3 分配器 allocators

1 C++ 内存配置操作和释放操作

```
1  class FOO{};
2  FOO *pf = new FOO;
3  delete pf;
```

对于上述代码，其在底层执行内容为：

line 2：new操作，首先调用::operator new分配内存 （2）调用Foo::Foo() 构造对象内容; ::operator new底层调用malloc分配内存。

line 3：delete操作，首先调用Foo::~Foo()将对象析构 （2）调用::operator delete释放内存; ::operator delete底层调用free释放内存。

出于分工的考量，STL 的allocators决定将这两个阶段分开。分别用 4 个函数来实现：

1. 内存的配置：alloc::allocate();
2. 对象的构造：::construct();
3. 对象的析构：::destroy();
4. 内存的释放：alloc::deallocate();

2 construct()和destroy()

construct()和destroy()主要负责对象的构造与析构。

construct()的源码为：

```
1  template <class T1, class T2>
2  inline void construct(T1* p, const T2& value) {
3    new (p) T1(value);                              //用placement new在 p 所指的对象上创建一个对象，value是初始化对象的值。
4  }
```

destory()的源码为：

```
1  template <class T>
2  inline void destroy(T* pointer) {
```

```
    pointer->~T();                                  //只是做了一层包装，将指针所指的对象析构---通过直接调用类的析构函数
}

template <class ForwardIterator>                    //destory的泛化版，接受两个迭代器为参数
inline void destroy(ForwardIterator first, ForwardIterator last) {
  __destroy(first, last, value_type(first));    //调用内置的__destory(),value_type()萃取迭代器所指元素的型别
}

template <class ForwardIterator, class T>
inline void __destroy(ForwardIterator first, ForwardIterator last, T*) {
  typedef typename __type_traits<T>::has_trivial_destructor trivial_destructor;
  __destroy_aux(first, last, trivial_destructor());        //trival_destructor()相当于用来判断迭代器所指型别是否有 trival destructor
}


template <class ForwardIterator>
inline void                                             //如果无 trival destructor ，那就要调用destroy()函数对两个迭代器之间的对象元素进行一个个析构
__destroy_aux(ForwardIterator first, ForwardIterator last, __false_type) {
  for ( ; first < last; ++first)
    destroy(&*first);
}

template <class ForwardIterator>                        //如果有 trival destructor，则什么也不用做。这更省时间
inline void __destroy_aux(ForwardIterator, ForwardIterator, __true_type) {}

inline void destroy(char*, char*) {}           //针对 char * 的特化版
inline void destroy(wchar_t*, wchar_t*) {}     //针对 wchar_t*的特化版
```

construct()比较好理解，就是直接调用new操作。

destory()的话就比较复杂，主要在于其有很多的特化版本（泛化、特化、偏特化可以百度了解），主要有以下版本：

1. 泛化版本 __destroy() （ForwardIterator, ForwardIterator）：

   根据是否是trival destructor（无关痛痒的析构函数）来进行选择

   1.1 特化版本（false）： __destroy_aux(ForwardIterator first, ForwardIterator last, __false_type)，即for循环一个个调用析构函数来析构。

   1.2 特化版本（true）： __destroy_aux(ForwardIterator, ForwardIterator, __true_type) {}，无关痛痒，什么都不做

2. 特化版本： （T *）对于传入一个对象的指针，直接调用析构函数
3. 特化版本： （char *, char *）char*型，什么都不做
4. 特化版本： （wchar_t *,wchar_t *）wchar_t *型，什么都不做

有这么多特化版本的原因还是因为trival destructor，对于trival destructor执行和不执行都一样，因此去执行那些trival destructor是很吃力不讨好的。

3 allocate()和deallocate()

allocate()和deallocate()主要负责与内存分配与释放相关的动作。

在STL源码中，allocate转调用::operator new实现，deallocate转调用::operator delete实现。

调用链路可理解为：

调用allocate分配内存->调用::operator new分配内存->调用malloc分配内存

调用deallocate释放内存->调用::operator delete释放内存->调用free释放内存

SGI对空间的配置和释放的设计哲学为：

1. 向 system heap 要求空间
2. 考虑多线程状态
3. 考虑内存不足时的应变措施
4. 考虑过多“小型区块”可能造成的内存碎片问题。

考虑到小型区块会导致内存破碎问题，SGI STL设计了一个双层级配置器。

其代码如下：

```
# ifdef __USE_MALLOC
typedef __malloc_alloc_template<0> malloc_alloc;
typedef malloc_alloc alloc; //使用第一级配置器
# else
typedef __default_alloc_template<__NODE_ALLOCATOR_THREADS, 0> alloc; // 使用第二级配置器
# endif
```

