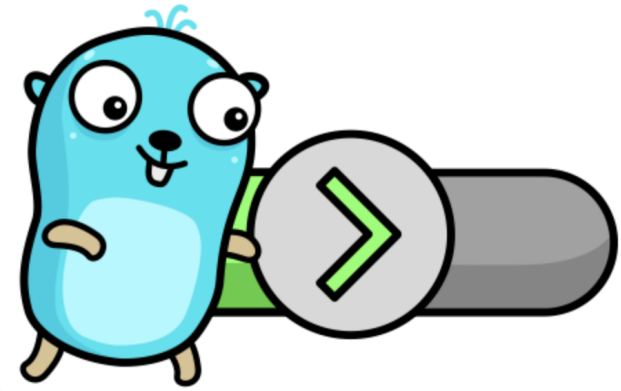
Опасный os.Exit

В этом уроке мы поговорим о функции, вызов которой часто используют вместо паникования.

Какие проблемы это за собой несёт? Сейчас узнаем!



os.Exit

os.Exit моментально завершает текущий процесс (не отдельно взятую горутину!) с заданным кодом возврата.

Эквивалентен вызову системного вызова **exit**:

```
// os/proc.go
package os

// Exit causes the current program to exit with the given status
code.
// Conventionally, code zero indicates success, non-zero an error.
// The program terminates immediately; deferred functions are not
run.
//
// For portability, the status code should be in the range [0, 125].
func Exit(code int) {
    if code == 0 {
        if testlog.PanicOnExit0() {
            // We were told to panic on calls to os.Exit(0).
            // This is used to fail tests that make an early
            // unexpected call to os.Exit(0).
```

```
            panic("unexpected call to os.Exit(0) during test")
        }

        // Give race detector a chance to fail the program.
        // Racy programs do not have the right to finish
successfully.
        runtime_beforeExit()
    }
    syscall.Exit(code)

}
```

Вы наверняка встречались с системным вызовом `exit`, когда программировали на Си  :

```
errno = 0;
int result = some_lib_func();
if (errno != 0) {
    perror("cannot do some operation");
    exit(EXIT_FAILURE);

}
```

Комментарий к `os.Exit` отражает практику POSIX-совместимых систем считать нулевой код возврата признаком успешного завершения программы, а ненулевой – признаком случившейся ошибки:

```
// Conventionally, code zero indicates success, non-zero an error.
```

Исторически сложилось, что

- код возврата 0 означает успешное завершение программы;
- код возврата 1 означает, что программа завершилась с какой-то ошибкой;
- код возврата 2 означает, что программе неверно переданы аргументы командной строки.

Хотя существуют и другие классификации.

