STL 算法 - kzangv - 博客园

STL 算法

STL算法概述

简介:

STL算法部分主要由头文件<algorithm>,<numeric>,<functional>组成。要使用 STL中的算法函数必须包含头文件<algorithm>，对于数值算法须包含<numeric>，<functional>中则定义了一些模板类，用来声明函数对象

注意:

编译器无法检测出所传递的迭代器是一个无效形式的迭代器,当然也无法给出算法函数错误的提示,因为迭代器并不是真实的类别,它只是传递给函数模板的一种参数格式而已

STL中算法分类:

- 操作对象
  - 直接改变容器的内容
  - 将原容器的内容复制一份,修改其副本,然后传回该副本
- 功能:
  - 非可变序列算法 指不直接修改其所操作的容器内容的算法
  - 可变序列算法 指可以修改它们所操作的容器内容的算法
  - 排序算法 包括对序列进行排序和合并的算法、搜索算法以及有序序列上的集合操作
  - 数值算法 对容器内容进行数值计算

查找算法(13个)：判断容器中是否包含某个值

adjacent_find	<algorithm>	在iterator对标识元素范围内,查找一对相邻重复元素,找到则返回指向这对元素的第一个元素的ForwardIterator .否则返回last.重载版本使用输入的二元操作符代替相等的判断
	函数原形	template<class FwdIt> FwdIt adjacent_find(FwdIt first, FwdIt last);
		template<class FwdIt, class Pred> FwdIt adjacent_find(FwdIt first, FwdIt last, Pred pr);
binary_search	<algorithm>	在有序序列中查找value,找到返回true.重载的版本实用指定的比较函数对象或函数指针来判断相等
	函数原形	template<class FwdIt, class T> bool binary_search(FwdIt first, FwdIt last, const T& val);
		template<class FwdIt, class T, class Pred> bool binary_search(FwdIt first, FwdIt last, const T& val,Pred pr);
count	<algorithm>	利用等于操作符,把标志范围内的元素与输入值比较,返回相等元素个数
	函数原形	template<class InIt, class Dist> size_t count(InIt first, InIt last,const T& val, Dist& n);
count_if	<algorithm>	利用输入的操作符,对标志范围内的元素进行操作,返回结果为true的个数
	函数原形	template<class InIt, class Pred, class Dist> size_t count_if(InIt first, InIt last, Pred pr);
equal_range	<algorithm>	功能类似equal，返回一对iterator，第一个表示lower_bound，第二个表示upper_bound
	函数原形	template<class FwdIt, class T> pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last,const T& val);
		template<class FwdIt, class T, class Pred> pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last,const T& val, Pred pr);
find	<algorithm>	利用底层元素的等于操作符,对指定范围内的元素与输入值进行比较.当匹配时,结束搜索,返回该元素的一个InputIterator
	函数原形	template<class InIt, class T> InIt find(InIt first, InIt last, const T& val);
find_end	<algorithm>	在指定范围内查找"由输入的另外一对iterator标志的第二个序列"的最后一次出现.找到则返回最后一对的第一个ForwardIterator,否则返回输入的"另外一对"的第一个ForwardIterator.重载版本使用用户输入的操作符代替等于操作
	函数原形	template<class FwdIt1, class FwdIt2> FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2);
		template<class FwdIt1, class FwdIt2, class Pred>

		FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2, Pred pr);
find_first_of	<algorithm>	在指定范围内查找"由输入的另外一对iterator标志的第二个序列"中任意一个元素的第一次出现。重载版本中使用了用户自定义操作符
	函数原形	template<class FwdIt1, class FwdIt2> FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2);
		template<class FwdIt1, class FwdIt2, class Pred> FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2, Pred pr);
find_if	<algorithm>	使用输入的函数代替等于操作符执行find
		template<class InIt, class Pred> InIt find_if(InIt first, InIt last, Pred pr);
lower_bound	<algorithm>	返回一个ForwardIterator，指向在有序序列范围内的可以插入指定值而不破坏容器顺序的第一个位置.重载函数使用自定义比较操作
	函数原形	template<class FwdIt, class T> FwdIt lower_bound(FwdIt first, FwdIt last, const T& val);
		template<class FwdIt, class T, class Pred> FwdIt lower_bound(FwdIt first, FwdIt last, const T& val, Pred pr);
upper_bound	<algorithm>	返回一个ForwardIterator,指向在有序序列范围内插入value而不破坏容器顺序的最后一个位置，该位置标志一个大于value的值.重载函数使用自定义比较操作
	函数原形	template<class FwdIt, class T> FwdIt upper_bound(FwdIt first, FwdIt last, const T& val);
		template<class FwdIt, class T, class Pred> FwdIt upper_bound(FwdIt first, FwdIt last, const T& val, Pred pr);
search	<algorithm>	给出两个范围，返回一个ForwardIterator,查找成功指向第一个范围内第一次出现子序列(第二个范围)的位置，查找失败指向last1,重载版本使用自定义的比较操作
	函数原形	template<class FwdIt1, class FwdIt2> FwdIt1 search(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2);
		template<class FwdIt1, class FwdIt2, class Pred> FwdIt1 search(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, Pred pr);
search_n	<algorithm>	在指定范围内查找val出现n次的子序列。重载版本使用自定义的比较操作
	函数原形	template<class FwdIt, class Dist, class T> FwdIt search_n(FwdIt first, FwdIt last,Dist n, const T& val);
		template<class FwdIt, class Dist, class T, class Pred> FwdIt search_n(FwdIt first, FwdIt last,Dist n, const T& val, Pred pr);

make_heap	<algorithm>	把指定范围内的元素生成一个堆。重载版本使用自定义比较操作
	函数原形	template<class RanIt> void make_heap(RanIt first, RanIt last);
		template<class RanIt, class Pred> void make_heap(RanIt first, RanIt last, Pred pr);
pop_heap	<algorithm>	并不真正把最大元素从堆中弹出，而是重新排序堆。它把first和last-1交换，然后重新生成一个堆。可使用容器的back来访问被"弹出"的元素或者使用pop_back进行真正的删除。重载版本使用自定义的比较操作
	函数原形	template<class RanIt> void pop_heap(RanIt first, RanIt last);
		template<class RanIt, class Pred> void pop_heap(RanIt first, RanIt last, Pred pr);
push_heap	<algorithm>	假设first到last-1是一个有效堆，要被加入到堆的元素存放在位置last-1，重新生成堆。在指向该函数前，必须先把元素插入容器后。重载版本使用指定的比较操作
	函数原形	template<class RanIt>void push_heap(RanIt first, RanIt last);
		template<class RanIt, class Pred> void push_heap(RanIt first, RanIt last, Pred pr);
sort_heap	<algorithm>	对指定范围内的序列重新排序，它假设该序列是个有序堆。重载版本使用自定义比较操作
	函数原形	template<class RanIt> void sort_heap(RanIt first, RanIt last);
		template<class RanIt, class Pred> void sort_heap(RanIt first, RanIt last, Pred pr);

equal	<algorithm>	如果两个序列在标志范围内元素都相等，返回true。重载版本使用输入的操作符代替默认的等于操作符
	函数原形	template<class InIt1, class InIt2> bool equal(InIt1 first, InIt1 last, InIt2 x);
		template<class InIt1, class InIt2, class Pred> bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);

includes	<algorithm>	判断第一个指定范围内的所有元素是否都被第二个范围包含，使用底层元素的<操作符，成功返回true。重载版本使用用户输入的函数
	函数原形	template<class InIt1, class InIt2> bool includes(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2);
		template<class InIt1, class InIt2, class Pred> bool includes(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, Pred pr);
lexicographical_compare	<algorithm>	比较两个序列。重载版本使用用户自定义比较操作
	函数原形	template<class InIt1, class InIt2> bool lexicographical_compare(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2);
		template<class InIt1, class InIt2, class Pred> bool lexicographical_compare(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, Pred pr);
max	<algorithm>	返回两个元素中较大一个。重载版本使用自定义比较操作
	函数原形	template<class T> const T& max(const T& x, const T& y);
		template<class T, class Pred> const T& max(const T& x, const T& y, Pred pr);
max_element	<algorithm>	返回一个ForwardIterator，指出序列中最大的元素。重载版本使用自定义比较操作
	函数原形	template<class FwdIt> FwdIt max_element(FwdIt first, FwdIt last);
		template<class FwdIt, class Pred> FwdIt max_element(FwdIt first, FwdIt last, Pred pr);
min	<algorithm>	返回两个元素中较小一个。重载版本使用自定义比较操作
	函数原形	template<class T> const T& min(const T& x, const T& y);
		template<class T, class Pred> const T& min(const T& x, const T& y, Pred pr);
min_element	<algorithm>	返回一个ForwardIterator，指出序列中最小的元素。重载版本使用自定义比较操作
	函数原形	template<class FwdIt> FwdIt min_element(FwdIt first, FwdIt last);
		template<class FwdIt, class Pred> FwdIt min_element(FwdIt first, FwdIt last, Pred pr);
mismatch	<algorithm>	并行比较两个序列，指出第一个不匹配的位置，返回一对iterator，标志第一个不匹配元素位置。如果都匹配，返回每个容器的last。重载版本使用自定义的比较操作
	函数原形	template<class InIt1, class InIt2> pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last, InIt2 x);
		template<class InIt1, class InIt2, class Pred> pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last, InIt2 x, Pred pr);

