Go 언어

언어 소개

GO 프로그래밍 언어는 2007년 구글(Google)에서 개발을 시작하여 2012년 GO 버젼 1.0을 완성하였다. 현재는 버젼 1.21이다. Go는 기존의 프로그래밍 언어의 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 가능하다. 마지막으로 Go의 큰 특징으로 Go는 Communicating Sequential Processes(CSP) 스타일의 Concurrent 프로그래밍을 지원한다.

Go 설치

Go 프로그래밍을 시작하기 위해 Go 공식 웹사이트인 해당 OS 버젼의 Go를 다운로드하여 설치한다. 윈도우즈에 Go를 설치하기 위해서는 MSI파일을 다운받아 실행하면 되는데, Go는 기본적으로 `C:\Program Files\Go` 폴더에 설치되며 (아주 오래된 버전에서는 `C:\go`에 설치되기도 했습니다), MSI 설치 프로그램이 `C:\Program Files\Go\bin`을 시스템 PATH 환경변수에 추가합니다. Go를 설치하고 해당 설치 디렉토리 밑에 bin 디렉토리를 보면 `go.exe` 파일이 있는데, 이 컴파일러로 go 프로그램을 컴파일하거나 실행할 수 있다. Go 프로그램은 파일 확장자 `.go` 를 갖는다. Go가 설치되었으면, 아래와 같은 명령으로 Go 버전을 간단하게 체크해 볼 수 있다.

```
C:\> go version
go version go1.21.3 windows/amd64
```

GO 환경변수 (go env)

Go는 여러 개의 자체 환경 변수들을 가지고 있는데, Go 환경변수는 go env 명령을 실행하여 확인할 수 있다. Go는 2개의 중요한 환경변수(GOROOT와 GOPATH)를 사용하는데, 이 환경변수들은 GO 설치시 자동으로 설정된다.

- GOROOT:
  - Go가 설치된 디렉토리(윈도우즈의 경우 디폴트로 `C:\Program Files\Go`)를 가리키며, Go 실행파일은 GOROOT/bin 폴더에 있다. 또한, Go의 표준 패키지 (예: fmt) 들은 GOROOT/src 폴더에 설치된다.
- GOPATH:
  - Go 프로젝트의 홈 디렉토리 역할을 하는 곳으로, 윈도우즈의 경우 디폴트로 `%USERPROFILE%\go` 폴더이며, 리눅스의 경우 `$HOME/go` 폴더이다. Go 1.11에서 모듈(Modules)이 도입되기 전에는 GOPATH가 소스 코드와 의존성을 관리하는 중요한 역할을 했습니다. 하지만 현재는 Go 모듈을 사용하여 프로젝트별로 의존성을 관리하는 것이 표준이며, GOPATH의 중요성은 크게 줄었습니다. `go get`으로 설치된 서드파티 실행 파일들이 여전히 `$GOPATH/bin`에 저장됩니다.

변수와 상수

변수

변수는 Go 키워드 var 를 사용하여 선언한다. var 키워드 뒤에 변수명을 적고, 그 뒤에 변수타입을 적는다. 예를 들어, 아래는 a 라는 정수(int) 변수를 선언한 것이다.

```
var a int
```

변수 선언문에서 변수 초기값을 할당할 수도 있다. 즉, float32 타입의 변수 f 에 11.0 이라는 초기값을 할당하기 위해서는 아래와 같이 쓸 수 있다.

```
var f float32 = 11.0
```

일단 선언된 변수는 그 뒤의 문장에서 해당 타입의 값을 할당할 수 있다. 만약 선언된 변수가 Go 프로그램 내에서 사용되지 않는다면, 에러를 발생시킨다. 따라서 사용되지 않는 변수는 프로그램에서 삭제한다. 동일한 타입의 변수가 여러개 있을 경우, 변수들을 나열하고 마지막에 타입을 한번만 지정할 수 있다.

```
var i, j, k int
```

복수 변수들이 선언된 상황에서 초기값을 지정할 수 있다. 초기값은 순서대로 변수에 할당된다. 예를 들어, 아래 코드의 경우 i는 1, j는 2, k는 3 이 할당된다.

```
var i, j, k int = 1, 2, 3
```

Go 에서는 할당되는 값을 보고 그 타입을 추론하는 기능이 자주 사용된다. 즉, 아래 코드에서 i는 정수형으로 1이 할당되고, s는 문자열로 Hi가 할당된다.

```
var i = 1
var s = "Hi"
```

변수를 선언하는 또 다른 방식으로 Short Assignment Statement(:=) 를 사용할 수 있다. 즉, `var i = 1` 을 쓰는 대신 `i := 1` 이라고 var 를 생략하고 사용할 수 있다. 하지만 이러한 표현은 함수(func) 내에서만 사용할 수 있으며, 함수 밖에서는 var를 사용해야 한다. Go에서 변수와 상수는 함수 밖에서도 사용할 수 있다.

상수

상수는 Go 키워드 const 를 사용하여 선언한다. const 키워드 뒤에 상수명을 적고, 그 뒤에 상수타입, 그리고 상수값을 할당한다.

```
const c int = 10
const s string = "Hi"
```

Go 에서는 할당되는 값을 보고 그 타입을 추론하는 기능이 자주 사용된다. 즉, 위의 경우 타입 int, string 을 생략하면 Go에서 자동으로 그 타입을 추론하게 된다.

```
const c = 10
const s = "Hi"
```

여러 개의 상수들 묶어서 지정할 수 있는데, 아래와 같이 괄호 안에 상수들을 나열하여 사용할 수 있다.

```
const (
    Visa = "Visa"
    Master = "MasterCard"
    Amex = "American Express"
)
```

한가지 유용한 팁으로 상수값을 0부터 순차적으로 부여하기 위해 iota 라는 identifier를 사용할 수 있다. 이 경우 iota가 지정된 Apple에는 0이 할당되고, 나머지 상수들을 순서대로 1씩 증가된 값을 부여받는다.

```
const (
    Apple = iota // 0
    Grape        // 1
    Orange       // 2
)
```

Go 키워드

다음 Go 에서 사용하는 예약 키워드들로 변수명, 상수명 등의 Identifier에 사용할 수 없다.

```
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
```

데이터 타입

Go 프로그래밍 언어는 다음과 같은 기본적인 데이타 타입들을 갖고 있다.

1. 부울린 타입(bool):
2. 문자열 타입(string):
   - string은 한번 생성되면 수정될 수 없는 Immutable 타입임
3. 정수형 타입(int): int8 int16 int32 int64 uint uint8 uint16 uint32 uint64 uintptr
4. Float 및 복소수 타입: float32 float64 complex64 complex128
5. 기타 타입:
   - byte: uint8과 동일하며 바이트 코드에 사용.
   - rune: int32과 동일하며 유니코드 코드포인트에 사용한다.

기본 자료형

- bool
- string
- int int8 int16 int32 int64
- uint uint8 uint16 uint32 uint64 uintptr
- byte
- rune
- float32 float64
- complex64 complex128

int 와 uintptr type은 보통 32-bit 시스템에서는 32 bit, 64-bit 시스템에서는 64 bit의 길이입니다. 정수 값이 필요할 때에는 특정한 이유로 크기를 정해야하거나 unsigned 정수 type을 사용해야하는 게 아니라면 int를 사용해야합니다.

Zero values

명시적인 초깃값 없이 선언된 변수는 그것의 zero value 가 주어집니다.

zero value 는 다음과 같습니다.

- 숫자 type에는 0
- boolean type에는 false
- string에는 “” (빈 문자열)

데이타 타입 변환 (Type Conversion)

하나의 데이타 타입에서 다른 데이타 타입으로 변환하기 위해서는 T(v) 와 같이 표현하고 이를 Type Conversion이라 부르는데, 여기서 T는 변환하고자 하는 타입을 표시하고, v는 변환될 값(value)을 지정한 것이다. 예를 들어, 정수 100을 float로 변경하기 위하여 float32(100) 처럼 표현하고, 문자열을 바이트배열로 변경하기 위하여 []byte(“ABC”) 처럼 표현할 수 있다. 참고로 Go에서 배열 표시 [] 는 데이타 타입 byte 앞에 표시한다.

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)
```

```
// 혹은 좀 더 간단히
i := 42
f := float64(i)
u := uint(f)
```

아래 예제는 여러 Type Conversion의 예를 보인 것으로, int와 uint/float간 변환과 문자열과 바이트배열간의 변환을 예시하고 있다. 한가지 주의할 점은 Go에서 타입간 변환은 명시적으로 지정해 주어야 한다는 점인데, 예를 들어 정수형 int에서 uint로 변환할 때, 암묵적(implicit) 변환이 일어나지 않으므로 uint(i) 처럼 반드시 변환을 지정해 주어야 한다. 만약 명시적 지정이 없이 변환이 일어나면 런타임 에러가 발생한다.

```
func main() {
    var i int = 100
    var u uint = uint(i)
    var f float32 = float32(i)
    println(f, u)
    str := "ABC"
    bytes := []byte(str)
    str2 := string(bytes)
    println(bytes, str2)
}
```

Go 연산자

Go 언어는 다른 언어에서와 비슷하게 산술연산자, 관계연산자, 논리연산자, Bitwise 연산자, 할당연산자, 포인터연산자 등을 지원한다.

산술연산자

산술연산자는 사칙연산자(+, -, *, /, %)와 증감연산자(++, --)를 사용한다.

관계연산자

관계연산자는 서로의 크기를 비교하거나 동일함을 체크하는데 사용된다.

```
a == b
a != c
a >= b
```

논리연산자

논리연산자는 AND, OR, NOT을 표현하는데 사용된다.

```
A && B
A || !(C && B)
```

Bitwise 연산자

Bitwise 연산자는 비트단위 연산을 위해 사용되는데, 바이너리 AND, OR, XOR와 바이너리 쉬프트 연산자가 있다.

```
c = (a & b) << 5
```

할당연산자

할당연산자는 값을 할당하는 = 연산자 외에 사칙연산, 비트연산을 축약한 +=, &=, <<= 같은 연산자들도 있다.

```
a = 100
a *= 10
a >>= 2
a |= 1
```

포인터연산자

포인터연산자는 C++와 같이 & 혹은 *을 사용하여 해당 변수의 주소를 얻어내거나 이를 반대로 Dereference 할 때 사용한다. Go 는 비록 포인터연산자를 제공하지만 포인터 산술 즉 포인터에 더하고 빼는 기능은 제공하지 않는다.

