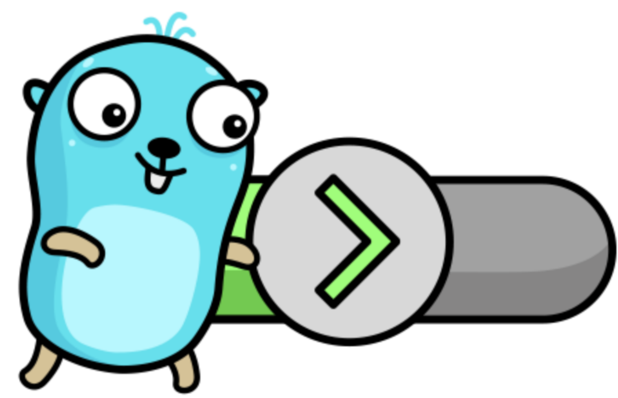
Неуловимые паники

В этом уроке мы лишний раз обратим внимание на **связь паники и текущей горутины**, а также посмотрим, какие бывают сбои программы, прервать которые **невозможно**.



Неуловимая паника

Городок в западно-американской степи. Салун. За столом сидят два ковбоя, местный и приезжий, и пьют виски. Вдруг по улице кто-то проносится на огромной скорости, паля во все стороны из пистолетов. В салуне никто и ухом не ведёт. Приезжий местному:

— Билл?

— Да, Гарри?

— Что это было, Билл?

— Это был Неуловимый Джо, Гарри.

— А почему его зовут Неуловимым Джо, Билл?

— Потому что его никто ещё не поймал, Гарри.

— А почему его никто ещё не поймал, Билл?

— Потому что он нафиг никому не нужен, Гарри.

Из предыдущих уроков мы знаем:

- список отложенных функций (**defer chain**) привязан к текущей горутине;
- `recover` должна быть вызвана в отложенной функции;
- `panic` (`runtime.gopanic`) и `recover` (`runtime.gorecover`) работают с `_panic`, взятой из текущей горутины.

```
package runtime

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

}
```

На основе вышеперечисленного мы можем сделать вывод, что **паника в другой горутине неуловима для** `recover()` **в текущей**:

```
// https://goplay.tools/snippet/PdvVT42B0-M
package main

// Бахнет. Паника в foo недостижима для recover() в main.
func main() {
    defer func() {
        recover()
    }()

    go foo()

    select {} // Simple blocking call.
}

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

/*
panic: sky is falling

goroutine 5 [running]:
main.foo()
     /tmp/sandbox638572740/prog.go:14 +0x39
created by main.main
```

```
    /tmp/sandbox638572740/prog.go:8 +0x57
[T+0000ms]

*/
```

Ладно бы она молчаливо где-то отрабатывала и никак не мешала основной программе, но паника, как подобает по-настоящему исключительной ситуации, **завершает процесс**, даже если она была вызвана не из горутины `main`.

Самое частое заблуждение у Golang-разработчиков, что паника в другой горутине не роняет программу, или что её можно остановить с помощью `recover()` в `main` – нет, нет и нет!

В отличие от Неуловимого Джо хотелось бы поймать любую панику, где бы она не случилась. В таком случае нам необходимо обложиться вызовом `recover()` в каждой запускаемой горутине:

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

import "fmt"

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

    recovered := make(chan struct{})
    go foo(recovered)

    <-recovered
    fmt.Println("OK")
}

func foo(recovered chan <-struct{}) {
    defer func() {
        defer close(recovered)
        recover()
    }()
    panic("sky is falling")
}
```

```
// OK
```

Тест "Что произойдёт?"

Продолжаем мучить вас сниппетами, которые можно встретить на собеседованиях:

```go
package main

import "fmt"

func main() {
    go panic("1")
    fmt.Println("I'm OK")

}
```

Выберите один вариант из списка

- ☐ Планировщик моментально переключится на новую горутину и программа запаникует, не успев ничего вывести на экран
- ☐ Горутина не успеет запуститься до выхода из программы, всё будет OK
- ☐ Поведение недетерменировано: возможны как оба варианта выше, так и то, что горутина успеет запуститься после печати на экран сообщения, но до выхода из программы, и мы получим
  I'm OK
  panic: 1

Тест "Global Recovery"

На просторах сети один автор предлагает следующий паттерн для "глобальной защиты от паники в Go":

