# 并发、进程、线程的基本概念和综述

## 1、并发

两个或者更多的任务（独立的活动）同时发生（进行）：一个程序同时执行多个独立的

任务；以往计算机，单核CPU（中央处理器）；某一时刻只能执行一个任务：有操作系统调度，每秒钟进行多次所谓的“任务切换”，并发的假象（不是真正的并发）；这种切换（上下文切换）是要时间开销的，比如操作系统要保存你切换时的各种状态，执行进度等信息，都需要时间，一会儿切换回来的时候要复原这些信息。

硬件发展，出现了多处理器计算机：用于服务器和高性能计算领域。台式机：在一块芯片上有多核（多个CPU）：双核、4核、8核、10核…能够实现真正的并行执行多个任务（硬件并发）；

## 2、可执行程序

磁盘上的一个文件，windows下，一个扩展名为.exe的。Linux下，一个扩展名为.out的。

## 3、进程

简单的说就是一个可执行程序运行起来了，就叫创建了一个进程。

## 4、线程

* 1. 每个进程（执行起来的可执行程序），都有一个主线程，这个主线程是唯一的，也就是一个进程中只能有一个主线程。
  2. 当你执行一个可执行程序，产生一个进程后，这个主线程就随着这个进程默默的启动起来了。例如在VS中，ctrl + f5运行这个程序的时候，实际上就是进程的主线程来执行（调用）这个main函数中的代码。（主线程与进程唇齿相依）。除了主线程之外，我们可以通过编写自己的代码来创建其他线程，其他线程走的是别的道路。

线程不是越多越好，每个线程，都需要一个独立的堆栈空间，线程之间的切换要保存很多中间状态；切换回耗费本该属于程序运行的时间。

* 1. 程序运行起来，生成一个进程，该进程所属的主线程开始自动运行；主线程从main()开始执行，那么我们自己创建的线程，也需要从一个函数开始运行（初始函数），一旦这个函数运行完毕，就代表着我们这个线程运行结束。整个程序是否执行完毕的标志是 主线程是否执行完，如果主线程执行完毕了，就代表整个进程执行完毕了；此时，一般情况下，如果其它子线程还没有执行完毕，那么这些子线程也会被操作系统强行终止。

## 总结

1. 线程是用来执行代码的。
2. 把线程这个东西理解成一条代码的执行通路（道路），一个新线程代表一条新的通路。
3. 一个线程自动包含一个主线程，主线程随着进程默默的启动并执行，我们可以通过编码来创建多个其他线程（非主线程）。
4. 因为主线程是自动启动的，所以一个进程中最少也是有一个线程（主线程）。
5. 多线程程序可以同时干多个事，所有运行效率高。但是到底有多高，并不是一个很容易评估和量化的东西。

# 并发的实现方法

两个或者更多的任务（独立的活动）同时发生（进行）。

**实现并发的手段：**

1. 我们通过多个进程实现并发。
2. 在单独的进程中，我创建多个线程来实现并发：自己写代码来创建除了主线程之外的其他线程。

## 多进程并发

Word启动后就是进程，IE浏览器启动也是进程。

进程之间的通信（同一个电脑上：管道、文件、消息队列、共享内存）；

不同的电脑上：socked通信技术。

## 2、多线程并发：单个进程中，创建了多个线程。

线程：感觉像轻量级的进程。每个线程都有自己的独立的运行路径，但是一个进程中的所有线程共享地址空间（共享内存）。全局变量，指针，引用都可以在线程之间传递，所以：使用多线程开销远远小于多进程。

共享内存带来的新问题，数据一致性的问题，多进程并发和多线程并发虽然可以混合使用，但是建议优先考虑多线程技术手段而不是多进程；

## 总结

和进程相比，线程有如下优点：

1. 线程启动速度更快，更轻量级；
2. 系统资源开销更少，执行速度更快，比如共享内存这种通信方式比任何其他的通信方式都快；

缺点：

1. 使用有一定难度，要小心处理数据的一致性问题。

# C++11新标准线程库

以往：windows：GreateThread(), \_beginthred(), \_beginthreadexe() 创建线程

Linux：pthread\_create(); 创建线程。

临界区，互斥量；

以往多线程代码不能跨平台；

POSIX thread(pthread; 这种方法可以跨平台，但是要做一番配置，用起来不是那么方便。

从C++11新标准，C++语言本身增加对多线程的支持，以为着可移植性（跨平台），这大大减少开发人员工作量了；

# 创建线程的多种方法（线程的启动结束）

## 1、创建多线程程序（VS2017中）

a) 包含一个头文件<thread>。

b) 创建一个线程的初始函数。

c) main 函数中写代码（）。

eg:

