小白视角：一文读懂社长的TinyWebServer

2020-06-02

小白视角：一文读懂社长的TinyWebServer(Raw_Version)

TL;DR 感谢社长，社长牛逼。

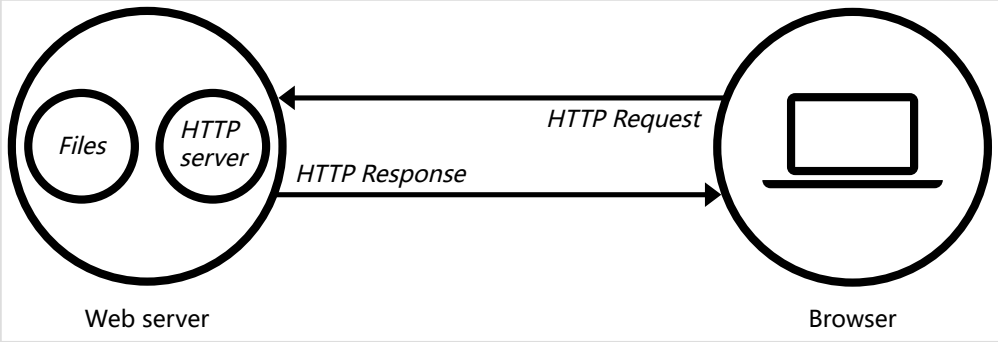
目录

预备知识：通读一遍《Linux高性能服务器编程》— 游双著

本文将带你从本人的小白视角，从头到尾彻底理解社长的TinyWebServer项目，明白每个部分都发生了什么，你的服务器程序又是如何处理，如何响应（response）来自客户端的用户请求的（requests）。

1. 什么是Web Server（网络服务器）

一个Web Server就是一个服务器软件（程序），或者是运行这个服务器软件的硬件（计算机）。其主要功能是通过HTTP协议与客户端（通常是浏览器（Browser））进行通信，来接收，存储，处理来自客户端的HTTP请求，并对其请求做出HTTP响应，返回给客户端其请求的内容（文件、网页等）或返回一个Error信息。



https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server

2. 用户如何与你的Web服务器进行通信

通常用户使用Web浏览器与相应服务器进行通信。在浏览器中键入“域名”或“IP地址:端口号”，浏览器则先将你的域名解析成相应的IP地址或者直接根据你的IP地址向对应的Web服务器发送一个HTTP请求。这一过程首先要通过TCP协议的三次握手建立与目标Web服务器的连接，然后HTTP协议生成针对目标Web服务器的HTTP请求报文，通过TCP、IP等协议发送到目标Web服务器上。

3. Web服务器如何接收客户端发来的HTTP请求报文呢?

Web服务器端通过 `socket` 监听来自用户的请求。

```
#include <sys/socket.h>
#include <netinet/in.h>
/* 创建监听socket文件描述符 */
int listenfd = socket(PF_INET, SOCK_STREAM, 0);
/* 创建监听socket的TCP/IP的IPV4 socket地址 */
struct sockaddr_in address;
bzero(&address, sizeof(address));
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);  /* INADDR_ANY：将套接字绑定到所有可用的
address.sin_port = htons(port);

int flag = 1;
/* SO_REUSEADDR 允许端口被重复使用 */
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &flag, sizeof(flag));
/* 绑定socket和它的地址 */
ret = bind(listenfd, (struct sockaddr*)&address, sizeof(address));
/* 创建监听队列以存放待处理的客户连接，在这些客户连接被accept()之前 */
ret = listen(listenfd, 5);
```

