C++线程池原理及创建（转）

本文给出了一个通用的线程池框架，该框架将与线程执行相关的任务进行了高层次的抽象，使之与具体的执行任务无关。具有动态伸缩性，它能根据执行任务的轻重自动调整线程池中线程的数量。文章的最后，我们给出一个简单示例程序，通过该们会发现，通过该线程池框架执行多线程任务是多么的简单。

为什么需要线程池

目前的大多数网络服务器，包括Web服务器、Email服务器以及数据库服务器等都具有一个共同点，就是单位时间内必须的连接请求，但处理时间却相对较短。
传统多线程方案中我们采用的服务器模型则是一旦接受到请求之后，即创建一个新的线程，由该线程执行任务。任务执行完毕出，这就是是“即时创建，即时销毁”的策略。尽管与创建进程相比，创建线程的时间已经大大的缩短，但是如果提交给线程时间较短，而且执行次数极其频繁，那么服务器将处于不停的创建线程，销毁线程的状态。

我们将传统方案中的线程执行过程分为三个过程：T1、T2、T3。

T1：线程创建时间
T2：线程执行时间，包括线程的同步等时间
T3：线程销毁时间

那么我们可以看出，线程本身的开销所占的比例为(T1+T3) / (T1+T2+T3)。如果线程执行的时间很短的话，这比开销可能占右。如果任务执行时间很频繁的话，这笔开销将是不可忽略的。

除此之外，线程池能够减少创建的线程个数。通常线程池所允许的并发线程是有上界的，如果同时需要并发的线程数超过部分线程将会等待。而传统方案中，如果同时请求数目为2000，那么最坏情况下，系统可能需要产生2000个线程。尽管这不数目，但是也有部分机器可能达不到这种要求。

因此线程池的出现正是着眼于减少线程池本身带来的开销。线程池采用预创建的技术，在应用程序启动之后，将立即创建程(N1)，放入空闲队列中。这些线程都是处于阻塞（Suspended）状态，不消耗CPU，但占用较小的内存空间。当任务到来择一个空闲线程，把任务传入此线程中运行。当N1个线程都在处理任务后，缓冲池自动创建一定数量的新线程，用于处理更任务执行完毕后线程也不退出，而是继续保持在池中等待下一次的任务。当系统比较空闲时，大部分线程都一直处于暂停状态销毁一部分线程，回收系统资源。

基于这种预创建技术，线程池将线程创建和销毁本身所带来的开销分摊到了各个具体的任务上，执行次数越多，每个任务程本身开销则越小，不过我们另外可能需要考虑进去线程之间同步所带来的开销。

构建线程池框架

一般线程池都必须具备下面几个组成部分：
线程池管理器:用于创建并管理线程池
工作线程: 线程池中实际执行的线程
任务接口: 尽管线程池大多数情况下是用来支持网络服务器，但是我们将线程执行的任务抽象出来，形成任务接口，从而是的的任务无关。
任务队列:线程池的概念具体到实现则可能是队列，链表之类的数据结构，其中保存执行线程。

我们实现的通用线程池框架由五个重要部分组成CThreadManage，CThreadPool，CThread，CJob，CWorkerThread，除还包括线程同步使用的类CThreadMutex和CCondition。
CJob是所有的任务的基类，其提供一个接口Run，所有的任务类都必须从该类继承，同时实现Run方法。该方法中实现具体的CThread是Linux中线程的包装，其封装了Linux线程最经常使用的属性和方法，它也是一个抽象类，是所有线程类的基类，其Run。
CWorkerThread是实际被调度和执行的线程类，其从CThread继承而来，实现了CThread中的Run方法。
CThreadPool是线程池类，其负责保存线程，释放线程以及调度线程。
CThreadManage是线程池与用户的直接接口，其屏蔽了内部的具体实现。
CThreadMutex用于线程之间的互斥。
CCondition则是条件变量的封装，用于线程之间的同步。

