Parallel Programming

I/O and concurrency 2

Morgan Ericsson

Today

» Context

» Useful patterns

» HTTP servers

» REST APIs

Remember the API to compute area

```go
func CalcArea(w http.ResponseWriter, r *http.Request) {
    switch r.Method {
    case "POST":
        w.WriteHeader(http.StatusOK)
    case "GET":
        w.WriteHeader(http.StatusOK)
    default:
        w.WriteHeader(http.StatusNotFound)
    }
}
```

Remember the API to compute area

```go
case "POST":
    var dims Dimensions
    data, _ := io.ReadAll(r.Body)

    if err := json.Unmarshal(data, &dims); err == nil {
        area := Area{dims.Height * dims.Width}
        if res, err := json.Marshal(area); err == nil {
            w.Write(res)
        }
    }
```

And the Python client

```python
import requests
import json

vals = {'height':10, 'width':20}
url = f'http://localhost:3000/area?height={vals["height"]}&width={vals["width"]}'
r = requests.get(url)
if r.ok:
    jo = r.json()
    print(f'The area is {jo["area"]}')
```

Wrap in function

- Assume we want to port the client to Go
- And wrap it inside a function, e.g.,
  - `func CalcArea(w, h int) (int, error) {}`
- Exposes many issues/new ideas!

Simple client

```go
url := "http://localhost:3000/area"
body := []byte(`{"height":30, "width":20}`)

resp, err := http.Post(url, "application/json", bytes.NewBuffer(body))
if err != nil {
    log.Fatal("Request failed:", err)
}
defer resp.Body.Close()

if resp.StatusCode != http.StatusOK {
    log.Fatal(resp.Status)
}
```

Simple client

```go
data, err := io.ReadAll(resp.Body)
if err != nil {
    log.Fatal("Error reading response")
}

// We should use the Area type and unmarshal to that, but...
var result map[string]int
json.Unmarshal(data, &result)
fmt.Println("The area is", result["area"])
```

Wrapping in a function

- How should we handle errors?
  - retries, timeouts, ...
- How should we act responsibly?
  - and not kill the server
- Blocking or non-blocking?

Context

- A context allows for “per request” data and functions
- An easy way to cancel requests
  - time-out / deadline
- Parent and child contexts
  - Values in tree
  - Cancels downwards

Context

```go
func FunWithContext(ctx context.Context) {
	// Do something with the context
}
```

Values

```go
func FunWithContext(ctx context.Context) {
    if val := ctx.Value("mydata"); val != nil {
        // The value is of type any, so
        // we need a type assertion
        fmt.Println("Value:", val.(string))
    }
}

func main() {
    c := context.Background()
    FunWithContext(c)
}
```

Values

```go
func FunWithContext(ctx context.Context) {
    if val := ctx.Value("mydata"); val != nil {
        fmt.Println("Value:", val.(string))
    }
}

func main() {
    c := context.WithValue(context.Background(), "mydata", "abc123")

    FunWithContext(c)
}
```

Values

```go
func FunWithContext(ctx context.Context) {
    if val := ctx.Value("more"); val != nil {
        fmt.Println("Value:", val.(int))
    }
}

func main() {
    c := context.WithValue(context.Background(), "mydata", "abc123")
    c = context.WithValue(c, "more", 123)
    FunWithContext(cc)
}
```

Remember the done channel

```go
func doSmth(done <-chan interface{}) {
    for {
        select {
        case <-done:
            return
        default:
        }

        // Do some work
    }
}
```

Remember the done channel

- Same functionality in the context
  - either explicit or timed cancel

Same but context

```go
func doSmth(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            return
        default:
        }

        // Do some work
    }
}
```

Same but context

```go
func main() {
    c, cancel := context.WithCancel(context.Background())
    go doSmth(c)
    cancel()
}
```

