通知状态的改变: C++ 条件变量

基本原理

- POSIX Condition Variable
  - C++ 中的条件变量与 POSIX 条件变量原理基本一致
  - 只不过 C++ 中的条件变量是通过面向对象的方式使用的
- std::condition_variable — 是一个类，该类的接口层面实现非常简单，寥寥几个方法而已
- 初始化
  - 在前面的 POSIX 条件变量中，我们可以通过动态初始化的方式，为我们的条件变量增添一些属性。但是 C++ 中的 condition_variable 却只有默认的构造函数
  - std::condition_variable cond; // 直接初始化即可
- 使用

同样地，condition_variable 对象必须搭配互斥量使用，并且我们通常会使用 std::unique_lock，或者说，只能是 std::unique_lock

```
int main() {
    std::mutex mtx;
    std::unique_lock<std::mutex> guard(mtx);  // 创建 unique_lock 实例

    std::condition_variable cond;                  // 初始化条件变量

    cond.wait(guard);                              // 一个参数的 wait
    cond.wait(guard, [](){return true});           // 两个参数的 wait

    cond.notify_one();                             // 通知一个线程
    cond.notify_all();                             // 通知全部线程
}
```

wait 方法

- 内部实现
  - **首先，wait 将判断条件是否成立，判断结果由第二个参数决定，也就是一个可调用对象的返回结果，通常是 true/false。当返回为 true 时，程序继续执行；当返回为 false 时，释放 unique_lock 中管理的互斥量，并调用 OS 相关的系统调用，使当前线程进入睡眠**
  - 如果我们调用的是 cond.wait(guard)，也就是不带参数的 wait 方法，相当于上面儿的可调用对象返回 false，此时只能由 notify_one 唤醒，唤醒后若获取到互斥量则继续执行
  - **当其它线程调用 notify_one 或者是 notify_all 时，wait 方法尝试获取 unique_lock 所管理的互斥量，若获取成功，则再次判断条件是否满足。若满足，线程继续执行；若不满足，重复第一个过程**
- 可以看到，condition_variable.wait() 方法实际上替我们封装了 "测试条件变量的判断条件" 那个 while 循环，我们只需要告诉它我们的判断依据即可

简易阻塞队列

- 设计
  - 对于一个阻塞队列而言，本质上仍然是一个队列，我们将数据从队列的尾部推入，从队列的头部弹出，因此可以使用 STL 中的 list，即双向链表实现队列
  - list 的 push_back() 和 pop_back() 方法均是非线程安全的，所以我们必须要使用一个互斥量来保护 list。push 数据和 pop 数据必须串行执行
  - 当队列为空时，此时消费者不能够再取出任何数据进行消费，因此，我们可以使用 wait 方法使当前线程阻塞，当生产者那边儿有数据 push 了再通知消费者进行消费
  - 当队列容量达到限制时，此时生产者不可以再往里面儿 push 数据了，因此，使用 wait 方式使得当前线程阻塞，当消费者那边儿消费了一条数据之后再通知生产者可以推入数据
- 经过上面的分析，我们需要使用 2 个条件变量才能够完成，假设一个叫 not_full，一个叫 not_empty。not_full 表示当前队列未满，not_empty 表示当前队列不为空。
  - 生产者 — **调用 not_full.wait() 以及 not_empty.notify_one()**
  - 消费者 — **调用 not_empty.wait() 以及 not_full.notify_one()**

```cpp
template <typename T>
class BlockingQueue {
private:
    size_t m_capacity;
    std::list<T> m_list;
    std::mutex m_mtx;
    std::condition_variable _not_full;
    std::condition_variable _not_empty;
public:
    BlockingQueue(size_t capacity = 4) : m_capacity(capacity) {}

    void push(const T& element) {
        std::unique_lock<std::mutex> guard(m_mtx);

        _not_full.wait(guard, [this]{
            return this->m_list.size() != this->m_capacity;
        });

        m_list.push_back(element);

        _not_empty.notify_one();
    }

    void pop() {
        std::unique_lock<std::mutex> guard(m_mtx);

        _not_empty.wait(guard, [this]{
            return this->m_list.size() != 0;
        });

        m_list.pop_front();

        _not_full.notify_one();
    }
};
```