件变量使用详解

ariable介绍

]可以使用条件变量（condition_variable）实现多个线程间的同步操作；当条件不满足时，相关线程被一
条件出现，这些线程才会被唤醒。

]下:

	构造对象 (公开成员函数)
	析构对象 (公开成员函数)
删除]	不可复制赋值 (公开成员函数)

	通知一个等待的线程 (公开成员函数)
	通知所有等待的线程 (公开成员函数)

	阻塞当前线程，直到条件变量被唤醒 (公开成员函数)
	阻塞当前线程，直到条件变量被唤醒，或到指定时限时长后 (公开成员函数)
	阻塞当前线程，直到条件变量被唤醒，或直到抵达指定时间点 (公开成员函数)

线程间共享的全局变量进行同步的一种机制，主要包括两个动作：

等待"条件变量的条件成立"而挂起；

程使"条件成立"，给出信号，从而唤醒被等待的线程。

条件变量的使用总是和一个互斥锁结合在一起；通常情况下这个锁是std::mutex，并且管理这个锁 只能是
std::mutex> RAII模板类。

骤，分别是使用以下两个方法实现：

使用的是condition_variable类成员wait 、wait_for 或 wait_until。

的是condition_variable类成员notify_one或者notify_all函数。

使用std::unique_lock<std::mutex>说明

k_guard都是管理锁的辅助类工具，都是RAII风格；它们是在定义时获得锁，在析构时释放锁。它们的主
e_lock锁机制更加灵活，可以再需要的时候进行lock或者unlock调用，不非得是析构或者构造时。它们的
函数就可以一目了然。

lock_guard<T>

造 lock_guard ，可选地锁定给定 公开成员函数)	
析构 lock_guard 对象，解锁底层互 公开成员函数)	
不可复制赋值 公开成员函数)	

类型	定义
mutex_type	Mutex

std::unique_lock<T>

成员函数

(构造函数)	构造 unique_lock ，可选地锁定提供的互斥 (公开成员函数)
(析构函数)	若占有关联互斥，则解锁之 (公开成员函数)
operator=	若占有则解锁互斥，并取得另一者的所有权 (公开成员函数)
锁定	
lock	锁定关联互斥 (公开成员函数)
try_lock	尝试锁定关联互斥，若互斥不可用则返回 (公开成员函数)
try_lock_for	试图锁定关联的定时可锁 (*TimedLockable*) 互斥，若互斥在给定时长中不可用则返回 (公开成员函数)
try_lock_until	尝试锁定关联可定时锁 (*TimedLockable*) 互斥，若抵达指定时间点互斥仍不可用则返回 (公开成员函数)
unlock	解锁关联互斥 (公开成员函数)

明

过成员函数wait()/wait_for()/wait_until()函数实现的。这里主要说明前面两个函数：

数

式：

```
 wait( std::unique_lock<std::mutex>& lock );
edicate 谓词函数，可以普通函数或者lambda表达式
late< class Predicate >
 wait( std::unique_lock<std::mutex>& lock, Predicate pred );
```

的线程阻塞直至条件变量被**通知**，或**虚假唤醒发生**，可选地循环**直至满足某谓词。**

员函数

式：

```
late< class Rep, class Period >
:cv_status wait_for( std::unique_lock<std::mutex>& lock,
                     const std::chrono::duration<Rep, Period>& rel_time);


late< class Rep, class Period, class Predicate >
 wait_for( std::unique_lock<std::mutex>& lock,
           const std::chrono::duration<Rep, Period>& rel_time,
           Predicate pred);
```

!当前线程阻塞直至条件变量被**通知**，或**虚假唤醒发生**，或者**超时返回。**

rel_time 所指定的关联时限则为 std::cv_status::timeout ，否则为 std::cv_status::no_timeout 。

rel_time 时限后谓词 pred 仍求值为 false 则为 false ，否则为 true 。

wait函数都在会阻塞时，自动释放锁权限，即调用unique_lock的成员函数unlock（），以便其他线程能
这就是条件变量只能和unique_lock一起使用的原因，否则当前线程一直占有锁，线程被阻塞。

