事件驱动库 libev 使用详解

libev 是一个通过 C 语言编写的，高性能的事件循环库，支持多种事件类型，与此类似的事件循环库还有 libevent、libubox 等，在此详细介绍下 libev 相关的内容。

简介

这是一个简单而且高性能的事件库，支持常规的 IO、定时器等事件，而且没有任何依赖，同时支持多线程模式。

关于 libev 详见官网 http://software.schmorp.de (国内会被墙)，其帮助文档可以参考 官方文档，安装完之后，可通过 `man 3 ev` 查看帮助信息，文档也在源码中保存了一份，可以通过 `man -l ev.3` 命令查看。

安装

安装可以源码安装，或者在 CentOS 中，最简单可通过如下方式安装。

```
----- 安装库
# yum install libev libev-devel
```

示例程序

如下是一个简单的示例程序。

```
#include <ev.h>
#include <stdio.h>

// every watcher type has its own typedef'd struct with the name ev_TYPE
ev_io stdin_watcher;
ev_timer timeout_watcher;

// all watcher callbacks have a similar signature
// this callback is called when data is readable on stdin
static void stdin_cb (EV_P_ ev_io *w, int revents)
{
        puts("stdin ready");
        // for one-shot events, one must manually stop the watcher
        // with its corresponding stop function.
        ev_io_stop(EV_A_ w);

        // this causes all nested ev_run's to stop iterating
        ev_break(EV_A_ EVBREAK_ALL);
}

// another callback, this time for a time-out
static void timeout_cb (EV_P_ ev_timer *w, int revents)
{
        puts("timeout");
        // this causes the innermost ev_run to stop iterating
        ev_break(EV_A_ EVBREAK_ONE);
}

int main (void)
{
        // use the default event loop unless you have special needs
        // struct ev_loop *loop = EV_DEFAULT; /* OR ev_default_loop(0) */
        EV_P EV_DEFAULT;

        // initialise an io watcher, then start it
        // this one will watch for stdin to become readable
        ev_io_init(&stdin_watcher, stdin_cb, /*STDIN_FILENO*/ 0, EV_READ);
        ev_io_start(EV_A_ &stdin_watcher);

        // initialise a timer watcher, then start it
        // simple non-repeating 5.5 second timeout
```

```
        ev_timer_init(&timeout_watcher, timeout_cb, 5.5, 0.);
        ev_timer_start(EV_A_ &timeout_watcher);

        ev_run(EV_A_ 0); /* now wait for events to arrive */

        ev_loop_destroy(EV_A);

        return 0;
}
```

可以通过如下命令编译。

```
----- 编译示例程序
$ gcc -lev example.c -o example
```

执行过程

libev 是一个事件循环，首先需要注册感兴趣的事件，libev 会监控这些事件，当事件发生时调用相应的处理函数，也就是回调函数。其处理过程为：

1. 初始化一个事件循环。可以通过 `ev_default_loop(0)` 或者 `EV_DEFAULT` 初始化，两者等价。
2. 定义事件类型。在 `ev.h` 中定义了各种类型的，如 `ev_io`、`ev_timer`、`ev_signal` 等。
3. 注册感兴趣事件。这里被称为 watchers ，这个是 C 结构体。
4. 启动监控。启动上步注册的事件，如 `ev_io_start()`、`ev_timer_start()` 等。
5. 启动 libev 循环。重复 1, 2 步，然后启动 libev 事件循环，直接执行 `ev_run()` 即可。

循环体

可以通过 `ev_default_loop()` 初始化一个默认循环，如果支持多实例可以使用 `ev_loop_new()` 创建一个新的，其中包括了部分入参用来标示如何进行处理。

可用标示有。

```
enum {
        /* the default */
        EVFLAG_AUTO      = 0x00000000U, /* not quite a mask */
        /* flag bits */
        EVFLAG_NOENV     = 0x01000000U, /* do NOT consult environment */
        EVFLAG_FORKCHECK = 0x02000000U, /* check for a fork in each iteration */
        /* debugging/feature disable */
        EVFLAG_NOINOTIFY = 0x00100000U, /* do not attempt to use inotify */
#if EV_COMPAT3
        EVFLAG_NOSIGFD   = 0, /* compatibility to pre-3.9 */
#endif
        EVFLAG_SIGNALFD  = 0x00200000U, /* attempt to use signalfd */
        EVFLAG_NOSIGMASK = 0x00400000U  /* avoid modifying the signal mask */
};
```

事件设置方式

这里以 `struct ev_io` 为例，有如下的两种设置方式。

将回调函数和关心的事件同时注册。

```
struct ev_io wstdin;
ev_io_init(&wstdin, stdin_hook, /*STDIN_FILENO*/ 0, EV_READ);
ev_io_start(EV_A &wstdin);
```

