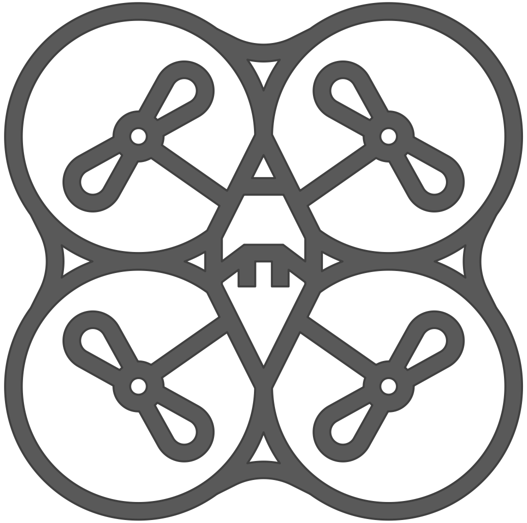


Go并发编程

晁岳攀
@colobu
https://colobu.com

目录

DIRECTORY

01

同步原语

基本同步原语、扩展
同步原语、原子操作

02

分布式同步原语

锁、栅栏、队列
Leader选举、STM

03

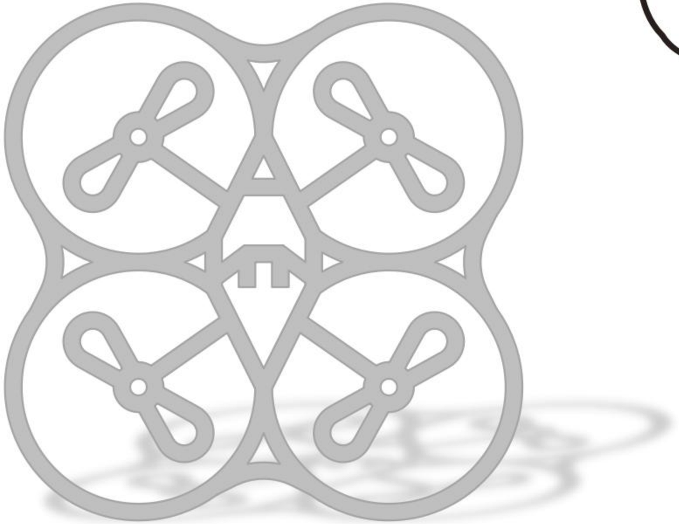
Channel

异常情况
三种应用场景
应用模式

04

内存模型

内存模型
happen before

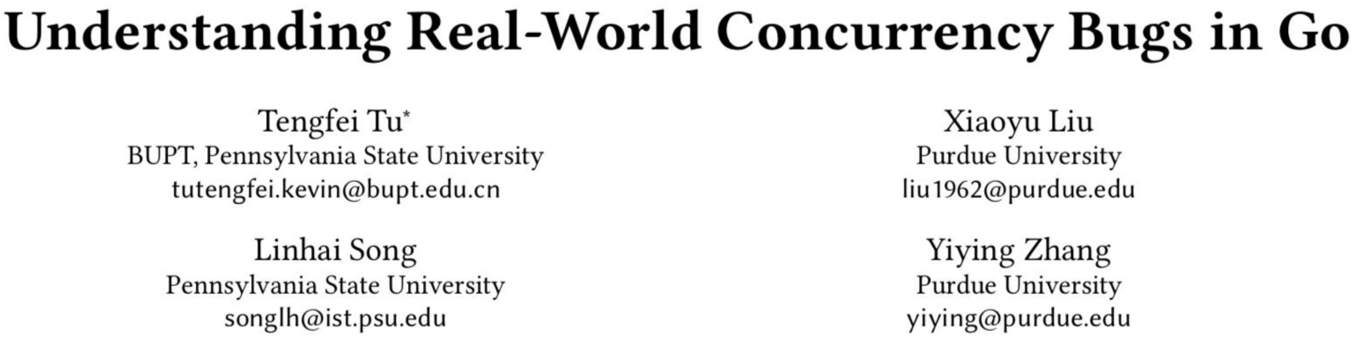


00

小故事

- Therac-25事件

Go并发BUG研究



Understanding Real-World Concurrency Bugs in Go

Tengfei Tu*
BUPT, Pennsylvania State University
tutengfei.kevin@bupt.edu.cn

Xiaoyu Liu
Purdue University
liu1962@purdue.edu

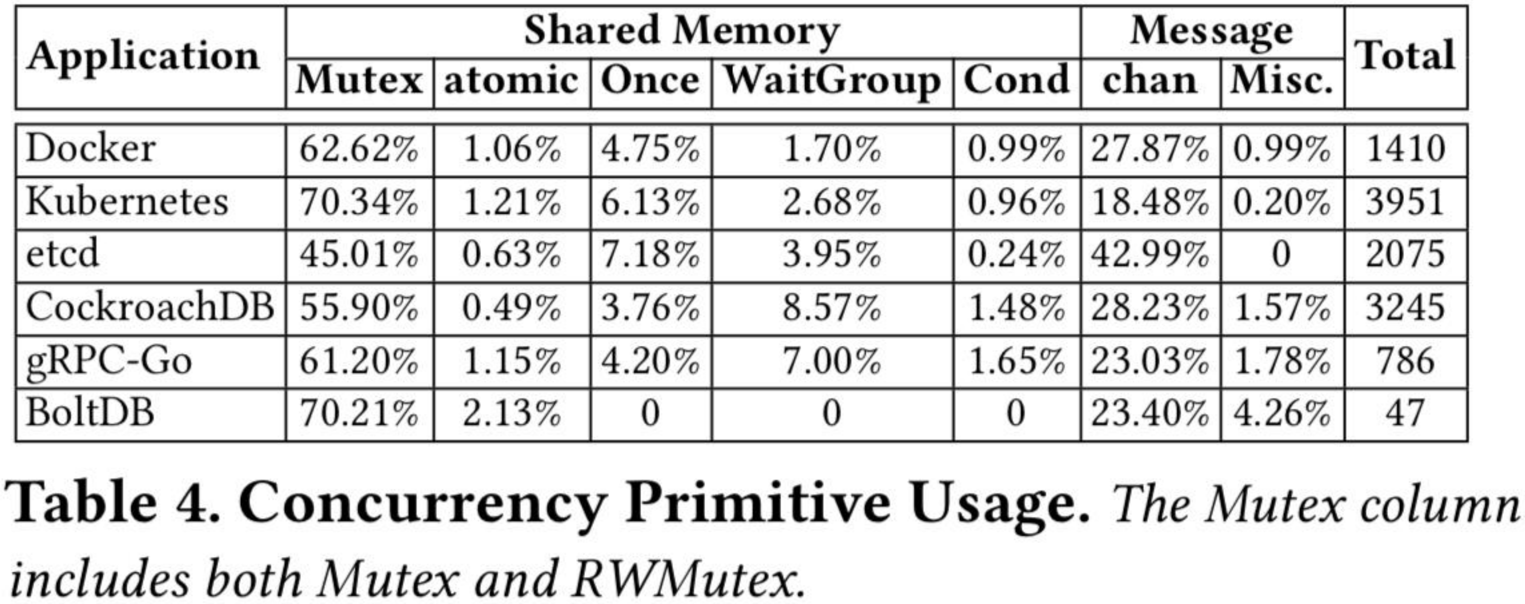
Linhai Song
Pennsylvania State University
songlh@ist.psu.edu

Yiying Zhang
Purdue University
yiying@purdue.edu

- 研究了 Docker、Kubernetes、etcd、gRPC、CockroachDB、BoltDB的bug
- 两种分类方法
  - share memory bugs 和 message passing bugs
  - blocking bugs和non-blocking bugs

Go并发BUG研究

基本同步原语的使用占比



Application	Shared Memory					Message		Total
	Mutex	atomic	Once	WaitGroup	Cond	chan	Misc.	
Docker	62.62%	1.06%	4.75%	1.70%	0.99%	27.87%	0.99%	1410
Kubernetes	70.34%	1.21%	6.13%	2.68%	0.96%	18.48%	0.20%	3951
etcd	45.01%	0.63%	7.18%	3.95%	0.24%	42.99%	0	2075
CockroachDB	55.90%	0.49%	3.76%	8.57%	1.48%	28.23%	1.57%	3245
gRPC-Go	61.20%	1.15%	4.20%	7.00%	1.65%	23.03%	1.78%	786
BoltDB	70.21%	2.13%	0	0	0	23.40%	4.26%	47

Table 4. Concurrency Primitive Usage. *The Mutex column includes both Mutex and RWMutex.*

Go并发BUG研究

两种类型的同步原语占比

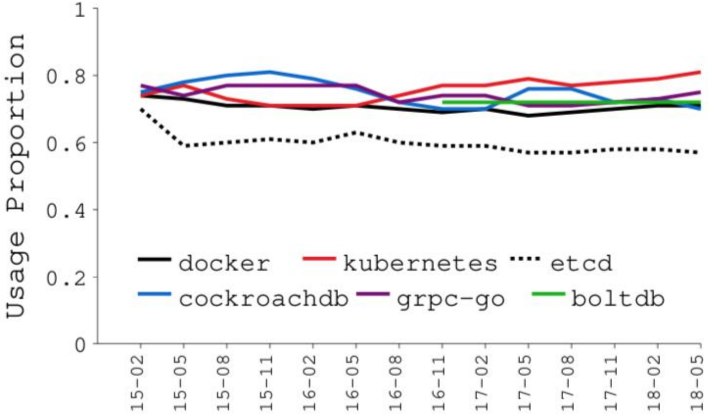


Figure 2. Usages of Shared-Memory Primitives over Time. *For each application, we calculate the proportion of shared-memory primitives over all primitives.*

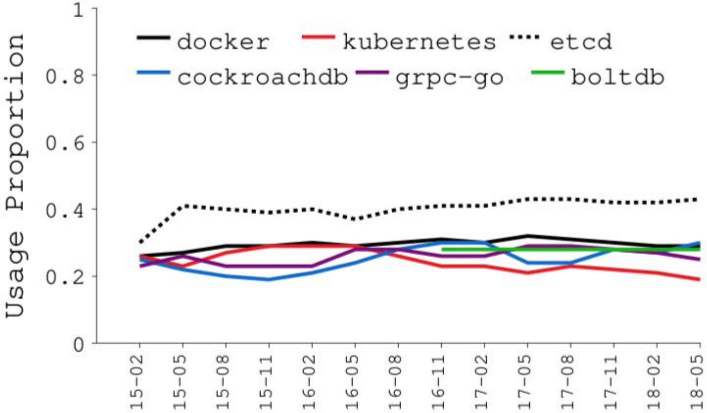


Figure 3. Usages of Message-Passing Primitives over Time. *For each application, we calculate the proportion of message-passing primitives over all primitives.*

Go并发BUG研究

bug存在的时间

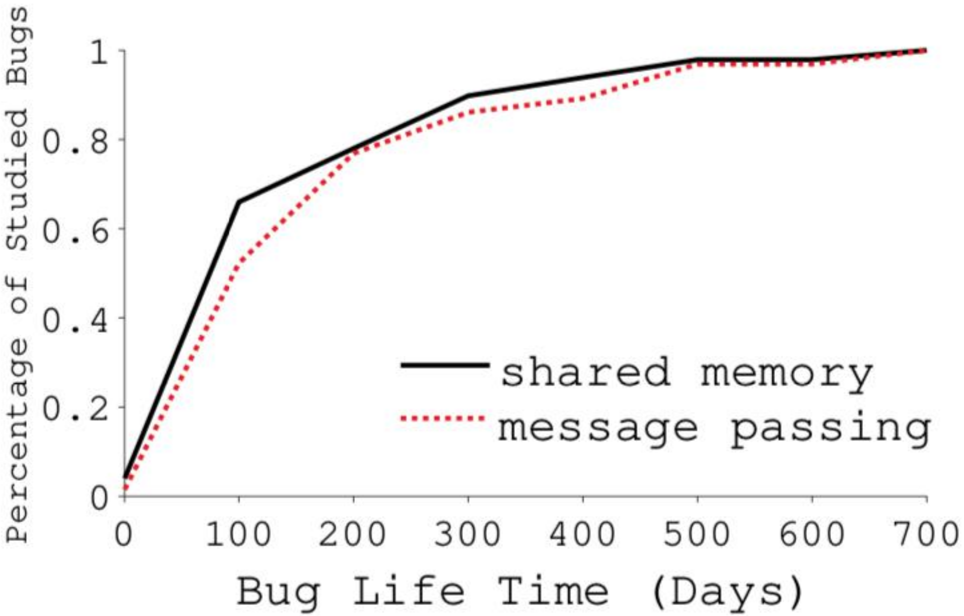
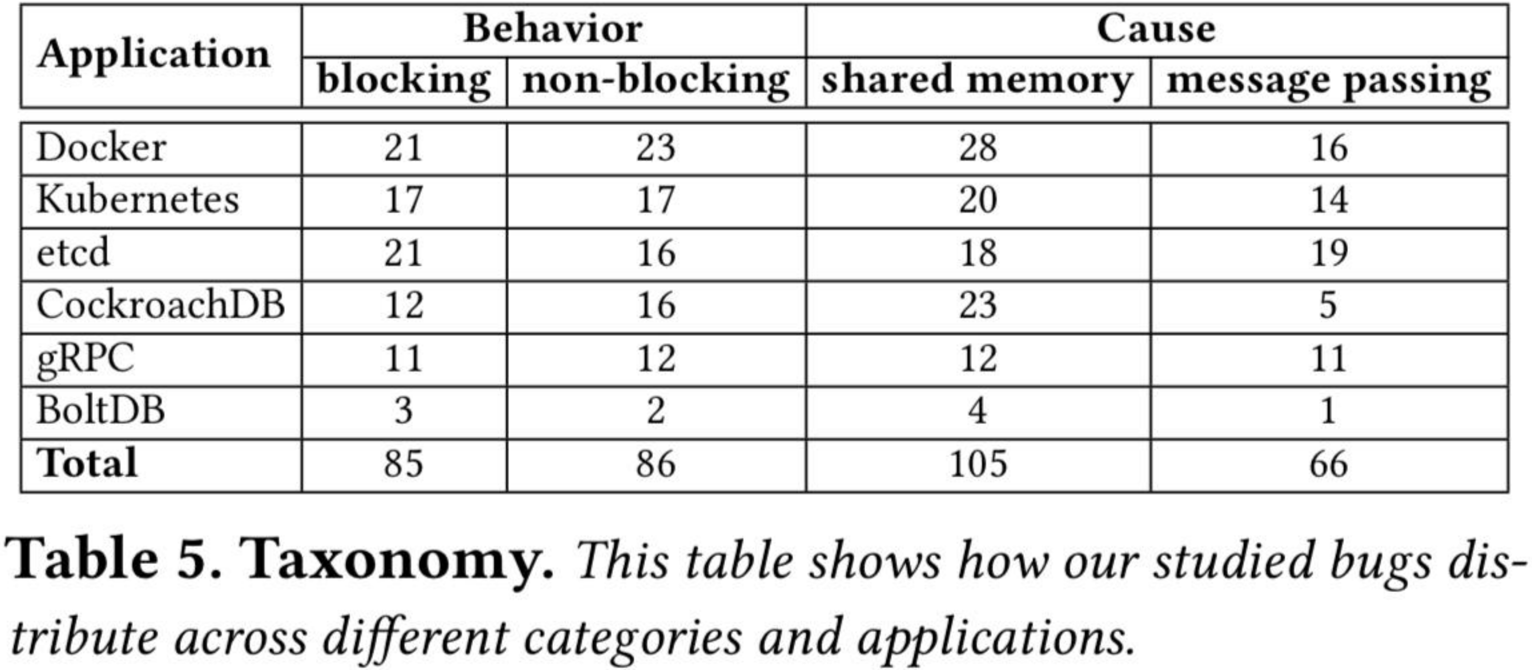


Figure 4. Bug Life Time. *The CDF of the life time of all shared-memory bugs and all message-passing bugs.*

Go并发BUG研究

bug 的分类数量

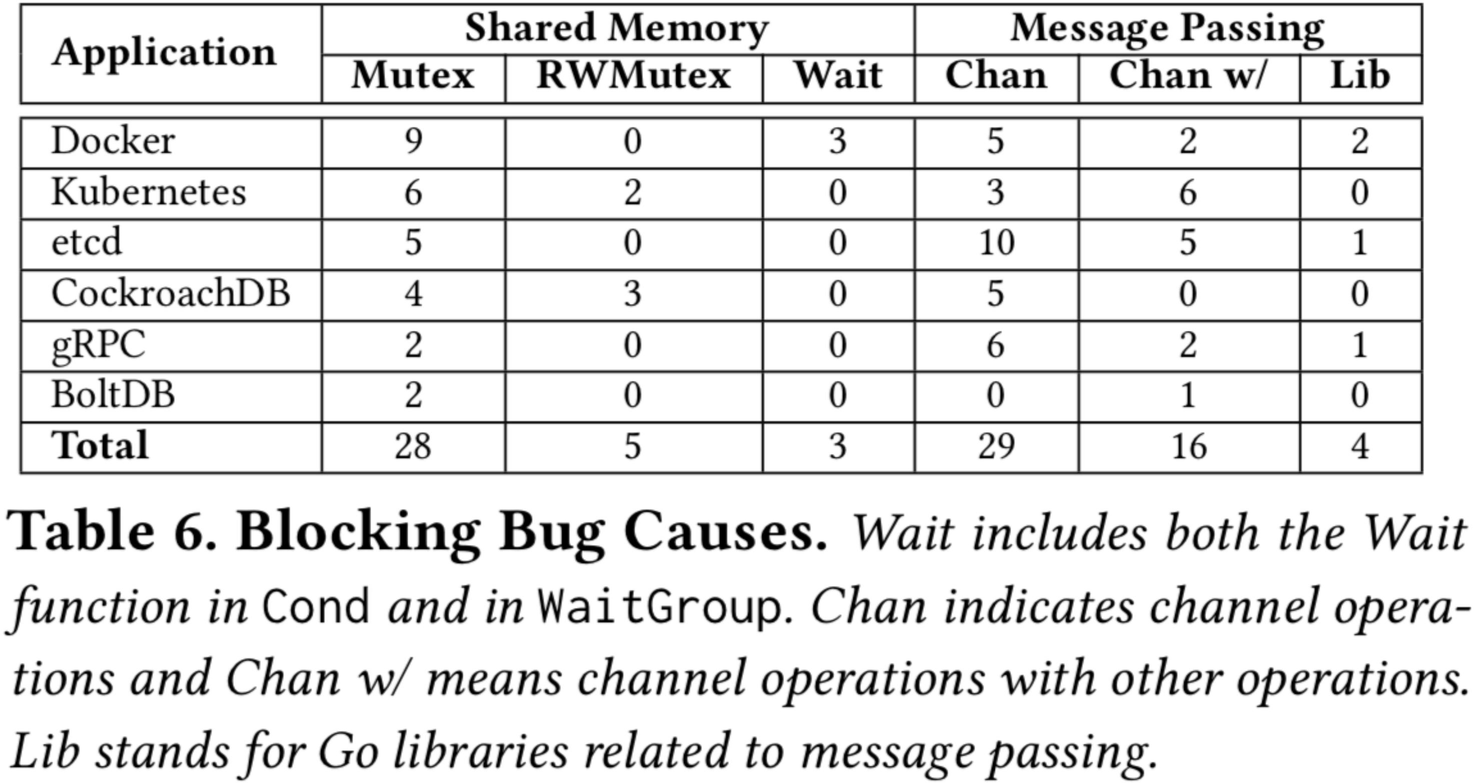


Application	Behavior		Cause	
	blocking	non-blocking	shared memory	message passing
Docker	21	23	28	16
Kubernetes	17	17	20	14
etcd	21	16	18	19
CockroachDB	12	16	23	5
gRPC	11	12	12	11
BoltDB	3	2	4	1
Total	85	86	105	66

Table 5. Taxonomy. *This table shows how our studied bugs distribute across different categories and applications.*

Go并发BUG研究

导致 blocking bugs 的原语

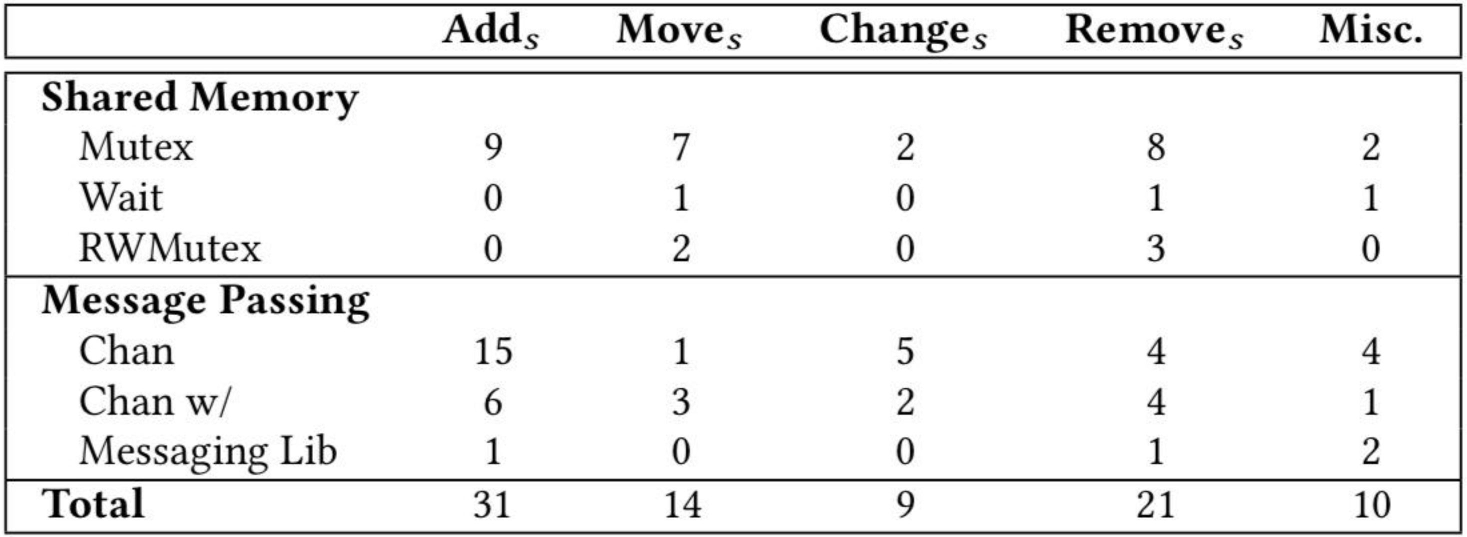


Application	Shared Memory			Message Passing		
	Mutex	**RWMutex**	**Wait**	**Chan**	**Chan w/**	**Lib**
Docker	9	0	3	5	2	2
Kubernetes	6	2	0	3	6	0
etcd	5	0	0	10	5	1
CockroachDB	4	3	0	5	0	0
gRPC	2	0	0	6	2	1
BoltDB	2	0	0	0	1	0
Total	28	5	3	29	16	4

Table 6. Blocking Bug Causes. *Wait includes both the Wait function in* `Cond` *and in* `WaitGroup`. *Chan indicates channel operations and Chan w/ means channel operations with other operations. Lib stands for Go libraries related to message passing.*

Go并发BUG研究

blocking bugs 的修复方式

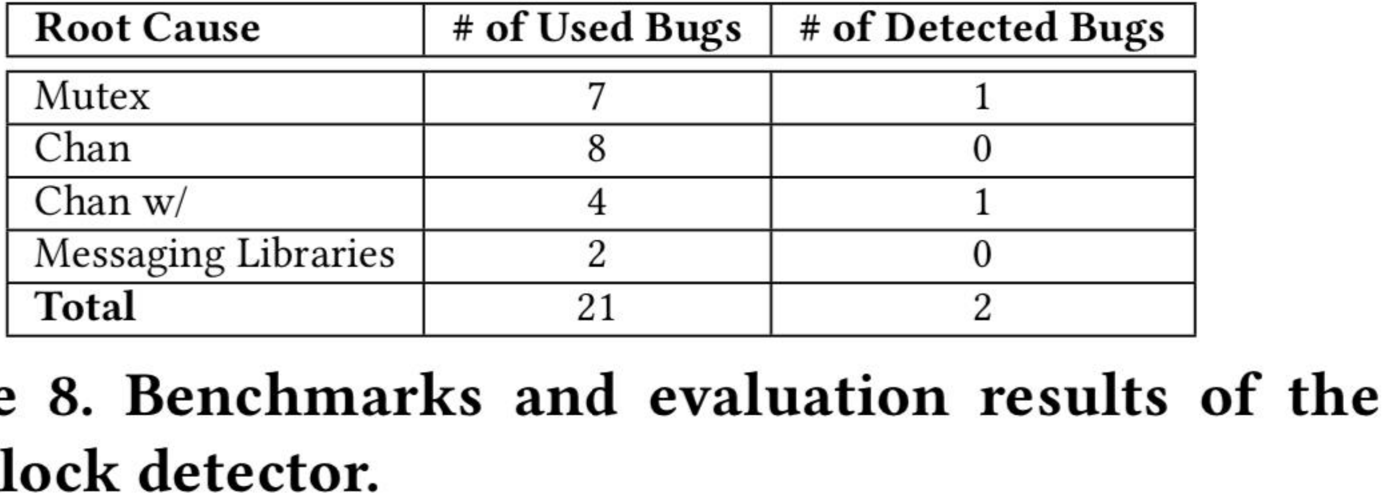


	Add_s	Move_s	Change_s	Remove_s	Misc.
Shared Memory					
Mutex	9	7	2	8	2
Wait	0	1	0	1	1
RWMutex	0	2	0	3	0
Message Passing					
Chan	15	1	5	4	4
Chan w/	6	3	2	4	1
Messaging Lib	1	0	0	1	2
Total	31	14	9	21	10

Table 7. Fix strategies for blocking bugs. *The subscript s stands for synchronization.*

Go并发BUG研究

能检测出的 blocking bugs

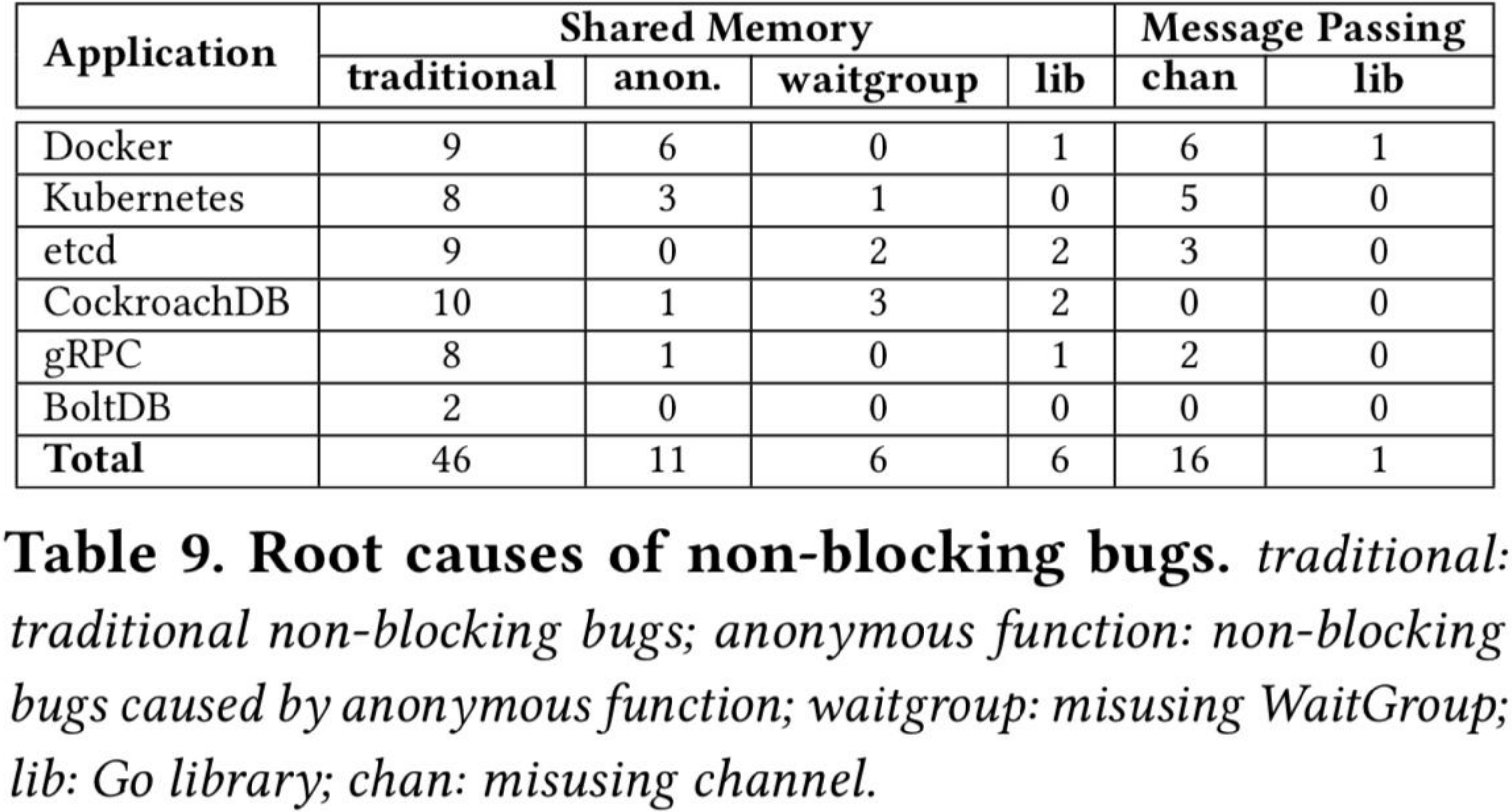


Root Cause	# of Used Bugs	# of Detected Bugs
Mutex	7	1
Chan	8	0
Chan w/	4	1
Messaging Libraries	2	0
Total	21	2

Table 8. Benchmarks and evaluation results of the deadlock detector.

Go并发BUG研究

导致 non-blocking bugs 原因

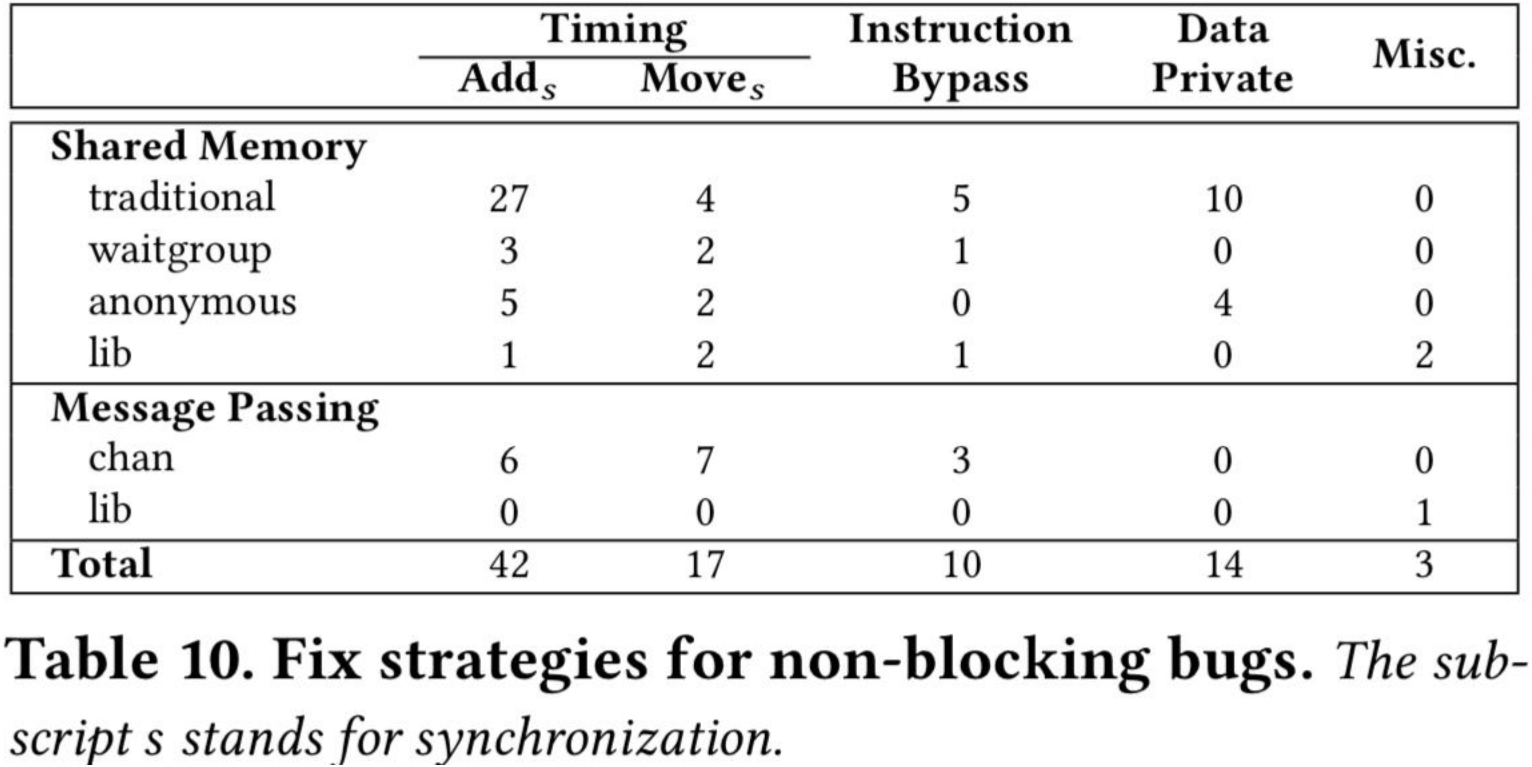


Application	Shared Memory				Message Passing	
	traditional	anon.	waitgroup	lib	chan	lib
Docker	9	6	0	1	6	1
Kubernetes	8	3	1	0	5	0
etcd	9	0	2	2	3	0
CockroachDB	10	1	3	2	0	0
gRPC	8	1	0	1	2	0
BoltDB	2	0	0	0	0	0
Total	46	11	6	6	16	1

Table 9. Root causes of non-blocking bugs. *traditional: traditional non-blocking bugs; anonymous function: non-blocking bugs caused by anonymous function; waitgroup: misusing WaitGroup; lib: Go library; chan: misusing channel.*

Go并发BUG研究

non-blocking bugs 的修复方式

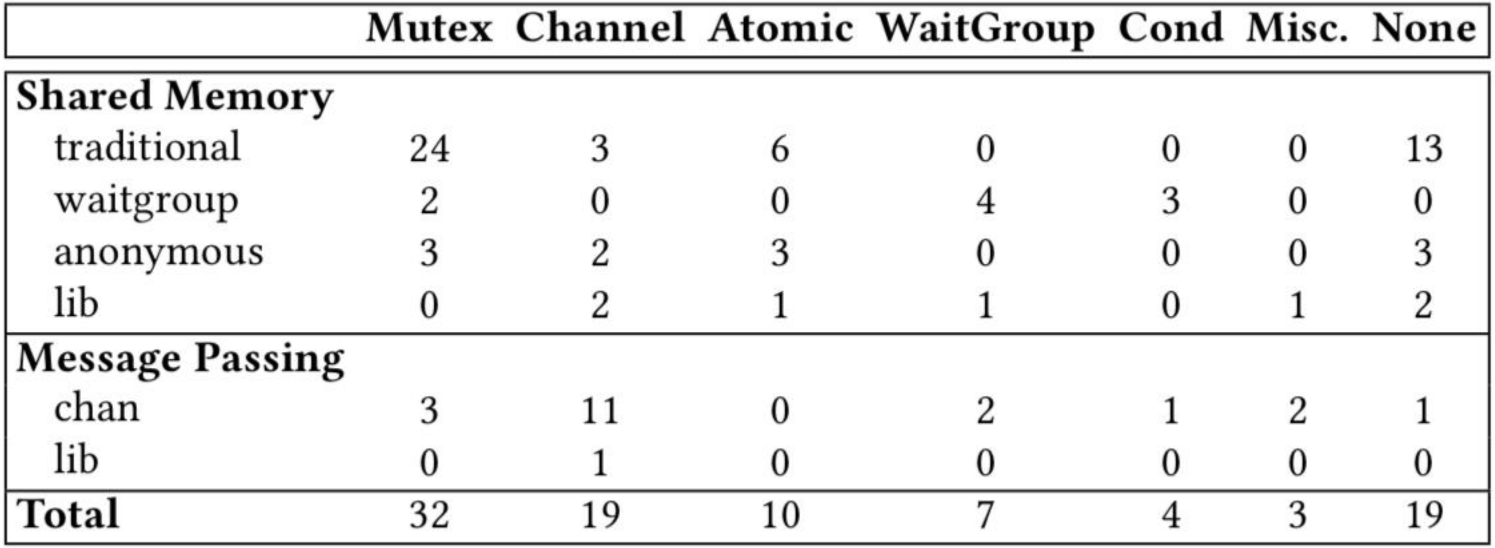


	Timing		Instruction	Data	Misc.
	Add$_s$	Move$_s$	Bypass	Private	
Shared Memory					
traditional	27	4	5	10	0
waitgroup	3	2	1	0	0
anonymous	5	2	0	4	0
lib	1	2	1	0	2
Message Passing					
chan	6	7	3	0	0
lib	0	0	0	0	1
Total	42	17	10	14	3

Table 10. Fix strategies for non-blocking bugs. *The subscript s stands for synchronization.*

Go并发BUG研究

non-blocking bugs 补丁中的原语

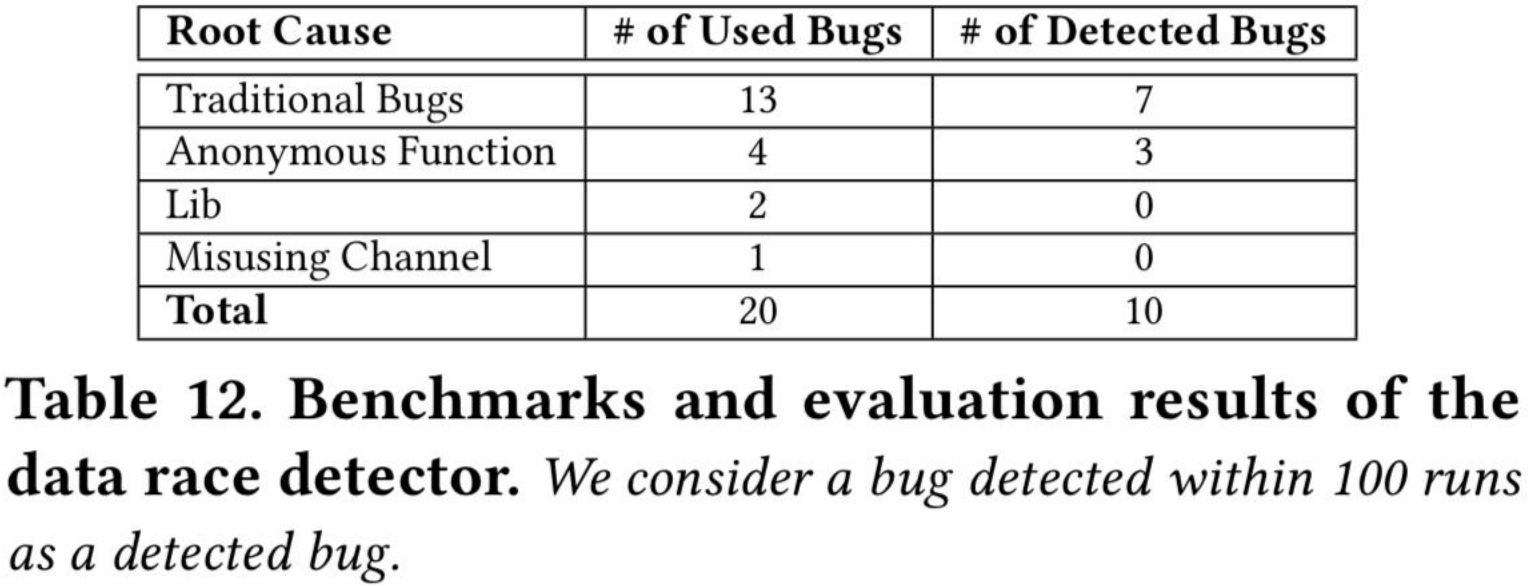


	Mutex	Channel	Atomic	WaitGroup	Cond	Misc.	None
Shared Memory							
traditional	24	3	6	0	0	0	13
waitgroup	2	0	0	4	3	0	0
anonymous	3	2	3	0	0	0	3
lib	0	2	1	1	0	1	2
Message Passing							
chan	3	11	0	2	1	2	1
lib	0	1	0	0	0	0	0
Total	32	19	10	7	4	3	19

Table 11. Synchronization primitives in patches of non-blocking bugs.

