



堆栈，堆栈，堆和栈的区别

Posted on 2006-01-21 16:23 任我行 阅读(143270) 评论(32) 编辑 收藏 引用 所属分类: C++ .

堆和栈的区别（转贴）

非本人作也!因非常经典,所以收归旗下,与众人阅之!原作者不祥!

堆和栈的区别

一、预备知识一程序的内存分配

一个由c/C++编译的程序占用的内存分为以下几个部分

- 1、栈区（stack）— 由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。
- 2、堆区（heap）— 一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。
- 3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。- 程序结束后有系统释放
- 4、文字常量区—常量字符串就是放在这里的。程序结束后由系统释放
- 5、程序代码区—存放函数体的二进制代码。

二、例子程序

这是一个前辈写的，非常详细

```
//main.cpp
int a = 0; 全局初始化区
char *p1; 全局未初始化区
main()
{
int b; 栈
char s[] = "abc"; 栈
char *p2; 栈
char *p3 = "123456"; 123456\0在常量区，p3在栈上。
static int c =0; 全局（静态）初始化区
p1 = (char *)malloc(10);
p2 = (char *)malloc(20);
分配得来10和20字节的区域就在堆区。
strcpy(p1, "123456"); 123456\0放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方。
}
```

二、堆和栈的理论知识

2.1申请方式

stack:

由系统自动分配。例如，声明在函数中一个局部变量 int b; 系统自动在栈中为b开辟空间

heap:

需要程序员自己申请，并指明大小，在c中malloc函数

如p1 = (char \*)malloc(10);

在C++中用new运算符

如p2 = (char \*)malloc(10);

但是注意p1、p2本身是在栈中的。

2.2

申请后系统的响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

2.3申请大小的限制

栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

2.4申请效率的比较：  
栈由系统自动分配，速度较快。但程序员是无法控制的。  
堆是由new分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便。  
另外，在WINDOWS下，最好的方式是用VirtualAlloc分配内存，他不是在堆，也不是在栈是直接 在进程的地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活。

2.5堆和栈中的存储内容  
栈： 在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。  
当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。  
堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

2.6存取效率的比较  
  
char s1[] = "aaaaaaaaaaaaaaaa";  
char \*s2 = "bbbbbbbbbbbbbbbbbb";  
aaaaaaaaaaaa是在运行时刻赋值的；  
而bbbbbbbbbbbb是在编译时就确定的；  
但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。

比如：  
#include  
void main()  
{  
char a = 1;  
char c[] = "1234567890";  
char \*p ="1234567890";  
a = c[1];  
a = p[1];  
return;  
}  
对应的汇编代码  
10: a = c[1];  
00401067 8A 4D F1 mov cl,byte ptr [ebp-0Fh]  
0040106A 88 4D FC mov byte ptr [ebp-4],cl  
11: a = p[1];  
0040106D 8B 55 EC mov edx,dword ptr [ebp-14h]  
00401070 8A 42 01 mov al,byte ptr [edx+1]  
00401073 88 45 FC mov byte ptr [ebp-4],al  
第一种在读取时直接就把字符串中的元素读到寄存器cl中，而第二种则要先把指针值读到edx中，在根据edx读取字符，显然慢了。

2.7小结：  
堆和栈的区别可以用如下的比喻来看出：  
使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。  
使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

## windows进程中的内存结构

在阅读本文之前，如果你连堆栈是什么多不知道的话，请先阅读文章后面的基础知识。

接触过编程的人都知道，高级语言都能通过变量名来访问内存中的数据。那么这些变量在内存中是如何存放的呢？程序又是如何使用这些变量的呢？下面就会对此进行深入的讨论。下文中的C语言代码如没有特别声明，默认都使用VC编译的release版。

首先，来了解一下 C 语言的变量是如何在内存分部的。C 语言有全局变量(Global)、本地变量(Local)，静态变量(Static)、寄存器变量(Regeister)。每种变量都有不同的分配方式。先来看下面这段代码：

```
#include <stdio.h>

int g1=0, g2=0, g3=0;

int main()
{
static int s1=0, s2=0, s3=0;
int v1=0, v2=0, v3=0;

//打印出各个变量的内存地址
```

```
printf("0x%08x\n",&v1); //打印各本地变量的内存地址
printf("0x%08x\n",&v2);
printf("0x%08x\n\n",&v3);
printf("0x%08x\n",&g1); //打印各全局变量的内存地址
printf("0x%08x\n",&g2);
printf("0x%08x\n\n",&g3);
printf("0x%08x\n",&s1); //打印各静态变量的内存地址
printf("0x%08x\n",&s2);
printf("0x%08x\n\n",&s3);
return 0;
}
```

编译后的执行结果是：

```
0x0012ff78
0x0012ff7c
0x0012ff80

0x004068d0
0x004068d4
0x004068d8

0x004068dc
0x004068e0
0x004068e4
```

输出的结果就是变量的内存地址。其中v1,v2,v3是本地变量，g1,g2,g3是全局变量，s1,s2,s3是静态变量。你可以看到这些变量在内存是连续分布的，但是本地变量和全局变量分配的内存地址差了十万八千里，而全局变量和静态变量分配的内存是连续的。这是因为本地变量和全局/静态变量是分配在不同类型的内存区域中的结果。对于一个进程的内存空间而言，可以在逻辑上分成3个部份：代码区，静态数据区和动态数据区。动态数据区一般就是“堆栈”。“栈(stack)”和“堆(heap)”是两种不同的动态数据区，栈是一种线性结构，堆是一种链式结构。进程的每个线程都有私有的“栈”，所以每个线程虽然代码一样，但本地变量的数据都是互不干扰。一个堆栈可以通过“基地址”和“栈顶”地址来描述。全局变量和静态变量分配在静态数据区，本地变量分配在动态数据区，即堆栈中。程序通过堆栈的基地址和偏移量来访问本地变量。



