

Lesson10---list

【本节目标】

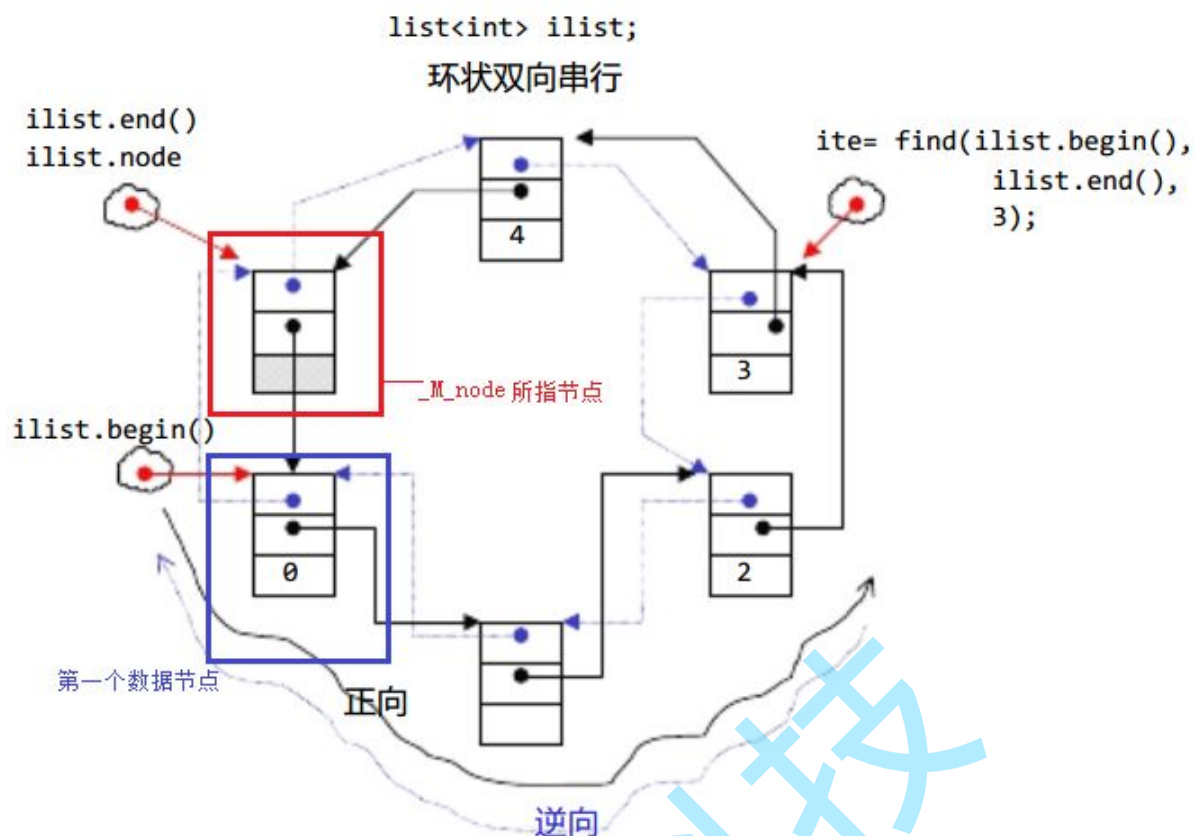
- 1. list的介绍及使用
- 2. list的深度剖析及模拟实现
- 3. list与vector的对比
- 4. 本节作业

1. list的介绍及使用

1.1 list的介绍

[list的文档介绍](#)

