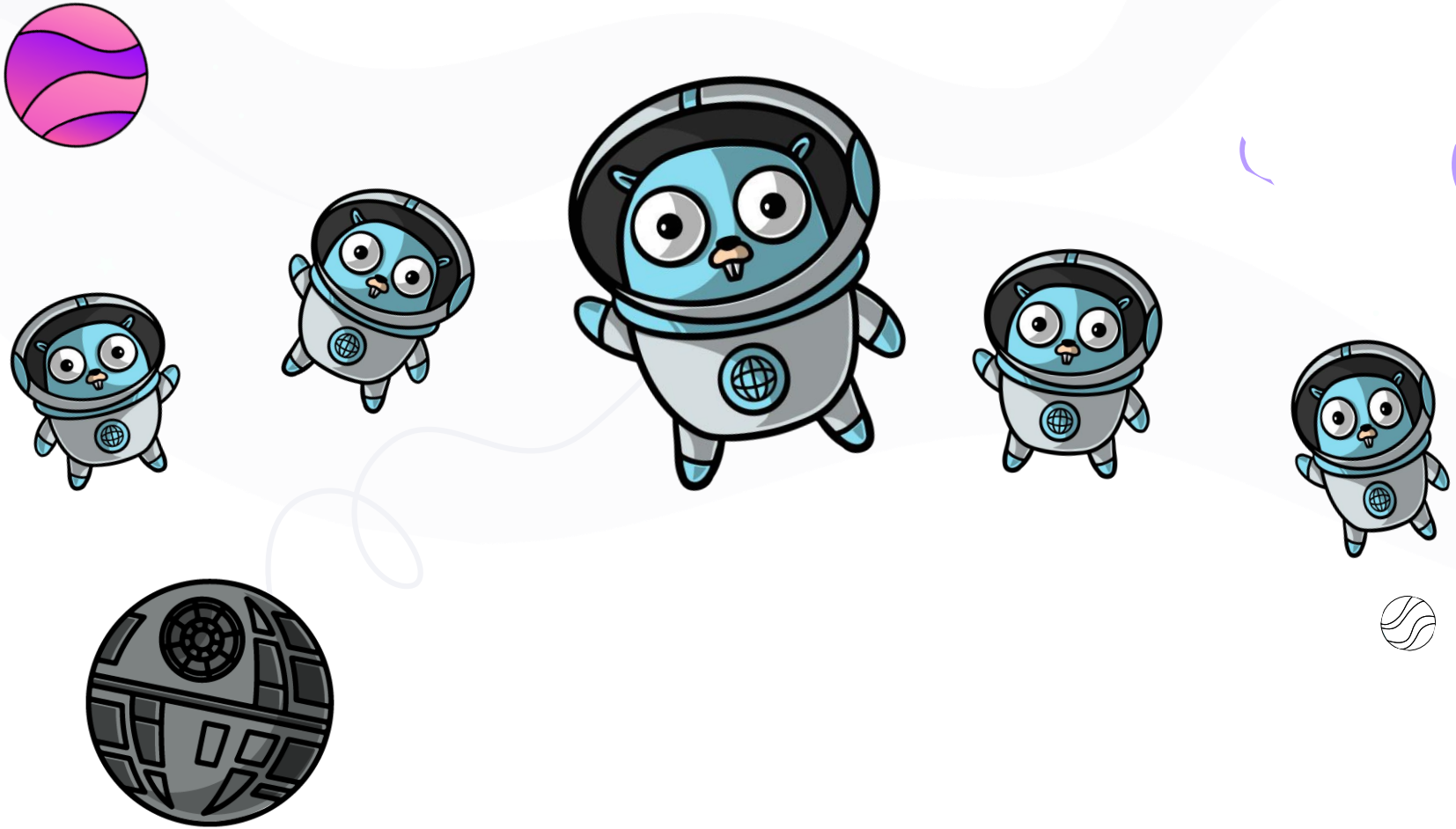


Agenda

- Software Development Lifecycle
- Requirements & Testing
- Unit tests in GO
  - Creating tests
  - Parallel run
  - Table testing
  - Build tags
  - Coverage
- Race conditions
- Benchmarking
- Fuzzing
- Profiling with pprof
- Common environment
- Mocks
- “testify” package

Software Development Lifecycle

Software Development Lifecycle

Planning & Analysis

The main goal is to collect business requirements. Done by project managers, analysts, stakeholders

Design

The requirements from the previous stage are translated into technical language. Usually done by solution architects, lead/senior engineers. The design phase generates also the requirements to the software/hardware

