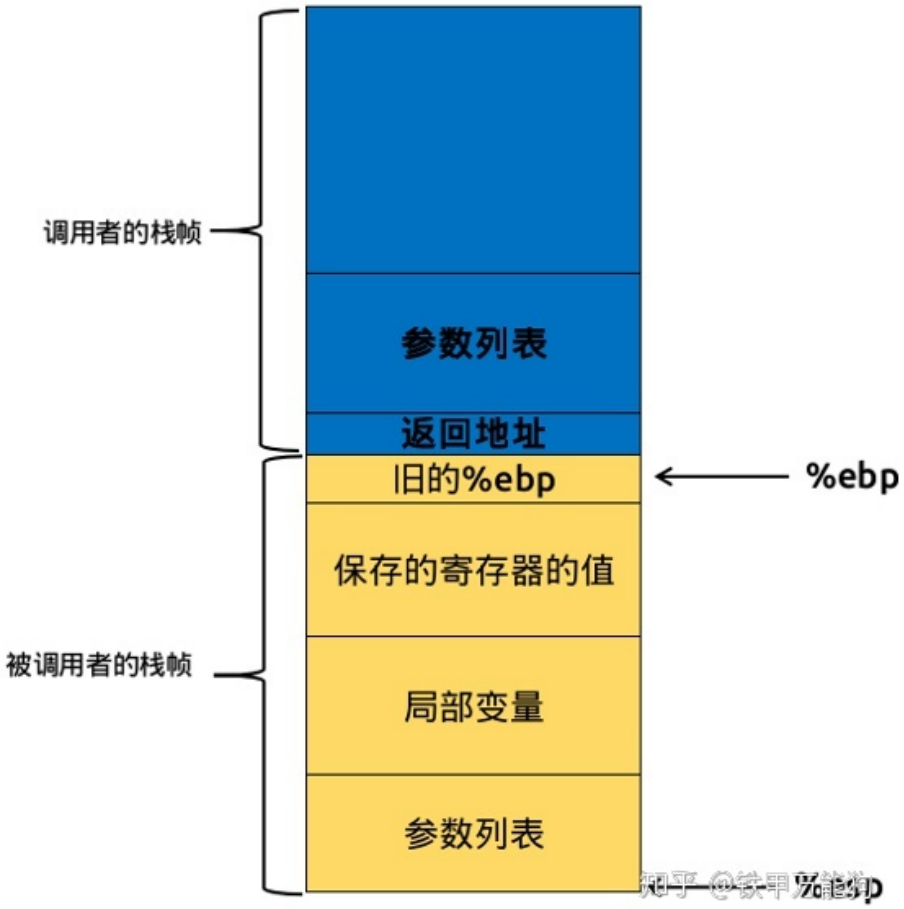
第7篇：C/C++程序栈-帧

铁甲万能狗
自由开发者，专攻C++/Python后端开发(简书平台同号)

本篇详细讲解有关IA32约定中的程序栈帧，我栈顶到栈底的方向逐一回顾一下。首先分析栈底层的内存细节，需要你具备非常基础的汇编语法知识即可。如果你毫无概念，可以自行查看我以前在简书写的汇编笔记，搜索关键字"**铁甲万能狗 汇编**"。



当前栈帧从栈顶**到**栈底**如下构成**

- **创建的参数表**为要调用的函数建立的参数
- **局部变量**:即在函数内部声明的变量(如果有)
- **保存的寄存器的上下文(如果有)**，当被调用函数返回之前，为调用者函数恢复原来的寄存器中的数据，如果当前架构有足够的空闲物理寄存器可供使用，一些寄存器信息可能不需要压入栈。
- 旧的帧指针，即前一栈帧的**ebp指针***，它指向前一帧的栈底。

调用者函数的栈帧

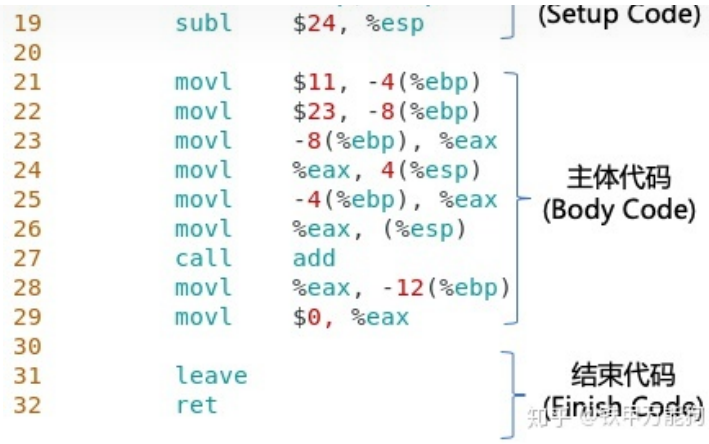
- **参数列表**
- **本地变量**（如果调用者函数内部有声明局部变量）
- **返回地址**:我们知道调用者函数内部的执行到call指令时会将call指令所在行的下一条指令的内存地址压入栈,当**被调用者函数内部**执行到ret指令后,从栈中弹出**返回地址**,以便调用者函数回到该地址对应的指令继续执行余下的指令。

```
int add(int x,int y){
      return x+y;
}
int main(void){
    int s=11;
    int b=23;
    b=b+s;
    return 0;
}
```

```
25 main:
26 .LFB1:
27         .cfi_startproc
28         pushl   %ebp
29         .cfi_def_cfa_offset 8
30         .cfi_offset 5, -8
31         movl    %esp, %ebp
32         .cfi_def_cfa_register 5
33         subl    $24, %esp
34         movl    $11, -4(%ebp)
35         movl    $23, -8(%ebp)
36         movl    -8(%ebp), %eax
37         movl    %eax, 4(%esp)
38         movl    -4(%ebp), %eax
39         movl    %eax, (%esp)
40         call    add
41         movl    %eax, -8(%ebp)
42         movl    $0, %eax
43         leave
44         .cfi_restore 5
45         .cfi_def_cfa 4, 4
46         ret
47         .cfi_endproc
48 .LFE1:
49         .size   main, .-main
50         .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-39)
   "
51         .section        .note.GNU-stack,"",@progbits
```


