[读书笔记]CSAPP：7[VB]机器级表示：函数

视频地址：

【精校中英字幕】2015 CMU 15-213 CSAPP 深入理解计算机系统 课程视频_哔哩哔哩 (゜-゜)つロ 干杯~-bilibiliwww.bilibili.com/video/av31289365?p=7![img](https://pic4.zhimg.com/v2-82eac1470b916682f49fd18b47cf7d23_180x120.jpg)

课件地址：

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/07-machine-procedures.pdfwww.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/07-machine-procedures.pdf

对应于书本的3.7。

如有错误请指出，谢谢。

小点：

- C中的取地址符`&`返回的是内存地址，所以一定要保存在内存中。
- 保存到内存中进行参数传输时，要求每个参数大小为8字节的倍数，即要求相对`%rsp`的偏移量为8的倍数
- 不会显示地操作程序计数器寄存器`%rip`，没有指令可以对其操作，只能通过类似`call`或`ret`间接对其操作。
- 栈顶指针`%rsp`是随着函数运行不断变化的。
- 函数可以假设“被调用者保存寄存器”的值是不变的，而可以用“调用者保存寄存器”来保存临时值。
- 某个函数要永久使用的值，要么保存在“被调用者保存寄存器”中，要么保存在内存中。
- 当函数需要使用“被调用者保存寄存器”时，就直接将其`push`到栈中，使用过后再`pop`重置。
- 无论是“被保存的寄存器”还是“局部变量”以及“参数构造区”，一开始如何申请这些区域，后面使用完后还会逆向地通过`%rsp`将这些区域释放掉，这是动态的过程，使得一个函数运行完时，`%rsp`指向的就是返回地址，就能直接通过`ret`返回到调用者的断点处。