Implementation

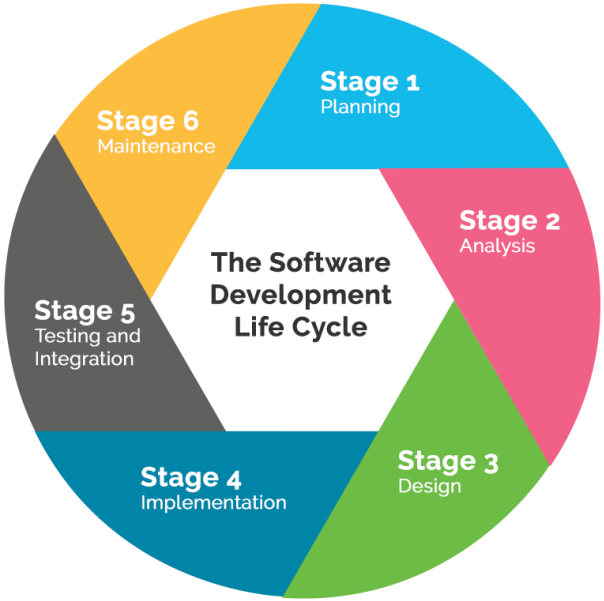
In this phase the code is produced so it is the focus for the developer/software engineer. Also includes unit tests implementation.

Testing

Verification the requirements are met. Performed by Devs & QA specialists

Delivery & Maintenance

Responsibility of DevOps, support engineers, delivery managers, dev team



Requirements & Testing

Requirements

Functional - are product features or functions that developers must implement to enable users to accomplish their tasks. So, it's important to make them clear both for the development team and the stakeholders. Generally, functional requirements describe system behavior under specific conditions.

write a function which takes integer as an input and returns fib number. If input > 100 return an error

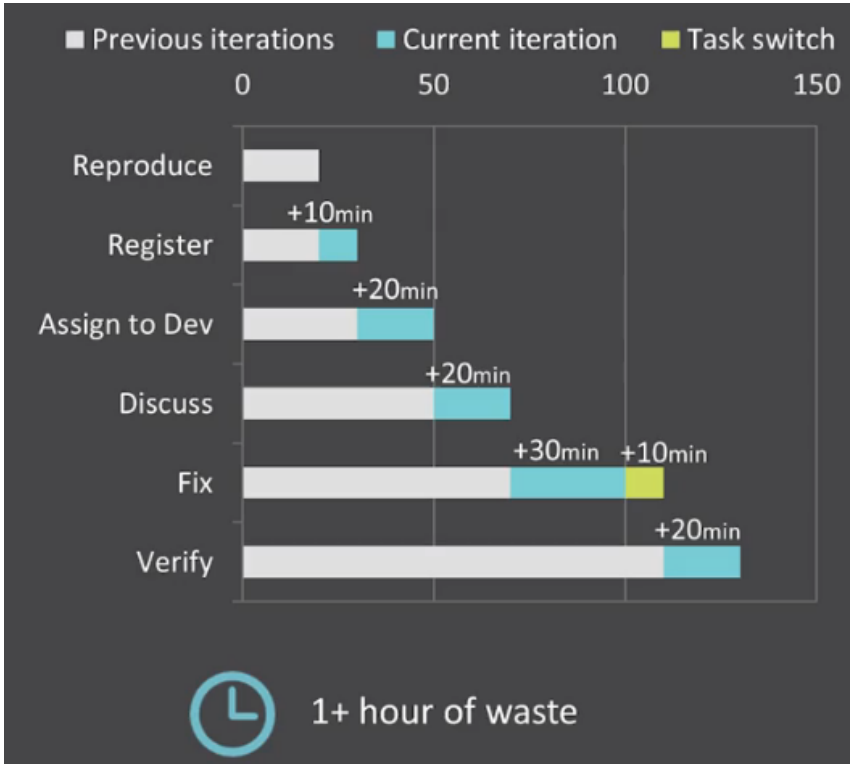
Non-functional - this type of requirements is also known as the system's quality attributes: Usability, Security, Reliability, Performance, Availability, Scalability

