map与set复习

1. 序列式容器和关联式容器

`string、vector、list、deque、array、forward_list` 等，这些容器统称为序列式容器，因为逻辑结构为线性序列的数据结构，两个位置存储的值之间一般没有紧密的关联关系，比如交换一下，他依旧是序列式容器。顺序容器中的元素是按他们在容器中的存储位置来顺序保存和访问的。

2. Set

> set文档

2.1 set类的介绍

- `T` 就是 `set` 底层关键字的类型，set默认要求T支持小于比较，如果不支持或者想按照自己的需求可以实现仿函数传给第二个模版参数。
- set底层存储数据的内存是从空间配置器申请的，如果需要可以自己实现内存池，传给第三个参数。
- set底层使用红黑树实现，增删查效率是***O(logN)***，迭代器遍历是走的搜索树的中序遍历，所以是有序的。

代码块

```
template < class T,                        // set::key_type/value_type
           class Compare = less<T>,        // set::key_compare/value_compare
           class Alloc = allocator<T>      // set::allocator_type
           > class set;
```

2.2 set的构造和迭代器

`set` 的支持正向和反向迭代遍历，遍历默认按升序顺序，因为底层是二叉搜索树，迭代器遍历走的中序；支持迭代器就意味着支持范围 `for` ， `set` 的 `iterator` 和 `const_iterator` 都不支持迭代器修改数据，修改关键字数据，破坏了底层搜索树的结构。

代码块

```
// empty (1) 无参默认构造
explicit set (const key_compare& comp = key_compare(),
```

```
                const allocator_type& alloc = allocator_type());

// range (2) 迭代器区间构造
template <class InputIterator>
    set (InputIterator first, InputIterator last,
        const key_compare& comp = key_compare(),
        const allocator_type& = allocator_type());

// copy (3) 拷贝构造
set (const set& x);

// initializer list (5) initializer 列表构造
set (initializer_list<value_type> il,
    const key_compare& comp = key_compare(),
    const allocator_type& alloc = allocator_type());

// 迭代器是一个双向迭代器
iterator -> a bidirectional iterator to const value_type

// 正向迭代器
iterator begin();
iterator end();

// 反向迭代器
reverse_iterator rbegin();
reverse_iterator rend();
```

2.3 set的增删查

代码块

```
Member types
key_type -> The first template parameter (T)
value_type -> The first template parameter (T)

// 单个数据插入，如果已经存在则插入失败
pair<iterator,bool> insert (const value_type& val);

// 列表插入，已经在容器中存在的值不会插入
void insert (initializer_list<value_type> il);

// 迭代器区间插入，已经在容器中存在的值不会插入
template <class InputIterator>
void insert (InputIterator first, InputIterator last);
```

```
14
15  // 查找val，返回val所在的迭代器，没有找到返回end()
16  iterator find (const value_type& val);
17
18  // 查找val，返回Val的个数
19  size_type count (const value_type& val) const;
20
21  // 删除一个迭代器位置的值
22  iterator erase (const_iterator position);
23
24  // 删除val，val不存在返回0，存在返回1
25  size_type erase (const value_type& val);
26
27  // 删除一段迭代器区间的值
28  iterator erase (const_iterator first, const_iterator last);
29
30  // 返回大于等val位置的迭代器
31  iterator lower_bound (const value_type& val) const;
32
33  // 返回大于val位置的迭代器
34  iterator upper_bound (const value_type& val) const;
```

2.4 multiset和set的差异

`multiset` 和 `set` 的使用基本完全类似，主要区别点在于 `multiset` 支持值冗余，那么 `insert/find/count/erase` 都围绕着支持值冗余有所差异。

3. Map

> map文档

3.1 map类的介绍

`map` 的声明如下， `Key` 就是 `map` 底层关键字的类型， `T` 是 `map` 底层 `value` 的类型， `set` 默认要求 `Key` 支持小于比较，如果不支持或者需要的话可以自行实现仿函数传给第二个模版参数， `map` 底层存储数据的内存是从空间配置器申请的。一般情况下，我们都不需要传后两个模版参数。 `map` 底层是用红黑树实现，增删查改效率是 ***O(logN)*** ，迭代器遍历是走的中序，所以是按key有序顺序遍历的。

代码块

```
1  template < class Key,                                    // map::key_type
```

```
2             class T,                                        // map::mapped_type
3             class Compare = less<Key>,                      // map::key_compare
4             class Alloc = allocator<pair<const Key,T> >     //
    map::allocator_type
5             > class map;
```

3.2 pair类型介绍

map底层的红黑树节点中的数据，使用pair<Key, T>存储键值对数据。

`std::pair` 是C++标准库中的一个实用模板类，定义在 `<utility>` 头文件中，用于将两个值组合成一个单一对象。它提供了一种简单的方式来存储和操作两个相关联的值，而不需要专门定义一个结构体或类。

基本特性

- `pair` 是一个模板类，可以存储任意两种类型的值
- 第一个元素通过 `first` 成员访问
- 第二个元素通过 `second` 成员访问
- 元素类型可以相同也可以不同

3.3 map的构造

`map` 的支持正向和反向迭代遍历，遍历默认按 `key` 的升序顺序，因为底层是二叉搜索树，迭代器遍历走

的中序；支持迭代器就意味着支持范围 `for` ， `map` 支持修改 `value` 数据，不支持修改 `key` 数据，修改关键字数据，破坏了底层搜索树的结构。