Если глянуть примеры вызовов `os.Exit(1)` и `os.Exit(2)` в сорцах гошки, то в целом они удовлетворяют утверждениям выше:

```
// https://github.com/golang/go master
$ grep --exclude "*test*" "os.Exit(1)" -R -B3 -A2 .

./cmd/trace/main.go-
./cmd/trace/main.go-  func dief(msg string, args ...interface{}) {
./cmd/trace/main.go-      fmt.Fprintf(os.Stderr, msg, args...)
./cmd/trace/main.go:      os.Exit(1)
./cmd/trace/main.go-  }
./cmd/trace/main.go-

./cmd/api/goapi.go-  fail := false
./cmd/api/goapi.go-  defer func() {
./cmd/api/goapi.go-      if fail {
./cmd/api/goapi.go:          os.Exit(1)
./cmd/api/goapi.go-      }
./cmd/api/goapi.go-  }()

./os/exec/read3.go-  bs, err := io.ReadAll(fd3)
./os/exec/read3.go-  if err != nil {
./os/exec/read3.go-      fmt.Printf("ReadAll from fd 3: %v\n", err)
./os/exec/read3.go:      os.Exit(1)
./os/exec/read3.go-  }
./os/exec/read3.go-

./syscall/mksyscall_windows.go-  cmd.Env = cmdEnv
./syscall/mksyscall_windows.go-  err := cmd.Run()
./syscall/mksyscall_windows.go-  if err != nil {
./syscall/mksyscall_windows.go:      os.Exit(1)
./syscall/mksyscall_windows.go-  }



// https://github.com/golang/go master
$ grep --exclude "*test*" "os.Exit(2)" -R -B3 -A2 .

./cmd/objdump/main.go- func usage() {
./cmd/objdump/main.go-   fmt.Fprintf(os.Stderr, "usage: go tool
objdump [-S] [-gnu] [-s symregexp] binary [start end]\n\n")
./cmd/objdump/main.go-   flag.PrintDefaults()
./cmd/objdump/main.go:   os.Exit(2)
```

```
./cmd/objdump/main.go- }
./cmd/objdump/main.go-

./cmd/asm/internal/flags/flags.go-    fmt.Fprintf(os.Stderr, "usage:
asm [options] file.s ...\n")
./cmd/asm/internal/flags/flags.go-    fmt.Fprintf(os.Stderr,
"Flags:\n")
./cmd/asm/internal/flags/flags.go-    flag.PrintDefaults()
./cmd/asm/internal/flags/flags.go:    os.Exit(2)
./cmd/asm/internal/flags/flags.go- }
./cmd/asm/internal/flags/flags.go-

./cmd/asm/main.go-   switch *flags.Spectre {
./cmd/asm/main.go-   default:
./cmd/asm/main.go-       log.Printf("unknown setting -spectre=%s",
*flags.Spectre)
./cmd/asm/main.go:       os.Exit(2)
./cmd/asm/main.go-   case "":
./cmd/asm/main.go-       // nothing

./cmd/go/internal/modfetch/codehost/shell.go-
./cmd/go/internal/modfetch/codehost/shell.go- func usage() {
./cmd/go/internal/modfetch/codehost/shell.go-
fmt.Fprintf(os.Stderr, "usage: go run shell.go vcs remote\n")
./cmd/go/internal/modfetch/codehost/shell.go:    os.Exit(2)
./cmd/go/internal/modfetch/codehost/shell.go- }
./cmd/go/internal/modfetch/codehost/shell.go-

./cmd/trace/main.go- func main() {
./cmd/trace/main.go-     flag.Usage = func() {
./cmd/trace/main.go-         fmt.Fprintln(os.Stderr, usageMessage)
./cmd/trace/main.go:         os.Exit(2)
./cmd/trace/main.go-     }
./cmd/trace/main.go-     flag.Parse()

./cmd/cgo/main.go- func usage() {
./cmd/cgo/main.go-    fmt.Fprint(os.Stderr, "usage: cgo -- [compiler
options] file.go ...\n")
./cmd/cgo/main.go-    flag.PrintDefaults()
./cmd/cgo/main.go:    os.Exit(2)
./cmd/cgo/main.go- }
```

Интересный пример в пакете flag: если произошла ошибка парсинга аргументов командной строки, то процесс завершается с кодом `2`; но если это был запрос помощи (`-h`/`-help`), то получим `0`:

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

func (f *FlagSet) Parse(arguments []string) error {
    // ...
        switch f.errorHandling {
        // ...
        case ExitOnError:
            if err == ErrHelp {
                os.Exit(0)
            }
            os.Exit(2)
    // ...

}
```

Тест "Для чего процессу нужен код возврата?"

Выберите один вариант из списка

- ☐ Это неиспользуемый исторически сложившийся рудимент.
- ☐ Для того, чтобы родительский процесс понимал, как отработал запущенный им дочерний процесс, и можно было построить логику поверх этого.
- ☐ Для идентификации процесса в таблице процессов.

Подводные камни os.Exit

Снова обратим внимание на комментарий к функции `os.Exit` (удивительно, как полезно читать документацию...):

```
// Exit causes the current program to exit with the given status
code.
```

```
// ...

// The program terminates immediately; deferred functions are not
run.
```

The program terminates immediately

T.e. не происходит никакого завершения запущенных горутин, ожидания закрытия ресурсов и пр. Процесс завершается as is, игнорируя наши хотелки.

Deferred functions are not run

Вторая неприятная особенность – отложенные функции запущены **не будут**.