一般来说，在设置了回调函数之后，很少会进行修改，在首次调用的时候需要修改关注的事件，那么此时就可以将设置回调和设置事件分开。

```
struct ev_io wstdin;
ev_init(&wstdin, stdin_hook);
... ... /* some time later. */
ev_io_set(&wstdin, /*STDIN_FILENO*/ 0, EV_READ);
ev_io_start(EV_A &wstdin);
```

定制适配

在使用 libev 时，可以作如下的适配。

初始化

在 libev 中，通过 `struct ev_loop` 结构定义了一个具体的循环实例，包含了事件循环所需要的所有数据，而同时 libev 提供了很多宏定义适配多实例模式，也就是需要使用多个 `loop` ，可以通过 `#define EV_MULTIPLICITY 1` 宏进行定义。

使用多实例模式时，一般函数的第一个入参就是 `loop` ，为此，libev 提供了一系列宏适配单实例和多实例模式，允许不修改代码直接编译即可。例如，`EV_P_` 作为函数定义、函数声明的第一个参数，当是多实例时，实际为 `struct ev_loop *loop,` ，而在单实例模式中就是空。

不过在定义初始化的时候有点问题，没有找到合适的宏定义，建议添加如下内容。

```
#if EV_MULTIPLICITY
# define EV_DEFAULT_DEC EV_P = EV_DEFAULT
#else
# define EV_DEFAULT_DEC ev_default_loop(0)
#endif
```

这样就可以在开始初始化的时候直接使用 `EV_DEFAULT_DEC` 即可。

定制化

默认 libev 会使用 `config.h` 作为配置文件，不过这个文件可能会根项目的配置文件冲突，可以通过宏 `EV_CONFIG_H` 定义，如果使用的时 CMake 作管理，那么通过如下方式定义。

```
ADD_DEFINITIONS(-DEV_CONFIG_H="evconfig.h")
```

在该文件中就可以进行部分的定制，例如可以使用如下内容。

```
#ifndef EV_EVCONFIG_H_
#define EV_EVCONFIG_H_

#define EV_PERIODIC_ENABLE 1
#define EV_STAT_ENABLE     1
#define EV_PREPARE_ENABLE  1
#define EV_CHECK_ENABLE    1
#define EV_IDLE_ENABLE     1
#define EV_FORK_ENABLE     1
#define EV_CLEANUP_ENABLE  1
#define EV_SIGNAL_ENABLE   1
#define EV_CHILD_ENABLE    1
#define EV_ASYNC_ENABLE    1
#define EV_EMBED_ENABLE    1
#define EV_WALK_ENABLE     0 /* not yet */

#define EV_USE_EPOLL       1
#define HAVE_EPOLL_CTL     1
#define HAVE_SYS_EPOLL_H   1

#define EV_AVOID_STDIO     1

#endif
```

使用示例

除了基础的 IO、定时器、信号的处理之外，同时还提供了一些循环中经常使用的 hook 处理，以及一些常用的场景。

```
### 基础事件能力
struct ev_io        IO事件，包括了Socket、Pipe
struct ev_timer     定时器，采用的是相对时间，基于Bin-Heap
struct ev_periodic  定时器，采用的是UTC时间，基于Bin-Heap
```

```
struct ev_signal    = 信号处理
struct ev_child     = SIGCHLD信号的处理
struct ev_stat      文件的监控，Linux中用的是inotify机制

### 扩展事件
struct ev_fork      = 是否是在子进程中运行 使用forks数组
struct ev_embed
struct ev_async     = 异步事件，内部采用Pipe实现 使用asyncs数组

### 循环流程Hook
struct ev_idle      空闲 使用idles数组
struct ev_prepare   每次循环在阻塞之前调用 使用prepares数组
struct ev_check     每次循环在事件处理之后调用 使用checks数组
struct ev_cleanup   当循环退出时会调用 使用cleanups数组，会在ev_loop_destroy()中触发
```

