标准库没给每个容器都定义成员函数来实现这些操作，而是定义了一组泛型算法。“泛型”是因为它们可以用于不同类型的元素和多种容器类型（不仅宝库标准库类型，还包括内置的数组类型）。

# 1概述：

大多数定义在头文件algorithm中，还在numeric中定义了一组数值泛型算法。一般来说这些算法不直接操作容器，而是遍历由两个迭代器指定的一个元素范围来进行操作，运行于迭代器之上，执行迭代器的操作。算法永远不会改变底层容器的大小，可能改变元素中的值，也可能移动元素，但不会添加删除。

## 标准库算法find：

传递给find的前两个参数是表示元素范围的迭代器，第三个参数是一个值。find将范围中每个元素与给定值进行比较它返回指向第一个等于给定值的元素的迭代器；如果范围中无匹配元素，则find返回第二个参数来表示搜索失败。由于指针就像内置数组上的迭代器一样，我们可以用find在数组中查找值：

int ia[] = {27,210,12,47,109,83};

int val = 83;

int \*result = find(begin(ia),end(ia),val);

虽然迭代器的使用令算法不依赖于容器类型，但大多数算法都使用了一个或多个元素类型上的操作。

# 2.初识泛型算法：

除了少数，标准库算法都对一个范围内的元素进行操作。我们将此元素范围成为“输入范围”，接受输入范围的算法总是使用前两个参数来表示此范围。理解算法的最基本方法就是了解它们是否读取元素、改变元素或是重排元素顺序。

## 2.1只读算法：

有find、count、accomulate、equal等。

2.1.1 accomulate算法：

accomulate定义在头文件numeric中，用于给数组求和，第三个参数是和的初始值，第三个参数的类型决定了函数中使用哪个加法运算符以及返回值类型，序列中元素类型必须与第三个参数匹配或能转换为第三个参数的类型，第三个参数为求和起点。对于只读算法通常最好使用cbegin()和end()，但如果计划使用算法返回的迭代器来改变元素的值，就需要使用begin()和end()的结果作为参数。

string sum = accumulate(v.cbegin(),v.cend(),string(“hehe”));//初识为hehe，后来逐渐往上加

string sum = accumulate(v.cbegin(),v.cend(), “hehe”);//错误，const char\*上没定义+运算符

2.1.2equal算法：

equal用于确定两个序列是否保存相同的值。接受三个迭代器，前两个表示第一个序列中的元素范围，第三个表示第二个序列的首元素。可以调用equal来比较两个不同类型的容器中的元素，元素类型也可以不同，只要能用==来比较两个元素类型即可。

只接受一个单一迭代器来表示第二个序列的算法，都假定第二个序列至少与第一个序列一样长，我们要确保不会试图访问不存在的元素。

## 2.2写容器元素的算法：

使用时确保序列原大小至少不少于我们要求算法写入的元素数目。

2.2.1fill算法：

接受一对迭代器表示一个范围，还接受一个值作为第三个参数，将这个值赋予输入序列中的每个元素。

2.2.2fill\_n算法：

一些算法接受一个迭代器来指出一个单独的目的位置，这些算法将新值赋予一个序列中的元素，该序列从目的位置迭代器指向的元素开始。

fille\_n接受一个单迭代器、一个计数值和一个值。注意计数值一定要有效，不能访问不存在的值。

fill\_n(v.begin(),v.size(),0);//将所有元素置为0

2.2.3back\_insert：

插入迭代器是一种向容器中添加元素的迭代器，当我们通过一个迭代器向容器元素赋值时，值被赋予迭代器指向的元素。当我们通过插入迭代器赋值时，一个与赋值号右侧值相等的元素被添加到容器中。

back\_inserter定义在头文件iterator中，back\_inserter接受一个指向容器的引用，返回一个与该容器绑定的插入迭代器。当我们通过此迭代器赋值时，赋值运算符会调用push\_back将一个具有给定值的元素添加到容器中。

vector<int> vec;

auto it = back\_inserter(vec);//通过它赋值将元素添加到vec中

\*it = 42;

fill\_n(back\_inserter(vec),10,0);//添加10个0到vec中，调用push\_back

2.2.4copy算法：

多个算法都提供所谓的拷贝，这些算法计算新元素的值，但不会将它们放置在输入序列的末尾，而是创建一个新序列保存这些结果。

向目的位置迭代器指向的输出序列中的元素写入数据的算法。此算法接受三个迭代器，前两个表示一个输入范围，第三个表示目的序列的起始位置。此算法将输入范围中的元素拷贝到目的序列中，传递给copy的目的序列至少要包含与输入序列一样多的元素。返回的是其目的位置迭代器（递增后）的值。

