[**Boost线程库学习笔记**](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html)

**一、创建一个线程**

**创建线程**

    boost::thread myThread(threadFun);

    需要注意的是：参数可以是函数对象或者函数指针。并且这个**函数无参数，并返回void类型**。

    当一个thread执行完成时，这个子线程就会消失。注意这个线程对象不会消失，它仍然是一个还处在它的生存期的C++对象。同理，当对一个堆上的线程对象的指针调用delete时候，线程对象被销毁，操作系统的线程并不能保证就消失。

**放弃时间片**

    boost::thread::yield();

    当前线程放弃余下的时间片。

**等待一个线程**

    myThread.join();

调用这个方法的线程进入wait状态，直到myThread代表的线程完成为止。如果它不结束的话，join方法就不会返回。join是一个等待子线程结束的最好的方法。如果主程序不调用join方法而直接结束，它的子线程有可能没有执行完成，但是所有的子线程也随之退出。不调用join方法，主线程就不会等待它的子线程。

[view source](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#viewSource)



[print](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#printSource)[?](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#about)

|  |  |
| --- | --- |
| 01 | #include <iostream> |
| 02 | #include <boost/thread/thread.hpp> | |

|  |  |  |
| --- | --- | --- |
| 03 | #include <boost/thread/xtime.hpp> | |
| 04 |  |

|  |  |
| --- | --- |
| 05 | struct MyThreadFunc { |
| 06 | void operator( )( ) { | |

|  |  |  |
| --- | --- | --- |
| 07 | // Do something long-running... | |
| 08 | } |

|  |  |  |
| --- | --- | --- |
| 09 | } threadFun; | |
| 10 |  |

|  |  |  |
| --- | --- | --- |
| 11 | int main( ) { | |
| 12 |  |

|  |  |  |
| --- | --- | --- |
| 13 | boost::thread myThread(threadFun); // Create a thread that starts | |
| 14 | // running threadFun |

|  |  |
| --- | --- |
| 15 |  |
| 16 | boost::thread::yield( );           // Give up the main thread's timeslice | |

|  |  |  |
| --- | --- | --- |
| 17 | // so the child thread can get some work | |
| 18 | // done. |

|  |  |
| --- | --- |
| 19 |  |
| 20 | // Go do some other work... | |

|  |  |
| --- | --- |
| 21 |  |
| 22 | myThread.join( );                  // The current (i.e., main) thread will wait | |

|  |  |  |
| --- | --- | --- |
| 23 | // for myThread to finish before it returns | |
| 24 |  |

|  |  |
| --- | --- |
| 25 | } |

**线程组**

    如果你需要创建几个线程，考虑使用一个线程组对象thread\_group来组织它们。一个thread\_group对象可以使用多种方法管理线程。首先，可以使用一个指向**动态创建的线程对象**的指针作为参数来调用add\_thread方法，将这个线程加入线程组。也可以直接使用线程组类的create\_thread方法，可不先创建线程而直接把线程加入到线程组中。

    当线程组对象的析构函数被调用时，它将删除（delete）所有这些通过add\_thread方法加入的线程指针。所以，只能将**堆上的线程对象指针**通过add\_thread方法加入线程组。remove\_thread方法从线程组删除某个线程的指针，但是我们仍需负责把线程本身内存释放掉。

    线程组对象的成员方法join\_all方法等待线程组中所有线程结束，才返回。

[view source](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#viewSource)



[print](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#printSource)[?](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#about)

|  |  |
| --- | --- |
| 01 | boost::thread\_group grp; |
| 02 | boost::thread \*p = new boost::thread(threadFun); | |

|  |  |  |
| --- | --- | --- |
| 03 | grp.add\_thread(p); | |
| 04 | //do something... |

|  |  |  |
| --- | --- | --- |
| 05 | grp.remove\_thread(p); | |
| 06 |  |

|  |  |
| --- | --- |
| 07 | grp.create\_thread(threadFun); |
| 08 | grp.create\_thread(threadFun);   //Now there are two threads in grp | |

|  |  |
| --- | --- |
| 09 |  |
| 10 | grp.join\_all();                 //Wait for all threads to finish | |

