C程序的内存管理 (https://casatwy.com/ccheng-xu-de-nei-cun-guan-li.html)

Date Mon 22 December 2014 Tags c (https://casatwy.com/tag/c.html) / memory (https://casatwy.com/tag/memory.html)

简述

自从自动内存管理嵌入到各种各样的语言之后，我们就很少会去关注这方面的事情了，这些功能的设计者和实现者们为此付出的努力值得我们称赞，期间也涌现了多种不同的内存管理方案。目前大部分语言的主流内存管理方案是Garbage Collector。苹果推出的Auto Reference Count也因为基于编译器自动添加手工计数的代码而带来了更好的性能提升。我在这里总结了一些从上古时代到如今在C程序下进行内存管理的技术。同时，也为你更深入地了解其他语言中内存管理模块的原理提供了知识背景。

管理固定大小的内存

栈内存

栈内存应该是最容易管理的了，只要理解 生存域 就能理解占内存的管理方式。生存域就是这样：

```
int main ()
{
    int array[] = {1,2,3,4,5};
                            ------------------|
    if ( i > 0 ) {          |
        ...                 |
        ...                 |
    }                       |
    foo(array);             |
                            |
    {                       |
        int k = 9;        ---               |
        array[1] = k;       |->k的生存域     |->array的生存域
        ...               ---               |
    }                       |
}                           |
                            |
void foo(int * const number) {              |
    int j = 0;              |
                          ---               |
    ...                     |               |
    number[2] = j;          |->j的生存域    |
    ...                     |               |
                          ------------------
}
```

基本上就是变量所处 { 和 } 之间的区域都叫这个变量的生存域。当程序执行到对应 } 的时候，就会将这个 } 对应生存域内的所有变量的内存释放给操作系统。

优点

- 程序员不需要为此做任何额外的事情，不用添加变量记录使用轨迹，也不用在内存中额外记录引用次数，程序自己会帮你处理好这些问题的
- 在 setjmp 和 longjmp 中，栈内存是安全的

setjmp 和 longjmp 是C99引入的可以模仿C++的exception行为的一种方案。一般情况下，所有在 longjmp 的内存在声明的时候都需要加 volatile ，让程序每次都从内存读取数值而不是在cache中读取。然而对于栈内存而言，可以不用 volatile 。

- 在多线程环境，递归环境，异步信号处理中，栈内存都是安全的

缺点

- 一般情况下，程序中都会有栈内存和堆内存混合使用，当函数在使用指针类的形参时，因为难以区分这个指针对应的内存是栈内存还是堆内存，因此就很难采取正确的内存管理措施。

上面的例子中， `foo` 函数接收了一个指针参数，如果这个指针是堆内存，那么就需要进行引用计数加一的操作（具体操作取决于内存管理方式），但现在在 `foo` 函数中其实无法区分这到底是栈内存还是堆内存，境况就比较尴尬了。

- 你必须要知道你的栈内存应当分配多大才合适，否则会引发安全问题

函数调用栈和变量内存栈其实是共用的，当内存溢出的时候，会覆盖掉对应的函数调用栈，这种覆盖不会引起程序中止或其他情况，十分安静。当函数执行完毕后，操作系统会取栈顶第一个元素作为跳转地址，然后就会直接跳转到被溢出的地址上了。举个例子：

exploit.c:

```
int main() {
    char string[1];
    printf("here i am");
    scanf("%s", string);
    return 0;
}
```

第一步程序进入main，然后为 `string` 分配了一个栈内存，此时栈内情况：

```
        |--------|
top -> | string |
        |________|
```

然后进入scanf函数，记住，操作系统记录函数调用的栈和存放栈变量的栈是同一个！此时栈内情况：

```
         |--------------|
top ->   | back address |  # 此时back address指向main函数中的 return 0; 这一指令的地址:0x1234
         |    0x1234    |
         |______________|
         |              |
         |    string    |
         |______________|
```

然后你在终端输入远大于string长度的数据，比如'npppp...'，然后回车，此时已经内存溢出，但scanf函数还未结束，此时程序也不会中止。此时栈内情况：

```
        |--------|
top -> | pppppp |   # 原来的back address被溢出成pppppp，而不是原来的0x1234了
        |________|
        |        |
        | string |
        |________|
```

