joker8

libev使用方法

1. libev简介

libev是个高性能跨平台的事件驱动框架，支持io事件，超时事件，子进程状态改变通知，信号通知，文件状态改变通知，还能用来实现wait/notify机制。libev对每种监听事件都用一个ev_type类型的数据结构表示，如ev_io, ev_timer, ev_child, ev_async分别用来表示文件监听器, timeout监听器, 子进程状态监听器, 同步事件监听器.

libev支持优先级, libev一次loop收集的事件按优先级先排序, 优先级高的事件回调先执行, 优先级低的后执行, 相同优先级则按事件到达顺序执行. libev优先级从[-2, 2], 默认优先级为0，libev注册watcher的流程如下：

```
static void type_cb(EV_P_ ev_type *watcher, int revents)
{
        // callback
}

static void ev_test()
{
#ifdef EV_MULTIPLICITY
        struct ev_loop *loop;
#else
        int loop;
#endif
        ev_type *watcher;

        loop = ev_default_loop(0);
        watcher = (ev_type *)calloc(1, sizeof(*watcher));
        assert(loop && watcher);

        ev_type_init(watcher, type_cb, ...);
        ev_start(EV_A_ watcher);

        ev_run(EV_A_ 0);

        /* 资源回收 */
        ev_loop_destroy(EV_A);
        free(watcher);
}
```

libev注册watcher可以分为四个步骤：

1. 创建一个loop和watcher
2. 初始化watcher，主要设置callback函数和定义watcher的参数
3. 激活watcher
4. 启动libev，开始loop收集事件

2. ev_io

libev内部使用后端select, poll, epoll(linux专有), kqueue(drawin), port(solaris10)实现io事件监听, 用户可以指定操作系统支持的后端或者由libev自动选择使用哪个后端，如linux平台上用户可以强制指定libev使用select作为后端。libev支持单例模式和多例模式, 假设我们链接的是多例模式的libev库, 且watcher使用默认优先级0.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>
```

```c
#include <ev.h>

static void io_cb(struct ev_loop *loop, ev_io *watcher, int revents)
{
        char buf[1024] = {0};

        /* 参数watcher即注册时的watcher */
        read(watcher->fd, buf, sizeof(buf) - 1);
        fprintf(stdout, "%s\n", buf);

        ev_break(loop, EVBREAK_ALL);
}

static void io_test()
{
        struct ev_loop *loop;
        ev_io *io_watcher;

        /* 指定libev使用epoll机制，关闭环境变量对libev影响 */
        loop = ev_default_loop(EVFLAG_NOENV | EVBACKEND_EPOLL);
        io_watcher = (ev_io *)calloc(1, sizeof(*io_watcher));
        assert(("can not alloc memory", loop && io_watcher));

        /* 设置监听标准输入的可读事件和回调函数 */
        ev_io_init(io_watcher, io_cb, STDIN_FILENO, EV_READ);
        ev_io_start(loop, io_watcher);

        /* libev开启loop */
        ev_run(loop, 0);

        /* 资源回收 */
        ev_loop_destroy(loop);
        free(io_watcher);
}

int main(void)
{
        io_test();

        return 0;
}
```

ev_io_init(ev, cb, fd, event)

- ev：ev_io
- cb： 回调函数
- fd：socket，pipe等句柄
- event：事件类型(EV_READ/EV_WRITE)

Makefile

```
target := main
CC := clang
CFLAGS += -g -Wall -fPIC
LDFLAGS += -lev

src += \
        main.c
obj := $(patsubst %.c, %.o, $(src))

$(target):$(obj)
        $(CC) -o $@ $^ $(LDFLAGS)

%.o:%.c
        $(CC) -o $@ -c $< $(CFLAGS)

