

# 实验五

计科21-2 2021011587 吴维皓

## 题目一

设计思路：

1. 在实验四线程池模型的基础上创建3个独立的线程池，分别是 `readmsg_pool`、`readfile_pool`、`sendmsg_pool`
2. `readmsg_pool` 是用来读取信息的，具体来说就是用来接收客户端响应请求，它对应一个 `readmsg_queue`。每来一个客户端请求，就将这个请求加入 `readmsg_queue` 中，然后 `readmsg_pool` 中的线程从该队列取出任务并执行。具体执行的函数为 `read_msg`。它首先从 `socket` 中将请求信息读取到 `buffer` 中，并对其进行解析（转化回车/换行符、判断请求类型、检查请求的安全性、处理默认请求），最后将解析后的 `buffer` 插入 `filename_queue`（不单是 `buffer`，还有对应的 `socket`）。
3. `filename_queue` 是 `readfile_pool` 的任务队列。线程池中的线程执行对应函数 `read_file`，从任务队列中的结点可以得到解析后的 `buffer` 和其对应的 `socket`，然后 `buffer` 中的文件拓展名确定文件类型，再从以 `buffer[5]` 为起点初打开文件。接下来是构建**HTTP响应头信息**（这里我构建好后直接向对应的 `socket` 发送），最后是从打开的文件中读取文件内容，再将读取到的信息插入 `msg_queue`（不单有文件内容，还有对应 `socket` 和内容的字节数）。
4. `msg_queue` 是 `sendmsg_pool` 的任务队列。线程池中的线程执行相应函数 `send_msg`，可以从队列的结点中得到要发送的文件内容、内容大小和对应的 `socket`，并利用 `write` 函数将内容发送到对应 `socket` 中

(源码在后面与题目二的源码一并给出)

## 题目二

```
rmqueue 当前长度为: 4
rmpool中线程的平均活跃时间: 170.332ms
rmpool中线程的平均阻塞时间: 37810.478ms
rmpool中线程的最高活跃数量: 55
rmpool中线程的最低活跃数量: 2
rmpool中线程的平均活跃数量: 28
```

```
rfqueue 当前长度为: 3
rfpool中线程的平均活跃时间: 261.605ms
rfpool中线程的平均阻塞时间: 42631.795ms
rfpool中线程的最高活跃数量: 48
rfpool中线程的最低活跃数量: 3
rfpool中线程的平均活跃数量: 25
```

```
smqueue 当前长度为: 1
smpool中线程的平均活跃时间: 7537.571ms
smpool中线程的平均阻塞时间: 29723.012ms
smpool中线程的最高活跃数量: 103
smpool中线程的最低活跃数量: 19
smpool中线程的平均活跃数量: 61

smqueue 当前长度为: 1
smpool中线程的平均活跃时间: 7663.370ms
smpool中线程的平均阻塞时间: 30218.931ms
smpool中线程的最高活跃数量: 103
smpool中线程的最低活跃数量: 18
smpool中线程的平均活跃数量: 60

smqueue 当前长度为: 1
smpool中线程的平均活跃时间: 7663.542ms
smpool中线程的平均阻塞时间: 30219.466ms
smpool中线程的最高活跃数量: 103
smpool中线程的最低活跃数量: 18
smpool中线程的平均活跃数量: 60

smqueue 当前长度为: 1
smpool中线程的平均活跃时间: 7664.056ms
smpool中线程的平均阻塞时间: 30221.100ms
smpool中线程的最高活跃数量: 103
smpool中线程的最低活跃数量: 17
smpool中线程的平均活跃数量: 60
```

## 题目三

分析：在题目一关于程序设计的思路中，我提到创建了3个独立的线程池，为了结构性和后面操作的方便，我没有对实验四的线程池相关代码，而是将其重构了3份，每份对应一个独立的线程池，这是代码结构上可能导致性能瓶颈的原因。接下来是逻辑结构，从题目二的监测结果可知：

一、线程池中线程的平均阻塞时间远大于平均活跃时间。首先是为什么阻塞时间那么长，从源码可知，其主要通过 `pthread_cond_wait(&pool->queue.has_jobs->cond, &pool->queue.mutex);` 循环等待队列的信号量，至于阻塞时间比活跃时间长这个结果，主要原因是任务产生速度小于处理速度。回到源码上，虽然对处理web请求进行了业务分离，但整个任务产生的流程仍是串行的，比如我要接收请求才能解析请求，解析请求才能获取文件内容，获取文件内容才能发送数据。这个流程就像一条流水线，想要提高整体性能就必须提高每个环节的性能。可以从“接收请求”这个环节入手，根据源码，每次接收到一个请求，就会往 `readmsg_pool` 的任务队列中加入一个任务，直到这个过程执行结束才继续监听下次请求。这里可以优化成“监听请求”和“添加队列”并行执行，有效提高任务的产生速度。

二、`smqueue` 中任务的数量往往比 `rmqueue` 和 `rfqueue` 多，原因是一次请求可能对应多个文件内容。根据这一结论，可以设置 `sendmsg_pool` 中线程数量多于 `readmsg_pool` 和 `readfile_pool`。从调试结果来看，`readmsg_pool`、`readfile_pool` 和 `sendmsg_pool` 中线程数分别为100，100，200时性能最佳。

- (100, 100, 200)

- [linux1@bogon web]\$ sudo /root/http\_load-12mar2006/http\_load -p 1000 -s 30 url.txt  
362103 fetches, 1000 max parallel, 9.88541e+07 bytes, in 30.0001 seconds  
273 mean bytes/connection  
12070.1 fetches/sec, 3.29513e+06 bytes/sec  
msecs/connect: 63.1184 mean, 7017.42 max, 0.017 min  
msecs/first-response: 6.02434 mean, 1707.54 max, 0.055 min  
HTTP response codes:  
code 200 -- 362103

- (50, 50, 100)

```
code 200 -- 279118
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 1000 -s 30 url.txt
279118 fetches, 1000 max parallel, 7.61992e+07 bytes, in 30 seconds
273 mean bytes/connection
9303.93 fetches/sec, 2.53997e+06 bytes/sec
msecs/connect: 9.97348 mean, 3009.89 max, 0.015 min
msecs/first-response: 2.08287 mean, 400.577 max, 0.06 min
HTTP response codes:
code 200 -- 279118
```

