计算机工作原理

本章重点介绍计算机的工作原理，具体涉及存储程序计算机工作模型、基本的汇编语 言，以及 C 语言程序汇编出来的汇编代码如何在存储程序计算机工作模型上一步步地执 行。其中重点分析了函数调用堆栈相关汇编指令，如 call/ret 和 pushl/popl。

### 1.1 存储程序计算机工作模型

存储程序计算机的概念虽然简单，但在计算机发展史上具有革命性的意义，至今为 止仍是计算机发展史上非常有意义的发明。一台硬件有限的计算机或智能手机能安装各 种各样的软件，执行各种各样的程序，这在人们看起来都理所当然，其实背后是存储程 序计算机的功劳。

存储程序计算机的主要思想是将程序存放在计算机存储器中，然后按存储器中的存储 程序的首地址执行程序的第一条指令，以后就按照该程序中编写好的指令执行，直至程序 执行结束。

相信很多人特别是学习计算机专业的人都听说过图灵机和冯·诺依曼机。图灵机关注 计算的哲学定义，是一种虚拟的抽象机器，是对现代计算机的首次描述。只要提供合适的 程序，图灵机就可以做任何运算。基于图灵机建造的计算机都是在存储器中存储数据，程 序的逻辑都是嵌入在硬件中的。

与图灵机不同，冯·诺依曼机是一个实际的体系结构，我们称作冯·诺依曼体系结构， 它至今仍是几乎所有计算机平台的基础。我们都知道“庖丁解牛”这个成语，比喻经过反 复实践，掌握了事物的客观规律，做事得心应手，运用自如。冯·诺依曼体系结构就是各 种计算机体系结构需要遵从的一个“客观规律”，了解它对于理解计算机和操作系统非常重要。下面，我们就来看看什么是冯·诺依曼体系结构。

在 1944～1945 年期间，冯·诺依曼指出程序和数据在逻辑上是相同的，程序也可以存 储在存储器中。冯·诺依曼体系结构的要点包括：

1、冯·诺依曼体系结构如图 1-1 所示，其中运算器、存储器、控制器、输入设备和 输出设备 5 大基本类型部件组成了计算机硬件；

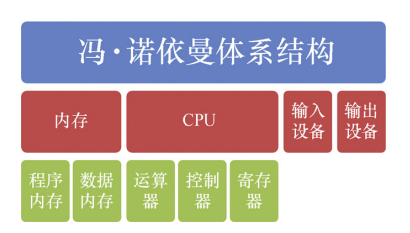


图1-1 冯·诺依曼体系结构

2、计算机内部采用二进制来表示指令和数据；

3、将编写好的程序和数据先存入存储器中，然后启动计算机工作，这就是存储程序 的基本含义。

计算机硬件的基础是 CPU，它与内存和输入/输出（I/O）设备进行交互，从输入设备 接收数据，向输出设备发送数据。CPU 由运算器（算术逻辑单元 ALU）、控制器和一些寄 存器组成。有一个非常重要的寄存器称为程序计数器（Program Counter，PC），在 IA32 （x86-32）中是 EIP，指示将要执行的下一条指令在存储器中的地址。C/C++程序员可以将 EIP 看作一个指针，因为它总是指向某一条指令的地址。CPU 就是从 EIP 指向的那个地址 取过来一条指令执行，执行完后 EIP 会自动加一，执行下一条指令，然后再取下一条指令 执行，CPU 像“贪吃蛇”一样总是在内存里“吃”指令。

CPU、内存和 I/O 设备通过总线连接。内存中存放指令和数据。

“计算机内部采用二进制来表示指令和数据”表明，指令和数据的功能和处理是不同 的，但都可以用二进制的方式存储在内存中。

上述第 3 个要点指出了冯·诺依曼体系结构的核心是存储程序计算机。

我们用程序员的思维来对存储程序计算机进行抽象，如图 1-2 所示。

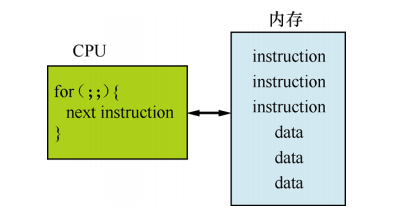


图1-2 存储程序计算机工作原理示意图

我们可以把 CPU 抽象成一个 for 循环，因为它总是在执行 next instruction（下一条指 令），然后从内存里取下一条指令来执行。从这个角度来看，内存保存指令和数据，CPU 负责解释和执行这些指令，它们通过总线连接起来。这里揭示了计算机可以自动化执行程 序的原理。

