Go Basics

Go Basics
- Why Go
- Milestones
- Installation & Configuration
- Development Tools
- Syntax & Go basics
- Significant features
- Downsides
- Typical occasions

Why Go

- C-like syntaxt
- rapid developing & high performance benchmark
- easy deploying: staticly linked when building executable binaries
- standard library support: db/net/encoding/test
- goroutine: write parallel programs in serial thinking
- "Google" brand

Milestones

- 2007.9 Rob Pike、Ken Thompson、Robert Griesemer started designing go
- 2009.11 Birth
- 2012.3 Go 1.0 released
- 2017.8 Go 1.9 released

Installation & Configuration

- Fetch go installer of your platform(unix/linux/windows supported) and install
- Define your **GOROOT**(where go installed) & **GOPATH**(workspace) environment variables
- Test `go` command, you should get:

> Go is a tool for managing Go source code.
>
> Usage:
>
> ```
> go command [arguments]
> ```
>
> The commands are:

```
build       compile packages and dependencies
clean       remove object files
doc         show documentation for package or symbol
env         print Go environment information
bug         start a bug report
fix         run go tool fix on packages
fmt         run gofmt on package sources
generate    generate Go files by processing source
get         download and install packages and dependencies
.....
```

- Examples

```
go fmt // format source code
go get "github.com/lwldcr/hbase1" // get a package from github
```

Development Tools

- IDE
  - LiteIDE
  - Gogland
  - Idea
- Text Editor
  - Sublime Text + GoSublime
  - Vim

Syntax & Go basics

- Hello world

```go
package main // must have this line

import "fmt"

func main() {
  fmt.Println("Hello, World! This is a Go program.")
}
```

  - package
    - "main": must contain a function: func main() {}, compiled as a binary
    - other: library package providing function for reuse and deliver
  - import
    - import anything you need, here "fmt" is a built-in packge, with frequently used text formating functions, like the `fmt.Prinln` above

        - import more than one packages should be written as a parenthesized list in go convention:

```
import (
  "fmt"
  "net/http"
  "errors"
)
```

    - func
        - stating of function, followed by a function name, parameters list inside parenthesis, returning values (optional)

```
func function1() { // a function with no parameters return nothing
}

func function2(i int, s string) { // a function with parameters, return nothing
}

func function3(i int, s string) int { // returns an int
}

func function4(i int, s string) (int, error) { // returns an int and an error
}
```

- Variables definition
    - Complicated way

```
var i int // state a new variable i whose type is int
i = 10 // assign 10 to i
fmt.Println(i)
```

    - Short way

```
i := 10 // state an int type variable i and assign it to 10
```

- Flow control
    - for

```
for i := 0; i < 10; i++ { // C & Java style usage
  fmt.Printf("i = %d\n", i)
}

for { // same as "while"
}

for i, v := range integer_slice { // typical usage in Go: looping
an array-like object
}
```

- if else
- switch
  - no fallthrough

```
i := 1
switch i {
  case 1:
  fmt.Println("1") // we dont have to add an "break" here
                   // the following "default" will never be
printed
  default:
  fmt.Println("default") // won't be called
}
```

- goto

```
var odd_sum, even_sum int
for i := 0; i < 10; i++ {
  if i % 2 == 0 {
      goto EVEN
  } else {
      goto ODD
  }
  EVEN:
  even_sum += i
  continue
  ODD:
  odd_sum += i
  continue
}
fmt.Println("odd_sum:", odd_sum, "even_sum:", even_sum)

>>> odd_sum: 25 even_sum: 20
```

- type definition

```
type DataType struct {
    Visible int // will be seen out of the original package
    invisible int // invisible out of the original package
    SubData SubDataType // nesting definition
}

type SubDataType struct {
    Data string
}
```

- defer, panic and recover
  - defer: call before function returns

```
func Query(query Query) ([]Data) {
  conn := getDbConn() // assume getDbConn() returns a db connection
  defer conn.Close() // close db connection before function returns
  // do your job here
  .....
}
```

  - panic & recover

- panic: raise error
- recover: got panic error, and try handling error

```
package main

import "fmt"

func main() {
    fmt.Printf("hello world")
    defer func() {
        if err := recover(); err != nil {
            fmt.Println("got panic error: ", err)
        }
    }()
    myPainc()
    fmt.Printf("This will not show")
}

func myPainc() {
    var x = 30
    var y = 0
    panic("Panic error") // raising an error
    var c = x / y
    fmt.Println(c) // this will not show
}
```

