event包实现了同一个进程内部的事件发布和订阅模式。

event.go

目前这部分代码被标记为Deprecated，告知用户使用Feed这个对象。 不过在代码中任然有使用。 而且这部分的代码也不多。 就简单介绍一下。

数据结构
TypeMux是主要的使用。 subm记录了所有的订阅者。 可以看到每中类型都可以有很多的订阅者。

```
// TypeMuxEvent is a time-tagged notification pushed to subscribers.
type TypeMuxEvent struct {
    Time time.Time
    Data interface{}
}

// A TypeMux dispatches events to registered receivers. Receivers can be
// registered to handle events of certain type. Any operation
// called after mux is stopped will return ErrMuxClosed.
//
// The zero value is ready to use.
//
// Deprecated: use Feed
type TypeMux struct {
    mutex   sync.RWMutex
    subm    map[reflect.Type][]*TypeMuxSubscription
    stopped bool
}
```

创建一个订阅,可以同时订阅多种类型。

```
// Subscribe creates a subscription for events of the given types. The
// subscription's channel is closed when it is unsubscribed
// or the mux is closed.
func (mux *TypeMux) Subscribe(types ...interface{}) *TypeMuxSubscription {
    sub := newsub(mux)
    mux.mutex.Lock()
    defer mux.mutex.Unlock()
    if mux.stopped {
        // set the status to closed so that calling Unsubscribe after this
        // call will short circuit.
```

```
        sub.closed = true
        close(sub.postC)
    } else {
        if mux.subm == nil {
            mux.subm = make(map[reflect.Type][]*TypeMuxSubscription)
        }
        for _, t := range types {
            rtyp := reflect.TypeOf(t)
            oldsubs := mux.subm[rtyp]
            if find(oldsubs, sub) != -1 {
                panic(fmt.Sprintf("event: duplicate type %s in Subscribe", rtyp))
            }
            subs := make([]*TypeMuxSubscription, len(oldsubs)+1)
            copy(subs, oldsubs)
            subs[len(oldsubs)] = sub
            mux.subm[rtyp] = subs
        }
    }
    return sub
}

// TypeMuxSubscription is a subscription established through TypeMux.
type TypeMuxSubscription struct {
    mux     *TypeMux
    created time.Time
    closeMu sync.Mutex
    closing chan struct{}
    closed  bool

    // these two are the same channel. they are stored separately so
    // postC can be set to nil without affecting the return value of
    // Chan.
    postMu sync.RWMutex
    // readC 和 postC 其实是同一个channel。 不过一个是从channel读 一个只从channel写
    // 单方向的channel
    readC  <-chan *TypeMuxEvent
    postC  chan<- *TypeMuxEvent
}

func newsub(mux *TypeMux) *TypeMuxSubscription {
    c := make(chan *TypeMuxEvent)
    return &TypeMuxSubscription{
        mux:     mux,
        created: time.Now(),
        readC:   c,
        postC:   c,
        closing: make(chan struct{}),
    }
```

```
}
```

发布一个event到TypeMux上面，这个时候所有订阅了这个类型的都会收到这个消息。

```
// Post sends an event to all receivers registered for the given type.
// It returns ErrMuxClosed if the mux has been stopped.
func (mux *TypeMux) Post(ev interface{}) error {
    event := &TypeMuxEvent{
        Time: time.Now(),
        Data: ev,
    }
    rtyp := reflect.TypeOf(ev)
    mux.mutex.RLock()
    if mux.stopped {
        mux.mutex.RUnlock()
        return ErrMuxClosed
    }
    subs := mux.subm[rtyp]
    mux.mutex.RUnlock()
    for _, sub := range subs {
        // 阻塞式的投递。
        sub.deliver(event)
    }
    return nil
}


func (s *TypeMuxSubscription) deliver(event *TypeMuxEvent) {
    // Short circuit delivery if stale event
    if s.created.After(event.Time) {
        return
    }
    // Otherwise deliver the event
    s.postMu.RLock()
    defer s.postMu.RUnlock()

    select {    //阻塞方式的方法
    case s.postC <- event:
    case <-s.closing:
    }
}
```

feed.go

目前主要使用的对象。取代了前面说的event.go内部的TypeMux

feed数据结构

```
// Feed implements one-to-many subscriptions where the carrier of events is a chann
el.
// Values sent to a Feed are delivered to all subscribed channels simultaneously.
// Feed 实现了 1对多的订阅模式，使用了channel来传递事件。 发送给Feed的值会同时被传递给所
有订阅的channel。
// Feeds can only be used with a single type. The type is determined by the first S
end or
// Subscribe operation. Subsequent calls to these methods panic if the type does no
t
// match.
// Feed只能被单个类型使用。这个和之前的event不同，event可以使用多个类型。 类型被第一个Sen
d调用或者是Subscribe调用决定。 后续的调用如果类型和其不一致会panic
// The zero value is ready to use.
type Feed struct {
    once      sync.Once        // ensures that init only runs once
    sendLock  chan struct{}    // sendLock has a one-element buffer and is empty wh
en held.It protects sendCases.
    removeSub chan interface{} // interrupts Send
    sendCases caseList         // the active set of select cases used by Send

    // The inbox holds newly subscribed channels until they are added to sendCases.
    mu     sync.Mutex
    inbox  caseList
    etype  reflect.Type
    closed bool
}
```

