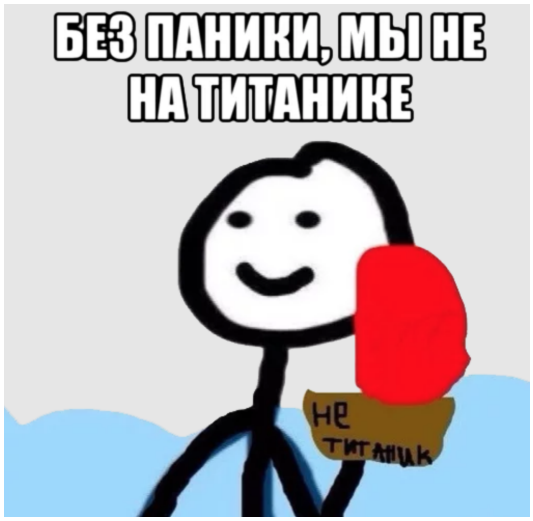
Механизм восстановления после паники: recover

В данном уроке мы научимся справляться с возникающими паниками, а также коснёмся первых подводных камней, связанных с этим.



Встроенная функция recover

Антагонистом `panic` является встроенная функция recover:

```
// src/builtin/builtin.go
package builtin

// The recover built-in function allows a program to manage behavior of a
// panicking goroutine. Executing a call to recover inside a deferred
// function (but not any function called by it) stops the panicking sequence
// by restoring normal execution and retrieves the error value passed to the
// call of panic. If recover is called outside the deferred function it will
// not stop a panicking sequence. In this case, or when the goroutine is not
// panicking, or if the argument supplied to panic was nil, recover returns
```

```
// nil. Thus the return value from recover reports whether the
goroutine is
// panicking.
func recover() interface{}
```

Она позволяет останавливать запущенную панику, а также получить аргумент, с которым та была вызвана (мы не просто так приводим функции вместе с их документацией – чаще всего она может дать ответы на все вопросы).

Пример использования:

```
// https://goplay.tools/snippet/d3BjJZhhcdD

func main() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Printf("recovered: %v (%T)", r, r)
        }
    }()

    _ = ([]int)(nil)[0]

    // Процесс завершится успешно, с кодом возврата 0.
}


// recovered: runtime error: index out of range [0] with length 0
(runtime.boundsError)
```

Если в момент вызова `recover` не происходило никакой паники, то функция вернёт `nil` (аналогично, если бы был вызов `panic(nil)`):

```
// https://goplay.tools/snippet/v_q_QqO8maR

func main() {
    defer func() {
        r := recover()
        fmt.Printf("recovered: %v (%T)", r, r)
    }()

    // No panic.
}
```

```
// recovered: <nil> (<nil>)
```

Обратите внимание на то, что `recover` вызывается из отложенной функции!

Так же следует иметь в виду, что после восстановления функция вернётся в место своего вызова с возвращаемыми значениями, заполненными по умолчанию, и программа продолжит свою работу:

```
// https://goplay.tools/snippet/mNNR-iUqhlp

func main() {
    a, b, c := noPanic()
    fmt.Printf("%v %v %q", a, b, c)
}

func noPanic() (int32, bool, string) {
    defer func() {
        recover()
    }()

    panic("sky is falling")
    return 1, true, "1"           // До этой строчки не дойдёт!
}


// 0 false ""
```

Если же на момент паники возвращаемые значения были чем-то заполнены, то при выходе из функции мы их и получим:

```
// https://goplay.tools/snippet/rG7UWavRAHD

func main() {
    a, b, c := noPanic()
    fmt.Printf("%v %v %q", a, b, c)
}

func noPanic() (a int32, b bool, c string) {
    defer func() {
        recover()
    }()
```

```
    a, b, c = 2, false, "2"

    panic("sky is falling")
    return 1, true, "1"
}


// 2 false "2"
```

Задача "Recover"

Ссылка на заготовку.

Вам необходимо реализовать небольшую функцию `Recover`:

```
// Recover вызывает fn, ловит панику и возвращает её аргумент.
// Если паники не было, то функция возвращает <nil>.

func Recover(fn func()) any
```

Больше подробностей и не понадобится 🙂

recover под капотом

Внимание! Приготовьтесь к непростому лонгриду.

