9.1 线程池

很多公司里，雇员通常会在办公室度过他们的办公时光(偶尔也会外出访问客户或供应商)，或是参加贸易展会。虽然外出可能很有必要，并且可能需要很多人一起去，不过对于一些特别的雇员来说，一趟可能就是几个月，甚至是几年。公司要给每个雇员都配一辆车，这基本上是不可能的，不过公司可以提供一些共用车辆；这样就会有一定数量车，来让所有雇员使用。当一个员工要去异地旅游时，那么他就可以从共用车辆中预定一辆，并在返回公司的时候将车交还。如果某天没有闲置的共用车辆，雇员就得不延后其旅程了。

线程池就是类似的一种方式，在大多数系统中，将每个任务指定给某个线程是不切实际的，不过可以利用现有的并发性，进行并发执行。线程池就提供了这样的功能，提交到线程池中的任务将并发执行，提交的任务将会挂在任务队列上。队列中的每一个任务都会被池中的工作线程所获取，当任务执行完成后，再回到线程池中获取下一个任务。

创建一个线程池时，会遇到几个关键性的设计问题，比如：可使用的线程数量，高效的任务分配方式，以及是否需要等待一个任务完成。

在本节，我们将看到线程池是如何解决这些问题的，从最简单的线程池开始吧！

9.1.1 最简单的线程池

作为最简单的线程池，其拥有固定数量的工作线程(通常工作线程数量与 `std::thread::hardware_concurrency()` 相同)。当工作需要完成时，可以调用函数将任务挂在任务队列中。每个工作线程都会从任务队列上获取任务，然后执行这个任务，执行完成后再回来获取新的任务。在最简单的线程池中，线程就不需要等待其他线程完成对应任务了。如果需要等待，就需要对同步进行管理。

下面清单中的代码就展示了一个最简单的线程池实现。

清单9.1 简单的线程池

```
1  class thread_pool
2  {
3    std::atomic_bool done;
```

```
  thread_safe_queue<std::function<void()> > work_queue;  // 1
  std::vector<std::thread> threads;  // 2
  join_threads joiner;  // 3

  void worker_thread()
  {
    while(!done)  // 4
    {
      std::function<void()> task;
      if(work_queue.try_pop(task))  // 5
      {
        task();  // 6
      }
      else
      {
        std::this_thread::yield();  // 7
      }
    }
  }

public:
  thread_pool():
    done(false),joiner(threads)
  {
    unsigned const thread_count=std::thread::hardware_concurrency();  // 8

    try
    {
      for(unsigned i=0;i<thread_count;++i)
      {
        threads.push_back(
          std::thread(&thread_pool::worker_thread,this));  // 9
      }
    }
    catch(...)
    {
      done=true;  // 10
      throw;
    }
  }

  ~thread_pool()
  {
    done=true;  // 11
  }
```

```
49
50    template<typename FunctionType>
51    void submit(FunctionType f)
52    {
53      work_queue.push(std::function<void()>(f));  // 12
54    }
55  };
```

实现中有一组工作线程②，并且使用了一个线程安全队列(见第6章)①来管理任务队列。这种情况下，用户不用等待任务，并且任务不需要返回任何值，所以可以使用 `std::function<void()>` 对任务进行封装。submit()函数会将函数或可调用对象包装成一个 `std::function<void()>` 实例，并将其推入队列中⑫。

线程始于构造函数：使用 `std::thread::hardware_concurrency()` 来获取硬件支持多少个并发线程⑧，这些线程会在worker_thread()成员函数中执行⑨。

