6.3 基于锁设计更加复杂的数据结构

栈和队列都很简单：接口相对固定，并且它们应用于比较特殊的情况。并不是所有数据结构都像它们一样简单；大多数数据结构支持更加多样化的操作。原则上，这将增大并行的可能性，但是也让对数据保护变得更加困难，因为要考虑对所有能访问到的部分。当为了并发访问对数据结构进行设计时，这一系列原有的操作，就变得越发重要，需要重点处理。

先来看看，在查询表的设计中，所遇到的一些问题。

6.3.1 编写一个使用锁的线程安全查询表

查询表或字典是一种类型的值(键值)和另一种类型的值进行关联(映射的方式)。一般情况下，这样的结构允许代码通过键值对相关的数据值进行查询。在 `C++` 标准库中，这种相关工具有：`std::map<>` , `std::multimap<>` , `std::unordered_map<>` 以及 `std::unordered_multimap<>` 。

查询表的使用与栈和队列不同。栈和队列上，几乎每个操作都会对数据结构进行修改，不是添加一个元素，就是删除一个，而对于查询表来说，几乎不需要什么修改。清单3.13中有个例子，是一个简单的域名系统(DNS)缓存，其特点是，相较于 `std::map<>` 削减了很多的接口。和队列和栈一样，标准容器的接口不适合多线程进行并发访问，因为这些接口在设计的时候都存在固有的条件竞争，所以这些接口需要砍掉，以及重新修订。

并发访问时， `std::map<>` 接口最大的问题在于——迭代器。虽然，在多线程访问(或修改)容器时，可能会有提供安全访问的迭代器，但这就问题棘手之处。要想正确的处理迭代器，你可能会碰到下面这个问题：当迭代器引用的元素被其他线程删除时，迭代器在这里就是个问题了。线程安全的查询表，第一次接口削减，需要绕过迭代器。 `std::map<>` (以及标准库中其他相关容器)给定的接口对于迭代器的依赖是很严重的，其中有些接口需要先放在一边，先对一些简单接口进行设计。

查询表的基本操作有：

- 添加一对“键值-数据”
- 修改指定键值所对应的数据

- 删除一组值
- 通过给定键值，获取对应数据

容器也有一些操作是非常有用的，比如：查询容器是否为空，键值列表的完整快照和“键值-数据”的完整快照。

如果你坚持之前的线程安全指导意见，例如：不要返回一个引用，并且用一个简单的互斥锁对每一个成员函数进行上锁，以确保每一个函数线程安全。最有可能的条件竞争在于，当一对“键值-数据”加入时；当两个线程都添加一个数据，那么肯定一个先一个后。一种方式是合并“添加”和“修改”操作，为一个成员函数，就像清单3.13对域名系统缓存所做的那样。

从接口角度看，有一个问题很是有趣，那就是*任意*(if any)部分获取相关数据。一种选择是允许用户提供一个“默认”值，在键值没有对应值的时候进行返回：

```
mapped_type get_value(key_type const& key, mapped_type default_value);
```

在种情况下，当default_value没有明确的给出时，默认构造出的mapped_type实例将被使用。也可以扩展成返回一个 `std::pair<mapped_type, bool>` 来代替mapped_type实例，其中bool代表返回值是否是当前键对应的值。另一个选择是，返回一个有指向数据的智能指针；当指针的值是NULL时，那么这个键值就没有对应的数据。

如我们之前所提到的，当接口确定时，那么(假设没有接口间的条件竞争)就需要保证线程安全了，可以通过对每一个成员函数使用一个互斥量和一个简单的锁，来保护底层数据。不过，当独立的函数对数据结构进行读取和修改时，就会降低并发的可能性。一个选择是使用一个互斥量去面对多个读者线程，或一个作者线程，如同在清单3.13中对 `boost::shared_mutex` 的使用一样。虽然，这将提高并发访问的可能性，但是在同一时间内，也只有一个线程能对数据结构进行修改。理想很美好，现实很骨感？我们应该能做的更好！

为细粒度锁设计一个映射结构

在对队列的讨论中(在6.2.3节)，为了允许细粒度锁能正常工作，需要对于数据结构的细节进行仔细的考虑，而非直接使用已存在的容器，例如 `std::map<>` 。这里列出三个常见关联容器的方式：

