Error handling and best practices, panic and recovery.

Session 05

Golang course by Exadel

17 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Agenda

- ▶️ Errors are just values
- ▶️ Best Practices to handle errors
- ▶️ Defer statement
- ▶️ Runtime panic and recovery
- ▶️ Next time...

Errors in Go

Errors general information

- ✓ Errors are values. Values can be programmed, and since errors are values, errors can be programmed.

- ✓ *nil* error value not equal to *nil*
  - Under the covers, interfaces are implemented *as two elements*, a type and a value.
  - An interface value is *nil* only if the inner value and type are both unset, *(nil, nil)*.
  - It's a good idea for functions that return errors always to use the *error* type in their signature (as we did above) rather than a concrete type such as *MyError, to help guarantee the error is created correctly.
  - Similar situations to those described here can arise whenever interfaces are used.
  - Just keep in mind that if any concrete value has been stored in the interface, the interface will not be nil.

The `error` type

- ✓ The error type is an interface type.
- ✓ An **error** variable represents any value that can describe itself as a string.
  - Error strings should not be capitalized (*unless beginning with proper nouns or acronyms*) or end with punctuation, since they are usually printed following other context.

```go
type error interface {
    Error() string
}
```

- ✓ The most commonly-used error implementation is the **errors** package's unexported **errorString** type.

```go
// errorString is a trivial implementation of error.
type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}
```

How to construct errors?

✓ You can construct one of these values with the **errors.New** function.

- It takes a string that it converts to an **errors.errorString** and returns as an **error** value.

```go
// New returns an error that formats as the given text.
func New(text string) error {
    return &errorString{text}
}
```

✓ Another way of constructing formatted '*error*' instance is calling *fmt.Errorf()* method:

```go
if f < 0 {
    return 0, fmt.Errorf("math: square root of negative number %g", f)
}
```

Error naming conventions

✓ Best practices are:

- Error types should be of the form **FooError**:

```
type ExitError struct {
//   ...
}
```

- Error values should be of the form **ErrFoo**:

```
var ErrFormat = errors.New("image: unknown format")
```

Check code example

Check code example at: `/code/errorexample_01/errors_check_test.go`

Errors example (same as on previous slide)

```go
type MyError struct {
}

func (m MyError) Error() string {
    return "my error"
}

var ErrBad = &MyError{}

func incorrectlyReturnsError() error {
    var p *MyError = nil
    if false {
        p = ErrBad
    }
    return p // Will always return a non-nil error.
}

func correctlyReturnsError() error {
    if true {
        return ErrBad
    }
    return nil
}
```

issue here

What can we do with errors?

- ▶️ Just check whether the error is `nil` or not - the simplest thing possible.
- ▶️ Check if the error is of `some type`?
  - Using `errors.As()` from the "errors" package (https://pkg.go.dev/errors#example-As)
- ▶️ Check if the error is of `some value`?
  - using `errors.Is()` from the "errors" package (https://pkg.go.dev/errors#example-Is)


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

Check code example

Check code example at: `/code/errorexample_01/reduce_number_of_checks_test.go`

Reduce number of checks for errors (same as on previous slide)

- ✓ During interaction with *bufio.Scaner* instance some error may occur and will be accumulated in '*err*' internal variable.
  - Then, after interaction/executing some logic, we can check for error at once:
  - *Err()* method declaration:

```go
// Err returns the first non-EOF error that was encountered by the Scanner.
func (s *Scanner) Err() error {
    if s.err == io.EOF {
        return nil
    }
    return s.err
}
```

  - Client code:

```go
scanner := bufio.NewScanner(input)
// perform some operations:
for scanner.Scan() {
    token := scanner.Text()
    // process token
}
// Check for error only once after:
if err := scanner.Err(); err != nil {
    // process the error
}
```

More info: "Errors are values" by Rob Pike (https://go.dev/blog/errors-are-values)

Another things to consider...

▶️ Error stack traces

- Performance considerations.
- Always include stacktrace only once, upon error wrapping! There is no deduplication.

▶️ Wrapping errors with a reason:

- `fmt.Errorf("bla bla bla: %w", err)`
- check github.com/pkg/errors (https://pkg.go.dev/github.com/pkg/errors)

▶️ Multi-errors:

- Hashicorp's "multierror" (github.com/hashicorp/go-multierror) package. My favourite.
- Uber's "multierr" (go.uber.org/multierr) package. Less popular.

Check code example

Check code example at: `/code/errorexample_01/errors_wrapping_test.go`

Defer

Defer general info

- ✓ A **defer statement** pushes a function call onto a list.
  - The list of saved calls is executed after the surrounding function returns.

- ✓ The behavior of defer statements is straightforward and predictable. There are three simple rules

Defer rule 1 / 3

✓ **Rule 1.** A deferred function's arguments are evaluated when the defer statement is evaluated.

```
func fn() {
// ................
    startTime := time.Now()
    defer l.Metrics.ResponseTime.Observe(time.Since(startTime).Seconds()) // BUG here
// ...............
}
```

Defer rule 2 / 3

✓ **Rule 2.** Deferred function calls are executed in Last In First Out (LIFO) order after the surrounding function returns.

```go
func b() {
    for i := 0; i < 4; i++ {
        defer fmt.Print(i)
    }
}
// OUTPUT:
// 3210
```

- Using this we can do nested cleanup nicely:

```go
func doSomethingWithDB() {
    db.Connect()
    defer db.Disconnect()

    // If Begin panics, only db.Disconnect() will be executed
    transaction.Begin()
    defer transaction.Close()

    // From here on, transaction.Close() will run first,
    // and then db.Disconnect()

    // ...
}
```

Check code example for the previous slide

Check code example at: `/code/deferexample_02/defer_test.go`

Defer rule 3 / 3

✓ **Rule 3.** Deferred functions may read and assign to the returning function's named return values.

```
func c() (i int) {
    defer func() { i++ }()
    return i
}
// returns 1
```

✓ This is convenient for modifying the error return value of a function.

