**[C++11 thread](http://blog.csdn.net/tujiaw/article/details/8245130)**

windows系统中，需要vs2010+sp1或vs2012才支持。  
  
  
1.线程的创建  
C++11线程类std::thread，头文件include <thread>  
首先，看一个最简单的例子：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/tujiaw/article/details/8245130)[在CODE上查看代码片](https://code.csdn.net/snippets/85436)

1. void my\_thread()
2. {
3. puts("hello, world");
4. }
6. **int** main(**int** argc, **char** \*argv[])
7. {
8. std::thread t(my\_thread);
9. t.join();
11. system("pause");
12. return 0;
13. }

void my\_thread()

{

puts("hello, world");

}

int main(int argc, char \*argv[])

{

std::thread t(my\_thread);

t.join();

system("pause");

return 0;

}

实例化一个线程对象t，参数my\_thread是一个函数，在线程创建完成后将被执行，  
t.join()等待子线程my\_thread执行完之后，主线程才可以继续执行下去，此时主线程会  
释放掉执行完后的子线程资源。  
  
  
当然，如果不想等待子线程，可以在主线程里面执行t.detach()将子线程从主线程里分离，  
子线程执行完成后会自己释放掉资源。分离后的线程，主线程将对它没有控制权了。  
相对于以前使用过的beginthread传多个参数需要传入struct地址，  
boost::thread传参需要bind，std::thread传参真的非常方便，而且可读性也很好。  
下面例子在实例化线程对象的时候，在线程函数my\_thread后面紧接着传入两个参数。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/tujiaw/article/details/8245130)[在CODE上查看代码片](https://code.csdn.net/snippets/85436)

1. #include <iostream>
2. #include <stdlib.h>
3. #include <thread>
4. #include <string>
6. void my\_thread(**int** num, const std::string& str)
7. {
8. std::cout << "num:" << num << ",name:" << str << std::endl;
9. }
11. **int** main(**int** argc, **char** \*argv[])
12. {
13. **int** num = 1234;
14. std::string str = "tujiaw";
15. std::thread t(my\_thread, num, str);
16. t.detach();
18. system("pause");
19. return 0;
20. }

#include <iostream>

#include <stdlib.h>

#include <thread>

#include <string>

void my\_thread(int num, const std::string& str)

{

std::cout << "num:" << num << ",name:" << str << std::endl;

}

int main(int argc, char \*argv[])

{

int num = 1234;

std::string str = "tujiaw";

std::thread t(my\_thread, num, str);

t.detach();

system("pause");

return 0;

}

2.互斥量  
多个线程同时访问共享资源的时候需要需要用到互斥量，当一个线程锁住了互斥量后，其他线程必须等待这个互斥量解锁后才能访问它。thread提供了四种不同的互斥量：  
独占式互斥量non-recursive (std::mutex)  
递归式互斥量recursive (std::recursive\_mutex)  
允许超时的独占式互斥量non-recursive that allows timeouts on the lock functions(std::timed\_mutex)  
允许超时的递归式互斥量recursive mutex that allows timeouts on the lock functions (std::recursive\_timed\_mutex)  
  
  
独占式互斥量  
独占式互斥量加解锁是成对的，同一个线程内独占式互斥量在没有解锁的情况下，再次对它进行加锁这是不对的，会得到一个未定义行为。  
如果你想thread1输出10次10，thread2输出10次20，如果你想看到一个正确的显示效果，下面程序是做不到的，因为在thread1输出的时候，  
thread2也会执行，输出的结果看起来有点乱（std::cout不是线程安全的），所以我们需要在它们访问共享资源的时候使用互斥量加锁。打开代码里面的三行注释就可以得到正确的结果了。在线程1中std::mutex使用成员函数lock加锁unlock解锁，看起来工作的很好，但这样是不安全的，你得始终记住lock之后一定要unlock，但是如果在它们中间出现了异常或者线程直接退出了unlock就没有执行，因为这个互斥量是独占式的，所以在thread1没有解锁之前，其他使用这个互斥量加锁的线程会一直处于等待状态得不到执行。lock\_guard模板类使用RAII手法封装互斥量，在实例化对象的时候帮你加锁，并且能保证在离开作用域的时候自动解锁，所以你应该用lock\_guard来帮你加解锁。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/tujiaw/article/details/8245130)[在CODE上查看代码片](https://code.csdn.net/snippets/85436)

1. #include <iostream>
2. #include <stdlib.h>
3. #include <thread>
4. #include <string>
5. #include <mutex>
7. **int** g\_num = 0;
8. std::mutex g\_mutex;
10. void thread1()
11. {
12. //g\_mutex.lock();
13. g\_num = 10;
14. for (**int** i=0; i<10; i++){
15. std::cout << "thread1:" << g\_num << std::endl;
16. }
17. //g\_mutex.unlock();
18. }
20. void thread2()
21. {
22. //std::lock\_guard<std::mutex> lg(g\_mutex);
23. g\_num = 20;
24. for (**int** i=0; i<10; i++){
25. std::cout << "thread2:" << g\_num << std::endl;
26. }
27. }
29. **int** main(**int** argc, **char** \*argv[])
30. {
31. std::thread t1(thread1);
32. std::thread t2(thread2);
33. t1.join();
34. t2.join();
36. system("pause");
37. return 0;
38. }

#include <iostream>

#include <stdlib.h>

#include <thread>

#include <string>

#include <mutex>

int g\_num = 0;

std::mutex g\_mutex;

void thread1()

{

//g\_mutex.lock();

g\_num = 10;

for (int i=0; i<10; i++){

std::cout << "thread1:" << g\_num << std::endl;

}

//g\_mutex.unlock();

}

void thread2()

{

//std::lock\_guard<std::mutex> lg(g\_mutex);

g\_num = 20;

for (int i=0; i<10; i++){

std::cout << "thread2:" << g\_num << std::endl;

}

}

int main(int argc, char \*argv[])

{

std::thread t1(thread1);

std::thread t2(thread2);

t1.join();

t2.join();

system("pause");

return 0;

}

递归式互斥量  
与独占式互斥量不同的是，同一个线程内在互斥量没有解锁的情况下可以再次进行加锁，不过他们的加解锁次数需要一致，递归式互斥量我们平时可能用得比较少些。  
  
  
允许超时的互斥量  
如果线程1对共享资源的访问时间比较长，这时线程2可能等不了那么久，所以设置一个超时时间，在超时时间内如果线程1中的互斥量还没有解锁，线程2就不等了，继续往下执行。  
lock\_guard只是提供了对互斥量最基本的加解锁封装，而unique\_lock提供了多种构造方法，使用起来更加灵活，对于允许超时的互斥量需要使用unnique\_lock来包装。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/tujiaw/article/details/8245130)[在CODE上查看代码片](https://code.csdn.net/snippets/85436)

