

《程序是怎样跑起来的》（下）



c747190cc2f5 (/u/c747190cc2f5)

2019.05.05 16:21* 字数 4194 阅读 17 评论 0 喜欢 0
(/u/c747190cc2f5)

学习笔记

第8章 从源文件到可执行文件

本章问题：

1. CPU 可以解析和运行的程序形式称为什么代码？
2. 将多个目标文件结合生成 EXE 文件的工具称为什么？
3. 扩展名为 .obj 的目标文件的内容，是源代码还是本地代码？
4. 把多个目标文件收录在一起的文件称为什么？
5. 仅包含 Windows 的 DLL 文件中存储的函数信息的文件称为什么？
6. 在程序运行时，用来动态申请分配的数据和对象的内存区域形式称为什么？

问题

本章重点：

编译器的功能；程序从源代码到可执行文件的流程；程序运行时的堆和栈。

8.1 计算机只能运行本地代码



代码清单 8-1 求解平均值的程序

```
#include <windows.h>
```

一个例子1

```
// 返回两个参数的平均值的函数
double Average(double a, double b) {
    return (a + b) / 2;
}

// 程序运行启始位置的函数
int WINAPI WinMain(HINSTANCE h, HINSTANCE d, LPSTR s, int m)
{
    double ave; // 保存平均值的变量
    char buff[80]; // 保存字符串的变量

    // 求解 123,456 的平均值
    ave = Average(123, 456);

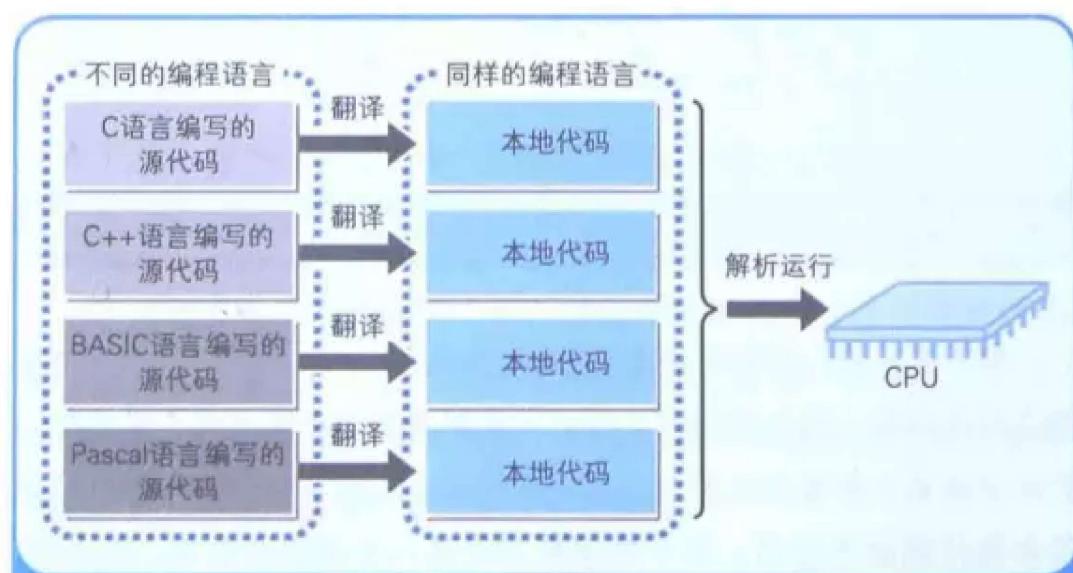
    // 编写显示在消息框中的字符串
    sprintf(buff, "平均值 = %f", ave); (3) (2)

    // 打开消息框
    MessageBox(NULL, buff, title, MB_OK); (4)

    return 0;
}
```