Привычным нам способом посмотрим, в какой ассемблерный код превращается вызов `recover`:

```
# main
    # ...
    LEAQ    +24(FP), AX
    CALL    runtime.gorecover(SB)

    # ...
```

Этот вызов генерируется следующим кодом компилятора:

```
// cmd/compile/internal/walk/builtin.go
package walk

// walkRecover walks an ORECOVER node.
func walkRecover(nn *ir.CallExpr, init *ir.Nodes) ir.Node {
    // Call gorecover with the FP of this frame.
    // FP is equal to caller's SP plus FixedFrameSize().
    var fp ir.Node = mkcall("getcallersp",
types.Types[types.TUINTPTR], init)
    if off := base.Ctxt.FixedFrameSize(); off != 0 {
        fp = ir.NewBinaryExpr(fp.Pos(), ir.OADD, fp, ir.NewInt(off))
    }
    fp = ir.NewConvExpr(fp.Pos(), ir.OCONVNOP,
types.NewPtr(types.Types[types.TINT32]), fp)
    return mkcall("gorecover", nn.Type(), init, fp)
```

```
}
```

Запомним, что `runtime.gorecover` принимает в качестве аргумента **FP** – указатель на текущий [фрейм](#):

FP == (virtual) frame pointer == arguments pointer (т.к. через FP ссылаются на аргументы функции и её локальные переменные). Можно считать аналогом аппаратного BP (base pointer).

The `FP` pseudo-register is a virtual frame pointer used to refer to function arguments. The compilers maintain a virtual frame pointer and refer to the arguments on the stack as offsets from that pseudo-register. (c) [A Quick Guide to Go's Assembler](#)

Реализация [runtime.gorecover](#) на первый взгляд тривиальна (суть работы функции сводится к `p.recovered = true`), но содержит интересный комментарий, который мы разберём в следующем шаге:

```
package runtime

// The implementation of the predeclared function recover.
// Cannot split the stack because it needs to reliably
// find the stack segment of its caller.
// ...
func gorecover(argp uintptr) interface{} {
    // Must be in a function running as part of a deferred call
during the panic.
    // Must be called from the topmost function of the call
    // (the function used in the defer statement).
    // p.argp is the argument pointer of that topmost deferred
function call.
    // Compare against argp reported by caller.
    // If they match, the caller is the one who can recover.
    gp := getg()
    p := gp._panic
    if p != nil && !p.goexit && !p.recovered && argp ==
uintptr(p.argp) {
        p.recovered = true
```

```
            return p.arg
        }
        return nil

}
```

Обратите внимание на условие `argp == uintptr(p.argp)`.

Вновь вернёмся к [реализации паники](), чтобы понять логический смысл `p.arg`, `p.argp` и `p.recovered`:

```
// runtime/runtime2.go
package runtime

// A _panic holds information about an active panic.
// ...
type _panic struct {
    argp      unsafe.Pointer // pointer to arguments of deferred call
run during panic
    arg       interface{}    // argument to panic
    recovered bool           // whether this panic is over
    // ...

}

// runtime/runtime.go
package runtime

// The implementation of the predeclared function panic.
func gopanic(e interface{}) {
    // ...
    var p _panic

    // Прикопали аргумент panic(), чтобы вернуть его в recover().
    p.arg = e
    // Добавили панику в список паник.
    p.link = gp._panic
    // Обновили указатель на список паник в горутине.
    gp._panic = (*_panic)(noescape(unsafe.Pointer(&p)))

    // ...
    // Бежим по списку defer'ов.
    for {
```

```
        d := gp._defer
        if d == nil {
            break
        }

        // ...
        // Начали ссылаться на текущий FP.
        // Важно, что при вызове отложенной фунции ниже, этот
указатель изменится.
        p.argp = unsafe.Pointer(getargp())

        // ...
        // Вызвали отложенную функцию, в которой возможно был вызов
recover().
        reflectcall(
            nil,
            unsafe.Pointer(d.fn),
            deferArgs(d),
            uint32(d.siz),
            uint32(d.siz),
            uint32(d.siz),
            &regs)

        // ...
        pc := d.pc                      // <- !!! Запоминаем "место
создания" обрабатываемого
        sp := unsafe.Pointer(d.sp)    //         в данный момент defer.

        // ...
        d.fn = nil
        gp._defer = d.link // Переходим к следующему defer.
        freedefer(d)

        // Если вызов recover() (и как следствие, gorecover())
прошёл.
        if p.recovered {
            // ...

            // То перекидываем горутину на следующую панику (мы
коснёмся этого позже).
            gp._panic = p.link


            // И хитро прыгаем к следующим defer'ам.
```

```
            // ...
            // Pass information about recovering frame to recovery.
            gp.sigcode0 = uintptr(sp)
            gp.sigcode1 = pc          // <- !!! Сохранили в горутине
полученные выше значения.

            mcall(recovery)
            throw("recovery failed") // mcall should not return
        }
    }

    // ...
}

// getargp returns the location where the caller
// writes outgoing function call arguments.
// ...
func getargp() uintptr {
    return getcallersp() + sys.MinFrameSize
}
```

Таким образом, "восстановление" паники сводится к вызову `mcall(recovery)`, при этом в горутине сохраняется **PC** (program counter) и **SP** (stack pointer) на момент завершения обработки `defer`, с помощью которого мы отложили функцию, в которой был вызов `recover()`.

