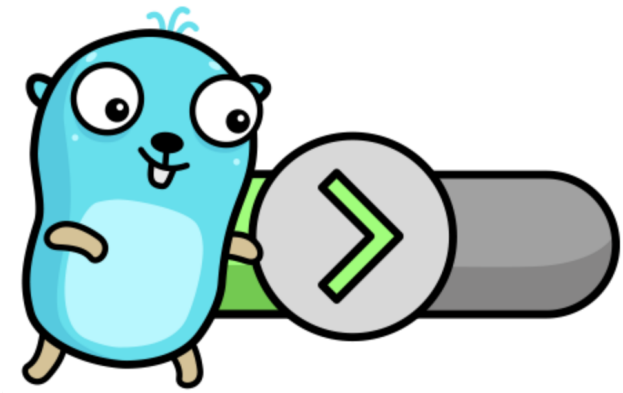
Прочие нестандартные модули

Уже на момент создания курса мы наблюдаем большое количество нестандартных модулей по работе с ошибками, и наверняка сообщество Go сделает ещё много новых.

Охватить и пройтись по всем мы не сможем, да это и не имеет смысла. Поэтому мы познакомим вас с ещё только парочкой библиотек, заслуживающих внимание и полезных в production-ready разработке на Go.



github.com/hashicorp/go-multierror

Популярная библиотека с небольшим API, позволяющая объединять несколько ошибок в одну:

```
var result error

if err := step1(); err != nil {
    result = multierror.Append(result, err)
}
if err := step2(); err != nil {
    result = multierror.Append(result, err)
}


return result
```

Например (тыц):

```
import (
    // ...
    "github.com/hashicorp/go-multierror"
    // ...
)

func main() {
    if err := mainImpl(); err != nil {
        log.Fatalf("silk-teardown error: %s", err)
    }
}

func mainImpl() error {
    // ...
    client := controller.NewClient(logger, httpClient,
cfg.ConnectivityServerURL)

    var errList error
    if err := client.ReleaseSubnetLease(cfg.UnderlayIP); err != nil {
        errList = multierror.Append(errList, fmt.Errorf("release
subnet lease: %s", err))
    }

    vtepFactory := &vtep.Factory{NetlinkAdapter:
&adapter.NetlinkAdapter{}}
    if err := vtepFactory.DeleteVTEP(cfg.VTEPName); err != nil {
        errList = multierror.Append(errList, fmt.Errorf("delete vtep:
%s", err))
    }

    return errList
}
```

Для мульти-ошибки работают `errors.Is` / `errors.As` по входящим в неё ошибкам, а также мы можем определить формат её вывода (исходник примера):

```
// https://goplay.tools/snippet/qqKz6XIKRpO

func main() {
    err1 := errors.New("an error 1")
    err2 := errors.New("an error 2")
```

```
    err := multierror.Append(io.EOF, err1, err2)

    fmt.Println(errors.Is(err, io.EOF)) // true
    fmt.Println(errors.Is(err, err1))   // true
    fmt.Println(errors.Is(err, err2))   // true
    fmt.Println()

    fmt.Println(err)
    /*
        3 errors occurred:
                * EOF
                * an error 1
                * an error 2
    */

    err.ErrorFormat = func(errors []error) string {
        var b strings.Builder
        b.WriteString("MY ERRORS:\n")
        for _, err := range errors {
            b.WriteString("\t - " + err.Error() + "\n")
        }
        return b.String()
    }
    fmt.Println(err)
    /*
        MY ERRORS:
                - EOF
                - an error 1
                - an error 2
    */

}
```

Где **hashicorp/go-multierror** может пригодиться:

- валидация данных: когда полезно накопить несколько ошибок, а не прерывать проверку на первом же невалидном поле;

- грамотный teardown приложения: когда при ошибке завершения работы одного из компонентов системы не нужно завершать процесс, а нужно дать остановиться остальным компонентам;

- ваши варианты?

