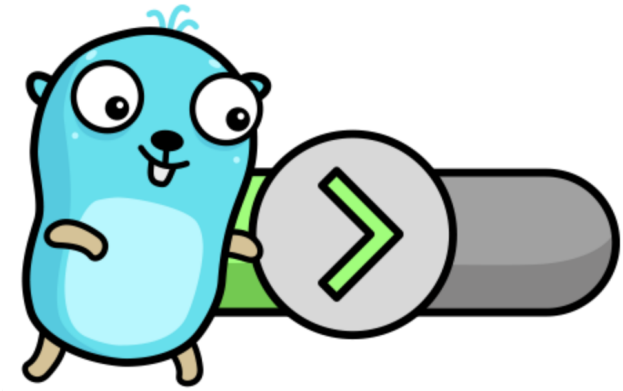
defer: игнорирование ошибок

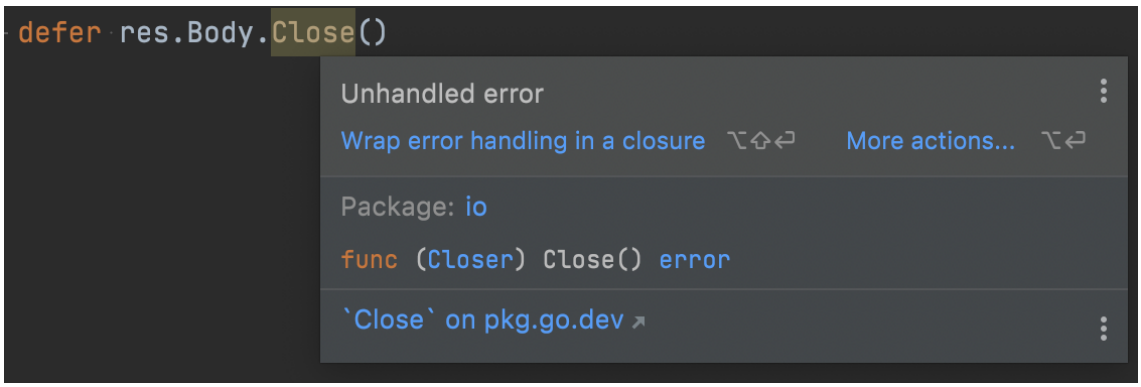
В этом уроке мы коснёмся вопроса игнорирования ошибок от отложенных функций и методов.

Эта тема более подробно раскрыта в курсе **"Продвинутая работа с ошибками в Go"**, здесь же мы просто дадим некоторые комментарии и посмотрим на приёмы и инструменты, позволяющие работать с подобными ошибками.



Постановка проблемы

В жизни каждого разработчика Go наступает момент, когда IDE начинает ругаться на необработанную в `defer` ошибку:



И возникают вопросы: что от меня хотят, как её обработать и надо ли вообще это делать?

Предполагая недоумение, IDE услужливо предлагает создать для вас замыкание с обработкой или явным игнорированием ошибки:

```
defer func(body io.ReadCloser) {
    if err := body.Close(); err != nil {

    }
}(res.Body)
```

```
defer func(body io.ReadCloser) {
    _ = body.Close()
}(res.Body)
```

Так как же быть?

Пути решения

В общем случае видятся следующие варианты:

1. Проигнорировать ошибку.

2. Залогировать ошибку.
3. Если верхнеуровневой ошибки нет, но есть ошибка в `defer`, то вернуть её.
4. Переписать ошибкой из `defer` (если она есть) верхнеуровневую ошибку.
5. Скомбинировать обе ошибки (своими или чужими инструментами).

Посмотрим, как это выглядит в коде на примере простой функции `GetPage` (закроем глаза на отсутствие DI и использование дефолтного клиента):

```
func GetPage(ctx context.Context, url string) ([]byte, error) {
    req, err := http.NewRequestWithContext(ctx, http.MethodGet, url,
nil)
    if err != nil {
        return nil, fmt.Errorf("build GET request: %v", err)
    }

    res, err := http.DefaultClient.Do(req)
    if err != nil {
        return nil, fmt.Errorf("do request: %w", err)
    }
    defer res.Body.Close() // Выполнится при одном из двух return
ниже.

    body, err := io.ReadAll(res.Body)
    if err != nil {
        return nil, fmt.Errorf("read body: %w", err)
    }

    return body, nil

}
```

