最新版Web服务器项目详解 - 05 http连接处理（中）

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

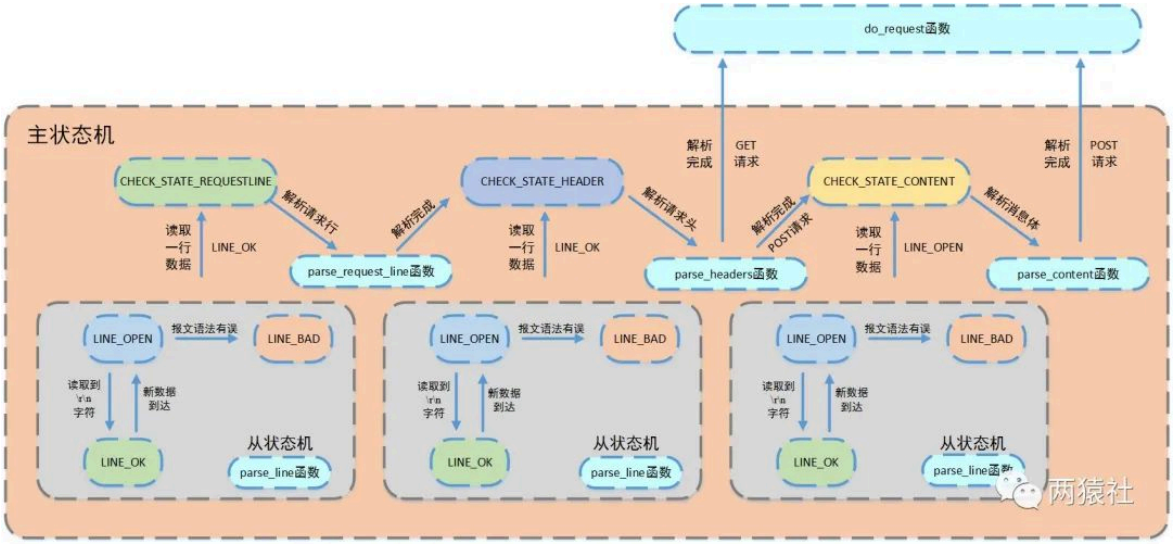
上篇，我们对http连接的基础知识、服务器接收请求的处理流程进行了介绍，本篇将结合流程图和代码分别对状态机和服务器解析请求报文进行详解。

流程图部分，描述主、从状态机调用关系与状态转移过程。

代码部分，结合代码对http请求报文的解析进行详解。

流程图与状态机

从状态机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。



主状态机

三种状态，标识解析位置。

- CHECK_STATE_REQUESTLINE，解析请求行
- CHECK_STATE_HEADER，解析请求头
- CHECK_STATE_CONTENT，解析消息体，仅用于解析POST请求

从状态机

三种状态，标识解析一行的读取状态。

- LINE_OK，完整读取一行
- LINE_BAD，报文语法有误
- LINE_OPEN，读取的行不完整

代码分析-http报文解析

上篇中介绍了服务器接收http请求的流程与细节，简单来讲，浏览器端发出http连接请求，服务器端主线程创建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后，工作线程从任务队列中取出一个任务进行处理。

各子线程通过process函数对任务进行处理，调用process_read函数和process_write函数分别完成报文解析与报文响应两个任务。

```
void http_conn::process()
{
    HTTP_CODE read_ret=process_read();

    //NO_REQUEST，表示请求不完整，需要继续接收请求数据
    if(read_ret==NO_REQUEST)
    {
        //注册并监听读事件
        modfd(m_epollfd,m_sockfd,EPOLLIN);
        return;
    }
```

```
12
13      //调用process_write完成报文响应
14      bool write_ret=process_write(read_ret);
15      if(!write_ret)
16      {
17          close_conn();
18      }
19      //注册并监听写事件
20      modfd(m_epollfd,m_sockfd,EPOLLOUT);
21  }
```

本篇将对**报文解析的流程和process_read函数细节**进行详细介绍。

HTTP_CODE含义

表示HTTP请求的处理结果，在头文件中初始化了八种情形，在报文解析时只涉及到四种。

- NO_REQUEST
  - 请求不完整，需要继续读取请求报文数据
- GET_REQUEST
  - 获得了完整的HTTP请求
- BAD_REQUEST
  - HTTP请求报文有语法错误
- INTERNAL_ERROR
  - 服务器内部错误，该结果在主状态机逻辑switch的default下，一般不会触发

