[**从零开始学C++之STL（三）：迭代器类vector::iterator 和 vector::reverse\_iterator 的实现、迭代器类型、常用的容器成员**](http://blog.csdn.net/jnu_simba/article/details/9454787)

标签： [迭代器](http://www.csdn.net/tag/%e8%bf%ad%e4%bb%a3%e5%99%a8)[常用的容器成员](http://www.csdn.net/tag/%e5%b8%b8%e7%94%a8%e7%9a%84%e5%ae%b9%e5%99%a8%e6%88%90%e5%91%98)

2013-07-24 20:18 10310人阅读 [评论](http://blog.csdn.net/jnu_simba/article/details/9454787#comments)(2) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/jnu_simba/article/details/9454787#report)

http://static.blog.csdn.net/images/category_icon.jpg分类：

从零开始学C++*（51）* http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

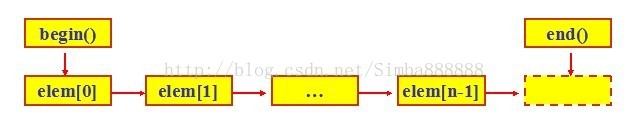
一、迭代器

迭代器是泛型指针

普通指针可以指向内存中的一个地址

迭代器可以指向容器中的一个位置

STL的每一个容器类模版中，都定义了一组对应的迭代器类。使用迭代器，算法函数可以访问容器中指定位置的元素，而无需关心元素的具体类型。



下面来稍微看一下vector<class>::iterator 和 vector<class>::reverse\_iterator 的源码：

 C++ Code

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 |  | template < class \_Ty,          class \_Alloc > class \_Vector\_val     : public \_CONTAINER\_BASE\_AUX\_ALLOC<\_Alloc> {     // base class for vector to hold allocator \_Alval protected:     \_Vector\_val(\_Alloc \_Al = \_Alloc())         : \_CONTAINER\_BASE\_AUX\_ALLOC<\_Alloc>(\_Al), \_Alval(\_Al)     {         // construct allocator from \_Al     }      typedef typename \_Alloc::template     rebind<\_Ty>::other \_Alty;      \_Alty \_Alval;   // allocator object for values };  template < class \_Ty,          class \_Ax > class vector     : public \_Vector\_val<\_Ty, \_Ax> {     // varying size array of values public:     .....     typedef \_Vector\_val<\_Ty, \_Ax> \_Mybase;      typedef typename \_Mybase::\_Alty \_Alloc;  //\_Alloc 的定义所在     typedef \_Vector\_iterator<\_Ty, \_Alloc> iterator;     typedef \_Vector\_const\_iterator<\_Ty, \_Alloc> const\_iterator;      //  friend class \_Vector\_iterator<\_Ty, \_Alloc>;     friend class \_Vector\_const\_iterator<\_Ty, \_Alloc>;      typedef std::reverse\_iterator<iterator> reverse\_iterator;     typedef std::reverse\_iterator<const\_iterator> const\_reverse\_iterator;     ....... };  template < class \_Ty,          class \_Alloc > class \_Vector\_iterator     : public \_Vector\_const\_iterator<\_Ty, \_Alloc> {     // iterator for mutable vector public:     typedef \_Vector\_iterator<\_Ty, \_Alloc> \_Myt;     typedef \_Vector\_const\_iterator<\_Ty, \_Alloc> \_Mybase;     ....... };  template < class \_Ty,          class \_Alloc > class \_Vector\_const\_iterator     : public \_Ranit < \_Ty, typename \_Alloc::difference\_type,       typename \_Alloc::const\_pointer, typename \_Alloc::const\_reference > {     // iterator for nonmutable vector public:     .....     typedef typename \_Alloc::pointer \_Tptr;     typedef random\_access\_iterator\_tag iterator\_category;    typedef \_Ty value\_type;    typedef typename \_Alloc::difference\_type difference\_type;    typedef typename \_Alloc::const\_pointer pointer;   typedef typename \_Alloc::const\_reference reference;   typedef const value\_type \_FARQ \*const\_pointer;  typedef const value\_type \_FARQ& const\_reference;     ....     \_Tptr \_Myptr;   // offset of element in vector };  template<class \_Ty> class allocator     : public \_Allocator\_base<\_Ty> {     // generic allocator for objects of class \_Ty public:     ......     typedef \_Allocator\_base<\_Ty> \_Mybase;     typedef typename \_Mybase::value\_type value\_type;     typedef value\_type \_FARQ \*pointer;     typedef value\_type \_FARQ& reference;     ... };  // TEMPLATE CLASS \_Allocator\_base template<class \_Ty> struct \_Allocator\_base {     // base class for generic allocators     typedef \_Ty value\_type; };  template<class \_RanIt> class reverse\_iterator     : public \_Revranit<\_RanIt, iterator<...> > {     // wrap iterator to run it backwards     ........     typedef reverse\_iterator<\_RanIt> \_Myt;     typedef \_RanIt iterator\_type;     ..............  };  // TEMPLATE CLASS \_Revranit template < class \_RanIt,          class \_Base > class \_Revranit     : public \_Base {     // wrap iterator to run it backwards     .... protected:     \_RanIt current; // the wrapped iterator }; |

