

网络程序设计

多线程编程

5.1 进程和线程

5.1.1 进程

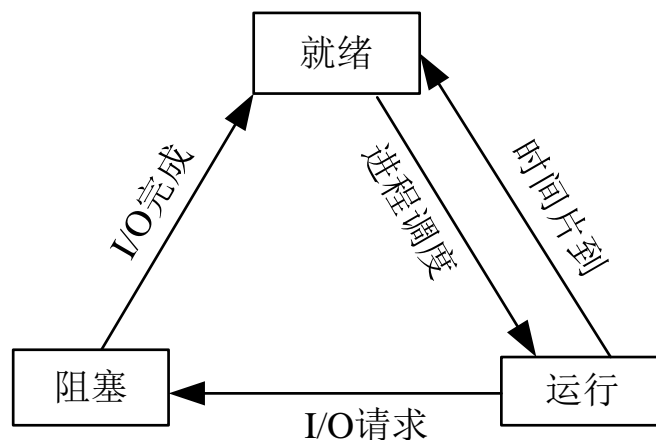
- 进程（**Process**）是现代操作系统理论的核心概念之一。
- 支持多任务并发执行的操作系统需要为多个并发执行的程序合理地分配内存、外设、**CPU**时间等资源，为了便于描述和实现系统中各程序运行过程的独立性、并发性、动态性以及它们相互之间因资源共享而引起的相互制约性，引入了进程的概念。

- 进程是指具有一定独立功能的程序在某个数据集合上的一次运行活动，是系统进行资源分配和调度运行的一个独立单位。
- 简单一点说，进程是程序在计算机上的一次执行活动，当你启动了一个程序，你就启动了一个进程，退出一个程序也就结束了一个进程。
- 在操作系统中引入进程的目，是为了使多个程序能并发执行，以改善系统资源利用率，提高系统吞吐量。

进程的基本状态有三种

- 就绪状态，进程获得了除**CPU**之外的一切所需资源，一旦获得**CPU**即可运行；
- 运行状态，进程获得了**CPU**等一切所需资源，正在**CPU**上运行；
- 阻塞状态，正在**CPU**上运行的进程，由于某种原因不再具备运行的条件而暂时停止运行。比如需要等待**I/O**操作完成、等待其它进程发来消息等。

- 当就绪进程的数目多于**CPU**的数目时，需要按一定的算法动态地将**CPU**分配给就绪进程队列中的某一个使之运行，这就是所谓的进程调度。
- 当分配给某个进程的运行时间（时间片）用完了时进程就会由运行状态回到就绪状态；
- 运行中的进程如果需要执行**I/O**操作，比如从键盘输入数据，就会进入到阻塞状态等待**I/O**操作完成，**I/O**操作完成后，就会转入就绪状态等待下一次调度。



5.1.2 线程

- 当从一个进程切换到另一个进程时，需要保护当前进程的状态（主要是虚拟内存映像，文件描述符，寄存器内容等），并恢复将要运行的进程的状态，这不仅消耗**CPU**的执行时间，还要占用较多的存储空间。
- 为了减少进程切换时的时空开销，使操作系统具有更好的并发性，人们在操作系统中又引入了线程的概念。

- 线程是进程内部的一个执行单元。它只是简单地扩展了**进程切换**的概念，它从进程间的切换转变成了同一个进程内的函数间的切换。
- 同一个进程中函数间的切换相对于进程来说所需的开销要小的多，它只需要保存少数几个寄存器、一个堆栈指针以及程序计数器等少量内容。
- 线程实现了进程内的并发性。在支持线程的操作系统中，一个进程内至少有一个线程，称为**主线程**，它无需由用户去主动创建，是由系统自动创建的。

- 系统创建好进程后，实际上就启动执行了该进程的主线程。
- 进程中除主线程外还可以有多个子个线程。子线程是用户根据需要创建的。
- 多个线程之间可以并发执行（包括主线程和各级子线程），一个线程可以创建和撤消另一个线程。
- 由于线程之间的相互制约，致使线程在运行中呈现出间断性。线程也有就绪、阻塞和运行三种基本状态。

线程的作用

- 同一进程中的多线程通常各自完成不同的工作，可以实现并行处理，避免了某项任务长时间占用CPU时间。
- 比如一个线程负责通过网络收发数据，另一个线程完成所需的计算工作，第三个线程来做文件输入输出，当其中一个由于某种原因阻塞后（比如通过网络收发数据的线程等待对方发送数据），另外的线程仍然能执行而不被阻塞。

- 当线程数目多于计算机的处理器（**CPU**）数目时，为了运行所有这些线程，操作系统为每个独立线程安排一些**CPU**时间，操作系统以轮换方式向线程提供时间片，这就给人一种假象，好象这些线程都在同时运行。
- 线程切换会消耗很多的**CPU**资源，在一定程度上会降低系统的性能。

- 线程自己不独自拥有系统资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。
- 一个进程中的所有线程都在该进程的虚拟地址空间中，共同使用该虚拟地址空间中的全局变量和系统资源，所以线程间的通讯非常方便。

5.1.3 进程与线程的差别

- 进程与线程的主要差别在于，多进程中的每个进程有自己的地址空间(**address space**)，而多线程则共享同一进程的地址空间；
- 进程是除**CPU**外的资源分配的单位，线程主要是执行和调度（**CPU**运行时间分配）的单位。
- 进程主要是资源（**CPU**除外）分配的单位。线程是执行和调度（**CPU**运行时间分配）的单位。

- 线程是进程内部的一个执行单元。每一个进程至少有一个主执行线程，它无需由用户去主动创建，是由系统自动创建的。
- 用户根据需要在应用程序中创建其它线程，多个线程并发地运行于同一个进程中。
- 一个进程中的所有线程都在该进程的虚拟地址空间中，共同使用该虚拟地址空间中的全局变量和系统资源，所以线程间的通讯非常方便。

- 多线程可以实现并行处理，避免了某项任务长时间占用**CPU**时间。
- 当线程数目多于计算机的处理器（**CPU**）数目时，为了运行所有这些线程，操作系统为每个独立线程安排一些**CPU**时间，操作系统以轮换方式向线程提供时间片，这就给人一种假象，好象这些线程都在同时运行。
- 尽管比进程间的切换要好的多，线程间的切换仍会消耗很多的**CPU**资源，在一定程度上也会降低系统的性能。

C++标准线程库

- C++ 11发布之前，C++并没有对多线程编程的专门支持，C++ 11通过标准库引入了对多线程的支持
- C++ 11标准库内部包裹了pthread库，因此，编译程序的时候需要加上-lpthread连接选项.
- 主要头文件
 - #include <thread>
 - #include <condition_variable>
 - #include <mutex>

std::thread类

- 您可以使用**thread**对象来观察和管理应用程序中的线程。
- 使用默认构造函数创建的**thread**对象不与任何线程关联。
- 使用可调用对象构造的**thread**对象将会创建新的线程并在该线程中执行可调用对象。
- **thread**对象可以移动，但不能复制。因此，一个线程只能与一个**thread**对象相关联。

std::thread类

- 每个运行中的线程都有一个类型为thread::id的唯一标识符。
- 函数this_thread::get_id返回当前线程的标识符。
- 成员函数thread::get_id返回由thread对象管理的线程的标识符。
- 对于默认构造的thread对象，thread::get_id方法返回一个对象值都一样。并且与关联线程的thread对象的id不同。
- [详细](#)

thread::id类

- 为进程中的每个执行线程提供唯一标识符。
- 定义:

```
class thread::id {  
    id() noexcept;  
};
```

默认构造函数构造的thread::id对象与正在运行的线程的thread::id对象不相等。

所有默认函数构造的thread::id对象都相等。

每个线程都有一个id，但此处的get_id与系统分配给线程的ID并不一定是同一个东东。如果想取得系统分配的线程ID，可以调用native_handle函数。

thread构造函数

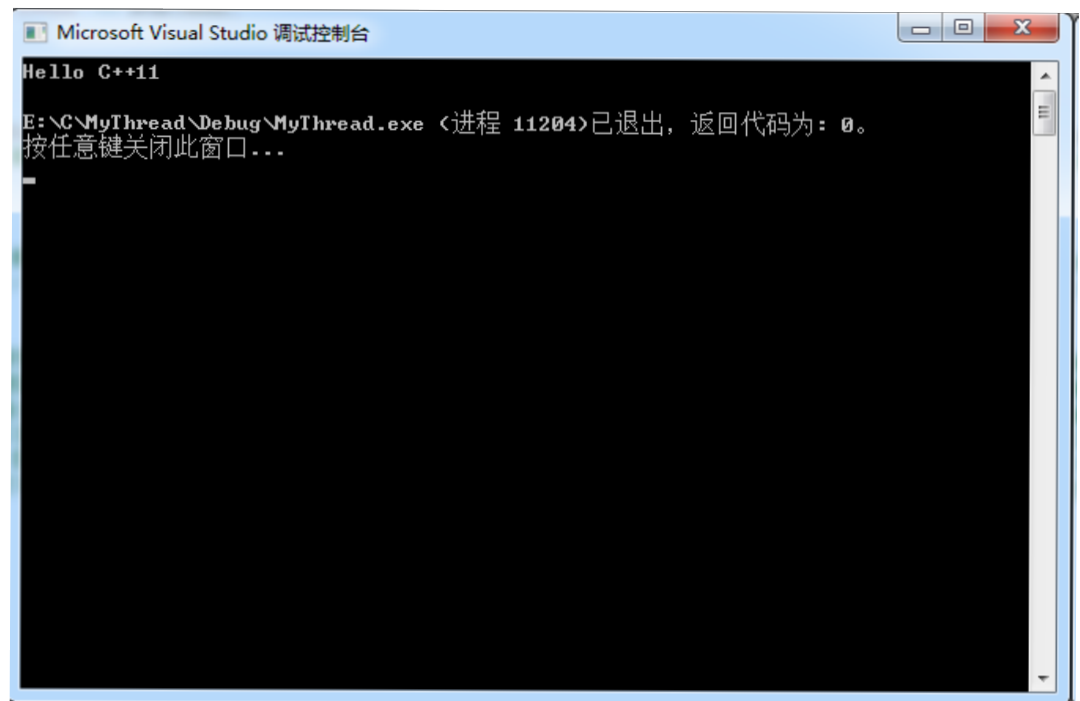
```
thread() noexcept;  
template <class Fn, class... Args>  
explicit thread(Fn&& F, Args&&... A);  
  
thread(thread&& Other) noexcept;
```

- 第一个是默认构造函数
- 第二个是构造运行实例线程,线程在实例成功构造时启动
- 第三个thread对象移动, other变成默认构造状态

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

void foo() {
    std::cout << "Hello C++11" << std::endl;
}

int main()
{
    std::thread thread(foo); // 启动线程foo
    thread.join(); // 等待线程执行完成
    return 0;
}
```

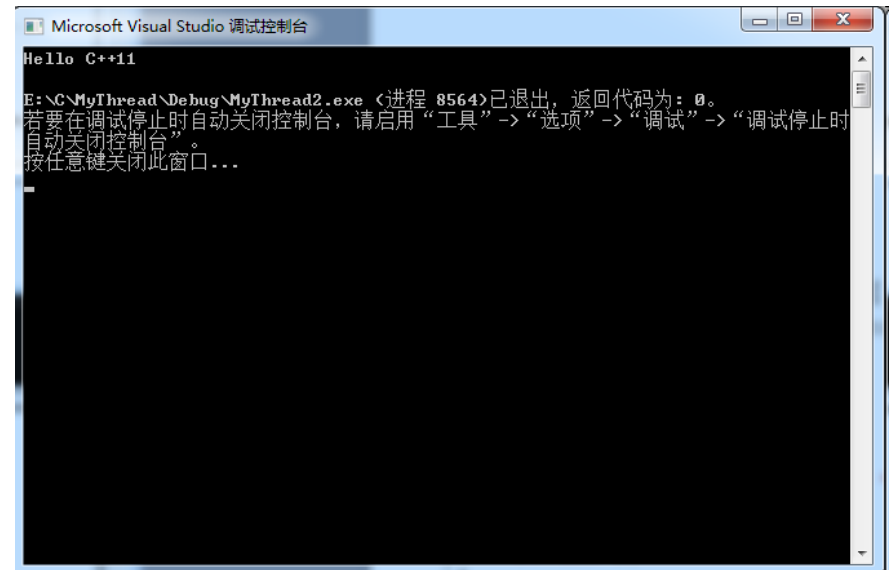


线程参数

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

void hello(const char* name) {
    std::cout << "Hello " << name << std::endl;
}

int main()
{
    std::thread thread(hello, "C++11");
    thread.join();
    return 0;
}
```

