言程序设计中，可以在一个工程中包括多个 C 语言程序文件。在汇编语言程序设计中，一个工程也可以由多个汇编源程序文件组成。一个汇编语言源程序文件称为一个模块，它经过汇编程序汇编后，将生成一个目标文件，也称为一个目标模块。最后将多个目标文件链接成一个可执行文件。

在设计开发比较复杂的程序时，常常需要一个团队才能完成。在设计时，一般采用自顶向下、逐步求精的模块化和结构化的方法，将一个设计任务按其需要实现的主要功能，分解为若干相对独立的模块，并确定好各模块之间的调用关系和参数传递方式，对其中的公共部分还可以抽出来作为独立的公用子模块供大家调用。每个人在设计自己的一部分时，还可以采用自顶向下、逐步求精的方法进一步细化，分解成一些更小的模块，并将各模块的功能逐步细化为一系列的处理步骤或某种程序设计语言的语句，分别编写、调试，最后再将它们的目标模块连接装配成一个完整的整体。

在 C 语言程序设计中，在由多个 C 程序组成一个工程时，会出现定义和引用全局变量的问题，以及外部函数说明和调用问题。在多个汇编语言程序组成工程中也要解决这两个问题。

假设工程中含有 mainp.asm 和 subp.asm。不论在哪个文件中的 data 段中定义的变量都是全局变量。若在一个文件中定义的全局变量要在另一个文件中使用，则需要用 public 伪指令将变量说明为公共符号，用 extern 说明外部符号。

(1) 说明公共符号语句

语句格式： public 符号 [, 符号]

功 能： public 后的符号是公共符号，可以被其他模块引用。

public 后可以出现多个符号，符号之间以西文的逗号分隔。在程序中也可以用多个 public 语句，每个语句说明一个或者多个符号。public 语句可以放在程序的任何位置，但一般都放在程序的开头。

(2) 说明外部符号语句

语句格式： extern 符号:类型 [, 符号:类型]

功 能：用来说明本模块中需要引用的、由其它模块所定义的符号，即外部符号。

extern 也可以写成 extrn，后面可以出现一个或多个“符号:类型”对。出现的符号在定义它们的模块中必须被 public 伪指令说明为公共符号。符号可以是符号常量、变量，其类型可为：ABS (符号常量的类型)、byte、sbyte、word、sword、dword、sdword、qword、sqword 等，所有的符号类型必须与它们原定义时的类型一致。

一个模块中可以调用另一个模块中定义的子程序（函数），在调用模块，需要对被调用的子程序进行说明。说明的方法使用原型说明伪指令 proto，语句格式为：

函数名 proto [函数类型] [语言类型] [[参数名]:参数类型] [, [参数名]:参数类型]

下面给出了一个具体的例子。

【例 11.1】对两个缓冲区中的有符号双字数据分别按从小到大的顺序排序，然后输出排序结果。

设计函数 `sort` 实现排序功能，`display` 实现显示功能。将这两个函数放在子模块(`subp.asm`)中，主模块(`mainp.asm`) 调用中子模块中的函数，在子模块中使用了主模块中定义的全局变量 `lpFmt`，该变量是调用 `printf` 显示数值时使用的格式串。`sort` 的算法思想参见 7.3 节多重循环程序设计中的例子。

`mainp.asm` 的程序清单如下。

```
.686P
.model flat, c
ExitProcess proto stdcall :dword
includelib kernel32.lib
printf      proto :vararg
getchar     proto
sort        proto :dword, :dword
display     proto :dword, :dword
includelib libcmt.lib
includelib legacy_stdio_definitions.lib
public lpFmt
.data
lpFmt db "%d ", 0
crlf db 0DH, 0AH, 0
buf1 sdword -10, 20, 30, -100, 25, 60
N1     dword ($-buf1)/4
buf2 sdword -70, 55, 200, -150, 125, 90, -50
N2     dword ($-buf2)/4
.stack 200
.code
main proc
    invoke sort, offset buf1, N1
    invoke sort, offset buf2, N2
    invoke display, offset buf1, N1
    invoke printf, offset crlf
    invoke display, offset buf2, N2
    invoke getchar
    invoke ExitProcess, 0
main endp
end
```

子模块 `subp.asm` 程序如下。

```
.686P
.model flat, c
printf proto :vararg
extern lpFmt:sbyte
.code
```

```

; sort : 排序子程序, 对一个双字类型的数组按从小到大的顺序排序
; buf  : 输入缓冲区的首地址, 也是排序结果存放的首地址
; num  : 元素的个数
sort proc buf:dword, num:dword
    local outloop_num:dword
    .if (num<2)      ; 元素少于 2 个, 不用排序
        ret
    .endif
    mov  eax, num
    dec  eax
    mov  outloop_num, eax ; 外循环的次数
    mov  ebx, buf ; 数据缓冲区的首地址在 ebx 中
    mov  esi, 0 ; 外循环的控制指针
Out_Loop: ; 外循环
    cmp  esi, outloop_num
    jae  exit
        ; 下面是内循环
    lea  edi, [esi+1]
    Inner_Loop:
        cmp  edi, num
        jae  Inner_Loop_Over
        mov  eax, [ebx][esi*4]
        cmp  eax, [ebx][edi*4]
        jle  Inner_Modify
        xchg eax, [ebx][edi*4]
        mov  [ebx][esi*4], eax
    Inner_Modify: ; 修改内循环的控制变量
        inc  edi
        jmp  Inner_Loop
    Inner_Loop_Over:
        inc  esi
        jmp  Out_Loop
exit:
    ret
sort endp
; display: 显示数据子程序
; buf : 缓冲区首地址
; num : 显示的元素个数
display proc buf:dword, num:dword
    mov  ecx, num
    mov  ebx, buf
    .while (num>0)
        invoke printf, offset lpFmt, dword ptr [ebx]
        add  ebx, 4

```

```

        dec    num
    .endw
    ret
display endp
end

```

从上面的例子中可以看到，每个模块都是由子程序组成的，就像 C 语言程序都是由函数组成一样。每一个模块都要以 end 来结束。编译器从文件开始向下编译，分析到 end 后，就不再对下面的语句进行编译。在多个模块中，**只有一个模块在 end 之后给出一个标号或子程序名字，这是整个程序的入口点。**

