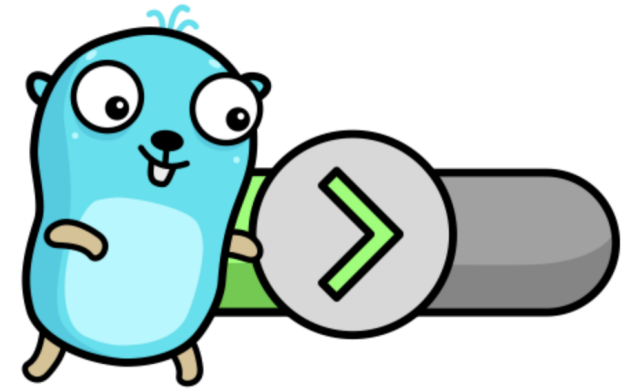
defer: внутреннее устройство и производительность

В предыдущих уроках мы вдоволь поиграли с различными особенностями `defer`, теперь пришло время влезть к нему под капот.

Бытует мнение, что `defer` имеет большие накладные расходы (overhead) и множественное его использование существенно замедляет программу.

В серии уроков мы разложим по полочкам, как устроен `defer` изнутри и действительно ли стоит избегать его.

P.S. Обратите внимание, что бенчмарки в данном уроке запускаются через **Go 1.12** – это сделано преднамеренно, чтобы в следующих уроках показать эволюцию `defer`.



Производительность defer

Давайте напишем небольшой пример (исходник):

```
package main

//go:noinline
func withoutDefer() (result int) {
    foo(&result)
    return
}

//go:noinline
func withDefer() (result int) {
    defer foo(&result)
    return
}

//go:noinline
func withDefers() (result int) {
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    return
}

func foo(i *int) {
    *i++
}
```

И бенчмарки для него:

```
package main

import "testing"

func Benchmark_withoutDefer(b *testing.B) {
    for n := 0; n < b.N; n++ {
        withoutDefer()
    }
}

func Benchmark_withDefer(b *testing.B) {
    for n := 0; n < b.N; n++ {
        withDefer()
    }
}

func Benchmark_withDefers(b *testing.B) {
    for n := 0; n < b.N; n++ {
        withDefers()
    }
}
```

Замерим всё это дело:

```
$ go1.12 version
go version go1.12 darwin/amd64

$ go1.12 test -benchmem -bench .
Benchmark_withoutDefer-12       2000000000  1.28 ns/op       0 B/op
0 allocs/op
Benchmark_withDefer-12          30000000    40.4 ns/op       0 B/op
0 allocs/op
Benchmark_withDefers-12         20000000    100 ns/op        0 B/op
0 allocs/op
PASS
ok      examples/02-defer-statement/performance-1           6.335s
```

Как и ожидалось, аллокаций не наблюдается. Но мы видим, что действительно наличие одного и трёх `defer` замедляет программу в ~30 и в ~80 раз

соответственно. Опустим замечание, что речь идёт о наносекундах, и зададимся вопросом – почему же так происходит?

defer под капотом

Для ответа на вопрос "Почему же так происходит?" придётся посмотреть, какой ассемблерный код для нашего примера генерирует компилятор:

- `withoutDefer`

```
# withoutDefer
        TEXT    "".withoutDefer(SB), NOSPLIT|ABIInternal, $0-8
        FUNCDATA        $0,
gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        FUNCDATA        $1,
gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        FUNCDATA        $3,
gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        PCDATA  $2, $0
        PCDATA  $0, $0
        MOVQ    $0, "".result+8(SP)
        XCHGL   AX, AX
        MOVQ    $1, "".result+8(SP)

        RET
```

