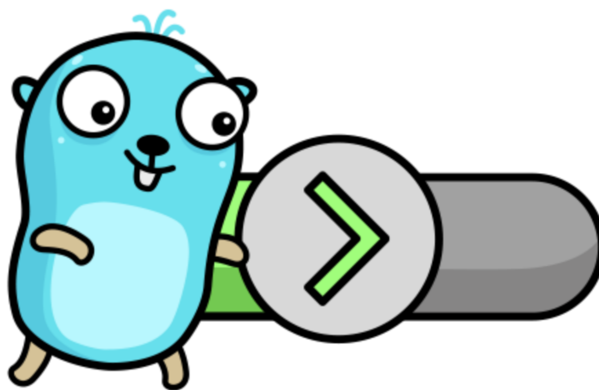
Паника: дополнительные главы

В этом уроке мы коснёмся небольшого количества оставшихся тем, связанных с паникой, но не нашедших своё место в предыдущих уроках.

Пишите, если от нас ускользнуло что-то интересное и волнующее вас, так как это последний урок, посвящённый непосредственно панике 🥲.



defer (func())(nil)()

Редкие и неприятные баги связаны с ошибкой откладывания непроинициализированных функций и методов (а точнее, проинициализированных [zero value](#) значением для них, т.е. `nil`).

При попытке вызвать `defer` от `nil`-функции, вы получите панику "**runtime error: invalid memory address or nil pointer dereference**" (что достаточно логично). Но неудобство заключается в том, что стектрейс от паники будет указывать на конец обрамляющей функции, ведь в момент выполнения отложенная функция находится именно там:

```
// https://goplay.tools/snippet/o9yHBvAOL-X
```

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      foo()
7  }
```

```
8
9  func foo() {
10     var f func()
11     defer f() // Аналогично `defer (func())(nil)()`.
12
13     fmt.Println("hello")
14     fmt.Println("world")
15
16 } // <- Паника указывает сюда.
17
```

```
/*
hello
world
panic: runtime error: invalid memory address or nil pointer
dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0
pc=0x47e01d]

goroutine 1 [running]:
main.foo()
	/tmp/sandbox4005020102/prog.go:16 +0xbd
main.main()
	/tmp/sandbox4005020102/prog.go:6 +0x17

*/
```

Кажется, что данная проблема не так страшна и в "реальной" жизни её легко обнаружить.

Но бывает так, что

- Функция, обрамляющая `defer`, очень длинная, и разработчик в принципе не видит `defer`, недоумевая (понятно, что тут больше вопрос к настройкам линтеров и архитектуре кода).

- В конце функции, на строчке, предшествующей той, на которую указывает паника, может располагаться код, который тоже может быть кандидатом

на панику `invalid memory address or nil pointer dereference`. И начинается дебаг (очевидно безрезультатный), почему именно этот код бахает.

Поэтому следует просто знать и помнить о данной особенности `defer`.

flag.PanicOnError

Полезно знать, что вы можете настраивать поведение пакета `flag` при ошибке парсинга аргументов командной строки:

```
// flag/flag.go
package flag

// ErrorHandling defines how FlagSet.Parse behaves if the parse
fails.
type ErrorHandling int

// These constants cause FlagSet.Parse to behave as described if the
parse fails.
const (
    ContinueOnError ErrorHandling = iota // Return a descriptive
error.
    ExitOnError                          // Call os.Exit(2) or for
-h/-help Exit(0).
    PanicOnError                         // Call panic with a
descriptive error.

)
```

Пример, чтобы поиграться:

```
$ cd examples/03-panic-concept/flag-error-handling

$ go run main.go -v
1.0.0

$ go run main.go -k
flag provided but not defined: -k
Usage of debug:
  -v    Print app version.
panic: flag provided but not defined: -k
```

```
goroutine 1 [running]:
flag.(*FlagSet).Parse(0x1400007a180, {0x14000010030, 0x1, 0x1})
        /usr/local/go/src/flag/flag.go:1021 +0x13c
main.main()
        examples/03-panic-concept/flag-error-handling/main.go:16
+0x144

exit status 2
```

Тестирование паник

Стандартная библиотека Go не предоставляет инструментов для тестирования паник (впрочем как и любых других инструментов для продвинутых и удобных проверок).

Поэтому приходится или писать свои хелперы или искать полезные функции среди сторонних модулей.

Уже знакомый вам, используемый авторами stretchr/testify (первое место по популярности на 2021 год), имеет ряд функций для работы с паниками:

- assert.Panics
- assert.PanicsWithError
- assert.PanicsWithValue
- assert.NotPanics

(и аналогичные в пакете require).

Пишите в комментариях, чем пользуетесь сами при тестировании в целом и при тестировании паник в частности.

Задача "AssertPanics"

Ссылка на заготовку.

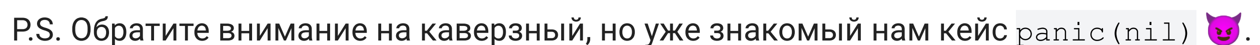
Вам необходимо реализовать пару функций

