实验5 Web服务器的业务分割模型

2021011615 田天成 计算机21-3

题目1.

> 实现上述的业务分割模型的Web服务程序。

要使用多个线程池来处理不同类型的任务，可以按照以下步骤设计架构：

1. **确定任务分类**：首先，根据任务的性质和需求进行分类。为每个分类创建专用的线程池。根据任务的性质配置每个线程池，例如：
    - **网络连接池**：这个池处理进来的连接请求。
    - **请求解析池**：解析请求可能是CPU密集型的，尤其是当处理大负载或复杂数据结构时。一个更高数量的线程将会适合于这里。
2. **任务排队和管理**：为每个线程池使用适当的排队策略。
3. **错误处理和韧性**：在每个池中实现健壮的错误处理。如果一个线程遇到异常，它不应该使整个池崩溃。

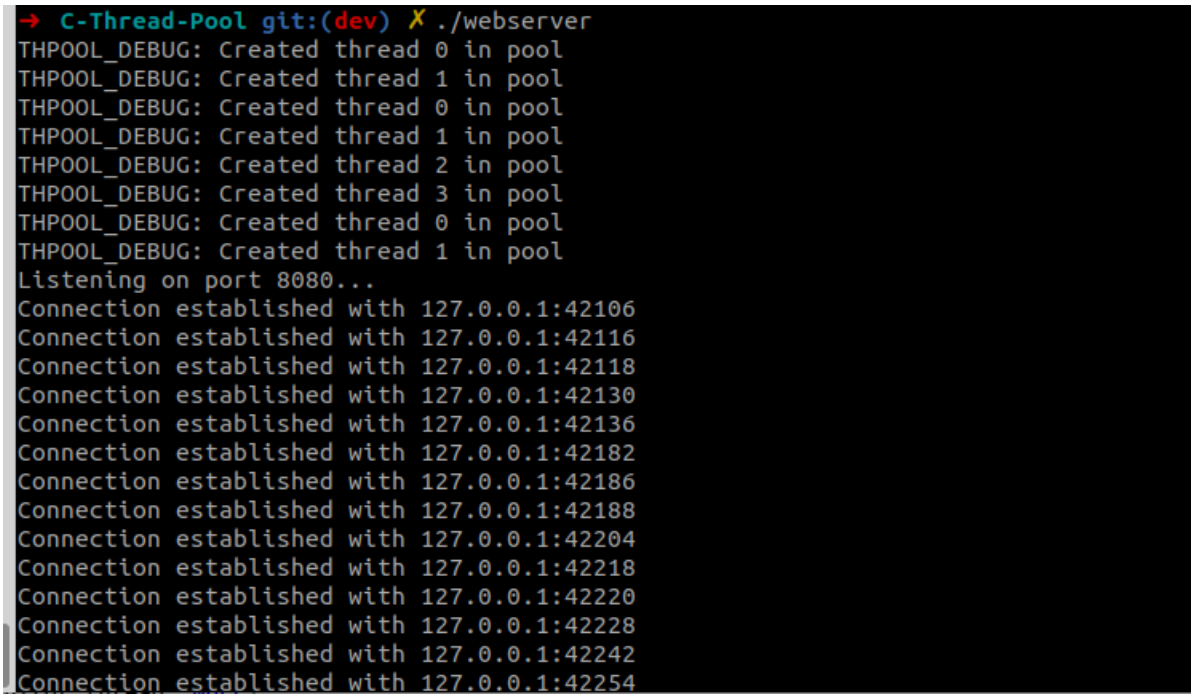
```c
#include "thpool.h" // 包含线程池头文件
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/sendfile.h>
#include <sys/stat.h>

#define PORT 8080 // 定义端口号
#define BUFFER_SIZE 4096 // 定义缓冲区大小
#define WEB_ROOT "./" // 服务器资源所在的目录

// 线程池定义
threadpool read_msg_threadpool; // 读取消息的线程池
threadpool read_file_threadpool; // 读取文件的线程池
threadpool send_msg_threadpool; // 发送消息的线程池

typedef struct
{
    int client_fd; // 客户端文件描述符
    char request[BUFFER_SIZE]; // 请求缓冲区
} client_request;

typedef struct
{
    client_request *request; // 指向客户端请求的指针
    char *filename; // 文件名
    int file_fd; // 文件描述符
    struct stat file_stat; // 存储文件大小和其他属性的结构
```

```c
    char *file_content; // 存储文件内容的缓冲区
    ssize_t file_content_length; // 文件内容的长度
} response_data;

// 函数原型
void read_request(int client_fd);
void read_file_and_respond(client_request *request);
void send_response(response_data *data);

int main()
{
    int server_fd, client_fd; // 服务器和客户端的文件描述符
    struct sockaddr_in address; // 地址结构
    int opt = 1;
    int addrlen = sizeof(address);

    // 初始化线程池
    read_msg_threadpool = thpool_init(2); // 初始化处理读取消息的线程池
    read_file_threadpool = thpool_init(4); // 初始化处理读取文件的线程池
    send_msg_threadpool = thpool_init(2); // 初始化处理发送消息的线程池

    // 创建套接字文件描述符
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // 强制将套接字绑定到端口8080
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
&opt, sizeof(opt)))
    {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET; // 设置地址族为IPv4
    address.sin_addr.s_addr = INADDR_ANY; // 监听任何地址
    address.sin_port = htons(PORT); // 设置端口号

    // 将套接字绑定到地址
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0)
    {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    // 监听套接字
    if (listen(server_fd, 3) < 0)
    {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    printf("Listening on port %d...\n", PORT);
```

```
88
89      while (1)
90      {
91          // 接受客户端连接
92          if ((client_fd = accept(server_fd, (struct sockaddr *)&address,
(socklen_t *)&addrlen)) < 0)
93          {
94              perror("accept");
95              continue;
96          }
97          printf("Connection established with %s:%d\n",
inet_ntoa(address.sin_addr), ntohs(address.sin_port));
98
99          // 将任务分配给read_msg_threadpool
100         thpool_add_work(read_msg_threadpool, (void (*)(void
*))read_request, (void *)(intptr_t)client_fd);
101     }
102
103     // 清理工作
104     thpool_destroy(read_msg_threadpool); // 销毁读取消息的线程池
105     thpool_destroy(read_file_threadpool); // 销毁读取文件的线程池
106     thpool_destroy(send_msg_threadpool); // 销毁发送消息的线程池
