

05 | 容器汇编 II：需要函数对象的容器

2019-12-06 吴咏炜

现代C++实战30讲

[进入课程 >](#)



讲述：吴咏炜

时长 11:38 大小 8.00M



你好，我是吴咏炜。

上一讲我们学习了 C++ 的序列容器和两个容器适配器，今天我们继续讲完剩下的标准容器 ([1])。

函数对象及其特化

在讲容器之前，我们需要首先来讨论一下两个重要的函数对象，`less` 和 `hash`。

我们先看一下 `less`，小于关系。在标准库里，通用的 `less` 大致是这样定义的：

```
1  template <class T>
2  struct less
3      : binary_function<T, T, bool> {
4      bool operator()(const T& x,
5                      const T& y) const
6      {
7          return x < y;
8      }
9  };
```

也就是说，`less` 是一个函数对象，并且是个二元函数，执行对任意类型的值的比较，返回布尔类型。作为函数对象，它定义了函数调用运算符（`operator()`），并且缺省行为是对指定类型的对象进行 `<` 的比较操作。

有点平淡无奇，是吧？原因是因为这个缺省实现在大部分情况下已经够用，我们不太需要去碰它。在需要大小比较的场合，C++ 通常默认会使用 `less`，包括我们今天会讲到的若干容器和排序算法 `sort`。如果我们需要产生相反的顺序的话，则可以使用 `greater`，大于关系。


计算哈希值的函数对象 `hash` 就不一样了。它的目的是把一个某种类型的值转换成一个无符号整数哈希值，类型为 `size_t`。它没有一个可用的默认实现。对于常用的类型，系统提供了需要的特化 [2]，类似于：

```
1  template <class T> struct hash;
2
3  template <>
4  struct hash<int>
5      : public unary_function<int, size_t> {
6      size_t operator()(int v) const
7          noexcept
8      {
9          return static_cast<size_t>(v);
10     }
11 };
```

这当然是一个极其简单的例子。更复杂的类型，如指针或者 `string` 的特化，都会更复杂。要点是，对于每个类，类的作者都可以提供 `hash` 的特化，使得对于不同的对象值，函

数调用运算符都能得到尽可能均匀分布的不同数值。

我们用下面这个例子来加深一下理解：

 复制代码

```
1 #include <algorithm>    // std::sort
2 #include <functional>    // std::less/greater/hash
3 #include <iostream>      // std::cout/endl
4 #include <string>         // std::string
5 #include <vector>         // std::vector
6 #include "output_container.h"
7
8 using namespace std;
9
10 int main()
11 {
12     // 初始数组
13     vector<int> v{13, 6, 4, 11, 29};
14     cout << v << endl;
15
16     // 从小到大排序
17     sort(v.begin(), v.end());
18     cout << v << endl;
19
20     // 从大到小排序
21     sort(v.begin(), v.end(),
22          greater<int>());
23     cout << v << endl;
24
25     cout << hex;
26
27     auto hp = hash<int*>();
28     cout << "hash(nullptr) = "
29          << hp(nullptr) << endl;
30     cout << "hash(v.data()) = "
31          << hp(v.data()) << endl;
32     cout << "v.data() = "
33          << static_cast<void*>(v.data())
34          << endl;
35
36     auto hs = hash<string>();
37     cout << "hash(\"hello\") = "
38          << hs(string("hello")) << endl;
39     cout << "hash(\"hellp\") = "
40          << hs(string("hellp")) << endl;
41 }
```

在 MSVC 下的某次运行结果如下所示：

```
{ 13, 6, 4, 11, 29 }
{ 4, 6, 11, 13, 29 }
{ 29, 13, 11, 6, 4 }

hash(nullptr) = a8c7f832281a39c5
hash(v.data()) = 7a0bdfd7df0923d2
v.data() = 000001EFFB10EAE0
hash("hello") = a430d84680aabd0b
hash("hellp") = a430e54680aad322
```

可以看到，在这个实现里，空指针的哈希值是一个非零的数值，指针的哈希值也和指针的数值不一样。要注意不同的实现处理的方式会不一样。事实上，我的测试结果是 GCC、Clang 和 MSVC 对常见类型的哈希方式都各有不同。

在上面的例子里，我们同时可以看到，这两个函数对象的值不重要。我们甚至可以认为，每个 `less`（或 `greater` 或 `hash`）对象都是等价的。关键在于其类型。以 `sort` 为例，第三个参数的类型确定了其排序行为。

