函数调用栈

因为函数调用栈涉及到操作系统的一些基础知识，因此先了解一些基础概念。

RAM 和 ROM

RAM（random access memory），又称为随机存取存储器，其特点是在断电时会丢失其存储的内容，因此主要用于存储短时间使用的程序。

ROM (random only memory)，只读内存，所存数据稳定，断电后所存的数据也不会更改。

寄存器

寄存器是CPU用来暂存指令、数据和地址的电脑存储器。寄存器是CPU的组成部分，因为在CPU内部，因此CPU对寄存器的的读写速度是非常快的，不需要IO传输。

以通用的X86-64架构来说，共有16个64位的通用寄存器，各寄存器的功能如下：

Register	Callee Save	Description
`%rax`		result register; also used in `idiv` and `imul` instructions.
`%rbx`	yes	miscellaneous register
`%rcx`		fourth argument register
`%rdx`		third argument register; also used in `idiv` and `imul` instructions.
`%rsp`		stack pointer
`%rbp`	yes	frame pointer
`%rsi`		second argument register
`%rdi`		first argument register
`%r8`		fifth argument register
`%r9`		sixth argument register
`%r10`		miscellaneous register
`%r11`		miscellaneous register
`%r12-%r15`	yes	miscellaneous registers

具体来说：

1. rax通常用于存储函数的返回结果，同时也用于乘法和除法指令中。
2. rsp是栈指针寄存器，通常指向栈顶位置，栈的pop和push操作，就是通过改变rsp的指向来实现的。
3. rbp是帧指针，用于表示当前帧的起始位置
4. %rdi, %rsi, %rdx, %rcx,%r8, %r9 六个寄存器用于存储函数调用时的6个参数
5. 被标识为“miscellaneous registers”的寄存器，属于通用性更为广泛的寄存器，编译器或汇编程序可以根据需要存储任何数据

指针指向值和存储值

区分一下指针指向值和存储值，是两个不同的概念。一个存储单元空间，有两个属性：

1. CPU访问这个存储单元需要依赖的地址值；
2. 这个存储单元所存储的数值，空间地址值和空间内存储的数值

比如说，%ebp寄存器存的指针地址是0x00004，地址0x00004处存放的是另一个值。存放的值可以是一个数值，也可以是另一个地址。

%ebp正是利用了这一点，%ebp寄存器存的是一个指针地址，该地址处存放的是另一个地址。

esp 和 ebp

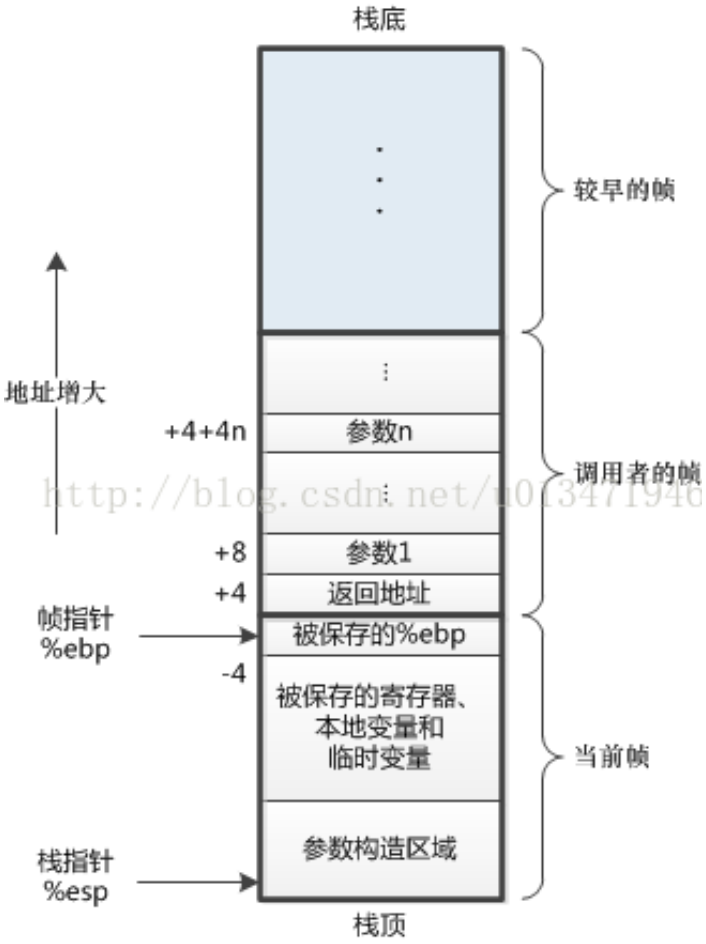
ebp指向的是当前调用者帧的地址，esp指向的是整个栈的栈顶地址。ebp和esp之间的就是当前栈帧。ebp标志起始地址，esp标志结束地址，这两个地址分别存储在%ebp寄存器和%esp寄存器中。

也就是说，当前栈帧的起始地址存储于%ebp寄存器中，结束地址存储于%esp寄存器中。

函数调用栈

了解了一些基本概念后，来看一下函数调用栈。

先看一张网上经典的图：



介绍函数调用栈、或者栈帧时，经常用到的就是这张图。这张图很经典，不过对于没相关知识的人来说有点难于理解。

因此，我自己做了两张图便于理解。实际上也没什么新内容，只不过是对于上图的拆分。

先看下测试代码：

```
int son(int x, int y) {
    int total = x + y;
    return total;
}


int father(int x, int y) {
    return son(x, y);
}


int main() {
    int a = 5, b = 10;
    int sum = father(a, b);
    return 0;
}
```

