C++底层原理探究（工具篇）

原创 小张 小张的code世界 2021-12-19 16:05

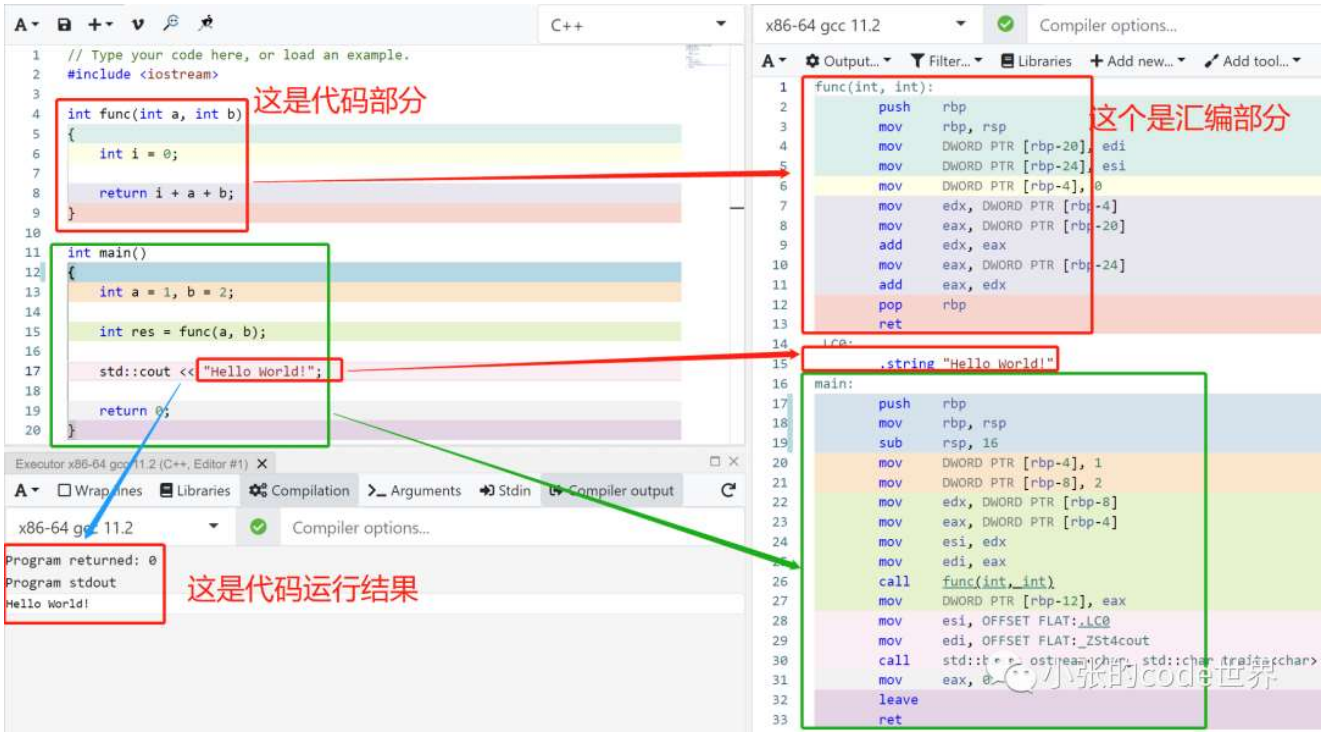
收录于合集
#领域驱动设计 1 #类的那些事儿 13

“工欲善其事必先利其器。”

在上一篇文章《C++底层原理探究（前言）》中，介绍了后面会分享学习C++底层原理的系列文章，大部分示例都会从汇编的角度去分析，当然涉及的汇编知识也不会太难，学习过《深入理解计算机系统》的读者一定不会有什么障碍。

但是，考虑到有一部分的读者，可能在这方面存在短板，我设计了这一篇文章。主要讲两部分内容，一部分就是**阅读简单的汇编代码**，另一部分就是推荐一个**学习工具——Compiler Explorer**。

Compiler Exporer是一个**在线编程工具**，非常的强大，可以到对应的汇编代码，也可以运行程序，还有其他一些高阶的操作。在此，把这个工具介绍给大家，大家可以视自己的需要去学习和使用。



0. 怎么阅读简单汇编程序

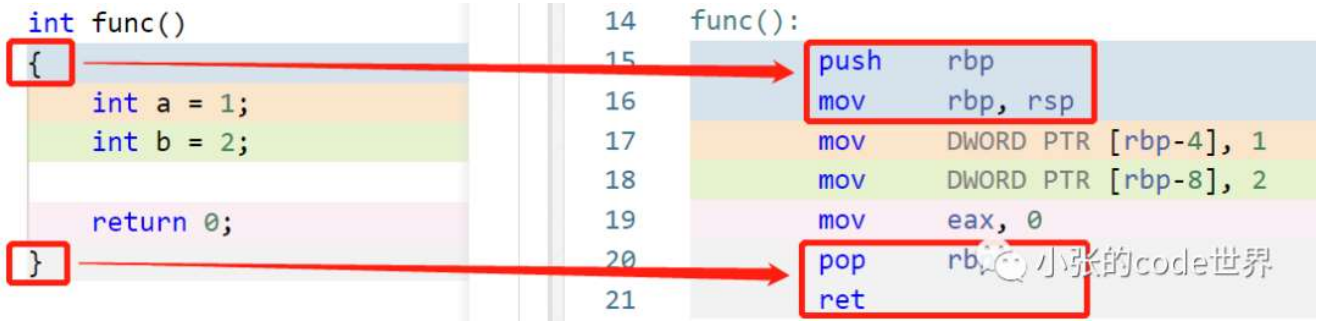
首先，我们看一个简单的汇编程序：

```
int func() // 函数名
{
    int a = 1; // 声明并初始化一个变量a
    int b = 2; // 声明并初始化一个变量b

    return 0; // 返回值为0
}
```

对应的汇编的代码如下：

```
func():
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], 1
        mov     DWORD PTR [rbp-8], 2
        mov     eax, 0
        pop     rbp
        ret
```