当有异常抛出时，线程启动就会失败，所以需要保证任何已启动的线程都能停止，并且能在这种情况下清理干净。当有异常抛出时，通过使用*try-catch*来设置done标志⑩，还有join_threads类的实例(来自于第8章)③用来汇聚所有线程。当然也需要析构函数：仅设置done标志⑪，并且join_threads确保所有线程在线程池销毁前全部执行完成。注意成员声明的顺序很重要：done标志和worker_queue必须在threads数组之前声明，而数据必须在joiner前声明。这就能确保成员能以正确的顺序销毁；比如，所有线程都停止运行时，队列就可以安全的销毁了。

worker_thread函数很简单：从任务队列上获取任务⑤，以及同时执行这些任务⑥，执行一个循环直到done标志被设置④。如果任务队列上没有任务，函数会调用 `std::this_thread::yield()` 让线程休息⑦，并且给予其他线程向任务队列上推送任务的机会。

一些简单的情况，这样线程池就足以满足要求，特别是任务没有返回值，或需要执行一些阻塞操作的时候。不过，在很多情况下，这样简单的线程池完全不够用，其他情况使用这样简单的线程池可能会出现问题，比如：死锁。同样，在简单例子中，使用 `std::async` 能提供更好的功能(如第8章中的例子)。

在本章中，我们将了解一下更加复杂的线程池实现，通过添加特性满足用户需求，或减少问题的发生几率。

首先，从已经提交的任务开始说起。

9.1.2 等待提交到线程池中的任务

第8章中的例子中，线程间的任务划分完成后，代码会显式生成新线程，主线程通常就是等待新线程在返回调用之前结束，确保所有任务都完成。使用线程池，就需要等待任务提交到线程池中，而非直接提交给单个线程。这与基于 `std::async` 的方法(第8章等待future的例子)类似，使用清单9.1中的简单线程池，使用第4章中提到的工具：条件变量和future。虽然，会增加代码的复杂度，不过，要比直接对任务进行等待的方式好很多。

通过增加线程池的复杂度，可以直接等待任务完成。使用submit()函数返回一个对任务描述的句柄，用来等待任务的完成。任务句柄会用条件变量或future进行包装，这样能使用线程池来简化代码。

一种特殊的情况是，执行任务的线程需要返回一个结果到主线程上进行处理。你已经在本书中看到多个这样的例子，比如：parallel_accumulate()(第2章)。这种情况下，需要用future对最终的结果进行转移。清单9.2展示了对简单线程池的修改，通过修改就能等待任务完成，以及在工作线程完成后，返回一个结果到等待线程中去，不过 `std::packaged_task<>` 实例是不可拷贝的，仅是可移动的，所以不能再使用 `std::function<>` 来实现任务队列，因为 `std::function<>` 需要存储可复制构造的函数对象。包装一个自定义函数，用来处理只可移动的类型。这就是一个带有函数操作符的类型擦除类。只需要处理那些没有函数和无返回的函数，所以这是一个简单的虚函数调用。

清单9.2 可等待任务的线程池

```
class function_wrapper
{
  struct impl_base {
    virtual void call()=0;
    virtual ~impl_base() {}
  };

  std::unique_ptr<impl_base> impl;
  template<typename F>
  struct impl_type: impl_base
  {
    F f;
    impl_type(F&& f_): f(std::move(f_)) {}
    void call() { f(); }
  };
public:
  template<typename F>
  function_wrapper(F&& f):
    impl(new impl_type<F>(std::move(f)))
  {}

```

```
  void operator()() { impl->call(); }

  function_wrapper() = default;

  function_wrapper(function_wrapper&& other):
    impl(std::move(other.impl))
  {}

  function_wrapper& operator=(function_wrapper&& other)
  {
    impl=std::move(other.impl);
    return *this;
  }

  function_wrapper(const function_wrapper&)=delete;
  function_wrapper(function_wrapper&)=delete;
  function_wrapper& operator=(const function_wrapper&)=delete;
};

class thread_pool
{
  thread_safe_queue<function_wrapper> work_queue;  // 使用function_wrapper，而非

  void worker_thread()
  {
    while(!done)
    {
      function_wrapper task;
      if(work_queue.try_pop(task))
      {
        task();
      }
      else
      {
        std::this_thread::yield();
      }
    }
  }
public:
  template<typename FunctionType>
  std::future<typename std::result_of<FunctionType()>::type>  // 1
    submit(FunctionType f)
  {
    typedef typename std::result_of<FunctionType()>::type
      result_type;  // 2
```

```
67
68      std::packaged_task<result_type()> task(std::move(f));  // 3
69      std::future<result_type> res(task.get_future());  // 4
70      work_queue.push(std::move(task));  // 5
71      return res;  // 6
72    }
73    // 休息一下
74  };
```