堆栈是一个先进后出的数据结构，栈顶地址总是小于等于栈的基地址。我们可以先了解一下函数调用的过程，以便对堆栈在程序中的作用有更深入的了解。不同的语言有不同的函数调用规定，这些因素有参数的压入规则和堆栈的平衡。windows API的调用规则和ANSI C的函数调用规则是不一样的，前者由被调函数调整堆栈，后者由调用者调整堆栈。两者通过“\_\_stdcall”和“\_\_cdecl”前缀区分。先看下面这段代码：

```
#include <stdio.h>

void __stdcall func(int param1,int param2,int param3)
{
int var1=param1;
int var2=param2;
int var3=param3;
printf("0x%08x\n",¶m1); //打印出各个变量的内存地址
printf("0x%08x\n",¶m2);
printf("0x%08x\n\n",¶m3);
printf("0x%08x\n",&var1);
printf("0x%08x\n",&var2);
printf("0x%08x\n\n",&var3);
return;
}

int main()
{
func(1,2,3);
```

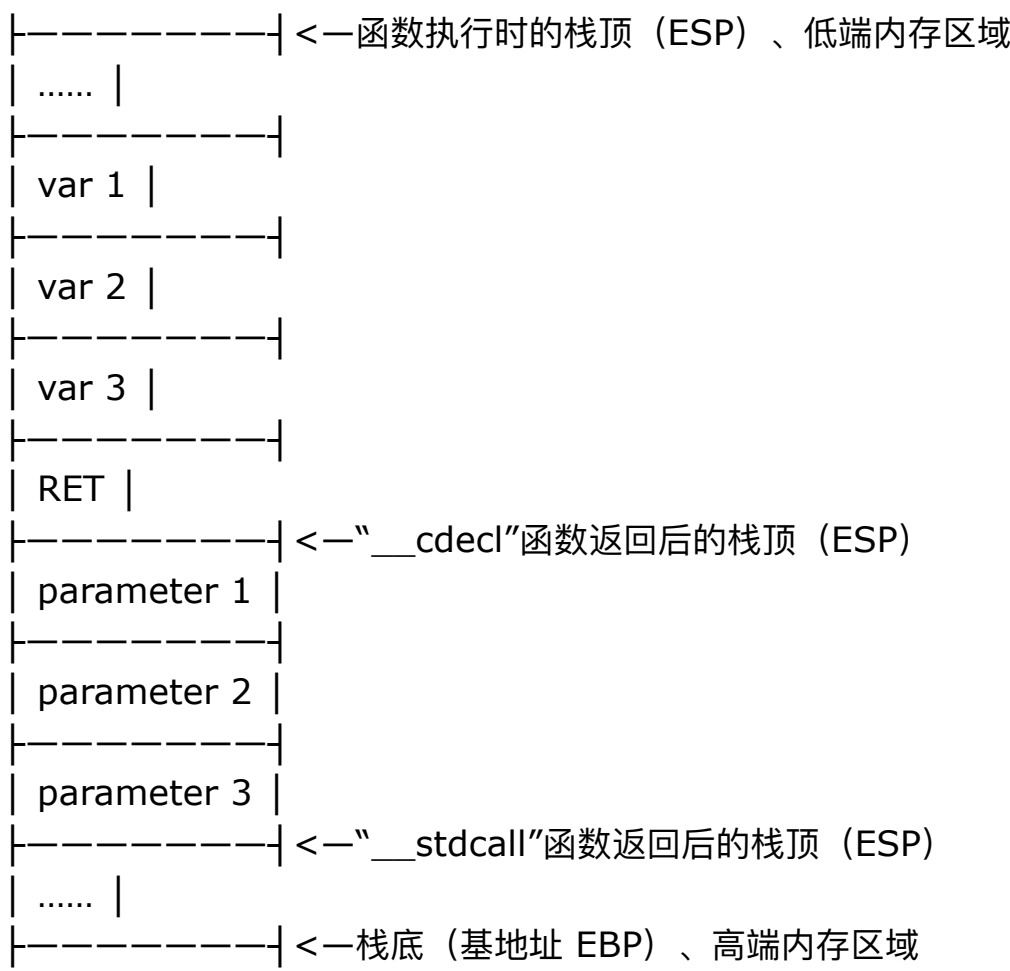


```
return 0;
}
```

编译后的执行结果是：

```
0x0012ff78
0x0012ff7c
0x0012ff80

0x0012ff68
0x0012ff6c
0x0012ff70
```



上图就是函数调用过程中堆栈的样子了。首先，三个参数以从右到左的次序压入堆栈，先压“param3”，再压“param2”，最后压入“param1”；然后压入函数的返回地址(RET)，接着跳转到函数地址接着执行（这里要补充一点，介绍UNIX下的缓冲溢出原理的文章中都提到在压入RET后，继续压入当前EBP，然后用当前ESP代替EBP。然而，有一篇介绍windows下函数调用的文章中说，在windows下的函数调用也有这一步骤，但根据我的实际调试，并未发现这一步，这还可以从param3和var1之间只有4字节的间隙这点看出来）；第三步，将栈顶(ESP)减去一个数，为本地变量分配内存空间，上例中是减去12字节(ESP=ESP-3\*4，每个int变量占用4个字节)；接着就初始化本地变量的内存空间。由于“\_\_stdcall”调用由被调函数调整堆栈，所以在函数返回前要恢复堆栈，先回收本地变量占用的内存(ESP=ESP+3\*4)，然后取出返回地址，填入EIP寄存器，回收先前压入参数占用的内存(ESP=ESP+3\*4)，继续执行调用者的代码。参见下列汇编代码：

```
;-----func 函数的汇编代码-----

:00401000 83EC0C sub esp, 0000000C //创建本地变量的内存空间
:00401003 8B442410 mov eax, dword ptr [esp+10]
:00401007 8B4C2414 mov ecx, dword ptr [esp+14]
:0040100B 8B542418 mov edx, dword ptr [esp+18]
:0040100F 89442400 mov dword ptr [esp], eax
:00401013 8D442410 lea eax, dword ptr [esp+10]
:00401017 894C2404 mov dword ptr [esp+04], ecx

..... (省略若干代码)

:00401075 83C43C add esp, 0000003C ;恢复堆栈，回收本地变量的内存空间
:00401078 C3 ret 000C ;函数返回，恢复参数占用的内存空间
;如果是“__cdecl”的话，这里是“ret”，堆栈将由调用者恢复

;-----函数结束-----

;-----主程序调用func函数的代码-----

:00401080 6A03 push 00000003 //压入参数param3
:00401082 6A02 push 00000002 //压入参数param2
:00401084 6A01 push 00000001 //压入参数param1
:00401086 E875FFFFFF call 00401000 //调用func函数
;如果是“__cdecl”的话，将在这里恢复堆栈，“add esp, 0000000C”
```

