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 rmqueuue 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)

```
[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
```

```c
//业务分离 + 性能监测
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <sys/stat.h>
#include <sys/prctl.h>
#include <stdbool.h>
#include <sys/time.h>
#include "head.h"

#define VERSION 23
```

```
#define BUFSIZE 8096
#define ERROR 42
#define LOG 44
#define FORBIDDEN 403
#define NOTFOUND 404
#ifndef SIGCLD
#define SIGCLD SIGCHLD
#endif

struct
{
    char *ext;
    char *filetype;
} extensions[] = {
    {"gif", "image/gif"},
    {"jpg", "image/jpg"},
    {"jpeg", "image/jpeg"},
    {"png", "image/png"},
    {"ico", "image/ico"},
    {"zip", "image/zip"},
    {"gz", "image/gz"},
    {"tar", "image/tar"},
    {"htm", "text/html"},
    {"html", "text/html"},
    {0, 0}};

readmsgpool *readmsg_pool;
readfilepool *readfile_pool;
sendmsgpool *sendmsg_pool;

int max_rmth, max_rfth, max_smth;
int min_rmth = 1000, min_rfth = 1000, min_smth = 1000; // 设一个比线程池容量大的数
double act_rm, act_rf, act_sm;                          // 各个线程池中线程的总活跃时间
double blc_rm, blc_rf, blc_sm;

inline int max(int x, int y)
{
    return x > y ? x : y;
}

inline int min(int x, int y)
{
    return x < y ? x : y;
}

void push_rdmsgqueue(readmsg_queue *queue, readmsg_node *newtask)
{
    readmsg_node *node = (readmsg_node *)malloc(sizeof(readmsg_node));
    if (!node)
    {
        perror("Error allocating memory for new task");
        exit(EXIT_FAILURE);
    }
```

```c

    // 将任务信息复制到新节点
    node->function = newtask->function;
    node->arg = newtask->arg;
    node->next = NULL;

    // 加锁，修改任务队列
    pthread_mutex_lock(&queue->mutex);

    if (queue->len == 0)
    {
        // 队列为空，直接添加新任务
        queue->front = node;
        queue->rear = node;
    }
    else
    {
        // 否则将新任务添加到队列尾部
        queue->rear->next = node;
        queue->rear = node;
    }
    queue->len++;

    // 通知等待在队列上的线程，有新任务到来
    pthread_cond_signal(&queue->has_jobs->cond);

    pthread_mutex_unlock(&queue->mutex);
}

void init_rdmsgqueue(readmsg_queue *queue)
{
    // 初始化互斥量(互斥访问任务队列)
    pthread_mutex_init(&queue->mutex, NULL);

    // 初始化条件变量(在队列为空时阻塞等待任务的到来)
    queue->has_jobs = (rmstaconv *)malloc(sizeof(rmstaconv));
    pthread_cond_init(&queue->has_jobs->cond, NULL);

    // 初始化任务队列
    queue->front = NULL;
    queue->rear = NULL;
    queue->len = 0;
}

readmsg_node *take_rdmsgqueue(readmsg_queue *queue) // 取出队首任务，并在队列中删除该任务
{
    // 加锁，访问任务队列
    pthread_mutex_lock(&queue->mutex);

    // 如果队列为空，等待任务到来
    while (queue->len == 0)
    {
        pthread_cond_wait(&queue->has_jobs->cond, &queue->mutex);
    }

```

```c

    readmsg_node *curtask = queue->front; // 取出队首任务
    queue->front = curtask->next;         // 更新队列头指针,指向下一个任务
    if (queue->len == 1)                  // 如果队列只有一个任务，更新尾指针
    {
        queue->rear = NULL;
    }
    queue->len--;

    // 解锁互斥量
    pthread_mutex_unlock(&queue->mutex);

    return curtask;
}

void destory_rdmsgqueue(readmsg_queue *queue)
{

    pthread_mutex_lock(&queue->mutex); // 加锁，访问任务队列

    while (queue->front != NULL) // 释放队列中所有任务节点
    {
        readmsg_node *curtask = queue->front;
        queue->front = curtask->next;
        free(curtask);
    }
    free(queue->has_jobs); // 释放条件变量

    pthread_mutex_unlock(&queue->mutex);

    pthread_mutex_destroy(&queue->mutex); // 销毁互斥量
}

struct readmsgpool *initreadmsgpool(int num_threads)
{
    readmsgpool *pool;
    pool = (readmsgpool *)malloc(sizeof(struct readmsgpool));
    pool->num_threads = 0;
    pool->num_working = 0;
    pool->is_alive = 1;
    pthread_mutex_init(&(pool->thcount_lock), NULL); // 初始化互斥量
    pthread_cond_init(&(pool->thread_all_idle), NULL); // 初始化条件变量
    init_rdmsgqueue(&pool->queue); // 初始化任务队列@
    pool->threads = (struct rmthread **)malloc(num_threads * sizeof(struct rmthread *)); // 创建线程数组

    int i;
    for (i = 0; i < num_threads; i++)
    {
        create_rmthread(pool, &pool->threads[i], i); // 在pool->threads[i]前加了个&
    }

```

```c
    // 每个线程在创建时,运行函数都会进行pool->num_threads++操作
    while (pool->num_threads != num_threads) // 忙等待，等所有进程创建完毕才返回
    {
    }
    return pool;
}

void waitreadmsgpool(readmsgpool *pool)
{
    pthread_mutex_lock(&pool->thcount_lock);
    while (pool->queue.len || pool->num_working) // 这里可能有问题?
