linux三种方式

概 述

在服务器中，我们需要处理I/O事件、信号以及定时事件。所谓定时事件，即在预期的时间点触发响应的逻辑，比如定期检测客户端连接的状态。Linux中提供了三种方式来实现定时机制：

1. **socket选项SO_RCVTIMEO和SO_SNDTIMEO**
2. **SIGALRM信号**
3. **I/O复用系统调用的超时参数**

实 现

一、socket选项SO_RCVTIMEO和SO_SNDTIMEO

我们可以通过`setsockopt()`函数来设置socket接受数据与发送数据的超时时间，`SO_RCVTIMEO`和`SO_SNDTIMEO`仅对系统提供的socket API(包括send、sendmsg、recv、recvmsg、accept和connect)有效。其系统调用函数与选项的对应关系和行为如下表：

系统调用	有效选项	系统调用超时后的行为
send	SO_SNDTIMEO	返回 -1，设置 errno 为 EAGAIN 或 EWOULDBLOCK
sendmsg	SO_SNDTIMEO	返回 -1，设置 errno 为 EAGAIN 或 EWOULDBLOCK
recv	SO_RCVTIMEO	返回 -1，设置 errno 为 EAGAIN 或 EWOULDBLOCK
recvmsg	SO_RCVTIMEO	返回 -1，设置 errno 为 EAGAIN 或 EWOULDBLOCK
accept	SO_RCVTIMEO	返回 -1，设置 errno 为 EAGAIN 或 EWOULDBLOCK
connect	SO_SNDTIMEO	返回 -1，设置 errno 为 EINPROGRESS

系统调用与选项对应表

以`connect()`和`SO_SNDTIMEO`为例，示例代码如下：

```
int main() {
    //...其他流程

    int ret;
    int socket_fd = socket(PF_INET, SOCK_STREAM, 0); //建立socket

    struct timeval timeout;
    timeout.tv_sec = 10; //设置的超时时间值
    timeout.tv_usec = 0;
    socklen_t len = sizeof(timeout);
    //使用setsockopt() SO_SNDTIMEO选项 设置connect超时时间，时间类型是timeval
    ret = setsockopt(socket_fd, SOL_SOCKET, SO_SNDTIMEO, &timeout, len);

    ret = connect(socket_fd, (struct sockaddr *)&server_addr, len);
    if (ret == -1) {
        if (errno == EINPROGRESS) { //错误码为EINPROGRESS时，表示超时
            cout << "connecting timeout" << endl;
        } else { //正常的connect失败
            cout << "connecting error" << endl;
        }
    }
```

```
    //...其他流程
}
```

二、SIGALRM信号

SIGALRM信号的处理比较复杂，且有多种实现方式，会在后续的文章中介绍

三、I/O复用系统调用的超时参数

我们可以使用I/O复用的系统调用来设置超时参数，并统一处理信号和I/O事件。以I/O复用的Epoll为例，示例代码如下：

```
#define TIMEOUT 5000 //自定义超时时间

int timeout = TIMEOUT;
time_t start = time(nullptr);
time_t end = time(nullptr);

while (1) {
    start = time(nullptr);
    //通过epoll_wait()系统调用，设置超时时间为 timeout
    int number = epoll_wait(epoll_fd, events, MAX_EVENT_NUMBER, timeout);
    if (ret < 0 && errno != EINTR) {
        cout << "epoll error " << errno << endl;
        break;
    }

    if (number == 0) { //epoll_wait成功，返回0，表示超时
        //... 处理超时任务
        timeout = TIMEOUT; //重置定时时间
        continue;
    }

    end = time(nullptr);
    timeout -= (end - start) * 1000; //更新超时参数
    // 若更新后的超时时间小于等于0，说明不仅有文件描述符就绪，并且超时时间也到了
    if (timeout <= 0) {
        //... 处理超时任务
        timeout = TIMEOUT; //重置定时时间
    }

    //...处理文件描述符事件
}
```

升序链表

概 述

在之前的文章中提到，在服务器中使用**socket选项**和**I/O复用系统调用的超时参数**来处理定时事件。接下来介绍利用**SIGALRM信号**来实现定时机制。本文介绍一种简单定时器的实现——基于升序链表的定时器，来处理SIGALRM信号。后续文章会介绍高效定时器的实现，即时间轮和时间堆。