typedef \_Vector\_iterator<\_Ty, \_Alloc> iterator; 可知iterator 只是类型定义，而\_Vector\_iterator 又继承自 \_Vector\_const\_iterator，这

个类有个成员\_Tptr \_Myptr;  进一步看\_Tptr 可以知道类型是value\_type\*， 假设现在使用的容器是vector<int>，那么value\_type 也就是

int， 那么实际上iterator 内部就只有一个int\* \_Myptr; 成员而已。即包装了一般的指针。很明显地，iterator 类里面一定重载了

operator\*, ->, ++, -- 等操作符，而这些操作符实际上还是对一般的指针\_Myptr 进行操作。举operator\* 来说：

 C++ Code

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 |  | // iterator  reference operator\*() const {     // return designated object     return ((reference) \*\*(\_Mybase \*)this); }  // const\_iterator reference operator\*() const {     // return designated object      .....     return (\*\_Myptr); } |

this 在这里是iterator\*， (\_Mybase \*)this就是const\_iterator\*，\*(\_Mybase \*)this就是const\_iterator 对象，\*\*(\_Mybase \*)this就是调用const\_iterator

的 operator\*()， 即 返回\*\_Myptr，即指向的元素，iterator 的 operator\* 返回的是引用 reference ，这其实是在allocator 类中定义的const\_reference，

即 const value\_type&， 假设现在的容器是vector<int> ，那么返回的也就是const int& ，即不能将其作为左值进行赋值，但能作为右值，如 cout<<\*it;

同样地, iterator 的 operator++ 也调用了 const\_iterator 的 operator++， 在函数里面也是执行  ++\_Myptr; 的操作，返回的是const\_iterator& ，而从

iterator 的 operator++ 返回的是iterator& 。

typedef std::reverse\_iterator<iterator> reverse\_iterator; 再来看 reverse\_iterator，继承自\_Revranit， 这个类有个成员 \_RanIt current;

也就是说有个 iterator 类成员，即包装了一个iterator 类成员，从这个角度看，reverse\_iterator 也可以算是一个适配器，利用 iterator

类的一些操作完成自身的功能。

××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××

上面介绍的是vector::iterator ，比如 list::iterator 实现是类似的，内部成员也是一个指针，不过是指向Node 结点的指针，如：

\_Nodeptr \_Ptr;// pointer to node

×××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××

简单的测试程序如下：

 C++ Code

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 |  | #include <vector> #include <list> #include <iostream>  using namespace std;  int main(void) {     vector<int> v;     v.push\_back(1);     v.push\_back(2);     v.push\_back(3);      vector<int>::iterator it;     for (it = v.begin(); it != v.end(); ++it)     {         cout << \*it << ' ';     }     cout << endl;      vector<int>::reverse\_iterator ri;     for (ri = v.rbegin(); ri != v.rend(); ++ri)     {         cout << \*ri << ' ';     }     cout << endl;       list<int> l;     l.push\_back(1);     l.push\_back(2);     l.push\_back(3);      list<int>::iterator it2;      for (it2 = l.begin(); it2 != l.end(); ++it2)     {         cout << \*it2 << ' ';     }     cout << endl;      return 0; } |

二、迭代器的类型 iterator\_category

输入迭代器

可以用来从序列中读取数据

输出迭代器

允许向序列中写入数据

前向迭代器

既是输入迭代器又是输出迭代器，并且可以对序列进行单向的遍历

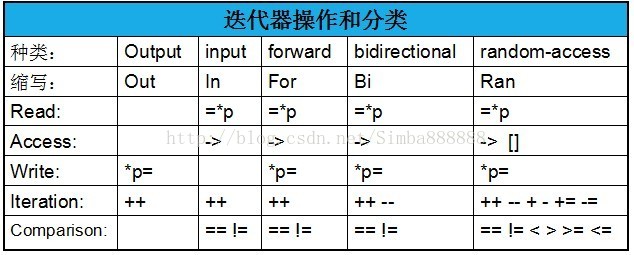
双向迭代器

与前向迭代器相似，但是在两个方向上都可以对数据遍历

随机访问迭代器

也是双向迭代器，但能够在序列中的任意两个位置之间进行跳转

下图是不同类型的迭代器能够实现的操作：



1、The standard-library container classes all support bidirectional iterators.

of curse the const iterator only meets the requirements for input iterators.