一个例子2

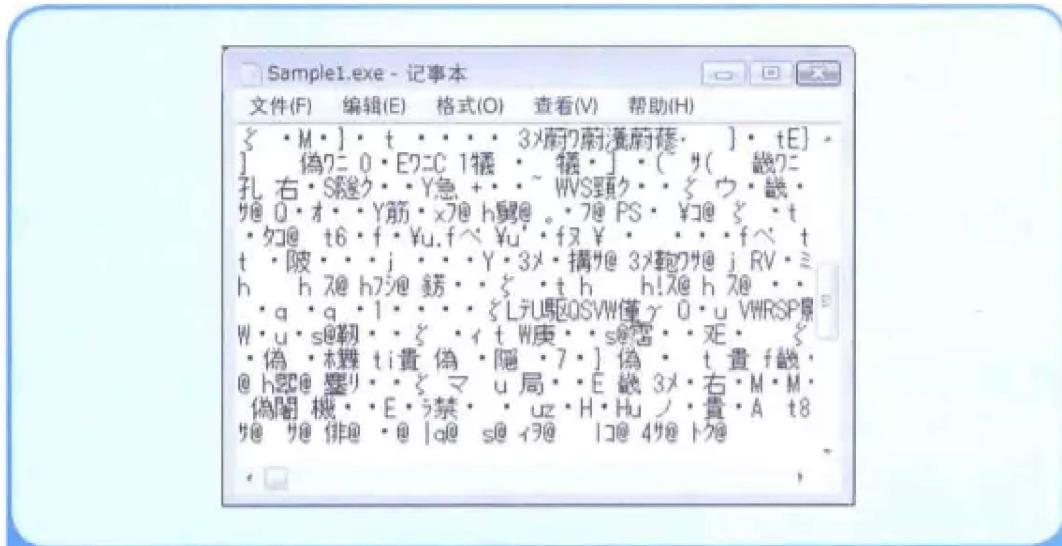
图中栗子的源代码文件命名为Sample1.c。



源代码需要转换成本地代码才能运行

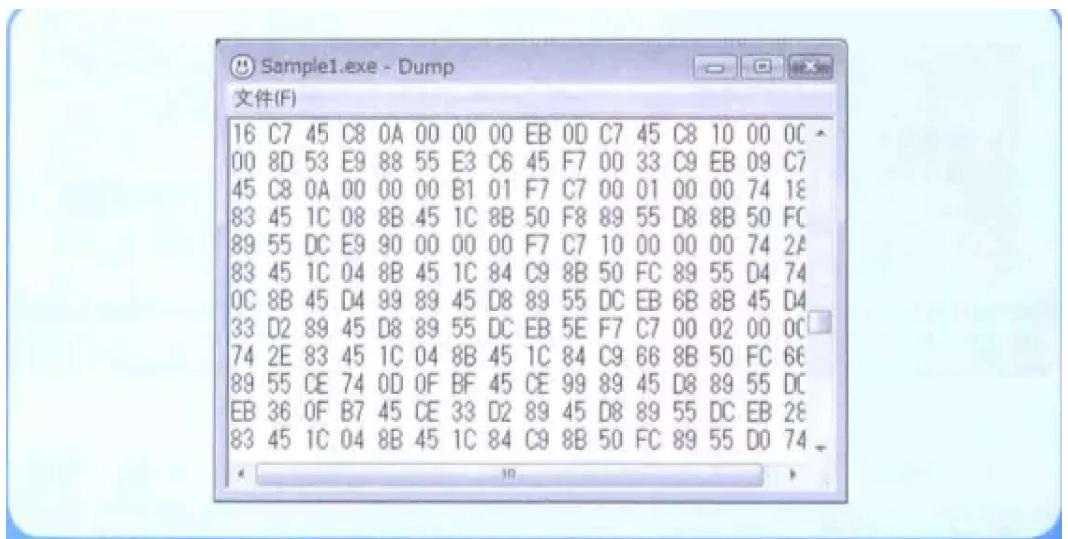
8.2 本地代码的内容

直接用记事本打开本地代码：



记事本打开本地代码

把本地代码Dump一下，每一个字节用2位16进制数（每个16进制数代表4位二进制数，2位16进制数恰好代表8位即1字节）来表示：



本地代码的真是面目是数值的罗列

8.3 编译器负责转换源代码

- 不同的语言有各自的编译器；
- 不同类的CPU有不同的机器语言，需要不同的编译器；

- 编译器也是应用程序，也需要运行环境；
- 存在交叉编译器，在某一环境下运行，可以生成另一环境下的本地代码。

```
bcc32 -W -c Sample1.c
```

通过命令行编译上面的栗子

8.4 仅靠编译是无法得到可执行文件的

- 编译生成的是.obj文件（目标文件），而不是.exe文件，无法直接运行；
- 如果源代码中引用了其它的函数（如上面例子中的sprintf()、MessageBox()），就需要把储存着这些函数的目标文件与此目标文件相结合；
- 完成此工作的是链接器，最后生成.exe文件

```
ilink32 -Tpe -c -x -aa c0w32.obj Sample1.obj, Sample1.exe,,  
import32.lib cw32.lib
```