Что происходит? 🤯

---

mcall позволяет горутине вызвать код по работе с собой же ценой того, что она будет *перепланирована*:

```
package runtime

// mcall switches from the g to the g0 stack and invokes fn(g),
// where g is the goroutine that made the call.
// mcall saves g's current PC/SP in g->sched so that it can be
restored later.
// ...
// mcall returns to the original goroutine g later, when g has been
rescheduled.
```

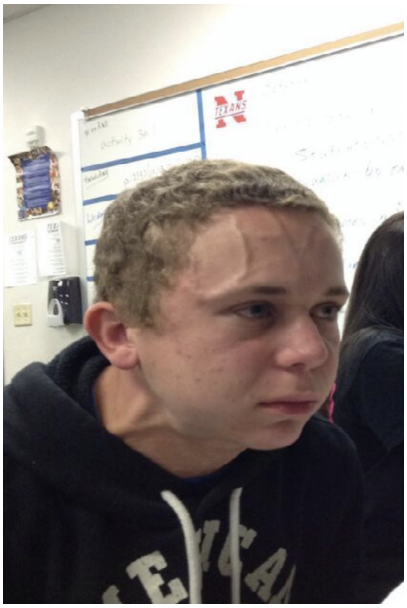
```
// fn must not return at all; typically it ends by calling schedule,
to let the m
// run other goroutines.
// ...

func mcall(fn func(*g))
```

(если есть ~~извращенцы~~ желающие посмотреть на реализацию `mcall` – вам [сюда](#)).

То есть, при восстановлении паники у нас происходит вызов runtime.recovery от текущей горутины, приостановка горутины и последующее её исполнение через *некоторое время*:

```
package runtime

// Unwind the stack after a deferred function calls recover
// after a panic. Then arrange to continue running as though
// the caller of the deferred function returned normally.
func recovery(gp *g) {
    // Info about defer passed in G struct.
    sp := gp.sigcode0
    pc := gp.sigcode1

    // ...

    // Make the deferproc for this d return again,
    // this time returning 1. The calling function will
    // jump to the standard return epilogue.
    gp.sched.sp = sp
    gp.sched.pc = pc
    gp.sched.lr = 0
    gp.sched.ret = 1
    gogo(&gp.sched)

}
```

И что мы видим? `recovery` "перемещает" поток выполнения горутины в место возврата `runtime.deferproc` (вы же ещё не забыли, что это?), и искусственно выставляет возвращаемое этой функцией значение в **единицу**!



