# 3.2 使用互斥量保护共享数据

当程序中有共享数据，肯定不想让其陷入条件竞争，或是不变量被破坏。那么，将所有访问共享数据结构的代码都标记为互斥岂不是更好？这样任何一个线程在执行这些代码时，其他任何线程试图访问共享数据结构，就必须等到那一段代码执行结束。于是，一个线程就不可能会看到被破坏的不变量，除非它本身就是修改共享数据的线程。

当访问共享数据前，使用互斥量将相关数据锁住，再当访问结束后，再将数据解锁。线程库需要保证，当一个线程使用特定互斥量锁住共享数据时，其他的线程想要访问锁住的数据，都必须等到之前那个线程对数据进行解锁后，才能进行访问。这就保证了所有线程能看到共享数据，而不破坏不变量。

互斥量是 C++ 中一种最通用的数据保护机制，但它不是“银弹”；精心组织代码来保护正确的数据(见3.2.2节)，并在接口内部避免竞争条件(见3.2.3节)是非常重要的。但互斥量自身也有问题，也会造成死锁(见3.2.4节)，或是对数据保护的太多(或太少)(见3.2.8节)。

## 3.2.1 C++中使用互斥量

C++中通过实例化 `std::mutex` 创建互斥量，通过调用成员函数lock()进行上锁，unlock()进行解锁。不过，不推荐实践中直接去调用成员函数，因为调用成员函数就意味着，必须记住在每个函数出口都要去调用unlock()，也包括异常的情况。C++标准库为互斥量提供了一个RAII语法的模板类 `std::lock_guard` ，其会在构造的时候提供已锁的互斥量，并在析构的时候进行解锁，从而保证了一个已锁的互斥量总是会被正确的解锁。下面的程序清单中，展示了如何在多线程程序中，使用 `std::mutex` 构造的 `std::lock_guard` 实例，对一个列表进行访问保护。 `std::mutex` 和 `std::lock_guard` 都在 `<mutex>` 头文件中声明。

清单3.1 使用互斥量保护列表

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> some_list;    // 1
std::mutex some_mutex;    // 2
```

```
7
8  void add_to_list(int new_value)
9  {
10   std::lock_guard<std::mutex> guard(some_mutex);    // 3
11   some_list.push_back(new_value);
12 }
13
14 bool list_contains(int value_to_find)
15 {
16   std::lock_guard<std::mutex> guard(some_mutex);    // 4
17   return std::find(some_list.begin(),some_list.end(),value_to_find) != some_lis
18 }
```

清单3.1中有一个全局变量①，这个全局变量被一个全局的互斥量保护②。add_to_list()③和list_contains()④函数中使用 `std::lock_guard<std::mutex>` ，使得这两个函数中对数据的访问是互斥的：list_contains()不可能看到正在被add_to_list()修改的列表。

虽然某些情况下，使用全局变量没问题，但在大多数情况下，互斥量通常会与保护的数据放在同一个类中，而不是定义成全局变量。这是面向对象设计的准则：将其放在一个类中，就可让他们联系在一起，也可对类的功能进行封装，并进行数据保护。在这种情况下，函数add_to_list和list_contains可以作为这个类的成员函数。互斥量和要保护的数据，在类中都需要定义为private成员，这会让访问数据的代码变的清晰，并且容易看出在什么时候对互斥量上锁。当所有成员函数都会在调用时对数据上锁，结束时对数据解锁，那么就保证了数据访问时不变量不被破坏。

当然，也不是总是那么理想，聪明的你一定注意到了：当其中一个成员函数返回的是保护数据的指针或引用时，会破坏对数据的保护。具有访问能力的指针或引用可以访问(并可能修改)被保护的数据，而不会被互斥锁限制。互斥量保护的数据需要对接口的设计相当谨慎，要确保互斥量能锁住任何对保护数据的访问，并且不留后门。

## 3.2.2 精心组织代码来保护共享数据

使用互斥量来保护数据，并不是仅仅在每一个成员函数中都加入一个 `std::lock_guard` 对象那么简单；一个迷失的指针或引用，将会让这种保护形同虚设。不过，检查迷失指针或引用是很容易的，只要没有成员函数通过返回值或者输出参数的形式向其调用者返回指向受保护数据的指针或引用，数据就是安全的。如果你还想往祖坟上刨，就没这么简单了。在确保成员函数不会传出指针或引用的同时，检查成员函数是否通过指针或引用的方式来调用也是很重要的(尤其是这个操作不在你的控制下时)。函数可能没在互斥量保护的区域内，存储着指针或者引用，这样就很危险。更危险的是：将保护数据作为一个运行时参数，如同下面清单中所示那样。

清单3.2 无意中传递了保护数据的引用

```
class some_data
{
  int a;
  std::string b;
public:
  void do_something();
};

class data_wrapper
{
private:
  some_data data;
  std::mutex m;
public:
  template<typename Function>
  void process_data(Function func)
  {
    std::lock_guard<std::mutex> l(m);
    func(data);    // 1 传递“保护”数据给用户函数
  }
};

some_data* unprotected;

void malicious_function(some_data& protected_data)
{
  unprotected=&protected_data;
}