1. list是可以在常数范围内在任意位置进行插入和删除的序列式容器，并且该容器可以前后双向迭代。
2. list的底层是双向链表结构，双向链表中每个元素存储在互不相关的独立节点中，在节点中通过指针指向其前一个元素和后一个元素。
3. list与forward_list非常相似：最主要的不同在于forward_list是单链表，只能朝前迭代，已让其更简单高效。
4. 与其他序列式容器相比(array, vector, deque)，list通常在任意位置进行插入、移除元素的执行效率更好。
5. 与其他序列式容器相比，list和forward_list最大的缺陷是不支持任意位置的随机访问，比如：要访问list的第6个元素，必须从已知的位置(比如头部或者尾部)迭代到该位置，在这段位置上迭代需要线性的时间开销；list还需要一些额外的空间，以保存每个节点的相关联信息(对于存储类型较小元素的大list来说这可能是一个重要的因素)



1.2 list的使用

list中的接口比较多，此处类似，只需要掌握如何正确的使用，然后再去深入研究背后的原理，已达到可扩展的能力。以下为list中一些常见的重要接口。

1.2.1 list的构造

构造函数 (constructor)	接口说明
list()	构造空的list
list (size_type n, const value_type& val = value_type())	构造的list中包含n个值为val的元素
list (const list& x)	拷贝构造函数
list (InputIterator first, InputIterator last)	用[first, last)区间中的元素构造list

```

1 // constructing lists
2 #include <iostream>
3 #include <list>
4
5 int main ()
6 {
7     std::list<int> l1; // 构造空的l1
8     std::list<int> l2 (4,100); // l2中放4个值为100的元素
9
10    std::list<int> l3 (l2.begin(), l2.end()); // 用l2的[begin(), end()) 左闭右开的区间构

```

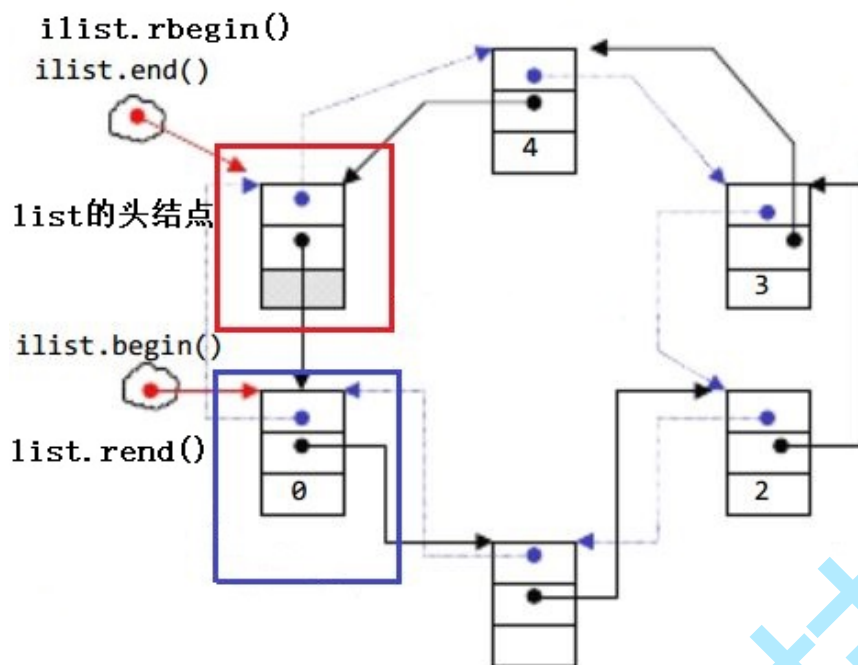
造13

```
10     std::list<int> l4 (13);                                // 用13拷贝构造14
11
12     // 以数组为迭代器区间构造15
13     int array[] = {16,2,77,29};
14     std::list<int> l5 (array, array + sizeof(array) / sizeof(int) );
15
16     // 用迭代器方式打印15中的元素
17     for(std::list<int>::iterator it = l5.begin(); it != l5.end(); it++)
18         std::cout << *it << " ";
19     std::cout<<endl;
20
21     // C++11范围for的方式遍历
22     for(auto& e : l5)
23         std::cout<< e << " ";
24
25     std::cout<<endl;
26     return 0;
27 }
```