- `withDefer`

```
# withDefer
        TEXT    "".withDefer(SB), ABIInternal, $32-8
        MOVQ    (TLS), CX
        CMPQ    SP, 16(CX)
        JLS     withDefer_pc108
        SUBQ    $32, SP
        MOVQ    BP, 24(SP)
        LEAQ    24(SP), BP
        FUNCDATA        $0,
gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        FUNCDATA        $1,
gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
```

```
        FUNCDATA        $3, gclocals·9fb7f0986f647f17cb53dda1484e0f7a(SB)
        PCDATA  $2, $0
        PCDATA  $0, $0
        MOVQ    $0, "".result+40(SP)
        MOVL    $8, (SP)
        PCDATA  $2, $1
        LEAQ    "".foo·f(SB), AX
        PCDATA  $2, $0
        MOVQ    AX, 8(SP)
        PCDATA  $2, $1
        LEAQ    "".result+40(SP), AX
        PCDATA  $2, $0
        MOVQ    AX, 16(SP)
        CALL    runtime.deferproc(SB)
        TESTL   AX, AX
        JNE     withDefer_pc92
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    24(SP), BP
        ADDQ    $32, SP
        RET
withDefer_pc92:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    24(SP), BP
        ADDQ    $32, SP

        RET
```

- withDefers

```
# withDefers
        TEXT    "".withDefers(SB), ABIInternal, $32-8
        MOVQ    (TLS), CX
        CMPQ    SP, 16(CX)
        JLS     withDefers_pc220
        SUBQ    $32, SP
        MOVQ    BP, 24(SP)
        LEAQ    24(SP), BP
        FUNCDATA        $0, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
```

```
        FUNCDATA        $1, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        FUNCDATA        $3, gclocals·1cf923758aae2e428391d1783fe59973(SB)
        PCDATA  $2, $0
        PCDATA  $0, $0
        MOVQ    $0, "".result+40(SP)
        MOVL    $8, (SP)
        PCDATA  $2, $1
        LEAQ    "".foo·f(SB), AX
        PCDATA  $2, $0
        MOVQ    AX, 8(SP)
        PCDATA  $2, $2
        LEAQ    "".result+40(SP), CX
        PCDATA  $2, $0
        MOVQ    CX, 16(SP)
        CALL    runtime.deferproc(SB)
        TESTL   AX, AX
        JNE     withDefers_pc204
        MOVL    $8, (SP)
        PCDATA  $2, $1
        LEAQ    "".foo·f(SB), AX
        PCDATA  $2, $0
        MOVQ    AX, 8(SP)
        PCDATA  $2, $2
        LEAQ    "".result+40(SP), CX
        PCDATA  $2, $0
        MOVQ    CX, 16(SP)
        CALL    runtime.deferproc(SB)
        TESTL   AX, AX
        JNE     withDefers_pc188
        MOVL    $8, (SP)
        PCDATA  $2, $1
        LEAQ    "".foo·f(SB), AX
        PCDATA  $2, $0
        MOVQ    AX, 8(SP)
        PCDATA  $2, $1
        LEAQ    "".result+40(SP), AX
        PCDATA  $2, $0
        MOVQ    AX, 16(SP)
        CALL    runtime.deferproc(SB)
        TESTL   AX, AX
        JNE     withDefers_pc172
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
```

```
        MOVQ    24(SP), BP
        ADDQ    $32, SP
        RET
withDefers_pc172:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    24(SP), BP
        ADDQ    $32, SP
        RET
withDefers_pc188:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    24(SP), BP
        ADDQ    $32, SP
        RET
withDefers_pc204:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    24(SP), BP
        ADDQ    $32, SP

        RET
```

Невооружённым взглядом видно, что по мере увеличения количества `defer` увеличивается количество ассемблерного кода.

В чём же дело?

Обратим внимание на `withDefers` и выделим следующие вызовы:

```
# withDefers
        TEXT    "".withDefers(SB), ABIInternal, $32-8

        # ...
        LEAQ    "".foo·f(SB), AX
        MOVQ    AX, 8(SP)
        LEAQ    "".result+40(SP), CX
        MOVQ    CX, 16(SP)
        # В аргументы deferproc попадает адрес отложенной функции foo
и её аргумента result.
        CALL    runtime.deferproc(SB)
        TESTL   AX, AX
        JNE     withDefers_pc204
```

```
        # ...
        CALL    runtime.deferproc(SB)
        TESTL   AX, AX
        JNE     withDefers_pc188

        # ...
        CALL    runtime.deferproc(SB)
        TESTL   AX, AX
        JNE     withDefers_pc172

        # ...
        CALL    runtime.deferreturn(SB)
        MOVQ    24(SP), BP
        ADDQ    $32, SP
        RET

withDefers_pc172:
        # ...
        CALL    runtime.deferreturn(SB)
        # ...
        RET
withDefers_pc188:
        # ...
        CALL    runtime.deferreturn(SB)
        # ...
        RET
withDefers_pc204:
        # ...
        CALL    runtime.deferreturn(SB)
        # ...

        RET
```

