7장. 인터페이스

2019년 10월 31일 목요일 오전 6:21

개요

인터페이스는 타입이다.
다른 타입의 동작을 일반화 하거나 추상화해서 표현한다.

Go 의 인터페이스는 단순히 필요한 메소드만 있으면 충분하다.
구인 광고의 Job Description 에 비유하면 적절하다 싶다.

백엔드 개발자 모집 - 파이썬 장고로 서버 만들 수 있는 사람.	회계 담당자 모집 - 급여 정산 가능한 사람

type Djangoer interface {
MakeServer(ip string, port int) *Server
}
	type Accountant interface {
	CalcSalary(employID string, workDay int) int
	}

장고 개발자가 되고 싶다.

```
func (m 나) MakeServer(ip string, port int) *Server {
	fmt.Println("사실 만들줄 모르지롱")
	return &Server{}
}
```

공부할 내용

1. 인터페이스 타입과 그 값의 기본 메커니즘
2. 표준 라이브러리의 주요 인터페이스 보기
3. 타입 단언 (type assertion)과 타입 변경 (type switch) 를 이용한 일반화 방법

7.1 인터페이스 규약

인터페이스는 추상 타입 Abstract type
다른 타입들은 구상 타입 Concrete type

```
func Fprintf(w io.Writer, format string, args ...interface{}) (int, error)
```

io.Writer 즉, 쓸줄 아는 놈이면 된다.
아래에 보면 알겠지만 정확히는
- 바이트 슬라이스인 p 를 받아서, int 인 n 과 error 인 err 을 리턴해줄 수 있는 놈이기만 하면 된다.

그래서 놈! 인거다. Writer

```
package io
// Writer is the interface that wraps the basic Write method.
type Writer interface {
	// Write writes len(p) bytes from p to the underlying data stream.
	// It returns the number of bytes written from p (0 <= n <= len(p))
	// and any error encountered that caused the write to stop early.
	// Write must return a non-nil error if it returns n < len(p).
	// Write must not modify the slice data, even temporarily.
	//
	// Implementations must not retain p.
	Write(p []byte) (n int, err error)
}
```

Write() 할 수 있는 Writer 이기만 하면 된다? 대체 가능하다는 말이다. Substitutability
장고 개발자 한 명 퇴사해도 구인광고 내어서 장고 서버 만들 수 있는 다른 사람으로 대체 가능하다는 거다.

bytecounter 예제: https://play.golang.org/p/nQS3XZqE8JI

```go
// Copyright © 2016 Alan A. A. Donovan & Brian W. Kernighan.
// License: https://creativecommons.org/licenses/by-nc-sa/4.0/

// See page 173.

// Bytecounter demonstrates an implementation of io.Writer that counts bytes.
package main

import (
    "fmt"
)

//!+bytecounter

type ByteCounter int

func (c *ByteCounter) Write(p []byte) (int, error) {
    *c += ByteCounter(len(p)) // convert int to ByteCounter
    return len(p), nil
}

//!-bytecounter

func main() {
    //!+main
    var c ByteCounter
    c.Write([]byte("hello"))
    fmt.Println(c) // "5", = len("hello")

    c = 0 // reset the counter
    var name = "Dolly"
    fmt.Fprintf(&c, "hello, %s", name)
    fmt.Println(c) // "12", = len("hello, Dolly")
    //!-main
}
```

- 언제 주소값을 쓰고, 언제 그냥 값을 쓰는가?
- Write() 메쏘드는 정확히는 ByteCounter 타입의 메쏘드가 아니고, *ByteCounter 의 메쏘드이다.

Exercise 7.1
Exercise 7.2
Exercise 7.3

7.2 인터페이스 타입

1) 인터페이스: 이런이런 메쏘드가 있어야 해
2) 구상타입에 해당 메쏘드를 구현한다
3) 알았어 너는 해당 인터페이스의 인스턴스로 인정하겠어

Reader 인터페이스는 읽을 수 있는 놈이어야 한다는 것
    = Read(p []byte) (n int, err error) 메쏘드가 있는 놈

Closer 인터페이스는 닫을 수 있는 놈이어야 한다는 것
    = Close() error 메쏘드가 있는 놈

node.js 개발자 인터페이스는 node.js 로 서버 구축할 수 있는 놈
react 개발자 인터페이스는 react 로 프론트앤드 개발할 수 있는 놈이여야 한다.

type 노드리액트풀스택개발자 interface {
    node.js 개발자
    react 개발자
}

Exercise 7.4
Exercise 7.5

7.3 인터페이스 충족

어떤 타입이, 어떤 인터페이스가 요구하는 모든 메쏘드가 있으면
타입이 인터페이스를 충족한다고한다. Interface Satisfaction

is a

- 어떤 구상 타입은 어떤 인터페이스의 한 종류이다. 인터페이스를 충족한다.
- 어떤 인터페이스를 만족하는 구상타입은 많을 수 있다는 것이다.

