线程池实现.md

# 线程池

## **池技术**

常见的池：线程池、内存池、数据库连接池、请求池、消息队列（消息池）、对象池 —— 目的：缓冲

## 线程池技术

线程池： 解决什么问题？ —— ①减少线程上下文切换的开销, 实现任务调度 ② 异步解耦作用：如写磁盘、落盘两个操作放在两个线程内实现; IO检测与读写分离

线程池特点：线程复用、控制最大并发数、管理线程

## **线程池组成**

一般的线程池由以下四部分组成：

* 管理组件/线程池管理器：用于创建并管理线程
* 执行队列/工作线程, 线程：线程池中的线程
* 任务队列, 任务：用于存放待处理的任务，提供一种缓冲机制
* 任务接口：每个任务必须实现的接口，提供一种缓冲机制

## 工作线程管理问题

* 如何添加线程
* 如何减少线程
* 增加和减少的策略

## 线程实现分析

**关键结构体：**

NWORKER: 工作线程，双向链表

NJOB： 任务队列节点，双向链表

NMANAGER: 线程池管理组件

**关键函数/宏：**

LL\_ADD(宏): 添加任务到任务队列

LL\_REMOVE(宏)： 移除任务队列中的任务节点

nThreadPoolCreat(nThreadPool \*pool, int numWorkers): 线程池创建

nThreadPoolDestroy(nThreadPool \*pool): 线程池销毁

nThreadPoolPush(nThreadPool \*pool, struct NJOB \*job): 线程池中添加任务

nThreadCallBack(void \*arg): 线程池任务描述

## 线程池工作的四种情况

**主线程当前没任务要执行，线程池中的任务队列为空闲状态**

此情况下所有工作线程处于空闲的等待状态pthread\_cond\_wait，任务缓冲队列为空

**主程序添加小于等于线程池中线程数量的任务**

此情况基于情况1，所有线程处于等待状态，主线程开始添加任务(数量小于线程数量)，添加后通知pthread\_cond\_signal()唤醒线程池中的线程开始获取任务执行。此时的任务缓冲队列还是空。

**主程序添加大于当前线程池中线程数量的任务**

此情况发生在情况2后面，所有工作线程都在工作中，主线程添加一个任务，添加后发现线程池中的任务用完了，于是存入任务缓冲队列。工作线程空闲后主动从任务队列取任务执行。

**主程序添加任务数量大于当前线程池中线程数量的任务，且任务缓冲队列已满**

此情况发生在情况3且设置了任务缓冲队列大小后面，主程序添加第N个任务，添加后发现池子中的线程用完了，任务缓冲队列也满了，于是进入等待状态、等待任务缓冲队列中的任务腾空通知。

但要注意这种情形会阻塞主线程，对此暂不限制任务队列大小。

## 线程池的简单实现1(基于C)

#include "pthread.h"
  
   
#define LL\_ADD(item, list) do { \
  
 item->prev = NULL; \
  
 item->next = list; \
  
 if(list != NULL) list->prev = item; \
  
 list = item; \
  
 } while(0)
  
   
#define LL\_REMOVE(item, list) do{ \
  
 if(item->prev != NULL) item->prev->next = item->next; \
  
 if(item->next != NULL) item->next->prev = item->prev; \
  
 if(list == item) list = item->next; \
  
 item->prev = item->next = NULL; \
  
 } while(0)
  
   
   
struct NWORKER{//执行队列
  
   
 pthread\_t thread;
  
 struct NMANAGER \*pool; //线程池
  
 int terminal;
  
   
 struct NWORKER \*prev;
  
 struct NWORKER \*next;
  
};
  
   
struct NJOB{ //任务队列
  
   
 void (\*func)(struct NJOB \*job);
  
 void \*user\_data;
  
   
 struct NJOB \*prev;
  
 struct NJOB \*next;
  
};
  
   
struct NMANAGER{//管理组件
  
   
 struct NWORKER \*workers;
  
 struct NJOB \*jobs;
  
   
 pthread\_cond\_t jobs\_cond; //等待条件满足 ： 任务到来
  
 pthread\_mutext\_t jobs\_mutex; //锁
  
   
};
  
   
typedef struct NMANAGER nThreadPool;
  
   
static void \*nThreadCallBack(void \*arg){ //描述线程的执行任务
  
   
 struct NWORKER \*worker = (struct NWORKER\*)arg;
  
   
 while(1){
  
   
 pthread\_mutex\_lock(&worder->pool->jobs\_mutex);
  
 while(worder->pool->jobs == NULL){
  
 if(worker->terminal) break;
  
 pthread\_cond\_wait(&worker->pool->jobs\_cond, worker->pool->jobs\_mutex); //情况1
  
 }
  
 if(worder->terminal){
  
 pthread\_mutex\_unlock(&worder->pool->jobs\_mutex);
  
 break;
  
 }
  
   
 struct NJOB \*job = worker->pool->jobs;
  
 LL\_REMOVE(job, worder->pool->jobs);
  
   
 pthread\_mutex\_unlock(&worder->pool->jobs\_mutex);
  
   
 job->func(job->user\_data);
  
 }
  
   
 free(worker);
  
 worker = NULL;
  
 pthread\_exit(NULL); //会实现线程栈的回收
  
 // ptread\_cancel(threadid); 在另外一个线程内结束该进程
  
}
  
   
   
//Thread Pool Creat
  
int nThreadPoolCreat(nThreadPool \*pool, int numWorkers) {
  
   
 if(numWorkers < 1) numWorkers = 1;
  
 if(pool == NULL) return -1;
  
 memset(pool, 0, sizeof(nThreadPool);
  
   
 pthread\_cond\_t black\_cond = PTHREAD\_COND\_INITIALIZER; //初始化条件变量
  
 memcpy(&pool->jobs\_cond, &black\_cond, sizeof(pthread\_cond\_t));
  
   
 pthread\_mutext\_t blank\_mutex = PTHREAD\_MUTEX\_INITIALIZER; //初始化互斥锁
  
 memcpy(&pool->jobs\_mutex, &blank\_mutex, sizeof(pthread\_mutext\_t));
  
   
 int i = 0;
  
 for(; i< numWorkers; i++){
  
   
 struct NWORKER \*worker = (struct NWORKER\*)malloc(sizeof(struct NWORKER));
  
 if(worker == NULL){
  
 perror("malloc");
  
 return -2;
  