data_wrapper x;
void foo()
{
  x.process_data(malicious_function);    // 2 传递一个恶意函数
  unprotected->do_something();    // 3 在无保护的情况下访问保护数据
}
```

例子中process_data看起来没有任何问题， `std::lock_guard` 对数据做了很好的保护，但调用用户提供的函数func①，就意味着foo能够绕过保护机制将函数 `malicious_function` 传递进去②，在没有锁定互斥量的情况下调用 `do_something()` 。

这段代码的问题在于根本没有保护，只是将所有可访问的数据结构代码标记为互斥。函数 `foo()` 中调用 `unprotected->do_something()` 的代码未能被标记为互斥。这种情况下，C++线程库无法提供任何帮助，只能由程序员来使用正确的互斥锁来保护数据。从乐观的角度上看，还是有方法可循的：切勿将受保护数据的指针或引用传递到互斥锁作用域之外，无论是函数返回值，还是存储在外部可见内存，亦或是以参数的形式传递到用户提供的函数中去。

虽然这是在使用互斥量保护共享数据时常犯的错误，但绝不仅仅是一个潜在的陷阱而已。下一节中，你将会看到，即便是使用了互斥量对数据进行了保护，条件竞争依旧可能存在。

## 3.2.3 发现接口内在的条件竞争

因为使用了互斥量或其他机制保护了共享数据，就不必再为条件竞争所担忧吗？并不是，你依旧需要确定数据受到了保护。回想之前双链表的例子，为了能让线程安全地删除一个节点，需要确保防止对这三个节点(待删除的节点及其前后相邻的节点)的并发访问。如果只对指向每个节点的指针进行访问保护，那就和没有使用互斥量一样，条件竞争仍会发生——除了指针，整个数据结构和整个删除操作需要保护。这种情况下最简单的解决方案就是使用互斥量来保护整个链表，如清单3.1所示。

尽管链表的个别操作是安全的，但不意味着你就能走出困境；即使在一个很简单的接口中，依旧可能遇到条件竞争。例如，构建一个类似于 `std::stack` 结构的栈(清单3.3)，除了构造函数和swap()以外，需要对 `std::stack` 提供五个操作：push()一个新元素进栈，pop()一个元素出栈，top()查看栈顶元素，empty()判断栈是否是空栈，size()了解栈中有多少个元素。即使修改了top()，使其返回一个拷贝而非引用(即遵循了3.2.2节的准则)，对内部数据使用一个互斥量进行保护，不过这个接口仍存在条件竞争。这个问题不仅存在于基于互斥量实现的接口中，在无锁实现的接口中，条件竞争依旧会产生。这是接口的问题，与其实现方式无关。

清单3.3 `std::stack` 容器的实现

```
template<typename T,typename Container=std::deque<T> >
class stack
{
public:
  explicit stack(const Container&);
  explicit stack(Container&& = Container());
  template <class Alloc> explicit stack(const Alloc&);
  template <class Alloc> stack(const Container&, const Alloc&);
  template <class Alloc> stack(Container&&, const Alloc&);
  template <class Alloc> stack(stack&&, const Alloc&);
```

```
11
12    bool empty() const;
13    size_t size() const;
14    T& top();
15    T const& top() const;
16    void push(T const&);
17    void push(T&&);
18    void pop();
19    void swap(stack&&);
20  };
```

虽然empty()和size()可能在被调用并返回时是正确的，但其的结果是不可靠的；当它们返回后，其他线程就可以自由地访问栈，并且可能push()多个新元素到栈中，也可能pop()一些已在栈中的元素。这样的话，之前从empty()和size()得到的结果就有问题了。

特别地，当栈实例是非共享的，如果栈非空，使用empty()检查再调用top()访问栈顶部的元素是安全的。如下代码所示：

```
1  stack<int> s;
2  if (! s.empty()){    // 1
3    int const value = s.top();    // 2
4    s.pop();    // 3
5    do_something(value);
6  }
```

以上是单线程安全代码：对一个空栈使用top()是未定义行为。对于共享的栈对象，这样的调用顺序就不再安全了，因为在调用empty()①和调用top()②之间，可能有来自另一个线程的pop()调用并删除了最后一个元素。这是一个经典的条件竞争，使用互斥量对栈内部数据进行保护，但依旧不能阻止条件竞争的发生，这就是接口固有的问题。

怎么解决呢？问题发生在接口设计上，所以解决的方法也就是改变接口设计。有人会问：怎么改？在这个简单的例子中，当调用top()时，发现栈已经是空的了，那么就抛出异常。虽然这能直接解决这个问题，但这是一个笨拙的解决方案，这样的话，即使empty()返回false的情况下，你也需要异常捕获机制。本质上，这样的改变会让empty()成为一个多余函数。

当仔细的观察过之前的代码段，就会发现另一个潜在的条件竞争在调用top()②和pop()③之间。假设两个线程运行着前面的代码，并且都引用同一个栈对象s。这并非罕见的情况，当为性能而使用线程时，多个线程在不同的数据上执行相同的操作是很平常的，并且共享同一个栈可以将工作分摊给它们。假设，一开始栈中只有两个元素，这时任一线程上的empty()和top()都存在竞争，只需要考虑可能的执行顺序即可。

当栈被一个内部互斥量所保护时，只有一个线程可以调用栈的成员函数，所以调用可以很好地交错，并且do_something()是可以并发运行的。在表3.1中，展示一种可能的执行顺序。

表3.1 一种可能执行顺序

| Thread A | Thread B |
| --- | --- |
| if (!s.empty); | |
| | if(!s.empty); |
| int const value = s.top(); | |
| | int const value = s.top(); |
| s.pop(); | |
| do_something(value); | s.pop(); |
| | do_something(value); |

当线程运行时，调用两次top()，栈没被修改，所以每个线程能得到同样的值。不仅是这样，在调用top()函数调用的过程中(两次)，pop()函数都没有被调用。这样，在其中一个值再读取的时候，虽然不会出现“写后读”的情况，但其值已被处理了两次。这种条件竞争，比未定义的empty()/top()竞争更加严重；虽然其结果依赖于do_something()的结果，但因为看起来没有任何错误，就会让这个Bug很难定位。

这就需要接口设计上有较大的改动，提议之一就是使用同一互斥量来保护top()和pop()。Tom Cargill[1]指出当一个对象的拷贝构造函数在栈中抛出一个异常，这样的处理方式就会有问题。在Herb Sutter[2]看来，这个问题可以从“异常安全”的角度完美解决，不过潜在的条件竞争，可能会组成一些新的组合。

说一些大家没有意识到的问题：假设有一个 `stack<vector<int>>` ，vector是一个动态容器，当你拷贝一个vetcor，标准库会从堆上分配很多内存来完成这次拷贝。当这个系统处在重度负荷，或有严重的资源限制的情况下，这种内存分配就会失败，所以vector的拷贝构造函数可能会抛出一个 `std::bad_alloc` 异常。当vector中存有大量元素时，这种情况发生的可能性更大。当pop()函数返回“弹出值”时(也就是从栈中将这个值移除)，会有一个潜在的问题：这个值被返回到调用函数的时候，栈才被改变；但当拷贝数据的时候，调用函数抛出一个异常会怎么样？ 如果事情真的发生了，要弹出的数据将会丢失；它的确从栈上移出了，但是拷贝失败了！ `std::stack` 的设计人员将这个操作分为两部分：先获取顶部元素(top())，然后从栈中移除(pop())。这样，在不能安全的

将元素拷贝出去的情况下，栈中的这个数据还依旧存在，没有丢失。当问题是堆空间不足，应用可能会释放一些内存，然后再进行尝试。

不幸的是，这样的分割却制造了本想避免或消除的条件竞争。幸运的是，我们还有的别的选项，但是使用这些选项是要付出代价的。

### 选项1： 传入一个引用

第一个选项是将变量的引用作为参数，传入pop()函数中获取想要的“弹出值”：

