GoLang Fundamental

Lawrence Zhou

www.mobilesoftsmith.com

- Introduction
- Development Environment Setup
- Basic Building Blocks
- Advanced Building Blocks
- Concurrency
- Error Handling
- Unit Testing
- Resources

Introduction

- What is Go?

Go, often referred to as Golang, is a general-purpose programming language that was developed at Google in November 2009.

- Why Go?
  A. Created by geniuses: Ken Thompson (B, C, UTF-8), Rob Pike (Unix, UTF-8), Robert Griesemer (Java Hotspot VM, V8 JavaScript engine), etc.
  B. Performance, Design for Multiple-cores, Concurrency
  C. Compiled, Cross-platform Native (no VM), Clean Syntax
  D. Garbage Collected
  E. Powerful Standard Library
  F. Simple and Fun

- Who is using Go?

Google, YouTube, Dropbox, Docker, Shopify, Bitbucket, Firefox, BBC, New York Times, MongoDB, IBM, GE Software, Intel, Github, Twitter, FaceBook, UBER, QiHoo, etc.

- What type projects can Go create?
  A. Web App/API, Network Services
  B. Scripts
  C. System Administration
  D. Image Processing
  E. Servers, Load Balancers
  F. Crypto
  G. IoT

Development Environment Setup

- Uninstalling the old version of Go

  $ sudo rm -r /usr/local/go

- Installing Go

  A. Downlad: https://golang.org/dl/

  B. Run the installer

  C. Verify installation: $ go version

- Setup Go environment variables

  A. GOROOT: Go install path(/usr/local/go)

  B. GOPATH: location of **single** Go **workspace**

  C. Add go executable path (/usr/local/go/bin) to PATH

- Setup Sublime for running Go program

A. Open menu -> Tools -> Build System -> New Build System

B. Copy following to the new build configure file (replace the path based on your system):

```
{
    "cmd": ["/usr/local/go/bin/go", "run", "$file"],
    "file_regex": "^[ ]*File \"(...*?)\", line (0-9}*)",
}
```

C. Save the file as go.sublime-build

D. A "go" menu item will appear in Tools -> Build System, select it

- Go workspace structure:

  [GOPATH]-

  |-bin

  |-pkg

  |-src

  A. bin: compiled executable binary created by "go install " command, it's better to add this path to PATH environment variable

  B. pkg: 3rd party dependencies (libraries, open source projects, etc.)

  C. src: your source code

Basic Building Blocks

- Packages, Variables, and Functions
- Flow Control
- Data structures and Types
- Methods and Interface

- Packages, Variables, and Functions

  A. Packages:

  1. Every Go program is made up of packages.
  2. Programs start running in package 'main'
  3. Use 'import' keyword to use other packages' functionalities

```
import "fmt"
```

or

```
import (
        "fmt"
)
```

  4. A function or variable is exported if it begins with a capital letter.

B. Functions:

1. A function can take 0 or more parameters, and return value. Basic form:

```
func add(x int, y int) int {
        return x + y
}
...
add(1, 3)
```

2. A function can return multiple values:

```
func swap(x, y string) (string, string) {
        return y, x
}
```

3. Function with Named Return Values

```
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}
```

4. Function with Unlimited Parameters (Variadic Function)

```
func variadicFunction(names ... string){
    for _, name := range names {
        fmt.Println(name)
    }
}
```

5. Function Expression:

```
functionExpression := func (name string) {
        fmt.Println("Hello", name, "This is a function expression")
}
...
functionExpression("Lawrence")
```

6. Anonymous Self-Executing Function:

```
func (name string) {
        fmt.Println("\nAnonymous self-executing function:")
        fmt.Println("Hello", name)
} ("Lowrence")
```

7. Function as Parameter (Callback):

```
func functionWithFunctionParameter(a, b int, callback func(string, int)) {
        c:= a + b
        callback("ok", c)
}
...
functionWithFunctionParameter(1, 2, func(result string, sum int) {
        fmt.Println("\nResult in callback:")
        fmt.Println(result, sum)
})
```