 }
  
   
 memset(worker, 0, sizeof(struct NWORKER));
  
 worker->pool = pool;
  
   
 int ret = pthread\_creat(&worder->thread, NULL, nThreadCallBack, worker);
  
 if(ret){
  
 perror("pthread\_cread");
  
 return -3;
  
 }
  
   
 LL\_ADD(worker, pool->workers);
  
 } // end for ; i< numWorkers; i++
  
}
  
   
// Thread Pool Destroy
  
int nThreadPoolDestroy(nThreadPool \*pool){
  
   
 struct NWORKER \*worker = NULL;
  
 for(work = pool->workers; worker != NULL; worker = worker->next){
  
 worker->terminal = 1;
  
 }
  
   
 pthread\_mutex\_lock(pool->jobs\_mutex);
  
 pthread\_cond\_broadcast(pool->jobs\_cond);
  
 pthread\_mutex\_unlock(pool->jobs\_mutex);
  
}
  
   
void nThreadPoolPush(nThreadPool \*pool, struct NJOB \*job){ //添加任务
  
   
 pthread\_mutex\_lock(&pool->jobs\_mutex);
  
   
 LL\_ADD(job, pool->jobs);
  
 pthread\_cond\_signal(&pool->jobs\_cond);
  
   
 pthread\_mutex\_unlock(&pool->jobs\_mutex);
  
   
}
  
   
   
#if 1 //debug
  
// 0 --> 1000的计数
  
// task -->
  
int main(){
  
   
 nThreadPool \*pool;
  
   
 for()
  
}
  
#endif

## 线程池的实现2(基于C)

https://www.cnblogs.com/zhaoosheLBJ/p/9337291.html

### 线程池结构

#### 线程池任务节点

线程池任务结点用来保存用户投递过来的的任务，并放入线程池中的线程来执行，任务结构如下：

//线程池任务节点  
struct worker\_t{  
 void \*(\*process)(void \*arg); //回调函数  
 int paratype; //函数类型(预留)  
 void \*arg； //回调函数参数  
 struct worker\_t \*next; //链表  
}

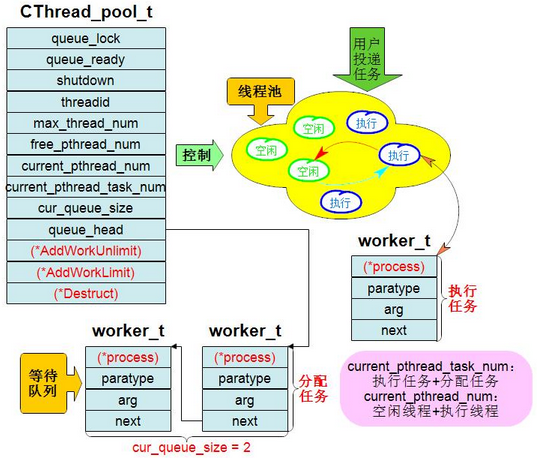
#### 线程池管理组件

线程池控制器用来对线程池进行控制管理，描述当前线程池的最基本信息，包括任务的投递，线

程池状态的更新与查询，线程池的销毁等，其结构如下：

//线程控制器  
struct cThreadPool\_t{  
 worker\_t \*queue\_head; //任务节点链表，保存所有投递的任务  
 pthread\_t \*threadid; //线程列表，以线程ID标识  
   
 pthread\_mutex\_t queue\_lock; //互斥锁  
 pthread\_cond\_t queue\_ready; //条件变量  
 int shutdown; //线程销毁标志  
   
 int max\_thread\_num; //线程池线程最大容量  
 int current\_pthread\_num; //当前线程池中线程数量  
 int current\_pthread\_task\_num; //当前已执行任务和已分配任务线程数量  
 int current\_wait\_queue\_num; //当前等待队列的任务数量  
 int free\_pthread\_num; //线程池允许最大的空闲数量  
   
 /\*\*  
 \* function: ThreadPoolAddWorkUnlimit  
 \* description: 向线程池投递任务  
 \* input param: pthis 线程池指针  
 \* process 回调函数  
 \* arg 回调函数参数  
 \* return Val: 0 成功  
 \* -1 失败  
 \*\*/  
 int (\*AddWorkUnlimit)(void \*pthis, void \*(\*process)(void \*arg), void \*arg);  
   
 /\*\*  
 \* function: ThreadPoolAddWorkLimit  
 \* description: 向线程池投递任务  
 \* input param: pthis 线程池指针  
 \* process 回调函数  
 \* arg 回调函数参数  
 \* return Val: 0 成功  
 \* -1 失败  
 \*\*/  
 int (\*AddWorkLimit)(void \*pthis, void \*(\*process)(void \*arg), void \*arg);  
   
 /\*\*  
 \* function: ThreadPoolGetThreadMaxNum  
 \* description: 获取线程池可以容纳的最大线程数  
 \* input param: pthis 线程池指针  
 \* return Val: 线程池能容纳的最大数量  
 \*\*/  
 int (\*GetThreadMaxNum)(void \*pthis);  
   
 /\*\*  
 \* function: ThreadPoolGetCurrentThreadNum  
 \* description: 获取线程池存放的线程数  
 \* input param: pthis 线程池指针  
 \* return Val: 线程池存放的线程数  
 \*\*/  
 int (\*GetCurrentThreadNum)(void \*pthis);  
   
 /\*\*  
 \* function: ThreadPoolGetCurrentWaitTaskNum  
 \* description: 获取线程池等待队列任务数  
 \* input param: pthis 线程池指针  
 \* return Val: 等待队列任务数  
 \*\*/  
 int (\*GetCurrentWaitTaskNum)(void\* pthis);  
   
 /\*\*  
 \* funciton: ThreadPoolDestroy  
 \* description: 销毁线程池  
 \* input param： pthis 线程池指针  
 \* return Val: 0 成功  
 \* -1 失败  
 \*\*/  
 int (\*Destroy)(void \*pthis);  
};