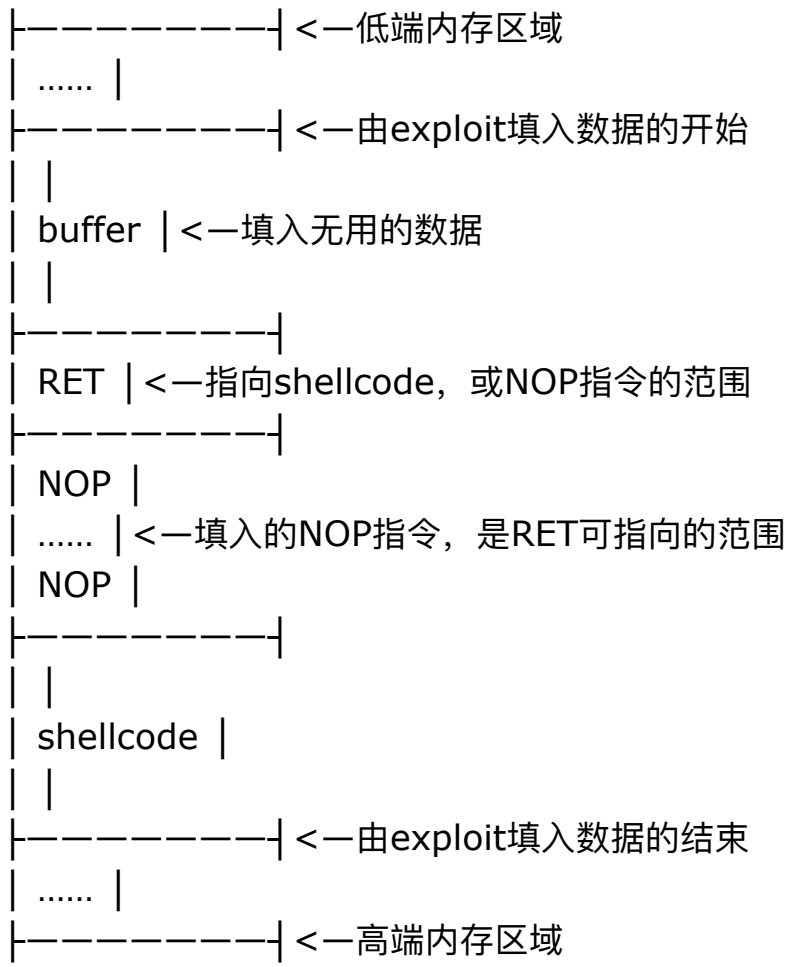
聪明的读者看到这里，差不多就明白缓冲溢出的原理了。先来看下面的代码：

```
#include <stdio.h>
#include <string.h>
```

```
void __stdcall func()
{
char lpBuff[8]="\0";
strcat(lpBuff,"AAAAAAAAAAAA");
return;
}

int main()
{
func();
return 0;
}
```

编译后执行一下回怎么样？哈，""0x00414141"指令引用的"0x00000000"内存。该内存不能为"read"。”，“非法操作”喽！"41"就是"A"的16进制的ASCII码了，那明显就是strcat这句出的问题了。"lpBuff"的大小只有8字节，算进结尾的\0，那strcat最多只能写入7个"A"，但程序实际写入了11个"A"外加1个\0。再来看看上面那幅图，多出来的4个字节正好覆盖了RET的所在的内存空间，导致函数返回到一个错误的内存地址，执行了错误的指令。如果能精心构造这个字符串，使它分成三部分，前一部份仅仅是填充的无意义数据以达到溢出的目的，接着是一个覆盖RET的数据，紧接着是一段shellcode，那只要着个RET地址能指向这段shellcode的第一个指令，那函数返回时就能执行shellcode了。但是软件的不同版本和不同的运行环境都可能影响这段shellcode在内存中的位置，那么要构造这个RET是十分困难的。一般都在RET和shellcode之间填充大量的NOP指令，使得exploit有更强的通用性。



windows下的动态数据除了可存放在栈中，还可以存放在堆中。了解C++的朋友都知道，C++可以使用new关键字来动态分配内存。来看下面的C++代码：

```
#include <stdio.h>
#include <iostream.h>
#include <windows.h>

void func()
{
char *buffer=new char[128];
char bufflocal[128];
static char buffstatic[128];
printf("0x%08x\n",buffer); //打印堆中变量的内存地址
printf("0x%08x\n",bufflocal); //打印本地变量的内存地址
printf("0x%08x\n",buffstatic); //打印静态变量的内存地址
}

void main()
{
func();
return;
}
```

