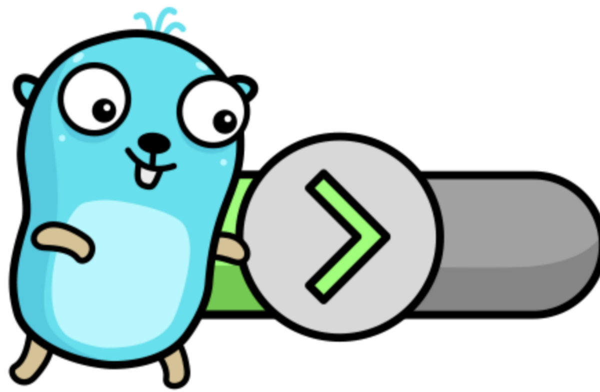
defer: доступ к внешней области видимости

В этом уроке мы узнаем, как с помощью `defer` менять возвращаемые окружающей функцией результаты, как это работает под капотом и какие есть gotcha.



Начнём с небольшого примера

Отложенная анонимная функция имеет доступ к переменным из внешней области видимости. Так как отложенная функция вызывается при выходе из обрамляющей её функции, то она будет работать с последними значениями, выставленными переменным во время работы обрамляющей функции:

```
func main() {
    i := 3                          // 1)

    defer func() {
        fmt.Println("i =", i)       // 3)
    }()

    i = 100                         // 2)
}

// Напечатает:
```

```
// i = 100
```

([пример](), чтобы поиграться).

Тест "Изменение возвращаемого значения через defer"

Мы слегка изменили пример из предыдущего шага, добавив в `defer` изменение возвращаемого значения:

```
func main() {
    fmt.Println(calculate())
}

func calculate() int {
    i := 3

    defer func() {
        i = 42                  // <- !!!
        fmt.Println("i =", i)
    }()

    i = 100
    return i
}
```

Что в итоге выведет на экран программа?

Выберите один вариант из списка

- ☐ i = 42

100
- [ ] i = 42
  42
- [ ] i = 3
  42
- [ ] i = 3
  3
- [ ] i = 3
  100
- [ ] i = 100
  100

RTFM

Почему так? Согласно [спецификации](#) имеем:

That is, if the surrounding function returns through an explicit return statement, ***deferred functions are executed after any result parameters are set by that return statement but before the function returns to its caller****.*

Т.е. сначала выставляются все результаты работы функции, затем вызываются отложенные функции и затем управление возвращается вызывающей функции дальше по стеку.

Таким образом, в предыдущем примере отложенная функция работает с **копией** возвращаемых результатов и, очевидно, не влияет на них.

Подтверждение этому можно найти, задизасемблировав пример из предыдущего шага (`i = 100` заменено на `i = get100()`, чтобы избежать некоторых оптимизаций компилятора):

```
func main() {
    calculate()
}

func calculate() int {
```

```
    i := 3

    defer func() {
        i = 42
    }()

    i = get100()
    return i

}

# https://godbolt.org/z/afzafE3nn

# func main()
        TEXT    "".main(SB), ABIInternal, $16-0

        SUBQ    $16, SP
        MOVQ    BP, 8(SP)
        LEAQ    8(SP), BP
        CALL    "".calculate(SB)
        MOVQ    8(SP), BP
        ADDQ    $16, SP
        RET

# func calculate() int
        TEXT    "".calculate(SB), ABIInternal, $48-8

        SUBQ    $48, SP
        MOVQ    BP, 40(SP)
        LEAQ    40(SP), BP
        XORPS   X0, X0
        MOVUPS  X0, 24(SP)

        # return value init
        MOVQ    $0, "".~r0+56(SP)

        # i := 3
        MOVQ    $3, "".i+16(SP)

        # defer func()
        LEAQ    "".calculate.func1·f(SB), AX
        MOVQ    AX, ""..autotmp_4+32(SP)
        LEAQ    "".i+16(SP), AX         # Запоминаем адрес переменной
i для дальнейшего использования в defer.
        MOVQ    AX, ""..autotmp_5+24(SP)
```

```
        # i = 100
        CALL    "".get100(SB)
        MOVQ    (SP), AX
        MOVQ    AX, "".i+16(SP)

        # set return value
        MOVQ    AX, "".~r0+56(SP)        # Копируем i в стек для
возврата вызывающей стороне.

        # call deferred func
        MOVQ    "".autotmp_5+24(SP), AX # Достаём адрес i для
использования в defer.
        MOVQ    AX, (SP)
        CALL    "".calculate.func1(SB)   # Уже после установки
результата вызываем defer.

        # return
        MOVQ    40(SP), BP
        ADDQ    $48, SP
        RET

# deferred func
        TEXT    "".calculate.func1(SB), NOSPLIT|ABIInternal, $0-8
        MOVQ    "".&i+8(SP), AX          # Хоть и перед нами
замыкание (работа со ссылкой на
                                         # переменную из внешней
области видимости), но результат
                                         # работы внешней функции уже
был выставлен ранее :(
        # i = 42
        MOVQ    $42, (AX)

        RET
```

