05｜代码封装（上）：函数是如何被调用的？

2021-12-15 于航

《深入C语言和程序运行原理》 课程介绍 >

讲述：于航

时长 15:26 大小 14.14M

你好，我是于航。

在前两讲中，我介绍了 C 语言中的运算符、表达式、语句是如何被编译器实现的。不知你是否还记得，在介绍运算符时，我没有展开讲解有关函数调用运算符的内容。接下来，我就用专门的两讲内容，来带你深入看看 C 语言中有关函数调用的那些事儿。

这一讲，我们首先来看 C 语言中，编译器实现函数调用时所遵循的一系列规则。这些规则实际影响着函数调用时，在如何传参、如何使用寄存器和栈内存等问题上的处理细节。

除此之外，由于 C 语言中的函数调用过程与栈内存密切相关，我还会介绍栈和栈帧的概念。栈是 C 程序在运行时用于存放临时数据的一块内存，而每一个栈帧都对应着栈内存中的一段数据，这些数据是在函数调用过程中所必须使用的。通过这一讲的学习，你能了解

到编译器对 C 函数调用的处理细节。而在下一讲中，我们将以此为基础，来深入探讨尾递归调用优化等更多函数调用的相关内容。

快速回顾 C 语言中函数的使用方式

函数的概念相信你已经十分熟悉了，这里我们先来快速回顾一下。

在编程语言中，函数是一种用于封装可重用代码的语法结构。函数可以接收从外部调用环境传入的数据，并在函数体内以复合语句的形式，使用这些数据构建独立的功能逻辑单元。**借助函数，我们可以将一个程序的实现过程拆分为多个子步骤，并以结构化的方式来构建程序。**这种方式可以减少程序中的重复代码，并通过抽象和替换来提高代码的整体可读性，以及可追溯性。

在 C 语言中，函数的定义与使用方式跟其他语言大同小异，我们先通过一个例子快速回顾一下。这里，你可以先停下来，尝试编译和运行下面这段代码，并观察其中的函数调用逻辑。需要注意的是，在编译时，我们要为编译器指定 “-lm” 参数，来让它链接程序运行所需要的数学库。

复制代码

```
#include <stdio.h>
#include <tgmath.h>
typedef struct {
  int x;
  int y;
} Point;
int foo(int x, int y, Point* p, int(handler)(int)) {
  return handler(x + y + p->x + p->y);
}
int handler(int n) {
  return sqrt(n);
}
int main(void) {
  int x = 2;
  int y = 3;
  Point p = { .x = 10, .y = 10 };
  printf("%d", foo(x, y, &p, handler));  // 5.
  return 0;
}
```

在 C 语言中，函数有两种传递参数的方式，即通过“值”传递和通过“指针”传递。其中，对于值传递的方式，编译器会在函数调用时，将传入函数的参数值进行复制。因此，在这种情况下，调用时传入函数的参数与在函数内部使用的参数是两个不同的实体。而使用指针形式传入的参数，因为指针所表示的地址在传入函数前后均不会发生变化，所以如果在函数内部修改指针参数所指向的值，则发生在该值上的变化，在函数调用完成后也将一直存在。

这里在代码中，函数 foo 一共接收了四个参数，其中整型参数 x 与 y 均以值的方式传递。而对于紧接着的结构体与函数类型的参数，它们对应的变量 p 与 handler 则以指针的形式传递。需要注意的是，对于函数指针来说，可以在声明和调用时为其省略通常用于表明指针类型的 “*” 符号。这意味着函数 foo 的定义也可以写成如下形式：

复制代码

```
int foo(int x, int y, Point* p, int(*handler)(int)) {
  return (*handler)(x + y + p->x + p->y);
}
```

C 函数的调用约定

我在 02 讲 中和你提到过，C 标准中并未规定，语言的各类语法结构应该以怎样的方式来实现。但实际上，从编译器的角度来看，每一个函数在被调用时，应该以怎样的方式通过机器指令来实现其调用过程，却存在着相应的事实标准。而通常，我们把编译器实现函数调用时所遵循的一系列规则称为函数的“调用约定（Calling Convention）”。

