**1. golang**

**아래의 내용은 golang에 대한 이야기이므로 복붙과 출처로 대처합니다..**

**Go 개발자들**

GO 프로그래밍 언어는 2007년 구글에서 개발을 시작하여 2012년 GO 버젼 1.0을 완성하였다. GO는 이후 계속 향상된 버젼을 내 놓았으며 2015년 말에는 1.5.2 버젼에 이르렀다.

흔히 golang 이라고도 불리우는 Go 프로그래밍 언어는 구글의 V8 Javascript 엔진 개발에 참여했던 Robert Griesemer, Bell Labs에서 유닉스 개발에 참여했던 Rob Pike, 그리고 역시 Bell Labs에서 유닉스 개발했으며 C 언어의 전신인 B 언어를 개발했던 Ken Thompson이 함께 개발하였다.

* 

***Robert Griesemer*** 스위스 쮜리히에서 컴퓨터 박사 학위를 받고 구글에서 V8 자바스크립트 엔진 개발에 참여하였다. Go 프로그래밍 언어의 초기 디자인에 참여한 구글 엔지니어.

* 

***Rob Pike*** 벨 연구소에서 유닉스 개발팀에 근무하였으며 특히 분산 OS인 Plan 9 개발로 유명하다. 또한 켄 톰슨과 함께 UTF-8 을 만든 개발자이다. 현재는 구글에서 Go 개발자로 근무.

* 

***Ken Thompson*** 미국 컴퓨터공학의 개척자로 알려진 Ken은 대부분의 시간을 벨 연구소에서 보냈다. 초창기 Unix OS를 설계하고 구현하였으며, C 언어의 전신인 B 언어를 개발하였다. 현재는 구글에서 Go 프로그래밍 언어 개발에 참여하고 있다.

**Go 프로그래밍 언어의 특성**

Go는 전통적인 컴파일, 링크 모델을 따르는 범용 프로그래밍 언어이다. Go는 일차적으로 시스템 프로그래밍을 위해 개발되었으며, C++, Java, Python의 장점들을 뽑아 만들어졌다. C++와 같이 Go는 컴파일러를 통해 컴파일되며, 정적 타입 (Statically Typed)의 언어이다. 또한 Java와 같이 Go는 Garbage Collection 기능을 제공한다. Go는 단순하고 간결한 프로그래밍 언어를 지향하였는데, Java의 절반에 해당하는 25개의 키워드만으로 프로그래밍이 가능하게 하였다. 마지막으로 Go의 큰 특징으로 Go는 Communicating Sequential Processes (CSP) 스타일의 Concurrent 프로그래밍을 지원한다.

[source](http://golang.site/go/article/1-Go-%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D-%EC%96%B8%EC%96%B4-%EC%86%8C%EA%B0%9C)

# 1-1. 설치 및 환경설정

작성자가 mac용 pc만 사용중이여서 다른 os의 경우 깊은 내용이 없습니다. 설치 및 환경설정은 구글느님의 힘으로 충분히 극복이 가능하실거라는 판단하에 대략적으로 작성합니다..

# 1. 설치

    golang을 설치하기 위해서는 [설치파일 다운로드](https://golang.org/dl/) 에서 각 OS별 설치파일을 다운로드한 후 설치를 진행하는 방법과 소스파일을 다운로드 하여서 사용자 설정에 맞춰서 설치하는 방법이 존재한다.

    아래는 각 OS별 설치에 대한 설명이다.

    1-1. Mac

### brew를 이용한 설치

                    brew가 설정되어 있다는 가정하에 설명한다.

            brew install go

### pkg파일을 이용한 설치

[pkg파일로 mac에 설치하기](http://golang.site/go/article/202-Mac%EC%97%90-Go-%EC%84%A4%EC%B9%98%ED%95%98%EA%B8%B0)

## 1-2. Linux

[Linux에 설치](http://golang.site/go/article/201-%EB%A6%AC%EB%88%85%EC%8A%A4%EC%97%90-Go-%EC%84%A4%EC%B9%98%ED%95%98%EA%B8%B0)

## 1-3. Window

                상단의 메인사이트에서 msi파일을 다운로드 하여 실행하여 설치를 진행한다.

## 2. 환경설정

    golang은 설치 후 환경설정이 필요하다. 이때 필요한 환경설정변수는 GOROOT와 GOPATH이다.

    GOROOT의 경우 설치파일을 이용한 설치나 brew를 이용한 설치를 하게되면 자동으로 설정이 잡힌다. 하지만 GOPATH는 사용자가 직접설정을 잡아줘야 한다.

    GOPATH를 설정하는 이유는 golang에서는 작업(개발)할 폴더를 따로 지정을 해주어야 한다. 그래서 이 폴더를 잡아주는 부분이 필요하다.

### 2-1. Mac, Linux

        export GOPATH=자신이 원하는 폴더를 full path로 작성

### 2-2. Window

        환경변수를 설정하는 창에서 GOPATH를 추가한다.

## 3. 개발툴

    golang개발을 위해 사용할 수 있는 툴은 많이 있다. IDE의 양대산맥인 IntelliJ(community, full)과 eclipse도 지원되며 최근에 Jetbrain에서 오픈한 [gogland](https://www.jetbrains.com/go/) 도 있다. IDE가 싫은 사람은 vi환경에서도 개발이 가능하며([vim-go](https://github.com/fatih/vim-go)) 기타 에디터급 툴인 서브라임 텍스트,아톰,vscode로도 golang관련 플러그인 셋팅으로 개발이 가능하다. 그리고 golang용 개발IDE인 [Liteide](https://github.com/visualfc/liteide) 도 있다. 셋팅이나 툴설치등은 검색으로 충분히 커버가 가능하니 이부분은 생략하겠음.

# 1-2. golang의 재미있고 골때리는 사항들 정리

﻿우선적으로 생각나는 사항들에 대하여 정리하였습니다. 차후 추가되는 부분은 다시 업데이트할 예정입니다.

1. golang의 시작지점은 main package의 main함수이다.

    패키지명이 main 인 경우, 컴파일러는 해당 패키지를 공유 라이브러리가 아닌 실행(executable) 프로그램으로 만든다. 그리고 이 main 패키지 안의 main() 함수가 프로그램의 시작점 즉 Entry Point가 된다.