首先，修改的是submit()函数①返回一个 `std::future<>` 保存任务的返回值，并且允许调用者等待任务完全结束。因为需要知道提供函数f的返回类型，所以使用 `std::result_of<>` : `std::result_of<FunctionType()>::type` 是FunctionType类型的引用实例(如，f)，并且没有参数。同样，函数中可以对result_type typedef②使用 `std::result_of<>` 。

然后，将f包装入 `std::packaged_task<result_type()>` ③，因为f是一个无参数的函数或是可调用对象，能够返回result_type类型的实例。向任务队列推送任务⑤和返回future⑥前，就可以从 `std::packaged_task<>` 中获取future④。注意，要将任务推送到任务队列中时，只能使用 `std::move()` ，因为 `std::packaged_task<>` 是不可拷贝的。为了对任务进行处理，队列里面存的就是function_wrapper对象，而非 `std::function<void()>` 对象。

现在线程池允许等待任务，并且返回任务后的结果。下面的清单就展示了，如何让parallel_accumuate函数使用线程池。

清单9.3 parallel_accumulate使用一个可等待任务的线程池

```
1  template<typename Iterator,typename T>
2  T parallel_accumulate(Iterator first,Iterator last,T init)
3  {
4    unsigned long const length=std::distance(first,last);
5
6    if(!length)
7      return init;
8
9    unsigned long const block_size=25;
10   unsigned long const num_blocks=(length+block_size-1)/block_size;  // 1
11
12   std::vector<std::future<T> > futures(num_blocks-1);
13   thread_pool pool;
14
15   Iterator block_start=first;
16   for(unsigned long i=0;i<(num_blocks-1);++i)
```

```
    {
      Iterator block_end=block_start;
      std::advance(block_end,block_size);
      futures[i]=pool.submit(accumulate_block<Iterator,T>());  // 2
      block_start=block_end;
    }
    T last_result=accumulate_block<Iterator,T>()(block_start,last);
    T result=init;
    for(unsigned long i=0;i<(num_blocks-1);++i)
    {
      result+=futures[i].get();
    }
    result += last_result;
    return result;
}
```

与清单8.4相比，有几个点需要注意一下。首先，工作量是依据使用的块数(num_blocks①)，而不是线程的数量。为了利用线程池的最大化可扩展性，需要将工作块划分为最小工作块。当线程池中线程不多时，每个线程将会处理多个工作块，不过随着硬件可用线程数量的增长，会有越来越多的工作块并发执行。

当你选择“因为能并发执行，最小工作块值的一试”时，就需要谨慎了。向线程池提交一个任务有一定的开销；让工作线程执行这个任务，并且将返回值保存在 `std::future<>` 中，对于太小的任务，这样的开销不划算。如果任务块太小，使用线程池的速度可能都不及单线程。

假设，任务块的大小合理，就不用为这些事而担心：打包任务、获取future或存储之后要汇入的 `std::thread` 对象；使用线程池的时候，这些都需要注意。之后，就是调用submit()来提交任务②。

线程池也需要注意异常安全。任何异常都会通过submit()返回给future，并在获取future的结果时，抛出异常。如果函数因为异常退出，线程池的析构函数会丢掉那些没有完成的任务，等待线程池中的工作线程完成工作。

