4.1 等待一个事件或其他条件

假设你在旅游，而且正在一辆在夜间运行的火车上。在夜间，如何在正确的站点下车呢？一种方法是整晚都要醒着，然后注意到了哪一站。这样，你就不会错过你要到达的站点，但是这样会让你感到很疲倦。另外，你可以看一下时间表，估计一下火车到达目的地的时间，然后在一个稍早的时间点上设置闹铃，然后你就可以安心的睡会了。这个方法听起来也很不错，也没有错过你要下车的站点，但是当火车晚点的时候，你就要被过早的叫醒了。当然，闹钟的电池也可能会没电了，并导致你睡过站。理想的方式是，无论是早或晚，只要当火车到站的时候，有人或其他东西能把你唤醒，就好了。

这和线程有什么关系呢？好吧，让我们来联系一下。当一个线程等待另一个线程完成任务时，它会有很多选择。第一，它可以持续的检查共享数据标志(用于做保护工作的互斥量)，直到另一线程完成工作时对这个标志进行重设。不过，就是一种浪费：线程消耗宝贵的执行时间持续的检查对应标志，并且当互斥量被等待线程上锁后，其他线程就没有办法获取锁，这样线程就会持续等待。因为以上方式对等待线程限制资源，并且在完成时阻碍对标识的设置。这种情况类似与，保持清醒状态和列车驾驶员聊了一晚上：驾驶员不得不缓慢驾驶，因为你分散了他的注意力，所以火车需要更长的时间，才能到站。同样的，等待的线程会等待更长的时间，这些线程也在消耗着系统资源。

第二个选择是在等待线程在检查间隙，使用 `std::this_thread::sleep_for()` 进行周期性的间歇(详见4.3节)：

```
bool flag;
std::mutex m;

void wait_for_flag()
{
  std::unique_lock<std::mutex> lk(m);
  while(!flag)
  {
    lk.unlock();  // 1 解锁互斥量
    std::this_thread::sleep_for(std::chrono::milliseconds(100));  // 2 休眠100m
    lk.lock();   // 3 再锁互斥量
  }
}
```

这个循环中，在休眠前②，函数对互斥量进行解锁①，并且在休眠结束后再对互斥量进行上锁，所以另外的线程就有机会获取锁并设置标识。

这个实现就进步很多，因为当线程休眠时，线程没有浪费执行时间，但是很难确定正确的休眠时间。太短的休眠和没有休眠一样，都会浪费执行时间；太长的休眠时间，可能会让任务等待线程醒来。休眠时间过长是很少见的情况，因为这会直接影响到程序的行为，当在高节奏游戏中，它意味着丢帧，或在一个实时应用中超越了一个时间片。

第三个选择(也是优先的选择)是，使用C++标准库提供的工具去等待事件的发生。通过另一线程触发等待事件的机制是最基本的唤醒方式(例如：流水线上存在额外的任务时)，这种机制就称为“条件变量”。从概念上来说，一个条件变量会与多个事件或其他条件相关，并且一个或多个线程会等待条件的达成。当某些线程被终止时，为了唤醒等待线程(允许等待线程继续执行)终止的线程将会向等待着的线程广播“条件达成”的信息。

4.1.1 等待条件达成

C++标准库对条件变量有两套实现： `std::condition_variable` 和 `std::condition_variable_any` 。这两个实现都包含在 `<condition_variable>` 头文件的声明中。两者都需要与一个互斥量一起才能工作(互斥量是为了同步)；前者仅限于与 `std::mutex` 一起工作，而后者可以和任何满足最低标准的互斥量一起工作，从而加上了*_any*的后缀。因为 `std::condition_variable_any` 更加通用，这就可能从体积、性能，以及系统资源的使用方面产生额外的开销，所以 `std::condition_variable` 一般作为首选的类型，当对灵活性有硬性要求时，我们才会去考虑 `std::condition_variable_any` 。

所以，如何使用 `std::condition_variable` 去处理之前提到的情况——当有数据需要处理时，如何唤醒休眠中的线程对其进行处理？以下清单展示了一种使用条件变量做唤醒的方式。

