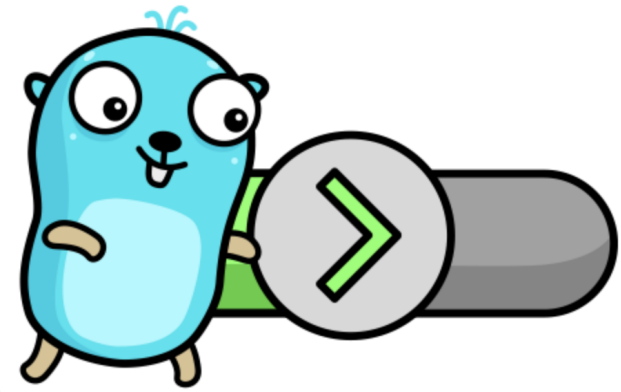
Паника во время паники

В этом уроке мы посмотрим на редкий кейс паники во время паники и поймём, для чего нужна была вся эта логика с фреймовыми указателями, которую мы разбирали ранее.



Паника при панике – это законно? Да!

Спецификация Go гласит:

```
The recover function allows a program to manage behavior of a
panicking goroutine. Suppose a function G defers
a function D that calls recover and a panic occurs in a function on
the same goroutine in which G is executing.
When the running of deferred functions reaches D, the return value of
D's call to recover will be the value

passed to the call of panic. If D returns normally, without starting
a new panic, the panicking sequence stops.
```

Обратим внимание на

```
If D returns normally, without starting a new panic, the panicking
sequence stops.
```

Т.е. отложенная функция D с `recover()` внутри остановит панику, если сама не запаникует. Значит сама возможность подобного допускается?


```
// https://goplay.tools/snippet/gUzoSY6QSDH
package main

import "fmt"

func main() {
    defer func() {
        defer func() {
            fmt.Println("before third panic")
            panic("triple kill")
        }()

        fmt.Println("before second panic")
        panic("double punch")
    }()

    fmt.Println("before first panic")
    panic("sky is falling")
}

/*
before first panic
before second panic
```

```
before third panic
panic: sky is falling
     panic: double punch
     panic: triple kill

goroutine 1 [running]:
main.main.func1.1()
     /tmp/sandbox017105870/prog.go:9 +0x95
panic(0x4a27a0, 0x4d8970)
     /usr/local/go-faketime/src/runtime/panic.go:965 +0x1b9
main.main.func1()
     /tmp/sandbox017105870/prog.go:13 +0xb7
panic(0x4a27a0, 0x4d8930)
     /usr/local/go-faketime/src/runtime/panic.go:965 +0x1b9
main.main()
     /tmp/sandbox017105870/prog.go:17 +0xb7

/*
```

Гошка скушала паники во время других паник и не подавилась, даже красиво вывела их все на экран (обратите внимание на отступ у последующих паник):

```
panic: sky is falling
    panic: double punch

    panic: triple kill
```

Как же это работает?