对于容器也是如此，函数对象的类型确定了容器的行为。

priority_queue

`priority_queue` 也是一个容器适配器。上一讲没有和其他容器适配器一起讲的原因就在于它用到了比较函数对象（默认是 `less`）。它和 `stack` 相似，支持 `push`、`pop`、`top` 等有限的操作，但容器内的顺序既不是后进先出，也不是先进先出，而是（部分）排序的结果。在使用缺省的 `less` 作为其 `Compare` 模板参数时，最大的数值会出现在容器的“顶部”。如果需要最小的数值出现在容器顶部，则可以传递 `greater` 作为其 `Compare` 模板参数。

下面的代码可以演示其功能：

```
1 #include <functional> // std::greater
```

 复制代码

```

2 #include <iostream>    // std::cout/endl
3 #include <memory>      // std::pair
4 #include <queue>        // std::priority_queue
5 #include <vector>       // std::vector
6 #include "output_container.h"
7
8 using namespace std;
9
10 int main()
11 {
12     priority_queue<
13         pair<int, int>,
14         vector<pair<int, int>>,
15         greater<pair<int, int>>>
16     > q;
17     q.push({1, 1});
18     q.push({2, 2});
19     q.push({0, 3});
20     q.push({9, 4});
21     while (!q.empty()) {
22         cout << q.top() << endl;
23         q.pop();
24     }
25 }

```

输出为:

```

(0, 3)
(1, 1)
(2, 2)
(9, 4)

```

关联容器

关联容器有 `set` (集合)、`map` (映射)、`multiset` (多重集) 和 `multimap` (多重映射)。跳出 C++ 的语境, `map` (映射) 的更常见的名字是关联数组和字典 [3], 而在 JSON 里直接被称为对象 (object)。在 C++ 外这些容器常常是无序的; 在 C++ 里关联容器则被认为是有序的。

我们可以通过以下的 `xeus-cling` 交互来体会一下。

```
1 #include <functional>
2 #include <map>
3 #include <set>
4 #include <string>
5 using namespace std;
```

[复制代码](#)

```
1 set<int> s{1, 1, 1, 2, 3, 4};
```

[复制代码](#)

```
1 s
```

[复制代码](#)

```
{ 1, 2, 3, 4 }
```

```
1 multiset<int, greater<int>> ms{1, 1, 1, 2, 3, 4};
```

[复制代码](#)

```
1 ms
```

[复制代码](#)

```
{ 4, 3, 2, 1, 1, 1 }
```

```
1 map<string, int> mp{
2     {"one", 1},
3     {"two", 2},
4     {"three", 3},
5     {"four", 4}
6 };
```

[复制代码](#)

```
1 mp
```

[复制代码](#)

```
{ "four" => 4, "one" => 1, "three" => 3, "two" => 2 }
```

```
1 mp.insert({"four", 4});
```

[复制代码](#)

```
1 mp
```

[复制代码](#)

```
{ "four" => 4, "one" => 1, "three" => 3, "two" => 2 }
```

```
1 mp.find("four") == mp.end()
```

[复制代码](#)

```
false
```

```
1 mp.find("five") == mp.end()
```

[复制代码](#)

```
(bool) true
```

```
1 mp["five"] = 5;
```

[复制代码](#)

```
1 mp
```


[复制代码](#)

```
{ "five" => 5, "four" => 4, "one" => 1, "three" => 3, "two" => 2  
}
```

```
1 multimap<string, int> mmp{
```

[复制代码](#)

```
2    {"one", 1},
3    {"two", 2},
4    {"three", 3},
5    {"four", 4}
6  };
```

 复制代码

```
1 mmp
```

```
{ "four" => 4, "one" => 1, "three" => 3, "two" => 2 }
```

 复制代码

```
1 mmp.insert({"four", -4});
```

 复制代码

```
1 mmp
```

```
{ "four" => 4, "four" => -4, "one" => 1, "three" => 3, "two" =>
2 }
```

可以看到，关联容器是一种有序的容器。名字带“multi”的允许键重复，不带的不允许键重复。set 和 multiset 只能用来存放键，而 map 和 multimap 则存放一个个键值对。

与序列容器相比，关联容器没有前、后的概念及相关的成员函数，但同样提供 insert、emplace 等成员函数。此外，关联容器都有 find、lower_bound、upper_bound 等查找函数，结果是一个迭代器：

find(k) 可以找到任何一个等价于查找键 k 的元素 ($!(x < k \ || \ k < x)$)

lower_bound(k) 找到第一个不小于查找键 k 的元素 ($!(x < k)$)

upper_bound(k) 找到第一个大于查找键 k 的元素 ($k < x$)