107     close(server_fd); // 关闭服务器文件描述符
108     return 0;
109 }
110
111 // 读取客户端请求的函数
112 void read_request(int client_fd)
113 {
114     client_request *request = malloc(sizeof(client_request));
115     request->client_fd = client_fd;
116     int bytes_read = recv(client_fd, request->request, BUFFER_SIZE, 0);
117     if (bytes_read <= 0)
118     {
119         close(client_fd);
120         free(request);
121         return;
122     }
123     thpool_add_work(read_file_threadpool, (void (*)(void
*))read_file_and_respond, request);
124 }
125
126 // 读取文件并响应的函数
127 void read_file_and_respond(client_request *request)
128 {
129     // 从HTTP请求中提取请求的URL
130     // 为简单起见，假设URL是请求中的第二个单词
131     char *token = strtok(request->request, " ");
132     token = strtok(NULL, " ");
133     if (token == NULL)
134     {
135         close(request->client_fd);
136         free(request);
137         return;
138     }
```

```
139
140     char filepath[256];
141     snprintf(filepath, sizeof(filepath), "%s%s", WEB_ROOT, token);
142
143     int file_fd = open(filepath, O_RDONLY);
144     if (file_fd < 0)
145     {
146         // 文件未找到，处理错误
147         close(request->client_fd);
148         free(request);
149         return;
150     }
151
152     // if (fstat(file_fd, &request->file_stat) < 0)
153     // {
154     //     // 获取文件状态错误
155     //     close(file_fd);
156     //     close(request->client_fd);
157     //     free(request);
158     //     return;
159     // }
160
161     response_data *data = malloc(sizeof(response_data));
162     data->request = request;
163     data->file_fd = file_fd;
164     data->filename = strdup(filepath);
165
166     // 使用fstat获取文件大小
167     if (fstat(file_fd, &data->file_stat) < 0)
168     {
169         // 处理错误
170         close(file_fd);
171         close(request->client_fd);
172         free(request);
173         free(data);
174         return;
175     }
176
177     // 为文件内容分配缓冲区
178     data->file_content = malloc(data->file_stat.st_size);
179     if (data->file_content == NULL)
180     {
181         // 处理内存分配错误
182     }
183     data->file_content_length = read(file_fd, data->file_content, data-
>file_stat.st_size);
184     if (data->file_content_length < 0)
185     {
186         // 处理读取错误
187     }
188
189     thpool_add_work(send_msg_threadpool, (void (*)(void *))send_response,
data);
190 }
191
```

```c
// 发送响应的函数
void send_response(response_data *data)
{
    // 发送文件前发送HTTP头
    char header[BUFFER_SIZE];
    sprintf(header, "HTTP/1.1 200 OK\r\nContent-Length: %ld\r\n\r\n", data->file_content_length);
    send(data->request->client_fd, header, strlen(header), 0);

    // 发送文件内容
    ssize_t sent_bytes = send(data->request->client_fd, data->file_content, data->file_content_length, 0);
    if (sent_bytes < 0)
    {
        perror("Error sending file content");
    }

    // 清理
    close(data->file_fd);
    close(data->request->client_fd);
    free(data->file_content);
    free(data->request);
    free(data->filename);
    free(data);
}
```