Т.е. на каждый `defer` в нашем коде мы получили пару вызовов `runtime.deferproc` и `runtime.deferreturn`, а также дополнительный `runtime.deferreturn` при выходе из функции:

Наверняка, в них-то и зарыта собака потери производительности!

runtime.deferproc

runtime.deferproc создаёт новый объект _defer через newdefer и помещает его в начало списка `defer`'ов, связанных с **текущей** горутиной. **Запомните этот факт**, он поможет нам в дальнейшем при работе с **паникой**.

При этом горутина ссылается на голову списка своих `defer`'ов, т.е. на `defer`, который был создан последним:

```
package runtime

type g struct {
    // ...
    _panic         *_panic // innermost panic - offset known to
liblink
    _defer         *_defer // innermost defer
    m              *m      // current m; offset known to arm liblink
    // ...
}

// A _defer holds an entry on the list of deferred calls.
// If you add a field here, add code to clear it in freedefer.
type _defer struct {
    siz     int32
    started bool
    sp      uintptr // sp at time of defer
    pc      uintptr
```

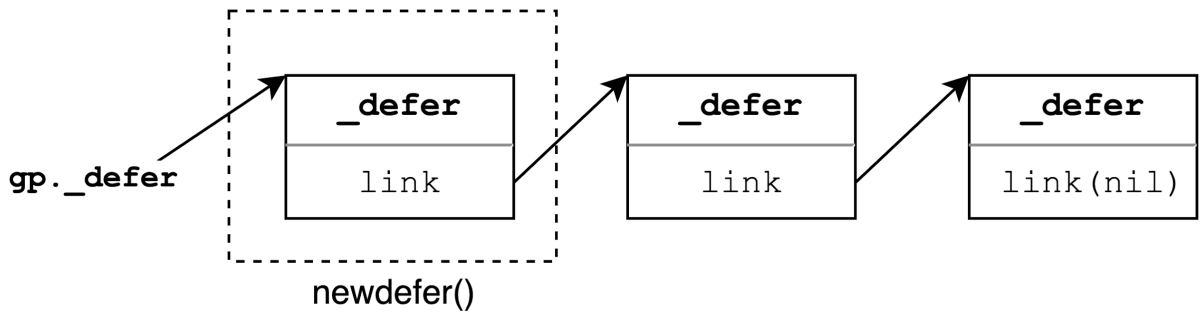
```
    fn      *funcval
    _panic  *_panic // panic that is running defer
    link    *_defer

}
```

По сути мы получили реализацию стека через односвязный список:

```
package runtime

// Create a new deferred function fn with siz bytes of arguments.
// The compiler turns a defer statement into a call to this.
// ...
func deferproc(siz int32, fn *funcval) { // arguments of fn follow fn
    // ...

    sp := getcallersp()
    argp := uintptr(unsafe.Pointer(&fn)) + unsafe.Sizeof(fn)
    callerpc := getcallerpc()

    d := newdefer(siz)
    // ...
    d.fn = fn                                // А вот и сама отложенная
функция.
    d.pc = callerpc
    d.sp = sp
    switch siz {                             // А вот и сохраняем аргументы
отложенной функции.
    case 0:
        // Do nothing.
    case sys.PtrSize:
        *(*uintptr)(deferArgs(d)) = *(*uintptr)(unsafe.Pointer(argp))
    default:
        memmove(deferArgs(d), unsafe.Pointer(argp), uintptr(siz))
    }

    // Обратите внимание на комментарий ниже, он соответствует
    // ассемблерному коду с предыдущего шага.
    // Что здесь происходит и зачем, мы узнаем дальше по курсу.
    // ↓

    // deferproc returns 0 normally.
    // a deferred func that stops a panic
```

```
    // makes the deferproc return 1.
    // the code the compiler generates always
    // checks the return value and jumps to the
    // end of the function if deferproc returns != 0.
    return0()
}

// The arguments associated with a deferred call are stored
// immediately after the _defer header in memory.
// ...
func deferArgs(d *_defer) unsafe.Pointer {
    if d.siz == 0 {
        return nil
    }
    return add(unsafe.Pointer(d), unsafe.Sizeof(*d))
}
```