```
var k int = 10
var p = &k  //k의 주소를 할당
println(*p) //p가 가리키는 주소에 있는 실제 내용을 출력
```

조건문

if문

if 문은 해당 조건이 맞으면 { } 블럭안의 내용을 실행한다. Go의 if 조건문은 아래 예제에서 보듯이 조건식을 괄호( )로 둘러 싸지 않아도 된다. 그리고 반드시 조건 블럭 시작 브레이스({)를 if문과 같은 라인에 두어야 한다. 이를 다음 라인에 두게 되면 에러를 발생시킨다.

그리고 한가지 주목할 점은 if 문의 조건식은 반드시 Boolean 식으로 표현되어야 한다는 것이다. 이점은 C/C++ 같은 다른 언어들이 조건식에 1, 0 과 같은 숫자를 쓸 수 있는 것과 대조적이다.

```
if k == 1 {  //같은 라인
    println("One")
}
```

if 문은 else if, 혹은 else 문을 함께 가질 수 있다. else if 문은 if 조건문이 거짓일 때 다시 다른 if 조건식을 검사하는 데 사용되며, else 문은 이전의 if 문들이 모두 거짓일 때 실행된다. if 문과 마찬가지로 else if 혹은 else 문의 블럭 시작 브레이스는 같은 라인에 있어야 한다.

```
if k == 1 {
    println("One")
} else if k == 2 {  //같은 라인
    println("Two")
} else {   //같은 라인
    println("Other")
}
```

if 문에서 조건식을 사용하기 이전에 간단한 문장(Optional Statement)을 함께 실행할 수 있다. 즉, 아래 예제처럼 val := i*2 라는 문장을 조건식 이전에 실행할 수 있는데, 여기서 주의할 점은 이때 정의된 변수 val는 if문 블럭 (혹은 if-else 블럭 scope) 안에서만 사용할 수 있다는 것이다. 이러한 Optional Statement 표현은 아래의 switch문, for문 등 Go의 여러 문법에서 사용할 수 있다.

```
if val := i * 2; val < max {
    println(val)
}
// 아래 처럼 사용하면 Scope 에러
val++
```

Switch 문

여러 값을 비교해야 하는 경우 혹은 다수의 조건식을 체크해야 하는 경우 switch 문을 사용한다. 다른 언어들과 비슷하게 switch 문 뒤에 하나의 변수(혹은 Expression)를 지정하고, case 문에 해당 변수가 가질 수 있는 값들을 지정하여, 각 경우에 다른 문장 블럭들을 실행할 수 있다. 복수개의 case 값들이 있을 경우는 아래 예제에서 보듯이 case 3,4 처럼 콤마를 써서 나열할 수 있다.

```
package main

func main() {
    var name string
    var category = 1
    switch category {
    case 1:
        name = "Paper Book"
    case 2:
        name = "eBook"
    case 3, 4:
        name = "Blog"
    default:
        name = "Other"
    }
    println(name)
    // Expression을 사용한 경우
    switch x := category << 2; x - 1 {
        // ...
    }
}
```

Go의 단순한 switch문 용법은 C++, C#, Java 등의 다른 언어와 비슷하지만, 다음 목록은 Go만의 특별한 용법들이다.

switch 뒤에 expression이 없을 수 있음

- 다른 언어는 switch 키워드 뒤에 변수나 expression 반드시 두지만, Go는 이를 쓰지 않아도 된다. 이 경우 Go는 switch expression을 true로 생각하고 첫번째 case문으로 이동하여 검사한다

case문에 expression을 쓸 수 있음

- 다른 언어의 case문은 일반적으로 리터럴 값만을 갖지만, Go는 case문에 복잡한 expression을 가질 수 있다

No default fall through

- 다른 언어의 case문은 break를 쓰지 않는 한 다음 case로 이동하지만, Go는 다음 case로 가지 않는다

Type switch

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

Go의 switch 문에서 한가지 특징적인 용법은 switch 뒤에 조건변수 혹은 Expression을 적지 않는 용법이다. 이 경우 각 case 조건문들을 순서대로 검사하여 조건에 맞는 경우 해당 case 블럭을 실행하고 switch문을 빠져나온다. 이 용법은 복잡한 if...else if...else if... 문장을 단순화하는데 유용하다.

```
func grade(score int) {
    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")
    }
}
```

Go의 또 다른 용법은 switch 변수의 타입을 검사하는 타입 switch가 있다. 아래 예제는 변수 v의 타입이 int

인지, bool, string 인지를 체크한 후 해당 case 블럭을 실행하는 예이다.

```
switch v.(type) {
case int:
    println("int")
case bool:
    println("bool")
case string:
    println("string")
default:
    println("unknown")
}
```

C 혹은 C# 과 같은 언어에서 case 문은 case 블럭 마지막에 break 문을 명시하여 switch 문을 빠져나온다. 만약 break 문이 없으면, case 문 밑의 모든 문장들을 실행해 버린다. Go는 case문 마지막에 break 문을 적든 break 문을 생략하든, 항상 break 하여 switch 문을 빠져나온다. 이는 Go 컴파일러가 자동으로 break 문을 각 case문 블럭 마지막에 추가하기 때문이다. Go에서 만약 이러한 디폴트 break 문을 사용하지 않고, C나 C#처럼 계속 밑의 문장들(다음 case문 코드 블럭들)을 실행하게 하려면, fallthrough 문을 명시해 주면 된다. fallthrough 문을 사용한 아래 예제를 실행하면, "2 이하/3 이하/default 도달"을 모두 출력하게 된다. 즉, 일단 case 2 에 도착한 후 fallthrough 가 있으므로, (val 값이 3이 아님에도) case 3의 코드 블럭을 계속 실행하고, case 3에도 fallthrough 가 있으므로 default 블럭을 계속 실행한다.

```
package main

import "fmt"

func main() {
    check(2)
}
func check(val int) {
    switch val {
    case 1:
        fmt.Println("1 이하")
        fallthrough
    case 2:
        fmt.Println("2 이하")
        fallthrough
    case 3:
        fmt.Println("3 이하")
        fallthrough
    default:
        fmt.Println("default 도달")
    }
}
```

for 문

Go 프로그래밍 언어에서 반복문은 for 루프를 이용한다. Go는 반복문에 for 하나 밖에 없다. for 루프는 다른 언어와 비슷하게 "for 초기값; 조건식; 증감 { … }" 의 형식을 따른다. 물론 초기값, 조건식, 증감식 등은 경우에 따라 생략할 수 있다. 다만, "초기값; 조건식; 증감"을 둘러싸는 괄호 ()를 생략하는데, 괄호를 쓰면 에러가 난다. 아래 예제는 1부터 100까지 더하는 for 루프문 예이다.

```
package main

func main() {
    sum := 0
    for i := 1; i <= 100; i++ {
        sum += i
    }
    println(sum)
}
```

for 문 - 조건식만 쓰는 for 루프

Go에서 for 루프는 초기값과 증감식을 생략하고 조건식만을 사용할 수 있는데, 이는 마치 for 루프가 다른 언어의 while 루프와 같이 쓰이도록 한다. 아래 예제에서 for 루프는 n이 100을 넘는지를 체크하는 조건식만을 사용하고 있다.

```
package main

func main() {
    n := 1
    for n < 100 {
        n *= 2
        //if n > 90 {
        //    break
        //}
    }
    println(n)
}
```

for 문 - 무한루프

for 루프로 무한루프를 만들려면 "초기값; 조건식; 증감" 모두를 생략하면 된다. 아래 예제는 무한루프를 만든 예이다. 무한루프를 빠져나오기 위해 Ctrl+C 를 누른다.

```
package main

func main() {
    for {
        println("Infinite loop")
    }
}
```

for range 문

for range 문은 컬렉션으로 부터 한 요소(element)씩 가져와 차례로 for 블럭의 문장들을 실행한다. 이는 다른 언어의 foreach와 비슷한 용법이다.

for range 문은 "for 인덱스,요소값 := range 컬렉션" 같이 for 루프를 구성하는데, range 키워드 다음의 컬렉션으로부터 하나씩 요소를 리턴해서 그 요소의 위치 인덱스와 값을 for 키워드 다음의 2개의 변수에 각각 할당한다.

아래 예제는 3명의 이름을 갖는 문자열 배열에서 문자열 인덱스(0, 1, 2)와 해당 이름을 차례로 가져와 프린트하는 코드이다.

```
names := []string{"홍길동", "이순신", "강감찬"}

for index, name := range names {
    println(index, name)
}
```

break, continue, goto 문

경우에 따라 for 루프내에서 즉시 빠져나올 필요가 있는데, 이때 break 문을 사용한다. 만약 for 루프의 중간에서 나머지 문장들을 실행하지 않고 for 루프 시작부분으로 바로 가려면 continue문을 사용한다. 그리고 기타 임의의 문장으로 이동하기 위해 goto 문을 사용할 수 있다. goto문은 for 루프와 관련없이 사용될 수 있다.

break문은 for 루프 이외에 switch문이나 select문에서도 사용할 수 있다. 하지만, continue문은 for 루프와 연관되어 사용된다.

```
package main
func main() {
    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")
}
```

break문은 보통 단독으로 사용되지만, 경우에 따라 "break 레이블"과 같이 사용하여 지정된 레이블로 이동할 수도 있다. break의 "레이블"은 보통 현재의 for 루프를 바로 위에 적게 되는데, 이러한 "break 레이블"은 현재의 루프를 빠져나와 지정된 레이블로 이동하고, break문의 직속 for 루프 전체의 다음 문장을 실행하게 한다. 아래 예제는 언뜻 보기에 무한루프를 돌 것 같지만, 실제로는 OK를 출력하고 프로그램을 정상 종료한다. 이는 "break L1" 문이 for 루프를 빠져나와 L1 레이블로 이동한 후, break가 있는 현재 for 루프를 건너뛰고 다음 문장인 println() 으로 이동하기 때문이다.