上面的汇编代码你会发现有很多以.cfi_为前缀的指令,**这些称为cfi指令集,主要用于C/C++调试器执行并获取程序栈帧的状态信息，跟生成的汇编代码没任何关系,编译后的可执行程序也不会执行它们，所以不要理会它们。**如果你非得转牛角尖(浪费时间)，可以参考如下链接。如果你想生成**干净的汇编代码**的话，可以在编译的时候，加上 -fno-asynchronous-unwind-tables这个选项，在这个示例中

```
gcc -S hello.c -o hello.s -fno-asynchronous-unwind-tables
```



干净的汇编代码

我们看到上图，下买你我们会逐步用图例来讲解。

- “**设定代码**”就是每个函数的汇编版本初始化栈帧中通用初始化的操作，下面会用详细的图例进行讲解
- “**主体代码**”是业务代码对应的汇编版本，即一系列局部变量初始化和call指令调用其他函数等。
- 标号3的位置是“**完成代码**”就是被调用函数返回值的处理和清理当前函数栈帧占用的内存。

反编译示例

其实我们可以进一步都我们的代码进一步反编译可以更加彻底了解内部的操作，通过如下指令

```
gcc -m32 -g hello.s -o hello.out;
objdump -d ./hello.out > hello.dmp;
```

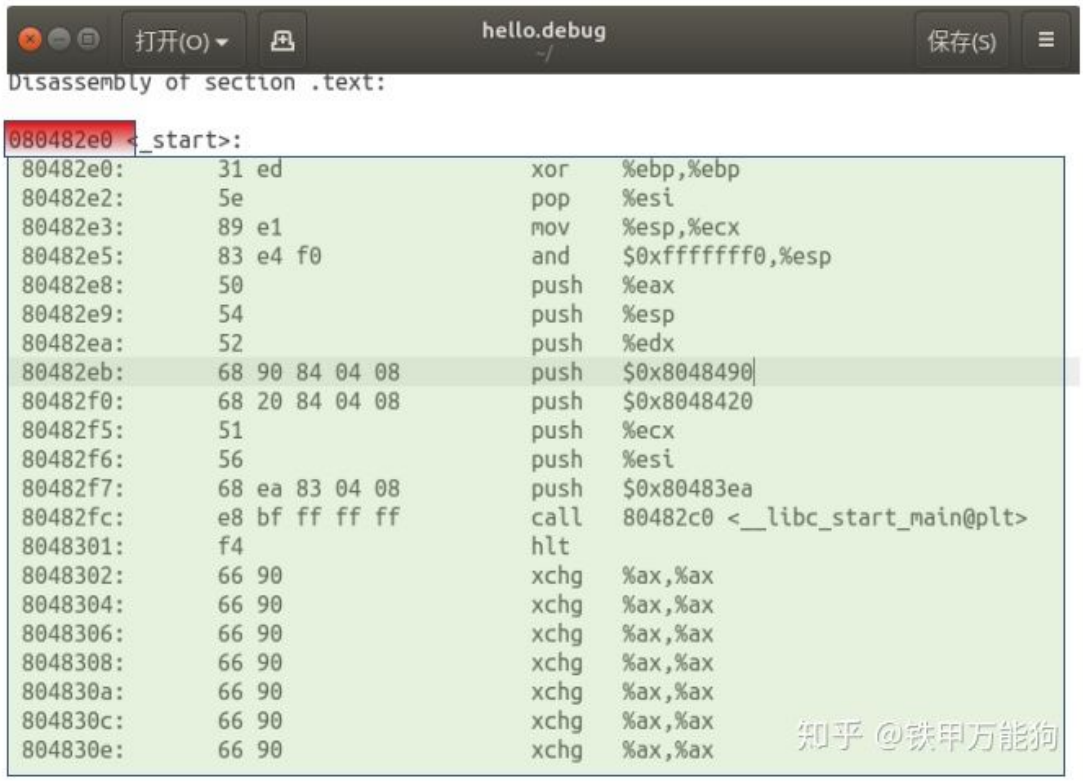
我们得到很多信息,首先我们搜索关键字main函数,我们会发现main函数的内存地址是**0x80483ea**,如图所示。

```
 80483de:       89 e5                    mov    %esp,%ebp
 80483e0:       8b 45 0c                 mov    0xc(%ebp),%eax
 80483e3:       8b 55 08                 mov    0x8(%ebp),%edx
 80483e6:       01 d0                    add    %edx,%eax
 80483e8:       5d                       pop    %ebp
 80483e9:       c3                       ret

080483ea <main>:
 80483ea:       55                       push   %ebp
 80483eb:       89 e5                    mov    %esp,%ebp
 80483ed:       83 ec 18                 sub    $0x18,%esp
 80483f0:       c7 45 fc 0b 00 00 00     movl   $0xb,-0x4(%ebp)
 80483f7:       c7 45 f8 17 00 00 00     movl   $0x17,-0x8(%ebp)
 80483fe:       8b 45 f8                 mov    -0x8(%ebp),%eax
 8048401:       89 44 24 04              mov    %eax,0x4(%esp)
 8048405:       8b 45 fc                 mov    -0x4(%ebp),%eax
 8048408:       89 04 24                 mov    %eax,(%esp)
 804840b:       e8 cd ff ff ff           call   80483dd <add>
 8048410:       89 45 f8                 mov    %eax,-0x8(%ebp)
 8048413:       b8 00 00 00 00           mov    $0x0,%eax
 8048418:       c9                       leave
 8048419:       c3                       ret
 804841a:       66 90                    xchg   %ax,%ax
 804841c:       66 90                    xchg   %ax,%ax
 804841e:       66 90                    xchg   %ax,%ax

08048420 <__libc_csu_init>:
 8048420:       55                       push   %ebp
```


