第3章 程序是如何执行的

本章讲授程序是如何在计算机里执行的，极为重要。一个计算系统至少包含了CPU和主存（Main Memory，或称为内存）。CPU是做运算的，主存是储存程序和变量（Variables）的。本章首先详细解释一行行的程序被CPU读后，如何控制CPU的运算和指示CPU去读写在主存中的变量；然后解释函数调用是如何执行的，其中涉及一些重要概念，那就是返回地址、局部变量、全局变量和栈的管理；接着介绍几种最常用的程序语言C、C++、Java和它们特性的比较；最后在讲述对计算机程序的领悟时，我们用非常有趣的“猜数字”例子讲述什么是人工智能（其实智能是程序计算出来的）。本章的这些知识对程序编写、编译器、操作系统、信息安全中蠕虫病毒的理解至关重要。

本书作者提供一个精心研发的汇编语言模拟器供教学使用，叫作SEAL。学生可利用此工具来设计和执行汇编语言程序，与本章内容密切配合，更充分地理解本章内容。此工具可以从前言中列出的清华大学出版社官方网站下载。

3.1 引例

程序员编写的程序（如Python、C、C++等）并不是计算机硬件可以直接识别的形式，计算机只能识别二进制的机器语言。本节就来探索一条程序语句在计算机中的执行过程。

程序的执行会牵扯到CPU和主存。如图3-1所示，计算机中有两个核心部件，分别是CPU和主存。CPU是做运算的，主存存储程序和相关的变量，每一条程序语句和每一个变量在内存中都有相应的内存地址。

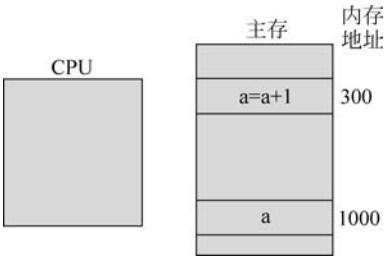


图3-1 计算机执行a=a+1语句

现在以下面一个简单的程序为例。在这个程序中只有一条语句a=a+1。大家看到这种带有“=”等号的语句时，要将“=”号左边和右边分开来分析。a=a+1这句的意思是：将等号右边的a+1计算出，然后将值赋给等号左边的变量a。等号右边的a是指变量a所存的值，而等号左边的a是指变量的位置。

接下来分析它是如何执行的。

第一，CPU先要读程序，从地址300处读取指令到CPU中，经过CPU的分析，CPU知

道接下来将要做的动作(也就是接下来的第二步);第二,CPU 会从地址 1000 处读变量 a 的值;第三,CPU 把这个值加 1;第四,CPU 将加 1 后的结果存回到地址 1000 处的 a。

3.2 a=a+1 的执行过程

其实,a=a+1 不是只有一个指令,它包含了数个基本指令。在本节,通过循序渐进的讲解,大家就会更清楚 a=a+1 的执行过程。

3.2.1 分解 a=a+1 的执行步骤

a=a+1 的执行可以分为三步。首先是 CPU 从主存中读取 a,接着 CPU 对 a 执行加 1 操作,最后 CPU 将运算后的结果存回主存。

如图 3-2 所示,主存中就会存储三条指令,依次是“读取 a 到 R”“R 加 1”“将 R 存回 a”。CPU 中有通用寄存器(Register)R 来存储变量 a。

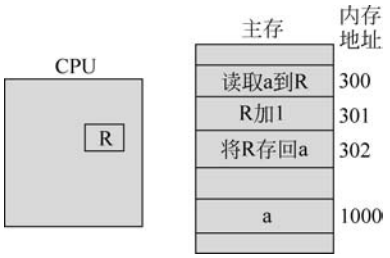


图 3-2 分解 a=a+1 执行步骤

CPU 读取变量 a 后,先存到寄存器 R 中。寄存器是 CPU 内的存储单元,是有限存储容量的高速存储部件。每一种类的 CPU 的寄存器的个数和使用会有少许的不同,但是每一个 CPU 都会有通用寄存器来给程序使用,编号 R1～R32 代表有 32 个通用寄存器。我们在运算 a=a+1 时,首先要把变量 a 读取到某一个寄存器 R,然后 CPU 再对寄存器 R 中的值进行运算。运算完成之后,CPU 才会将值存回主存。现在很多 CPU 不能直接对内存做运算,必须要先读到寄存器里,然后在寄存器上做运算,运算完后,再把结果存回内存里。

第一步,CPU 从地址 300 处读取第一条语句,CPU 执行“读取 a 到 R”语句,就会从地址 1000 处读取变量 a 的值到寄存器 R 中;

第二步,CPU 从地址 301 处读取第二条语句,执行“R 加 1”语句,CPU 会对 R 执行加 1 的操作;

第三步,CPU 再从地址 302 处读取第三条语句,执行“将 R 存回 a”语句,就把寄存器 R 中变量 a 的值存回主存中地址 1000 处。

3.2.2 CPU 中的核心部件

执行 a=a+1 时,讲到 CPU 需要从主存中相应地址处读取语句。这一节会解释以下问题:CPU 如何知道语句的地址?从主存中读取的语句存放在哪里?CPU 是怎样完成加法运算的?

如图 3-3 所示，CPU 中有寄存器 R、PC、IR、ALU 这些部件。现在我们来细看 CPU 中的这几个核心部件。

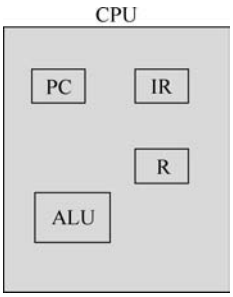


图 3-3　CPU 的核心部件

语句地址的存储——程序计数器(Program Counter，PC)

程序计数器 PC 是一个“特殊”寄存器部件。在计算机执行程序时，PC 始终指向主存中的某条指令语句(即该条语句在主存的地址)。CPU 就是读取 PC 所指向的那条指令来执行。

在顺序执行程序语句时，PC 通过顺序加 1(对 32 位 CPU，一个指令要占用 4 字节，所以其实是加 4；对 64 位 CPU，是加 8)自动指向下一条将要执行的程序语句。对于一些控制结构语句，如 if、for、while 等控制结构，程序的执行将会分叉，这时 PC 的值就不只是顺序加 1 来指向下一条程序语句。在 3.3 节会讲到更多细节。

CPU 中存储程序语句——指令寄存器(Instruction Register，IR)

指令寄存器 IR 也是个特殊寄存器，它被用来存放从主存中读取的程序指令。CPU 从主存中读取程序指令到 IR 之后，由特定的部件来解读这条程序指令，并执行相应的操作。

执行运算——算术逻辑单元(Arithmetic Logic Unit，ALU)

ALU 是处理器中进行真实运算的部件。执行程序指令时，CPU 把寄存器中的值输入 ALU 中，ALU 做完运算后把结果存回寄存器。

介绍了 CPU 中的核心部件后，我们就知道了 CPU 的主要作用。

3.2.3　汇编指令的概念

为了便于理解程序执行的精髓，本章设计了几条 CPU 常用的“汇编指令”，来表达 CPU 的操作。实际的 CPU 中有各自的汇编指令集，功能强大而复杂。

在 a=a+1 之前再加入另一条程序语句 a=10，也就是先给变量 a 赋值，然后让 a 加 1。即

```
a = 10
a = a + 1
```

现在计算机顺序执行“a=10，a=a+1”语句，CPU 需要做以下几个操作，即“R 赋值”“将 R 存回 a”“读取 a 到 R”“R 加 1”和“将 R 存回 a”。分别用下面几条指令来表示每个操作。

1. “读取 a 到 R”操作——load 指令

程序语句中的“读取 a 到 R”，表示 CPU 将变量 a 读取到寄存器 R 中。设计指令 load 表示“读取 a 到 R”操作，那么 load 指令中需要有两个“操作数”，一个操作数是变量 a 的地址，另一个操作数是存储的寄存器。

格式：load R1，(address)

> **小明**：“操作数”是什么意思？
>
> **阿珍**：汇编指令由“操作码”和“操作数”组成。操作码是指令执行的基本动作。在 load R1，(address)指令中，load 是操作码，其后的寄存器 R1 和(address)都是操作数。操作码的英文叫作 operator，操作数的英文叫作 operand。

注：address 是内存地址，(address)表示这个地址内存储的值。

【例 3-1】 load R1，(1000)

该指令表示，将主存地址 1000 处的变量值读取到寄存器 R1 中。如图 3-4 中箭头所示是 load 指令执行的操作。(address)也可以用一个寄存器值加上一个偏移量(十六进制的常数)来表示，例如 load R1，04h(R2)。假设 R2 的值是 996，那就是将主存地址 996＋4＝1000 处的值读取到寄存器 R1 中。

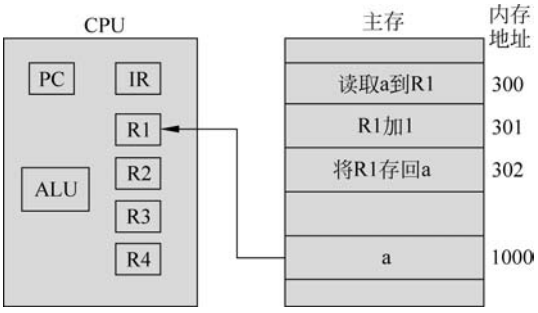


图 3-4　load 指令

2. “R 赋值”操作——mov 指令

程序语句中的“R 赋值”，表示给寄存器 R 赋一个值。设计指令 mov 来完成“R 赋值”操作，那么 mov 指令中需要有两个操作数，一个操作数是赋给的值；另一个操作数是寄存器。

格式 1：mov R1，constant

注：mov 指令有两个操作数，前一个是寄存器，后一个是十六进制的常数。

【例 3-2】 mov R1，0Ah

该指令执行的操作，是将一个常数值 0Ah(十进制的 10)赋给寄存器 R1。

我们希望，mov 指令还可以把一个寄存器中的值赋给另一个寄存器，那么 mov 指令的两个操作数都是寄存器。

格式 2：mov R2，R1

注：mov 指令有两个寄存器操作数。

该指令执行的操作，是将后一个寄存器 R1 中的值赋给寄存器 R2 中。

3. “R 加 1”操作——add 指令

程序语句中的“R 加 1”，表示将寄存器 R 的值加 1。设计指令 add 来完成“R 加 1”操作，那么 add 指令中需要有三个操作数，一个操作数是与变量 R 相加的值，一个操作数是存储变量 R 的寄存器，还有一个操作数是存储运算结果的寄存器。

格式 1：add R2，R1，constant

注：add 指令有三个操作数，一个是常数，还有两个是寄存器，这两个寄存器中后一个是进行运算的寄存器，前一个是存储运算结果的寄存器。该指令表示 R2＝R1＋constant。

【例 3-3】 add R1，R1，01h

该指令表示将寄存器 R1 中的数值加 1，并将结果存回寄存器 R1。如果寄存器 R1 中的值最初是 06h，执行该指令后，寄存器 R1 中的值就为 07h。

我们希望，add 指令还可以把两个寄存器中的值相加，赋给另一个寄存器中的变量，那么 add 指令的三个操作数都是寄存器。

格式 2：add R1，R1，R2

注：add 指令有三个寄存器操作数。

该指令执行的操作，是将后两个寄存器 R1、R2 中的值相加，结果赋给寄存器 R1，也就是 R1＝R1＋R2。

4. 减法指令 sub

同 add 指令的格式一样。“sub R2，R1，constant”代表了 R2 ＝ R1 － constant，“sub R3，R1，R2”代表了 R3＝R1－R2。

5. 左移位指令 shiftl

“shiftl R3，R1，05h”代表寄存器 R1 的二进制数左移 5 位，移出的那 5 位填 0，再将最终值存入 R3。05h 也可以用一个寄存器表示，例如，“shiftl R3，R1，R2”代表 R1 的二进制值向左移(R2)位数，存入 R3。左移指令就相当于将 R1 做乘法。R1 左移一位，R1 值就相当于乘 2，R1 左移 2 位，R1 值就相当于乘 4。

6. 右移位指令 shiftr

向右移位，移完后的那些最高位填 0。

7. “将 R 存回 a”操作——store 指令

程序语句中的“存回”，表示 CPU 将寄存器 R 中的值存回主存中。设计指令 store 表示“存回 R”操作，那么 store 指令中需要有两个操作数，一个操作数是寄存器 R；另一个操作数是要存回的地址 a。

格式：store (address)，R1

注：address 是内存地址，(address)表示要存回的地址，R1 是寄存器。也就是(address)＝R1。

【例 3-4】 store (500)，R1

该指令表示将寄存器 R1 中的值存回主存地址 500 处。(address)也可以用一个寄存器值加上一个偏移量(十六进制的常数)来表示，例如 store 04h(R2)，R1。假设 R2 的值是 496，那就是将 R1 的值存回主存地址 496＋4＝500 处。

> **小明**：程序执行时，为什么 CPU 先把变量读取到寄存器中，再转移到 ALU 中进行运算，而不是直接把变量读取到 ALU 中进行运算呢？
>
> **沙老师**：因为 CPU 和主存之间的数据读写速度远远比 CPU 与寄存器之间的速度慢。CPU 寄存器的读写只要 1 个单位时间，而对主存的读写可能要高达 50 个单位时间。如果每次 ALU 运算的输入都要从主存读取，那就要花很长的时间了。用寄存器保存了程序执行时的中间运算结果，执行多次快速 ALU 运算后，再将结果存回内存中，这样会比每次运算都从内存来输入输出要快得多。

3.2.4 a＝a＋1 的完整执行过程

为了让计算机执行“a＝10，a＝a＋1”程序语句，用几条汇编指令来指示 CPU 的操作。先把“a＝10，a＝a＋1”这两条程序语句用相应的汇编指令来表示，汇编指令的执行步骤如下。

(1) 程序开始执行时，变量 a 存储在主存地址 1000 处。a＝10，a＝a＋1 程序语句有五条汇编指令，从地址 301 处开始顺序存储每条指令。程序开始执行时，PC 指向汇编程序的

首地址 301 处。

(2) 如图 3-5 所示,CPU 从地址 301 处开始执行,PC 值为 301,CPU 从地址 301 处读取 mov 指令到 IR,解读并执行 mov 指令,给寄存器 R1 中的变量 a 赋初值 10,然后 PC 加 1,指向下一条汇编指令。

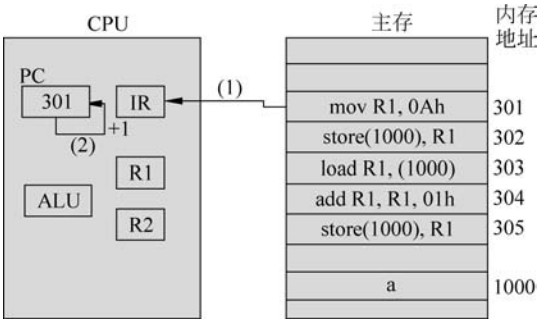


图 3-5 mov 指令的执行

(注:图中的 CPU 和主存是实际的计算机部件的简单示意图。计算机的 CPU 中通常有 32 个寄存器,为简单起见,此处只画出 R1、R2 这两个寄存器)

(3) 如图 3-6 所示,PC 值为 302,CPU 从地址 302 处读取 store 指令到 IR,解读并执行 store 指令,将寄存器 R1 中变量 a 的值存回主存地址 1000 处,然后 PC 加 1,指向下一条汇编指令。

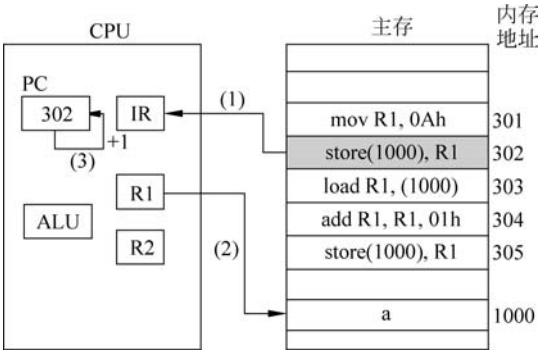


图 3-6 store 指令的执行

(4) 执行 load 指令,如图 3-4 所示,之后 PC 指向 add 指令。如图 3-7 所示,PC 值为 304,CPU 从地址 304 处读取 add 指令到 IR,解读并执行 add 指令,将寄存器 R1 中变量 a 的值加 1,并将结果再存回寄存器 R1,然后 PC 加 1,指向下一条汇编指令。

小明:好像第三个语句的 store 和第四个语句的 load 是可以去掉的,是吗?

