libev库的用法

Yeolar 2012-12-16 21:12

libev是一个高性能的事件循环库，比libevent库的性能要好。Nodejs就是采用它作为底层库。libev的官方文档在 这里 (http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod) ，文档比较长。本文结合里面的例子对它的用法做些简单的总结。

目录

例子

首先从官方的例子开始：

```
// a single header file is required
#include <ev.h>

#include <stdio.h> // for puts

// 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;

  // 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 (loop, &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 (loop, &timeout_watcher);

  // now wait for events to arrive
  ev_run (loop, 0);

  // break was called, so exit
  return 0;
}
```

这个例子首先创建了一个事件循环，然后注册了两个事件：读取标准输入事件和超时事件。在终端输入或超时后，结束事件循环。

事件循环

使用libev的核心是事件循环，可以用 ev_default_loop 或 ev_loop_new 函数创建循环，或者直接使用 EV_DEFAULT 宏，区别是 ev_default_loop 创建的事件循环不是线程安全的，而 ev_loop_new 创建的事件循环不能捕捉信号和子进程的观察器。大多数情况下，可以像下面这样使用：

```
if (!ev_default_loop (0))
  fatal ("could not initialise libev, bad $LIBEV_FLAGS in environment?");
```

或者明确选择一个后端：

```
struct ev_loop *epoller = ev_loop_new (EVBACKEND_EPOLL | EVFLAG_NOENV);
if (!epoller)
  fatal ("no epoll found here, maybe it hides under your chair");
```

如果需要动态分配循环的话，建议使用 ev_loop_new 和 ev_loop_destroy 。

在创建子进程后，且想要使用事件循环时，需要先在子进程中调用 ev_default_fork 或 ev_loop_fork 来重新初始化后端的内核状态，它们分别对应 ev_default_loop 和 ev_loop_new 来使用。

ev_run 启动事件循环。它的第二个参数为0时，将持续运行并处理循环直到没有活动的事件观察器或者调用了 ev_break 。另外两个取值是 EVRUN_NOWAIT 和 EVRUN_ONCE 。

ev_break 跳出事件循环（在全部已发生的事件处理完之后）。第二个参数为 EVBREAK_ONE 或 EVBREAK_ALL 来指定跳出最内层的 ev_run 或者全部嵌套的 ev_run 。

ev_suspend 和 ev_resume 用来暂停和重启事件循环，比如在程序挂起的时候。

观察器

接下来创建观察器，它主要包括类型、触发条件和回调函数。将它注册到事件循环上，在满足注册的条件时，会触发观察器，调用它的回调函数。

上面的例子中已经包含了IO观察器和计时观察器，此外还有周期观察器、信号观察器、文件状态观察器等等。

初始化和设置观察器使用 ev_init 和 ev_TYPE_set ，也可以直接使用 ev_TYPE_init 。

在特定事件循环上启动观察器使用 ev_TYPE_start 。 ev_TYPE_stop 停止观察器，并且会释放内存。

libev中将观察器分为4种状态：初始化、启动/活动、等待、停止。

libev中的观察器还支持优先级。

不同类型的观察器就不详细解释了，只把官方的一些例子贴在这里吧。

ev_io

获取标准输入：

```
static void
stdin_readable_cb (struct ev_loop *loop, ev_io *w, int revents)
{
  ev_io_stop (loop, w);
  .. read from stdin here (or from w->fd) and handle any I/O errors
}

ev_io stdin_readable;
ev_io_init (&stdin_readable, stdin_readable_cb, STDIN_FILENO, EV_READ);
ev_io_start (loop, &stdin_readable);
```

ev_timer

创建一个60s之后启动的计时器：

```
static void
one_minute_cb (struct ev_loop *loop, ev_timer *w, int revents)
{
  .. one minute over, w is actually stopped right here
}

ev_timer mytimer;
ev_timer_init (&mytimer, one_minute_cb, 60., 0.);
ev_timer_start (loop, &mytimer);
```

