STL 阅读笔记

sharwen

目录

1 allocator

allocator 是 STL 模板的基础，主要承载工作分为三个方面：1、内存分配；2、对象构造；3、对象析构；4、数据初始化；5、内存回收。

1.1 内存分配

allocator 内存分配为层次内存分配，在 SGI-STL 源码中，为两级内存分配：第一级配置器使用场景为一次分配内存 $More\ Than\ \mathbf{128BYTE}$即超过 128 字节时，使用第一级内存配置器；第二级配置是一次分配内存 $Less\ Than\ \mathbf{128BYTE}$即少于 128 字节时，使用第二级配置器。使用两级配置器的目的是尽量减少内存碎片化的问题。

1.1.1 第一级配置器

第一级配置内存分配是对 malloc，realloc，calloc 和 free 的再次封装，因为这些函数是 C 语言的内存操作函数，不自带类似 C++ set_new_handler，因此 STL 为对内存分配失败加入了类似 C++ 的 new_hander 处理机制，允许客户端自己设定内存分配失败的处理函数。内存分配失败处理 body 是个 for 循环函数，如下：

```
// 不断尝试释放、配置
for (;;) {
  __my_malloc_handler = __malloc_alloc_oom_handler;
  if (0 == __my_malloc_handler) { __THROW_BAD_ALLOC; }
    (*__my_malloc_handler)();  // 调用处理例程，企图释放内存
    __result = malloc(__n);   // 再次尝试配置内存
  if (__result) return(__result);
}

```

1.1.2 第二级配置器

第二级配置器是 allocator 内存分配的精髓，为了降低内存碎片化问题，STL 将申请一个内存 pool 用作小块内存的分配回收和管理。第一级配置器维护 $\mathbf{16 = MAX\_SIZE/8}$ 个空闲内存链表，其中 MAX_SIZE = 128。

index 从 0 开始。空闲链表分别代表 $chunk\ size$ 为 8,16,24,32,40,48,56,64,72,80,88,96,104,112,120,128。分配会根据请求分配内存的大小，决定从对应的 free_list 中分配空闲内存。index = _S_round_up(size) / 8; 在内存对齐上，有一个有意思的写法，灵活利用了位运算 (ALIGN = 8)。

```
// 将任何小额区块的内存需求量上调至 8 的倍数
static size_t
_S_round_up(size_t __bytes)
{ return (((__bytes) + (size_t) _ALIGN-1) & ~((size_t) _ALIGN - 1)); }

```

第二级内存配置器的基本思想，分为：

1、如果申请内存大小向 8 对齐后对应的 free_list 中存在空闲内存，则将内存返回给客户端使用；

2、如果 1 中不存在空闲内存，则调用 _S_refill 函数申请内存，默认申请大小为 **20***size，其中 size 为内存客户端申请大小向 8 对齐后的大小，如果申请块 ***equal 1***，则直接返回给客户端使用，如果返回的 ***greater than1***，则第一块给客户端，剩下的块添加对 size 对应的 free_list 链表中；

3、_S_chunk_alloc，是第二级内存分配的核心，维护一个通过 malloc 申请的 heap 内存，第二步是调用此函数进行内存分配。此函数内存分陪时候会存在以下情况。

1) 如果申请大小（20*size）***LessThan***heap 中内存总额大小，则直接将 start 起始内存返回，并更新 heap 的 start 位置。

2) 如果 heap 内存总额大小 ***Lessthan*** 20 * size 但是够一个以上内存块的分配，则分配最大块数给调用者，余下的内存保留在维护的 heap 当中。

3) 如果一块内存都不够分配，则将 heap 中的内存挂载到 heap 内存大小对应的 free_list 当中。并进行下一步计算本次内存申请大小：注意此次申请大小为 2 倍上层调用的大小 + 总 heap 大小偏移 »4 的总和，此函数调用次数的增加会上升每次申请大小的偏移大小。

```
1  //_S_heap_size是内存配置器维护的heap总申请大小
2  size_t __bytes_to_get = 2 * __total_bytes + _S_round_up(_S_heap_size >> 4);
3  ...
4  _S_heap_size += __bytes_to_get;
5
```

