本节内容

函数调用

机器级表示

—— call、ret 指令

高级语言的函数调用

```
void main(){
   ...
   P();
   ...
   return;
}
```

```
void P(){
   ...
   Q();
   ...
   caller();
   ...
   return;
}
```

```
void Q(){
   ...
   return;
}
```

```
int caller(){
   int temp1=125;
   int temp2=80;
   int sum=add(temp1,temp2);
   return sum;
}
```

函数调用栈
（一片内存区域）

```
int add(int x, int y){
   return x+y;
}
```

函数的栈帧（Stack Frame）：保存函数大括号内定义的局部变量、保存函数调用相关的信息

高级语言的函数调用

```
void main(){
   ...
   P();
   ...
   return;
}
```

```
void P(){
   ...
   Q();
   ...
   caller();
   ...
   return;
}
```

```
void Q(){
   ...
   return;
}
```

```
int caller(){
   int temp1=125;
   int temp2=80;
   int sum=add(temp1,temp2);
   return sum;
}
```

```
int add(int x, int y){
   return x+y;
}
```

函数调用栈
（一片内存区域）

函数的栈帧（Stack Frame）：保存函数大括号内定义的局部变量、保存函数调用相关的信息

函数调用指令：　call <函数名>
函数返回指令：　ret

x86汇编语言的函数调用

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

通常用函数名作为函数起始地址的<标号>


```
int caller(){
  int temp1=125;
  int temp2=80;
  int sum=add(temp1,temp2);
  return sum;
}
```

```
int add(int x, int y){
  return x+y;
}
```

call、ret指令

注：x86处理器中
程序计数器 PC （Program Counter）
通常被称为 IP（Instruction Pointer）

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

IP旧值→

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

IP新值→

过得去

回得来

通常用函数名作为函数起始地址的<标号>

函数调用指令：　call <函数名>
函数返回指令：　ret

call 指令的作用：
①将IP旧值压栈保存（保存在函数的栈帧顶部）
②设置IP新值，无条件转移至被调用函数的第一条指令

ret 指令的作用：
从函数的栈帧顶部找到 IP旧值，将其出栈并恢复IP寄存器

总结：函数调用的机器级表示

调用者：

……

- 执行 call 指令

……

返回地址压入栈顶、并跳转到被调用函数第一条指令

被调用者：

……

- 一系列处理逻辑
- 执行 ret 指令

从栈顶找到返回地址，出栈并恢复 IP 值

call 指令的作用：
①将IP旧值压栈保存（保存在函数的栈帧顶部）
②设置IP新值，无条件转移至被调用函数的第一条指令

ret 指令的作用：
从函数的栈帧顶部找到 IP旧值，将其出栈并恢复IP寄存器

小朋友，你是否有很多问号？

```
int caller(){
   int temp1=125;
   int temp2=80;
   int sum=add(temp1,temp2);
   return sum;
}
```

```
int add(int x, int y){
   return x+y;
}
```

如何传递调用参数、返回值？

如何访问栈帧里的数据？

栈帧内可能包含哪些内容？

为什么倒过来了？

本节内容

函数调用

机器级表示

——如何访问栈帧？

小朋友，你是否有很多问号？

```
int caller(){
   int temp1=125;
   int temp2=80;
   int sum=add(temp1,temp2);
   return sum;
}
```

```
int add(int x, int y){
   return x+y;
}
```

栈为什么倒过来了？

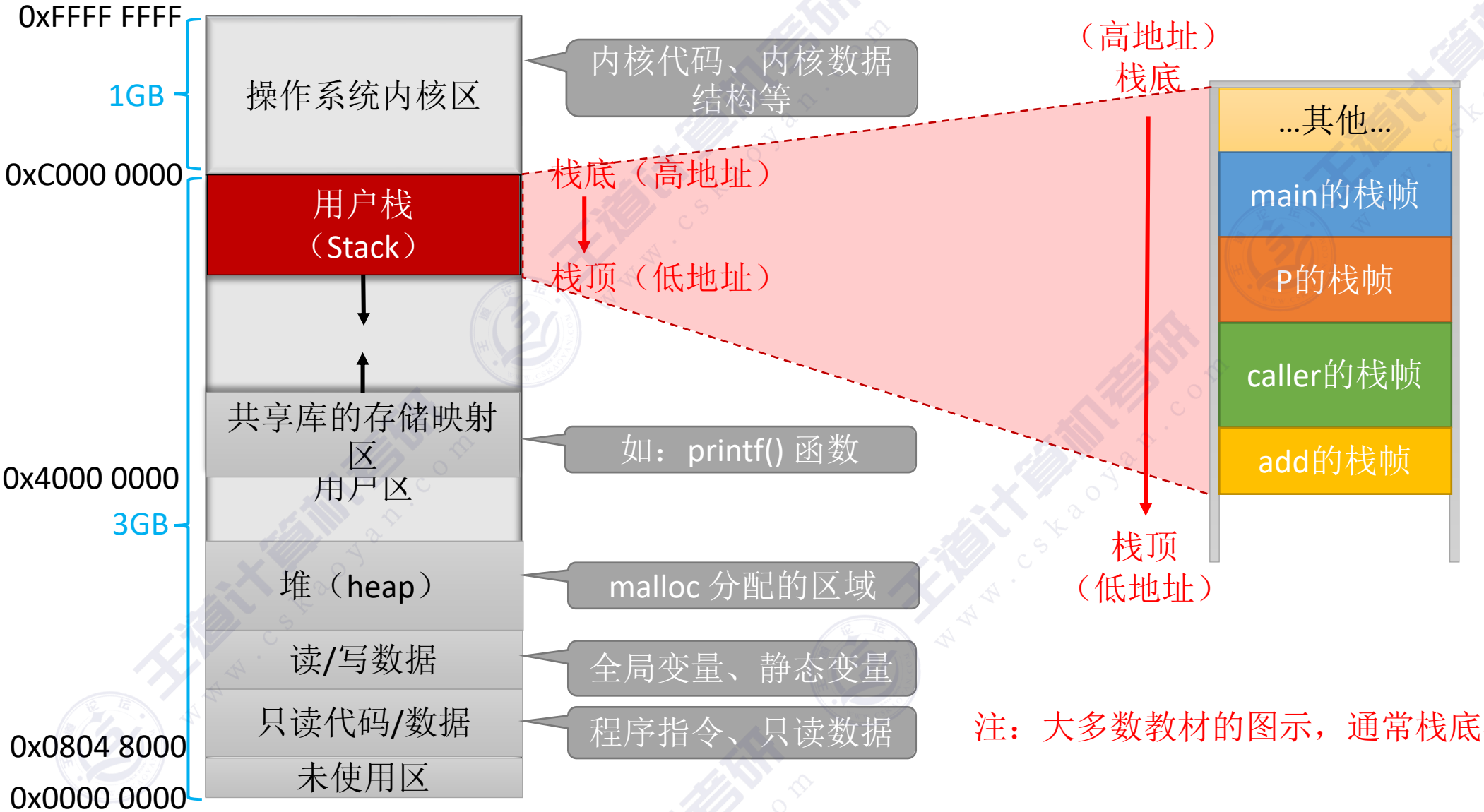
如何传递调用参数、返回值？

如何访问栈帧里的数据？

栈帧内可能包含哪些内容？

函数调用栈在内存中的位置

32位系统，进程虚拟地址空间为4GB



注：大多数教材的图示，通常栈底在上，栈顶在下

标记栈帧范围：EBP、ESP寄存器

每个寄存器都是32bit

31 0

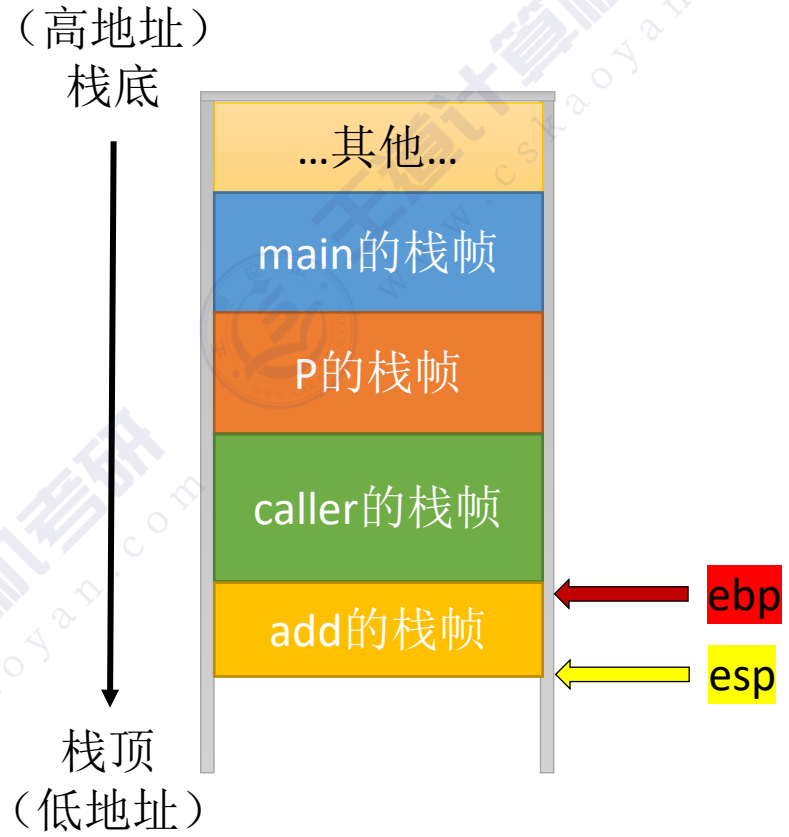
EAX
EBX
ECX
EDX
ESI
EDI
EBP
ESP

通用寄存器（X = 未知）
E = Extended = 32bit

变址寄存器（I = Index）
S = Source，D=Destination

堆栈基指针（Base Pointer）

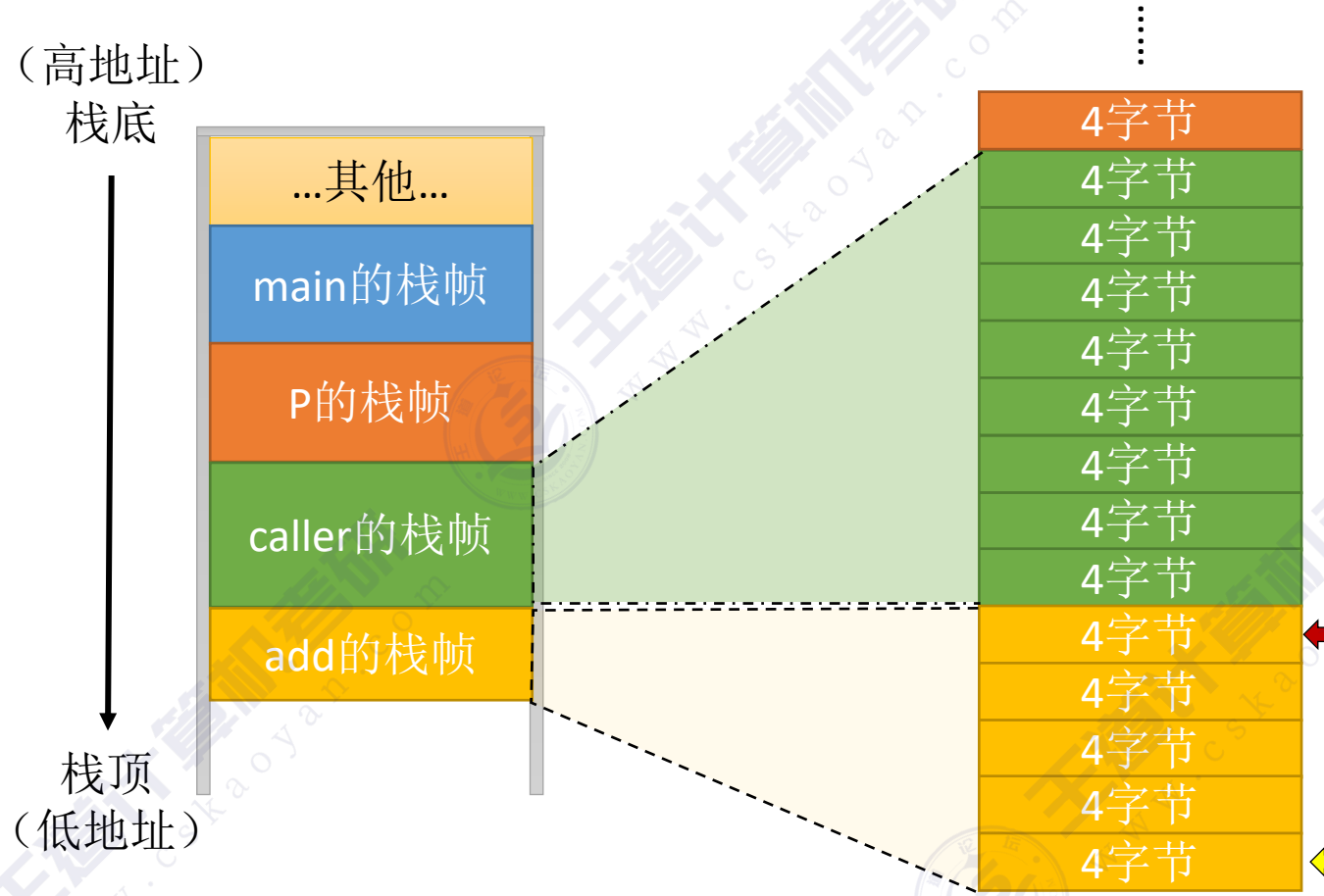
堆栈顶指针（Stack Pointer）



ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

标记栈帧范围： EBP、ESP寄存器

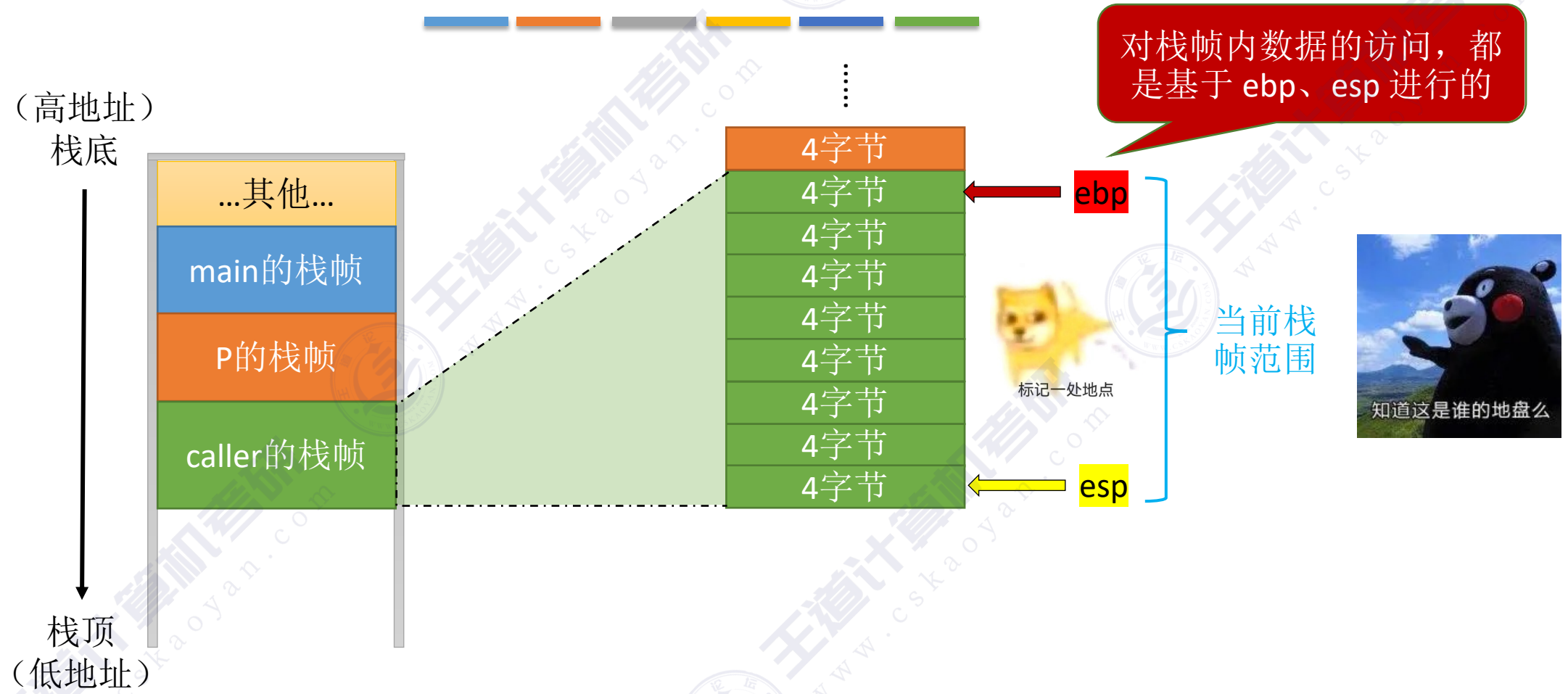


ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

注：x86 系统中，默认以4字节为栈的操作单位

标记栈帧范围： EBP、ESP寄存器



ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

注：x86 系统中，默认以4字节为栈的操作单位

访问栈帧数据：push、pop 指令

eax寄存器：211

push、pop 指令实现入栈、出栈操作，x86 默认以4字节为单位。指令格式如下：

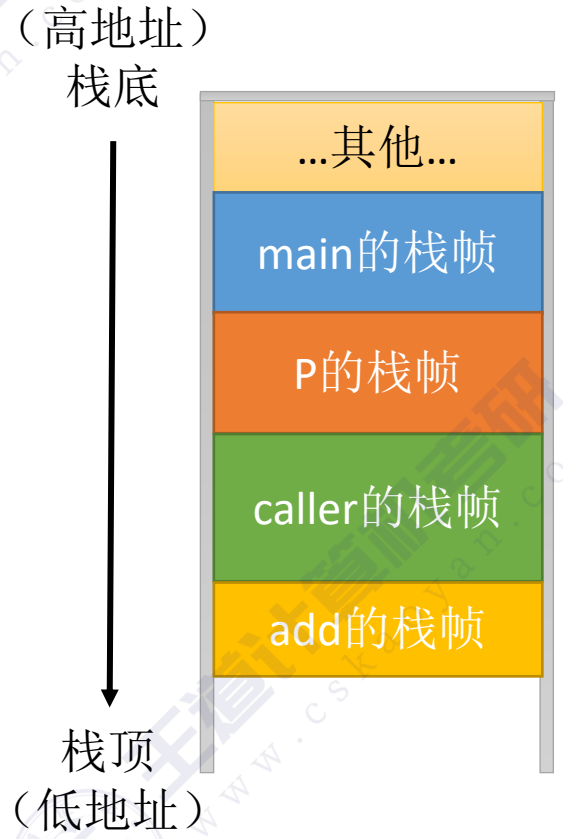
Push 🐶　　// 先让esp减4，再将 🐶 压入
Pop 🐴　　//栈顶元素出栈写入 🐴，再让 esp加4

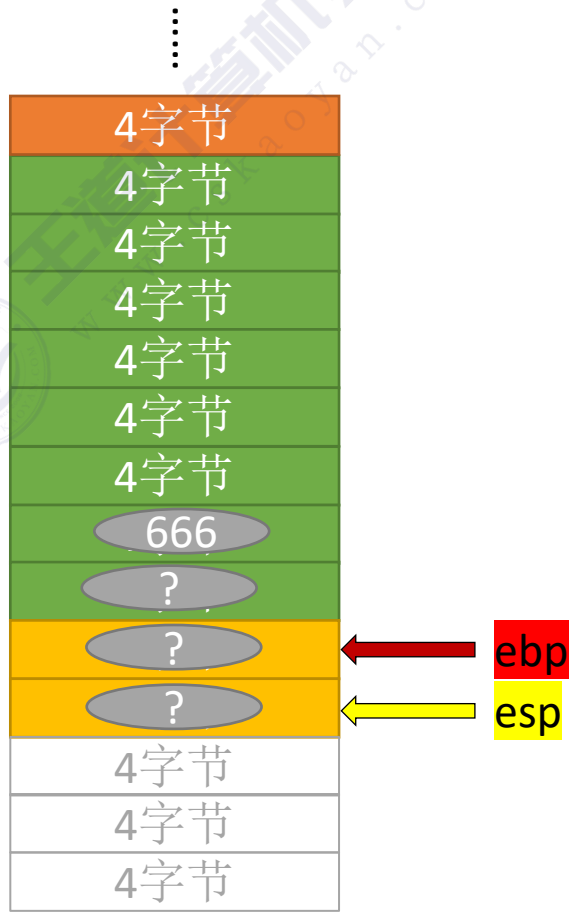
注1：🐶 可以是立即数、寄存器、主存地址
注2：🐴 可以是寄存器、主存地址

例：

```
push eax          #将寄存器eax的值压栈
push 985          #将立即数985压栈
push [ebp+8]      #将主存地址[ebp+8]里的数据压栈

pop eax           #栈顶元素出栈，写入寄存器eax
pop [ebp+8]       #栈顶元素出栈，写入主存地址[ebp+8]
```





ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

访问栈帧数据：push、pop 指令

eax寄存器：211

push、pop 指令实现入栈、出栈操作，x86 默认以4字节为单位。指令格式如下：

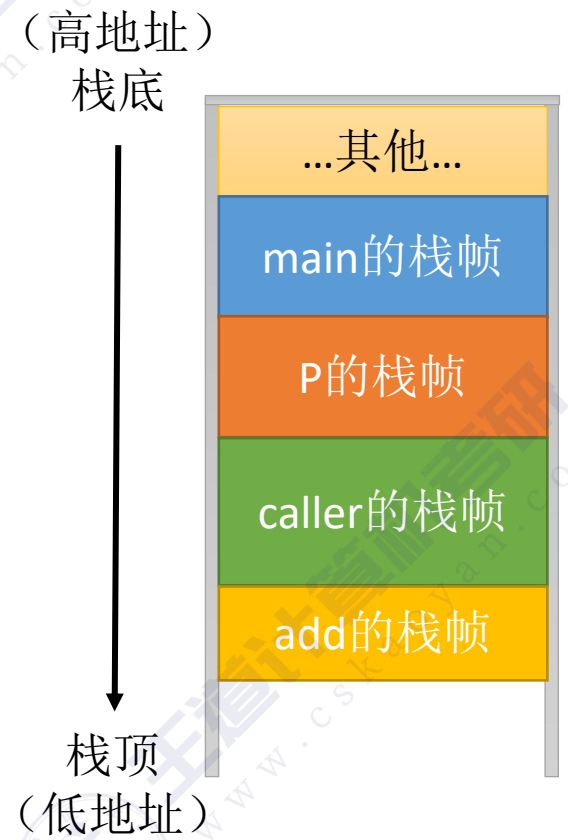
Push 🐶　　// 先让esp减4，再将 🐶 压入
Pop 🐴　　//栈顶元素出栈写入 🐴，再让 esp加4

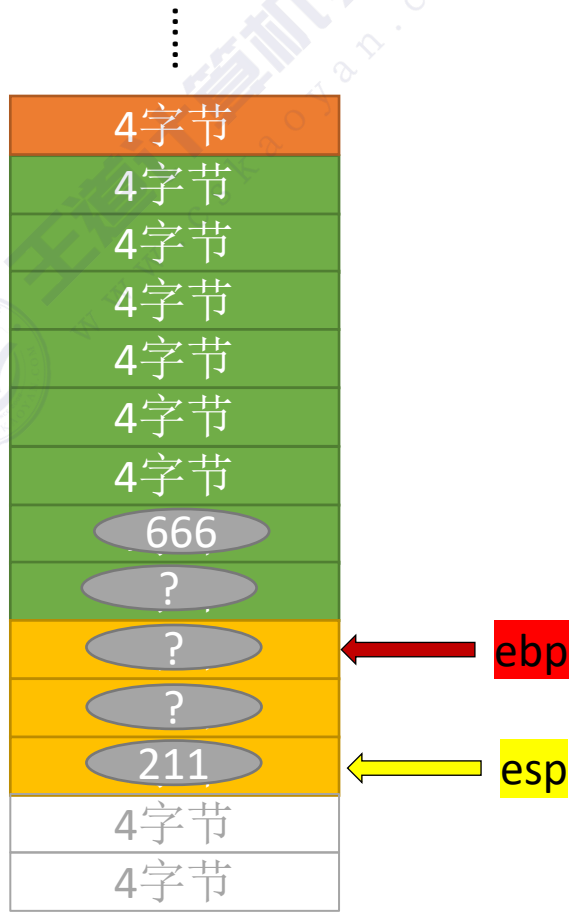
注1：🐶 可以是立即数、寄存器、主存地址
注2：🐴 可以是寄存器、主存地址

例：

```
push eax          #将寄存器eax的值压栈
push 985          #将立即数985压栈
push [ebp+8]      #将主存地址[ebp+8]里的数据压栈

pop eax           #栈顶元素出栈，写入寄存器eax
pop [ebp+8]       #栈顶元素出栈，写入主存地址[ebp+8]
```





ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

访问栈帧数据：push、pop 指令

push、pop 指令实现入栈、出栈操作，x86 默认以4字节为单位。指令格式如下：

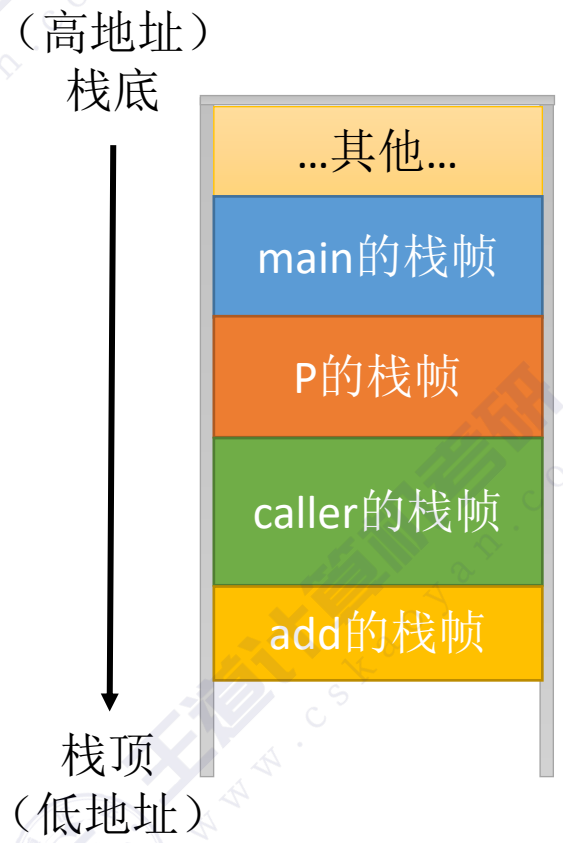
Push 🐶　　// 先让esp减4，再将 🐶 压入
Pop 🐴　　//栈顶元素出栈写入 🐴，再让 esp加4

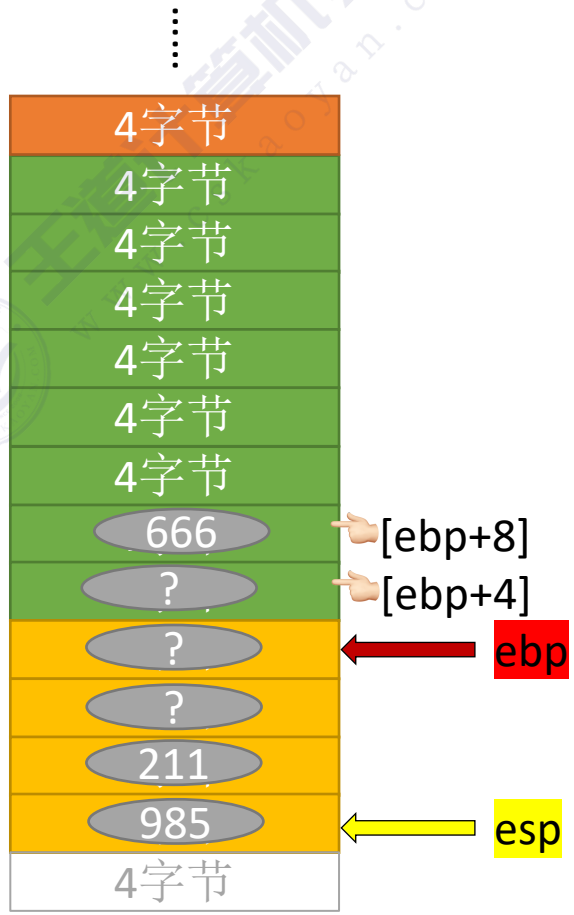
注1：🐶 可以是立即数、寄存器、主存地址
注2：🐴 可以是寄存器、主存地址

例：

```
push eax          #将寄存器eax的值压栈
push 985          #将立即数985压栈
push [ebp+8]      #将主存地址[ebp+8]里的数据压栈

pop eax           #栈顶元素出栈，写入寄存器eax
pop [ebp+8]       #栈顶元素出栈，写入主存地址[ebp+8]
```





ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

访问栈帧数据：push、pop 指令

eax寄存器：211

push、pop 指令实现入栈、出栈操作，x86 默认以4字节为单位。指令格式如下：

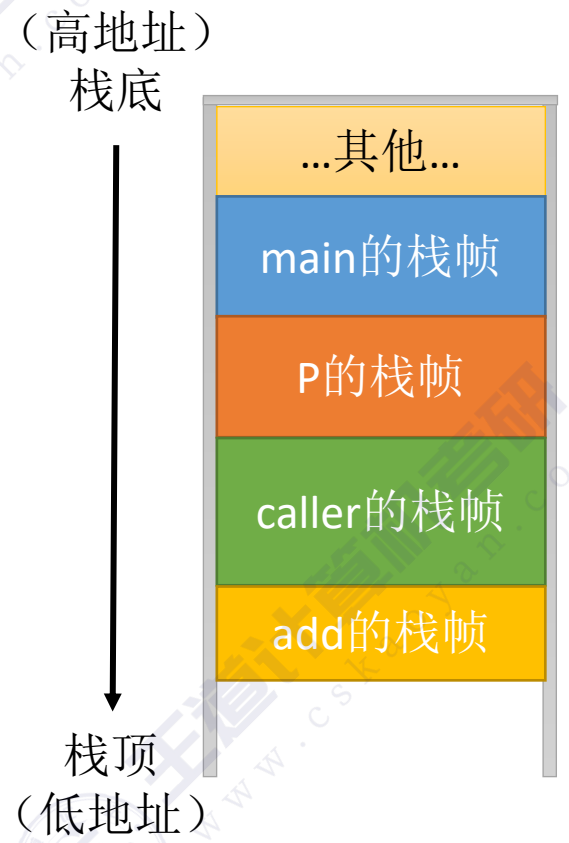
Push 🐶　　// 先让esp减4，再将 🐶 压入
Pop 🐴　　//栈顶元素出栈写入 🐴，再让 esp加4

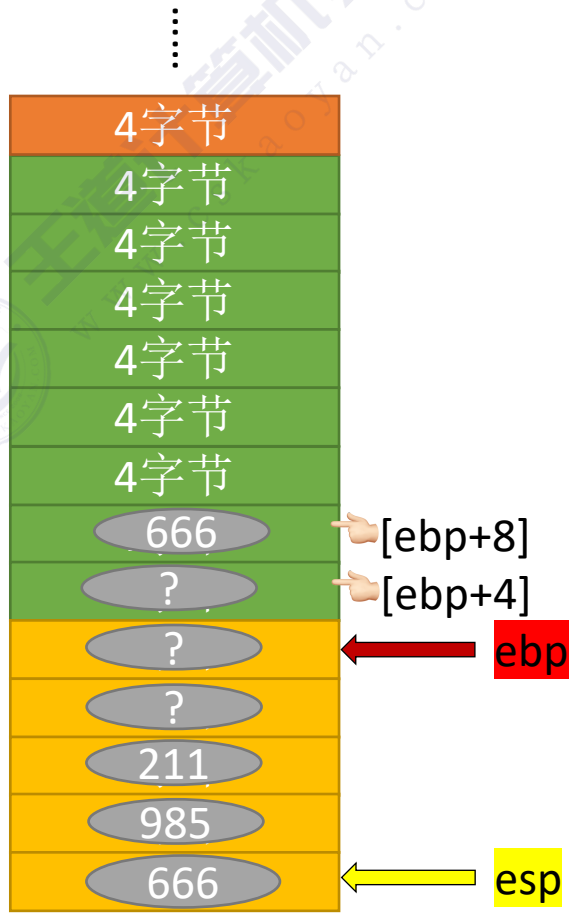
注1：🐶 可以是立即数、寄存器、主存地址
注2：🐴 可以是寄存器、主存地址

例：

```
push eax            #将寄存器eax的值压栈
push 985            #将立即数985压栈
push [ebp+8]        #将主存地址[ebp+8]里的数据压栈

pop eax             #栈顶元素出栈，写入寄存器eax
pop [ebp+8]         #栈顶元素出栈，写入主存地址[ebp+8]
```





ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

访问栈帧数据：push、pop 指令

push、pop 指令实现入栈、出栈操作，x86 默认以4字节为单位。指令格式如下：

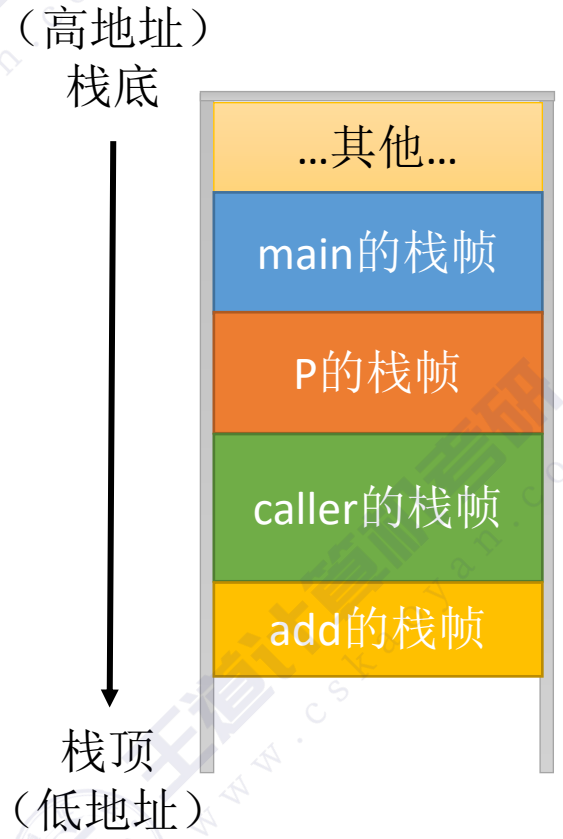
Push 🐶　　// 先让esp减4，再将 🐶 压入
Pop 🐴　　//栈顶元素出栈写入 🐴，再让 esp加4

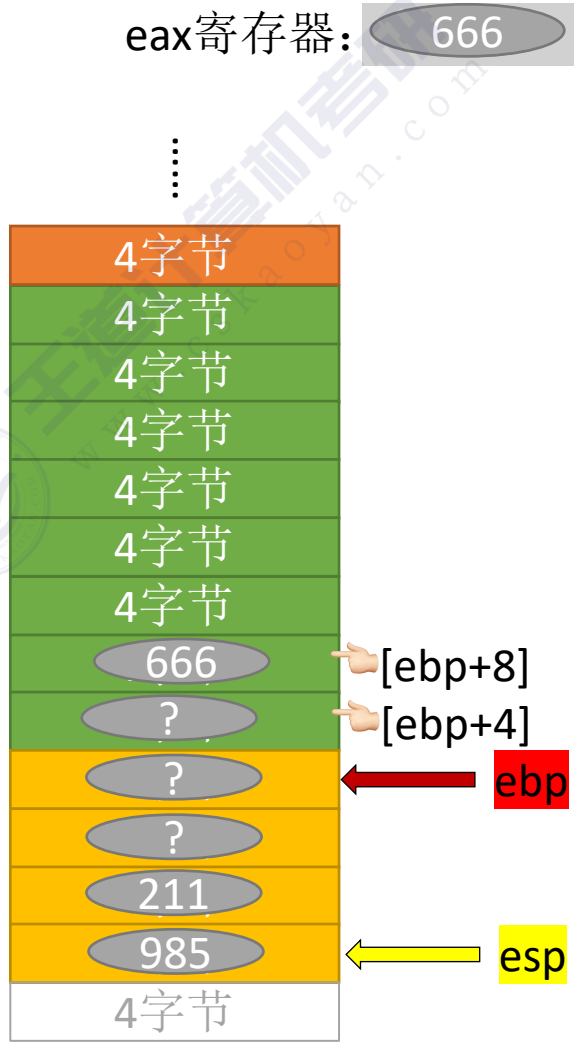
注1：🐶 可以是立即数、寄存器、主存地址
注2：🐴 可以是寄存器、主存地址

例：

```
push eax          #将寄存器eax的值压栈
push 985          #将立即数985压栈
push [ebp+8]      #将主存地址[ebp+8]里的数据压栈

pop eax           #栈顶元素出栈，写入寄存器eax
pop [ebp+8]       #栈顶元素出栈，写入主存地址[ebp+8]
```





ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

访问栈帧数据：push、pop 指令

push、pop 指令实现入栈、出栈操作，x86 默认以4字节为单位。指令格式如下：

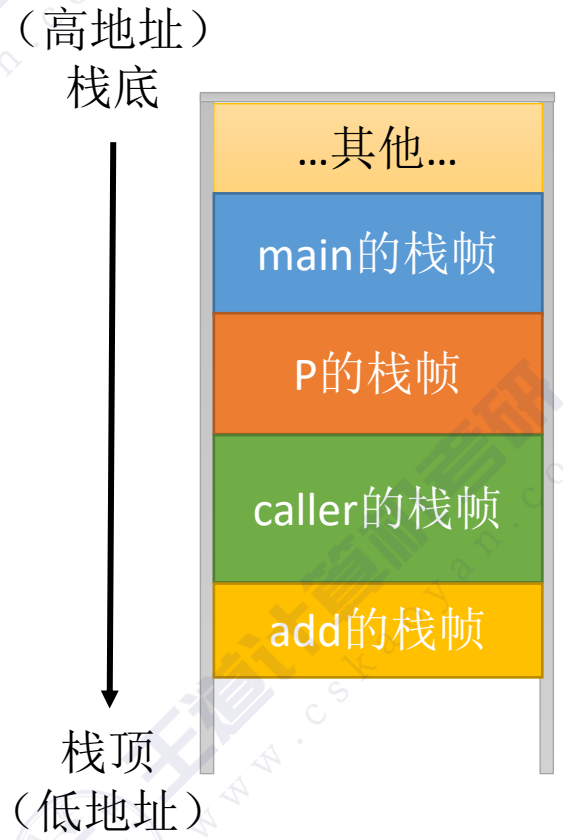
Push 🐶　　// 先让esp减4，再将 🐶 压入
Pop 🐴　　//栈顶元素出栈写入 🐴，再让 esp加4

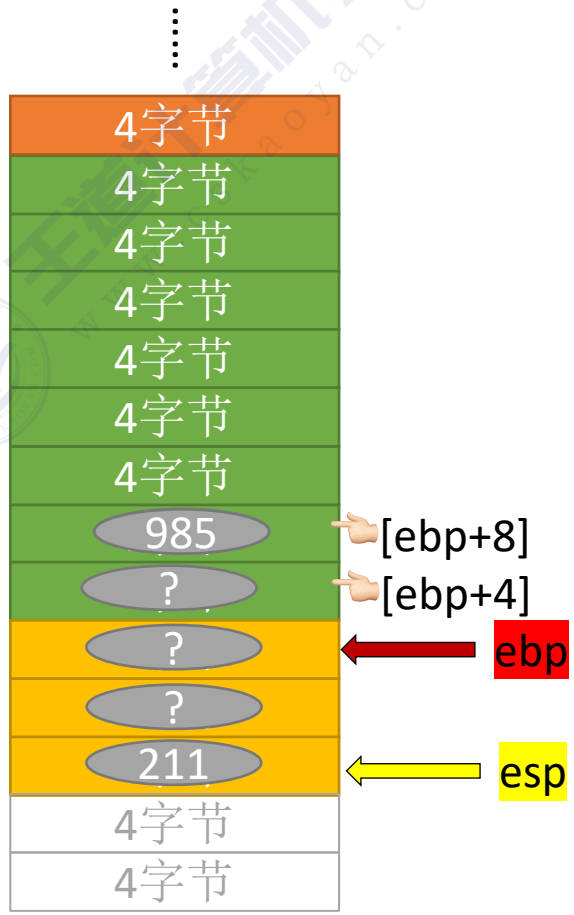
注1：🐶 可以是立即数、寄存器、主存地址
注2：🐴 可以是寄存器、主存地址

例：

```
push eax            #将寄存器eax的值压栈
push 985            #将立即数985压栈
push [ebp+8]        #将主存地址[ebp+8]里的数据压栈

pop eax             #栈顶元素出栈，写入寄存器eax
pop [ebp+8]         #栈顶元素出栈，写入主存地址[ebp+8]
```





ebp：指向当前栈帧的“底部”