实现

1. 实现基于升序链表的定时器

示例代码通过对STL中`List<>`的封装，实现了基于升序链表的定时器；定时器类中的回调函数，也是使用C++11中的`std::function()`来实现，代码简单易懂。原书中(《Linux高性能服务器编程》 游双 著)则是手写了链表的相关操作来实现升序链表，感兴趣的可以自行学习下，这里不再展示。

```
//Time_list.h
#include <netinet/in.h> //sockaddr_in
#include <list>
#include <functional>

#define BUFFER_SIZE 64

class util_timer; //前向声明

//客户端数据
struct client_data {
    sockaddr_in address; //socket地址
    int sockfd; //socket文件描述符
    char buf[BUFFER_SIZE]; //数据缓存区
    util_timer* timer; //定时器
};

//定时器类
class util_timer {
public:
    time_t expire; //任务超时时间(绝对时间)
    std::function<void(client_data*)> callBackFunc; //回调函数
    client_data* userData; //用户数据
};

class Timer_list {

public:
    explicit Timer_list();
    ~Timer_list();

public:
    void add_timer(util_timer* timer); //添加定时器
    void del_timer(util_timer* timer); //删除定时器
    void adjust_timer(util_timer* timer); //调整定时器
    void tick(); //处理链表上到期的任务

private:
    std::list<util_timer*> m_timer_list; //定时器链表
};

//Timer_list.cpp
#include "Timer_list.h"
#include <time.h>

Timer_list::Timer_list() {
```

```
}

Timer_list::~Timer_list() {
    m_timer_list.clear();
}

void Timer_list::add_timer(util_timer* timer) { //将定时器添加到链表
    if (!timer) return;
    else {
        auto item = m_timer_list.begin();
        while (item != m_timer_list.end()) {
            if (timer->expire < (*item)->expire) {
                m_timer_list.insert(item, timer);
                return;
            }
            item++;
        }
        m_timer_list.emplace_back(timer);
    }
}

void Timer_list::del_timer(util_timer* timer) { //将定时器从链表删除
    if (!timer) return;
    else {
        auto item = m_timer_list.begin();
        while (item != m_timer_list.end()) {
            if (timer == *item) {
                m_timer_list.erase(item);
                return;
            }
            item++;
        }
    }
}

void Timer_list::adjust_timer(util_timer *timer) { //调整定时器在链表中的位置
    del_timer(timer);
    add_timer(timer);
}

void Timer_list::tick() { //SIGALRM信号触发，处理链表上到期的任务
    if (m_timer_list.empty()) return;
    time_t cur = time(nullptr);

    //检测当前定时器链表中到期的任务。
    while (!m_timer_list.empty()) {
        util_timer* temp = m_timer_list.front();
        if (cur < temp->expire) break;
        temp->callBackFunc(temp->userData);
        m_timer_list.pop_front();
    }
}
```

2. 运 用

我们可以使用上述的升序定时器链表来处理一段时间内非活动的客户端连接。服务器利用`alarm()`周期性的触发`SIGALRM`信号，主循环收到该信号后，执行定时器链表上的定时任务，即关闭非活动的客户端连接。