线程池的时序很简单。CThreadManage直接跟客户端打交道，其接受需要创建的线程初始个数，并接受客户端提交的任务。具体的非抽象的任务。CThreadManage的内部实际上调用的都是CThreadPool的相关操作。CThreadPool创建具体的线程交的任务分发给CWorkerThread，CWorkerThread实际执行具体的任务。

理解系统组件

文章档案(36)
2016年12月 (1)
2016年11月 (1)
2016年9月 (3)
2016年7月 (1)
2016年6月 (2)
2016年1月 (1)
2015年11月 (2)
2015年10月 (3)
2015年9月 (11)
2015年8月 (1)
2015年7月 (1)
2015年6月 (3)
2015年5月 (1)
2015年1月 (2)
2014年12月 (1)
2014年4月 (2)
最新评论
1. Re:据磁力链获得BT种子
@任意球import bencodefrom hashlib import sha1with open ("9f9165d9a281a9b8e782cd5176bbcc8256fd1871.torren.....
2. Re:据磁力链获得BT种子
@任意球你可以自己解析torrent 文件，用bencode对种子进行decode，获得info字段，对该字段的数据进行一次encode，在进行一次sha1就是磁链的info_hash...
3. Re:据磁力链获得BT种子
BT种子转磁链有方法嘛LZ
4. Re:据磁力链获得BT种子
@XiaoKaixuanvuze是有概率成功，不知道是不是因为有限制还是天天抽风...
5. Re:据磁力链获得BT种子
@XiaoKaixuan刚又点击了下你博文上的3个连接，n0808,bitcomet,用浏览器打开，无法下载。vuze倒是可以下载...
阅读排行榜
1. 据磁力链获得BT种子(16284)
2. Windows下的 Axel下载工具 - 移植自Linux(2013)
3. 不注册COM组件直接调用接口(1853)
4. 迅雷Vip账号共享器（持续更新）(865)
5. 基于Win32 SDK实现的一个简易线程池(774)
评论排行榜
1. 据磁力链获得BT种子(22)
2. 迅雷Vip账号共享器（持续更新）(5)
3. 不注册COM组件直接调用接口(3)
4. 编程实现Windows瞬间关机(1)
推荐排行榜
1. 据磁力链获得BT种子(2)
2. 不注册COM组件直接调用接口(2)
3. 编程实现Windows瞬间关机(1)
4. 基于Win32 SDK实现的一个简易线程池(1)
5. 万能图标提取器2013(1)

下面我们分开来了解系统中的各个组件。

CThreadManage

CThreadManage的功能非常简单，其提供最简单的方法，其类定义如下：

```
class CThreadManage
{
private:
    CThreadPool*    m_Pool;
    int           m_NumOfThread;
protected:
public:
    void     SetParallelNum(int num);
    CThreadManage();
    CThreadManage(int num);
    virtual ~CThreadManage();

    void    Run(CJob* job,void* jobdata);
    void    TerminateAll(void);
};
```

其中m_Pool指向实际的线程池；m_NumOfThread是初始创建时候允许创建的并发的线程个数。另外Run和TerminateAll方单，只是简单的调用CThreadPool的一些相关方法而已。其具体的实现如下:

```
CThreadManage::CThreadManage(){
    m_NumOfThread = 10;
    m_Pool = new CThreadPool(m_NumOfThread);
}
CThreadManage::CThreadManage(int num){
    m_NumOfThread = num;
    m_Pool = new CThreadPool(m_NumOfThread);
}
CThreadManage::~CThreadManage(){
    if(NULL != m_Pool)
    delete m_Pool;
}
void CThreadManage::SetParallelNum(int num){
    m_NumOfThread = num;
}
void CThreadManage::Run(CJob* job,void* jobdata){
    m_Pool->Run(job,jobdata);
}
void CThreadManage::TerminateAll(void){
    m_Pool->TerminateAll();
}
```