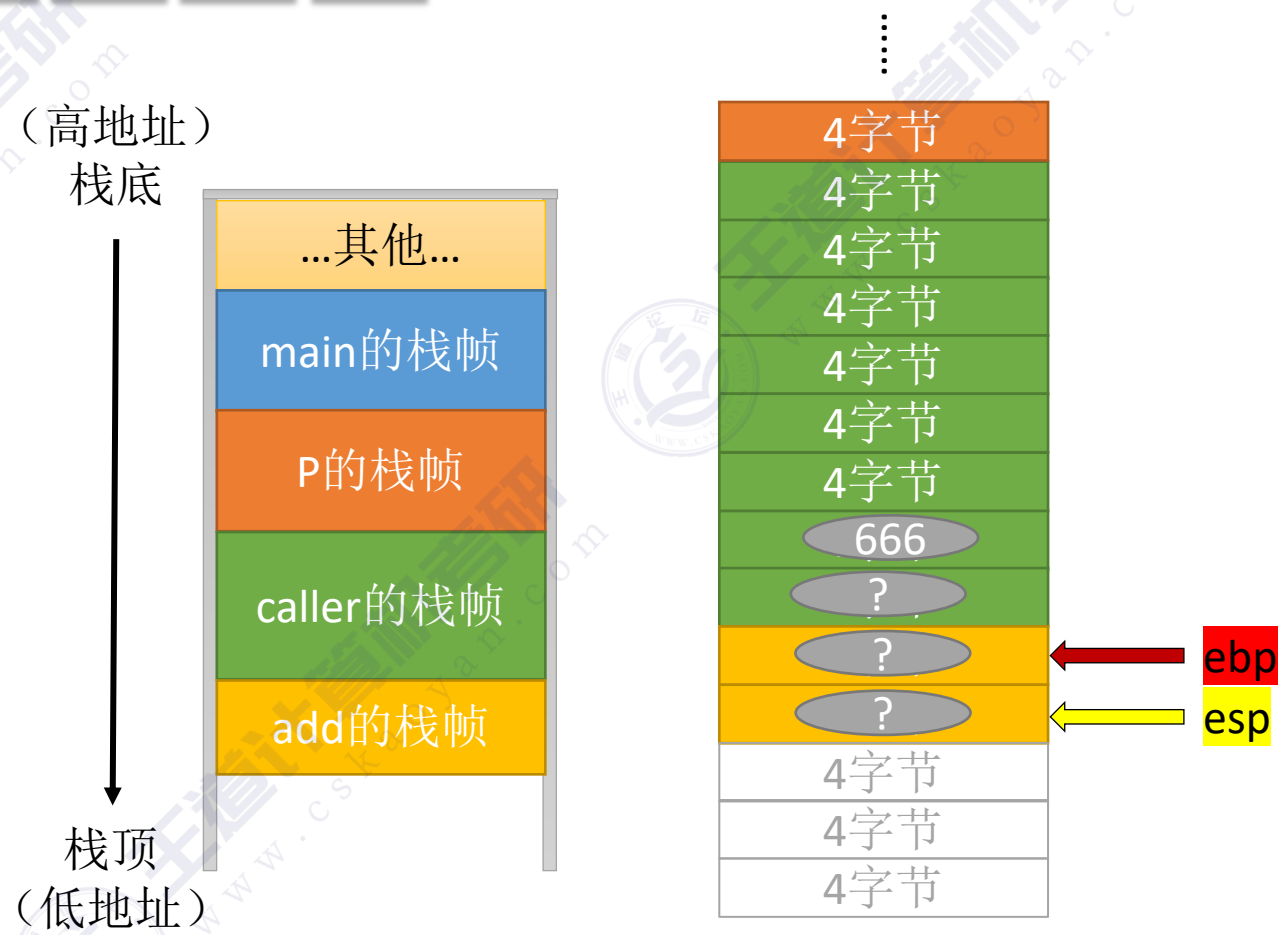
esp：指向当前栈帧的“顶部”

访问栈帧数据：mov 指令

eax寄存器：211

例：

```
sub esp, 12            #栈顶指针-12
mov [esp+8], eax       #将eax的值复制到主存[esp+8]
mov [esp+4], 958       #将985复制到主存[esp+4]
mov eax, [ebp+8]       #将主存[ebp+8]的值复制到eax
mov [esp],eax          #将eax的值复制到主存[esp]
add esp, 8             #栈顶指针+8
```



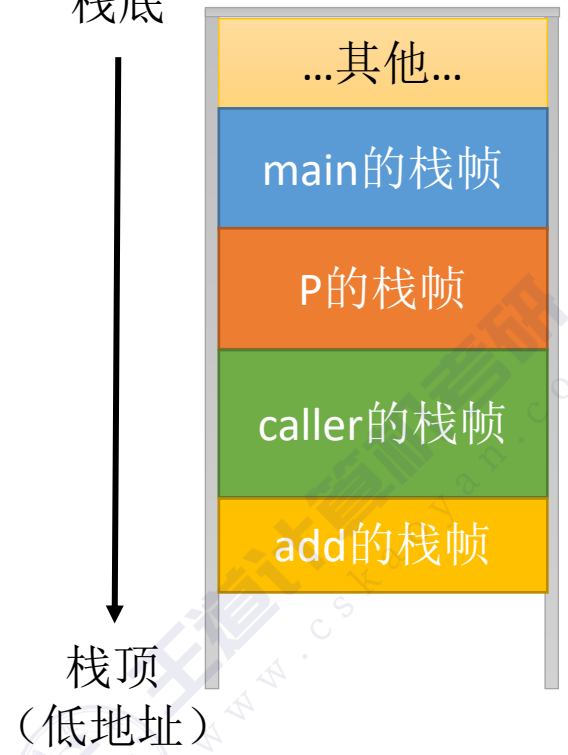
ebp：指向当前栈帧的“底部”

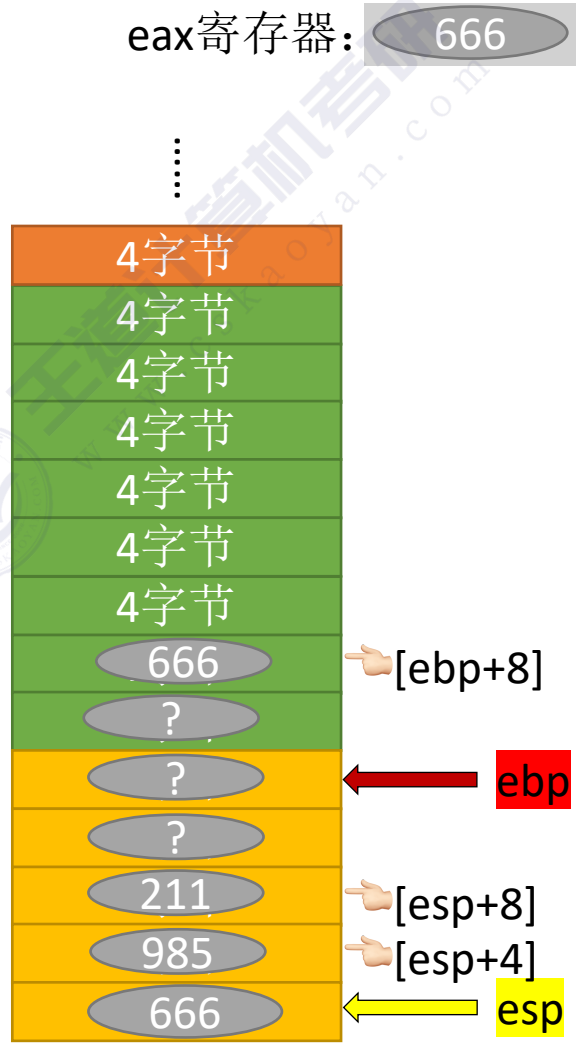
esp：指向当前栈帧的“顶部”

访问栈帧数据：mov 指令

例：

```
sub esp, 12              #栈顶指针-12
mov [esp+8], eax         #将eax的值复制到主存[esp+8]
mov [esp+4], 958         #将985复制到主存[esp+4]
mov eax, [ebp+8]         #将主存[ebp+8]的值复制到eax
mov [esp],eax            #将eax的值复制到主存[esp]
add esp, 8               #栈顶指针+8
```





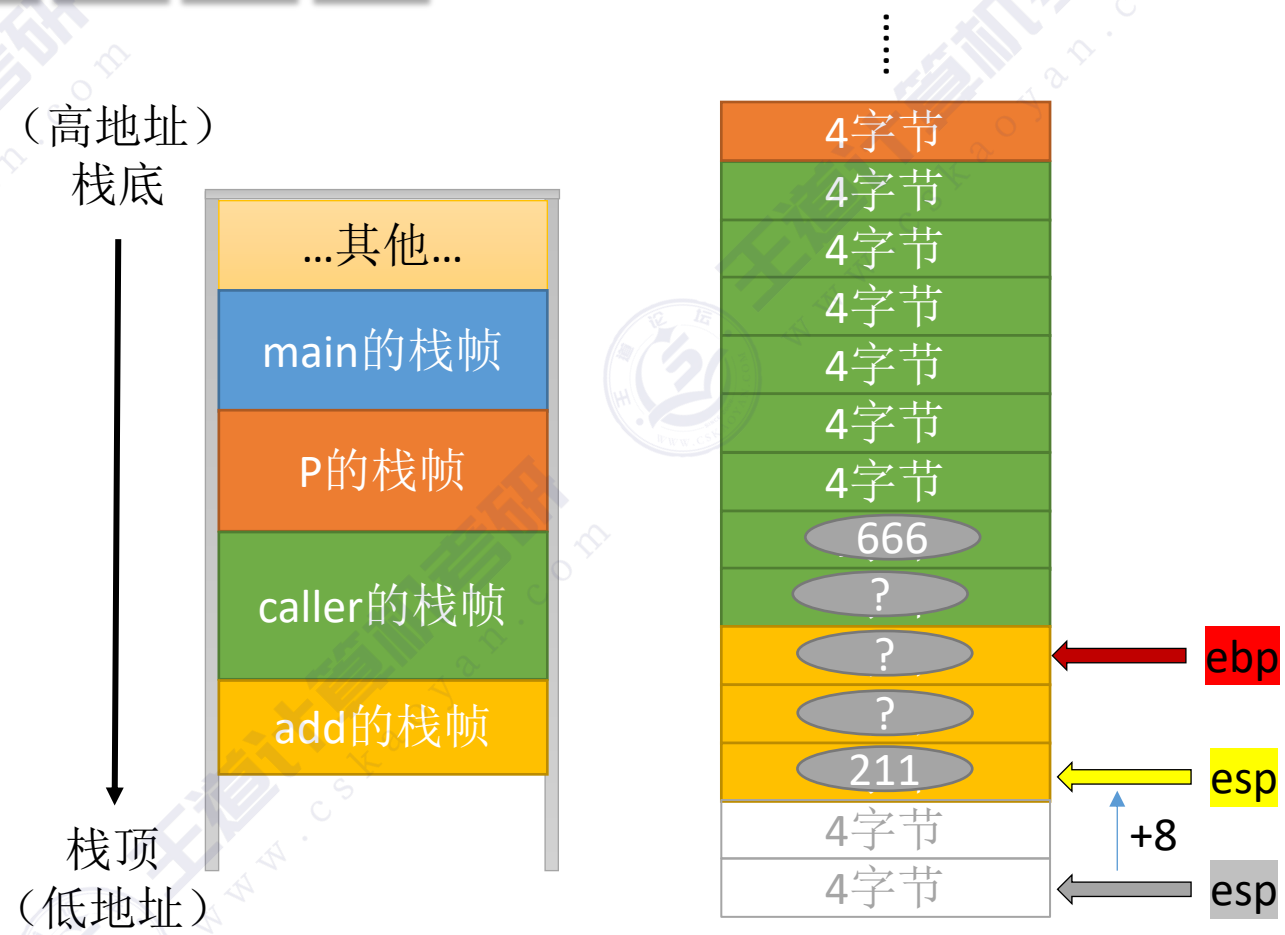
ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

访问栈帧数据：mov 指令

例：

```
sub esp, 12            #栈顶指针-12
mov [esp+8], eax       #将eax的值复制到主存[esp+8]
mov [esp+4], 958       #将985复制到主存[esp+4]
mov eax, [ebp+8]       #将主存[ebp+8]的值复制到eax
mov [esp],eax          #将eax的值复制到主存[esp]
add esp, 8             #栈顶指针+8
```



- 可以用 mov 指令，结合 esp、ebp 指针访问栈帧数据
- 可以用减法/加法指令，即 sub/add 修改栈顶指针 esp 的值

ebp：指向当前栈帧的“底部”

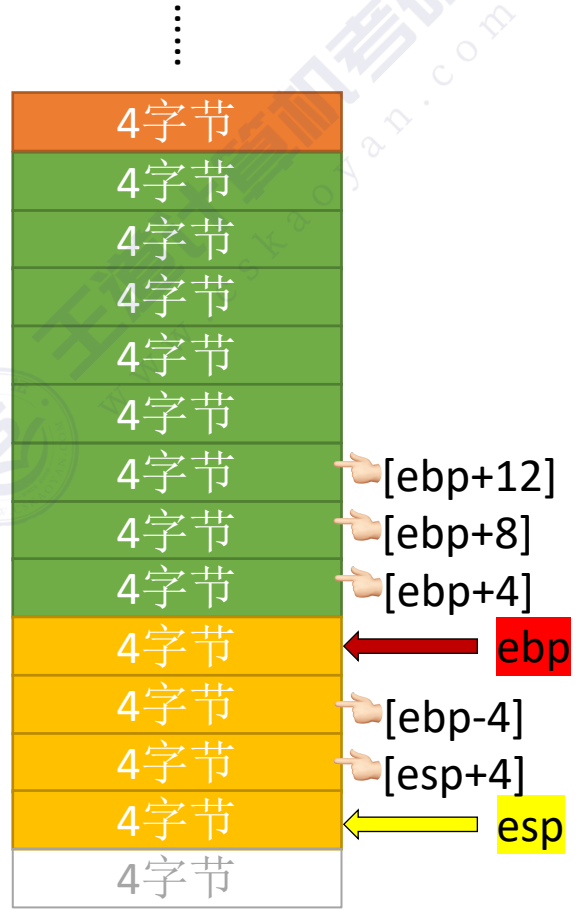
esp：指向当前栈帧的“顶部”

