**带你深入理解STL之Set和Map**

发表于 2016-09-09 | 分类于 [STL](http://zcheng.ren/categories/STL/) | [2](http://zcheng.ren/2016/09/09/STLSetAndMap/#comments) | 921

在上一篇博客中，讲到了STL中关于红黑树的实现，理解起来比较复杂，正所谓前人种树，后人乘凉，RBTree把树都种好了，接下来就该set和map这类关联式容器来“乘凉”了。

STL的set和map都是基于红黑树实现的，和stack和queue都是基于deque一样，它们仅仅是调用了RBTree提供的接口函数，然后进行外层封装即可。本篇博客理解起来比较轻松，set和map的源代码也不多，大家可以慢慢“品味”。另外，还会介绍multiset和multimap这两个容器，并给出它们的区别和应用等。还等什么呢？走吧，带你理解理解set和map吧！

**set**

set是一种关联式容器，其特性如下：

* set以RBTree作为底层容器
* 所得元素的只有key没有value，value就是key
* 不允许出现键值重复
* 所有的元素都会被自动排序
* 不能通过迭代器来改变set的值，因为set的值就是键

针对这五点来说，前四点都不用再多作说明，第五点需要做一下说明。如果set中允许修改键值的话，那么首先需要删除该键，然后调节平衡，在插入修改后的键值，再调节平衡，如此一来，严重破坏了set的结构，导致iterator失效，不知道应该指向之前的位置，还是指向改变后的位置。所以STL中将set的迭代器设置成const，不允许修改迭代器的值。

**set的数据结构**

|  |
| --- |
|  |
| // 比较器默认采用less，内部按照升序排列，配置器默认采用alloc  template <class Key, class Compare = less<Key>, class Alloc = alloc>  class set  {  public:  // 在set中key就是value, value同时也是key  typedef Key key\_type;  typedef Key value\_type;  // 用于比较的函数  typedef Compare key\_compare;  typedef Compare value\_compare;  private:  // 内部采用RBTree作为底层容器  typedef rb\_tree<key\_type, value\_type,  identity<value\_type>, key\_compare, Alloc> rep\_type;  rep\_type t; // t为内部RBTree容器  public:  // 用于提供iterator\_traits<I>支持  typedef typename rep\_type::const\_pointer pointer;  typedef typename rep\_type::const\_pointer const\_pointer;  typedef typename rep\_type::const\_reference reference;  typedef typename rep\_type::const\_reference const\_reference;  typedef typename rep\_type::difference\_type difference\_type;  // 设置成const迭代器，set的键值不允许修改  typedef typename rep\_type::const\_iterator iterator;  typedef typename rep\_type::const\_iterator const\_iterator;  // 反向迭代器  typedef typename rep\_type::const\_reverse\_iterator reverse\_iterator;  typedef typename rep\_type::const\_reverse\_iterator const\_reverse\_iterator;  typedef typename rep\_type::size\_type size\_type;  iterator begin() const { return t.begin(); }  iterator end() const { return t.end(); }  reverse\_iterator rbegin() const { return t.rbegin(); }  reverse\_iterator rend() const { return t.rend(); }  bool empty() const { return t.empty(); }  size\_type size() const { return t.size(); }  size\_type max\_size() const { return t.max\_size(); }  // 返回用于key比较的函数  key\_compare key\_comp() const { return t.key\_comp(); }  // 由于set的性质, value比较和key使用同一个比较函数  value\_compare value\_comp() const { return t.key\_comp(); }  // 声明了两个友元函数，重载了==和<操作符  friend bool operator== \_\_STL\_NULL\_TMPL\_ARGS (const set&, const set&);  friend bool operator< \_\_STL\_NULL\_TMPL\_ARGS (const set&, const set&);  // ...  } | |

**set的构造函数**

set提供了如下几个构造函数用于初始化一个set

|  |
| --- |
|  |

|  |
| --- |
| // 注：下面相关函数都在set类中定义，为了介绍方便才抽出来单独讲解  // 空构造函数，初始化一个空的set  set() : t(Compare()) {}  // 支持自定义比较器，如set<int,greater<int> > myset的初始化  explicit set(const Compare& comp) : t(comp) {}  // 实现诸如set<int> myset(anotherset.begin(),anotherset.end())这样的初始化  template <class InputIterator>  set(InputIterator first, InputIterator last)  : t(Compare()) { t.insert\_unique(first, last); }  // 支持自定义比较器的初始化操作  template <class InputIterator>  set(InputIterator first, InputIterator last, const Compare& comp)  : t(comp) { t.insert\_unique(first, last); }  // 以另一个set来初始化  set(const set<Key, Compare, Alloc>& x) : t(x.t) {}  // 赋值运算符函数  set<Key, Compare, Alloc>& operator=(const set<Key, Compare, Alloc>& x)  {  t = x.t;  return \*this;  } |

