Programming in Go

Matt Holiday
Christmas 2020

Customizing Errors

Simple Errors

Most of the time, errors are just strings

```
func (h HAL9000) OpenPodBayDoors() error {
    if h.kill {
        return fmt.Errorf("I'm sorry %s, I can't do that", h.victim)
    }

    . . .
}
```

Error types

Errors in Go are objects satisfying the `error` interface:

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

Any *concrete* type with `Error()` can represent an error

```
type Fizgig struct{}

func (f Fizgig) Error() string {
    return "Your fizgig is bent"
}
```

A custom error type

We're going to build out a custom error type

```
type errKind int

const (
    _               errKind = iota // so we don't start at 0
    noHeader
    cantReadHeader
    invalidHdrType
    . . .
}

type WaveError struct {
    kind  errKind
    value int
    err   error
}
```

A custom error type

We use different formats depending on the situation

```
func (e WaveError) Error() string {
    switch e.kind {
    case noHeader:
        return "no header (file too short?)"

    case cantReadHeader:
        return fmt.Sprintf("can't read header[%d]: %s", e.value, e.err.Error())

    case invalidHdrType:
        return "invalid header type"

    case invalidChkLength:
        return fmt.Sprintf("invalid chunk length: %d", e.value)
    . . .
}
```

A custom error type

We have a couple of helper methods to generate errors

```
// with returns an error with a particular value (e.g., header type)
func (e WaveError) with(val int) WaveError {
    e1 := e
    e1.value = val
    return e1
}

// from returns an error with a particular location and
// underlying error (e.g., from the standard library)
func (e WaveError) from(pos int, err error) WaveError {
    e1 := e
    e1.value = pos
    e1.err = err
    return e1
}
```

A custom error type

And we have some prototype errors we can return or customize

```
var (
    HeaderMissing        = WaveError{kind: noHeader}
    HeaderReadFailed     = WaveError{kind: cantReadHeader}
    InvalidHeaderType    = WaveError{kind: invalidHdrType}
    InvalidChunkLength   = WaveError{kind: invalidChkLength}
    InvalidChunkType     = WaveError{kind: invalidChkType}
    InvalidDataLength    = WaveError{kind: invalidLength}

    . . .
)
```

The custom error type in use

Here's an example of those errors in use

```
func DecodeHeader(b []byte) (*Header, []byte, error) {
    var err error
    var pos int

    header := Header{TotalLength: uint32(len(b))}
    buf := bytes.NewReader(b)

    if len(b) < headerSize {
        return &header, nil, HeaderMissing
    }

    if err = binary.Read(buf, binary.BigEndian, &header.riff); err != nil {
        return &header, nil, HeaderReadFailed.from(pos, err)
    }

    . . .
```

Wrapped errors

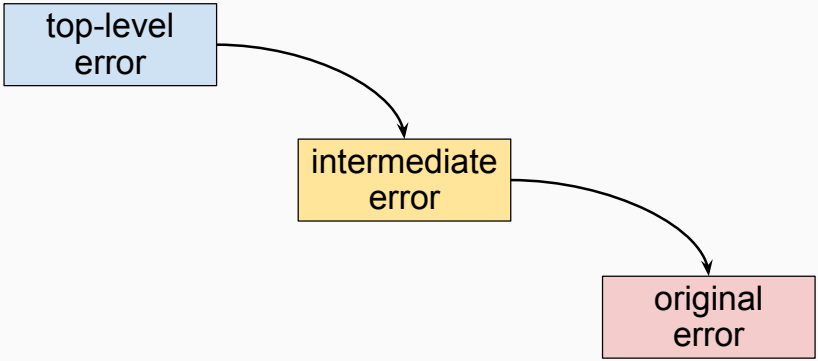
Starting with Go 1.13, we can wrap one error in another

```
func (h HAL9009) OpenPodBayDoors() error {
    . . .

    if h.err != nil {
        return fmt.Errorf("I'm sorry %s, I can't: %w", h.victim, h.err)
    }

    . . .
}
```

The easiest way to do that is to use the %w format verb with `fmt.Errorf()`

Wrapped errors

Wrapping errors gives us an error chain we can unravel



Wrapped errors

Custom error types may now unwrap their internal errors

```
type WaveError struct {
    kind  errKind
    value int
    err   error
}

func (w *WaveError) Unwrap() error {
    return w.err
}
```

errors.Is

We can check whether an error has another error in its chain

`errors.Is` compares with an error **variable**, not a type

```
. . .

if audio, err = DecodeWaveFile(fn); err != nil {
    if errors.Is(err, os.ErrPermission) {
        // let's report a security violation

        . . .
    }

    . . .
}
```