---

set_union	<algorithm>	构造一个有序序列，包含两个序列中所有的不重复元素。重载版本使用自定义的比较操作
	函数原形	template<class InIt1, class InIt2, class OutIt> OutIt set_union(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
		template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_union(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2,OutIt x, Pred pr);
set_intersection	<algorithm>	构造一个有序序列，其中元素在两个序列中都存在。重载版本使用自定义的比较操作
	函数原形	template<class InIt1, class InIt2, class OutIt> OutIt set_intersection(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x);
		template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_intersection(InIt1 first1, InIt1 last1,InIt2 first2,InIt2 last2, OutIt x, Pred pr);
set_difference	<algorithm>	构造一个有序序列，该序列仅保留第一个序列中存在的而第二个中不存在的元素。重载版本使用自定义的比较操作
	函数原形	template<class InIt1, class InIt2, class OutIt> OutIt set_difference(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x);
		template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);
set_symmetric_difference	<algorithm>	构造一个有序序列，该序列取两个序列的对称差集(并集-交集)
	函数原形	template<class InIt1, class InIt2, class OutIt>

		OutIt set_symmetric_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
		template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_symmetric_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);

排列组合算法(2个)提供计算给定集合按一定顺序的所有可能排列组合

next_permutation	<algorithm>	取出当前范围内的排列，并重新排序为下一个排列。重载版本使用自定义的比较操作
	函数原形	template<class BidIt> bool next_permutation(BidIt first, BidIt last);
		template<class BidIt, class Pred> bool next_permutation(BidIt first, BidIt last, Pred pr);
prev_permutation	<algorithm>	取出指定范围内的序列并将它重新排序为上一个序列。如果不存在上一个序列则返回false。重载版本使用自定义的比较操作
	函数原形	template<class BidIt> bool prev_permutation(BidIt first, BidIt last);
		template<class BidIt, class Pred> bool prev_permutation(BidIt first, BidIt last, Pred pr);

排序和通用算法(14个)：提供元素排序策略

inplace_merge	<algorithm>	合并两个有序序列，结果序列覆盖两端范围。重载版本使用输入的操作进行排序
	函数原形	template<class BidIt> void inplace_merge(BidIt first, BidIt middle, BidIt last);
		template<class BidIt, class Pred> void inplace_merge(BidIt first, BidIt middle, BidIt last, Pred pr);
merge	<algorithm>	合并两个有序序列，存放到另一个序列。重载版本使用自定义的比较
	函数原形	template<class InIt1, class InIt2, class OutIt> OutIt merge(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x);
		template<class InIt1, class InIt2, class OutIt, class Pred> OutIt merge(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x, Pred pr);
nth_element	<algorithm>	将范围内的序列重新排序，使所有小于第n个元素的元素都出现在它前面，而大于它的都出现在后面。重载版本使用自定义的比较操作
	函数原形	template<class RanIt> void nth_element(RanIt first, RanIt nth, RanIt last);
		template<class RanIt, class Pred> void nth_element(RanIt first, RanIt nth, RanIt last, Pred pr);
partial_sort	<algorithm>	对序列做部分排序，被排序元素个数正好可以被放到范围内。重载版本使用自定义的比较操作
	函数原形	template<class RanIt> void partial_sort(RanIt first, RanIt middle, RanIt last);
		template<class RanIt, class Pred> void partial_sort(RanIt first, RanIt middle, RanIt last, Pred pr);
partial_sort_copy	<algorithm>	与partial_sort类似，不过将经过排序的序列复制到另一个容器
	函数原形	template<class InIt, class RanIt> RanIt partial_sort_copy(InIt first1, InIt last1,RanIt first2, RanIt last2);
		template<class InIt, class RanIt, class Pred> RanIt partial_sort_copy(InIt first1, InIt last1,RanIt first2, RanIt last2, Pred pr);
partition	<algorithm>	对指定范围内元素重新排序，使用输入的函数，把结果为true的元素放在结果为false的元素之前
	函数原形	template<class BidIt, class Pred> BidIt partition(BidIt first, BidIt last, Pred pr);
random_shuffle	<algorithm>	对指定范围内的元素随机调整次序。重载版本输入一个随机数产生操作
	函数原形	template<class RanIt> void random_shuffle(RanIt first, RanIt last);
		template<class RanIt, class Fun> void random_shuffle(RanIt first, RanIt last, Fun& f);
reverse	<algorithm>	将指定范围内元素重新反序排序
	函数原形	template<class BidIt> void reverse(BidIt first, BidIt last);
reverse_copy	<algorithm>	与reverse类似，不过将结果写入另一个容器
	函数原形	template<class BidIt, class OutIt> OutIt reverse_copy(BidIt first, BidIt last, OutIt x);