당연한 이야기일 수 있겠지만
IntSet 타입과 *IntSet 타입은 다른 것이다.

```
type IntSet struct { /* ... */ }
func (*IntSet) String() string
var _ = IntSet{}.String() // compile error: String requires *IntSet receiver

but we can cal l it on an IntSet variable: -> 이건 알아서 해준거다.
var s IntSet
var _ = s.String() // OK: s is a variable and &s has a String method

However, since only *IntSet has a String method, only *IntSet satisfies the fmt.Stringer interface:
var _ fmt.Stringer = &s // OK
var _ fmt.Stringer = s // compile error: IntSet lacks String method
```

어떤 구상타입이 Read, Write, Close 가 모두 가능하더라도
Reader 가 되었다면 Write, Close 는 할 수 없다.

만약 내가 백엔드, 프론트엔드, 웹퍼블리싱이 가능하더라도
백엔드 엔지니어로 취업하고 그에 맞게 급여를 받는다면
회사가 나에게 프론트엔드, 웹퍼블리싱 업무를 요구할 수 없다.

빈 인터페이스에는 무슨 타입이든 할당할 수 있다.
- https://play.golang.org/p/PWf3ru_cGGW

그러면 뭐하는가? 할 수 있는게 없다. -> 따라서 다시 값을 특정 타입으로 단언하고 (assertion) 빼내야 한다. ex) 이 빈 인터페이스는 정수 타입입니다.
https://play.golang.org/p/7aNrWPiS9Wf

7.4 flag.Value 로 플래그 분석

책과는 반대로 접근해보자

실행

```
$ go build gopl.io/ch7/tempflag
$ ./tempflag
20°C
$ ./tempflag -temp -18C
-18°C
$ ./tempflag -temp 212°F
100°C
$ ./tempflag -temp 273.15K
invalid value "273.15K" for flag -temp: invalid temperature "273.15K"
Usage of ./tempflag:
  -temp value
        the temperature (default 20°C)
$ ./tempflag -help
Usage of ./tempflag:
  -temp value
        the temperature (default 20°C)
```

1) tempflag 라고 실행하면 기본값인 20도가 나온다

2) -temp 플래그를 걸고 온도값과 단위값을 적으면 섭씨로 계산되어 나온다
    - 화씨든 섭씨든 마찬가지

3) 단위값이 엉뚱하면 에러가 뜨고, Usage 가 나온다

4) -help 옵션을 넣으면 사용법이 나온다

tempflag 라는 프로그램의 구현

```
var temp = tempconv.CelsiusFlag("temp", 20.0, "the temperature")

func main() {
    flag.Parse()
    fmt.Println(*temp)
}
```

1) tempconv.CelsiusFlag() 라는 함수를 실행하고 그 결과값을 temp 함수에담는다.

2) flag.Parse() 를 실행했을때에 *temp 에는 결과값이 나온다.

그럼 tempconv 패키지의 CelsiusFlag() 함수는 무얼까?

```
// CelsiusFlag defines a Celsius flag with the specified name,
// default value, and usage, and returns the address of the flag variable.
// The flag argument must have a quantity and a unit, e.g., "100C".
func CelsiusFlag(name string, value Celsius, usage string) *Celsius {
    f := celsiusFlag{value}
    flag.CommandLine.Var(&f, name, usage)
    return &f.Celsius
}
```

1) name, value, usage 를 파라미터로 받아서 *Celsius 타입을 회신해주는 녀석이다.
    - name 은 플래그의 이름
    - value 는 플래그를 먹이지 않았을 경우의 디폴트값
    - usage 는 잘못된 사용이나, -help 시의 사용법 소개이다.

2) 실제 구현을 보자
    - celciusFlag 라는 구조체를 초기값을 넣어서 하나 생성하고
    - flag.CommandLine.Var() 라는 함수를 통해 플래그를 등록한다.

https://golang.org/src/flag/flag.go?s=29585:29633#L852
- 그리고 리턴값은 f.Celsius 의 주소값이다.