```go
// cmd/my-cool-service/main.go
package main

import (
    "log"
    "os"
)
```

```go
func main() {
    defer recoverPanic()
    runApp()
}

func runApp() {
    panic("some error in application logic")
}

func recoverPanic() {
    if err := recover(); err != nil {
        log.Println("unexpected error:", err)
        os.Exit(1)
    }
}
```

Выберите, от чего подобный `recover()` действительно спасёт.

Выберите все подходящие ответы из списка

- ☐ От паник, произошедших дальше по стеку от вызова runApp()
- ☐ От паник, произошедших в отложенных функциях, дальше по стеку от вызова runApp()
- ☐ От паник, произошедших в горутинах, запущенных в функциях дальше по стеку от вызова runApp()

Задача "Recoverer"

[Ссылка на заготовку](#).

Вам необходимо реализовать тип `Recoverer`, который позволяет безопасно (защищая от паник) выполнять функции как синхронно, так и в отдельной горутине:

```
type RecoveryHandler func(err any)

type GoroutineStarter interface {
    Go(func())
}

// NewRecoverer создаёт новый Recoverer, в качестве аргументов
принимает
// обработчик паники по умолчанию и инструмент для запуска горутин.

func NewRecoverer(h RecoveryHandler, s GoroutineStarter) *Recoverer
```

Тип должен удовлетворять следующему интерфейсу:

```
type IRecoverer interface {
    // Do синхронно выполняет функцию f.
    // Возможная паника обрабатывается хендлером по умолчанию.
    Do(f func())

    // DoWithRecoveryHandler синхронно выполняет функцию f.
    // Возможная паника обрабатывается хендлером handler.
    DoWithRecoveryHandler(f func(), handler RecoveryHandler)

    // Go выполняет функцию f в новой горутине, созданной с помощью
GoroutineStarter.
    // Возможная паника обрабатывается хендлером по умолчанию.
```

```
    Go(f func())

    // GoWithRecoveryHandler выполняет функцию f в новой горутине,
созданной с помощью GoroutineStarter.
    // Возможная паника обрабатывается хендлером handler.
    GoWithRecoveryHandler(f func(), handler RecoveryHandler)

}
```

Больше подробностей см. в заготовке задачи и тестах.

Не забудьте прогнать своё решение через race detector:

```
$ cd tasks/03-panic-concept/recoverer
$ go test -race .
ok      tasks/03-panic-concept/recoverer    0.224s
```

Задача "DummyRecoverer"

Ссылка на заготовку.

Очевидно, что паникующий код хотелось бы обнаруживать и исправлять ещё на стадии тестирования. Поэтому было решено написать `DummyRecoverer` – тип,

имеющий идентичное с `Recoverer` (из предыдущей задачи) API, но при этом не отлавливающий паники *от слова совсем*.

До этого момента мы уделяли много внимания тому, как писать железобетонный код; теперь же можно расслабиться, получить удовольствие и написать код, который должен бахать 😈.

Тест "История одного defer"

[Исходник примера](#).

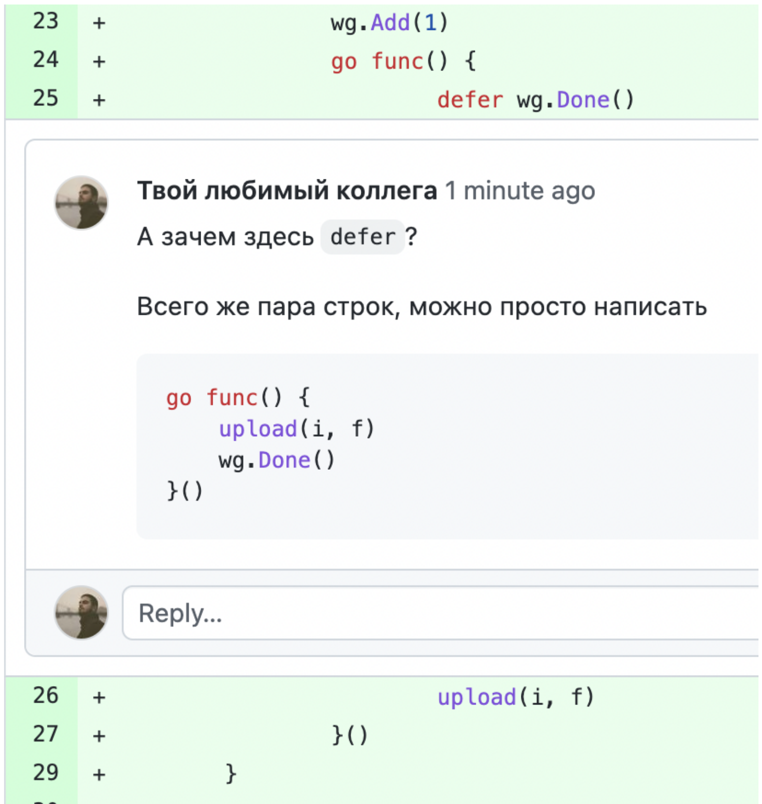
Перед вами кусочек только что разработанного модуля для конкурентной загрузки файлов на сервер:

```
for i, f := range files {
    i, f := i, f

    wg.Add(1)
    go func() {
        defer wg.Done()
        upload(i, f)
    }()

}
```

На ревью поступил следующий комментарий:



Разработчик согласился с замечанием и переписал код:

```
for i, f := range files {
    i, f := i, f

    wg.Add(1)
    go func() {
        upload(i, f)
        wg.Done()
    }()

}
```

Затем проект претерпел рефакторинг и было решено все `go`-вызовы оборачивать в хелпер `safeGo`:

```
func safeGo(f func()) {
    defer func() {
```

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

Не обошло это стороной и наш модуль:

```go
for i, f := range files {
    i, f := i, f

    wg.Add(1)
    safeGo(func() {
        upload(i, f)
        wg.Done()
    })
}
```

Функция `upload`, непосредственно работающая с файлом, активно дорабатывалась: добавлялись оптимизации, логирование прогресса загрузки и пр. улучшательства, одно из которых в какой-то момент привело к панике внутри этой функции во время работы сервиса в проде.

Исходя из этого выберите, что произошло с программой дальше?

Выберите все подходящие ответы из списка

- ☐ Программа упала из-за возникшей в upload паники
- ☐ Паника, возникшая в upload, была прервана safeGo
- ☐ Если загрузка файлов происходила в рамках синхронного приложения (например, CLI-тулзы), то в момент wg.Wait() мы получили deadlock и программа упала с "fatal error"
- ☐ Программа продолжила свою работу без негативных последствий

- [ ] Если загрузка файлов происходила в рамках асинхронного приложения (например, внутри хендлера HTTP-сервера), то мы получили висячие из-за wg.Wait() горутины, как следствие, с течением времени – утечку ресурсов и отсутствие реагирования приложения на внешние запросы

Задача "errgroup"

Ссылка на заготовку.

Вам необходимо реализовать `errgroup.Group` – тип, позволяющий

- запускать функции в отдельных горутинах;
- ожидать завершения работы функций;
- получать первую ненулевую ошибку, произошедшую во время работы функций.

Отличительной особенностью нашего типа является его устойчивость к паникам внутри запускаемых функций:

- если паникнули ошибкой (значением типа `error`), то группа запишет её как есть (для дальнейшей выдачи в `Wait()`);
- если паникнули строкой (значением типа `string`), то группа создаст на её основе ошибку;
- иначе группа создаст ошибку на основе строкового представления значения, полученного с помощью спецификатора `%#v`.

Также к тексту ошибок, полученным в результате паники, прибавляется специальный префикс.

Больше подробностей см. в заготовке задачи и тестах.

Не забудьте прогнать своё решение через race detector:

```
$ cd tasks/03-panic-concept/errgroup
$ go test -race .
```

```
ok      tasks/03-panic-concept/errgroup      0.901s
```

Задача "Go Helper"

Ссылка на заготовку.

Вам необходимо реализовать следующую фунцию:

```go
var ErrPanicOccurred = errors.New("panic occurred")

// Go позволяет запустить функцию fn в отдельной горутине
// и получить результат её работы через выходной канал.
// Если функция запаникует, то в канале окажется ErrPanicOccurred.

func Go(fn func() error) <-chan error
```

Больше подробностей см. в заготовке задачи и тестах 🤷‍♂️.

Задача "Ваше мнение"

На [просторах сети]() можно встретить следующий код:

```
ctx, cancel := context.WithCancel(context.Background())
go func() {
    if x := recover(); x != nil {
        log.Printf("[WARN] run time panic:\n%v", x)
        panic(x)
    }

    // catch signal and invoke graceful termination
    stop := make(chan os.Signal, 1)
    signal.Notify(stop, os.Interrupt, syscall.SIGTERM)
    <-stop
    log.Printf("[WARN] interrupt signal")
    cancel()

}()
```

Что вы думаете конкретно об этой части?