因为SGI使用了双层级配置器，因此需要对外提供一个接口，从而符合标准：

```
template<class T, class Alloc>
class simple_alloc {

public:
    static T *allocate(size_t n)
                { return 0 == n? 0 : (T*) Alloc::allocate(n * sizeof (T)); }
    static T *allocate(void)
                { return (T*) Alloc::allocate(sizeof (T)); }
    static void deallocate(T *p, size_t n)
                { if (0 != n) Alloc::deallocate(p, n * sizeof (T)); }
    static void deallocate(T *p)
                { Alloc::deallocate(p, sizeof (T)); }
};
```

对于足够大和足够小的定义决定了应该使用哪一级配置器，对于SGI STL而言，小于等于128bytes视为足够小。当配置区块超过 128 bytes时，调用第一级配置器。当配置区块小于 128 bytes时，采取第二级配置器。

第一层配置器直接使用malloc()和free()。

第二层配置器则使用 memory pool 的方式。

3.1 第一级配置器

```
//以下是第一级配置器
template <int inst>
class __malloc_alloc_template {

private:

//以下函数用来处理内存不足的情况
static void *oom_malloc(size_t);

static void *oom_realloc(void *, size_t);

static void (* __malloc_alloc_oom_handler)();

public:

static void * allocate(size_t n)
{
    void *result = malloc(n);                       //第一级配置器，直接使用malloc()
    //如果内存不足，则调用内存不足处理函数oom_alloc()来申请内存
    if (0 == result) result = oom_malloc(n);
    return result;
}

static void deallocate(void *p, size_t /* n */)
{
    free(p);            //第一级配置器直接使用 free()
```

```
}

static void * reallocate(void *p, size_t /* old_sz */, size_t new_sz)
{
    void * result = realloc(p, new_sz);            //第一级配置器直接使用realloc()
    //当内存不足时，则调用内存不足处理函数oom_realloc()来申请内存
    if (0 == result) result = oom_realloc(p, new_sz);
    return result;
}

//设置自定义的out-of-memory handle就像set_new_handle()函数
static void (* set_malloc_handler(void (*f)()))()
{
    void (* old)() = __malloc_alloc_oom_handler;
    __malloc_alloc_oom_handler = f;
    return(old);
}
};

template <int inst>
void (* __malloc_alloc_template<inst>::__malloc_alloc_oom_handler)() = 0;    //内存处理函数指针为空，等待客户端赋值

template <int inst>
void * __malloc_alloc_template<inst>::oom_malloc(size_t n)
{
    void (* my_malloc_handler)();
    void *result;

    for (;;) {                                                    //不断尝试释放、配置、再释放、再配置
        my_malloc_handler = __malloc_alloc_oom_handler;           //设定自己的oom(out of memory)处理函数
        if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }        //如果没有设定自己的oom处理函数，毫不客气的抛出异常
        (*my_malloc_handler)();                                   //设定了就调用oom处理函数
        result = malloc(n);                                       //再次尝试申请
        if (result) return(result);
    }
}

template <int inst>
void * __malloc_alloc_template<inst>::oom_realloc(void *p, size_t n)
{
    void (* my_malloc_handler)();
    void *result;

    for (;;) {
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }    //如果自己没有定义oom处理函数，则编译器毫不客气的抛出异常
```

```
        (*my_malloc_handler)();                                  //执行自定义的oom处理函数
        result = realloc(p, n);                                  //重新分配空间
        if (result) return(result);                              //如果分配到了，返回指向内存的指针
    }
}
```

上述代码的流程为：

1. 通过allocate()申请内存，通过deallocate()来释放内存，通过reallocate()重新分配内存。
2. 当allocate()或reallocate()分配内存不足时会调用oom_malloc()或oom_remalloc()来处理。
3. 当oom_malloc() 或 oom_remalloc()还是没能分配到申请的内存时，会转入以下两步中的一步：

   3.1 调用用户自定义的内存分配不足处理函数(这个函数通过set_malloc_handler() 来设定)，然后继续申请内存。

   3.2 如果用户未定义内存分配不足处理函数，程序就会抛出bad_alloc异常或利用exit(1)终止程序。

3.2 第二级配置器

在第二级配置器中，SGI 第二层配置器定义了一个 *free-lists*，这个*free-list*是一个数组，各自管理大小分别为8,16,24,32,40....128bytes的小额区块。

*free-list*节点结构为：

```
union obj{
    union obj * free_list_link;
    char client_date[1];
};
```

