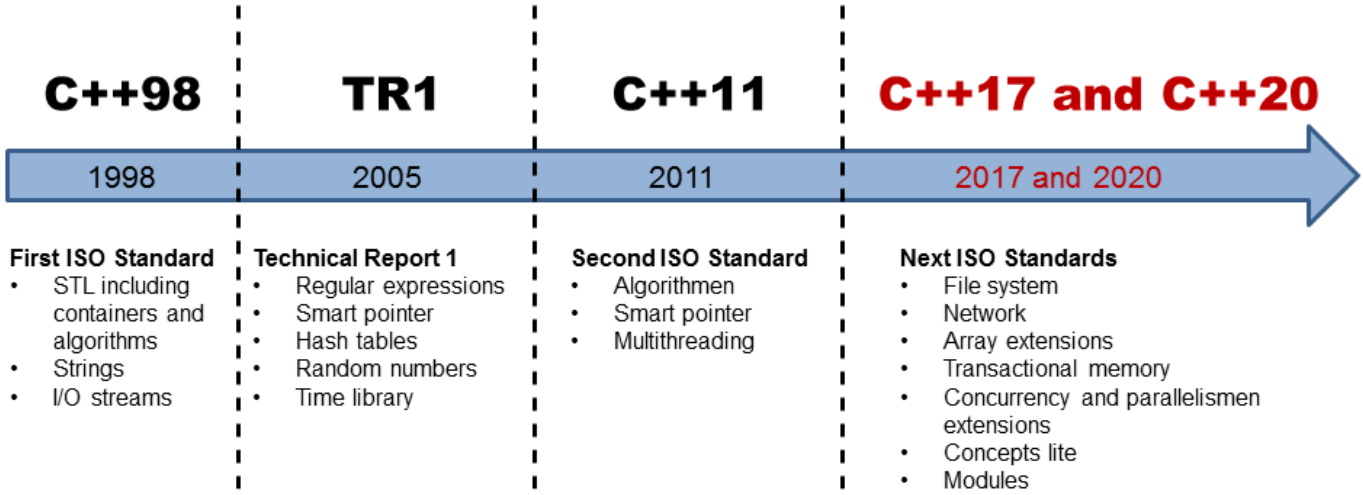
C++11 Thread

焦伟鹏，2023-07-30初稿



从 C++11 开始，标准库里已经包含了对线程的支持，std::thread是C++11标准库中的多线程的支持库，pthread.h 是标准库没有添加多线程之前的在Linux上用的多线程库。std::thread 是面向对象的多线程库，使用简单，推荐在项目中使用 std::thread 代替 pthread.h。

编译环境

头文件

C++11新标准引入了四个支持多线程的文件，<atomic>、<thread>、<mutex>、<condition_variable>、<future>。

线程相关 `#include <thread>`

互斥锁相关 `#include <mutex>`

条件变量相关 `#include <condition_variable>`

编译选项

修改 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(main)

set(CMAKE_CXX_FLAGS ${CMAKE_CXX_FLAGS} "-O3 -std=c++17 ")

find_package(Threads)
add_executable(${PROJECT_NAME} ${CMAKE_SOURCE_DIR}/src/main.cpp)
target_link_libraries (${PROJECT_NAME} ${CMAKE_THREAD_LIBS_INIT})
```

跨平台

如果在 Windows 环境中运行，如果程序中引入的 <unistd.h> 头文件，则需要改为 <Windows.h>，sleep()语句改为 Sleep()语句。

主要成员函数

POSIX 标准中，线程所执行函数的参数和返回值都必须为 void* 类型。而 thread 类创建的线程可以执行任意的函数，即不对函数的参数和返回值做具体限定。

线程管理

成员函数	功 能
get_id()	获取当前 thread 对象的线程 ID。
joinable()	判断当前线程是否支持调用 join() 成员函数。
join()	阻塞当前 thread 对象所在的线程，直至 thre 线程执行完毕后，所在线程才能继续执行。
detach()	将当前线程从调用该函数的线程中分离出去， 行。
swap()	交换两个线程的状态。

`<thread>` 头文件中不仅定义了 thread 类，还提供了一个名为 this_thread 的命名空间，此空间中包含一些功能实用的函数

函数	功 能

get_id()	获得当前线程的 ID
yield()	阻塞当前线程，直至条件成熟
sleep_until()	阻塞当前线程，直至某个时间点为止
sleep_for()	阻塞当前线程指定的时间（例如阻塞 5 秒）

创建线程