C. Variables and Constants:

1. Formal var Declaration:

```
var firstName string = "Lawrence"
var a, b, c int = 1, 2, 3
var (
    x, y, z = 4, true, "Yeah"
    r, s, t string = "r", "s", "t"
)
```

2. Short Variable Declaration (inside function only):

```
lastName := "Zhou"
i, j, k := true, 5.0, "Yes"
```

3. Constants:

```
const Pi = 3.1415926
or
const(
    Pi = 3.1415926
    HttpPrefix = "https://"
    IsDebug = true
)
```

- Flow Control:
    A. For loop:
        1. Standard:

```
sum := 0
for i := 0; i < 10; i++ {
        sum += i
}
```

        2. Simplified:

```
sum := 1
for ; sum < 1000; {
        sum += sum
}
```

        3. While like loop

```
sum := 1
for sum < 2000 {
        sum += sum
}
```

        4. Endless loop:

```
for {
        ...
        continue
        ...
        break
}
```

B. Conditions (If...else...)

C. Switches:

1. Default is not fall through (no break required)

```
switch os := runtime.GOOS; os {
case "darwin":
        fmt.Println("OS X.")
case "linux":
        fmt.Println("Linux.")
default:
        fmt.Printf("%s.", os)
}
```

2. "fallthrough" keyword is used to force fallthrough

```
switch "Marcus" {
case "Marcus":
        fmt.Println("This")
        fallthrough
case "Jenny":
        fmt.Println("is")
        fallthrough
case "Julian":
        fmt.Println("fallthrough")
        fallthrough
default:
        fmt.Println("test")

}
```

3. Switch with no expression (it's a shortcut of if...else if ... else):

```
t := time.Now()
switch { // same as switch true
case t.Hour() < 12:
	fmt.Println("Good morning!")
case t.Hour() < 17:
	fmt.Println("Good afternoon.")
default:
	fmt.Println("Good evening.")
}
```

4. Switch with multiple values:

```
switch "Jenny" {
case "Tim", "Jenny":
	fmt.Println("What's up? Tim or Jenny")
case "Marcus", "Medhi":
	fmt.Println("What's up? Marcus or Medhi")
case "Julian", "Sushant":
	fmt.Println("What's up? Julian or Sushant")
default:
	fmt.Println("What's up? Nobody")
}
```

D. Defer:

```
func demoDefer(){
	fmt.Println("\nDefer demo:")
	defer fmt.Println("world") // run last

	fmt.Print("hello ") // run first
	fmt.Print("Lawrence ") // run second

	// stacking defers
	fmt.Println("counting")

	for i := 0; i < 10; i++ {
		defer fmt.Println(i)
	}

	fmt.Println("done")
}
```

- Data Structures and Types

  A. Pointers:

  A pointer holds the memory address of a value. The type *T is a pointer to a T value. Its zero value is nil.

```
var p *int
i := 42
p = &i
fmt.Println(*p) // not like C, there is no p++
```

  B. Structs:

A struct is a collection of fields

```
type Vertex struct {
    X int
    Y int
}
...
v := Vertex{1, 2}
v.X = 4
fmt.Println(v.X)
```

C.Arrays:

1. Expression:

```
var a [2]string
a[0] = "Hello"
a[1] = "World"
...
primes := [6]int{2, 3, 5, 7, 11, 13}
```

D. Slices:

1. A slice is a dynamically-sized, flexible view into the elements of an array.

```
primes := [6]int{2, 3, 5, 7, 11, 13} // array
var s []int = primes[1:4]    // slice
```

2. A slice does not store any data, it just describes a section of an underlying array (slices are like references to arrays).

3. Creating a slice with make function:

```
a := make([]int, 5)  // len(a)=5
b := make([]int, 0, 5) // len(b)=0, cap(b)=5
```

4.Range:

The range form of the for loop iterates over a slice or map.

When ranging over a slice, two values are returned for each iteration. The first is the index, and the second is a copy of the element at that index.

```
var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

for i, v := range pow {
    fmt.Printf("2**%d = %d\n", i, v)
}
```

E.Maps:

1. A map maps keys to values.

var m map[string]string

...