然后scanf函数完成任务，取栈顶地址返回，此时就会将 `ppppp` 取出，作为地址返回。由于这个地址是非法地址，程序此时就会报 `can't access` 的错误，程序终止。我们可以更加别有用心一点，计算好溢出长度（一般情况下溢出长度跟栈内存的长度不一样，需要通过fuzz手段来确定对应长度），将溢出后覆盖的地址设为可以访问的地址，比如这里假设 `printf("here i am");` 的地址为0x1230，我们在输入string的时候输入 `npp...pp1230` (此处的1230其实是对应1230这个数字的ASCII码)，那么栈内情况就会变成：

```
        |--------|
top -> | 0x1230 |   # printf("here i am");
        |________|
        |        |
        | string |
        |________|
```

输入完毕按回车，你会看到程序就跑到printf那边去了，又输出了一次"here i am"。这就是一个典型的内存溢出漏洞，及其攻击方式。如果使用的是堆内存，在内存溢出之后因为溢出的那部分的内存使用权在管态，程序会直接中止，并且报 `address can't access` 的错误。

- 内存的合法访问范围被限制在一个固定的区域中

大多数情况我们其实是希望在另一个生存域中也能访问到对应内存的，使用栈内存就很容易出现野指针，因为你不知道什么时候它就被回收了。只有在递归调用的时候，栈内存才相对比较安全。

- 拥有这个栈内存的函数不能将这个栈内存的指针返回出去

这同样也是由于在这个函数以外的地方已经超出了变量生存域的原因。return出去的就变成野指针了。

总结

由于栈内存有以上这些优缺点（特点），我们一半都不太会用栈变量来管理广泛使用的内存和大片内存。一方面是由于栈变量有生存域限制，广泛使用的内存基本上都不能有这些限制。另一方面也是C程序调用参数的方案主要是值传递，大片内存一般都不会传整个变量，这样会导致操作系统copy的时候开销太大，如果使用传址调用，则会遇到被调用函数不知道如何管理指针所对应的那片内存的尴尬。

静态分配内存

所谓静态分配内存，就是在最外面的地方使用 `static` 来声明变量，由于这个变量被声明成了全局静态变量，这个变量所占有的内存也就变成了全局静态内存，即使是跨文件的情况也能通过变量名访问这片内存。这种做法突破了栈内存关于生存域的限制。

优点

- 这是解决生存域问题最简单的方案，全局可用。

由于这个是全局变量，任何地方想要用到它，只要 `extern` 一下就好。它在程序刚起来的时候（正式调用main函数之前）就会被操作系统分配，直到程序退出才会被操作系统回收。如果你的这个变量是一个全局单例，使用这种方案就不太容易会出错。

- 由于这是单独的一片内存区域，函数可以将最终结果存放在这个区域，然后返回指针，避免了值传递的低效率。

实际做法可以先申请一大片内存，比如 `uint8_t buffer[65536];` ，然后函数把生产出来的结果存放在buffer的某个位置，比如 `buffer[10]` ，然后把 `buffer[10]` 的指针作为结果return出去。 `buffer[20]` 或者其他的地方就可以给其他函数使用了。只要调用者和被调用者约定好偏移，这么做问题就不大。但如果涉及多线程、信号处理、递归等情况时，这种做法就不行了。另外一种情况就是这样的做法内存利用效率也不高，内存碎片会比较多，因为我们往往在约定的时候会划一个比较大的范围，大部分情况下，这一块很少是被完整利用的。

你也可以用循环使用的方法，比如说这一次使用0-9，下一次使用10-19，再下一次使用20-29这样，然后用完了再回头从0开始用，这样风险和成本都会不小，你要协调好各自函数操作这个buffer的关系，尽量少出bug。不过这正是编程的乐趣所在，不是么？:D

还有一种终极方案，需要结合下面提到的类似引用计数的方案来操作：在申请了这片buffer的同时，维护一个bitmap，1表示对应位置内存有用，0表示对应位置内存可以另作他用。通过引用计数的方案来维护这个bitmap，这样每次需要使用内存的时候，通过bitmap就可以找到有效内存的偏移，这样就总能保证取出的内存是可以使用的。这种方案相对而言还是比较广泛的，能在分配偏移的时候根据bitmap的情况采取一些策略来减少碎片的产生。但由于是静态buffer，因此就要求程序使用内存的数量可预测。

- 在静态内存中存的值会一直都在，你可以拿它作为多线程交换数据用，或者作为跨函数的中间结果的缓存用。

举个例子：

```
    static int globalVariable;

    char foo() {
        ...
        globalVariable = mid_result;    // globalVariable存储了foo()函数运行过程中的一个中间结果
        ...
        return result;                  // foo()函数的真正任务是返回result
    }

    void bar() {
        ...
        something = count + globalVariable; // bar()函数需要使用foo()函数的某个中间结果来完成任务，foo()正好存储了这个中间结果，
可以直接拿来给bar()用
        ...
    }
```

