

## 关于迭代器失效的几种情况\_哪些场景会导致迭代器失效\_Shining-LY的博客-CSDN博客

成就一亿技术人!


之前就做题的时候就经常碰到与迭代器失效有关的问题，但是一直对这个问题也没有深究，处于似懂非懂的状态，今天就对迭代器失效这部分知识做一个总结。

### 迭代器

迭代器（[iterator](#)）是一个可以对其执行类似指针的操作（如：解除引用（`operator*`）和递增（`operator++`））的对象，我们可以将它理解成为一个指针。但它又不是我们所谓普通的指针，我们可以称之为广义指针，你可以通过`sizeof（vector::iterator）`来查看，所占内存并不是4个字节。

如下图所示：

```
vector<int>::iterator it;
cout << sizeof(it) << endl;
/*for (it = vec.begin(); it
```



12  
请按任意键继续. . .

[https://blog.csdn.net/qq\\_37964547](https://blog.csdn.net/qq_37964547)

这里我们定义了一个`vector`迭代器，对其求`sizeof（）`，发现是12个字节，并不是一个指针的大小。

那么我们常说的迭代器失效到底是什么呢？都有哪些场景会导致失效问题呢？我们一起来看看以下具体场景及解决办法。

### 一、序列式容器迭代器失效

对于序列式容器，例如`vector`、`deque`：由于序列式容器是组合式容器，当当前元素的`iterator`被删除后，其后的所有元素的迭代器都会失效，这是因为`vector`、`deque`都是连续存储的一段空间，所以当对其进行`erase`操作时，其后的每一个元素都会向前移一个位置。

```
#include<iostream>
using namespace std;

#include<vector>

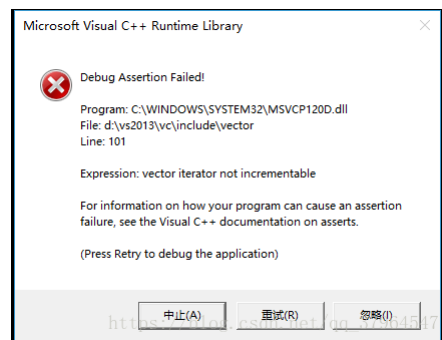
void VectorTest()
{
    vector<int> vec;
    for (int i = 0; i < 5; i++)
    {
        vec.push_back(i);
    }
    vector<int>::iterator it;
    cout << sizeof(it) << endl;
    for (it = vec.begin(); it != vec.end(); it++)
    {
        if (*it>2)
            vec.erase(it); //此处会发生迭代器失效
    }
    for (it = vec.begin(); it != vec.end(); it++)
        cout << *it << " ";
    cout << endl;
}

int main()
{
    VectorTest();
    system("pause");
    return 0;
}
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18

19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29

运行结果，程序终止：



给出的报错信息是：vector iterator not incrementable

已经失效的迭代器不能进行++操作，所以程序中断了。不过vector的erase操作可以返回下一个有效的迭代器，所以只要我们每次执行删除操作的时候，将下一个有效迭代器返回就可以顺利执行后续操作了，代码修改如下：

```
void VectorTest()
{
    vector<int> vec;
    for (int i = 0; i < 5; i++)
    {
        vec.push_back(i);
    }
    vector<int>::iterator it;
    cout << sizeof(it) << endl;
    for (it = vec.begin(); it != vec.end(); )
    {
        if (*it==3)
        {
            it = vec.erase(it); //更新迭代器it
        }
        it++;
    }
    for (it = vec.begin(); it != vec.end(); it++)
        cout << *it << " ";
    cout << endl;
}
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16

17  
18  
19  
20  
21  
22

运行结果:

```

8  vector<int> vec;
9  for (int i = 0; i < 5; i++)
10 {
11     vec.push_back(i);
12 }
13 vector<int>::iterator it;
14 cout << sizeof(it) << endl;
15 for (it = vec.begin(); it != vec.end(); it++)
16 {
17     if (*it==3)
18     {
19         it = vec.erase(it);
20     }
21 }
22 for (it = vec.begin(); it != vec.end(); it++)

```

C:\Users\Liuyan\documents\visual studio 2019

```

12
0 1 2 4
请按任意键继续. . .

