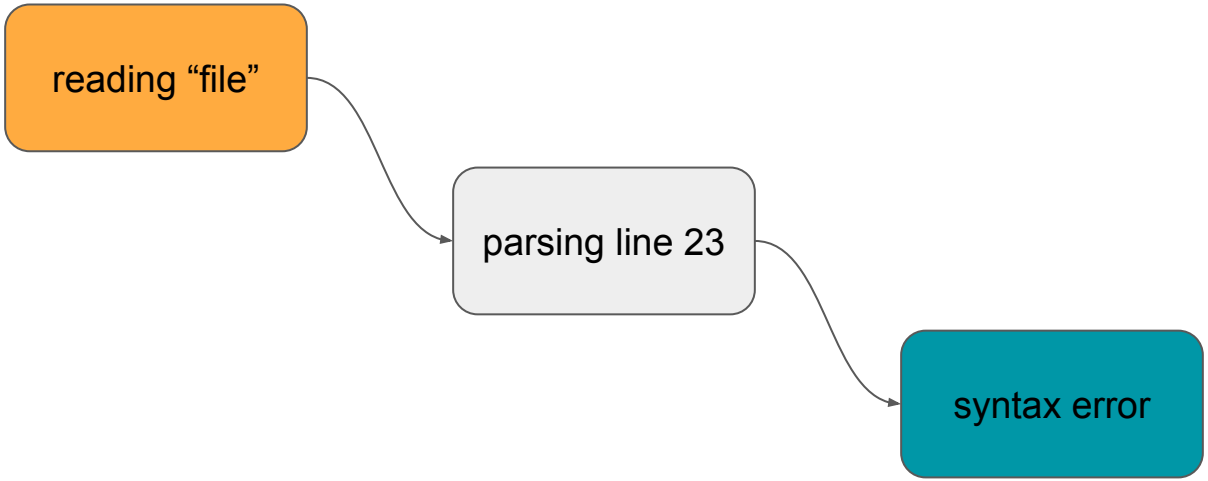
Working with Error Wrapping

Jonathan Amsterdam

GopherCon

November 13, 2020

Wrapping and the error chain



Error-wrapping support in Go 1.13

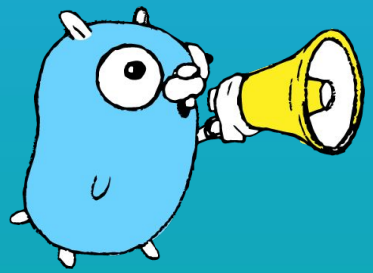
errors.Unwrap
errors.Is
errors.As

fmt.Errorf(" ... %w ... ")

Two audiences for errors

People
- Diagnose
- Debug

Programs
- Retry
- Try an alternative
- Provide better messages for people



An Error Story

The Details

Using Wrapping Effectively

Features That Didn’t Make It

A Config struct

```
type Config struct {
    DatabaseURL string
    MaxTasks    int
    // ...
}
```

Returning errors unadorned

```
func ReadConfig(filename string) (*Config, error) {
    f, err := os.Open(filename)
    if err != nil {
        return nil, err
    }
    defer f.Close()
    var c Config
    if err := json.NewDecoder(f).Decode(&c); err != nil {
        return nil, err
    }
    return &c, nil
}
```

return the error directly

User-facing function

```
func displayConfigForUser(filename string) {
    c, err := ReadConfig(filename)
    if err != nil {
        fmt.Printf("failed: %v\n", err)
        explainError(err)
        return
    }
    fmt.Printf("%+v\n", c)
}
```

Deciding with errors before 1.13

```
func explainError(err error) {
    if err == io.ErrUnexpectedEOF {
        fmt.Println("That file ended unexpectedly.")
    } else {
        switch e := err.(type) {
        case *os.PathError: // problem reading file
            if os.IsNotExist(e) {
                fmt.Println("That file doesn't exist.")
            } else {
                fmt.Println("Something about reading that file is bad.")
            }
        case *json.SyntaxError:
            fmt.Println("Are you sure that's a JSON file?")
        }
    }
}
```

compare to “sentinel” error

type switch

predicate function

Annotating errors

```
func ReadConfig(filename string) (*Config, error) {
    f, err := os.Open(filename)
    if err != nil {
        return nil, fmt.Errorf("reading: %v", err)
    }
    defer f.Close()
    var c Config
    if err := json.NewDecoder(f).Decode(&c); err != nil {
        return nil, fmt.Errorf("decoding JSON: %v", err)
    }
    return &c, nil
}
```