```go
type TestingT interface {
    Errorf(format string, args ...any)
    Helper()
}

func AssertPanics(t TestingT, f func()) bool { /*...*/ }


func AssertNotPanics(t TestingT, f func()) bool { /*...*/ }
```

Их алгоритм работы разберём на примере `AssertPanics`:

- Функция помечает себя как [тестовый хелпер](#).
- Запускает функцию `f` и возвращает `true`, если в ней произошла паника и `false` в обратном случае.
- Если паники не было, то тест помечается как завершившийся неуспешно (с помощью [t.Errorf](#)), недаром наша функция называется `AssertPanics`.

Больше подробностей см. в заготовке задачи и тестах.



P.S. Обратите внимание на каверзный, но уже знакомый нам кейс `panic(nil)` 😈.

Библиотеки для работы с паникой

Библиотек для работ с паникой кот наплакал. Чаще всего люди пишут свои кастомные обёртки, не вынося это в open source (наподобие реализованного нами ранее [Recoverer](#)).

Безопасные (с точки зрения паник) запускаторы функций и горутинок

- **[studiosol/async](#)**

```go
err := async.Run(ctx,
    func(ctx context.Context) error {
        user, err = user.Get(ctx, id)
        return err
    },
    func(ctx context.Context) error {
        songs, err = song.GetByUserID(ctx, id)
        return err
    },
    func(ctx context.Context) error {
        photos, err = photo.GetByUserID(ctx, id)
        return err
    },

)
```

- kenkyu392/go-safe

```go
// The panic that occurs in the function is automatically handled as an error.
fn := safe.Func(func() error {
    // Something that might cause a panic...
    return nil
})


if err := fn(); err != nil {
    // ...

}
```

Можно подружить с `errgroup`:

```go
eg := new(errgroup.Group)
for i := 0; i < n; i++ {
    eg.Go(fn)
}

if err := eg.Wait(); err != nil {
    // ...

}
```

- **[VividCortex/robustly]**

```
robustly.Run(func() { /* ... */ }, nil)
```

```
go robustly.Run(func() { /* ... */ }, &robustly.RunOptions{ /* ... */
})
```

Умная штука, рестартующая функцию, если в ней случилась паника. Призвана минимизировать проблемы, связанные с ошибками, которые случаются крайне редко. Своеобразный backoff. Если функция упрямо продолжит паниковать, то значит мы имеем дело не с временной, а с постоянной ошибкой, и `robustly` запаникует тоже.

Красивый вывод стектрейса с помощью maruel/panicparse

Интересная приблуда, судя по блогу умеет в HTML-интерфейс, а также парсит стектрейс не только от паник, но и от **data race**'ов и **deadlock**'ов. С помощью `panicparse` удобно оформлять баги.

Пример форматирования длиннющего трейса от ошибки внутри Shopify/sarama (модуль для работы с Kafka):

