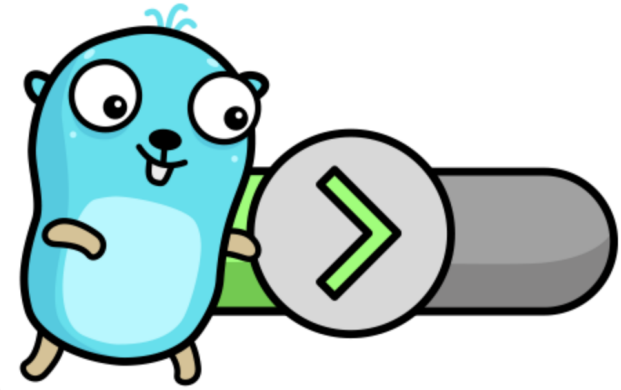
Про метод Error

В этом уроке мы поговорим о том, почему не стоит пользоваться методом `Error` для сравнения ошибок и что делать, если без этого всё-таки не обойтись.



Не используйте Error для определения сущности ошибки

Мы уже неоднократного говорили (и ещё будем говорить), что возвращаемая методом `Error` строка

```
package builtin

// The error built-in interface type is the conventional interface
for
// representing an error condition, with the nil value representing
no error.
type error interface {
    Error() string
}
```

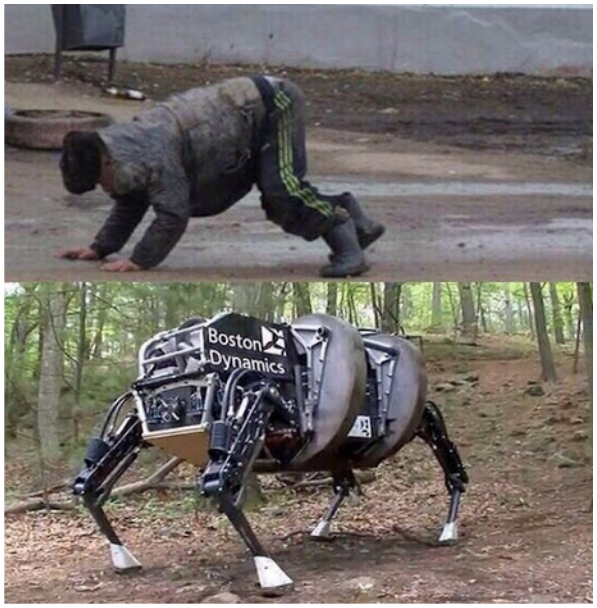
предназначена для **использования людьми вне программного кода** (дебаг, просмотр логов и т.д.) и не стоит строить обработку ошибок поверх неё.

Чем чреват подобный код?

```
if err != nil {
```

```
    if err.Error() == "unexpected EOF" {
        // Что-то делаем.
    }
    return nil, err
}
```

1. Оборачивание ошибки (в текст / в другую ошибку) ломает данную проверку.
2. Изменение текста ошибки (например, исправление опечатки) ломает данную проверку.
3. **Вы не можете гарантировать, что работаете с ошибкой из нужного вам пакета** (а не с ошибкой из другого пакета, имеющей такой же текст).



Что делать, если без сравнения с текстом ошибки ну никак не обойтись? Узнаем дальше.

Тест "Коварный Error"

К каким изменениям из списка ниже **устойчив** данный код?

```
var ErrInvalidWrite = errors.New("invalid write result")

func DoSomething() error {
    return ErrInvalidWrite
}

// ...

if err := DoSomething(); err != nil {
    if strings.Contains(err.Error(), "invalid write result") { // <-
!!!
        // ...
    }
    return nil, err
}
```

1)

```
func DoSomething() error {
    return fmt.Errorf("cannot write: %v", ErrInvalidWrite) // Врапим
ошибку через %v.

}
```

2)

```
var ErrInvalidWriteToFile = errors.New("invalid write result")

func DoSomething() error {
    return ErrInvalidWriteToFile // Возвращаем другую ошибку, но с
таким же текстом.

}
```

3)

```
func DoSomething() error {
    return fmt.Errorf("cannot write: %w", ErrInvalidWrite) // Врапим
ошибку через %w.

}
```

4)

```
var ErrInvalidWrite = errors.New("invalid writte result") // В какой-то момент появилась опечатка `tt`.
```

5)

```
type OperationError struct {
    op  string
    err error
}

func (o *OperationError) Error() string {
    return "cannot do " + o.op
}

func (o *OperationError) Unwrap() error {
    return o.err
}

func DoSomething() error {
    return &OperationError{op: "write", err: ErrInvalidWrite} // Врапим ошибку в другой тип.

}
```

Выберите все подходящие ответы из списка

1)
2)
3)
4)
5)

Hyrum's Law (The Law of Implicit Interfaces)

Закон Хайрама (закон неявных интерфейсов) гласит, что

With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviours of your system will be depended on by somebody.

