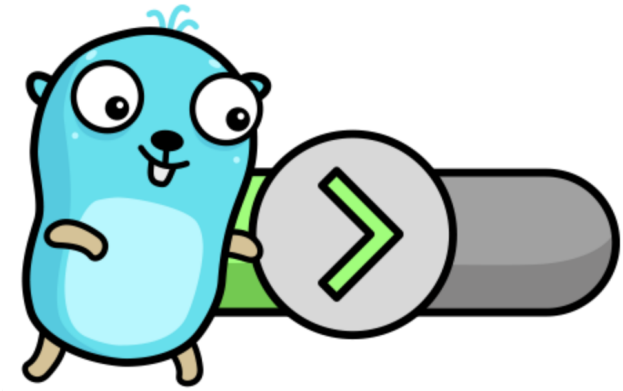
Стандартный пакет errors

В этом и последующих уроках мы рассмотрим средства работы с ошибками в стандартной библиотеке Go.



А можно меньше кода?

В прошлом уроке для выполнения [задачи "Sentinel Errors"] нам потребовалось реализовать ряд однотипных структур, удовлетворяющих интерфейсу `error`:

```
var (
    ErrAlreadyDone       error = new(AlreadyDoneError)
    ErrInconsistentData error = new(InconsistentDataError)
    ErrInvalidID         error = new(InvalidIDError)
    ErrNotFound          error = new(NotFoundError)
    ErrNotReady          error = new(NotReadyError)
)

type AlreadyDoneError struct{}
func (e *AlreadyDoneError) Error() string { return "job is already
done" }

type InconsistentDataError struct{}
func (e *InconsistentDataError) Error() string { return "job payload
is corrupted" }

type InvalidIDError struct{}
func (e *InvalidIDError) Error() string { return "invalid job id" }

type NotFoundError struct{}
```

```
func (e *NotFoundError) Error() string { return "job wasn't found"}

type NotReadyError struct{}

func (e *NotReadyError) Error() string { return "job is not ready to be performed"}
```

Мы видим, что возникает своеобразная копипаста, когда нам не нужно хранить дополнительную информацию в ошибке, а нужно лишь её значение и текст.

Поэтому хотелось бы упростить создание sentinel ошибок, что мы и сделаем в последующих задачах.

Задача "Встраивание ошибки"

Ссылка на заготовку.

Одно из первых, что приходит на ум, глядя на предыдущий шаг – это воспользоваться возможностью встраивания структур (struct embedding), а именно выделить общий тип `Err`, с помощью которого создать наши ошибки:

```
var (
    ErrAlreadyDone      error = &AlreadyDoneError{Err{"job is already done"}}
    ErrInconsistentData error = &InconsistentDataError{Err{"job payload is corrupted"}}
    ErrInvalidID        error = &InvalidIDError{Err{"invalid job id"}}
    ErrNotReady         error = &NotReadyError{Err{"job is not ready to be performed"}}
    ErrNotFound         error = &NotFoundError{Err{"job wasn't found"}}

)
```

Это вам и необходимо провернуть в рамках данной задачи.

Задача "Фабрика ошибок"

Ссылка на заготовку.

Фантазии на тему встраивания ошибок – это интересно, но слабоприменимо в реальной жизни.

На этот раз нам требуется реализовать функцию `NewError`:

```
// NewError возвращает новое значение-ошибку, текст которой является
msg.
// Две ошибки с одинаковым текстом, созданные через NewError, не
равны между собой:
//
//  NewError("end of file") != NewError("end of file")
//

func NewError(msg string) error
```

errors.New

Поздравляю, мы реализовали errors.New:

```
package errors

// New returns an error that formats as the given text.
// Each call to New returns a distinct error value even if the text
is identical.
func New(text string) error {
    return &errorString{text}
}

// errorString is a trivial implementation of error.
type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}
```

Данная функция полезна для создания ошибок-значений "in-place" и позволяет не городить однотипные структуры.

Она активно используется в стандартной библиотеке и компиляторе Go:

```
// https://github.com/golang/go master
$ grep -r "errors.New" . | wc -l
```

```
    1424
```

Давайте посмотрим на основные случаи применения.