纯文本 ▾ 制表符宽度：8 ▾ 第 150 行，第 57 列 ▾ 插入

再进一步，通过80483ea这个关键字，我们进一步搜索，发现几个关键的信息：

- main函数对应的内存地址落在<_start>这个代码段，而<_start>这个标签是整个汇编程序的全局入口，这类似于C程序的main函数。
- 我们还发现在<_start>内部main的函数地址0x80483ea也被压入栈。如下图所示。
- **<_start>**标签的内存地址是0x80482e0,这个地址可以作为我们进一步顺藤摸瓜的的条件。

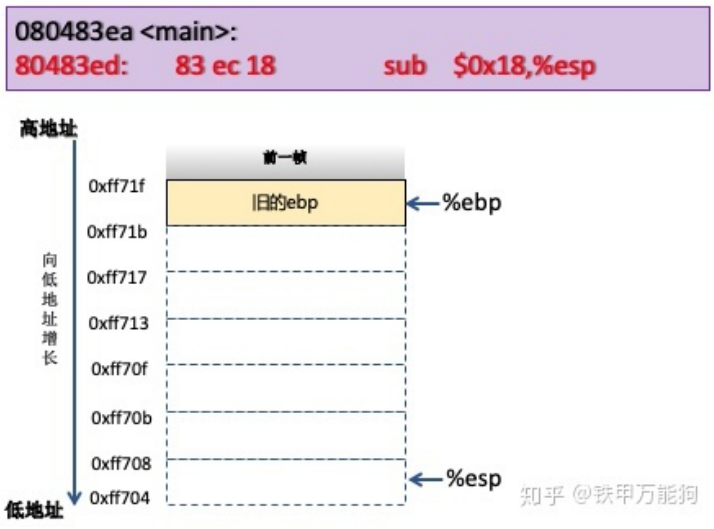


调用了__blibc_start_main等相关底层库函数用于支持我们main函数的执行。因为我们这里讨论的是跟用户定义的函数相关的话题，C底层的函数不是我们考虑的问题。

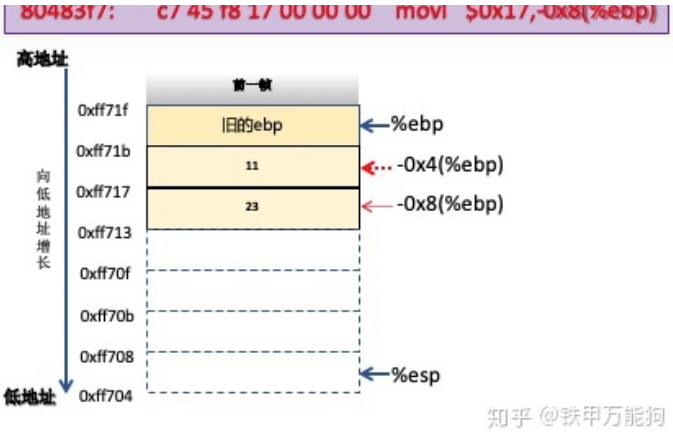
main函数的栈帧

- **push %esp**指令是用于前面一帧的ebp指针地址压入栈,与此同时%esp会从原来前一帧的位置向低地址下移始终指向最新的**栈顶**(暗含的指令**subl 1,%esp**)
- **push %esp**实际上就是当前栈帧对前一帧状态的一个备份操作。以便能够在当前栈还原前一帧的ebp指针。
- **mov %esp,%ebp**将%esp的值覆盖%ebp的值，即当前esp指针指向的位置就是当面main函数的栈底。

- **sub $0x18,%esp**,对esp指针中的地址值作减法操作esp下移动24个字节,这实质上已经改变esp指针中的地址值，这样做的目的是**为我们接下来main内部的局部变量和要调用的被调用函数“add”需要用到的参数列表分配所需的栈空间。**

