C++ STL vector容器迭代器用法详解

在《STL array随机访问迭代器》一节中，详细介绍了 array 容器迭代器，vector 容器迭代器和前者有很多相同之处。比如，vector 容器的迭代器也是随机访问迭代器，并且 vector 模板类提供的操作迭代器的成员函数也和 array 容器一样（如表 1 所示）。

表 1 vector 支持迭代器的成员函数

成员函数	功能
begin()	返回指向容器中第一个元素的正向迭代器；如果是 const 类型容器，在该函数返回的是常量正向迭代器。
end()	返回指向容器最后一个元素之后一个位置的正向迭代器；如果是 const 类型容器，在该函数返回的是常量正向迭代器。此函数通常和 begin() 搭配使用。
rbegin()	返回指向最后一个元素的反向迭代器；如果是 const 类型容器，在该函数返回的是常量反向迭代器。
rend()	返回指向第一个元素之前一个位置的反向迭代器。如果是 const 类型容器，在该函数返回的是常量反向迭代器。此函数通常和 rbegin() 搭配使用。
cbegin()	和 begin() 功能类似，只不过其返回的迭代器类型为常量正向迭代器，不能用于修改元素。
cend()	和 end() 功能相同，只不过其返回的迭代器类型为常量正向迭代器，不能用于修改元素。
crbegin()	和 rbegin() 功能相同，只不过其返回的迭代器类型为常量反向迭代器，不能用于修改元素。
crend()	和 rend() 功能相同，只不过其返回的迭代器类型为常量反向迭代器，不能用于修改元素。

> 除此之外，C++ 11 新添加的 begin() 和 end() 全局函数也同样适用于 vector 容器。即当操作对象为 vector 容器时，其功能分别和表 1 中的 begin()、end() 成员函数相同，具体用法本节后续会做详细介绍。

表 1 中这些成员函数的具体功能如图 2 所示。

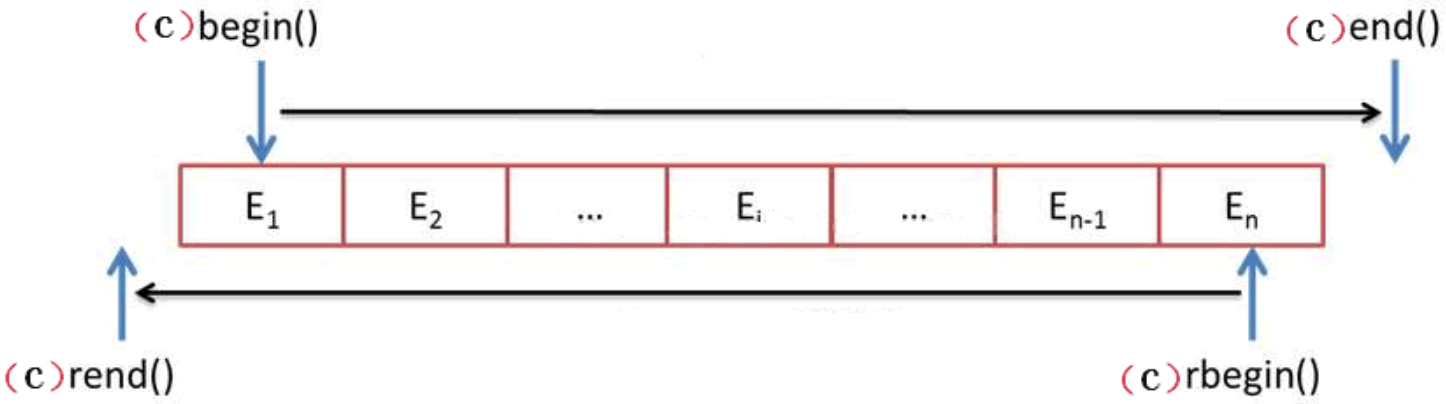

图 2 迭代器的具体功能示意图

从图 2 可以看出，这些成员函数通常是成对使用的，即 begin()/end()、rbegin()/rend()、cbegin()/cend()、crbegin()/crend() 各自成对搭配使用。其中，begin()/end() 和 cbegin/cend() 的功能是类似的，同样 rbegin()/rend() 和 crbegin()/crend() 的功能是类似的。

> 值得一提的是，以上函数在实际使用时，其返回值类型都可以使用 auto 关键字代替，编译器可以自行判断出该迭代器的类型。

vector容器迭代器的基本用法

vector 容器迭代器最常用的功能就是遍历访问容器中存储的元素。

首先来看 begin() 和 end() 成员函数，它们分别用于指向「首元素」和「尾元素+1」 的位置，下面程序演示了如何使用 begin() 和 end() 遍历 vector 容器并输出其中的元素：

```
01.  #include <iostream>
02.  //需要引入 vector 头文件
03.  #include <vector>
04.  using namespace std;
05.  int main()
06.  {
07.      vector<int>values{1,2,3,4,5};
08.      auto first = values.begin();
09.      auto end = values.end();
10.      while (first != end)
11.      {
12.          cout << *first << " ";
13.          ++first;
14.      }
15.      return 0;
16.  }
```

输出结果为：

```
1 2 3 4 5
```

可以看到，迭代器对象是由 vector 对象的成员函数 begin() 和 end() 返回的。我们可以像使用普通指针那样上使用它们。比如代码中，在保存了元素值后，使用前缀 `++` 运算符对 first 进行自增，当 first 等于 end 时，所有的元素都被设完值，循环结束。