函数与汇编的对应关系如下：

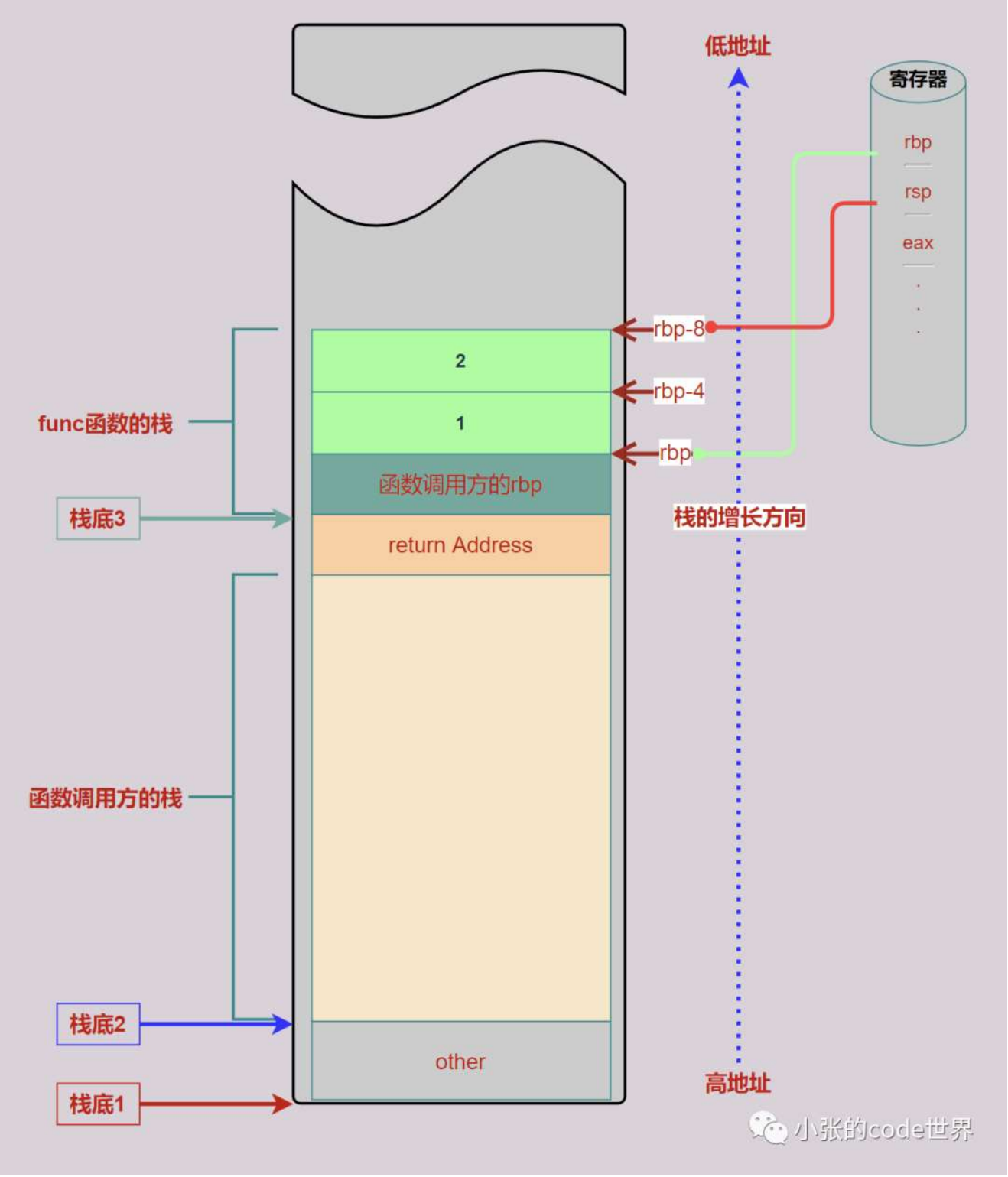


这里先对这个汇编代码逐行进行解释：

1. 函数名`func`；
2. 将函数调用方（调用`func`函数的一方）的栈基寄存器`rbp`的值压栈，保存在`func`函数栈中，因为寄存器`rbp`属于被调用者保护；
3. 将栈顶指针`rsp`的值存入寄存器`rbp`，此时的`rbp`已经变为`func`函数栈的栈底进入`func`函数栈；
4. 将四个字节的1存入栈地址为`rbp-4`处，可见栈地址`rbp-4`处被命名为变量`a`；
5. 将四个字节的2存入栈地址为`rbp-8`处，可见栈地址`rbp-8`处被命名为变量`b`；
6. 将返回值`0`存入寄存器`eax`，函数返回值总是放在`eax`寄存器中，函数调用方就可以通过寄存器`eax`拿到；
7. `rbp`从`func`函数栈中弹出，将`rbp`的值恢复为函数调用方的栈基指针；

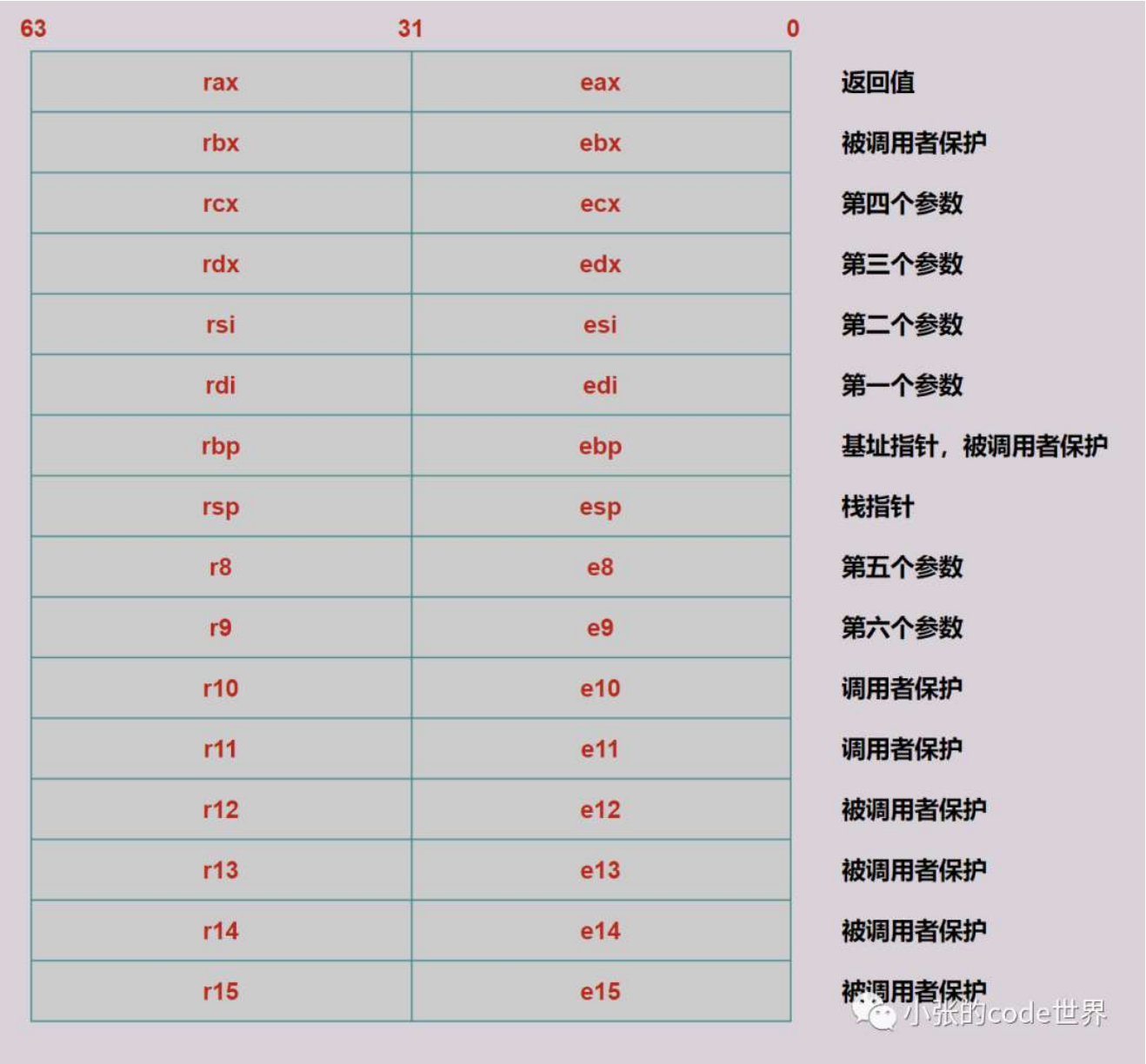
8. ret，返回到函数调用方。

栈帧的示意图如下：



寄存器基本知识

阅读了简单的程序对应的汇编语言后，介绍一下寄存器的基本知识。x86-64中的16个通用目的寄存器如下图所示。如果想详细的了解程序的机器级表示可以学习《深入理解计算机原理》的第三章《程序的机器表示》，详细的讲解了寄存器以及指令。



63	31	0
rax	eax	返回值
rbx	ebx	被调用者保护
rcx	ecx	第四个参数
rdx	edx	第三个参数
rsi	esi	第二个参数
rdi	edi	第一个参数
rbp	ebp	基址指针，被调用者保护
rsp	esp	栈指针
r8	e8	第五个参数
r9	e9	第六个参数
r10	e10	调用者保护
r11	e11	调用者保护
r12	e12	被调用者保护
r13	e13	被调用者保护
r14	e14	被调用者保护
r15	e15	被调用者保护

`rip`指令地址寄存器，用来存储 CPU 即将要执行的指令地址。每次 CPU 执行完相应的汇编指令之后，`rip`寄存器的值就会自行累加；`rip`无法直接赋值，`call, ret, jmp`等指令可以修改`rip`。