Для создания возвращаемого "на лету" значения:

```
./src/net/smtp/auth.go:    return "", nil, errors.New("unencrypted
connection")
./src/net/smtp/auth.go:    return "", nil, errors.New("wrong host
name")
./src/net/smtp/auth.go:    return nil, errors.New("unexpected server
challenge")


./src/net/smtp/smtp.go:    return errors.New("smtp: Hello called
after other methods")
```

Для создания приватных ошибок пакета:

```
./src/net/http/transport.go:    errTooManyIdleHost =
errors.New("http: putIdleConn: too many idle connections for host")
./src/net/http/transport.go:    errCloseIdleConns = errors.New("http:
CloseIdleConnections called")
./src/net/http/transport.go:    errReadLoopExiting =
errors.New("http: persistConn.readLoop exiting")

./src/net/http/transport.go:    errIdleConnTimeout =
errors.New("http: idle connection timeout")
```

Для создания публичных ошибок пакета (sentinel errors):

```
./src/database/sql/sql.go:    var ErrNoRows = errors.New("sql: no
rows in result set")
./src/database/sql/sql.go:    var ErrTxDone = errors.New("sql:
transaction has already been committed or rolled back")

./src/strconv/atoi.go:    var ErrRange = errors.New("value out of
range")

./src/strconv/atoi.go:    var ErrSyntax = errors.New("invalid
syntax")
```

В тестах:

```
./src/encoding/json/decode_test.go:
```

```
    {in: `{"B": "False"}`, ptr: new(B),
     err: errors.New(`json: invalid use of ,string struct tag, trying
to unmarshal "False" into bool`)},

./src/encoding/json/decode_test.go:
    {in: `{"B": "nul"}`, ptr: new(B),
     err: errors.New(`json: invalid use of ,string struct tag, trying
to unmarshal "nul" into bool`)},

./src/encoding/json/decode_test.go:
    {in: `{"B": [2, 3]}`, ptr: new(B),
     err: errors.New(`json: invalid use of ,string struct tag, trying
to unmarshal unquoted value into bool`)},

./src/net/http/transport_test.go:    fakeErr := errors.New("fake
error")
./src/net/http/transport_test.go:    errValue := errors.New("specific
error value")

./src/net/http/transport_test.go:    var errFakeRoundTrip =
errors.New("fake roundtrip")
```

А вот как выглядят sentinel errors здорового человека:

```
var (
    ErrAlreadyDone       = errors.New("job is already done")
    ErrInconsistentData = errors.New("job payload is corrupted")
    ErrInvalidID         = errors.New("invalid job id")
    ErrNotFound          = errors.New("job wasn't found")
    ErrNotReady          = errors.New("job is not ready to be
performed")

)
```

Тест "Какие ошибки существуют?"

Выберите ошибки, которые существуют в стандартной библиотеке Golang и созданы через `errors.New`.

А с какими ошибками сталкиваетесь вы в повседневной разработке на Go?

- [ ] context.Canceled
- [ ] context.DeadlineExceeded
- [ ] io.EOF
- [ ] context.TimeoutArrived
- [ ] io.ErrUnexpectedEOF
- [ ] os.PathError
- [ ] os.LinkError
- [ ] http.ErrHandlerTimeout
- [ ] http.ErrBodyNotAllowed
- [ ] sql.ErrNoRows
- [ ] sql.ErrTooMuchRows
- [ ] exec.ErrProcessTerminated
- [ ] io.EndOfFile
- [ ] flag.ErrHelp
- [ ] bufio.ErrFinalToken

fmt.Errorf

Для создания форматированных ошибок нам пригодится функция fmt.Errorf:

```
// Errorf formats according to a format specifier and returns the
string as a
// value that satisfies error.

func Errorf(format string, a ...interface{}) error
```

Она работает подобно fmt.Sprintf, только возвращает ошибку:

```
// https://goplay.tools/snippet/B4TE85AhzrV

func main() {
    const name, id = "Anthony", 24

    var err error = fmt.Errorf("user %q (id %d) not found", name, id)
    fmt.Println(err.Error()) // user "Anthony" (id 24) not found
}
```

