

6.2 基于锁的并发数据结构

基于锁的并发数据结构设计，需要确保访问线程持有锁的时间最短。对于只有一个互斥量的数据结构来说，这十分困难。需要保证数据不被锁之外的操作所访问到，并且还要保证不会在固有结构上产生条件竞争(如第3章所述)。当你使用多个互斥量来保护数据结构中不同的区域时，问题会暴露的更加明显，当操作需要获取多个互斥锁时，就有可能产生死锁。所以，在设计时，使用多个互斥量时需要格外小心。

在本节中，你将使用6.1.1节中的指导建议，来设计一些简单的数据结构——使用互斥量和锁的方式来保护数据。每一个例子中，都是在保证数据结构是线程安全的前提下，对数据结构并发访问的概率(机会)进行提高。

我们先来看看在第3章中栈的实现，这个实现就是一个十分简单的数据结构，它只使用了一个互斥量。但是，这个结构是线程安全的吗？它离真正的并发访问又有多远呢？

6.2.1 线程安全栈——使用锁

我们先把第3章中线程安全的栈拿过来看看：(这里试图实现一个线程安全版的 `std::stack<>`)

清单6.1 线程安全栈的类定义

```
1  #include <exception>
2
3  struct empty_stack: std::exception
4  {
5      const char* what() const throw();
6  };
7
8  template<typename T>
9  class threadsafe_stack
10 {
11 private:
12     std::stack<T> data;
13     mutable std::mutex m;
14 public:
```

```
15     threadsafe_stack() {}
16     threadsafe_stack(const threadsafe_stack& other)
17     {
18         std::lock_guard<std::mutex> lock(other.m);
19         data=other.data;
20     }
21
22     threadsafe_stack& operator=(const threadsafe_stack&) = delete;
23
24     void push(T new_value)
25     {
26         std::lock_guard<std::mutex> lock(m);
27         data.push(std::move(new_value)); // 1
28     }
29     std::shared_ptr<T> pop()
30     {
31         std::lock_guard<std::mutex> lock(m);
32         if(data.empty()) throw empty_stack(); // 2
33         std::shared_ptr<T> const res(
34             std::make_shared<T>(std::move(data.top()))); // 3
35         data.pop(); // 4
36         return res;
37     }
38     void pop(T& value)
39     {
40         std::lock_guard<std::mutex> lock(m);
41         if(data.empty()) throw empty_stack();
42         value=std::move(data.top()); // 5
43         data.pop(); // 6
44     }
45     bool empty() const
46     {
47         std::lock_guard<std::mutex> lock(m);
48         return data.empty();
49     }
50 };
```

来看看指导意见是如何应用的。

首先，互斥量`m`能保证基本的线程安全，那就是对每个成员函数进行加锁保护。这就保证在同一时间内，只有一个线程可以访问到数据，所以能够保证，数据结构的“不变量”被破坏时，不会被其他线程看到。

其次，在`empty()`和`pop()`成员函数之间会存在潜在的竞争，不过代码会在`pop()`函数上锁时，显式的查询栈是否为空，所以这里的竞争是非恶性的。`pop()`通过对弹出值的直接返回，就可避免 `std::stack<>` 中`top()`和`pop()`两成员函数之间的潜在竞争。

再次，这个类中也有一些异常源。对互斥量上锁可能会抛出异常，因为上锁操作是每个成员函数所做的第一个操作，所以这是极其罕见的(因为这意味着问题不在锁上，就是在系统资源上)。因无数据修改，所以其是安全的。因解锁一个互斥量是不会失败的，所以段代码很安全，并且使用 `std::lock_guard<>` 也能保证互斥量上锁的状态。

对`data.push()`①的调用可能会抛出一个异常，不是拷贝/移动数据值时，就是内存不足的时候。不管是哪种，`std::stack<>` 都能保证其实安全的，所以这里也没有问题。

在第一个重载`pop()`中，代码可能会抛出一个`empty_stack`的异常②，不过数据没有被修改，所以其是安全的。对于`res`的创建③，也可能会抛出一个异常，这有两方面的原因：对

`std::make_shared` 的调用，可能无法分配出足够的内存去创建新的对象，并且内部数据需要对新对象进行引用；或者，在拷贝或移动构造到新分配的内存中返回时抛出异常。两种情况下，`C++`运行库和标准库能确保这里不会出现内存泄露，并且新创建的对象(如果有的话)都能被正确销毁。因为没有对栈进行任何修改，所以这里也不会有问题。当调用`data.pop()`④时，其能确保不抛出异常，并且返回结果，所以这个重载`pop()`函数“异常-安全”。

第二个重载`pop()`类似，除了在拷贝赋值或移动赋值的时候会抛出异常⑤，当构造一个新对象和一个 `std::shared_ptr` 实例时都不会抛出异常。同样，在调用`data.pop()`⑥（这个成员函数保证不会抛出异常）之前，依旧没有对数据结构进行修改，所以这个函数也为“异常-安全”。