CThread

CThread 类实现了对Linux中线程操作的封装，它是所有线程的基类，也是一个抽象类，提供了一个抽象接口Run，所有的C实现该Run方法。CThread的定义如下所示：

```
class CThread
{
private:
    int           m_ErrCode;
    Semaphore     m_ThreadSemaphore;  //the inner semaphore, which is used to realize
    unsigned      long m_ThreadID;
    bool          m_Detach;       //The thread is detached
    bool          m_CreateSuspended;  //if suspend after creating
    char*         m_ThreadName;
    ThreadState m_ThreadState;      //the state of the thread
protected:
```

```
    void      SetErrcode(int errcode){m_ErrCode = errcode;}
    static void* ThreadFunction(void*);
public:
    CThread();
    CThread(bool createsuspended,bool detach);
    virtual ~CThread();
    virtual void Run(void) = 0;
    void      SetThreadState(ThreadState state){m_ThreadState = state;}

    bool      Terminate(void);    //Terminate the threa
    bool      Start(void);        //Start to execute the thread
    void      Exit(void);
    bool      Wakeup(void);

    ThreadState  GetThreadState(void){return m_ThreadState;}
    int       GetLastError(void){return m_ErrCode;}
    void      SetThreadName(char* thrname){strcpy(m_ThreadName,thrname);}
    char*     GetThreadName(void){return m_ThreadName;}
    int       GetThreadID(void){return m_ThreadID;}

    bool      SetPriority(int priority);
    int       GetPriority(void);
    int       GetConcurrency(void);
    void      SetConcurrency(int num);
    bool      Detach(void);
    bool      Join(void);
    bool      Yield(void);
    int       Self(void);
};
```

线程的状态可以分为四种，空闲、忙碌、挂起、终止(包括正常退出和非正常退出)。由于目前Linux线程库不支持挂起操作，[
处的挂起操作类似于暂停。如果线程创建后不想立即执行任务，那么我们可以将其“暂停”，如果需要运行，则唤醒。有一点
是，一旦线程开始执行任务，将不能被挂起，其将一直执行任务至完毕。
线程类的相关操作均十分简单。线程的执行入口是从Start()函数开始，其将调用函数ThreadFunction，ThreadFunction再调
函数，执行实际的任务。