#### 线程池运行结构



img

* 图中线程池中的“空闲”和“执行”分别表示空闲线程和执行线程，二者是相互转换的。当用户投递任务过来，则用空闲线程来执行该任务，且空闲线程转换为执行线程；当任务执行完成后，执行线程状态变为空闲线程。
* 创建线程池时，正常情况下会创建一定数量的线程，所有线程初始化为空闲线程，线程阻塞等待用户投递任务。
* 用户投递的任务首先放入等待队列queue\_head链表中，如果线程池中有空闲线程则放入空闲线程中执行，否则根据条件选择继续等待空闲线程或新建一个线程来执行，新建的线程将放入线程池中。
* 执行的任务会从等待队列中脱离，并在任务执行完成后释放任务节点worker\_t

### 线程池控制/部分函数

#### 线程池创建

创建max\_num个线程ThreadPoolRoutine，即空闲线程

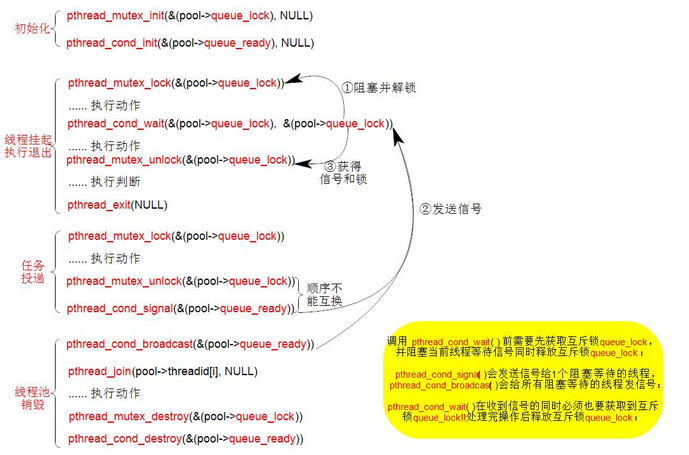
/\*\*  
\* function: ThreadPoolConstruct  
\* description: 构建线程池  
\* input param: max\_num 线程池可容纳的最大线程数  
\* free\_num 线程池允许存在的最大空闲线程，超过则将线程释放回操作系统  
\* return Val: 线程池指针  
\*\*/  
cThreadPool\_t\* ThreadPoolConstruct(int max\_num, int free\_num){  
 int i = 0;  
   
 cTreadPool\_t \*pool = (cThreadPool\_t \*)malloc(sizeof(cThreadPool\_t));  
 if(pool == NULL)  
 return NULL;  
 memset(pool, 0, sizeof(cTreadPool\_t));  
   
 //初始化互斥锁  
 pthread\_mutex\_init(&(pool->queue\_lock), NULL);  
 //初始化条件变量  
 pthread\_cond\_init(&(pool->queue\_ready), NULL);  
 //变量赋初值  
 pool->queue\_head = NULL;  
 pool->max\_thread\_num = max\_num;  
 pool->current\_wait\_queue\_num = 0;  
 pool->current\_pthread\_task\_num = 0;  
 pool->shutdown = 0;  
 pool->current\_pthread\_num = 0;  
 pool->free\_thread\_num = free\_num;  
 pool->threadid = NULL;  
 pool->threadid = (pthread\_t \*)malloc(max\_num\*sizeof(pthread\_t));  
 memset(pool->thread\_id, 0, max\_num\*sizeof(pthread\_t));  
 //函数指针赋值  
 pool->AddWorkUnlimit = ThreadPoolAddWorkUnlimit;  
 pool->AddWorkLimit = ThreadPoolAddWorkLimit;  
 pool->Destroy = ThreadPoolDestroy;  
 pool->GetThreadMaxNum = ThreadPoolGetThreadMaxNum;  
 pool->GetCurrentThreadNum = ThreadPoolGetCurrentThreadNum;  
 pool->GetCurrentTaskThreadNum = ThreadPoolGetCurrentTaskThreadNum;  
 pool->GetCurrentWaitTaskNum = ThreadPoolGetCurrentWaitTaskNum;  
   
 for(int i = 0; i < max\_num; i++){  
 pool->current\_pthread\_num++; //当前线程池中线程数  
 //创建线程  
 int ret = pthread\_creat(&(pool->threadid[i]), NULL, ThreadPoolRoutine, (void \*)pool);  
 if(ret){  
 perror("pthread\_cread");  
 return -3;  
 }  
 usleep(1000);  
 }  
 return pool;  
}

#### 投递任务

/\*\*  
\* function: ThreadPoolAddWorkLimit  
\* description: 向线程池投递任务，无空闲线程则阻塞  
\* input param: pthis 线程池指针  
\* process 回调函数  
\* arg 回调函数参数  
\* return Val: 0 成功  
\* -1 失败  
\*\*/  
int ThreadPoolAddWorkLimit(void \*pthis, void \*(\*process)(void \*arg), void \*arg){  
   
 cTreadPool\_t \*pool = (cTreadPool\_t \*)pthis;  
   
 //为添加的任务队列节点分配内存  
 worker\_t \*new\_worker = (worker\_t \*)malloc(sizeof(worker\_t));  
 if(new\_worker == NULL){  
 perror("malloc");  
 return -1;  
 }  
   
 new\_worker->process = process; //回调函数，在线程ThreadPoolRoutine()中执行  
 new\_worker->arg = arg; //回调函数参数  
 new\_worker->next = NULL;  
   
 pthread\_mutex\_lock(&(pool->queue\_lock)); //加锁  
   
 //插入新任务节点  
 worker\_t \*member = pool->queue\_head; //指向任务队列链表整体  
 if(member != NULL){ //将任务插到队尾  
 while(member->next != NULL)  
 member = member->next;  
 member->next = new\_worker;  
 }else{  
 pool->queue\_head = new\_worker;   
 }  
   
 assert(pool->queue\_head != NULL);  
 pool->current\_wait\_queue\_num++; //等待队列加1  
   
 //空闲线程数量 = 当前线程池存放的线程 - 当前已经执行任务和已分配任务的线程数目和  
 int free\_thread\_num = pool->current\_pthread\_num - pool->current\_pthread\_task\_num;  
 //如果没有空闲线程且池中当前线程数不超过可容纳最大线程数量  
 if((free\_thread\_num == 0) && (pool->current\_thread\_num < pool->current\_pthread\_task\_num){//条件为真->进行新线程创建  
 int current\_pthread\_num = pool->current\_pthread\_num;  
   
