容器适配器

2021年7月28日 9:51

容器适配器

1、介绍

容器适配器是一个封装了序列容器的类模板，它在一般序列容器的基础上提供了一些不同的功能。之所以称作适配器类，是因为它可以通过适配容器现有的接口来提供不同的功能。

2、分类

C++中定义了3种容器适配器，它们让容器提供的接口变成了我们常用的3种数据结构：栈stack、队列queue和优先队列priority_queue。默认情况下，栈和队列都是基于deque实现的，而优先级队列则是基于vector实现的。当然，我们也可以指定自己的实现方式。但是由于数据结构的关系，我们也不能胡乱指定。栈的特点是后进先出，所以它关联的基本容器可以是任意一种顺序容器，因为这些容器类型结构都可以提供栈的操作有求，它们都提供了push_back、pop_back和back操作。 队列queue的特点是先进先出，适配器要求其关联的基础容器必须提供pop_front操作，因此其不能建立在vector容器上；对于优先级队列，由于它要求支持随机访问的功能，所以可以建立在vector或者deque上，不能建立在list上。

（1）stack<T>：是一个封装了deque<T>容器的适配器类模板，默认实现的是一个后入先出（Last-In-First-Out，LIFO）的压入栈。stack<T>模板定义在头文件stack中。

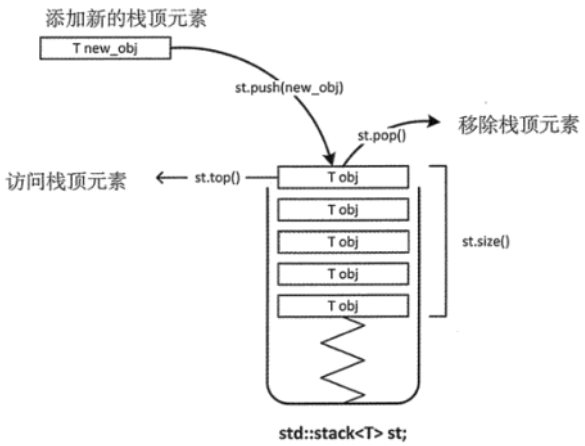
（2）queue<T>：是一个封装了deque<T>容器的适配器类模板，默认实现的是一个先入先出（First-In-First-Out，FIFO）的队列。可以为它指定一个符合确定条件的基础容器。queue<T>模板定义在头文件queue中。

（3）priority_queue<T>：是一个封装了vector<T>容器的适配器类模板，默认实现的是一个会对元素排序，从而保证最大元素总在队列最前面的队列。priority_queue<T> 模板定义在头文件 queue 中。

3、stack（STL stack）容器适配器用法

3.1介绍

stack 栈适配器是一种单端开口的容器，实际上该容器模拟的就是栈存储结构，即无论是向里存数据还是从中取数据，都只能从这一个开口实现操作。



stack 适配器的开头端通常称为栈顶。由于数据的存和取只能从栈顶处进行操作，因此对于存取数据，stack 适配器有这样的特性，即每次只能访问适配器中位于最顶端的元素，也只有移除 stack 顶部的元素之后，才能访问位于栈中的元素。

3.2 stack使用介绍

3.2.1 stack容器适配器的创建

由于 stack 适配器以模板类 stack<T,Container=deque<T>>（其中 T 为存储元素的类型，Container 表示底层容器的类型）的形式位于<stack>头文件中，并定义在 std 命名空间里。因此，在创建该容器之前，程序中应包含以下 2 行代码：

```
#include <stack>
using namespace std;
```

（1）创建一个不包含任何元素的 stack 适配器，并采用默认的 deque 基础容器：

```
std::stack<int> values;
```

上面这行代码，就成功创建了一个可存储 int 类型元素，底层采用 deque 基础容器的 stack 适配器。

（2）上面提到，stack<T,Container=deque<T>> 模板类提供了 2 个参数，通过指定第二个模板类型参数，我们可以使用除 deque 容器外的其它序列式容器，只要该容器支持 empty()、size()、back()、push_back()、pop_back() 这 5 个成员函数即可。序列式容器中同时包含这 5 个成员函数的，有 vector、deque 和 list 这 3 个容器。因此，stack 适配器的基础容器可以是它们 3 个中任何一个。例如，下面展示了如何定义一个使用 list 基础容器的 stack 适配器：

```
std::stack<std::string, std::list<int>> values;
```

（3）可以用一个基础容器来初始化 stack 适配器，只要该容器的类型和 stack 底层使用的基础容器类型相同即可。例如：

```
std::list<int> values {1, 2, 3};
std::stack<int,std::list<int>> my_stack (values);
```

