线程池原理及创建（C++实现）

时间:2010-02-25 14:40:43 来源:网络 作者:未知 点击:2963 次

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。另外该线程池具有动态伸缩性，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

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。另外该线程池具有动态伸缩性，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，通过该示例程序，我们会发现，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处理数目巨大的连接请求，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。任务执行完毕后，线程退出，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的任务是执行时间较短，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，这比开销可能占到 20%-50%左右。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上界，那么一部分线程将会等待。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。尽管这不是一个很大的数目，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，将立即创建一定数量的线程(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。当任务到来后，缓冲池选择一个空闲线程，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，用于处理更多的任务。在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，线程池自动销毁一部分线程，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，每个任务所分担到的线程本身开销则越小，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，从而是的线程池与具体的任务无关。

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，除此之外框架中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任务逻辑。

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，具有一个接口 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。

它们的类的继承关系如下图所示：

线程池的时序很简单，如下图所示。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。这儿的任务是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，并把客户端提交的任务分发给 CWorkerThread，CWorkerThread 实际执行具体的任务。

理解系统组件

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 方法也非常简单，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 CThread 都必须实现该 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，因此，我们的此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。有一点必须注意的是，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 再调用实际的 Run 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 对线程进行保存。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;
```

```
CCondition      m_BusyCond; //m_BusyCond is used to sync busy thread list

CCondition      m_IdleCond; //m_IdleCond is used to sync idle thread list

CCondition      m_IdleJobCond;  //m_JobCond is used to sync job list

CCondition      m_MaxNumCond;



vector<CWorkerThread*>   m_ThreadList;

vector<CWorkerThread*>   m_BusyList;    //Thread List

vector<CWorkerThread*>   m_IdleList; //Idle List



CThreadPool();

CThreadPool(int initnum);

virtual ~CThreadPool();
```

```
void    SetMaxNum(int maxnum){m_MaxNum = maxnum;}

int     GetMaxNum(void){return m_MaxNum;}

void    SetAvailLowNum(int minnum){m_AvailLow = minnum;}

int     GetAvailLowNum(void){return m_AvailLow;}

void    SetAvailHighNum(int highnum){m_AvailHigh = highnum;}

int     GetAvailHighNum(void){return m_AvailHigh;}

int     GetActualAvailNum(void){return m_AvailNum;}

int     GetAllNum(void){return m_ThreadList.size();}

int     GetBusyNum(void){return m_BusyList.size();}

void    SetInitNum(int initnum){m_InitNum = initnum;}

int     GetInitNum(void){return m_InitNum;}
```

```
    void    TerminateAll(void);

    void    Run(CJob* job,void* jobdata);

};

CThreadPool::CThreadPool()

{

    m_MaxNum = 50;

    m_AvailLow = 5;

    m_InitNum=m_AvailNum = 10 ;

    m_AvailHigh = 20;



    m_BusyList.clear();

    m_IdleList.clear();
```

```
    for(int i=0;i<m_InitNum;i++){

    CWorkerThread* thr = new CWorkerThread();

    thr->SetThreadPool(this);

    AppendToIdleList(thr);

    thr->Start();

    }

}



CThreadPool::CThreadPool(int initnum)

{

    assert(initnum>0 && initnum<=30);

    m_MaxNum   = 30;
```

```
m_AvailLow = initnum-10>0?initnum-10:3;

m_InitNum=m_AvailNum = initnum ;

m_AvailHigh = initnum+10;



m_BusyList.clear();

m_IdleList.clear();

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

CWorkerThread* thr = new CWorkerThread();

AppendToIdleList(thr);

thr->SetThreadPool(this);

   thr->Start();      //begin the thread,the thread wait for job

   }

}
```

```
CThreadPool::~CThreadPool()

{

  TerminateAll();

}




void CThreadPool::TerminateAll()

{

   for(int i=0;i < m_ThreadList.size();i++) {

   CWorkerThread* thr = m_ThreadList[i];

   thr->Join();

   }

   return;
```

```
}



CWorkerThread* CThreadPool::GetIdleThread(void)

{

   while(m_IdleList.size() ==0 )

   m_IdleCond.Wait();



   m_IdleMutex.Lock();

   if(m_IdleList.size() > 0 )

   {

   CWorkerThread* thr = (CWorkerThread*)m_IdleList.front();

   printf("Get Idle thread %dn",thr->GetThreadID());
```

```
    m_IdleMutex.Unlock();

    return thr;