fib() function should work 30ns or less

Testing types

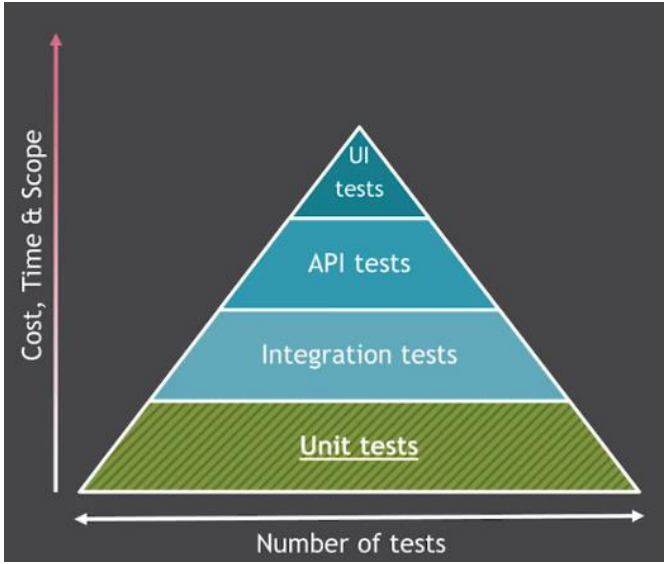
Functional Testing	Non-functional Testing
- Unit Testing - Integration/API Testing - System Testing - Sanity Testing - Smoke Testing - Interface Testing - Regression Testing - Beta/Acceptance Testing	*- Performance Testing/Benchmarks* - Load Testing - Stress Testing - Volume Testing - Security Testing - Compatibility Testing - Install Testing - Recovery Testing - Reliability Testing - Usability Testing - Compliance Testing - Localization Testing

Cost of bug


Previous iterations
Current iteration
Task switch
0
50
100
150
Reproduce
Register
+10min
Assign to Dev
+20min
Discuss
+20min
Fix
+30min
+10min
Verify
+20min
1+ hour of waste

Early Defect Detection & Testing Pyramid

- Reduces Fix and Remediation Cost
- Reduces Overall Delivery Time Spend
- Increased Developer and QA Staff Productivity
- Reduces Business Risk Due to Outages
- Improves Application Security and Overall Code Quality



Unit Tests Principles / F.I.R.S.T. Rules

Fast – tests should be quick. They should run promptly. When tests run slowly, you don't want to run them frequently. And if you don't run them frequently, you won't find problems early enough to fix them easily.

Independent (or Isolated) – tests should not depend on each other. One test should not set the conditions for the next test. You should be able to run tests in any order you like. When tests depend on each other, the first one to fail causes a cascade of downstream failures, making diagnosis difficult and hiding downstream defects.

Repeatable – tests should be repeatable in any environment: in the production environment, in the QA environment, and even on your laptop while heading home by subway without a network connection. Test results must be the same every time and at every location.

Self-validating – tests should have a Boolean output. They either pass or fail. You should not have to read through a log file or compare different files to see whether the tests pass. If the tests aren't self-validating, then failure can become subjective and running the tests can require a long manual evaluation.

Timely - tests should be written at the proper time, immediately before production code. Testing post-facto requires developers to refactor working code and make additional effort to have tests fulfilling these FIRST principles.

GoLang Community

Unit tests in GO

Go bult-in testing tool

Run all tests	go test -v ./...
Run specific test	go test -v -run "^TestNameToEmail_ErrNameIsEmpty$"
Search for race conditions	go test --race go run --race (also OK)
Test specific file with imports	go test -v email_test.go email.go errors.go
Get some help	go help test

Built-in testing package

Go offers the package, containing widely used testing tools. Four most important are

- T - Does all the job around testing: Run, Fail and between these two.
- B - Benchmarking. Runs code in cycles, calculates execution time.
- M - Main entrypoint, prepare the environment.
- F - Fuzz for preparing ang run fuzzing tests.

Creating tests

Test file in go **MUST** follow rules:

- Name ends with ***_test.go***
- Package name is same as other files OR same with ***_test*** suffix
- functions in test file MUST start with **Test** prefix and accept one param ****testing.T***

Test example

CODE

```
package main

import (
   "errors"
   "testing"
)

func Division(a, b int64) (int64, error) {
   if b == 0 {
      return 0, errors.New("division by zero")
   }
   return a / b, nil
}

func TestDivisionOK(t *testing.T) {
   var a int64 = 4;    var b int64 = 2;    var expect int64 = 2
   r, err := Division(a, b)
   if err != nil {
      t.Errorf("Error returned while not awaited: %s", err.Error())
   }
   if r != expect {
      t.Errorf("Expected: %d, got %d", expect, r)
   }
}
```

RESULT

```
=== RUN   TestDivisionOK
--- PASS: TestDivisionOK (0.00s)
PASS
```

https://play.golang.org/p/Xo4WCYjkgeF

*Note: it's a playground restriction to put test funcs in the same file, it's not how gophers usually do it.

