（1）构造和析构的基本工具

内存分配之后，接下来就是对内存的初始化，C++中即使用构造函数来初始化，析构函数来销毁对象。创建对象的时候，C++编译器会自动的调用构造函数，销毁对象时会自动调用析构函数。即：C++创建和销毁对象的过程如下：

创建对象：

分配内存；

调用构造函数初始化内存块；

销毁对象：

调用析构函数；

释放内存；

如果STL直接使用操作符new和delete来分配和释放内存则没有问题，因为它们会分别做两步，但是为了效率，SGI的STL实现并未使用C++的操作符new和delete，而是自己去管理内存，将分配内存和创建对象、销毁对象的两步操作分别分开。内存分配和释放由alloc类来管理；对象的构造和析构由::construct和 ::deatroy来负责。

问题来了？C++编译器不是会自动调用构造和析构函数吗？为什么STL的容器要自己调用呢？

自动调用指的是对于继承体系而言的，又不是对于容器而言的。换句话说，vector和它将要存放的东西没有任何继承上的关系。因此：STL的容器要自己调用它存放的对象的构造和析构函数。

SGI STL负责构造和析构对象内容的文件是：stl_construct.h，该文件内容如下：

```
#include <new.h>

__STL_BEGIN_NAMESPACE

//销毁类T的对象，函数模版
template <class T>
inline void destroy(T* pointer)
{
    pointer->~T();
}
//用T2的实例构造T1对象，函数模版，使用placement new进行内存放置
template <class T1, class T2>
inline void construct(T1* p, const T2& value)
{
    new (p) T1(value);
}
//销毁一个范围内的所有对象，函数模版
template <class ForwardIterator>
inline void
    __destroy_aux(ForwardIterator first, ForwardIterator last, __false_type)
{
    for ( ; first < last; ++first)
        destroy(&*first);
}

template <class ForwardIterator>
inline void __destroy_aux(ForwardIterator, ForwardIterator, __true_type)
{
```

```
}

//销毁一个范围内的类型为T的所有对象。
//如果__type_traits<T>::has_trivial_destructor的结果为false类型，
//表明T需要显式调用析构函数
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());
}

//destroy调用__destroy，__destroy调用__destroy_aux，__destroy_aux循环调用destroy
template <class ForwardIterator>
inline void destroy(ForwardIterator first, ForwardIterator last)
{
    __destroy(first, last, value_type(first));
}

inline void destroy(char*, char*)
{}
inline void destroy(wchar_t*, wchar_t*)
{}

__STL_END_NAMESPACE
```

为了更加清楚的理解上述代码，需要理解几个概念：
POD类型，trivial类型的构造函数、析构函数、赋值操作符、拷贝构造函数。这五个是一体的。trivial的意思是无关紧要的，就是说在这些函数中不会出现资源分配，也就是说，对于这些类的对象在销毁时析构函数什么也不做，例如不需要关闭socket、释放动态内存、解锁、或者关闭数据库连接等。只要将对象占有的内存释放即可，因此，本质上，可以不调用析构函数，而直接释放内存。SGI STL正是基于此进行了优化，例如对于destroy(ForwardIterator first, ForwardIterator last)，如果这个迭代器范围非常大，那么调用无关痛痒的析构函数也会带来很大的开销，为此，对于这种情况进行了特化，什么也不做，直接释放内存即可。以上编程技巧还使用了type_traits编程技巧（后边介绍）。分析了专门负责对象构造与析构的函数后接着分析批量处理内存值的工具。这些函数被实现在stl_uninitialized.h中。
SGI STL定义了四个全局函数作用于未初始化的空间上，定义了一个全局函数作用于已经初始化的内存空间上。这样的功能对于实现容器非常有帮助。分别如下：
construct、uninitialized_copy、uninitialized_fill、uninitialized_fill_n和destroy。construct和destroy的实现已经分析。接着分析另外三个的实现：
1、uninitialized_copy
该函数模版原型如下：

```
template <class InputIterator, class ForwardIterator>
inline ForwardIterator uninitialized_copy(InputIterator first,
       InputIterator last,ForwardIterator result)
```

uninitialized_copy()使我们能够将将内存的分配与对象的构造行为分离开来。该函数会对 [first, last

) 之间的每个对象调用拷贝构造函数。来初始化result指向的内存。注意该函数不会去检查内存是否越界，也就是说，必须确保 [first, last) 是有效的，同时确保result指向的内存空间足够大。这个函数对于实现容器很有帮助。因为容器的全区间构造函数通常分为两个步骤：

- 分配内存，足以包含范围内的所有元素；
- 使用该函数，在该内存上构造元素。
  C++要求该函数具有类似于事务的概念“要么构造出所有元素，要么不构造任何东西”。