沙老师:没错。从高阶语言转换为低阶的汇编语言,这个过程是由一个程序叫作编译器来完成的。编译器会做进一步的优化,将这两个语句去掉。

(5) 执行 store 指令,同图 3-6,该指令把寄存器 R1 中变量 a 加 1 后的值存回主存地址 1000 处。

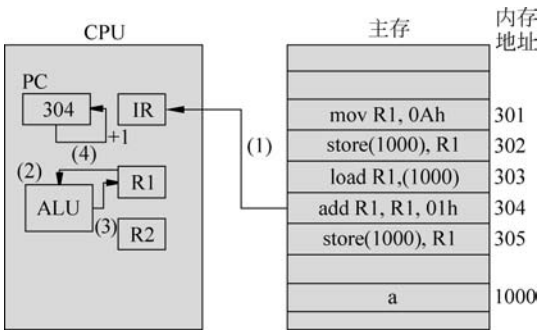


图 3-7　执行 add 指令

小结

本节探索了“a＝10，a＝a＋1”这种简单赋值语句在计算机中是如何执行的。首先，我们描述出计算机执行这些程序时，CPU 需要做哪些操作。但计算机并不理解我们的描述，所以 CPU 设计者就设计了相应的“汇编指令”来指示 CPU 的操作。简单起见，我们并没有使用真实的某一种计算机体系下的汇编指令集，而是设计了可以表达相似功能的指令来为大家讲述基本概念。通过使用汇编指令，告诉 CPU 要进行的工作，从而正确地执行程序。

练习题 3.2.1：CPU 执行程序语句 a＝20 时，基本的操作是什么？

练习题 3.2.2：指令“load R2，(1200)”执行的操作是什么？

练习题 3.2.3：用 shiftl 指令将 R1 值乘以 10h(十进制的 16)，存入 R3。

练习题 3.2.4：用 shiftl 和 add 指令将 R1 值乘以 0ch(十进制的 12)，存入 R3。add 指令用的越少越好。

练习题 3.2.5：假设变量 x 在主存地址 600 处，变量 y 在主存地址 604 处。请写出“x＝x－y”的汇编程序。

练习题 3.2.6：对于向右移指令 shiftr，我们是将移完后的那些位元填 0，但是普通 CPU 还有一个向右移的指令，它是填上原来的最高位值，也就是原来的最高位是 1，右移后所有的新位元就填 1，原来的最高位是 0，右移后就填 0。这个指令叫“算数右移”，请问这个指令的作用是什么？

提示：可能和负数的表示有关。

练习题 3.2.7：假设变量 x 存储在主存地址 600 处，执行完下列汇编指令后，地址 600 处存储的数据是多少？

```
load  R1, (600)
mov  R1, 09h
store  (600), R1
```

3.3　控制结构的执行

要解决一个问题，一定会用到控制语句，会用到一些分支判断的程序语句，如 if-else 语句、for 循环、while 循环等。那么这些语句的执行逻辑是怎样的呢？本节我们来探索控制结构的执行过程，首先学习 if-else 选择语句在计算机中是怎样执行的。

3.3.1 if-else 选择语句

不同的程序语言定义了不同的 if-else 选择、for 循环等控制结构的表达形式。但是 if-else、for 语句的执行逻辑是不变的。

如图 3-8 所示是 if-else 选择语句的简化形式。

if-else 的执行逻辑是：先判断 if 后面的表达式，如果表达式成立，则执行语句块 A，否则就执行语句块 B。在图 3-8 中，如果 x 小于 y，则执行语句块 A，否则执行语句块 B。if-else 选择语句，只选择其中一个语句块来执行，之后执行 if-else 结构后面的语句块。

图 3-8 if-else 选择语句的简化表达

那么，我们怎么把 if-else 的执行逻辑告诉计算机呢？

首先，我们需要比较 x 和 y 的大小，由一条语句“比较 x 是否小于 y”来告诉 CPU 应该进行判断操作。

接下来，CPU 应该保存这个比较结果，根据比较结果，产生以下两种执行的情况。其一，选择顺序执行语句块 A，执行完 A 的所有语句后直接跳转到语句块 C；其二，不执行语句块 A，而是选择跳转到语句块 B，执行完 B 的所有语句后，顺序执行语句块 C。

小明：“直接跳转到语句块 C”和“选择跳转到语句块 B”，这两个“跳转”操作有什么区别？

阿珍：“直接跳转到语句块 C”表示必须跳转到语句块 C；“选择跳转到语句块 B”表示 CPU 先做判断工作，根据这个判断的结果，来选择是否要跳转到语句块 B。

3.3.2 分支跳转指令

我们用自己的语言描述了 CPU 在执行 if-else 选择语句时的操作。现在需要新增几条汇编指令，来指示 CPU 执行的操作。本节将介绍“比较 x 是否小于 y”“选择跳转到语句块 B”操作如何用相应的汇编指令来表示。

1. “比较 x 是否小于 y”——slt 指令

我们设计指令 slt 来告诉 CPU 进行比较操作，slt 需要三个操作数，后两个操作数依次是存储变量 x 和变量 y 的寄存器，另一个寄存器用来保存比较结果。

格式 1：slt R4，R1，R2

该指令执行的操作，即比较寄存器 R1 中的数值是否小于 R2 中的数值，如果小于，则将寄存器 R4 置 1，否则置 0。

我们还希望 slt 能够比较寄存器中的变量和一个数值的大小。那么 slt 的后两个操作数分别是保存变量的寄存器、常数值，而另一个寄存器用来保存比较结果。

格式 2：slt R4，R1，constant

该指令执行的操作，即比较寄存器 R1 和常数值 constant，如果 R1 中的数值小于 constant，则寄存器 R4 置 1，否则置 0。

【例 3-5】 slt R4，R1，0Ah

该指令表示，比较 R1 寄存器中的数值是否小于 0Ah(即十进制的 10)，如果小于，则 R4 寄存器置 1，否则置 0。

2. “判断小于或等于”——sle 指令

sle 和 slt 的格式完全一样。例如“sle R4，R1，constant”，即比较寄存器 R1 和常数值 constant，如果 R1 中的数值小于或等于 constant，则寄存器 R4 置 1，否则置 0。

3. “选择跳转到语句块”操作——beqz 指令

CPU 已经将比较的结果保存到寄存器，接下来，CPU 根据寄存器中的值(0 或 1)来判断执行哪一个语句块。指令 beqz 来查看寄存器中的值是否为 0。如果为 0，CPU 将不再按顺序执行下一条语句，而是跳转到另一个语句块。对于将要跳转到的语句块，我们可以用一个“标签(Label)”来标记。beqz 需要两个操作数，前一个操作数是存储比较结果的寄存器，另一个寄存器是一个标签。

这里 beqz 代表了 branch if equals zero 的意思。

格式：beqz R4，label

注：“标签”术语第一次在本书中提及，汇编程序中有些指令块用标签 label1、label2 等标记，执行时就可以根据条件跳转，或者直接跳转到这些指令块处执行。beqz 指令是根据条件来跳转的指令。它有两个操作数，一个是保存比较结果寄存器；另一个是标签。

【例 3-6】 beqz R4，label2

该指令表示，如果寄存器 R4 中的数值为零，则跳转到标签 label2 标记的指令块处。

4. “直接跳转到语句块”操作——goto 指令

格式：goto label

注：beqz 指令是根据条件来选择是否跳转，goto 指令是告诉 CPU 进行直接跳转的指令。它只有一个操作数，即“标签”label。

执行操作：跳转到标签 label 标记的指令处。

【例 3-7】 goto label3

表示跳转到标签 label3 标记的指令处执行。

3.3.3 if-else 选择语句的执行

现在，我们把 if-else 选择语句翻译为汇编指令。在前个小节的 if-else 结构中，我们用到两个变量 x 和 y 在 if x＜y 中。假定已经把 x 和 y 分别读取到寄存器 R1 和 R2 中。用汇编指令表示 CPU 在执行 if-else 选择语句时的操作，如图 3-9 所示。

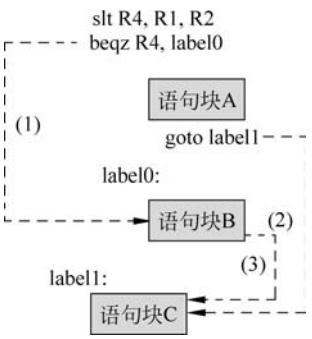


图 3-9 汇编指令表述 if-else 的执行

slt 指令比较 x 和 y 的大小，如果 x 小于 y，则寄存器 R4 置 1，否则置 0；

beqz 指令判断 R4 的值，根据 R4 是否为 0，有两种执行情况。

其一，R4 为 1，即 x 小于 y，则顺序执行语句块 A，也就是

程序中 if 之后的语句。执行完语句块 A 的所有语句后,会执行 goto 指令,直接跳转到语句块 C,如图 3-9 中虚线(2)所示。

其二,R4 为 0,即 x 不小于 y,则跳转到语句块 B 执行,也就是程序中 else 之后的语句,如图 3-9 中虚线(1)所示。执行完语句块 B 中的所有语句后,顺序执行语句块 C,如图 3-9 中虚线(3)所示。

现在我们来细看 if-else 选择语句的执行过程。

(1) 我们假定,if-else 选择语句翻译后的汇编指令从地址 304 处开始存储在主存,所使用到的变量 x 和 y 已经从主存地址 1000、1001 处分别读取到寄存器 R1 和 R2 中。

(2) 如图 3-10 所示,执行 slt 指令,CPU 先将 slt 指令读取到指令寄存器 IR,进行解读。之后 CPU 将寄存器 R1 和寄存器 R2 中的数值转移到 ALU 中。对于比较运算,ALU 通过对两个数值做减法来判断。最终将比较的结果存回到寄存器 R4 中。PC 加 1,指向下一条指令 beqz。

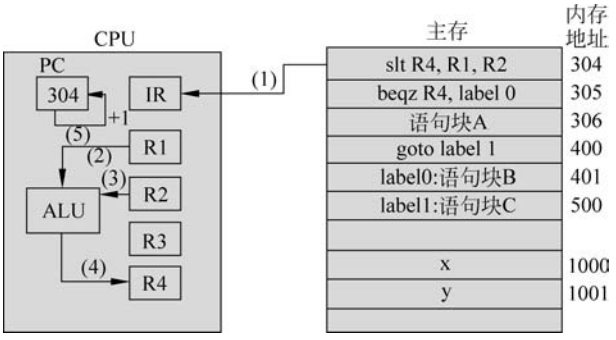


图 3-10　执行 slt 指令

(3) 如图 3-11 所示,执行 beqz 指令,CPU 先将 beqz 指令读取到指令寄存器 IR,进行解读,之后 CPU 判断寄存器 R4 的值。

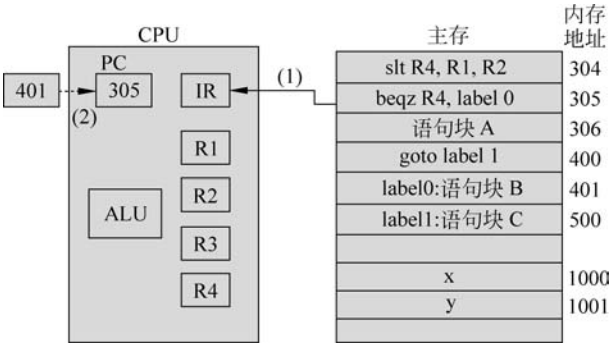


图 3-11　执行 beqz 指令,假如 x≥y

(4) 变量 x 和 y 有两种大小关系,若 x 不小于 y(x≥y),CPU 将按照下面的步骤(5)和步骤(6)执行。否则,按照步骤(7)～(9)执行。

(5) 当 x≥y 时 R4 中值为 0,则跳转到 label 0 处执行。如图 3-11 中虚线(2)所示,PC 值变为 401,指向 label 0 处,即语句块 B 的第一条语句。

(6) 执行完语句块 B 中的所有语句后，结束 if-else 选择语句。此后 PC 值为 500，顺序执行语句块 C。

(7) 当 x＜y 时，R4 是 1，执行 beqz 指令，不跳转到 label0 处执行，而是顺序执行语句块 A 的第一条语句。这时 PC 的值为 306，指向语句块 A 的第一条语句。

(8) 执行至语句块 A 的最后一条语句时，PC 值为 400，指向 goto 指令。

(9) 如图 3-12 所示，CPU 执行 goto 指令，跳转到 label1。如图 3-12 中虚线(2)所示，PC 值变为 500，直接执行语句块 C，结束 if-else 选择语句。

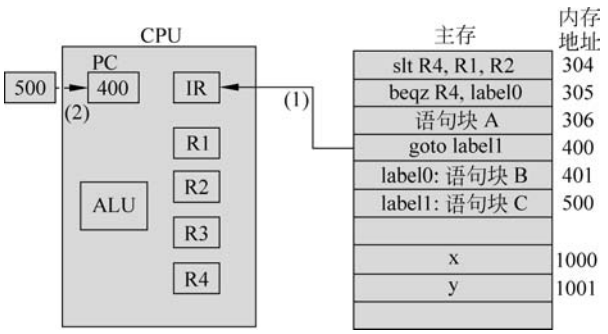


图 3-12　执行 goto 指令

3.3.4　while 循环语句的执行

if-else 选择语句能够使我们选择执行某一个语句块，接下来，我们需要考虑如何重复执行语句。我们希望计算机能够重复执行某一个语句块。

在程序设计语言中，通常有两种循环控制结构，即 while 循环和 for 循环。我们先来了解一下 while 循环的执行逻辑。

基本 while 语句

if-else 选择语句根据表达式的真与假来选择其中一个语句块执行。我们需要计算机循环执行某一个语句块，而循环都需要有一个终止条件，那么，我们也可以根据表达式的真与假，来决定是否终止。如果表达式的值为假，我们就终止执行，否则继续重复执行。

图 3-13 是一个 while 循环的例子。我们通过这个例子来了解 while 循环的执行过程，如下：

图 3-13　while 循环结构的例子

(1) 比较变量 x 和 y 的大小，如果 x 小于 y，则执行语句块 A，否则执行语句块 B。

(2) 重复判断变量 x 是否小于 y，如果小于，则重复执行语句块 A。直到变量 x 不再小于 y，此时不执行语句块 A，而是结束 while 循环，执行语句块 B。

汇编指令描述 while 语句的执行

下面我们来看一下，用汇编指令如何表述 while 循环的执行逻辑。

如图 3-14 所示，假定变量 x 和 y 已经分别读取到寄存器 R1 和 R2 中。我们将计算机执行 while 循环语句时 CPU 需要做的操作翻译为汇编指令，如图 3-14 所示。

此后，CPU 将执行图 3-14 所示的汇编指令。步骤如下。

(1) CPU 执行 slt 指令,比较寄存器中的变量 x 和 y 的大小,并将比较结果保存到寄存器 R4 中。如果 x 小于 y,则 R4 置 1,否则置 0。

(2) CPU 执行 beqz 指令,如果 R4 中值为 0(就是 R1 不小于 R2),就跳转到步骤(5)。否则,R4=1(即 R1 小于 R2),则不跳转,顺序执行步骤(3)。

(3) CPU 顺序执行下一条语句,也就是语句块 A 中的第一条语句,并顺序执行完语句块 A 中的所有语句。

(4) CPU 执行 goto 指令,执行后的结果是跳转到 slt 指令,如图虚线(1)所示。即跳转到步骤(1)。

(5) 结束 while 循环结构。跳转到 label0 处,执行语句块 B,如图虚线(2)所示。

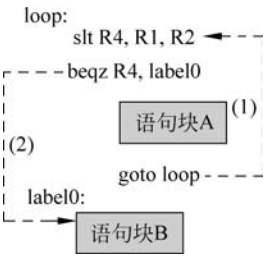


图 3-14 用汇编指令表述 while 循环的执行

(注:完整的汇编指令还应该包括,将变量 x 和 y 分别读取到寄存器 R1、R2 中并赋值的操作,之后变量 x 和 y 就分别有了初值)

3.3.5 for 循环语句的执行

编写程序时,for 循环很常用。通常,for 循环是用来告诉计算机要重复执行语句块达到多少次,但是一些程序语言,如 Python 中的 for 循环有时也不需要在程序中表示需要执行的次数。

1. 基本 for 循环结构

不同的程序语言中,定义了不同的 for 循环语句形式,但 for 循环的执行逻辑却是大同小异。如图 3-15 所示为 for 循环语句的基本形式。通常,for 循环语句会有一个变量 i 来控制循环次数,每执行一次语句块,变量 i 的值会做相应的变化。假定需要循环执行 10 次,变量 i 取初值 0,执行语句块 A。之后变量 i 取值 1,执行语句块 A。接下来变量 i 取值 2,重复执行语句块 A,直到变量 i 的值不再小于 10,就不再重复执行语句块 A,而是终止 for 循环,即执行语句块 B。

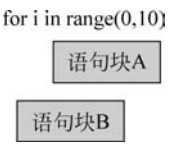


图 3-15 基本 for 循环结构

(注:语句"for i in range(0,10)"表示 0≤i<10,i 取值 0,1,2,…,9)

现在,我们来细看 for 循环结构的执行逻辑。

(1) 我们有一个变量 i 来记录循环次数,先决定一个寄存器来代表 i。

(2) 给变量 i 赋一个初值。

(3) 比较变量 i 是否小于设定的常数,如果小于,则执行步骤(4),否则跳转到步骤(5)。

(4) 执行语句块 A。然后变量 i 加 1,之后直接跳转到步骤(3)。

(5) 结束 for 循环。执行语句块 B。

沙老师:虽然 while 循环可以取代 for 循环,但是 for 循环比较明白易懂,所以"遍历"就用 for 循环,能用 for 循环就用 for 循环,尤其是 Python 更要多用 for 循环。另外,while 循环的条件语句可繁可简。当 while 循环的条件语句太复杂时,请把这个条件语句用函数来表示,会比较清楚。