- 二叉树，比如：红黑树
- 有序数组
- 哈希表

二叉树的方式，不会对提高并发访问的概率；每一个查找或者修改操作都需要访问根节点，因此，根节点需要上锁。虽然，访问线程在向下移动时，这个锁可以进行释放，但相比横跨整个数

据结构的单锁，并没有什么优势。

有序数组是最坏的选择，因为你无法提前言明数组中哪段是有序的，所以你需要用一个锁将整个数组锁起来。

那么就剩哈希表了。假设有固定数量的桶，每个桶都有一个键值(关键特性)，以及散列函数。这就意味着你可以安全的对每个桶上锁。当你再次使用互斥量(支持多读者单作者)时，你就能将并发访问的可能性增加N倍，这里N是桶的数量。当然，缺点也是有的：对于键值的操作，需要有合适的函数。C++标准库提供 `std::hash<>` 模板，可以直接使用。对于特化的类型，比如int，以及通用库类型 `std::string` ，并且用户可以简单的对键值类型进行特化。如果你去效仿标准无序容器，并且获取函数对象的类型作为哈希表的模板参数，用户可以选择是否特化 `std::hash<>` 的键值类型，或者提供一个独立的哈希函数。

那么，让我们来看一些代码吧。怎样的实现才能完成一个线程安全的查询表？下面就是一种方式。

清单6.11 线程安全的查询表

```
template<typename Key,typename Value,typename Hash=std::hash<Key> >
class threadsafe_lookup_table
{
private:
  class bucket_type
  {
  private:
    typedef std::pair<Key,Value> bucket_value;
    typedef std::list<bucket_value> bucket_data;
    typedef typename bucket_data::iterator bucket_iterator;

    bucket_data data;
    mutable boost::shared_mutex mutex;  // 1

    bucket_iterator find_entry_for(Key const& key) const  // 2
    {
      return std::find_if(data.begin(),data.end(),
      [&](bucket_value const& item)
      {return item.first==key;});
    }
  public:
    Value value_for(Key const& key,Value const& default_value) const
    {
      boost::shared_lock<boost::shared_mutex> lock(mutex);  // 3
```

```
25        bucket_iterator const found_entry=find_entry_for(key);
26        return (found_entry==data.end())?
27          default_value:found_entry->second;
28      }
29
30      void add_or_update_mapping(Key const& key,Value const& value)
31      {
32        std::unique_lock<boost::shared_mutex> lock(mutex);  // 4
33        bucket_iterator const found_entry=find_entry_for(key);
34        if(found_entry==data.end())
35        {
36          data.push_back(bucket_value(key,value));
37        }
38        else
39        {
40          found_entry->second=value;
41        }
42      }
43
44      void remove_mapping(Key const& key)
45      {
46        std::unique_lock<boost::shared_mutex> lock(mutex);  // 5
47        bucket_iterator const found_entry=find_entry_for(key);
48        if(found_entry!=data.end())
49        {
50          data.erase(found_entry);
51        }
52      }
53    };
54
55    std::vector<std::unique_ptr<bucket_type> > buckets;  // 6
56    Hash hasher;
57
58    bucket_type& get_bucket(Key const& key) const  // 7
59    {
60      std::size_t const bucket_index=hasher(key)%buckets.size();
61      return *buckets[bucket_index];
62    }
63
64  public:
65    typedef Key key_type;
66    typedef Value mapped_type;
67
68    typedef Hash hash_type;
69    threadsafe_lookup_table(
```

```
    unsigned num_buckets=19,Hash const& hasher_=Hash()):
    buckets(num_buckets),hasher(hasher_)
  {
    for(unsigned i=0;i<num_buckets;++i)
    {
      buckets[i].reset(new bucket_type);
    }
  }

  threadsafe_lookup_table(threadsafe_lookup_table const& other)=delete;
  threadsafe_lookup_table& operator=(
    threadsafe_lookup_table const& other)=delete;

  Value value_for(Key const& key,
                  Value const& default_value=Value()) const
  {
    return get_bucket(key).value_for(key,default_value);  // 8
  }

  void add_or_update_mapping(Key const& key,Value const& value)
  {
    get_bucket(key).add_or_update_mapping(key,value);  // 9
  }

  void remove_mapping(Key const& key)
  {
    get_bucket(key).remove_mapping(key);  // 10
  }
};
```