调用约定规定了函数调用时需要关注的一系列问题，比如：如何将实参传递给被调用函数、如何将返回值从被调用函数中返回、如何管理寄存器，以及如何管理栈内存，等等。调用约定并非 C 语言标准的一部分，因此实际上每个编译器都可以使用自己独有的调用约定，来实现 C 函数的调用过程。但相应地，这也会导致另外一个问题：当具有外部链接的函数在多个不同编译单元内被使用，且这些不同编译单元对应的源文件通过不同的编译器进行编译时，那么它们各自生成的对象文件可能无法再被整合在一起，并生成最终的可执行文件。

幸运的是，对于 C 语言来说，运行在 x86-64 平台上的编译器基本都会根据所在操作系统的不同，选择使用几种常见的调用约定事实标准。比如，对于 Windows 来说，编译器会采用专有的 Microsoft x64 或 Vector 调用约定。而在 Unix 和类 Unix 系统上，则会使用

名为 System V AMD64 ABI（后简称 “SysV” ）的调用约定。类似地，对于 i386（IA32）、8086 等其他平台，它们也都有着对应的调用约定事实标准。**而较为统一的调用约定，也在一定程度上保证了 C 程序在同一平台不同编译器下的最大可移植性。**

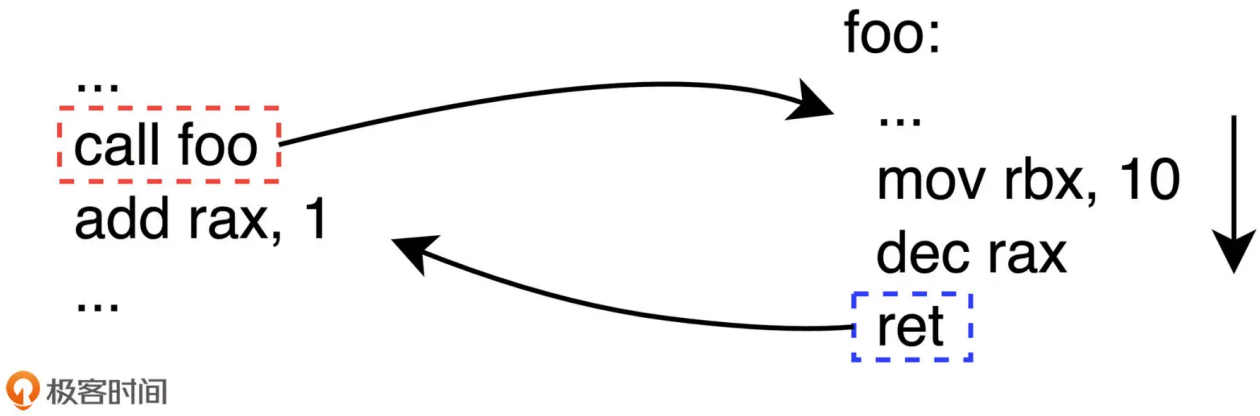
接下来，让我们看看 SysV 调用约定中都规定了哪些重要的实现细节。为了更直观地观察这些内容，让我们先来编写一段简单的 C 代码，并在 x86-64 平台上使用默认优化等级，通过 GCC 编译生成它所对应的汇编代码。具体如下图所示（在后面提到这张图时，我会统一用 “图 A” 代替）：

```c
#include <stdio.h>
int bar() {
  return 10;
}
int foo(
  int a,
  int b,
  int c,
  int d,
  int e,
  int f,
  int g,
  int h) {
    int n = 10;
    return n + bar();
}
int main(void) {
  int x = 1;
  int y = 2;
  printf("%d", foo(x, y, 3, 4, 5, 6, 7, 8));
  return 0;
}
```

```asm
bar:
        push    rbp
        mov     rbp, rsp
        mov     eax, 10
        pop     rbp
        ret
foo:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 40
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-24], esi
        mov     DWORD PTR [rbp-28], edx
        mov     DWORD PTR [rbp-32], ecx
        mov     DWORD PTR [rbp-36], r8d
        mov     DWORD PTR [rbp-40], r9d
        mov     DWORD PTR [rbp-4], 10
        mov     eax, 0
        call    bar
        mov     edx, DWORD PTR [rbp-4]
        add     eax, edx
        leave
        ret
.LC0:
        .string "%d"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 1
        mov     DWORD PTR [rbp-8], 2
        mov     esi, DWORD PTR [rbp-8]
        mov     eax, DWORD PTR [rbp-4]
        push    8
        push    7
        mov     r9d, 6
        mov     r8d, 5
        mov     ecx, 4
        mov     edx, 3
        mov     edi, eax
        call    foo
        add     rsp, 16
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        leave
        ret
```