2. for 循环执行过程

下面，我们也用汇编指令的形式表达出 CPU 执行 for 循环语句时应该做的动作，如图 3-16 所示。

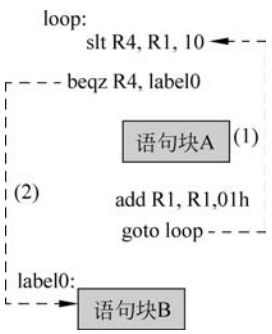


图 3-16　汇编指令表述 for 循环的执行

(1) CPU 执行 slt 指令，比较寄存器 R1 中的变量 i 和 10 的大小，并将比较结果保存到寄存器 R4。如果 i 小于 10，则 R4 置 1，否则置 0。

(2) CPU 执行 beqz 指令，如果寄存器 R4 中值为 1，则顺序执行步骤(3)，否则跳转到 label0，如图 3-16 虚线(2)所示。

(3) CPU 执行语句块 A 的第一条指令。之后，CPU 顺序执行完语句块 A 的所有语句。

(4) CPU 执行 add 指令，给寄存器 R1 中的变量 i 加 1。

(5) CPU 执行 goto 指令，执行后的结果是跳转到 slt 指令，如图虚线(1)所示，即跳转到步骤(1)。

其实，for 循环的执行过程和 while 循环很相似。在图 3-14 中，while 循环的语句块 A 中通常也有一条语句来更改循环变量 x 的值，否则变量 x 一直保持初值，就一直小于 10，那么就会一直执行语句块 A，这就是常说的“死循环”。

> **沙老师**：其实在汇编语言里跳转指令中的 label 并不是绝对地址。例如“goto X”，在真实的指令集里这些 X 是相对于现在指令地址的正负偏移量。**CPU 要算出目标地址就是 PC＝PC＋偏移量 X**。各位把我下面这句话记下来，这非常重要：CPU 执行两种计算：第一，地址的计算；第二，程序中变量的计算。地址的计算隐藏在程序执行后面，是 CPU 一直在做的计算，你们不可不知道啊！

在 Python 中，for 循环和 while 循环里面可以出现 break 语句，只要碰到 break 语句，就马上跳出循环。后面还可以跟着 else 语句，假如跳出循环的原因是因为碰到了 break 而跳出，循环后的 else 就不会执行；假如是正常离开循环，else 后面的程序块就会执行。详情请参阅第 4 章的 Python 介绍。

小结

本节逐步探索了 if-else 选择语句、while 循环语句、for 循环语句在计算机中的执行过程。我们还是先描述出计算机执行这些程序时 CPU 需要执行的操作，然后用相应的汇编指令来表示这些操作。在本小节，我们又添加了 slt、goto label、beqz R label 这些指令来表示 CPU 执行的操作。执行程序时，CPU 总是一条一条地取指令，解读，最后执行相应的操作。程序的执行，就是 CPU 不断取指令、执行指令的过程。

3.4　关于 Python 的函数调用

我们已经学习了基本语句 a＝a＋1 和控制结构语句(if-else 选择语句、while 循环语句、for 循环语句)的执行过程，下一步我们就要探索函数调用在计算机中的执行过程了。在此

之前,我们需要了解什么是函数,什么是函数调用,函数调用中的一些变量的作用范围等。本小节,我们将初步了解到 Python 中函数调用的相关内容。

3.4.1 函数的基本概念

回顾一下高中数学中的函数。在数学中,假设要实现 $z+xy^2$ 这个计算。对于乘法计算,定义一个函数 $f(x, y)=xy^2$,它有两个参数 x 和 y。计算 xy^2 后得到一个值,作为函数的返回值,赋给 $f(x,y)$。这样就可以用 $z+f(x, y)$来表示上面的运算,对于 $f(x, y)$运算,将会调用到已经定义的函数 $f(x, y)=xy^2$。

可以看到,数学中的函数有参数,有返回值,需要先定义,后调用。另外,还可以多处调用。也就是说,一旦定义了函数 $f(x, y)=xy^2$,我们在后面用到式子 xy^2 时,都可以用 $f(x, y)$代替,即所说的"多次调用"。

程序语言中的函数和数学中的函数的基本概念是相似的。程序语言中的函数也有参数和返回值,以及定义与调用。我们稍后将会看到,程序中的函数,就是将一些程序语句结合在一起的部件,通过多次调用,函数可以不止一次地在程序中运行。那么程序中使用函数会有什么好处呢?

第一,将大问题分成许多小问题。函数可以将程序分成多个子程序段,程序员可以独立编写各个子程序,实现了程序开发流程的分解。每个函数实现特定的功能,我们可以针对这个函数来撰写程序。

第二,便于检测错误。一个函数写好之后,我们会验证其实现的正确性。程序是由多个函数组成的。我们确定了每一个函数是正确后,总程序出错的可能性就会降低。另外函数的代码量小,也便于检测错误。

小明:有没有什么程序是不用函数的?

沙老师:有意义的程序都会用函数。我想一个程序的结果总要输出吧!不管是输出到屏幕或硬盘的文件系统,都要调用 I/O 输出函数,例如 print 函数。操作系统提供了这类函数供程序来调用。操作系统的功能之一就是提供一大堆的系统函数来"服务"程序。为了安全,程序不能直接使用 I/O 硬件,一定要请求操作系统的服务,让操作系统来使用 I/O 硬件。我们在操作系统的章节会做解释的。

第三,实现"封装"和"重用"。封装的意思是隐蔽细节。例如函数 GCD(x, y)是返回 x 和 y 的最大公约数。"封装"的特点体现在,对于各个求两数的最大公约数 GCD 的操作,都只需要传递两个参数 x 和 y 给函数 GCD,函数 GCD 会返回相应的结果,而不必关注 GCD 操作的具体实现。"重用"的特点体现在,各个程序都可以直接调用已经写好的 GCD 函数来实现最大公约数的计算,而不用重复编写代码。一个写好的函数,可以被多次调用,这种"重用"提高了程序的开发效率。

第四,便于维护。每一个函数都必须要有清楚的界面和注释,包含了功能、输入的参数、返回值的解释等。让人知道怎样调用这个函数。只要函数的界面不变,被调用函数的细节改变是不会影响全局的。

小明：函数真的这么有用啊？

沙老师：函数是非常有用的。一个好的编程的诀窍是：先从上而下，再从下而上。从上而下(Top-Down)决定了架构，要编写哪些函数和每一个函数的功能。再从下而上(Bottom-Up)，编写和检错每一个函数。这样程序就编成了。一个程序的美丑基本上就看你的程序是怎么分工、怎么定义和怎么使用函数了。

3.4.2 Python 函数入门

对于计算 $z+xy^2$，数学中用函数表达如下：

(1) 函数定义：$f(x, y)= xy^2$。

(2) 参数为 x 和 y。

(3) 返回值是 xy^2 的结果。

(4) 调用方式为 $z+f(a, b)$，a 和 b 是分别传递给函数 f 的具体数值。

Python 函数表达如下。

(1) 函数定义

```
def f(x, y):
    return x * y * y
```

Python 函数的定义由关键字 def 开始，后面跟上函数名和括号，括号里面是函数的参数，接着是冒号，最后就是函数体的内容。Python 函数定义的语法形式如下：

```
def 函数名(参数 1,参数 2,…):
    函数体
```

(2) 在上面定义的函数 f 中，参数也有两个，即 x 和 y，这些参数是函数 f 的“局部变量”，也就是它们的生命范围只限制在这个函数中(“局部变量”“全局变量”的相关概念，我们在后面会进行更详细的讲解)。调用函数 f 时，会传递实际的值赋给函数 f 的参数。每一个函数中都可以有 0 个、1 个或更多个参数，相邻参数之间用逗号隔开。形式如下：

```
参数 1,参数 2,参数 3,…
```

(3) 函数 f 中有一个关键字 return，其后跟的值就是本函数将返回的值，即“返回值”。假设函数 f0 调用函数 f，return 语句是将被调用的函数 f 的计算结果返回给调用 f 的函数 f0 中的变量。return 关键字后面可以是一个数值，也可以为一个表达式，在执行 return 语句后函数结束。一个函数可能有多条 return 语句，执行到第一条 return 语句时将结束函数。形式如下：

```
return 返回值或者表达式
```

如果进行调用的函数 f0 不需要被调函数 f 返回结果，那么被调函数就不需要 return 语句，即没有返回值。当然，Python 中的被调函数还可以返回多个值。

(4) 调用方式为“c=f(a, b)”。其中，a 和 b 是传递给函数 f 的值。比如，在函数 f0 中有这样一条语句“c=f(3, 2)”，3 和 2 就是函数 f0 传递给函数 f 的两个参数，即在 f 函数中

的局部变量 x 和 y 的值分别被设为 3 和 2。之后执行函数 f,计算 3×2×2 的结果并返回,返回的值赋给函数 f0 的变量 c。进行函数调用时,函数 f0 称为“主调函数”,而函数 f 称为“被调函数”。调用语句形式如下:

主调函数中的变量 = 被调函数名(参数 1,实数 2,…)

```
#<程序:计算 4 + 3 * 2² >
def f(x, y):
    return x * y * y
#主函数部分
c = 4 + f(3, 2)
print (c)
```

在计算 $4+3\times2^2$ 时,使用 Python 函数的示例#<程序:计算 $4+3*2^2$ >。运行示例程序,将会输出结果 16。

3.4.3 局部变量与全局变量

在函数中出现的变量,可以分为局部变量和全局变量。先记下这条规则:在函数中假如没有 global 语句,所有在等号左边出现的变量以及参数都是“局部变量(Local variables)”,它只能被这个函数访问,而不能被其他函数访问。在有些程序中,还有“嵌套函数”,嵌套函数是指在函数中再定义函数,但本书不使用这个功能,所以本书不谈“嵌套函数”。在本书中的变量就是两层,一层在函数内,一层在函数外,在函数之外被赋值的变量是“全局变量(Global variables)”。我们把局部变量搞清楚后,那些在函数中出现的变量,不是局部变量,就是全局变量。需要注意的是,在 Python 中,非函数和类里写的变量都是全局变量。

先来看这样一个例子#<程序:打印局部变量 a 和全局变量 a >。

```
#<程序:打印局部变量 a 和全局变量 a>
a = 10                          #函数外
def func():
    a = 20                      #函数内,局部变量的赋值,不会改变全局变量
    print(a)                    #函数内
func()
print(a)                        #函数外的 a
```

这里,func()函数里面和外面的变量名是一样的,都为 a,但输出结果却是不同的。a=10 语句中的变量 a 是函数外被赋值的变量,它为这个文件的全局变量,而 func()函数中 a=20 语句中的变量 a,是在 func()函数中被赋值的(在等号左边),就是局部变量。外部的变量 a 和 func()函数内部的变量 a 是不同的变量,只是拥有相同的变量名而已。所以,前面的例子将会输出 20 和 10。

判断函数内部的变量 a 是否为局部变量的方法:①不出现在 global 语句里面;②出现在函数参数中,或者出现在函数语句的等号左边。

在前面这个例子中,如果在函数中使用 global 语句来声明变量 a,那么这个变量 a 就是

全局变量 a，如＃<程序：关键字 global 引用全局变量>所示。

```
＃<程序：关键字 global 引用全局变量>
a = 10
def func():
    global a                    ＃宣告这个是全局变量
    a = 20
    print(a)
func()
print(a)
```

global 语句包含了关键字 global，后面跟着一个或多个用逗号分开的变量名。

在这个例子中，global 语句后跟着变量 a，表明该函数内使用的变量 a 是全局的。所以，func()函数中的 a＝20 语句会修改全局变量 a 的值，程序会输出 20 和 20。

然而，在不使用 global 语句声明某变量是全局时，如果这个变量出现在函数语句的等号左边，那么它就是局部变量。请看例子＃<程序：a，b，c 是否为局部变量？>。

```
＃<程序：a, b, c 是否为局部变量?>
b,c = 2,4
def g_func():
    a = b * c                   ＃a 是局部变量
    d = a                       ＃d 是局部变量,其他都是全局变量
    print(a,d)
g_func()
print(b,c)
>>>                             ＃输出结果
8 8
2 4
```

这里的函数 g_func()中，变量 a 和 d 是局部变量，因为它们没有被声明为 global 且出现在等号左边。变量 b 和 c 是全局变量，尽管它们没有被声明为 global，但是它们不是函数的参数，且只是出现函数中语句的等号右边。

练习题 3.4.1：运行下面这个程序，将会输出什么？在 g-func()函数中哪些是局部变量？

```
b, c = 2, 4
def g_func(d):
    global a
    a = d * c
g_func(b)
print(a)
```

练习题 3.4.2：运行下面这个程序，将会输出什么？

```
a = 10
def func():
    x = a
```

```
    print(x)
func()
print(a)
```


练习题 3.4.3：变量 a,b 是否为局部变量？再分析这个程序会输出什么？

```
a = 10
def func(b):
    c = a + b
    print(c)
func(1)
```

接下来，为加深理解，我们给出一个更复杂的 Python 程序 #<程序：四则运算例子>。

```
#<程序：四则运算例子>
def do_div(a, b):
    c = a/b                          #a, b, c 都是 do_div()函数中的局部变量
    print (c)
    return c
def do_mul(a, b):
    global c
    c = a * b                        #a, b 是 do_mul()函数的局部变量,c 是全局变量
    print (c)
    return c
def do_sub(a, b):
    c = a - b                        #a, b, c 都是 do_sub()函数中的局部变量
    c = do_mul(c, c)
    c = do_div(c, 2)
    print (c)
    return c
def do_add(a, b):                    #参数 a 和 b 是 do_add()函数中的局部变量
    global c
    c = a + b                        #全局变量 c,修改了 c 的值
    c = do_sub(c, 1)                 #再次修改了全局变量 c 的值
    print (c)
#所有函数外先执行:
a = 3                                #全局变量 a
b = 2                                #全局变量 b
c = 1                                #全局变量 c
do_add(a, b)                         #全局变量 a 和 b 作为参数传递给 do_add()函数
print (c)                            #全局变量 c
```

输出的结果是 16，8，8，8，8。我们来分析一下这个程序的执行过程。

(1) 调用 do_add()函数，将全局变量 a 和 b 传递给 do_add()函数。

(2) do_add()函数中，声明了全局变量 c。全局变量 c 的值改为 5。调用了 do_sub()函数，将全局变量 c 和数字 1 传递给 do_sub()函数，并将 do_sub()函数的结果返回给全局变量 c，即再次修改了 c 的值。

(3) do_sub()函数将参数 a 和 b 做减法，并将减法结果赋值给局部变量 c，此时局部变

量 c 的值为 4。注意，此时全局变量 c 的值仍为 5。调用 do_mul()函数，将局部变量 c 的值(为 4)传递给 do_mul()函数。

(4) do_mul()函数声明了全局变量 c，并将参数 a 和 b 相乘的结果赋值全局变量 c，全局变量 c 的值变为 16。打印出本程序的第一个结果，即 16。然后将结果返回给 do_sub()函数的局部变量 c。也就是说，do_sub()函数里的局部变量 c 的值不再是 4，而是 16。

(5) 调用 do_div()函数，并将局部变量 c 的值(为 16)和数字 2 传递给 do_div()函数。do_div()函数将参数 a 和 b 相除的结果赋值给局部变量 c，局部变量 c 的值为 8。注意，此时全局变量 c 的值仍为 16。打印出本程序的第二个结果，即局部变量 c 的值 8。然后将局部变量 c 的值 8 返回给 do_sub()函数的局部变量 c。

(6) 调用 do_div()函数的过程结束，程序返回到 do_sub()函数，打印出本程序的第三个结果，即 do_sub()函数的局部变量 c 的值 8。

(7) 调用 do_sub()函数的过程结束，并将 do_sub()函数的局部变量 c 的值 8 返回到 do_add()函数中，赋给全局变量 c。打印出本程序的第四个结果，即全局变量 c 的值 8。

(8) 调用 do_add()函数的过程结束，程序返回，打印出本程序的第五个结果，即全局变量 c 的值 8。

所以，程序最终输出的结果依次为 16，8，8，8，8。

> **沙老师**："global a"语句的目的是让全局变量 a 出现在函数里被赋值改变！这是不美的编程方法。函数应该像是一个黑盒子，它只有参数的输入和 return 的输出，细节过程是被隐蔽的。这个黑盒子不应该偷偷地改变了外部的全局变量。大家尽量不要用 global 语句，好吗？

上面这个程序是函数调用中稍微复杂的情形，并且用了 global 语句来声明全局变量。如果把 global 语句放在不同的函数中，输出结果会发生什么变化呢？

练习题 3.4.4：修改前面的程序，去掉 do_add()函数中的"global c"语句，分析程序将会输出什么？

练习题 3.4.5：执行下面的程序会出现什么错误？

```
#<程序：参数 a 能成为 global>
    a = 10
    def func(a):
        global a
        a = 20
        print (a)
    func(a)
    print (a)
```

练习题 3.4.6：结合下面的程序，思考一下，如果 func()函数中的某个等号左边和右边出现一个同样的变量名，如同下一个程序，为什么会出现错误？

```
local variable 'a' referenced before assignment.
```

```
#<程序：打印变量 a>
a = 10
def func():
    a = a + 10
    print(a)
func()
print(a)
```

