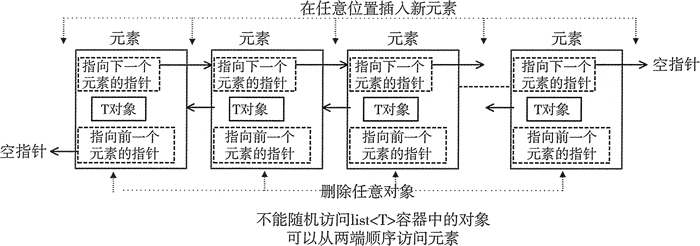
[STL](http://c.biancheng.net/stl/" \t "_blank) list 容器，又称双向链表容器，即该容器的底层是以双向链表的形式实现的。这意味着，list 容器中的元素可以分散存储在内存空间里，而不是必须存储在一整块连续的内存空间中。  
  
图 1 展示了 list 双向链表容器是如何存储元素的。

  
图 1 list 双向链表容器的存储结构示意图

可以看到，list 容器中各个元素的前后顺序是靠[指针](http://c.biancheng.net/c/80/" \t "_blank)来维系的，每个元素都配备了 2 个指针，分别指向它的前一个元素和后一个元素。其中第一个元素的前向指针总为 null，因为它前面没有元素；同样，尾部元素的后向指针也总为 null。  
  
基于这样的存储结构，list 容器具有一些其它容器（array、vector 和 deque）所不具备的优势，即它可以在序列已知的任何位置快速插入或删除元素（时间复杂度为O(1)）。并且在 list 容器中移动元素，也比其它容器的效率高。  
  
使用 list 容器的缺点是，它不能像 array 和 vector 那样，通过位置直接访问元素。举个例子，如果要访问 list 容器中的第 6 个元素，它不支持容器对象名[6]这种语法格式，正确的做法是从容器中第一个元素或最后一个元素开始遍历容器，直到找到该位置。

实际场景中，如何需要对序列进行大量添加或删除元素的操作，而直接访问元素的需求却很少，这种情况建议使用 list 容器存储序列。

list 容器以模板类 list<T>（T 为存储元素的类型）的形式在<list>头文件中，并位于 std 命名空间中。因此，在使用该容器之前，代码中需要包含下面两行代码：

1. #include <list>
2. **u**[**sin**](http://c.biancheng.net/ref/sin.html)**g** **namespace** std;

注意，std 命名空间也可以在使用 list 容器时额外注明，两种方式都可以。

**list容器的创建**

根据不同的使用场景，有以下 5 种创建 list 容器的方式供选择。  
  
1) 创建一个没有任何元素的空 list 容器：

1. std::list<int> values;

和空 array 容器不同，空的 list 容器在创建之后仍可以添加元素，因此创建 list 容器的方式很常用。  
  
2) 创建一个包含 n 个元素的 list 容器：

1. std::list<int> values(10);

通过此方式创建 values 容器，其中包含 10 个元素，每个元素的值都为相应类型的默认值（int类型的默认值为 0）。  
  
3) 创建一个包含 n 个元素的 list 容器，并为每个元素指定初始值。例如：

1. std::list<int> values(10, 5);

如此就创建了一个包含 10 个元素并且值都为 5 个 values 容器。  
  
4) 在已有 list 容器的情况下，通过拷贝该容器可以创建新的 list 容器。例如：

1. std::list<int> value1(10);
2. std::list<int> value2(value1);

注意，采用此方式，必须保证新旧容器存储的元素类型一致。  
  
5) 通过拷贝其他类型容器（或者普通数组）中指定区域内的元素，可以创建新的 list 容器。例如：

1. //拷贝普通数组，创建list容器
2. int a[] = { 1,2,3,4,5 };
3. std::list<int> values(a, a+5);
4. //拷贝其它类型的容器，创建 list 容器
5. std::array<int, 5>arr{ 11,12,13,14,15 };
6. std::list<int>values(arr.begin()+2, arr.end());//拷贝arr容器中的{13,14,15}

