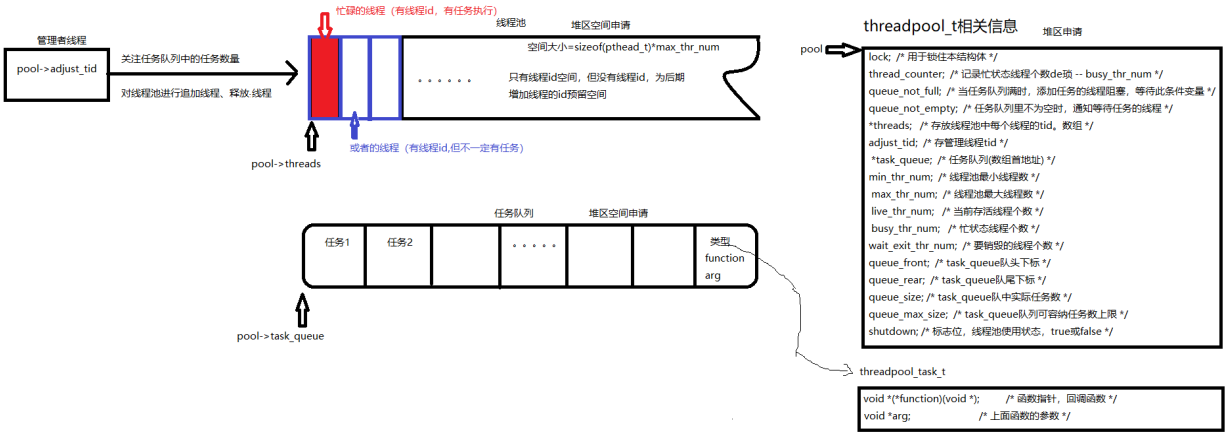
知识点1【线程池】（了解）



知识点2【线程池源码分析】（了解）

05_threadpool.h
1.33KB

05_pthread_pool.c
16.46KB

05_threadpool.h

```c
#ifndef __THREADPOOL_H_
#define __THREADPOOL_H_

typedef struct threadpool_t threadpool_t;

/**
 * @function threadpool_create
 * @descCreates a threadpool_t object.
 * @param thr_num  thread num
 * @param max_thr_num  max thread size
 * @param queue_max_size   size of the queue.
 * @return a newly created thread pool or NULL
 */
```

```
14 threadpool_t *threadpool_create(int min_thr_num, int max_thr_num, int qu
eue_max_size);
15
16 /**
17  * @function threadpool_add
18  * @desc add a new task in the queue of a thread pool
19  * @param pool     Thread pool to which add the task.
20  * @param function Pointer to the function that will perform the task.
21  * @param argument Argument to be passed to the function.
22  * @return 0 if all goes well,else -1
23  */
24 int threadpool_add(threadpool_t *pool, void*(*function)
(void *arg), void *arg);
25
26 /**
27  * @function threadpool_destroy
28  * @desc Stops and destroys a thread pool.
29  * @param pool  Thread pool to destroy.
30  * @return 0 if destory success else -1
31  */
32 int threadpool_destroy(threadpool_t *pool);
33
34 /**
35  * @desc get the thread num
36  * @pool pool threadpool
37  * @return # of the thread
38  */
39 int threadpool_all_threadnum(threadpool_t *pool);
40
41 /**
42  * desc get the busy thread num
43  * @param pool threadpool
44  * return # of the busy thread
45  */
46 int threadpool_busy_threadnum(threadpool_t *pool);
47
48 #endif
49
```

05_pthread_pool.c

```
1 #include <stdlib.h>
2 #include <pthread.h>
```

```c
#include <unistd.h>
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <errno.h>
#include "05_threadpool.h"

#include <sys/socket.h> //socket
#include <netinet/in.h> //struct sockaddr_in
#include <arpa/inet.h>  //inet_pton  inet_addr

typedef struct
{
    int cfd;                //存放已连接套接字
    struct sockaddr_in addr; //存放客户端的信息
} CLIENT_MSG;

#define DEFAULT_TIME 10         /*10s检测一次*/
#define MIN_WAIT_TASK_NUM 10    /*如果queue_size > MIN_WAIT_TASK_NUM 添加新的线程到线程池*/
#define DEFAULT_THREAD_VARY 10 /*每次创建和销毁线程的个数*/
#define true 1
#define false 0

typedef struct
{
    void *(*function)(void *); /* 函数指针，回调函数 */
    void *arg;                 /* 上面函数的参数 */
} threadpool_task_t;           /* 各子线程任务结构体 */

/* 描述线程池相关信息 */
struct threadpool_t
{
    pthread_mutex_t lock;           /* 用于锁住本结构体 */
    pthread_mutex_t thread_counter; /* 记录忙状态线程个数de琐 -- busy_thr_num */

    pthread_cond_t queue_not_full;  /* 当任务队列满时，添加任务的线程阻塞，等待此条件变量 */
    pthread_cond_t queue_not_empty; /* 任务队列里不为空时，通知等待任务的线程 */
```

```
41
42      pthread_t *threads;              /* 存放线程池中每个线程的tid。数组 */
43      pthread_t adjust_tid;            /* 存管理线程tid */
44      threadpool_task_t *task_queue; /* 任务队列(数组首地址) */
45
46      int min_thr_num;        /* 线程池最小线程数 */
47      int max_thr_num;        /* 线程池最大线程数 */
48      int live_thr_num;       /* 当前存活线程个数 */
49      int busy_thr_num;       /* 忙状态线程个数 */
50      int wait_exit_thr_num; /* 要销毁的线程个数 */
51
52      int queue_front;     /* task_queue队头下标 */
53      int queue_rear;      /* task_queue队尾下标 */
54      int queue_size;      /* task_queue队中实际任务数 */
55      int queue_max_size; /* task_queue队列可容纳任务数上限 */
56
57      int shutdown; /* 标志位，线程池使用状态，true或false */
58  };
59
60  void *threadpool_thread(void *threadpool);
61
62  void *adjust_thread(void *threadpool);
63
64  int is_thread_alive(pthread_t tid);
65  int threadpool_free(threadpool_t *pool);
66  //创建TCP服务器