3) flag.CommandLine.Var() 함수를 유추해보면
- 플래그의 이름을 등록해준다. 사용법을 등록해준다.
- 플래그가 쓰이지 않으면 기본값인 value 를 담고 있게 된다
- 플래그가 쓰이면 clesiusFlag 타입의 Set() 함수가 값을 처리할 것이다.

gopl.io/ch7/tempconv

```
// *celsiusFlag satisfies the flag.Value interface.
type celsiusFlag struct{ Celsius }

func (f *celsiusFlag) Set(s string) error {
    var unit string
    var value float64
    fmt.Sscanf(s, "%f%s", &value, &unit) // no error check needed
    switch unit {
    case "C", "°C":
        f.Celsius = Celsius(value)
        return nil
    case "F", "°F":
        f.Celsius = FToC(Fahrenheit(value))
        return nil
    }
    return fmt.Errorf("invalid temperature %q", s)
}
```

Set() 함수가 어떻게 구현되어 있나 보자.

- fmt.Sscanf() 를 통해서 s 를 %f%s 로 구분하여 각각 value, unit 에 담고
- unit 을 기준으로 섭씨, 화씨이면 case 문으로 처리해서 f.Celsius 에 담아준다.
- 해당사항이 없으면 에러를 발생시킨다.
    - Set() 의 리턴값이 nil 이 아닐 경우 usage 가 출력되나 보다.

```
package flag

// Value is the interface to the value stored in a flag.
type Value interface {
    String() string
    Set(string) error
}
```

flag 의 Value 라는 녀석의 주소값이

1) flag.CommandLine.Var() 의 맨 처음 파라미터 인듯 싶다.
2) 그리고 Value 라는 인터페이스는
    - String() string
    - Set(string) error
    두 메쏘드가 구현되어 있어야 한다.

7.5 인터페이스 값

```
package main

import (
    "os"
    "fmt"
)

func main(){
    var w interface{}
    fmt.Printf("%T\n", w)

    w = 1
    fmt.Printf("%T\n", w)
    fmt.Printf("%d\n", w)

    w = 1.1
    fmt.Printf("%T\n", w)
    fmt.Printf("%f\n", w)

    w = os.Stdout
    fmt.Printf("%T\n", w)

    // <nil>
    // int
    // 1
    // float64
    // 1.100000
    // *os.File
}
```

두번째 읽으며 이해한 것은

1) 보통의 변수는 Type 이 컴파일시 정해지고 나면
    - 값은 동적으로 변하겠지만
    - Type 은 변하지 않는다는 것이다.

2) 하지만 interface 는 타입도 변한다.
    - Type 도 동적으로 변할 수 있고
    - Value 도 동적으로 변할 수 있다.

7.5.1 주의: nil 포인터가 있는 인터페이스는 ni 이 아니다.

var buf *bytes.Buffer 라고 선언하면

- type 은 *bytes.Buffer 이고
- value 는 nil 이 된다. 초기값이다.

이 인터페이스 변수의 값은 nil 이지만 변수는 nil 이 아니다.

즉 buf != nil 이다.

7.6 sort.Interface 로 정렬

1) sort.Interface 타입을 충족하려면, Len(), Less(), Swap() 함수가 구현되어 있어야 한다.
2) 그러고 나면 아래처럼 sort.Sort() 를 그 타입에 먹일 수 있다.

```go
package main

import (
    "sort"
    "fmt"
)

type StringSlice []string
func (p StringSlice) Len() int {return len(p)}
func (p StringSlice) Less(i, j int) bool {return p[i] < p[j]}
func (p StringSlice) Swap(i, j int)  { p[i], p[j] = p[j], p[i]}

func main(){
    s := []string {"hahaha", "no way", "good job", "hot summer"}

    fmt.Println(s)
    sort.Sort(StringSlice(s))
    fmt.Println(s)
}

// [hahaha no way good job hot summer]
// [good job hahaha hot summer no way]
```

구조체 슬라이스의 정렬의 경우는 포인터를 저장하는 경우 정렬을 위해 교환하는 값의 크기가 주소값으로 작아지므로 더 빨라진다.

그리고 심지어는 sort.Reverse() 를 먹이면 Less() 가 반대로 먹히는 타입을 정의한 셈이 되어
sort.Sort(sort.Reverse()) 를 먹이면 별도의 추가 메쏘드 구현 없이도 역정렬이 가능해진다.

구조체 안에 슬라이스와 정렬을 위한 Less 함수를 내장할 수도 있다.
- 슬라이스 변수의 정렬을 위해
- Less 함수를 정의해주면 된다.