链接上面的栗子

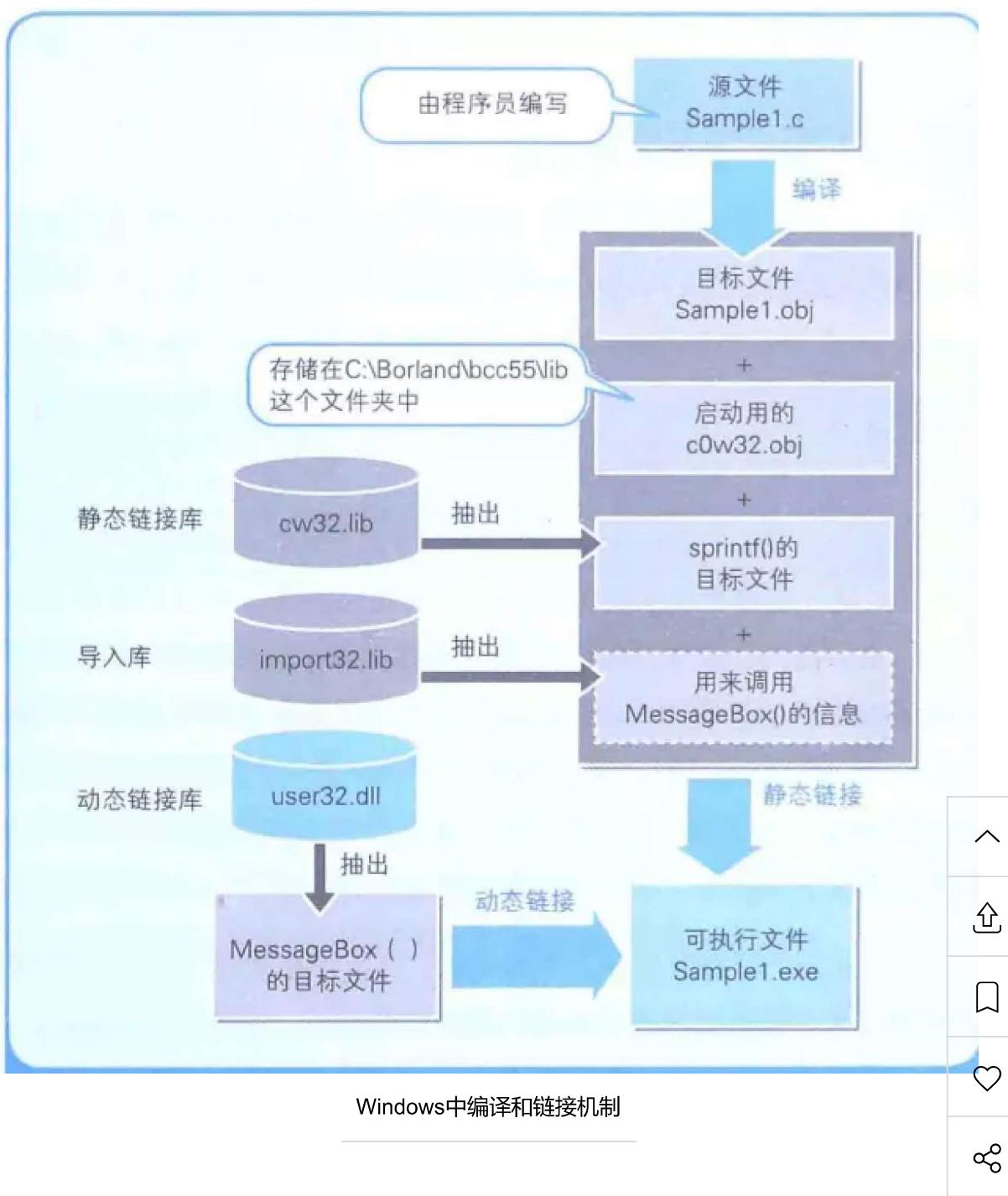
8.5 启动及库文件

- 在链接的命令中，c0w32.obj记述的是同所有程序起始位置相结合的处理内容，称为程序的启动，即使未调用其它目标文件的函数，也必须要进行链接，并和启动结合起来；
- 扩展名为.lib的文件称为库文件，是多个目标文件的集合。链接器指定库文件后，会把需要的函数从库文件中提取出来，例子中的sprintf()储存在cw32.lib中，MessageBox()实际上是储存在user32.dll中（会在后面说明原因）；
- 库文件可以简化链接过程，当需要很多个库函数时，只需要链接数个库文件就可以了；
- sprintf()等函数，不通过源代码而是通过库函数和编译器一起提供，称之为标准函数，标准函数以目标文件形式集合在库文件中，不会暴露源码，可以避免造成商业损失。

8.6 DLL文件及导入库

- Windows以函数的形式为应用提供了各种功能，称之为API (Application Programming Interface)，上面的栗子中MessageBox()就是Windows提供的一种API；
- Windows中的API并非储存在通常的库函数中，而是储存在DLL (动态链接库) 中， DLL是程序运行时动态结合的文件；
- 与DLL相反，存储目标文件的实体，并直接与EXE结合的称之为静态链接库，例如cw32.lib；
- 通过导入库文件，EXE在执行时会从DLL调出函数的信息就会写在EXE中。

用下图总结下：



8.7 可执行文件运行时的必要条件

EXE文件中函数和变量的内存地址是如何来表示的呢？

- 在EXE文件的开头给函数和变量分配了虚拟内存地址，程序运行时，虚拟内存地址会转换为实际内存地址；
- 链接器会在开头，追加转换内存地址所必需的信息，这个信息称为再配置信息；
- 在配置信息，就成了函数和变量的相对地址；
- 在源代码中，函数和变量是分散记数的，在链接后的EXE中，函数和变量就会变成连续排列的组，这样就可以用相对于起始地址的偏移量来表示函数和变量。

链接后EXE的构造

8.8 程序加载时会生成堆和栈

- 程序加载到内存后，会生成栈和堆；
- 栈存储局部变量，堆用来存储程序运行时的任意数据及对象的内存区域；
- 栈中对数据进行存储和舍弃的代码，由编译器自动生成，而堆的内存空间则需要程序员明确申请分配或者释放，在高级语言中，编译器会自动生成指定栈和堆大小的代码；



加载到内存中的程序由4部分组成

8.9 有点难度的Q&A



问题答案：

答案

第9章 操作系统和应用的关系

本章问题：



问题

本章重点

9.1 操作系统功能的历史

- 操作系统的前身是监控程序，作用是加载和运行程序；
- 后来人们发现很多程序都有共同的部分，例如用键盘输入、用显示屏输出等，于是把这些程序也加入了监控程序当中；
- 于是，更多的有用程序被加入了监控程序中，渐渐变身成为了操作系统.....

