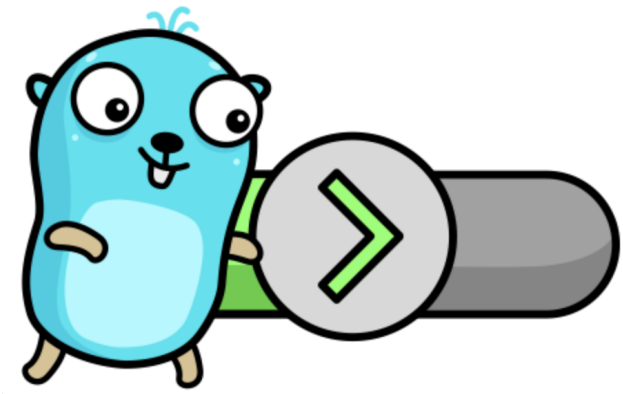
Оператор defer и его свойства

В этом уроке мы познакомимся с **defer statement** и его базовыми свойствами.

В данном и нескольких последующих уроках не будет большого количества задач – мы порешаем их в конце модуля, после знакомства с внутренним устройством `defer` и его влиянием на производительность программы.



Оператор defer

Отложенной (**deferred**) функцией считается функция, вызов которой отложен с помощью инструкции defer.

Отложенные функции выполняются после завершения окружающей их функции (которая может завершить свою работу естественным путём или благодаря return или ***панике***, о которой мы поговорим позже):

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

import "fmt"

func main() {
    defer func() {
        fmt.Println("HELLO!")   // Отработает последней.
    }()
    fmt.Println("WORLD")        // Отработает первой.
}
```

```
/*
WORLD
HELLO!
*/
```

При этом важно понимать, что откладывать можно только ***вызов*** функции или метода, а не сам вызываемый объект:

```
// Не скомпилируется.
defer func() {
    fmt.Println("HELLO!")

} // Нет скобочек - нет вызова анонимной функции.
```

Если вызовов `defer` несколько, то при выходе из функции они будут выполнены в обратном порядке, т.е. каждый вызов `defer` кладёт свой аргумент-функцию в [стек](#) отложенных функций:

```
// https://goplay.tools/snippet/w72lz_8tBW9
package main

import "fmt"

func main() {
    defer fmt.Println("one!")     // 4)
    defer fmt.Println("two!")     // 3)
    defer fmt.Println("three!")   // 2)

    fmt.Println("return")         // 1)
}

/*
return
three!
two!
one!

*/
```

Если вы боитесь запутаться в порядке вызовов отложенных функций, то можете объединить их в один `defer`:

```
// https://goplay.tools/snippet/D2VN8lDJBGS
package main

import "fmt"

func main() {
    defer func() {
        fmt.Println("three!")   // 2
        fmt.Println("two!")     // 3
        fmt.Println("one!")     // 4
    }()

    fmt.Println("return")       // 1
}

/*
return
three!
two!
one!

*/
```

Как мы видим из примеров выше, откладывать можно не только анонимные функции, но и именованные функции (в том числе и методы).

Для чего defer вообще нужен?

Прежде чем мы приступим к обзору многочисленных свойств `defer`, давайте посмотрим на исторические предпосылки к его появлению.

В этом нам поможет несколько сниппетов из исходного кода ОС Linux, написанной на Си – языке, являющегося одним из прародителей Go:

```
// kernel/dma/mapping.c

static struct sg_table *alloc_single_sgt(struct device *dev, size_t
size,
        enum dma_data_direction dir, gfp_t gfp)
{
    struct sg_table *sgt;
    struct page *page;

    sgt = kmalloc(sizeof(*sgt), gfp);
    if (!sgt)
        return NULL;
    if (sg_alloc_table(sgt, 1, gfp))
        goto out_free_sgt;                          // <- !!!
    page = __dma_alloc_pages(dev, size, &sgt->sgl->dma_address, dir,
gfp);
    if (!page)
        goto out_free_table;                        // <- !!!
    sg_set_page(sgt->sgl, page, PAGE_ALIGN(size), 0);
    sg_dma_len(sgt->sgl) = sgt->sgl->length;
    return sgt;
out_free_table:
    sg_free_table(sgt);
out_free_sgt:
    kfree(sgt);
    return NULL;