    }

    m_IdleMutex.Unlock();



    return NULL;

}



//add an idle thread to idle list

void CThreadPool::AppendToIdleList(CWorkerThread* jobthread)

{

    m_IdleMutex.Lock();

    m_IdleList.push_back(jobthread);
```

```
    m_ThreadList.push_back(jobthread);

    m_IdleMutex.Unlock();

}



//move and idle thread to busy thread

void CThreadPool::MoveToBusyList(CWorkerThread* idlethread)

{

    m_BusyMutex.Lock();

    m_BusyList.push_back(idlethread);

    m_AvailNum--;

    m_BusyMutex.Unlock();
```

```
    m_IdleMutex.Lock();

    vector<CWorkerThread*>::iterator pos;

    pos = find(m_IdleList.begin(),m_IdleList.end(),idlethread);

    if(pos !=m_IdleList.end())

    m_IdleList.erase(pos);

    m_IdleMutex.Unlock();

}



void CThreadPool::MoveToIdleList(CWorkerThread* busythread)

{

    m_IdleMutex.Lock();

    m_IdleList.push_back(busythread);
```

```
    m_AvailNum++;

    m_IdleMutex.Unlock();



    m_BusyMutex.Lock();

    vector<CWorkerThread*>::iterator pos;

    pos = find(m_BusyList.begin(),m_BusyList.end(),busythread);

    if(pos!=m_BusyList.end())

    m_BusyList.erase(pos);

    m_BusyMutex.Unlock();



    m_IdleCond.Signal();

    m_MaxNumCond.Signal();

}
```

```
//create num idle thread and put them to idlelist

void CThreadPool::CreateIdleThread(int num)

{

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

   CWorkerThread* thr = new CWorkerThread();

   thr->SetThreadPool(this);

   AppendToIdleList(thr);

   m_VarMutex.Lock();

   m_AvailNum++;

   m_VarMutex.Unlock();

   thr->Start();      //begin the thread,the thread wait for job
```

```
    }

}



void CThreadPool::DeleteIdleThread(int num)

{

    printf("Enter into CThreadPool::DeleteIdleThreadn");

    m_IdleMutex.Lock();

    printf("Delete Num is %dn",num);

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

    CWorkerThread* thr;

    if(m_IdleList.size() > 0 ){

        thr = (CWorkerThread*)m_IdleList.front();

        printf("Get Idle thread %dn",thr->GetThreadID());
```

```
    }

    vector<CWorkerThread*>::iterator pos;

    pos = find(m_IdleList.begin(),m_IdleList.end(),thr);

    if(pos!=m_IdleList.end())

       m_IdleList.erase(pos);

    m_AvailNum--;

    printf("The idle thread available num:%d n",m_AvailNum);

    printf("The idlelist            num:%d n",m_IdleList.size());

    }

    m_IdleMutex.Unlock();

}
```

```
void CThreadPool::Run(CJob* job,void* jobdata)

{

   assert(job!=NULL);



   //if the busy thread num adds to m_MaxNum,so we should wait

   if(GetBusyNum() == m_MaxNum)

      m_MaxNumCond.Wait();



   if(m_IdleList.size()<m_AvailLow)

   {

   if(GetAllNum()+m_InitNum-m_IdleList.size() < m_MaxNum )
CreateIdleThread(m_InitNum-m_IdleList.size());

   else
```

```
    CreateIdleThread(m_MaxNum-GetAllNum());

}



CWorkerThread*  idlethr = GetIdleThread();

if(idlethr !=NULL)

{

idlethr->m_WorkMutex.Lock();

MoveToBusyList(idlethr);

idlethr->SetThreadPool(this);

job->SetWorkThread(idlethr);

printf("Job is set to thread %d n",idlethr->GetThreadID());

idlethr->SetJob(job,jobdata);

}
```

```
}
```

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，其状态变为忙碌状态，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:

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

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这些线程，因此线程执行 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 中设定四个变量：

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，此时有必要增加空闲线程池的数目。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，此时将删除多余的空闲线程，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在 m_AvailLow 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，使得线程池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，而且也不能创建新的线程，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，创建后的线程数目应该为 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，就是现有的线程总数加上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。

```
if(GetAllNum()+m_InitNum-m_IdleList.size() < m_MaxNum )
```

```
        CreateIdleThread(m_InitNum-m_IdleList.size());

    else

        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)个。

(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，同时将其移入忙碌队列。

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 GetIdleThread()中可能阻塞的线程。