Timeout

```go
func main() {
    var wg sync.WaitGroup
    c, cancel := context.WithTimeout(context.Background(), 2*time.Second)

    wg.Add(1)
    go func() {
        defer wg.Done()
        defer cancel() // Should cancel if no timeout to cleanup
        doSmth(c)
    }()

    wg.Wait()
    fmt.Println("Done!")
}
```

Deadline

- Similar to Timeout, but absolute time rather than offset
  - `WithTimeout(..., 2 * time.Second)`
  - `WithDeadline(..., time.Now().Add(2 * time.Second))`

Parents and children

- All the childrens’ done channels are closed when the parent’s done channel is closed
  - so, parent with shorter timeout/deadline will cancel children before their timeout/deadline
- Cancelling a child does not cancel the parent

Example

```go
func main() {
    c1, cf1 := context.WithCancel(context.Background())
    c2, cf2 := context.WithCancel(c1)
    c3, cf3 := context.WithCancel(c2)

    // cf3() cancels things called with c3
    // cf2() cancels things called with c3 and c2
    // cf1() cancels things called with c3, c2, and c1
}
```

Example

```go
func main() {
    var wg sync.WaitGroup
    c, _ := context.WithTimeout(context.Background(), 2*time.Second)
    cc, cancel2 :=  context.WithTimeout(c, 5*time.Second)

    wg.Add(1)
    go func() {
        defer wg.Done()
        defer cancel2()
        doSmth(cc)
    }()

    wg.Wait()
    fmt.Println("Done!")
}
```

Remember SimLatency

```go
func simLatency(w http.ResponseWriter, r *http.Request) {
    time.Sleep(15*time.Second)
    io.WriteString(w, "ok")
}

func main() {
    // ...

    http.HandleFunc("/latency", simLatency)
    http.ListenAndServe(":3000", nil)
}
```

Can we get the client to time out?

- Yes, contexts!
- But we need to pass a context with the request
- `NewRequestWithContext()`
- We will do it in two steps,
  - first, change to request and client
  - then, add timeout

Previous version

```go
url := "http://localhost:3000/simlatency"

resp, err := http.Get(url)
if err != nil {
    log.Fatal("Request failed:", err)
}
defer resp.Body.Close()

if resp.StatusCode != http.StatusOK {
    log.Fatal(resp.Status)
}
```

Request and client

```go
url := "http://localhost:3000/latency"

req, err := http.NewRequest("GET", url, nil)
if err != nil {
    log.Fatal("Error creating the request")
}

client := &http.Client{}
resp, err := client.Do(req)
if err != nil {
    log.Fatal("Request failed")
}

// ...
```

With timeout

```go
url := "http://localhost:3000/latency"
c, cancel := context.WithTimeout(context.Background(), 5*time.Second)

req, err := http.NewRequestWithContext(c, "GET", url, nil)
if err != nil {
    if errors.Is(err, context.DeadlineExceeded) {
        log.Fatal("Timeout")
    } else {
        log.Fatal("Error creating the request")
    }
}

client := &http.Client{}
resp, err := client.Do(req)
if err != nil {
    log.Fatal("Request failed")
}

// ...
```

We can of course cancel...

```go
// ...
go func() {
    defer cancel()
    time.Sleep(2 * time.Second)
}()

client := &http.Client{}
resp, err := client.Do(req)
if err != nil {
    switch {
    case errors.Is(err, context.DeadlineExceeded):
        log.Fatal("Timeout")
    case errors.Is(err, context.Canceled):
        log.Fatal("Canceled")
    default:
        log.Fatal(err)
    }
}
```

Back to the CalcArea function

```go
func CalcArea(w, h int) (int, error) {
    // do the request
    // extract and return the area
    // or a sane error
}

func main() {
    if area, ok := CalcArea(20, 30); ok == nil {
        fmt.Printf("The area is %d", area)
    }
}
```

Wait...

- Did we actually fix anything?
- Remember,
  - I/O is slow
  - blocking can be bad
- New CalcArea blocks waiting for I/O
- We know how to fix that!

