Copyright 2013-2017, RX-M LLC

Error handling

Copyright 2013-2017, RX-M LLC

Objectives

- Explore Go error management
- Understand:
  - Errors
  - panic()
  - recover()
  - defer

Defer

- defer statement
  - An ordinary function call prefixed by the defer keyword
- Deferred function and argument expressions are evaluated when the statement is executed
- The actual call is deferred until the function completes
  - By return statement, falling off the end, exiting or panicking
- Any number of calls may be deferred
  - Deferred calls are executed in reverse order in which they were deferred
- A defer statement is often used with paired operations
  - e.g. open/close, connect/disconnect, lock/unlock
  - Performs like a finally block in some languages, ensuring that resources are released in all cases
  - The right place for a defer statement is immediately after the resource has been successfully acquired
- Deferred functions run after return statements have updated the function's result variables
  - An anonymous function can access its enclosing function's variables, including named results, even change them … (!)
- The example program defers the HTTP response body close operation
  - Also demonstrates use of nested anonymous functions for internal recursion
  - This function uses a closure to call itself (required since the function is anonymous)
  - The code uses the golang net/html extension package (not part of the std lib) to parse the HTML document

```go
package main

import (
    "fmt"
    "net/http"
    "strings"

    "golang.org/x/net/html"
)

func title(url string) (*html.Node, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()

    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        return nil, fmt.Errorf("% s has type %s, not text/html", url, ct)
    }
    doc, err := html.Parse(resp.Body)
    if err != nil {
        return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    var f func(n *html.Node) *html.Node
    f = func(n *html.Node) *html.Node {
        if n.Type == html.ElementNode && n.Data == "title" {
            return n.FirstChild
        }
        for c := n.FirstChild; c != nil; c = c.NextSibling {
            p := f(c)
            if p != nil {
                return p
            }
        }
        return nil
    }
    return f(doc), nil
}

func main() {
    el, _ := title("http://rx-m.com")
    fmt.Println(el.Data)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 20:51:43 server.go:73: Using API v1
2017/04/10 20:51:43 debugger.go:68: launching process with args: [d:\dev\go\exampl
API server listening at: 127.0.0.1:2345
2017/04/10 20:51:43 debugger.go:414: continuing
RX-M - DevOps & Cloud Native Consulting and Training - RX-M
```

Errors

Copyright 2013-2017, RX-M LLC

- Some functions always succeed (200)
- Some functions fail if preconditions are not met (400)
  - Errors are a result of bad calls
- Some functions given proper preconditions fail due to factors beyond the functions control (500)
  - Any function that does I/O
- Errors are an important part of a package's API
  - Go considers errors one of a set of expected behaviors
  - If a function can fail it is expected to return an error result
    - Conventionally the last thing in the return list
  - If the failure has only one possible cause the result is (by convention) a Boolean called "ok"

```go
package main

import "fmt"

func div(x, y int) (res int, ok bool) {
    if y == 0 {
        return 0, false
    }
    return x / y, true
}

func main() {
    ans, ok := div(8, 4)
    if ok {
        fmt.Println("First try: ", ans)
    }
    ans, ok = div(8, 0)
    if ok {
        fmt.Println("Second try: ", ans)
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 15:11:57 server.go:73: Using API v1
2017/04/10 15:11:57 debugger.go:68: launching pr
API server listening at: 127.0.0.1:2345
2017/04/10 15:11:57 debugger.go:414: continuing
First try:  2
```

error

Copyright 2013-2017, RX-M LLC

- Non trivial (bool) errors are indicated by returning an “error” instance as a final return value
- error is a built-in interface type
  - An error may be:
    - nil - implies success
    - non-nil - implies failure
      - Such an error has a message string obtained by calling its Error method
      - To print the message: fmt.Println(err) or fmt.Printf("%v", err) of fmt.Println(err.Error())
      - Usually when a function returns a non-nil error its other results are undefined
        - Some rare functions return partial results in error cases
- Go does not offer a traditional exception handling system
  - Go supports a type of exception for reporting truly unexpected errors that indicate a bug (not to be used for reporting routine errors)
  - The Go philosophy is that exceptions entangle the description of an error with the control flow required to handle it
    - This leads to routine errors being reported to the end user in a stack trace, full of information about the structure of the program but lacking intelligible context about what went wrong
  - Go programs use ordinary control-flow mechanisms (e.g. if and return) to respond to errors
    - This style undeniably demands that more attention be paid to error-handling logic
    - precisely the point

```go
package main

import (
    "errors"
    "fmt"
)

func posdiv(x, y int) (res int, err error) {
    if y < 0 {
        return 0, errors.New("attempted posdiv division with a negative number")
    }
    if y == 0 {
        return 0, errors.New("attempted division by 0")
    }
    return x / y, nil
}

func main() {
    ans, err := posdiv(8, -4)
    if err != nil {
        fmt.Println("Error: ", err.Error())
    } else {
        fmt.Println("Result: ", ans)
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 15:57:03 server.go:73: Using API v1
2017/04/10 15:57:03 debugger.go:68: launching process with args: [d:\dev\go\example\src\
API server listening at: 127.0.0.1:2345
2017/04/10 15:57:03 debugger.go:414: continuing
Error:  attempted posdiv division with a negative number
```