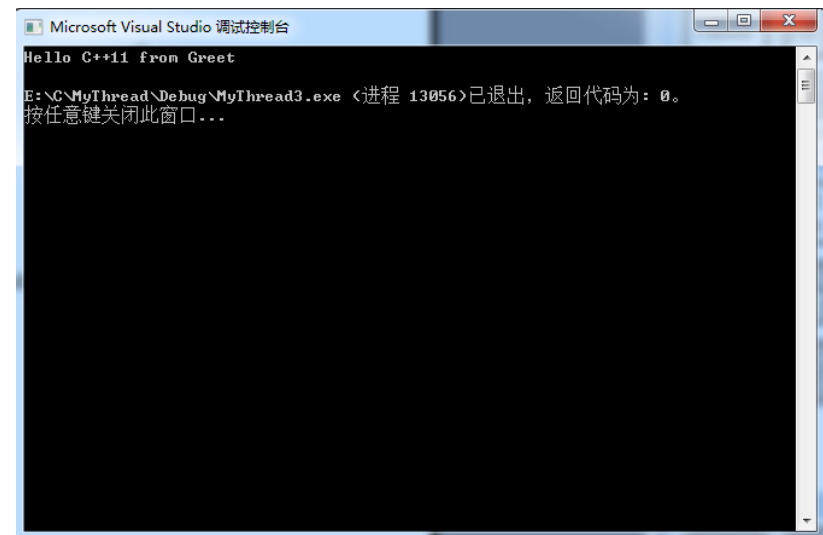


类成员函数做为线程入口

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

class Greet
{
    const char* owner = "Greet";
public:
    void SayHello(const char* name) {
        std::cout << "Hello " << name << " from " << this->owner << std::endl;
    }
};

int main()
{
    Greet greet;
    std::thread thread(&Greet::SayHello, &greet, "C++11");
    thread.join();
    return 0;
}
```



拷贝

```
std::thread a(foo);  
std::thread b;  
b = a;
```

- 当执行以上代码时，会发生什么？最终foo线程是由**a**管理，还是**b**来管理？
- 答案是由**b**来管理。**std::thread**被设计为只能由一个实例来维护线程状态，以及对线程进行操作。因此当发生赋值操作时，会发生线程所有权转移。

std::thread类方法

Requirements

Header: <thread>

Namespace: std

Public Methods

Name	Description
detach	Detaches the associated thread from the thread object.
get_id	Returns the unique identifier of the associated thread.
hardware_concurrency	Static. Returns an estimate of the number of hardware thread contexts.
join	Blocks until the associated thread completes.
joinable	Specifies whether the associated thread is joinable.
native_handle	Returns the implementation-specific type that represents the thread handle.
swap	Swaps the object state with a specified thread object.

[详细](#)

join

- 阻塞当前线程，执行thread对象关联的线程过程直到完成
- 定义：

```
void join();
```

如果成功调用，后续再对这个thread类对象调用get_id就会返回具有唯一性的thread::id
如果调用不成功，则get_id返回值不变

thread::joinable

Specifies whether the associated thread is *joinable*.

C++

```
bool joinable() const noexcept;
```

Return Value

true if the associated thread is *joinable*; otherwise, **false**.

Remarks

A thread object is *joinable* if `get_id() != id()`.

thread::detach

Detaches the associated thread. The operating system becomes responsible for releasing thread resources on termination.

C++

Copy

```
void detach();
```

Remarks

After a call to `detach`, subsequent calls to `get_id` return `id`.

If the thread that's associated with the calling object is not joinable, the function throws a `system_error` that has an error code of `invalid_argument`.

If the thread that's associated with the calling object is invalid, the function throws a `system_error` that has an error code of `no_such_process`.

detach以后就失去了对线程的所有权，不能再调用**join**了，因为线程已经分离出去了，不再归该实例管了。判断线程是否还有对线程的所有权的一个简单方式是调用**joinable**函数，返回**true**则有，否则为无。

小结

- a. C++ 11中创建线程非常简单，使用`std::thread`类就可以，`thread`类定义于`thread`头文件，构造`thread`对象时传入一个可调用对象作为参数（如果可调用对象有参数，把参数同时传入），这样构造完成后，新的线程马上被创建，同时执行该可调用对象；
- b. 用`std::thread`默认的构造函数构造的对象不关联任何线程；判断一个`thread`对象是否关联某个线程，使用`joinable()`接口，如果返回`true`，表明该对象关联着某个线程（即使该线程已经执行结束）；
- c. “joinable”的对象析构前，必须调用`join()`接口等待线程结束，或者调用`detach()`接口解除与线程的关联，否则会抛异常；
- d. 正在执行的线程从关联的对象`detach`后会自主执行直至结束，对应的对象变成不关联任何线程的对象，`joinable()`将返回`false`；
- e. `std::thread`没有拷贝构造函数和拷贝赋值操作符，因此不支持复制操作（但是可以`move`），也就是说，没有两个 `std::thread`对象会表示同一执行线程；
- f. 容易知道，如下几种情况下，`std::thread`对象是不关联任何线程的（对这种对象调用`join`或`detach`接口会抛异常）：
 - 默认构造的`thread`对象；
 - 被移动后的`thread`对象；
 - `detach` 或 `join` 后的`thread`对象；

一个多线程例子

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

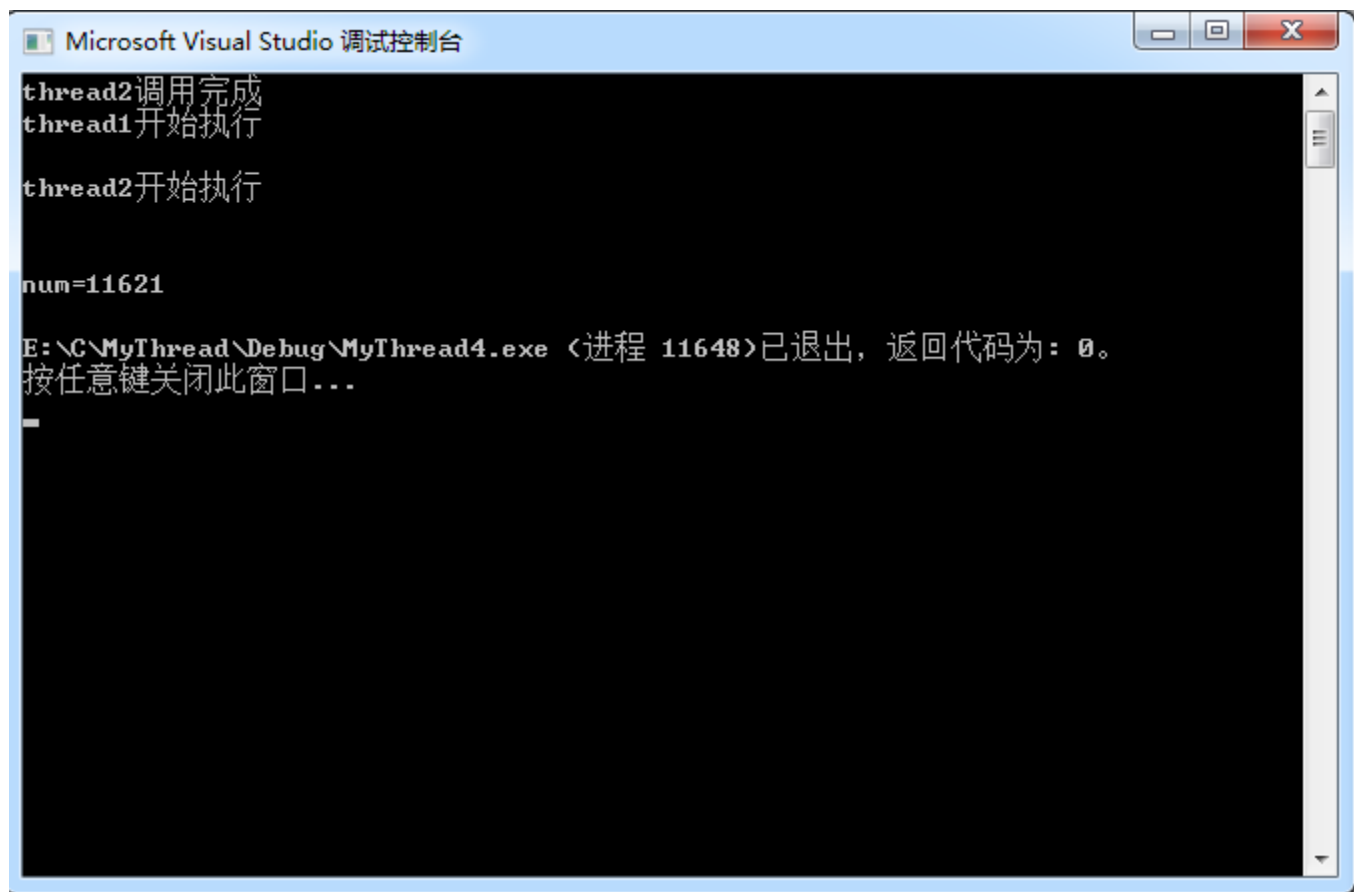
using namespace std::chrono;

int increase(int *p, int times) {
    std::cout << "thread1开始执行\n" << std::endl;
    for (int i = 0; i < times; i++)
        ++* p;
    return 0;
}

int main()
{
    int num = 0;
    //线程1调用increase函数
    std::thread thread1(increase, &num, 10000);
    //线程2调用lambda
    std::thread thread2(
        [&]() {
            std::cout << "thread2开始执行\n" << std::endl;
            for (int i = 0; i < 10000; i++)
                ++num;
        }
    );

    std::cout << "thread2调用完成\n" << std::endl;

    thread1.join(); //等待线程1结束
    thread2.join(); //等待线程2结束
    std::cout << "num=" << num << std::endl;
}
```



```
Microsoft Visual Studio 调试控制台

thread2调用完成
thread1开始执行

thread2开始执行

num=11621

E:\C\MyThread\Debug\MyThread4.exe <进程 11648>已退出, 返回代码为: 0。
按任意键关闭此窗口...
```

这个程序创建了两个线程，分别对变量num进行了10000次++操作，由于两个线程同时运行，++num也没有加锁保护，所以最后的输出结果在10000到20000之间，有一定随机性，也证明了++num不是原子操作；

常用的std::this_thread方法

Public Functions

Name	Description
get_id	Uniquely identifies the current thread of execution.
sleep_for	Blocks the calling thread.
sleep_until	Blocks the calling thread at least until the specified time.
swap	Exchanges the states of two thread objects.
yield	Signals the operating system to run other threads, even if the current thread would ordinarily continue to run.

[详细](#)


线程暂停

- 从外部让线程暂停，会引发很多并发问题。这大概也是**std::thread**并没有直接提供**pause**函数的原因。但有时线程在运行时，确实需要“停顿”一段时间怎么办呢？
- 可以使用**std::this_thread::sleep_for**或**std::this_thread::sleep_until**

sleep_for

Blocks the calling thread.

C++

 Copy

```
template <class Rep,  
class Period>  
inline void sleep_for(const chrono::duration<Rep, Period>& Rel_time);
```

Parameters

Rel_time

A [duration](#) object that specifies a time interval.

Remarks

The function blocks the calling thread for at least the time that's specified by *Rel_time*. This function does not throw any exceptions.

sleep_until

Blocks the calling thread at least until the specified time.

C++

```
template <class Clock, class Duration>
void sleep_until(const chrono::time_point<Clock, Duration>& Abs_time);

void sleep_until(const xtime *Abs_time);
```

Parameters

Abs_time

Represents a point in time.

Remarks

This function does not throw any exceptions.

```
⊞ #include <thread>
    #include <iostream>
    #include <chrono>

    using namespace std::chrono;

⊞ void pausable() {
    // sleep 500毫秒
    std::this_thread::sleep_for(milliseconds(500));
    // sleep 到指定时间点
    std::this_thread::sleep_until(system_clock::now() + milliseconds(500));
}

⊞ int main()
{
    std::thread thread(pausable);
    thread.join();

    return 0;
}
```

yield

Signals the operating system to run other threads, even if the current thread would ordinarily continue to run.