1. std::timed\_mutex g\_timed\_mutex;
2. void thread1()
3. {
4. std::unique\_lock<std::timed\_mutex> tl(g\_timed\_mutex);
5. ::Sleep(3000); // 睡眠3秒
6. puts("thread1");
7. }
9. void thread2()
10. {
11. std::unique\_lock<std::timed\_mutex> tl(g\_timed\_mutex, std::chrono::milliseconds(1000)); // 超时时间1秒
12. puts("thread2");
13. }
15. **int** main(**int** argc, **char** \*argv[])
16. {
17. std::thread t1(thread1);
18. ::Sleep(100); // 让线程1先启动
19. std::thread t2(thread2);
20. t1.join();
21. t2.join();
23. system("pause");
24. return 0;
25. }

std::timed\_mutex g\_timed\_mutex;

void thread1()

{

std::unique\_lock<std::timed\_mutex> tl(g\_timed\_mutex);

::Sleep(3000); // 睡眠3秒

puts("thread1");

}

void thread2()

{

std::unique\_lock<std::timed\_mutex> tl(g\_timed\_mutex, std::chrono::milliseconds(1000)); // 超时时间1秒

puts("thread2");

}

int main(int argc, char \*argv[])

{

std::thread t1(thread1);

::Sleep(100); // 让线程1先启动

std::thread t2(thread2);

t1.join();

t2.join();

system("pause");

return 0;

}

注意死锁  
有时，一个操作需要对一个以上的mutex加锁，这时请注意了，这样很可能造成死锁。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/tujiaw/article/details/8245130)[在CODE上查看代码片](https://code.csdn.net/snippets/85436)

1. struct Widget
2. {
3. std::mutex mutex\_;
4. std::string str\_;
5. };
7. void foo(Widget& w1, Widget& w2)
8. {
9. std::unique\_lock<std::mutex> t1(w1.mutex\_);
10. std::unique\_lock<std::mutex> t2(w2.mutex\_);
11. // do something
12. }
13. Widget g\_w1, g\_w2;

struct Widget

{

std::mutex mutex\_;

std::string str\_;

};

void foo(Widget& w1, Widget& w2)

{

std::unique\_lock<std::mutex> t1(w1.mutex\_);

std::unique\_lock<std::mutex> t2(w2.mutex\_);

// do something

}

Widget g\_w1, g\_w2;

当一个线程调用foo(g\_w1, g\_w2),另外一个线程调用foo(g\_w2, g\_w1)的时候，  
线程1：                线程2：  
w1.mutex\_.lock         ...  
...                    w2.mutex\_.lock  
...                    ...  
w2.mutex\_.lock等待     ...  
                       w1.mutex\_lock等待  
可能的执行顺序：  
线程1中的w1上锁；  
线程2中的w2上锁；  
线程1中的w2上锁，此时由于w2已经在线程2中上过锁了，所以必须等待；  
线程2中的w1上锁，此时由于w1已经在线程1中上过锁了，所以必须等待；  
这样两个线程都等不到对方释放锁，都处于等待状态造成了死锁。  
thread提供了一个std::lock函数可以对多个互斥量同时加锁，每个线程里面的  
w1和w2会同时上锁，他们之间就没有间隙了，如上将foo函数改为如下形式就可以了：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/tujiaw/article/details/8245130)[在CODE上查看代码片](https://code.csdn.net/snippets/85436)

1. void foo(Widget& w1, Widget& w2)
2. {
3. std::unique\_lock<std::mutex> t1(w1.mutex\_, std::defer\_lock);
4. std::unique\_lock<std::mutex> t2(w2.mutex\_, std::defer\_lock);
5. std::lock(t1, t2);
6. // do something
7. }

void foo(Widget& w1, Widget& w2)

{

std::unique\_lock<std::mutex> t1(w1.mutex\_, std::defer\_lock);

std::unique\_lock<std::mutex> t2(w2.mutex\_, std::defer\_lock);

std::lock(t1, t2);

// do something

}

在实例化的时候先不要加锁，等到两个对象都创建完成之后再一起加锁。  
  
  
在初始化的时候保护数据

如果你的数据仅仅只在初始化的时候进行保护，使用一个互斥量是不行的，在初始化完成后会导致没必要的同步，C++11提供了一些方法来解决这个问题。

3.线程间同步，条件变量

如果我们在线程间共享数据，经常会存在一个线程等待另外一个线程的情况，它们之间存在先后关系。

这个与互斥量不同，互斥量是保证多个线程的时候当前只有一个线程访问加锁的代码块，它们之间是不存在先后关系的。

如下例子：线程1需要等到线程2将flag设置为非0才进行打印

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/tujiaw/article/details/8245130)[在CODE上查看代码片](https://code.csdn.net/snippets/85436)

1. #include <iostream>
2. #include <thread>
3. #include <mutex>
4. #include <condition\_variable>
5. #include <functional>
7. class Foo
8. {
9. public:
10. Foo()
11. : flag\_(0)
12. , thread1\_(std::bind(&Foo::threadFunc1, this))
13. , thread2\_(std::bind(&Foo::threadFunc2, this))
14. {
15. }
17. ~Foo()
18. {
19. thread1\_.join();
20. thread2\_.join();
21. }
23. private:
24. void threadFunc1()
25. {
26. {
27. std::unique\_lock<std::mutex> ul(mutex\_);
28. while (0 == flag\_) {
29. cond\_.wait(ul);
30. }
31. std::cout << flag\_ << std::endl;
32. }
33. }
35. void threadFunc2()
36. {
37. // 为了测试，等待3秒
38. std::this\_thread::sleep\_for((std::chrono::milliseconds(3000)));
39. std::unique\_lock<std::mutex> ul(mutex\_);
40. flag\_ = 100;
41. cond\_.notify\_one();
42. }
44. **int** flag\_;
45. std::mutex mutex\_;
46. std::condition\_variable cond\_;
47. std::thread thread1\_;
48. std::thread thread2\_;
49. };
51. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
52. {
53. Foo f;
55. system("pause");
56. return 0;
57. }

#include <iostream>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <functional>

class Foo

{

public:

Foo()

: flag\_(0)

, thread1\_(std::bind(&Foo::threadFunc1, this))

, thread2\_(std::bind(&Foo::threadFunc2, this))

{

}

~Foo()

{

thread1\_.join();

thread2\_.join();

}

private:

void threadFunc1()

{

{

std::unique\_lock<std::mutex> ul(mutex\_);

while (0 == flag\_) {

cond\_.wait(ul);

}

std::cout << flag\_ << std::endl;

}

}

void threadFunc2()