这个实现中使用了 `std::vector<std::unique_ptr<bucket_type>>` ⑥来保存桶，其允许在构造函数中指定构造桶的数量。默认为19个，其是一个任意的质数;哈希表在有质数个桶时，工作效率最高。每一个桶都会被一个 `boost::shared_mutex` ①实例锁保护，来允许并发读取，或对每一个桶，只有一个线程对其进行修改。

因为桶的数量是固定的，所以get_bucket()⑦可以无锁调用，⑧⑨⑩也都一样。并且对桶的互斥量上锁，要不就是共享(只读)所有权的时候③，要不就是在获取唯一(读/写)权的时候④⑤。这里的互斥量，可适用于每个成员函数。

这三个函数都使用到了find_entry_for()成员函数②，在桶上用来确定数据是否在桶中。每一个桶都包含一个“键值-数据”的 `std::list<>` 列表，所以添加和删除数据，就会很简单。

已经从并发的角度考虑了，并且所有成员都会被互斥锁保护，所以这样的实现就是“异常安全”的吗？value_for是不能修改任何值的，所以其不会有问题；如果value_for抛出异常，也不会对数据结构有任何影响。remove_mapping修改链表时，将会调用erase，不过这就能保证没有异常抛出，那么这里也是安全的。那么就剩add_or_update_mapping了，其可能会在其两个if分支上抛出异常。push_back是异常安全的，如果有异常抛出，其也会将链表恢复成原来的状态，所以这个分支是没有问题的。唯一的问题就是在赋值阶段，这将替换已有的数据；当复制阶段抛出异常，用于原依赖的始状态没有改变。不过，这不会影响数据结构的整体，以及用户提供类型的属性，所以你可以放心的将问题交给用户处理。

在本节开始时，我提到查询表的一个*可有可无*(nice-to-have)的特性，会将选择当前状态的快照，例如，一个 `std::map<>` 。这将要求锁住整个容器，用来保证拷贝副本的状态是可以索引的，这将要求锁住所有的桶。因为对于查询表的“普通”的操作，需要在同一时间获取一个桶上的一个锁，而这个操作将要求查询表将所有桶都锁住。因此，只要每次以相同的顺序进行上锁(例如，递增桶的索引值)，就不会产生死锁。实现如下所示：

清单6.12 获取整个threadsafe_lookup_table作为一个 `std::map<>`

```
std::map<Key,Value> threadsafe_lookup_table::get_map() const
{
  std::vector<std::unique_lock<boost::shared_mutex> > locks;
  for(unsigned i=0;i<buckets.size();++i)
  {
    locks.push_back(
      std::unique_lock<boost::shared_mutex>(buckets[i].mutex));
  }
  std::map<Key,Value> res;
  for(unsigned i=0;i<buckets.size();++i)
  {
    for(bucket_iterator it=buckets[i].data.begin();
        it!=buckets[i].data.end();
        ++it)
    {
      res.insert(*it);
    }
  }
  return res;
}
```

清单6.11中的查询表实现，就增大的并发访问的可能性，这个查询表作为一个整体，通过单独的操作，对每一个桶进行锁定，并且通过使用 `boost::shared_mutex` 允许读者线程对每一个桶进行并发访问。如果细粒度锁和哈希表结合起来，会更有效的增加并发的可能性吗？

在下一节中，你将使用到一个线程安全列表(支持迭代器)。

6.3.2 编写一个使用锁的线程安全链表

链表类型是数据结构中的一个基本类型，所以应该是比较好修改成线程安全的，对么？其实这取决于你要添加什么样的功能，这其中需要你提供迭代器的支持。为了让基本数据类型的代码不会太复杂，我去掉了一些功能。迭代器的问题在于，STL类的迭代器需要持有容器内部属于的引用。当容器可被其他线程修改时，有时这个引用还是有效的；实际上，这里就需要迭代器持有锁，对指定的结构中的部分进行上锁。在给定STL类迭代器的生命周期中，让其完全脱离容器的控制是很糟糕的。