```
// https://goplay.tools/snippet/3FqyJJB16qI
package main

import (
    "fmt"
    "os"
    "time"
)

// Мы запускаем n спящих горутин и убиваем процесс во время их сна.

func main() {
    defer fmt.Println("EXIT FROM main") // <- Это мы не увидим.

    const n = 3
    for i := 0; i < n; i++ {
        go worker(i)
    }

    go killer(1)

    // Если убрать блокировку, то это будет эквивалентно
    // вызову os.Exit(0) в конце функции main и горутины
    // выше даже не успеют запуститься.
    select {}
}

func worker(i int) {
    fmt.Printf("worker %d: start\n", i)
    time.Sleep(10 * time.Second)
```

```
    fmt.Printf("worker %d: stop\n", i) // <- Это мы не увидим.
}

func killer(code int) {
    defer fmt.Printf("EXIT FROM killer") // <- Это мы не увидим.

    time.Sleep(time.Second)
    fmt.Println("DEAD")
    os.Exit(code)
}

/*
worker 1: start
worker 2: start
worker 0: start
DEAD

Process finished with the exit code 1

*/
```

Следствия особенностей работы os.Exit

Самая частая ошибка, связанная с использованием `os.Exit` – это **утечка контекста и ресурсов** при неудачном старте приложения или при его *жёстком* завершении (ungraceful shutdown).

Представим, что мы

- конструируем различные компоненты нашего приложения;
- если какой-то конструктор вернул ошибку, то выходим с помощью `log.Fatal` / `os.Exit`;
- запускаем различные фоновые задачи от корневого контекста;
- стартуем приложение, при ошибке старта так же выходим с помощью `log.Fatal` / `os.Exit`.

(исходник примера, для простоты чтения опущены проверки ошибок от отложенных функций)

```
16  func main() {
17      ctx, cancel := signal.NotifyContext(context.Background(),
os.Interrupt)
18      defer cancel()
19
20      // Запускаем фоновые задачи от корневого контекста.
21      wrk, err := worker.New()
22      mustNil(err, "build worker")                    // <- Утечка
defer'ов выше.
23      defer wrk.Wait()
24      go wrk.Run(ctx)
25
26      // Конструируем приложение.
27      cache, err := filecache.New()
28      mustNil(err, "build cache")                     // <- Утечка
defer'ов выше.
29      defer cache.CleanUp()
30
31      storage, err := store.New()
32      mustNil(err, "build storage")                   // <- Утечка
defer'ов выше.
33      defer storage.Close()
34
35      service, err := app.New(storage, cache)
36      mustNil(err, "build app")                       // <- Утечка
defer'ов выше.
37
38      // Запускаем приложение.
39      err = service.Start(ctx)
40      mustNil(err, "start service")
41  }
42
43  func mustNil(err error, msg string) {
44      if err != nil {
45          log.Fatal(msg + ": " + err.Error())
46      }
47  }
48

/*
worker run
2077/12/12 18:37:29 start service: unexpected error

*/
```

Напомним, что вызов log.Fatal идентичен завершению процесса с предварительным выводом сообщения на экран:

```
// log/log.go
package log

var std = New(os.Stderr, "", LstdFlags)

// New creates a new Logger.
func New(out io.Writer, prefix string, flag int) *Logger {
    return &Logger{out: out, prefix: prefix, flag: flag}
}

// Fatal is equivalent to Print() followed by a call to os.Exit(1).
func Fatal(v ...interface{}) {
    std.Output(2, fmt.Sprint(v...))
    os.Exit(1)
}
```

Как следствие при `log.Fatal` не будет вызван ни один из наших `defer`:

```
18  defer cancel()          // Утечка контекста.
23  defer wrk.Wait()        // Утечка горутины.
29  defer cache.CleanUp()   // Утечка ресурсов.

33  defer storage.Close()   // Утечка ресурсов.
```

Особенно неприятно, когда приложение состоит из множества компонентов, имеет долгое время запуска и в момент запуска его прерывают по какой-то причине сигналом извне – и мы вместо отмены контекста и graceful shutdown приложения получаем undefined behaviour.

Как починить?

1) Заменить `log.Fatal` на `log.Panic` (исходник примера):

```
// log/log.go
package log

// Panic is equivalent to l.Print() followed by a call to panic().
```

```
func (l *Logger) Panic(v ...interface{}) {
    s := fmt.Sprint(v...)
    l.Output(2, s)
    panic(s)

}

package main

// ...

func mustNil(err error, msg string) {
    if err != nil {
        log.Panic(msg + ": " + err.Error())    // <- Раньше был
log.Fatal!
    }
}

/*
worker run
2077/12/12 21:38:09 start service: unexpected error
storage connection closed
filecache cleaned out
worker done
panic: start service: unexpected error

*/
```

Что приведёт к завершению процесса через панику, и все отложенные функции **будут выполнены**.

---

2) Вынести реализацию `main` в отдельную функцию и уже на самом верхнем уровне решать, что делать (исходник примера):

```
func main() {
    if err := mainImpl(); err != nil {
        log.Fatal(err)
    }
}

func mainImpl() error {
    ctx, cancel := signal.NotifyContext(context.Background(),
os.Interrupt)
```

```
    defer cancel()

    // ...