Тест "hashicorp/go-multierror: Gotcha 1 – Неудобное API"

Ссылка на пример.

```
// https://goplay.tools/snippet/WyXyJBW_5pk
package main

import (
    "fmt"

    "github.com/hashicorp/go-multierror"
)

func main() {
    if err := collectErrors(); err != nil {
        fmt.Println("errors!")
    } else {
        fmt.Println("ok")
    }
}

func collectErrors() error {
    var err error
    err = multierror.Append(err, foo())
    err = multierror.Append(err, bar())
    return err
}

func foo() error {
    return nil
}
```

```
func bar() error {
    return nil
}
```

Запустив код выше, к своему удивлению разработчик увидел на экране

```
errors!
```

Исходя из этого выберите верные утверждения ниже.

Выберите все подходящие ответы из списка

- ☐ multierror.Append без труда "глотает" nil-ошибки и возвращает в таком случае error(nil)
- ☐ При вызове multierror.Append нужно удостовериться, что мы добавляем не-nil ошибку
- ☐ Возвращаемое multierror.Append значение имеет тип error
- ☐ Следующий код способен починить функцию collectErrors:

```
if err.(*multierror.Error).Len() == 0 {
    return nil
}
return err
```

Тест "hashicorp/go-multierror: Gotcha 2 – Старые песни о главном"

Ссылка на пример.

```
// https://goplay.tools/snippet/_4IDnGdggRV
package main

import (
    "fmt"
```

```
    "github.com/hashicorp/go-multierror"
)

func main() {
    if err := collectErrors(); err != nil {
        fmt.Println("errors!")
    } else {
        fmt.Println("ok")
    }
}

func collectErrors() error {
    var mErr *multierror.Error

    if err := foo(); err != nil {
        mErr = multierror.Append(mErr, err)
    }

    if err := bar(); err != nil {
        mErr = multierror.Append(mErr, err)
    }

    if mErr != nil {
        mErr.ErrorFormat = func(errors []error) string {
            return fmt.Sprintf("%v", errors)
        }
    }

    return mErr
}

func foo() error {
    return nil
}

func bar() error {
    return nil
}
```

Запустив код выше, к своему удивлению разработчик увидел на экране

```
errors!
```

Вам нужно исправить одну строчку в функции `collectErrors` так, чтобы она заработала верно. Программа при этом должна компилироваться!

Исправленную строчку необходимо вставить как ответ.

Тест "hashicorp/go-multierror: Gotcha 3 – Будьте бдительны"

Ссылка на пример.

```
// https://goplay.tools/snippet/IKRt3WqWned
package main

import (
    "errors"
    "fmt"

    "github.com/hashicorp/go-multierror"
)

func main() {
    if err := collectErrors(); err != nil {
        fmt.Println("errors!", err)
    } else {
        fmt.Println("ok")
    }
}

func collectErrors() error {
    var err error

    if err := foo(); err != nil {
        err = multierror.Append(err, err)
    }

    if err := bar(); err != nil {
        err = multierror.Append(err, err)
    }

    return err
}
```

```
func foo() error {
    return errors.New("error from foo")
}

func bar() error {
    return errors.New("error from bar")
}
```

Запустив код выше, к своему удивлению разработчик увидел на экране

```
ok
```

т.е. обратную предыдущим примерам ситуацию.

Быстро обнаружив проблему, он переписал кусок функции следующим образом:

```
if err = foo(); err != nil {
    err = multierror.Append(err, err)
}

if err = bar(); err != nil {
    err = multierror.Append(err, err)

}
```

Что теперь выведет его программа?

github.com/uber-go/multierr

Интересная и лаконичная библиотека от компании Uber, не пытающаяся быть комбайном для обработки ошибок. По сути это сестрёнка **hashicorp/go-multierror**, но гораздо менее популярная.

API у неё тоже небольшое:

- `Append(left error, right error) error`
- `AppendInto(into *error, err error) (errored bool)`

- `AppendInvoke(into *error, invoker Invoker)`
- `Combine(errors ...error) error`
- `Errors(err error) []error`

---

multierr.Combine и multierr.Append

```
package multierr // import "go.uber.org/multierr"

// Combine combines the passed errors into a single error.
func Combine(errors ...error) error { /* ... */ }

// Append appends the given errors together. Either value may be nil.
//
// This function is a specialization of Combine for the common case where
// there are only two errors.

func Append(left error, right error) error { /* ... */ }
```

Не будем подробно останавливаться на данных функциях – это своего рода аналоги `multierror.Append` из **hashicorp/go-multierror** и `errors.CombineErrors` из **cockroachdb/errors**.