```
std::vector<int> result;
some_stack.pop(result);
```

大多数情况下，这种方式还不错，但有明显的缺点：需要构造出一个栈中类型的实例，用于接收目标值。对于一些类型，这样做是不现实的，因为临时构造一个实例，从时间和资源的角度上来看，都是不划算。对于其他的类型，这样也不总能行得通，因为构造函数需要的一些参数，在代码的这个阶段不一定可用。最后，需要可赋值的存储类型，这是一个重大限制：即使支持移动构造，甚至是拷贝构造(从而允许返回一个值)，很多用户自定义类型可能都不支持赋值操作。

### 选项2：无异常抛出的拷贝构造函数或移动构造函数

对于有返回值的pop()函数来说，只有“异常安全”方面的担忧(当返回值时可以抛出一个异常)。很多类型都有拷贝构造函数，它们不会抛出异常，并且随着新标准中对“右值引用”的支持(详见附录A，A.1节)，很多类型都将会有一个移动构造函数，即使他们和拷贝构造函数做着相同的事情，它也不会抛出异常。一个有用的选项可以限制对线程安全的栈的使用，并且能让栈安全的返回所需的值，而不会抛出异常。

虽然安全，但非可靠。尽管能在编译时可使用 `std::is_nothrow_copy_constructible` 和 `std::is_nothrow_move_constructible` 类型特征，让拷贝或移动构造函数不抛出异常，但是这种方式的局限性太强。用户自定义的类型中，会有不抛出异常的拷贝构造函数或移动构造函数的类型， 那些有抛出异常的拷贝构造函数，但没有移动构造函数的类型往往更多（这种情况会随着人们习惯于C++11中的右值引用而有所改变)。如果这些类型不能被存储在线程安全的栈中，那将是多么的不幸。

### 、选项3：返回指向弹出值的指针

第三个选择是返回一个指向弹出元素的指针，而不是直接返回值。指针的优势是自由拷贝，并且不会产生异常，这样你就能避免Cargill提到的异常问题了。缺点就是返回一个指针需要对对象的内存分配进行管理，对于简单数据类型(比如：int)，内存管理的开销要远大于直接返回值。对于选择这个方案的接口，使用 `std::shared_ptr` 是个不错的选择；不仅能避免内存泄露(因为当对象中

指针销毁时，对象也会被销毁)，而且标准库能够完全控制内存分配方案，也就不需要new和delete操作。这种优化是很重要的：因为堆栈中的每个对象，都需要用new进行独立的内存分配，相较于非线程安全版本，这个方案的开销相当大。

**选项4：“选项1 + 选项2”或“选项1 + 选项3”**

对于通用的代码来说，灵活性不应忽视。当你已经选择了选项2或3时，再去选择1也是很容易的。这些选项提供给用户，让用户自己选择对于他们自己来说最合适，最经济的方案。

**例：定义线程安全的堆栈**

清单3.4中是一个接口没有条件竞争的堆栈类定义，它实现了选项1和选项3：重载了pop()，使用一个局部引用去存储弹出值，并返回一个 `std::shared_ptr<>` 对象。它有一个简单的接口，只有两个函数：push()和pop();

清单3.4 线程安全的堆栈类定义(概述)

```
#include <exception>
#include <memory>  // For std::shared_ptr<>

struct empty_stack: std::exception
{
  const char* what() const throw();
};

template<typename T>
class threadsafe_stack
{
public:
  threadsafe_stack();
  threadsafe_stack(const threadsafe_stack&);
  threadsafe_stack& operator=(const threadsafe_stack&) = delete; // 1 赋值操作被

  void push(T new_value);
  std::shared_ptr<T> pop();
  void pop(T& value);
  bool empty() const;
};
```

削减接口可以获得最大程度的安全,甚至限制对栈的一些操作。栈是不能直接赋值的，因为赋值操作已经删除了①(详见附录A，A.2节)，并且这里没有swap()函数。栈可以拷贝的，假设栈中的元素

可以拷贝。当栈为空时，pop()函数会抛出一个empty_stack异常，所以在empty()函数被调用后，其他部件还能正常工作。如选项3描述的那样，使用 `std::shared_ptr` 可以避免内存分配管理的问题，并避免多次使用new和delete操作。堆栈中的五个操作，现在就剩下三个：push(), pop()和empty()(这里empty()都有些多余)。简化接口更有利于数据控制，可以保证互斥量将一个操作完全锁住。下面的代码将展示一个简单的实现——封装 `std::stack<>` 的线程安全堆栈。

清单3.5 扩充(线程安全)堆栈

```
#include <exception>
#include <memory>
#include <mutex>
#include <stack>

struct empty_stack: std::exception
{
  const char* what() const throw() {
    return "empty stack!";
  };
};

template<typename T>
class threadsafe_stack
{
private:
  std::stack<T> data;
  mutable std::mutex m;

public:
  threadsafe_stack()
    : data(std::stack<T>()){}

  threadsafe_stack(const threadsafe_stack& other)
  {
    std::lock_guard<std::mutex> lock(other.m);
    data = other.data; // 1 在构造函数体中的执行拷贝
  }

  threadsafe_stack& operator=(const threadsafe_stack&) = delete;

  void push(T new_value)
  {
    std::lock_guard<std::mutex> lock(m);
    data.push(new_value);
```