程序执行结果为：

```
0x004107d0
0x0012ff04
0x004068c0
```

可以发现用new关键字分配的内存即不在栈中，也不在静态数据区。VC编译器是通过windows下的“堆(heap)”来实现new

关键字的内存动态分配。在讲“堆”之前，先来了解一下和“堆”有关的几个API函数：

HeapAlloc 在堆中申请内存空间  
HeapCreate 创建一个新的堆对象  
HeapDestroy 销毁一个堆对象  
HeapFree 释放申请的内存  
HeapWalk 枚举堆对象的所有内存块  
GetProcessHeap 取得进程的默认堆对象  
GetProcessHeaps 取得进程所有的堆对象  
LocalAlloc  
GlobalAlloc

当进程初始化时，系统会自动为进程创建一个默认堆，这个堆默认所占内存的大小为1M。堆对象由系统进行管理，它在内存中以链式结构存在。通过下面的代码可以通过堆动态申请内存空间：

```
HANDLE hHeap=GetProcessHeap();
char *buff=HeapAlloc(hHeap,0,8);
```

其中hHeap是堆对象的句柄，buff是指向申请的内存空间的地址。那这个hHeap究竟是什么呢？它的值有什么意义吗？看看下面这段代码吧：

```
#pragma comment(linker,"/entry:main") //定义程序的入口
#include <windows.h>

_CRTIMP int (__cdecl *printf)(const char *, ...); //定义STL函数printf
/*-----*/
写到这里，我们顺便来复习一下前面所讲的知识：
(*注)printf函数是C语言的标准函数库中函数，VC的标准函数库由msvcrt.dll模块实现。
由函数定义可见，printf的参数个数是可变的，函数内部无法预先知道调用者压入的参数个数，函数只能通过分析第一个参
数字符串的格式来获得压入参数的信息，由于这里参数的个数是动态的，所以必须由调用者来平衡堆栈，这里便使用了
__cdecl调用规则。BTW，Windows系统的API函数基本上是__stdcall调用形式，只有一个API例外，那就是wsprintf，
它使用__cdecl调用规则，同printf函数一样，这是由于它的参数个数是可变的缘故。
-----*/

void main()
{
HANDLE hHeap=GetProcessHeap();
char *buff=HeapAlloc(hHeap,0,0x10);
char *buff2=HeapAlloc(hHeap,0,0x10);
HMODULE hMsvcrt=LoadLibrary("msvcrt.dll");
printf=(void *)GetProcAddress(hMsvcrt,"printf");
printf("0x%08x\n",hHeap);
printf("0x%08x\n",buff);
printf("0x%08x\n\n",buff2);
}
```

执行结果为：

```
0x00130000
0x00133100
0x00133118
```

hHeap的值怎么和那个buff的值那么接近呢？其实hHeap这个句柄就是指向HEAP首部的地址。在进程的用户区存着一个叫PEB(进程环境块)的结构，这个结构中存放着一些有关进程的重要信息，其中在PEB首地址偏移0x18处存放的ProcessHeap就是进程默认堆的地址，而偏移0x90处存放了指向进程所有堆的地址列表的指针。windows有很多API都使用进程的默认堆来存放动态数据，如windows 2000下的所有ANSI版本的函数都是在默认堆中申请内存来转换ANSI字符串到Unicode字符串的。对一个堆的访问是顺序进行的，同一时刻只能有一个线程访问堆中的数据，当多个线程同时有访问要求时，只能排队等待，这样便造成程序执行效率下降。

最后来说说内存中的数据对齐。所位数据对齐，是指数据所在的内存地址必须是该数据长度的整数倍，DWORD数据的内存起始地址能被4除尽，WORD数据的内存起始地址能被2除尽，x86 CPU能直接访问对齐的数据，当他试图访问一个未对齐的数据时，会在内部进行一系列的调整，这些调整对于程序来说是透明的，但是会降低运行速度，所以编译器在编译程序时会尽量保证数据对齐。同样一段代码，我们来看看用VC、Dev-C++和Icc三个不同编译器编译出来的程序的执行结果：

```
#include <stdio.h>

int main()
{
int a;
char b;
int c;
printf("0x%08x\n",&a);
printf("0x%08x\n",&b);
printf("0x%08x\n",&c);
return 0;
}
```



这是用VC编译后的执行结果：

0x0012ff7c

0x0012ff7b

0x0012ff80

变量在内存中的顺序：b(1字节)-a(4字节)-c(4字节)。

这是用Dev-C++编译后的执行结果：

0x0022ff7c

0x0022ff7b

0x0022ff74

变量在内存中的顺序：c(4字节)-中间相隔3字节-b(占1字节)-a(4字节)。

这是用lcc编译后的执行结果：

0x0012ff6c

0x0012ff6b

0x0012ff64

变量在内存中的顺序：同上。

三个编译器都做到了数据对齐，但是后两个编译器显然没VC“聪明”，让一个char占了4字节，浪费内存哦。

基础知识：

堆栈是一种简单的数据结构，是一种只允许在其一端进行插入或删除的线性表。允许插入或删除操作的一端称为栈顶，另一端称为栈底，对堆栈的插入和删除操作被称为入栈和出栈。有一组CPU指令可以实现对进程的内存实现堆栈访问。其中，POP指令实现出栈操作，PUSH指令实现入栈操作。CPU的ESP寄存器存放当前线程的栈顶指针，EBP寄存器中保存当前线程的栈底指针。CPU的EIP寄存器存放下一个CPU指令存放的内存地址，当CPU执行完当前的指令后，从EIP寄存器中读取下一条指令的内存地址，然后继续执行。

参考：《Windows下的HEAP溢出及其利用》by: isno

《windows核心编程》by: Jeffrey Richter

摘要： 讨论常见的堆性能问题以及如何防范它们。（共 9 页）

前言

您是否是动态分配的 C/C++ 对象忠实且幸运的用户？您是否在模块间的往返通信中频繁地使用了“自动化”？您的程序是否因堆分配而运行起来很慢？不仅仅您遇到这样的问题。几乎所有项目迟早都会遇到堆问题。大家都想说，“我的代码真正好，只是堆太慢”。那只是部分正确。更深入理解堆及其用法、以及会发生什么问题，是很有用的。

