迭代器适配器

2021年7月28日 9:50

迭代器适配器

1、介绍

之所以称为迭代器适配器，是因为这些迭代器是在输入迭代器、输出迭代器、正向迭代器、双向迭代器或者随机访问迭代器这些基础迭代器的基础上实现的。也就是说，使用迭代器适配器的过程中，其本质就是在操作某种基础迭代器。

C++ STL 标准库中迭代器大致分为 5 种类型，分别是输入迭代器、输出迭代器、前向迭代器、双向迭代器以及随机访问迭代器。值得一提的是，这 5 种迭代器是 STL 标准库提供的最基础的迭代器，很多场景中遍历容器的需求，它们并不适合。

2、分类

名称	功能
反向迭代器（reverse_iterator）	又称“逆向迭代器”，其内部重新定义了递增运算符（++）和递减运算符（--），专门用来实现对容器的逆序遍历。
安插型迭代器（inserter或者 insert_iterator）	通常用于在容器的任何位置添加新的元素，需要注意的是，此类迭代器不能被运用到元素个数固定的容器（比如 array）上。
流迭代器（istream_iterator / ostream_iterator） 流缓冲区迭代器（istreambuf_iterator / ostreambuf_iterator）	输入流迭代器用于从文件或者键盘读取数据；相反，输出流迭代器用于将数据输出到文件或者屏幕上。 输入流缓冲区迭代器用于从输入缓冲区中逐个读取数据；输出流缓冲区迭代器用于将数据逐个写入输出流缓冲区。
移动迭代器（move_iterator）	此类型迭代器是 C++ 11 标准中新添加的，可以将某个范围的类对象移动到目标范围，而不需要通过拷贝去移动。

下面样例，演示了用反向迭代器适配器遍历 list 容器的实现过程：

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    std::list<int> values{ 1,2,3,4,5 };
    //找到遍历的起点和终点，这里无需纠结定义反向迭代器的语法，后续会详细讲解
    std::reverse_iterator<std::list<int>::iterator> begin = values.rbegin();
    std::reverse_iterator<std::list<int>::iterator> end = values.rend();
    while (begin != end) {
        cout << *begin << " ";
        //注意，这里是 ++，因为反向迭代器内部互换了 ++ 和 -- 的含义
        ++begin;
    }
    return 0;
}
5 4 3 2 1
```

3、反向迭代器适配器（reverse_iterator）

3.1介绍

反向迭代器适配器（reverse_iterator），可简称为反向迭代器或逆向迭代器，常用来对容器进行反向遍历，即从容器中存储的最后一个元素开始，一直遍历到第一个元素。

值得一提的是，反向迭代器底层可以选用双向迭代器或者随机访问迭代器作为其基础迭代器。不仅如此，通过对 ++（递增）和 --（递减）运算符进行重载，使得：

- 当反向迭代器执行 ++ 运算时，底层的基础迭代器实则在执行 -- 操作，意味着反向迭代器在反向遍历容器；
- 当反向迭代器执行 -- 运算时，底层的基础迭代器实则在执行 ++ 操作，意味着反向迭代器在正向遍历容器。

反向迭代器的模板类定义如下：

```
template <class Iterator>
    class reverse_iterator;
```

注意，Iterator 模板参数指的是模板类中所用的基础迭代器的类型，只能选择双向迭代器或者随机访问迭代器。这意味着，如果想使用反向迭代器实现逆序遍历容器，则该容器的迭代器类型必须是双向迭代器或者随机访问迭代器。

3.2使用

3.2.1反向迭代器的创建

reverse_iterator 模板类中共提供了 3 种创建反向迭代器的方法，这里以 vector<int> 容器的随机访问迭代器作为基础迭代器为例。

（1）调用该类的默认构造方法，即可创建了一个不指向任何对象的反向迭代器，例如：

```
std::reverse_iterator<std::vector<int>::iterator> my_reiter;
```

（2）当然，在创建反向迭代器的时候，我们可以直接将一个基础迭代器作为参数传递给新建的反向迭代器。例如：