```

这样删除后`it`指向的元素后，返回的是下一个元素的迭代器，这个迭代器是`vector`内存调整后新的有效的迭代器。此时就可以进行正确的删除与访问操作了。

上面只是举了删除元素造成的`vector`迭代器失效问题，对于`vector`的插入元素也可以同理得到验证，这里就不再进行举例了。

### vector迭代器失效问题总结

- (1) 当执行`erase`方法时，指向删除节点的迭代器全部失效，指向删除节点之后的全部迭代器也失效
- (2) 当进行`push_back` () 方法时，`end`操作返回的迭代器肯定失效。
- (3) 当插入(`push_back`)一个元素后，`capacity`返回值与没有插入元素之前相比有改变，则需要重新加载整个容器，此时`first`和`end`操作返回的迭代器都会失效。
- (4) 当插入(`push_back`)一个元素后，如果空间未重新分配，指向插入位置之前的元素的迭代器仍然有效，但指向插入位置之后元素的迭代器全部失效。

### deque迭代器失效总结:

- (1) 对于`deque`,插入到除首尾位置之外的任何位置都会导致迭代器、指针和引用都会失效，但是如果在首尾位置添加元素，迭代器会失效，但是指针和引用不会失效
- (2) 如果在首尾之外的任何位置删除元素，那么指向被删除元素外其他元素的迭代器全部失效
- (3) 在其首部或尾部删除元素则只会使指向被删除元素的迭代器失效。

## 二、关联式容器迭代器失效

对于关联容器(如`map`,`set`,`multimap`,`multiset`)，删除当前的`iterator`，仅仅会使当前的`iterator`失效，只要在`erase`时，递增当前`iterator`即可。这是因为`map`之类的容器，使用了红黑树来实现，插入、删除一个结点不会对其他结点造成影响。`erase`迭代器只是被删元素的迭代器失效，但是返回值为`void`，所以要采用`erase(iter++)`的方式删除迭代器。

首先来看一下`map`迭代器失效的一个例子:

```

void mapTest()
{
    map<int, int>m;
    for (int i = 0; i < 10; i++)
    {
        m.insert(make_pair(i, i + 1));
    }
    map<int, int>::iterator it;
    for (it = m.begin(); it != m.end(); it++)
    {
        if ( (it->first) >5)
            m.erase(it);
    }
}

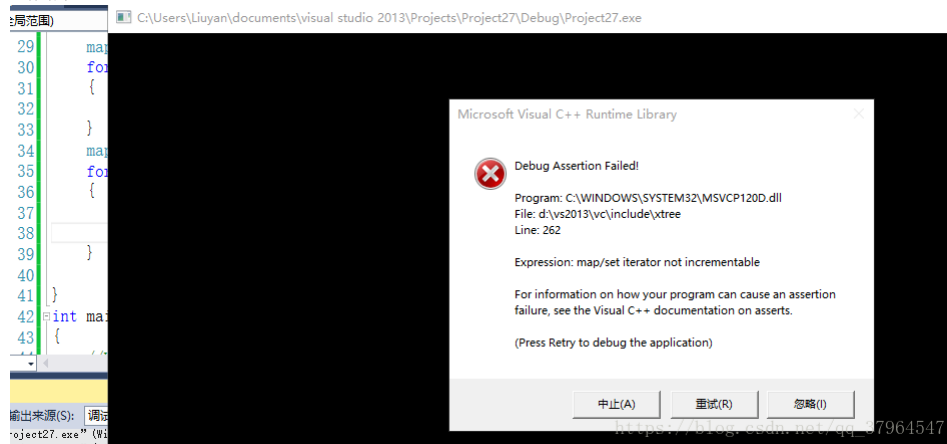
int main()
{
    mapTest();
    system("pause");
    return 0;
}

