**程序的内存布局——函数调用栈的那点事**

原创 2015年05月22日 00:14:56

* 标签：
* [栈](http://so.csdn.net/so/search/s.do?q=栈&t=blog) /
* [操作系统](http://so.csdn.net/so/search/s.do?q=操作系统&t=blog) /
* [内存](http://so.csdn.net/so/search/s.do?q=内存&t=blog) /
* [malloc](http://so.csdn.net/so/search/s.do?q=malloc&t=blog) /
* [c语言](http://so.csdn.net/so/search/s.do?q=c语言&t=blog)

[注]此文是《程序员的自我修养》的读书总结，其中掺杂着一些个人的理解，若有不对，欢迎拍砖。

程序的内存布局

现代的应用程序都运行在一个虚拟内存空间里，在32位的系统里，这个内存空间拥有4GB的寻址能力。现代的应用程序可以直接使用32位的地址进行寻址，整个内存是一个统一的地址空间，用户可以使用一个32位的指针访问任意内存位置。  
【关于虚拟地址空间的介绍，看这里<http://blog.csdn.net/yang_yulei/article/details/24385573>】  
  
在进程的不同地址区间上有着不同的地位，Windows在默认情况下会将高地址的2GB空间分配给内核，而Linux默认将高地址的1GB空间分配给内核，具体的内存布局如下图：

(1)**代码区**：这个区域存储着被装入执行的二进制机器代码，处理器会到这个区域取指并执行。

(2)**数据区：**用于存储全局变量、常量。  
(3)**堆区**：进程可以在堆区动态地请求一定大小的内存，并在用完之后归还给堆区。动态分配和回收是堆区的特点。  
(4)**栈区**：用于动态地存储函数之间的关系，以保证被调用函数在返回时恢复到母函数中继续执行。

高级语言写出的程序经过编译链接，最终会变成可执行文件。当可执行文件被装载运行后，就成了所谓的进程。  
可执行文件代码段中包含的二进制级别的机器代码会被装入内存的代码区(.text)；  
处理器将到内存的这个区域一条一条地取出指令和操作数，并送入运算逻辑单元进行运算；  
如果代码中请求开辟动态内存，则会在内存的堆区分配一块大小合适的区域返回给代码区的代码使用；  
当函数调用发生时，函数的调用关系等信息会动态地保存在内存的栈区，以供处理器在执行完被调用函数的代码时，返回母函数。  
  
如果把计算机看成一个有条不紊的工厂，我们可以得到如下类比：  
 \* CPU是干活的工人。  
 \* 数据区、堆区、栈区等则是用来存放原料、半成品、成品等各种东西的场所。  
 \* 存放在代码区的指令则告诉CPU要做什么，怎么做，到哪里去领原材料，用什么工具来做，做完以后把成品放到哪个货仓去。

栈

在经典的操作系统里，栈总是**向下增长**的。栈顶由esp寄存器定位。压栈操作使栈顶的地址减小，弹出操作使栈顶地址增大。

**当函数调用的时候发生了什么？**

例如：

**[cpp]** [view plain](https://blog.csdn.net/yang_yulei/article/details/45795591) [copy](https://blog.csdn.net/yang_yulei/article/details/45795591)

1. int main(void)
2. {
3. foo(1,2,3) ;
4. return 0 ;
5. }

当方法main需要调用foo时，它的标准行为：

1、在main方法的调用栈中，将 foo的参数从右向左 依次push到栈中。  
2、把main方法当前指令的 下一条指令地址 （即return address）push到栈中。(隐藏在call指令中)  
3、使用call指令调用目标函数体foo。

请注意，以上3步都处于main的调用栈，其中ebp保存其栈底，而esp保存其栈顶。  
接下来，在**foo函数中**：

1、**push ebp**:                  将ebp的当前值push到栈中，即保存ebp。  
2、**mov ebp,esp**:            将esp的值赋给ebp，则意味着进入了foo方法的调用栈。  
3、**[可选]sub esp, XXX**: 在栈上分配XXX字节的临时空间。（抬高栈顶）(编译器根据函数中的局部变量的总大小确定临时空间的大小)  
4、**[可选]push XXX**:        保存（push）一些寄存器的值。

【注意：push寄存器的值，这一操作，可以在分配临时空间之前，也可在其之后，《程序员的自我修养》写的是在开辟临时变量之后】  
（编译器中保存的有相应的变量名对应的临时空间中的位置）  
  