```
//创建并初始化一个 myvector 容器
std::vector<int> myvector{1,2,3,4,5};
//创建并初始化 my_reiter 迭代器
std::reverse_iterator<std::vector<int>::iterator> my_reiter(myvector.end());
```

我们知道，反向迭代器是通过操纵内部的基础迭代器实现逆向遍历的，但是反向迭代器的指向和底层基础迭代器的指向并不相同。以上面创建的 my_reiter 为例，其内部的基础迭代器指向的是 myvector 容器中元素 5 之后的位置，但是 my_reiter 指向的却是元素 5。

也就是说，反向迭代器的指向和其底层基础迭代器的指向具有这样的关系，即反向迭代器的指向总是距离基础迭代器偏左 1 个位置；反之，基础迭代器的指向总是距离反向迭代器偏右 1 个位置处。

（3）除了第 2 种初始化方式之外，reverse_iterator 模板类还提供了一个复制（拷贝）构造函数，可以实现直接将一个反向迭代器复制给新建的反向迭代器。比如：

```
//创建并初始化一个 vector 容器
std::vector<int> myvector{1,2,3,4,5};
//调用复制构造函数初始化反向迭代器的 2 种方式
std::reverse_iterator<std::vector<int>::iterator> my_reiter(myvector.rbegin());
//std::reverse_iterator<std::vector<int>::iterator> my_reiter = myvector.rbegin();
```

由此，my_reiter 反向迭代器指向的就是 myvector 容器中最后一个元素（也就是 5）之后的位置。

3.2.2成员函数

重载运算符	功能
operator*	以引用的形式返回当前迭代器指向的元素。
operator+	返回一个反向迭代器，其指向距离当前指向的元素之后 n 个位置的元素。此操作要求基础迭代器为随机访问迭代器。
operator++	重载前置 ++ 和后置 ++ 运算符。
operator+=	当前反向迭代器前进 n 个位置，此操作要求基础迭代器为随机访问迭代器。
operator-	返回一个反向迭代器，其指向距离当前指向的元素之前 n 个位置的元素。此操作要求基础迭代器为随机访问迭代器。
operator--	重载前置 -- 和后置 -- 运算符。
operator-=	当前反向迭代器后退 n 个位置，此操作要求基础迭代器为随机访问迭代器。
operator->	返回一个指针，其指向当前迭代器指向的元素。
operator[n]	访问和当前反向迭代器相距 n 个位置处的元素。

```
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
int main() {
    //创建并初始化一个 vector 容器
    std::vector<int> myvector{ 1,2,3,4,5,6,7,8 };
    //创建并初始化一个反向迭代器
    std::reverse_iterator<std::vector<int>::iterator> my_reiter(myvector.rbegin());//指向 8
    cout << *my_reiter << endl;// 8
    cout << *(my_reiter + 3) << endl;// 5
    cout << *(++my_reiter) << endl;// 7
    cout << my_reiter[4] << endl;// 3
    return 0;
}
8
5
7
3
```

除此之外，reverse_iterator 模板类还提供了 base() 成员方法，该方法可以返回当前反向迭代器底层所使用的基础迭代器。举个例子：

```
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
int main() {
    //创建并初始化一个 vector 容器
    std::vector<int> myvector{ 1,2,3,4,5,6,7,8 };
    //创建并初始化反向迭代器 begin，其指向元素 1 之前的位置
    std::reverse_iterator<std::vector<int>::iterator> begin(myvector.begin());
    //创建并初始化反向迭代器 begin，其指向元素 8
    std::reverse_iterator<std::vector<int>::iterator> end(myvector.end());
    //begin底层基础迭代器指向元素 1，end底层基础迭代器指向元素 8 之后的位置
    for (auto iter = begin.base(); iter != end.base(); ++iter) {
        std::cout << *iter << ' ';
    }
    return 0;
}
1 2 3 4 5 6 7 8
```

4、插入迭代器适配器（insert_iterator）

插入迭代器适配器（insert_iterator），简称插入迭代器或者插入器，其功能就是向指定容器中插入元素。值得一提的是，根据插入位置的不同，C++ STL 标准库提供了 3 种插入迭代器。

迭代器适配器	功能
back_insert_iterator	在指定容器的尾部插入新元素，但前提必须是提供有 push_back() 成员方法的容器（包括 vector、deque 和 list）。
front_insert_iterator	在指定容器的头部插入新元素，但前提必须是提供有 push_front() 成员方法的容器（包括 list、deque 和 forward_list）。
insert_iterator	在容器的指定位置之前插入新元素，前提是该容器必须提供有 insert() 成员方法。

4.1 back_insert_iterator迭代器

back_insert_iterator 迭代器可用于在指定容器的末尾处添加新元素。需要注意的是，由于此类型迭代器的底层实现需要调用指定容器的 push_back() 成员方法，这就意味着，此类型迭代器并不适用于 STL 标准库中所有的容器，它只适用于包含 push_back() 成员方法的容器。C++ STL 标准库中，提供有 push_back() 成员方法的容器包括 vector、deque 和 list。

和反向迭代器不同，back_insert_iterator 插入迭代器的定义方式仅有一种，其语法格式如下：