CThreadPool

CThreadPool是线程的承载容器，一般可以将其实现为堆栈、单向队列或者双向队列。在我们的系统中我们使用STL Vector
存。CThreadPool的实现代码如下：

```
class CThreadPool
{
friend class CWorkerThread;
private:
    unsigned int m_MaxNum;   //the max thread num that can create at the same time
    unsigned int m_AvailLow; //The min num of idle thread that shoule kept
    unsigned int m_AvailHigh;    //The max num of idle thread that kept at the same time
    unsigned int m_AvailNum; //the normal thread num of idle num;
    unsigned int m_InitNum;  //Normal thread num;
protected:
    CWorkerThread* GetIdleThread(void);

    void    AppendToIdleList(CWorkerThread* jobthread);
    void    MoveToBusyList(CWorkerThread* idlethread);
    void    MoveToIdleList(CWorkerThread* busythread);

    void    DeleteIdleThread(int num);
    void    CreateIdleThread(int num);
public:
    CThreadMutex m_BusyMutex;    //when visit busy list,use m_BusyMutex to lock and unlock
    CThreadMutex m_IdleMutex;    //when visit idle list,use m_IdleMutex to lock and unlock
    CThreadMutex m_JobMutex; //when visit job list,use m_JobMutex to lock and unlock
    CThreadMutex m_VarMutex;
```

```
24
25      CCondition       m_BusyCond; //m_BusyCond is used to sync busy thread list
26      CCondition       m_IdleCond; //m_IdleCond is used to sync idle thread list
27      CCondition       m_IdleJobCond;  //m_JobCond is used to sync job list
28      CCondition       m_MaxNumCond;
29
30      vector<CWorkerThread*>   m_ThreadList;
31      vector<CWorkerThread*>   m_BusyList;     //Thread List
32      vector<CWorkerThread*>   m_IdleList; //Idle List
33
34      CThreadPool();
35      CThreadPool(int initnum);
36      virtual ~CThreadPool();
37
38      void    SetMaxNum(int maxnum){m_MaxNum = maxnum;}
39      int     GetMaxNum(void){return m_MaxNum;}
40      void    SetAvailLowNum(int minnum){m_AvailLow = minnum;}
41      int     GetAvailLowNum(void){return m_AvailLow;}
42      void    SetAvailHighNum(int highnum){m_AvailHigh = highnum;}
43      int     GetAvailHighNum(void){return m_AvailHigh;}
44      int     GetActualAvailNum(void){return m_AvailNum;}
45      int     GetAllNum(void){return m_ThreadList.size();}
46      int     GetBusyNum(void){return m_BusyList.size();}
47      void    SetInitNum(int initnum){m_InitNum = initnum;}
48      int     GetInitNum(void){return m_InitNum;}
49
50      void    TerminateAll(void);
51      void    Run(CJob* job,void* jobdata);
52  };
53  CThreadPool::CThreadPool()
54  {
55      m_MaxNum = 50;
56      m_AvailLow = 5;
57      m_InitNum=m_AvailNum = 10 ;
58      m_AvailHigh = 20;
59
60      m_BusyList.clear();
61      m_IdleList.clear();
62      for(int i=0;i<m_InitNum;i++){
63      CWorkerThread* thr = new CWorkerThread();
64      thr->SetThreadPool(this);
65      AppendToIdleList(thr);
66      thr->Start();
67      }
68  }
69
70  CThreadPool::CThreadPool(int initnum)
71  {
72      assert(initnum>0 && initnum<=30);
73      m_MaxNum   = 30;
74      m_AvailLow = initnum-10>0?initnum-10:3;
75      m_InitNum=m_AvailNum = initnum ;
76      m_AvailHigh = initnum+10;
77
78      m_BusyList.clear();
79      m_IdleList.clear();
80      for(int i=0;i<m_InitNum;i++){
81      CWorkerThread* thr = new CWorkerThread();
82      AppendToIdleList(thr);
83      thr->SetThreadPool(this);
84      thr->Start();       //begin the thread,the thread wait for job
85      }
86  }
87
88  CThreadPool::~CThreadPool()
```

```
89 {
90    TerminateAll();
91 }
92
93 void CThreadPool::TerminateAll()
94 {
95     for(int i=0;i < m_ThreadList.size();i++) {
96     CWorkerThread* thr = m_ThreadList[i];
97     thr->Join();
98     }
99     return;
100 }
101
102 CWorkerThread* CThreadPool::GetIdleThread(void)
103 {
104     while(m_IdleList.size() ==0 )