代码块

```
1   // empty (1) 无参默认构造
2   explicit map (const key_compare& comp = key_compare(),
3                 const allocator_type& alloc = allocator_type());
4
5   // range (2) 迭代器区间构造
6   template <class InputIterator>
7   map (InputIterator first, InputIterator last,
8               const key_compare& comp = key_compare(),
9               const allocator_type& = allocator_type());
10
11  // copy (3) 拷贝构造
```

```
map (const map& x);

// initializer list (5) initializer 列表构造
map (initializer_list<value_type> il,
        const key_compare& comp = key_compare(),
        const allocator_type& alloc = allocator_type());

// 迭代器是一个双向迭代器
iterator -> a bidirectional iterator to const value_type

// 正向迭代器
iterator begin();
iterator end();

// 反向迭代器
reverse_iterator rbegin();
reverse_iterator rend();
```

3.4 map的增删查

`map` 增接口，插入的pair键值对数据，跟set所有不同，但是查和删的接口只用关键字 `key` 跟 `set` 是完全类似的，不过 `find` 返回 `iterator` ，不仅仅可以确认 `key` 在不在，还找到 `key` 映射的 `value` ，同时通过迭代还可以修改 `value` 。

代码块

```
Member types
key_type -> The first template parameter (Key)
mapped_type -> The second template parameter (T)
value_type -> pair<const key_type,mapped_type>

// 单个数据插入，如果已经key存在则插入失败,key存在相等value不相等也会插入失败
pair<iterator,bool> insert (const value_type& val);

// 列表插入，已经在容器中存在的值不会插入
void insert (initializer_list<value_type> il);

// 迭代器区间插入，已经在容器中存在的值不会插入
template <class InputIterator>
void insert (InputIterator first, InputIterator last);

// 查找k，返回k所在的迭代器，没有找到返回end()
iterator find (const key_type& k);

```

```
// 查找k，返回k的个数
size_type count (const key_type& k) const;

// 删除一个迭代器位置的值
iterator erase (const_iterator position);

// 删除k，k存在返回0，存在返回1
size_type erase (const key_type& k);

// 删除一段迭代器区间的值
iterator erase (const_iterator first, const_iterator last);

// 返回大于等k位置的迭代器
iterator lower_bound (const key_type& k);

// 返回大于k位置的迭代器
const_iterator lower_bound (const key_type& k) const;
```

3.5 map的数据修改

`map` 第一个支持修改的方式时通过迭代器，迭代器遍历时或者 `find` 返回 `key` 所在的 `iterator` 修改， `map` 还有一个非常重要的修改接口 `operator[]` ，但是 `operator[]` 不仅仅支持修改，还支持插入数据和查找数据，所以他是一个多功能复合接口。

需要注意从内部实现角角，map这里把我们传统说的value值，给的是 `T` 类型， `typedef` 为 `mapped_type` 。而 `value_type` 是红黑树结点中存储的 `pair` 键值对值。日常使用我们还是习惯将这里的 `T` 映射值叫做value。

代码块

```
Member types
key_type -> The first template parameter (Key)
mapped_type -> The second template parameter (T)
value_type -> pair<const key_type,mapped_type>

// 查找k，返回k所在的迭代器，没有找到返回end()，如果找到了通过iterator可以修改key对应的
// mapped_type值
iterator find (const key_type& k);

// 文档中对insert返回值的说明
/*The single element versions (1) return a pair, with its member pair::first
set to an iterator pointing to either the newly inserted element or to the
```

```
element with an equivalent key in the map. The pair::second element in the pair
is set to true if a new element was inserted or false if an equivalent key already existed. */
// insert插入一个pair<key, T>对象
/* 1、如果key已经在map中，插入失败，则返回一个pair<iterator,bool>对象，返回pair对象first是key所在结点的迭代器，second是false */
/* 2、如果key不在在map中，插入成功，则返回一个pair<iterator,bool>对象，返回pair对象first是新插入key所在结点的迭代器，second是true */
/* 也就是说无论插入成功还是失败，返回pair<iterator,bool>对象的first都会指向key所在的迭
代器*/
/* 那么也就意味着insert插入失败时充当了查找的功能，正是因为这一点，insert可以用来实现operator[] */
/* 需要注意的是这里有两个pair，不要混淆了，一个是map底层红黑树节点中存的pair<key, T>，另
 一个是insert返回值pair<iterator,bool> */

pair<iterator,bool> insert (const value_type& val);

mapped_type& operator[] (const key_type& k);

// operator的内部实现
mapped_type& operator[] (const key_type& k)
{
    // 1、如果k不在map中，insert会插入k和mapped_type默认值，同时[]返回结点中存储
    // mapped_type值的引用，那么我们可以通过引用修改返映射值。所以[]具备了插入+修改功能
    // 2、如果k在map中，insert会插入失败，但是insert返回pair对象的first是指向key结点的
    // 迭代器，返回值同时[]返回结点中存储mapped_type值的引用，所以[]具备了查找+修改的功能
    pair<iterator, bool> ret = insert({ k, mapped_type() });
    iterator it = ret.first;
    return it->second;
}
```

3.6 multimap和map的差异

`multimap` 和 `map` 的使用基本完全类似，主要区别点在于 `multimap` 支持关键值 `key` 冗余，那么 `insert/find/count/erase` 都围绕着支持关键值 `key` 冗余有所差异，这里跟 `set` 和 `multiset` 完全一样，比如 `find` 时，有多个 `key` ，返回中序第一个。其次就是 `multimap` 不支持 `[]` ，因为支持 `key` 冗余， `[]` 就只能支持插入了，不能支持修改。