```cpp
#include <iostream>
#include <thread>
#include <pthread.h>

using namespace std;

void SayHello() {
    cout << "Hello World" << endl;
}

int main() {
    std::thread t1(SayHello);
    // 等待子线程结束才退出当前线程
    pthread_exit(nullptr);
}
```

如果不加 `pthread_exit(nullptr)`，会报 `libc++abi: terminating` 程序终止的错误。

线程分离

可以通过 detach() 函数，将子线程和主线分离，子线程可以独立继续运行，即使主线程结束，子线程也不会结束。

```cpp
#include <iostream>
#include <thread>
using namespace std::literals::chrono_literals;
using namespace std;

void SayHello() {
    cout << "Hello World" << endl;
}

int main() {
    std::thread t1(SayHello);
```

```
    t1.detach();
    this_thread::sleep_for(10ms);
    // 低于C++17使用这行代码  this_thread::sleep_for(chrono::milliseconds(10));
    return 0;
}
```

线程合并

join() 函数可以在当前线程等待线程运行结束。

```
#include <iostream>
#include <thread>
using namespace std::literals::chrono_literals;
using namespace std;

void SayHello() {
    this_thread::sleep_for(10ms);
    cout << "Hello World" << endl;
}

int main() {
    std::thread t1(SayHello);
    t1.join();
    return 0;
}
```

注意，每个thread 对象在调用析构函数销毁前，要么调用 join() 函数令主线程等待子线程执行完成，要么调用 detach() 函数将子线程和主线程分离，两者比选其一，否则程序可能存在以下两个问题：

- 线程占用的资源将无法全部释放，造成内存泄漏；
- 当主线程执行完成而子线程未执行完时，程序执行将引发异常。

传递参数

传值

```
#include <iostream>
#include <thread>
using namespace std::literals::chrono_literals;
using namespace std;

void SayHello(int id, string name) {
```

```
    this_thread::sleep_for(10ms);
    cout << "ID:" << id << ", Hello " << name << endl;
}

int main() {
    std::thread t1(SayHello, 1, "Wiki");
    t1.join();
    return 0;
}
```

传引用

```
#include <iostream>
#include <thread>

using namespace std;

void thread_func3(int &c)
{
    cout << "thread_func3(): &c = " << &c
    cout << " --> c + 10 =" << (c += 10) << endl;
}

int main()
{
    int c = 10;

    thread t3(thread_func3, ref(c));
    t3.join();
    cout << "main --> 3 : &c = " << &c << ", c = " << c << endl;

    return 0;
}
```

传类

```
#include <iostream>
#include <thread>

using namespace std;

void thread_func3(int &c)
{
    cout << "thread_func3(): &c = " << &c
    cout << " --> c + 10 =" << (c += 10) << endl;
```

```
}

int main()
{
    int c = 10;

    thread t3(thread_func3, ref(c));
    t3.join();
    cout << "main --> 3 : &c = " << &c << ", c = " << c << endl;

    return 0;
}

```

最好使用取地址符&的方式传入类函数，避免兼容问题。

线程睡眠

```
using namespace std::literals::chrono_literals;
// 让当前线程睡眠 10 毫秒
this_thread::sleep_for(10ms);
// 低于C++17使用这行代码  this_thread::sleep_for(chrono::milliseconds(10));
// 让当前线程睡眠 5 秒
this_thread::sleep_for(5s);
```

线程号

```
// thread::get_id / this_thread::get_id
#include <iostream>       // std::cout
#include <thread>         // std::thread, std::thread::id, std::this_thread::get_id
#include <chrono>         // std::chrono::seconds

std::thread::id main_thread_id = std::this_thread::get_id();

void is_main_thread() {
  if ( main_thread_id == std::this_thread::get_id() )
    std::cout << "This is the main thread.\n";
  else
    std::cout << "This is not the main thread.\n";
}

int main()
{
```

```
  is_main_thread();
  std::thread th (is_main_thread);
  th.join();
}

```

数据本地化

`thread_local`

C++11中提供了thread_local，thread_local定义的变量在每个线程都保存一份副本，而且互不干扰，在线程退出的时候自动销毁。