Go并发BUG研究

能检测出的 non-blocking bugs



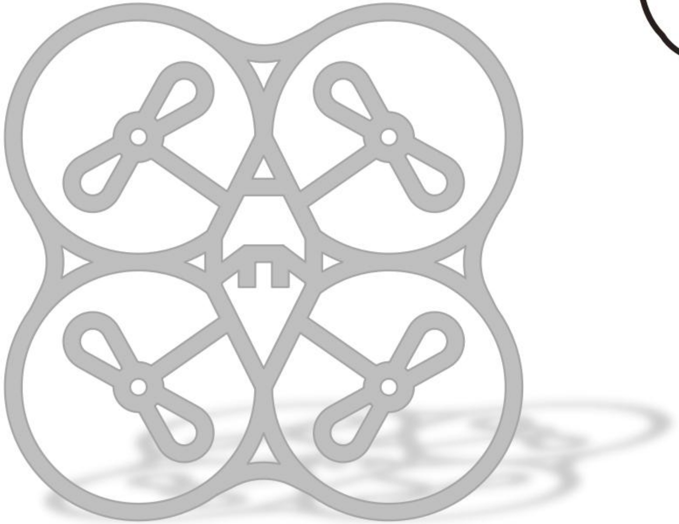
Root Cause	# of Used Bugs	# of Detected Bugs
Traditional Bugs	13	7
Anonymous Function	4	3
Lib	2	0
Misusing Channel	1	0
Total	20	10

Table 12. Benchmarks and evaluation results of the data race detector. *We consider a bug detected within 100 runs as a detected bug.*

01

同步原语

- ✓ 基本同步原语
- ✓ 扩展的同步原语
- ✓ 原子操作



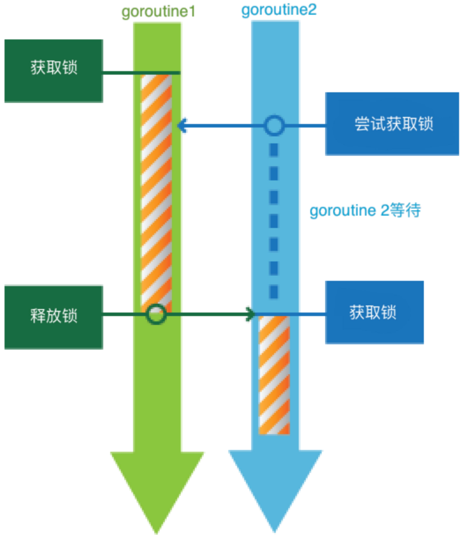
基本同步原语

基本同步原语

Mutex

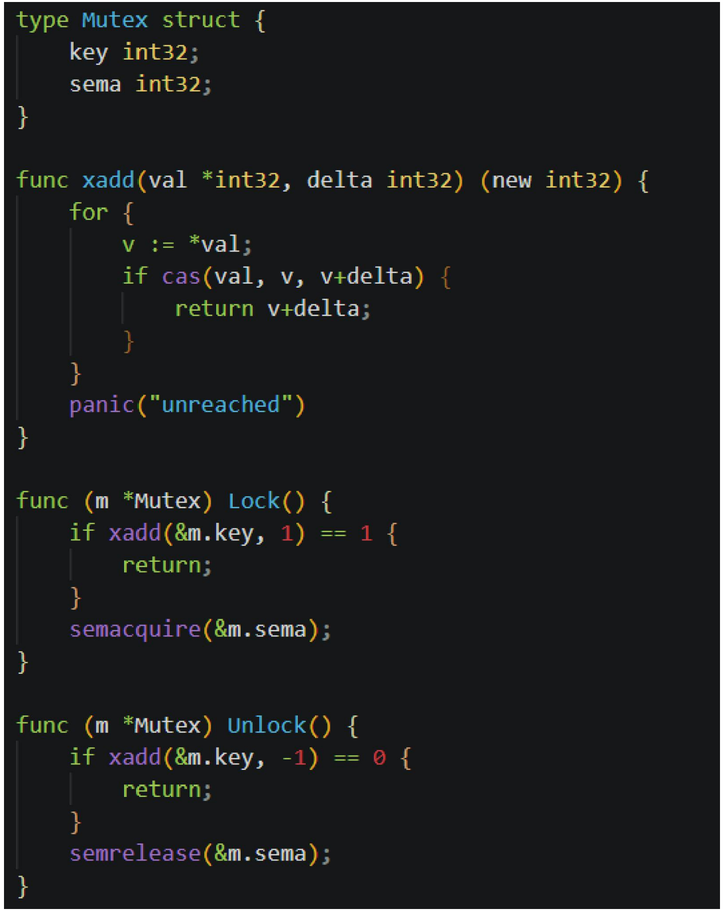
- 互斥锁 **Mutual exclusion**
- 任何时间只允许一个goroutine在临界区域运行
- 使用时避免死锁
- 追求公平

- 零值是未锁状态
- Unlock未加锁的Mutex会panic
- 加锁的Mutex不和这个特定的goroutine关联
- 非重入锁



基本同步原语

Mutex 初版(2008)



```go
type Mutex struct {
    key int32;
    sema int32;
}

func xadd(val *int32, delta int32) (new int32) {
    for {
        v := *val;
        if cas(val, v, v+delta) {
            return v+delta;
        }
    }
    panic("unreached")
}

func (m *Mutex) Lock() {
    if xadd(&m.key, 1) == 1 {
        return;
    }
    semacquire(&m.sema);
}

func (m *Mutex) Unlock() {
    if xadd(&m.key, -1) == 0 {
        return;
    }
    semrelease(&m.sema);
}
```

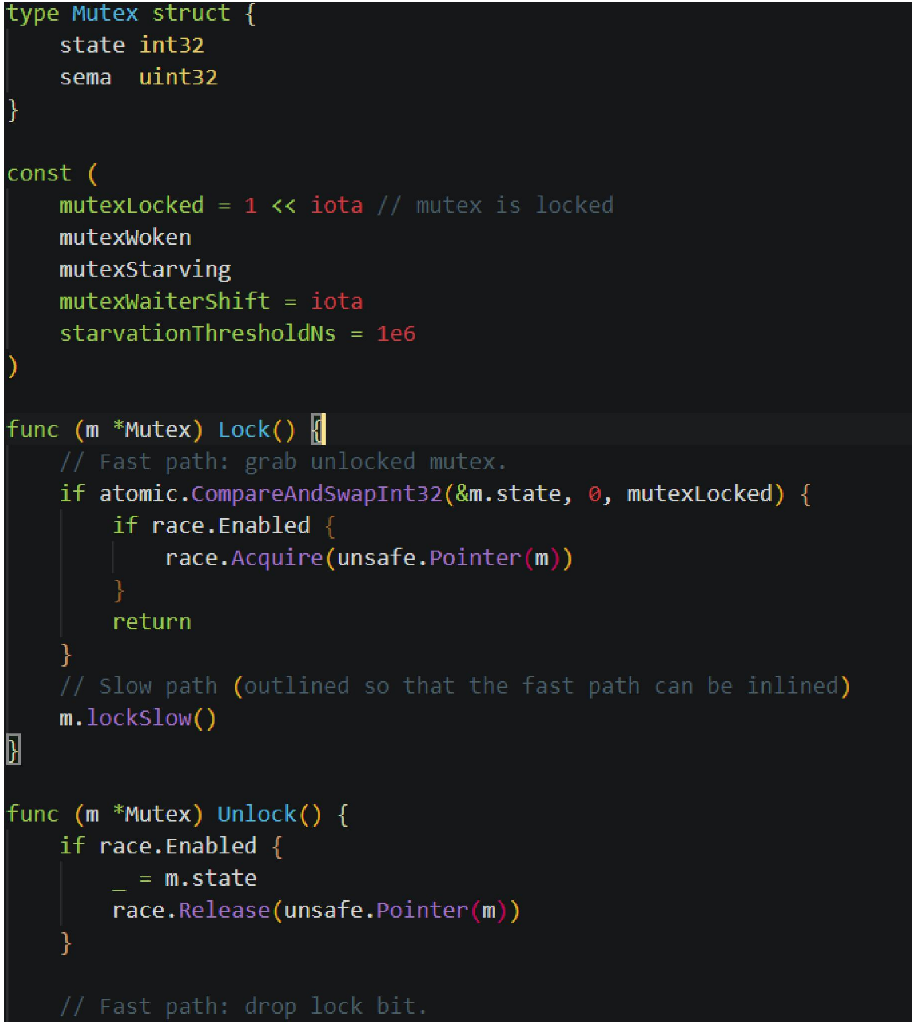
基本同步原语

Mutex 演化

- **2012年**， commit dd2074c8做了一次大的改动，它将waiter count(等待者的数量)和锁标识分开来(内部实现还是合用使用state字段)。新来的 goroutine 占优势，会有更大的机会获取锁。
- **2015年**， commit edcad863, Go 1.5中mutex实现为全协作式的，增加了spin机制，一旦有竞争，当前goroutine就会进入调度器。在临界区执行很短的情况下可能不是最好的解决方案。
- **2016年**，commit 0556e262, Go 1.9中增加了饥饿模式，让锁变得更公平，不公平的等待时间限制在1毫秒，并且修复了一个大bug,唤醒的goroutine总是放在等待队列的尾部会导致更加不公平的等待时间。
- **2019年**，commit 41cb0ae inline优化，将slow path抽取出来，保留fast path以便内联优化

基本同步原语

Mutex 当前实现



```go
type Mutex struct {
    state int32
    sema  uint32
}

const (
    mutexLocked = 1 << iota // mutex is locked
    mutexWoken
    mutexStarving
    mutexWaiterShift = iota
    starvationThresholdNs = 1e6
)

func (m *Mutex) Lock() {
    // Fast path: grab unlocked mutex.
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        if race.Enabled {
            race.Acquire(unsafe.Pointer(m))
        }
        return
    }
    // Slow path (outlined so that the fast path can be inlined)
    m.lockSlow()
}

func (m *Mutex) Unlock() {
    if race.Enabled {
        _ = m.state
        race.Release(unsafe.Pointer(m))
    }

    // Fast path: drop lock bit.
```

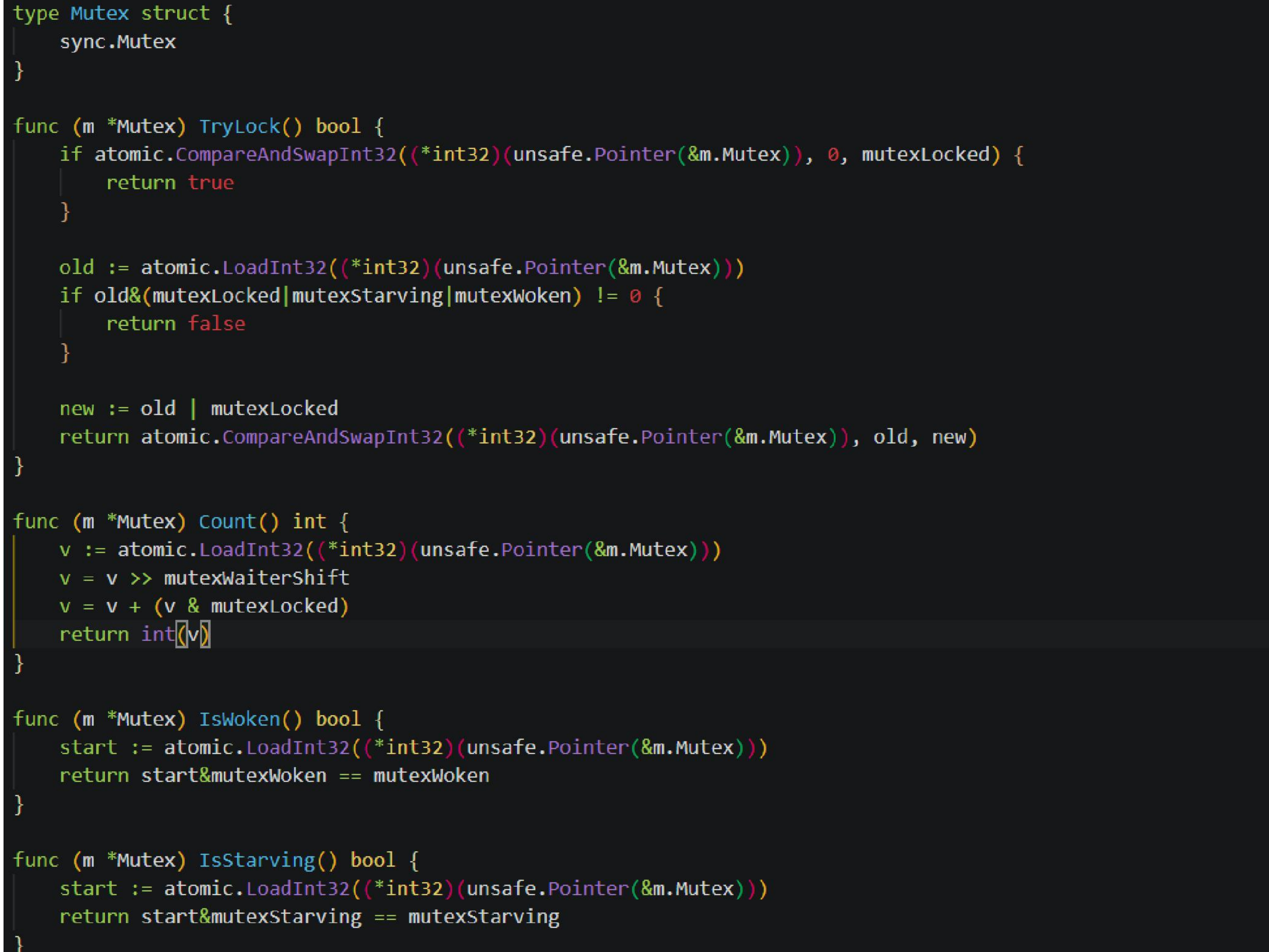
基本同步原语

Mutex 当前实现的逻辑

- 互斥锁有两种状态：正常状态和饥饿状态。
- 在正常状态下，所有等待锁的goroutine按照FIFO顺序等待。唤醒的goroutine不会直接拥有锁，而是会和新请求锁的goroutine竞争锁的拥有。如果一个等待的goroutine超过1ms没有获取锁，那么它将会把锁转变为饥饿模式。
- 在饥饿模式下，锁的所有权将从unlock的gorutine直接交给交给等待队列中的第一个。新来的goroutine将不会尝试去获得锁，即使锁看起来是unlock状态, 也不会去尝试自旋操作，而是放在等待队列的尾部。
- 如果一个等待的goroutine获取了锁，并且满足一以下其中的任何一个条件：(1)它是队列中的最后一个；(2)它等待的时候小于1ms。它会将锁的状态转换为正常状态。

基本同步原语

Mutex 扩展



```go
type Mutex struct {
    sync.Mutex
}

func (m *Mutex) TryLock() bool {
    if atomic.CompareAndSwapInt32((*int32)(unsafe.Pointer(&m.Mutex)), 0, mutexLocked) {
        return true
    }

    old := atomic.LoadInt32((*int32)(unsafe.Pointer(&m.Mutex)))
    if old&(mutexLocked|mutexStarving|mutexWoken) != 0 {
        return false
    }

    new := old | mutexLocked
    return atomic.CompareAndSwapInt32((*int32)(unsafe.Pointer(&m.Mutex)), old, new)
}

func (m *Mutex) Count() int {
    v := atomic.LoadInt32((*int32)(unsafe.Pointer(&m.Mutex)))
    v = v >> mutexWaiterShift
    v = v + (v & mutexLocked)
    return int(v)
}

func (m *Mutex) IsWoken() bool {
    start := atomic.LoadInt32((*int32)(unsafe.Pointer(&m.Mutex)))
    return start&mutexWoken == mutexWoken
}

func (m *Mutex) IsStarving() bool {
    start := atomic.LoadInt32((*int32)(unsafe.Pointer(&m.Mutex)))
    return start&mutexStarving == mutexStarving
}
```

基本同步原语

Mutex 前人踩的坑

- gRPC #12345
- etcd #12345
- Docker #12345

基本同步原语

RWMutex

- 可以被一堆的reader持有，或者被一个writer持有
- 适合大并发read的场景
- 零值是未加锁的状态
- writer的Lock相对后续的reader的RLock优先级高
- 禁止递归读锁

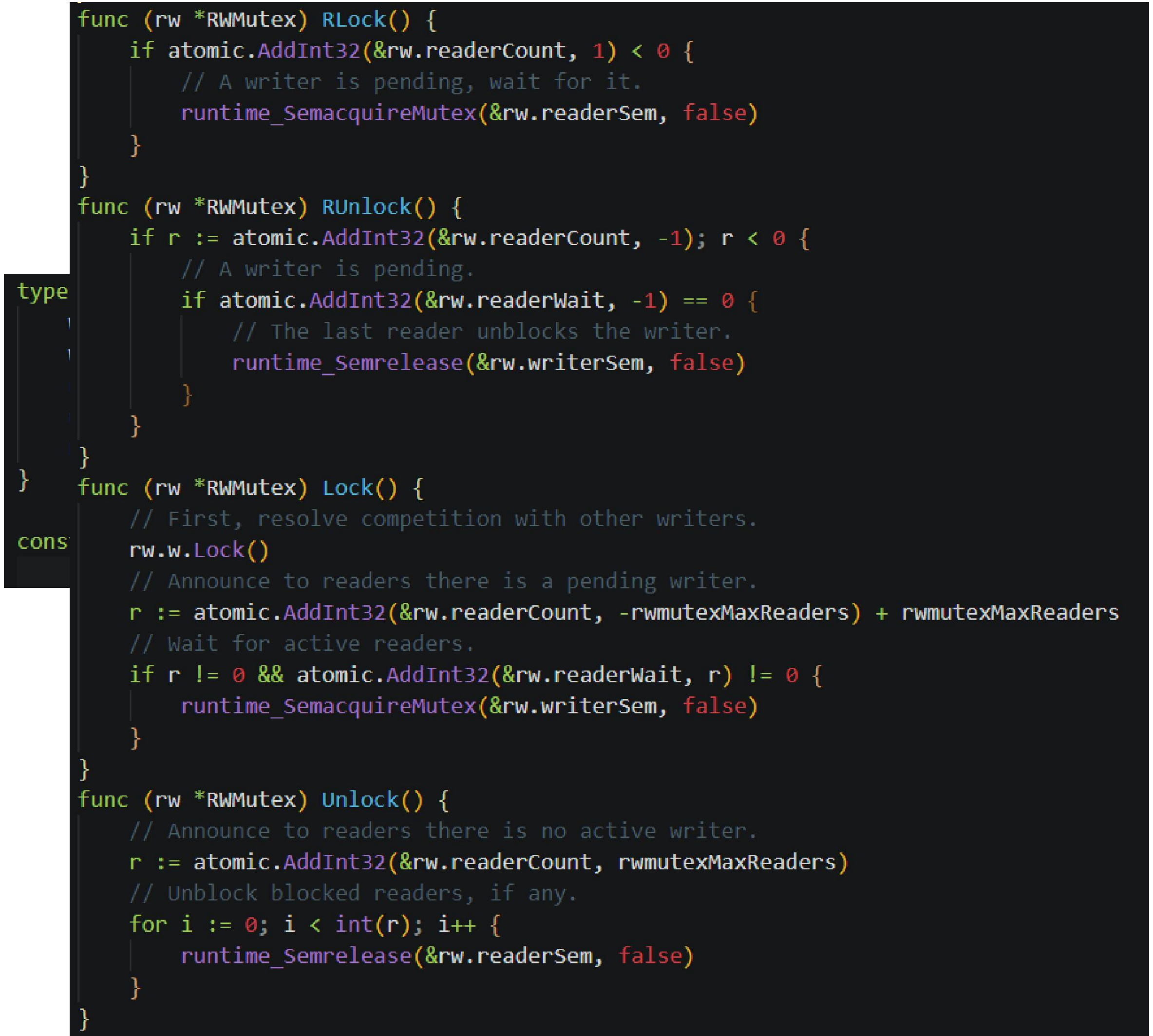
基本同步原语

RWMutex

- 可以被一堆的reader持有，或者被一个writer持有
- 适合大并发read的场景
- 零值是未加锁的状态
- writer的Lock相对后续的reader的RLock优先级高
- 禁止递归读锁

基本同步原语

RWMutex 当前实现

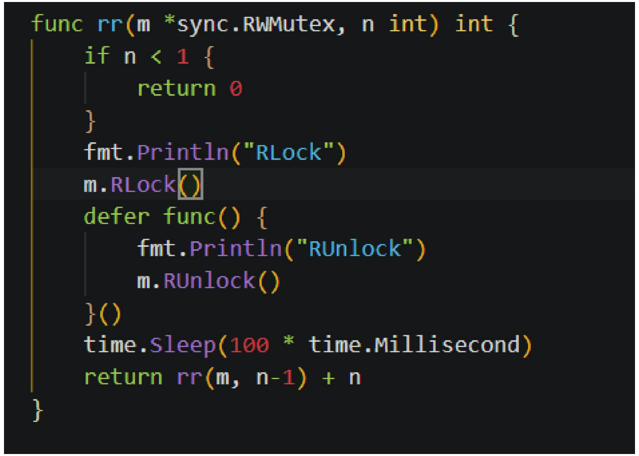


```
type

}

cons
```

```go
func (rw *RWMutex) RLock() {
    if atomic.AddInt32(&rw.readerCount, 1) < 0 {
        // A writer is pending, wait for it.
        runtime_SemacquireMutex(&rw.readerSem, false)
    }
}
func (rw *RWMutex) RUnlock() {
    if r := atomic.AddInt32(&rw.readerCount, -1); r < 0 {
        // A writer is pending.
        if atomic.AddInt32(&rw.readerWait, -1) == 0 {
            // The last reader unblocks the writer.
            runtime_Semrelease(&rw.writerSem, false)
        }
    }
}
func (rw *RWMutex) Lock() {
    // First, resolve competition with other writers.
    rw.w.Lock()
    // Announce to readers there is a pending writer.
    r := atomic.AddInt32(&rw.readerCount, -rwmutexMaxReaders) + rwmutexMaxReaders
    // Wait for active readers.
    if r != 0 && atomic.AddInt32(&rw.readerWait, r) != 0 {
        runtime_SemacquireMutex(&rw.writerSem, false)
    }
}
func (rw *RWMutex) Unlock() {
    // Announce to readers there is no active writer.
    r := atomic.AddInt32(&rw.readerCount, rwmutexMaxReaders)
    // Unblock blocked readers, if any.
    for i := 0; i < int(r); i++ {
        runtime_Semrelease(&rw.readerSem, false)
    }
}
```

基本同步原语

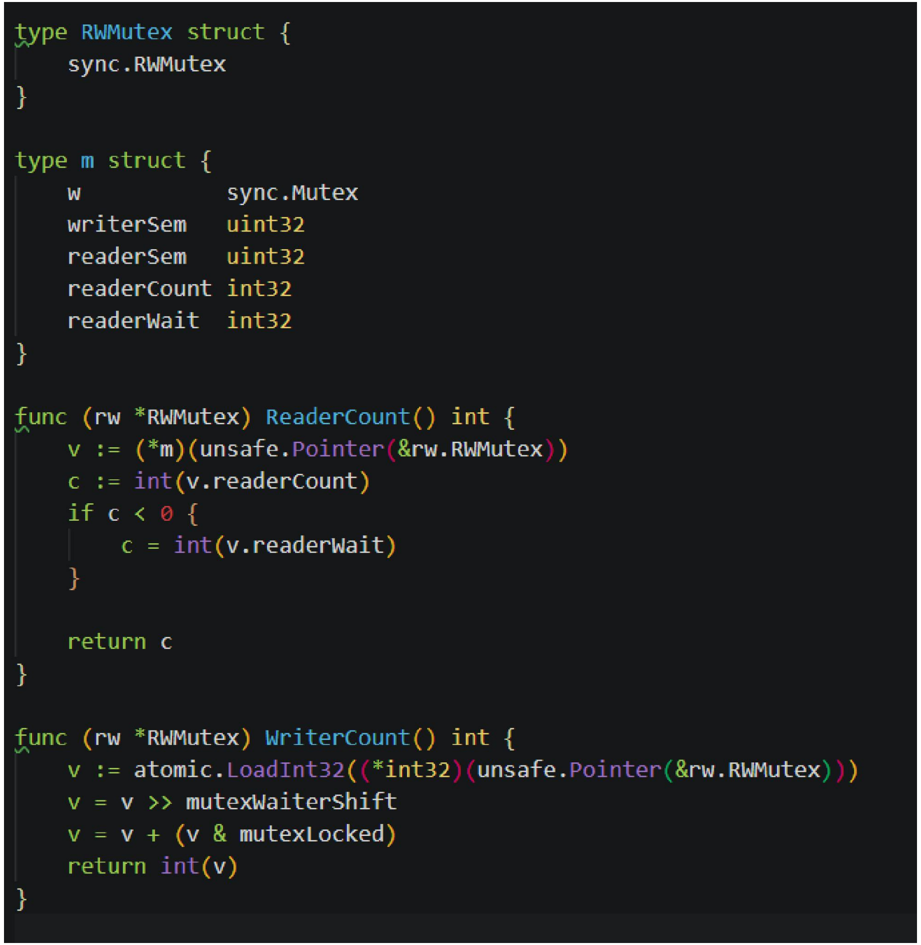
RWMutex 递归的坑



```go
func rr(m *sync.RWMutex, n int) int {
    if n < 1 {
        return 0
    }
    fmt.Println("RLock")
    m.RLock()
    defer func() {
        fmt.Println("RUnlock")
        m.RUnlock()
    }()
    time.Sleep(100 * time.Millisecond)
    return rr(m, n-1) + n
}
```

基本同步原语

RWMutex 扩展



```go
type RWMutex struct {
    sync.RWMutex
}

type m struct {
    w           sync.Mutex
    writerSem   uint32
    readerSem   uint32
    readerCount int32
    readerWait  int32
}

func (rw *RWMutex) ReaderCount() int {
    v := (*m)(unsafe.Pointer(&rw.RWMutex))
    c := int(v.readerCount)
    if c < 0 {
        c = int(v.readerWait)
    }

    return c
}

func (rw *RWMutex) WriterCount() int {
    v := atomic.LoadInt32((*int32)(unsafe.Pointer(&rw.RWMutex)))
    v = v >> mutexWaiterShift
    v = v + (v & mutexLocked)
    return int(v)
}
```

基本同步原语

RWMutex 前人踩过得坑

- gRPC #12345
- etcd #12345
- Docker #12345

基本同步原语

Cond

- Mutex有些情况下不适用(通知机制)
- Monitor vs. Mutex， Monitor= Mutex + Condition Variables
- Condition variable是一组等待同一个条件的goroutine的容器

- 每个Cond和一个Locker相关联
- 改变条件或者调用Wait需要获取锁

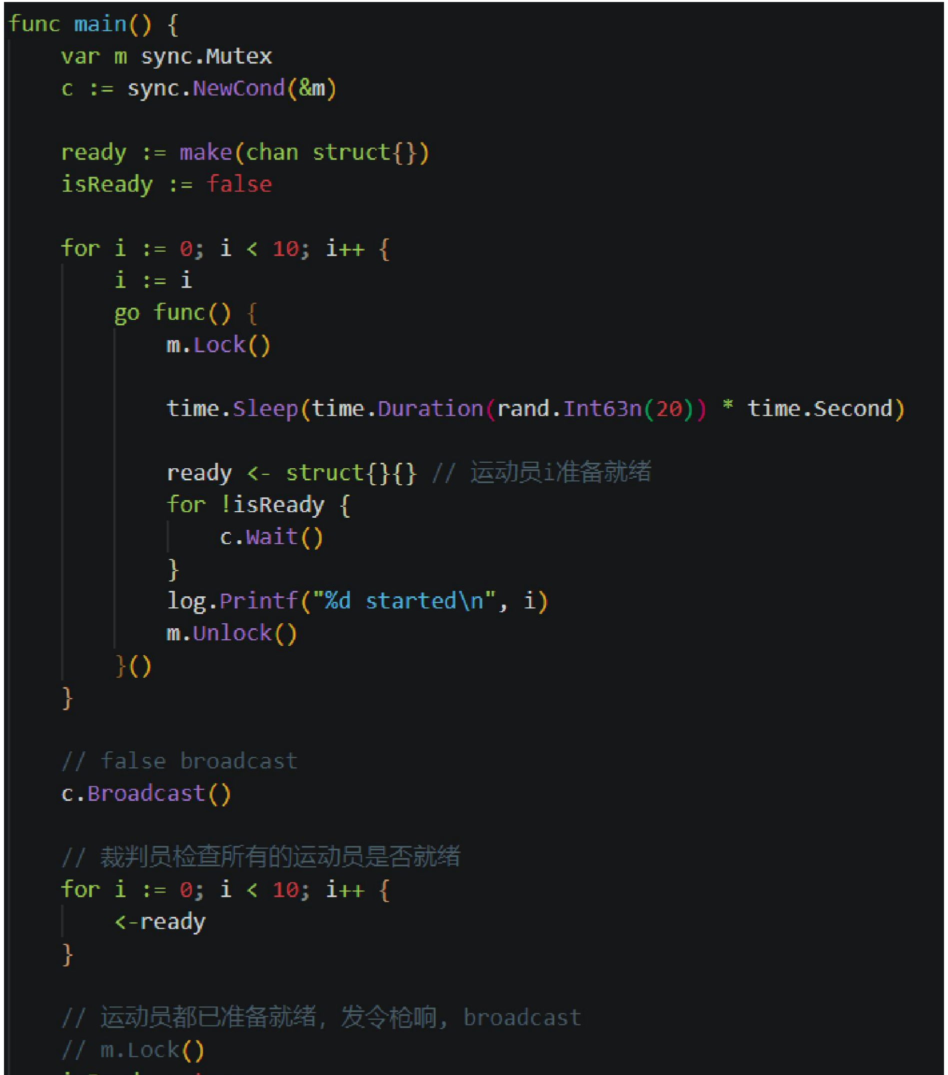
基本同步原语

Cond

func (*Cond) Broadcast

func (*Cond) Signal

func (*Cond) Wait



```
func main() {
    var m sync.Mutex
    c := sync.NewCond(&m)

    ready := make(chan struct{})
    isReady := false

    for i := 0; i < 10; i++ {
        i := i
        go func() {
            m.Lock()

            time.Sleep(time.Duration(rand.Int63n(20)) * time.Second)

            ready <- struct{}{} // 运动员i准备就绪
            for !isReady {
                c.Wait()
            }
            log.Printf("%d started\n", i)
            m.Unlock()
        }()
    }

    // false broadcast
    c.Broadcast()

    // 裁判员检查所有的运动员是否就绪
    for i := 0; i < 10; i++ {
        <-ready
    }

    // 运动员都已准备就绪，发令枪响，broadcast
    // m.Lock()
```

基本同步原语

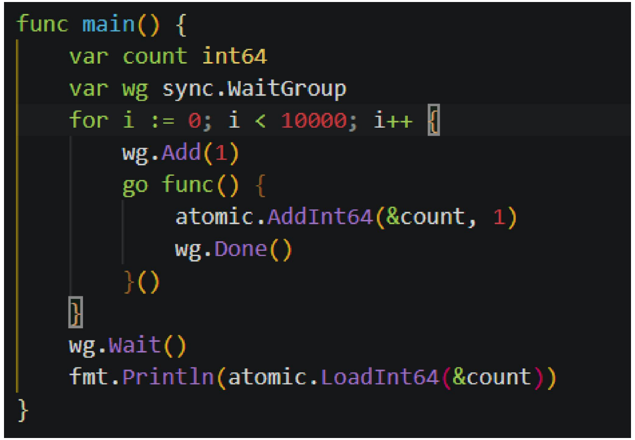
Waitgroup

- 等待一组goroutine完成 (Java CountdownLatch/CyclicBarrier)
- Add参数可以是负值；如果计数器小于0, panic
- 当计数器为0的时候，阻塞在Wait方法的goroutine都会被释放
- 可重用，但是……

基本同步原语

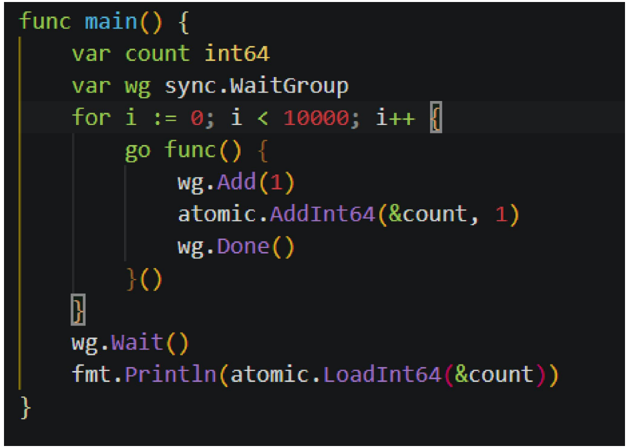
Waitgroup Add一定要在Wait之前设置好

✓



```go
func main() {
    var count int64
    var wg sync.WaitGroup
    for i := 0; i < 10000; i++ {
        wg.Add(1)
        go func() {
            atomic.AddInt64(&count, 1)
            wg.Done()
        }()
    }
    wg.Wait()
    fmt.Println(atomic.LoadInt64(&count))
}
```

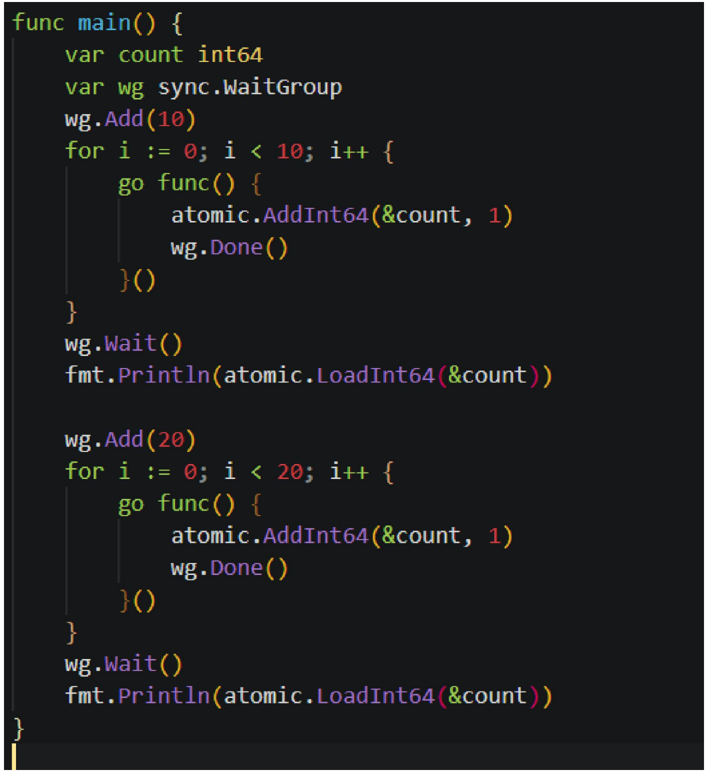
✗



```go
func main() {
    var count int64
    var wg sync.WaitGroup
    for i := 0; i < 10000; i++ {
        go func() {
            wg.Add(1)
            atomic.AddInt64(&count, 1)
            wg.Done()
        }()
    }
    wg.Wait()
    fmt.Println(atomic.LoadInt64(&count))
}
```