注意，初始化后的 my_stack 适配器中，栈顶元素为 3，而不是 1。另外在第 2 行代码中，stack 第 2 个模板参数必须显式指定为 list<int>（必须为 int 类型，和存储类型保持一致），否则 stack 底层将默认使用 deque 容器，也就无法用 lsit 容器的内容来初始化 stack 适配器。

（4）还可以用一个 stack 适配器来初始化另一个 stack 适配器，只要它们存储的元素类型以及底层采用的基础容器类型相同即可。例如：

```
std::list<int> values{ 1, 2, 3 };
std::stack<int, std::list<int>> my_stack1(values);
std::stack<int, std::list<int>> my_stack=my_stack1;
//std::stack<int, std::list<int>> my_stack(my_stack1);
```

可以看到，和使用基础容器不同，使用 stack 适配器给另一个 stack 进行初始化时，有 2 种方式，使用哪一种都可以。注意，第 3、4 种初始化方法中，my_stack 适配器的数据是经过拷贝得来的，也就是说，操作 my_stack 适配器，并不会对 values 容器以及 my_stack1 适配器有任何影响；反过来也是如此。

3.2.2 stack容器适配器支持的成员函数

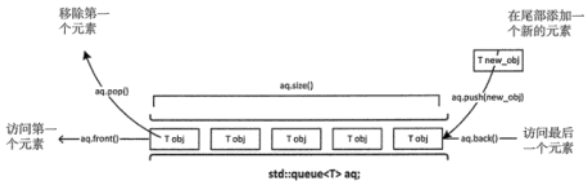
成员函数	功能
empty()	当 stack 栈中没有元素时，该成员函数返回 true；反之，返回 false。
size()	返回 stack 栈中存储元素的个数。
top()	返回一个栈顶元素的引用，类型为 T&。如果栈为空，程序会报错。
push(const T& val)	先复制 val，再将 val 副本压入栈顶。这是通过调用底层容器的 push_back() 函数完成的。
push(T&& obj)	以移动元素的方式将其压入栈顶。这是通过调用底层容器的有右值引用参数的 push_back() 函数完成的。
pop()	弹出栈顶元素。
emplace(arg...)	arg... 可以是一个参数，也可以是多个参数，但它们都只用于构造一个对象，并在栈顶直接生成该对象，作为新的栈顶元素。
swap(stack<T> & other_stack)	将两个 stack 适配器中的元素进行互换，需要注意的是，进行互换的 2 个 stack 适配器中存储的元素类型以及底层采用的基础容器类型，都必须相同。

```
#include <iostream>
#include <stack>
#include <list>
using namespace std;
int main()
{
    //构建 stack 容器适配器
    list<int> values{ 1, 2, 3 };
    stack<int, list<int>> my_stack(values);
    //查看 my_stack 存储元素的个数
    cout << "size of my_stack: " << my_stack.size() << endl;
    //将 my_stack 中存储的元素依次弹栈，直到其为空
    while (!my_stack.empty())
    {
        cout << my_stack.top() << endl;
        //将栈顶元素弹栈
        my_stack.pop();
    }
    return 0;
}
size of my_stack: 3
3
2
1
```

4、queue容器适配器用法

4.1介绍

和 stack 栈容器适配器不同，queue 容器适配器有 2 个开口，其中一个开口专门用来输入数据，另一个专门用来输出数据。



这种存储结构最大的特点是，最先进入 queue 的元素，也可以最先从 queue 中出来，即用此容器适配器存储数据具有“先进先出（简称 "FIFO" ）”的特点，因此 queue 又称为队列适配器。

4.2 queue使用介绍

4.2.1 queue容器适配器的创建

queue 容器适配器以模板类 queue<T,Container=deque<T>>（其中 T 为存储元素的类型，Container 表示底层容器的类型）的形式位于<queue>头文件中，并定义在 std 命名空间里。因此，在创建该容器之前，程序中应包含以下 2 行代码：

```
#include <queue>
using namespace std;
```

（1）创建一个空的 queue 容器适配器，其底层使用的基础容器选择默认的 deque 容器

```
std::queue<int> values;
```

通过此行代码，就可以成功创建一个可存储 int 类型元素，底层采用 deque 容器的 queue 容器适配器。

（2）当然，也可以手动指定 queue 容器适配器底层采用的基础容器类型。queue 容器适配器底层容器可以选择 deque 和 list。作为 queue 容器适配器的基础容器，其必须提供 front()、back()、push_back()、pop_front()、empty() 和 size() 这几个成员函数，符合条件的序列式容器仅有 deque 和 list。例如，下面创建了一个使用 list 容器作为基础容器的空 queue 容器适配器：

```
std::queue<int, std::list<int>> values;
```