1. #include <iostream>
2. #include <thread>
3. **using** **namespace** std;
4. **void** myThread()
5. {
6. cout << “我的线程开始执行了！！！” << endl;
7. /\*
8. 实现的功能；
9. \*/
10. cout << “我的线程执行完毕了！！！” << endl;
11. }
12. **int** main()
13. {
14. //创建线程对象
15. **thread** myToJob(myThread);
16. //阻塞主线程并等待子线程执行完毕
17. myToJob.join();
18. }

thread myToJob(myThread);

作用：

a) 创建了线程，线程执行起点（入口）myThread()。

b) myThreadx线程开始执行。

myToJob.join();

作用：

a) 主线程阻塞到这里等待myThread()执行完，当子线程执行完毕，这个join就执行完毕，主线程就继续往下走。

### （1）thread

a) 是个标准库里的类；

#### 类成员函数：

##### **join()**

加入/汇合；就是阻塞，阻塞主线程，让主线程等待子线程执行完毕，然后子线程和主线程汇合，然后主线程再往下走。如果主线程执行完毕了，子线程还没有执行完毕，这种代码是不合格的，程序也是不稳定的。

##### **detach()**

分离；也就是主线程和不和子线程汇合了，主线程可以不用等待子线程是否自行完毕而退出（传统多线程程序主线程要等待子线程执行完毕，然后自己再最后退出）。一旦detach()之后，与这个主线程关联的thread对象就会失去与这个主线程的关联。此时这个子线程就会驻留在后台运行。这个子线程就相当于被C++运行时库接管，当这个子线程执行完毕以后，由运行时库负责清理该线程相关的资源（守护线程）。

##### **joinable()**

判断是否可以成功使用join()或者detach()，返回true（可以join或者detach）或者返回false（不能join或者detach）。

#### 关于thread的传参问题

##### 传参类型为类

如果传递类对象，避免隐式类型转换。全部都在创建线程这一行就构建出临时对象来，然后在函数参数里，用引用来接，否则系统还会多构造以此对象，浪费资源。关于传参类型为类的情况，因为线程会调用类的拷贝构造函数，因此，就算是以下这种类型（使用类的引用），线程内依然无法改变main函数中类的成员变量：

1. #include <iostream>
2. #include <thread>
3. **using** **namespace** std;
5. **class** A
6. {
7. **public**:
8. **mutable** **int** m\_i;
9. //类型转换的构造函数，可以把一个int转换成一个类A对象。
10. A(**int** a) : m\_i(a) {cout << "[A::A(int a)构造函数执行]" << endl;}
11. A(**const** A &a) : m\_i(a.m\_i) {cout << "[A::A(const A)拷贝构造函数执行" << endl;}
12. ~A() {cout << "[A::~A()析构函数执行" << endl;}
13. }
15. **void** myPrint(**const** **int** i, **const** A &pmybuff)
16. {
17. pmybuff.m\_i = 0;
18. cout << &pmybuff << endl;
19. **return**;
20. }
22. **int** main()
23. {
24. **int** mvar = 1;
25. **A** mysecondpar(12);
26. **thread** myToJob(myPrint, mvar, mysecondpar);
28. myToJob.detach();
29. **return** 0;
30. }

mutable关键字是（可以理解为const的对立）允许修改由mutable声明的变量，即使我们在线程中修改类A中m\_i的值，也不会修改到main函数中传递进来的那么类A的对象mysecondpar，因为线程中的类是由线程重新构建的（我们可以打印类的拷贝构造函数来证实这一现象）。若要正真意义上的改变main函数中的类A的变量（反馈到main函数的mysecondpar对象），就要利用关键字std::ref()函数，这样才可以使线程中的改动反馈到main函数中。

std::ref 用于包装按引用传递的值。  
std::cref 用于包装按const引用传递的值。

##### 传参类型为智能指针

1. #include <iostream>
2. #include <thread>
3. **using** **namespace** std;
5. **void** MyThread(unique\_ptr<**int**> pmybuf)
6. {
7. cout << "子线程开始执行了！！！" << endl;
8. }
10. **int** main()
11. {
12. unique\_ptr<**int**> myp(**new** **int**(100));
13. std::**thread** myToJob(MyThread, std::move(myp));
14. **return** 0;
15. }