 //新增线程  
 pool->threadid = (pthread\_t \*)realloc(pool->threadid, (current\_thread\_num + 1)\*sizeof(pthread\_t));  
   
 pthread\_creat(&(pool->threadid[current\_pthread\_num]), NULL, ThreadPoolRoutine, (void \*)pool);  
   
 //当前线程中线程总数加一  
 ++pool->current\_pthread\_num;  
   
 //分配任务线程数加一  
 pool->current\_pthread\_task\_num++;  
   
 pthread\_mutex\_unlock(&(pool->queue\_lock));//解锁  
   
 //唤醒一个线程来执行任务  
 pthread\_cond\_signal(&(pool->queue\_ready));  
   
 return 0;//成功添加任务  
 }  
}

投递任务时先创建一个任务结点保存回调函数和函数参数，并将任务结点放入等待队列中。如果没有空闲线程切线程数量未达到最大数量的限制，则会创建一个新的线程。realloc()会在保存原始内存中数据不变的基础上新增一个sizeof(pthread\_t)大小内存，用于新增一个线程。之后更新current\_pthread\_num和current\_pthread\_task\_num；并发送信号pthread\_cond\_signal(&(pool->queue\_read))，给一个处于条件阻塞等待状态的线程，即线程ThreadPoolRoutine()中pthread\_cond\_wait(&(pool->queue\_ready), &(pool->queu\_lock))阻塞等待信号。

实现线程池的一个关键点在于互斥锁和条件变量： pthread\_mutex\_t queue\_head和pthread\_cond\_t queue\_ready。



#### 执行线程

/\*\*  
\* function: ThreadPoolRoutine  
\* description: 线程池中线程的执行函数  
\* input param: arg 线程池指针  
\*\*/  
void \*ThreadPoolRoutine(void \*arg){  
 cThreadPool\_t \*pool = (cThreadPool \*)arg;  
   
 while(1){  
 //上锁， pthread\_cond\_wait()调用会解锁  
 pthread\_mutex\_lock(&(pool->queue\_lock));  
   
 //队列中没有任务  
 while(pool->current\_wait\_queue\_num == 0 && !pool->shutdown){  
 pthread\_cond\_wait(&(pol->queue\_ready), &(pool->queue\_lock));//条件锁阻塞，等待条件信号  
 }  
   
 if(pool->shutdown){  
 pthread\_mutex\_unlock(&(pool->queue\_lock));  
 break;  
 }  
   
 assert(pool->current\_wait\_queue\_num != 0);  
 assert(pool->queue\_head != NULL);  
   
 pool->current\_wait\_queue\_num--; //等待任务减一，准备执行任务  
 worker\_t \*worker = pool->queue\_head; //去等待任务队列节点头  
 pool->queue\_head = worker->next; //移除头节点  
 pthread\_mutex\_unlock(&(pool->queue\_lock));  
   
 (\*(worker->process))(worker->arg); //执行回调函数  
   
 pthread\_mutex\_lock(&(pool->queue\_lock));  
 pool->current\_pthread\_task\_num--; //函数执行结束  
 free(worker); //释放任务节点  
 worker = NULL;  
   
 if((pool->current\_pthread\_num - pool->current\_pthread\_task\_num) > pool->free\_pthread\_num) {  
 pthread\_mutex\_unlock(&(pool->queue\_lock);  
 break; //当线程池中空闲数量超过free\_pthread\_num 则将线程释放回操作系统  
 }  
 pthread\_mutex\_unlock(&(pool->queue\_lock));  
 }  
 pool->current\_pthread\_num--;  
 pthread\_exit(NULL);  
   
 return (void \*)NULL;  
}

#### 销毁线程池

/\*\*  
\* function: ThreadPoolDestroy  
\* description: 销毁线程池  
\* input param: pthis 线程池指针  
\* return val: 0 成功  
\* -1 失败  
\*\*/  
int ThreadPoolDestroy(void \*pthis){  
 int i;  
 cThreadPool\_t \*pool = (cThreadPool\_t \*)pthis;  
   
 if(pool->shutdown)  
 return -1; //已销毁  
   
 pool->shutdown = 1; //销毁标志位  
   
 //唤醒所有pthread\_cond\_wait()等待线程  
 pthread\_cond\_broadcast(&(pool->queu\_ready));  
 for(int i = 0; i < pool->current\_thread\_num; ++i)  
 pthread\_join(pool->threadid[i], NULL); //等待所有线程执行结束  
   
 free(pool->threadid);  
   
 //销毁任务队列链表  
 worker\_t \*head = NULL;  
 while(pool\_queue\_head != NULL){  
 head = pool->queue\_head;  
 pool->queue\_head = pool->queue\_head->next;  
 free(head);  
 }  
   
 //销毁锁  
 pthread\_mutex\_destroy(&(pool->queue\_lock));  
 pthread\_cond\_destroy(&(pool->queue\_ready));  
   
 free(pool);  
 pool = NULL;  
   
 return 0;  
}

销毁线程池只需要将销毁标志位置1，线程池中的线程就会停止运行。然后需要销毁创建线程池时申请的资源。

### 完整代码

#### cThreadPool.h

/\*\*  
 \* 线程池头文件  
 \*  
 \*\*/  
  
#ifndef \_CTHREADPOOL\_H\_  
#define \_CTHREADPOOL\_H\_  
  
#include <pthread.h>  
  
/\*线程池可容纳最大线程数\*/  
#define DEFAULT\_MAX\_THREAD\_NUM 100  
  
/\*线程池允许最大的空闲线程，超过则将线程释放回操作系统\*/  
#define DEFAULT\_FREE\_THREAD\_NUM 10  
  
typedef struct worker\_t worker\_t;  
typedef struct cThreadPool\_t cThreadPool\_t;  
  
/\*线程池任务节点\*/  
struct worker\_t {  
 void \* (\* process)(void \* arg); /\*回调函数\*/  
 int paratype; /\*函数类型(预留)\*/  
 void \* arg; /\*回调函数参数\*/  
 struct worker\_t \* next; /\*链接下一个任务节点\*/  
};  
  