题目2.

> 在程序里面设置性能监测代码，通过定时打印这些性能参数，能够分析此Web服务程序的运行状态。例如，监测线程池中线程的平均活跃时间和阻塞时间，线程最高活跃数量、最低活跃数量、平均活跃数量；消息队列中消息的长度等。除此以外还可以利用相关系统命令来监测 系统的I/O、内存、CPU等设备性能。

a. 消息队列的长度

1. 定义用于存放线程池指针的结构体和获取每个线程池状态的函数

```
typedef struct thpool_collection {
    thpool_* read_msg_thpool;
    thpool_* read_file_thpool;
    thpool_* send_msg_thpool;
} thpool_collection;

void get_thpool_status(thpool_* thpool, thpool_monitor *monitor) {
    pthread_mutex_lock(&thpool->thcount_lock);
    monitor->num_threads_alive = thpool->num_threads_alive;
    monitor->num_threads_working = thpool->num_threads_working;
    pthread_mutex_unlock(&thpool->thcount_lock);

    pthread_mutex_lock(&thpool->jobqueue.rwmutex);
    monitor->job_queue_length = thpool->jobqueue.len;
    pthread_mutex_unlock(&thpool->jobqueue.rwmutex);
}
```

2. 定义监控函数

```
void *monitor_all_thpools(void *arg) {
    thpool_collection *thpools = (thpool_collection*)arg;
    thpool_monitor read_msg_monitor, read_file_monitor, send_msg_monitor;

    while (1) {
        sleep(5); // Monitoring every 5 seconds

        get_thpool_status(thpools->read_msg_thpool, &read_msg_monitor);
        get_thpool_status(thpools->read_file_thpool, &read_file_monitor);
        get_thpool_status(thpools->send_msg_thpool, &send_msg_monitor);

        // Log the status for each thread pool
        // ... [Print statements for each thread pool]
    }
    return NULL;
}
```

3. 更新主函数

创建 `thpool_collection` 实例并传递给监控线程

```
int main() {
    // Initialize thread pools
    thpool_* read_msg_threadpool = thpool_init(1);
    thpool_* read_file_threadpool = thpool_init(4);
    thpool_* send_msg_threadpool = thpool_init(1);

    // Create and populate the thpool collection structure
    thpool_collection thpools = {
        .read_msg_thpool = read_msg_threadpool,
        .read_file_thpool = read_file_threadpool,
        .send_msg_thpool = send_msg_threadpool
```

```
    };

    // Start the monitoring thread
    pthread_t monitor_thread;
    if (pthread_create(&monitor_thread, NULL, monitor_all_thpools, &thpools)
!= 0) {
        perror("Failed to create monitor thread");
        // Clean up the thread pools
        return 1;
    }

    // Rest of  server logic...

    // Cleanup
    pthread_join(monitor_thread, NULL);
    thpool_destroy(read_msg_threadpool);
    thpool_destroy(read_file_threadpool);
    thpool_destroy(send_msg_threadpool);
    // Other cleanup if necessary
    return 0;
}
```

```
Read Message ThreadPool - Threads alive: 2, Threads working: 2, Job queue length: 1
Read File ThreadPool - Threads alive: 4, Threads working: 0, Job queue length: 0
Send Message ThreadPool - Threads alive: 2, Threads working: 0, Job queue length: 0
Read Message ThreadPool - Threads alive: 2, Threads working: 1, Job queue length: 1
Read File ThreadPool - Threads alive: 4, Threads working: 0, Job queue length: 0
Send Message ThreadPool - Threads alive: 2, Threads working: 2, Job queue length: 0
Read Message ThreadPool - Threads alive: 2, Threads working: 2, Job queue length: 4
Read File ThreadPool - Threads alive: 4, Threads working: 0, Job queue length: 0
Send Message ThreadPool - Threads alive: 2, Threads working: 1, Job queue length: 0
```