对于上例中的子程序，可以用一些伪指令来控制循环、分支转移，程序如下。

```

sort  proc buf:dword, num:dword
    local  outloop_num:dword
    .if (num<2)
        ret
    .endif
    mov  eax, num
    dec  eax
    mov  outloop_num, eax ; 外循环的次数
    mov  ebx, buf ; 数据缓冲区的首地址在 ebx 中
    mov  esi, 0 ; 外循环的控制指针
    .while (esi < outloop_num)
        ; 下面是内循环
        lea  edi, [esi+1]
        mov  eax, [ebx][esi*4]
        .while (edi < num)
            .if  eax >= sdword ptr [ebx][edi*4]
                xchg eax, [ebx][edi*4]
            .endif
            inc  edi
        .endw
        mov  [ebx][esi*4], eax
        inc  esi
    .endw
    ret
sort  endp

```

在编写程序时，要注意有符号或无符号的数据的差别。例如，在“`.if eax >= sdword ptr [ebx][edi*4]`”中要加 `sdword ptr`，即说明数据类型是有符号数，否则会当成无符号数的比较。

11.2 C 程序和汇编语言程序的混合

在程序开发需要时，可以根据各模块所要完成的功能选用不同的计算机语言编程，充分发挥各种程序设计语言的优势。例如，一般问题的处理可用 C++ 语言编程；与硬件有关的部分或者对执行效率要求很高的部分可用汇编语言编程，这样就发挥汇编语言程序占用存储空

间小、运行速度快、能直接控制硬件的优点。最后将它们经编译和汇编后生成的目标文件连接成一个整体。这样编写程序效率高，简短、清晰、可靠性高，容易阅读，方便修改和扩充，便于交流。对于一些功能还可以建立函数库，供不同的程序开发任务（工程）使用。

11.2.1 函数的申明和调用

可以在 C 语言程序中调用用汇编语言编写的函数，反过来也一样。

【例 11.2】 编写一个程序，输入 5 个整型数据，对它们按从小到大的顺序排序，输出排序结果。

主程序（mainp.c）实现数据的输入和输出，用 C 语言编写，程序如下。排序函数 sort 用汇编语言编写，见 11.1 节。

```
#include <stdio.h>
#include <conio.h>
void sort (int *, int);
int main()
{
    int a[5];
    int i;
    for (i = 0; i < 5; i++)
        scanf_s("%d", &a[i]);
    printf("\n result after sort \n");
    sort(a, 5);
    for (i = 0; i < 5; i++)
        printf("%d ", a[i]);
    _getch();
    return 0;
}
```

从该程序中，可以看到对于调用用汇编语言编写的函数与调用用 C 语言编写的函数没有差别。在汇编语言程序模块中，对要被外部调用的函数，可以不用做特别的说明。这就像多个 C 程序文件组成一个工程一样，一个文件中定义的函数，被另一个文件中的函数调用，在定义函数的文件中不需要做特别的说明，在调用函数的文件中要进行函数说明。

在汇编语言程序中，调用 C 语言的函数的例子在前面的例子中已出现过多次，如 printf、scanf_s、_getch、clock 等，只要正确的使用函数原型说明伪指令 proto 即可。

注意，在 C 语言程序的文件后缀名为 cpp 时，如将上面的 mainp.c 改为 mainp.cpp，编译时没有报错，但在链接时会报错，“无法解析的外部符号 void _cdecl sort(int *,int) (?sort@@YAXPAHH@Z)”。这是因为编译器看到文件是 cpp 时，按照 C Plus Plus (C++) 的规范解析符号，会产生一个新的名称。C++ 是面向对象的程序设计语言，允许出现参数不同的同名函数（函数重载），为了区分这些可能出现的同名函数，采用了名称修饰规则将其变成了另一个符号。对于汇编语言程序在编译时保持了原有的名字，因而在链接时出现了占不到符号的情况。

为了解决符号名称的问题，在 C++ 程序中，可以使用 extern “C” 来说明按 C 语言的规则来解析符号。从 maip.c 变成 mainp.cpp，只需要做如下修改

原函数的说明： void sort (int *, int);

修改后的说明： extern "C" void sort (int *, int);

另外，需要注意的两个细节。

(1) 函数名的大小写要一致

在 C、C++ 语言程序设计中，函数名称是区分大小写的。而在汇编语言程序中，在默认状态下，名称是不区分大小写的。为了让 C 语言程序能调用汇编语言写的函数，要求两者的函数命名一致。

(2) 语言类型申明要一致

在汇编语言程序中，可使用存储模型说明伪指令 `model` 指定语言类型。语言类型包括 `c`、`pascal`、`stdcall` 等。也可以在函数定义为指令 `proc` 中指定语言类型。当两处说明的语言类型不同时，以在 `proc` 中说明的语言类型为准。当 `proc` 中未指明语言类型时，才使用模型说明伪指令中的语言类型。

若在汇编语言程序中，使用的语言类型 `C`，则在 C 程序中说明也要用相同的申明。

`extern "C" void sort(int *, int);` 等价于 `extern "C" void __cdecl sort(int *, int);`

若在汇编语言程序中，使用的语言类型 `stdcall`，则在 C 程序中说明也要用 `stdcall` 申明。说明语言如下。

`extern "C" void __stdcall sort(int *, int);`

注意，Windows 操作系统提供的 API 函数一般都是 `stdcall` 类型的。

11.2.1 变量的申明和调用

除了函数调用外，还有一个问题是全局变量的定义和引用。在汇编语言程序的 `data` 段中定义的变量为全局变量，若该变量要给其他模块（其他汇编语言程序、C 程序、C++）使用，则需要在汇编语言模块中，用 `public` 来说明为公共符号。反过来，如汇编语言程序模块中使用了其他模块（其他汇编语言程序、C 程序）中定义的全局变量，则需要用 `extern` 或者 `extrn` 来说明外部符号。在 C 语言程序中，使用汇编模块中的全局变量，则它像引用另一个 C 程序中定义的全局变量一样，用 `extern` 来说明。但在 C++ 程序中，引用汇编模块中的全局变量，则要写出 `extern "C"` 的形式。对于变量名，在 C++ 程序中同样会使用修饰的名称。