最后，`empty()`也不会修改任何数据，所以也是“异常-安全”函数。

当调用持有一个锁的用户代码时，这里有两个地方可能会产生死锁：进行拷贝构造或移动构造(①，③)和在对数据项进行拷贝赋值或移动赋值操作⑤的时候；还有一个潜在死锁的地方在于用户定义的操作符`new`。当这些函数，无论是以直接调用栈的成员函数的方式，还是在成员函数进行操作时，对已经插入或删除的数据进行操作的方式，对锁进行获取，都可能造成死锁。不过，用户要对栈负责，当栈未对一个数据进行拷贝或分配时，用户就不能想当然的将其添加到栈中。

所有成员函数都使用 `std::lock_guard<>` 来保护数据，所以栈的成员函数能有“线程安全”的表现。当然，构造与析构函数不是“线程安全”的，不过这也不成问题，因为对实例的构造与析构只能有一次。调用一个不完全构造对象或是已销毁对象的成员函数，无论在那种编程方式下，都不可取。所以，用户就要保证在栈对象完成构建前，其他线程无法对其进行访问；并且，一定要保证在栈对象销毁后，所有线程都要停止对其进行访问。

即使在多线程情况下，并发的调用成员函数是安全的(因为使用锁)，也要保证在单线程的情况下，数据结构做出正确反应。序列化线程会隐性的限制程序性能，这就是栈争议声最大的地方：当一

一个线程在等待锁时，它就会无所事事。同样的，对于栈来说，等待添加元素也是没有意义的，所以当需要一个线程需要等待时，其会定期检查`empty()`或`pop()`，以及对`empty_stack`异常进行关注。这样的现实会限制栈的实现的方式，在线程等待的时候，会浪费宝贵的资源去检查数据，或是要求用户编写外部等待和提示代码(例如，使用条件变量)，这就使内部锁失去存在的意义——这意味着资源的浪费。第4章中的队列，就是一种使用条件内部变量进行等待的数据结构，接下来我们就了解一下。

6.2.2 线程安全队列——使用锁和条件变量

第4章中的线程安全队列，在清单6.2中重现一下。和使用仿 `std::stack<>` 建立的栈很像，这里队列的建立也是参照了 `std::queue<>`。不过，与标准容器的接口不同，我们要设计的是能在多线程下安全并发访问的数据结构。

清单6.2 使用条件变量实现的线程安全队列

```
1  template<typename T>
2  class threadsafe_queue
3  {
4  private:
5      mutable std::mutex mut;
6      std::queue<T> data_queue;
7      std::condition_variable data_cond;
8
9  public:
10     threadsafe_queue()
11     {}
12
13     void push(T new_value)
14     {
15         std::lock_guard<std::mutex> lk(mut);
16         data_queue.push(std::move(data));
17         data_cond.notify_one(); // 1
18     }
19
20     void wait_and_pop(T& value) // 2
21     {
22         std::unique_lock<std::mutex> lk(mut);
23         data_cond.wait(lk, [this]{return !data_queue.empty();});
24         value=std::move(data_queue.front());
25         data_queue.pop();
```

```

26     }
27
28     std::shared_ptr<T> wait_and_pop()    // 3
29     {
30         std::unique_lock<std::mutex> lk(mut);
31         data_cond.wait(lk,[this]{return !data_queue.empty();});    // 4
32         std::shared_ptr<T> res(
33             std::make_shared<T>(std::move(data_queue.front())));
34         data_queue.pop();
35         return res;
36     }
37
38     bool try_pop(T& value)
39     {
40         std::lock_guard<std::mutex> lk(mut);
41         if(data_queue.empty())
42             return false;
43         value=std::move(data_queue.front());
44         data_queue.pop();
45         return true;
46     }
47
48     std::shared_ptr<T> try_pop()
49     {
50         std::lock_guard<std::mutex> lk(mut);
51         if(data_queue.empty())
52             return std::shared_ptr<T>();    // 5
53         std::shared_ptr<T> res(
54             std::make_shared<T>(std::move(data_queue.front())));
55         data_queue.pop();
56         return res;
57     }
58
59     bool empty() const
60     {
61         std::lock_guard<std::mutex> lk(mut);
62         return data_queue.empty();
63     }
64 };

```

除了在push()①中调用data_cond.notify_one(), 以及wait_and_pop()②③, 6.2中对队列的实现与6.1中对栈的实现十分相近。两个重载try_pop()除了在队列为空时抛出异常, 其他的与6.1中pop()函数完全一样。不同的是, 在6.1中对值的检索会返回一个bool值, 而在6.2中, 当指针指向空值的

时候会返回NULL指针⑤，这同样也是实现栈的一个有效途径。所以，即使排除掉wait_and_pop()函数，之前对栈的分析依旧适用于这里。

wait_and_pop()函数是等待队列向栈进行输入的一个解决方案；比起持续调用empty()，等待线程调用wait_and_pop()函数和数据结构处理等待中的条件变量的方式要好很多。对于data_cond.wait()的调用，直到队列中有一个元素的时候，才会返回，所以你就不用担心会出现一个空队列的情况了，还有，数据会一直被互斥锁保护。因为不变量这里并未发生变化，所以函数不会添加新的条件竞争或是死锁的可能。

