List复习

1. list的介绍及使用

1.1 list的介绍

list的文档介绍

1.2 list的特点

list是C++标准模板库(STL)中的一个双向链表容器。以下是其主要特点：

基本特性

- 双向链表结构：每个元素包含指向前驱和后继节点的指针
- 非连续存储：元素在内存中不是连续存储的
- 动态大小：可以根据需要动态增长或缩小

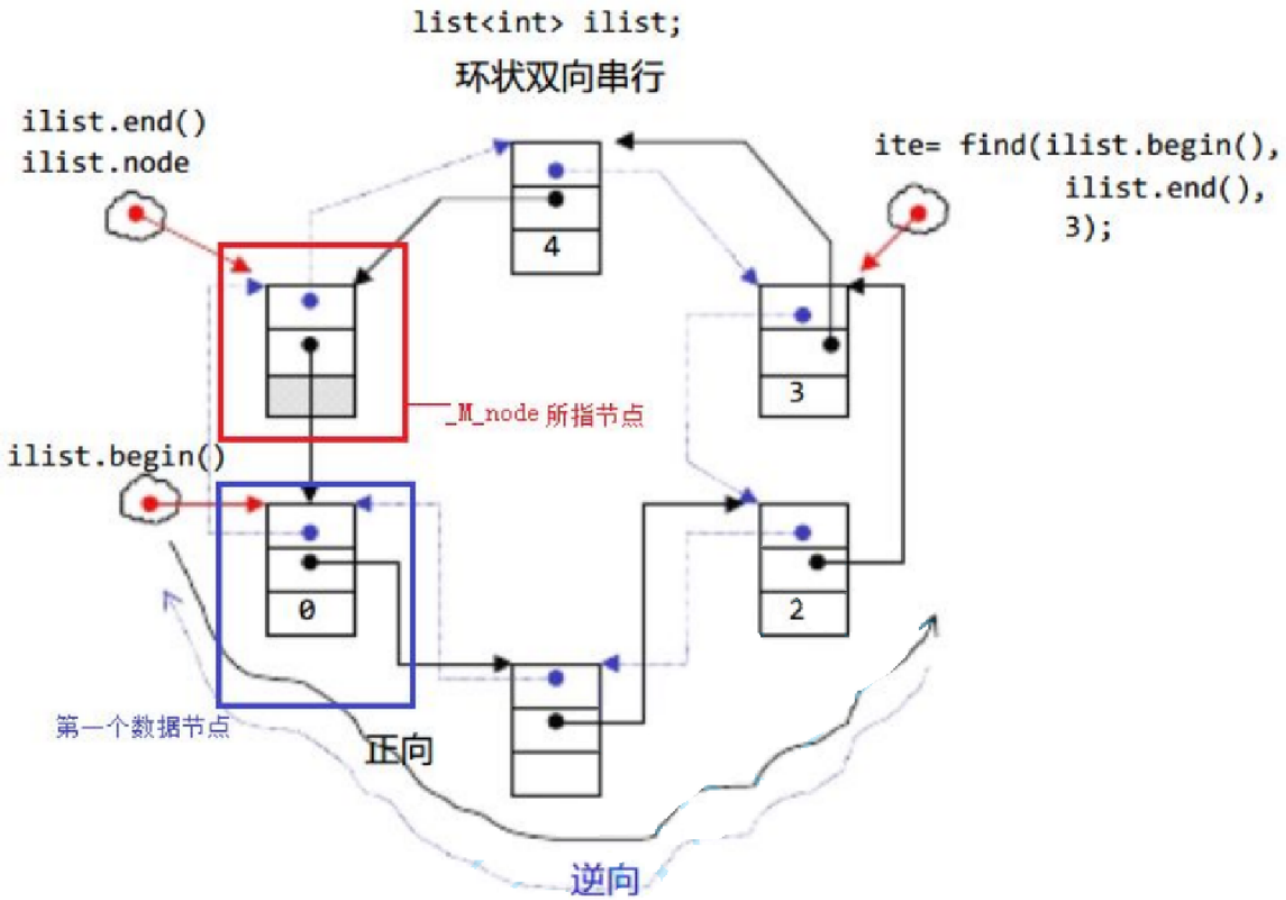
主要优点

a. 高效的插入和删除：
   - 在任何位置插入/删除元素的时间复杂度都是O(1)
   - 特别适合频繁在中间位置进行插入/删除操作的场景

b. 不失效的迭代器：
   - 插入操作不会使现有迭代器失效
   - 只有删除操作会使指向被删除元素的迭代器失效

主要缺点

- 随机访问性能差：访问第n个元素需要O(n)时间
- 内存开销较大：每个元素需要额外存储前后指针



1.3 list的使用

1.3.1 listt的构造

构造函数(constructor)	接口说明
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()	构造空的list
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

1.3.2 list iterator的使用

> list的迭代器可以看做是一个指针，指针指向list中的某个节点

函数声明	接口说明

begin + end	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
rbegin + rend	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

