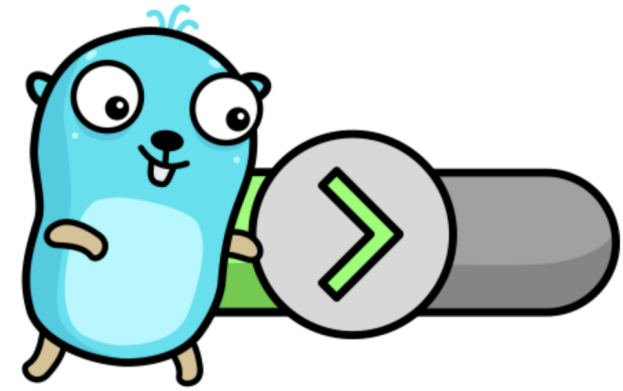
Dynamic & static assertion в Go

В этом уроке мы рассмотрим один из популярных кейсов использования паники – организации ассертов в Go.



Понятие "assertion"

Процитируем [википедию]():

```
Assertion в программировании — оператор, в котором предикат
(логическое выражение)
должен иметь всегда истинное значение в данной части кода. Программы
проверяют утверждения,
фактически оценивая предикат во время выполнения кода, и, если в
действительности предикат
ложен, программа преднамеренно останавливается или генерирует
исключение.

Утверждения могут делать код удобнее для прочтения, помогать
компилятору скомпилировать

код или помогать обнаружить дефекты в программе.
```

Например, ассерты в Си реализованы через макрос [assert()](), который позволяет проверять результат выражения и если оно ложно (равно нулю), то на экран выводится ошибка и процесс завершается с помощью функции `abort` ([исходник примера]()):

```c
#include <assert.h>
#include <stdlib.h>

void strcpy_strict(char *dst, char *src) {
    assert(dst && "dst string ptr is NULL");
    assert(src && "src string ptr is NULL");

    // Copying.
    while(*dst++ = *src++);
}

int main() {
    strcpy_strict(NULL, "hello world");

}
```

```
$ ./assert.out
Assertion failed: (dst && "dst string ptr is NULL"), function
strcpy_strict, file assert.c, line 5.
[1]    33981 abort      ./assert
```

При этом есть возможность "отключить" ассерты с помощью `#define NDEBUG` или указания компилятору флага `-DNDEBUG`:

```
$ gcc -DNDEBUG -o assert.out assert.c
$ ./assert.out

[1]    13566 segmentation fault  ./assert.out
```

А в Python ассерты реализованы через выражение assert (исходник примера):

```python
def avg(marks):
    assert len(marks) != 0, 'empty marks list'
    return round(sum(marks)/len(marks), 2)

marks = []
print(avg(marks))
```

```
$ python marks.py
```

```
Traceback (most recent call last):
  File "marks.py", line 6, in <module>
    print(avg(marks))
  File "marks.py", line 2, in avg
    assert len(marks) != 0, 'empty marks list'
AssertionError: empty marks list
```

Их так же можно отключить с помощью флага `-O (optimize generated bytecode slightly)`:

```
$ python -O marks.py
Traceback (most recent call last):
  File "marks.py", line 8, in <module>
    print(avg([]))
  File "marks.py", line 3, in avg
    return round(sum(marks) / len(marks), 2)

ZeroDivisionError: integer division or modulo by zero
```

Пишите в комментариях, в каких ещё языках есть подобный механизм и как он выглядит.

Assertion в Go

Как видно из примеров с предыдущего шага, ассерты помогают вставлять своеобразные проверки, позволяющие как можно раньше выявить так называемые "ошибки программирования" (не программы) – ошибки, связанные с невнимательностью, опечатками, неверной логикой реализации задачи и т.п.