创建一个10s超时的超时器：

```
static void
timeout_cb (struct ev_loop *loop, ev_timer *w, int revents)
{
  .. ten seconds without any activity
}

ev_timer mytimer;
ev_timer_init (&mytimer, timeout_cb, 0., 10.); /* note, only repeat used */
ev_timer_again (&mytimer); /* start timer */
ev_run (loop, 0);

// and in some piece of code that gets executed on any "activity":
// reset the timeout to start ticking again at 10 seconds
ev_timer_again (&mytimer);
```

ev_periodic

创建一个小时为单位的周期定时器：

```
static void
clock_cb (struct ev_loop *loop, ev_periodic *w, int revents)
{
  ... its now a full hour (UTC, or TAI or whatever your clock follows)
}

ev_periodic hourly_tick;
ev_periodic_init (&hourly_tick, clock_cb, 0., 3600., 0);
ev_periodic_start (loop, &hourly_tick);
```

或者自定义周期计算方式：

```
#include <math.h>

static ev_tstamp
my_scheduler_cb (ev_periodic *w, ev_tstamp now)
{
  return now + (3600. - fmod (now, 3600.));
}

ev_periodic_init (&hourly_tick, clock_cb, 0., 0., my_scheduler_cb);
```

如果想从当前时间开始：

```
ev_periodic hourly_tick;
ev_periodic_init (&hourly_tick, clock_cb,
                  fmod (ev_now (loop), 3600.), 3600., 0);
ev_periodic_start (loop, &hourly_tick);
```

ev_signal

在收到 `SIGINT` 时做些清理：

```
static void
sigint_cb (struct ev_loop *loop, ev_signal *w, int revents)
{
  ev_break (loop, EVBREAK_ALL);
}

ev_signal signal_watcher;
ev_signal_init (&signal_watcher, sigint_cb, SIGINT);
ev_signal_start (loop, &signal_watcher);
```

ev_child

`fork` 一个新进程，给它安装一个child处理器等待进程结束：

```
ev_child cw;

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

pid_t pid = fork ();

if (pid < 0)
  // error
else if (pid == 0)
  {
    // the forked child executes here
    exit (1);
  }
else
  {
    ev_child_init (&cw, child_cb, pid, 0);
    ev_child_start (EV_DEFAULT_ &cw);
  }
```

ev_stat

监控/etc/passwd是否有变化：

```
static void
passwd_cb (struct ev_loop *loop, ev_stat *w, int revents)
{
  /* /etc/passwd changed in some way */
  if (w->attr.st_nlink)
    {
      printf ("passwd current size  %ld\n", (long)w->attr.st_size);
      printf ("passwd current atime %ld\n", (long)w->attr.st_mtime);
      printf ("passwd current mtime %ld\n", (long)w->attr.st_mtime);
    }
  else
    /* you shalt not abuse printf for puts */
    puts ("wow, /etc/passwd is not there, expect problems. "
          "if this is windows, they already arrived\n");
}

...
ev_stat passwd;

ev_stat_init (&passwd, passwd_cb, "/etc/passwd", 0.);
ev_stat_start (loop, &passwd);
```

其他功能

libev还支持很多其他的有用的功能，比如自定义观察器，在线程中使用等等，这些请看官方文档。

Python绑定

libev提供了C和C++接口，很多其他语言也有对应的第三方接口。

libev的Python扩展是 pyev (http://code.google.com/p/pyev/) ，文档见 http://packages.python.org/pyev/ (http://packages.python.org/pyev/) 。

libev中的基础功能在pyev中基本上都有对应的绑定。

官方给了两个例子，如其中的展示基本用法的例子：

```python
import signal
import pyev

def sig_cb(watcher, revents):
    print("got SIGINT")
    loop = watcher.loop
    # optional - stop all watchers
    if loop.data:
        print("stopping watchers: {0}".format(loop.data))
        while loop.data:
            loop.data.pop().stop()
    # unloop all nested loop
    print("stopping the loop: {0}".format(loop))
    loop.stop(pyev.EVBREAK_ALL)

def timer_cb(watcher, revents):
    watcher.data += 1
    print("timer.data: {0}".format(watcher.data))
    print("timer.loop.iteration: {0}".format(watcher.loop.iteration))
    print("timer.loop.now(): {0}".format(watcher.loop.now()))

if __name__ == "__main__":
    loop = pyev.default_loop()
    # initialise and start a repeating timer
    timer = loop.timer(0, 2, timer_cb, 0)
    timer.start()
    # initialise and start a Signal watcher
    sig = loop.signal(signal.SIGINT, sig_cb)
    sig.start()
    loop.data = [timer, sig] # optional
    # now wait for events to arrive
    loop.start()
```

http://www.yeolar.com/note/2012/12/16/libev/