**list容器可用的成员函数**

表 2 中罗列出了 list 模板类提供的所有成员函数以及各自的功能。

|  |  |
| --- | --- |
| 表 2 list 容器可用的成员函数 | |
| **成员函数** | **功能** |
| begin() | 返回指向容器中第一个元素的双向迭代器。 |
| end() | 返回指向容器中最后一个元素所在位置的下一个位置的双向迭代器。 |
| rbegin() | 返回指向最后一个元素的反向双向迭代器。 |
| rend() | 返回指向第一个元素所在位置前一个位置的反向双向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| empty() | 判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。 |
| size() | 返回当前容器实际包含的元素个数。 |
| max\_size() | 返回容器所能包含元素个数的最大值。这通常是一个很大的值，一般是 232-1，所以我们很少会用到这个函数。 |
| front() | 返回第一个元素的引用。 |
| back() | 返回最后一个元素的引用。 |
| assign() | 用新元素替换容器中原有内容。 |
| emplace\_front() | 在容器头部生成一个元素。该函数和 push\_front() 的功能相同，但效率更高。 |
| push\_front() | 在容器头部插入一个元素。 |
| pop\_front() | 删除容器头部的一个元素。 |
| emplace\_back() | 在容器尾部直接生成一个元素。该函数和 push\_back() 的功能相同，但效率更高。 |
| push\_back() | 在容器尾部插入一个元素。 |
| pop\_back() | 删除容器尾部的一个元素。 |
| emplace() | 在容器中的指定位置插入元素。该函数和 insert() 功能相同，但效率更高。 |
| insert() | 在容器中的指定位置插入元素。 |
| erase() | 删除容器中一个或某区域内的元素。 |
| swap() | 交换两个容器中的元素，必须保证这两个容器中存储的元素类型是相同的。 |
| resize() | 调整容器的大小。 |
| clear() | 删除容器存储的所有元素。 |
| splice() | 将一个 list 容器中的元素插入到另一个容器的指定位置。 |
| remove(val) | 删除容器中所有等于 val 的元素。 |
| remove\_if() | 删除容器中满足条件的元素。 |
| unique() | 删除容器中相邻的重复元素，只保留一个。 |
| merge() | 合并两个事先已排好序的 list 容器，并且合并之后的 list 容器依然是有序的。 |
| sort() | 通过更改容器中元素的位置，将它们进行排序。 |
| reverse() | 反转容器中元素的顺序。 |

除此之外，[C++](http://c.biancheng.net/cplus/" \t "_blank) 11 标准库还新增加了 begin() 和 end() 这 2 个函数，和 list 容器包含的 begin() 和 end() 成员函数不同，标准库提供的这 2 个函数的操作对象，既可以是容器，还可以是普通数组。当操作对象是容器时，它和容器包含的 begin() 和 end() 成员函数的功能完全相同；

如果操作对象是普通数组，则 begin() 函数返回的是指向数组第一个元素的指针，同样 end() 返回指向数组中最后一个元素之后一个位置的指针（注意不是最后一个元素）。  
  
list 容器还有一个std::swap(x , y)非成员函数（其中 x 和 y 是存储相同类型元素的 list 容器），它和 swap() 成员函数的功能完全相同，仅使用语法上有差异。  
  
如下代码演示了表 2 中部分成员函数的用法：

1. #include <iostream>
2. #include <list>
3. **using** **namespace** std;
4. int main()
5. {
6. //创建空的 list 容器
7. std::list<double> values;
8. //向容器中添加元素
9. values.push\_back(3.1);
10. values.push\_back(2.2);
11. values.push\_back(2.9);
12. cout << "values size：" << values.size() << endl;
13. //对容器中的元素进行排序
14. values.sort();
15. //使用迭代器输出list容器中的元素
16. **for** (std::list<double>::iterator it = values.begin(); it != values.end(); ++it) {
17. std::cout << \*it << " ";
18. }
19. **return** 0;
20. }

运行结果为：

values size：3  
2.2 2.9 3.1