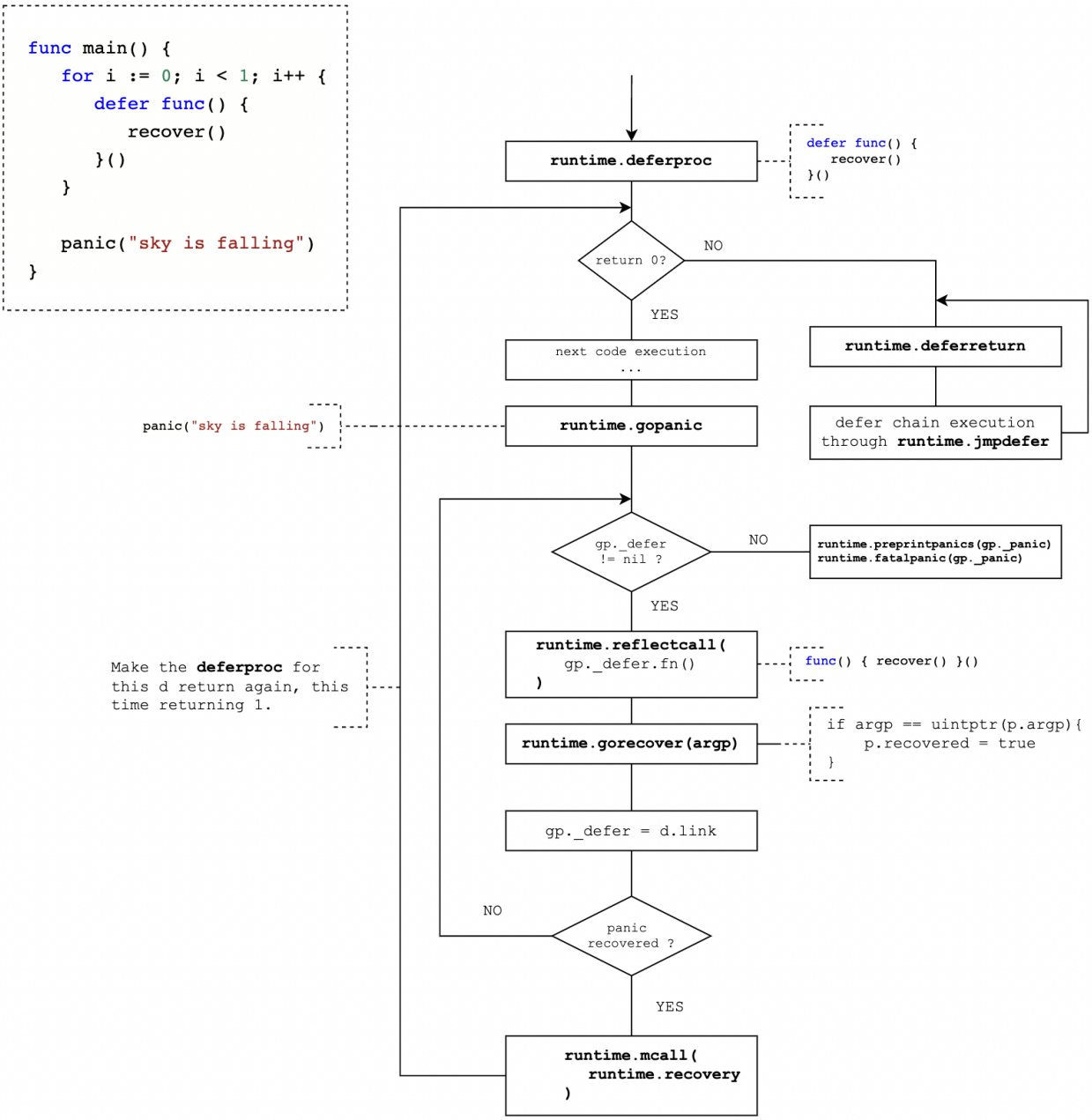
Внимательный читатель вспомнил, что мы уже обсуждали код возврата `runtime.deferproc` и именно в этом и зарыта собака:

```
# 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".
        # ...

withDefers_pc204:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)     # Код, выполняющий defer
chain.
        MOVQ    24(SP), BP
        ADDQ    $32, SP

        RET
```

Таким образом, через хаки рантайма происходит необходимое для обработки всех `defer` зацикливание при условии, что очередной `defer` восстанавливает

случившуюся панику (если паника не прерывается, то обработка всех `defer` сводится к простому циклу внутри `runtime.gopanic`).

Всё вышесказанное можно отобразить в виде следующей схемы:



При этом мы опустили немалую часть кода, отвечающего за обработку `defer` различного типа, паники во время паники, паники во время `defer` и т.д.

К слову, вопрос к аудитории – а через какие функции рантайма происходит работа с встроенными (open) `defer`? Ведь, как мы помним, `runtime.deferproc` в таких случаях пропадает 🙂.

Какие практические следствия можно вынести из данного разбора?

0) Компилятор Go всё-таки зелен в сравнении с GCC, LLVM и пр. и некоторые вещи сделаны в нём неэлегантно и неочевидно. Особенно от этого страдают разработчики, оптимизирующие компилятор под свои нужды.

1) Даже если паника была прервана, **функция**, в которой она произошла, **не продолжает свою работу**.

2) Независимо от того, была ли паника прервана или нет – **все отложенные функции будут выполнены**.

3) При неправильной работе с `recover()` **прервать панику не получится** (паника может быть вне действия `recover`). Подробнее об этом далее.

P.P.S. Если к концу этого шага у вас не взорвался мозг, то пишите, что думаете обо всё этом в комментарии 🥲

Тест "defer и recover"

Как обычно, постарайтесь, не запуская кода, понять, что выведет программа и завершится ли она успешно?

```
func main() {
    defer fmt.Println("before recover")
    {
        defer func() { recover() }()
    }
    defer fmt.Println("after recover")
```

```
    defer fmt.Println("before panic")
    {
        panic("sky is falling")
    }
    defer fmt.Println("after panic")

}
```

Не забывайте про порядок выполнения `defer`'ов!

