汇编语言入门教程

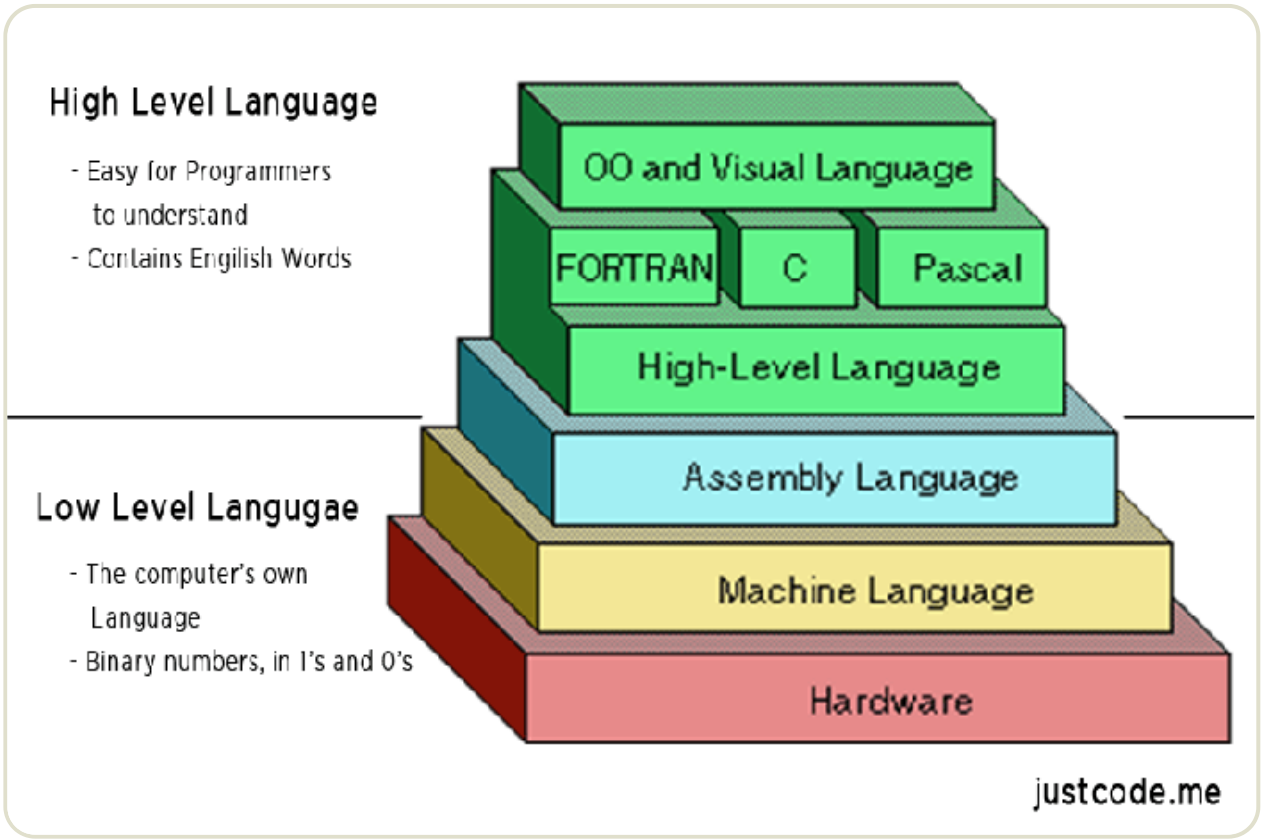
作者： 阮一峰

分享按钮

日期： 2018年1月21日

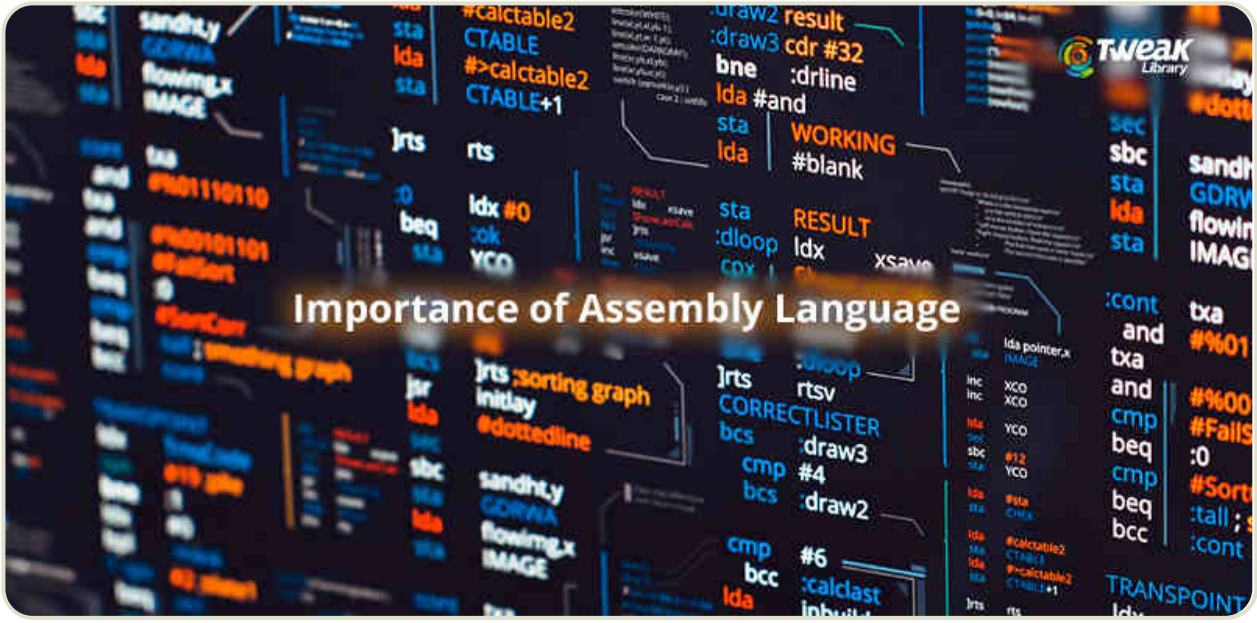
学习编程其实就是学高级语言，即那些为人类设计的计算机语言。

但是，计算机不理解高级语言，必须通过编译器转成二进制代码，才能运行。学会高级语言，并不等于理解计算机实际的运行步骤。



计算机真正能够理解的是低级语言，它专门用来控制硬件。汇编语言就是低级语言，直接描述/控制 CPU 的运行。如果你想了解 CPU 到底干了些什么，以及代码的运行步骤，就一定要学习汇编语言。

汇编语言不容易学习，就连简明扼要的介绍都很难找到。下面我尝试写一篇最好懂的汇编语言教程，解释 CPU 如何执行代码。