而在foo方法调用完毕后，便执行前面阶段的逆操作：

1、**保存返回值:**          通常将函数的返回值保存在寄存器eax中。  
2、**[可选]恢复（pop）一些寄存器的值**。  
3、**mov esp,ebp**:     恢复esp同时回收局部变量空间。（恢复原栈顶）  
4、**pop ebp:**             将栈顶的值赋给ebp，即恢复main调用栈的栈底。（恢复原栈底）  
5、**ret**:                        从栈顶获得之前保留的return address，并跳转到此位置继续执行。

main方法先将foo方法所需的参数压入栈中，然后再改变ebp，进入foo方法的调用栈。

因此，如果在foo方法中需要访问那些参数，则需要根据当前ebp中的值，再向高地址偏移后进行访问——因为高地址才是main方法的调用栈。

也就是说，地址ebp + 8存放了foo方法的第1个参数，地址ebp + 12存放了foo方法的第2个参数，以此类推。那么地址ebp + 4存放了什么呢？它存放的是return address，即foo方法返回后，需要继续执行下去的main方法指令的地址。

【注意】  
若需在函数中保存被调函数保存寄存器(如ESI、EDI)，则编译器在保存EBP值时进行保存，或延迟保存直到局部变量空间被分配。在栈帧中并未为被调函数保存寄存器的空间指定标准的存储位置。  
  
【注：几个相关的寄存器（关于详细的介绍，见王爽汇编）】  
(1)**esp**：栈指针寄存器(extended stack pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的栈顶。  
(2)**ebp**：基址指针寄存器(extended base pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部。（ebp在当前栈帧内位置固定，故函数中对大部分数据的访问都基于ebp进行）  
(3)**eip**：指令寄存器(extended instruction pointer)，其内存放着一个指针，该指针永远指向下一条等待执行的指令地址。 可以说如果控制了EIP寄存器的内容，就控制了进程——我们让eip指向哪里，CPU就会去执行哪里的指令。eip可被jmp、call和ret等指令隐含地改变(事实上它一直都在改变)(ret指令就是把当前栈顶保存的返回值地址 弹到eip中)  
  
函数栈帧的大小并不固定，一般与其对应函数的局部变量多少有关。函数运行过程中，其栈帧大小也是在不停变化的。

调用惯例

函数的调用方和被调用方对于函数如何调用需要遵守同样的约定，函数才能被正确地调用，这样的约定称为\*\*调用惯例\*\*。

\* 函数参数的传递顺序和方式  
  调用惯例要规定参数压栈的顺序：是从左至右，还是从右至左。有些调用惯例还允许使用寄存器传递参数，以提高性能。  
\* 栈的维护方式  
  （谁负责弹出形参？）  
   在被调函数返回时，需要将被压入栈中的参数全部弹出，以使得栈在函数调用前后保持一致。这个弹出的工作可以由函数的调用方完成，也也可以由被函数完成。  
\* 名字修饰规则  
   为了链接的时候对调用惯例进行区分，调用惯例要对函数本身的名字进行修饰，不同的调用惯例有不同的名字修饰策略。

| **调用惯例** | **谁弹形参** | **参数压栈方向** | **名字修饰** |
| --- | --- | --- | --- |
| cdecl | 调用方 | 从右至左 | 下划线+函数名 |
| stdcall | 被调方 | 从右至左 | 下划线+函数名@参数字节数 |
| pascal | 被调方 | 从左至右 | 较复杂 |
| fastcall | 被调方 | 头两个参数放入寄存器，其它从右至左 | @函数名字名@参数字节数 |

**\_cdecl**

是CDeclaration的缩写，表示C语言默认的函数调用方法：所有参数从右到左依次入栈，这些参数**由调用者清除**，称为手动清栈。被调用函数无需要求调用者传递多少参数，调用者传递过多或者过少的参数，甚至完全不同的参数都不会产生编译阶段的错误。（典型的如printf函数）  
  
**\_stdcall**

是Standard Call的缩写，是C++的标准调用方式：所有参数从右到左依次入栈。这些堆栈中的参数**由被调用的函数在返回后清除**，使用的指令是 retn X，X表示参数占用的字节数，CPU在ret之后自动弹出X个字节的堆栈空间。称为自动清栈。函数在编译的时候就必须确定参数个数，并且调用者必须严格的控制参数的生成，不能多，不能少，否则返回后会出错。  
几乎我们写的每一个WINDOWS API函数都是\_stdcall类型的，因为不同的编译器产生栈的方式不尽相同，调用者不一定能正常的完成清除工作。如果使用\_stdcall，上面的问题就解决了，函数自己解决清除工作。所以，在跨平台的调用中，我们都使用\_stdcall（虽然有时是以WINAPI的样子出现）。  
但当我们遇到这样的函数如printf()它的参数是可变的，不定长的，被调用者事先无法知道参数的长度，事后的清除工作也无法正常的进行，因此，这种情况我们只能使用\\_cdecl。到这里我们有一个结论，如果你的程序中没有涉及可变参数，最好使用\_stdcall关键字。

