Chapter 15 문자열

날짜	@November 24, 2021
발표자	김윤서

15.1 문자열

문자열은 문자 집합, 타입명은 string, 큰따옴표나 백쿼트로 묶어서 표시.

백쿼트: 문자열 안의 특수 문자가 일반 문자처럼 처리

```
package main

import "fmt"

func main() {

  str1 := "Hello\t'World'\n"
  str2 := `Go is "awesome"!\nGo is simple and\t'powerful'`
  fmt.Println(str1)
  fmt.Println(str2)

  poet1 := "죽는 날까지 하늘을 우러러\n한 점 부끄럼이 없기를,\n잎새에 이는 바람에도\n나는 괴로워했다.\n"
  poet2 := `죽는 날까지 하늘을 우러러
한 점 부끄럼이 없기를,
잎새에 이는 바람에도
나는 괴로워했다.`

  fmt.Println(poet1)
  fmt.Println(poet2)

}
```

출력문

```
Hello   'World'

Go is "awesome"!\nGo is simple and\t'powerful'
죽는 날까지 하늘을 우러러
```

```
한 점 부끄럼이 없기를,
잎새에 이는 바람에도
나는 괴로워했다.

죽는 날까지 하늘을 우러러
한 점 부끄럼이 없기를,
잎새에 이는 바람에도
나는 괴로워했다.
```

- 큰따옴표: 특수 문자 인식 vs 백쿼트: 특수문자 그대로 출력
- 큰따옴표: 한 줄만 묶음 vs 백쿼트: 여러 줄 묶음
- 큰따옴표 사용해서 특수 문자를 문자 그대로 출력하고 싶을 때? 역슬래시 2번

15.1.1 UTF-8 문자코드

Go는 UTF-8 문자코드를 표준 문자코드로 사용.

UTF-8: 다국어 문자 지원, 문자열 크기 절약. 영문자, 숫자, 일부 특수 문자 1바이트 & 그외 2~3 바이트로 표현. ANSI 코드와 1:1 대응.

15.1.2 rune 타입으로 한 문자 담기

문자 하나를 표현하는 데 rune 타입 사용. UTF-8 한 글자 1~3 바이트 → 3바이트 제공 X, rune 타입은 4바이트 정수 타입인 int32 타입의 별칭 타입.

```
type rune int32
```

문자 한 개는 작은따옴표로 묶음

```
package main

import "fmt"

func main() {
  var char rune = '한'

  fmt.Printf("%T\n", char)
  fmt.Println(char)
  fmt.Printf("%c\n", char)
}
```

출력문

```
int32
54620
한
```

- %T: 타입 출력 → rune == int32

15.1.3 len()으로 문자열 크기 알아내기

문자열 크기 → len() 내장 함수. 크기는 문자 수가 아닌 문자열이 차지하는 메모리 크기

```
package main

import "fmt"

func main() {
  str1 := "가나다라마"
  str2 := "abcde"

  fmt.Printf("len(str1) = %d\n", len(str1))
  fmt.Printf("len(str2) = %d\n", len(str2))
}
```

출력문

```
len(str1) = 15
len(str2) = 5
```

- UTF-8 한글은 글자당 3바이트, 영문자는 글자당 1바이트

15.1.4 []rune 타입 변환으로 글자 수 알아내기

string 타입, rune 슬라이스 타입인 []rune 타입은 상호 타입 변환이 가능. 슬라이스는 길이가 변할 수 있는 배열.

```
package main

import "fmt"

func main() {
  str := "Hello World"
```

```
  // ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘ ‘, ‘W’, ‘o’, ‘r’, ‘l’, ‘d’ 문자코드 배열
  runes := []rune{72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100}

  fmt.Println(str)
  fmt.Println(string(runes))
}
```

출력문

```
Hello World
Hello World
```

- 문자열은 UTF-8 코드값의 배열인 rune 배열로 나타낼 수 있음.
- string 타입과 []rune 타입은 상호 타입 변환 가능. rune 배열의 각 요소에 문자열의 각 글자 대입

```
package main

import "fmt"

func main() {
  str := "hello 월드"
  runes := []rune(str)

  fmt.Printf("len(str) = %d\n", len(str))
  fmt.Printf("len(runes) = %d\n", len(runes))
}
```

출력문

```
len(str) = 12
len(runes) = 8
```

- len(str) = 12 → 문자열의 바이트 길이
- len(runes) = 8 → 배열의 요소 개수
- string: 연속된 바이트 메모리 vs []rune: 글자들의 배열

15.1.5 string 타입을 []byte로 타입 변환할 수 있다

string 타입과 []byte 타입은 상호 타입 변환 가능. []byte는 byte (1바이트 부호 없는 정수 타입)의 가변 길이 배열. 모든 문자열은 1바이트 배열로 변환 가능.