```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21

运行结果:



这里显示迭代器失效，不能进行++操作，只要稍作修改就可以了：

```
void mapTest()  
{  
    map<int, int> m;  
    for (int i = 0; i < 10; i++)  
    {  
        m.insert(make_pair(i, i + 1));  
    }  
    map<int, int>::iterator it;  
  
    for (it = m.begin(); it != m.end(); )  
    {  
        if (it->first == 5)  
            m.erase(it++);  
        it++;  
    }  
    for (it = m.begin(); it != m.end(); it++)  
    {  
        cout << (*it).first << " ";  
    }  
    cout << endl;  
}
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17

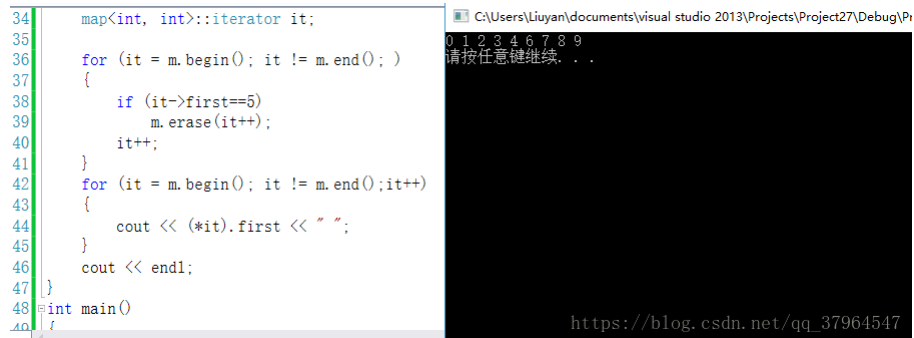
18

19

20

21

运行结果:



```

34 map<int, int>::iterator it;
35
36 for (it = m.begin(); it != m.end(); )
37 {
38     if (it->first==5)
39         m.erase(it++);
40     it++;
41 }
42 for (it = m.begin(); it != m.end(); it++)
43 {
44     cout << (*it).first << " ";
45 }
46 cout << endl;
47 }
48 #include <iostream>
49 #include <map>
50 using namespace std;
51 int main()
52 {
53     map<int, int> m;
54     m[0] = 0; m[1] = 1; m[2] = 2; m[3] = 3; m[4] = 4; m[5] = 5; m[6] = 6; m[7] = 7; m[8] = 8; m[9] = 9;
55     cout << "Map contents before erasing: ";
56     for (it = m.begin(); it != m.end(); it++)
57         cout << (*it).first << " ";
58     cout << endl;
59     // Erase the element with key 5
60     it = m.begin();
61     while (it != m.end())
62     {
63         if (*it).first == 5)
64             m.erase(it++);
65         else
66             it++;
67     }
68     cout << "Map contents after erasing: ";
69     for (it = m.begin(); it != m.end(); it++)
70         cout << (*it).first << " ";
71     cout << endl;
72     return 0;
73 }

```

0 1 2 3 4 6 7 8 9  
请按任意键继续. . .

[https://blog.csdn.net/qq\\_37964547](https://blog.csdn.net/qq_37964547)

此时就可以成功删除key值为5的元素了，而且迭代器++也没有问题了。

这里主要解释一下**erase(it++)**的执行过程：这句话分三步走，先把**iter**传值到**erase**里面，然后**iter**自增，然后执行**erase**，所以**iter**在失效前已经自增了。

**map**是关联容器，以红黑树或者平衡二叉树组织数据，虽然删除了一个元素，整棵树也会调整，以符合红黑树或者二叉树的规范，但是单个节点在内存中的地址没有变化，变化的是各节点之间的指向关系。