远端的很多用户会尝试去 `connect()` 这个Web Server上正在 `listen` 的这个 `port` ，而监听到的这些连接会排队等待被 `accept()` 。由于用户连接请求是随机到达的异步事件，每当监听socket（ `listenfd` ）`listen` 到新的客户连接并且放入监听队列，我们都需要告诉我们的Web服务器有连接来了， `accept` 这个连接，并分配一个逻辑单元来处理这个用户请求。而且，我们在处理这个请求的同时，还需要继续监听其他客户的请求并分配其另一逻辑单元来处理（并发，同时处理多个事件，后面会提到使用线程池实现并发）。这里，服务器通过**epoll**这种I/O复用技术（还有select和poll）来实现对监听socket（ `listenfd` ）和连接socket（客户请求）的同时监听。注意I/O复用虽然可以同时监听多个文件描述符，但是它本身是阻塞的，并且当有多个文件描述符同时就绪的时候，如果不采取额外措施，程序则只能按顺序处理其中就绪的每一个文件描述符，所以为提高效率，我们将在这部分通过线程池来实现并发（多线程并发），为每个就绪的文件描述符分配一个逻辑单元（线程）来处理。

```
#include <sys/epoll.h>
/* 将fd上的EPOLLIN和EPOLLET事件注册到epollfd指示的epoll内核事件中 */
void addfd(int epollfd, int fd, bool one_shot) {
    epoll_event event;
    event.data.fd = fd;
    event.events = EPOLLIN | EPOLLET | EPOLLRDHUP;
    /* 针对connfd，开启EPOLLONESHOT，因为我们希望每个socket在任意时刻都只被一个线程处理
    if(one_shot)
        event.events |= EPOLLONESHOT;
    epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &event);
    setnonblocking(fd);
}
```

```
/* 创建一个额外的文件描述符来唯一标识内核中的epoll事件表 */
int epollfd = epoll_create(5);
/* 用于存储epoll事件表中就绪事件的event数组 */
epoll_event events[MAX_EVENT_NUMBER];
/* 主线程往epoll内核事件表中注册监听socket事件，当listen到新的客户连接时，listenfd变为就
addfd(epollfd, listenfd, false);
/* 主线程调用epoll_wait等待一组文件描述符上的事件，并将当前所有就绪的epoll_event复制到ev
int number = epoll_wait(epollfd, events, MAX_EVENT_NUMBER, -1);
/* 然后我们遍历这一数组以处理这些已经就绪的事件 */
for(int i = 0; i < number; ++i) {
    int sockfd = events[i].data.fd;  // 事件表中就绪的socket文件描述符
    if(sockfd == listenfd) {  // 当listen到新的用户连接，listenfd上则产生就绪事件
        struct sockaddr_in client_address;
        socklen_t client_addrlength = sizeof(client_address);
        /* ET模式 */
        while(1) {
            /* accept()返回一个新的socket文件描述符用于send()和recv() */
            int connfd = accept(listenfd, (struct sockaddr *) &client_address, &cl
            /* 并将connfd注册到内核事件表中 */
            users[connfd].init(connfd, client_address);
            /* ... */
        }
    }
    else if(events[i].events & (EPOLLRDHUP | EPOLLHUP | EPOLLERR)) {
        // 如有异常，则直接关闭客户连接，并删除该用户的timer
        /* ... */
    }
    else if(events[i].events & EPOLLIN) {
        /* 当这一sockfd上有可读事件时，epoll_wait通知主线程。*/
        if(users[sockfd].read()) { /* 主线程从这一sockfd循环读取数据，直到没有更多数据
            pool->append(users + sockfd);  /* 然后将读取到的数据封装成一个请求对象并插
            /* ... */
        }
        else
            /* ... */
    }
    else if(events[i].events & EPOLLOUT) {
        /* 当这一sockfd上有可写事件时，epoll_wait通知主线程。主线程往socket上写入服务器
        if(users[sockfd].write()) {
            /* ... */
        }
        else
            /* ... */
    }
}
```

服务器程序通常需要处理三类事件：I/O事件，信号及定时事件。有两种事件处理模式：

- Reactor模式：要求主线程（I/O处理单元）只负责监听文件描述符上是否有事件发生（可读、可写），若有，则立即通知工作线程（逻辑单元），将socket可读可写事件放入请求队列，交给工作线程处理。
- Proactor模式：将所有的I/O操作都交给主线程和内核来处理（进行读、写），工作线程仅负责处理逻辑，如主线程读完成后 `users[sockfd].read()`，选择一个工作线程来处理客户请求 `pool->append(users + sockfd)`。

通常使用同步I/O模型（如 `epoll_wait` ）实现Reactor，使用异步I/O（如 `aio_read` 和 `aio_write` ）实现Proactor。但在此项目中，我们使用的是**同步I/O模拟的Proactor**事件处理模式。那么什么是同步I/O，什么是异步I/O呢?
廖雪峰：异步IO一节给出解释

