C++实现线程池。

                     代码地址：<https://github.com/ithzhang/ThreadpoolLib.git>

本文介绍的线程池采用C++语言，在windows平台下实现。此版本为Version 1.0，以后还会推出功能更完备的后续版本。本着技术分享的精神写作本文同时公布源代码。欢迎大家指出该线程池存在的问题并对当前性能进行讨论。

**适用场景：**

   1.需要大量的线程来完成任务，且完成任务的时间比较短。

   2.对性能要求苛刻的应用，比如要求服务器迅速相应客户请求。

   3.接受突发性的大量请求，但不至于使服务器因此产生大量线程的应用。

**不适合在以下场景下使用：**

   1.可能会长时间运行的任务。

   2.具有良好的优先级控制。（本线程池仅仅实现了简单的优先级控制，有两种优先级：普通级和高级）。

**使用到的数据结构：**

**任务队列**：任务缓冲区，用于存储要执行任务的队列。可以调用线程池成员函数向该队列中增加任务。

**空闲线程堆栈**：用于存储空闲线程。空闲线程堆栈中会被压入指定数量的线程类对象指针。线程对象个数等于创建线程时初始线程个数。

**活动线程链表**：用以存储当前正在执行任务的线程。当有任务到来时，线程会从空闲堆栈转移到活动链表中。任务完成，且任务队列中没有任务时，会从活动链表转移到空闲堆栈中。本文中我称其为线程状态转换。

**调度机制：**

   1.向任务队列添加任务后，会检查此时空闲线程堆栈中是否有空闲线程，如有则从任务队列队首取出任务执行。

   2.当线程执行完当前任务，准备转移到空闲堆栈时，也会检查当前任务队列是否为空。若不为空，则继续取出任务执行。否则，转换到空闲线程堆栈。

除上述两种调度机制外，没有采用其他机制。

在创建线程池时会指定一个初始线程个数。此处我采取的是：一次性创建用户指定的线程，并加入到空闲线程堆栈。以后这个数量无法更改，且不会随着任务的多寡而增添或减少。

所有处于空闲队列中的线程都由于等待事件对象触发而处于阻塞态。等待事件对象成功的线程会进入到活动线程链表中。

**使用到的类：**

**CTask类**：任务基类。每个任务应继承自此类，并实现taskProc成员函数。

**CMyThread类**：工作线程类。每个类管理一个线程。同时关联一个任务类对象。

**CThreadPool类**：线程池类，用以创建并管理线程池，同时实现对线程池内线程的调度。

**CMyStack类**：空闲线程堆栈，用以存储空闲的工作线程。

**CMyList类**：活动线程队列。用以存储目前正在执行任务的线程。

**CTaskQueue类**：任务队列。用以存储要执行的任务。

**CMyMutex类**：互斥类。用于实现线程互斥访问。CMyStack，CMyList和CMyQueue内部都使用了CMyMutex类。它们是线程安全的。

MyThread类和CThreadPool类为核心类。其余为辅助类。

**CTask类**

   CTask是任务基类，所以非常简单，仅仅提供接口。

其声明如下：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ithzhang/article/details/9020283?reload)

1. **class** CTask
2. {
3. **public**:
4. CTask(int id);
5. ~CTask(**void**);
6. **public**:
7. **virtual** **void** taskProc()=0;
8. bool getID();
9. **private**:
10. int m\_ID;
11. };

具体的任务类应继承自此基类，并实现taskProc函数。在该函数实现需要线程池执行的任务。

如：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ithzhang/article/details/9020283?reload)

1. //TestTask.h
2. #include "task.h"
3. **class** CTestTask :
4. **public** CTask
5. {
6. **public**:
7. CTestTask(int id);
8. ~CTestTask(**void**);
9. **public**:
10. **virtual** **void** taskProc();
11. };
12. //TestTask.cpp
13. #include "TestTask.h"
14. CTestTask::CTestTask(int id)
15. :CTask(id)
16. {
17. }
18. CTestTask::~CTestTask(**void**)
19. {
20. }
21. **void** CTestTask::taskProc()
22. {
23. //模拟任务。
24. **for**(int i=0;i<10000;i++)
25. {
26. **for**(int j=0;j<10000;j++)
27. {
28. int temp=1;
29. temp++;
30. }
31. }
32. }

**CMyStack空闲线程堆栈类**

CMyStack类用以存储空闲线程。内部采用stack实现。之所以采用栈来存储线程类对象，是因为：当一个线程执行完任务后，如果此时任务队列没有新任务，该线程就被压入到空闲线程栈。此后当有新任务到来时，栈顶元素，也就是刚刚被压入的线程会被弹出执行新任务。由于该线程是最近才被压入，其对应内存空间位于内存中的概率比其他线程的概率要大。这在一定程度上可以节省从系统页交换文件交换到物理内存的开销。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ithzhang/article/details/9020283?reload)