什么是堆？

（如果您已经知道什么是堆，可以跳到“什么是常见的堆性能问题？”部分）

在程序中，使用堆来动态分配和释放对象。在下列情况下，调用堆操作：

事先不知道程序所需对象的数量和大小。

对象太大而不适合堆栈分配程序。

堆使用了在运行时分配给代码和堆栈的内存之外的部分内存。下图给出了堆分配程序的不同层。



GlobalAlloc/GlobalFree：Microsoft Win32 堆调用，这些调用直接与每个进程的默认堆进行对话。

LocalAlloc/LocalFree：Win32 堆调用（为了与 Microsoft Windows NT 兼容），这些调用直接与每个进程的默认堆进行对话。

COM 的 IMalloc 分配程序（或 CoTaskMemAlloc / CoTaskMemFree）：函数使用每个进程的默认堆。自动化程序使用“组件对象模型 (COM)”的分配程序，而申请的程序使用每个进程堆。

C/C++ 运行时 (CRT) 分配程序：提供了 malloc() 和 free() 以及 new 和 delete 操作符。如 Microsoft Visual Basic 和 Java 等语言也提供了新的操作符并使用垃圾收集来代替堆。CRT 创建自己的私有堆，驻留在 Win32 堆的顶部。

Windows NT 中，Win32 堆是 Windows NT 运行时分配程序周围的薄层。所有 API 转发它们的请求给 NTDLL。

Windows NT 运行时分配程序提供 Windows NT 内的核心堆分配程序。它由具有 128 个大小从 8 到 1,024 字节的空闲列表的前端分配程序组成。后端分配程序使用虚拟内存来保留和提交页。

在图表的底部是“虚拟内存分配程序”，操作系统使用它来保留和提交页。所有分配程序使用虚拟内存进行数据的存取。

分配和释放块不就那么简单吗？为何花费这么长时间？

堆实现的注意事项

传统上，操作系统和运行时库是与堆的实现共存的。在一个进程的开始，操作系统创建一个默认堆，叫做“进程堆”。如果没

有其他堆可使用，则块的分配使用“进程堆”。语言运行时也能在进程内创建单独的堆。（例如，C 运行时创建它自己的堆。）除这些专用的堆外，应用程序或许多已载入的动态链接库（DLL）之一可以创建和使用单独的堆。Win32 提供一整套 API 来创建和使用私有堆。有关堆函数（英文）的详尽指导，请参见 MSDN。

当应用程序或 DLL 创建私有堆时，这些堆存在于进程空间，并且在进程内是可访问的。从给定堆分配的数据将在同一个堆上释放。（不能从一个堆分配而在另一个堆释放。）

在所有虚拟内存系统中，堆驻留在操作系统的“虚拟内存管理器”的顶部。语言运行时堆也驻留在虚拟内存顶部。某些情况下，这些堆是操作系统堆中的层，而语言运行时堆则通过大块的分配来执行自己的内存管理。不使用操作系统堆，而使用虚拟内存函数更利于堆的分配和块的使用。

典型的堆实现由前、后端分配程序组成。前端分配程序维持固定大小块的空闲列表。对于一次分配调用，堆尝试从前端列表找到一个自由块。如果失败，堆被迫从后端（保留和提交虚拟内存）分配一个大块来满足请求。通用的实现有每块分配的开销，这将耗费执行周期，也减少了可使用的存储空间。

Knowledge Base 文章 Q10758，“用 calloc() 和 malloc() 管理内存”（搜索文章编号），包含了有关这些主题的更多背景知识。另外，有关堆实现和设计的详细讨论也可在下列著作中找到：“Dynamic Storage Allocation: A Survey and Critical Review”，作者 Paul R. Wilson、Mark S. Johnstone、Michael Neely 和 David Boles；“International Workshop on Memory Management”，作者 Kinross, Scotland, UK, 1995 年 9 月(<http://www.cs.utexas.edu/users/oops/papers.html>)（英文）。

Windows NT 的实现（Windows NT 版本 4.0 和更新版本）使用了 127 个大小从 8 到 1,024 字节的 8 字节对齐块空闲列表和一个“大块”列表。“大块”列表（空闲列表[0]）保存大于 1,024 字节的块。空闲列表容纳了用双向链表链接在一起的对象。默认情况下，“进程堆”执行收集操作。（收集是将相邻空闲块合并成一个大块的操作。）收集耗费了额外的周期，但减少了堆块的内部碎片。

单一全局锁保护堆，防止多线程式的使用。（请参见“Server Performance and Scalability Killers”中的第一个注意事项，George Reilly 所著，在“MSDN Online Web Workshop”上（站点：<http://msdn.microsoft.com/workshop/server/iis/tencom.asp>（英文）。）单一全局锁本质上是用来保护堆数据结构，防止跨多线程的随机存取。若堆操作太频繁，单一全局锁会对性能有不利的影

什么是常见的堆性能问题？  
以下是您使用堆时会遇到的最常见问题：

分配操作造成的速度减慢。光分配就耗费很长时间。最可能导致运行速度减慢原因是空闲列表没有块，所以运行时分配程序代码会耗费周期寻找较大的空闲块，或从后端分配程序分配新块。

释放操作造成的速度减慢。释放操作耗费较多周期，主要是启用了收集操作。收集期间，每个释放操作“查找”它的相邻块，取出它们并构造成较大块，然后再把此较大块插入空闲列表。在查找期间，内存可能会随机碰到，从而导致高速缓存不能命中，性能降低。