Логируем ошибку:

```
defer func() {
    if err := res.Body.Close(); err != nil {
        log.Println("response body close error: " + err.Error())
    }
```

```
}()
```

Считаем ошибку в отложенной операции более приоритетной:

```
func GetPage(ctx context.Context, url string) (body []byte, err
error) {
    // ...

    defer func() {
        if err2 := res.Body.Close(); err2 != nil {
            err = fmt.Errorf("response body close: %w", err2)
        }
    }()

    body, err = io.ReadAll(res.Body)
    // ...

}
```

Цепляем к возвращаемой ошибке текст ошибки от отложенной операции:

```
func GetPage(ctx context.Context, url string) (body []byte, err
error) {
    // ...

    defer func() {
        if err2 := res.Body.Close(); err2 != nil {
            if err != nil {
                err = fmt.Errorf("%w: before resp body close err:
%v", err, err2)
            }
        }
    }()

    body, err = io.ReadAll(res.Body)
    // ...

}
```

Комбинируем варианты выше:

```
func GetPage(ctx context.Context, url string) (body []byte, err
error) {
    // ...
```

```
    defer func() {
        if err2 := res.Body.Close(); err2 != nil {
            if err != nil {
                err = fmt.Errorf("%w: before resp body close err:
%v", err, err2)
            } else {
                err = fmt.Errorf("response body close: %w", err2)
            }
        }
    }()

    body, err = io.ReadAll(res.Body)
    // ...

}
```

Мы видим, что возникает достаточно много телодвижений и элегантность кода теряется.

Для понимания, нужно ли уходить в подобные дебри, и для выбора одного из вариантов выше следует ответить на ~~очевидные~~ вопросы:

- Важно ли нам знать об ошибке отложенной операции?
- Наши действия, если ошибка возникнет и мы узнаем о ней?
- Какие побочные эффекты ошибка несёт за собой?
- Что будет, если мы проигнорируем её?
- Какая из ошибок более приоритетная: изначальная или от отложенной операции?

Чаще всего оказывается, что ошибку можно игнорировать и ничего страшного не произойдёт, но если вы пишите **robust reliable другие-модные-слова-из-вакансий application** или просто сомневаетесь, то хотя бы залогируйте её.

В следующих шагах мы дадим комментарии к наиболее популярным откладываемым операциям, а затем посмотрим на нестандартные модули для работы с ошибками в `defer`.

(*http.Response).Body.Close

Хоть мы и использовали данный метод в качестве примера в предыдущем шаге, в общем случае ошибку от него **можно игнорировать**:

- Большинство реализаций тела ответа всегда в `Close` возвращают `nil`.
- Не очень понятно, что с ошибкой делать, особенно если тело ответа было успешно вычитано и с ним уже можно работать.

Даже разработчики Go отмечают:

I did not encounter any concrete situation where reporting the error is meaningful.

Most users ignore error from resp.Body.Close().

Что подтверждается примером из официальной документации:

```
func main() {
    res, err := http.Get("http://www.google.com/robots.txt")
    if err != nil {
        log.Fatal(err)
    }

    body, err := io.ReadAll(res.Body)
    res.Body.Close()                    // Ошибка игнорируется.
    if res.StatusCode > 299 {
        log.Fatalf("Response failed with status code: %d and\nbody:
%s\n", res.StatusCode, body)
    }
    if err != nil {
        log.Fatal(err)
    }
```

```
    fmt.Printf("%s", body)
}

// Обратим внимание, что после успешного выполнения запроса тело ответа
// закрывают в любом случае, независимо от дальнейшей работы с ответом.
//
// Вопрос к опытным слушателям курса - а почему в примере тело ответа закрыли не через defer? :)

//
```

(*os.File).Close

Тут правила простые:

- Если мы читаем файл, то достаточно `defer f.Close()`.

- Если мы пишем в файл, то `Close` следует вызвать явно, опционально оставив `defer` (данный подход мы обсуждали в начале урока): `defer f.Close() + return f.Close()`, также можно завершать запись через синхронизацию: `defer f.Close() + return f.Sync()`.

В обоих вариантах ошибку от `defer f.Close()` **можно не проверять** – файловый дескриптор скорее всего в любом случае будет закрыт:

This can occur because the Linux kernel always releases the file descriptor early in the close operation, freeing it for reuse.

