Go并发编程实践

晁岳攀

@colobu 微博
http://colobu.com

探探 Gopher China 2019

Agenda

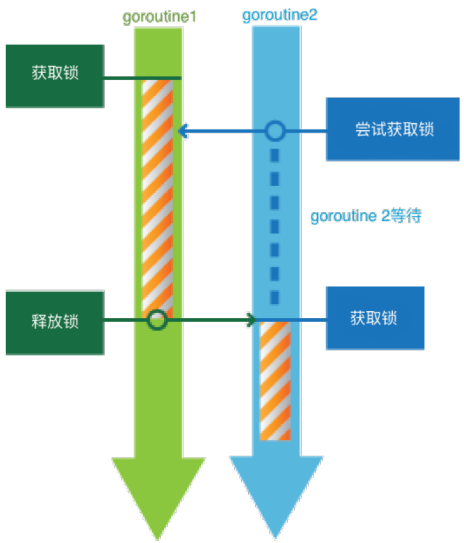
- 基本同步原语
- 扩展同步原语
- 原子操作
- Channel
- 内存模型

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
- 避免死锁
- 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



Mutex - 初版(2008)

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

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

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


Mutex

- 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

Mutex - 当前实现

```
type Mutex struct {
  state int32
  sema  uint32
}
const (
  mutexLocked = 1 << iota
  mutexWoken
  mutexStarving
  mutexWaiterShift = iota
)
```

```
func (m *Mutex) Lock() {
  if atomic.CompareAndSwapInt32(
  &m.state, 0, mutexLocked)
{    return
  }
  m.lockSlow()
}

func (m *Mutex) Unlock() {
  new := atomic.AddInt32(&m.state,
-mutexLocked)
  if new != 0 {
  m.unlockSlow(new)
  }
}
```


Mutex - 实现逻辑

- 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。

Mutex - TryLock, Count

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

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


Mutex - Iswoken, IsStarving

```
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
}
```


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

RWMutex - 数据结构

```
type RWMutex struct {
  w  Mutex  // for pending writers
  writerSem uint32 // semaphore for writers to wait for
completing readers
  readerSem uint32 // semaphore for readers to wait for
completing writers
  readerCount int32  // number of pending readers
  readerWait  int32  // number of departing readers
}
```


RWMutex - 递归的问题

```
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
}
```


RWMutex

```
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.Po
  v = v >> mutexWaiterShift
  v = v + (v & mutexLocked)
  return int(v)
}
```


Cond

- Mutex有些情况下不适用
- 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

Cond

func (*Cond) Broadcast

```
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(rand...)
    ready <- struct{}{}
    for!isReady {
      c.Wait()
    }
     m.Unlock()
   }()
}
```

func (*Cond) Signal

```
c.Broadcast() // false op
c.Signal() // false op

for i := 0; i < 10; i++ {
  <-ready
}
isReady = true
c.Broadcast()
```

func (*Cond) Wait

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……

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))
}
```


Waitgroup - 可重用

```
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))
```


Waitgroup - 多次Wait和多次Done

```
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(&cou
nt))
```

```
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() //多了一次Done
wg.Wait()

fmt.Println(atomic.LoadInt64(&cou
nt))
```


Waitgroup - Add和Wait并发调用

```
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()
    }
  }()
}
```

```
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
```


Waitgroup - Wait未完成就Add

```
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
```


Once

- 只执行一次初始化 `func (o *Once) Do(f func())`
- 避免死锁
- 即使f panic, Once也认为它完成了

Once

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

```
time.Sleep(time.Second)

once.Do(func() {
  fmt.Println("exec Do #2")
  count = 1 / count
})
```


Once - 单例

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

Once - 单例

```
type Once struct {
  m    Mutex
  done uint32
}
func (o *Once) Do(f func()) {
  if atomic.LoadUint32(&o.done) == 1 {
    return
  }
  // Slow-path.
  o.m.Lock()
  defer o.m.Unlock()
  if o.done == 0 {
    defer atomic.StoreUint32(&o.done, 1)
    f()
  }
}
```


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
- 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() {}
```


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

Pool

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)
}
```


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)
}
```


Pool

json包错误使用 go#27735

```
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)
}
```


Once

```
type Once struct {
  m    Mutex
  done uint32
}
func (o *Once) Do(f func()) {
  if atomic.LoadUint32(&o.done) == 1 {
    return
  }
  // Slow-path.
  o.m.Lock()
  defer o.m.Unlock()
  if o.done == 0 {
    defer atomic.StoreUint32(&o.done, 1)
    f()
  }
}
```