1.2.2 list iterator的使用

此处，大家可暂时将迭代器理解成一个指针，该指针指向list中的某个节点。

函数声明	接口说明
<u>begin</u> + <u>end</u>	返回第一个元素的迭代器+返回最后一个元素下一个位置的迭代器
<u>rbegin</u> + <u>rend</u>	返回第一个元素的reverse_iterator,即end位置，返回最后一个元素下一个位置的reverse_iterator,即begin位置



【注意】

1. `begin`与`end`为正向迭代器，对迭代器执行`++`操作，迭代器向后移动
2. `rbegin(end)`与`rend(begin)`为反向迭代器，对迭代器执行`++`操作，迭代器向前移动

```

1  #include <iostream>
2  using namespace std;
3  #include <list>
4
5  void print_list(const list<int>& l)
6  {
7      // 注意这里调用的是list的 begin() const, 返回list的const_iterator对象
8      for (list<int>::const_iterator it = l.begin(); it != l.end(); ++it)
9      {
10         cout << *it << " ";
11         // *it = 10; 编译不通过
12     }
13
14     cout << endl;
15 }
16
17 int main()
18 {
19     int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
20     list<int> l(array, array + sizeof(array) / sizeof(array[0]));
21     // 使用正向迭代器正向list中的元素
22     for (list<int>::iterator it = l.begin(); it != l.end(); ++it)
23
24         cout << *it << " ";

```

```

24     cout << endl;
25
26     // 使用反向迭代器逆向打印list中的元素
27     for (list<int>::reverse_iterator it = l.rbegin(); it != l.rend(); ++it)
28         cout << *it << " ";
29     cout << endl;
30
31     return 0;
32 }

```

1.2.3 list capacity

函数声明	接口说明
<u>empty</u>	检测list是否为空，是返回true，否则返回false
<u>size</u>	返回list中有效节点的个数

1.2.4 list element access

函数声明	接口说明
<u>front</u>	返回list的第一个节点中值的引用
<u>back</u>	返回list的最后一个节点中值的引用

1.2.5 list modifiers

函数声明	接口说明
<u>push front</u>	在list首元素前插入值为val的元素
<u>pop front</u>	删除list中第一个元素
<u>push back</u>	在list尾部插入值为val的元素
<u>pop back</u>	删除list中最后一个元素
<u>insert</u>	在list position 位置中插入值为val的元素
<u>erase</u>	删除list position位置的元素
<u>swap</u>	交换两个list中的元素
<u>clear</u>	清空list中的有效元素

```

1  #include <list>
2
3  void PrintList(list<int>& l)
4  {
5      for (auto& e : l)
6          cout << e << " ";
7      cout << endl;
8  }
9
10 //=====
11 // push_back/pop_back/push_front/pop_front
12 void TestList1()
13 {
14     int array[] = { 1, 2, 3 };
15     list<int> L(array, array+sizeof(array)/sizeof(array[0]));
16
17     // 在list的尾部插入4, 头部插入0
18     L.push_back(4);
19     L.push_front(0);
20     PrintList(L);
21
22     // 删除list尾部节点和头部节点
23     L.pop_back();
24     L.pop_front();
25     PrintList(L);
26 }
27
28 //=====
29 // insert /erase
30 void TestList3()
31 {
32     int array1[] = { 1, 2, 3 };
33     list<int> L(array1, array1+sizeof(array1)/sizeof(array1[0]));
34
35     // 获取链表中第二个节点
36     auto pos = ++L.begin();
37     cout << *pos << endl;
38
39     // 在pos前插入值为4的元素
40     L.insert(pos, 4);
41     PrintList(L);
42
43     // 在pos前插入5个值为5的元素
44     L.insert(pos, 5, 5);
45     PrintList(L);
46
47     // 在pos前插入[v.begin(), v.end)区间中的元素
48     vector<int> v{ 7, 8, 9 };
49     L.insert(pos, v.begin(), v.end());
50     PrintList(L);
51

