最新版Web服务器项目详解 - 04 http连接处理（上）

原创 互联网猿 两猿社 2020-04-21 14:30

互联网猿|两猿社

本文内容

在服务器项目中，http请求的处理与响应至关重要，关系到用户界面的跳转与反馈。这里，社长将其分为上、中、下三个部分来讲解，具体的：

- 上篇，梳理基础知识，结合代码分析http类及请求接收
- 中篇，结合代码分析请求报文解析
- 下篇，结合代码分析请求报文响应

基础知识方面，包括epoll、HTTP报文格式、状态码和有限状态机。

代码分析方面，首先对服务器端处理http请求的全部流程进行简要介绍，然后结合代码对http类及请求接收进行详细分析。

epoll

epoll涉及的知识较多，这里仅对API和基础知识作介绍。更多资料请查阅资料，或查阅 游双的Linux高性能服务器编程 第9章 I/O复用

epoll_create函数

```
1  #include <sys/epoll.h>
2  int epoll_create(int size)
3
```

创建一个指示epoll内核事件表的文件描述符，该描述符将用作其他epoll系统调用的第一个参数，size不起作用。

epoll_ctl函数

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)

```

该函数用于操作内核事件表监控的文件描述符上的事件：注册、修改、删除

- epfd：为epoll_creat的句柄
- op：表示动作，用3个宏来表示：
  - EPOLL_CTL_ADD (注册新的fd到epfd)，
  - EPOLL_CTL_MOD (修改已经注册的fd的监听事件)，
  - EPOLL_CTL_DEL (从epfd删除一个fd)；
- event：告诉内核需要监听的事件

上述event是epoll_event结构体指针类型，表示内核所监听的事件，具体定义如下：

```
struct epoll_event {
__uint32_t events; /* Epoll events */
epoll_data_t data; /* User data variable */
};
```

- events描述事件类型，其中epoll事件类型有以下几种
  - EPOLLIN：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）
  - EPOLLOUT：表示对应的文件描述符可以写
  - EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）
  - EPOLLERR：表示对应的文件描述符发生错误
  - EPOLLHUP：表示对应的文件描述符被挂断；
  - EPOLLET：将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)而言的
  - EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

epoll_wait函数

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)

```

该函数用于等待所监控文件描述符上有事件的产生，返回就绪的文件描述符个数

- events：用来存内核得到事件的集合，

- maxevents：告之内核这个events有多大，这个maxevents的值不能大于创建epoll_create()时的size，
- timeout：是超时时间
  - -1：阻塞
  - 0：立即返回，非阻塞
  - >0：指定毫秒
- 返回值：成功返回有多少文件描述符就绪，时间到时返回0，出错返回-1

select/poll/epoll

- 调用函数
  - select和poll都是一个函数，epoll是一组函数
- 文件描述符数量
  - select通过线性表描述文件描述符集合，文件描述符有上限，一般是1024，但可以修改源码，重新编译内核，不推荐
  - poll是链表描述，突破了文件描述符上限，最大可以打开文件的数目
  - epoll通过红黑树描述，最大可以打开文件的数目，可以通过命令ulimit -n number修改，仅对当前终端有效
- 将文件描述符从用户传给内核
  - select和poll通过将所有文件描述符拷贝到内核态，每次调用都需要拷贝
  - epoll通过epoll_create建立一棵红黑树，通过epoll_ctl将要监听的文件描述符注册到红黑树上
- 内核判断就绪的文件描述符
  - select和poll通过遍历文件描述符集合，判断哪个文件描述符上有事件发生
  - epoll_create时，内核除了帮我们在epoll文件系统里建了个红黑树用于存储以后epoll_ctl传来的fd外，还会再建立一个list链表，用于存储准备就绪的事件，当epoll_wait调用时，仅仅观察这个list链表里有没有数据即可。
  - epoll是根据每个fd上面的回调函数(中断函数)判断，只有发生了事件的socket才会主动的去调用 callback函数，其他空闲状态socket则不会，若是就绪事件，插入list
- 应用程序索引就绪文件描述符
  - select/poll只返回发生了事件的文件描述符的个数，若知道是哪个发生了事件，同样需要遍历
  - epoll返回的发生了事件的个数和结构体数组，结构体包含socket的信息，因此直接处理返回的数组即可
- 工作模式
  - select和poll都只能工作在相对低效的LT模式下
  - epoll则可以工作在ET高效模式，并且epoll还支持EPOLLONESHOT事件，该事件能进一步减少可读、可写和异常事件被触发的次数。
- 应用场景
  - 当所有的fd都是活跃连接，使用epoll，需要建立文件系统，红黑书和链表对于此来说，效率反而不高，不如selece和poll
  - 当监测的fd数目较小，且各个fd都比较活跃，建议使用select或者poll