105     m_IdleCond.Wait();
106
107     m_IdleMutex.Lock();
108     if(m_IdleList.size() > 0 )
109     {
110     CWorkerThread* thr = (CWorkerThread*)m_IdleList.front();
111     printf("Get Idle thread %dn",thr->GetThreadID());
112     m_IdleMutex.Unlock();
113     return thr;
114     }
115     m_IdleMutex.Unlock();
116
117     return NULL;
118 }
119
120 //add an idle thread to idle list
121 void CThreadPool::AppendToIdleList(CWorkerThread* jobthread)
122 {
123     m_IdleMutex.Lock();
124     m_IdleList.push_back(jobthread);
125     m_ThreadList.push_back(jobthread);
126     m_IdleMutex.Unlock();
127 }
128
129 //move and idle thread to busy thread
130 void CThreadPool::MoveToBusyList(CWorkerThread* idlethread)
131 {
132     m_BusyMutex.Lock();
133     m_BusyList.push_back(idlethread);
134     m_AvailNum--;
135     m_BusyMutex.Unlock();
136
137     m_IdleMutex.Lock();
138     vector<CWorkerThread*>::iterator pos;
139     pos = find(m_IdleList.begin(),m_IdleList.end(),idlethread);
140     if(pos !=m_IdleList.end())
141     m_IdleList.erase(pos);
142     m_IdleMutex.Unlock();
143 }
144
145 void CThreadPool::MoveToIdleList(CWorkerThread* busythread)
146 {
147     m_IdleMutex.Lock();
148     m_IdleList.push_back(busythread);
149     m_AvailNum++;
150     m_IdleMutex.Unlock();
151
152     m_BusyMutex.Lock();
153     vector<CWorkerThread*>::iterator pos;
```

```
154     pos = find(m_BusyList.begin(),m_BusyList.end(),busythread);
155     if(pos!=m_BusyList.end())
156     m_BusyList.erase(pos);
157     m_BusyMutex.Unlock();
158 
159     m_IdleCond.Signal();
160     m_MaxNumCond.Signal();
161 }
162 
163 //create num idle thread and put them to idlelist
164 void CThreadPool::CreateIdleThread(int num)
165 {
166     for(int i=0;i<num;i++){
167     CWorkerThread* thr = new CWorkerThread();
168     thr->SetThreadPool(this);
169     AppendToIdleList(thr);
170     m_VarMutex.Lock();
171     m_AvailNum++;
172     m_VarMutex.Unlock();
173     thr->Start();       //begin the thread,the thread wait for job
174     }
175 }
176 
177 void CThreadPool::DeleteIdleThread(int num)
178 {
179     printf("Enter into CThreadPool::DeleteIdleThreadn");
180     m_IdleMutex.Lock();
181     printf("Delete Num is %dn",num);
182     for(int i=0;i<num;i++){
183     CWorkerThread* thr;
184     if(m_IdleList.size() > 0 ){
185             thr = (CWorkerThread*)m_IdleList.front();
186             printf("Get Idle thread %dn",thr->GetThreadID());
187     }
188 
189     vector<CWorkerThread*>::iterator pos;
190     pos = find(m_IdleList.begin(),m_IdleList.end(),thr);
191     if(pos!=m_IdleList.end())
192         m_IdleList.erase(pos);
193     m_AvailNum--;
194     printf("The idle thread available num:%d n",m_AvailNum);
195     printf("The idlelist              num:%d n",m_IdleList.size());
196     }
197     m_IdleMutex.Unlock();
198 }
199 void CThreadPool::Run(CJob* job,void* jobdata)
200 {
201     assert(job!=NULL);
202 
203     //if the busy thread num adds to m_MaxNum,so we should wait
204     if(GetBusyNum() == m_MaxNum)
205         m_MaxNumCond.Wait();
206 
207     if(m_IdleList.size()<m_AvailLow)
208     {
209     if(GetAllNum()+m_InitNum-m_IdleList.size() < m_MaxNum )
210         CreateIdleThread(m_InitNum-m_IdleList.size());
211     else
212         CreateIdleThread(m_MaxNum-GetAllNum());
213     }
214 
215     CWorkerThread*  idlethr = GetIdleThread();
216     if(idlethr !=NULL)
217     {
218     idlethr->m_WorkMutex.Lock();
```

```
219     MoveToBusyList(idlethr);
220     idlethr->SetThreadPool(this);
221     job->SetWorkThread(idlethr);
222     printf("Job is set to thread %d n",idlethr->GetThreadID());
223     idlethr->SetJob(job,jobdata);
224     }
225 }
```