Но в языке Go встроенная функция `assert` отсутствует и вот как это объясняют разработчики языка:

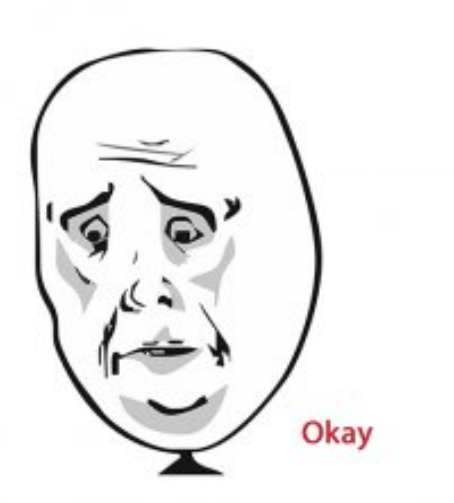
Why does Go not have assertions?

Go doesn't provide assertions. They are undeniably convenient, but our experience has been that programmers use them as a crutch to avoid thinking about proper error

handling and reporting. Proper error handling means that servers continue to operate instead of crashing after a non-fatal error. Proper error reporting means that errors are direct and to the point, saving the programmer from interpreting a large crash trace. Precise errors are particularly important when the programmer seeing the errors is not familiar with the code.

We understand that this is a point of contention. There are many things in the Go language and libraries that differ from modern practices, simply because we feel it's sometimes worth trying a different approach.

TL;DR: если мы вам дадим ассерты, то вы будете крашить программу направо и налево вместо того, чтобы создавать и пробрасывать наверх конкретные ошибки.



Assert через panic

Но на деле получается, что разработчики не отказываются от ассертов, а используют для этого панику. Более того, и стандартная библиотека Go противоречит постулату из предыдущего шага.

А проблема анализа большого стектрейса и понимания, что к чему

Proper error reporting means that errors are direct and to the point, saving the programmer from interpreting a large crash trace.

решается тем, что обычно просто паникуют строкой с конкретным сообщением об ошибке:

```
// cmd/vendor/github.com/google/pprof/internal/driver/config.go:
    panic(fmt.Sprintf("unsupported config field type %v",
f.field.Type))

// cmd/compile/internal/syntax/printer.go:
    panic("expected non-empty []byte")

// cmd/api/goapi.go:
    panic("unknown object: " + obj.String())

// math/big/prime.go:
    panic("negative n for ProbablyPrime")


// и т.д.
```

*(*int)(nil) = 0

Продолжая тему противоречий между советами от создателей Go и их же кодом, вспомним, что когда мы разбирали внутреннее устройство паники, мы заметили,

что в рантайме можно увидеть ассерты, реализованные с помощью присвоения значения по `nil`-указателю:

```
// https://github.com/golang/go master
$ grep "(*int)(nil) = " -R .
./test/chan/select5.go:    case **(**int)(nil) = <-dummy:
./test/chan/select5.go:    case **(**int)(nil) = <-nilch:
./test/fixedbugs/issue17381.go: *(*int)(nil) = t[0]
./test/fixedbugs/issue32477.go: *(*int)(nil) = 0 // trigger a segv
./test/fixedbugs/issue27518b.go:      *(*int)(nil) = 0
./src/runtime/os_plan9.go: *(*int)(nil) = 0
./src/runtime/internal/atomic/atomic_arm.go:          *(*int)(nil) =
0 // crash on unaligned uint64
./src/runtime/internal/atomic/atomic_arm.go:          *(*int)(nil) =
0 // crash on unaligned uint64
./src/runtime/internal/atomic/atomic_arm.go:          *(*int)(nil) =
0 // crash on unaligned uint64
./src/runtime/internal/atomic/atomic_arm.go:          *(*int)(nil) =
0 // crash on unaligned uint64
./src/runtime/internal/atomic/atomic_arm.go:          *(*int)(nil) =
0 // crash on unaligned uint64
./src/runtime/panic.go:    *(*int)(nil) = 0 // not reached
./src/runtime/panic.go:    *(*int)(nil) = 0 // not reached
./src/runtime/panic.go:    *(*int)(nil) = 0 // not reached
./src/runtime/panic.go:    *(*int)(nil) = 0 // not reached

./src/runtime/stack_test.go:     *(*int)(nil) = 0
```

Интересный приём! Код

```
*(*int)(nil) = 0
```

аналогичен коду

```
var i *int

*i = 0
```

И при его выполнении мы получим

```
panic: runtime error: invalid memory address or nil pointer
dereference

[signal SIGSEGV: segmentation violation code=0x2 addr=0x0
pc=0x102d88544]
```

Стоит ли использовать подобные ухищрения в повседневной разработке?

