Go Errors

WASA: Web and Software Architecture

Enrico Bassetti

Handle errors

Errors in Go

- No “exceptions” (a.k.a. try-catch)
- No “special return values”
- No global functions for getting errors (à la PHP / C)

Errors in Go are normal values! Typed as error

Where?

Errors are always returned back as return value. The best practice is to return them as the last value in your return values set.

```
func myFunction(a, b string) (string, int, error)
```

Errors have priority!

Always handle errors!! Handle them immediately!

```go
var someString string
// ...
value, err := strconv.ParseInt(someString, 10, 64)
if err != nil {
  // What now?!?
}
```

Errors in function? Handle it!

Strategy 1: handle internally

```go
// parseOrZero returns the parsed number,
// or zero if parse is not possible
func parseOrZero(someString string) int64 {
  value, err := strconv.ParseInt(someString, 10, 64)
  if err != nil {
    return 0
  }
  return value
}
```

Errors in function? Return to caller

Strategy 2: pass the error to the caller

```go
func parseAndIncrement(strValue string) (int64, error) {
  value, err := strconv.ParseInt(strValue, 10, 64)
  if err != nil {
    // Cleanup here if necessary, or use defer
    return 0, fmt.Errorf("error parsing value: %w", err)
  }
  value++
  return value, nil
}
```

Which strategy?

Which strategy? It depends on the specific context/situation.

Sometimes you can do a mix: handle the error internally and also emit an error.

Or, you can manage some errors in your function and return others.

DON'T PANIC!

```go
func main() {
  var someString = "12345"
  value, err := strconv.ParseInt(someString, 10, 64)
  if err != nil {
    panic(err)
  }
}
```

When panic? Only when you're writing throwaway code.

Never panic in a function. Panic only in main()!

Create new errors

errors.New

```go
// ParseLatitude parses the string as latitude and
// checks the data validity (latitude range)
func ParseLatitude(str string) (Latitude, error) {
  latitude, err := strconv.ParseFloat(str, 64)
  if err != nil {
    return 0, err
  } else if latitude < -90 || latitude > 90 {
    return 0, errors.New("value out of range")
  }
  return Latitude(latitude), nil
}
```

Declare errors

```go
var ErrOutOfRange = errors.New("value out of range")

// ParseLatitude parses the string as latitude and
// checks the data validity (latitude range)
func ParseLatitude(str string) (Latitude, error) {
  latitude, err := strconv.ParseFloat(str, 64)
  if err != nil {
    return 0, err
  } else if latitude < -90 || latitude > 90 {
    return 0, ErrOutOfRange
  }
  return Latitude(latitude), nil
}
// ...continue
```

Declare errors (continued)

```go
// ...see previous slide

func main() {
  // ...
  latitude, err := ParseLatitude(latitudeInString)
  if errors.Is(err, ErrOutOfRange) {
    // Handle as invalid range
  } else {
    // Handle in another way
  }
}
```

Declaring structured errors

What if we want to pass structured info back to the caller via error?

error type is actually an interface!

```go
// builtin/builtin.go:280
type error interface {
    Error() string
}
```

New types of errors

```go
type MyError struct {
  LineNumber uint
  Message string
}

func (e *MyError) Error() string {
  return fmt.Sprintf("Error in line %d: %s",
                                   e.LineNumber, e.Message)
}

func myFunction(str string) error {
  // ...
  return &MyError{LineNumber: 2, Message: "Missing :"}
}
```