基本同步原语

Waitgroup 一定条件下可重用

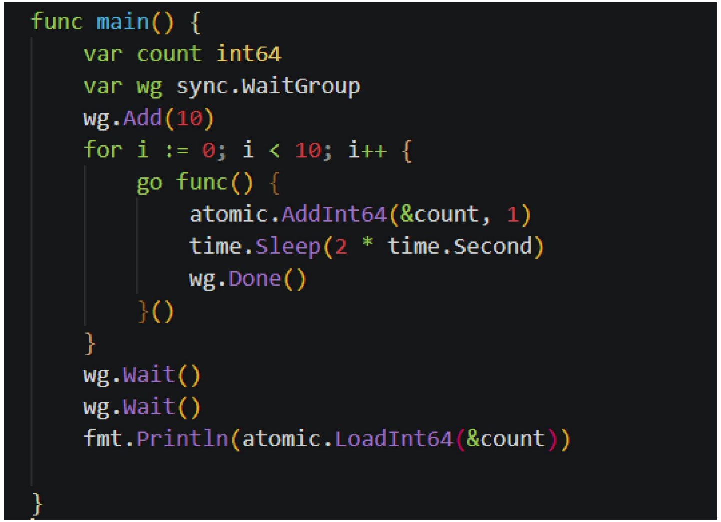


```go
func main() {
    var count int64
    var wg sync.WaitGroup
    wg.Add(10)
    for i := 0; i < 10; i++ {
        go func() {
            atomic.AddInt64(&count, 1)
            wg.Done()
        }()
    }
    wg.Wait()
    fmt.Println(atomic.LoadInt64(&count))

    wg.Add(20)
    for i := 0; i < 20; i++ {
        go func() {
            atomic.AddInt64(&count, 1)
            wg.Done()
        }()
    }
    wg.Wait()
    fmt.Println(atomic.LoadInt64(&count))
}
```

基本同步原语

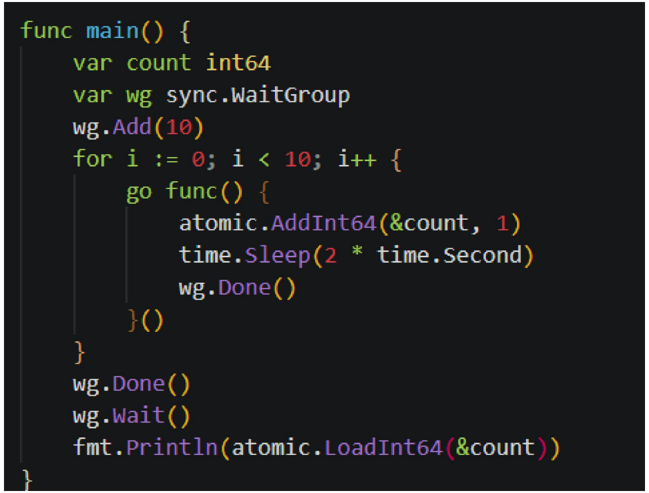
Waitgroup 多次Wait和多次Done

✓



```go
func main() {
    var count int64
    var wg sync.WaitGroup
    wg.Add(10)
    for i := 0; i < 10; i++ {
        go func() {
            atomic.AddInt64(&count, 1)
            time.Sleep(2 * time.Second)
            wg.Done()
        }()
    }
    wg.Wait()
    wg.Wait()
    fmt.Println(atomic.LoadInt64(&count))

}
```

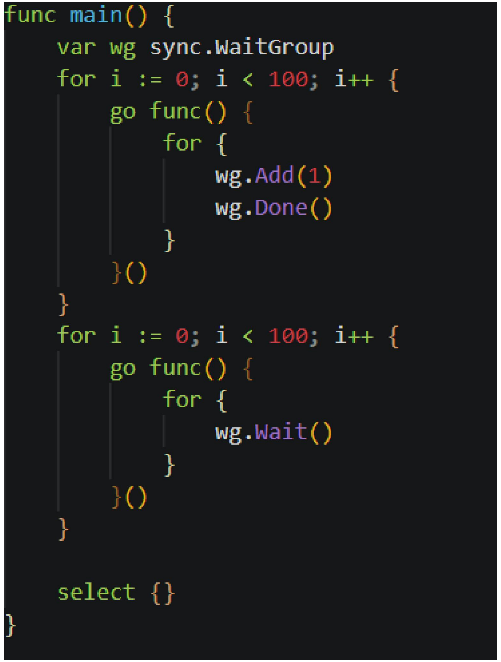
✗



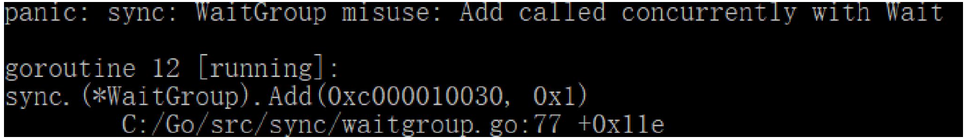
```go
func main() {
    var count int64
    var wg sync.WaitGroup
    wg.Add(10)
    for i := 0; i < 10; i++ {
        go func() {
            atomic.AddInt64(&count, 1)
            time.Sleep(2 * time.Second)
            wg.Done()
        }()
    }
    wg.Done()
    wg.Wait()
    fmt.Println(atomic.LoadInt64(&count))
}
```

基本同步原语

Waitgroup Wait和Add并发调用



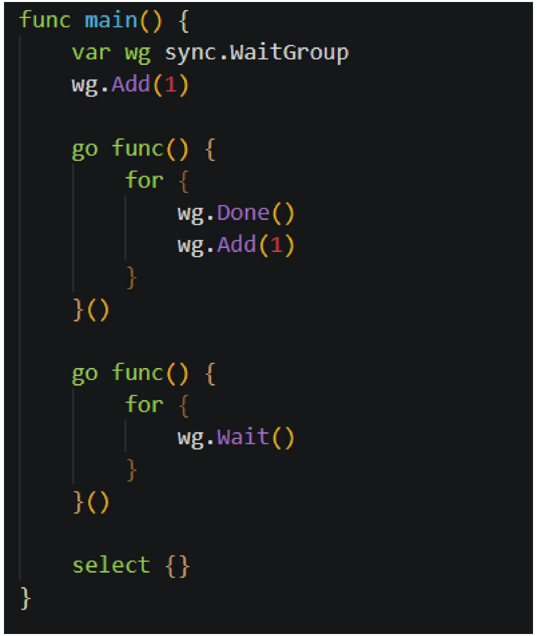
```go
func main() {
    var wg sync.WaitGroup
    for i := 0; i < 100; i++ {
        go func() {
            for {
                wg.Add(1)
                wg.Done()
            }
        }()
    }
    for i := 0; i < 100; i++ {
        go func() {
            for {
                wg.Wait()
            }
        }()
    }

    select {}
}
```



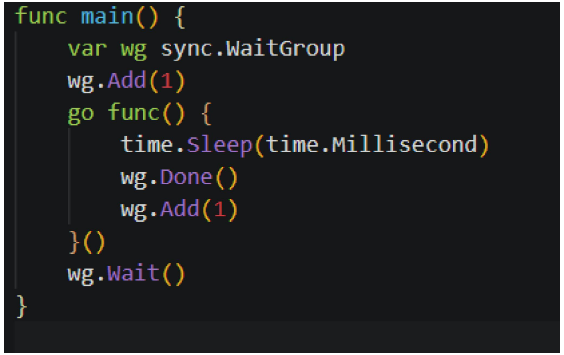
```
panic: sync: WaitGroup misuse: Add called concurrently with Wait

goroutine 12 [running]:
sync.(*WaitGroup).Add(0xc000010030, 0x1)
        C:/Go/src/sync/waitgroup.go:77 +0x11e
```

基本同步原语

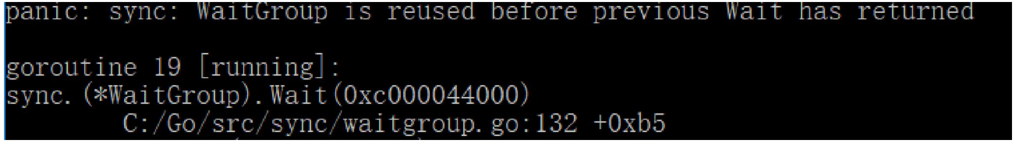
Waitgroup Wait未完成时就调用Add



```go
func main() {
    var wg sync.WaitGroup
    wg.Add(1)

    go func() {
        for {
            wg.Done()
            wg.Add(1)
        }
    }()

    go func() {
        for {
            wg.Wait()
        }
    }()

    select {}
}
```



```go
func main() {
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        time.Sleep(time.Millisecond)
        wg.Done()
        wg.Add(1)
    }()
    wg.Wait()
}
```



```
panic: sync: WaitGroup is reused before previous Wait has returned

goroutine 19 [running]:
sync.(*WaitGroup).Wait(0xc000044000)
        C:/Go/src/sync/waitgroup.go:132 +0xb5
```

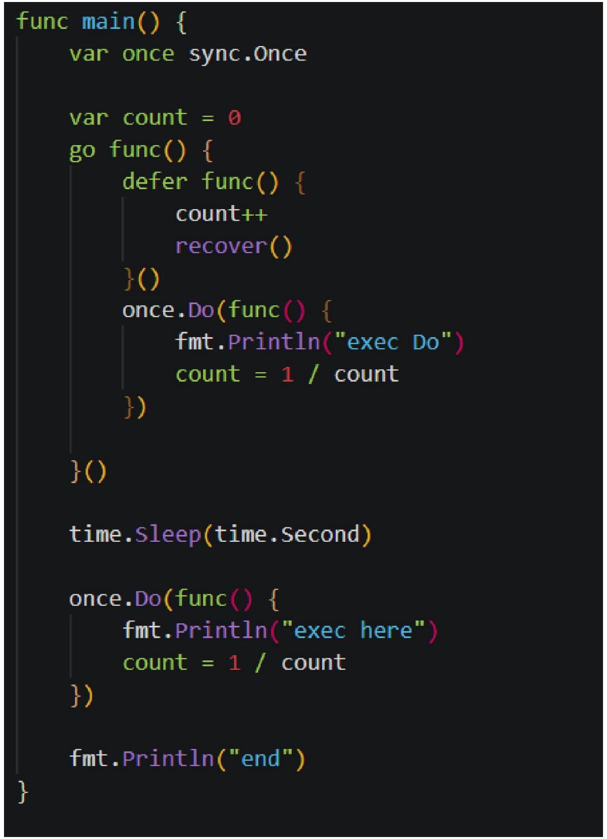
基本同步原语

Once

- 只执行一次初始化
- 避免死锁
- 即使f panic, Once也认为它完成了

基本同步原语

Once f panic也被认为初始化完成



```go
func main() {
    var once sync.Once

    var count = 0
    go func() {
        defer func() {
            count++
            recover()
        }()
        once.Do(func() {
            fmt.Println("exec Do")
            count = 1 / count
        })

    }()

    time.Sleep(time.Second)

    once.Do(func() {
        fmt.Println("exec here")
        count = 1 / count
    })

    fmt.Println("end")
}
```

基本同步原语

Once f内不要再调用此once

基本同步原语

单例

- package级别的常量
- package 变量 (eager)
- init函数 (eager)
- GetInstance() (lazy)
- 通过sync.Once或者类似实现

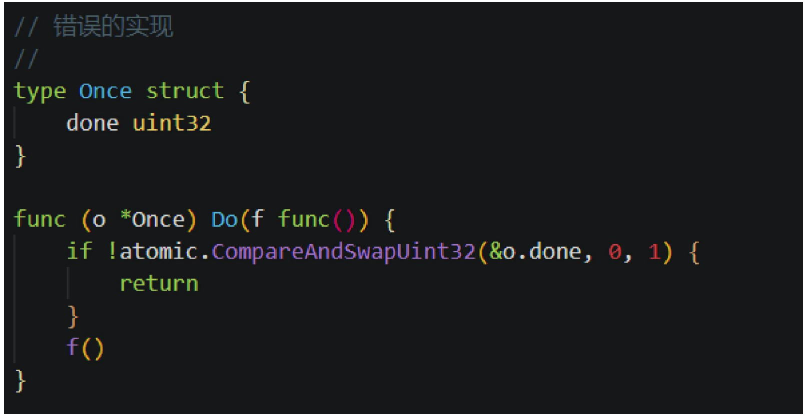
基本同步原语

单例的问题

- io.EOF, http.DefaultClient认为修改

基本同步原语

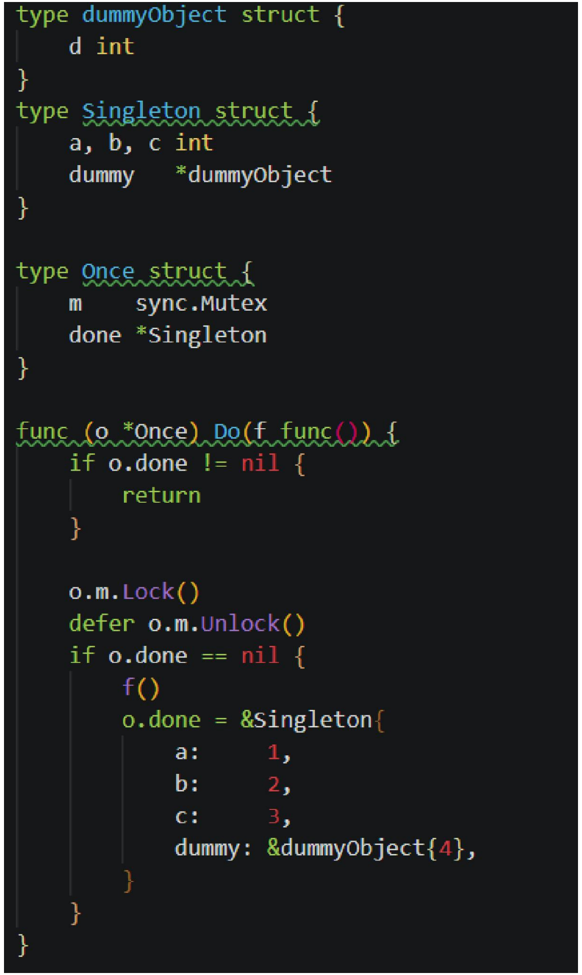
单例 错误的实现



```go
// 错误的实现
//
type Once struct {
    done uint32
}

func (o *Once) Do(f func()) {
    if !atomic.CompareAndSwapUint32(&o.done, 0, 1) {
        return
    }
    f()
}
```

基本同步原语

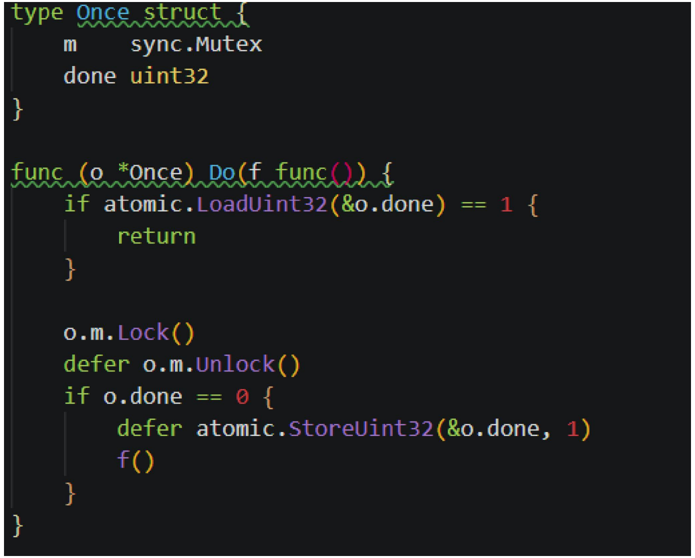
单例 错误的实现2



```go
type dummyObject struct {
    d int
}
type Singleton struct {
    a, b, c int
    dummy   *dummyObject
}

type Once struct {
    m    sync.Mutex
    done *Singleton
}

func (o *Once) Do(f func()) {
    if o.done != nil {
        return
    }

    o.m.Lock()
    defer o.m.Unlock()
    if o.done == nil {
        f()
        o.done = &Singleton{
            a:      1,
            b:      2,
            c:      3,
            dummy: &dummyObject{4},
        }
    }
}
```

基本同步原语

单例 正确的实现



```go
type Once struct {
    m    sync.Mutex
    done uint32
}

func (o *Once) Do(f func()) {
    if atomic.LoadUint32(&o.done) == 1 {
        return
    }

    o.m.Lock()
    defer o.m.Unlock()
    if o.done == 0 {
        defer atomic.StoreUint32(&o.done, 1)
        f()
    }
}
```

基本同步原语

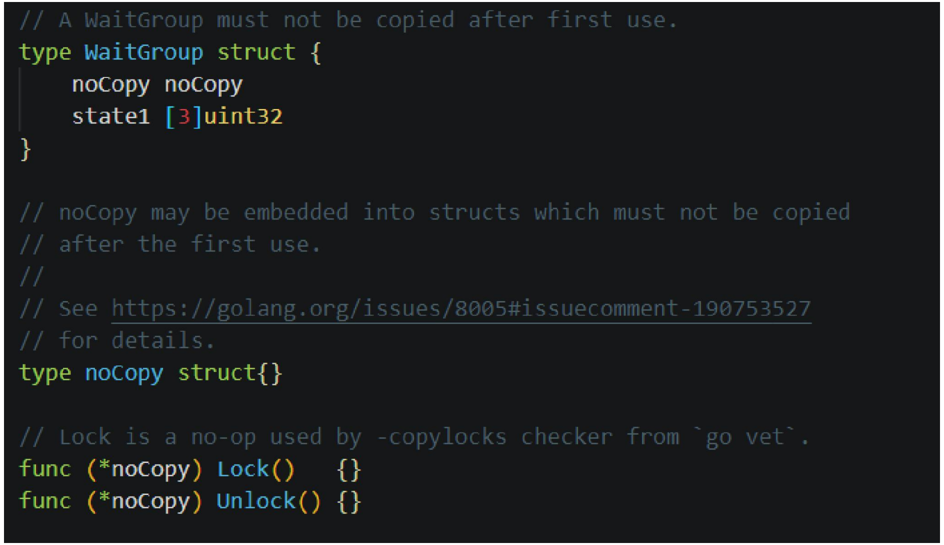
A XXX must not be copied after first use.

- 零值是无锁的
- 使用后是有状态的
- Copy也会copy状态

基本同步原语

A XXX must not be copied after first use.

- go vet可以检查
- 通过嵌入noCopy帮助vet工具检查



```
// A WaitGroup must not be copied after first use.
type WaitGroup struct {
    noCopy noCopy
    state1 [3]uint32
}

// noCopy may be embedded into structs which must not be copied
// after the first use.
//
// See https://golang.org/issues/8005#issuecomment-190753527
// for details.
type noCopy struct{}

// Lock is a no-op used by -copylocks checker from `go vet`.
func (*noCopy) Lock()   {}
func (*noCopy) Unlock() {}
```

基本同步原语

Copy总在不经意间发生

- 嵌套的struct包含非指针类型的同步原语
- 变量赋值
- 函数/方法传参
- 函数/方法的返回值
- 方法的Receiver
- range语句
- ……

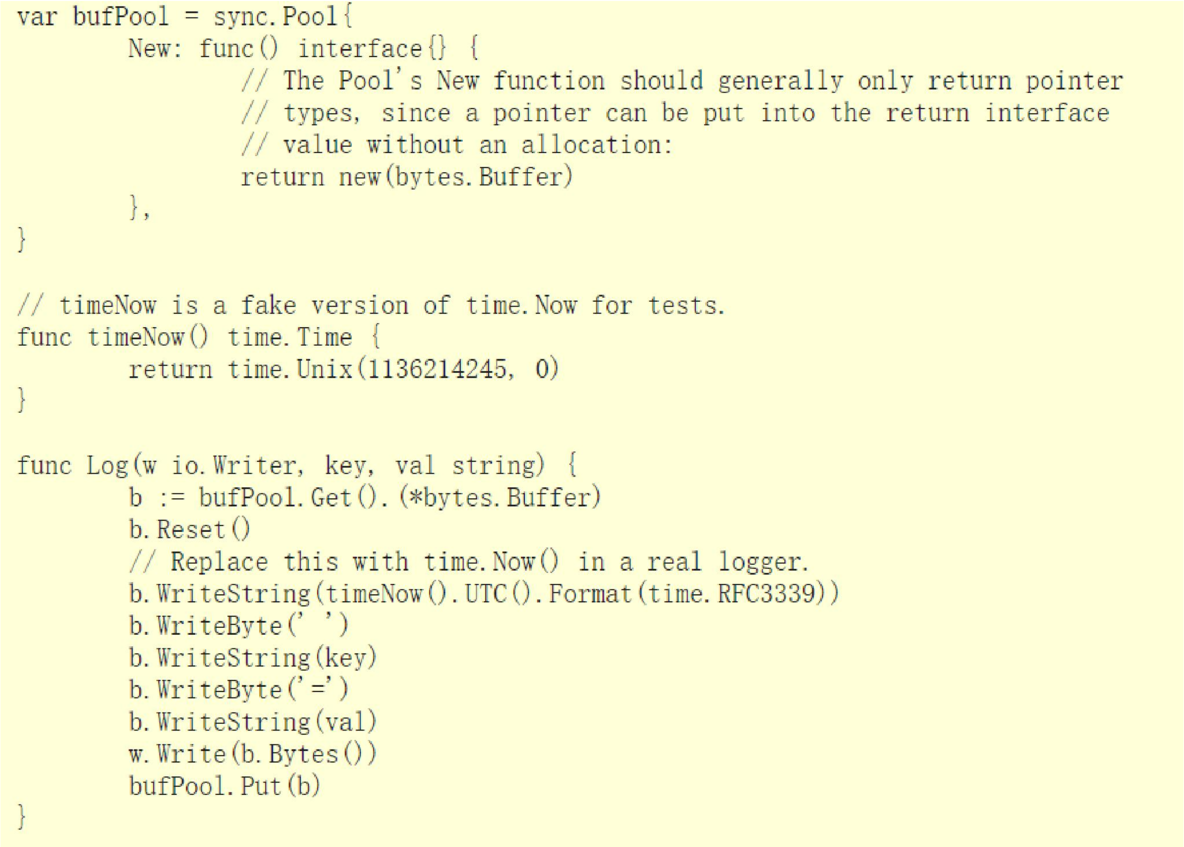
基本同步原语

Pool

- 临时对象池
- 可能在任何时候任意的对象都可能被移除
- 可以安全地并发访问
- 装箱/拆箱

基本同步原语

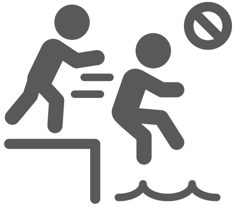
Pool　　容易内存泄漏 go#23199

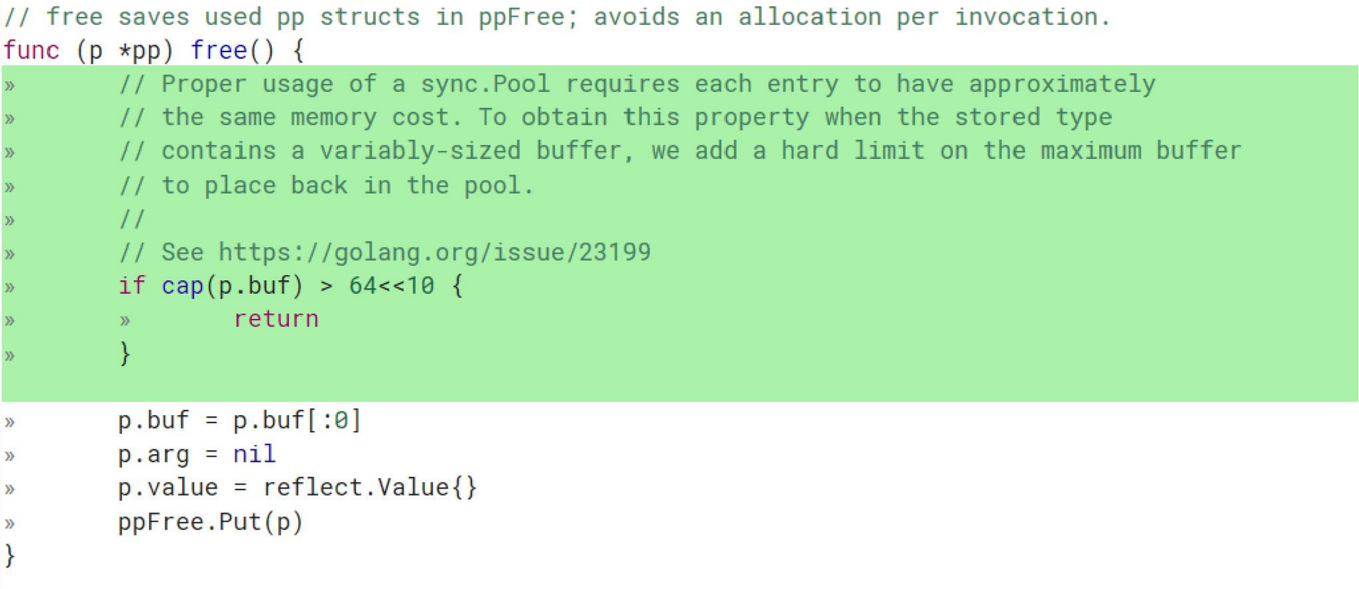


```
var bufPool = sync.Pool{
        New: func() interface{} {
                // The Pool's New function should generally only return pointer
                // types, since a pointer can be put into the return interface
                // value without an allocation:
                return new(bytes.Buffer)
        },
}

// timeNow is a fake version of time.Now for tests.
func timeNow() time.Time {
        return time.Unix(1136214245, 0)
}

func Log(w io.Writer, key, val string) {
        b := bufPool.Get().(*bytes.Buffer)
        b.Reset()
        // Replace this with time.Now() in a real logger.
        b.WriteString(timeNow().UTC().Format(time.RFC3339))
        b.WriteByte(' ')
        b.WriteString(key)
        b.WriteByte('=')
        b.WriteString(val)
        w.Write(b.Bytes())
        bufPool.Put(b)
}
```

基本同步原语

Pool　　fmt包错误使用 go#27740

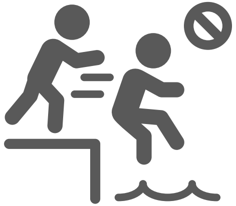


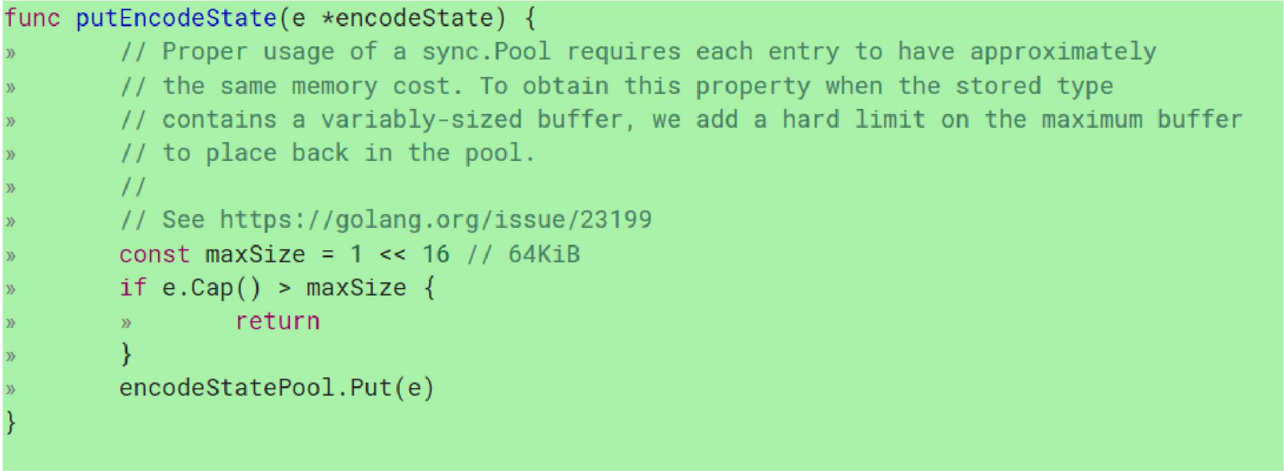


```
// free saves used pp structs in ppFree; avoids an allocation per invocation.
func (p *pp) free() {
»       // Proper usage of a sync.Pool requires each entry to have approximately
»       // the same memory cost. To obtain this property when the stored type
»       // contains a variably-sized buffer, we add a hard limit on the maximum buffer
»       // to place back in the pool.
»       //
»       // See https://golang.org/issue/23199
»       if cap(p.buf) > 64<<10 {
»       »       return
»       }

»       p.buf = p.buf[:0]
»       p.arg = nil
»       p.value = reflect.Value{}
»       ppFree.Put(p)
}
```

基本同步原语

Pool　　json包错误使用 go#2773

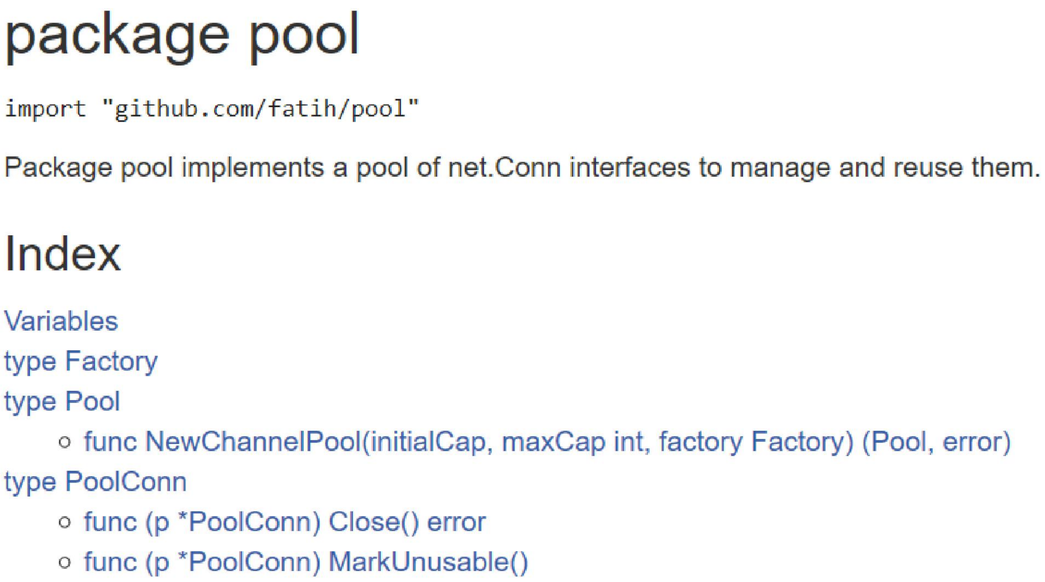




```
func putEncodeState(e *encodeState) {
»       // Proper usage of a sync.Pool requires each entry to have approximately
»       // the same memory cost. To obtain this property when the stored type
»       // contains a variably-sized buffer, we add a hard limit on the maximum buffer
»       // to place back in the pool.
»       //
»       // See https://golang.org/issue/23199
»       const maxSize = 1 << 16 // 64KiB
»       if e.Cap() > maxSize {
»       »       return
»       }
»       encodeStatePool.Put(e)
}
```

基本同步原语

Pool　　Socket连接池



package pool

```
import "github.com/fatih/pool"
```

Package pool implements a pool of net.Conn interfaces to manage and reuse them.

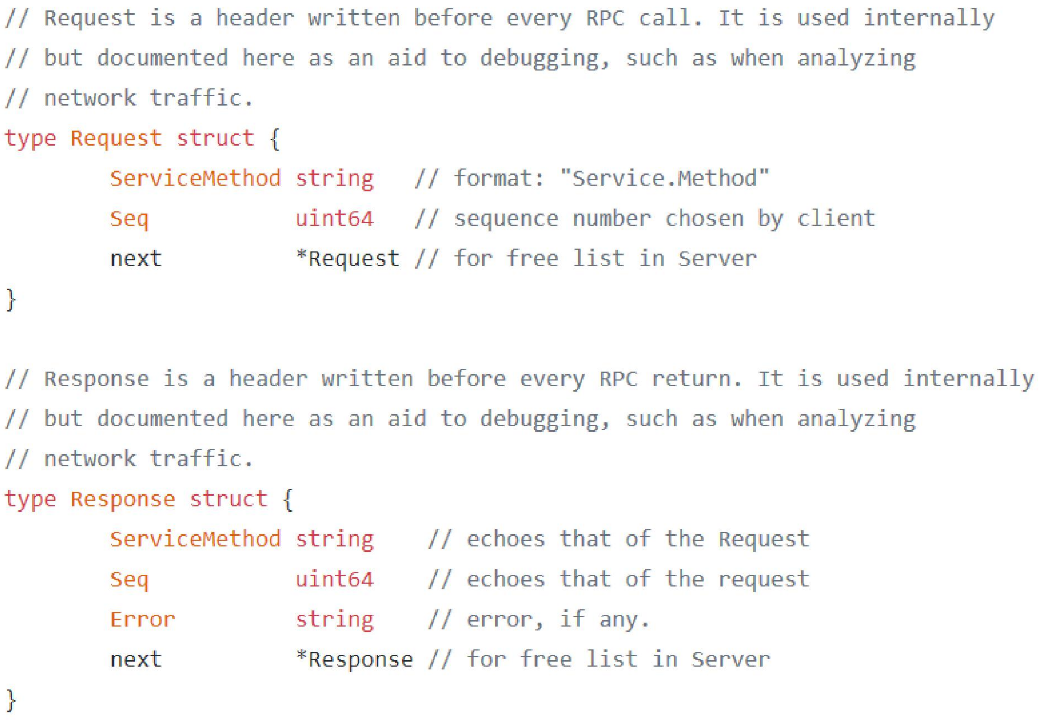
Index

Variables
type Factory
type Pool
- func NewChannelPool(initialCap, maxCap int, factory Factory) (Pool, error)

type PoolConn
- func (p *PoolConn) Close() error
- func (p *PoolConn) MarkUnusable()

基本同步原语

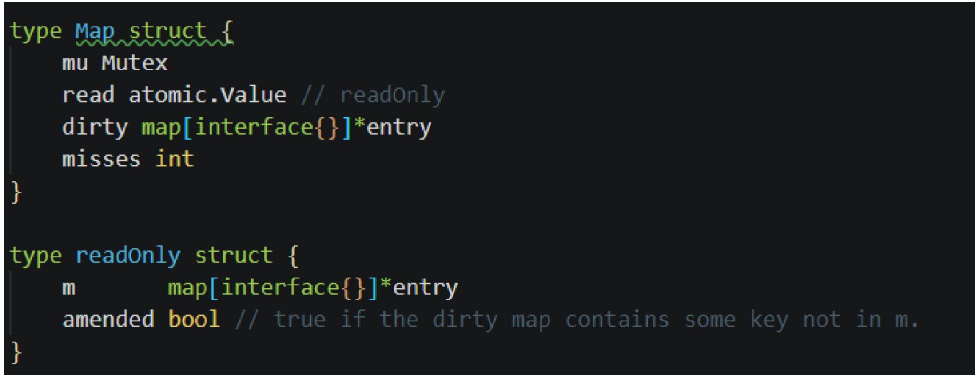
Pool　　使用链表进行重用



```
// Request is a header written before every RPC call. It is used internally
// but documented here as an aid to debugging, such as when analyzing
// network traffic.
type Request struct {
	ServiceMethod string   // format: "Service.Method"
	Seq           uint64   // sequence number chosen by client
	next          *Request // for free list in Server
}

// Response is a header written before every RPC return. It is used internally
// but documented here as an aid to debugging, such as when analyzing
// network traffic.
type Response struct {
	ServiceMethod string    // echoes that of the Request
	Seq           uint64    // echoes that of the request
	Error         string    // error, if any.
	next          *Response // for free list in Server
}
```

基本同步原语

sync.Map



```
type Map struct {
    mu Mutex
    read atomic.Value // readOnly
    dirty map[interface{}]*entry
    misses int
}

type readOnly struct {
    m       map[interface{}]*entry
    amended bool // true if the dirty map contains some key not in m.
}
```

基本同步原语

sync.Map

- 两个场景
  - 设置一次，多次读(比如cache)
  - 多个goroutine并发的读、写、更新不同的key

- 装箱/拆箱
- Range进行遍历,可能会加锁
- 没有Len方法，并且也不会添加

基本同步原语

context.Context

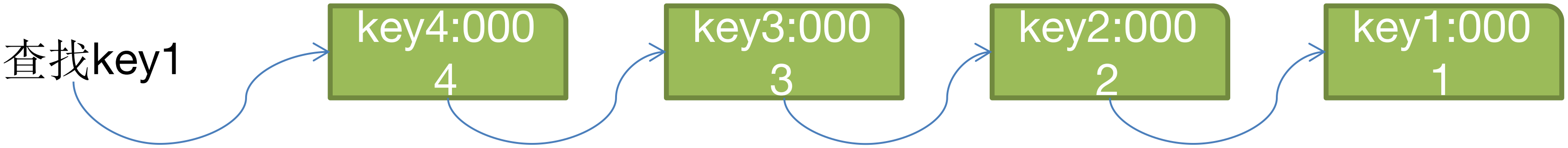
- 传递上下文(本来含义)
- 取消goroutine的运行(扩展功能。主动取消和超时取消)

- 一般用在函数的第一个参数
- 不要嵌入到struct中

基本同步原语

context.Context 链式查找 (WithValue)

```
ctx := context.Background()
ctx = context.TODO()
ctx = context.WithValue(ctx, "key1", "0001")
ctx = context.WithValue(ctx, "key2", "0001")
ctx = context.WithValue(ctx, "key3", "0001")
ctx = context.WithValue(ctx, "key4", "0004")

fmt.Println(ctx.Value("key1"))
```