- (100, 100, 100)

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 1000 -s 30 url.txt
281859 fetches, 1000 max parallel, 7.69475e+07 bytes, in 30 seconds
273 mean bytes/connection
9395.29 fetches/sec, 2.56491e+06 bytes/sec
msecs/connect: 8.55179 mean, 3005.24 max, 0.016 min
msecs/first-response: 2.40784 mean, 406.461 max, 0.06 min
HTTP response codes:
code 200 -- 281859
```

- (100, 100, 150)

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 1000 -s 30 url.txt
344214 fetches, 1000 max parallel, 9.39704e+07 bytes, in 30 seconds
273 mean bytes/connection
11473.8 fetches/sec, 3.13235e+06 bytes/sec
msecs/connect: 39.4423 mean, 3018.89 max, 0.015 min
msecs/first-response: 7.03617 mean, 959.485 max, 0.057 min
HTTP response codes:
code 200 -- 344214
```

- (150, 150, 150)

```
http://192.168.88.132:8016/index.html: byte count wrong
http://192.168.88.132:8016/index.html: byte count wrong
274691 fetches, 1000 max parallel, 7.49076e+07 bytes, in 30.0007 seconds
272.698 mean bytes/connection
9156.14 fetches/sec, 2.49686e+06 bytes/sec
msecs/connect: 54.3821 mean, 7020.38 max, 0.017 min
msecs/first-response: 12.8313 mean, 16449.6 max, 0.068 min
304 bad byte counts
HTTP response codes:
code 200 -- 274387
```

- (100, 100, 250)

- [linux1@bogon web]\$ sudo /root/http\_load-12mar2006/http\_load -p 1000 -s 30 url.txt  
356801 fetches, 1000 max parallel, 9.74067e+07 bytes, in 30 seconds  
273 mean bytes/connection  
11893.4 fetches/sec, 3.24688e+06 bytes/sec  
msecs/connect: 64.2849 mean, 7029.44 max, 0.016 min  
msecs/first-response: 5.87905 mean, 577.564 max, 0.04 min  
HTTP response codes:  
code 200 -- 356801

- (150, 150, 200)

- [linux1@bogon web]\$ sudo /root/http\_load-12mar2006/http\_load -p 1000 -s 30 url.txt  
356039 fetches, 1000 max parallel, 9.71986e+07 bytes, in 30.007 seconds  
273 mean bytes/connection  
11865.2 fetches/sec, 3.2392e+06 bytes/sec  
msecs/connect: 61.7635 mean, 7021.23 max, 0.02 min  
msecs/first-response: 6.98114 mean, 802.701 max, 0.052 min  
HTTP response codes:  
code 200 -- 356039

- (150, 150, 250)

- [linux1@bogon web]\$ sudo /root/http\_load-12mar2006/http\_load -p 1000 -s 30 url.txt  
302097 fetches, 1000 max parallel, 8.24725e+07 bytes, in 30 seconds  
273 mean bytes/connection  
10069.9 fetches/sec, 2.74908e+06 bytes/sec  
msecs/connect: 73.4402 mean, 15041 max, 0.017 min  
msecs/first-response: 8.33653 mean, 6483.56 max, 0.062 min  
HTTP response codes:  
code 200 -- 302097

```

1 //业务分离 + 性能监测
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <string.h>
6 #include <errno.h>
7 #include <string.h>
8 #include <fcntl.h>
9 #include <signal.h>
10 #include <sys/types.h>
11 #include <sys/socket.h>
12 #include <netinet/in.h>
13 #include <arpa/inet.h>
14 #include <pthread.h>
15 #include <sys/stat.h>
16 #include <sys/prctl.h>
17 #include <stdbool.h>
18 #include <sys/time.h>
19 #include "head.h"
20
21 #define VERSION 23

```

```

22 #define BUFSIZE 8096
23 #define ERROR 42
24 #define LOG 44
25 #define FORBIDDEN 403
26 #define NOTFOUND 404
27 #ifndef SIGCLD
28 #define SIGCLD SIGCHLD
29 #endif
30
31 struct
32 {
33     char *ext;
34     char *filetype;
35 } extensions[] = {
36     {"gif", "image/gif"},
37     {"jpg", "image/jpg"},
38     {"jpeg", "image/jpeg"},
39     {"png", "image/png"},
40     {"ico", "image/ico"},
41     {"zip", "image/zip"},
42     {"gz", "image/gz"},
43     {"tar", "image/tar"},
44     {"htm", "text/html"},
45     {"html", "text/html"},
46     {0, 0}};
47
48 readmsgpool *readmsg_pool;
49 readfilepool *readfile_pool;
50 sendmsgpool *sendmsg_pool;
51
52 int max_rmth, max_rfth, max_smth;
53 int min_rmth = 1000, min_rfth = 1000, min_smth = 1000; // 设一个比线程池容量
    大的数
54 double act_rm, act_rf, act_sm; // 各个线程池中线程的
    总活跃时间
55 double blc_rm, blc_rf, blc_sm;
56
57 inline int max(int x, int y)
58 {
59     return x > y ? x : y;
60 }
61
62 inline int min(int x, int y)
63 {
64     return x < y ? x : y;
65 }
66
67 void push_rmsgqueue(readmsg_queue *queue, readmsg_node *newtask)
68 {
69     readmsg_node *node = (readmsg_node *)malloc(sizeof(readmsg_node));
70     if (!node)
71     {
72         perror("Error allocating memory for new task");
73         exit(EXIT_FAILURE);
74     }