rotate	<algorithm>	将指定范围内元素移到容器末尾，由middle指向的元素成为容器第一个元素
	函数原形	template<class FwdIt> void rotate(FwdIt first, FwdIt middle, FwdIt last);
rotate_copy	<algorithm>	与rotate类似，不过将结果写入另一个容器
	函数原形	template<class FwdIt, class OutIt> OutIt rotate_copy(FwdIt first, FwdIt middle, FwdIt last, OutIt x);
sort	<algorithm>	以升序重新排列指定范围内的元素。重载版本使用自定义的比较操作
	函数原形	template<class RanIt> void sort(RanIt first, RanIt last);
		template<class RanIt, class Pred> void sort(RanIt first, RanIt last, Pred pr);
stable_sort	<algorithm>	与sort类似，不过保留相等元素之间的顺序关系
	函数原形	template<class BidIt> void stable_sort(BidIt first, BidIt last);
		template<class BidIt, class Pred> void stable_sort(BidIt first, BidIt last, Pred pr);
stable_partition	<algorithm>	与partition类似，不过不保证保留容器中的相对顺序
	函数原形	template<class FwdIt, class Pred> FwdIt stable_partition(FwdIt first, FwdIt last, Pred pr);

删除和替换算法(15个)

copy	<algorithm>	复制序列
	函数原形	template<class InIt, class OutIt> OutIt copy(InIt first, InIt last, OutIt x);
copy_backward	<algorithm>	与copy相同，不过元素是以相反顺序被拷贝
	函数原形	template<class BidIt1, class BidIt2> BidIt2 copy_backward(BidIt1 first, BidIt1 last, BidIt2 x);
iter_swap	<algorithm>	交换两个ForwardIterator的值
	函数原形	template<class FwdIt1, class FwdIt2> void iter_swap(FwdIt1 x, FwdIt2 y);
remove	<algorithm>	删除指定范围内所有等于指定元素的元素。注意，该函数不是真正删除函数。内置函数不适合使用remove和remove_if函数
	函数原形	template<class FwdIt, class T> FwdIt remove(FwdIt first, FwdIt last, const T& val);
remove_copy	<algorithm>	将所有不匹配元素复制到一个制定容器，返回OutputIterator指向被拷贝的末元素的下一个位置
	函数原形	template<class InIt, class OutIt, class T> OutIt remove_copy(InIt first, InIt last, OutIt x, const T& val);
remove_if	<algorithm>	删除指定范围内输入操作结果为true的所有元素
	函数原形	template<class FwdIt, class Pred> FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
remove_copy_if	<algorithm>	将所有不匹配元素拷贝到一个指定容器
	函数原形	template<class InIt, class OutIt, class Pred> OutIt remove_copy_if(InIt first, InIt last, OutIt x, Pred pr);
replace	<algorithm>	将指定范围内所有等于vold的元素都用vnew代替
	函数原形	template<class FwdIt, class T> void replace(FwdIt first, FwdIt last,const T& vold, const T& vnew);
replace_copy	<algorithm>	与replace类似，不过将结果写入另一个容器
	函数原形	template<class InIt, class OutIt, class T> OutIt replace_copy(InIt first, InIt last, OutIt x,const T& vold, const T& vnew);
replace_if	<algorithm>	将指定范围内所有操作结果为true的元素用新值代替
	函数原形	template<class FwdIt, class Pred, class T> void replace_if(FwdIt first, FwdIt last,Pred pr, const T& val);
replace_copy_if	<algorithm>	与replace_if，不过将结果写入另一个容器
	函数原形	template<class InIt, class OutIt, class Pred, class T> OutIt replace_copy_if(InIt first, InIt last, OutIt x, Pred pr, const T& val);
swap	<algorithm>	交换存储在两个对象中的值
	函数原形	template<class T> void swap(T& x, T& y);
swap_range	<algorithm>	将指定范围内的元素与另一个序列元素值进行交换
	函数原形	template<class FwdIt1, class FwdIt2> FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last, FwdIt2 x);