```cpp
#include <iostream>
#include <thread>
using namespace std::literals::chrono_literals;
using namespace std;
thread_local int t_l_counter = 0;

void sub_thread() {
    cout << "flag1 t_l_counter: " << t_l_counter << endl; // 看到的是副线程的全局变量 0
    t_l_counter = 2; // 将副线程全局变量设为2
}

int main() {
    // C++11中提供了thread_local，thread_local定义的变量在每个线程都保存一份副本，而且互不干扰，在线程退出的时候自动销毁。
    t_l_counter = 1; // 主线程将主线程的全局变量设为1
    std::thread t1(sub_thread); // 副线程看到的全局变量是副线程的全局变量仍为0
    t1.join();
    cout << "flag2 t_l_counter: " << t_l_counter << endl; // 此全局变量为主线程的1
    return 0;
}
```

线程所有权转移

std::move函数

```cpp
#include <iostream>
#include <thread>
```

```cpp

using namespace std;

void thread_func(int &a)
{
    cout << "thread_func: a = " << (a += 10) << endl;
}

int main()
{
    int x = 10;
    thread t1(thread_func, ref(x));
    thread t2(move(t1)); // t1 线程失去所有权

    thread t3;
    t3 = move(t2); // t2 线程失去所有权

    // t1.join(); //执行会报错：已放弃（核心已转储）
    t3.join();

    cout << "main end: x = " << x << endl;

    return 0;
}
```

std::swap交换各自线程的句柄，std::swap(Thread1,Thread2)，Thread1.swap(Thread2)均可用。

异步

> C++11中还引入了async函数,是一种更高层的异步方式.它不必显式手写join函数来让主线程等待子线程.它可以用std::future类型的变量来接收线程函数运行的结果,并通过get()的方法来获得结果.这样就不必像thread那样提前定义一个全局变量,在线程函数中进行赋值操作. 此外async还可以指定线程创建策略–是立马执行还是延迟执行.

```cpp
#include <iostream>
#include <chrono>
#include <thread>
#include <future>

using namespace std;

int sum = 0;
int add(int a, int b)
```

```
{
    sum = a + b;
    return sum;
}

int main()
{
    thread t(add, 2, 3);
    t.join();
    cout << "sum: " << sum << endl;

    auto f = async(add, 3, 4);
    cout << "result: " << f.get() << endl;
}
```

执行结果:

```
sum: 5

result: 7
```

同步锁

无锁线程

```
#include <iostream>
#include <thread>
using namespace std;
void test() {
    cout << "task start thread ID: " << this_thread::get_id() << endl;
    this_thread::sleep_for(10ms);
    cout << "task end thread ID: " << this_thread::get_id() << endl;
}
int main() {
    std::thread t1(test);
    std::thread t2(test);
    std::thread t3(test);
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```

运行结果：

```
task start thread ID: task start thread ID: task start thread ID: 0x70000fab00000x70000fb33000

0x70000fbb6000
task end thread ID: 0x70000fab0000
task end thread ID: task end thread ID: 0x70000fb33000
0x70000fbb6000
```

以上代码数据的结果是无序的，如果我们需要同一时间只有一个线程在test函数中执行代码，那么就要加锁，lock() 用于加锁，而unlock() 解锁。

互斥锁

单次加锁

std::mutex，最基本的 Mutex 类

> 独占互斥量，只能加锁一次
> std::mutex 是C++11 中最基本的互斥量，std::mutex 对象提供了独占所有权的特性——即不支持递归地对 std::mutex 对象上锁，而 std::recursive_lock 则可以递归地对互斥量对象上锁。

std::mutex成员函数：

```
构造函数，std::mutex不允许拷贝构造，也不允许 move 拷贝，最初产生的 mutex 对象是处于 unlocked 状态的。
lock()，调用线程将锁住该互斥量
unlock()， 解锁，释放对互斥量的所有权。
try_lock()，尝试锁住互斥量，如果互斥量被其他线程占有，则当前线程也不会被阻塞。
```

```
#include <iostream>
#include <thread>
#include <mutex> // 锁的头文件
using namespace std::literals::chrono_literals;
using namespace std;
std::mutex g_mutex;

void test() {
    g_mutex.lock();
    cout << "task start thread ID: " << this_thread::get_id() << endl;
    this_thread::sleep_for(10ms);
```

```
    cout << "task end thread ID: " << this_thread::get_id() << endl;
    g_mutex.unlock();
}
int main() {
    std::thread t1(test);
    std::thread t2(test);
    std::thread t3(test);
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```

运行结果：