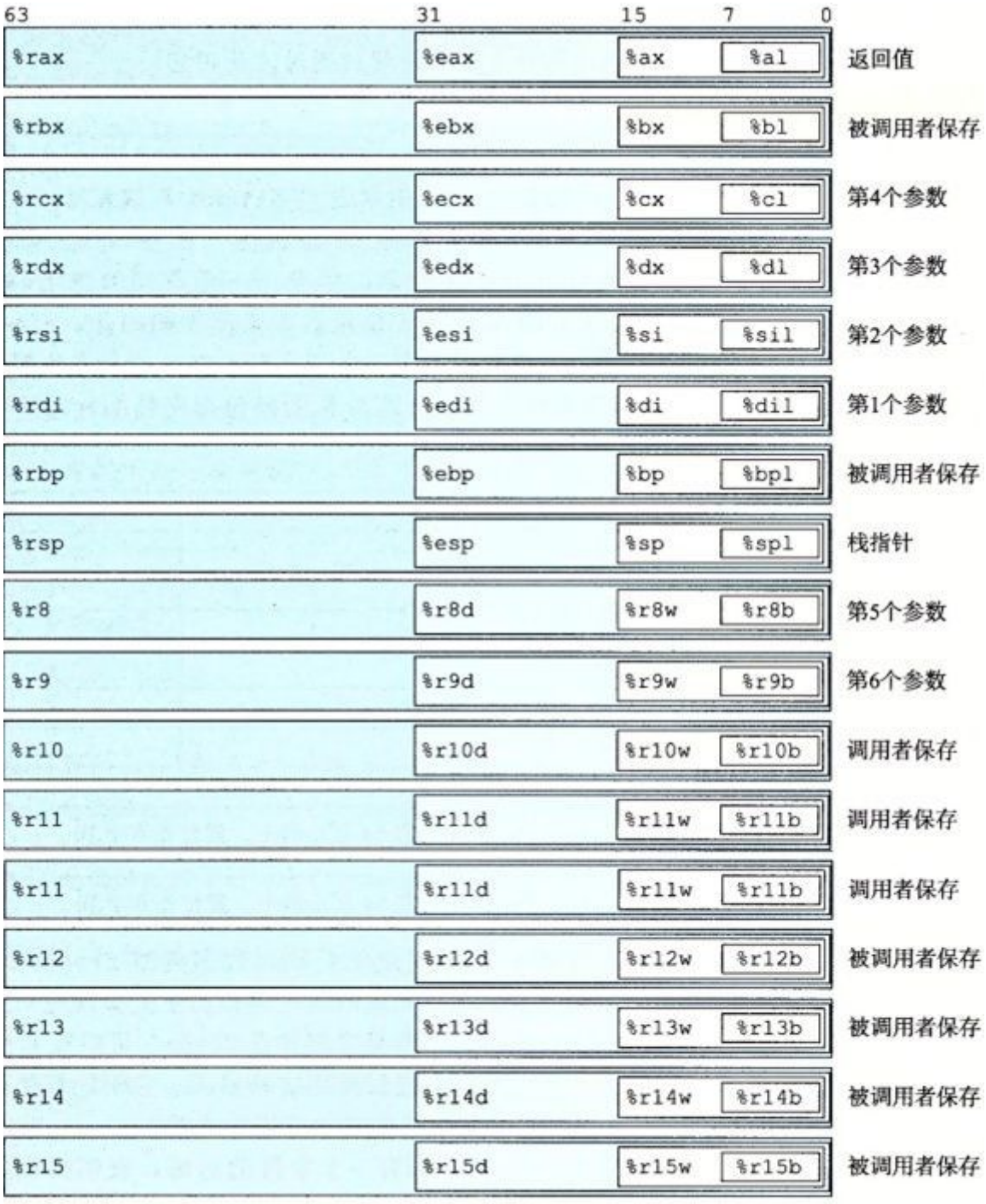


图 3-2 整数寄存器。所有 16 个寄存器的低位部分都可以作为字(16 位)、双字(32 位)和四字(64 位)数字来访问

- 进入一个函数时，首先将要使用的“被调用者保存寄存器”`push`到栈中，然后**通过`%rsp`来申请一段固定大小的空间**，用来存放局部变量和参数构造区，最后再释放申请的空间。

要提供对函数的机器级支持，必须处理许多不同的属性。我们假设函数P调用函数Q，Q执行后返回P。这个过程包括以下一个或多个**机制**：

- **传递控制**：在进入函数Q的时候，程序计数器要设置为Q的代码的起始位置。从函数Q返回时，要把程序计数器设置为P中调用Q后面那条指令的地址，即从P中的断点处继续执行。
- **传递数据**：函数P必须能够向函数Q传递一个或多个参数，而函数Q必须能够向函数P返回一个值。
- **分配和释放内存**：开始时，函数Q可能需要为局部变量分配空间，而在返回前，又要释放这些存储空间。

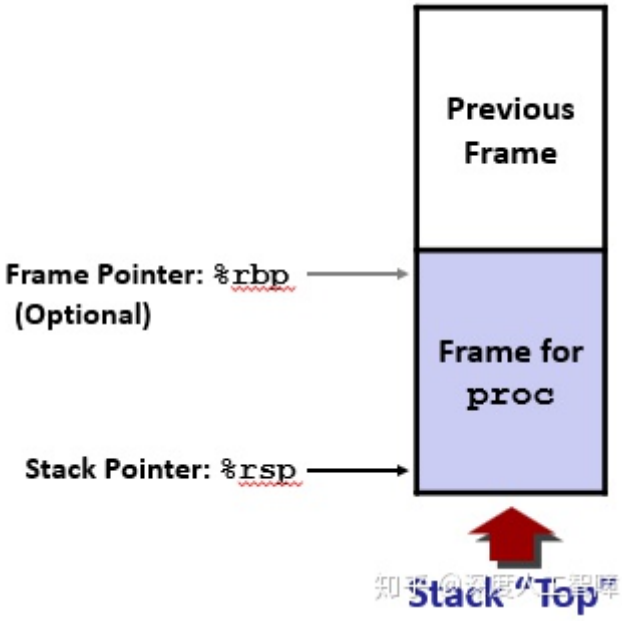
要想详细了解这些机制，我们首先要知道在内存中是如何保存函数的。

1 运行时栈

C语言的函数调用机制一个**关键特性**在于使用了栈的内存管理原则。通过栈的先进后出的性质，能够在内存中保证函数调用并返回的顺序。这里首先需要知道一个概念——**栈帧**。

栈帧：当函数需要的存储空间超出寄存器能够存放的大小，或者调用别的函数需要保存额外数据时，就会在栈上分配一个空间，这个空间称为函数的**栈帧（Stack Frame）**。相对的，当某个函数的所有局部变量都能保存在寄存器中，并且不会调用任何的函数时，就无需开辟该函数的栈帧了。当给一个函数创建栈帧时，编译器会给函数分配**所需**的**定长**的栈帧，在函数开始时就分配好后就不会改变了，所以栈顶指针`%rsp`就知道当函数返回时，需要释放多少空间。而有些函数需要变长的栈帧，这部分内容可参考深度人工智障：[读书笔记]CSAPP：10[VB]机器级表示：进阶。