提示：Python 语句中，首先决定出现在等号左边的 a 为局部变量，然后运算右边的 a+10，而这时 a 是没有值的。

关于局部变量和全局变量的更多细节，本小节就不过多讲解。如果大家遇到了这些问题，可以进行进一步的探索。如果有同学对 Python 中内置的_builtin_模块或者嵌套函数的使用感兴趣，也可以查阅资料，进行深入的学习。

小结

在本小节，我们先简单介绍了程序中的函数是什么，Python 函数的基本特点，以及函数的定义、调用、参数传递等。我们重点讲解了 Python 函数中的局部变量，当一个变量不出现在 global 语句里面，且出现在函数参数中，或者出现在函数语句的等号左边时，才能够被称为本函数的局部变量。其实，我们很少用到 global 语句，因为这样会在某一个函数中修改全局变量，对其他函数来说是隐藏的，可能会引起程序出错。局部变量、全局变量的概念对我们编写程序起着极其重要的作用。

3.5 函数调用过程的分析

在 3.4 节，我们了解了 Python 函数调用的相关内容，下面我们继续探索函数调用在计算机中的执行过程。在分析函数调用过程之前，我们先讲一下“栈(Stack)”的基础知识。

栈是一种非常重要的数据结构，它按照先进后出的原则存储数据，即先进入的数据被压入栈底，最后的数据在栈顶，需要取数据的时候从栈顶开始弹出数据。所以它的特色是“先进后出”或“后进先出”。

栈的特别之处在于，我们只能从一端放数据和取数据，就像一个桶一样，只能从桶口放东西和取东西。图 3-17(a)表示在栈中没有数据，此时栈底和栈顶指向同一个位置；将数据 1 放入栈中，执行压入(push)操作，如图 3-17(b)所示，1 被放入栈中，栈顶向上移；将数据 5 放入栈中，执行压入(push)操作，如图 3-17(c)所示，5 被放入栈中，栈顶向上移。

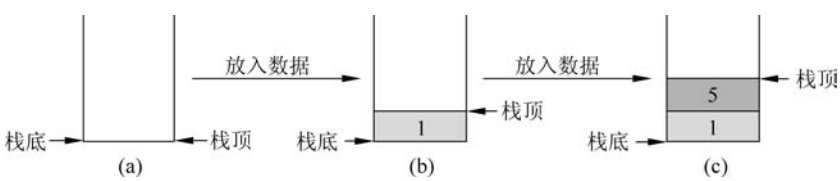


图 3-17　一个栈连续放入数据的过程

图 3-17 所示为连续放入数据的过程，下面我们来看从栈中取数据的过程。如图 3-18 所示为初始状态，有 3 个数据存在栈中；执行一次取数据(pop)操作，则在最顶上的数据 8 被弹出，得到数据 8，此时栈中的情况如图 3-18(b)所示；继续执行一次取数据(pop)操作，在最顶上的数据 5 被弹出，得到数据 5，此时栈中的情况如图 3-18(c)所示。总之，栈的基本操作就是 push 和 pop。

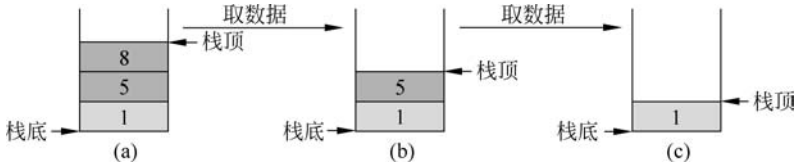


图 3-18　一个栈连续取数据的过程

由于栈的这种特殊的结构，在我们计算机科学中有着非常广泛的应用。例如给定一个单词 stack，想要把这个单词中的字母翻转，应用栈是很容易实现的，只需要将 s,t,a,c,k 这 5 个字母依次存入栈中，然后再取出就可以得到 k,c,a,t,s 了。

沙老师：用编程来解决问题时，我们常用的一些数据结构包含了数组(Array)、栈(Stack)、队列(Queue)、树(Tree)、图(Graph)等。Stack 是后进先出，Queue 是先进先出。有趣的是计算机里 Stack 用得多，而在人类社会里 Queue 用得多。想想我们在排队时，假如用 Stack 的方式，最后进的人最先得到服务，会怎么样？

小明：那也不错，大家都礼让别人，“抢着”做最后一个。

3.5.1　返回地址的存储

通过前面的学习，我们知道当执行一条指令时，总是根据 PC 中存放的指令地址，将指令由内存取到指令寄存器 IR 中。程序在执行时按顺序依次执行每一条语句，即执行完一条语句后，继续执行该语句的下一条语句。因此，PC 每次都通过加 1 来指向下一条将要执行的程序语句。

但也有一些例外，遇到这些例外情况时，不按顺序依次执行程序中的语句。这些例外如下。

(1) 调用函数；

(2) 函数调用后的返回；

(3) 控制结构，比如 if、for、while 等。

在本小节中，我们主要讲解函数调用及函数调用后的返回。

首先要明白一个基本概念：主调函数和被调函数。主调函数是指调用其他函数的函数；被调函数是指被其他函数调用的函数。一个函数很可能既调用别的函数，又被另外的函数调用。如图 3-19 所示的函数调用中，fun0 函数调用 fun1 函数，fun0 函数就是主调函数，fun1 函数就是被调函数。fun1 函数又调用 fun2 函数，此时 fun1 函数就是主调函数，fun2 函数就是被调函数。

发生函数调用时，程序会跳转到被调函数的第一条语句，然后按顺序依次执行被调函数

中的语句。函数调用后返回时,程序会返回到主调函数中调用函数的语句的后一条语句继续执行。换句话说也就是“从哪里离开,就回到哪里”。

例如,图 3-19 中的函数调用执行顺序如下。

(1) fun0 函数从函数的第一条语句开始执行,然后调用 fun1 函数,程序跳转到 fun1 函数的第一条语句,顺序执行 fun1 函数中的语句;

(2) fun1 函数调用 fun2 函数,程序跳转到 fun2 函数的第一条语句,然后按顺序执行 fun2 函数;

(3) fun2 函数执行完后,返回到 fun1 函数,继续执行 fun1 函数中“调用 fun2 函数语句”的下一条语句。在图中我们将 B 标示在该条语句旁边,表示该条语句的地址为 B。返回后按顺序执行 fun1 函数后面的语句;

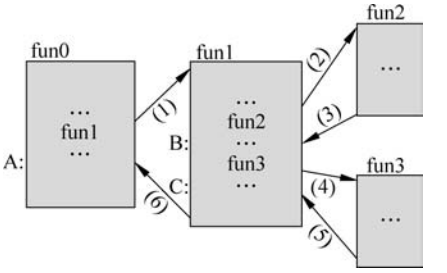


图 3-19 函数调用

(4) fun1 函数调用 fun3 函数,程序跳转到 fun3 函数的第一条语句,然后按顺序执行完 fun3 函数;

(5) fun3 函数执行完后,返回到 fun1 函数,继续执行 fun1 函数中“调用 fun3 函数语句”的下一条语句。在图中我们将 C 标示在该条语句旁边,表示该条语句的地址为 C。返回后按顺序执行 fun1 函数后面的语句;

(6) fun1 函数执行完后,返回到 fun0 函数,继续执行 fun0 函数中“调用 fun1 函数语句”的下一条语句。在图中我们将 A 标示在该条语句旁边,表示该条语句的地址为 A。返回后按顺序执行 fun0 函数后面的语句。执行步骤与图 3-19 中(1)~(6)一一对应。

我们在看具体的函数时,很容易看出发生函数调用时会跳转到哪一条语句,也很容易看出函数返回时会返回到哪一条语句。但是,这是因为我们是作为一个“局外人”来看,我们可以纵观整个程序。而 CPU 执行程序时,并不知道整个程序的执行步骤是怎样的,完全是“走一步,看一步”。前面我们提到过,CPU 都是根据 PC 中存放的指令地址找到要执行的语句。函数返回时,是“从哪里离开,就回到哪里”。但是当函数要从被调函数中返回时,PC 怎么知道调用时是从哪里离开的呢?

答案就是——将函数的“返回地址”保存起来。

因为在发生函数调用时的 PC 值是知道的。在主调函数中的函数调用的下一条语句的地址即为当前 PC 值加 1,也就是函数返回时需要的“返回地址”。我们只需将该返回地址保存起来,在被调函数执行完成、要返回主调函数中时,将返回地址送到 PC。这样,程序就可以往下继续执行了。

我们要合理地管理返回地址。观察函数调用及返回过程可发现,函数调用的特点是:越早被调用的函数,越晚返回。比如 fun1 函数比 fun2 函数先被调用,fun1 函数比 fun2 函数后返回;fun1 函数比 fun3 函数先被调用,fun1 函数比 fun3 函数后返回。这一特点刚好满足“后进先出”的要求,因此我们采用“栈”来保存返回地址。栈的基本操作就是压入和弹出。“压入 a”就是存放 a 在栈顶上;“弹出”就是将栈顶的值取出来,而后栈中就少了一个数据了。

图 3-20 给出了保存返回地址的过程。在图 3-19 中,调用过程(1)发生时,需要压入保存返回地址 A,栈的状态如图 3-20 中(a)所示;调用过程(2)发生时,需要压入保存返回地址

B,栈的状态如图 3-20 中(b)所示；返回过程(3)发生时,需要弹出返回地址 B,栈的状态如图 3-20 中(c)所示；调用过程(4)发生时,需要压入保存返回地址 C,栈的状态如图 3-20 中(d)所示；返回过程(5)发生时,需要弹出返回地址 C,栈的状态如图 3-20 中(e)所示；返回过程(6)发生时,需要弹出返回地址 A,此时栈被清空,图中未画出具体情况。所以函数调用时系统用栈来管理。

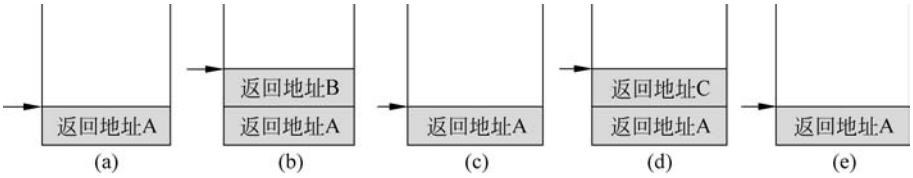


图 3-20　返回地址的存储

3.5.2　函数调用时栈的管理

事实上,函数的局部变量也是和返回地址绑定在一起用栈来管理。在本小节中,我们先为大家讲解局部变量的存储情况。

局部变量

我们用图 3-21 中的函数调用的例子来讨论变量的存储情况。

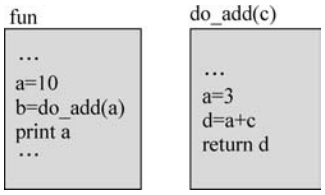


图 3-21　函数调用实例

在图 3-21 的函数中,fun 函数要调用 do_add()函数。fun 函数里有变量 a,a 的值为 10；在 do_add()函数里也有变量 a,a 的值为 3。虽然这两个函数中的变量 a 有相同的名字,但显然两个函数中 a 的值是不同的。fun 函数里的变量 a 和 do_add()函数里的变量 a 是两个不同的变量,即这两个变量需要存放在不同的地方。

do_add()函数中的局部变量 a 只有在函数内才有意义；局部变量的存储一定是和函数的开始与结束息息相关的。局部变量与返回地址一样,也是存在栈里,当函数开始执行时,这个函数的局部变量在栈里被设立(压入),当函数结束时,这个函数的局部变量和返回地址都会被弹出。

参数传递

在图 3-21 的例子中,调用函数时有参数的传递。fun 函数调用 do_add()函数时,需将 fun 函数里变量 a 的值传递给 do_add()函数里的变量 c。那么 fun 函数是怎样把变量 a 的值传递给变量 c 的呢？事实上,在调用有参数传递的函数时,变量 c 也是 do_add()函数里的局部变量,该局部变量由 fun 函数里的变量 a 来初始化。比如 fun 函数里变量 a 的值为 10,当调用 do_add()函数时,局部变量 c 就复制变量 a 的值 10。因此,在 do_add()函数里局部变量 c 的初始值就为 10。

返回值

在 do_add()函数中,最后有一条返回语句“return d”。表明在执行完 do_add()函数后,需要将局部变量 d 的值传递给主调函数 fun 函数的变量 b。与参数传递同理,在传递返回值时,也是将局部变量 d 的值赋值给主调函数中的变量 b。我们讲过,局部变量只在函数内

有意义,离开函数后该局部变量就失效。比如 do_add()函数里的局部变量 d,执行 do_add()函数时 d 是有意义的。但执行完 do_add()函数后,返回到 fun 函数中,do_add()函数里的局部变量 d 就失效了。因此在弹出 d 时需要用一个寄存器将返回值 d 保存起来,所以在外面的调用函数可以来读取这个值。

局部变量是在函数执行的时候才会存在。当函数结束后,这些局部变量就不存在了。如前所述,局部变量的调用和栈的操作模式"后进先出"的形式是相同的。这就是为什么返回地址是压入栈里,同样地,局部变量也会压到相对应的栈里面。当函数执行时,这个函数的每一个局部变量就会在栈里有一个空间。在栈中存放此函数的局部变量和返回地址的这一块区域叫作此函数的**栈帧**(**Frame**)。当此函数结束时,这一块栈帧就会被弹出。

接下来通过图 3-21 的例子来说明函数调用时这些信息的存储情况。图 3-22 展示了该例执行过程中栈的变化情况。

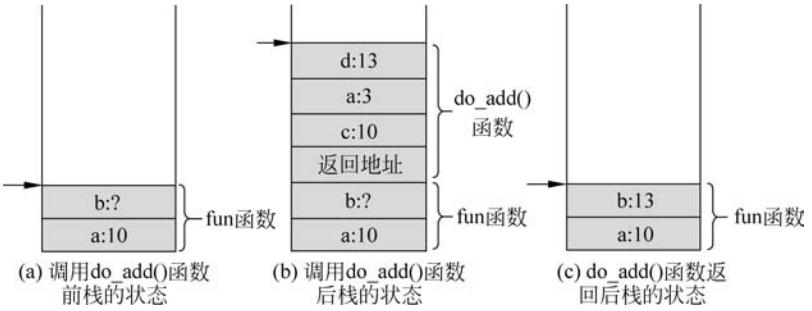


图 3-22　函数调用时栈的状态变化

在该例中,从函数 fun 开始执行(在函数 fun 之前,可能还有其他函数调用 fun,栈中也还存有其他数据,这里不详细讨论)。

(1) 调用 do_add()函数前执行的操作(该执行步骤中的(1)～(3)分别与图 3-22(a)、图 3-22(b)、图 3-22(c)中栈的状态一一对应)。

① fun 的局部变量 a 压入栈中,其值为 10;

② 局部变量 b 压入栈,由于 b 的值还未知,因此先为 b 预留出空间。

(2) 调用 do_add()函数时执行的操作。

① 返回地址压入栈中;

② 局部变量 c 的值 10 压入栈中。此处注意,c 是 do_add()函数中的局部变量,c 的值是通过复制 fun 函数中的局部变量 a 的值得到的;

③ 压入 do_add()函数中的局部变量 a,其值为 3;

④ 执行 a+c,其中 a=3,c=10,相加后得 d 的值为 13。

(3) do_add()函数返回时执行的操作。

① do_add()函数执行完后,依次弹出 do_add()函数的局部变量,由于需要将 d 的值返回,因此在弹出 d 时需要用一个寄存器将返回值 d 保存起来;

② 然后弹出返回地址,将返回地址传到 PC;

③ 返回到 fun 函数,fun 中的局部变量 b 的值即为 do_add()函数中的返回值 d,此时将寄存器中的值赋值给 b。

在将数据压入和弹出栈时，都需要用到栈顶的地址，因此需要将栈顶地址记录下来。在函数调用时，用一个寄存器将栈顶地址保存起来，称为栈顶指针 SP。另外还有一个帧指针 FP，用来指向栈中函数信息的底端。这样，栈就被分成了一段一段的空间，这样的一段空间我们就称为栈帧。

每个栈帧对应一次函数调用，在栈帧中存放了前面介绍的函数调用中的返回地址、局部变量值等。每次发生函数调用时，都会有一个栈帧被压入栈的最顶端；调用返回后，相应的栈帧便被弹出。当前正在执行的函数的栈帧总是处于栈的最顶端。

以图 3-19 中函数 fun1 依次调用 fun2 和 fun3 为例，图 3-23 中(a)～(d)为调用过程中栈空间的信息情况。首先在栈中将 fun1 函数的信息都存储起来，SP 与 FP 分别指向存储 fun1 信息的栈空间的顶端和底端，如图 3-23(a)所示；然后 fun1 函数调用 fun2 函数，在栈中将 fun2 函数的信息都存储起来，存储位置位于 fun1 函数的信息的顶部，SP 与 FP 分别指向存储 fun2 信息的栈空间的顶端和底端，如图 3-23(b)所示；fun2 函数执行完后，要返回到 fun1 函数中，fun2 函数的信息被弹出，SP 与 FP 分别指向存储 fun1 信息的栈空间的顶端和底端，如图 3-23(c)所示；fun1 函数又调用 fun3 函数，在栈中将 fun3 函数的信息都存储起来，存储位置位于 fun1 函数的信息的顶部，SP 与 FP 分别指向存储 fun3 信息的栈空间的顶端和底端，如图 3-23(d)所示；fun3 函数和 fun1 函数执行完后，也会分别返回，相应的信息会从栈中弹出，栈的状态未在图中画出。