# 2. golang에는 접근제어자가 없다.

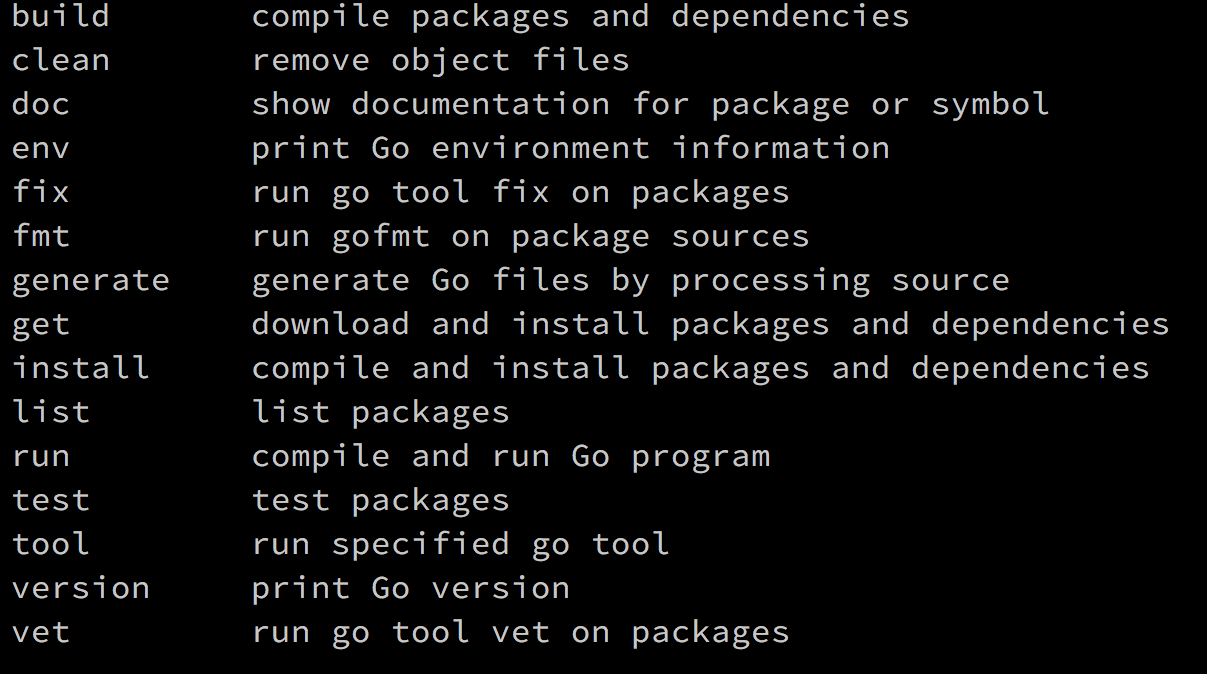
    golang에는 접근제어자(public,private..)등이 없다. 정확히 말하자면 없다기 보다는 지시어로 사용하는 접근제어자가 만들어져 있지 않다. 그래서 golang에서는 특이하고 골때리는 방법으로 접근제어를 하는데, 함수명이 소문자로 시작하면 private이고 대문자로 시작하면 public가 된다. 이런 사항은 golang에서 사용하는 변수,struct등 모든 사항에 해당된다.

    func getData() {}        //private function  
    func GetData() {}        //public function

# 3. gofmt의 존재.

    golang실행문 중에 gofmt라는 것이 존재한다. 이녀석은 어찌보면 어처구니 없고 어찌보면 고마운 녀석인데, 이녀석이 하는 일은 golang으로 작성된 파일을 formatting해주는 역활을 한다. gofmt는 go 프로그램을 읽은 후 들여쓰기, 한줄길이, 수직정렬,주석정렬등을 해준다.

[gofmt 참고](https://golang.org/cmd/gofmt/)



# 4. 세미콜론(;) 생략

    golang 프로그램 작성시 세미콜론은 입력하지 않는다. 대신 golang의 구문분석기가 알아서 내부적으로 세미콜론을 만들어준다. 몇가지 규칙이 내부적으로 존재하는데 새로운 라인이 시작되었을때 앞라인의 마지막 끝나는 부분이 (++,  --, ), })로 끝나면 자체적으로 세미콜론을 해당 구분자 뒤에 삽입해준다. 또는 내장함수나 명령어등의 뒤에서도 자체적으로 세미콜론을 생성해준다. 그래서 우리가 개발소스를 작성할 시에는 세미콜론을 입력하지 않아도 된다. 여기서 한가지 문제는 제어문(if문,for문)등을 사용할때가 있다. 개인 취향이지만 몇몇 사람은 if문이나 for문등의 첫문장이 끝날때 중괄호( { ) 를 새로운 라인에 작성하는 사람이 있다. 이렇게 작성되어지면 golang은 내부적으로 중괄호 다음에 세미콜론을 만들어주기 때문에 오류가 발생한다. 그래서 gofmt에서 자체적으로 이렇게 작성된 구문을 다시 정렬해주는 역활을 해준다.

# 5. class와 상속이 없다.

    golang에는 class와 상속이 없다. 그래서 처음 golang가 나왔을때 이 부분에 대한 논란이 많았다. class와 상속이 없어서 객체지향적인 코딩이 불가능하다는 판단이 많았지만 golang은 이 부분을 충분히 보완할 수 있게 준비해 두었다. 이 부분은 뒤에서 자세히 설명하겠다.

# 6. 포인터가 존재한다. (포인터는 잘몰라서 golang에서 사용하는 부분만 기술)

    golang에서는 포인터 연산은 없지만 포인터가 존재한다. 그래서 값을 넘겨주고 받을때 포인터를 이용한 처리가 가능하다.

# 7. 예외(Exception) 가 없다.

    golang는 예외가 없다. C언어와 같이 반환 값이 에러인지 아닌지를 비교하는 방법으로 에러를 처리한다. 대신 에러만을 전문적으로 처리하는 error 타입을 내장하고 있다. Go 프로그램은 error 값을 검사하는 것으로 에러 상태를 확인 할 수 있다.

# 8. 제네릭(generics)를 지원하지 않는다.

# 9. 애매한 함수형 프로그래밍

    어느정도 특징이 있기는 하지만 지양점은 아니다.

# 10. 유닛테스트가 쉽다.

    golang 내부적으로 유닛테스트를 제공하기 때문에 간단하게 작성하고 테스트할 수 있다. 방법은 테스트할 파일명(myapp\_test.go)이라고 만들고 테스트할 함수명앞에 Test를 붙여주고 인자값으로 (t \*testing.T)를 넘겨주고 cmd에서 go test를 실행하면 확인이 가능하다.

# 11. godoc을 이용하여 쉽게 문서를 만들어낼 수 있다.

    godoc 명령어를 이용하여 작성한 golang프로그램을 문서화할 수 있다. 문서화 작업을 하기위해서는 선행작업이 필요한데 이때 반듯이 필요한 부분이 주석이다. 최상단 package 선언문 위에 해당 패키지에 대한 주석을 멀티라인 주석문(/\*  \*/)을 이용해 기술하고 각 함수 또는 변수등에 대한 주석을 기술하면 godoc은 이 주석들을 판단하여 문서로 만들어준다.

[godoc 참고](https://godoc.org/golang.org/x/tools/cmd/godoc)

## 12. 사용하지 않는 변수,import문등이 있으면 컴파일 에러가 발생한다.

    변수를 선언 (var a = 1) 한 후 이 변수를 사용하지 않을 경우 golang에서는 컴파일 에러를 발생시킨다. 이 상황은 import문도 마찬가지이다. 선언된 import문중 사용되지 않는 경우가 있으면 컴파일 에러가 발생하니 확인이 필요하다.

## 13. golang의 예약 키워드

break default func interface select

case defer go map struct

chan else goto package switch

const fallthrough if range type

continue for import return var

## 14. go tools

* go build : 실행파일을 만들기 위한 명령어이다. cross compile를 위해서는 cmd에서 GOOS=컴파일하고자 하는 OS명(window,linux,mac) go build 명령어를 이용하여 빌드가 가능하다.

**1-3. 기본문법**

1. package 및 package 관리

    아래는 golang의 기본 폴더 구조이다. golang에서는 패키지 관리를 위해 내부적으로 지원하는 명령어가 있다. 명령어는 go get gitURL (예 : go get github.com/mattn/go-sqlite3)를 이용하여 사용할 수 있다.

    go get을 사용하기 위해서는 인터넷이 되는 환경이여야 하며 외부 패키지를 다운받기 위해서는 해당 패키지가 github, bitbucket 또는 직접 구성한 git서버가 있어야 한다.

    이 말인즉슨 go에서 패키지 프로그램은 프로젝트 저장소 단위로 관리되어진다는 말이다.