这里存在一个问题，CPU 能识别什么样的指令，我们这里需要有一个定义。学过编程的 读者基本都知道 API（Application Program Interface），也就是应用程序编程接口。而对于程 序员来讲，还有一个称为 ABI（Application Binary Interface）的接口，它主要是一些指令的 编码。在指令编码方面，我们不会涉及那么具体的细节，而只会涉及和汇编相关的内容。至 于这些指令是如何编码成二进制机器指令的，我们不必关心，有兴趣的读者可以查找指令编 码的相关资料。此外，这些指令会涉及一些寄存器，这些寄存器有些约定，我们约定什么样 的指令该用什么寄存器。同时，我们也需要了解寄存器的布局。还有，大多数指令可以直接 访问内存，对于 x86-32 计算机指令集来讲，这也是一个重要的概念。对于 x86-32 计算机， 有一个 EIP 寄存器指向内存的某一条指令，EIP 是自动加一的（不是一个字节，也不是 32 位， 而是加一条指令），虽然 x86-32 中每条指令占的存储空间不一样，但是它能智能地自动加到 下一条指令，它还可以被其他指令修改，如 call、ret、jmp 等，这些指令对应 C 语言中的函 数调用、return 和 if else 语句。

现在绝大多数具有计算功能的设备，小到智能手机，大到超级计算机，基本的核心部 分可以用冯·诺依曼体系结构（存储程序计算机）来描述。因此，存储程序计算机是一个 非常基本的概念，是我们理解计算机系统工作原理的基础。

## 1.2 x86-32 汇编基础

Intel 处理器系列也称为 x86，经过不断的发展，体系结构经历了 16 位（8086，1978）、 32 位（i386，1985）和 64 位（Pentium 4E，2004）几个关键阶段。32 位的体系结构称为 IA32 （Intel Architecture 32bit），64 位体系结构称为 x86-64，但为了明确区分两者，本书中把 32 位体系结构称作 x86-32。本书与 Linux 内核采用的汇编格式保持一致，采用 AT&T 汇编格式。

### 1.2.1 x86-32 CPU 的寄存器

为了便于读者理解，下面先来介绍 16 位的 8086 CPU 的寄存器。8086 CPU 中总共有 14 个 16 位的寄存器：AX、BX、CX、DX、SP、BP、SI、DI、IP、FLAG、CS、DS、SS 和 ES。这 14 个寄存器分为通用寄存器、控制寄存器和段寄存器 3 种类型。

通用寄存器又分为数据寄存器、指针寄存器和变址寄存器。

AX、BX、CX 和 DX 统称为数据寄存器。

1、AX（Accumulator）：累加寄存器，也称为累加器。

2、 BX（Base）：基地址寄存器。

3、 CX（Count）：计数器寄存器。

4、 DX（Data）：数据寄存器。

SP 和 BP 统称为指针寄存器。

1、 SP（Stack Pointer）：堆栈指针寄存器。

2、 BP（Base Pointer）：基指针寄存器。

SI 和 DI 统称为变址寄存器。

1、 SI（Source Index）：源变址寄存器。

2、 DI（Destination Index）：目的变址寄存器。

控制寄存器主要分为指令指针寄存器和标志寄存器。

1、 IP（Instruction Pointer）：指令指针寄存器。

2、 FLAG：标志寄存器。

段寄存器主要有代码段寄存器、数据段寄存器、堆栈段寄存器和附加段寄存器。

1、 CS（Code Segment）：代码段寄存器。

2、 DS（Data Segment）：数据段寄存器。

3、 SS（Stack Segment）：堆栈段寄存器。

4、 ES（Extra Segment）：附加段寄存器。

以上数据寄存器 AX、BX、CX 和 DX 都可以当作两个单独的 8 位寄存器来使用，如 图 1-3 所示，以 AX 寄存器为例。

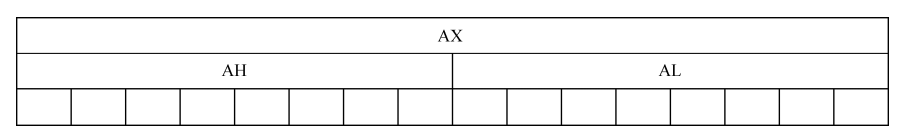


图1-3 AX 寄存器示意图

1、AX 寄存器可以分为两个独立的 8 位的 AH 和 AL 寄存器。

2、BX 寄存器可以分为两个独立的 8 位的 BH 和 BL 寄存器。

3、CX 寄存器可以分为两个独立的 8 位的 CH 和 CL 寄存器。

4、DX 寄存器可以分为两个独立的 8 位的 DH 和 DL 寄存器。

除了上面 4 个数据寄存器以外，其他寄存器均不可以分为两个独立的 8 位寄存器。注 意，每个分开的寄存器都有自己的名称，可以独立存取。程序员可以利用数据寄存器的这 种“可分可合”的特性，灵活地处理字/字节的信息。

