文章编号:1006-2475(2010)07-0038-03

半同步半异步线程池的设计与实现

刘　燚

(南昌大学计算中心,江西 南昌 330031)

摘要:在服务器应用开发中线程池技术被广泛地用于处理大量的并发任务请求,线程池设计的好坏决定服务器的并发处理性能和效率。本文介绍用于复杂并行系统设计的半同步半异步设计模式,根据该模式设计并实现一个线程池,该线程池具有效率高、稳定性好的特点。

关键词:线程池; 半同步半异步; 多线程; 设计模式
中图分类号:TP311　　文献标识码:A　　doi: 10.3969/j.issn.1006-2475.2010.07.011

Design and Implementation of Half Sync/Half Async Thread Pool

LIU Yi

(Computing Center, Nanchang University, Nanchang 330031, China)

Abstract: The thread pool technology is widely used in server application developing to process a large number of concurrent task request and the quality of thread pool design determines the performance and efficiency of the sever. This article introduces the half sync/half async design pattern which is used to design complex concurrent system and by which design and implement a thread pool which is characterized with high efficiency and good stability.

Key words: thread pool; half sync/half async; multi thread; design pattern

0 引　言

计算机技术的发展已经步入了网络时代,各种分布式应用层出不穷,位于分布式应用核心的服务器端要处理大量的并发任务处理请求,传统的多线程技术采用为每一个请求创建一个线程的方式来处理任务请求,在任务数量不是很多的情况下能勉强应付,但是如果任务数量过多的话就会出现很多问题。大量的线程的创建和销毁将消耗过多的系统资源,大量的活动线程将占用过多系统资源,并且增加了线程上下文切换的开销,接到请求再创建线程的处理方式也会给任务的处理带来很大的处理延迟,线程池技术很好地解决了这些问题。线程池技术通过在系统中预先创建一定数量的线程,任务请求到来时分配一个线程去处理,有效地降低了处理延迟,并且线程池的线程是可重用的,避免了大量创建线程浪费系统资源,提高了并行处理的效率。管理和维护线程池也是需要消耗系统资源的,通过对线程池的设计进行优化可以减少系统资源消耗,提高处理效率。

1 线程池的设计

1.1 半同步半异步设计模式介绍

半同步半异步设计模式最早是由 ACE 的作者 Douglas C. Schmidt 提出的,他发现在操作系统和分布式应用等复杂并行系统中广泛存在着这样一种设计模式。复杂并行系统通常都存在着需要在各种抽象层面上处理的同步和异步的任务,半同步半异步设计模式总结了系统处理这些任务的方式,把系统分解成了 3 个层次:异步层、排队层、同步层,如图 1 所示。异步层异步地处理如网络接口中断等底层的任务来提高服务质量;同步层在一个独立的线程或进程中同步地处理如数据库查询、文件传输等上层任务以简化并行编程;排队层则负责协调异步层和同步层之间的通讯[1]。

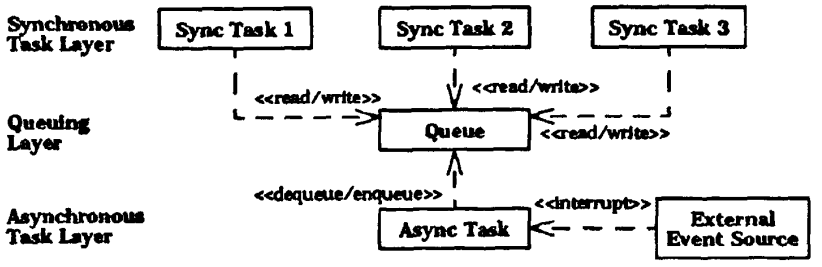

图 1　半同步半异步设计模式

收稿日期:2010-02-11

作者简介:刘燚(1985-),男,江西南昌人,南昌大学信息工程学院硕士研究生,研究方向:计算机网络,并行程序设计。

1.2 半同步半异步线程池设计

服务器系统中通常都存在着大量的异步的并发处理任务,而这些任务必须被同步地完成,线程池就是协调系统中异步层和同步层的核心部件。一般线程池由以下 4 个部分组成:

(1)任务接口:负责接收外部的异步的任务请求并把任务请求的相关信息加入任务队列。

(2)任务队列:用于缓存接收到的任务请求。

(3)管理模块:负责创建并管理线程池。

(4)工作线程:执行需要同步完成的处理任务。

一种传统的线程池设计方法是由管理模块负责任务的处理分配,当任务队列还有未处理任务时,找一个空闲的工作线程把任务分配给它处理,工作线程处理完任务后回到线程池等待管理模块给它分配新的任务,这种处理方式管理开销比较大,如果任务数量较多且执行时间很短的话效率就会很低。另一种线程池设计方法是采用主从模式(leader/follower),线程池中有一个主线程和一些从线程,主线程等待处理任务的到来,任务到来后主线程执行任务处理并推举一个从线程成为新的主线程,线程执行完任务后回到线程池成为从线程,这种处理方式能有效地减少维护线程池的开销[2]。

以上两种设计方法处理每一个任务时都需要工作线程进行两次线程状态切换(从挂起到运行、从运行到挂起),如果线程池执行的是很多执行时间很短的任务,线程上下文切换的开销将非常高。对于这种情况,有些人提出了一次给工作线程分配多个处理任务的方法[3],这种方法能减少线程上下文切换的开销,增加了任务吞吐量,但是也有其局限性。如果任务队列中同时存在运行时间很短的运算任务和运行时间很长的 IO 任务,一次分配多个任务可能会使运行时间很短的任务分配到执行时间很长的任务后面,增加处理时延,不能满足一些对处理时延有要求的应用的需要。针对这些情况,本文采用这样一种方法,所有工作线程都竞争性地等待处理任务的到来,取到工作任务的线程执行完任务后继续去取新的任务执行,如果没有任务则在线程池中等待,这样在处理任务很多时可以减少很多线程切换的开销,同时也使执行时间很短的任务得到了快速的执行。

2 线程池的实现[4-5]

线程池采用 C ++ 语言实现,限于篇幅,这里给出关键部分的实现。线程的任务队列采用标准库的 queue 模板来实现,减少了编程的复杂度和出错的可能性,由于 queue 默认是采用 deque 来实现的,效率要比采用链表要高,任务队列中存放的是以结构体 THREADPOOL_TASK 表示的任务信息,代码如下:

```
typedef void ( * THREADPOOL_FUNCTION)(LPVOID);
typedef struct _tagTHREADPOOL_TASK
{THREADPOOL_FUNCTION  pFunction;
//任务处理函数
LPVOID pPara;// 函数参数
}THREADPOOL_TASK, * PTHREADPOOL_TASK;
queue < THREADPOOL_TASK > m_TaskQueue;
```

线程池通过 ThreadPool 类来管理,类定义如下

```
typedef unsigned int (__stdcall * THREAD_PROC)
(void * );
typedef struct _tagTHREAD_INFO
{ unsigned long ThreadHandle;// 线程句柄
unsigned int ThreadID;// 线程 ID
}
THREAD_INFO, * PTHREAD_INFO;
class ThreadPool
{ queue < THREADPOOL_TASK > m_TaskQueue;
vector < THREAD_INFO > m_ThreadVector;
UINT m_ThreadCount;
HANDLE m_TaskQueueSync;
HANDLE m_ThreadVectorSync;
public:
static unsigned int __stdcall WorkerThreadStart( void *
pPara);
public:
ThreadPool( );
virtual ~ThreadPool( );
bool Initialize( UINT ThreadCount, UINT MaxThreadCount,
UINT MaxTaskCount);
void Uninitialize( );
bool AssignTask(PTHREADPOOL_TASK pTask);
void WorkerThreadStart( );
private:
UINT m_MaxThreadCount;
UINT m_MaxTaskCount;
private:
void NotifyAll( UINT nMsg, WPARAM wParam, LPARAM
lParam);
void RemoveThread( UINT ThreadCount);
void AddNewThread( UINT ThreadCount);
};
```

线程池通过 Initialize 函数初始化,ThreadCount 表是初始线程数,MaxThreadCount 表示线程池允许的最大线程数,MaxTaskCount 表示任务队列可缓存的最大任务数,通过 AddNewThread 向线程池中增加新的工作线程,RemoveThread 移除多余的工作线程,ThreadCount 表示要创建或移除的线程数。线程池中的工作线程信息存放在一个结构体中,记录线程的句柄和线程 ID,便于管理线程,这些线程信息被组织到一个 vector 中,vector 在尾端插入删除数据的效率较高,而线程信息也是只在尾部添加删除,故采用 vec-