查找key1

key4:0004

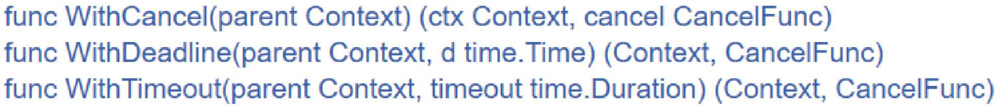
key3:0003

key2:0002

key1:0001

基本同步原语

context.Context



```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, d time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

- 控制程序的运行
- 都返回子context和cancelFunc
- cancelFunc只需调用一次，后续的调用不会做额外的工作
- cancelFunc被调用，或者Parent的Done被close，这个子context的Done也会被close, Err值也会被设置
- 尽早的调用cancelFunc释放资源
- WithDeadline/withTimeout也会和parent的时间进行比较，使用最早的deadline

基本同步原语

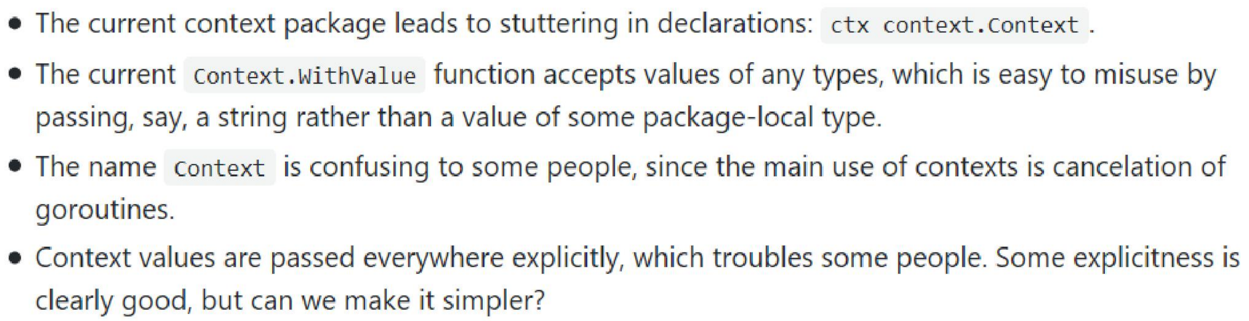
context.Context pros

- 方便传递上下文(request-scoped)
- 可以控制子goroutine的运行(channel的Done模式)
- 无限级的函数传递

基本同步原语

context.Context cons

- Context isn’t for cancellation
- Context should go away for Go 2
- 函数污染，想象一下Reader/Writer等所有的函数都不得不增加context作为第一个参数

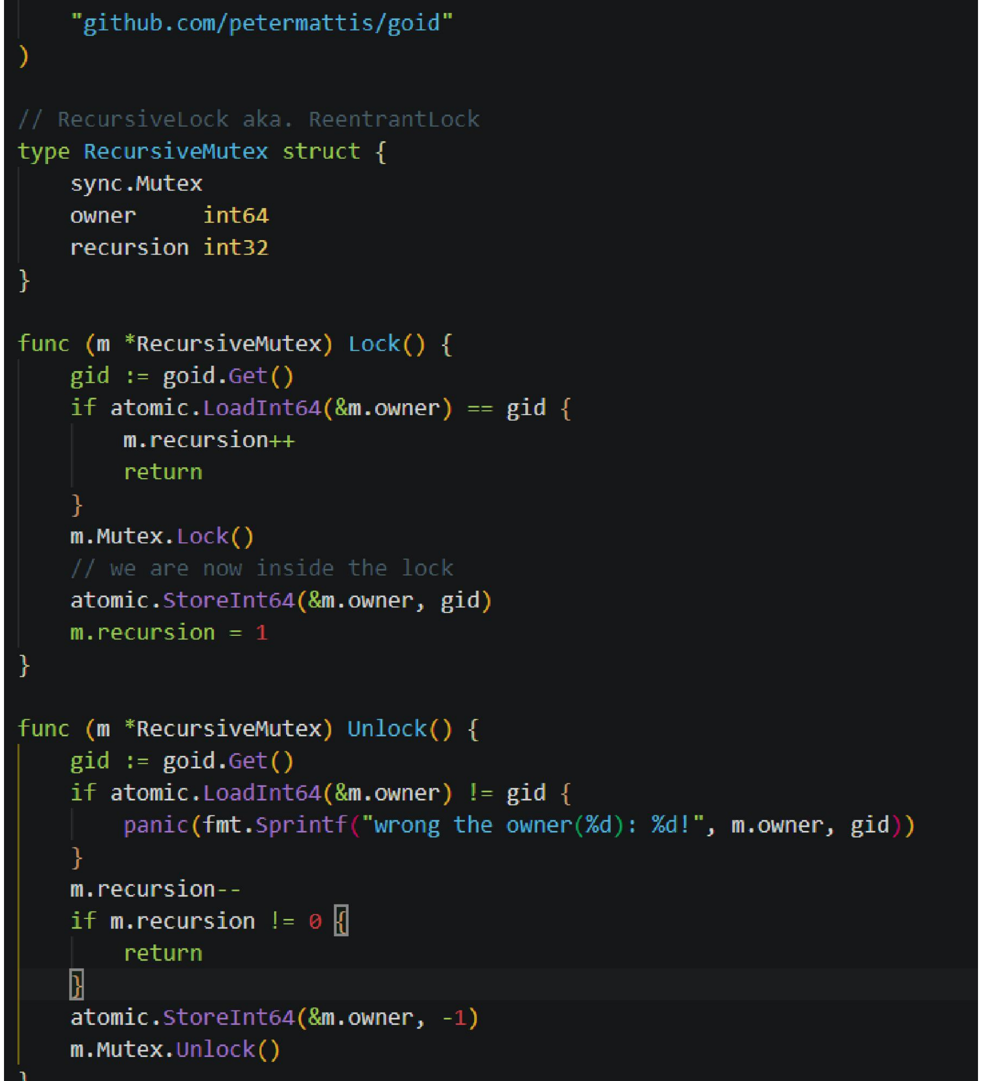


- The current context package leads to stuttering in declarations: `ctx context.Context`.
- The current `Context.WithValue` function accepts values of any types, which is easy to misuse by passing, say, a string rather than a value of some package-local type.
- The name `Context` is confusing to some people, since the main use of contexts is cancelation of goroutines.
- Context values are passed everywhere explicitly, which troubles some people. Some explicitness is clearly good, but can we make it simpler?

扩展同步原语

基本同步原语

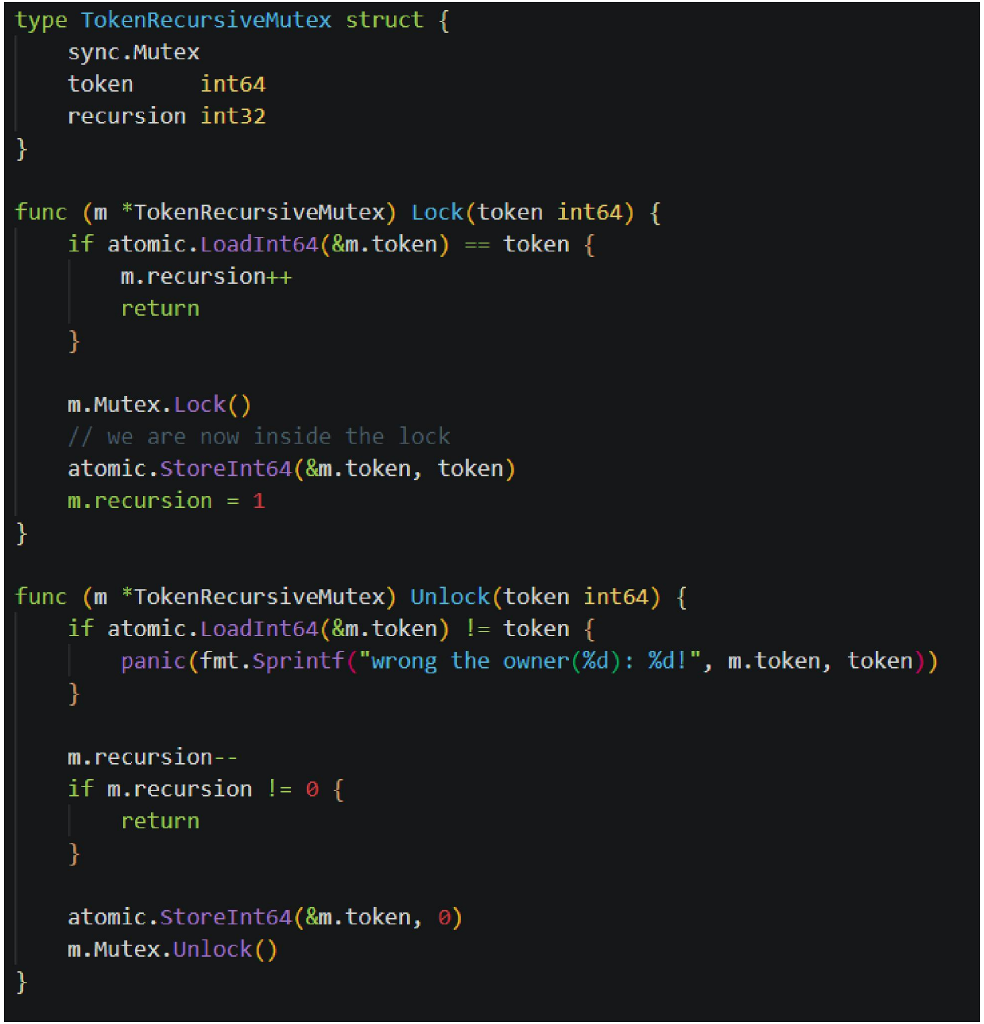
ReentrantLock goid



```go
    "github.com/petermattis/goid"
)

// RecursiveLock aka. ReentrantLock
type RecursiveMutex struct {
    sync.Mutex
    owner     int64
    recursion int32
}

func (m *RecursiveMutex) Lock() {
    gid := goid.Get()
    if atomic.LoadInt64(&m.owner) == gid {
        m.recursion++
        return
    }
    m.Mutex.Lock()
    // we are now inside the lock
    atomic.StoreInt64(&m.owner, gid)
    m.recursion = 1
}

func (m *RecursiveMutex) Unlock() {
    gid := goid.Get()
    if atomic.LoadInt64(&m.owner) != gid {
        panic(fmt.Sprintf("wrong the owner(%d): %d!", m.owner, gid))
    }
    m.recursion--
    if m.recursion != 0 {
        return
    }
    atomic.StoreInt64(&m.owner, -1)
    m.Mutex.Unlock()
}
```

基本同步原语

ReentrantLock token



```go
type TokenRecursiveMutex struct {
    sync.Mutex
    token     int64
    recursion int32
}

func (m *TokenRecursiveMutex) Lock(token int64) {
    if atomic.LoadInt64(&m.token) == token {
        m.recursion++
        return
    }

    m.Mutex.Lock()
    // we are now inside the lock
    atomic.StoreInt64(&m.token, token)
    m.recursion = 1
}

func (m *TokenRecursiveMutex) Unlock(token int64) {
    if atomic.LoadInt64(&m.token) != token {
        panic(fmt.Sprintf("wrong the owner(%d): %d!", m.token, token))
    }

    m.recursion--
    if m.recursion != 0 {
        return
    }

    atomic.StoreInt64(&m.token, 0)
    m.Mutex.Unlock()
}
```

扩展同步原语

Semaphore

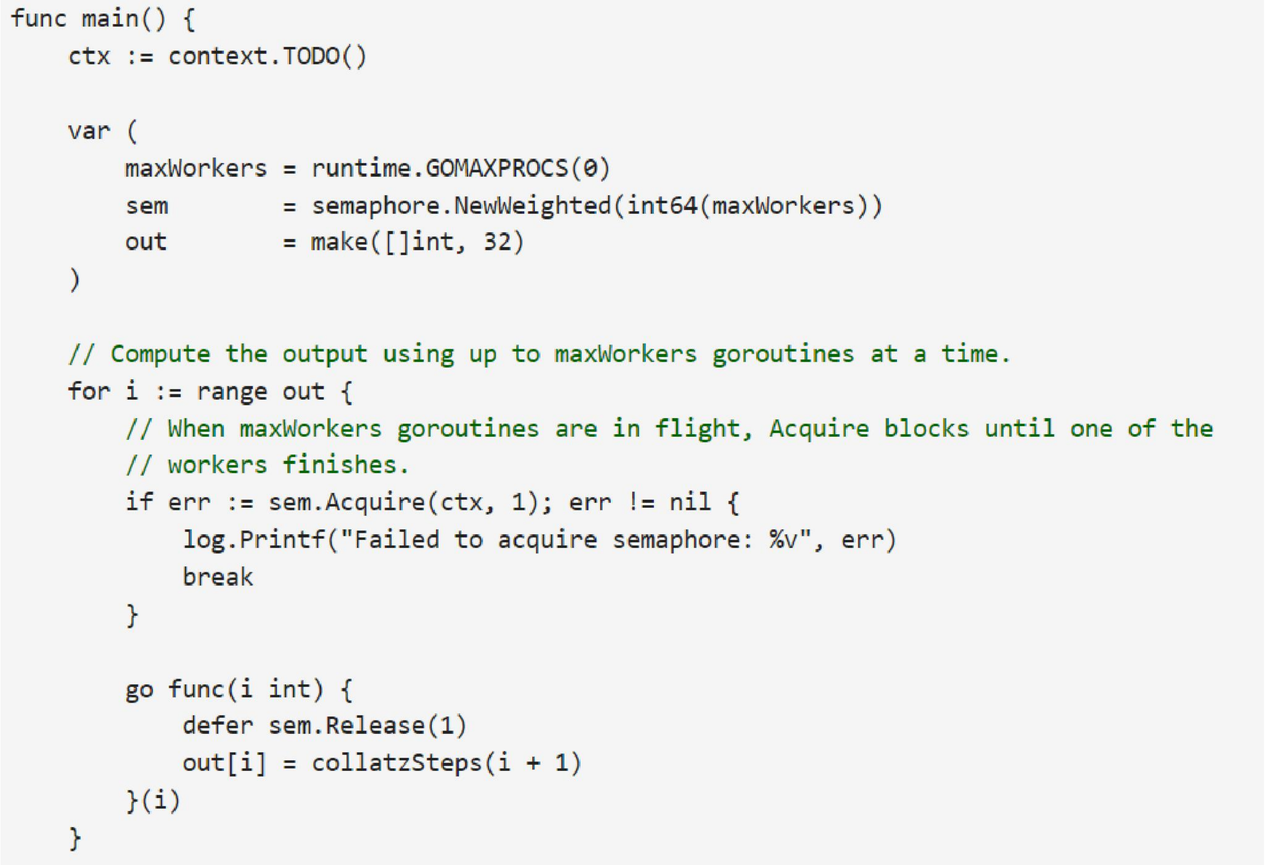
- Dijkstra提出并发访问通用资源的同步原语
- 初始化一个非负的值S
- P(wait) 减一，如果S小于0，阻塞本goroutine进入临界区
- V(signal)加一，如果S不为负值，其它goroutine可以进入临界区

- 二进制信号量可以实现锁(0,1)
- 计数信号量

扩展同步原语

Semaphore

golang.org/x/sync/semaphore

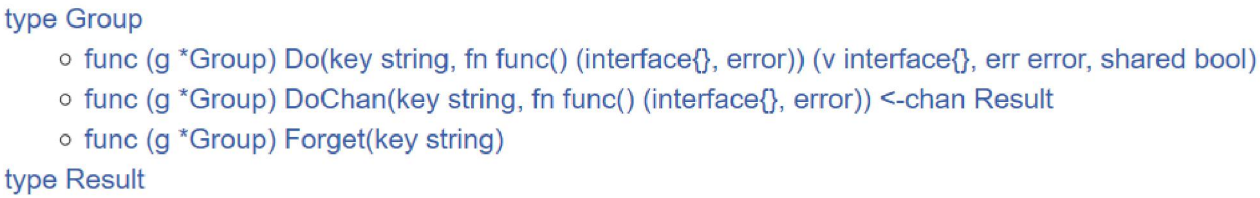


```go
func main() {
    ctx := context.TODO()

    var (
        maxWorkers = runtime.GOMAXPROCS(0)
        sem        = semaphore.NewWeighted(int64(maxWorkers))
        out        = make([]int, 32)
    )

    // Compute the output using up to maxWorkers goroutines at a time.
    for i := range out {
        // When maxWorkers goroutines are in flight, Acquire blocks until one of the
        // workers finishes.
        if err := sem.Acquire(ctx, 1); err != nil {
            log.Printf("Failed to acquire semaphore: %v", err)
            break
        }

        go func(i int) {
            defer sem.Release(1)
            out[i] = collatzSteps(i + 1)
        }(i)
    }
```

扩展同步原语

SingleFlight

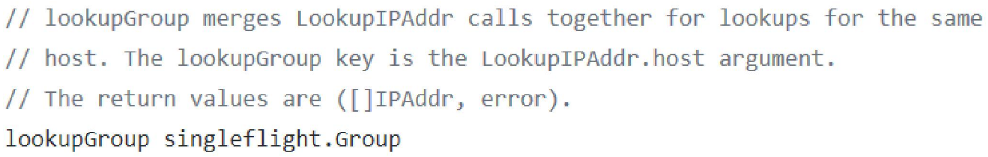
golang.org/x/sync/singleflight
go/src/internal/singleflight/singleflight.go



type Group

- func (g *Group) Do(key string, fn func() (interface{}, error)) (v interface{}, err error, shared bool)
- func (g *Group) DoChan(key string, fn func() (interface{}, error)) <-chan Result
- func (g *Group) Forget(key string)

type Result

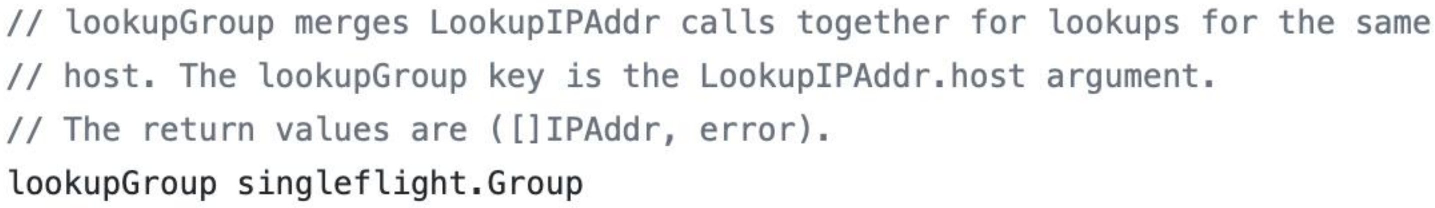


```
// lookupGroup merges LookupIPAddr calls together for lookups for the same
// host. The lookupGroup key is the LookupIPAddr.host argument.
// The return values are ([]IPAddr, error).
lookupGroup singleflight.Group
```

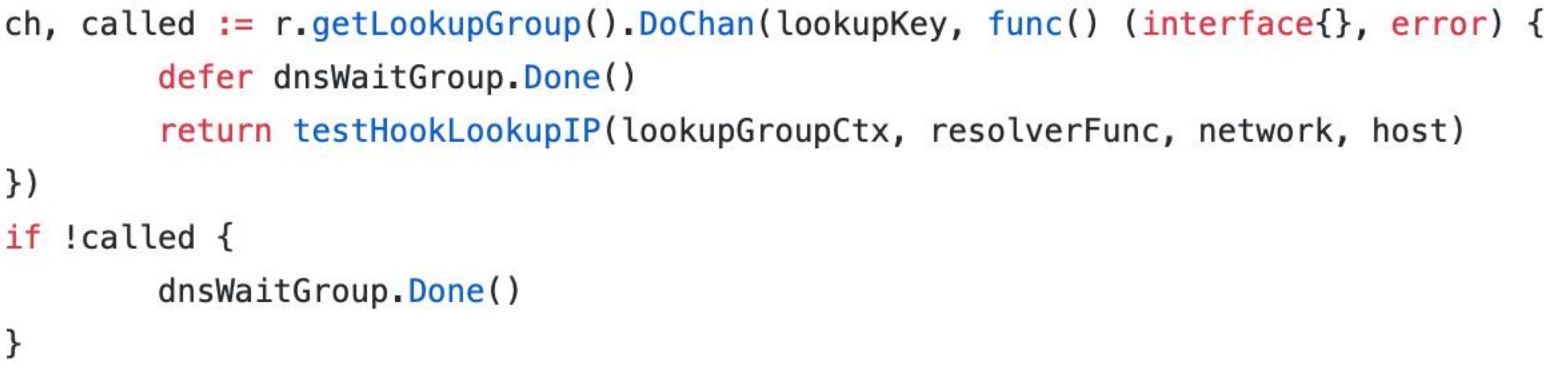
扩展同步原语

SingleFlight 应用

- 标准库



```
// lookupGroup merges LookupIPAddr calls together for lookups for the same
// host. The lookupGroup key is the LookupIPAddr.host argument.
// The return values are ([]IPAddr, error).
lookupGroup singleflight.Group
```

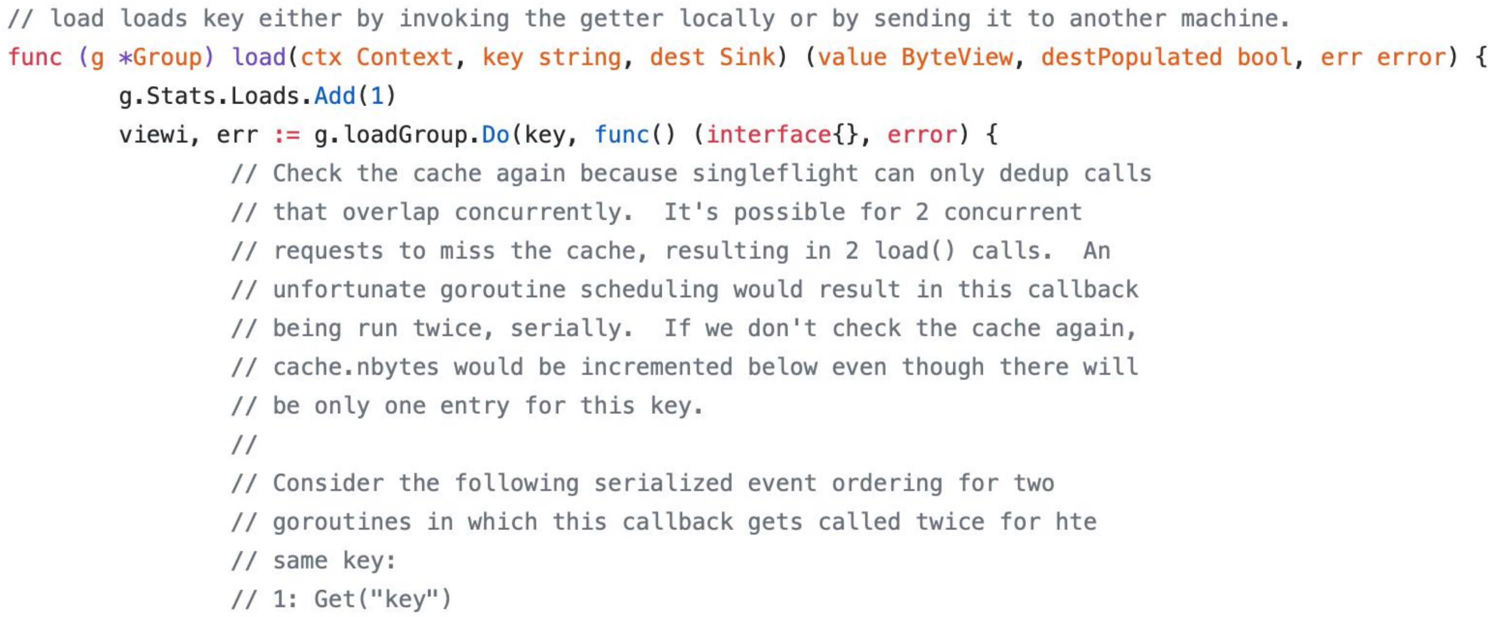


```
ch, called := r.getLookupGroup().DoChan(lookupKey, func() (interface{}, error) {
        defer dnsWaitGroup.Done()
        return testHookLookupIP(lookupGroupCtx, resolverFunc, network, host)
})
if !called {
        dnsWaitGroup.Done()
}
```

扩展同步原语

SingleFlight 应用

- groupcache

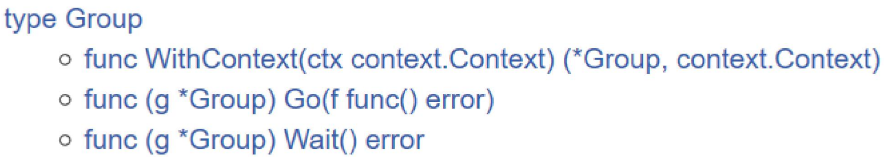


```go
// load loads key either by invoking the getter locally or by sending it to another machine.
func (g *Group) load(ctx Context, key string, dest Sink) (value ByteView, destPopulated bool, err error) {
	g.Stats.Loads.Add(1)
	viewi, err := g.loadGroup.Do(key, func() (interface{}, error) {
		// Check the cache again because singleflight can only dedup calls
		// that overlap concurrently.  It's possible for 2 concurrent
		// requests to miss the cache, resulting in 2 load() calls.  An
		// unfortunate goroutine scheduling would result in this callback
		// being run twice, serially.  If we don't check the cache again,
		// cache.nbytes would be incremented below even though there will
		// be only one entry for this key.
		//
		// Consider the following serialized event ordering for two
		// goroutines in which this callback gets called twice for hte
		// same key:
		// 1: Get("key")
```

扩展同步原语

ErrGroup

golang.org/x/sync/semaphore



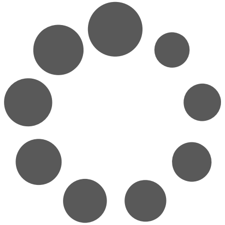
type Group

- func WithContext(ctx context.Context) (*Group, context.Context)
- func (g *Group) Go(f func() error)
- func (g *Group) Wait() error

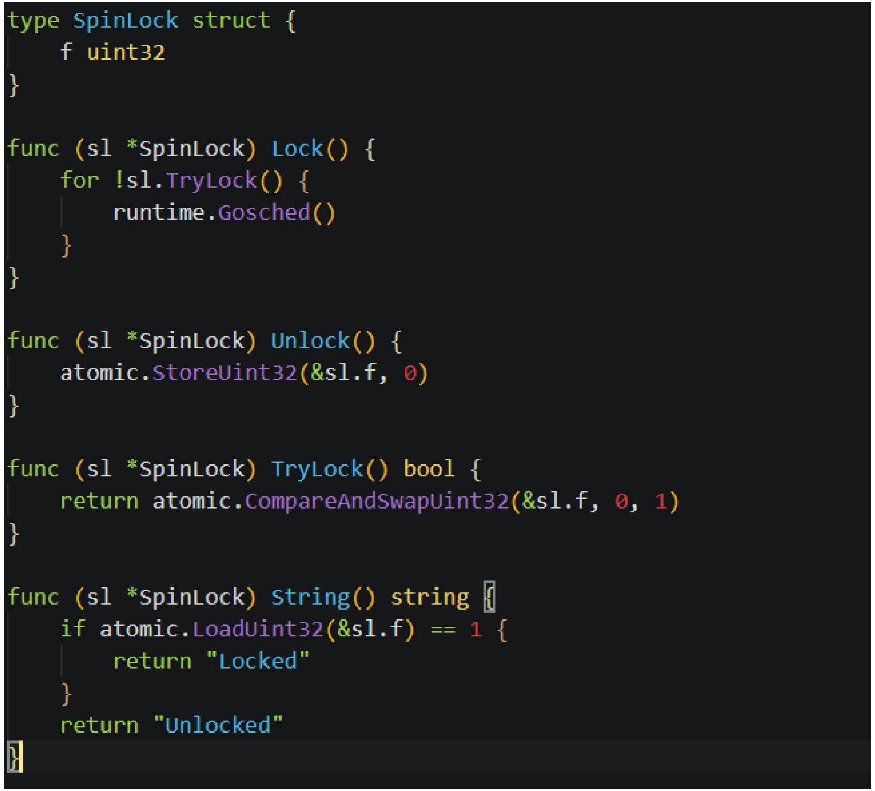
- Wait会等待所有的goroutine执行完后才释放
- 如果想遇到第一个err就返回，使用Context

扩展同步原语

SpinLock



- 自旋锁
- 有些场景很高效，但是
- 非公平
- 处理器忙等待

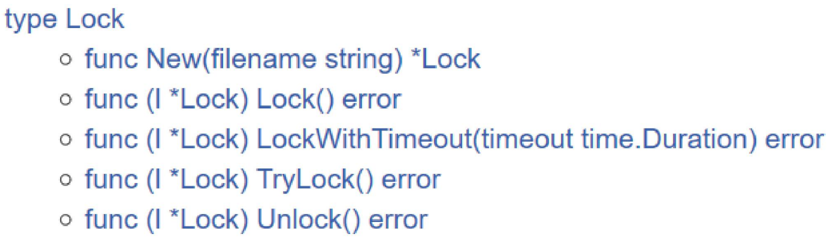


```
type SpinLock struct {
    f uint32
}

func (sl *SpinLock) Lock() {
    for !sl.TryLock() {
        runtime.Gosched()
    }
}

func (sl *SpinLock) Unlock() {
    atomic.StoreUint32(&sl.f, 0)
}

func (sl *SpinLock) TryLock() bool {
    return atomic.CompareAndSwapUint32(&sl.f, 0, 1)
}

func (sl *SpinLock) String() string {
    if atomic.LoadUint32(&sl.f) == 1 {
        return "Locked"
    }
    return "Unlocked"
}
```

扩展同步原语

FileLock

github.com/juju/fslock
跨进程的Mutex



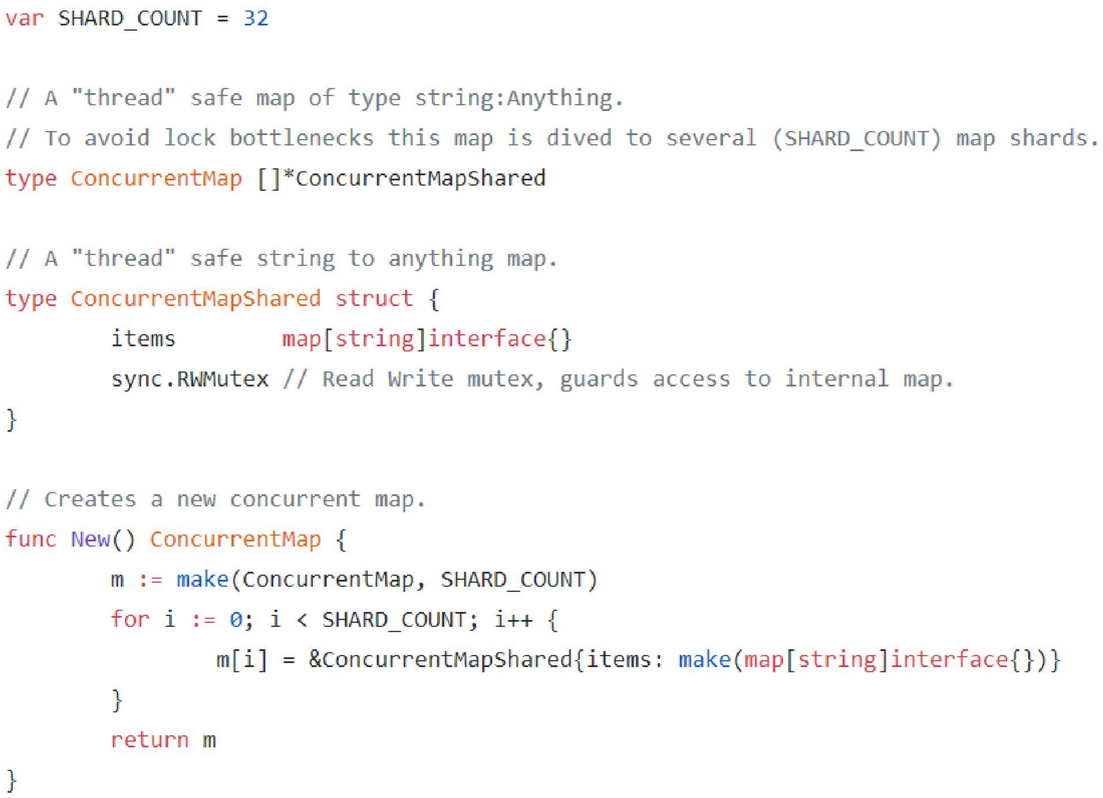
type Lock

- func New(filename string) *Lock
- func (l *Lock) Lock() error
- func (l *Lock) LockWithTimeout(timeout time.Duration) error
- func (l *Lock) TryLock() error
- func (l *Lock) Unlock() error

扩展同步原语

github.com/orcaman/concurrent-map

concurrent-map



```go
var SHARD_COUNT = 32

// A "thread" safe map of type string:Anything.
// To avoid lock bottlenecks this map is dived to several (SHARD_COUNT) map shards.
type ConcurrentMap []*ConcurrentMapShared

// A "thread" safe string to anything map.
type ConcurrentMapShared struct {
        items        map[string]interface{}
        sync.RWMutex // Read Write mutex, guards access to internal map.
}

// Creates a new concurrent map.
func New() ConcurrentMap {
        m := make(ConcurrentMap, SHARD_COUNT)
        for i := 0; i < SHARD_COUNT; i++ {
                m[i] = &ConcurrentMapShared{items: make(map[string]interface{})}
        }
        return m
}
```

原子操作

原子操作

atomic 数据类型

- int32
- int64
- uint32
- uint64
- uintptr
- unsafe.Pointer

原子操作

atomic 函数

- AddXXX (整数类型)
- CompareAndSwapXXX：cas
- LoadXXX：读取
- StoreXXX：存储
- SwapXXX：交换

原子操作

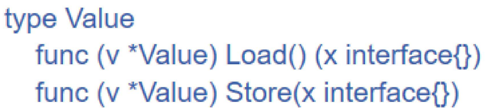
atomic 函数

- 有Add没有Subtract？
  - 有符号的类型，可以使用Add负数
  - 无符号的类型，可以使用AddUint32(&x, ^uint32(c-1)),AddUint64(&x, ^uint64(c-1))
  - 无符号类型减一， AddUint32(&x, ^uint32(0))， AddUint64(&x, ^uint64(0))

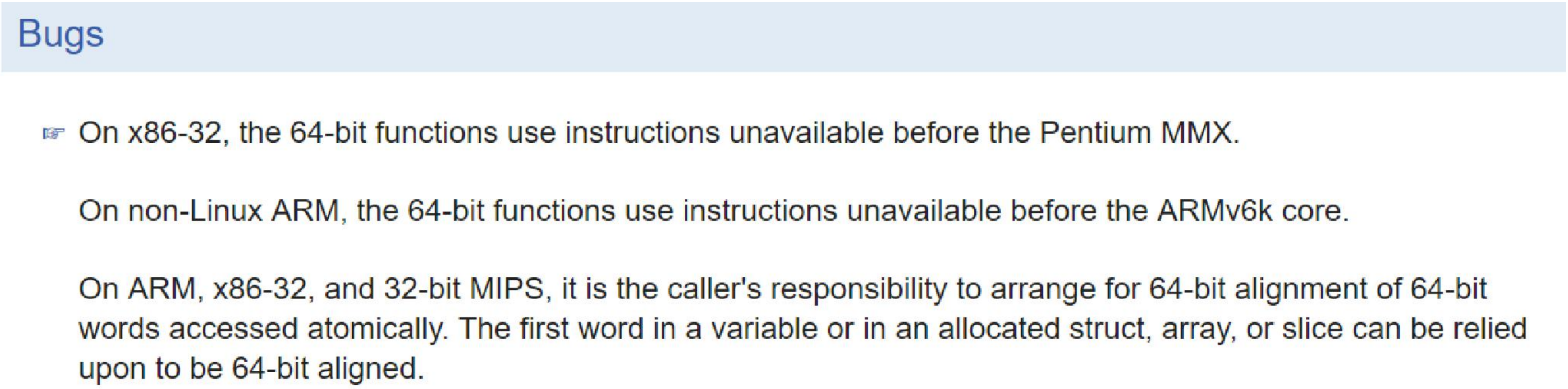
原子操作

atomic 通用对象

- Value



```
type Value
    func (v *Value) Load() (x interface{})
    func (v *Value) Store(x interface{})
```



Bugs

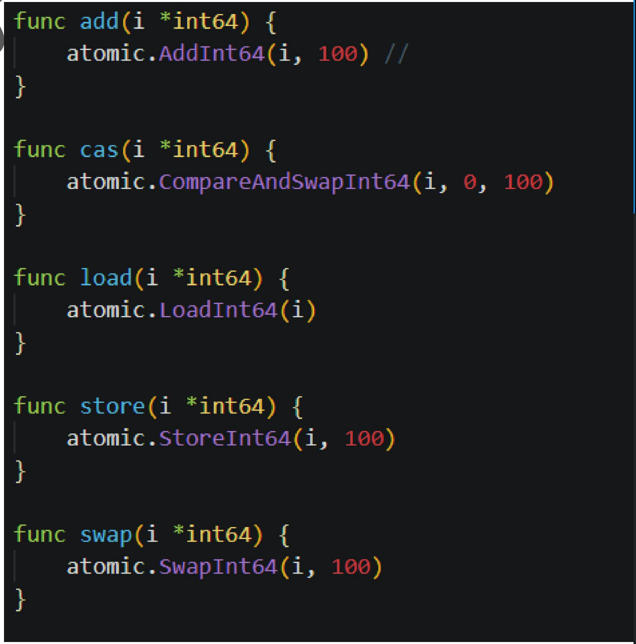
☞ On x86-32, the 64-bit functions use instructions unavailable before the Pentium MMX.

On non-Linux ARM, the 64-bit functions use instructions unavailable before the ARMv6k core.

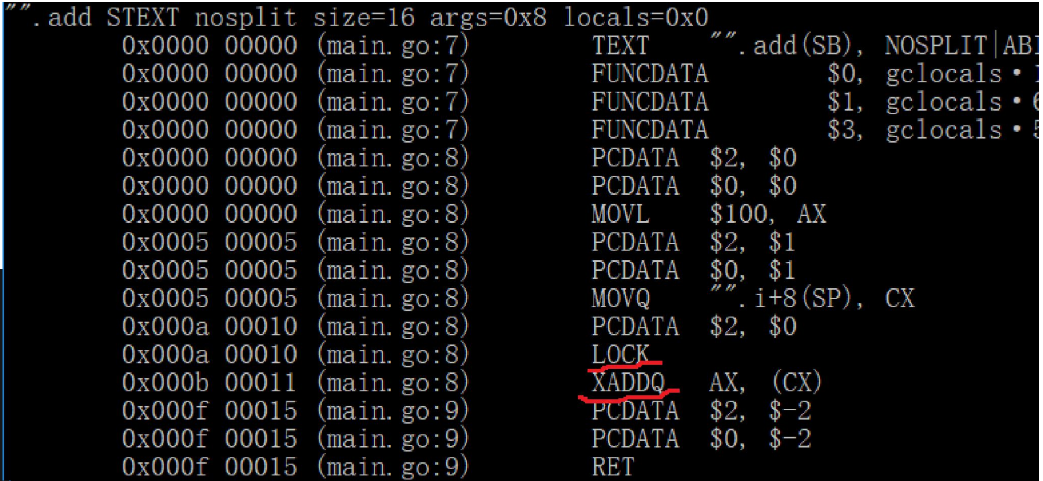
On ARM, x86-32, and 32-bit MIPS, it is the caller's responsibility to arrange for 64-bit alignment of 64-bit words accessed atomically. The first word in a variable or in an allocated struct, array, or slice can be relied upon to be 64-bit aligned.

原子操作

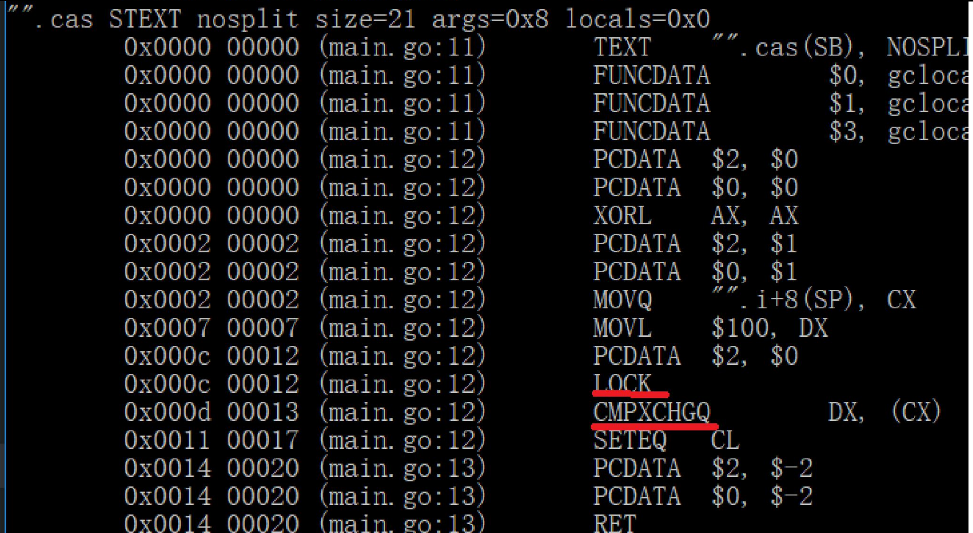
atomic 实现 AMD64 (windows)