```
go func() {
    if x := recover(); x != nil {
        log.Printf("[WARN] run time panic:\n%v", x)
        panic(x)
    }
    // ...

}()
```

Для чего это сделано? Имеет ли подобное смысл?

Данная задача имеет свободную форму ответа – любой ответ будет считаться верным. Она нужна больше для самопроверки, попробуйте представить ход своих мыслей, если бы, например, вам задали такой вопрос на собеседовании.

Но ничего не мешает опубликовать свои рассуждения в дополнение к авторским на вкладке "Решения" 👍.

Тест "Recovery Middleware"

Исходник примера.

Приложение представляет собой HTTP-сервер, реализованный с помощью библиотеки echo, в некоторых хендлерах которого происходят конкурентные вычисления.

Последнее время участились падения сервиса из-за случайных ошибок, приводящих к паникам. Тимлид попросил своего стажёра повысить отказоустойчивость приложения и добавить код, необходимый для прерывания и логирования паник.

Стажёр, недолго думая, порыскал в сети и добавил к серверу **recovery middleware**:

```
e.Use(middleware.RecoverWithConfig(middleware.RecoverConfig{
    StackSize: 1 << 10, // 1 KB
    LogLevel:  log.ERROR,

}))
```

Достаточно ли этого, чтобы возможная паника в случайным образом взятом обработчике запроса больше не "роняла" приложение?

Выберите все подходящие ответы из списка

- [ ] Recovery middleware спасёт от паник внутри любых хендлеров, потому что каждый запрос выполняется в отдельной горутине и recover() покрывает как эту горутину, так и созданные в ней горутины
- [ ] Recovery middleware не спасёт от паник внутри новых горутин, порождённых хендлером
- [ ] Recovery middleware не спасёт ни от каких паник
- [ ] Recovery middleware не спасёт от паник внутри функций, отложенных во время работы хендлера
- [ ] Recovery middleware спасёт от паник внутри хендлеров, в которых отсутствует создание горутин

Неуловимые сбои

До этого мы говорили о паниках в других горутинах, неуловимых из текущей. Но также в Go существует множество сбоев, которых в принципе нельзя восстановить никакими `recover()`.

Их можно обнаружить с помощью поиска вызовов уже знакомой нам функции runtime.throw:

```
// https://github.com/golang/go master
$ grep -R "\tthrow(" . | sort -R | head -n 30
./runtime/lock_sema.go:         throw("notetsleepg on g0")
./runtime/trace.go:       throw("missing traceGCSweepStart")
./runtime/proc.go:        throw("gfput: bad status (not Gdead)")
./runtime/lockrank_on.go:               throw("inconsistent world stop
count")
./runtime/chan.go:        throw("makechan: bad alignment")
./runtime/mgcmark.go:           throw("markroot: bad index")
./runtime/os_netbsd.go:         throw("runtime.newosproc")
./runtime/mbitmap.go:                   throw("heapBitsSetType: called
with non-pointer type")
./runtime/mpagealloc.go:                        throw("bad summary data")
./runtime/cgocall.go:throw("misaligned stack in cgocallback")
./runtime/mbitmap.go:                   throw("bad number of remaining
words")
./runtime/mcache.go:      throw("refill of span with free space
remaining")
./runtime/os_aix.go:      throw("syscall clock_gettime failed")
./runtime/os_aix.go:      throw("syscall clock_gettime failed")
./runtime/runtime1.go:          throw("xchg64 failed")
./runtime/runtime1.go:          throw("xchg64 failed")
./runtime/mgcpacer.go:                  throw("trigger underflow")
./runtime/map_fast32.go:        throw("concurrent map read and map
write")
```

```
./runtime/map_fast32.go:        throw("concurrent map read and map
write")
./runtime/stack.go:          throw("shrinkstack at bad time")
./runtime/mstats.go:         throw("short slice passed to readGCStats")
./runtime/os3_solaris.go:    throw("exitThread")
./runtime/cgocall.go:        throw("can't happen")
./runtime/mbitmap.go:        throw("heapBitsSetTypeGCProg: small
allocation")
./runtime/cgocall.go:throw("cgo not implemented")
./runtime/race.go:           throw("raceinit: race build must use cgo")
./runtime/time.go:           throw("bad timer heap len")
./runtime/mprof.go:          throw("bad use of bucket.mp")
./runtime/proc.go:           throw("runtime: sudog with non-nil c")