{

// 为了测试，等待3秒

std::this\_thread::sleep\_for((std::chrono::milliseconds(3000)));

std::unique\_lock<std::mutex> ul(mutex\_);

flag\_ = 100;

cond\_.notify\_one();

}

int flag\_;

std::mutex mutex\_;

std::condition\_variable cond\_;

std::thread thread1\_;

std::thread thread2\_;

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

Foo f;

system("pause");

return 0;

}

从代码可以看出，虽然线程1明显比线程2快些（人为制造等待3秒），但是线程1还是会等待线程2将flag设置为非0后才进行打印的。

这里有几个地方需要注意：

1> Foo类成员变量定义的顺序，mutex和cond必须在thread的前面。

原因是：如果线程的定义在前面，线程初始化完成之后立马会执行线程函数，而线程函数里用到了mutex和cond，此时如果mutex和cond还没初始化完成，就会出现内存错误。

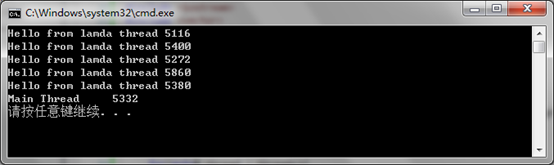
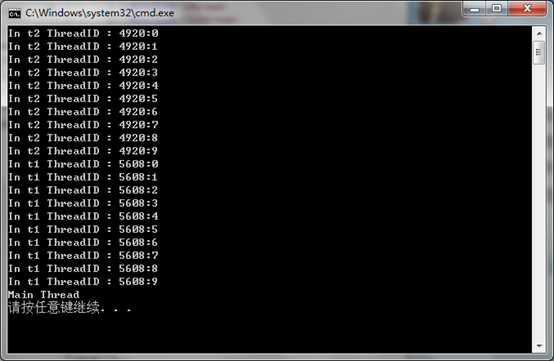
2>由于同时有两个线程需要操作flag变量，所以在读写的时候要加锁， std::unique\_lock<std::mutex>会保证构造的时候加锁，离开作用域调用析构的时候解锁，所以不用担心加解锁不匹配。

3>threadFunc1中的while (0 == flag\_)， 必须这样写不能写成if (0 == flag\_)，因为在多核CPU下会存在虚假唤醒（ spurious wakes）的情况。

4>cond\_.wait(ul);条件变量在wait的时候会释放锁的，所以其他线程可以继续执行。

4.线程池

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/tujiaw/article/details/8245130)[在CODE上查看代码片](https://code.csdn.net/snippets/85436)

1. #include <iostream>
2. #include <stdlib.h>
3. #include <functional>
4. #include <thread>
5. #include <string>
6. #include <mutex>
7. #include <condition\_variable>
8. #include <vector>
9. #include <memory>
10. #include <assert.h>
11. #include <algorithm>
12. #include <queue>
13. #include <process.h>
14. #include <Windows.h>
16. class ThreadPool
17. {
18. public:
19. typedef std::function<void()> Task;
21. ThreadPool(**int** num)
22. : num\_(num)
23. , maxQueueSize\_(0)
24. , running\_(false)
25. {
26. }
28. ~ThreadPool()
29. {
30. if (running\_) {
31. stop();
32. }
33. }
35. ThreadPool(const ThreadPool&) = delete;
36. void operator=(const ThreadPool&) = delete;
38. void setMaxQueueSize(**int** maxSize)
39. {
40. maxQueueSize\_ = maxSize;
41. }
43. void start()
44. {
45. assert(threads\_.empty());
46. running\_ = true;
47. threads\_.reserve(num\_);
48. for (**int** i = 0; i<num\_; i++) {
49. threads\_.push\_back(std::thread(std::bind(&ThreadPool::threadFunc, this)));
50. }
51. }
53. void stop()
54. {
55. {
56. std::unique\_lock<std::mutex> ul(mutex\_);
57. running\_ = false;
58. notEmpty\_.notify\_all();
59. }
61. for (auto &iter : threads\_) {
62. iter.join();
63. }
64. }
66. void run(const Task &t)
67. {
68. if (threads\_.empty()) {
69. t();
70. }
71. else {
72. std::unique\_lock<std::mutex> ul(mutex\_);
73. while (isFull()) {
74. notFull\_.wait(ul);
75. }
76. assert(!isFull());
77. queue\_.push\_back(t);
78. notEmpty\_.notify\_one();
79. }
80. }
82. private:
83. **bool** isFull() const
84. {
85. return maxQueueSize\_ > 0 && queue\_.size() >= maxQueueSize\_;
86. }
88. void threadFunc()
89. {
90. printf("create id:%d\n", ::GetCurrentThreadId());
91. while (running\_) {
92. Task task(take());
93. if (task) {
94. task();
95. }
96. }
97. printf("thread id:%d\n", ::GetCurrentThreadId());
98. }
100. Task take()
101. {
102. std::unique\_lock<std::mutex> ul(mutex\_);
103. while (queue\_.empty() && running\_) {
104. notEmpty\_.wait(ul);
105. }
106. Task task;
107. if (!queue\_.empty()) {
108. task = queue\_.front();
109. queue\_.pop\_front();
110. if (maxQueueSize\_ > 0) {
111. notFull\_.notify\_one();
112. }
113. }
114. return task;
115. }
117. private:
118. **int** num\_;
119. std::mutex mutex\_;
120. std::condition\_variable notEmpty\_;
121. std::condition\_variable notFull\_;
122. std::vector<std::thread> threads\_;
123. std::deque<Task> queue\_;
124. **size\_t** maxQueueSize\_;
125. **bool** running\_;
126. };
128. void fun()
129. {
130. printf("[id:%d] hello, world!\n", ::GetCurrentThreadId());
131. }
133. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
134. {
135. {
136. printf("main thread id:%d\n", ::GetCurrentThreadId());
137. ThreadPool pool(3);
138. pool.setMaxQueueSize(100);
139. pool.start();
140. //std::this\_thread::sleep\_for(std::chrono::milliseconds(3000));
142. for (**int** i = 0; i < 1000; i++) {
143. pool.run(fun);
144. }
145. std::this\_thread::sleep\_for(std::chrono::milliseconds(3000));
146. }
148. system("pause");
149. return 0;
150. }
151. C++11开始支持多线程编程，之前多线程编程都需要系统的支持，在不同的系统下创建线程需要不同的API如pthread\_create()，Createthread()，beginthread()等，使用起来都比较复杂，C++11提供了新头文件<thread>、<mutex>、<atomic>、<future>等用于支持多线程。
152. 使用C++11开启一个线程是比较简单的，下面来看一个简单的例子：
153. #include <thread>
154. #include <iostream>
156. void hello()
157. {
158. std::cout << "Hello from thread " << std::endl;
159. }
161. int main()
162. {
163. std::thread t1(hello);
164. t1.join();
165. std::cout<<"Main Thread"<<std::endl;
166. return 0;
167. }
168. 运行结果：
169. 
170. 说明，通过thread 类直接申明一个线程t1,参数是这个线程执行的回调函数的地址，通过jion()方法阻塞主线程，直到t1线程执行结束为止。
172. C++11支持Lambda表达式，因此一个新线程的回调函数也可以是有一个Lambda表达式的形式，但是注意如果使用Lambda表达式最好不要使用引用的方式，应该使用值传递的方式来访问数据，在多线程中使用引用容易造成混乱。下面这个例子稍微复杂，创建了多个子线程，并使用了get\_id()方法来获取当前线程的id。
173. #include <thread>
174. #include <iostream>
175. #include <vector>
177. int main()
178. {
179. std::vector<std::thread> threads;
181. for(int i = 0; i < 5; ++i){
182. threads.push\_back(std::thread([](){
183. std::cout << "Hello from lamda thread " << std::this\_thread::get\_id() << std::endl;
184. }));
185. }
187. for(auto& thread : threads){
188. thread.join();
189. }
191. std::cout<<"Main Thread"<<"\t"<<std::this\_thread::get\_id()<<std::endl;
192. return 0;
193. }
194. 运行结果：
195. 
196. 上述代码中，使用vector来存放每个线程，线程的回调函数通过Lambda表达式产生，注意后面join的使用方式。
198. 可以通过sleep\_for来使线程睡眠一定的时间：
199. #include <thread>
200. #include <iostream>
201. #include <mutex>
202. using namespace std;
204. int main()
205. {
206. std::mutex m;
207. thread t1([&m]()
208. {
209. std::this\_thread::sleep\_for (chrono::seconds(10));
210. for(int i=0;i<10;i++)
211. {
212. m.lock();
213. cout <<  "In t1 ThreadID : " << std::this\_thread::get\_id() << ":" << i << endl;
214. m.unlock ();
215. }
216. } );
218. thread t2([&m]()
219. {
220. std::this\_thread::sleep\_for (chrono::seconds(1));
221. for(int i=0;i<10;i++)
222. {
223. m.lock ();
224. cout <<  "In t2 ThreadID : " << std::this\_thread::get\_id() << ":" << i << endl;
225. m.unlock();
226. }
227. } );
228. t1.join();
229. t2.join();
231. cout<<"Main Thread"<<endl;
233. return 0;
234. }
235. 运行结果：
236. 
237. 可以看出，由于线程t1睡眠的时间较长，t2先执行了。
238. 延时有这几种类型：nanoseconds、microseconds、milliseconds、seconds、minutes、hours。
239. 在使用多线程的程序中操作共享数据的时候一定要小心，由于线程的乱序执行，可能会得到意想不到的结果。通过下面的程序来看：
240. #include <thread>
241. #include <iostream>
242. #include <vector>
243. #include <mutex>
245. struct Counter {
246. std::mutex mutex;
247. int value;
249. Counter() : value(0) {}
251. void increment(){
252. // mutex.lock();                【1】表示没有使用锁
253. ++value;
254. // mutex.unlock();              【1】
255. }
257. void decrement(){
258. mutex.lock();
259. --value;
260. mutex.unlock();
261. }
262. };
264. int main(){
265. Counter counter;
267. std::vector<std::thread> threads;
269. for(int i = 0; i < 5; ++i){
270. threads.push\_back(std::thread([&](){
271. for(int i = 0; i < 10000; ++i){
272. counter.increment();
273. }
274. }));
275. }
277. for(auto& thread : threads){
278. thread.join();
279. }
281. std::cout << counter.value << std::endl;
283. return 0;
284. }
285. 运行结果：
286. 【1】
287. 
288. 运行结果：（使用了锁）
289. 
290. 说明：由于创建线程是使用lambda表达式，并使用引用的方式访问counter这个变量，当没有使用lock来保护的时候（情况【1】），执行的结果可能不像预期的5000（程序的意思是每个线程使counter中的value自加1000次，5个线程运行结束的时候应该是5000），当没有使用锁的时候自加的操作可能被其他线程打断，因此结果可能会小于5000。

