互斥量、lock_guard 与 unique_lock

互斥量

互斥量，或者说互斥锁，是在多线程环境下保护共享变量的一种机制，通常在多个线程对某一个共享变量同时进行读/写时使用。若该共享变量为只读，则不需要互斥量保护

使用互斥量的两个要点

- 获取互斥量与释放互斥量必须成对出现，即使是有异常发生
- 在保证程序运行正确的前提下，尽可能减少临界区的运行时间

std::mutex

- 初始化
  - std::mutex 使用默认构造函数，直接初始化即可
  - std::mutex mtx;
- 加锁/解锁
  - mtx.lock();
  - mtx.unlock();

需设计为异常安全的

实际上，在 C++ 并发编程中我们很少直接使用 std::mutex，而是向智能指针那样，使用一个"包装类"

std::lock_guard

std::lock_guard 和 shard_ptr 一样，为类模板，同样是利用 RAII 来管理资源

使用

```
std::mutex mtx;
std::lock_guard<std::mutex> guard(mtx);
```

本质上就是在构造函数中调用 **mtx.lock()** 获取互斥量，在析构函数中调用 **mtx.unlock()** 释放互斥量

所以，lock_guard 一定是异常安全的，即使有异常发生，互斥量依然能顺利释放

尽管 lock_guard 能够保证异常安全，但是由 lock_guard 所管理的 mutex 却无法提前释放，也就是说我们必须要等到 guard 局部变量被回收才能够释放该互斥量，在灵活性上会有一些小问题

std::adopt_lock

我们也可以通过传递该参数，告诉 lock_guard 当前互斥量已经加锁，调用构造函数的时候就别再调用 mtx.lock() 了

std::unique_lock

std::unique_lock 和 std::lock_guard 所实现的功能是一样的，都是用来管理互斥量的。但是，unique_lock 要比 lock_guard 更加灵活，我们可以对其执行延迟加锁、提前解锁

```
std::mutex mtx;
std::unique_lock<std::mutex> guard(mtx);    // 初始化时加锁
guard.unlock();                             // 可提前解锁
guard.lock();                               // 也可继续加锁
```

因此，在后面我们将会看到，条件变量 condition_variable 需使用 unique_lock，而不是 lock_guard