注意：栈顶的栈帧对应了正在运行的函数。



所以每个函数的栈帧就作为栈的基本元素，来起到函数调用时先进后出的效果，会在栈中保存之前所有还未返回的函数的栈帧，将之前的函数先挂起。这里提供了`PUSH`和`POP`指令对栈进行操作，也可以直接对栈顶指针`%rsp`进行操作。

注意：因为未返回的函数都会在内存中保存自己的栈帧，而栈的空间是有限的，所以当调用过多时，会造成栈的溢出。

我们先简单介绍下函数调用时，可能会做的一些操作：

- **当函数P调用函数Q运行时**：
- - 由于x86-64只提供6个寄存器来传递函数输入值，所以当函数P传递给函数Q的参数多于6个时，需要函数P在自己的栈帧中存储好这些输入参数。
  - 会先将返回地址压入栈中，表明当函数Q返回时，要从函数P中的哪个位置继续执行，这个作为P的栈帧的一部分。
- **函数Q运行时**：函数Q会扩展当前栈的边界，分配函数Q的栈帧所需的空间，可以用来保存寄存器的值、分配局部变量空间，为函数Q调用其他函数设置参数。
- **函数Q返回时**：释放分配给函数Q的栈帧，并且让程序计数器调用返回地址，继续从函数P的断点处继续执行。

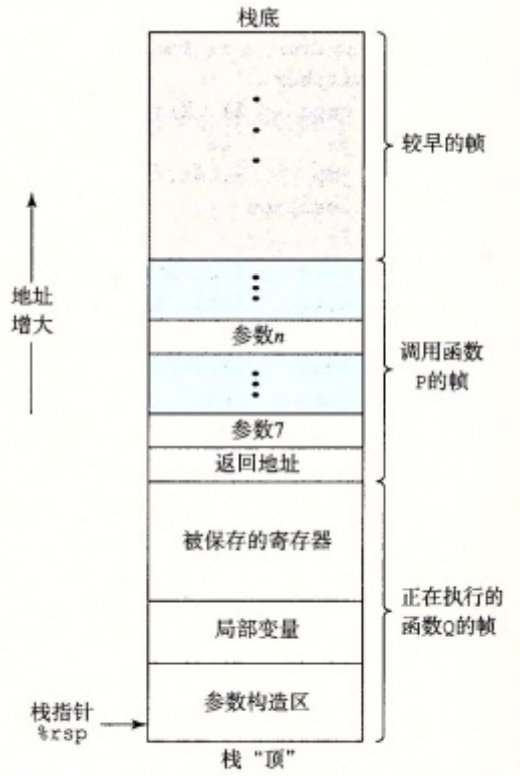


图 3-25 通用的栈帧结构(栈用来传递参数、存储返回信息、保存寄存器、以及局部存储。省略了不必要的部分)

为了能够完成以上过程，就需要上面介绍的3中机制相互配合，接下来会介绍每个函数在栈帧上的各个组成部分，依次为：被保存的寄存器-->局部变量-->参数构造区-->返回地址。

2 栈帧的组成部分

2.1 被保存的寄存器

寄存器是所有函数共享的资源，当函数P调用Q时，如果函数Q改变了函数P保存在寄存器的值，则当函数Q返回时，函数P就无法完全从断点继续执行，因为寄存器中的值已经被函数Q改变了。

我们对除了栈指针`%rsp`外的所有寄存器分成两类：

- **被调用者保存寄存器：** `%rbx`、`%rbp`和`%r12`~`%r15`。这部分寄存是由被调用者，即Q保存的。如果Q改变了这部分寄存器的值，就需要将其保存在Q自己栈帧中的**“被保存的寄存器”**中。当Q返回时，再将这部分寄存器的值根据内存复原。所以函数P可以假设“被调用者保存寄存器”的值是始终不变的。
- **调用者保存寄存器：** 除了上面的寄存器外，都属于被调用者保存寄存器。 任何函数都能修改这些寄存器的值，并且不会保存在“被保存的寄存器”中，所以P要自己将这部分寄存的内容保存起来。所以函数P可以假设“调用者保存寄存器”的值是变化的，需要自己保存，可以用这部分寄存器保存临时值。