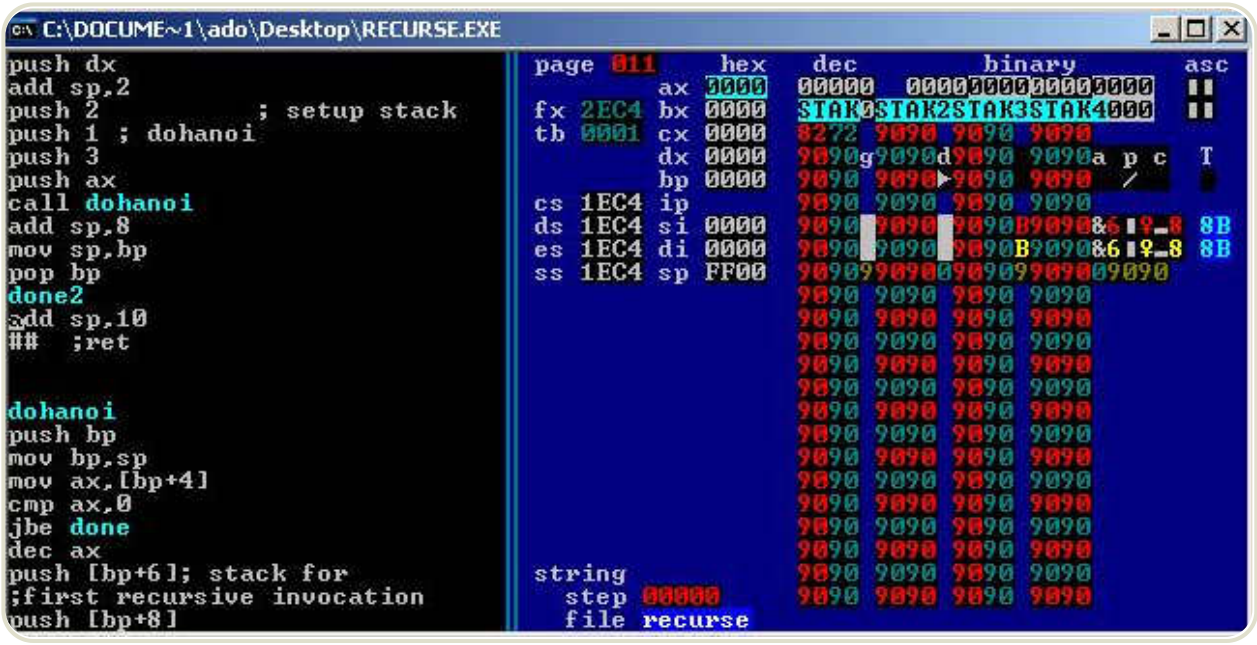


一、汇编语言是什么?

我们知道，CPU 只负责计算，本身不具备智能。你输入一条指令（instruction），它就运行一次，然后停下来，等待下一条指令。

这些指令都是二进制的，称为操作码（opcode），比如加法指令就是 `00000011` 。编译器的作用，就是将高级语言写好的程序，翻译成一条条操作码。

对于人类来说，二进制程序是不可读的，根本看不出来机器干了什么。为了解决可读性的问题，以及偶尔的编辑需求，就诞生了汇编语言。



汇编语言是二进制指令的文本形式，与指令是一一对应的关系。比如，加法指令 `00000011` 写成汇编语言就是 ADD。只要还原成二进制，汇编语言就可以被 CPU 直接执行，所以它是最底