C++

```
inline void yield() noexcept;
```

其作用是当前线程“放弃”执行，让操作系统调度另一线程继续执行

比如说你的线程需要等待某个操作完成，如果你直接用一个循环不断判断这个操作是否完成就会使得这个线程占满CPU时间，这会造成资源浪费。

这时候你可以判断一次操作是否完成，如果没有完成就调用yield交出时间片，过一会儿再来判断是否完成，这样这个线程占用CPU时间会大大减少。

```
while(!isDone()); // Bad  
while(!isDone()) yield(); // Good
```

线程停止

- 一般情况下当线程函数执行完成后，线程“自然”停止。
- 但在`std::thread`中有一种情况会造成线程**异常终止**，那就是：**析构**。
- 当`std::thread`实例析构时，如果线程还在运行，则线程会被强行终止掉，这可能会造成资源的泄漏，因此尽量在析构前`join`一下，以确保线程成功结束。
- 如果确实想提前让线程结束怎么办呢？
- 一个简单的方法是使用“共享变量”，线程定期地去检测该量，如果需要退出，则停止执行，退出线程函数。使用“共享变量”需要注意，在多核、多**CPU**的情况下需要使用“原子”操作

mutex

- **mutex** 又称互斥量，用于提供对共享变量的互斥访问。
- C++11中**mutex**相关的类都在<mutex>头文件中。
- 共四种互斥类：

序号	名称	用途
1	std::mutex	最基本也是最常用的互斥类
2	std::recursive_mutex	同一线程内可递归(重入)的互斥类
3	std::timed_mutex	除具备mutex功能外，还提供了带时限请求锁定的能力
4	std::recursive_timed_mutex	同一线程内可递归(重入)的timed_mutex

与std::thread一样，mutex相关类不支持拷贝构造、不支持赋值。
同时mutex类也不支持move语义(move构造、move赋值)。

[详细](#)

mutex主要函数

- **lock, try_lock, unlock**
 - mutex的标准操作，四个mutex类都支持这些操作，但是不同类在行为上有些微的差异。
- **try_lock_for, try_lock_until**
 - 这两个函数仅用于timed系列的mutex，函数最多会等待指定的时间，如果仍未获得锁，则返回false。除超时设定外，这两个函数与try_lock行为一致。

lock

- 锁住互斥量。调用**lock**时有三种情况：
 - 如果互斥量没有被锁住，则调用线程将该**mutex**锁住，直到调用线程调用**unlock**释放。
 - 如果**mutex**已被其它线程**lock**，则调用线程将被阻塞，直到其它线程**unlock**该**mutex**。
 - 如果当前**mutex**已经被调用者线程锁住，则**std::mutex**死锁，而**recursive**系列则成功返回。

try_lock

- 尝试锁住mutex，调用该函数同样也有三种情况：
 - 如果互斥量没有被锁住，则调用线程将该mutex锁住(返回true)，直到调用线程调用unlock释放。
 - 如果mutex已被其它线程lock，则调用线程将失败，并返回false。
 - 如果当前mutex已经被调用者线程锁住，则std::mutex死锁，而recursive系列则成功返回true。

unlock

- 解锁mutex，释放对mutex的所有权。
- 值得一提的时，对于recursive系列mutex，unlock次数需要与lock次数相同才可以完全解锁。
- If the mutex is not currently locked by the calling thread, it causes *undefined behavior*. (如果mutex未加锁，调用unlock会导致未定义的各种奇怪结果)

```

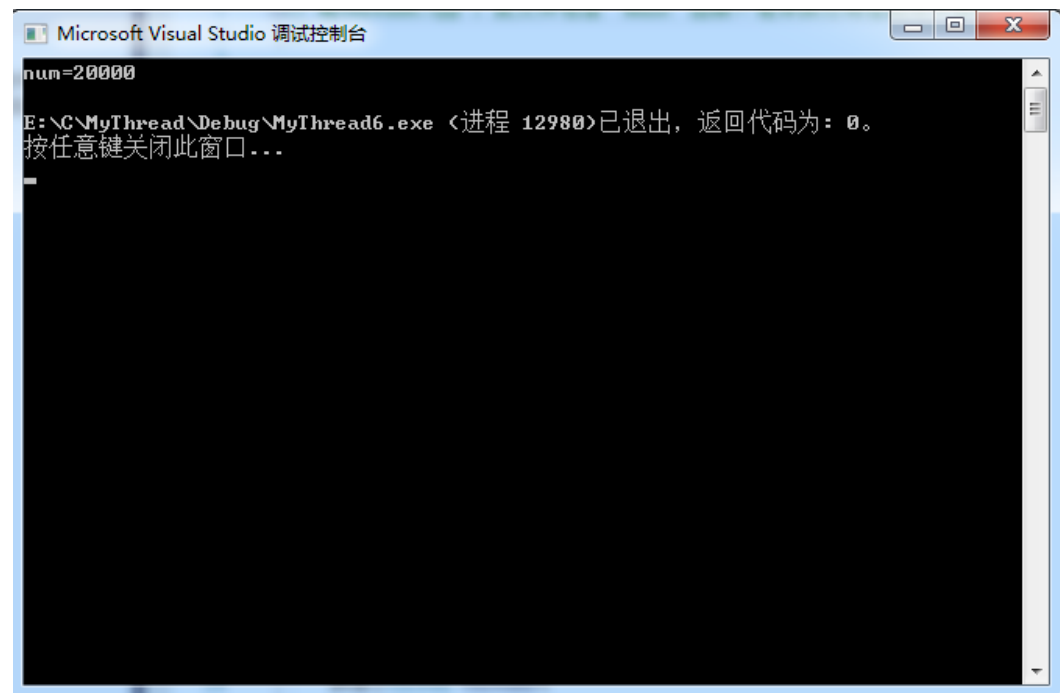
#include <thread>
#include <mutex>
#include <iostream>

int increase(int* p, int times, std::mutex &mtx) {
    for (int i = 0; i < times; i++) {
        mtx.lock();
        ++* p;
        mtx.unlock();
    }
    return 0;
}

int main()
{
    int num = 0;
    std::mutex mtx;
    //线程1调用increase函数
    std::thread thread1(
        [&]() {
            increase(&num, 10000, mtx);
        }
    );
    //线程2调用lambda
    std::thread thread2(
        [&]() {
            for (int i = 0; i < 10000; i++) {
                mtx.lock();
                ++num;
                mtx.unlock();
            }
        }
    );

    thread1.join(); //等待线程1结束
    thread2.join(); //等待线程2结束
    std::cout << "num=" << num << std::endl;
}

```



try_lock_for, try_lock_until

- 这两个函数仅用于timed系列的mutex (std::timed_mutex, std::recursive_timed_mutex),
- 函数最多会等待指定的时间，如果仍未获得锁，则返回**false**。
- 除超时设定外，这两个函数与try_lock行为一致。

```
// 等待指定时长
template <class Rep, class Period>
    try_lock_for(const chrono::duration<Rep, Period>& rel_time);
// 等待到指定时间
template <class Clock, class Duration>
    try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
```

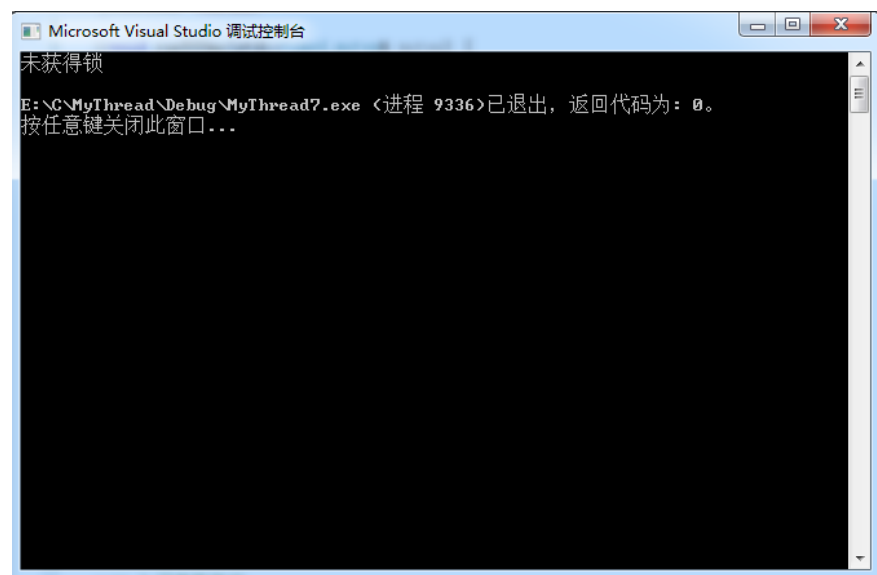
```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

void run500ms(std::timed_mutex& mutex) {
    //auto这里是变量类型自动推断
    auto _500ms = std::chrono::milliseconds(500);
    if (mutex.try_lock_for(_500ms)) {
        std::cout << "获得了锁" << std::endl;
    }
    else {
        std::cout << "未获得锁" << std::endl;
    }
}

int main()
{
    std::timed_mutex mutex;

    mutex.lock();
    std::thread thread(run500ms, std::ref(mutex));
    thread.join();
    mutex.unlock();

    return 0;
}
```



lock_guard类

- lock_guard利用了C++ RAII的特性，在构造函数中上锁，析构函数中解锁。lock_guard是一个模板类，其原型为

```
template <class Mutex>                explicit lock_guard(mutex_type& Mtx);  
class lock_guard;                      lock_guard(mutex_type& Mtx, adopt_lock_t);
```

模板参数Mutex代表互斥量，可以是std::mutex, std::timed_mutex, std::recursive_mutex, std::recursive_timed_mutex中的任何一个，也可以是std::unique_lock，这些都提供了lock和unlock的能力。

lock_guard仅用于上锁、解锁，不对mutex承担供任何生命周期的管理，因此在使用的时候，**请确保lock_guard管理的mutex一直有效。**

同其它mutex类一样，**lock_guard不允许拷贝，即拷贝构造和赋值函数被声明为delete。**

lock_guard的设计保证了即使程序在锁定期间发生了异常，也会安全的释放锁，不会发生死锁。

```

1  #include <iostream>
2  #include <mutex>
3
4  std::mutex mutex;
5
6  void safe_thread() {
7      try {
8          std::lock_guard<std::mutex> _guard(mutex);
9          throw std::logic_error("logic error");
10     } catch (std::exception &ex) {
11         std::cerr << "[caught] " << ex.what() << std::endl;
12     }
13 }
14 int main() {
15     safe_thread();
16     // 此处仍能上锁
17     mutex.lock();
18     std::cout << "OK, still locked" << std::endl;
19     mutex.unlock();
20
21     return 0;
22 }

```

程序输出

```

1  [caught] logic error
2  OK, still locked

```

unique_lock类

- lock_guard提供了简单上锁、解锁操作，但当我们更需要更灵活的操作时便无能为力了。这些就需要unique_lock上场了。
- unique_lock拥有对Mutex的所有权，一旦初始化了unique_lock，它就接管了该mutex，在unique_lock结束生命周期前(析构前)，其它地方就不要再直接使用该mutex了。

```
template <class Mutex>  
class unique_lock;
```

[详细](#)

unique_lock Constructor

Constructs a `unique_lock` object.

C++

```
unique_lock() noexcept;
unique_lock(unique_lock&& Other) noexcept;
explicit unique_lock(mutex_type& Mtx);

unique_lock(mutex_type& Mtx, adopt_lock_t Adopt);

unique_lock(mutex_type& Mtx, defer_lock_t Defer) noexcept;
unique_lock(mutex_type& Mtx, try_to_lock_t Try);

template <class Rep, class Period>
unique_lock(mutex_type& Mtx,
            const chrono::duration<Rep, Period>
            Rel_time);

template <class Clock, class Duration>
unique_lock(mutex_type& Mtx,
            const chrono::time_point<Clock, Duration>
            Abs_time);

unique_lock(mutex_type& Mtx,
            const xtime* Abs_time) noexcept;
```

[其他详细](#)

Remarks

The first constructor constructs an object that has an associated mutex pointer value of 0.

The second constructor moves the associated mutex status from *Other*. After the move, *Other* is no longer associated with a mutex.

The remaining constructors store & *Mtx* as the stored `mutex` pointer. Ownership of the `mutex` is determined by the second argument, if it exists.

Remarks

The first constructor constructs an object that has an associated mutex pointer value of 0.

The second constructor moves the associated mutex status from *Other*. After the move, *Other* is no longer associated with a mutex.

The remaining constructors store `& Mtx` as the stored `mutex` pointer. Ownership of the `mutex` is determined by the second argument, if it exists.

No argument	Ownership is obtained by calling the <code>lock</code> method on the associated <code>mutex</code> object.
Adopt	Ownership is assumed. <code>Mtx</code> must be locked when the constructor is called.
Defer	The calling thread is assumed not to own the <code>mutex</code> object. <code>Mtx</code> must not be locked when the constructor is called.
Try	Ownership is determined by calling <code>try_lock</code> on the associated <code>mutex</code> object. The constructor throws nothing.
Rel_time	Ownership is determined by calling <code>try_lock_for(Rel_time)</code> .
Abs_time	Ownership is determined by calling <code>try_lock_until(Abs_time)</code> .

~unique_lock Destructor

Releases any resources that are associated with the `unique_lock` object.