Parallel run

- When running tests go compiler creates 2 queues waiting for each test to finish.
- One queue waits for ALL tests to finish and quit testing process
- Another queue waits for EVERY test to finish before continue to a next one.
- This second queue can be skipped by calling t.Parallel() from any test func.

```
func sum(a, b int) int {
    return a + b
}

func Test_1(t *testing.T) {
    t.Parallel()
    fmt.Println(sum(1, 0))
}

func Test_2(t *testing.T) {
    t.Parallel()
    fmt.Println(sum(1, 1))
}
```

https://play.golang.org/p/2w1vfcqmQNT

It actually doesn't make any use, just an illustration

Table testing

Table testing is an automated testing methodology. It's not a specific one for Go exclusively.

- TT expects code to have prepared testing table(slice) with **input** data and **expected** outcome
- Cycling through rows of the table code is executed for every case.
- It's a fast way to run different cases in one function.
- Plus, we can run it even faster.

Table test example

CODE

```
var ErrDivisionByZero = errors.New("division by zero")

func Test_Division(t *testing.T) {
  tData := []struct {
    A       int64
    B       int64
    Expected int64
    Err     error
  }{
    {A: 4, B: 2, Expected: 2, Err: nil},
    {A: 4, B: 0, Expected: 0, Err: ErrDivisionByZero},
  }
   for k, v := range tData {
      got, err := Division(v.A, v.B)
      if err != nil && err != v.Err {
         t.Errorf("[%d] error happend while not expected: %s", k, err.Error())
      }
      if got != v.Expected {
         t.Errorf("[%d] expected: %d, got %d", k, v.Expected, got)
      }
   }
}
func Division(a, b int64) (int64, error) {
   if b == 0 {
      return 0, ErrDivisionByZero
   }
   return a / b, nil
}
```

RESULT

```
=== RUN   Test_Division
--- PASS: Test_Division (0.00s)
PASS
```

https://play.golang.org/p/2s2g2sWITrs

*Note: it's a playground restriction to put test funcs in the same file, it's not how gophers usually do it.

Making table test example parallel

CODE

```
var ErrDivisionByZero = errors.New("division by zero")

func Test_Division(t *testing.T) {
   t.Parallel()
   tData := []struct {
      A       int64
      B       int64
      Expected int64
      Err     error
   }{
      {A: 4, B: 2, Expected: 2, Err: nil},
      {A: 4, B: 0, Expected: 0, Err: ErrDivisionByZero},
   }
   for k, tcase := range tData {
      v := tcase
      t.Run(fmt.Sprintf("test_%d", k), func(t *testing.T) {
         t.Parallel()
         got, err := Division(v.A, v.B)
         if err != nil && err != v.Err{
            t.Errorf("[%d] error happend while not expected: %s", k, err.Error())
         }
         if got != v.Expected {
            t.Errorf("[%d] expected: %d, got %d", k, v.Expected, got)
         }
      })
   }
}
```

RESULT

```
=== RUN   Test_Division
=== PAUSE Test_Division
=== CONT  Test_Division
=== RUN   Test_Division/test_0
=== PAUSE Test_Division/test_0
=== RUN   Test_Division/test_1
=== PAUSE Test_Division/test_1
=== CONT  Test_Division/test_0
=== CONT  Test_Division/test_1
--- PASS: Test_Division (0.00s)
    --- PASS: Test_Division/test_0 (0.00s)
    --- PASS: Test_Division/test_1 (0.00s)
PASS
```

https://play.golang.org/p/jfBMnYnYTbr

Making table test example parallel

CODE

```
var ErrDivisionByZero = errors.New("division by zero")

func Test_Division(t *testing.T) {
   t.Parallel()
   tData := map[string]struct {
      A       int64
      B       int64
      Expected int64
      Err     error
   }{
      "success":             {A: 4, B: 2, Expected: 2, Err: nil},
      "division by zero": {A: 4, B: 0, Expected: 0, Err: ErrDivisionByZero},    }
   for name, tcase := range tData {
      v := tcase
      t.Run(name, func(t *testing.T) {
         t.Parallel()
         got, err := Division(v.A, v.B)
         if err != nil && err != v.Err{
            t.Errorf("[%s] error happend while not expected: %s", name, err.Error())
         }
         if got != v.Expected {
            t.Errorf("[%s] expected: %d, got %d", name, v.Expected, got)
         }
      })
   }
}
```