此时尝试直接用 malloc 申请内存，如果申请成功，则递归调用 _S_chunk_alloc 更新成功申请内存的块数量。如果失败，则尝试是否能从已有的 free_list 中找到可用的内存，进行以下步骤：

a) 从申请内存块对应的 free_list 索引开始，循环查找 free_list 中是否存在空闲的内存块，如果存在，则递归调用自己更新 heap 指针和申请成功的内存块大小。

b) 如果 free_list 中不存在，则调用一级配置器，寄希望于 malloc 和 new_hander，如果成功递归调用自己更新 heap 指针和申请成功块大小，如果均失败，则抛出失败异常。

在第二级配置中，STL 巧妙利用了联合体来管理空闲内存，对配置起数据就是空闲内存块链表指针，对客户端来讲看到的就是真实的数据。

```
1  union _Obj {
2    union _Obj* _M_free_list_link;
3    char _M_client_data[1];    /* The client sees this.        */
4  };
5
```

1.2 对象构造

对象构造没什么好说的，仅仅是对 C++ operator new 的封装。见图1。

1.3 对象析构

对象析构主要分为两部分，分为两部分的标准是看对象是否存在 ***trivial deconstructor***，首先接受以下 trivial deconstructor 的含义，指的是对象使用默认析构函数，无自定义析构函数。见图1。

对于使用默认析够函数的对象，allocator 为了提升效率，啥也不做。

对于自定义析够函数的对象，allocator 通过循环一个一个对象调用对象的析够函数进行析够。

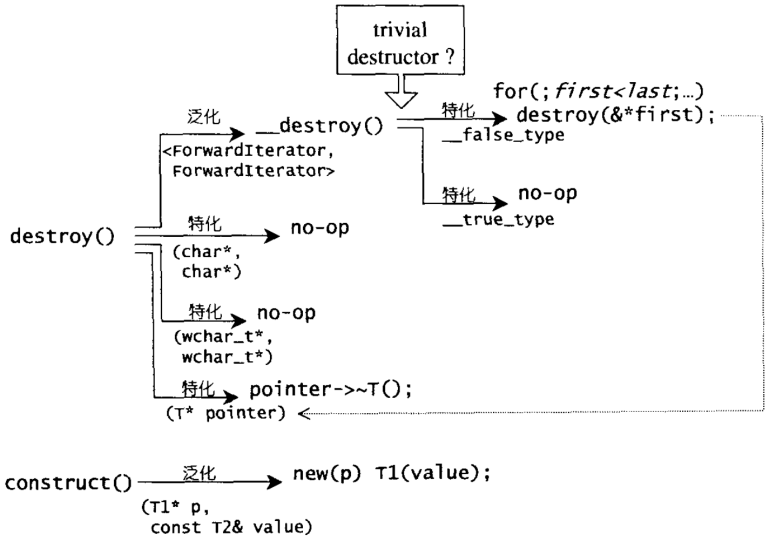

图 1: STL constructor deconstructor example

1.4 数据初始化

数据初始化主要包括 uninitialized_copy,uninitialized_copy_n,uninitialized_fill,uninitialized_fill_n 函数，这些函数核心都在于区分对象是否为 is_POD_type(pod,Plain Old Data)，

如果是 POD_type，则直接使用 STL 中的 copy 或者 fill 模版；

如果不是 POD_type，则利用循环依次对 [start_iterend_iter) 进行对象构造

1.5 内存回收

内存回收和内存分配同样分为两级，

当回收内存向上对 8 对齐后，如果大于 128byte，直接利用 free；当回收内存向上对 8 对齐后，小于等于 128byte，则作为 free block 挂载到对应的 free_list 索引链表当中去。

2 迭代器与 traits

2.1 迭代器分类

迭代器分为五类:

1、Input Iterator（只读）

2、Output Iterator（只写）

3、Forward Iterator

4、bidirectional iterator

5、random access iterator

前三类迭代器支持 operator++，第四种外加支持 operator–，第五种除了前两种 operator 外，同时支持指针算术运算能力。如图2

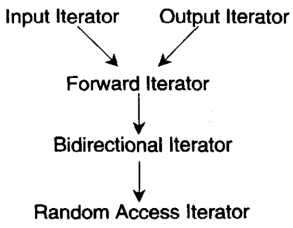

图 2: iiterator_classify and inherit relationships

2.2 traits 判断 POD

为了提升效率，SGI-STL 包含了非标准的特性，通过判断特定属性来决定模板的重载：

```
template <class _Tp>
struct __type_traits {
  typedef __true_type      this_dummy_member_must_be_first;

  typedef __false_type     has_trivial_default_constructor;
  typedef __false_type     has_trivial_copy_constructor;
  typedef __false_type     has_trivial_assignment_operator;
  typedef __false_type     has_trivial_destructor;
  typedef __false_type     is_POD_type;
};

```

对于 POD 类型，SGI-STL 对其进行了特化，五个成员均定义为 __true_type，如：

```
__STL_TEMPLATE_NULL struct __type_traits<char> { //__STL_TEMPLATE_NULL 为 template<>
  typedef __true_type     has_trivial_default_constructor;
  typedef __true_type     has_trivial_copy_constructor;
  typedef __true_type     has_trivial_assignment_operator;
  typedef __true_type     has_trivial_destructor;
  typedef __true_type     is_POD_type;
};

```

3 序列容器

3.1 vector

vector 内存空间策略：连续线性内存空间，与 ***array*** 的区别是可以动态分配内存；

vector 内存分配：a) 如果插入 position 内存不足，则以以前的元素个数的两倍空间申请新内存，然后将原来的数据:1)uninitialized_copy begin-position;2)construct(position,val);3)uninitialized_copy position - finish;4) 将原来占用的内存空间返还给系统。

vector erase 策略：1) 删除迭代器区间范围（start-last），将 last-finish copy 到 start 开始处，然后将 copy 返回的迭代器 i，到 finish 给 destroy 掉，同时更新大小。2) 删除一个元素 position，直接 copy position+1 - finish 到 position 处，然后 destroy 掉 finish-1 处的元素，更新大小。因此时间复杂度O(N);

vector insert 策略：1) 如果剩余的空间满足插入元素个数，如果插入位置 position 的位置之后的元素大于插入元素个数，则 a) 将最后 n 个元素 uninitialized_copy 到 finish 开始处（因为之后的元素没有

