Понятие паники: panic

Пришло время поговорить о том, ради чего мы все здесь собрались – о панике!

Исключительная ситуация

Согласно Wiki:

Во время выполнения программы могут возникать ситуации, когда состояние ... компьютерной системы в целом делает дальнейшие вычисления в соответствии с базовым алгоритмом невозможными или бессмысленными.

Т.е. суть **исключительной ситуации (исключения)** заключается в её названии – это нечто из ряда вон выходящее, ситуация, когда программа из-за влияния внешних сил или по вине ошибки разработчика уходит с пути своего *нормального* выполнения.

Запустим простую программу на Python (исходники примеров):

```
# -*- coding: utf-8 -*-
```

```
def div(a, b):
    return a / b


print("div(42, 0) = %d" % div(42, 0))
print("OK")  # Не напечатается!
```

```
$ python example.py
Traceback (most recent call last):
  File "example.py", line 7, in <module>
    print("div(42, 0) = %d" % div(42, 0))
  File "example.py", line 4, in div
    return a / b
ZeroDivisionError: integer division or modulo by zero

$ echo $?
1
```

Мы получили исключение ZeroDivisionError, а процесс прекратил свою работу, завершившись с ненулевым кодом ответа.

Перепишем пример выше на Go:

```
package main

import "fmt"

func div(a, b int) int {
    return a / b
}

func main() {
    fmt.Printf("div(42, 0) = %v\n", div(42, 0))
    fmt.Println("OK") // Не напечатается!
}
```

```
$ go run example.go
panic: runtime error: integer divide by zero

goroutine 1 [running]:
main.div(...)

advanced-dealing-with-panic-in-go/examples/03-panic-concept/exception
/example.go:6
main.main()

advanced-dealing-with-panic-in-go/examples/03-panic-concept/exception
/example.go:10 +0x24
exit status 2

$ echo $?
1
```

Как и ожидалось, наша программа **запаниковала**!

Паникование

Ошибки во время выполнения программы на Go, такие как выход за границы массива, разыменование `nil`-указателя и пр. несуразности вызывают ***паникование (run-time panicking)*** программы и эквивалентны вызову встроенной (built-in) функции [panic](#) от аргумента, реализующего интерфейс [runtime.Error](#):

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

func panic(v interface{})
```

```
// runtime/error.go
package runtime

// The Error interface identifies a run time error.
type Error interface {
    error

    // RuntimeError is a no-op function but
    // serves to distinguish types that are run time
```

```
    // errors from ordinary errors: a type is a
    // run time error if it has a RuntimeError method.
    RuntimeError()

}
```

Во время выполнения функции F при явном вызове `panic` или при run-time панике:

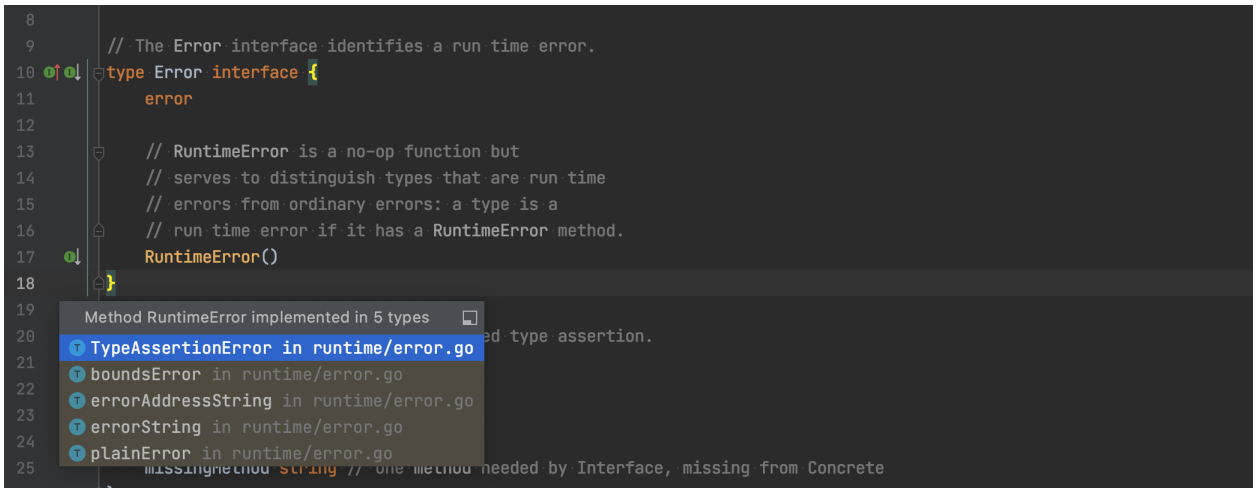
- функция прекращает свою работу;
- затем выполнение переходит функциям, отложенным F;
- затем выполнение переходит функциям, отложенным функцией, вызывающей F (её родителем);
- и так далее вверх по стеку вызовов, пока не будут выполнены функции, отложенные выполняющейся горутиной (напомним, что функция `main` является корневой горутиной в программах на Go) ;
- затем программа завершится с ненулевым кодом возврата и мы увидим стектрейс паники и собственно саму ошибку.

Этот процесс завершения программы с последовательным вызовом своих и вышележащих отложенных функций и называется ***паникованием***:

```
// https://goplay.tools/snippet/ilVe9s_oQ_W
package main

func main() {
    var a []int
    _ = a[1]
}

/*
panic: runtime error: index out of range [1] with length 0

goroutine 1 [running]:
main.main()
     /tmp/sandbox1240453622/prog.go:5 +0x1a

Process finished with exit code 2
```

```
*/
```

Давайте посмотрим, какие run-time паники мы можем встретить, программируя на Go:



Все они определены в [runtime/error.go](), и по названиям можно догадаться, за что данные ошибки отвечают:

```
// runtime/*

panic(&TypeAssertionError{concrete: typ, asserted: &inter.typ,
missingMethod: m.init()})
panic(errorString("makeslice: len out of range"))
panic(errorAddressString{msg: "invalid memory address or nil pointer
dereference", addr: addr})


// и т.д.
```

Встроенная функция panic

Ранее мы упомянули тот факт, что паника может произойти не только благодаря рантайму языка Go, но и благодаря нашему явному вызову функции [panic]():

```
package builtin
```

```
// ...

// The panic built-in function stops normal execution of the current
// goroutine. When a function F calls panic, normal execution of F
stops
// immediately. Any functions whose execution was deferred by F are
run in
// the usual way, and then F returns to its caller. To the caller G,
the
// invocation of F then behaves like a call to panic, terminating G's
// execution and running any deferred functions. This continues until
all
// functions in the executing goroutine have stopped, in reverse
order. At
// that point, the program is terminated with a non-zero exit code.
This
// termination sequence is called panicking and can be controlled by
the
// built-in function recover.
func panic(v interface{})
```

Реализация встроенных функций скрыта и лежит на плечах рантайма. Но как видно по аргументу `v interface {}`, паниковать можно чем угодно (необязательно значением, удовлетворяющим `error` или `runtime.Error`) и Go оставляет это на вашу совесть.

Например, паникуем числом:

```
// https://goplay.tools/snippet/PxG8FKZ89RW
package main

func main() {
    panic(42)
}

/*
panic: 42

goroutine 1 [running]:
main.main()
    /tmp/sandbox893914853/prog.go:4 +0x39

*/
```

Паникуем строкой:

```
package main

func main() {
    panic("42")
}

/*
panic: 42

goroutine 1 [running]:
main.main()
    /tmp/sandbox893914853/prog.go:4 +0x39

*/
```

Паникуем структурой:

```
package main

func main() {
    panic(struct{ UsedID int }{100})
}

/*
panic: (struct { UsedID int }) 0x4c8d60

goroutine 1 [running]:
main.main()
    /tmp/sandbox893914853/prog.go:4 +0x39

*/
```

и т.д.

Тест "Что произойдёт при запуске программы?"