**二、使资源是线程安全的**

保证同一时刻多个线程不会同时修改同一个共享资源，那么这个程序是线程安全的，或者是串行化访问资源的。可以使用mutex类来控制线程的并发问题。

[view source](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#viewSource)



[print](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#printSource)[?](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#about)

|  |  |
| --- | --- |
| 01 | #include <iostream> |
| 02 | #include <boost/thread/thread.hpp> | |

|  |  |  |
| --- | --- | --- |
| 03 | #include <string> | |
| 04 |  |

|  |  |  |
| --- | --- | --- |
| 05 | // A simple queue class; don't do this, use std::queue | |
| 06 | template<typename T> |

|  |  |  |
| --- | --- | --- |
| 07 | class Queue { | |
| 08 | public: |

|  |  |
| --- | --- |
| 09 | Queue( ) {} |
| 10 | ~Queue( ) {} |

|  |  |
| --- | --- |
| 11 |  |
| 12 | void enqueue(const T& x) { | |

|  |  |
| --- | --- |
| 13 | // Lock the mutex for this queue |
| 14 | boost::mutex::scoped\_lock lock(mutex\_); | |

|  |  |
| --- | --- |
| 15 | list\_.push\_back(x); |
| 16 | // A scoped\_lock is automatically destroyed (and thus unlocked) | |

|  |  |  |
| --- | --- | --- |
| 17 | // when it goes out of scope | |
| 18 | } |

|  |  |
| --- | --- |
| 19 |  |
| 20 | T dequeue( ) { | |

|  |  |  |
| --- | --- | --- |
| 21 | boost::mutex::scoped\_lock lock(mutex\_); | |
| 22 |  |

|  |  |
| --- | --- |
| 23 | if (list\_.empty( )) |
| 24 | throw "empty!";     // This leaves the current scope, so the | |

|  |  |  |
| --- | --- | --- |
| 25 | T tmp = list\_.front( ); // lock is released | |
| 26 | list\_.pop\_front( ); |

|  |  |
| --- | --- |
| 27 | return(tmp); |
| 28 | } // Again: when scope ends, mutex\_ is unlocked | |

|  |  |
| --- | --- |
| 29 |  |
| 30 | private: | |

|  |  |
| --- | --- |
| 31 | std::list<T> list\_; |
| 32 | boost::mutex mutex\_; | |

|  |  |  |
| --- | --- | --- |
| 33 | }; | |
| 34 |  |

|  |  |  |
| --- | --- | --- |
| 35 | Queue<std::string> queueOfStrings; | |
| 36 |  |

|  |  |  |
| --- | --- | --- |
| 37 | void sendSomething( ) { | |
| 38 | std::string s; |

|  |  |
| --- | --- |
| 39 | for (int i = 0; i < 10; ++i) { |
| 40 | queueOfStrings.enqueue("Cyrus"); | |

|  |  |  |
| --- | --- | --- |
| 41 | } | |
| 42 | } |

|  |  |
| --- | --- |
| 43 |  |
| 44 | void recvSomething( ) { | |

|  |  |  |
| --- | --- | --- |
| 45 | std::string s; | |
| 46 |  |

|  |  |
| --- | --- |
| 47 | for (int i = 0; i < 10; ++i) { |
| 48 | try {s = queueOfStrings.dequeue( );} | |

|  |  |  |
| --- | --- | --- |
| 49 | catch(...) {} | |
| 50 | } |

|  |  |
| --- | --- |
| 51 | } |
| 52 |  | |

|  |  |
| --- | --- |
| 53 | int main( ) { |
| 54 | boost::thread thr1(sendSomething); | |

|  |  |  |
| --- | --- | --- |
| 55 | boost::thread thr2(recvSomething); | |
| 56 |  |

|  |  |
| --- | --- |
| 57 | thr1.join( ); |
| 58 | thr2.join( ); |

|  |  |
| --- | --- |
| 59 | } |

    mutex对象本身并不知道它代表什么，它仅仅是被多个消费者线程使用的资源访问的锁定解锁标志。在某个时刻，只有一个线程可以锁定这个mutex对象，这就阻止了同一时刻有多个线程并发访问共享资源。一个mutex就是一个简单的信号机制。