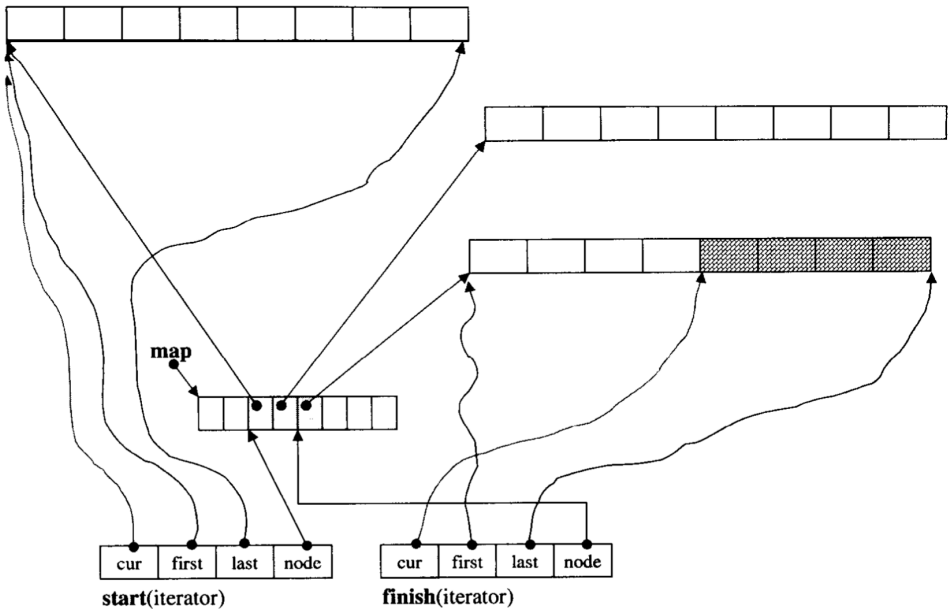

图 3: deque memory allocate

构造，所以 uninitialized_copy），b)copy_backward position - (finish-n) 到 old_finish 开始处，最后 fill 掉 n 个元素到 position 开始处；如果剩余个数小于插入个数，a)unitialized_fill_n 多余的元素；b) 将 position-old-finish 位置的元素 unitialized_copy position - old_finish 到 finish 开始处，c)fill postion-old_finish。2) 如果内存不够，则按照 vector 内存分配进行操作。因此时间复杂度为O(N)。

vector iterator 实际就是value_type*，是一个 random access iterator。

3.2 list

list 是一个双向链表，有一个重要的性质，就是插入和删除不会影响除了被删除的迭代器外其余的迭代器仍然有效。同时 list 是一个环双向链表，有一个空白 end()，可以通过一个指针按照同一方向遍历玩整个链表。

list 是一个 bidirectional_iterator;

list 是动态申请内存，因此不存在 vector 的扩容问题，但是也失去了随机访问的能力。

3.3 deque

deque 也是动态申请内存，但是与 list 不同，他的迭代器是一个个 random access iterator。

deque 内存分配，如图3所示。

deque 的内存组织，分为两部分，一个中控，为 value_type **，其余是数据存储部分，buffer_size 一般固定，默认 512。随机访问迭代器是通过计算的方式实现的，而不是向 vector 那种天生支持算术运算。

deque 的 map 分配策略，由于 deque 两端都能够插入或者删除，map 分配出现在插入的情况，有一种极端情况就是频繁只在一段插入数据，这会导致另一段的 map 不怎么使用，这种情况如果频繁插入一端 map 没空间了，则重新在原 map 上调整起始位置，移动 map 映射的位置。map_size > 2*new_colum_num，就会搬运映射而不是重新分配空间。如果不满足这个条件，则重新分配一块风

大的空间 new_map_size = map_size + max(map_size,nodes_to_add) + 2，然后将原来的映射拷贝到新 map 的中间位置。