    ├── pkg //패키지의 소스코드를 빌드해서 만들어진 라이브러리 파일(.a - ar archive 파일)이 위치

    │

    ├── src //패키지의 소스코드가 위치

    |

    └── bin //패키지가 main 함수를 포함할 경우 실행 파일이 만들어 지는데, 이들 실행파일이 복사됨

<https://github.com/coma333ryu/utils> 해당 링크에 샘플용 라이브러리 파일을 올려두었다. 해당 내용을 go 패키지관리 시스템으로 다운로드 하는 방법은 아래와 같다.

1. go get github.com/coma333ryu/utils 명령어로 라이브러리를 로컬로 당겨온다.

2. GOPATH 아래에 /pkg/darwin\_amd64/github.com/coma333ryu/utils.a 가 다운로드되었는지 확인한다.

3. 사용하고자 하는 라이브러리를 소스에서 import한 후 해당 라이브러리의 함수를 사용해 본다.

package main

import (

"github.com/coma333ryu/utils"

)

func main() {

println(utils.AddNum(1, 2, 3, 4, 5))

}

<https://github.com/coma333ryu/exampleGolang/blob/master/basic0/basic0.go>

* golang 패키지 import시 표준 패키지는 GOROOT/pkg 에서 그리고 사용자 패키지나 3rd Party 패키지의 경우 GOPATH/pkg 에서 패키지를 찾게 된다.
* init()함수 사용가능하고, init 함수는 패키지가 로드되면서 실행되는 함수로 별도의 호출 없이 자동으로 호출된다.
* import에 alias를 사용가능함.

package main

import (

mongo "other/mongo/db"

mysql "other/mysql/db"

\_ "other/xlib" //xlib패키지 내부에 init함수가 있을시 init함수만 사용되어지고 나머지는 무시하겠다는 의미

)

func main() {

mondb := mongo.Get()

mydb := mysql.Get()

//...

}

2. 기본골격

   package main

   import "fmt"

   func main() {  
 fmt.Println("Hell World")  
 }

* package

    모든 go파일은 package선언으로 부터 시작되어진다. go 프로그램은 실행 프로그램과 라이브러리 두가지 종류로 분류된다.

    실행 프로그램으로 만들기 위해서는 반드시 package main으로 선언되어야 한다.

* import

    외부 package를 사용하기위해 쓴다.

    라이브러리 타입의 go코드를 사용하기위해 사용된다.

    여러 라이브러리를 import할때는 아래와 같이 사용한다.

    import (  
 "testing"  
 "fmt"  
    )

* func

    함수선언을 위해 사용된다.

    func ==> 함수선언

    main ==> 함수명

    () ==> 함수 인자값

    리턴값 ==> 인자값 옆에 공백을 주고 리턴값을 선언할 수 있음.

**3. go 프로그램 실행**

    go run 실행할 파일명.go 로 파일을 실행한다. (go run helloworld.go)

**4. 변수, 상수**

* 변수

package main

import "fmt"

func main() {

var a string = "initial"

fmt.Println(a)

var b, c int = 1, 2

fmt.Println(b, c)

var d = true        //타입추론

fmt.Println(d)

var e int

fmt.Println(e)

        //Short Assignment Statement ( := ) 라고 부르며 var를 생략하고 변수생성시 사용

//함수 내부에서만 사용가능

f := "short"

fmt.Println(f)

}

* 상수

package main

import "fmt"

const c int = 10

const s string = "Hi"

// const c = 10

// const s = "Hi"

const (

Visa = "Visa"

Master = "MasterCard"

Amex = "American Express"

)

//상수값을 0부터 순차적으로 부여하기 위해 iota 라는 identifier를 사용할 수 있다.

const (

Apple = iota // 0

Grape // 1

Orange // 2

)

func main() {

fmt.Println("Apple", Apple)

fmt.Println("Grape", Grape)

}

**5. 데이터 타입**

boolean 타입

bool

String 타입

string: string은 한번 생성되면 수정될 수 없는 Immutable 타입임

정수형 타입

int int8 int16 int32 int64

uint uint8 uint16 uint32 uint64 uintptr

Float 및 복소수 타입

float32 float64 complex64 complex128

기타 타입

byte: uint8과 동일하며 바이트 코드에 사용

rune: int32과 동일하며 유니코드 코드포인트에 사용한다

* 문자열 (``)와 ("")

    Back Quote (` `)로 둘러 싸인 문자열은 Raw String Literal이라 부른다. Raw String 그대로의 값을 갖는다.

    Double Quote (" ")로 둘러 싸인 문자열은 Interpreted String Literal이라 부른다. 복수 라인에 걸쳐 쓸 수 없으며, 인용부호 안의 Escape 문자열들은 특별한 의미로 해석된다.

package main

import "fmt"

func main() {

rawLiteral := `aaaaa\n

bbbb\n

cccc`

interLiteral := "dddd\neeee"

fmt.Println("rawLiteral", rawLiteral)

fmt.Println("interLiteral", interLiteral)

}

* 타입변환

var iVal int = 100

var uintVal uint = uint(iVal)

var floatVal float32 = float32(iVal)

println(floatVal, uintVal)

str := "ABC"

bytes := []byte(str)

str2 := string(bytes)

println(bytes, str2)

<https://github.com/coma333ryu/exampleGolang/blob/master/basic1/basic1.go>

**6. 조건문, 반복문**

* 조건문 (if)

package main

import "fmt"

func main() {

//()가 없어도 된다.

if 7%2 == 0 {

fmt.Println("7 is even")

} else {

fmt.Println("7 is odd")

}

//if문 내부에서 간단한 실행문을 사용할 수 있음.

//num값의 scope는 해당 if문 내에서만 사용가능.

if num := 9; num < 0 {

fmt.Println(num, "is negative")

} else if num < 10 {

fmt.Println(num, "has 1 digit")

} else {

fmt.Println(num, "has multiple digits")

}

// println("num======>", num)

}

* 조건문(switch)

package main

import (

"fmt"

"time"

)

func main() {

//case문에 break문이 필요없음. Go 컴파일러가 자동으로 break 문을 각 case문 블럭 마지막에 추가하시 때문임.

//fallthrough문을 작성하여 아래의 casse문들을 실행 할 수 있음.

i := 2

fmt.Print("Write ", i, " as ")

switch i {

case 1:

fmt.Println("one")

fallthrough

case 2:

fmt.Println("two")

fallthrough

case 3:

fmt.Println("three")

}

//case문에 여러개의 표현식을 사용할 수 있음.

switch time.Now().Weekday() {

case time.Saturday, time.Sunday:

fmt.Println("It's the weekend")

default:

fmt.Println("It's a weekday")

}

//case문에 표현식을 작성 할 수 있음.

score := 100

switch {

case score >= 90:

println("A")

case score >= 80:

println("B")

case score >= 70:

println("C")

case score >= 60:

println("D")

default:

println("No Hope")

}

//switch문에 표현식이 없을 수도 있음.

//이러한 경우 switch가 true라 생각하고 첫번째 case문을 실행함.

t := time.Now()

switch {

case t.Hour() < 12:

fmt.Println("It's before noon")

default:

fmt.Println("It's after noon")

}

//다른 언어의 switch는 일반적으로 변수의 값을 기준으로 case로 분기하지만, Go는 그 변수의 Type에 따라 case로 분기할 수 있다.

whatAmI := func(i interface{}) {

switch t := i.(type) {

case bool:

fmt.Println("I'm a bool")

case int:

fmt.Println("I'm an int")

default:

fmt.Printf("Don't know type %T\n", t)

}

}

whatAmI(true)

whatAmI(1)

whatAmI("hey")

}

* 반복문(for)

package main

import (

"fmt"

"time"

)

func main() {

//golang에서의 반복문은 for만 존재함.