    给mutex加解锁有多种策略，最简单的是使用scoped\_lock类，它使用一个mutex参数来构造，并一直锁定这个mutex直到对象被销毁。如果这个正在被构造的mutex已经被别的线程锁定的话，当前线程就会进入wait状态，直到这个锁被解开。

**三、读写锁**

    mutex有一个美中不足，它不区分读和写。线程如果只是进行读操作，mutex强制线程串行化访问资源，效率低。而且这种操作不需要排他性访问。基于这个原因，Boost线程库提供了read\_write\_mutex。

[view source](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#viewSource)



[print](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#printSource)[?](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#about)

|  |  |
| --- | --- |
| 01 | #include <iostream> |
| 02 | #include <boost/thread/thread.hpp> | |

|  |  |  |
| --- | --- | --- |
| 03 | #include <boost/thread/read\_write\_mutex.hpp> | |
| 04 | #include <string> |

|  |  |
| --- | --- |
| 05 |  |
| 06 | template<typename T> | |

|  |  |  |
| --- | --- | --- |
| 07 | class Queue { | |
| 08 | public: |

|  |  |  |
| --- | --- | --- |
| 09 | | Queue( ) :  // Use a read/write mutex and give writers priority |
| 10 | rwMutex\_(boost::read\_write\_scheduling\_policy::writer\_priority){} | | |

|  |  |  |
| --- | --- | --- |
| 11 | ~Queue( ) {} | |
| 12 |  |

|  |  |
| --- | --- |
| 13 | void enqueue(const T& x) { |
| 14 | // Use a r/w lock since enqueue updates the state | |

|  |  |  |
| --- | --- | --- |
| 15 | boost::read\_write\_mutex::scoped\_write\_lock writeLock(rwMutex\_); | |
| 16 | list\_.push\_back(x); |

|  |  |  |
| --- | --- | --- |
| 17 | } | |
| 18 |  |

|  |  |
| --- | --- |
| 19 | T dequeue( ) { |
| 20 | // Again, use a write lock | |

|  |  |  |
| --- | --- | --- |
| 21 | boost::read\_write\_mutex::scoped\_write\_lock writeLock(rwMutex\_); | |
| 22 |  |

|  |  |  |
| --- | --- | --- |
| 23 | if (list\_.empty( )) | |
| 24 | throw "empty!"; |

|  |  |  |
| --- | --- | --- |
| 25 | T tmp = list\_.front( ); | |
| 26 | list\_.pop\_front( ); |

|  |  |  |
| --- | --- | --- |
| 27 | return(tmp); | |
| 28 | } |

|  |  |
| --- | --- |
| 29 |  |
| 30 | T getFront( ) { | |

|  |  |
| --- | --- |
| 31 | // This is a read-only operation, so you only need a read lock |
| 32 | boost::read\_write\_mutex::scoped\_read\_lock readLock(rwMutex\_); |

|  |  |  |
| --- | --- | --- |
| 33 | if (list\_.empty( )) | |
| 34 | throw "empty!"; |

|  |  |  |
| --- | --- | --- |
| 35 | return(list\_.front( )); | |
| 36 | } |

|  |  |
| --- | --- |
| 37 |  |
| 38 | private: | |

|  |  |
| --- | --- |
| 39 | std::list<T> list\_; |
| 40 | boost::read\_write\_mutex rwMutex\_; | |

