Error-Handling

This lesson introduces error-handling in Go.

Introduction

Go does not have an exception-handling mechanism, like the try/catch in Java or .NET. For instance, you cannot throw exceptions. Instead, it has a *defer-panic-and-recover* mechanism. The designers of Go thought that the try/catch mechanism is overused and that the throwing of exceptions in lower layers to higher layers of code uses too many resources. The mechanism they devised for Go can 'catch' an exception, but it is much lighter. Even then, it should only be used as a last resort.

How then does Golang deal with normal errors by default? The Go way to handle errors is for functions and methods to return an error object as their only or last return value—or nil if no error occurred—and for the code calling functions to always check the error they receive.

> **Note**: Never ignore errors because ignoring them can lead to program crashes.

Handle the errors and return from the function in which the error occurred with an error message to the user: that way, if something goes wrong, your program will continue to function, and the user will be notified. The purpose of panic-and-recover is to deal with genuinely exceptional (so unexpected) problems and not with normal errors.

Go makes a distinction between critical and non-critical errors: non-critical errors are returned as normal return values, whereas for critical errors, the panic-recover mechanism is used.

Library routines often return some sort of error indication to the calling function. In the preceding chapters, we saw the idiomatic way in Go to detect and report error conditions:

- A function which can result in an error returns two variables, a value, and an error-code; the latter is *nil* in cases of success and *!= nil* in cases of an error condition.
- After the function call, the error is checked. In case of an error (if error != nil), the execution of the current function (or if necessary, the entire program) is stopped.

In the following code, `Func1` from package `pack1` is tested on its return code:

```
if value, err := pack1.Func1(param1); err != nil {
  fmt.Printf("Error %s in pack1.Func1 with parameter %v", err.Error(), par
am1)
  return // or: return err
}
// Process(value)
```

Always assign an error to a variable within a compound if-statement; this makes for clearer code.

Instead of `fmt.Printf`, corresponding methods of the `log` package could be used, or even a `panic`, if it doesn't matter that the program aborts.

Go has a built-in error interface type:

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

Error values are used to indicate an abnormal state. The package `errors` contains an `errorString` struct, which implements the error interface. To stop the execution of a program in an error-state, we can use os.Exit(1).

```
error interface
errors.New(text string) error
err.Error() string
```

Defining errors

Whenever you need a new error-type, you can make one with the function `errors.New` from the `errors` package (which you will have to import), and give it an appropriate error-string, as follows:

```
err := errors.New("math - square root of negative number")
```

Below, you see a simple example of its use:

```
package main
import (
    "errors"
    "fmt"
)

var errNotFound error = errors.New("Not found error")

func main() {
    fmt.Printf("error: %v", errNotFound)
}
// error: Not found error
```

Errors Definition

At **line 3**, we import the `errors` package. At **line 7**, we make a new `error` instance `errNotFound`, with a specific message. Then, at **line 10**, we print out this error, which displays its *message* (**%v** is the *default* formatting).

Applied in a function testing the parameter of a square root function, this could be used as:

```
func Sqrt(f float64) (float64, error) {
  if f < 0 {
    return 0, errors.New("math - square root of negative number")
  }
  // implementation of Sqrt
}
```

You could call this function as follows:

```
if f, err := Sqrt(-1); err != nil {
  fmt.Printf("Error: %s\n", err)
}
```

Because `fmt.Printf` automatically uses the `Error()` method for `err`, the error-string **“Error: math - square root of negative number”** is printed out. Because there will often be a prefix like `Error:`, it is preferred not to start your error string with a capital letter.

In most cases, it is useful to make a *custom error struct type*, which apart from the (low level) error-message also contains other useful information, such as the operation which was taking place (open file, …), the full path-name or URL which was involved, and so on. The `String()` method then provides an informative concatenation of all this information. As an example, see `PathError`, which can be issued from an `os.Open`:

```
// PathError records an error and the operation and file path that cause
d it.
type PathError struct {
  Op string // "open", "unlink", and so on
  Path string // The associated file.
  Err error // Returned by the system call.
}

func (e *PathError) String() string {
  return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

In case different possible error-conditions occur, it may be useful to test with a type assertion or type switch for the exact error. Possibly, try a remedy or a recovery of the error-situation:

```
// err != nil
if e, ok := err.(*os.PathError); ok {
  // remedy situation
}
```

Or:

Or:

```
switch err := err.(type) {
  case ParseError:
    PrintParseError(err)
  case PathError:
    PrintPathError(err)
  ...
  default:
    fmt.Printf("Not a special error, just %s\n", err)
}
```

As a 2nd example, consider the `json` package. This specifies a `SyntaxError` type that the `json.Decode()` function returns when it encounters a syntax error parsing a JSON document:

```
type SyntaxError struct {
  msg string // description of error
  // error occurred after reading Offset bytes, from which line and colum
n can be obtained
  Offset int64
}

func (e *SyntaxError) String() string { return e.msg }
```

In the calling code, you could again test whether the error is of this type with a type assertion, like this:

```
if serr, ok := err.(*json.SyntaxError); ok {
  line, col := findLine(f, serr.Offset)
  return fmt.Errorf("%s:%d:%d: %v", f.Name(), line, col, err)
}
```

A package can also define its own specific `Error` with additional methods, like `net.Error`:

```
package net

type Error interface {
  error
  Timeout() bool // Is the error a timeout?
  Temporary() bool // Is the error temporary?
}
```

As you have seen in all the examples, the following naming convention is applied: Error types end in *Error*, and error variables are called (or start with) *err* or *Err*.

The `syscall` is the low-level, external package, which provides a primitive interface to the underlying operating system's calls; these return integer error-codes. The type `syscall.Errno` implements the Error interface. Most `syscall` functions return a result and a possible error, like:

```
r, err := syscall.Open(name, mode, perm)

if err != 0 {
  fmt.Println(err.Error())
}
```

`os` also provides a standard set of error-variables like `os.EINVAL`, which come from syscall - errors:

```
var (
EPERM Error = Errno(syscall.EPERM)
ENOENT Error = Errno(syscall.ENOENT)
ESRCH Error = Errno(syscall.ESRCH)
EINTR Error = Errno(syscall.EINTR)
EIO Error = Errno(syscall.EIO)
...
)
```

Making an error-object with `fmt`

Often, you will want to return a more informative string with, for example, the value of the wrong parameter inserted; this is accomplished with the `fmt.Errorf()` function. It works exactly like `fmt.Printf()`, taking a format string with one or more format specifiers and a corresponding number of variables to be substituted. But, instead of printing the message, it generates an error object with that message. Applied to our Sqrt-example from above:

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

While reading from the command-line, we generate an error with a usage message when a help-flag is given:

```go
if len(os.Args) > 1 && (os.Args[1] == "-h" || os.Args[1] == "--help") {
  err = fmt.Errorf("usage: %s infile.txt outfile.txt", filepath.Base(os.Ar
gs[0]))
return
}
```

Making error objects to catch errors is the basics of the error-handling. What if we found an error at runtime? What is the behavior of a Go program in this case? And what can we do in this regard? To get a hold on all of this information, move to the next lesson.