int a1[] = {0,1,2,3,4,5,6,7,8,9};

int a2[sizeof(a1)/sizeof(\*a1)];//牛逼

auto ret = copy(begin(a1),end(a1),a2);//ret指向拷贝到a2的尾元素之后的位置

2.2.5replace算法：

读入一个序列，并将其中所有等于给定值的元素都改为另一个值。此算法接受4个参数：前两个是迭代器，表示输入序列，后两个一个是要搜索的值，另一个是新值，将等于第一个值的元素换成第二个值。

2.2.6replace\_copy算法：

在replace上保留原序列不变，在replace的基础上接受额外的第三个迭代器参数，指出调整后序列的保存位置。

replace\_copy(ilist.begin(),ilist.end(),back\_inserter(ivec),0,42);

## 2.3重排元素的算法：

2.3.1sort：

利用元素类型的<运算符来实现排序。有两个参数：起始位置、终止位置。

2.3.2unique：

重排vector，使不重复的元素出现在vector的开始部分。返回最后一个不重复元素之后位置的迭代器。使用前必须先排序！由于算法不能执行容器的操作，我们使用vector的erase成员来完成真正的删除操作。

auto end\_unique = unique(words.begin(),wprds.end());

words.erase(end\_unique,words.end());

# 3定制操作：

## 3.1向算法传递函数：

3.1.1谓词：

谓词是一个可调用的表达式，其返回结果是一个能用作条件的值。用来重载算法，标准库算法所使用的谓词分两类：一元谓词（只能接受单一参数）、二元谓词。元素类型必须能转换成谓词的参数类型。

3.1.2stable\_sort算法：

这种稳定排序算法维持相等元素的原有顺序。

3.1.3partition算法：

接受一个谓词，对容器内容进行划分，使得谓词为true的值排在容器的前半部分，谓词为false的值排在后半部分。返回一个迭代器，指向最后一个使谓词为true的元素之后的位置。

## 3.2lambda表达式：

有时我们希望进行的操作需要更多参数，超出了算法对谓词的限制。如果在编写划分序列的谓词时，可以不必为每个可能的大小都编写一个独立的谓词，显然更有实际价值。

3.2.1find\_if算法：

查找第一个具有特定大小的元素。find\_if算法接受一对迭代器，表示一个范围，第三个参数是一个谓词。该算法对输入序列中的每个元素调用给定的这个谓词。它返回第一个使谓词返回非0的元素，如果不存在这样的元素，返回尾末迭代器。

3.2.2介绍lambda：

我们可以向一个算法传递任何类别的可调用对象。对于一个对象或一个表达式，如果可以对其使用调用运算符，则它为可调用的。如果e是一个可调用的表达式，则我们可以写：e(args)，其中args是一个逗号分隔的一个或多个参数的列表。

可调用对象分四种：①函数，②函数指针，③重载了函数调用运算符的类，④lambda表达式。

一个lambda表达式表示一个可调用的代码单元，可以理解为一个未命名的内联函数。一个lambda具有一个返回类型、一个参数列表、一个函数体，但与函数不同，它可能定义在函数内部。一个lambda具有如下形式：

[capture list](parameter list) -> return type {function body}

capture list（捕获列表）是一个lambda所在函数中定义的局部变量的列表（通常为空）；parameter list、return type、function body分别表示返回类型、参数列表、函数体。注意：lambda必须使用尾置返回来指定类型。参数列表和返回类型可忽略，捕获列表和函数体必须包含。lambda的调用方式与普通函数的调用方式相同，都是使用调用运算符。如果lambda函数体包含任何单一return语句之外的内容，且未指定返回类型，则返回void。

auto f = []{return 42;}//定义了可调用对象f，它不接受参数，返回42

cout<<f()<<endl;//打印42

与普通函数不同，lambda不能有默认参数，因此一个lambda调用的实参数目永远与形参数目相等。一旦初始化完毕，就可以执行函数体了。

[](const string &a,const string &b)

{return a.size() < b.size();}

可以使用此lambda来调用stable\_sort：

stable\_sort(words.begin(),words.end(),[](const string &a,const string &b){return a.size() < b.size();});//形参a和b就是words.begin()和words.end()中的元素