[std::thread](http://en.cppreference.com/w/cpp/thread/thread)类代表了一个可执行的线程，它来自头文件<thread>。与其它创建线程的API（比如 Windows API中的[CreateThread](http://msdn.microsoft.com/en-us/library/windows/desktop/ms682453.aspx)）不同的是， 它可以使用普通函数、lambda函数以及仿函数（实现了operator（）函数的类）。另外，它还允许向线程函数传递任意数量的参数。

1. #include <thread> void func()
2. { // do some work } int main()
3. {
4. std::thread t(func);
5. t.join(); return 0;
6. }

在上面的例子中，t是一个线程对象，函数func()运行于该线程之中。调用[join](http://en.cppreference.com/w/cpp/thread/thread/join)函数后，该调用线程（本例中指的就是主线程）就会在join进来进行执行的线程t结束执行之前，一直处于阻塞状态。如果该线程函数执行结束后返回了一个值，该值也将被忽略。不过，该函数可以接受任意数量的参数。

1. void func(int i, double d, const std::string& s)
2. {
3. std::cout << i << ", " << d << ", " << s << std::endl;
4. } int main()
5. {
6. std::thread t(func, 1, 12.50, "sample");
7. t.join(); return 0;
8. }

尽管我们可以向线程函数传递任意数量的参数，但是，所有的参数都是按值传递的。如果需要将参数按引用进行传递，那么就一定要象下例所示一样，把该参数封装到 [std::ref或者std::cref](http://en.cppreference.com/w/cpp/utility/functional/ref)之中。

1. void func(int& a)
2. {
3. a++;
4. } int main()
5. { int a = 42;
6. std::thread t(func, std::ref(a));
7. t.join();
9. std::cout << a << std::endl; return 0;
10. }

上面程序打印结果为43，但要不是将a封装到std::ref之中的话，输出的将是42。

除join方法之外，这个线程类还提供了另外几个方法：

* [swap](http://en.cppreference.com/w/cpp/thread/thread/swap): 将两个线程对象的底层句柄进行交换
* [detatch](http://en.cppreference.com/w/cpp/thread/thread/detach): 允许执行该方法的线程独立于本线程对象的执行而继续执行。脱离后的线程就再也不能执行join了（你不能等待到它执行结束了）

1. <span style="font-family:'Courier New', Arial;font-size:9pt;line-height:1.5;">int</span><span style="font-family:'Courier New', Arial;font-size:9pt;line-height:1.5;"> main()</span> {
2. std::thread t(funct);
3. t.detach(); return 0;
4. }