在上图中，左侧为 C 代码，右侧为对应的汇编代码，相同颜色的代码块表示源代码与汇编代码之间的对应关系。在 C 代码中，我们定义了名为 bar 与 foo 的两个函数，并在 foo 中调用了 bar。bar 函数不接收任何参数，调用后直接返回整型值 10。foo 函数共接收 8 个参数，调用后返回其内部整型变量 n 与函数 bar 调用返回值的和。在 main 函数中，定义有两个整型局部变量 x 与 y，而当函数 foo 被调用时，直接使用这两个局部变量，以及另外的 6 个字面量数字值作为它的参数。

实际上，在 x86-64 的机器指令中，函数调用是通过 `call` 指令来完成。而每一个函数体在执行完毕后，都需要再通过 `ret` 指令来退出函数的执行，并转移代码执行流程到之前函数调用指令的下一条指令上。你可以通过下面这张图来直观地感受这个流程。其中，箭头标注出了代码的整体执行顺序。



接下来，我们来具体看看 SysV 调用约定中都规定了函数调用时的哪些内容。

参数传递

SysV 调用约定的第一个规则是：在调用函数时，对于整型和指针类型的实参，需要分别使用寄存器 rdi、rsi、rdx、rcx、r8、r9，按函数定义时参数从左到右的顺序进行传值。而若一个函数接收的参数超过了 6 个，则余下参数将通过栈内存进行传送。此时，多出来的参数将按照从右往左（RTL）的顺序被逐个压入栈中。关于这一点，你可以通过图 A 右侧第 30 到 40 行红框内的汇编代码得到验证。

这里，函数 foo 在调用前，分别用寄存器 edi、esi 存放局部变量 x 与 y 的值，并用寄存器 edx、ecx、r8d、r9d 存放字面量值 3、4、5、6（如果你还不了解寄存器 rdi 与 edi 的关系，可以在课前热身一讲中得到答案）。而多出来的另外两个字面量值参数 7 和 8 ，则直接通过 `push` 指令被放在了栈内存中。你需要注意这里指令操作它们的先后顺序，因为

要保证这些参数以从右向左的顺序被放入栈中。另外，由于 x、y 为局部变量，因此最开始它们会被存储在栈内存中。

除此之外，对于浮点参数，编译器将会使用另外的 xmm0 到 xmm7，共 8 个寄存器进行存储。对于更宽的值，也可能会使用 ymm 与 zmm 寄存器来替代 xmm 寄存器。而上面提到的 xmm、ymm、zmm 寄存器，都是由 x86 指令集架构中名为 AVX（Advanced Vector Extensions）的扩展指令集使用的。这些指令集一般专门用于浮点数计算以及 SIMD 相关的处理过程。

