Concurrency: Context, Timeout and Rate Control

30 March 2020

Ákos Frohner

Overview

Part 1: Goroutines and Channels
Part 2: Lock, WaitGroup and Data Races
Part 3: (this) 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: watching file changes

```
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)
        }
    }
}
```

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

Exercise 7: original code

```
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

Exercise 7: solution

```
func HashAll() FileSet {
    // TODO: parallelize the checksum calculation
    results := make(FileSet)
    mu := sync.Mutex{}
    wg := sync.WaitGroup{}
    for _, path := range Files() {
        wg.Add(1)
        go func(path string) {
            defer wg.Done()
            hash := Hash(path)
            mu.Lock()
            defer mu.Unlock()
            results[path] = hash
        }(path)
    }
    wg.Wait()
    return results
}
```

How to...

- cancel unnecessary work?
- return with guaranteed latency?
- avoid overloading backends?

Goroutine without cancellation

```
    finished := make(chan struct{})
    go func() {
        time.Sleep(100 * time.Millisecond)
        finished <- struct{}{}
    }()
    <-finished
```

```
$ go run cancel_free.go
took  100.314093ms
```

Cancelling 1 goroutine

```
    finished, cancel := make(chan struct{}), make(chan struct{})
    go func() {
        select {
        case <-time.After(100 * time.Millisecond):
        case <-cancel:
        }
        finished <- struct{}{}
    }()
    go func() { cancel <- struct{}{} }()
    <-finished
```

```
$ go run cancel_1.go
took  19.174µs
```

Cancelling 2 goroutines

```
    finished, cancel := make(chan struct{}), make(chan struct{})
    f := func() {
        select {
        case <-time.After(100 * time.Millisecond):
        case <-cancel:
        }
        finished <- struct{}{}
    }
    go f()
    go f()
    go func() { cancel <- struct{}{} }()
    <-finished
    <-finished
```

```
$ go run cancel_2.go
took  100.373519ms
```

Cancelling 2 goroutines - explained

```
    finished, cancel := make(chan struct{}), make(chan struct{})
    f := func(i int) {
        select {
        case <-time.After(100 * time.Millisecond):
        case <-cancel:
            fmt.Println("cancelled")
        }
        fmt.Println(i, time.Since(start))
        finished <- struct{}{}
    }
    go f(1)
    go f(2)
    go func() { cancel <- struct{}{} }()
    <-finished
    <-finished
```

```
$ go run cancel_2_debug.go
cancelled
1 15.127µs
2 100.433792ms
took  100.586796ms
```

Cancelling 2 goroutines - fixed

Need to send a cancel signal to every goroutine:

```
    finished, cancel := make(chan struct{}), make(chan struct{})
    f := func(i int) {
        select {
        case <-time.After(100 * time.Millisecond):
        case <-cancel:
            fmt.Println("cancelled")
        }
        fmt.Println(i, time.Since(start))
        finished <- struct{}{}
    }
    go f(1)
    go f(2)
    go func() {
        cancel <- struct{}{}
        cancel <- struct{}{}
    }()
    <-finished
    <-finished
```

Cancelling n goroutines

```
finished, cancel := make(chan struct{}), make(chan struct{})
f := func(i int) {
    select {
    case <-time.After(100 * time.Millisecond):
    case <-cancel:
        fmt.Println("cancelled")
    }
    fmt.Println(i, time.Since(start))
    finished <- struct{}{}
}
for i := range work {
    go f(i)
}
go func() {
    for range work {
        cancel <- struct{}{}
    }
}()
for range work {
    <-finished
}
```

Goroutines may start other goroutines… how many?

Cancelling n goroutines - broadcast

close(chan) = broadcast to all blocked goroutines