C++

```
~unique_lock() noexcept;
```

Remarks

If the calling thread owns the associated `mutex`, the destructor releases ownership by calling `unlock` on the `mutex` object.

unique_lock类的主要方法

Public Methods

Name	Description
<code>lock</code>	Blocks the calling thread until the thread obtains ownership of the associated <code>mutex</code> .
<code>mutex</code>	Retrieves the stored pointer to the associated <code>mutex</code> .
<code>owns_lock</code>	Specifies whether the calling thread owns the associated <code>mutex</code> .
<code>release</code>	Disassociates the <code>unique_lock</code> object from the associated <code>mutex</code> object.
<code>swap</code>	Swaps the associated <code>mutex</code> and ownership status with that of a specified object.
<code>try_lock</code>	Attempts to obtain ownership of the associated <code>mutex</code> without blocking.
<code>try_lock_for</code>	Attempts to obtain ownership of the associated <code>mutex</code> without blocking.
<code>try_lock_until</code>	Attempts to obtain ownership of the associated <code>mutex</code> without blocking.
<code>unlock</code>	Releases ownership of the associated <code>mutex</code> .

std::call_once

- 该函数的作用顾名思义：保证call_once调用的函数只被执行一次。
- 该函数需要与std::once_flag配合使用。
- std::once_flag被设计为对外封闭的，即外部没有任何渠道可以改变once_flag的值，仅可以通过std::call_once函数修改。

[详细](#)

```

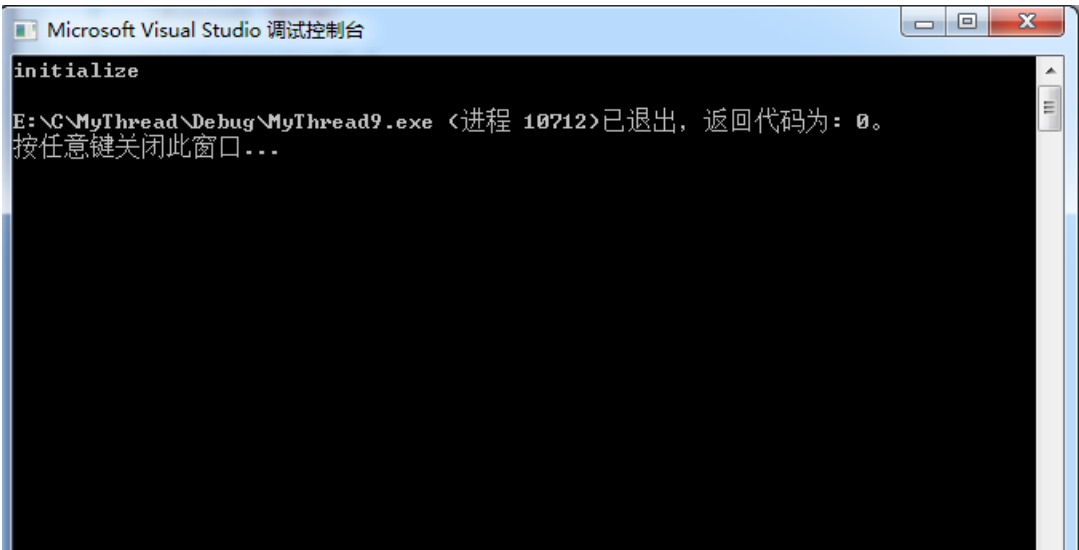
#include <iostream>
#include <thread>
#include <mutex>

void initialize() {
    //__FUNCTION__用来获取当前函数名
    std::cout << __FUNCTION__ << std::endl;
}

std::once_flag of;
void my_thread() {
    std::call_once(of, initialize);
}

int main()
{
    std::thread threads[10];
    //C++11特性，基于范围的for循环
    //默认遍历是只读的，若要可写需用&(即用引用而不是赋值 )
    for (std::thread& thr : threads) {
        thr = std::thread(my_thread);
    }
    for (std::thread& thr : threads) {
        thr.join();
    }
    return 0;
}

```



<condition_variable>头文件结构

Classes	description
condition_variable	提供了std::unique_lock 相关联的条件变量
condition_variable_any	提供与任何锁类型相关联的条件变量

Enum classes	description
cv_status	列出在条件变量上限时等待的可能结果(枚举)

Functions	description
notify_all_at_thread_exit	当这个线程完全完成(函数)时，调度调用notify_all来调用

[详细](#)

condition_variable类

- 条件变量类是一个同步原语，它可以用来阻塞一个线程或多个线程，直到另一个线程同时修改一个共享变量(条件)，并通知条件变量。
- 条件变量是能够阻塞调用线程的对象，直到通知恢复为止。
- 它使用**unique_lock**(在互斥锁上)来锁定线程，当它的一个**wait**函数被调用时。线程被阻塞，直到被另一个线程唤醒，该线程调用同一个条件变量对象上的**notify**函数。
- 类型条件变量的对象总是使用**unique_lock < mutex >**等待:对于可以使用任何类型的可锁定类型的选项，参见[condition_variable_any](#)。

condition_variable类成员

Members

Constructors

<code>condition_variable</code>	Constructs a <code>condition_variable</code> object.
---------------------------------	--

Functions

<code>native_handle</code>	Returns the implementation-specific type representing the <code>condition_variable</code> handle.
----------------------------	---

<code>notify_all</code>	Unblocks all threads that are waiting for the <code>condition_variable</code> object.
-------------------------	---

<code>notify_one</code>	Unblocks one of the threads that are waiting for the <code>condition_variable</code> object.
-------------------------	--

<code>wait</code>	Blocks a thread.
-------------------	------------------

<code>wait_for</code>	Blocks a thread, and sets a time interval after which the thread unblocks.
-----------------------	--

<code>wait_until</code>	Blocks a thread, and sets a maximum point in time at which the thread unblocks.
-------------------------	---


[详细](#)

构造函数

condition_variable

Constructs a `condition_variable` object.

C++

 Copy

```
condition_variable();
```


Remarks

If not enough memory is available, the constructor throws a [system_error](#) object that has a `not_enough_memory` error code. If the object cannot be constructed because some other resource is not available, the constructor throws a `system_error` object that has a `resource_unavailable_try_again` error code.

wait

Blocks a thread.

C++

 Copy

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

template <class Predicate>
void wait(unique_lock<mutex>& Lck, Predicate Pred);
```

Parameters

Lck

A [unique_lock<mutex>](#) object.

Pred


Any expression that returns **true** or **false**.

Remarks

The first method blocks until the `condition_variable` object is signaled by a call to [notify_one](#) or [notify_all](#). It can also wake up spuriously.

In effect, the second method executes the following code.

C++

 Copy

```
while(!Pred())
    wait(Lck);
```

wait函数

- 1、当前线程的执行(该线程将锁定lck的互斥锁)将被阻塞，直到被通知。
- 2、在阻塞线程的时刻，该函数将自动调用lck.unlock()，允许其他锁定的线程继续运行。
- 3、一旦notify(显式地，通过其他线程)，函数就解除阻塞状态并调用lck.lock()，在调用函数时将lck留在相同的状态。然后函数返回(注意，最后一个互斥锁可能会在返回之前阻塞线程)。
- 4、一般情况下，函数会被另一个线程的调用唤醒，无论是对成员notify_one还是成员notify_all。但是某些实现可能产生虚假的唤醒调用，而不需要调用这些函数。因此，该函数的用户将确保满足恢复的条件。
- 5、定义(2)如果指定pred是个返回值为bool型的函数指针,如果pred返回false，则函数只会阻塞，并且只有当它变为true时，notify才能解除线程(这对于检查伪唤醒调用特别有用)。如果pred返回true则不阻塞。

wait函数执行过程

- `lck.unlock()`, 释放对锁的占用
- 阻塞
- 收到`notify`消息
- `lck.lock()`, 重新占用锁（此时可能阻塞，等待其它进程`unlock`释放锁）

例1

```
#include <iostream>           // std::cout
#include <thread>               // std::thread
#include <mutex>                 // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable
```

```
std::mutex mtx;
std::condition_variable cv;
bool ready = false;
```

```
void print_id(int id) {
    std::unique_lock<std::mutex> lck(mtx);
    while (!ready) cv.wait(lck);
    // ...
    std::cout << "thread " << id << '\n';
    //unique_lock析构函数调用mtx的unlock
}
```

```
void go() {
    std::unique_lock<std::mutex> lck(mtx);
    ready = true;
    cv.notify_all();
    //unique_lock析构函数调用mtx的unlock
}
```

```
int main()
{
    std::thread threads[10];
    // spawn 10 threads:
    for (int i = 0; i < 10; ++i)
        threads[i] = std::thread(print_id, i);

    std::cout << "10 threads ready to race...\n";
    go();           // go!

    for (auto& th : threads) th.join();

    return 0;
}
```

Microsoft Visual Studio 调试控制台

10 threads ready to race...

thread 9
thread 8
thread 5
thread 4
thread 1
thread 7
thread 0
thread 3
thread 6
thread 2

E:\C\MyThread\Debug\MyThread10.exe <进程 8072>已退出, 返回代码为
按任意键关闭此窗口...

例2

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    // Wait until main() sends data
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, [] {return ready; });

    // after the wait, we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // Send data back to main()
    processed = true;
    std::cout << "Worker thread signals data processing completed\n";

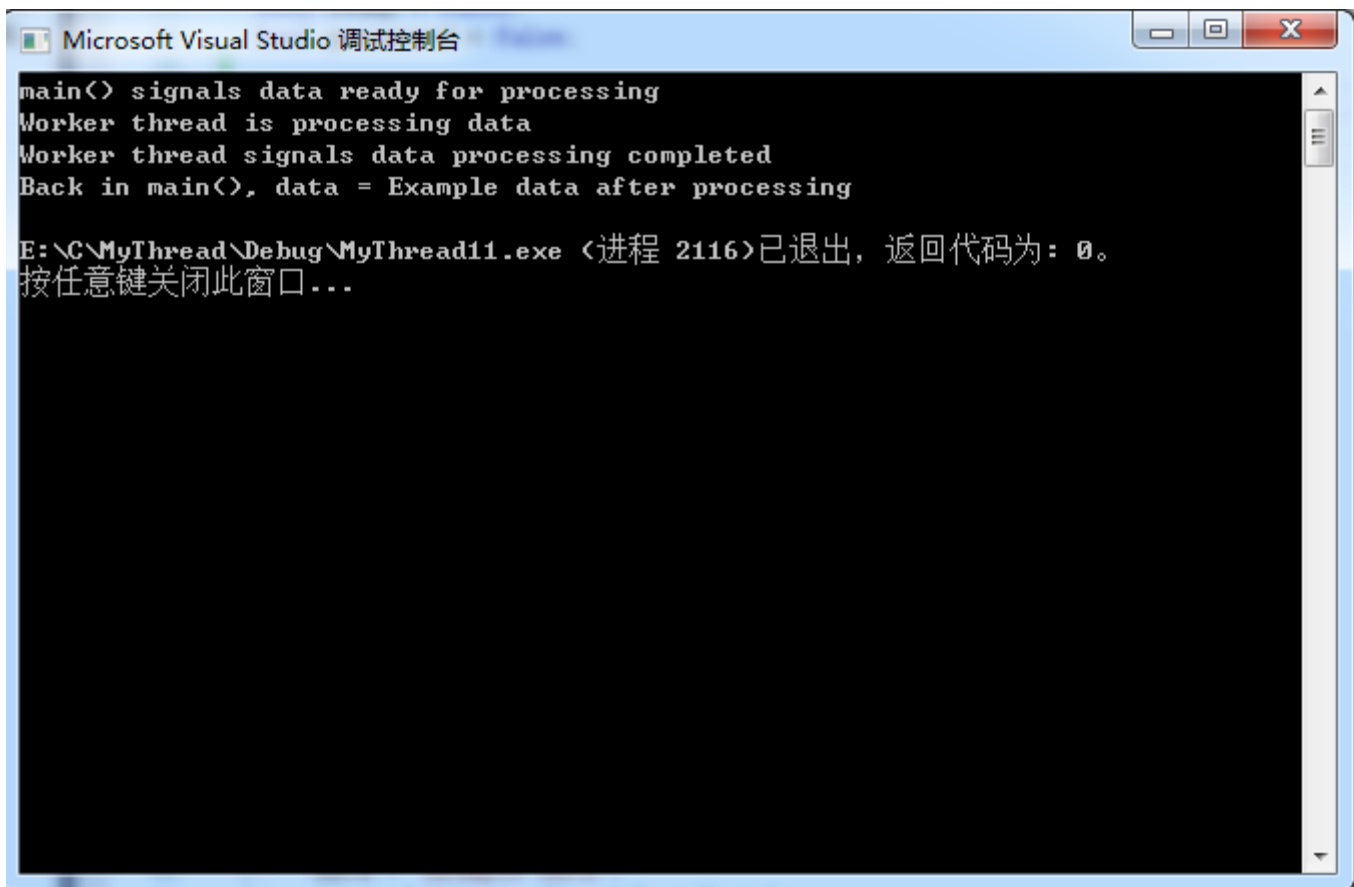
    // Manual unlocking is done before notifying, to avoid waking up
    // the waiting thread only to block again (see notify_one for details)
    lk.unlock();
    cv.notify_one();
}
```

```
int main()
{
    std::thread worker(worker_thread);

    data = "Example data";
    // send data to the worker thread
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, [] {return processed; });
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
    return 0;
}
```



The image shows a screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio 调试控制台" and standard Windows window controls (minimize, maximize, close). The main area is a black console with white text. The output shows the following sequence of events: the main thread signals data ready for processing, a worker thread starts processing, the worker thread signals completion, and the main thread receives the data. Finally, the program exits with a return code of 0.

```
main() signals data ready for processing
Worker thread is processing data
Worker thread signals data processing completed
Back in main(), data = Example data after processing

E:\C\MyThread\Debug\MyThread11.exe <进程 2116>已退出，返回代码为：0。
按任意键关闭此窗口...
```


wait_for

Blocks a thread, and sets a time interval after which the thread unblocks.

C++

```
template <class Rep, class Period>
cv_status wait_for(
    unique_lock<mutex>& Lck,
    const chrono::duration<Rep, Period>& Rel_time);

template <class Rep, class Period, class Predicate>
bool wait_for(
    unique_lock<mutex>& Lck,
    const chrono::duration<Rep, Period>& Rel_time,
    Predicate Pred);
```

Parameters

Lck

A [`unique_lock<mutex>`](#) object.

