Go Error Handling

by Doug Wilson

to err is human,
but it happens to
computers too

Signaling errors

Signaling errors

Most familiar languages signal errors through the use of a `throw` keyword. In Go, the error is signaled through a *return value*.

```go
func DoThing() error {
    return errors.New("do thing failed")
}
```

Signaling errors

Functions will likely need to return a non-error value, though. In Go, functions can return multiple values. Thus the error can just be *another* return value.

```
func Sum(a int, b int) (int, error) {
   return 0, errors.New("sum failure")
}
```

Signaling errors

The error value can be returned in any way a function likes, but the *convention* is to return the error as the right-most (last) return value.

This is similar to the callback style in Node.js, where the error is another returned value.

Handling errors

Handling errors

Once an error is signaled by a function, it should be handled by the caller. Since errors are return values, a conditional statement is used in the caller to branch for an error condition.

```
sum, err := Sum(40, 2)
if err != nil {
    fmt.Println(err)
}
```

Errors must be explicitly checked for

Handling errors

This is both tedious and flexible. The conditionals are verbose, though being values can simplify flows where two operations are performed and errors are checked after both are called.

```
erra := DoThingA()
errb := DoThingB()
if erra != nil || errb != nil {
    fmt.Println("oh no!")
}
```

Error type

Error type

The built-in `error` type in Go is an interface type with the following simple definition:

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

The only thing defined is a method `Error()` which returns a string. This is similar to a `Message` property in other languages.

Structured errors

Structured errors

A single string message may be OK for humans, but to program in error handling, structured errors is useful. For example, knowing what file path was not found to create it and try again.

```
type PathError struct {
    Op   string
    Path string
    Err  error
}
```

Structured errors

The underlying struct can be accessed through a type assertion, since the convention will return the value as just the error interface.

```go
f, err := os.Open("/foo.txt")
if err, ok := err.(*os.PathError); ok {
    fmt.Println("File at path", err.Path, "failed to open")
    return
}
```

Stack traces

Stack traces

By default, the built-in errors capture no stack trace. This may be surprising compared to many other languages which capture a stack trace within the error object on creation.

Don’t be confused if you’ve seen stack traces in Go; a `panic` prints a stack trace which gives the *point of the panic*.

Stack traces

Go provides the `runtime/debug` package to print or capture the stack trace as the call site.

```
sum, err := Sum(40, 2)
if err != nil {
    fmt.Println(err)
    debug.PrintStack()
}
```

Stack traces

Including a stack trace in the error object is DIY for the time being in Go. Common methods are including the stack trace at the end of the message (as returned by `Error()`) or including a `[]byte` property on the underlying `struct`.

```
return fmt.Errorf("failure\n%s", debug.Stack())
```

go forth and error

Thank You!