Golang 网络库 evio 一些问题 /bug 和思考

作者：Allenxuxu
原文链接：https://ld246.com/article/1565926947655
来源网站：链滴

Fast event-loop networking for Go

最近翻了 evio 的源码，发现一些问题，主要集中在 linux 平台 epoll 上和读写的处理。

- 用来唤醒 epoll 的 eventfd 写入数据没有读出
- listen 的 fd 注册到所有事件循环，epoll 的惊群问题
- loopWrite 在内核缓冲区满，无法一次写入时，出现写入数据丢失

eventfd 的使用问题

在 internal/internal_linux.go 中封装了 epoll 的使用 API 。

```
// Poll ...
type Poll struct {
    fd    int // epoll fd
    wfd   int // wake fd
    notes noteQueue
}
```

在 OpenPoll 时，会创建一个 eventfd 并将 fd 赋值给 Poll 的 wfd 成员， 并且注册到 epoll 监听可事件。

当需要唤醒当前 epoll 时，提供了 Trigger 方法

```
// Trigger ...
func (p *Poll) Trigger(note interface{}) error {
    p.notes.Add(note)
    _, err := syscall.Write(p.wfd, []byte{0, 0, 0, 0, 0, 0, 0, 1})
    return err
}
```

这是往刚刚提到的 eventfd 中写入八字节数据，此时 epol l会被唤醒 epoll_wait 函数返回。 但是，evo 并没有去把 8 个字节的数据读取出来，内核缓冲区会不断积压，并且 evio 使用的是 epoll 的LT模（默认模式），只要缓冲区中有数据，epoll 就会不断唤醒。这应该算是一个 bug 吧。

listen 的 fd 注册到所有事件循环，epoll 的惊群问题

evio 可以指定启动多个事件循环。evio 将 listen fd 注册到每一个事件循环中（epoll）监听可读事，所以当一个连接到来时，所有的事件循环都会唤醒。

```
// create loops locally and bind the listeners.
    for i := 0; i < numLoops; i++ {
        l := &loop{
            idx:     i,
            poll:    internal.OpenPoll(),
            packet:  make([]byte, 0xFFFF),
            fdconns: make(map[int]*conn),
        }
        for _, ln := range listeners {
            l.poll.AddRead(ln.fd)
        }
        s.loops = append(s.loops, l)
    }
    // start loops in background
    s.wg.Add(len(s.loops))
    for _, l := range s.loops {
        go loopRun(s, l)
    }
```

这并不是一个 bug ，因为最终只有一个线程可以accept调用返回成功，其他线程（协程）的accept数调用返回EAGAIN错误，作者也做出了处理。

```
    nfd, sa, err := syscall.Accept(fd)
    if err != nil {
        if err == syscall.EAGAIN {
            return nil
        }
        return err
    }
```

并且作者还利用每个事件循环都会被唤醒，来做客户端连接的负载均衡策略。

LeastConnections : 当存在其他事件循环的注册的客户端连接数比当前事件循环的连接数少的时候，接 return nil 。当有两个最下连接数相同的时候，也没关系，因为 accept 会保证只有一个可以成功。

RoundRobin： 原理也是一样，每个事件循环都会去判断 int(atomic.LoadUintptr(&s.accepted)) % en(s.loops)，轮到自己了，才继续执行，否则 return nil 。

```
if ln.fd == fd {
    if len(s.loops) > 1 {
        switch s.balance {
        case LeastConnections:
            n := atomic.LoadInt32(&l.count)
            for _, lp := range s.loops {
                if lp.idx != l.idx {
```

```
                if atomic.LoadInt32(&lp.count) < n {
                    return nil // do not accept
                }
            }
        }
    case RoundRobin:
        idx := int(atomic.LoadUintptr(&s.accepted)) % len(s.loops)
        if idx != l.idx {
            return nil // do not accept
        }
        atomic.AddUintptr(&s.accepted, 1)
    }
}
```

这样的做法没有问题，但是个人觉得 muduo 的做法似乎更好。

loopWrite 在内核缓冲区满，无法一次写入时，出现写入数丢失

```
func loopWrite(s *server, l *loop, c *conn) error {
    if s.events.PreWrite != nil {
        s.events.PreWrite()
    }
    n, err := syscall.Write(c.fd, c.out)
    if err != nil {
        if err == syscall.EAGAIN {
            return nil
        }
        return loopCloseConn(s, l, c, err)
    }
    if n == len(c.out) {
        c.out = nil
    } else {
        c.out = c.out[n:]
    }
    if len(c.out) == 0 && c.action == None {
        l.poll.ModRead(c.fd)
    }
    return nil
}
```

