# 弄明白“traits编程技法” 即 萃取

# 内存处理基本工具（ <stl\_uninitialized.h> ）

1. [\_\_first , \_\_last)的值，复制给 \_\_result
2. inline \_ForwardIter \_\_uninitialized\_copy\_aux(\_InputIter \_\_first, \_InputIter \_\_last, \_ForwardIter \_\_result, \_\_true\_type) ;
3. \_ForwardIter \_\_uninitialized\_copy\_aux(\_InputIter \_\_first, \_InputIter \_\_last, \_ForwardIter \_\_result, \_\_false\_type);
4. inline \_ForwardIter \_\_uninitialized\_copy (\_InputIter \_\_first , \_InputIter \_\_last,\_ForwardIter \_\_result, \_Tp\*);
5. inline \_ForwardIter uninitialized\_copy(\_InputIter \_\_first, \_InputIter \_\_last, \_ForwardIter \_\_result)
6. inline char\* uninitialized\_copy(const char\* \_\_first, const char\* \_\_last, char\* \_\_result)
7. inline wchar\_t\* uninitialized\_copy(const wchar\_t\* \_\_first, const wchar\_t\* \_\_last,wchar\_t\* \_\_result
8. [ \_\_first , \_\_first+\_\_count )的值，复制给 \_\_result
9. pair<\_InputIter, \_ForwardIter> \_\_uninitialized\_copy\_n(\_InputIter \_\_first, \_Size \_\_count, \_ForwardIter \_\_result, input\_iterator\_tag);
10. inline pair<\_RandomAccessIter, \_ForwardIter> \_\_uninitialized\_copy\_n(\_RandomAccessIter \_\_first, \_Size \_\_count, \_ForwardIter \_\_result, random\_access\_iterator\_tag);
11. inline pair<\_InputIter, \_ForwardIter> \_\_uninitialized\_copy\_n(\_InputIter \_\_first, \_Size \_\_count, \_ForwardIter \_\_result);
12. 将[\_\_first , \_\_last )的空间，全部填充成\_\_x
13. inline void \_\_uninitialized\_fill\_aux( \_ForwardIter \_\_first, \_ForwardIter \_\_last , const \_Tp& \_\_x, \_\_true\_type);
14. void \_\_uninitialized\_fill\_aux(\_ForwardIter \_\_first, \_ForwardIter \_\_last, const \_Tp& \_\_x, \_\_false\_type);
15. inline void \_\_uninitialized\_fill(\_ForwardIter \_\_first , \_ForwardIter \_\_last , const \_Tp& \_\_x, \_Tp1\*);
16. inline void uninitialized\_fill(\_ForwardIter \_\_first , \_ForwardIter \_\_last , const \_Tp& \_\_x);
17. [\_\_first , \_\_last)的值 ，复制给[ result , result + (last1 - first1) ) ；

[first2, last2) 的值，复制给 [ result , result + (last1 - first1) + (last2 - first2) )

1. inline \_ForwardIter \_\_uninitialized\_copy\_copy(\_InputIter1 \_\_first1 , \_InputIter1 \_\_last1 , \_InputIter2 \_\_first2, \_InputIter2 \_\_last2 , \_ForwardIter \_\_result)
2. [\_\_result, \_\_mid)区间 ，全部填充为\_\_x ；[\_\_first , \_\_last)的值 ，复制给 [ mid , mid+(\_\_last - \_\_first) ]
3. inline \_ForwardIter \_\_uninitialized\_fill\_copy(\_ForwardIter \_\_result, \_ForwardIter \_\_mid , const \_Tp& \_\_x , \_InputIter \_\_first , \_InputIter \_\_last)
4. [\_\_first1, \_\_last1)区间 ，复制给 [\_\_first2, \_\_first2+(\_\_last1 - \_\_first1) ] ；

[first2 + (last1 - first1), last2)区间， 填充为 x ；

1. inline void \_\_uninitialized\_copy\_fill(\_InputIter \_\_first1, \_InputIter \_\_last1,\_ForwardIter \_\_first2, \_ForwardIter \_\_last2 , const \_Tp& \_\_x)

# 对象的构造与析构（ < stl\_construct.h> ）

1. 对象构造：定位new表达式，new \_\_T1() 使用的是\_\_p指向的地址；
2. inline void \_Construct(\_T1\* \_\_p, const \_T2& \_\_value);
3. inline void \_Construct(\_T1\* \_\_p);
4. 对象析构：
   * 1. inline void \_Destroy(\_Tp\* \_\_pointer);

//析构 \_\_pointer 指向的对象

* + 1. void \_\_destroy\_aux(\_ForwardIterator \_\_first, \_ForwardIterator \_\_last, \_\_false\_type) ;

//析构[first,last)区间内所有的对象；

* + 1. inline void \_\_destroy\_aux(\_ForwardIterator, \_ForwardIterator, \_\_true\_type) {}
    2. inline void \_\_destroy(\_ForwardIterator \_\_first, \_ForwardIterator \_\_last, \_Tp\*);
    3. inline void \_Destroy(\_ForwardIterator \_\_first, \_ForwardIterator \_\_last);
    4. inline void \_Destroy(char\*, char\*){}
    5. inline void \_Destroy(int\*, int\*){}
    6. inline void \_Destroy(long\*, long\*){}
    7. inline void \_Destroy(float\*, float\*){}
    8. inline void \_Destroy(double\*, double\*){}
    9. inline void \_Destroy(wchar\_t\*, wchar\_t\*){}