Ключевыми отличиями являются:

- Более приятные сигнатуры функций.
- Combine умеет "пропускать" `nil`-ошибки.
- Возвращаемая мульти-ошибка умеет сама красиво форматироваться и поддерживает `errors.Is` и `errors.As`. для входящих в неё ошибок.

```
// Пример от авторов библиотеки.
err := multierr.Combine(
    reader.Close(),
    writer.Close(),
    pipe.Close(),
```

```
)
```

---

multierr.AppendInvoke

Invoke-функциональность – одна из особенностей **uber-go/multierr**, которой нет в других библиотеках.

Ставьте лайк, если надоели инспекции в IDE, ругань линтеров или муки совести из-за заигноренных ошибок в конструкциях вроде таких:

```
tx, err := s.database.Begin(ctx)
if err != nil {
    return err
}
defer func() {
    _ = tx.Rollback(ctx) // Явное игнорирование ошибки.
}()

// ...

f, err := os.Open(path)
if err != nil {
    return err
}
defer f.Close() // Неявное игнорирование ошибки.
```

uber-go/multierr позволяет почти элегантно решить эту проблему, цепляя вторичную ошибку от cleanup-операции к основной ошибке (исходник примера):

```
// processFile - пример штатного использования multierr.AppendInvoke.
// Обратите внимание, что multierr.AppendInvoke используется вместе с
// именованным возвращаемым аргументом err.
func processFile(path string) (err error) {
    f, err := os.Open(path)
    if err != nil {
        return fmt.Errorf("open file: %v", err)
    }
    defer multierr.AppendInvoke(&err, multierr.Close(f))

    scanner := bufio.NewScanner(f)
```

```
    defer multierr.AppendInvoke(&err, multierr.Invoke(scanner.Err))

    for scanner.Scan() {
        fmt.Println(scanner.Text())
    }
    return nil

}
```

В примере выше мы видим:

```
package multierr // import "go.uber.org/multierr"

// Invoker is an operation that may fail with an error. Use it with
// AppendInvoke to append the result of calling the function into an
error.
// This allows you to conveniently defer capture of failing
operations.
type Invoker interface {
    Invoke() error
}

// AppendInvoke appends the result of calling the given Invoker into
the
// provided error pointer. Use it with named returns to safely defer
// invocation of fallible operations until a function returns, and
capture the
// resulting errors.
func AppendInvoke(into *error, invoker Invoker) {
    AppendInto(into, invoker.Invoke())
}

// AppendInto appends an error into the destination of an error
pointer and
// returns whether the error being appended was non-nil.
func AppendInto(into *error, err error) (errored bool) {
    if into == nil {
        panic("misuse of multierr.AppendInto: into pointer must not
be nil")
    }

    if err == nil {
        return false
    }
    *into = Append(*into, err)
```

```
    return true

}
```

Ничего сложного :)

Также есть прекрасный **функциональный тип**, позволяющий легко приводить ваши функции к интерфейсу `multierr.Invoker`:

```
package multierr // import "go.uber.org/multierr"

type Invoke func() error

// Invoke calls the supplied function and returns its result.

func (i Invoke) Invoke() error { return i() }
```

И хелпер на его основе (использование которого мы видели выше) для типов, реализующих `io.Closer`:

```
package multierr // import "go.uber.org/multierr"

// Close builds an Invoker that closes the provided io.Closer. Use it
with
// AppendInvoke to close io.Closers and append their results into an
error.
func Close(closer io.Closer) Invoker {
    return Invoke(closer.Close)

}
```

Тест "uber-go/multierr: named return"

Ссылка на пример.

Что может пойти не так при использовании `multierr.AppendInvoke`, если вызывающая функция не будет возвращать `err` как именованный аргумент?

```
func bad() error {
    // ...
    err := getError()
    defer multierr.AppendInvoke(&err, multierr.Close(closer))
```

```
    return err
}
```

На всякий случай попробуйте самостоятельно написать и так и так, чтобы увидеть разницу.

А затем выберите верные утверждения.