堆竞争造成的速度减慢。当两个或多个线程同时访问数据，而且一个线程继续进行之前必须等待另一个线程完成时就发生竞争。竞争总是导致麻烦；这也是目前多处理器系统遇到的最大问题。当大量使用内存块的应用程序或 DLL 以多线程方式运行（或运行于多处理器系统上）时将导致速度减慢。单一锁定的使用一常用的解决方案一意味着使用堆的所有操作是序列化的。当等待锁定时序列化会引起线程切换上下文。可以想象交叉路口闪烁的红灯处走走停停导致的速度减慢。竞争通常会导致线程和进程的上下文切换。上下文切换的开销是很大的，但开销更大的是数据从处理器高速缓存中丢失，以及后来线程复活时的数据重建。

堆破坏造成的速度减慢。造成堆破坏的原因是应用程序对堆块的不正确使用。通常情形包括释放已释放的堆块或使用已释放的堆块，以及块的越界重写等明显问题。（破坏不在本文讨论范围之内。有关内存重写和泄漏等其他细节，请参见 Microsoft Visual C++(R) 调试文档。）

频繁的分配和重分配造成的速度减慢。这是使用脚本语言时非常普遍的现象。如字符串被反复分配，随重分配增长和释放。不要这样做，如果可能，尽量分配大字符串和使用缓冲区。另一种方法就是尽量少用连接操作。竞争是在分配和释放操作中导致速度减慢的问题。理想情况下，希望使用没有竞争和快速分配/释放的堆。可惜，现在还没有这样的通用堆，也许将来会有。

在所有的服务器系统中（如 IIS、MSProxy、DatabaseStacks、网络服务器、Exchange 和其他），堆锁定实在是个大瓶颈。处理器数越多，竞争就越会恶化。

尽量减少堆的使用  
现在您明白使用堆时存在的问题了，难道您不想拥有能解决这些问题的超级魔棒吗？我可希望有。但没有魔法能使堆运行加快一因此不要期望在产品出货之前的最后一星期能够大为改观。如果提前规划堆策略，情况将会大大好转。调整使用堆的方法，减少对堆的操作是提高性能的良方。

如何减少使用堆操作？通过利用数据结构内的位置可减少堆操作的次数。请考虑下列实例：

```
struct ObjectA {  
    // objectA 的数据  
}
```

```
struct ObjectB {  
    // objectB 的数据
```



```
}

// 同时使用 objectA 和 objectB

//
// 使用指针
//
struct ObjectB {
    struct ObjectA * pObjA;
    // objectB 的数据
}

//
// 使用嵌入
//
struct ObjectB {
    struct ObjectA pObjA;
    // objectB 的数据
}

//
// 集合 – 在另一对象内使用 objectA 和 objectB
//

struct ObjectX {
    struct ObjectA  objA;
    struct ObjectB  objB;
}
```

避免使用指针关联两个数据结构。如果使用指针关联两个数据结构，前面实例中的对象 A 和 B 将被分别分配和释放。这会  
增加额外开销—我们要避免这种做法。

把带指针的子对象嵌入父对象。当对象中有指针时，则意味着对象中有动态元素（百分之八十）和没有引用的新位置。嵌入  
增加了位置从而减少了进一步分配/释放的需求。这将提高应用程序的性能。

合并小对象形成大对象（聚合）。聚合减少分配和释放的块的数量。如果有几个开发者，各自开发设计的不同部分，则最终  
会有许多小对象需要合并。集成的挑战就是要找到正确的聚合边界。

内联缓冲区能够满足百分之八十的需要（aka 80-20 规则）。个别情况下，需要内存缓冲区来保存字符串/二进制数据，但  
事先不知道总字节数。估计并内联一个大小能满足百分之八十需要的缓冲区。对剩余的百分之二十，可以分配一个新的缓冲  
区和指向这个缓冲区的指针。这样，就减少分配和释放调用并增加数据的位置空间，从根本上提高代码的性能。

在块中分配对象（块化）。块化是以组的方式一次分配多个对象的方法。如果对列表的项连续跟踪，例如对一个 {名称，  
值} 对的列表，有两种选择：选择一是为每一个“名称-值”对分配一个节点；选择二是分配一个能容纳（如五个）“名称-  
值”对的结构。例如，一般情况下，如果存储四对，就可减少节点的数量，如果需要额外的空间数量，则使用附加的链表指  
针。  
块化是友好的处理器高速缓存，特别是对于 L1-高速缓存，因为它提供了增加的位置 —不用说对于块分配，很多数据块会在  
同一个虚拟页中。

正确使用 \_amblksiz。C 运行时 (CRT) 有它的自定义前端分配程序，该分配程序从后端（Win32 堆）分配大小  
为 \_amblksiz 的块。将 \_amblksiz 设置为较高的值能潜在地减少对后端的调用次数。这只对广泛使用 CRT 的程序适用。  
使用上述技术将获得的好处会因对象类型、大小及工作量而有所不同。但总能在性能和可伸缩性方面有所收获。另一方面，  
代码会有点特殊，但如果经过深思熟虑，代码还是很容易管理的。

其他提高性能的技术  
下面是一些提高速度的技术：

使用 Windows NT5 堆  
由于几个同事的努力和辛勤工作，1998 年初 Microsoft Windows(R) 2000 中有了几个重大改进：

改进了堆代码内的锁定。堆代码对每堆一个锁。全局锁保护堆数据结构，防止多线程式的使用。但不幸的是，在高通信量的  
情况下，堆仍受困于全局锁，导致高竞争和低性能。Windows 2000 中，锁内代码的临界区将竞争的可能性减到最小,从而  
提高了可伸缩性。

使用 “Lookaside”列表。堆数据结构对块的所有空闲项使用了大小在 8 到 1,024 字节（以 8-字节递增）的快速高速缓  
存。快速高速缓存最初保护在全局锁内。现在，使用 lookaside 列表来访问这些快速高速缓存空闲列表。这些列表不要求锁  
定，而是使用 64 位的互锁操作，因此提高了性能。