```
package main

func main() {
    i := 0
L1:
    for {

        if i == 0 {
            break L1
        }
    }
    println("OK")
}
```

함수

함수는 여러 문장을 묶어서 실행하는 코드 블럭의 단위이다. Go에서 함수는 func 키워드를 사용하여 정의한다. func 뒤에 함수명을 적고 괄호 ( ) 안에 그 함수에 전달하는 파라미터들을 적게 된다. 함수 파라미터는 0개 이상 사용할 수 있는데, 각 파라미터는 파라미터명 뒤에 int, string 등의 파라미터 타입을 적어서 정의한다. 함수의 리턴 타입은 파라미터 괄호 ( ) 뒤에 적게 되는데, 이는 C와 같은 다른 언어에서 리턴 타입을 함수명 앞에 쓰는 것과 대조적이다. 함수는 패키지 안에 정의되며

호출되는 함수가 호출하는 함수의 반드시 앞에 위치해야 할 필요는 없다.

아래 예제는 say라는 함수를 정의한 예이다. say() 함수는 문자열 msg 파라미터를 하나 갖고 있으며, 리턴 값이 없으므로 별도의 리턴타입을 정의하지 않았다.

```
package main
func main() {
    msg := "Hello"
    say(msg)
}
func say(msg string) {
    println(msg)
}
```

Pass By Reference

Go에서 파라미터를 전달하는 방식은 크게 Pass By Value와 Pass By Reference로 나뉜다.

- Pass By Value: 위의 [1. 함수]의 예제에서는 msg의 값 "Hello" 문자열이 복사되어 함수 say()에 전달된다. 즉, 만약 파라미터 msg의 값이 say() 함수 내에서 변경된다하더라도 호출함수 main()에서의 msg 변수는 변함이 없다.
- Pass By Reference: 아래의 예제에서처럼 msg 변수 앞에 & 부호를 붙이면 msg 변수의 주소를 표시하게 된다. 흔히 포인터라 불리우는 이 용법을 사용하면 함수에 msg 변수의 값을 복사하지 않고 msg 변수의 주소를 전달하게 된다. 피호출 함수 say()에서는 *string 과 같이 파라미터가 포인터임을 표시하고 이때 say 함수의 msg는 문자열이 아니라 문자열을 갖는 메모리 영역의 주소를 갖게 된다. msg 주소에 데이타를 쓰기 위해서는 *msg = "" 과 같이 앞에 *를 붙이는데 이를 흔히 Dereferencing 이라 한다.

아래 예제의 경우 main 함수의 msg 변수의 값이 say 함수에서 Changed 로 변경되었으므로 main 함수의 println() 에서 변경된 값이 출력된다.

```
package main
func main() {
    msg := "Hello"
    say(&msg)
    println(msg) //변경된 메시지 출력
}

func say(msg *string) {
    println(*msg)
    *msg = "Changed" //메시지 변경
}
```

Variadic Function (가변인자함수)

함수에 고정된 수의 파라미터들을 전달하지 않고 다양한 숫자의 파라미터를 전달하고자 할 때 가변 파라미터를 나타내는 ... (3개의 마침표)을 사용한다. 즉 문자열 가변 파라미터를 나타내기 위해서 ...string 과 같이 표현한다. 가변 파라미터를 갖는 함수를 호출할 때는 0개, 1개, 2개, 혹은 n개의 동일타입 파라미터를 전달할 수 있다. 이렇게 가변 파라미터를 받아들이는 함수를 Variadic Function(가변인자함수)라고 부른다. 아래 예제는 say 함수에 4개의 문자열을 전달할 수도 있고, 1개의 문자열을 전달할 수도 있음을 예시하고 있다.

```
package main
func main() {
    say("This", "is", "a", "book")
    say("Hi")
}
func say(msg ...string) {
    for _, s := range msg {
        println(s)
    }
}
```

함수 리턴값

Go 프로그래밍 언어에서 함수는 리턴값이 없을 수도, 리턴값이 하나 일 수도, 또는 리턴값이 복수 개일 수도 있다. C 언어에서 void 혹은 하나의 값만을 리턴하는 것과 대조적으로 Go 언어는 복수개의 값을 리턴할 수 있다. Go 언어는 또한 Named Return Parameter 라는 기능을 제공하는데, 이는 리턴되는 값들을 (함수에 정의된) 리턴 파라미터들에 할당할 수 있는 기능이다. 함수에서 리턴값이 있는 경우는 func 문의 (파라미터 괄호 다음) 마지막에 리턴값의 타입을 정의해 준다. 그리고 값을 리턴하기 위해 함수내에서 return 키워드를 사용한다. 예를 들어, 아래 예제는 sum() 함수의 리턴 타입이 int 임을 표시하고 있으며, sum 함수 마지막에 return s 와 같이 정수 s의 값을 리턴하고 있다.

```
package main

func main() {
    total := sum(1, 7, 3, 5, 9)
    println(total)
}
func sum(nums ...int) int {
    s := 0
    for _, n := range nums {
        s += n
    }
    return s
}
```

Go에서 복수 개의 값을 리턴하기 위해서는 해당 리턴 타입들을 괄호 ( ) 안에 적어 준다. 예를 들어, 처음 리턴값이 int이고 두번째 리턴값이 string 인 경우 (int, string) 과 같이 적어 준다. 아래 예제는 sum() 함수에 가변인수로 숫자들이 전달될 때, 그 숫자들의 갯수와 합계를 함께 리턴하는 코드이다.

```
package main

func main() {
    count, total := sum(1, 7, 3, 5, 9)
    println(count, total)
}
func sum(nums ...int) (int, int) {
    s := 0      // 합계
    count := 0  // 요소 갯수
    for _, n := range nums {
        s += n
        count++
    }
    return count, s
}
```

Go에서 Named Return Parameter들에 리턴값들을 할당하여 리턴할 수 있는데, 이는 리턴되는 값들이 여러 개일 때, 코드 가독성을 높이는 장점이 있다. 예를 들어, 위의 sum() 함수를 Named Return Parameter를 이용하여 고쳐쓰면 다음과 같다. 아래 예제에서 func 라인의 마지막 리턴타입 정의 부분을 보면, (int, int) 가 아니라 (count int, total int) 처럼 정의되어 있음을 볼 수 있다. 즉, 리턴 파라미터명과 그 타입을 함께 정의한 것이다. 그리고 함수 내에서는 이 count, total에 결과값을 직접 할당하고 있음을 볼 수 있다. 또한 마지막에 return 문이 있는 것을 볼 수 있는데, 실제 return 문에는 아무 값들을 리턴하지 않지만, 그래도 리턴되는 값이 있을 경우에는 빈 return 문을 반드시 써 주어야 한다 (이를 생략하면 에러 발생).

```
func sum(nums ...int) (count int, total int) {
    for _, n := range nums {
        total += n
    }
    count = len(nums)
    return
}
```

익명함수

함수명을 갖지 않는 함수를 익명함수(Anonymous Function)이라 부른다. 일반적으로 익명함수는 그 함수 전체를 변수에 할당하거나 다른 함수의 파라미터에 직접 정의되어 사용되곤 한다. 아래 예제는 main() 함수 안에서 익명함수가 선언되어 sum이라는 변수에 할당되고 있음을 보여준다. sum에 할당된 익명함수를 보면 func 바로 뒤에 함수명이 생략되었음을 알 수 있다. 이 점을 제외하고 함수의 정의 내용을 동일한다. 일단 익명함수가 변수에 할당된 이후에는 변수명이 함수명과 같이 취급되며 "변수명(파라미터들)" 형식으로 함수를 호출할 수 있다.

```
package main

func main() {
    sum := func(n ...int) int { //익명함수 정의
        s := 0
        for _, i := range n {
            s += i
        }
        return s
    }
    result := sum(1, 2, 3, 4, 5) //익명함수 호출
    println(result)
}
```

클로저 (Closure)

Go 언어에서 함수는 Closure로서 사용될 수도 있다. Closure는 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.

아래 예제에서 nextValue() 함수는 int를 리턴하는 익명함수(func() int)를 리턴하는 함수이다. Go 언어에서 함수는 일급함수로서 다른 함수로부터 리턴되는 리턴값으로 사용될 수 있다. 그런데 여기서 이 익명함수가 그 함수 바깥에 있는 변수 i 를 참조하고 있다. 익명함수 자체가 로컬 변수로 i 를 갖는 것이 아니기 때문에 (만약 그렇게 되면 함수 호출시 i는 항상 0으로 설정된다) 외부 변수 i 가 상태를 계속 유지하는 즉 값을 계속 하나씩 증가시키는 기능을 하게 된다. 예제에서 next := nextValue() 에서 Closure 함수를 next라는 변수에 할당한 후에, 계속 next()를 3번 호출하

는데 이때마다 Clouse 함수내의 변수 i는 계속 증가된 값을 가지고 있게 된다. 이것은 마치 next 라는 함수값이 변수 i 를 내부에 유지하고 있는 모양새이다. 그러나 만약 `anotherNext := nextValue()`와 같이 새로운 Closure 함수값을 생성한다면, 변수 i는 초기 0을 갖게 되므로 다시 1부터 카운팅을 하게 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
}
```

컬렉션

배열(Array)

배열은 연속적인 메모리 공간에 동일한 타입의 데이타를 순서적으로 저장하는 자료구조이다. Go에서 배열의 첫번째 요소는 0번, 그 다음은 1번, 2번 등으로 인덱스를 매긴다.

