最新版Web服务器项目详解 - 08 定时器处理非活动连接（下）

原创 互联网猿 两猿社 2020-04-22 11:50

本文内容

定时器处理非活动连接模块，主要分为两部分，其一为定时方法与信号通知流程，其二为定时器及其容器设计、定时任务的处理。

本篇对第二部分进行介绍，具体的涉及到定时器设计、容器设计、定时任务处理函数和使用定时器。

定时器设计，将连接资源和定时事件等封装起来，具体包括连接资源、超时时间和回调函数，这里的回调函数指向定时事件。

定时器容器设计，将多个定时器串联组织起来统一处理，具体包括升序链表设计。

定时任务处理函数，该函数封装在容器类中，具体的，函数遍历升序链表容器，根据超时时间，处理对应的定时器。

代码分析-使用定时器，通过代码分析，如何在项目中使用定时器。

定时器设计

项目中将连接资源、定时事件和超时时间封装为定时器类，具体的，

- 连接资源包括客户端套接字地址、文件描述符和定时器
- 定时事件为回调函数，将其封装起来由用户自定义，这里是删除非活动socket上的注册事件，并关闭

- 定时器超时时间 = 浏览器和服务器连接时刻 + 固定时间(TIMESLOT)，可以看出，定时器使用绝对时间作为超时值，这里alarm设置为5秒，连接超时为15秒。

```
//连接资源结构体成员需要用到定时器类
//需要前向声明
class util_timer;

//连接资源
struct client_data
{
    //客户端socket地址
    sockaddr_in address;

    //socket文件描述符
    int sockfd;

    //定时器
    util_timer* timer;
};

//定时器类
class util_timer
{
public:
    util_timer() : prev( NULL ), next( NULL ){}

public:
    //超时时间
    time_t expire;
    //回调函数
    void (*cb_func)( client_data* );
    //连接资源
    client_data* user_data;
    //前向定时器
    util_timer* prev;
    //后继定时器
    util_timer* next;
};
```

定时事件，具体的，从内核事件表删除事件，关闭文件描述符，释放连接资源。

```
//定时器回调函数
void cb_func(client_data *user_data)
{
    //删除非活动连接在socket上的注册事件
    epoll_ctl(epollfd, EPOLL_CTL_DEL, user_data->sockfd, 0);
    assert(user_data);

    //关闭文件描述符
    close(user_data->sockfd);

    //减少连接数
    http_conn::m_user_count--;
}
```

定时器容器设计

项目中的定时器容器为带头尾结点的升序双向链表，具体的为每个连接创建一个定时器，将其添加到链表中，并按照超时时间升序排列。执行定时任务时，将到期的定时器从链表中删除。