如下简单介绍各种事件的使用方法。

Timer Watcher

可以设置定时器的启动时间，以及循环的时间间隔。

```
#include <time.h>
#include <stdio.h>
#include <stdint.h>

#include "libev/ev.h"

ev_timer timeout_watcher;
ev_timer repeate_watcher;
ev_timer oneshot_watcher;

// another callback, this time for a time-out
static void timeout_cb (EV_P_ ev_timer *w, int revents)
{
        (void) w;
        (void) revents;
        printf("timeout at %ju\n", (uintmax_t)time(NULL));

        /* this causes the innermost ev_run to stop iterating */
        ev_break(EV_A_ EVBREAK_ONE);
}
static void repeate_cb (EV_P_ ev_timer *w, int revents)
{
        (void) w;
        (void) revents;
        printf("repeate at %ju\n", (uintmax_t)time(NULL));
}
static void oneshot_cb (EV_P_ ev_timer *w, int revents)
{
        (void) w;
        (void) revents;
        printf("oneshot at %ju\n", (uintmax_t)time(NULL));
        ev_timer_stop(EV_A_ w);
}

int main (void)
{
        time_t result;
        EV_DEFAULT_DEC; /* OR ev_default_loop(0) */

        result = time(NULL);
        printf("  start at %ju\n", (uintmax_t)result);

        /* run only once in 2s later */
        ev_timer_init(&oneshot_watcher, oneshot_cb, 2.0, 0.);
        ev_timer_start(EV_A_ &oneshot_watcher);

        /* run in 5 seconds later, and repeat every second */
        ev_timer_init(&repeate_watcher, repeate_cb, 5., 1.);
        ev_timer_start(EV_A_ &repeate_watcher);

        /* timeout in 10s later, and also quit. */
        ev_timer_init(&timeout_watcher, timeout_cb, 10., 0.);
        ev_timer_start(EV_A_ &timeout_watcher);

        /* now wait for events to arrive. */
        ev_run(EV_A_ 0);
```

```
        ev_loop_destroy(EV_A);

        return 0;
}
```

Periodic Watcher

Periodic 可以理解为类似于 crontab ，不像 timer 基于的是相对时间，改调度基于的是日历时间或者说是墙上时间。

这也就意味着，时间会受手动调整的影响，有可能比真实的时间快或者慢。

例如，启动一个 periodic 时钟在 10 秒后触发，但是此时又将时间调整到了一个月之后，那么这一事件将在 1month+10seconds 之后触发，而如果使用的是 timer ，那么无论时间如何调整都会在 10s 后触发。

```
#include <time.h>
#include <stdio.h>
#include <stdint.h>

#include "ev.h"

#define log_info(fmt, args...) printf("%ju " fmt, time(NULL), ##args)

static void minute_tick_hook(EV_P_ ev_periodic *w, int revents)
{
        (void) loop;
        (void) w;
        (void) revents;
        log_info("invoking\n");
}

int main (void)
{
        struct ev_loop *loop = ev_default_loop(0);

        ev_periodic minute_tick;
        ev_periodic_init(&minute_tick, minute_tick_hook, 0., 60., 0);
        ev_periodic_start(EV_A_ &minute_tick);

        log_info("start\n");

        /* now wait for events to arrive. */
        ev_run(EV_A_ 0);

        ev_loop_destroy(EV_A);

        return 0;
}
```

Signal Watcher

在收到 `SIGINT` 时做些清理，直接退出。

```
#include <ev.h>
#include <stdio.h>
#include <signal.h>

static void sigint_cb(EV_P_ ev_signal *w, int revents)
{
        (void) revents;

        printf("catch SIGINT, signal number %d.\n", w->signum);
        ev_break(EV_A_ EVBREAK_ALL);
}

int main (void)
{
        ev_signal wsig;

        // use the default event loop unless you have special needs
        EV_DEFAULT_DEC; /* OR ev_default_loop(0) */

        ev_signal_init(&wsig, sigint_cb, SIGINT);
        ev_signal_start(EV_A_ &wsig);
```

```
        ev_run(EV_A_ 0); // now wait for events to arrive

        ev_loop_destroy(EV_A);

        return 0;
}
```

Child Watcher

fork 一个新进程，给它安装一个 child 处理器等待进程结束，实际上会等待接受 `SIGCHLD` 信号，然后调用相应的事件。

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

#include "libev/ev.h"

static void child_cb (EV_P_ ev_child *w, int revents)
{
        (void) revents;
        ev_child_stop(EV_A_ w);
        printf ("process %d exited with status %x\n", w->rpid, w->rstatus);
}

int main (void)
{
        /* use the default event loop unless you have special needs */
        EV_DEFAULT_DEC; /* OR ev_default_loop(0) */
        ev_child cw;

        pid_t pid = fork();
        if (pid < 0) {  /* error */
                perror("fork()");
                exit(EXIT_FAILURE);
        } else if (pid == 0) {  /* child, the forked child executes here */
                sleep(1);
                exit(EXIT_SUCCESS);
        }

        printf("parent %d child %d forked.\n", getpid(), pid);

        /* parent */
        ev_child_init(&cw, child_cb, pid, 0);
        ev_child_start(EV_A_ &cw);

        /* now wait for events to arrive */
        ev_run(EV_A_ 0);

        ev_loop_destroy(EV_A);

        return 0;
}
```

在 libev 中，实际上也是通过注册一个 `SIGCHILD` 信号进行处理的，其回调函数是 `childcb` 。

Fork Watcher

在 libev 中提供了一个 fork 事件的监控，libev 会在循环中自动检测是否调用了 `fork()` 函数，如果是那么会重新设置事件驱动回调函数。

注意，默认不会自动检测，需要设置相关的参数，例如 `ev_default_loop(EVFLAG_FORKCHECK)`，这样才会在每次循环的时候检测。

除了自动判断，也可以在 `fork()` 子进程之后调用 `ev_loop_fork()` 函数。