当内核缓冲区满时， syscall.Write(c.fd, c.out) 会无法全部写入，可以通过返回值得到已经写入的个。

```
if n == len(c.out) {
    c.out = nil
} else {
    c.out = c.out[n:]
}
```

作者也做出了处理，判断了 返回值 n 不是 c.out 长度的情况，c.out = c.out[n:] 。

但是作者并没有去注册 epoll 可写事件啊。

在当前文件搜索 ModReadWrite ，注册可读可写的事件，共有两处。一次是 loopWake 函数，一次在 loopRead 函数。会不会作者在 loopRead 方法中了做了处理，规避了没有注册可写事件这种情况？

我们看下 loopRead

```
func loopRead(s *server, l *loop, c *conn) error {
    var in []byte
    n, err := syscall.Read(c.fd, l.packet)
    if n == 0 || err != nil {
        if err == syscall.EAGAIN {
            return nil
        }
        return loopCloseConn(s, l, c, err)
    }
    in = l.packet[:n]
    if !c.reuse {
        in = append([]byte{}, in...)
    }
    if s.events.Data != nil {
        out, action := s.events.Data(c, in)
        c.action = action
        if len(out) > 0 {
            c.out = append([]byte{}, out...)
        }
    }
    if len(c.out) != 0 || c.action != None {
        l.poll.ModReadWrite(c.fd)
    }
    return nil
}
```

果然，作者做了处理！ 当 s.events.Data(c, in) 函数返回，如果 c.out 有数据，就注册可读可写事件。

所以，执行的流程是：

1. 客户端有数据到来，loopRead 函数执行

2. 调用客户注册的回调函数 events.Data 函数，客户将需要的写入给客户端的数据返回，evio 将需写给客户端数据存到 c.out , 然后监听可读可写事件

3. eopll 可写事件唤醒，执行 loopWrite 直接 write 数据。 如果写完就重新注册，只注册可读事件 如果没写完，就不重新注册，还是可读可写事件都监听

当缓冲区有空间了时，epoll 又会唤醒继续 loopWrite 。

似乎没问题，但是仔细想一想，会不会有这种情况呢：

内核的缓冲区满了，第一次没写完，等待缓冲区可写。此时客户端又来了数据，继续执行 loopRead 调用用户回调函数，又有要写入的数据。这是来看看处理逻辑

```
if s.events.Data != nil {
    out, action := s.events.Data(c, in)
    c.action = action
    if len(out) > 0 {
        c.out = append([]byte{}, out...)
```

```
    }
}
```

c.out = append([]byte{}, out...) 这里，之前没写完存在 c.out 里的数据直接被清空了啊。这样要写的数据就丢失了一部分啊。

思考

evio 速度非常快，但是翻了源码，发现 evio 并没有刻意去减少 epoll 的唤醒次数，相反 evio 利用 epoll 的多次唤醒去做操作。

比如，调用客户回调后，并没有直接处理 action 的状态，反而是先把 action 存起来，增加注册 fd 可写事件，让epoll 再唤醒，在 loopAction 中再来处理 action。先不说这样会不会有问题，这样让 epoll 频繁唤醒似乎不妥。

evio 的处理 read 和 write 的方式，也导致多次的内存拷贝，换种方式，性能还可以再次提升。evio linux 环境（epoll）下，单元测试因为 用来唤醒 epoll 的 eventfd 写入数据没有读出 这个bug，单测试并不能通过。在 ubuntu 环境下跑 evio 的压测，显示性能并没有 stdlib 好。

evio 非常轻量，这也说明它非常简单，使用起来还是非常不方便，并且对于 epoll 的处理还有很多可优化的地方。而且，作者似乎很忙。。。 PR也不理，Issues 也不理。所以决定自己撸一个了，更好，更快速：eviop 。eviop 是想优化 evio，但是由于 evio 的代码耦合性问题，举步维艰，所以干脆部重写，撸了 gev。

推荐库

- gev 一个轻量、快速的基于 Reactor 模式的非阻塞 TCP 网络库。

相关文章

- evio源码解析
- Golang 网络库 evio 一些问题/bug和思考
- gev: Go 实现基于 Reactor 模式的非阻塞 TCP 网络库