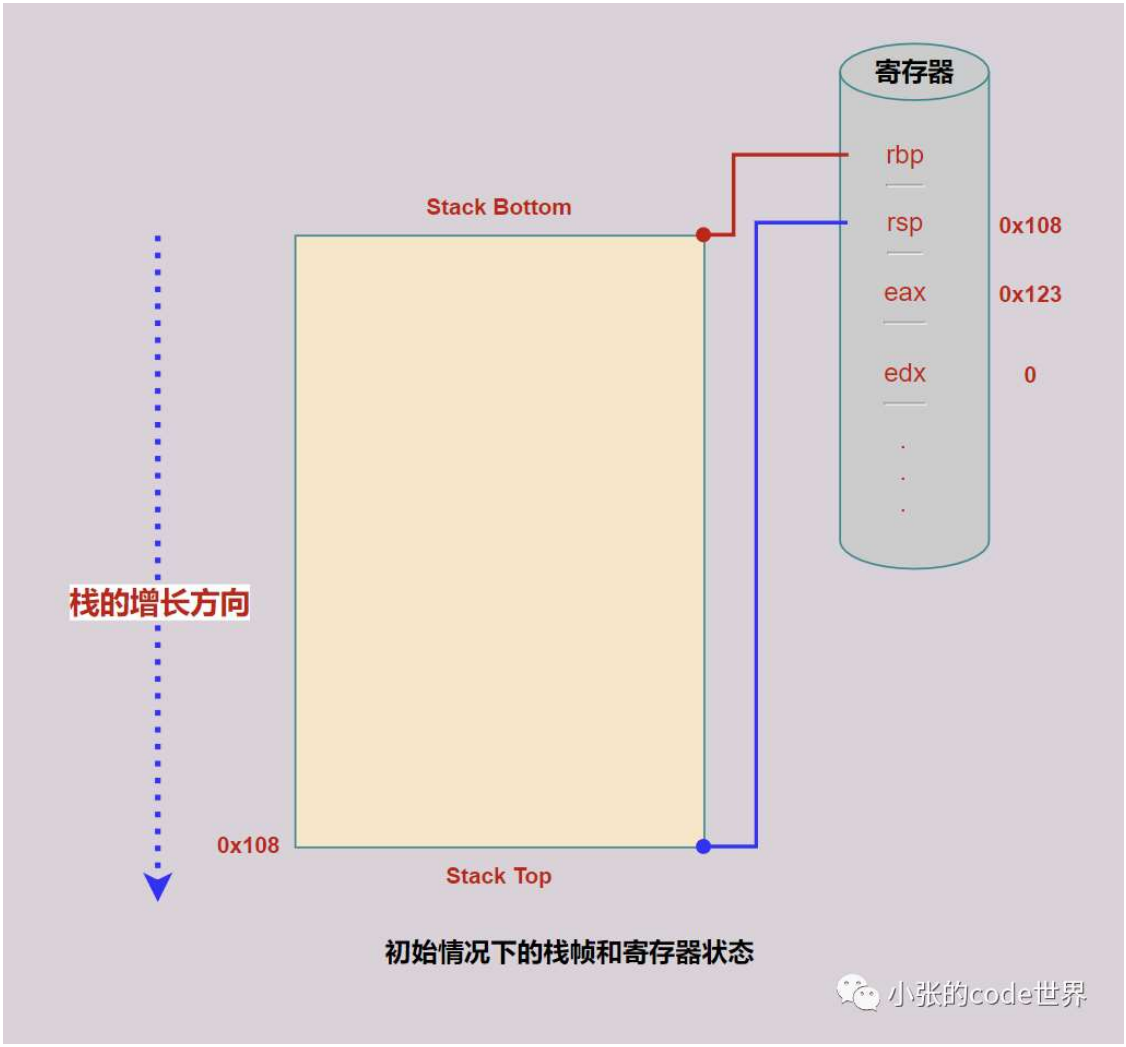
`rbp`栈基地址寄存器，保存当前栈帧的栈底地址。

`rsp`栈指针寄存器，保存当前栈顶。

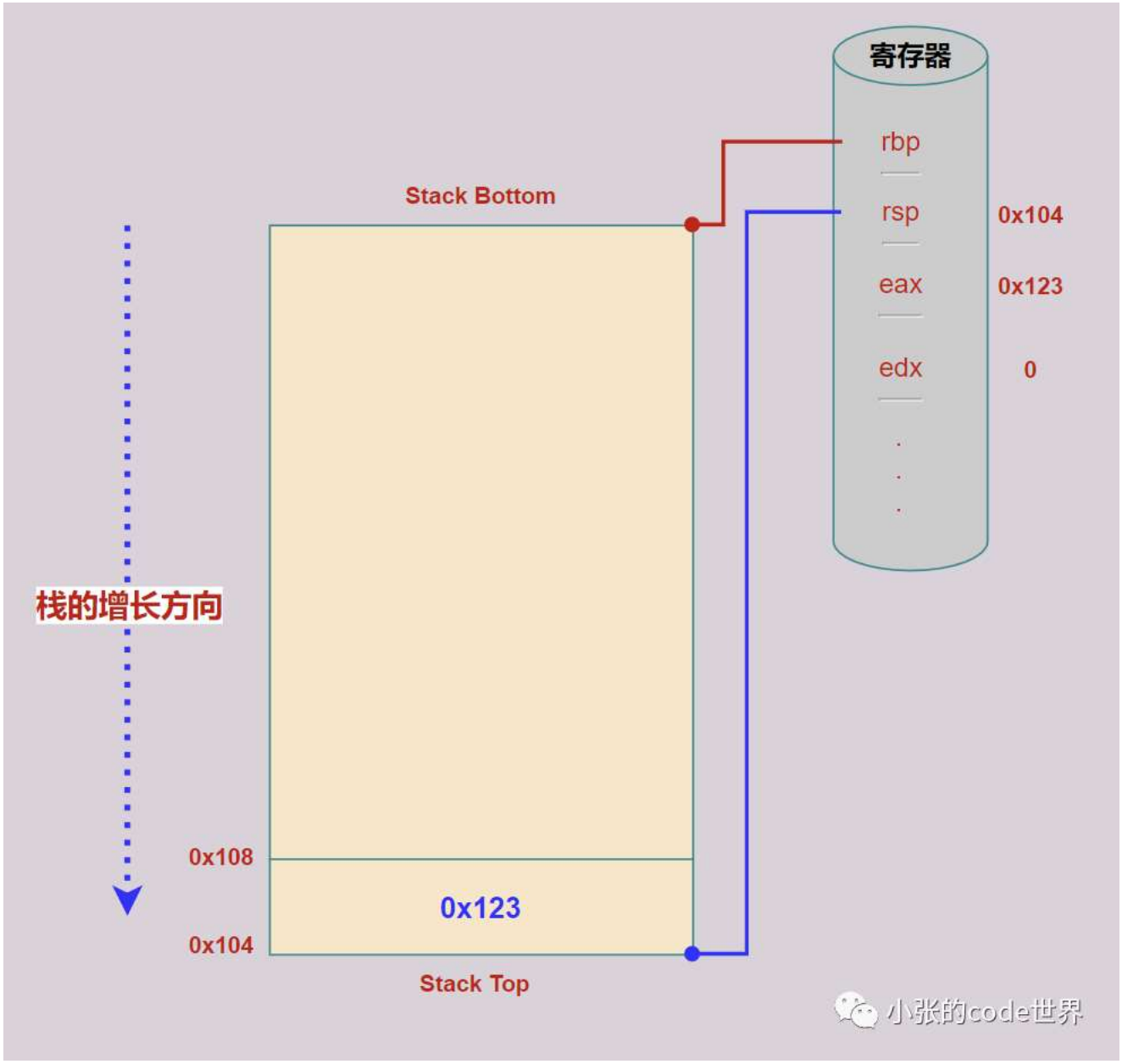
Intel的汇编格式是`opcode destination, source`，例如：
`mov rbp, rsp`表示的是将寄存器`rsp`的值存储到寄存器`rbp`中；
`mov DWORD PTR [rbp-4], 1`表示将四个字节的4存储到栈地址为`[rbp-4]`的内存中；
`push rbp`表示的是将的寄存器`rbp`的值压入程序栈中；
`pop rbp`表示的是将程序栈顶的值保存到寄存器`rbp`中。