Customized tests

We can provide the `Is()` method for our custom type (only useful if we export our error *variables* also)

```
type WaveError struct {
    kind  errKind
    . . .
}

func (w *WaveError) Is(t error) bool {
    e, ok := t.(*WaveError)  // reflection again

    if !ok {
        return false
    }

    return e.errKind == w.errKind
}
```

errors.As

We can get an error of an underlying type if it's in the chain

`errors.As` looks for an error **type**, not a value

```
. . .

if audio, err = DecodeWaveFile(fn); err != nil {
    var e os.PathError   // a struct

    if errors.As(err, &e) {
        // let's pass back just the underlying file error

        return e
    }

    . . .
```

A Philosophy of Error Handling

Errors in Go

When it comes to errors, you may fall into one of these camps:

1. you hate constantly writing if/else blocks
2. you think writing if/else blocks makes things clearer
3. **you don't care because you're too busy writing code**

Normal errors

Normal errors result from *input* or *external conditions*
(for example, a “file not found” error)

Go code handles this case by returning the `error` type

```
// Not exactly os.Open, but shows the basic logic

func Open(name string, flag int, perm FileMode) (*File, error) {
    r, e := syscall.Open(name, flag|syscall.O_CLOEXEC, syscallMode(perm))

    if e != nil {
        return nil, &PathError{"open", name, e}
    }

    return newFile(uintptr(r), name, kindOpenFile), nil
}
```

Abnormal errors

Abnormal errors result from *invalid program logic*
(for example, a nil pointer)

For program logic errors, Go code does a `panic`

```
func (d *digest) checkSum() [Size]byte {
    // finish writing the checksum
    . . .

    if d.nx != 0 {            // panic if there's data left over
        panic("d.nx != 0")
    }

    . . .
```

When your program has a logic bug

“Fail hard, fail fast”

When your program has a logic bug

If your server crashes, it will get immediate attention

- logs are often noisy
- so proactive log searches for “problems” are rare

We want evidence of the failure as close as possible in time and space to the original defect in the code

- connect the crash to logs that explain the context
- traceback from the point closest to the broken logic

In a distributed system, *crash failures* are the safest type to handle

- it’s better to die than to be a zombie or babble or corrupt the DB
- not crashing may lead to *Byzantine failures*

When should we panic?

Only when the error was caused by our own programming defect, e.g.

- we can't walk a data structure we built
- we have an off-by-one bug encoding bytes

In other words,

> `panic` *should be used when our assumptions of our own programming design or logic are* **wrong**

These cases might use an "assert" in other programming languages

When should we panic?

A *B-tree* data structure satisfies several invariants:

1. every path from the root to a leaf has the same length
2. if a node has n children, it contains $n - 1$ keys
3. every node (except the root) is at least half full
4. the root has at least two children if it is not a leaf
5. subnode keys fall between the keys of the parent node that lie on either side of the subnode pointer

If any of these is ever false, the B-tree methods should **panic!**

Exception handling

Exception handling was popularized to allow “graceful degradation” of safety-critical systems (e.g., Ada and flight control software)

Ironically, most safety-critical systems are built without using exceptions!

Exception handling introduces invisible control paths through code

So code with exceptions is harder to analyze (automatically or by eye)

Exception handling

Officially, Go doesn't support exception handling as in other languages

Practically, it does — in the form of `panic` & `recover`

`panic` in a function will still cause deferred function calls to run

Then it will stop only if it finds a valid `recover` call in a `defer`
as it unwinds the stack

Panic and recover

Recovery from `panic` only works inside `defer`

```
func abc() {
    panic("omg")
}

func main() {
    defer func() {
        if p := recover(); p != nil {
            // what else can you do?

            fmt.Println("recover:", p)
        }
    }()

    abc()
}
```

Define errors out of existence

Error (edge) cases are one of the primary sources of complexity

The best way to deal with many errors is to make them impossible

Design your abstractions so that most (or all) operations are safe:

- reading from a nil map
- appending to a nil slice
- deleting a non-existant item from a map
- taking the length of an unitialized string

Try to reduce edge cases that are hard to test or debug (or even think about!)

Proactively Prevent Problems™

Every piece of data in your software should start life in a valid state

Every transformation should leave it in a valid state

- break large programs into small pieces you can understand
- hide information to reduce the chance of corruption
- avoid clever code and side effects
- avoid unsafe operations
- assert your invariants
- never ignore errors
- **test, test, test**

Never, ever accept input from a user (or environment) without validation

Error handling culture

“Go programmers think about the failure case first.

We solve the ‘what if ...’ case first. This leads to programs where failures are handled at the point of writing, rather than the point they occur in production. The verbosity of

```
if err != nil {
    return err
}
```

is outweighed by the value of deliberately handling each failure condition at the point at which it occurs. Key to this is the cultural value of handling each and every error explicitly.” — Dave Cheney