注意： 当函数P调用函数Q时，“被调用者保存寄存器”就会保存在函数Q的栈帧中，所以当函数Q返回时，这部分寄存器会被重置为函数P使用时的状态。而**其他寄存器的值是需要函数P自己保存的，所以函数P需要自己开辟局部变量区域来保存其他寄存器的值。**

步骤：

```
//1.函数一进来，就需要通过push指令将自己要使用的“被调用者保存寄存器”保存在自己的栈帧中
//比如使用了%rbx和%rbp
  pushq %rbx    //保存%rbx到栈帧中
  pushq %rbp    //保存%rbp到栈帧中
... //可以使用“被调用者保存寄存器”来保存值
//2.当“被调用者保存寄存器”不够保存当前函数的值时，需要开辟局部变量空间保存其他值
//比如保存8字节值
  subq $8, %rsp //将栈指针下移8个字节
  movq %rdi, (%rsp)  //将需要保存的值保存到栈上
```

```
//3.调用别的函数
  call func //调用函数func，则“被调用者保存寄存器”会保存在函数func的栈帧中
... //可以继续使用“被调用者保存寄存器”，因为函数func返回时会重置这些寄存器到原始值
//4.释放局部变量空间
  addq $8, %rsp
//5.重置“被调用者保存寄存器”的值，注意顺序要相反
  popq %rbp
  popq %rbx
```

综上所述：

1. 将要使用的“被调用者保存寄存器”`push`到栈中。（存储调用当前函数的函数的值）
2. 将除了“被调用者保存寄存器”的其他寄存器保存在空闲的“被调用者保存寄存器”中，如果保存不下，就将其保存在内存的“局部变量”区域
3. 调用其他函数
4. 释放 “局部变量”区域
5. 将“被调用者保存寄存器”的值通过`pop`从栈中恢复。注意：顺序要和`push`时相反

注意：如果“被调用者保存寄存器”还没有使用完毕时，可以在调用别的函数之前将其他寄存器的值保存到“被调用者保存寄存器”中。

2.2 局部变量

当函数需要保存的数据不多时，就会将数据保存在“被调用者保存寄存器”中。但是以下情况必须**保存在内存中**，该部分称为该函数的**局部变量**：

- “被调用者保存寄存器”不足以保存所有的本地数据
- 当一个局部变量使用取地址符&时，指的是返回该变量在内存中的地址，就必须将其保存在内存中
- 当局部变量是数组或结构时

以以下函数为例

```
long call_proc()
{
    long  x1 = 1; int  x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```



```
call_proc:
  //1. 因为x1、x2、x3和x4都有取地址符，所以都是保存在内存中的，首先要通过%rsp分配存储空间
  subq $32, %rsp        //将栈顶指针%rsp下移32，扩展了32字节空间
  //2. 将数据保存在内存中
  movq $1, 24(%rsp)     //因为x1为long，所以需要8字节空间，所以将1保存在24(%rsp)处
  movl $2, 20(%rsp)     //因为x2为int，所以需要4字节空间
  movw $3, 18(%rsp)     //因为x3为short，所以需要2字节空间
  movb $4, 17(%rsp)     //因为x4为char，所以需要1字节空间
  //3. 依次根据proc传入参数的顺序保存在内存和寄存器中
  //因为总共要传入8个参数，超过了6个参数，所以要将最后的x4和&x4保存在内存中，这里要求和8字节对齐
  leaq 17(%rsp), %rax   //获取&x4，因为不能直接在内存间mov，所以需要先保存到寄存器%rax
  movq %rax, 8(%rsp)    //参数8：将&x4保存在内存8(%rsp)中，这里要求是8的倍数
  movl $4, (%rsp)       //参数7：将x4保存在内存(%rsp)中
  leaq 18(%rsp), %r9    //参数6：将&x3保存在%r9中
```