返回值传递

对于函数调用产生的返回值，SysV 调用约定也有相应的规则：当函数调用产生整数类型的返回值，且小于等于 64 位时，通过寄存器 rax 进行传递；当大于 64 位，小于等于 128 位时，则使用寄存器 rax 与 rdx 分别存储返回值的低 64 位与高 64 位。你可以参考图 A 右侧第 4、21、47 行蓝框内的代码，来验证这个规则。这三行代码分别处理了函数 bar、foo，以及 main 的返回值。需要注意的是，对于复合类型（比如结构体）的返回值，编译器可能会直接使用栈内存进行“中转”。

对于浮点数类型的返回值，同参数传递类似，编译器会默认使用 xmm0 与 xmm1 寄存器进行存储。而当返回值过大时，则会选择性使用 ymm 与 zmm 来替代 xmm 寄存器。

寄存器使用

SysV 调用约定对寄存器的使用也作出了规定：对于寄存器 rbx、rbp、rsp，以及 r12 到 r15，若被调用函数需要使用它们，则需要该函数在使用之前将这些寄存器中的值进行暂存，并在函数退出之前恢复它们的值（callee-saved）。而对于其他寄存器，则根据调用方的需要，自行保存和恢复它们的值（caller-saved）。

堆栈清理

每一个函数在调用结束前，都需要由它自身完成堆栈的清理工作。比如在图 A 所示的代码中，foo 函数在被调用时，它在栈内存中分配了对应的空间，用于存放局部变量 n 的值。而在该函数执行完毕，准备退出前，便需要由它自己将之前在栈上分配的数据清理干净。而这个任务是可以由 `leave` 指令来完成的。我会在接下来讲解“栈帧”时，再深入介绍与该指令相关的内容。

除此之外，对于 foo 函数被调用前所传入实参的清理工作，则是由调用函数，也就是这里的 main 函数来完成的。可以看到，当 foo 函数调用结束，程序执行流程返回到之前 `call` 指令的下一条指令时，程序通过 `add` 指令修改了 rsp 寄存器的值。通过这种方式，main 函数对之前放入栈中传递给函数 foo 的实参进行了清理。

其他约定

除此之外，SysV 调用约定还有下面这几点规定：

> 函数在被 `call` 指令调用前，需要保证栈顶于 16 字节对齐，也就是栈顶的所在地址值（以字节为单位）是 16 的倍数；
>
> 从栈顶向上保留 128 字节作为 “Red Zone” ；
>
> 不同于用户函数的调用过程，系统调用（System Call）函数需使用寄存器 rdi、rsi、rdx、r10、r8、r9 传递参数。

我们来重点看看第二点：Red Zone 是位于栈顶向上（低地址方向）的一段固定长度的内存段，这块区域通常可以被函数调用栈中的“叶子”函数（即不再调用其他函数的函数）使用。这样，在需要额外的栈内存时，就能在一定条件下省去先调整栈内存大小的过程。而有关第三点中涉及到的与系统调用相关的内容，我将在这门课的“C 程序运行原理篇”中再为你深入讲解。