有一点非常重要，值得注意：线程函数中要是抛出了异常的话，使用通常的try-catch方式是捕获不到该异常的。换句话说，下面这种做法行不通：

1. try {
2. std::thread t1(func);
3. std::thread t2(func);
5. t1.join();
6. t2.join();
7. } catch(const std::exception& ex)
8. {
9. std::cout << ex.what() << std::endl;
10. }

要在线程间传递异常，你可以先在线程函数中捕获它们，然后再将它们保存到一个合适的地方，随后再让另外一个线程从这个地方取得这些异常。

1. std::vector<std::exception\_ptr>  g\_exceptions; void throw\_function()
2. { throw std::exception("something wrong happened");
3. } void func()
4. { try {
5. throw\_function();
6. } catch(...)
7. {
8. std::lock\_guard<std::mutex> lock(g\_mutex);
9. g\_exceptions.push\_back(std::current\_exception());
10. }
11. } int main()
12. {
13. g\_exceptions.clear();
15. std::thread t(func);
16. t.join(); for(auto& e : g\_exceptions)
17. { try { if(e != nullptr)
18. {
19. std::rethrow\_exception(e);
20. }
21. } catch(const std::exception& e)
22. {
23. std::cout << e.what() << std::endl;
24. }
25. } return 0;
26. }

要获得更多关于捕获并传递异常的知识，你可以阅读[在主线程中处理工作线程抛出的C++异常](http://binglongx.wordpress.com/2010/01/03/handling-c-exceptions-thrown-from-worker-thread-in-the-main-thread/)以及[怎样才能在线程间传递异常？](http://stackoverflow.com/questions/233127/how-can-i-propagate-exceptions-between-threads)。

在深入讨论之前还有一点值得注意，头文件<thread>里还在命名空间std::this\_thread中提供了一些辅助函数：

* [get\_id](http://en.cppreference.com/w/cpp/thread/get_id): 返回胆怯线程的id
* [yield](http://en.cppreference.com/w/cpp/thread/yield): 让调度器先运行其它的线程，这在忙于等待状态时很有用
* [sleep\_for](http://en.cppreference.com/w/cpp/thread/sleep_for): 将当前线程置于阻塞状态，时间不少于参数所指定的时间段
* [sleep\_util](http://en.cppreference.com/w/cpp/thread/sleep_until): 在指定的时刻来临前，一直将当前的线程置于阻塞状态

**锁**

在上一个例子中，我需要对g\_exceptions这个vector进行同步访问，以确保同一个时刻只能有一个线程向其中压入新元素。为了实现同步，我使用了一个互斥量，并在该互斥量上进行了锁定。互斥量是一个核心的同步原语，C++11的<mutex>头文件中包含了四种不同的互斥量。

* [mutex](http://en.cppreference.com/w/cpp/thread/mutex): 提供了核心的[lock()](http://en.cppreference.com/w/cpp/thread/mutex/lock)函数和[unlock()](http://en.cppreference.com/w/cpp/thread/mutex/unlock)函数，以及非阻塞式的[try\_lock()](http://en.cppreference.com/w/cpp/thread/mutex/try_lock)方法，该方法在互斥量不可用时会立即返回。
* [recursive\_mutex](http://en.cppreference.com/w/cpp/thread/recursive_mutex): 运行在同一线程中，多次获得同一个互斥量。
* [timed\_mutex](http://en.cppreference.com/w/cpp/thread/timed_mutex): 同第一条中的mutex类似，但它还带来了另外两个方法[try\_lock\_for()](http://en.cppreference.com/w/cpp/thread/timed_mutex/try_lock_for)和[try\_lock\_until()](http://en.cppreference.com/w/cpp/thread/timed_mutex/try_lock_until)，分别用于在某个时间段内或在某个时刻到来之前获得该互斥量。
* [recursive\_timed\_mutex](http://en.cppreference.com/w/cpp/thread/recursive_timed_mutex): 结合了第二条的timed\_mutex和第三条的recusive\_mutex。

以下所列就是一个使用std::mutex（注意其中get\_id()和sleep\_for()这两个前文所述的辅助函数的用法）的例子。

1. #include <iostream>
2. #include <thread>
3. #include <mutex>
4. #include <chrono>
6. std::mutex g\_lock; void func()
7. {
8. g\_lock.lock();
10. std::cout << "entered thread " << std::this\_thread::get\_id() << std::endl;
11. std::this\_thread::sleep\_for(std::chrono::seconds(rand() % 10));
12. std::cout << "leaving thread " << std::this\_thread::get\_id() << std::endl;
14. g\_lock.unlock();
15. } int main()
16. {
17. srand((unsigned int)time(0));
19. std::thread t1(func);
20. std::thread t2(func);
21. std::thread t3(func);
23. t1.join();
24. t2.join();
25. t3.join(); return 0;
26. }

其输出将类似如下所示：

1. entered thread 10144 leaving thread 10144 entered thread 4188 leaving thread 4188 entered thread 3424 leaving thread 3424

lock()和unlock()这两个方法顾名思义，头一个方法用来对互斥量进行加锁，如果互斥量不可得便会处于阻塞状态；第二个方法用来对互斥量进行解锁。

接下来的这个例子演示的是一个简单的线程安全的容器（内部使用的是std::vector）。这个容器具有添加单个元素的add()方法以及添加一批元素的addrange()方法，addrange()方法内只是简单的调用了add()方法。

1. template <typename T> class container
2. {
3. std::mutex \_lock;
4. std::vector<T> \_elements; public: void add(T element)
5. {
6. \_lock.lock();
7. \_elements.push\_back(element);
8. \_lock.unlock();
9. } void addrange(int num, ...)
10. {
11. va\_list arguments;
13. va\_start(arguments, num); for (int i = 0; i < num; i++)
14. {
15. \_lock.lock();
16. add(va\_arg(arguments, T));
17. \_lock.unlock();
18. }
20. va\_end(arguments);
21. } void dump()
22. {
23. \_lock.lock(); for(auto e : \_elements)
24. std::cout << e << std::endl;
25. \_lock.unlock();
26. }
27. }; void func(container<int>& cont)
28. {
29. cont.addrange(3, rand(), rand(), rand());
30. } int main()
31. {
32. srand((unsigned int)time(0));
34. container<int> cont;
36. std::thread t1(func, std::ref(cont));
37. std::thread t2(func, std::ref(cont));
38. std::thread t3(func, std::ref(cont));
40. t1.join();
41. t2.join();
42. t3.join();
44. cont.dump(); return 0;
45. }