我们注意到汇编总是以`push rbp`开始，以`pop rbp`结束，有的函数会以`leave`结束，实际等价于`pop rbp`。所以为大家解析一下`push`和`pop`的执行过程中，栈帧的变化。

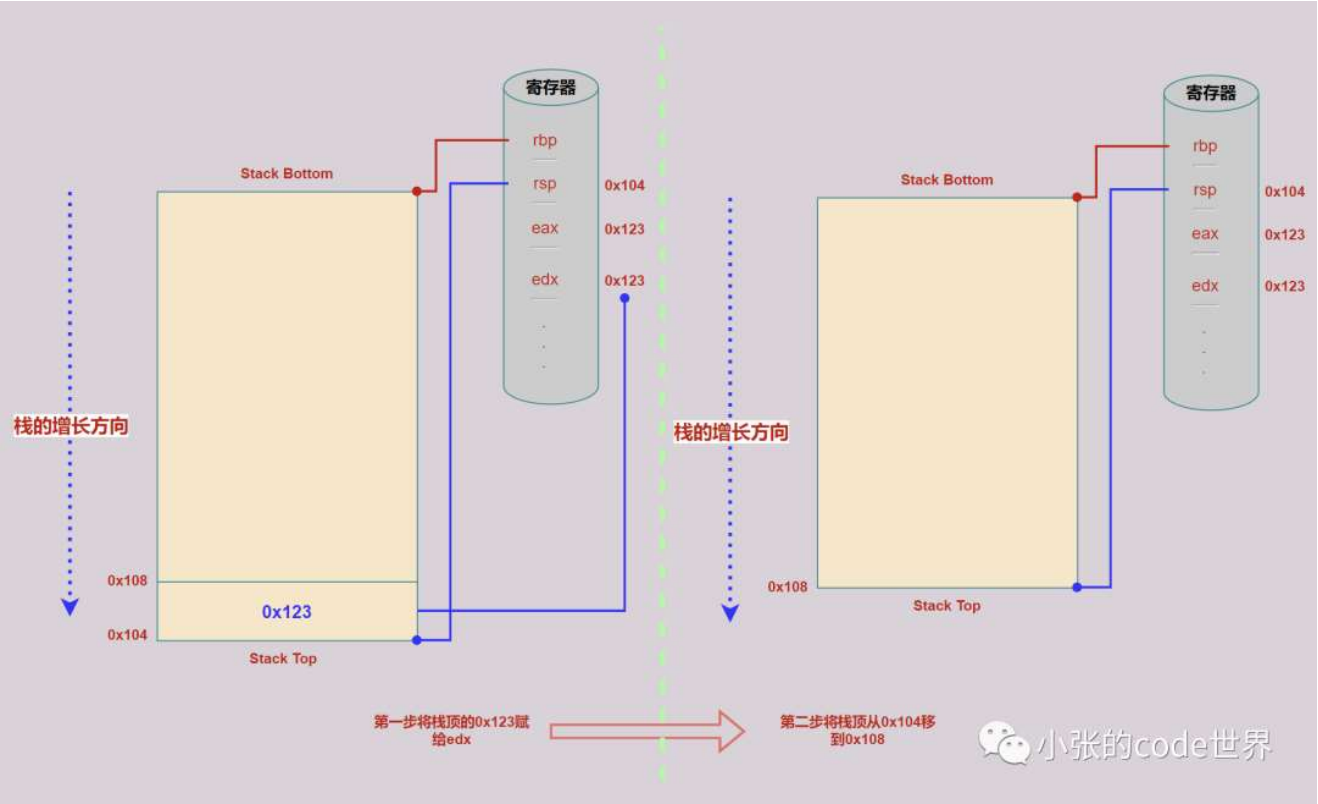
初始时的栈帧如下所示：



执行一次push eax之后为：



执行一次pop edx之后为：



有了这些基础就可以继续深入学习了。

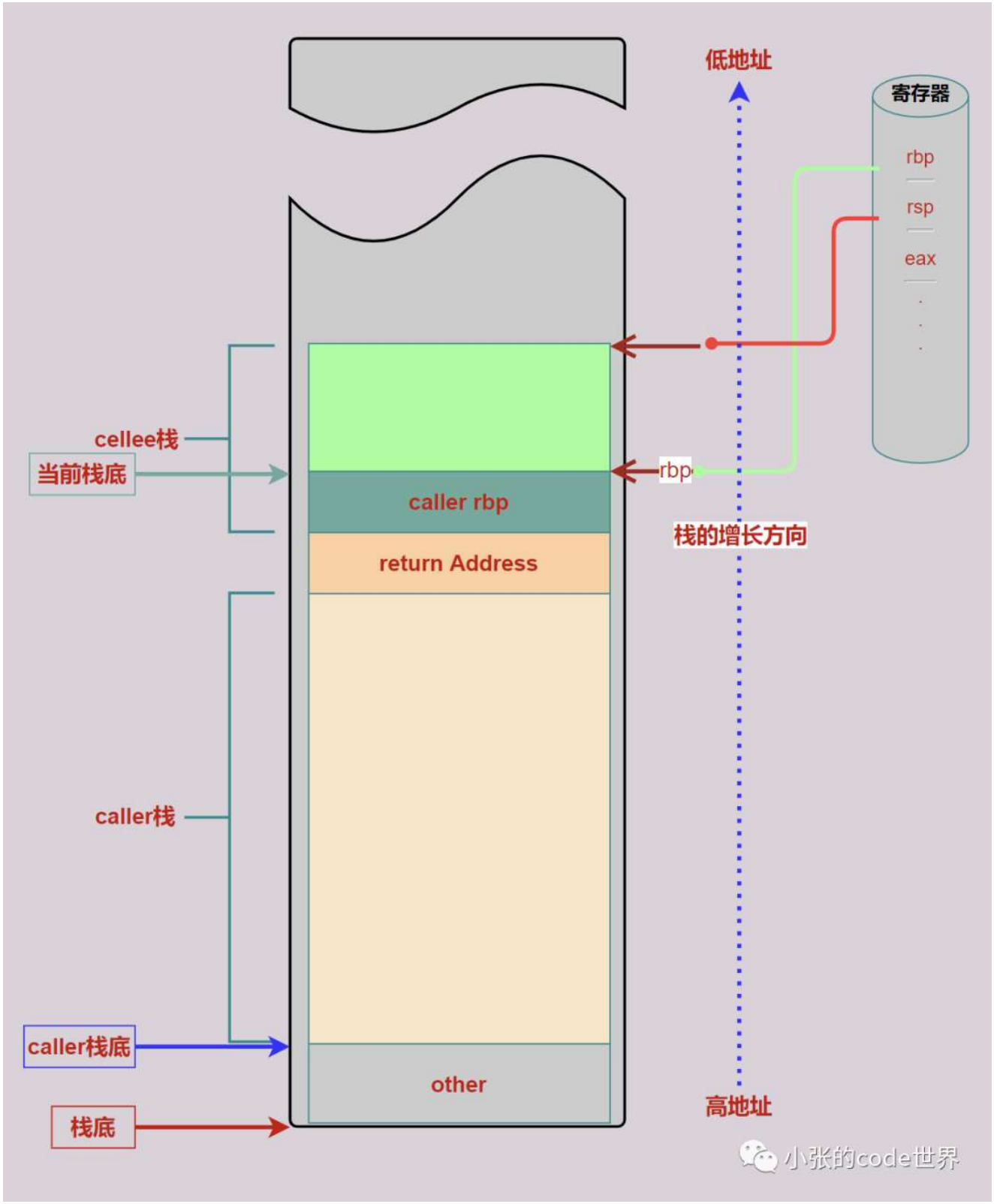
从函数调用看汇编

要理解汇编，光会看指令远远不够，需要通过函数的调用栈来理解汇编。

当调用函数时，就要压入一个新的栈帧，发起**调用函数（caller）**的栈帧称之为**调用者栈帧**，**被调用函数（callee）**的栈帧则称之为**当前栈帧**（由之前的分析我们可以看出`rsp`和`rbp`之间的内存空间就是当前栈帧）。