```
func add(i *int64) {
    atomic.AddInt64(i, 100) //
}

func cas(i *int64) {
    atomic.CompareAndSwapInt64(i, 0, 100)
}

func load(i *int64) {
    atomic.LoadInt64(i)
}

func store(i *int64) {
    atomic.StoreInt64(i, 100)
}

func swap(i *int64) {
    atomic.SwapInt64(i, 100)
}
```



```
"".add STEXT nosplit size=16 args=0x8 locals=0x0
        0x0000 00000 (main.go:7)      TEXT    "".add(SB), NOSPLIT|ABI
        0x0000 00000 (main.go:7)      FUNCDATA        $0, gclocals·
        0x0000 00000 (main.go:7)      FUNCDATA        $1, gclocals·
        0x0000 00000 (main.go:7)      FUNCDATA        $3, gclocals·
        0x0000 00000 (main.go:8)      PCDATA  $2, $0
        0x0000 00000 (main.go:8)      PCDATA  $0, $0
        0x0000 00000 (main.go:8)      MOVL    $100, AX
        0x0005 00005 (main.go:8)      PCDATA  $2, $1
        0x0005 00005 (main.go:8)      PCDATA  $0, $1
        0x0005 00005 (main.go:8)      MOVQ    "".i+8(SP), CX
        0x000a 00010 (main.go:8)      PCDATA  $2, $0
        0x000a 00010 (main.go:8)      LOCK
        0x000b 00011 (main.go:8)      XADDQ   AX, (CX)
        0x000f 00015 (main.go:9)      PCDATA  $2, $-2
        0x000f 00015 (main.go:9)      PCDATA  $0, $-2
        0x000f 00015 (main.go:9)      RET
```

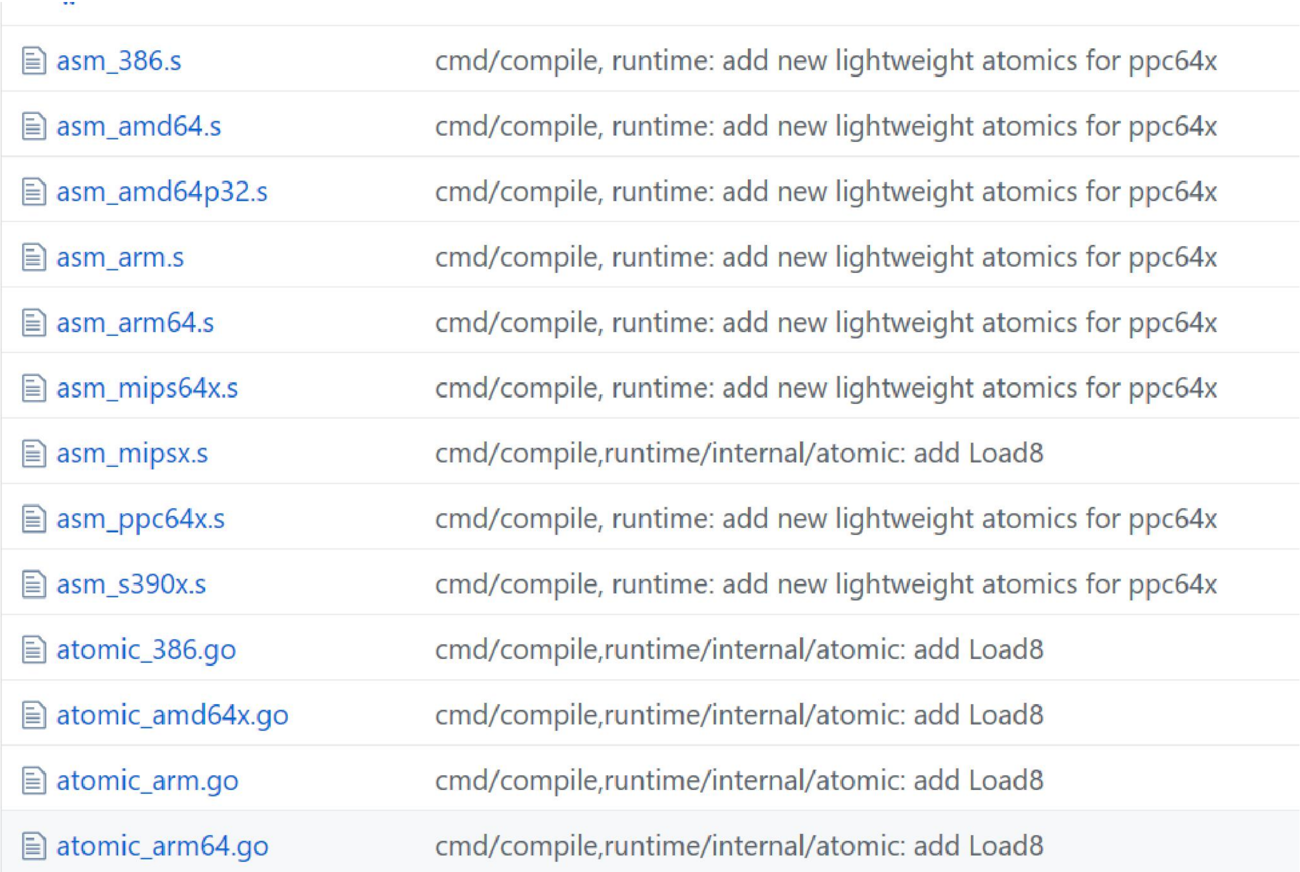


```
"".cas STEXT nosplit size=21 args=0x8 locals=0x0
        0x0000 00000 (main.go:11)     TEXT    "".cas(SB), NOSPLI
        0x0000 00000 (main.go:11)     FUNCDATA        $0, gcloca
        0x0000 00000 (main.go:11)     FUNCDATA        $1, gcloca
        0x0000 00000 (main.go:11)     FUNCDATA        $3, gcloca
        0x0000 00000 (main.go:12)     PCDATA  $2, $0
        0x0000 00000 (main.go:12)     PCDATA  $0, $0
        0x0000 00000 (main.go:12)     XORL    AX, AX
        0x0002 00002 (main.go:12)     PCDATA  $2, $1
        0x0002 00002 (main.go:12)     PCDATA  $0, $1
        0x0002 00002 (main.go:12)     MOVQ    "".i+8(SP), CX
        0x0007 00007 (main.go:12)     MOVL    $100, DX
        0x000c 00012 (main.go:12)     PCDATA  $2, $0
        0x000c 00012 (main.go:12)     LOCK
        0x000d 00013 (main.go:12)     CMPXCHGQ        DX, (CX)
        0x0011 00017 (main.go:12)     SETEQ   CL
        0x0014 00020 (main.go:13)     PCDATA  $2, $-2
        0x0014 00020 (main.go:13)     PCDATA  $0, $-2
        0x0014 00020 (main.go:13)     RET
```

原子操作

atomic 实现 386及其它

/runtime/internal/atomic

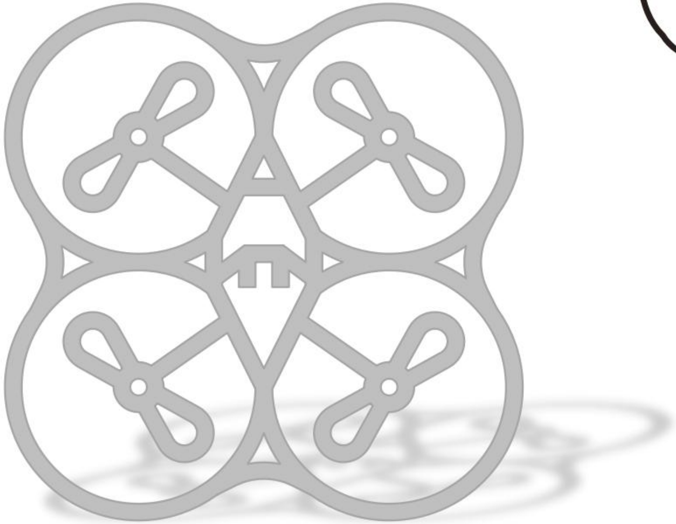


asm_386.s	cmd/compile, runtime: add new lightweight atomics for ppc64x
asm_amd64.s	cmd/compile, runtime: add new lightweight atomics for ppc64x
asm_amd64p32.s	cmd/compile, runtime: add new lightweight atomics for ppc64x
asm_arm.s	cmd/compile, runtime: add new lightweight atomics for ppc64x
asm_arm64.s	cmd/compile, runtime: add new lightweight atomics for ppc64x
asm_mips64x.s	cmd/compile, runtime: add new lightweight atomics for ppc64x
asm_mipsx.s	cmd/compile,runtime/internal/atomic: add Load8
asm_ppc64x.s	cmd/compile, runtime: add new lightweight atomics for ppc64x
asm_s390x.s	cmd/compile, runtime: add new lightweight atomics for ppc64x
atomic_386.go	cmd/compile,runtime/internal/atomic: add Load8
atomic_amd64x.go	cmd/compile,runtime/internal/atomic: add Load8
atomic_arm.go	cmd/compile,runtime/internal/atomic: add Load8
atomic_arm64.go	cmd/compile,runtime/internal/atomic: add Load8

原子操作

Lock-free算法

- non-blocking 算法
  - lock-free:保证系统的吞吐率,
  - wait-free：保证线程的吞吐率
- 实现：atomic read-modify-write, 实现基本的数据结构
- 例外：不使用CAS
  - ringbuffer: single reader single writer
  - read-copy-write: single writer(lock-free), n readers (wait-free)
  - read-copy-write: m writer (with lock), n readers (lock-free)



02 分布式同步原语

分布式同步原语

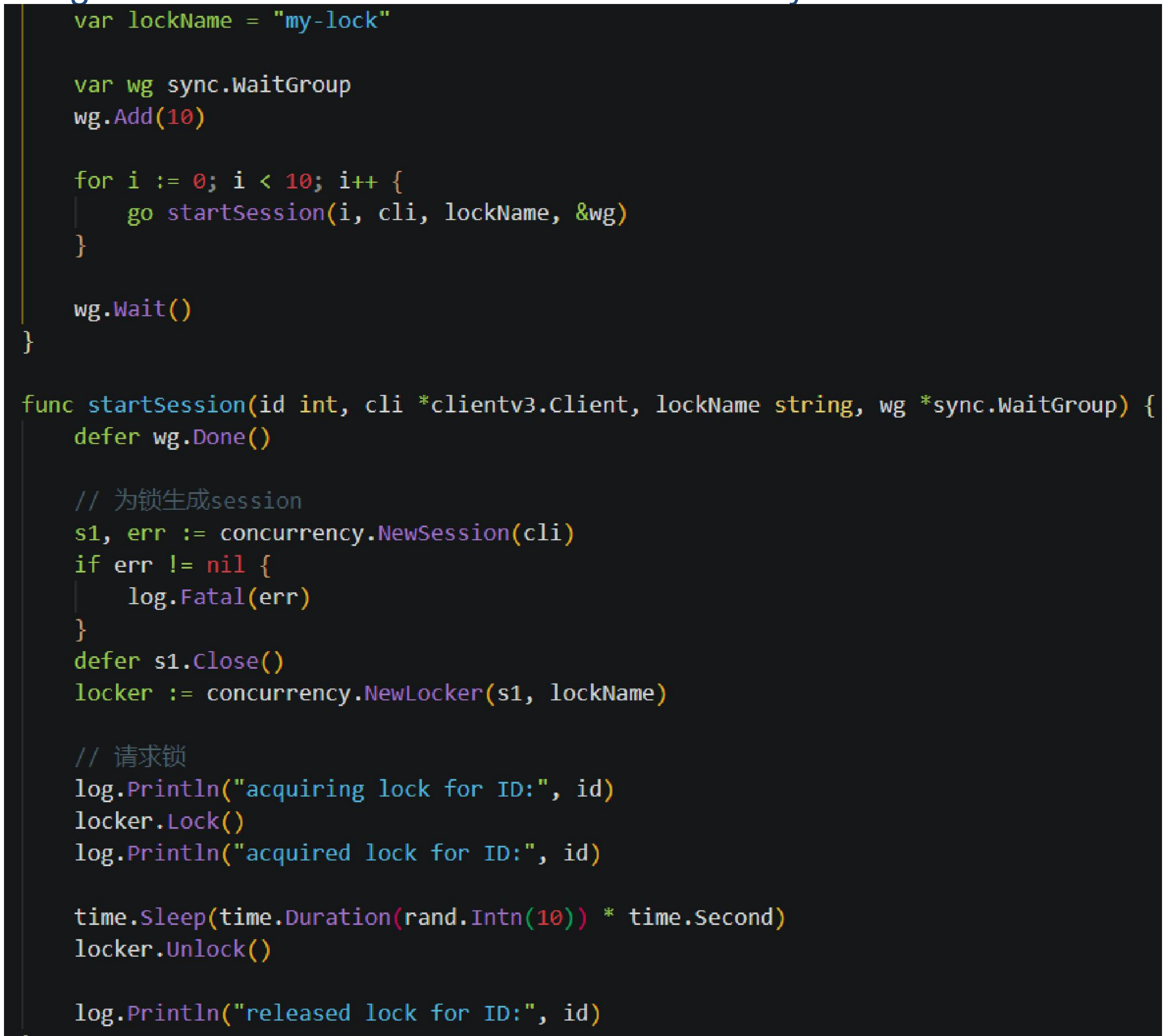
etcd vs zookeepr vs consul

- zookeeper
  - ZooKeeper Recipes and Solutions
  - Apache Curator Recipes
- consul
  - 官方不支持
- etcd
  - contrib/recipes
  - clientv3/concurrency
- redis
  - redlock

分布式同步原语

Locker

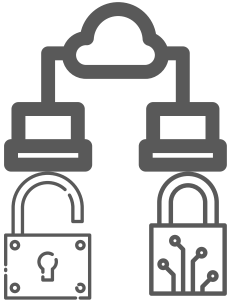
github.com/coreos/etcd/clientv3/concurrency



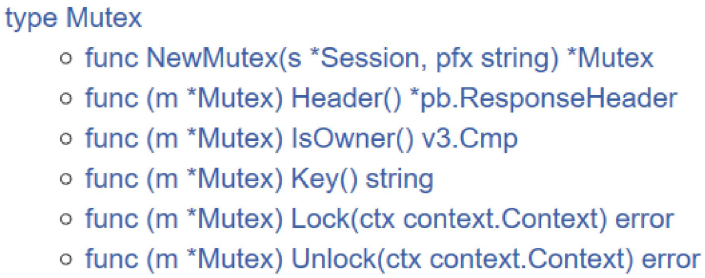
```go
    var lockName = "my-lock"

    var wg sync.WaitGroup
    wg.Add(10)

    for i := 0; i < 10; i++ {
        go startSession(i, cli, lockName, &wg)
    }

    wg.Wait()
}

func startSession(id int, cli *clientv3.Client, lockName string, wg *sync.WaitGroup) {
    defer wg.Done()

    // 为锁生成session
    s1, err := concurrency.NewSession(cli)
    if err != nil {
        log.Fatal(err)
    }
    defer s1.Close()
    locker := concurrency.NewLocker(s1, lockName)

    // 请求锁
    log.Println("acquiring lock for ID:", id)
    locker.Lock()
    log.Println("acquired lock for ID:", id)

    time.Sleep(time.Duration(rand.Intn(10)) * time.Second)
    locker.Unlock()

    log.Println("released lock for ID:", id)
```

分布式同步原语

Mutex



github.com/coreos/etcd/clientv3/concurrency



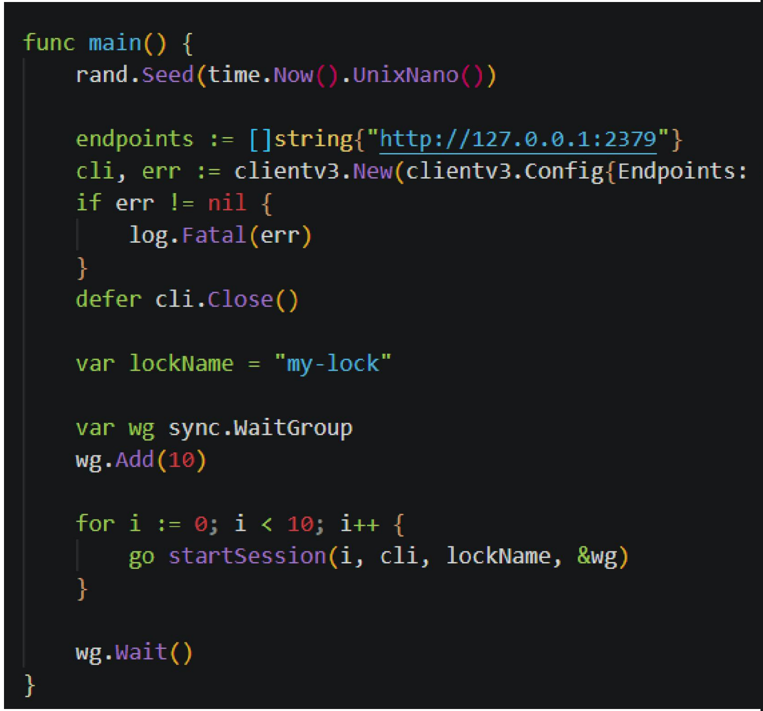
type Mutex

- func NewMutex(s *Session, pfx string) *Mutex
- func (m *Mutex) Header() *pb.ResponseHeader
- func (m *Mutex) IsOwner() v3.Cmp
- func (m *Mutex) Key() string
- func (m *Mutex) Lock(ctx context.Context) error
- func (m *Mutex) Unlock(ctx context.Context) error

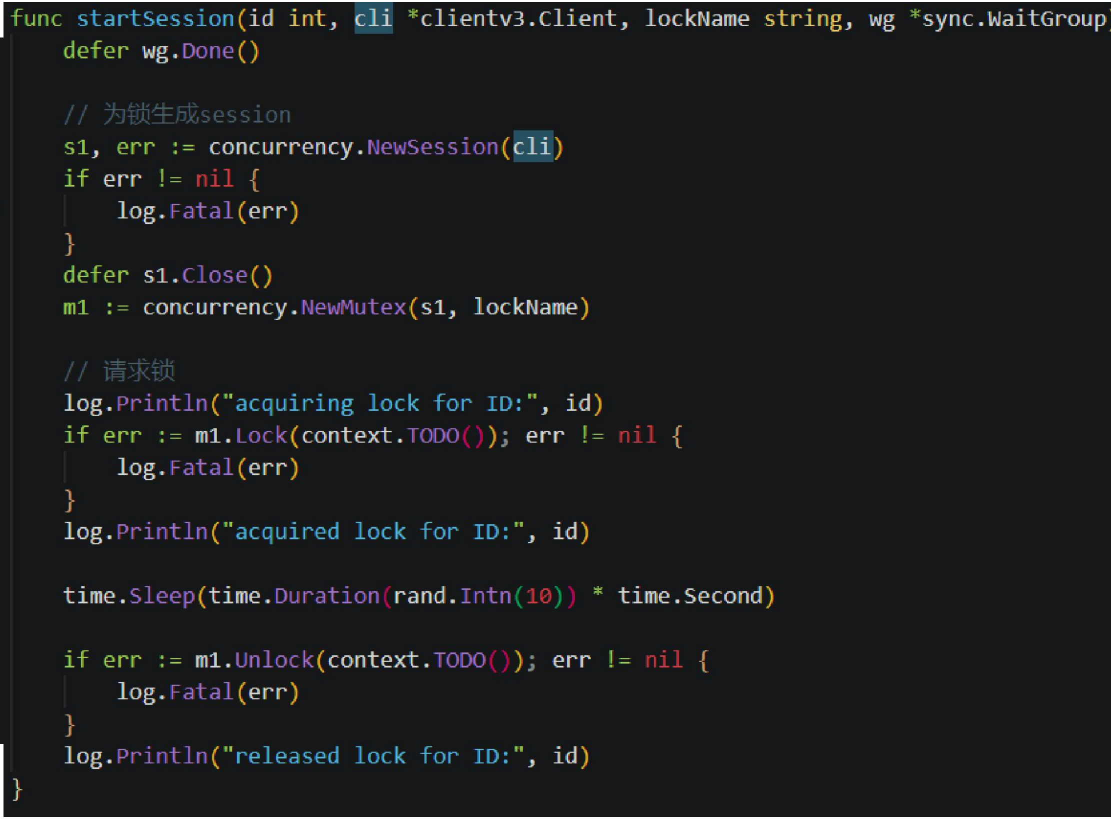
分布式同步原语

github.com/coreos/etcd/clientv3/concurrency

Mutex



```go
func main() {
    rand.Seed(time.Now().UnixNano())

    endpoints := []string{"http://127.0.0.1:2379"}
    cli, err := clientv3.New(clientv3.Config{Endpoints:
    if err != nil {
        log.Fatal(err)
    }
    defer cli.Close()

    var lockName = "my-lock"

    var wg sync.WaitGroup
    wg.Add(10)

    for i := 0; i < 10; i++ {
        go startSession(i, cli, lockName, &wg)
    }

    wg.Wait()
}
```

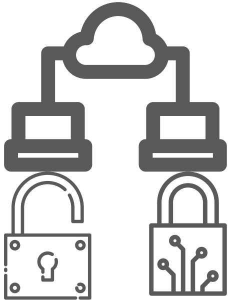


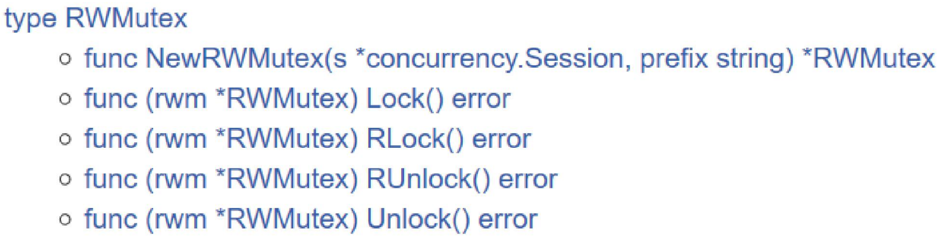
```go
func startSession(id int, cli *clientv3.Client, lockName string, wg *sync.WaitGroup
    defer wg.Done()

    // 为锁生成session
    s1, err := concurrency.NewSession(cli)
    if err != nil {
        log.Fatal(err)
    }
    defer s1.Close()
    m1 := concurrency.NewMutex(s1, lockName)

    // 请求锁
    log.Println("acquiring lock for ID:", id)
    if err := m1.Lock(context.TODO()); err != nil {
        log.Fatal(err)
    }
    log.Println("acquired lock for ID:", id)

    time.Sleep(time.Duration(rand.Intn(10)) * time.Second)

    if err := m1.Unlock(context.TODO()); err != nil {
        log.Fatal(err)
    }
    log.Println("released lock for ID:", id)
}
```

分布式同步原语

github.com/coreos/etcd/contrib/recipes

RWMutex





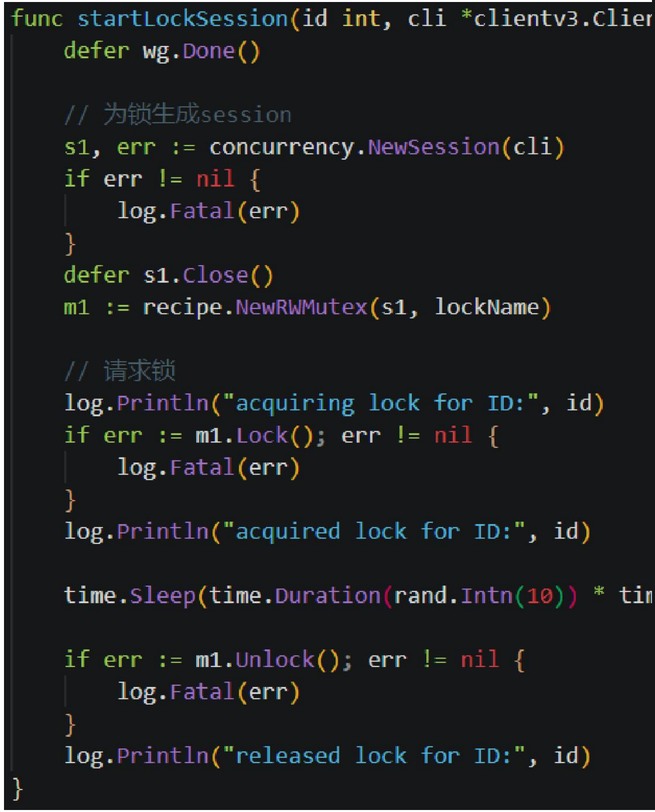
type RWMutex

- func NewRWMutex(s *concurrency.Session, prefix string) *RWMutex
- func (rwm *RWMutex) Lock() error
- func (rwm *RWMutex) RLock() error
- func (rwm *RWMutex) RUnlock() error
- func (rwm *RWMutex) Unlock() error

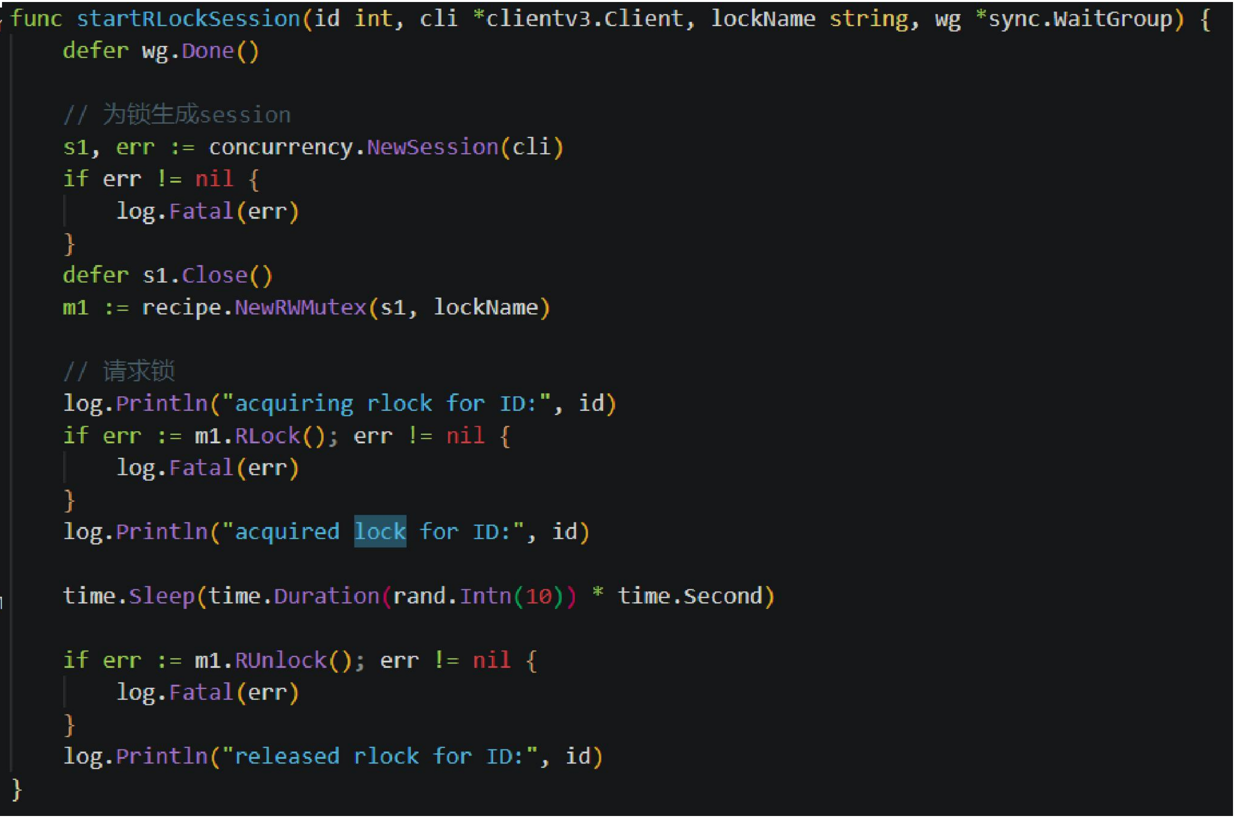
分布式同步原语

github.com/coreos/etcd/contrib/recipes

RWMutex



```go
func startLockSession(id int, cli *clientv3.Clier
    defer wg.Done()

    // 为锁生成session
    s1, err := concurrency.NewSession(cli)
    if err != nil {
        log.Fatal(err)
    }
    defer s1.Close()
    m1 := recipe.NewRWMutex(s1, lockName)

    // 请求锁
    log.Println("acquiring lock for ID:", id)
    if err := m1.Lock(); err != nil {
        log.Fatal(err)
    }
    log.Println("acquired lock for ID:", id)

    time.Sleep(time.Duration(rand.Intn(10)) * tin

    if err := m1.Unlock(); err != nil {
        log.Fatal(err)
    }
    log.Println("released lock for ID:", id)
}
```

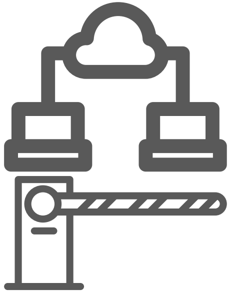


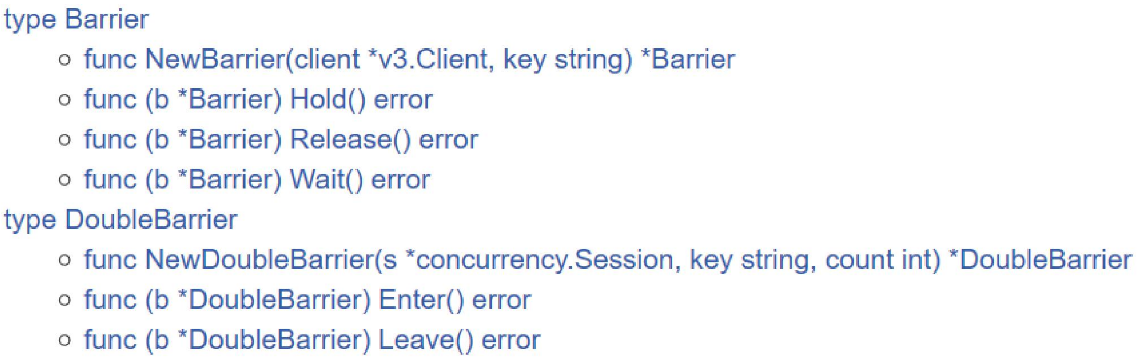
```go
func startRLockSession(id int, cli *clientv3.Client, lockName string, wg *sync.WaitGroup) {
    defer wg.Done()

    // 为锁生成session
    s1, err := concurrency.NewSession(cli)
    if err != nil {
        log.Fatal(err)
    }
    defer s1.Close()
    m1 := recipe.NewRWMutex(s1, lockName)

    // 请求锁
    log.Println("acquiring rlock for ID:", id)
    if err := m1.RLock(); err != nil {
        log.Fatal(err)
    }
    log.Println("acquired lock for ID:", id)

    time.Sleep(time.Duration(rand.Intn(10)) * time.Second)

    if err := m1.RUnlock(); err != nil {
        log.Fatal(err)
    }
    log.Println("released rlock for ID:", id)
}
```

分布式同步原语

github.com/coreos/etcd/contrib/recipes

Barrier





type Barrier

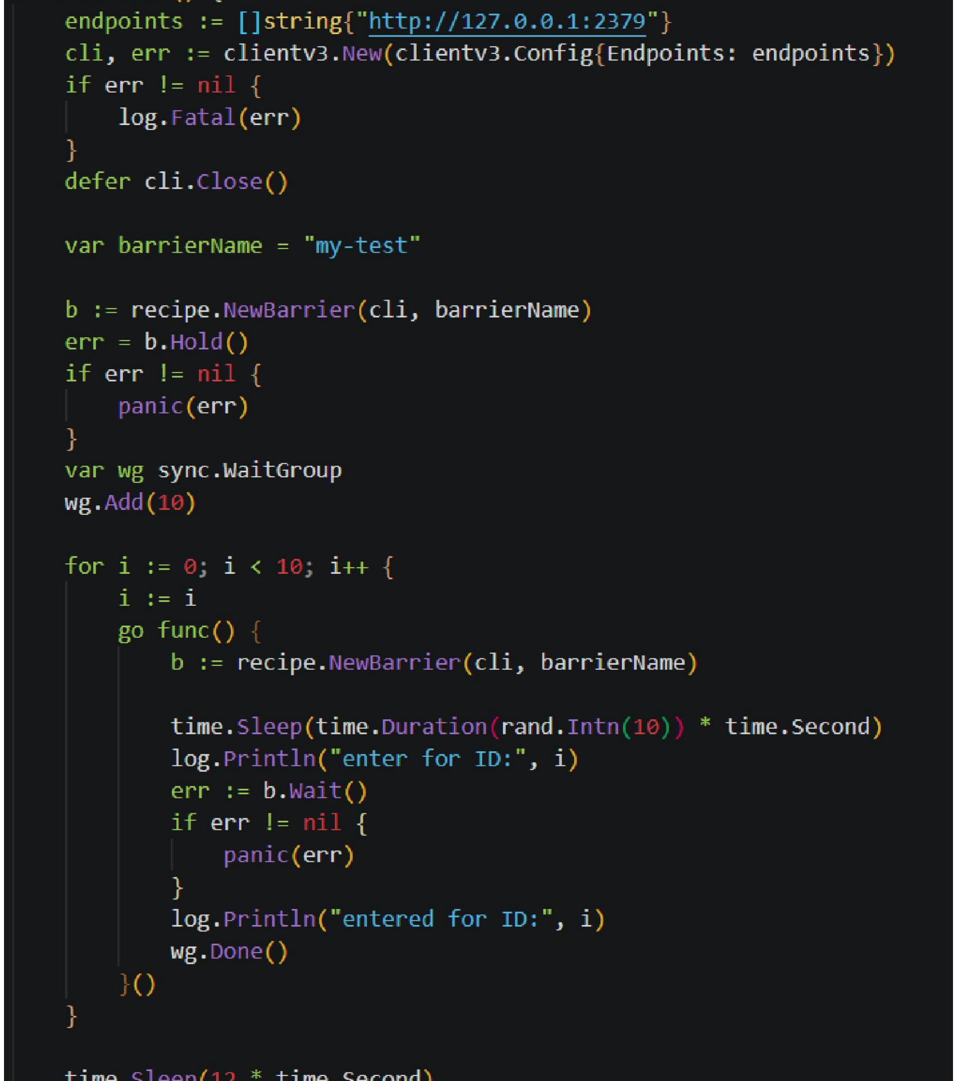
- func NewBarrier(client *v3.Client, key string) *Barrier
- func (b *Barrier) Hold() error
- func (b *Barrier) Release() error
- func (b *Barrier) Wait() error

type DoubleBarrier

- func NewDoubleBarrier(s *concurrency.Session, key string, count int) *DoubleBarrier
- func (b *DoubleBarrier) Enter() error
- func (b *DoubleBarrier) Leave() error

分布式同步原语

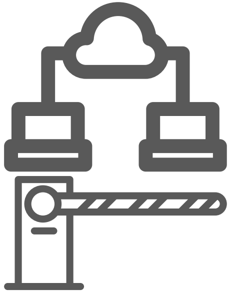
Barrier

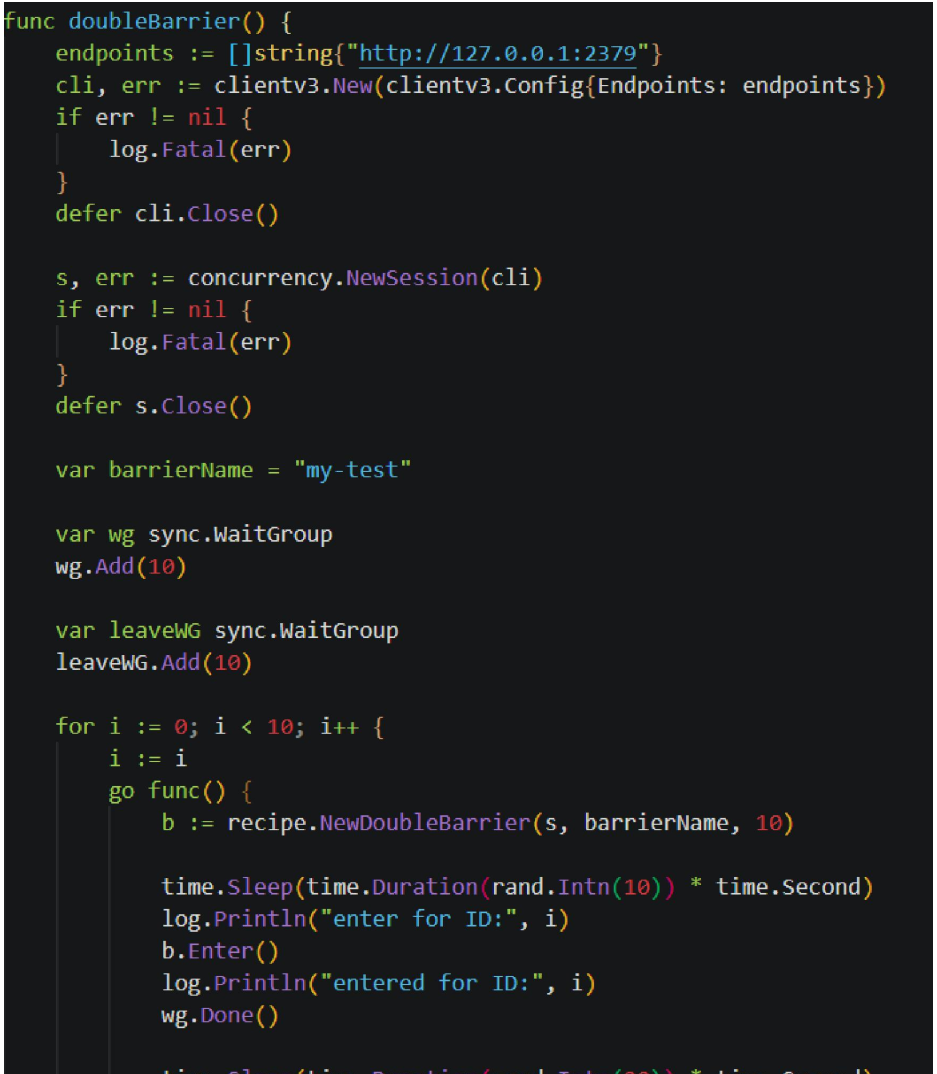