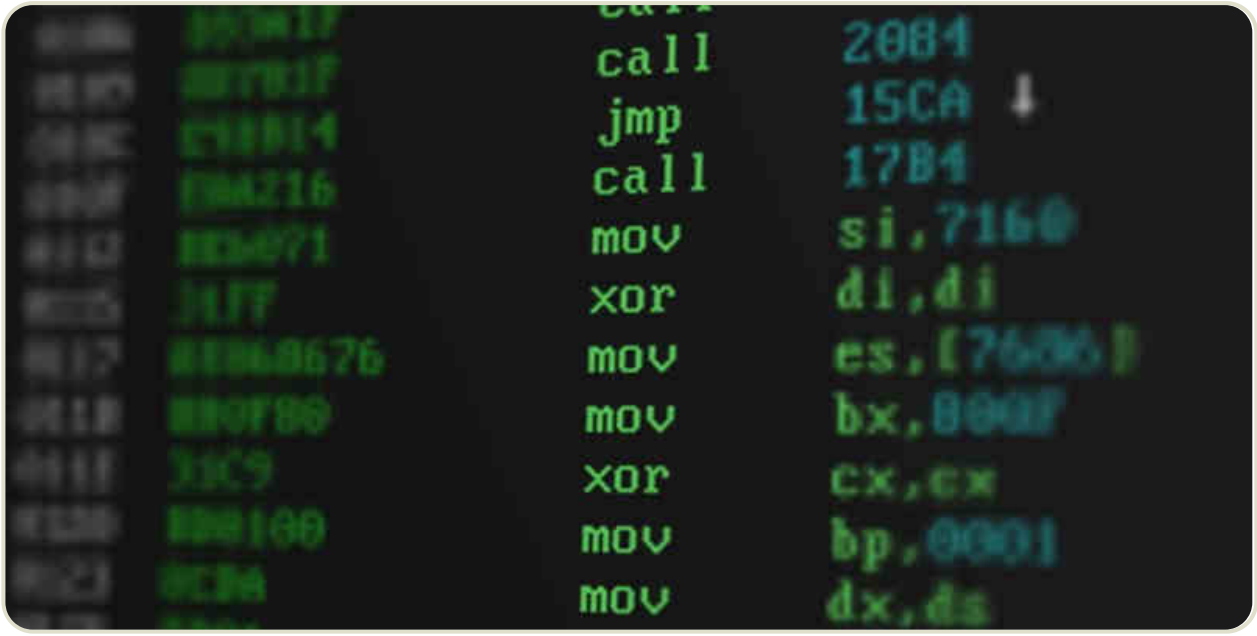
层的低级语言。

二、来历

最早的时候，编写程序就是手写二进制指令，然后通过各种开关输入计算机，比如要做加法了，就按一下加法开关。后来，发明了纸带打孔机，通过在纸带上打孔，将二进制指令自动输入计算机。

为了解决二进制指令的可读性问题，工程师将那些指令写成了八进制。二进制转八进制是轻而易举的，但是八进制的可读性也不行。很自然地，最后还是用文字表达，加法指令写成ADD。内存地址也不再直接引用，而是用标签表示。

这样的话，就多出一个步骤，要把这些文字指令翻译成二进制，这个步骤就称为 assembling，完成这个步骤的程序就叫做 assembler。它处理的文本，自然就叫做 aseembly code。标准化以后，称为 assembly language，缩写为 asm，中文译为汇编语言。



每一种 CPU 的机器指令都是不一样的，因此对应的汇编语言也不一样。本文介绍的是目前最常见的 x86 汇编语言，即 Intel 公司的 CPU 使用的那一种。

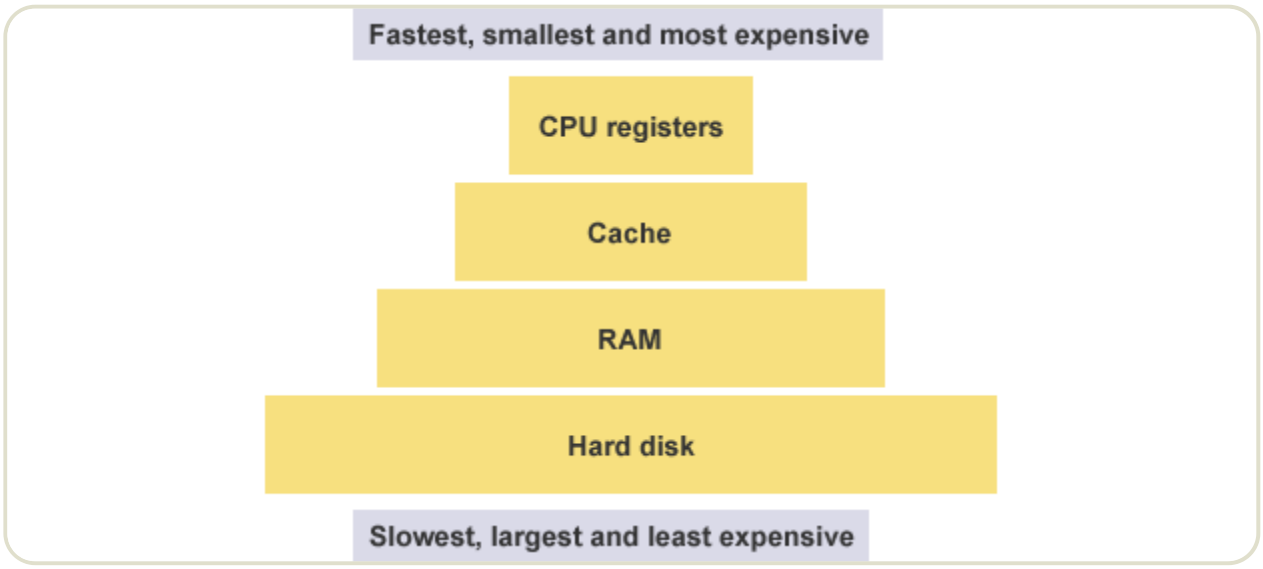
三、寄存器

学习汇编语言，首先必须了解两个知识点：寄存器和内存模型。

先来看寄存器。CPU 本身只负责运算，不负责储存数据。数据一般都储存在内存之中，CPU 要用的时候就去内存读写数据。但是，CPU 的运算速度远高于内存的读写速度，为了避免被