关键代码如下：

```
//main.cpp
#include <iostream>
#include <netinet/in.h> //sockaddr_in
#include <string.h> //memset()
#include <arpa/inet.h> //inet_pton()
#include <assert.h> //assert()
#include <sys/epoll.h> //epoll
#include <fcntl.h> //fcntl()
#include <unistd.h> //close() alarm()
#include <signal.h>
#include "Timer_list.h" //利用升序定时器链表

#define FD_LIMIT 65535 //最大客户端数量
#define MAX_EVENT_NUMBER 1024
#define TIMESLOT 10 //定时时间

using namespace std;

static int pipefd[2]; //信号通讯管道
static Timer_list timer_list; //创建升序定时器链表类
static int epoll_fd = 0;

int setnonblocking(int fd) {
    //...
}

void addfd(int epoll_fd, int sock_fd) {
    //...
}

void sig_handler(int sig) {
    //...
}

void addsig(int sig) {
    //...
}

//SIGALRM 信号的处理函数
void timer_handler() {
    timer_list.tick(); //调用升序定时器链表类的tick() 处理链表上到期的任务
    alarm(TIMESLOT); //再次发出 SIGALRM 信号
}

//定时器回调函数 删除socket上注册的事件并关闭此socket
void timer_callback(client_data* user_data) {
    epoll_ctl(epoll_fd, EPOLL_CTL_DEL, user_data->sockfd, 0);
```

```
    if (user_data) {
        close(user_data->sockfd);
        cout << "close fd : " << user_data->sockfd << endl;
    }
}

int main() {
    //... socket

    //注册信号处理
    addsig(SIGTERM); //终止进程，即kill
    addsig(SIGALRM); //计时器到期
    bool stop_server = false;

    client_data* users = new client_data[FD_LIMIT]; //客户端数据数组
    bool timeout = false;
    alarm(TIMESLOT); //开始定时

    while (!stop_server) {
        //... epoll_wait

        for (int i = 0; i < ret; ++i) {
            int sockfd = events[i].data.fd;
            if (sockfd == sock_fd) { //处理新的客户端连接
                struct sockaddr_in client_addr;
                socklen_t len = sizeof(client_addr);
                int conn_fd = accept(sock_fd, (struct sockaddr*)&client_addr,
&len);
                addfd(epoll_fd, conn_fd);

                users[conn_fd].address = client_addr;
                users[conn_fd].sockfd = conn_fd;

                //创建定时器
                util_timer* timer = new util_timer;
                timer->userData = &users[conn_fd];      //设置用户数据
                timer->callBackFunc = timer_callback;   //设置回调函数
                time_t cur = time(nullptr);
                timer->expire = cur + 3 * TIMESLOT;     //设置客户端活动时间
                users[conn_fd].timer = timer;           //绑定定时器
                timer_list.add_timer(timer);    //将定时器添加到升序链表中

            } else if ((sockfd == pipefd[0]) && (events[i].events & EPOLLIN)) {
//处理信号
                ret = recv(pipefd[0], buf, sizeof(buf), 0);
                if (ret <= 0) continue;
                else {
                    for (auto &item : buf){
                        switch (item) {
                            case SIGTERM: {
                                cout << " SIGTERM: Kill" << endl;
                                stop_server = true;
                            }
                            case SIGALRM: {
                                //用timeout来标记有定时任务
                                //先不处理，因为定时任务优先级不高，优先处理其他事件
```

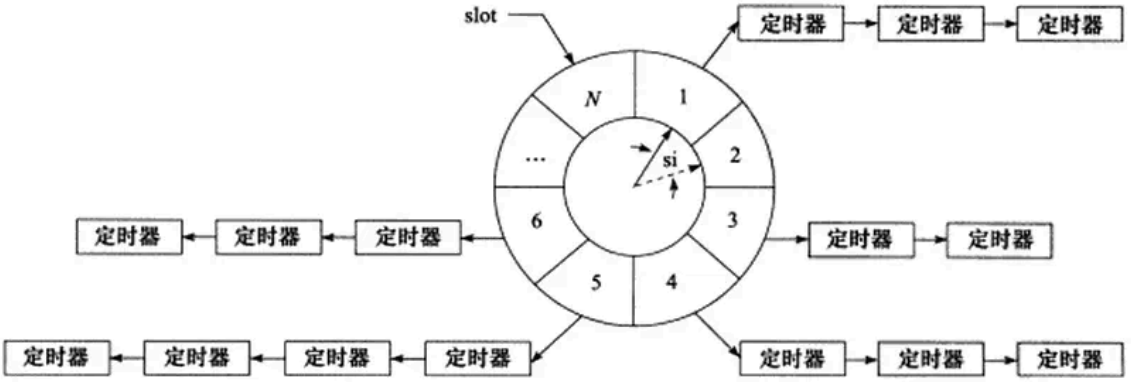
```
                                timeout = true;
                                break;
                            }
                        }
                    }
                }
            } else if (events[i].events & EPOLLIN) { //处理客户端数据
                while (1) {
                    //... recv 客户端数据

                    util_timer* timer = users[sockfd].timer; //获取对应定时器
                    if (ret < 0) {
                        if (errno != EAGAIN) { //读错误
                            timer_callback(&users[sockfd]);
                            if (timer) timer_list.del_timer(timer);
                        }
                        break;
                    } else if (ret == 0) { //对端关闭socket连接
                        timer_callback(&users[sockfd]);
                        if (timer) timer_list.del_timer(timer);
                    } else {
                        if (timer) { //客户端有数据可读，调整对应定时器的时间
                            time_t cur = time(nullptr);
                            timer->expire = cur + 3 * TIMESLOT;
                            timer_list.adjust_timer(timer);
                        }
                    }
                }
            } else {
                //...
            }
        }

        //最后处理定时任务
        if (timeout) {
            timer_handler();
            timeout = false;
        }
    }
    //... close
    return 0;
}
```