注意，在 C 语言程序中，要按 C 语言的语法来申明引用的外部的全局变量，在汇编语言程序中要按汇编语言的语法规则来写。

例如，在 “.c” 文件中有：

```
int x;
extern int y;
```

在 “.cpp” 文件中有：

```
extern "C" int z;
```

在汇编源程序中有：

```
public y
public z
extern x:sdword
y sdword .....
z sdword .....
```

C 语言中的 `int` 类型与汇编语言中的 `sdword` 类型对应。

在全局变量为数组时，要注意 C 语言和汇编语言的差异，汇编语言程序中是给出的元素地址是按字节编址的，而 C 语言程序中，编译器会自动的由元素下标转换为相应的字节地址。

例如，在 C 语言程序中，设定义有全局变量 `int x[5];`；在汇编语言模块中说明为 `extern x:sdword`。

```
mov x, 10      等价于 x[0]=10;
mov x[4], 20   等价于 x[1]=20;
mov x[8], 30   等价于 x[2]=30;
```

`mov x[8], 30` 等价于 `mov x[2*type sdword], 30`。

对于结构类型的变量,用法是类似的。只是要注意 C 程序编译时结构中字段的对齐方式,在编译时,代码生成中的结构成员对齐应采用“1 字节 (/Zp1)”方式,或者在程序中使用 `#pragma pack(1)` 语句。

11.3 内嵌汇编

内嵌汇编是在 C 程序中插入汇编语句(inline assembly)。

内嵌汇编程序和一般的汇编程序段并没有太大的差别,汇编指令它可以出现在任何允许 C/C++ 语句出现的地方,只需要在汇编指令前加上 `__asm` (`asm` 的前面有两个下划线),具体形式有两种,一种是将一段汇编语句指令用“{}”括起来,前面加上 `__asm`, 如:

```
__asm
{
    汇编语言指令
}
```