```go
endpoints := []string{"http://127.0.0.1:2379"}
cli, err := clientv3.New(clientv3.Config{Endpoints: endpoints})
if err != nil {
    log.Fatal(err)
}
defer cli.Close()

var barrierName = "my-test"

b := recipe.NewBarrier(cli, barrierName)
err = b.Hold()
if err != nil {
    panic(err)
}
var wg sync.WaitGroup
wg.Add(10)

for i := 0; i < 10; i++ {
    i := i
    go func() {
        b := recipe.NewBarrier(cli, barrierName)

        time.Sleep(time.Duration(rand.Intn(10)) * time.Second)
        log.Println("enter for ID:", i)
        err := b.Wait()
        if err != nil {
            panic(err)
        }
        log.Println("entered for ID:", i)
        wg.Done()
    }()
}

time.Sleep(12 * time.Second)
```

分布式同步原语

Barrier



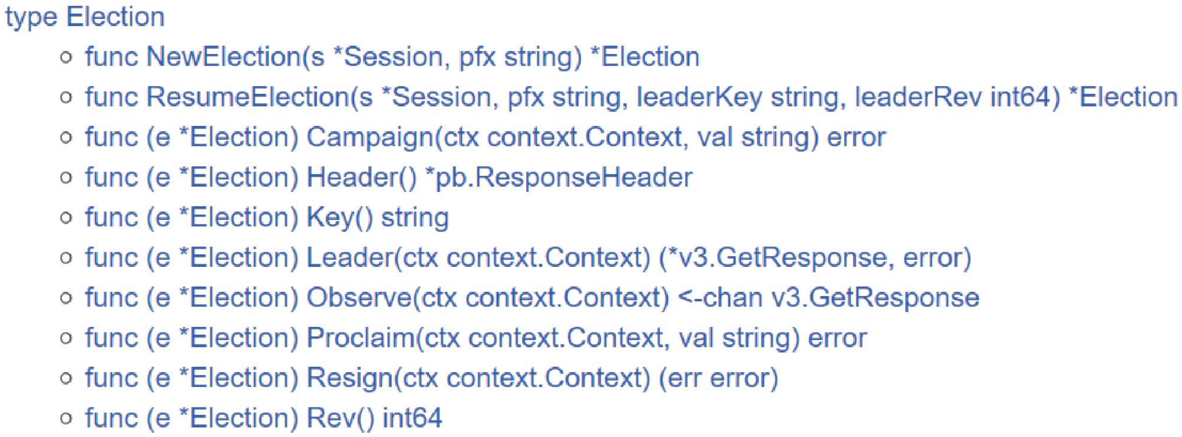


```go
func doubleBarrier() {
    endpoints := []string{"http://127.0.0.1:2379"}
    cli, err := clientv3.New(clientv3.Config{Endpoints: endpoints})
    if err != nil {
        log.Fatal(err)
    }
    defer cli.Close()

    s, err := concurrency.NewSession(cli)
    if err != nil {
        log.Fatal(err)
    }
    defer s.Close()

    var barrierName = "my-test"

    var wg sync.WaitGroup
    wg.Add(10)

    var leaveWG sync.WaitGroup
    leaveWG.Add(10)

    for i := 0; i < 10; i++ {
        i := i
        go func() {
            b := recipe.NewDoubleBarrier(s, barrierName, 10)

            time.Sleep(time.Duration(rand.Intn(10)) * time.Second)
            log.Println("enter for ID:", i)
            b.Enter()
            log.Println("entered for ID:", i)
            wg.Done()

            time.Sleep(time.Duration(rand.Intn(10)) * time.Second)
```

分布式同步原语

Leader Election

github.com/coreos/etcd/clientv3/concurrency

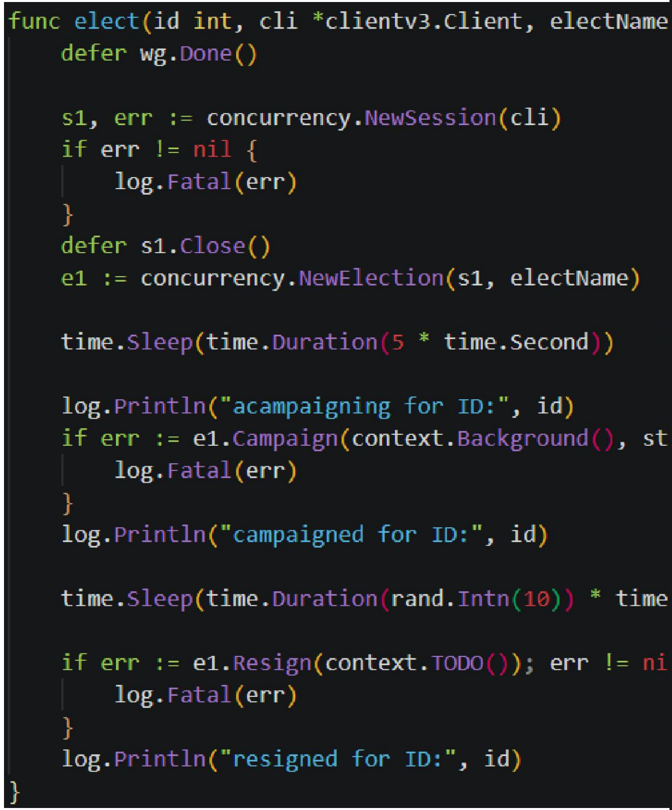


type Election

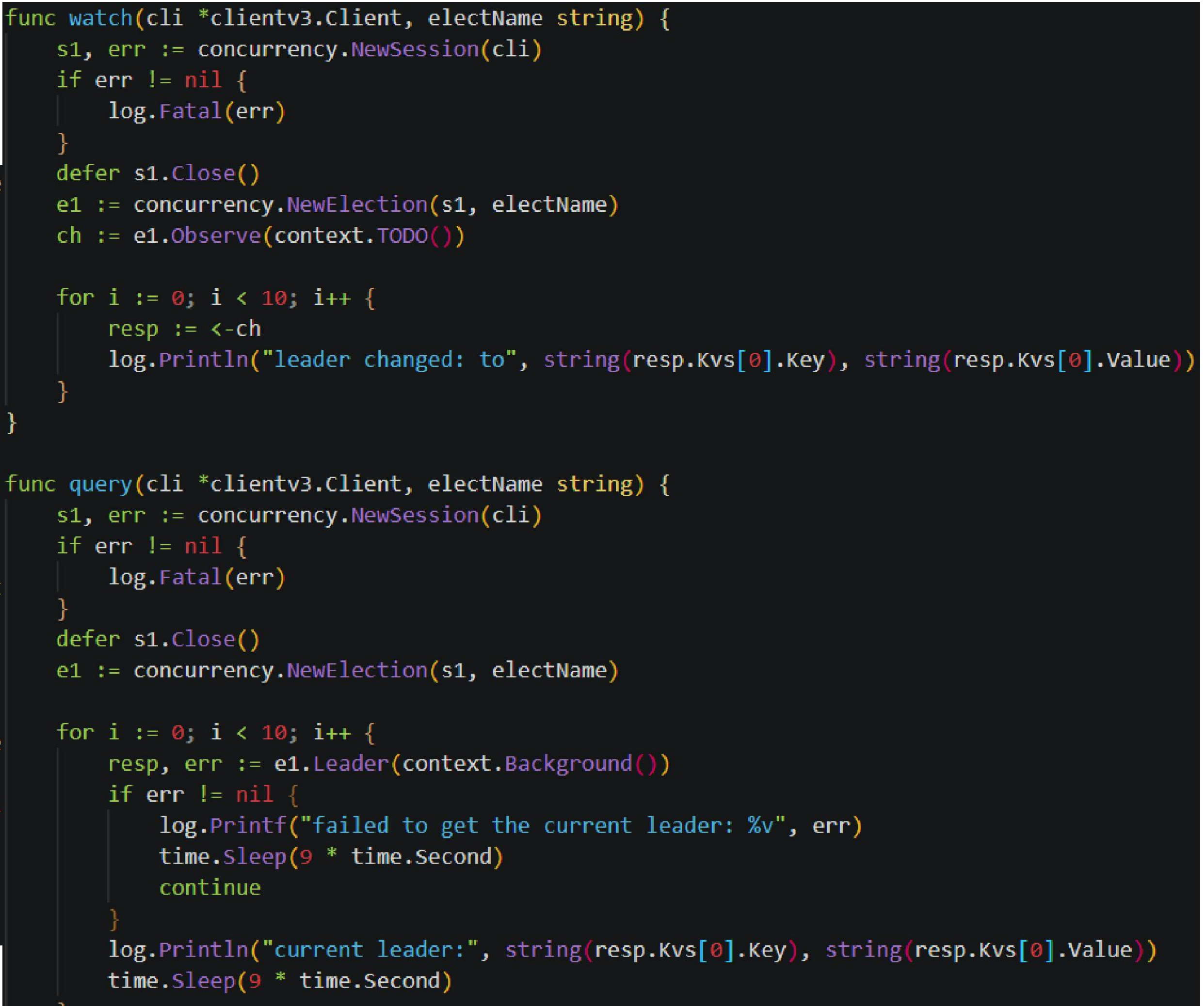
- func NewElection(s *Session, pfx string) *Election
- func ResumeElection(s *Session, pfx string, leaderKey string, leaderRev int64) *Election
- func (e *Election) Campaign(ctx context.Context, val string) error
- func (e *Election) Header() *pb.ResponseHeader
- func (e *Election) Key() string
- func (e *Election) Leader(ctx context.Context) (*v3.GetResponse, error)
- func (e *Election) Observe(ctx context.Context) <-chan v3.GetResponse
- func (e *Election) Proclaim(ctx context.Context, val string) error
- func (e *Election) Resign(ctx context.Context) (err error)
- func (e *Election) Rev() int64

分布式同步原语

Leader Election



```
func elect(id int, cli *clientv3.Client, electName
    defer wg.Done()

    s1, err := concurrency.NewSession(cli)
    if err != nil {
        log.Fatal(err)
    }
    defer s1.Close()
    e1 := concurrency.NewElection(s1, electName)

    time.Sleep(time.Duration(5 * time.Second))

    log.Println("acampaigning for ID:", id)
    if err := e1.Campaign(context.Background(), st
        log.Fatal(err)
    }
    log.Println("campaigned for ID:", id)

    time.Sleep(time.Duration(rand.Intn(10)) * time

    if err := e1.Resign(context.TODO()); err != ni
        log.Fatal(err)
    }
    log.Println("resigned for ID:", id)
}
```

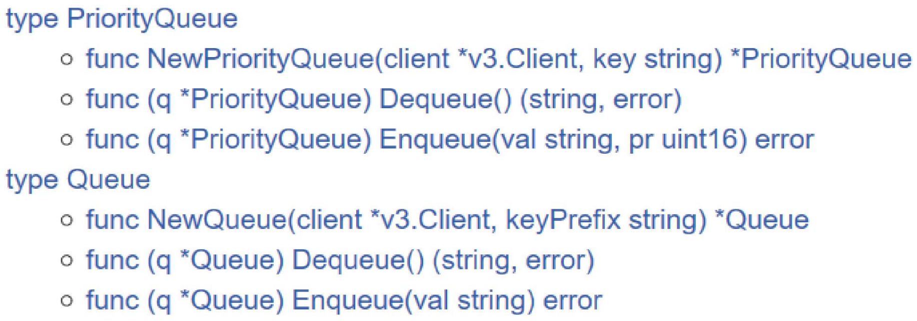


```
func watch(cli *clientv3.Client, electName string) {
    s1, err := concurrency.NewSession(cli)
    if err != nil {
        log.Fatal(err)
    }
    defer s1.Close()
    e1 := concurrency.NewElection(s1, electName)
    ch := e1.Observe(context.TODO())

    for i := 0; i < 10; i++ {
        resp := <-ch
        log.Println("leader changed: to", string(resp.Kvs[0].Key), string(resp.Kvs[0].Value))
    }
}

func query(cli *clientv3.Client, electName string) {
    s1, err := concurrency.NewSession(cli)
    if err != nil {
        log.Fatal(err)
    }
    defer s1.Close()
    e1 := concurrency.NewElection(s1, electName)

    for i := 0; i < 10; i++ {
        resp, err := e1.Leader(context.Background())
        if err != nil {
            log.Printf("failed to get the current leader: %v", err)
            time.Sleep(9 * time.Second)
            continue
        }
        log.Println("current leader:", string(resp.Kvs[0].Key), string(resp.Kvs[0].Value))
        time.Sleep(9 * time.Second)
```

分布式同步原语

队列

github.com/coreos/etcd/contrib/recipes



type PriorityQueue

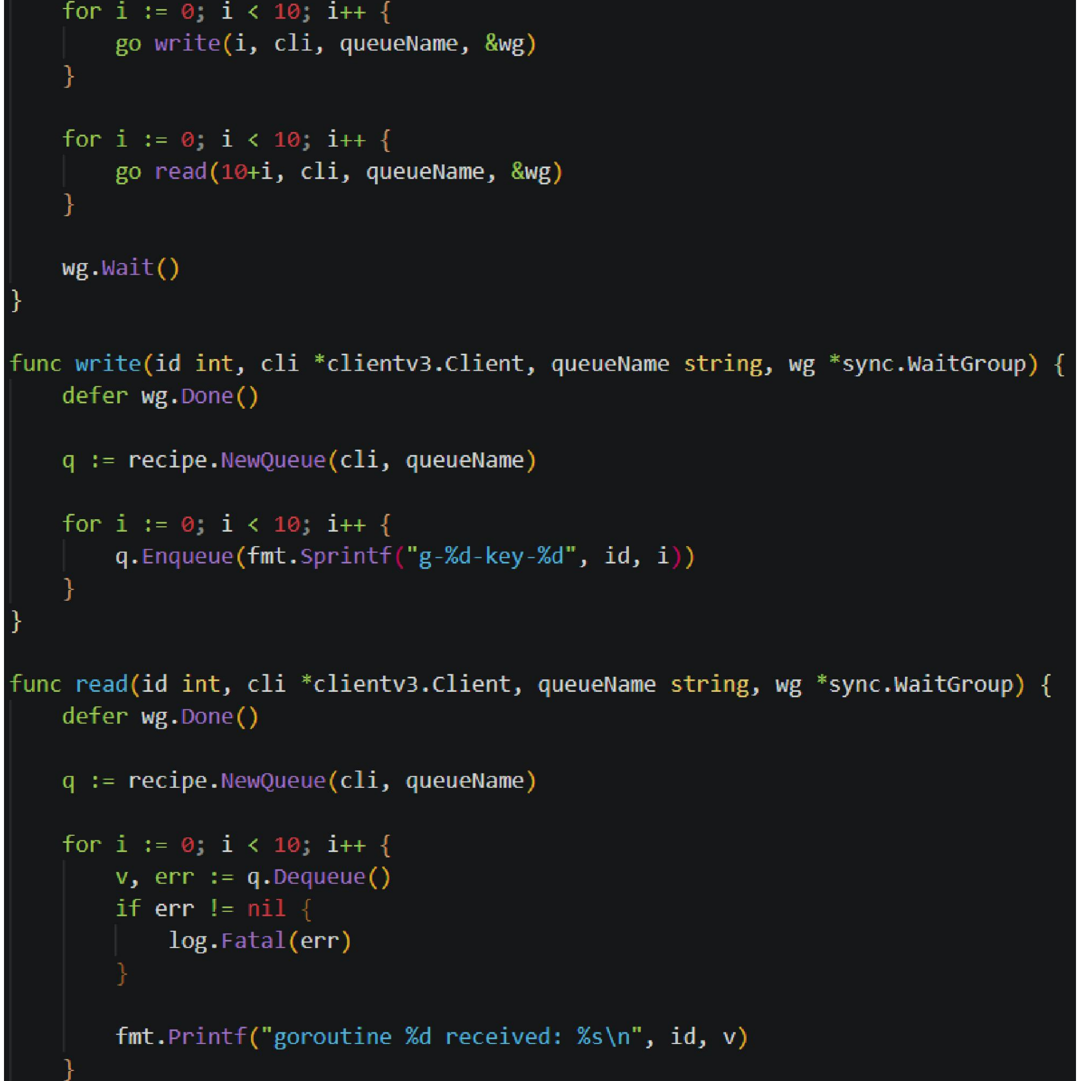
- func NewPriorityQueue(client *v3.Client, key string) *PriorityQueue
- func (q *PriorityQueue) Dequeue() (string, error)
- func (q *PriorityQueue) Enqueue(val string, pr uint16) error

type Queue

- func NewQueue(client *v3.Client, keyPrefix string) *Queue
- func (q *Queue) Dequeue() (string, error)
- func (q *Queue) Enqueue(val string) error

分布式同步原语

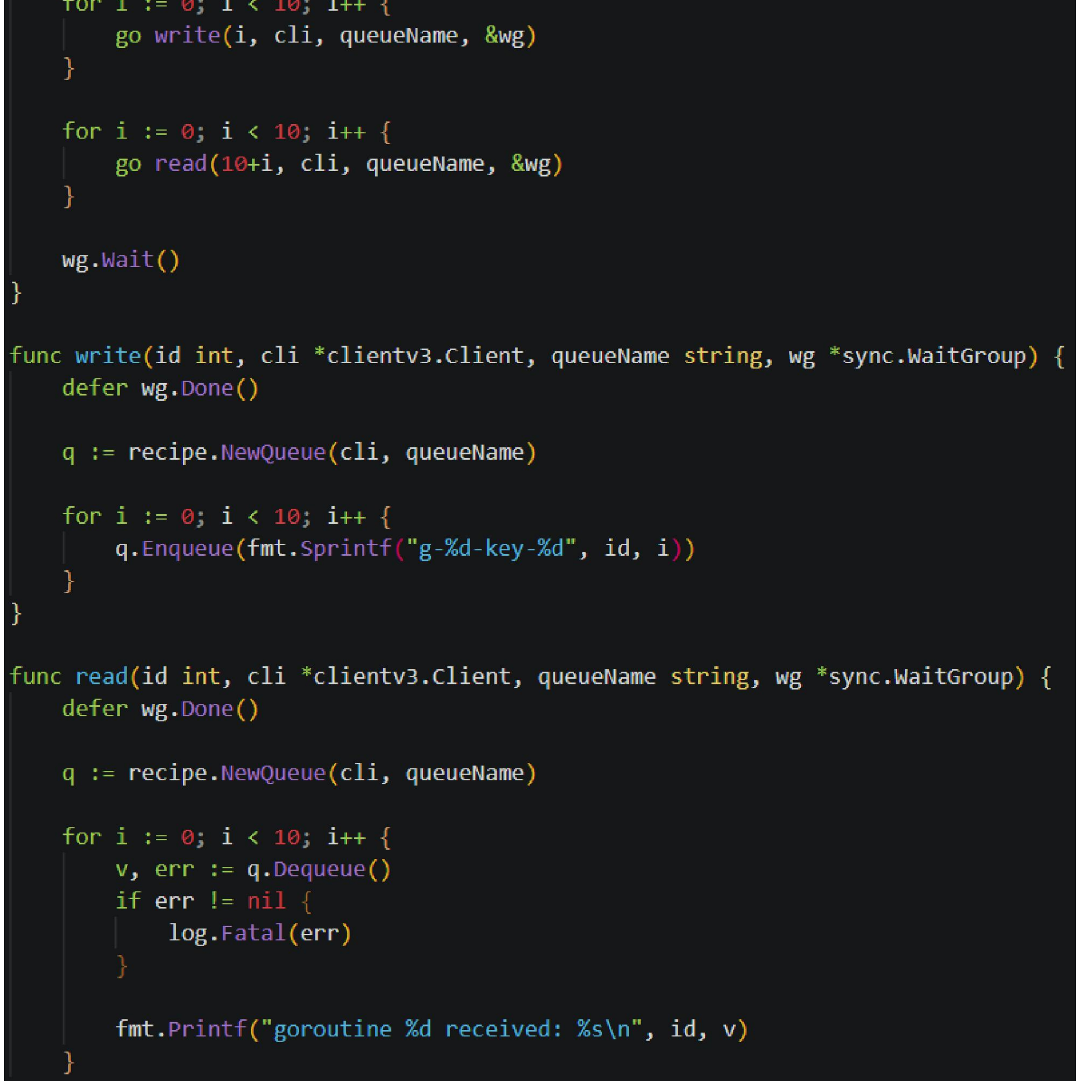
队列



```go
    for i := 0; i < 10; i++ {
        go write(i, cli, queueName, &wg)
    }

    for i := 0; i < 10; i++ {
        go read(10+i, cli, queueName, &wg)
    }

    wg.Wait()
}

func write(id int, cli *clientv3.Client, queueName string, wg *sync.WaitGroup) {
    defer wg.Done()

    q := recipe.NewQueue(cli, queueName)

    for i := 0; i < 10; i++ {
        q.Enqueue(fmt.Sprintf("g-%d-key-%d", id, i))
    }
}

func read(id int, cli *clientv3.Client, queueName string, wg *sync.WaitGroup) {
    defer wg.Done()

    q := recipe.NewQueue(cli, queueName)

    for i := 0; i < 10; i++ {
        v, err := q.Dequeue()
        if err != nil {
            log.Fatal(err)
        }

        fmt.Printf("goroutine %d received: %s\n", id, v)
    }
```

分布式同步原语

优先级队列

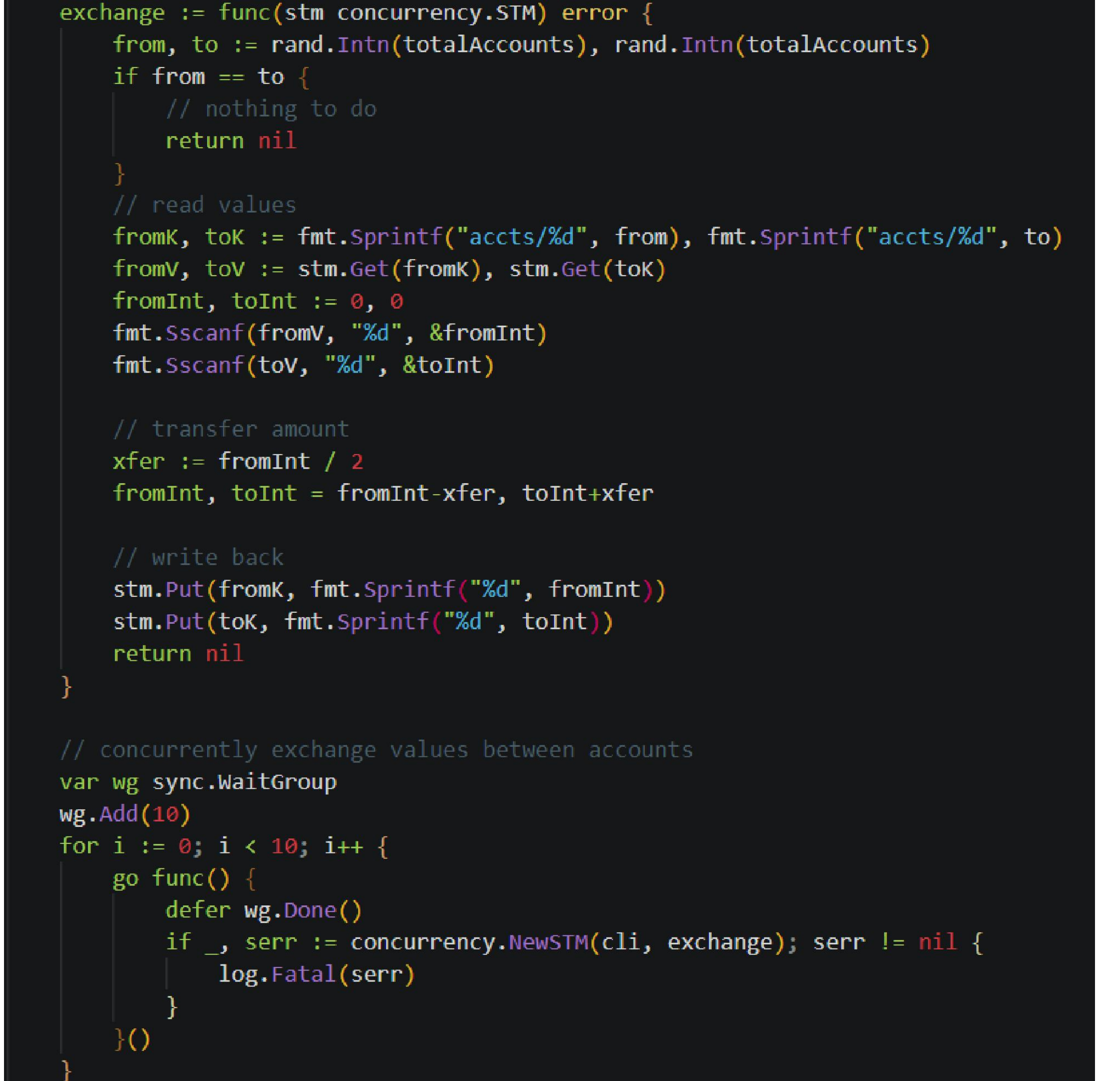


```
    for i := 0; i < 10; i++ {
        go write(i, cli, queueName, &wg)
    }

    for i := 0; i < 10; i++ {
        go read(10+i, cli, queueName, &wg)
    }

    wg.Wait()
}

func write(id int, cli *clientv3.Client, queueName string, wg *sync.WaitGroup) {
    defer wg.Done()

    q := recipe.NewQueue(cli, queueName)

    for i := 0; i < 10; i++ {
        q.Enqueue(fmt.Sprintf("g-%d-key-%d", id, i))
    }
}

func read(id int, cli *clientv3.Client, queueName string, wg *sync.WaitGroup) {
    defer wg.Done()

    q := recipe.NewQueue(cli, queueName)

    for i := 0; i < 10; i++ {
        v, err := q.Dequeue()
        if err != nil {
            log.Fatal(err)
        }

        fmt.Printf("goroutine %d received: %s\n", id, v)
    }
```

分布式同步原语

STM

software transactional memory

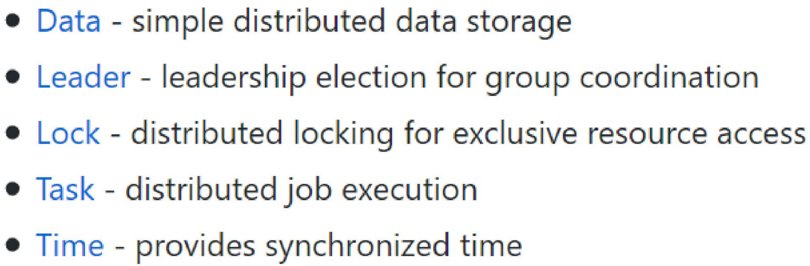


```go
exchange := func(stm concurrency.STM) error {
    from, to := rand.Intn(totalAccounts), rand.Intn(totalAccounts)
    if from == to {
        // nothing to do
        return nil
    }
    // read values
    fromK, toK := fmt.Sprintf("accts/%d", from), fmt.Sprintf("accts/%d", to)
    fromV, toV := stm.Get(fromK), stm.Get(toK)
    fromInt, toInt := 0, 0
    fmt.Sscanf(fromV, "%d", &fromInt)
    fmt.Sscanf(toV, "%d", &toInt)

    // transfer amount
    xfer := fromInt / 2
    fromInt, toInt = fromInt-xfer, toInt+xfer

    // write back
    stm.Put(fromK, fmt.Sprintf("%d", fromInt))
    stm.Put(toK, fmt.Sprintf("%d", toInt))
    return nil
}

// concurrently exchange values between accounts
var wg sync.WaitGroup
wg.Add(10)
for i := 0; i < 10; i++ {
    go func() {
        defer wg.Done()
        if _, serr := concurrency.NewSTM(cli, exchange); serr != nil {
            log.Fatal(serr)
        }
    }()
}
```

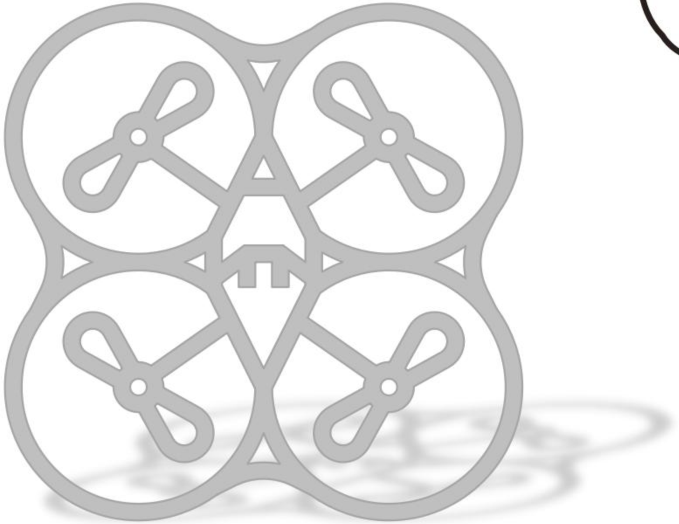
分布式同步原语

micro/go-sync

github.com/micro/go-sync



- Data - simple distributed data storage
- Leader - leadership election for group coordination
- Lock - distributed locking for exclusive resource access
- Task - distributed job execution
- Time - provides synchronized time



03

Channel

Channel

功能

- 信号 (shutdown/close/finish)
- 数据交流 (queue/stream)
- 锁 (mutex)

Channel

特别场景

	nil	not empty	empty	full	not full	closed
receive	block	value	block	value	value	drained read, return zero value
send	block	write value	write value	block	write value	panic
close	panic	closed, drained read, return zero value	closed, return zero value for read	closed, drained read, return zero value	closed, drained read, return zero value	panic

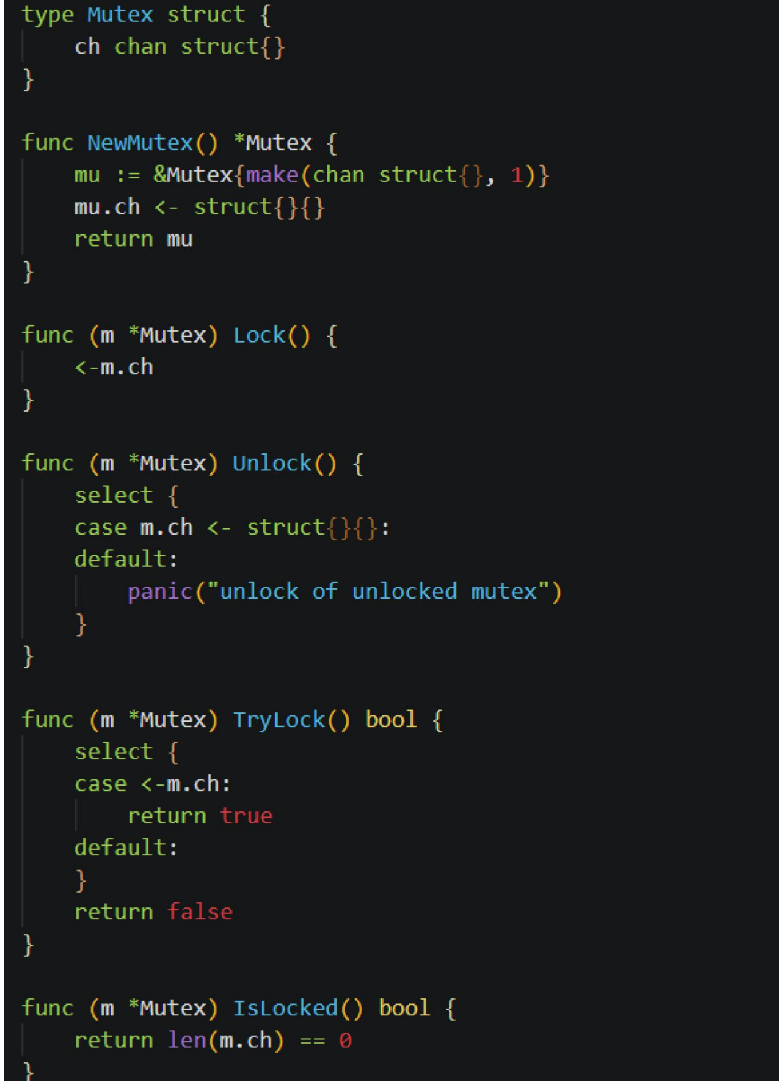
Channel

内部实现

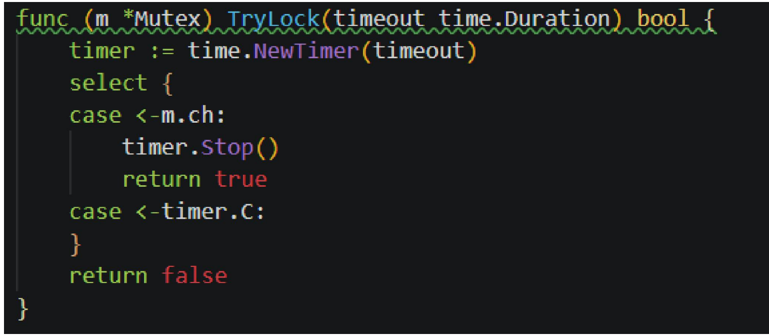
- 源代码：https://golang.org/src/runtime/chan.go
- GopherCon 2017: Kavya Joshi - Understanding Channels
- Go 语言 Channel 实现原理精要

Channel

Locker



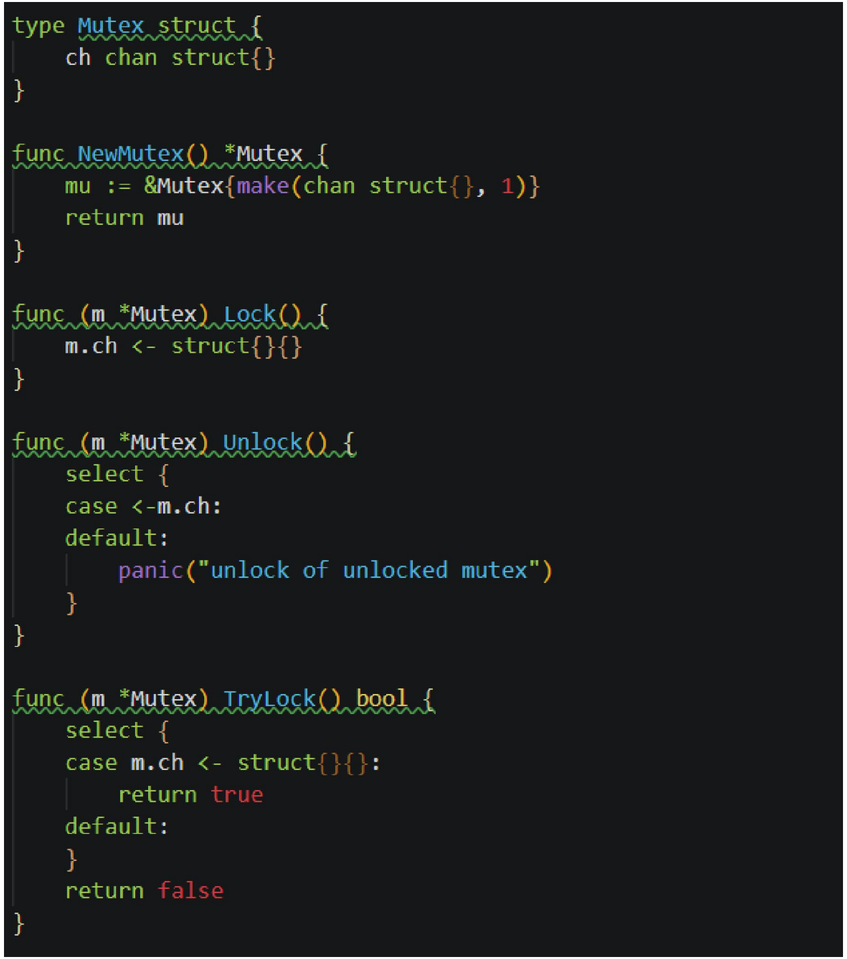
```go
type Mutex struct {
    ch chan struct{}
}

func NewMutex() *Mutex {
    mu := &Mutex{make(chan struct{}, 1)}
    mu.ch <- struct{}{}
    return mu
}

func (m *Mutex) Lock() {
    <-m.ch
}

func (m *Mutex) Unlock() {
    select {
    case m.ch <- struct{}{}:
    default:
        panic("unlock of unlocked mutex")
    }
}

func (m *Mutex) TryLock() bool {
    select {
    case <-m.ch:
        return true
    default:
    }
    return false
}

func (m *Mutex) IsLocked() bool {
    return len(m.ch) == 0
}
```



```go
func (m *Mutex) TryLock(timeout time.Duration) bool {
    timer := time.NewTimer(timeout)
    select {
    case <-m.ch:
        timer.Stop()
        return true
    case <-timer.C:
    }
    return false
}
```

Channel

Locker