了解了 16 位的 8086 CPU 的寄存器之后，我们再来看 32 位的寄存器。IA32 所含有的 寄存器包括：

 4、个数据寄存器（EAX、EBX、ECX 和 EDX）。

 2、个变址和指针寄存器（ESI 和 EDI）。

 2、个指针寄存器（ESP 和 EBP）。

 6、个段寄存器（ES、CS、SS、DS、FS 和 GS）。

 1、个指令指针寄存器（EIP）。

 1、个标志寄存器（EFlags）。

32 位寄存器只是把对应的 16 位寄存器扩展到了 32 位，如图 1-4 所示为 EAX 寄存器示 意图，它增加了一个 E。所有开头为 E 的寄存器，一般是 32 位的。 EAX 累加寄存器、EBX 基址寄存器、ECX 计数寄存器和 EDX 数据寄存器都是通用寄 存器，程序员在写汇编码时可以自己定义如何使用它们。EBP 是堆栈基址指针，比较重要；ESI、EDI 是变址寄存器；ESP 也比较重要，它是堆栈栈顶寄存器。这里可能会涉及堆栈的概 念，学过数据结构课程的读者应该知道堆栈的概念，本书后面会具体讲到 push 指令压栈和 pop 指令出栈，它是向一个堆栈里面压一个数据和从堆栈里面弹出一个数据。这些都是 32 位 的通用寄存器。

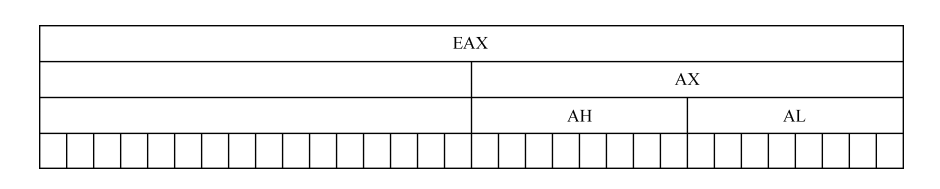


图1-4 EAX寄存器示意图

值得注意的是在 16 位 CPU 中，AX、BX、CX 和 DX 不能作为基址和变址寄存器来存放 存储单元的地址，但在 32 位 CPU 中，32 位寄存器 EAX、EBX、ECX 和 EDX 不仅可以传送 数据、暂存数据保存算术逻辑运算结果，还可以作为指针寄存器，因此这些 32 位寄存器更 具通用性。

除了通用寄存器外，还有一些段寄存器。虽然段寄存器在本书中用得比较少，但还是 要了解一下。除了 CS、DS、ES 和 SS 外，还有其他附加段寄存器 FS 和 GS。常用的是 CS 寄存器和 SS 寄存器。我们的指令都存储在代码段，在定位一个指令时，使用 CS:EIP 来准 确指明它的地址。也就是说，首先需要知道代码在哪一个代码段里，然后需要知道指令在 代码段内的相对偏移地址 EIP，一般用 CS:EIP 准确地标明一个指令的内存地址。还有堆栈 段，每一个进程都有自己的堆栈段（在 Linux 系统里，每个进程都有一个内核态堆栈和一 个用户态堆栈）。 标志寄存器的功能细节比较复杂烦琐，本书就不仔细介绍了，读者知道 标志寄存器可以保存当前的一些状态就可以了。

现在主流的计算机大多都是采用 64 位的 CPU，那么我们也需要简单了解一下 x86-64 的寄存器。实际上，64 位和 32 位的寄存器差别也不大，它只是从 32 位扩展到了 64 位。 前面带个“R”的都是指 64 位寄存器，如 RAX、RBX、RCX、RDX、RBP、RSI、RSP， 还有 Flags 改为了 RFLAGS，EIP 改为了 RIP。另外，还增加了更多的通用寄存器，如 R8、 R9 等，这些增加的通用寄存器和其他通用寄存器只是名称不一样，在使用中都是遵循调用 者使用规则，简单说就是随便用。