deque 在插入和 erase 的时候，移动元素会判断前面和后面元素的多少，移动元素少的方向。

3.4 stack

stack 是默认使用 deque 作为底层存储容器，去掉了 deque 的前端操作，只保留了尾部的操作。

stack 没有迭代器。

stack 能够手动指定底层容器。

3.5 queue

queue 默认使用 deque 作为底层容器，而不使用 deque 的 push_front 和 pop_back 函数调用，形成 FIFO。

queue 没有迭代器。

queue 能够手动指定底层容器。

3.6 priority_queue

priority_queue 底层使用大/小顶堆作为数据处理方式，默认容器为 vector

priorty_queue 也不提供迭代器。

3.7 slist

slist 是一个单向链表，只能在头部插入，即 push_front，因此只提供单项迭代器 operator++。

4 关联式容器

4.1 RB Tree

red-black tree 的定义：是平衡二叉搜索树的一种，需要满足以下条件：

1、节点只有 red 和 black

2、根结点是 black

3、如果节点是 red，则子节点一定是 black

4、任一节点到 NULL（树端点）的任何路径，black 节点数量相等

4.2 set

底层容器是红黑树，自动排序，key 为 const 不可修改。

set 由于底层采用的是红黑树，因此删除迭代器不会影响其他迭代器，除了被删除的迭代器。

set 使用红黑树的 insert_unique 接口，因此不允许有重复键值。

4.3 map

map 底层同样是红黑树，元素为 pair，键值不可修改，但是 second 值可以修改

map 由于底层采用的是红黑树，因此删除迭代器不会影响其他迭代器，除了被删除的迭代器。

map 使用红黑树的 insert_unique 接口，因此不允许有重复键值。

4.4 multiset

multiset 使用红黑树的 insert_equal 接口，允许相同的键值存在，其余特性与 set 相同。

4.5 multimap

multimap 使用红黑树的 insert_equal 接口，允许相同的键值存在，其余特性与 map 相同。

4.6 hash_table

STL hash_table 采用的方式是桶 + 开链方式。

虽然开练法不需要要求 table 大小为质数，但是 STL 中还是将表设置为质数大小。

```
static const unsigned long __stl_prime_list[__stl_num_primes] =
{
  53ul,          97ul,          193ul,         389ul,         769ul,
  1543ul,        3079ul,        6151ul,        12289ul,       24593ul,
  49157ul,       98317ul,       196613ul,      393241ul,      786433ul,
  1572869ul,     3145739ul,     6291469ul,     12582917ul,    25165843ul,
  50331653ul,    100663319ul,   201326611ul,   402653189ul,   805306457ul,
  1610612741ul,  3221225473ul,  4294967291ul
};

```

4.6.1 hash_table 内存分配策略

STL 中 hash_table 内存配置，backet 使用的 vector 容器，开链没有使用 list 或者 deque 或者 slist，而是自维护一个 single_forward_link。

hash_table 扩桶原则：

```
  const size_type __old_n = _M_buckets.size();
  if (__num_elements_hint > __old_n) {
    const size_type __n = _M_next_size(__num_elements_hint);
      if (__n > __old_n) {
       vector<_Node*, _All> __tmp(__n, (_Node*)(0),
       _M_buckets.get_allocator());
       __STL_TRY {
       for (size_type __bucket = 0; __bucket < __old_n; ++__bucket) {
       _Node* __first = _M_buckets[__bucket];
       while (__first) {
         size_type __new_bucket = _M_bkt_num(__first->_M_val, __n);
          _M_buckets[__bucket] = __first->_M_next;
          __first->_M_next = __tmp[__new_bucket];
         __tmp[__new_bucket] = __first;
```

```
15            __first = _M_buckets[__bucket];
16          }
17        }
18        _M_buckets.swap(__tmp);
19      }
20
```

判断 hash_table 中存在的元素个数 +1 是否大于桶的数量，如果大于则扩容。即判断装载因子是否大于 1。分配新的桶后，将以前的元素逐一计算新桶的位置，然后逐一插入。

在释放旧桶的时候使用了一个trick，申请一个局部桶，将元素插入到新桶后，利用 vector 自带的 swap 交换两个桶，故离开函数后，旧的桶自动释放。

hast_table 插入数据，先找到对应的桶，同时遍历桶中的节点，插入存在两种情况，一不允许插入相同键值（insert_unique），如果存在相同 key 的，则返回 pair<iterator(cur,this),bool>。否则新建节点头插法插入到桶中，返回 pair<iterator(new_node,this),true>；二是允许插入相同键值 (hash_equal，LLVM 中提供的是 insert_multi)，同样，首先找桶，然后遍历桶中的元素，如果存在相同的键值，则立马插入，并返回指向 new_node 的迭代器，否则新建节点，头插法插入到对应的桶中。这两个接口在后续分别得到 hash_map,hash_set，和 hash_multimap,hash_multiset。