Т.е. при большом количестве потребителей нашего API, все действия нашего API (даже если они не являются частью публичного контракта) в конечном итоге станут кем-то зависимыми.

Например, возвращаясь к предыдущему шагу: меняя опечатку в сообщении sentinel ошибки вашего пакета, будьте готовы, что-то кто-то придёт и скажет, что у него перестала работать регулярка на текст этой ошибки, и вы сломали так называемую обратную совместимость вашего API.

Мы считаем, что данный закон не повод класть болт на лучшие гошные практики, и подобных товарищей следует просить пересаживаться с текста ошибки на православную обработку ошибок с использованием `errors.Is` / `errors.As`.

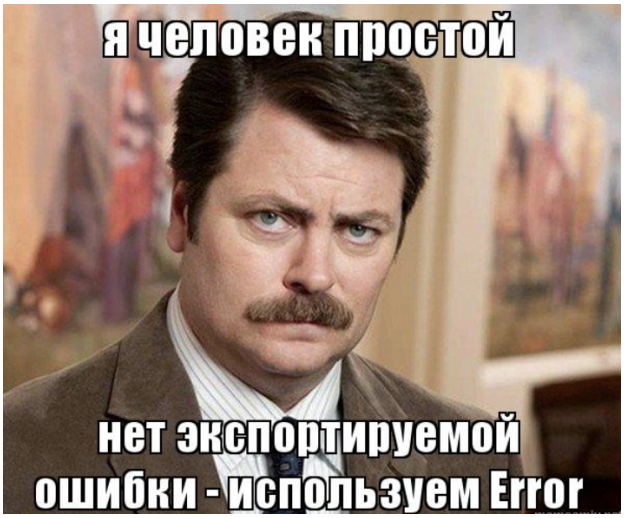
Если же им не хватает какой-то специфичной ошибки от нас, то можно вполне договориться и предоставить её (тем более , как часто бывает, может ошибка уже и есть, просто является приватной).

Пишите в комментариях, что думаете об этом :)

P.S. Hyrum Wright – инженер из Google, работающий над инструментами и инфраструктурой для крупномасштабного изменения кода.

P.P.S. Прочие интересные законы представлены здесь.

Когда без Error всё-таки не обойтись



Бывает такое, что вам необходимо заложиться на специфичную ошибку, а сторонний модуль её не экспортирует или ошибка не является sentinel в принципе.

Чаще всего подобное случается при работе с "сырыми" низкоуровневыми вызовами внешней системы, например, вызовами базы данных, очереди сообщений и т.д.

В таком случае, вопреки лучшим практикам, единственным выходом является работа с результатом `Error`.

Посмотрим на [боевой пример](#) подобного в модуле для работы с [Redis](#) – **github.com/go-redis/redis**:

```
package redis

type Scripter interface {
    Eval(ctx context.Context, script string, keys []string, args
...interface{}) *Cmd
    EvalSha(ctx context.Context, sha1 string, keys []string, args
...interface{}) *Cmd
    // ...
}

type Script struct {
    src, hash string
}
```

```
func NewScript(src string) *Script {
    h := sha1.New()
    // ...
    return &Script{
        src:  src,
        hash: hex.EncodeToString(h.Sum(nil)),
    }
}

// Run optimistically uses EVALSHA to run the script. If script does
not exist
// it is retried using EVAL.
func (s *Script) Run(ctx context.Context, c Scripter, keys []string,
args ...interface{}) *Cmd {
    r := s.EvalSha(ctx, c, keys, args...)
    if err := r.Err(); err != nil && strings.HasPrefix(err.Error(),
"NOSCRIPT ") {
        return s.Eval(ctx, c, keys, args...)
    }
    return r
}

func (s *Script) Eval(ctx context.Context, c Scripter, keys []string,
args ...interface{}) *Cmd {
     return c.Eval(ctx, s.src, keys, args...)
}

func (s *Script) EvalSha(ctx context.Context, c Scripter, keys
[]string, args ...interface{}) *Cmd {
     return c.EvalSha(ctx, s.hash, keys, args...)

}
```

Вы можете создать скрипт и запустить его с помощью метода `Run` через сторонний запускатор скриптов `Scripter` (обычно это просто обёртка над командами в редиску).