- 当监测的fd数目非常大，成千上万，且单位时间只有其中的一部分fd处于就绪状态，这个时候使用epoll能够明显提升性能

ET、LT、EPOLLONESHOT

- LT水平触发模式
  - epoll_wait检测到文件描述符有事件发生，则将其通知给应用程序，应用程序可以不立即处理该事件。
  - 当下一次调用epoll_wait时，epoll_wait还会再次向应用程序报告此事件，直至被处理
- ET边缘触发模式
  - epoll_wait检测到文件描述符有事件发生，则将其通知给应用程序，应用程序必须立即处理该事件
  - 必须要一次性将数据读取完，使用非阻塞I/O，读取到出现eagain
- EPOLLONESHOT
  - 一个线程读取某个socket上的数据后开始处理数据，在处理过程中该socket上又有新数据可读，此时另一个线程被唤醒读取，此时出现两个线程处理同一个socket
  - 我们期望的是一个socket连接在任一时刻都只被一个线程处理，通过epoll_ctl对该文件描述符注册epolloneshot事件，一个线程处理socket时，其他线程将无法处理，**当该线程处理完后，需要通过epoll_ctl重置epolloneshot事件**

HTTP报文格式

HTTP报文分为请求报文和响应报文两种，每种报文必须按照特有格式生成，才能被浏览器端识别。

其中，浏览器端向服务器发送的为请求报文，服务器处理后返回给浏览器端的为响应报文。

请求报文

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。

其中，请求分为两种，GET和POST，具体的：

- **GET**

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

- **POST**

```
POST / HTTP1.1
Host:www.wrox.com
User-Agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0
Content-Type:application/x-www-form-urlencoded
Content-Length:40
Connection: Keep-Alive
空行
name=Professional%20Ajax&publisher=Wiley
```

- **请求行**，用来说明请求类型,要访问的资源以及所使用的HTTP版本。
  GET说明请求类型为GET，/562f25980001b1b106000338.jpg(URL)为要访问的资源，该行的最后一部分说明使用的是HTTP1.1版本。
- **请求头部**，紧接着请求行（即第一行）之后的部分，用来说明服务器要使用的附加信息。
  - HOST，给出请求资源所在服务器的域名。
  - User-Agent，HTTP客户端程序的信息，该信息由你发出请求使用的浏览器来定义,并且在每个请求中自动发送等。
  - Accept，说明用户代理可处理的媒体类型。
  - Accept-Encoding，说明用户代理支持的内容编码。
  - Accept-Language，说明用户代理能够处理的自然语言集。
  - Content-Type，说明实现主体的媒体类型。
  - Content-Length，说明实现主体的大小。
  - Connection，连接管理，可以是Keep-Alive或close。
- **空行**，请求头部后面的空行是必须的即使第四部分的请求数据为空，也必须有空行。
- **请求数据**也叫主体，可以添加任意的其他数据。

响应报文

HTTP响应也由四个部分组成，分别是：状态行、消息报头、空行和响应正文。

```
HTTP/1.1 200 OK
Date: Fri, 22 May 2009 06:07:21 GMT
Content-Type: text/html; charset=UTF-8
空行
<html>
      <head></head>
      <body>
            <!--body goes here-->
      </body>
</html>
```

- 状态行，由HTTP协议版本号， 状态码， 状态消息 三部分组成。
  第一行为状态行，（HTTP/1.1）表明HTTP版本为1.1版本，状态码为200，状态消息为OK。
- 消息报头，用来说明客户端要使用的一些附加信息。
  第二行和第三行为消息报头，Date:生成响应的日期和时间；Content-Type:指定了MIME类型的HTML(text/html),编码类型是UTF-8。
- 空行，消息报头后面的空行是必须的。
- 响应正文，服务器返回给客户端的文本信息。空行后面的html部分为响应正文。

