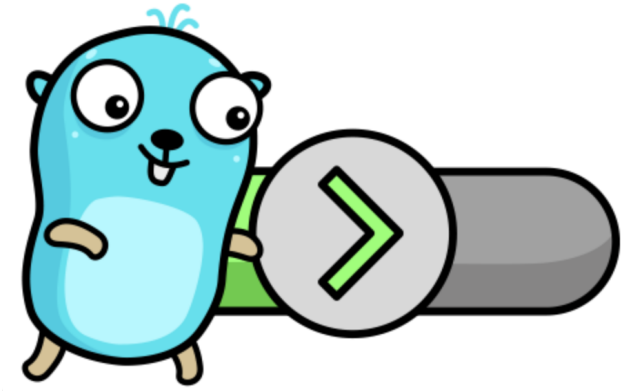
Непопулярный runtime.Goexit

В этом уроке мы рассмотрим созвучную с `os.Exit` функцию, но имеющую абсолютно другую природу 🙂.



runtime.Goexit

Функция runtime.Goexit позволяет завершить горутину, в которой она была вызвана, при этом никак не затрагивая остальные горутины:

```
package runtime

// Goexit terminates the goroutine that calls it. No other goroutine
is affected.
// Goexit runs all deferred calls before terminating the goroutine.
Because Goexit
// is not a panic, any recover calls in those deferred functions will
return nil.
//
// Calling Goexit from the main goroutine terminates that goroutine
// without func main returning. Since func main has not returned,
// the program continues execution of other goroutines.
// If all other goroutines exit, the program crashes.

func Goexit()
```

Мы не будем останавливаться на её реализации, но обозначим основные моменты:

- перед завершением горутины выполняются все отложенные (deferred) вызовы функций и методов;
- `runtime.Goexit` не приводит к панике и факт её вызова нельзя "отловить" через `recover`;
- `runtime.Goexit` не завершает текущий процесс подобно `os.Exit`;
- `runtime.Goexit` никак не влияет на прочие горутины.

```
// https://goplay.tools/snippet/nDoUpC507ad
package main

import (
    "fmt"
    "runtime"
    "sync"
    "time"
)

// Мы запускаем n горутин и через разное время убиваем
// каждую из них с помощью runtime.Goexit.

func main() {
    var wg sync.WaitGroup

    for i, sleep := range []time.Duration{
        150 * time.Millisecond,
        100 * time.Millisecond,
        50 * time.Millisecond,
    } {
        wg.Add(1)
        go func(i int, sleep time.Duration) {
            defer func() {
                wg.Done()
                fmt.Printf("recovered #%d: %v\n", i, recover()) // <-
Всегда будет <nil>.
            }()

            worker(i, sleep)
        }(i, sleep)
    }
```

```
    wg.Wait()
    fmt.Println("DONE")
}

func worker(i int, sleep time.Duration) {
    defer fmt.Printf("worker #%d was killed\n", i)

    time.Sleep(sleep)
    runtime.Goexit() // <- Убиваем воркер, без влияния на остальные воркеры.

    fmt.Println("after work") // <- Этого мы не увидим, unreachable code.
}

/*
worker #2 was killed
recovered #2: <nil>
worker #1 was killed
recovered #1: <nil>
worker #0 was killed
recovered #0: <nil>
DONE
*/
```

И самый интересный факт: если вы вызовите `runtime.Goexit` в `main`, то это прервёт main-горутину, но не приведёт к выходу из функции `main`, как следствие – не приведёт к выходу из программы, и все остальные горутины продолжат свою работу. Затем, когда запущенных горутин не останется, мы получим **deadlock** и процесс завершится с ненулевым кодом возврата:

```
// https://goplay.tools/snippet/8_n83Ri6tkK
package main

import (
    "fmt"
    "runtime"
    "time"
)
```

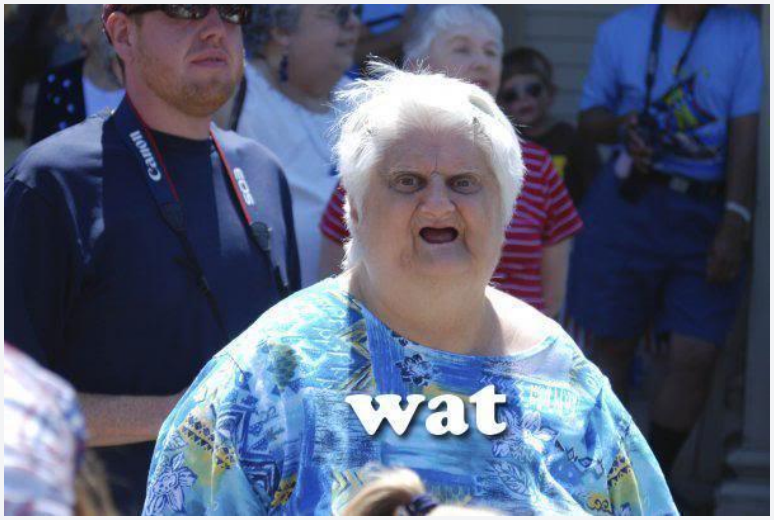
```go
func main() {
    defer fmt.Println("deferred operation") // <- Это мы увидим.

    // Все горутинки полностью и успешно завершат
    // свою работу несмотря на Goexit ниже.
    go worker()
    go worker()
    go worker()
    runtime.Goexit()

    fmt.Println("hello!") // <- Это мы не увидим, unreachable code.
}

func worker() {
    for i := 0; i < 3; i++ {
        time.Sleep(time.Millisecond * 100)
        fmt.Println("work: ", i)
    }
}

/*
deferred operation
work:  0
work:  0
work:  0
work:  1
work:  1
work:  1
work:  2
work:  2
work:  2
fatal error: no goroutines (main called runtime.Goexit) - deadlock!
*/
```