RESULT

```
=== RUN   Test_Division
=== PAUSE Test_Division
=== CONT  Test_Division
=== RUN   Test_Division/success
=== PAUSE Test_Division/success
=== RUN   Test_Division/division_by_zero
=== PAUSE Test_Division/division_by_zero
=== CONT  Test_Division/success
=== CONT  Test_Division/division_by_zero
--- PASS: Test_Division (0.00s)
    --- PASS: Test_Division/success (0.00s)
    --- PASS: Test_Division/division_by_zero
(0.00s)
PASS
```

https://go.dev/play/p/HNnhb4kkw9T

Parallel run trap

- Calling **t.Parallel** from test func sends a signal to parent test func and says "don't wait for me"
- As usual with parallel programming it's up to developer to get rid of race conditions.
- In terms of table testing it's a bit different problem due to scopes.
- It actually cycles through the test cases (row) and to avoid run one test case for each **t.Run** call
- https://play.golang.org/p/T05scOCjiIe
- Creating an internal copy of **tCase** is vital here.

Build tags

- Build tag is not testing-specific feature, while widely used in testing
- Build tags allow create separated build scenarios: including files or ignore them. Like it's possible to run mocked test separated from tests on real data, or integration test from unit testing.
- Tags can be combined in logical way
- Test files with any tag set would be ignored, if called without any tag and vice-versa, not tagged files will be ignored if tag is requested

Defining	// +build tag_name
Multiply AND	// +build tag_one,tag_two
Multiply OR	// +build tag_one tag_two
Exclude	// +build !tag_name
Run	go test -tags tag_name
Run multy	go test -tags "tag_one tag_two"

Coverage

HIGHLIGHT GOES HERE

- We are not arguing if test coverage is important for business purposes, but focusing on tools we have in go.
- go test --cover ./... | grep coverage
- All parameters for calling go test are applicable here too

OUTPUT

```
ok      gtest_example/app/internal/db   (cached)
    coverage: 9.1% of statements
ok      gtest_example/app/internal/grabber      (c
ached)      coverage: 86.7% of statements
ok      gtest_example/app/internal/notification
(cached)      coverage: 100.0% of statements
ok      gtest_example/app/utils/naming  (cached)
      coverage: 80.0% of statements
```

Coverage

HIGHLIGHT GOES HERE

- go test -v -coverprofile=/tmp/c.out
- go tool cover -html=/tmp/c.out

OUTPUT

gtest_example/app/utils/naming/email.go (66.7%) not tracked not covered covered

```
func validateName(name string) error {
        if len(name) == 0 {
                return ErrNameIsEmpty
        }
        return nil
}

func emalifyName(s string) string {
        s = strings.ToLower(s)
        pattern := regexp.MustCompile("([^a-z])")
        return pattern.ReplaceAllString(s, ".")
}

func addDomain(emlName, domain string) (string, error) {
        parts := strings.Split(emlName, "@")
        if len(parts) > 2 {
                return "", ErrInvalidNameForEmail
        }
        if len(parts) == 2 {
                if parts[1] == domain {
                        return emlName, nil
                }
                return "", ErrAlreadyEmailWrongDomain
        }
        return fmt.Sprintf("%s@%s", emlName, domain), nil
}

func NameToEmail(name string) (eml string, err error) {
        if err := validateName(name); err != nil {
                return "", errors.Wrap(err, "can't convert name to email")
        }
        emlName := emalifyName(name)
        if eml, err = addDomain(emlName, "acme.com"); err != nil {
                return "", errors.Wrap(err, "can't convert name to email")
        }
        return eml, nil
}
```

Race conditions

Race conditions

A **race condition** is the behavior of a software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when one or more of the possible behaviors is undesirable. Race conditions can occur especially in multithreaded or distributed software programs.

Race detector in GO

Data races are among the most common and hardest to debug types of bugs in concurrent systems. A data race occurs when two goroutines access the same variable concurrently and at least one of the accesses is a write.

```
$ go test -race mypkg        // to test the package
$ go run -race mysrc.go      // to run the source file
$ go build -race mycmd       // to build the command
$ go install -race mypkg     // to install the package
```

Benchmarking

What is benchmark file

Benchmark file in go **MUST** follow rules:
Very similar to test functions

- Name ends with **_test.go**
- Package name is same as other files OR same with **_test** suffix
- functions in test file MUST start with **Benchmark** prefix and accept one param ***testing.B**

What is benchmark file

Benchmark runs as test with additional flag *–bench=*

- It is possible to set limits on benchmarks by times(x) and seconds(s):
- go test -bench=. -benchtime=10x
- go test -bench=. -benchtime=10s
- Same tags flag rules are applicable to benchmarks tests
- It's not advised to manually set `b.N` in bench function

Also, you can use -benchmem flag to measure memory usage of your program.

