给出了运行时栈的通用结构，包括把它划分为栈帧。当前正在执行的过程的帧总是在栈顶。当过程 P 调用过程 Q 时，会把返回地址压入栈中，指明当 Q 返回时，要从 P 程序的哪个位置继续执行。我们把这个返回地址当做 P 的栈帧的一部分，因为它存放的是与 P 相关的状态。Q 的代码会扩展当前栈的边界，分配它的栈帧所需的空间。在这个空间中，它可以保存寄存器的值，分配局部变量空间，为它调用的过程设置参数。大多数过程的栈帧都是定长的，在过程的开始就分配好了。但是有些过程需要变长的帧，这个问题会在 3.10.5 节中讨论。通过寄存器，过程 P 可以传递最多 6 个整数值(也就是指针和整数)，但是如果 Q 需要更多的参数，P 可以在调用 Q 之前在自己的栈帧里存储好这些参数。

为了提高空间和时间效率，x86-64 过程只分配自己所需要的栈帧部分。例如，许多过程有 6 个或者更少的参数，那么所有的参数都可以通过寄存器传递。因此，图 3-25 中画出的某些栈帧部分可以省略。实际上，许多函数甚至根本不需要栈帧。当所有的局部变量都可以保存在寄存器中，而且该函数不会调用任何其他函数(有时称之为叶子过程，此时把过程调用看做树结构)时，就可以这样处理。例如，到目前为止我们仔细审视过的所有函数都不需要栈帧。

3.7.2　转移控制

将控制从函数 P 转移到函数 Q 只需要简单地把程序计数器(PC)设置为 Q 的代码的起始位置。不过，当稍后从 Q 返回的时候，处理器必须记录好它需要继续 P 的执行的代码位置。在 x86-64 机器中，这个信息是用指令 call Q 调用过程 Q 来记录的。该指令会把地址 A 压入栈中，并将 PC 设置为 Q 的起始地址。压入的地址 A 被称为返回地址，是紧跟在 call 指令后面的那条指令的地址。对应的指令 ret 会从栈中弹出地址 A，并把 PC 设置为 A。

下表给出的是 call 和 ret 指令的一般形式：

指令		描述
call	*Label*	过程调用
call	**Operand*	过程调用
ret		从过程调用中返回

(这些指令在程序 OBJDUMP 产生的反汇编输出中被称为 callq 和 retq。添加的后缀'q'只是为了强调这些是 x86-64 版本的调用和返回，而不是 IA32 的。在 x86-64 汇编代码中，这两种版本可以互换。)

call 指令有一个目标，即指明被调用过程起始的指令地址。同跳转一样，调用可以是直接的，也可以是间接的。在汇编代码中，直接调用的目标是一个标号，而间接调用的目标是 * 后面跟一个操作数指示符，使用的是图 3-3 中描述的格式之一。

图 3-26 说明了 3.2.2 节中介绍的 multstore 和 main 函数的 call 和 ret 指令的执行情况。下面是这两个函数的反汇编代码的节选：

```
     Beginning of function multstore
1    0000000000400540 <multstore>:
2      400540:  53                    push   %rbx
3      400541:  48 89 d3              mov    %rdx,%rbx
       . . .
     Return from function multstore
4      40054d:  c3                    retq
       . . .
```