b. 线程的平均活跃和阻塞时间

想在现有的线程池实现中添加一个监控模块，以监测线程的平均活跃时间和阻塞时间。以下是对代码的修改：

1. **添加新的数据结构**：为了跟踪每个线程的活跃和阻塞时间，我们需要在 `thread` 结构体中添加新的字段。例如，可以添加 `time_t last_active_time` 和 `time_t last_block_time`，以及累计活跃和阻塞时间的字段。
2. **更新线程函数**：在 `thread_do` 函数中，在线程开始工作时更新 `last_active_time`，在工作完成后更新 `last_block_time`。这样，可以计算出每个线程的活跃和阻塞时间。
3. **添加监控功能**：已经定义了一个 `monitor_thpool` 函数，该函数可以进一步扩展以包括计算和打印每个线程的平均活跃时间和阻塞时间的逻辑。
4. **线程同步**：考虑到多线程环境中的数据完整性，确保在更新和读取线程统计信息时使用适当的锁。

以下是一些代码示例来说明这些更改：

添加新字段到 `thread` 结构体

```c
typedef struct thread {
    // ... 现有字段 ...

    time_t last_active_time;
    time_t last_block_time;
    double total_active_time;
    double total_block_time;
} thread;
```

更新 `thread_do` 函数

```c
static void *thread_do(struct thread *thread_p) {
    // ... 现有逻辑 ...

    while (threads_keepalive) {
        thread_p->last_block_time = time(NULL);
        // 等待工作
        bsem_wait(thpool_p->jobqueue.has_jobs);

        if (threads_keepalive) {
            thread_p->total_block_time += difftime(time(NULL), thread_p->last_block_time);
            thread_p->last_active_time = time(NULL);

            // 执行工作
            // ...

            thread_p->total_active_time += difftime(time(NULL), thread_p->last_active_time);
        }
    }
    // ...
}
```

扩展 `monitor_thpool` 函数

```c
void *monitor_thpool(void *arg) {
    thpool_ *thpool = (thpool_ *)arg;
    // ...

    while (1) {
        sleep(10);
        get_thpool_status(thpool, &monitor);

        // 计算平均活跃时间和阻塞时间
        double avg_active_time = 0;
        double avg_block_time = 0;

        for (int i = 0; i < thpool->num_threads_alive; i++) {
            avg_active_time += thpool->threads[i]->total_active_time;
            avg_block_time += thpool->threads[i]->total_block_time;
        }
        avg_active_time /= thpool->num_threads_alive;
        avg_block_time /= thpool->num_threads_alive;
```

```
19
20          printf("平均活跃时间: %f, 平均阻塞时间: %f\n", avg_active_time,
    avg_block_time);
21          // ...
22      }
23  }
```

执行命令`➜  http_load-09Mar2016 ./http_load -parallel 100 -seconds 100 urls.txt`

c.线程最高活跃数量、最低活跃数量、平均活跃数量;

1. **修改线程池结构体**:
   在`thpool_`结构体中添加字段来存储最高活跃数量、最低活跃数量、总活跃数量和活跃统计次数。总活跃数量和活跃统计次数将用于计算平均活跃数量。

```
1  typedef struct thpool_ {
2      // ... 现有字段 ...
3
4      int max_active_threads;
5      int min_active_threads;
6      long long total_active_threads; // 长期运行的线程池可能需要长整型来避免溢出
7      long long active_count;          // 统计次数
8  } thpool_;
```

2. **初始化新字段**:
   在创建线程池时，初始化这些新字段。最初，可以将最小活跃数量设置为线程池的最大容量。

```
1  struct thpool_ *thpool_init(int num_threads) {
2      // ... 现有初始化代码 ...
3
4      thpool_p->max_active_threads = 0;
5      thpool_p->min_active_threads = num_threads; // 初始时假设所有线程都可能
   活跃
6      thpool_p->total_active_threads = 0;
7      thpool_p->active_count = 0;
8
9      // ...
10 }
```