在简单的例子中，这个线程池工作的还算不错，因为这里的任务都是相互独立的。不过，当任务队列中的任务有依赖关系时，这个线程池就不能胜任了。

9.1.3 等待依赖任务

快速排序算法为例，原理很简单：数据与中轴数据项比较，在中轴项两侧分为大于和小于的两个序列，然后再对这两组序列进行排序。这两组序列会递归排序，最后会整合成一个全排序序列。要将这个算法写成并发模式，需要保证递归调用能够使用硬件的并发能力。

回到第4章，第一次接触这个例子，我们使用 `std::async` 来执行每一层的调用，让标准库来选择，是在新线程上执行这个任务，还是当对应get()调用时，进行同步执行。运行起来很不错，因为每一个任务都在其自己的线程上执行，或当需要的时候进行调用。

当回顾第8章时，使用了一个固定线程数量(根据硬件可用并发线程数)的结构体。在这样的情况下，使用了栈来挂起要排序的数据块。当每个线程在为一个数据块排序前，会向数据栈上添加一组要排序的数据，然后对当前数据块排序结束后，接着对另一块进行排序。这里，等待其他线程完成排序，可能会造成死锁，因为这会消耗有限的线程。有一种情况很可能会出现，就是所有线程都在等某一个数据块被排序，不过没有线程在做排序。通过拉取栈上数据块的线程，对数据块进行排序，来解决这个问题；因为，已处理的指定数据块，就是其他线程都在等待排序的数据块。

如果只用简单的线程池进行替换，例如：第4章替换 `std::async` 的线程池。只有固定数量的线程，因为线程池中没有空闲的线程，线程会等待没有被安排的任务。因此，需要和第8章中类似的解决方案：当等待某个数据块完成时，去处理未完成的数据块。如果使用线程池来管理任务列表和相关线程——使用线程池的主要原因——就不用再去访问任务列表了。可以对线程池做一些改动，自动完成这些事情。

最简单的方法就是在thread_pool中添加一个新函数，来执行任务队列上的任务，并对线程池进行管理。高级线程池的实现可能会在等待函数中添加逻辑，或等待其他函数来处理这个任务，优先的任务会让其他的任务进行等待。下面清单中的实现，就展示了一个新run_pending_task()函数，对于快速排序的修改将会在清单9.5中展示。

清单9.4 run_pending_task()函数实现

```
void thread_pool::run_pending_task()
{
  function_wrapper task;
  if(work_queue.try_pop(task))
  {
    task();
  }
  else
  {
    std::this_thread::yield();
  }
}
```

run_pending_task()的实现去掉了在worker_thread()函数的主循环。函数任务队列中有任务的时候，执行任务；要是没有的话，就会让操作系统对线程进行重新分配。

下面快速排序算法的实现要比清单8.1中版本简单许多，因为所有线程管理逻辑都被移入到线程池。

清单9.5 基于线程池的快速排序实现

```
template<typename T>
struct sorter  // 1
{
  thread_pool pool;  // 2

  std::list<T> do_sort(std::list<T>& chunk_data)
  {
    if(chunk_data.empty())
    {
      return chunk_data;
    }

    std::list<T> result;
    result.splice(result.begin(),chunk_data,chunk_data.begin());
    T const& partition_val=*result.begin();

    typename std::list<T>::iterator divide_point=
      std::partition(chunk_data.begin(),chunk_data.end(),
                     [&](T const& val){return val<partition_val;});

    std::list<T> new_lower_chunk;
    new_lower_chunk.splice(new_lower_chunk.end(),
                           chunk_data,chunk_data.begin(),
                           divide_point);

    std::future<std::list<T> > new_lower=  // 3
      pool.submit(std::bind(&sorter::do_sort,this,
                            std::move(new_lower_chunk)));

    std::list<T> new_higher(do_sort(chunk_data));

    result.splice(result.end(),new_higher);
    while(!new_lower.wait_for(std::chrono::seconds(0)) ==
      std::future_status::timeout)
    {
      pool.run_pending_task();  // 4
```

```
    }

    result.splice(result.begin(),new_lower.get());
    return result;
  }
};

template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
  if(input.empty())
  {
    return input;
  }
  sorter<T> s;

  return s.do_sort(input);
}
```