```
task start thread ID: 0x70000e4f2000
task end thread ID: 0x70000e4f2000
task start thread ID: 0x70000e46f000
task end thread ID: 0x70000e46f000
task start thread ID: 0x70000e3ec000
task end thread ID: 0x70000e3ec000
```

`try_lock()` 这个函数用来处理分支，如果拿不到锁就做其他事情。

```
#include <iostream>
#include <thread>
#include <mutex> // 锁的头文件
using namespace std::literals::chrono_literals;
using namespace std;
std::mutex g_mutex;
void mutex_try_lock_test() {
    if(g_mutex.try_lock()) // 线程资源上锁
    {
        cout << "task start thread ID: " << this_thread::get_id() << endl;
        this_thread::sleep_for(10ms);
        cout << "task end thread ID: " << this_thread::get_id() << endl;
        g_mutex.unlock(); // 线程资源解锁
    }
    else
    {
        // do something
        this_thread::sleep_for(5ms);
        cout << "fail to get lock thread ID: " << this_thread::get_id() <<
endl;
```

```
    }

}

int main() {
    std::thread t1(mutex_try_lock_test);
    std::thread t2(mutex_try_lock_test);
    t1.join();
    t2.join();
    return 0;
}
```

```
task start thread ID: 140642267166272
fail to get lock thread ID: 140642258773568
task end thread ID: 140642267166272
```

多次加锁

std::recursive_timed_mutex，递归 Mutex 类

> 递归的独占互斥量，允许同一个线程，同一个互斥量，多次被lock，用法和非递归的一样 跟windows的临界区是一样的，但是调用次数是有上限的，效率也低一些。和 std::mutex 不同的是：
> std::recursive_mutex 允许同一个线程对互斥量多次上锁（即递归上锁），来获得对互斥量对象的多层所有权，
> std::recursive_mutex 释放互斥量时需要调用与该锁层次深度相同次数的 unlock()，可理解为 lock() 次数和 unlock() 次数相同。

对于一些递归的函数，每次递归都需要加锁，普通的 **std::mutex** 在第一次使用就已经加锁了，再次递归就会因为拿不到锁阻塞。而 **std::recursive_timed_mutex** 就适用这种情况，对于同一个线程的内容可多次加锁。

单次定时锁

std::timed_mutex，定时 Mutex 类

带超时的互斥量,独占互斥量 这个就是拿不到锁会等待一段儿时间，但是超过设定时间，就继续执行
std::time_mutex 比 std::mutex 多了两个成员函数，try_lock_for()，try_lock_until()。

```
template< class Rep, class Period > bool try_lock_for( const
std::chrono::duration<Rep,Period>& timeout_duration );
//  (C++11 起)
template< class Rep, class Period > bool try_lock_for( const
```

```
std::chrono::duration<Rep,Period>& timeout_duration );
//  (C++11 起)
```

- `try_lock_for`:尝试锁定互斥，若互斥在指定的时限时期中不可用则返回。
- `try_lock_until`:尝试锁定互斥，若直至抵达指定时间点互斥不可用则返回。

```
参数
timeout_duration - 要阻塞的最大时长
返回值
若成功获得锁则为 true ，否则为 false 。
异常
执行期间时钟、时间点或时长可能抛出的异常（标准库提供的时钟、时间点和时长决不抛出）
```

多次定时锁

`std::recursive_timed_mutex，定时递归 Mutex`

带超时的，递归的，独占互斥量，允许同一个线程，同一个互斥量，多次被lock，用法和非递归的一样

读写锁

`std::shared_mutex`

读写锁shared_mutex表示资源可以同时被多个线程读，但只能同时被一个线程写。
不能同时被读线程和写线程占有，只能有其中一种。
使用lock_shared，unlock_shared进行读线程的锁定与解锁。
使用lock，unlock进行写线程的锁定与解锁。