### （2）用类对象（可调用对象）

eg：

1. **class** TA
2. {
3. **int** &m\_i;
4. TA(**int** &i) : m\_i(i){}
5. **void** operator()(/\*参数\*/)   //不能带参数
6. {
7. cout << "我的线程operator()开始执行了" << endl;
8. cout << "我的线程operator()结束执行了" << endl;
9. cout << "m\_i的值为：" << m\_i << endl;
10. }
11. }
13. **int** main()
14. {
15. **int** myi = 6;
16. TA ta(myi);
17. **thread** myToJob(ta);
18. myToJob.join();
19. //myToJob.detach();
20. }

该段代码，同样手动创建了一个线程，但是，若采用detach()函数分离线程的话，那么上述的代码会出现一个预料之外的错误（当线程分离以后，主线程执行完后，进程退出，子线程由系统的运行时库接管，此时，由于引用的是主线程中的变量，而主线程已经退出，变量已经释放，因此子线程中引用的变量便会出现意料之外的错误）。

另外的一个疑问，一旦调用了detach()，那么主线程执行结束了，我这里用的这个对象ta还在吗？（对象不在了），该线程里的这个对象实际上是被复制到线程中去；执行完主线程后，它会被销毁，但是所复制的TA的对象依然存在。所以只要你这个TA类对象里没有引用，没有指针，那么就不会产生问题。（检测是否符合可以在类中实现构造函数、拷贝构造函数、析构函数来检测这种机制）。

### （3）用lambda表达式

eg：

1. #include <iostream>
2. #include <thread>
3. **using** **namespace** std;
5. **int** main()
6. {
7. auto myLamThread = []
8. {
9. cout << "我的线程函数开始执行了！！！" << endl;
10. /\*
11. 功能代码
12. \*/
13. cout << "我的线程函数执行结束了！！！" << endl;
14. };
16. **thread** myToJob(myLamThread);
17. myToJob.join();
18. //myToJob.detach();
20. cout << "我是主线程！！！" << endl;
21. **return** 0;
22. }

### （4）用成员函数指针做线程函数

1. #include <iostream>
2. #include <thread>
3. **using** **namespace** std;
5. **class** A
6. {
7. **public**:
8. **void** Thread\_work(**int** num)
9. {
10. cout << "类内子线程启动！！！" << endl;
11. }
12. }
14. **int** main()
15. {
16. A aaa;
17. **thread** myToJob(&A::Thread\_work, aaa, 15);
18. myToJob.join();
20. **return** 0;
21. }

默认情况下，该线程同样会调用类A的拷贝构造函数来构造一个类，因此，这里使用join()或是detach()都是可以的。若不想线程调用拷贝函数的话（即不在构造类A，而是直接使用main函数中的类的话，就要使用std::ref(aaa)或是&aaa函数了，此时如果再使用detach()的话就存在危险）。

## 多线程详解

### （1）多线程的创建

创建10个线程，线程入口函数统一使用myThread();

1. #include <iostream>
2. #include <thread>
3. #include <vector>
4. **using** **namespace** std;
6. **void** myThread(**int** num)
7. {
8. cout << "编号为" << num << "的线程开始执行了！！！" << endl;
9. /\*
10. 函数功能模块
11. \*/
12. cout << "编号为" << num << "的线程执行结束了！！！" << endl;
13. **return**;
14. }
16. **int** main()
17. {
18. vector<**thread**> myThreads;
19. **for** (**int** i = 0; i < 10; i++)
20. {
21. myThreads.push\_back(**thread**(myThread, i));
22. }
23. **for** (**int** i = 0; i < myThreads.size(); i++)
24. {
25. myThreads.at(i).join();
26. }
27. cout << "主线程函数执行！！！" << endl;
28. system("pause");
29. **return** 0;
30. }