Наше мнение – нет. Разве что выпендриться перед коллегами или почувствовать себя разработчиком компилятора. Но лучше отдайте предпочтение `panic` с понятным, конкретным сообщением или полноценной ошибке (`error`).

Задача "Assert"

Ссылка на заготовку.

Основной посыл данной задачи – это локально поиграться с билдтегами и обратить внимание на тесты.

Перед вами функция

```
func Assert(cond bool, msg string, args ...any)
```

Которая паникует сообщением `msg` при невыполнении условия `cond`.

Пример использования:

```
func ConsumeTopic(t string) error {
    assert.Assert(t != "", "kafka topic must be defined")

    // Consuming logic
    // ...

}
```

В начале урока мы обсуждали, что сишный макрос `assert()` можно "выключить" с помощью определения `NDEBUG`. Вот и наш `Assert` можно заменить на пустышку с помощью соответствующего билдтега:

```
func main() {
    var v *int
    assert.Assert(v != nil, "v must be initialized")
```

```
    fmt.Println("I'm OK")

}

$ cd 03-panic-concept/assert/testdata

$ go run main.go
panic: v must be initialized

goroutine 1 [running]:
.../tasks/03-panic-concept/assert.Assert(...)
        tasks/03-panic-concept/assert/assert.go:9
main.main()
        tasks/03-panic-concept/assert/testdata/main.go:11 +0x68
exit status 2

$ go run -tags NDEBUG main.go
I'm OK
```

"Заглушка" для функции находится в файле **assert_dummy.go**, а в **assert.go** находится паникующая функция – её-то и необходимо вам реализовать.

Static assertion через init()

Проверки, о которых мы говорили до этого, являются **динамическими**. Т.е. происходящими во время работы программы, а не во время компиляции. Соответственно, чтобы отловить ошибку с помощью таких ассертов, нам нужно или покрыть это место тестами или сделать так, чтобы работающая программа перешла в необходимое состояние.

Было бы здорово иметь **статические** проверки, которые не требуют работающей программы, и защищают нас от ошибок уже на этапе компиляции.

Например, в C++ есть static_assert (исходник примера):

```
#include <type_traits>

template <class T>
```

```
void swap(T& a, T& b) {
    static_assert(std::is_copy_constructible<T>::value, "swap
requires copying");
    auto c = b;
    b = a;
    a = c;
}

struct no_copy {
    no_copy(const no_copy&) = delete;
    no_copy() = default;
};

int main() {
    int a, b;
    swap(a, b);

    no_copy nc_a, nc_b;
    swap(nc_a, nc_b);

}
```

```
$ g++ -std=c++17 static_assert.cpp
static_assert.cpp:5:5: error: static_assert failed due to requirement
"swap requires copying"
    static_assert(std::is_copy_constructible<T>::value, "swap
requires copying");
    ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
static_assert.cpp:6:10: error: call to deleted constructor of
'no_copy'
    auto c = b;
    ^

2 errors generated.
```

А что может предложить Go?

К сожалению, в Go нет аналогичного механизма для проверок на этапе компиляции. Но мы можем осуществлять проверки на этапе импортирования пакета с помощью функции `init()` (spec#Package_initialization, Effective Go#init), по сути – на этапе запуска нашей программы (или тестов, импортирующих данных пакет).

Данный приём слегка освещён в Effective Go:

... real library functions should avoid panic. If the problem can be masked or worked around, it's always better to let things continue to run rather than taking down the whole program.