//()는 사용하지 않음. 사용시 컴파일 에러발생함.

sum := 0

for i := 1; i <= 100; i++ {

sum += i

}

println("sum =========>", sum)

//조건식만 사용하여도 가능함. (타 언어의 while문과 같이 사용하기 위해)

num := 1

for num <= 3 {

fmt.Println(num)

num = num + 1

}

//무한루프

// for {

// println("Infinite loop")

// }

//for range문 (foreach와 비슷)

names := []string{"AAA", "BBB", "CCC"}

for index, name := range names {

println(index, name)

}

//index가 필요 없을시 \_문을 사용하여 무시할 수 있음.

for \_, name := range names {

println(name)

}

//break : for문을 빠져나올 때 사용

//continue : for 루프의 중간에서 나머지 문장들을 실행하지 않고 for 루프 시작부분으로 가려할 때 사용.

//goto : 기타 임의의 문장으로 이동하기 위해 사용.

var a = 1

for a < 15 {

if a == 5 {

a += a

continue // for루프 시작으로

}

a++

if a > 10 {

break //루프 빠져나옴

}

}

if a == 11 {

goto END //goto 사용예

}

println(a)

END:

println("End")

//break문이 레이블과 같이 사용되어 특정 레이블로 이동이 가능.

idx := 0

L1:

for {

if idx == 0 {

break L1

}

}

println("OK")

}

<https://github.com/coma333ryu/exampleGolang/blob/master/basic2/basic2.go>

**7. 함수**

* 함수

package main

//func : 함수 키워드

//Pass By Value 파라메터 전달방식

func say(msg string) {

println(msg)

}

//Pass By Reference

//\*string 과 같이 파라미터가 포인터임을 표시

//say2함수 내부의 msg는 문자열이 아니라 문자열을 갖는 메모리 영역의 주소값

func say2(msg \*string) {

println(\*msg)

\*msg = "World" //메시지 변경 (Dereferencing)

}

// Variadic Function (가변인자함수)

func say3(msg ...string) {

for \_, s := range msg {

println(s)

}

}

//함수에 리턴값이 있을 경우 리턴 타임을 파라메터() 다음에 기술한다.

//함수의 마지막에 return문과 함께 리턴값을 기술한다.

func sum(nums ...int) int {

sum := 0

for \_, num := range nums {

sum += num

}

return sum

}

func sum2(nums ...int) (int, int) {

s := 0

count := 0

for \_, n := range nums {

s += n

count++

}

return count, s

}

//Named Return Parameter : 리턴값들을 할당하여 리턴

//count와 total에 함수내부에서 직접 값을 할당함.

//return 문에는 아무 값들을 리턴하지 않지만, 그래도 리턴되는 값이 있을 경우에는 빈 return 문을 반드시 써 주어야 한다 (이를 생략하면 에러 발생).

func sum3(nums ...int) (count int, total int) {

for \_, n := range nums {

total += n

}

count = len(nums)

return

}

//재귀함수

func fact(n int) int {

if n == 0 {

return 1

}

return n \* fact(n-1)

}

func main() {

msg := "Hello"

say(msg)

//변수앞에 & 부호를 붙이면 msg 변수의 주소를 표시

//msg 변수의 주소를 전달

say2(&msg)

println("======", msg) //변경된 메시지 출력

say3("This", "is", "a", "book")

say3("Hi")

total := sum(1, 7, 3, 5, 9)

println("total ========>", total)

count, total := sum2(1, 7, 3, 5, 9)

println("count and total ======> ", count, total)

count1, total1 := sum3(1, 7, 3, 5, 9)

println("count1 and total1 ======> ", count1, total1)

println("fact(7)", fact(7))

}

<https://github.com/coma333ryu/exampleGolang/blob/master/basic3/basic3.go>

* 익명함수

package main

func calc(f func(int, int) int, a int, b int) int {

result := f(a, b)

return result

}

func main() {

//익명함수 : 함수 이름이 없는 함수

sum := func(n ...int) int {

s := 0

for \_, i := range n {

s += i

}

return s

}

result := sum(1, 2, 3, 4, 5)

println(result)

//변수 add 에 익명함수 할당

add := func(i int, j int) int {

return i + j

}

//add 함수 전달

r1 := calc(add, 10, 20)

println(r1)

//직접 첫번째 파라미터에 익명함수를 정의함

r2 := calc(func(x int, y int) int { return x - y }, 10, 20)

println(r2)

}

<https://github.com/coma333ryu/exampleGolang/blob/master/basic4/basic4.go>

* 클로저
* 함수 바깥에 있는 변수를 참조하는 함수값(function value)를 일컫는데, 이때의 함수는 바깥의 변수를 마치 함수 안으로 끌어들인 듯이 그 변수를 읽거나 쓸 수 있게 된다.

package main

func nextValue() func() int {

i := 0

return func() int {

i++

return i

}

}

func main() {

next := nextValue()

println(next()) // 1

println(next()) // 2

println(next()) // 3

anotherNext := nextValue()

println(anotherNext()) // 1 다시 시작

println(anotherNext()) // 2

}

<https://github.com/coma333ryu/exampleGolang/blob/master/basic5/basic5.go>

**8. 배열, 슬라이스**

* 배열

package main

func main() {

var a [3]int

a[0] = 1

a[1] = 2

a[2] = 3

println(a[0])

var a1 = [3]int{1, 2, 3}

var a3 = [...]int{1, 2, 3} //배열크기 자동으로

println(a1[1])

println(a3[2])

var multiArr = [2][3]int{

{1, 2, 3},

{4, 5, 6}, //끝에 콤마 추가

}

println("multiArr", multiArr[1][2])

}

* 슬라이스
* 배열을 기본으로 만들어진 좀더 기능이 많은 배열.
* 고정된 크기를 미리 지정하지 않을 수 있고, 차후 그 크기를 동적으로 변경할 수도 있고, 또한 부분 배열을 발췌할 수도 있다.
* 연속된 메모리공간을 활용하는 것이라 용량에 제한이 있다.

package main

import "fmt"

func main() {

var slice1 []int //슬라이스 변수 선언

slice1 = []int{1, 2, 3} //슬라이스에 리터럴값 지정

slice1[1] = 10

fmt.Println(slice1)

println(len(slice1), cap(slice1))

//내장함수 make를 이용하여 슬라이스 생성

//make 인자 ==> make(슬라이스 타입, 슬라이스 길이, 슬라이스 내부 배열의 최대 길이)

slice2 := make([]int, 5, 10)

println(len(slice2), cap(slice2)) // len 5, cap 10

//슬라이스에 별도의 길이와 용량을 지정하지 않으면, 기본적으로 길이와 용량이 0 인 슬라이스를 만드는데, 이를 Nil Slice 라 하고, nil 과 비교하면 참을 리턴한다.

var nilSlice []int

if nilSlice == nil {

println("Nil Slice")

}

println(len(nilSlice), cap(nilSlice))

num := []int{1, 2, 3, 4, 5}

fmt.Println("num =======> ", num)

fmt.Println("len =======> ", len(num))

fmt.Println("cap =======> ", cap(num))

//append() ==> 슬라이스에 새로운 요소 추가.

//append(슬라이스,추가할 값)

//배열은 고정된 크기로 그 크기 이상의 데이타를 임의로 추가할 수 없지만, 슬라이스는 자유롭게 새로운 요소를 추가할 수 있다.

