
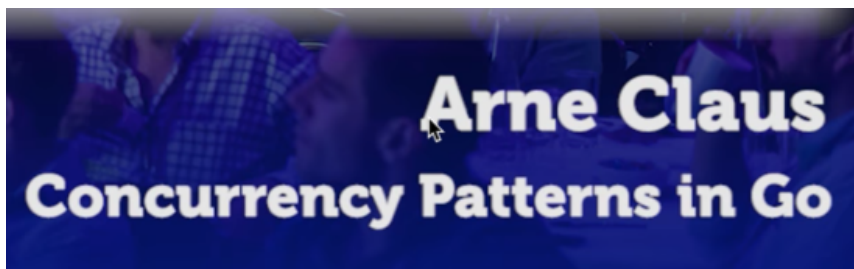


发件人: 张国庆 ultrazgq@icloud.com 
主题:
日期: 2018年6月15日 下午7:42
收件人:

张

Golang UK Conference 2017 | Arne Claus - Concurrency Patterns in Go

2018年6月15日 星期五
下午7:03

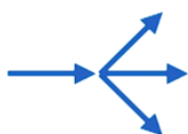


Making channels non-blocking

```
func TryReceiveWithTimeout(c <-chan int, duration time.Duration) (data int, more, ok bool) {  
    select {  
        case data, more = <-c:  
            return data, more, true  
  
        case <-time.After(duration): // time.After() returns a channel  
            return 0, true, false  
    }  
}
```

Shape your data flow

- Channels are streams of data
- Dealing with multiple streams is the true power of select



Fan-out



Funnel



Turnout

Concurrency in practice

- Avoid blocking, avoid race conditions
- Use channels to avoid shared state.
Use select to manage channels.
- Where channels don't work:
 - Try to use tools from the sync package first
 - In simple cases or when *really* needed: try lockless code

nc package

Golang UK Conference 2017 | Arne Claus - Concurrency Patterns in Go

按 esc 即可退出全屏模式

Quit channel

```
func Turnout(Quit <-chan int, InA, InB, OutA, OutB chan int) {  
    // variable declaration left out for readability  
    for {  
        select {  
            case data = <-InA:  
            case data = <-InB:  
  
            case <-Quit: // remember: close generates a message  
                        // Actually this is an anti-pattern ...  
                        // ... but you can argue that quit acts as a delegate  
                close(InA)  
                close(InB)  
  
                Fanout(InA, OutA, OutB) // Flush the remaining data  
                Fanout(InB, OutA, OutB)  
                return  
        }  
    }  
    // ...  
}
```

Arne Claus
Concurrency Patterns in Go

16:42 / 31:51

YouTube

Atomic

Spinning CAS

- You need a **state** variable and a „**free**“ constant
- Use CAS (CompareAndSwap) in a loop:
 - If state is **not free**: try again until it is
 - If state is **free**: set it to something else
- If you managed to change the state, you „own“ it

state, you

Ticket storage

- We need an **indexed data structure**, a **ticket** and a **done** variable
- A function draws a new ticket by adding 1 to the ticket
- Every ticket number is **unique** as we never decrement
- Treat the **ticket as an index** to store your data
- Increase done to extend the „ready to read“ range

ready to re

Ticket storage

```
type TicketStore struct {
    ticket *uint64
    done   *uint64
    slots  []string // for simplicity: imagine this to be infinite
}

func (ts *TicketStore) Put(s string) {
    t := atomic.AddUint64(ts.ticket, 1) - 1 // draw a ticket
    slots[t] = s                             // store your data
    for !atomic.CompareAndSwapUint64(ts.done, t, t+1) { // increase done
        runtime.Gosched()
    }
}

func (ts *TicketStore) GetDone() []string {
    return ts.slots[:atomic.LoadUint64(ts.done)+1] // read up to done
}
```

Guidelines for non-blocking code

- Don't switch between atomic and non-atomic functions
- Target and exploit situations which enforce uniqueness
- Avoid changing two things at a time
 - Sometimes you can exploit bit operations
 - Sometimes intelligent ordering can do the trick
 - Sometimes it's just not possible at all

ring can do
ible at all

Concurrency in practice

- Avoid blocking, avoid race conditions
- Use channels to avoid shared state.
Use select to manage channels.
- Where channels don't work:
 - Try to use tools from the sync package first
 - In simple cases or when *really* needed: try lockless code

c package
ly needed.