　大多数的网络服务器，包括Web服务器都具有一个特点，就是单位时间内必须处理数目巨大的连接请求，但是处理时间却是比较短的。在传统的多线程服务器模型中是这样实现的：一旦有个请求到达，就创建一个新的线程，由该线程执行任务，任务执行完毕之后，线程就退出。这就是"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，而且执行次数非常频繁，那么服务器就将处于一个不停的创建线程和销毁线程的状态。这笔开销是不可忽略的，尤其是线程执行的时间非常非常短的情况。

　　线程池就是为了解决上述问题的，它的实现原理是这样的：在应用程序启动之后，就马上创建一定数量的线程，放入空闲的队列中。这些线程都是处于阻塞状态，这些线程只占一点内存，不占用CPU。当任务到来后，线程池将选择一个空闲的线程，将任务传入此线程中运行。当所有的线程都处在处理任务的时候，线程池将自动创建一定的数量的新线程，用于处理更多的任务。执行任务完成之后线程并不退出，而是继续在线程池中等待下一次任务。当大部分线程处于阻塞状态时，线程池将自动销毁一部分的线程，回收系统资源。

　　下面是一个简单线程池的实现，这个线程池的代码是我参考网上的一个例子实现的，由于找不到出处了，就没办法注明参考自哪里了。它的方案是这样的：程序启动之前，初始化线程池，启动线程池中的线程，由于还没有任务到来，线程池中的所有线程都处在阻塞状态，当一有任务到达就从线程池中取出一个空闲线程处理，如果所有的线程都处于工作状态，就添加到队列，进行排队。如果队列中的任务个数大于队列的所能容纳的最大数量，那就不能添加任务到队列中，只能等待队列不满才能添加任务到队列中。

　　主要由两个文件组成一个threadpool.h头文件和一个threadpool.c源文件组成。源码中已有重要的注释，就不加以分析了。

　　threadpool.h文件：

struct job

{

void\* (\*callback\_function)(void \*arg); //线程回调函数

void \*arg; //回调函数参数

struct job \*next;

};

struct threadpool

{

int thread\_num; //线程池中开启线程的个数

int queue\_max\_num; //队列中最大job的个数

struct job \*head; //指向job的头指针

struct job \*tail; //指向job的尾指针

pthread\_t \*pthreads; //线程池中所有线程的pthread\_t

pthread\_mutex\_t mutex; //互斥信号量

pthread\_cond\_t queue\_empty; //队列为空的条件变量

pthread\_cond\_t queue\_not\_empty; //队列不为空的条件变量

pthread\_cond\_t queue\_not\_full; //队列不为满的条件变量

int queue\_cur\_num; //队列当前的job个数

int queue\_close; //队列是否已经关闭

int pool\_close; //线程池是否已经关闭

};

//================================================================================================

//函数名： threadpool\_init

//函数描述： 初始化线程池

//输入： [in] thread\_num 线程池开启的线程个数

// [in] queue\_max\_num 队列的最大job个数

//输出： 无

//返回： 成功：线程池地址 失败：NULL

//================================================================================================

struct threadpool\* threadpool\_init(int thread\_num, int queue\_max\_num);

//================================================================================================

//函数名： threadpool\_add\_job

//函数描述： 向线程池中添加任务

//输入： [in] pool 线程池地址

// [in] callback\_function 回调函数

// [in] arg 回调函数参数

//输出： 无

//返回： 成功：0 失败：-1

//================================================================================================

int threadpool\_add\_job(struct threadpool \*pool, void\* (\*callback\_function)(void \*arg), void \*arg);

//================================================================================================

//函数名： threadpool\_destroy

//函数描述： 销毁线程池

//输入： [in] pool 线程池地址

//输出： 无

//返回： 成功：0 失败：-1

//================================================================================================

int threadpool\_destroy(struct threadpool \*pool);

//================================================================================================

//函数名： threadpool\_function

//函数描述： 线程池中线程函数

//输入： [in] arg 线程池地址

//输出： 无

//返回： 无

//================================================================================================

void\* threadpool\_function(void\* arg);

[复制代码](javascript:void(0);)

　　threadpool.c文件：

[复制代码](javascript:void(0);)

#include "threadpool.h"

struct threadpool\* threadpool\_init(int thread\_num, int queue\_max\_num)