add helpful information

Errorf(%v) breaks error inspection

```
func explainError(err error) {
    if err == io.ErrUnexpectedEOF {                    broken!
        fmt.Println("That file ended unexpectedly.")
    } else {
        switch e := err.(type) {                       broken!
        case *os.PathError: // problem reading file
            if os.IsNotExist(e) {                      broken!
                fmt.Println("That file doesn't exist.")
            } else {
                fmt.Println("Something about reading that file is bad.")
            }
        case *json.SyntaxError:                        broken!
            fmt.Println("Are you sure that's a JSON file?")
        }
    }
}
```

Error wrapping

We want to add information to errors for *people* without hiding them from *programs*.

Solution: *Wrap* one error inside another.

- Both messages print, as before
- The wrapped error can be retrieved

Wrapping errors in Go 1.13

```
func ReadConfig(filename string) (*Config, error) {
    f, err := os.Open(filename)
    if err != nil {
        return nil, fmt.Errorf("reading: %w", err)
    }
    defer f.Close()
    var c Config
    if err := json.NewDecoder(f).Decode(&c); err != nil {
        return nil, fmt.Errorf("decoding JSON: %w", err)
    }
    return &c, nil
}
```

add helpful information and wrap

Old ways of error inspection are still broken

```
func explainError(err error) {
    if err == io.ErrUnexpectedEOF {                    still broken
        fmt.Println("That file ended unexpectedly.")
    } else {
        switch e := err.(type) {                       still broken
        case *os.PathError: // problem reading file
            if os.IsNotExist(e) {                      still broken
                fmt.Println("That file doesn't exist.")
            } else {
                fmt.Println("Something about reading that file is bad.")
            }
        case *json.SyntaxError:                        still broken
            fmt.Println("Are you sure that's a JSON file?")
        }
    }
}
```

Use errors.Is instead of ==

```
func explainError(err error) {
    switch {
    case errors.Is(err, io.ErrUnexpectedEOF):
        fmt.Println("That file ended unexpectedly.")
    case errors.Is(err, os.ErrNotExist):
        fmt.Println("That file doesn't exist.")
    default:
    // ...



    }
}
```

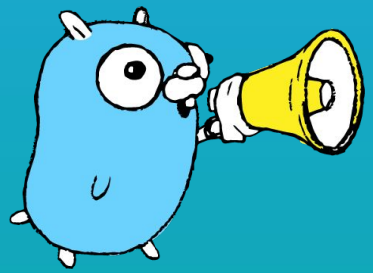
like ==, but unwraps

replaces os predicates

Use errors.As instead of type assertions

```
func explainError(err error) {
    switch {
    case errors.Is(err, io.ErrUnexpectedEOF):
        fmt.Println("That file ended unexpectedly.")
    case errors.Is(err, os.ErrNotExist):
        fmt.Println("That file doesn't exist.")
    default:
        var perr *os.PathError
        if errors.As(err, &perr) {
            fmt.Printf("Something about %s %q is bad.\n", perr.Op, perr.Path)
        }
        var jerr *json.SyntaxError
        if errors.As(err, &jerr) {
            fmt.Println("Are you sure that's a JSON file?")
        }
    }}
```