.PHONY:clean

clean:
        @rm -rf $(target) *.o
```

libev内部使用一个大的循环来收集各种watcher注册的事件，如果没有注册ev_timer和ev_periodic，则libev内部使用的后端采用59.743s作为超时事件，如果select作为后端，则select的超时设置为59.743s，这样可以降低cpu占用率，对一个fd可以注册的watcher数量不受限(或者说只受内存限制)，比如可以对标准输入的可读事件注册100个watcher，当有用户输入时所有100个watcher的回调都能执行(当然回调中还是只有一个read操作成功)。

3. ev_timer

ev_timer可以用来实现定时器, ev_timer不受墙上时间影响，如设置一个1小时定时器，把当前系统时间调快1小时不能让定时器立刻超时，超时依旧发生在1小时后，如下是一个简单的例子：

```
static void timer_cb(struct ev_loop *loop, ev_timer *w, int revents)
{
        fprintf(stdout, "%fs timeout\n", w->repeat);
        ev_break(loop, EVBREAK_ALL);
}

static void timer_test()
{
        struct ev_loop *loop;
        ev_timer *timer_watcher;

        loop = ev_default_loop(EVFLAG_NOENV);
        timer_watcher = calloc(1, sizeof(*timer_watcher));
        assert(("can not alloc memory", loop && timer_watcher));

        ev_timer_init(timer_watcher, timer_cb, 0., 3600.);
        ev_timer_start(loop, timer_watcher);

        ev_run(loop, 0);

        ev_loop_destroy(loop);
        free(timer_watcher);
}
```

ev_timer_init(ev, cb, ofs, iva)

- ev: ev_timer
- cb: 回调函数
- ofs，iva: 超时事件为(now + ofs + ival * N), now为当前时间，N为正整数

如果ival参数为0，则timer是一次性的定时器，超时后libev自动stop timer
可以在回调中重新设置timer超时，并重新启动timer。

```
static void timer_cb(struct ev_loop *loop, ev_timer *watcher, int revents)
{
        fprintf(stdout, "%fs timeout\n", watcher->repeat);
        watcher->repeat += 5.;
        ev_timer_again(loop, watcher);
}
```

上面介绍了libev是在一个大的循环中监听所有watcher的事件，只有ev_io类型的watcher时，libev后端以59.743s作为超时(如select超时)，这时用户注册一个3s的timer，那么libev会不会因为后端超时太长导致定时器检测非常不准呢？答案是不会，libev保证后端超时时间不大于定时器超时时间，注册一个3s timer，则libev自动调整到3s以内loop一次，这样保证timer超时能及时被检测到，同时也带来更高的cpu占用率。

4. ev_periodic

ev_timer做为定时器很方便，但是对应指定到某时刻发生超时就比较困难，比如每天00:00:00触发超时，每天08：00开灯，18：00关灯等等，ev_periodic可以很好的应付这种场景，ev_periodic基于墙上时间，所以受墙上时间影响，如注册1小时后超时的ev_peroidic，同时系统时间调快1小时，ev_periodic立马能超时。如下例子指定每天凌晨发生超时：

```
static void periodic_cb(struct ev_loop *loop, ev_periodic *watcher, int revents)
{
        fprintf(stdout, "00:00:00 now, time to sleep");
}

ev_tstamp my_schedule(ev_periodic *watcher, ev_tstamp now)
{
        time_t cur;
        struct tm tm;

        time(&cur);
        localtime_r(&cur, &tm);

        tm.tm_wday += 1;
        tm.tm_hour = 0;
        tm.tm_sec = 0;
        tm.tm_min = 0;
        tm.tm_mon -= 1;
        tm.tm_year -= 1900;

        return mktime(&tm);
}
```

```
static void periodic_test()
{
        struct ev_loop *loop;
        ev_periodic *periodic_watcher;

        loop = ev_default_loop(0);
        periodic_watcher = (ev_periodic *)calloc(1, sizeof(*periodic_watcher));
        assert(("can not alloc memory", loop && periodic_watcher));

        ev_periodic_init(periodic_watcher, periodic_cb, 0, 0, my_schedule);
        ev_periodic_start(loop, periodic_watcher);

        ev_run(loop, 0);