Map

- 两个cases
  - 设置一次，多次读
  - 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

扩展同步原语

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()
  atomic.StoreInt64(&m.owner, gid)
  m.recursion = 1
}
```

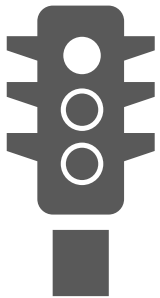
```
func (m *RecursiveMutex) Unlock()
{
 gid := goid.Get()
 if atomic.LoadInt64(&m.owner) !=
gid { panic(...)}
  m.recursion--
  if m.recursion != 0 {
    return
  }
  atomic.StoreInt64(&m.owner, -1)
  m.Mutex.Unlock()
}
```


ReentrantLock

```
type TRMutex struct {
  sync.Mutex
  token     int64
  recursion int32
}
func (m *TRMutex ) Lock(t int64)
{
  if atomic.LoadInt64(&m.token)
== t{
    m.recursion++
    return
  }
  m.Mutex.Lock()
  atomic.StoreInt64(&m.token, t)
  m.recursion = 1
}
```

```
func (m *TRMutex ) Unlock(t int64)
{
  if
atomic.LoadInt64(&m.token) != t{
    panic(...}

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

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

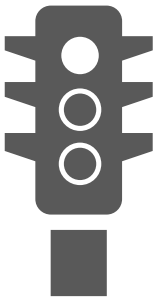

Semaphore



- Dijkstra提出并发访问通用资源的并发原语
- 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

Semaphore



golang.org/x/sync/semaphore

```
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)
    }
```


SingleFlight

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

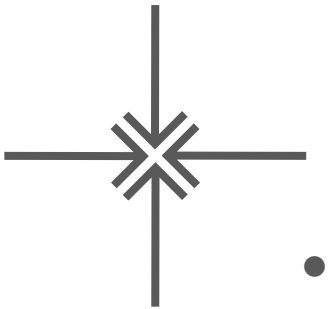
type Group
- func (g *Group) Do(key string, fn func() (interface{}, error)) (v interface{}, err error, shared bool)
- func (g *Group) DoChan(key string, fn func() (interface{}, error)) <-chan Result
- func (g *Group) Forget(key string)

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
```


ErrGroup

golang.org/x/sync/semaphore

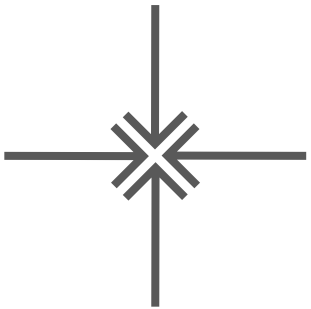


type Group
- func WithContext(ctx context.Context) (*Group, context.Context)
- func (g *Group) Go(f func() error)
- func (g *Group) Wait() error

- 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

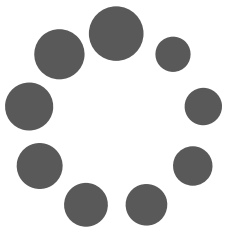
ErrGroup

golang.org/x/sync/semaphore



```
var g errgroup.Group
var urls = []string{
    "http://www.golang.org/",
    "http://www.google.com/",
    "http://www.somestupidname.com/",
}
for _, url := range urls {
    // Launch a goroutine to fetch the URL.
    url := url // https://golang.org/doc/faq#closures_and_goroutines
    g.Go(func() error {
        // Fetch the URL.
        resp, err := http.Get(url)
        if err == nil {
            resp.Body.Close()
        }
        return err
    })
}
// Wait for all HTTP fetches to complete.
if err := g.Wait(); err == nil {
    fmt.Println("Successfully fetched all URLs.")
}
```


SpinLock



- 自旋锁
- 有时候很高效，但是
- 公平性
- 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)
}
```


fslock

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

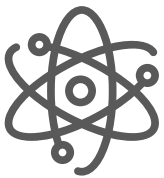
concurrent-map

github.com/orcaman/concurrent-map

```
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
}
```