```
▶ pp kafka_sarama_bug.txt
fatal error: signal_recv: inconsistent state

1: running [Created by signal.Notify.func1.1 @ signal.go:150]
    runtime         panic.go:1116           throw(string(0x22f4739, len=31))
    signal          sigqueue.go:140         signal_recv(0x0)
    signal          signal_unix.go:23       loop()
    runtime         asm_amd64.s:1374        goexit()
1: chan receive [5 minutes]
    waiter          waiter.go:59            (*Waiter).Wait(*Waiter(#214))
    main            registry.go:150         (*ServiceRegistry).Wait(#93)
    main            main.go:74              cmdRun(*Context(#356))
    cli             app.go:528              HandleAction(interface{}(#2), *Context(#356), 0x0, 0x0)
    cli             command.go:174          Command.Run(*Context(#4), 0x3, 0x0, 0x0, 0x0, 0x0, 0x0, #22, 0x13, #27, ...)
    cli             app.go:279              (*App).Run(*App(#108), []string(#86 len=4 cap=4), 0x0, 0x0)
    main            main.go:216             main()
1: chan receive [5 minutes] [Created by main.cmdRun @ main.go:60]
    main            main.go:64              cmdRun.func1(*Context(#126))
1: runnable [Created by sarama.(*Broker).Open.func1 @ broker.go:211]
    time            time.go:241             Time.After(*Time(0xbfe0b757dfb23a30), #81, #63, 0xbfe0ba89f2716fd8, #669, #63, #79)
    go-metrics      sample.go:175           (*ExpDecaySample).update(*ExpDecaySample(#517), Time(0xbfe0b757dfb23a30), 44858688, 0x80)
    go-metrics      sample.go:142           (*ExpDecaySample).Update(*ExpDecaySample(#517), 128)
    go-metrics      histogram.go:199        (*StandardHistogram).Update(*StandardHistogram(#295), 0x80)
    sarama          broker.go:1359          (*Broker).updateRequestLatencyAndInFlightMetrics(*Broker(#440), Duration(#64))
    sarama          broker.go:1335          (*Broker).updateIncomingCommunicationMetrics(*Broker(#440), 10, Duration(#64))
    sarama          broker.go:886           (*Broker).responseReceiver(*Broker(#440))
    sarama          utils.go:43             withRecover(func(#296))
1: select [Created by healthcheck.(*HealthChecker).check @ healthcheck.go:154]
    sarama          broker.go:774           (*Broker).sendAndReceive(*Broker(#405), protocolBody(#54), protocolBody(#55), 0x0, 0x0)
    sarama          broker.go:283           (*Broker).GetMetadata(*Broker(#405), *MetadataRequest(#624), 0x0, 0x0, 0x0)
    sarama          client.go:850           (*client).tryRefreshMetadata(*client(#117), []string(0x0 len=0 cap=0), 3, Time(0x0), 0x0, 0x0, 0x0)
    sarama          client.go:450           (*client).RefreshMetadata(*client(#117), string(0x0, len=0), string(0x0, len=0), string(0x0, len=0))
    kafka           kafka.go:170            (*Client).HealthCheck(*Client(#124), Context(#46), 0x0, 0x0)
    healthcheck     healthcheck.go:174      (*HealthChecker).check.func1(*HealthChecker(#595), Context(#60), Time(#632), 0x5, #194, #46, #594)
7: select [Created by sarama.(*consumer).newBrokerConsumer @ consumer.go:722]
    sarama          broker.go:774           (*Broker).sendAndReceive(*, #50, *, #51, *, 0, 0)
    sarama          broker.go:356           (*Broker).Fetch(*, *, 0, 0, 0)
    sarama          consumer.go:913         (*brokerConsumer).fetchNewMessages(*, 0, 0, 0)
    sarama          consumer.go:781         (*brokerConsumer).subscriptionConsumer(*)
    sarama          utils.go:43             withRecover(*)
1: chan receive [Created by tdispatcher.(*Dispatcher).Run @ dispatcher.go:87]
    sarama          sync_producer.go:97     (*syncProducer).SendMessage(*syncProducer(#359), *ProducerMessage(#645), 0x0, 0x0, 0x0, 0x0)
    kafka           producer.go:54          (*producer).Produce(*producer(#560), Context(#46), Message(0x0), 0x0, 0x0, 0x0, #291, 0x49, ...)
    tdispatcher     methods.go:32           (*Dispatcher).Send(*Dispatcher(#89), Context(#46), *DispatcherMessage(#648), 0x0, 0x0)
    tdispatcher     methods.go:126          (*Dispatcher).produceHeartbeat(*Dispatcher(#89), Context(#45))
    tdispatcher     dispatcher.go:98        (*Dispatcher).Run.func1(*Dispatcher(#89), Context(#60))
```

Делитесь известными вам библиотеками в комментариях!

Паника и инфраструктура

Сухим списком приведём популярные полезности, связанные с паникой относительно вашего приложения в целом:

- echo: Recover Middleware
- Fiber: Recover middleware
- Gin Web Framework: Custom Recovery behavior
- grpc-ecosystem/go-grpc-middleware/recovery
- mitchellh/panicwrap

- Sentry Documentation: Handling Panics
- unrolled/recovery

(Необязательно ими пользоваться, но полезно знать, где можно почерпнуть вдохновение).

Выводы

Не стесняйтесь рыскать по чужим проектам и обогащаться различными практиками. Не бойтесь и привыкайте читать исходный код инструментов, которыми пользуетесь 🙂.

Этим уроком мы заканчиваем рассмотрение громадины под названием **Понятие паники в Go**.

До конца курса осталось буквально пара уроков, держитесь!