Concurrency: Lock, Group and Data Races

23 March 2020

Ákos Frohner

Overview

Part 1: Goroutines and Channels
Part 2: (this) Lock, WaitGroup and Data Races
Part 3: Context, Timeout, Rate Control

Examples are in the **ex?** subdirectories of
github.com/golang-elte-2020-public/concurrency (https://github.com/hu-univ-golang/golang-elte-2020-public/tree/master/concurrency)

Last exercise: parallelize checksum calculation

```
    // TODO: parallelize the checksum calculation
    for _, path := range Files() {
        hash, err := Hash(path)
        if err != nil {
            fmt.Printf("ERROR: %s\n", err)
            continue
        }
        fmt.Printf("%x\t%s\n", hash, path)
    }
```

Solution:

```
type result struct {
    path string
    hash []byte
    err  error
}
```

```
    files := Files()
    ch := make(chan *result, len(files))
    for _, path := range files {
        go func(path string) {
            hash, err := Hash(path)
            ch <- &result{path: path, hash: hash, err: err}
        }(path)
    }

    for range files {
        r := <-ch
        if r.err != nil {
            fmt.Printf("ERROR in %q: %s\n", r.path, r.err)
        } else {
            fmt.Printf("%x\t%s\n", r.hash, r.path)
        }
    }
```

How to...

- store the result?
- handle unknown number of inputs?
- handle errors?

Use case: Bazel

(re)building a project by tracking (file) dependencies

- run "go build ex3.go", when "ex3.go" changes
- change ~ different checksum

Continuously tracks and updates a path→hash map.

bazel.build/ (https://bazel.build/)

Simplified example

Simplified environment to focus on the language constructs:

```
func Hash(path string) int {
    time.Sleep(100 * time.Millisecond)
    return len(path) // *not* collision free hash
}
```

```
func Files() []string {
    return []string{"ex1/ex1.go", "ex2/cksum.go", "ex3/ex3.go"}
}
```

Exercise 3: collecting results

```
// TODO: parallelize the calculation
results := make(map[string]int)
for _, path := range Files() {
    results[path] = Hash(path)
}
```

play.golang.org/p/0yjPboEDb1k (https://play.golang.org/p/0yjPboEDb1k)

NOT a solution

```
// TODO: parallelize the calculation
ch := make(chan struct{})
results := make(map[string]int)
files := Files()
for _, path := range files {
    go func(path string) {
        results[path] = Hash(path)
        ch <- struct{}{}
    }(path)
}

for range files {
    <-ch
}
```

Data race detection

```
$ go run -race no.go
==================
WARNING: DATA RACE
Write at 0x00c000078150 by goroutine 9: ...
  main.main.func1()
      .../concurrency/ex3/no.go:15 +0x69

Previous write at 0x00c000078150 by goroutine 7: ...
  main.main.func1()
      .../concurrency/ex3/no.go:15 +0x69

Goroutine 9 (running) created at:
  main.main()
      .../concurrency/ex3/no.go:14 +0x13d

Goroutine 7 (running) created at:
  main.main()
      .../concurrency/ex3/no.go:14 +0x13d
==================
map[ex1/ex1.go:10 ex2/cksum.go:12 ex3/ex3.go:10]
Found 1 data race(s)
```

Mutex

Sometimes all you need is a lock:

- sync.Mutex: Lock, Unlock
- sync.RWMutex: Lock, Unlock, RLock, RUnlock

godoc.org/sync (https://godoc.org/sync)

Exercise 3…

Exercise 3: solution with Mutex

```go
// TODO: parallelize the calculation
ch := make(chan struct{})
mu := sync.Mutex{}
results := make(map[string]int)
files := Files()
for _, path := range files {
    go func(path string) {
        mu.Lock()
        results[path] = Hash(path)
        mu.Unlock()
        ch <- struct{}{}
    }(path)
}

for range files {
    <-ch
}
```

Exercise 3: solution with channel

```
// TODO: parallelize the calculation
type result struct {
    path string
    hash int
}
ch := make(chan *result)
files := Files()
for _, path := range files {
    go func(path string) {
        hash := Hash(path)
        ch <- &result{path, hash}
    }(path)
}

results := make(map[string]int)
for range files {
    r := <-ch
    results[r.path] = r.hash
}
```

How to...