- 同步（阻塞）I/O：在一个线程中，CPU执行代码的速度极快，然而，一旦遇到IO操作，如读写文件、发送网络数据时，就需要等待IO操作完成，才能继续进行下一步操作。这种情况称为同步IO。
- 异步（非阻塞）I/O：当代码需要执行一个耗时的IO操作时，它只发出IO指令，并不等待IO结果，然后就去执行其他代码了。一段时间后，当IO返回结果时，再通知CPU进行处理。

Linux下有三种IO复用方式：epoll，select和poll，为什么用epoll，它和其他两个有什么区别呢？（参考StackOverflow上的一个问题：Why is epoll faster than select?）

- 对于select和poll来说，所有文件描述符都是在用户态被加入其文件描述符集合的，每次调用都需要将整个集合拷贝到内核态；epoll则将整个文件描述符集合维护在内核态，每次添加文件描述符的时候都需要执行一个系统调用。系统调用的开销是很大的，而且在有很多短期活跃连接的情况下，epoll可能会慢于select和poll由于这些大量的系统调用开销。
- select使用线性表描述文件描述符集合，文件描述符有上限；poll使用链表来描述；epoll底层通过红黑树来描述，并且维护一个ready list，将事件表中已经就绪的事件添加到这里，在使用epoll_wait调用时，仅观察这个list中有没有数据即可。
- select和poll的最大开销来自内核判断是否有文件描述符就绪这一过程：每次执行select或poll调用时，它们会采用遍历的方式，遍历整个文件描述符集合去判断各个文件描述符是否有活动；epoll则不需要去以这种方式检查，当有活动产生时，会自动触发epoll回调函数通知epoll文件描述符，然后内核将这些就绪的文件描述符放到之前提到的ready list中等待epoll_wait调用后被处理。
- select和poll都只能工作在相对低效的LT模式下，而epoll同时支持LT和ET模式。
- 综上，当监测的fd数量较小，且各个fd都很活跃的情况下，建议使用select和poll；当监听的fd数量较多，且单位时间仅部分fd活跃的情况下，使用epoll会明显提升性能。

`Epoll` 对文件操作符的操作有两种模式：LT（电平触发）和ET（边缘触发），二者的区别在于当你调用 `epoll_wait` 的时候内核里面发生了什么：

- LT（电平触发）：类似 `select`，LT会去遍历在epoll事件表中每个文件描述符，来观察是否有我们感兴趣的事件发生，如果有（触发了该文件描述符上的回调函数），`epoll_wait` 就会以非阻塞的方式返回。若该epoll事件没有被处理完（没有返回 `EWOULDBLOCK`），该事件还会被后续的 `epoll_wait` 再次触发。
- ET（边缘触发）：ET在发现有我们感兴趣的事件发生后，立即返回，并且 `sleep` 这一事件的 `epoll_wait`，不管该事件有没有结束。

在使用ET模式时，必须要保证该文件描述符是非阻塞的（确保在没有数据可读时，该文件描述符不会一直阻塞）；并且每次调用 `read` 和 `write` 的时候都必须等到它们返回 `EWOULDBLOCK`（确保所有数据都已读完或写完）。

4. Web服务器如何处理以及响应接收到的HTTP请求报文呢?

该项目使用线程池（半同步半反应堆模式）并发处理用户请求，主线程负责读写，工作线程（线程池中的线程）负责处理逻辑（HTTP请求报文的解析等等）。通过之前的代码，我们将 `listenfd` 上到达的 `connection` 通过 `accept()` 接收，并返回一个新的socket文件描述符 `connfd` 用于和用户通信，并对用户请求返回响应，同时将这个 `connfd` 注册到内核事件表中，等用户发来请求报文。这个过程是：通过 `epoll_wait` 发现这个 `connfd` 上有可读事件了（`EPOLLIN`），主线程就将这个HTTP的请求报文读进这个连接socket的读缓存中 `users[sockfd].read()`，然后将该任务对象（指针）插入线程池的请求队列中 `pool->append(users + sockfd);`，线程池的实现还需要依靠**锁机制**以及**信号量**机制来实现线程同步，保证操作的原子性。
在线程池部分做几点解释，然后大家去看代码的时候就更容易看懂了：

- 所谓线程池，就是一个 `pthread_t` 类型的普通数组，通过 `pthread_create()` 函数创建 `m_thread_number` 个**线程**，用来执行 `worker()` 函数以执行每个请求处理函数（HTTP请求的 `process` 函数），通过 `pthread_detach()` 将线程设置成脱离态（detached）后，当这一线程运行结束时，它的资源会被系统自动回收，而不再需要在其它线程中对其进行 `pthread_join()` 操作。
- 操作工作队列一定要加锁（`locker`），因为它被所有线程共享。
- 我们用信号量来标识请求队列中的请求数，通过 `m_queuestat.wait();` 来等待一个请求队列中待处理的HTTP请求，然后交给线程池中的空闲线程来处理。

