虽然在给定时刻只有一个过程是活动的，我们仍然必须确保当一个过程(调用者)调用另一个过程(被调用者)时，被调用者不会覆盖调用者稍后会使用的寄存器值。为此，x86-64 采用了一组统一的寄存器使用惯例，所有的过程(包括程序库)都必须遵循。

根据惯例，寄存器%rbx、%rbp 和%r12～%r15 被划分为被调用者保存寄存器。当过程 P 调用过程 Q 时，Q 必须保存这些寄存器的值，保证它们的值在 Q 返回到 P 时与 Q 被调用时是一样的。过程 Q 保存一个寄存器的值不变，要么就是根本不去改变它，要么就是把原始值压入栈中，改变寄存器的值，然后在返回前从栈中弹出旧值。压入寄存器的值会在栈帧中创建标号为“保存的寄存器”的一部分，如图 3-25 中所示。有了这条惯例，P 的代码就能安全地把值存在被调用者保存寄存器中(当然，要先把之前的值保存到栈上)，调用 Q，然后继续使用寄存器中的值，不用担心值被破坏。

所有其他的寄存器，除了栈指针%rsp，都分类为调用者保存寄存器。这就意味着任何函数都能修改它们。可以这样来理解“调用者保存”这个名字：过程 P 在某个此类寄存器中有局部数据，然后调用过程 Q。因为 Q 可以随意修改这个寄存器，所以在调用之前首先保存好这个数据是 P(调用者)的责任。

来看一个例子，图 3-34a 中的函数 P。它两次调用 Q。在第一次调用中，必须保存 x 的值以备后面使用。类似地，在第二次调用中，也必须保存 Q(y)的值。图 3-34b 中，可以看到 GCC 生成的代码使用了两个被调用者保存寄存器：%rbp 保存 x 和%rbx 保存计算出来的

```
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

a) 调用函数

```
    long P(long x, long y)
    x in %rdi, y in %rsi
 1  P:
 2    pushq   %rbp              Save %rbp
 3    pushq   %rbx              Save %rbx
 4    subq    $8, %rsp          Align stack frame
 5    movq    %rdi, %rbp        Save x
 6    movq    %rsi, %rdi        Move y to first argument
 7    call    Q                 Call Q(y)
 8    movq    %rax, %rbx        Save result
 9    movq    %rbp, %rdi        Move x to first argument
10    call    Q                 Call Q(x)
11    addq    %rbx, %rax        Add saved Q(y) to Q(x)
12    addq    $8, %rsp          Deallocate last part of stack
13    popq    %rbx              Restore %rbx
14    popq    %rbp              Restore %rbp
15    ret
```

b) 调用函数生成的汇编代码

图 3-34　展示被调用者保存寄存器使用的代码。在第一次调用中，必须保存 x 的值，第二次调用中，必须保存 Q(y)的值