67  int create_tcp_socket(unsigned short port)
68  {
69      //创建tcp监听套接字
70      int lfd = socket(AF_INET, SOCK_STREAM, 0);
71      if (lfd < 0)
72      {
73          perror("socket");
74          _exit(-1);
75      }
76
77      //设置端口复用
78      int yes = 1;
79      setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));
80
81      //bind给lfd绑定固定的ip port
```

```
82      struct sockaddr_in my_addr;
83      bzero(&my_addr, sizeof(my_addr));
84      my_addr.sin_family = AF_INET;
85      my_addr.sin_port = htons(port);
86      my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
87      int ret = bind(lfd, (struct sockaddr *)&my_addr, sizeof(my_addr));
88      if (ret < 0)
89      {
90          perror("bind");
91          _exit(-1);
92      }
93
94      //listen进行监听
95      listen(lfd, 128);
96
97      return lfd;
98  }
99  //threadpool_create(3,100,100);
100  threadpool_t *threadpool_create(int min_thr_num, int max_thr_num, int qu
eue_max_size)
101  {
102      int i;
103      threadpool_t *pool = NULL;
104      do
105      {
106          if ((pool = (threadpool_t *)malloc(sizeof(threadpool_t))) == NUl
L)
107          {
108              printf("malloc threadpool fail");
109              break; /*跳出do while*/
110          }
111
112          pool->min_thr_num = min_thr_num;
113          pool->max_thr_num = max_thr_num;
114          pool->busy_thr_num = 0;
115          pool->live_thr_num = min_thr_num; /* 活着的线程数 初值=最小线程
数 */
116          pool->wait_exit_thr_num = 0;
117          pool->queue_size = 0; /* 有0个产品 */
118          pool->queue_max_size = queue_max_size;
119          pool->queue_front = 0;
```

```
120         pool->queue_rear = 0;
121         pool->shutdown = false; /* 不关闭线程池 */
122 
123         /* 根据最大线程上限数， 给工作线程数组开辟空间, 并清零 */
124         pool->threads = (pthread_t *)malloc(sizeof(pthread_t) * max_thr_
num);
125         if (pool->threads == NULL)
126         {
127             printf("malloc threads fail");
128             break;
129         }
130         memset(pool->threads, 0, sizeof(pthread_t) * max_thr_num);
131 
132         /* 队列开辟空间 */
133         pool->task_queue = (threadpool_task_t *)malloc(sizeof(threadpool
_task_t) * queue_max_size);
134         if (pool->task_queue == NULL)
135         {
136             printf("malloc task_queue fail\n");
137             break;
138         }
139 
140         /* 初始化互斥琐、条件变量 */
141         if (pthread_mutex_init(&(pool->lock), NULL) != 0 || pthread_mute
x_init(&(pool->thread_counter), NULL) != 0 || pthread_cond_init(&(pool->que
ue_not_empty), NULL) != 0 || pthread_cond_init(&(pool->queue_not_full), NUL
L) != 0)
142         {
143             printf("init the lock or cond fail\n");
144             break;
145         }
146 
147         //启动工作线程
148         pthread_attr_t attr;
149         pthread_attr_init(&attr);
150         pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
151         for (i = 0; i < min_thr_num; i++)
152         {
153             pthread_create(&(pool-
>threads[i]), &attr, threadpool_thread, (void *)pool); /*pool指向当前线程池*/
154             printf("start thread 0x%x...\n", (unsigned int)pool-
>threads[i]);
```

```
155         }
156
157         //创建管理者线程
158         pthread_create(&(pool->adjust_tid), &attr, adjust_thread, (void *)pool);
159
160         return pool;
161
162     } while (0);
163
164     /* 前面代码调用失败时,释放poll存储空间 */
165     threadpool_free(pool);
166
167     return NULL;
168 }
169
170 /* 向线程池中 添加一个任务 */
171 //threadpool_add(thp, process, (void*)&num[i]);   /* 向线程池中添加任务 process: 小写---->大写*/
172
173 int threadpool_add(threadpool_t *pool, void *(*function)(void *arg), void *arg)