被调用的函数运行结束后回收栈帧，回到调用者栈帧。这一过程都是自动的，由系统分配与销毁，无需手动调度。栈帧中，最重要的是帧指针`rbp`和栈指针`rsp`，有了这两个指针，我们就可以刻画一个完整的栈帧。

如下图所示：



通过以下示例再理解一下函数调用的汇编，代码如下：

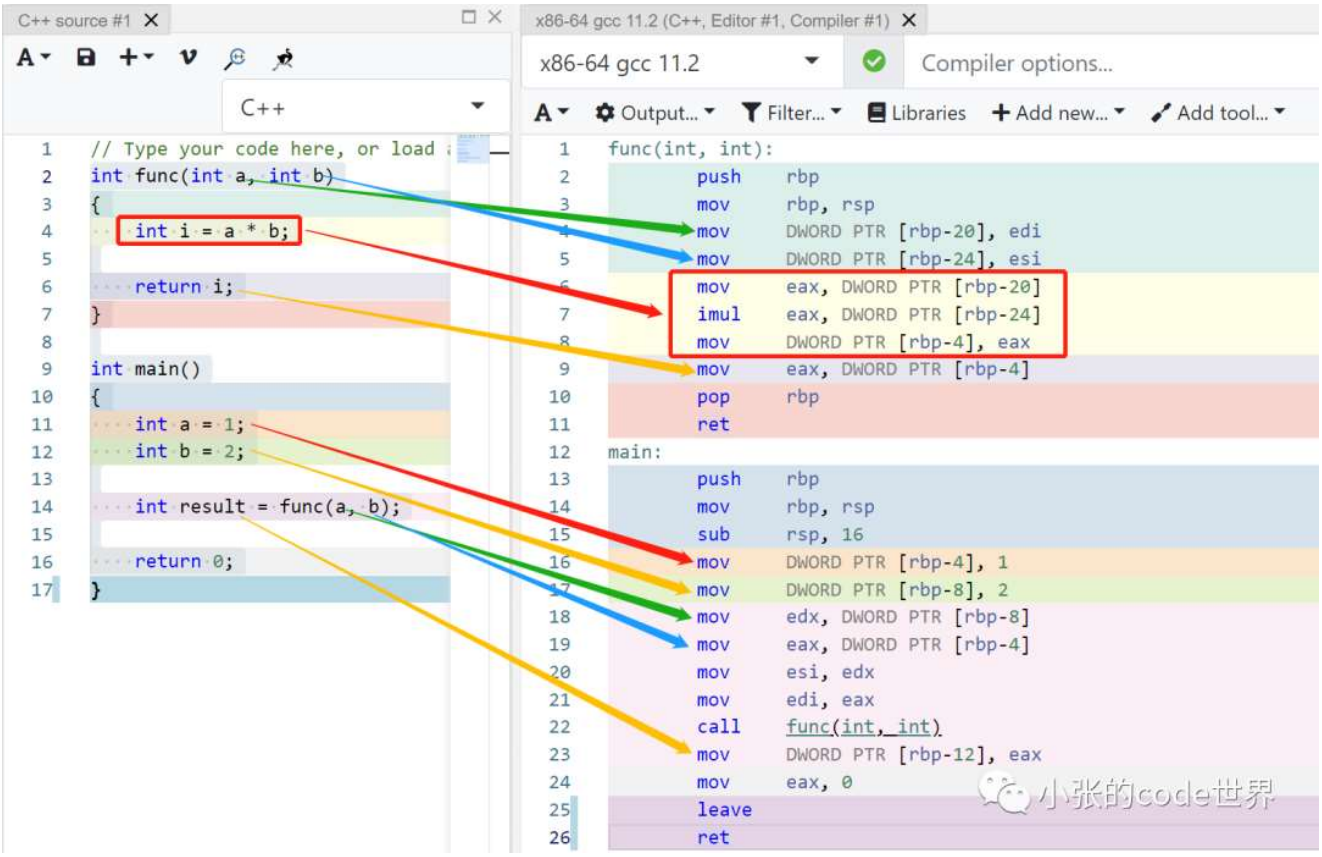
```
int func(int a, int b)
{
    int i = a * b;

    return i;
}
```

```
7
8  int main()
9  {
10     int a = 1;
11     int b = 2;
12
13     int result = func(a, b);
14
15     return 0;
16 }
```

对应的汇编代码如下：

```
1  func(int, int):
2          push    rbp
3          mov     rbp, rsp
4          mov     DWORD PTR [rbp-20], edi
5          mov     DWORD PTR [rbp-24], esi
6          mov     eax, DWORD PTR [rbp-20]
7          imul    eax, DWORD PTR [rbp-24]
8          mov     DWORD PTR [rbp-4], eax
9          mov     eax, DWORD PTR [rbp-4]
10         pop     rbp
11         ret
12 main:
13         push    rbp
14         mov     rbp, rsp
15         sub     rsp, 16
16         mov     DWORD PTR [rbp-4], 1
17         mov     DWORD PTR [rbp-8], 2
18         mov     edx, DWORD PTR [rbp-8]
19         mov     eax, DWORD PTR [rbp-4]
20         mov     esi, edx
21         mov     edi, eax
22         call    func(int, int)
23         mov     DWORD PTR [rbp-12], eax
24         mov     eax, 0
25         leave
26         ret
```