内部数据结构算法也得到改进。  
这些改进避免了对分配高速缓存的需求，但不排除其他的优化。使用 Windows NT5 堆评估您的代码；它对小于 1,024 字

节(1 KB) 的块（来自前端分配程序的块）是最佳的。GlobalAlloc() 和 LocalAlloc() 建立在同一堆上，是存取每个进程堆的通用机制。如果希望获得高的局部性能，则使用 Heap(R) API 来存取每个进程堆，或为分配操作创建自己的堆。如果需要大块操作，也可以直接使用 VirtualAlloc() / VirtualFree() 操作。

上述改进已在 Windows 2000 beta 2 和 Windows NT 4.0 SP4 中使用。改进后，堆锁的竞争率显著降低。这使所有 Win32 堆的直接用户受益。CRT 堆建立于 Win32 堆的顶部，但它使用自己的小块堆，因而不能从 Windows NT 改进中受益。（Visual C++ 版本 6.0 也有改进的堆分配程序。）

使用分配高速缓存

分配高速缓存允许高速缓存分配的块，以便将来重用。这能够减少对进程堆（或全局堆）的分配/释放调用的次数，也允许最大限度的重用曾经分配的块。另外，分配高速缓存允许收集统计信息,以便较好地理解对象在较高层次上的使用。

典型地，自定义堆分配程序在进程堆的顶部实现。自定义堆分配程序与系统堆的行为很相似。主要的差别是它在进程堆的顶部为分配的对象提供高速缓存。高速缓存设计成一套固定大小（如 32 字节、64 字节、128 字节等）。这一个很好的策略，但这种自定义堆分配程序丢失与分配和释放的对象相关的“语义信息”。

与自定义堆分配程序相反，“分配高速缓存”作为每类分配高速缓存来实现。除能够提供自定义堆分配程序的所有好处之外，它们还能够保留大量语义信息。每个分配高速缓存处理程序与一个目标二进制对象关联。它能够使用一套参数进行初始化，这些参数表示并发级别、对象大小和保持在空闲列表中的元素的数量等。分配高速缓存处理程序对象维持自己的私有空闲实体池（不超过指定的阈值）并使用私有保护锁。合在一起，分配高速缓存和私有锁减少了与主系统堆的通信量，因而提供了增加的并发、最大限度的重用和较高的可伸缩性。

需要使用清理程序来定期检查所有分配高速缓存处理程序的活动情况并回收未用的资源。如果发现没有活动，将释放分配对象的池，从而提高性能。

可以审核每个分配/释放活动。第一级信息包括对象、分配和释放调用的总数。通过查看它们的统计信息可以得出各个对象之间的语义关系。利用以上介绍的许多技术之一，这种关系可以用来减少内存分配。

分配高速缓存也起到了调试助手的作用，帮助您跟踪没有完全清除的对象数量。通过查看动态堆栈返回踪迹和除没有清除的对象之外的签名，甚至能够找到确切的失败的调用者。

MP 堆

MP 堆是对多处理器友好的分布式分配的程序包，在 Win32 SDK（Windows NT 4.0 和更新版本）中可以得到。最初由 JVert 实现，此处堆抽象建立在 Win32 堆程序包的顶部。MP 堆创建多个 Win32 堆，并试图将分配调用分布到不同堆，以减少在所有单一锁上的竞争。

本程序包是好的步骤——一种改进的 MP-友好的自定义堆分配程序。但是，它不提供语义信息和缺乏统计功能。通常将 MP 堆作为 SDK 库来使用。如果使用这个 SDK 创建可重用组件，您将大大受益。但是，如果在每个 DLL 中建立这个 SDK 库，将增加工作设置。

重新思考算法和数据结构

要在多处理器机器上伸缩，则算法、实现、数据结构和硬件必须动态伸缩。请看最经常分配和释放的数据结构。试问，“我能用不同的数据结构完成此工作吗？”例如，如果在应用程序初始化时加载了只读项的列表，这个列表不必是线性链接的列表。如果是动态分配的数组就非常好。动态分配的数组将减少内存中的堆块和碎片，从而增强性能。

减少需要的小对象的数量减少堆分配程序的负载。例如，我们在服务器的关键处理路径上使用五个不同的对象，每个对象单独分配和释放。一起高速缓存这些对象，把堆调用从五个减少到一个，显著减少了堆的负载，特别当每秒钟处理 1,000 个以上的请求时。

如果大量使用“Automation”结构，请考虑从主线代码中删除“Automation BSTR”，或至少避免重复的 BSTR 操作。（BSTR 连接导致过多的重分配和分配/释放操作。）

摘要

对所有平台往往都存在堆实现，因此有巨大的开销。每个单独代码都有特定的要求，但设计能采用本文讨论的基本理论来减少堆之间的相互作用。

评价您的代码中堆的使用。

改进您的代码，以使用较少的堆调用：分析关键路径和固定数据结构。

在实现自定义的包装程序之前使用量化堆调用成本的方法。

如果对性能不满意，请要求 OS 组改进堆。更多这类请求意味着对改进堆的更多关注。

要求 C 运行时组针对 OS 所提供的堆制作小巧的分配包装程序。随着 OS 堆的改进，C 运行时堆调用的成本将减小。

操作系统（Windows NT 家族）正在不断改进堆。请随时关注和利用这些改进。

Murali Krishnan 是 Internet Information Server (IIS) 组的首席软件设计工程师。从 1.0 版本开始他就设计 IIS，并成功发行了 1.0 版本到 4.0 版本。Murali 组织并领导 IIS 性能组三年 (1995-1998), 从一开始就影响 IIS 性能。他拥有威斯康星州 Madison 大学的 M.S.和印度 Anna 大学的 B.S.。工作之外，他喜欢阅读、打排球和家庭烹饪。