```

```

75
76 // 将任务信息复制到新节点
77 node->function = newtask->function;
78 node->arg = newtask->arg;
79 node->next = NULL;
80
81 // 加锁, 修改任务队列
82 pthread_mutex_lock(&queue->mutex);
83
84 if (queue->len == 0)
85 {
86     // 队列为空, 直接添加新任务
87     queue->front = node;
88     queue->rear = node;
89 }
90 else
91 {
92     // 否则将新任务添加到队列尾部
93     queue->rear->next = node;
94     queue->rear = node;
95 }
96 queue->len++;
97
98 // 通知等待在队列上的线程, 有新任务到来
99 pthread_cond_signal(&queue->has_jobs->cond);
100
101 pthread_mutex_unlock(&queue->mutex);
102 }
103
104 void init_rdmqueue(readmsg_queue *queue)
105 {
106     // 初始化互斥量(互斥访问任务队列)
107     pthread_mutex_init(&queue->mutex, NULL);
108
109     // 初始化条件变量(在队列为空时阻塞等待任务的到来)
110     queue->has_jobs = (rmstaconv *)malloc(sizeof(rmstaconv));
111     pthread_cond_init(&queue->has_jobs->cond, NULL);
112
113     // 初始化任务队列
114     queue->front = NULL;
115     queue->rear = NULL;
116     queue->len = 0;
117 }
118
119 readmsg_node *take_rdmqueue(readmsg_queue *queue) // 取出队首任务, 并在队列
120 // 中删除该任务
121 {
122     // 加锁, 访问任务队列
123     pthread_mutex_lock(&queue->mutex);
124
125     // 如果队列为空, 等待任务到来
126     while (queue->len == 0)
127     {
128         pthread_cond_wait(&queue->has_jobs->cond, &queue->mutex);
129     }

```

```

129
130     readmsg_node *curtask = queue->front; // 取出队首任务
131     queue->front = curtask->next;          // 更新队列头指针,指向下一个任务
132     if (queue->len == 1)                   // 如果队列只有一个任务,更新尾指针
133     {
134         queue->rear = NULL;
135     }
136     queue->len--;
137
138     // 解锁互斥量
139     pthread_mutex_unlock(&queue->mutex);
140
141     return curtask;
142 }
143
144 void destroy_rdmqueue(readmsg_queue *queue)
145 {
146
147     pthread_mutex_lock(&queue->mutex); // 加锁,访问任务队列
148
149     while (queue->front != NULL) // 释放队列中所有任务节点
150     {
151         readmsg_node *curtask = queue->front;
152         queue->front = curtask->next;
153         free(curtask);
154     }
155     free(queue->has_jobs); // 释放条件变量
156
157     pthread_mutex_unlock(&queue->mutex);
158
159     pthread_mutex_destroy(&queue->mutex); // 销毁互斥量
160 }
161
162 struct readmsgpool *initreadmsgpool(int num_threads)
163 {
164     readmsgpool *pool;
165     pool = (readmsgpool *)malloc(sizeof(struct readmsgpool));
166     pool->num_threads = 0;
167     pool->num_working = 0;
168     pool->is_alive = 1;
169     pthread_mutex_init(&(pool->thcount_lock), NULL);
170     // 初始化互斥量
171     pthread_cond_init(&(pool->thread_all_idle), NULL);
172     // 初始化条件变量
173     init_rdmqueue(&pool->queue);
174     // 初始化任务队列@
175     pool->threads = (struct rmthread **)malloc(num_threads *
176     sizeof(struct rmthread *)); // 创建线程数组
177
178     int i;
179     for (i = 0; i < num_threads; i++)
180     {
181         create_rmthread(pool, &pool->threads[i], i); // 在pool->threads[i]
182         前加了个&
183     }

```

```

179 // 每个线程在创建时,运行函数都会进行pool->num_threads++操作
180 while (pool->num_threads != num_threads) // 忙等待,等所有进程创建完毕才返回
181 {
182 }
183 return pool;
184 }
185
186 void waitreadmsgpool(readmsgpool *pool)
187 {
188     pthread_mutex_lock(&pool->thcount_lock);
189     while (pool->queue.len || pool->num_working) // 这里可能有问题?
190     {
191         pthread_cond_wait(&pool->thread_all_idle, &pool->thcount_lock);
192     }
193     pthread_mutex_unlock(&pool->thcount_lock);
194 }
195
196 void destoryreadmsgpool(readmsgpool *pool)
197 {
198     // 等待线程执行完任务队列中的所有任务,并且任务队列为空 @
199     waitreadmsgpool(pool);
200     pool->is_alive = 0; // 关闭线程池运行
201     pthread_cond_broadcast(&pool->queue.has_jobs->cond); // 唤醒所有等待在任务队列上的线程(让它们检查 is_alive 的状态并退出)
202
203     destory_rdmqueue(&pool->queue); // 销毁任务队列 @
204
205     // 销毁线程指针数组,并释放为线程池分配的内存 @
206     int i;
207     for (i = 0; i < pool->num_threads; i++)
208     {
209         free(pool->threads[i]);
210     }
211     free(pool->threads);
212
213     // 销毁线程池的互斥量和条件变量
214     pthread_mutex_destroy(&pool->thcount_lock);
215     pthread_cond_destroy(&pool->thread_all_idle);
216
217     // 释放线程池结构体内存
218     free(pool);
219 }
220
221 void *rmthread_do(struct rmthread *pthread)
222 {
223     // 设置线程名称
224     char thread_name[128] = {0};
225     sprintf(thread_name, "thread-pool-%d", pthread->id);
226
227     prctl(PR_SET_NAME, thread_name);
228
229     // 获得/绑定线程池
230     readmsgpool *pool = pthread->pool;
231