//슬라이스 용량(capacity)이 아직 남아 있는 경우는 그 용량 내에서 슬라이스의 길이(length)를 변경하여 데이타를 추가하고,

//용량(capacity)을 초과하는 경우 현재 용량의 2배에 해당하는 새로운 Underlying array 을 생성하고

//기존 배열 값들을 모두 새 배열에 복제한 후 다시 슬라이스를 할당한다.

num = append(num, 6)

fmt.Println("new num =======> ", num)

fmt.Println("new len =======> ", len(num))

fmt.Println("new cap =======> ", cap(num))

//슬라이스에 다른 슬라이스 추가하기.

sliceA := []int{1, 2, 3}

sliceB := []int{4, 5, 6}

sliceA = append(sliceA, sliceB...)

//sliceA = append(sliceA, 4, 5, 6)

fmt.Println("sliceA ======> ", sliceA)

//슬라이스 복사

sliceC := make([]int, len(sliceA))

copy(sliceC, sliceA)

fmt.Println("copy sliceC ====> ", sliceC)

//부분 슬라이스(Sub-slice)

//슬라이스[처음인덱스:마지막인덱스], 마지막인덱스는 원하는 인덱스+1

        //num[:3] ==> 첫 인덱스(0번째)부터 3번째까지 : 1,2,3 (마지막인덱스는 원하는 인덱스+1 을 사용)

//처음 인덱스가 생략되면 0 이, 마지막 인덱스가 생략되면 그 슬라이스의 마지막 인덱스가 자동 대입된다.

sliced1 := num[:3]

fmt.Println("sliced1 =======> ", sliced1)

fmt.Println("sliced1 len =======> ", len(sliced1))

fmt.Println("sliced1 cap =======> ", cap(sliced1))

sliced3 := sliced1[:4]

fmt.Println("sliced3 =======> ", sliced3)

fmt.Println("sliced3 len =======> ", len(sliced3))

fmt.Println("sliced3 cap =======> ", cap(sliced3))

}

<https://github.com/coma333ryu/exampleGolang/blob/master/basic6/basic6.go>

**9. map**

* 해시테이블(Hash table)을 구현한 자료구조
* map[Key타입]Value타입으로 선언

package main

import (

"fmt"

)

func main() {

//map를 생성하기 위해서는 make()함수를 사용해야 한다.

var idMap = make(map[int]string)

idMap[0] = "Zero"

idMap[1] = "One"

idMap[2] = "Two"

fmt.Println("idMap", idMap)

//리터럴(literal)을 사용한 초기화

idMap2 := map[int]string{

0: "Zero Val",

1: "One Val",

2: "Two Val",

}

fmt.Println("idMap2", idMap2)

//map 데이터 읽기

fmt.Println("idMap[0]", idMap[0])

fmt.Println("idMap2[1]", idMap2[1])

//map 삭제

//delete(삭제할 map,삭제할 key value)

delete(idMap, 1)

fmt.Println("after delete idMap", idMap)

//map key의 존재유무

// val : 해당 map의 value, exists : 해당 key의 존재유무(true,false)

// val, exists := idMap2[3]

\_, exists := idMap2[3]

if !exists {

println("No idMap2[3] key")

}

//for range를 이용한 map데이터 읽기

for key, val := range idMap2 {

fmt.Println("idMap2 range", key, val)

}

}

<https://github.com/coma333ryu/exampleGolang/blob/master/basic7/basic7.go>

**9. struct**

* 필드들로 이루어진 타입을 갖는 컬렉션이다.
* 메서드가 없다.
* 메서드는 없지만 receiver를 이용하여 메서드를 구현할 수 있다.
* 타언어에서의 class와 유사한 역활을 한다.

package main

import (

    "fmt"

)

type person struct {

    name string

    age  int

}

type rect struct {

    width, height int

}

//person 생성자

func ㅜnewPerson(newName string, newAge int) \*person {

    return &person{name: newName, age: newAge}

}

//golang용 메서드 value receiver

//rect가 복사되어 전달된다.

func (r rect) area() int {

    r.width = 20

    return r.width \* r.height

}

//golang용 메서드 포인터 receiver

//reat의 포인터가 전달된다.

func (r \*rect) area2() int {

    r.width++

    return r.width \* r.height

}

//struct는 mutable이고, 다른 함수의 파라메터로 전달시 Pass by Value에 따라 객체를 복사해서 전달하게 된다.

func changePersonVal(personStruct person) person {

    personStruct.name = "New Person"

    personStruct.age = 15

    return personStruct

}

//그래서 Pass by Reference로 struct를 전달하고자 한다면, struct의 포인터를 전달해야 한다.

func changePersonVal2(personStruct \*person) \*person {

    personStruct.name = "New Person"

    personStruct.age = 15

    return personStruct

}

func main() {

    //struct 생성

    firstPerson := person{}

    secondPerson := person{"Second Person", 11}

    //내장함수 new를 이용한 struct 생성

    //new를 이용하여 struct생성하면 반환되는 struct는 포인터값(\*person)이 된다.

    thirdPerson := new(person)

    forthPerson := person{"Forth Person", 14}

    newPerson := ㅜnewPerson("New Person", 40)

    //person에 필드값 할당

    firstPerson.name = "First Person"

    firstPerson.age = 10

    thirdPerson.name = "Third Person"

    thirdPerson.age = 12

    fmt.Println("firstPerson", firstPerson)

    fmt.Println("secondPerson", secondPerson)

    fmt.Println("thirdPerson", \*thirdPerson)

    fmt.Println("newPerson", newPerson)

    newPerson1 := changePersonVal(forthPerson)

    fmt.Println("forthPerson 111====>", forthPerson)

    fmt.Println("newPerson1", newPerson1)

    newPerson2 := changePersonVal2(&forthPerson)

    fmt.Println("newPerson2", newPerson2)

    fmt.Println("forthPerson 222====>", forthPerson)

    println("========================= start method =========================")

    //reat struct가 복사되어 전달되기 때문에 원천 reat의 값에 영향을 주지 않는다.

    rect := rect{10, 20}

    area := rect.area()

    fmt.Println("rect width and area", rect.width, area)

    //reat 포인터가 전달되기 때문에 원천 reat가 변경되어진다.

    area2 := rect.area2()

    fmt.Println("rect width and area new ", rect.width, area2)

}

<https://github.com/coma333ryu/exampleGolang/blob/master/basic8/basic8.go>

**10. interface**

* 메서드들의 집합
* 메서드의 구현은 반듯이 struct를 이용해 구현된다.
* 다른 interface에 상속이 가능하다.
* interface는 하나의 타입으로 변수와 매개변수로 사용될 수 있다.
* 빈 인터페이스(interface{})를 사용가능하며, 빈 인터페이스 안에 어떠한 타입이라도 담아서 사용가능하다.

package main

import "math"

import "fmt"

//Shape Interface 구현

//type명령어를 사용하고 interface라고 작성하면 됨.