缺点

- 这是解决生存域问题最粗暴的方案，业界喷的也比较多

它虽然保证了内存全局可用，但由于只有程序结束之后对应的内存才能被回收，这导致内存的使用效率不高。另一方面，全局变量很容易造成代码恶化，当这个变量被多个使用者使用的时候，你就不能保证你每次从这里取出来的数据一定是你期望的数据，因此在设计的时候我们一般都是需要谨慎考虑是否要引入全局变量的。

- 容易引发命名空间的问题

由于全局变量哪儿都有用，一旦出现两个重名的全局变量就坑爹了，这样的bug还特别难调，只有特别熟悉整个项目代码的人才能想到可能是重名导致了问题。

- 多线程情况下会有坑

由于它全局变量的特点，我们可能会想到使用它来作为不同线程交换数据的地方。这是可以做到的，但是需要对这个变量进行临界区改造，不光要保证同一时间只有一个线程使用它，还要根据具体需求，保证同一功能区块内，只有一个线程使用它。否则就会出现1号线程还没完全跑完这个变量相关的代码。就被2号线程给把值改了，后面1号线程怎么跑就都不会出正确结果了。做临界区改造的成本还是蛮大的，一般都是使用PV操作来执行这样的任务，需要为此额外开辟内存来记录变量引用次数，以及设计一个runloop使之能够让其他线程执行wait()操作。

动态分配内存，并使用引用计数来管理

这是相对简单且容易采用的办法，实现方案有很多。实际使用时候的体验就类似于苹果在没推出ARC (Automatic Reference Count) 时候的MRC (Manual Reference Count)。具体原理是这样：

```
1. 使用malloc()或calloc()申请内存，并为这个内存初始化引用次数为1
2. 在内存使用过程中，如果当前片段需要成为这块内存的拥有者，那就调用一个方法使得这个内存的引用值加一
3. 当拥有者使用这块内存完毕，就再调用一个方法使得这个内存的引用值减一
4. 当引用值为0时，调用free()，让操作系统将内存回收
```

然后具体的实现方案也可以是以下这些：

1. 让引用计数跟随指针的方案

可以将内存、存储引用计数的变量打包成一个 `struct` ，然后以这个 `struct` 为单位去向操作系统要内存。操作引用计数的函数就只要修改这个 `struct` 的引用计数变量就好了，当引用计数变量值为0时，自动调用free()函数就好。

1. 引用计数不跟随指针的方案

先申请一大片内存，然后再开辟一个bitmap负责维护这一大片内存的使用情况，bitmap中的0表示某一区域的内存可用，其它数字则表示这片区域内存的引用计数。每次代码需要申请内存的时候，根据bitmap的情况从它维护的这一大片内存中挖出一块来给出去。当那一大片内存不够用的时候，就再申请一大片，然后再开辟一个bitmap来维护这片新的内存，当某一大片内存使用过后，bitmap数值全都是0的时候，就把这一大片内存一起回收掉。总的来看就是一大片内存由bitmap维护，然后bitmap通过引用计数来维护。

方案1的优点

- 内存碎片极小，约等于没有

因为是随用随取，基本上不太可能出现有内存碎片的情况，内存使用效率高。

- 实现方案简单

其实只要定义一个struct，然后写一个retain()函数（用于增加引用计数），和一个release()函数（用于减小引用计数），把相关逻辑都封装在这两个方法里面就好了。特别简单。

方案1的缺点

- 性能低下

这种方案有潜在的可能去频繁申请和释放内存，然而这两种操作，尤其是内存申请，是很消耗性能的。对于一般的客户端程序来说，或许可以忍受，但是在服务端程序中，往往都会因为这个缺点而采取方案2。

方案2的优点

- 性能好

由于一次申请了一大片内存，只要这片内存还够，后续的内存需求都可以不需要通过向操作系统来申请，性能就好很多。

方案2的缺点

- 长时间运行后，内存会有大量内存碎片的存在

第一轮使用内存的时候，差不多是可以做到内存碎片很小的。随着运行时间的增长，bitmap中维护的内存来来回回地被重用之后，就会出很多的内存碎片。而且这时往往不适合进行内存碎片整理，否则会造成很多野指针。假设运行期间一共分了10片大内存，极端情况就是10片大内存每片都只是使用了很小的一块，由于不能做内存整理，就导致内存浪费比较大。某种程度上讲，这属于内存泄漏。