```go
type Mutex struct {
    ch chan struct{}
}

func NewMutex() *Mutex {
    mu := &Mutex{make(chan struct{}, 1)}
    return mu
}

func (m *Mutex) Lock() {
    m.ch <- struct{}{}
}

func (m *Mutex) Unlock() {
    select {
    case <-m.ch:
    default:
        panic("unlock of unlocked mutex")
    }
}

func (m *Mutex) TryLock() bool {
    select {
    case m.ch <- struct{}{}:
        return true
    default:
    }
    return false
}
```

Channel

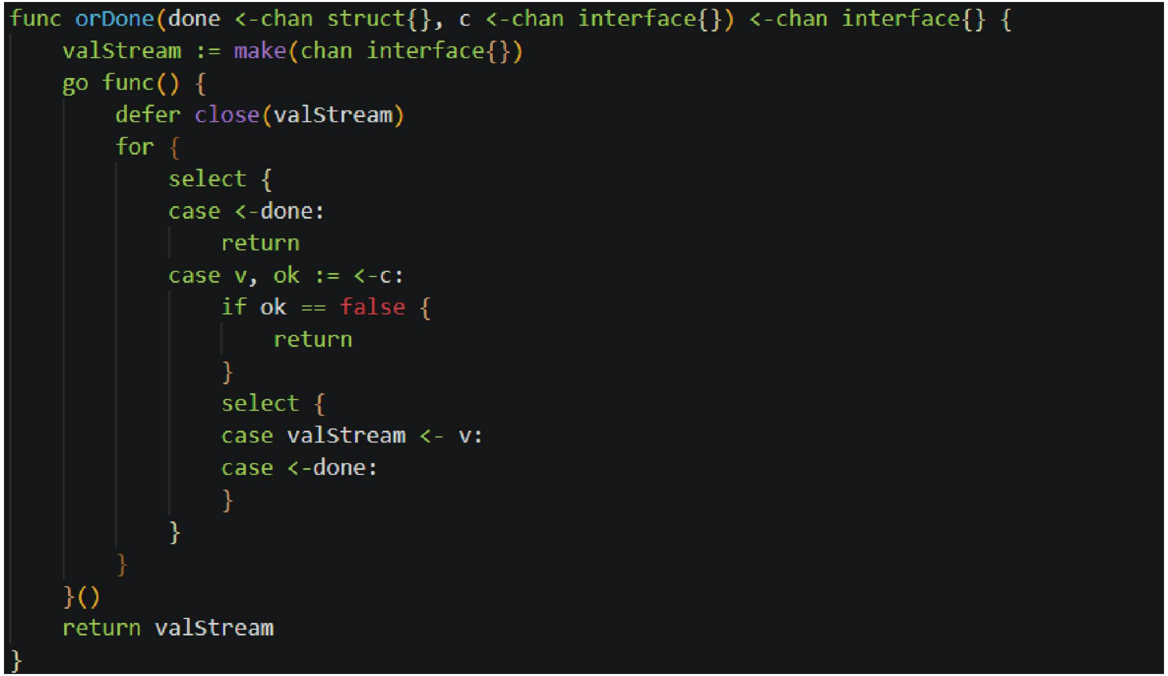
Channel vs Mutex

过度使用channel和goroutine

- Channel
  - 传递数据的owner
  - 分发任务单元
  - 交流异步结果
  - 任务编排
- Mutex
  - cache
  - 状态
  - 临界区

Channel

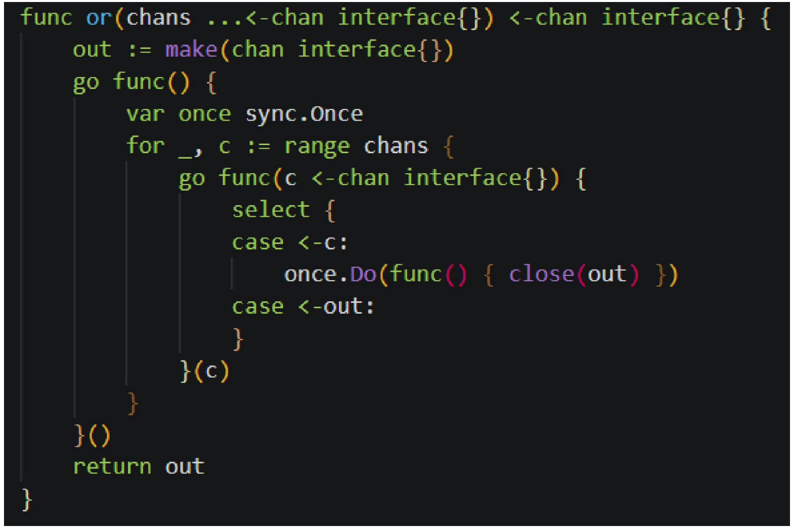
Or-Done



```go
func orDone(done <-chan struct{}, c <-chan interface{}) <-chan interface{} {
    valStream := make(chan interface{})
    go func() {
        defer close(valStream)
        for {
            select {
            case <-done:
                return
            case v, ok := <-c:
                if ok == false {
                    return
                }
                select {
                case valStream <- v:
                case <-done:
                }
            }
        }
    }()
    return valStream
}
```

Channel

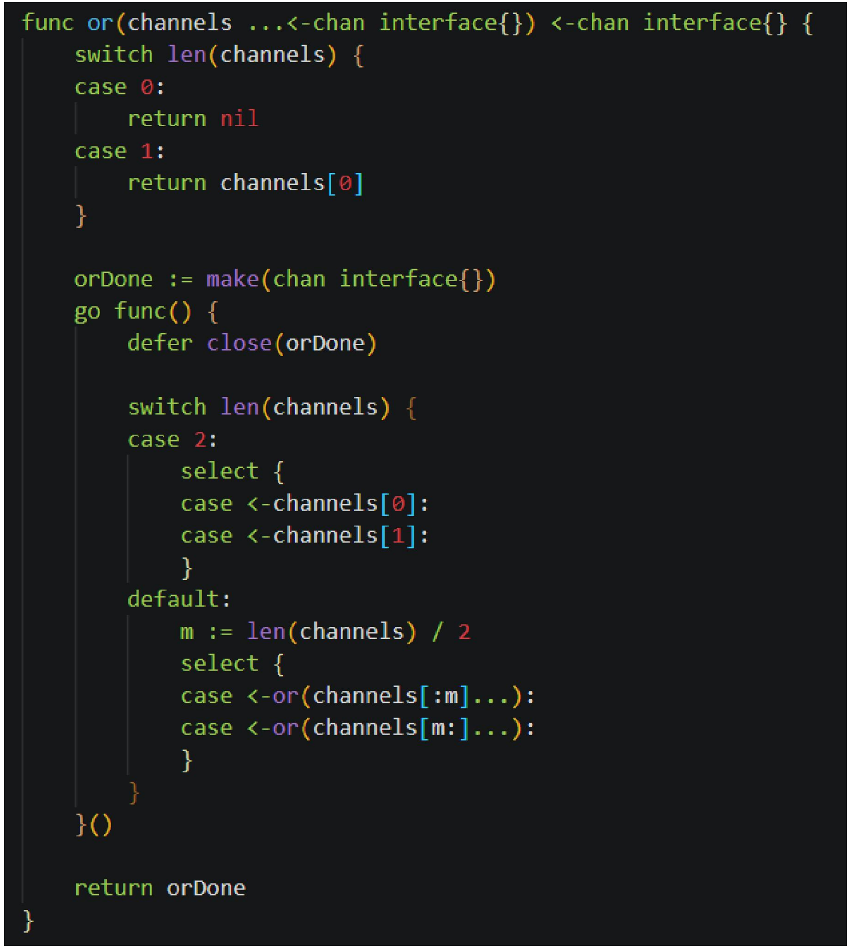
Or-Done goroutine



```go
func or(chans ...<-chan interface{}) <-chan interface{} {
    out := make(chan interface{})
    go func() {
        var once sync.Once
        for _, c := range chans {
            go func(c <-chan interface{}) {
                select {
                case <-c:
                    once.Do(func() { close(out) })
                case <-out:
                }
            }(c)
        }
    }()
    return out
}
```

Channel

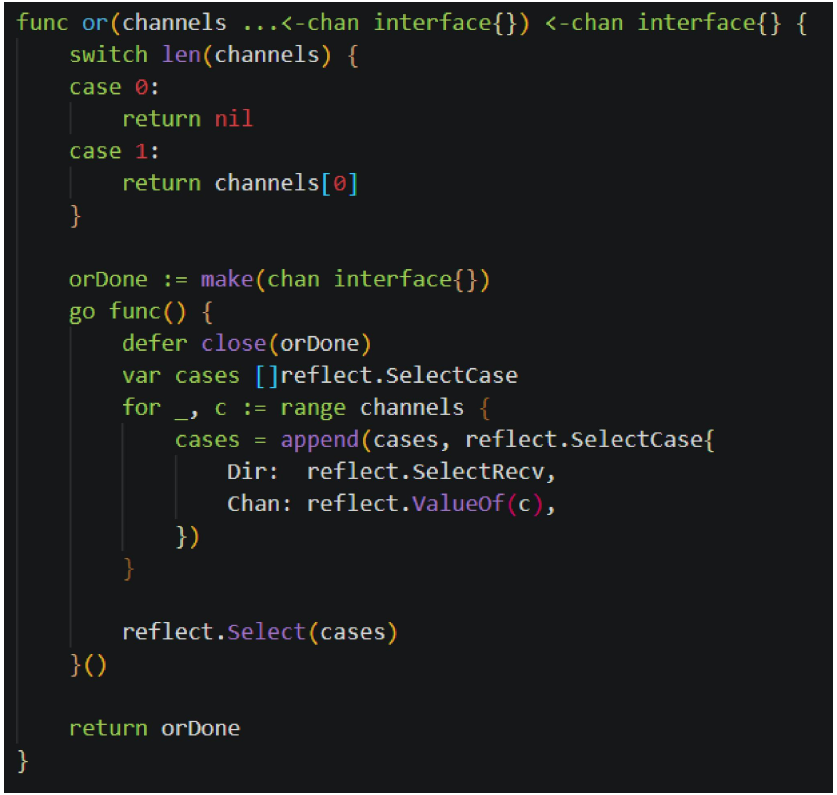
Or-Done 二分法递归



```go
func or(channels ...<-chan interface{}) <-chan interface{} {
    switch len(channels) {
    case 0:
        return nil
    case 1:
        return channels[0]
    }

    orDone := make(chan interface{})
    go func() {
        defer close(orDone)

        switch len(channels) {
        case 2:
            select {
            case <-channels[0]:
            case <-channels[1]:
            }
        default:
            m := len(channels) / 2
            select {
            case <-or(channels[:m]...):
            case <-or(channels[m:]...):
            }
        }
    }()

    return orDone
}
```

Channel

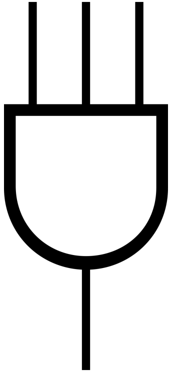
Or-Done 反射

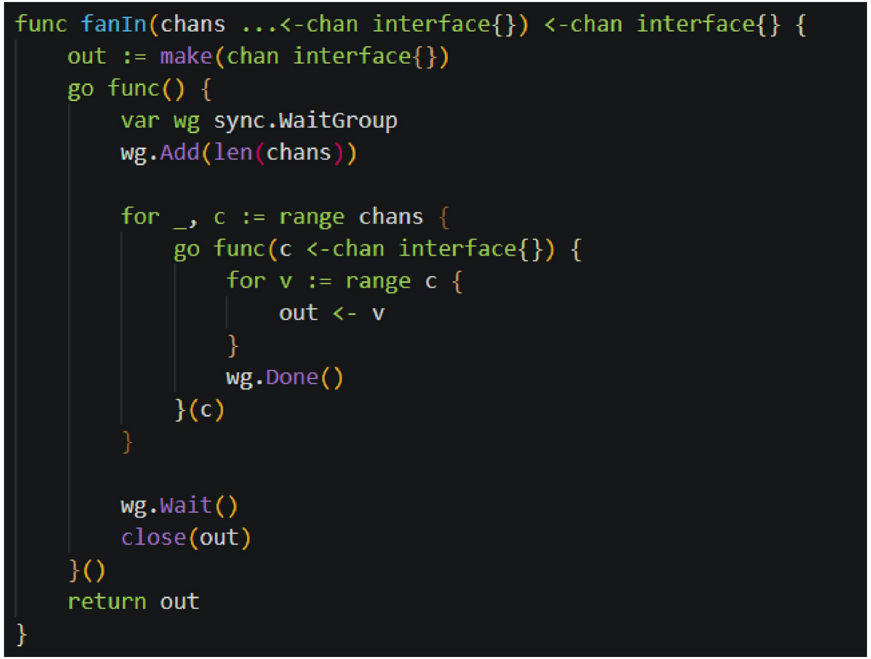


```go
func or(channels ...<-chan interface{}) <-chan interface{} {
    switch len(channels) {
    case 0:
        return nil
    case 1:
        return channels[0]
    }

    orDone := make(chan interface{})
    go func() {
        defer close(orDone)
        var cases []reflect.SelectCase
        for _, c := range channels {
            cases = append(cases, reflect.SelectCase{
                Dir:  reflect.SelectRecv,
                Chan: reflect.ValueOf(c),
            })
        }

        reflect.Select(cases)
    }()

    return orDone
}
```

Channel

Fan In goroutine

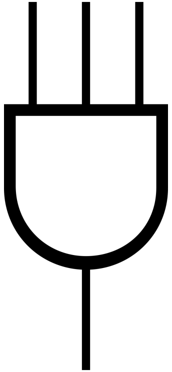


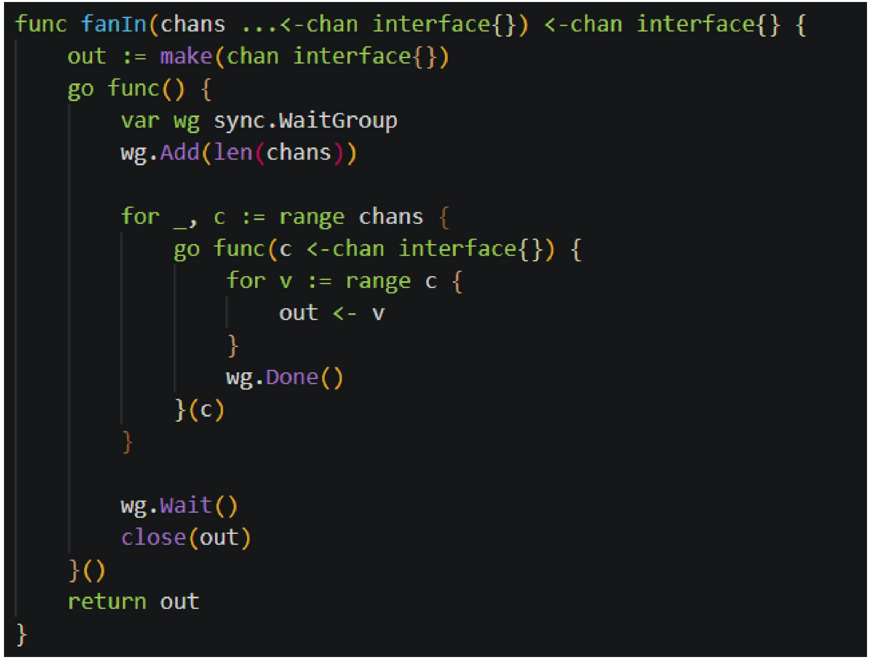


```go
func fanIn(chans ...<-chan interface{}) <-chan interface{} {
    out := make(chan interface{})
    go func() {
        var wg sync.WaitGroup
        wg.Add(len(chans))

        for _, c := range chans {
            go func(c <-chan interface{}) {
                for v := range c {
                    out <- v
                }
                wg.Done()
            }(c)
        }

        wg.Wait()
        close(out)
    }()
    return out
}
```

Channel

Fan In goroutine

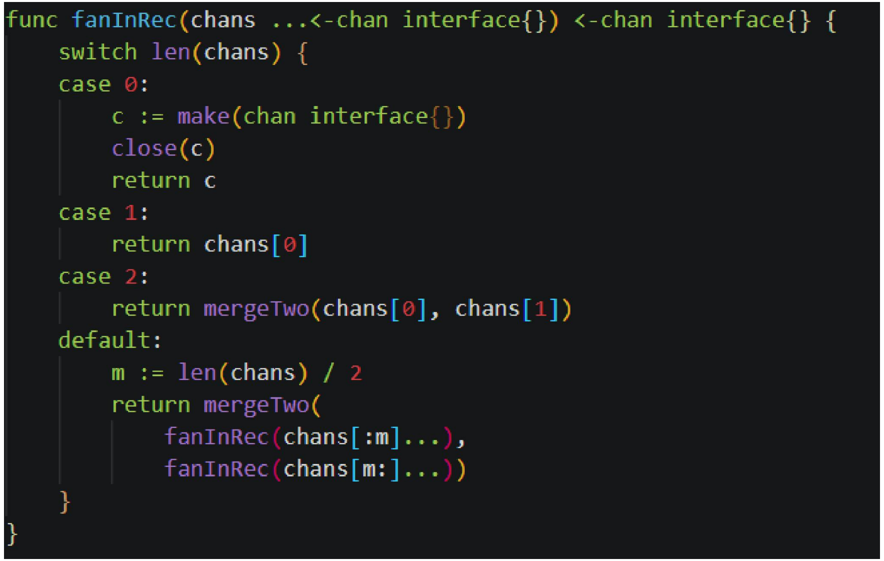




```go
func fanIn(chans ...<-chan interface{}) <-chan interface{} {
    out := make(chan interface{})
    go func() {
        var wg sync.WaitGroup
        wg.Add(len(chans))

        for _, c := range chans {
            go func(c <-chan interface{}) {
                for v := range c {
                    out <- v
                }
                wg.Done()
            }(c)
        }

        wg.Wait()
        close(out)
    }()
    return out
}
```

Channel

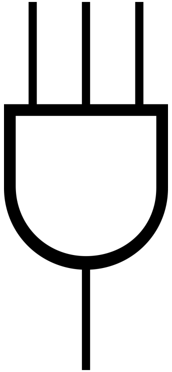
Fan In 递归

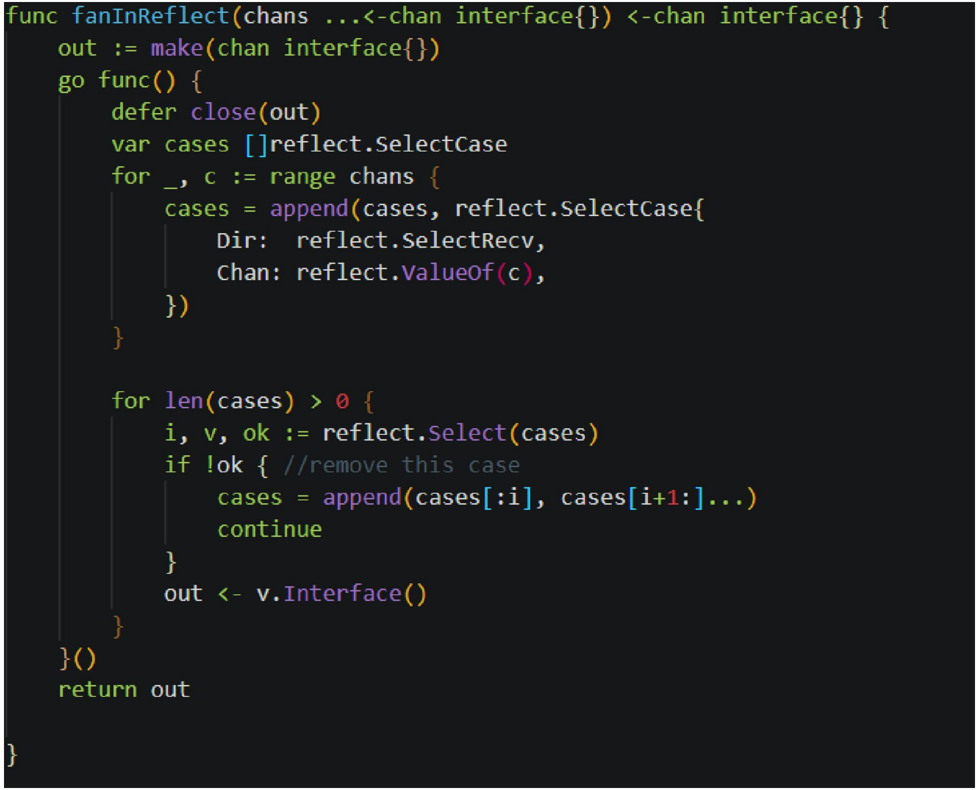


```go
func fanInRec(chans ...<-chan interface{}) <-chan interface{} {
    switch len(chans) {
    case 0:
        c := make(chan interface{})
        close(c)
        return c
    case 1:
        return chans[0]
    case 2:
        return mergeTwo(chans[0], chans[1])
    default:
        m := len(chans) / 2
        return mergeTwo(
            fanInRec(chans[:m]...),
            fanInRec(chans[m:]...))
    }
}
```

Channel

Fan In 反射

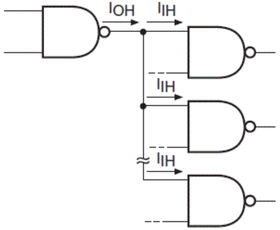


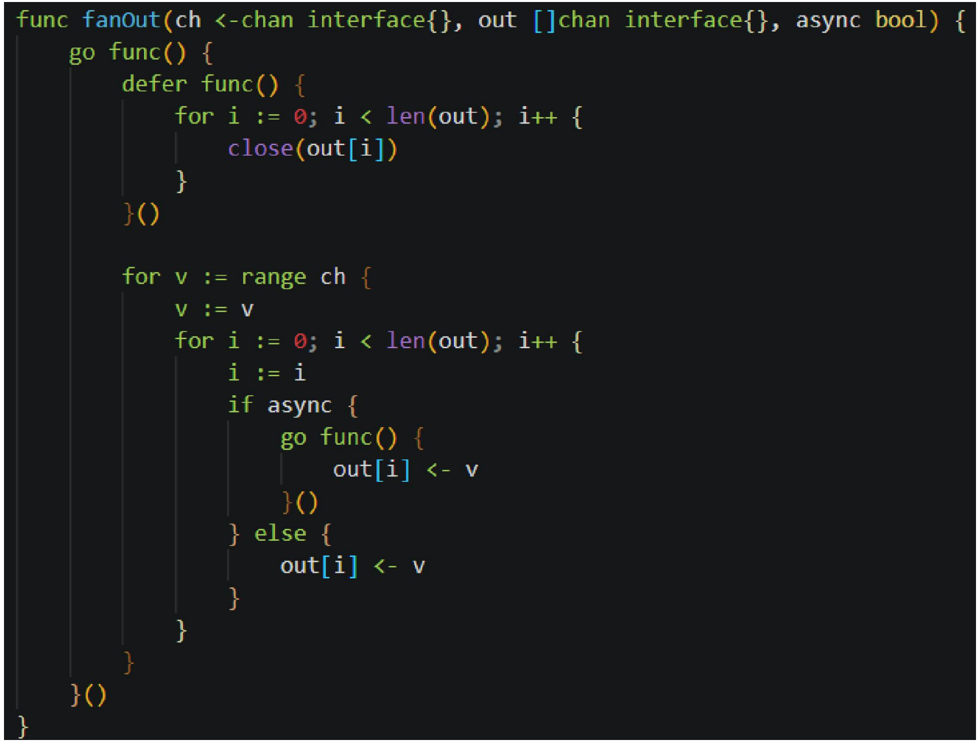


```go
func fanInReflect(chans ...<-chan interface{}) <-chan interface{} {
    out := make(chan interface{})
    go func() {
        defer close(out)
        var cases []reflect.SelectCase
        for _, c := range chans {
            cases = append(cases, reflect.SelectCase{
                Dir:  reflect.SelectRecv,
                Chan: reflect.ValueOf(c),
            })
        }

        for len(cases) > 0 {
            i, v, ok := reflect.Select(cases)
            if !ok { //remove this case
                cases = append(cases[:i], cases[i+1:]...)
                continue
            }
            out <- v.Interface()
        }
    }()
    return out

}
```

Channel

Fan Out goroutine

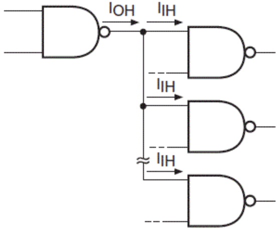


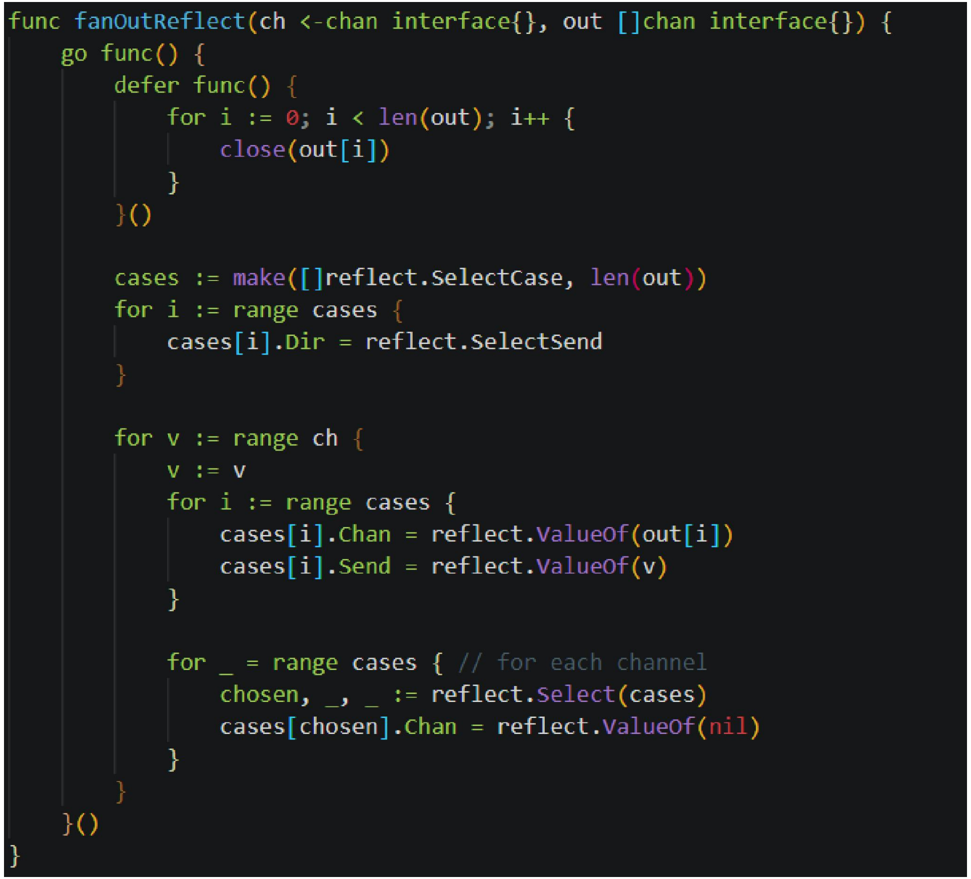


```go
func fanOut(ch <-chan interface{}, out []chan interface{}, async bool) {
    go func() {
        defer func() {
            for i := 0; i < len(out); i++ {
                close(out[i])
            }
        }()

        for v := range ch {
            v := v
            for i := 0; i < len(out); i++ {
                i := i
                if async {
                    go func() {
                        out[i] <- v
                    }()
                } else {
                    out[i] <- v
                }
            }
        }
    }()
}
```

Channel

Fan Out 反射

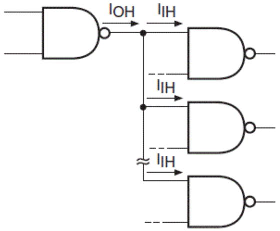


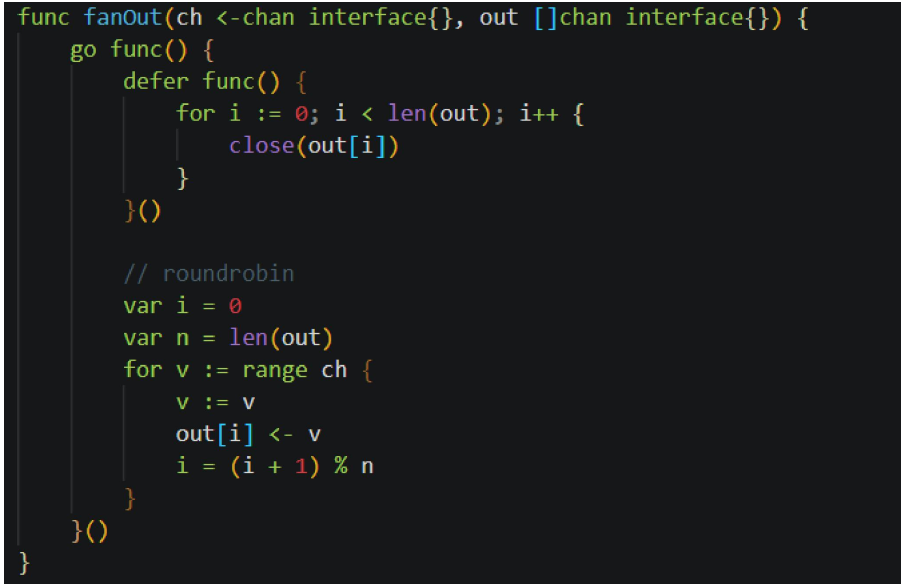


```go
func fanOutReflect(ch <-chan interface{}, out []chan interface{}) {
    go func() {
        defer func() {
            for i := 0; i < len(out); i++ {
                close(out[i])
            }
        }()

        cases := make([]reflect.SelectCase, len(out))
        for i := range cases {
            cases[i].Dir = reflect.SelectSend
        }

        for v := range ch {
            v := v
            for i := range cases {
                cases[i].Chan = reflect.ValueOf(out[i])
                cases[i].Send = reflect.ValueOf(v)
            }

            for _ = range cases { // for each channel
                chosen, _, _ := reflect.Select(cases)
                cases[chosen].Chan = reflect.ValueOf(nil)
            }
        }
    }()
}
```

Channel

Fan out roundrobin

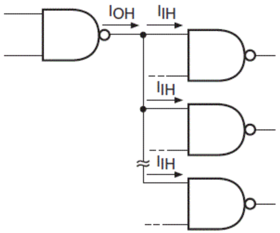


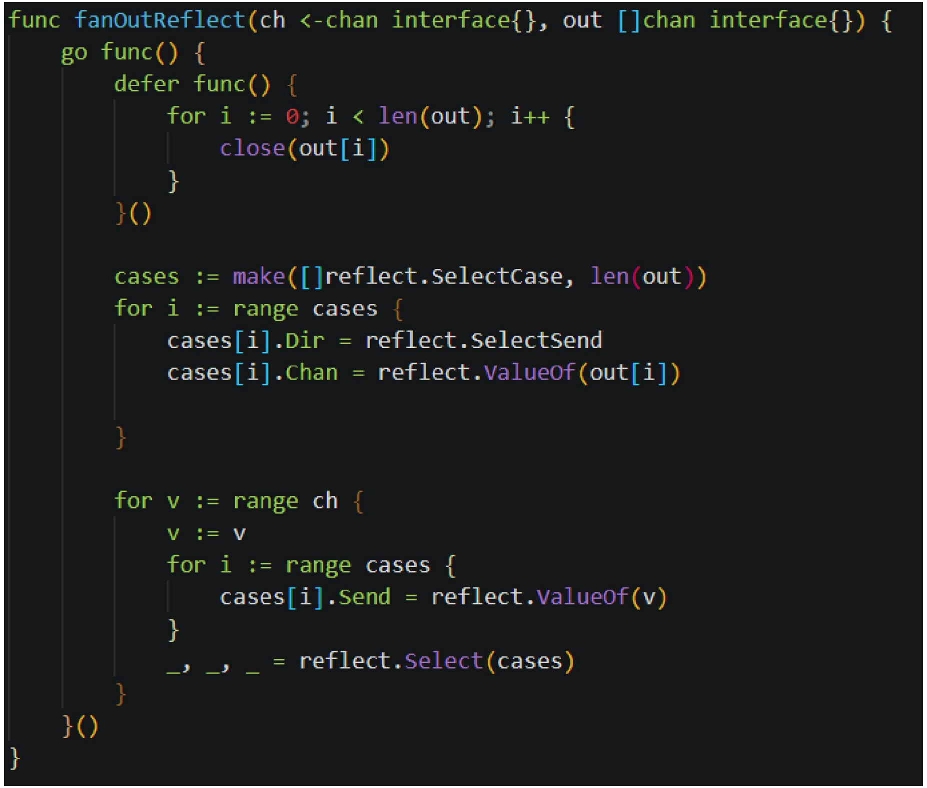


```
func fanOut(ch <-chan interface{}, out []chan interface{}) {
    go func() {
        defer func() {
            for i := 0; i < len(out); i++ {
                close(out[i])
            }
        }()

        // roundrobin
        var i = 0
        var n = len(out)
        for v := range ch {
            v := v
            out[i] <- v
            i = (i + 1) % n
        }
    }()
}
```

Channel

Fan out 反射

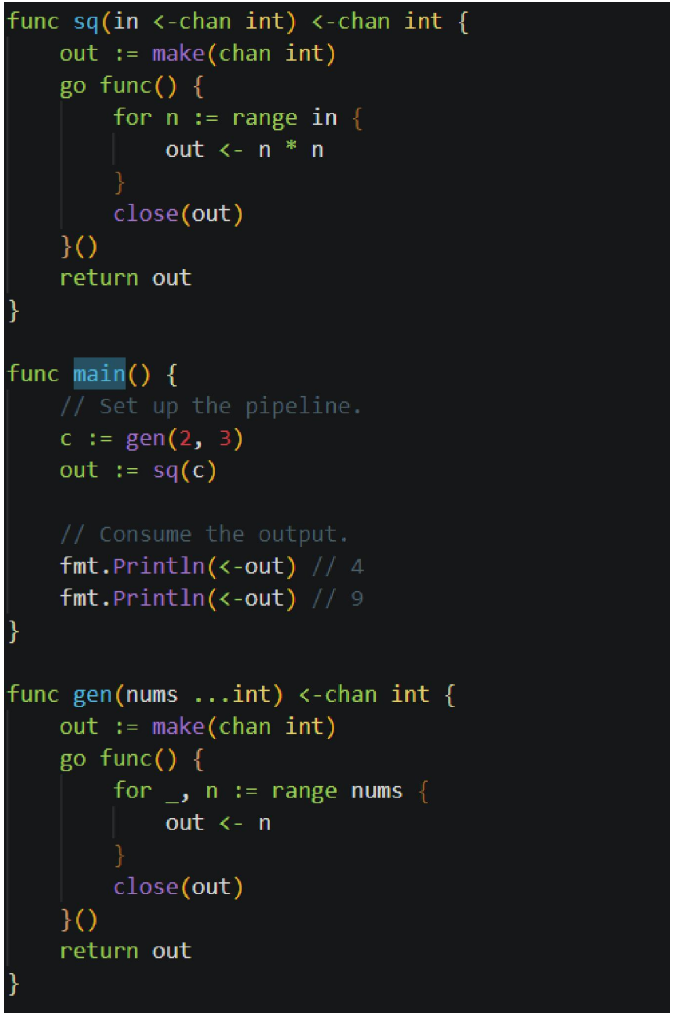




```go
func fanOutReflect(ch <-chan interface{}, out []chan interface{}) {
    go func() {
        defer func() {
            for i := 0; i < len(out); i++ {
                close(out[i])
            }
        }()

        cases := make([]reflect.SelectCase, len(out))
        for i := range cases {
            cases[i].Dir = reflect.SelectSend
            cases[i].Chan = reflect.ValueOf(out[i])

        }

        for v := range ch {
            v := v
            for i := range cases {
                cases[i].Send = reflect.ValueOf(v)
            }
            _, _, _ = reflect.Select(cases)
        }
    }()
}
```

Channel

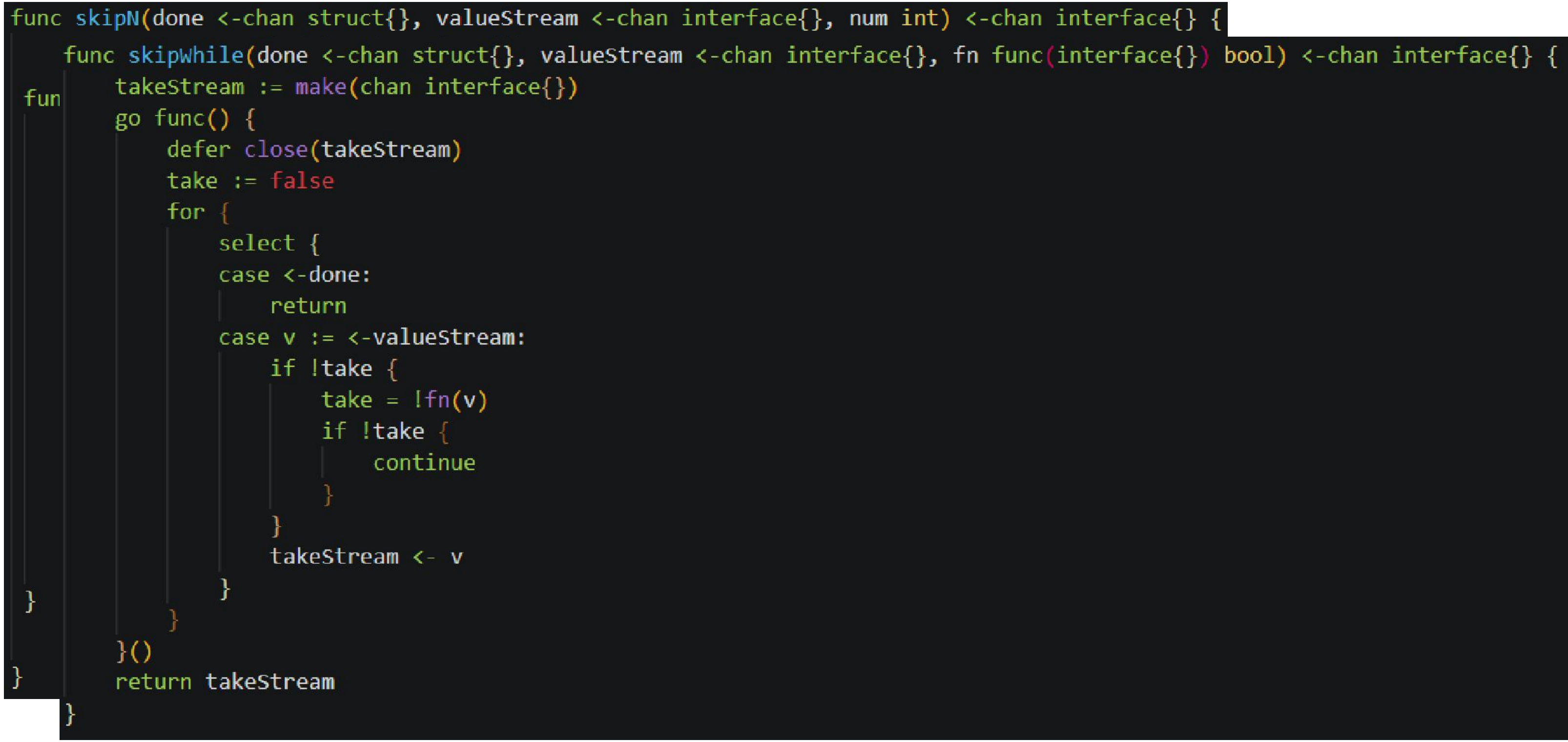
Pipeline