174 {
175     pthread_mutex_lock(&(pool->lock));
176
177     /* ==为真，队列已经满，  调wait阻塞 */
178     while ((pool->queue_size == pool->queue_max_size) && (!pool->shutdown))
179     {
180         pthread_cond_wait(&(pool->queue_not_full), &(pool->lock));
181     }
182
183     if (pool->shutdown)
184     {
185         pthread_cond_broadcast(&(pool->queue_not_empty));
186         pthread_mutex_unlock(&(pool->lock));
187         return 0;
188     }
189
190     /* 清空 工作线程 调用的回调函数 的参数arg */
191     if (pool->task_queue[pool->queue_rear].arg != NULL)
192     {
```

```
193             pool->task_queue[pool->queue_rear].arg = NULL;
194         }
195 
196         /*添加任务到任务队列里*/
197         pool->task_queue[pool->queue_rear].function = function;
198         pool->task_queue[pool->queue_rear].arg = arg;
199         pool->queue_rear = (pool->queue_rear + 1) % pool->queue_max_size; /
* 队尾指针移动, 模拟环形 */
200         pool->queue_size++;
201 
202         /*添加完任务后，队列不为空，唤醒线程池中 等待处理任务的线程*/
203         pthread_cond_signal(&(pool->queue_not_empty));
204         pthread_mutex_unlock(&(pool->lock));
205 
206         return 0;
207     }
208 
209     /* 线程池中各个工作线程 */
210     void *threadpool_thread(void *threadpool)
211     {
212         threadpool_t *pool = (threadpool_t *)threadpool;
213         threadpool_task_t task;
214 
215         while (true)
216         {
217             /* Lock must be taken to wait on conditional variable */
218             /*刚创建出线程，等待任务队列里有任务，否则阻塞等待任务队列里有任务后再
唤醒接收任务*/
219             pthread_mutex_lock(&(pool->lock));
220 
221             /*queue_size == 0 说明没有任务，调 wait 阻塞在条件变量上，若有任
务，跳过该while*/
222             while ((pool->queue_size == 0) && (!pool->shutdown))
223             {
224                 printf("thread 0x%x is waiting\n", (unsigned int)pthread_sel
f());
225                 pthread_cond_wait(&(pool->queue_not_empty), &(pool-
>lock)); //暂停到这
226 
227                 /*清除指定数目的空闲线程，如果要结束的线程个数大于0，结束线程*/
228                 if (pool->wait_exit_thr_num > 0)
229                 {
```

```
230                 pool->wait_exit_thr_num--;
231
232                 /*如果线程池里线程个数大于最小值时可以结束当前线程*/
233                 if (pool->live_thr_num > pool->min_thr_num)
234                 {
235                     printf("thread 0x%x is exiting\n", (unsigned int)pth
read_self());
236                     pool->live_thr_num--;
237                     pthread_mutex_unlock(&(pool->lock));
238                     //pthread_detach(pthread_self());
239                     pthread_exit(NULL);
240                 }
241             }
242         }
243
244         /*如果指定了true，要关闭线程池里的每个线程，自行退出处理---销毁线程池
*/
245         if (pool->shutdown)
246         {
247             pthread_mutex_unlock(&(pool->lock));
248             printf("thread 0x%x is exiting\n", (unsigned int)pthread_sel
f());
249             //pthread_detach(pthread_self());
250             pthread_exit(NULL); /* 线程自行结束 */
251         }
252
253         /*从任务队列里获取任务，是一个出队操作*/
254         task.function = pool->task_queue[pool->queue_front].function;
255         task.arg = pool->task_queue[pool->queue_front].arg;
256
257         pool->queue_front = (pool->queue_front + 1) % pool->queue_max_si
ze; /* 出队，模拟环形队列 */
258         pool->queue_size--;
259
260         /*通知可以有新的任务添加进来*/
261         pthread_cond_broadcast(&(pool->queue_not_full));
262
263         /*任务取出后，立即将 线程池琐 释放*/
264         pthread_mutex_unlock(&(pool->lock));
265
266         /*执行任务*/
267         printf("thread 0x%x start working\n", (unsigned int)pthread_self
```

```