|  |  |  |
| --- | --- | --- |
| 41 | }; | |
| 42 |  |

|  |  |  |
| --- | --- | --- |
| 43 | Queue<std::string> queueOfStrings; | |
| 44 |  |

|  |  |  |
| --- | --- | --- |
| 45 | void sendSomething( ) { | |
| 46 | std::string s; |

|  |  |
| --- | --- |
| 47 |  |
| 48 | for (int i = 0; i < 10; ++i) { | |

|  |  |  |
| --- | --- | --- |
| 49 | queueOfStrings.enqueue("Cyrus"); | |
| 50 | } |

|  |  |
| --- | --- |
| 51 | } |
| 52 |  | |

|  |  |  |
| --- | --- | --- |
| 53 | void checkTheFront( ) { | |
| 54 | std::string s; |

|  |  |
| --- | --- |
| 55 |  |
| 56 | for (int i = 0; i < 10; ++i) { | |

|  |  |  |
| --- | --- | --- |
| 57 | try {s = queueOfStrings.getFront( );} | |
| 58 | catch(...) {} |

|  |  |  |
| --- | --- | --- |
| 59 | } | |
| 60 | } |

|  |  |
| --- | --- |
| 61 |  |
| 62 | int main( ) { | |

|  |  |
| --- | --- |
| 63 |  |
| 64 | boost::thread thr1(sendSomething); | |

|  |  |  |
| --- | --- | --- |
| 65 | boost::thread\_group grp; | |
| 66 |  |

|  |  |
| --- | --- |
| 67 | grp.create\_thread(checkTheFront); |
| 68 | grp.create\_thread(checkTheFront); |

|  |  |
| --- | --- |
| 69 | grp.create\_thread(checkTheFront); |
| 70 | grp.create\_thread(checkTheFront); |

|  |  |
| --- | --- |
| 71 |  |
| 72 | thr1.join( ); | |

|  |  |  |
| --- | --- | --- |
| 73 | grp.join\_all( ); | |
| 74 | } |

     注意Queue的构造函数中队读写锁rwMutex的初始化。同一时刻，可能有多个读写线程要锁定一个read\_write\_mutex，而这些锁的调度策略依赖于构造这个mutex时选定的调度策略。Boost库中提供了四种调度策略：

1）reader\_priority：等待读锁的线程优先于等待写锁的线程

2）writer\_priority：等待写锁的线程优先于等待读锁的线程

3）alternating\_single\_read：在读锁和写锁之间交替

4）alternating\_many\_reads：在读锁和写锁之间交替，这个策略将在两个写锁之间使得所有的在这个queue上挂起的读锁都被允许。

     选择使用哪种策略要慎重，因为使用前两种的话可能会导致某些锁始终不能成功，出现**饿死**的现象。

**死锁、饿死和竞态条件**

    1）死锁，是涉及至少2个线程和2个资源的情况。线程A和B，资源X和Y。A锁定了X，而B锁定了Y。此时A和B有彼此想要对方的资源，死锁就出现了。

死锁的预防有两种方法。一种是，通过小心的按照一定的顺序对不同的mutex来加锁。另一种是，使用Boost提供的try\_mutex互斥量和scoped\_try\_lock。或者使用**时间锁**。scoped\_try\_lock对try\_mutex加锁时，可能成功，也可能失败，但不会阻塞。时间锁则有一个超时时间。

[view source](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#viewSource)



[print](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#printSource)[?](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#about)

|  |  |  |
| --- | --- | --- |
| 01 | bool dequeue(T& x) | |
| 02 | { |

|  |  |  |
| --- | --- | --- |
| 03 | boost::try\_mutex::scope\_try\_lock lock(tryMutex\_); | |
| 04 | if(!lock.locked()) |

|  |  |  |
| --- | --- | --- |
| 05 | return false; | |
| 06 | else{ |

|  |  |
| --- | --- |
| 07 | if (list\_.empty()) |
| 08 | throw "empty!"; | |

|  |  |
| --- | --- |
| 09 | x = list\_.front(); |
| 10 | list\_.pop\_front(); |

|  |  |  |
| --- | --- | --- |
| 11 | return true; | |
| 12 | } |

|  |  |
| --- | --- |
| 13 | } |
| 14 | private: | |

|  |  |
| --- | --- |
| 15 | boost::try\_mutex tryMutex\_; |

    2）饿死，如果你正在使用write\_priority策略，并且你有很多创建写锁的线程，那么读锁的线程就可能饿死。

    3）竞态条件，

[view source](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#viewSource)