При этом `Run` пытается сначала выполнить скрипт по его хэшу (используя метод `s.EvalSha`, который транслируется в одноимённую команду в Redis), а если редиска ругается, что такого скрипта нет, то `Run` считает, что перед ним не хэш загруженного скрипта, а полноценных исходный Lua код, и пытается выполнить его через s.Eval.

Проблема в том, что в `Run` нет возможности получить ошибку вида `ErrNoScript`, так как вызов редиски абсолютно за пределами текущего гошного кода. И всё, что нам остаётся – это работать с текстом, возвращаемым как результат выполнения команды в Redis.

Если повторить данную операцию руками, то можно увидеть, что это за текст:

```
127.0.0.1:6379> evalsha d0badc1a97a25214df75d50b089333a4aef087cd 0

(error) NOSCRIPT No matching script. Please use EVAL.
```

Что соответствует условию в гошном коде выше:

```
if err := r.Err(); err != nil && strings.HasPrefix(err.Error(),
"NOSCRIPT ") {
    return s.Eval(ctx, c, keys, args...)

}
```

Пользуемся opaque errors

Если сравнение с результатом `Error` неизбежно, то наиболее поддерживаемым вариантом будет вынести это дело в соответствующую вспомогательную функцию:

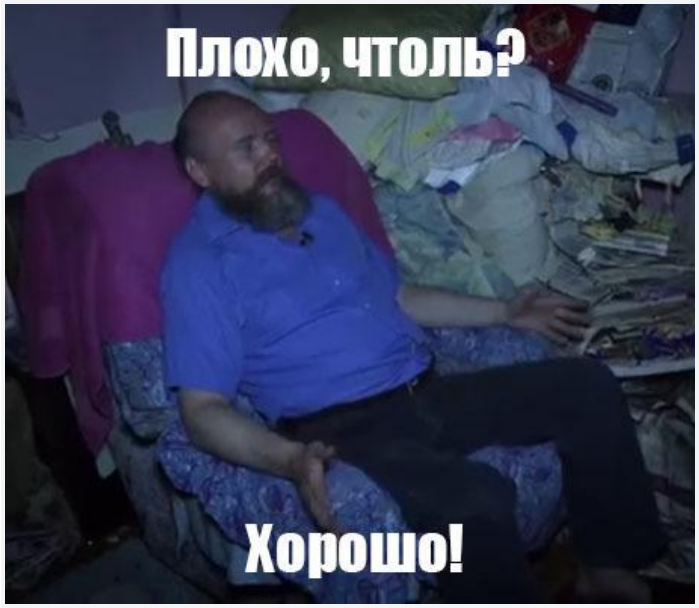
```
// Плохо.
if err := r.Err(); err != nil && strings.HasPrefix(err.Error(),
"NOSCRIPT ") {
    return s.Eval(ctx, c, keys, args...)

}

// Хорошо.
if err := r.Err(); isNoScriptError(err) {
    return s.Eval(ctx, c, keys, args...)
}

func isNoScriptError(err error) bool {
    return err != nil && strings.HasPrefix(err.Error(), "NOSCRIPT ")

}
```



Давайте посмотрим на парочку подобных примеров.

```
import "github.com/go-redis/redis/v8"

// ...

func isNoSuchKeyError(err error) bool {
    return err != nil && strings.HasPrefix(err.Error(), "ERR no such
key")

}
```

```
import (
    "fmt"
    "github.com/go-pg/migrations/v8"
)

// ...
    _, _, err := migrations.Run(s.opts.dbClient, "init")
    if !isMigrationsAlreadyExistsError(err) {
        return fmt.Errorf("cannot init migration: %v", err)
    }
// ...

func isMigrationsAlreadyExistsError(err error) bool {
```

```
    return err != nil && strings.Contains(err.Error(), `relation
"gopg_migrations" already exists`)

}
```

```
import "github.com/go-pg/pg/v10"

func IsDuplicateIntegrityViolationError(err error) bool {
    var pgErr pg.Error
    return errors.As(err, &pgErr) &&
        pgErr.IntegrityViolation() &&
        strings.Contains(pgErr.Error(), "duplicate")
}
```