为什么要使用线程池?
当你需要限制你应用程序中同时运行的线程数时，线程池非常有用。因为启动一个新线程会带来性能开销，每个线程也会为其堆栈分配一些内存等。为了任务的并发执行，我们可以将这些任务任务传递到线程池，而不是为每个任务动态开启一个新的线程。

:star:线程池中的线程数量是依据什么确定的?

> 在StackOverflow上面发现了一个还不错的回答，意思是：
> 线程池中的线程数量最直接的限制因素是中央处理器(CPU)的处理器(processors/cores)的数量 N：
> 如果你的CPU是4-cores的，对于CPU密集型的任务(如视频剪辑等消耗CPU计算资源的任务)来说，那线程池中的线程数量最好也设置为4（或者+1防止其他因素造成的线程阻塞）；对于IO密集型的任务，一般要多于CPU的核数，因为线程间竞争的不是CPU的计算资源而是IO，IO的处理一般较慢，多于cores数的线程将为CPU争取更多的任务，不至在线程处理IO的过程造成CPU空闲导致资源浪费，公式： 最佳线程数 = CPU当前可使用的Cores数 * 当前CPU的利用率 * (1 + CPU等待时间 / CPU处理时间)（还有回答里面提到的Amdahl准则可以了解一下）

OK，接下来说说每个 read() 后的HTTP请求是如何被处理的，我们直接看这个处理HTTP请求的入口函数：

```
1   void http_conn::process() {
2       HTTP_CODE read_ret = process_read();
3       if(read_ret == NO_REQUEST) {
4           modfd(m_epollfd, m_sockfd, EPOLLIN);
5           return;
6       }
7       bool write_ret = process_write(read_ret);
8       if(!write_ret)
9           close_conn();
10      modfd(m_epollfd, m_sockfd, EPOLLOUT);
11  }
```

首先， process_read() ，也就是对我们读入该 connfd 读缓冲区的请求报文进行解析。
HTTP请求报文由请求行（request line）、请求头部（header）、空行和请求数据四个部分组成有。两种请求报文（例子来自社长的详解文章：
GET（Example）

```
1   GET /562f25980001b1b106000338.jpg HTTP/1.1
2   Host:img.mukewang.com
3   User-Agent:Mozilla/5.0 (Windows NT 10.0; WOW64)
4   AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.106 Safari/537.36
5   Accept:image/webp,image/*,*/*;q=0.8
6   Referer:http://www.imooc.com/
7   Accept-Encoding:gzip, deflate, sdch
8   Accept-Language:zh-CN,zh;q=0.8
9   空行
10  请求数据为空
```

POST（Example，注意POST的请求内容不为空）

```
1  POST / HTTP1.1
2  Host:www.wrox.com
3  User-Agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.507
4  Content-Type:application/x-www-form-urlencoded
5  Content-Length:40
6  Connection: Keep-Alive
7  空行
8  name=Professional%20Ajax&publisher=Wiley
```

GET和POST的区别

- 最直观的区别就是GET把参数包含在URL中，POST通过request body传递参数。
- GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
- GET请求在URL中传送的参数是有长度限制。（大多数）浏览器通常都会限制url长度在2K个字节，而（大多数）服务器最多处理64K大小的url。
- GET产生一个TCP数据包；POST产生两个TCP数据包。对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；而对于POST，浏览器先发送header，服务器响应100（指示信息—表示请求已接收，继续处理）continue，浏览器再发送data，服务器响应200 ok（返回数据）。

参考社长的文章：最新版Web服务器项目详解 - 05 http连接处理（中）

`process_read()` 函数的作用就是将类似上述例子的请求报文进行解析，因为用户的请求内容包含在这个请求报文里面，只有通过解析，知道用户请求的内容是什么，是请求图片，还是视频，或是其他请求，我们根据这些请求返回相应的HTML页面等。项目中使用**主从状态机**的模式进行解析，从状态机（`parse_line`）负责读取报文的一行，主状态机负责对该行数据进行解析，主状态机内部调用从状态机，从状态机驱动主状态机。每解析一部分都会将整个请求的 `m_check_state` 状态改变，状态机也就是根据这个状态来进行不同部分的解析跳转的：