### 1.2.2 数据格式

在 Intel 的术语规范中，字（Word）表示 16 位数据类型；在 IA32 中，32 位数称为双 字（Double Words）；在 x86-64 中，64 位数称为四字（Quad Words）。图 1-5 所示为 C 语言中基本类型的 IA32 表示，其中列出的汇编代码后缀在汇编代码中会经常看到。

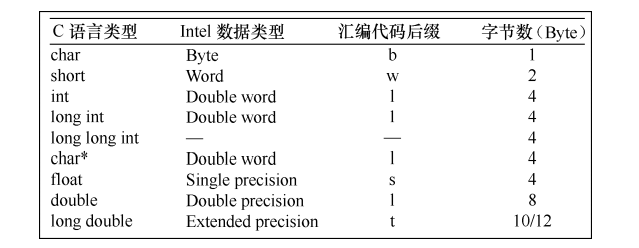


图1-5 C语言中基本类型的IA32表示

### 1.2.3 寻址方式和常用汇编指令

汇编指令包含操作码和操作数，其中操作数分为以下 3 种：

1、立即数即常数，如$8，用$开头后面跟一个数值；

 2、寄存器数，表示某个寄存器中保存的值，如%eax；而对字节操作而言，是8个单字节寄存器中的一个，如%al（EAX 寄存器中的低 8 位）；

 3、存储器引用，根据计算出的有效地址来访问存储器的某个位置。

还有一些常见的汇编指令，我们来看它们是如何工作的。最常见的汇编指令是 mov 指 令，movl 中的 l 是指 32 位，movb 中的 b 是指 8 位，movw 中的 w 是指 16 位，movq 中的 q 是指 64 位。我们以 32 位为主进行介绍。

首先介绍寄存器寻址（Register mode）。所谓寄存器寻址就是操作的是寄存器，不和内 存打交道，如%eax，其中%开头后面跟一个寄存器名称。

movl %eax，%edx

上述代码把寄存器%eax 的内容放到%edx 中。如果把寄存器名当作 C 语言代码中的变 量名，它就相当于：

edx = eax;

立即寻址（immediate）是用一个$开头后面跟一个数值。例如：

movl $0x123， %edx

就是把 0x123 这个十六进制的数值直接放到 EDX 寄存器中。如果把寄存器名当作 C 语言代码中的变量名，它就相当于：

edx = 0x123;

立即寻址也和内存没有关系。

直接寻址（direct）是直接用一个数值，开头没有$符号。开头有$符号的数值表示这是 一个立即数；没有$符号表示这是一个地址。例如：

movl 0x123， %edx

就是把十六进制的 0x123 内存地址所指向的那块内存里存储的数据放到 EDX 寄存器 里，这相当于 C 语言代码：

edx = \*(int\*)0x123;

把 0x123 这个数值强制转化为一个 32 位的 int 型变量的指针，再用一个\*取它指向的 值，然后放到 EDX 寄存器中，这就称为直接寻址。换句话说，就是用内存地址直接访问内 存中的数据。

间接寻址（indirect）就是寄存器加个小括号。举例说明，%ebx 这个寄存器中存的值 是一个内存地址，加个小括号表示这个内存地址所存储的数据，我们把它放到 EDX 寄存 器中：

move (%ebx), %edx

就相当于：

edx = \*(int\*)ebx;

把这个 EBX 寄存器中存储的数值强制转化为一个 32 位的 int 型变量的指针，再用一个 \*取它指向的值，然后放到 EDX 寄存器中，这称为间接寻址。

变址寻址（displaced）比间接寻址稍微复杂一点。例如：

movl 4(%ebx), %edx

读者会发现代码中“（%ebx）”前面出现了一个 4，也就是在间接寻址的基础上，在原 地址上加上一个立即数 4，相当于：

edx = \*(int\*)(ebx+4)

把这个 EBX 寄存器存储的数值加 4，然后强制转化为一个 32 位的 int 类型的指针，再 用一个\*取它指向的值，然后放到 EDX 寄存器中，这称为变址寻址。

如上所述的 CPU 对寄存器和内存的操作方法，都是比较基础的知识，需要牢固掌握。

x86-32 中的大多数指令都能直接访问内存，但还有一些指令能直接对内存操作，如 push/pop。它们根据 ESP 寄存器指向的内存位置进行压栈和出栈操作，注意这是指令执行 过程中默认使用了特定的寄存器。