Выберите один вариант из списка

- [ ] Бахнет!
  before panic
  after recover
  before recover
  sky is falling

- [ ] Завершится успешно.
  before panic
  after recover
  before recover

- [ ] Бахнет!
  before recover
  after recover
  before panic
  sky is falling

- [ ] Завершится успешно.
  before recover
  after recover
  before panic

- [ ] Завершится успешно.

after panic
before panic
after recover
before recover

- [ ] Завершится успешно.
  before panic
  after recover

Тест "Паника после паники"

```
func main() {
    process()
    fmt.Println("OK")
}

func process() {
    defer func() {
        recover()
    }()

    panic(1)
    panic(2)

}
```

Выберите один вариант из списка

- [ ] Бахнет!
- [ ] Не бахнет, зуб даю.
- [ ] Невозможно определить

Нерабочий recover или правило сопоставления panic с recover

На предыдущем шаге мы просили обратить внимание на следующее:

1) В `runtime.gopanic` при паниковании, перед вызовом отложенной функции, мы получаем указатель на текущий фрейм:

```
p.argp = unsafe.Pointer(getargp())

// Ниже p.argp изменится из-за особенностей реализации
runtime.reflectcall.
reflectcall(
    nil,
    unsafe.Pointer(d.fn),
    deferArgs(d),
    uint32(d.siz),
    uint32(d.siz),
    uint32(d.siz),
    &regs,

)
```

2) При этом `runtime.gorecover` принимает в качестве аргумента тоже фреймовый указатель:

```
LEAQ    +24(FP), AX

CALL    runtime.gorecover(SB)
```

3) В момент вызова `runtime.gorecover` сравнивает **FP** из аргумента и **FP** в момент паники, и только если они совпадают, то помечает панику как `recovered` (что отражено в комментарии внутри функции):

```
package runtime

func gorecover(argp uintptr) interface{} {
    // Must be in a function running as part of a deferred call
during the panic.
    // Must be called from the topmost function of the call
    // (the function used in the defer statement).
    // p.argp is the argument pointer of that topmost deferred
function call.
    // Compare against argp reported by caller.
    // If they match, the caller is the one who can recover.
    gp := getg()
```

```
    p := gp._panic
    if /* ... */ && argp == uintptr(p.argp) {
        p.recovered = true
        return p.arg
    }
    return nil

}
```

Другими словами, `runtime.gorecover` проверяет *стековое расстояние* между собой и паникой и останавливает панику, только если `recover` находился на вершине стека отложенной функции 🤯.

В исходном коде рантайма это выглядит просто, и звучит логично, но на деле (на уровне конечного машиноспецифичного ассемблерного кода) это работает крайне неочевидно.

Так что пока посмотрим на практическую сторону вопроса.

- Начнём с одной популярной ошибки, связанной с `recover`:

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

func main() {
    defer recover()                              // <- Не сработает,
recover() слишком "близко".

    v := calculate(1, 0)
    fmt.Println(v)
}

func calculate(alpha, n int) int {
    if n == 0 {
        panic(`"n" cannot be 0 due to algo`)
    }
    return alpha / n
}

/*
panic: "n" cannot be 0 due to algo
```

```
goroutine 1 [running]:
main.calculate(...)
      /tmp/sandbox817290803/prog.go:14
main.main()
      /tmp/sandbox817290803/prog.go:8 +0x65

*/
```

Чинится просто:

```
// https://goplay.tools/snippet/1AdmuXzH1SM

func main() {
    defer func() { recover() }()

// ...
```

- Следующая ошибка выглядит так:

```
// https://goplay.tools/snippet/kGQjGM8dsLo

func main() {
    say("hello!", 3)
}

func say(s string, n int) {
    defer func() {
        logPanic()                              // <- Не сработает,
recover() слишком "далеко".
        cleanUp()
    }()
    for i := 0; i < n; i++ {
        if i == 1 {
            panic("keep silence")
        }
        fmt.Println(s)
    }
}

func logPanic() {
```

```
    if err := recover(); err != nil {
        fmt.Println("got panic:", err)
    }
}

func cleanUp() {
    /* ... */
}

/*
hello!
panic: keep silence

goroutine 1 [running]:
main.say({0x494ac5, 0x6}, 0x3)
     /tmp/sandbox58563336/prog.go:20 +0xf4
main.main()
     /tmp/sandbox58563336/prog.go:9 +0x2a

*/
```

Как чинить:

```
// https://goplay.tools/snippet/tPXIIs-LZ55

// Вариант 1: избавляемся от лишнего фрейма путём избавления от
функции logPanic():
func say(s string, n int) {
    defer func() {
        if err := recover(); err != nil {
            fmt.Println("got panic:", err)
        }

        cleanUp()
    }()

// ...