```
typedef std::shared_lock<std::shared_mutex> ReadLock; // 读锁
typedef std::lock_guard<std::shared_mutex> WriteLock; // 写锁
```

原子操作

atomic本意为原子，官方的解释是**原子操作是最小且不可并行化的操作**。这意味着即使是多线程，也要像同步进行一样**同步操作**atomic对象，从而省去了mutex上锁、解锁的时间消耗。

```
#include <iostream>
```

```
#include <thread>
#include <atomic>
using namespace  std;

atomic_int  n = 0;

void plus_n(){
    for(int i =0 ;i < 10000;i++){
        n++;
    }
}
int main(){
    thread th[100];
    for(thread &x : th){
        x = thread(plus_n);
    }
    for(thread &x: th){
        x.join();
    }
    cout << " n : " << n << endl;
    return 0;
}

```

其中 `std::atomic_int` 是 `std::atomic<int>` 的别名。

同步锁封装类

可以创建一个 ScopeMutex 类，**通过构造函数和析构函数实现加锁和解锁**，ScopeMutex 的作用域只在 {} 之内加锁。

```
#include <iostream>
#include <thread>
using namespace std::literals::chrono_literals;
using namespace std;
std::mutex g_mutex;

class ScopeMutex {
public:
    explicit ScopeMutex(std::mutex &mutex) {
        this->mutex = &mutex;
        this->mutex->lock();
    }

```

```
    ~ScopeMutex() {
        this->mutex->unlock();
    }

    std::mutex *mutex;
};

void test() {
    cout << "task prepare thread ID: " << this_thread::get_id() << endl;
    {
        ScopeMutex scopeMutex(g_mutex);
        cout << "task start thread ID: " << this_thread::get_id() << endl;
        this_thread::sleep_for(10ms);
        cout << "task end thread ID: " << this_thread::get_id() << endl;
    }

}

int main() {
    std::thread t1(test);
    std::thread t2(test);
    std::thread t3(test);
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```

运行结果:

```
task prepare thread ID: task prepare thread ID: 0x70000d9bd0000x70000d8b7000
task start thread ID: 0x70000d9bd000

task prepare thread ID: 0x70000d93a000
task end thread ID: 0x70000d9bd000
task start thread ID: 0x70000d8b7000
task end thread ID: 0x70000d8b7000
task start thread ID: 0x70000d93a000
task end thread ID: 0x70000d93a000
```

std::lock_guard

使用lock_guard相比于mutex更加安全，它是基于作用域的，能够自解锁，当该对象创建时，它会像mutex.lock()一样获得互斥锁，当生命周期结束时，它会自动析构(unlock)，不会因为某个线程异常退出而影响其他线程。

声明一个局部的std::lock_guard对象，在其构造函数中进行加锁，在其析构函数中进行解锁。最终的结果就是：

1. 创建即加锁，作用域结束自动解锁。从而使用std::lock_guard()就可以替代lock()与unlock()。
2. 通过设定作用域，使得std::lock_guard在合适的地方被析构。
3. 通过使用{}来调整作用域范围，可使得互斥量m在合适的地方被解锁。

```
#include<iostream>
#include<thread>
#include<mutex>
using namespace std;
// 实例化互斥锁
mutex m;
// 线程函数
void thread1_func(int num)
{
    // 使用lock_guard构造了一个变量lg对锁m进行锁定
    lock_guard<mutex> lg(m);
    cout << num << endl;
    num = num + 1;
    cout << num << endl;
    // 不需要对lg进行任何操作，它作为临时变量被调用析构函数释放时，也同时解锁了
}
```