Simple benchmark

CODE

```
package main

import (
   "testing"
)

func BenchmarkTestRecursiveFibonacci_10(b *testing.B) {
   for i := 0; i < b.N; i++ {
      RecursiveFibonacci(10)
   }
}

func RecursiveFibonacci(n uint) uint {
   if n <= 1 {
      return n
   }
   return RecursiveFibonacci(n-1) + RecursiveFibonacci(n-2)

}
```

RESULT

```
> go test -bench=.
goos: linux
goarch: amd64
pkg: github.com/wshaman/learn/tasks/fib_bench
cpu: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
BenchmarkTestRecursiveFibonacci_10-
8     3137852              375.6 ns/op
PASS
ok      github.com/wshaman/learn/tasks/fib_bench        1.570s
```

**It's impossible to run bench on playground*

Comparison

CODE

```
package main

import (
  "testing"
)

func BenchmarkReqFib_10(b *testing.B) {
  for i := 0; i < b.N; i++ {
    RecFib(10)
  }
}

func BenchmarkFib_10(b *testing.B) {
  for i := 0; i < b.N; i++ {
    Fib(10)
  }
}
```

RESULT

```
❯ go test -bench=.
goos: linux
goarch: amd64
pkg: github.com/wshaman/learn/tasks/fib_bench
cpu: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
BenchmarkRecFib_10-8     3130125              373.5 ns/op
BenchmarkFib_10-8       297618974             4.066 ns/op
PASS
```

**It's impossible to run bench on playground*

Compare memory usage

CODE

```
package benchmem

import "testing"

const input = "qwertyuiopasdfghjklzxcvbnm"

func BenchmarkHeavySlicer(b *testing.B) {
    for i := 0; i < b.N; i++ {
        heavySlicer(input)
    }
}

func BenchmarkLightSlicer(b *testing.B) {
    for i := 0; i < b.N; i++ {
        lightSlicer(input)
    }
}
```

RESULT

```
❯ go test -bench . –benchmem
goos: linux
goarch: amd64
pkg: grow/benchmem
cpu: Intel(R) Core(TM) i5-10310U CPU @ 1.70GHz
BenchmarkHeavySlicer-8      6469356      183.5 ns/op      248 B/op      5 allocs/op
BenchmarkLightSlicer-8      22586743     50.88 ns/op      112 B/op      1 allocs/op
PASS
```

See full code here: https://go.dev/play/p/puuc4mLA4py
Note: you can't run benchmarks on playground.

Fuzzing

Fuzzing in GO

- Fuzzing is a technique where you automagically generate input values for your functions to find bugs.
- Fuzzing is being released as part of the standard library in Go 1.18.
- Fuzzing will be part of the regular testing library since it is a kind of test.

Creating fuzz tests

Test file in go **MUST** follow rules:

- Name ends with ***_test.go***
- Package name is same as other files OR same with ***_test*** suffix
- functions in test file MUST start with **Fuzz** prefix and accept one param ****testing.F***

At first, the testing.F must be provided with a seed corpus which you should consider example data. This is the data that the fuzzer will use and modify into new inputs that are tried.
The seed should reflect how the input to your function should look as much as possible to get the best results of the fuzzer.

To start fuzzing test the follow command can be used regular ***go test*** command with ***–fuzz=*** flag.

By default, fuzzing tests will run forever until failures occur. It means that we need to cancel this tests or wait until an error.

```
go test --fuzz=. -fuzztime=10s
```

Fuzzing test example

CODE

```
package fuzzing

import (
   "strconv"
   "testing"
)

func uniqueParser(input string) (int, error) {
   output, err := strconv.Atoi(input)
   if err != nil {
      return 0, err
   }
   return output, nil
}

func FuzzUniqueParser(f *testing.F) {
   f.Add("1")
   f.Fuzz(func(t *testing.T, input string) {
      _, err := uniqueParser(input)
      if err != nil {
         t.Errorf("Error: %v", err)
      }
   })
}
```