}

// kernel/sched/core.c

static int migrate_swap_stop(void *data)
{
    struct migration_swap_arg *arg = data;
    struct rq *src_rq, *dst_rq;
    int ret = -EAGAIN;

    if (!cpu_active(arg->src_cpu) || !cpu_active(arg->dst_cpu))
        return -EAGAIN;
```

```
    src_rq = cpu_rq(arg->src_cpu);
    dst_rq = cpu_rq(arg->dst_cpu);

    double_raw_lock(&arg->src_task->pi_lock,
            &arg->dst_task->pi_lock);
    double_rq_lock(src_rq, dst_rq);

    if (task_cpu(arg->dst_task) != arg->dst_cpu)
        goto unlock;                                    // <- !!!

    if (task_cpu(arg->src_task) != arg->src_cpu)
        goto unlock;                                    // <- !!!

    if (!cpumask_test_cpu(arg->dst_cpu, arg->src_task->cpus_ptr))
        goto unlock;                                    // <- !!!

    if (!cpumask_test_cpu(arg->src_cpu, arg->dst_task->cpus_ptr))
        goto unlock;                                    // <- !!!

    __migrate_swap_task(arg->src_task, arg->dst_cpu);
    __migrate_swap_task(arg->dst_task, arg->src_cpu);

    ret = 0;

unlock:
    double_rq_unlock(src_rq, dst_rq);
    raw_spin_unlock(&arg->dst_task->pi_lock);
    raw_spin_unlock(&arg->src_task->pi_lock);

    return ret;

}

// kernel/events/core.c

static int pmu_dev_alloc(struct pmu *pmu)
{
    int ret = -ENOMEM;

    pmu->dev = kzalloc(sizeof(struct device), GFP_KERNEL);
    if (!pmu->dev)
        goto out;                                       // <- !!!

    pmu->dev->groups = pmu->attr_groups;
    device_initialize(pmu->dev);
```

```
    ret = dev_set_name(pmu->dev, "%s", pmu->name);
    if (ret)
        goto free_dev;                                  // <- !!!

    dev_set_drvdata(pmu->dev, pmu);
    pmu->dev->bus = &pmu_bus;
    pmu->dev->release = pmu_dev_release;
    ret = device_add(pmu->dev);
    if (ret)
        goto free_dev;                                  // <- !!!

    if (pmu->nr_addr_filters)
        ret = device_create_file(pmu->dev,
&dev_attr_nr_addr_filters);

    if (ret)
        goto del_dev;                                   // <- !!!

    if (pmu->attr_update)
        ret = sysfs_update_groups(&pmu->dev->kobj, pmu->attr_update);

    if (ret)
        goto del_dev;                                   // <- !!!

out:
    return ret;

del_dev:
    device_del(pmu->dev);

free_dev:
    put_device(pmu->dev);
    goto out;                                           // Здесь вообще
goto "назад"!

}

// kernel/events/core.c

static int __init perf_event_sysfs_init(void)
{
    struct pmu *pmu;
    int ret;

    mutex_lock(&pmus_lock);
```

```
    ret = bus_register(&pmu_bus);
    if (ret)
        goto unlock;                                    // <- !!!

    list_for_each_entry(pmu, &pmus, entry) {
        if (!pmu->name || pmu->type < 0)
            continue;

        ret = pmu_dev_alloc(pmu);
        WARN(ret, "Failed to register pmu: %s, reason %d\n",
pmu->name, ret);
    }
    pmu_bus_running = 1;
    ret = 0;

unlock:
    mutex_unlock(&pmus_lock);

    return ret;

}
```

Мы видим, что переход к cleanup-операциям, которые необходимо выполнить при завершении работы функции (как успешном, так и неуспешном), повсеместно происходит с помощью goto.

Несмотря на то, что меткам стараются давать осмысленные имена (`out`, `unlock`, `free_dev` и т.д.), это всё равно может порождать запутанный код и вести к багам, связанным с утечками ресурсов.

А как код выше выглядел бы в Go?

defer vs goto

Исходник примеров.

Перепишем последний сниппет из предыдущего шага с Си на Go:

```
func perfEventSysfsInit() error {
    var err error
```

```
    pmusLock.Lock()

    if err = busRegister(); err != nil {
        goto unlock                          // Это абсолютно валидный
и компилируемый Go код.
    }

    for _, pmu := range pmus {
        if pmu.name == "" || pmu.type_ < 0 {
            continue
        }

        if err = pmuDevAlloc(pmu); err != nil {
            log.Printf("Failed to register pmu: %s, reason %v",
pmu.name, err)
        }
    }
    pmuBusRunning = 1
    err = nil

unlock:
    pmusLock.Unlock()

    return err

}
```

Избавимся от `goto` путём дублирования cleanup-операции (в нашем случае это освобождение мьютекса) по местам выхода из функции. Это также позволит избавиться от работы с ошибкой уровня функции `var err error` и позволит возвращать ошибки явно в местах их возникновения:

```
func perfEventSysfsInit() error {
    pmusLock.Lock()

    if err := busRegister(); err != nil {
        pmusLock.Unlock()                    // <- !!!
        return err
    }

    for _, pmu := range pmus {
        if pmu.name == "" || pmu.type_ < 0 {
            continue
```

```
        }

        if err := pmuDevAlloc(pmu); err != nil {
            log.Printf("Failed to register pmu: %s, reason %v",
pmu.name, err)
        }
    }
    pmuBusRunning = 1

    pmusLock.Unlock()                              // <- !!!
    return nil

}
```

Стало ли лучше? Непонятно, выглядит как шаг вперёд и два назад: да, теперь в наш `goto` не будут тыкать пальцем на ревью, но при дальнейшем рефакторинге и появлении новых мест выхода из функций, нужно не забывать разблокировать мьютекс:

```
// ...
    for _, pmu := range pmus {
        if pmu.name == "" || pmu.type_ < 0 {
            return                                 // Разработчик
заменил continue на return,
        }                                          // но забыл про
мьютекс. Получим deadlock.

// ...
```

Здесь нам на помощь и приходит `defer`, благодаря которому код выше превращается в идиоматически верный код на Go:

```
func perfEventSysfsInitV3() error {
    pmusLock.Lock()
    defer pmusLock.Unlock()                        // <- !!!

    if err := busRegister(); err != nil {
        return err
    }

    for _, pmu := range pmus {
```

```
        if pmu.name == "" || pmu.type_ < 0 {
            continue
        }

        if err := pmuDevAlloc(pmu); err != nil {
            log.Printf("Failed to register pmu: %s, reason %v",
pmu.name, err)
        }
    }
    pmuBusRunning = 1

    return nil

}
```

Таким образом, `defer` помогает единым кодом выполнять операции по освобождению ресурсов в функциях и методах, имеющих множество точек выхода и при этом избегать конструкций вида `goto`.

Обычно к списку *откладываемых* операций относят:

- закрытие различных соединений и итераторов;
- отмену контекста;
- декремент счётчиков;
- закрытие файловых дескрипторов;
- выход из критической секции;
- осуществление замера времени;
- работу с именованными возвращаемыми значениями (в том числе обогащение и/или установку возвращаемой ошибки);

и т.д. (пишите свои варианты в комментариях).

Кроме того у `defer` есть ещё одно важное преимущество перед `goto` – при возникновении программной ошибки (исключительной ситуации / паники) наша

функция может завершиться, так и не добравшись до cleanup-кода, что может породить утечку ресурсов. Отложенная же через `defer` функция выполнится в любом случае ~~за редким исключением~~ – подробнее об этом мы поговорим в следующем модуле.

Задача "Хитрый defer"

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

```
func main() {
    defer fmt.Println("1")
    defer func() {
        fmt.Println("2")
        fmt.Println("3")
    }()

    {
        defer fmt.Println("SCOPE")
    }

    for i := 11; i <= 13; i++ {
        defer fmt.Println(i)
        fmt.Println(i)            // <- Обратите внимание.
    }

    fmt.Println("RETURN")
}
```

Например:

```
SCOPE
11
12
13
RETURN
1
2
3
11
12

13
```

Напишите текст

Свойства defer: возвращаемые значения и scope

Предыдущая задача показала нам несколько свойств `defer`.

1) Если отложенная функция имеет возвращаемое значение (или значения), то они игнорируются:

```go
f, err := os.Open("file.txt")
if err != nil {
    // ...
}
defer f.Close()

// defer'у всё равно, что Close может завершиться с ошибкой:

// func (f *File) Close() error
```

Важно понимать, что подобные вызовы приводят к необработанной ошибке:

Что с этим делать мы узнаем в отдельном уроке дальше по курсу.

2) Отложенная функция вызывается при выходе из текущей функции, а не при выходе из текущей области видимости!

Это может сбить с толку, например, разработчиков C++, которые привыкли, что при выходе из текущего scope локальные объекты разрушаются, вызываются деструкторы и т.д. – `defer` работает не так:

```
{
    defer fmt.Println("SCOPE") // Вызовется после return!
}

for i := 11; i <= 13; i++ {
    defer fmt.Println(i) // Вызовется после return!
    fmt.Println(i)

}
```

Задача "Утечка файловых дескрипторов"

Ссылка на заготовку.

Разработчик реализовал следующую функцию обработки массива файлов:

```
func ProcessFiles(paths []string) ([]string, error) {
    results := make([]string, 0, len(paths))

    for _, p := range paths {
        f, err := os.Open(p)
        if err != nil {
            return nil, fmt.Errorf("open file %q: %v", p, err)
        }
        defer f.Close()

        // ...

        r, err := processFile(f)
        if err != nil {
            return nil, fmt.Errorf("process file %q: %v", p, err)
        }
```

```
        // ...
    }

    return results, nil

}
```

И всё было хорошо, но на большом количестве файлов она начала возвращать ошибку:

```
open /var/folders/6t/v80c8sfs5zqf38b2yhzq592h0000gn/T/index.txt: too many open files
```

Вам предлагается исправить функцию так, чтобы ключевая логика не поменялась, но утечек больше не было.

К сожалению, мы не можем проверить эту задачу средствами Stepik (спасибо его read-only файловой системе), поэтому она остаётся на самостоятельное изучение, совесть и желание студента:

```
$ cd tasks/02-defer-statement/file-descriptors-leak
$ go test -v ./...
=== RUN   TestProcessFiles
--- PASS: TestProcessFiles (0.92s)
=== RUN   TestProcessFiles_SkipAll
--- PASS: TestProcessFiles_SkipAll (0.71s)
PASS
ok      tasks/02-defer-statement/file-descriptors-leak       1.916s
```

Не спешите жмакать "Отправить" и переходить к следующему шагу! Откройте задачу в IDE и поиграйтесь с кодом. Данная проблема не часто, но выстреливает в реальном коде у серьёзных дядек (в различных вариациях, но смысл один и тот же), и её понимание очень важно.

Свойства defer: вычисление аргументов откладываемой функции

Хоть `defer` и выполняет функцию по выходу из программы, аргументы вызываемой функции вычисляются в момент самого вызова `defer`, т.е. сразу же:

```
// https://goplay.tools/snippet/uNh3gVmHIxY

func main() {
    i := 3
    defer fmt.Println("i =", i)
    i = 100
}

// Напечатает:

// i = 3
```

Это работает и для более сложных аргументов, являющихся результатами сложносоставных выражений, вызовов функций и т.д.

```
// https://goplay.tools/snippet/hJ1wgVkJ1pr

func main() {
    s := map[int]string{100: "foo"}

    defer fmt.Println("v =", s[100]+"_deferred")
    s[100] = "bar"
}

// Напечатает:

// v = foo_deferred
```

Задача "defer и вызов анонимной функции"

Видоизменим последний пример из предыдущего шага:

```
func main() {
    s := map[int]string{100: "foo"}

    defer func() {
        fmt.Println("v =", s[100]+"_deferred")
    }()
    s[100] = "bar"

}
```

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

Напишите текст

Задача "defer и вызов метода - 1"

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

```
type User struct {
    email string
}

func NewUser(e string) *User {
    return &User{email: e}
}

func (u *User) SetEmail(e string) { u.email = e }
func (u *User) Email() string     { return u.email }

func processUser(u *User) {
    defer fmt.Printf("user %q was processed", u.Email())
    defer u.SetEmail("unknown")
    // ...
}

func main() {
    u := NewUser("info@golang-courses.ru")
    processUser(u)