Named results спешат на помощь

Так как же всё-таки менять возвращаемые значения с помощью `defer`?

Ответ на этот вопрос, как обычно, подсказывает спецификация:

For instance, if the deferred function is a function literal and the surrounding function has ***named result parameters*** *that are in scope within the literal, the deferred function may access and modify the result parameters before they are returned.*

Лёгким движением руки именуем возвращаемое значение и его изменение в `defer` не проходит бесследно:

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

import "fmt"

func main() {
    fmt.Println(calculate()) // 42
}

func calculate() (i int) {
    i = 3

    defer func() {
        i = 42                      // <- !!!
        fmt.Println("i =", i)
    }()

    i = 100
    return i
}
```

Ради интереса посмотрим, что изменилось в ассемблере:

```
# https://godbolt.org/z/f78sja6Ms

# func main()
        TEXT    "".main(SB), ABIInternal, $16-0

        SUBQ    $16, SP
        MOVQ    BP, 8(SP)
        LEAQ    8(SP), BP
        CALL    "".calculate(SB)
        MOVQ    8(SP), BP
        ADDQ    $16, SP
```

```
        RET

# func calculate() (i int)
        TEXT    "".calculate(SB), ABIInternal, $40-8

        SUBQ    $40, SP
        MOVQ    BP, 32(SP)
        LEAQ    32(SP), BP
        XORPS   X0, X0
        MOVUPS  X0, 16(SP)

        # return value init
        MOVQ    $0, "".i+48(SP)

        # i = 3
        MOVQ    $3, "".i+48(SP)

        # defer func()
        LEAQ    "".calculate.func1·f(SB), AX
        MOVQ    AX, ""..autotmp_3+24(SP)
        LEAQ    "".i+48(SP), AX # Запоминаем адрес переменной i для
дальнейшего использования в defer.
                                # Очевидно, что в этом примере - это
адрес возвращаемого значения.
        MOVQ    AX, ""..autotmp_4+16(SP)

        # i = 100
        CALL    "".get100(SB)
        MOVQ    (SP), AX
        MOVQ    AX, "".i+48(SP)

        # Нет никаких копирований переменных в результат.
        # Мы всегда работаем сразу с возвращаемым значением
(благодаря наличию нейминга у него).

        # call deferred func
        MOVQ    ""..autotmp_4+16(SP), AX # Достаём адрес i для
использования в defer.
        MOVQ    AX, (SP)
        CALL    "".calculate.func1(SB)

        # return
        MOVQ    32(SP), BP
        ADDQ    $40, SP
        RET
```

```
# deferred func
        TEXT    "".calculate.func1(SB), NOSPLIT|ABIInternal, $0-8
        MOVQ    "".&i+8(SP), AX
        MOVQ    $42, (AX)

        RET
```

Какая мораль? Если вы видите, что автор функции меняет в `defer` значение возвращаемой переменной и она не является именованным параметром результата, то скорее всего перед вами баг.

Задача "Transaction Rollback"

Ссылка на заготовку.

Перед вами функция `accounts.Create`, которая в одной транзакции создаёт и заполняет табличку с банковскими счетами:

```go
package accounts

import "fmt"

// ...

func Create(db DB) error {
    var err error

    tx, err := db.Begin()
    if err != nil {
        return fmt.Errorf("begin: %w", err)
    }

    defer func() {
        if err != nil {
            if errRollback := tx.Rollback(); errRollback != nil {
                err = fmt.Errorf("rollback: %w", errRollback)
            }
        } else {
            err = tx.Commit()
        }
    }()
```