初始化 初始化会被once来保护保证只会被执行一次。

```
func (f *Feed) init() {
    f.removeSub = make(chan interface{})
    f.sendLock = make(chan struct{}, 1)
    f.sendLock <- struct{}{}
    f.sendCases = caseList{{Chan: reflect.ValueOf(f.removeSub), Dir: reflect.Select
Recv}}
}
```

订阅，订阅投递了一个channel。 相对与event的不同。event的订阅是传入了需要订阅的类型，然后channel是在event的订阅代码里面构建然后返回的。 这种直接投递channel的模式可能会更加灵活。

然后根据传入的channel生成了SelectCase。放入inbox。

```
// Subscribe adds a channel to the feed. Future sends will be delivered on the chan
nel
// until the subscription is canceled. All channels added must have the same elemen
t type.
//
// The channel should have ample buffer space to avoid blocking other subscribers.
// Slow subscribers are not dropped.
func (f *Feed) Subscribe(channel interface{}) Subscription {
    f.once.Do(f.init)

    chanval := reflect.ValueOf(channel)
    chantyp := chanval.Type()
    if chantyp.Kind() != reflect.Chan || chantyp.ChanDir()&reflect.SendDir == 0 { /
/ 如果类型不是channel。 或者是channel的方向不能发送数据。那么错误退出。
        panic(errBadChannel)
    }
    sub := &feedSub{feed: f, channel: chanval, err: make(chan error, 1)}

    f.mu.Lock()
    defer f.mu.Unlock()
    if !f.typecheck(chantyp.Elem()) {
        panic(feedTypeError{op: "Subscribe", got: chantyp, want: reflect.ChanOf(ref
lect.SendDir, f.etype)})
    }
    // Add the select case to the inbox.
    // The next Send will add it to f.sendCases.
    cas := reflect.SelectCase{Dir: reflect.SelectSend, Chan: chanval}
    f.inbox = append(f.inbox, cas)
    return sub
}
```

Send方法,feed的Send方法不是遍历所有的channel然后阻塞方式的发送。这样可能导致慢的客户端影响快的客户端。 而是使用反射的方式使用SelectCase。 首先调用非阻塞方式的TrySend来尝试发送。这样如果没有慢的客户端。数据会直接全部发送完成。 如果TrySend部分客户端失败。 那么后续在循环Select的方式发送。 我猜测这也是feed会取代event的原因。

```
// Send delivers to all subscribed channels simultaneously.
// It returns the number of subscribers that the value was sent to.
func (f *Feed) Send(value interface{}) (nsent int) {
    f.once.Do(f.init)
    <-f.sendLock

    // Add new cases from the inbox after taking the send lock.
```

```
    f.mu.Lock()
    f.sendCases = append(f.sendCases, f.inbox...)
    f.inbox = nil
    f.mu.Unlock()

    // Set the sent value on all channels.
    rvalue := reflect.ValueOf(value)
    if !f.typecheck(rvalue.Type()) {
        f.sendLock <- struct{}{}
        panic(feedTypeError{op: "Send", got: rvalue.Type(), want: f.etype})
    }
    for i := firstSubSendCase; i < len(f.sendCases); i++ {
        f.sendCases[i].Send = rvalue
    }

    // Send until all channels except removeSub have been chosen.
    cases := f.sendCases
    for {
        // Fast path: try sending without blocking before adding to the select set.
        // This should usually succeed if subscribers are fast enough and have free
        // buffer space.
        for i := firstSubSendCase; i < len(cases); i++ {
            if cases[i].Chan.TrySend(rvalue) {
                nsent++
                cases = cases.deactivate(i)
                i--
            }
        }
        if len(cases) == firstSubSendCase {
            break
        }
        // Select on all the receivers, waiting for them to unblock.
        chosen, recv, _ := reflect.Select(cases)
        if chosen == 0 /* <-f.removeSub */ {
            index := f.sendCases.find(recv.Interface())
            f.sendCases = f.sendCases.delete(index)
            if index >= 0 && index < len(cases) {
                cases = f.sendCases[:len(cases)-1]
            }
        } else {
            cases = cases.deactivate(chosen)
            nsent++
        }
    }

    // Forget about the sent value and hand off the send lock.
    for i := firstSubSendCase; i < len(f.sendCases); i++ {
        f.sendCases[i].Send = reflect.Value{}
```

```
    }
    f.sendLock <- struct{}{}
    return nsent
}
```