tor 来实现。m_TaskQueueSync 和 m_ThreadVectorSync 分别用来同步对任务队列和线程管理信息的访问,保护数据的完整性。任务是通过类接口函数 AssignTask 加入到任务队列中的,参数采用传递指针的形式减少数据拷贝带来的开销,实现代码如下:

```
bool ThreadPool:: AssignTask ( PTHREADPOOL _ TASK pTask)
{bool bRet = false;
WaitForSingleObject( m_TaskQueueSync, THREAD_POOL_WAIT_TIMEOUT);
if ( m_TaskQueue. size( ) < m_MaxTaskCount)
{m_TaskQueue. push( * pTask);
bRet = true;
NotifyAll( WM_THREADPOOL_TASK,0,0 );
}
ReleaseMutex( m_TaskQueueSync);
return bRet;
}
void ThreadPool::NotifyAll( UINT Msg, WPARAM wParam, LPARAM lParam)
{for ( int i =0;i < m_ThreadVector. size( ); i ++ )
{ if( m_ThreadVector[ i]. ThreadID ! =0 )
PostThreadMessage( m_ThreadVector[ i]. ThreadID, Msg, wParam, lParam );
}
}
```

线程池收到任务请求后把请求放入任务队列中并发送消息通知所有工作线程有工作任务到来,工作线程获取消息后去任务队列中取出任务并执行,工作线程函数代码如下:

```
void ThreadPool::WorkerThreadStart( )
{MSG msg = {0};
while( GetMessage( &msg, NULL,0,0)! = WM_QUIT)
{if( msg. message == WM_THREADPOOL_TASK)
{THREADPOOL_TASK TaskInfo;
WaitForSingleObject( m_TaskQueueSync, THREAD_POOL_WAIT_TIMEOUT);
if ( m_TaskQueue. empty( ) == false)
{TaskInfo = m_TaskQueue. front( );
m_TaskQueue. pop( );
}
ReleaseMutex( m_TaskQueueSync);
if ( TaskInfo. pFunction ! = NULL)
TaskInfo. pFunction( TaskInfo. pPara);
}
}
}
```

工作线程收到的任务通知要比实际任务多很多,但是如果任务队列没有任务的话,通知就会被抛弃,不影响线程池的正常运行。通过这种方式可以使工作线程在处理完一个任务后接着处理下一个任务,避免了线程的频繁挂起,减少了线程切换的开销。

3 结束语

线程池的设计方式多种多样,本文设计的线程池目标是站在兼顾通用性和效率的角度来设计的,在为具体应用设计线程池时往往需要根据应用的需求(如处理延迟、吞吐量等)和处理任务的特点来优化线程池设计,采用不同的策略调整线程池的工作线程数,才能达到最好的效果。

参考文献:

[1] Douglas C Schmidt. Half Sync/Half Async Pattern[EB/OL]. http://www. cs. wustl. edu/ ~ schmidt/PDF/HS-HA. pdf,1999-11-26.

[2] 李刚,金蓓弘. 两种线程池的实现和性能评价[J]. 计算机工程与设计,2007,28(7):1489-1492.

[3] 赵海,李志蜀,韩学为,等. 线程池的优化设计[J],四川大学学报(自然科学版),2005,42(1):63-67.

[4] Stanley B Lippman, Josée LaJoie, Barbara E Moo. C ++ Primer(第 4 版)[M]. 李师贤,蒋爱军,梅晓勇,等译. 北京:人民邮电出版社,2006.

[5] [美]理查德. Windows 核心编程(第 4 版)[M]. 黄陇,李虎译. 北京:机械工业出版社,2008.

[6] Sutter Herb. Choose concurrency-friendly data structures [J]. Dr. Dobb's Journal: The World of Software Development,2008,33(7):57-59.

[7] Erich Gamma, Richard Helm, Ralph Johnson, et al. 设计模式:可复用面向对象软件的基础[M]. 李英军,马晓星,蔡敏,等译. 北京:机械工业出版社,2000.

[8] 郭辉. 多线程的效率[J]. 计算机应用,2008,28(S2):141-143.

[9] Jim Beveridge, Robert Wiener. Win32 多线程程序设计[M]. 侯捷译. 武汉:华中科技大学出版社,2002.

[10] 欧阳志鹏,沈富可. 基于改进线程池技术服务器程序的设计与实现[J]. 计算机与数字工程,2005,33(10):133-136.

[11] Yibei Ling, Tracy Mullen, Xiaola Lin. Analysis of optimal thread pool size[J]. Operating Systems Review,2000,34(2):42-55.

[12] 吴丹,傅秀芬,苏磊,等. 多线程编程模型的研究与应用[J]. 广东工业大学学报,2008,25(1):47-49.

[13] 夏利,赵静波,井惟栋,等. 基于对象池模式的自适应线程池技术[J]. 东北大学学报(自然科学版),2006,27(10):1091-1094.

[14] Nathan R Tallent, John M Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications [C]//Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2009:229-240.