./runtime/os_windows.go:     throw("bad newosproc0")
```

Большинство вызовов `throw` являются ассертами, т.е. их невозможно получить при отсутствии логических ошибок в компиляторе и при *нормальной* работе рантайма. Но некоторые из сбоев всё же можно повторить руками, и наверняка вы уже встречались с ними (исходники примеров) 😈.

- **fatal error: concurrent map read and map write**

```go
// https://goplay.tools/snippet/3JnGWcWDjfJ

func main() {
    defer handlePanic()

    m := map[string]int{}
    go func() {
        defer handlePanic()
        for {
            m["x"] = 42
        }
    }()
    for {
        _ = m["x"]
    }
}

/*
```

```
fatal error: concurrent map read and map write

goroutine 1 [running]:
runtime.throw({0x499b90, 0x10})
	/usr/local/go-faketime/src/runtime/panic.go:1198 +0x71
fp=0xc0000a2ed0 sp=0xc0000a2ea0 pc=0x42fb71
runtime.mapaccess1_faststr(0x404f79, 0xc0000a6210, {0x4948dc, 0x1})
	/usr/local/go-faketime/src/runtime/map_faststr.go:21 +0x3a5
fp=0xc0000a2f38 sp=0xc0000a2ed0 pc=0x40fd65
main.main()
	/tmp/sandbox1162044493/prog.go:16 +0xa5 fp=0xc0000a2f80
sp=0xc0000a2f38 pc=0x47e3a5
runtime.main()
	/usr/local/go-faketime/src/runtime/proc.go:255 +0x213
fp=0xc0000a2fe0 sp=0xc0000a2f80 pc=0x4321b3
runtime.goexit()
	/usr/local/go-faketime/src/runtime/asm_amd64.s:1581 +0x1
fp=0xc0000a2fe8 sp=0xc0000a2fe0 pc=0x45ac21

goroutine 20 [runnable]:
main.main.func1()
	/tmp/sandbox1162044493/prog.go:12 +0x57
created by main.main
	/tmp/sandbox1162044493/prog.go:9 +0x8

*/
```

- **fatal error: go of nil func value**

```
// https://goplay.tools/snippet/t3O7CzuV8wd

func main() {
	defer handlePanic()

	var f func()
	go f()
}

/*
fatal error: go of nil func value

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

```
    /tmp/sandbox3436230012/prog.go:9 +0x3f

*/
```

- **fatal error: stack overflow**

```
// https://goplay.tools/snippet/5Pdi9gYwY13

type fatArray [1 << 20]int64

func main() {
    defer handlePanic()

    var f func(a fatArray)
    f = func(a fatArray) {
        f(a)
    }
    f(fatArray{})
}

/*
runtime: goroutine stack exceeds 1000000000-byte limit
runtime: sp=0x1401f8ffb50 stack=[0x1401f100000, 0x1403f100000]
fatal error: stack overflow

runtime stack:
runtime.throw({0x10478d61a, 0xe})
        /usr/local/go/src/runtime/panic.go:1198 +0x54
runtime.newstack()
        /usr/local/go/src/runtime/stack.go:1088 +0x56c
runtime.morestack()
        /usr/local/go/src/runtime/asm_arm64.s:303 +0x70

*/
```

- **fatal error: all goroutines are asleep - deadlock!**

```
// https://goplay.tools/snippet/sE9jcdB4rsE

func main() {
     defer handlePanic()
```

```
    select {}
}

/*
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [select (no cases)]:
main.main()
    /tmp/sandbox2100114146/prog.go:7 +0x3b

*/
```

и многие другие!

debug.SetPanicOnFault

[Исходник примера](#).

Залезание в чужое адресное пространство, повреждение памяти во время работы рантайма и пр. причины сигнала [SIGSEGV](#), по умолчанию приводят к аварийному невосстанавливаемому (**unrecoverable**) сбою:

```
// https://goplay.tools/snippet/p4xMGtlEtSu

const kernelSpaceAddr = uintptr(0xffffffffffffffff)

func main() {
    defer handlePanic()

    v := *(*byte)(unsafe.Pointer(kernelSpaceAddr))  // Пытаемся
залезть туда, куда нам нельзя.
    fmt.Println(v)
}

/*
unexpected fault address 0xffffffffffffffff
fatal error: fault
[signal SIGSEGV: segmentation violation code=0x1
addr=0xffffffffffffffff pc=0x47df61]

goroutine 1 [running]:
runtime.throw({0x49490e, 0x10101d961870108})
    /usr/local/go-faketime/src/runtime/panic.go:1198 +0x71
fp=0xc000068ed8 sp=0xc000068ea8 pc=0x42f791
```

```
runtime.sigpanic()
	/usr/local/go-faketime/src/runtime/signal_unix.go:742 +0x2f6
fp=0xc000068f28 sp=0xc000068ed8 pc=0x442ff6
main.main()
	/tmp/sandbox3074600064/prog.go:13 +0x41 fp=0xc000068f80
sp=0xc000068f28 pc=0x47df61
runtime.main()
	/usr/local/go-faketime/src/runtime/proc.go:255 +0x213
fp=0xc000068fe0 sp=0xc000068f80 pc=0x431dd3
runtime.goexit()
	/usr/local/go-faketime/src/runtime/asm_amd64.s:1581 +0x1
fp=0xc000068fe8 sp=0xc000068fe0 pc=0x45a841

*/
```