> hello world got panic error: Panic error
>
> Process finished with exit code 0

Significant features

- Slice
  - definition

    ```
    intslice := make([]int, 10, 15) // make an int slice, with 10 elements, capicity 15
    ```

  - data structure: a pointer to an array, a length, a capacity
  - as function parameters: go functions always copy data, so passing huge arrays costs much, use slices instead
  - related reading: [Go slices are not dynamic arrays](https://appliedgo.net/slices/), 译文
- Channel
  - definition

```
ch1 := make(chan int, 10) // make a new channel with buffer size of 10
ch2 := make(chan int) // make a new channel without buffer, block read & write
for i := 0; i < 10; i++ {
  ch1 <- i // you cannot continueously write into a channel with no buffer
  ch2 <- i // this will cause error
}
```

- in & out

```
for i := 0; i < 10; i++ { // in function1
    ch1 <- i
}
```

```
for {
  select {
    case i := <- ch1:
    fm.Println("got:", i)
  default:
    fmt.Println("got nothing")
    time.Sleep(1 * time.Microsecond)
  }
}
```

- Interface

```
type Person interface { // define an interface
  SayHello()
}

type Man struct {
  Name string
  Birthday string
}

func (m Man) SayHello() { // type Man implements SayHello() function
                          // Man could be used as Person type
  fmt.Println("hello, this is", m.Name)
}

type Woman struct {
   Name string
   Married bool
}

func (w Woman) SayHello() {
  fmt.Println("hi, i'm", w.Name)
}

func Greeting(p Person) {
  p.SayHello() // call SayHello()
}

func main() {
  m := Man{Name:"Bruce"}
  w := Woman{Name:"Alice", Married:false}
  Greeting(m)
  Greeting(w)
}
```

> hello, this is Bruce
>
> hi, i'm Alice

- Pointers

```
func swap(i *int, j *int) {
  temp := *i
  *i = *j
  *j = temp
  // *i, *j = *j, *i
}

func main() {
  i, j := 1, 10
  fmt.Printf("i = %d, j = %d\n", i, j)

  swap(&i, &j)
  fmt.Printf("i = %d, j = %d\n", i, j)
}
```

> i = 1, j = 10
>
> i = 10, j = 1

- Goroutine

```
// "net/http" server.go
// http.ListenAndServe()
func (srv *Server) Serve(l net.Listener) error {
    .....
    for {
        rw, e := l.Accept()
        ......
        go c.serve(ctx) // for every request, server starts a new
Goroutine to serve
    }
}
```

- Sync

```
package main

import (
    "fmt"
    "sync"
)

func produce(ch chan int, sig chan int, wg *sync.WaitGroup) {
    for i := 0; i < 10; i++ {
        ch <- i
        fmt.Println("append:", i)
    }
    fmt.Println("appending done")
```

```
    sig <- 0 // append done signal
    wg.Done()
}

func consume(ch chan int, sig chan int, wg *sync.WaitGroup) {
    fmt.Println("start consuming..")
    for {
        select {
        case i, ok := <- ch:
            if !ok {
                fmt.Println("error")
                wg.Done()
                return
            }
            fmt.Println("got:", i)
        case <- sig:
            fmt.Println("got stop signal")
            wg.Done()
            return
        }
    }
}

func main() {
    ch := make(chan int) // make a data channel
    sig := make(chan int) // make a signal channel
    var wg sync.WaitGroup // state a WaitGroup variable
    wg.Add(2) // add 2

    go consume(ch, sig, &wg) // start consuming goroutine
    go produce(ch, sig, &wg) // start produce

    wg.Wait() // wait until all goroutines done
    close(ch) // close channels
    close(sig)
}
```

- Powerful built-in Libraries
    - net

- encoding
- glog

Downsides

- Notorious "stop the world" (fixed in Go 1.5)
- Fixed coding style(easy to read others' code)
- No classes(struct), no inheritance(nested struct), no generic type(interface{})
- Erro handling, someone feels elegant, most users think ugly

```
if a, err := function1(); err != nil { // call function1 and check
error
  handle_error()
  return
}

if b, err := function2(); err != nil { // call function2 and check
error
  handle_error()
  return
}
```

Typical occasions

- High performance HTTP server
- A powerful game server
- API service
- Data processing
- My projects
  - hf-themis - service for monitoring and sending alert messages
  - hf-dataservice - API services, have data communicating with Redis/Hbase/Impala

>