总结：如何访问栈帧？

方法一：

Push 🐶　　// 先让esp减4，再将 🐶 压入
Pop 🐴　　//栈顶元素出栈写入 🐴，再让 esp加4

注1：🐶 可以是立即数、寄存器、主存地址
注2：🐴 可以是寄存器、主存地址



方法二：

mov 指令，结合 esp、ebp 指针访问栈帧数据

注：可以用减法/加法指令，即 sub/add 修改栈顶指针 esp 的值

ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

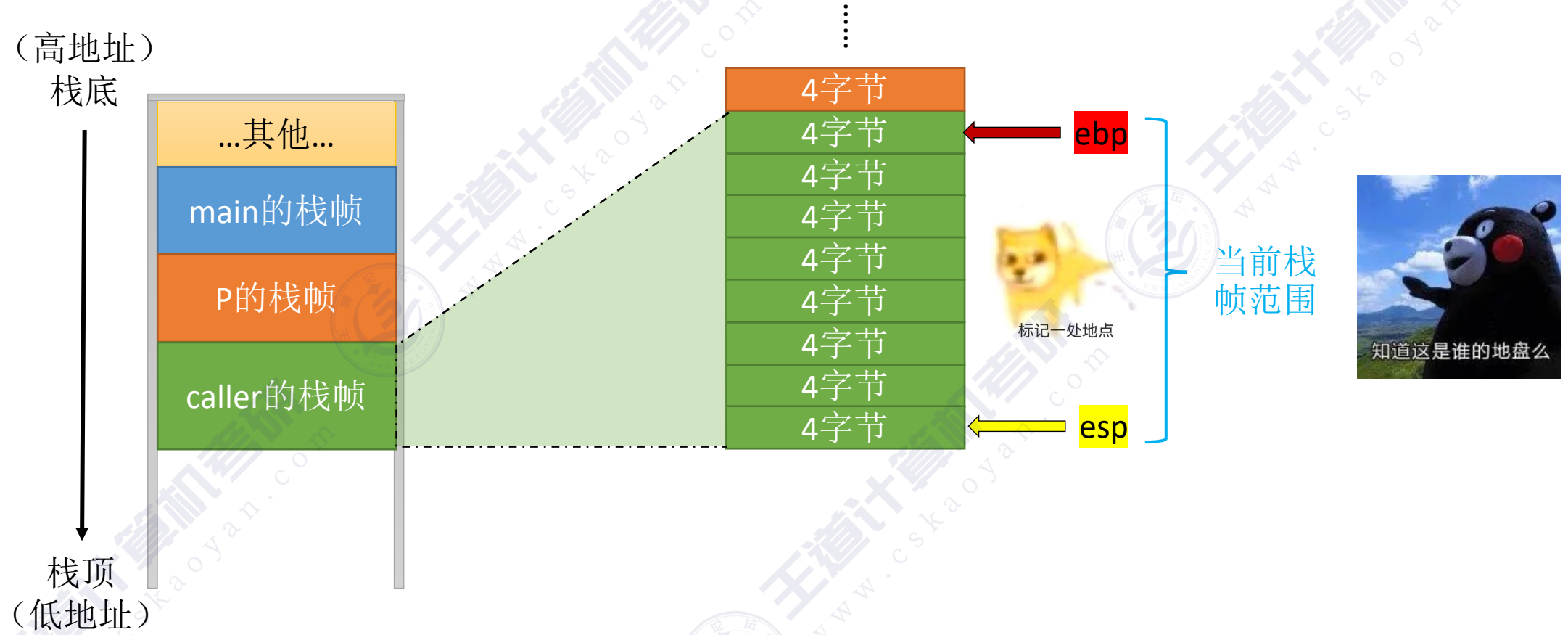
本节内容

函数调用

机器级表示

——如何切换栈帧？

标记栈帧范围： EBP、ESP寄存器

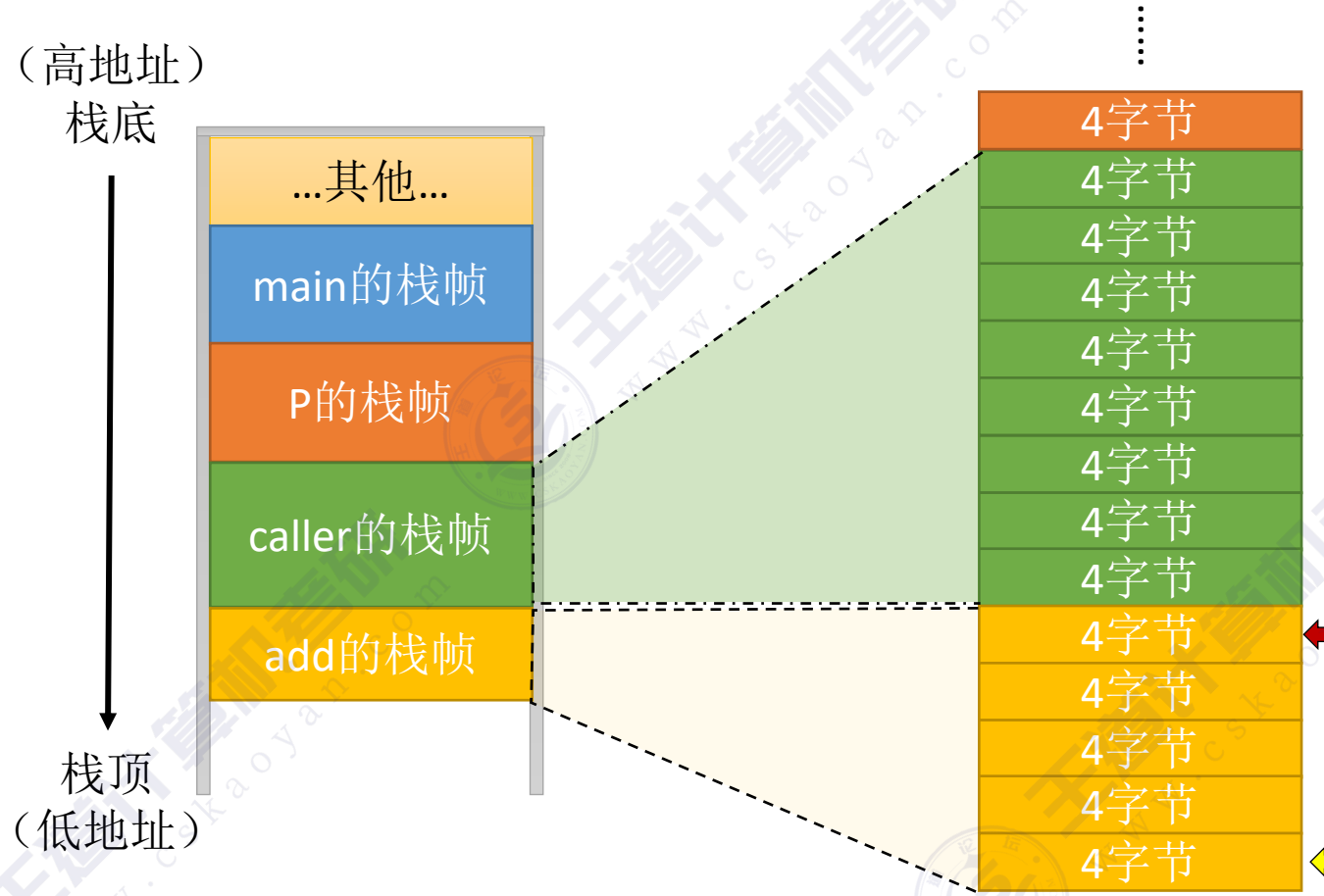


ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

注：x86 系统中，默认以4字节为栈的操作单位

标记栈帧范围： EBP、ESP寄存器

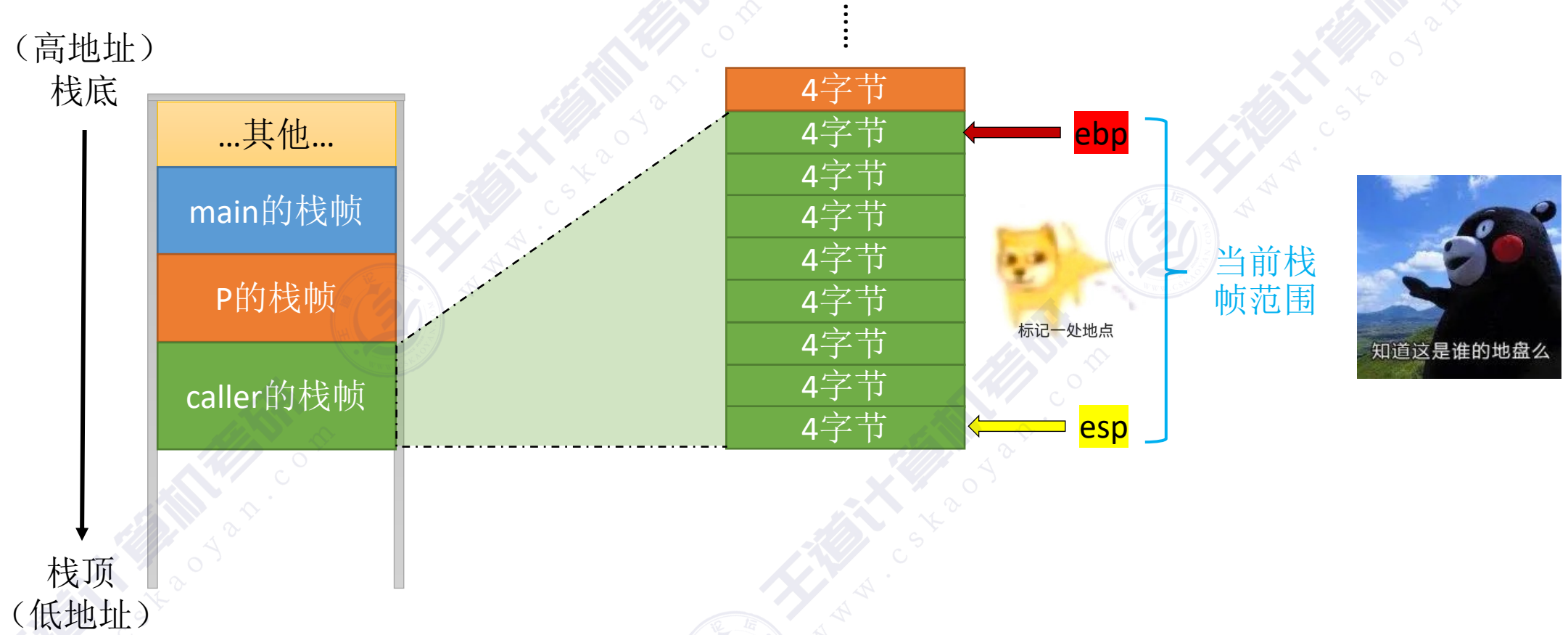


ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

注：x86 系统中，默认以4字节为栈的操作单位

标记栈帧范围： EBP、ESP寄存器



ebp：指向当前栈帧的“底部”

esp：指向当前栈帧的“顶部”

注：x86 系统中，默认以4字节为栈的操作单位

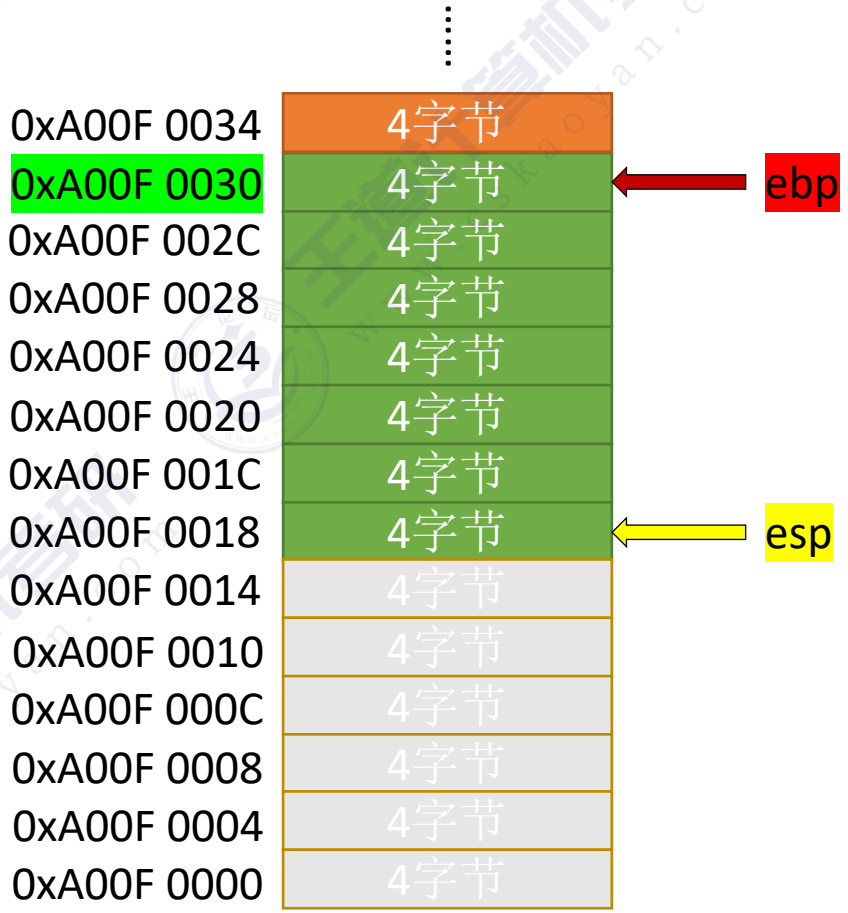
函数调用时，如何切换栈帧？

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

执行→ call add

IP→ mov [ebp-4],eax

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```



call 指令的作用：

①将IP旧值压栈保存（效果相当于 push IP）

②设置IP新值，无条件转移至被调用函数的第一条指令（效果相当于 jmp add）

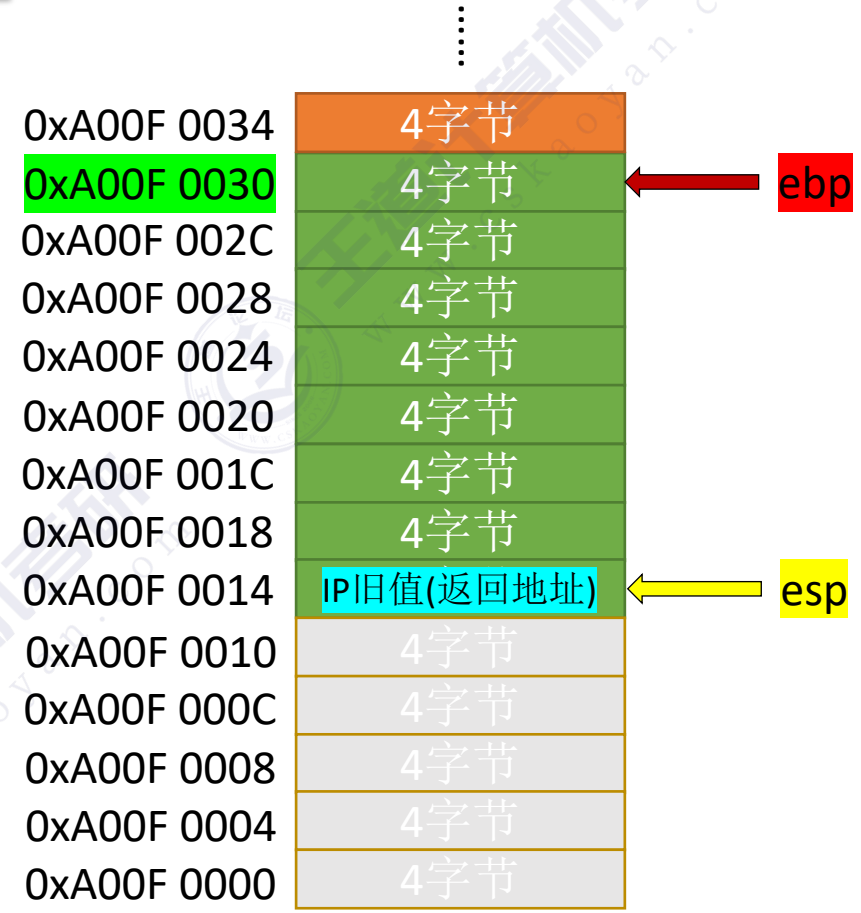
函数调用时，如何切换栈帧？

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

执行→ `call add`

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

IP→ `push ebp`



call 指令的作用：
①将IP旧值压栈保存（效果相当于 push IP）
②设置IP新值，无条件转移至被调用函数的第一条指令（效果相当于 jmp add）

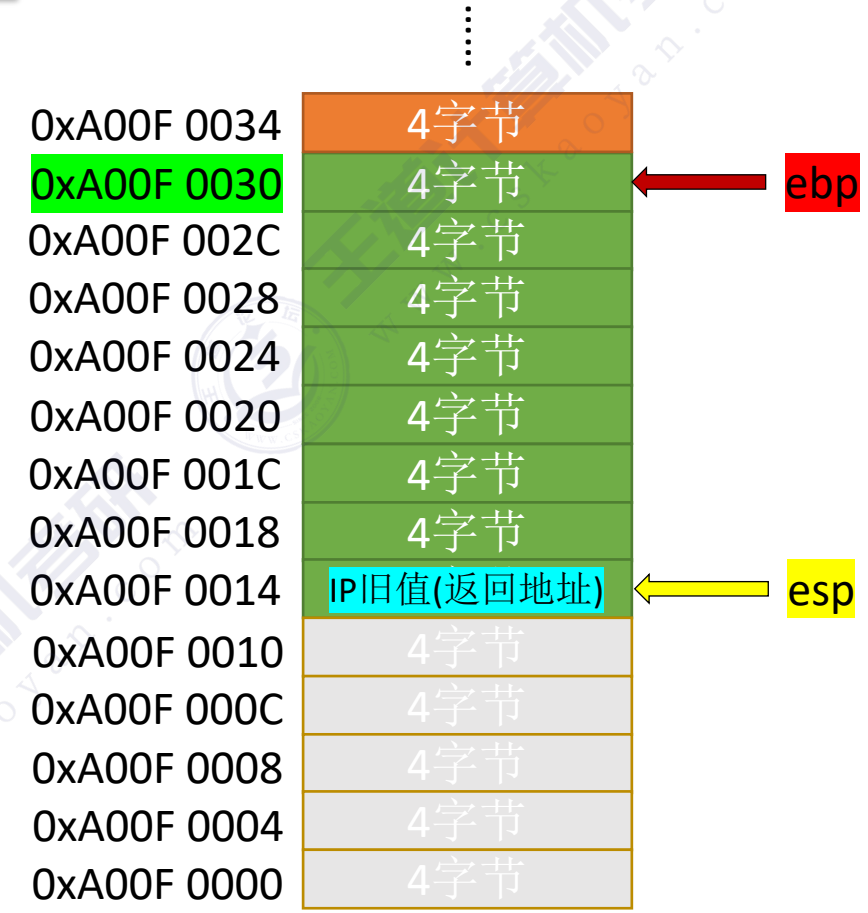
函数调用时，如何切换栈帧？

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

执行→

IP→

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```



call 指令的作用：

①将IP旧值压栈保存（效果相当于 push IP）

②设置IP新值，无条件转移至被调用函数的第一条指令（效果相当于 jmp add）

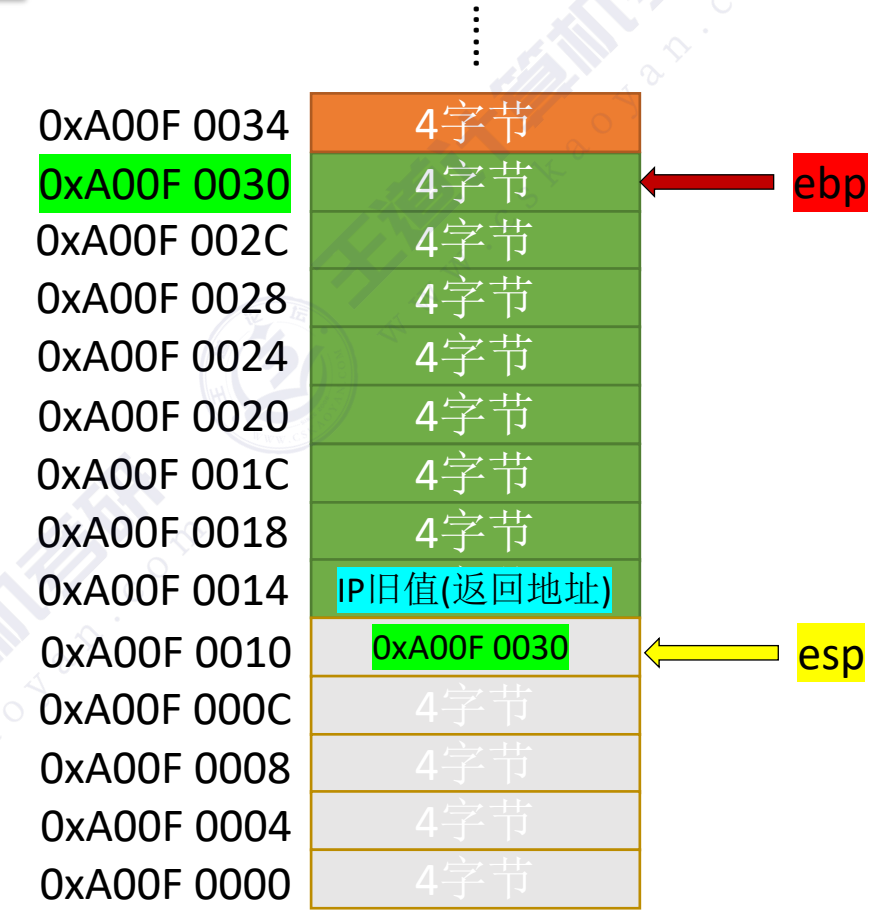
函数调用时，如何切换栈帧？

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

执行→ mov ebp,esp

IP→ mov eax,[ebp+12]



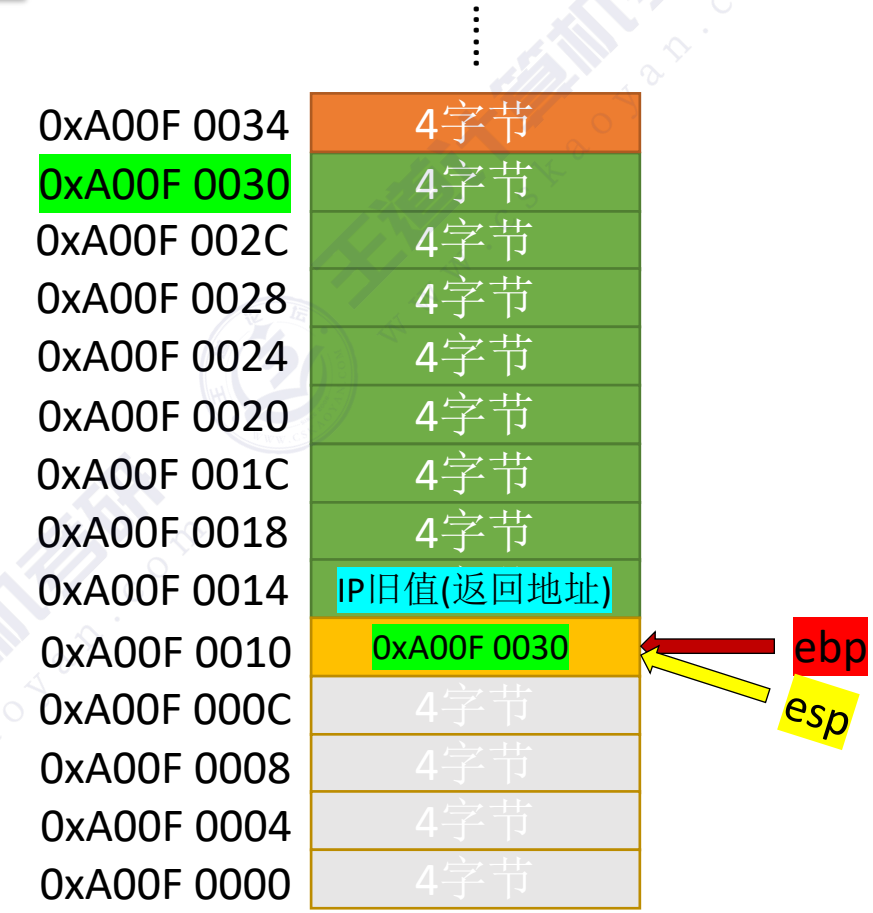
函数调用时，如何切换栈帧？

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

执行→ mov ebp,esp

IP→ mov eax,[ebp+12]



