



Chapter 6 연산자

📅 날짜	@November 3, 2021
👤 발표자	김하연

6.1 산술 연산자

산술 연산자는 숫자 연산을 하는 연산자이다. 사칙 연산, 비트연산, 시프트 연산이 속한다.

산술 연산자

≡ 구분	Aa 연산자	≡ 연산	≡ 피연산자 타입
사칙연산과 나머지	\pm	덧셈	정수, 실수, 복소수, 문자열
	$=$	뺄셈	정수, 실수, 복소수
	$*$	곱셈	정수, 실수, 복소수
	$/$	나눗셈	정수, 실수, 복소수
	$\%$	나머지	정수
비트 연산	$\&$	AND 비트 연산	정수
	$ $	OR 비트 연산	정수
	\wedge	XOR 비트 연산	정수
	$\&\wedge$	비트 클리어	정수
시프트 연산	$\leq\leq$	왼쪽 시프트	정수 << 양의 정수
	$\geq\geq$	오른쪽 시프트	정수 >> 양의 정수

6.1.1 연산의 결과 타입

- Go 언어에서 모든 연산자의 각 항의 타입은 항상 같다. (시프트 연산 제외)
- 다른 타입을 연산하려면 타입 변환 후 연산해야한다.
- 연산의 결과타입도 인수 타입과 같다.

```

package main

import "fmt"

func main() {
    var x int32 = 7
    var y int32 = 3

    var s float32 = 3.14
    var t float32 = 5

    fmt.Println("x + y = ", x+y)
    fmt.Println("x - y = ", x-y)
    fmt.Println("x * y = ", x*y)
    fmt.Println("x / y = ", x/y)
    fmt.Println("x % y = ", x%y)

    fmt.Println("s * t = ", s*t)
    fmt.Println("s / t = ", s/t)
}

```

출력값

```

x + y = 10
x - y = 4
x * y = 21
x / y = 2 //❶
x % y = 1
s * t = 15.700001
s / t = 0.628




```




❶ 7/3의 수행 결과는 2.333... → 2 (정수 타입)

6.1.2 비트 연산자

- &, |, ^, &^는 비트 단위로 연산하는 비트 연산자이다.
- 피연산자로 정수만 가능하다.
- 비트 단위로 연산을 수행한다 → **정숫값을 2진수로 표현한 뒤 계산한다**




&(AND 연산자)

 A	 B	 A & B
0	0	<u>0</u>

 A	 B	 A & B
1	0	<u>0</u>
0	1	<u>0</u>
1	1	<u>1</u>




- A와 B 모두 1인 비트만 1이 된다.
- 예) $10 \& 34 \rightarrow 0000\ 1010 \& 0010\ 0010 = 0000\ 00010 \rightarrow 2$

|(OR 연산자)

 A	 B	 A B
0	0	<u>0</u>
1	0	<u>1</u>
0	1	<u>1</u>
1	1	<u>1</u>

- A와 B 중 하나라도 1이면 1이 된다.
- 예) $= 0010\ 1010 \rightarrow 42$

^(XOR 연산자)

 A	 B	 A ^ B
0	0	<u>0</u>
1	0	<u>1</u>
0	1	<u>1</u>
1	1	<u>0</u>

- A와 B가 다르면 1이 된다.
- 예) $10 \wedge 34 \rightarrow 0000\ 1010 \wedge 0010\ 0010 = 0010\ 1000 \rightarrow 40$
- ^A처럼 단독으로 사용 가능하다. \rightarrow 비트 반전

```
package main

import "fmt"
```

```
func main() {
    var x1 int8 = 34    //8비트 정수, 0010 0010
    var x2 int16 = 34   //16비트 정수, 0000 0000 0010 0010
    var x3 uint8 = 34   //8비트 부호 없는 정수
    var x4 uint16 = 34  //16비트 부호 없는 정수

    fmt.Printf("%d = %d, \t %08b\n", x1, ^x1, uint8(^x1))
    fmt.Printf("%d = %d, \t %016b\n", x2, ^x2, uint16(^x2))
    fmt.Printf("%d = %d, \t %08b\n", x3, ^x3, ^x3)
    fmt.Printf("%d = %d, \t %016b\n", x4, ^x4, ^x4)
}
```