```
36    }
37
38    std::shared_ptr<T> pop()
39    {
40      std::lock_guard<std::mutex> lock(m);
41      if(data.empty()) throw empty_stack(); // 在调用pop前，检查栈是否为空
42
43      std::shared_ptr<T> const res(std::make_shared<T>(data.top())); // 在修改堆栈
44      data.pop();
45      return res;
46    }
47
48    void pop(T& value)
49    {
50      std::lock_guard<std::mutex> lock(m);
51      if(data.empty()) throw empty_stack();
52
53      value=data.top();
54      data.pop();
55    }
56
57    bool empty() const
58    {
59      std::lock_guard<std::mutex> lock(m);
60      return data.empty();
61    }
62  };
```

堆栈可以拷贝——拷贝构造函数对互斥量上锁，再拷贝堆栈。构造函数体中①的拷贝使用互斥量来确保复制结果的正确性，这样的方式比成员初始化列表好。

之前对top()和pop()函数的讨论中，恶性条件竞争已经出现，因为锁的粒度太小，需要保护的操作并未全覆盖到。不过，锁住的颗粒过大同样会有问题。还有一个问题，一个全局互斥量要去保护全部共享数据，在一个系统中存在有大量的共享数据时，因为线程可以强制运行，甚至可以访问不同位置的数据，抵消了并发带来的性能提升。在第一版为多处理器系统设计Linux内核中，就使用了一个全局内核锁。虽然这个锁能正常工作，但在双核处理系统的上的性能要比两个单核系统的性能差很多，四核系统就更不能提了。太多请求去竞争占用内核，使得依赖于处理器运行的线程没有办法很好的工作。随后修正的Linux内核加入了一个细粒度锁方案，因为少了很多内核竞争，这时四核处理系统的性能就和单核处理的四倍差不多了。

使用多个互斥量保护所有的数据，细粒度锁也有问题。如前所述，当增大互斥量覆盖数据的粒度时，只需要锁住一个互斥量。但是，这种方案并非放之四海皆准，比如：互斥量正在保护一个独

立类的实例；这种情况下，锁的状态的下一个阶段，不是离开锁定区域将锁定区域还给用户，就是有独立的互斥量去保护这个类的全部实例。当然，这两种方式都不理想。

一个给定操作需要两个或两个以上的互斥量时，另一个潜在的问题将出现：死锁。与条件竞争完全相反——不同的两个线程会互相等待，从而什么都没做。

## 3.2.4 死锁：问题描述及解决方案

试想有一个玩具，这个玩具由两部分组成，必须拿到这两个部分，才能够玩。例如，一个玩具鼓，需要一个鼓锤和一个鼓才能玩。现在有两个小孩，他们都很喜欢玩这个玩具。当其中一个孩子拿到了鼓和鼓锤时，那就可以尽情的玩耍了。当另一孩子想要玩，他就得等待另一孩子玩完才行。再试想，鼓和鼓锤被放在不同的玩具箱里，并且两个孩子在同一时间里都想要去敲鼓。之后，他们就去玩具箱里面找这个鼓。其中一个找到了鼓，并且另外一个找到了鼓锤。现在问题就来了，除非其中一个孩子决定让另一个先玩，他可以把自己的那部分给另外一个孩子；但当他们都紧握着自己所有的部分而不给予，那么这个鼓谁都没法玩。

现在没有孩子去争抢玩具，但线程有对锁的竞争：一对线程需要对他们所有的互斥量做一些操作，其中每个线程都有一个互斥量，且等待另一个解锁。这样没有线程能工作，因为他们都在等待对方释放互斥量。这种情况就是死锁，它的最大问题就是由两个或两个以上的互斥量来锁定一个操作。

避免死锁的一般建议，就是让两个互斥量总以相同的顺序上锁：总在互斥量B之前锁住互斥量A，就永远不会死锁。某些情况下是可以这样用，因为不同的互斥量用于不同的地方。不过，事情没那么简单，比如：当有多个互斥量保护同一个类的独立实例时，一个操作对同一个类的两个不同实例进行数据的交换操作，为了保证数据交换操作的正确性，就要避免数据被并发修改，并确保每个实例上的互斥量都能锁住自己要保护的区域。不过，选择一个固定的顺序(例如，实例提供的第一互斥量作为第一个参数，提供的第二个互斥量为第二个参数)，可能会适得其反：**在参数交换了之后**，两个线程试图在相同的两个实例间进行数据交换时，程序又死锁了！

很幸运，C++标准库有办法解决这个问题， `std::lock` ——可以一次性锁住多个(两个以上)的互斥量，并且没有副作用(死锁风险)。下面的程序清单中，就来看一下怎么在一个简单的交换操作中使用 `std::lock` 。

清单3.6 交换操作中使用 `std::lock()` 和 `std::lock_guard`

```
1 // 这里的std::lock()需要包含<mutex>头文件
2 class some_big_object;
3 void swap(some_big_object& lhs,some_big_object& rhs);
```

```
class X
{
private:
  some_big_object some_detail;
  std::mutex m;
public:
  X(some_big_object const& sd):some_detail(sd){}