std::unique_lock

功能类似lock_guard，但使用更加灵活。即使已经使用了unique_lock，依然可以手动使用lock()与unlock()进行锁定和解锁操作，但使用了lock_guard之后，就不能再手动操作了。

```
(1)unique_lock(mutex, adopt_lock_t) //传递被使用过的mutex，且已经被上过锁，通过。无上锁动作，不阻塞。
(2)unique_lock(mutex, defer_lock_t) //传递被使用过的mutex，未被上过锁。无上锁动作，不阻塞。
(3)unique_lock(mutex, try_to_lock_t) //任何状态的mutex。尝试上锁，不阻塞。
(4)unique_lock(_Mutex& _Mtx, const chrono::duration<_Rep, _Period>&
_Rel_time) //在指定时间长内尝试获取传递的mutex的锁返回。若无法获取锁，会阻塞到指定时间长。
(5)unique_lock(mutex_type& m,std::chrono::time_point<Clock,Duration>
const& absolute_time) //在给定时间点尝试获取传递的mutex锁返回。若无法获取锁，会阻塞到指定时间点。
(6)unique_lock(unique_lock&& _Other)
//将已经创建的unique_lock锁的所有权转移到新的锁。保持之前锁的状态，不阻塞 ```
```

unique_lock除了拥有跟std::mutex一样的三个成员函数意外，还提供release()函数。release()返回它所管理的mutex对象指针，并释放所有权；也就是说，这个unique_lock和mutex不再有关系。严格区分unlock()与release()的区别，不要混淆。

```
        std::unique_lock<std::mutex> sbguard(my_mutex);
        std::mutex *ptx = sbguard.release(); //获取指针
        msgRecvQueue.push_back(i);
        ptx->unlock(); //解锁

```

`std::lock_guard` 与 `std::unique_lock` 区别

lock_guard:

```
没有提供加锁和解锁的接口。
通过构造函数和析构函数控制锁的作用范围，创造对象的时候加锁，离开作用域的时候解锁；
```

unique_lock:

```
提供了lock()和unlock()接口，能记录现在处于上锁还是没上锁状态。
可以通过构造函数和析构函数控制锁的作用范围。
在构造函数中延时加锁，在需要的时候手动加锁和解锁。
在析构的时候，会根据当前状态来决定是否要进行解锁（lock_guard就一定会解锁）。
```

赋值操作

unique_lock和lock_guard都不能复制
lock_guard不能移动，但是unique_lock可以移动。

```
// unique_lock 可以移动，不能复制
std::unique_lock[std::mutex](std::mutex) guard1(_mu);
std::unique_lock[std::mutex](std::mutex) guard2 = guard1;  // error
std::unique_lock[std::mutex](std::mutex) guard2 = std::move(guard1); // ok

// lock_guard 不能移动，不能复制
std::lock_guard[std::mutex](std::mutex) guard1(_mu);
std::lock_guard[std::mutex](std::mutex) guard2 = guard1;  // error
std::lock_guard[std::mutex](std::mutex) guard2 = std::move(guard1); // error
```

资源消耗

```
unique_lock更加灵活，因为它要维持mutex的状态，但也因此对于资源的消耗明显要大一些，同时效率也比lock_guard更低一点。
lock_guard虽然笨重一些,但是资源消耗相对要小一点。
```

另外，unique_lock还在条件变量的使用时发挥作用。

其他类型

std::once_flag（提供一种方式，可以保证其实例绑定的函数，能且仅能被执行一次）

adopt_lock_t(假设调用方线程已拥有互斥的所有权)

std::defer_lock_t(不获得互斥的所有权)

std::try_to_lock_t（尝试获得互斥的所有权而不阻塞）

std::call_once（如果多个线程需要同时调用某个函数，call_once 可以保证多个线程对该函数只调用一次）

条件变量

C++ 11标准提供了两种表示条件变量的类，分别是 condition_variable 和 condition_variable_any，它们都定义在 `<condition_variable>` 头文件中。我们知道，为了避免线程间抢夺资源，条件变量通常和互斥锁搭配使用，condition_variable 类表示的条件变量只能和 unique_lock 类表示的互斥锁（可自行加锁和解锁）搭配使用，而 condition_variable_any 类表示的条件变量可以和任意类型的互斥锁搭配使用（例如递归互斥锁、定时互斥锁等）。

条件变量是利用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：一个线程等待”条件变量的条件成立”而挂起；另一个线程使”条件成立”（给出条件成立信号）。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。