RESULT

```
❯ go test –fuzz=.
fuzz: elapsed: 0s, gathering baseline coverage: 0/1 completed
fuzz: elapsed: 0s, gathering baseline coverage: 1/1 completed,
now fuzzing with 8 workers
fuzz: minimizing 31-byte failing input file
fuzz: elapsed: 0s, minimizing
--- FAIL: FuzzUniqueParser (0.04s)
   --- FAIL: FuzzUniqueParser (0.00s)
      fuzzing_test.go:21: Error: strconv.Atoi: parsing "A": invalid
syntax

   Failing input written to
testdata/fuzz/FuzzUniqueParser/e9e3ffbe3b3a072c05b41faa7f6
9ead9344b5b040762dfde7273491dc50e7197
   To re-run:
   go test -
run=FuzzUniqueParser/e9e3ffbe3b3a072c05b41faa7f69ead934
4b5b040762dfde7273491dc50e7197
FAIL
exit status 1
FAIL    grow/fuzzing    0.041s
```

**It's impossible to run fuze tests on playground*

Profiling with pprof

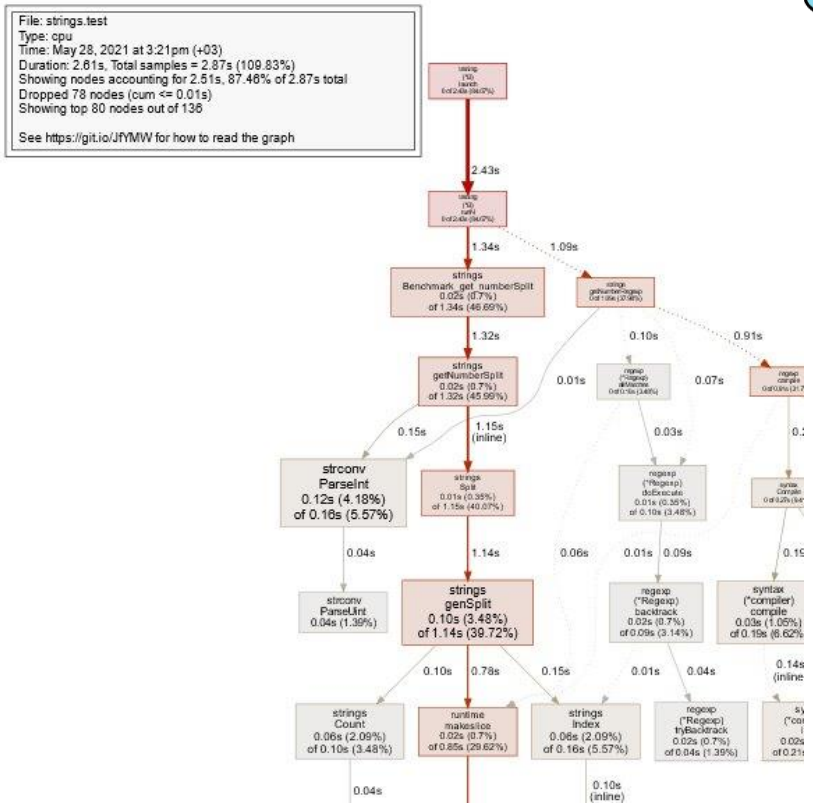
Go profiler

- Go test command can save cpu and memory profiles
- go test -cpuprofile cpu.out -memprofile mem.out -bench=.
- There is a special **pprof** tool in Go for visualization of the profiling results. It has a CLI interface and allow to generate different kind of representations.
- Most popular variants are **web** and **png(web actually opens generated png file in default browser)**
- **go tool pprof -web cpu.out**
- **go tool pprof -png mem.out**
- CLI allows to try some options, like inuse_object

CPU profiling visualization

HIGHLIGHT GOES HERE

- Tree representation, vertical lines shows func usage, horizontal lines for calls inside func
- Wider arrows means more CPU time



Go profiler in a runtime

- There's also built-it option to add pprof output to a web-server with just importing `_ "net/http/pprof"` package.
- Pretty much the same, it generates output for running service, but allow to setup profilers from code, enable/disable in specific parts.
- Generates outputs until stopped.
- To collect runtime data, profiler stops the run, making snapshot and releases run lock. Not advised on real systems, may provide even more problems than resolves.

Common environment

Set values before starting tests

testing.M differs from other testing structs.

