Chapter4

배열 - 슬라이스 - 맵

배열 (Array)

- 원소의 타입이 동일하다.
- 원소를 순차적으로 저장한다. (순서가 있다)
- 원소를 저장할 수 있는 크기가 고정되어 있다.

배열 선언

```
// int 형 배열 선언
var intArr [3]int

// 각 인덱스에 값 추가
intArr[0] = 1
intArr[1] = 2
intArr[2] = 3
```

배열 선언 및 초기화 (배열 리터럴)

- 선언과 동시에 각 배열을 초기화 할 수 있다.
- [ ] 부분에 ...을 사용하면 초기화 하는 원소의 개수를 배열의 길이로 정하게 할 수 있다.

```
compile_langs := [2]string{"Go", "C"}

repl_langs := [...]string{"Scala", "Kotlin", "Swift"}
```

배열의 부분 초기화

- 배열은 선언 시 해당 타입으로 초기화 된다.
- 부분 초기화를 하게되면 선언하지 않은 부분은 기본값으로 초기화 된다.

```
languages := [3]string{0:"Java", 2:"Python"}
// languages[0] => Java
// languages[1] => ""(빈 문자열)
// languages[2] => Python
```

배열의 원소에 접근하기

배열 변수에 [ ] 연산자를 사용하여 접근하고자 하는 원소의 index 를 지정하면 원하는 원소에 접근이 가능하다.

```
arr := [5]int{1, 2, 3, 4, 5}
fmt.Println(arr[1]) // 2
```

배열의 포인터 원소에 접근하기

배열의 원소가 포인터인 경우에는 *연산자를 사용하여 접근이 가능하다.

```
ptrArr := [2]*int{new(int), new(int)}


*ptrArr[0] = 20
*ptrArr[1] = 30


fmt.Println(ptrArr[0])  // 0xc82005e028
fmt.Println(*ptrArr[0]) // 20
```

배열은 값(Value) 타입

- Go 언어는 배열을 값으로 취급한다.
- 배열을 서로 대입할 경우 원본 값이 복사된다.

```
var status [4]int
httpStatus := [4]int{200, 300, 400, 500}

status = httpStatus
// status : [200 300 400 500], httpStatus : [200 300 400 500]
status[0] = 201
// status : [201 300 400 500], httpStatus : [200 300 400 500]
```

배열 복사 (1)

- 배열은 동일한 크기일 경우에만 복사가 가능하다.
- 이는 크기와는 무관하다. (크기가 작은 배열을 큰 배열에 대입 할 수 없음)

```
var arr1 [3]int
arr2 := [2]int{200, 300}
arr1 = arr2 // compile error
// cannot use arr2 (type [2]int) as type [3]int in assignment
```

배열 복사 (2)

배열의 원소가 포인터일 경우, 해당 포인터가 복사된다.

```
var pokemons [3]*string
monsters := [3]*string{new(string), new(string), new(string)}
*monsters[0], *monsters[1], *monsters[2] = "피카츄", "꼬부기", "잠만보"

pokemons = monsters    // [0xc82000a330 0xc82000a340 0xc82000a350]
// pokemons, monsters : [피카츄 꼬부기 잠만보]

*monsters[2] = "파이리" // [0xc82000a330 0xc82000a340 0xc82000a350]
// pokemons, monsters : [피카츄 꼬부기 파이리]
```

다차원 배열

배열 리터럴을 사용하면 다차원 배열을 선언할 수 있다.

```
matrix := [2][2]int{{1,2}, {3,4}}
```

다차원 배열 역시 [ ]를 사용하여 해당 원소에 접근할 수 있다.

```
matrix[0][0] // 1
matrix[1][1] // 4
```

슬라이스 (Slice)

- 배열이 가지고 있는 기능은 기본적으로 제공한다.
- 배열과는 달리 동적으로 크기를 조절할 수 있다.
- 내부적으로는 배열의 포인터를 참조하고 있기 때문에 배열이 가지고 있는 문제에 대한 해결책이 될 수 있다.