- store the result ✓
- handle unknown number of inputs?
- handle errors?

Exercise 4: wait for all results

The results are collected in a shared map.

```
    // TODO: wait for all results
    results := make(map[string]int)
    mu := sync.Mutex{}
    for _, path := range Files() {
        go func(path string) {
            mu.Lock()
            defer mu.Unlock()
            results[path] = Hash(path)
        }(path)
    }
```

play.golang.org/p/8Z6ELbwcKkB (https://play.golang.org/p/8Z6ELbwcKkB)

Wait (group)

A thread-safe counter to wait until a group of goroutines finish.

```
var wg sync.WaitGroup
```

- wg.Add(x) to add x to the counter
- wg.Done() to decrement the counter
- wg.Wait() to block until the counter becomes zero

```
wg.Add(1)
go func() {
  defer wg.Done()
  ...
}()
wg.Wait()
```

Exercise 4...

Exercise 4: solution

```
    // TODO: wait for all results
    results := make(map[string]int)
    mu := sync.Mutex{}
    wg := sync.WaitGroup{}
    for _, path := range Files() {
        wg.Add(1)
        go func(path string) {
            defer wg.Done()
            mu.Lock()
            defer mu.Unlock()
            results[path] = Hash(path)
        }(path)
    }
    wg.Wait()
```

Range over channel revisited

Current code executes fix number of receives:

```
files := Files()
...
results := make(map[string]int)
for range files {
  r := <-ch
  results[r.path] = r.hash
}
```

We should read from the channel until it is closed:

```
ch := make(chan *result)
...
results := make(map[string]int)
for r := range ch {
  results[r.path] = r.hash
}
```

Exercise 5: close the channel!

Exercise 5: wait for last send

```
    // TODO: close the channel after the last send
    type result struct {
        path string
        hash int
    }
    ch := make(chan *result)
    for _, path := range Files() {
        go func(path string) {
            hash := Hash(path)
            ch <- &result{path, hash}
        }(path)
    }
    time.Sleep(200 * time.Millisecond) // to wait for all results
    close(ch)                          // that "range ch" works

    results := make(map[string]int)
    for r := range ch {
        results[r.path] = r.hash
    }
```

play.golang.org/p/GcZ3pPiBd5b (https://play.golang.org/p/GcZ3pPiBd5b)

Exercise 5...

Exercise 5: deadlock

```
ch := make(chan *result)
wg := sync.WaitGroup{}
for _, path := range Files() {
    wg.Add(1)
    go func(path string) {
        defer wg.Done()
        hash := Hash(path)
        ch <- &result{path, hash}
    }(path)
}
wg.Wait()
close(ch) // cannot close before last send

results := make(map[string]int)
for r := range ch {
    results[r.path] = r.hash
}
```

play.golang.org/p/iTSdC0h_tKi (https://play.golang.org/p/iTSdC0h_tKi)

Exercise 5: solution

```
ch := make(chan *result)
wg := sync.WaitGroup{}
for _, path := range Files() {
    wg.Add(1)
    go func(path string) {
        defer wg.Done()
        hash := Hash(path)
        ch <- &result{path, hash}
    }(path)
}
go func() {
    wg.Wait()
    close(ch)
}()

results := make(map[string]int)
for r := range ch {
    results[r.path] = r.hash
}
```

How to...

- store the result ✓
- handle unknown number of inputs ✓
- handle errors?

Hash may return an error

```go
func Hash(path string) (int, error) {
    time.Sleep(100 * time.Millisecond)
    if len(path) == 12 {
        return 0, fmt.Errorf("cannot calculate hash for %q", path)
    }
    return len(path), nil
}
```

Exercise 6: handle errors

Using a shared map for the results:

```
    // TODO: do not print results in case of any error
    results := make(map[string]int)
    mu := sync.Mutex{}
    wg := sync.WaitGroup{}
    for _, path := range Files() {
        wg.Add(1)
        go func(path string) {
            defer wg.Done()
            if hash, err := Hash(path); err == nil {
                mu.Lock()
                defer mu.Unlock()
                results[path] = hash
            } else {
                fmt.Printf("ERROR %s\n", err)
            }
        }(path)
    }
    wg.Wait()
    fmt.Println(results)
```

play.golang.org/p/LtfgGhZOS1F(https://play.golang.org/p/LtfgGhZOS1F)