监控程序是操作系统的雏形



初期的操作系统 = 监控系统 + 输入输出程序

操作系统是多个程序的集合体

9.2 要意识到操作系统的存在

应用程序通过操作系统间接向硬件发送指令。

比如printf()、time()函数运行的结果，是面向操作系统而非硬件的，操作系统接到指令后，首先解释这些指令，然后会对时钟IC和显示器的IO进行控制。

应用程序通过操作系统间接控制硬件



9.3 系统调用和高级编程语言的移植性

- 操作系统的硬件控制功能，通常是通过一些小的函数集合体的形式提供的，这些函数及调用这些函数的行为统称为系统调用；
- C语言等高级语言一般不依赖于操作系统，因为希望不会因为操作系统的不同而重写大量的代码；所以高级语言使用独立的函数名，然后在编译时转换成相应的系统调用；
- 高级语言也可以直接进行系统调用，不过会影响可移植性；

高级编程语言的函数调用在编译后变成了系统调用

9.4 操作系统和高级编程语言使硬件抽象化

操作系统的系统调用，给程序的编写带来的巨大的方便。

9.5 Windows操作系统的特征

- 32位操作系统（书有点过时了）：在过去的16位操作系统中，处理32位的数据类型相当于处理两次16位数据类型，要花费更多的时间；有了32位系统，处理一次就够了，因此使用32位数据类型不会降低运行速度；
- 通过API函数集实现系统调用：API通过多个DLL文件来提供；
- 提供采用了GUI的用户界面：编写GUI较为困难，因为操作流程是由用户决定而非程序员决定，因此要考虑所有可能的情况；
- 通过WYSIWYG来打印输出；
- 提供多任务功能；
- 提供网络功能及数据库功能；
- 通过即插即用实现设备驱动的自动设定；



问题答案：

问题答案

第10章 通过汇编语言了解程序的实际构成

本章问题：



问题

本章重点：

当然是汇编。

10.1 汇编语言和本地代码是一一对应的

1. 使用汇编语言有助于理解本地代码。直接打开本地代码只能看到数值的罗列，通过添加助记符，如加法运算add，比较运算cmp等，可以更好地理解决本地代码。使用助记符的语言被称为汇编语言，通过查看汇编语言的源代码，可以更容易地理解决本地代码。
2. 汇编语言和本地代码是一一对应的，所以可以通过本地代码反汇编得到汇编代码。但是高级语言和本地代码不是一一对应的，所以反编译到高级语言比较困难，而且完全还原是不太可能的。

10.2 通过编译器输出汇编语言的源代码

原书作者通过编译命令把源代码输出为汇编代码，汇编文件的后缀为.asm(assemble)。

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)
{
    return a + b;
}

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

源代码



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

```

汇编



10.3 不会转换成本地代码的伪指令

汇编语言的源代码，是由转换成本地代码的指令（操作码）和针对汇编器的 伪指令 组成的。

伪指令 负责把程序的构造及汇编的方法指示给汇编器，但是 伪指令 本身是无法转换成本地代码的。

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
```

伪指令

其中由伪指令 `segment` 和 `ends` 围起来的部分，是程序中命令和数据的集合体，称之为段定义。`_TEXT`、`_DATA`、`_BSS` 是段定义的名称。

`group` 表示把 `_BSS` 和 `_DATA` 这两个段定义汇总名为 `DGROUP` 的组。

`_AddNum proc` 和 `_AddNum endp` 围起来的部分，是函数 `AddNum` 的范围，同理 `_MyFunc proc` 和 `_MyFunc endp` 围起来的部分表示函数 `MyFunc` 的范围。这两个函数都置于 `_TEXT` 中，表示属于 `_TEXT` 段定义。虽然源代码中的指令和数据比较混乱，但是通过段定义，编之后会转换成划分整齐的本地代码。

`end` 表示源代码的结束。



10.4 汇编语言的意思是“操作码”+“操作数”

操作码 是指令动作， 操作数 是指令对象。

常用操作码

主要寄存器

10.5 最常用的mov指令

mov 指令中有两个操作数，分别指定数据的存储地和来源。

如果操作数没有用 [] 围起来，就表示对其值进行操作，否则的话会把值解释为内存地址，然后对相应地址中的值进行操作。

两种情况



10.6 对栈进行push和pop

栈是存储临时数据的区域，数据的读取要符合先进后出原则。

栈模型

10.7 函数调用机制

函数调用需要依赖栈的作用。

下图为在MyFunc函数中调用AddNum函数的处理内容：

函数调用



- (1)、(2)、(7)、(8) 的处理适用于C语言中所有函数。
- (3) 和 (4) 表示传递给AddNum的函数通过push入栈。
- (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，AddNum处理完毕后，程序流程会必须回到 (6) 这一行。可以在调用AddNum之前把 (6) 指令的内存地址入栈，调用完毕后ret指令再把 (6) 指令的内存地址pop出来，从而使程序流程回到 (6)；
- (6) 是把栈指针向高位移动两位，实际上就是使123和456出栈；
- 源代码中有个变量c指向AddNum(123,456)的运算结果，但是c变量在之后并没有用到，所以编译器就会自动优化，没有生成与之相关的汇编代码。

10.8 函数内部的处理

下图为call AddNum后AddNum函数内部的处理过程。



函数内部的处理

- ebp在(1)、(5)中入栈、出栈，是为了把ebp的值还原到函数调用前的状态，因为ebp之前可能在其它地方被使用；
- 指令(2)中把栈指针寄存器esp的值赋给ebp，这是因为在mov指令中方括号[]中的参数不允许使用esp，所以要用ebp来代替；
- 使用栈中的数据是通过方括号中的ebp + 相应字节数来实现的，比如(3)中通过[ebp + 8]指定栈中的123，并用mov存储在eax中；
- 指令(4)中的add指令把123和456相加存储在eax中；
- **函数的参数通过栈来传递，返回值是通过寄存器来返回；**
- 指令(6)的ret运行后，函数返回目的地的内存地址会自动出栈，程序流程就会跳至函数调用的下一行；
- 可按照图10-4、10-5中a、b、c、d、e、f的顺序来看函数调用时栈的状态变化。