One possible counterexample is during initialization: if the library truly cannot set itself up, it might be reasonable to panic, so to speak.

```
var user = os.Getenv("USER")

func init() {
      if user == "" {
      panic("no value for $USER")
       }

}
```

И ещё несколько примеров из самого компилятора и его тестов:

```
// runtime/mgcstack.go
package runtime

func init() {
    if unsafe.Sizeof(stackWorkBuf{}) > unsafe.Sizeof(workbuf{}) {
        panic("stackWorkBuf too big")
    }
    if unsafe.Sizeof(stackObjectBuf{}) > unsafe.Sizeof(workbuf{}) {
        panic("stackObjectBuf too big")
    }

}

// time/zoneinfo_test.go
package time_test

func init() {
```

```
    if time.ZoneinfoForTesting() != nil {
        panic(fmt.Errorf("zoneinfo initialized before first
LoadLocation"))
    }

}

// test/convert4.go
package main

func init() {
    if &ss[0] != &s5[0] {
        panic("s5 conversion failed")
    }
    if &ss[0] != &s10[0] {
        panic("s5 conversion failed")
    }
    if ns0 != nil {
        panic("ns0 should be nil")
    }
    if zs0 == nil {
        panic("zs0 should not be nil")
    }

}

// syscall/dll_windows.go
package syscall

// We use this for computing the absolute path for system DLLs on
systems
// where SEARCH_SYSTEM32 is not available.
var systemDirectoryPrefix string

func init() {
    n := uint32(MAX_PATH)
    for {
        b := make([]uint16, n)
        l, e := getSystemDirectory(&b[0], n)
        if e != nil {
            panic("Unable to determine system directory: " +
e.Error())
        }
        if l <= n {
            systemDirectoryPrefix = UTF16ToString(b[:l]) + "\\"
```

```
            break
        }
        n = l
    }

}
```

Задача "Валидация конечного автомата"

Ссылка на заготовку.

Вы разработчик игровых движков и активно пользуетесь конечными автоматами:

```
type State string

const (
    StateInitial State = "initial"
    StateEnd     State = "end"
)

// FSM - finite-state machine.
type FSM map[State][]State
```

Вам необходимо реализовать функцию, позволяющую валидировать заданный автомат, например:

```
const (
    StateWaitForBet      State = "wait-for-bet"
    StateWaitForSpin     State = "wait-for-spin"
    StateWheelSpinning   State = "wheel-spinning"
    StateNumberReceived State = "number-received"
)

var RouletteFSM = FSM{
    StateInitial:         {StateWaitForBet},
    StateWaitForBet:      {StateWaitForSpin},
    StateWaitForSpin:     {StateWheelSpinning},
    StateWheelSpinning:   {StateNumberReceived},
    StateNumberReceived: {StateEnd},
}

func init() {
    if err := Validate(RouletteFSM); err != nil {
        panic(err)
    }
}
```

Функция `Validate` выглядит следующим образом:

```
// Validate проверяет, что конечный автомат удовлетворяет следующим условиям:
// - в нём имеется единственная начальная вершина (StateInitial);
// - в нём имеется как минимум одна конечная вершина (StateEnd);
// - в нём существуют пути из начальной вершины в конечную (несмотря на возможные циклы);
// - в нём отсутствуют тупиковые вершины (т.е. не позволяющие перейти из начальной вершины в конечную).

func Validate(f FSM) error
```

При этом считаем, что:

- Количество вершин не может больше 50.
- Циклы возможны, но невозможна ситуация, когда каждая вершина связана с каждой (образуя [полный граф](#)).

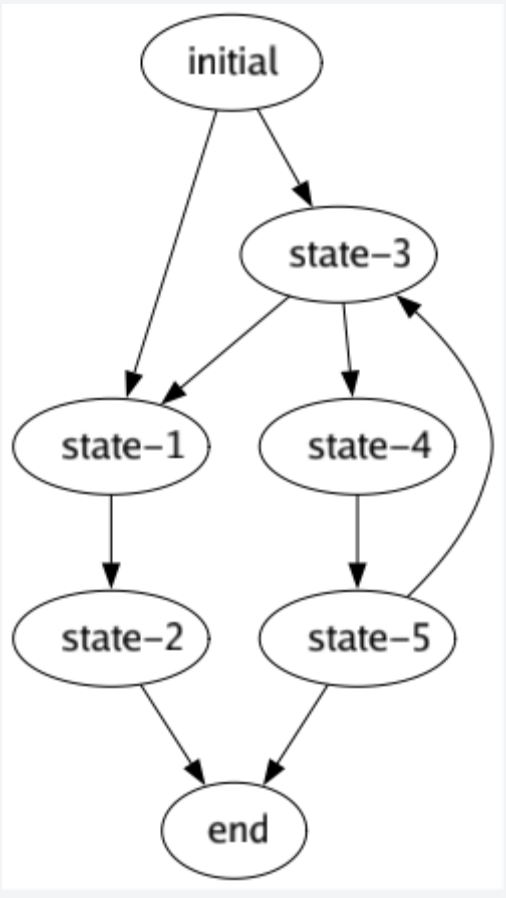
Больше подробностей см. в заготовке задачи и тестах.