拖慢，CPU 都自带一级缓存和二级缓存。基本上，CPU 缓存可以看作是读写速度较快的内存。

但是，CPU 缓存还是不够快，另外数据在缓存里面的地址是不固定的，CPU 每次读写都要寻址也会拖慢速度。因此，除了缓存之外，CPU 还自带了寄存器（register），用来储存最常用的数据。也就是说，那些最频繁读写的数据（比如循环变量），都会放在寄存器里面，CPU 优先读写寄存器，再由寄存器跟内存交换数据。



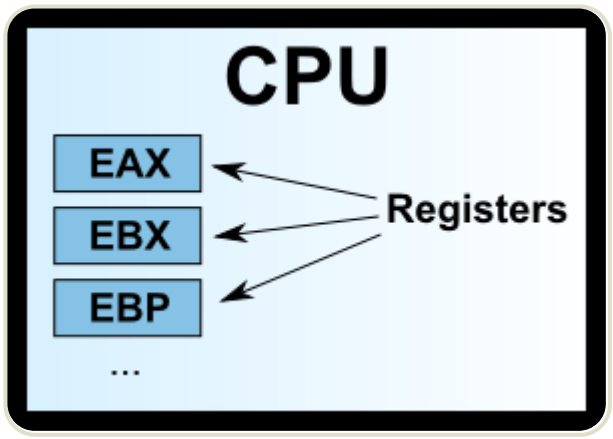
寄存器不依靠地址区分数据，而依靠名称。每一个寄存器都有自己的名称，我们告诉 CPU 去具体的哪一个寄存器拿数据，这样的速度是最快的。有人比喻寄存器是 CPU 的零级缓存。

四、寄存器的种类

早期的 x86 CPU 只有8个寄存器，而且每个都有不同的用途。现在的寄存器已经有100多个了，都变成通用寄存器，不特别指定用途了，但是早期寄存器的名字都被保存了下来。