10.9 始终确保全局变量用的内存空间

简单地说，在Borland C++中，初始化和非初始化的全局变量分别被划到两个不同的段定义中，未被初始化的变量都会被设定为0进行初始化。

10.10 临时确保局部变量用的内存空间

临时变量储存在寄存器和栈中。

由于寄存器的访问速度较快，寄存器空闲时就使用寄存器来存储局部变量，否则就用栈。

函数调用完毕后，栈中局部变量的值就会被销毁（通过恢复栈指针的方式）。

10.11 循环处理的实现方法

循环源代码：

代码清单 10-8 执行循环处理的 C 语言源代码

```
// 定义 MySub 函数
void MySub()
{
    // 不做任何处理
}

// 定义 MyFunc 函数
Void MyFunc()
{
    int i;
    for (i = 0; i < 10; i++)
    {
        // 重复调用 MySub 函数 10 次
        MySub();
    }
}
```

循环源代码

代码清单 10-9 将代码清单 10-8 中的 for 语句转换成汇编语言的结果

@4	xor ebx, ebx ; 将 eax 寄存器清 0	
	call _MySub ; // 调用 MySub 函数	
	inc ebx ; // ebx 寄存器的值加 1	
	cmp ebx, 10 ; // 将 ebx 寄存器的值和 10 进行比较	
	jl short @4 ; // 如果小于 10 就跳转到 @4	

循环汇编代码



- 对ebx执行xor异或运算，使ebx清零，这比mov ebx 0 要快，ebx清零其实就等价于 i=0；
- ebx初始化后调用MySub，然后返回，进行第三行指令，把ebx加一；
- cmp是把ebx的值与10比较，结果存储在标志寄存器中
- jl 是jump on less than，如果前面的比较指令的值是“小”的话，就跳转至@4处的指令，从而实现了循环。

10.12 条件分支的实现方法

与循环类似。

条件分支C++源代码：

代码清单 10-11 进行条件分支的 C 语言源代码

```
// 定义 MySub1 函数
void MySub1()
{
    // 不做任何处理
}

// 定义 MySub2 函数
void MySub2()
{
    // 不做任何处理
}

// 定义 MySub3 函数
void MySub3()
{
    // 不做任何处理
}

// 定义 MyFunc 函数
void MyFunc()
{
    int a = 123;
    // 根据条件调用不同的函数
    if (a > 100)
    {
        MySub1();
    }
    else if (a < 50)
    {
        MySub2();
    }
    else
    {
        MySub3();
    }
}
```



条件分支汇编代码：

代码清单 10-12 将代码清单 10-11 的 MyFunc 函数转换成汇编语言后的结果

```

_MyFunc proc near
    push ebp;
    mov ebp,esp;

    mov eax,123      ; 把 123 存入 eax 寄存器中
    cmp eax,100      ; 把 eax 寄存器的值同 100 进行比较
    jle short @@     ; 比 100 小时, 跳转到 @@ 标签
    call _MySub1     ; 调用 MySub1 函数
    jmp short @11    ; 跳转到 @11 标签
@@:   cmp eax,50      ; 把 eax 寄存器的值同 50 进行比较
    jge short @10    ; 大于 50 时, 跳转到 @10 标签
    call _MySub2     ; 调用 MySub2 函数
    jmp short @11    ; 跳转到 @11 标签
@10:  call _MySub3     ; 调用 MySub3 函数
@11:  pop ebp
    ret
_MyFunc endp

```

10.13 了解程序运行方式的必要性

了解程序运行方式有助于我们更好的理解程序出错的原因。

下面的代码是两个函数更新同一个全局变量：

代码清单 10-13 两个函数更新同一个全局变量数值的 C 语言程序

```

// 定义全局变量
int counter = 100;

// 定义 MyFunc1 函数
void MyFunc1()
{
    counter *= 2;
}

// 定义 MyFunc2 函数
void MyFunc2 ()
{
    counter *=2;
}

```

实际的汇编代码：

代码清单 10-14 将全局变量的值翻倍这一部分转换成汇编语言源代码的结果

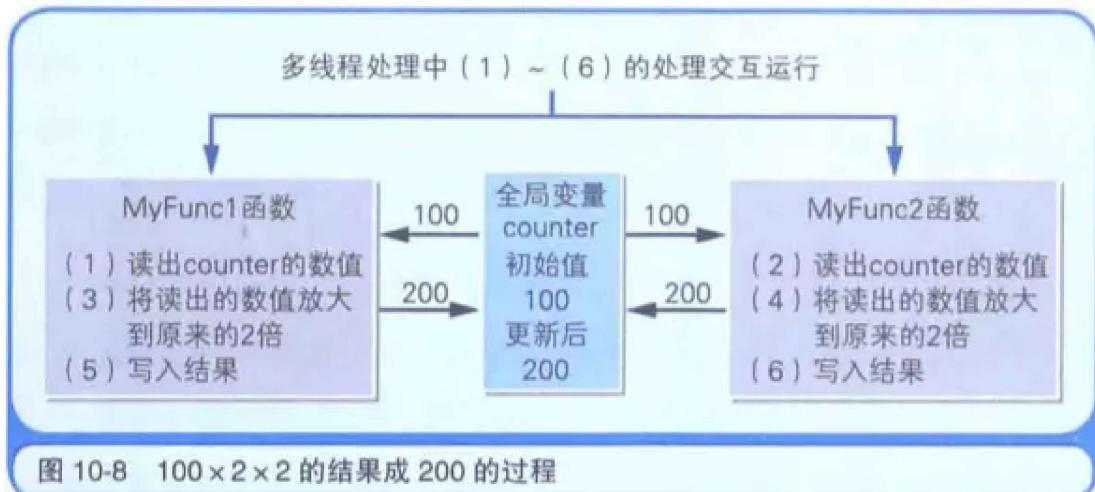
```

mov eax,dword ptr[_counter] ; 将 counter 的值读入 eax 寄存器
add eax, eax                ; 将 eax 寄存器的值扩大至原来的 2 倍
mov dword ptr[_counter],eax ; 将 eax 寄存器的数值存入 counter 中