```
package main

import "fmt"

func main() {
    doWork()
```

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

func doWork() {
    panic(nil)               // <- !!!
}
```

Выберите один вариант из списка

- [ ] Программа не скомпилируется, паниковать "ничем" нельзя
- [ ] Программа скомпилируется, но паника не произойдёт, так как компилятор "вырезает" паники с nil-аргументом
- [ ] Произойдёт паника, мы увидим "panic: nil" и стектрейс текущей горутины, затем на экране напечатается "OK" и процесс завершится
- [ ] Произойдёт паника, мы увидим "panic: nil" и стектрейс текущей горутины, затем процесс завершится

panic под капотом

Привычным нам способом посмотрим на ассемблер простой, но паникующей программы:

```
package main

func main() {
    a := 42
    panic(a)
}
```

```
# main
main_pc0:
        TEXT    "".main(SB), ABIInternal, $24-0
        CMPQ    SP, 16(R14)
        PCDATA  $0, $-2
        JLS     main_pc46
        PCDATA  $0, $-1
        SUBQ    $24, SP
        MOVQ    BP, 16(SP)
```

```
        LEAQ    16(SP), BP
        FUNCDATA        $0,
gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        FUNCDATA        $1,
gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        MOVL    $42, AX
        PCDATA  $1, $0
        CALL    runtime.convT64(SB)
        MOVQ    AX, BX
        LEAQ    type.int(SB), AX
        CALL    runtime.gopanic(SB)
        XCHGL   AX, AX
main_pc46:
        NOP
        PCDATA  $1, $-1
        PCDATA  $0, $-2
        CALL    runtime.morestack_noctxt(SB)
        PCDATA  $0, $-1

        JMP     main_pc0
```

Нас интересует, во что превращается вызов `panic`:

```
       MOVL    $42, AX
        CALL    runtime.convT64(SB)   # Магия interface{}
        MOVQ    AX, BX
        LEAQ    type.int(SB), AX
        CALL    runtime.gopanic(SB)   # panic(a)
```

А превращается он в вызов runtime.gopanic (листинг упрощён):

```
package runtime

// The implementation of the predeclared function panic.
func gopanic(e interface{}) {
    gp := getg()

    // ...
    var p _panic
    p.arg = e
    p.link = gp._panic
    gp._panic = (*_panic)(noescape(unsafe.Pointer(&p)))
```

```
    atomic.Xadd(&runningPanicDefers, 1)

    // ...
    for {
        d := gp._defer
        if d == nil {
            break
        }

        // ...
        d.started = true
        d._panic = (*_panic)(noescape(unsafe.Pointer(&p)))


        // ...
        reflectcall(nil,
            unsafe.Pointer(d.fn),
            deferArgs(d),
            uint32(d.siz),
            uint32(d.siz),
            uint32(d.siz),
            &regs)

        p.argp = nil
        d._panic = nil

        // ...
        pc := d.pc
        sp := unsafe.Pointer(d.sp)

        d.fn = nil
        gp._defer = d.link
        freedefer(d)

        if p.recovered {
            // ...
        }
    }

    preprintpanics(gp._panic)

    fatalpanic(gp._panic) // should not return
    *(*int)(nil) = 0      // not reached

}
```

Таким образом, горутинка ссылается не только на список `defer` (точнее – на его голову), но и на список паник:

```
package runtime

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

// 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
    link      *_panic        // link to earlier panic
    pc        uintptr        // where to return to in runtime if this
panic is bypassed
    sp        unsafe.Pointer // where to return to in runtime if this
panic is bypassed
    recovered bool           // whether this panic is over
    aborted   bool           // the panic was aborted
    goexit    bool

}
```