  friend void swap(X& lhs, X& rhs)
  {
    if(&lhs==&rhs)
      return;
    std::lock(lhs.m,rhs.m); // 1
    std::lock_guard<std::mutex> lock_a(lhs.m,std::adopt_lock); // 2
    std::lock_guard<std::mutex> lock_b(rhs.m,std::adopt_lock); // 3
    swap(lhs.some_detail,rhs.some_detail);
  }
};
```

首先，检查参数是否是不同的实例，因为操作试图获取 `std::mutex` 对象上的锁，所以当其被获取时，结果很难预料。(一个互斥量可以在同一线程上多次上锁，标准库中 `std::recursive_mutex` 提供这样的功能。详情见3.3.3节)。然后，调用 `std::lock()` ①锁住两个互斥量，并且两个 `std:lock_guard` 实例已经创建好②③。提供 `std::adopt_lock` 参数除了表示 `std::lock_guard` 对象可获取锁之外，还将锁交由 `std::lock_guard` 对象管理，而不需要 `std::lock_guard` 对象再去构建新的锁。

这样，就能保证在大多数情况下，函数退出时互斥量能被正确的解锁(保护操作可能会抛出一个异常)，也允许使用一个简单的“return”作为返回。还有，需要注意的是，当使用 `std::lock` 去锁lhs.m或rhs.m时，可能会抛出异常；这种情况下，异常会传播到 `std::lock` 之外。当 `std::lock` 成功的获取一个互斥量上的锁，并且当其尝试从另一个互斥量上再获取锁时，就会有异常抛出，第一个锁也会随着异常的产生而自动释放，所以 `std::lock` 要么将两个锁都锁住，要不一个都不锁。

虽然 `std::lock` 可以在这情况下(获取两个以上的锁)避免死锁，但它没办法帮助你获取其中一个锁。这时，不得不依赖于开发者的纪律性(译者：也就是经验)，来确保你的程序不会死锁。这并不简单：死锁是多线程编程中一个令人相当头痛的问题，并且死锁经常是不可预见的，因为在大多数时间里，所有工作都能很好的完成。不过，也一些相对简单的规则能帮助写出“无死锁”的代码。

## 3.2.5 避免死锁的进阶指导

虽然锁是产生死锁的一般原因，但也不排除死锁出现在其他地方。无锁的情况下，仅需要每个 `std::thread` 对象调用join()，两个线程就能产生死锁。这种情况下，没有线程可以继续运行，因为他们正在互相等待。这种情况很常见，一个线程会等待另一个线程，其他线程同时也会等待第一个线程结束，所以三个或更多线程的互相等待也会发生死锁。为了避免死锁，这里的指导意见为：当机会来临时，不要拱手让人。以下提供一些个人的指导建议，如何识别死锁，并消除其他线程的等待。

### 避免嵌套锁

第一个建议往往是最简单的：一个线程已获得一个锁时，再别去获取第二个。如果能坚持这个建议，因为每个线程只持有一个锁，锁上就不会产生死锁。即使互斥锁造成死锁的最常见原因，也可能会在其他方面受到死锁的困扰(比如：线程间的互相等待)。当你需要获取多个锁，使用一个 `std::lock` 来做这件事(对获取锁的操作上锁)，避免产生死锁。

### 避免在持有锁时调用用户提供的代码

第二个建议是次简单的：因为代码是用户提供的，你没有办法确定用户要做什么；用户程序可能做任何事情，包括获取锁。你在持有锁的情况下，调用用户提供的代码；如果用户代码要获取一个锁，就会违反第一个指导意见，并造成死锁(有时，这是无法避免的)。当你正在写一份通用代码，例如3.2.3中的栈，每一个操作的参数类型，都在用户提供的代码中定义，就需要其他指导意见来帮助你。

### 使用固定顺序获取锁

当硬性条件要求你获取两个以上(包括两个)的锁，并且不能使用 `std::lock` 单独操作来获取它们;那么最好在每个线程上，用固定的顺序获取它们获取它们(锁)。3.2.4节中提到一种当需要获取两个互斥量时，避免死锁的方法：关键是如何在线程之间，以一定的顺序获取锁。一些情况下，这种方式相对简单。比如，3.2.3节中的栈——每个栈实例中都内置有互斥量，但是对数据成员存储的操作上，栈就需要带调用用户提供的代码。虽然，可以添加一些约束，对栈上存储的数据项不做任何操作，对数据项的处理仅限于栈自身。这会给用户提供的栈增加一些负担，但是一个容器很少去访问另一个容器中存储的数据，即使发生了也会很明显，所以这对于通用栈来说并不是一个特别沉重的负担。

其他情况下，这就不会那么简单了，例如：3.2.4节中的交换操作，这种情况下你可能同时锁住多个互斥量(但是有时不会发生)。当回看3.1节中那个链表连接例子时，将会看到列表中的每个节点都会有一个互斥量保护。为了访问列表，线程必须获取他们感兴趣节点上的互斥锁。当一个线程删除一个节点，它必须获取三个节点上的互斥锁：将要删除的节点，两个邻接节点(因为他们也会被修改)。同样的，为了遍历链表，线程必须保证在获取当前节点的互斥锁前提下，获得下一个节点的锁，要保证指向下一个节点的指针不会同时被修改。一旦下一个节点上的锁被获取，那么第一个节点的锁就可以释放了，因为没有持有它的必要性了。

这种“手递手”锁的模式允许多个线程访问列表，为每一个访问的线程提供不同的节点。但是，为了避免死锁，节点必须以同样的顺序上锁：如果两个线程试图用互为反向的顺序，使用“手递手”锁遍历列表，他们将执行到列表中间部分时，发生死锁。当节点A和B在列表中相邻，当前线程可能会同时尝试获取A和B上的锁。另一个线程可能已经获取了节点B上的锁，并且试图获取节点A上的锁——经典的死锁场景。

当A、C节点中的B节点正在被删除时，如果有线程在已获取A和C上的锁后，还要获取B节点上的锁时，当一个线程遍历列表的时候，这样的情况就可能发生死锁。这样的线程可能会试图首先锁住A节点或C节点(根据遍历的方向)，但是后面就会发现，它无法获得B上的锁，因为线程在执行删除任务的时候，已经获取了B上的锁，并且同时也获取了A和C上的锁。

这里提供一种避免死锁的方式，定义遍历的顺序，所以一个线程必须先锁住A才能获取B的锁，在锁住B之后才能获取C的锁。这将消除死锁发生的可能性，在不允许反向遍历的列表上。类似的约定常被用来建立其他的数据结构。

### 使用锁的层次结构

虽然，这对于定义锁的顺序，的确是一个特殊的情况，但锁的层次的意义在于提供对运行时约定是否被坚持的检查。这个建议需要对你的应用进行分层，并且识别在给定层上所有可上锁的互斥量。当代码试图对一个互斥量上锁，在该层锁已被低层持有时，上锁是不允许的。你可以在运行时对其进行检查，通过分配层数到每个互斥量上，以及记录被每个线程上锁的互斥量。下面的代码列表中将展示两个线程如何使用分层互斥。

清单3.7 使用层次锁来避免死锁

```
hierarchical_mutex high_level_mutex(10000); // 1
hierarchical_mutex low_level_mutex(5000);  // 2

int do_low_level_stuff();

int low_level_func()
{
  std::lock_guard<hierarchical_mutex> lk(low_level_mutex); // 3
  return do_low_level_stuff();
}

void high_level_stuff(int some_param);

void high_level_func()
{
  std::lock_guard<hierarchical_mutex> lk(high_level_mutex); // 4
  high_level_stuff(low_level_func()); // 5
```