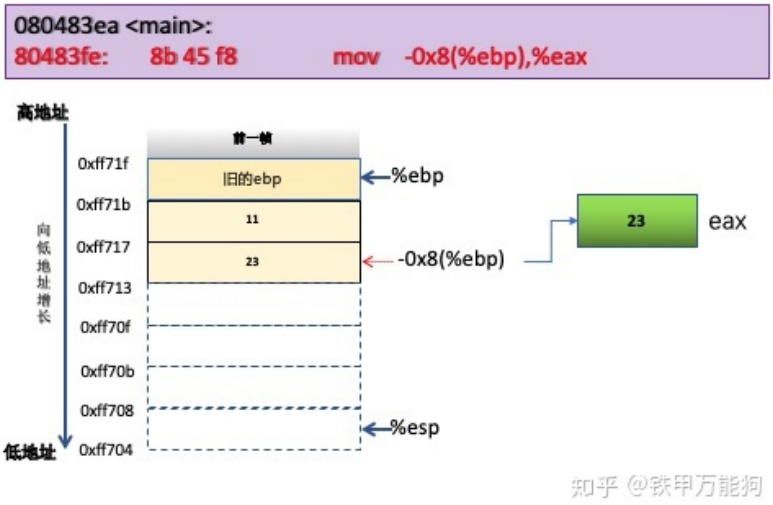


- **movl $0xb,-4(%ebp)** 和**movl $0x17,-4(%ebp)**分别将变量值11和23写入main栈内已分配的空间，这两条指令通过对ebp相对寻址的方式找到局部变量的栈内存位置,相对寻址并没有改变%ebp中的地址值。



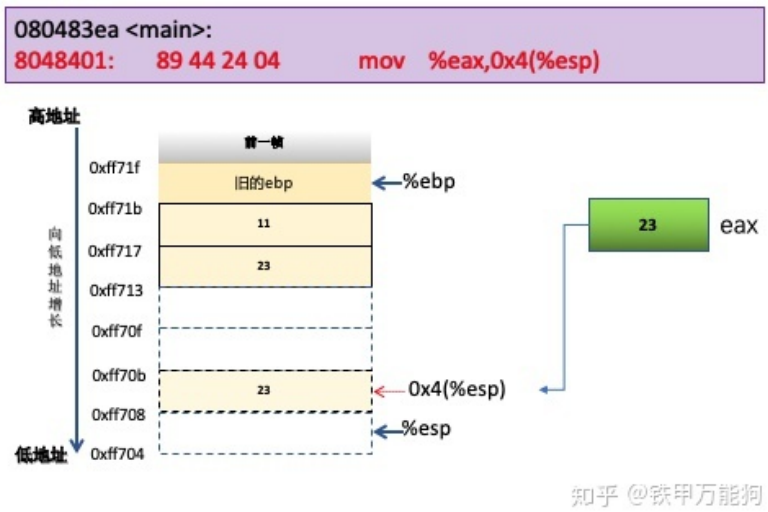
分配局部变量

- **movl -4(%ebp),%eax**指令就是将第二个局部变量缓存到eax寄存器中。

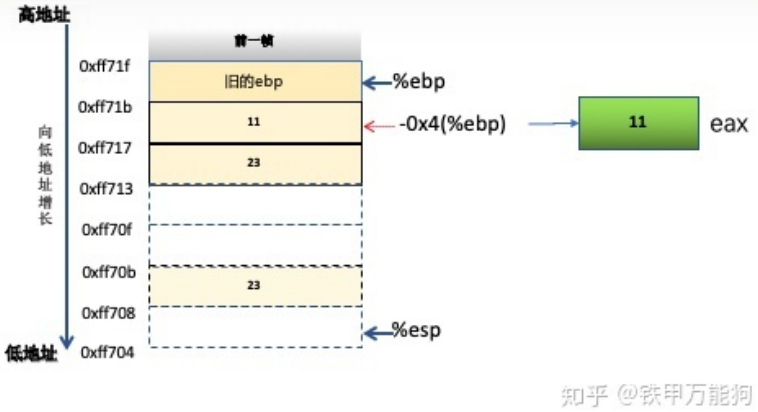


创建参数列表

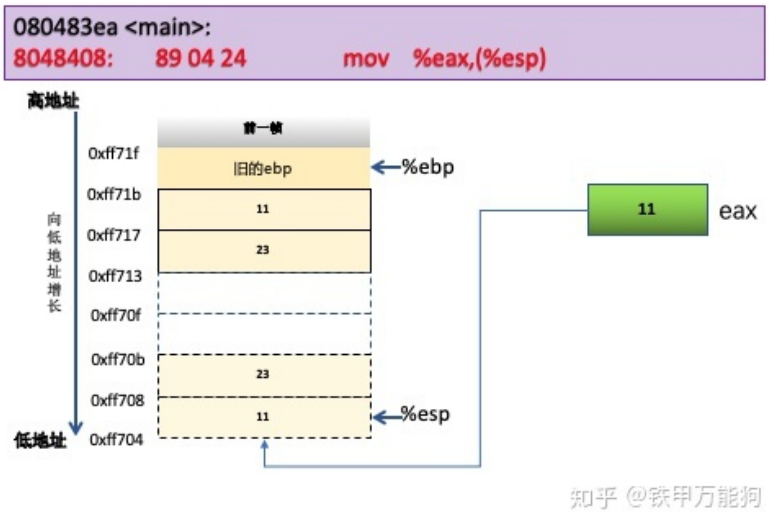
- **movl %eax,0x4(%esp)**指令将刚才eax缓存的变量值分配到栈的参数域。