替代方案就是提供迭代函数，例如，将for_each作为容器本身的一部分。这就能让容器来对迭代的部分进行负责和锁定，不过这将违反第3章指导意见对避免死锁建议。为了让for_each在任何情况下都有用，在其持有内部锁的时候，必须调用用户提供的代码。不仅如此，而且需要传递一个对容器中元素的引用到用户代码中，为的就是让用户代码对容器中的元素进行操作。你可以为了避免传递引用，而传出一个拷贝到用户代码中；不过当数据很大时，拷贝所要付出的代价也很大。

所以，可以将避免死锁的工作(因为用户提供的操作需要获取内部锁)，还有避免对引用(不被锁保护)进行存储时的条件竞争，交给用户去做。这样的链表就可以被查询表所使用了，这样很安全，因为你知道这里的实现不会有任何问题。

那么剩下的问题就是哪些操作需要列表所提供。如果你愿在花点时间看一下清单6.11和6.12中的代码，你会看到下面这些操作是需要的：

- 向列表添加一个元素
- 当某个条件满足时，就从链表中删除某个元素
- 当某个条件满足时，从链表中查找某个元素
- 当某个条件满足时，更新链表中的某个元素
- 将当前容器中链表中的每个元素，复制到另一个容器中

提供了这些操作，我们的链表才能是一个比较好的通用容器，这将帮助我们添加更多功能，比如，在指定位置上插入元素，不过这对于我们查询表来说就没有必要了，所以这里就算是给读者们留的一个作业吧。

使用细粒度锁最初的想法，是为了让链表每个节点都拥有一个互斥量。当链表很长时，那么就会有很多的互斥量!这样的好处是对于链表中每一个独立的部分，都能实现真实的并发：其真正感兴

趣的是对持有的节点群进行上锁，并且在移动到下一个节点的时，对当前节点进行释放。下面的清单中将展示这样的一个链表实现。

清单6.13 线程安全链表——支持迭代器

```
template<typename T>
class threadsafe_list
{
  struct node  // 1
  {
    std::mutex m;
    std::shared_ptr<T> data;
    std::unique_ptr<node> next;
    node():  // 2
      next()
    {}

    node(T const& value):  // 3
      data(std::make_shared<T>(value))
    {}
  };

  node head;

public:
  threadsafe_list()
  {}

  ~threadsafe_list()
  {
    remove_if([](node const&){return true;});
  }

  threadsafe_list(threadsafe_list const& other)=delete;
  threadsafe_list& operator=(threadsafe_list const& other)=delete;

  void push_front(T const& value)
  {
    std::unique_ptr<node> new_node(new node(value));  // 4
    std::lock_guard<std::mutex> lk(head.m);
    new_node->next=std::move(head.next);  // 5
    head.next=std::move(new_node);  // 6
  }

```

```
template<typename Function>
void for_each(Function f)  // 7
{
  node* current=&head;
  std::unique_lock<std::mutex> lk(head.m);  // 8
  while(node* const next=current->next.get())  // 9
  {
    std::unique_lock<std::mutex> next_lk(next->m);  // 10
    lk.unlock();  // 11
    f(*next->data);  // 12
    current=next;
    lk=std::move(next_lk);  // 13
  }
}

template<typename Predicate>
std::shared_ptr<T> find_first_if(Predicate p)  // 14
{
  node* current=&head;
  std::unique_lock<std::mutex> lk(head.m);
  while(node* const next=current->next.get())
  {
    std::unique_lock<std::mutex> next_lk(next->m);
    lk.unlock();
    if(p(*next->data))  // 15
    {
       return next->data;  // 16
    }
    current=next;
    lk=std::move(next_lk);
  }
  return std::shared_ptr<T>();
}

template<typename Predicate>
void remove_if(Predicate p)  // 17
{
  node* current=&head;
  std::unique_lock<std::mutex> lk(head.m);
  while(node* const next=current->next.get())
  {
    std::unique_lock<std::mutex> next_lk(next->m);
    if(p(*next->data))  // 18
    {
      std::unique_ptr<node> old_next=std::move(current->next);
```

```
85          current->next=std::move(next->next);
86          next_lk.unlock();
87        }  // 20
88        else
89        {
90          lk.unlock();  // 21
91          current=next;
92          lk=std::move(next_lk);
93        }
94      }
95    }
96  };
```