另一种方法是在每条汇编指令之前加 `__asm` 关键字, 例如:

```
__asm mov eax, sum
__asm mov ebx, 1
```

在内嵌汇编中可以使用汇编语言的注释,即以“;”开头到行尾的部分为注释,另外也可以使用 C/C++ 风格的注释。

【例 11.3】 计算从 1 累加到 100 的和, 并且显示出和。

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    int sum;
    sum=0;
    __asm {
        mov eax, sum        ; eax 用来存放和
        mov ebx, 1          ; ebx 为循环计算器
    L1:
        cmp ebx, 100
        jg L2
        add eax, ebx
        inc ebx
        jmp L1
    L2:mov sum, eax
    }
    printf("%d\n", sum);
    return 0;
}
```

在该例中给出了对单个整型变量 `sum` 的访问方法,对于数组变量和结构变量的访问是类似于纯汇编语言程序的访问方法,可以简单的认为它们就是汇编语言模块中定义的变量。

11.4 模块程序设计中的注意事项

程序设计是一门艺术。实现相同功能的程序可以多种多样，编程的变化多、弹性大，经过精心设计，能将程序的效率发挥得淋漓尽致。就如同画家的画笔，如果只是为了谋生，用它画出的只是廉价的商品；一旦投入自己的理想与心智，画出的作品将升华成为艺术，进入一个更高的境界！

但是，从另外的角度来看，程序设计也是一门工程。工程就有工程的规范要遵循，不能随心所欲。从总体和全局的角度来掌控程序设计，更要将当成一个工程。不论是模块的划分、变量和函数的命名、注释的写法等都要遵循工程的规范，遵循一定的原则。

1. 模块的划分

在模块化程序设计中，首要解决的关键问题是正确地划分模块。模块划分的原则是模块内具有高内聚度、模块间具有低耦合度。具体的可参考下面的八条规则。

- (1) 各个模块应在功能上和逻辑上相互独立，尽量截然分开。
- (2) 每个模块的结构应尽量设计成单入口单出口的形式，避免使用转移语句在模块间转来转去。单入口单出口的模块便于调试、阅读和理解，它的可靠性更高。
- (3) 各模块间的接口应该简单，要尽量减少公共符号的个数，尽量不共用数据存储单元，在结构或编排上有联系的数据应放在一个模块中，以免互相影响，造成查错困难。
- (4) 力求使模块具有通用性，通用性越强的模块利用率越高。这就要求在模块中尽可能不用全局变量。
- (5) 一个模块既不能过大也不能过小。过大的模块功能复杂，通用性较差；模块过小则会造成要很多模块组装在一起才能完成较大功能，增大组装的工作量。一般情况下，一个模块的长度应在一个屏幕上能完整的显示出来。
- (6) 如果一个程序段被很多模块所公用，则它应是一个独立的模块。
- (7) 如果若干个程序段处理的数据是公用的，则这些程序段应放在一个模块中。
- (8) 若两个程序段的利用率差别很大，则应分属于两个模块。

2. 变量和函数的命名

变量和函数的命名最重要的原则是容易记忆、容易理解。看到一个名字，很容易想到它对应的功能和作用。在程序设计中，变量和函数名等广泛采用的是匈牙利表示法。这是为了纪念 Microsoft 公司的匈牙利籍的程序员 Charles Simonyi 提出的一套变量和函数的命名方法。在匈牙利表示法中，变量名以一个小写字母开始，代表变量的类型。后面附以变量的名字，变量名以意义明确的大小写混合字母序列所构成。这种方案允许每个变量都附有表征变量类型的信息。

常用的前缀有：b (byte), w (word), dw (dword), h (handle, 句柄), lp (long pointer, 指针), sz (string zero, 以 0 结尾的字符串), lpsz (指向以 0 结尾的字符串的指针), f (float)。

例如，用一个变量来保存一个图像的宽度，宽度不超出 10000，可以将变量命名为：wWidth。定义时，可用语句 `local wWidth:word`。如果程序中想写语句 `“mov eax, wWidth”`，就比较容易知道两个操作数的类型不匹配，而不用等到编译时指出错误。

函数和变量命名方式相同，但是没有前缀。换句话说，函数名的第一个字母要大写。此外，所有的类型和常量都是大写字母，但名字中可以允许使用下划线。

3. 注释

如果一个程序没有注释，阅读起来是很困难的，但是写注释也有很多讲究。

- (1) 不要写无意义的注释

例如 `mov eax, 0` ；将 0 送给 `eax`。

上面的注释就是无意义的注释，不要认为阅读程序的人连基本指令的功能都不清楚。

(2) 注释表达的含义准确到位

这实际上是考验写注释的人表达能力。例如，在一个函数的功能注释中，“将组数 `a` 中的数重新排列”、“将组数 `a` 中的数排序”、“将组数 `a` 中的数按从小到大的顺序排序”，三者表达的准确性差别是很大的。

(3) 在函数或子程序前应有注释

函数的功能、输入参数、输出参数、返回值都应在函数开头前进行注释。若函数的处理较复杂，还应对应算法思想和处理过程进行注释。由于汇编语言非常琐碎，还应对函数中寄存器的功能分配进行注释，让一些寄存器在函数中的用途保持不变，这样思路不会被随意打乱，才能更容易理解程序。

(4) 对于结构和变量进行注释

对于程序中定义的结构类型应给予注释，说明结构存放什么样的信息，各个字段的含义。对于程序中用途的一些变量（非临时变量）也应该给予注释，说明其含义。

11.5 可执行文件的格式

运行在微软 Windows 操作系统下的可执行二进制文件采用 PE（Portable Executable，可移植的执行体）格式。本节只是对 PE 格式进行简单的介绍。通过分析二进制的执行文件，可以了解各种信息的存放规律，探索文件格式设计的奥秘。对于分析和防治 PE 文件病毒、文件加密和解密任务，则需要更深入地掌握 PE 文件格式。

PE 文件格式是由 COFF（Common Object File Format，通用目标文件格式）发展出来的，在结构上相似，都是基于段的结构。在 64 位 Windows 操作系统中，PE 文件格式稍微有点修改，称为 PE32+ 格式。PE 文件由 DOS MZ 头部、DOS 实模式残余程序（Dos Stub）、PE 文件标志、PE 文件头、PE 文件可选头、各节（Section）头部、各节的具体数据依次组成。

DOS 部分	DOS MZ 头	IMAGE_DOS_HEADER
	DOS Stub	
PE 头 IMAGE_NT_HEADERS32	PE 文件标志	‘PE’，0,0（50 45 00 00）
	PE 文件头	IMAGE_FILE_HEADER
	PE 文件可选头	IMAGE_OPTIONAL_HEADER32
节表	Section 1 头（节 1 的头部）	IMAGE_SECTION_HEADER
	Section头
	Section n 头（节 n 的头部）	IMAGE_SECTION_HEADER
节	Section 1（节 1 的实际数据）	
	Section	
	Section n（节 n 的实际数据）	

图 11.1 PE 文件的结构

1、DOS MZ 头

PE 文件中包括 DOS 部分的主要目的是为了可执行文件的向下兼容。假设在 Windows 系统中生成了一个 PE 文件，将该文件放入 DOS 系统下去执行，DOS 系统无法识别 PE 文件头，就会出现错误。在 PE 文件中包含的 DOS 部分是标准的 DOS MZ 格式（即 DOS 下 EXE 文件格式），可在 DOS 下执行，从而解决了兼容性问题。

PE 文件的开头是 MS-DOS MZ 头部，它由 64 个字节组成，它对应的结构为

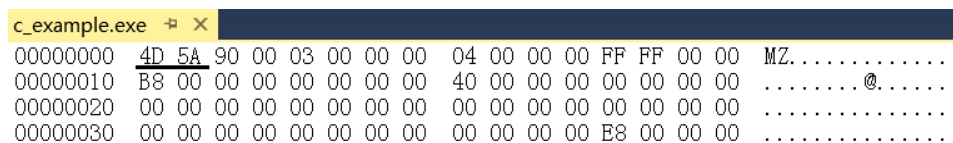
IMAGE_DOS_HEADER (DOS 头映像), 详细的定义在 winnt.h 中。

```
typedef struct _IMAGE_DOS_HEADER {    // DOS .EXE header
    WORD   e_magic;                  // Magic number
    .....
    WORD   e_ip;                     // Initial IP value
    WORD   e_cs;                     // Initial (relative) CS value
    WORD   e_lfarlc;                 // File address of relocation table
    .....
    LONG   e_lfanew;                 // File address of new exe header
} IMAGE_DOS_HEADER;
```

IMAGE_DOS_HEADER 开头是一个字节数据字段 e_magic (魔术数字), 占两个字节, 它的值是 “4D 5A”, 是一种特定的标记, 对应 ‘MZ’ 的 ASCII 码, 即 e_magic=0x5A4D。有了 DOS MZ 头, 一旦程序在 DOS (Disk Operating System, 磁盘操作系统) 下执行, DOS 就能识别出这是有效的执行体, 然后运行紧随 MZ header 之后的 DOS 程序, 即 DOS Stub。该程序是从 e_lfarlc 指明的文件位置开始的 (重定位表在文件中的起始位置), 以该地址为参考, 用 e_ip 和 e_cs 指明第一条指令的相对地址, 这样就实现了对 DOS 系统的兼容。

DOS 头中最后 4 个字节 “E8 00 00 00”, 即 e_lfanew 对应的数值为 0x000000E8, 表示新 exe 头在文件中的位置, 即 PE 文件标志在文件中的位置。注意, 在 IMAGE_DOS_HEADER 中前面的字段与标准的 DOS 文件头相同, Windows 在后面添加了 5 个字段, 其中就包括 e_lfanew。DOS 不是对这几个新加的字段进行解释的。

如图 11.2, 给出了可执行文件 c_example.exe 文件的 MZ 头信息。



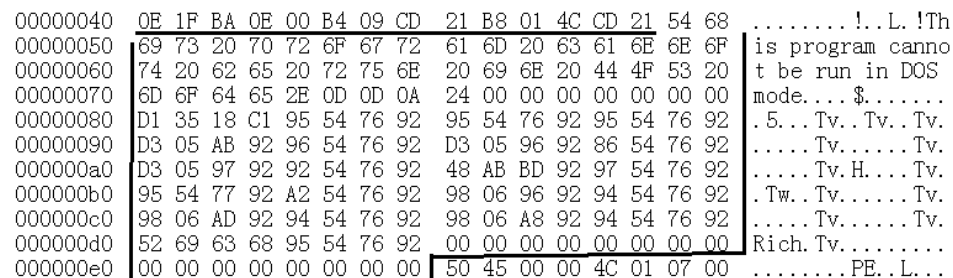
```
c_example.exe  [icon] X
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ.....
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  .....@.....
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000030  00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00  .....

```

图 11.2 可执行文件的 MZ 头

2、DOS Stub

DOS 实模式残余程序 (Stub) 实际上是个有效的 EXE, 在不支持 PE 文件格式的操作系统中, 它将简单显示一个错误提示, 大多数情况下它是由汇编编译器自动生成。通常它简单调用 DOS 操作系统提供的功能号为 9 的中断 21H 程序来显示字符串 “This program cannot be run in dos mode”。当然这不是必须的, 可以要保留其大小, 使用 00 来填充这些字节。如图 11.3, 给出了可执行文件 c_example.exe 文件的 DOS Stub 信息, 它从文件的 0040H 处开始到 00E7H 处结束。