    {
        pthread_cond_wait(&pool->thread_all_idle, &pool->thcount_lock);
    }
    pthread_mutex_unlock(&pool->thcount_lock);
}

void destoryreadmsgpool(readmsgpool *pool)
{
    // 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 @
    waitreadmsgpool(pool);
    pool->is_alive = 0;                                  // 关闭线程池运行
    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)

    destory_rdmsgqueue(&pool->queue); // 销毁任务队列 @

    // 销毁线程指针数组，并释放为线程池分配的内存 @
    int i;
    for (i = 0; i < pool->num_threads; i++)
    {
        free(pool->threads[i]);
    }
    free(pool->threads);

    // 销毁线程池的互斥量和条件变量
    pthread_mutex_destroy(&pool->thcount_lock);
    pthread_cond_destroy(&pool->thread_all_idle);

    // 释放线程池结构体内存
    free(pool);
}

void *rmthread_do(struct rmthread *pthread)
{
    // 设置线程名称
    char thread_name[128] = {0};
    sprintf(thread_name, "thread-pool-%d", pthread->id);

    prctl(PR_SET_NAME, thread_name);

    // 获得/绑定线程池
    readmsgpool *pool = pthread->pool;

```

```
    pthread_mutex_lock(&pool->thcount_lock);
    pool->num_threads++; // 对创建线程数量进程统计@
    pthread_mutex_unlock(&pool->thcount_lock);

    while (pool->is_alive)
    {
        // 如果队列中还有任务，则继续运行；否则阻塞 @
        struct timeval start1, end1;
        gettimeofday(&start1, NULL);
        pthread_mutex_lock(&pool->queue.mutex);
        while (pool->queue.len == 0 && pool->is_alive)
        {
            pthread_cond_wait(&pool->queue.has_jobs->cond, &pool->queue.mutex);
        }
        pthread_mutex_unlock(&pool->queue.mutex);
        gettimeofday(&end1, NULL);

        if (pool->is_alive)
        {
            pthread_mutex_lock(&pool->thcount_lock);
            pool->num_working++; // 对工作线程数量进行统计@
            pthread_mutex_unlock(&pool->thcount_lock);

            // 取任务队列队首，并执行
            int aveth, max_, min_; // 设置局部变量保证线程线程安全
            double totala, totalb;
            struct timeval start2, end2;
            void *(*func)(void *);
            void *arg;
            readmsg_node *curtask = take_rdmsgqueue(&pool->queue); // 取出队首任务，并在队列中删除该任务@（自己实现take_taskqueue）
            if (curtask)
            {
                func = curtask->function;
                arg = curtask->arg;
                gettimeofday(&start2, NULL);
                func(arg); // 执行任务
                gettimeofday(&end2, NULL);

                act_rm += (end2.tv_sec - start2.tv_sec) * 1000.0 + (end2.tv_usec - start2.tv_usec) / 1000.0;
                totala = act_rm; // 保证下面打印时输出的是当前的时间
                blc_rm += (end1.tv_sec - start1.tv_sec) * 1000.0 + (end1.tv_usec - start1.tv_usec) / 1000.0;
                totalb = blc_rm;
                max_rmth = max(max_rmth, pool->num_working);
                max_ = max_rmth;
                min_rmth = min(min_rmth, pool->num_working);
                min_ = min_rmth;
                aveth = (max_ + min_) / 2;

                pthread_mutex_lock(&pool->thcount_lock);
                printf("\nrmqueue 当前长度为：%d\n", pool->queue.len + 1); // 加上被取出的1个
```

```
                printf("rmpool中线程的平均活跃时间：%.3fms\n", totala / aveth);
                printf("rmpool中线程的平均阻塞时间：%.3fms\n", totalb / aveth);
                printf("rmpool中线程的最高活跃数量：%d\n", max_);
                printf("rmpool中线程的最低活跃数量：%d\n", min_);
                printf("rmpool中线程的平均活跃数量：%d\n", aveth);
                pthread_mutex_unlock(&pool->thcount_lock);

                free(arg);     // 释放参数
                free(curtask); // 释放任务