```



```

232 pthread_mutex_lock(&pool->thcount_lock);
233 pool->num_threads++; // 对创建线程数量进行统计@
234 pthread_mutex_unlock(&pool->thcount_lock);
235
236 while (pool->is_alive)
237 {
238     // 如果队列中还有任务，则继续运行；否则阻塞 @
239     struct timeval start1, end1;
240     gettimeofday(&start1, NULL);
241     pthread_mutex_lock(&pool->queue.mutex);
242     while (pool->queue.len == 0 && pool->is_alive)
243     {
244         pthread_cond_wait(&pool->queue.has_jobs->cond, &pool->
245 >queue.mutex);
246     }
247     pthread_mutex_unlock(&pool->queue.mutex);
248     gettimeofday(&end1, NULL);
249
250     if (pool->is_alive)
251     {
252         pthread_mutex_lock(&pool->thcount_lock);
253         pool->num_working++; // 对工作线程数量进行统计@
254         pthread_mutex_unlock(&pool->thcount_lock);
255
256         // 取任务队列队首，并执行
257         int aveth, max_, min_; // 设置局部变量保证线程安全
258         double totala, totalb;
259         struct timeval start2, end2;
260         void *(*func)(void *);
261         void *arg;
262         readmsg_node *curtask = take_rdmqueue(&pool->queue); // 取出
263         // 队首任务，并在队列中删除该任务@（自己实现take_taskqueue）
264         if (curtask)
265         {
266             func = curtask->function;
267             arg = curtask->arg;
268             gettimeofday(&start2, NULL);
269             func(arg); // 执行任务
270             gettimeofday(&end2, NULL);
271
272             act_rm += (end2.tv_sec - start2.tv_sec) * 1000.0 +
273 (end2.tv_usec - start2.tv_usec) / 1000.0;
274             totala = act_rm; // 保证下面打印时输出的是当前的时间
275             b1c_rm += (end1.tv_sec - start1.tv_sec) * 1000.0 +
276 (end1.tv_usec - start1.tv_usec) / 1000.0;
277             totalb = b1c_rm;
278             max_rmth = max(max_rmth, pool->num_working);
279             max_ = max_rmth;
280             min_rmth = min(min_rmth, pool->num_working);
281             min_ = min_rmth;
282             aveth = (max_ + min_) / 2;
283
284             pthread_mutex_lock(&pool->thcount_lock);
285             printf("\nrmqueue 当前长度为: %d\n", pool->queue.len + 1);
286             // 加上被取出的1个

```

```

282         printf("rmpool中线程的平均活跃时间: %.3fms\n", totala /
aveth);
283         printf("rmpool中线程的平均阻塞时间: %.3fms\n", totalb /
aveth);
284         printf("rmpool中线程的最高活跃数量: %d\n", max_);
285         printf("rmpool中线程的最低活跃数量: %d\n", min_);
286         printf("rmpool中线程的平均活跃数量: %d\n", aveth);
287         pthread_mutex_unlock(&pool->thcount_lock);
288
289         free(arg);        // 释放参数
290         free(curtask);    // 释放任务
291     }
292     pthread_mutex_lock(&pool->thcount_lock);
293     pool->num_working--;
294     pthread_mutex_unlock(&pool->thcount_lock);
295     // 当工作线程数量为0时, 表示任务全部完成, 此时运行阻塞在
waitreadmsgpool上的线程 @
296     if (pool->num_working == 0 && pool->queue.len == 0)
297         pthread_cond_signal(&pool->thread_all_idle);
298     }
299 }
300
301 // 线程执行完任务将要退出, 需改变线程池中的线程数量 @
302 pthread_mutex_unlock(&pool->thcount_lock);
303 pool->num_threads--;
304 pthread_mutex_unlock(&pool->thcount_lock);
305 return NULL;
306 }
307
308 int create_rmthread(struct readmsgpool *pool, struct rmthread **pthread,
int id)
309 {
310     *pthread = (struct rmthread *)malloc(sizeof(struct rmthread));
311     if (*pthread == NULL)
312     {
313         perror("create_thread(): could not allocate memory for
thread\n");
314         return -1;
315     }
316
317     // 设置该线程的属性
318     (*pthread)->pool = pool;
319     (*pthread)->id = id;
320
321     pthread_create(&(*pthread)->pthread, NULL, (void *)rmthread_do,
(*pthread)); // 创建线程
322     pthread_detach((*pthread)->pthread);
// 线程分离
323     return 0;
324 }
325
326 void push_fnamequeue(filename_queue *queue, filename_node *newtask)
327 {
328     filename_node *node = (filename_node *)malloc(sizeof(filename_node));
329     if (!node)

```

```

330     {
331         perror("Error allocating memory for new task");
332         exit(EXIT_FAILURE);
333     }
334
335     // 将任务信息复制到新节点
336     node->function = newtask->function;
337     node->arg = newtask->arg;
338     node->next = NULL;
339
340     // 加锁，修改任务队列
341     pthread_mutex_lock(&queue->mutex);
342
343     if (queue->len == 0)
344     {
345         // 队列为空，直接添加新任务
346         queue->front = node;
347         queue->rear = node;
348     }
349     else
350     {
351         // 否则将新任务添加到队列尾部
352         queue->rear->next = node;
353         queue->rear = node;
354     }
355     queue->len++;
356
357     // 通知等待在队列上的线程，有新任务到来
358     pthread_cond_signal(&queue->has_jobs->cond);
359
360     pthread_mutex_unlock(&queue->mutex);
361 }
362
363 void init_fnamequeue(filename_queue *queue)
364 {
365     // 初始化互斥量(互斥访问任务队列)
366     pthread_mutex_init(&queue->mutex, NULL);
367
368     // 初始化条件变量(在队列为空时阻塞等待任务的到来)
369     queue->has_jobs = (rfstaconv *)malloc(sizeof(rfstaconv));
370     pthread_cond_init(&queue->has_jobs->cond, NULL);
371
372     // 初始化任务队列
373     queue->front = NULL;
374     queue->rear = NULL;
375     queue->len = 0;
376 }
377
378 filename_node *take_fnamequeue(filename_queue *queue) // 取出队首任务，并在队
列中删除该任务
379 {
380     // 加锁，访问任务队列
381     pthread_mutex_lock(&queue->mutex);
382
383     // 如果队列为空，等待任务到来