```
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ....!..L.!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.
00000080  D1 35 18 C1 95 54 76 92 95 54 76 92 95 54 76 92  .5...Tv..Tv..Tv.
00000090  D3 05 AB 92 96 54 76 92 D3 05 96 92 86 54 76 92  ....Tv.....Tv.
000000a0  D3 05 97 92 92 54 76 92 48 AB BD 92 97 54 76 92  ....Tv.H....Tv.
000000b0  95 54 77 92 A2 54 76 92 98 06 96 92 94 54 76 92  .Tw..Tv.....Tv.
000000c0  98 06 AD 92 94 54 76 92 98 06 A8 92 94 54 76 92  ....Tv.....Tv.
000000d0  52 69 63 68 95 54 76 92 00 00 00 00 00 00 00 00  Rich.Tv.....
000000e0  00 00 00 00 00 00 00 00 50 45 00 00 4C 01 07 00  ....PE..L...

```

图 11.3 DOS Stub 示例

在图 11.3 中, 可以看到 DOS Stub 的前 14 个字节是 “0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21”, 这一串机器码对应的汇编语言程序如下:

```

0E:      push  cs
1F:      pop   ds          ; (ds)= (cs)
BA0E00:  mov   dx, 000EH
B409:      mov   ah, 09
CD21:      int  21H        ; 9 号功能调用, 输出从 ds: 000EH 处开始的串
B8014C:  mov   ax, 4C01H
CD21:      int  21H        ; 返回操作系统

```

该程序将显示从 000E 处开始的一个串, 串以 '\$' (即 24H) 为结束符, '\$' 不会显示出来。在图 11.3 的右边可以看到显示的串的信息。

3、PE 头

① PE 头的结构

在 Dos Stub 之后是 PE 文件标志、PE 文件头和 PE 文件可选头, 这三部分的结构由 IMAGE_NT_HEADERS32 或者 IMAGE_NT_HEADERS64 定义, 该结构定义在 winnt.h 中。

```

typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32;

```

Signature 由 4 个字节组成, 表示 PE 文件标志。对于 c_example.exe, 从上面的分析可知, PE 文件标志是从文件地址 000000E8H 处开始的, 它内容是 “50 45 00 00”, 即 Signature 的值为 0x00004550。

② PE 文件头

PE 文件头 FileHeader 的结构由 IMAGE_FILE_HEADER (映像文件头) 定义, 结构如下。

```

typedef struct _IMAGE_FILE_HEADER {
    WORD Machine;
    WORD NumberOfSections;
    DWORD TimeDateStamp;
    DWORD PointerToSymbolTable;
    DWORD NumberOfSymbols;
    WORD SizeOfOptionalHeader;
    WORD Characteristics;
} IMAGE_FILE_HEADER;

```

该结构的长度是 20 个字节。

Machine : 2 个字节, 是 CPU 标识, 指明了文件的运行平台, 即编译链接生成该文件时的平台。由于不同平台上的指令机器码是不同的, 在运行程序时需要此时的运行平台是否适用于执行文件中标明的平台。例如, 在 c_example.exe 中, Machine 对应的值为 0x014C, 表示 Intel 80386 CPU; 在 winnt.h 中可以看到机器表示的宏定义, 如 “#define IMAGE_FILE_MACHINE_I386 0x014c”。

NumberOfSections: 2 个字节, 表示程序的节数。

TimeDateStamp: 4 个字节, 文件创建的日期时间戳时间, 是从 1970 年 1 月 1 日 00:00:00 开始算起, 到文件创建时之间秒数; 时间标准是格林尼治时间。

PointerToSymbolTable: 4 个字节, COFF 符号表在文件中的起始位置, 当没有 COFF 符号表时, 该值为 0;

NumberOfSymbols: 4 个字节, COFF 符号表中符号的个数。PointerToSymbolTable 和

NumberOfSymbols 与调试用的符号表有关。

SizeOfOptionalHeader: 2 个字节, 是 IMAGE_OPTIONAL_HEADER32 结构的大小 (224 个字节, 即 0x00E0) 或 IMAGE_OPTIONAL_HEADER64 结构的大小 (240 个字节);

Characteristics: 2 个字节, 文件属性, 由 16 个二进制位组成, 每位代表的含义也定义在 winnt.h 中, 例如:

```
#define IMAGE_FILE_EXECUTABLE_IMAGE 0x0002 // File is executable
#define IMAGE_FILE_32BIT_MACHINE 0x0100 // 32 bit word machine
```

在图 11.4 中的文件属性为 0x0102, 即表示是一个 32 位的可执行文件。