与此同时，还可以使用全局的 begin() 和 end() 函数来从容器中获取迭代器，比如将上面代码中第 8、9 行代码用如下代码替换：

```
01.  auto first = std::begin(values);
02.  auto end = std::end (values);
```

cbegin()/cend() 成员函数和 begin()/end() 唯一不同的是，前者返回的是 const 类型的正向迭代器，这就意味着，由 cbegin() 和 cend() 成员函数返回的迭代器，可以用来遍历容器内的元素，也可以访问元素，但是不能对所存储的元素进行修改。

举个例子：

```
01.  #include <iostream>
02.  //需要引入 vector 头文件
03.  #include <vector>
04.  using namespace std;
05.  int main()
06.  {
07.      vector<int>values{1,2,3,4,5};
08.      auto first = values.cbegin();
09.      auto end = values.cend();
10.      while (first != end)
11.      {
12.          //*first = 10;不能修改元素
13.          cout << *first << " ";
14.          ++first;
15.      }
16.      return 0;
17.  }
```

程序第 12 行，由于 first 是 const 类型的迭代器，因此不能用于修改容器中元素的值。

vector 模板类中还提供了 rbegin() 和 rend() 成员函数，分别表示指向最后一个元素和第一个元素前一个位置的随机访问迭代器，又称它们为反向迭代器（如图 2 所示）。

> 需要注意的是，在使用反向迭代器进行 ++ 或 -- 运算时，++ 指的是迭代器向左移动一位，-- 指的是迭代器向右移动一位，即这两个运算符的功能也“互换”了。

反向迭代器用于以逆序的方式遍历容器中的元素。例如：

```
#include <iostream>
//需要引入 vector 头文件
#include <vector>
using namespace std;
int main()
{
    vector<int>values{1,2,3,4,5};
    auto first = values.rbegin();
    auto end = values.rend();
    while (first != end)
    {
        cout << *first << " ";
        ++first;
    }
    return 0;
}
```

运行结果为：

```
5 4 3 2 1
```

可以看到，从最后一个元素开始循环，遍历输出了容器中的所有元素。结束迭代器指向第一个元素之前的位置，所以当 first 指向第一个元素并 +1 后，循环就结束了。

当然，在上面程序中，我们也可以使用 for 循环：

```
for (auto first = values.rbegin(); first != values.rend(); ++first) {
    cout << *first << " ";
}
```

crbegin()/crend() 组合和 rbegin()/crend() 组合唯一的区别在于，前者返回的迭代器为 const 类型，即不能用来修改容器中的元素，除此之外在使用上和后者完全相同。

> 有关 crbegin()/crend() 成员函数，这里不再给出具体实例，有兴趣的读者，可自行编写代码进行测试。

vector容器迭代器的独特之处

和 array 容器不同，vector 容器可以随着存储元素的增加，自行申请更多的存储空间。因此，在创建 vector 对象时，我们可以直接创建一个空的 vector 容器，并不会影响后续使用该容器。

但这会产生一个问题，即在初始化空的 vector 容器时，不能使用迭代器。也就是说，如下初始化 vector 容器的方法是不行的：

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int>values;
    int val = 1;
    for (auto first = values.begin(); first < values.end(); ++first, val++) {
        *first = val;
        //初始化的同时输出值
        cout << *first;
    }
    return 0;
}
```

运行程序可以看到，什么也没有输出。这是因为，对于空的 vector 容器来说，begin() 和 end() 成员函数返回的迭代器是相等的，即它们指向的是同一个位置。

> 所以，对于空的 vector 容器来说，可以通过调用 push_back() 或者借助 resize() 成员函数实现初始化容器的目的。

除此之外，vector 容器在申请更多内存的同时，容器中的所有元素可能会被复制或移动到新的内存地址，这会导致之前创建的迭代器失效。

举个例子：

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int>values{1,2,3};
```

```
07.     cout << "values 容器首个元素的地址：" << values.data() << endl;
08.     auto first = values.begin();
09.     auto end = values.end();
10.     //增加 values 的容量
11.     values.reserve(20);
12.     cout << "values 容器首个元素的地址：" << values.data() << endl;
13.     while (first != end) {
14.         cout << *first;
15.         ++first;
16.     }
17.     return 0;
18. }
```

运行程序，显示如下信息并崩溃：

```
values 容器首个元素的地址：0096DFE8
values 容器首个元素的地址：00965560
```

可以看到，values 容器在增加容量之后，首个元素的存储地址发生了改变，此时再使用先前创建的迭代器，显然是错误的。因此，为了保险起见，每当 vector 容器的容量发生变化时，我们都要对之前创建的迭代器重新初始化一遍：

```
01. #include <iostream>
02. #include <vector>
03. using namespace std;
04. int main()
05. {
06.     vector<int>values{1,2,3};
07.     cout << "values 容器首个元素的地址：" << values.data() << endl;
08.     auto first = values.begin();
09.     auto end = values.end();
10.     //增加 values 的容量
11.     values.reserve(20);
12.     cout << "values 容器首个元素的地址：" << values.data() << endl;
13.     first = values.begin();
14.     end = values.end();
15.     while (first != end) {
16.         cout << *first ;
17.         ++first;
18.     }
19.     return 0;
20. }
```

运行结果为：

```
values 容器首个元素的地址：0164DBE8
values 容器首个元素的地址：01645560
123
```