- `parse_request_line(text)`，解析请求行，也就是GET中的 `GET /562f25980001b1b106000338.jpg HTTP/1.1` 这一行，或者POST中的 `POST / HTTP1.1` 这一行。通过请求行的解析我们可以判断该HTTP请求的类型（GET/POST），而请求行中最重要的部分就是 `URL` 部分，我们会将这部分保存下来用于后面的生成HTTP响应。
- `parse_headers(text);`，解析请求头部，GET和POST中 空行 以上，请求行以下的部分。
- `parse_content(text);`，解析请求数据，对于GET来说这部分是空的，因为这部分内容已经以明文的方式包含在了请求行中的 `URL` 部分了；只有POST的这部分是有数据的，项目中的这部分数据为用户名和密码，我们会根据这部分内容做登录和校验，并涉及到与数据库的连接。

OK，经过上述解析，当得到一个完整的，正确的HTTP请求时，就到了 do_request 代码部分，我们需要首先对GET请求和不同POST请求（登录，注册，请求图片，视频等等）做不同的预处理，然后分析目标文件的属性，若目标文件存在、对所有用户可读且不是目录时，则使用 mmap 将其映射到内存地址 m_file_address 处，并告诉调用者获取文件成功。

抛开 mmap 这部分，先来看看这些不同请求是怎么来的：
假设你已经搭好了你的HTTP服务器，然后你在本地浏览器中键入 localhost:9000 ，然后回车，这时候你就给你的服务器发送了一个GET请求，什么都没做，然后服务器端就会解析你的这个HTTP请求，然后发现是个GET请求，然后返回给你一个静态HTML页面，也就是项目中的 judge.html 页面，那POST请求怎么来的呢？这时你会发现，返回的这个 judge 页面中包含着一些 新用户 和 已有账号 这两个 button 元素，当你用鼠标点击这个 button 时，你的浏览器就会向你的服务器发送一个POST请求，服务器段通过检查 action 来判断你的POST请求类型是什么，进而做出不同的响应。

```
/* judge.html */
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>WebServer</title>
    </head>
    <body>
    <br/>
    <br/>
    <div align="center"><font size="5"> <strong>欢迎访问</strong></font></div>
    <br/>
        <br/>
        <form action="0" method="post">
            <div align="center"><button type="submit">新用户</button></div>
        </form>
        <br/>
        <form action="1" method="post">
            <div align="center"><button type="submit" >已有账号</button></div>
        </form>

        </div>
    </body>
</html>
```

5. 数据库连接池是如何运行的

在处理用户注册，登录请求的时候，我们需要将这些用户的用户名和密码保存下来用于新用户的注册及老用户的登录校验，相信每个人都体验过，当你在一个网站上注册一个用户时，应该经常会遇到“您的用

户名已被使用”，或者在登录的时候输错密码了网页会提示你“您输入的用户名或密码有误”等等类似情况，这种功能是服务器端通过用户键入的用户名密码和数据库中已记录下来的用户名密码数据进行校验实现的。若每次用户请求我们都需要新建一个数据库连接，请求结束后我们释放该数据库连接，当用户请求连接过多时，这种做法过于低效，所以类似**线程池**的做法，我们构建一个数据库连接池，预先生成一些数据库连接放在那里供用户请求使用。

(找不到 `mysql/mysql.h` 头文件的时候，需要安装一个库文件： `sudo apt install libmysqlclient-dev`)

我们首先看单个数据库连接是如何生成的：

1. 使用 `mysql_init()` 初始化连接
2. 使用 `mysql_real_connect()` 建立一个到mysql数据库的连接
3. 使用 `mysql_query()` 执行查询语句
4. 使用 `result = mysql_store_result(mysql)` 获取结果集
5. 使用 `mysql_num_fields(result)` 获取查询的列数， `mysql_num_rows(result)` 获取结果集的行数
6. 通过 `mysql_fetch_row(result)` 不断获取下一行，然后循环输出
7. 使用 `mysql_free_result(result)` 释放结果集所占内存
8. 使用 `mysql_close(conn)` 关闭连接