Выберите все подходящие ответы из списка

- ☐ Отложенные (deferred) функции не могут модифицировать неименованные возвращаемые аргументы.
- ☐ Отложенные (deferred) функции могут модифицировать именованные возвращаемые аргументы.
- ☐ Именованный возвращаемый аргумент находится вне зоны видимости отложенной (deferred) функции.
- ☐ Объявленная в теле функции переменная находится вне зоны видимости отложенной (deferred) функции.
- ☐ При возврате объявленной в теле функции переменной происходит её копирование в возвращаемый аргумент.
- ☐ Оператор return устанавливает результаты функций перед выполнением любых отложенных (deferred) функций.

Тест "uber-go/multierr: вычисление аргументов AppendInvoke"

Ссылка на пример.

Для каких целей функция `multierr.AppendInvoke` вторым аргументом принимает интерфейс `multierr.Invoker` (по сути –функцию, возвращающую ошибку) вместо, например, явной ошибки?

```
// Хорошо.
defer multierr.AppendInvoke(&err, multierr.Close(c))

// Плохо.
```

```
defer multierr.AppendInto(&err, c.Close())
```

Напомним, что

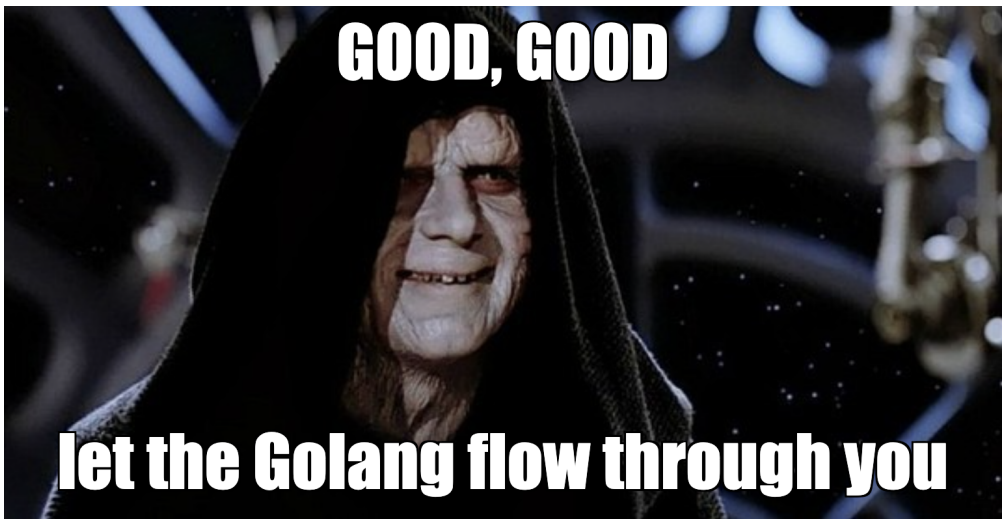
```
func AppendInvoke(into *error, invoker Invoker) {
    AppendInto(into, invoker.Invoke())

}
```

На всякий случай попробуйте самостоятельно написать и так и так, чтобы увидеть разницу.

А затем выберите верный ответ на вопрос.

- ☐ Особенной цели нет, такую сигнатуру сделали разработчики Uber.
- ☐ Чтобы избегать вычисления аргументов функции-литерала при её defer'e.
- ☐ Чтобы была возможность модифицировать возвращаемый функцией аргумент.

github.com/emperror/emperror



The Emperor takes care of all errors personally.

Библиотека с интересным названием (удачно расположившем её первой в списке awesome-go) и предисловием. А в остальном, исходя из нашего субъективного опыта, не очень подходит для того, чтобы взять и переехать на неё целиком в продакшене. Утащить к себе пару фичей – можно, но не более.

В своём описании библиотека рекламирует три вещи: логирование ошибок, обработку паник и фильтрацию ошибок.

Логирование ошибок

emperror интегрирован с двумя библиотеками для логирования logur/logur и sirupsen/logrus. Первая, кстати, является разработкой непосредственно автора **emperror**.