```

可以看出，counter的值乘2是把counter的值读入累加寄存器后实现的。MyFunc1和MyFunc2都是把counter的值乘2，最后理应是4倍，但是MyFunc1运行时如果尚未来的及

把eax中两倍的数值写入到counter中去MyFunc2就读取了counter的值，最后运算的结果是counter的值只变为了原来的两倍。



因此为了避免这种错误，我们可以采用以函数或者C语言代码的行为单位来禁止线程切换的锁定方法。

问题答案：

1. 助记符
2. 汇编
3. 反汇编
4. .asm
5. 构成程序的命令和数据的集合组
6. 将程序流程跳转到其他地址时需要用到该指令

解析 ······

1. 汇编语言是通过利用助记符来记述程序的。
2. 使用汇编器这个工具来进行汇编。
3. 通过反汇编，得到人们可以理解的代码。
4. .asm 是 assembler(汇编器) 的略写。
5. 在高级编程语言的源代码中，即使指令和数据在编写时是分散的，编译后也会在段定义中集合汇总起来。大家看过汇编语言的源代码后，就会清楚了。
6. 在汇编语言中，通过跳转指令，可以实现循环和条件分支。



答案

第11章 硬件控制方法

本章提问：

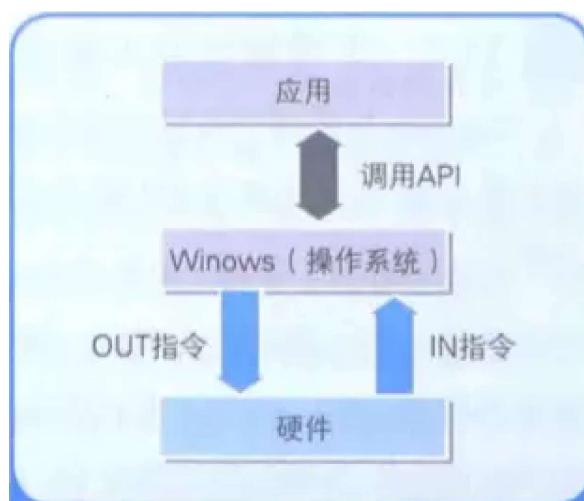


1. 在汇编语言中，是用什么指令来同外围设备进行输入输出操作的？
2. I/O 是什么的缩写？
3. 用来识别外围设备的编号称为什么？
4. IRQ 是什么的缩写？
5. DMA 是什么的缩写？
6. 用来识别具有 DMA 功能的外围设备的编号称为什么？

问题

11.1 应用和硬件无关？

Windows应用通过调用Windows操作系统的API（系统调用）来间接控制硬件。



应用通过操作系统间接控制硬件



下面是一个 ，调用Windows API 中的 TextOut 函数在窗口中显示字符串。

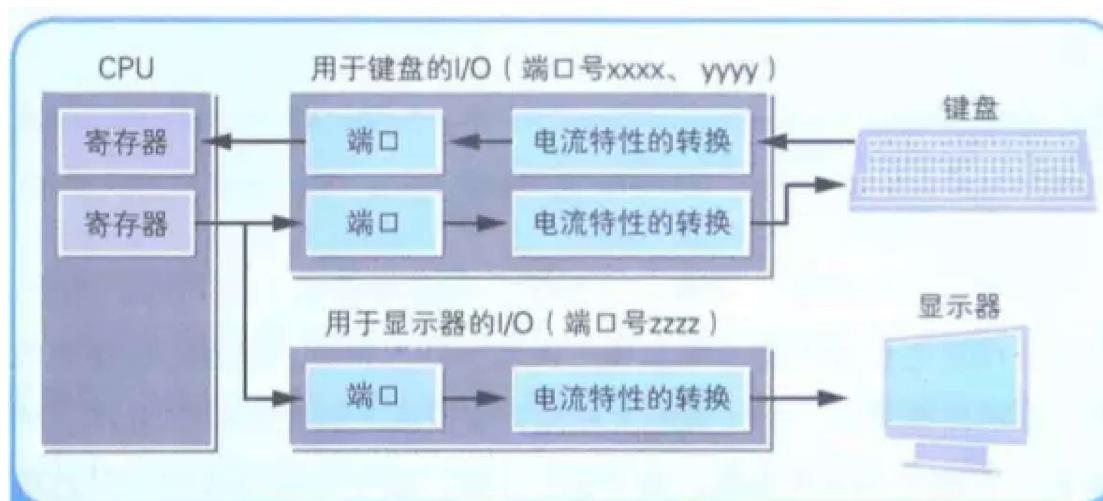
代码清单 11-1 TextOut 函数的语法 (C 语言)

```
BOOL TextOut(
    HDC hdc,           // 设备描述表的句柄
    int nXStart,        // 显示字符串的 x 坐标
    int nYStart,        // 显示字符串的 y 坐标
    LPCTSTR lpString,   // 指向字符串的指针
    int cbString        // 字符串的文字数
);
```