与清单8.1相比，这里将实际工作放在sorter类模板的do_sort()成员函数中执行①，即使例子中仅对thread_pool实例进行包装②。

线程和任务管理，在线程等待的时候，就会少向线程池中提交一个任务③，并且执行任务队列上未完成的任务④。需要显式的管理线程和栈上要排序的数据块。当有任务提交到线程池中，可以使用 `std::bind()` 绑定this指针到do_sort()上，绑定是为了让数据块进行排序。这种情况下，需要对new_lower_chunk使用 `std::move()` 将其传入函数，数据移动要比拷贝的方式开销少。

虽然，使用等待其他任务的方式，解决了死锁问题，这个线程池距离理想的线程池很远。

首先，每次对submit()的调用和对run_pending_task()的调用，访问的都是同一个队列。在第8章中，当多线程去修改一组数据，就会对性能有所影响，所以需要解决这个问题。

9.1.4 避免队列中的任务竞争

线程每次调用线程池的submit()函数，都会推送一个任务到工作队列中。就像工作线程为了执行任务，从任务队列中获取任务一样。这意味着随着处理器的增加，在任务队列上就会有很多的竞争，这会让性能下降。使用无锁队列会让任务没有明显的等待，但是乒乓缓存会消耗大量的时间。

为了避免乒乓缓存，每个线程建立独立的任务队列。这样，每个线程就会将新任务放在自己的任务队列上，并且当线程上的任务队列没有任务时，去全局的任务列表中取任务。下面列表中的实现，使用了一个thread_local变量，来保证每个线程都拥有自己的任务列表(如全局列表那样)。

清单9.6 线程池——线程具有本地任务队列

```
class thread_pool
{
  thread_safe_queue<function_wrapper> pool_work_queue;

  typedef std::queue<function_wrapper> local_queue_type;  // 1
  static thread_local std::unique_ptr<local_queue_type>
    local_work_queue;  // 2

  void worker_thread()
  {
    local_work_queue.reset(new local_queue_type);  // 3
    while(!done)
    {
      run_pending_task();
    }
  }

public:
  template<typename FunctionType>
  std::future<typename std::result_of<FunctionType()>::type>
    submit(FunctionType f)
  {
    typedef typename std::result_of<FunctionType()>::type result_type;

    std::packaged_task<result_type()> task(f);
    std::future<result_type> res(task.get_future());
    if(local_work_queue)  // 4
    {
      local_work_queue->push(std::move(task));
    }
    else
    {
      pool_work_queue.push(std::move(task));  // 5
    }
    return res;
  }

  void run_pending_task()
```

```
  {
    function_wrapper task;
    if(local_work_queue && !local_work_queue->empty())  // 6
    {
      task=std::move(local_work_queue->front());
      local_work_queue->pop();
      task();
    }
    else if(pool_work_queue.try_pop(task))  // 7
    {
      task();
    }
    else
    {
      std::this_thread::yield();
    }
  }
// rest as before
};
```

因为不希望非线程池中的线程也拥有一个任务队列，使用 `std::unique_ptr<>` 指向线程本地的工作队列②；这个指针在worker_thread()中进行初始化③。 `std:unique_ptr<>` 的析构函数会保证在线程退出的时候，工作队列被销毁。

submit()会检查当前线程是否具有一个工作队列④。如果有，就是线程池中的线程，可以将任务放入线程的本地队列中；否者，就像之前一样将这个任务放在线程池中的全局队列中⑤。

run_pending_task()⑥中的检查和之前类似，只是要对是否存在本地任务队列进行检查。如果存在，就会从队列中的第一个任务开始处理；注意本地任务队列可以是一个普通的 `std::queue<>` ①，因为这个队列只能被一个线程所访问，就不存在竞争。如果本地线程上没有任务，就会从全局工作列表上获取任务⑦。