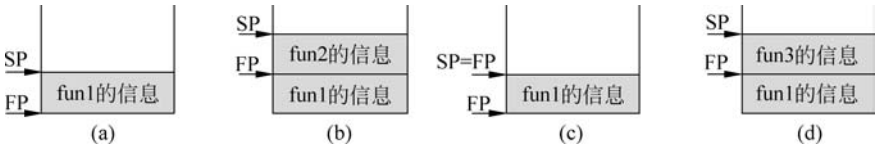


图 3-23　函数调用时栈空间的信息

由于函数调用时，要不断地将一些数据压入栈中，SP 的位置是不断变化的，而 FP 的位置相对于局部变量的位置是确定的，因此函数的局部变量的地址一般通过帧指针 FP 来计算，而非栈顶指针 SP。

综合前面所讲到的知识，可以总结出：一个函数调用过程就是将数据(包括参数和返回值)和控制信息(返回地址等)从一个函数传递到另一个函数。在执行被调函数的过程中，还要为被调函数的局部变量分配空间，在函数返回时释放这些空间。这些工作都是由栈来完成的，所传参数的地址可以简单地从 FP 算出来。图 3-24 展示了栈帧的通用结构。

为了使大家对函数调用时信息的存储了解得更加清晰，下面通过图 3-25 中递归函数的例子，将前面所讲的需要存储的信息综合在一起，来研究函数调用时对栈的管理。

在该例中，从函数 pre()开始执行(该执行步骤中的(1)～(5)分别与图 3-26 的(a)～(e)中栈的状态一一对应)。

(1) pre()函数调用 fac(1)函数前执行的操作。

① pre()的局部变量 m 压入栈中，其值为 1；

② 局部变量 f 压入栈，由于 f 的值还未知，因此先为 f 预留出空间。

(2) pre()函数调用 fac(1)函数时执行的操作。

① 返回地址压入栈中；

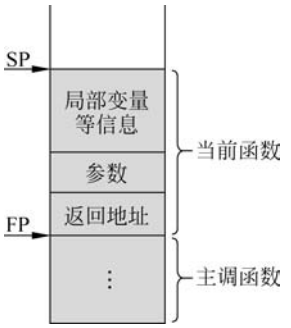


图 3-24　栈帧结构

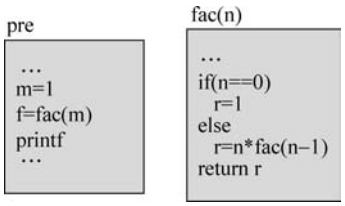


图 3-25　递归调用实例

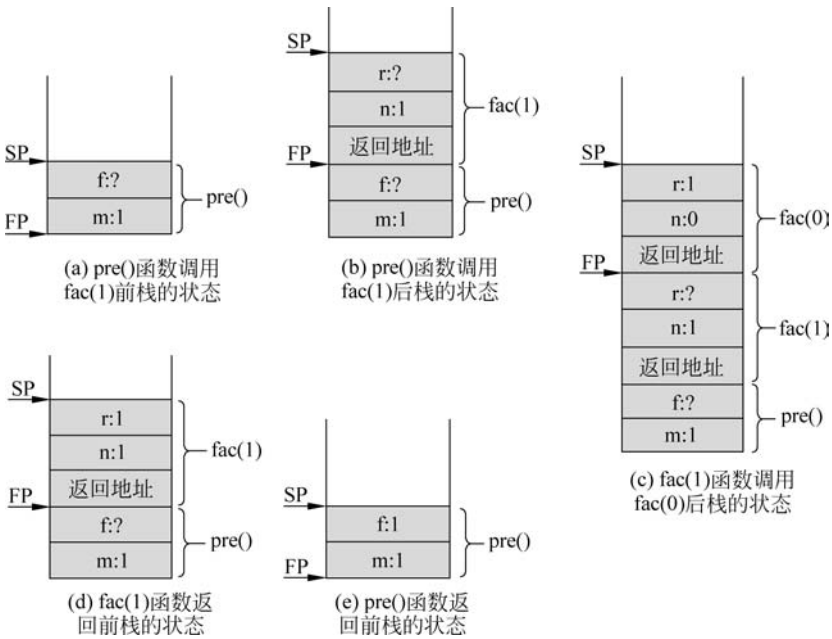


图 3-26　递归函数调用的栈示意图

② fac(1)的局部变量 n 压入栈中,其值为 1;

③ 局部变量 r 压入栈,由于 r 的值还未知,因此先为 r 预留出空间。

(3) fac(1)函数调用 fac(0)时执行的操作。

① 返回地址压入栈中;

② fac(0)的局部变量 n 压入栈中,其值为 0;

③ 此时递归达到了终止条件(n==0),结束递归,局部变量 r 压入栈,r 值为 1。

(4) fac(0)函数返回时执行的操作。

① fac(0)函数执行完后,依次弹出 fac(0)的局部变量。在弹出 r 时用一个寄存器将返回值 r 保存起来;

② 弹出返回地址,将返回地址传到 PC;

③ SP=FP,令 SP 指回 fac(1)栈帧的顶部,令 FP 指回 fac(1)栈帧的底部;

④ 继续执行 fac(1)函数,fac(1)中的局部变量 r 的值即为 fac(0)函数中的返回值乘

以 n。

(5) fac(1)函数返回时执行的操作。

① fac(1)函数执行完后,依次弹出 fac(1)的局部变量。在弹出 r 时用一个寄存器将返回值 r 保存起来;

② 弹出返回地址,将返回地址传到 PC;

③ SP=FP,令 SP 指回 pre()栈帧的顶部,令 FP 指回 pre()栈帧的底部;

④ 继续执行函数 pre(),pre()中的局部变量 f 的值即为 fac(1)函数中的返回值 r,此时将寄存器中的值赋值给 f。

各类微处理器对函数调用的处理方式会有所差异,同一体系结构中对不同语言的函数调用的处理方式也会有少许的差异。但通过栈存储局部变量和返回地址等信息,这一点是共同的。我们不需要对函数调用中的每一个执行的细节都了解清楚,大家只要对这个过程有一个初步的认识,知道每一次函数调用对应一个栈帧,栈帧中包含了返回地址、局部变量值等信息。还有一点需要注意,在本书中所用的 Python 语言属于解释性语言,Python 中发生函数调用时所建立的栈不是编译时建立的(像 C 语言等是在编译时就建好了栈),是在有需要的时候再建立的。

我们用一个因数分解的 Python 程序,来讲解 Python 程序运行时栈的建立过程。这个例子是用递归的方式来调用函数。

```
#<程序:因数分解> Print all the prime factors (>=2) of x. By Edwin Sha
import math                          #为了要调用平方根函数,此函数在 math 包里
def factors(x):                      #找到 x 的因数
    y=int(math.sqrt(x))
    for i in range(2,y+1):           #检查从 2 到 x 的平方根是否为 x 的因数
        if (x %i ==0):               #发现 i 是 x 的因数
            print("Factor:",i);
            factors(x//i)            #递归调用自己,参数变小是 x//i
            break                    #跳出 for 循环
    else:                            #假如正常离开循环,没有碰到 break,就执行 else 内的 print,
                                     #x 是质数
        print("Prime Factor:",x)
    print("局部变量: 参数 x:%d, 变量 y:%d" %(x,y))
    return

#函数外,先执行的部分
factors(18)      #找出 18 的所有因数
```

运行该因数分解的 Python 程序后,会输出什么呢? 我们先要先讨论这个 Python 程序的执行顺序以及栈的建立过程。

第 1 步,该程序从非函数定义的第一条语句开始执行,即语句 factors(18)开始执行。首先建立一个 main 函数的栈帧,栈帧中保存的信息为 main 函数中的信息。如图 3-27(a)所示。

第 2 步,第一次调用函数 factors(x)。先保存函数的返回地址。压入局部变量 x,值为 18;压入局部变量 y,值为 4(语句 y=int(math.sqrt(x))表示:y 的值等于 x 的值开平方

根后取下整。事实上,调用 math.sqrt(x)函数时也要建栈帧,大家知道即可,这里我们不详细讲解);压入局部变量 i,值为 2。此时程序执行到 if 语句,if 语句中的表达式值为真,因此会执行 print 语句,由于局部变量 i 值为 2,输出"Factor: 2"。栈的状态如图 3-27(b)所示。

第 3 步,第二次调用函数 factors(x)。先保存函数的返回地址。压入局部变量 x,值为 9;压入局部变量 y,值为 3;压入局部变量 i,值为 2。由于 if 语句中的表达式值为假,i 值会加 1,变为 3。此时 if 语句中的表达式值为真,因此会执行 print 语句,由于局部变量 i 值为 3,输出"Factor: 3"。栈的状态如图 3-27(c)所示。

第 4 步,第三次调用函数 factors(x)。先保存函数的返回地址。压入局部变量 x,值为 3;压入局部变量 y,值为 1。由于 i 值不能满足大于或等于 2 并且小于 2,所以不执行 for 循环,跳转执行 else 中的 print 语句,由于局部变量 i 值为 3,所以输出"Prime Factor: 3"。之后顺序执行下一条 print 语句,由于局部变量 x 值为 3,y 值为 1,所以输出"局部变量: 参数 x: 3,变量 y: 1"。栈的状态如图 3-27(d)所示。

第 5 步,程序顺序执行到 return 语句,弹出顶端的栈帧,返回到第二次调用函数 factors(x)后的状态。程序返回到语句 factors(x//i),顺序执行 break 语句,退出 for 循环。是由 break 跳出的所以不会执行 else 中的 print 语句,由于局部变量 x 值为 9,y 值为 3,所以输出"局部变量: 参数 x: 9,变量 y: 3"。栈的状态如图 3-27(e)所示。

第 6 步,程序顺序执行到 return 语句,弹出顶端的栈帧,返回到第一次调用函数 factors(x)后的状态。程序返回到语句 factors(x//i),顺序执行 break 语句,退出 for 循环。是由 break 跳出的所以不会执行 else 中的 print 语句,由于局部变量 x 值为 18,y 值为 4,所以输出"局部变量: 参数 x: 18,变量 y: 4"。栈的状态如图 3-27(f)所示。

第 7 步,程序顺序执行到 return 语句,弹出顶端的栈帧,返回到第一次调用函数 factors(x)前的状态。栈的状态如图 3-27(g)所示。程序返回到语句 factors(18)。执行完 main 函数后,弹出顶端的栈帧,此时栈为空(图中未画出)。

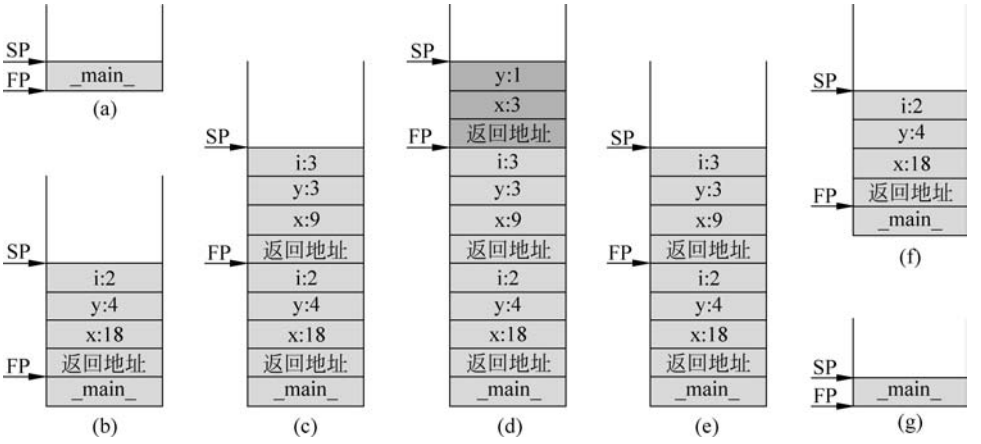


图 3-27 因数分解程序的栈示意图

在上述步骤中,第 1 步至第 4 步为函数的调用过程,第 5 步至第 7 步为函数的返回过程。

程序运行的结果：

```
Factor: 2
Factor: 3
Prime Factor: 3
局部变量: 参数 x:3,变量 y:1
局部变量: 参数 x:9,变量 y:3
局部变量: 参数 x:18,变量 y:4
```

经过之前的分析，我们知道程序运行的顺序，知道每一步输出的结果，也了解函数调用时建立栈帧的过程。程序运行的结果与我们的分析一致。

3.5.3 SEAL 中函数调用栈帧的建立

本书配套了一个笔者开发的汇编语言模拟器，称为 SEAL（Simple Educational Assembly Language）。通过 SEAL，读者可以撰写并执行本章所描述的汇编语言程序，并便于侦错除虫。从前言中介绍的出版社网站可以免费下载此工具，请读者根据所附的使用手册及范例程序进行学习。SEAL 实现了 24 条“高级”汇编语言指令，同时模拟了容量为 10000 的内存及 18 个寄存器，其中包含 16 个 64 位通用寄存器 R0～R15（可存储的数据大小为-2^{63}～$2^{63}-1$，支持十进制整型数和十六进制数）、1 个 PC 寄存器和 1 个 SP 堆栈指针寄存器。

3.5.2 节只简单描述了函数调用时栈的管理，还有一些细节需要读者进一步了解，尤其是在函数结束后 FP 如何返回主调函数的栈帧部位。本节将详细介绍 SEAL 中编写的函数调用是如何建立栈帧的。建立栈帧的过程叫作函数的连接（linkage）。SEAL 所使用的 linkage 规则与 x86 基本一致（x86 是工业界非常通用的 CPU 类型），读者可以由本节的学习了解到函数调用时关于栈帧建立的所有细节，这对奠定读者的计算机基础有莫大的助益。

接下来通过一个使用函数调用对两个数求和的示例来描述栈帧的建立过程。首先，给出使用函数调用对两个数求和的 Python 程序＃＜Python 代码：函数调用两个数求和>。

```
＃＜Python 代码:函数调用两个数求和>
def add(a,b):
    c = a + b
    return c
if __name__ == "__main__":
    x = 5
    y = 6
    print(add(x,y))
```

在本例中，假设调用函数称为主函数，被调用函数称为子函数。在以上代码中，主函数调用 add 子函数对两个数求和。那么在主函数调用 add()子函数时，CPU 执行了哪些指令？函数的栈帧是如何一步步地建立起来的？下面先给出以上 Python 程序＃＜Python 代码：函数调用两个数求和>所对应的汇编程序＃<汇编代码：函数调用两个数求和>。

```
#<汇编代码:函数调用两个数求和>
mov R15,10000        #R15 表示 fp,fp = 10000
mov sp,R15           #sp = fp
sub sp,sp,2          #sp 从 10000 开始向下开辟 2 个空间给局部变量 x 和 y,sp = sp-2
mov R2,5             #x = 5
mov R3,6             #y = 6
store -1(R15),R3     #x
store -2(R15),R2     #y

push R3              #传参数 b
push R2              #传参数 a
call Ladd            #调用函数 add(a,b), 返回值存储在 R1 中
goto Lprint

#add 函数有两个参数 a 和 b,将和放到 R1 中返回
Ladd:                #add(a,b)
push R15             #将旧的 fp 值压入栈内
mov R15,sp           #新的 fp = sp
sub sp,sp,1          #留一个空间,用于存储局部变量 c
push R2              #R2 将在函数中被更改,故先存入栈内,在 return 之前会 pop 出来该值
push R3              #R3 将在函数中被更改,故先存入栈内,在 return 之前会 pop 出来该值
load R2,2(R15)       #R2 = a
load R3,3(R15)       #R3 = b
add R1,R2,R3
store -1(R15),R1     #存储 c

Lreturn:
pop R3               #pop 出初始 R3 中的值
pop R2               #pop 出初始 R2 中的值
mov sp,R15           #sp = fp
pop R15              #重置 fp 为旧的 fp
ret

Lprint:
_pr R1
```

在函数调用中,需要用到栈操作指令 push、pop 及函数调用指令 call、调用返回指令 ret。在子函数开始执行前,需要在栈的顶部建立一个栈帧,SP 指针指向栈帧的顶部,FP 指针指向栈帧的底部(SP 和 FP 是两个寄存器)。一个栈帧中保存了主函数所传的参数值、函数内的所有局部变量、函数返回的 PC 值(也就是主函数调用子函数后的下一条指令的位置),以及主函数的 FP 值。总而言之,一个函数在执行前必须要将该函数的 FP 与 SP 建立起来,这个过程也就是本节一开始提到的函数的连接(linkage)。

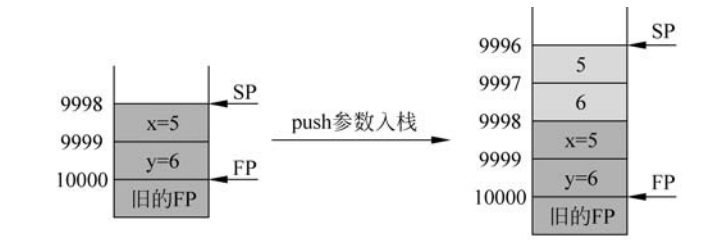
在 SEAL 中假设将 R15 作为 FP,而 SP 是一个专用的寄存器。当然,也可以将其他寄存器作为 FP,只要在编译函数时有一个统一的规则即可。SP 的值可以用汇编指令 add 和 sub 来更改,也可以用 push 或 pop 指令进行加 1 或减 1 运算。当执行 push R 时,该指令将 SP 减 1,然后将寄存器 R 的值存入 SP 所指的地址;当执行 pop R 时,该指令会将 SP 所指