Many other implementations similarly always close the file descriptor even if they subsequently report an error on return from close().

POSIX.1 is currently silent on this point, but there are plans to mandate this behavior in the next major release of the standard.

Почему при записи в файл в целом стоит дополнять `defer` явным вызовом `Close`? Потому что здесь всё-таки важно проверить ошибку от операции закрытия файла, которая может указать на проблемы с последней операцией записи в файл:

*A careful programmer will check the return value of **close**(), since it is quite possible that errors on a previous **write(2)** operation are reported only on the final **close**() that releases the open file description. Failing to check the return value when closing a file may lead to silent loss of data. This can especially be observed with NFS and with disk quota.*

При этом `Close` не гарантирует запись данных на диск (которая произойдёт, но позже), поэтому в зависимости от контекста его лучше заменить на (*os.File).Sync:

*A successful close does not guarantee that the data has been successfully saved to disk, as the kernel uses the buffer cache to defer writes. Typically, filesystems do not flush buffers when a file is closed. If you need to be sure that the data is physically stored on the underlying disk, use **fsync(2)**. (It will depend on the disk hardware at this point.)*

*A careful programmer who wants to know about I/O errors may precede **close**() with a call to **fsync(2)**.*

Тезисы выше выглядят в коде следующим образом (исходник примера):

```
func ReadNFromFile(fname string, nn int, to io.Writer) error {
```

```
    f, err := os.Open(fname)
    if err != nil {
        return fmt.Errorf("open file: %w", err)
    }
    defer f.Close()

    data := make([]byte, nn)
    n, err := f.Read(data)
    if err != nil {
        return fmt.Errorf("read from file: %w", err)
    }

    _, err = to.Write(data[:n])
    return err

}

func WriteToFile(fname, text string) error {
    f, err := os.OpenFile(fname, os.O_WRONLY|os.O_APPEND, 0)
    if err != nil {
        return fmt.Errorf("open file: %w", err)
    }
    defer f.Close()

    if _, err := f.WriteString(text); err != nil {
        return fmt.Errorf("write to file: %w", err)
    }

    return f.Sync()

}
```

Задача "Периодическая синхронизация"

Ссылка на заготовку.

Вам необходимо реализовать функцию, которая периодически синхронизирует нечто синхронизируемое:

```
type Syncer interface {
    Sync() error
}

// Sync синхронизирует входной s через интервалы времени, равные period.
// Является блокирующей функцией.

func Sync(ctx context.Context, s Syncer, period time.Duration) error
```

Например, её можно использовать для того, чтобы в фоне сбрасывать на диск файл с логами.

- Если входящий контекст отменяется, то функция должна прекратить свою работу и вернуть `nil`.
- Если очередной вызов `s.Sync()` завершается ошибкой, то функция должна прекратить свою работу и вернуть эту ошибку.

Больше подробностей см. в заготовке задачи и тестах.

(*sql.Tx).Rollback

В общем случае паттерн работы с SQL-транзакциями в Go выглядит следующим образом:

```
func (s *Storage) CreateOrder(ctx context.Context, quantity,
customerID int) (int64, error) {
    tx, err := s.db.BeginTx(ctx, nil)
    if err != nil {
        return 0, fmt.Errorf("begin tx: %w", err)
    }
    defer tx.Rollback() // The rollback will be ignored if the tx has
been committed later in the function.

    // N calls:
    // tx.QueryRowContext
    // tx.ExecContext
    // etc.

    if err = tx.Commit(); err != nil {
        return 0, fmt.Errorf("commit tx: %w", err)
    }

    return orderID, nil

}
```

Ошибку от `tx.Rollback()` редко проверяют по следующим соображениям:

- Ошибка скорее всего будет связана с проблемным соединением или специфичной реализацией драйвера к базе. Но несмотря на результат работы функции транзакция больше не будет валидной и не будет зафиксирована в БД.