从实现上看，主要涉及双向链表的插入，删除操作，其中添加定时器的事件复杂度是O(n),删除定时器的事件复杂度是O(1)。

升序双向链表主要逻辑如下，具体的，

- 创建头尾节点，其中头尾节点没有意义，仅仅统一方便调整
- add_timer函数，将目标定时器添加到链表中，添加时按照升序添加
  - 若当前链表中只有头尾节点，直接插入
  - 否则，将定时器按升序插入
- adjust_timer函数，当定时任务发生变化,调整对应定时器在链表中的位置
  - 客户端在设定时间内有数据收发,则当前时刻对该定时器重新设定时间，这里只是往后延长超时时间
  - 被调整的目标定时器在尾部，或定时器新的超时值仍然小于下一个定时器的超时，不用调整
  - 否则先将定时器从链表取出，重新插入链表
- del_timer函数将超时的定时器从链表中删除
  - 常规双向链表删除结点

```cpp
//定时器容器类
class sort_timer_lst
{
public:
    sort_timer_lst() : head( NULL ), tail( NULL ) {}
    //常规销毁链表
    ~sort_timer_lst()
    {
        util_timer* tmp = head;
        while( tmp )
        {
            head = tmp->next;
            delete tmp;
            tmp = head;
        }
    }

    //添加定时器，内部调用私有成员add_timer
    void add_timer( util_timer* timer )
    {
        if( !timer )
        {
            return;
        }
        if( !head )
        {
            head = tail = timer;
            return;
        }

        //如果新的定时器超时时间小于当前头部结点
        //直接将当前定时器结点作为头部结点
        if( timer->expire < head->expire )
        {
            timer->next = head;
            head->prev = timer;
            head = timer;
            return;
        }

        //否则调用私有成员，调整内部结点
        add_timer( timer, head );
    }

    //调整定时器，任务发生变化时，调整定时器在链表中的位置
    void adjust_timer( util_timer* timer )
    {
        if( !timer )
        {
            return;
        }
        util_timer* tmp = timer->next;

        //被调整的定时器在链表尾部
        //定时器超时值仍然小于下一个定时器超时值，不调整
        if( !tmp || ( timer->expire < tmp->expire ) )
        {
            return;
        }

        //被调整定时器是链表头结点，将定时器取出，重新插入
        if( timer == head )
        {
            head = head->next;
            head->prev = NULL;
            timer->next = NULL;
```

```
67              add_timer( timer, head );
68          }
69
70          //被调整定时器在内部，将定时器取出，重新插入
71          else
72          {
73              timer->prev->next = timer->next;
74              timer->next->prev = timer->prev;
75              add_timer( timer, timer->next );
76          }
77      }
78
79      //删除定时器
80      void del_timer( util_timer* timer )
81      {
82          if( !timer )
83          {
84              return;
85          }
86
87          //链表中只有一个定时器，需要删除该定时器
88          if( ( timer == head ) && ( timer == tail ) )
89          {
90              delete timer;
91              head = NULL;
92              tail = NULL;
93              return;
94          }
95
96          //被删除的定时器为头结点
97          if( timer == head )
98          {
99              head = head->next;
100             head->prev = NULL;
101             delete timer;
102             return;
103         }
104
105         //被删除的定时器为尾结点
106         if( timer == tail )
107         {
108             tail = tail->prev;
109             tail->next = NULL;
110             delete timer;
111             return;
112         }
113
114         //被删除的定时器在链表内部，常规链表结点删除
115         timer->prev->next = timer->next;
116         timer->next->prev = timer->prev;
117         delete timer;
118     }
119
120 private:
121     //私有成员，被公有成员add_timer和adjust_time调用
122     //主要用于调整链表内部结点
123     void add_timer( util_timer* timer, util_timer* lst_head )
124     {
125         util_timer* prev = lst_head;
126         util_timer* tmp = prev->next;
127
128         //遍历当前结点之后的链表，按照超时时间找到目标定时器对应的位置，常规双向链表插入
129         while( tmp )
130         {
131             if( timer->expire < tmp->expire )
132             {
```

```
                prev->next = timer;
                timer->next = tmp;
                tmp->prev = timer;
                timer->prev = prev;
                break;
            }
            prev = tmp;
            tmp = tmp->next;
        }

        //遍历完发现，目标定时器需要放到尾结点处
        if( !tmp )
        {
            prev->next = timer;
            timer->prev = prev;
            timer->next = NULL;
            tail = timer;
        }

    }

private:
    //头尾结点
    util_timer* head;
    util_timer* tail;
};
```

定时任务处理函数

使用统一事件源，SIGALRM信号每次被触发，主循环中调用一次定时任务处理函数，处理链表容器中到期的定时器。

具体的逻辑如下，

- 遍历定时器升序链表容器，从头结点开始依次处理每个定时器，直到遇到尚未到期的定时器
- 若当前时间小于定时器超时时间，跳出循环，即未找到到期的定时器
- 若当前时间大于定时器超时时间，即找到了到期的定时器，执行回调函数，然后将它从链表中删除，然后继续遍历

```
//定时任务处理函数
void tick()
{
    if( !head )
    {
        return;
    }

    //获取当前时间
    time_t cur = time( NULL );
    util_timer* tmp = head;

    //遍历定时器链表
    while( tmp )
    {
        //链表容器为升序排列
        //当前时间小于定时器的超时时间，后面的定时器也没有到期
        if( cur < tmp->expire )
        {
```

```
            break;
        }

        //当前定时器到期，则调用回调函数，执行定时事件
        tmp->cb_func( tmp->user_data );

        //将处理后的定时器从链表容器中删除，并重置头结点
        head = tmp->next;
        if( head )
        {
            head->prev = NULL;
        }
        delete tmp;
        tmp = head;
    }
}
```