函数返回值传递

一般情况下，寄存器eax是传递返回值的通道，函数将返回值存储在eax中，返回后函数的调用方再读取eax。

但是eax本身只有4字节，那么大于4字节的返回值是如何传递的呢？

对于返回5~8字节数据的情况，一般采用eax和edx联合返回的方式进行的。其中eax存储返回值的低4字节，edx存储返回值的高4字节。  
  
**对于超过8字节的返回类型：**

**[cpp]** [view plain](https://blog.csdn.net/yang_yulei/article/details/45795591) [copy](https://blog.csdn.net/yang_yulei/article/details/45795591)

1. typedef struct big\_thing
2. {
3. char buf[128] ;
4. } big\_thing ;
6. big\_thing return\_test();
8. //---------------------------
9. int main(void)
10. {
11. big\_thing n = return\_test() ;
12. }
14. big\_thing return\_test()
15. {
16. big\_thing b ;
17. b.buf[0] = 0 ;
18. return b ;
19. }

分析这段代码：  
首先，在主调函数main中，肯定有一个128字节的变量n，在被调函数return\_test中，肯定有一个128字节的变量b。  
那被调函数如何返回128字节的变量？直接从b拷贝到n么？你这样直接改变主调函数中变量的值，似乎不符合返回值传值的规则。  
  
那么实际上，编译器是怎么设计大尺寸返回值传递的呢？  
\* main函数在其栈中的局部变量区域中额外开辟一片空间，将其一部分作为传递返回值的临时对象temp。  
\* 将temp对象的地址作为隐藏参数传递给return\_test函数。  
\* return\_test函数将数据拷贝给temp对象，并**将temp对象的地址用eax传出**。  
\* return\_test返回后，main函数将eax指向的**temp对象的内容拷贝给n**。  
  
  
  
（return\_test是没有真正的参数的，只有一个“伪参数”由函数的调用方悄悄传入）  
  
【总结】  
函数返回值的传递：小于8字节的返回值，以寄存器为中转。大于8字节的，以主调函数中新开辟的同样大小的**中间变量temp**为中转。

C语言对于尺寸太大的返回值类型，会使用一个临时的栈上内存区域作为中转，结果返回值对象会被拷贝两次。故不到万不得已，不要轻易返回大尺寸对象。

C++函数的返回值传递

C++处理大返回值略有不同，其可能是像C那样，1次拷贝到栈上的临时对象里，然后把临时对象拷贝到存储返回值的对象里。  
但，有些编译器会进行返回值优化RVO(Return Value Optimization)，这样，对象拷贝会减少一次，即没有临时对象temp了，直接拷贝到主调函数的相应对象中。  
例如：

**[cpp]** [view plain](https://blog.csdn.net/yang_yulei/article/details/45795591) [copy](https://blog.csdn.net/yang_yulei/article/details/45795591)

1. #include <iostream>
2. using namespace std ;
4. struct cpp\_obj
5. {
6. cpp\_obj()
7. {
8. cout<< "ctor\n" ;
9. }
10. cpp\_obj(const cpp\_obj& c)
11. {
12. cout<< "copy ctor\n" ;
13. }
14. cpp\_obj& operator=(const cpp\_obj& rhs)
15. {
16. cout<< "operator=\n" ;
17. return \*this ;
18. }
19. ~cpp\_obj()
20. {
21. cout<< "dtor\n" ;
22. }
23. } ;
25. cpp\_obj foo()
26. {
27. cpp\_obj b ;
28. cout << "before foo return\n" ;
29. return b ;
30. }
32. int main()
33. {
34. cpp\_obj n ;
35. n = foo() ;
36. cout << "before main return\n" ;
37. return 0 ;
38. }
40. //---------运行结果---------
41. ctor
42. ctor
43. before foo return
44. operator=
45. dtor
46. before main return
47. dtor

此例子是在g++下编译运行。此例就没有设置一个临时变量temp，而是直接把被调函数局部变量的值直接拷贝到主调函数中去。

NRV

C++对于返回值还有一种更“激进”的优化策略——**NRV**（Named Return Value）具名返回值优化  
这种优化是甚至连**被调函数中的局部变量都不要了**！直接在主调函数中操作对象（根据隐藏参数传入的对象的引用）。

NRV优化简单的说：

主调函数中有一条语句，Cobj a = f（）；其中f（）是一个函数，函数里边申请了一个Cobj的对象b，然后把它返回。在对象返回的时候，一般情况下要调用拷贝函数，把函数f（）里边的局部对象b拷贝到函数外部的对象a。  
但是如果用了NRV优化，那就不必要调用拷贝构造函数，编译器可以这样做，把a的地址传递进函数f（），然后不让f（）申请要返回的对象b的空间，用a的地址来代替b的地址，这样当要返回对象b的时候，就不必要拷贝了，因为b就是a，省去了局部变量b，省去了拷贝的过程。