异常安全在这里的会有一些变化，当不止一个线程等待对队列进行推送操作是，只会有一个线程，因得到data_cond.notify_one()，而继续工作着。但是，如果这个工作线程在wait_and_pop()中抛出一个异常，例如：构造新的 std::shared_ptr<> 对象④时抛出异常，那么其他线程则会永世长眠。当这种情况是不可接受时，这里的调用就需要改成data_cond.notify_all()，这个函数将唤醒所有的工作线程，不过，当大多线程发现队列依旧是空时，又会耗费很多资源让线程重新进入睡眠状态。第二种替代方案是，当有异常抛出的时候，让wait_and_pop()函数调用notify_one()，从而让另一个线程可以去尝试索引存储的值。第三种替代方案就是，将 std::shared_ptr<> 的初始化过程移到push()中，并且存储 std::shared_ptr<> 实例，而非直接使用数据的值。将 std::shared_ptr<> 拷贝到内部 std::queue<> 中，就不会抛出异常了，这样wait_and_pop()又是安全的了。下面的程序清单，就是根据第三种方案进行修改的。

清单6.3 持有 std::shared_ptr<> 实例的线程安全队列

```

1  template<typename T>
2  class threadsafe_queue
3  {
4  private:
5      mutable std::mutex mut;
6      std::queue<std::shared_ptr<T> > data_queue;
7      std::condition_variable data_cond;
8  public:
9      threadsafe_queue()
10     {}
11
12     void wait_and_pop(T& value)
13     {
14         std::unique_lock<std::mutex> lk(mut);
15         data_cond.wait(lk,[this]{return !data_queue.empty();});
16         value=std::move(*data_queue.front()); // 1
17         data_queue.pop();
18     }
19
20     bool try_pop(T& value)

```

```
21 {
22     std::lock_guard<std::mutex> lk(mut);
23     if(data_queue.empty())
24         return false;
25     value=std::move(*data_queue.front()); // 2
26     data_queue.pop();
27     return true;
28 }
29
30 std::shared_ptr<T> wait_and_pop()
31 {
32     std::unique_lock<std::mutex> lk(mut);
33     data_cond.wait(lk,[this]{return !data_queue.empty();});
34     std::shared_ptr<T> res=data_queue.front(); // 3
35     data_queue.pop();
36     return res;
37 }
38
39 std::shared_ptr<T> try_pop()
40 {
41     std::lock_guard<std::mutex> lk(mut);
42     if(data_queue.empty())
43         return std::shared_ptr<T>();
44     std::shared_ptr<T> res=data_queue.front(); // 4
45     data_queue.pop();
46     return res;
47 }
48
49 void push(T new_value)
50 {
51     std::shared_ptr<T> data(
52         std::make_shared<T>(std::move(new_value))); // 5
53     std::lock_guard<std::mutex> lk(mut);
54     data_queue.push(data);
55     data_cond.notify_one();
56 }
57
58 bool empty() const
59 {
60     std::lock_guard<std::mutex> lk(mut);
61     return data_queue.empty();
62 }
63 };
```


为让 `std::shared_ptr<>` 持有数据的结果显而易见：弹出函数会持有一个变量的引用，为了接收这个新值，必须对存储的指针进行解引用①，②；并且，在返回到调用函数前，弹出函数都会返回一个 `std::shared_ptr<>` 实例，这里实例可以在队列中做检索③，④。

`std::shared_ptr<>` 持有数据的好处：新的实例分配结束时，不会被锁在`push()`⑤当中(而在清单6.2中，只能在`pop()`持有锁时完成)。因为内存分配操作的需要在性能上付出很高的代价(性能较低)，所以使用 `std::shared_ptr<>` 的方式对队列的性能有很大的提升，其减少了互斥量持有的时间，允许其他线程在分配内存的同时，对队列进行其他的操作。

如同栈的例子，使用互斥量保护整个数据结构，不过会限制队列对并发的支持；虽然，多线程可能被队列中的各种成员函数所阻塞，但是仍有一个线程能在任意时间内进行工作。不过，这种限制的部分来源是因为在实现中使用了 `std::queue<>`；因为使用标准容器的原因，数据处于保护中。要对数据结构实现进行具体的控制，需要提供更多细粒度锁，来完成更高级的并发。

6.2.3 线程安全队列——使用细粒度锁和条件变量

在清单6.2和6.3中，使用一个互斥量对一个数据队列(`data_queue`)进行保护。为了使用细粒度锁，需要看一下队列内部的组成结构，并且将一个互斥量与每个数据相关联。

对于队列来说，最简单的数据结构就是单链表了，就如图6.1那样。队列里包含一个头指针，其指向链表中的第一个元素，并且每一个元素都会指向下一个元素。从队列中删除数据，其实就是将头指针指向下一个元素，并将之前头指针指向的值进行返回。

向队列中添加元素是要从结尾进行的。为了做到这点，队列里还有一个尾指针，其指向链表中的最后一个元素。新节点的加入将会改变尾指针的`next`指针，之前最后一个元素将会指向新添加进来的元素，新添加进来的元素的`next`将会使新的尾指针。当链表为空时，头/尾指针皆为`NULL`。