// 메서드들의 집합

type Shape interface {

area() float64

perimeter() float64

}

//Rect 정의

type Rect struct {

width, height float64

}

//Circle 정의

type Circle struct {

radius float64

}

func (r Rect) area() float64 {

return r.width \* r.height

}

func (r Rect) perimeter() float64 {

return 2 \* (r.width + r.height)

}

func (c Circle) area() float64 {

return math.Pi \* c.radius \* c.radius

}

func (c Circle) perimeter() float64 {

return 2 \* math.Pi \* c.radius

}

//Interface를 함수의 파라메터로 받을 수 있다.

func useShapeInterface(shape Shape) {

fmt.Println("Single Shape", shape)

fmt.Println("Single area method", shape.area())

fmt.Println("Single perimeter method", shape.perimeter())

}

//멀티 파라메터도 가능함.

func useMultiShapeInterface(shapes ...Shape) {

for \_, shape := range shapes {

areaMethod := shape.area()

perimeterMethod := shape.perimeter()

fmt.Println("Multi Shape", shape)

fmt.Println("Multi area method", areaMethod)

fmt.Println("Multi perimeter method", perimeterMethod)

}

}

func printEmptyInterface(v interface{}) {

fmt.Println(v)

}

func main() {

rect := Rect{width: 3, height: 5}

circle := Circle{radius: 4}

useShapeInterface(rect)

useShapeInterface(circle)

useMultiShapeInterface(rect, circle)

var x interface{}

x = 1

x = "Tom"

printEmptyInterface(x)

}

<https://github.com/coma333ryu/exampleGolang/blob/master/basic9/basic9.go>

**11. error**

* 내장 타입으로 error 라는 interface 타입을 갖는다.

package main

import "errors"

import "fmt"

//사용자 정의용 Error struct

type paramError struct {

param int

msg string

}

//Error method 구현

func (e \*paramError) Error() string {

return fmt.Sprintf("%d - %s", e.param, e.msg)

}

//errors package를 이용한 error처리

func checkFunction1(param int) (int, error) {

if param == 42 {

return -1, errors.New("It is 42")

}

return param + 3, nil

}

//사용자 정의 paramError struct를 이용한 error처리

func checkFunction2(param int) (int, error) {

if param == 42 {

return -1, &paramError{param, "This param is error."}

}

return param + 3, nil

}

func main() {

for \_, i := range []int{7, 42} {

if r, e := checkFunction1(i); e != nil {

fmt.Println("checkFunction1 failed:", e)

} else {

fmt.Println("checkFunction1 worked:", r)

}

}

for \_, i := range []int{7, 42} {

if r, e := checkFunction2(i); e != nil {

fmt.Println("checkFunction2 failed:", e)

} else {

fmt.Println("checkFunction2 worked:", r)

}

}

\_, e := checkFunction2(42)

if ae, ok := e.(\*paramError); ok {

fmt.Println(ae.param)

fmt.Println(ae.msg)

}

}

<https://github.com/coma333ryu/exampleGolang/blob/master/basic10/basic10.go>

**12. defer && panic && recover**

* defer는 특정 문장 혹은 함수를 나중에 (defer를 호출하는 함수가 리턴하기 직전에) 실행하게 한다.
* panic은 현재 함수를 즉시 멈추고 현재 함수에 defer 함수들을 모두 실행한 후 즉시 리턴한다. 이러한 panic 모드 실행 방식은 상위함수에도 똑같이 적용되고, 계속 콜스택을 타고 올라가며 적용된다. 그리고 마지막에는 프로그램이 에러를 내고 종료하게 된다.
* recover는 panic 함수에 의한 패닉상태를 다시 정상상태로 되돌리는 함수이다.

package main

import (

"fmt"

"os"

)

func firstStart() {

fmt.Println("First Start Function")

}

func secondStart() {

fmt.Println("Second Start Function")

}

func thirdStart() {

defer fmt.Println("Third End Function")

fmt.Println("Third Start Function")

}

func openFile(fn string) {

//panic 함수에 의한 패닉상태를 다시 정상상태로 되돌리는 함수

defer func() {

if r := recover(); r != nil {

fmt.Println("OPEN ERROR", r)

}

}()

f, err := os.Open(fn)

//현재 함수를 즉시 멈추고 현재 함수에 defer 함수들을 모두 실행한 후 즉시 리턴

if err != nil {

panic(err)

}

defer f.Close()

}

func main() {

thirdStart()

//특정 문장 혹은 함수를 나중에 (defer를 호출하는 함수가 리턴하기 직전에) 실행

defer secondStart()

firstStart()

openFile("Invalid.txt")

fmt.Println("Done")

}

<https://github.com/coma333ryu/exampleGolang/blob/master/basic11/basic11.go>

**13. goroutine**

* Go 런타임이 관리하는 경량의 실행 스레드이다.
* "go" 키워드를 사용하여 함수를 호출하면, 런타임시 새로운 goroutine을 실행한다.
* goroutine은 비동기적으로 함수루틴을 실행하므로, 여러 코드를 동시에 실행하는데 사용된다.
* 익명함수에 대해 사용할 수도 있다.

package main

import (

"fmt"

"sync"

"time"

)

func say(s string) {

for i := 0; i < 10; i++ {

fmt.Println(s, "\*\*\*", i)

}

}

func main() {

say("Sync")

go say("Async1")

go say("Async2")

go say("Async3")

time.Sleep(time.Second \* 3)

var wait sync.WaitGroup

wait.Add(2)

go func() {

defer wait.Done()

fmt.Println("Hello")

}()

go func(msg string) {

defer wait.Done()

fmt.Println(msg)

}("Hi")

wait.Wait()

}

<https://github.com/coma333ryu/exampleGolang/blob/master/basic12/basic12.go>

**14. channel**

* goroutine들 사이 데이타를 주고 받는데 사용된다.
* 한 채널이 준비될 때까지 다른 채널에서 대기함으로써 별도의 lock을 걸지 않고 데이타를 동기화하는데 사용된다.
* 기본채널은 동기적 수행.
* close()함수를 사용하여 채널을 닫을 수 있다. 채널을 닫게 되면, 해당 채널로는 더이상 송신을 할 수 없지만, 채널이 닫힌 이후에도 계속 수신은 가능하다.

package main

import "fmt"

func main() {

// c := make(chan int)

//수신루틴이 없으므로 데드락

// c <- 1

// fmt.Println(<-c)

//버퍼채널 생성

//make(chan type, N) : N는 생성할 버퍼의 갯수입력.

ch := make(chan int, 1)

//수신자가 없더라도 보낼 수 있다.

ch <- 101

fmt.Println(<-ch)

ch2 := make(chan int, 2)

// 채널에 송신

ch2 <- 1

ch2 <- 2

// 채널을 닫는다

close(ch2)

// 채널 수신

println(<-ch2)

println(<-ch2)

        //채널 수신시 return값은 채널 메세지와, 닫힘여부(true/false)이다.

\_, success := <-ch2

if !success {

println("더이상 데이타 없음.")

}

//채널값을 가져올 때 range문 사용가능.

// for channelMsg := range ch2 {

// println(channelMsg)

// }

}

**14-1. Unbuffered Channel, Buffered Channel**

* Unbuffered Channel : 하나의 수신자가 데이타를 받을 때까지 송신자가 데이타를 보내는 채널에 묶여 있게 된다.
* Buffered Channel : 수신자가 받을 준비가 되어 있지 않을 지라도 지정된 버퍼만큼 데이타를 보내고 계속 다른 일을 수행할 수 있다.
* 버퍼채널은 비동기적 수행.

package main

import "fmt"

func main() {

// c := make(chan int)

//수신루틴이 없으므로 데드락

// c <- 1

// fmt.Println(<-c)

//버퍼채널 생성

//make(chan type, N) : N는 생성할 버퍼의 갯수입력.

ch := make(chan int, 1)

//수신자가 없더라도 보낼 수 있다.

ch <- 101

fmt.Println(<-ch)

}

**14-2. 채널 송수신 방향**

* 채널을 함수의 파라미터도 전달할 때, 일반적으로 송수신을 모두 하는 채널을 전달하지만, 특별히 해당 채널로 송신만 할 것인지 혹은 수신만할 것인지를 지정할 수도 있다.
* 송신 파라미터는 (p chan<- int)와 같이 chan<- 을 사용하고, 수신 파라미터는 (p <-chan int)와 같이 <-chan 을 사용한다.
* 만약 송신 채널 파라미터에서 수신을 한다거나, 수신 채널에 송신을 하게되면, 에러가 발생한다.