배열의 선언은 `var 변수명 [배열크기] 데이타타입` 과 같이 하는데, 배열크기를 데이타타입 앞에 써 주는 것이 C, Java 같은 다른 언어들과 다르다. Go에서 배열의 배열크기는 Type을 구성하는 한 요소이다. 즉, [3]int와 [5]int는 서로 다른 타입으로 인식된다. 배열이 선언된 후에 각 배열의 요소를 인덱스를 사용하여 읽거나 쓸 수 있다.

```
package main

func main() {
    var a [3]int  //정수형 3개 요소를 갖는 배열 a 선언
    a[0] = 1
    a[1] = 2
    a[2] = 3
    println(a[1]) // 2 출력
}
```

배열의 초기화

배열을 정의할 때, 초기값을 설정할 수도 있다. 초기값은 "[배열크기] 데이타타입" 뒤에 { } 괄호를 두고 초기값을 순서대로 적으면 된다. 즉, 위 예제의 배열 초기화를 다음과 같이 할 수 있다. 만약 초기화 과정에서 [⋯]를 사용하여 배열크기를 생략하면 자동으로 초기화 요소 숫자만큼 배열크기가 정해진다.

```
var a1 = [3]int{1, 2, 3}
var a3 = [...]int{1, 2, 3} //배열크기 자동으로
```

다배열 배열

Go 언어는 다차원 배열을 지원한다. 다차원 배열은 배열크기 부분을 복수 개로 설정하여 선언한다. 예를 들어, `3 x 4 x 5` 차원 정수 배열을 만들려면, 다음과 같이 배열을 정의한다. 다차원 배열의 사용은 일차원 배열과 유사하게 각 차원별 배열인덱스를 지정하여 사용한다.

```
var multiArray [3][4][5]int  // 정의
multiArray[0][1][2] = 10     // 사용
```

다차원 배열의 초기화

다차원 배열의 초기화는 단차원 배열의 초기화와 비슷하다. 다만, 다차원이므로 배열 초기값 안에 다시 배열값을 넣는 형태를 띤다. 즉, 아래 예제에서 보듯이, 2개의 행과 3개의 열을 이루는 배열이 초기화되었는데, 1행이 `{1,2,3}` 처럼 묶여서 표현되고 있다.

```
func main() {
    var a = [2][3]int{
        {1, 2, 3},
        {4, 5, 6},  //끝에 콤마 추가
    }
    println(a[1][2])
}
```

슬라이스(Slice)

Go 배열은 고정된 배열크기 안에 동일한 타입의 데이타를 연속적으로 저장하지만, 배열의 크기를 동적으로 증가시키거나 부분 배열을 발췌하는 등의 기능을 가지고 있지 않다. Go Slice는 내부적으로 배열에 기초하여 만들어 졌지만 배열의 이런 제약점들을 넘어 개발자에게 편리하고 유용한 기능들을 제공한다. 슬라이스는 배열과 달리 고정된 크기를 미리 지정하지 않을 수 있고, 차후 그 크기를 동적으로 변경할 수도 있고, 또한 부분 배열을 발췌할 수도 있다.

Go Slice 선언은 배열을 선언하듯이 `"var v []T"` 처럼 하는데 배열과 달리 크기는 지정하지 않는다. 예를 들어, 정수형 Slice 변수 a를 선언하기 위해서 `"var a []int"` 처럼 선언할 수 있다.

```
package main
import "fmt"

func main() {
    var a []int         //슬라이스 변수 선언
    a = []int{1, 2, 3} //슬라이스에 리터럴 값 지정
    a[1] = 10
    fmt.Println(a)      // [1, 10, 3]출력
}
```

Go에서 Slice를 생성하는 또 다른 방법으로 Go의 내장함수 make() 함수를 이용할 수 있다. make() 함수로 슬라이스를 생성하면, 개발자가 슬라이스의 길이(Length)와 용량(Capacity)을 임의로 지정할 수 있는 장점이 있다. make() 함수의 첫번째 파라미터에 생성할 슬라이스 타입을 지정하고, 두번째는 Length (슬라이스의 길이), 그리고 세번째는 Capacity (내부 배열의 최대 길이)를 지정하면, 모든 요소가 Zero value인 슬라이스를 만들게 된다. 여기서 만약 세번째 Capacity 파라미터를 생략하면 Capacity는 Length와 같은 값을 갖는다. 그리고 슬라이스의 길이 및 용량은 내장함수 `len()`, `cap()`을 써서 확인할 수 있다.

```
func main() {
    s := make([]int, 5, 10)
    println(len(s), cap(s)) // 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.

```
func main() {
    var s []int
    if s == nil {
        println("Nil Slice")
    }
    println(len(s), cap(s)) // 모두 0
}
```

부분 슬라이스(Sub-slice)

슬라이스에서 일부를 발췌하여 부분 슬라이스를 만들 수 있다. 부분 슬라이스는 "슬라이스[처음인덱스:마지막인덱스]" 형식으로 만드는데, 예를 들어 슬라이스 s 에 대해 인덱스 2부터 4까지 데이타를 갖는 부분 슬라이스를 만드려면, s[2:5]와 같이 표현한다. 여기서 [2:4]가 아니라 [2:5]으로 쓰는데, 마지막인덱스는 원하는 인덱스+1 을 사용한다. 즉, 처음인덱스는 Inclusive 이며, 마지막인덱스는 Exclusive이다.

```
package main
import "fmt"

func main() {
    s := []int{0, 1, 2, 3, 4, 5}
    s = s[2:5]
    fmt.Println(s) //2,3,4 출력
}
```

슬라이스 인덱스는 처음/마지막 둘 중 하나 혹은 둘 다를 생략할 수도 있다. 처음 인덱스가 생략되면 0 이, 마지막 인덱스가 생략되면 그 슬라이스의 마지막 인덱스가 자동 대입된다. 즉, 처음 0 부터 인덱스 4까지를 포함하기 위해서는 [:5]를, 인덱스 2부터 마지막까지 포함하기 위해서는 [2:]와 같이 쓸 수 있다. 만약 [:]와 같이 모두 생략하면, 전체를 표현한다.

```
s := []int{0, 1, 2, 3, 4, 5}
s = s[2:5]      // 2, 3, 4
s = s[1:]       // 3, 4
fmt.Println(s) // 3, 4 출력
```

슬라이스 추가,병합(append)과 복사(copy)

배열은 고정된 크기로 그 크기 이상의 데이타를 임의로 추가할 수 없지만, 슬라이스는 자유롭게 새로운 요소를 추가할 수 있다. 슬라이스에 새로운 요소를 추가하기 위해서는 Go 내장함수인 append()를 사용한다. append()의 첫 파라미터는 슬라이스 객체이고, 두번째는 추가할 요소의 값이다. 또한 여러 개의 요소 값들을 한꺼번에 추가하기 위해서는 append() 두번째 파라미터 뒤에 계속하여 값을 추가할 수 있다.

```
func main() {
    s := []int{0, 1}
    // 하나 확장
    s = append(s, 2)        // 0, 1, 2
    // 복수 요소들 확장
    s = append(s, 3, 4, 5) // 0,1,2,3,4,5
    fmt.Println(s)
}
```

내장함수 append()가 슬라이스에 데이타를 추가할 때, 내부적으로 다음과 같은 일이 일어난다. 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 (주: 아래 내부구조 참조) 을 생성하고 기존 배열 값들을 모두 새 배열에 복제한 후 다시 슬라이스를 할당한다. 아래 예제는 길이 0/용량 3의 슬라이스에 1부터 15까지의 숫자를 계속 추가하면서 슬라이스의 길이와 용량이 어떻게 변하는 지를 체크하는 코드이다. 이 코드를 실행하면 1~3까지는 기존의 용량 3을 사용하고, 4~6까지는 용량 6을, 7~12는 용량 12, 그리고 13 15는 용량 24의 슬라이스가 사용되고 있음을 알 수 있다.

```
package main

import "fmt"

func main() {
    // len=0, cap=3 인 슬라이스
    sliceA := make([]int, 0, 3)
    // 계속 한 요소씩 추가
    for i := 1; i <= 15; i++ {
        sliceA = append(sliceA, i)
        // 슬라이스 길이와 용량 확인
        fmt.Println(len(sliceA),
cap(sliceA))
    }
    fmt.Println(sliceA) // 1 부터 15 까지
숫자 출력
}
```

한 슬라이스를 다른 슬라이스 뒤에 병합하기 위해서는 아래 예제와 같이 append()를 사용한다. 이 append 함수에서는 2개의 슬라이스를 파라미터로 갖는데, 처음 슬라이스 뒤에 두번째 파라미터의 슬라이스를 추가하게 된다. 여기서 한가지 주의할 것은 두번째 슬라이스 뒤에 ... 을 붙인다는 것인데, 이 ellipsis(...)는 해당 슬라이스의 컬렉션을 표현하는 것으로 두번째 슬라이스의 모든 요소들의 집합을 나타낸다. 즉, 아래 예제에서 sliceB... 는 4, 5, 6으로 치환된다고 볼 수 있다.

```
package main

import "fmt"

func main() {
    sliceA := []int{1, 2, 3}
    sliceB := []int{4, 5, 6}
    sliceA = append(sliceA, sliceB...)
    //sliceA = append(sliceA, 4, 5, 6)
    fmt.Println(sliceA) // [1 2 3 4 5 6]
출력
}
```

이러한 추가/확장 기능과 더불어, Go 슬라이스는 내장함수 copy()를 사용하여 한 슬라이스를 다른 슬라이스로 복사할 수도 있다. 아래 예제는 3개의 요소를 갖는 소스 슬라이스를 그 2배의 크기 즉 6개를 갖는 타겟슬라이스로 복사하는 예를 보여준다.

```
func main() {
    source := []int{0, 1, 2}
    target := make([]int, len(source),
cap(source)*2)
    copy(target, source)
    fmt.Println(target)  // [0 1 2 ] 출
력
    println(len(target), cap(target)) //
3, 6 출력
}
```

슬라이스의 내부구조

슬라이스는 내부적으로 사용하는 배열의 부분 영역인 세그먼트에 대한 메타 정보를 가지고 있다. 슬라이스는 크게 3개의 필드로 구성되어 있는데, 첫째 필드는 내부적으로 사용하는 배열에 대한 포인터 정보이고, 두번째는 세그먼트의 길이를, 그리고 마지막으로 세번째는 세그먼트의 최대 용량(Capacity)이다.

처음 슬라이스가 생성될 때, 만약 길이와 용량이 지정되었다면, 내부적으로 용량(Capacity)만큼의 배열을 생성하고, 슬라이스 첫번째 필드에 그 배열의 처음 메모리 위치를 지정한다. 그리고, 두번째 길이 필드는 지정된 (첫 배열요소로부터의) 길이를 갖게되고, 세번째 용량 필드는 전체 배열의 크기를 갖는다.

맵(Map)

Map은 키(Key)에 대응하는 값(Value)을 신속히 찾는 해시테이블(Hash table)을 구현한 자료구조이다. Go 언어는 Map 타입을 내장하고 있는데, "map[Key타입]Value타입" 과 같이 선언할 수 있다. 예를 들어 정수를 키로하고 문자열을 값으로 하는 맵 변수 idMap을 선언하기 위해서는 다음과 같이 할 수 있다.

```
var idMap map[int]string
```

이때 선언된 변수 idMap은 (map은 reference 타입이므로) nil 값을 갖으며, 이를 Nil Map이라 부른다. Nil map에는 어떤 데이타를 쓸 수 없는데, map을 초기화하기 위해 make()함수를 사용할 수 있다.