# 空间配置器（ <stl\_alloc.h> ）

1. 空间配置器属性:
2. allocator::value\_type
3. allocator::pointer
4. allocator::const\_pointer
5. allocator::reference
6. allocator::const\_reference
7. allocator::size\_type
8. allocator::difference\_type
9. allocator::rebind
10. 空间配置器的接口：
11. allocator::allocator();
12. allocator::allocator(const allocator&)
13. template<class U> allocator::allocator(const allocator<U>&)
14. allocator::~allocator()
15. pointer allocator::address(reference x) ;

//返回某个对象的地址, a.address(x) 等同于 &x

1. const\_pointer allocator::address(const reference x) const

//返回某个const对象的地址

1. pointer allocator::allocate(size\_type n , const void\* = 0)

//配置空间，足以存储n个T对象；

//第二个参数是个提示，实现上可能会利用它来增加区域性，或完全忽略之；

1. pointer allocator::deallocate(pointer p , size\_type n )

//归还先前配置的空间

1. size\_type allocator::max\_size() const

//返回可成功配置的最大量

1. void allocator::construct(pointer p , const T& x)

//等同于 new ( (const void \*) p ) T(x);

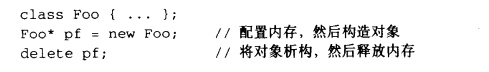
1. void allocator::destroy(pointer p)

//等同于 p->~T();

1. SGI标准空间配置器（std::allocator）

SGI从未使用，主要是因为效率不佳，只是把C++ ::operator new 和 ::operator delete做了一层简单包装；

1. SGI特殊空间配置器（std::alloc）



new操作内含两阶段操作：

* 1. ::operator new 配置内存 ；
  2. ::Foo::Foo() 构造对象；

delete操作内涵两段操作：

* 1. Foo::~Foo() 析构对象；
  2. ::operator delete 释放内存；

1. 为精密分工，STL alloc 决定将两段操作区分开来:

内存配置操作 -- alloc::allocate()；

内存释放操作 -- alloc::deallocate();

对象构造操作 -- ::construct(); 详情见< stl\_construct.h>

对象析构操作 -- ::destroy(); 详情见< stl\_construct.h>

1. 对象构造前的空间配置及对象析构后的空间释放由<stl\_alloc.h>负责，SGI对此设计哲学如下：
   * + 1. 向system heap要求空间；
       2. 考虑多线程状态；
       3. 考虑内存不足时应变策略；
       4. 考虑过多“小型区块”可能造成的内存碎片（fragment）问题；

STL如何应对小型区块可能造成的内存碎片问题？

SGI设计了双层级配置器：

第一级配置器：直接使用malloc 和 free ;

第二级配置器：采用不同策略；

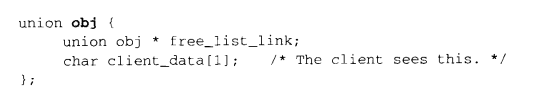
策略（次层配置策略）如下：

1. 区块（>128bytes）时，移交给“一级配置器”处理；
2. 区块（<=128bytes）时，以 "内存池(memory pool)" 方式管理；

具体实施方式：

* + - * 1. 每次配置一大块内存，并维护对应之自由链表（free-list）；
        2. 下次若再有相同大小的内存需求，就直接从free-list中拨出。
        3. 如果客端释还小额区块，就由配置器回收到free-list中；

为了方便管理，SGI第二级配置器主动将任何小额区块的内存需求量上调至8的倍数；（例如：客端要求30 bytes , 就自动调整为32 bytes）,并维护16个free-list( union obj free\_list[16] )，各自管理大小分别为：8，16，24，32，40，48，56，64，72，80，88，96，104，112，120，128byte的小额区块；



Union使得SGI STL不会为了维护链表所必须的指针而造成另一种内存浪费；

STL如何考虑内存不足时的应变策略？

第一级配置器：实现类似C++ new handler的机制 ，因为第一级配置器并非使用::operator new来配置内存，所以不能直接运用C++ new handler机制，该机制成为out-of-memory处理机制；

New handler机制：要求系统在内存配置需求无法被满足时，在抛出std::bad\_alloc异常状态之前，调用一个客端指定的函数（处理例程）；new handler解决内存不足做法有特定模式；

SGI第一级配置器的allocate() 和 realloc()都是在调用malloc() 和 realloc()不成功后，改调用oom\_malloc()和oom\_realloc()。后两者内部都有内循环，不断调用“内存不足处理例程”

设计“New handle内存不足处理例程”是客端的责任，设定“New handle内存不足处理例程”也是客端的责任；

1. STL整个设计开放“第几级配置器”，取决于\_\_USE\_MALLOC宏；

#if \_\_USE\_MALLOC //第一级配置器

typedef \_\_malloc\_alloc\_template<0> malloc\_alloc;

typedef malloc\_alloc alloc;

#else //第二级配置器

Typedef \_\_default\_alloc\_template<\_\_NODE\_ALLOCATOR\_THREADS,0> alloc;

#endif

1. STL包装了一个接口，使配置器接口能够符合STL规格class simple\_alloc ,STL容器全部使用这个接口；
2. 空间配置函数(allocate)/空间释放函数（deallocate）/重新填充free\_list（refill）
3. 内存池（memory pool）

从内存池取空间给free list 使用，是chunk\_alloc()的工作；