11.2 支持硬件输入输出的IN指令和OUT指令

IN 指令是把指定端口号的端口数据存储在CPU内的寄存器中，**OUT** 指令是把CPU寄存器中的数据输出到指定端口号的端口当中。

每个硬件都会有各自的I/O控制器，一个控制器可以控制多个端口，端口就是内存，储存着要交换的数据，端口用端口号来识别。



11.3 编写测试用的输入输出程序

这个例子是通过编写程序控制计算机内部的蜂鸣器发声。

小知识：C代码可以和汇编代码混写，但是汇编代码必须写在asm{}的大括号里。

在AT兼容机中，蜂鸣器的端口号是61H（末尾的H表示的是16进制数的意思），通过把向蜂鸣器端口发送的数据的后两位设为1或0，来控制蜂鸣器发声、关闭。

实现方法：

- 与0进行OR运算，不改变原二进制序列；与1进行AND运算，不改变原二进制序列。



2. 因此，可以让数据与03H(二进制为00000011)进行OR运算，这样数据前6位不变，后两位变为1，进而控制蜂鸣器发声；同样，与FCH(11111100)进行AND运算，前6位不变，后两位为0，进而控制蜂鸣器关闭。

代码如下：

代码清单 11-2 利用 IN/OUT 指令来控制蜂鸣器的程序示例

```
void main() {
    // 计数器
    int i;

    // 蜂鸣器发声

    _asm {
        IN EAX, 61H
        OR EAX, 03H
        OUT 61H, EAX
    } —————— (1)

    // 等待一段时间
    for (i = 0; i < 1000000; i++) ; —————— (2)

    // 蜂鸣器停止发声
    _asm {
        IN EAX, 61H
        AND EAX, 0FCH
        OUT 61H, EAX
    } —————— (3)
}
```

通过IN和OUT指令，在寄存器和61H端口中输入输出数据，控制蜂鸣器发声。

程序在低版本Windows中可以运行，高版本Windows禁止了应用直接控制硬件，这个程序会被禁止运行。

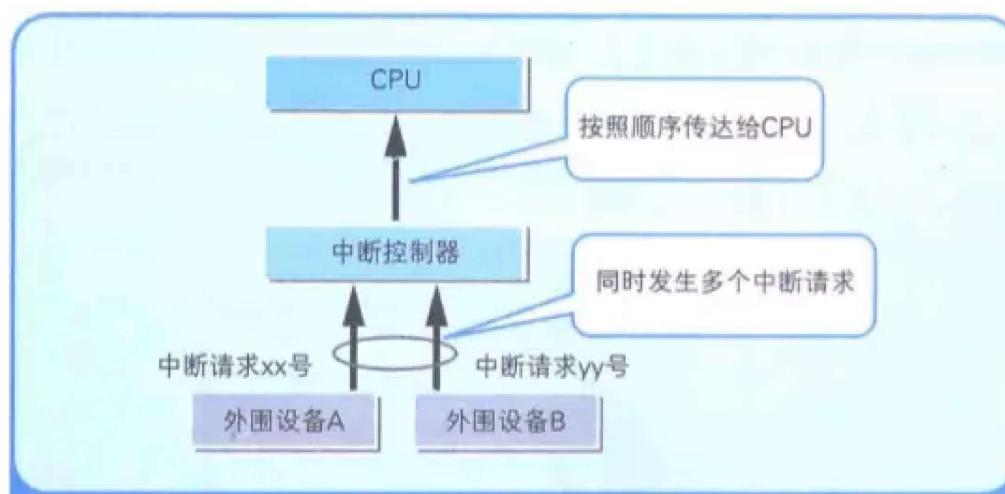
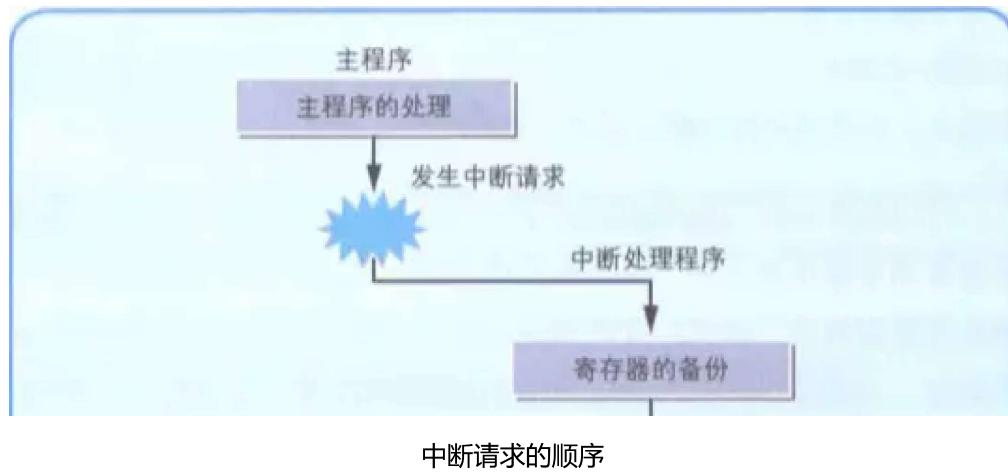
11.4 外围设备的中断请求