출력값

```
^34 =   -35,      11011101
^34 =   -35,      1111111111011101
^34 =    221,      11011101
^34 = 65501,      1111111111011101
```

- 부호가 있고 없고에 따라 비트 반전한 값이 다를 수 있다.
- ^x1값이 음수 값이기 때문에 그대로 표현하면 -가 표시된다. → 모든 비트를 표시하기 위해 uint8()로 타입 변환을 했다

&^(비트 클리어 연산자)

- 특정 비트를 0으로 바꾸는 연산자
- 우변값에 해당하는 비트를 클리어한다
- ^ → & 수행
- 예) 10 &^ 2
 - 1단계: ^ 연산 수행 → 2 = 1000 0010 → ^2 = 0111 1101
 - 2단계: & 연산 수행 → 0000 1010 & 1111 1101 = 0000 1000
 - 10 = 0000 1010와 0000 1000 비교: 2에 해당하는 비트가 0으로 바뀌었다.

6.1.3 시프트 연산자

- 비트를 왼쪽 또는 오른쪽으로 밀거나 당기는 연산자이다.
- << (왼쪽 시프트)와 >> (오른쪽 시프트)를 지원

<<(왼쪽 시프트)

- 오른쪽 피연산자값 만큼 전체 비트를 왼쪽으로 밀어낸다.
- 빈 자리는 0으로 채워지고, 자릿수를 벗어난 비트는 버려진다.
- 오른쪽 피연산자는 반드시 양의 정수이다. (아닐 시 비정상 종료)

```
package main

import "fmt"

func main() {
    var x int8 = 4
    var y int8 = 64
    fmt.Printf("x:%08b x<<2: %08b x<<2: %d\n", x, x<<2, x<<2)
    fmt.Printf("y:%08b y<<2: %08b y<<2: %d\n", y, y<<2, y<<2)
}
```

출력값

```
x:00000100 x<<2: 00010000 x<<2: 16
y:01000000 y<<2: 00000000 y<<2: 0
```

- $x \ll 2 = x$ 의 제곱 = 16 (단, 범위를 벗어나면 2의 승수배가 나오지 않는다.)
- $y \ll 2 = 0$ (256이 아니다! 8비트를 벗어나서 01이 버려져 0이 되었다.)

>>(오른쪽 시프트)

- 비트값을 오른쪽으로 민다.
- 오른쪽 피연산자는 반드시 양의 정수이다.
- 왼쪽에 추가되는 비트는 최상위 비트값과 같은 비트값이 추가된다.
 - 부호 있는 정수: 부호와 같은 값으로 (음수이면 1, 양수이면 0)
 - 부호 없는 정수: 0으로

```
package main

import "fmt"

func main() {
```

```

var x int8 = 16    //부호 비트값이 0인 수
var y int8 = -128 //부호 비트값이 1인 수
var z int8 = -1    //모든 비트값이 1인 정수
var w uint8 = 128 //최상위 비트값이 1인 양수
fmt.Printf("x:%08b x>>2: %08b x>>2: %d\n", x, x>>2, x>>2)
fmt.Printf("y:%08b y>>2: %08b y>>2: %d\n", uint8(y), uint8(y>>2), y>>2)
fmt.Printf("z:%08b z>>2: %08b z>>2: %d\n", uint8(z), uint8(z>>2), z>>2)
fmt.Printf("w:%08b w>>2: %08b w>>2: %d\n", uint8(w), uint8(w>>2), w>>2)
}

```

출력값

```

x:00010000 x>>2: 00000100 x>>2: 4
y:10000000 y>>2: 11100000 y>>2: -32
z:11111111 z>>2: 11111111 z>>2: -1
w:10000000 w>>2: 00100000 w>>2: 32

```

- $x \gg 2 = x$ 를 2의 승수로 나눈 결과 = 4
- $y = -128 = 1000\ 0000$, 최상위 비트값이 1이므로 오른쪽으로 밀어낼 때마다 1이 채워진다. $\rightarrow y \gg 2 = 1110\ 0000 = -32$
- 값의 경계에서는 올바른 2의 승수로 나눈 값을 표시하지 못한다. ($-1 \gg 2 = -1$)
- w 는 부호 없는 정수이기 때문에 왼쪽에 0이 채워진다.