И снова runtime.gopanic

Вернёмся к нашей излюбленной [runtime.gopanic] и рассмотрим куски её реализации, которые до этого скрывали от вас:

```
package runtime

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

    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
        }

        // If defer was started by earlier panic or Goexit (and,
since we're back here, that
        // triggered a new panic), take defer off list. An earlier
panic will not continue
        // running, but we will make sure below that an earlier
Goexit does continue running.
        //
        if d.started {                    // Если ранее мы уже вызывали
этот defer.
            if d._panic != nil {
                d._panic.aborted = true // И внутри него была паника,
мы её прерываем.
            }
            d._panic = nil
            // ...
            d.fn = nil
            gp._defer = d.link            // И переходим к следующему
defer.
            freedefer(d)
            continue
        }

        // Mark defer as started, but keep on list, so that traceback
        // can find and update the defer's argument frame if stack
growth
        // or a garbage collection happens before executing d.fn.
        d.started = true

        // Record the panic that is running the defer.
        // If there is a new panic during the deferred call, that
panic
        // will find d in the list and will mark d._panic (this
panic) aborted.
        d._panic = (*_panic)(noescape(unsafe.Pointer(&p)))
```

```
        // ...
        // Вызов отложенной через текущий defer функции. Если во
время него произойдёт паника,
        // то мы снова окажемся в начале runtime.gopanic, а если
внутри функции есть recover(),
        // то он восстановит текущую панику и мы перейдём к коду
ниже.
        p.argp = unsafe.Pointer(getargp())
        // ...
        reflectcall(nil, unsafe.Pointer(d.fn), deferArgs(d),
uint32(d.siz), uint32(d.siz), uint32(d.siz), &regs)

        // ...
        if p.recovered {
            gp._panic = p.link

            // Если сработал recover() для текущей паники и
существует ранее прерванная паника,
            // то восстанавливаем поток выполнения программы,
используя её.
            if gp._panic != nil && gp._panic.goexit &&
gp._panic.aborted {
                gp.sigcode0 = uintptr(gp._panic.sp)
                gp.sigcode1 = uintptr(gp._panic.pc)
                mcall(recovery)
                throw("bypassed recovery failed") // mcall should not
return
            }

            // ...
            // Aborted panics are marked but remain on the g.panic
list.
            // Remove them from the list.
            for gp._panic != nil && gp._panic.aborted {
                gp._panic = gp._panic.link
            }

            // ...
            // Уже знакомый нам переход.
            gp.sigcode0 = uintptr(sp)
            gp.sigcode1 = pc
            mcall(recovery)
            throw("recovery failed") // mcall should not return
        }
```

```
    }

    // ran out of deferred calls - old-school panic now
    // Because it is unsafe to call arbitrary user code after
freezing
    // the world, we call preprintpanics to invoke all necessary
Error
    // and String methods to prepare the panic strings before
startpanic.
    preprintpanics(gp._panic)

    fatalpanic(gp._panic)           // Внутри вызов printpanics.
    // ...
}

// Print all currently active panics. Used when crashing.
// Should only be called after preprintpanics.
func printpanics(p *_panic) {
    if p.link != nil {
        printpanics(p.link)
        if !p.link.goexit {
            print("\t")              // Знакомые нам отступы.
        }
    }
    if p.goexit {
        return
    }
    print("panic: ")
    printany(p.arg)
    if p.recovered {
        print(" [recovered]")
    }
    print("\n")
}
```

Из кода выше можно вывести следующие основные моменты:

- паники накапливаются в список паник текущей горутины (поэтому мы можем, например, распечатать их все), аналогично тому, как

накапливаются отложенные функции;

- очередная паника прерывает выполнение текущей паники и "встаёт на её место";

- выполняемый в текущий момент `defer` знает, какая паника произошла последней (даже если она случилась не в текущем, а в следующем фрейме);

- `recover()` может восстановить панику только в текущем фрейме, соответственно возможна ситуация, когда вложенная паника будет восстановлена, но затем управление перейдёт к предыдущей панике и программа всё равно упадёт.

Возвращаясь к фрейм пойнтерам

В предыдущем уроке мы разбирали пример, когда разработчик думал, что "внутренний" `recover()` отловит как возможные "внутренние", так и "внешние" паники:

```
// 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 {        // Ошибка, это не
подействует на "внешнюю" панику!
                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")
}
```

Но мы с вами помним, что `runtime.gorecover` сверяет фреймовые указатели, чтобы сопоставить `panic()` и `recover()`:

```
package runtime

func gorecover(argp uintptr) interface{} {
    // ...
    if /* ... */ && argp == uintptr(p.argp) {
        p.recovered = true
        return p.arg
    }
    return nil

}
```

Так вот, это сделано **специально** для того, чтобы пары `panic`-`recover` не пересекались!