Разработчики Go называют этот список ***defer chain***:



При этом newdefer:

- аллоцирует объект `_defer` на куче;
- аллоцирует место под аргументы отложенной функции и "прикапывает" их для дальнейшего использования;
- работает с пулом `defer`'ов (своеобразный кэш на каждом логическом Go-процессоре P) и системным стеком, что требует определённых синхронизаций;

- делает новый `_defer` головой связного списка `_defer`'ов и перевязывает горутину на него.

```
package runtime

// Allocate a Defer, usually using per-P pool.
// Each defer must be released with freedefer.
// ...
func newdefer(siz int32) *_defer {
    var d *_defer
    // ...
    gp := getg()         // Получаем указатель на текущую горутину.

    // Много кода по работе с пулом defer'ов,
    // аллокации памяти для d и её заполнения.
    // ...

    d.link = gp._defer  // Работаем с defer chain.
    gp._defer = d
    return d

}
```

Всё это и занимает драгоценное процессорное время.

Тест "С помощью какой структуры данных реализован defer?

Выберите один вариант из списка

- ☐ Массив
- ☐ Куча
- ☐ Дек
- ☐ Однонаправленный связный список
- ☐ Двунаправленный связный список
- ☐ B+ дерево

runtime.deferreturn

Функция runtime.deferreturn является антагонистом `runtime.deferproc`: именно она и выполняет цепочку `defer`'ов, созданных до этого:

```
package runtime

// Run a deferred function if there is one.
//
// The compiler inserts a call to this at the end of any
// function which calls defer.
//
// If there is a deferred function, this will call runtime.jmpdefer,
// which will jump to the deferred function such that it appears
// to have been called by the caller of deferreturn at the point
// just before deferreturn was called. The effect is that deferreturn
// is called again and again until there are no more deferred
functions.
// ...
func deferreturn(arg0 uintptr) {
    gp := getg()                // Получаем указатель на текущую
горутину.
    d := gp._defer
    if d == nil {               // Нет defer'ов и выполнять нечего.
        return
    }

    sp := getcallersp()
    if d.sp != sp {
        return
    }

    // ...
    switch d.siz {              // Достаём аргументы отложенной
функции.
    case 0:
        // Do nothing.
    case sys.PtrSize:
        *(*uintptr)(unsafe.Pointer(&arg0)) =
*(*uintptr)(deferArgs(d))
    default:
        memmove(unsafe.Pointer(&arg0), deferArgs(d), uintptr(d.siz))
    }
    fn := d.fn                  // Достаём отложенную функцию.
    d.fn = nil

    gp._defer = d.link          // Удаляем defer из defer chain.
```

```
    freedefer(d)

    // Имя на руках отложенную функцию и её аргументы,
    // делаем хитрый jump.
    jmpdefer(fn, uintptr(unsafe.Pointer(&arg0)))
}
```

Мы видим, что `runtime.deferreturn`:

- находит функцию, отложенную последним `defer`'ом в горутине;
- "перевешивает" горутину на `defer`, находящийся перед ним и освобождает текущий `defer` с помощью [freedefer](#);
- переходит к выполнению отложенной функции с помощью [runtime.jmpdefer](#).

При этом `runtime.jmpdefer` работает таким ~~неочевидным~~ образом, что после выполнения отложенной функции возвращает управление на место перед вызовом `runtime.deferreturn`. По сути мы получаем цикл вида (псевдокод):

```
d := gp._defer
for d != nil { // Пока есть defer'ы.
    d()        // Выполняем defer.
    d = d.link // Переходим к предыдущему defer.
}
```

Этим и объясняется **обратный порядок вызовов** `defer` при выходе из функции.