Adding channels

```go
func main() {
    ch := make(chan int)
    go func() {
        if area, ok := CalcArea(20, 30); ok == nil {
            ch <- area
        }
        close(ch)
    }()

    if a, ok := <-ch; ok {
        fmt.Printf("The area is %d", a)
    }
}
```

Adding channels

- We love channels
- But they can be annoying
- And a bit verbose

Async python

```python
async def main():
  pl = {'height':20, 'width':30}
  url = 'http://localhost:3000/area'
  async with ClientSession() as session:
    async with session.post(url, json=pl) as resp:
      # Do stuff!
      res = await resp.json()
      print("The area is", res["area"])

asyncio.run(main())
```

Emulate in Go?

```go
type Future[T any] interface {
    Result() (T, error)
}
```

- We define a type for futures
- Contains a single method that blocks waiting for the result (and error)
- Similar to futures in Python or Java

InnerFuture

```go
type InnerFuture[T any] struct {
	once sync.Once
	wg sync.WaitGroup

	res T
	err error
	resCh <-chan T
	errCh <-chan error
}
```

Result

```go
func (f *InnerFuture[T]) Result() (T, error) {
    f.once.Do(func() {
        f.wg.Add(1)
        defer f.wg.Done()
        f.res = <-f.resCh
        f.err = <-f.errCh
    })

    f.wg.Wait()

    return f.res, f.err
}
```

- The blocking result is implemented using channels
- Can be called multiple times, only waits the first time (`once.Do`)

New `CalcArea`

```go
func CalcArea(h, w int) Future[int] {
    resCh := make(chan int)
    errCh := make(chan error)

    go func() {
        // Do all the stuff

        resCh <- // res here (if any)
        errCh <- // error here (if any)
    }()

    return &InnerFuture[int]{resCh: resCh, errCh: errCh}
}
```

- The new `CalcArea` returns an `InnerFuture`
- That we can “wait for” by calling result

Calling

```go
future := CalcArea(20, 30)
// do stuff

if res, err := future.Result(); res == nil {
    fmt.Printf("The area is %d", res)
}
```

Retries

- What should we do if there is a failure?
- Try multiple times? In a loop?
- Probably not a great idea
    - Some errors will probably not fix themselves
    - And if they will, maybe not in fractions of seconds

Retries

```go
type WorkFn func(context.Context) (string, error)

func Retry(fn WorkFn, retries int, delay time.Duration)
```

Retries

```go
func Retry(fn WorkFn, retries int, delay time.Duration) WorkFn {
    return func(ctx context.Context) (string, error) {
        for r := 0; ; r++ {
            resp, err := fn(ctx)
            if err == nil || r >= retries {
                return resp, err
            }

            select {
            case <-time.After(delay):
            case <-ctx.Done():
                return "", ctx.Err()
            }
        }
    }
}
```

Retries

```
r := Retry(MyWorker, 5, 2*time.Second)
res, err := r(context.Background())
```

Throttle

- We should not hammer the server
- There is often an acceptable rate for calls
    - Calls per time window
- Can figured out via headers and error 429 (too many requests)

Throttle

- Can be implemented in different ways
  - E.g., sleep between calls
- We will use token bucket

Token bucket

- Bucket contains tokens
- To make a call, we take a token from the bucket
- The bucket has a capacity
- Tokens are refilled over time

The refiller

```
refiller := func() {
    ticker := time.NewTicker(d)
    go func() {
        defer ticker.Stop()
        for {
            select {
            case <-ctx.Done():
                return
            case <-ticker.C:
                t := tokens + refill
                if t > max { t = max }
                tokens = t
            }
        }
    }
}
```

Throttle

```go
func Throttle(fn WorkFn, max int, refill int, d time.Duration) WorkFn {
    var tokens = max
    var once sync.Once

    return func(ctx context.Context) (string, error) {
        if ctx.Err() != nil {
            return "", ctx.Err()
        }

        once.Do(refiller)

        if tokens <= 0 {
            return "", errors.New("too many calls")
        }
        tokens--
        return fn(ctx)
    }
}
```