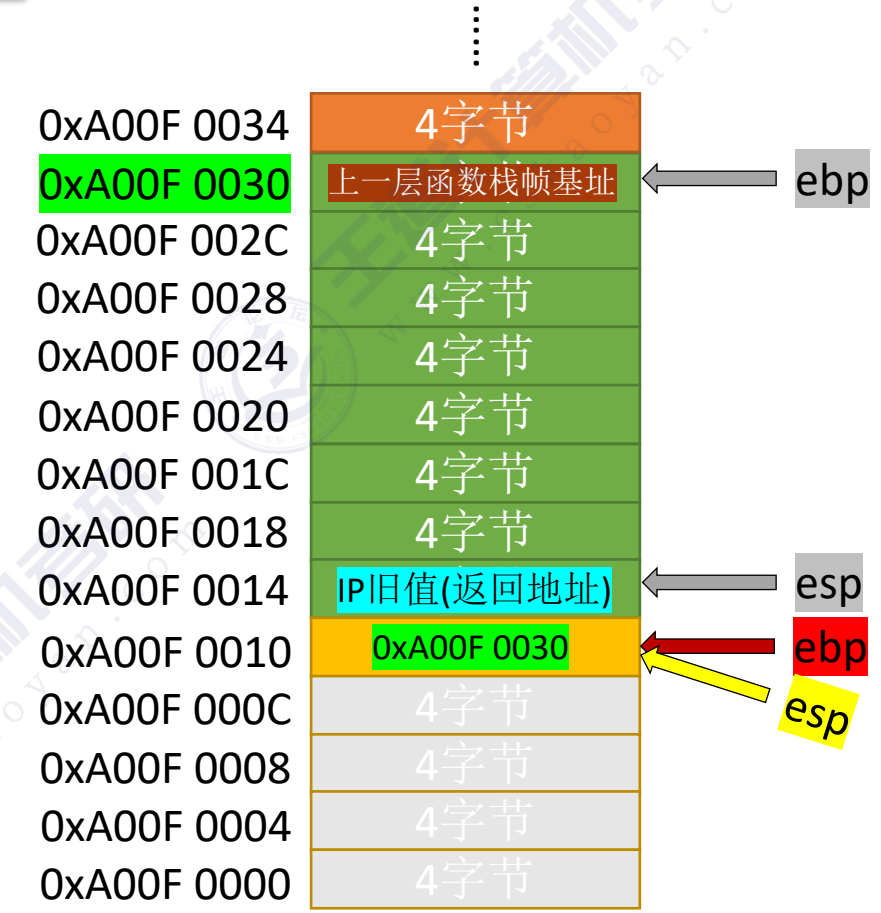
函数调用时，如何切换栈帧？

在每个函数开头的“例行处理”

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

IP旧值→

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```



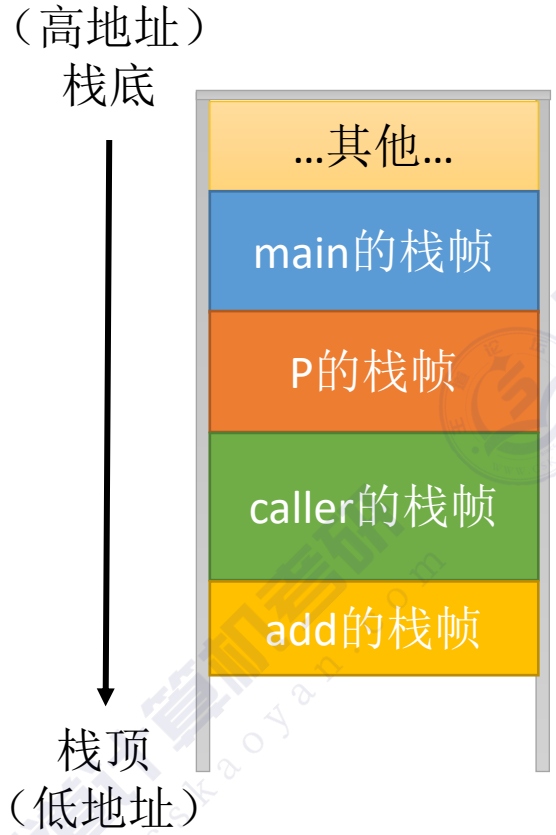
指令：enter

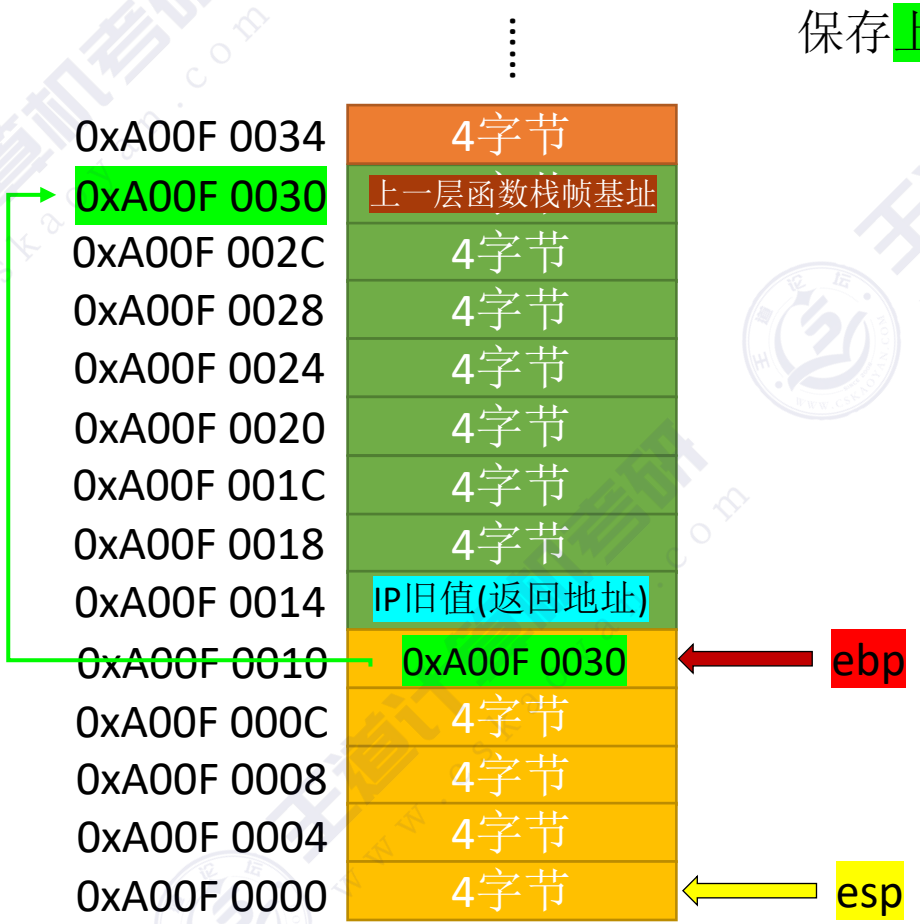
等价

```
push ebp          #保存上一层函数的栈帧基址（ebp旧值）
mov ebp,esp       #设置当前函数的栈帧基址（ebp新值）
```

函数返回时，如何切换栈帧？

注：每个栈帧底部，用于保存上一层栈帧的基址

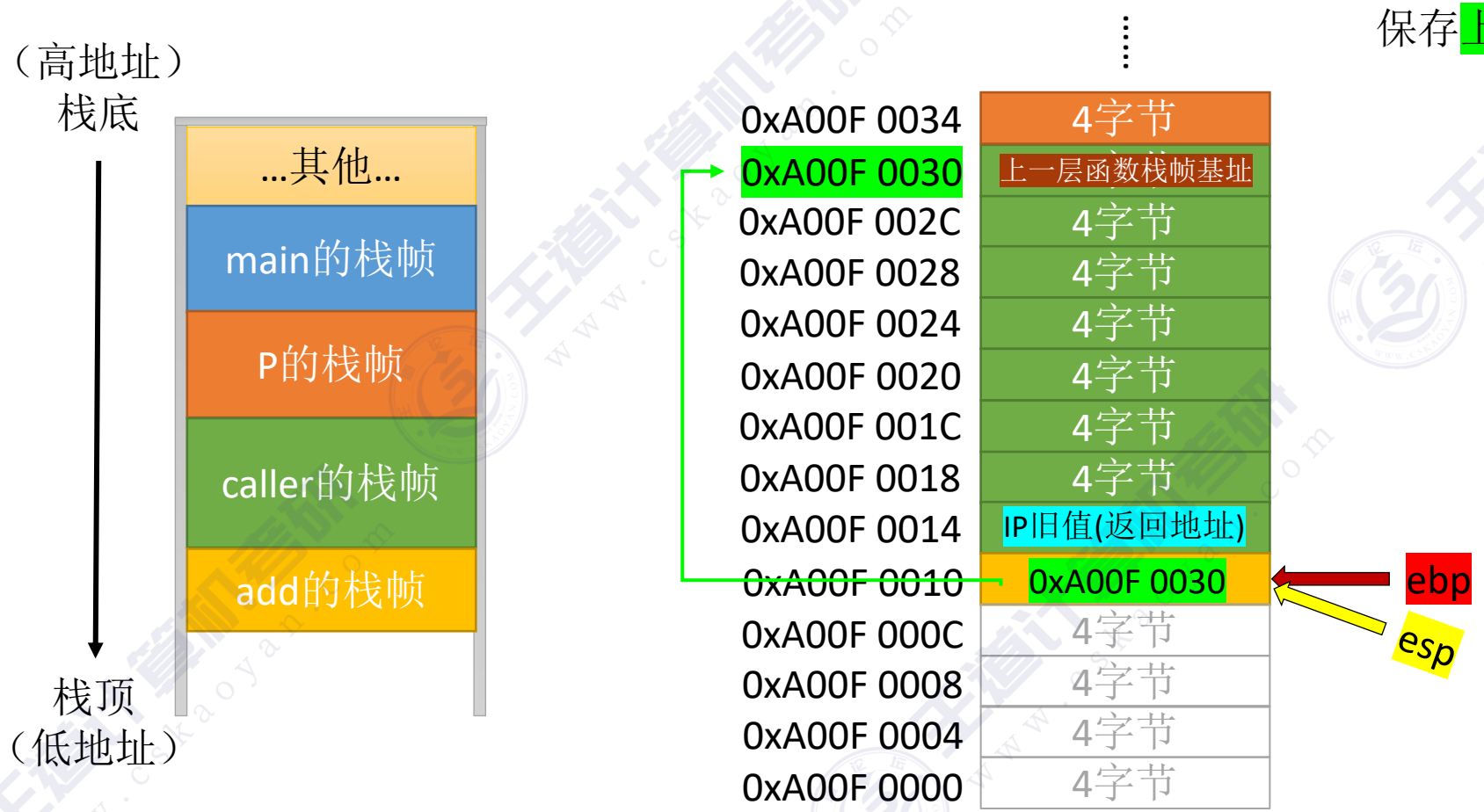





```
mov esp, ebp        #让esp指向当前栈帧的底部
pop ebp             #将esp所指元素出栈，写入寄存器ebp
```

函数返回时，如何切换栈帧？

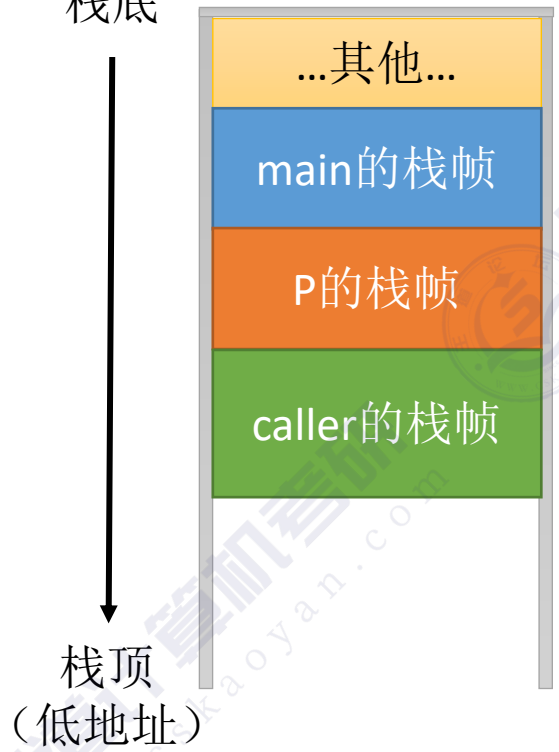
注：每个栈帧底部，用于保存上一层栈帧的基址

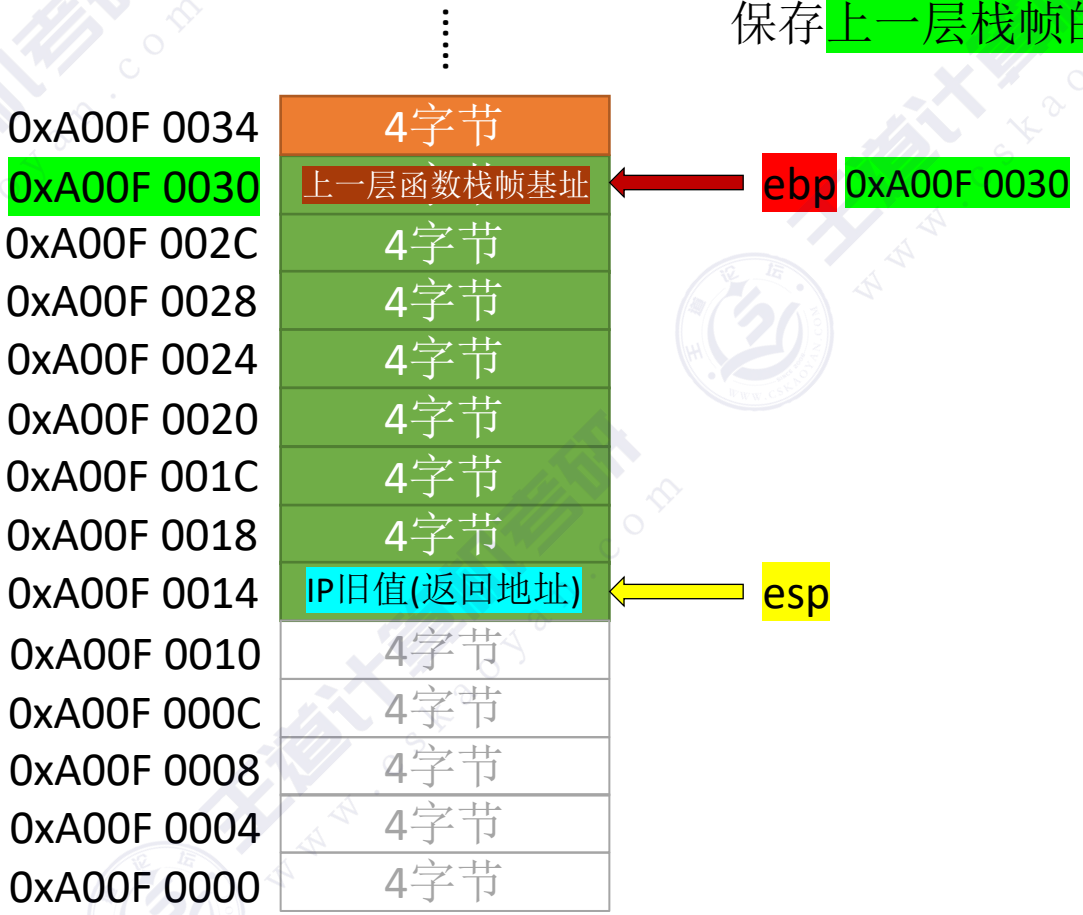



```
mov esp, ebp        #让esp指向当前栈帧的底部
pop ebp             #将esp所指元素出栈，写入寄存器ebp
```

函数返回时，如何切换栈帧？

注：每个栈帧底部，用于保存上一层栈帧的基址





指令：leave

效果等价

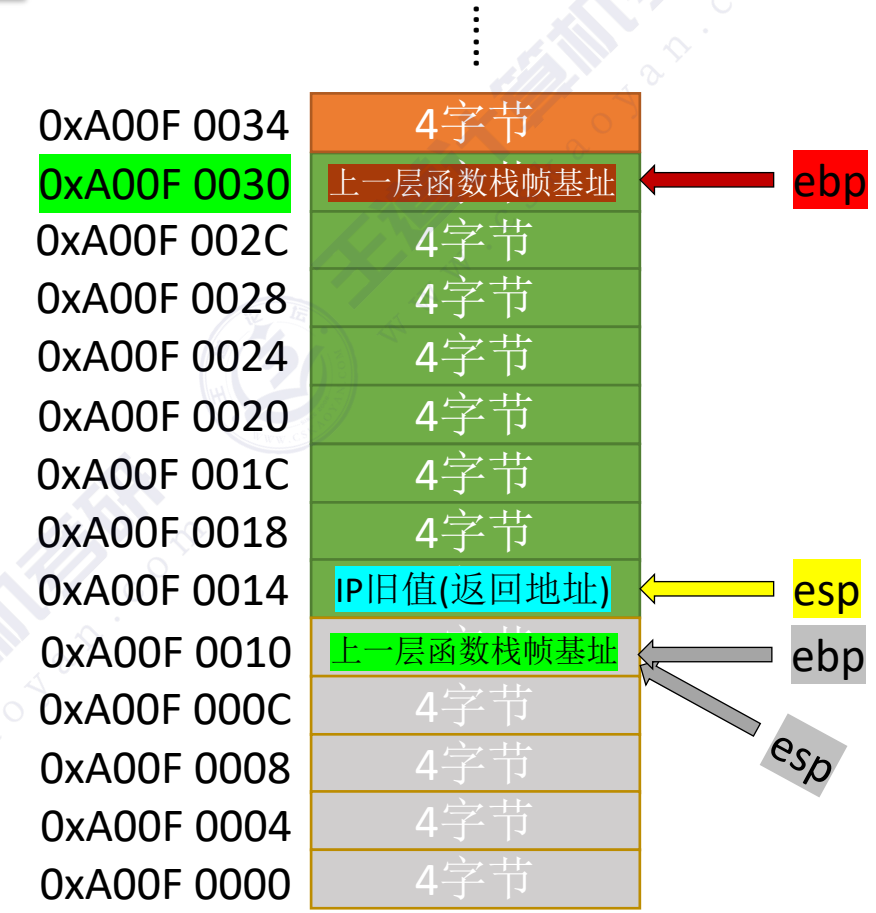
```
mov esp, ebp      #让esp指向当前栈帧的底部
pop ebp           #将esp所指元素出栈，写入寄存器ebp
```

函数返回时，如何切换栈帧？

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
IP旧值→ mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

每个函数 ret 前的“例行处理”



指令：leave

等价

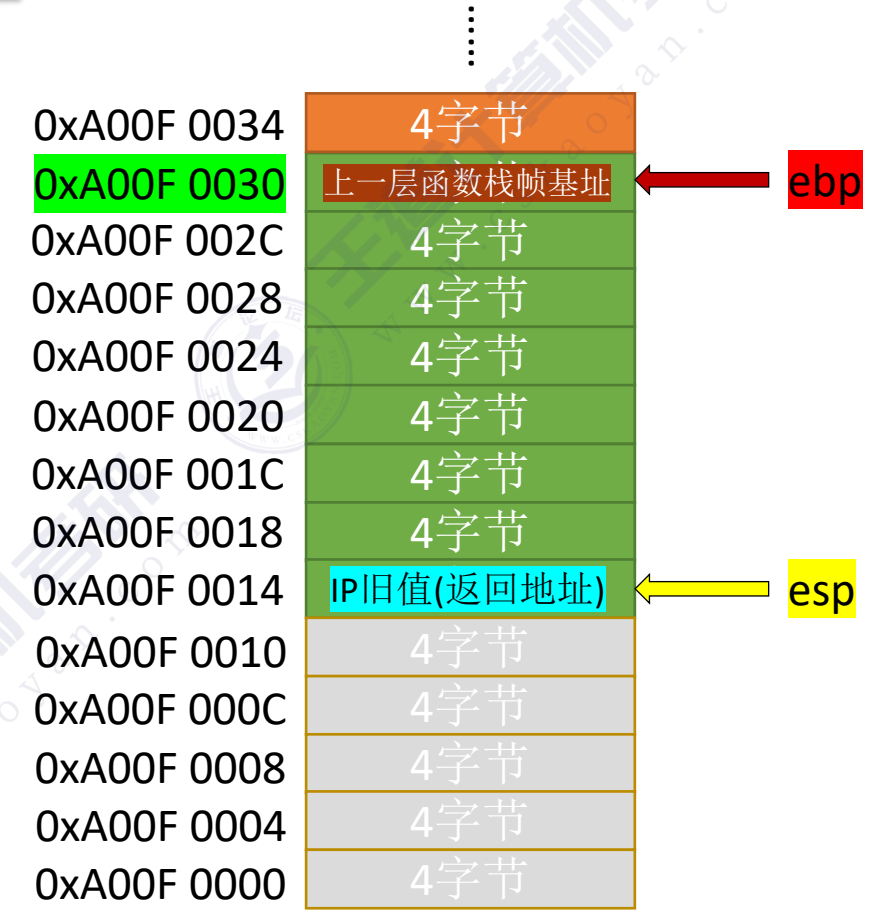
```
mov esp, ebp      #让esp指向当前栈帧的底部
pop ebp           #将esp所指元素出栈，写入寄存器ebp
```