```

```

384     while (queue->len == 0)
385     {
386         pthread_cond_wait(&queue->has_jobs->cond, &queue->mutex);
387     }
388
389     filename_node *curtask = queue->front; // 取出队首任务
390     queue->front = curtask->next;          // 更新队列头指针,指向下一个任务
391     if (queue->len == 1)                  // 如果队列只有一个任务,更新尾指针
392     {
393         queue->rear = NULL;
394     }
395     queue->len--;
396
397     // 解锁互斥量
398     pthread_mutex_unlock(&queue->mutex);
399
400     return curtask;
401 }
402
403 void destroy_fnamequeue(filename_queue *queue)
404 {
405
406     pthread_mutex_lock(&queue->mutex); // 加锁,访问任务队列
407
408     while (queue->front != NULL) // 释放队列中所有任务节点
409     {
410         filename_node *curtask = queue->front;
411         queue->front = curtask->next;
412         free(curtask);
413     }
414     free(queue->has_jobs); // 释放条件变量
415
416     pthread_mutex_unlock(&queue->mutex);
417
418     pthread_mutex_destroy(&queue->mutex); // 销毁互斥量
419 }
420
421 struct readfilepool *initreadfilepool(int num_threads)
422 {
423     readfilepool *pool;
424     pool = (readfilepool *)malloc(sizeof(struct readfilepool));
425     pool->num_threads = 0;
426     pool->num_working = 0;
427     pool->is_alive = 1;
428     pthread_mutex_init(&(pool->thcount_lock), NULL);
429     // 初始化互斥量
430     pthread_cond_init(&(pool->thread_all_idle), NULL);
431     // 初始化条件变量
432     init_fnamequeue(&pool->queue);
433     // 初始化任务队列@
434     pool->threads = (struct rfthread **)malloc(num_threads *
sizeof(struct rfthread *)); // 创建线程数组

```

```

435     {
436         create_rfthread(pool, &pool->threads[i], i); // 在pool->threads[i]
前加了个&
437     }
438     // 每个线程在创建时,运行函数都会进行pool->num_threads++操作
439     while (pool->num_threads != num_threads) // 忙等待, 等所有进程创建完毕才返回
440     {
441     }
442     return pool;
443 }
444
445 void waitreadfilepool(readfilepool *pool)
446 {
447     pthread_mutex_lock(&pool->thcount_lock);
448     while (pool->queue.len || pool->num_working) // 这里可能有问题?
449     {
450         pthread_cond_wait(&pool->thread_all_idle, &pool->thcount_lock);
451     }
452     pthread_mutex_unlock(&pool->thcount_lock);
453 }
454
455 void destoryreadfilepool(readfilepool *pool)
456 {
457     // 等待线程执行完任务队列中的所有任务, 并且任务队列为空 @
458     waitreadfilepool(pool);
459     pool->is_alive = 0; // 关闭线程池运行
460     pthread_cond_broadcast(&pool->queue.has_jobs->cond); // 唤醒所有等待在任务队列上的线程(让它们检查 is_alive 的状态并退出)
461
462     destory_fnamequeue(&pool->queue); // 销毁任务队列 @
463
464     // 销毁线程指针数组, 并释放为线程池分配的内存 @
465     int i;
466     for (i = 0; i < pool->num_threads; i++)
467     {
468         free(pool->threads[i]);
469     }
470     free(pool->threads);
471
472     // 销毁线程池的互斥量和条件变量
473     pthread_mutex_destroy(&pool->thcount_lock);
474     pthread_cond_destroy(&pool->thread_all_idle);
475
476     // 释放线程池结构体内存
477     free(pool);
478 }
479
480 void *rfthread_do(struct rfthread *pthread)
481 {
482     // 设置线程名称
483     char thread_name[128] = {0};
484     sprintf(thread_name, "thread-pool-%d", pthread->id);
485
486     prctl(PR_SET_NAME, thread_name);

```

```

487
488 // 获得/绑定线程池
489 readfilepool *pool = pthread->pool;
490
491 pthread_mutex_lock(&pool->thcount_lock);
492 pool->num_threads++; // 对创建线程数量进程统计@
493 pthread_mutex_unlock(&pool->thcount_lock);
494
495 while (pool->is_alive)
496 {
497     // 如果队列中还有任务，则继续运行；否则阻塞 @
498     struct timeval start1, end1;
499     gettimeofday(&start1, NULL);
500     pthread_mutex_lock(&pool->queue.mutex);
501     while (pool->queue.len == 0 && pool->is_alive)
502     {
503         pthread_cond_wait(&pool->queue.has_jobs->cond, &pool->
504 >queue.mutex);
505     }
506     pthread_mutex_unlock(&pool->queue.mutex);
507     gettimeofday(&end1, NULL);
508
509     if (pool->is_alive)
510     {
511         pthread_mutex_lock(&pool->thcount_lock);
512         pool->num_working++; // 对工作线程数量进行统计@
513         pthread_mutex_unlock(&pool->thcount_lock);
514
515         // 取任务队列队首，并执行
516         int aveth, max_, min_;
517         double totala, totalb;
518         struct timeval start2, end2;
519         void *(*func)(void *);
520         void *arg;
521         filename_node *curtask = take_fnamequeue(&pool->queue); // 取
522 出队首任务，并在队列中删除该任务@（自己实现take_fnamequeue）
523         if (curtask)
524         {
525             func = curtask->function;
526             arg = curtask->arg;
527             gettimeofday(&start2, NULL);
528             func(arg); // 执行任务
529             gettimeofday(&end2, NULL);
530
531             act_rf += (end2.tv_sec - start2.tv_sec) * 1000.0 +
532 (end2.tv_usec - start2.tv_usec) / 1000.0;
533             totala = act_rf;
534             blc_rf += (end1.tv_sec - start1.tv_sec) * 1000.0 +
535 (end1.tv_usec - start1.tv_usec) / 1000.0;
536             totalb = blc_rf;
537             max_rfth = max(max_rfth, pool->num_working);
538             max_ = max_rfth;
539             min_rfth = min(min_rfth, pool->num_working);
540             min_ = min_rfth;
541             aveth = (max_ + min_) / 2;