注意，在手动指定基础容器的类型时，其存储的数据类型必须和 queue 容器适配器存储的元素类型保持一致。

（3）可以用基础容器来初始化 queue 容器适配器，只要该容器类型和 queue 底层使用的基础容器类型相同即可。

```
std::deque<int> values{1,2,3};
std::queue<int> my_queue(values);
```

由于 my_queue 底层采用的是 deque 容器，和 values 类型一致，且存储的也都是 int 类型元素，因此可以用 values 对 my_queue 进行初始化。

（4）还可以直接通过 queue 容器适配器来初始化另一个 queue 容器适配器，只要它们存储的元素类型以及底层采用的基础容器类型相同即可。

```
std::deque<int> values{1,2,3};
std::queue<int> my_queue1(values);
std::queue<int> my_queue(my_queue1);
//或者使用
//std::queue<int> my_queue = my_queue1;
```

值得一提的是，第 3、4 种初始化方法中 my_queue 容器适配器的数据是经过拷贝得来的，也就是说，操作 my_queue 容器适配器中的数据，并不会对 values 容器以及 my_queue1 容器适配器有任何影响；反过来也是如此。

4.2.2 queue容器适配器支持的成员函数

成员函数	功能
empty()	如果 queue 中没有元素的话，返回 true。
size()	返回 queue 中元素的个数。
front()	返回 queue 中第一个元素的引用。如果 queue 是常量，就返回一个常引用；如果 queue 为空，返回值是未定义的。
back()	返回 queue 中最后一个元素的引用。如果 queue 是常量，就返回一个常引用；如果 queue 为空，返回值是未定义的。
push(const T& obj)	在 queue 的尾部添加一个元素的副本。这是通过调用底层容器的成员函数 push_back() 来完成的。
emplace()	在 queue 的尾部直接添加一个元素。
push(T&& obj)	以移动的方式在 queue 的尾部添加元素。这是通过调用底层容器的具有右值引用参数的成员函数 push_back() 来完成的。
pop()	删除 queue 中的第一个元素。
swap(queue<T> &other_queue)	将两个 queue 容器适配器中的元素进行互换，需要注意的是，进行互换的 2 个 queue 容器适配器中存储的元素类型以及底层采用的基础容器类型，都必须相同。

和 stack 一样，queue 也没有迭代器，因此访问元素的唯一方式是遍历容器，通过不断移除访问过的元素，去访问下一个元素。

```
#include <iostream>
#include <queue>
#include <list>
using namespace std;
int main()
{
    //构建 queue 容器适配器
    std::deque<int> values{ 1,2,3 };
    std::queue<int> my_queue(values);//{1,2,3}
    //查看 my_queue 存储元素的个数
    cout << "size of my_queue: " << my_queue.size() << endl;
    //访问 my_queue 中的元素
    while (!my_queue.empty())
    {
        cout << my_queue.front() << endl;
        //访问过的元素出队列
        my_queue.pop();
    }
    return 0;
}
size of my_queue: 3
1
2
3
```

5、priority_queue容器适配器用法

5.1介绍

priority_queue 容器适配器模拟的也是队列这种存储结构，即使用此容器适配器存储元素只能“从一端进（称为队尾），从另一端出（称为队头）”，且每次只能访问 priority_queue 中位于队头的元素。但是，priority_queue 容器适配器中元素的存和取，遵循的并不是 “First in,First out”（先入先出）原则，而是“First in，Largest out”原则。直白的翻译，指的就是先进队列的元素并不一定先出队列，而是优先级最大的元素最先出队列。每个 priority_queue 容器适配器在创建时，都制定了一种排序规则。根据此规则，该容器适配器中存储的元素就有了优先级高低之分。举个例子，假设当前有一个 priority_queue 容器适配器，其制定的排序规则是按照元素值从大到小进行排序。根据此规则，自然是 priority_queue 中值最大的元素的优先级最高。priority_queue 容器适配器为了保证每次从队头移除的都是当前优先级最高的元素，每当有新元素进入，它都会根据既定的排序规则找到优先级最高的元素，并将其移动到队列的队头；同样，当 priority_queue 从队头移除出一个元素之后，它也会再找到当前优先级最高的元素，并将其移动到队头。基于 priority_queue 的这种特性，因此该容器适配器有被称为优先级队列。STL 中，priority_queue 容器适配器的定义如下：

```
template <typename T,
        typename Container=std::vector<T>,
        typename Compare=std::less<T> >
class priority_queue{
    //......
}
```