```
std::back_insert_iterator<Container> back_it (container);
```

其中，Container 用于指定插入的目标容器的类型；container 用于指定具体的目标容器。

```
//创建一个 vector 容器
std::vector<int> foo;
//创建一个可向 foo 容器尾部添加新元素的迭代器
std::back_insert_iterator< std::vector<int> > back_it(foo);
```

在此基础上，back_insert_iterator 迭代器模板类中还对赋值运算符（=）进行了重载，借助此运算符，我们可以直接将新元素插入到目标容器的尾部。例如：

```
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
int main() {
    //创建一个 vector 容器
    std::vector<int> foo;
    //创建一个可向 foo 容器尾部添加新元素的迭代器
    std::back_insert_iterator< std::vector<int> > back_it(foo);
    //将 5 插入到 foo 的末尾
    back_it = 5;
    //将 4 插入到当前 foo 的末尾
    back_it = 4;
    //将 3 插入到当前 foo 的末尾
    back_it = 3;
    //将 6 插入到当前 foo 的末尾
    back_it = 6;
    //输出 foo 容器中的元素
    for (std::vector<int>::iterator it = foo.begin(); it != foo.end(); ++it)
        std::cout << *it << ' ';
    return 0;
}
5 4 3 6
```

通过借助赋值运算符，我们依次将 5、4、3、6 插入到 foo 容器中的末尾。这里需要明确的是，每次插入新元素时，该元素都会插入到当前 foo 容器的末尾。换句话说，程序中 11-17 行的每个赋值语句，都可以分解为以下这 2 行代码：

```
//pos表示指向容器尾部的迭代器，value 表示要插入的元素
pos = foo.insert(pos,value);
++pos;
```

除此之外，C++ STL 标准库为了方便用户创建 back_insert_iterator 类型的插入迭代器，提供了 back_inserter() 函数，其语法格式如下：

```
template <class Container>
    back_insert_iterator<Container> back_inserter (Container& x);
```

其中，Container 表示目标容器的类型。

显然在使用该函数时，只需要为其传递一个具体的容器（vector、deque 或者 list）做参数，此函数即可返回一个 back_insert_iterator 类型的插入迭代器。因此，上面程序中的第 9 行代码，可替换成如下语句：

```
std::back_insert_iterator< std::vector<int> > back_it = back_inserter(foo);
```

通过接收 back_inserter() 的返回值，我们也可以得到完全一样的 back_it 插入迭代器。

4.2 front_insert_iterator迭代器

和 back_insert_iterator 正好相反，front_insert_iterator 迭代器的功能是向目标容器的头部插入新元素。并且，由于此类型迭代器的底层实现需要借助目标容器的 push_front() 成员方法，这意味着，只有包含 push_front() 成员方法的容器才能使用该类型迭代器。C++ STL 标准库中，提供有 push_front() 成员方法的容器，仅有 deque、list 和 forward_list。

值得一提的是，定义 front_insert_iterator 迭代器的方式和 back_insert_iterator 完全相同，并且 C++ STL 标准库也提供了 front_inserter() 函数来快速创建该类型迭代器。

```
#include <iostream>
#include <iterator>
#include <forward_list>
using namespace std;
int main() {
    //创建一个 forward_list 容器
    std::forward_list<int> foo;
    //创建一个前插入迭代器
    //std::front_insert_iterator< std::forward_list<int> > front_it(foo);
    std::front_insert_iterator< std::forward_list<int> > front_it = front_inserter(foo);
    //向 foo 容器的头部插入元素
    front_it = 5;
    front_it = 4;
    front_it = 3;
    front_it = 6;
    for (std::forward_list<int>::iterator it = foo.begin(); it != foo.end(); ++it)
        std::cout << *it << ' ';
    return 0;
}
```