```

```

52 // 删除pos位置上的元素
53 L.erase(pos);
54 PrintList(L);
55
56 // 删除list中[begin, end)区间中的元素，即删除list中的所有元素
57 L.erase(L.begin(), L.end());
58 PrintList(L);
59 }
60
61 // resize/swap/clear
62 void TestList4()
63 {
64 // 用数组来构造list
65 int array1[] = { 1, 2, 3 };
66 list<int> l1(array1, array1+sizeof(array1)/sizeof(array1[0]));
67 PrintList(l1);
68
69 // 交换l1和l2中的元素
70 l1.swap(l2);
71 PrintList(l1);
72 PrintList(l2);
73
74 // 将l2中的元素清空
75 l2.clear();
76 cout<<l2.size()<<endl;
77 }

```

list中还有一些操作，需要用到时大家可参阅list的文档说明。

1.2.6 list的迭代器失效

前面说过，此处大家可将迭代器暂时理解成类似于指针，**迭代器失效即迭代器所指向的节点的无效，即该节点被删除了**。因为list的底层结构为带头节点的双向循环链表，因此在list中进行插入时是会导致list的迭代器失效的，只有在删除时才会失效，并且失效的只是指向被删除节点的迭代器，其他迭代器不会受到影响。

```

1 void TestListIterator1()
2 {
3     int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
4     list<int> l(array, array+sizeof(array)/sizeof(array[0]));
5
6     auto it = l.begin();
7     while (it != l.end())
8     {
9         // erase()函数执行后，it所指向的节点已被删除，因此it无效，在下一次使用it时，必须先给
其赋值
10        l.erase(it);
11        ++it;
12    }
13 }
14
15 // 改正
16 void TestListIterator()
17 {

```

```

18     int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
19     list<int> l(array, array+sizeof(array)/sizeof(array[0]));
20
21     auto it = l.begin();
22     while (it != l.end())
23     {
24         l.erase(it++);    // it = l.erase(it);
25     }
26 }

```

2. list的模拟实现

2.1 模拟实现list

要模拟实现list，必须要熟悉list的底层结构以及其接口的含义，通过上面的学习，这些内容已基本掌握，现在我们来模拟实现list。

```

1  namespace bite
2  {
3      // List的节点类
4      template<class T>
5      struct ListNode
6      {
7          ListNode(const T& val = T())
8              : _pPre(nullptr)
9              , _pNext(nullptr)
10             , _val(val)
11          {}
12
13          ListNode<T>* _pPre;
14          ListNode<T>* _pNext;
15          T _val;
16      };
17
18      /*
19      List 的迭代器
20      迭代器有两种实现方式，具体应根据容器底层数据结构实现：
21      1. 原生态指针，比如：vector
22      2. 将原生态指针进行封装，因迭代器使用形式与指针完全相同，因此在自定义的类中必须实现以下
        方法：
23          1. 指针可以解引用，迭代器的类中必须重载operator*()
24          2. 指针可以通过->访问其所指空间成员，迭代器类中必须重载operator->()
25          3. 指针可以++向后移动，迭代器类中必须重载operator++()与operator++(int)
26             至于operator--()/operator--(int)释放需要重载，根据具体的结构来抉择，双向链表可
        以向前          移动，所以需要重载，如果是forward_list就不需要重载--
27          4. 迭代器需要进行是否相等的比较，因此还需要重载operator==( )与operator!=( )
28      */
29      template<class T, class Ref, class Ptr>
30      class ListIterator
31      {
32          typedef ListNode<T>* PNode;
33          typedef ListIterator<T, Ref, Ptr> Self;
34      public:

```



```

35     ListIterator(PNode pNode = nullptr)
36         : _pNode(pNode)
37     {}
38
39     ListIterator(const Self& l)
40         : _pNode(l._pNode)
41     {}
42
43     T& operator*(){return _pNode->_val;}
44     T* operator->(){return &(operator*());}
45
46     Self& operator++()
47     {
48         _pNode = _pNode->_pNext;
49         return *this;
50     }
51
52     Self operator++(int)
53     {
54         Self temp(*this);
55         _pNode = _pNode->_pNext;
56         return temp;
57     }
58
59     Self& operator--();
60     Self& operator--(int);
61
62     bool operator!=(const Self& l){return _pNode != l._pNode;}
63     bool operator==(const Self& l){return _pNode == l._pNode;}
64
65     PNode _pNode;
66 };
67
68 template<class T>
69 class list
70 {
71     typedef ListNode<T> Node;
72     typedef Node* PNode;
73
74 public:
75     typedef ListIterator<T, T&, T*> iterator;
76     typedef ListIterator<T, const T&, const T*> const_iterator;
77 public:
78     //////////////////////////////////////////
79     // List的构造
80     list()
81     {
82         CreateHead();
83     }
84
85     list(int n, const T& value = T())
86     {
87         CreateHead();

```

```

88         for (int i = 0; i < n; ++i)
89             push_back(value);
90     }
91
92     template <class Iterator>
93     list(Iterator first, Iterator last)
94     {
95         CreateHead();
96         while (first != last)
97         {
98             push_back(*first);
99             ++first;
100         }
101     }
102
103     list(const list<T>& l)
104     {
105         CreateHead();
106
107         // 用l中的元素构造临时的temp,然后与当前对象交换
108         list<T> temp(l.cbegin(), l.cend());
109         this->swap(temp);
110     }
111
112     list<T>& operator=(const list<T> l)
113     {
114         this->swap(l);
115         return *this;
116     }
117
118     ~list()
119     {
120         clear();
121         delete _pHead;
122         _pHead = nullptr;
123     }
124
125     //////////////////////////////////////
126     // List Iterator
127     iterator begin(){return iterator(_pHead->pNext);}
128     iterator end(){return iterator(_pHead);}
129     const_iterator begin(){return const_iterator(_pHead->pNext);}
130     const_iterator end(){return const_iterator(_pHead);}
131     //////////////////////////////////////
132     // List Capacity
133     size_t size()const;
134     bool empty()const;
135     //////////////////////////////////////
136     // List Access
137     T& front();
138     const T& front()const;
139     T& back();
140     const T& back()const;

```

```

141 ////////////////////////////////////////////////////
142 // List Modify
143 void push_back(const T& val){insert(begin(), val);}
144 void pop_back(){erase(--end());}
145 void push_front(const T& val){insert(begin(), val);}
146 void pop_front(){erase(begin());}
147
148 // 在pos位置前插入值为val的节点
149 iterator insert(iterator pos, const T& val)
150 {
151     PNode pNewNode = new Node(val);
152     PNode pCur = pos._pNode;
153     // 先将新节点插入
154     pNewNode->_pPre = pCur->_pPre;
155     pNewNode->_pNext = pCur;
156     pNewNode->_pPre->_pNext = pNewNode;
157     pCur->_pPre = pNewNode;
158     return iterator(pNewNode);
159 }
160
161 // 删除pos位置的节点，返回该节点的下一个位置
162 iterator erase(iterator pos)
163 {
164     // 找到待删除的节点
165     PNode pDel = pos._pNode;
166     PNode pRet = pDel->_pNext;
167
168     // 将该节点从链表中拆下来并删除
169     pDel->_pPre->_pNext = pDel->_pNext;
170     pDel->_pNext->_pPre = pDel->_pPre;
171     delete pDel;
172
173     return iterator(pRet);
174 }
175
176 void clear();
177 void swap(List<T>& l);
178 private:
179 void CreateHead()
180 {
181     _pHead = new Node;
182     _pHead->_pPre = _pHead;
183     _pHead->_pNext = _pHead;
184 }
185 private:
186 PNode _pHead;
187 };
188 }