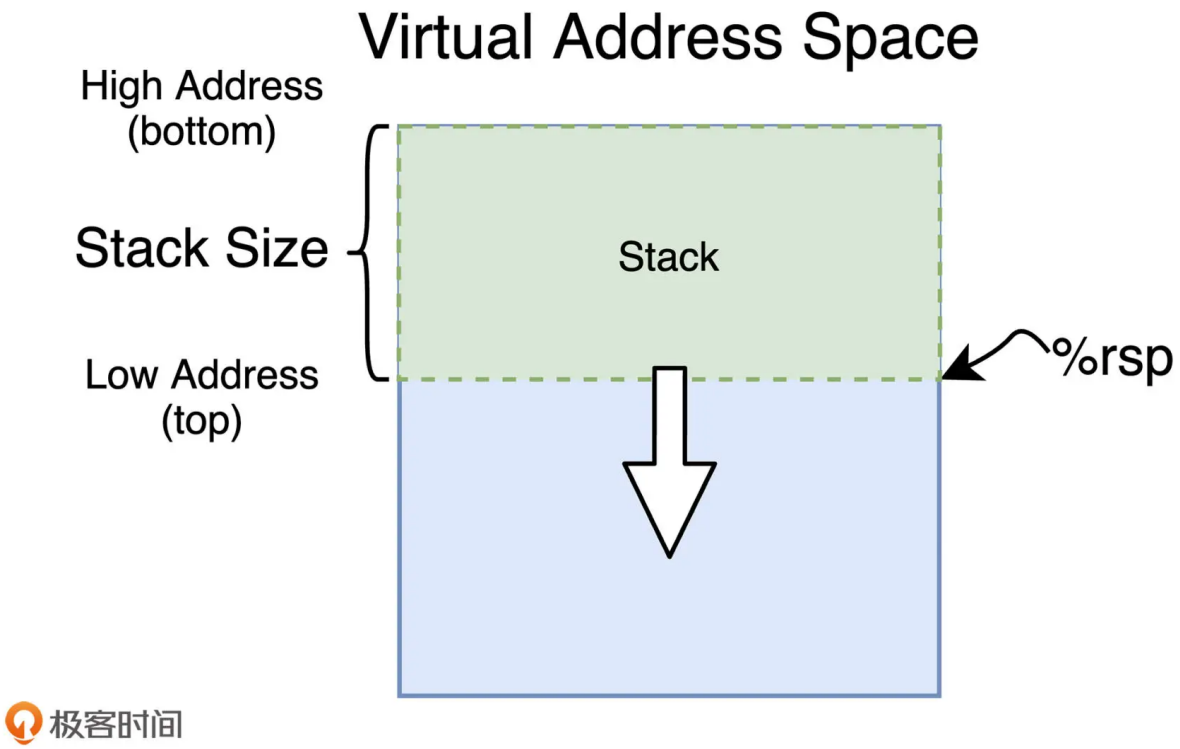
保存函数调用信息的栈帧

函数的调用过程伴随着栈内存中数据的不断变化。从整体上来看，每一个函数在调用时，都会在栈内存中呈现出基本相同的数据布局结构。而通过这种方式划分出来的，对应于每一次函数调用的栈内存数据块，我们一般称它为“栈帧”。栈帧中存放有与每个函数调用相关的返回地址、实参、局部变量、返回值，以及暂存的寄存器值等信息。

在进程的 VAS 中，栈内存是从高地址向低地址逐渐增长的，即栈底位于高地址处，栈顶位于低地址处。而当一个函数在执行过程中需要使用更多的栈内存空间时，便需要首先通过某种方式来扩大进程的可用栈内存大小。

通过操作寄存器 rsp，我们便可完成这个操作。rsp 寄存器又被称为 Stack Pointer，该寄存器中一直存放着当前栈内存顶部（低位地址）的地址。也就是说，**rsp 寄存器的值决定**