/\*线程控制器\*/  
struct cThreadPool\_t {  
 pthread\_mutex\_t queue\_lock; /\*互斥锁\*/  
 pthread\_cond\_t queue\_ready; /\*条件变量\*/  
   
 worker\_t \* queue\_head; /\*任务节点链表 保存所有投递的任务\*/  
 int shutdown; /\*线程池销毁标志 1-销毁\*/  
 pthread\_t \* threadid; /\*线程ID\*/  
   
 int max\_thread\_num; /\*线程池可容纳最大线程数\*/  
 int current\_pthread\_num; /\*当前线程池存放的线程\*/  
 int current\_pthread\_task\_num; /\*当前已经执行任务和已分配任务的线程数目和\*/  
 int current\_wait\_queue\_num; /\*当前等待队列的的任务数目\*/  
 int free\_pthread\_num; /\*线程池允许最大的空闲线程数/\*/  
   
 /\*\*  
 \* function: ThreadPoolAddWorkUnlimit  
 \* description: 向线程池投递任务  
 \* input param: pthis 线程池指针  
 \* process 回调函数  
 \* arg 回调函数参数  
 \* return Valr: 0 成功  
 \* -1 失败  
 \*/   
 int (\* AddWorkUnlimit)(void \* pthis, void \* (\* process)(void \* arg), void \* arg);  
   
 /\*\*  
 \* function: ThreadPoolAddWorkLimit  
 \* description: 向线程池投递任务,无空闲线程则阻塞  
 \* input param: pthis 线程池指针  
 \* process 回调函数  
 \* arg 回调函数参数  
 \* return Val: 0 成功  
 \* -1 失败  
 \*/   
 int (\* AddWorkLimit)(void \* pthis, void \* (\* process)(void \* arg), void \* arg);  
   
 /\*\*  
 \* function: ThreadPoolGetThreadMaxNum  
 \* description: 获取线程池可容纳的最大线程数  
 \* input param: pthis 线程池指针  
 \*/   
 int (\* GetThreadMaxNum)(void \* pthis);  
   
 /\*\*  
 \* function: ThreadPoolGetCurrentThreadNum  
 \* description: 获取线程池存放的线程数  
 \* input param: pthis 线程池指针  
 \* return Val: 线程池存放的线程数  
 \*/   
 int (\* GetCurrentThreadNum)(void \* pthis);  
   
 /\*\*  
 \* function: ThreadPoolGetCurrentTaskThreadNum  
 \* description: 获取当前正在执行任务和已经分配任务的线程数目和  
 \* input param: pthis 线程池指针  
 \* return Val: 当前正在执行任务和已经分配任务的线程数目和  
 \*/   
 int (\* GetCurrentTaskThreadNum)(void \* pthis);  
   
 /\*\*  
 \* function: ThreadPoolGetCurrentWaitTaskNum  
 \* description: 获取线程池等待队列任务数  
 \* input param: pthis 线程池指针  
 \* return Val: 等待队列任务数  
 \*/   
 int (\* GetCurrentWaitTaskNum)(void \* pthis);  
   
 /\*\*  
 \* function: ThreadPoolDestroy  
 \* description: 销毁线程池  
 \* input param: pthis 线程池指针  
 \* return Val: 0 成功  
 \* -1 失败  
 \*/   
 int (\* Destroy)(void \* pthis);   
};  
  
/\*\*  
 \* function: ThreadPoolConstruct  
 \* description: 构建线程池  
 \* input param: max\_num 线程池可容纳的最大线程数  
 \* free\_num 线程池允许存在的最大空闲线程,超过则将线程释放回操作系统  
 \* return Val: 线程池指针   
 \*/   
cThreadPool\_t \* ThreadPoolConstruct(int max\_num, int free\_num);  
  
/\*\*  
 \* function: ThreadPoolConstructDefault  
 \* description: 创建线程池,以默认的方式初始化,未创建线程  
 \*  
 \* return Val: 线程池指针   
 \*/   
cThreadPool\_t \* ThreadPoolConstructDefault(void);  
  
#endif // \_CTHREADPOOL\_H\_

#### cThreadPool.c

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <pthread.h>  
#include <assert.h>  
  
#include "cThreadPool.h"  
  
void \* ThreadPoolRoutine(void \* arg);  
  
/\*\*  
\* function: ThreadPoolAddWorkLimit  
\* description: 向线程池投递任务，无空闲线程则阻塞  
\* input param: pthis 线程池指针  
\* process 回调函数  
\* arg 回调函数参数  
\* return Val: 0 成功  
\* -1 失败  
\*\*/  
int ThreadPoolAddWorkLimit(void \*pthis, void \*(\*process)(void \*arg), void \*arg){  
   
 cTreadPool\_t \*pool = (cTreadPool\_t \*)pthis;  
   
 //为添加的任务队列节点分配内存  
 worker\_t \*new\_worker = (worker\_t \*)malloc(sizeof(worker\_t));  
 if(new\_worker == NULL){  
 perror("malloc");  
 return -1;  
 }  
   
 new\_worker->process = process; //回调函数，在线程ThreadPoolRoutine()中执行  
 new\_worker->arg = arg; //回调函数参数  
 new\_worker->next = NULL;  
   
 pthread\_mutex\_lock(&(pool->queue\_lock)); //加锁  
   
 //插入新任务节点  
 worker\_t \*member = pool->queue\_head; //指向任务队列链表整体  
 if(member != NULL){ //将任务插到队尾  
 while(member->next != NULL)  
 member = member->next;  
 member->next = new\_worker;  
 }else{  
 pool->queue\_head = new\_worker;   
 }  
   
 assert(pool->queue\_head != NULL);  
 pool->current\_wait\_queue\_num++; //等待队列加1  
   
 //空闲线程数量 = 当前线程池存放的线程 - 当前已经执行任务和已分配任务的线程数目和  
 int free\_thread\_num = pool->current\_pthread\_num - pool->current\_pthread\_task\_num;  
 //如果没有空闲线程且池中当前线程数不超过可容纳最大线程数量  
 if((free\_thread\_num == 0) && (pool->current\_thread\_num < pool->current\_pthread\_task\_num){//条件为真->进行新线程创建  
 int current\_pthread\_num = pool->current\_pthread\_num;  
   