2. The make function returns a map of the given type, initialized and ready for use.

m = make(map[string]int)

m["one"] = 1

3. Map literals:

```
m = map[string]int {
      "one": 1,
      "two": 2,
      "three": 3,
}
```

4. Mutating Maps:

Insert or update: m[key] = elem

Retrieve: elem = m[key]

Delete: delete(m, key)

Test if a key exists:

elem, ok = m[key]

ok is a bool

- Methods and Interface

  A. Methods:

  1. A method is a function with a special receiver argument.

```
type Vertex struct {
        X, Y float64
}
func (v Vertex) Abs() float64 {
        return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
func (v *Vertex) Scale(f float64) {
        v.X = v.X * f
        v.Y = v.Y * f
}
func main() {
        v := Vertex{3, 4}
        v.Scale(10)  // pointer indirection
        fmt.Println(v.Abs())
}
```

B. Interfaces

1. An interface type is defined as a set of method signatures.

```
type Abser interface {
	Abs() float64
}
type Vertex struct {
	X, Y float64
}
func (v *Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
func main() {
	var a Abser
	v := Vertex{3, 4}
	a = &v // a *Vertex implements Abser

	// In the following line, v is a Vertex (not *Vertex)
	// and does NOT implement Abser.
	a = v // error!
	fmt.Println(a.Abs())
}
```

2. Interface values with nil underlying values and Nil interface values

```
type I interface {
	M()
}
type T struct {
	S string
}
func (t *T) M() {}
func main() {
	var i I
	var t *T
	i.M() // panic error, Nil interface value
	i = t
	describe(i)
	i.M() // OK, Interface with nil underlying value
}
func describe(i I) {
	fmt.Printf("(%v, %T)\n", i, i)
}
```

3.The empty interface

interface{}

An empty interface may hold values of any type. (Every type implements at least zero methods.)

4. Type assertions：

A type assertion provides access to an interface value's underlying concrete value.

t := i.(T)

t, ok := i.(T)

ok is a bool

5. Type switches

A type switch is a construct that permits several type assertions in series.

```
switch v := i.(type) {
case T:
    // here v has type T
case S:
    // here v has type S
default:
    // no match; here v has the same type as i
}
```

Advanced Building Blocks

- Goroutines
- Channels
- Range and Close
- Select

- Goroutines

    1. A goroutine is a lightweight thread managed by the Go runtime.

    go f(x, y, z)

- Channels

    1.Channels are a typed conduit through which you can send and receive values with the channel operator, <-.

    ch <- v // Send v to channel ch.

    v := <-ch // Receive from ch, and assign value to v.

    2. Like maps and slices, channels must be created before use:

    ch := make(**chan** int)

    3. By default, sends and receives **block** until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.

```
c := make(chan int)
go func() {
        for i := 0; i < 10; i++ {
                c <- i
        }
}()
go func() {
        for {
                fmt.Println(<-c)
        }
}()
time.Sleep(time.Second)
```

4.Buffered Channels

Channels can be buffered. Provide the buffer length as the second argument to make to initialize a buffered channel:

```
ch := make(chan int, 100)
```

Sends to a buffered channel block only when the buffer is full. Receives block when the buffer is empty.

5. Channel Direction:

When using channels as function parameters, you can specify if a channel is meant to only send or receive values. This specificity increases the type-safety of the program.

```
func ping(pings chan<- string, msg string) {
    pings <- msg
}
// The pong function accepts one channel for receives (pings) and
a second for sends (pongs).
func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}
func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pongs)
}
```

- Range and Close

A sender can close a channel to indicate that no more values will be sent. Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression: after

```
v, ok := <-ch
```

The loop for i := range c receives values from the channel repeatedly until it is closed.

Note: Only the sender should close a channel, never the receiver. Sending on a closed channel will cause a panic.

Channels aren't like files; you don't usually need to close them. Closing is only necessary when the receiver must be told there are no more values coming, such as to terminate a range loop.

range example:

```
func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c)
}

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}
```

- Select

    1. The select statement lets a goroutine wait on multiple communication operations.

    A select blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

```
func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}
func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}
```

2. Default Selection

The default case in a select is run if no other case is ready.