[print](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#printSource)[?](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#about)

|  |  |  |
| --- | --- | --- |
| 1 | if(q.getFront() == "Cyrus"){ | |
| 2 | str = q.dequeue(); |

|  |  |  |
| --- | --- | --- |
| 3 | //.... | |
| 4 | } |

       这个代码在单线程环境中工作很好，因为q在第一行和第二行代码之间不会被修改。多线程环境中则会出现问题。此为竞态条件。解决的方法是为Queue添加一个成员函数dequeueIfEquals，在函数执行过程中始终锁定互斥量。

**四、从一个线程中给另一个线程发送通知**

    当需要线程等待某个事物时，可以创建一个condition对象，然后通过这个对象来通知那些在等待的线程。

[view source](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#viewSource)



[print](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#printSource)[?](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#about)

|  |  |
| --- | --- |
| 01 | #include <iostream> |
| 02 | #include <boost/thread/thread.hpp> | |

|  |  |  |
| --- | --- | --- |
| 03 | #include <boost/thread/condition.hpp> | |
| 04 | #include <boost/thread/mutex.hpp> |

|  |  |
| --- | --- |
| 05 | #include <list> |
| 06 | #include <string> | |

|  |  |
| --- | --- |
| 07 |  |
| 08 | class Request { /\*...\*/ }; | |

|  |  |
| --- | --- |
| 09 |  |
| 10 | // A simple job queue class; don't do this, use std::queue | |

|  |  |  |
| --- | --- | --- |
| 11 | template<typename T> | |
| 12 | class JobQueue { |

|  |  |
| --- | --- |
| 13 | public: |
| 14 | JobQueue( ) {} | |

|  |  |  |
| --- | --- | --- |
| 15 | ~JobQueue( ) {} | |
| 16 |  |

|  |  |
| --- | --- |
| 17 | void submitJob(const T& x)  { |
| 18 | boost::mutex::scoped\_lock lock(mutex\_); | |

|  |  |
| --- | --- |
| 19 | list\_.push\_back(x); |
| 20 | workToBeDone\_.notify\_one( ); | |

|  |  |  |
| --- | --- | --- |
| 21 | } | |
| 22 |  |

|  |  |
| --- | --- |
| 23 | T getJob( )  { |
| 24 | boost::mutex::scoped\_lock lock(mutex\_); | |

|  |  |
| --- | --- |
| 26 | workToBeDone\_.wait(lock); |

|  |  |
| --- | --- |
| 28 | T tmp = list\_.front( ); |

|  |  |  |
| --- | --- | --- |
| 29 | list\_.pop\_front( ); | |
| 30 | return(tmp); |

|  |  |  |
| --- | --- | --- |
| 31 | } | |
| 32 |  |

|  |  |
| --- | --- |
| 33 | private: |
| 34 | std::list<T> list\_; | |

|  |  |
| --- | --- |
| 35 | boost::mutex mutex\_; |
| 36 | boost::condition workToBeDone\_; | |

|  |  |  |
| --- | --- | --- |
| 37 | }; | |
| 38 |  |

|  |  |  |
| --- | --- | --- |
| 39 | JobQueue<Request> myJobQueue; | |
| 40 |  |

|  |  |  |
| --- | --- | --- |
| 41 | void boss( ) { | |
| 42 | for (;;) { |

|  |  |  |
| --- | --- | --- |
| 43 | // Get the request from somewhere | |
| 44 | Request req; |