虽然一个lambda可以出现在一个函数中，使用其局部变量，但它只能使用那些明确指明的变量。一个lambda通过将局部变量包含在其捕获列表中来指出将会使用这些变量。捕获列表指引lambda在其内部包含访问局部变量所需信息。

[sz1,sz2](const string &a)

{return a.size() >= sz1 && a.size() <= s;}

auto wc = find\_if(words.begin(),words.end(),[sz](const string &a){return a.size() > sz;});

3.2.3for\_each算法：

此算法接受一个可调用对象，并对输入序列中每个元素调用此对象。

for\_each(wc,words.end(),[](const string &a){cout<<s<<” ”;});

捕获列表只用于局部非static变量，lambda可以直接使用局部static变量和它所在函数之外声明的名字。

## 3.3lambda捕获和返回：

当定义一个lambda时，编译器生成一个与lambda对应的新的（未命名的）类类型。当向一个函数传递一个lambda时，同时定义了一个新类型和该类型的一个对象：传递的参数是此编译器生成的类类型的未命名对象。

默认的情况下，从lambda生成的类都包含一个对应该lambda所捕获的变量的数据成员。类似任何普通类的数据成员，lambda的数据成员也在lambda创建时被初始化。

类似参数传递，变量的捕获方式也可以是值或引用，采用值捕获的前提是变量可以拷贝。

3.3.1值捕获：

与参数不同，被捕获的变量的值是在lambda创建时拷贝，而不是调用时拷贝。非指针类型通常采用该方式。

3.3.2引用捕获：

当以引用方式捕获一个变量时，必须保证在lambda执行时变量是存在的。如果我们捕获一个指针或迭代器，或采用引用捕获方式，就必须确保在lambda执行时，绑定到迭代器、指针、引用的对象仍然存在，而且保证对象具有预期的值。

void biggies(vector<string> &words,vector<string>::size\_type sz,ostream &os = cout,char c = ‘ ’)

{

………………

for\_each(words.begin(),words.end(),[&os,c](const string &s){os<<s<<c;})//不能拷贝，因此捕获os的唯一方法就是捕获其引用

}

3.3.3隐式捕获：

除了显式列出我们希望使用的来自所在函数的变量之外，还可以让编译器根据lambda体中的代码来推断我们要使用哪些变量。为了指示编译器推断捕获列表，应在捕获列表中写一个&或=。&告诉编译器采用捕获引用方式，=告诉编译器使用值捕获方式。

wc = find\_if(words.begin(),words.end(),[=](const string &s){return s.size() > sz;})

如果我们希望对一部分变量采用值捕获，其他变量采用引用捕获，可以混合使用显式捕获和隐式捕获，而且要求第一个元素必须是一个&或=。显式捕获的变量必须使用与隐式捕获不同的方式，如果隐式捕获是引用，则显示捕获必须采用值方式。

for\_each(words.begin(),words.end(),[=,&os](const string &s){os<<s<<c;})；

|  |  |
| --- | --- |
| lambda捕获列表 | |
| [] | 空捕获列表 |
| [names] | names是一个都好分隔的名字列表，默认是值拷贝 |
| [&] | 隐式捕获列表，采用引用捕获方式 |
| [=] | 显式捕获列表，采用值捕获方式 |
| [&,identifier\_list] | identifier\_list是一个都好分隔的列表，包含0个或多个来自所在函数的变量，这些变量采用值捕获方式而任何隐式捕获的变量都采用引用捕获的方式捕获，identifier\_list中的名字前面不能使用& |
| [=,identifier\_list] | 与上面的相反 |

3.3.4可变lambda：

默认情况下，对于一个值被拷贝的变量，lambda不会改变其值。如果我们希望能改变一个被捕获的变量的值，就必须在参数列表首加上关键字mutable。

auto f = [v1] () mutable {return ++v1;};

3.3.5transform算法：

接受三个迭代器和一个可调用对象。前两个迭代器表示输入序列，第三个迭代器表示目的位置。算法对输入序列中每个元素调用可调用对象，并将其结果写到目的位置，目的位置迭代器与表示输入序列开始位置的迭代器可以是相同的，替代自己。

transform(v.begin(),v.end(),v.begin(),[](int i){return I < 0? –i: i;});