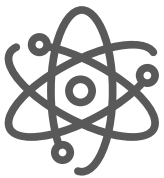

原子操作

atomic 数据类型



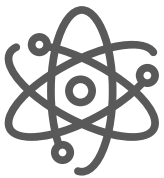
- int32
- int64
- uint32
- uint64
- uintptr
- unsafe.Pointer

atomic 方法



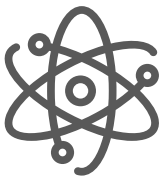
- AddXXX (整数类型)
- CompareAndSwapXXX：cas
- LoadXXX：读取
- StoreXXX：存储
- SwapXXX：交换

atomic 方法



- 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))

atomic



- Value

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

Bugs

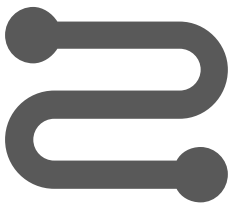
☞ On x86-32, the 64-bit functions use instructions unavailable before the Pentium MMX.

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.

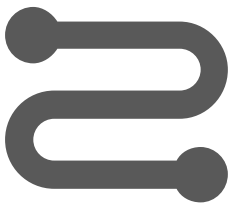
Channel

功能



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

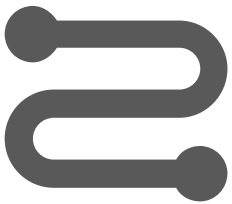
特殊情况



	nil	not empty	empty	full	not full
Receive	**block**	value	**block**	value	value
Send	**block**	write value	write value	**block**	write value
close	**panic**	closed, drained read, return zero value	closed, return zero value	closed, drained read, return zero value	closed, drained read, return zero value

Channel

Locker

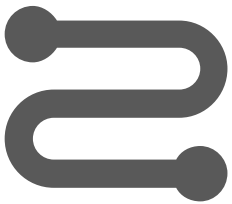


```
type Mutex struct {
  ch chan struct{}
}
func NewMutex() *Mutex {
  mu := &Mutex{make(chan struct{},
1)}
  mu.ch <- struct{}{}
  return mu
}
func (m *Mutex) Lock() {
  <-m.ch
}
func (m *Mutex) Unlock() {
  select {
  case m.ch <- struct{}{}:
  default:
    panic("unlock of unlocked mutex")
  }
}
```

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

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


Channel

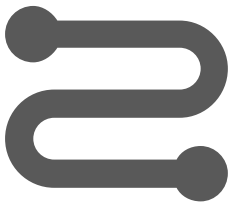
Locker



```
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
}

func (m *Mutex) IsLocked() bool {
  return len(m.ch) == 1
}
```


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

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

- Docker, Kubernetes, and gRPC, 171 concurrency bugs
- 58% by message passing

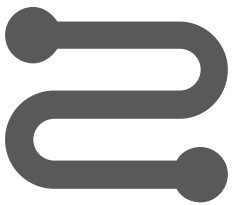
Application	Shared Memory					Message		Total
	Mutex	atomic	Once	WaitGroup	Cond	chan	Misc.	
Docker	62.62%	1.06%	4.75%	1.70%	0.99%	27.87%	0.99%	1410
Kubernetes	70.34%	1.21%	6.13%	2.68%	0.96%	18.48%	0.20%	3951
etcd	45.01%	0.63%	7.18%	3.95%	0.24%	42.99%	0	2075
CockroachDB	55.90%	0.49%	3.76%	8.57%	1.48%	28.23%	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.*

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

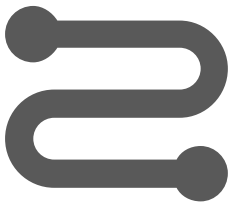
Table 5. Taxonomy. *This table shows how our studied bugs distribute across different categories and applications.*

Or-done



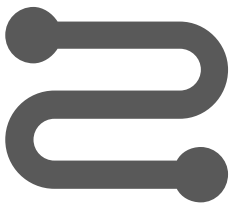
```
func orDone(done <-chan struct{}, c <-chan interface{}) <-chan
interface{} {
  outStream := make(chan interface{})
  go func() {
    defer close(outStream)
    for {
      select {
      case <-done: return
      case v, ok := <-c:
        if ok == false { return }
        select {
        case outStream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


Or - goroutine



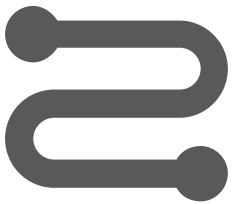
```
func or(chans ...<-chan interface{}) <-chan interface{} {
  orDone := make(chan interface{})
  go func() {
    var once sync.Once
    for _, c := range chans {
      go func(c <-chan interface{}) {
        select {
        case <-c:
          once.Do(func() { close(orDone) })
        case <-orDone: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