```go
func sq(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for n := range in {
            out <- n * n
        }
        close(out)
    }()
    return out
}

func main() {
    // Set up the pipeline.
    c := gen(2, 3)
    out := sq(c)

    // Consume the output.
    fmt.Println(<-out) // 4
    fmt.Println(<-out) // 9
}

func gen(nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        for _, n := range nums {
            out <- n
        }
        close(out)
    }()
    return out
}
```

Channel

Stream – Skip

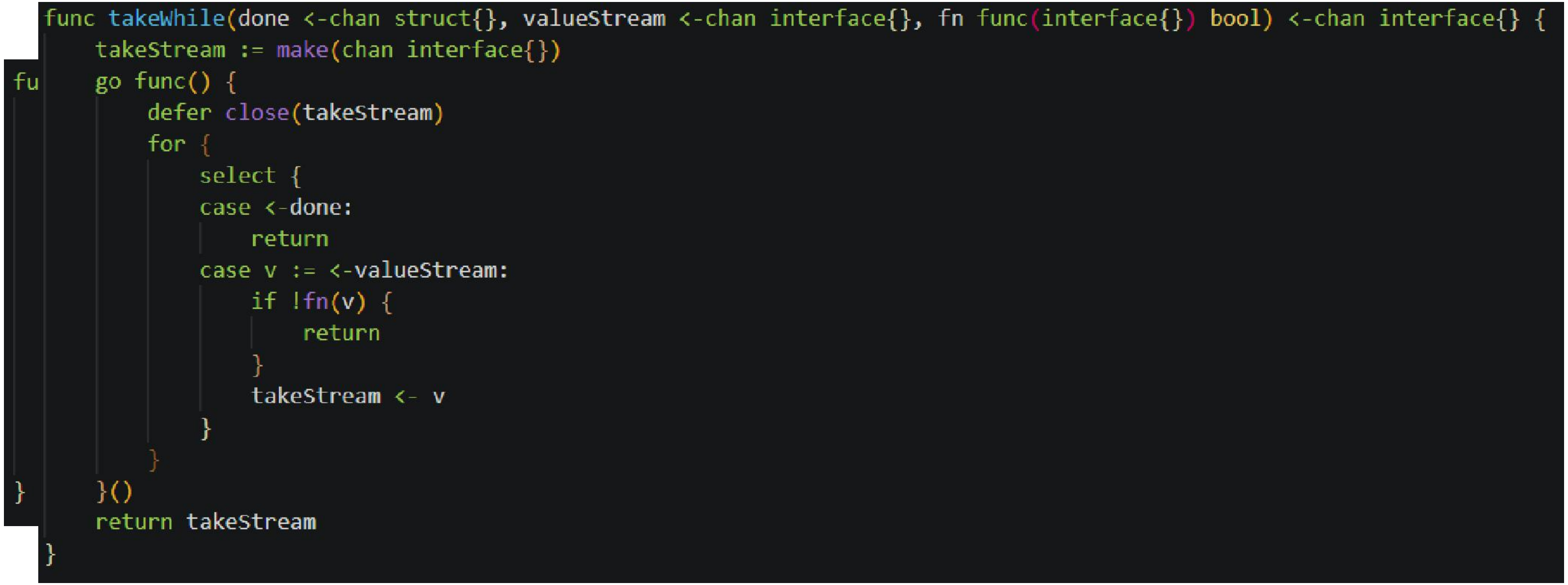


```go
func skipN(done <-chan struct{}, valueStream <-chan interface{}, num int) <-chan interface{} {
 fun
 }
}
```

```go
func skipWhile(done <-chan struct{}, valueStream <-chan interface{}, fn func(interface{}) bool) <-chan interface{} {
    takeStream := make(chan interface{})
    go func() {
        defer close(takeStream)
        take := false
        for {
            select {
            case <-done:
                return
            case v := <-valueStream:
                if !take {
                    take = !fn(v)
                    if !take {
                        continue
                    }
                }
                takeStream <- v
            }
        }
    }()
    return takeStream
}
```

Channel

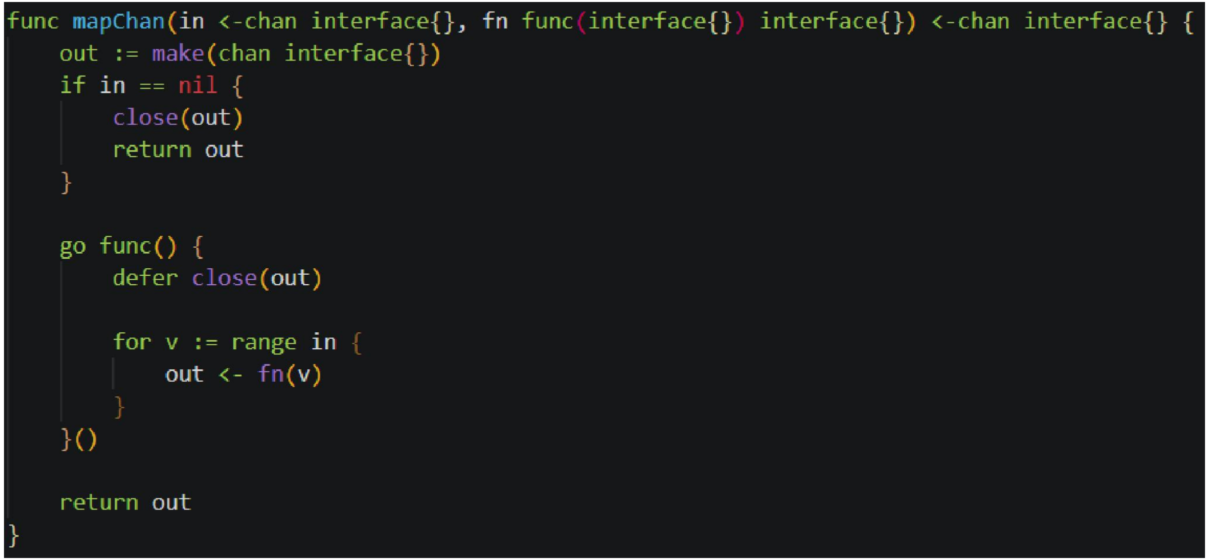
Stream – Take



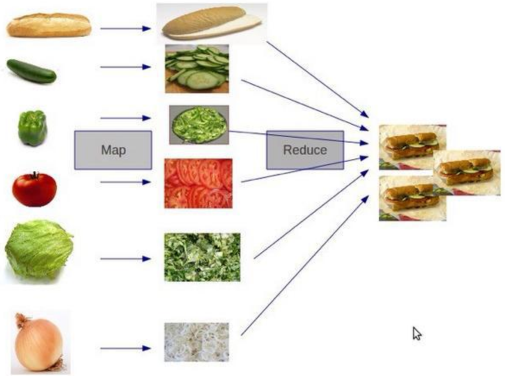
```go
func takeWhile(done <-chan struct{}, valueStream <-chan interface{}, fn func(interface{}) bool) <-chan interface{} {
    takeStream := make(chan interface{})
    go func() {
        defer close(takeStream)
        for {
            select {
            case <-done:
                return
            case v := <-valueStream:
                if !fn(v) {
                    return
                }
                takeStream <- v
            }
        }
    }()
    return takeStream
}
```

Channel

Stream – Map

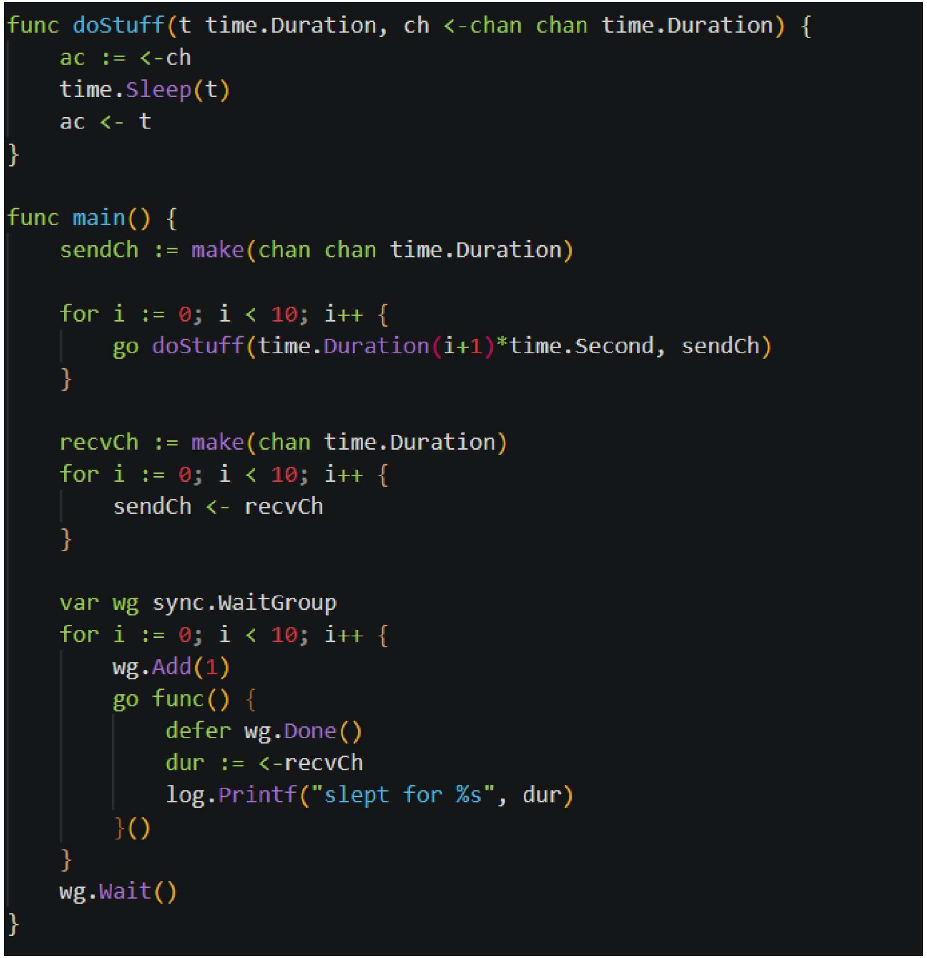


```go
func mapChan(in <-chan interface{}, fn func(interface{}) interface{}) <-chan interface{} {
    out := make(chan interface{})
    if in == nil {
        close(out)
        return out
    }

    go func() {
        defer close(out)

        for v := range in {
            out <- fn(v)
        }
    }()

    return out
}
```

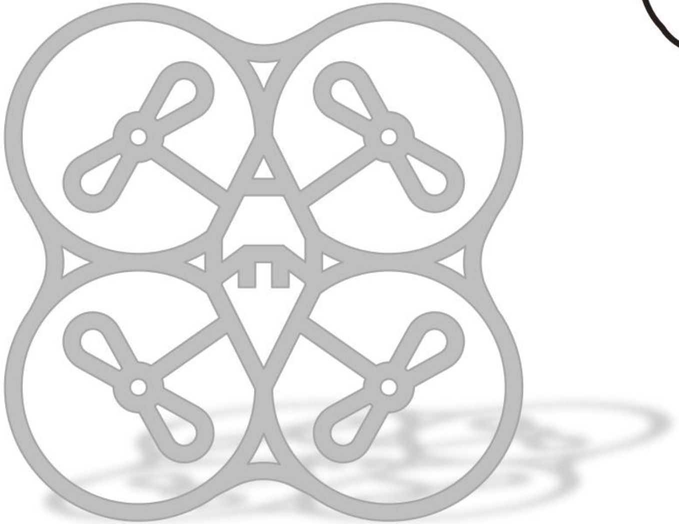


Channel

Channels over channel



```go
func doStuff(t time.Duration, ch <-chan chan time.Duration) {
    ac := <-ch
    time.Sleep(t)
    ac <- t
}

func main() {
    sendCh := make(chan chan time.Duration)

    for i := 0; i < 10; i++ {
        go doStuff(time.Duration(i+1)*time.Second, sendCh)
    }

    recvCh := make(chan time.Duration)
    for i := 0; i < 10; i++ {
        sendCh <- recvCh
    }

    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            dur := <-recvCh
            log.Printf("slept for %s", dur)
        }()
    }
    wg.Wait()
}
```



04

内存模型

- 单击此处添加小标题
- 单击此处添加小标题

内存模型

内存模型描述了线程(纤程)通过内存的交互，以及对数据的共享使用

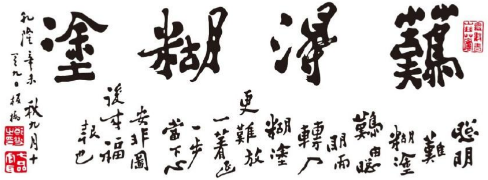
历史

Java Memory Model 第一个尝试定义内存模型的编程语言
WG21/N2429：Concurrency memory model (final revision),C11/C++11

Go 内存模型

定义了一个条件：对同一个变量，如何保证在一个goroutine对此变量读的时候，能观察到其它goroutine对此变量的写。

修改一个同时被多个goroutine并发访问的变量的时候，需要串行化访问。



Don't be clever

内存模型

Happens Before

历史

happens-before关系是指两个事件结果之间的关系。如果一个事件 happens before 另外一个事件，那么结果应该反映这一点，即使事件是无序执行的。

- a $\longrightarrow$ b： 同一个进程中事件a在事件b之前发生
- a $\longrightarrow$ b： 事件a发送消息，事件b接受这个消息

- transitive(传递性): $\forall a, b, c,$ if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$
- irreflexive (反自反性): $\forall a, a \nrightarrow a$
- antisymmetric(非对称性): $\forall a, b,$ where $a \neq b,$ if $a \rightarrow b$ then $b \nrightarrow a$

内存模型

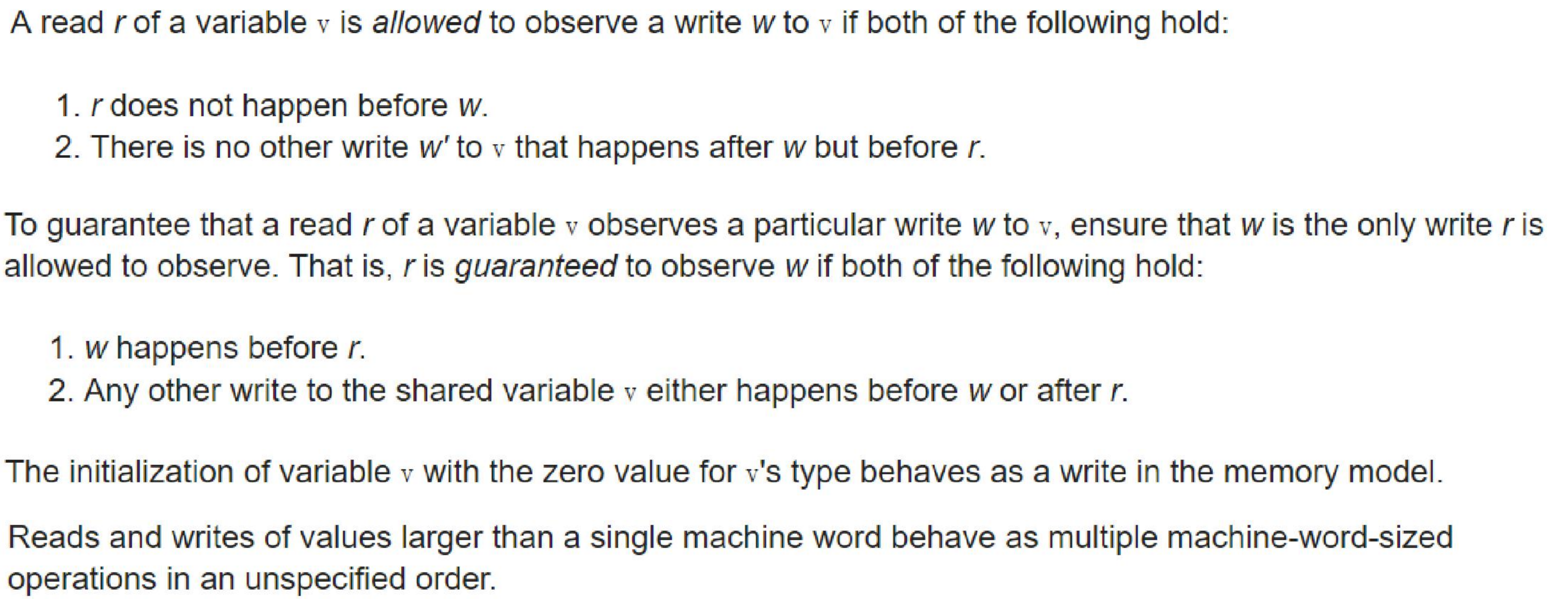
Happens Before

单个goroutine内

读写执行的顺序和程序定义顺序一致
乱序执行不影响程序的行为

内存模型

Happens Before



A read *r* of a variable v is *allowed* to observe a write *w* to v if both of the following hold:

1. *r* does not happen before *w*.
2. There is no other write *w'* to v that happens after *w* but before *r*.

To guarantee that a read *r* of a variable v observes a particular write *w* to v, ensure that *w* is the only write *r* is allowed to observe. That is, *r* is *guaranteed* to observe *w* if both of the following hold:

1. *w* happens before *r*.
2. Any other write to the shared variable v either happens before *w* or after *r*.

The initialization of variable v with the zero value for v's type behaves as a write in the memory model.

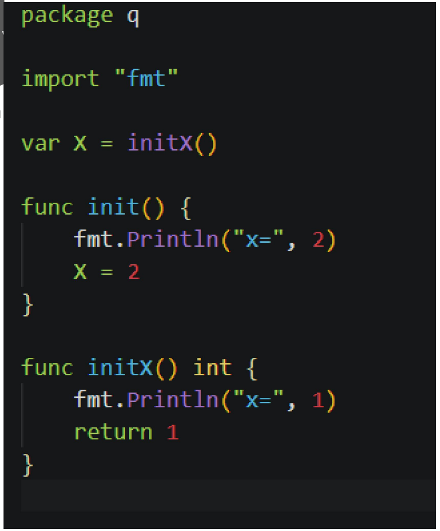
Reads and writes of values larger than a single machine word behave as multiple machine-word-sized operations in an unspecified order.

内存模型

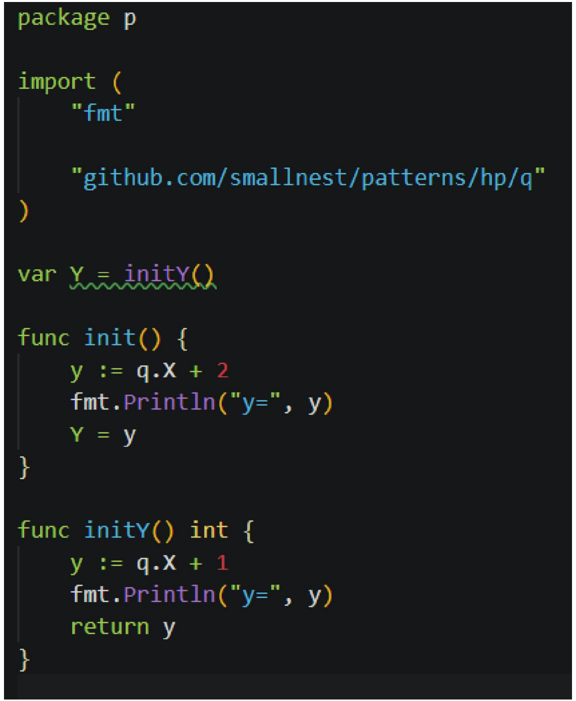
init函数

init的执行是在单个goroutine中执行的

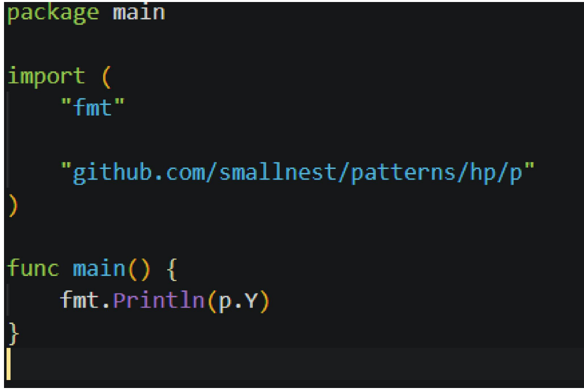
1. 如果package p 引入了 package q, 那么q的init函数一定 happens before p 的init之前。
2. main函数在所有引入的init函数执行



```go
package q

import "fmt"

var X = initX()

func init() {
    fmt.Println("x=", 2)
    X = 2
}

func initX() int {
    fmt.Println("x=", 1)
    return 1
}
```



```go
package p

import (
    "fmt"

    "github.com/smallnest/patterns/hp/q"
)

var Y = initY()

func init() {
    y := q.X + 2
    fmt.Println("y=", y)
    Y = y
}

func initY() int {
    y := q.X + 1
    fmt.Println("y=", y)
    return y
}
```

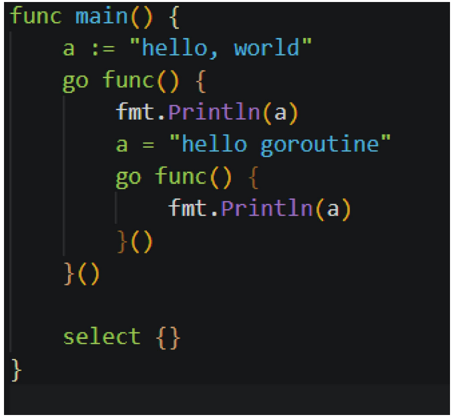


```go
package main

import (
    "fmt"

    "github.com/smallnest/patterns/hp/p"
)

func main() {
    fmt.Println(p.Y)
}
```

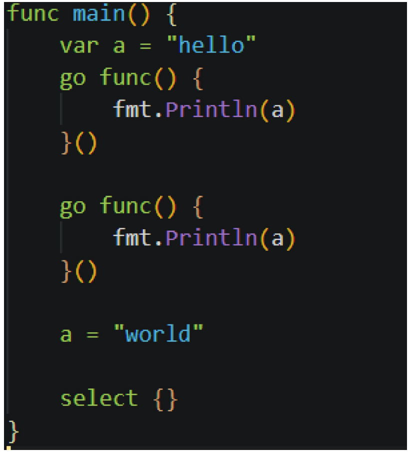
内存模型

goroutine

- goroutine的创建happens before所有此goroutine中的操作
- goroutine的销毁happens after所有此goroutine中的操作



```
func main() {
    a := "hello, world"
    go func() {
        fmt.Println(a)
        a = "hello goroutine"
        go func() {
            fmt.Println(a)
        }()
    }()

    select {}
}
```



```
func main() {
    var a = "hello"
    go func() {
        fmt.Println(a)
    }()

    go func() {
        fmt.Println(a)
    }()

    a = "world"

    select {}
}
```

内存模型

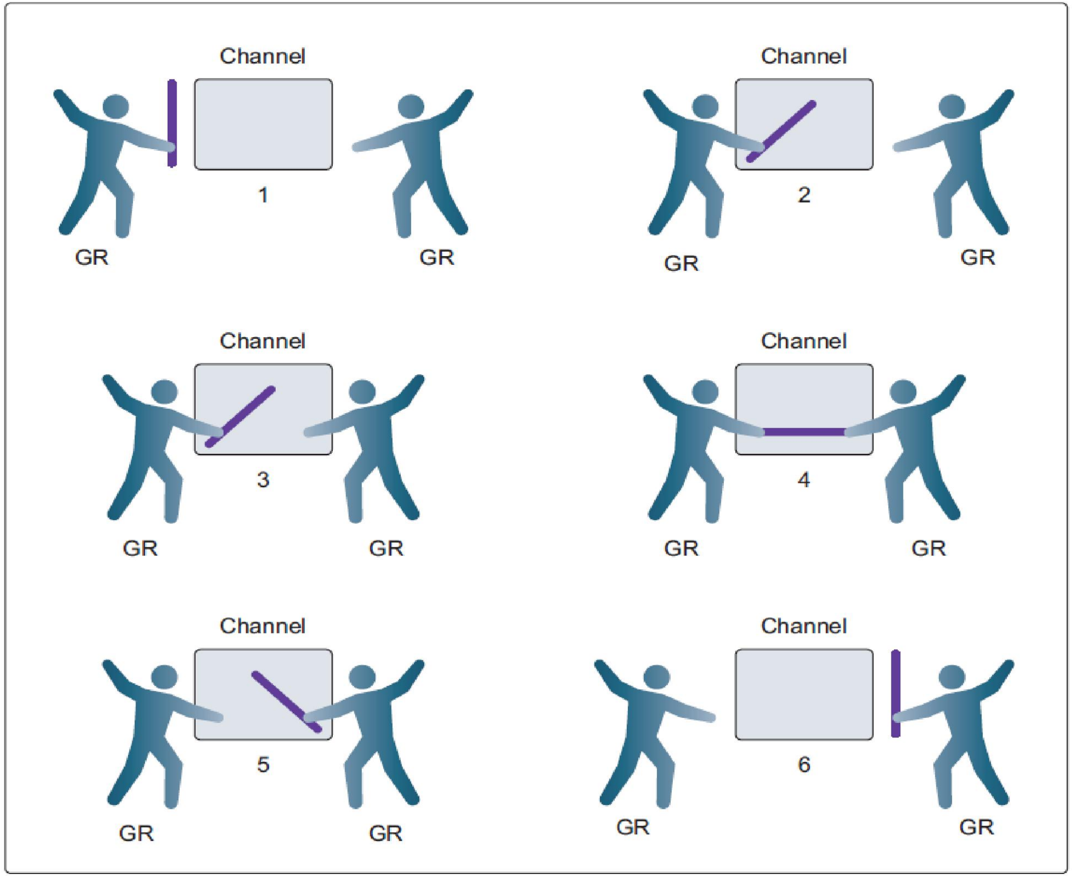
Channel

- 第**n**个send一定happen before第n个receive完成, 不管是buffered channel还是unbuffered channel
- 对于capacity 为**m**的channel,第n个receive一定happen before第**(n+m)** send完成
- **m=0** unbuffered。第n个receive一定happen before第n个send完成
- channel的close一定happen before receive端得到通知，得到通知意味着receive收到一个因为channel close而收到的零值

注意 send/send completes，receive/receive completes的区别

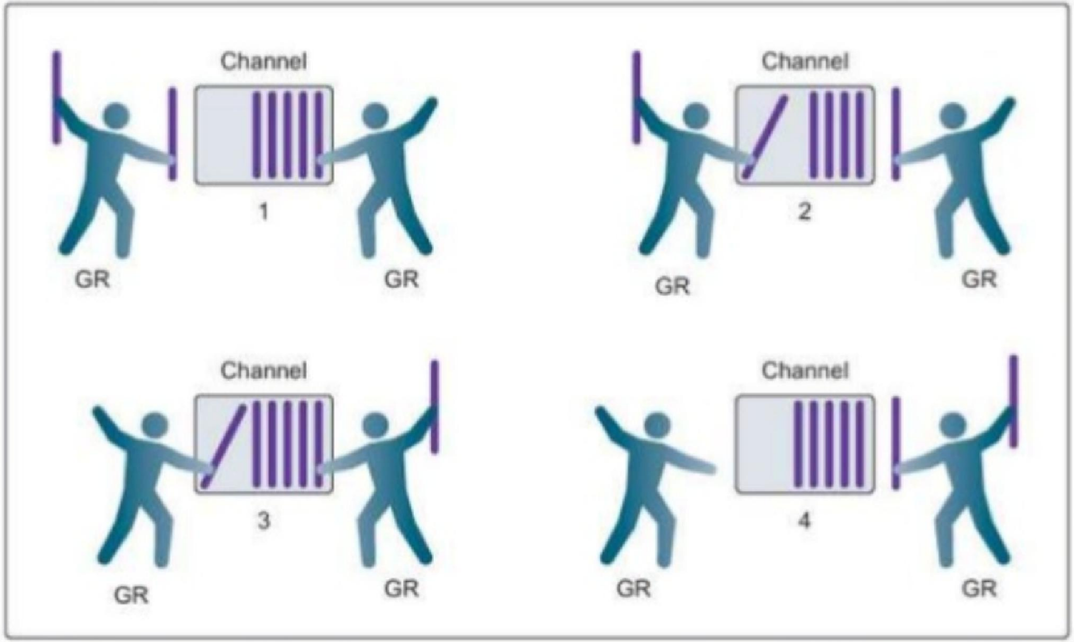
内存模型

Channel unbuffered



内存模型

Channel buffered



内存模型

Mutex/RWMutex

- 对于Mutex/RWMutx **m**, 第**n**个成功的 **m.Unlock** 一定happen before 第 **n+1 m.Lock**方法调用的返回
- 对于RWMutex **rw**，如果它的第**n**个**rw.Lock**已返回，那么它的第**n**个成功的**rw.Unlock**的方法调用一定happen before 任何一个**rw.RLock**方法调用的返回（它们 happen after 第**n**个**rw.Lock**方法调用返回）
- 对于RWMutex **rw**,如果它的第**n**个**rw.RLock**已返回，接着第**m (m < n)**个**rm.RUnlock**方法调用一定happen before 任意的 rw.Lock(它们 happen after 第**n**个**rw.RLock**方法调用返回之后)

内存模型

Waitgroup

- 对于 Waitgroup b, 对于其计数器不是0的时候，假如此时刻之后有一组wg.Add(n),并且我们确信只有最后一组方法调用使其计数器最后复原为0，那么这组wg.Add 方法调用一定happen before 这一时刻之后发生的wg.Wait
- wg.Done()也是wg.Add(-1)

内存模型

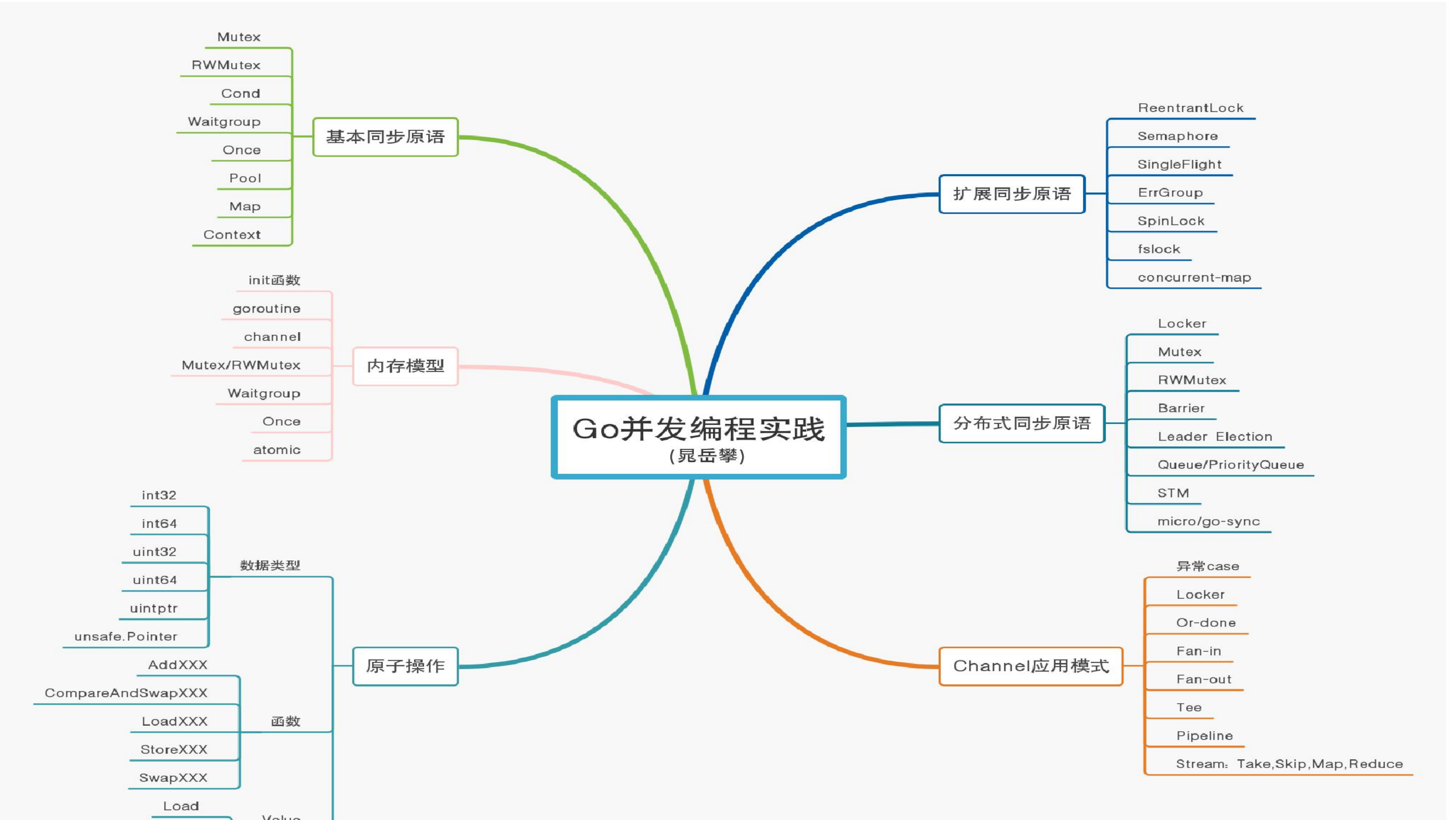
Once

- once.Do方法的执行一定happen before 任何一个once.Do方法的返回

内存模型

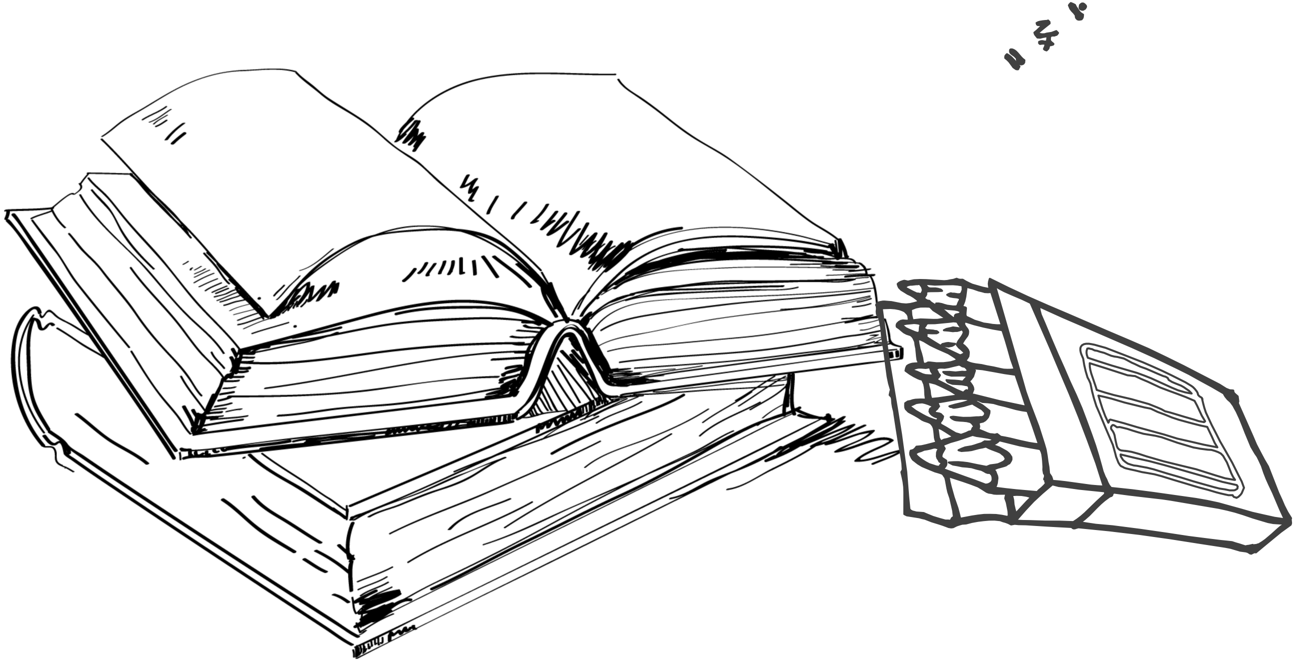
Atomic

- 没有官方的保证
- 建议是不要依赖atomic保证内存的顺序
- #5045 历史悠久的讨论，还没close



Go并发编程实践
(晁岳攀)

- 基本同步原语
  - Mutex
  - RWMutex
  - Cond
  - Waitgroup
  - Once
  - Pool
  - Map
  - Context
- 内存模型
  - init函数
  - goroutine
  - channel
  - Mutex/RWMutex
  - Waitgroup
  - Once
  - atomic
- 原子操作
  - 数据类型
    - int32
    - int64
    - uint32
    - uint64
    - uintptr
    - unsafe.Pointer
  - 函数
    - AddXXX
    - CompareAndSwapXXX
    - LoadXXX
    - StoreXXX
    - SwapXXX
  - Value
    - Load
- 扩展同步原语
  - ReentrantLock
  - Semaphore
  - SingleFlight
  - ErrGroup
  - SpinLock
  - fslock
  - concurrent-map
- 分布式同步原语
  - Locker
  - Mutex
  - RWMutex
  - Barrier
  - Leader Election
  - Queue/PriorityQueue
  - STM
  - micro/go-sync
- Channel应用模式
  - 异常case
  - Locker
  - Or-done
  - Fan-in
  - Fan-out
  - Tee
  - Pipeline
  - Stream：Take,Skip,Map,Reduce



The End