1. //MyStack.h
2. #pragma once
3. #include<stack>
4. #include "MyMutex.h"
5. **class** CMyThread ;
6. **class** CMyStack
7. {
8. **public**:
9. CMyStack(**void**);
10. ~CMyStack(**void**);
11. **public**:
12. CMyThread\* pop();
13. bool push(CMyThread\*);
14. int getSize();
15. bool isEmpty();
16. bool clear();
17. **private**:
18. std::stack<CMyThread\*> m\_stack;
19. CMyMutex m\_mutext;
20. };

**CMyList活动线程链表类**

CMyList类用以存储正在执行任务的线程。内部采用list实现。活动线程在执行完任务后，可以被随时从活动链表中删除。之所以使用链表是因为在链表中删除某一元素的开销很小。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ithzhang/article/details/9020283?reload)

1. //MyList.h
2. #pragma once
3. #include <list>
4. #include "MyMutex.h"
5. **class** CMyThread;
6. **class** CMyList
7. {
8. **public**:
9. CMyList(**void**);
10. ~CMyList(**void**);
11. **public**:
12. bool addThread(CMyThread\*t);
13. bool removeThread(CMyThread\*t);
14. int getSize();
15. bool isEmpty();
16. bool clear();
17. **private**:
18. std::list<CMyThread\*>m\_list;
19. CMyMutex m\_mutex;
20. };

**CMyQueue任务队列类**

CMyQueue用以存储要执行的任务。内部采用双向队列实现。具有简单的优先级控制机制。当普通的优先级任务到来时，会正常入队。当高优先级任务到来时会插入到对首。线程池在调度时会简单的从队首取出任务并执行。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ithzhang/article/details/9020283?reload)

1. //MyQueue.h
2. #pragma once
3. #include<deque>
4. #include"MyMutex.h"
5. **class** CTask;
6. **class** CMyQueue
7. {
8. **public**:
9. CMyQueue(**void**);
10. ~CMyQueue(**void**);
11. **public**:
12. CTask\*pop();
13. bool push(CTask\*t);
14. bool pushFront(CTask\*t);、
15. bool isEmpty();
16. bool clear();
17. **private**:
18. std::deque<CTask\*>m\_TaskQueue;
19. CMyMutex m\_mutex;
20. };

**CMyMutex互斥类**

CMyMutex类用于控制线程互斥访问。内部采用CRITICAL\_SECTION实现 。在对活动线程链表、空闲线程堆栈、任务队列进行访问时都需要进行互斥访问控制。防止多线程同时访问导致的状态不一致的情况出现。

类声明如下：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ithzhang/article/details/9020283?reload)

1. //MyMutex.h
2. #pragma once
3. #include "windows.h"
4. **class** CMyMutex
5. {
6. **public**:
7. CMyMutex(**void**);
8. ~CMyMutex(**void**);
9. **public**:
10. bool Lock();
11. bool Unlock();
12. **private**:
13. CRITICAL\_SECTION m\_cs;
14. };

**CMyThread工作线程类**

CMyThread类用于管理一个线程。该类内部有一个CTask\*成员和一个事件对象。CTask\*成员为与该线程关联的任务。调用assignTask可以为该线程设置对应的任务。

类声明如下：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ithzhang/article/details/9020283?reload)

1. //MyThread.h
2. #pragma once
3. #include "windows.h"
4. **class** CTask;
5. **class** CBaseThreadPool;
6. **class** CMyThread
7. {
8. **public**:
9. CMyThread(CBaseThreadPool\*threadPool);
10. ~CMyThread(**void**);
11. **public**:
12. bool startThread();
13. bool suspendThread();
14. bool resumeThread();
15. bool assignTask(CTask\*pTask);
16. bool startTask();
17. **static** DWORD WINAPI threadProc(LPVOID pParam);
18. DWORD m\_threadID;
19. HANDLE m\_hThread;
20. **private**:
21. HANDLE m\_hEvent;
22. CTask\*m\_pTask;
23. CBaseThreadPool\*m\_pThreadPool;
24. };

startThread用于创建入口函数为threadProc的线程。在该线程内部会循环等待一个事件对象。当没有任务到来时，线程就会在该事件对象上挂起。当新任务到来，线程池会将该线程对应的事件对象触发，然后执行其对应的任务。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ithzhang/article/details/9020283?reload)

1. DWORD WINAPI CMyThread::threadProc( LPVOID pParam )
2. {
3. CMyThread \*pThread=(CMyThread\*)pParam;
4. **while**(!pThread->m\_bIsExit)
5. {
6. DWORD ret=WaitForSingleObject(pThread->m\_hEvent,INFINITE);
7. **if**(ret==WAIT\_OBJECT\_0)
8. {
9. **if**(pThread->m\_pTask)
10. {
11. pThread->m\_pTask->taskProc();、
12. **delete** pThread->m\_pTask;
13. pThread->m\_pTask=NULL;
14. pThread->m\_pThreadPool->SwitchActiveThread(pThread);
15. }
16. }
17. }
18. **return** 0;
19. }