在CThreadPool中存在两个链表，一个是空闲链表，一个是忙碌链表。Idle链表中存放所有的空闲进程，当线程执行任务时候
忙碌状态，同时从空闲链表中删除，并移至忙碌链表中。在CThreadPool的构造函数中，我们将执行下面的代码:

```
1 for(int i=0;i<m_InitNum;i++)
2     {
3     CWorkerThread* thr = new CWorkerThread();
4     AppendToIdleList(thr);
5     thr->SetThreadPool(this);
6     thr->Start();       //begin the thread,the thread wait for job
7     }
```

在该代码中，我们将创建m_InitNum个线程，创建之后即调用AppendToIdleList放入Idle链表中，由于目前没有任务分发给
此线程执行Start后将自己挂起。
事实上，线程池中容纳的线程数目并不是一成不变的，其会根据执行负载进行自动伸缩。为此在CThreadPool中设定四个变
m_InitNum：处世创建时线程池中的线程的个数。
m_MaxNum:当前线程池中所允许并发存在的线程的最大数目。
m_AvailLow:当前线程池中所允许存在的空闲线程的最小数目，如果空闲数目低于该值，表明负载可能过重，此时有必要增加
数目。实现中我们总是将线程调整为m_InitNum个。
m_AvailHigh：当前线程池中所允许的空闲的线程的最大数目，如果空闲数目高于该值，表明当前负载可能较轻，此时将删除
程，删除后调整数也为m_InitNum个。
m_AvailNum：目前线程池中实际存在的线程的个数，其值介于m_AvailHigh和m_AvailLow之间。如果线程的个数始终维持
和m_AvailHigh之间，则线程既不需要创建，也不需要删除，保持平衡状态。因此如何设定m_AvailLow和m_AvailHigh的值
最大可能的保持平衡态，是线程池设计必须考虑的问题。
线程池在接受到新的任务之后，线程池首先要检查是否有足够的空闲池可用。检查分为三个步骤：
(1)检查当前处于忙碌状态的线程是否达到了设定的最大值m_MaxNum，如果达到了，表明目前没有空闲线程可用，而且
的线程，因此必须等待直到有线程执行完毕返回到空闲队列中。
(2)如果当前的空闲线程数目小于我们设定的最小的空闲数目m_AvailLow，则我们必须创建新的线程，默认情况下，创建
应该为m_InitNum，因此创建的线程数目应该为(当前空闲线程数与m_InitNum);但是有一种特殊情况必须考虑，就是现有的
创建后的线程数可能超过m_MaxNum，因此我们必须对线程的创建区别对待。

```
1 if(GetAllNum()+m_InitNum-m_IdleList.size() < m_MaxNum )
2         CreateIdleThread(m_InitNum-m_IdleList.size());
3     else
4         CreateIdleThread(m_MaxNum-GetAllNum());
```

如果创建后总数不超过m_MaxNum，则创建后的线程为m_InitNum；如果超过了，则只创建(m_MaxNum-当前线程总数)
(3)调用GetIdleThread方法查找空闲线程。如果当前没有空闲线程，则挂起；否则将任务指派给该线程，同时将其移入忙
当线程执行完毕后，其会调用MoveToIdleList方法移入空闲链表中，其中还调用m_IdleCond.Signal()方法，唤醒GetIdleTh
塞的线程。