- Если вызовы `Rollback` и `Commit` не разделены так, что вызывается **только** один из двух методов, то после коммита (неважно, успешного или нет) при вызове отложенного ролбека мы получим ошибку sql.ErrTxDone, которую нет смысла обрабатывать. Этот факт позволяет нам писать `defer tx.Rollback()`, не задумываясь, не помешает ли это коммиту:

```go
// database/sql/sql.go
package sql

// Commit commits the transaction.
func (tx *Tx) Commit() error {
    // ...
    if !atomic.CompareAndSwapInt32(&tx.done, 0, 1) {
        return ErrTxDone
    }
    // ...
}

// Rollback aborts the transaction.
func (tx *Tx) Rollback() error {
    return tx.rollback(false)
}

func (tx *Tx) rollback(discardConn bool) error {
    if !atomic.CompareAndSwapInt32(&tx.done, 0, 1) {
        return ErrTxDone
    }
    // ...

}
```

В серьёзных приложениях работа с транзакциями обычно обёрнута в дополнительные функции / хелперы. Тогда нам ничто не мешает перестраховаться и прицепить ошибку отката к оригинальной ошибке или хотя бы залогировать её:

```go
import "database/sql"

func RunInTransaction(tx *sql.Tx, fn func(*sql.Tx) error) error {
    if err := fn(tx); err != nil {
        if rErr := tx.Rollback(); rErr != nil {
            log.Println("tx.Rollback failed:", rErr)
            // err = fmt.Errorf("%w and after rollback error: %v",
err, rErr)
        }
        return err
    }
    return tx.Commit()
}
```

Как видно и `defer` не понадобился, но с ним можно сделать что-нибудь вроде:

```
defer func() {
    if rErr := tx.Rollback(); !errors.Is(rErr, sql.ErrTxDone) {
        // Работаем с rErr.
        // ...
    }

}()
```

(*sql.Rows).Close

Согласно документации и мнению разработчиков Go общий паттерн работы с `sql.Rows` выглядит следующим образом:

```
func (r *Repo) PrintUsersByAge(ctx context.Context, age int, out
io.Writer) error {
    rows, err := r.db.QueryContext(ctx, "SELECT name FROM users WHERE
age=?", age)
    if err != nil {
        return fmt.Errorf("query: %w", err)
    }
    defer rows.Close() // <- !!!

    for rows.Next() {
        var name string
        if err := rows.Scan(&name); err != nil {
            return fmt.Errorf("scan: %w", err)
        }

        if _, err := out.Write([]byte(name)); err != nil {
            return fmt.Errorf("write: %w", err)
        }
    }
    return rows.Err() // <- !!!
}
```

И снова мы видим **игнорирование ошибки**, теперь от `rows.Close`.

Данный факт коррелирует с официальным [туториалом](#) по **database/sql**:

The error returned by `rows.Close()` is the only exception to the general rule that it's best to capture and check for errors in all database operations. If `rows.Close()` returns an error, it's unclear what you should do. Logging the error message or panicing might be the only sensible thing, and if that's not sensible, then perhaps you should just ignore the error.

Задача "AttachTo"

[Ссылка на заготовку](#).

Разработчик устал думать о том, какие ошибки можно игнорировать в `defer`, а какие нет и решил, что все ~~профессии~~ ошибки нужны, все ошибки важны!

Вам необходимо помочь ему реализовать функцию `AttachTo`:

```
// AttachTo выполняет функцию f и комбинирует ошибку от неё с target.

func AttachTo(target *error, f func() error)
```

Пример использования:

```
func GetPage(ctx context.Context, url string) (body []byte, err
error) {
    // ...

    defer errors.AttachTo(&err, res.Body.Close)

    // ...

}
```

Требования к работе функции:

- Если обе ошибки (и исходная и от отложенной операции) нулевые, то ничего не происходит.

- Если обе ошибки ненулевые, то в возвращаемую ошибку записывается их комбинация – такая ошибка, от которой `errors.Is` по каждой из составляющих её ошибок будет возвращать `true`. Текст такой ошибки выглядит следующим образом:
- `"<текст исходной ошибки> (and after <текст ошибки от отложенной операции>)"`
- В обратном случае в возвращаемую ошибку записывается та ошибка, которая не является `nil`.

- Если `AttachTo` вызывают от нулевого указателя или функции, то `AttachTo` паникует строкой `"invalid usage of AttachTo"` (да, формально мы ещё не знакомы с паникой, но ничто не мешает сделать первые шаги в её сторону).