```
  movl $3, %r8d         //参数5：将x3保存在%r8中
  leaq 20(%rsp), %rcx   //参数4：将&x2保存在%rcx中
  movl $2, %edx         //参数3：将x2保存在%rdx中
  leaq 24(%rsp), %rsi   //参数2：将&x1保存在%rsi中
  movl $1, %edi         //参数1：将x1保存在%rdi中
  //4. 调用函数
  call proc
  //5. 根据%rsp将保存在内存中的数据进行计算
...
  //6. 删除空间
  addq $32, %rsp
```

可以将其总结为以下几步：

1. 申请局部空间，通过对栈顶指针`%rsp`减掉一个值
2. 根据数据大小，通过`%rsp`索引将数据保存在内存空间中
3. 根据传入参数顺序，将其保存到内存和寄存器中。**注意：**参数大小要为8字节的倍数。
4. 释放局部空间，通过对栈顶指针`%rsp`加上`1.`中的值

2.3 参数构造区

主要任务：函数P必须能够向函数Q传递一个或多个参数，而函数Q必须能够向函数P返回一个值。

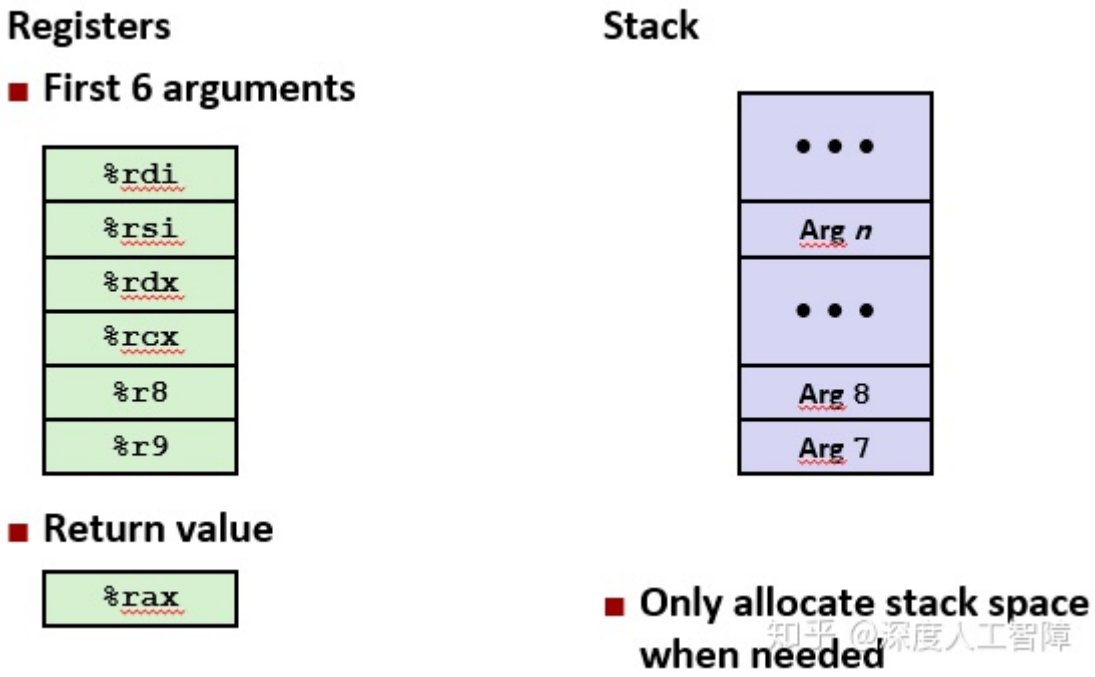
在函数间传递数据，主要**通过寄存器**进行，x86-64提供了6个用于传递**参数**的寄存器，根据参数的顺序，需要放入特定的寄存器中。x86-64将寄存器`%rax`作为函数**返回值**的寄存器。

注意：这些寄存器只能用来保存整数或指针类型。

操作数大小(位)	参数数量					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

图 3-28 传递函数参数的寄存器。寄存器是按照特殊顺序来使用的，而使用的名字是根据参数的大小来确定的

如果某个函数要传递超过6个参数的话，就需要将第7个到第n个参数保存在**栈**中，然后通过**栈顶指针**`%rsp`进行索引其中第7个参数在栈顶位置。要求每个参数的大小要为8字节的倍数。这部分区域称为**参数构造区**。保存在寄存器中的参数访问起来比保存在内存中快很多。



由于栈顶还要保存一个8字节的返回地址，所以第7个参数的地址为`8(%rsp)`，如果第7个元素大小不超过8字节，则第8个元素的地址为`16(%rsp)`，以此类推。

2.4 返回地址

主要任务：在进入函数Q的时候，程序计数器要设置为Q的代码的起始位置。从函数Q返回时，要把程序计数器设置为P中调用Q后面那条指令的地址，即从P中的断点处继续执行。

x86-64提供了一组指令来完成上述操作

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

- `call`：可以直接将函数名作为跳转目标，其编码的计算方式和`jmp`相同。相当于`push`和设置`%rip`的组合。
-
  - 将栈指针减8，留出保存返回地址的空间
  - 将紧跟`call`指令后面那条指令的地址作为返回地址，保存到栈中。
  - 将程序计数器设置为调用函数的地址。
- `ret`：从当前函数返回，不需要操作数。相当于设置`%rip`和`pop`的组合。
-
  - 将程序计数器设置为栈顶元素。
  - 将栈指针加8。

注意：在64位操作系统中，返回地址是64位8字节的。

以以下汇编代码为例