Так вот, если посмотреть на интеграцию с **sirupsen/logrus**, то увидим обёртку над логером, позволяющей в одну строчку логировать ошибку с деталями в виде полей. Детали вытаскивает уже третья библиотека – emperror/errors:

```
// https://goplay.tools/snippet/tWQivtzpfRz
package main

import (
    "errors"

    emperrors "emperror.dev/errors"
    logrushandler "emperror.dev/handler/logrus"
    "github.com/sirupsen/logrus"
)

func main() {
    logger := logrus.New()
    handler := logrushandler.New(logger)

    err := errors.New("an error")
    handler.Handle(err)

    err2 := emperrors.WithDetails(err, "userID", 3587, "requestID",
"4cfdc2e157eefe6facb983b1d557b3a1")
    handler.Handle(err2)
}
```

```
/*
level=error msg="an error"
level=error msg="an error" requestID=4cfdc2e157eefe6facb983b1d557b3a1
userID=3587
*/
```

Вау-эффекта не производит. Не выглядит какой-то удобной фичей, ради которой можно подсаживаться на **emperror/errors**.

Panics and recovers

Мы не будем подробно разбирать механизм работы этой фичи, потому что в данном курсе в целом избегали понятия **паники**. Ей посвящён целый отдельный курс – **"Продвинутая работа с паникой в Go"**.

Вкратце – с помощью `emperror.HandleRecover` вы можете зарегистрировать обработчик паник, созданных через `emperror.Panic`:

```
package main

import (
    "fmt"

    "emperror.dev/emperror"
    "emperror.dev/errors"
)

var panicHandler = emperror.ErrorHandlerFunc(func(err error) {
    if err == nil {
        return
    }
    fmt.Println("panic handler called:", err)
})

func main() {
    defer emperror.HandleRecover(panicHandler)
    emperror.Panic(doSomething())
}

func doSomething() error {
```

```
    return errors.New("error from doSomething")
}

// panic handler called: error from doSomething
```

За подробностями обращайтесь в исходники модуля, а также в наши примеры.

Фильтрация ошибок

Sometimes you might not want to handle certain errors that reach the error handler.

A common example is a catch-all error handler in a server. You want to return business errors to the client.

Идея выглядит прикольно. Прочитав описание, кажется, что эта фича позволит разделять поток ошибок сервиса на две части:

- В первом – ошибки бизнес-логики. Их можно и, наверное, нужно возвращать клиенту как есть, чтобы он понимал, в чем дело и, по возможности, как-то сам их обработал.

- Во втором – внутренние ошибки самого сервиса. Например, базуля прилегла или поход в нижестоящий сервис отвалился. Клиента они мало волнуют, и вряд ли он сможет с ними что-то сделать. Поэтому такие ошибки можно прятать за каким-нибудь "internal server error" сообщением. Более того безопасники потом скажут нам спасибо.

Но на самом деле всё обстоит не так :(

Фильтрация в **emperror** делит поток ошибок на две части, но ошибки, которые попадают в фильтр, игнорируются (исходник примера):

```
var (
    err1        = errors.New("error 1")
    err2        = errors.New("error 2")
    errsToSkip = []error{err1, err2}

    // ...
    errMatcher = emperror.ErrorMatcher(func(err error) bool {
        for i := range errsToSkip {
            if needDiscard := errors.Is(err, errsToSkip[i]);
needDiscard {
                return true
            }
        }
        return false
    })
)

func main() {
    handler := emperror.WithFilter(errHandler, errMatcher)

    handler.Handle(err1)                          // Обработчик не
сработает.
    handler.Handle(errors.New("unknown error")) // Обработчик
сработает.
}
```

Честно говоря, не очень понятная фича. Где её можно применить – загадка.

Пишите о своих вариантах в комментариях.

github.com/upspin/upspin/errors

Строго говоря, upspin.io – это вообще не библиотека для работы с ошибками. Это фреймворк, который скорее мёртв, чем жив, предназначенный для решения проблемы обмена данными между людьми. А в исходниках этого фреймворка есть пакет **errors**.

P.S. на досуге можно посмотреть, как Роб Пайк рассказывает про **upspin**, для чего он и зачем.

---

upspin.io/errors

Но вернёмся обратно к ошибкам. **upspin.io/errors** заслуживает внимания, потому что там реализовали то, о чем каждый из нас мечтал и боялся сделать. То, чем можно вдохновиться и перелопатить половину своего проекта 🤦.