 //新增线程  
 pool->threadid = (pthread\_t \*)realloc(pool->threadid, (current\_thread\_num + 1)\*sizeof(pthread\_t));  
   
 pthread\_creat(&(pool->threadid[current\_pthread\_num]), NULL, ThreadPoolRoutine, (void \*)pool);  
   
 //当前线程中线程总数加一  
 ++pool->current\_pthread\_num;  
   
 //分配任务线程数加一  
 pool->current\_pthread\_task\_num++;  
   
 //唤醒一个线程来执行任务  
 pthread\_cond\_signal(&(pool->queue\_ready));  
   
 pthread\_mutex\_unlock(&(pool->queue\_lock));//解锁  
   
 return 0;//成功添加任务  
 }  
}  
   
/\*\*  
 \* function: ThreadPoolAddWorkUnlimit  
 \* description: 向线程池投递任务  
 \* input param: pthis 线程池指针  
 \* process 回调函数  
 \* arg 回调函数参数  
 \* return Valr: 0 成功  
 \* -1 失败  
 \*/  
int  
ThreadPoolAddWorkUnlimit(void \* pthis, void \* (\* process)(void \* arg), void \* arg)  
{  
 // int FreeThreadNum = 0;  
 // int CurrentPthreadNum = 0;  
   
 cThreadPool\_t \* pool = (cThreadPool\_t \*)pthis;  
   
 /\*给新任务队列节点分配内存\*/  
 worker\_t \* new\_worker = (worker\_t \*)malloc(sizeof(worker\_t));  
 if(NULL == new\_worker)  
 return -1;  
   
 new\_worker->process = process; // 回调函数  
 new\_worker->arg = arg; // 回调函数参数  
 new\_worker->next = NULL;  
   
 pthread\_mutex\_lock(&(pool->queue\_lock));  
   
 /\*新节点插入任务队列链表操作\*/  
 worker\_t \* member = pool->queue\_head;  
 if(member != NULL) {  
 while(member->next != NULL)  
 member = member->next;  
   
 member->next = new\_worker; // 插入队列链表尾部  
 } else   
 pool->queue\_head = new\_worker; // 插入到头(也就是第一个节点,之前链表没有节点)  
   
 assert(pool->queue\_head != NULL);  
 pool->current\_wait\_queue\_num++; // 当前等待队列的的任务数目+1  
   
 int FreeThreadNum = pool->current\_pthread\_num - pool->current\_pthread\_task\_num;  
 /\*只判断是否没有空闲线程\*/  
 if(0 == FreeThreadNum) {  
 int CurrentPthreadNum = pool->current\_pthread\_num;  
 pool->threadid = (pthread\_t \*)realloc(pool->threadid,  
 (CurrentPthreadNum+1)\*sizeof(pthread\_t));  
   
 pthread\_create(&(pool->threadid[CurrentPthreadNum]),NULL,  
 ThreadPoolRoutine, (void \*)pool);  
 pool->current\_pthread\_num++;  
 if(pool->current\_pthread\_num > pool->max\_thread\_num)  
 pool->max\_thread\_num = pool->current\_pthread\_num;  
   
 pool->current\_pthread\_task\_num++;  
 pthread\_mutex\_unlock(&(pool->queue\_lock));  
 pthread\_cond\_signal(&(pool->queue\_ready));  
 return 0;  
 }  
   
 pool->current\_pthread\_task\_num++;  
 pthread\_cond\_signal(&(pool->queue\_ready));  
 pthread\_mutex\_unlock(&(pool->queue\_lock));  
// usleep(10);   
 return 0;   
}  
   
/\*\*  
 \* function: ThreadPoolGetThreadMaxNum  
 \* description: 获取线程池可容纳的最大线程数  
 \* input param: pthis 线程池指针  
 \* return val: 线程池可容纳的最大线程数  
 \*/   
int  
ThreadPoolGetThreadMaxNum(void \* pthis)  
{  
 int num = 0;   
 cThreadPool\_t \* pool = (cThreadPool\_t \*)pthis;  
   
 pthread\_mutex\_lock(&(pool->queue\_lock));  
 num = pool->max\_thread\_num;  
 pthread\_mutex\_unlock(&(pool->queue\_lock));  
   
 return num;  
}  
  
/\*\*  
 \* function: ThreadPoolGetCurrentThreadNum  
 \* description: 获取线程池存放的线程数  
 \* input param: pthis 线程池指针  
 \* return Val: 线程池存放的线程数  
 \*/   
int   
ThreadPoolGetCurrentThreadNum(void \* pthis)  
{  
 int num = 0;  
 cThreadPool\_t \* pool = (cThreadPool\_t \*)pthis;  
   
 pthread\_mutex\_lock(&(pool->queue\_lock));  
 num = pool->current\_pthread\_num;  
 pthread\_mutex\_unlock(&(pool->queue\_lock));  
   
 return num;   
}  
  
/\*\*  
 \* function: ThreadPoolGetCurrentTaskThreadNum  
 \* description: 获取当前正在执行任务和已经分配任务的线程数目和  
 \* input param: pthis 线程池指针  
 \* return Val: 当前正在执行任务和已经分配任务的线程数目和  
 \*/   
int  
ThreadPoolGetCurrentTaskThreadNum(void \* pthis)  
{  
 int num = 0;  
 cThreadPool\_t \* pool = (cThreadPool\_t \*)pthis;  
   
 pthread\_mutex\_lock(&(pool->queue\_lock));  
 num = pool->current\_pthread\_task\_num;  
 pthread\_mutex\_unlock(&(pool->queue\_lock));  
   
 return num;   
}  
  
/\*\*  
 \* function: ThreadPoolGetCurrentWaitTaskNum  
 \* description: 获取线程池等待队列任务数  
 \* input param: pthis 线程池指针  
 \* return Val: 等待队列任务数  
 \*/   
int  
ThreadPoolGetCurrentWaitTaskNum(void \* pthis)  
{  
 int num = 0;  
 cThreadPool\_t \* pool = (cThreadPool\_t \*)pthis;  
   