Больше подробностей в заготовке и тестах.

go.uber.org/multierr: AppendInvoke

Было бы глупо думать, что в Uber ещё не кумекали, как решить проблему игнорирования ошибок от отложенных операций 😄.

go.uber.org/multierr – это интересная и лаконичная библиотека, основной задачей которой служит предоставление инструмента по комбинированию N ошибок в 1. Но, как следствие, мы имеем и полезные для нашей темы функции и типы:

- multierr.AppendInvoke
- multierr.Invoke
- multierr.Close

Примеры использования из официальной документации:

```
func sendRequest(req Request) (err error) {
    conn, err := openConnection()
    if err != nil {
        return err
    }

    defer multierr.AppendInvoke(&err, multierr.Close(conn)) // <- !!!
    // ...

}

func processFile(path string) (err error) {
    f, err := os.Open(path)
    if err != nil {
        return err
    }

    defer multierr.AppendInvoke(&err, multierr.Close(f))  // <- !!!
    return processReader(f)
}

func processReader(r io.Reader) (err error) {
    scanner := bufio.NewScanner(r)
```

```
    defer multierr.AppendInvoke(&err, multierr.Invoke(scanner.Err))
// <- !!!

    for scanner.Scan() {
        // ...
    }
    // ...

}
```

(исходник для поиграться).

Обращаем внимание, что `multierr.AppendInvoke` принимает указатель на именованный возвращаемый аргумент!

cockroachdb/errors: CombineErrors

Вряд ли вы захотите себе тащить этот (не побоимся этого слова) фреймворк по работе с ошибками ради пары функций, но посмотрим, что предлагают ребята из Cockroach Labs.

А предлагают они следующее: "прицепить" второстепенную ошибку к первой так, чтобы она отображалась при её логировании, но не влияла на анализ возникновения причины ошибок (грубо говоря, не реагировала на `errors.Is` и подобные методы).

Эту идею реализуют следующие функции:

- errors.WithSecondaryError (используется в `errors.CombineErrors`)
- errors.CombineErrors

Посмотрим на пример (исходник):

```
package main

import (
    "context"
    "fmt"
    "io"

    "github.com/cockroachdb/errors"
)

func main() {
    err := io.EOF
    err = errors.WithSecondaryError(err, context.Canceled)

    fmt.Println(err)
    /*
        EOF
    */

    fmt.Printf("%+v", err)
    /*
        EOF
        (1) secondary error attachment
          | context canceled
          | (1) context canceled
          | Error types: (1) *errors.errorString
        Wraps: (2) EOF
        Error types: (1) *secondary.withSecondaryError (2)
*errors.errorString
    */

    fmt.Println()
    fmt.Println(errors.Is(err, io.EOF))            // true
    fmt.Println(errors.Is(err, context.Canceled)) // false

}
```

`errors.WithSecondaryError` позволяет нам комбинировать ошибки так, что текст от обеих присутствует в выводе, но "настоящей"/"главной" ошибкой считается только первая.

Это чем-то похоже на приём, который мы рассматривали в начале урока:

```
defer func() {
    if err2 := res.Body.Close(); err2 != nil {
        if err != nil {
            err = fmt.Errorf("%w: before resp body close err: %v",
err, err2)
        } else {
            err = fmt.Errorf("response body close: %w", err2)
        }
    }

}()
```

Теперь он будет выглядеть следующим образом:

```
defer func() {
    // Дополняем основную ошибку второстепенной, если она возникнет.
    // При этом, если есть только второстепенная ошибка, то она и
будет возвращена.
    //
    // ВАЖНО: использование именованного возвращаемого значения.
    err = errors.CombineErrors(err, response.Body.Close())

}()
```

Важное отличие от варианта с `fmt.Errorf`: при использовании `errors.CombineErrors` второстепенная ошибка попадёт на экран только при подробной печати (через спецификатор `%+v`).

Важное отличие от **go.uber.org/multierr**: там все ошибки считаются равноправными – они все фигурируют при обычной печати на экран и влияют на результат `errors.Is`, `errors.As` и т.д.

Промежуточные выводы

Мы узнали, что не всегда стоит бросаться править инспекции от IDE (или от вашего коллеги-злостного-ревьювера) и для начала хорошо бы разобраться – может окажется, что в данном конкретном случае игнорировать ошибку в `defer` допустимо.

Если всё-таки нет, то мы можем:

- Залогировать ошибку в `defer`.
- Скомбинировать ошибку в `defer` с исходной ошибкой своими или сторонними инструментами.
- Отказаться от `defer` в пользу явного вызова или оставить и `defer` и явный вызов (тут будьте бдительны).