|  |  |  |
| --- | --- | --- |
| 45 | myJobQueue.submitJob(req); | |
| 46 | } |

|  |  |
| --- | --- |
| 47 | } |
| 48 |  | |

|  |  |  |
| --- | --- | --- |
| 49 | void worker( ) { | |
| 50 | for (;;) { |

|  |  |  |
| --- | --- | --- |
| 51 | Request r(myJobQueue.getJob( )); | |
| 52 | // Do something with the job... |

|  |  |  |
| --- | --- | --- |
| 53 | } | |
| 54 | } |

|  |  |
| --- | --- |
| 55 |  |
| 56 | int main( ) { | |

|  |  |
| --- | --- |
| 57 | boost::thread thr1(boss); |
| 58 | boost::thread thr2(worker); | |

|  |  |  |
| --- | --- | --- |
| 59 | boost::thread thr3(worker); | |
| 60 |  |

|  |  |
| --- | --- |
| 61 | thr1.join( ); |
| 62 | thr2.join( ); |

|  |  |  |
| --- | --- | --- |
| 63 | thr3.join( ); | |
| 64 | } |

boost::mutex::scoped\_lock lock(mutex\_);

workToBeDone\_.wait(lock);

     这两行代码，第一行锁定这个mutex对象。第二行代码解开这个mutex上的锁，然后进行等待或者休眠，直到它的条件得到了满足。这个mutex互斥对象的解锁让其他的线程能够使用这个mutex对象，它们中的某个需要设置这个等待条件，之后通知另外的线程。

    notify\_all函数，通知那些所有正在等待某个条件变为真的线程，那些线程随后进入运行状态。wait方法做两件事情：它一直等待直到有人在它正等待的condition上调用notify\_one或notify\_all，然后它就试图锁定相关的mutex。当调用的是notify\_all时，尽管多个等待的线程都尽量去获得下一个锁，但谁将获得依赖于这个mutex的类型和使用的优先策略。

    一个condition对象能让消费者线程休眠，因此在还没有碰到一个condition时处理器可以去处理别的事情。例如一个web服务器使用一个工作线程池来处理进来的请求。当没有需求进来时，让这些子线程处于等待状态比让它们循环的查询或者睡眠然后偶尔唤醒来检查这个队列，要好很多。

**五、只初始化一次共享资源**

[view source](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#viewSource)



[print](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#printSource)[?](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#about)

|  |  |
| --- | --- |
| 01 | #include <iostream> |
| 02 | #include <boost/thread/thread.hpp> | |

|  |  |  |
| --- | --- | --- |
| 03 | #include <boost/thread/once.hpp> | |
| 04 |  |

|  |  |  |
| --- | --- | --- |
| 05 | // Some sort of connection class that should only be initialized once | |
| 06 | struct Conn { |

|  |  |
| --- | --- |
| 07 | static void init( ) {++i\_;} |
| 08 | static boost::once\_flag init\_; | |

|  |  |  |
| --- | --- | --- |
| 09 | static int i\_; | |
| 10 | // ... |

|  |  |  |
| --- | --- | --- |
| 11 | }; | |
| 12 |  |

|  |  |
| --- | --- |
| 13 | int Conn::i\_ = 0; |
| 14 | boost::once\_flag Conn::init\_ = BOOST\_ONCE\_INIT; | |

|  |  |
| --- | --- |
| 15 |  |
| 16 | void worker( ) { | |

|  |  |  |
| --- | --- | --- |
| 17 | boost::call\_once(Conn::init, Conn::init\_); | |
| 18 | // Do the real work... |

|  |  |
| --- | --- |
| 19 | } |
| 20 |  | |

|  |  |  |
| --- | --- | --- |
| 21 | Conn c;  // You probably don't want to use a global, so see the | |
| 22 | // next Recipe |

