2.3 转移线程所有权

假设要写一个在后台启动线程的函数，想通过新线程返回的所有权去调用这个函数，而不是等待线程结束再去调用；或完全与之相反的想法：创建一个线程，并在函数中转移所有权，都必须要等待线程结束。总之，新线程的所有权都需要转移。

这就是移动引入 `std::thread` 的原因，C++标准库中有很多*资源占有*(resource-owning)类型，比如 `std::ifstream` , `std::unique_ptr` 还有 `std::thread` 都是可移动，但不可拷贝。这就说明执行线程的所有权可以在 `std::thread` 实例中移动，下面将展示一个例子。例子中，创建了两个执行线程，并且在 `std::thread` 实例之间(t1,t2和t3)转移所有权：

```
void some_function();
void some_other_function();
std::thread t1(some_function);            // 1
std::thread t2=std::move(t1);             // 2
t1=std::thread(some_other_function);    // 3
std::thread t3;                             // 4
t3=std::move(t2);                         // 5
t1=std::move(t3);                         // 6 赋值操作将使程序崩溃
```

首先，新线程开始与t1相关联。当显式使用 `std::move()` 创建t2后②，t1的所有权就转移给了t2。之后，t1和执行线程已经没有关联了；执行some_function的函数现在与t2关联。

然后，与一个临时 `std::thread` 对象相关的线程启动了③。为什么不显式调用 `std::move()` 转移所有权呢？因为，所有者是一个临时对象——移动操作将会隐式的调用。

t3使用默认构造方式创建④，与任何执行线程都没有关联。调用 `std::move()` 将与t2关联线程的所有权转移到t3中⑤。因为t2是一个命名对象，需要显式的调用 `std::move()` 。移动操作⑤完成后，t1与执行some_other_function的线程相关联，t2与任何线程都无关联，t3与执行some_function的线程相关联。

最后一个移动操作，将some_function线程的所有权转移⑥给t1。不过，t1已经有了一个关联的线程(执行some_other_function的线程)，所以这里系统直接调用 `std::terminate()` 终止程序继续运行。这样做（不抛出异常， `std::terminate()` 是*noexcept*函数)是为了保证与 `std::thread` 的析构函数的行为一致。2.1.1节中，需要在线程对象被析构前，显式的等待线程完成，或者分离

它；进行赋值时也需要满足这些条件(说明：不能通过赋一个新值给 `std::thread` 对象的方式来"丢弃"一个线程)。

`std::thread` 支持移动，就意味着线程的所有权可以在函数外进行转移，就如下面程序一样。

清单2.5 函数返回 `std::thread` 对象

```
std::thread f()
{
  void some_function();
  return std::thread(some_function);
}

std::thread g()
{
  void some_other_function(int);
  std::thread t(some_other_function,42);
  return t;
}
```

当所有权可以在函数内部传递，就允许 `std::thread` 实例可作为参数进行传递，代码如下：

```
void f(std::thread t);
void g()
{
  void some_function();
  f(std::thread(some_function));
  std::thread t(some_function);
  f(std::move(t));
}
```

`std::thread` 支持移动的好处是可以创建thread_guard类的实例(定义见 清单2.3)，并且拥有其线程的所有权。当thread_guard对象所持有的线程已经被引用，移动操作就可以避免很多不必要的麻烦；这意味着，当某个对象转移了线程的所有权后，它就不能对线程进行加入或分离。为了确保线程程序退出前完成，下面的代码里定义了scoped_thread类。现在，我们来看一下这段代码：

清单2.6 scoped_thread的用法

```
class scoped_thread
{
  std::thread t;
```

```
public:
  explicit scoped_thread(std::thread t_):                 // 1
    t(std::move(t_))
  {
    if(!t.joinable())                                     // 2
      throw std::logic_error(“No thread”);
  }
  ~scoped_thread()
  {
    t.join();                                            // 3
  }
  scoped_thread(scoped_thread const&)=delete;
  scoped_thread& operator=(scoped_thread const&)=delete;
};

struct func; // 定义在清单2.1中

void f()
{
  int some_local_state;
  scoped_thread t(std::thread(func(some_local_state)));    // 4
  do_something_in_current_thread();
}                                                        // 5
```

与清单2.3相似，不过这里新线程是直接传递到scoped_thread中④，而非创建一个独立的命名变量。当主线程到达f()函数的末尾时，scoped_thread对象将会销毁，然后加入③到的构造函数①创建的线程对象中去。而在清单2.3中的thread_guard类，就要在析构的时候检查线程是否"可加入"。这里把检查放在了构造函数中②，并且当线程不可加入时，抛出异常。

`std::thread` 对象的容器，如果这个容器是移动敏感的(比如，标准中的 `std::vector<>`)，那么移动操作同样适用于这些容器。了解这些后，就可以写出类似清单2.7中的代码，代码量产了一些线程，并且等待它们结束。

清单2.7 量产线程，等待它们结束

```
void do_work(unsigned id);

void f()
{
  std::vector<std::thread> threads;
  for(unsigned i=0; i < 20; ++i)
  {
    threads.push_back(std::thread(do_work,i)); // 产生线程
```

```
  }
  std::for_each(threads.begin(),threads.end(),
                std::mem_fn(&std::thread::join)); // 对每个线程调用join()
}
```

我们经常需要线程去分割一个算法的总工作量，所以在算法结束的之前，所有的线程必须结束。清单2.7说明线程所做的工作都是独立的，并且结果仅会受到共享数据的影响。如果f()有返回值，这个返回值就依赖于线程得到的结果。在写入返回值之前，程序会检查使用共享数据的线程是否终止。操作结果在不同线程中转移的替代方案，我们会在第4章中再次讨论。

将 `std::thread` 放入 `std::vector` 是向线程自动化管理迈出的第一步：并非为这些线程创建独立的变量，并且将他们直接加入，可以把它们当做一个组。创建一组线程(数量在运行时确定)，可使得这一步迈的更大，而非像清单2.7那样创建固定数量的线程。