Use a default case to try a send or receive without blocking:

```
select {
case i := <-c:
    // use i
default:
    // receiving from c would block
}
```

Default Selection example:

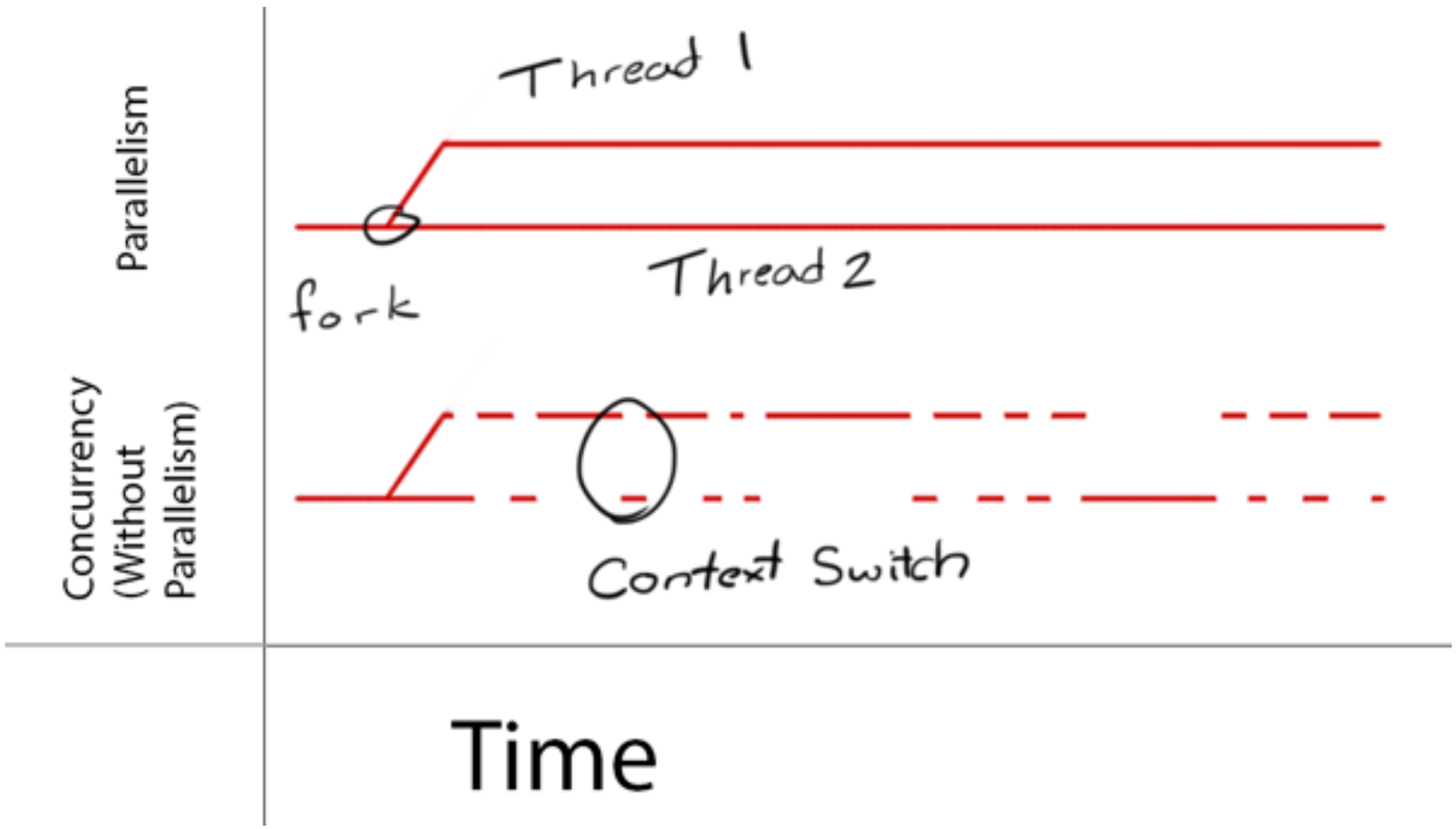
```
func main() {
	tick := time.Tick(100 * time.Millisecond)
	boom := time.After(500 * time.Millisecond)
	for {
		select {
		case <-tick:
			fmt.Println("tick.")
		case <-boom:
			fmt.Println("BOOM!")
			return
		default:
			fmt.Println("    .")
			time.Sleep(50 * time.Millisecond)
		}
	}
}
```