- 实现方案相对复杂

你要实现一个bitmap，以及bitmap相关的维护函数，比方案1会复杂很多。

关于方案1和方案2的总结

你会发现，方案1的优点就是方案2的缺点，方案1的缺点就是方案2的优点，具体采用哪种方案，是要通过实际情况考虑的。如果考虑性能更多，那就选择方案2，如果考虑内存更加高效，那就选择方案1。事实上针对方案2容易导致的内存碎片问题，也有一个优化方案，就是再额外做一层抽象，让逻辑上连续的内存在实际上可以不连续。于是我们存放数据的时候可以根据bitmap的碎片进行见缝插针式的内存使用。这个优化方案看上去很好，但实际上增大了代码的复杂度，你需要维护一个B树来进行这个抽象，在实际应用中有点儿得不偿失。

这一类目还没结束，上面只是讨论了实现引用计数的两种方案的优缺点，下面我要说一下使用 引用计数 来进行内存管理其自身的优缺点。这些优缺点在方案1和方案2中都是普遍存在的。

引用计数方案的优点

- 提高内存使用效率

从宏观上看，引用计数方案能够提高内存使用效率，程序能够通过引用计数来知道哪片内存不再使用了，这片不再使用的内存就能够及时被操作系统回收。

- 没有生存域限制，没有命名空间限制

内存只要被malloc后，就一直可用，除非被free。由于不通过一个全局变量来hold住这块内存，我们可以不用关心命名空间冲突的问题。

引用计数方案的缺点

- 对程序员有更加高的要求

相比于栈内存来说，进行动态内存分配要写的代码量要更大一些，管理要更细致一些，否则很容易出内存泄漏或者野指针的问题。何时进行引用计数的加一，何时进行引用计数的减一，这些需要程序员能够了解得非常清楚，有时候忘记调用增一或减一的方案，就会引发bug。调用这些辅助方法本身跟业务逻辑无关，但它们确实会影响业务逻辑。

在某些情况下也容易出现循环引用，从而导致内存无法被回收，比如这样：

```
A: { ... B.a ... }
B: { ... A.a ... }
```

当要回收A的内存时，由于它声明了B.a的所有权，A的内存是不能被回收的，只有等B被回收后，A才能被回收。然而此时B又对A.a声明了所有权，B是不能被回收的，因此形成了一个死循环（有一个术语专门用来描述这种死循环：retain cycle），造成A和B都不能被回收，从而导致内存泄漏。一般情况下要避免这个问题，就要分清楚内存所对应的变量在你的程序中的逻辑层级关系。一般是高层级的变量拥有低层级的变量（对该变量进行引用加一），低层级的变量不拥有高层级的变量（引用计数不加一），遵守这个原则就能够避免retian cycle的出现。

- longjmp、setjmp的情况下很难处理引用计数对应的内存

前面说过longjmp、setjmp就类似于exception机制。在某一段代码触发longjmp的时候，这段代码所相关的变量对应的内存其实就已经不需要再用了，这时候这些内存的引用计数往往都大于0，甚至在当前上下文你都不一定能够确定哪些变量是有用哪些变量是无用的，这时你就无法通过引用计数机制来回收内存。

不过说起来exception机制在刚从C++诞生的时候业界都很欢乐，终于有了一种程序出错而可以不中止程序的方案了。但随着时间的推进，人们越来越认识到exception机制有很多坑，近年大家都不推荐在程序中使用exception机制来处理运行时错误，更多的是采用error number，error对象的机制。这方面具体的讨论不在本文范围内，大家可以各自Google一下。

链式内存分配

这是为了解决引用计数在longjmp和setjmp情况下内存不容易管理的缺点。它是这样实现的：

```
1. 实现一个链表
2. 链表的节点包含指向已经申请内存的指针、这块内存的引用计数、这块内存当前所处的函数指针（用于标志作用域）
3. 每次申请内存的时候都生成一个这样的节点，然后挂在链表上
4. 每次释放内存的时候都将这个节点删除，并free()对应的内存
5. longjmp之后根据函数指针遍历链表，找到所有对应函数指针的节点，计数减一，若减一之后计数为0，则释放内存
```

实质上就是用链表来进行内存使用的跟踪，这样的链表在程序中可以一个也可以多个，然后由一个总表去维护这些链表，这么做可以防止遍历太长的链表造成太大的性能消耗。

优点

- longjmp、setjmp有效

这个自然不必多说，这个方案就是为解决这个问题而诞生的。

- 链表带来了非常好的灵活性

实际操作中，可以每个模块一个链表，甚至每个功能一个链表。链表不光可以用于longjmp，你也可以写一个仅在debug模式下启用的功能，这个功能用于统计每个函数内存的使用量，这在调试优化的时候是个非常好的数据来源。