- EAX
- EBX
- ECX
- EDX
- EDI
- ESI
- EBP
- ESP

上面这8个寄存器之中，前面七个都是通用的。ESP 寄存器有特定用途，保存当前 Stack 的地址（详见下一节）。

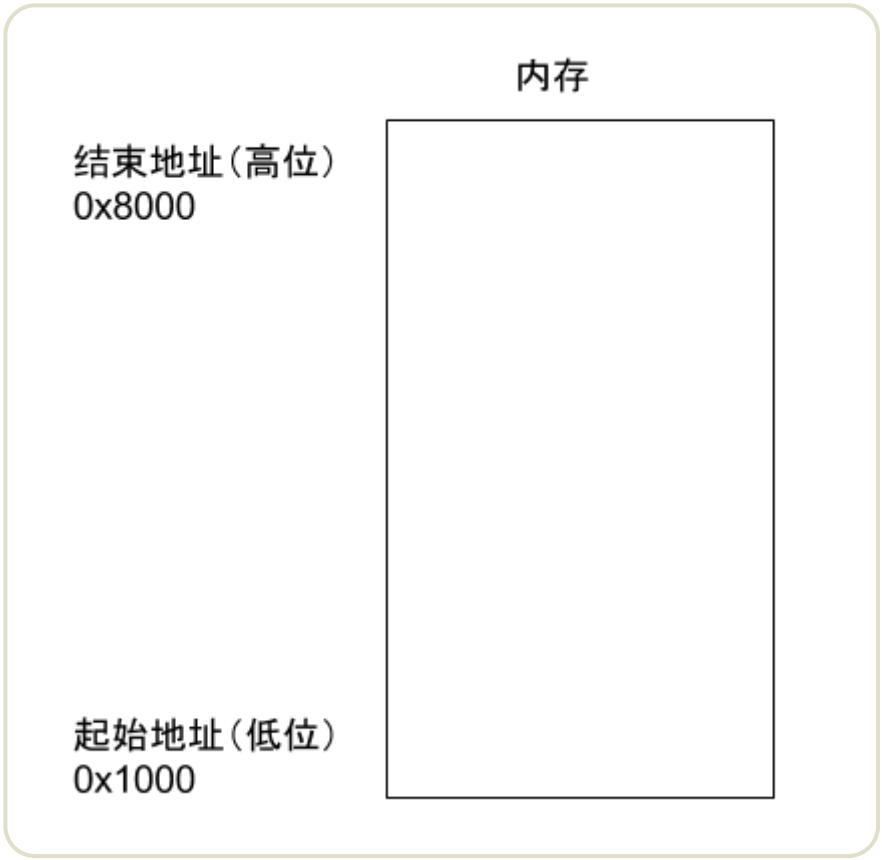


我们常常看到 32位 CPU、64位 CPU 这样的名称，其实指的就是寄存器的大小。32 位 CPU 的寄存器大小就是4个字节。

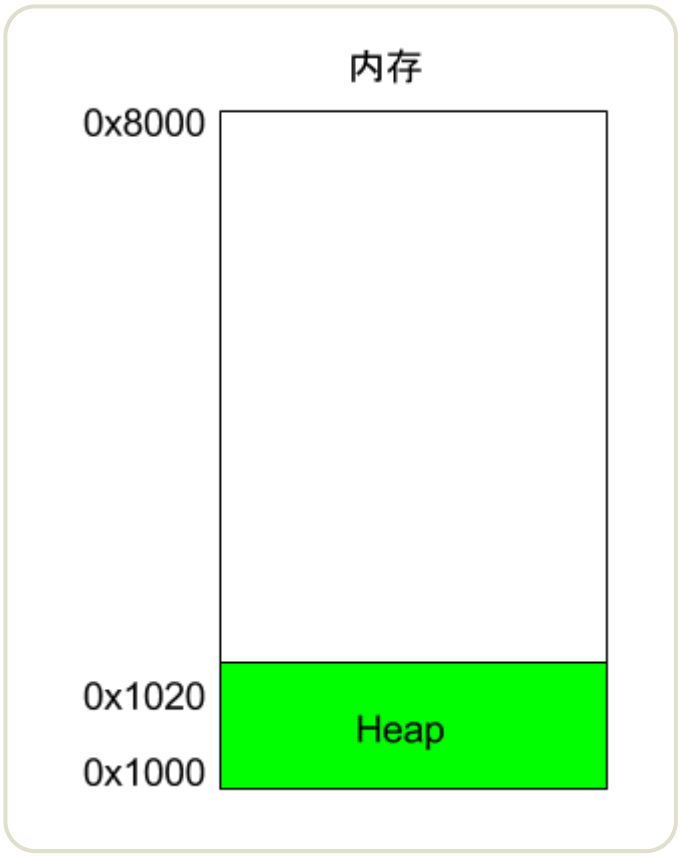
五、内存模型：Heap

寄存器只能存放很少量的数据，大多数时候，CPU 要指挥寄存器，直接跟内存交换数据。所以，除了寄存器，还必须了解内存怎么储存数据。

程序运行的时候，操作系统会给它分配一段内存，用来储存程序和运行产生的数据。这段内存有起始地址和结束地址，比如从 `0x1000` 到 `0x8000` ，起始地址是较小的那个地址，结束地址是较大的那个地址。



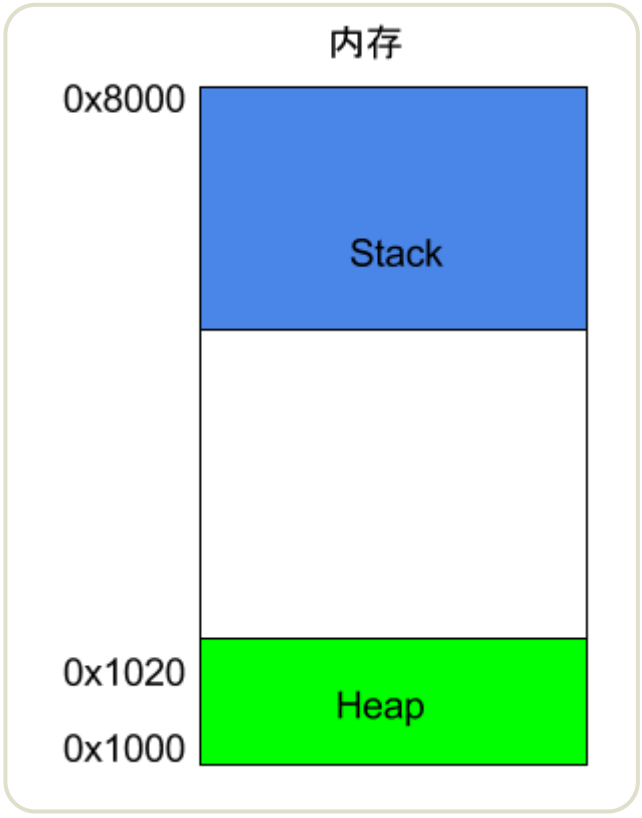
程序运行过程中，对于动态的内存占用请求（比如新建对象，或者使用 malloc 命令），系统就会从预先分配好的那段内存之中，划出一部分给用户，具体规则是从起始地址开始划分（实际上，起始地址会有一段静态数据，这里忽略）。举例来说，用户要求得到10个字节内存，那么从起始地址 0x1000 开始给他分配，一直分配到地址 0x100A ，如果再要求得到22个字节，那么就分配到 0x1020 。



这种因为用户主动请求而划分出来的内存区域，叫做 Heap（堆）。它由起始地址开始，从低位（地址）向高位（地址）增长。Heap 的一个重要特点就是不会自动消失，必须手动释放，或者由垃圾回收机制来回收。

六、内存模型：Stack

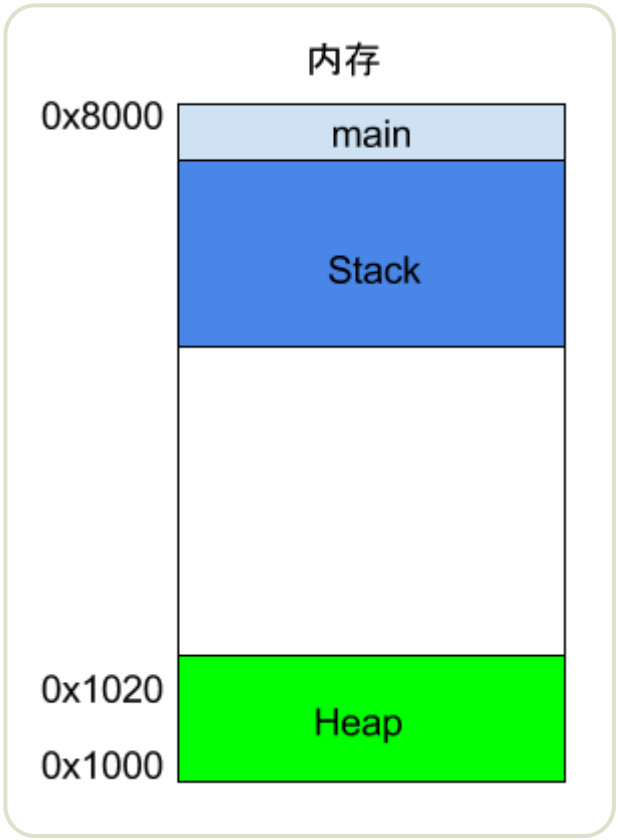
除了 Heap 以外，其他的内存占用叫做 Stack（栈）。简单说，Stack 是由于函数运行而临时占用的内存区域。