这样就能有效避免竞争，不过当任务分配不均时，造成的结果就是：某个线程本地队列中有很多任务的同时，其他线程无所事事。例如：举一个快速排序的例子，只有一开始的数据块能在线程池上被处理，因为剩余部分会放在工作线程的本地队列上进行处理，这样的使用方式也违背使用线程池的初衷。

幸好，这个问题是有解：本地工作队列和全局工作队列上没有任务时，可从别的线程队列中窃取任务。

9.1.5 窃取任务

为了让没有任务的线程能从其他线程的任务队列中获取任务，就需要本地任务列表可以进行访问，这样才能让run_pending_tasks()窃取任务。需要每个线程在线程池队列上进行注册，或由线程池指定一个线程。同样，还需要保证数据队列中的任务适当的被同步和保护，这样队列的不变量就不会被破坏。

实现一个无锁队列，让其拥有线程在其他线程窃取任务的时候，能够推送和弹出一个任务是可能的；不过，这个队列的实现就超出了本书的讨论范围。为了证明这种方法的可行性，将使用一个互斥量来保护队列中的数据。我们希望任务窃取是一个不常见的现象，这样就会减少对互斥量的竞争，并且使得简单队列的开销最小。下面，实现了一个简单的基于锁的任务窃取队列。

清单9.7 基于锁的任务窃取队列

```
class work_stealing_queue
{
private:
  typedef function_wrapper data_type;
  std::deque<data_type> the_queue;  // 1
  mutable std::mutex the_mutex;

public:
  work_stealing_queue()
  {}

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

  void push(data_type data)  // 2
  {
    std::lock_guard<std::mutex> lock(the_mutex);
    the_queue.push_front(std::move(data));
  }

  bool empty() const
  {
    std::lock_guard<std::mutex> lock(the_mutex);
    return the_queue.empty();
  }

  bool try_pop(data_type& res)  // 3
  {
```

```
30      std::lock_guard<std::mutex> lock(the_mutex);
31      if(the_queue.empty())
32      {
33        return false;
34      }
35
36      res=std::move(the_queue.front());
37      the_queue.pop_front();
38      return true;
39    }
40
41    bool try_steal(data_type& res)  // 4
42    {
43      std::lock_guard<std::mutex> lock(the_mutex);
44      if(the_queue.empty())
45      {
46        return false;
47      }
48
49      res=std::move(the_queue.back());
50      the_queue.pop_back();
51      return true;
52    }
53  };
```

这个队列对 `std::deque<fuction_wrapper>` 进行了简单的包装①，就能通过一个互斥锁来对所有访问进行控制了。push()②和try_pop()③对队列的前端进行操作，try_steal()④对队列的后端进行操作。

这就说明每个线程中的“队列”是一个后进先出的栈，最新推入的任务将会第一个执行。从缓存角度来看，这将对性能有所提升，因为任务相关的数据一直存于缓存中，要比提前将任务相关数据推送到栈上好。同样，这种方式很好的映射到某个算法上，例如：快速排序。之前的实现中，每次调用do_sort()都会推送一个任务到栈上，并且等待这个任务执行完毕。通过对最新推入任务的处理，就可以保证在将当前所需数据块处理完成前，其他任务是否需要这些数据块，从而可以减少活动任务的数量和栈的使用次数。try_steal()从队列末尾获取任务，为了减少与try_pop()之间的竞争；使用在第6、7章中的所讨论的技术来让try_pop()和try_steal()并发执行。