3.3.6指定lambda返回类型：

当我们需要为一个lambda定义返回类型时，必须使用尾置返回类型。

transform(v.begin(),v.end(),v.begin(),

[](int i) -> int

{if(i < 0) return –i; else return i;});//多语句

3.3.7count\_if算法：

接受一对迭代器，表示一个输入范围，还接受一个谓词，会对范围中每个元素执行。返回一个计数值，表示谓词有多少次为真。

## 3.4参数绑定：

如果我们需要在很多地方使用相同的操作或一个操作需要很多语句才能完成，通常应该定义一个函数，而不是多次编写相同的lambda表达式。3

如果一个lambda的捕获列表为空，通常可以用函数来代替它，就像前面sort中的com一样。

但如果lambda的捕获列表不为空，就不是那么容易了。

在头文件functional中，可以将bind函数看作一个通用的函数适配器，它接受一个可调用对象，生成一个新的可调用对象来”适应“原对象的参数列表。可以用blind修正参数的值、绑定给定可调用对象中的参数、重新安排其顺序。格式为：

auto newCallable = bind(callable,arg\_list);

newCallable本身是一个可调用对象，arg\_list是一个逗号分隔的参数列表，对应给定的callable的参数。当我们调用newCallable时，newCallable会调用callable，并传递给它arg\_list中的参数。

arg\_list中的参数可能包含形如\_n的名字，这些参数是占位符，表示newCallable的参数，它们占据了传递给newCallable的参数的”位置“。数值n表示生成的可调用对象中参数的位置，比如\_1为第一个参数。

bool check\_size(const &s,string::size\_type sz)

{return s.size() >= sz;}

auto check6 = bind(check\_size,\_1,6);

bool b = check6(s);//相当于把函数需要传入的参数数目减小了

auto wc = find\_if(words.begin(),words.end(),bind(check\_size,\_1,sz));

auto g = bind(f,a,b,\_2,c,\_1);//给参数换位置！

名字\_n定义在一个名为placeholders的命名空间中，这个命名空间本身定义在std命名空间中，为了使用这些名字，两个空间都要写上。

using std::placeholders::\_1;

using namespace std::placeholders;

默认情况下，bind的那些不是占位符的参数以值的形式被拷贝到bind返回的可调用对象中。如果我们希望传递给bind一个对象而又不拷贝它，就必须使用标准库ref函数,该函数在functional头文件中。函数返回一个对象，包含给定的引用，次对象是可以拷贝的。标准库还有一个cref函数，生成一个保存const引用的类。

for\_each(words.begin(),words.end(),bind(ref(os),\_1,’’));

# 4.再探迭代器：

在头文件iterator中定义了几种迭代器：

①插入迭代器：迭代器被绑定到一个容器上，用来向容器插入元素

②流迭代器：迭代器被绑定到输入输出流上，可用来遍历所关联的IO流

③反向迭代器：除了forward\_list外的标准库容器都有。

④移动迭代器：这些专用的迭代器不是拷贝其中的元素，而是移动它们。

## 4.1插入迭代器：

是一种迭代器适配器，它接受一个容器，生成一个迭代器，能实现向给定容器添加元素。

|  |  |
| --- | --- |
| 插入迭代器操作 | |
| it = t; | 在it指定的当前位置插入值t。依赖于插入迭代器的不同种类，此赋值会分别调用push\_back、push\_front、insert。 |
| \*it,++it,it++ | 这些操作虽然存在，但不会生效，每个操作都返回it。 |

插入迭代器有三种类型：

①back\_inserter：创建一个使用push\_back的迭代器。

②front\_inserter：创建一个使用push\_front的迭代器。

③inserter：创建一个使用insert的迭代器。此函数接受第二个参数，这个参数必须是一个指向给定容器的迭代器，元素将被插入到给定迭代器所表示的元素之前。

当调用it = inserter(c,iter)时，得到一个迭代器，接下来使用它时，会将元素插入到iter原来所指向的元素之前的位置。\*it = val;等同于：it = c.insert(it,val);++it;

list<int> lst = {1,2,3,4};

list<int> lst2,lst3;

copy(lst.cbegin(),lst.cend(),front\_inserter(lst2));

copy(lst.cbegin(),lst.cend(),inserter(lst3,lst3.begin()));

上例中当调用front\_inserter(c)时，得到一个插入迭代器，接下来会调用push\_front。当每个元素被插入到容器c中时，它变为c的新的首元素。因此，front\_inserter生成的迭代器会将插入的元素序列的顺序颠倒过来，而inserter和back\_inserter则不会。

## 4.2unique\_copy算法：

接受三个迭代器，第三个表示拷贝不重复元素的目的位置

## 4.3iostream迭代器：

虽然iostream类型不是容器，但标准库定义了可以用于这些IO类型对象的迭代器。这些迭代器将他们对应的流当作一个特定类型的元素序列来处理。通过使用流迭代器，可以用泛型算法从流对象读取、写入数据。当创建一个流迭代器时，必须指定迭代器将要读写的对象类型。