```
finished, cancel := make(chan struct{}), make(chan struct{})
f := func(i int) {
    select {
    case <-time.After(100 * time.Millisecond):
    case <-cancel:
        fmt.Println("cancelled")
    }
    fmt.Println(i, time.Since(start))
    finished <- struct{}{}
}
for i := range work {
    go f(i)
}
go func() {
    close(cancel)
}()
for range work {
    <-finished
}
```

Context

```
ctx := context.Background()
```

Context for request scoped values across API boundaries:

- cancellation
- deadline or timeout
- authentication tokens
- authorization decisions

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

Context cancellation

```
ctx, cancel := context.WithCancel(ctx)
defer cancel()
```

- ctx.Done() - returns a channel that is closed by cancel()
- ctx.Err() - returns an error, once the channel is closed

```
select {
case <-ctx.Done():
  return ctx.Err()
}
```

Cancelling n goroutines - context

```
finished := make(chan struct{})
ctx, cancel := context.WithCancel(context.Background())
f := func(ctx context.Context, i int) {
    select {
    case <-time.After(100 * time.Millisecond):
    case <-ctx.Done():
        fmt.Println(ctx.Err())
    }
    fmt.Println(i, time.Since(start))
    finished <- struct{}{}
}
for i := range work {
    go f(ctx, i)
}
go func() {
    cancel()
}()
for range work {
    <-finished
}
```

Reminder: Hash() function called by errgroup

```
    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
        })
    }
```

Make use of the context returned here:

```
eg, ctx := errgroup.WithContext(context.Background())
```

Exercise 8: cancellable Hash()

```
$ go run ex8pre.go
ERROR cannot calculate hash for "ex2/cksum.go"
took  100.36225ms
```

Hash() keeps running, even when everything else is cancelled:

- stop, when other goroutines stop
- return "cancelled" as error

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

play.golang.org/p/9IcfRw-rORl (https://play.golang.org/p/9IcfRw-rORl)

Exercise 8...

Exercise 8: solution

```
    eg, ctx := errgroup.WithContext(context.Background())
    for _, path := range Files() {
        path := path
        eg.Go(func() error {
            hash, err := Hash(ctx, path)
            if err != nil {
                return err
            }
```

```
func Hash(ctx context.Context, path string) (int, error) {
    if len(path) == 12 {
        return 0, fmt.Errorf("cannot calculate hash for %q", path)
    }
    select {
    case <-time.After(100 * time.Millisecond):
    case <-ctx.Done():
        return 0, ctx.Err()
    }
    return len(path), nil
}
```

How to...

- cancel unnecessary work ✓
- return with guaranteed latency?
- avoid overloading backends?

Exercise 9: variable processing time

```
func translate(word string) string {
    time.Sleep(time.Duration(rand.Intn(200)) * time.Millisecond)
    return word
}
```

```
    eg, _ := errgroup.WithContext(context.Background())
    for _, word := range words {
        word := word
        eg.Go(func() error {
            translated := translate(word)
            mu.Lock()
            defer mu.Unlock()
            work[word] = translated
            return nil
        })
    }
    eg.Wait()
```

All translations **must** finish within 100ms.

Context with timeout

```
ctx, cancel := context.WithDeadline(ctx, time.Now().Add(100 * time.Millisecond))
defer cancel()
```

deadline = time.Now().Add(timeout)

```
ctx, cancel := context.WithTimeout(ctx, 100 * time.Millisecond)
defer cancel()
```

Combine this with the error group!

play.golang.org/p/t-cLLUpGXTm (https://play.golang.org/p/t-cLLUpGXTm)

Exercise 9…

Exercise 9: solution

```
func translate(ctx context.Context, word string) string {
    select {
    case <-time.After(time.Duration(rand.Intn(200)) * time.Millisecond):
    case <-ctx.Done():
        return ""
    }
    return word
}
```

