Start Date: 27/06/2019

**WINDOWS**:: SETUP GOMOD COMMAND IN WINDOWS POWERSHELL:

**$env:GO111MODULE = 'on'**

<https://www.ostechnix.com/install-go-language-linux/>

GO LANG BASICS:

1. What is rune type?

**Rune literals are just 32-bit integer values** (however they're untyped constants, so their type can change). They represent unicode codepoints. For example, the rune literal 'a' is actually the number 97

* Rune is a Type.
* RUNE occupies 32bit and is meant to represent a [Unicode](http://en.wikipedia.org/wiki/Unicode)[CodePoint](http://en.wikipedia.org/wiki/Code_point)(https://en.wikipedia.org/wiki/Code\_point). As an analogy the english characters set encoded in 'ASCII' has 128 code points. Thus is able to fit inside a byte (8bit). From this (erroneous) assumption C treated characters as 'bytes' char, and 'strings' as a 'sequence of characters' char\*.
* In golang then a string is a sequence of bytes. However, since multiple bytes can represent a rune code-point, a string value can also contain runes. So, it can be converted to a []rune, or vice versa.

To summarize, here are the salient points:

* Go source code is always UTF-8.
* A string holds arbitrary bytes.
* A string literal, absent byte-level escapes, always holds valid UTF-8 sequences.
* Those sequences represent Unicode code points, called runes.
* No guarantee is made in Go that characters in strings are normalized.

GO COMMANDS::

**go build**

* Need the package file .a in $GOPATH/pkg, use go install instead.
* If the package is the main package, it will generate an executable file in the same folder. If you want the file to be generated in $GOPATH/bin, use go install or go build -o ${PATH\_HERE}/a.exe.
* go build will compile all the files in the folder.
* go build chooses the one that's associated with your operating system. For example, it only compiles array\_linux.go in Linux systems, and ignores all the others.

EX: They could be named as follows:

array\_linux.go | array\_darwin.go | array\_windows.go | array\_freebsd.go

## go clean

This command is for cleaning files that are generated by compilers

## go fmt and gofmt:

Go, there is only one code style which is enforced.

go fmt is just an alias, which runs the command 'gofmt -l -w' on the packages named by the import paths.

We usually use gofmt -w instead of go fmt. The latter will not rewrite your source files after formatting code. gofmt -w src formats the whole project.

## go install

This command compiles all packages and generates files, then moves them to $GOPATH/pkg or $GOPATH/bin

## go test

This command loads all files whose name include \*\_test.go and generates test files

## godoc

Many people say that we don't need any third-party documentation for programming in Go (actually I've made a [CHM](https://github.com/astaxie/godoc) already). Go has a powerful tool to manage documentation natively.

use the godoc fmt Printf and godoc -src fmt Printf commands to view the source code.

Execute the godoc -http=:8080 command, then open 127.0.0.1:8080 in your browser.

go fix // upgrade code from an old version before go1 to a new version after go1

go version // get information about your version of Go

go env // view environment variables about Go

go list // list all installed packages

go run // compile temporary files and run the application

Go Interfaces:

* **Interface{} Says nothing - ROB Pike**

**Principles to Follow in GO while creating the interfaces:**

Machine generated alternative text:
Gopher China 2017 
bigger the interface, 
"The 
the weaker the 
abstraction" 
- Rob Pike in his Co Proverbs 

Purpose Of interfaces:

Machine generated alternative text:
Gopher China 2017 
why do we use interfaces? 
writing generic algorithms 
hiding implementation details 
providing interception points 

In Go We have two type of Interfaces:

\* Abstract Types

\* Concrete Types

Abstract Types In GO:

Machine generated alternative text:
abstract types in Go 
they describe behavior 
io.Reader 
Gopher China 2017 
fmt-Stringer 
io.Writer 
they define a set of methods, without specifring the receiver 
type Positiver interface 
Positive() bool 

ABSTRACT TYPES:

Machine generated alternative text:
Gopher China 2017 
Abstract Data Types 
Mathematical model for data types 
Defined by its behavior in terms of: 
possible values, 
possible operations on data of this type, 
and the behavior of these operations 

Concrete Types In GO:

Machine generated alternative text:
concrete types in Go 
they describe a memory layout 
Gopher China 2017 
int64 
int16 
int32 
behavior attached to data through methods 
type Number i" 
func (n Numbe 
return n 
ositive() bool 

Union Of Interface:

Machine generated alternative text:
Gopher China 2017 
union of interfaces 
type ReadWriter interface 
Reader 
Writer 

Robust Principles in GO:

Machine generated alternative text:
Gopher China 2017 
"Return concrete types, 
receive interfaces as 
parameters" 
- Robustness Principle applied to Go (me) 

Machine generated alternative text:
Gopher China 2017 
Hiding implementation details 
Use interfaces to hide implementation details: 
decouple implementation from API 
easily switch between implementations / or provide multiple ones 

CALL DISPATCH:

Machine generated alternative text:
call dispatch 
Concrete types: static 
known at compilation 
very efficient 
can't intercept 
'i, 
Gopher China 2017 
Abstract types: dynamic 
unknown at compilation 
less efficient 
easy to intercept 

TYPE ASSERTIONS IN INTERFACES :

Machine generated alternative text:
Gopher China 2017 
type assertions from interface to concrete type 
do(v interface()) 
func 
i 
i, 
v. (int) 
ok v. (int) 
// will panic if v is not int 
// will return false 