Defer tricks (1/2)

✓ Using defer to clean up readability:

- Consider the following function that saves a user to an in-memory map called `s.users`, protected by a sync.Mutex called "`s.mu`":

```go
func (s *Service) SaveUser(id string) bool {
    // do stuff first
    s.mu.Lock()
    id, present := s.users[id]
    if present {
      s.UpdateUser(id)
      s.mu.Unlock() // unlock here 1st time
      return false
    } else {
      user := s.AddUser(id)
      s.users[id] = user
      s.mu.Unlock() // and unlock here 2nd time, not good
    }
    // do more stuff...
    return true
}
```

- This kind of code is typical, and includes many `Unlock()` calls.

Defer tricks (2/2)

- It's possible to use `defer` in this situation by using an inline `func` *to wrap a set of operations instead*:

```go
func (s *Service) SaveUser(id string) {
    // do stuff first...
    func() {
        s.mu.Lock()
        defer s.mu.Unlock()
        // safely modify the map
        id, present := s.users[id]
        if present {
              s.UpdateUser(id)
             return
        }
        user := s.AddUser(id)
        s.users[id] = user
    }() // call a function!
    // do more stuff...
}
```

Used anonymous function execution!

Runtime panic

Panic

- ✓ When the function F calls **panic()**, execution of F stops, *any deferred functions in F are executed normally*, and then F returns to its caller.
  - To the caller, F then behaves like a call to **panic()**.
  - The process continues *up the stack* until all functions in the current goroutine have returned, at which point the **program crashes**.

- ✓ Panics can be initiated by invoking ***panic()*** directly
  - Never call ***panic(nil)***, because then there is no way to distinguish panic from no-panic case in **recover()** function.

- ✓ Execution errors such as attempting to index an *array out of bounds* trigger a run-time panic
  - equivalent to a call of the built-in function **panic()** with a value of the implementation-defined interface type runtime.Error.

Check code example

Check code example at: `/code/panicexample_03/panic_protection_test.go`

Panic Example

- Panic is not the same as `exception` in JVM languages or other languages.
- In Go the convention is simple: always use `error` values instead of `panic()`.
- Check the code example.
- To print stacktrace, use: `debug.Stack()` from "runtime/debug" (https://pkg.go.dev/runtime/debug#Stack) package.

Panic Gotcha

✓ The example below invokes the function argument fn and protects callers from run-time panics raised by fn:

```go
func protect(fn func()) {
    defer func() {
        log.Println("done")  // Println executes normally even if there is a panic
        if x := recover(); x != nil {
            log.Printf("Run time panic: %v", x)
        }
    }()
    log.Println("Start execution of g()")
    fn()
}
```

✓ By convention, no explicit *panic()* should be allowed to cross a package boundary.

- Indicating error conditions to callers should be done by returning error value.
- *Within a package*, however, especially if there are deeply nested calls to non-exported functions, it can be useful (and improve readability) to use *panic()* to indicate error conditions which should be translated into error for the calling function.

Recover

Recover introduction

- ✓ A *built-in function* that regains control of a panicking goroutine.
- ✓ `recover` is only useful inside deferred functions.
  - During normal execution, a call to `recover` will return nil and have no other effect.
- ✓ If the *current* goroutine is panicking, a call to `recover` will capture the value given to `panic` and resume normal execution.
- ✓ A panic cannot be recovered by a different goroutine.

Source: "PanicAndRecover" by Google Go Wiki (https://github.com/golang/go/wiki/PanicAndRecover)

Check code example

Check code example at: `/code/recoveryexample_04/recovery_example_test.go`

Recover example (same as on the previous slide)

```go
package main
import "fmt"

func main() {
	f()
	fmt.Println("Returned normally from f.")
}

func f() {
	defer func() {
		if r := recover(); r != nil {
			fmt.Println("Recovered in f", r)
		}
	}()
	fmt.Println("Calling g.")
	g(0)
	fmt.Println("Returned normally from g.")
}

func g(i int) {
	if i > 3 {
		fmt.Println("Panicking!")
		panic(fmt.Sprintf("%v", i))
	}
	defer fmt.Println("Defer in g", i)
	fmt.Println("Printing in g", i)
	g(i + 1)
}
```

Recover gotcha

✓ The return value of `recover()` is `nil` if **any** of the following conditions holds:

- panic's argument was `nil` - NEVER DO THIS!
- the goroutine is *not panicking*;
- recover() was not called **directly** by a deferred function:

```go
func doRecover() {
    fmt.Println("recovered =>",recover()) //prints: recovered => <nil>
}

func main() {
    defer func() {
        doRecover() //panic is not recovered!!!
    }()

    panic("not good")
}
```

Check code example (same as on the previous slide)

Check code example at: `/code/recoveryexample_04/recovery_gotcha_test.go`

Next time...

▶️ Homework:

- Investigate and play with libraries:
  - https://pkg.go.dev/github.com/pkg/errors
  - https://pkg.go.dev/github.com/hashicorp/go-multierror

▶️ Session06:

Type system in Go: overview

Thank you

Golang course by Exadel

17 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)