请看下面的例子。

```
int main() {
   int a = 2;
   int b = 3;
}
```

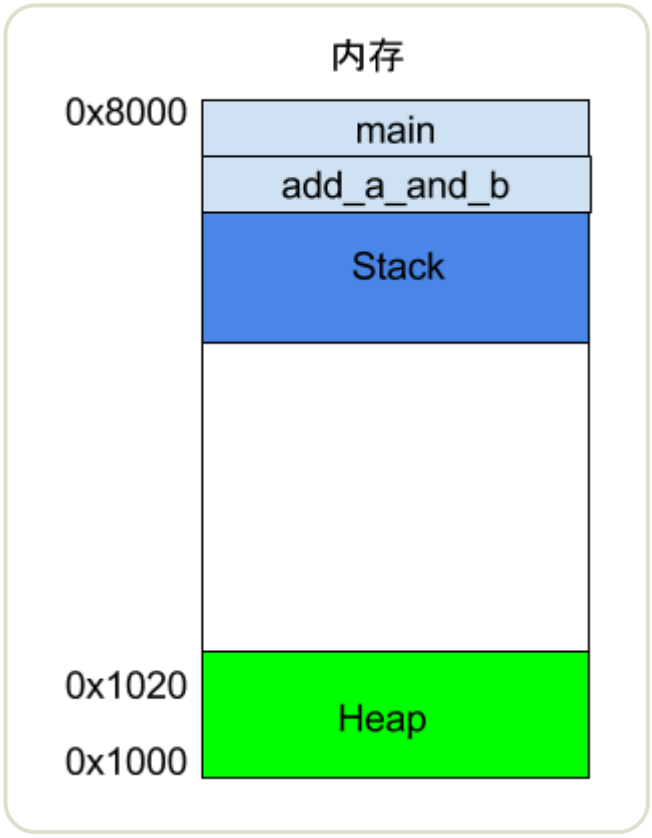
上面代码中，系统开始执行 main 函数时，会为它在内存里面建立一个帧（frame），所有 main 的内部变量（比如 a 和 b ）都保存在这个帧里面。 main 函数执行结束后，该帧就会被回收，释放所有的内部变量，不再占用空间。



如果函数内部调用了其他函数，会发生什么情况?

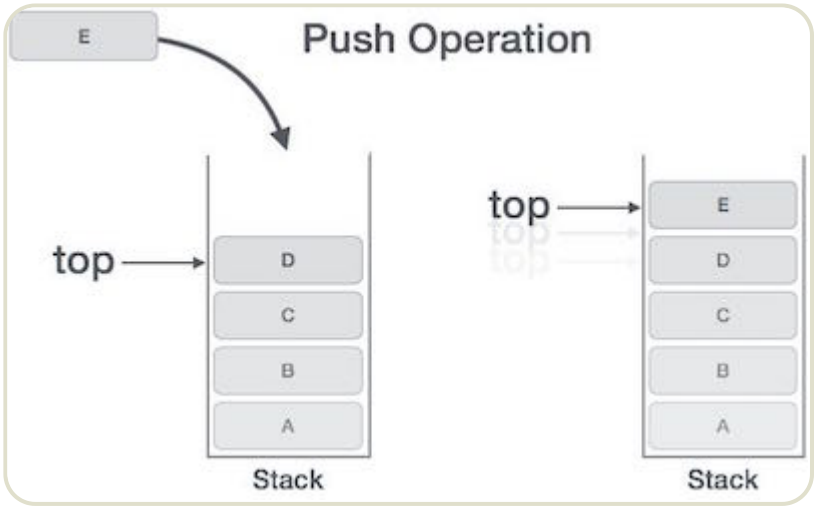
```
int main() {
   int a = 2;
   int b = 3;
   return add_a_and_b(a, b);
}
```

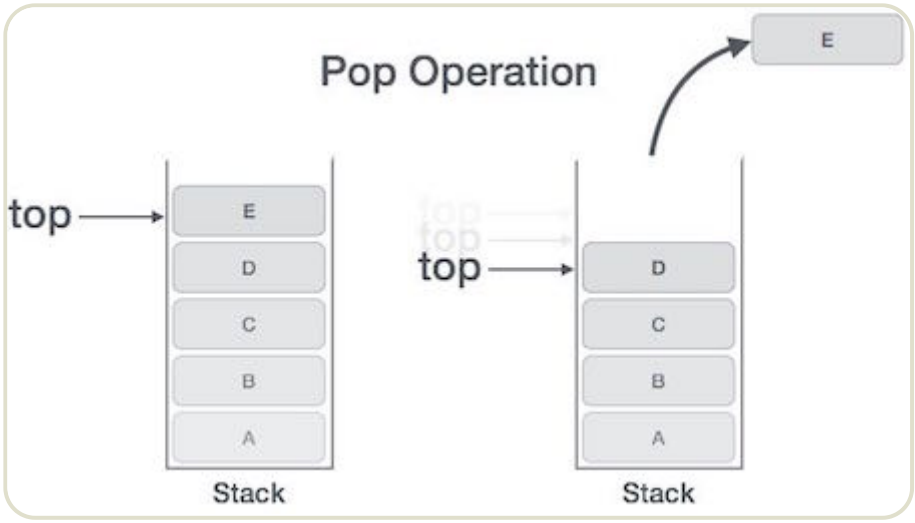
上面代码中， main 函数内部调用了 add_a_and_b 函数。执行到这一行的时候，系统也会为 add_a_and_b 新建一个帧，用来储存它的内部变量。也就是说，此时同时存在两个帧：main 和 add_a_and_b 。一般来说，调用栈有多少层，就有多少帧。