6.2 비교 연산자

양변을 비교해서 조건에 만족하면 불리언값 true를, 만족하지 못할 경우 false를 반환하는 연산자이다. $==, \neq, <, >, \leq, \geq$ 연산자를 제공한다.

비교 연산자

Aa 연산자	≡ 설명	≡ 반환값
$==$	같다	참이면 true
\neq	다르다	
\leq	작다	
\geq	크다	
\leq	작거나 같다	
\geq	크거나 같다	

비교 연산자는 분기문(if문, switch문)과 반복문(for문)에서 주로 사용한다.

비교 연산자를 사용할 때 몇 가지 주의할 점이 있다. 부호 있는 정수를 사용할 때 발생하는 오버플로와 언더플로 문제, 실수끼리의 비교이다.

6.2.1 정수 오버플로

- 오버플로: 정수가 정수 타입의 범위를 벗어난 경우 값이 비정상적으로 변화하는 현상
- 예) $x < x + 1$ 가 false가 되는 경우

```
package main

import "fmt"

func main() {
    var x int8 = 127 //8비트 부호가 있는 정수 최댓값

    fmt.Printf("%d < %d + 1: %v\n", x, x, x < x+1)
    fmt.Printf("x\t= %4d, %08b\n", x, x)
    fmt.Printf("x + 1\t= %4d, %08b\n", x+1, x+1)
    fmt.Printf("x + 2\t= %4d, %08b\n", x+2, x+2)
    fmt.Printf("x + 3\t= %4d, %08b\n", x+3, x+3)

    var y int8 = -128 //8비트 부호있는 정수 최솟값
    fmt.Printf("%d > %d - 1: %v\n", y, y, y > y-1)
    fmt.Printf("y\t= %4d, %08b\n", y, y)
    fmt.Printf("y - 1\t= %4d, %08b\n", y-1, y-1)
}
```

출력값

```
127 < 127 + 1: false
x      = 127, 01111111
x + 1  = -128, -10000000
x + 2  = -127, -1111111
x + 3  = -126, -1111110
-128 > -128 - 1: false
y      = -128, -10000000
y - 1  = 127, 01111111
```

- int8 타입 값의 범위: -128 ~ 127
- $x + 1 = 0111\ 1111 + 1 = 1000\ 000 \rightarrow$ 가장 작은 값이 된다.
- 즉, $127 < 127 + 1$ 은 false가 된다.

6.3.1 정수 언더플로

- 오버플로와 반대로 정수 타입이 표현할 수 있는 가장 작은 값에서 -1을 했을 때 가장 큰 값으로 바뀐다.
- $y - 1 = 1000\ 0000 - 1 = 0111\ 1111 \rightarrow$ 가장 큰 값이 된다.
- 즉, $-128 > -128 - 1$ 은 false가 된다.

6.2.3 float 비교 연산

```
package main

import "fmt"

func main() {
    var a float64 = 0.1
    var b float64 = 0.2
    var c float64 = 0.3

    fmt.Printf("%f + %f == %f : %v\n", a, b, c, a+b == c)
    fmt.Println(a + b)
}
```

출력값

```
0.100000 + 0.200000 == 0.300000 : false
0.30000000000000004
```

- $0.1 + 0.2 == 0.3$ 을 수행한 결과 `false`가 출력된다.
- 이유: float64 표현 방식으로 생긴 오차 때문이다. float 표현은 이런 오차를 가지고 있기 때문에 같다 연산 시 예기치 못한 오류가 발생 가능하다.

6.3 실수 오차

컴퓨터에서 실숫값을 표현할 때 지수부와 소수부로 나뉘어서 표현한다. 컴퓨터는 지수부와 소수부가 10진수 기준이 아니라 2진수 기준으로 되어 있어 10진수 실수를 정확히 표현하기 어렵다.