{

struct threadpool \*pool = NULL;

do

{

pool = malloc(sizeof(struct threadpool));

if (NULL == pool)

{

printf("failed to malloc threadpool!\n");

break;

}

pool->thread\_num = thread\_num;

pool->queue\_max\_num = queue\_max\_num;

pool->queue\_cur\_num = 0;

pool->head = NULL;

pool->tail = NULL;

if (pthread\_mutex\_init(&(pool->mutex), NULL))

{

printf("failed to init mutex!\n");

break;

}

if (pthread\_cond\_init(&(pool->queue\_empty), NULL))

{

printf("failed to init queue\_empty!\n");

break;

}

if (pthread\_cond\_init(&(pool->queue\_not\_empty), NULL))

{

printf("failed to init queue\_not\_empty!\n");

break;

}

if (pthread\_cond\_init(&(pool->queue\_not\_full), NULL))

{

printf("failed to init queue\_not\_full!\n");

break;

}

pool->pthreads = malloc(sizeof(pthread\_t) \* thread\_num);

if (NULL == pool->pthreads)

{

printf("failed to malloc pthreads!\n");

break;

}

pool->queue\_close = 0;

pool->pool\_close = 0;

int i;

for (i = 0; i < pool->thread\_num; ++i)

{

pthread\_create(&(pool->pthreads[i]), NULL, threadpool\_function, (void \*)pool);

}

return pool;

} while (0);

return NULL;

}

int threadpool\_add\_job(struct threadpool\* pool, void\* (\*callback\_function)(void \*arg), void \*arg)

{

assert(pool != NULL);

assert(callback\_function != NULL);

assert(arg != NULL);

pthread\_mutex\_lock(&(pool->mutex));

while ((pool->queue\_cur\_num == pool->queue\_max\_num) && !(pool->queue\_close || pool->pool\_close))

{

pthread\_cond\_wait(&(pool->queue\_not\_full), &(pool->mutex)); //队列满的时候就等待

}

if (pool->queue\_close || pool->pool\_close) //队列关闭或者线程池关闭就退出

{

pthread\_mutex\_unlock(&(pool->mutex));

return -1;

}

struct job \*pjob =(struct job\*) malloc(sizeof(struct job));

if (NULL == pjob)

{

pthread\_mutex\_unlock(&(pool->mutex));

return -1;

}

pjob->callback\_function = callback\_function;

pjob->arg = arg;

pjob->next = NULL;

if (pool->head == NULL)

{

pool->head = pool->tail = pjob;

pthread\_cond\_broadcast(&(pool->queue\_not\_empty)); //队列空的时候，有任务来时就通知线程池中的线程：队列非空

}

else

{

pool->tail->next = pjob;

pool->tail = pjob;

}

pool->queue\_cur\_num++;

pthread\_mutex\_unlock(&(pool->mutex));

return 0;

}

void\* threadpool\_function(void\* arg)

{

struct threadpool \*pool = (struct threadpool\*)arg;

struct job \*pjob = NULL;

while (1) //死循环

{

pthread\_mutex\_lock(&(pool->mutex));

while ((pool->queue\_cur\_num == 0) && !pool->pool\_close) //队列为空时，就等待队列非空

{

pthread\_cond\_wait(&(pool->queue\_not\_empty), &(pool->mutex));

}

if (pool->pool\_close) //线程池关闭，线程就退出

{

pthread\_mutex\_unlock(&(pool->mutex));

pthread\_exit(NULL);

}

pool->queue\_cur\_num--;

pjob = pool->head;

if (pool->queue\_cur\_num == 0)

{

pool->head = pool->tail = NULL;

}

else

{

pool->head = pjob->next;

}

if (pool->queue\_cur\_num == 0)

{

pthread\_cond\_signal(&(pool->queue\_empty)); //队列为空，就可以通知threadpool\_destroy函数，销毁线程函数

}

if (pool->queue\_cur\_num == pool->queue\_max\_num - 1)

{

pthread\_cond\_broadcast(&(pool->queue\_not\_full)); //队列非满，就可以通知threadpool\_add\_job函数，添加新任务

}

pthread\_mutex\_unlock(&(pool->mutex));