清单4.1 使用 `std::condition_variable` 处理数据等待

```
std::mutex mut;
std::queue<data_chunk> data_queue;  // 1
std::condition_variable data_cond;

void data_preparation_thread()
{
  while(more_data_to_prepare())
  {
    data_chunk const data=prepare_data();
```

```
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);  // 2
    data_cond.notify_one();  // 3
  }
}

void data_processing_thread()
{
  while(true)
  {
    std::unique_lock<std::mutex> lk(mut);  // 4
    data_cond.wait(
         lk,[]{return !data_queue.empty();});  // 5
    data_chunk data=data_queue.front();
    data_queue.pop();
    lk.unlock();  // 6
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

首先，你拥有一个用来在两个线程之间传递数据的队列①。当数据准备好时，使用 `std::lock_guard` 对队列上锁，将准备好的数据压入队列中②，之后线程会对队列中的数据上锁。然后调用 `std::condition_variable` 的notify_one()成员函数，对等待的线程(如果有等待线程)进行通知③。

在另外一侧，你有一个正在处理数据的线程，这个线程首先对互斥量上锁，但在这里 `std::unique_lock` 要比 `std::lock_guard` ④更加合适——且听我细细道来。线程之后会调用 `std::condition_variable` 的成员函数wait()，传递一个锁和一个lambda函数表达式(作为等待的条件⑤)。Lambda函数是 `C++11` 添加的新特性，它可以让一个匿名函数作为其他表达式的一部分，并且非常合适作为标准函数的谓词，例如wait()函数。在这个例子中，简单的lambda函数 `[]{return !data_queue.empty();}` 会去检查data_queue是否不为空，当data_queue不为空——那就意味着队列中已经准备好数据了。附录A的A.5节有Lambda函数更多的信息。

wait()会去检查这些条件(通过调用所提供的lambda函数)，当条件满足(lambda函数返回true)时返回。如果条件不满足(lambda函数返回false)，wait()函数将解锁互斥量，并且将这个线程(上段提到的处理数据的线程)置于阻塞或等待状态。当准备数据的线程调用notify_one()通知条件变量时，处理数据的线程从睡眠状态中苏醒，重新获取互斥锁，并且对条件再次检查，在条件满足的情况下，从wait()返回并继续持有锁。当条件不满足时，线程将对互斥量解锁，并且重新开始等待。这就是为什么用 `std::unique_lock` 而不使用 `std::lock_guard` ——等待中的线程必须在等待期间

解锁互斥量，并在这这之后对互斥量再次上锁，而 `std::lock_guard` 没有这么灵活。如果互斥量在线程休眠期间保持锁住状态，准备数据的线程将无法锁住互斥量，也无法添加数据到队列中；同样的，等待线程也永远不会知道条件何时满足。

清单4.1使用了一个简单的lambda函数用于等待⑤，这个函数用于检查队列何时不为空，不过任意的函数和可调用对象都可以传入wait()。当你已经写好了一个函数去做检查条件(或许比清单中简单检查要复杂很多)，那就可以直接将这个函数传入wait()；不一定非要放在一个lambda表达式中。在调用wait()的过程中，一个条件变量可能会去检查给定条件若干次；然而，它总是在互斥量被锁定时这样做，当且仅当提供测试条件的函数返回true时，它就会立即返回。当等待线程重新获取互斥量并检查条件时，如果它并非直接响应另一个线程的通知，这就是所谓的*伪唤醒*(spurious wakeup)。因为任何伪唤醒的数量和频率都是不确定的，这里不建议使用一个有副作用的函数做条件检查。当你这样做了，就必须做好多次产生副作用的心理准备。

解锁 `std::unique_lock` 的灵活性，不仅适用于对wait()的调用；它还可以用于有待处理但还未处理的数据⑥。处理数据可能是一个耗时的操作，并且如你在第3章见到的，你就知道持有锁的时间过长是一个多么糟糕的主意。

使用队列在多个线程中转移数据(如清单4.1)是很常见的。做得好的话，同步操作可以限制在队列本身，同步问题和条件竞争出现的概率也会降低。鉴于这些好处，现在从清单4.1中提取出一个通用线程安全的队列。

4.1.2 使用条件变量构建线程安全队列

当你正在设计一个通用队列时，花一些时间想想有哪些操作需要添加到队列实现中去，就如之前在3.2.3节看到的线程安全的栈。可以看一下C++标准库提供的实现，找找灵感； `std::queue<>` 容器的接口展示如下：

清单4.2 `std::queue` 接口

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

```
10
11    void swap(queue& q);
12
13    bool empty() const;
14    size_type size() const;
15
16    T& front();
17    const T& front() const;
18    T& back();
19    const T& back() const;
20
21    void push(const T& x);
22    void push(T&& x);
23    void pop();
24    template <class... Args> void emplace(Args&&... args);
25  };
```

当你忽略构造、赋值以及交换操作时，你就剩下了三组操作：1. 对整个队列的状态进行查询(empty()和size());2.查询在队列中的各个元素(front()和back())；3.修改队列的操作(push(), pop()和emplace())。这就和3.2.3中的栈一样了，因此你也会遇到在固有接口上的条件竞争。因此，你需要将front()和pop()合并成一个函数调用，就像之前在栈实现时合并top()和pop()一样。与清单4.1中的代码不同的是：当使用队列在多个线程中传递数据时，接收线程通常需要等待数据的压入。这里我们提供pop()函数的两个变种：try_pop()和wait_and_pop()。try_pop() ，尝试从队列中弹出数据，总会直接返回(当有失败时)，即使没有值可检索；wait_and_pop()，将会等待有值可检索的时候才返回。当你使用之前栈的方式来实现你的队列，你实现的队列接口就可能会是下面这样：

清单4.3 线程安全队列的接口

```
1  #include <memory> // 为了使用std::shared_ptr
2
3  template<typename T>
4  class threadsafe_queue
5  {
6  public:
7    threadsafe_queue();
8    threadsafe_queue(const threadsafe_queue&);
9    threadsafe_queue& operator=(
10       const threadsafe_queue&) = delete;  // 不允许简单的赋值
11
12   void push(T new_value);
13
14   bool try_pop(T& value);  // 1
```

