一、调用惯例的差异

cdecl，stdcall，_fastcall 是三种函数调用协议，函数调用协议会影响函数参数的入栈方式、栈内数据的清除方式、编译器函数名的修饰规则等。

1.调用协议常用场合

- stdcall：Windows API 默认的函数调用协议
- cdecl：C/C++ 默认的函数调用协议
- fastcall：适用于对性能要求较高的场合

2.函数参数入栈方式

- stdcall：函数参数**由右向左**入栈
- cdecl：函数参数**由右向左**入栈
- fastcall：**从左开始不大于 4 字节的参数放入 CPU 的 ECX 和 EDX 寄存器**，其余参数**从右向左**入栈。（fastcall 调用协议在寄存器中放入不大于 4 字节的参数，故性能较高，适用于需要高性能的场合）

3.栈内数据清除方式

- stdcall：函数调用结束后由**被调用函数**清除栈内数据
- cdecl：函数调用结束后由**函数调用者**清除栈内数据
- fastcall：函数调用结束后由**被调用函数**清除栈内数据
- 注意：
    - 1、不同编译器设定的栈结构不尽相同，跨平台开发时由函数调用者清除栈内数据不可行
    - 2、某些函数的参数是可变的，如 printf 函数，这样的函数只能由函数的调用者清除栈内数据。

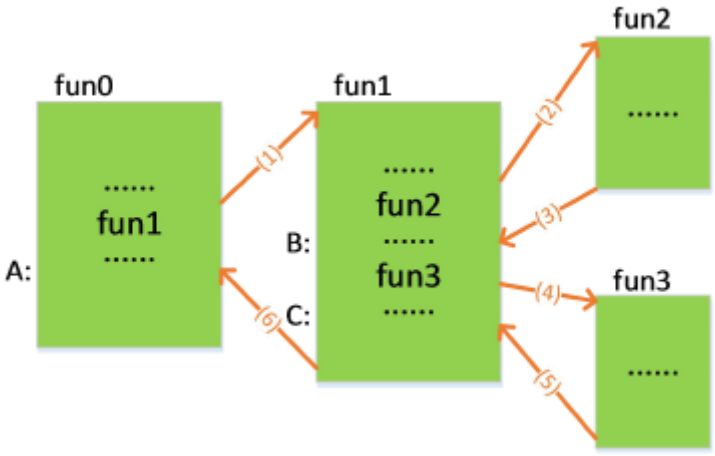
二、函数调用、返回时的栈状态的变化

1. 返回地址的存储

执行一条指令时，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，PC 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：

1. 调用函数
2. 函数调用后的返回
3. 控制结构（if else，while，for 等）

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，一个函数既调用别的函数又被另外的函数调用

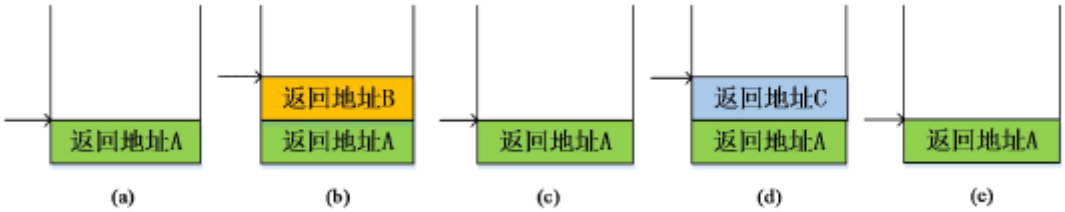


（上图中，fun0 函数调用 fun1，fun0 函数就是主调函数，fun1 是被调函数）

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”。

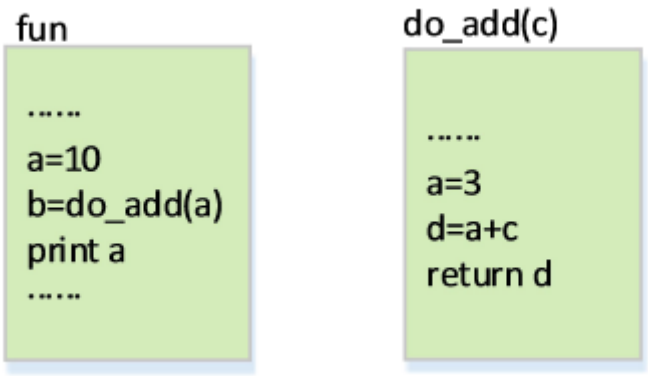
CPU 执行程序时，并不知道整个程序的执行步骤是怎样的，完全是“走一步，看一步”。前面提到过，CPU 都是根据 PC 中存放的指令地址找到要执行的语句。函数返回时，是“从哪里离开，就回到哪里”。但是当函数要从被调函数中返回时，PC 怎么知道调用时是从哪里离开的呢？答案就是——将函数的“返回地址”保存起来。因为在发生函数调用时的 PC 值是知道的。在主调函数中的函数调用的下一条语句的地址即为当前 PC 值加 1，也就是函数返回时需要的“返回地址”。我们只需将该返回地址保存起来，在被调函数执行完成后，要返回主调函数中时，将返回地址送到 PC。这样，程序就可以往下继续执行了。