Если снять CPU-профиль для наших бенчмарков, то можно увидеть, что `runtime.deferreturn` ещё более "жирный", чем `runtime.deferproc`:

```
$ go1.12 test -cpuprofile defers.out -bench .

$ go tool pprof defers.out
(pprof) top runtime 5
Showing top 5 nodes out of 17
      flat  flat%   sum%        cum   cum%
     690ms 12.41% 12.41%      980ms 17.63%  runtime.deferreturn
     680ms 12.23% 24.64%      690ms 12.41%  runtime.newdefer
     300ms  5.40% 30.04%     1030ms 18.53%  runtime.deferproc
     270ms  4.86% 34.89%      270ms  4.86%  runtime.freedefer
     100ms  1.80% 36.69%      100ms  1.80%  runtime.jmpdefer
(pprof)

// При ещё большем количестве defer'ов можно заметить, что их
выполнение
// и освобождение гораздо "тяжелее" инициализации (что в целом
очевидно):

Showing top 5 nodes out of 24
      flat  flat%   sum%        cum   cum%
     690ms 18.80% 18.80%     1270ms 34.60%  runtime.deferreturn
     340ms  9.26% 28.07%      350ms  9.54%  runtime.freedefer
     330ms  8.99% 37.06%      330ms  8.99%  runtime.jmpdefer
     180ms  4.90% 41.96%      180ms  4.90%  runtime.newdefer
     160ms  4.36% 46.32%      390ms 10.63%  runtime.deferproc
```

Тест "runtime.deferreturn много не бывает"

Сколько вызовов `runtime.deferreturn` мы увидим в ассемблере после компиляции данной функции в Go 1.12?

```
func withTenDefers()  {
    defer foo()
    defer foo()
    defer foo()
    defer foo()
    defer foo()
    defer foo()
    defer foo()
    defer foo()
    defer foo()
    defer foo()

}
```

([подсказка])

Введите численный ответ

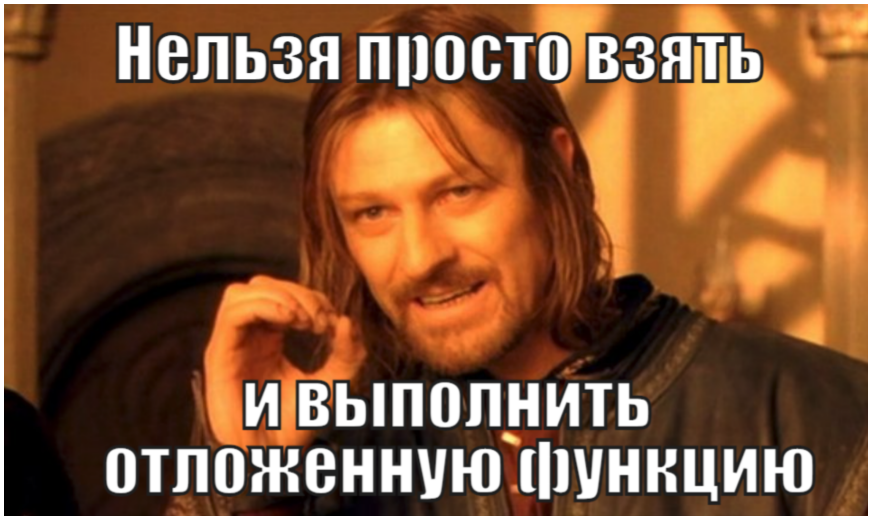
_____

runtime.deferreturn: не всё так просто :)

После понимания устройства `runtime.deferreturn` в голове возникает следующая картина:

```
func withDefers() {
    defer foo() // runtime.deferproc()
    defer foo() // runtime.deferproc()
    defer foo() // runtime.deferproc()
    // Единственный вызов, который по сути превращается в цикл обработки defer'ов выше:
    // runtime.deferreturn()

}
```

Т.е. кажется, что нам достаточно одной `runtime.deferreturn` в конце функции, чтобы обработать все `defer`'ы, но почему тогда в предыдущих примерах мы видели несколько `deferreturn`?