WaitGroup with error handling

```
import "golang.org/x/sync/errgroup"
```

- sync.WaitGroup() → errgroup.WithContext(ctx)
- wg.Add(1),go,wg.Done() → eg.Go(func() error)
- wg.Wait() → err := eg.Wait()

```
eg, ectx := errgroup.WithContext(ctx)
eg.Go(func() error {
  ...
  return nil
})
err := eg.Wait()
```

godoc.org/golang.org/x/sync/errgroup (https://godoc.org/golang.org/x/sync/errgroup)

Exercise 6…

Exercise 6: solution

```go
// TODO: do not print results in case of any error
results := make(map[string]int)
mu := sync.Mutex{}
eg, _ := errgroup.WithContext(context.Background())
for _, path := range Files() {
    path := path
    eg.Go(func() error {
        hash, err := Hash(path)
        if err != nil {
            return err
        }
        mu.Lock()
        defer mu.Unlock()
        results[path] = hash
        return nil
    })
}
if err := eg.Wait(); err != nil {
    fmt.Printf("ERROR %s\n", err)
} else {
    fmt.Println(results)
}
```

How to...

- store the result ✓
- handle unknown number of inputs ✓
- handle errors ✓

Exercise 7: watching file changes

- periodically scan all files under a directory
- compare their checksum with the previous scan

Exercise 7: Hash() returns Hashed

```
type Hashed struct {
    Path string
    Hash []byte
    Err  error // Hash is invalid, in case of an error
}
```

```
func Hash(path string) *Hashed {
    f, err := os.Open(path)
    if err != nil {
        return &Hashed{Path: path, Err: err}
    }
    defer f.Close()

    h := sha1.New()
    if _, err := io.Copy(h, f); err != nil {
        return &Hashed{Path: path, Err: err}
    }
    return &Hashed{Path: path, Hash: h.Sum(nil)}
}
```

Exercise 7: HashedEqual()

```
func HashedEqual(before, after *Hashed) bool {
    if before == nil || after == nil {
        return before == nil && after == nil
    }
    if before.Path != after.Path {
        return false
    }
    if be, ae := before.Err != nil, after.Err != nil; be || ae {
        return be == ae
    }
    if len(before.Hash) != len(after.Hash) {
        return false
    }
    for i := 0; i < len(before.Hash); i++ {
        if before.Hash[i] != after.Hash[i] {
            return false
        }
    }
    return true
}
```

Exercise 7: CompareFilesets()

```
type FileSet map[string]*Hashed
```

```
func CompareFileSets(before, after FileSet) (added, edited, deleted []string) {
    for bp, bh := range before {
        switch ah, has := after[bp]; {
        case !has:
            deleted = append(deleted, bp)
        case !HashedEqual(bh, ah):
            edited = append(edited, bp)
        }
    }
    for ap := range after {
        if _, has := before[ap]; !has {
            added = append(added, ap)
        }
    }
    return added, edited, deleted
}
```

Exercise 7: periodically compare

```
func main() {
    prev := HashAll()
    for ts := range time.Tick(time.Second) {
        curr := HashAll()
        added, edited, deleted := CompareFileSets(prev, curr)
        prev = curr
        if len(added)+len(edited)+len(deleted) > 0 {
            fmt.Printf("files have changed at %v\n", ts)
            fmt.Printf("\tadded: %q\n", added)
            fmt.Printf("\tedited: %q\n", edited)
            fmt.Printf("\tdeleted: %q\n", deleted)
        }
    }
}
```

Exercise 7: checksum again

```
func HashAll() FileSet {
    // TODO: parallelize the checksum calculation
    results := make(FileSet)
    for _, path := range Files() {
        results[path] = Hash(path)
    }
    return results
}
```

- store the results in a shared map
- wait until all results arrive

Thank you

Ákos Frohner
szamcsi@gmail.com (mailto:szamcsi@gmail.com)