```
000000e0 00 00 00 00 00 00 00 00 50 45 00 00 4C 01 07 00 .....PE..L...
000000f0 46 C6 12 5D 00 00 00 00 00 00 00 00 E0 00 02 01 F..].....
```

图 11.4 PE 文件标志和 PE 文件头

其他的文件属性还包括: 是否为 DLL 文件、是否为系统文件 (如驱动程序)、是否包含调试信息、能否从可移动盘运行、能否从网络运行、是否存在符号信息、小尾/大尾方式等。

③ PE 文件的阅读工具

以二进制形式将文件内容映射到文件结构上是一件琐碎的工作, 在掌握二进制文件的解读方法后, 可以使用一些更直观显示信息的工具来解读文件结构。PEViewer 就是其中之一。如图 11.5 所示, 用 PEViewer 开一个 c_example.exe 文件, 可以较清楚地看到文件的各组成部分, 每部分的组成字段、字段在文件中的位置、取值以及含义。

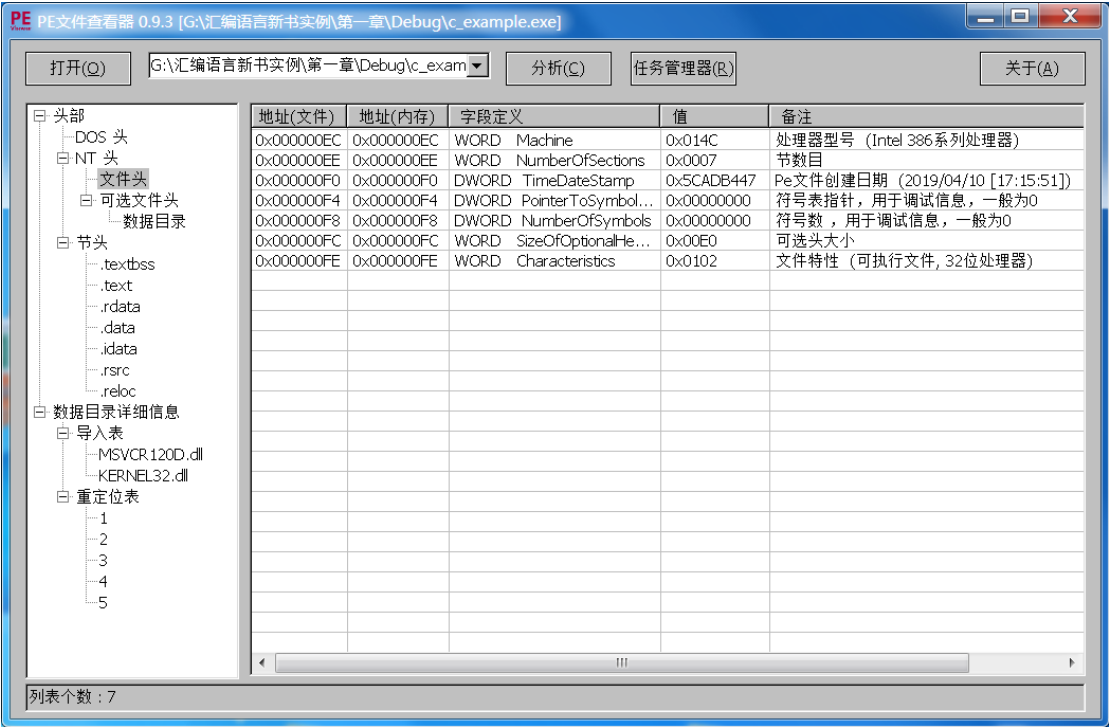


图 11.5 c_example.exe 的 PE 文件头信息

④ PE 文件可选头

PE 文件可选头 OptionalHeader 由结构 IMAGE_OPTIONAL_HEADER32 或 IMAGE_OPTIONAL_HEADER64 定义。该结构较大, 字段较多。图 11.6 给出了 c_example.exe 的可选文件头。

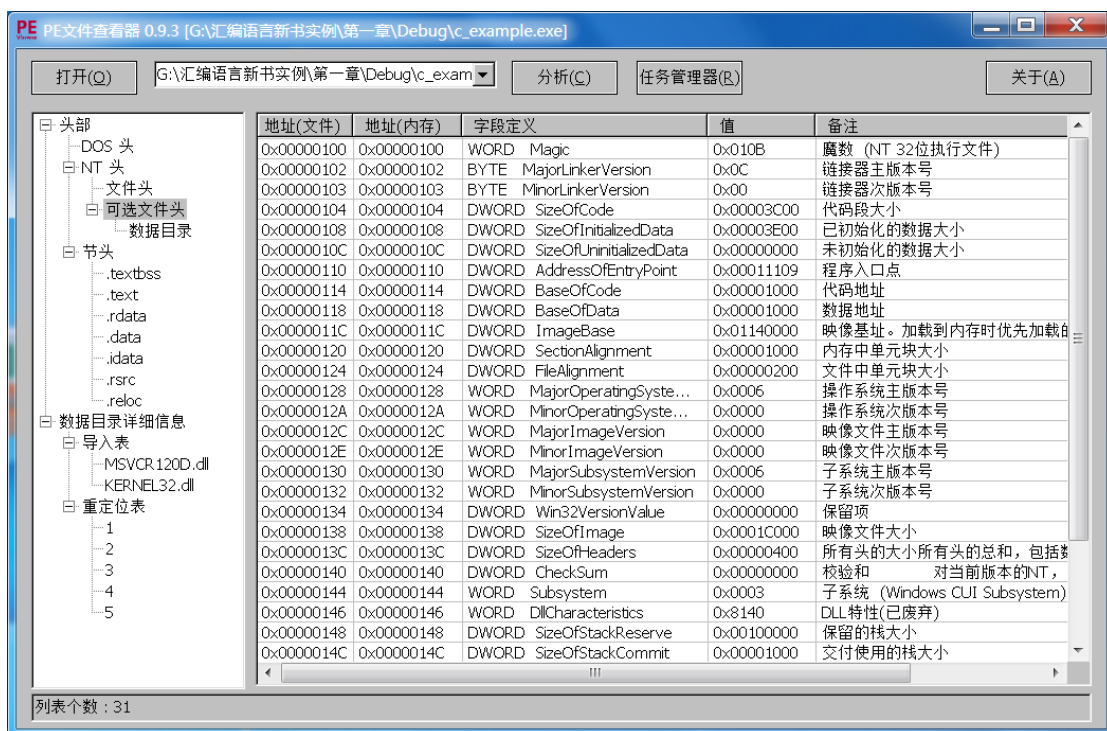


图 11.6 c_example.exe 的 PE 可选文件头信息

此结构中大部分字段都不重要，重要的字段有：AddressOfEntryPoint、ImageBase、SectionAlignment、FileAlignment、Subsystem、DataDirectory。

AddressOfEntryPoint：执行程序入口地址，该地址并不是程序入口点在 exe 文件中的位置，而是将程序装载到内存后，程序入口点在内存中的位置与文件在内存中的起始位置之间的距离，是一个相对的虚拟地址（Relative Virtual Address, RVA）。设将 PE 文件装入到起始地址为 0x00400000 的内存中，RVA 为 0x00011109，则程序的入口点在内存 0x00411109 处。注意，在程序的装载时，DOS 头、PE 头、节表、节都要装入内存，但是装入后节的大小会发生变化，故 RVA 不等于文件中的位置偏移。

ImageBase：如果 ImageBase 指定的地址未被其占用，Windows 会优先将文件装载到该地址处。链接器在生成可执行文件时，使用该地址来生成机器代码，若能装入此处，就不需要进行重定位操作，装载速度最快。