            }
            pthread_mutex_lock(&pool->thcount_lock);
            pool->num_working--;
            pthread_mutex_unlock(&pool->thcount_lock);
            // 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 @
            if (pool->num_working == 0 && pool->queue.len == 0)
                pthread_cond_signal(&pool->thread_all_idle);
        }
    }

    // 线程执行完任务将要退出，需改变线程池中的线程数量 @
    pthread_mutex_unlock(&pool->thcount_lock);
    pool->num_threads--;
    pthread_mutex_unlock(&pool->thcount_lock);
    return NULL;
}

int create_rmthread(struct readmsgpool *pool, struct rmthread **pthread, int id)
{
    *pthread = (struct rmthread *)malloc(sizeof(struct rmthread));
    if (*pthread == NULL)
    {
        perror("create_thread(): Could not allocate memory for thread\n");
        return -1;
    }

    // 设置该线程的属性
    (*pthread)->pool = pool;
    (*pthread)->id = id;

    pthread_create(&(*pthread)->pthread, NULL, (void *)rmthread_do, (*pthread)); // 创建线程
    pthread_detach((*pthread)->pthread);                                          // 线程分离
    return 0;
}

void push_fnamequeue(filename_queue *queue, filename_node *newtask)
{
    filename_node *node = (filename_node *)malloc(sizeof(filename_node));
    if (!node)
```

```c
    {
        perror("Error allocating memory for new task");
        exit(EXIT_FAILURE);
    }

    // 将任务信息复制到新节点
    node->function = newtask->function;
    node->arg = newtask->arg;
    node->next = NULL;

    // 加锁，修改任务队列
    pthread_mutex_lock(&queue->mutex);

    if (queue->len == 0)
    {
        // 队列为空，直接添加新任务
        queue->front = node;
        queue->rear = node;
    }
    else
    {
        // 否则将新任务添加到队列尾部
        queue->rear->next = node;
        queue->rear = node;
    }
    queue->len++;

    // 通知等待在队列上的线程，有新任务到来
    pthread_cond_signal(&queue->has_jobs->cond);

    pthread_mutex_unlock(&queue->mutex);
}

void init_fnamequeue(filename_queue *queue)
{
    // 初始化互斥量(互斥访问任务队列)
    pthread_mutex_init(&queue->mutex, NULL);

    // 初始化条件变量(在队列为空时阻塞等待任务的到来)
    queue->has_jobs = (rfstaconv *)malloc(sizeof(rfstaconv));
    pthread_cond_init(&queue->has_jobs->cond, NULL);

    // 初始化任务队列
    queue->front = NULL;
    queue->rear = NULL;
    queue->len = 0;
}

filename_node *take_fnamequeue(filename_queue *queue) // 取出队首任务，并在队列中删除该任务
{
    // 加锁，访问任务队列
    pthread_mutex_lock(&queue->mutex);

    // 如果队列为空，等待任务到来
```

```
    while (queue->len == 0)
    {
        pthread_cond_wait(&queue->has_jobs->cond, &queue->mutex);
    }

    filename_node *curtask = queue->front; // 取出队首任务
    queue->front = curtask->next;          // 更新队列头指针,指向下一个任务
    if (queue->len == 1)                   // 如果队列只有一个任务，更新尾指针
    {
        queue->rear = NULL;
    }
    queue->len--;

    // 解锁互斥量
    pthread_mutex_unlock(&queue->mutex);

    return curtask;
}

void destory_fnamequeue(filename_queue *queue)
{

    pthread_mutex_lock(&queue->mutex); // 加锁，访问任务队列

    while (queue->front != NULL) // 释放队列中所有任务节点
    {
        filename_node *curtask = queue->front;
        queue->front = curtask->next;
        free(curtask);
    }
    free(queue->has_jobs); // 释放条件变量

    pthread_mutex_unlock(&queue->mutex);

    pthread_mutex_destroy(&queue->mutex); // 销毁互斥量
}

struct readfilepool *initreadfilepool(int num_threads)
{
    readfilepool *pool;
    pool = (readfilepool *)malloc(sizeof(struct readfilepool));
    pool->num_threads = 0;
    pool->num_working = 0;
    pool->is_alive = 1;