代码很简单，就是main函数调用了father()函数，father()函数调用了son()函数。

计算机用栈来传递函数参数，存储返回信息。栈中又分为多个帧，一个函数占用一个帧。当调用函数时，会生成一个新的帧；当函数调用完毕时，会通过指针移动，释放一个帧。

栈分配的方向是从高到低分配。也就是说，栈底的地址最高，栈顶的地址最低。

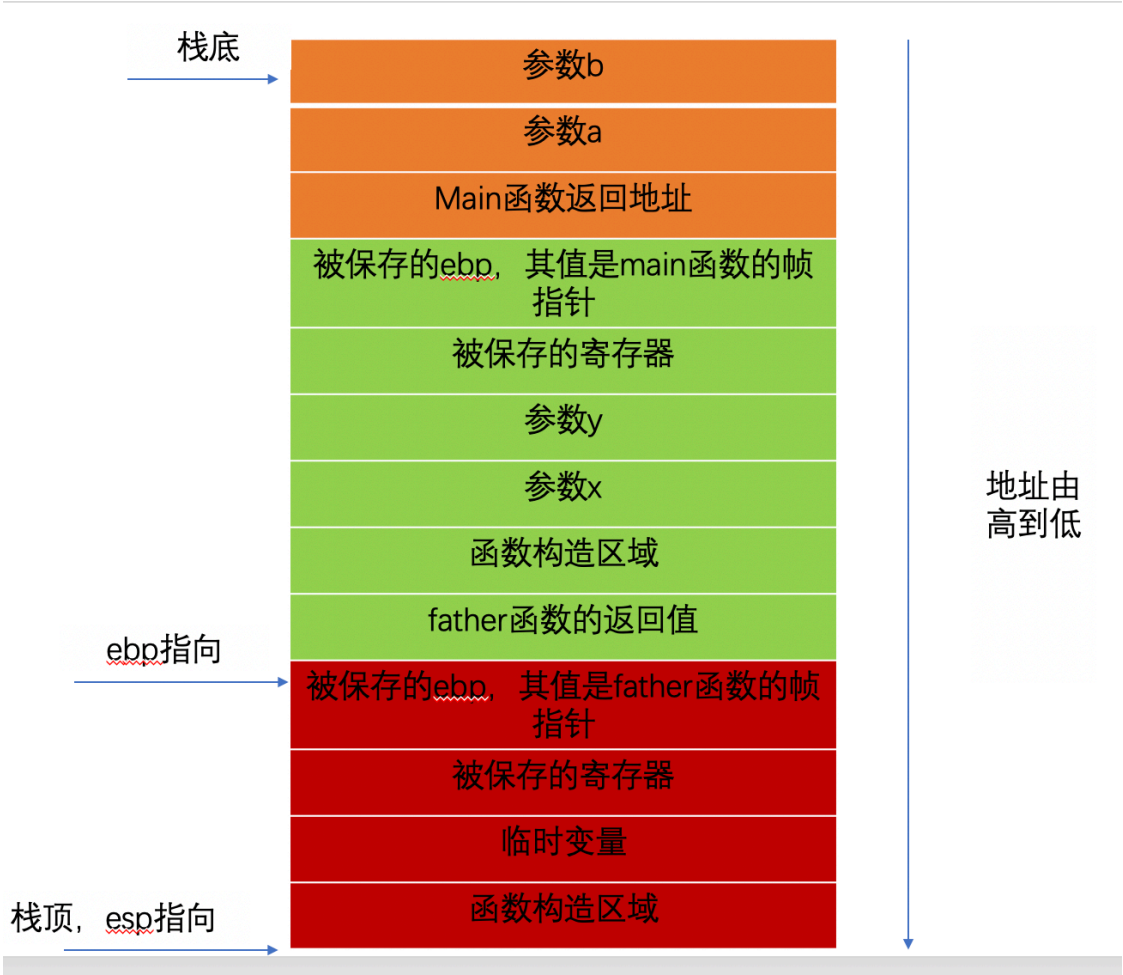
上面的示例代码中，执行到fahter函数，未执行到son函数时，栈帧的状态如下图：

其中，橙色区域是main函数的栈帧，绿色区域是father函数的栈帧。此时，栈顶就是father函数最后的位置，ebp指向fahter函数栈帧开始的位置，该位置内部所存的是main函数的帧指针。

注意，main栈帧最后存的是main函数的返回地址。其实就是执行完father函数后，应该返回到哪个位置继续执行main函数。

OK，到目前位置一切顺利。加入son函数再看一下。

当执行到son函数时，栈帧的状态如下图：



橙色区域为main函数的栈帧，绿色区域为father函数的栈帧，红色区域为son函数的栈帧。注意：

1. fahter函数的栈帧最后存的是father函数的返回值，其目的是执行完son函数后，能够继续执行father函数
2. esp，也就是栈顶指针会随着函数push不断向下移动，地址不断减小
3. ebp此时指向了son函数栈帧的起始值，其内保存的是father函数的帧指针

这样，当son函数执行完毕时，能够根据ebp的指向，找到father函数的位置。father函数执行完毕后，能够根据ebp的指向，找到main函数的位置。是不是类似于函数调用栈?

实际上，函数的调用栈就是根据栈帧指针得到的。