

Lesson10---list

【本节目标】

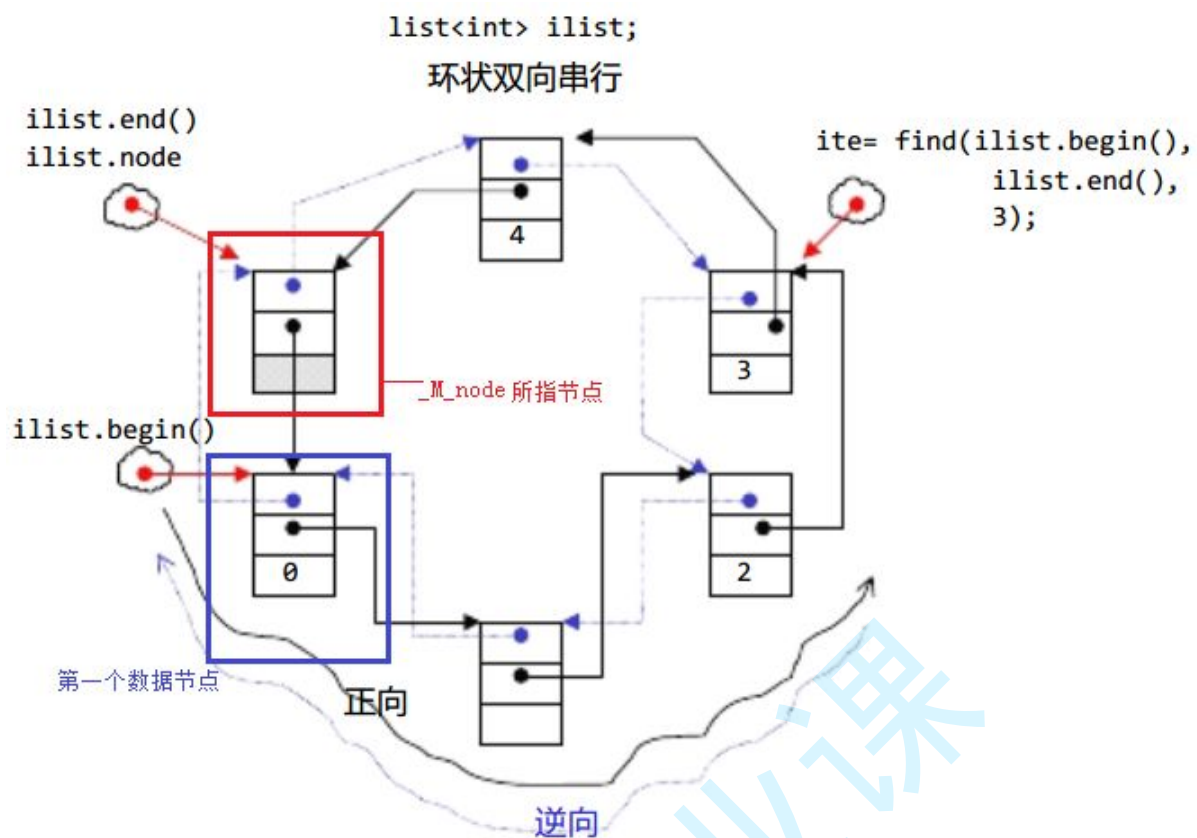
- 1. list的介绍及使用
- 2. list的深度剖析及模拟实现
- 3. list与vector的对比