Or - 二分法递归



```
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:]...):
    }
  }
}()
```

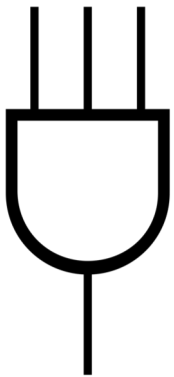

Or - 反射



```
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
}
```


Fan-in

- 递归

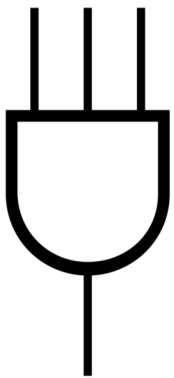


```
func fanInRec(chans ...<-chan
interface{}) <-chan interface{} {
  switch len(chans) {
  case 0:
    c := make(chan interface{})
    close(c)
    return c
  case 1:
    return chans[0]
  case 2:
    return mergeTwo(chans[0],
chans[1])
  default:
    m := len(chans) / 2
    return mergeTwo(
      fanInRec(chans[:m]...),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

```
func mergeTwo(a, b <-chan
interface{}) <-chan interface{} {
  c := make(chan interface{})
  go func() {
    defer close(c)
    for a != nil || b != nil {
      select {
      case v, ok := <-a:
        if !ok {
          a = nil
          continue
        }
        c <- v
      case v, ok := <-b:
        if !ok {
          b = nil
          continue
        }
        c <- v
    }
  }()
  return c
}
```

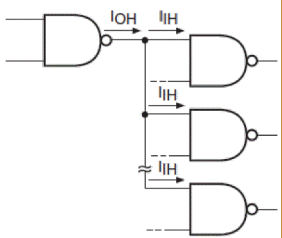

Fan-in

- 反射



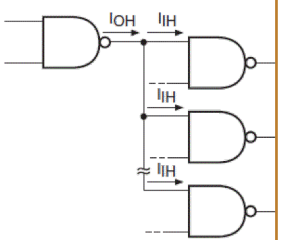
```
func fanInReflect(chans ...<-chan interface{}) <-chan interface{} {
  out := make(chan interface{})
  go func() {
    defer close(out)
    var cases []reflect.SelectCase
    for _, c := range chans {
      cases = append(cases, reflect.SelectCase{
        Dir:  reflect.SelectRecv,
        Chan: reflect.ValueOf(c),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


Fan-out



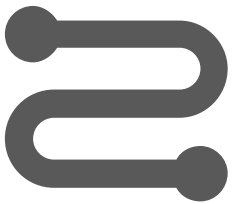

```
func fanOutReflect(ch <-chan interface{}, out []chan interface{}) {
  go func() {
    defer func() {
      for i := 0; i < len(out); i++ {
        close(out[i])
      }
    }()

    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)
    }
  }()
}
```


Tee



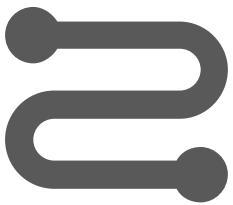

```
func fanOutReflect(ch <-chan interface{}, out []chan interface{}) {
  go func() {
    defer func() {
      for i := 0; i < len(out); i++ {
        close(out[i])
      }
    }()
    cases := make([]reflect.SelectCase, len(out))
    for i := range cases {
      cases[i].Dir = reflect.SelectSend
    }
    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)
      }
    }
  }()
}
```


Pipeline



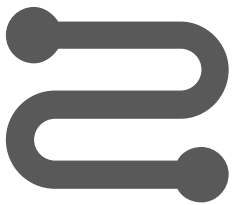
```
func sq(in <-chan int) <-chan int {
  out := make(chan int)
  go func() {
    for n := range in {
      out <- n * n
    }
    close(out)
  }()
  return out
}

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

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


Stream - Skip



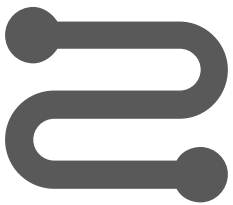
```
func skipN(done <-chan struct{}, valueStream <-chan interface{}, num int)
<-chan interface{} {
  takeStream := make(chan interface{})
  go func() {
    defer close(takeStream)
    for i := 0; i < num; i++ {
      select {
      case <-done: return
      case <-valueStream:
      }
    }
    for {
      select {
      case <-done: return
      case takeStream <- <-valueStream:
      }
    }
  }()