    pthread_mutex_init(&(pool->thcount_lock), NULL);
// 初始化互斥量
    pthread_cond_init(&(pool->thread_all_idle), NULL);
// 初始化条件变量
    init_fnamequeue(&pool->queue);
// 初始化任务队列@
    pool->threads = (struct rfthread **)malloc(num_threads *
sizeof(struct rfthread *)); // 创建线程数组

    int i;
    for (i = 0; i < num_threads; i++)
```

```
    {
        create_rfthread(pool, &pool->threads[i], i); // 在pool->threads[i]前加了个&
    }
    // 每个线程在创建时,运行函数都会进行pool->num_threads++操作
    while (pool->num_threads != num_threads) // 忙等待，等所有进程创建完毕才返回
    {
    }
    return pool;
}

void waitreadfilepool(readfilepool *pool)
{
    pthread_mutex_lock(&pool->thcount_lock);
    while (pool->queue.len || pool->num_working) // 这里可能有问题?
    {
        pthread_cond_wait(&pool->thread_all_idle, &pool->thcount_lock);
    }
    pthread_mutex_unlock(&pool->thcount_lock);
}

void destoryreadfilepool(readfilepool *pool)
{
    // 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 @
    waitreadfilepool(pool);
    pool->is_alive = 0;                                   // 关闭线程池运行
    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)

    destory_fnamequeue(&pool->queue); // 销毁任务队列 @

    // 销毁线程指针数组，并释放为线程池分配的内存 @
    int i;
    for (i = 0; i < pool->num_threads; i++)
    {
        free(pool->threads[i]);
    }
    free(pool->threads);

    // 销毁线程池的互斥量和条件变量
    pthread_mutex_destroy(&pool->thcount_lock);
    pthread_cond_destroy(&pool->thread_all_idle);

    // 释放线程池结构体内存
    free(pool);
}

void *rfthread_do(struct rfthread *pthread)
{
    // 设置线程名称
    char thread_name[128] = {0};
    sprintf(thread_name, "thread-pool-%d", pthread->id);

    prctl(PR_SET_NAME, thread_name);
```

```
    // 获得/绑定线程池
    readfilepool *pool = pthread->pool;

    pthread_mutex_lock(&pool->thcount_lock);
    pool->num_threads++; // 对创建线程数量进程统计@
    pthread_mutex_unlock(&pool->thcount_lock);

    while (pool->is_alive)
    {
        // 如果队列中还有任务，则继续运行；否则阻塞 @
        struct timeval start1, end1;
        gettimeofday(&start1, NULL);
        pthread_mutex_lock(&pool->queue.mutex);
        while (pool->queue.len == 0 && pool->is_alive)
        {
            pthread_cond_wait(&pool->queue.has_jobs->cond, &pool->queue.mutex);
        }
        pthread_mutex_unlock(&pool->queue.mutex);
        gettimeofday(&end1, NULL);

        if (pool->is_alive)
        {
            pthread_mutex_lock(&pool->thcount_lock);
            pool->num_working++; // 对工作线程数量进行统计@
            pthread_mutex_unlock(&pool->thcount_lock);

            // 取任务队列队首，并执行
            int aveth, max_, min_;
            double totala, totalb;
            struct timeval start2, end2;
            void *(*func)(void *);
            void *arg;
            filename_node *curtask = take_fnamequeue(&pool->queue); // 取出队首任务，并在队列中删除该任务@（自己实现take_fnamequeue）
            if (curtask)
            {
                func = curtask->function;
                arg = curtask->arg;
                gettimeofday(&start2, NULL);
                func(arg); // 执行任务
                gettimeofday(&end2, NULL);

                act_rf += (end2.tv_sec - start2.tv_sec) * 1000.0 + (end2.tv_usec - start2.tv_usec) / 1000.0;
                totala = act_rf;
                blc_rf += (end1.tv_sec - start1.tv_sec) * 1000.0 + (end1.tv_usec - start1.tv_usec) / 1000.0;
                totalb = blc_rf;
                max_rfth = max(max_rfth, pool->num_working);
                max_ = max_rfth;