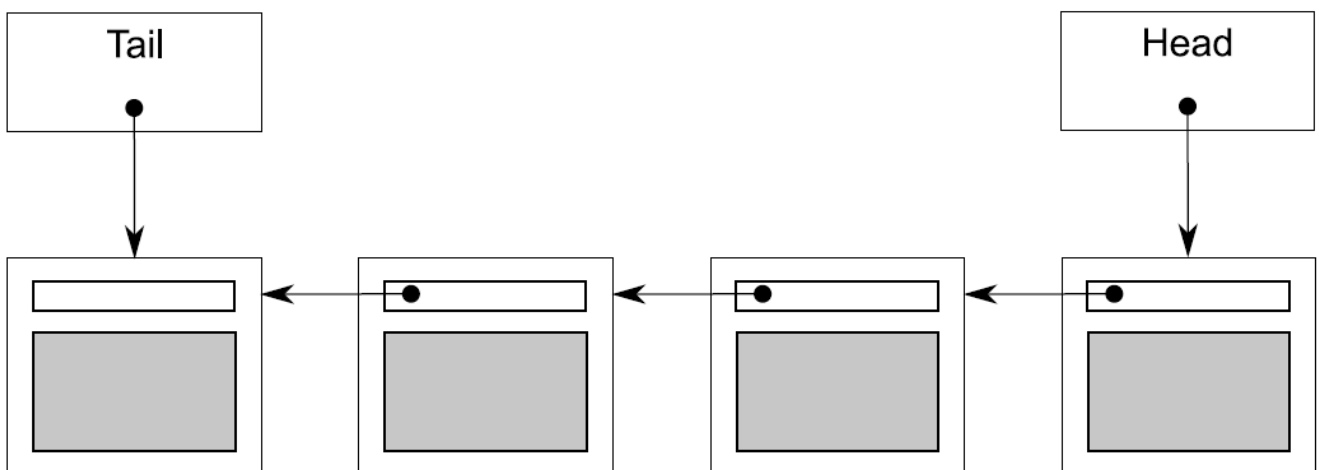


图6.1 用单链表表示的队列

下面的清单中的代码，是一个简单队列的实现，基于清单6.2代码的精简版本；因为这个队列仅供单线程使用，所以这实现中只有一个`try_pop()`函数；并且，没有`wait_and_pop()`函数。

清单6.4 队列实现——单线程版

```
1  template<typename T>
2  class queue
3  {
4  private:
5      struct node
6      {
7          T data;
8          std::unique_ptr<node> next;
9
10         node(T data_):
11             data(std::move(data_))
12         {}
13     };
14
15     std::unique_ptr<node> head; // 1
16     node* tail; // 2
17
18 public:
19     queue()
20     {}
21     queue(const queue& other)=delete;
22     queue& operator=(const queue& other)=delete;
23     std::shared_ptr<T> try_pop()
24     {
25         if(!head)
26         {
27             return std::shared_ptr<T>();
28         }
29         std::shared_ptr<T> const res(
30             std::make_shared<T>(std::move(head->data)));
31         std::unique_ptr<node> const old_head=std::move(head);
32         head=std::move(old_head->next); // 3
33         return res;
34     }
35
36     void push(T new_value)
37     {
```

```

38     std::unique_ptr<node> p(new node(std::move(new_value)));
39     node* const new_tail=p.get();
40     if(tail)
41     {
42         tail->next=std::move(p); // 4
43     }
44     else
45     {
46         head=std::move(p); // 5
47     }
48     tail=new_tail; // 6
49 }
50 };

```

首先，注意在清单6.4中使用了 `std::unique_ptr<node>` 来管理节点，因为其能保证节点(其引用数据的值)在删除时候，不需要使用`delete`操作显式删除。这样的关系链表，管理着从头结点到尾节点的每一个原始指针。

虽然，这种实现对于单线程来说没什么问题，但是，当你在多线程情况下，尝试使用细粒度锁时，就会出现问题。因为在给定的实现中有两个数据项(**head**①和**tail**②)；即使，使用两个互斥量，来保护头指针和尾指针，也会出现问题。

显而易见的问题就是`push()`可以同时修改头指针⑤和尾指针⑥，所以`push()`函数会同时获取两个互斥量。虽然会将两个互斥量都上锁，但这还不是太糟糕的问题。糟糕的问题是`push()`和`pop()`都能访问`next`指针指向的节点：`push()`可更新`tail->next`④，而后`try_pop()`读取`read->next`③。当队列中只有一个元素时，`head==tail`，所以`head->next`和`tail->next`是同一个对象，并且这个对象需要保护。不过，“在同一个对象在未被`head`和`tail`同时访问时，`push()`和`try_pop()`锁住的是同一个锁”，就不对了。所以，你就没有比之间实现更好的选择了。这里会“柳暗花明又一村”吗？

通过分离数据实现并发

你可以使用“预分配一个虚拟节点(无数据)，确保这个节点永远在队列的最后，用来分离头尾指针能访问的节点”的办法，走出这个困境。对于一个空队列来说，`head`和`tail`都属于虚拟指针，而非空指针。这个办法挺好，因为当队列为空时，`try_pop()`不能访问`head->next`了。当添加一个节点入队列时(这时有真实节点了)，`head`和`tail`现在指向不同的节点，所以就不会在`head->next`和`tail->next`上产生竞争。这里的缺点是，你必须额外添加一个间接层次的指针数据，来做虚拟节点。下面的代码描述了这个方案如何实现。