解析报文整体流程

process_read通过while循环，将主从状态机进行封装，对报文的每一行进行循环处理。

- 判断条件
  - 主状态机转移到CHECK_STATE_CONTENT，该条件涉及解析消息体
  - 从状态机转移到LINE_OK，该条件涉及解析请求行和请求头部
  - 两者为或关系，当条件为真则继续循环，否则退出
- 循环体
  - 从状态机读取数据
  - 调用get_line函数，通过m_start_line将从状态机读取数据间接赋给text
  - 主状态机解析text

```
//m_start_line是行在buffer中的起始位置，将该位置后面的数据赋给text
//此时从状态机已提前将一行的末尾字符\r\n变为\0\0，所以text可以直接取出完整的行进行解析
char* get_line(){
    return m_read_buf+m_start_line;
}


http_conn::HTTP_CODE http_conn::process_read()
{
    //初始化从状态机状态、HTTP请求解析结果
    LINE_STATUS line_status=LINE_OK;
    HTTP_CODE ret=NO_REQUEST;
    char* text=0;

    //这里为什么要写两个判断条件？第一个判断条件为什么这样写？
    //具体的在主状态机逻辑中会讲解。

    //parse_line为从状态机的具体实现
    while((m_check_state==CHECK_STATE_CONTENT && line_status==LINE_OK)||((line_sta
    {
        text=get_line();

        //m_start_line是每一个数据行在m_read_buf中的起始位置
        //m_checked_idx表示从状态机在m_read_buf中读取的位置
        m_start_line=m_checked_idx;

        //主状态机的三种状态转移逻辑
        switch(m_check_state)
        {
            case CHECK_STATE_REQUESTLINE:
            {
                //解析请求行
                ret=parse_request_line(text);
                if(ret==BAD_REQUEST)
                    return BAD_REQUEST;
                break;
            }
            case CHECK_STATE_HEADER:
            {
                //解析请求头
                ret=parse_headers(text);
                if(ret==BAD_REQUEST)
                    return BAD_REQUEST;

                //完整解析GET请求后，跳转到报文响应函数
                else if(ret==GET_REQUEST)
                {
                    return do_request();
                }
                break;
            }
            case CHECK_STATE_CONTENT:
            {
                //解析消息体
                ret=parse_content(text);

                //完整解析POST请求后，跳转到报文响应函数
                if(ret==GET_REQUEST)
                    return do_request();

                //解析完消息体即完成报文解析，避免再次进入循环，更新line_status
                line_status=LINE_OPEN;
                break;
            }
            default:
            return INTERNAL_ERROR;
```

```
        }
    }
    return NO_REQUEST;
}
```

从状态机逻辑

上一篇的基础知识讲解中，对于HTTP报文的讲解遗漏了一点细节，在这里作为补充。

在HTTP报文中，每一行的数据由\r\n作为结束字符，空行则是仅仅是字符\r\n。因此，可以通过查找\r\n将报文拆解成单独的行进行解析，项目中便是利用了这一点。

从状态机负责读取buffer中的数据，将每行数据末尾的\r\n置为\0\0，并更新从状态机在buffer中读取的位置m_checked_idx，以此来驱动主状态机解析。

- 从状态机从m_read_buf中逐字节读取，判断当前字节是否为\r
  - 接下来的字符是\n，将\r\n修改成\0\0，将m_checked_idx指向下一行的开头，则返回LINE_OK
  - 接下来达到了buffer末尾，表示buffer还需要继续接收，返回LINE_OPEN
  - 否则，表示语法错误，返回LINE_BAD
- 当前字节不是\r，判断是否是\n（**一般是上次读取到\r就到了buffer末尾，没有接收完整，再次接收时会出现这种情况**）
  - 如果前一个字符是\r，则将\r\n修改成\0\0，将m_checked_idx指向下一行的开头，则返回LINE_OK
- 当前字节既不是\r，也不是\n
  - 表示接收不完整，需要继续接收，返回LINE_OPEN

```
//从状态机，用于分析出一行内容
//返回值为行的读取状态，有LINE_OK,LINE_BAD,LINE_OPEN

//m_read_idx指向缓冲区m_read_buf的数据末尾的下一个字节
//m_checked_idx指向从状态机当前正在分析的字节
http_conn::LINE_STATUS http_conn::parse_line()
{
    char temp;
    for(;m_checked_idx<m_read_idx;++m_checked_idx)
    {
        //temp为将要分析的字节
        temp=m_read_buf[m_checked_idx];

        //如果当前是\r字符，则有可能会读取到完整行
        if(temp=='\r'){

            //下一个字符达到了buffer结尾，则接收不完整，需要继续接收
            if((m_checked_idx+1)==m_read_idx)
                return LINE_OPEN;
            //下一个字符是\n，将\r\n改为\0\0
            else if(m_read_buf[m_checked_idx+1]=='\n'){
                m_read_buf[m_checked_idx++]='\0';
                m_read_buf[m_checked_idx++]='\0';
                return LINE_OK;
            }
            //如果都不符合，则返回语法错误
```

```
            return LINE_BAD;
        }

        //如果当前字符是\n，也有可能读取到完整行
        //一般是上次读取到\r就到buffer末尾了，没有接收完整，再次接收时会出现这种情况
        else if(temp=='\n')
        {
            //前一个字符是\r，则接收完整
            if(m_checked_idx>1&&m_read_buf[m_checked_idx-1]=='\r')
            {
                m_read_buf[m_checked_idx-1]='\0';
                m_read_buf[m_checked_idx++]='\0';
                return LINE_OK;
            }
            return LINE_BAD;
        }
    }

    //并没有找到\r\n，需要继续接收
    return LINE_OPEN;
}
```