#### 总结

* 1. 多个线程执行顺序是乱的，跟操作系统内部对线程的运行调度机制有关。
  2. 主线程等待所有子线程运行结束，最后主线程结束。

### （2）多线程的读写（互斥量）

#### a) 互斥量的基本概念：

保护共享数据，操作时，某个线程用代码吧共享数据锁住、操作数据、解锁。其他想操作共享数据的线程必须等待解锁。

#### b) 互斥量的用法：

lock(),unlock();

步骤：

先lock()，操作共享数据，unlock(); lock()和unlock() 要成对使用，有lock()必然要有unlock()；

eg：

1. #include <iostream>
2. #include <thread>
3. #include <list>
4. #include <mutex>      //线程锁的头文件
6. **using** **namespace** std;
7. **class** A
8. {
9. **public**:
10. //把收到的消息入到一个队列的线程
11. **void** inMesRecvQueue()
12. {
13. **for** (**int** i = 0; i < 100000; i++)
14. {
15. cout << "inMesRecvQueue()执行，插入一个元素" << i << endl;
16. **m\_mutex.lock();**
17. m\_msgRecvQueue.push\_back(i);
18. **m\_mutex.unlock();**
19. }
20. }
22. **bool** outMsgLULProc(**int** &command)
23. {
24. **m\_mutex.lock();**
25. **if** (!m\_msgRecvQueue.empty())
26. {
27. command = m\_msgRecvQueue.front();       //返回第一个元素，但不检查元素是否存在
28. m\_msgRecvQueue.pop\_front();             //移除第一个元素，但不返回
29. **m\_mutex.unlock();**
30. **return** **true**;
31. }
32. **m\_mutex.unlock();**
33. **return** **false**;
34. }
36. **void** outMsgRecvQueue()
37. {
38. **int** command = 0;
39. **for** (**int** i = 0; i < 100000; i++)
40. {
41. **bool** result = outMsgLULProc(command);
42. //消息不为空
43. **if** (result == **true**)
44. {
45. cout << "outMsgRecvQueue()执行，取出第一个元素" << command << endl;
46. }
47. **else**
48. {
49. //消息队列为空
50. cout << "outMsgRecvQueue()执行，但目前消息队列为空" << i << endl;
51. }
52. }
53. }
55. **private**:
56. list<**int**> m\_msgRecvQueue;
57. mutex m\_mutex;
58. };

61. **int** main()
62. {
63. A myToJob;
64. **thread** myInMsgJob(&A::inMesRecvQueue, &myToJob);
65. **thread** myOutMsgJob(&A::outMsgRecvQueue, &myToJob);
67. myInMsgJob.join();
68. myOutMsgJob.join();
70. **return** 0;
71. }

#### c) lock\_guard模板：

为防止忘记unlock(), 引入了一个叫 std::lock\_guard 的类模板：你忘记了unlock不要紧，我替你unlock，类似于智能指针，取代了lock() 和unlock()。

std::lock\_guard类模板：直接取代lock()和unlock()；也就是说，你用了lock\_guard之后，再不能使用lock()和unlock();

eg:

改造类A中的outMsgLULProc() 函数：

1. **bool** outMsgLULProc(**int** &command)
2. {
3. lock\_guard<mutex> m\_guard(m\_mutex);
4. //m\_mutex.lock();
5. **if** (!m\_msgRecvQueue.empty())
6. {
7. command = m\_msgRecvQueue.front();       //返回第一个元素，但不检查元素是否存在
8. m\_msgRecvQueue.pop\_front();             //移除第一个元素，但不返回
9. //m\_mutex.unlock();
10. **return** **true**;
11. }
12. //m\_mutex.unlock();
13. **return** **false**;
14. }

该模板的执行原理可以理解为：类lock\_guard在构建时执行构造函数时执行lock函数，在函数结束时，执行析构函数时执行unlock，以此来达到加锁解锁的操作。

#### d) unique\_lock

unique\_lock是一个模板类，工作中，一般使用lock\_guard（推荐使用）；unique\_lock比lock\_guard 灵活很多，效率上差一点，内存占用多一点。