还需要特别说明的是，本书中使用的是 AT&T 汇编格式，这也是 Linux 内核使用的汇 编格式，与 Intel 汇编格式略有不同。我们在搜索资料时可能会遇到 Intel 汇编代码，一般 来说，全是大写字母的一般是 Intel 汇编，全是小写字母的一般是 AT&T 汇编。本书中的代 码用到的寄存器名称都遵守 AT&T 汇编格式采用全小写的方式，而正文中需要使用寄存器 名称一般使用大写，因为它们是首字母缩写。

还有几个重要的指令：pushl/popl 和 call/ret。pushl 表示 32 位的 push，如：

pushl %eax

就是把 EAX 寄存器的值压到堆栈栈顶。它实际上做了这样两个动作，其中第一个动 作为：

subl $4， %esp

把堆栈的栈顶 ESP 寄存器的值减 4。因为堆栈是向下增长的，所以用减指令 subl，也 就是在栈顶预留出一个存储单元。第二个动作为：

movl %eax， (%esp)

把 ESP 寄存器加一个小括号（间接寻址），就是把 EAX 寄存器的值放到 ESP 寄存器所 指向的地方，这时 ESP 寄存器已经指向预留出的存储单元了。

接下来介绍 popl 指令，如：

popl %eax

就是从堆栈的栈顶取一个存储单元（32 位数值），从堆栈栈顶的位置放到 EAX 寄存器 里，这称为出栈。出栈同样对应两个操作：

movl (%esp), %eax

addl $4, %esp

第一步是把栈顶的数值放到 EAX 寄存器里，然后用指令 addl 把栈顶加 4，相当于栈向 上回退了一个存储单元的位置，也就是栈在收缩。每次执行指令 pushl 栈都在增长，执行 指令 popl 栈都在收缩。

call 指令是函数调用，调用一个地址。例如：

call 0x12345

上述代码实际上做了两个动作，如下两条伪指令，注意，这两个动作并不存在实际对 应的指令，我们用“（\*）”来特别标记一下，这两个动作是由硬件一次性完成的。出于安全 方面的原因，EIP 寄存器不能被直接使用和修改。

pushl %eip (\*)

movl $0x12345, %eip (\*)

上述伪指令先是把当前的 EIP 寄存器压栈，把 0x12345 这个立即数放到 EIP 寄存器里， 该寄存器是用来告诉 CPU 下一条指令的存储地址的。把当前的 EIP 寄存器的值压栈就是把 下一条指令的地址保存起来，然后给 EIP 寄存器又赋了一个新值 0x12345，也就是 CPU 执行 的下一条指令就是从 0x12345 位置取得的。

再看与 call 指令对应的指令 ret，ret 指令是函数返回，例如：

ret

上述代码实际上做了一个动作，如下一条伪指令，注意，这个动作并不存在实际对应 的指令，我们用“（\*）”来特别标记一下，这个动作是由硬件一次性完成的。出于安全方面 的原因，EIP 寄存器不能被直接使用和修改。

popl %eip(\*)

也就是把当前堆栈栈顶的一个存储单元（一般是由 call 指令压栈的内容）放到 EIP 寄 存器里。

上述 pushl/popl 和 call/ret 汇编指令对应执行的动作汇总如图 1-6 所示。

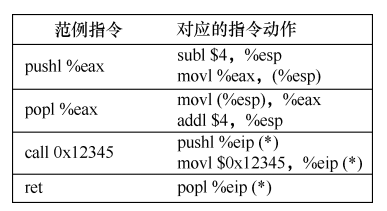


图1-6 pushl/popl和call/ret汇编指令

总结一下，call 指令对应了 C 语言里我们调用一个函数，也就是 call 一个函数的 起始地址。ret 指令是把调用函数时压栈的 EIP 寄存器的值（即 call 指令的下一条指令 的地址）还原到 EIP 寄存器里，ret 指令之后的下一条指令也就回到函数调用位置的下 一条指令。换句话说就是函数调用结束了，继续执行函数调用之后的下一条指令，这和 C 语言中的函数调用过程是严格对应的。但是需要注意的是，带个“（\*）”的指令 表示这些指令都是不能被程序员直接使用的，是伪指令。因为 EIP 寄存器不能被程序 员直接修改，只能通过专用指令（如 call、ret 和 jmp 等）间接修改。若程序员可以直接 修改 EIP 寄存器，那么会有严重的安全隐患。读者可以思考一下为什么？我们就不展开 讨论了。