Concurrency

- Concurrency and Parallelism
- Race and Deadlock
- WaitGroup
- Mutex
- Atomicity
- Semaphores

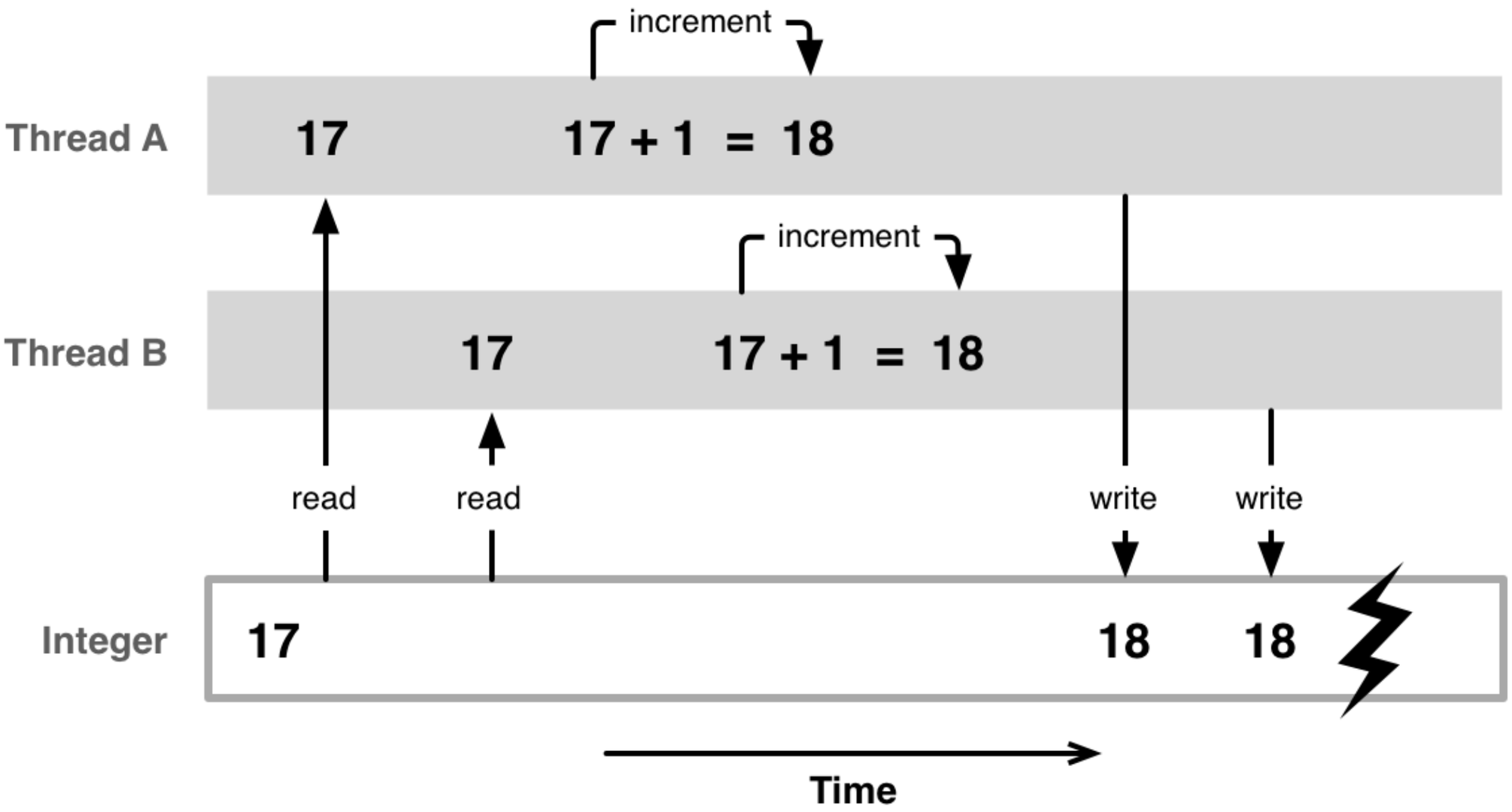
- Concurrency and Parallelism

Concurrency is not parallelism. Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.