Но функция debug.SetPanicOnFault позволяет смягчить это поведение, заменив аварию программы на панику:

```
// runtime/debug/garbage.go
package debug

// SetPanicOnFault controls the runtime's behavior when a program
faults
// at an unexpected (non-nil) address. Such faults are typically
caused by
// bugs such as runtime memory corruption, so the default response is
to crash
// the program. Programs working with memory-mapped files or unsafe
// manipulation of memory may cause faults at non-nil addresses in
less
// dramatic situations; SetPanicOnFault allows such programs to
request
// that the runtime trigger only a panic, not a crash.
// The runtime.Error that the runtime panics with may have an
additional method:
//     Addr() uintptr
// If that method exists, it returns the memory address which
triggered the fault.
// The results of Addr are best-effort and the veracity of the result
// may depend on the platform.
// SetPanicOnFault applies only to the current goroutine.
// It returns the previous setting.
```

```go
func SetPanicOnFault(enabled bool) bool
```

Более того в каких-то случаях мы теперь можем получить адрес памяти, который привёл к панике:

```go
const kernelSpaceAddr = uintptr(0xffffffffffffffff)

func main() {
    defer handleFault()

    debug.SetPanicOnFault(true)

    v := *(*byte)(unsafe.Pointer(kernelSpaceAddr)) // Пытаемся залезть туда, куда нам нельзя.
    fmt.Println(v)
}

func handleFault() {
    if r := recover(); r != nil {
        fmt.Println("recovered:", r)

        var addressable interface {
            Addr() uintptr
        }
        if err, ok := r.(error); ok && errors.As(err, &addressable) {
            fmt.Println("panic from addr:", addressable.Addr())
        }
    }
}

/*
recovered: runtime error: invalid memory address or nil pointer dereference
panic from addr: 18446744073709551615

*/
```

Немногочисленные примеры применения данного debug API можно посмотреть [здесь](), правда большинство из них взято из форков [golang/go]() и как следствие сводится к оригинальному [TestSetPanicOnFault]().

Тест "Какие из сбоев являются recoverable?

Выберите все подходящие ответы из списка

- ☐ fatal error: go of nil func value
- ☐ fatal error: concurrent map read and map write
- ☐ unexpected fault address 0xff8ffffffffaf00 + debug.SetPanicOnFault(true)
- ☐ fatal error: stack overflow
- ☐ fatal error: all goroutines are asleep - deadlock!
- ☐ runtime error: invalid memory address or nil pointer dereference
- ☐ interface conversion: interface {} is string, not []int
- ☐ runtime: program exceeds 10000-thread limit
  fatal error: thread exhaustion
- ☐ fatal error: sync: unlock of unlocked mutex
- ☐ fatal error: trace: alloc too large
- ☐ runtime error: index out of range [1] with length 1
- ☐ fatal error: runtime: out of memory
- ☐ send on closed channel

Выводы

Благодаря данному уроку мы узнали, что **паники неразрывно связаны со своей горутиной**. Следует знать и помнить, какие проблемы это за собой несёт.

Также мы познакомились с **фатальными сбоями программы на Go**, так что стоит иметь в виду, что невозможно сделать абсолютно отказоустойчивое приложение (даже если вы повсеместно обложитесь `recover()`), и подобные вопросы стоит решать не только на уровне кода, но и на уровне сопутствующей инфраструктуры.

Будем считать, что рубикон пройден – мы рассмотрели основную базу, необходимую для понимания внутреннего устройства и уверенного применения `panic()` и `recover()`. Осталось дело за малым – узнать, когда и где стоит применять данный механизм, а когда лучше воздержаться от этого.

Вперёд навстречу лучшим практикам!