CWorkerThread
CWorkerThread是CThread的派生类，是事实上的工作线程。在CThreadPool的构造函数中，我们创建了一定数量的CWor
旦这些线程创建完毕，我们将调用Start()启动该线程。Start方法最终会调用Run方法。Run方法是个无限循环的过程。在没有
任务的时候，m_Job为NULL，此时线程将调用Wait方法进行等待，从而处于挂起状态。一旦线程池将具体的任务分发给该线
醒，从而通知线程从挂起的地方继续执行。CWorkerThread的完整定义如下:

```
1 class CWorkerThread:public CThread
2 {
3 private:
4     CThreadPool*  m_ThreadPool;
5     CJob*     m_Job;
6     void*     m_JobData;
7
8     CThreadMutex m_VarMutex;
```

```
 9     bool      m_IsEnd;
10 protected:
11 public:
12     CCondition   m_JobCond;
13     CThreadMutex m_WorkMutex;
14     CWorkerThread();
15     virtual ~CWorkerThread();
16     void Run();
17     void    SetJob(CJob* job,void* jobdata);
18     CJob*   GetJob(void){return m_Job;}
19     void    SetThreadPool(CThreadPool* thrpool);
20     CThreadPool* GetThreadPool(void){return m_ThreadPool;}
21 };
22 CWorkerThread::CWorkerThread()
23 {
24     m_Job = NULL;
25     m_JobData = NULL;
26     m_ThreadPool = NULL;
27     m_IsEnd = false;
28 }
29 CWorkerThread::~CWorkerThread()
30 {
31     if(NULL != m_Job)
32     delete m_Job;
33     if(m_ThreadPool != NULL)
34     delete m_ThreadPool;
35 }
36
37 void CWorkerThread::Run()
38 {
39     SetThreadState(THREAD_RUNNING);
40     for(;;)
41     {
42     while(m_Job == NULL)
43         m_JobCond.Wait();
44
45     m_Job->Run(m_JobData);
46     m_Job->SetWorkThread(NULL);
47     m_Job = NULL;
48     m_ThreadPool->MoveToIdleList(this);
49     if(m_ThreadPool->m_IdleList.size() > m_ThreadPool->GetAvailHighNum())
50     {
51 m_ThreadPool->DeleteIdleThread(m_ThreadPool->m_IdleList.size()-m_T
52 hreadPool->GetInitNum());
53     }
54     m_WorkMutex.Unlock();
55     }
56 }
57 void CWorkerThread::SetJob(CJob* job,void* jobdata)
58 {
59     m_VarMutex.Lock();
60     m_Job = job;
61     m_JobData = jobdata;
62     job->SetWorkThread(this);
63     m_VarMutex.Unlock();
64     m_JobCond.Signal();
65 }
66 void CWorkerThread::SetThreadPool(CThreadPool* thrpool)
67 {
68     m_VarMutex.Lock();
69     m_ThreadPool = thrpool;
70     m_VarMutex.Unlock();
71 }
```

当线程执行任务之前首先必须判断空闲线程的数目是否低于m_AvailLow，如果低于，则必须创建足够的空闲线程，使其m_InitNum个，然后将调用MoveToBusyList()移出空闲队列，移入忙碌队列。当任务执行完毕后，其又调用MoveToIdleLis列，移入空闲队列，等待新的任务。

除了Run方法之外，CWorkerThread中另外一个重要的方法就是SetJob，该方法将实际的任务赋值给线程。当没有任何m_Job为NULL的时候，线程将调用m_JobCond.Wait进行等待。一旦Job被赋值给线程，其将调用m_JobCond.Signal方法唤于m_JobCond属于线程内部的变量，每个线程都维持一个m_JobCond，只有得到任务的线程才被唤醒，没有得到任务的将续一个线程何时被唤醒，其都将从等待的地方继续执行m_Job->Run(m_JobData)，这是线程执行实际任务的地方。

在线程执行给定Job期间，我们必须防止另外一个Job又赋给该线程，因此在赋值之前，通过m_VarMutex进行锁定， Jo于的Job将不能关联到该线程；任务执行完毕，我们调用m_VarMutex.Unlock()进行解锁，此时，线程又可以接受新的执行任在线程执行任务结束后返回空闲队列前，我们还需要判断当前空闲队列中的线程是否高于m_AvailHigh个。如果超过m_Avai从其中删除(m_ThreadPool->m_IdleList.size()-m_ThreadPool->GetInitNum())个线程，使线程数目保持在m_InitNum个。