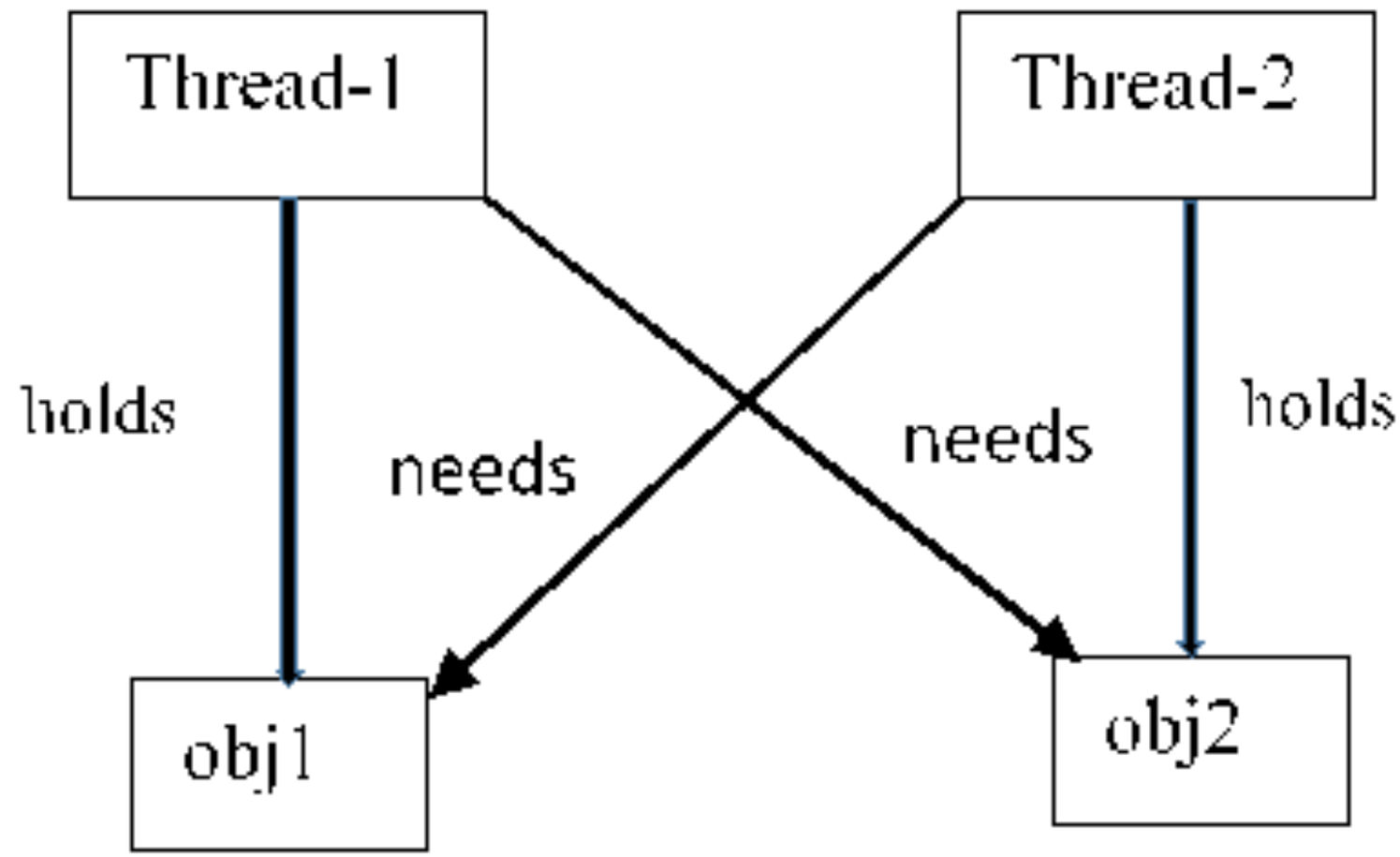
- Race and Deadlock

  A. Race Condition:



You can use "-race" to check if there is a race in your code. For example:
$ go run -race main.go

B. Deadlock



- WaitGroup

A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. Then each of the goroutines runs and calls Done when finished. At the same time, Wait can be used to block until all goroutines have finished.

```
var wg sync.WaitGroup
func main() {
	wg.Add(2)
	go foo()
	go bar()
	wg.Wait()
}
func foo() {
	for i := 0; i < 15; i++ {
		fmt.Println("Foo:", i)
	}
	wg.Done()
}
func bar() {
	for i := 0; i < 15; i++ {
		fmt.Println("Bar:", i)
	}
	wg.Done()
}
```

- Mutex

A Mutex is a mutual exclusion lock. Mutexes can be created as part of other structures; the zero value for a Mutex is an unlocked mutex.

```
var wg sync.WaitGroup
var counter int
var mutex sync.Mutex

func main() {
        wg.Add(2)
        go incrementor("Foo:")
        go incrementor("Bar:")
        wg.Wait()
        fmt.Println("Final Counter:", counter)
}

func incrementor(s string) {
        for i := 0; i < 20; i++ {
                time.Sleep(time.Duration(rand.Intn(20)) * time.Millisecond)
                mutex.Lock()
                counter++
                fmt.Println(s, i, "Counter:", counter)
                mutex.Unlock()
        }
        wg.Done()
}
```

- Atomicity

Package atomic provides low-level atomic memory primitives useful for implementing synchronization algorithms.