Вернёмся к комментарию в конце `deferproc`:

```
package runtime

func deferproc(siz int32, fn *funcval) {
    // ...

    // deferproc returns 0 normally.
    // a deferred func that stops a panic
    // makes the deferproc return 1.
    // the code the compiler generates always
    // checks the return value and jumps to the
    // end of the function if deferproc returns != 0.
    return0()
    // No code can go here - the C return register has
    // been set and must not be clobbered.

}
```

И посмотрим на return0, которая, конечно же, реализована на ассемблере:

```
package runtime

// return0 is a stub used to return 0 from deferproc.
// It is called at the very end of deferproc to signal
// the calling Go function that it should not jump
// to deferreturn.
// in asm_*.s
```

```
func return0()

TEXT runtime·return0(SB), NOSPLIT, $0
      MOVL  $0, AX

      RET
```

Таким образом, в "нормальной" ситуации после `deferproc` мы имеем регистр `AX`, выставленный в 0.

Посмотрим ещё разик на `withDefers`:

```
# withDefers
        TEXT    "".withDefers(SB), ABIInternal, $32-8

        # ...
        CALL    runtime.deferproc(SB)
        TESTL   AX, AX                          # Мы видим, что после
каждой deferproc
        JNE     withDefers_pc204                # вставлена проверка, что
"ЕСЛИ AX == 1, то
                                                # прыгай к соответствующей
deferreturn".
        # ...
        CALL    runtime.deferproc(SB)
        TESTL   AX, AX
        JNE     withDefers_pc188

        # ...
        CALL    runtime.deferproc(SB)
        TESTL   AX, AX
        JNE     withDefers_pc172

        # Финальная, располагающаяся в конце функции deferreturn.
        # Если ни одна из deferproc выше не выставила AX в 1, то
        # именно эта defereturn будет обрабатывать цепочку defer'ов.
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    24(SP), BP
        ADDQ    $32, SP
        RET
withDefers_pc172:
        XCHGL   AX, AX
```

```
        CALL    runtime.deferreturn(SB)
        MOVQ    24(SP), BP
        ADDQ    $32, SP
        RET
withDefers_pc188:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    24(SP), BP
        ADDQ    $32, SP
        RET
withDefers_pc204:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    24(SP), BP
        ADDQ    $32, SP

        RET
```

Т.е. при каком-то магическом стечении обстоятельств после `deferproc` может оказаться `AX == 1`, и тогда код прыгнет к отдельностоящей `deferreturn`.

В 99% случаях (при нормальной работе программы) данные ветки не понадобятся и использоваться не будут, но когда такое всё-таки может произойти и как это вообще работает, мы узнаем в следующих уроках, познакомившись с понятием **паники**.

Задача "defer at home"

Ссылка на заготовку.

Me: mum can i have a ps5

Mum: no we already have a ps5 at home.

Ps5 at home:

Мы предлагаем вам реализовать свой тип `Defer`, удовлетворяющий следующему интерфейсу:

```
type IDefer interface {
    // Defer добавляет функцию в стек отложенных функций.
    Defer(f func())

    // Execute выполняет функции, добавленные через Defer,
    // в порядке, обратном порядку добавления.
    Execute()

}
```

Пример использования:

```
d := NewDefer()
for _, c := range "hello" {
    c := c
    d.Defer(func() { fmt.Print(string(c))})
}
d.Execute()

// Output:

// olleh
```

Предполагается, что `Execute` вызывается единожды и после него работа с объектом типа `*Defer` заканчивается. Вызов нескольких `Execute` подряд или `Defer - Execute - Defer - Execute` считается undefined behavior.

Назад в будущее

Мы узнали, что `defer` действительно небесплатный (эх, за всё в этом мире приходится платить), и познакомились с `runtime.deferproc` и `runtime.deferreturn`, стоящими за этим. Уже сейчас становится понятно, что `defer` крайне непростая в реализации штука и требует неочевидной коллаборации компилятора и рантайма языка (ух, это мы ещё до паники не добрались 🙈).

В следующем уроке мы посмотрим, какие оптимизации сделали разработчики Go, чтобы нивелировать накладные расходы от `defer` и сделать его более приятным к использованию.

После этого мы сможем удостовериться, действительно ли всё так плохо, как показалось?