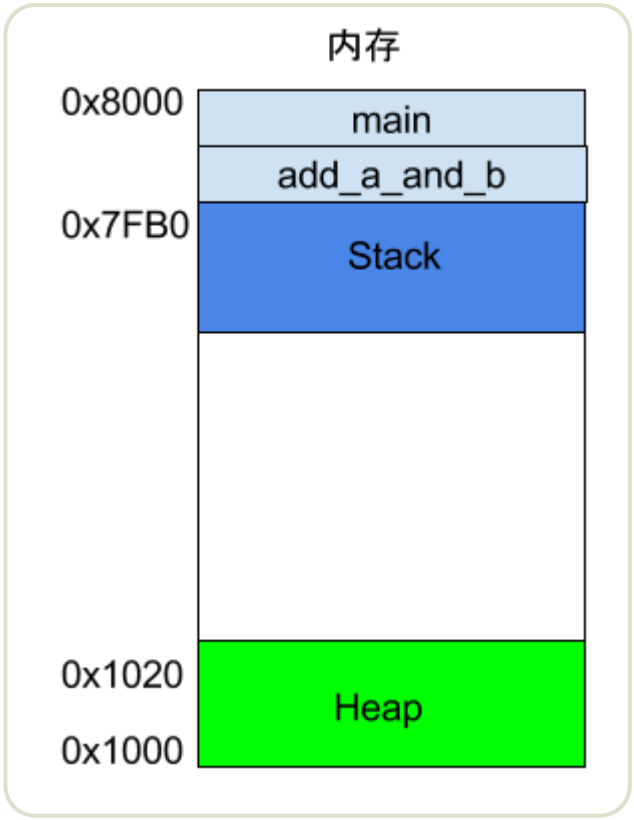
等到 add_a_and_b 运行结束，它的帧就会被回收，系统会回到函数 main 刚才中断执行的地方，继续往下执行。通过这种机制，就实现了函数的层层调用，并且每一层都能使用自己的本地变量。

所有的帧都存放在 Stack，由于帧是一层层叠加的，所以 Stack 叫做栈。生成新的帧，叫做"入栈"，英文是 push；栈的回收叫做"出栈"，英文是 pop。Stack 的特点就是，最晚入栈的帧最早出栈（因为最内层的函数调用，最先结束运行），这就叫做"后进先出"的数据结构。每一次函数执行结束，就自动释放一个帧，所有函数执行结束，整个 Stack 就都释放了。





Stack 是由内存区域的结束地址开始，从高位（地址）向低位（地址）分配。比如，内存区域的结束地址是 `0x8000` ，第一帧假定是16字节，那么下一次分配的地址就会从 `0x7FF0` 开始；第二帧假定需要64字节，那么地址就会移动到 `0x7FB0` 。



七、CPU 指令

7.1 一个实例

了解寄存器和内存模型以后，就可以来看汇编语言到底是什么了。下面是一个简单的程序 `example.c` 。

```
int add_a_and_b(int a, int b) {
   return a + b;
}

int main() {
   return add_a_and_b(2, 3);
}
```

gcc 将这个程序转成汇编语言。

```
$ gcc -S example.c
```

上面的命令执行以后，会生成一个文本文件 example.s ，里面就是汇编语言，包含了几十行指令。这么说吧，一个高级语言的简单操作，底层可能由几个，甚至几十个 CPU 指令构成。CPU 依次执行这些指令，完成这一步操作。

example.s 经过简化以后，大概是下面的样子。

```
_add_a_and_b:
   push   %ebx
   mov    %eax, [%esp+8]
   mov    %ebx, [%esp+12]
   add    %eax, %ebx
   pop    %ebx
   ret

_main:
   push   3
   push   2
   call   _add_a_and_b
   add    %esp, 8
   ret
```

可以看到，原程序的两个函数 add_a_and_b 和 main ，对应两个标签 _add_a_and_b 和 _main 。每个标签里面是该函数所转成的 CPU 运行流程。

每一行就是 CPU 执行的一次操作。它又分成两部分，就以其中一行为例。

```
push   %ebx
```

这一行里面， push 是 CPU 指令， %ebx 是该指令要用到的运算子。一个 CPU 指令可以有零个到多个运算子。

下面我就一行一行讲解这个汇编程序，建议读者最好把这个程序，在另一个窗口拷贝一份，省得阅读的时候再把页面滚动上来。

7.2 push 指令

根据约定，程序从 _main 标签开始执行，这时会在 Stack 上为 main 建立一个帧，并将 Stack 所指向的地址，写入 ESP 寄存器。后面如果有数据要写入 main 这个帧，就会写在 ESP 寄存器所保存的地址。