HTTP状态码

HTTP有5种类型的状态码，具体的：

- 1xx：指示信息--表示请求已接收，继续处理。
- 2xx：成功--表示请求正常处理完毕。
  - 200 OK：客户端请求被正常处理。
  - 206 Partial content：客户端进行了范围请求。
- 3xx：重定向--要完成请求必须进行更进一步的操作。
  - 301 Moved Permanently：永久重定向，该资源已被永久移动到新位置，将来任何对该资源的访问都要使用本响应返回的若干个URI之一。
  - 302 Found：临时重定向，请求的资源现在临时从不同的URI中获得。
- 4xx：客户端错误--请求有语法错误，服务器无法处理请求。
  - 400 Bad Request：请求报文存在语法错误。
  - 403 Forbidden：请求被服务器拒绝。
  - 404 Not Found：请求不存在，服务器上找不到请求的资源。
- 5xx：服务器端错误--服务器处理请求出错。
  - 500 Internal Server Error：服务器在执行请求时出现错误。

有限状态机

有限状态机，是一种抽象的理论模型，它能够把有限个变量描述的状态变化过程，以可构造可验证的方式呈现出来。比如，封闭的有向图。

有限状态机可以通过if-else,switch-case和函数指针来实现，从软件工程的角度看，主要是为了封装逻辑。

带有状态转移的有限状态机示例代码。

```
STATE_MACHINE(){
    State cur_State = type_A;
    while(cur_State != type_C){
        Package _pack = getNewPackage();
        switch(){
            case type_A:
                process_pkg_state_A(_pack);
                cur_State = type_B;
                break;
            case type_B:
                process_pkg_state_B(_pack);
                cur_State = type_C;
                break;
        }
    }
}
```

该状态机包含三种状态：type_A，type_B和type_C。其中，type_A是初始状态，type_C是结束状态。

状态机的当前状态记录在cur_State变量中，逻辑处理时，状态机先通过getNewPackage获取数据包，然后根据当前状态对数据进行处理，处理完后，状态机通过改变cur_State完成状态转移。

有限状态机一种逻辑单元内部的一种高效编程方法，在服务器编程中，服务器可以根据不同状态或者消息类型进行相应的处理逻辑，使得程序逻辑清晰易懂。

http处理流程

首先对http报文处理的流程进行简要介绍，然后具体介绍http类的定义和服务器接收http请求的具体过程。

http报文处理流程

- 浏览器端发出http连接请求，主线程创建http对象接收请求并将所有数据读入对应buffer，将该对象插入任务队列，工作线程从任务队列中取出一个任务进行处理。(**本篇讲**)
- 工作线程取出任务后，调用process_read函数，通过主、从状态机对请求报文进行解析。(**中篇讲**)
- 解析完之后，跳转do_request函数生成响应报文，通过process_write写入buffer，返回给浏览器端。(**下篇讲**)