```
}

void thread_a()  // 6
{
  high_level_func();
}

hierarchical_mutex other_mutex(100); // 7
void do_other_stuff();

void other_stuff()
{
  high_level_func();  // 8
  do_other_stuff();
}

void thread_b() // 9
{
  std::lock_guard<hierarchical_mutex> lk(other_mutex); // 10
  other_stuff();
}
```

thread_a()⑥遵守规则，所以它运行的没问题。另一方面，thread_b()⑨无视规则，因此在运行的时候肯定会失败。thread_a()调用high_level_func()，让high_level_mutex④上锁(其层级值为10000①)，为了获取high_level_stuff()的参数对互斥量上锁，之后调用low_level_func()⑤。low_level_func()会对low_level_mutex上锁，这就没有问题了，因为这个互斥量有一个低层值5000②。

thread_b()运行就不会顺利了。首先，它锁住了other_mutex⑩，这个互斥量的层级值只有100⑦。这就意味着，超低层级的数据已被保护。当other_stuff()调用high_level_func()⑧时，就违反了层级结构：high_level_func()试图获取high_level_mutex，这个互斥量的层级值是10000，要比当前层级值100大很多。因此hierarchical_mutex将会产生一个错误，可能会是抛出一个异常，或直接终止程序。在层级互斥量上产生死锁，是不可能的，因为互斥量本身会严格遵循约定顺序，进行上锁。这也意味，当多个互斥量在是在同一级上时，不能同时持有多个锁，所以“手递手”锁的方案需要每个互斥量在一条链上，并且每个互斥量都比其前一个有更低的层级值，这在某些情况下无法实现。

例子也展示了另一点， `std::lock_guard<>` 模板与用户定义的互斥量类型一起使用。虽然hierarchical_mutex不是 `C++` 标准的一部分，但是它写起来很容易；一个简单的实现在列表3.8中展示出来。尽管它是一个用户定义类型，它可以用于 `std::lock_guard<>` 模板中，因为它的实现有三个成员函数为了满足互斥量操作：lock(), unlock() 和 try_lock()。虽然你还没见过try_lock()怎么使用，但是其使用起来很简单：当互斥量上的锁被一个线程持有，它将返回false，而不是等待

调用的线程，直到能够获取互斥量上的锁为止。在 `std::lock()` 的内部实现中，try_lock()会作为避免死锁算法的一部分。

列表3.8 简单的层级互斥量实现

```
class hierarchical_mutex
{
  std::mutex internal_mutex;

  unsigned long const hierarchy_value;
  unsigned long previous_hierarchy_value;

  static thread_local unsigned long this_thread_hierarchy_value;  // 1

  void check_for_hierarchy_violation()
  {
    if(this_thread_hierarchy_value <= hierarchy_value)  // 2
    {
      throw std::logic_error(“mutex hierarchy violated”);
    }
  }

  void update_hierarchy_value()
  {
    previous_hierarchy_value=this_thread_hierarchy_value;  // 3
    this_thread_hierarchy_value=hierarchy_value;
  }

public:
  explicit hierarchical_mutex(unsigned long value):
      hierarchy_value(value),
      previous_hierarchy_value(0)
  {}

  void lock()
  {
    check_for_hierarchy_violation();
    internal_mutex.lock();  // 4
    update_hierarchy_value();  // 5
  }

  void unlock()
  {
    this_thread_hierarchy_value=previous_hierarchy_value;  // 6
```

```
    internal_mutex.unlock();
  }

  bool try_lock()
  {
    check_for_hierarchy_violation();
    if(!internal_mutex.try_lock())  // 7
      return false;
    update_hierarchy_value();
    return true;
  }
};
thread_local unsigned long
     hierarchical_mutex::this_thread_hierarchy_value(ULONG_MAX);  // 8
```

这里重点是使用了thread_local的值来代表当前线程的层级值：this_thread_hierarchy_value①。它被初始化为最大值⑧，所以最初所有线程都能被锁住。因为其声明中有thread_local，所以每个线程都有其拷贝副本，这样线程中变量状态完全独立，当从另一个线程进行读取时，变量的状态也完全独立。参见附录A，A.8节，有更多与thread_local相关的内容。

所以，第一次线程锁住一个hierarchical_mutex时，this_thread_hierarchy_value的值是ULONG_MAX。由于其本身的性质，这个值会大于其他任何值，所以会通过check_for_hierarchy_vilation()②的检查。在这种检查方式下，lock()代表内部互斥锁已被锁住④。一旦成功锁住，你可以更新层级值了⑤。

当你现在锁住另一个hierarchical_mutex时，还持有第一个锁，this_thread_hierarchy_value的值将会显示第一个互斥量的层级值。第二个互斥量的层级值必须小于已经持有互斥量检查函数②才能通过。

现在，最重要的是为当前线程存储之前的层级值，所以你可以调用unlock()⑥对层级值进行保存；否则，就锁不住任何互斥量(第二个互斥量的层级数高于第一个互斥量)，即使线程没有持有任何锁。因为保存了之前的层级值，只有当持有internal_mutex③，且在解锁内部互斥量⑥之前存储它的层级值，才能安全的将hierarchical_mutex自身进行存储。这是因为hierarchical_mutex被内部互斥量的锁所保护着。

try_lock()与lock()的功能相似，除了在调用internal_mutex的try_lock()⑦失败时，不能持有对应锁，所以不必更新层级值，并直接返回false。

虽然是运行时检测，但是它没有时间依赖性——不必去等待那些导致死锁出现的罕见条件。同时，设计过程需要去拆分应用，互斥量在这样的情况下可以消除可能导致死锁的可能性。这样的设计练习很有必要去做一下，即使你之后没有去做，代码也会在运行时进行检查。

超越锁的延伸扩展

如我在本节开头提到的那样，死锁不仅仅会发生在锁之间；死锁也会发生在任何同步构造中(可能会产生一个等待循环)，因此这方面也需要有指导意见，例如：要去避免获取嵌套锁等待一个持有锁的线程是一个很糟糕的决定，因为线程为了能继续运行可能需要获取对应的锁。类似的，如果去等待一个线程结束，它应该可以确定这个线程的层级，这样一个线程只需要等待比起层级低的线程结束即可。可以用一个简单的办法去确定，以添加的线程是否在同一函数中被启动，如同在3.1.2节和3.3节中描述的那样。

当代码已经能规避死锁， `std::lock()` 和 `std::lock_guard` 能组成简单的锁覆盖大多数情况，但是有时需要更多的灵活性。在这些情况，可以使用标准库提供的 `std::unique_lock` 模板。如 `std::lock_guard` ，这是一个参数化的互斥量模板类，并且它提供很多RAII类型锁用来管理 `std::lock_guard` 类型，可以让代码更加灵活。

## 3.2.6 std::unique_lock——灵活的锁

`std::unqiue_lock` 使用更为自由的不变量，这样 `std::unique_lock` 实例不会总与互斥量的数据类型相关，使用起来要比 `std:lock_guard` 更加灵活。首先，可将 `std::adopt_lock` 作为第二个参数传入构造函数，对互斥量进行管理；也可以将 `std::defer_lock` 作为第二个参数传递进去，表明互斥量应保持解锁状态。这样，就可以被 `std::unique_lock` 对象(不是互斥量)的lock()函数的所获取，或传递 `std::unique_lock` 对象到 `std::lock()` 中。清单3.6可以轻易的转换为清单3.9，使用 `std::unique_lock` 和 `std::defer_lock` ①，而非 `std::lock_guard` 和 `std::adopt_lock` 。代码长度相同，几乎等价，唯一不同的就是： `std::unique_lock` 会占用比较多的空间，并且比 `std::lock_guard` 稍慢一些。保证灵活性要付出代价，这个代价就是允许 `std::unique_lock` 实例不带互斥量：信息已被存储，且已被更新。

清单3.9 交换操作中 `std::lock()` 和 `std::unique_lock` 的使用