**set的操作函数**

**insert**

插入函数，调用RBTree的插入函数即可

|  |
| --- |
|  |

|  |
| --- |
| typedef pair<iterator, bool> pair\_iterator\_bool;  // 由于set不允许键值重复，所以必须调用RBTree的insert\_unique函数  // second表示插入操作是否成功  pair<iterator,bool> insert(const value\_type& x)  {  pair<typename rep\_type::iterator, bool> p = t.insert\_unique(x);  return pair<iterator, bool>(p.first, p.second);  }  // 在position处插入元素, 但是position仅仅是个提示, 如果给出的位置不能进行插入,  // STL会进行查找, 这会导致很差的效率  iterator insert(iterator position, const value\_type& x)  {  typedef typename rep\_type::iterator rep\_iterator;  return t.insert\_unique((rep\_iterator&)position, x);  }  // 将[first，last)区间内的元素插入到set中  template <class InputIterator>  void insert(InputIterator first, InputIterator last)  {  t.insert\_unique(first, last);  } |

**erase**

擦除函数，用于擦除单个元素或者区间内的元素，直接调用RBTree的函数即可

|  |
| --- |
|  |

|  |
| --- |
| // 擦除指定位置的元素, 会导致内部的红黑树重新排列  void erase(iterator position)  {  typedef typename rep\_type::iterator rep\_iterator;  t.erase((rep\_iterator&)position);  }  // 会返回擦除元素的个数, 其实就是标识set内原来是否有指定的元素  size\_type erase(const key\_type& x)  {  return t.erase(x);  }  // 擦除指定区间的元素, 会导致红黑树有较大变化  void erase(iterator first, iterator last)  {  typedef typename rep\_type::iterator rep\_iterator;  t.erase((rep\_iterator&)first, (rep\_iterator&)last);  } |

**clean**

清除整个set容器，直接调用RBTree的clean函数即可

|  |
| --- |
|  |

|  |
| --- |
| void clear() { t.clear(); } |

**find**

查找函数，RBTree也提供了，直接调用即可

|  |
| --- |
|  |

|  |
| --- |
| // 查找指定的元素  iterator find(const key\_type& x) const { return t.find(x); } |

**count**

查找制定元素的个数

|  |
| --- |
|  |

|  |
| --- |
| // 返回指定元素的个数, set不允许键值重复，其实就是测试元素是否在set中  size\_type count(const key\_type& x) const { return t.count(x); } |

**重载操作符**

set重载了==和<操作符，基本上都是调用RBTree的接口函数即可，如下所示：

|  |
| --- |
|  |

|  |
| --- |
| template <class Key, class Compare, class Alloc>  inline bool operator==(const set<Key, Compare, Alloc>& x,  const set<Key, Compare, Alloc>& y) {  return x.t == y.t;  }  template <class Key, class Compare, class Alloc>  inline bool operator<(const set<Key, Compare, Alloc>& x,  const set<Key, Compare, Alloc>& y) {  return x.t < y.t;  } |

**其他操作函数**

|  |
| --- |
|  |

|  |
| --- |
| // 返回小于当前元素的第一个可插入的位置  iterator lower\_bound(const key\_type& x) const  {  return t.lower\_bound(x);  }  // 返回大于当前元素的第一个可插入的位置  iterator upper\_bound(const key\_type& x) const  {  return t.upper\_bound(x);  }  // 返回与指定键值相等的元素区间  pair<iterator,iterator> equal\_range(const key\_type& x) const  {  return t.equal\_range(x);  } |

**multiset**

multiset相对于set来说，区别就是multiset允许键值重复，在multiset中调用的是RBTree的insert\_equal函数，其他的基本与set相同。

其他的就不赘述了，下面列举一下跟set不同的地方：

|  |
| --- |
|  |

|  |
| --- |
| // 初始化函数，  // 注意！！！！插入操作采用的是RBTree的insert\_equal，而不是insert\_unique  template <class InputIterator>  multiset(InputIterator first, InputIterator last)  : t(Compare()) { t.insert\_equal(first, last); }  template <class InputIterator>  multiset(InputIterator first, InputIterator last, const Compare& comp)  : t(comp) { t.insert\_equal(first, last); }  // 插入元素, 注意, 插入的元素key允许重复  iterator insert(const value\_type& x)  {  return t.insert\_equal(x);  }  // 在position处插入元素, 但是position仅仅是个提示, 如果给出的位置不能进行插入,  // STL会进行查找, 这会导致很差的效率  iterator insert(iterator position, const value\_type& x)  {  typedef typename rep\_type::iterator rep\_iterator;  return t.insert\_equal((rep\_iterator&)position, x);  } |