// https://goplay.tools/snippet/ZDadOEB2a5j

// Вариант 2: избавляемся от лишнего фрейма через defer без
промежуточной анонимной функции:
func say(s string, n int) {
    defer cleanUp()
    defer logPanic()
```

```
// ..
```

- Ситуация: отложенная функция тоже может паникнуть и разработчик одним `recover` хочет убить двух зайцев:

```
// https://goplay.tools/snippet/CK3GEigK_zF

func main() {
    say("hello!", 3)
}

func say(s string, n int) {
    defer func() {
        defer func() {
            if err := recover(); err != nil {
                fmt.Println("got panic:", err)
            }
        }()

        cleanUp()
    }()
    for i := 0; i < n; i++ {
        if i == 1 {
            panic("keep silence")
        }
        fmt.Println(s)
    }
}

func cleanUp() {
    panic("while cleanup")
}

/*
hello!
got panic: while cleanup
panic: keep silence
```

```
goroutine 1 [running]:
main.say({0x494ac5, 0x6}, 0x3)
	/tmp/sandbox1988511657/prog.go:23 +0xda
main.main()
	/tmp/sandbox1988511657/prog.go:8 +0x2a

*/
```

`recover()` отловил панику внутри первого (внешнего) `defer`, но никак не повлиял на панику за его пределами.

Итого, для верной работы `recover` необходимо, чтобы выполнялись следующие условия:

```
// Must be in a function running as part of a deferred call during the panic.

// Must be called from the topmost function of the call (the function used in the defer statement).
```

1. Вызов `recover()` должен быть внутри отложенной функции (и, как следствие, не может быть отложен сам).

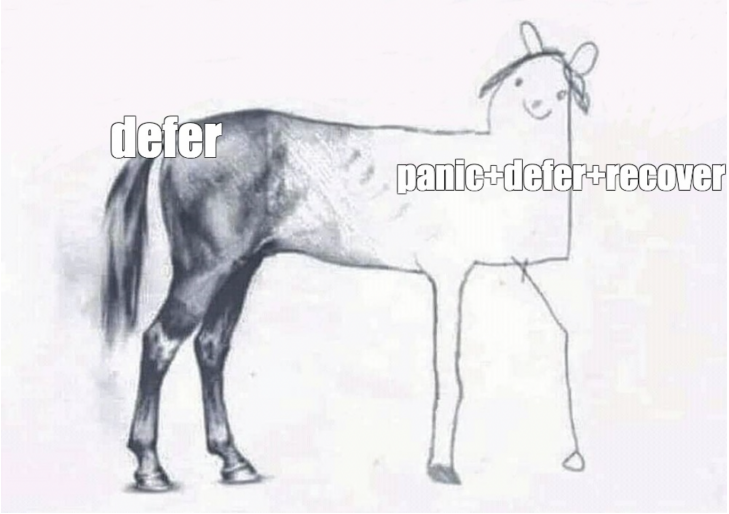
2. При этом вызов должен находиться в "самом верху" откладываемой функции, т.е. если вы отложили функцию, внутри вызвали пару других функций, углубившись по стеку вызовов, а затем уже там вызвали `recover`, то она **не отловит вышестоящие паники**; аналогично для `recover()` в `defer`, вложенном в другой `defer` и т.п.

3. Можно сказать, что `recover()` действует на `panic()` в пределах фрейма, в котором находится объявление `defer`.

Все эти неочевидные моменты изредка выстреливают у грамотных разработчиков и часто у людей, которые решают программировать через **panic-recover** (привыкнув к исключениям в других языках) вместо использования православного проброса ошибок (**error propagation & handling**).

Вопрос, почему механизм `recover()` выглядит именно таким образом, мы обсудим в следующем уроке.

Тест "Не бахает! "



Выберите сниппеты, код в которых не бахает 🙂.

1)

```
func main() {
    defer recover()
    foo()
    fmt.Println("I'm OK")
}

func foo() {
    panic("sky is falling")

}
```

2)

```
func main() {
    var a[1]int
```

```
    fmt.Println("I'm OK", a[0])

}
```

3)

```
func main() {
    defer func() {
        recover()
    }()

    *(*int)(nil) = 0
    fmt.Println("I'm OK")

}
```

4)