파일을 쓰거나 네트워크로 데이터 전송하는 경우 io.Writer 인터페이스 사용, io.Writer 인터페이스는 []byte 타입을 인수로 받음. → []byte로 변환해야 함.

15.2 문자열 순회

15.2.1 인덱스를 사용해 바이트 단위 순회하기

```
package main

import "fmt"

func main() {
  str := "Hello 월드!"

  for i := 0; i < len(str); i++ {
    //바이트 단위로 출력
    fmt.Printf("타입:%T 값:%d 문자값:%c\n", str[i], str[i], str[i])
  }
}
```

출력문

```
타입:uint8 값:72 문자값:H
타입:uint8 값:101 문자값:e
타입:uint8 값:108 문자값:l
타입:uint8 값:108 문자값:l
타입:uint8 값:111 문자값:o
타입:uint8 값:32 문자값:
타입:uint8 값:236 문자값:ì
타입:uint8 값:155 문자값:
타입:uint8 값:148 문자값:
타입:uint8 값:235 문자값:ë
타입:uint8 값:147 문자값:
타입:uint8 값:156 문자값:
타입:uint8 값:33 문자값:!
```

- 인덱스를 사용해 각 바이트값을 출력
- 요소의 타입 uint8 == byte, 1바이트 크기 영문자는 잘 표시되지만 3바이트 크기 한글은 깨져 표시

15.2.2 []rune으로 타입 변환 후 한 글자씩 순회하기

```
package main

import "fmt"

func main() {
  str := "Hello 월드!"
  arr := []rune(str)

  for i := 0; i < len(arr); i++ {
    fmt.Printf("타입:%T 값:%d 문자값:%c\n", arr[i], arr[i], arr[i])
  }
}
```

출력문

```
타입:int32 값:72 문자값:H
타입:int32 값:101 문자값:e
타입:int32 값:108 문자값:l
타입:int32 값:108 문자값:l
타입:int32 값:111 문자값:o
타입:int32 값:32 문자값:
타입:int32 값:50900 문자값:월
타입:int32 값:46300 문자값:드
타입:int32 값:33 문자값:!
```

- []rune으로 타입 변환 → 한 문자씩 이뤄진 배열
- len() 문자열 글자 개수 반환 → 문자 단위로 순회
- 별도의 배열을 할당 → 불필요한 메모리 사용

15.2.3 range 키워드를 이용해 한 글자씩 순회하기

```
package main

import "fmt"

func main() {
  str := "Hello 월드!"
  for _, v := range str {
    fmt.Printf("타입:%T 값:%d 문자:%c\n", v, v, v)
  }
}
```

출력문

```
타입:int32 값:72 문자:H
타입:int32 값:101 문자:e
타입:int32 값:108 문자:l
타입:int32 값:108 문자:l
타입:int32 값:111 문자:o
타입:int32 값:32 문자: 
타입:int32 값:50900 문자:월
타입:int32 값:46300 문자:드
타입:int32 값:33 문자:!
```

- range 이용한 문자열 순회, 인덱스값은 사용하지 않기 때문에 밑줄_로 무효화
- 모든 문자 타입 = int32 = rune
- 불필요한 메모리 낭비 X

15.3 문자열 합치기

+, += 연산으로 문자열을 이을 수 있다.

15.3.1 문자열 비교하기

연산자 ==, ≠ 로 문자열이 같은지 같지 않은지 비교한다.