или вариант выше можно переписать в более чистом виде

```
import "github.com/go-pg/pg/v10"

// duplicateKeyErrorID содержит код ошибки PSQL вида
//  ERROR #23505 duplicate key value violates unique constraint
"idx_users_on_email"
const duplicateKeyErrorID = "23505"

func IsDuplicateIntegrityViolationError(err error) bool {
    var pgErr pg.Error
    return errors.As(err, &pgErr) && (pgErr.Field('C') ==
duplicateKeyErrorID)

}
```

Задача "Скрытые ошибки text/template"

Ссылка на заготовку.

Разработчику потребовалось полагаться на специфичные ошибки, возвращаемые функциями (*Template).Parse и (*Template).Execute.

Его интересуют следующие ошибки:

- Ошибка использования в шаблоне приватного поля структуры

```
`template: example:1:3: executing "example" at <.name>: name is an unexported field of ...`
```

- Ошибка вызова в шаблоне неизвестной функции

```
`template: example:1: function "XXX" not defined`
```

При этом `Parse` в принципе не возвращает ошибок, с которыми можно работать (все ошибки создаются налету через `fmt.Errorf`), а `Execute` возвращает экземпляр типа `ExecError`, но по своей сути он тоже бесполезен (так как является обёрткой над `fmt.Errorf`-ошибками):

```
package template

// ExecError is the custom error type returned when Execute has
// an error evaluating its template.
type ExecError struct {
    Name string // Name of template.
    Err  error  // Pre-formatted error.
}

func (e ExecError) Error() string {
    return e.Err.Error()
}

func (e ExecError) Unwrap() error {
    return e.Err

}
```

Ничего не остаётся, как работать с `Error`. Что вам и необходимо сделать, реализовав следующие функции:

```
// IsFunctionNotDefinedError говорит, является ли err ошибкой неопределённой в шаблоне функции.
func IsFunctionNotDefinedError(err error) bool {}

// IsExecUnexportedFieldError говорит, является ли err template.ExecError,
// а именно ошибкой использования неэкспортируемого поля структуры.

func IsExecUnexportedFieldError(err error) bool {}
```

Имейте в виду, что входящие ошибки могут быть завраплены.

Задача "Docker Error"

Ссылка на заготовку.

Перед вами тип `Docker`, предоставляющий API для работы с докером через сторонний `Executor`:

```
package docker

import "context"

type Executor interface {
    Exec(ctx context.Context, cmd string, args ...any) error
}

type Docker struct{}

func (d *Docker) RunContainer(ctx context.Context, e Executor, image string) error {
    if err := e.Exec(ctx, "run", image); err != nil {
        return newDockerError(err)
    }
    return nil
}

func (d *Docker) StopContainer(ctx context.Context, e Executor, containerID string) error {
    if err := e.Exec(ctx, "stop", containerID); err != nil {
        return newDockerError(err)
```

```
    }
    return nil
}

func (d *Docker) ExecContainerCmd(ctx context.Context, e Executor,
containerID, cmd string) error {
    if err := e.Exec(ctx, "exec", containerID, cmd); err != nil {
        return newDockerError(err)
    }
    return nil

}
```

Проблема в том, что `Executor` возвращает "сырые" ошибки, являющимися примитивной обёрткой над текстом ошибок, которые возвращает непосредственно демон.

Примеры таких ошибок:

```
$ docker run rabbitmq:4-management
docker: Error response from daemon: pull access denied for
rabbitmq:4-management

$ docker stop unknown
Error response from daemon: No such container: unknown

$ docker exec -it 7aeae4613083 /bin/bash

Error response from daemon: Container 7aeae4613083 is not running
```

Вам необходимо реализовать тип `Error`, предоставляющий более приятное API к докерным ошибкам, обозначенным выше:

```
dockerErr := errors.New(`Error response from daemon: pull access
denied for rabbitmq:4-management`)
err := newDockerError(dockerErr)

// Error response from daemon: pull access denied for
rabbitmq:4-management
fmt.Println(err.Error())
// true

fmt.Println(err.IsPullAccessDeniedError())
```

Подробности в заготовке задачи и тестах.