对于一个数据库连接池来讲，就是预先生成多个这样的数据库连接，然后放在一个链表中，同时维护最大连接数 `MAX_CONN` ， 当前可用连接数 `FREE_CONN` 和当前已用连接数 `CUR_CONN` 这三个变量。同样注意在对连接池操作时（获取，释放），要用到锁机制，因为它被所有线程共享。

6. 什么是CGI校验

OK，弄清楚了数据库连接池的概念及实现方式，我们继续回到第4部分，对用户的登录及注册等POST请求，服务器是如何做校验的。当点击 新用户 按钮时，服务器对这个POST请求的响应是：返回用户一个登录界面；当你在用户名和密码框中输入后，你的POST请求报文中会连同你的用户名密码一起发给服务器，然后我们拿着你的用户名和密码在数据库连接池中取出一个连接用于 `mysql_query()` 进行查询，逻辑很简单，同步线程校验 `SYNSQL` 方式相信大家都能明白，但是这里社长又给出了其他两种校验方式，CGI什么的，就很容易让小白一时摸不到头脑，接下来就简单解释一下CGI是什么。

CGI（通用网关接口），它是一个运行在Web服务器上的程序，在编译的时候将相应的 `.cpp` 文件编程成 `.cgi` 文件并在主程序中调用即可（通过社长的 `makefile` 文件内容也可以看出）。这些CGI程序通常通过客户在其浏览器上点击一个 `button` 时运行。这些程序通常用来执行一些信息搜索、存储等任务，而且通常会生成一个动态的HTML网页来响应客户的HTTP请求。我们可以发现项目中的 `sign.cpp` 文件就是我们的CGI程序，将用户请求中的用户名和密码保存在一个 `id_passwd.txt` 文件中，通过将数据库中的用户名和密码存到一个 `map` 中用于校验。在主程序中通过 `execl(m_real_file, &flag, name, password, NULL);` 这句命令来执行这个CGI文件，这里CGI程序仅用于校验，并未直接返回给用户响

应。这个CGI程序的运行通过多进程来实现，根据其返回结果判断校验结果（使用 pipe 进行父子进程的通信，子进程将校验结果写到pipe的写端，父进程在读端读取）。

7. 生成HTTP响应并返回给用户

通过以上操作，我们已经对读到的请求做好了处理，然后也对目标文件的属性作了分析，若目标文件存在、对所有用户可读且不是目录时，则使用 mmap 将其映射到内存地址 m_file_address 处，并告诉调用者获取文件成功 FILE_REQUEST 。 接下来要做的就是根据读取结果对用户做出响应了，也就是到了 process_write(read_ret); 这一步，该函数根据 process_read() 的返回结果来判断应该返回给用户什么响应，我们最常见的就是 404 错误了，说明客户请求的文件不存在，除此之外还有其他类型的请求出错的响应，具体的可以去百度。然后呢，假设用户请求的文件存在，而且已经被 mmap 到 m_file_address 这里了，那么我们就将做如下写操作，将响应写到这个 connfd 的写缓存 m_write_buf 中去：

```
case FILE_REQUEST: {
    add_status_line(200, ok_200_title);
    if(m_file_stat.st_size != 0) {
        add_headers(m_file_stat.st_size);
        m_iv[0].iov_base = m_write_buf;
        m_iv[0].iov_len = m_write_idx;
        m_iv[1].iov_base = m_file_address;
        m_iv[1].iov_len = m_file_stat.st_size;
        m_iv_count = 2;
        bytes_to_send = m_write_idx + m_file_stat.st_size;
        return true;
    }
    else {
        const char* ok_string = "<html><body></body></html>";
        add_headers(strlen(ok_string));
        if(!add_content(ok_string))
            return false;
    }
}
```

首先将 状态行 写入写缓存， 响应头 也是要写进 connfd 的写缓存（HTTP类自己定义的，与socket无关）中的，对于请求的文件，我们已经直接将其映射到 m_file_address 里面，然后将该 connfd 文件描述符上修改为 EPOLLOUT （可写）事件，然后 epoll_Wait 监测到这一事件后，使用 writev 来将响应信息和请求文件**聚集写**到**TCP Socket**本身定义的发送缓冲区（这个缓冲区大小一般是默认的，但我们也可以通过 setsockopt 来修改）中，交由内核发送给用户。OVER!