这个程序执行起来会进入死锁状态。其原因在于，该容器多次尝试获取同一个互斥量而之前却并没有释放该互斥量，这么做是行不通的。这正是[std::recursive\_mutex](http://en.cppreference.com/w/cpp/thread/recursive_mutex)的用武之地，它允许同一个线程多次获得同一个互斥量，可重复获得的最大次数并未具体说明，但一旦查过一定次数，再对lock进行调用就会抛出[std::system错误](http://en.cppreference.com/w/cpp/error/system_error)。为了修复上面所列代码的死锁问题（不通过修改addrange方法的实现，让它不对lock和unlock方法进行调用)，我们可以将互斥量改为[std::recursive\_mutex](http://en.cppreference.com/w/cpp/thread/recursive_mutex)。

1. template <typename T> class container
2. {
3. std::recursive\_mutex \_lock; // ...
4. };

经过修改之后，该程序的输出会同如下所示类似：

1. 6334 18467 41 6334 18467 41 6334 18467 41

明眼的读者可能已经发现了，每次调用func()所产生的数字序列都完全相同。这是因为对srad的初始化是要分线程进行的，对[srand()](http://www.cplusplus.com/reference/cstdlib/srand/)的调用只是在主线程中进行了初始化。在其它的工作线程中，srand并没有得到初始化，所以每次产生的数字序列就是完全相同的了。

显式的加锁和解锁可能会导致一定的问题，比如忘了解锁或者加锁的顺序不对都有可能导致死锁。本标准提供了几个类和函数用于帮助解决这类问题。使用这些封装类就能够以相互一致的、RAII风格的方式使用互斥量了，它们可以在相应的代码块的范围内进行自动的加锁和解锁动作。这些封装类包括：

* [lock\_guard](http://en.cppreference.com/w/cpp/thread/lock_guardv): 该类的对象在构造之时会试图获得互斥量的拥有权（通过调用lock()实现），而在析构之时会自动释放它所获得的互斥量（通过调用unlock()实现）。这是一个不可复制的类。
* [unique\_lock](http://en.cppreference.com/w/cpp/thread/unique_lock): 是一个通用的互斥量封装类。与lock\_quard不同，它还支持延迟加锁、时间锁、递归锁、锁所有权的转移并且还支持使用条件变量。这也是一个不可复制的类，但它是可以移动的类。

使用这些封装类，我们可以象这样来改写我们的容器：

1. template <typename T> class container
2. {
3. std::recursive\_mutex \_lock;
4. std::vector<T> \_elements; public: void add(T element)
5. {
6. std::lock\_guard<std::recursive\_mutex> locker(\_lock);
7. \_elements.push\_back(element);
8. } void addrange(int num, ...)
9. {
10. va\_list arguments;
12. va\_start(arguments, num); for (int i = 0; i < num; i++)
13. {
14. std::lock\_guard<std::recursive\_mutex> locker(\_lock);
15. add(va\_arg(arguments, T));
16. }
18. va\_end(arguments);
19. } void dump()
20. {
21. std::lock\_guard<std::recursive\_mutex> locker(\_lock); for(auto e : \_elements)
22. std::cout << e << std::endl;
23. }
24. };

有人会说，既然dump()方法并不会对容器的状态做出任何修改，所以它应该定义为congst的方法。但要是你真的这么改了之后，编译器就会报告出如下的错误：

1. ‘std::lock\_guard<\_Mutex>::lock\_guard(\_Mutex &)' : cannot convert parameter 1 from ‘const std::recursive\_mutex' to ‘std::recursive\_mutex &'

互斥量（无论使用的是哪一种实现）必须要获得和释放，这就意味着要调用非常量型的lock()和unlock()方法。所以，从逻辑上讲，lock\_guard不能在定义中添加const（因为该方法定义为const的话，互斥量也就必需是const的了）这个问题有个解决办法，可以让 mutex变为mutable的。成为 mutable之后就可以在const函数中对状态进行修改了。不过，这种用法应该只用于隐藏的或者“元”状态（比如，对计算结果或者查询到的数据进行缓存，以供下次调用时直接使用而无需再次计算或查询；再比如，对 只是对对象的实际状态起着辅助作用的互斥量中的位进行修改）。

1. template <typename T> class container
2. {
3. mutable std::recursive\_mutex \_lock;
4. std::vector<T> \_elements; public: void dump() const {
5. std::lock\_guard<std::recursive\_mutex> locker(\_lock); for(auto e : \_elements)
6. std::cout << e << std::endl;
7. }
8. };

这些封装类都具有可以接受一个用来指导加锁策略的参数的构造器，可用的加锁策略有：

* defer\_lockof typedefer\_lock\_t: 不要取得互斥量的拥有权
* try\_to\_lockof typetry\_to\_lock\_t: 在不会被阻塞的情况下尝试获得互斥量的拥有权
* adopt\_lockof typeadopt\_lock\_t: 假设调用线程已经获得了互斥量的拥有权

这些策略的定义如下所示：

1. struct defer\_lock\_t { };
2. struct try\_to\_lock\_t { };
3. struct adopt\_lock\_t { };
4. constexpr std::defer\_lock\_t defer\_lock = std::defer\_lock\_t();
5. constexpr std::try\_to\_lock\_t try\_to\_lock = std::try\_to\_lock\_t();
6. constexpr std::adopt\_lock\_t adopt\_lock = std::adopt\_lock\_t();

除了这些互斥量的封装类，本标准还提供了几个用来对一个或多个互斥量进行加锁的方法。

* [lock](http://en.cppreference.com/w/cpp/thread/lock): 使用一种可避免死锁的算法对互斥量进行加锁（通过调用tolock()、try\_lock()以及unlock()）。
* [try\_lock](http://en.cppreference.com/w/cpp/thread/try_lock): 通过调用try\_lock()i按照参数里指定的互斥量的顺序对多个互斥量进行加锁。

这里举一个造成死锁的例子：我们有一个保存元素的容器，还有一个叫做exchange()的方法，用来将一个元素从一个容器中取出来放入另外一个容器。为了成为线程安全的函数，这个函数通过获得每个容器的互斥量，对两个容器的访问进行了同步处理。

1. template <typename T> class container
2. { public:
3. std::mutex \_lock;
4. std::set<T> \_elements; void add(T element)
5. {
6. \_elements.insert(element);
7. } void remove(T element)
8. {
9. \_elements.erase(element);
10. }
11. }; void exchange(container<int>& cont1, container<int>& cont2, int value)
12. {
13. cont1.\_lock.lock();
14. std::this\_thread::sleep\_for(std::chrono::seconds(1)); // <-- forces context switch to simulate the deadlock  cont2.\_lock.lock();
16. cont1.remove(value);
17. cont2.add(value);
19. cont1.\_lock.unlock();
20. cont2.\_lock.unlock();
21. }

假设这个函数是从两个不同的线程中进行调用的，在第一个线程中有一个元素从第一个容器中取出来，放到了第二个容器中，在第二个线程中该元素又从第二个容器中取出来放回到了第一个容器中。这样会导致死锁（如果线程上下文正好在获得第一个锁的时候从一个线程切换到了另一个线程的时候就会发生死锁）。