然后，开始执行第一行代码。

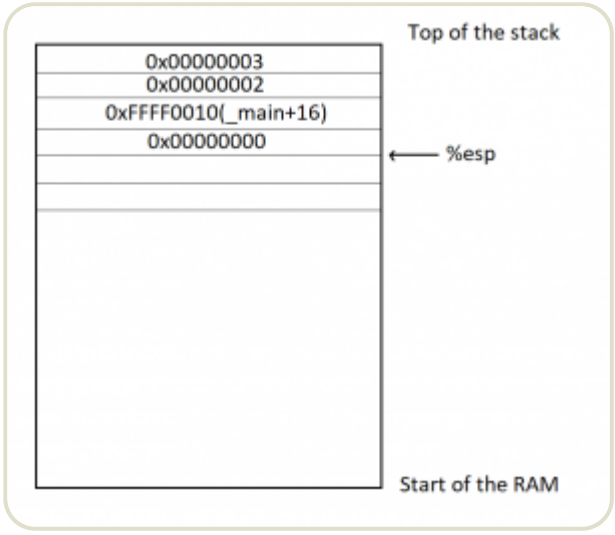
```
push   3
```

push 指令用于将运算子放入 Stack，这里就是将 3 写入 main 这个帧。

虽然看上去很简单， push 指令其实有一个前置操作。它会先取出 ESP 寄存器里面的地址，将其减去4个字节，然后将新地址写入 ESP 寄存器。使用减法是因为 Stack 从高位向低位发展，4个字节则是因为 3 的类型是 int ，占用4个字节。得到新地址以后， 3 就会写入这个地址开始的四个字节。

```
push   2
```

第二行也是一样， push 指令将 2 写入 main 这个帧，位置紧贴着前面写入的 3 。这时，ESP 寄存器会再减去 4个字节（累计减去8）。



7.3 call 指令

第三行的 call 指令用来调用函数。

```
call   _add_a_and_b
```

上面的代码表示调用 add_a_and_b 函数。这时，程序就会去找 _add_a_and_b 标签，并为该函数建立一个新的帧。

下面就开始执行 _add_a_and_b 的代码。

```
push   %ebx
```

这一行表示将 EBX 寄存器里面的值，写入 _add_a_and_b 这个帧。这是因为后面要用到这个寄存器，就先把里面的值取出来，用完后再写回去。

这时， push 指令会再将 ESP 寄存器里面的地址减去4个字节（累计减去12）。

7.4 mov 指令

mov 指令用于将一个值写入某个寄存器。

```
mov    %eax, [%esp+8]
```

这一行代码表示，先将 ESP 寄存器里面的地址加上8个字节，得到一个新的地址，然后按照这个地址在 Stack 取出数据。根据前面的步骤，可以推算出这里取出的是 2 ，再将 2 写入 EAX 寄存器。

下一行代码也是干同样的事情。

```
mov    %ebx, [%esp+12]
```

上面的代码将 ESP 寄存器的值加12个字节，再按照这个地址在 Stack 取出数据，这次取出的是 3 ，将其写入 EBX 寄存器。

7.5 add 指令

add 指令用于将两个运算子相加，并将结果写入第一个运算子。

```
add    %eax, %ebx
```

上面的代码将 EAX 寄存器的值（即2）加上 EBX 寄存器的值（即3），得到结果5，再将这个结果写入第一个运算子 EAX 寄存器。

7.6 pop 指令

pop 指令用于取出 Stack 最近一个写入的值（即最低位地址的值），并将这个值写入运算子指定的位置。

```
pop    %ebx
```

上面的代码表示，取出 Stack 最近写入的值（即 EBX 寄存器的原始值），再将这个值写回 EBX 寄存器（因为加法已经做完了，EBX 寄存器用不到了）。

注意， pop 指令还会将 ESP 寄存器里面的地址加4，即回收4个字节。

7.7 ret 指令

ret 指令用于终止当前函数的执行，将运行权交还给上层函数。也就是，当前函数的帧将被回收。

```
ret
```

可以看到，该指令没有运算子。

随着 add_a_and_b 函数终止执行，系统就回到刚才 main 函数中断的地方，继续往下执行。

```
add    %esp, 8
```

上面的代码表示，将 ESP 寄存器里面的地址，手动加上8个字节，再写回 ESP 寄存器。这是因为 ESP 寄存器的是 Stack 的写入开始地址，前面的 pop 操作已经回收了4个字节，这里再回收8个字节，等于全部回收。

```
ret
```

最后， main 函数运行结束， ret 指令退出程序执行。

八、参考链接

- Introduction to reverse engineering and Assembly, by Youness Alaoui
- x86 Assembly Guide, by University of Virginia Computer Science

（完）

文档信息

- 发表日期： 2018年1月21日

相关文章

- **2021.01.27:** 异或运算 XOR 教程

  > 大家比较熟悉的逻辑运算，主要是"与运算"（AND）和"或运算"（OR），还有一种"异或运算"（XOR），也非常重要。

- **2019.11.17:** 容错，高可用和灾备

  > 标题里面的三个术语，很容易混淆，专业人员有时也会用错。

- **2019.11.03:** 关于计算机科学的50个误解

  > 计算机科学（Computer Science，简称 CS）是大学的热门专业。但是，社会上对这个专业有很多误解，甚至本专业的学生也有误解。

- **2019.10.29:** 你所不知道的 AI 进展

  > 人工智能现在是常见词汇，大多数人可能觉得，它是学术话题，跟普通人关系不大。

Weibo | Twitter | GitHub

Email: yifeng.ruan@gmail.com