- It has only one allowed signature **func TestMain(m *testing.M).**
- Because of previous point, there could be only one initializer func in running test
- It **IS** allowed to create multiple **TestMain** funcs in the same package if **strictly** split by tag.
- m.Run() must be called from **TestMain** in order to start tests
- m.Run() returns exit code.
- Some testing tools from other languages support SetUp() and TearDown() functions, that are called before and after all tests. There is a testify.Suite to comply same behavior. (@see: https://pkg.go.dev/github.com/stretchr/testify/suite)

Running with TestMain

CODE

```
package main
import (
   "fmt"
   "os"
   "testing"
)

func RecFib(n uint) uint {
   if n <= 1 {
      return n
   }
   return RecFib(n-1) + RecFib(n-2)
}

func TestRecFib(t *testing.T) {
   in, expected := uint(10), uint(55)
   if got := RecFib(in); expected != got {
      t.Errorf("got: %d, want: %d", got, expected)
   }
}

func TestMain(m *testing.M) {
   fmt.Println("Hello from main")
   os.Exit(m.Run())
}
```

RESULT

```
> go test -v .
Hello from main
=== RUN   TestRecFib
--- PASS: TestRecFib (0.00s)
PASS
```

**Playground ignores TestMain(m *testing.M) func*

Mocks

3rd party mocking

- Go offers built-in tools to "fake" calls.
- It's a common way to replace **Transport** in http.Client to call for mocked http response
- This way allows use one injection and keep other code intact
- Implementation of responding server is totally up to developer and it's highly customized

```
fakeHttpClient := &http.Client{
  Timeout: 10 * time.Second,
  Transport = newMockTransport()
}

func (t *mockTransport) RoundTrip(req *http.Request) (*http.Response, error) {
  // Create mocked http.Response
  response := &http.Response{
    Header:     make(http.Header),
    Request:    req,
    StatusCode: http.StatusOK,
  }
  return response, nil
}

type RoundTripper interface{
    RoundTrip(*Request) (*Response, error)
}
```

DB mocking

- Database connection based. Really depends on the way DB queries are executed.
- One of the cases: use interface for models(queries) and different struct implementations – one for real DB connection, one for stub
- Stub struct implements DB interface and returns preset data instead of real queries.

```
type database interface {
  UserList(*db.DB)  ([]User, error)
}

func (um userMock) UserList(_ *db.DB) ([]User, error) {
  return []User{
    {  ID:    0, Name:  "stub", Email: "stub@example.com"  },
  }, nil
}

func (um userModel) UserList(dbObj *db.DB) ([]User, error) {
  rows, err := dbObj.Conn.Query("select id, name, email from
users")
  return rowsToUsers(rows)
}
```

“testify” package

Testify package

It's a third-party package widely used in different project

- import "github.com/stretchr/testify/assert"
- It's not advised to use 3rd party libs in play.golang
- **Assert** struct just compares given result
- **Require** is assert or fail version.
- Both have the same set of comparators
- Eg: Equal(), Error(), ErrorIs(),…

```
import (
   "fmt"
   "math/rand"
   "testing"
   "time"

   "github.com/stretchr/testify/assert"
)

func TestCalcPi(t *testing.T) {
   res := CalcPi(90000000)
   fmt.Println(res)
   assert.InDeltaf(t, 3.14, res, 0.02, "failed!")
}

func CalcPi(numPoints int) float64 {
   rand.Seed(time.Now().Unix())
   inCircle := 0
   for i := 0; i < numPoints; i++ {
      x := rand.Float64()
      y := rand.Float64()
      if x*x+y*y <= 1 {
         inCircle++
      }
   }
   return (float64(inCircle) / float64(numPoints)) * 4
}
```

Testify.Mock

- Makes mock more "OOP"
- Allows to use mock generators
- Encapsulates transport modification
- Could be advised for larger projects

```
import "github.com/stretchr/testify/mock"

type service interface {
	SendMessage(string) error
}

type emlServiceMock struct {
	mock.Mock
}

func (ms *emlServiceMock) SendMessage(msg string) error {
	return ms.Called(msg).Error(0)
}

func TestMessenger(t *testing.T){
	eml := emlServiceMock{}
	eml.On("SendMessage", "test message").Return(errors.New("could not send"))
}
```

Questions

Homework

Homework

You will be provided with code which we should cover by tests and achieve as much coverage as possible.

File which should be tested consists of two parts, separated from each other by "/“.
The first part consists of a structure with implemented methods Len, Less and Swap, which means the implementation of the sort.Interface interface.
The second part consists of implementing a matrix to store numbers in rows and columns.

Thanks

Russian:
https://wearecommunity.io/events/ru-golang-united-school-07-testing/talks/44445

English:
https://wearecommunity.io/events/en-golang-united-school-07-testing/talks/44446