```
var wg sync.WaitGroup
var counter int64

func main() {
	wg.Add(2)
	go incrementor("Foo:")
	go incrementor("Bar:")
	wg.Wait()
	fmt.Println("Final Counter:", counter)
}

func incrementor(s string) {
	for i := 0; i < 20; i++ {
		time.Sleep(time.Duration(rand.Intn(3)) * time.Millisecond)
		atomic.AddInt64(&counter, 1)
		fmt.Println(s, i, "Counter:", atomic.LoadInt64(&counter)) // access
without race
	}
	wg.Done()
}
```

- Semaphores

In computer science, a semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multiprogramming operating system.

A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions. The variable is then used as a condition to control access to some system resource.

```
func main() {
	c := make(chan int)
	done := make(chan bool)
	go func() {
		for i := 0; i < 10; i++ {
			c <- i
		}
		done <- true
	}()
	go func() {
		for i := 0; i < 10; i++ {
			c <- i
		}
		done <- true
	}()
	go func() {
		<-done
		<-done
		close(c)
	}()
	for n := range c {
		fmt.Println(n)
	}
}
```

Error Handling

- Use golint to Improve Your Code
- Four Common Ways to Handle Errors
- Handling Errors & Logging Errors to a File
- Custom Errors
- Idiomatic Error Handling
- Provide Context with Errors

- Use golint to Improve Your Code

  A. golint will check go syntax of your code

  B. Install:

  go get -u github.com/golang/lint/golint

  C. Usage:

  check all files under current directory:

  golint ./...