[📄 复制代码](#)

```
1 mp.find("four")->second
```

4

[📄 复制代码](#)

```
1 mp.lower_bound("four")->second
```

4

[📄 复制代码](#)

```
1 (--mp.upper_bound("four"))->second
```

4

[📄 复制代码](#)

```
1 mmp.lower_bound("four")->second
```

4

[📄 复制代码](#)

```
1 (--mmp.upper_bound("four"))->second
```


-4

如果你需要在 `multimap` 里精确查找满足某个键的区间的话，建议使用 `equal_range`，可以一次性取得上下界（半开半闭）。如下所示：

[📄 复制代码](#)

```
1 #include <tuple>
2 multimap<string, int>::iterator
3   lower, upper;
```

```
4 std::tie(lower, upper) =  
5 mmp.equal_range("four");
```

 复制代码


```
1 (lower != upper) // 检测区间非空
```

true

 复制代码

```
1 lower->second
```

4

 复制代码

```
1 (--upper)->second
```

-4

如果在声明关联容器时没有提供比较类型的参数，缺省使用 `less` 来进行排序。如果键的类型提供了比较算符 `<` 的重载，我们不需要做任何额外的工作。否则，我们就需要对键类型进行 `less` 的特化，或者提供一个其他的函数对象类型。

对于自定义类型，我推荐尽量使用标准的 `less` 实现，通过重载 `<`（及其他标准比较运算符）对该类型的对象进行排序。存储在关联容器中的键一般应满足严格弱序关系（strict weak ordering; [4]），即：

对于任何该类型的对象 x ：! $(x < x)$ （非自反）

对于任何该类型的对象 x 和 y ：如果 $x < y$ ，则 $!(y < x)$ （非对称）

对于任何该类型的对象 x 、 y 和 z ：如果 $x < y$ 并且 $y < z$ ，则 $x < z$ （传递性）

对于任何该类型的对象 x 、 y 和 z ：如果 x 和 y 不可比 ($!(x < y)$ 并且 $!(y < x)$) 并且 y 和 z 不可比，则 x 和 z 不可比（不可比的传递性）

大部分情况下，类型是可以满足这些条件的，不过：

如果类型没有一般意义上的大小关系（如复数），我们一定要别扭地定义一个大小关系吗？

通过比较来进行查找、插入和删除，复杂度为对数 $O(\log(n))$ ，有没有达到更好的性能的方法？

无序关联容器

从 C++11 开始，每一个关联容器都有一个对应的无序关联容器，它们是：

`unordered_set`

`unordered_map`

`unordered_multiset`

`unordered_multimap`

这些容器和关联容器非常相似，主要的区别就在于它们是“无序”的。这些容器不要求提供一个排序的函数对象，而要求一个可以计算哈希值的函数对象。你当然可以在声明容器对象时手动提供这样一个函数对象类型，但更常见的情况是，我们使用标准的 `hash` 函数对象及其特化。

下面是一个示例（这次我们暂不使用 `xeus-cling`，因为它在输出复数时有限制，不能显示其数值）：

```
1 #include <complex>           // std::complex
2 #include <iostream>          // std::cout/endl
3 #include <unordered_map>      // std::unordered_map
4 #include <unordered_set>      // std::unordered_set
5 #include "output_container.h"
6
7 using namespace std;
8
```

 复制代码

```

 9 namespace std {
10
11 template <typename T>
12 struct hash<complex<T>> {
13     size_t
14     operator()(const complex<T>& v) const
15         noexcept
16     {
17         hash<T> h;
18         return h(v.real()) + h(v.imag());
19     }
20 };
21
22 } // namespace std
23
24 int main()
25 {
26     unordered_set<int> s{
27         1, 1, 2, 3, 5, 8, 13, 21
28     };
29     cout << s << endl;
30
31     unordered_map<complex<double>,
32                 double>
33     umc{{{1.0, 1.0}, 1.4142},
34         {{3.0, 4.0}, 5.0}};
35     cout << umc << endl;
36 }

```

输出可能是（顺序不能保证）：

```

{ 21, 5, 8, 3, 13, 2, 1 }
{ (3,4) => 5, (1,1) => 1.4142 }

```

请注意我们在 `std` 名空间中添加了特化，这是少数用户可以向 `std` 名空间添加内容的情况之一。正常情况下，向 `std` 名空间添加声明或定义是禁止的，属于未定义行为。