```
    // A lot of tx.Exec() calls.
    // ...

    return err
}
```

Если во время работы функции не произошло никаких ошибок, то транзакцию необходимо **закоммитить**, а в обратном случае – **откатить**. Если во время отката произошла ошибка, то она считается более приоритетной, чем первоначальная и в таком случае именно ошибка отката (`errRollback`) ожидается на выходе из функции.

В следующем уроке мы узнаем, какие есть приёмы, чтобы совмещать обе ошибки (и первоначальную и ошибки от отложенных операций), а сейчас **вам необходимо исправить код так, чтобы он работал согласно условиям выш**е.

Больше подробностей см. в заготовке задачи и тестах.

Задача "ParseToken для безопасников"

*Данная задача позаимствована из курса **"Продвинутая работа с ошибками в Go"**.*

Ссылка на заготовку.

Разработчику поступила задача написать функцию разбора и валидации JSON WEB токена (JWT). При этом сервис поддерживает единственный алгоритм подписи токена – `HS256`.

Примером JWT является

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCIs
ImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJlZ
F9hdCI6MTYxMTY0MTkyNn0.IY-RsEAxCiDUch8_Rk77HvEDE8qhf1Zd3IUtxGlwd9w

Если раскрыть его, то получим

```
{
  "alg": "HS256",
  "typ": "JWT"
}
.
{
  "subject": "1234567890",
  "email": "john@gmail.com",
  "scopes": ["admin"],
  "expired_at": 1611641926
}
.
```

`signature`

Недолго думая, наш коллега написал функцию `ParseToken` (см. исходник в ссылке на заготовку):

```
package jwt

// ParseToken парсит и валидирует токен jwt, проверяя, что он
подписан
// алгоритмом HMAC SHA256 с использованием ключа secret.

func ParseToken(jwt, secret []byte) (Token, error)
```

Опустив детали, можно выделить следующий алгоритм работы функции:

```
ParseToken:
    // Проверка входных аргументов.
    // Проверка формата токена.
    // Парсинг (parseHeader) и валидация заголовка токена.
    // Валидация подписи токена (verifySignature).
    // Парсинг (parsePayload) и валидация пайлоада.
```

```
    // Возврат данных из пайлоада (нижележащего токена) или ошибки.
```

Функция была протестирована и вмёржена и, казалось бы, что работа над `ParseToken` закончена, но тут пришли ребята из ИБ и попросили собирать статистику, какие пользователи пытаются подсунуть нам невалидные токены (или кто-то пытается сделать это за них).

Вам требуется переписать `ParseToken` таким образом, чтобы, когда это возможно, она возвращала ошибку, из которой можно получить **email** пользователя, связанного с ней:

```
token, err := jwt.ParseToken(data, secret)

var emailer interface {
    Email() string
}
if errors.As(err, &emailer) {
    // Use emailer.Email()
    // ...
}
if err != nil {
    return fmt.Errorf("parse token: %w", err)
}
```

- Принципиально сигнатуру функции (количество и типы принимаемых и возвращаемых аргументов) менять нельзя:

```
func ParseToken(jwt, secret []byte) (Token, error)
```

- Все идентификаторы из известного `ParseToken` вам доступны (`byteDot`, `parsePayload`, `parseHeader`, `verifySignature`, ошибки и пр.) – определять их не нужно.
- Доступные импорты:

```
import (
    "bytes"
    "crypto/hmac"
    "crypto/sha256"
    "encoding/base64"
    "encoding/json"
    "errors"
    "fmt"
    "time"

)
```

- В тестах используются разные **JWT** с разными **email**.
- Можно и нужно определять новые типы.

Постарайтесь реализовать эту задачу с помощью `defer`, а именно:

- Реализовать тип-обёртку над ошибкой, предоставляющий метод получения email.
- С помощью `defer` с определённого момента, когда email известен, оборачивать все возвращаемые ошибки в объект-обёртку, конструируя его на основе ошибки и адреса эл. почты.

Тест "Указатели и defer – 1"

Постарайтесь, не запуская кода, написать, что выведет программа:

```
func main() {
    fmt.Println(*calculate())
}

func calculate() *int {
    i := 3

    defer func() {
        i = 42
    }()

    i = 100
    return &i
}
```

Выберите один вариант из списка

- ☐ 0
- ☐ 3
- ☐ 42
- ☐ 100

Тест "Указатели и defer – 2"

Постарайтесь, не запуская кода, написать, что выведет программа:

```
func main() {
    fmt.Println(*calculate())
}

func calculate() *int {
    i := new(int)
    *i = 3

    defer func() {
        i = intPtr(42)
        *i = 50
    }()
```

```
    *i = 100
    return i
}

func intPtr(i int) *int {
    return &i

}
```

Выберите один вариант из списка

- ☐ 0
- ☐ 3
- ☐ 42
- ☐ 50
- ☐ 100

Тест "Указатели и defer – 3"

Постарайтесь, не запуская кода, написать, что выведет программа:

```
func main() {
    fmt.Println(calculate())
}

func calculate() int {
    i := 3

    defer changeResult(&i)

    i = 100
    return i
}

func changeResult(i *int) {
    *i = 42

}
```

Выберите один вариант из списка

- [ ] 100
- [ ] 3
- [ ] 42
- [ ] 0

Полезное следствие из свойства defer

Полезным следствием из свойства `defer` отрабатывать после выставления результатов функции является то, что мы можем снимать блокировку после копирования ресурса из критической секции в результат и избегать гонок доступа к данным.

Разберём на примере.

Нам нужно реализовать **concurrency safe** счётчик, имеющий следующий простой интерфейс:

```
type ICounter interface {
    // Inc увеличивает значение счётчика на 1.
    Inc()
    // Value возвращает текущее значение счётчика.
    Value() int

}
```

Создадим простой тип, содержащий непосредственно счётчик и мьютекс для него:

```
import "sync"

type Counter struct {
    mu sync.RWMutex
    i  int

}
```

С инкрементом всё очевидно:

```
func (s *Counter) Inc() {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.i++
}
```

```
// Или.

func (s *Counter) Inc() {
    s.mu.Lock()
    s.i++
    s.mu.Unlock()

}
```

А как быть с получением значения?

```
func (s *Counter) Value() int { // Гонка!
    return s.i

}

func (s *Counter) Value() int { // Уже лучше, но можно ли изящнее?
    s.mu.RLock()
    i := s.i
    s.mu.RUnlock()
    return i

}

func (s *Counter) Value() int { // Супер!
    s.mu.RLock()                // 1) Выставляем блокировку.
    defer s.mu.RUnlock()        // 3) Снимаем блокировку.
    return s.i                  // 2) Под блокировкой копируем
результат "наружу".
}
```

Собираем всё вместе:

```
import "sync"

type Counter struct {
    mu sync.RWMutex
    i  int
}

func (s *Counter) Inc() {
```

```
    s.mu.Lock()
    defer s.mu.Unlock() // Несмотря на кажущуюся сложность - это типичный способ организации подобного кода.
                        // В последующих уроках мы узнаем, что оверхеда на производительность
                        // данный вызов не даёт, но он защищает нас от будущих изменений метода
                        // и возможной баги, связанной с тем, что на выходе забудут освободить
                        // блокировку. Данный пример, конечно, утрирован, но в реальной жизни
                        // старайтесь всегда cleanup-операции выполнять через defer, следуя
                        // так называемому Go Way.
    s.i++
}

func (s *Counter) Value() int {
    s.mu.RLock()
    defer s.mu.RUnlock()
    return s.i

}
```

Naked return

Небольшой оффтоп от темы defer.

Именованные возвращаемые значения полезны для увеличения чистоты кода и понимания сигнатуры функции. Например:

```
// Не так очевидно.
func GetList(ctx context.Context) ([]feed.Object, *Cursor, *Cursor, error) { /* ... */ }

// Уже лучше.
func GetList(ctx context.Context) (page []feed.Object, higher, lower *Cursor, err error) { /* ... */ }
```

Но также они позволяют делать так называемые "голые" (naked / bare) возвраты. Простой пример:

```
func (c *cache) Get(userID string) (u *User, missed bool, err error)
{
    u, err = c.cache.Get(userID)
    if err != nil && !errors.Is(err, ErrNotFound) {
        err = fmt.Errorf("get user from cache: %w", err)
        return // <- Naked return.
    }