```
idMap = make(map[int]string)
```

make() 함수의 첫번째 파라미터로 map 키워드와 [키타입]값타입 을 지정하는데, 이때의 make()함수는 해시테이블 자료구조를 메모리에 생성하고 그 메모리를 가리키는 map value를 리턴한다 (map value는 내부적으로 runtime.hmap 구조체를 가리키는 포인터이다). 따라서 idMap 변수는 이 해시테이블을 가리키는 map을 가리키게 된다.

map은 make() 함수를 써서 초기화할 수도 있지만, 리터럴(literal)을 사용해 초기화할 수도 있다. 리터럴 초기화는 "map[Key타입]Value타입 { key:value }" 와 같이 Map 타입 뒤 { } 괄호 안에 "키: 값" 들을 열거하면 된다.

```
//리터럴을 사용한 초기화
tickers := map[string]string{
    "GOOG": "Google Inc",
    "MSFT": "Microsoft",
    "FB":   "FaceBook",
}
```

Map 사용

처음 map이 make() 함수에 의해 초기화 되었을 때는, 아무 데이타가 없는 상태이다. 이때 새로운 데이타를 추가하기 위해서는 "map변수[키] = 값" 과 같이 해당 키에 그 값을 할당하면 된다. 예를 들면, 아래 예제에서 키 901 에 Apple을 할당하면 새 해시 키-값 쌍이 추가된다. 만약 키 901의 값이 이미 존재했다면, 추가대신 값만 갱신한다.

```
package main

func main() {
    var m map[int]string
    m = make(map[int]string)
    //추가 혹은 갱신
    m[901] = "Apple"
    m[134] = "Grape"
    m[777] = "Tomato"
    // 키에 대한 값 읽기
    str := m[134]
    println(str)
    noData := m[999] // 값이 없으면 nil 혹
은 zero 리턴
    println(noData)
    // 삭제
    delete(m, 777)
}
```

map에서 특정 키에 대해 값을 읽을 때는 "map변수[키]"를 읽으면 된다. 즉, 위 예제에서 m[134]는 Grape라는 값을 str 변수에 할당하게 된다. 만약 map안에 찾는 키가 존재하지 않는다면 reference 타입인 경우 nil을 value 타입인 경우 zero를 리턴한다.

map에서 특정 키와 그 값을 삭제하기 위해서는 delete()함수를 사용한다.

Map 키 체크

map을 사용하는 경우 종종 map안에 특정 키가 존재하는지를 체크할 필요가 있다. 이를 위해 Go에선 "map변수[키]" 읽기를 수행할 때 2개의 리턴값을 리턴한다. 첫번째는 키에 상응하는 값이고, 두번째는 그 키가 존재하는지 아닌지를 나타내는 bool 값이다. 즉, 아래 예제에서 val, exists := tickers["MSFT"] 의 val 에는 Microsoft라는 값이 리턴되고, 변수 exists에는 키가 존재하므로 true가 리턴된다.

```
package main

func main() {
    tickers := map[string]string{
        "GOOG": "Google Inc",
        "MSFT": "Microsoft",
        "FB":   "FaceBook",
        "AMZN": "Amazon",
    }
    // map 키 체크
    val, exists := tickers["MSFT"]
    if !exists {
        println("No MSFT ticker")
    }
}
```

for 루프를 사용한 Map 열거

Map이 갖고 있는 모든 요소들을 출력하기 위해, for range 루프를 사용할 수 있다. Map 컬렉션에 for range를 사용하면, Map 키와 Map 값 2개의 데이타를 리턴한다. 아래 예제는 for 루프를 사용하여 맵으로부터 Key와 Value를 하나씩 가져와서 출력하는 코드이다.

```
package main

import "fmt"

func main() {
    myMap := map[string]string{
```

```
        "A": "Apple",
        "B": "Banana",
        "C": "Charlie",
    }
    // for range 문을 사용하여 모든 맵 요소
출력
    // Map은 unordered 이므로 순서는 무작위
    for key, val := range myMap {
        fmt.Println(key, val)
    }
}
```

Go 패키지

Go는 패키지(Package)를 통해 코드의 모듈화, 코드의 재사용 기능을 제공한다. Go는 패키지를 사용해서 작은 단위의 컴포넌트를 작성하고, 이러한 작은 패키지들을 활용해서 프로그램을 작성할 것을 권장한다.

Go는 실제 프로그램 개발에 필요한 많은 패키지들을 표준 라이브러리(Standard Library)로 제공한다. 이러한 표준 라이브러리 패키지들은 `GOROOT/src` (옛 버전의 경우 GOROOT/pkg) 안에 존재한다. GOROOT 환경변수는 Go 설치 디렉토리를 가리키는데, 보통 Go 설치시 자동으로 추가된다. 즉, 윈도우즈에서 Go를 설치했을 경우 디폴트로 `C:\Program Files\Go` 에 설치되며, GOROOT는 `C:\Program Files\Go`를 가리킨다.

Go에 사용하는 표준패키지는 https://pkg.go.dev/std 에 자세히 설명되어 있다.

Main 패키지

일반적으로 패키지는 라이브러리로서 사용되지만, "main" 이라고 명명된 패키지는 Go Compiler에 의해 특별하게 인식된다. 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. 패키지를 공유 라이브러리로 만들 때에는, main 패키지나 main 함수를 사용해서는 안된다.

패키지 Import

다른 패키지를 프로그램에서 사용하기 위해서는 import 를 사용하여 패키지를 포함시킨다. 예를 들어, Go의 표준 라이브러리인 fmt 패키지를 사용하기 위하여, import "fmt" 와 같이 해당 패키지를 포함시킬 것을 선언해 준다. Import 후에는 아래 예제처럼 fmt 패키지의 Println() 함수를 호출하여 사용할 수 있다.

```
package main

import "fmt"

func main(){
 fmt.Println("Hello")
}
```

패키지를 import 할 때, Go 컴파일러는 GOROOT 혹은 GOPATH 환경변수를 검색하는데, 표준 패키지는 GOROOT 안의 패키지에서 그리고 사용자 패키지나 3rd Party 패키지는 GOPATH/pkg 에서 패키지를 찾게 된다.

GOPATH 환경변수는 3rd Party 패키지를 갖는 라이브러리 디렉토리나 사용자 패키지가 있는 작업 디렉토리를 지정하게 되는데, 복수 개일 경우 세미콜론(윈도우즈의 경우)을 사용하여 연결한다.

패키지 Scope

패키지 내에는 함수, 구조체, 인터페이스, 메서드 등이 존재하는데, 이들의 이름(Identifier)이 첫문자를 대문자로 시작하면 이는 public 으로 사용할 수 있다. 즉, 패키지 외부에서 이들을 호출하거나 사용할 수 있게 된다. 반면, 이름이 소문자로 시작하면 이는 non-public 으로 패키지 내부에서만 사용될 수 있다.

패키지 init 함수와 alias

개발자가 패키지를 작성할 때, 패키지 실행시 처음으로 호출되는 `init()` 함수를 작성할 수 있다. 즉, init 함수는 패키지가 로드되면서 실행되는 함수로 별도의 호출 없이 자동으로 호출된다.

```
package testlib

var pop map[string]string

func init() {    // 패키지 로드시 map 초기화
    pop = make(map[string]string)
}
```

경우에 따라 패키지를 import 하면서 단지 그 패키지 안의 `init()` 함수만을 호출하고자 하는 케이스가 있다. 이런 경우는 패키지 `import` 시 _ 라는 alias 를 지정한다. 아래는 other/xlib 패키지를 호출하면서 _ alias를 지정한 예이다.

```
package main
import _ "other/xlib"
```

만약 패키지 이름이 동일하지만, 서로 다른 버젼 혹은 서로 다른 위치에서 로딩하고자 할 때는, 패키지 alias를 사용해서 구분할 수 있다.

```
import (
    mongo "other/mongo/db"
    mysql "other/mysql/db"
)
func main() {
    mondb := mongo.Get()
    mydb := mysql.Get()
    // ...
}
```

사용자 정의 패키지 생성

개발자는 사용자 정의 패키지를 만들어 재사용 가능한 컴포넌트를 만들어 사용할 수 있다. 사용자 정의 라이브러리 패키지는 일반적으로 폴더를 하나 만들고 그 폴더 안에 `.go` 파일들을 만들어 구성한다. 하나의 서브 폴더 안에 있는 `.go` 파일들은 동일한 패키지명을 가지며, 패키지명은 해당 폴더의 이름과 같게 한다. 즉, 해당 폴더에 있는 여러 go 파일들은 하나의 패키지로 묶인다.

예제로 간단한 패키지를 만들기 위해 `/src` 폴더 안에 (임의의 폴더명으로) `24lab.net/testlib` 폴더를 생성한 후에 다음 코드를 `music.go` 라는 파일에 저장한다. 여기서 패키지명은 폴더명과 동일하게 testlib로 정해주어야 한다. 패키지 폴더 안에 여러 파일들이 있을 경우에도, 동일하게 testlib 패키지명을 사용한다.

```
package testlib

import "fmt"

var pop map[string]string

func init() {
    pop = make(map[string]string)
    pop["Adele"] = "Hello"
    pop["Alicia Keys"] = "Fallin'"
    pop["John Legend"] = "All of Me"
}
// GetMusic : Popular music by singer
(외부에서 호출 가능)
func GetMusic(singer string) string {
    return pop[singer]
}
func getKeys() {  // 내부에서만 호출 가능
    for _, kv := range pop {
        fmt.Println(kv)
    }
}
```

Go 패키지 생성 예제

이제 사용자 정의 패키지를 사용하기 위해서 /src 안에 다음과 같은 코드를 작성해 보자. (주: main 패키지가 반드시 /src 밑에 있을 필요는 없다).

```
package main

import "24lab.net/testlib"

func main() {
    song := testlib.GetMusic("Alicia
Keys")
    println(song)
}
```

여기 코드에선 24lab.net/testlib 패키지를 import 하고 해당 패키지의 Export 함수인 GetMusic()을 호출하고 있다. 24lab.net/testlib 패키지가 있는 위치를 찾기 위해 GOROOT와 GOPATH의 경로를 사용하는데, GOROOT와 GOPATH에 있는 각 루트폴더의 src 밑에 24lab.net/testlib 폴더를 순서대로 찾게 된다. 즉, GOPATH가 `C:\GoApp;C:\GoSrc` 인경우, 지정된 라이브러리를 찾기 위해 다음과 같은 폴더를 순차적으로 검색하게 된다.