При создании механизма паники разработчики языка посчитали, что будет крайне неочевидно, если `recover()`'ы будут останавливать паники из "чужих" фреймов:

It's important that recover only work when called directly from a deferred function, so that a deferred function can itself call a function that uses a deferred recover. That is, given this code

```
func f() {
    defer func() {
        g()
    }()
    panic(1)
}

func g() bool {
    defer func() {
        if x := recover(); x != nil {
            // ...
        }
    }()

    return someFunction()

}
```

the recover in g's deferred function must not pick up the panic in f. ***If it did, it would be impossible for an independent package to reliably use panic/recover in a controlled way.***
(c) Ian Lance Taylor

Изобразим наглядно, о каких "парах" идёт речь выше:

```go
package main

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

        defer func() {
            recover()

            defer func() {
                recover()
            }()

            panic("triple kill")
        }()

        panic("double punch")
    }()

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

(надеемся, что этот "светофор" не вызовет у вас эпилепсию, главное – уловить закономерность)

Данный пример можно переписать следующим образом:

```go
// https://goplay.tools/snippet/XEFaHuxLELa

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

func aaa() {
    defer func() {
        recover()
        bbb()
    }()
    panic("double punch")
}
```

```
func bbb() {
    defer func() {
        recover()
    }()
    panic("triple kill")

}
```

Таким образом, каждый `recover` связан с исключительно с паникой в функции, в которой он отложен.

Выглядит не так уж плохо, как казалось на первый взгляд, не правда ли? 🙂

Тест "You Shall Not Pass!"

Мы понимаем, что с практической точки зрения данный тест имеет мало смысла (так как маловероятно, что вы встретите подобные ухищрения в боевом коде), но хочется ещё разок закрепить понимание связки `panic`-`recover`.

Выберите сниппеты, код в которых **не будет** паниковать.

1)

```
func main() {
    defer func() {
        defer func() {
            panic("3")
        }()
```

```
        panic("2")
    }()
    panic("1")

}
```

2)

```
func main() {
    defer func() {
        fmt.Println("recover for 1:", recover())

        defer func() {
            fmt.Println("recover for 2:", recover())

            defer func() {
                fmt.Println("recover for 3 ", recover())
            }()
            panic("3")
        }()
        panic("2")
    }()
    panic("1")

}
```

3)

```
func main() {
    defer func() {
        defer func() {
            defer func() {
                fmt.Println("recover for 3 only:", recover())
            }()
            panic("3")
        }()
        panic("2")
    }()
    panic("1")

}
```

4)

```
func main() {
```

```
    defer func() {
        fmt.Println("recover for 1:", recover())

        defer func() {
            fmt.Println("recover for 2:", recover())
            panic("3")
        }()
        panic("2")
    }()
    panic("1")
}
```

5)

```
func main() {
    defer func() {
        fmt.Println("recover for 1 only:", recover())

        defer func() {
            panic("3")
        }()
        panic("2")
    }()
    panic("1")

}
```

6)

```
func main() {
    defer func() {
        fmt.Println("recover for 1 only:", recover())
    }()

    defer func() {
        defer func() {
            panic("3")
        }()
        panic("2")
    }()
    panic("1")

}
```

7)

```
func main() {
    defer func() {
        defer func() {
            fmt.Println("recover for 1 only:", recover())
        }()

        defer func() {
            defer func() {
                panic("3")
            }()
            panic("2")
        }()
    }()
    panic("1")

}
```

8)

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

func foo() {
    defer func() {
        panic("2")
    }()
    panic("1")

}
```

Промежуточные выводы

Что тут можно сказать? Не грози Гошному компилятору, попивая смузи у себя за макбуком.

Иными словами – будьте бдительны, если после восстановления паники выполняете сложные `cleanup`-операции, т.к. они могут привести к следующей панике, которая уже не будет восстановлена.

Также помните, что `panic` с `recover` связаны пофреймово, но мы надеемся, что вам не придётся использовать `recover()` за пределами какого-нибудь хелпера или мидлвары, а в них вы будете использовать простой шаблонный код вида

```
defer func() {
    if r := recover(); r != nil {
        // Working with r.
        // ...
    }

}()
```

В следующем уроке мы рассмотрим один из главных подводных камней паники и на этом закончим рассмотрение механизмов паникования в принципе и наконец-то перейдём к лучшим практикам.