like type switch/assertion, but unwraps



An Error Story

The Details

Using Wrapping Effectively

Features That Didn’t Make It

Any error can define an Unwrap method

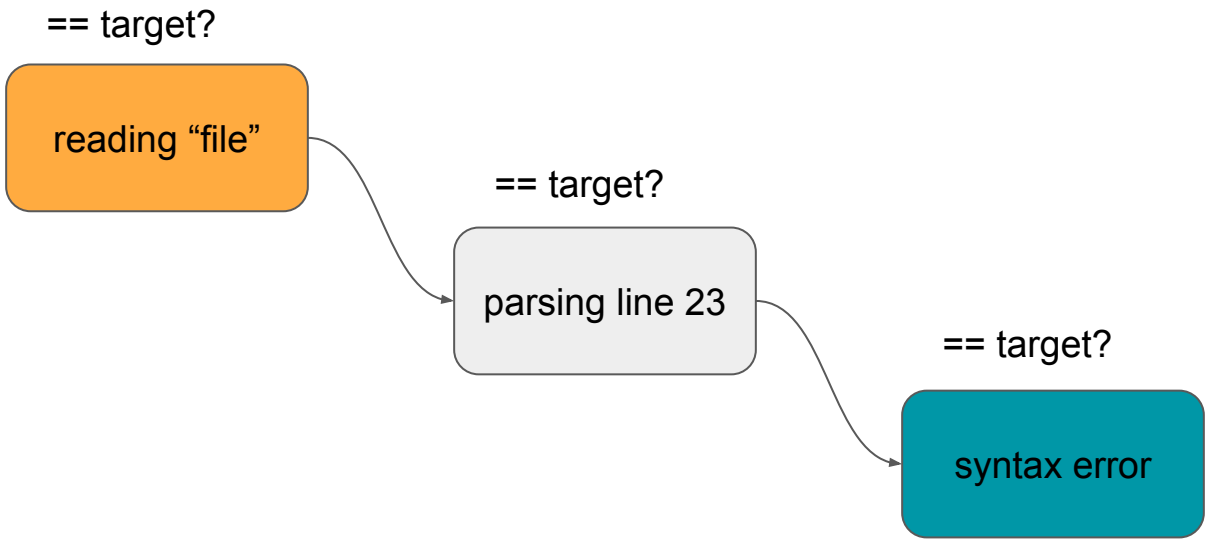
```
package os

type PathError struct {
    // ...
    Err error   exported for backward compatibility
}

func (p *PathError) Unwrap() error { return p.Err }
```

errors.Unwrap

```
func Unwrap(err error) error {
    u, ok := err.(interface { Unwrap() error })
    if !ok {
        return nil
    }
    return u.Unwrap()
}
```

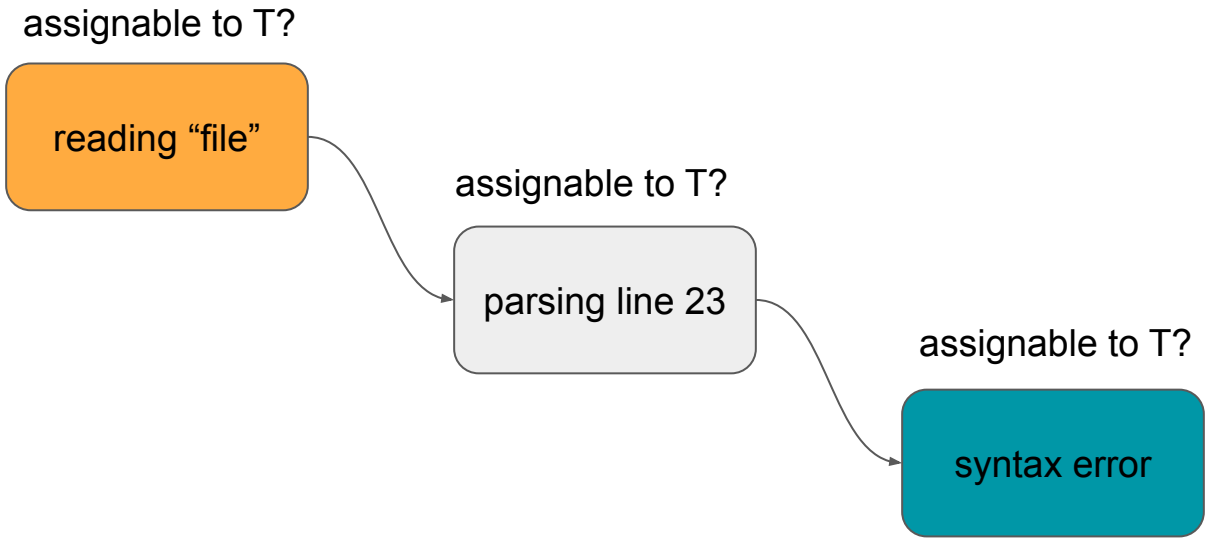
errors.Is(err, target)

Is any error in `err`'s chain equal to `target`?



errors.As(err, target)

`target` is a pointer to an error type T



Calling errors.As

```
var perr *os.PathError
if errors.As(err, &perr) {
    fmt.Println(perr.Op, perr.Path)
}
```

pass a pointer to the error type

```
perr, ok := err.(*os.PathError)
```

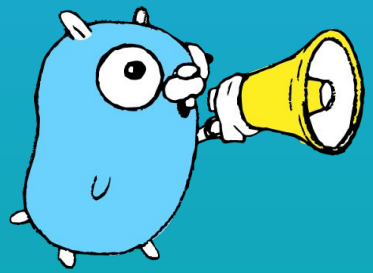
```
package os

type PathError { ... }

func (e *PathError) Error() string { ... }
```

*the error type is *PathError*

fmt.Errorf(...%w..., ...err...)