1. 两个线程操作同一临界区时，通过互斥锁保护，若A线程已经加锁，B线程再加锁时候会被阻塞，直到A释放锁，B再获得锁运行，进程B必须不停的主动获得锁、检查条件、释放锁、再获得锁、再检查、再释放，一直到满足运行的条件的时候才可以（而此过程中其他线程一直在等待该线程的结束），这种方式是比较消耗系统的资源的。
2. 而条件变量同样是阻塞，还需要通知才能唤醒，线程被唤醒后，它将重新检查判断条件是否满足，如果还不满足，该线程就休眠了，应该仍阻塞在这里，等待条件满足后被唤醒，节省了线程不断运行浪费的资源。
3. 这个过程一般用while语句实现。当线程B发现被锁定的变量不满足条件时会自动的释放锁并把自身置于等待状态，让出CPU的控制权给其它线程。其它线程 此时就有机会去进行操作，当修改完成后再通知那些由于条件不满足而陷入等待状态的线程。这是一种通知模型的同步方式，

大大的节省了CPU的计算资源，减少了线程之间的竞争，而且提高了线程之间的系统工作的效率。这种同步方式就是条件变量。

成员函数	功 能
wait()	阻塞当前线程，等待条件成立。
wait_for()	阻塞当前线程的过程中，该函数会自动调用 u 锁互斥锁，从而令其他线程使用公共资源。当 过了指定的等待时间（比如 3 秒），该函数会 () 函数对互斥锁加锁，同时令线程继续执行。
wait_until()	和 wait_for() 功能类似，不同之处在于，wai 以设定一个具体时间点（例如 2021年4月8日 间），当条件成立或者等待时间超过了指定的 自动对互斥锁加锁，同时线程继续执行。
notify_one()	向等待队列中的第一个线程线程发送“条件成立
notify_all()	向所有等待的线程发送“条件成立”的信号。

wait函数

```
void wait (unique_lock<mutex>& lck);
```

wait函数，wait（）可依次拆分为三个操作：释放互斥锁、等待条件变量发来信号唤醒当前线程（此时当前线程一直阻塞）、当获得信号时再次获取互斥锁，其有两种用法：
（1）wait（unique_lock &lck）当前线程的执行会被阻塞，直到收到 notify 为止。

（2）wait（unique_lock &lck，Predicate pred）当前线程仅在pred=false时阻塞；如果pred=true时，不阻塞

- 当前线程调用wait()后将被阻塞并且函数会解锁互斥量，直到另外某个线程调用notify_one或者notify_all唤醒当前线程；一旦当前线程获得通知(notify)，wait()函数也是自动调用lock()，同理不 能使用lock_guard对象。
- 如果wait没有第二个参数，第一次调用默认条件不成立，直接解锁互斥量并阻塞到本行，直到某一 个线程调用notify_one或notify_all为止，被唤醒后，wait重新尝试获取互斥量，如果得不到，线程会卡在这里，直到获取到互斥量，然后无条件地继续进行后面的操作。
- 如果wait包含第二个参数，如果第二个参数不满足，那么wait将解锁互斥量并堵塞到本行，直到某 一个线程调用notify_one或notify_all为止，被唤醒后，wait重新尝试获取互斥量，如果得不到，线 程会卡在这里，直到获取到互斥量，然后继续判断第二个参数，如果表达式为false，wait对互斥量解锁，然后休眠，如果为true，则进行后面的操作。