Exercise 7.8
Exercise 7.9
Exercise 7.10

7.7 http.Handler 인터페이스

http.ListenAndServe() 부터 보자

1) ip/port 정보를 넣어주고
2) 거기로 들어오는 녀석들을 처리할 Handler 를 넣어주면 된다.

func ListenAndServe

```
func ListenAndServe(addr string, handler Handler) error
```

ListenAndServe listens on the TCP network address addr and then calls Serve with handler to handle requests on incoming connections. Accepted connections are configured to enable TCP keep-alives.

The handler is typically nil, in which case the DefaultServeMux is used.

ListenAndServe always returns a non-nil error.

▾ Example

```
package main

import (
        "io"
        "log"
        "net/http"
)

func main() {
        // Hello world, the web server

        helloHandler := func(w http.ResponseWriter, req *http.Request) {
                io.WriteString(w, "Hello, world!\n")
        }

        http.HandleFunc("/hello", helloHandler)
        log.Fatal(http.ListenAndServe(":8080", nil))
}
```

그러면 http.Handler 는 뭘까?

매우 간단한 인터페이스이다.
ServeHTTP() 라는 메쏘드가 있는 타입이면 http.Handler 인터페이스를 충족한다.

http.ResponseWriter 와 *http.Request 를 받아서 뭔가 알아서 하면 되는거다.

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

gopl.io/ch7/http1

```
func main() {
    db := database{"shoes": 50, "socks": 5}
    log.Fatal(http.ListenAndServe("localhost:8000", db))
}

type dollars float32

func (d dollars) String() string { return fmt.Sprintf("$%.2f", d) }

type database map[string]dollars

func (db database) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}
```

http1 예제는

database 라는 타입이 ServeHTTP() 메쏘드를 가지고 있다.
따라서 http.ListenAndServe() 의 파라미터중 http.Handler 자리에 들어갈 수 있는 것이다.

gopl.io/ch7/http2

```
func (db database) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    switch req.URL.Path {
    case "/list":
        for item, price := range db {
            fmt.Fprintf(w, "%s: %s\n", item, price)
        }
    case "/price":
        item := req.URL.Query().Get("item")
        price, ok := db[item]
        if !ok {
            w.WriteHeader(http.StatusNotFound) // 404
            fmt.Fprintf(w, "no such item: %q\n", item)
            return
        }
        fmt.Fprintf(w, "%s\n", price)
    default:
        w.WriteHeader(http.StatusNotFound) // 404
        fmt.Fprintf(w, "no such page: %s\n", req.URL)
    }
}
```

httpt2 예제는

좀더 기교를 부려본 것이다.

1) 들어온 request 들을 req.URL.Path 별로 나누고
2) 또 거기에서 Query 를 추출하여 거기에 맞춰 응답 하도록 (=w 에 쓰도록) 구현한 것이다.

gopl.io/ch7/http3

```
func main() {
    db := database{"shoes": 50, "socks": 5}
    mux := http.NewServeMux()
    mux.Handle("/list", http.HandlerFunc(db.list))
    mux.Handle("/price", http.HandlerFunc(db.price))
    log.Fatal(http.ListenAndServe("localhost:8000", mux))
}

type database map[string]dollars

func (db database) list(w http.ResponseWriter, req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}

func (db database) price(w http.ResponseWriter, req *http.Request) {
    item := req.URL.Query().Get("item")
    price, ok := db[item]
    if !ok {
        w.WriteHeader(http.StatusNotFound) // 404
        fmt.Fprintf(w, "no such item: %q\n", item)
        return
    }
    fmt.Fprintf(w, "%s\n", price)
}
```

mux 를 먹이면 좀더 깔끔해 지는것이 보일 것이다.

HandlerFunc() 가 재미있는 녀석인데, 이건 함수가 아니라 함수 타입이다.

- 타입은 메쏘드를 가질 수 있으며, 이 함수 타입은 ServeHTTP() 메쏘드를 가지고 있다.
- 따라서 mux.Handle() 에 쓰일 수 있는 것이다.

gopl.io/ch7/http4

```
func main() {
    db := database{"shoes": 50, "socks": 5}
    http.HandleFunc("/list", db.list)
    http.HandleFunc("/price", db.price)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}
```

자 이건 위에것과 뭐가 다른가?

1) http.NewServeMux() 로 mux 를 생성하지 않았다.
   - 패키지 안에 디폴트로 선언된 mux 를 쓰겠다는 것이다. (DefaultServeMux)
2) 이 경우에는 http.ListenAndServe 에 mux 정보를 줄 필요가 없는 것이다.
   - 내부를 보면 nil 인 경우엔 DefaultServemux 를 쓰라고 되어 있겠지?

Exercise 7.11
Exercise 7.12