Propagating errors

Copyright 2013-2017, RX-M LLC

- Errors in low level routines must often be propagated by mid level routines that have no stake in the error
  - return err
- A call which produces an error such that the callee does not have enough context to produce a meaningful explanation may require the caller to augment it to keep the error meaningful
  - Go allows new errors to be fabricated from lower level error messages
    - Fmt.Errorf("%v", err)
    - errors.Wrap(err, "read failed")
  - Error messages are frequently chained in this way
- Therefore error messages should:
  - Avoid newlines
  - Not be capitalized
  - Provide a meaningful description of the problem
  - Supply sufficient/relevant detail
  - Be consistent
    - errors returned by the same function/package should be of similar form
    - e.g. The os package guarantees that every error returned by a file operation describes the nature of the failure (permission denied) and the name of the file
- Resulting errors may be long, but they will be self-contained and easily parsed by tools like grep

```go
package main

import "errors"
import "fmt"

func posdiv(x, y int) (res int, err error) {
    if y < 0 {
        return 0, errors.New("attempted posdiv division with a negative number")
    }
    if y == 0 {
        return 0, errors.New("attempted division by 0")
    }
    return x / y, nil
}

func multidiv(x, y1, y2 int) (res int, err error) {
    ans1, e1 := posdiv(x, y1)
    ans2, e2 := posdiv(x, y2)
    if e1 != nil || e2 != nil {
        return 0, fmt.Errorf("multidiv component failure, y1: %v, y2: %v", e1, e2)
    }
    return ans1 + ans2, nil
}

func main() {
    ans, err := multidiv(8, 2, -4)
    if err != nil {
        fmt.Println("Error: ", err.Error())
    } else {
        fmt.Println("Result: ", ans)
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 16:09:56 server.go:73: Using API v1
2017/04/10 16:09:56 debugger.go:68: launching process with args: [d:\dev\go\example\src\debug]
API server listening at: 127.0.0.1:2345
2017/04/10 16:09:56 debugger.go:414: continuing
Error:  multidiv component failure, y1: <nil>, y2: attempted posdiv division with a negative number
```

Retry Errors

Copyright 2013-2017, RX-M LLC

- Transient/unpredictable problems may require code to retry the failed operation
  - Perhaps with a delay
  - Perhaps with a limit on the number of attempts or time spent
  - Transient errors recovered from should not go unrecorded
- The Go log package
  - Implements simple logging
  - Defines a Logger type with methods for formatting output
  - Has a predefined 'standard' Logger accessible through helper functions Print[f|ln], Fatal[f|ln], and Panic[f|ln]
    - The Fatal functions call os.Exit(1) after writing the log message
    - The Panic functions call panic after writing the log message
  - The standard logger writes to standard error and prints the date and time of each logged message
  - Every log message is output on a separate line
    - if the message being printed does not end in a newline, the logger will add one
- Stderr can be printed to directly with the fmt package
  - fmt.Fprintf(os.Stderr, “big problem!”)

```
1  package main
2
3  import "log"
4
5  func main() {
6      log.Printf("connection failed retrying")
7  }
8
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 16:31:22 server.go:73: Using API v1
2017/04/10 16:31:22 debugger.go:68: launching process
API server listening at: 127.0.0.1:2345
2017/04/10 16:31:23 debugger.go:414: continuing
2017/04/10 16:31:23 connection failed retrying
```