SectionAlignment：节装入内存时的地址对齐单位，节装入的地址必须是能被该值整除的。在图 11.6 中，SectionAlignment = 0x1000，即 4096。假设一个节在文件中的大小为 1024，装入内存后，也要给它分配 4096 个字节的空间，这样一个节的起始地址才能是 4096 的倍数。

FileAlignment：节在文件中的对齐单位，即在文件中以 FileAlignment 为单位为节分配文件中占用的空间。

Subsystem：程序使用界面的子系统。在图 11.6 中，Subsystem=3，表示 Windows 控制台界面（IMAGE_SUBSYSTEM_WINDOWS_CUI=3）。程序运行时，Windows 操作系统将自动为程序建立一个控制台窗口。

DataDirectory：它是一个数组，共 16 个元素，每个元素的长度为 8 个字节，包括 2 项 VirtualAddress 和 Size，用来指明各个数据块的位置和大小。数据按用途可分为导出表、导入表、资源、异常、安全、重定位等多种类型。与 AddressOfEntryPoint 一样，VirtualAddress 是一个相对的虚拟地址 (RVA)，如图 11.7 所示，给出了可选文件头中的最后的部分 IMAGE_data_DIRECTORY。

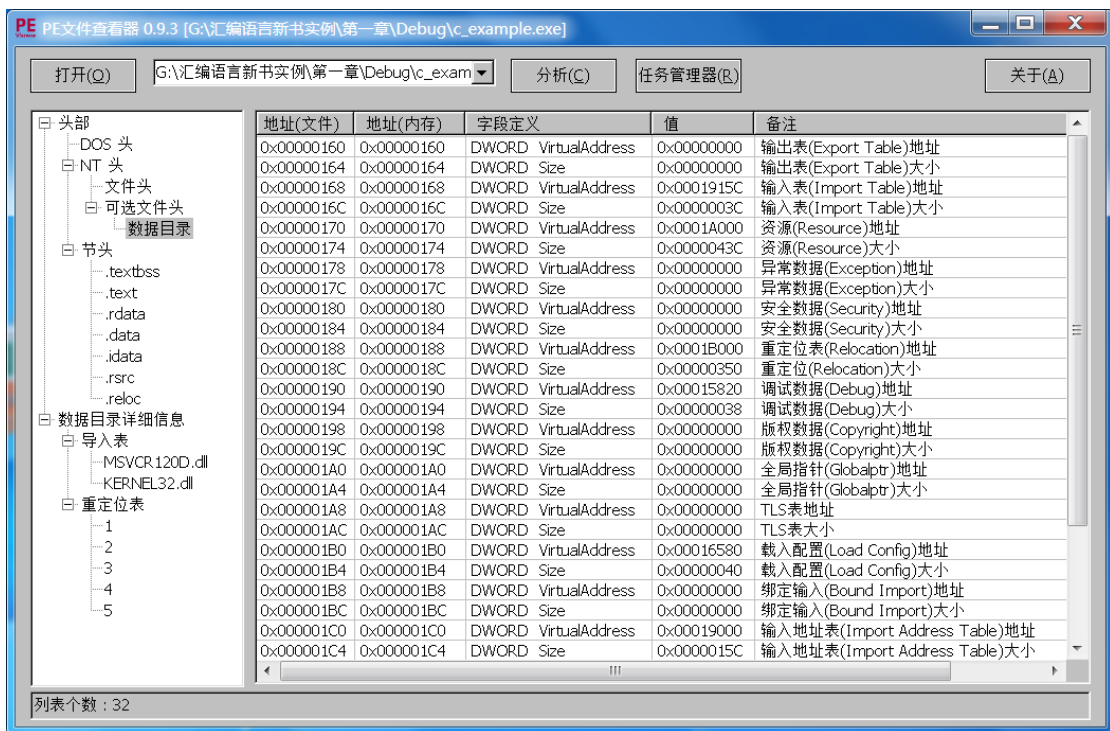


图 11.7 可选文件头中最后的部分-数据目录

4、节表

在可选文件头之后是节表，即程序的各个组成段（节）的描述信息。

在 `winnt.h` 中，其定义如下：

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME]; // IMAGE_SIZEOF_SHORT_NAME=8
    union {
        DWORD    PhysicalAddress;
        DWORD    VirtualSize; // 节的大小，在对齐处理前的实际大小
    } Misc;
    DWORD    VirtualAddress; // 节被装载到内存后的相对虚拟地址(RVA)
    DWORD    SizeOfRawData; // 在磁盘文件上的大小（用 FileAlignment 对齐后）
    DWORD    PointerToRawData; // 装载前在磁盘文件上的偏移量（从文件头算起）
    DWORD    PointerToRelocations; // 重定位指针
    DWORD    PointerToLinenumbers; // 行数指针
    WORD     NumberOfRelocations; // 重定位数目
    WORD     NumberOfLinenumbers; // 行数数目
    DWORD    Characteristics; // 节的属性
} IMAGE_SECTION_HEADER;
```