예를 들어, $0.375 = 3e-1 + 7e-2 + 5e-3$ 이다. 하지만 컴퓨터는 2진수 숫자 체계를 사용한다. 0.375 를 2진수로 나타내면 $1*2^{(-1)} + 1*2^{(-3)}$ 이다. 대부분의 소수점 이하 숫자들은 2의 음수 승수로 표현하기 어렵다. 0.376 을 표현하려 해도 아무리 작은 2의 마이너스 승수값을 더해도 절대 0.376 값이 나오지 않는다.

그래서 0.376값은 float32 타입으로 최대한 가깝게 표현한 값이 0.37599998712539..이다. 오차가 발생할 수 밖에 없다.

6.3.1 작은 오차 무시하기

- 실숫값을 정확히 표현할 수 없어서 오차가 생길 수 밖에 없다. 그래서 아주 작은 오차는 무시하는 방법으로 값을 비교할 수 있다.

```
package main

import "fmt"

const epsilon = 0.000001 //매우 작은 값

func equal(a, b float64) bool {
    if a > b {
        if a-b <= epsilon { //작은 차이는 무시한다
            return true
        } else {
            return false
        }
    } else {
        if b-a <= epsilon {
            return true
        } else {
            return false
        }
    }
}

func main() {
    var a float64 = 0.1
    var b float64 = 0.2
    var c float64 = 0.3

    fmt.Printf("%.18f + %.18f = %.18f\n", a, b, a+b)
    fmt.Printf("%.18f == %.18f : %v\n", c, a+b, equal(a+b, c))

    a = 0.000000000000004
    b = 0.000000000000002
    c = 0.000000000000007

    fmt.Printf("%g == %g : %v\n", c, a+b, equal(a+b, c))
}
```

출력값

```
0.1000000000000000006 + 0.2000000000000000011 = 0.3000000000000000044
0.2999999999999999989 == 0.3000000000000000044 : true
7e-13 == 6.0000000000000001e-13 : true
```

- 매우 작은 상숫값을 선언한다. 무시할 오차 한계를 정의한 값이다.
- `equal()` 함수는 두 값의 차이가 `epsilon`보다 작을 경우 두 값이 같다고 간주한다.
- 소수점 이하 18자리짜리 출력하면 0.1이 정확히 0.1이 아니고 0.2가 정확히 0.2가 아니다. 그래서 0.1 + 0.2 역시 정확히 0.3이 아니라 오차가 발생한다.
- `c`값 역시 0.3이 아니다. `a + b`와 `c` 값의 차이가 `epsilon`값보다 작기 때문에 두 값을 같은 값으로 간주하여 `true`가 출력된다.

6.3.2 오차를 없애는 더 나은 방법

- 하지만 지정한 오차를 무시하는 것은 좋은 방법은 아니다. 얼마만큼의 오차가 무시할만큼 작은 오차인지가 문제가 된다. `float64`의 범위는 $e^{-138} \sim e^{308}$ 까지 매우 크다. 위 예제의 `epsilon`값이 200.345같은 값에 비하면 매우 작지만 0.0000234에 비하면 크게 작지 않다. 즉 경우에 따라 `epsilon`값이 무시할 만큼 작거나 그렇지 않기도 한다.
- 그럼 어떻게 해야할까요? 가장 간편하고 좋은 방법은 1비트 차이만큼 비교하는 것이다. 실수 표현은 지수부와 소수부로 나뉘지기 때문에 해당 지수부 표현에서 가장 작은 차이는 가장 오른쪽 비트값 하나만큼이다.

0.29999998211...

0 011.1110.1 001.1001.1001.1001.1001.1001.

0.3000001192...

0 011.1110.1 001.1001.1001.1001.1001.1010.