Stopping Execution

Copyright 2013-2017, RX-M LLC

- If the error is unrecoverable it may be best to terminate execution
  - The os package provides an func Exit(code int) function to terminate execution
    - The code passed to Exit is passed to the system as the process exit code (0 typically indicates success, all other values indicate failure)
  - This decision should be reserved for the most abstract parts of the program (e.g. main())
- Go error handling style
  - Invoke risky function
    - Checking for error
    - Deal with Failure
      - If failure causes the function to return the success logic follows at the outer level
    - Deal with Success
  - Functions include initial checks to reject errors (DbC*) followed by the substance of the function minimally indented

** In Design by Contract style*

Distinguished Errors

Copyright 2013-2017, RX-M LLC

- Some packages define specific errors
  - Referred to as distinguished errors
  - io.EOF is a good example
    - In an end-of-file condition there is no information to report
    - This error has a fixed message: "EOF"
  - if err == io.EOF
    - A legal expression because the io.EOF variable is always returned when an EOF event occurs

EOF is the error returned by Read when no more input is available. Functions should return EOF only to signal a graceful end of input. If the EOF occurs unexpectedly in a structured data stream, the appropriate error is either ErrUnexpectedEOF or some other error giving more detail.

io package errors

```
var EOF = errors.New("EOF")
```

ErrClosedPipe is the error used for read or write operations on a closed pipe.

```
var ErrClosedPipe = errors.New("io: read/write on closed pipe")
```

ErrNoProgress is returned by some clients of an io.Reader when many calls to Read have failed to return any data or error, usually the sign of a broken io.Reader implementation.

```
var ErrNoProgress = errors.New("multiple Read calls return no data or error")
```

ErrShortBuffer means that a read required a longer buffer than was provided.

```
var ErrShortBuffer = errors.New("short buffer")
```

ErrShortWrite means that a write accepted fewer bytes than requested but failed to return an explicit error.

```
var ErrShortWrite = errors.New("short write")
```

ErrUnexpectedEOF means that EOF was encountered in the middle of reading a fixed-size block or data structure.

```
var ErrUnexpectedEOF = errors.New("unexpected EOF")
```

Panics

Copyright 2013-2017, RX-M LLC

- In Go execution errors such as attempting to index an array out of bounds trigger a run-time panic
  - Can also be triggered by the built-in function panic(string)
- Panics are designed to fail fast, stopping all operations quickly
- A common use of panic is to abort when a function returns an error value that we don't know how to (or want to) handle
  - Last resort, idiomatic go code returns meaningful errors
- During a typical panic, normal execution stops, all deferred function calls are executed, and the program crashes with a log message
  - This log message includes the panic value, which is usually an error message of some sort a stack trace showing the stack of function calls that were active at the time of the panic
  - This log message often has enough information to diagnose the root cause of the problem without running the program again, so it should always be included in a bug report about a panicking program

```
1  package main
2
3  func main() {
4      panic(3)
5  }
6
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 21:13:04 server.go:73: Using API v1
2017/04/10 21:13:04 debugger.go:68: launching proc
API server listening at: 127.0.0.1:2345
2017/04/10 21:13:04 debugger.go:414: continuing
2017/04/10 21:13:08 debugger.go:414: continuing
panic: 3

goroutine 1 [running]:
main.main()
        d:/dev/go/example/src/example.go:4 +0x5f
```

Recover

Copyright 2013-2017, RX-M LLC

- Giving up is usually the right response to a panic
- In some cases it may be possible to recover
  - Or clean up before quitting (e.g. a web server could close connections before crashing)
- If the built-in recover() function is called within a deferred function that is panicking the current state of panic ends and recover() returns the panic code
  - The function that was panicking does not continue where it left off but returns normally
- If recover is called at any other time, it has no effect and returns nil
- The example defers an anonymous function (self invoking due to the trailing parenthesis)
  - The panic code was 3 so the Println() displays this as the result of recover()
  - The program exits normally

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       defer func() {
7           fmt.Println(recover())
8       }()
9       panic(3)
10  }
11
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL
2017/04/10 21:22:38 server.go:73: Using API v1
2017/04/10 21:22:38 debugger.go:68: launching proce
API server listening at: 127.0.0.1:2345
2017/04/10 21:22:38 debugger.go:414: continuing
3
```

Copyright 2013-2017, RX-M LLC

Summary

- Go provides unique Go specific function features
  - Deferred functions, no exceptions, …
- Go error management is explicit using tools such as:
  - Errors
  - panic()
  - recover()
  - defer

Copyright 2013-2017, RX-M LLC

Lab: Error handling

- Error handling