15.3.2 문자열 대소 비교하기: >, <, ≤, ≥

>, <, ≤, ≥ 연산자로 문자열 간 대소를 비교한다. 첫 글자부터 하나씩 값을 비교해서 그 글자에 해당하는 유니코드값이 다를 경우 대소를 반환한다.

```
package main

import "fmt"

func main() {
  str1 := "BBB"
  str2 := "aaaaAAA"
  str3 := "BBAD"
  str4 := "ZZZ"

  fmt.Printf("%s > %s : %v\n", str1, str2, str1 > str2)
  fmt.Printf("%s < %s : %v\n", str1, str3, str1 < str3)
  fmt.Printf("%s <= %s : %v\n", str1, str4, str1 <= str4)
}
```

출력문

```
BBB > aaaaAAA : false
BBB < BBAD : false
BBB <= ZZZ : true
```

- "BBB"와 "aaaaAA" 대소 비교: B 66번 < a 97번
- "BBB"와 "BBAD" 대소 비교: B 66번 > A 65번
- "BBB"와 "ZZZ" 대소 비교: B 66번 < Z 90번
- 문자열 대소 비교 시 문자열 길이와 상관없이 앞글자부터 (같은 위치에 있는 글자끼리) 비교한다.

15.4 문자열 구조

15.4.1 string 구조 알아보기

string 타입은 GO 언어에서 제공하는 내장 타입 → reflect 패키지 안의 StringHeader 구조체를 통해 내부 구현 엿보기

```
type StringHeader struct{
  Data uintptr
  Len  int
}
```

- Data: uintptr타입, 문자열의 데이터가 있는 메모리 주소
- Len: int타입, 문자열의 길이

15.4.2 string끼리 대입하기

```
package main

import "fmt"

func main() {
  str1 := "안녕하세요. 한글 문자열입니다."
  str2 := str1

  fmt.Printf(str1)
  fmt.Printf("\n")
  fmt.Printf(str2)
}
```

출력문

```
안녕하세요. 한글 문자열입니다.
안녕하세요. 한글 문자열입니다.
```

- 구조체 변수가 복사될 때 구조체 크기만큼 메모리가 복사된다.
- str1의 Data와 Len값만 str2에 복사한다.

```
package main

import (
  "fmt"
  "reflect"
  "unsafe"
)

func main() {
  str1 := "Hello World!"
  str2 := str1

  stringHeader1 := (*reflect.StringHeader)(unsafe.Pointer(&str1)) // ❷ Data값 추출
  stringHeader2 := (*reflect.StringHeader)(unsafe.Pointer(&str2)) // ❸  Data값 추출

  fmt.Println(stringHeader1)
  fmt.Println(stringHeader2)
}
```

출력문

```
&{14050818 12}
&{14050818 12}
```

- st1의 모든 필드값이 str2에 복사되어 같은 메모리 데이터를 가리킨다.
- string → unsafe.Pointer → *reflect.StringHeader
- string 변수가 가리키는 문자열이 아무리 길어도 string 변수끼리 대입 연산에서는 16바이트 값만 복소될 뿐 문자열 데이터는 복사되지 않는다.

15.5 문자열은 불변이다

문자열은 불변이다. string 타입이 가리키는 문자열의 일부만 변경할 수 없다. 그래서 아래 코드는 컴파일 에러가 발생한다.

```
var str string = "Hello World"
str = "How are you?"  //가능
str[2] = 'a'          //Error!!
```

- str값을 다른 문자열로 바꾸기 → Data 포인터값, Len값 모두 변경
- 일부 변경 X

```
package main

import "fmt"

func main() {
  var str string = "Hello World"
  var slice []byte = []byte(str)

  slice[2] = 'a'

  fmt.Println(str)
  fmt.Printf("%s\n", slice)
}
```

출력문

```
Hello World
Healo World
```

- []byte 슬라이스로 타입 변환. str이 가리키는 메모리 공간과 slice가 가리키는 메모리 공간은 서로 다르다.
- slice의 요솟값 변경 → str 변경 X (주소가 서로 다름)
- Go 언어는 슬라이스로 타입 변환 시 문자열을 복사해 새로운 메모리 공간을 만들어 슬라이스가 가리키게 한다. 그래야 불변 원칙을 지킬 수 있다.