地址的值 load 进寄存器 R，然后将 SP 加 1。假设主函数已经有一个栈帧，栈底是 FP，栈顶是 SP，如图 3-28(a)所示。在主函数中有两个局部变量 x 和 y，x 的地址是－2(R15)，y 的地址是－1(R15)，读者可以参阅主程序中关于 x＝5 和 y＝6 的汇编代码。

接下去主函数将调用子函数 add(x,y)，下面详细描述函数调用时建立新栈帧的过程。

1. 参数的传递

将参数以反向的顺序 push 入栈中，所以先 push y 的值，再 push x 的值，如图 3-28(b)所示。



(a) 主函数调用add子函数前栈的状态　　(b) 传递参数后栈的状态

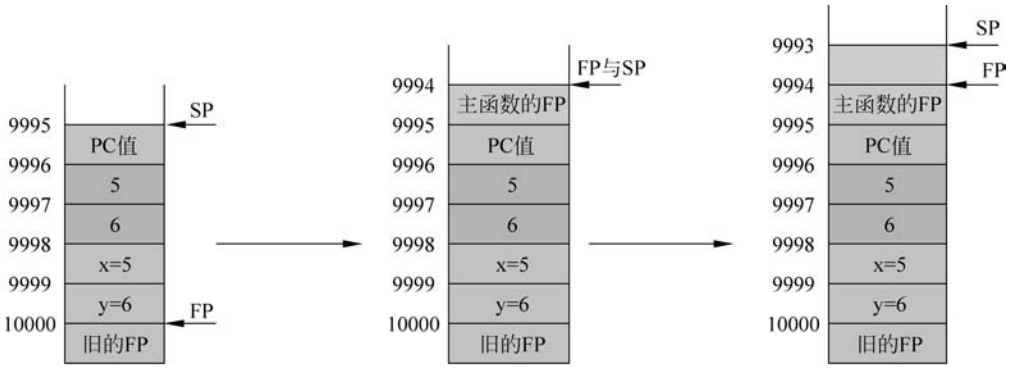
图 3-28　执行 call 指令前栈的状态

2. 执行 call 指令

call 指令会做两件事：第一，将 PC 值 push 入栈，如图 3-29(a)所示，这个 PC 值指向 call Ladd 的下一条指令，也就是子函数执行完后返回的地址；第二，执行 goto Ladd(即把 PC 值设成 Ladd 的地址)。

3. 函数起始的三条指令

这三条指令是所有函数开始时都会有的三条类似的指令：①“push R15”，将主函数的 FP 值存入栈中；②“mov R15, sp”，将新的 FP 指向 SP 的位置，即令 FP 指向此时栈的顶端，如图 3-29(b)所示；③“sub sp,sp,1”，将 SP 再上移，留出局部变量的空间，这里的 add 函数只有一个局部变量，所以只需要留一个位置，如果有 n 个局部变量，SP 就要减 n。执行这三条指令之后，栈的状态如图 3-29(c)所示。



(a) 存入返回地址后栈的状态　　(b) 将主函数的FP入栈后栈的状态　　(c) 最终栈的状态

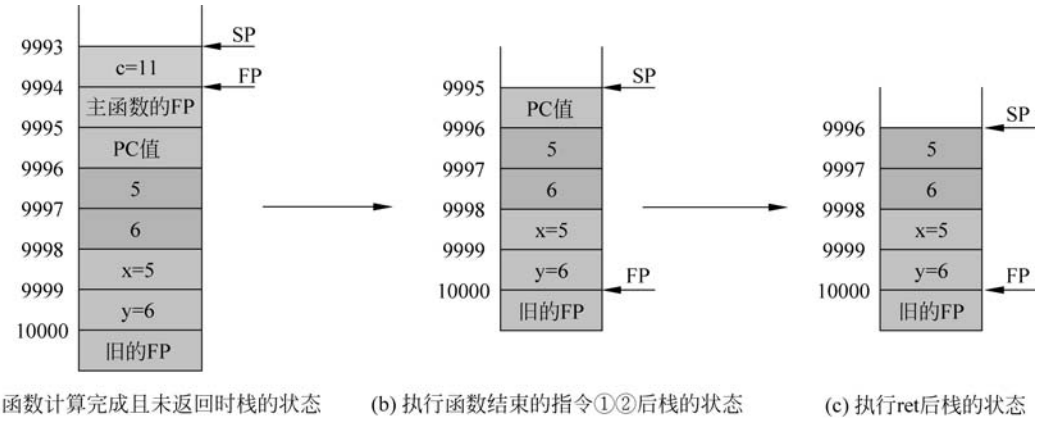
图 3-29　执行 call 指令之后栈的状态

4. add 函数中的计算

一个函数的栈帧建立之后,FP 会固定,SP 会随着函数中的 push、pop 指令而更改,所以通常是用 FP 作为基准位置来得到参数或函数内的局部变量的地址。在 add 函数中参数 a 的地址是 2(R15),参数 b 的地址是 3(R15),局部变量 c 的地址是 -1(R15),请参阅汇编代码中的相关 load、store 语句。函数的结果用 R1 传回主函数。

5. 函数结束的三条指令

这三条指令会将栈帧返回主函数的栈帧状态:①"mov sp,R15",将 SP 下拉到 FP 所指的位置;②"pop R15",返回主函数的 FP 值;③ret,相当于"pop pc",也就是返回到主函数调用子函数的下一条指令。函数返回时栈的状态如图 3-30 所示。



(a) 函数计算完成且未返回时栈的状态　(b) 执行函数结束的指令①②后栈的状态　(c) 执行ret后栈的状态

图 3-30　函数返回时栈的状态

经　验　谈

(1) 主函数将参数值压入栈后,这些参数的位置就会被子函数作为变量来使用,如本例中子函数的参数 a 和 b,其地址分别是 2(R15)和 3(R15)。所以,参数变量是属于子函数的局部变量。

(2)主函数是将参数的"值"传递给子函数,这种方式叫作"call by value"(值传递),是一种较为通用的参数传递方式,C 语言就采用这种方式。当然,这个值也可以用来传递参数的地址,只不过子函数要进行相应的更改,就如同在 C 语言中传递一个指针,那么子函数中就是对指针做运算,这里不再额外说明。

(3) 函数的返回值一般用寄存器 R1 返回,假如有多个返回值,可以用多个寄存器返回,但是要事先约定好。

(4) 除了返回的寄存器 R1 之外,其他的寄存器应该在函数调用后保持与函数调用之前相同的值,所以在函数计算开始前,需要将函数中会被更改的寄存器值 push 到栈中保存,在 return 前再逐一 pop 回来。例如,函数 add 更改了寄存器 R2 和 R3,所以在更改之

前先将 R2、R3 的值 push 入栈，在 return 前再 pop 返回原来的 R2、R3 的值，参见上述汇编代码。

(5) 返回主函数后，参数仍然留在栈中，主函数可以将参数 pop 出栈，但是需要付出消耗 pop 指令的代价，所以主函数常常“坐视不管”，让其留在栈中。但是对于递归函数，每次调用返回后必须将传递的参数 pop 出栈，否则后续的出栈操作可能会错位。

至此，函数调用时函数栈帧的建立过程介绍完毕。可以看到#<汇编代码：函数调用两个数求和>中的最后一条指令“_pr R1”，这是为了将结果输出到控制台，在 SEAL 中添加并实现的一个打印指令，可以将希望输出的值打印出来。

介绍了使用函数调用对两数求和时函数栈帧的建立过程之后，相信大家对函数调用已经有了一个全面的了解。接下来使用一个更复杂的示例介绍函数栈帧的建立过程，继续介绍前一个示例中未涉及的一些细节并详细说明该示例中每条指令的执行过程。

先给出函数调用求三个数中最小值的 Python 程序#< Python 代码：三个数求最小值>和汇编程序#<汇编代码：三个数求最小值>。

```
#< Python 代码:三个数求最小值>
def get_min(x,y):
    if x <= y:
        return x
    else:
        return y
if __name__ == "__main__":
    a = 7
    b = 18
    c = 9
    print(get_min(get_min(a,b),c))
#<汇编代码:三个数求最小值>
mov R15,300          #R15 表示 fp,将基地址设置为 300
mov sp,R15           #sp = fp
sub sp,sp,3          #sp 从 300 开始向下开辟 3 个空间给局部变量 a、b、c,sp = sp - 3

mov R2,7             #a = 7
mov R3,18            #b = 18
mov R4,9             #c = 9
store -1(R15),R4
store -2(R15),R3
store -3(R15),R2

push R3              #传参数 b
push R2              #传参数 a

call Lget_min        #调用函数 get_min(a,b), 返回值存储在 R1 中

push R4              #传参数 c
push R1              #传 a 和 b 中的较小值
```

```
call Lget_min          #再次调用 get_min(R1,c), 返回值存储在 R1 中
goto Lprint

#get_min 函数有两个参数 a 和 b,将最小值放到 R1 中返回
Lget_min:              #get_min(a,b)
push R15               #将旧的 fp 值存入栈内
mov R15,sp             #新的 fp 等于 sp
#sub sp,sp,0           #由于此函数没有局部变量,所以可以去掉
push R2                #R2 将在函数中被更改,所以先存入栈内,在 return 之前会 pop 出来
push R3                #R3 将在函数中被更改,所以先存入栈内,在 return 之前会 pop 出来
push R4                #R4 将在函数中被更改,所以先存入栈内,在 return 之前会 pop 出来
load R2, 2(R15)        #R2 存放 x 的值
load R3, 3(R15)        #R3 存放 y 的值
sle R4, R2, R3         #R4 = (R2 <= R3)
beqz R4, L100          #if (R4 == 0) goto L100
mov R1,R2              #R1 = R2,结果存储在 R1 中
goto Lreturn

L100:
mov R1,R3              #R1 = R3,结果存在 R1 中

Lreturn:
pop R4                 #返回初始 R4 中的值
pop R3
pop R2
mov sp,R15             #sp = R15
pop R15                #重设 R15 的值为原有的 R15
ret                    #pop pc

Lprint:
_pr R1                 #打印最后的结果
```

在本例中,主调函数要调用子函数 get_min()找出三个数中的最小值,需要比较两次才可以求得,因此子函数要被调用两次。

同样,要先设置主函数的 FP、SP 且初始时 SP 与 FP 指向同一地址,本例中将初始的 FP 和 SP 设置为 300。同时还需要为主函数的局部变量开辟足够的空间,本例中主函数有 3 个局部变量,所以需要开辟 3 个空间,使用 sub 指令对 SP 进行减 3 操作,于是 SP 指向开辟空间后的栈顶。将三个局部变量使用 store 指令按照从高到低的地址依次进行存储,此时栈的状态如图 3-31(a)所示。

接着第一次调用 get_min()子函数,子函数被调用前需要将前两个数作为参数传递给子函数,使用两条 push 指令进行参数的传递。这里所传参数为 a 和 b,所以将 a 和 b 分别入栈,此时栈的状态如图 3-31(b)所示。

传完参数之后执行指令“call Lget_min”,开始对 get_min()函数进行调用。

执行 call 指令之后,依然需要进行两步操作,第一步是将返回地址 PC 值压入栈中,第

二步是跳转到子函数开始执行，此时栈的状态如图 3-32(a)所示。在子函数开始时，依然需要进行三步操作：①将主函数栈帧的 FP 存储起来；②将 SP 的值赋给 FP，作为子函数栈帧的 FP；③假设子函数的局部变量需要 n 个空间，则将 SP 上移 n 个位置。

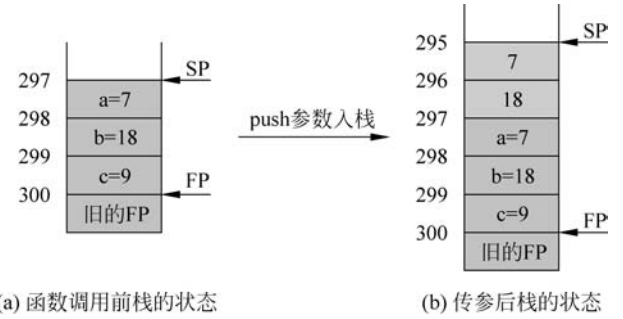


图 3-31　执行 call 指令前栈的状态

先将主函数的 FP 压入栈中，此时栈的状态如图 3-32(b)所示；再将 SP 的值赋给 FP，即将 FP 上移到 SP 所指的位置，此时栈的状态如图 3-32(c)所示；在本例中子函数没有局部变量，所以 SP 不需要上移进行空间预留，因此栈的状态保持不变。

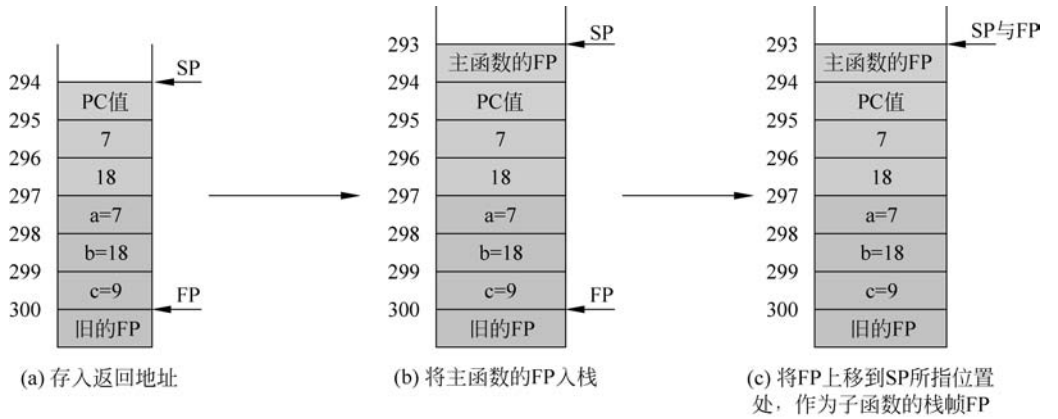


图 3-32　执行 call 指令后栈的状态

紧接着执行三条 push 指令，分别将 R2、R3、R4 压入栈中，以确保数据的干净与安全，此时栈的状态如图 3-33 所示。参数 x 的地址是 2(R15)，参数 y 的地址是 3(R15)，比较后较小的值由 R1 返回。

接下来将要返回时需要执行三条指令。指令“mov sp，R15”把 FP 值赋给 SP，即令 SP 下移，如图 3-34(a)所示；指令“pop R15”返回主函数的 FP 值，如图 3-34(b)所示；指令 ret 相当于“pop pc”，也就是返回到主函数调用子函数的下一条指令，在本例中是第二次调用子函数时进行传参的指令，如图 3-34(c)所示。

这里假设主函数不将原来的两个参数 a 和 b 弹出，这样并不会影响程序的正确性。接下来将两个新的参数 c 和 R1 依次压入栈中，如图 3-35 所示，然后再执行“call Lget_min”，栈帧的建立方式如前所述，这里不再赘述。最后得到的最小值由 R1 返回。

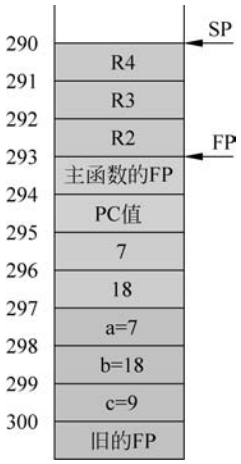


图 3-33　push 需要保护的寄存器的值

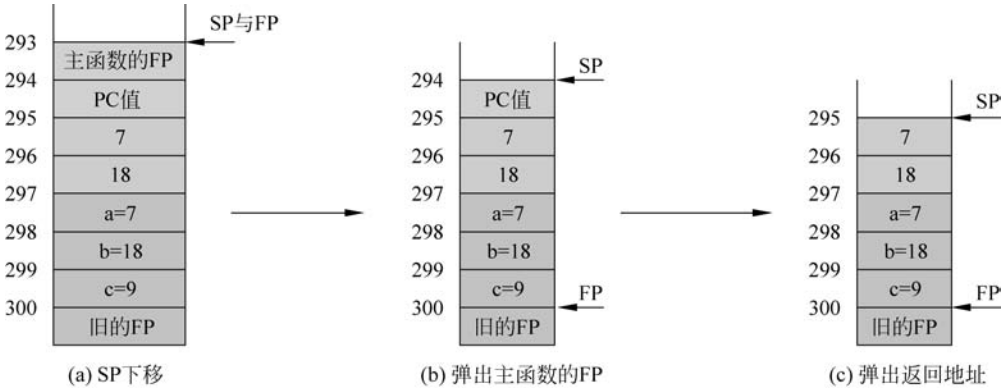


图 3-34　函数返回时栈的状态

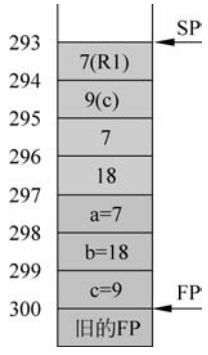


图 3-35　第二次函数调用传参后栈的状态

经 验 谈

由本例可知，只要遵循函数连接的标准规则，一个函数可以被多次调用而依旧能正确地执行。同学们也可以尝试如何依次建立和返回递归函数的栈帧。但是需要注意，在程序执行前，栈中要保留足够的空间，使得每个函数在调用时能够有足够的空间来建立它的栈帧，尤其对于递归函数而言，栈空间的大小更为重要。