 pthread\_mutex\_lock(&(pool->queue\_lock));  
 num = pool->current\_wait\_queue\_num;  
 pthread\_mutex\_unlock(&(pool->queue\_lock));  
   
 return num;   
}  
  
/\*\*  
\* function: ThreadPoolDestroy  
\* description: 销毁线程池  
\* input param: pthis 线程池指针  
\* return val: 0 成功  
\* -1 失败  
\*\*/  
int ThreadPoolDestroy(void \*pthis){  
 int i;  
 cThreadPool\_t \*pool = (cThreadPool\_t \*)pthis;  
   
 if(pool->shutdown)  
 return -1; //已销毁  
   
 pool->shutdown = 1; //销毁标志位  
   
 //唤醒所有pthread\_cond\_wait()等待线程  
 pthread\_cond\_broadcast(&(pool->queu\_ready));  
 for(int i = 0; i < pool->current\_thread\_num; ++i)  
 pthread\_join(pool->threadid[i], NULL); //等待所有线程执行结束  
   
 free(pool->threadid);  
   
 //销毁任务队列链表  
 worker\_t \*head = NULL;  
 while(pool\_queue\_head != NULL){  
 head = pool->queue\_head;  
 pool->queue\_head = pool->queue\_head->next;  
 free(head);  
 }  
   
 //销毁锁  
 pthread\_mutex\_destroy(&(pool->queue\_lock));  
 pthread\_cond\_destroy(&(pool->queue\_ready));  
   
 free(pool);  
 pool = NULL;  
   
 return 0;  
}  
   
/\*\*  
\* function: ThreadPoolRoutine  
\* description: 线程池中线程的执行函数  
\* input param: arg 线程池指针  
\*\*/  
void \*ThreadPoolRoutine(void \*arg){  
 cThreadPool\_t \*pool = (cThreadPool \*)arg;  
   
 while(1){  
 //上锁， pthread\_cond\_wait()调用会解锁  
 pthread\_mutex\_lock(&(pool->queue\_lock));  
   
 //队列中没有任务  
 while(pool->current\_wait\_queue\_num == 0 && !pool->shutdown){  
 pthread\_cond\_wait(&(pol->queue\_ready), &(pool->queue\_lock));//条件锁阻塞，等待条件信号  
 }  
   
 if(pool->shutdown){  
 pthread\_mutex\_unlock(&(pool->queue\_lock));  
 break;  
 }  
   
 assert(pool->current\_wait\_queue\_num != 0);  
 assert(pool->queue\_head != NULL);  
   
 pool->current\_wait\_queue\_num--; //等待任务减一，准备执行任务  
 worker\_t \*worker = pool->queue\_head; //去等待任务队列节点头  
 pool->queue\_head = worker->next; //移除头节点  
 pthread\_mutex\_unlock(&(pool->queue\_lock));  
   
 (\*(worker->process))(worker->arg); //执行回调函数  
   
 pthread\_mutex\_lock(&(pool->queue\_lock));  
 pool->current\_pthread\_task\_num--; //函数执行结束  
 free(worker); //释放任务节点  
 worker = NULL;  
   
 if((pool->current\_pthread\_num - pool->current\_pthread\_task\_num) > pool->free\_pthread\_num) {  
 pthread\_mutex\_unlock(&(pool->queue\_lock);  
 break; //当线程池中空闲数量超过free\_pthread\_num 则将线程释放回操作系统  
 }  
 pthread\_mutex\_unlock(&(pool->queue\_lock));  
 }  
 pool->current\_pthread\_num--;  
 pthread\_exit(NULL);  
   
 return (void \*)NULL;  
}  
   
/\*\*  
\* function: ThreadPoolConstruct  
\* description: 构建线程池  
\* input param: max\_num 线程池可容纳的最大线程数  
\* free\_num 线程池允许存在的最大空闲线程，超过则将线程释放回操作系统  
\* return Val: 线程池指针  
\*\*/  
cThreadPool\_t\* ThreadPoolConstruct(int max\_num, int free\_num){  
 int i = 0;  
   
 cTreadPool\_t \*pool = (cThreadPool\_t \*)malloc(sizeof(cThreadPool\_t));  
 if(pool == NULL)  
 return NULL;  
 memset(pool, 0, sizeof(cTreadPool\_t));  
   
 //初始化互斥锁  
 pthread\_mutex\_init(&(pool->queue\_lock), NULL);  
 //初始化条件变量  
 pthread\_cond\_init(&(pool->queue\_ready), NULL);  
 //变量赋初值  
 pool->queue\_head = NULL;  
 pool->max\_thread\_num = max\_num;  
 pool->current\_wait\_queue\_num = 0;  
 pool->current\_pthread\_task\_num = 0;  
 pool->shutdown = 0;  
 pool->current\_pthread\_num = 0;  
 pool->free\_thread\_num = free\_num;  
 pool->threadid = NULL;  
 pool->threadid = (pthread\_t \*)malloc(max\_num\*sizeof(pthread\_t));  
 memset(pool->thread\_id, 0, max\_num\*sizeof(pthread\_t));  
 //函数指针赋值  
 pool->AddWorkUnlimit = ThreadPoolAddWorkUnlimit;  
 pool->AddWorkLimit = ThreadPoolAddWorkLimit;  
 pool->Destroy = ThreadPoolDestroy;  
 pool->GetThreadMaxNum = ThreadPoolGetThreadMaxNum;  
 pool->GetCurrentThreadNum = ThreadPoolGetCurrentThreadNum;  
 pool->GetCurrentTaskThreadNum = ThreadPoolGetCurrentTaskThreadNum;  
 pool->GetCurrentWaitTaskNum = ThreadPoolGetCurrentWaitTaskNum;  
   
 for(int i = 0; i < max\_num; i++){  
 pool->current\_pthread\_num++; //当前线程池中线程数  
 //创建线程  
 int ret = pthread\_creat(&(pool->threadid[i]), NULL, ThreadPoolRoutine, (void \*)pool);  
 if(ret){  
 perror("pthread\_cread");  
 return -3;  
 }  
 usleep(1000);  
 }  
 return pool;  
}  
   