В стандартной библиотеке огромное количество примеров использования `fmt.Errorf`:

```
// https://github.com/golang/go master
$ grep -rE "fmt.Errorf" . | wc -l
    1947
```

Например,

```
fmt.Errorf("%s is not supported on %s/%s", network, runtime.GOOS, runtime.GOARCH)

fmt.Errorf("request failed: %v", err)

fmt.Errorf("failed to write data to file: %v", err)

fmt.Errorf("args=%v out=%q err=%v ", cmd.Args, string(out), err)

fmt.Errorf("optional header size(%d) is less minimum size (%d) of PE32 optional header", sz, oh32MinSz)

fmt.Errorf("illegal paragraph embedding level: %d", embeddingLevel)

fmt.Errorf("last linebreak was %d, want %d", prev, textLength)

fmt.Errorf("invalid slice index: %d > %d", idx[1], idx[2])


// и т.д.
```

Из списка выше можно выделить

```
fmt.Errorf("request failed: %v", err)
fmt.Errorf("failed to write data to file: %v", err)

fmt.Errorf("args=%v out=%q err=%v ", cmd.Args, string(out), err)
```

так как здесь в создании ошибки участвует другая ошибка `err`.

Таким образом, мы получаем новую ошибку, в которой присутствует текст старой ошибки. Данный приём является **врапингом ошибки в текст** (обогащение ошибки контекстом), подробнее о котором мы поговорим в следующем уроке.

Обычно для форматирования ошибки (получения её строкового представления) используют один из [спецификаторов](#) (**fmt verbs**):

- `%v` – the value in a default format;
- `%s` – the uninterpreted bytes of the string or slice.

Получаем, что следующие операции эквивалентны:

```
// https://goplay.tools/snippet/mY02749wiMO
import (
    "fmt"
    "io"
)

func main() {
    for _, err := range []error{
        fmt.Errorf("request failed: %v", io.EOF),
        fmt.Errorf("request failed: %v", io.EOF.Error()),
        fmt.Errorf("request failed: %s", io.EOF),
        fmt.Errorf("request failed: %s", io.EOF.Error()),
    } {
        fmt.Println(err)
    }
}

/*
request failed: EOF
request failed: EOF
request failed: EOF
request failed: EOF

*/
```

Метод `Error` необязательно вызывать благодаря [поддержке](#) форматирования ошибок в пакете `fmt`:

```
switch verb {
case 'v', 's', 'x', 'X', 'q':
    // Is it an error or Stringer?
    // ...
    switch v := p.arg.(type) {
    case error:
        handled = true
        defer p.catchPanic(p.arg, verb, "Error")
        p.fmtString(v.Error(), verb)
        return

    // ...
    }

}
```

Но хорошим тоном будет использование одного из двух вариантов:

```
fmt.Errorf("request failed: %v", io.EOF)          // Наиболее
встречаемый вариант.

fmt.Errorf("request failed: %s", io.EOF.Error()) // Избыточно, но мы
явно подчёркиваем

                                                  // форматирование
ошибочной строки.
```

Задача "Безопасный факториал"

Ссылка на заготовку.

Вам необходимо реализовать функцию `Calculate`, которая рекурсивно считает факториал входного числа, но дополнительно проверяет его валидность:

```
const maxDepth = 256

var (
    ErrNegativeN = ...
    ErrTooDeep   = ...
)

// Calculate рекурсивно считает факториал входного числа n.
// Если число меньше нуля, то возвращается ошибка ErrNegativeN.
```

```
// Если для вычисления факториала потребуется больше maxDepth
фреймов, то Calculate вернёт ErrTooDeep.
func Calculate(n int) (int, error) {
    // ...

}
```

В тексте ошибки `ErrTooDeep` должна присутствовать константа `maxDepth`.

Здесь и далее уже можно пользоваться функциями `errors.New` и `fmt.Errorf` (и не только), с которыми мы познакомились в предыдущих шагах.

Промежуточные итоги

Мы познакомились со стандартном пакетом `errors` и научились создавать с его помощью простейшие ошибки.

Пришло время рассмотреть понятие **врапинга ошибок** в Go.