  return takeStream
}}
```


Stream - Take

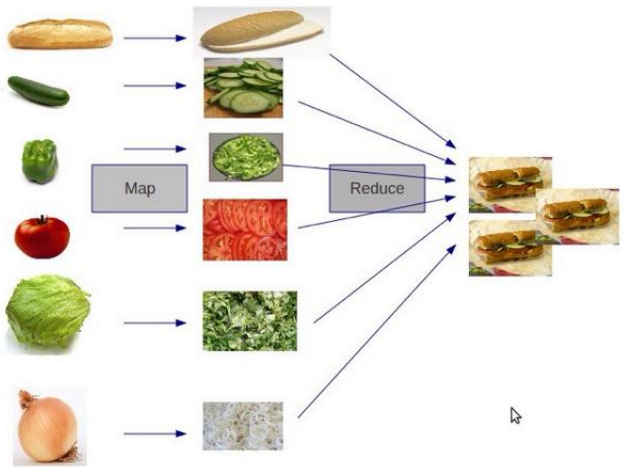


```
func takeN(done <-chan struct{}, valueStream <-chan interface{},
num int) <-chan interface{} {
  takeStream := make(chan interface{})
  go func() {
    defer close(takeStream)
    for i := 0; i < num; i++ {
      select {
      case <-done:
        return
      case takeStream <- <-valueStream:
      }
    }
  }()
  return takeStream
}
```

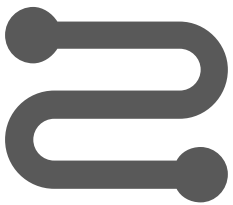

Stream - Map



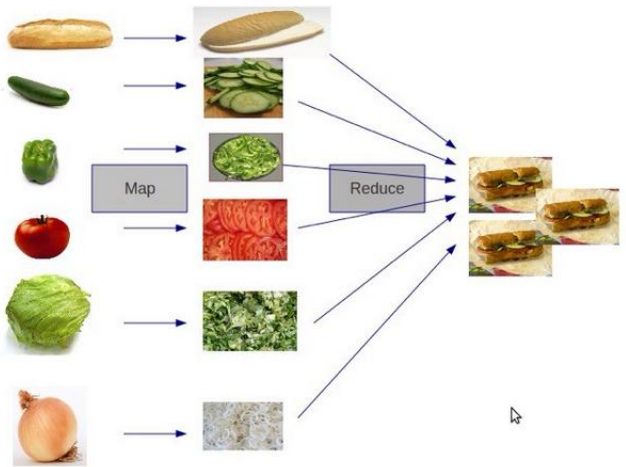
```
func mapChan(in <-chan interface{}, fn func(interface{})
interface{}) <-chan interface{} {
  out := make(chan interface{})
  if in == nil {
    close(out)
    return out
  }

  go func() {
    defer close(out)

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

  return out
}
```



Stream - Reduce



```
func reduce(in <-chan interface{}, fn func(r, v interface{})
interface{}) interface{} {
  if in == nil {
    return nil
  }

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

  return out
}
```



内存模型

内存模型

内存模型描述了线程(goroutine)通过内存的交互，以及对数据的共享使用

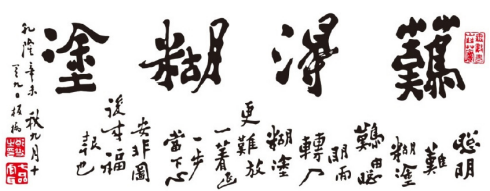
历史

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：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。
通过channel或者其它同步原语实现串行化访问。



Don't be clever

happen-before

Memory Order Guarantee

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

happen before

内存操作的偏序(partial order)。

定义了两个事件结果的先后顺序。

- **a** → **b**: **a** happens before **b** 或 **b** happens after **a**
- **a** does not happen before **b**, and does not happen after **b**, **a** 和 **b** 同时发生

happen-before

strict partial order

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

happen-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.

happen-before

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。main函数

```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)
}
```


happen-before

go 语句

- 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

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

    select {}
}
```

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

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

    a = "world"

    select {}
}
```


happen-before

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

happen-before

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)

happen-before

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)**

happen-before

Once

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

happen-before

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**

- 基本同步原语
- 扩展同步原语
- 原子操作
- Channel
- 内存模型