Error-Handling Scheme with Closures

This lesson discusses closures for error-handling purposes in detail.

Whenever a function returns, we should test whether it results in an error: this can lead to repetitive and tedious code. Combining the *defer/panic/recover* mechanisms with closures can result in a far more elegant scheme that we will now discuss. However, it is only applicable when all functions have the same signature, which is rather restrictive. A good example of its use is in web applications, where all handler functions are of the following type:

```
func handler1(w http.ResponseWriter, r *http.Request) { ... }
```

Suppose all functions have the signature: `func f(a type1, b type2)`. The number of parameters and their types is irrelevant. We give this type a name: `fType1 = func f(a type1, b type2)`.

Our scheme uses 2 helper functions:

- *check*: a function which tests whether an error occurred, and panics if so: `func check(err error) { if err != nil { panic(err) } }`
- *errorhandler*: this is a wrapper function. It takes a function `fn` of our type `fType1` as parameter, and returns such a function by calling `fn`. However, it contains the *defer/recover mechanism*:

```
func errorHandler(fn fType1) fType1 {
  return func(a type1, b type2) {
    defer func() {
      if e, ok := recover().(error); ok {
        log.Printf("run time panic: %v", err)
      }
    }()
    fn(a, b)
  }
}
```

When an error occurs, it is recovered and printed on the log. Apart from simply printing the error, the application could also produce a customized output for the user by using the `template` package. In a real project, a less obtrusive name than the `errorHandler` could be used, like `call(f1)`.

The `check()` function is used in every called function, like this:

```
func f1(a type1, b type2) {
  ...
  f, _, err := // call function/method
  check(err)
  t, err = // call function/method
  check(err)
  _, err = // call function/method
  check(err)
  ...
}
```

The `main()` or other caller-function should then call the necessary functions wrapped in `errorHandler`, like this:

```
func main() {
  errorHandler(f1)
  errorHandler(f2)
  ...
}
```

By using this mechanism, all errors are recovered, and the error-checking code after a function call is reduced to check(err). In this scheme, different error-handlers have to be used for different function types; they could be hidden inside an error-handling package. Alternatively, a more general approach could be using a slice of empty interface as a parameter and return type. We will apply this in the web application in Chapter 13.

That's it about errors and handling them in different possible ways. The next lesson brings you a challenge to solve.