```
C:\Program
Files\Go\src\24lab.net\testlib (from
$GOROOT)
C:\GoApp\src\24lab.net\testlib (from
$GOPATH)
C:\GoSrc\src\24lab.net\testlib
```

구조체 (struct)

Go에서 struct는 Custom Data Type을 표현하는데 사용되는데, Go의 struct는 필드들의 집합체이며 필드들의 컨테이너이다. Go에서 struct는 필드 데이타만을 가지며, (행위를 표현하는) 메서드를 갖지 않는다.

Go 언어는 객체지향 프로그래밍(Object Oriented Programming, OOP)을 고유의 방식으로 지원한다. 즉, Go에는 전통적인 OOP 언어가 가지는 클래스, 객체, 상속 개념이 없다. 전통적인 OOP의 클래스(class)는 Go 언어에서 Custom 타입을 정의하는 struct로 표현되는데, 전통적인 OOP의 클래스가 필드와 메서드를 함께 갖는 것과 달리 Go 언어의 struct는 필드만을 가지며, 메서드는 별도로 분리하여 정의한다 (Go Method에서 설명).

Struct 선언

struct를 정의하기 위해서는 Custom Type을 정의하는데 사용하는 type 문을 사용한다. 예를 들어 name과 age 필드를 갖는 person 이라는 struct를 정의하기 위해서는 아래와 같은 type문을 사용할 수 있다. 만약 이 person 구조체를 패키지 외부에서 사용할 수 있게 하려면 (Go 패키지에서 설명하였듯이) struct명을 Person으로 변경하면 된다.

```
package main

import "fmt"

// struct 정의
type person struct {
    name string
    age  int
}
func main() {
    // person 객체 생성
    p := person{}
    // 필드값 설정
    p.name = "Lee"
    p.age = 10
    fmt.Println(p)
}
```

Struct 객체 생성

선언된 struct 타입으로부터 객체를 생성하는 방법은 몇 가지 방법들이 있다. 위의 예제처럼 `person{}` 를 사용하여 빈 person 객체를 먼저 할당하고, 나중에 그 필드값을 채워넣는 방법이 있다. struct 필드를 엑세스하기 위해서는 .(dot)을 사용한다.

struct 객체를 생성할 때, 초기값을 함께 할당하는 방법도 있다. 즉, 아래 첫번째 예처럼, struct 필드값을 순서적으로 {} 괄호안에 넣을 수 있으며, 두번째 예처럼 순서에 상관없이 필드명을 지정하고(named field) 그 값을 넣을 수 도 있다. 특히 두번째 예처럼 필드명을 지정하는 경우, 만약 일부 필드가 생략될 경우 생략된 필드들은 Zero value (정수인 경우 0, float인 경우 0.0, string인 경우 “”, 포인터인 경우 nil 등)를 갖는다.

```
var p1 person
p1 = person{"Bob", 20}
p2 := person{name: "Sean", age: 50}
```

또 다른 객체 생성 방법으로 Go 내장함수 new()를 쓸 수 있다. new()를 사용하면 모든 필드를 Zero value로 초기화하고 person 객체의 포인터(*person)를 리턴한다. 객체 포인터인 경우에도 필드 엑세스 시 .(dot)을 사용하는데, 이 때 포인터는 자동으로 Dereference 된다 (이는 C에서 포인터의 경우 → 을 사용하는 문법과 다르다).

```
p := new(person)
p.name = "Lee"  // p가 포인터라도 . 을 사용한다
```

Go에서 struct는 기본적으로 mutable 개체로서 필드값이 변화할 경우 (별도로 새 개체를 만들지 않고) 해당 개체 메모리에서 직접 변경된다. 하지만, struct 개체를 다른 함수의 파라미터로 넘긴다면, Pass by Value에 따라 객체를 복사해서 전달하게 된다. 그래서 만약 Pass by Reference로 struct를 전달하고자 한다면, struct의 포인터를 전달해야 한다.

생성자(constructor) 함수

때로 구조체(struct)의 필드가 사용 전에 초기화되어야 하는 경우가 있다. 예를 들어, struct 의 필드가 map 타입인 경우 map을 사전에 미리 초기화해 놓으면, 외부 struct 사용자가 매번 map을 초기화 해야 된다는 것을 기억할 필요가 없다. 이러한 목적을 위해 생성자 함수를 사용할 수 있다. 생성자 함수는 struct를 리턴하는 함수로서 그 함수 본문에서 필요한 필드를 초기화한다.

아래 예제에서 생성자 함수 `newDict()`는 dict라는 struct의 map 필드를 초기화한 후 그 struct 포인터를 리턴하고 있다. 이어 `main()` 함수에서 struct 개체를 만들 때 dict 를 직접 생성하지 않고 대신 `newDict()` 함수를 호출하여 이미 초기화된 data 맵 필드를 사용하고 있다.

```
package main

type dict struct {
    data map[int]string
}
//생성자 함수 정의
func newDict() *dict {
    d := dict{}
    d.data = map[int]string{}
    return &d //포인터 전달
}
func main() {
    dic := newDict() // 생성자 호출
    dic.data[1] = "A"
}
```

메서드

앞에서(Go 구조체) 언급했듯이 Go 언어는 객체지향 프로그래밍(OOP)을 고유의 방식으로 지원한다. 타 언어의 OOP의 클래스가 필드와 메서드를 함께 갖는 것과 달리 Go 언어에서는 struct가 필드만을 가지며, 메서드는 별도로 분리되어 정의된다. Go 메서드는 특별한 형태의 func 함수이다. 메서드는 함수 정의에서 func 키워드와 함수명 사이에 “그 함수가 어떤 struct를 위한 메서드인지”를 표시하게 된다. 흔히 receiver로 불리우는 이 부분은 메서드가 속한 struct 타입과 struct 변수명을 지정하는데, struct 변수명은 함수 내에서 마치 입력 파라미터처럼 사용된다. 예를 들어, 아래 예제는 Rect라는 struct를 정의하고 `area()` 라는 메서드를 정의하고 있다. func와 `area()` 사이에 Rect 타입의 r 이 정의되고 이를 함수 본문에서 사용하고 있다. 메서드가 선언된 이후에는 Rect 구조체의 객체는 rect.`area()` 문장처럼 `area()` 메소드를 struct 객체로부터 직접 호출할 수 있다.

```
package main
//Rect - struct 정의
type Rect struct {
    width, height int
}
//Rect의 area() 메소드
func (r Rect) area() int {
    return r.width * r.height
}
func main() {
    rect := Rect{10, 20}
    area := rect.area() //메서드 호출
    println(area)
}
```

Value receiver vs. Pointer receiver

Value receiver는 struct의 데이타를 복사(copy)하여 전달하며, Pointer receiver는 struct의 포인터만을 전달한다. Value receiver의 경우 만약 메서드내에서 그 struct의 필드값이 변경되더라도 호출자의 데이타는 변경되지 않는 반면, 포인터 receiver는 메서드 내의 필드값 변경이 그대로 호출자에서 반영된다.

위의 `Rect.area()` 메서드는 Value receiver를 표현한 것으로 만약 area() 메서드 내에서 width나 height가 변경되더라도 main() 함수의 rect 구조체의 필드값에는 변화가 없다. 하지만, 아래와 같이 이를 포인터 receiver로 변경한다면, 메서드 내 `r.width++` 필드 변경분이 main() 함수에서도 반영되기 때문에 출력값이 11, 220 이 된다.

```
// 포인터 Receiver
func (r *Rect) area2() int {
    r.width++
    return r.width * r.height
}
func main() {
    rect := Rect{10, 20}
    area := rect.area2() //메서드 호출
    println(rect.width, area) // 11 220 출력
}
```

인터페이스

구조체(struct)가 필드들의 집합체라면, interface는 메서드들의 집합체이다. interface는 타입(type)이 구현해야 하는 메서드 원형(prototype)들을 정의한다. 하나의 사용자 정의 타입이 interface를 구현하기 위해서는 단순히 그 인터페이스가 갖는 모든 메서드들을 구현하면 된다. 인터페이스는 struct와 마찬가지로 type 문을 사용하여 정의한다.

```
type Shape interface {
    area() float64
    perimeter() float64
}
```

인터페이스 구현

인터페이스를 구현하기 위해서는 해당 타입이 그 인터페이스의 메서드들을 모두 구현하면 되므로, 위의 Shape 인터페이스를 구현하기 위해서는 `area()`, `perimeter()` 2개의 메서드만 구현하면 된다. 예를 들어 Rect와 Circle 이라는 2개의 타입이 있을 때, Shape 인터페이스를 구현하기 위해서는 아래와 같이 각 타입별로 2개의 메서드를 구현해 주면 된다.

```
//Rect 정의
type Rect struct {
    width, height float64
}
//Circle 정의
type Circle struct {
    radius float64
}
//Rect 타입에 대한 Shape 인터페이스 구현
func (r Rect) area() float64 { return r.width * r.height }
```

```
func (r Rect) perimeter() float64 {
     return 2 * (r.width + r.height)
}
//Circle 타입에 대한 Shape 인터페이스 구현
func (c Circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c Circle) perimeter() float64 {
    return 2 * math.Pi * c.radius
}
```

인터페이스 사용

인터페이스를 사용하는 일반적인 예로, 함수가 파라미터로 인터페이스를 받아들이는 경우를 들 수 있다. 함수 파라미터가 interface 인 경우, 이는 어떤 타입이든 해당 인터페이스를 구현하기만 하면 모두 입력 파라미터로 사용될 수 있다는 것을 의미한다. 아래 예제에서 `showArea()` 함수는 Shape 인터페이스들을 파라미터로 받아들이고 있는데, 따라서 Rect와 Circle 처럼 Shape 인터페이스를 구현한 타입 객체들을 파라미터로 받을 수 있다. `showArea()` 함수 내에서 해당 인터페이스가 가진 메서드 즉 `area()` 혹은 `perimeter()`을 사용할 수 있다.

```
func main() {
    r := Rect{10., 20.}
    c := Circle{10}
    showArea(r, c)
}
func showArea(shapes ...Shape) {
    for _, s := range shapes {
        a := s.area() //인터페이스 메서드 호출
        println(a)
    }
}
```

인터페이스 타입

Go 프로그래밍을 하다보면 흔히 빈 인터페이스(empty interface)를 자주 접하게 되는데, 이는 흔히 인터페이스 타입(interface type)으로도 불리운다. 예를 들어, 여러 표준패키지들의 함수 Prototype을 살펴보면, 아래와 같이 빈 interface 가 자주 등장함을 볼 수 있다. 빈 interface는 `interface{}` 와 같이 표현한다.