代码分析-如何使用定时器

服务器首先创建定时器容器链表，然后用统一事件源将异常事件，读写事件和信号事件统一处理，根据不同事件的对应逻辑使用定时器。

具体的，

- 浏览器与服务器连接时，创建该连接对应的定时器，并将该定时器添加到链表上
- 处理异常事件时，执行定时事件，服务器关闭连接，从链表上移除对应定时器
- 处理定时信号时，将定时标志设置为true
- 处理读事件时，若某连接上发生读事件，将对应定时器向后移动，否则，执行定时事件
- 处理写事件时，若服务器通过某连接给浏览器发送数据，将对应定时器向后移动，否则，执行定时事件

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

//创建定时器容器链表
static sort_timer_lst timer_lst;

//创建连接资源数组
client_data *users_timer = new client_data[MAX_FD];

//超时默认为False
bool timeout = false;

//alarm定时触发SIGALRM信号
alarm(TIMESLOT);

while (!stop_server)
{
    int number = epoll_wait(epollfd, events, MAX_EVENT_NUMBER, -1);
    if (number < 0 && errno != EINTR)
    {
        break;
    }

    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)
        {
            //初始化客户端连接地址
            struct sockaddr_in client_address;
            socklen_t client_addrlength = sizeof(client_address);

            //该连接分配的文件描述符
            int connfd = accept(listenfd, (struct sockaddr *)&client_address, &cl

            //初始化该连接对应的连接资源
            users_timer[connfd].address = client_address;
            users_timer[connfd].sockfd = connfd;

            //创建定时器临时变量
            util_timer *timer = new util_timer;
            //设置定时器对应的连接资源
            timer->user_data = &users_timer[connfd];
            //设置回调函数
            timer->cb_func = cb_func;

            time_t cur = time(NULL);
            //设置绝对超时时间
            timer->expire = cur + 3 * TIMESLOT;
            //创建该连接对应的定时器，初始化为前述临时变量
            users_timer[connfd].timer = timer;
            //将该定时器添加到链表中
            timer_lst.add_timer(timer);
        }
        //处理异常事件
        else if (events[i].events & (EPOLLRDHUP | EPOLLHUP | EPOLLERR))
        {
            //服务器端关闭连接，移除对应的定时器
            cb_func(&users_timer[sockfd]);

```

```
            util_timer *timer = users_timer[sockfd].timer;
            if (timer)
            {
                timer_lst.del_timer(timer);
            }
        }

        //处理定时器信号
        else if ((sockfd == pipefd[0]) && (events[i].events & EPOLLIN))
        {
            //接收到SIGALRM信号，timeout设置为True
        }

        //处理客户连接上接收到的数据
        else if (events[i].events & EPOLLIN)
        {
            //创建定时器临时变量，将该连接对应的定时器取出来
            util_timer *timer = users_timer[sockfd].timer;
            if (users[sockfd].read_once())
            {
                //若监测到读事件，将该事件放入请求队列
                pool->append(users + sockfd);

                //若有数据传输，则将定时器往后延迟3个单位
                //对其在链表上的位置进行调整
                if (timer)
                {
                    time_t cur = time(NULL);
                    timer->expire = cur + 3 * TIMESLOT;
                    timer_lst.adjust_timer(timer);
                }
            }
            else
            {
                //服务器端关闭连接，移除对应的定时器
                cb_func(&users_timer[sockfd]);
                if (timer)
                {
                    timer_lst.del_timer(timer);
                }
            }
        }
       else if (events[i].events & EPOLLOUT)
       {
           util_timer *timer = users_timer[sockfd].timer;
           if (users[sockfd].write())
           {
                //若有数据传输，则将定时器往后延迟3个单位
                //并对新的定时器在链表上的位置进行调整
                if (timer)
                {
                    time_t cur = time(NULL);
                    timer->expire = cur + 3 * TIMESLOT;
                    timer_lst.adjust_timer(timer);
                }
            }
            else
            {
                //服务器端关闭连接，移除对应的定时器
                cb_func(&users_timer[sockfd]);
                if (timer)
                {
                    timer_lst.del_timer(timer);
                }
            }
       }
```

```
133         }
134         //处理定时器为非必须事件，收到信号并不是立马处理
135         //完成读写事件后，再进行处理
136         if (timeout)
137         {
138             timer_handler();
139             timeout = false;
140         }
141     }
```

有小伙伴问，连接资源中的address是不是有点鸡肋？

确实如此，项目中虽然对该变量赋值，但并没有用到。类似的，可以对比HTTP类中address属性，只在日志输出中用到。

但不能说这个变量没有用，因为我们可以找到客户端连接的ip地址，用它来做一些业务，比如通过ip来判断是否异地登录等等。

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。