                min_rfth = min(min_rfth, pool->num_working);
                min_ = min_rfth;
                aveth = (max_ + min_) / 2;

```

```c

                pthread_mutex_lock(&pool->thcount_lock);
                printf("\nrfqueue 当前长度为：%d\n", pool->queue.len + 1); // 加上被取出的1个
                printf("rfpool中线程的平均活跃时间：%.3fms\n", totala / aveth);
                printf("rfpool中线程的平均阻塞时间：%.3fms\n", totalb / aveth);
                printf("rfpool中线程的最高活跃数量：%d\n", max_);
                printf("rfpool中线程的最低活跃数量：%d\n", min_);
                printf("rfpool中线程的平均活跃数量：%d\n", aveth);
                pthread_mutex_unlock(&pool->thcount_lock);

                free(arg);     // 释放参数
                free(curtask); // 释放任务
            }
            pthread_mutex_lock(&pool->thcount_lock);
            pool->num_working--;
            pthread_mutex_unlock(&pool->thcount_lock);
            // 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 @
            if (pool->num_working == 0 && pool->queue.len == 0)
                pthread_cond_signal(&pool->thread_all_idle);
        }
    }

    // 线程执行完任务将要退出，需改变线程池中的线程数量 @
    pthread_mutex_unlock(&pool->thcount_lock);
    pool->num_threads--;
    pthread_mutex_unlock(&pool->thcount_lock);
    return NULL;
}

int create_rfthread(struct readfilepool *pool, struct rfthread **pthread, int id)
{
    *pthread = (struct rfthread *)malloc(sizeof(struct rfthread));
    if (*pthread == NULL)
    {
        perror("create_thread(): Could not allocate memory for thread\n");
        return -1;
    }

    // 设置该线程的属性
    (*pthread)->pool = pool;
    (*pthread)->id = id;

    pthread_create(&(*pthread)->pthread, NULL, (void *)rfthread_do, (*pthread)); // 创建线程
    pthread_detach((*pthread)->pthread); // 线程分离
    return 0;
}

```

```c
void push_msgqueue(msg_queue *queue, msg_node *newtask)
{
    msg_node *node = (msg_node *)malloc(sizeof(msg_node));
    if (!node)
    {
        perror("Error allocating memory for new task");
        exit(EXIT_FAILURE);
    }

    // 将任务信息复制到新节点
    node->function = newtask->function;
    node->arg = newtask->arg;
    node->next = NULL;

    // 加锁，修改任务队列
    pthread_mutex_lock(&queue->mutex);

    if (queue->len == 0)
    {
        // 队列为空，直接添加新任务
        queue->front = node;
        queue->rear = node;
    }
    else
    {
        // 否则将新任务添加到队列尾部
        queue->rear->next = node;
        queue->rear = node;
    }
    queue->len++;

    // 通知等待在队列上的线程，有新任务到来
    pthread_cond_signal(&queue->has_jobs->cond);

    pthread_mutex_unlock(&queue->mutex);
}

void init_msgqueue(msg_queue *queue)
{
    // 初始化互斥量(互斥访问任务队列)
    pthread_mutex_init(&queue->mutex, NULL);

    // 初始化条件变量(在队列为空时阻塞等待任务的到来)
    queue->has_jobs = (smstaconv *)malloc(sizeof(smstaconv));
    pthread_cond_init(&queue->has_jobs->cond, NULL);

    // 初始化任务队列
    queue->front = NULL;
    queue->rear = NULL;
    queue->len = 0;
}

msg_node *take_msgqueue(msg_queue *queue) // 取出队首任务，并在队列中删除该任务
{
    // 加锁，访问任务队列
```

```
    pthread_mutex_lock(&queue->mutex);

    // 如果队列为空，等待任务到来
    while (queue->len == 0)
    {
        pthread_cond_wait(&queue->has_jobs->cond, &queue->mutex);
    }

    msg_node *curtask = queue->front; // 取出队首任务
    queue->front = curtask->next;     // 更新队列头指针,指向下一个任务
    if (queue->len == 1)              // 如果队列只有一个任务，更新尾指针
    {
        queue->rear = NULL;
    }
    queue->len--;

    // 解锁互斥量
    pthread_mutex_unlock(&queue->mutex);

    return curtask;
}