http类

这一部分代码在TinyWebServer/http/http_conn.h中，主要是http类的定义。

```
class http_conn{
    public:
        //设置读取文件的名称m_real_file大小
        static const int FILENAME_LEN=200;
        //设置读缓冲区m_read_buf大小
        static const int READ_BUFFER_SIZE=2048;
        //设置写缓冲区m_write_buf大小
        static const int WRITE_BUFFER_SIZE=1024;
        //报文的请求方法，本项目只用到GET和POST
        enum METHOD{GET=0,POST,HEAD,PUT,DELETE,TRACE,OPTIONS,CONNECT,PATH};
        //主状态机的状态
        enum CHECK_STATE{CHECK_STATE_REQUESTLINE=0,CHECK_STATE_HEADER,CHECK_STATE
        //报文解析的结果
        enum HTTP_CODE{NO_REQUEST,GET_REQUEST,BAD_REQUEST,NO_RESOURCE,FORBIDDEN_R
        //从状态机的状态
        enum LINE_STATUS{LINE_OK=0,LINE_BAD,LINE_OPEN};

    public:
        http_conn(){}
        ~http_conn(){}

    public:
        //初始化套接字地址，函数内部会调用私有方法init
        void init(int sockfd,const sockaddr_in &addr);
        //关闭http连接
        void close_conn(bool real_close=true);
        void process();
        //读取浏览器端发来的全部数据
        bool read_once();
        //响应报文写入函数
        bool write();
        sockaddr_in *get_address(){
            return &m_address;
        }
        //同步线程初始化数据库读取表
        void initmysql_result();
        //CGI使用线程池初始化数据库表
        void initresultFile(connection_pool *connPool);

    private:
        void init();
        //从m_read_buf读取，并处理请求报文
        HTTP_CODE process_read();
        //向m_write_buf写入响应报文数据
        bool process_write(HTTP_CODE ret);
        //主状态机解析报文中的请求行数据
        HTTP_CODE parse_request_line(char *text);
        //主状态机解析报文中的请求头数据
        HTTP_CODE parse_headers(char *text);
        //主状态机解析报文中的请求内容
        HTTP_CODE parse_content(char *text);
        //生成响应报文
        HTTP_CODE do_request();

        //m_start_line是已经解析的字符
        //get_line用于将指针向后偏移，指向未处理的字符
        char* get_line(){return m_read_buf+m_start_line;};

        //从状态机读取一行，分析是请求报文的哪一部分
        LINE_STATUS parse_line();

        void unmap();

        //根据响应报文格式，生成对应8个部分，以下函数均由do_request调用
        bool add_response(const char* format,...);
        bool add_content(const char* content);
```

```
67          bool add_status_line(int status,const char* title);
68          bool add_headers(int content_length);
69          bool add_content_type();
70          bool add_content_length(int content_length);
71          bool add_linger();
72          bool add_blank_line();
73
74      public:
75          static int m_epollfd;
76          static int m_user_count;
77          MYSQL *mysql;
78
79      private:
80          int m_sockfd;
81          sockaddr_in m_address;
82
83          //存储读取的请求报文数据
84          char m_read_buf[READ_BUFFER_SIZE];
85          //缓冲区中m_read_buf中数据的最后一个字节的下一个位置
86          int m_read_idx;
87          //m_read_buf读取的位置m_checked_idx
88          int m_checked_idx;
89          //m_read_buf中已经解析的字符个数
90          int m_start_line;
91
92          //存储发出的响应报文数据
93          char m_write_buf[WRITE_BUFFER_SIZE];
94          //指示buffer中的长度
95          int m_write_idx;
96
97          //主状态机的状态
98          CHECK_STATE m_check_state;
99          //请求方法
100         METHOD m_method;
101
102         //以下为解析请求报文中对应的6个变量
103         //存储读取文件的名称
104         char m_real_file[FILENAME_LEN];
105         char *m_url;
106         char *m_version;
107         char *m_host;
108         int m_content_length;
109         bool m_linger;
110
111         char *m_file_address;         //读取服务器上的文件地址
112         struct stat m_file_stat;
113         struct iovec m_iv[2];         //io向量机制iovec
114         int m_iv_count;
115         int cgi;                     //是否启用的POST
116         char *m_string;                 //存储请求头数据
117         int bytes_to_send;           //剩余发送字节数
118         int bytes_have_send;         //已发送字节数
119 };
```

在http请求接收部分，会涉及到init和read_once函数，但init仅仅是对私有成员变量进行初始化，不用过多讲解。

这里，对read_once进行介绍。read_once读取浏览器端发送来的请求报文，直到无数据可读或对方关闭连接，读取到m_read_buffer中，并更新m_read_idx。

```
//循环读取客户数据，直到无数据可读或对方关闭连接
bool http_conn::read_once()
{
    if(m_read_idx>=READ_BUFFER_SIZE)
    {
        return false;
    }
    int bytes_read=0;
    while(true)
    {
        //从套接字接收数据，存储在m_read_buf缓冲区
        bytes_read=recv(m_sockfd,m_read_buf+m_read_idx,READ_BUFFER_SIZE-m_read_idx
        if(bytes_read==-1)
        {
            //非阻塞ET模式下，需要一次性将数据读完
            if(errno==EAGAIN||errno==EWOULDBLOCK)
                break;
            return false;
        }
        else if(bytes_read==0)
        {
            return false;
        }
        //修改m_read_idx的读取字节数
        m_read_idx+=bytes_read;
    }
    return true;
}
```

epoll相关代码

项目中epoll相关代码部分包括非阻塞模式、内核事件表注册事件、删除事件、重置EPOLLONESHOT事件四种。

- 非阻塞模式

```
//对文件描述符设置非阻塞
int setnonblocking(int fd)
{
    int old_option = fcntl(fd, F_GETFL);
    int new_option = old_option | O_NONBLOCK;
    fcntl(fd, F_SETFL, new_option);
    return old_option;
}
```