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

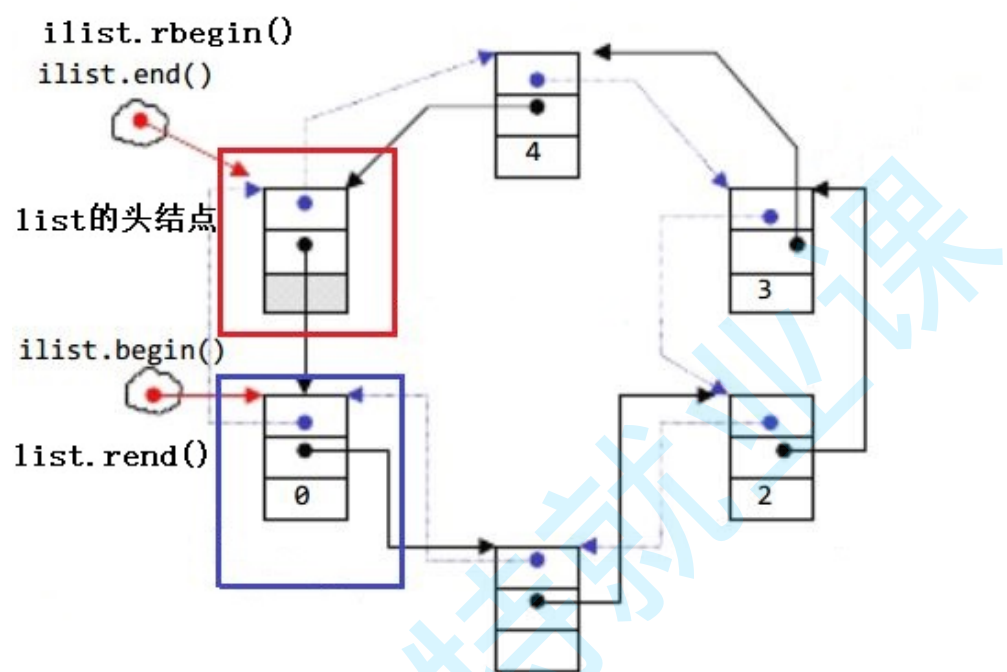
| 构造函数 (constructor) | 接口说明 |
|--|----------------------------|
| list (size_type n, const value_type& val = value_type()) | 构造的list中包含n个值为val的元素 |
| list() | 构造空的list |
| list (const list& x) | 拷贝构造函数 |
| list (InputIterator first, InputIterator last) | 用[first, last)区间中的元素构造list |

[list的构造使用代码演示](#)

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 |
|---|--|
| 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

[list的迭代器使用代码演示](#)

1.2.3 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 |

1.2.4 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 |

[list的插入和删除使用代码演示](#)

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。

[list的模拟实现](#)

2.2 list的反向迭代器

通过前面例子知道，反向迭代器的++就是正向迭代器的--，反向迭代器的--就是正向迭代器的++，因此反向迭代器的实现可以借助正向迭代器，即：反向迭代器内部可以包含一个正向迭代器，对正向迭代器的接口进行包装即可。

```

1  template<class Iterator>
2  class ReverseListIterator
3  {
4      // 注意：此处typename的作用是明确告诉编译器，Ref是Iterator类中的类型，而不是静态成员变量
5      // 否则编译器编译时就不知道Ref是Iterator中的类型还是静态成员变量
6      // 因为静态成员变量也是按照 类名::静态成员变量名 的方式访问的
7  public:
8      typedef typename Iterator::Ref Ref;
9      typedef typename Iterator::Ptr Ptr;
10     typedef ReverseListIterator<Iterator> Self;
11  public:
12     //////////////////////////////////////
13     // 构造
14     ReverseListIterator(Iterator it): _it(it){}
15     //////////////////////////////////////
16     // 具有指针类似行为
17     Ref operator*(){
18         Iterator temp(_it);
19         --temp;
20         return *temp;
21     }
22     Ptr operator->(){ return &(operator*());}
23     //////////////////////////////////////
24     // 迭代器支持移动
25     Self& operator++(){

```

```

26         --_it;
27         return *this;
28     }
29     Self operator++(int){
30         Self temp(*this);
31         --_it;
32         return temp;
33     }
34     Self& operator--(){
35         ++_it;
36         return *this;
37     }
38     Self operator--(int)
39     {
40         Self temp(*this);
41         ++_it;
42         return temp;
43     }
44     //////////////////////////////////////
45     // 迭代器支持比较
46     bool operator!=(const Self& l)const{    return _it != l._it;}
47     bool operator==(const Self& l)const{    return _it == l._it;}
48     Iterator _it;
49 };

```

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 |