*** Задание со звёздочкой**

В fsm/renderer лежит заготовка для рисовалки графа состояний с помощью **goccy/go-graphviz**. Вам предлагается довести её до рабочего состояния, что поможет отрисовывать тест-кейсы при необходимости (для большего понимания).

Пример конечного автомата и его визуализации:

```
const (
    state1 fsm.State = "state-1"
    state2 fsm.State = "state-2"
    state3 fsm.State = "state-3"
    state4 fsm.State = "state-4"
    state5 fsm.State = "state-5"
)

var fsmExample = fsm.FSM{
    fsm.StateInitial: {state1, state3},
    state1:           {state2},
    state2:           {fsm.StateEnd},
    state3:           {state1, state4},
    state4:           {state5},
    state5:           {state3, fsm.StateEnd},

}
```



panic("not implemented")

В Python есть исключение NotImplementedError, которое используется в качестве "бахающей" заглушки для нереализованных методов; методов абстрактного класса, нуждающихся в переопределении и т.д., например:

```
#
https://github.com/calston/tensor/blob/7c0c99708b5dbff97f3895f705e119
96b608549d/tensor/objects.py#L126

class Source(object):
    """Source parent class"""

    sync = False
    ssh = False

    def __init__(self, config, queueBack, tensor):
        self.config = config
        self.tensor = tensor
```

```
        self.running = False

    def createLog(self, type, data, evtime=None, hostname=None):
        """Creates an Event object from the Source configuration"""

        return Event(None, type, data, 0, self.ttl,
            hostname=hostname or self.hostname,
            evtime=evtime,
            tags=self.tags,
            type='log'
        )

    def get(self):
        raise NotImplementedError()

    def sshGet(self):
        raise NotImplementedError("This source does not implement SSH
remote checks")
```

В Go нечто подобное делают через выражение

```
panic("not implemented") // Текст может быть другим, но с тем же
смыслом.
```

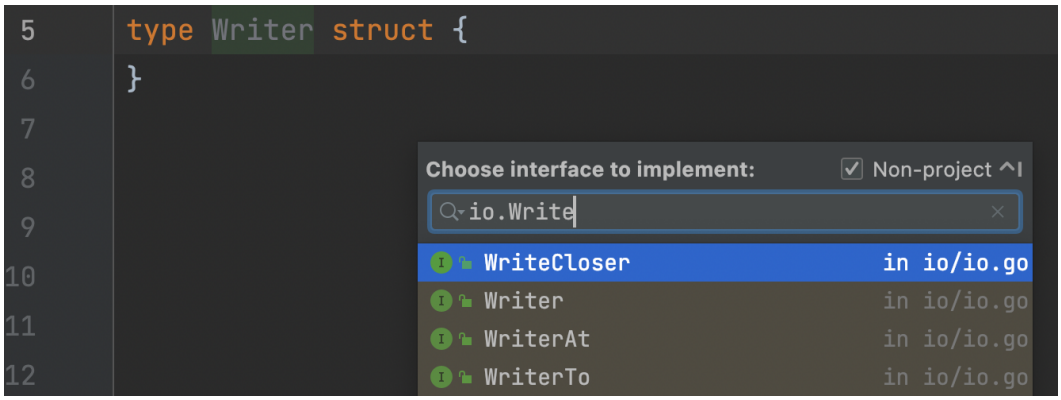
Помечая тем самым нереализованные методы и функции, вызов которых или не ожидается в принципе или вызов которых напомнит нам, что мы забыли их реализовать 🙂