|  |  |
| --- | --- |
| 23 |  |
| 24 | int main( ) { | |

|  |  |
| --- | --- |
| 25 |  |
| 26 | boost::thread\_group grp; | |

|  |  |
| --- | --- |
| 27 |  |
| 28 | for (int i = 0; i < 100; ++i) | |

|  |  |  |
| --- | --- | --- |
| 29 | grp.create\_thread(worker); | |
| 30 |  |

|  |  |  |
| --- | --- | --- |
| 31 | grp.join\_all( ); | |
| 32 |  |

|  |  |  |
| --- | --- | --- |
| 33 | std::cout << c.i\_ << '\n'; // c.i\_ = 1 | |
| 34 | } |

    一个共享资源不得不在某个地方被初始化，并且你希望第一次使用这个资源的线程来完成初始化工作。一个once\_flag类型和call\_once函数能够保证多个线程不会重复的初始化同一个对象。首先，必须使用BOOST\_ONCE\_INIT宏来初始化这个once\_flag对象。boost::once\_flag Conn::init\_ = BOOST\_ONCE\_INIT; 之后调用call\_once函数，boost::call\_once(Conn::init, Conn::init\_); 第一个形参是希望被执行一次的初始化函数的地址。

**六、给线程函数传递一个参数**

[view source](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#viewSource)



[print](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#printSource)[?](http://www.cnblogs.com/younes/archive/2010/06/06/1752745.html#about)

|  |  |  |
| --- | --- | --- |
| 01 | #include <iostream> | |
| 02 | #include <string> |

|  |  |
| --- | --- |
| 03 | #include <functional> |
| 04 | #include <boost/thread/thread.hpp> | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | // A typedef to make the declarations below easier to read | |

|  |  |  |
| --- | --- | --- |
| 07 | typedef void (\*WorkerFunPtr)(const std::string&); | |
| 08 |  |

|  |  |  |
| --- | --- | --- |
| 09 | template<typename FunT,   // The type of the function being called | |
| 10 | typename ParamT> // The type of its parameter |

|  |  |
| --- | --- |
| 11 | struct Adapter { |
| 12 | Adapter(FunT f, ParamT& p) : // Construct this adapter and set the | |

|  |  |  |
| --- | --- | --- |
| 13 | f\_(f), p\_(&p) {}          // members to the function and its arg | |
| 14 |  |

|  |  |  |
| --- | --- | --- |
| 15 | void operator( )( ) { // This just calls the function with its arg | |
| 16 | f\_(\*p\_); |

|  |  |
| --- | --- |
| 17 | } |
| 18 | private: | |

|  |  |
| --- | --- |
| 19 | FunT    f\_; |
| 20 | ParamT\* p\_;  // Use the parameter's address to avoid extra copying | |

|  |  |  |
| --- | --- | --- |
| 21 | }; | |
| 22 |  |

|  |  |  |
| --- | --- | --- |
| 23 | void worker(const std::string& s) { | |
| 24 | std::cout << s << '\n'; |

|  |  |
| --- | --- |
| 25 | } |
| 26 |  | |

|  |  |  |
| --- | --- | --- |
| 27 | int main( ) { | |
| 28 |  |

|  |  |
| --- | --- |
| 29 | std::string s1 = "This is the first thread!"; |
| 30 | std::string s2 = "This is the second thread!"; | |

|  |  |
| --- | --- |
| 31 |  |
| 32 | boost::thread thr1(Adapter<WorkerFunPtr, std::string>(worker, s1)); | |

|  |  |  |
| --- | --- | --- |
| 33 | boost::thread thr2(Adapter<WorkerFunPtr, std::string>(worker, s2)); | |
| 34 |  |

|  |  |
| --- | --- |
| 35 | thr1.join( ); |
| 36 | thr2.join( ); |

|  |  |
| --- | --- |
| 37 | } |

    使用这个函数适配器类模板，你就可以给线程函数传递参数了。如果你需要传递多个参数，仅需要在这个适配器中增加另一个类型和成员变量。