2、uninitialized_fill

该函数模版原型如下：

```
template <class ForwardIterator, class T>
inline void uninitialized_fill(ForwardIterator first,
                               ForwardIterator last,  const T& x)
```

uninitialized_fill()与uninitialized_copy类似，只不过输出结果为 [first, last) ，即产生x的复制品。

C++要求该函数具有类似于事务的概念“要么构造出所有元素，要么不构造任何东西”。

3、uninitialized_fill_n

该函数模版原型如下：

```
template <class ForwardIterator, class Size, class T>
inline ForwardIterator uninitialized_fill_n(ForwardIterator first,
                                            Size n, const T& x)
```

uninitialized_fill_n与uninitialized_fill类似，只不过输出范围不是迭代器，而是用n表述要复制x的次数n。

C++要求该函数具有类似于事务的概念“要么构造出所有元素，要么不构造任何东西”。

注意：不管输入或者输出，这三个函数都不会检查内存是否越界，内存的正确分配由客户端(调用者)来保证。

接下来分析这三个函数的具体实现。

这三个函数在实现时用到了type_traits都编程技法。这些技法在平时的开发中可能用不到，但是在实现这些库的时候就用到了，应该细细体会。

- uninitialized_copy

该函数模版如下：

```
template <class InputIterator, class ForwardIterator>
inline ForwardIterator uninitialized_copy(InputIterator first,
          InputIterator last, ForwardIterator result)
{
    return __uninitialized_copy(first, last, result, value_type(result));
}
```

该函数取出结果迭代器的值的类型，该值的类型是一个指向某种类型的指针。然后调用函数模版__uninitialized_copy，__uninitialized_copy的实现如下：

```
//拷贝两个迭代器之间的对象到一个内存块
//POD类型，其赋值操作符和拷贝构造函数一样，则调用该函数模版
template <class InputIterator, class ForwardIterator>
```

```
inline ForwardIterator __uninitialized_copy_aux(InputIterator first,
           InputIterator last,ForwardIterator result,__true_type)
{
    return copy(first, last, result);
}

//非POD类型，调用该函数模版
template <class InputIterator, class ForwardIterator>
ForwardIterator __uninitialized_copy_aux(InputIterator first,
         InputIterator last, ForwardIterator result,__false_type)
{
    ForwardIterator cur = result;
    __STL_TRY
    {
        for ( ; first != last; ++first, ++cur)
        {
            construct(&*cur, *first);
        }
        return cur;
    }
    __STL_UNWIND(destroy(result, cur));
}

//该函数先萃取出迭代器指向的对象的POD特征，然后调用对应的函数，来确定是循环调用构造
//函数呢还是简单的赋值（对于POD类型）
template <class InputIterator, class ForwardIterator, class T>
inline ForwardIterator __uninitialized_copy(InputIterator first,
       InputIterator last,ForwardIterator result, T*)
{
    typedef typename __type_traits<T>::is_POD_type is_POD;
    return __uninitialized_copy_aux(first, last, result, is_POD());
}
```

为了追求性能上的极致，对于char类型的构造，直接调用最底层的内存拷贝函数：

```
inline char* uninitialized_copy(const char* first, const char* last,char* result)
{
    memmove(result, first, last - first);
    return result + (last - first);
}

inline wchar_t* uninitialized_copy(const wchar_t* first, const wchar_t* last,
                                   wchar_t* result)
{
    memmove(result, first, sizeof(wchar_t) * (last - first));
    return result + (last - first);
}
```

- uninitialized_fill

该函数模版如下：

```
//用类型为T的对象实例填充两个迭代器之间的内存块
template <class ForwardIterator, class T>
```

```
inline void __uninitialized_fill_aux(ForwardIterator first,
                    ForwardIterator last,  const T& x, __true_type)
{
    fill(first, last, x);
}

template <class ForwardIterator, class T>
void __uninitialized_fill_aux(ForwardIterator first,
                  ForwardIterator last,  const T& x, __false_type)
{
    ForwardIterator cur = first;
    __STL_TRY
    {
        for ( ; cur != last; ++cur)
        {
            construct(&*cur, x);
        }
    }
    __STL_UNWIND(destroy(first, cur));
}

template <class ForwardIterator, class T, class T1>
inline void __uninitialized_fill(ForwardIterator first,
                             ForwardIterator last,  const T& x, T1*)
{
    typedef typename __type_traits<T1>::is_POD_type is_POD;
    __uninitialized_fill_aux(first, last, x, is_POD());

}

template <class ForwardIterator, class T>
inline void uninitialized_fill(ForwardIterator first,
                           ForwardIterator last,  const T& x)
{
    __uninitialized_fill(first, last, x, value_type(first));
}
```