从实际的工程角度，无序关联容器的主要优点在于其性能。关联容器和 `priority_queue` 的插入和删除操作，以及关联容器的查找操作，其复杂度都是 $O(\log(n))$ ，而无序关联容器的实现使用哈希表 [5]，可以达到平均 $O(1)$ ！但这取决于我们是否使用了一个好的哈希函数：在哈希函数选择不当的情况下，无序关联容器的插入、删除、查找性能可能成为最差情况的 $O(n)$ ，那就比关联容器糟糕得多了。

array

我们讲的最后一个容器是 C 数组的替代品。C 数组在 C++ 里继续存在，主要是为了保留和 C 的向后兼容性。C 数组本身和 C++ 的容器相差是非常大的：


C 数组没有 `begin` 和 `end` 成员函数（虽然可以使用全局的 `begin` 和 `end` 函数）

C 数组没有 `size` 成员函数（得用一些模板技巧来获取其长度）

C 数组作为参数有退化行为，传递给另外一个函数后那个函数不再能获得 C 数组的长度和结束位置


在 C 的年代，大家有时候会定义这样一个宏来获得数组的长度：

```
1 #define ARRAY_LEN(a) \
2     (sizeof(a) / sizeof((a)[0]))
```

 复制代码

如果在一个函数内部对数组参数使用这个宏，结果肯定是错的。现在 GCC 会友好地发出警告：


```
1 void test(int a[8])
2 {
3     cout << ARRAY_LEN(a) << endl;
4 }
```

 复制代码

```
warning: sizeof on array function parameter will return size of 'int *' instead of
'int [8]' [-Wsizeof-array-argument]
    cout << ARRAY_LEN(a) << endl;
```

C++17 直接提供了一个 `size` 方法，可以用于提供数组长度，并且在数组退化成指针的情况下会直接失败：

```
1 #include <iostream> // std::cout/endl
2 #include <iterator> // std::size
```

 复制代码

```

3
4 void test(int arr[])
5 {
6     // 不能编译
7     // std::cout << std::size(arr)
8     //             << std::endl;
9 }
10
11 int main()
12 {
13     int arr[] = {1, 2, 3, 4, 5};
14     std::cout << "The array length is "
15               << std::size(arr)
16               << std::endl;
17     test(arr);
18 }

```

此外，C 数组也没有良好的复制行为。你无法用 C 数组作为 `map` 或 `unordered_map` 的键类型。下面的代码演示了失败行为：

 复制代码

```

1 #include <map> // std::map
2
3 typedef char mykey_t[8];
4
5 int main()
6 {
7     std::map<mykey_t, int> mp;
8     mykey_t mykey{"hello"};
9     mp[mykey] = 5;
10    // 轰，大段的编译错误
11 }

```

如果不用 C 数组的话，我们该用什么来替代呢？

我们三个可以考虑的选项：


如果数组较大的话，应该考虑 `vector`。`vector` 有最大的灵活性和不错的性能。

对于字符串数组，当然应该考虑 `string`。

如果数组大小固定（C 的数组在 C++ 里本来就是大小固定的）并且较小的话，应该考虑 `array`。`array` 保留了 C 数组在栈上分配的特点，同时，提供了 `begin`、`end`、`size`

等通用成员函数。

`array` 可以避免 C 数组的种种怪异行径。上面的失败代码，如果使用 `array` 的话，稍作改动就可以通过编译：

 复制代码

```
1 #include <array>          // std::array
2 #include <iostream>       // std::cout/endl
3 #include <map>             // std::map
4 #include "output_container.h"
5
6 typedef std::array<char, 8> mykey_t;
7
8 int main()
9 {
10     std::map<mykey_t, int> mp;
11     mykey_t mykey{"hello"};
12     mp[mykey] = 5; // OK
13     std::cout << mp << std::endl;
14 }
```

输出则是意料之中的：

```
{ hello => 5 }
```

内容小结

本讲介绍了 C++ 的两个常用的函数对象，`less` 和 `hash`；然后介绍了用到这两个函数对象的容器适配器、关联容器和无序关联容器；最后，通过例子展示了为什么我们应当避免 C 数组而考虑使用 `array`。通过这两讲，我们已经完整地了解了 C++ 提供的标准容器。

课后思考

请思考一下：

1. 为什么大部分容器都提供了 `begin`、`end` 等方法？
2. 为什么容器没有继承一个公用的基类？

欢迎留言和我交流你的看法。

参考资料

[1] cppreference.com, “Containers library” .