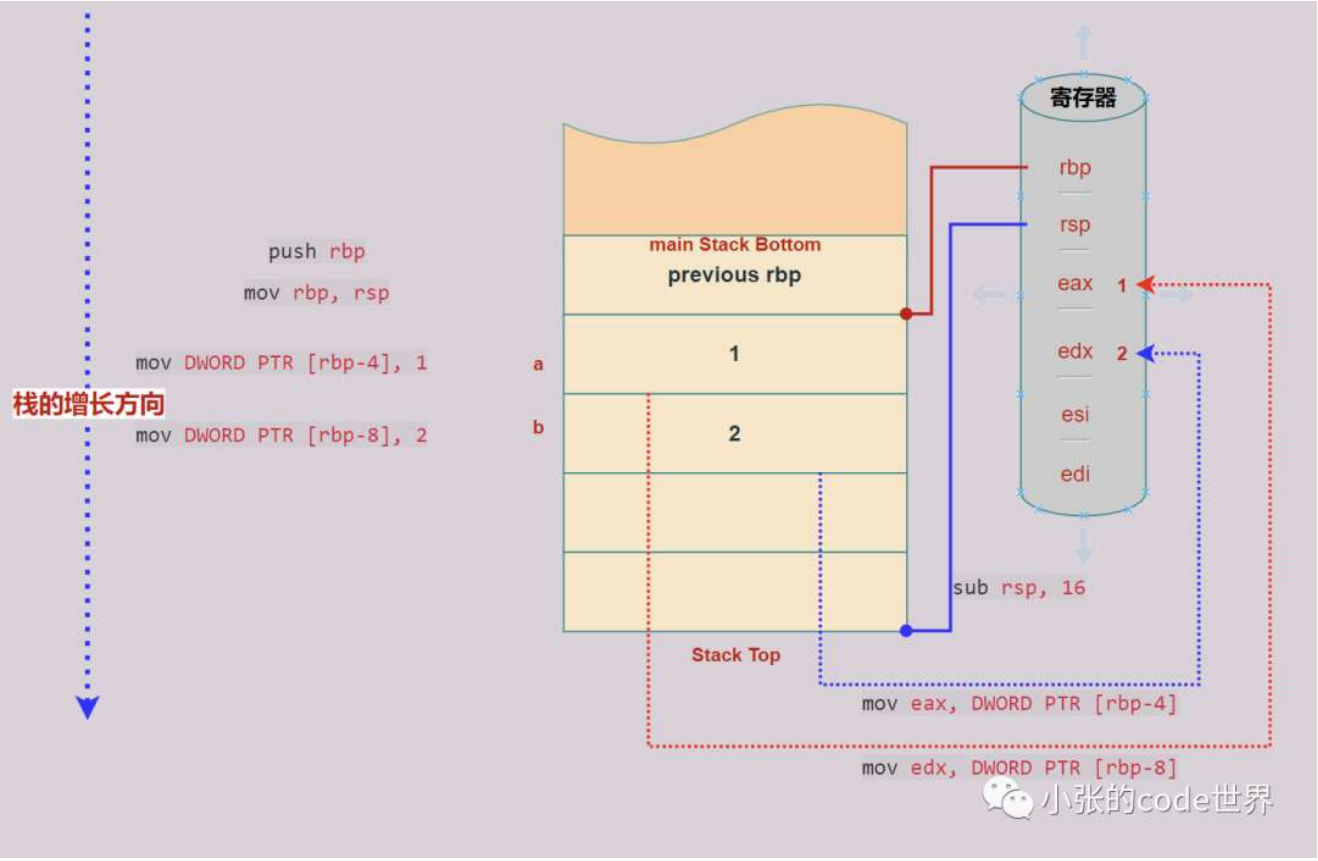
对应关系如下（C++代码和汇编按颜色对应）：



main通过call指令调用func(int, int)，在call指令之前的指令可以分为两部分，首先是寄存器保护和维护，以及数据维护：

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
mov     DWORD PTR [rbp-4], 1
mov     DWORD PTR [rbp-8], 2
mov     edx, DWORD PTR [rbp-8]
mov     eax, DWORD PTR [rbp-4]
```

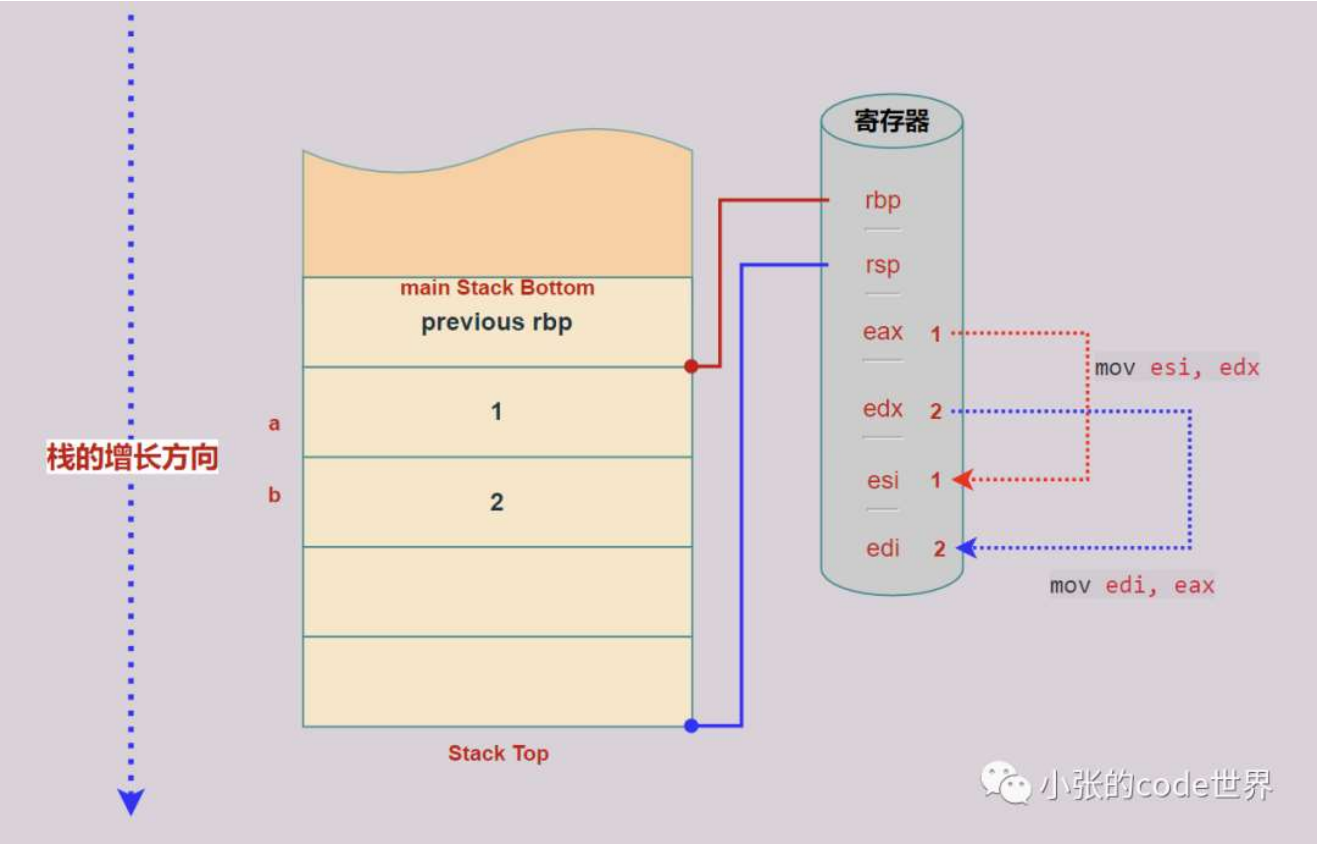
上述指令执行完成后的栈帧如图所示，在图中标识出了指令执行后对应的结果：



第二部分是，准备好参数传递（即将caller中的参数传递到callee中去），参数传递主要依赖于寄存器。也就是说callee的前6个入参可以使用寄存器传递，第一个参数放置在`esi`中，第二个参数放置在`edi`中，其他的参数也依次放置在对应的寄存器中。在callee中，可以到对应的寄存器中取相应的值。

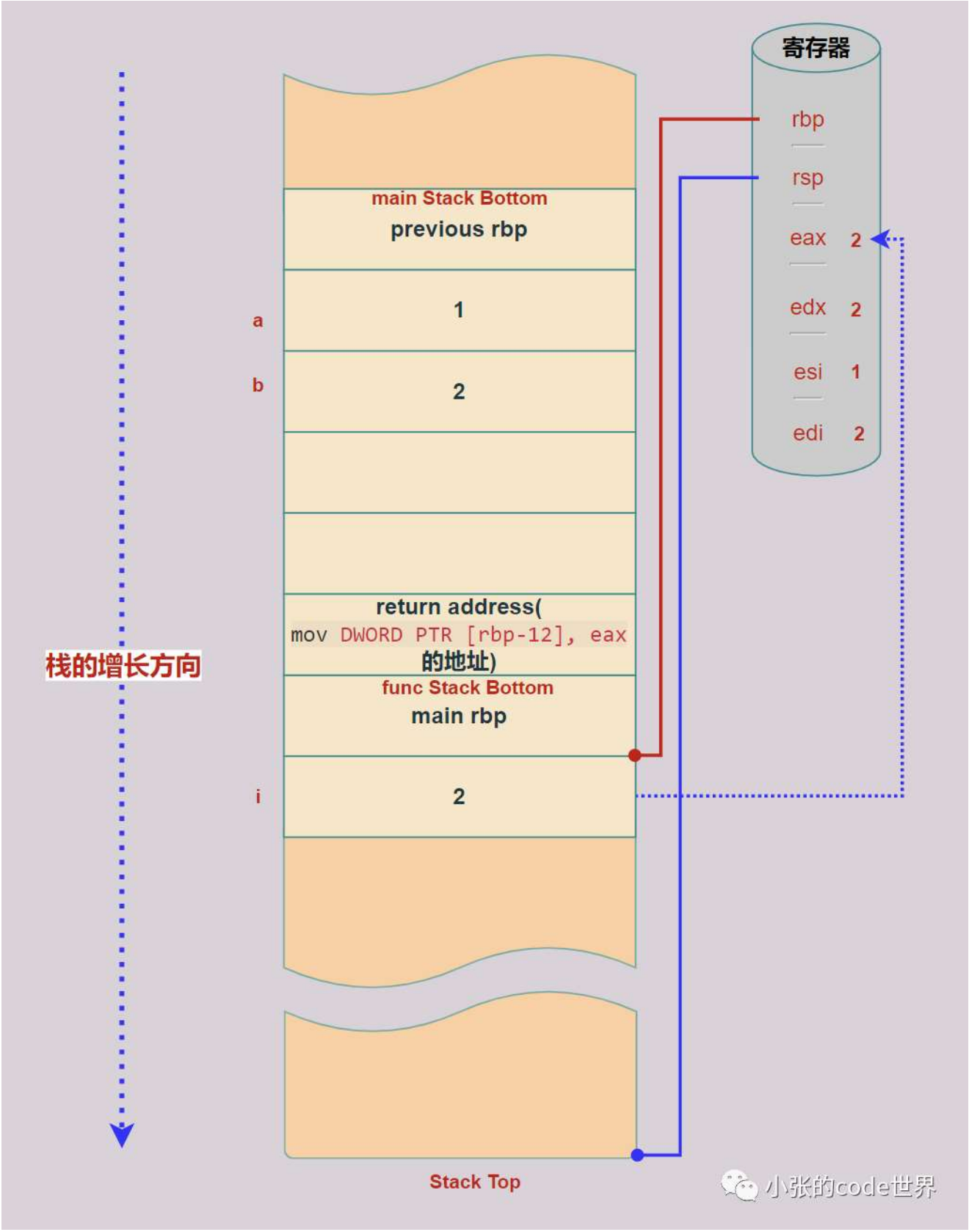
由于`func(int, int)`只有两个入参，因此只需要两个寄存器：