Rel_time

A `chrono::duration` object that specifies the amount of time before the thread wakes up.

Pred

Any expression that returns **true** or **false**.

Return Value

The first method returns `cv_status::timeout` if the wait terminates when *Rel_time* has elapsed. Otherwise, the method returns `cv_status::no_timeout`.

The second method returns the value of *Pred*.

wait_for函数

- 1、当前执行线程在`rel_time`时间内或者在被`notify`通知之前(该线程将锁定`lck`的互斥锁)被阻塞(如果后者先发生的话)。
- 2、在阻塞线程的时刻，该函数将自动调用`lck.unlock()`，允许其他的线程使用互斥锁。
- 3、一旦通知或一次`rel_time`时间段已经过了，函数将会解除阻塞并调用`lck.lock()`，将`lck`与调用函数时的状态保持一致。然后函数返回(注意，最后一个互斥锁可能会在返回之前阻塞线程)。
- 4、一般情况下，函数会被另一个线程的调用唤醒，无论是对成员`notify_one`还是成员`notify_all`。但是某些实现可能产生虚假的唤醒调用，而不需要调用这些函数。因此，该函数的用户将确保满足恢复的条件。
- 5、如果指定`pred`(定义2)，如果`pred`返回`false`，则函数只会阻塞，并且只有当它变为`true`时，通知才能解除线程(这对于检查伪唤醒调用特别有用)。
- 6.定义1，返回值是`cv_status::timeout`或`cv_status::no_timeout`
- 7.定义2，返回`bool`值，`true`表示由`notify`解除阻塞，`false`表示超时解除阻塞

```

#include <iostream>           // std::cout
#include <thread>              // std::thread
#include <chrono>              // std::chrono::seconds
#include <mutex>               // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable, std::cv_status

std::condition_variable cv;
int value;

void read_value() {
    std::cin >> value;
    cv.notify_one();
}

int main()
{
    std::cout << "Please, enter an integer (I'll be printing dots): \n";
    std::thread th(read_value);

    std::mutex mtx;
    std::unique_lock<std::mutex> lck(mtx);
    while (cv.wait_for(lck, std::chrono::seconds(1)) == std::cv_status::timeout) {
        //因为超时唤醒，就输出.
        std::cout << '.' << std::endl;
    }
    //因为notify唤醒则输出value
    std::cout << "You entered: " << value << '\n';
    th.join();

    return 0;
}

```



```
Microsoft Visual Studio 调试控制台

Please, enter an integer <I'll be printing dots>:
.
.
.
.
123457.

You entered: 123457

E:\C\MyThread\Debug\MyThread12.exe <进程 6972>已退出, 返回代码为: 0。
按任意键关闭此窗口...
```

```

#include <iostream>
#include <atomic>
#include <condition_variable>
#include <thread>
#include <chrono>
//using namespace std::chrono_literals;

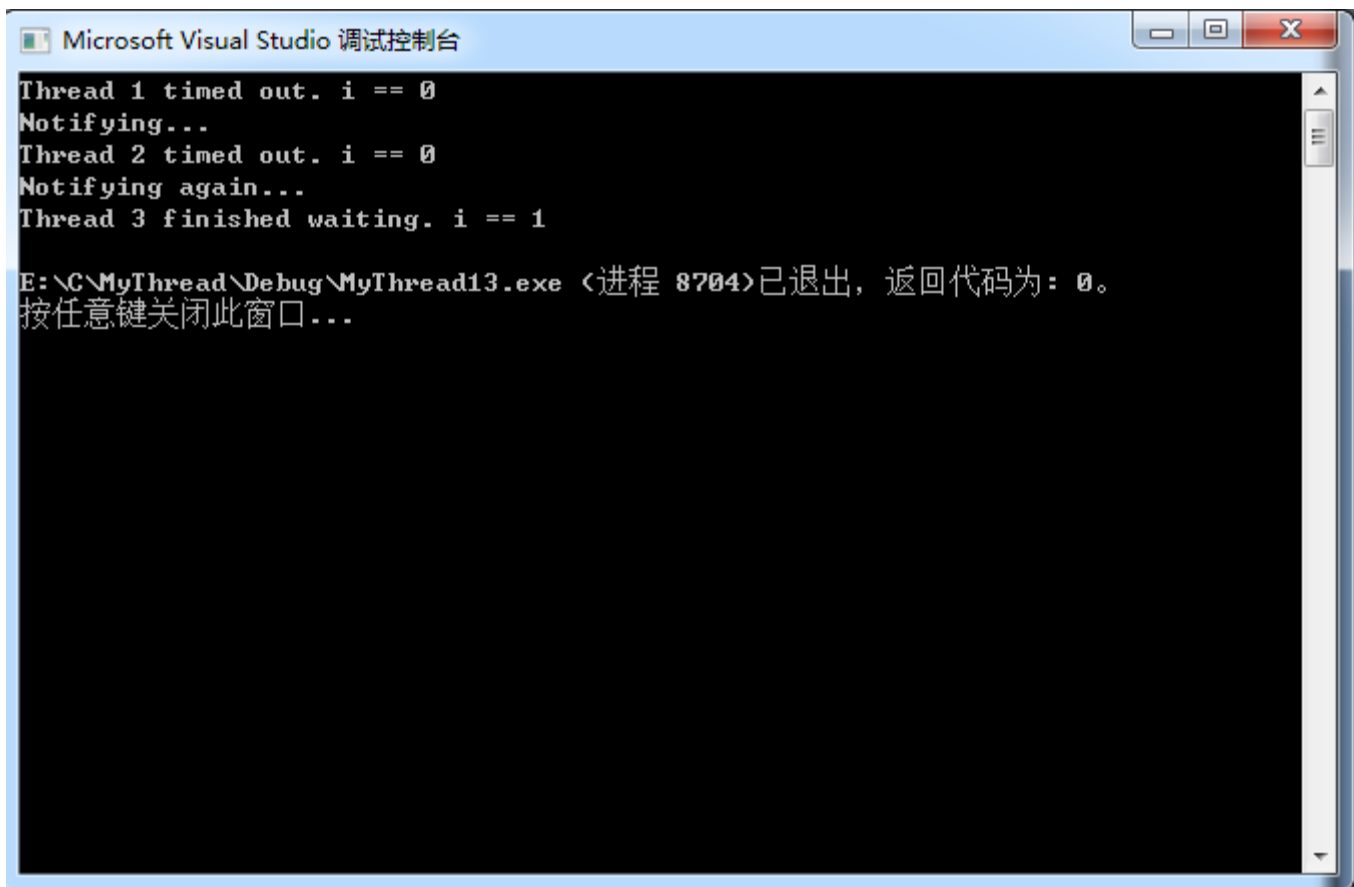
std::condition_variable cv;
std::mutex cv_m;
int i;

void waits(int idx)
{
    std::unique_lock<std::mutex> lk(cv_m);
    if (cv.wait_for(lk, idx * std::chrono::milliseconds(100), [] {return i == 1; }))
        std::cerr << "Thread " << idx << " finished waiting. i == " << i << '\n';
    else
        std::cerr << "Thread " << idx << " timed out. i == " << i << '\n';
}

void signals()
{
    std::this_thread::sleep_for(std::chrono::milliseconds(120));
    std::cerr << "Notifying...\n";
    cv.notify_all();
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    {
        std::lock_guard<std::mutex> lk(cv_m);
        i = 1;
    }
    std::cerr << "Notifying again...\n";
    cv.notify_all();
}

int main()
{
    std::thread t1(waits, 1), t2(waits, 2), t3(waits, 3), t4(signals);
    t1.join(); t2.join(); t3.join(); t4.join();
    return 0;
}

```



The image shows a screenshot of the 'Microsoft Visual Studio 调试控制台' (Debug Console) window. The window has a light blue title bar with standard Windows window controls (minimize, maximize, close). The main area is a black console with white text. The text shows the execution of a program with three threads. Thread 1 and Thread 2 both timed out when `i == 0`, and the program notified them. Thread 3 finished waiting when `i == 1`. At the bottom, a message indicates that the process `E:\C\MyThread\Debug\MyThread13.exe` (PID 8704) has exited with a return code of 0, and prompts the user to press any key to close the window.

```
Microsoft Visual Studio 调试控制台


Thread 1 timed out. i == 0
Notifying...
Thread 2 timed out. i == 0
Notifying again...
Thread 3 finished waiting. i == 1

E:\C\MyThread\Debug\MyThread13.exe <进程 8704>已退出, 返回代码为: 0。
按任意键关闭此窗口...
```

wait_until

Blocks a thread, and sets a maximum point in time at which the thread unblocks.

C++

 Copy

```
template <class Clock, class Duration>
cv_status wait_until(
    unique_lock<mutex>& Lck,
    const chrono::time_point<Clock, Duration>& Abs_time);

template <class Clock, class Duration, class Predicate>
bool wait_until(
    unique_lock<mutex>& Lck,
    const chrono::time_point<Clock, Duration>& Abs_time,
    Predicate Pred);

cv_status wait_until(
    unique_lock<mutex>& Lck,
    const xtime* Abs_time);

template <class Predicate>
bool wait_until(
    unique_lock<mutex>& Lck,
    const xtime* Abs_time,
    Predicate Pred);
```

Parameters

Lck

A [unique_lock<mutex>](#) object.

Abs_time

A [chrono::time_point](#) object.

Pred

Any expression that returns **true** or **false**.

[链接](#)

Wait_until

- 1、当前线程的执行(该线程将锁定**lck**的互斥锁)会被阻塞，直到被**notify**通知或到达**abs_time**时间点，无论那一个第一次发生。
- 2、在线程还处在阻塞时，该函数将自动调用**lck.unlock()**，允许其他的线程使用互斥锁。
- 3、一旦被通知或一旦**abs_time**时间到了，函数就会解除阻塞状态并调用**lck.lock()**。在调用函数时将**lck**与函数的状态保持在同一状态。然后函数返回(注意，最后一个互斥锁可能会在返回之前阻塞线程)。
- 4、一般情况下，函数会被另一个线程的调用唤醒，无论是对成员**notify_one**还是成员**notify_all**。但是某些实现可能产生虚假的唤醒调用，而不需要调用这些函数。因此，该函数的使用者将要确保满足恢复的条件。
- 5、(定义2)如果指定**pred**，如果**pred**返回**false**，则函数只会阻塞，并且只有当它变为**true**时，通知才能解除线程(这对于检查伪唤醒调用特别有用)。


```

#include <iostream>
#include <atomic>
#include <condition_variable>
#include <thread>
#include <chrono>
//using namespace std::chrono_literals;

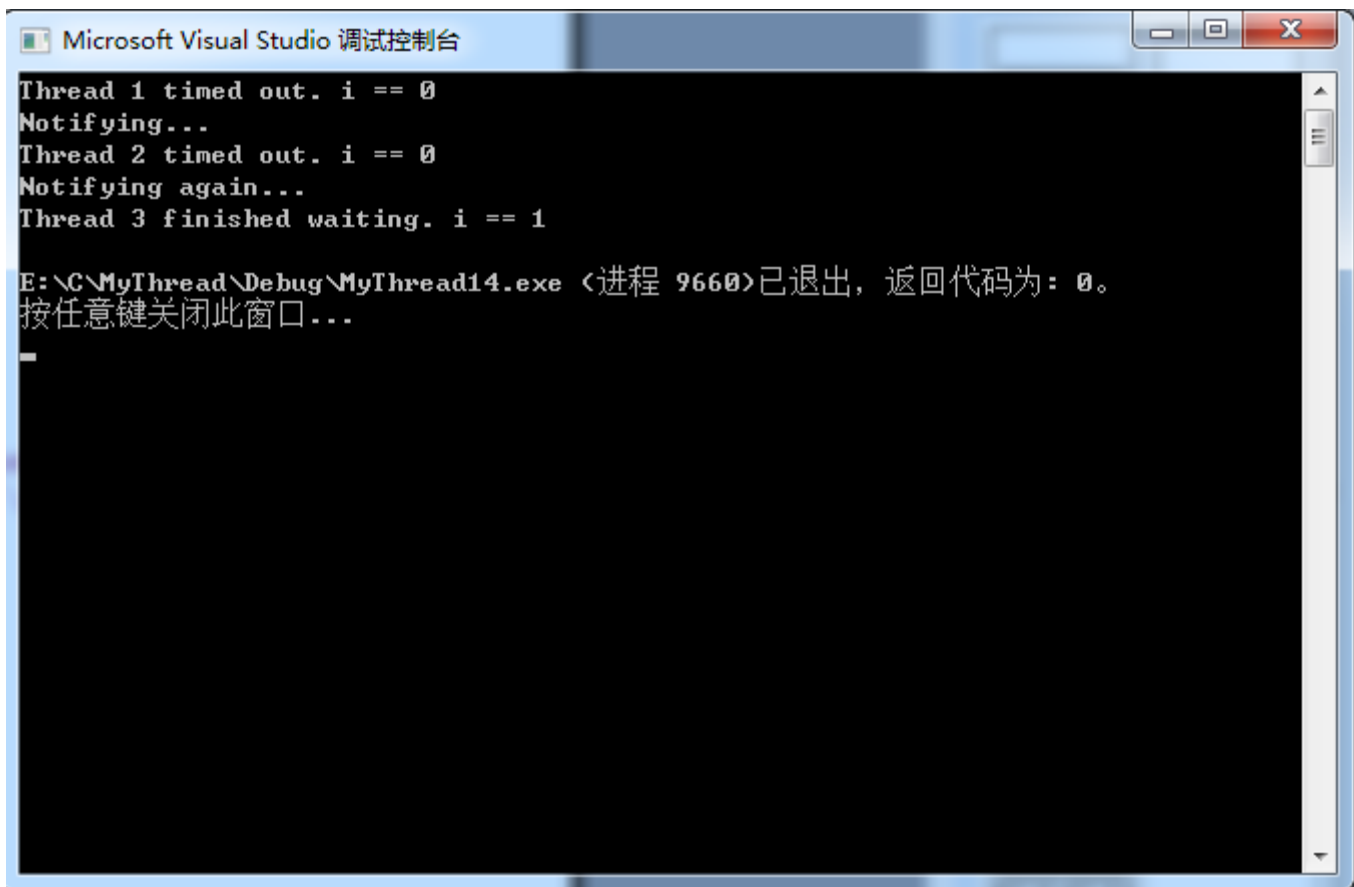
std::condition_variable cv;
std::mutex cv_m;
std::atomic<int> i{ 0 };

void waits(int idx)
{
    std::unique_lock<std::mutex> lk(cv_m);
    auto now = std::chrono::system_clock::now();
    if (cv.wait_until(lk, now + idx * std::chrono::milliseconds(100), []() {return i == 1; }))
        std::cerr << "Thread " << idx << " finished waiting. i == " << i << '\n';
    else
        std::cerr << "Thread " << idx << " timed out. i == " << i << '\n';
}

void signals()
{
    std::this_thread::sleep_for(std::chrono::milliseconds(120));
    std::cerr << "Notifying...\n";
    cv.notify_all();
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    i = 1;
    std::cerr << "Notifying again...\n";
    cv.notify_all();
}

int main()
{
    std::thread t1(waits, 1), t2(waits, 2), t3(waits, 3), t4(signals);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    return 0;
}