```
func main() {
    defer func() {
        func() {
            recover()
        }()
    }()
    panic("sky is falling")
}
```

5)

```
func main() {
    defer func() {
        recover()
    }()

    *(*int)(nil) = 0
    panic("sky is falling")
}

func recover() {

}
```

6)

```
func main() {
    defer recoverable()
    panic("sky is falling")
}

func recoverable() {
    defer func() {
        recover()
    }()

}
```

7)

```
var r = func() { recover() }

func main() {
    defer r()
    panic("sky is falling")

}
```

8)

```
func main() {
    defer func() {
        cleanUp()
    }()
    fmt.Println("I'm OK")
}

func cleanUp() {
    defer func() {
        recover()
    }()

    panic("sky is falling - 2")
}
```

9)

```
func main() {
    defer recoverable()
```

```
    panic("sky is falling")
}

func recoverable() {
    recover()

}
```

10)

```
func main() {
    panic("sky is falling")
    recover()

}
```

11)

```
func main() {
    a := []int{99, 1, 42}

    var total int
    for i := 0; i <= len(a); i++ {
        total += a[i]
    }
    fmt.Println(total)
}

defer func() {
    // Global recovering.
    recover()
}()
```

12)

```
func init() {
    defer func() {
        recover()
    }()
}

func init() {
    panic("sky is falling")
```

```
}

func main() {
    // Go!

}
```

13)

```
func init() {
    panic("sky is falling")
}

func init() {
    defer func() {
        recover()
    }()
}

func main() {
    // Go!

}
```

Тест "recover() много не бывает"

Постарайтесь, не запуская кода, понять, что выведет программа (без учёта стектрейса)?

```
func main() {
   defer func() { fmt.Println(recover()) }()
   defer func() { fmt.Println(recover()) }()
   defer func() { fmt.Println(recover()) }()
   work()
}

func work() {
   panic("holiday")

}
```

Выберите один вариант из списка

- [ ] holiday
  <nil>
  <nil>
  panic: holiday

- [ ] panic: call recover() without panic

- [ ] holiday
  <nil>
  <nil>

- [ ] <nil>
  <nil>
  <nil>

Задача "InterfaceEqual"

Ссылка на заготовку.

Вам необходимо реализовать функцию, которая говорит, являются ли два интерфейса равными:

```
func InterfaceEqual(lhs, rhs any) bool
```

Но тут есть подводный камень, ведь согласно спецификации:

Interface values are comparable. Two interface values are equal if they have identical dynamic types and equal dynamic values or if both have value `nil`.

A comparison of two interface values with identical dynamic types ***causes a run-time panic*** *if values of that type are not comparable.*

Так как мы привыкли писать надёжный и отказоустойчивый код, необходимо сделать так, чтобы `InterfaceEqual` не ронял программу при ситуации, описанной выше, а просто возвращал `false`.

Задача "LogError"

Ссылка на заготовку.

Вам необходимо реализовать функцию `LogError`:

```
// LogError записывает в out сообщение об ошибке err.

func LogError(err error, out io.Writer)
```

Функция с помощью писателя `out` пишет сообщение в следующем формате:

```
error occurred: %err msg%
```

Где `%err msg%` равно:

- `<nil>` – если ошибка нулевая;
- `(<тип ошибки>).Error() panic: <сообщение от паники>` – если при получении строкового представления ошибки произошла паника;
- строковое представление ошибки (`err.Error()`) – в обратном случае (т.е. при нормальном позитивном сценарии).

Больше подробностей см. в заготовке задачи и тестах.

Задача "Transact Helper"

Ссылка на заготовку.

Продолжая тему работы с транзакциями, мы предлагаем написать вам вспомогательную функцию `Transact`:

```
type DB interface {
    Begin() (Tx, error)
}

type Tx interface {
    Exec(q string) error
    Commit() error
    Rollback() error
}
```

```
// Transact запускает функцию f в созданной для неё с помощью db транзакции.
// При успешном завершении функции транзакция фиксируется, иначе – откатывается.