슬라이스 소스 코드

src/runtime/slice.go

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

슬라이스 소스코드를 살펴보면, 내부적으로 **array**의 포인터를 참조하고 있음을 짐작 할 수 있다.

슬라이스 정의 (1)

배열과는 달리 []에 슬라이스 원소의 개수를 정의하지 않는다.

```
var slice []int
slice = make([]int, 10)
```

리터럴을 사용하면 슬라이스의 정의 와 초기화를 동시에 할 수 있다.

```
slice := []int{1, 2, 3, 4}
```

슬라이스 정의 (2)

- 슬라이스를 정의할 때 길이 뿐만 아니라 **용량** 도 지정할 수 있다.
- 용량은 슬라이스에 여분의 공간을 확보 할 수 있도록 해 준다.

```
slice := make([]int, 10, 20)
fmt.Println(len(slice)) // 10
fmt.Println(cap(slice)) // 20
```

슬라이스에 값 추가하기 (1)

- **append**함수를 사용하면 슬라이스에 원소를 추가할 수 있다.
- **append**함수는 슬라이스와 여러 원소를 인자로 받는다.
- **append**함수는 원소가 추가된 새로운 슬라이스를 반환한다.

```
slice := []int{}
slice = append(slice, 20, 30 40)
// [20 30 40]
```

슬라이스에 값 추가하기 (2)

- **append**를 사용하여 슬라이스에 다른 슬라이스를 추가할 수 있다.
- 이 경우, 추가되는 슬라이스 뒤에 **...**을 입력해야 한다.

```
slice1 := []int{1, 2, 3, 4, 5}
slice2 := []int{100, 200, 300}
slice1 = append(slice1, slice2...)
```

append와 용량의 관계

- 원본 슬라이스를 다른 슬라이스에 대입할 경우 두 슬라이스는 같은 원소를 공유한다.
- **appned**를 사용하여 원소를 추가하는 과정에서 용량을 초과할 경우, **append**는 기존 값을 복사한 새로운 슬라이스를 반환한다.
- 이 경우, 새로운 슬라이스는 원본 슬라이스와 더이상 같은 원소를 공유하지 않게 된다.
- 따라서 원본 슬라이스에서 기존 값을 변경하더라도 새로운 슬라이스에는 전혀 영향을 주지 못한다.

append와 용량의 관계

```
slice1 := make([]int, 2, 3)
slice1[0], slice[1] = 1, 2
slice2 := slice1
// slice1=>[1 2], len(2), cap(3), slice2=>[1 2], len(2), cap(3)
slice2[0] = 3
// slice1=>[3 2], len(2), cap(3), slice2=>[3 2], len(2), cap(3)
slice2 = append(slice2, 4, 5, 6)
// slice1=>[3 2], len(2), cap(3), slice2=>[3 2 4 5 6], len(5), cap(6)
slice2[0] = 100
// slice1=>[3 2], len(2), cap(3), slice2=>[100 2 4 5 6], len(5), cap(6)
```

슬라이스 복사

- **`[start:end]`** : start index 부터 end-1 index 까지 슬라이스를 잘라낸다.
- **`[start:end:max]`** 위와 동일하게 슬라이스를 잘라내는 대신 **`max - start`** 만큼의 용량을 설정한다.
- 단, **max**값은 원본 슬라이스의 용량값을 넘을 수 없다.

```
original := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
copyone := original[2:4:10]
// [3 4] len:2, cap:8
copy2 := original[2:4:11] // PANIC!
```

nil 슬라이스

- 선언만 하고 **make**나 리터럴로 초기화 하지 않으면 **nil** 슬라이스가 된다.
- **nil** 슬라이스는 내부 배열 포인터도 **nil** 이다.
- **nil** 슬라이스라도 **append()**, **len()**, **cap()** 함수는 사용할 수 있다.
- 초기화가 되어 있지 않기 때문에 **index**를 이용하여 값에 접근할 수 는 없다.

```
var slice []int // 슬라이스를 선언만 함.
fmt.Println(slice[0]) // PANIC!
```