- 0.3은 `float32` 타입에서 위 두 값 중 하나로 표현해야 한다. 0.3과 정확히 같지는 않지만 아주 조금 작거나 아주 조금 크다. 두 값은 가장 마지막 비트 차이밖에 나지 않는다. 즉 0.3을 표현할 수 있는 값의 실수 타입 범위에서는 가장 작은 차이이다. 그래서 **만약 어떤 값이 이 두 값 사이라면 0.3과 같다고 간주하는 것이다.**
- 그럼 어떻게 가장 마지막 비트가 1비트만큼 차이나는지 알 수 있을까? Go언어에서는 `math` 패키지에서 `Nextafter()` 함수를 제공한다.

```
func Nextafter(x, y float64) (r float64)
```

- 이 함수는 float64 타입 2개를 받아서 float64 타입 하나를 반환한다. x에서 y를 향해서 1비트만 조정된 값을 반환한다. 만약 x가 y보다 작다면 x에서 1비트만큼 증가시키고 그렇지 않다면 x에서 1비트만큼 감소시킨 값을 반환한다.
- 즉, 가장 작은 오차만큼을 y를 향해서 더하거나 빼준다. 이 함수를 이용해 실숫값 대소비교를 할 수 있다.

```
package main

import (
    "fmt"
    "math"
)

func equal(a, b float64) bool {
    return math.Nextafter(a, b) == b //Nextafter() 로 값 비교.
}

func main() {
    var a float64 = 0.1
    var b float64 = 0.2
    var c float64 = 0.3

    fmt.Printf("%.18f + %.18f = %.18f\n", a, b, a+b)
    fmt.Printf("%.18f == %.18f : %v\n", c, a+b, equal(a+b, c))

    a = 0.0000000000000004
    b = 0.0000000000000002
    c = 0.0000000000000007

    fmt.Printf("%g == %g : %v\n", c, a+b, equal(a+b, c))
}
```

출력값

```
0.100000000000000006 + 0.200000000000000011 = 0.300000000000000044
0.299999999999999989 == 0.300000000000000044 : true
7e-13 == 6.000000000000001e-13 : false
```

- Nextafter() 함수를 이용해 값의 정밀도에 따라서 가장 작은 비트값만큼의 오차 범위만 인정한다.
- 큰 값과 작은 값 모두 제대로 처리되었다.

- 잊지 말아야 할 것은 어디까지나 **오차를 무시하는 방법**이라는 점이다. 그 오차가 작을 뿐, 정확한 계산은 아니다.
- math/big 패키지에서 제공하는 Float 객체를 이용하면 정밀도를 직접 조정할 수 있어 더 정확한 수치 계산을 할 수 있다.

```
package main

import (
    "fmt"
    "math/big"
)

func main() {
    a, _ := new(big.Float).SetString("0.1")
    b, _ := new(big.Float).SetString("0.2")
    c, _ := new(big.Float).SetString("0.3")

    d := new(big.Float).Add(a, b)
    fmt.Println(a, b, c, d)
    fmt.Println(c.Cmp(d))
}
```

출력값

```
0.1 0.2 0.3 0.3
0
```

- c와 a와 b를 비교한다. x.Cmp(y) 형식을 사용하였다. 반환값은 x가 작은 경우 -1, 큰 경우 1, 같은 경우 0이다. 0이 출력됐으니 두 값은 같다.

정리 : 허용 오차를 줄이는 방법 1) 매우 작은 값을 설정해서 오차를 무시하는 방법 2) Nextafter() 함수를 사용하는 방법 3) math/big의 Float 패키지를 사용하는 방법

6.4 논리 연산자

논리 연산자는 불리언 피연산자를 대상으로 연산해 결과로 true나 false를 반환한다. &&, ||, ! 연산자를 제공한다.