练习题 3.5.1：将<程序：因数分解>中的 break 改为 return，factors(18)会输出什么结果？用 Python 运行试试看。

练习题 3.5.2：将<程序：因数分解>中的 if 块改写成如下程序，factors(18)的输出结果是什么？用 Python 运行试试看。

```
if (x % i == 0):
    print("Factor:",i)
    x = x//i
    factors(x)
    break
```

练习题 3.5.3：请阅读 SEAL 模拟器的使用文档，练习其中的范例。

练习题 3.5.4：请描述建立函数栈帧时一开始的三个指令的功能分别是什么。

练习题 3.5.5：假设 main 函数调用子函数 f，子函数 f 结束后需要返回 main 函数，请问 SP 和 FP 如何变化才能实现返回主调用函数的栈帧？

练习题 3.5.6：请参照 SEAL 的函数调用、栈帧建立方式，完成如下计算：输入三个数值，返回中间大小的值。请实现函数 median(a,b,c)，其中 a、b、c 是整数，返回的值存储在 R1 中。

练习题 3.5.7：按照 SEAL 的栈帧建立方式，函数内部的局部变量的地址是如何获得的？例如，函数 f 中有 a、b、c 三个局部变量，变量定义的顺序为 a、b、c(即存储 a 的地址最小)，请问 a 的地址在 SEAL 中是多少(用 FP 来描述)？

练习题 3.5.8：有递归函数 $f(n)=f(n-1)+n$，$f(0)=0$。假设每次调用函数 f 时，会产生 k 字节的栈帧，现在主函数调用 f(100)，请问在此计算过程中共产生多少字节的栈帧？

小结

本小节讨论计算机在执行函数调用时需要存储的信息(返回地址、局部变量)，以及如何用栈管理这些信息。通过解决这些问题，我们进一步清楚了执行函数调用的过程。

3.6 几种通用的编程语言

语言是工具，是用来沟通的工具。沟通的内容重要，工具也是重要，否则无法准确地沟通内容。所谓“工欲善其事，必先利其器”。据了解现在人类社会有 5000 多种语言，和计算

机相关的语言也非常多,有些是历久弥新,有些是老态龙钟,有些是渐渐消失,有些是异军突起,不一而足。计算机相关的语言可以分成通用型的语言和专用型的语言。专用型的语言是为了某种特殊用途而使用的语言。例如,在设计硬件时,现在工业界都会使用 VHDL 或 Verilog 这类语言,计算机专业学生将来上"数字逻辑电路"课程时会用到,也就是设计逻辑电路就如同编写程序般简单。在设计数据库时,最通用的是 SQL 语言。在设计网页时,HTML、JavaScript、PHP、ASP 等语言常会使用。在设计并行程序给多核系统执行时,MPI、openMP 等语言(或语言库)常被使用。而通用型的语言也是非常多,如 C、C++、Java、Python、Ruby、Smalltalk、Objective-C、C#、Basic、Perl、Delphi、Ada、Lisp、ML、Fortran、COBOL 等。

我们先来看一下 TIOBE 2020 年 4 月发布的编程语言排行榜(如表 3-1 所示)。

表 3-1 TIOBE 编程语言排行榜

Apr 2020	Apr 2019	Change	Programming Language	Ratings/%
1	1		Java	16.73
2	2		C	16.72
3	4	^	Python	9.31
4	3	v	C++	6.78
5	6	^	C#	4.74
6	5	v	Visual Basic	4.72
7	7		JavaScript	2.38
8	9	^	PHP	2.37
9	8	v	SQL	2.17
10	16	^^	R	1.54
11	19	^^	Swift	1.52
12	18	^^	Go	1.36
13	13		Ruby	1.25
14	10	vv	Assembly language	1.16
15	22	^^	PL/SQL	1.05
16	14	v	Perl	0.97

TIOBE 排行榜能显示当下最热门、使用最多的编程语言。在本节中,我们将简单介绍 C、C++、Java 这几种编程语言。

小明:沙老师,我想谈谈英文这个语言。我是中国人,堂堂正正的中国人,我为什么要学英文?

沙老师:你是中国人和你学不学英文有关系吗?大部分的现代知识都是用英文撰写的,我们学了英文这个工具,才能第一手地接触到这些知识。更现实的原因是这个世界越来越小了,你要与世界交流就必须要学好英文。我语重心长地说,你把中文和英文学好,你这一生前途光明。不要等你吃亏后,你才想到沙老师说的话。

每一种语言都有相应的编译器,只有在相应的编译器环境下,程序员才能编写相应的程

序并运行。

程序是为了实现某一种功能。在本书中,我们介绍的程序功能都是最基本的测试效果,并不是这个编程语言的应用。实现输出“Hello,world!”的功能只是我们学习这门编程语言的基础,这类功能与真正的应用开发相差很远。同学们如果想深入某一种语言,还需要自己不断参与实际的应用开发,才能更好地理解这门语言的精髓。

1. C 语言

C 语言于 1972 年由美国贝尔实验室的 D. M. Ritchie 开发成功。C 语言最初只是作为编写 UNIX 操作系统的一种工具,只在贝尔实验室内部使用。经过后来的不断改进,功能更丰富,应用也更广泛。到 20 世纪 80 年代,C 语言已经风靡全世界,大多数系统软件和许多应用软件都是用 C 语言编写的。

提到语言,必须要讲的一个概念就是结构化编程语言或叫作面向过程语言(Procedure Oriented Programming)和面向对象的编程语言(Object Oriented Programming)的区别。C 语言就是典型的结构化编程语言,二者的区别需要我们认真学习了不同的语言之后,加以比较才能真正领会。通俗地讲,面向过程的编程侧重设计一步步的“过程”来解决一个事件;而面向对象的编程侧重描述一个对象,且描述这个对象的代码可以被多次使用。

例如,我们要计算一个砖头的体积。在面向过程的 C 语言里面,我们就需要输入长、宽、高这三个数据,相乘之后输出来结果。这个计算乘积的函数与砖头的关联并不明显。在面向对象的编程中,我们可以定义一个变量形态叫作砖头“类”,这个砖头除了有长宽高这些数据外,还有计算体积(volumn())、表面积(surface())等的函数(这些函数叫作“方法”——Method),这些函数是属于这个类的。在程序里可以方便地宣告任何变量为砖头类(例如变量 x),这个变量 x 叫作一个对象(Object)。这个变量 x 不仅代表了数据,同时也包含了所有和砖头相关的方法。我们要计算 x 的体积,就用 x. volumn()来计算。这只是面向过程和面向对象编程中较小的一个差别,即封装的特征,还有面向对象编程中继承和多态这两个特征,也需要我们真正使用这种语言之后才能理解清楚。

最早的面向对象程序设计语言就是 C++语言。C++是由 AT&T 公司贝尔实验室的 Bjarne Stroustrup 博士及其同事于 20 世纪 80 年代初在 C 语言的基础上开发成功的。C++保留了 C 语言原有的所有优点,增加了面向对象的机制。

当然,面向过程和面向对象并不是相互对立的,而是相互补充的。C++也可以用来进行面向过程的编程。例如在面向对象编程中,对象的方法需要使用面向过程的思想来编写。

【例 3-8】 最小的 C 程序,只执行一个标准输出。

```
#<程序:C 中的输出>
#include<stdio.h>
void main(){
    printf("%s","hello world.");
}
```

stdio.h 头文件包含了 C 标准输入输出库函数相关的定义和声明,所有需要输入或输出的 C 程序都需要使用这个头文件。main 是主函数,即程序的入口点,大括号“{…}”表示

main 的函数体。printf 是标准输出函数,其参数分别表示输出格式和输出语句。“%s”表示将输出一个字符串,而“hello world.”是将要输出的字符串。

接下来我们看 C 语言如何实现这个简单 Python 程序:


```
#<程序:Python 数组连起来>
mx = [1,2,3]
my = [8,9]
print(mx + my)              #输出是[1,2,3,8,9]
```

以下是 C 语言的实现。读者有个感觉就好,我们不加解释。

```
#include <stdio.h>
#include <malloc.h>
void main(){
    int mx[3] = {1,2,3};
    int my[2] = {8,9};
    int i,j;
    int * x = (int *)malloc(sizeof(int) * (3 + 2));      //动态产生一个新数组 x,长度是 5
    for(i = 0;i < 3;i++){                                //i 从 0 到 2
        x[i] = mx[i];
    }
    for(j = 0;j < 2;j++){                                //i 从 0 到 1
        x[i + j] = my[j];
    }
    for(i = 0;i < 5;i++){                                //i 从 0 到 4
        printf("%d",x[i]);
    }
    printf("\n");
}
```

2. C++

C++是目前使用最广泛的面向对象程序设计语言。实际上,C++同时支持面向过程的程序设计和面向对象的程序设计。如前所述,C 到 C++的演进是由 Bjarne Stroustrup 博士完成的。他在 C 语言的基础上增加了类的概念,包括类的访问属性、构造方法等。

小明:沙老师,我还是想问问怎么学好英文。我真的很用功,我甚至背英文字典。你说我用不用功?

沙老师:傻孩子!字典是用来查的,不是用来背的。看小说学英文是对的。背字典学英文?恰好背道而驰。我曾经写了一篇文章叫作《学好英文的秘诀》,在网上或许可以找到。这个秘诀就是“不要学”,通俗地讲就是不要用“逻辑”“思维”来学语言,要浑然天成,要自然。唯一的方法就是多读、多讲、多写。你们学编程语言也是如此,要多写!

C++提供两种定义类型的构造,即类和结构体。结构体的概念与 C 语言中的相似。C++与 C 语言的关系很密切,熟悉 C 语言的人也可以很快掌握 C++。

【例 3-9】 最小的 C++程序，只执行一个标准输出。

```
#<程序：C++中的输出>
#include <iostream>
int main(){
    std::cout <<"hello world.\n";
}
```

这个函数实现输出“hello world.”到屏幕上。

程序的 iostream 提供了输入输出流设施，任何需要有输入或输出的 C++程序都需要包含这个头文件。程序入口点则是 int main()，main 就是函数名，大括号“{…}”表示 main 的函数体。后花括号“}”就是程序结束处。std 是“名空间”，cout 是标准输出设备的名称，“<<”是操作命令，表示将其后的字符串输出到屏幕上。“std::cout”表示是开发环境提供的标准库中的 cout，而不是程序员自己定义的 cout。

以下是将两个数组连起来的 C++程序。读者有个感觉就好，我们不加解释。

```
#include <iostream>
#include <vector>              //vector 是 C++已经有的类模板，比较好用，有点像 Python 的 list
using namespace std;

int main(){
    int mx[3] = {1,2,3};
    int my[2] = {8,9};
    vector <int> x(mx,mx + 3);                          //将 mx 复制到 x 里面
    for(int i = 0;i < 2;i++){
        x.push_back(my[i]);                             //将 my 依序压入 x 的末尾
    }
    for(vector <int>::iterator it = x.begin();it!= x.end();it++){      //将 x 从开始依次输出
        cout << * it <<" ";
    }
    cout << endl;
    return 0;
}
```

3. Java 语言

Java 起源于 20 世纪 90 年代初。在 Sun 公司的 Green 项目中，项目小组成员使用 C++开发系统时遇到了很多问题，另辟蹊径，开发了这个小型的计算机语言。相对于 C++，这款语言提供了更好的简单性和可靠性。最初它被命名为 Oak，即橡树。1995 年，这款语言正式更名为 Java。

Java 是印度尼西亚爪哇岛的英文名称，因盛产咖啡而闻名。Java 语言的标志就是一杯正冒着热气的咖啡，而且 Java 语言中的许多类库名称也与咖啡有关，如 JavaBeans(咖啡豆)、NetBeans(网络豆)、ObjectBeans(对象豆)等。

对于 Java，我们需要知道它有 3 个开发平台，即 JavaSE(Java2 Platform Standard Edition，Java 平台标准版)、JavaEE(Java2 Platform Enterprise Edition，Java 平台企业版)、JavaME(Java2 Platform Micro Edition，Java 平台微型版)。开发平台，可以简单地理解为开发应用软件时使用到的一系列的工具(所说的工具涉及接口、库等概念，暂时不做详细介

绍)。这三种应用平台针对不同的开发需求,如 JavaME 主要是为在移动设备和嵌入式设备(如手机、电视机顶盒和打印机)上运行的应用程序提供一个健壮、灵活的环境。

关于开发环境 Eclipse、Myeclipse,以及 Java Web 应用的 Web 服务器——Tomcat 等,在此也不做详细介绍。但是我们要知道,Java 语言既可以编写应用程序(即在自己的计算机上独立运行,像 C 语言一样),也可以编写小程序(Applet),存储在服务器上并由浏览器运行,即 Web 开发。

不同于 C++语言,Java 是纯面向对象的,程序都是由类组成的。

【例 3-10】 最小的 Java 程序,只执行一个标准输出。

```
#<程序: Java 中的输出>
public class doOut{
    Public static void main(String[] args){
        System.out.println("hello world.");
    }
}
```

System.out.println 是标准输出函数,且输出语句后换行。输出语句中不限制输出格式,Java 对于所有的输出都作为一个字符串来原样输出。

接下来是 Java 实现数组连起来的程序。读者有个感觉就好,我们不加解释。

```
import java.util.Vector;
public class MergeClass{
  public static void main(String[] args){
      int mx[] = {1,2,3} ;
      int my[] = {8,9};
      int len_y = my.length;
      Vector x = new Vector();                          //x 是个 Vector 对象(object)
      for(int i = 0;i < mx.length;i++){
          x.add(mx[i]);                                 // 加入(append)mx 的值到 x
      }
      for(int i = 0;i < my.length;i++){
          x.add(my[i]);                                 // 加入(append)my 的值到 x
      }
      for(int index = 0;index < x.size();index++){
         System.out.print(x.elementAt(index) + " ");
      }
      System.out.print("\n");
  }
}
```

小结

本小节,我们介绍了 C、C++、Java 语言的起源、特点等。程序语言的学习过程是相通的,学习了一门语言之后,再学习其他语言就变得非常容易。每一门语言都有其独到之处。同学们在今后的实际演练中,会更加深刻地意识到,其实并不存在所谓的“最好的程序语言”。

3.7 对计算机程序的领悟

程序的英文是 Program，程序语言是 Programming Language，而算法的英文是 Algorithms。语言、程序和算法是三位一体的。语言是工具，算法是解题的想法，而程序是用某种语言来实现算法的技术。本章主要是谈程序和计算机语言。计算机有了计算机语言以后，才有了程序，才有了多彩多姿的生命，就像是人类有了语言和文字后才有了蓬勃的文明发展。人类的语言文字在描述如何解决问题时，是不清楚的，是有缺陷的。程序是人类文明中第一次有方法能清楚地描述解决问题的步骤，这是个伟大的进步。

在第 5 章我们会有一个有趣的例子讲如何解决走迷宫的问题。一个复杂的迷宫，可以想象是由 $n \times n$ 的方格所组成，有些方格是面墙，有些方格是通路，如何让你朋友从起点走到终点？你和你朋友都不知道迷宫内的组合情形，你的朋友只有走进去后，依照当时的情形来决定如何走下去。请问你要如何向你的朋友来描述他应该依循的解决方案，使得他能遵循你的方案来走过任何复杂的迷宫？大家当作一个练习题试试看，是不是用人类语言很难去描述你们心中的解法？但是计算机语言就清楚了，例如第 5 章用 Python 语言来解迷宫问题，短短的一段程序就能清楚无误地描述应该遵循的方法，你的朋友只要遵循 Python 程序描述的方法，他就能走过任何复杂的迷宫。

程序和计算机语言具备了清晰的语义、严谨的逻辑、巧妙的结构，这是本节所要谈的领悟。另外，我们谈谈智能和程序的关系。人工智能的英文为 artificial intelligence，也就是人所造的智能。大家不要把智能看得太神秘了。有一个电视节目叫作“最强大脑”，展示出人类似乎匪夷所思的智能来，例如念给你 100 个任意数字，你先按从头到尾的顺序念出来，再从尾到头地念出来，我不相信你能做得到，太难了！又如给你两面墙，第一面墙是 1000 个魔术方块所拼构而成的，第二面墙是第一面墙的翻版，除了有一个魔术方块是不一样的，其他的方块都完全一样，请你在 10 分钟内找到这个不一样的魔术方块。你行吗？（你认为计算机要用多久找到这个不同的方块，1 秒内吧！）再举一个例子，在国际象棋和围棋的比赛中，计算机的表现已经超过人类最顶尖的棋手了。这些智能不神秘，都是程序所表现出来的智能。人工智能就是程序所计算出来的罢了。本节用一个例子来展现“智能”不过就是程序计算出来的罢了！

3.7.1 清晰的语义

首先，计算机语言必须要非常清晰明了。我们在生活中互相交流、传达信息，需要借助语言，但是生活中的语言往往表达得不够准确。比如当有人问路时，我们可能会说：“超市再往前走一段路，一会儿就到了。”这里的“一段”和“一会儿”所传达出的信息就不够明确，可能是 1 分钟的路程，也可能是 5 分钟的路程。英文中也存在语言描述模糊的现象。英文的 slim 表示“瘦”，fat 表示“胖”。按照我们的思维习惯，会认为 slim chance 是“机会小”的意思，而 fat chance 是“机会大”的意思。但事实上，slim chance 和 fat chance 意思完全相同，都表示“机会小”。同样地，我们与计算机通信，也需要有计算机语言。但是，计算机可不像我们人脑一样灵活。为了能够清楚地将我们的意思传达给它，同时从它那里得到正确的反馈信息，计算机语言不能是模棱两可的，更不能具有歧义，必须清晰准确。因此，在计算机语

言中,我们的思想是用清楚的、无二义性的方式来描述的。这种清晰明了的语言形式,使得计算机语言有一种不同于其他语言的明了之美。有些人会问,清楚明了有什么美的?在男女朋友交往时,如果一方说话也像计算机语言一样明确清晰,也许很多时候另外一方就不会这么苦恼了。

3.7.2 严谨的逻辑

除了语义清晰,计算机语言具有严谨但不乏灵活的逻辑之美。众所周知,数学的逻辑非常严谨。计算机语言的逻辑亦是如此,用计算机语言来解决问题时,我们的解题思路是非常清晰的,并且可以用完全逻辑性的形式语言来描述。因此,在解决某些问题时,计算机语言有很大的优势。比如将一串数字 2、8、4、12、5 按递增的方式进行排序,通用的数学模型只是讲"what"——什么是递增序列,而没有讲"how"——如何转变一个非递增序列为递增序列。与数学逻辑相比,计算机语言可以清楚地描述"how"。因为它有循环语句,有条件控制流程等。语言形式严谨,兼具结构组合灵活,这样的语言怎能不美!