缺点

- 依旧不能解决retain cycle的问题

链式内存分配的方案本质上还是属于引用计数，只是解决了exception情况下的内存处理，但并没有解决retain cycle，程序员依旧需要当心这种情况。

管理不是固定大小的内存

长度可变的数组

这个是C99引入的新特性，就是这样：

```
void foo(int n) {
    int array[n];
    array[n-1] = 1;
    printf("%d", array(n-1));
}
```

大部分C语言教程会说这种写法是错误的。因为array占用的是栈内存，栈内存是不能在运行期间动态分配的。但自从C99标准之后，这种写法就不会引起编译错误了。长度可变的数组倒是一个蛮不错的功能，以前要实现长度可变的数组，大部分都会用链表去做，使用这个功能之后就省事儿很多了。而且栈内存也是内存呀，当然可以拿它来做别的事情了~

优点

- 栈内存的所有优点它都有

多线程，安全。exception，安全。递归，安全。异步信号处理，安全。不用写额外的内存管理代码，方便。

- 类似动态内存分配的效果

你可以将长度作为参数去构建你的数组，然后将这片内存另做它用。这是原始的栈内存分配不能做到的。

缺点

- 栈内存的所有缺点它都有

生存域限制，接收参数的内存管理方案易混淆，这些都依旧是缺点。唯一好的地方就是你可以不用知道传递的数据有多大了，但是即便这样，还是

不能做到真正动态内存那般随用随取，因此使用的时候也还是会有颇多限制。

动态内存重分配

有时候一片内存不够大，但是又不适合再开辟一个新内存，你就会需要将原来的内存进行重新分配。比如说你有一个缓冲区用于存放数据，当数据大于缓冲区容量时，你需要让缓冲区容量变大。如果开辟一个新内存，缓冲区的连续性就破被破坏了，所以你会对缓冲区进行内存的重新分配，这样就能放得下数据了。一般来说，这是对未知长度的数据的一种处理方案，重分配的函数就是 `realloc()`。重分配时，操作系统会将旧的那片内存的数据复制到新的扩展过的内存里，这样就能保证连续性了。

优点

- 能够跟前面提到的动态内存管理方案相结合，提供连续的，更大片的内存
- 因为本质还是动态内存操作，所以动态内存所有的优点它都有
- 可以引申一种新的内存管理方案

我们可以约定由函数的调用者负责申请和管理内存，然后将指针传递进去，子函数发现内存不够的时候，重新分配一下就行。然后这片内存的管理工作就由调用者来管理，子函数运行期间，内存都是可用的。子函数运行完毕，父函数拿到结果之后，一般来说也就是直接free了。这种方案其实也是相对使用比较广泛的一种内存管理方案。

缺点

- 性能消耗厉害

因为有一个额外的copy操作，如果你频繁进行重分配的话，copy和内存申请都会带来更多的性能消耗，一半儿而言重分配的时候也尽量大片大片地扩展，免得出现经常不够用然后经常要重分配的情况。

- 动态内存管理的缺点也都有

你还是需要引入其他的内存管理方案来管理重分配的内存，前面提到了很多动态内存管理的方案，在这个时候也都需要根据情况采用，而且在扩展内存方面，由于扩展后是一片新内存，旧的指针变量保存的地址就会失效成为野指针。

内存回收器（GC，Garbage Collector）

GC的话题太大，实现方案也多种多样，有跟踪内存使用路径的，也有基于引用计数的。C环境下有一个libgc (https://launchpad.net/libgc)库实现了一个内存回收器，惠普也搞过一个GC，BDWgc (http://sourceforge.net/projects/bdwgc/)，使用的人也很多。目前业界大部分语言是自带GC的，java，python，php，javascript都有，各自实现的方案也不一样，改造改造移植到C来也不是不行。所以真要评个优点缺点很难，因为不太好拿一个具象的东西进行分析。一般认为GC是内存管理的终极方案，虽然也有优缺点，但是基本上都优于上面提到的各种方案，而且还省时省力。只是我们要注意的是，同一套程序里不能同时存在两种GC，然而同一套程序里是可以存在多种内存管理方案的，这算是相对普适的一个特点了吧。

总结

C环境下面做内存管理其实是个苦力活，而且也没有什么全能方案能够解决所有的问题。我个人倾向使用引用计数，同时不使用exception机制来处理程序错误。写这篇文章的目的也是为了总结一下各种情况下的内存管理方案。文末推荐一篇文章(http://www.ibm.com/developerworks/aix/tutorials/au-memorymanager/)，是教你如何写一个属于自己的Memory Manager的，虽然不是GC，但也属于相对成熟的一种内存管理方案。比较长，也有点儿难度，但你都把我文章看完了，相信你再看那篇文章应该不是问题。

Comments

Back to top