void destory_msgqueue(msg_queue *queue)
{

    pthread_mutex_lock(&queue->mutex); // 加锁，访问任务队列

    while (queue->front != NULL) // 释放队列中所有任务节点
    {
        msg_node *curtask = queue->front;
        queue->front = curtask->next;
        free(curtask);
    }
    free(queue->has_jobs); // 释放条件变量

    pthread_mutex_unlock(&queue->mutex);

    pthread_mutex_destroy(&queue->mutex); // 销毁互斥量
}

struct sendmsgpool *initsendmsgpool(int num_threads)
{
    sendmsgpool *pool;
    pool = (sendmsgpool *)malloc(sizeof(struct sendmsgpool));
    pool->num_threads = 0;
    pool->num_working = 0;
    pool->is_alive = 1;
    pthread_mutex_init(&(pool->thcount_lock), NULL); // 初始化互斥量
    pthread_cond_init(&(pool->thread_all_idle), NULL); // 初始化条件变量
    init_msgqueue(&pool->queue); // 初始化任务队列@
    pool->threads = (struct smthread **)malloc(num_threads * sizeof(struct smthread *)); // 创建线程数组
```

```c

    int i;
    for (i = 0; i < num_threads; i++)
    {
        create_smthread(pool, &pool->threads[i], i); // 在pool->threads[i]前加了个&
    }
    // 每个线程在创建时,运行函数都会进行pool->num_threads++操作
    while (pool->num_threads != num_threads) // 忙等待，等所有进程创建完毕才返回
    {
    }
    return pool;
}

void waitsendmsgpool(sendmsgpool *pool)
{
    pthread_mutex_lock(&pool->thcount_lock);
    while (pool->queue.len || pool->num_working) // 这里可能有问题?
    {
        pthread_cond_wait(&pool->thread_all_idle, &pool->thcount_lock);
    }
    pthread_mutex_unlock(&pool->thcount_lock);
}

void destorysendmsgpool(sendmsgpool *pool)
{
    // 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 @
    waitsendmsgpool(pool);
    pool->is_alive = 0;                                 // 关闭线程池运行
    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)

    destory_msgqueue(&pool->queue); // 销毁任务队列 @

    // 销毁线程指针数组，并释放为线程池分配的内存 @
    int i;
    for (i = 0; i < pool->num_threads; i++)
    {
        free(pool->threads[i]);
    }
    free(pool->threads);

    // 销毁线程池的互斥量和条件变量
    pthread_mutex_destroy(&pool->thcount_lock);
    pthread_cond_destroy(&pool->thread_all_idle);

    // 释放线程池结构体内存
    free(pool);
}

void *smthread_do(struct smthread *pthread)
{
    // 设置线程名称
    char thread_name[128] = {0};

```

```
    sprintf(thread_name, "thread-pool-%d", pthread->id);

    prctl(PR_SET_NAME, thread_name);

    // 获得/绑定线程池
    sendmsgpool *pool = pthread->pool;

    pthread_mutex_lock(&pool->thcount_lock);
    pool->num_threads++; // 对创建线程数量进程统计@
    pthread_mutex_unlock(&pool->thcount_lock);

    while (pool->is_alive)
    {
        // 如果队列中还有任务，则继续运行；否则阻塞 @
        struct timeval start1, end1;
        gettimeofday(&start1, NULL);
        pthread_mutex_lock(&pool->queue.mutex);
        while (pool->queue.len == 0 && pool->is_alive)
        {
            pthread_cond_wait(&pool->queue.has_jobs->cond, &pool->queue.mutex);
        }
        pthread_mutex_unlock(&pool->queue.mutex);
        gettimeofday(&end1, NULL);

        if (pool->is_alive)
        {
            pthread_mutex_lock(&pool->thcount_lock);
            pool->num_working++; // 对工作线程数量进行统计@
            pthread_mutex_unlock(&pool->thcount_lock);

            // 取任务队列队首，并执行
            int aveth, max_, min_;
            double totala, totalb;
            struct timeval start2, end2;
            void *(*func)(void *);
            void *arg;
            msg_node *curtask = take_msgqueue(&pool->queue); // 取出队首任务，并在队列中删除该任务@（自己实现take_msgqueue）
            if (curtask)
            {
                func = curtask->function;