        ev_loop_destroy(loop);
        free(periodic_watcher);
}
```

ev_periodic_init(ev, cb, ofs, ival，schedule)

- ev：ev_periodic
- cb：回调函数
- ofs，ival：ofs + ceil((now - ofs) / ival) * ival // now表示当前时间戳
- schedule：用户自定义函数，该函数返回下一次ev_periodic超时时间

1. 如果schedule不为空，则ev_periodic超时时间为：ofs + ceil((now - ofs) / ival) * ival，表示从当前时间开始，经过ofs时间后所有能被ival整数的点，注册ev_periodic，ifs = 1, ival = 10, 当前时间为1604649458(2020-10-615:57:38)，则ev_periodic第一次经过2s就发生超时了。
2. 如果schedule存在，则ofs，ival参数被忽略，ev_periodic超时时间由schedule()返回值指定

5. ev_child

libev支持监听子进程状态变化, 如子进程退出, 内部用waitpid去实现, libev限制只能用default loop去监听子进程状态变化, 如果以ev_loop_new()创建的loop则不行, 通过ev_default_loop()创建default loop时libev内部自动注册了SIGCHILD信号处理函数, 需要在自己代码处理SIGCHILD的话， 可以在ev_default_loop()之后注册SIGCHIL处理以覆盖libev中的默认处理, 如下是一个简单的例子:

```
static void child_cb(struct ev_loop *loop, ev_child *watcher, int revents)
{
        fprintf(stdout, "pid:%d exit, status:%d\n", watcher->rpid, watcher->rstatus);
}

static void child_test()
{
        pid_t pid;
        struct ev_loop *loop;
        ev_child *child_watcher;

        switch (pid = fork()) {
                case 0:
                        sleep(5);
                        fprintf(stdout, "child_pid:%d\n", getpid());
                        exit(EXIT_SUCCESS);
                default:
                        {
                                loop = ev_default_loop(0);
                                child_watcher = (ev_child*)calloc(1, sizeof(*child_watcher));
                                assert(("can not alloc memory", loop && child_watcher));

                                ev_child_init(child_watcher, child_cb, 0, 1);
                                ev_child_start(loop, child_watcher);

                                ev_run(loop, 0);

                                /* 资源回收 */
                                ev_loop_destroy(loop);
                                free(child_watcher);
                        }
        }
}
```

ev_child_init(ev, cb, pid, trace)

- ev：ev_child

- cb：回调函数
- pid：子进程pid
- trace：设置为1
  个人觉得libev的default loop默认注册SIGCHILD处理并不好，最好还是在自己代码中做处理。

6. ev_async

可以通过libev的async来实现wait/notify机制, 用户注册多个ev_async监听器, 在其他地方调用ev_async_send()即可触发ev_async注册的回调, libev内部用eventfd(linux平台)和pipe(win32)实现, 个人觉得linux平台上直接用eventfd更完美, 如下是简单例子.

```
static void *routine(void *args)
{
        static size_t count = 0;
        ev_async *watcher = (ev_async *)args;
        struct ev_loop *loop = (struct ev_loop *)watcher->data;

        while (count++ < 10) {
                ev_async_send(loop, watcher);
                sleep(1);
        }

        return NULL;
}

static void async_cb(struct ev_loop *loop, ev_async *watcher, int revents)
{
        fprintf(stdout, "get the order, start move...\n");
}

static void async_test()
{
        pthread_t pid;
        struct ev_loop *loop;
        ev_async *async_watcher;

        loop = ev_default_loop(0);
        async_watcher = (ev_async *)calloc(1, sizeof(*async_watcher));
        assert(("can not alloc memory", loop && async_watcher));

        ev_async_init(async_watcher, async_cb);
        ev_async_start(loop, async_watcher);

        async_watcher->data = loop;
        pthread_create(&pid, NULL, routine, async_watcher);

        ev_run(loop, 0);

        /* 资源回收 */
        ev_loop_destroy(loop);
        free(async_watcher);
}
```

ev_async_init(ev，cb)

- ev：ev_async
- cb：回调函数

7. ev_prepare/ev_idle

libev每次loop收集各种事件之前都会先调用ev_prepare的回调函数(如果有的话), 如果存在比ev_idle优先级更高的监听有事件待处理, 则ev_idle的事件不会处理, 如存在优先级1,2的事件待处理, 则优先级为1的ev_idle的事件不会被处理, 只有在优先级1,2的所有事件都处理完后才会把ev_idle的事件添加到带处理的事件队列中去.