生产者-消费者

```
void produce() {
  mutex_lock(&mutex);
  while (!(count < n)) wait(&cv, &mutex);
  printf("("); count++;
  mutex_unlock(&mutex);
  broadcast(&cv);
}

void consume() {
  mutex_lock(&mutex);
  while (!(count > 0)) wait(&cv, &mutex);
  printf(")"); count--;
  mutex_unlock(&mutex);
  broadcast(&cv);
}
```

```
#include <iostream>
#include <thread>
#include <pthread.h>
#include <mutex> // 锁的头文件
#include <condition_variable> // 条件变量的头文件
#include <list>
using namespace std;
using namespace std::literals::chrono_literals; // c++14 特性
std::mutex g_mutex;
condition_variable g_con;
list<int> products;

void producer() {
    int product_id = 0;
```

```
    while (true) {
        std::unique_lock<std::mutex> lock(g_mutex); // 上锁
        products.push_back(++product_id);
        cout << "[producer]生产者 生产: " << product_id << endl;
        lock.unlock(); // 完成生产解锁
        // 通知消费者消费
        g_con.notify_one(); // 资源已经被释放，通知消费者去使用资源
        if (product_id > 50) {
            break;
        }
        this_thread::sleep_for(2ms);
    }
}
void consumer(){
    while (true) {
        std::unique_lock<std::mutex> lock(g_mutex); // 上锁
        if (products.empty()) {
            cout << "没有产品，等待" << endl;
            // 没有产品消费，wait线程休眠并解锁资源，直到受到通知
            g_con.wait(lock); // 线程等待 释放锁 得到信号后会重新上锁
        } else {
            int product_id = products.front();
            products.pop_front();
            cout << "[consumer]消费者 消费: " << product_id << endl;
            this_thread::sleep_for(2ms);
            if (product_id > 50) break;
        }
    }
}

int main() {
    // 生产者-消费者模式
    std::thread t1(producer);
    consumer();
    t1.join();
    return 0;
}
```

```
没有产品，等待
[producer]生产者 生产: 1
[consumer]消费者 消费: 1
没有产品，等待
[producer]生产者 生产: 2
[consumer]消费者 消费: 2
没有产品，等待
[producer]生产者 生产: 3
```

```
[consumer]消费者 消费: 3
没有产品，等待
[producer]生产者 生产: 4
[consumer]消费者 消费: 4
没有产品，等待
[producer]生产者 生产: 5
[consumer]消费者 消费: 5
没有产品，等待
[producer]生产者 生产: 6
[consumer]消费者 消费: 6
没有产品，等待
[producer]生产者 生产: 7
[consumer]消费者 消费: 7
没有产品，等待
```

生产者占用资源去生产，生产完会释放独占的资源，并通知消费者消费

消费者会去看生产者生产产品没，没有就休眠，等待。得到信号会重新占有资源消费产品。

个人理解： 老妈在厨房作菜，由于厨房面积小，只能一个人活动（锁住资源），做完一道，老妈停工让出厨房，喊儿子吃菜（notify_one）。儿子到厨房吃完后，发现没有新的菜，让出厨房等待下盘菜叫他（wait）。多年后，老妈生了二胎，改成了含儿子们吃菜（notify_all），但只有厨房还是只能活动一个人，只有跑得快的儿子能吃到（资源竞争）。

例程

任务对列

```cpp
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>
#include <chrono>
#include <condition_variable>
#include <functional>

class TaskQueue {
public:
    void push(int task) {
        {
            std::unique_lock<std::mutex> lock(mutex_);
            tasks_.push_back(task);
        }
```

```
        cond_.notify_one();
    }

    int pop() {
        std::unique_lock<std::mutex> lock(mutex_);
        cond_.wait(lock, [this] { return !tasks_.empty(); });
        int task = tasks_.front();
        tasks_.erase(tasks_.begin());
        return task;
    }

private:
    std::vector<int> tasks_;
    std::mutex mutex_;
    std::condition_variable cond_;
};

std::once_flag flag;

void run_once() {
    std::cout << "Run once." << std::endl;
}

void worker(TaskQueue& taskQueue) {
    std::call_once(flag, run_once);

    while (true) {
        int task = taskQueue.pop();

        if (task == -1) {
            break;
        }

        std::cout << "Processing task: " << task << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

int main() {
    TaskQueue taskQueue;
    std::vector<std::thread> threads;

    for (int i = 0; i < 4; ++i) {
        threads.push_back(std::thread(worker, std::ref(taskQueue)));
    }

    for (int i = 0; i < 10; ++i) {
        taskQueue.push(i);
```

```
    }

    for (int i = 0; i < 4; ++i) {
        taskQueue.push(-1);
    }

    for (auto& thread : threads) {
        thread.join();
    }

    return 0;
}
```

在这个示例中，使用了以下组件：

std::mutex用于保护TaskQueue的内部数据结构。
std::unique_lock用于自动上锁和解锁互斥量。
std::condition_variable用于线程间的通知。
std::once_flag和std::call_once组合使用，确保run_once()函数只被执行一次。

环形队列

略