- Four Common Ways to Handle Errors

```
_, err := os.Open("no-file.txt")
if err != nil {
    fmt.Println("err happened", err) // display error
    log.Println("err happened", err) // log error
    log.Fatalln(err) // exit status 1, log error
    panic(err)       // panic error, show file path and
line #, exit status 2
}
```

- Handling Errors & Logging Errors to a File

  A. No "try/catch" block in go

  B. By leveraging function can return multiple values feature, functions in Go typically return a value of a built-in error type, along with other values returned from a function

```
func GetById(id string) (models.Task, error) {
    var task models.Task
    // Implementation here
    return task, nil // multiple return values
}
...
task, err:= GetById ("105")
if err != nil {
   log.Fatal(err)
}
//Implementation here if error is nill
```

C. Log error to a file:

```
func init() {
	nf, err := os.Create("log.txt")
	if err != nil {
		fmt.Println(err)
	}
	log.SetOutput(nf)
}
func main() {
	_, err := os.Open("no-file.txt")
	if err != nil {
		//		fmt.Println("err happened", err)
		log.Println("err happened", err)
		//		log.Fatalln(err)
		//		panic(err)
	}
}
```

- Custom Errors

  A. Creating custom error by build-in functions:

  1. errors.New:

  ...

  return 0, **errors.New**("norgate math: square root of negative number")

  2. fmt.Errorf:

  ...

  return 0, **fmt.Errorf**("norgate math: square root of negative number")

  B. Custom Error Type

  Error interface:

  type error interface {

  Error() string

  }

Custom error type example:

```
type MyError struct {
	When time.Time
	What string
}
func (e *MyError) Error() string {
	return fmt.Sprintf("at %v, %s",
		e.When, e.What)
}
func run() error {
	return &MyError{
		time.Now(),
		"it didn't work",
	}
}
func main() {
	if err := run(); err != nil {
		fmt.Println(err)
	}
}
```

- Idiomatic Error Handling

```
var ErrNorgateMath = errors.New("norgate math: square root of negative number")
func main() {
	fmt.Printf("%T\n", ErrNorgateMath)
	_, err := Sqrt(-10)
	if err != nil {
		log.Fatalln(err)
	}
}
func Sqrt(f float64) (float64, error) {
	if f < 0 {
		return 0, ErrNorgateMath
	}
	// implementation
	return 42, nil
}
```

- Provide Context with Errors

```
...
func Sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, fmt.Errorf("norgate math again: square root of negative number: %v", f)
    }
    // implementation
    return 42, nil
}
```

Unit Testing

- Unit Test in Go
- Benchmarking in Go
- Practices, Cross-Compilation, and the Go Tool

- Unit Test in Go
  A. Creating files appended with _test.go for testing
  B. Import "testing" package
  C. Write functions with the signature TextXxxx(*test.T)
  D. Use command "go test"

go test -run '' # Run all tests.

go test -run Foo # Run top-level tests matching "Foo", such as "TestFooBar".

go test -run Foo/A= # For top-level tests matching "Foo", run subtests matching "A=".

go test -run /A=1 # For all top-level tests, run subtests matching "A=1".

- Benchmarking in Go
  A. Creating files appended with _test.go for testing
  B. Import "testing" package
  C. Benchmark test function is of form
     func BenchmarkXxx(*testing.B)
  D. Use go test -bench={wildcard} to run benchmark tests

  go test -bench=. # run all benchmarks

Go test functions example:

```
func Adder(xs ...int) int {
	res := 0
	for _, v := range xs {
		res += v
	}
	return res
}
...
func TestAdder(t *testing.T) {
	result := Adder(4, 7)
	if result != 11 {
		t.Fatal("4 + 7 did not equal 11")
	}
}
...
func BenchmarkAdder(b *testing.B) {
	for i := 0; i < b.N; i++ {
		Adder(4, 7)
	}
}
```

- Effective Go Programming
  A. init() function
  B. import for side effect
     import _ "net/http/pprof"
  C. Use type switch and type detection

- Cross-Compiling in Go
  A. Set GOOS and GOARCH as per https://golang.org/doc/install/source#environment
  B. Run "go build" command
  $ env GOOS=windows GOARCH=amd64 go build -o math.exe math.go

- Go Tool Overview
  go help, go doc, go fmt

- Resources
  1. Go document: https://golang.org/doc/
  2. Go by example: https://gobyexample.com
  3. Go Tutorials and Courses: https://hackr.io/tutorials/learn-go
  4. Why I Love Golang: https://hackernoon.com/why-i-love-golang-90085898b4f7