8. 服务器优化：定时器处理非活动链接

项目中，我们预先分配了 MAX_FD 个http连接对象：

```
// 预先为每个可能的客户连接分配一个http_conn对象
http_conn* users = new http_conn[MAX_FD];
```

如果某一用户 connect() 到服务器之后，长时间不交换数据，一直占用服务器端的文件描述符，导致连接资源的浪费。这时候就应该利用定时器把这些超时的非活动连接释放掉，关闭其占用的文件描述符。这种情况也很常见，当你登录一个网站后长时间没有操作该网站的网页，再次访问的时候你会发现需要重新登录。
项目中使用的是 SIGALRM信号 来实现定时器，利用 alarm 函数周期性的触发 SIGALRM 信号，信号处理函数利用管道通知主循环，主循环接收到该信号后对升序链表上所有定时器进行处理，若该段时间内没有交换数据，则将该连接关闭，释放所占用的资源。（具体请参考 Linux高性能服务器编程 第11章 定时器和社长庖丁解牛：07定时器篇），我们接下来看项目中的具体实现。

```
/* 定时器相关参数 */
static int pipefd[2];
static sort_timer_lst timer_lst

/* 每个user（http请求）对应的timer */
client_data* user_timer = new client_data[MAX_FD];
/* 每隔TIMESLOT时间触发SIGALRM信号 */
alarm(TIMESLOT);
/* 创建管道，注册pipefd[0]上的可读事件 */
int ret = socketpair(PF_UNIX, SOCK_STREAM, 0, pipefd);
/* 设置管道写端为非阻塞 */
setnonblocking(pipefd[1]);
/* 设置管道读端为ET非阻塞，并添加到epoll内核事件表 */
addfd(epollfd, pipefd[0], false);

addsig(SIGALRM, sig_handler, false);
addsig(SIGTERM, sig_handler, false);
```

alarm 函数会定期触发 SIGALRM 信号，这个信号交由 sig_handler 来处理，每当监测到有这个信号的时候，都会将这个信号写到 pipefd[1] 里面，传递给主循环：

主循环中：

```
/* 处理信号 */
else if(sockfd == pipefd[0] && (events[i].events & EPOLLIN)) {
    int sig;
```

```
        char signals[1024];
        ret = recv(pipefd[0], signals, sizeof(signals), 0);
        if(ret == -1) {
            continue;  // handle the error
        }
        else if(ret == 0) {
            continue;
        }
        else {
            for(int j = 0; j < ret; ++j) {
                switch (signals[j]) {
                    case SIGALRM: {
                        timeout = true;
                        break;
                    }
                    case SIGTERM: {
                        stop_server = true;
                    }
                }
            }
        }
    }
```

当我们在读端 `pipefd[0]` 读到这个信号的的时候，就会将 `timeout` 变量置为 `true` 并跳出循环，让 `timer_handler()` 函数取出来定时器容器上的到期任务，该定时器容器是通过升序链表来实现的，从头到尾对检查任务是否超时，若超时则调用定时器的回调函数 `cb_func()` ，关闭该socket连接，并删除其对应的定时器 `del_timer` 。

```
void timer_handler() {
    /* 定时处理任务 */
    timer_lst.tick();
    /* 重新定时以不断触发SIGALRM信号 */
    alarm(TIMESLOT);
}
```

定时器优化

这个基于升序双向链表实现的定时器存在着其固有缺点：

- 每次遍历添加和修改定时器的效率偏低(O(n))，使用最小堆结构可以降低时间复杂度降至(O(logn))。
- 每次以固定的时间间隔触发 `SIGALRM` 信号，调用 `tick` 函数处理超时连接会造成一定的触发浪费，举个例子，若当前的 `TIMESLOT=5` ，即每隔5ms触发一次 `SIGALRM` ，跳出循环执行 `tick` 函数，这

时如果当前即将超时的任务距离现在还有 20ms ，那么在这个期间， SIGALRM 信号被触发了4次， tick 函数也被执行了4次，可是在这4次中，前三次触发都是无意义的。对此，我们可以动态的设置 TIMESLOT 的值，每次将其值设置为**当前最先超时的定时器与当前时间的时间差**，这样每次调用 tick 函数，超时时间最小的定时器必然到期，并被处理，然后在从时间堆中取一个最先超时的定时器的时间与当前时间做时间差，更新 TIMESLOT 的值。