##### 成员函数介绍：

###### lock()

在unique\_lock使用defer\_lock标志的时候，因为标志的作用是创建一个没有加锁的mutex；因此，创建完unique\_lock的对象后，还需调用该类的类函数lock()加锁。

###### unlock()

1. //没有加锁的m\_mutex
2. std::unique\_lock<std::mutex> m\_unique\_lock(m\_mutex, std::defer\_lock);
3. //不用自己unlock()
4. m\_unique\_lock.lock();

###### try\_lock()

尝试给互斥量加锁，如果拿不到锁，则返回false，如果拿到了锁，返回true，这个函数不阻塞的；

eg：

1. **bool** outMsgLULProc(**int** &command)
2. {
3. //lock\_guard<mutex> m\_guard(m\_mutex);
4. **unique\_lock<mutex> m\_uniqueLock(m\_mutex, defer\_lock);**
5. **if** (**m\_uniqueLock.try\_lock()**)
6. {
7. //拿到了锁
8. cout << "拿到了锁！！！！！！！！！！！！！！！！！！！！！！！！！！！！！" << endl;
9. **if** (!m\_msgRecvQueue.empty())
10. {
11. command = m\_msgRecvQueue.front();       //返回第一个元素，但不检查元素是否存在
12. m\_msgRecvQueue.pop\_front();             //移除第一个元素，但不返回
13. **return** **true**;
14. }
15. **return** **false**;
16. }
17. **else**
18. {
19. //没有拿到锁
20. cout << "没有拿到锁，尝试做一些其他的！！！" << endl;
21. }
22. }

前提是使用std::defer\_lock标志（即初始化了一个没有加锁的mutex）。

###### release()

返回它所管理的mutex对象的指针，并释放所有权；也就是说，这个unique\_lock和mutex不再有关系。如果原来的mutex对象处于加锁状态，你有责任接管过来并负责解锁。（release返回的是原始mutex的指针）。

eg：

1. **void** inMesRecvQueue()
2. {
3. **for** (**int** i = 0; i < 100000; i++)
4. {
5. cout << "inMesRecvQueue()执行，插入一个元素" << i << endl;
6. **unique\_lock<mutex> m\_uniqueLock(m\_mutex);**
7. **mutex \*ptx = m\_uniqueLock.release();**
8. m\_msgRecvQueue.push\_back(i);
9. **ptx->unlock();**
10. }
11. }

###### 扩充 ：

为什么有时候需要unlock()：因为你lock锁住的代码段越少，执行越快，整个程序运行速率越高。有人也把锁头锁住代码的多少称为锁的粒度，粒度一般用粗细来描述；

* + 1. 锁住的代码少，这个粒度叫细，执行效率高；
    2. 锁住的代码多，这个粒度叫粗，执行效率就低。

要学会尽量选择合适粒度的代码进行保护，粒度太细，可能漏掉共享数据的保护，粒度太粗，影响效率。

##### （2）unique\_lock的参数：

###### **1**、std::adopt\_lock：

表示这个互斥量已经被lock了（**你必须将这个互斥量提前lock了，否则会报异常**）。

###### 2、std::try\_to\_lock：

表示我们会尝试用mutex的lock()去锁定这个mutex，但是如果没有成功，我也会立即返回，并不会阻塞在那里，**用这个try\_to\_lock的前提是你自己不能先去lock**。