Back to the server

- We can use many of the tricks discussed on the server as well
- In some cases, a bit more complicated
- For example, we need to create, populate, propagate the context

Back to the server

```go
func simLatency(w http.ResponseWriter, r *http.Request) {
    time.Sleep(15*time.Second)
    io.WriteString(w, "ok")
}

func main() {
    // ...

    http.HandleFunc("/latency", simLatency)
    http.ListenAndServe(":3000", nil)
}
```

Back to the server

```go
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

- `http.Handler` is an interface
- A `Server` has a handler that it can call

Back to the server

```go
func main() {
    mux := http.NewServeMux()

    mux.HandleFunc("/latency", simLatency)
    srv := &http.Server{Handler:mux, Addr:"0.0.0.0:3000"}
    srv.ListenAndServe()
}
```

Context 1

- `Server` has a field, `BaseContext`
- The base context is a function that is used to create the context for each request
- If nil, `context.Background()` is used

Stupid example

```go
ctx, _ := context.WithCancel(context.Background())
srv := &http.Server{
    Handler:mux,
    Addr:"0.0.0.0:3000",
    BaseContext: func(l net.Listener) context.Context {
        ctx = context.WithValue(ctx, "SleepTime", 15)
        return ctx
    },
}
```

Stupid example

```go
func simLatency(w http.ResponseWriter, r *http.Request) {
	ctx := r.Context()
	st := ctx.Value("SleepTime").(int))
	time.Sleep(st*time.Second)
	io.WriteString(w, "ok")
}
```

Mux and ServeHTTP

- Since `ServeMux` can be a handler, it must have a `ServeHTTP` function
- This method basically finds the right handler for the route and calls ServeHTTP on that handler
- The handler is specified via the HandleFunc function

Really?

```go
func simLatency(w http.ResponseWriter, r *http.Request) {
    time.Sleep(15*time.Second)
    io.WriteString(w, "ok")
}
```

- » Where is the `ServeHTTP` function?

Really

```go
type HandlerFunc func(ResponseWriter, *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

Chaining handler functions

```go
func mw1(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        ctx := r.Context()
        ctx = context.WithValue(ctx, "turtles", true)
        next.ServeHTTP(w, r.WithContext(ctx))
    })
}
```

Chaining handler functions

```go
mux.Handle("/latency", mw1(http.HandlerFunc(simLatency)))
```

Chaining handler functions

```go
func simLatency(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()
    turtles := ctx.Value("turtles").(bool)
    if turtles {
        fmt.Println("All the way down...")
    }
    time.Sleep(15*time.Second)
    io.WriteString(w, "ok")
}
```

So, what do we do with the context?

- Keep information that is specific to the request
- Should not “live” before or after the request
- Usually used to “decorate” the request
  - Checking authentication
  - Extracting useful values
  - …

An example

A key-value store

- Assume we want to create a server that allows us to store and retrieve values
- We use a REST-style interface
- A `/kvs/<value>` end-point, where `<value>` is the key we want to do something with
  - PUT to set/update
  - GET to retrieve
  - DELETE to remove

Helper functions

- We assume that there are methods Put, Get, and Delete, that operate on the unlying data structure
- This allows us to modify how the data is stored without changing the handlers

Helper functions

```go
var kvs = make(map[string]string)

var ErrorNoSuchKey = errors.New("No such key")

func Get(key string) (string, error) {
    if value, ok := kvs[key]; ok {
        return value, nil
    } else {
        return "", ErrorNoSuchKey
    }
}
```

Handlers

- We use the endpoint `/kvs/` followed by the name of the key
- How can we extract the key name from the URL in the request?
- Go 1.22 adds new ways to define endpoints that help
  - `{key}` represents a variable that can be accessed via the request
  - The method can be defined together with the endpoint