其实现思维和上边的uninitialized_copy是一样的，不再深入分析了。

- uninitialized_fill_n

该函数模版如下：

```
template <class ForwardIterator, class Size, class T>
inline ForwardIterator __uninitialized_fill_n_aux(
        ForwardIterator first, Size n, const T& x, __true_type)
{
    return fill_n(first, n, x);
}

template <class ForwardIterator, class Size, class T>
ForwardIterator __uninitialized_fill_n_aux(ForwardIterator first,
                                   Size n,const T& x, __false_type)
{
    ForwardIterator cur = first;
```

```
    __STL_TRY {
        for ( ; n > 0; --n, ++cur)
        {
            construct(&*cur, x);
        }
        return cur;
    }
    __STL_UNWIND(destroy(first, cur));
}

template <class ForwardIterator, class Size, class T, class T1>
inline ForwardIterator __uninitialized_fill_n(ForwardIterator
                                first, Size n, const T& x, T1*)
{
    typedef typename __type_traits<T1>::is_POD_type is_POD;
    return __uninitialized_fill_n_aux(first, n, x, is_POD());

}

template <class ForwardIterator, class Size, class T>
inline ForwardIterator uninitialized_fill_n(ForwardIterator first,
                                            Size n, const T& x)
{
    return __uninitialized_fill_n(first, n, x, value_type(first));
}
```

其实现思维和上边的uninitialized_copy也是一样的，不再深入分析了。
接着看SGI STL提供的其他几个函数模版，这几个只不过是对上边三个简单的封装。

```
//*************************************************************************//
//以下不过是上边函数的封装。
// Copies [first1, last1) into [result, result + (last1 - first1)), and
//  copies [first2, last2) into
//  [result, result + (last1 - first1) + (last2 - first2)).

template <class InputIterator1, class InputIterator2, class ForwardIterator>
inline ForwardIterator __uninitialized_copy_copy(InputIterator1 first1,
                    InputIterator1 last1, InputIterator2 first2,
                    InputIterator2 last2, ForwardIterator result)
{
    ForwardIterator mid = uninitialized_copy(first1, last1, result);
    __STL_TRY
    {
        return uninitialized_copy(first2, last2, mid);
    }
    __STL_UNWIND(destroy(result, mid));
}

// Fills [result, mid) with x, and copies [first, last) into
//  [mid, mid + (last - first)).
template <class ForwardIterator, class T, class InputIterator>
inline ForwardIterator __uninitialized_fill_copy(ForwardIterator result,
                                    ForwardIterator mid,const T& x,
```

```
                                InputIterator first, InputIterator last)
{
    uninitialized_fill(result, mid, x);
    __STL_TRY
    {
        return uninitialized_copy(first, last, mid);
    }
    __STL_UNWIND(destroy(result, mid));
}

// Copies [first1, last1) into [first2, first2 + (last1 - first1)), and
//  fills [first2 + (last1 - first1), last2) with x.
template <class InputIterator, class ForwardIterator, class T>
inline void __uninitialized_copy_fill(InputIterator first1,
   InputIterator last1, ForwardIterator first2, ForwardIterator last2,const T& x)
{
    ForwardIterator mid2 = uninitialized_copy(first1, last1, first2);
    __STL_TRY
    {
        uninitialized_fill(mid2, last2, x);
    }
    __STL_UNWIND(destroy(first2, mid2));
}
//*************************************************************************//
```

接着看看SGI STL特有的函数模版：uninitialized_copy_n

```
//*************************************************************************//
//以下函数拷贝一个first迭代器指向的对象列表(只拷贝count个，该对象列表的长度
//必须>=count)到一个result指向的内存块返回两个迭代器值。
template <class InputIterator, class Size, class ForwardIterator>
pair<InputIterator, ForwardIterator> __uninitialized_copy_n(
   InputIterator first, Size count,ForwardIterator result,input_iterator_tag)
{
    ForwardIterator cur = result;
    __STL_TRY
    {
        for ( ; count > 0 ; --count, ++first, ++cur)
        {
            construct(&*cur, *first);
        }
        return pair<InputIterator, ForwardIterator>(first, cur);
    }
    __STL_UNWIND(destroy(result, cur));
}
template <class RandomAccessIterator, class Size, class ForwardIterator>
inline pair<RandomAccessIterator, ForwardIterator> __uninitialized_copy_n
(RandomAccessIterator first, Size count,ForwardIterator result,
  random_access_iterator_tag)
{
    RandomAccessIterator last = first + count;
    return make_pair(last, uninitialized_copy(first, last, result));
}
```

```
template <class InputIterator, class Size, class ForwardIterator>
inline pair<InputIterator, ForwardIterator> uninitialized_copy_n(
    InputIterator first, Size count,ForwardIterator result)
{
    return __uninitialized_copy_n(first, count, result,iterator_category(first));
}
//*****************************************************************//
```

作者：许富博