```
     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
       . . .
     Call to multstore from main
5      400563:  e8 d8 ff ff ff        callq  400540 <multstore>
6      400568:  48 8b 54 24 08        mov    0x8(%rsp),%rdx
```

从第5行的`main`函数开始，调用了`multstore`函数。

1. 当前程序计数器为`400563`，`callq 400540`会先将下一行的地址`400568`压入栈中，并将程序计数器设置为`400540`；
2. 执行完`multstore`函数后，运行第4行的`retq`时，会将程序计数器设置为栈顶元素，并将栈顶元素出栈。

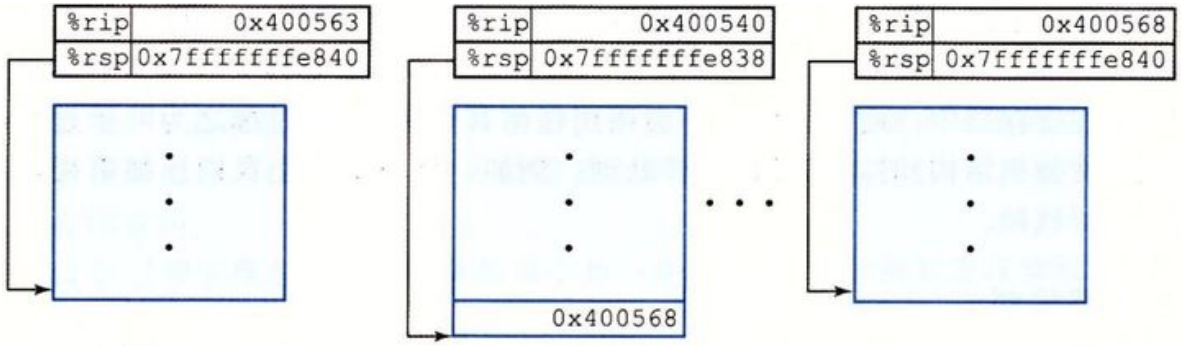


图 3-26 call 和 ret 函数的说明。call 指令将控制转移到一个函数的起始，而 ret 指令返回到这次调用后面的那条指令

我们这里可以计算一下为什么第5行的编码是`e8 d8 ff ff ff`。首先，这个`e8`是`call`指令的编码，而`call`的目标地址为`400540`，该下一行的地址为`400568`，则计算目标地址和下一行地址的差，得到`400540-400568=ffffffd8`就是对应的编码，而这里是小端机，所以需要对字节进行翻转，就得到`d8ffffff`。

综上所述：

1. （被保存的寄存器）函数P将要使用的“被调用者保存寄存器”通过`push`保存在函数的栈帧中。
2. （局部变量）如果函数P使用了“调用者保存寄存器”，就需要将其保存在栈中，才能调用函数Q。并且函数P根据需要申请空间来保存其他局部变量。
3. （参数构造区）函数P将参数保存在寄存器中，如果超过6个参数，就申请空间保存到内存中。
4. （返回地址）函数P使用`call`指令调用函数Q，会将`call`的下一行指令的地址压入栈中，并将程序计数器指向函数Q的第一条指令的地址。
5. 当函数Q运行时会随着使用动态申请和释放局部变量，当函数Q运行完时，首先使用栈“被调用者保存寄存器”的值，然后使用`ret`指令返回将程序计数器设置为栈顶的返回地址，最后将栈顶的返回地址弹出。

我们可以发现有趣的一点是，无论是“被保存的寄存器”还是“局部变量”以及“参数构造区”，一开始如何申请这些区域，后面使用完后还会逆向地通过`%rsp`将这些区域释放掉，这是动态的过程，使得一个函数运行完时，`%rsp`指向的就是返回地址，就能直接通过`ret`返回到调用者的断点处。