4.7 hash_set

hash_set 使用 hash_table 作为底层数据结构，并调用 insert_unique 接口。

在现在的 C++ 中，hash_set 已经更名为 unorderd_set。

4.8 hash_map

hash_map 底层使用 hash_table 作为底层容器，并调用 insert_unique 接口。

在现在的 C++ 中，hash_set 已经更名为 unorderd_map。

4.9 hash_multiset

hash_multiset 使用 hash_table 作为底层数据结构，并调用 insert_equal 接口。

在现在的 C++ 中，STL 提供 hash 版本的 multiset 更名为 unordered_multiset。

4.10 hash_multimap

hash_multimap 使用 hash_table 作为底层数据结构，并调用 insert_equal 接口。

在现在的 C++ 中，STL 提供 hash 版本的 multimap 更名为 unordered_multimap。

5 some algorithm

在 STL 中，通过迭代器得到容器的元素类型其实是强制转换为指针为T*，然后通过模版传入 T*，最后在函数中使用真正数据类型 T 来得到类型变量。

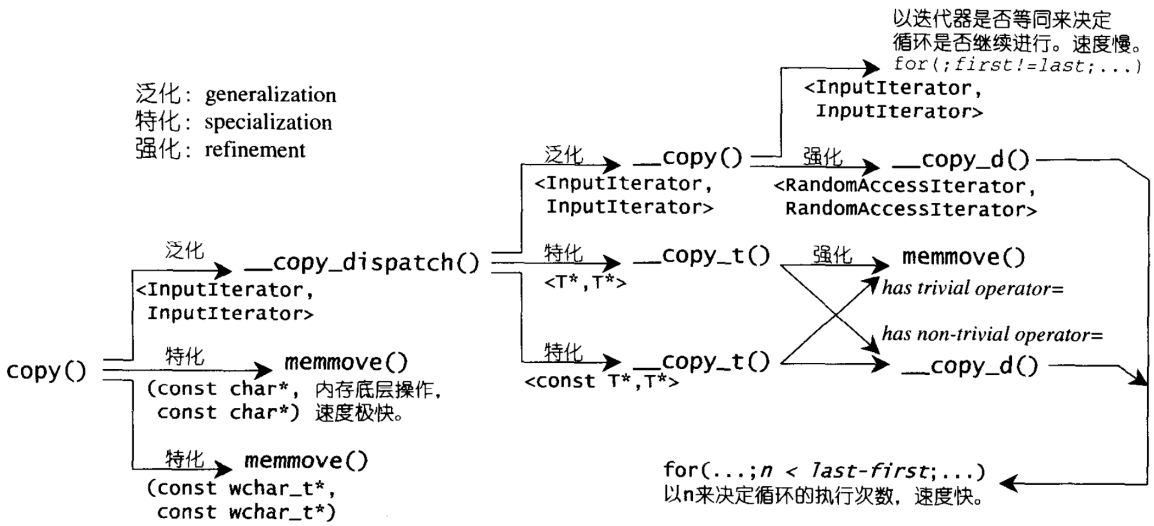

图 4: copy

```
inline typename iterator_traits<_Iter>::value_type*__value_type(const _Iter&){
  return static_cast<typename iterator_traits<_Iter>::value_type*>(0);
}


template <class _ForwardIter1, class _ForwardIter2, class _Tp>
inline void __iter_swap(_ForwardIter1 __a, _ForwardIter2 __b, _Tp*) {
  _Tp __tmp = *__a;
  *__a = *__b;
  *__b = __tmp;
}

template <class _ForwardIter1, class _ForwardIter2>
inline void iter_swap(_ForwardIter1 __a, _ForwardIter2 __b) {
  __STL_REQUIRES(_ForwardIter1, _Mutable_ForwardIterator);
  __STL_REQUIRES(_ForwardIter2, _Mutable_ForwardIterator);
  __STL_CONVERTIBLE(typename iterator_traits<_ForwardIter1>::value_type,
  typename iterator_traits<_ForwardIter2>::value_type);
  __STL_CONVERTIBLE(typename iterator_traits<_ForwardIter2>::value_type,
  typename iterator_traits<_ForwardIter1>::value_type);
  __iter_swap(__a, __b, __VALUE_TYPE(__a));
}

```

5.1 copy

copy 函数为了优化效率无所不用其极端，copy 写了很多特化版本有的直接调用 memmove，有的通过赋值运算符，如图4。

6 adapter

配接器的直观解释：扮演转换器的角色，使得原本接口不兼容而不能一起合作的 classes，能够一起运作。

参考文献

[1] STL 源码剖析，侯捷等