```
static void idle_cb(struct ev_loop *loop, ev_idle *watcher, int revents)
{
        fprintf(stdout, "no one has higher priority than me now\n");
        ev_idle_stop(loop, watcher);
}

static void prepare_cb(struct ev_loop *loop, ev_prepare *watcher, int revents)
{
        fprintf(stdout, "prepare_cb\n");
}

static void ev_test()
```

```
{
        struct ev_loop *loop;
        ev_io *io_watcher;
        ev_idle *idle_watcher;
        ev_prepare *prepare_watcher;

        loop = ev_default_loop(0);
        prepare_watcher = (ev_prepare *)calloc(1, sizeof(*prepare_watcher));
        idle_watcher = (ev_idle *)calloc(1, sizeof(*idle_watcher));
        io_watcher = (ev_io *)calloc(1, sizeof(*io_watcher));
        assert(("can not alloc memory", loop && prepare_watcher && io_watcher && idle_watcher)

        ev_io_init(io_watcher, io_cb, STDIN_FILENO, EV_READ);
        ev_prepare_init(prepare_watcher, prepare_cb);
        ev_idle_init(idle_watcher, idle_cb);

        ev_prepare_start(loop, prepare_watcher);
        ev_io_start(loop, io_watcher);
        ev_idle_start(loop, idle_watcher);

        ev_run(loop, 0);

        ev_loop_destroy(loop);
        free(io_watcher);
        free(idle_watcher);
        free(prepare_watcher);
}
```

ev_prepare(ev, cb)/ev_idle(ev, cb)

- ev：ev_prepare/ev_idle
- cb：回调函数

可以看到每次输入前都先有ev_prepare的回调，只有不存在优先级别idle高的事件待处理时才会处理idle的回调。

8. ev_stat

不建议使用，还不如用个定时器自己去检测文件是否改动

9. ev_fork

注册ev_fork，在libev自动检测到fork调用（开启了EVFLAG_FORKCHECK），或者用户调用ev_loop_fork()通知libev有fork调用时ev_fork回调被触发

10. ev_cleanup

注册ev_cleanup的watcher，在libev销毁时调用ev_cleanup的回调，用来做一些清理工作

11. 简单回显服务器

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <assert.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#include <ev.h>

/* client number limitation */
#define MAX_CLIENTS 1000

/* message length limitation */
#define MAX_MESSAGE_LEN (256)

#define err_message(msg) \
        do {perror(msg); exit(EXIT_FAILURE);} while(0)

/* record the number of clients */
static int client_number;

static int create_serverfd(char const *addr, uint16_t u16port)
{
        int fd;
```

```
        struct sockaddr_in server;

        fd = socket(AF_INET, SOCK_STREAM, 0);
        if (fd < 0) err_message("socket err\n");

        server.sin_family = AF_INET;
        server.sin_port = htons(u16port);
        inet_pton(AF_INET, addr, &server.sin_addr);

        if (bind(fd, (struct sockaddr *)&server, sizeof(server)) < 0) err_message("bind err\n")

        if (listen(fd, 10) < 0) err_message("listen err\n");

        return fd;
}

static void read_cb(EV_P_ ev_io *watcher, int revents)
{
        ssize_t ret;
        char buf[MAX_MESSAGE_LEN] = {0};

        ret = recv(watcher->fd, buf, sizeof(buf) - 1, MSG_DONTWAIT);
        if (ret > 0) {
                write(watcher->fd, buf, ret);

        } else if ((ret < 0) && (errno == EAGAIN || errno == EWOULDBLOCK)) {
                return;

        } else {
                fprintf(stdout, "client closed (fd=%d)\n", watcher->fd);
                --client_number;
                ev_io_stop(EV_A_ watcher);
                close(watcher->fd);
                free(watcher);
        }
}

static void accept_cb(EV_P_ ev_io *watcher, int revents)
{
        int connfd;
        ev_io *client;

        connfd = accept(watcher->fd, NULL, NULL);
        if (connfd > 0) {
                if (++client_number > MAX_CLIENTS) {
                        close(watcher->fd);

                } else {
                        client = calloc(1, sizeof(*client));
                        ev_io_init(client, read_cb, connfd, EV_READ);
                        ev_io_start(EV_A_ client);
                }

        } else if ((connfd < 0) && (errno == EAGAIN || errno == EWOULDBLOCK)) {
                return;

        } else {
                close(watcher->fd);
                ev_break(EV_A_ EVBREAK_ALL);
                /* this will lead main to exit, no need to free watchers of clients */
        }
}

static void start_server(char const *addr, uint16_t u16port)
{
        int fd;
#ifdef EV_MULTIPLICITY
        struct ev_loop *loop;
#else
        int loop;
#endif
        ev_io *watcher;

        fd = create_serverfd(addr, u16port);
        loop = ev_default_loop(EVFLAG_NOENV);
        watcher = calloc(1, sizeof(*watcher));
        assert(("can not alloc memory\n", loop && watcher));

        /* set nonblock flag */
        fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) | O_NONBLOCK);

        ev_io_init(watcher, accept_cb, fd, EV_READ);
```

```
        ev_io_start(EV_A_ watcher);
        ev_run(EV_A_ 0);

        ev_loop_destroy(EV_A);
        free(watcher);
}

static void signal_handler(int signo)
{
        switch (signo) {
                case SIGPIPE:
                        break;
                default:
                        // unreachable
                        break;
        }
}

int main(void)
{
        signal(SIGPIPE, signal_handler);
        start_server("127.0.0.1", 10009);

        return 0;
}
```