```
  std::shared_ptr<T> try_pop();  // 2

  void wait_and_pop(T& value);
  std::shared_ptr<T> wait_and_pop();

  bool empty() const;
};
```

就像之前对栈做的那样，在这里你将很多构造函数剪掉了，并且禁止了对队列的简单赋值。和之前一样，你也需要提供两个版本的try_pop()和wait_for_pop()。第一个重载的try_pop()①在引用变量中存储着检索值，所以它可以用来返回队列中值的状态；当检索到一个变量时，他将返回true，否则将返回false(详见A.2节)。第二个重载②就不能做这样了，因为它是用来直接返回检索值的。当没有值可检索时，这个函数可以返回NULL指针。

那么问题来了，如何将以上这些和清单4.1中的代码相关联呢？好吧，我们现在就来看看怎么去关联。你可以从之前的代码中提取push()和wait_and_pop()，如以下清单所示。

清单4.4 从清单4.1中提取push()和wait_and_pop()

```
#include <queue>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
{
private:
  std::mutex mut;
  std::queue<T> data_queue;
  std::condition_variable data_cond;
public:
  void push(T new_value)
  {
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(new_value);
    data_cond.notify_one();
  }

  void wait_and_pop(T& value)
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    value=data_queue.front();
```

```
    data_queue.pop();
  }
};
threadsafe_queue<data_chunk> data_queue;  // 1

void data_preparation_thread()
{
  while(more_data_to_prepare())
  {
    data_chunk const data=prepare_data();
    data_queue.push(data);  // 2
  }
}

void data_processing_thread()
{
  while(true)
  {
    data_chunk data;
    data_queue.wait_and_pop(data);  // 3
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

线程队列的实例中包含有互斥量和条件变量，所以独立的变量就不需要了①，并且调用push()也不需要外部同步②。当然，wait_and_pop()还要兼顾条件变量的等待③。

另一个wait_and_pop()函数的重载写起来就很琐碎了，剩下的函数就像从清单3.5实现的栈中一个个的粘过来一样。最终的队列实现如下所示。

清单4.5 使用条件变量的线程安全队列(完整版)

```
#include <queue>
#include <memory>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
{
private:
```

```
  mutable std::mutex mut;  // 1 互斥量必须是可变的
  std::queue<T> data_queue;
  std::condition_variable data_cond;
public:
  threadsafe_queue()
  {}
  threadsafe_queue(threadsafe_queue const& other)
  {
    std::lock_guard<std::mutex> lk(other.mut);
    data_queue=other.data_queue;
  }

  void push(T new_value)
  {
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(new_value);
    data_cond.notify_one();
  }

  void wait_and_pop(T& value)
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    value=data_queue.front();
    data_queue.pop();
  }

  std::shared_ptr<T> wait_and_pop()
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
  }

  bool try_pop(T& value)
  {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
      return false;
    value=data_queue.front();
    data_queue.pop();
    return true;
  }
```

```
55
56    std::shared_ptr<T> try_pop()
57    {
58      std::lock_guard<std::mutex> lk(mut);
59      if(data_queue.empty())
60        return std::shared_ptr<T>();
61      std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
62      data_queue.pop();
63      return res;
64    }
65
66    bool empty() const
67    {
68      std::lock_guard<std::mutex> lk(mut);
69      return data_queue.empty();
70    }
71  };
```

empty()是一个const成员函数，并且传入拷贝构造函数的other形参是一个const引用；因为其他线程可能有这个类型的非const引用对象，并调用变种成员函数，所以这里有必要对互斥量上锁。如果锁住互斥量是一个可变操作，那么这个互斥量对象就会标记为可变的①，之后他就可以在empty()和拷贝构造函数中上锁了。

条件变量在多个线程等待同一个事件时，也是很有用的。当线程用来分解工作负载，并且只有一个线程可以对通知做出反应，与清单4.1中使用的结构完全相同；运行多个数据实例——*处理线程*(processing thread)。当新的数据准备完成，调用notify_one()将会触发一个正在执行wait()的线程，去检查条件和wait()函数的返回状态(因为你仅是向data_queue添加一个数据项)。 这里不保证线程一定会被通知到，即使只有一个等待线程被通知时，所有处线程也有可能都在处理数据。

另一种可能是，很多线程等待同一事件，对于通知他们都需要做出回应。这会发生在共享数据正在初始化的时候，当处理线程可以使用同一数据时，就要等待数据被初始化(有不错的机制可用来应对；可见第3章，3.3.1节)，或等待共享数据的更新，比如，*定期重新初始化*(periodic reinitialization)。在这些情况下，准备线程准备数据数据时，就会通过条件变量调用notify_all()成员函数，而非直接调用notify_one()函数。顾名思义，这就是全部线程在都去执行wait()(检查他们等待的条件是否满足)的原因。

当等待线程只等待一次，当条件为true时，它就不会再等待条件变量了，所以一个条件变量可能并非同步机制的最好选择。尤其是，条件在等待一组可用的数据块时。在这样的情况下，*期望*(future)就是一个适合的选择。