清单6.13中的threadsafe_list<>是一个单链表，可从node的结构①中看出。一个默认构造的node，作为链表的head，其next指针②指向的是NULL。新节点都是被push_front()函数添加进去的；构造第一个新节点④，其将会在堆上分配内存③来对数据进行存储，同时将next指针置为NULL。然后，你需要获取head节点的互斥锁，为了让设置next的值⑤，也就是插入节点到列表的头部，让头节点的head.next指向这个新节点⑥。目前，还没有什么问题：你只需要锁住一个互斥量，就能将一个新的数据添加进入链表，所以这里不存在死锁的问题。同样，(缓慢的)内存分配操作在锁的范围外，所以锁能保护需要更新的一对指针。那么，现在来看一下迭代功能。

首先，来看一下for_each()⑦。这个操作需要对队列中的每个元素执行Function(函数指针)；在大多数标准算法库中，都会通过传值方式来执行这个函数，这里要不就传入一个通用的函数，要不就传入一个有函数操作的类型对象。在这种情况下，这个函数必须接受类型为T的值作为参数。在链表中，会有一个“手递手”的上锁过程。在这个过程开始时，你需要锁住head及节点⑧的互斥量。然后，安全的获取指向下一个节点的指针(使用get()获取，这是因为你对这个指针没有所有权)。当指针不为NULL⑨，为了继续对数据进行处理，就需要对指向的节点进行上锁⑩。当你已经锁住了那个节点，就可以对上一个节点进行释放了⑪，并且调用指定函数⑫。当函数执行完成时，你就可以更新当前指针所指向的节点(刚刚处理过的节点)，并且将所有权从next_lk移动移动到lk⑬。因为for_each传递的每个数据都是能被Function接受的，所以当需要的时，需要拷贝到另一个容器的时，或其他情况时，你都可以考虑使用这种方式更新每个元素。如果函数的行为没什么问题，这种方式是完全安全的，因为在获取节点互斥锁时，已经获取锁的节点正在被函数所处理。

find_first_if()⑭和for_each()很相似；最大的区别在于find_first_if支持函数(谓词)在匹配的时候返回true，在不匹配的时候返回false⑮。当条件匹配，只需要返回找到的数据⑯，而非继续查找。你可以使用for_each()来做这件事，不过在找到之后，继续做查找就是没有意义的了。

remove_if()⑰就有些不同了，因为这个函数会改变链表；所以，你就不能使用for_each()来实现这个功能。当函数(谓词)返回true⑱，对应元素将会移除，并且更新current->next⑲。当这些都做完，你就可以释放next指向节点的锁。当 `std::unique_ptr<node>` 的移动超出链表范围⑳，这个

节点将被删除。这种情况下，你就不需要更新当前节点了，因为你只需要修改next所指向的下一个节点就可以。当函数(谓词)返回false，那么移动的操作就和之前一样了(21)。

那么，所有的互斥量中会有死锁或条件竞争吗？答案无疑是“否”，这里要看提供的函数(谓词)是否有良好的行为。迭代通常都是使用一种方式，都是从head节点开始，并且在释放当前节点锁之前，将下一个节点的互斥量锁住，所以这里就不可能会有不同线程有不同的上锁顺序。唯一可能出现条件竞争的地方就是在remove_if()⑳中删除已有节点的时候。因为，这个操作在解锁互斥量后进行(其导致的未定义行为，可对已上锁的互斥量进行破坏)。不过，在考虑一阵后，可以确定这的确是安全的，因为你还持有前一个节点(当前节点)的互斥锁，所以不会有新的线程尝试去获取你正在删除的那个节点的互斥锁。

这里并发概率有多大呢？细粒度锁要比单锁的并发概率大很多，那我们已经获得了吗？是的，你已经获取了：同一时间内，不同线程可以在不同节点上工作，无论是其使用for_each()对每一个节点进行处理，使用find_first_if()对数据进行查找，还是使用remove_if()删除一些元素。不过，因为互斥量必须按顺序上锁，那么线程就不能交叉进行工作。当一个线程耗费大量的时间对一个特殊节点进行处理，那么其他线程就必须等待这个处理完成。在完成后，其他线程才能到达这个节点。