注：示例代码中，因为I/O事件优先级更高，所以服务器优先处理I/O事件，最后处理定时事件，这样会导致定时任务不能精确地按照预期时间执行。

时间轮

概 述

上篇文章中实现了基于升序链表的定时器，此定时器存在着当定时器数量变多，效率也会变低的问题。对此，我们使用时间轮在解决。贴一下书中展示的简单时间轮的结构图



时间轮

基于升序链表的定时器是将所有的定时器放到一条链表中来管理，而时间轮则是使用哈希表的思想，将定时器散列到不同的链表中，即图中时间轮的槽`(slot)`。时间轮的每次转动称为一个滴答`(tick)`，滴答的间隔时间称为槽间隔`(slot interval)`，即**心搏时间**。

实现

鉴于哈希表的思想，我们使用C++11中的`unordered_map`结构来实现时间轮，key对应时间轮中的槽，value使用`list<>`来实现槽中的定时器链表。同样的，定时器类中的回调函数，使用`std::function()`来实现。简单定时器的实现如下：

```
//TimerWheel.h
#include <netinet/in.h>
#include <list>
#include <functional>
#include <unordered_map>

#define BUFFER_SIZE 0xFFFF  //缓存区数据大小
#define TIMESLOT    1       //定时时间

class tw_timer; //前向声明

//客户端数据
struct client_data {
    sockaddr_in address; //socket地址
    int sockfd; //socket文件描述符
    char buf[BUFFER_SIZE]; //数据缓存区
    tw_timer *timer; //定时器
};

//定时器类
class tw_timer {
public:
    tw_timer(int rot, int ts) : rotation(rot), time_slot(ts){}

public:
    int rotation; //记录定时器在时间轮转多少圈后生效
    int time_slot; //记录定时器属于时间轮上的哪个槽
    std::function<void(client_data *)> callBackFunc; //回调函数
    client_data *user_data; //用户数据
};
```

```
class TimerWheel {
public:
    explicit TimerWheel();
    ~TimerWheel();

public:
    tw_timer* AddTimer(int timeout); //根据超时时间 timeout 添加定时器
    void DelTimer(tw_timer *timer); //删除定时器
    void Tick(); //心搏函数

private:
    static const int N = 60; //时间轮上槽的数量
    static const int SI = TIMESLOT; //每 1s 时间轮转动一次，即槽间隔为 1s
    int m_cur_slot; //时间轮的当前槽
    std::unordered_map<int, std::list<tw_timer *>> m_slots; //时间轮的槽，其中每个元素指向一个定时器链表
};

//TimerWheel.cpp
#include "TimerWheel.h"

TimerWheel::TimerWheel() : m_cur_slot(0) {
    for (int i = 0; i < N; ++i) {
        m_slots[i] = std::list<tw_timer *>();
    }
}

TimerWheel::~TimerWheel() {
    for (int i = 0; i < N; ++i) {
        auto iter = m_slots[i].begin();
        while (iter != m_slots[i].end()) {
            delete *iter;
            iter++;
        }
    }
    m_slots.clear();
}

tw_timer *TimerWheel::AddTimer(int timeout) {
    if (timeout < 0) return nullptr;
    int ticks = timeout < SI ? 1 : timeout / SI;
    int rotation = ticks / N; //计算待插入的定时器在时间轮转动多少圈后触发
    int ts = (m_cur_slot + (ticks % N)) % N; //计算待插入的定时器应该插入到时间轮的哪个槽
    tw_timer* timer = new tw_timer(rotation, ts);
    m_slots[ts].push_front(timer);
    return timer;
}

void TimerWheel::DelTimer(tw_timer *timer) {
    if (!timer) return;
    int ts = timer->time_slot;
    m_slots[ts].remove(timer);
    delete timer;
}
```

```
void TimerWheel::Tick() {
    auto iter = m_slots[m_cur_slot].begin();
    while (iter != m_slots[m_cur_slot].end()) {
        if ((*iter)->rotation > 0) { //定时器的 ratation 值大于0，则在这一轮中不起作用
            (*iter)->rotation--;
            iter++;
        } else { //定时器已到期，执行定时任务，最后删除该定时器
            (*iter)->callBackFunc((*iter)->user_data);
            delete *iter;
            iter = m_slots[m_cur_slot].erase(iter);
        }
    }
    m_cur_slot = ++m_cur_slot % N;
}
```