```

```

538
539         pthread_mutex_lock(&pool->thcount_lock);
540         printf("\nrffqueue 当前长度为: %d\n", pool->queue.len + 1);
// 加上被取出的1个
541         printf("rffpool中线程的平均活跃时间: %.3fms\n", totala /
aveth);
542         printf("rffpool中线程的平均阻塞时间: %.3fms\n", totalb /
aveth);
543         printf("rffpool中线程的最高活跃数量: %d\n", max_);
544         printf("rffpool中线程的最低活跃数量: %d\n", min_);
545         printf("rffpool中线程的平均活跃数量: %d\n", aveth);
546         pthread_mutex_unlock(&pool->thcount_lock);
547
548         free(arg); // 释放参数
549         free(curtask); // 释放任务
550     }
551     pthread_mutex_lock(&pool->thcount_lock);
552     pool->num_working--;
553     pthread_mutex_unlock(&pool->thcount_lock);
554     // 当工作线程数量为0时, 表示任务全部完成, 此时运行阻塞在
waitreadfilepool上的线程 @
555     if (pool->num_working == 0 && pool->queue.len == 0)
556         pthread_cond_signal(&pool->thread_all_idle);
557     }
558 }
559
560 // 线程执行完任务将要退出, 需改变线程池中的线程数量 @
561 pthread_mutex_unlock(&pool->thcount_lock);
562 pool->num_threads--;
563 pthread_mutex_unlock(&pool->thcount_lock);
564 return NULL;
565 }
566
567 int create_rfthread(struct readfilepool *pool, struct rfthread **pthread,
int id)
568 {
569     *pthread = (struct rfthread *)malloc(sizeof(struct rfthread));
570     if (*pthread == NULL)
571     {
572         perror("create_thread(): Could not allocate memory for
thread\n");
573         return -1;
574     }
575
576     // 设置该线程的属性
577     (*pthread)->pool = pool;
578     (*pthread)->id = id;
579
580     pthread_create(&(*pthread)->pthread, NULL, (void *)rfthread_do,
(*pthread)); // 创建线程
581     pthread_detach((*pthread)->pthread);
// 线程分离
582     return 0;
583 }
584

```

```

585 void push_msgqueue(msg_queue *queue, msg_node *newtask)
586 {
587     msg_node *node = (msg_node *)malloc(sizeof(msg_node));
588     if (!node)
589     {
590         perror("Error allocating memory for new task");
591         exit(EXIT_FAILURE);
592     }
593
594     // 将任务信息复制到新节点
595     node->function = newtask->function;
596     node->arg = newtask->arg;
597     node->next = NULL;
598
599     // 加锁, 修改任务队列
600     pthread_mutex_lock(&queue->mutex);
601
602     if (queue->len == 0)
603     {
604         // 队列为空, 直接添加新任务
605         queue->front = node;
606         queue->rear = node;
607     }
608     else
609     {
610         // 否则将新任务添加到队列尾部
611         queue->rear->next = node;
612         queue->rear = node;
613     }
614     queue->len++;
615
616     // 通知等待在队列上的线程, 有新任务到来
617     pthread_cond_signal(&queue->has_jobs->cond);
618
619     pthread_mutex_unlock(&queue->mutex);
620 }
621
622 void init_msgqueue(msg_queue *queue)
623 {
624     // 初始化互斥量(互斥访问任务队列)
625     pthread_mutex_init(&queue->mutex, NULL);
626
627     // 初始化条件变量(在队列为空时阻塞等待任务的到来)
628     queue->has_jobs = (smstaconv *)malloc(sizeof(smstaconv));
629     pthread_cond_init(&queue->has_jobs->cond, NULL);
630
631     // 初始化任务队列
632     queue->front = NULL;
633     queue->rear = NULL;
634     queue->len = 0;
635 }
636
637 msg_node *take_msgqueue(msg_queue *queue) // 取出队首任务, 并在队列中删除该任务
638 {
639     // 加锁, 访问任务队列

```



```

640 pthread_mutex_lock(&queue->mutex);
641
642 // 如果队列为空，等待任务到来
643 while (queue->len == 0)
644 {
645     pthread_cond_wait(&queue->has_jobs->cond, &queue->mutex);
646 }
647
648 msg_node *curtask = queue->front; // 取出队首任务
649 queue->front = curtask->next;      // 更新队列头指针，指向下一个任务
650 if (queue->len == 1)               // 如果队列只有一个任务，更新尾指针
651 {
652     queue->rear = NULL;
653 }
654 queue->len--;
655
656 // 解锁互斥量
657 pthread_mutex_unlock(&queue->mutex);
658
659 return curtask;
660 }
661
662 void destroy_msgqueue(msg_queue *queue)
663 {
664
665     pthread_mutex_lock(&queue->mutex); // 加锁，访问任务队列
666
667     while (queue->front != NULL) // 释放队列中所有任务节点
668     {
669         msg_node *curtask = queue->front;
670         queue->front = curtask->next;
671         free(curtask);
672     }
673     free(queue->has_jobs); // 释放条件变量
674
675     pthread_mutex_unlock(&queue->mutex);
676
677     pthread_mutex_destroy(&queue->mutex); // 销毁互斥量
678 }
679
680 struct sendmsgpool *init_sendmsgpool(int num_threads)
681 {
682     sendmsgpool *pool;
683     pool = (sendmsgpool *)malloc(sizeof(struct sendmsgpool));
684     pool->num_threads = 0;
685     pool->num_working = 0;
686     pool->is_alive = 1;
687     pthread_mutex_init(&(pool->thcount_lock), NULL);
688     // 初始化互斥量
689     pthread_cond_init(&(pool->thread_all_idle), NULL);
690     // 初始化条件变量
691     init_msgqueue(&pool->queue);
692     // 初始化任务队列@
693     pool->threads = (struct smthread **)malloc(num_threads *
        sizeof(struct smthread *)); // 创建线程数组