OK，现在拥有了一个很不错的任务队列，并且支持窃取；那这个队列将如何在线程池中使用呢？这里简单的展示一下。

清单9.8 使用任务窃取的线程池

```
class thread_pool
{
  typedef function_wrapper task_type;

  std::atomic_bool done;
  thread_safe_queue<task_type> pool_work_queue;
  std::vector<std::unique_ptr<work_stealing_queue> > queues;  // 1
  std::vector<std::thread> threads;
  join_threads joiner;

  static thread_local work_stealing_queue* local_work_queue;  // 2
  static thread_local unsigned my_index;

  void worker_thread(unsigned my_index_)
  {
    my_index=my_index_;
    local_work_queue=queues[my_index].get();  // 3
    while(!done)
    {
      run_pending_task();
    }
  }

  bool pop_task_from_local_queue(task_type& task)
  {
    return local_work_queue && local_work_queue->try_pop(task);
  }

  bool pop_task_from_pool_queue(task_type& task)
  {
    return pool_work_queue.try_pop(task);
  }

  bool pop_task_from_other_thread_queue(task_type& task)  // 4
  {
    for(unsigned i=0;i<queues.size();++i)
    {
      unsigned const index=(my_index+i+1)%queues.size();  // 5
      if(queues[index]->try_steal(task))
      {
        return true;
      }
    }
    return false;
  }
```

```
46
47  public:
48    thread_pool():
49      done(false),joiner(threads)
50    {
51      unsigned const thread_count=std::thread::hardware_concurrency();
52
53      try
54      {
55        for(unsigned i=0;i<thread_count;++i)
56        {
57          queues.push_back(std::unique_ptr<work_stealing_queue>(  // 6
58                           new work_stealing_queue));
59          threads.push_back(
60            std::thread(&thread_pool::worker_thread,this,i));
61        }
62      }
63      catch(...)
64      {
65        done=true;
66        throw;
67      }
68    }
69
70    ~thread_pool()
71    {
72      done=true;
73    }
74
75    template<typename FunctionType>
76    std::future<typename std::result_of<FunctionType()>::type> submit(
77      FunctionType f)
78    {
79      typedef typename std::result_of<FunctionType()>::type result_type;
80      std::packaged_task<result_type()> task(f);
81      std::future<result_type> res(task.get_future());
82      if(local_work_queue)
83      {
84        local_work_queue->push(std::move(task));
85      }
86      else
87      {
88        pool_work_queue.push(std::move(task));
89      }
90      return res;
```

```
91    }
92
93    void run_pending_task()
94    {
95      task_type task;
96      if(pop_task_from_local_queue(task) ||  // 7
97         pop_task_from_pool_queue(task) ||  // 8
98         pop_task_from_other_thread_queue(task))  // 9
99      {
100       task();
101     }
102     else
103     {
104       std::this_thread::yield();
105     }
106   }
107 };
```

这段代码与清单9.6很相似。第一个不同在于，每个线程都有一个work_stealing_queue，而非只是普通的 `std::queue<>` ②。当每个线程被创建，就创建了一个属于自己的工作队列⑥，每个线程自己的工作队列将存储在线程池的全局工作队列中①。列表中队列的序号，会传递给线程函数，然后使用序号来索引对应队列③。这就意味着线程池可以访问任意线程中的队列，为了给闲置线程窃取任务。run_pending_task()将会从线程的任务队列中取出一个任务来执行⑦，或从线程池队列中获取一个任务⑧，亦或从其他线程的队列中获取一个任务⑨。

pop_task_from_other_thread_queue()④会遍历池中所有线程的任务队列，然后尝试窃取任务。为了避免每个线程都尝试从列表中的第一个线程上窃取任务，每一个线程都会从下一个线程开始遍历，通过自身的线程序号来确定开始遍历的线程序号。

使用线程池有很多好处，还有很多很多的方式能为某些特殊用法提升性能，不过这就留给读者作为作业吧。

特别是还没有探究动态变换大小的线程池，即使线程被阻塞的时候(例如：I/O或互斥锁)，程序都能保证CPU最优的使用率。

下面，我们将来了解一下线程管理的“高级”用法——中断线程。