清单6.5 带有虚拟节点的队列

```

1  template<typename T>

```

```
2  class queue
3  {
4  private:
5      struct node
6      {
7          std::shared_ptr<T> data; // 1
8          std::unique_ptr<node> next;
9      };
10
11     std::unique_ptr<node> head;
12     node* tail;
13
14 public:
15     queue():
16         head(new node),tail(head.get()) // 2
17     {}
18     queue(const queue& other)=delete;
19     queue& operator=(const queue& other)=delete;
20
21     std::shared_ptr<T> try_pop()
22     {
23         if(head.get()==tail) // 3
24         {
25             return std::shared_ptr<T>();
26         }
27         std::shared_ptr<T> const res(head->data); // 4
28         std::unique_ptr<node> old_head=std::move(head);
29         head=std::move(old_head->next); // 5
30         return res; // 6
31     }
32
33     void push(T new_value)
34     {
35         std::shared_ptr<T> new_data(
36             std::make_shared<T>(std::move(new_value))); // 7
37         std::unique_ptr<node> p(new node); //8
38         tail->data=new_data; // 9
39         node* const new_tail=p.get();
40         tail->next=std::move(p);
41         tail=new_tail;
42     }
43 };
```

`try_pop()`不需要太多的修改。首先，你可以拿`head`和`tail`③进行比较，这就要比检查指针是否为空的好，因为虚拟节点意味着`head`不可能是空指针。`head`是一个 `std::unique_ptr<node>` 对象，你需要使用`head.get()`来做比较。其次，因为`node`现在存在数据指针中①，你就可以对指针进行直接检索④，而非构造一个T类型的新实例。`push()`函数改动最大：首先，你必须在堆上创建一个T类型的实例，并且让其与一个 `std::shared_ptr<>` 对象相关联⑦(节点使用 `std::make_shared` 就是为了避免内存二次分配，避免增加引用次数)。创建的新节点就成为了虚拟节点，所以你不需要为`new_value`提供构造函数⑧。反而这里你需要将`new_value`的副本赋给之前的虚拟节点⑨。最终，为了让虚拟节点存在在队列中，你不得不使用构造函数来创建它②。

那么现在，我确信你会对如何对如何修改队列，让其变成一个线程安全的队列感到惊讶。好吧，现在的`push()`只能访问`tail`，而不能访问`head`，这就是一个进步`try_pop()`可以访问`head`和`tail`，但是`tail`只需在最初进行比较，所以所存在的时间很短。重大的提升在于，虚拟节点意味着`try_pop()`和`push()`不能对同一节点进行操作，所以这里已经不再需要互斥了。那么，你只需要使用一个互斥量来保护`head`和`tail`就够了。那么，现在应该锁哪里？

我们的目的是为了最大程度的并发化，所以你需要上锁的时间，要尽可能的小。`push()`很简单：互斥量需要对`tail`的访问进行上锁，这就意味着你需要对每一个新分配的节点进行上锁⑧，还有在你当前尾节点进行赋值的时候⑨也需要上锁。锁需要持续到函数结束时才能解开。

`try_pop()`就不简单了。首先，你需要使用互斥量锁住`head`，一直到`head`弹出。实际上，互斥量决定了哪一个线程来进行弹出操作。一旦`head`被改变⑤，你才能解锁互斥量；当在返回结果时，互斥量就不需要进行上锁了⑥。这使得访问`tail`需要一个尾互斥量。因为，你需要只需要访问`tail`一次，且只有在访问时才需要互斥量。这个操作最好是通过函数进行包装。事实上，因为代码只有在成员需要`head`时，互斥量才上锁，这项也需要包含在包装函数中。最终代码如下所示。

清单6.6 线程安全队列——细粒度锁版

```
1  template<typename T>
2  class threadsafe_queue
3  {
4  private:
5      struct node
6      {
7          std::shared_ptr<T> data;
8          std::unique_ptr<node> next;
9      };
10     std::mutex head_mutex;
11     std::unique_ptr<node> head;
12     std::mutex tail_mutex;
13     node* tail;
14 }
```

```
15     node* get_tail()
16     {
17         std::lock_guard<std::mutex> tail_lock(tail_mutex);
18         return tail;
19     }
20
21     std::unique_ptr<node> pop_head()
22     {
23         std::lock_guard<std::mutex> head_lock(head_mutex);
24         if(head.get()==get_tail())
25         {
26             return nullptr;
27         }
28         std::unique_ptr<node> old_head=std::move(head);
29         head=std::move(old_head->next);
30         return old_head;
31     }
32 public:
33     threadsafe_queue():
34     head(new node),tail(head.get())
35     {}
36     threadsafe_queue(const threadsafe_queue& other)=delete;
37     threadsafe_queue& operator=(const threadsafe_queue& other)=delete;
38
39     std::shared_ptr<T> try_pop()
40     {
41         std::unique_ptr<node> old_head=pop_head();
42         return old_head?old_head->data:std::shared_ptr<T>();
43     }
44
45     void push(T new_value)
46     {
47         std::shared_ptr<T> new_data(
48             std::make_shared<T>(std::move(new_value)));
49         std::unique_ptr<node> p(new node);
50         node* const new_tail=p.get();
51         std::lock_guard<std::mutex> tail_lock(tail_mutex);
52         tail->data=new_data;
53         tail->next=std::move(p);
54         tail=new_tail;
55     }
56 };
```