客户端:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <assert.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#include <pthread.h>

#define err_message(msg) \
        do {perror(msg); exit(EXIT_FAILURE);} while(0)

static int create_clientfd(char const *addr, uint16_t u16port)
{
        int fd;
        struct sockaddr_in server;

        fd = socket(AF_INET, SOCK_STREAM, 0);
        if (fd < 0) err_message("socket err\n");

        server.sin_family = AF_INET;
        server.sin_port = htons(u16port);
        inet_pton(AF_INET, addr, &server.sin_addr);

        if (connect(fd, (struct sockaddr *)&server, sizeof(server)) < 0) perror("connect err\n"

        return fd;
}

static void *routine(void *args)
{
        int fd;
        char buf[128];

        fd = create_clientfd("127.0.0.1", 10009);

        for (; ;) {
                write(fd, "Hello", strlen("hello"));

                memset(buf, '\0', sizeof(buf));
                read(fd, buf, sizeof(buf) - 1);
                fprintf(stdout, "pthreadid:%ld %s\n", pthread_self(), buf);
                usleep(100 * 1000);
        }
}

int main(void)
{
        pthread_t pids[4];
```

```
        for (int i = 0; i < sizeof(pids)/sizeof(pthread_t); ++i) {
                pthread_create(pids + i, NULL, routine, 0);
        }

        for (int i = 0; i < sizeof(pids)/sizeof(pthread_t); ++i) {
                pthread_join(pids[i], 0);
        }

        return 0;
}
```

Makefile

```
all:server client

server_src += \
        server.c

server_obj := $(patsubst %.c, %.o, $(server_src))

client_src += \
        client.c

client_obj:= $(patsubst %.c, %.o, $(client_src))

CC := clang

CFLAGS += -Wall -fPIC

server:$(server_obj)
        $(CC) -o $@ $^ -lev
%.o:%.c
        $(CC) -o $@ -c $< $(CFLAGS)

client:$(client_obj)
        $(CC) -o $@ $^ -lpthread
%.o:%.c
        $(CC) -o $@ -c $< $(CFLAGS)

.PHONY:clean all

clean:
        @rm -rf server client *.o
```

放弃很容易，但是坚持真的很酷，静享此刻，强风吹拂

分类: libev

» 更多新闻...