```
1         mov     esi, edx
2         mov     edi, eax
```



通过指令跳转到`func(int, int)`中去执行。和`main`一样，首先进行寄存器保护和维护，以及数据维护；然后进行计算，并将返回值放入寄存器`eax`中。

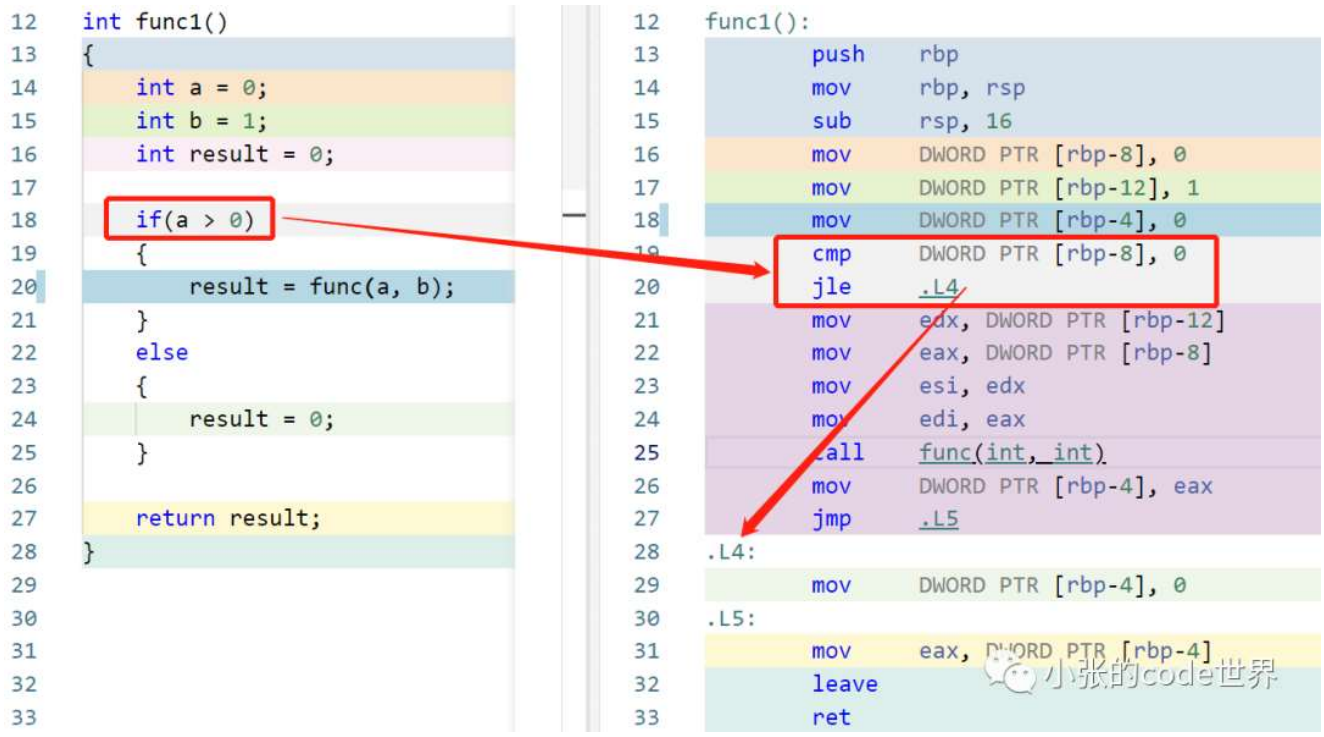
```
1         push    rbp
2         mov     rbp, rsp
3         mov     DWORD PTR [rbp-20], edi
4         mov     DWORD PTR [rbp-24], esi
5         mov     eax, DWORD PTR [rbp-20]
6         imul    eax, DWORD PTR [rbp-24]
7         mov     DWORD PTR [rbp-4], eax
8         mov     eax, DWORD PTR [rbp-4]
```



最后，通过pop rbp恢复寄存器rbp为caller的栈底，并通过ret指令返回到caller的栈中，继续执行mov DWORD PTR [rbp-12], eax，将func(int, int)的返回值赋值给result。

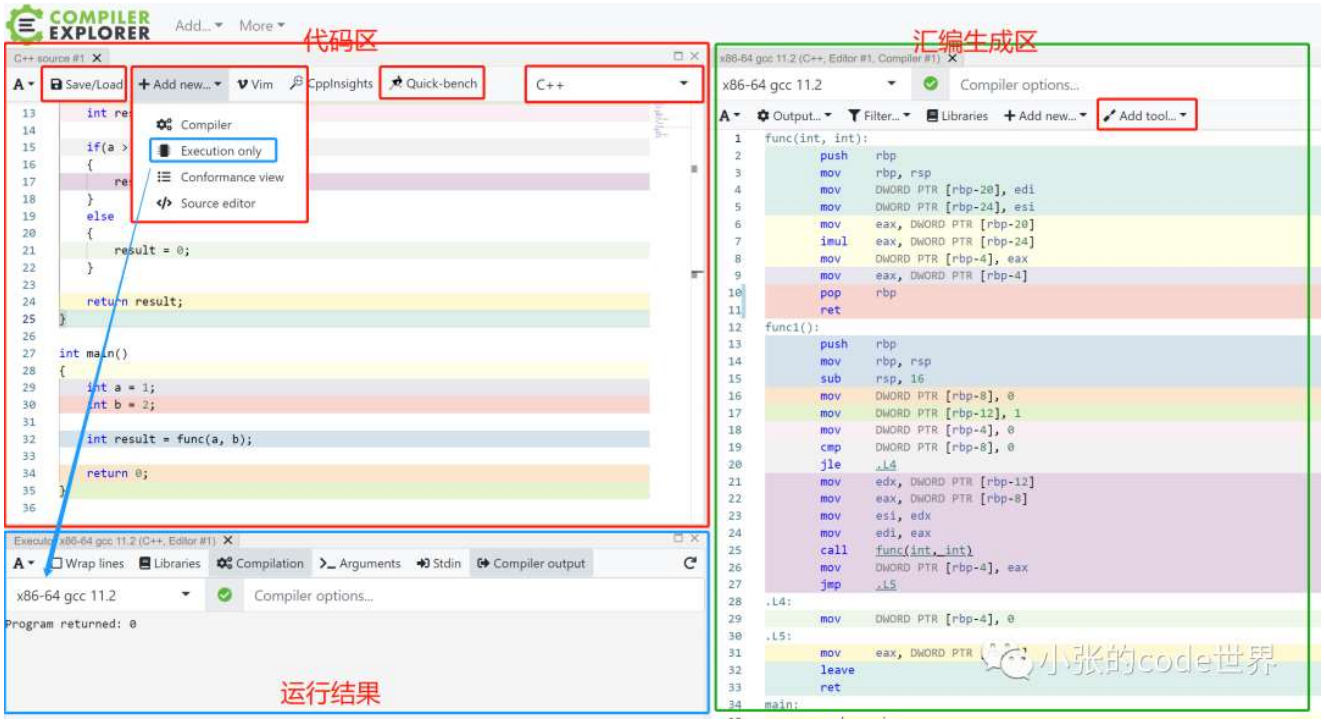
控制语句

如下图所示，大家可以使用Compiler Explorer去生成和学习。控制语句的汇编通过cmp指令计算一个结果，修改标志寄存器，指令jle再去检查对应的值，来判断跳转到哪里去执行。