- 接着下面的两条指令跟上面的两条都是做相同的事情，寄存器只是一个变量值的一个中转站。



- 到这里我们在main栈帧中参数区的参数表和局部变量中的顺序是相反的。
- 之前main栈帧在设置代码的阶段在分配了24个字节的空间，但实际上目前我们用到只有了16个字节，这种情况是非常常见的。但许多的文章根本就没向读取明确说明这一点。另外一种比较复杂的情况，如果入栈的是一个**用户自定义的数据类型**的struct，可能会和其他基本数据类似进行内存对齐，出去空的字节块的情况是非常频繁的。



- **call 80483dd**就是执行add函数的地址，该地址位于80483dd的位置。如果你有看前文，call指令等价如下几个以下几条指令操作
  - call指令的下一条指令的地址会作为返回地址入栈(即:**push 8048410**)。
  - esp指针指向栈顶(即**subl $1,%esp**)。
  - 最后隐式执行jmp 80483dd这条指令 ,此时将从main函数的上下文跳转到被调用函数add的上下文。

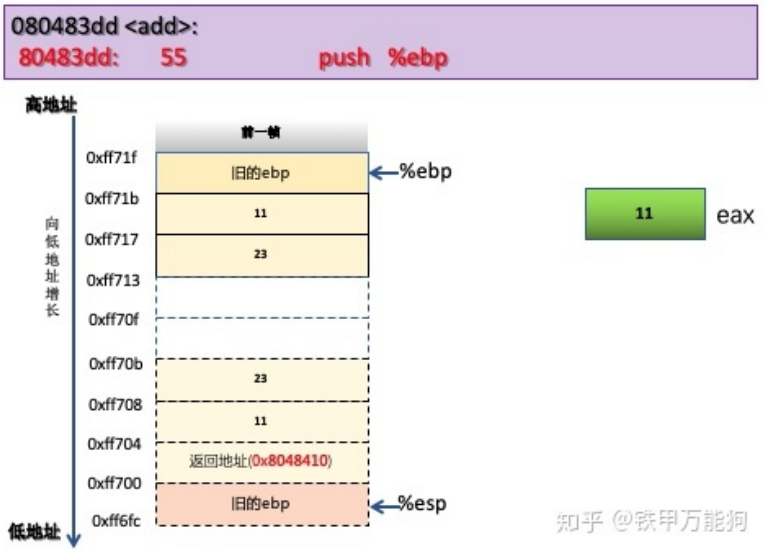
返回地址入栈

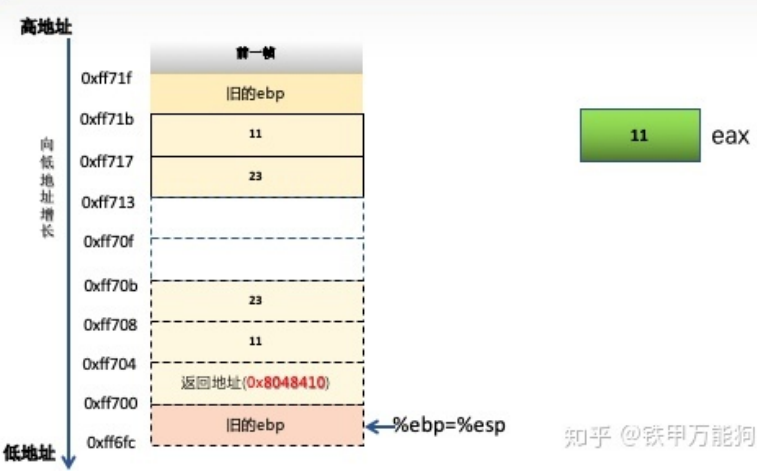
add函数的栈帧

当main执行call指令后，跳转到add函数的上下文，如下图