- 内核事件表注册新事件，开启EPOLLONESHOT，针对客户端连接的描述符，listenfd不用开启

```
void addfd(int epollfd, int fd, bool one_shot)
{
    epoll_event event;
    event.data.fd = fd;

#ifdef ET
    event.events = EPOLLIN | EPOLLET | EPOLLRDHUP;
#endif

#ifdef LT
    event.events = EPOLLIN | EPOLLRDHUP;
#endif

```

```
    if (one_shot)
        event.events |= EPOLLONESHOT;
    epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &event);
    setnonblocking(fd);
}
```

- 内核事件表删除事件

```
void removefd(int epollfd, int fd)
{
    epoll_ctl(epollfd, EPOLL_CTL_DEL, fd, 0);
    close(fd);
}
```

- 重置EPOLLONESHOT事件

```
void modfd(int epollfd, int fd, int ev)
{
    epoll_event event;
    event.data.fd = fd;

#ifdef ET
    event.events = ev | EPOLLET | EPOLLONESHOT | EPOLLRDHUP;
#endif

#ifdef LT
    event.events = ev | EPOLLONESHOT | EPOLLRDHUP;
#endif

    epoll_ctl(epollfd, EPOLL_CTL_MOD, fd, &event);
}
```

服务器接收http请求

浏览器端发出http连接请求，主线程创建http对象接收请求并将所有数据读入对应buffer，将该对象插入任务队列，工作线程从任务队列中取出一个任务进行处理。

```
//创建MAX_FD个http类对象
http_conn* users=new http_conn[MAX_FD];

//创建内核事件表
epoll_event events[MAX_EVENT_NUMBER];
epollfd = epoll_create(5);
assert(epollfd != -1);

//将listenfd放在epoll树上
addfd(epollfd, listenfd, false);

//将上述epollfd赋值给http类对象的m_epollfd属性
http_conn::m_epollfd = epollfd;

while (!stop_server)
{
    //等待所监控文件描述符上有事件的产生
    int number = epoll_wait(epollfd, events, MAX_EVENT_NUMBER, -1);
    if (number < 0 && errno != EINTR)
    {
        break;
    }
    //对所有就绪事件进行处理
    for (int i = 0; i < number; i++)
    {
        int sockfd = events[i].data.fd;

        //处理新到的客户连接
        if (sockfd == listenfd)
        {
            struct sockaddr_in client_address;
            socklen_t client_addrlength = sizeof(client_address);
//LT水平触发
#ifdef LT
            int connfd = accept(listenfd, (struct sockaddr *)&client_address, &cli
            if (connfd < 0)
            {
                continue;
            }
            if (http_conn::m_user_count >= MAX_FD)
            {
                show_error(connfd, "Internal server busy");
                continue;
            }
            users[connfd].init(connfd, client_address);
#endif

//ET非阻塞边缘触发
#ifdef ET
            //需要循环接收数据
            while (1)
            {
                int connfd = accept(listenfd, (struct sockaddr *)&client_address,
                if (connfd < 0)
                {
                    break;
                }
                if (http_conn::m_user_count >= MAX_FD)
                {
                    show_error(connfd, "Internal server busy");
                    break;
                }
                users[connfd].init(connfd, client_address);
            }
            continue;
#endif
```

```
67          }
68
69          //处理异常事件
70          else if (events[i].events & (EPOLLRDHUP | EPOLLHUP | EPOLLERR))
71          {
72              //服务器端关闭连接
73          }
74
75          //处理信号
76          else if ((sockfd == pipefd[0]) && (events[i].events & EPOLLIN))
77          {
78          }
79
80          //处理客户连接上接收到的数据
81          else if (events[i].events & EPOLLIN)
82          {
83              //读入对应缓冲区
84              if (users[sockfd].read_once())
85              {
86                  //若监测到读事件，将该事件放入请求队列
87                  pool->append(users + sockfd);
88              }
89              else
90              {
91                 //服务器关闭连接
92              }
93          }
94
95      }
96  }
```

如果本文对你有帮助， 阅读原文 star一下服务器项目，我们需要你的星星^_^.

完。

上一篇
最新版Web服务器项目详解 - 03 半同步半反应堆线程池（下）

下一篇
最新版Web服务器项目详解 - 05 http连接处理（中）

阅读原文