3. **更新活跃线程统计**:
   在`thread_do`函数或任何适当的位置更新这些统计。每次线程开始工作时，更新最大和最小活跃线程数量，并累加总活跃线程数。

```
1  static void *thread_do(struct thread *thread_p) {
```

```
    // ... 现有代码 ...

    while (threads_keepalive) {
        // ...

        if (threads_keepalive) {
            pthread_mutex_lock(&thpool_p->thcount_lock);
            thpool_p->num_threads_working++;
            if (thpool_p->num_threads_working > thpool_p->max_active_threads) {
                thpool_p->max_active_threads = thpool_p->num_threads_working;
            }
            if (thpool_p->num_threads_working < thpool_p->min_active_threads) {
                thpool_p->min_active_threads = thpool_p->num_threads_working;
            }
            thpool_p->total_active_threads += thpool_p->num_threads_working;
            thpool_p->active_count++;
            pthread_mutex_unlock(&thpool_p->thcount_lock);

            // 执行工作 ...

            pthread_mutex_lock(&thpool_p->thcount_lock);
            thpool_p->num_threads_working--;
            pthread_mutex_unlock(&thpool_p->thcount_lock);
        }
    }

    // ...
}
```

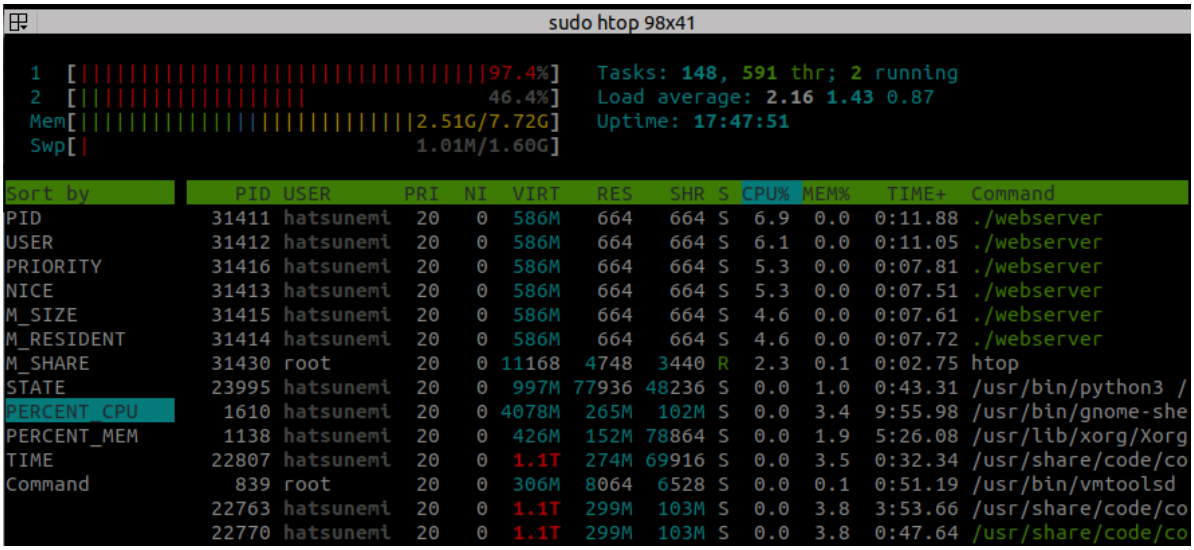
4. **计算平均活跃数量**:
   在监控功能中，现在可以计算平均活跃线程数量。确保处理分母为零的情况。