/\*\*  
 \* function: ThreadPoolConstructDefault  
 \* description: 创建线程池,以默认的方式初始化,未创建线程  
 \*  
 \* return Val: 线程池指针   
 \*/   
cThreadPool\_t \*   
ThreadPoolConstructDefault(void)  
{  
 cThreadPool\_t \* pool = (cThreadPool\_t \*)malloc(sizeof(cThreadPool\_t));  
 if(NULL == pool)  
 return NULL;  
   
 memset(pool, 0, sizeof(cThreadPool\_t));  
   
 pthread\_mutex\_init(&(pool->queue\_lock), NULL);  
 pthread\_cond\_init(&(pool->queue\_ready), NULL);  
   
 pool->queue\_head = NULL;  
 pool->max\_thread\_num = DEFAULT\_MAX\_THREAD\_NUM; // 默认值  
 pool->current\_wait\_queue\_num = 0;  
 pool->current\_pthread\_task\_num = 0;  
 pool->shutdown = 0;  
 pool->current\_pthread\_num = 0;  
 pool->free\_pthread\_num = DEFAULT\_FREE\_THREAD\_NUM; // 默认值  
 pool->threadid = NULL;  
 /\*该函数指针赋值\*/  
 pool->AddWorkUnlimit = ThreadPoolAddWorkUnlimit;  
 pool->AddWorkLimit = ThreadPoolAddWorkLimit;  
 pool->Destroy = ThreadPoolDestroy;  
 pool->GetThreadMaxNum = ThreadPoolGetThreadMaxNum;  
 pool->GetCurrentThreadNum = ThreadPoolGetCurrentThreadNum;  
 pool->GetCurrentTaskThreadNum = ThreadPoolGetCurrentTaskThreadNum;  
 pool->GetCurrentWaitTaskNum = ThreadPoolGetCurrentWaitTaskNum;  
   
 return pool;  
}

#### 测试main.c文件

#include <stdio.h>   
#include <stdlib.h>   
#include <unistd.h>   
#include <sys/types.h>   
#include <pthread.h>   
#include <assert.h>   
#include <string.h>  
  
#include "cThreadPool.h"  
  
  
void \* thread\_1(void \* arg);  
void \* thread\_2(void \* arg);  
void \* thread\_3(void \* arg);  
void DisplayPoolStatus(cThreadPool\_t \* pPool);  
  
int nKillThread = 0;  
  
int main() {  
 cThreadPool\_t \* pThreadPool = NULL;  
   
 pThreadPool = ThreadPoolConstruct(5, 1);  
 int nNumInput = 5;  
 char LogInput[] = "OK!";  
  
 DisplayPoolStatus(pThreadPool);  
 /\*可用AddWorkLimit()替换看执行的效果\*/  
 pThreadPool->AddWorkUnlimit((void \*)pThreadPool, (void \*)thread\_1, (void \*)NULL);  
 /\*  
 \* 没加延迟发现连续投递任务时pthread\_cond\_wait()会收不到信号pthread\_cond\_signal() !!  
 \* 因为AddWorkUnlimit()进去后调用pthread\_mutex\_lock()把互斥锁锁上,导致pthread\_cond\_wait()  
 \* 收不到信号!!也可在AddWorkUnlimit()里面加个延迟,一般情况可能也遇不到这个问题  
 \*/  
 usleep(10);   
 pThreadPool->AddWorkUnlimit((void \*)pThreadPool, (void \*)thread\_2, (void \*)nNumInput);  
 usleep(10);  
 pThreadPool->AddWorkUnlimit((void \*)pThreadPool, (void \*)thread\_3, (void \*)LogInput);  
 usleep(10);  
 DisplayPoolStatus(pThreadPool);  
  
 nKillThread = 1;  
 usleep(100); /\*\*< 先让线程退出 \*/  
 DisplayPoolStatus(pThreadPool);  
 nKillThread = 2;  
 usleep(100);  
 DisplayPoolStatus(pThreadPool);  
 nKillThread = 3;  
 usleep(100);  
 DisplayPoolStatus(pThreadPool);  
  
 pThreadPool->Destroy((void\*)pThreadPool);  
 return 0;  
}  
  
void \*   
thread\_1(void \* arg)  
{  
 printf("Thread 1 is running !\n");  
 while(nKillThread != 1)  
 usleep(10);  
 return NULL;  
}  
  
void \* thread\_2(void \* arg)  
{  
 int nNum = (int)arg;  
   
 printf("Thread 2 is running !\n");  
 printf("Get Number %d\n", nNum);  
 while(nKillThread != 2)  
 usleep(10);  
 return NULL;  
}  
  
void \* thread\_3(void \* arg)  
{  
 char \* pLog = (char \*)arg;  
   
 printf("Thread 3 is running !\n");  
 printf("Get String %s\n", pLog);  
 while(nKillThread != 3)  
 usleep(10);  
 return NULL;  
}  
  
void DisplayPoolStatus(cThreadPool\_t \* pPool)  
{  
 static int nCount = 1;  
   
 printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("nCount = %d\n", nCount++);  
 printf("max\_thread\_num = %d\n", pPool->GetThreadMaxNum((void \*)pPool));  
 printf("current\_pthread\_num = %d\n", pPool->GetCurrentThreadNum((void \*)pPool));  
 printf("current\_pthread\_task\_num = %d\n", pPool->GetCurrentTaskThreadNum((void \*)pPool));  
 printf("current\_wait\_queue\_num = %d\n", pPool->GetCurrentWaitTaskNum((void \*)pPool));  
 printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
}

#### MakeFile

CC = gcc  
CFLAGS = -g -Wall -o2  
LIB = -lpthread  
  
RUNE = $(CC) $(CFLAGS) $(object) -o $(exe) $(LIB)  
RUNO = $(CC) $(CFLAGS) -c $< -o $@ $(LIB)  
  
.RHONY:clean  
  
  
object = main.o cThreadPool.o  
exe = cThreadpool  
  
$(exe):$(object)  
 $(RUNE)  
  
%.o:%.c cThreadPool.h  
 $(RUNO)  
%.o:%.c  
 $(RUNO)  
  
  
clean:  
 -rm -rf \*.o cThreadpool \*~