让我们用挑剔的目光来看一下上面的代码，并考虑6.1.1节中给出的指导意见。在你观察不变量前，你需要确定的状态有：

- `tail->next == nullptr`
- `tail->data == nullptr`
- `head == tail`(意味着空列表)
- 单元素列表 `head->next = tail`
- 在列表中的每一个节点`x`，`x!=tail`且`x->data`指向一个T类型的实例，并且`x->next`指向列表中下一个节点。`x->next == tail`意味着`x`就是列表中最后一个节点
- 顺着`head`的`next`节点找下去，最终会找到`tail`

这里的`push()`很简单：仅修改了被`tail_mutex`的数据，因为新的尾节点是一个空节点，并且其`data`和`next`都为旧的尾节点(实际上的尾节点)设置好，所以其能维持不变量的状态。

有趣的部分在于`try_pop()`上。事实证明，不仅需要对`tail_mutex`上锁，来保护对`tail`的读取；还要保证在从头读取数据时，不会产生数据竞争。如果没有这些互斥量，当一个线程调用`try_pop()`的同时，另一个线程调用`push()`，那么这里操作顺序将不可预测。尽管，每一个成员函数都持有一个互斥量，这些互斥量能保护数据不会同时被多个线程访问到；并且，队列中的所有数据来源，都是通过调用`push()`得到的。因为线程可能会无序的方位同一数据，所以这里就会有数据竞争(正如你在第5章看到的那样)，以及未定义行为。幸运的是，在`get_tail()`中的`tail_mutex`解决了所有的问题。因为调用`get_tail()`将会锁住同名锁，就像`push()`一样，这就为两个操作规定好了顺序。要不就是`get_tail()`在`push()`之前被调用，这种情况下，线程可以看到旧的尾节点，要不就是在`push()`之后完成，这种情况下，线程就能看到`tail`的新值，以及新数据前的真正`tail`的值。

当`get_tail()`调用前，`head_mutex`已经上锁，这一步也是很重要的哦。如果不这样，调用`pop_head()`时就会被`get_tail()`和`head_mutex`所卡住，因为其他线程调用`try_pop()`(以及`pop_head()`)时，都需要先获取锁，然后阻止从下面的过程中初始化线程：

```

1  std::unique_ptr<node> pop_head() // 这是个有缺陷的实现
2  {
3      node* const old_tail=get_tail(); // ① 在head_mutex范围外获取旧尾节点的值
4      std::lock_guard<std::mutex> head_lock(head_mutex);
5
6      if(head.get()==old_tail) // ②
7      {
8          return nullptr;
9      }
10     std::unique_ptr<node> old_head=std::move(head);
11     head=std::move(old_head->next); // ③
12     return old_head;
13 }
```


这是一个有缺陷的实现，调用`get_tail()`是在锁的范围之外，你可能也许会发现`head`和`tail`，在你初始化线程，并获取`head_mutex`时，发生了改变。并且，不只是返回尾节点时，返回的不是尾节点了，其值甚至都不列表中的值了。即使`head`是最后一个节点，这也意味着`head`和`old_tail`②比较失败。因此，当你更新`head`③时，可能会将`head`移到`tail`之后，这样的话就意味着数据结构遭到了破坏。在正确实现中(清单6.6)，需要保证在`head_mutex`保护的范围内调用`get_tail()`。这就能保证没有其他线程能对`head`进行修改，并且`tail`会向正确的方向移动(当有新节点添加进来时)，这样就很安全了。`head`不会传递给`get_tail()`的返回值，所以不变量的状态时稳定的。

当使用`pop_head()`更新`head`时(从队列中删除节点)，互斥量就已经上锁了，并且`try_pop()`可以提取数据，并在确实有个数据的时候删除一个节点(若没有数据，则返回 `std::shared_ptr<>` 的空实例)，因为只有一个线程可以访问这个节点，所以根据我们所掌握的知识，认为这个操作是安全的。

接下来，外部接口就相当于清单6.2代码中的子集了，所以同样的分析结果：对于固有接口来说，不存在条件竞争。

异常是很有趣的东西。虽然，你已经改变了数据的分配模式，但是异常可能从别的地方袭来。`try_pop()`中的对锁的操作会产生异常，并直到锁获取才能对数据进行修改。因此，`try_pop()`是异常安全的。另一方面，`push()`可以在堆上新分配出一个T的实例，以及一个`node`的新实例，这里可能会抛出异常。但是，所有分配的对象都赋给了智能指针，那么当异常发生时，他们就会被释放掉。一旦锁被获取，`push()`中的操作就不会抛出异常，所以`push()`也是异常安全的。