了进程所能够使用的栈内存大小。因此，通过减小该寄存器的值，我们便能够扩大进程的可用栈内存空间。你可以通过下图，直观地体会到它们之间的关系：



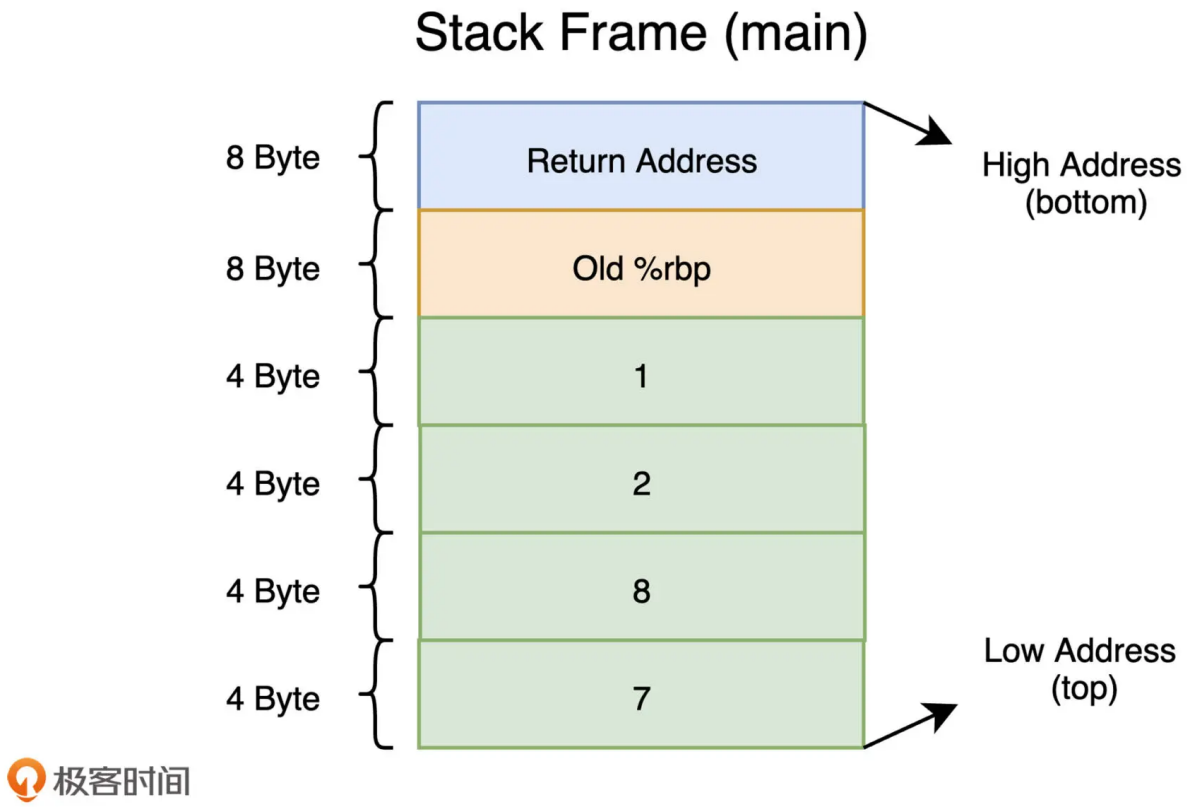
现在让我们把目光移动到函数 bar 身上，来详细看看，它在通过 `call` 指令调用后都发生了什么。

当 `call` 指令执行时，函数执行完毕后的返回地址会被首先推入栈中。以 bar 函数为例，当该函数被调用时，图 A 中右侧代码第 20 行对应的机器指令地址便会被存放到栈内存中。接下来，函数的第一行指令 `push rbp` 会将当前寄存器 rbp 的值暂存到栈中，以便在函数执行完毕后恢复该寄存器的值。rbp 寄存器又被称为 Frame Pointer，即“栈帧寄存器”。通常情况下，它被用来存储函数调用前的“栈高度”，即寄存器 rsp 的旧值，以便用于在函数执行过程中进行栈帧中数据的寻址，并在函数退出前把栈中的数据恢复到函数调用前的状态。

紧接着，第二句指令 `mov rbp, rsp` 便将存有此刻栈高度的寄存器 rsp 的值“备份”到寄存器 rbp 中。当函数体的内容（第三条语句）执行完毕后，程序通过 `pop` 指令恢复寄存器 rbp 的值，并通过 `ret` 指令将程序的执行转移到函数调用前，存入栈中的那个返回地址上去。

在函数 bar 的执行过程中，由于我们没有在栈上分配任何数据，因此在函数实际执行结束前，也并不需要对栈进行任何清理工作。所以你会发现，和 foo 函数与 main 函数相比，bar 函数在 `ret` 指令之前少执行了一条 `leave` 指令。而事实上，这条指令便会通过恢复寄存器 rsp 的值来“清理”栈上的数据，并同时恢复寄存器 rbp 的值。