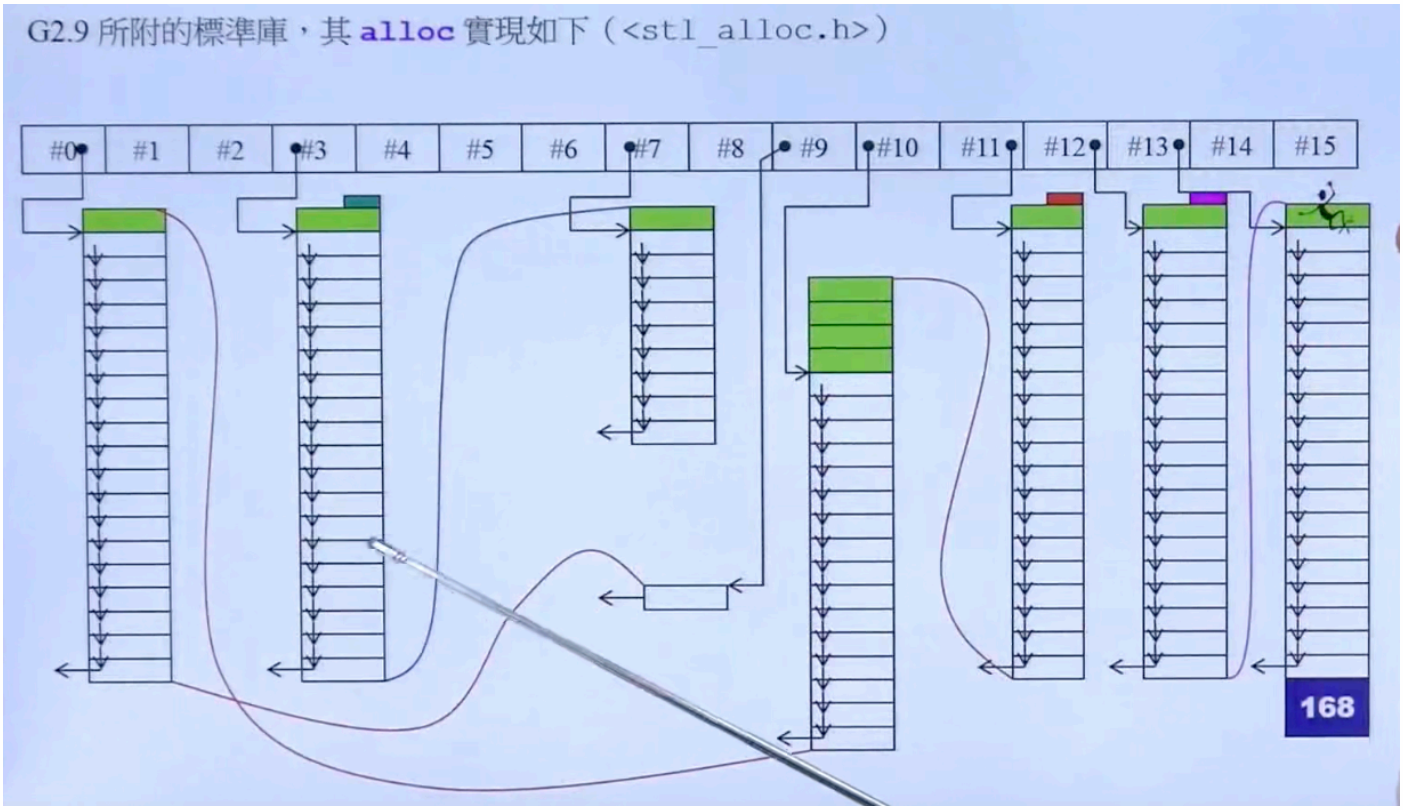
第二级配置器的部分实现源码如下：

```
template <bool threads, int inst>
class __default_alloc_template {

private:
  // Really we should use static const int x = N
  // instead of enum { x = N }, but few compilers accept the former.
# ifndef __SUNPRO_CC
    enum {__ALIGN = 8}; //小型区块上调边界
    enum {__MAX_BYTES = 128}; // 小型区块的上界
    enum {__NFREELISTS = __MAX_BYTES/__ALIGN}; // free-list的节点个数
# endif
  // 将bytes上调至8的倍数
  static size_t ROUND_UP(size_t bytes) {
        return (((bytes) + __ALIGN-1) & ~(__ALIGN - 1));
  }
__PRIVATE:
  union obj {
```

```
        union obj * free_list_link;
        char client_data[1];    /* The client sees this.        */
  };
private:
    //16个free-list
    static obj * __VOLATILE free_list[__NFREELISTS];
    //根据大小计算该用哪个区块，32->3
    static  size_t FREELIST_INDEX(size_t bytes) {
        return (((bytes) + __ALIGN-1)/__ALIGN - 1);
  }

  // Returns an object of size n, and optionally adds to size n free list.
  static void *refill(size_t n);
  // Allocates a chunk for nobjs of size "size".  nobjs may be reduced
  // if it is inconvenient to allocate the requested number.
  static char *chunk_alloc(size_t size, int &nobjs);

  // Chunk allocation state.
  static char *start_free; //内存池起始位置，只在chunk_alloc()中变化。
  static char *end_free;   //内存池结束位置，只在chunk_alloc()中变化。
  static size_t heap_size;

public:

  /* n must be > 0      */
  static void * allocate(size_t n)
  {
    obj * __VOLATILE * my_free_list;
    obj * __RESTRICT result;

    if (n > (size_t) __MAX_BYTES) {
        return(malloc_alloc::allocate(n));
    }
    my_free_list = free_list + FREELIST_INDEX(n);
    // Acquire the lock here with a constructor call.
    // This ensures that it is released in exit or during stack
    // unwinding.
#       ifndef _NOTHREADS
        /*REFERENCED*/
        lock lock_instance;
#       endif
    result = *my_free_list;
    if (result == 0) {
        void *r = refill(ROUND_UP(n));
        return r;
    }
    *my_free_list = result -> free_list_link;
    return (result);
  };

  /* p may not be 0 */
  static void deallocate(void *p, size_t n)
```

```
  {
    obj *q = (obj *)p;
    obj * __VOLATILE * my_free_list;

    if (n > (size_t) __MAX_BYTES) {
        malloc_alloc::deallocate(p, n);
        return;
    }
    my_free_list = free_list + FREELIST_INDEX(n);
    // acquire lock
#       ifndef _NOTHREADS
        /*REFERENCED*/
        lock lock_instance;
#       endif /* _NOTHREADS */
    q -> free_list_link = *my_free_list;
    *my_free_list = q;
    // lock is released here
  }

  static void * reallocate(void *p, size_t old_sz, size_t new_sz);

} ;