当任务执行完之后，线程内部会调用线程池的SwitchActiveThread成员函数，该函数用以将线程从活动状态转变为空闲态。也就是从活动线程链表转移到空闲线程栈中。同时线程继续等待事件对象触发。

在此函数内部，在转换之前会检查任务队列中是否还有任务，如果有任务，线程会继续从任务队列取出任务继续执行，而不会切换到空闲态。直到任务队列中没有任务时才会执行状态切换操作。

**CMyThreadPool线程池类**

任务队列、活动线程链表、空闲线程队列都作为线程池的成员变量，由线程池维护。

类声明如下：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ithzhang/article/details/9020283?reload)

1. //MyThreadPool.h
2. #pragma once
3. #include<list>
4. #include "MyMutex.h"
5. #include "MyStack.h"
6. #include "MyList.h"
7. #include"MyQueue.h"
8. **class** CMyThread;
9. **class** CTask;
10. **enum** PRIORITY
11. {
12. NORMAL,
13. HIGH
14. };
15. **class** CBaseThreadPool
16. {
17. **public**:
18. **virtual** CMyThread\* PopIdleThread()=0;
19. **virtual** CTask\*GetNewTask()=0;
20. //virtual bool  ExecuteNewTask(CTask \*task)=0;
21. **virtual** bool SwitchActiveThread(CMyThread\*)=0;
22. };
23. **class** CMyThreadPool:**public** CBaseThreadPool
24. {
25. **public**:
26. CMyThreadPool(int num);
27. ~CMyThreadPool(**void**);
28. **public**:
29. **virtual** CMyThread\* PopIdleThread();
30. **virtual** bool SwitchActiveThread(CMyThread\*);
31. **virtual** CTask\*GetNewTask();
32. **public**:
33. //priority为优先级。高优先级的任务将被插入到队首。
34. bool addTask(CTask\*t,PRIORITY priority);
35. bool start();//开始调度。
36. bool destroyThreadPool();
37. **private**:
38. int m\_nThreadNum;
39. bool m\_bIsExit;
41. CMyStack m\_IdleThreadStack;
42. CMyList m\_ActiveThreadList;
43. CMyQueue m\_TaskQueue;
44. };

addTask函数用于向任务队列中添加任务。添加任务后，会检查空闲线程堆栈中是否为空，如不为空则弹出栈顶线程执行任务。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ithzhang/article/details/9020283?reload)

1. bool CMyThreadPool::addTask( CTask\*t,PRIORITY priority )
2. {
3. assert(t);
4. **if**(!t||m\_bIsExit)
5. **return** **false**;
6. CTask \*task=NULL;
7. **if**(priority==PRIORITY::NORMAL)
8. {
9. m\_TaskQueue.push(t);//压入任务队列尾部。
10. }
11. **else** **if**(PRIORITY::HIGH)
12. {
13. m\_TaskQueue.pushFront(t);//高优先级任务，压到队首。
14. }
15. **if**(!m\_IdleThreadStack.isEmpty())//存在空闲线程。调用空闲线程处理任务。
16. {
17. task=m\_TaskQueue.pop();//取出列头任务。
18. **if**(task==NULL)
19. {
20. //std::cout<<"任务取出出错。"<<std::endl;
21. **return** 0;
22. }
23. CMyThread\*pThread=PopIdleThread();
24. m\_ActiveThreadList.addThread(pThread);//加入到活动链表。
25. pThread->assignTask(task);//将任务与线程关联。
26. pThread->startTask();//开始任务，内部对事件对象进行触发。
27. }
29. }

    switchActiveThread函数用以在线程结束任务之后，将自己切换到空闲态。在切换之前会检查任务队列是否有任务，如有任务，则取出继续执行。直到任务队列为空时，才将自己切换到空闲态。由各线程类对象调用。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ithzhang/article/details/9020283?reload)

1. bool CMyThreadPool::SwitchActiveThread( CMyThread\*t)
2. {
3. **if**(!m\_TaskQueue.isEmpty())//任务队列不为空，继续取任务执行。
4. {
5. CTask \*pTask=NULL;
6. pTask=m\_TaskQueue.pop();
7. t->assignTask(pTask);
8. t->startTask();
9. }
10. **else**//任务队列为空，该线程挂起。
11. {
12. m\_ActiveThreadList.removeThread(t);
13. m\_IdleThreadStack.push(t);
14. }
15. **return** **true**;
16. }

     代码地址：<https://github.com/ithzhang/ThreadpoolLib.git>