**map**

map和set一样是关联式容器，它们的底层容器都是红黑树，区别就在于map的值不作为键，键和值是分开的。它的特性如下：

* map以RBTree作为底层容器
* 所有元素都是键+值存在
* 不允许键重复
* 所有元素是通过键进行自动排序的
* map的键是不能修改的，但是其键对应的值是可以修改的

在map中，一个键对应一个值，其中键不允许重复，不允许修改，但是键对应的值是可以修改的，原因可以看上面set中的解释。下面就一起来看看STL中的map的源代码。

**map的数据结构**

|  |
| --- |
|  |

|  |
| --- |
| // 默认比较器为less<key>,元素按照键的大小升序排列  template <class Key, class T, class Compare = less<Key>, class Alloc = alloc>  class map {  public:  typedef Key key\_type; // key类型  typedef T data\_type; // value类型  typedef T mapped\_type;  typedef pair<const Key, T> value\_type; // 元素类型, 要保证key不被修改  typedef Compare key\_compare; // 用于key比较的函数  private:  // 内部采用RBTree作为底层容器  typedef rb\_tree<key\_type, value\_type,  identity<value\_type>, key\_compare, Alloc> rep\_type;  rep\_type t; // t为内部RBTree容器  public:  // 用于提供iterator\_traits<I>支持  typedef typename rep\_type::const\_pointer pointer;  typedef typename rep\_type::const\_pointer const\_pointer;  typedef typename rep\_type::const\_reference reference;  typedef typename rep\_type::const\_reference const\_reference;  typedef typename rep\_type::difference\_type difference\_type;  // 注意：这里与set不一样，map的迭代器是可以修改的  typedef typename rep\_type::iterator iterator;  typedef typename rep\_type::const\_iterator const\_iterator;  // 反向迭代器  typedef typename rep\_type::const\_reverse\_iterator reverse\_iterator;  typedef typename rep\_type::const\_reverse\_iterator const\_reverse\_iterator;  typedef typename rep\_type::size\_type size\_type;  // 常规的返回迭代器函数  iterator begin() { return t.begin(); }  const\_iterator begin() const { return t.begin(); }  iterator end() { return t.end(); }  const\_iterator end() const { return t.end(); }  reverse\_iterator rbegin() { return t.rbegin(); }  const\_reverse\_iterator rbegin() const { return t.rbegin(); }  reverse\_iterator rend() { return t.rend(); }  const\_reverse\_iterator rend() const { return t.rend(); }  bool empty() const { return t.empty(); }  size\_type size() const { return t.size(); }  size\_type max\_size() const { return t.max\_size(); }  // 返回用于key比较的函数  key\_compare key\_comp() const { return t.key\_comp(); }  // 由于map的性质, value和key使用同一个比较函数, 实际上我们并不使用value比较函数  value\_compare value\_comp() const { return value\_compare(t.key\_comp()); }  // 注意: 这里有一个常见的陷阱, 如果访问的key不存在, 会新建立一个  T& operator[](const key\_type& k)  {  return (\*((insert(value\_type(k, T()))).first)).second;  }  // 重载了==和<操作符，后面会有实现  friend bool operator== \_\_STL\_NULL\_TMPL\_ARGS (const map&, const map&);  friend bool operator< \_\_STL\_NULL\_TMPL\_ARGS (const map&, const map&);  } |

**map的构造函数**

map提供了一下的构造函数来初始化一个map

|  |
| --- |
|  |

|  |
| --- |
| // 空构造函数，直接调用RBTree的空构造函数  map() : t(Compare()) {}  explicit map(const Compare& comp) : t(comp) {}  // 提供类似map<int,int> myMap(anotherMap.begin(),anotherMap.end())的初始化  template <class InputIterator>  map(InputIterator first, InputIterator last)  : t(Compare()) { t.insert\_unique(first, last); }  // 提供类似map<int,int> myMap(anotherMap.begin(),anotherMap.end(),less<int>)初始化  template <class InputIterator>  map(InputIterator first, InputIterator last, const Compare& comp)  : t(comp) { t.insert\_unique(first, last); }  // 提供类似map<int> maMap(anotherMap)的初始化  map(const map<Key, T, Compare, Alloc>& x) : t(x.t) {}  // 重载=操作符，赋值运算符  map<Key, T, Compare, Alloc>& operator=(const map<Key, T, Compare, Alloc>& x)  {  t = x.t;  return \*this;  } |