```

第二级配置器的结构：



3.2.1 allocate()

allocate()的源码：

```
static void * allocate(size_t n)
{
    obj * __VOLATILE * my_free_list;
    obj * __RESTRICT result;

    //要申请的空间大于128bytes就调用第一级配置
    if (n > (size_t) __MAX_BYTES) {
        return(malloc_alloc::allocate(n));
    }
    //寻找 16 个free lists中恰当的一个
    my_free_list = free_list + FREELIST_INDEX(n);
    result = *my_free_list;
    if (result == 0) {
        //没找到可用的free list，准备新填充free list
        void *r = refill(ROUND_UP(n));
        return r;
    }
    *my_free_list = result -> free_list_link;
    return (result);
};
```

ROUND_UP函数源码如下，其作用为：将要申请的内存字节数上调为8的倍数。

```
static size_t ROUND_UP(size_t bytes) {
    return (((bytes) + __ALIGN-1) & ~(__ALIGN - 1));
}
```

refill函数源码如下，其作用为：向内存池申请20块大小为n的一大块内存，将其挂在free-list上，并返回之。这个refill函数如allocate中所描述，就是在没找到可用的free-list时使用的，即我想要一块大小为32bytes的内存，然而发现没有了，此时就调用refill，去申请20个32bytes的内存以供使用。

```
/* Returns an object of size n, and optionally adds to size n free list.*/
/* We assume that n is properly aligned.                                */
/* We hold the allocation lock.                                         */
template <bool threads, int inst>
void* __default_alloc_template<threads, inst>::refill(size_t n)
{
    int nobjs = 20;
    char * chunk = chunk_alloc(n, nobjs);
    obj * __VOLATILE * my_free_list;
    obj * result;
    obj * current_obj, * next_obj;
    int i;

    if (1 == nobjs) return(chunk);
    my_free_list = free_list + FREELIST_INDEX(n);
```

```
16
17      /* Build free list in chunk */
18        result = (obj *)chunk;
19        *my_free_list = next_obj = (obj *)(chunk + n);
20        for (i = 1; ; i++) {
21          current_obj = next_obj;
22          next_obj = (obj *)((char *)next_obj + n);
23          if (nobjs - 1 == i) {
24              current_obj -> free_list_link = 0;
25              break;
26          } else {
27              current_obj -> free_list_link = next_obj;
28          }
29        }
30      return(result);
31  }
```

3.2.2 deallocate()

deallocate()的实现则较为简单，等同于一个链表插入操作，源码如下：

```
1   static void deallocate(void *p, size_t n)
2   {
3       obj *q = (obj *)p;
4       obj * __VOLATILE * my_free_list;
5
6       //如果要释放的字节数大于128，则调第一级配置器
7       if (n > (size_t) __MAX_BYTES) {
8           malloc_alloc::deallocate(p, n);
9           return;
10      }
11      //寻找对应的位置
12      my_free_list = free_list + FREELIST_INDEX(n);
13      //以下两步将待释放的块加到链表上
14      q -> free_list_link = *my_free_list;
15      *my_free_list = q;
16  }
```

参考文献：

[1] 侯捷.STL源码剖析[M].武汉：华中科技大学出版社，2002.6：43-69.

[2] https://www.cnblogs.com/zhuwbox/p/3699977.html