```

```

691
692     int i;
693     for (i = 0; i < num_threads; i++)
694     {
695         create_smthread(pool, &pool->threads[i], i); // 在pool->threads[i]
前加了个&
696     }
697     // 每个线程在创建时,运行函数都会进行pool->num_threads++操作
698     while (pool->num_threads != num_threads) // 忙等待, 等所有进程创建完毕才返
回
699     {
700     }
701     return pool;
702 }
703
704 void waitsendmsgpool(sendmsgpool *pool)
705 {
706     pthread_mutex_lock(&pool->thcount_lock);
707     while (pool->queue.len || pool->num_working) // 这里可能有问题?
708     {
709         pthread_cond_wait(&pool->thread_all_idle, &pool->thcount_lock);
710     }
711     pthread_mutex_unlock(&pool->thcount_lock);
712 }
713
714 void destorysendmsgpool(sendmsgpool *pool)
715 {
716     // 等待线程执行完任务队列中的所有任务, 并且任务队列为空 @
717     waitsendmsgpool(pool);
718     pool->is_alive = 0; // 关闭线程池运行
719     pthread_cond_broadcast(&pool->queue.has_jobs->cond); // 唤醒所有等待在任
务队列上的线程(让它们检查 is_alive 的状态并退出)
720
721     destory_msgqueue(&pool->queue); // 销毁任务队列 @
722
723     // 销毁线程指针数组, 并释放为线程池分配的内存 @
724     int i;
725     for (i = 0; i < pool->num_threads; i++)
726     {
727         free(pool->threads[i]);
728     }
729     free(pool->threads);
730
731     // 销毁线程池的互斥量和条件变量
732     pthread_mutex_destroy(&pool->thcount_lock);
733     pthread_cond_destroy(&pool->thread_all_idle);
734
735     // 释放线程池结构体内存
736     free(pool);
737 }
738
739 void *smthread_do(struct smthread *pthread)
740 {
741     // 设置线程名称
742     char thread_name[128] = {0};

```

```

743     sprintf(thread_name, "thread-pool-%d", pthread->id);
744
745     prctl(PR_SET_NAME, thread_name);
746
747     // 获得/绑定线程池
748     sendmsgpool *pool = pthread->pool;
749
750     pthread_mutex_lock(&pool->thcount_lock);
751     pool->num_threads++; // 对创建线程数量进行统计@
752     pthread_mutex_unlock(&pool->thcount_lock);
753
754     while (pool->is_alive)
755     {
756         // 如果队列中还有任务，则继续运行；否则阻塞 @
757         struct timeval start1, end1;
758         gettimeofday(&start1, NULL);
759         pthread_mutex_lock(&pool->queue.mutex);
760         while (pool->queue.len == 0 && pool->is_alive)
761         {
762             pthread_cond_wait(&pool->queue.has_jobs->cond, &pool->
763 >queue.mutex);
764         }
765         pthread_mutex_unlock(&pool->queue.mutex);
766         gettimeofday(&end1, NULL);
767
768         if (pool->is_alive)
769         {
770             pthread_mutex_lock(&pool->thcount_lock);
771             pool->num_working++; // 对工作线程数量进行统计@
772             pthread_mutex_unlock(&pool->thcount_lock);
773
774             // 取任务队列队首，并执行
775             int aveth, max_, min_;
776             double totala, totalb;
777             struct timeval start2, end2;
778             void *(*func)(void *);
779             void *arg;
780             msg_node *curtask = take_msgqueue(&pool->queue); // 取出队首任
781             务，并在队列中删除该任务@（自己实现take_msgqueue）
782             if (curtask)
783             {
784                 func = curtask->function;
785                 arg = curtask->arg;
786                 gettimeofday(&start2, NULL);
787                 func(arg); // 执行任务
788                 gettimeofday(&end2, NULL);
789
790                 act_sm += (end2.tv_sec - start2.tv_sec) * 1000.0 +
791 (end2.tv_usec - start2.tv_usec) / 1000.0;
792                 totala = act_sm;
793                 b1c_sm += (end1.tv_sec - start1.tv_sec) * 1000.0 +
794 (end1.tv_usec - start1.tv_usec) / 1000.0;
795                 totalb = b1c_sm;
796                 max_smth = max(max_smth, pool->num_working);
797                 max_ = max_smth;

```

```

794         min_smth = min(min_smth, pool->num_working);
795         min_ = min_smth;
796         aveth = (max_ + min_) / 2;
797         pthread_mutex_lock(&pool->thcount_lock);
798         printf("\nsmqueue 当前长度为: %d\n", pool->queue.len + 1);
799         printf("smpool中线程的平均活跃时间: %.3fms\n", totala /
aveth);
800         printf("smpool中线程的平均阻塞时间: %.3fms\n", totalb /
aveth);
801         printf("smpool中线程的最高活跃数量: %d\n", max_);
802         printf("smpool中线程的最低活跃数量: %d\n", min_);
803         printf("smpool中线程的平均活跃数量: %d\n", aveth);
804         pthread_mutex_unlock(&pool->thcount_lock);
805         free(curtask); // 释放任务
806     }
807     pthread_mutex_lock(&pool->thcount_lock);
808     pool->num_working--;
809     pthread_mutex_unlock(&pool->thcount_lock);
810     // 当工作线程数量为0时, 表示任务全部完成, 此时运行阻塞在waitThreadPool
上的线程 @
811     if (pool->num_working == 0 && pool->queue.len == 0)
812         pthread_cond_signal(&pool->thread_all_idle);
813     }
814 }
815
816 // 线程执行完任务将要退出, 需改变线程池中的线程数量 @
817 pthread_mutex_unlock(&pool->thcount_lock);
818 pool->num_threads--;
819 pthread_mutex_unlock(&pool->thcount_lock);
820 return NULL;
821 }
822
823 int create_smthread(struct sendmsgpool *pool, struct smthread **pthread,
int id)
824 {
825     *pthread = (struct smthread *)malloc(sizeof(struct smthread));
826     if (*pthread == NULL)
827     {
828         perror("create_thread(): could not allocate memory for
thread\n");
829         return -1;
830     }
831
832     // 设置该线程的属性
833     (*pthread)->pool = pool;
834     (*pthread)->id = id;
835
836     pthread_create(&(*pthread)->pthread, NULL, (void *)smthread_do,
(*pthread)); // 创建线程
837     pthread_detach((*pthread)->pthread);
838     // 线程分离
839     return 0;
840 }
841 void logger(int type, char *s1, char *s2, int socket_fd)