```
    ctx, cancel := context.WithTimeout(ctx, 100*time.Millisecond)
    defer cancel()
    eg, ectx := errgroup.WithContext(ctx)
    for _, word := range words {
        word := word
        eg.Go(func() error {
            translated := translate(ectx, word)
            mu.Lock()
            defer mu.Unlock()
            work[word] = translated
            return nil
        })
    }
    eg.Wait()
```

How to reduce the error rate?

Running the solution 100x times:

- median latency=100.249466ms
- #error/#request=98/200

We can replace 'translate' by 'quickTranslate':

```
func quickTranslate(ctx context.Context, word string) string {
    ch := make(chan string, 3)
    go func() { ch <- translate(ctx, word) }()
    go func() { ch <- translate(ctx, word) }()
    go func() { ch <- translate(ctx, word) }()
    return <-ch
}
```

- median latency=73.480041ms
- #error/#request=25/200

Watch Jeff Dean on reducing tail lantecy (https://www.youtube.com/watch?v=1-3Ahy7Fxsc)

How to...

- cancel unnecessary work ✓
- return with guaranteed latency ✓
- avoid overloading backends?

Exercise 10: limited resource

```
type limited struct {
    mu    sync.Mutex
    limit int
    count int
    max   int
}
```

```
func (l *limited) start() {
    l.mu.Lock()
    defer l.mu.Unlock()
    l.count++
    l.check()
}
```

```
func (l *limited) end() {
    l.mu.Lock()
    defer l.mu.Unlock()
    l.count--
    l.check()
}
```

Exercise 10: overload check

```
func (l *limited) check() {
    if l.count-l.limit > l.max {
        l.max = l.count - l.limit
    }
}
```

```
func (l *limited) process(i int) {
    l.start()
    defer l.end()
    time.Sleep(100 * time.Millisecond)
}
```

Exercise 10: overload

```
    l := &limited{limit: 100}
    done := make(chan struct{})
    for _, i := range work {
        go func(i int) {
            l.process(i)
            done <- struct{}{}
        }(i)
    }
    for range work {
        <-done
    }
```

- avoid overload
- minimize the total processing time

```
$ go run limited.go
103.454188ms
maximum overload:  900
```

play.golang.org/p/vmZ3ZNifn_V (https://play.golang.org/p/vmZ3ZNifn_V)

Exercise 10...

Exercise 10: semaphore channel

```
l := &limited{limit: 100}
sem := make(chan struct{}, l.limit)
done := make(chan struct{})
for _, i := range work {
    sem <- struct{}{}
    go func(i int) {
        l.process(i)
        <-sem
        done <- struct{}{}
    }(i)
}
for range work {
    <-done
}
```

Exercise 10: worker pool

```
l := &limited{limit: 100}
todo := make(chan int)
done := make(chan struct{})
for n := 0; n < l.limit; n++ {
    go func() {
        for i := range todo {
            l.process(i)
            done <- struct{}{}
        }
    }()
}
go func() {
    for _, i := range work {
        todo <- i
    }
}()
for range work {
    <-done
}
```

How to...

- cancel unnecessary work ✓
- return with guaranteed latency ✓
- avoid overloading backends ✓

Exercise 11: rate limited watch

```
    // TODO: max 100 concurrent I/O
    results := make(FileSet)
    mu := sync.Mutex{}
    wg := sync.WaitGroup{}
    for _, path := range Files() {
        wg.Add(1)
        go func(path string) {
            defer wg.Done()
            hash := Hash(path)
            mu.Lock()
            defer mu.Unlock()
            results[path] = hash
        }(path)
    }
    wg.Wait()
```

Recommended reading/watching

Bryan C. Mills - Rethinking Classical Concurrency Patterns

drive.google.com/file/d/1nPdvhB0PutEJzdCq5ms6UI58dp50fcAN/view (https://drive.google.com/file/d/1nPdvhB0PutEJzdCq5ms6UI58dp50fcAN/view)

www.youtube.com/watch?v=5zXAHh5tJqQ (https://www.youtube.com/watch?v=5zXAHh5tJqQ)

Thank you

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