```
#include <ev.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

static void fork_callback(EV_P_ ev_fork *w, int revents)
{
        (void) w;
        (void) revents;
```

```
        printf("[%d] fork callback\n", getpid());
}

static void timeout_callback(EV_P_ ev_timer *w,int revents)
{
        (void) w;
        (void) revents;

        printf("[%d] time out\n", getpid());
}

int main(void)
{
        struct ev_loop *loop;
        ev_fork wfork;
        ev_timer wtimer;

        loop = ev_default_loop(EVFLAG_FORKCHECK);

        ev_fork_init(&wfork, fork_callback);
        ev_fork_start(EV_A_ &wfork);

        ev_timer_init(&wtimer, timeout_callback, 1., 1.);
        ev_timer_start(EV_A_ &wtimer);

        pid_t pid;

        pid = fork();
        if (pid < 0) {
                return -1;
        } else if (pid == 0) {
                printf("[%d] Child\n", getpid());
                //ev_loop_fork(EV_A);
                ev_run(EV_A_ 0);
                ev_loop_destroy(EV_A);
                return 0;
        }

        printf("[%d] Parent\n", getpid());

        ev_run(EV_A_ 0);
        ev_loop_destroy(EV_A);

        return 0;
}
```

在如上的示例中，使用的是多实例模式，会在子进程中重新执行，所以最好的方式是，如果不需要最好直接关闭。

另外，在创建 epoll 对象时，入参使用了 `EPOLL_CLOEXEC` 参数，也就意味着在 fork 进程时会自动关闭文件描述符。

Async Watcher

通常用于多个线程之间的事件同步，该事件允许在不同的线程中发送事件消息，内部使用 PIPE 进行通讯。

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/syscall.h>

#include <ev.h>

#define log_info(fmt, args...) printf("%ju lwpid(%lu) " fmt, time(NULL), syscall(SYS_gettid), ##args)

struct ev_loop *work_loop = NULL;
static struct ev_async wasync;

static void async_cb(EV_P_ struct ev_async *w, int revents)
{
        (void) w;
        log_info("async hook call, event %d loop %p.\n", revents, loop);
}

void *ev_create(void *p)
{
        (void) p;
```

```
        log_info("worker thread start!\n");

        sleep(3);
        ev_async_init(&wasync, async_cb);
        ev_async_start(work_loop, &wasync);
        ev_run(work_loop, 0);

        return NULL;
}

int main(void)
{
        int num = 0;
        pthread_t tid;

        work_loop = ev_loop_new(EVFLAG_AUTO);

        pthread_create(&tid, NULL, ev_create, NULL);
        log_info("main thread start!\n");

        while(1) {
                log_info("send async #%d times.\n", num);
                ev_async_send(work_loop, &wasync);
                sleep(1);
                num++;
        }

        return 0;
}
```

如上，实际上启动顺序是不影响的，每次起一个线程都会建立一个 PIPE 管道。

多实例

也就是在多线程中，初始化不同的实例。

```
#include <ev.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/syscall.h>

#define log_info(fmt, args...) printf("%ju lwpid(%lu) " fmt, time(NULL), syscall(SYS_gettid), ##args)

static void repeate_hook(EV_P_ ev_timer *w, int revents)
{
        (void) w;
        (void) revents;
        (void) loop;

        log_info("repeate\n");
}

void *child1(void *arg)
{
        (void) arg;
        EV_P = ev_loop_new(0);

        ev_timer wtimer;

        log_info("child thread started.\n");

        ev_timer_init(&wtimer, repeate_hook, 0., 1.);
        ev_timer_start(EV_A_ &wtimer);

        ev_run(EV_A_ 0);

        return NULL;
}

int main (void)
{
        EV_DEFAULT_DEC; /* default */
        ev_timer wtimer;
        pthread_t tid1;

        pthread_create(&tid1, NULL, child1, NULL);

        ev_timer_init(&wtimer, repeate_hook, 0., 1.);
```

```
        ev_timer_start(EV_A_ &wtimer);

        /* now wait for events to arrive. */
        ev_run(EV_A_ 0);

        pthread_join(tid1, NULL);

        return 0;
}
```

注意，全局只能有一个默认的 `struct ev_loop` ，在线程中，需要通过 `ev_loop_new()` 再新建一个。

参考

相关的文档可以参考 metacpan ev.pod ，也可以在该网站获取最新的版本，例如 libev 4.27 。

如果喜欢这里的文章，而且又不差钱的话，欢迎打赏个早餐 ^_^