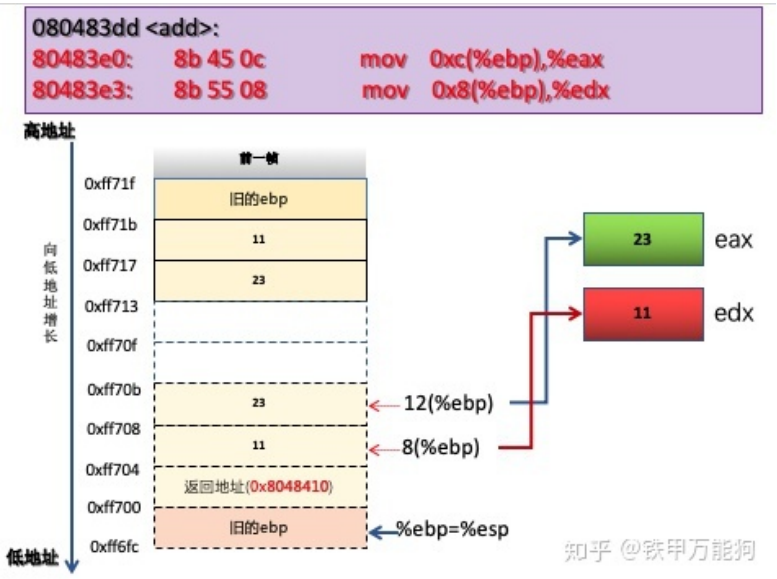
add函数的上下文

- 这里关于add函数的**设定代码**部分，同理跟之前main函数的设定代码部分的分析没多大出入。

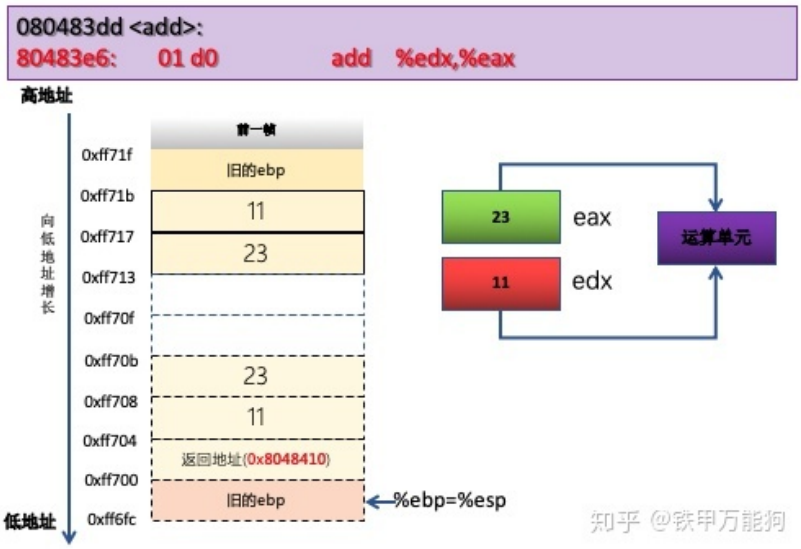


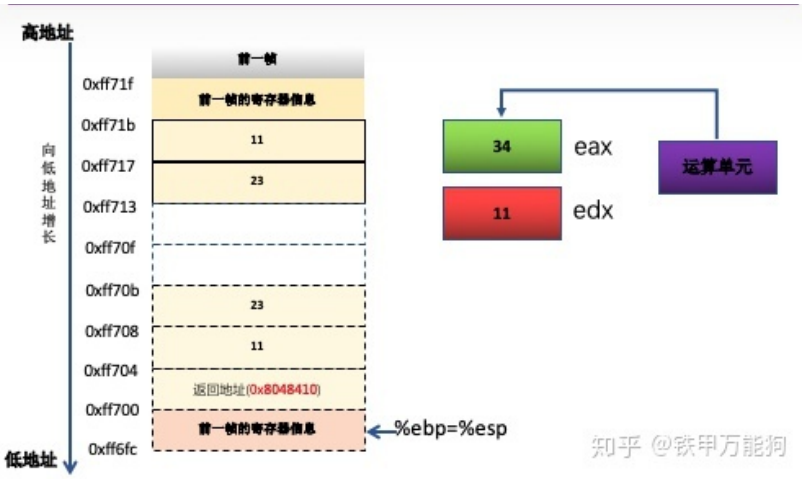


- **mov 0xc(%ebp),%eax** 和**mov 0x8(%ebp),%eax**分别从上一帧main的栈帧获取参数23和参数11，分别加载到寄存器eax和寄存器edx，以便下一步计算使用，如下图所示。



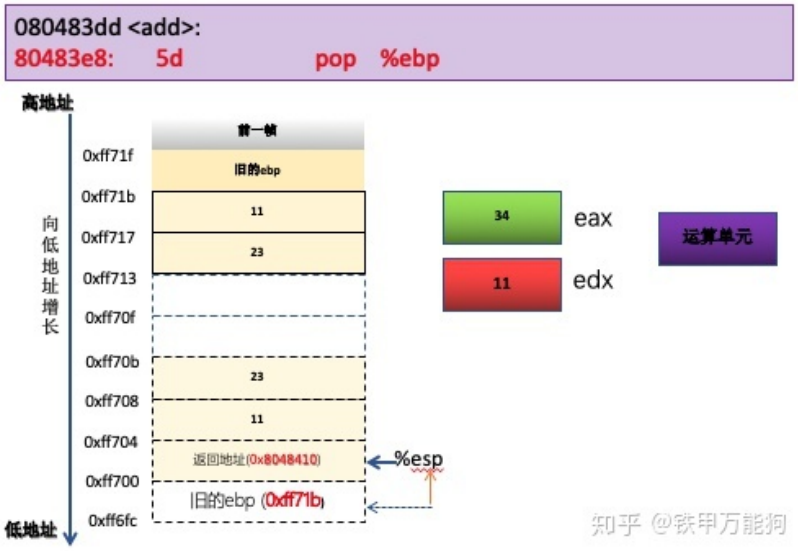
- **add %edx,%eax**指令和刚才寄存器edx和寄存器eax的参数值副本执行加法运算,**计算后的结果将保存到eax寄存器。**





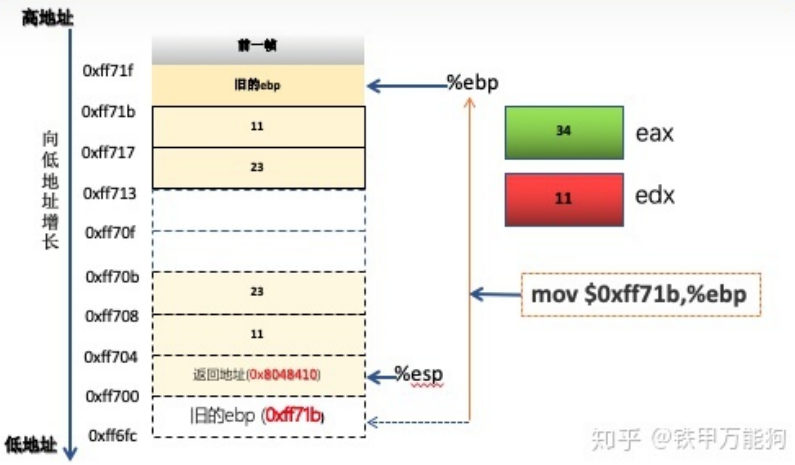
- **pop %ebp**指令其实弹出上一栈main栈帧的ebp地址告知CPU让寄存器ebp**将(注意我用的字眼,表示还没有执行完成的状态)**重新指向main栈帧的栈底(即:图中我假设的地址0xff71b)，实质上pop等价如下几条指令的操作。