主状态机逻辑

主状态机初始状态是CHECK_STATE_REQUESTLINE，通过调用从状态机来驱动主状态机，在主状态机进行解析前，从状态机已经将每一行的末尾\r\n符号改为\0\0，以便于主状态机直接取出对应字符串进行处理。

- CHECK_STATE_REQUESTLINE
  - 主状态机的初始状态，调用parse_request_line函数解析请求行
  - 解析函数从m_read_buf中解析HTTP请求行，获得请求方法、目标URL及HTTP版本号
  - 解析完成后主状态机的状态变为CHECK_STATE_HEADER

```
//解析http请求行，获得请求方法，目标url及http版本号
http_conn::HTTP_CODE http_conn::parse_request_line(char *text)
{
    //在HTTP报文中，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，其中各个部分
    //请求行中最先含有空格和\t任一字符的位置并返回
    m_url=strpbrk(text," \t");

    //如果没有空格或\t，则报文格式有误
    if(!m_url)
    {
        return BAD_REQUEST;
    }

    //将该位置改为\0，用于将前面数据取出
    *m_url++='\0';

    //取出数据，并通过与GET和POST比较，以确定请求方式
    char *method=text;
    if(strcasecmp(method,"GET")==0)
        m_method=GET;
    else if(strcasecmp(method,"POST")==0)
    {
        m_method=POST;
        cgi=1;
    }
    else
        return BAD_REQUEST;

    //m_url此时跳过了第一个空格或\t字符，但不知道之后是否还有
    //将m_url向后偏移，通过查找，继续跳过空格和\t字符，指向请求资源的第一个字符
    m_url+=strspn(m_url," \t");

    //使用与判断请求方式的相同逻辑，判断HTTP版本号
    m_version=strpbrk(m_url," \t");
    if(!m_version)
        return BAD_REQUEST;
    *m_version++='\0';
    m_version+=strspn(m_version," \t");

    //仅支持HTTP/1.1
    if(strcasecmp(m_version,"HTTP/1.1")!=0)
        return BAD_REQUEST;

    //对请求资源前7个字符进行判断
    //这里主要是有些报文的请求资源中会带有http://，这里需要对这种情况进行单独处理
    if(strncasecmp(m_url,"http://",7)==0)
    {
        m_url+=7;
        m_url=strchr(m_url,'/');
    }

    //同样增加https情况
    if(strncasecmp(m_url,"https://",8)==0)
    {
        m_url+=8;
        m_url=strchr(m_url,'/');
    }

    //一般的不会带有上述两种符号，直接是单独的/或/后面带访问资源
    if(!m_url||m_url[0]!='/')
        return BAD_REQUEST;

    //当url为/时，显示欢迎界面
    if(strlen(m_url)==1)
        strcat(m_url,"judge.html");

```

```
67      //请求行处理完毕，将主状态机转移处理请求头
68      m_check_state=CHECK_STATE_HEADER;
69      return NO_REQUEST;
70  }
```

解析完请求行后，主状态机继续分析请求头。在报文中，请求头和空行的处理使用的同一个函数，这里通过判断当前的text首位是不是\0字符，若是，则表示当前处理的是空行，若不是，则表示当前处理的是请求头。

