Programming in Go

Matt Holiday
Christmas 2020

Conventional Synchronization

Conventional synchronization

Package `sync`, for example:

- `Mutex`
- `Once`
- `Pool`
- `RWMutex`
- `WaitGroup`

Package `sync/atomic` for atomic scalar reads & writes

We saw a use of `WaitGroup` in the “file walk” example

Mutual exclusion

What if multiple goroutines must read & write some data?

We must make sure only **one** of them can do so at any instant
(in the so-called “critical section”)

We accomplish this with some type of lock:

- acquire the lock before accessing the data
- any other goroutine will **block** waiting to get the lock
- release the lock when done

Mutexes in action

```
type SafeMap struct {
    sync.Mutex          // not safe to copy
    m map[string]int
}

// so methods must take a pointer, not a value
func (s *SafeMap) Incr(key string) {
    s.Lock()
    defer s.Unlock()

    // only one goroutine can execute this
    // code at the same time, guaranteed
    s.m[key]++
}
```

Using `defer` is a good habit — avoid mistakes

RWMutexes in action

Sometimes we need to prefer readers to (infrequent) writers

```
type InfoClient struct {
    mu        sync.RWMutex
    token     string
    tokenTime time.Time
    TTL       time.Duration
}


func (i *InfoClient) CheckToken() (string, time.Duration) {
    i.mu.RLock()
    defer i.mu.RUnlock()

    return i.token, i.TTL - time.Since(i.tokenTime)
}
```

RWMutexes in action

```
func (i *InfoClient) ReplaceToken(ctx context.Context) (string, error) {
    token, ttl, err := i.getAccessToken(ctx)

    if err != nil {
        return "", err
    }

    i.mu.Lock()

    defer i.mu.Unlock()

    i.token = token
    i.tokenTime = time.Now()
    i.TTL = time.Duration(ttl) * time.Second

    return token, nil
}
```

Atomic primitives

```
func do() int {
    var n int64
    var w sync.WaitGroup

    for i := 0; i < 1000; i++ {
        w.Add(1)

        go func() {
            n++                          // DATA RACE
            w.Done()
        }()
    }

    w.Wait()
    return int(n)
}
```

Atomic primitives

```
func do() int {
    var n int64
    var w sync.WaitGroup

    for i := 0; i < 1000; i++ {
        w.Add(1)

        go func() {
            atomic.AddInt64(&n, 1)  // fixed
            w.Done()
        }()
    }

    w.Wait()
    return int(n)
}
```

Only-once execution

A `sync.Once` object allows us to ensure a function runs only once (only the first call to `Do` will call the function passed in)

```
var once sync.Once
var x    *singleton

func initialize() {
    x = NewSingleton()
}

func handle(w http.ResponseWriter, r *http.Request) {
    once.Do(initialize)
    . . .
}
```

Checking `x == nil` in the handler is **unsafe!**

Pool

A `Pool` provides for efficient & safe reuse of objects, but it's a container of `interface{}`

```
var bufPool = sync.Pool{
    New: func() interface{} {
        return new(bytes.Buffer)
    },
}

func Log(w io.Writer, key, val string) {
    b := bufPool.Get().(*bytes.Buffer)    // more reflection
    b.Reset()
    // write to it
    w.Write(b.Bytes())
    bufPool.Put(b)
}
```

Wait, there's more!

Other primitives:

- Condition variable
- Map (safe container; uses `interface{}`)
- WaitGroup