eg：

1. #include <iostream>
2. #include <thread>
3. #include <list>
4. #include <mutex>      //线程锁的头文件
6. **using** **namespace** std;
8. **class** A
9. {
10. **public**:
11. //把收到的消息入到一个队列的线程
12. **void** inMesRecvQueue()
13. {
14. **for** (**int** i = 0; i < 100000; i++)
15. {
16. cout << "inMesRecvQueue()执行，插入一个元素" << i << endl;
17. **lock\_guard<mutex> m\_guard(m\_mutex);**
18. std::chrono::milliseconds dura(2000);   //1秒 = 1000毫秒，所以2000毫秒 = 2秒。
19. std::this\_thread::sleep\_for(dura);;     //休息一定的时长。
20. m\_msgRecvQueue.push\_back(i);
21. }
22. }
24. **bool** outMsgLULProc(**int** &command)
25. {
26. //lock\_guard<mutex> m\_guard(m\_mutex);
27. **unique\_lock<mutex> m\_uniqueLock(m\_mutex, try\_to\_lock);**
28. **if** (**m\_uniqueLock.owns\_lock()**)
29. {
30. //拿到了锁
31. cout << "拿到了锁！！！！！！！！！！！！！！！！！！！！！！！！！！！！！" << endl;
32. **if** (!m\_msgRecvQueue.empty())
33. {
34. command = m\_msgRecvQueue.front();       //返回第一个元素，但不检查元素是否存在
35. m\_msgRecvQueue.pop\_front();             //移除第一个元素，但不返回
36. **return** **true**;
37. }
38. **return** **false**;
39. }
40. **else**
41. {
42. //没有拿到锁
43. cout << "没有拿到锁，尝试做一些其他的！！！" << endl;
44. }
45. }
47. **void** outMsgRecvQueue()
48. {
49. **int** command = 0;
50. **for** (**int** i = 0; i < 100000; i++)
51. {
52. **bool** result = outMsgLULProc(command);
53. //消息不为空
54. **if** (result == **true**)
55. {
56. cout << "outMsgRecvQueue()执行，取出第一个元素" << command << endl;
57. }
58. **else**
59. {
60. //消息队列为空
61. cout << "outMsgRecvQueue()执行，但目前消息队列为空" << i << endl;
62. }
63. }
64. }
66. **private**:
67. list<**int**> m\_msgRecvQueue;
68. mutex m\_mutex;
69. };

72. **int** main()
73. {
74. A myToJob;
75. **thread** myInMsgJob(&A::inMesRecvQueue, &myToJob);
76. **thread** myOutMsgJob(&A::outMsgRecvQueue, &myToJob);
78. myInMsgJob.join();
79. myOutMsgJob.join();
81. **return** 0;
82. }

代码中利用

**unique\_lock<mutex> m\_uniqueLock(m\_mutex, try\_to\_lock);**

**m\_uniqueLock.owns\_lock();**

通过判断own\_lock()的值来确认m\_uniqueLock是否拿到了锁，若没有，则继续向下执行，不会阻塞。

###### std::defer\_lock：

用这个std::defer\_lock的前提是你自己不能先lock()，否则会报异常。defer\_lock的意思就是并没有给mutex加锁：初始化了一个没有加锁的mutex。

##### unique\_lock所有权的传递 mutex

std::unique\_lock<std::mutex> m\_unique\_lock(m\_mutex); //所有权的概念

m\_unique\_lock拥有m\_mutex的所有权

m\_unique\_lock可以把自己对mutex(m\_mutex)的所有权转移给其他的unique\_lock对象；

所以，unique\_lock对象这个mutex的所有权是属于可以转移，但是不能复制。

**方法：**

* 1. std::move
  2. return std::unique\_lock<std::mutex>

eg：

方法a:

1. **void** inMesRecvQueue()
2. {
3. **for** (**int** i = 0; i < 100000; i++)
4. {
5. cout << "inMesRecvQueue()执行，插入一个元素" << i << endl;
6. unique\_lock<mutex> m\_uniqueLock1(m\_mutex);
7. unique\_lock<mutex> m\_uniqueLock2(std::move(m\_uniqueLock1));
8. std::chrono::milliseconds dura(2000);   //1秒 = 1000毫秒，所以2000毫秒 = 2秒。
9. std::this\_thread::sleep\_for(dura);;     //休息一定的时长。
10. m\_msgRecvQueue.push\_back(i);
11. }
12. }

方法b:

1. std::unique\_lock<std::mutex> rtn\_unique\_lock()
2. {
3. std::unique\_lock<std::mutex> tmpguard(m\_mutex);
4. return tmpguard;            //从函数返回一个局部的unique\_lock对象是可以的。
5. //返回这种局部对象的tmpguard会导致系统生成临时unique\_lock对象，
6. //并调用unique\_lock的移动构造函数
7. }
9. /\*
10. ...
11. \*/
13. **std::unique\_lock<std::mutex> m\_unique\_lock = rtn\_unique\_lock();**