[🔗 https://en.cppreference.com/w/cpp/container](https://en.cppreference.com/w/cpp/container)

[1a] cppreference.com, “容器库” . [🔗 https://zh.cppreference.com/w/cpp/container](https://zh.cppreference.com/w/cpp/container)

[2] cppreference.com, “Explicit (full) template specialization” .

[🔗 https://en.cppreference.com/w/cpp/language/template_specialization](https://en.cppreference.com/w/cpp/language/template_specialization)

[2a] cppreference.com, “显式（全）模板特化” .

[🔗 https://zh.cppreference.com/w/cpp/language/template_specialization](https://zh.cppreference.com/w/cpp/language/template_specialization)

[3] Wikipedia, “Associative array” .

[🔗 https://en.wikipedia.org/wiki/Associative_array](https://en.wikipedia.org/wiki/Associative_array)

[3a] 维基百科, “关联数组” . [🔗 https://zh.wikipedia.org/zh-cn/ 关联数组](https://zh.wikipedia.org/zh-cn/关联数组)

[4] Wikipedia, “Weak ordering” . [🔗 https://en.wikipedia.org/wiki/Weak_ordering](https://en.wikipedia.org/wiki/Weak_ordering)

[5] Wikipedia, “Hash table” . [🔗 https://en.wikipedia.org/wiki/Hash_table](https://en.wikipedia.org/wiki/Hash_table)

[5a] 维基百科, “哈希表” . [🔗 https://zh.wikipedia.org/zh-cn/ 哈希表](https://zh.wikipedia.org/zh-cn/哈希表)

点击查看 

打卡学习 C++ 拒绝从入门到放弃



PC 端用户扫码参与



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04 | 容器汇编 I：比较简单的若干容器

下一篇 06 | 异常：用还是不用，这是个问题

精选留言 (13)

 写留言



qinsi

2019-12-07

既然关联容器的key需要满足strict weak ordering，那么sort的比较函数是不是也需要满足？比如sort(v.begin(), v.end(), less_equal<int>());是否可行？

展开 

作者回复：是，也必须满足。标准里明确这么要求了。

如果你传了 less_equal 这样的条件，结果可能是正确，也可能是错误，也可能是程序崩溃。具体发生什么，取决于排序算法的实现。出了问题，就是调用者的锅，不是 sort 的 bug。

以冒泡排序为例，我试了一下，如果是用检验有没有交换来决定是否退出排序，那么，在元素有重复的情况下使用 less_equal 来排序，会导致死循环。



lyfei

2019-12-08

老师你好，那为什么unordered_map会使用到operator==的呢？

我感觉他不是应该把数据转到hash值，然后保存起来，也感觉没有比较的过程，哪个地方体现了==这个运算符呀？

展开

作者回复: 不是。哈希是哈希，哈希可能有冲突的，相同哈希值也要看键是相同还是不同，来决定是覆盖还是加一项。去看一下数据结构里面的哈希。



中年男子

2019-12-06

1、begin、end是迭代器，主要用于对不同类型的容器提供统一的遍历容器的辅助

2不同容器内存分配方式不同，实现不同，基类方法无法做到统一，非要用继承只能定义虚函数

多用组合、少用继承（抖个机灵）

展开

作者回复: 哈哈，是这样的。



lyfei

2019-12-07

老师你好，我在使用无序容器unordered_map时，key是使用了自定义的类型，所以需要给hash进行特化，但是我编译的时候出了问题：

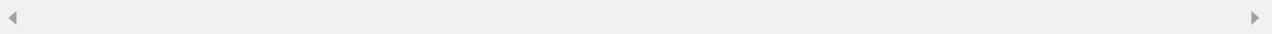
```
"/usr/include/c++/7/bits/stl_function.h:356:20: error: no match for 'operator=='  
(operand types are 'const Test<int>' and 'const Test<int>')  
    { return __x == __y; }" ...
```

展开

作者回复: 正如错误信息提示的，你的类没有定义相等比较。把定义改成下面这样子就可以了（完整起见，我也加了!= 的定义）：

```
template <typename T>
```

```
class Test {
public:
    T a_t;
    bool operator==(const Test& rhs) const
    {
        return a_t == rhs.a_t;
    }
    bool operator!=(const Test& rhs) const
    {
        return !operator==(rhs);
    }
};
```