```
6 3 4 5
```

4.3 insert_iterator迭代器

当需要向容器的任意位置插入元素时，就可以使用 insert_iterator 类型的迭代器。需要说明的是，该类型迭代器的底层实现，需要调用目标容器的 insert() 成员方法。但幸运的是，STL 标准库中所有容器都提供有 insert() 成员方法，因此 insert_iterator 是唯一可用于关联式容器的插入迭代器。

定义 insert_iterator 类型迭代器的语法格式如下：

```
std::insert_iterator<Container> insert_it (container,it);
```

其中，Container 表示目标容器的类型，参数 container 表示目标容器，而 it 是一个基础迭代器，表示新元素的插入位置。

和前 2 种插入迭代器相比，insert_iterator 迭代器除了定义时需要多传入一个参数，它们的用法完全相同。除此之外，C++ STL 标准库中还提供有 inserter() 函数，可以快速创建 insert_iterator 类型迭代器。

```
#include <iostream>
#include <iterator>
#include <list>
using namespace std;
int main() {
    //初始化为 {5,5}
    std::list<int> foo(2,5);
    //定义一个基础迭代器，用于指定要插入新元素的位置
    std::list<int>::iterator it = ++foo.begin();
    //创建一个 insert_iterator 迭代器
    //std::insert_iterator< std::list<int> > insert_it(foo, it);
    std::insert_iterator< std::list<int> > insert_it = inserter(foo, it);
    //向 foo 容器中插入元素
    insert_it = 1;
    insert_it = 2;
    insert_it = 3;
    insert_it = 4;
    //输出 foo 容器存储的元素
    for (std::list<int>::iterator it = foo.begin(); it != foo.end(); ++it)
        std::cout << *it << ' ';
    return 0;
}
```

```
5 1 2 3 4 5
```

需要注意的是，如果 insert_iterator 迭代器的目标容器为关联式容器，由于该类型容器内部会自行对存储的元素进行排序，因此我们指定的插入位置只起到一个提示的作用，即帮助关联式容器从指定位置开始，搜索正确的插入位置。但是，如果提示位置不正确，会使的插入操作的效率更加糟糕。

5、流迭代器适配器（stream_iterator）

流迭代器也是一种迭代器适配器，不过和之前讲的迭代器适配器有所差别，它的操作对象不再是某个容器，而是流对象。即通过流迭代器，我们可以读取指定流对象中的数据，也可以将数据写入到流对象中。通常情况下，我们经常使用的 cin、cout 就属于流对象，其中 cin 可以获取键盘输入的数据，cout 可以将指定数据输出到屏幕上。除此之外，更常见的还有文件 I/O 流等。

介于流对象又可细分为输入流对象（istream）和输出流对象（ostream），C++ STL 标准库中，也对应的提供了 2 类流迭代

器：

- 将绑定到输入流对象的迭代器称为输入流迭代器（istream_iterator），其可以用来读取输入流中的数据；
- 将绑定到输出流对象的迭代器称为输出流迭代器（ostream_iterator），其用来将数据写入到输出流中。

5.1输入流迭代器（istream_iterator)

输入流迭代器用于直接从指定的输入流中读取元素，该类型迭代器本质上就是一个输入迭代器，这意味着假设 p 是一个输入流迭代器，则其只能进行 ++p、p++、*p 操作，同时输入迭代器之间也只能使用 == 和 != 运算符。

实际上，输入流迭代器的底层是通过重载 ++ 运算符实现的，该运算符内部会调用operator >>读取数据。也就是说，假设 iit 为输入流迭代器，则只需要执行 ++iit 或者 iit++，即可读取一个指定类型的元素。

创建输入流迭代器的方式有 3 种，分别为：

（1）调用 istream_iterator 模板类的默认构造函数，可以创建出一个具有结束标志的输入流迭代器。要知道，当我们从输入流中不断提取数据时，总有将流中数据全部提取完的那一时刻，这一时刻就可以用此方式构建的输入流迭代器表示。