_one

下：

```
ify_one() noexcept;
在 *this 上等待，则调用 notify_one 会解阻塞(唤醒)等待线程之一。

ify_all() noexcept;
在 *this 上等待，则解阻塞（唤醒)全部等待线程。
```

wait类型函数返回时要不是因为被唤醒，要不是因为超时才返回，但是在实际中发现，因此操作系统的原
满足条件时，它也会返回，这就导致了虚假唤醒。因此，我们一般都是使用带有谓词参数的wait函数，
Predicate pred) 类型的函数等价于：

```
pred()) //while循环，解决了虚假唤醒的问题

(lock);
```

虚假唤醒的时，代码形式如下：

```
足xxx条件)

有虚假唤醒，wait函数可以一直等待，直到被唤醒或者超时，没有问题。
```

```
实际中却存在虚假唤醒，导致假设不成立，wait不会继续等待，跳出if语句，
前执行其他代码，流程异常
();


马
```

使用while语句解决：

```
(xxx条件) )

假唤醒发生，由于while循环，再次检查条件是否满足，
则继续等待，解决虚假唤醒
();

马
```

条件变量，解决生产者-消费者问题，该问题主要描述如下：

题，也称有限缓冲问题，是一个多进程/线程同步问题的经典案例。该问题描述了共享固定大小缓冲区的
—即所谓的“生产者”和“消费者”,在实际运行时会发生的问题。

是**生成一定量的数据放到缓冲区中**，然后重复此过程。**与此同时，费者也在缓冲区消耗这些数据。**该问
保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。

必须让**生产者在缓冲区满时休眠（要么干脆就放弃数据），等到下次消费者消耗缓冲区中的数据的时
被唤醒，开始往缓冲区添加数据。**

费者在缓冲区空时进入休眠，等到生产者往缓冲区添加数据之后，再唤醒消费者。

码如下：

```
ex g_cvMutex;
dition_variable g_cv;


ue<int> g_data_deque;
最大数目
t  MAX_NUM = 30;

xt_index = 0;

 消费者线程个数
t PRODUCER_THREAD_NUM  = 3;
t CONSUMER_THREAD_NUM = 3;

oducer_thread(int thread_id)

le (true)

  std::this_thread::sleep_for(std::chrono::milliseconds(500));
  //加锁
  std::unique_lock <std::mutex> lk(g_cvMutex);
  //当队列未满时，继续添加数据
  g_cv.wait(lk, [](){ return g_data_deque.size() <= MAX_NUM; });
  g_next_index++;
  g_data_deque.push_back(g_next_index);
  std::cout << "producer_thread: " << thread_id << " producer data: " << g_next_index;
  std::cout << " queue size: " << g_data_deque.size() << std::endl;
  //唤醒其他线程
  g_cv.notify_all();
  //自动释放锁



nsumer_thread(int thread_id)
```

```
e (true)

std::this_thread::sleep_for(std::chrono::milliseconds(550));
//加锁
std::unique_lock <std::mutex> lk(g_cvMutex);
//检测条件是否达成
g_cv.wait( lk,   []{ return !g_data_deque.empty(); });
//互斥操作，消息数据
int data = g_data_deque.front();
g_data_deque.pop_front();
std::cout << "\tconsumer_thread: " << thread_id << " consumer data: ";
std::cout << data << " deque size: " << g_data_deque.size() << std::endl;
//唤醒其他线程
g_cv.notify_all();
//自动释放锁




()

:thread arrRroducerThread[PRODUCER_THREAD_NUM];
:thread arrConsumerThread[CONSUMER_THREAD_NUM];

(int i = 0; i < PRODUCER_THREAD_NUM; i++)

arrRroducerThread[i] = std::thread(producer_thread, i);


(int i = 0; i < CONSUMER_THREAD_NUM; i++)

arrConsumerThread[i] = std::thread(consumer_thread, i);


(int i = 0; i < PRODUCER_THREAD_NUM; i++)

arrRroducerThread[i].join();


(int i = 0; i < CONSUMER_THREAD_NUM; i++)

arrConsumerThread[i].join();


rn 0;
```