268         pthread_mutex_lock(&(pool->thread_counter)); /*忙状态线程数变量琐
*/
269         pool->busy_thr_num++;                       /*忙状态线程数+1*/
270         pthread_mutex_unlock(&(pool->thread_counter));
271
272         (*(task.function))(task.arg); /*执行回调函数任务*/
273         //task.function(task.ar
g);                                         /*执行回调函数任务*/
274
275         /*任务结束处理*/
276         printf("thread 0x%x end working\n", (unsigned int)pthread_self()
277         pthread_mutex_lock(&(pool->thread_counter));
278         pool->busy_thr_num--; /*处理掉一个任务，忙状态数线程数-1*/
279         pthread_mutex_unlock(&(pool->thread_counter));
280     }
281
282     pthread_exit(NULL);
283 }
284
285 /* 管理线程 */
286 void *adjust_thread(void *threadpool)
287 {
288     int i;
289     threadpool_t *pool = (threadpool_t *)threadpool;
290     while (!pool->shutdown)
291     {
292
293         //sleep(DEFAULT_TIME);                                  /*定
时 对线程池管理*/
294         sleep(2);
295         pthread_mutex_lock(&(pool->lock));
296         int queue_size = pool->queue_size;     /* 关注 任务数 */
297         int live_thr_num = pool->live_thr_num; /* 存活 线程数 */
298         pthread_mutex_unlock(&(pool->lock));
299
300         pthread_mutex_lock(&(pool->thread_counter));
301         int busy_thr_num = pool->busy_thr_num; /* 忙着的线程数 */
302         pthread_mutex_unlock(&(pool->thread_counter));
303         printf("--------queue_size=%d------\n", queue_size);
304         /* 创建新线程 算法： 任务数大于最小线程池个数，且存活的线程数少于最大
线程个数时 如：30>=10 && 40<100*/
```

```
305         //if (queue_size >= MIN_WAIT_TASK_NUM && live_thr_num < pool->ma
x_thr_num)
306         if (queue_size >= 4 && live_thr_num < pool->max_thr_num)
307         {
308 
309             pthread_mutex_lock(&(pool->lock));
310             int add = 0;
311 
312             /*一次增加 DEFAULT_THREAD 个线程*/
313             for (i = 0; i < pool->max_thr_num && add < DEFAULT_THREAD_VA
RY && pool->live_thr_num < pool->max_thr_num; i++)
314             {
315                 if (pool->threads[i] == 0 || !is_thread_alive(pool->thre
ads[i]))
316                 {
317                     pthread_create(&(pool->threads[i]), NULL, threadpool
_thread, (void *)pool);
318                     add++;
319                     pool->live_thr_num++;
320                 }
321             }
322 
323             pthread_mutex_unlock(&(pool->lock));
324         }
325 
326         /* 销毁多余的空闲线程 算法：忙线程X2 小于 存活的线程数 且 存活的线程
数 大于 最小线程数时*/
327         if ((busy_thr_num * 2) < live_thr_num && live_thr_num > pool->mi
n_thr_num)
328         {
329             /* 一次销毁DEFAULT_THREAD个线程, 隨機10個即可 */
330             pthread_mutex_lock(&(pool->lock));
331             pool->wait_exit_thr_num = DEFAULT_THREAD_VARY; /* 要销毁的线
程数 设置为10 */
332             pthread_mutex_unlock(&(pool->lock));
333 
334             for (i = 0; i < DEFAULT_THREAD_VARY; i++)
335             {
336                 /* 通知处在空闲状态的线程, 他们会自行终止*/
337                 pthread_cond_signal(&(pool->queue_not_empty));
338             }
339         }
```

```
340     }
341
342     return NULL;
343 }
344
345 int threadpool_destroy(threadpool_t *pool)
346 {
347     int i;
348     if (pool == NULL)
349     {
350         return -1;
351     }
352     pool->shutdown = true;
353
354     /*先销毁管理线程*/
355     //pthread_join(pool->adjust_tid, NULL);
356
357     for (i = 0; i < pool->live_thr_num; i++)
358     {
359         /*通知所有的空闲线程*/
360         pthread_cond_broadcast(&(pool->queue_not_empty));
361     }
362
363     /*for (i = 0; i < pool->live_thr_num; i++)
364     {
365         pthread_join(pool->threads[i], NULL);
366     }*/