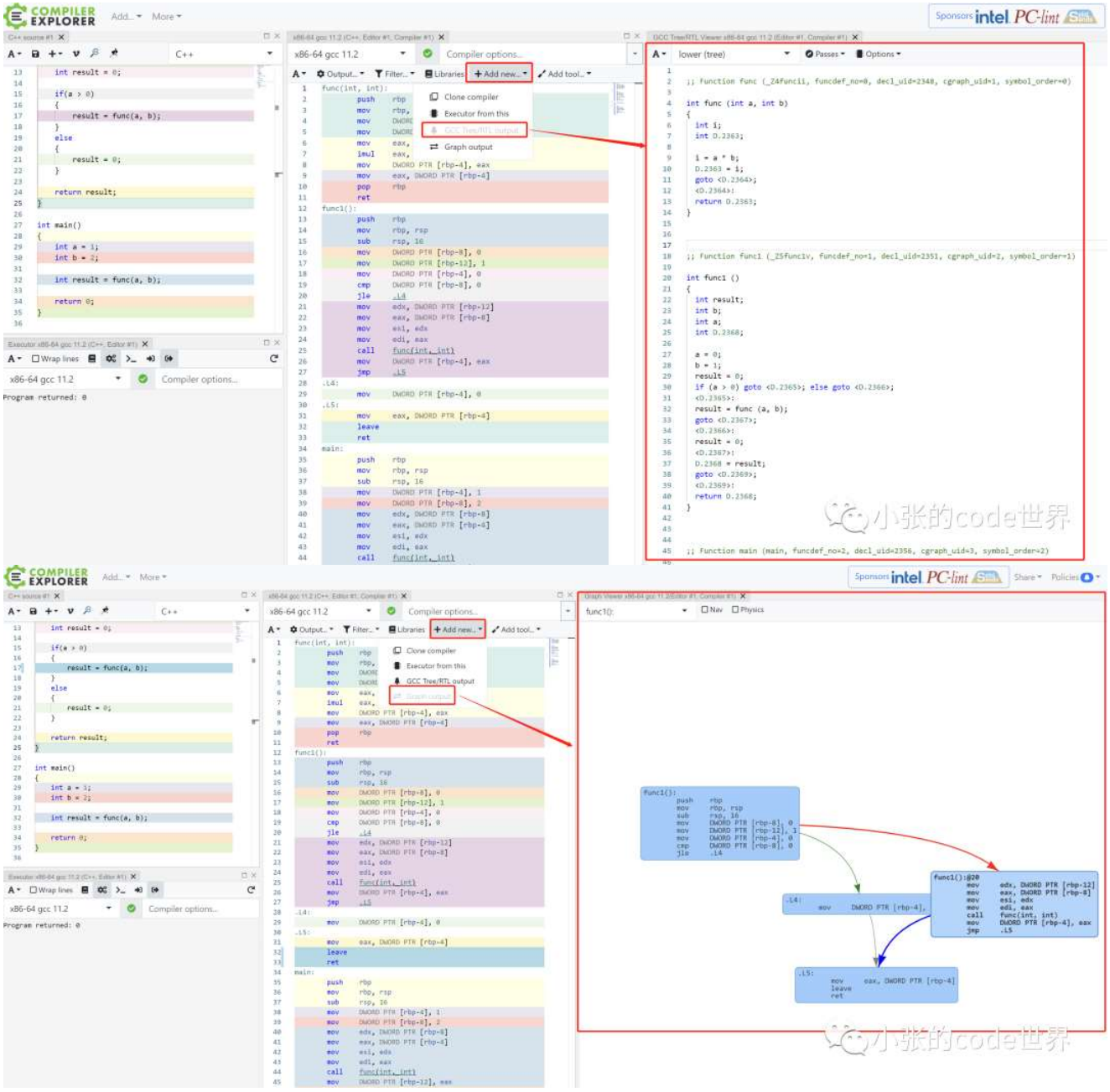


1. Compiler Explorer介绍

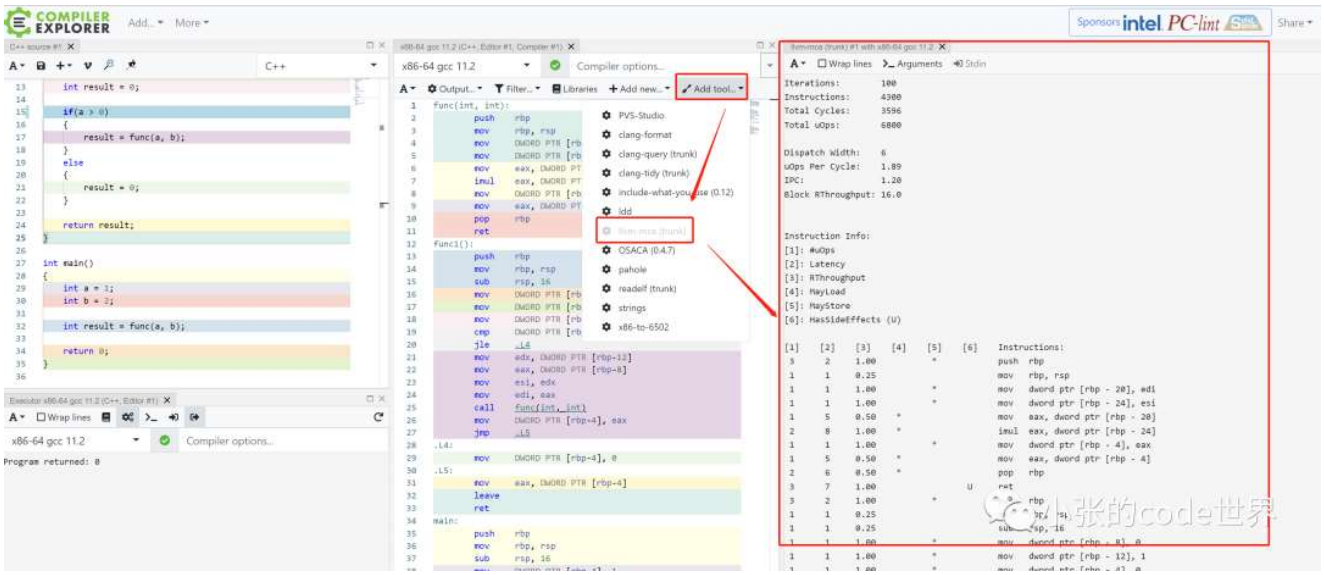
接下来介绍一下Compiler Explorer。日常使用的话，使用图中标识出来的三部分就行了。代码和汇编每一句都是一一对应，通过颜色来链接。



可以查看编译器细节的工具：


COMPILER EXPLORER
Sponsors intel PC-lint
x86-64 gcc 11.2
Compiler options...
Clone compiler
Executor from this
GCC Tree/RTL output
Graph output
Add new...
Add tool...
Program returned: 0
小张的code世界

提供性能分析的工具：


COMPILER EXPLORER
Sponsors intel PC-lint
x86-64 gcc 11.2
Compiler options...
Add tool...
PVS-Studio
clang-format
clang-query (trunk)
clang-tidy (trunk)
include-what-you-use (0.12)
ldd
OSACA (0.4.7)
pahole
readelf (trunk)
strings
x86-to-6502
Program returned: 0
小张的code世界

关于Compiler Explorer的更多介绍可以参见B站上的介绍视频，视频的链接为：https://b23.tv/BV1pJ411w7kh/p93

喜欢此内容的人还喜欢

设计模式之命令模式（二）
小张的code世界