```



The image shows a screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio 调试控制台" and standard Windows window controls (minimize, maximize, close). The console output is as follows:

```
Thread 1 timed out. i == 0  
Notifying...  
Thread 2 timed out. i == 0  
Notifying again...  
Thread 3 finished waiting. i == 1  
  
E:\C\MyThread\Debug\MyThread14.exe <进程 9660>已退出, 返回代码为: 0。  
按任意键关闭此窗口...
```

The output indicates that three threads were executed. Thread 1 and Thread 2 timed out when `i` was 0, and Thread 3 finished waiting when `i` was 1. The application then exited with a return code of 0.

notify_all

Unblocks all threads that are waiting for the `condition_variable` object.

C++

```
void notify_all() noexcept;
```

notify_one

Unblocks one of the threads that are waiting on the `condition_variable` object.

C++

```
void notify_one() noexcept;
```

notify_one

- 1、打开当前等待该条件的一个线程。如果没有线程在等待，则该函数什么都不做。如果多于一个，则不指定哪个线程被选中。
- 2、该指令针对这个单独的条件变量。这使得`notify_one()`不可能被延迟。

notify_all

- 解锁当前等待该条件的所有线程。如果没有线程在等待，则该函数什么都不做。
- 注意：
 - 1、*notify_one()/notify_all()*和*wait()/wait_for()/wait_until()* 三个原子部分中的每一个影响都在一个完全的顺序(解锁+等待, 唤醒, 锁)中, 可以被视为一个原子变量的修改顺序: 该命令是特定于这个单独的条件变量的。

condition_variable_any类

- 1、与条件变量一样，除了它的等待函数可以将任何可锁类型作为参数(条件变量对象只能使用`unique_lock < mutex >`)。除此之外，它们是相同的。
- 2、`std::condition_variable_any`是`std::condition_variable`的泛化。而`std::condition_variable`只能用于`std::unique_lock < std::mutex >`，`condition_variable_any`可以操作任何满足基本要求的锁。参见`std::condition_variable`，用于描述条件变量的语义。类`std::condition_variable_any`是一个标准布局类。它不可拷贝构造的，移动可构造的，不可拷贝复制和移动复制。
- 3、如果锁是`std::unique_lock`，`std::condition_variable`可以提供更好的性能。

condition_variable_any类

Members

Constructors

`condition_variable_any`

Constructs a `condition_variable_any` object.

Functions

`notify_all`

Unblocks all threads that are waiting for the `condition_variable_any` object.

`notify_one`

Unblocks one of the threads that are waiting for the `condition_variable_any` object.

`wait`

Blocks a thread.

`wait_for`

Blocks a thread, and sets a time interval after which the thread unblocks.

`wait_until`


Blocks a thread, and sets a maximum point in time at which the thread unblocks.

[详细](#)

wait

Blocks a thread.

C++

 Copy

```
template <class Lock>
void wait(Lock& Lck);

template <class Lock, class Predicate>
void wait(Lock& Lck, Predicate Pred);
```

Parameters

Lck

A `mutex` object of any type.

Pred


Any expression that returns **true** or **false**.

Remarks

The first method blocks until the `condition_variable_any` object is signaled by a call to [notify_one](#) or [notify_all](#). It can also wake up spuriously.

The second method in effect executes the following code.

C++

 Copy

```
while (!Pred())
    wait(Lck);
```



```

#include <iostream>           // std::cout
#include <thread>              // std::thread, std::this_thread::yield
#include <mutex>               // std::mutex
#include <condition_variable> // std::condition_variable_any

std::mutex mtx;
std::condition_variable_any cv;

int cargo = 0;
bool shipment_available() { return cargo != 0; }

void consume(int n) {
    for (int i = 0; i < n; ++i) {
        mtx.lock();
        cv.wait(mtx, shipment_available);
        // consume:
        std::cout << cargo << '\n';
        cargo = 0;
        mtx.unlock();
    }
}

int main()
{
    std::thread consumer_thread(consume, 10);

    // produce 10 items when needed:
    for (int i = 0; i < 10; ++i) {
        while (shipment_available()) std::this_thread::yield();
        mtx.lock();
        cargo = i + 1;
        cv.notify_one();
        mtx.unlock();
    }

    consumer_thread.join();

    return 0;
}

```

Microsoft Visual Studio 调试控制台

1
2
3
4
5
6
7
8
9
10

E:\C\MyThread\Debug\MyThread15.exe <进程 10140>已退出
按任意键关闭此窗口...

std::notify_all_at_thread_exit

```
void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);
```

- 当调用线程退出时，等待在cond上的所有线程都被通知恢复执行。
- 该函数还获得由lk管理的mutex对象上的锁的所有权，该对象在内部由函数存储，并在线程退出时解锁(仅在通知所有线程之前)，行为如下：
- 1.lk.unlock();
2 cond.notify_all();
- 注意：
如果lock. mutex()没有被当前线程锁定，则调用此函数是未定义的行为。

```

#include <iostream>           // std::cout
#include <thread>              // std::thread
#include <mutex>               // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void print_id(int id) {
    std::unique_lock<std::mutex> lck(mtx);
    while (!ready) cv.wait(lck);
    // ...
    std::cout << "thread " << id << '\n';
}

void go() {
    std::unique_lock<std::mutex> lck(mtx);
    std::notify_all_at_thread_exit(cv, std::move(lck));
    ready = true;
}

int main()
{
    std::thread threads[10];
    // spawn 10 threads:
    for (int i = 0; i < 10; ++i)
        threads[i] = std::thread(print_id, i);
    std::cout << "10 threads ready to race...\n";

    std::thread(go).detach(); // go!

    for (auto& th : threads) th.join();

    return 0;
}

```

Microsoft Visual Studio 调试控制台

10 threads ready to race...

thread 8

thread 4

thread 0

thread 9

thread 5

thread 1

thread 6

thread 2

thread 7

thread 3

E:\C\MyThread\Debug\MyThread16.exe <进程 8152>已退出，
按任意键关闭此窗口...

多线程的其他工具

- std::promise类
- std::future类
- std::packaged_task类
- std::atomic_flag结构体
- std::atomic结构体

使用Win32API函数实现多线程

- Win32 提供了一系列的API函数来完成线程的创建、挂起、恢复、终结以及通信等工作。下面将选取其中的一些重要函数进行说明。

(1) CreateThread 函数

- 函数格式

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    DWORD dwStackSize,  
    LPTHREAD_START_ROUTINE lpStartAddress,  
    LPVOID lpParameter,  
    DWORD dwCreationFlags,  
    LPDWORD lpThreadId);
```

- 该函数在其调用进程的进程空间里创建一个新的线程，并返回已建线程的句柄。如果创建成功则返回线程的句柄，否则返回NULL。

- 函数参数说明

- lpThreadAttributes: 指向一个 **SECURITY_ATTRIBUTES** 结构的指针，该结构决定了线程的安全属性，一般置为 **NULL**;
- dwStackSize: 指定线程的堆栈深度，一般设置为0;
- lpStartAddress: 线程起始地址，通常为线程函数名，其类型 **LPTHREAD_START_ROUTINE** 是一个函数指针：
`typedef unsigned long (_stdcall *LPTHREAD_START_ROUTINE) (void * lpParameter);`

- lpParameter: 线程函数的参数，其类型 **LPVOID** 的定义为: `typedef void * LPVOID;`
- dwCreationFlags: 控制线程创建的附加标志。该参数为0，则线程在被创建后立即开始执行；如果该参数为**CREATE_SUSPENDED**，则创建线程后该线程处于挂起状态，直至函数 **ResumeThread** 被调用；
- lpThreadId: 该参数返回所创建线程的ID。

SuspendThread函数

- **DWORD SuspendThread(HANDLE hThread);**
 - 该函数用于挂起指定的线程，如果函数执行成功，则线程的执行被终止。
- **DWORD ResumeThread(HANDLE hThread);**
 - 该函数用于结束线程的挂起状态，执行线程。
- **VOID ExitThread(DWORD dwExitCode);**
 - 该函数用于线程终结自身的执行，主要在线程的执行函数中被调用。
 - 参数**dwExitCode**用来设置线程的退出码。
 - 线程退出码可用**GetExitCodeThread**获得

- **BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);**
 - 各参数含义如下：
 - **hThread**: 将被终结的线程的句柄；
 - **dwExitCode**: 用于指定线程的退出码。
- 一般情况下，线程运行结束之后，线程函数正常返回，但是应用程序可以调用**TerminateThread**强行终止某一线程的执行。
- 使用**TerminateThread()**终止某个线程的执行是不安全的，可能会引起系统不稳定；虽然该函数立即终止线程的执行，但并不释放线程所占用的资源。因此，一般不建议使用该函数。

多线程编程的步骤

1. 编写线程函数

- 所有线程必须从一个指定的函数开始执行，该函数称为线程函数，它必须具有下列原型：

**DWORD ThreadFunc(LPVOID
lpvThreadParm);**

- 该函数输入一个**LPVOID**型的参数，可以是一个**DWORD**型的整数，也可以是一个指向一个缓冲区的指针， 返回一个**DWORD**型的值。

2. 创建一个线程

- 一个进程的主线程是由操作系统自动生成，如果要想让一个主线程创建额外的线程，可以调用**CreateThread**函数完成。

5.2.3 使用MFC类库实现多线程

- MFC的CWinThread类封装了Windows API的多线程机制，每个CWinThread对象都代表一个线程。
- MFC中的线程包含两种类型，一种是用户界面线程，也称用户接口线程，另一种是工作线程。
- 两种线程用于满足不同任务的处理需求。

- 工作线程适用于那些不要求用户输入并且比较消耗时间的任务。
- 工作线程编程较为简单，设计思路与Win32 **SDK**的多线程编程基本一致，首先编写线程函数，然后创建并启动线程。

- 创建并启动线程需要调用**AfxBeginThread()**，该函数有两个版本，一个用于创建工作线程，另一个版本用来创建用户界面线程。用于创建工作线程的版本的格式如下：