- CHECK_STATE_HEADER
  - 调用parse_headers函数解析请求头部信息
  - 判断是空行还是请求头，若是空行，进而判断content-length是否为0，如果不是0，表明是POST请求，则状态转移到CHECK_STATE_CONTENT，否则说明是GET请求，则报文解析结束。
  - 若解析的是请求头部字段，则主要分析connection字段，content-length字段，其他字段可以直接跳过，各位也可以根据需求继续分析。
  - connection字段判断是keep-alive还是close，决定是长连接还是短连接
  - content-length字段，这里用于读取post请求的消息体长度

```
1   //解析http请求的一个头部信息
2   http_conn::HTTP_CODE http_conn::parse_headers(char *text)
3   {
4       //判断是空行还是请求头
5       if(text[0]=='\0')
6       {
7           //判断是GET还是POST请求
8           if(m_content_length!=0)
9           {
10              //POST需要跳转到消息体处理状态
11              m_check_state=CHECK_STATE_CONTENT;
12              return NO_REQUEST;
13          }
14          return GET_REQUEST;
15      }
16      //解析请求头部连接字段
17      else if(strncasecmp(text,"Connection:",11)==0)
18      {
19          text+=11;
20
21          //跳过空格和\t字符
22          text+=strspn(text," \t");
23          if(strcasecmp(text,"keep-alive")==0)
24          {
25              //如果是长连接，则将linger标志设置为true
26              m_linger=true;
27          }
28      }
29      //解析请求头部内容长度字段
30      else if(strncasecmp(text,"Content-length:",15)==0)
31      {
32          text+=15;
33          text+=strspn(text," \t");
34          m_content_length=atol(text);
35      }
36      //解析请求头部HOST字段
37      else if(strncasecmp(text,"Host:",5)==0)
38      {
```

```
        text+=5;
        text+=strspn(text," \t");
        m_host=text;
    }
    else{
        printf("oop!unknow header: %s\n",text);
    }
    return NO_REQUEST;
}
```

如果仅仅是GET请求，如项目中的欢迎界面，那么主状态机只设置之前的两个状态足矣。

因为在上篇推文中我们曾说道，GET和POST请求报文的区别之一是有无消息体部分，GET请求没有消息体，当解析完空行之后，便完成了报文的解析。

但后续的登录和注册功能，为了避免将用户名和密码直接暴露在URL中，我们在项目中改用了POST请求，将用户名和密码添加在报文中作为消息体进行了封装。

为此，我们需要在解析报文的部分添加解析消息体的模块。

```
while((m_check_state==CHECK_STATE_CONTENT && line_status==LINE_OK)||((line_status=p
```

那么，这里的判断条件为什么要写成这样呢？

在GET请求报文中，每一行都是\r\n作为结束，所以对报文进行拆解时，仅用从状态机的状态line_status=parse_line())==LINE_OK语句即可。

但，在POST请求报文中，消息体的末尾没有任何字符，所以不能使用从状态机的状态，这里转而使用主状态机的状态作为循环入口条件。

那后面的&& line_status==LINE_OK又是为什么？

解析完消息体后，报文的完整解析就完成了，但此时主状态机的状态还是CHECK_STATE_CONTENT，也就是说，符合循环入口条件，还会再次进入循环，这并不是我们所希望的。

为此，增加了该语句，并在完成消息体解析后，将line_status变量更改为LINE_OPEN，此时可以跳出循环，完成报文解析任务。

- CHECK_STATE_CONTENT
  - 仅用于解析POST请求，调用parse_content函数解析消息体
  - 用于保存post请求消息体，为后面的登录和注册做准备

```
//判断http请求是否被完整读入
http_conn::HTTP_CODE http_conn::parse_content(char *text)
{
    //判断buffer中是否读取了消息体
    if(m_read_idx>=(m_content_length+m_checked_idx)){

        text[m_content_length]='\0';

        //POST请求中最后为输入的用户名和密码
        m_string = text;

```

```
12              return GET_REQUEST;
13          }
14          return NO_REQUEST;
15      }
```

状态机和HTTP报文解析是项目中最繁琐的部分，这次我们一举解决掉它，希望对各位小伙伴在理解项目的过程中有所帮助。

下篇，我们将对HTTP报文响应进行详解。

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。

Web服务器-原始版本 13

Web服务器-原始版本 · 目录

上一篇
最新版Web服务器项目详解 - 04 http连接处理（上）

下一篇
最新版Web服务器项目详解 - 06 http连接处理（下）

阅读原文

项目搞成现在这个样子，我有责任

两猿社