1. int main()
2. {
3. srand((unsigned int)time(NULL));
5. container<int> cont1;
6. cont1.add(1);
7. cont1.add(2);
8. cont1.add(3);
10. container<int> cont2;
11. cont2.add(4);
12. cont2.add(5);
13. cont2.add(6);
15. std::thread t1(exchange, std::ref(cont1), std::ref(cont2), 3);
16. std::thread t2(exchange, std::ref(cont2), std::ref(cont1), 6);
18. t1.join();
19. t2.join(); return 0;
20. }

要解决该问题，你可以使用以能够避免死锁的方式获得锁的std::lock：

1. void exchange(container<int>& cont1, container<int>& cont2, int value)
2. {
3. std::lock(cont1.\_lock, cont2.\_lock);
5. cont1.remove(value);
6. cont2.add(value);
8. cont1.\_lock.unlock();
9. cont2.\_lock.unlock();
10. }

**条件变量**

C++11还提供了对另外一个同步原语的支持，这个原语就是[条件变量](http://en.wikipedia.org/wiki/Condition_variable" \l "Waiting_and_signaling)。使用条件变量可以将一个或多个线程进入阻塞状态，直到收到另外一个线程的通知，或者超时或者发生了[虚假唤醒](http://en.wikipedia.org/wiki/Spurious_wakeup)，才能退出阻塞状态。头文件<condition\_variable>中包含的条件变量有两种实现：

* [condition\_variable](http://en.cppreference.com/w/cpp/thread/condition_variable): 要求任何想等待该条件变量的线程必需先获得[std::unique\_lock](http://en.cppreference.com/w/cpp/thread/unique_lock)锁。
* [condition\_variable\_any](http://en.cppreference.com/w/cpp/thread/condition_variable_any): 该实现更加通用，它可以用于任何满足基本条件的锁（只要实现了lock()和unlock()方法即可）。因为它使用起来代价要更高一些（从性能和操作系统的字样的角度讲），所以，应该在只有它所提供的额外的灵活性是必不可少的情况下才会选用它。

下面说说条件变量的工作原理：

* 必须至少要有一个等待条件变为true的线程。等待中的线程必须首先获得一个unique\_lock锁。 该锁将会传递给wait()方法，然后wait()方法会释放互斥量并将该线程暂停，直到条件变量得到相应的信号。当接受到信号，线程被唤醒后，该锁就又被重新获得了。
* 必须至少要有一个线程发送信号使得条件变为true。信号可以通过调用[notify\_one()](http://en.cppreference.com/w/cpp/thread/condition_variable/notify_one)来发送，发用这个方法发送后就会将处于阻塞状态的等待该条件获得信号的线程中的某一个线程（任意一个线程）恢复执行；还可以通过调用[notify\_all()](http://en.cppreference.com/w/cpp/thread/condition_variable/notify_all)将等待该条件的所以线程唤醒。
* 因为在多处理器的环境下，要让条件唤醒成为完全可预测会有一些复杂情况难以克服，所以就会出现一些虚假唤醒。也就是说，线程甚至在没有人向条件变量发送信号的情况下就有可能会被唤醒。因此，在线程唤醒后，仍然需要检测条件是不是还为true。而且因为虚假唤醒可能会多次发生，所以该检测必须用一个循环来进行。

以下代码给出了一个利用状态变量来同步线程的例子：几个工作线程可能在他们运行的时候产生错误并且他们把这些错误放到队列里面。一个记录线程会通过从队列得到并输出错误来处理这些错误代码。当有错误发生的时候，工作线程会发信号给记录线程。记录线程一直在等待着状态变量接收信号。为了防止虚假的唤醒，所以记录线程的等待是发生在一个以检测布尔值(boolean)的循环之中的。

1. #include <thread>
2. #include <mutex>
3. #include <condition\_variable>
4. #include <iostream>
5. #include <queue>
6. #include <random>
8. std::mutex              g\_lockprint;
9. std::mutex              g\_lockqueue;
10. std::condition\_variable g\_queuecheck;
11. std::queue<int>         g\_codes; bool g\_done; bool g\_notified; void workerfunc(int id, std::mt19937& generator)
12. { // print a starting message  {
13. std::unique\_lock<std::mutex> locker(g\_lockprint);
14. std::cout << "[worker " << id << "]\trunning..." << std::endl;
15. } // simulate work  std::this\_thread::sleep\_for(std::chrono::seconds(1 + generator() % 5)); // simulate error  int errorcode = id\*100+1;
16. {
17. std::unique\_lock<std::mutex> locker(g\_lockprint);
18. std::cout  << "[worker " << id << "]\tan error occurred: " << errorcode << std::endl;
19. } // notify error to be logged  {
20. std::unique\_lock<std::mutex> locker(g\_lockqueue);
21. g\_codes.push(errorcode);
22. g\_notified = true;
23. g\_queuecheck.notify\_one();
24. }
25. } void loggerfunc()
26. { // print a starting message  {
27. std::unique\_lock<std::mutex> locker(g\_lockprint);
28. std::cout << "[logger]\trunning..." << std::endl;
29. } // loop until end is signaled  while(!g\_done)
30. {
31. std::unique\_lock<std::mutex> locker(g\_lockqueue); while(!g\_notified) // used to avoid spurious wakeups  {
32. g\_queuecheck.wait(locker);
33. } // if there are error codes in the queue process them  while(!g\_codes.empty())
34. {
35. std::unique\_lock<std::mutex> locker(g\_lockprint);
36. std::cout << "[logger]\tprocessing error:  " << g\_codes.front()  << std::endl;
37. g\_codes.pop();
38. }
40. g\_notified = false;
41. }
42. } int main()
43. { // initialize a random generator  std::mt19937 generator((unsigned int)std::chrono::system\_clock::now().time\_since\_epoch().count()); // start the logger  std::thread loggerthread(loggerfunc); // start the working threads  std::vector<std::thread> threads; for(int i = 0; i < 5; ++i)
44. {
45. threads.push\_back(std::thread(workerfunc, i+1, std::ref(generator)));
46. } // work for the workers to finish  for(auto& t : threads)
47. t.join(); // notify the logger to finish and wait for it  g\_done = true;
48. loggerthread.join(); return 0;
49. }
50. Running this code produces an output that looks like this (notice this output is different with each run because each worker thread works, i.e. sleeps, for a random interval):
51. [logger]        running...
52. [worker 1]      running...
53. [worker 2]      running...
54. [worker 3]      running...
55. [worker 4]      running...
56. [worker 5]      running...
57. [worker 1]      an error occurred: 101 [worker 2]      an error occurred: 201 [logger]        processing error: 101 [logger]        processing error: 201 [worker 5]      an error occurred: 501 [logger]        processing error: 501 [worker 3]      an error occurred: 301 [worker 4]      an error occurred: 401 [logger]        processing error: 301 [logger]        processing error: 401