可以看到，priority_queue 容器适配器模板类最多可以传入 3 个参数，它们各自的含义如下：

- typename T：指定存储元素的具体类型；
- typename Container：指定 priority_queue 底层使用的基础容器，默认使用 vector 容器。
- typename Compare：指定容器中评定元素优先级所遵循的排序规则，默认使用std::less<T>按照元素值从大到小进行排序，还可以使用std::greater<T>按照元素值从小到大排序，但更多情况下是使用自定义的排序规则。

作为 priority_queue 容器适配器的底层容器，其必须包含 empty()、size()、front()、push_back()、pop_back() 这几个成员函数，STL 序列式容器中只有 vector 和 deque 容器符合条件。

5.2 priority_queue使用介绍

5.2.1 priority_queue容器适配器的创建

由于 priority_queue 容器适配器模板位于<queue>头文件中，并定义在 std 命名空间里，因此在试图创建该类型容器之前，程序中需包含以下 2 行代码：

```
#include <queue>
using namespace std;
```

（1）创建一个空的 priority_queue 容器适配器，第底层采用默认的 vector 容器，排序方式也采用默认的 std::less<T> 方法

```
std::priority_queue<int> values;
```

（2）可以使用普通数组或其它容器中指定范围内的数据，对 priority_queue 容器适配器进行初始化

```
//使用普通数组
int values[]{4,1,3,2};
std::priority_queue<int>copy_values(values,values+4);//{4,2,3,1}
//使用序列式容器
std::array<int,4>values{ 4,1,3,2 };
std::priority_queue<int>copy_values(values.begin(),values.end());//{4,2,3,1}
```

注意，以上 2 种方式必须保证数组或容器中存储的元素类型和 priority_queue 指定的存储类型相同。另外，用来初始化的数组

或容器中的数据不需要有序，priority_queue 会自动对它们进行排序。

（3）还可以手动指定 priority_queue 使用的底层容器以及排序规则，比如：

```
int values[]{ 4,1,2,3 };
std::priority_queue<int, std::deque<int>, std::greater<int> >copy_values(values, values+4);//{1,3,2,4}
```

事实上，std::less<T> 和 std::greater<T> 适用的场景是有限的，更多场景中我们会使用自定义的排序规则。

5.2.2 priority_queue容器适配器支持的成员函数

成员函数	功能
empty()	如果 priority_queue 为空的话，返回 true；反之，返回 false。
size()	返回 priority_queue 中存储元素的个数。
top()	返回 priority_queue 中第一个元素的引用形式。
push(const T& obj)	根据既定的排序规则，将元素 obj 的副本存储到 priority_queue 中适当的位置。
push(T&& obj)	根据既定的排序规则，将元素 obj 移动存储到 priority_queue 中适当的位置。
emplace(Args&&... args)	Args&&... args 表示构造一个存储类型的元素所需要的数据（对于类对象来说，可能需要多个数据构造出一个对象）。此函数的功能是根据既定的排序规则，在容器适配器适当的位置直接生成该新元素。
pop()	移除 priority_queue 容器适配器中第一个元素。
swap(priority_queue<T>& other)	将两个 priority_queue 容器适配器中的元素进行互换，需要注意的是，进行互换的 2 个 priority_queue 容器适配器中存储的元素类型以及底层采用的基础容器类型，都必须相同。

和 queue 一样，priority_queue 也没有迭代器，因此访问元素的唯一方式是遍历容器，通过不断移除访问过的元素，去访问下一个元素。

```
#include <iostream>
#include <queue>
#include <array>
#include <functional>
using namespace std;
int main()
{
    //创建一个空的priority_queue容器适配器
    std::priority_queue<int>values;
    //使用 push() 成员函数向适配器中添加元素
    values.push(3);//{3}
    values.push(1);//{3,1}
    values.push(4);//{4,1,3}
    values.push(2);//{4,2,3,1}
    //遍历整个容器适配器
    while (!values.empty())
    {
        //输出第一个元素并移除。
        std::cout << values.top()<<" ";
        values.pop();//移除队头元素的同时，将剩余元素中优先级最大的移至队头
    }
    return 0;
}
```

```
4 3 2 1
```

来自 <https://blog.csdn.net/qq_21989927/article/details/109392756>