函数返回时，如何切换栈帧？

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

IP旧值→ `mov [ebp-4],eax`

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```



ret 指令的作用：
从函数的栈帧顶部找到 IP旧值，将其出栈并恢复IP寄存器

总结：函数调用的机器级表示

除了main函数，其他所有函数的汇编代码结构都一样！

调用者：

……

- 执行 call 指令

返回地址压入栈顶、并跳转到被调用函数第一条指令

……

被调用者：

- 保存上一层函数栈帧，设置当前函数栈帧

push ebp
mov ebp,esp

或：enter指令

- 一系列处理逻辑
- 恢复上一层函数的栈帧

mov esp, ebp
pop ebp

或：leave指令

- 执行 ret 指令

从栈顶找到返回地址，出栈并恢复 IP 值

一个函数的汇编代码框架：

被调用者：

- 保存上一层函数栈帧，设置当前函数栈帧
- 一系列处理逻辑
- 恢复上一层函数的栈帧
- 执行 ret 指令

例：2019年真题

45.（16 分）已知 $f(n)=n!=n\times(n-1)\times(n-2)\times\cdots\times2\times1$，计算 $f(n)$的 C 语言函数 f1 的源程序（阴影部分）及其在 32 位计算机 M 上的部分机器级代码如下：

```
int   f1(int n){
1    00401000    55              push  ebp
     …           …               …
     if(n>1)
11   00401018    83 7D 08 01     cmp  dword ptr [ebp+8],1
12   0040101C    7E 17           jle  f1+35h (00401035)
     return n*f1(n-1);
13   0040101E    8B 45 08        mov  eax, dword ptr [ebp+8]
14   00401021    83 E8 01        sub  eax, 1
15   00401024    50              push  eax
16   00401025    E8 D6 FF FF FF  call  f1 ( 00401000)
     …           …               …
19   00401030    0F AF C1        imul  eax, ecx
20   00401033    EB 05           jmp  f1+3Ah (0040103a)
     else return 1;
21   00401035    B8 01 00 00 00  mov  eax,1
     }
     …           …               …
26   00401040    3B EC           cmp  ebp, esp
     …           …               …
30   0040104A    C3              ret
```

push ebp
mov ebp,esp

mov esp, ebp
pop ebp

或：leave指令

其中，机器级代码行包括行号、虚拟地址、机器指令和汇编指令，计算机 M 按字节编址，int 型数据占 32 位。请回答下列问题：

一个函数的汇编代码框架：

被调用者：

- 保存上一层函数栈帧，设置当前函数栈帧
- 一系列处理逻辑
- 恢复上一层函数的栈帧
- 执行 ret 指令

例：2017年真题

44.（10 分）在按字节编址的计算机 M 上，题 43 中 f1 的部分源程序（阴影部分）与对应的机器级代码（包括指令的虚拟地址）如下：

其中，机器级代码行包括行号、虚拟地址、机器指令和汇编指令。请回答下列问题。

```
                int f1( unsigned n)
1       00401020    55          push ebp
        ……          ……          ……
                    for(unsigned i=0; i<= n -1; i++)
        ……          ……          ……
20      0040105E    39 4D F4    cmp dword ptr [ebp-0Ch],ecx
        ……          ……          ……
                    {       power *= 2;
        ……          ……          ……
23      00401066    D1 E2       shl   edx,1
        ……          ……          ……
                    return sum;
        ……          ……          ……
35      0040107F    C3          ret
```

push ebp
mov ebp,esp

mov esp, ebp
pop ebp

或：leave指令

本节内容

函数调用

机器级表示

——栈帧内包含哪些内容？
——参数、返回值传递

问题回顾

```
int caller(){
   int temp1=125;
   int temp2=80;
   int sum=add(temp1,temp2);
   return sum;
}
```

```
int add(int x, int y){
   return x+y;
}
```

如何传递调用参数、返回值？

如何访问栈帧里的数据？

栈帧内可能包含哪些内容？

栈为什么倒过来了？

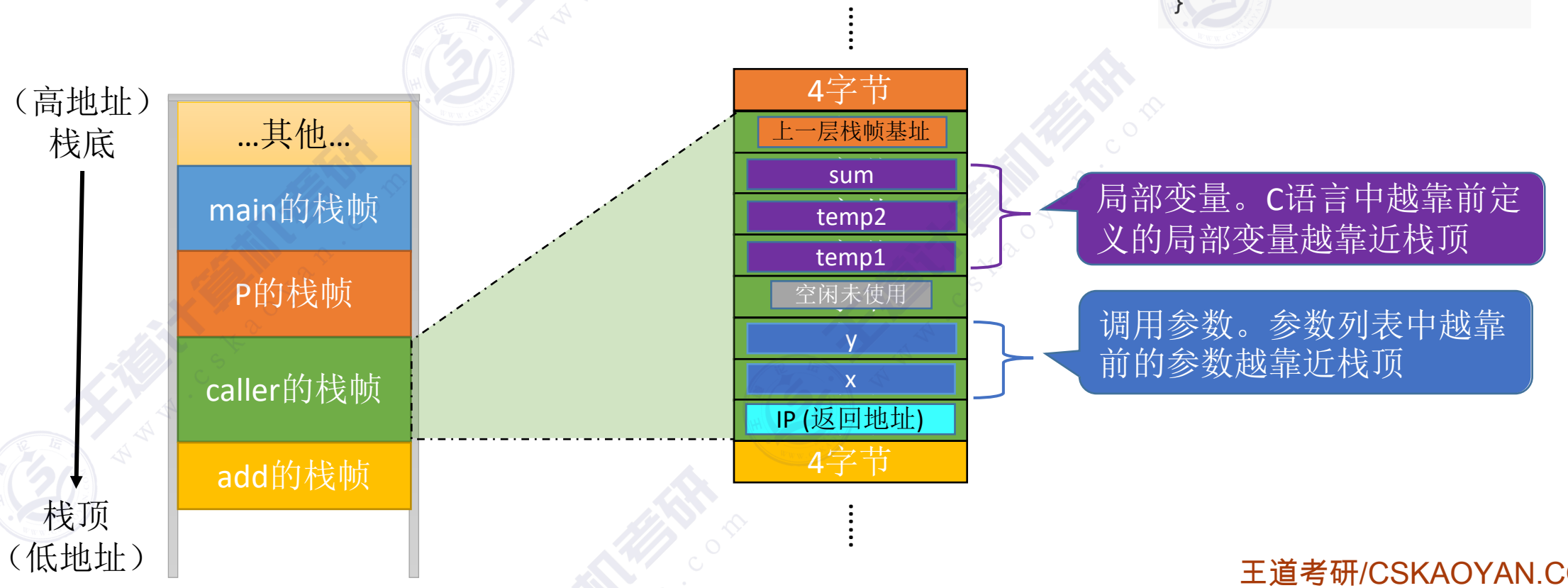
GNU/Linux家的好东西，开源、自由、非常流行

一个栈帧内可能包含哪些内容？

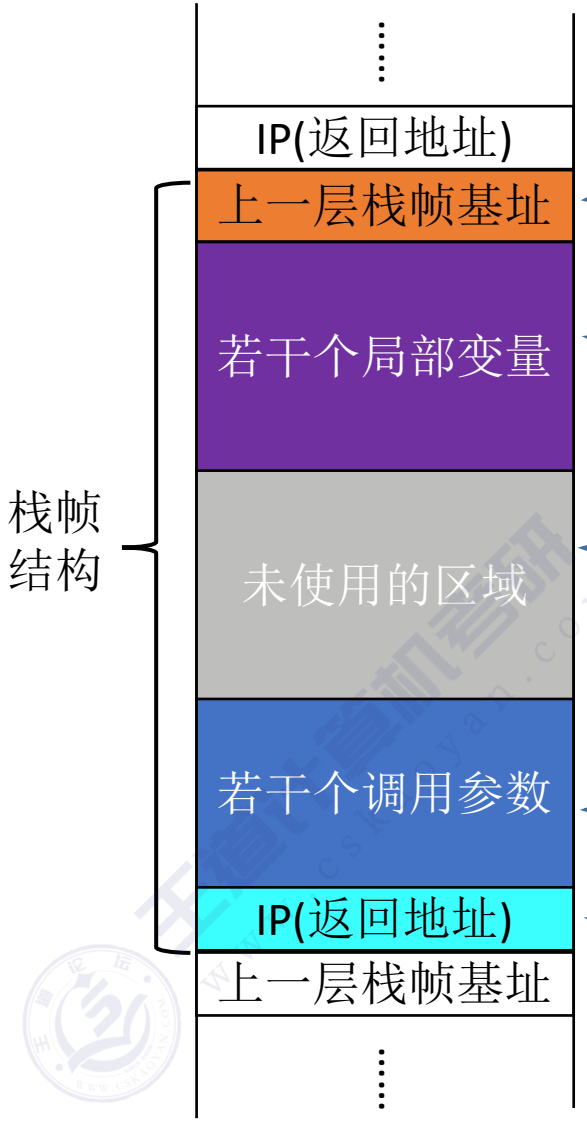
- gcc 编译器将每个栈帧大小设置为 16B 的整数倍（当前函数的栈帧除外），因此栈帧内可能出现空闲未使用的区域。
- 通常将局部变量集中存储在栈帧底部区域
- 通常将调用参数集中存储在栈帧顶部区域
- 栈帧最底部一定是上一层栈帧基址（ebp旧值）
- 栈帧最顶部一定是返回地址（当前函数的栈帧除外）

```
int caller(){
   int temp1=125;
   int temp2=80;
   int sum=add(temp1,temp2);
   return sum;
}
```

```
int add(int x, int y){
   return x+y;
}
```



一个栈帧内可能包含哪些内容？



每个函数开头都要执行：
push ebp
mov ebp,esp

一定存在。用于恢复上一层函数的栈帧

局部变量。C语言中越靠前定义的局部变量越靠近栈顶

不一定存在。有些函数可能不定义局部变量

栈帧大小为16B整数倍，因此可能产生未使用区域

不一定存在。如果其他部分刚好是16B整数倍，则不会留下“零头”

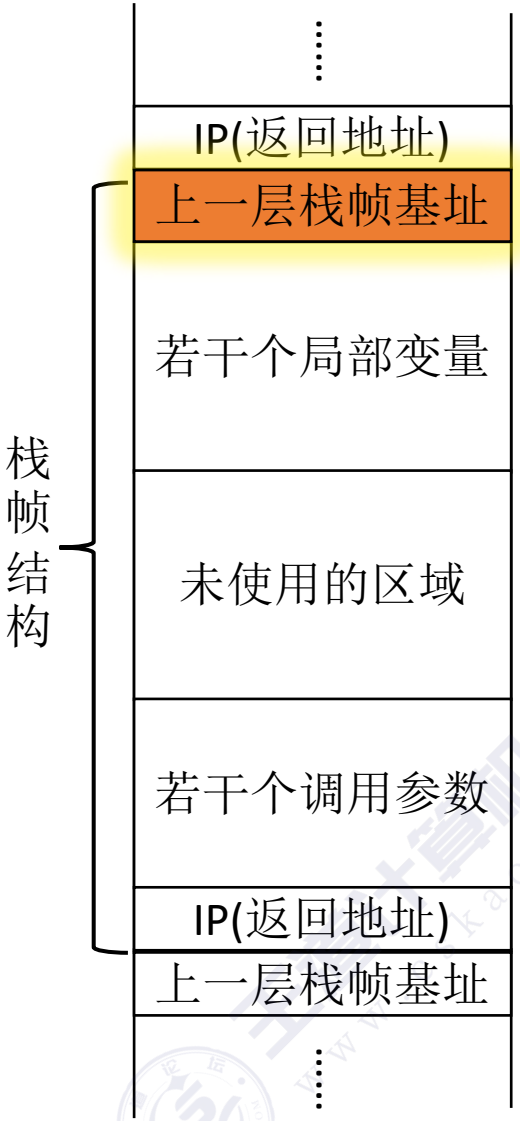
调用参数。参数列表中越靠前的参数越靠近栈顶

不一定存在。有些函数调用不需要传参数

函数调用时，call 指令将IP寄存器值（返回地址）压栈保存

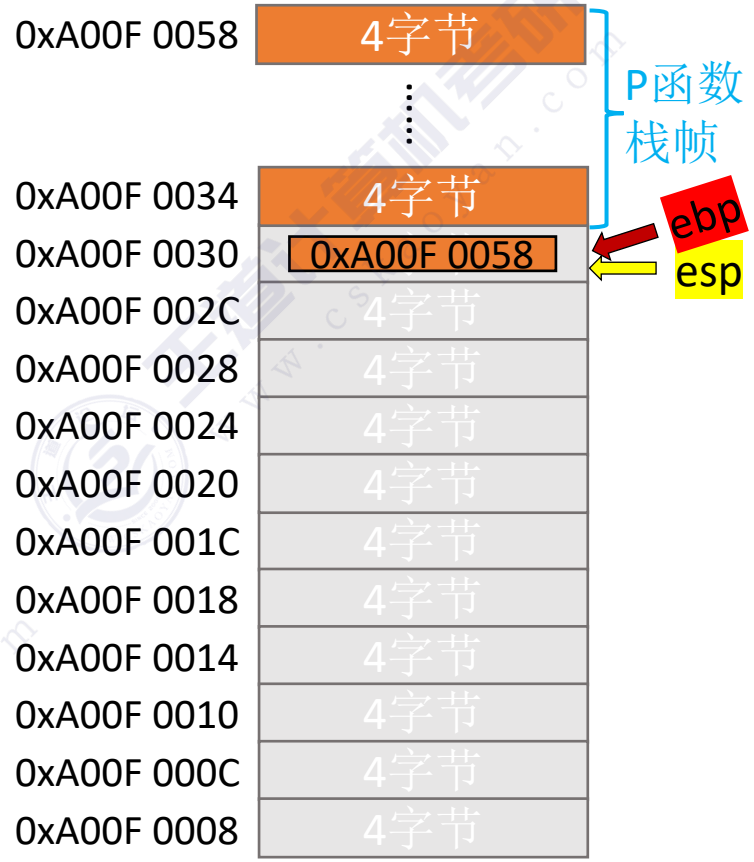
一定存在（发生调用时）。但凡调用其他函数，就必须记录返回地址

汇编代码实战




```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

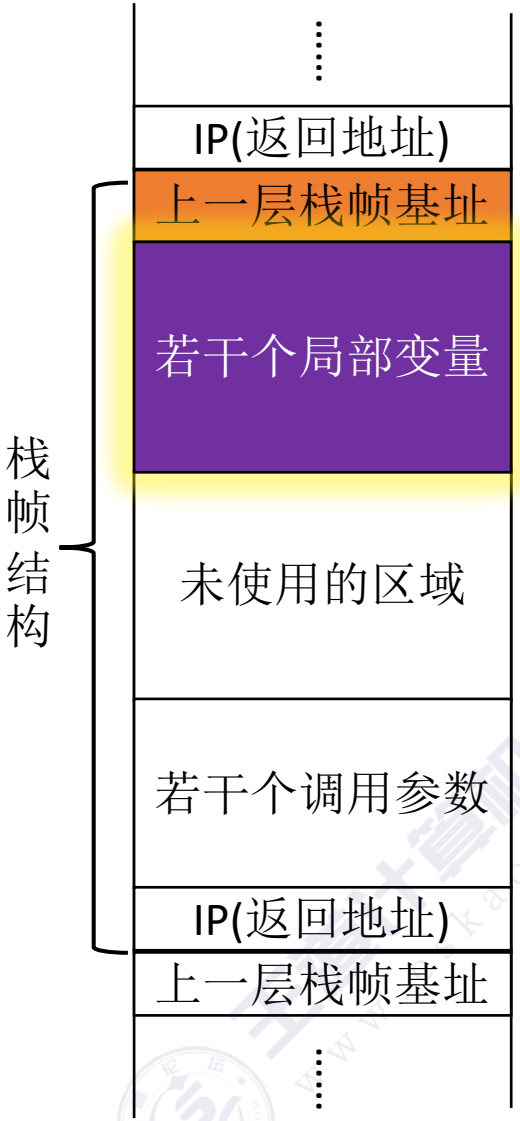



```
int caller(){
  int temp1=125;
  int temp2=80;
  int sum=add(temp1,temp2);
  return sum;
}
```

eax: edx:

```
int add(int x, int y){
  return x+y;
}
```

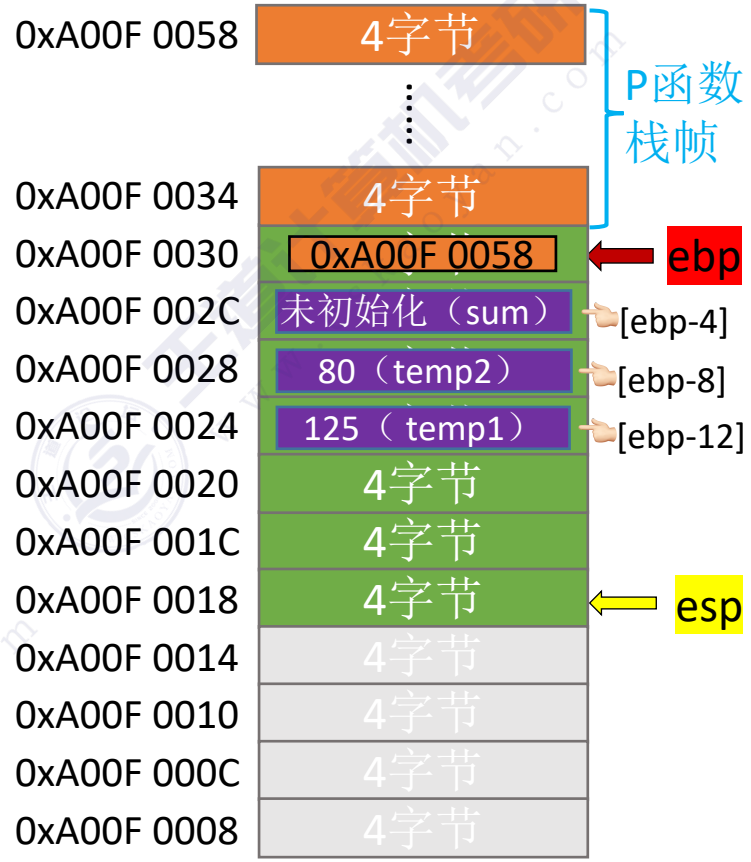
访问当前函数的局部变量：[ebp-4]、[ebp-8]...



汇编代码实战：访问局部变量

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

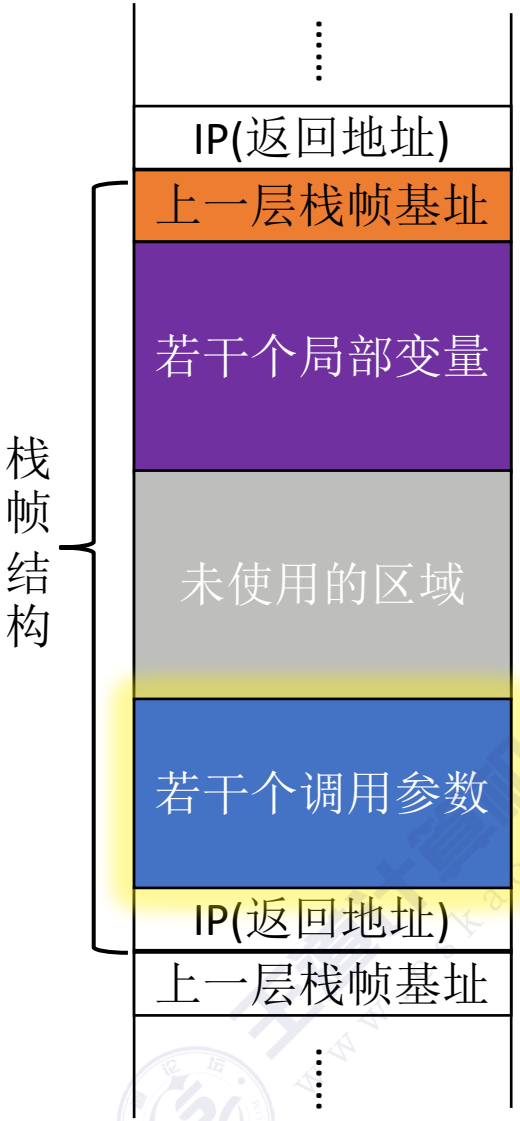