如上所示的wait()方法有两个重载：

1.一个是只有一个唯一锁；这个重载释放锁，封锁线程和把线程加入都是等待这一个状态变量的线程队列里面；当状态变量被信号通知后或者是一个假唤醒发生，这些线程就会被唤醒。但他们中任何一个发生时，锁就被重新获得然后函数返回。

2.另外一个是对于唯一锁的添加，它也是使用一个循环的谓语直到它返回false;这个重载可以用来防止假式唤醒。它基本上是与以下是等价的：

1. while(!predicate())
2. wait(lock);

因此在上例中，通过使用重载的wait函数以及一个验证队列状态（空或不空）的断言，就可以避免使用布尔变量g\_notified了：

1. void workerfunc(int id, std::mt19937& generator)
2. { // print a starting message  {
3. std::unique\_lock<std::mutex> locker(g\_lockprint);
4. std::cout << "[worker " << id << "]\trunning..." << std::endl;
5. } // simulate work  std::this\_thread::sleep\_for(std::chrono::seconds(1 + generator() % 5)); // simulate error  int errorcode = id\*100+1;
6. {
7. std::unique\_lock<std::mutex> locker(g\_lockprint);
8. std::cout << "[worker " << id << "]\tan error occurred: " << errorcode << std::endl;
9. } // notify error to be logged  {
10. std::unique\_lock<std::mutex> locker(g\_lockqueue);
11. g\_codes.push(errorcode);
12. g\_queuecheck.notify\_one();
13. }
14. } void loggerfunc()
15. { // print a starting message  {
16. std::unique\_lock<std::mutex> locker(g\_lockprint);
17. std::cout << "[logger]\trunning..." << std::endl;
18. } // loop until end is signaled  while(!g\_done)
19. {
20. std::unique\_lock<std::mutex> locker(g\_lockqueue);
22. g\_queuecheck.wait(locker, [&](){return !g\_codes.empty();}); // if there are error codes in the queue process them  while(!g\_codes.empty())
23. {
24. std::unique\_lock<std::mutex> locker(g\_lockprint);
25. std::cout << "[logger]\tprocessing error:  " << g\_codes.front() << std::endl;
26. g\_codes.pop();
27. }
28. }
29. }

除了这个wait()重载方法，还有另外两个进行类似重载的等待方法，都有用了一个用来避免虚假唤醒的断言：

* [wait\_for](http://en.cppreference.com/w/cpp/thread/condition_variable/wait_for): 在条件变量收到信号或者指定的超时发生前，一直都将线程置于阻塞状态。
* [wait\_until](http://en.cppreference.com/w/cpp/thread/condition_variable/wait_until): 在条件变量收到信号或者指定的时刻到来前，一直都将线程处于阻塞状态。

这两个函数不带断言的重载函数会返回一个[cv\_status](http://en.cppreference.com/w/cpp/thread/cv_status)状态，该状态用来表明线程被唤醒了到底是因为发生了超时还是因为条件变量收到了信号抑或是发生了虚假唤醒。

本标准还提供了一个叫做[notified\_all\_at\_thread\_exit](http://en.cppreference.com/w/cpp/thread/notify_all_at_thread_exit)的函数，它实现了一种机制，在该机制下，我们可以通知其它线程，某个给定的线程执行结束了，并销毁了所有的thread\_local对象。之所以引入该函数，是因为如果使用了thread\_local后，采用join()之外的机制等待线程可能会导致不正确甚至是致命的行为，出现这样的问题是因为 thread\_local的析构函数甚至可能会在原本处于等待中的线程继续执行后被执行了而且还可能已经执行完成了。（有关这方面更多的情况可参见[N3070](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3070.html)和[N2880](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2880.html)）。 一般情况下，notified\_all\_at\_thread\_exitTypically必须正好在线程生成前调用。下面给出一个例子，演示一下 notify\_all\_at\_thread\_exit是如何同condition\_variable一起使用来对两个线程进行同步处理的：

1. std::mutex              g\_lockprint;
2. std::mutex              g\_lock;
3. std::condition\_variable g\_signal; bool g\_done; void workerfunc(std::mt19937& generator)
4. {
5. {
6. std::unique\_lock<std::mutex> locker(g\_lockprint);
7. std::cout << "worker running..." << std::endl;
8. }
10. std::this\_thread::sleep\_for(std::chrono::seconds(1 + generator() % 5));
12. {
13. std::unique\_lock<std::mutex> locker(g\_lockprint);
14. std::cout << "worker finished..." << std::endl;
15. }
17. std::unique\_lock<std::mutex> lock(g\_lock);
18. g\_done = true;
19. std::notify\_all\_at\_thread\_exit(g\_signal, std::move(lock));
20. } int main()
21. { // initialize a random generator  std::mt19937 generator((unsigned int)std::chrono::system\_clock::now().time\_since\_epoch().count());
23. std::cout << "main running..." << std::endl;
25. std::thread worker(workerfunc, std::ref(generator));
26. worker.detach();
28. std::cout << "main crunching..." << std::endl;
30. std::this\_thread::sleep\_for(std::chrono::seconds(1 + generator() % 5));
32. {
33. std::unique\_lock<std::mutex> locker(g\_lockprint);
34. std::cout << "main waiting for worker..." << std::endl;
35. }
37. std::unique\_lock<std::mutex> lock(g\_lock); while(!g\_done) // avoid spurious wake-ups  g\_signal.wait(lock);
39. std::cout << "main finished..." << std::endl; return 0;
40. }

如果工作线程是在主线程结束之前结束的，输出将会是如下所示：

1. main running...
2. worker running...
3. main crunching...
4. worker finished...
5. main waiting for worker...
6. main finished...

如果是主线程在工作线程结束之前结束的，输出将会是如下所示：

1. main running...
2. worker running...
3. main crunching...
4. main waiting for worker...
5. worker finished...
6. main finished...

**结束语**

C++11标准使得C++开发人员能够以一种标准的和平台独立的方式来编写多线程代码。本文一一讲述了标准所支持的线程和同步机制。<thread>头文件提供了名为thread的类（另外还包含了一些辅助类或方法），该类代表了一个执行线程。头文件<mutex>提供了几种互斥量的实现，以及对线程进行同步访问的封装类。头文件<condition\_variable>为条件变量提供了两种实现，利用这些实现可以让一个或多个线程进入阻塞状态，直到从收到来自另外一个或多个线程的通知、或者发生超时或虚假唤醒为止才会被唤醒。推荐在这方面再阅读一些别的资料来获得更详细的信息。