    // No user in cache.
    {
        u, err = c.storage.GetUser(userID)
        if err != nil {
            err = fmt.Errorf("get user from storage: %w", err)
            return // <- Naked return.
        }

        if err = c.cache.Set(userID, u); err != nil {
            err = fmt.Errorf("save user in cache: %w", err)
            return // <- Naked return.
        }

        missed = true
    }

    return // <- Naked return.
}
```

Проблема подобных `return`'ов состоит в том, что не всегда очевидно, что в данный момент возвращается из функции. Более сложный пример можно найти в go/types.MissingMethod.

Согласно гошным лучшим практикам рекомендуется использовать голые возвраты только для функций в несколько строк:

Naked returns are okay if the function is a handful of lines. ***Once it's a medium sized function, be explicit with your return values.*** *Corollary: it's not worth it to name result parameters just because it enables you to use naked returns. Clarity of docs is always more important than saving a line or two in your function.*

Finally, in some cases ***you need to name a result parameter in order to change it in a deferred closure. That is always OK.***

Более того существуют:

- Предложение в целом выпилить возможность голого возврата ([proposal: Go 2: remove bare return]()).
- Линтер [nakedret]() (входит в **golangci-lint**), проверяющий это дело.

Так что же делать? На самом деле отказ от **naked return** не означает отказ от именованных возвращаемых параметров, и мы без проблем можем использовать и то и другое разом:

```
func (c *cache) Get(userID string) (u *User, missed bool, err error)
{
    u, err = c.cache.Get(userID)
    if err != nil && !errors.Is(err, ErrNotFound) {
        return nil, false, fmt.Errorf("get user from cache: %w", err)
    }

    // No user in cache.
    {
        u, err = c.storage.GetUser(userID)
        if err != nil {
            return nil, false, fmt.Errorf("get user from storage:
%w", err)
        }

        if err = c.cache.Set(userID, u); err != nil {
```

```
            return nil, false, fmt.Errorf("save user in cache: %w", err)
        }
    }

    return u, true, nil

}
```

Пишите в комментариях, какой вариант вам больше по вкусу и каким практикам следуете вы 🙂

Тест "Named results shadowing"

У именованных возвращаемых значений есть ещё одна проблемка – они часто подвергаются затенению (**shadowing**).

Простой пример того, о чём идёт речь:

```
// https://goplay.tools/snippet/iYZeDM6KhB5

func main() {
    n := 0
    if true {
        n := 1 // Здесь создаётся новая n, а предыдущая n становится "затенённой".
        n++
    }
    fmt.Println(n) // 0, а не 2!
}
```

Большинство случаев **shadowing** отлавливаются компилятором (если вы не сидите на совсем древней версии гошки) или **go vet**, но бывает, что ошибка всё равно проскакивает, так что будьте бдительны.

В данном тесте вам необходимо ответить на вопрос: какая из ветвей выполнится в `main`?

```
func main() {
    if _, err := calculate(); err != nil {
        // 1
    } else {
        // 2
    }
}

func calculate() (n int, err error) {
    defer func() {
        // Если в результате (возможно) сложных и множественных вычислений ниже
        // мы получили отрицательное число, то значит в алгоритме ошибка.
        // Далее по курсу мы узнаем, что для таких случаев лучше подходит паника,
        // но цель этого примера состоит в другом.
        if n < 0 {
            n, err := 0, fmt.Errorf("invalid formula realization: got negative number %d", n)
        }
    }()

    // ...
    n = -1
    return

}
```

Исходник, чтобы поиграться (код выше и код по ссылке **отличаются**).

Промежуточные выводы

Отложенная функция имеет доступ к переменным из внешней области видимости и, как следствие, **может менять возвращаемые функцией результаты**, но, чтобы это работало, **они должны быть именованными**.

Код выполняется в следующем порядке:

1. Отрабатывает основная функция.
2. Происходит копирование значений в результаты функции (в случае именованных результатов копирования не происходит, так как они инициализируются заранее и работа идёт сразу с возвращаемым значением – см. ассемблер в начале урока).
3. Отрабатывают отложенные (deferred) функции.

Из этого мы получаем следующие следствия:

- отложенные функции работают с последними значениями, выставленными переменным во время работы обрамляющей функции;
- можно откладывать выход из критической секции, элегантно избегая гонок при выставлении результатов функции.

При использовании именованных возвращаемых аргументах следует помнить о таких понятиях, как **shadowing** и **naked return**.