9. 服务器优化：日志

参考：

- 最新版Web服务器项目详解 - 09 日志系统（上）
- 最新版Web服务器项目详解 - 10 日志系统（下）
- muduo第五章：高效的多线程日志

日志，由服务器自动创建，并记录运行状态，错误信息，访问数据的文件。
这部分内容个人感觉相对抽象一点，涉及单例模式以及单例模式的两种实现方式：懒汉模式和恶汉模式，以及条件变量机制和生产者消费者模型。这里大概就上述提到的几点做下简单解释，具体的还是去看参考中社长的笔记。
单例模式
最常用的设计模式之一，保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。
实现思路：私有化它的构造函数，以防止外界创建单例类的对象；使用类的私有静态指针变量指向类的唯一实例，并用一个公有的静态方法获取该实例。

- **懒汉模式**：即非常懒，不用的时候不去初始化，所以在第一次被使用时才进行初始化（实例的初始化放在 getinstance 函数内部）
  - 经典的线程安全懒汉模式，使用双检测锁模式（ p == NULL 检测了两次）
  - 利用局部静态变量实现线程安全懒汉模式
- **饿汉模式**：即迫不及待，在程序运行时立即初始化（实例的初始化放在 getinstance 函数外部， getinstance 函数仅返回该唯一实例的指针）。

日志系统的运行机制

- 日志文件
  - 局部变量的懒汉模式获取实例
  - 生成日志文件，并判断同步和异步写入方式
- 同步
  - 判断是否分文件
  - 直接格式化输出内容，将信息写入日志文件

- 异步
    - 判断是否分文件
    - 格式化输出内容，将内容写入阻塞队列，创建一个写线程，从阻塞队列取出内容写入日志文件

10. 压测（非常关键）

一个服务器项目，你在本地浏览器键入 `localhost:9000` 发现可以运行无异常还不够，你需要对他进行压测（即服务器并发量测试），压测过了，才说明你的服务器比较稳定了。社长的项目是如何压测的呢?
用到了一个压测软件叫做Webbench，可以直接在社长的Gtihub里面下载，解压，然后在解压目录打开终端运行命令（ `-c` 表示客户端数， `-t` 表示时间）：

```
./webbench -c 10001 -t 5 http://127.0.0.1:9006/
```

直接解压的 `webbench-1.5` 文件夹下的 `webbench` 文件可能会因为权限问题找不到命令或者无法执行，这时你需要重新编译一下该文件即可：

```
gcc webbench.c -o webbench
```

然后我们就可以压测得到结果了（我本人电脑的用户数量 `-c` 设置为 `10500` 会造成资源不足的错误）：

```
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Benchmarking: GET http://127.0.0.1:9006/
10001 clients, running 5 sec.

Speed=1044336 pages/min, 2349459 bytes/sec.
Requests: 87028 susceed, 0 failed.
```

Webbench是什么，介绍一下原理

父进程fork若干个子进程，每个子进程在用户要求时间或默认的时间内对目标web循环发出实际访问请求，父子进程通过管道进行通信，子进程通过管道写端向父进程传递在若干次请求访问完毕后记录到的总信息，父进程通过管道读端读取子进程发来的相关信息，子进程在时间到后结束，父进程在所有子进程退出后统计并给用户显示最后的测试结果，然后退出。

11. 这个服务器的不足在哪（希望大家可以献计献策）

待完善。

12. 如何在此基础添加功能把社长的变成自己的（小声儿bb）

到目前为止，你已经把社长的项目完整的理解了一遍，甚至把各部分怎么实现的背了下来，但是这仍然是社长的项目，你没有做什么改进，你甚至没有把定时器的实现改成时间轮模式自己去动手做一做，而我们应该自己动手这个项目中的一些不足的地方，去优化它，或者再此基础上实现一些新的功能，让它多少有点自己的东西在里面。下面对此提出几点可以做的方向，后面我如果做出来的话也会更新在这里。

- 添加文件上传功能，实现与服务器的真正交互。
- 试着调用摄像头，来实现一些更有趣的功能。

参考资料

1. 《Linux高性能服务器编程》
2. 社长本人的庖丁解牛
3. Web Server(Wikipedia)
4. Beej's Guide to Network Programming 简体中文版
5. Blocking I/O, Nonblocking I/O, And Epoll
6. Scalable I/O: Events- Vs Multithreading-based
7. CGI Programming for C Programmers

‹ RL_Importance_Sampling

DL_Federated_Learning ›

© 2020 ♥ Hugo

Powered by Hexo | Theme - NexT.Pisces