func Transact(db DB, f func(tx Tx) error) error
```

На что стоит обратить внимание:

- Начало новой транзакции может завершиться неуспешно.
- Если `fn` вернула ошибку, то транзакцию следует откатить, а ошибку вернуть "наверх".
- Если `fn` запаниковала, то транзакцию следует откатить, а программа должна **продолжить паниковать**.
- Если `fn` завершилась успешно, то транзакцию следует зафиксировать.
- Если фиксирование транзакции завершилось ошибкой, то мы должны вернуть эту ошибку "наверх".

Допущения:

- Ради упрощения кода, считаем, что ошибка от отката транзакции нас не интересует и её можно игнорировать.
- Методы интерфейсов `Tx` и `DB` не могут паниковать.
- Считаем, что принятие решения по поводу паники – вне зоны ответственности нашей функции. Поэтому мы не должны мешать паникованию программы, но со своей стороны должны избежать утечек ресурсов и коммита частично выполненной транзакции.

Больше подробностей см. в заготовке задачи и тестах.

Задача "Panic Stack"

Ссылка на заготовку.

Вам необходимо реализовать простую функцию, которая ловит панику и выводит стектрейс от места её возникновения:

```
type Logger interface {
    Logf(format string, args ...any)
}


func LogPanicWithTrace(l Logger)
```

Пример использования функции:

```
func TestLogPanicWithTrace_Example(t *testing.T) {
    defer LogPanicWithTrace(t)
    panic("sky is falling")
}

/*
sky is falling
goroutine 35 [running]:
runtime/debug.Stack()
    /usr/local/go/src/runtime/debug/stack.go:24 +0x94
github.com/www-golang-courses-ru/advanced-dealing-with-panic-in-go/tasks/03-panic-concept/panic-stack.LogPanicWithTrace({0x1042f5240, 0xc000107a00})

/Users/anthony/advanced-dealing-with-panic-in-go/tasks/03-panic-concept/panic-stack/log_panic_with_trace.go:13 +0x54
panic({0x1042bbc80, 0x1042f42f0})
    /usr/local/go/src/runtime/panic.go:1052 +0x2ac
github.com/www-golang-courses-ru/advanced-dealing-with-panic-in-go/tasks/03-panic-concept/panic-stack.TestLogPanicWithTrace_Example(0xc000107a00)

/Users/anthony/advanced-dealing-with-panic-in-go/tasks/03-panic-concept/panic-stack/log_panic_with_trace_test.go:16 +0x80
testing.tRunner(0xc000107a00, 0x1042f3850)
    /usr/local/go/src/testing/testing.go:1259 +0x19c
created by testing.(*T).Run
    /usr/local/go/src/testing/testing.go:1306 +0x5bc
```

```
*/
```

Формат логируемого сообщения: `<строковое представление значения паники><new line><стектрейс>`.

- Строковое представление значения, которым паниковали, эквивалентно его форматированию через спецификатор `%v` (в "реальной" жизни там вероятнее всего будет или строка или ошибка, с различным дальнейшим флоу в зависимости от её типа).

- Для получения стектрейса вам понадобится один из методов пакета runtime/debug.

Больше подробностей см. в заготовке задачи и тестах.

Задача "Recover V2"

Ссылка на заготовку.

Мы помним, что механизм восстановления паники в Go имеет ещё одну "особенность". Если разработчик по ошибке или специально (😱) паникует `nil`'ом, то через `recover()` невозможно будет определить, была ли паника на самом деле:

```
func foo() {
    defer func() {
        // Паники не было в принципе?
        // Или кто-то вызвал panic от nil?
        if err := recover(); err == nil {
            // ...
        }
    }()
    // ...

}
```

Вам необходимо усовершенствовать функцию `Recover` из предыдущей задачи таким образом, чтобы она кроме значения паники возвращала ещё флаг, а произошла ли паника в принципе?

```
// Recover вызывает fn, ловит панику и возвращает её аргумент и true.
// Если паники не было, то функция вторым аргументом вернёт false.

func Recover(fn func()) (any, bool)
```

Больше подробностей см. в изменившихся тестах.

Промежуточные выводы

Мы узнали, что не так страшна паника, как её малюют, и у нас есть механизм прерывания (восстановления, остановки) `panic` в виде встроенной функции `recover`.

Чтобы не наступать на различные грабли, необходимо следовать следующим правилам:

1) Использовать следующий шаблон для восстановления паникущей горутины:

```
// Вызов recover должен находиться в defer, обязательно до паникущего кода.
// Вызов recover должен быть на самом верхнем уровне в defer, непосредственно в отложенной функции.
// Нельзя откладывать непосредственно recover(), вызов должен быть обёрнут в отложенную функцию.
func foo() {
    defer func() {
        if err := recover(); err != nil {
            // ...
        }
    }()

}
```

2) Быть бдительным при любом шаге в сторону от шаблона выше.

3) Ни в коем случае не паниковать `nil`'ом:

```
func foo() {
    panic(nil) // ALARM!

}
```

В следующих уроках мы обсудим, что происходит при панике во время паники, а также продолжим тему неуловимых паник.

Вперёд!