```
CWinThread* AfxBeginThread(  
    AFX_THREADPROC pfnThreadProc,  
    LPVOID pParam,  
    int nPriority=THREAD_PRIORITY_NORMAL,  
    UINT nStackSize = 0,  
    DWORD dwCreateFlags = 0,
```

- **pfnThreadProc**: 是线程函数。线程函数一般具有如下形式:
`UINT ThreadFunction(LPVOID pParam)`
- **pParam** 是传递给线程函数的参数。
- **nPriority**: 是线程的优先级, 缺省是 `THREAD_PRIORITY_NORMAL`, 若为0, 则使用创建线程的优先级, 除此之外, 其可选值还包括 `THREAD_PRIORITY_LOWEST`、`THREAD_PRIORITY_HIGHEST`、`THREAD_PRIORITY_IDLE`。
- **nStackSize**说明了线程的堆栈尺寸, 若为0则堆栈尺寸与创建线程相同。
- **dwCreateFlags**: 线程创建时的标志, 指定线程的初始状态, 如果为 `CREATE_SUSPENDED`, 则线程在创建后就被挂起, 调用 `ResumeThread` 函数后线程继续运行, 如果为0, 那么线程在创建后立即执行。

- 返回值为CWinThread类对象指针，它的成员变量 m_hThread为线程句柄。程序应该把AfxBeginThread返回的CWinThread指针保存起来，以便对创建的线程进行控制。

- 用户界面线程通常用来处理用户输入输出产生的消息和事件，通常用于编写具有多个线程并且每个线程又都需要有用户接口的应用程序。
- 通常MFC应用程序的主线程是CWinApp派生类的对象，是由MFC的应用程序向导（MFC APP Wizard）自动创建的，该对象就是用户界面接口线程的一个典型例子。
- CWinApp是从CWinThread类派生出来的。在编写其他的用户界面线程时也必须编写CWinThread类的派生类，该过程可使用“类向导”（ClassWizard）来完成。

- 要创建并启动用户界面线程，首先就要借助 **ClassWizard** 创建一个派生于**CWinThread**的新的线程类。
- 创建完新的线程类后就可以使用**MFC** 提供的**AfxBeginThread**函数另一个版本来创建并启动用户界面线程了。**AfxBeginThread**函数另一版本的原型为：

```
CWinThread* AfxBeginThread(  
    CRuntimeClass* pThreadClass,  
    int nPriority = THREAD_PRIORITY_NORMAL,  
    UINT nStackSize = 0,  
    DWORD dwCreateFlags = 0,  
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL  
);
```

- 参数pThreadClass是一个指向CRuntimeClass（运行时类）的对象指针，该对象是用RUNTIME_CLASS宏从CWinThread的派生类创建的。RUNTIME_CLASS宏可以从一个CObject的派生类返回一个指向CRuntimeClass对象的指针，其格式如下：

RUNTIME_CLASS（ClassName）

- ClassName是一个CObject派生类的类名，要求该派生类中必须是使用了宏DECLARE_DYNAMIC、DECLARE_DYNCREATE或DECLARE_SERIAL。
- 其它参数以及函数的返回值与前面第一个版本的AfxBeginThread是一样的。

例5.1

```
#include "windows.h"
#include "iostream"
using namespace std;
//定义线程函数，该线程函数无参数
void ThreadFun1()
{
    int i;
    for(i=1;i<100;i++)
    {
        Sleep(1000); //阻塞1000毫秒
        cout <<i<< ",This is Thread 1\n";
    }
}
```

```
HANDLE hThread1; DWORD ThreadID1;
int main(int argc, char* argv[])
{
    int j;
    hThread1=CreateThread(NULL,0,(LPTHREAD_START
    _ROUTINE)ThreadFun1, NULL, 0, &ThreadID1);
    for(j=1;j<10;j++)
    {
        Sleep(1000);
        cout<<j<<" ,This is MainThread!\n";
    }
    return 0;
}
```

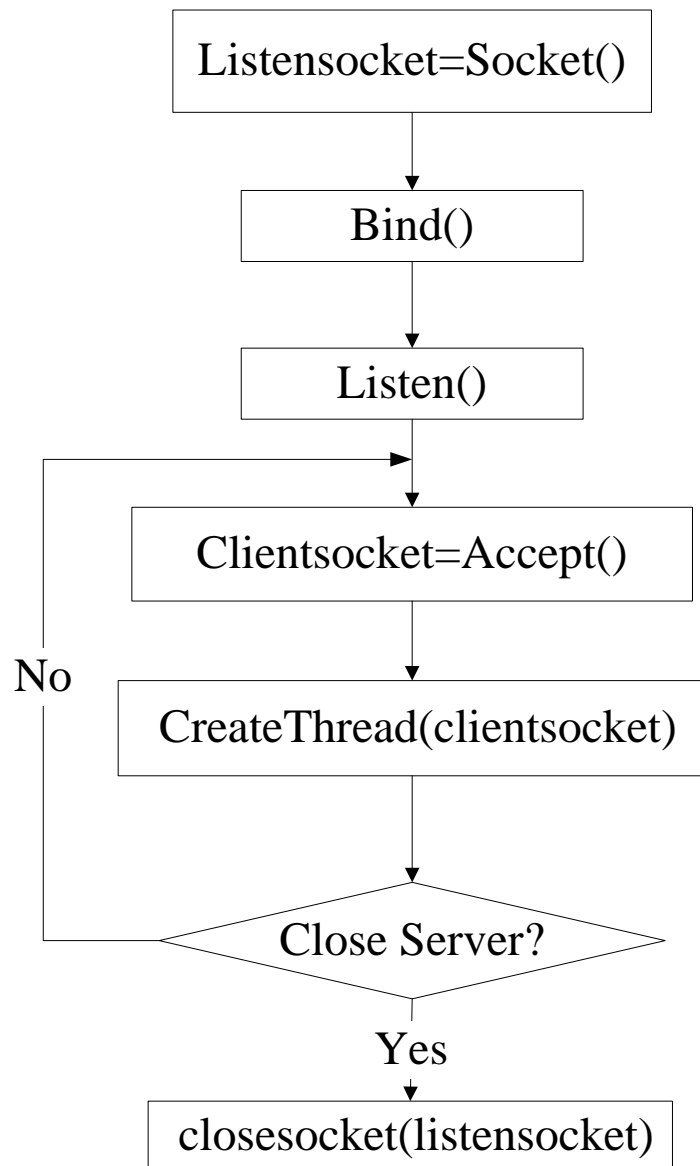
线程函数的参数传递

- 创建线程时可以给线程传递参数。
- 由**CreateThread**函数原形可以看出，线程函数可以有一个**void** 指针类型参数，该参数值在创建线程时由**CreateThread**函数的第四个参数传入。
- 能直接传递的参数个数值只有一个，如果需要传送的值为多个，可定义为结构体。

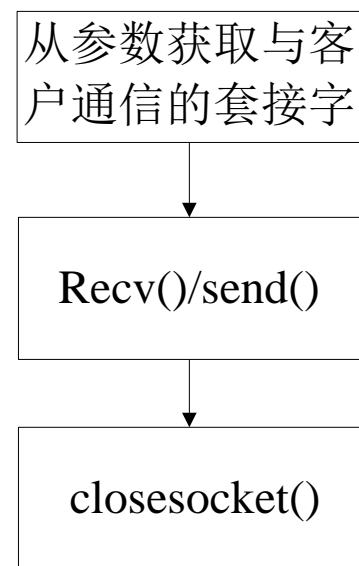
5.3 TCP服务器端程序的多线程编程

- 通常服务器可同时为多客户提供服务，即可同时与多个客户机保持通信。
- 前边我们已学的服务器端程序的编写方法并不支持这一功能。
- 解决问题的方法是采用多线程或多进程技术。本课程只介绍采用多线程的方法。

主程序



线程函数



一个服务器端多线程网络通信的例子

服务器端代码

MTPServer.cpp

```
// MTPSever.cpp : 此文件包含“main”函数。程序执行将在此处开始并结束。
```

```
//
```

```
#include <WinSock2.h>
```

```
#include <ws2tcpip.h>
```

```
#include <iostream>
```

```
#include <thread>
```

```
#include <string.h>
```

```
#include "ListenThread.h"
```

```
#pragma comment(lib, "Ws2_32.lib")
```

```
//自定义的默认端口号,缓冲区长度
```

```
#define DEFAULT_PORT "27015"
```

```
int main()
```

```
{
```

```
    WSADATA wsaData; //用于Winsock初始化的数据结构
```

```
    int iResult; //执行结果返回值。为0表示成功
```

```
    struct addrinfo* result = NULL, hints; //地址信息结构
```

```
    SOCKET listenSocket = INVALID_SOCKET; //用来监听的套接字
```

```
    ListenThread listen_thread; //监听线程对象
```

```
    char cmd[10]; //命令
```

```
    //CString strCmd="";
```

```
    //初始化套接字
```

```
    iResult = WSASStartup(MAKEWORD(2, 2), &wsaData);
```

```
    if (iResult != 0) {
```

```
        printf("WSASStartup执行失败, 错误码: %d\n", iResult);
```

```
        return 1;
```

```
    }
```

```
//解析服务器地址和端口号
ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE; //当地址无法解析时就置为0.0.0.0, 便于bind
iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
if (iResult != 0) {
    printf("getaddrinfo执行错误, 错误码: %d\n", iResult);
    WSACleanup();
    return 1;
}
//为面向连接的服务器创建套接字
listenSocket = socket(result->ai_family, (int)result->ai_socktype, result->ai_protocol);
if (listenSocket == INVALID_SOCKET) {
    printf("socket执行错误, 错误码: %d\n", WSAGetLastError());
    freeaddrinfo(result);
    WSACleanup();
    return 1;
}
//为套接字绑定地址和端口号
iResult = bind(listenSocket, result->ai_addr, (int)result->ai_addrlen);
if (iResult == SOCKET_ERROR) {
    printf("bind失败, 错误码: %d\n", WSAGetLastError());
    freeaddrinfo(result);
    closesocket(listenSocket);
    WSACleanup();
    return 1;
}
freeaddrinfo(result);
```

```

//监听连接请求
iResult = listen(listenSocket, SOMAXCONN);
if (iResult == SOCKET_ERROR) {
    printf("listen失败, 错误码:%d\n", WSAGetLastError());
    closesocket(listenSocket);
    WSACleanup();
    return 1;
}

//开启监听线程, 接受客户请求
std::thread thread(&ListenThread::startServer, &listen_thread, listenSocket);
//等待输入结束指令
do {
    system("cls");
    std::cout << "输入close结束服务器运行 • \n";
    std::cin >> cmd;
} while (strcmp(cmd, "close"));
std::cout << "正在停止服务.....\n";
//停止服务器
listen_thread.stopServer();
//等待服务程序停止
thread.join();

//已经不需监听了, 释放监听套接字
closesocket(listenSocket);
WSACleanup();

std::cout << "服务已经停止!结束服务程序 • \n";
return 0;
}

```

ListenThread.h

```
#pragma once
#include <WinSock2.h>
#include <vector>
#include "ClientThread.h"

class ListenThread
{
public:
    void startServer(SOCKET listenSocket);
    void stopServer();
private:
    bool running = true;
    SOCKET listenSocket; //监听套接字
    SOCKET clientSocket = INVALID_SOCKET; //用来和客户端通信的套接字
    std::vector<ClientThread> threadsList; //线程集合
};
```

ListenThread.cpp

```
#include <stdio.h>
#include <thread>
#include "ListenThread.h"

void ListenThread::startServer(SOCKET listenSocket)
{
    running = true;
    this->listenSocket = listenSocket;
    while (running) {
        //接受客户端的连接请求，返回连接套接字
        clientSocket = accept(listenSocket, NULL, NULL);
        if (clientSocket == INVALID_SOCKET) {
            printf("accept失败，错误码:%d\n", WSAGetLastError());
            closesocket(listenSocket);
        }
        else { //客户端连接成功
            //运行客户端线程
            ClientThread* ct=new ClientThread();
            std::thread thread(&ClientThread::process_msg, ct, clientSocket, std::ref(threadsList));
            thread.detach(); //避免thread对象自动析构导致abort() has been called报错
        }
    }
}
```



```
void ListenThread::stopServer()
{
    running = false;
    //关闭监听套接字
    shutdown(listenSocket, SD_BOTH);
    closesocket(listenSocket);
    //关闭客户端线程
    int count = threadsList.size();
    for (int i = count - 1; i >= 0; i--) {
        threadsList[i].shut();
    }
}
```

ClientThread.h

```
#pragma once
#include <WinSock2.h>
#include <vector>
class ClientThread
{
public:
    void process_msg(SOCKET socket, std::vector<ClientThread>& threadsList);
    void shut();
    bool operator==(ClientThread Tmp);

private:
    void end_process_msg( std::vector<ClientThread>& threadsList);
    SOCKET socket;
};
```

```

#include "ClientThread.h"
#include <algorithm>

//自定义的默认端口号,缓冲区长度
#define DEFAULT_PORT "27015"
#define DEFAULT_BUFLen 512

void ClientThread::process_msg(SOCKET socket, std::vector<ClientThread>& threadsList)
{
    int iResult; //执行结果返回值. 为0表示成功
    char recvbuf[DEFAULT_BUFLen]; //接收数据缓冲区
    int recvbuflen = DEFAULT_BUFLen; //接收数据缓冲区的长度
    int iSendResult; //调用send时, 成功发送数据的字节数