package main

import "fmt"

func main() {

ch := make(chan string, 1)

sendChan(ch)

receiveChan(ch)

}

//송신채널 파라메터

func sendChan(ch chan<- string) {

ch <- "Data"

// x := <-ch // 에러발생

}

//수신채널 파라메터

func receiveChan(ch <-chan string) {

data := <-ch

fmt.Println(data)

}

**14-3. select**

* 복수 채널들을 기다리면서 준비된 (데이타를 보내온) 채널을 실행하는 기능을 제공한다.
* 여러 개의 case문에서 각각 다른 채널을 기다리다가 준비가 된 채널 case를 실행하는 것이다.
* case 채널들이 준비되지 않으면 계속 대기하게 되고, 가장 먼저 도착한 채널의 case를 실행한다.
* 만약 복수 채널에 신호가 오면, Go 런타임이 랜덤하게 그 중 한 개를 선택한다.
* 하지만, select문에 default 문이 있으면, case문 채널이 준비되지 않더라도 계속 대기하지 않고 바로 default문을 실행한다.

package main

import (

"fmt"

"time"

)

func main() {

/\*

c1 := make(chan string)

c2 := make(chan string)

go func() {

time.Sleep(time.Second \* 1)

c1 <- "one"

}()

go func() {

time.Sleep(time.Second \* 2)

c2 <- "two"

}()

for i := 0; i < 2; i++ {

select {

case msg1 := <-c1:

fmt.Println("received", msg1)

case msg2 := <-c2:

fmt.Println("received", msg2)

}

}

\*/

c1 := make(chan string, 1)

go func() {

time.Sleep(time.Second \* 2)

c1 <- "result 1"

}()

select {

case res := <-c1:

fmt.Println(res)

case <-time.After(time.Second \* 1):

fmt.Println("timeout 1")

}

c2 := make(chan string, 1)

go func() {

time.Sleep(time.Second \* 2)

c2 <- "result 2"

}()

select {

case res := <-c2:

fmt.Println(res)

case <-time.After(time.Second \* 3):

fmt.Println("timeout 2")

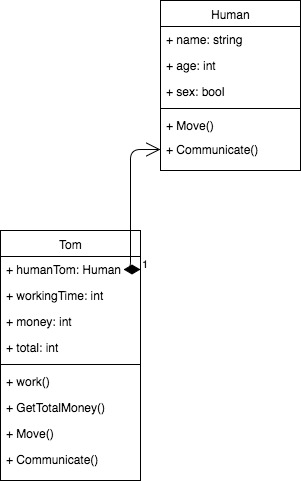
}

}

**1-4. golang의 상속**

**Golang 객체지향 프로그래밍.**

* golang에는 클래스, 객체, 상속을 지원하지 않는다.
* 하지만 composition을 이용하여 상속을 구현할 수 있다.
* 즉 embedding방식으로 상속을 구성할 수 있다.



* 참고자료 : <https://golangkorea.github.io/post/go-start/object-oriented/>

<https://github.com/coma333ryu/exampleGolang/tree/master/oop>

**Todo List**

**TodoGolang**

* 간단한 Todo List를 golang와 golang의 템플릿을 이용하여 구현.
* 구조

├── README.md

├── common

│   ├── constants                     //상수

│   │   └── constants.go

│   ├── databases                     //db connection && init

│   │   └── sqliteDb

│   │   └── sqlite3\_init.go

│   ├── exceptions                    //error 처리

│   │   └── todo\_error.go

│   ├── logger                        //로그 처리

│   │   └── todo\_logger.go

│   ├── routes                        //라우터 설정

│   │   └── todo\_routes.go

│   └── validaters                    //validation

│   └── validater.go

├── templates                         //view template

│   └── todo.html

├── todo                              //todo business 처리

│   ├── controller                    // controller

│   │   └── todo\_controller.go

│   ├── model                         // tb\_todo table에 대한 dto

│   │   └── todo\_dto.go

│   └── services                      // interface와 구현체

│   ├── impl

│   │   └── todo\_impl.go

│   └── todo\_service.go

├── todo.db                           // sqlite3 db file

├── todo\_main.go                      // main

└── web                               // static folder

└── assets

├── css

│   └── main.css

└── js

└── jquery-3.2.0.min.js

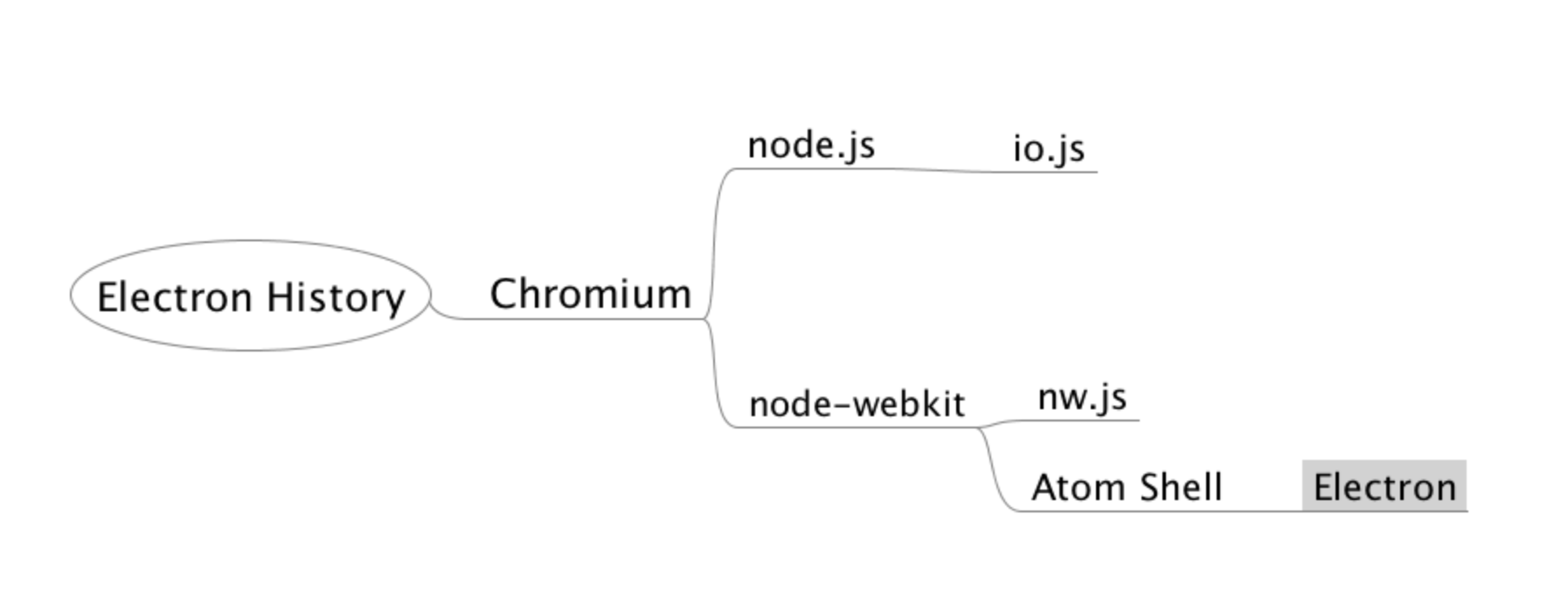
* <https://github.com/coma333ryu/todoGolang>

**2. Electron**

**1. Electron?**

* github에서 만든 HTML, CSS 및 JavaScript로 크로스 플랫폼 데스크톱 응용 프로그램을 만들기 위한 오픈 소스 라이브러리.