```
int caller(){
  int temp1=125;
  int temp2=80;
  int sum=add(temp1,temp2);
  return sum;
}
```

eax: edx:

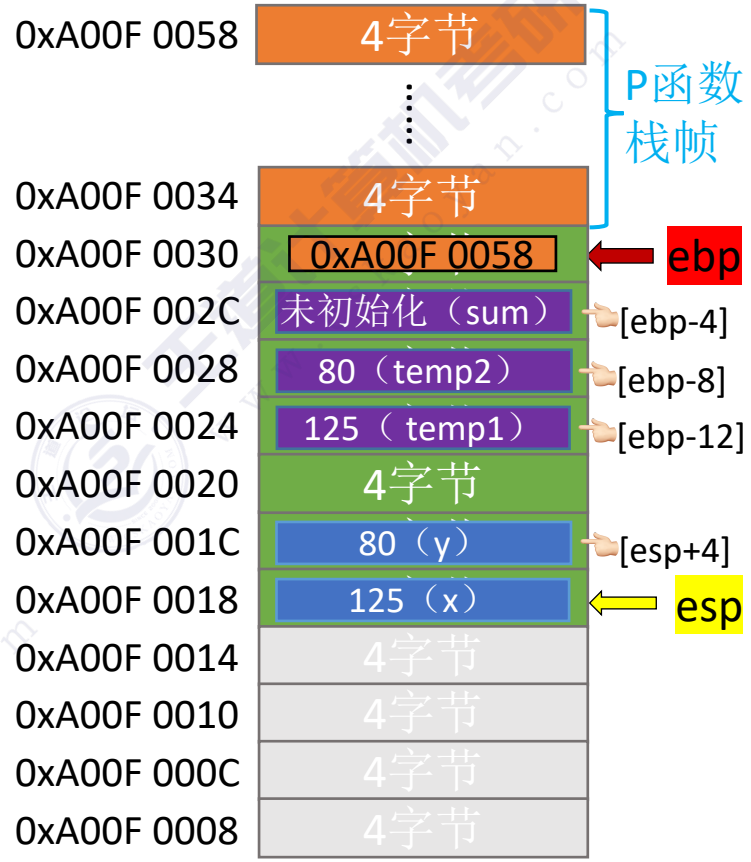
```
int add(int x, int y){
  return x+y;
}
```

汇编代码实战：传递参数




```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

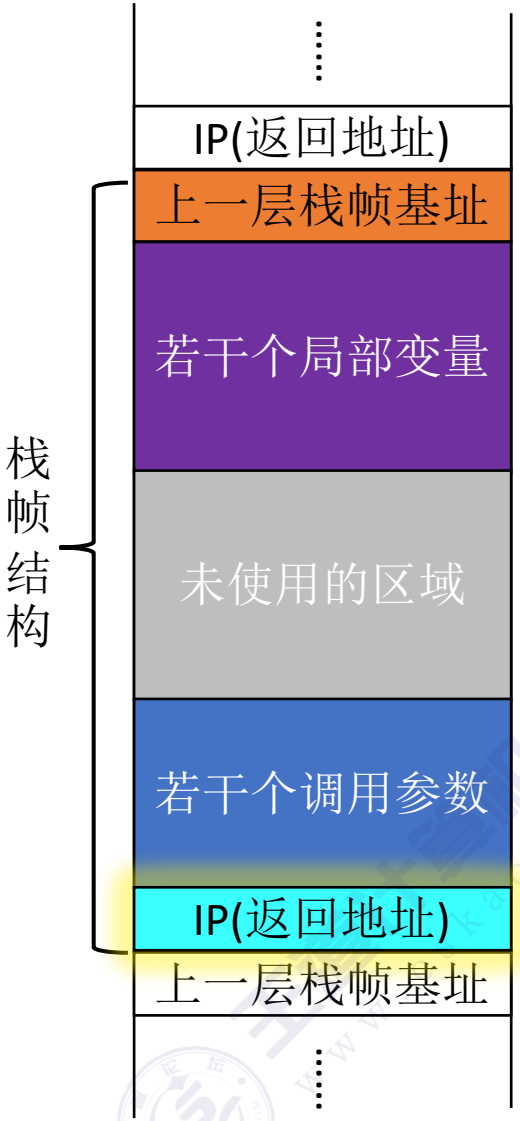



```
int caller(){
  int temp1=125;
  int temp2=80;
  int sum=add(temp1,temp2);
  return sum;
}
```

eax: 125　　edx:

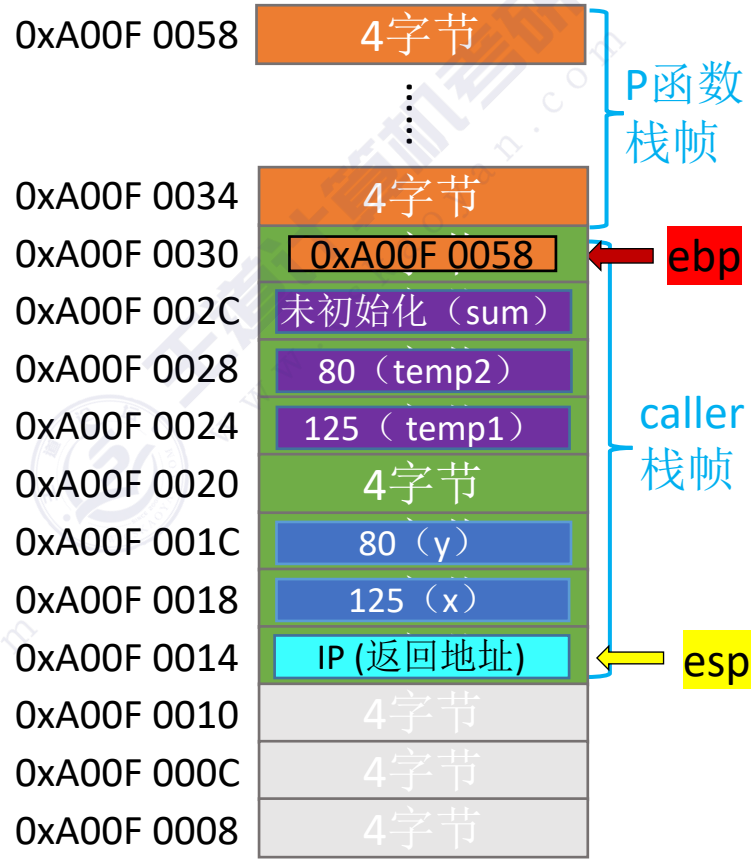
```
int add(int x, int y){
  return x+y;
}
```

汇编代码实战：函数调用




```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

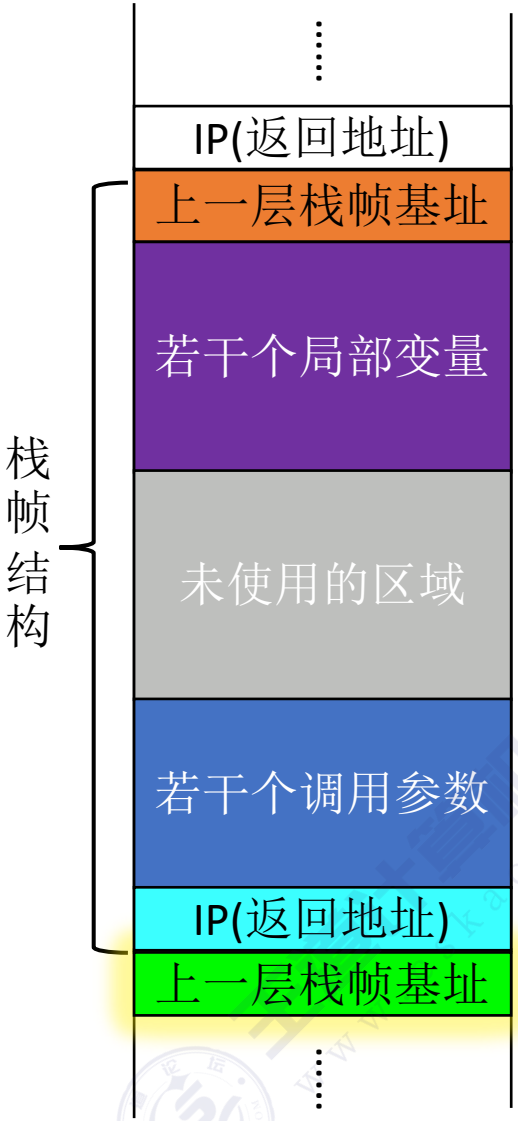



```
int caller(){
  int temp1=125;
  int temp2=80;
  int sum=add(temp1,temp2);
  return sum;
}
```

eax: 125　　edx:

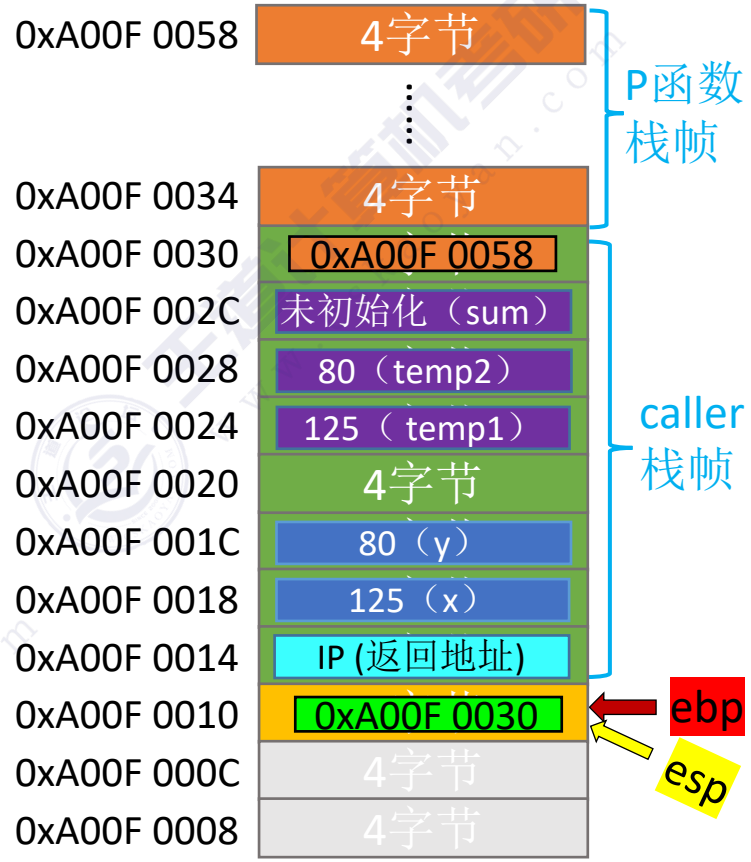
```
int add(int x, int y){
  return x+y;
}
```

汇编代码实战：切换栈帧




```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

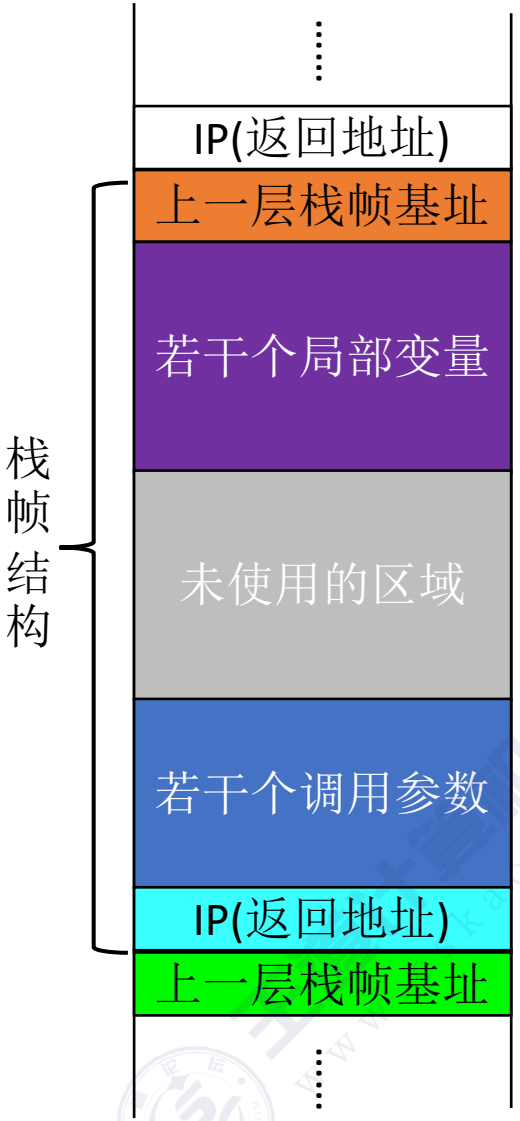



```
int caller(){
  int temp1=125;
  int temp2=80;
  int sum=add(temp1,temp2);
  return sum;
}
```

eax: 125　　edx:

```
int add(int x, int y){
  return x+y;
}
```

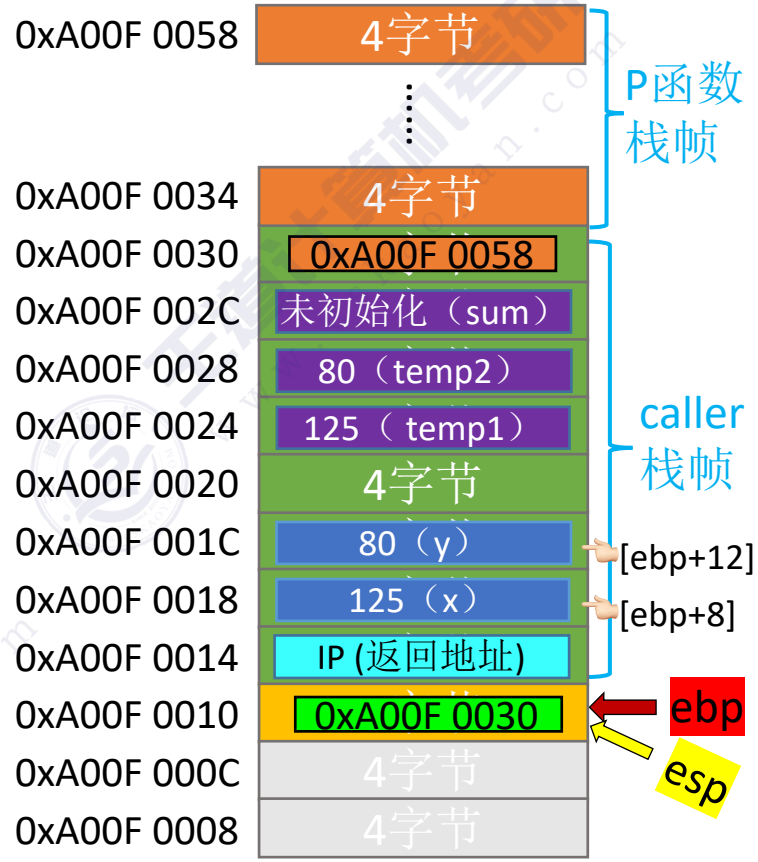
访问上一层函数传过来的参数：[ebp+8]、[ebp+12]...



汇编代码实战：访问参数

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

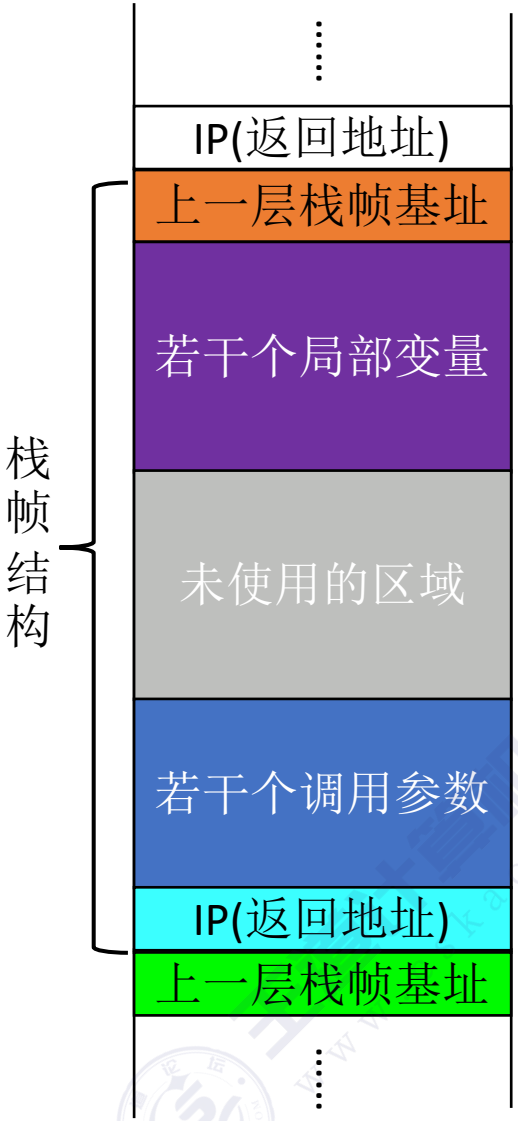



```
int caller(){
  int temp1=125;
  int temp2=80;
  int sum=add(temp1,temp2);
  return sum;
}
```

eax: 80　　edx: 125

```
int add(int x, int y){
  return x+y;
}
```

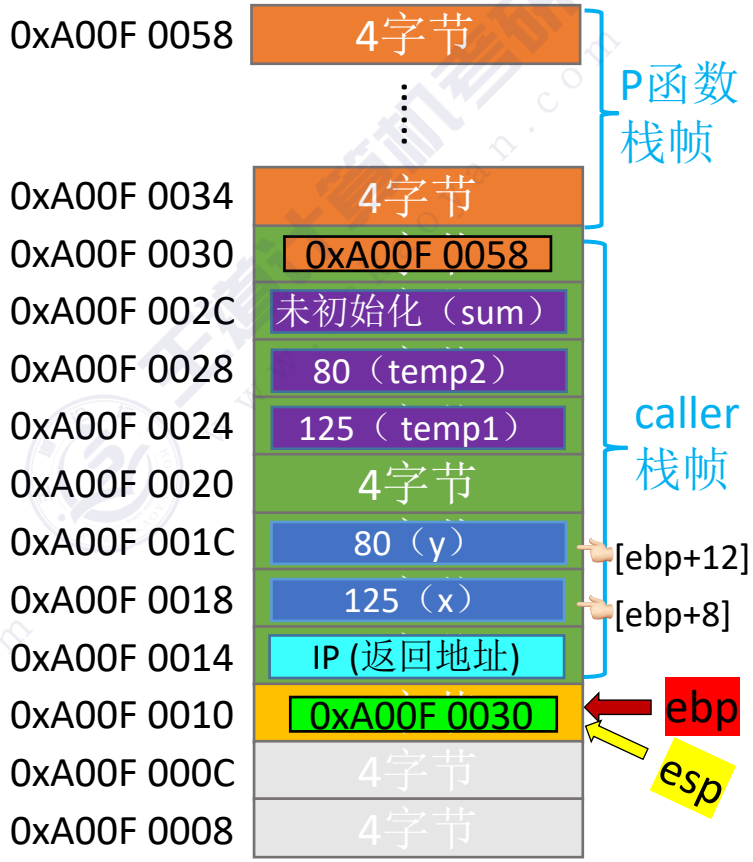
通常用eax寄存器返回结果



汇编代码实战：传递返回值

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

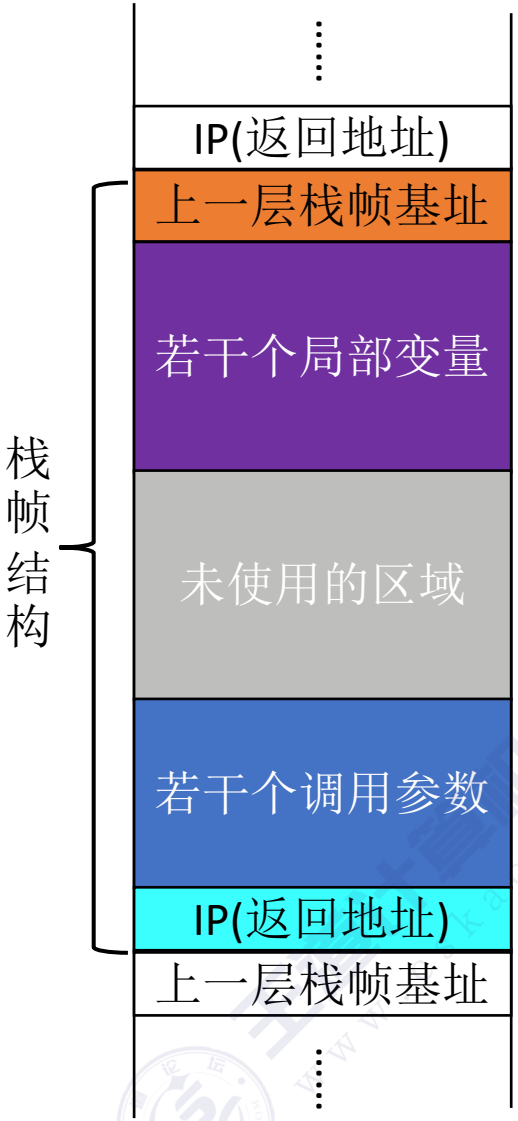



```
int caller(){
  int temp1=125;
  int temp2=80;
  int sum=add(temp1,temp2);
  return sum;
}
```

eax: 205　　edx: 125

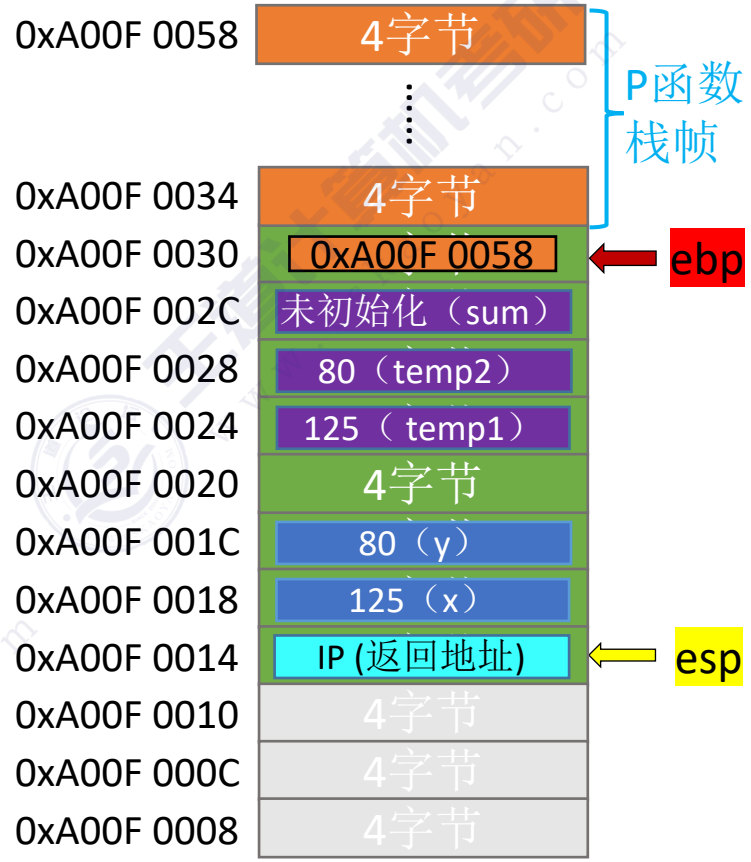
```
int add(int x, int y){
  return x+y;
}
```