Returns an error containing `err`.

```
werr := fmt.Errorf("wrapped: %w", err)

werr.Error() == "wrapped: " + err.Error()

werr.Unwrap() == err
```



An Error Story

The Details

Using Wrapping Effectively

Features That Didn’t Make It

Moving to Wrapped Errors

Start using errors.Is/As on errors that might be wrapped.

Not necessary if the function explicitly documents the error values/types it returns (assuming backwards compatibility).

Wrapping Returned Errors

Be careful about changing

```
    return err
```

to

```
    return fmt.Errorf("...%w...", err)
```

Keep your promises

Error-checking is a notorious API-breaker

Don’t wrap pseudo-errors (io.EOF)

Wrapping Returned Errors

Always compatible to replace %v with %w

But may introduce future compatibility problems

Wrapped errors are part of your API

Did we go too far?

```
func ReadConfig(filename string) (*Config, error) {
    f, err := os.Open(filename)
    if err != nil {
        return nil, fmt.Errorf("reading: %w", err)
    }
    defer f.Close()
    var c Config
    if err := json.NewDecoder(f).Decode(&c); err != nil {
        return nil, fmt.Errorf("decoding JSON: %w", err)
    }
    return &c, nil
}
```

json.SyntaxError is now part of our API.
Do we want that?

Errors and Codes

Numeric error codes are common and useful.

```
const (
    NotFound = iota + 1
    InvalidArgument
    Unknown
)
```

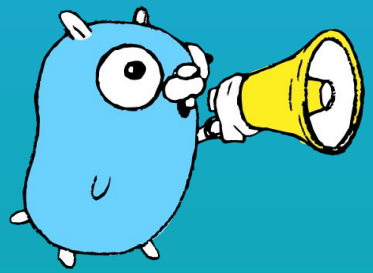
Errors and Codes

Use sentinel errors as (or with) codes.

```
var (
    NotFound = errors.New("not found")
    InvalidArgument = errors.New("invalid argument")
    Unknown = errors.New("unknown")
)
```

```
fmt.Errorf("retrieving module %s: %w", modulePath, NotFound)
```

```
if errors.Is(err, NotFound) ...
```



An Error Story

The Details

Using Wrapping Effectively

Features That Didn’t Make It

Stack trace example

```
func main() { compute(1) }

func compute(i int) { div(i, i-1) }

func div(x, y int) { fmt.Println(x / y) }
```

```
panic: runtime error: integer divide by zero
goroutine 1 [running]:
main.div(...)
    /tmp/sandbox224023570/prog.go:16
main.compute(0x1)
    /tmp/sandbox224023570/prog.go:12 +0xb5
main.main()
    /tmp/sandbox224023570/prog.go:8 +0x2a
```

Stack trace problems

Too much detail

Not enough information

Multiple copies

Goroutines

We tried...

Added to `errors.New`

But the many

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

were slow.

An Alternative to Stack Traces

Build traces manually by wrapping errors at function and goroutine boundaries.

Only for important functions.

Include arguments or other useful information.

How we don’t do it

```
func processZip(path, version string, z *zip.Reader) (*Module, error) {
    sourceInfo, err := source.ModuleInfo(path, version)
    if err != nil {
        return nil, fmt.Errorf("processZip(%q, %q): %w", path, version, err)
    }
    readmes, err := extractReadmesFromZip(path, version, z)
    if err != nil {
        return nil, fmt.Errorf("processZip(%q, %q): %w", path, version, err)

    }
    ...
```

A bit better

```
func processZip(path, version string, z *zip.Reader) (_ *Module, err error) {
    defer func() {
        if err != nil {
            err = fmt.Errorf("processZip(%q, %q): %w", path, version, err)
        }
    }()
    sourceInfo, err := source.ModuleInfo(path, version)
    if err != nil {
        return nil, err
    }
    readmes, err := extractReadmesFromZip(path, version, z)
    if err != nil {
        return nil, err
    }
    ...
```