Тест "runtime.Goexit -> os.Exit"

Постарайтесь, не запуская кода, написать, что выведет последний пример из предыдущего шага, если в нём поменять `runtime.Goexit` на `os.Exit`:

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

    go worker()
    go worker()
    go worker()
    os.Exit(0)    // Было runtime.Goexit().

    fmt.Println("hello!")
}

func worker() {
    for i := 0; i < 3; i++ {
        time.Sleep(time.Millisecond * 100)
        fmt.Println("work: ", i)
    }

}
```

P.S. Отсутствие вывода эквивалентно пробелу.

Использование runtime.Goexit

Одним из более-менее оправданных примеров использования `runtime.GoExit` (и на данный момент единственным, представленным в стандартной библиотеке) является предоставление API по досрочному завершению запущенной вами, но более **неконтролируемой** вами, горутины.

Например (исходник примера):

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

func main() {
    r := NewRunner()

    r.Run("first fn", func(r *R) {
        fmt.Println("first function is done")
    })

    r.Run("second fn", func(r *R) {
        defer fmt.Println("second function is done (defer)")

        r.Interrupt() // Внутри функции пользуемся API того, кто эту функцию будет запускать.
        fmt.Println("unreachable code")
    })

    fmt.Println()
    fmt.Println("first fn was interrupted?", r.WasInterrupted("first fn"))    // false
    fmt.Println("second fn was interrupted?", r.WasInterrupted("second fn")) // true
}

// ...

func (r *R) Interrupt() {
    r.interrupted[r.name] = true
    runtime.Goexit()

}
```

(В качестве самостоятельной работы подумайте, как реализовать подобное без использования `runtime.Goexit()`. Код внутри `main` и вывод программы поменяться не должен).

Как вы уже могли догадаться, стандартная библиотека Go использует `runtime.Goexit` в реализации функций `(*testing.T).SkipNow` и `(*testing.T).FailNow`:

```
// testing/testing.go
package testing

// T is a type passed to Test functions to manage test state and
support formatted test logs.
type T struct {
    common
    isParallel bool
    context    *testContext // For running tests and subtests.
}

// SkipNow marks the test as having been skipped and stops its
execution
// by calling runtime.Goexit.
// If a test fails (see Error, Errorf, Fail) and is then skipped,
// it is still considered to have failed.
// Execution will continue at the next test or benchmark. See also
FailNow.
// SkipNow must be called from the goroutine running the test, not
from
// other goroutines created during the test. Calling SkipNow does not
stop
// those other goroutines.
func (c *common) SkipNow() {
    c.mu.Lock()
    c.skipped = true
    c.finished = true
    c.mu.Unlock()
    runtime.Goexit()                                    // <- !!!
}

// FailNow marks the function as having failed and stops its
execution
// by calling runtime.Goexit (which then runs all deferred calls in
the
// current goroutine).
// Execution will continue at the next test or benchmark.
// FailNow must be called from the goroutine running the
// test or benchmark function, not from other goroutines
// created during the test. Calling FailNow does not stop
```

```
// those other goroutines.
func (c *common) FailNow() {
    c.Fail()

    // Calling runtime.Goexit will exit the goroutine, which
    // will run the deferred functions in this goroutine,
    // which will eventually run the deferred lines in tRunner,
    // which will signal to the test loop that this test is done.
    // ...
    c.mu.Lock()
    c.finished = true
    c.mu.Unlock()
    runtime.Goexit()                                    // <- !!!

}
```

Таким образом, пакет `testing` предоставляет разработчику API для пропуска или моментального завершения тестовой горутины и понятно, что сам `testing` не может контроллировать и знать, где именно разработчик решит заиспользовать этот код (и не забудет ли он после вызова условного `t.Skip()` вставить `return`).