Machine generated alternative text:
Gopher China 2017 
type assertions from interface to concrete type 
func do(v interface()) 
v. (type) 
select t 
// t is of type int 
case int: 
fmt.Println(Ugot int *d", t) 
// t is of type interface() 
default : 
sure what type") 

NOTE: **Avoid the abstract to concrete assertions**

* **Use type assertions to extend the behaviors.**
* **Use type assertions to classify the errors.**

**ERROR TALK:**

Machine generated alternative text:
Don't just check 
errors, handle 
them gracefully 
Go Proverb 
Dave Cheney - GopherCon 2016 
Gopher China 2017 

Adding the new **method to the interfaces with breaking the existing code**:

Machine generated alternative text:
Gopher China 2017 
type assertions as evolution mechanism 
Step 1: add the method to your concrete Wpe implementations 
Step 2: define an interface containing the new method 
Step 3: document it 

Example:

Machine generated alternative text:
Gopher China 2017 
http.Pusher 
Pusher interface 
type 
Push(target string, opts *PushOptions) error 
func handler(w http.ResponseWriter, r *http.Request) 
if p, ok w.(http.pusher); ok 
p. nil) 

**Goroutines**

What is a goroutine? It's an independently executing function, launched by a go statement.

It has its own call stack, which grows and shrinks as required.

It's very cheap. It's practical to have thousands, even hundreds of thousands of goroutines.

It's not a thread.

There might be only one thread in a program with thousands of goroutines.

Instead, goroutines are multiplexed dynamically onto threads as needed to keep all the goroutines running.

But if you think of it as a very cheap thread, you won't be far off.

From <[*https://talks.golang.org/2012/concurrency.slide#17*](https://talks.golang.org/2012/concurrency.slide#17)>

[**Concurrency is not parallelism**](https://blog.golang.org/concurrency-is-not-parallelism)

<https://blog.golang.org/concurrency-is-not-parallelism>

when people hear the word *concurrency* they often think of *parallelism*, a related but quite distinct concept.

In programming, concurrency is the *composition* of independently executing processes, while parallelism is the simultaneous *execution* of (possibly related) computations.

Concurrency is about *dealing with* lots of things at once. Parallelism is about *doing* lots of things at once.

**What is concurrency?**

Concurrency is the composition of independently executing computations.

Concurrency is a way to structure software, particularly as a way to write clean code that interacts well with the real world.

It is not parallelism.

From <[*https://talks.golang.org/2012/concurrency.slide#6*](https://talks.golang.org/2012/concurrency.slide#6)>

**Concurrency is not parallelism**

Concurrency is not parallelism, although it enables parallelism.

If you have only one processor, your program can still be concurrent but it cannot be parallel.

On the other hand, a well-written concurrent program might run efficiently in parallel on a multiprocessor. That property could be important...

For more on that distinction, see the link below. Too much to discuss here.

[golang.org/s/concurrency-is-not-parallelism](http://golang.org/s/concurrency-is-not-parallelism)

From <[*https://talks.golang.org/2012/concurrency.slide#7*](https://talks.golang.org/2012/concurrency.slide#7)>

**Channels**

**Channels are first-class values, just like strings or integers.**

A channel in Go provides a connection between two goroutines, allowing them to communicate.

// Declaring and initializing.  
 var c chan int  
 c = make(chan int)  
 // or  
 **c := make(chan int)**

// Sending on a channel.  
 **c <- 1**

// Receiving from a channel.  
 // The "arrow" indicates the direction of data flow.  
 **value = <-c**

Channel Basics:

* Create channel with `make` Command.

Ex: ch := make(chan int)

* Send message into channel

Ex: ch <- val

* Receive message from channel

Ex: val := <- ch

* Can have multiple sender and receivers.

* Note: By default channels are Bi-directional constructs , So you can send and receive the data into a channel.

Restricting the data flow in channels:

* We can restrict the channels by changing the direction of arrow before and after the chan key word in the function.

Machine generated alternative text:
Search 
SUMMARY 
• Restricting data flow 
• Channel can be cast into send-only or receive only versions 
• Send-only: Chan int 
• Receive-only: c-chan int 
on the receiving side so for example in 
the argument 

Buffered Channels:

* Channels block sender side till receiver is available.
* Block receiver side till message is available.
* Can decouple sender and receiver with Buffered channels.

Ex: ch := make(chan int, 50)

* Use buffered channels when sender and receiver have asymmetric loading

For … range loop with channels:

Machine generated alternative text:
Search 
SUMMARY 
• For...range loops with channels 
• Use to monitor channel and process messages as they arrive 
• Loop exits when channel is closed 
is going to detect that and it will go 
ahead and 

Select statements in channels:

* Multiple channels receive the message simultaneously then the behavior is actually undefined so because of highly parallel nature of many go application you can get into the situations where message arrive on two at virtually the same time so one of the those cases will get the nod from the select block but you cannot be sure which one going get it so there is no rule like switch block first matches is going to get it, it could be anyone so the ordering of the cases in your select statement really doesn't matter from the standpoint of how those conflicts are going to get resolved.

If you do want non-blocking select statement remember that you can add the default case in there. If there are no message on any of the monitored channels then default case go ahead and fire so the select statement process and execution of the go routine will continue from there.

Machine generated alternative text:
Search 
SUMMARY 
• Select statements 
• Allows goroutine to monitor several channels at once 
• Blocks if all channels block 
• If multiple channels receive value simultaneously, behavior is undefined 
hearing your feedback and seeing the 
value 