**2. History**



**3. 설치**

* node install :  <http://www.libqa.com/wiki/743>
* electron install : npm -g install electron-prebuilt

**4. 시작**

* 만들고자 하는 디렉토리를 생성하거나 생성된 디렉토리로 이동하여 npm init명령어로 package.json파일을 만들어서 시작한다.
* electron 프로세스가 시작시 처음 실행되는 자바스크립트 파일은 package.json 에 "main"으로 정의되어지며 기본적으로는 index.js를 바라보게 된다.

mkdir hellow-electron

cd hellow-electron

npm init

$ npm init

This utility will walk you through creating a package.json file.

It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields

and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and

save it as a dependency in the package.json file.

Press ^C at any time to quit.

name: (hellow-electron)

version: (1.0.0)

description: hellow electron

entry point: (index.js)

test command:

git repository:

keywords: electron

author: coma333ryu

license: (ISC) MIT

About to write to /Users/yoonjeongbu/Coma/dev/electrons/hellow-electron/package.json:

{

"name": "hellow-electron",

"version": "1.0.0",

"description": "hellow electron",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [

"electron"

],

"author": "coma333ryu",

"license": "MIT"

}

Is this ok? (yes)

* package.json의 main에 정의된 자바스크립트 파일을 생성하고 아래의 기본 샘플소스를 붙여 넣는다.

vi index.js

const electron = require('electron')

// Module to control application life.

const app = electron.app

// Module to create native browser window.

const BrowserWindow = electron.BrowserWindow

const path = require('path')

const url = require('url')

// Keep a global reference of the window object, if you don't, the window will

// be closed automatically when the JavaScript object is garbage collected.

let mainWindow

function createWindow () {

  // Create the browser window.

  mainWindow = new BrowserWindow({width: 800, height: 600})

  // and load the index.html of the app.

  mainWindow.loadURL(url.format({

    pathname: path.join(\_\_dirname, 'index.html'),

    protocol: 'file:',

    slashes: true

  }))

  // Open the DevTools.

  mainWindow.webContents.openDevTools()

  // Emitted when the window is closed.

  mainWindow.on('closed', function () {

    // Dereference the window object, usually you would store windows

    // in an array if your app supports multi windows, this is the time

    // when you should delete the corresponding element.

    mainWindow = null

  })

}

// This method will be called when Electron has finished

// initialization and is ready to create browser windows.

// Some APIs can only be used after this event occurs.

app.on('ready', createWindow)

// Quit when all windows are closed.

app.on('window-all-closed', function () {

  // On OS X it is common for applications and their menu bar

  // to stay active until the user quits explicitly with Cmd + Q

  if (process.platform !== 'darwin') {

    app.quit()

  }

})

app.on('activate', function () {

  // On OS X it's common to re-create a window in the app when the

  // dock icon is clicked and there are no other windows open.

  if (mainWindow === null) {

    createWindow()

  }

})

// In this file you can include the rest of your app's specific main process

// code. You can also put them in separate files and require them here.

* electron 프로세스가 실행후 index.js를 실행하면서 최초 로딩하는 html을 설정(mainWindow.loadURL)하는데 현재는 index.html이므로 해당 html파일을 생성한다.

vi index.html

<!DOCTYPE html>

<html>

  <head>

    <meta charset="UTF-8">

    <title>Hello World!</title>

  </head>

  <body>

    <h1>Hello World!</h1>

    <!-- All of the Node.js APIs are available in this renderer process. -->

    We are using Node.js <script>document.write(process.versions.node)</script>,

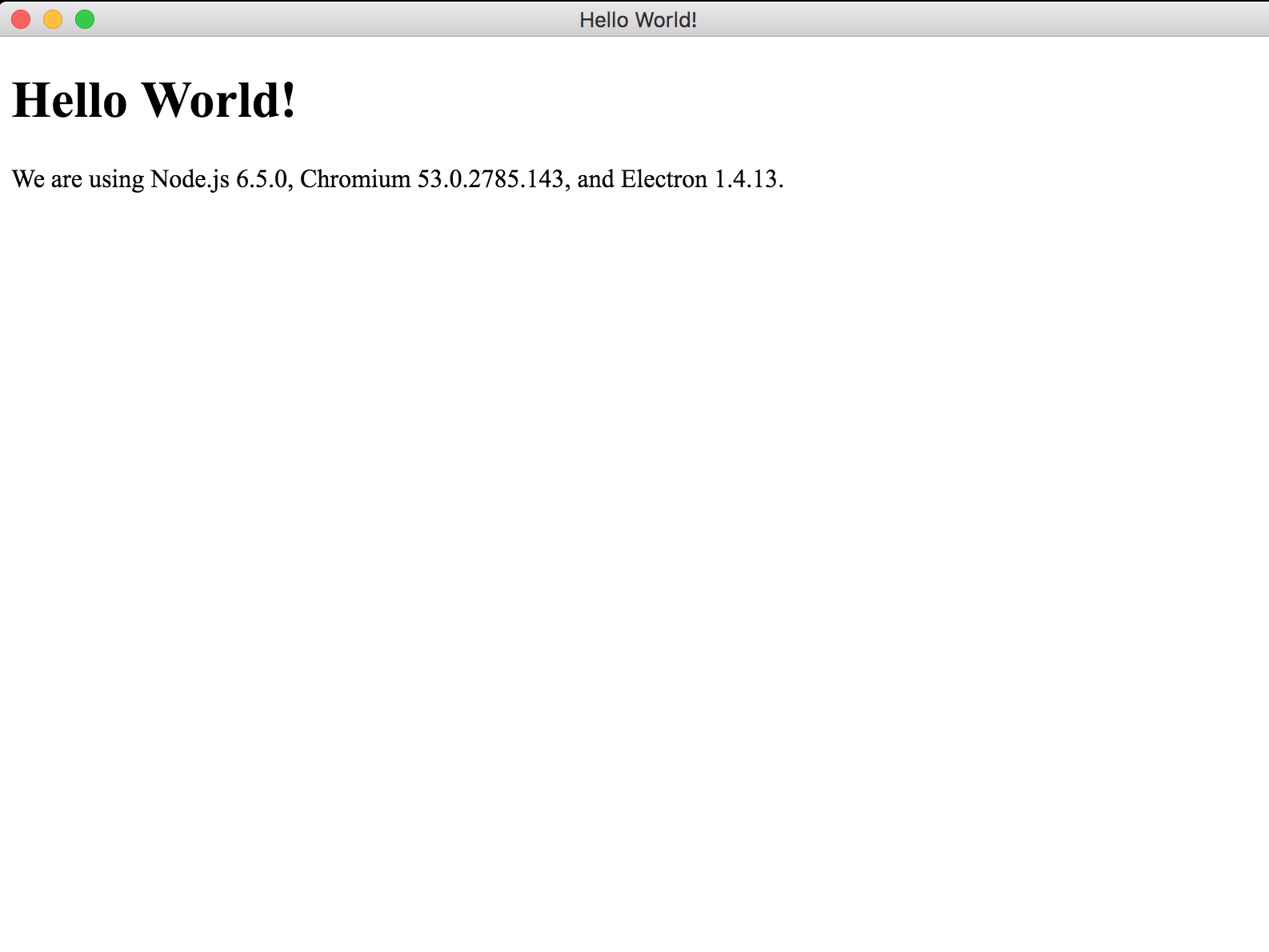
    Chromium <script>document.write(process.versions.chrome)</script>,

    and Electron <script>document.write(process.versions.electron)</script>.

  </body>

</html>

* electron . 명령어를 실행하여 electron이 실행되면 OK.



**5. 참고**

<https://github.com/electron/electron/tree/master/docs-translations/ko-KR>

샘플 : <https://github.com/coma333ryu/todoElectron>