```
std::istream_iterator<double> eos;
```

由此，即创建了一个可读取 double 类型元素，并代表结束标志的输入流迭代器。

（2）除此之外，还可以创建一个可用来读取数据的输入流迭代器，比如：

```
std::istream_iterator<double> iit(std::cin);
```

这里创建了一个可从标准输入流 cin 读取数据的输入流迭代器。值得注意的一点是，通过此方式创建的输入流迭代器，其调用的构造函数中，会自行尝试去指定流中读取一个指定类型的元素。

（3）istream_iterator 模板类还支持用已创建好的 istream_iterator 迭代器为新建 istream_iterator 迭代器初始化，例如，在上面 iit 的基础上，再创建一个相同的 iit2 迭代器：

```
std::istream_iterator<double> iit2(iit1);
```

由此，就创建好了一个和 iit1 完全相同的输入流迭代器。

下面程序演示了输入流迭代器的用法：

```
#include <iostream>
#include <iterator>
using namespace std;
int main() {
    //用于接收输入流中的数据
    double value1, value2;
    cout << "请输入 2 个小数：";
    //创建表示结束的输入流迭代器
    istream_iterator<double> eos;
    //创建一个可逐个读取输入流中数据的迭代器，同时这里会让用户输入数据
    istream_iterator<double> iit(cin);
    //判断输入流中是否有数据
    if (iit != eos) {
        //读取一个元素，并赋值给 value1
        value1 = *iit;
    }
    //如果输入流中此时没有数据，则用户要输入一个；反之，如果流中有数据，iit 迭代器后移一位，做读取下一个元素做准备
    iit++;
    if (iit != eos) {
        //读取第二个元素，赋值给 value2
        value2 = *iit;
    }
    //输出读取到的 2 个元素
    cout << "value1 = " << value1 << endl;
    cout << "value2 = " << value2 << endl;
    return 0;
}
请输入 2 个小数：1.2 2.3
value1 = 1.2
value2 = 2.3
```

注意，只有读取到 EOF 流结束符时，程序中的 iit 才会和 eos 相等。另外，Windows 平台上使用 Ctrl+Z 组合键输入 ^Z 表示 EOF 流结束符，此结束符需要单独输入，或者输入换行符之后再输入才有效。

5.2输出流迭代器（ostream_iterator)

和输入流迭代器恰好相反，输出流迭代器用于将数据写到指定的输出流（如 cout）中。另外，该类型迭代器本质上属于输出迭代器，假设 p 为一个输出迭代器，则它能执行 ++p、p++、*p=t 以及 *p++=t 等类似操作。其次，输出迭代器底层是通过重载赋值（=）运算符实现的，即借助该运算符，每个赋值给输出流迭代器的元素都会被写入到指定的输出流中。

ostream_iterator 模板类中也提供了 3 种创建 ostream_iterator 迭代器的方法。

（1）通过调用该模板类的默认构造函数，可以创建了一个指定输出流的迭代器：

```
std::ostream_iterator<int> out_it(std::cout);
```

由此，我们就创建了一个可将 int 类型元素写入到输出流（屏幕）中的迭代器。

（2）在第一种方式的基础上，还可以为写入的元素之间指定一个分隔符，例如：

```
std::ostream_iterator<int> out_it(std::cout, ",");
```

和第一种写入方式不同之处在于，此方式在向输出流写入 int 类型元素的同时，还会附带写入一个逗号（,）。

（3）另外，在创建输出流迭代器时，可以用已有的同类型的迭代器，为其初始化。例如，利用上面已创建的 out_it，再创建一个完全相同的 out_it1：

```
std::ostream_iterator<int> out_it1(out_it);
```

下面程序演示了 ostream_iterator 输出流迭代器的功能：

```
#include <iostream>
#include <iterator>
#include <string>
using namespace std;
int main() {
    //创建一个输出流迭代器
    ostream_iterator<string> out_it(cout);
    //向 cout 输出流写入 string 字符串
    *out_it = "http://c.biancheng.net/stl/";
    cout << endl;
    //创建一个输出流迭代器，设置分隔符 ,
    ostream_iterator<int> out_it1(cout, ",");
    //向 cout 输出流依次写入 1、2、3
    *out_it1 = 1;
    *out_it1 = 2;
    *out_it1 = 3;
    return 0;
}
```