运 用

时间轮的使用与升序链表定时器的使用相似，整体代码可以参考上一篇文章中实现--运用的代码。这里展示核心的内容：

```
int main() {
    //... socket

    //...注册信号处理

    while (!stop_server) {
        //... epoll_wait

        for (int i = 0; i < ret; ++i) {
            int sockfd = events[i].data.fd;
            if (sockfd == sock_fd) { //处理新的客户端连接
                //...accept

                //创建定时器
                tw_timer *timer = m_timerWheel.AddTimer(10); //添加定时器，超时时间
10秒
                timer->user_data = &m_user[conn_fd]; //设置用户数据
                timer->callBackFunc = TimerCallBack; //设置回调函数

            } else if ((sockfd == pipefd[0]) && (events[i].events & EPOLLIN)) {
//处理信号
                //...
            } else if (events[i].events & EPOLLIN) { //处理客户端数据
                //...
            } else {
                //...
            }
        }

        //...处理定时任务
    }
    //... close
```

```
    return 0;
}
```

时间堆

概 述

在之前文章中实现的定时器，都是以固定的频率调用心搏函数`tick()`，从而处理到期的定时器任务。**时间堆**则采用另一种思路：将所有定时器中到期时间最小的值作为心搏间隔。当调用心搏函数时，此定时器任务必定到期，处理完此定时器任务后，再从所有定时器中到期时间最小的那个，将此到期时间设为下一次的心搏间隔，如此反复，实现定时。

实 现

从时间堆定时器的设计思路中不难看出，使用小根堆结构最适合处理这种方案。我们通过对STL中`priority_queue<>`的封装来实现时间堆(有关优先级队列的知识这里不做展示了)。书中则是手写了一个基于数组的小根堆来实现时间堆，感兴趣的可以自行学习下，这里也不再展示。示例代码如下：

```
//TimerHeap.h
#include <netinet/in.h>
#include <functional>
#include <vector>
#include <queue>

#define BUFFER_SIZE 0xFFFF  //缓存区数据大小
#define TIMESLOT    30      //定时时间

class heap_timer; //前向声明

// 客户端数据
struct client_data {
    sockaddr_in address;
    int sockfd;
    char buf[BUFFER_SIZE];
    heap_timer *timer;
};

// 定时器类
class heap_timer {
public:
    heap_timer(int delay) {
        expire = time(nullptr) + delay;
    }
public:
    time_t expire; //定时器生效的绝对时间
    std::function<void(client_data *)> callBackFunc; //回调函数
    client_data *user_data; //用户数据
};

struct cmp { //比较函数，实现小根堆
    bool operator () (const heap_timer* a, const heap_timer* b) {
        return a->expire > b->expire;
```

```
    }
};

class TimerHeap {
public:
    explicit TimerHeap();
    ~TimerHeap();

public:
    void AddTimer(heap_timer *timer); //添加定时器
    void DelTimer(heap_timer *timer); //删除定时器
    void Tick(); //心搏函数
    heap_timer* Top();

private:
    std::priority_queue<heap_timer *, std::vector<heap_timer *>, cmp>
m_timer_pqueue; //时间堆
};

//TimerHeap.cpp
#include "TimerHeap.h"

TimerHeap::TimerHeap() = default;

TimerHeap::~TimerHeap() {
    while (!m_timer_pqueue.empty()) {
        delete m_timer_pqueue.top();
        m_timer_pqueue.pop();
    }
}

void TimerHeap::AddTimer(heap_timer *timer) {
    if (!timer) return;
    m_timer_pqueue.emplace(timer);
}

void TimerHeap::DelTimer(heap_timer *timer) {
    if (!timer) return;
    // 将目标定时器的回调函数设为空，即延时销毁
    // 减少删除定时器的开销，但会扩大优先级队列的大小
    timer->callBackFunc = nullptr;
}

void TimerHeap::Tick() {
    time_t cur = time(nullptr);
    while (!m_timer_pqueue.empty()) {
        heap_timer *timer = m_timer_pqueue.top();
        //堆顶定时器没有到期，退出循环
        if (timer->expire > cur) break;
        //否则执行堆顶定时器中的任务
        if (timer->callBackFunc) timer->callBackFunc(timer->user_data);
        m_timer_pqueue.pop();
    }
}

heap_timer *TimerHeap::Top() {
```

```
    if (m_timer_pqueue.empty()) return nullptr;
    else return m_timer_pqueue.top();
}
```