### 1.2.4 汇编代码范例解析

我们已经对指令和寄存器有了大致的了解，下面做一个练习来验证我们的理解。在堆 栈为空栈的情况下，执行如下汇编代码片段之后，堆栈和寄存器都发生了哪些变化？

1 push $8

2 movl %esp, %ebp

3 subl $4, %esp

4 movl $8, (%esp)

我们分析这段汇编代码每一步都做了什么动作。首先在堆栈为空栈的情况下，EBP 和 ESP 寄存器都指向栈底。

第 1 行语句是将立即数 8 压栈（即先把 ESP 寄存器的值减 4，然后把立即数 8 放入当 前堆栈栈顶位置）。

第 2 行语句是把 ESP 寄存器的值放到 EBP 寄存器里，就是把 ESP 寄存器存储的内容放 到 EBP 寄存器中，把 EBP 寄存器也指向当前 ESP 寄存器所指向的位置。换句话说，在堆栈 中又新建了一个逻辑上的空栈，这一点理解起来并不容易，读者暂时理解不了也没有关系。 本书后面会将 C 语言程序汇编成汇编代码来分析函数调用是如何实现的，其中会涉及函数调 用堆栈框架。

第 3 行语句中的指令是 subl，是把 ESP 寄存器存储的数值减 4，也就是说，栈顶指针 ESP 寄存器向下移了一个存储单元（4 个字节）。

最后一行语句是把立即数 8 放到 ESP 寄存器所指向的内存地址，也就是把立即数 8 通 过间接寻址放到堆栈栈顶。

本例是关于栈和寄存器的一些操作的，我们可以对照上述文字说明一步一步跟踪堆栈 和寄存器的变化过程，以便更加准确地理解指令的作用。

再来看一段汇编代码，同样在堆栈为空栈的情况下，执行如下汇编代码片段之后，堆 栈和寄存器都发生了哪些变化？

1 pushl $8

2 movl %esp, %ebp

3 pushl $8

同样我们也分析一下这段汇编代码每一步都做了什么动作。首先在堆栈为空栈的情况 下 EBP 和 ESP 寄存器都指向栈底。

第 1 行语句是将立即数 8 压栈，即堆栈多了一个存储单元并存了一个立即数 8，同时 也改变了 ESP 寄存器。

第 2 行语句把 ESP 寄存器的值放到 EBP 寄存器里，堆栈空间没有变化，但 EBP 寄存 器发生了变化。

第 3 行语句将立即数 8 压栈，即堆栈多了一个存储单元并存了一个立即数 8。

读者会发现，这个例子和上一个例子的实际效果是完全一样的。

小试牛刀之后，再看下面这段更加复杂一点的汇编代码：

1 pushl $8

2 movl %esp, %ebp

3 pushl %esp

4 pushl $8

5 addl $4, %esp

6 popl %esp

这段汇编代码同样首先在堆栈为空栈的情况下 EBP 和 ESP 寄存器都指向栈底。

第 1 行语句是将立即数 8 压栈，即堆栈多了一个存储单元并保存立即数 8，同时也改 变了 ESP 寄存器。

第 2 行语句是把 ESP 寄存器的值放到 EBP 寄存器里，堆栈空间没有变化，但 EBP 寄 存器发生了变化。

第 3 行语句是把 ESP 寄存器的内容压栈到堆栈栈顶的存储单元里。需要注意的是，pushl 指令本身会改变 ESP 寄存器。“pushl %esp”语句相当于如下两条指令：

subl $4, %esp

movl %esp, (%esp)

显然，在保存 ESP 寄存器的值到堆栈中之前改变了 ESP 寄存器，保存到栈顶的数据应 该是当前 ESP 寄存器的值减 4。ESP 寄存器的值发生了变化，同时栈空间多了一个存储单 元保存变化后的 ESP 寄存器的值。

第 4 行语句是将立即数 8 压栈，即堆栈多了一个存储单元保存立即数 8，同时也改变了 ESP 寄存器。

第 5 行语句是把 ESP 寄存器的值加 4，这相当于堆栈空间收缩了一个存储单元。

最后一条语句相当于如下两条指令：

movl (%esp), %esp

addl $4, %esp

也就是把当前栈顶的数据放到 ESP 寄存器中，然后又将 ESP 寄存器加 4。这一段代码 比较复杂，因为 ESP 寄存器既作为操作数，又被 pushl/popl 指令在执行过程中使用和修改。 读者需要仔细分析和思考这段汇编代码以理解整个执行过程，本书后续内容会结合 C 代码 的函数调用和函数返回，来进一步理解这段汇编代码中涉及的建立一个函数调用堆栈和拆 除一个函数调用堆栈。