每个设备会有自己的中断编号。

收到中断请求后，CPU会把当前任务暂时挂起来处理中断请求。

计算机通过中断控制器来管理中断请求。





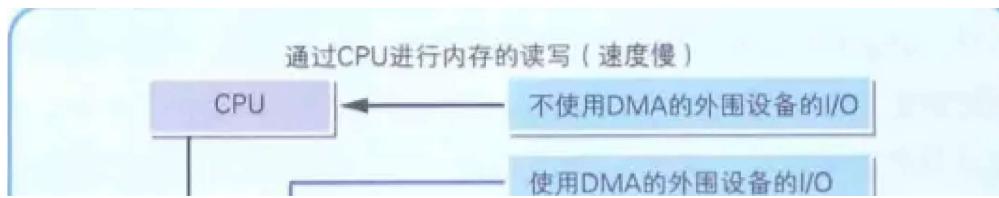
11.5 用中断来实现实时处理

如题。

11.6 DMA可实现短时间内传输大量数据

DMA, Direct Memory Access, 指不通过CPU的情况下，外围设备直接和主内存进行数据传送，这样会更快，DMA不是必选项，也有自己动编号。

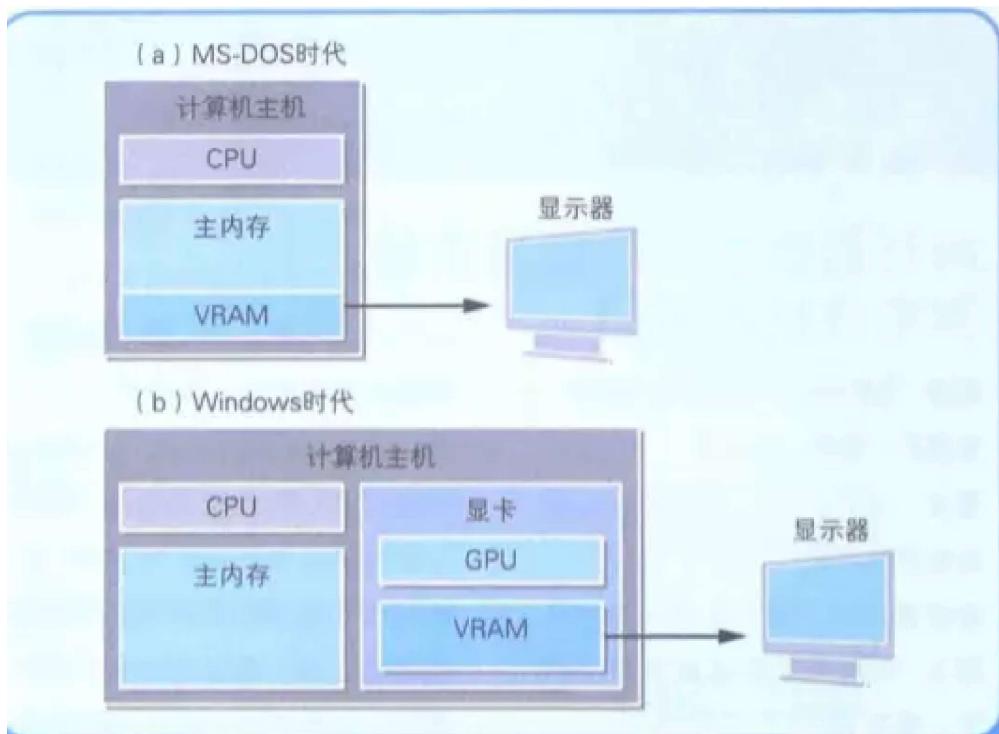




11.7 文字及图片的显示机制

显示器中显示的内容储存在VRAM (Video RAM) 中，在程序中，向VRAM写入数据，就会在显示器中显示出来，实现该功能的程序，由BIOS (Basic Input Output System) 提供，并借助中断运行。

现代计算机中，显卡等专用硬件一般都配置有与主内存相独立的VRAM和 GPU (Graphics Processing Unit 图形处理器)，过去的VRAM是主内存的一部分。



问题答案：



1. IN 指令和 OUT 指令
2. Input/Output
3. I/O 地址或 I/O 端口号
4. Interrupt Request
5. Direct Memory Access
6. DMA 通道

解析 ······

1. 在 x86 系列 CPU 用的汇编语言中，通过 IN 指令来实现 I/O 输入，OUT 指令来实现 I/O 输出。
2. 用来实现计算机主机和外围设备输入输出交互的 IC 称为 I/O 控制器或简称为 I/O。

答案

第12章 让计算机“思考”

略

完。

小礼物走一走，来简书关注我

赞赏支持



程序是怎样跑起来的 (/nb/36570283)

© 著作权归作者所有



 c747190cc2f5 (/u/c747190cc2f5)
写了 25958 字，被 1 人关注，获得了 3 个喜欢
(/u/c747190cc2f5)



一个菜鸟的挣扎...



喜欢**更多分享**

写下你的评论...

评论 [关闭评论](#)

智慧如你，不想发表一点想法咩~