```
func Marshal(v interface{}) ([]byte, error);
func Println(a ...interface{}) (n int, err error);
```

Empty interface는 메서드를 전혀 갖지 않는 빈 인터페이스로서, Go의 모든 Type은 적어도 0개의 메서드를 구현하므로, 흔히 Go에서 모든 Type을 나타내기 위해 빈 인터페이스를 사용한다. 즉, 빈 인터페이스는 어떠한 타입도 담을 수 있는 컨테이너라고 볼 수 있으며, 여러 다른 언어에서 흔히 일컫는 Dynamic Type 이라고 볼 수 있다.

- empty interface는 C#, Java 에서의 object라 볼 수 있으며, 혹은 C/C++ 에서의 void* 와 같다고 볼 수 있다)

아래 예제에서 인터페이스 타입 x는 정수 1을 담았다가 다시 문자열 Tom을 담고 있는데, 실행 결과는 마지막에 담은 Tom을 출력한다.

```
package main

import "fmt"

func main() {
    var x interface{}
    x = 1
    x = "Tom"
    printIt(x)
}
func printIt(v interface{}) {
    fmt.Println(v) //Tom
}
```

Type Assertion

Interface type의 x와 타입 T에 대하여 `x.(T)`로 표현했을 때, 이는 x가 nil이 아니며, x는 T 타입에 속한다는 점을 확인(assert)하는 것으로 이러한 표현을 "Type Assertion"이라 부른다. 만약 x가 nil 이거나 x의 타입이 T가 아니라면, 런타임 에러가 발생할 것이고, x가 T 타입인 경우는 T 타입의 x를 리턴한다. 즉, 아래 예제에서 변수 j는 a.(int)로부터 int형 변수 j가 된다.

```
func main() {
    var a interface{} = 1
    i := a       // a와 i 는 dynamic type, 값은 1
    j := a.(int) // j는 int 타입, 값은 1
    println(i)   // 포인터주소 출력
    println(j)   // 1 출력
}
```

에러처리

Go는 내장 타입으로 error 라는 interface 타입을 갖는다. Go 에러는 이 error 인터페이스를 통해서 주고 받게 되는데, 이 interface는 다음과 같은 하나의 메서드를 갖는다. 개발자는 이 인터페이스를 구현하는 커스텀 에러 타입을 만들 수 있다.

```
type error interface {
    Error() string
}
```

Go 에러처리

Go 함수가 결과와 에러를 함께 리턴한다면, 이 에러가 nil 인지를 체크해서 에러가 없는지를 체크할 수 있다. 예를 들어, `os.Open()` 함수는 func Open(name string) (`file *File`, `err error`) 과 같은 함수 원형을 갖는 것으로 첫번째는 File 포인터를 두번째는 error 인터페이스를 리턴한다. 그래서 이경우 두번째 error를 체크해서 `nil` 이면 에러가 없는 것이고, nil 이 아니면 `err.Error()` 로부터 해당 에러를 알 수 있다. 아래 예제는 파일을 오픈하는데 에러가 발생하면 에러메시지를 출력하고 빠져나가는 예이다 (주: log.Fatal() 은 메시지를 출력하고 os.Exit(1)을 호출하여 프로그램을 종료한다).

```
package main

import (
    "log"
    "os"
)

func main() {
    f, err := os.Open("C:\\temp\\1.txt")
    if err != nil {
        log.Fatal(err.Error())
    }
    println(f.Name())
}
```

또 다른 에러처리로서 error의 Type을 체크해서 에러 타입별로 별도의 에러 처리를 하는 방식이 있다. 아래 예제에서 otherFunc()를 호출한 후 error가 err로 리턴되었을 때, 이 err의 타입별로 다른 처리를 하는 것을 볼 수 있다. 디폴트의 경우는 에러타입이 `nil`인 경우로서 에러가 없는 경우이고, 에러가 있으면 다음 case문에서 그 에러타입이 MyError인지를 체크하고, 아니면 다음 case에서 일반 에러 케이스를 처리한다. 모든 에러는 error 인터페이스를 구현하므로 마지막 case문은 모든 에러에 적용된다.

```
_, err := otherFunc()
switch err.(type) {
default: // no error
    println("ok")
case MyError:
    log.Print("Log my error")
case error:
    log.Fatal(err.Error())
}
```

defer와 panic

지연실행 defer

Go 언어의 defer 키워드는 특정 문장 혹은 함수를 나중에 (defer를 호출하는 함수가 리턴하기 직전에) 실행하게 한다. 일반적으로 defer는 C#, Java 같은 언어에서의 finally 블럭처럼 마지막에 Clean-up 작업을 위해 사용된다. 아래 예제는 파일을 Open 한 후 바로 파일을 Close하는 작업을 defer로 쓰고 있다. 이는 차후 문장에서 어떤 에러가 발생하더라도 항상 파일을 Close할 수 있도록 한다.

```
package main

import "os"

func main() {
    f, err := os.Open("1.txt")
    if err != nil {
        panic(err)
    }
    // main 마지막에 파일 close 실행
    defer f.Close()
    // 파일 읽기
    bytes := make([]byte, 1024)
    f.Read(bytes)
    println(len(bytes))
}
```

panic 함수

Go 내장함수인 `panic()`함수는 현재 함수를 즉시 멈추고 현재 함수에 defer 함수들을 모두 실행한 후 즉시 리턴한다. 이러한 panic 모드 실행 방식은 다시 상위함수에도 똑같이 적용되고, 계속 콜스택을 타고 올라가며 적

용된다. 그리고 마지막에는 프로그램이 에러를 내고 종료하게 된다.

```
package main

import "os"

func main() {
    // 잘못된 파일명을 넣음
    openFile("Invalid.txt")
    // openFile() 안에서 panic이 실행되면
    // 아래 println 문장은 실행 안됨
    println("Done")
}
func openFile(fn string) {
    f, err := os.Open(fn)
    if err ≠ nil {
        panic(err)
    }
    defer f.Close()
}
```

recover 함수

Go 내장함수인 `recover()`함수는 panic 함수에 의한 패닉상태를 다시 정상상태로 되돌리는 함수이다. 위의 panic 예제에서는 main 함수에서 `println()` 이 호출되지 못하고 프로그램이 crash 하지만, 아래와 예제와 같이 recover 함수를 사용하면 panic 상태를 제거하고 `openFile()`의 다음 문장인 `println()` 을 호출하게 된다.

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // 잘못된 파일명을 넣음
    openFile("Invalid.txt")
    // recover에 의해
    // 이 문장 실행됨
    println("Done")
}
func openFile(fn string) {
    // defer 함수. panic 호출시 실행됨
    defer func() {
        if r := recover(); r ≠ nil {
            fmt.Println("OPEN ERROR", r)
        }
    }()
    f, err := os.Open(fn)
    if err ≠ nil {
        panic(err)
    }
    defer f.Close()
}
```

고루틴 (goroutine)

고루틴은 Go 런타임이 관리하는 Lightweight 논리적(혹은 가상적) 쓰레드이다. Go에서 “go” 키워드를 사용하여 함수를 호출하면, 런타임시 새로운 goroutine을 실행한다. goroutine은 비동기적으로(asynchronously) 함수루틴을 실행하므로, 여러 코드를 동시에(Concurrently) 실행하는데 사용된다.

“goroutine은 OS 쓰레드보다 훨씬 가볍게 비동기 Concurrent 처리를 구현하기 위하여 만든 것으로, 기본적으로 Go 런타임이 자체 관리한다. 메모리 측면에서도 OS 쓰레드가 1 메가바이트의 스택을 갖는 반면, goroutine은 이보다 훨씬 작은 몇 킬로바이트의 스택을 갖는다. Go 런타임은 Go루틴을 관리하면서 Go 채널을 통해 Go루틴 간의 통신을 쉽게 할 수 있도록 하였다.”

아래 예제에서 main 함수를 보면, 먼저 `say()`라는 함수를 동기적으로 호출하고, 다음으로 동일한 `say()` 함수를 비동기적으로 3번 호출하고 있다. 첫번째 동기적 호출은 `say()` 함수가 완전히 끝났을 때 다음 문장으로 이동하고, 다음 3개의 go `say()` 비동기 호출은 별도의 Go루틴들에서 동작하면서, 메인루틴은 계속 다음 문장(여기서는 time.Sleep)을 실행한다. 여기서 goroutine들은 그 실행순서가 일정하지 않으므로 프로그램 실행시 마다 다른 출력 결과를 나타낼 수 있다.

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 10; i++ {
        fmt.Println(s, "***", i)
    }
}
func main() {
    // 함수를 동기적으로 실행
    say("Sync")
    // 함수를 비동기적으로 실행
    go say("Async1")
    go say("Async2")
    go say("Async3")
    // 3초 대기
    time.Sleep(time.Second * 3)
}
```

익명함수 Go루틴

Go루틴(goroutine)은 익명함수(anonymous function)에 대해 사용할 수도 있다. 즉, go 키워드 뒤에 익명함수를 바로 정의하는 것으로, 이는 익명함수를 비동기로 실행하게 된다. 아래 예제에서 첫번째 익명함수는 간단히 Hello 문자열을 출력하는데, 이를 goroutine으로 실행하면 비동기적으로 그 익명함수를 실행하게 된다. 두번째 익명함수는 파라미터를 전달하는 예제로 익명함수에 파라미터가 있는 경우, go 익명함수 바로 뒤에 파라미터(Hi)를 함께 전달하게 된다.

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    // WaitGroup 생성. 2개의 Go루틴을 기다림.
    var wait sync.WaitGroup
    wait.Add(2)
    // 익명함수를 사용한 goroutine
    go func() {
        defer wait.Done() //끝나면 .Done() 호출
        fmt.Println("Hello")
    }()
    // 익명함수에 파라미터 전달
    go func(msg string) {
        defer wait.Done() //끝나면 .Done() 호출
        fmt.Println(msg)
    }("Hi")
    wait.Wait() //Go루틴 모두 끝날 때까지 대기
}
```

여기서 sync.WaitGroup을 사용하고 있는데, 이는 기본적으로 여러 Go루틴들이 끝날 때까지 기다리는 역활을 한다. WaitGroup을 사용하기 위해서는 먼저 `Add()` 메소드에 몇 개의 Go루틴을 기다릴 것인지 지정하고, 각 Go루틴에서 `Done()` 메서드를 호출한다 (여기서는 defer 를 사용하였다). 그리고 메인루틴에서는 Wait() 메서드를 호출하여, Go루틴들이 모두 끝나기를 기다린다.