汇编代码实战：恢复上一层栈帧




```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

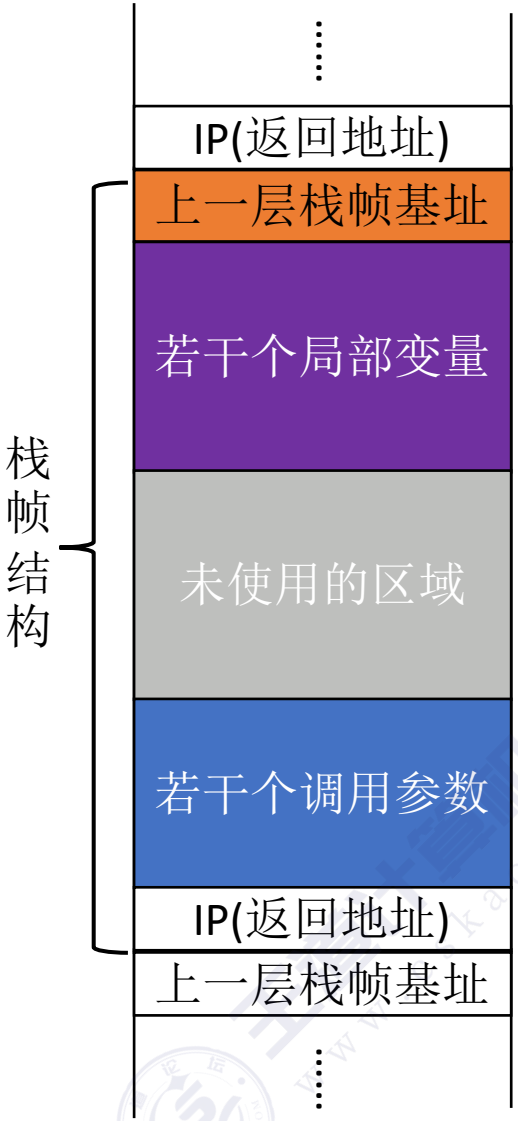



```
int caller(){
  int temp1=125;
  int temp2=80;
  int sum=add(temp1,temp2);
  return sum;
}
```

eax: 205　　edx: 125

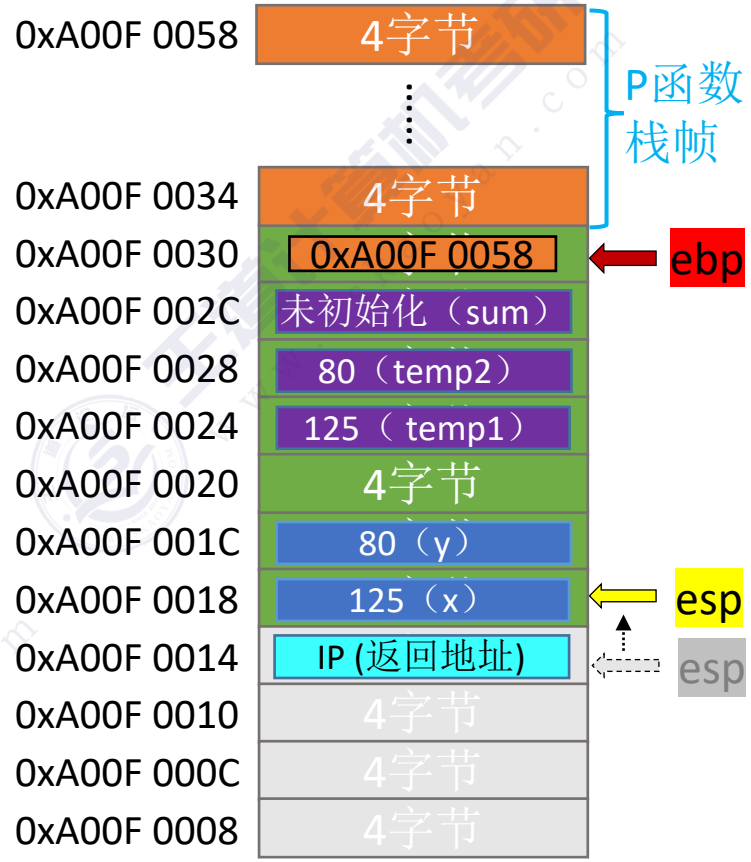
```
int add(int x, int y){
  return x+y;
}
```

汇编代码实战：函数调用返回




```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```

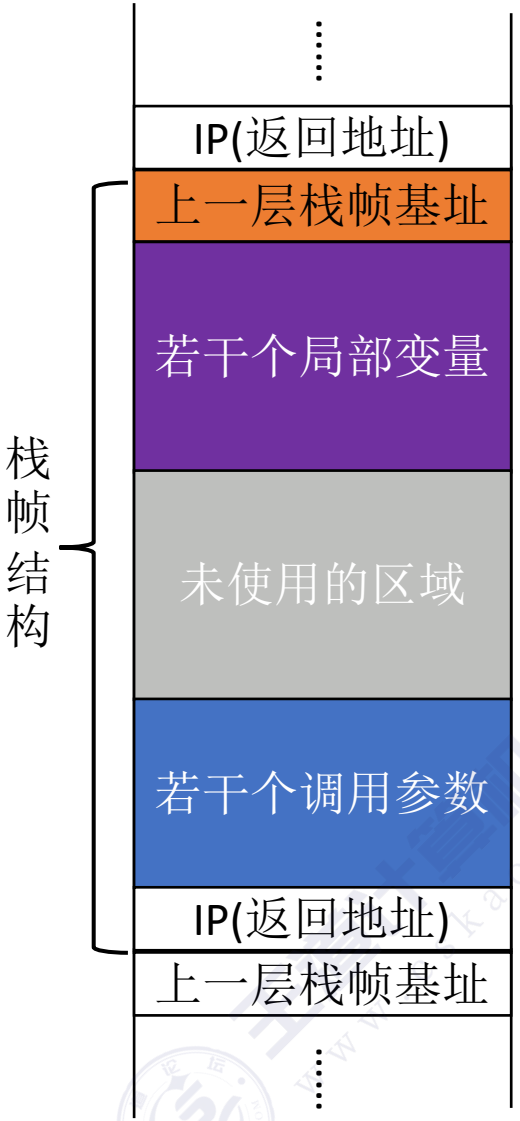



```
int caller(){
  int temp1=125;
  int temp2=80;
  int sum=add(temp1,temp2);
  return sum;
}
```

eax: 205　　edx: 125

```
int add(int x, int y){
  return x+y;
}
```

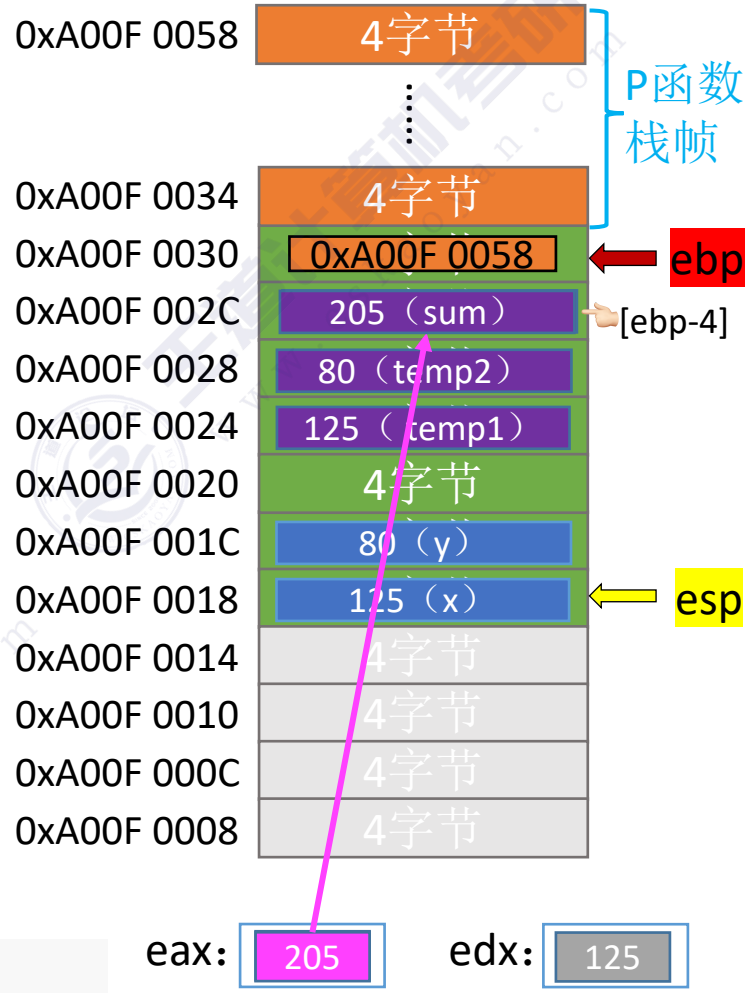
通常用eax寄存器返回结果



汇编代码实战：使用返回值

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

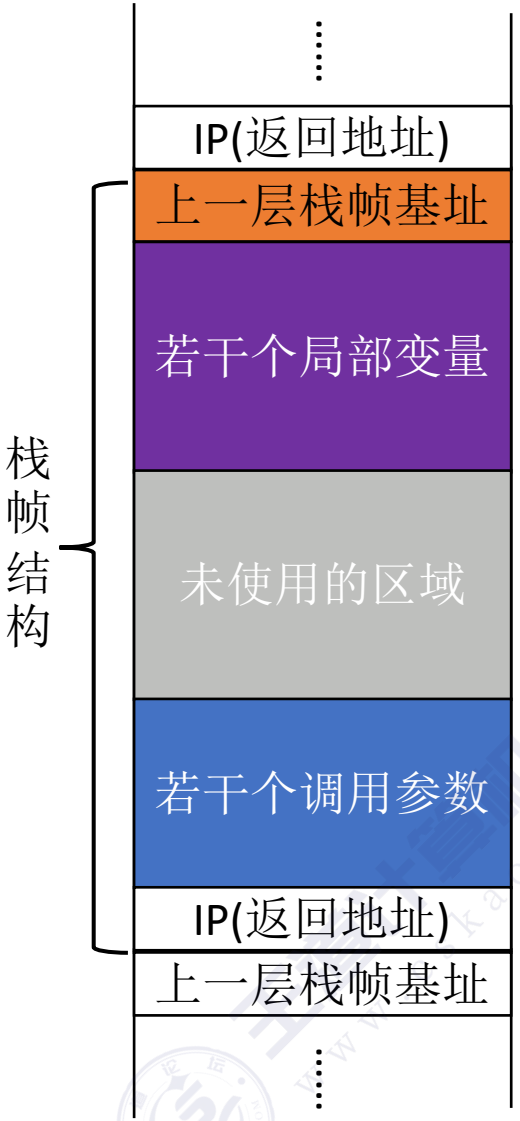
```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```




```
int caller(){
  int temp1=125;
  int temp2=80;
  int sum=add(temp1,temp2);
  return sum;
}
```

```
int add(int x, int y){
  return x+y;
}
```

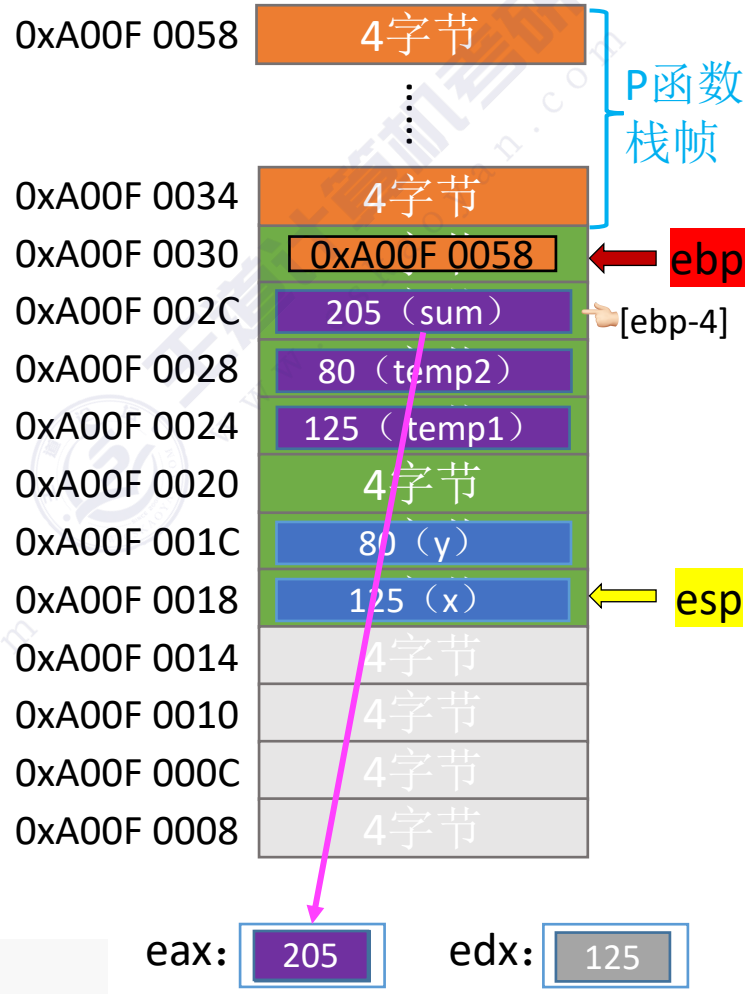
通常用eax寄存器返回结果



汇编代码实战：传递返回值

```
caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov [esp],eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret
```




```
int caller(){
  int temp1=125;
  int temp2=80;
  int sum=add(temp1,temp2);
  return sum;
}
```

```
int add(int x, int y){
  return x+y;
}
```

总结：函数调用的机器级表示

调用者：

……

- 将调用参数写入当前栈帧的顶部区域

可用 push 或 mov 指令实现

- 执行 call 指令

返回地址压入栈顶、并跳转到被调用函数第一条指令

- 使用返回值

通过eax寄存器

……

被调用者：

- 保存上一层函数栈帧，设置当前函数栈帧

push ebp
mov ebp,esp
或：enter指令

- 初始化局部变量

[ebp-4]、 [ebp-8]...

- 一系列处理逻辑
- 向上层函数传递返回值

通过eax寄存器

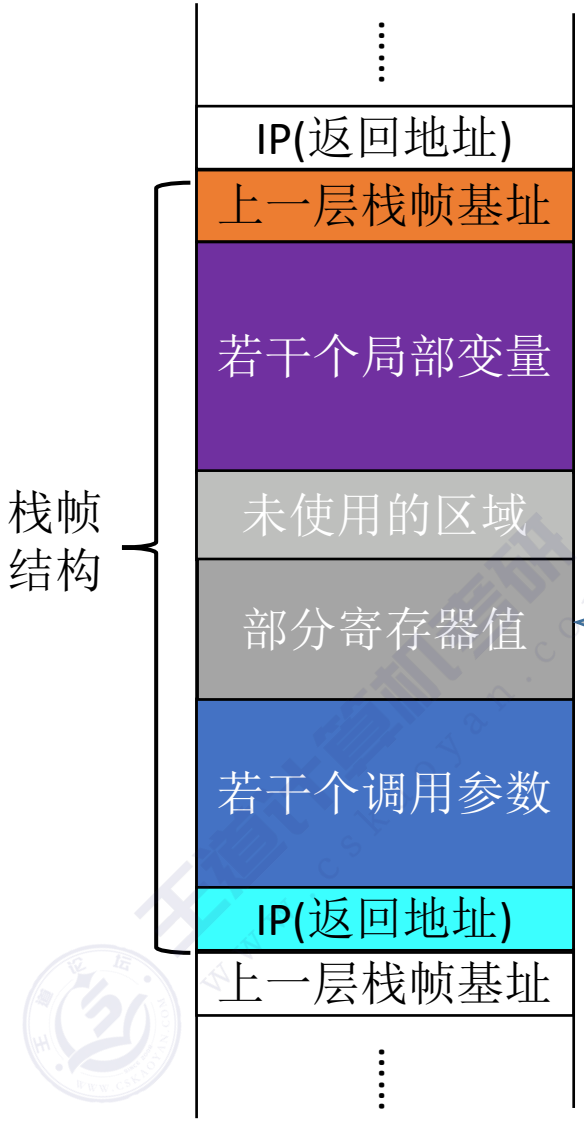
- 恢复上一层函数的栈帧

mov esp, ebp
pop ebp
或：leave指令

- 执行 ret 指令

从栈顶找到返回地址，出栈并恢复 IP 值

拓展：一个栈帧内可能包含哪些内容？



调用其他函数前，如果有必要，可将某些寄存器（如： eax、edx、ecx ）的值入栈保存，防止中间结果被破坏

不一定存在。如果这些寄存器值不是运算的中间结果，则可以不保存

总结：函数调用的机器级表示

调用者：

……

保存必要的寄存器

如 eax、edx、ecx

- 将调用参数写入当前栈帧的顶部区域

可用 push 或 mov 指令实现

- 执行 call 指令

返回地址压入栈顶、并跳转到被调用函数第一条指令

- 使用返回值

通过eax寄存器

恢复必要的寄存器

……

被调用者：

- 保存上一层函数栈帧，设置当前函数栈帧

push ebp
mov ebp,esp
或：enter指令

- 初始化局部变量

[ebp-4]、 [ebp-8]...

- 一系列处理逻辑
- 向上层函数传递返回值

通过eax寄存器

- 恢复上一层函数的栈帧

mov esp, ebp
pop ebp
或：leave指令

- 执行 ret 指令

从栈顶找到返回地址，出栈并恢复 IP 值