

54 Stack vs Heap Memory in C++

堆与栈内存的比较

当我们的程序开始的时候，它被分成了一堆不同的内存区域，除了堆和栈还有很多别的东西，但我们最关心的就是这两个：栈和堆。

1. 什么是栈和堆？

应用程序启动后，操作系统要做的就是将整个程序加载到内存，并分配一大堆物理RAM（[随机存取存储器：Random Access Memory](#)）以便我们的实际应用程序可以运行。

栈和堆是RAM中实际存在的两个区域：

栈`stack`通常是一个预定义大小的内存区域，通常约为2兆字节（2MB）左右；

堆`heap`也是一个预定义了默认值的区域，但是它可以增长，并随着应用程序的进行而改变。

重要的是要知道这两个内存区域的实际物理位置都是在RAM中，很多人倾向于认为栈存储在CPU缓存中或类似的位置，它确实因我们不停访问而在缓存中活跃，但不是所有的栈内存都会存储在这里，这并不是它的工作方式。只要记住这两个内存区域的实际位置都在我们的内存中，这就是为什么内存中有两个不同区域的原因。

2. 栈和堆的区别

我们的程序中，内存是用来存储运行程序所需的数据的，不管是从局部变量还是从文件中读取的东西。而栈和堆就是我们可以存储数据的地方。

它们的工作原理非常不同，但本质上做的事情是一样的：我们可以要求C++从栈或者堆中给我们一些内存，顺利的话它会给我们一个要求大小的内存块。而不同之处在于，它会**如何分配内存**。

分别在栈和堆上定义一个 `int`，`array`，`struct`：

```
C++

int value = 5;
int array[5];
Vector3 vector;

// 我们需要new关键字来在堆上分配

int* hvalue = new int ;
*hvalue = 5;
int* harray = new int[5];
Vector3* hvector = new Vector3(); // 圆括号是可选加不加的
new int[5];
```

主要的区别就是我们在堆上分配内存需要用 `new` 关键字

下面进入内存窗口检查：

```
struct Vector3
{
    float x, y, z;

    Vector3()
        : x(10), y(11), z(12) {}
};

int main()
{
    int value = 5; //打个断点
    int array[5];
    Vector3 vector;
    array[0] = 1;
    array[1] = 2;
    array[2] = 3;
    array[3] = 4;
    array[4] = 5;

    int* hvalue = new int ;
    *hvalue = 5;
    int* harray = new int[5];
    Vector3* hvector = new Vector3();
    harray[0] = 1;
    harray[1] = 2;
    harray[2] = 3;
    harray[3] = 4;
    harray[4] = 5;

    LOG(value);
    LOG(*hvalue);
    std::cin.get();
}
```

2.1 栈分配

内存中查看&value如图：cccc在debug模式下表示我们还没有初始化这个value


```
Address: 0x00880428
0x00880428  05 00 00 00
0x00880437  00 90 d5 87
0x00880446  00 0c e8 f0
0x00880455  00 00 00 02
0x00880464  00 00 00 00

Main.cpp x Static.cpp
HelloWorld (Glob
25
26 int* hvalue = new int
27 *hvalue = 5;
28 int* harray = new int
```

可以发现hvalue和harray的地址差的很远

```
Address: 0x00880708
0x00880708  01 00 00 00 02 00 00 00 03 00
0x00880717  cd cd cd cd fd fd fd fd 16
0x00880726  00 00 e8 b6 87 00 d8 d5 87 00
0x00880735  dd dd dd dd dd dd dd dd dd dd
0x00880744  00 00 00 00 00 00 00 00 00 00

Main.cpp x Static.cpp
HelloWorld (Global Scope) ma
31 harray[1] = 2;
32 harray[2] = 3;
33 harray[3] = 4; ≤ 4ms elapsed harray[3] = -842150451
```

在堆中分配 `new` 后要调用 `delete` 关键字来释放内存，用智能指针的 `make` 也一样会帮你调用关键字，所以我们需要手动去释放内存。

```
delete hvalue;
delete[] harray;
delete hvector;
```

而栈中分配内存时，一旦这个作用域结束，你在栈中分配的所有内存都会被弹出，内存被释放。这个作用域可以是任何形式，可以函数作用域，比如main函数，或者只是个空作用域，甚至可以使for、while循环等任何作用域

语句：

```
C++

int main()
{
    {
        int value = 5;
        int array[5];
        array[0] = 1;
        array[1] = 2;
        array[2] = 3;
        array[3] = 4;
        array[4] = 5;
        Vector3 vector;
    }
}
```

当这个作用域结束时，所有在这个作用域内栈内分配的东西，都会被弹出、被释放、被回收了，因为栈移动到了它原来的位置（我们进入这个作用域之前的位置）。

这也是栈和堆之间重要的区别之一，释放内存没有任何开销，因为栈释放内存与分配一样，不需要将栈指针反向移动然后返回栈指针地址，我们只需要弹出栈中的东西，栈指针自然就回到了作用域开始之前。（一条CPU的删除指令就可以释放所有东西）

3. new关键字实际做了什么

具体见[39 The NEW Keyword in C++](#)

new 关键字实际上调用了一个叫做 **malloc** 的函数（memory allocate）的缩写，这样做通常会调用底层操作系统或平台的特定函数，这将在堆上为你分配内存。当你启动应用时，你会被分配到一定数量的物理RAM，而你的程序会维护一个叫 **free list**（空闲列表）的东西，它的作用是跟踪哪些内存块是空闲的并储存它们的位置。当你使用 **malloc** 请求堆内存时，它可以浏览空闲列表，找到一块符合大小要求的内存块，然后返回你一个它的指针，并记录分配的大小和它现在是否被分配的情况（这样你就不能使用这块内存了）。

这里想说的重点是，在堆上分配内存是一大坨事情，而在栈上分配内存就像一条CPU指令。这两种主要内存的区别就是分配方式的区别，可以从汇编指令中看到，声明变量时栈分配的指令就一两行，而堆分配就是一大段指令了，之后还要调用 **delete**，这又是大段指令。

所以事实上，如果可能的话你应该**尽量在栈上分配内存**。在堆上分配的唯一原因是如果你不能在栈上分配，比如你需要让它的声明周期比你在处理的作用域更长，或者你特别需要更多的数据，比如我想加载一个50MB的纹理材质，这就不适合在栈上分配，因此你不得不在堆上分配。

性能的不同是因为分配方式，所以理论上如果你在运行你的程序前在堆上预先分配一个内存块，然后从这个预先分配的内存块中进行堆分配，那栈、堆分配就基本一样了，你唯一可能要处理的就是 **cpu cache miss** 的问题（缓存不命中），但miss的数量可能不够造成麻烦。所以当你调用 **new** 时，你需要检查free list，请求内存再记录所有内容，这就是堆相比于栈慢的地方，而实际的访问（CPU、缓存）通常可以忽略不计