因为没有修改任何接口，所以不会死锁。在实现内部也不会有死锁；唯一需要获取两个锁的是`pop_head()`，这个函数需要获取`head_mutex`和`tail_mutex`，所以不会产生死锁。

那么剩下的问题就都在实际并发的可行性上了。这个结构对并发访问的考虑要多于清单6.2中的代码，因为这里锁粒度更加的小，并且更多的数据不在锁的保护范围内。比如，在`push()`中，新节点和新数据的分配都不需要锁来保护。这就意味着多线程情况下，节点及数据的分配是“安全”并发的。同一时间内，只有一个线程可以将它的节点和数据添加到队列中，所以代码中只是简单使用了指针赋值的形式，相较于基于 `std::queue<>` 的实现中，对于 `std::queue<>` 的内部操作进行上锁，这个结构中就不需要了。

同样，`try_pop()`持有`tail_mutex`也只有很短的时间，只为保护对`tail`的读取。因此，当有数据`push`进队列后，`try_pop()`几乎及可以完全并发调用了。同样在执行中，对`head_mutex`的持有时间也是极短的。当并发访问时，这就会增加对`try_pop()`的访问次数；且只有一个线程，在同一时间内可以访问`pop_head()`，且多线程情况下可以删除队列中的旧节点，并且安全的返回数据。

等待数据弹出

OK, 所以清单6.6提供了一个使用细粒度锁的线程安全队列, 不过只有`try_pop()`可以并发访问(且只有一个重载存在)。那么在清单6.2中方便的`wait_and_pop()`呢? 你能通过细粒度锁实现一个相同功能的接口吗?

当然, 答案是“是的”, 不过的确有些困难, 困难在哪里? 修改`push()`是相对简单的: 只需要在函数体末尾添加`data_cond.notify_one()`函数的调用即可(如同清单6.2中那样)。当然, 事实并没有那么简单: 你使用细粒度锁, 是为了保证最大程度的并发。当将互斥量和`notify_one()`混用的时, 如果被通知的线程在互斥量解锁后被唤醒, 那么这个线程就不得不等待互斥量上锁。另一方面, 当解锁操作在`notify_one()`之前调用, 那么互斥量可能会等待线程醒来, 来获取互斥锁(假设没有其他线程对互斥量上锁)。这可能是一个微小的改动, 但是对于一些情况来说, 就显得很重要了。

`wait_and_pop()`就有些复杂了, 因为需要确定在哪里等待, 也就是函数在哪里执行, 并且需要确定哪些互斥量需要上锁。等待的条件是“队列非空”, 这就意味着`head!=tail`。这样写的话, 就需要同时获取`head_mutex`和`tail_mutex`, 并对其进行上锁, 不过在清单6.6中已经使用`tail_mutex`来保护对`tail`的读取, 以及不用和自身记忆性比较, 所以这种逻辑也同样适用于这里。如果有函数让`head!=get_tail()`, 你只需要持有`head_mutex`, 然后你就可以使用锁, 对`data_cond.wait()`的调用进行保护。当你将等待逻辑添加入结构当中, 那么实现的方式与`try_pop()`基本上是一样的。

对于`try_pop()`和`wait_and_pop()`的重载都需要深思熟虑。当你将返回 `std::shared_ptr<>` 替换为从“`old_head`后索引出的值, 并且拷贝赋值给`value`参数”进行返回时, 那么这里将会存在异常安全问题。数据项在互斥锁未上锁的情况下被删除, 将剩下的数据返回给调用者。不过, 当拷贝赋值抛出异常(可能性很大)时, 数据项将会丢失, 因为它没有被返回队列原来的位置上。

当T类型有无异常抛出的移动赋值操作, 或无异常抛出的交换操作时, 你可以使用它, 不过, 你肯定更喜欢一种通用的解决方案, 无论T是什么类型, 这个方案都能使用。在这种情况下, 在节点从列表中删除前, 你就不得不将有可能抛出异常的代码, 放在锁保护的范围内, 来保证异常安全性。这也就意味着你需要对`pop_head()`进行重载, 查找索引值在列表改动前的位置。

相比之下, `empty()`就更加的简单: 只需要锁住`head_mutex`, 并且检查`head==get_tail()`(详见清单6.10)就可以了。最终的代码, 在清单6.7, 6.8, 6.9和6.10中。

清单6.7 可上锁和等待的线程安全队列——内部机构及接口

```
1  template<typename T>
2  class threadsafe_queue
3  {
4  private:
5      struct node
6      {
7          std::shared_ptr<T> data;
8          std::unique_ptr<node> next;
```

```

9     };
10
11     std::mutex head_mutex;
12     std::unique_ptr<node> head;
13     std::mutex tail_mutex;
14     node* tail;
15     std::condition_variable data_cond;
16 public:
17     threadsafe_queue():
18         head(new node), tail(head.get())
19     {}
20     threadsafe_queue(const threadsafe_queue& other)=delete;
21     threadsafe_queue& operator=(const threadsafe_queue& other)=delete;
22
23     std::shared_ptr<T> try_pop();
24     bool try_pop(T& value);
25     std::shared_ptr<T> wait_and_pop();
26     void wait_and_pop(T& value);
27     void push(T new_value);
28     bool empty();
29 };