4.3.1istream\_iterator操作：

一个istream\_iterator使用>>来读取流，要求要读取的类型必须定义了输入运算符。

istream\_iterator<int> in\_iter(cin),eof;//eof自动指向尾后

vector<int> vec(in\_iter,eof);//从迭代器范围构造vec

|  |  |
| --- | --- |
| istream\_iterator操作 | |
| istream\_iterator<T> in(is); | in从输入流is读取类型为T的值 |
| istream\_iterator<T> end; | 读取类型为T的值，默认初始化迭代器，表示尾后位置 |
| in1 == in2 | in1和in2必须读取相同类型 |
| in1 != in2 |
| \*in | 返回从流中读取的值 |
| in->mem | 与(\*in).mem的含义相同 |
| ++in,in++ | 使用元素类型所定义的>>运算符从输入流中读取下一个值。 |

由于算法使用迭代器操作来处理数据，而流迭代器又至少支持某些迭代器操作，因此我们至少可以用某些算法来操作流迭代器。

当我们将一个istream\_iterator绑定到一个流时，标准库并不保证迭代器立即从流读取数据，标准库只保证在我们第一次解引用迭代器之前，从流中读取数据的操作已经完成了。如果我们创建了一个istream\_iterator，没有使用就销毁了，或者我们正在从两个不同的对象同步读取一个流，那么何时读取可能就很重要了。

4.3.2ostream\_iterator操作：

一个istream\_iterator使用<<来写流，必须将ostream\_terator绑定到一个指定的流，不允许空的或表示尾后位置的ostream\_iterator。

|  |  |
| --- | --- |
| ostream\_iterator | |
| ostream\_iterator<T> out(os); | out将类型为T的值写到输出流os中 |
| ostream\_iterator<T> out(os,d); | out将类型为T的值写到输出流os中，每个值后面都输出一个d。d必须指向一个空字符结尾的字符型数组或字符串字面常量。 |
| out = val; | 用<<运算符将val写入到out所绑定的ostream中。val的类型必须与out可写的类型兼容。 |
| \*out,++out,out++ | 这些运算符是存在的，但不对out做任何事情，每个运算符都返回out。 |

ostream\_iterator<int> out(cout,” ”);

for(auto e:vec)

out = e;

//通过调用copy来打印vec中的元素，这比编写循环更简单。

copy(vec.begin(),vec.end(),out);

## 4.3反向迭代器：

虽然颠倒的递增和递减运算符的含义可能看起来令人混淆，但是这样做使我们可以用算法透明地向前或向后处理容器。

sort(vec.rbegin(),vec.rend());//按逆序排序

流迭代器不支持递减运算，因为不可能在一个六中反向移动，因此不可能从一个forward\_list或一个流迭代器创建反向迭代器。

auto rcomma = find(line.crbegin(),line.crend(),’,’);//在一个逗号分隔的列表中查找最后一个元素

cout<<string(line.crbegin(),rcomma)<<endl;//产生错误

cout<<string(rcomma.base(),line.cend())<<endl;//正确，转换成普通迭代器

反向迭代器的目的是表示元素范围，而这些范围是不对称的，这导致一个重要的结果：当我们从一个普通迭代器初始化一个反向迭代器，或是给一个反向迭代器赋值时，结果迭代器与原迭代器指向的并不是相同的元素。

# 5.泛型算法结构：

任何算法的最基本特征是它要求其迭代器提供哪些操作。算法所要求的迭代器操作可以分为5个迭代器类别，每个算法都会对它的每个迭代器参数指明需提供哪类迭代器。

## 5.1五类迭代器：

|  |  |
| --- | --- |
| 迭代器类别 | |
| 输入迭代器（istream\_iterator） | 只读不写；单遍扫描；只能递增。必须支持：①==、!=，②++，③\*，只会出现在赋值运算符右边④->。 |
| 输出迭代器（ostream\_iterator） | 只写不读；单遍扫描；只能递增。必须支持：①++，②\*，只会出现在赋值运算符左边。 |
| 前向迭代器（forward\_list上的迭代器） | 可读写；多遍扫描；只能递增。①==、!=，②++，③\*，出现在赋值运算符的左边/右边，④->。 |
| 双向迭代器（除了forward\_list其他标准库上的迭代器） | 可读写；多遍扫描；可递增递减。①==、!=，②++、--，③\*，出现在赋值运算符的左边/右边，④->。 |
| 随机访问迭代器 | 可读写；多遍扫描，支持全部迭代器运算。必须支持：①<=、<、>、>=，②+、+=、-、-=，用于迭代器和一个整数，③-，用于两个迭代器上，④下标运算符，⑤==、!=，⑥++、--，⑦\*，⑧-> |