```

2.2 对模拟的bite::list进行测试

```

1 // 正向打印链表
2 template<class T>

```

```

3 void PrintList(const bite::list<T>& l)
4 {
5     auto it = l.cbegin();
6     while (it != l.cend())
7     {
8         cout << *it << " ";
9         ++it;
10    }
11
12    cout << endl;
13 }
14
15 // 测试List的构造
16 void TestList1()
17 {
18     bite::list<int> l1;
19     bite::list<int> l2(10, 5);
20     PrintList(l2);
21
22     int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
23     bite::list<int> l3(array, array+sizeof(array)/sizeof(array[0]));
24     PrintList(l3);
25
26     bite::list<int> l4(l3);
27     PrintList(l4);
28
29     l1 = l4;
30     PrintList(l1);
31     PrintListReverse(l1);
32 }
33
34 // PushBack()/PopBack()/PushFront()/PopFront()
35 void TestList2()
36 {
37     // 测试PushBack与PopBack
38     bite::list<int> l;
39     l.push_back(1);
40     l.push_back(2);
41     l.push_back(3);
42     PrintList(l);
43
44     l.pop_back();
45     l.pop_back();
46     PrintList(l);
47
48     l.pop_back();
49     cout << l.size() << endl;
50
51     // 测试PushFront与PopFront
52     l.push_front(1);
53     l.push_front(2);
54     l.push_front(3);
55
56     PrintList(l);

```

```

56
57     l.pop_front();
58     l.pop_front();
59     PrintList(l);
60
61     l.pop_front();
62     cout << l.size() << endl;
63 }
64
65 void TestList3()
66 {
67     int array[] = { 1, 2, 3, 4, 5 };
68     bite::list<int> l(array, array+sizeof(array)/sizeof(array[0]));
69
70     auto pos = l.begin();
71     l.insert(l.begin(), 0);
72     PrintList(l);
73
74     ++pos;
75     l.insert(pos, 2);
76     PrintList(l);
77
78     l.erase(l.begin());
79     l.erase(pos);
80     PrintList(l);
81
82     // pos指向的节点已经被删除, pos迭代器失效
83     cout << *pos << endl;
84
85     auto it = l.begin();
86     while (it != l.end())
87     {
88         it = l.erase(it);
89     }
90     cout << l.size() << endl;
91 }

```

3. list与vector的对比

vector与list都是STL中非常重要的序列式容器，由于两个容器的底层结构不同，导致其特性以及应用场景不同，其主要不同如下：

	vector	list
底层结构	动态顺序表，一段连续空间	带头结点的双向循环链表
随机访问	支持随机访问，访问某个元素效率 $O(1)$	不支持随机访问，访问某个元素效率 $O(N)$
插入和删除	任意位置插入和删除效率低，需要搬移元素，时间复杂度为 $O(N)$ ，插入时有可能需要增容，增容：开辟新空间，拷贝元素，释放旧空间，导致效率更低	任意位置插入和删除效率高，不需要搬移元素，时间复杂度为 $O(1)$
空间利用率	底层为连续空间，不容易造成内存碎片，空间利用率高，缓存利用率高	底层节点动态开辟，小节点容易造成内存碎片，空间利用率低，缓存利用率低
迭代器	原生态指针	对原生态指针(节点指针)进行封装
迭代器失效	在插入元素时，要给所有的迭代器重新赋值，因为插入元素有可能会重新扩容，致使原来迭代器失效，删除时，当前迭代器需要重新赋值否则会失效	插入元素不会导致迭代器失效，删除元素时，只会导致当前迭代器失效，其他迭代器不受影响
使用场景	需要高效存储，支持随机访问，不关心插入删除效率	大量插入和删除操作，不关心随机访问