unique	<algorithm>	清除序列中重复元素，和remove类似，它也不能真正删除元素。重载版本使用自定义比较操作
	函数原形	template<class FwdIt> FwdIt unique(FwdIt first, FwdIt last);
		template<class FwdIt, class Pred> FwdIt unique(FwdIt first, FwdIt last, Pred pr);
unique_copy	<algorithm>	与unique类似，不过把结果输出到另一个容器
	函数原形	template<class InIt, class OutIt> OutIt unique_copy(InIt first, InIt last, OutIt x);
		template<class InIt, class OutIt, class Pred> OutIt unique_copy(InIt first, InIt last, OutIt x, Pred pr);

生成和变异算法

fill	<algorithm>	将输入值赋给标志范围内的所有元素
	函数原形	template<class FwdIt, class T> void fill(FwdIt first, FwdIt last, const T& x);
fill_n	<algorithm>	将输入值赋给first到first+n范围内的所有元素
	函数原形	template<class OutIt, class Size, class T> void fill_n(OutIt first, Size n, const T& x);
for_each	<algorithm>	用指定函数依次对指定范围内所有元素进行迭代访问，返回所指定的函数类型。该函数不得修改序列中的元素
	函数原形	template<class InIt, class Fun> Fun for_each(InIt first, InIt last, Fun f);
generate	<algorithm>	连续调用输入的函数来填充指定的范围
	函数原形	template<class FwdIt, class Gen> void generate(FwdIt first, FwdIt last, Gen g);
generate_n	<algorithm>	与generate函数类似，填充从指定iterator开始的n个元素
	函数原形	template<class OutIt, class Pred, class Gen> void generate_n(OutIt first, Dist n, Gen g);
transform	<algorithm>	将输入的操作作用与指定范围内的每个元素，并产生一个新的序列。重载版本将操作作用在一对元素上，另外一个元素来自输入的另外一个序列。结果输出到指定容器
	函数原形	template<class InIt, class OutIt, class Unop> OutIt transform(InIt first, InIt last, OutIt x, Unop uop);
		template<class InIt1, class InIt2, class OutIt, class Binop> OutIt transform(InIt1 first1, InIt1 last1, InIt2 first2,OutIt x, Binop bop);

accumulate	<numeric>	iterator对标识的序列段元素之和，加到一个由val指定的初始值上。重载版本不再做加法，而是传进来的二元操作符被应用到元素上
	函数原形	template<class InIt, class T> T accumulate(InIt first, InIt last, T val);
		template<class InIt, class T, class Pred> T accumulate(InIt first, InIt last, T val, Pred pr);
partial_sum	<numeric>	创建一个新序列，其中每个元素值代表指定范围内该位置前所有元素之和。重载版本使用自定义操作代替加法
	函数原形	template<class InIt, class OutIt> OutIt partial_sum(InIt first, InIt last,OutIt result);
		template<class InIt, class OutIt, class Pred> OutIt partial_sum(InIt first, InIt last,OutIt result, Pred pr);
product	<numeric>	对两个序列做内积(对应元素相乘，再求和)并将内积加到一个输入的初始值上。重载版本使用用户定义的操作
	函数原形	template<class InIt1, class InIt2, class T> T product(InIt1 first1, InIt1 last1,Init2 first2, T val);
		template<class InIt1, class InIt2, class T,class Pred1, class Pred2> T product(InIt1 first1, InIt1 last1,Init2 first2, T val, Pred1 pr1, Pred2 pr2);
adjacent_difference	<numeric>	创建一个新序列，新序列中每个新值代表当前元素与上一个元素的差。重载版本用指定二元操作计算相邻元素的差
	函数原形	template<class InIt, class OutIt> OutIt adjacent_difference(InIt first, InIt last,OutIt result);
		template<class InIt, class OutIt, class Pred> OutIt adjacent_difference(InIt first, InIt last,OutIt result, Pred pr);