#### e) 死锁：

概念：

比如我又两把锁（死多这个问题 是由至少两把锁头也就是两个互斥量才能产生）；金锁（JinLock），银锁（YinLock），两个线程A、B。

1. 线程A执行的时候，这个线程线索金锁，把金锁lock() 成功了，然后它去lock()银锁，出现上下文切换。
2. 线程B执行了，这个线程先锁银锁，因为银锁还没有被锁，所以银锁会lock()成功，线程B要去lock()金锁。
3. 线程A因为拿不到银锁头，流程走不下去（所以后边代码有解锁金锁头的但是流程走不下去，所以金锁头解不开）。
4. 线程B因为拿不到金锁头，流程走不下去（所以后边代码有解锁银锁头的但是流程走不下去，所以银锁头解不开）。
5. 两个线程都晾在这里，你等我、我等你（卡死状态）；

eg：

1. #include <iostream>
2. #include <thread>
3. #include <list>
4. #include <mutex>      //线程锁的头文件
6. **using** **namespace** std;
8. **class** A
9. {
10. **public**:
11. //把收到的消息入到一个队列的线程
12. **void** inMesRecvQueue()
13. {
14. **for** (**int** i = 0; i < 100000; i++)
15. {
16. cout << "inMesRecvQueue()执行，插入一个元素" << i << endl;
17. **m\_mutex1.lock();**
18. **m\_mutex2.lock();**
19. m\_msgRecvQueue.push\_back(i);
20. **m\_mutex2.unlock();**
21. **m\_mutex1.unlock();**
22. }
23. }
25. **bool** outMsgLULProc(**int** &command)
26. {
27. **m\_mutex2.lock();**
28. **m\_mutex1.lock();**
29. **if** (!m\_msgRecvQueue.empty())
30. {
31. command = m\_msgRecvQueue.front();       //返回第一个元素，但不检查元素是否存在
32. m\_msgRecvQueue.pop\_front();             //移除第一个元素，但不返回
33. **m\_mutex1.unlock();**
34. **m\_mutex2.unlock();**
35. **return** **true**;
36. }
37. **m\_mutex1.unlock();**
38. **m\_mutex2.unlock();**
39. **return** **false**;
40. }
42. **void** outMsgRecvQueue()
43. {
44. **int** command = 0;
45. **for** (**int** i = 0; i < 100000; i++)
46. {
47. **bool** result = outMsgLULProc(command);
48. //消息不为空
49. **if** (result == **true**)
50. {
51. cout << "outMsgRecvQueue()执行，取出第一个元素" << command << endl;
52. }
53. **else**
54. {
55. //消息队列为空
56. cout << "outMsgRecvQueue()执行，但目前消息队列为空" << i << endl;
57. }
58. }
59. }
61. **private**:
62. list<**int**> m\_msgRecvQueue;
63. mutex m\_mutex1;
64. mutex m\_mutex2;
65. };

68. **int** main()
69. {
70. A myToJob;
71. **thread** myInMsgJob(&A::inMesRecvQueue, &myToJob);
72. **thread** myOutMsgJob(&A::outMsgRecvQueue, &myToJob);
74. myInMsgJob.join();
75. myOutMsgJob.join();
77. **return** 0;
78. }

此时，当程序运行时，便会出现死锁。

#### f) 死锁的一般解决方案：

只要保证这两个互斥量上锁的顺序一致就不会死锁。

#### g) std::lock() 函数模板：

功能：以此锁住两个或者多个以上的互斥量（至少两个，多了不限，1个不行）；它不存在这种因为在多线程中 因为锁的顺序问题导致死锁的风险问题。

std::lock()：如果互斥量中有一个没锁住，它就在那里等着，等所有互斥量都锁住，它才能往下走（返回）；要么两个互斥量都锁住，要么两个互斥量都没锁住。如果只锁了一个，另一个没有锁成功，则它立即将已经锁住的解锁。

eg：

1. **bool** outMsgLULProc(**int** &command)
2. {
3. **lock(m\_mutex1, m\_mutex2);**
4. **if** (!m\_msgRecvQueue.empty())
5. {
6. command = m\_msgRecvQueue.front();       //返回第一个元素，但不检查元素是否存在
7. m\_msgRecvQueue.pop\_front();             //移除第一个元素，但不返回
8. **m\_mutex1.unlock();**
9. **m\_mutex2.unlock();**
10. **return** **true**;
11. }
12. **m\_mutex1.unlock();**
13. **m\_mutex2.unlock();**
14. **return** **false**;
15. }