 [http://community.csdn.net/Expert/FAQ/FAQ\\_Index.asp?id=172835](http://community.csdn.net/Expert/FAQ/FAQ_Index.asp?id=172835)

我在学习对象的生存方式的时候见到一种是在堆栈(stack)之中，如下

```
CObject object;
还有一种是在堆(heap)中 如下
CObject* pobject=new CObject();
```

请问

- (1) 这两种方式有什么区别？
- (2) 堆栈与堆有什么区别？

- 
- 1) about stack, system will allocate memory to the instance of object automatically, and to the heap, you must allocate memory to the instance of object with new or malloc manually.
  - 2) when function ends, system will automatically free the memory area of stack, but to the heap, you must free the memory area manually with free or delete, else it will result in memory leak.
  - 3)栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
  - 4)堆上分配的内存可以有我们自己决定，使用非常灵活。
- 

Feedback

# re: 堆栈，堆栈，堆和栈的区别 回复 更多评论  
2006-07-25 18:04 by Teresa

求教：  
问个问题，堆栈大小的限制是受寄存器地址为数的限制么？  
如果不是，你知道是受什么的限制么？  
  
谢谢啦先！

# re: 堆栈，堆栈，堆和栈的区别 回复 更多评论  
2006-07-28 13:22 by 任我行

栈：是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。  
堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。  
应用文章中的话，在编译时，可以用/STACK命令来设置栈的大小，VC中默认时2M，不过也有一个限制，堆的最大空间和机器位数有关，32机上是4G吧。

# re: 堆栈，堆栈，堆和栈的区别 回复 更多评论  
2006-08-10 16:00 by relearn\_C

非常好的文章，感谢您的分享。赞一个！

# re: 堆栈，堆栈，堆和栈的区别 回复 更多评论  
2006-08-14 15:29 by jitongwang

版主很强  
多谢几篇啊  
顶一下

# re: 堆栈，堆栈，堆和栈的区别 回复 更多评论  
2006-10-11 06:02 by vic

楼主加我MSN  
mars\_lavigne@hotmail.com  
我非常想找人辅导  
因为换了专业,老师正在讲这个  
我听不懂,而且还是外语,所以希望能听到中文讲解



谢谢

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2006-10-12 17:01 by myself

非常遗憾，网络环境不能用任何聊天工具，只能Email。  
在私人留言里留言吧！

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2006-12-01 10:58 by butler

感觉这个文章非常精典，我把它也复制到我的BLOG上面去了。  
谢谢楼主对此文的收集整理！

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2006-12-05 17:02 by Aaron

没得说，太棒了！！

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2006-12-23 10:17 by 陌生人

确实是非常非常好的文章！谢了

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2007-04-03 13:55 by nighteblis

good...虽然还有很多看不懂呢。

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2008-01-22 13:17 by 分段方法地方


很好，很强大

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2008-05-28 14:05 by 萝卜

看不懂

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2008-07-03 11:55 by 疯子

对于整理这些知识的人 我想说 太强 对于楼主 我想说 强

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2008-10-24 18:38 by vrs\_Saber

LZ，请问下文章前面说的是函数地址最先入栈，可是为什么后面的那个图确实参数先入栈，函数地址在入栈，最后是变量？？

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2009-04-17 20:17 by 创意产品网

学习了

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2009-04-29 17:14 by 创意家居


不错不错

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2009-06-01 18:32 by bneliao


非常不错，写得很好

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2009-09-09 22:40 by 雷勇


栈区（stack）— 由编译器自动分配释放  
我觉得这句话就是错的

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2010-04-15 19:51 by ff

高手

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2010-05-10 14:30 by 叶叶

受益匪浅

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2011-03-11 20:30 by Bourbon


写的太好了，解了很多困惑。

 **# re: 堆栈，堆栈，堆和栈的区别[未登录]** [回复](#) [更多评论](#)  
2011-05-18 18:40 by 王强


程序的内存分配可以画个图展示一下吗？谢谢！

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2011-08-01 22:31 by 牛b


难得一见的好文章！

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2011-08-24 23:03 by 凌雨


@Teresa  
写得太好了

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2011-12-28 00:57 by 吴同学


我一定会仔细研究!!

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2012-03-06 21:28 by VB小青年

辛苦了啊，我看都费劲.....楼主NB

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2012-04-16 15:39 by Lorainee

非常棒，写的很到位。

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2012-05-03 17:45 by material2010

相见恨晚

 **# re: 堆栈，堆栈，堆和栈的区别** [回复](#) [更多评论](#)  
2012-07-13 11:06 by zhangleifly

转走先咯～谢

# re: 堆栈，堆栈，堆和栈的区别 回复 更多评论  
2012-10-23 17:05 by 李冬

呵呵，当初应聘的时候被问到了堆和栈的区别，当时回答的一般啊。如果当时就看了这篇文章，估计应聘的时候能把那考官吓死吧。

# re: 堆栈，堆栈，堆和栈的区别 回复 更多评论  
2013-05-02 16:30 by lucas

感谢楼主，不过有个小问题  
2.5堆和栈中的存储内容  
栈： 在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。  
  
这个是不是应该是 先压参数，然后再压返回地址的说？

# re: 堆栈，堆栈，堆和栈的区别 回复 更多评论  
2015-06-10 11:23 by Grubby

“原作者不祥!”  
这句话，真实包含深深的怨念

刷新评论列表

只有注册用户登录后才能发表评论。

【推荐】超50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码

库

双11全球狂欢节

阿里云

不只是5折

云服务器50%OFF

瓜分千万红包

充值返5% 新购返10%

立即参加

广告

双11全球狂欢节

阿里云

不只是5折

云服务器50%OFF

瓜分千万红包

充值返5% 新购返10%

立即参加

广告

管理