```
class some_big_object;
void swap(some_big_object& lhs,some_big_object& rhs);
class X
{
private:
  some_big_object some_detail;
  std::mutex m;
public:
  X(some_big_object const& sd):some_detail(sd){}
  friend void swap(X& lhs, X& rhs)
```

```
  {
    if(&lhs==&rhs)
      return;
    std::unique_lock<std::mutex> lock_a(lhs.m,std::defer_lock); // 1
    std::unique_lock<std::mutex> lock_b(rhs.m,std::defer_lock); // 1 std::def_l
    std::lock(lock_a,lock_b); // 2 互斥量在这里上锁
    swap(lhs.some_detail,rhs.some_detail);
  }
};
```

列表3.9中，因为 `std::unique_lock` 支持lock(), try_lock()和unlock()成员函数，所以能将 `std::unique_lock` 对象传递到 `std::lock()` ②。这些同名的成员函数在低层做着实际的工作，并且仅更新 `std::unique_lock` 实例中的标志，来确定该实例是否拥有特定的互斥量，这个标志是为了确保unlock()在析构函数中被正确调用。如果实例拥有互斥量，那么析构函数必须调用unlock()；但当实例中没有互斥量时，析构函数就不能去调用unlock()。这个标志可以通过owns_lock()成员变量进行查询。

可能如你期望的那样，这个标志被存储在某个地方。因此， `std::unique_lock` 对象的体积通常要比 `std::lock_guard` 对象大，当使用 `std::unique_lock` 替代 `std::lock_guard` ，因为会对标志进行适当的更新或检查，就会做些轻微的性能惩罚。当 `std::lock_guard` 已经能够满足你的需求，那么还是建议你继续使用它。当需要更加灵活的锁时，最好选择 `std::unique_lock` ，因为它更适合于你的任务。你已经看到一个递延锁的例子，另外一种情况是锁的所有权需要从一个域转到另一个域。

## 3.2.7 不同域中互斥量所有权的传递

`std::unique_lock` 实例没有与自身相关的互斥量，一个互斥量的所有权可以通过移动操作，在不同的实例中进行传递。某些情况下，这种转移是自动发生的，例如:当函数返回一个实例；另些情况下，需要显式的调用 `std::move()` 来执行移动操作。从本质上来说，需要依赖于源值是否是左值——一个实际的值或是引用——或一个右值——一个临时类型。当源值是一个右值，为了避免转移所有权过程出错，就必须显式移动成左值。 `std::unique_lock` 是可移动，但不可赋值的类型。附录A，A.1.1节有更多与移动语句相关的信息。

一种使用可能是允许一个函数去锁住一个互斥量，并且将所有权移到调用者上，所以调用者可以在这个锁保护的范围内执行额外的动作。

下面的程序片段展示了：函数get_lock()锁住了互斥量，然后准备数据，返回锁的调用函数：