В **upspin.io/errors** завели универсальный тип errors.Error, содержащий ряд полей, совокупность которых полностью описывает ошибку с точки зрения домена фреймворка. Профит – используем везде один тип:

```
package errors

type Error struct {
    // Path is the Upspin path name of the item being accessed.
    Path upspin.PathName
    // User is the Upspin name of the user attempting the operation.
    User upspin.UserName
    // Op is the operation being performed, usually the name of the
method
```

```
    // being invoked (Get, Put, etc.). It should not contain an at
sign @.
    Op Op
    // Kind is the class of error, such as permission failure,
    // or "Other" if its class is unknown or irrelevant.
    Kind Kind
    // The underlying error that triggered this one, if any.
    Err error

    // Stack information; used only when the 'debug' build tag is
set.
    stack

}
```

Поля `Path`, `User`, `Op` – информационные, используются по большей части в методе `(*Error).Error` (нейминг на уровне) и сопутствующих `Marshal`/`Unmarshal`.

А вот поле `Kind` сужает класс возникающих ошибок до конечного множества. С помощью него реализована функция `errors.Is`, способная заменить собой всякие **sentinel**, **type assertion** и **opaque** проверки:

```
package errors

// Kind defines the kind of error this is, mostly for use by systems
// such as FUSE that must act differently depending on the error.
type Kind uint8

// Kinds of errors.
//
// The values of the error kinds are common between both
// clients and servers. Do not reorder this list or remove
// any items since that will change their values.
// New items must be added only to the end.
const (
    Other          Kind = iota // Unclassified error. This value is
not printed in the error message.
    Invalid                    // Invalid operation for this type of
item.
    Permission                 // Permission denied.
    IO                         // External I/O error such as network
failure.
    Exist                      // Item already exists.
```

```
    NotExist                    // Item does not exist.
    IsDir                       // Item is a directory.
    NotDir                      // Item is not a directory.
    NotEmpty                    // Directory not empty.
    Private                     // Information withheld.
    Internal                    // Internal error or inconsistency.
    CannotDecrypt               // No wrapped key for user with read
access.
    Transient                   // A transient error.
    BrokenLink                  // Link target does not exist.
)

// Is reports whether err is an *Error of the given Kind.
// If err is nil then Is returns false.
func Is(kind Kind, err error) bool {
    e, ok := err.(*Error)
    if !ok {
        return false
    }
    if e.Kind != Other {
        return e.Kind == kind
    }
    if e.Err != nil {
        return Is(kind, e.Err)
    }
    return false

}
```

Например, ошибки определенного типа игнорируются в некоторых случаях. Или, наоборот, при возникновении ошибки типа `errors.NotExist` делается попытка graceful-обработки.

Резюме

Заведение одного универсального типа ошибки на весь проект, конечно, может решить проблему хаоса и разрозненности по работе с ошибками, но тут надо быть аккуратными и делать это осознанно. Есть мнение, что данный подход подходит далеко не всем, потому что введением единого типа мы загоняем себя в определённые рамки, внутри которых придётся работать.

Подведём итоги

Новых идей по обработке ошибок много, а библиотек ещё больше. Если на ваш взгляд появилось что-то заслуживающее внимания или мы просто прошли мимо, обязательно пишите в комментариях – разберём, обсудим, сделаем урок.

На этой ноте мы заканчиваем рассмотрение нестандартных модулей по работе с ошибками в Go. Надеемся, что у вас сложилось понимание, достаточно ли для жизни стандартной библиотеки или всё-таки придётся выходить за её пределы. Второе в любом случае полезно – всегда приятно почерпнуть интересные идеи и практики извне. Для этого рекомендуем периодически заходить в секцию **Error Handling** списка awesome-go.

Более того сообщество Go может влиять на эволюцию языка, поэтому не ровен час, как в стандартный пакет **errors** ~~занесут стектрейсы~~ добавят что-нибудь нового под давлением общественности (хотя бытует мнение, что core-команда Go гнёт свою планку до последнего, не взирая на мнение со стороны).

В следующем модуле мы обсудим **лучшие практики по работе с ошибками в Go**.