**map的操作函数**

**insert**

同set一样，直接调用RBTree的插入函数即可，注意map不允许键值重复，所以调用的是insert\_unique

|  |
| --- |
|  |

|  |
| --- |
| // 对于相同的key, 只允许出现一次, bool标识  pair<iterator,bool> insert(const value\_type& x) { return t.insert\_unique(x); }  // 在position处  插入元素, 但是position仅仅是个提示, 如果给出的位置不能进行插入,  // STL会进行查找, 这会导致很差的效率  iterator insert(iterator position, const value\_type& x)  {  return t.insert\_unique(position, x);  }  // 将[first，last)区间内的元素插入到map中  template <class InputIterator>  void insert(InputIterator first, InputIterator last) {  t.insert\_unique(first, last);  } |

**erase**

同set，直接调用即可

|  |
| --- |
|  |

|  |
| --- |
| // 擦除指定位置的元素, 会导致内部的红黑树重新排列  void erase(iterator position) { t.erase(position); }  // 会返回擦除元素的个数, 其实就是标识map内原来是否有指定的元素  size\_type erase(const key\_type& x) { return t.erase(x); }  void erase(iterator first, iterator last) { t.erase(first, last); } |

**clean**

同set，直接调用即可

|  |
| --- |
|  |

|  |
| --- |
| void clear() { t.clear(); } |

**find**

|  |
| --- |
|  |

|  |
| --- |
| // 查找指定key的元素  iterator find(const key\_type& x) { return t.find(x); }  const\_iterator find(const key\_type& x) const { return t.find(x); }  ` |

**重载运算符**

上面介绍到map重载了[],==和<运算符，[]的实现已经介绍过，下面是==和<的实现

|  |
| --- |
|  |

|  |
| --- |
| // 比较map直接是对其底层容器t的比较，直接调用RBTree的比较函数即可  template <class Key, class T, class Compare, class Alloc>  inline bool operator==(const map<Key, T, Compare, Alloc>& x,  const map<Key, T, Compare, Alloc>& y)  {  return x.t == y.t;  }  template <class Key, class T, class Compare, class Alloc>  inline bool operator<(const map<Key, T, Compare, Alloc>& x,  const map<Key, T, Compare, Alloc>& y)  {  return x.t < y.t;  } |

**其他操作函数**

|  |
| --- |
|  |

|  |
| --- |
| // 返回小于当前元素的第一个可插入的位置  iterator lower\_bound(const key\_type& x) {return t.lower\_bound(x); }  const\_iterator lower\_bound(const key\_type& x) const  {  return t.lower\_bound(x);  }  // 返回大于当前元素的第一个可插入的位置  iterator upper\_bound(const key\_type& x) {return t.upper\_bound(x); }  const\_iterator upper\_bound(const key\_type& x) const  {  return t.upper\_bound(x);  }  // 返回与指定键值相等的元素区间  pair<iterator,iterator> equal\_range(const key\_type& x)  {  return t.equal\_range(x);  } |

**multimap**

multimap和map的关系就跟multiset和set的关系一样，multimap允许键的值相同，因此在插入操作的时候用到insert\_equal()，除此之外，基本上与map相同。

下面就仅仅列出不同的地方

|  |
| --- |
|  |

|  |
| --- |
| template <class Key, class T, class Compare = less<Key>, class Alloc = alloc>  class multimap  {  // ... 其他地方与map相同  // 注意下面这些函数都调用的是insert\_equal，而不是insert\_unique  template <class InputIterator>  multimap(InputIterator first, InputIterator last)  : t(Compare()) { t.insert\_equal(first, last); }  template <class InputIterator>  multimap(InputIterator first, InputIterator last, const Compare& comp)  : t(comp) { t.insert\_equal(first, last); }  // 插入元素, 注意, 插入的元素key允许重复  iterator insert(const value\_type& x) { return t.insert\_equal(x); }  // 在position处插入元素, 但是position仅仅是个提示, 如果给出的位置不能进行插入,  // STL会进行查找, 这会导致很差的效率  iterator insert(iterator position, const value\_type& x)  {  return t.insert\_equal(position, x);  }  // 插入一个区间内的元素  template <class InputIterator>  void insert(InputIterator first, InputIterator last)  {  t.insert\_equal(first, last);  }  // ...其余地方和map相同  } |

**总结**

总的来说，这四类容器仅仅只是在RBTree上进行了一层封装，首先，set和map的区别就在于键和值是否相同，set中将值作为键，支持STL的提供的一些交集、并集和差集等运算；map的键和值不同，每个键都有自己的值，键不能重复，但是值可以重复。

multimap和multiset就在map和set的基础上，使他们的键可以重复，除此之外基本等同。

关于map和set的用法，大家可以在其源代码中找到，这里就不在赘述，本系列博客也是基于源码的角度来分析这些容器。到此，end！