```

```

842 {
843     ...
844 }
845
846 void *read_msg(void *data)
847 {
848     webparam *param1 = (webparam *)data;
849     int fd = param1->fd, hit = param1->hit;
850     int j;
851     long i, ret;
852     char buffer[BUFSIZE + 1]; /* 缓存 */
853
854     ret = read(fd, buffer, BUFSIZE); // 从socket 读取 web 请求内容(读socket)
855     if (ret == 0 || ret == -1)
856     { /* 读取失败 */
857         logger(FORBIDDEN, "failed to read browser request", "", fd);
858     }
859     else
860     {
861         if (ret > 0 && ret < BUFSIZE) // 确保读取到的数据以 null 字符 ('\0')
            结尾
862             buffer[ret] = 0;
863         else
864             buffer[0] = 0;
865         for (i = 0; i < ret; i++)
866             if (buffer[i] == '\r' || buffer[i] == '\n')
867                 buffer[i] = '*';
868         logger(LOG, "request", buffer, hit);
869         if (strncmp(buffer, "GET ", 4) != 0)
870         {
871             logger(FORBIDDEN, "only simple get operation supported",
buffer, fd);
872         }
873         for (i = 4; i < BUFSIZE; i++)
874         {
875             if (buffer[i] == ' ')
876             {
877                 buffer[i] = 0;
878                 break;
879             }
880         }
881         for (j = 0; j < i - 1; j++)
882             if (buffer[j] == '.' && buffer[j + 1] == '.')
883             {
884                 logger(FORBIDDEN, "parent directory (..) path names not
supported", buffer, fd);
885             }
886         if (!strncmp(&buffer[0], "GET /\0", 6))
887             (void)strcpy(buffer, "GET /index. html");
888
889         filename_node *curtask = (filename_node
*)malloc(sizeof(filename_node));
890         curtask->next = NULL;
891         curtask->function = read_file;
892

```

```

893     webparam *param2 = (webparam *)malloc(sizeof(webparam));
894     param2->hit = hit;
895     param2->fd = fd;
896     strcpy(param2->buffer, buffer);
897     curtask->arg = (void *)param2;
898     push_fnamequeue(&readfile_pool->queue, curtask);
899 }
900 return NULL;
901 }
902
903 void *read_file(void *data)
904 {
905     webparam *param1 = (webparam *)data;
906     int fd = param1->fd, hit = param1->hit;
907     int file_fd, buflen;
908     long i, ret, len;
909     char *fstr;
910     char buffer[BUFSIZE + 1];
911     strcpy(buffer, param1->buffer);
912
913     // 根据文件扩展名确定文件类型
914     buflen = strlen(buffer);
915     fstr = (char *)0; // 初始化为指向空NULL
916     for (i = 0; extensions[i].ext != 0; i++)
917     {
918         len = strlen(extensions[i].ext);
919         if (!strncmp(&buffer[buflen - len], extensions[i].ext, len))
920         {
921             fstr = extensions[i].filetype;
922             break;
923         }
924     }
925     if (fstr == 0)
926         logger(FORBIDDEN, "file extension type not supported", buffer,
927 fd);
928
929     // 打开文件
930     if ((file_fd = open(&buffer[5], O_RDONLY)) == -1) // &buffer[5] 表示从
931 buffer 字符数组的第五个元素开始的地址，即文件路径的起始位置
932     {
933         logger(NOTFOUND, "failed to open file", &buffer[5], fd);
934     }
935     logger(LOG, "send", &buffer[5], hit);
936     len = (long)lseek(file_fd, (off_t)0, SEEK_END); /* 使用lseek 获得文件长
937 度,该方法比较低效*/
938     (void)lseek(file_fd, (off_t)0, SEEK_SET); /* 想想还有什么方法可获取
939 */
940     (void)sprintf(buffer, "http/1.1 200 ok\nserver: nweb/%d.0\ncontent-
941 length:%ld\nconnection: close\ncontent-type: %s\n\n", VERSION, len,
fstr);
942     logger(LOG, "header", buffer, hit);
943     (void)write(fd, buffer, strlen(buffer)); // 响应头直接发
944
945     while ((ret = read(file_fd, buffer, BUFSIZE)) > 0)
946     {

```

```

942     msg_node *curtask = (msg_node *)malloc(sizeof(msg_node));
943     curtask->next = NULL;
944     curtask->function = send_msg;
945
946     webparam *param2 = malloc(sizeof(webparam));
947     param2->hit = hit;
948     param2->fd = fd;
949     memcpy(param2->buffer, buffer, BUFSIZE); // 这里复制数据不能用
strncpy, strncpy遇到0会停
950     param2->ret = ret;
951     curtask->arg = (void *)param2;
952     push_msgqueue(&sendmsg_pool->queue, curtask);
953 }
954 close(file_fd);
955 return NULL;
956 }
957
958 void *send_msg(void *data)
959 {
960     webparam *param1 = (webparam *)data;
961     int fd = param1->fd;
962     char *message = param1->buffer;
963     long ret = param1->ret;
964
965     (void)write(fd, message, ret); // 写socket
966     usleep(10000);
967     close(fd); // 这里为什么要关闭呢？不是还有其他线程需要往这个端口发送数据吗？
968     return NULL;
969 }
970
971 int main(int argc, char **argv)
972 {
973     .
974     .
975     .
976
977     readmsg_pool = initreadmsgpool(100);
978     readfile_pool = initreadfilepool(100);
979     sendmsg_pool = initsendmsgpool(200);
980     for (hit = 1;; hit++)
981     {
982         length = sizeof(cli_addr);
983         if ((socketfd = accept(listenfd, (struct sockaddr *)&cli_addr,
&length)) < 0)
984             logger(ERROR, "system call", "accept", 0);
985
986         readmsg_node *curtask = (readmsg_node
*)malloc(sizeof(readmsg_node));
987         curtask->next = NULL;
988         curtask->function = read_msg;
989
990         webparam *param = (webparam *)malloc(sizeof(webparam));
991         param->hit = hit;
992         param->fd = socketfd;
993         curtask->arg = (void *)param;

```

```
994         push_rdmsgqueue(&readmsg_pool->queue, curtask);
995     }
996     destoryreadmsgpool(readmsg_pool);
997     destoryreadfilepool(readfile_pool);
998     destorysendmsgpool(sendmsg_pool);
999     return 0;
1000 }
```