15.5.1 문자열 합산

string 타입 간 합 연산이 어떻게 일어나는지

```
package main

import (
  "fmt"
  "reflect"
  "unsafe"
)

func main() {
  var str string = "Hello"
  stringheader := (*reflect.StringHeader)(unsafe.Pointer(&str))
  addr1 := stringheader.Data

  str += " World"
  addr2 := stringheader.Data

  str += " Welcome!"
  addr3 := stringheader.Data

  fmt.Println(str)
  fmt.Printf("addr1:\t%x\n", addr1)
  fmt.Printf("addr2:\t%x\n", addr2)
  fmt.Printf("addr3:\t%x\n", addr3)
}
```

출력문

```
Hello World Welcome!
addr1:  4b63f5
addr2:  c00008c050
addr3:  c000092000
```

- 합 연산 후의 Data 필드값 즉 주소값이 모두 다르게 출력되었다
- Go언어는 기존 문자열 메모리 공간을 건드리지 않고 새로운 메모리 공간을 만들어서 두 문자열을 합치기 때문에 string 합연산 이후 주솟값이 변경된다. → 문자열 불변 원칙이 준수된다.
- string 합 연산을 빈번하게 하면 메모리가 낭비된다. → strings 패키지의 Builder를 이용해 메모리 낭비 줄일 수 있다.

```
package main

import (
  "fmt"
```

```
  "strings"
)

func ToUpper1(str string) string {
  var rst string
  for _, c := range str {
    if c >= 'a' && c <= 'z' {
      rst += string('A' + (c - 'a'))
    } else {
      rst += string(c)
    }
  }
  return rst
}

func ToUpper2(str string) string {
  var builder strings.Builder
  for _, c := range str {
    if c >= 'a' && c <= 'z' {
      builder.WriteRune('A' + (c - 'a'))
    } else {
      builder.WriteRune(c)
    }
  }
  return builder.String()
}

func main() {
  var str string = "Hello World"

  fmt.Println(ToUpper1(str))
  fmt.Println(ToUpper2(str))
}
```

출력문

```
HELLO WORLD
HELLO WORLD
```

- ToUpper1() 함수: 합 연산을 사용해 문자를 더한다. Go 언어 내부에서는 합 연산을 사용할 때마다 새로운 메모리 공간을 할당하여 두 문자열을 더함 → 메모리 공간 낭비와 성능 문제 발생
- ToUpper2() 함수: strings.Builder 객체를 이용해 문자를 더한다. strings.Builder는 내부에 슬라이스를 가지고 있기 때문에 WriteRune() 메서드를 통해 문자를 더할 때 메번 메모리를 새로 생성하지 않고 기존 메모리공간에 빈자리가 있으면 그냥 더하게 된다. → 메모리 공간 낭비 X

15.5.2 왜 문자열은 불변 원칙을 지키려 할까?

이유: 예기치 못한 버그를 방지하기 위해서

만약 문자열 불변 원칙이 없다면 문자열이 언제라도 변화할 수 있게 되어 string 타입 변수를 안심하고 사용할 수 없는 경우가 발생한다.

```
func ChangeString(str3 string){
  str3[4] = 'T'
}

func main(){
  str := "Hello World"
  str2 := str

  ChangeString(str)
}
```

string 타입이 복사될 때 문자열 전체가 복사되는 것이 아닌 Data, Len 필드값만 복사된다. str, str2, str3 모두 같은 문자열을 가리키게 된다. 문자열 일부 값을 변경하면 str, str2, str3 모두 변경된 문자열을 가리키게 된다. 만약 string 변숫값이 코드 전반에 걸쳐서 여러 곳으로 복사됐다면 언제 어디에서 문자열이 변경되는지 알 수 없어서 많은 버그를 양산할 수 있다.

연습문제

1. 학교종이 땡땡땡
2. range

```
package main

import (
  "fmt"
  "strings"
)

func ToUpper(str string) string {
  var builder strings.Builder
  for _, v := range str {
    if v >= 'a' && v <= 'z' {
      builder.WriteRune('A' + (v - 'a'))
    } else {
      builder.WriteRune(v)
    }
  }
  return builder.String()
```

```
}

func main() {
  str := "hello World!"

  fmt.Println(ToUpper(str))
}
```

3. 16 * 2 + 8 = 40 바이트