(\*(pjob->callback\_function))(pjob->arg); //线程真正要做的工作，回调函数的调用

free(pjob);

pjob = NULL;

}

}

int threadpool\_destroy(struct threadpool \*pool)

{

assert(pool != NULL);

pthread\_mutex\_lock(&(pool->mutex));

if (pool->queue\_close || pool->pool\_close) //线程池已经退出了，就直接返回

{

pthread\_mutex\_unlock(&(pool->mutex));

return -1;

}

pool->queue\_close = 1; //置队列关闭标志

while (pool->queue\_cur\_num != 0)

{

pthread\_cond\_wait(&(pool->queue\_empty), &(pool->mutex)); //等待队列为空

}

pool->pool\_close = 1; //置线程池关闭标志

pthread\_mutex\_unlock(&(pool->mutex));

pthread\_cond\_broadcast(&(pool->queue\_not\_empty)); //唤醒线程池中正在阻塞的线程

pthread\_cond\_broadcast(&(pool->queue\_not\_full)); //唤醒添加任务的threadpool\_add\_job函数

int i;

for (i = 0; i < pool->thread\_num; ++i)

{

pthread\_join(pool->pthreads[i], NULL); //等待线程池的所有线程执行完毕

}

pthread\_mutex\_destroy(&(pool->mutex)); //清理资源

pthread\_cond\_destroy(&(pool->queue\_empty));

pthread\_cond\_destroy(&(pool->queue\_not\_empty));

pthread\_cond\_destroy(&(pool->queue\_not\_full));

free(pool->pthreads);

struct job \*p;

while (pool->head != NULL)

{

p = pool->head;

pool->head = p->next;

free(p);

}

free(pool);

return 0;

}

[复制代码](javascript:void(0);)

　　测试文件main.c文件：

[复制代码](javascript:void(0);)

#include "threadpool.h"

void\* work(void\* arg)

{

char \*p = (char\*) arg;

printf("threadpool callback fuction : %s.\n", p);

sleep(1);

}

int main(void)

{

struct threadpool \*pool = threadpool\_init(10, 20);

threadpool\_add\_job(pool, work, "1");

threadpool\_add\_job(pool, work, "2");

threadpool\_add\_job(pool, work, "3");

threadpool\_add\_job(pool, work, "4");

threadpool\_add\_job(pool, work, "5");

threadpool\_add\_job(pool, work, "6");

threadpool\_add\_job(pool, work, "7");

threadpool\_add\_job(pool, work, "8");

threadpool\_add\_job(pool, work, "9");

threadpool\_add\_job(pool, work, "10");

threadpool\_add\_job(pool, work, "11");

threadpool\_add\_job(pool, work, "12");

threadpool\_add\_job(pool, work, "13");

threadpool\_add\_job(pool, work, "14");

threadpool\_add\_job(pool, work, "15");

threadpool\_add\_job(pool, work, "16");

threadpool\_add\_job(pool, work, "17");

threadpool\_add\_job(pool, work, "18");

threadpool\_add\_job(pool, work, "19");

threadpool\_add\_job(pool, work, "20");

threadpool\_add\_job(pool, work, "21");

threadpool\_add\_job(pool, work, "22");

threadpool\_add\_job(pool, work, "23");

threadpool\_add\_job(pool, work, "24");

threadpool\_add\_job(pool, work, "25");

threadpool\_add\_job(pool, work, "26");

threadpool\_add\_job(pool, work, "27");

threadpool\_add\_job(pool, work, "28");

threadpool\_add\_job(pool, work, "29");

threadpool\_add\_job(pool, work, "30");

threadpool\_add\_job(pool, work, "31");

threadpool\_add\_job(pool, work, "32");

threadpool\_add\_job(pool, work, "33");

threadpool\_add\_job(pool, work, "34");

threadpool\_add\_job(pool, work, "35");

threadpool\_add\_job(pool, work, "36");

threadpool\_add\_job(pool, work, "37");

threadpool\_add\_job(pool, work, "38");

threadpool\_add\_job(pool, work, "39");

threadpool\_add\_job(pool, work, "40");

sleep(5);

threadpool\_destroy(pool);

return 0;

}

[复制代码](javascript:void(0);)

　　用gcc编译，运行就可以看到效果，1到40个回调函数分别被执行。

s