논리 연산자

 연산자	 연산	 설명
---	--	--

Aa 연산자	≡ 연산	≡ 설명
&&	AND	양변이 true이면 true 반환
	OR	양변 중 하나라도 true이면 true 반환
!	NOT	true이면 false, false이면 true 반환

&&(AND 논리 연산자)

≡ A	≡ B	Aa A && B
false	false	<u>false</u>
true	false	<u>false</u>
false	true	<u>false</u>
true	true	<u>true</u>

||(OR 논리 연산자)

≡ A	≡ B	Aa A && B
false	false	<u>false</u>
true	false	<u>true</u>
false	true	<u>true</u>
true	true	<u>true</u>

!(NOT 논리 연산자)

≡ A	Aa !A
false	<u>true</u>
true	<u>false</u>

6.5 대입 연산자

= 대입 연산자는 우변값을 좌변(메모리 공간)에 복사한다. 좌변은 반드시 저장할 공간이 있는 변수가 와야 한다.

```
var a int
a = 10
```

대입 연산자는 아무런 값도 반환하지 않는다.

```
var a int
var b int
a = b = 10 //오류 발생 - b = 10 구문은 어떤 결과도 반환하지 않음
```

아래와 같이 수정한다.

```
var a int
var b int
b = 10
a = b
```

6.5.1 복수 대입 연산자

- 여러 값을 한 번에 대입할 수 있다.
- 우변 개수에 맞춰서 좌변 변수 개수도 맞춰야 한다.
- 예) a, b = 3, 4 → a 변수에는 3 대입, b 변수에는 4 대입

```
package main

import "fmt"

func main() {
    var a int = 10
    var b int = 20

    a, b = b, a

    fmt.Println(a, b)
}
```

출력값

```
20 10
```

- a와 b의 값을 서로 바꾼다.

6.5.2 복합 대입 연산자

- 대입 연산자 앞에 다른 산술 연산자를 붙여 변수의 값과 연산의 결과를 다시 변수에 대입한다.
- 모든 산술 연산자는 다 복합 대입 연산자로 쓸 수 있다. ($+=$, $-=$, $*=$, $/=$, $\%=$, $\&=$, $|=$, $\wedge=$ 등)

6.5.3 증감 연산자

- 변수값을 1 증가하거나 1 감소하는 구문은 자주 사용된다.
- $++$, $--$ 를 제공한다.
- $a++$ 은 $a+=1$ 과 같은 의미. $a--$ 은 $a-=1$ 과 같은 의미

6.5.4 그 외 연산자

다른 장에서 다루는 연산자

Aa 연산자	≡ 설명
[]	배열의 요소에 접근할 때 사용
.	구조체나 패키지 요소에 접근할 때 사용
&	변수의 메모리 주솟값 반환
*	포인터 변수가 가리키는 메모리 주소 접근
...	슬라이스 요소들에 접근 또는 가변 인수 만들 때 사용
:	배열의 일부분을 집어올 때 사용
←	채널에서 값을 빼거나 넣을 때 사용

6.6 연산자 우선순위

- 우선순위가 높은 연산자, 같은 우선순위라면 좌측부터 연산된다.

연산자 우선순위

Aa 우선순위	≡ 연산자
5	$*/\%<<>>\&\&\wedge$
4	$+ - \wedge$
3	$== \neq < \leq > \geq$
2	$\&\&$
1	$ $

```
package main

import "fmt"

func main() {
    fmt.Println(3*4^7<<2+3*5 == 7)
}
```

출력값

```
false
```

1. $3*4 = 12$
2. $7<<2 = 28$
3. $3*5 = 15$
4. $12 \wedge 28 = 16$
5. $16 + 15 = 31$
6. $31 == 7 : \text{false}$

연습문제

1. $30 << 2 = 30 * 4 = 120$ (<127이므로 오류 발생하지 않음)
 $120 += 8 \rightarrow 128$ 이 되지만 int8의 범위를 벗어나 오버플로 발생
 답: -127
2. $a \mid= 2 \rightarrow a = 0000\ 0010$
 $a \mid= 4 \rightarrow a = 0000\ 0110$
 $a \mid= 8 \rightarrow a = 0000\ 1110$
 $a \wedge^= b \rightarrow a \& 1111\ 1011 = 0000\ 1010$
 답: 10
3. $x <<= 7 \rightarrow 1000\ 0000$

$x \gg 7 \rightarrow 1111\ 1111$ (오른쪽으로 밀 때, 최상위 비트값인 1로 채우게 된다)

답: -1