```
http://c.biancheng.net/stl/
1,2,3,
```

在实际场景中，输出流迭代器常和 copy() 函数连用，即作为该函数第 3 个参数。比如：

```
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>    // std::copy
using namespace std;
int main() {
    //创建一个 vector 容器
    vector<int> myvector;
    //初始化 myvector 容器
    for (int i = 1; i < 10; ++i) {
        myvector.push_back(i);
    }
    //创建输出流迭代器
    std::ostream_iterator<int> out_it(std::cout, ", ");
    //将 myvector 容器中存储的元素写入到 cout 输出流中
    std::copy(myvector.begin(), myvector.end(), out_it);
    return 0;
}
```

```
1, 2, 3, 4, 5, 6, 7, 8, 9,
```

6、流缓冲区迭代器（streambuf_iterator)

我们知道在 C++ STL 标准库中，流迭代器又细分为输入流迭代器和输出流迭代器，流缓冲区迭代器也是如此，其又被细分为输入流缓冲区迭代器和输出流缓冲区迭代器：

- 输入流缓冲区迭代器（istreambuf_iterator）：从输入流缓冲区中读取字符元素；
- 输出流缓冲区迭代器（ostreambuf_iterator）：将连续的字符元素写入到输出缓冲区中。

流缓冲区迭代器和流迭代器最大的区别在于，前者仅仅会将元素以字符的形式（包括 char、wchar_t、char16_t 及 char32_t 等）读或者写到流缓冲区中，由于不会涉及数据类型的转换，读写数据的速度比后者要快。

6.1输入流缓冲区迭代器（istreambuf_iterator)

istreambuf_iterator 输入流缓冲区迭代器的功能是从指定的流缓冲区中读取字符元素。值得一提的是，该类型迭代器本质是一个输入迭代器，即假设 p 是一个输入流迭代器，则其只能进行 ++p、p++、*p 操作，同时迭代器之间也只能使用 == 和 != 运算符。

创建输入流缓冲区迭代器的常用方式，有以下 2 种：

（1）通过调用 istreambuf_iterator 模板类中的默认构造函数，可以创建一个表示结尾的输入流缓冲区迭代器。要知道，当我们从流缓冲区中不断读取数据时，总有读取完成的那一刻，这一刻就可以用此方式构建的流缓冲区迭代器表示。

```
std::istreambuf_iterator<char> end_in;
```

其中，<> 尖括号中用于指定从流缓冲区中读取的字符类型。

（2）当然，我们还可以指定要读取的流缓冲区，比如：

```
std::istreambuf_iterator<char> in{ std::cin };
```

除此之外，还可以传入流缓冲区的地址，比如：

```
std::istreambuf_iterator<char> in {std::cin.rdbuf()};
```

其中，rdbuf() 函数的功能是获取指定流缓冲区的地址。

下面程序演示了输入流缓冲区迭代器的用法：

```
#include <iostream>     // std::cin, std::cout
#include <iterator>     // std::istreambuf_iterator
#include <string>       // std::string
using namespace std;
int main() {
    //创建结束流缓冲区迭代器
    istreambuf_iterator<char> eos;
    //创建一个从输入缓冲区读取字符元素的迭代器
    istreambuf_iterator<char> iit(cin);
```

```
    string mystring;
    cout << "向缓冲区输入元素：\n";
    //不断从缓冲区读取数据，直到读取到 EOF 流结束符
    while (iit != eos) {
        mystring += *iit++;
    }
    cout << "string：" << mystring;
    return 0;
}
向缓冲区输入元素：
abc ↙
^Z ↙
string：
abc
```

注意，只有读取到 EOF 流结束符时，程序中的 iit 才会和 eos 相等。在 Windows 平台上，使用 Ctrl+Z 组合键输入 ^Z 表示 EOF 流结束符，此结束符需要单独输入，或者输入换行符之后再输入才有效。

6.2输出流缓冲区迭代器（ostreambuf_iterator)

和 istreambuf_iterator 输入流缓冲区迭代器恰恰相反，ostreambuf_iterator 输出流缓冲区迭代器用于将字符元素写入到指定的流缓冲区中。实际上，该类型迭代器本质上是一个输出迭代器，这意味着假设 p 为一个输出迭代器，则它仅能执行 ++p、p++、*p=t 以及 *p++=t 操作。另外，和 ostream_iterator 输出流迭代器一样，istreambuf_iterator 迭代器底层也是通过重载赋值（=）运算符实现的。换句话说，即通过赋值运算符，每个赋值给输出流缓冲区迭代器的字符元素，都会被写入到指定的流缓冲区中。

在此基础上，创建输出流缓冲区迭代器的常用方式有以下 2 种：

（1）通过传递一个流缓冲区对象，即可创建一个输出流缓冲区迭代器，比如：