1. **mov -(%esp),%esp** :esp将指向0xff700即上一帧main的**返回地址**所在帧的内存地址。



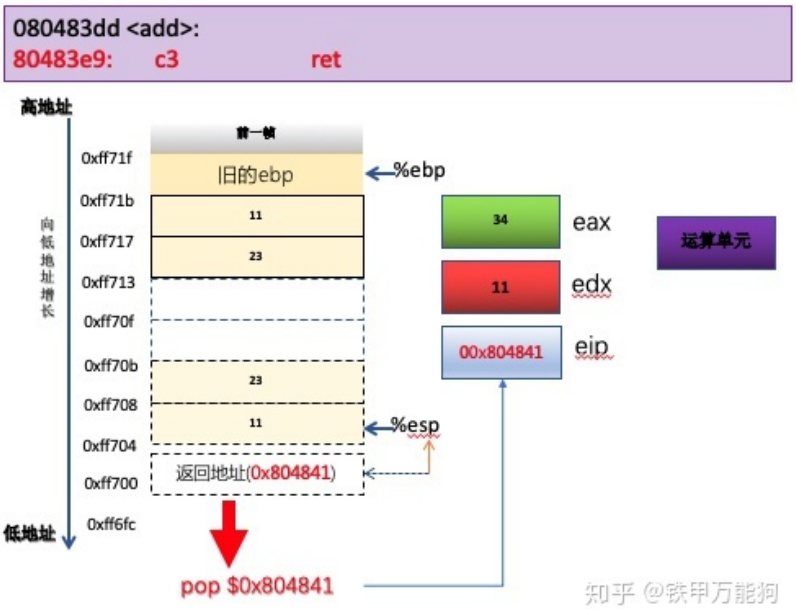
mov -(%esp),%esp

2. **mov $0xff71b,%ebp** :被弹出的0xff71b的地址值会覆盖ebp目前的内容，此时ebp会指向如图中的0xff71b



mov $0xff71b,%ebp执行时的状态

- **ret**指令主要隐式地执行两条指令
  - **pop 0x804841**从栈中弹出这个返回地址
  - **mov 0x804841,%eip**这条指令会将返回地址覆盖eip寄存器中的值，在没有语句之时,但尚未转移到返回地址的瞬间状态如下图。

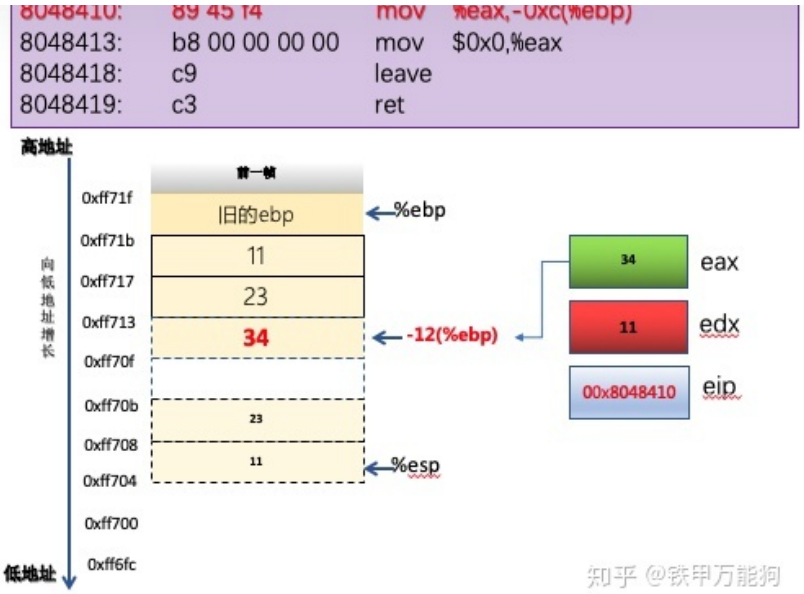


ret指令

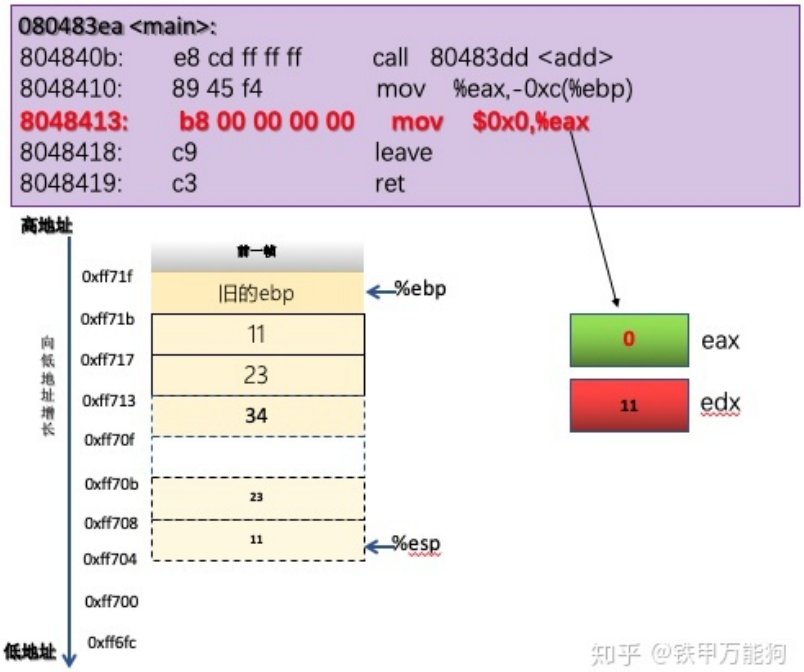
按照惯例，被调用者函数的返回值会放在eax寄存器中，eax的选择是相当随意的，可能是%ecx或%edx等，具体根据不同的C/C++编译器的实现而定。