进一步观察 main 函数的实现细节，你会发现函数在执行时使用栈的痕迹。比如汇编代码中的第 29 行，这里通过 `sub` 指令减小了寄存器 rsp 的值，以将当前的可用栈空间扩大 16 个字节。接着，通过第 30、31 行指令，函数为局部变量 x 和 y 分配相应的栈内存，并将初始值 1 和 2 分别存放到了栈上 rbp-4 与 rbp-8 的位置，每一个占用 4 字节大小。随后，在代码的第 34、35 行，借助 `push` 指令，额外的两个 4 字节参数值同样被存放到了栈内存中。此时，main 函数对应的栈帧内容如下图所示：



到这里，相信你已经对函数的调用过程以及栈帧的概念有了大致的了解。可以看到的是，随着嵌套函数的不断调用，每一个调用过程所产生的栈帧都会按照函数的调用顺序被依次存放在栈内存中。而当嵌套函数的层级足够深，导致栈内存已达到可用的最大值，进而无法再存放栈帧时，便会发生我们常见的 “Stack Overflow” ，即“栈溢出”的问题。而在下一讲中，我会带你一起看看，如何借助“尾递归优化”技巧来解决这个问题。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

这一讲，我首先带你快速回顾了 C 语言中函数的具体用法，然后介绍了编译器在实现 C 函数调用时需要关注的一系列规则，即 C 函数中的调用约定。在类 Unix 系统上，编译器通常会使用名为 System V AMD64 ABI 的调用约定，来作为实现函数调用的事实标准。SysV 调用约定中规定了函数在调用时需要注意的参数传递、寄存器使用，以及堆栈清理等方面的具体规则。

每一个被调用函数都会在栈内存中存放与其对应的栈帧结构。栈帧中包含着函数在被调用时需要的所有关键信息，其中包括函数返回地址、某些寄存器的旧值、函数调用过程中局部变量的值，等等。

思考题

在这一讲的最后，我们来一起做个思考题吧。

在不使用 `leave` 指令的情况下，你知道应该如何进行栈清理，并恢复寄存器 rbp 与 rsp 的值吗？而与它对应的 `enter` 指令又有什么作用呢？欢迎在评论区留下你的答案。

今天的课程就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

上一篇 04 | 控制逻辑：表达式和语句是如何协调程序运行的？

下一篇 06 | 代码封装（下）：函数是如何被调用的？

精选留言 (4)

写留言

2021-12-17

老师，请教文章里的几个问题:
1. 函数传参时参数的传递顺序是从右到左，也就是从8到7到6，最后到y和x，但这里是先y再x，然后再从右到左，这是为啥？还有传递x的时候为啥先放到eax ，最后再放到edi 里的，编译器做的什么优化吗？
2. 清理栈上的数据就是一个leave或者操作rsp指令，清理是指将所有数据置为0吗？还有...

展开

=

2021-12-16

不使用leave，可以使用
mov rsp,rbp
pop rbp
来恢复rsp和rbp的值。
对于enter，它和leave相反，用于自动创建栈帧，运用相当于...

展开

作者回复: 回答正确！

=

2021-12-15

老师好，请问如果一个函数现有的栈帧大小不够用了，当继续向栈中push数据的时候，rsp中保存的值是动态变化的吗

展开

作者回复: 是的，push 指令会自动修改 rsp 中的值。对于它的执行，你可以理解为两步：第一步是减小 rsp 中的值，“腾出”足够的栈内存；第二步是把值放入这块内存中。

爱新觉罗晓松

2021-12-15

在不使用 leave 指令的情况下，应该使用mov rsp, rbp 和 pop rbp, 将rbp寄存器的内容复

制到rsp寄存器中，以释放分配给该过程的所有堆栈空间。然后，从堆栈恢复rsp寄存器的旧值。
enter跟 push rbp 和 mov rbp, rsp等价，在调用函数时，创建堆栈帧。

展开

作者回复: 回答正确！