```
std::ostreambuf_iterator<char> out_it (std::cout);
```

同样，尖括号 <> 中用于指定要写入字符的类型，可以是 char、wchar_t、char16_t 以及 char32_t 等。

（2）还可以借助 rdbuf()，传递一个流缓冲区的地址，也可以成功创建输出流缓冲区迭代器：

```
std::ostreambuf_iterator<char> out_it (std::cout.rdbuf());
```

下面程序演示了输出流缓冲区迭代器的用法：

```
#include <iostream>     // std::cin, std::cout
#include <iterator>     // std::ostreambuf_iterator
#include <string>       // std::string
#include <algorithm>    // std::copy
int main() {
    //创建一个和输出流缓冲区相关联的迭代器
    std::ostreambuf_iterator<char> out_it(std::cout); // stdout iterator
    //向输出流缓冲区中写入字符元素
    *out_it = 'S';
    *out_it = 'T';
    *out_it = 'L';
    //和 copy() 函数连用
    std::string mystring("\nhttp://c.biancheng.net/stl/");
    //将 mystring 中的字符串全部写入到输出流缓冲区中
    std::copy(mystring.begin(), mystring.end(), out_it);
    return 0;
}
STL
http://c.biancheng.net/stl/
```

7、移动迭代器（move_iterator)

move_iterator 迭代器适配器，又可简称为移动迭代器，其可以实现以移动而非复制的方式，将某个区域空间中的元素移动至另一个指定的空间。

举个例子，前面讲了 vector 容器，该类型容器支持如下初始化的方式。

```
#include <iostream>
#include <vector>
#include <list>
#include <string>
using namespace std;
int main()
{
    //创建并初始化一个 vector 容器
    vector<string> myvec{ "STL","Python","Java" };
    //再次创建一个 vector 容器，利用 myvec 为其初始化
    vector<string>othvec(myvec.begin(), myvec.end());

    cout << "myvec:" << endl;
    //输出 myvec 容器中的元素
    for (auto ch : myvec) {
        cout << ch << " ";
    }
    cout << endl << "othvec:" << endl;
    //输出 othvec 容器中的元素
    for (auto ch : othvec) {
        cout << ch << " ";
    }
    return 0;
}
myvec:
STL Python Java
othvec:
STL Python Java
```

注意程序第 11 行，初始化 othvec 容器是通过复制 myvec 容器中的元素实现的。也就是说，othvec 容器从 myvec 容器中复

制了一份 "STL"、"Python"、"Java" 并存储起来，此过程不会影响 myvec 容器。那么，如果不想采用复制的方式，而就是想 myvec 容器中存储的元素全部移动到 othvec 容器中，该怎么办呢？没错，就是采用移动迭代器。

实现 move_iterator 移动迭代器的模板类定义如下：

```
template <class Iterator>
    class move_iterator;
```

可以看到，在使用此迭代器时，需要传入一个基础迭代器 Iterator。

注意，此基础迭代器的类型虽然没有明确要求，但该模板类中某些成员方法的底层实现，需要此基础迭代器为双向迭代器或者随机访问迭代器。也就是说，如果指定的 Iterator 类型仅仅是输入迭代器，则某些成员方法将无法使用。

move_iterator 模板类中，提供了 4 种创建 move_iterator 迭代器的方法。

（1）通过调用该模板类的默认构造函数，可以创建一个不指向任何对象的移动迭代器。比如：

```
//将 vector 容器的随机访问迭代器作为新建移动迭代器底层使用的基础迭代器
typedef std::vector<std::string>::iterator Iter;
//调用默认构造函数，创建移动迭代器
std::move_iterator<Iter>mIter;
```

由此，我们就创建好了一个 mIter 移动迭代器，该迭代器底层使用的是 vector 容器的随机访问迭代器，但这里没有为此基础迭代器明确指向，所以 mIter 迭代器也不指向任何对象。

（2）当然，在创建 move_iterator 迭代器的同时，也可以为其初始化。比如：