3



糖

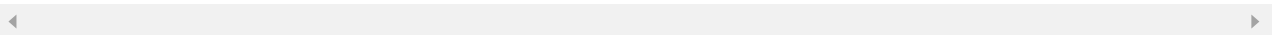
2019-12-07

1. 首先是为了遍历容器方便，其次为了保证std接口的一致性
2. 我认为是不必要的，因为，如果用类的继承一般会产生虚函数，这也就导致多存储一个虚函数表，在内存上产生了不必要的浪费。

展开 ∨

作者回复: 基本正确。

2 实际上还是因为这儿继承真的没啥用。有了泛型，继承基本是种浪费。而且，继承用基类的指针或引用才有用，而C++里的容器类一般都是当值类型来用的。



糖

2019-12-07

老师，我又有问题了！

、、、

```
template <typename T>
struct hash<complex<T>> {
    size_t operator()(const complex<T>& v) const noexcept { ...
```

展开 ∨

作者回复: 超出范围，在此处我们是不在乎的——我们不需要一个正确的数值。哈希值本来就是有冲撞的。



贾陆华

2019-12-07

1. 看到一个注释笔误，是从大到小吧

// 从小到大排序

```
sort(v.begin(), v.end(),
```

```
    greater<int>());
```

```
cout << v << endl;...
```

展开 ▾

作者回复: 1. 哈，还真的是。典型的拷贝粘贴错误.....

2. 想一想，C++ 里到处是半开半闭区间。右界一直是会超出有效范围的。

💬 1



传说中的成大大

2019-12-06

第一问 大概是因为可以通过begin()方法的返回值迭代到end() 就像数组或者链表 等等都可以从头遍历到尾 这也是为啥子 有些线性容器 删除以后返回的是下一个元素的迭代器 而map set这种容器无法通过迭代器进行迭代 所以调用erase函数返回void

第二问 大概是因为各个容器的存储方式不太一样吧 所以导致操作不一样 像priority_queue 和queue他们的操作方式就不一样 queue插入或者删除元素只需要移动指针或者下标...

展开 ▾

作者回复: 一、前面正确。关于map错了。可以迭代，并且现在erase已经返回迭代器了（C++98 时不行）。

二、跟这个关系不大。OO语言里也是推荐继承接口而不是实现的。



方阳

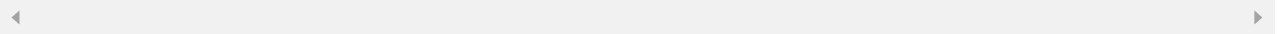
2019-12-06

1.方便算法遍历容器

2.容器内部也有一些继承和复合，容器是独立的组件，没必要继承公用基类。

展开 ▾

作者回复: 好, 基本正确。



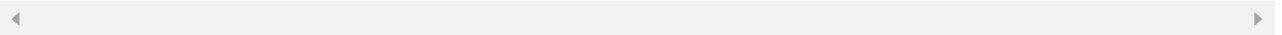
罗乾林

2019-12-06

- 1、为algorithm服务
- 2、继承必然要用虚函数, 性能有损失

作者回复: 1 是激发大家思考, 现在多想想就行了。

2 这也算是个点。我再等等其他回答。😊



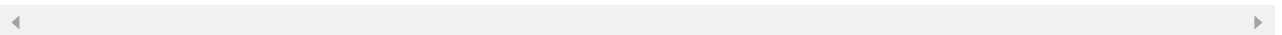
李义盛爱王唐硕

2019-12-06

2.继承是强耦合。 继承导致关系混乱。

展开 ∨

作者回复: 算是一个点。真要继承, 不会更好用, 因为容器的接口差异往往很小, 这儿多一个, 那儿少一个, 要多少接口才能表达啊。



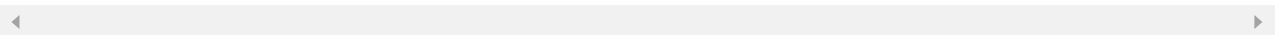
总统老唐

2019-12-06

又更新一课, 之前还有同学说更新慢, 现在看, 是更新太快了, 因为都是干货, 每一课都要花大力气学

作者回复: 我还真希望更新慢点呢, 但编辑不肯啊.....写文字、造例子都挺花力气的。

第 4、5 讲应该还好吧, 烧脑的东西不多。再往下可能会又比较干些, 尤其到了模板元编程的部分。





hello world

2019-12-06

1. 因为统一要为迭代器服务
2. 等大佬们!

作者回复: 同学动作很快, 但有点调皮啊。 😊