关于NRV要注意两点：（自己总结的，若有不对，请拍砖）

1、在被调函数foo中，其局部变量声明处即是调用主调函数main中对象的默认构造函数处。main中的对象定义处，只是开辟一个空间，当时并不调用构造函数。  
2、为何在主调函数中 CObj obj = foo() 会触发NRV优化  
     而分开写： CObj obj ; obj = foo() ; 没有NRV优化呢？

    因为：  
    程序员必须给class X定义拷贝构造函数才能触发NRV优化，不然还是按照最初的较慢的方式执行。（我们的第二种方式没有涉及到拷贝构造函数，故不会触发NRV优化）  
    但现在的编译器即使去掉类中的拷贝构造函数，也一样会有NRV优化，但必须是向在对象初始化时调用子函数才会有NRV。  
    (若没有NRV优化，则被调函数中会生成局部对象，但这个局部对象直接拷贝到主函数相应的对象中，也不会像C那样还要生成一个临时变量)  
  
若把上面的例子的调用方式改为： cpp\_obj n = foo() ;  
则会触发NRV优化，执行结果就是：

**[cpp]** [view plain](https://blog.csdn.net/yang_yulei/article/details/45795591) [copy](https://blog.csdn.net/yang_yulei/article/details/45795591)

1. //cpp\_obj n = foo() ;改为：
2. foo(n) ;
4. //foo实际就被改为：
5. void foo(cpp\_obj& \_\_result)
6. {
7. // 调用\_\_result的默认构造函数
8. \_\_result.cpp\_obj::cpp\_obj();
9. // 处理\_\_result
10. return;
11. }
13. //---------NRV后的运行结果---------
14. ctor
15. before foo return
16. before main return
17. dtor

（一定注意：只有CObj obj = foo();形式的调用才会有NRV优化！）

关于NRV优化详细见《深入理解C++对象模型》

堆

堆是一块巨大的内存空间，常常占据整个虚拟地址空间的绝大部分。在这片空间里，程序可以请求一块连续内存，并自由地使用，这块内存在程序主动放弃之前都会一直保持有效。在C语言中我们可以用malloc函数在堆上申请空间。  
  
malloc的实现：  
操作系统内核管理着进程的地址空间，它通过的有系统调用，若让malloc调用这个系统调用实现申请内存，可完成这个工作。  
但是，这样做性能较差，因为每次进行申请释放空间都需要进行系统调用，系统调用的开销比较大，会进行内核态和用户态的切换。  
比较好的做法是程序向操作系统申请一块适当大小的堆空间，然后由程序自己管理这块空间，管理着堆空间分配的往往是程序的运行库（一般是操作系统提供的共享库）。  
malloc实际上就是对这共享库中函数的包装。  
  
**"批发-零售"类比：**  
**运行库相当于是向操作系统批发了一块较大的堆空间，然后零售给程序用**。运行库在向程序零售空间时，必须管理此空间，不能把一块空间出售两次。  
当空间不够用时，运行库再向操作系统批发（调用OS相应的系统调用）。  
注意：这个运行库一般也是操作系统或语言提供给我们的，其包含了管理堆空间的算法，其运行在用户态下。  
（我们自己也可以实现这个分配算法，但常用的分配算法已经被各种系统、库实现了无数遍，没有必要重复发明轮子）  
每个进程在创建时都会有一个默认堆，这个堆在进程启动时创建，并且直到进程结束都一直存在。在Windows中默认堆大小为1MB。  
（注意：在Windows中堆不一定是向上增长的）  
  
问：malloc申请的空间是不是连续的？  
答：若“空间”指的是虚拟空间的话，那么答案是连续的，即每一次malloc分配后返回的空间都可以看做是一块连续的地址。（进程中可能存在多个堆，但一次能够分配的最大堆空间取决于最大的那个堆）  
如果空间值的是物理空间，则不一定连续，因为一块连续的虚拟地址空间有可能是若干个不连续的物理页拼凑成的。  
  
**堆空间管理算法**

\* 1、空闲链表法  
   把堆中各个空闲块按链表的方式连接起来，当用户请求时遍历链表找到合适的块。

\* 2、位图（这个思想好）  
   将整个堆划分为大量的大小相同的块。当用户请求时分配整数个空间给用户。我们可以用一个整数数组的位来记录分配状况。  
   （每个块只有头/使用/空闲三种状态，即用两个位就可表示一个块，因此称为位图。头是用来标记定界的作用）