Empty 슬라이스

- Empty 슬라이스는 초기화는 하지만 값이 아무것도 없는 슬라이스다.
- Empty 슬라이스는 내부 배열 포인터가 **nil**이 아니다.
- **length**가 **0**이기 때문에 **index**를 통해 값에 접근할 수 없다.
- 기능상으로는 **nil**슬라이스와 동일하다.

```
emptySlice1 := make([]int, 0) // 빈 슬라이스
emptySlice2 := []int{}        // 위와 동일
```

맵(Map)

- 키(key) : 값(value) 으로 구성되어 있다.
- 맵에 들어있는 데이터는 순서가 없다.

맵 선언

make 함수 또는 리터럴을 사용하여 맵을 선언하고 초기화 할 수 있다.

```
httpSuccess := make(map[int]string)
httpSuccess[200] = "Ok"
httpSuccess[302] = "Found"
//map[200:Ok 302:Found]

httpError := map[int]string{404:"NotFound", 502:"Bad Gateway"}
// map[404:NotFound 502:Bad Gateway]
```

맵에서 키와 값의 조건

- == 연산자를 이용한 비교가 가능한 타입이면 키로 사용이 가능하다.
- 슬라이스, 함수, 슬라이스를 가진 함수는 키로 사용이 불가능하다.
- 값에는 제한이 없다.

맵에서 값 가져오기 (1)

- 키를 이용하면 맵에서 값을 가져올 수 있다.
- 이 경우 해당 키에 대한 값과 키가 존재하는지에 대한 `Bool` 값을 반환한다.

```
value, exists := httpStatus[200]
if exists {
    // 해당 키가 존재한다.
}
```

맵에서 값 가져오기 (2)

- 반환값을 하나만 지정하면 해당 키에 대한 값을 반환한다.
- 해당 키에 대한 값이 없으면 **value** 에 해당하는 타입의 제로값을 반환한다.

```
code := httpStatus[200]

if code != "" {
    // code 의 타입은 문자열이므로
    // 제로값인 빈 문자열이 반환되었는지를 확인한다.
}
```

nil 맵

- 맵을 선언만 하고 초기화를 하지 않으면 **nil** 값을 갖게 된다.
- 이 경우에는 값을 추가할 수 없다.
- **nil** 맵에서도 키를 사용하여 값을 가져오는 동작은 가능하다.

```
var myMap map[string]string
myMap["KANG"] = "Kyungkoo" // PANIC!
value, exists := myMap["KANG"]
// value 는 빈 문자열
// exists 는 false
```

배열, 슬라이스, 맵 Loop

배열, 슬라이스, 맵은 **for**와 **range**를 사용하여 같은 방법으로 순회할 수 있다.
또한 기존의 **for** loop를 이용한 순회도 가능하다.

```
arr := [...]int{100, 200, 300}

for index, value := range arr {
}

for index := 0; index < len(arr); index ++ {
}
```

배열 Loop

배열은 **index**와 **value** 값을 얻는다.

```
arr := [...]stirng{"A", "B", "C"}

for index, value := range arr {
  fmt.Printf("%d:%v\n", index, value)
}
```

슬라이스 Loop

슬라이스는 **index**와 **value** 값을 얻는다.

```
slice := []string{"A", "B", "C"}

for index, value := range slice {
    fmt.Printf("%d:%v\n", index, value)
}
```

맵 Loop

맵은 key와 value 값을 얻는다.

```
dict := map[string]string{"A":"에이", "B":"비", "C":"씨"}

for key, value := range dict {
  fmt.Printf("%v:%v\n", key, value)
}
```