被调用者函数在执行ret指令时,会将(计算过)的适合4个字节的任意类型的返回值保存到(通常是%eax)寄存器,也可能是其他寄存器, x86环境中的eax寄存器只有4个字节。如果要返回大于4个字节的数据类型，最好的方式是返回一个自定义类型的对象的指针，而不是对象本身。

- **返回**时，调用者函数在%eax寄存器(也可能是其他寄存器)中找到返回值。如下图,我们的main函数栈帧，会将eax寄存器的返回值保存到便来那个域当中。



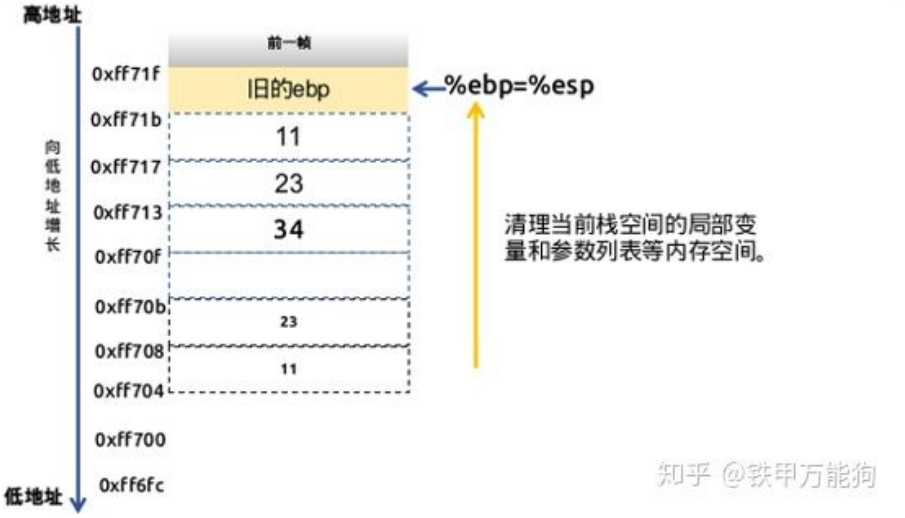
- 当然，**main函数也会存在返回值**的情况，下面同样的情况，main函数在执行ret指令之前也将返回值0,写入的寄存器eax。



main的返回值也被加载到eax寄存器，等待返回给更上一帧的C/C++库函数

- 最后要说的指令就是leave指令，它隐式等价执行如下语句，他们用于清理main栈帧中的所有局部变量和参数等占用的栈空间。
  - **mov %ebp,%esp**
  - **pop %ebp**

```
8048410:    89 45 f4          mov    %eax,-0xc(%ebp)
8048413:    b8 00 00 00 00    mov    0x0,%eax
8048418:    c9                leave
8048419:    c3                ret
```



leave指令的执行

后记

本篇的解决了前一篇提出的许多问题点，并以详实的例子覆盖了栈的大部分话题。目前没有具体提及到寄存器保存约定的细节，因为用到例子的计算就两个操作数字的加法运算本来就不需要栈将寄存器的状态数据进行入栈操作。而实际上，在高层语言设计复杂的算法的时候当转译成汇编的代码不外乎乘除加减，位运算等一系列基础的运算操作以及大量的加载和存储的指令集(move),那么必然涉及到被调用函数在call被调用函数之前，需要寄存器作为操作数参与一系列的运算和存储计算的中间结果，但CPU中的物理寄存器数量是有限的资源，并且同一个寄存器也可能被其他线程中的函数向其写入数据，那么之前计算的计算结果必然遭到破坏。因此就有了栈寄存器状态执行入栈保存其数据状态，等到需要的时候，再从栈内弹出以供调用函数使用，那么这个话题后面有空会慢慢补上。

发布于 2020-08-15 03:10

寄存器　汇编语言　C / C++

文章被以下专栏收录

C/C++内存管理
侧重C/C++内存模型，反编译分析

推荐阅读

第4篇:C/C++ 结构体及其数组的内存对齐

数据对象本节谈及的内存对齐,而在进行这个话题之前，我先引入

C 程序运行时内存结构分析！秀儿，这操作不是我等凡人能接...

一、实验知识 静态变量存储在静态存储区，局部变量存储在动态

概念就是数据对象(Data Object)，简称 对象,像基于C衍生出来其...
铁甲万能狗 发表于C/C++...

指向栈顶，EIP指向正在执行指令的下一条指令，三个寄存器中...
关于编程哪... 发表于C/C++...

C语言编程... 发表

1 条评论 切换为时间排序

写下你的评论...

呵呵哒 2021-01-23

通俗易懂

赞