    // Запускаем приложение.
    return service.Start(ctx)
}

/*
storage connection closed
filecache cleaned out
worker done
worker run
2077/12/12 19:50:44 unexpected error

*/
```

Иногда можно встретить и такой вариант (но он видится нам менее предпочтительным, если нет нужды в многообразии кодов возврата):

```
func main() {
    os.Exit(mainImpl())
}

func mainImpl() int {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    // ...
    if err := service.Start(ctx); err != nil {
        log.Println(err)
        return 1
    }
    return 0
}

// И сюда же различные вариации:
// - возвращать в mainImpl и код и ошибку
//     return 1, err                        // Нарушаем конвенцию о
неиспользовании
//                                          // возвращаемых значений
при ошибке.
// - возвращать в mainImpl что-то вроде

//     return newErrorWithCode(err, 1)      // Уже лучше.
```

В целом наличие единственной точки выхода из программы является хорошей практикой и уходит корнями в Си.

Данный подход избавляет нас от ряда проблем, делая код более безопасным и чистым.

Тест "Выберите подходящие для завершения процесса места"

Выберите все подходящие ответы из списка

- ☐ Внутренняя доменная логика
- ☐ package main
- ☐ Экспортируемая вашей библиотекой функция
- ☐ Обработка ошибки получения данных из хранилища
- ☐ Обработка ошибки ответа внешнего стороннего API

Тест "go-critic"

Какой из checker'ов, входящих в go-critic, обнаруживает, что вы вызываете `os.Exit` / `log.Fatal` внутри функций, в которых содержится `defer`?

Необходимо написать имя checker'а в соответствии с документацией go-critic.

Напишите текст

_____

Тест "Опечатка"

Исходник примера.

При очередном запуске тестов обнаружился плавающий кейс, который к тому же почему-то приводил к завершению всего тестового процесса (остальная сотня тестов оставалась без внимания):

```
$ cd tasks/03-panic-concept/fatal-typo
$ go test -v .
=== RUN   TestSum
=== RUN   TestSum/#00
=== RUN   TestSum/#01
2077/12/12 23:44:49 Sum(2, 2) != 5
FAIL    tasks/03-panic-concept/fatal-typo    0.098s
FAIL
```

Вам необходимо исправить одну единственную строчку, чтобы всё заработало как нужно и несмотря на падение одного теста, остальные тесты были запущены (и не были пропущены):

```
$ go test -v .
=== RUN   TestSum
=== RUN   TestSum/#00
=== RUN   TestSum/#01
    ops_test.go:21: Sum(2, 2) != 5
=== RUN   TestSum/#02
--- FAIL: TestSum (0.00s)
    --- PASS: TestSum/#00 (0.00s)
    --- FAIL: TestSum/#01 (0.00s)
    --- PASS: TestSum/#02 (0.00s)
=== RUN   TestDiff
=== RUN   TestDiff/#00
=== RUN   TestDiff/#01
=== RUN   TestDiff/#02
--- PASS: TestDiff (0.00s)
    --- PASS: TestDiff/#00 (0.00s)
    --- PASS: TestDiff/#01 (0.00s)
    --- PASS: TestDiff/#02 (0.00s)
FAIL
FAIL    tasks/03-panic-concept/fatal-typo    0.274s
FAIL
```

В качестве ответа нужно ввести **исправленную строку полностью**.

Напишите текст

_____

Выводы

Теперь вы знаете, что ответить, когда вас спросят, зачем вы паникуете в `main` вместо того, чтобы использовать привычный `log.Fatal` / `os.Exit`.

- Старайтесь выходить из вашего приложения в единственной точке.
- Помните, что при завершении процесса отложенные функции **не выполняются**.
- Помните про паттерн выделения логики функции `main` в отдельную функцию.