Мы сильно упростили `runtime.gopanic` в листинге, но основная идея понятна. Функцию можно переписать в виде псевдокода:

```
func gopanic(e interface{}) {
    gp := getg()      // Получи текущую горутинку.

    var p _panic
    p.arg = e         // Схоронили аргумент panic() в мета-информацию
о панике.
    gp._panic = &p    // Схоронили указатель на текущую панику в
текущей горутине.

    d := gp._defer    // Пока есть defer'ы.
    for d != nil {
```

```
            d()           // Выполняем defer.
            if p.recovered {
                // Логика, если паника была восстановлена во время
выполнения defer строчкой выше.
                // Что это значит - узнаем в следующем уроке.
            }
            d = d.link    // Переходим к предыдущему defer.
        }

        fatalpanic(gp._panic) // Печатаем стек и завершаем процесс.
        *(*int)(nil) = 0      // Вставляем assert, чтобы себя проверить.

}
```

Ничего не напоминает?

Мы помним, что `defer`'ы складываются в текущей горутине. Независимо от количества вложенных вызовов различных функций, `defer`'ы каждой из них будут находится в едином списке, указатель на голову которого хранится в структуре горутины.

Функция `panic` работает с этим списком и ничего не знает о `defer` за пределами паникующей горутины!

Запомним данный факт, он нам пригодится в дальнейшем.

Тест "runtime.fatalpanic"

Какой сигнал рантайм Go *возможно* пошлёт своему процессу, чтобы завершить его при панике, если программа исполняется в ОС из семейства Unix?

Формат ответа соответствует константам и макросам из signal.h: `SIGINT`, `SIGHUP`, `SIG_IGN` и т.д

.

.

.

(подсказка, куда смотреть)

Напишите текст

_____

Тест "Чужие defer"

Постарайтесь, не запуская кода, выбрать, что выведет программа (без учёта стектрейса):

```
func main() {
    defer fmt.Println("main")

    go func() {
        defer fmt.Println("go 1")
        time.Sleep(time.Second) // Задержка нужна, чтобы горутины
были живы на момент выхода из main.
    }()

    go func() {
        defer fmt.Println("go 2")
        time.Sleep(time.Second)
    }()

    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

Верно решили **10** учащихся

Из всех попыток **58%** верных

- ☐ before panic

main
go 2
go 1
panic: sky is falling

- [ ] before panic
  main
  go 1
  go 2
  panic: sky is falling

- [ ] before panic
  main
  panic: sky is falling

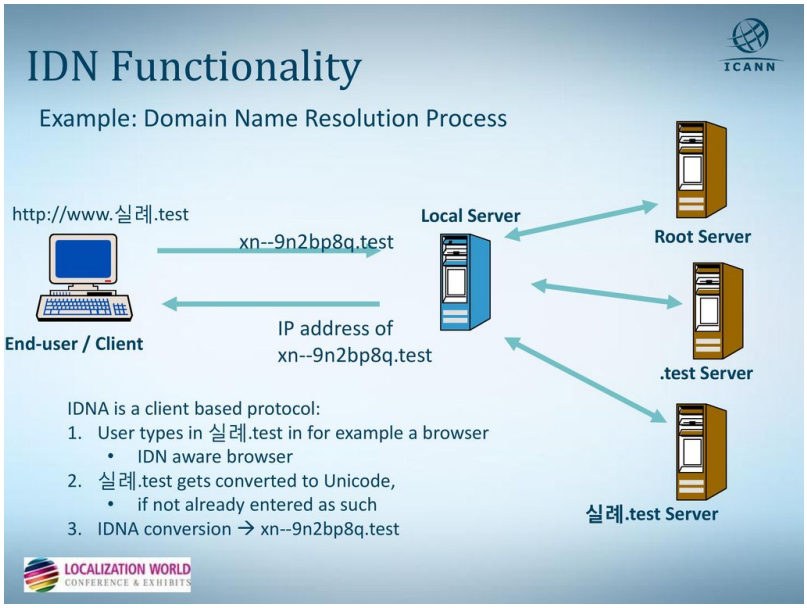
- [ ] main
  before panic
  panic: sky is falling

- [ ] after panic
  before panic
  main
  panic: sky is falling

Невозможно определить, так как порядок запуска горутин недетерменирован

Задача "IDNA Protocol"

Ссылка на заготовку.



Протокол **IDNA** (Internationalized Domain Names in Applications) позволяет поддерживать доменные имена, содержащие нелатинские символы (чаще всего символы национальных алфавитов) путём преобразования их на стороне клиента в ASCII-совместимую (**ACE** – ASCII Compatible Encoding) последовательность.

Перед выполнением задания рекомендуем ознакомиться со следующими ссылками, чтобы лучше "въехать" в тему:

- Wiki: IDN
- Wiki: Punycode
- Punicode-конвертер

Процесс выполнения задания состоит из двух частей.

1) Вам необходимо реализовать функцию `encodeDigit`, которая используется в уже готовой функции `punyEncode`:

Функция `punyEncode` осуществляет преобразование входной строки в Punycode в соответствии с RFC 3492 и, если говорить простыми словами, работает следующим образом:

- для каждого не-ASCII символа во входной строке `punyEncode` генерирует набор чисел;
- каждое из этих чисел с помощью `encodeDigit` превращается в ASCII-символ;
- полученный символ добавляется в результирующую строку.

При этом алгоритм Punycode-кодирования устроен так, что генерируемые числа входят в диапазон `[0, 35]`, и любое другое число свидетельствует об ошибке в программировании алгоритма – в таком случае `encodeDigit` должна паниковать:

```
// encodeDigit возвращает символ по коду:
//  - 0..25 отображаются в ASCII a..z
//  - 26..35 отображаются в ASCII 0..9
// При любом другом значении digit функция паникует.

func encodeDigit(digit int32) byte
```

2) После того, как вы убедитесь, что тесты в **punycode_test.go** проходят, можно приступать к реализации функции `idna.ToACE`:

```
package idna

// ToACE превращает domain в ACE-последовательность.

func ToACE(domain string) (string, error)
```

Алгоритм её работы следующий:

- входящее доменное имя разбивается на части (уровни) по точке;
- каждая из частей кодируется в Punycode с помощью функции `punyEncode`; при этом, если кодирование действительно было (домен содержал не ASCII-символы), то к закодированной строке добавляется так называемый ACE prefix `"xn--"`;
- закодированные части "склеиваются" назад через точку.

Примеры:

```
1)
go-proverbs.github.io -> go-proverbs.github.io

- Каждый из доменов уже является ACE-последовательностью, ничего не изменилось

2)
lagom-är-bäst.com -> xn--lagom-r-bst-q8ad.com

 - "lagom-är-bäst" преобразовался в "xn--" + "lagom-r-bst-q8ad"
 - "com" остался без изменений

3)
котомастер.рф -> xn--80aknijargfe.xn--p1ai

 - "котомастер" преобразовался в "xn--" + "80aknijargfe"
 - "рф" преобразовался в "xn--" + "p1ai"
```

- Менять сигнатуру и содержание функции `punyEncode` нельзя.
- Изобретать космолётов в `ToACE` не нужно ("настоящая" реализация согласно RFC 5891 гораздо сложнее) – достаточно, чтобы тесты проходили.

Больше подробностей в заготовке задачи и тестах.

Промежуточные выводы

- Мы узнали, что в Go существует механизм исключительных ситуаций, называемый **паникой**.

- Паниковать может как сама гошка во время выполнения программы (если вы делаете что-то из ряда вон выходящее), так и вы самостоятельно с помощью вызова встроенной функции `panic`.
- Паниковать можно чем угодно, так как аргумент `panic` имеет тип `interface{}`.
- Паника выполняет все функции, которые были отложены (**deferred**) до момента её возникновения.
- Паника работает в рамках текущей горутины и **не выполняет** функции, отложенные в "чужих" горутинах.
- Любой код после вызова `panic` недостижим, включая оставшиеся `defer`'ы.

В следующих уроках мы научимся справляться с паникой и познакомимся с механизмом её "восстановления".