}
```

Напишите текст

Задача "defer и вызов метода - 2"

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

```
type User struct {
```

```
    email string
}

func NewUser(e string) *User {
    return &User{email: e}
}

func (u *User) SetEmail(e string) { u.email = e }
func (u *User) Email() string     { return u.email }
func (u *User) PrintEmail()       { fmt.Printf("user %q was
processed", u.Email()) }

func processUser(u *User) {
    defer u.PrintEmail()
    defer u.SetEmail("unknown")
    // ...
}

func main() {
    u := NewUser("info@golang-courses.ru")
    processUser(u)

}
```

Напишите текст

Задача "defer и вызов метода - 3"

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

```
type User struct {
    email string
}

func NewUser(e string) *User {
    return &User{email: e}
}

func (u *User) SetEmail(e string) { u.email = e }
func (u *User) Email() string     { return u.email }
func (u User) PrintEmail()        { fmt.Printf("user %q was
processed", u.Email()) }
```

```
func processUser(u *User) {
    defer u.PrintEmail()
    defer u.SetEmail("unknown")
    // ...
}

func main() {
    u := NewUser("info@golang-courses.ru")
    processUser(u)
}
```

P.S. Обратите внимание на изменение сигнатуры метода `PrintEmail`.

Напишите текст

---

defer и встроенные функции

Откладывать с помощью `defer` можно только вызов функции или метода, при этом вызов нельзя обрамлять в скобочки:

```
func main() {
    defer (fmt.Println("HELLO")) // Не скомпилируется.
}


// ./main.go:7:8: expression in defer must not be parenthesized
```

Кроме того есть список исключений из [встроенных (built-in)](#) функций, к которым нельзя применять `defer`:

- `append`
- `cap`
- `complex`
- `imag`
- `len`

- `make`
- `new`
- `real`

Также дополнительно нельзя отложить:

- `unsafe.Alignof`
- `unsafe.Offsetof`
- `unsafe.Sizeof`

```
func main() {
    defer new(struct{}) // Не скомпилируется.
}


// ./main.go:4:2: defer discards result of new(struct {})

func main() {
    defer int64(100) // Не скомпилируется.
}


// ./main.go:4:2: defer requires function call, not conversion
```

Как видим, в случае ошибки компилятор поможет нам. Это стало возможным благодаря данному issue, во времена, когда гошный компилятор ещё был написан на Си.

Тест "Выберите валидные (компилируемые) варианты использования defer"

На самом деле мы не любим задачи, которые требуют от вас быть компилятором.

Так что цель этого теста – просто проверить, что вы не проигнорировали материал предыдущих шагов.

Выберите все подходящие ответы из списка

- ☐ defer func() { nums = append(nums, 10) }()
- ☐ defer float64(100)
- ☐ defer (mu.Unlock())
- ☐ defer { result = float64(100) }
- ☐ defer (*Service).StaticMethod(&svc)
- ☐ defer func() { fmt.Println(`¯\_(ツ)_/¯`) }
- ☐ defer go worker()
- ☐ defer (go worker())
- ☐ defer func() { go worker() }()
- ☐ defer worker()

Промежуточные выводы

Мы познакомились с `defer` – оператором, позволяющим нам **откладывать** выполнение функций и методов на момент завершения текущей функции.

Мы узнали, что

- `defer` игнорирует возвращаемые значения откладываемой функции;
- на `defer` никак не влияет завершение текущей области видимости – отложенная функция отработает только при выходе из текущей функции;
- аргументы отложенной функции вычисляются сразу же, в момент обработки `defer`;
- можно откладывать любые функции, в том числе методы и анонимные функции;
- нельзя делать `defer {/*...*/}`, `defer (/*...*/)`, `defer go /*...*/` и пр. несуразности, от которых компилятор нас спасёт.

В следующем уроке мы посмотрим, как с помощью `defer` менять возвращаемые текущей функцией результаты и разберём на уровне ассемблера, почему это вообще работает (или может не работать).