                arg = curtask->arg;
                gettimeofday(&start2, NULL);
                func(arg); // 执行任务
                gettimeofday(&end2, NULL);

                act_sm += (end2.tv_sec - start2.tv_sec) * 1000.0 + (end2.tv_usec - start2.tv_usec) / 1000.0;
                totala = act_sm;
                blc_sm += (end1.tv_sec - start1.tv_sec) * 1000.0 + (end1.tv_usec - start1.tv_usec) / 1000.0;
                totalb = blc_sm;
                max_smth = max(max_smth, pool->num_working);
                max_ = max_smth;
```

```
                min_smth = min(min_smth, pool->num_working);
                min_ = min_smth;
                aveth = (max_ + min_) / 2;
                pthread_mutex_lock(&pool->thcount_lock);
                printf("\nsmqueue 当前长度为：%d\n", pool->queue.len + 1);
                printf("smpool中线程的平均活跃时间：%.3fms\n", totala / aveth);
                printf("smpool中线程的平均阻塞时间：%.3fms\n", totalb / aveth);
                printf("smpool中线程的最高活跃数量：%d\n", max_);
                printf("smpool中线程的最低活跃数量：%d\n", min_);
                printf("smpool中线程的平均活跃数量：%d\n", aveth);
                pthread_mutex_unlock(&pool->thcount_lock);
                free(curtask); // 释放任务
            }
            pthread_mutex_lock(&pool->thcount_lock);
            pool->num_working--;
            pthread_mutex_unlock(&pool->thcount_lock);
            // 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 @
            if (pool->num_working == 0 && pool->queue.len == 0)
                pthread_cond_signal(&pool->thread_all_idle);
        }
    }

    // 线程执行完任务将要退出，需改变线程池中的线程数量 @
    pthread_mutex_unlock(&pool->thcount_lock);
    pool->num_threads--;
    pthread_mutex_unlock(&pool->thcount_lock);
    return NULL;
}

int create_smthread(struct sendmsgpool *pool, struct smthread **pthread, int id)
{
    *pthread = (struct smthread *)malloc(sizeof(struct smthread));
    if (*pthread == NULL)
    {
        perror("create_thread(): Could not allocate memory for thread\n");
        return -1;
    }

    // 设置该线程的属性
    (*pthread)->pool = pool;
    (*pthread)->id = id;

    pthread_create(&(*pthread)->pthread, NULL, (void *)smthread_do, (*pthread)); // 创建线程
    pthread_detach((*pthread)->pthread);
// 线程分离
    return 0;
}

void logger(int type, char *s1, char *s2, int socket_fd)
```

```
{
    ...
}

void *read_msg(void *data)
{
    webparam *param1 = (webparam *)data;
    int fd = param1->fd, hit = param1->hit;
    int j;
    long i, ret;
    char buffer[BUFSIZE + 1]; /* 缓存 */

    ret = read(fd, buffer, BUFSIZE); // 从socket 读取 web 请求内容(读socket)
    if (ret == 0 || ret == -1)
    { /* 读取失败 */
        logger(FORBIDDEN, "failed to read browser request", "", fd);
    }
    else
    {
        if (ret > 0 && ret < BUFSIZE) // 确保读取到的数据以 null 字符 ('\0')
结尾
            buffer[ret] = 0;
        else
            buffer[0] = 0;
        for (i = 0; i < ret; i++)
            if (buffer[i] == '\r' || buffer[i] == '\n')
                buffer[i] = '*';
        logger(LOG, "request", buffer, hit);
        if (strncmp(buffer, "GET ", 4) != 0)
        {
            logger(FORBIDDEN, "only simple get operation supported",
buffer, fd);
        }
        for (i = 4; i < BUFSIZE; i++)
        {
            if (buffer[i] == ' ')
            {
                buffer[i] = 0;
                break;
            }
        }
        for (j = 0; j < i - 1; j++)
            if (buffer[j] == '.' && buffer[j + 1] == '.')