2、The vector and deque iterators are random-access iterators. however, the list iterator is not; it supports only bidirectional iterators.etc,it does not support [] operation.

3、If c is a container that supports push\_back, then back\_inserter(c)  is an output iterator that meets no other iterator requirements.

4、The stream iterators are defined in the <iterator> header,  the istream\_iterator<class name> meets the requirements for input iterators; the ostream\_iterator<class name>  meets the requirements for output iterators.

 we can use it like this:

 C++ Code

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 |  | vector<int> v; // read ints from the standard input and append them to v // istream\_iterator<int>() indicate "EOF" copy(istream\_iterator<int>(cin), istream\_iterator<int>(), back\_inserter(v));   // write the elements of v each seperated from other by a space copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " ");       // no seperation between elements!      copy(v.begin(), v.end(), ostream\_iterator<int>(cout); |

不同的迭代器支持不同的操作集，而各种算法也要求相应的迭代器具有最小的操作集。因此，可以将算法的迭代器分为下面五类：



  除了输出迭代器，其他类别的迭代器形成了一个层次结构：需要低级类别迭代器的地方，可使用任意一种更高级的迭代器。例如，对于需要输入迭代器的算法，可传递前向、双向或随机访问迭代器调用该算法。而反之则不行。注意：向算法传递无效的迭代器类别所引起的错误，无法保证会在编译时被捕获到。

   map, set, list类型提供双向迭代器，而string, vector和deque容器上定义的迭代器都是随机访问迭代器，用作访问内置数组元素的指针也是随机访问迭代器。istream\_iterator是输入迭代器，ostream\_iterator是输出迭代器。

   另外，虽然map和set类型提供双向迭代器，但关联容器只能使用这部分算法的一个子集。因为关联容器的键是const对象。因此，关联容器不能使用任何写序列元素的算法。只能使用与关联容器绑在一起的迭代器来提供用于读操作的实参。因此，在处理算法时，最好将关联容器上的迭代器视为支持自减运算的输入迭代器，而不是完整的双向迭代器。

  最后需要注意的是，stack、queue、priority\_queue 都不支持任一种迭代器，它们都是容器适配器类型，stack是用vector/deque/list对象创建了一个先进后出容器，queue是用deque或list对象创建了一个先进先出容器，priority\_queue是用vector/deque创建了一个排序队列。

三、常用的容器成员

下面列举的成员中，有一些是所有容器共有的，有些是特有的，注意区别：













四、迭代器失效的问题（摘自 http://blog.csdn.net/hackbuteer1/article/details/7734382）

vector迭代器的几种失效的情况：   
1、当插入（push\_back）一个元素后，end操作返回的迭代器肯定失效。  
2、当插入(push\_back)一个元素后，capacity返回值与没有插入元素之前相比有改变，则需要重新分配整个容器，此时first和end操作返回的迭代器都会失效。  
3、当进行删除操作（erase，pop\_back）后，指向删除点的迭代器全部失效；指向删除点后面的元素的迭代器也将全部失效。

 deque迭代器的失效情况： 在C++Primer一书中是这样限定的：   
1、在deque容器首部或者尾部插入元素不会使得任何迭代器失效。   
2、在其首部或尾部删除元素则只会使指向被删除元素的迭代器失效。  
3、在deque容器的任何其他位置的插入和删除操作将使指向该容器元素的所有迭代器失效。

 list的迭代器好像很少情况下会失效，也许就只是在删除的时候，指向被删除节点的迭代器会失效吧，其他的还没有发现。  
先看两条规制：  
1、对于节点式容器(map, list, set)元素的删除、插入操作会导致指向该元素的迭代器失效，其他元素迭代器不受影响。  
2、对于顺序式容器(vector)元素的删除、插入操作会导致指向该元素以及后面的元素的迭代器失效。

众所周之当使用一个容器的insert或者erase函数通过迭代器插入或删除元素"可能"会导致迭代器失效，因此建议我们获取insert或者erase返回的迭代器，以便用重新获取新的有效的迭代器进行正确的操作：