В "реальной" же повседневной практике `runtime.Goexit` не встречается от слова совсем. Полезно знать об этом механизме, но его применение стоит под вопросом, так как:

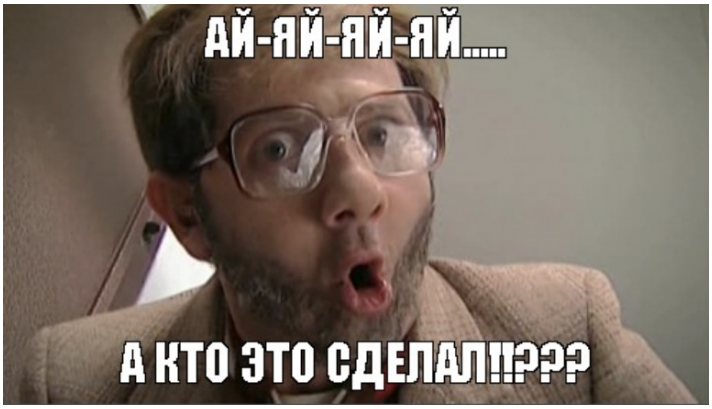
- усложняет понимание кода;
- требует от разработчика хорошей уверенности в своих действиях и знания, к чему они приведут;
- побуждают разработчика отказываться от каналов, грамотной организации конкурентного кода, использования контекста и т.д.

Небольшой набор ~~не самых лучших~~ примеров использования `runtime.Goexit` можно посмотреть [здесь](#).

Пишите, что вы думаете об этом в комментариях, а также делитесь примерами ситуаций, когда по вашему мнению без `runtime.Goexit` не обойтись.

Задача "Что произошло?"

Ссылка на заготовку.



Вам необходимо реализовать хелпер, запускающий функцию и позволяющий понять, от чего она завершилась:

- от паники внутри;
- от вызова `runtime.Goexit`;
- от штатного выхода из функции.

```
type ExitReason int

const (
    // ExitReasonRegularReturn означает, что функция завершилась в
штатном режиме.
    ExitReasonRegularReturn ExitReason = iota + 1

    // ExitReasonPanic означает, что функция запаниковала.
    ExitReasonPanic

    // ExitReasonGoexit означает, что функция вызвала runtime.Goexit.
    ExitReasonGoexit
)

// WhatHappened вызывает fn и сообщает о статусе её завершения.
```

```
func WhatHappened(fn func()) ExitReason
```

Больше подробностей см. в заготовке задачи и тестах.

Удачи! В этой задаче она понадобится как никогда 🙂.

runtime.Goexit vs panic

В позапрошлом шаге мы рекомендовали задуматься перед использованием `runtime.Goexit`. Если же вы решили миксовать эту функцию с паниками, то рекомендуем задуматься вдвойне 😃.

Посмотрим на небольшой пример:

```
// https://goplay.tools/snippet/Qu-eextB-ry
package main

import "runtime"

func main() {
    c := make(chan struct{})
    go func() {
        defer close(c)
        defer runtime.Goexit()
        panic("boom!")
    }()

    <-c
    fmt.Println("OK")
}
```

Постарайтесь, не запуская кода, представить, что произойдёт?

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

Точно представили? Ещё не поздно пролистать наверх и подумать самостоятельно.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

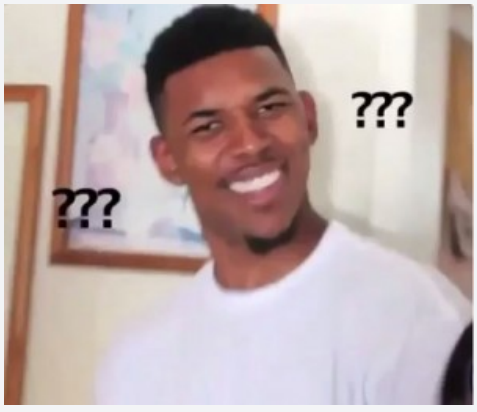
.

Как мы помним, независимо от того, что паника происходит в отдельной горутине, она крашит всю программу.

Но в данном случае `runtime.Goexit` не только убивает горутину, но и останавливает панику в ней!

```
$ go run main.go
OK
$ echo $?
0


# Верно вплоть до Go 1.18
```



Если для вас подобное поведение оказалось ожидаемым, то ничего не имеем против, т.к. даже для разработчиков Go не совсем очевидно, каким образом механизмы `runtime.Goexit` и `panic` должны пересекаться и должны ли:

golang/go: Issues: runtime: a runtime.Goexit call will cancel a panic, which seems unexpected/incorrect

В любом случае уже сейчас понятно, что стоит быть бдительным и по возможности держаться подальше от подобного кода 🙂.

Выводы

В этом уроке мы получили ~~очередную бомбу~~ очередное знание, которым полезно обладать, но которое не стоит спешить применять в реальной жизни.

Надеемся, что вам понравилось!