            {
                logger(FORBIDDEN, "parent directory (..) path names not
supported", buffer, fd);
            }
        if (!strncmp(&buffer[0], "GET /\0", 6))
            (void)strcpy(buffer, "GET /index. html");

        filename_node *curtask = (filename_node
*)malloc(sizeof(filename_node));
        curtask->next = NULL;
        curtask->function = read_file;

```

```
        webparam *param2 = (webparam *)malloc(sizeof(webparam));
        param2->hit = hit;
        param2->fd = fd;
        strcpy(param2->buffer, buffer);
        curtask->arg = (void *)param2;
        push_fnamequeue(&readfile_pool->queue, curtask);
    }
    return NULL;
}

void *read_file(void *data)
{
    webparam *param1 = (webparam *)data;
    int fd = param1->fd, hit = param1->hit;
    int file_fd, buflen;
    long i, ret, len;
    char *fstr;
    char buffer[BUFSIZE + 1];
    strcpy(buffer, param1->buffer);

    // 根据文件扩展名确定文件类型
    buflen = strlen(buffer);
    fstr = (char *)0; // 初始化为指向空NULL
    for (i = 0; extensions[i].ext != 0; i++)
    {
        len = strlen(extensions[i].ext);
        if (!strncmp(&buffer[buflen - len], extensions[i].ext, len))
        {
            fstr = extensions[i].filetype;
            break;
        }
    }
    if (fstr == 0)
        logger(FORBIDDEN, "file extension type not supported", buffer, fd);

    // 打开文件
    if ((file_fd = open(&buffer[5], O_RDONLY)) == -1) // &buffer[5] 表示从 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
    {
        logger(NOTFOUND, "failed to open file", &buffer[5], fd);
    }
    logger(LOG, "send", &buffer[5], hit);
    len = (long)lseek(file_fd, (off_t)0, SEEK_END); /* 使用lseek 获得文件长度,该方法比较低效*/
    (void)lseek(file_fd, (off_t)0, SEEK_SET);       /* 想想还有什么方法可获取 */
    (void)sprintf(buffer, "http/1.1 200 ok\nserver: nweb/%d.0\ncontent-length:%ld\nconnection: close\ncontent-type: %s\n\n", VERSION, len, fstr);
    logger(LOG, "header", buffer, hit);
    (void)write(fd, buffer, strlen(buffer)); // 响应头直接发

    while ((ret = read(file_fd, buffer, BUFSIZE)) > 0)
    {
```

```
        msg_node *curtask = (msg_node *)malloc(sizeof(msg_node));
        curtask->next = NULL;
        curtask->function = send_msg;

        webparam *param2 = malloc(sizeof(webparam));
        param2->hit = hit;
        param2->fd = fd;
        memcpy(param2->buffer, buffer, BUFSIZE); // 这里复制数据不能用strcpy，strcpy遇到0会停
        param2->ret = ret;
        curtask->arg = (void *)param2;
        push_msgqueue(&sendmsg_pool->queue, curtask);
    }
    close(file_fd);
    return NULL;
}

void *send_msg(void *data)
{
    webparam *param1 = (webparam *)data;
    int fd = param1->fd;
    char *message = param1->buffer;
    long ret = param1->ret;

    (void)write(fd, message, ret); // 写socket
    usleep(10000);
    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？
    return NULL;
}

int main(int argc, char **argv)
{
    .
    .
    .

    readmsg_pool = initreadmsgpool(100);
    readfile_pool = initreadfilepool(100);
    sendmsg_pool = initsendmsgpool(200);
    for (hit = 1;; hit++)
    {
        length = sizeof(cli_addr);
        if ((socketfd = accept(listenfd, (struct sockaddr *)&cli_addr, &length)) < 0)
            logger(ERROR, "system call", "accept", 0);

        readmsg_node *curtask = (readmsg_node *)malloc(sizeof(readmsg_node));
        curtask->next = NULL;
        curtask->function = read_msg;

        webparam *param = (webparam *)malloc(sizeof(webparam));
        param->hit = hit;
        param->fd = socketfd;
        curtask->arg = (void *)param;
```

```
        push_rdmsgqueue(&readmsg_pool->queue, curtask);
    }
    destoryreadmsgpool(readmsg_pool);
    destoryreadfilepool(readfile_pool);
    destorysendmsgpool(sendmsg_pool);
    return 0;
}
```