1.3.3 list capacity

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

1.3.4 list element access

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

1.3.5 list modifiers

函数声明	接口说明
push_front	在list首元素前插入值为val的元素
pop_front	删除list中第一个元素
push_back	在list尾部插入值为val的元素
pop_back	删除list中最后一个元素
insert	在list position 位置中插入值为val的元素
erase	删除list position位置的元素

swap	交换两个list中的元素
clear	清空list中的有效元素

1.3.6 list迭代器失效

list 迭代器失效的情况

a. 元素被删除时：
   - 指向被删除元素的迭代器会失效
   - 其他迭代器仍然有效

b. 整个 list 被销毁或清空时：
   - 所有迭代器都会失效

c. 使用 splice 转移元素时：
   - 被转移元素的迭代器仍然有效，只是现在指向新容器中的元素

list 迭代器不会失效的情况

a. 插入新元素：
   - 在任何位置插入元素都不会使任何迭代器失效

b. 合并 lists：
   - 使用 `merge` 或 `splice` 合并 lists 不会使迭代器失效

示例代码

代码块

```
#include <iostream>
#include <list>

void TestListIterator1()
{
    int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    list<int> l(array, array+sizeof(array)/sizeof(array[0]));
    auto it = l.begin();
    while (it != l.end())
    {
        // 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
        l.erase(it);
        ++it;
    }
}
```

```
17
18  // 改正
19  void TestListIterator()
20  {
21      int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
22      list<int> l(array, array+sizeof(array)/sizeof(array[0]));
23      auto it = l.begin();
24      while (it != l.end())
25      {
26          l.erase(it++); // it = l.erase(it);
27      }
28  }
29
```

2. list的模拟实现

代码块

```
1   #pragma once
2   #include<iostream>
3   namespace List
4   {
5       template<class T>
6       class ListNode
7       {
8       public:
9           T _data;
10          ListNode<T>* _prev;
11          ListNode<T>* _next;
12
13          ListNode(const T& data = T())
14              :_data(data), _prev(nullptr), _next(nullptr)
15          { }
16      };
17
18      template<class T>
19      struct ListIterator
20      {
21          typedef ListNode<T> Node;
22          typedef ListIterator<T> Self;
23
24          Node* _node;
25
26          ListIterator(Node* node) :_node(node) {}
```

```
27          ListIterator() :_node(nullptr) {}
28
29          T& operator* () const { return _node->_data; }
30          T* operator->() const { return (operator*()); }
31
32          Self& operator++()
33          {
34              _node = _node->_next;
35              return *this;
36          }
37          Self& operator++(T)
38          {
39              Self tmp = *this;
40              ++(*this);
41              return tmp;
42          }
43
44          Self& operator--()
45          {
46              _node = _node->_prev;
47              return *this;
48          }
49          Self& operator--(T)
50          {
51              Self tmp = *this;
52              --(*this);
53              return tmp;
54          }
55
56          bool operator==(const Self& s) { return s._node == _node; }
57          bool operator!=(const Self& s) { return s._node != _node; }
58      };
59
60      template<class T>
61      class list
62      {
63      public:
64          typedef ListNode<T> Node;
65          typedef ListIterator<T> iterator;
66
67          iterator begin() const { return _head->_next; }
68          iterator end() const { return _head; }
69
70          list() :_head(new Node(T()))
71          {
72              _head->_next = _head->_prev = _head;
73          }
```

```
74
75          list(const list& x) :_head(new Node(T()))
76          {
77              for (const auto& val : x) push_back(val);
78          }
79
80          void swap(list& x)
81          {
82              std::swap(_head, x._head);
83              std::swap(_size, x._size);
84          }
85
86          bool empty() const { return _size == 0; }
87
88          size_t size() const { return _size; }
```

代码块

```
1   void push_back(const T& data) { insert(end(), data); }
2           void push_front(const T& data) { insert(begin(), data); }
3
4           iterator insert(iterator pos, const T& val)
5           {
6               Node* newnode = new Node(val);
7               iterator it = begin();
8               while (it != pos) it++;
9
10              it._node->_prev->_next = newnode;
11              newnode->_prev = it._node->_prev;
12              it._node->_prev = newnode;
13              newnode->_next = it._node;
14
15              _size++;
16
17              return it;
18          }
19
20          void insert(iterator pos, size_t n, const T& val)
21          {
22              Node* cur = pos._node->_prev;
23              Node* next = cur->_next;
24              while (n--)
25              {
26                  Node* newnode = new Node(val);
27                  cur->_next = newnode;
28                  newnode->_prev = cur;
29
```

```
30                  cur = newnode;
31              }
32
33              cur->_next = next;
34              next->_prev = cur;
35          }
36
37          void pop_back() { erase(end()); }
38          void pop_front() { erase(begin()); }
39
40          void erase(iterator pos)
41          {
42              if (pos == _head) pos = _head->_prev;
43              Node* prev = pos._node->_prev;
44              Node* next = pos._node->_next;
45
46              prev->_next = next;
47              next->_prev = prev;
48              delete pos._node;
49
50              _size--;
51          }
52
53          iterator erase(iterator first, iterator last)
54          {
55              if (first == last) return last;
56
57              // 获取要删除的起始和结束节点
58              Node* first_node = first._node;
59              Node* last_node = last._node->_prev;  // 删除到last的前一个节点
60
61              // 重新连接链表
62              Node* before_first = first_node->_prev;
63              Node* after_last = last_node->_next;
64
65              before_first->_next = after_last;
66              after_last->_prev = before_first;
67
68              // 删除范围内的所有节点
69              Node* current = first_node;
70              while (current != after_last) {
71                      Node* next_node = current->_next;
72                      delete current;
73                      current = next_node;
74                      _size--;  // 减少元素计数
75              }
76
```

```
77              return iterator(after_last);  // 返回指向last的迭代器
78          }
79
80      private:
81          Node* _head;
82          size_t _size = 0;
83      };
84
85  }
```

3. list与vector的对比

	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	原生态指针	对原生态指针(节点指针)进行封装

代器		
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导致重新扩容，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