다중 CPU 처리

Go는 디폴트로 1개의 CPU를 사용한다. 즉, 여러 개의 Go 루틴을 만들더라도, 1개의 CPU에서 작업을 시분할하여 처리한다 (Concurrent 처리). 만약 머신이 복수개의 CPU를 가진 경우, Go 프로그램을 다중 CPU에서 병렬처리 (Parallel 처리)하게 할 수 있는데, 병렬처리를 위해서는 아래와 같이 runtime.GOMAXPROCS(CPU 수) 함수를 호출하여야 한다(여기서 CPU 수는 Logical CPU 수를 가리킨다).

```
package main

import (
    "runtime"
)

func main() {
    // 4개의 CPU 사용
    runtime.GOMAXPROCS(4)
    // ...
}
```

Concurrency (혹은 Concurrent 처리)와 Parallelism (혹은 Parallel 처리)는 비슷하게 들리지만, 전혀 다른 개념이다.

“프로그래밍에서 동시성은 독립적으로 실행되는 프로세스를 구성하는 것이고, 병렬성은 (관련성이 있을 수 있는) 계산을 동시에 실행하는 것입니다. 동시성은 한 번에 많은 일을 처리하는 것입니다. 병렬성은 한 번에 많은 작업을 수행하는 것입니다.”

채널

Go 채널은 그 채널을 통하여 데이타를 주고 받는 통로라 볼 수 있는데, 채널은 make() 함수를 통해 미리 생성되어야 하며, 채널 연산자 ← 을 통해 데이타를 보내고 받는다. 채널은 흔히 goroutine들 사이 데이타를 주고받는데 사용되는데, 상대편이 준비될 때까지 채널에서

대기함으로써 별도의 lock을 걸지 않고 데이타를 동기화하는데 사용된다.

아래 예제는 정수형 채널을 생성하고, 한 goroutine 에서 그 채널에 123이란 정수 데이타를 보낸 후, 이를 다시 메인 루틴에서 채널로부터 123 데이타를 받는 코드이다. 채널을 생성할 때는 make() 함수에 어떤 타입의 데이타를 채널에서 주고 받을지를 미리 지정해 주어야 한다. 채널로 데이타를 보낼 때는 채널명 ← 데이타 와 같이 사용하고, 채널로부터 데이타를 받을 경우는 ← 채널명 와 같이 사용한다. 아래 예제에서 메인 루틴은 마지막에서 채널로부터 데이타를 받고 있는데, 상대편 goroutine에서 데이타를 전송할 때까지는 계속 대기하게 된다. 따라서, 이 예제에서는 time.Sleep() 이나 fmt.Scanf() 같이 goroutine 이 끝날 때까지 기다리는 코드를 적지 않았다.

```
package main

func main() {
  // 정수형 채널을 생성한다
  ch := make(chan int)
  go func() {
    ch ← 123   //채널에 123을 보낸다
  }()
  var i int
  i = ← ch  // 채널로부터 123을 받는다
  println(i)
}
```

Go 채널은 수신자와 송신자가 서로를 기다리는 속성때문에, 이를 이용하여 (다음 예제와 같이) Go루틴이 끝날 때까지 기다리는 기능을 구현할 수 있다. 즉, 익명함수를 사용한 한 Go 루틴에서 어떤 작업이 실행되고 있을 때, 메인루틴은 ←done 에서 계속 수신하며 대기하고 있게 된다. 익명함수 Go 루틴에서 작업이 끝난 후, done채널에 true를 보내면, 수신자 메인루틴은 이를 받고 프로그램을 끝내게 된다.

```
package main

import "fmt"

func main() {
    done := make(chan bool)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(i)
        }
        done ← true
    }()
    // 위의 Go루틴이 끝날 때까지 대기
    ←done
}
```

Go 채널 버퍼링

Go 채널은 2가지의 채널이 있는데, Unbuffered Channel과 Buffered Channel이 있다. 위의 예제에서의 Go 채널은 Unbuffered Channel로서 이 채널에서는 하나의 수신자가 데이타를 받을 때까지 송신자가 데이타를 보내는 채널에 묶여 있게 된다. 하지만, Buffered Channel을 사용하면 비록 수신자가 받을 준비가 되어 있지 않을 지라도 지정된 버퍼만큼 데이타를 보내고 계속 다른 일을 수행할 수 있다. 버퍼 채널은 make(chan type, N) 함수를 통해 생성되는데, 두번째 파라미터 N에 사용할 버퍼 갯수를 넣는다. 예를 들어, make(chan int, 10)은 10개의 정수형을 갖는 버퍼 채널을 만든다. 버퍼 채널을 이용하지 않는 경우, 아래와 같은 코드는 에러를 발생시킨다. 왜냐하면 메인루틴에서 채널에 1을 보내면서 상대편 수신자를 기다리고 있는데, 이 채널을 받는 수신자 Go루틴이 없기 때문이다.

```
package main

import "fmt"

func main() {
  c := make(chan int)
  c ← 1   //수신루틴이 없으므로 데드락
  fmt.Println(←c) //코멘트해도 데드락 (별도의 Go루틴없기 때문)
}
```

하지만 아래와 같이 버퍼채널을 사용하면, 수신자가 당장 없더라도 최대버퍼 수까지 데이타를 보낼 수 있으므로, 에러가 발생하지 않는다.

```
package main

import "fmt"

func main() {
    ch := make(chan int, 1)
    //수신자가 없더라도 보낼 수 있다.
    ch ← 101
    fmt.Println(←ch)
}
```

채널 파라미터

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. 아래 예제에서 만약 sendChan() 함수 안에서 x := ← ch 를 실행하면 송신전용 채널에 수신을 시도하므로 에러가 발생한다.

```
package main

import "fmt"

func main() {
    ch := make(chan string, 1)
    sendChan(ch)
    receiveChan(ch)
}
func sendChan(ch chan← string) {
    ch ← "Data"
    // x := ←ch // 에러발생
}
func receiveChan(ch ←chan string) {
    data := ←ch
    fmt.Println(data)
}
```

채널 닫기

채널을 오픈한 후 데이타를 송신한 후, close()함수를 사용하여 채널을 닫을 수 있다. 채널을 닫게 되면, 해당 채널로는 더이상 송신을 할 수 없지만, 채널이 닫힌 이후에도 계속 수신은 가능하다. 채널 수신에 사용되는 ← ch 은 두개의 리턴값을 갖는데, 첫째는 채널 메시지이고, 두번째는 수신이 제대로 되었는가를 나타낸다. 만약 채널이 닫혔다면, 두번째 리턴값은 false를 리턴한다.

```
package main

func main() {
    ch := make(chan int, 2)
    // 채널에 송신
    ch ← 1
    ch ← 2
    // 채널을 닫는다
    close(ch)
    // 채널 수신
    println(←ch)
    println(←ch)
    if _, success := ←ch; !success {
        println("더이상 데이타 없음.")
    }
}
```

채널 range 문

채널에서 송신자가 송신을 한 후, 채널을 닫을 수 있다. 그리고 수신자는 임의의 갯수의 데이타를 채널이 닫힐 때까지 계속 수신할 수 있다. 아래 예제는 이러한 송수신 방법을 표현한 것으로, 수신자는 채널이 닫히는 것을 체크하면서 계속 루프를 돌게 된다. 방법1은 무한 for 루프 안에서 if 문으로 수신 채널의 두번째 파라미터를 체크하는 방식이고, 방법2는 방법1과 동일한 표현이지만, for range문으로 보다 간결하게 표현한 것이다. 채널 range문은 range 키워드 다음의 채널로부터 계속 수신하다가 채널이 닫힌 것을 감지하면 for 루프를 종료한다.

```
package main

func main() {
    ch := make(chan int, 2)
    // 채널에 송신
    ch ← 1
    ch ← 2
    // 채널을 닫는다
    close(ch)
    // 방법1: 채널이 닫힌 것을 감지할 때까지 계속 수신
    /*
    for {
        if i, success := ←ch; success {
            println(i)
        } else {
            break
        }
    }
    */
    // 방법2: 위 표현과 동일한 채널 range 문
    for i := range ch {
        println(i)
    }
}
```

채널 select 문

Go의 select문은 복수 채널들을 기다리면서 준비된 (데이타를 보내온) 채널을 실행하는 기능을 제공한다. 즉, select문은 여러 개의 case문에서 각각 다른 채널을 기다리다가 준비가 된 채널 case를 실행하는 것이다.

select문은 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.

아래 예제는 for 루프 안에 select 문을 쓰면서 두개의 goroutine이 모두 실행되기를 기다리고 있다. 첫번째 `run1()`이 1초간 실행되고 done1 채널로부터 수신하여 해당 case를 실행하고, 다시 for 루프를 돈다. for 루프를 다시 돌면서 다시 select문이 실행되는데, 다음 `run2()`가 2초후에 실행되고 done2 채널로부터 수신하여 해당 case를 실행하게 된다. done2 채널 case문에 break EXIT 이 있는데, 이 문장으로 인해 for 루프를 빠져나와 EXIT 레이블로 이동하게 된다. Go의 "break 레이블" 문은 `C/C#` 등의 언어에서의 goto 문과 다른데, Go에서는 해당 레이블로 이동한 후 자신이 빠져나온 루프 다음 문장을 실행하게 된다. 따라서, 여기서는 for 루프 다음 즉 `main()` 함수의 끝에 다다르게 된다.

```
package main

import "time"

func main() {
    done1 := make(chan bool)
    done2 := make(chan bool)
    go run1(done1)
    go run2(done2)
EXIT:
    for {
        select {
        case <-done1:
            println("run1 완료")
        case <-done2:
            println("run2 완료")
            break EXIT
        }
    }
}
func run1(done chan bool) {
    time.Sleep(1 * time.Second)
    done <- true
}
func run2(done chan bool) {
    time.Sleep(2 * time.Second)
    done <- true
}
```