```
std::unique_lock<std::mutex> get_lock()
{
  extern std::mutex some_mutex;
  std::unique_lock<std::mutex> lk(some_mutex);
  prepare_data();
  return lk;  // 1
}
void process_data()
{
  std::unique_lock<std::mutex> lk(get_lock());  // 2
  do_something();
}
```

lk在函数中被声明为自动变量，它不需要调用 `std::move()` ，可以直接返回①(编译器负责调用移动构造函数)。process_data()函数直接转移 `std::unique_lock` 实例的所有权②，调用do_something()可使用的正确数据(数据没有受到其他线程的修改)。

通常这种模式会用于已锁的互斥量，其依赖于当前程序的状态，或依赖于传入返回类型为 `std::unique_lock` 的函数(或以参数返回)。这样的用法不会直接返回锁，不过网关类的一个数据成员可用来确认已经对保护数据的访问权限进行上锁。这种情况下，所有的访问都必须通过网关类：当你想要访问数据，需要获取网关类的实例(如同前面的例子，通过调用get_lock()之类函数)来获取锁。之后你就可以通过网关类的成员函数对数据进行访问。当完成访问，可以销毁这个网关类对象，将锁进行释放，让别的线程来访问保护数据。这样的一个网关类可能是可移动的(所以他可以从一个函数进行返回)，在这种情况下锁对象的数据必须是可移动的。

`std::unique_lock` 的灵活性同样也允许实例在销毁之前放弃其拥有的锁。可以使用unlock()来做这件事，如同一个互斥量： `std::unique_lock` 的成员函数提供类似于锁定和解锁互斥量的功能。 `std::unique_lock` 实例在销毁前释放锁的能力，当锁没有必要在持有的时候，可以在特定的代码分支对其进行选择性的释放。这对于应用性能来说很重要，因为持有锁的时间增加会导致性能下降，其他线程会等待这个锁的释放，避免超越操作。

## 3.2.8 锁的粒度

3.2.3节中，已经对锁的粒度有所了解：锁的粒度是一个*摆手术语*(hand-waving term)，用来描述通过一个锁保护着的数据量大小。*一个细粒度锁*(a fine-grained lock)能够保护较小的数据量，*一个粗粒度锁*(a coarse-grained lock)能够保护较多的数据量。选择粒度对于锁来说很重要，为了保护对应的数据，保证锁有能力保护这些数据也很重要。我们都知道，在超市等待结账的时候，正在结账的顾客突然意识到他忘了拿蔓越莓酱，然后离开柜台去拿，并让其他的人都等待他回来；或

者当收银员，准备收钱时，顾客才去翻钱包拿钱，这样的情况都会让等待的顾客很无奈。当每个人都检查了自己要拿的东西，且能随时为拿到的商品进行支付，那么的每件事都会进行的很顺利。

这样的道理同样适用于线程：如果很多线程正在等待同一个资源(等待收银员对自己拿到的商品进行清点)，当有线程持有锁的时间过长，这就会增加等待的时间(别等到结账的时候，才想起来蔓越莓酱没拿)。在可能的情况下，锁住互斥量的同时只能对共享数据进行访问；试图对锁外数据进行处理。特别是做一些费时的动作，比如：对文件的输入/输出操作进行上锁。文件输入/输出通常要比从内存中读或写同样长度的数据慢成百上千倍，所以除非锁已经打算去保护对文件的访问，要么执行输入/输出操作将会将延迟其他线程执行的时间，这很没有必要(因为文件锁阻塞住了很多操作)，这样多线程带来的性能效益会被抵消。

`std::unique_lock` 在这种情况下工作正常，在调用unlock()时，代码不需要再访问共享数据；而后当再次需要对共享数据进行访问时，就可以再调用lock()了。下面代码就是这样的一种情况：

```
void get_and_process_data()
{
  std::unique_lock<std::mutex> my_lock(the_mutex);
  some_class data_to_process=get_next_data_chunk();
  my_lock.unlock();  // 1 不要让锁住的互斥量越过process()函数的调用
  result_type result=process(data_to_process);
  my_lock.lock(); // 2 为了写入数据，对互斥量再次上锁
  write_result(data_to_process,result);
}
```

不需要让锁住的互斥量越过对process()函数的调用，所以可以在函数调用①前对互斥量手动解锁，并且在之后对其再次上锁②。

这能表示只有一个互斥量保护整个数据结构时的情况，不仅可能会有更多对锁的竞争，也会增加锁持锁的时间。较多的操作步骤需要获取同一个互斥量上的锁，所以持有锁的时间会更长。成本上的双重打击也算是为向细粒度锁转移提供了双重激励和可能。

如同上面的例子，锁不仅是能锁住合适粒度的数据，还要控制锁的持有时间，以及什么操作在执行的同时能够拥有锁。一般情况下，执行必要的操作时，尽可能将持有锁的时间缩减到最小。这也就意味有一些浪费时间的操作，比如：获取另外一个锁(即使你知道这不会造成死锁)，或等待输入/输出操作完成时没有必要持有一个锁(除非绝对需要)。

清单3.6和3.9中，交换操作需要锁住两个互斥量，其明确要求并发访问两个对象。假设用来做比较的是一个简单的数据类型(比如:int类型)，将会有什么不同么？int的拷贝很廉价，所以可以很容易的进行数据复制，并且每个被比较的对象都持有该对象的锁，在比较之后进行数据拷贝。这就意

味着，在最短时间内持有每个互斥量，并且你不会在持有一个锁的同时再去获取另一个。下面的清单中展示了一个在这样情景中的Y类，并且展示了一个相等比较运算符的等价实现。

列表3.10 比较操作符中一次锁住一个互斥量

```
class Y
{
private:
  int some_detail;
  mutable std::mutex m;
  int get_detail() const
  {
    std::lock_guard<std::mutex> lock_a(m);  // 1
    return some_detail;
  }
public:
  Y(int sd):some_detail(sd){}

  friend bool operator==(Y const& lhs, Y const& rhs)
  {
    if(&lhs==&rhs)
      return true;
    int const lhs_value=lhs.get_detail();  // 2
    int const rhs_value=rhs.get_detail();  // 3
    return lhs_value==rhs_value;  // 4
  }
};
```

在这个例子中，比较操作符首先通过调用get_detail()成员函数检索要比较的值②③，函数在索引值时被一个锁保护着①。比较操作符会在之后比较索引出来的值④。注意：虽然这样能减少锁持有的时间，一个锁只持有一次(这样能消除死锁的可能性)，这里有一个微妙的语义操作同时对两个锁住的值进行比较。

列表3.10中，当操作符返回true时，那就意味着在这个时间点上的lhs.some_detail与在另一个时间点的rhs.some_detail相同。这两个值在读取之后，可能会被任意的方式所修改；两个值会在②和③处进行交换，这样就会失去比较的意义。等价比较可能会返回true，来表明这两个值时相等的，实际上这两个值相等的情况可能就发生在一瞬间。这样的变化要小心，语义操作是无法改变一个问题的比较方式：当你持有锁的时间没有达到整个操作时间，就会让自己处于条件竞争的状态。

有时，只是没有一个合适粒度级别，因为并不是所有对数据结构的访问都需要同一级的保护。这个例子中，就需要寻找一个合适的机制，去替换 `std::mutex` 。

[1] Tom Cargill, “Exception Handling: A False Sense of Security,” in C++ Report 6, no. 9 (November–December 1994). Also available at http://www.informit.com/content/images/020163371x/supplements/Exception_Handling_Article.html.

[2] Herb Sutter, Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions (Addison Wesley Pro-fessional, 1999).