```

向队列中添加新节点是相当简单的——下面的实现与上面的代码差不多。

清单6.8 可上锁和等待的线程安全队列——推入新节点

```

1  template<typename T>
2  void threadsafe_queue<T>::push(T new_value)
3  {
4      std::shared_ptr<T> new_data(
5          std::make_shared<T>(std::move(new_value)));
6      std::unique_ptr<node> p(new node);
7      {
8          std::lock_guard<std::mutex> tail_lock(tail_mutex);
9          tail->data=new_data;
10         node* const new_tail=p.get();
11         tail->next=std::move(p);
12         tail=new_tail;
13     }
14     data_cond.notify_one();
15 }

```

如同之前所提到的，复杂部分都在`pop`那边，所以提供帮助性函数去简化这部分就很重要了。下一个清单中将展示`wait_and_pop()`的实现，以及先关的帮助函数。

清单6.9 可上锁和等待的线程安全队列——`wait_and_pop()`

```
1  template<typename T>
2  class threadsafe_queue
3  {
4  private:
5      node* get_tail()
6      {
7          std::lock_guard<std::mutex> tail_lock(tail_mutex);
8          return tail;
9      }
10
11     std::unique_ptr<node> pop_head() // 1
12     {
13         std::unique_ptr<node> old_head=std::move(head);
14         head=std::move(old_head->next);
15         return old_head;
16     }
17
18     std::unique_lock<std::mutex> wait_for_data() // 2
19     {
20         std::unique_lock<std::mutex> head_lock(head_mutex);
21         data_cond.wait(head_lock, [&]{return head.get() != get_tail();});
22         return std::move(head_lock); // 3
23     }
24
25     std::unique_ptr<node> wait_pop_head()
26     {
27         std::unique_lock<std::mutex> head_lock(wait_for_data()); // 4
28         return pop_head();
29     }
30
31     std::unique_ptr<node> wait_pop_head(T& value)
32     {
33         std::unique_lock<std::mutex> head_lock(wait_for_data()); // 5
34         value=std::move(*head->data);
35         return pop_head();
36     }
37 public:
38     std::shared_ptr<T> wait_and_pop()
39     {
```

```
40     std::unique_ptr<node> const old_head=wait_pop_head();
41     return old_head->data;
42 }
43
44 void wait_and_pop(T& value)
45 {
46     std::unique_ptr<node> const old_head=wait_pop_head(value);
47 }
48 };
```

清单6.9中所示的pop部分的实现中有一些帮助函数来降低代码的复杂度，例如pop_head()①和wait_for_data()②，这些函数分别是删除头结点和等待队列中有数据弹出的。wait_for_data()特别值得关注，因为其不仅等待使用lambda函数对条件变量进行等待，而且它还会将锁的实例返回给调用者③。这就需要确保同一个锁在执行与wait_pop_head()重载④⑤的相关操作时，已持有锁。pop_head()是对try_pop()代码的复用，将在下面进行展示：

清单6.10 可上锁和等待的线程安全队列——try_pop()和empty()

```
1  template<typename T>
2  class threadsafe_queue
3  {
4  private:
5      std::unique_ptr<node> try_pop_head()
6      {
7          std::lock_guard<std::mutex> head_lock(head_mutex);
8          if(head.get()==get_tail())
9          {
10             return std::unique_ptr<node>();
11          }
12          return pop_head();
13      }
14
15      std::unique_ptr<node> try_pop_head(T& value)
16      {
17          std::lock_guard<std::mutex> head_lock(head_mutex);
18          if(head.get()==get_tail())
19          {
20             return std::unique_ptr<node>();
21          }
22          value=std::move(*head->data);
23          return pop_head();
24      }
25  public:
```

```
26     std::shared_ptr<T> try_pop()
27     {
28         std::unique_ptr<node> old_head=try_pop_head();
29         return old_head?old_head->data:std::shared_ptr<T>();
30     }
31
32     bool try_pop(T& value)
33     {
34         std::unique_ptr<node> const old_head=try_pop_head(value);
35         return old_head;
36     }
37
38     bool empty()
39     {
40         std::lock_guard<std::mutex> head_lock(head_mutex);
41         return (head.get()==get_tail());
42     }
43 };
```

这个队列的实现将作为第7章无锁队列的基础。这是一个无界队列;线程可以持续向队列中添加数据项,即使没有元素被删除。与之相反的就是有界队列,在有界队列中,队列在创建的时候最大长度就已经是固定的了。当有界队列满载时,尝试在向其添加元素的操作将会失败或者阻塞,直到有元素从队列中弹出。在任务执行时(详见第8章),有界队列对于线程间的工作花费是很有帮助的。其会阻止线程对队列进行填充,并且可以避免线程从较远的地方对数据项进行索引。

无界队列的实现,很容易扩展成,可在`push()`中等待跳进变量的定长队列。相对于等待队列中具有数据项(`pop()`执行完成后),你就需要等待队列中数据项小于最大值就可以了。对于有界队列更多的讨论,已经超出了本书的范围,就不再多说;现在越过队列,向更加复杂的数据结构进发。