对于 `c_example.asm` 共有 7 个节，分别是 `textbss`、`text`、`rdata`、`data`、`idata`、`rsrc`、`reloc`。在图 11.8 中给出了 `text` 节的信息。

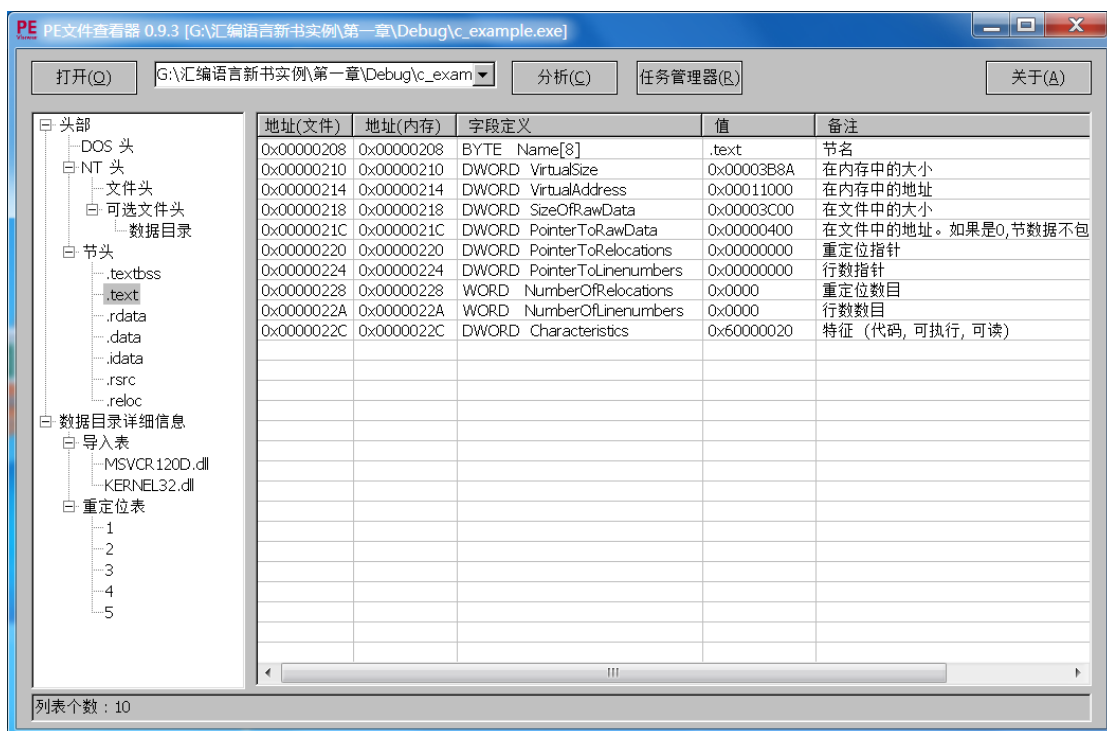


图 11.8 text 节表信息

在 text 节表信息中, 可以看到代码段是从文件 c_example.exe 的 0x00000400 开始的, 它由 PointerToRawData 指明, 该代码段映射到内存中的虚拟地址是 0x00011000, 由 VirtualAddress 指明。由 PE 可选文件头的 IMAGE_OPTIONAL_HEADER32 中的 AddressOfEntryPoint (0x00011009 见图 11.6) 知, 要执行的程序的第一条执行在 $0x00011009 - 0x00011000 + 0x00000400 = 0x00000509$ 处。文件 0x00000509 处的机器代码是 “E9 52 0A 00 00”, 是一条 jmp 指令。

习题 11

11.1 设有一个缓冲区中连续存放有多个学生信息记录, 每个记录中包括学生的姓名 (10 个字节, 以 0 结束)、和一个考试成绩 (字数据)。用多模块方法编写一个程序, 完成按分数从高到低的排序, 输出排序后的结果 (姓名和成绩)。要求:

- ① 主模块中完成数据显示功能;
- ② 子模块中完成排序功能, 子模块中不得使用外部变量;
- ③ 两个模块都用汇编语言编写。

学生人数自定, 学生的初始信息自定。

11.2 定义一个学生结构, 存放学生的姓名 (10 个字节) 和考试成绩 (字数据)。用多模块方法编写一个程序, 完成如下功能。

① 主程序中定义结构数组变量, 程序运行时输入学生的姓名和成绩; 按成绩从高到低输出学生的姓名和成绩;

② 子模块完成排序功能, 子模块中不得使用外部变量;

③ 主程序模块用 C 语言编写; 子模块用汇编语言编写。

上机实践 11

11.1 设计实现一个网店商品信息查询的程序。

有一个老板开了一个网店，网店里 n 种商品销售。每种商品的信息包括：

商品名称（10 个字节，名称不足部分补 0）

进货价（字类型）

销售价（字类型）

进货总数（字类型）

已售数量（字类型）

利润率（字类型）

利润率由程序自动计算， $\text{利润率}(\%) = (\text{销售价} * \text{已售数量} - \text{进货价} * \text{进货总数}) * 100 / (\text{进货价} * \text{进货总数})$ 。

系统有两种类型的用户，一是老板，二是一般的访客。老板管理网店信息时需要输入自己的名字（10 个字节，不足部分补 0）和密码（6 个字节，不足部分补 0），登录后可查看商品的全部信息；一般访客无需登录，可以查看网店中商品的名称，销售价格。他们都可以只查询某一种商品的信息（输入商品名后查询，对一般访客只能查该商品的销售价格）。