运用

因为时间堆以最小到期时间作为心搏间隔，所以我们在服务器类中定义一个变量来记录当前是否在进行定时任务，从而来判断是否触发`SIGALRM`信号，核心代码如下：

```
// SIGALRM信号处理函数
void Server::TimerHandler() {
    m_timerHeap.Tick(); //调用时间堆的心搏函数 Tick() 处理到期的任务
    heap_timer *tmp = nullptr;
    //判断当前是否在进行定时任务，没有则触发 SIGALRM 信号
    if (!s_isAlarm && (tmp = m_timerHeap.Top())) {
        time_t delay = tmp->expire - time(nullptr);
        if (delay <= 0) delay = 1;
        alarm(delay); //触发 SIGALRM 信号
        s_isAlarm = true; //设置当前已有定时任务在进行
    }
}

// 定时器回调函数
void Server::TimerCallBack(client_data *user_data) {
    epoll_ctl(s_epollFd, EPOLL_CTL_DEL, user_data->sockfd, 0);
    if (user_data) {
        close(user_data->sockfd);
        s_clientsList.remove(user_data->sockfd);
        s_isAlarm = false; //设置当前没有进行定时任务
        cout << "Server: close socket fd : " << user_data->sockfd << endl;
    }
}

void Server::Run() {
    //...
    while (!stop_server) {
        //... epoll_wait

        for (int i = 0; i < ret; ++i) {
            int sockfd = events[i].data.fd;
            if (sockfd == sock_fd) { //处理新的客户端连接
                //...accept

                //创建定时器
                heap_timer *timer = new heap_timer(TIMESLOT);
                timer->user_data = &m_user[conn_fd];
                timer->callBackFunc = TimerCallBack;
                m_timerHeap.AddTimer(timer);
                if (!s_isAlarm) { //当前没有进行定时任务
                    s_isAlarm = true;
                    alarm(TIMESLOT); //触发 SIGALRM 信号
                }
```

```
            } else if ((sockfd == pipefd[0]) && (events[i].events & EPOLLIN)) {
//处理信号
                //...
            } else if (events[i].events & EPOLLIN) { //处理客户端数据
                //...
                heap_timer *timer = m_user[sockfd].timer;

                // 更新时间堆定时器
                if (timer) {
                    timer->expire += TIMESLOT;
                    // 更新定时器后，需要设置当前没有进行定时任务
                    // 从而在 TimerHandler() 中重新触发 SIGALRM 信号
                    s_isAlarm = false;
                }

            } else {
                //...
            }
        }
        //...处理定时任务
    }
    //... close
    return 0;
}
```

服务器程序的运行结果与之前实现的定时器类似，这里就不展示了。

作者：荏苒何从cc
链接：https://www.jianshu.com/p/e880f398530d
来源：简书
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。