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 函数先调用，但是返回的时候 fun1 晚于 fun2 返回。这一特点正是"后进先出"，所以采用栈来保存返回地址



如上图调用过程(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，栈的状态如图中(c)所示；调用过程过程(4)发生时，需要压入保存返回地址 C，栈的状态如图中(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

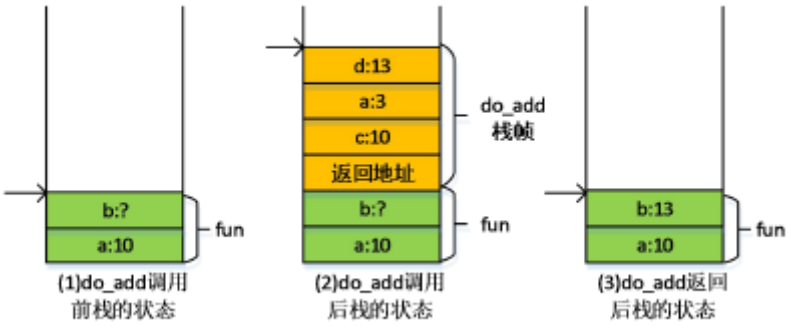
2. 函数调用时栈的管理



如上图所示，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，这两个变量需要存放在不同的地方。局部变量 a 只在 do_add 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。局部变量如同返回地址般也是存在栈里。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。

当函数调用时，do_add 函数里局部变量 c 就复制 fun 函数里变量 a 的值。在函数返回时，与参数传递同理，在传递返回值时也是将 do_add 函数里的值赋值给主调函数中的变量 b。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。

局部变量的调用是和栈的操作模式“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，同样的，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做此函数的栈帧(frame)。当此函数结束时，这一块栈帧就会被弹出。



调用 do_add() 函数前执行的操作:

1. fun 的局部变量 a 压入栈中，其值为 10
2. 局部变量 b 压入栈中，由于 b 的值还未知，因此先为 b 预留空间

调用 do_add() 函数时执行的操作:

1. 返回地址压到栈中
2. 局部变量 c 的值 10 压入栈中（c 的值是通过复制 fun 函数中变量 a 得到的）
3. 压入 do_add 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

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() 的局部变量，由于需要将 d 的值返回，因此在弹出 d 的时候需要一个寄存器将返回值 d 保存起来

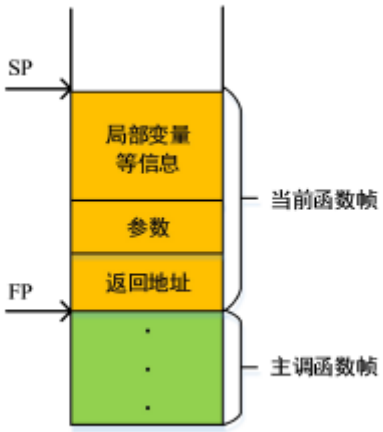
2. 弹出返回地址，将返回地址传到 PC
3. 返回到 fun 函数，fun 中的局部变量 b 的值即为 do_add() 中的返回值 d，此时将寄存器中的值赋给 b。

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。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。

由于函数调用时，要不断的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，而非栈指针 SP。

3. 综合，可以总结:

1. 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。
2. 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的从 FP 算出来。下图展示了栈帧的通用结构



三、不同优化级（O1-O3）对调用惯例的影响

- -O1: 目的是在不影响编译速度的前提下，尽量采用一些优化算法降低代码大小和可执行代码的运行速度。
- -O2: 会牺牲部分编译速度，除了执行 -O1 所执行的所有优化之外，还会采用几乎所有的目标配置支持的优化算法，用以提高目标代码的运行速度。
- -O3: 除了执行 -O2 所有的优化选项之外，一般都是采取很多向量化算法，提高代码的并行执行程度，利用现代 CPU 中的流水线，cache 等。