```
//创建一个 vector 容器
std::vector<std::string> myvec{ "one","two","three" };
//将 vector 容器的随机访问迭代器作为新建移动迭代器底层使用的基础迭代器
typedef std::vector<std::string>::iterator Iter;
//创建并初始化移动迭代器
std::move_iterator<Iter>mIter(myvec.begin());
```

这里，我们创建了一个 mIter 移动迭代器，同时还为底层使用的随机访问迭代器做了初始化，即令其指向 myvec 容器的第一个元素。

（3）move_iterator 模板类还支持用已有的移动迭代器初始化新建的同类型迭代器，比如，在上面创建好 mIter 迭代器的基础上，还可以向如下这样为新建的移动迭代器初始化：

```
std::move_iterator<Iter>mIter2(mIter);
//还可以使用 = 运算符，它们是等价的
//std::move_iterator<Iter>mIter2 = mIter;
```

这样创建的 mIter2 迭代器和 mIter 迭代器完全一样。也就是说，mIter2 底层会复制 mIter 迭代器底层使用的基础迭代器。

（4）以上 3 种创建 move_iterator 迭代器的方式，其本质都是直接调用 move_iterator 模板类中的构造方法实现的。除此之外，C++ STL 标准库还提供了一个 make_move_iterator() 函数，通过调用此函数可以快速创建一个 move_iterator 迭代器。

```
typedef std::vector<std::string>::iterator Iter;
std::vector<std::string> myvec{ "one","two","three" };
//将 make_move_iterator() 的返回值赋值给同类型的 mIter 迭代器
std::move_iterator<Iter>mIter = make_move_iterator(myvec.begin());
```

示例：

```
#include <iostream>
#include <vector>
#include <list>
#include <string>
using namespace std;
int main()
{
    //创建并初始化一个 vector 容器
    vector<string> myvec{ "STL","Python","Java" };
    //再次创建一个 vector 容器，利用 myvec 为其初始化
    vector<string>othvec(make_move_iterator(myvec.begin()), make_move_iterator(myvec.end()));

    cout << "myvec:" << endl;
    //输出 myvec 容器中的元素
    for (auto ch : myvec) {
        cout << ch << " ";
    }
    cout << endl << "othvec:" << endl;
    //输出 othvec 容器中的元素
    for (auto ch : othvec) {
        cout << ch << " ";
    }
    return 0;
}
myvec:
othvec:
STL Python Java
```

通过和上面程序做对比不难看出它们的区别，由于程序第 11 行为 othvec 容器初始化时，使用的是移动迭代器，其会将 myvec 容器中的元素直接移动到 othvec 容器中。注意，即便通过移动迭代器将容器中某区域的元素移动到了其他容器中，该区域内仍可能残留有之前存储的元素，但这些元素是不能再被使用的，否则极有可能使程序产生各种其他错误。

和其他迭代器适配器一样，move_iterator 模板类中也提供有 base() 成员方法，通过该方法，我们可以获取到当前移动迭代器底层所使用的基础迭代器。

```
#include <iostream>
#include <vector>
#include <list>
#include <string>
using namespace std;
int main()
```

```
{
    typedef std::vector<std::string>::iterator Iter;

    //创建并初始化一个 vector 容器
    vector<std::string> myvec{ "STL","Java","Python" };
    //创建 2 个移动迭代器
    std::move_iterator<Iter>begin = make_move_iterator(myvec.begin());
    std::move_iterator<Iter>end = make_move_iterator(myvec.end());
    //以复制的方式初始化 othvec 容器
    vector <std::string> othvec(begin.base(), end.base());

    cout << "myvec:" << endl;
    //输出 myvec 容器中的元素
    for (auto ch : myvec) {
        cout << ch << " ";
    }
    cout << endl << "othvec:" << endl;
    //输出 othvec 容器中的元素
    for (auto ch : othvec) {
        cout << ch << " ";
    }
    return 0;
}
myvec:
STL Java Python
othvec:
STL Java Python
```

来自 <http://c.biancheng.net/stl/iterator_adaptor/>