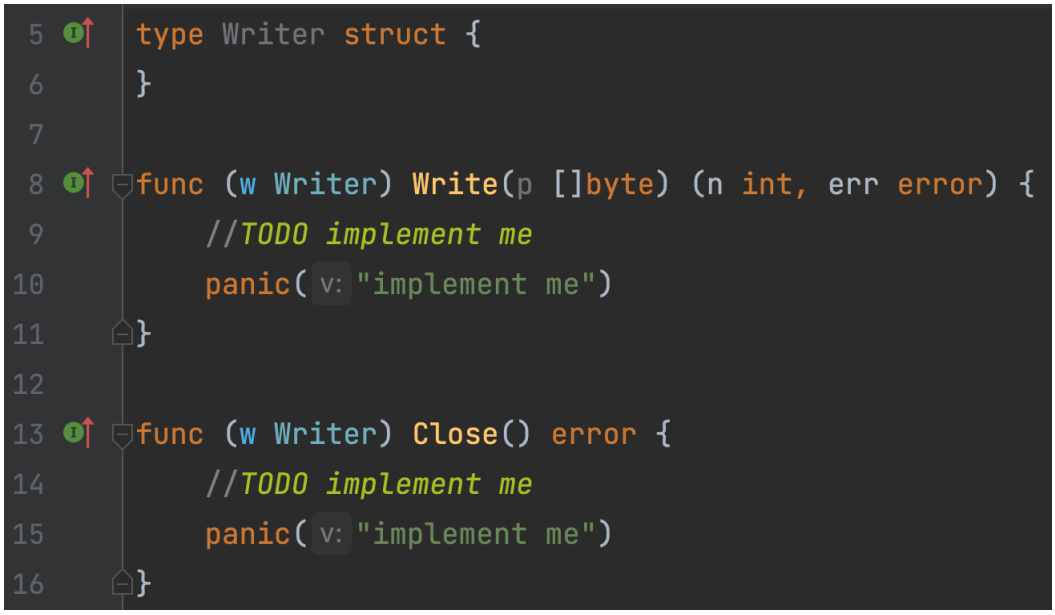
Примеры из компилятора:

```
// src/syscall/time_nofake.go
package syscall

const faketime = false

func faketimeWrite(fd int, p []byte) int {
    // This should never be called since faketime is false.
    panic("not implemented")
}
```

```
// src/runtime/os_js.go
package runtime

// Stubs so tests can link correctly. These should never be called.
func open(name *byte, mode, perm int32) int32         { panic("not
implemented") }
func closefd(fd int32) int32                          { panic("not
implemented") }

func read(fd int32, p unsafe.Pointer, n int32) int32 { panic("not
implemented") }

// src/math/arith_s390x.go
package math

const haveArchLog2 = false

func archLog2(x float64) float64 {
    panic("not implemented")

}
```

Когда мы просим GoLand сделать так, чтобы структура удовлетворяла заданному интерфейсу, он генерирует нам соответствующие методы и вставляет в них знакомые нам заглушки:





```
type Writer struct {
}

func (w Writer) Write(p []byte) (n int, err error) {
    //TODO implement me
    panic( v: "implement me")
}

func (w Writer) Close() error {
    //TODO implement me
    panic( v: "implement me")
}
```

В любом случае это довольно опасная штука и, если есть подозрение, что нереализованный метод может проскочить в конечную версию приложения (а не быть таковым только на стадии разработки и отладки), то лучше закрыть его ошибкой:

```
import "errors"

var ErrNotImplemented = errors.New("not implemented")

type Writer struct {
}

func (w Writer) Write(p []byte) (n int, err error) {
    return 0, ErrNotImplemented
}

func (w Writer) Close() error {
    return ErrNotImplemented

}
```

P.S. Пишите в комментариях известные вам аналогичные подходы в других языках.

Выводы

В этом уроке мы освежили в памяти понятие **assertion**, а также узнали, как реализовывать динамические и статические ассерты в Go с помощью уловок компилятора и паники.

В следующем уроке мы познакомимся с лучшими практиками нейминга паникующих функций.