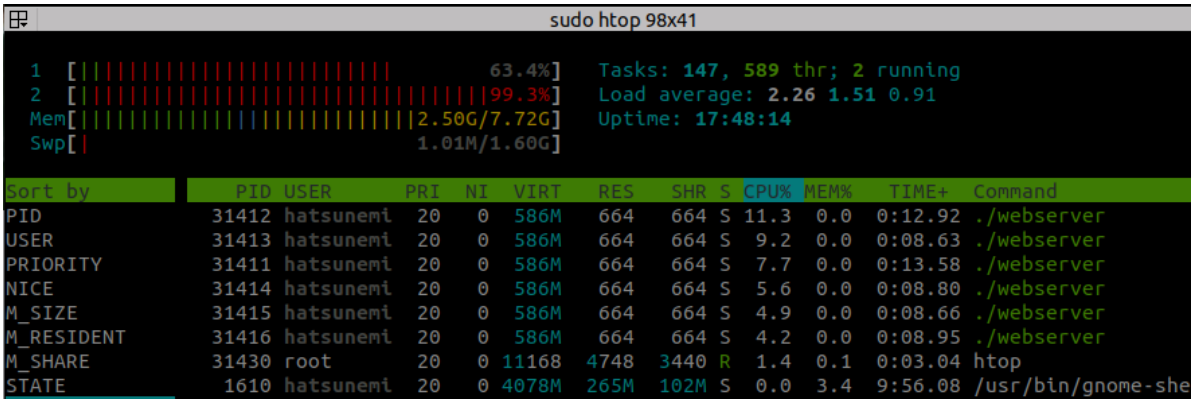
```
void *monitor_all_thpools(void *arg) {
    // ... 现有监控代码 ...

    double avg_active_threads = thpool_p->active_count ? (double)thpool_p->total_active_threads / thpool_p->active_count : 0;
    printf("最高活跃数量: %d, 最低活跃数量: %d, 平均活跃数量: %f\n", thpool_p->max_active_threads, thpool_p->min_active_threads, avg_active_threads);

    // ...
}
```

```
********************************************************************************************
Read Message ThreadPool - Threads alive: 2, Threads working: 0, Job queue length: 0
Read File ThreadPool - Threads alive: 4, Threads working: 0, Job queue length: 0
Send Message ThreadPool - Threads alive: 2, Threads working: 0, Job queue length: 0
********************************************************************************************
Read Message ThreadPool - 平均活跃时间: 76.500000 秒, 平均阻塞时间: 26.500000 秒
Read File ThreadPool - 平均活跃时间: 3.500000 秒, 平均阻塞时间: 99.500000 秒
Send Message ThreadPool - 平均活跃时间: 13.000000 秒, 平均阻塞时间: 89.500000 秒
********************************************************************************************
Read Message ThreadPool - Max active threads: 2, Min active threads: 1, Avg active threads: 1.83
Read File ThreadPool - Max active threads: 4, Min active threads: 1, Avg active threads: 1.17
Send Message ThreadPool - Max active threads: 2, Min active threads: 1, Avg active threads: 1.61
```

d. 利用相关系统命令来监测系统的I/O、内存、CPU等设备性能。

- **htop**: 类似于 `top`，但用户界面更友好，提供了更多视觉上的信息。上左区显示了CPU、物理内存和交换分区的信息；上右区显示了任务数量、平均负载和连接运行时间等信息；下方为进程区域，显示出当前系统中的所有进程。





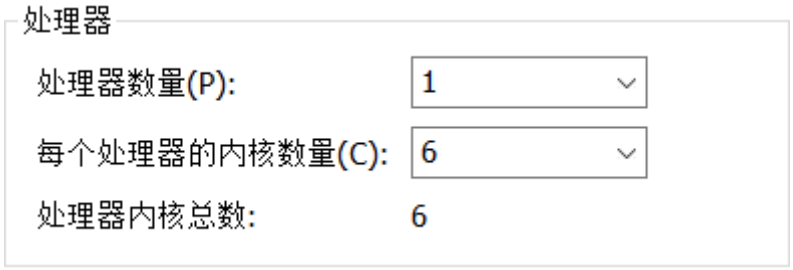
题目3.

> 通过上述的性能参数和系统命令，对Web服务程序进行逻辑分析，发现当前程序存在性能瓶颈的原因。进而通过控制各个线程池中的线程数量和消息队列长度，来改善此程序的性能。

在设置线程池的大小时，需要考虑几个重要因素，包括CPU密集型任务与I/O密集型任务的区别、线程池数量对性能的影响，以及最佳实践和动态调整的方法。

1. **CPU密集型任务与I/O密集型任务**：CPU密集型任务主要消耗CPU资源，如内存中的数据排序。这种任务的线程数通常设置为CPU核心数加1（N+1），这样可以保证即使有线程暂停，也能充分利用CPU的空闲时间。而I/O密集型任务，因为大部分时间花在等待I/O操作完成上，所以线程数可以设置为CPU核心数的两倍（2N）。
2. **线程池数量的影响**：线程池设置过大或过小都可能带来问题。设置过小可能导致任务在队列中排队等待，导致CPU未充分利用；设置过大可能导致大量线程同时争夺CPU资源，增加上下文切换的成本，从而降低效率。

测试环境



测试数据

`./http_load -parallel 1000 -seconds 5 urls.txt`

Experiment Number	Thread Pool 1 Size	Thread Pool 2 Size	Thread Pool 3 Size	Fetches/sec
1	1	1	1	25790
2	2	1	1	21799.8
3	1	2	1	26264.4
4	1	3	1	16053.5
5	1	1	2	25912
6	1	1	3	24757.2
7	1	1	4	20798
8	1	2	3	15203.7
9	1	3	2	19871.2
10	2	2	2	16545.1