367
368     threadpool_free(pool);
369
370     return 0;
371 }
372
373 int threadpool_free(threadpool_t *pool)
374 {
375     if (pool == NULL)
376     {
377         return -1;
378     }
379
380     if (pool->task_queue)
```

```
381     {
382         free(pool->task_queue);
383     }
384 
385     if (pool->threads)
386     {
387         free(pool->threads);
388         pthread_mutex_lock(&(pool->lock));
389         pthread_mutex_destroy(&(pool->lock));
390         pthread_mutex_lock(&(pool->thread_counter));
391         pthread_mutex_destroy(&(pool->thread_counter));
392         pthread_cond_destroy(&(pool->queue_not_empty));
393         pthread_cond_destroy(&(pool->queue_not_full));
394     }
395 
396     free(pool);
397     pool = NULL;
398 
399     return 0;
400 }
401 
402 int threadpool_all_threadnum(threadpool_t *pool)
403 {
404     int all_threadnum = -1;
405 
406     pthread_mutex_lock(&(pool->lock));
407     all_threadnum = pool->live_thr_num;
408     pthread_mutex_unlock(&(pool->lock));
409 
410     return all_threadnum;
411 }
412 
413 int threadpool_busy_threadnum(threadpool_t *pool)
414 {
415     int busy_threadnum = -1;
416 
417     pthread_mutex_lock(&(pool->thread_counter));
418     busy_threadnum = pool->busy_thr_num;
419     pthread_mutex_unlock(&(pool->thread_counter));
420 
421     return busy_threadnum;
```

```
}

int is_thread_alive(pthread_t tid)
{
    int kill_rc = pthread_kill(tid, 0); //发0号信号，测试线程是否存活
    if (kill_rc == ESRCH)
    {
        return false;
    }

    return true;
}

/*测试*/

#if 1
/* 线程池中的线程，模拟处理业务 */
void *deal_client_fun(void *arg)
{
    CLIENT_MSG *p = (CLIENT_MSG *)arg;

    //打印客户端的信息
    char ip[16] = "";
    unsigned short port = 0;
    inet_ntop(AF_INET, &p->addr.sin_addr.s_addr, ip, 16);
    port = ntohs(p->addr.sin_port);
    printf("%s %hu connected\n", ip, port);

    //while获取客户端的请求 并回应
    while (1)
    {
        unsigned char buf[1500] = "";
        int len = recv(p->cfd, buf, sizeof(buf), 0);
        if (len <= 0)
        {
            printf("%s %hu 退出了\n", ip, port);
            close(p->cfd);
            break;
        }
        else
```

```
462         {
463             printf("%s %d:%s\n", ip, port, buf);
464             send(p->cfd, buf, len, 0);
465         }
466     }
467 
468     //释放堆区空间
469     if (p != NULL)
470     {
471         free(p);
472         p = NULL;
473     }
474 
475     return NULL;
476 }
477 int main(int argc, char const *argv[])
478 {
479     if (argc != 2)
480     {
481         printf("./a.out 8000\n");
482         _exit(-1);
483     }
484 
485     //创建监听套接字
486     int lfd = create_tcp_socket(atoi(argv[1]));
487 
488     /*threadpool_t *threadpool_create(int min_thr_num, int max_thr_nu
m, int queue_max_size);*/
489     threadpool_t *thp = threadpool_create(3, 100, 100); /*创建线程池，池
里最小3个线程，最大100，队列最大100*/
490     printf("pool inited");
491 
492     //4、while-->accept提取客户端
493     int count = 0;
494     while (1)
495     {
496         struct sockaddr_in cli_addr;
497         socklen_t cli_len = sizeof(cli_addr);
498         int cfd = accept(lfd, (struct sockaddr *)&cli_addr, &cli_len);
499         count++;
500         CLIENT_MSG *p = (CLIENT_MSG *)calloc(1, sizeof(CLIENT_MSG));
```

```
501             p->cfd = cfd;
502             p->addr = cli_addr;
503 
504             threadpool_add(thp, deal_client_fun, (void *)p); /* 向线程池中添加任务 */
505         }
506 
507         //关闭监听套接字
508         close(lfd);
509 
510         sleep(10); /* 等子线程完成任务 */
511         threadpool_destroy(thp);
512 
513         return 0;
514 }
515 
516 #endif
```