    //存储套接字
    this->socket = socket;
    //将客户通信对象存储到列表, 注意this是指针
    threadsList.push_back(*this);
    //持续接收数据, 直到对方关闭连接
    do {
        iResult = recv(socket, recvbuf, recvbuflen, 0);
        if (iResult > 0) {
            //成功收到数据
            printf("接收字节数: %d\n", iResult);
            //将缓冲区的内容回送给客户端
            iSendResult = send(socket, recvbuf, iResult, 0);
            if (iSendResult == SOCKET_ERROR) {
                printf("send失败, 错误码: %d\n", WSAGetLastError());
                end_process_msg(threadsList); //结束处理
                return;
            }
            printf("发送字节数: %d\n", iSendResult);
        }
        else if (iResult == 0) {
            //连接关闭
            printf("客户端连接关闭...\n");
        }
        else {
            //接收发生错误
            printf("recv失败, 错误码: %d\n", WSAGetLastError());
            end_process_msg(threadsList); //结束处理
            return;
        }
    } while (iResult > 0);
    //关闭连接
    iResult = shutdown(socket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
        printf("shutdown失败, 错误码: %d\n", WSAGetLastError());
    }
    end_process_msg(threadsList); //结束处理
    return;
}

```

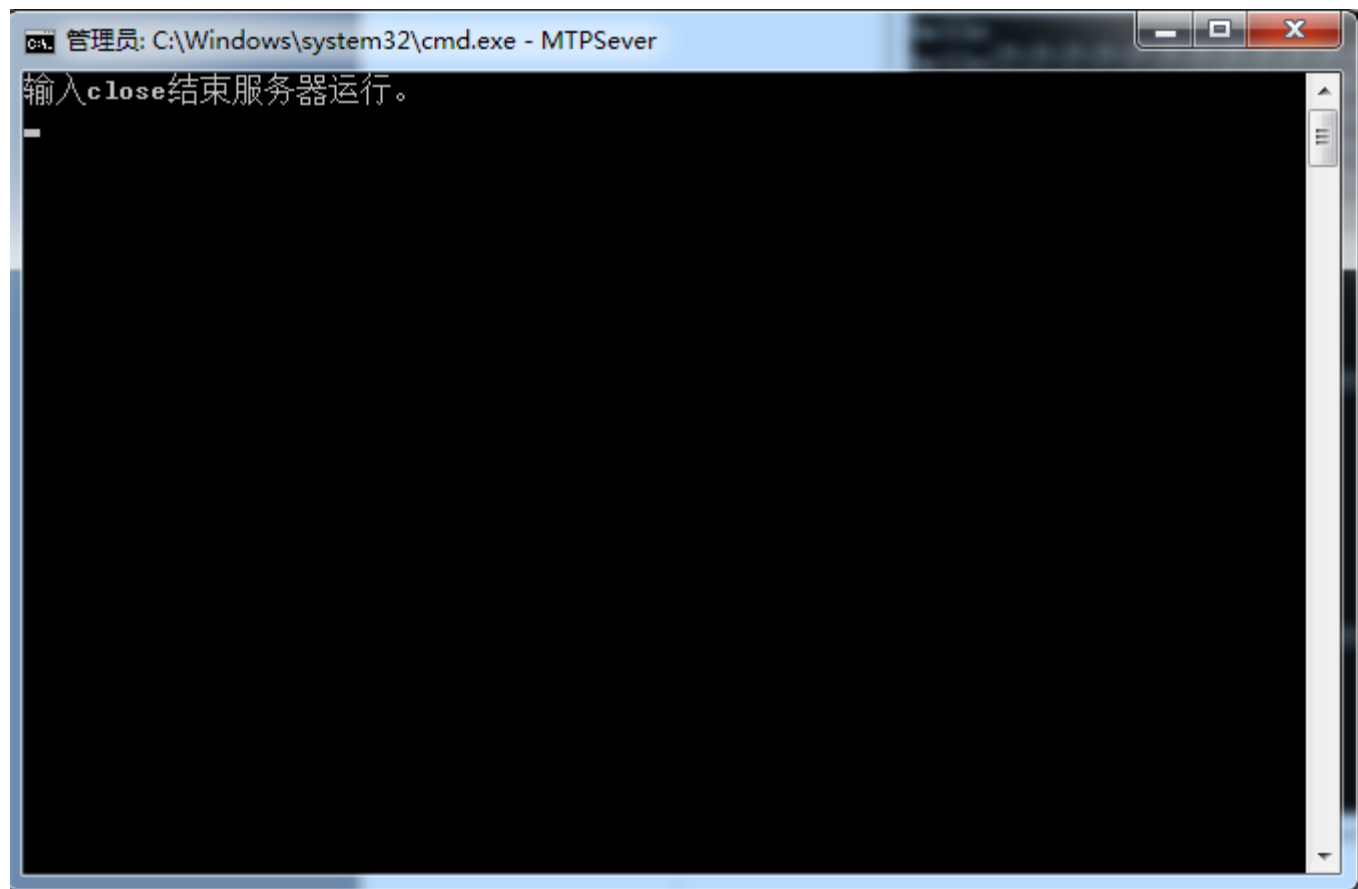
```

void ClientThread::shut()
{
    //关闭套接字
    closesocket(socket);
}

//重载比较运算符
bool ClientThread::operator==(ClientThread tmp)
{
    if (this->socket == tmp.socket)
        return true;
    else
        return false;
}

//结束客户端线程时的处理
void ClientThread::end_process_msg(std::vector<ClientThread>& threadsList)
{
    //关闭套接字
    closesocket(socket);
    //将线程处理类从队列移除
    std::vector<ClientThread>::iterator iter = find(threadsList.begin(), threadsList.end(), *this);
    if (iter != threadsList.end()) {
        threadsList.erase(iter);
        //printf("出队一个客户端处理线程\n");
    }
    //调用析构函数
    delete this;
}

```



客户端程序

```

#include <WinSock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <string.h>
#include <iostream>

#pragma comment(lib, "Ws2_32.lib")

//自定义的默认端口号,缓冲区长度
#define DEFAULT_PORT "27015"
#define DEFAULT_BUFLen 512

int main(int argc, char** argv)
{
    WSADATA wsaData;
    int iResult;
    struct addrinfo* result = NULL, * ptr = NULL, hints;
    SOCKET connectSocket = INVALID_SOCKET;
    char recvbuf[DEFAULT_BUFLen];
    int recvbuflen = DEFAULT_BUFLen;
    char msg[DEFAULT_BUFLen]; //消息

    //验证参数合法性
    if (argc != 2) {
        printf("命令用法: %s 服务器域名或ip\n", argv[0]);
        return 1;
    }

    //初始化套接字
    iResult = WSASStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSASStartup执行失败, 错误码: %d\n", iResult);
        return 1;
    }
}

```

```

//解析服务器地址和端口号
ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
iResult = getaddrinfo(argv[1], DEFAULT_PORT, &hints, &result);
if (iResult != 0) {
    printf("getaddrinfo执行错误, 错误码: %d\n", iResult);
    WSACleanup();
    return 1;
}
//尝试连接服务器地址, 直到成功
for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {
    //创建套接字
    connectSocket = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
    if (connectSocket == INVALID_SOCKET) {
        printf("socket执行失败, 错误码: %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //向服务器请求连接
    iResult = connect(connectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        closesocket(connectSocket);
        connectSocket = INVALID_SOCKET;
        continue;
    }
    break;
}
freeaddrinfo(result);
if (connectSocket == INVALID_SOCKET) {
    printf("无法连接到服务器!\n");
    WSACleanup();
    return 1;
}

```



```

//等待输入消息
std::cout << "输入bye结束通信。 \n";
do {
    std::cin >> msg;
    //发送信息
    iResult = send(connectSocket, msg, (int)strlen(msg), 0);
    if (iResult == SOCKET_ERROR) {
        printf("发送失败, 错误码:%d\n", WSAGetLastError());
        closesocket(connectSocket);
        WSACleanup();
        return 1;
    }
    //接收信息
    iResult = recv(connectSocket, recvbuf, recvbuflen, 0);
    if (iResult > 0) {
        std::string str(recvbuf, 0, iResult);
        std::cout << str << std::endl;
    }
    else if (iResult == 0) {
        printf("连接关闭\n");
        closesocket(connectSocket);
        WSACleanup();
        return 1;
    }
    else {
        printf("接收数据失败, 错误码: %d\n", WSAGetLastError());
        closesocket(connectSocket);
        WSACleanup();
        return 1;
    }
} while (strcmp(msg, "bye"));

```

```
//数据发送结束，调用shutdown()函数声明不再发送数据，此时客户端仍然可以接收数据
```

```
iResult = shutdown(connectSocket, SD_SEND);
```

```
if (iResult == SOCKET_ERROR) {
```

```
    printf("shutdown执行出错，错误码：%d\n", WSAGetLastError());
```

```
    closesocket(connectSocket);
```

```
    WSACleanup();
```

```
    return 1;
```

```
}
```

```
//持续接收数据，直到服务器关闭连接
```

```
do {
```

```
    iResult = recv(connectSocket, recvbuf, recvbuflen, 0);
```

```
    if (iResult > 0)
```

```
        printf("接收字节数：%d\n", iResult);
```

```
    else if (iResult == 0)
```

```
        printf("连接关闭\n");
```

```
    else
```

```
        printf("接收数据失败，错误码：%d\n", WSAGetLastError());
```

```
} while (iResult > 0);
```

```
//关闭套接字
```

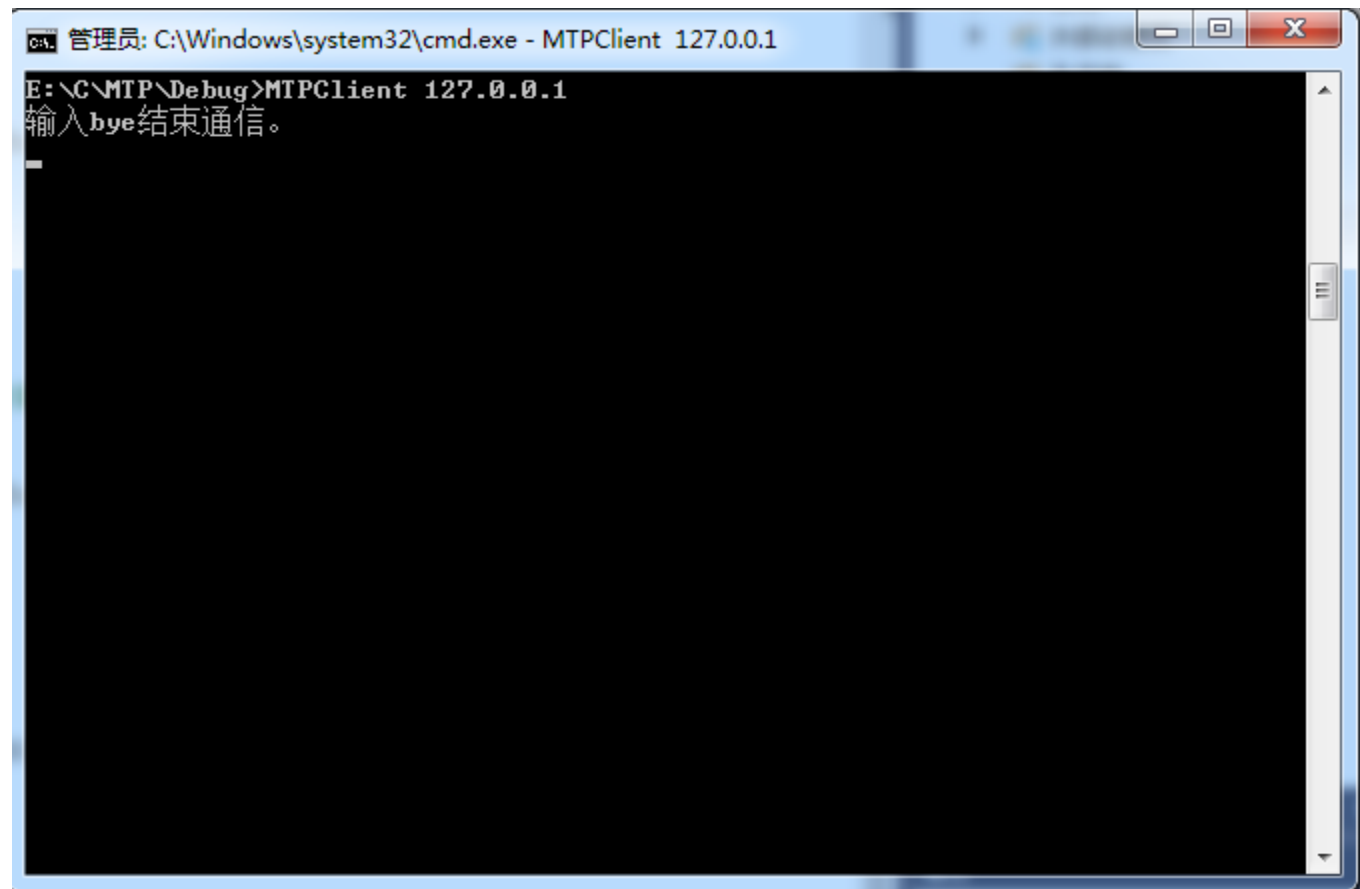
```
closesocket(connectSocket);
```

```
//释放资源
```

```
WSACleanup();
```

```
return 0;
```

```
}
```



A screenshot of a Windows command prompt window. The title bar reads "管理员: C:\Windows\system32\cmd.exe - MTPClient 127.0.0.1". The command prompt shows the user is in the directory "E:\C\MTP\Debug" and has executed the command "MTPClient 127.0.0.1". Below the command, the text "输入bye结束通信。" (Enter bye to end communication.) is displayed. The rest of the window is black.

```
管理员: C:\Windows\system32\cmd.exe - MTPClient 127.0.0.1
E:\C\MTP\Debug>MTPClient 127.0.0.1
输入bye结束通信。
```