CJob

CJob类相对简单，其封装了任务的基本的属性和方法，其中最重要的是Run方法，代码如下：

```
class CJob
{
private:
    int      m_JobNo;        //The num was assigned to the job
    char*    m_JobName;      //The job name
    CThread  *m_pWorkThread;     //The thread associated with the job
public:
    CJob( void );
    virtual ~CJob();

    int      GetJobNo(void) const { return m_JobNo; }
    void     SetJobNo(int jobno){ m_JobNo = jobno;}
    char*    GetJobName(void) const { return m_JobName; }
    void     SetJobName(char* jobname);
    CThread *GetWorkThread(void){ return m_pWorkThread; }
    void     SetWorkThread ( CThread *pWorkThread ){
        m_pWorkThread = pWorkThread;
    }
    virtual void Run ( void *ptr ) = 0;
};
CJob::CJob(void)
:m_pWorkThread(NULL)
,m_JobNo(0)
,m_JobName(NULL)
{
}
CJob::~CJob(){
    if(NULL != m_JobName)
    free(m_JobName);
}
void CJob::SetJobName(char* jobname)
{
    if(NULL !=m_JobName)    {
        free(m_JobName);
        m_JobName = NULL;
    }
    if(NULL !=jobname)    {
        m_JobName = (char*)malloc(strlen(jobname)+1);
        strcpy(m_JobName,jobname);
    }
}
```

线程池使用示例

至此我们给出了一个简单的与具体任务无关的线程池框架。使用该框架非常的简单，我们所需要的做的就是派生CJob类，将务实现在Run方法中。然后将该Job交由CThreadManage去执行。下面我们给出一个简单的示例程序

```
```

```
class CXJob:public CJob
{
public:
    CXJob(){i=0;}
    ~CXJob(){}
    void Run(void* jobdata)    {
        printf("The Job comes from CXJOB\n");
        sleep(2);
    }
};

class CYJob:public CJob
{
public:
    CYJob(){i=0;}
    ~CYJob(){}
    void Run(void* jobdata)    {
        printf("The Job comes from CYJob\n");
    }
};

main()
{
    CThreadManage* manage = new CThreadManage(10);
    for(int i=0;i<40;i++)
    {
        CXJob*   job = new CXJob();
        manage->Run(job,NULL);
    }
    sleep(2);
    CYJob* job = new CYJob();
    manage->Run(job,NULL);
    manage->TerminateAll();
}
```


CXJob和CYJob都是从Job类继承而来，其都实现了Run接口。CXJob只是简单的打印一句”The Job comes from CXJob”打印”The Job comes from CYJob”，然后均休眠2秒钟。在主程序中我们初始创建10个工作线程。然后分别执行40次CXJ CYJob。

线程池使用后记

线程池适合场合:
事实上，线程池并不是万能的。它有其特定的使用场合。线程池致力于减少线程本身的开销对应用所产生的影响，这是有前提 线程本身开销与线程执行任务相比不可忽略。如果线程本身的开销相对于线程任务执行开销而言是可以忽略不计的，那么此时 的好处是不明显的，比如对于FTP服务器以及Telnet服务器，通常传送文件的时间较长，开销较大，那么此时，我们采用线程 的方法，我们可以选择“即时创建，即时销毁”的策略。
总之线程池通常适合下面的几个场合：
(1) 单位时间内处理任务频繁而且任务处理时间短
(2) 对实时性要求较高。如果接受到任务后在创建线程，可能满足不了实时要求，因此必须采用线程池进行预创建。
(3) 必须经常面对高突发性事件，比如Web服务器，如果有足球转播，则服务器将产生巨大的冲击。此时如果采取传统方法，大量产生线程，销毁线程。此时采用动态线程池可以避免这种情况的发生。

结束语

本文给出了一个简单的通用的与任务无关的线程池的实现，通过该线程池能够极大的简化Linux下多线程的开发工作。该线程 善开发工作还在进行中，希望能够得到你的建议和支持。
参考资料
http://www-900.ibm.com/developerWorks/cn/java/j-jtp0730/index.shtml
POSIX多线程程序设计，David R.Butenhof 译者：于磊 曾刚，中国电力出版社
C++面向对象多线程编程，CAMERON HUGHES等著 周良忠译，人民邮电出版社
Java Pro,结合线程和分析器池,Edy Yu

分类: C++,所有文章

标签: 多线程, 线程池

0

« 上一篇：万能图标提取器2013
» 下一篇：Windows下的 Axel下载工具 - 移植自Linux

posted @ 2014-04-02 13:37 XiaoKaixuan 阅读(5103) 评论