#### h) std::lock\_guard() 的std::adopt\_lock参数：

1. **bool** outMsgLULProc(**int** &command)
2. {
3. **lock(m\_mutex1, m\_mutex2);**
4. **lock\_guard<mutex> m\_guard1(m\_mutex1, adopt\_lock);**
5. **lock\_guard<mutex> m\_guard2(m\_mutex2, adopt\_lock);**
6. **if** (!m\_msgRecvQueue.empty())
7. {
8. command = m\_msgRecvQueue.front();       //返回第一个元素，但不检查元素是否存在
9. m\_msgRecvQueue.pop\_front();             //移除第一个元素，但不返回
10. **return** **true**;
11. }
12. **return** **false**;
13. }

这样联合使用可以不再去单独解锁。默认情况下使用lock\_guard的话，其构造函数会自动lock()，当使用std::adopt\_lock这个参数时，便会实现在构造函数时不再lock()，而是直接使用该行代码之前的lock；

## 3、详解多线程程序

### （1）关于thread里的坑

使用thread类中的detach函数，会存在一些问题（若创建子线程时用到了主线程中的一些变量，主线程结束，子线程继续访问这些变量就会出现一些不可预料的错误）。关于测试这些bug的方法，重写构造函数、拷贝函数、析构函数、并打印主、子线程的ID，以此来测试。

eg：

1. #include <iostream>
2. #include <thread>
3. **using** **namespace** std;
5. **class** A
6. {
7. **public**:
8. **int** m\_i;
9. //类型转换的构造函数，可以把一个int转换成一个类A对象。
10. A(**int** a) : m\_i(a) {cout << "[A::A(int a)构造函数执行]" << endl;}
11. A(**const** A &a) : m\_i(a.m\_i) {cout << "[A::A(const A)拷贝构造函数执行" << endl;}
12. ~A() {cout << "[A::~A()析构函数执行" << endl;}
13. }
15. **void** myPrint(**const** **int** i, **const** A &pmybuff)
16. {
17. cout << &pmybuff << endl;
18. **return**;
19. }
21. **int** main()
22. {
23. **int** mvar = 1;
24. **int** mysecondpar = 12;
25. **thread** myToJob(myPrint, mvar, mysecondpar);
27. myToJob.join();         //使用该函数不会存在问题
28. myToJob.detach();       //使用该函数会存在主线程已经结束，
29. //但是线程还未开始（未触发类A的构造函数）。
30. **return** 0;
31. }

运行结果可以看出，类A的构造函数并没有调用，因此该函数存在bug，解决的办法可以为：

将25行的 **thread** myToJob(myPrint, mvar, mysecondpar);

修改为 **thread** myToJob(myPrint, mvar, A(mysecondpar));

**即在创建线程的同时构造临时对象的方法传递参数**

#### 总结（针对使用detach() 而言，join() 不存在这些问题）：

a) 若传递int这种简单类型参数，建议都是值传递，不要用引用。防止节外生枝。

b) 如果传递类对象，避免隐式类型转换。全部都在创建线程这一行就构建出临时对象来，然后在函数参数里，用引用来接，否则系统还会多构造以此对象，浪费资源。

c) 建议不使用detach()，只使用join(),这样就不存在局部变量失效导致线程对内存的非法引用问题。

### （2）线程ID

线程ID的概念：ID是个数字，每个线程（不管是主线程还是子线程）实际上都对应着一个数字，而且每个线程对应的这个数字都不同。线程ID可以用C++标准库里的函数来获取：std::this\_thread::get\_id();来获取。

# 扩充

## windows中的延时函数：

std::chrono::milliseconds dura(2000); //1秒 = 1000毫秒，所以20000毫秒 = 20秒。

std::this\_thread::sleep\_for(dura);; //休息一定的时长。