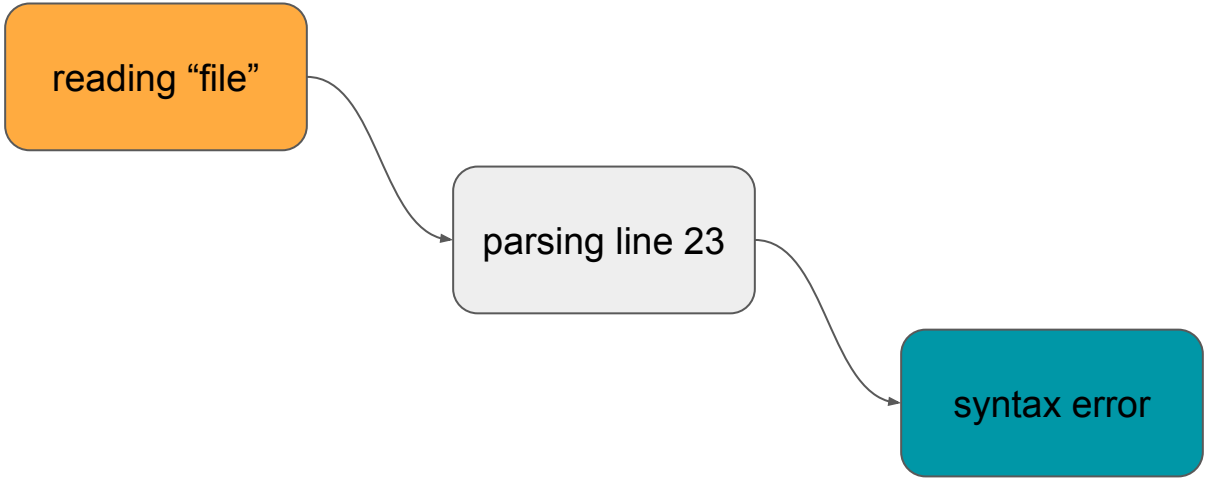
Wrap

```
func Wrap(errp *error, format string, args ...interface{}) {
    if *errp != nil {
        s := fmt.Sprintf(format, args...)
        *errp = fmt.Errorf("%s: %w", s, *errp)
    }
}
```

Execution Traces With Wrap

```
func processZip(path, version string, z *zip.Reader) (_ *Module, err error) {
    defer Wrap(&err, "processZip(%q, %q)", path, version)

    sourceInfo, err := source.ModuleInfo(path, version)
    if err != nil {
        return nil, err
    }
    readmes, err := extractReadmesFromZip(path, version, z)
    if err != nil {
        return nil, err
    }
    ...
```

Standard Formatting




```
fmt.Printf("%v", err)
```

```
reading "file": parsing line 23: syntax error
```

Detailed Formatting

```
fmt.Printf("%+v", err)
```

```
reading "file"
    cmd/prog/reader.go:122
parsing line 23
    iff x > 3 {
    cmd/prog/parser.go:85
syntax error
    cmd/prog/parser.go:214
```

Formatting Removed

Couldn’t find something simple enough.

See golang.org/x/xerrors

Detail Formatting with fmt.Formatter

```
type DetailError struct {
    msg, detail string
    err         error
}

func (e *DetailError) Unwrap() error { return e.err }

func (e *DetailError) Error() string {
    if e.err == nil {
        return e.msg
    }
    return e.msg + ": " + e.err.Error()
}
```

Detail Formatting with fmt.Formatter

```
func (e *DetailError) Format(s fmt.State, c rune) {
    if !s.Flag('+') || c != 'v' {
        fmt.Fprintf(s, spec(s, c), e.Error())
        return
    }
    fmt.Fprintln(s, e.msg)
    if e.detail != "" {    // write detail preceded by a tab
        io.WriteString(s, "\t")
        fmt.Fprintln(s, e.detail)
    }
    if e.err != nil {  // recursively handle the wrapped error
        if ferr, ok := e.err.(fmt.Formatter); ok {
            ferr.Format(s, c)
        } else {
            fmt.Fprintf(s, spec(s, c), e.err); io.WriteString(s, "\n")
}}}
```

References

Package doc: https://pkg.go.dev/errors

Blog Post: https://blog.golang.org/go1.13-errors

FAQ: https://golang.org/wiki/ErrorValueFAQ

pkg.go.dev errors package:
https://pkg.go.dev/golang.org/x/pkgsite/internal/derrors

detail formatting: https://github.com/jba/errfmt

Thank you.

jba@google.com