除了输出迭代器之外，一个高层次类别的迭代器支持低层次类别迭代器的所有操作。

输入迭代器\*it++保证是有效的，但递增它可能导致所有其他指向流的迭代器失效，结果就是不能保证输入迭代器的状态可以保存下来并用来访问元素。因此，输入迭代器只能用于单遍扫描算法。

我们只能向一个输出迭代器赋值一次。类似输入迭代器，输出迭代器只能用于单遍扫描算法。用作目的位置的迭代器通常都是输出迭代器。

我们可以保存前向迭代器的状态，使用前向迭代器的算法可以对序列进行多遍扫描。

## 5.2算法形参模式规范：

大多数算法具有以下四种形式之一：①alg(beg,end,other args);②alg(beg,end,dest,other args);③alg(beg,end,beg2,other args);④alg(beg,end,beg2,end2,other args);

dest参数是一个表示算法可以写入的目的位置的迭代器。算法假定：按其需要写入数据，不管写入多少个元素都是安全的。dest也可以是一个ostream\_iterator，会数据写入到一个输出流，不管要写入多少个元素都没有问题。

接受单独的beg2或是beg2和end2的算法用这些迭代器表示第二个输入范围。这些算法通常使用第二个范围中的元素与第一个输入范围结合起来进行运算。接受单独beg2的算法假定从beg2开始的序列与beg和end所表示的范围至少一样大。

## 5.3命名规范：

接受一个元素值的算法通常有另一个不同名的版本，该版本接受一个谓词代替元素值。接受谓词参数的算法都有附加的\_if前缀，注意：这俩不是重载！

find(beg,end,val);

find\_if(beg,end,pred);

写到额外空间的算法都在名字后面附加一个\_copy：

reverse(beg,end);

reverse\_copy(beg,end,dest);

一些算法同时提供\_copy和\_if版本，这些版本接受一个目的位置迭代器和一个谓词：

remove\_copy\_if(v.begin(),v.end(),back\_inserter(v2),[](int i){return I % 2;});

# 6.特定容器算法：

链表类型list和forward\_list定义了几个成员函数形式的算法，应优先使用成员函数版本而不是通用算法。定义了独有的sort、merge、remove、reserve、unique。通用版本的sort要求随机访问迭代器，因此不能用于list和forward\_list，因为这两个类型分别提供双向迭代器和前向迭代器。链表类型定义的其他算法的通用版本可以用于链表，但代价太高。链表特有版本与通用版本间的一个至关重要的区别就是链表版本会改变底层的容器。

|  |  |
| --- | --- |
| list和forward\_list成员函数版本的算法 | |
| lst.merge(lst2); | 将来自于lst2的元素合并入lst，lst和lst2都必须是有序的，合并后，lst2为空。返回void。 |
| lst.merge(lst2,copm); |
| lst.remove(val); | 调用erase删除掉与给定值或一元谓词为真的每个元素，返回void。 |
| lst.remove\_if(pred); |
| lst.reverse(); | 反转，返回void。 |
| lst.sort(); | 排序，返回void。 |
| lst.sort(comp); |
| lst.unique(); | 调用erase删除同一个值的连续拷贝，第一个版本用==，第二个版本用给定的二元谓词，返回void。 |
| lst.unique(pred); |
| lst.splice(p,lst2);或flst.splice\_after(p,lst2); | p是一个指向lst中元素的迭代器，或一个指向flst首前位的迭代器。函数将lst2的所有元素移动到lst中p之前的位置或是flst中p之后的位置。将元素从lst2中删除，lst2的类型必须与lst或flst相同，且不能是同一个链表。 |
| lst.splice(p,lst2,p2);或flst.splice\_after(p,lst2,p2); | p2是一个指向lst2中位置的有效的迭代器。将p2指向的元素移动到lst中，或将p2之后的元素移动到flst中。lst2可以是与lst或flst相同的链表。 |
| lst.splice(p,lst2,b,e);或flst.splice\_after(p,lst2,b,e); | b和e必须表示lst2中合法范围，将给定范围内的元素从lst2移动到lst或flst。lst2与lst（或flst）可以是相同的链表，但p不能指向给定范围中的元素。 |