3.7.3 巧妙的结构

说完了计算机程序在语言风格和逻辑上的美,现在我们来看看它在结构上的美。计算机程序在结构上有一个非常有趣的特点,那就是采用了函数的调用,这使得计算机程序具有一种精巧的美。如第 2 章所言,简单的开关构建了复杂的计算机硬件系统。同理,一个复杂的软件也是由许多简单的函数一层一层调用而形成的,层次结构非常清晰。比如网络系统以及 Linux、Windows 等操作系统皆是如此。

一个大系统中,一个程序可能有上百万行代码、上万个函数调用,是上千人合作数年完成的。如果仅仅因为一个人改一个函数中的一行代码,难道就要牵一发而动全身,所有的函数都需要更改吗?

遇到这种问题时,函数调用的巧妙就发挥得淋漓尽致了。

计算机程序中,函数的实现千变万化。函数调用中,即使函数的实现改变了,只要函数的调用方式不变,调用它的程序就无须做任何改变。如图 3-36 所示,factors()函数调用 sqrt(n)函数。在 sqrt(n)函数中,即使其中的程序改变了,只要 sqrt()还是正确的和对 sqrt(n)的调用方式不变,我们完全可以按照原来的方式继续调用,factors()函数无须做任何改变。

这种函数调用的结构,使得程序的主函数精巧、明了,使得程序的修改更加容易,程序的结构变得具有一种排列紧凑、疏密得当的美感。

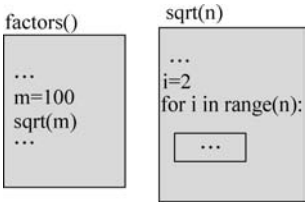


图 3-36 函数调用示例

一段程序其实也可以折射出人生的许多道理和启示。如果将程序比作人生,那么里面的每一个语句都是我们人生路上不可或缺的经历。那些看似简单的指令在 CPU 中有条不紊地一步一步执行,最后能够让计算机完成很多复杂的工作。由此可以看到,我们在做事的时候,不可以因为它简单而忽视它,应该脚踏实地地做好每一件事,一件一件的小事做好了,才能完成最终的目标。做人也是同样的道理,一步一个脚印,脚踏实地地走下去,才会守得云开见月明。

3.7.4 智能是程序计算出来的

计算机被广泛应用于日常生活，我们可以用计算机搜索想要知道的信息，可以用计算机求解数学问题。那么计算机是怎么做到这些的呢？难道计算机也是有智能的，也可以像人一样思考吗？在没有学过计算机科学的人看来，计算机真是太可怕了，它能做很多人都做不来的事，将来会不会有一天就像很多美国科幻电影里演的，人类被计算机所取代？

下面我们通过一个简单的游戏来看看计算机的智能是什么样的。

这个游戏叫猜数字，由两个人都各自选定一个秘密的三位数（也可以是四位数或更多位数），然后相互猜对方的数字。用几个 A 来表示对方猜的三位数中有几个数是完全正确的，用几个 B 来表示有几个数正确但是位置不对。对于重复的数字不可以重复计算，看谁先猜到对方的数字。

比如：我选了一个秘密数字 732。

对方猜：057。

我回答：0 A 1 B（1B 是因为 7 在我的数字里，但是它的位置不对）。

对方猜：582。

我回答：1 A 0 B（A 是因为 2 是完全正确的）。

对方猜：563。

我回答：0 A 1 B。

对方猜：672。

我回答：1 A 1 B。

对方猜：732。

我回答：3 A 0 B（答对了）。

游戏的规则如上面所述，有兴趣的同学可以做一下这个游戏，看看谁能先猜出来，能用几步猜出来。

可能有人会觉得这完全是靠运气嘛！其实不完全是靠运气，这里面可是有技巧的，仔细想想你是怎么猜的呢？

一个简单的方法如下。

首先随便猜数字，直到出现不是“0A0B”的时候，后面的数字就不完全是随便猜的了。比如猜“057”之后，对方回答“0A1B”，那么下面猜测的数字要和“057”是“0A1B”的关系，因为正确答案一定就在这些数字里。

“582”与“057”是“0A1B”的关系，猜测“582”之后，对方回答“1A0B”，下面猜测的数字要与“057”是“0A1B”的关系，而与“582”是“1A0B”的关系。

“563”与“057”是“0A1B”的关系，而与“582”是“1A0B”的关系。猜测“563”之后，对方回答“0A1B”。下面要猜的数字要与“057”和“563”是“0A1B”的关系，而与“582”是“1A0B”关系。

可以猜“672”，对方回答“1A1B”。然后找到与“057”和“563”是“0A1B”的关系，和“582”是“1A0B”关系，和“672”是“1A1B”的数字。进而找到“732”，得到 3A0B。

上述方法就是一个能够解决猜数字问题的计算思维。那么怎么将它应用于计算机呢？根据上述计算思维，计算机对猜数字问题的“思维”如图 3-37 所示。

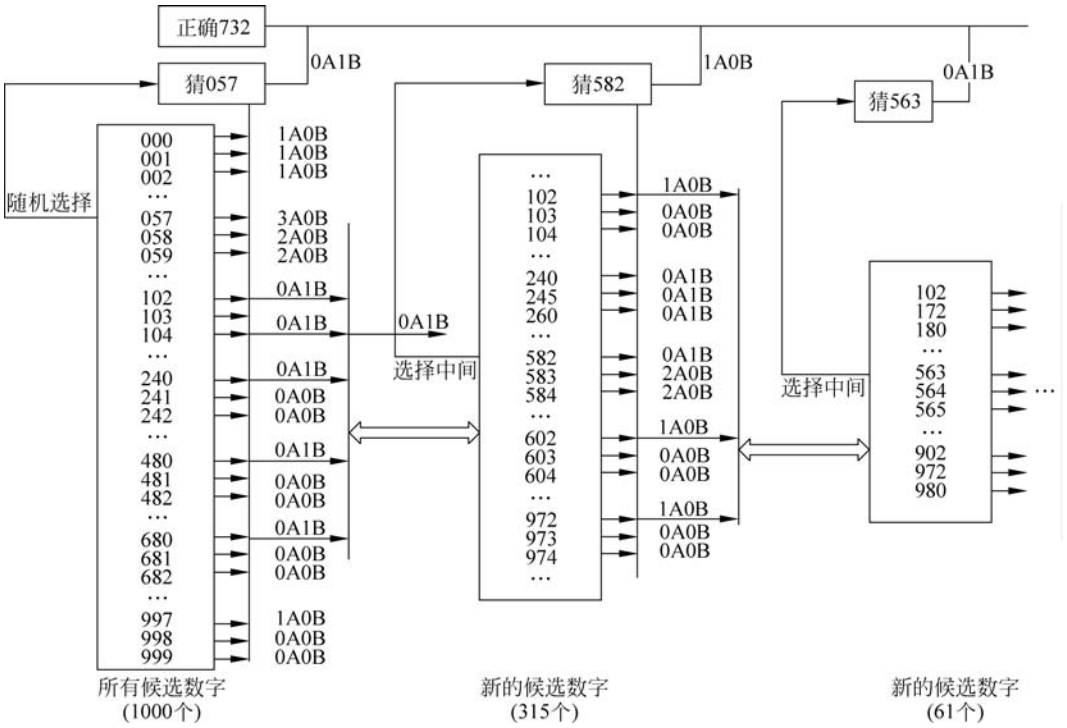


图 3-37　计算机解决猜数字问题的“思维”图

首先,计算机会将所有可能的三位数(000～999)全部列举出来。在这 1000 个三位数中,随机选择“057”进行猜测。

根据“057”给出的结果“0A1B”,对列举出来的 000～999 进行筛选。符合与“057”形成“0A1B”关系的数字被筛选出来,共有 315 个候选数字。从这些候选数字中,选择排在中间的数字“582”进行猜测。

根据“582”给出的结果“1A0B”,对上次的 315 个候选数字进行筛选。符合与“582”形成“1A0B”关系的数字被筛选出来,共有 61 个候选数字。继续从这些候选数字中,选择排在中间的数字“563”进行猜测。

根据“563”给出的结果“0A1B”,对上次的 61 个候选数字进行筛选。符合与“563”形成“1A0B”关系的数字被筛选出来。依次类推,直至猜到正确数字“732”为止。

应用上述的计算思维,计算机利用它强大的计算能力,能够在非常短的时间内得到正确的结果。

当然,另一方面,我们也应该谨记,我们是创造程序的人,千万不要变成 CPU,变成机器人——只是根据指示做事。要多思考,有创新意识,做一个有思想的人。别忘了,计算机程序是我们所写出来的。

其实,智能还是神秘难测的,或许应该叫作智慧吧。深层的智慧有哲学层面的意义。我们前面所讲的智能是计算机程序能展现出的“人工智能”。那么是否还有人工智能外的智慧呢?也就是计算机无法展现出来的智慧呢?科学家可以证明计算机不能解决所有的问题(或计算机不能证明出所有的定理),有些问题是计算机解决不了的,例如 Halting Problem

(停机问题)。这个问题是输入一个程序,假如这个程序被断定总是能停止就输出 yes,假如被断定有可能永不停止就输出 no。科学家证明世界上没有任何程序能 100%解决这个问题。其实这个证明很短,通过构筑自相矛盾的递归来证明,几句话罢了(第 10 章会给出完整的证明过程),然而在哲学层面上的意义是很深的。

在哲学层面的"智慧"是不一样的,例如佛教中的智慧被称为"般若",和俗称的"智慧"有所区别。佛教中认为真正的般若智慧是离开文字、离开语言、离开分别、离开思维,甚至离开智慧,无可言说后的"所得",然而所得也不可所得,一旦"有所得"后就不是般若了,所以也是"无所得"。所谓"色即是空,空即是色,色不异空,空不异色,无智亦无得,以无所得故"。大家看看就是,有个基本认识就好,知道程序所显现的人工智能和哲学层面上的智慧是有差异的。

练习题 3.7.1:拿牌游戏。假设面前有三堆扑克牌,其中每堆各有 10 张牌。两个人交替从某一堆中拿牌,谁拿到最后一张牌谁就输,请找出所有必赢的拿牌方式。

练习题 3.7.2:用 Python 实现猜数字游戏。在这个作业里不考虑有重复数字的 3 位数(例如 335)。

小结

本节我们对计算机是否具有智能给出了解答。其实,计算机的智能是计算出来的。我们以猜数字的游戏为例,向大家展示了计算机的"思路"。这种"思路"其实是计算机程序的编写者赋予的。计算机应用人类赋予的计算思维和其强大的计算能力,可以又快又准确地解决很多问题。

习题 3

习题 3.1:假设寄存器 R1 中存储的数值为 10,执行完下面两条指令后,寄存器 R2 中存储的值是什么?

```
mov R2, R1
add R2, R2, 10
```

习题 3.2:假设寄存器 R1 中存储的数值为 20,执行完下面两条指令后,主存地址 800 处存储的值是什么?

```
add R2, R1, 30
store (800), R2
```

习题 3.3:假设寄存器 R1、R2 中的值分别为 10 和 15,执行完下面这段汇编指令后,寄存器 R2 中存储的值是什么?

```
slt R1, R2, R1
beqz R1,label0
mov R2, R1
label0:
    add R2, R1, 10
```

习题 3.4:假设变量 a、b、c 分别读取到寄存器 R1、R2、R3 中,请写出下面这段程序对应

的汇编指令。

```
if a < b
    c = a + b
else
    c = b
```

习题 3.5：在习题 3.4 中，修改程序的第一条语句为“if a＜＝b”，请写出修改后的程序对应的汇编指令。

习题 3.6：假设变量 a、b、c 分别读取到寄存器 R1、R2、R3 中，请写出下面这段程序对应的汇编指令。

```
if a < b
    c = a + b
else
    c = b
while c < 10
    c = c + 10
```

习题 3.7：有如下汇编代码：

```
mov R1,02h
_____________        将寄存器 R1 中的值左移 1 位后存入寄存器 R2 中
_____________        将寄存器 R2 中的值左移 2 位后存入寄存器 R3 中
add R4,R3,R2
```

(1) 根据旁边的注释，写出对应的汇编指令。

(2) 这 4 条指令执行结束后，各寄存器中的值为多少？

(3) 说明这段汇编代码完成的功能。

习题 3.8：假设变量 a、b 分别被读取到寄存器 R1、R2 中，请写出下面这段汇编指令完成了什么功能。

```
loop:
    slt R4,R1,0Ah
    beqz R4,label0
    add R1,R1,R2
    goto loop
label0:
    add R2,R2,01h
```

习题 3.9：假设寄存器 R1、R2 中的值分别为 20 和 30，执行完下面这段汇编指令后，主存中地址 1000 处存储的值是什么？

```
loop:
    slt R4,R2, R1
    beqz R4,label0
    add R1, R1, 15
    goto loop
label0:
    store (1000), R1
```

习题 3.10：假设变量 i、a、b 分别读取到寄存器 R1、R2、R3 中，分析下面这段汇编指令。

```
loop:
      slt R4,R1, 0Ah
      beqz R4,label0
      add R2, R2, R3
      sub R3, R3, 01h
      add R1, R1, 01h
      goto loop
label0:
```

(1) 说明这段汇编指令执行的功能。

(2) 假设变量 a 和 b 的值分别为 10 和 20，这段汇编指令执行完成后，寄存器 R2、R3 中的内容分别是多少？

习题 3.11：假设变量 a、b 分别存储在主存地址 1000、1008 处，现在要执行 a 右移 b 位的操作，并把结果存回地址 1024 处，写出相应的汇编指令。

习题 3.12：请写出下面程序中的局部变量、全局变量，并写出程序的运行结果。

```
a = 10
b = 30
def  func():
     global  a
     a = b
     print(a)
func()
print(a)
```

习题 3.13：在习题 3.12 的程序中，func()函数中去掉“global a”语句，程序的运行结果会有什么变化吗？

习题 3.14：请写出下面程序的运行结果。

```
a = 10
b = 30
def  func():
     global a
     a = a + b
     return a
b = func()
print(a,b)
```

习题 3.15：将习题 3.14 的程序稍做修改，请写出下面程序的运行结果。

```
a = 10
b = 30
def  func(a,b):
     a = a + b
     return a
b = func(a,b)
print(a,b)
```

习题 3.16：请写出下面程序输出的结果。

```
def  func(b):
     a = b + 10
     print(b)
     b = 15
     print(a,b)
func(20)
```

习题 3.17：结合栈的特点，讲一讲在进行函数调用时，为什么要用栈来保存调用函数的信息？

习题 3.18：请写出下面递归函数的输出结果。

```
def  func(a):
     if a == 1:
         return 1
     return a * func(a - 1)
b = func(5)
print(b)
```

习题 3.19：给定如下 Python 程序：

```
def  do_sub(y):
     z = 4
     z = y - z
     return z
x = do_sub(13)
```

(1) 画出调用 do_sub()函数后的栈帧示意图。

(2) 画出返回后的栈帧示意图。

习题 3.20：给定如下 Python 程序：

```
x = 3
y = 4
def  func():
     global  x
     x = y
     z = x * y
func()
```

画出调用 func()函数后的栈帧示意图。

习题 3.21：给定如下两个 Python 程序：

(1)
```
y = 5
def  func(z):
     global  x
     x = z - y
     print (x)
func(11)
```

(2)
```
def  func(z):
     y = 5
```

```
        x = z - y
        print (x)
    func(11)
```

(1) 以上两个程序分别会输出什么？

(2) 两个程序的栈帧中存储的数据相同吗？为什么？