Handlers

```
mux := http.NewServeMux()

mux.HandleFunc("GET /kvs/{key}", KVSGet)
```

- We define a handler for all GET requests to `/kvs/` followed by a key name
    - e.g., `GET /kvs/my-key`
- Note that “key” is just a name

Implementing the GET handler

```go
func KVSGet(w http.ResponseWriter, r *http.Request) {
    key := r.PathValue("key") // Getting the value from the request path

    value, err := Get(key) // We us the helper to get a value or an error
    if err != nil {
        if errors.Is(err, ErrorNoSuchKey) {
            http.Error(w, err.Error(), http.StatusNotFound)
        } else {
            http.Error(w, err.Error(), http.StatusInternalServerError)
        }
        return
    }

    w.Write([]byte(value)) // If no error, send the value
}
```

And a PUT (to make testing easier)

```go
func KVSPut(w http.ResponseWriter, r *http.Request) {
    key := r.PathValue("key")

    value, err := io.ReadAll(r.Body) // Get the value for the key
    defer r.Body.Close()

    if err != nil { // No value in the request
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    if err = Put(key, string(value)); err != nil { // Setting key failed
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    w.WriteHeader(http.StatusCreated) // Key was set
}
```

Aside: Testing

- We can implement a Go or Python client to test
- We can also use various tools, e.g., curl or wget
- We can set the key *aa* to “Hello!”
  - `curl -X PUT -d 'Hello!' -v http://localhost:3000/kvs/aa`
- And fetch it
  - `curl -v http://localhost:3000/kvs/aa`

Delete for completeness

```go
func KVSDel(w http.ResponseWriter, r *http.Request) {
    key := r.PathValue("key")

    err := Delete(key)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    w.WriteHeader(http.StatusOK)
}
```

The main function

```
	mux := http.NewServeMux()

	mux.HandleFunc("PUT /kvs/{key}", KVSPut)
	mux.HandleFunc("GET /kvs/{key}", KVSGet)
	mux.HandleFunc("DELETE /kvs/{key}", KVSDel)

	http.ListenAndServe(":3000", mux)
```

Persistent storage?

- The keys are currently stored in memory only
- If the server terminates, all the keys are gone
- Multiple ways we can fix that
  - we will add a database for persistent storage
  - mainly to show how to use databases in Go

Databases in Go

- Drivers for various databases, e.g., MySQL
  - “github.com/go-sql-driver/mysql”
- Distributed as Go packages
- Use “database/sql” for API

Connecting to a database

```go
var db *sql.DB
db, err := sql.Open("mysql", "kvsadm@/kvs")
if err != nil {
    log.Fatal(err)
}
```

Testing the connection

» We can use Ping to test the connection `err := db.Ping()`

Rewriting the helpers

- We implemented the main functionality using helper functions
- We can easily modify these to use a database instead
  - Global variable for easy access
  - Initialize in the main method

Rewriting the helpers

```go
func Get(key string) (string, error) {
    row := db.QueryRow("SELECT value FROM kvs WHERE key = ?", key)

    var res string
    if err := row.Scan(&res); err == nil {
        return res, ""
    } else {
        return "", ErrorNoSuchKey
    }
}
```

Query

- QueryRow returns zero or one rows
    - we use Scan to find the value in the row
    - this is where any errors (e.g., no rows) are triggered
- If we expect multiple rows, we should use Query
- Which returns rows and a possible error
- Use Next on the rows to iterate over the result
- And Scan to get the results

Contexts

- We can pass a context to most of the database functions
  - E.g., QueryContext, QueryRowContext and PingContext
- Can be used to, e.g., cancel the operation

Rewriting the helpers

```go
func Delete(key string) error {
    res, err := db.Exec("DELETE FROM vks WHERE key = ?", key)
    return err
}
```

- We use Exec since we do not expect any result from the delete query
- We could check the returned result for, e.g., rows affected