Go语言之语法糖讲解

语法糖（英语：Syntactic sugar）是由英国计算机科学家彼得·兰丁发明的一个术语，指计算机语言中添加的某种语法，这种语法对语言的功能没有影响，但是更方便程序员使用。

语法糖通常是用来简化代码编写的，特性就是使用语法糖前后编译的结果是相同的。

变量声明

变量

变量可以通过变量名访问。
Go 语言变量名由字母、数字、下划线组成，其中首个字符不能为数字。
声明变量的一般形式是使用 var 关键字：

```
var identifier type
```

可以一次声明多个变量：

```
var identifier1, identifier2 type
```

- 优先局部变量

局部变量和全局变量重名，优先访问局部变量。

```
package main

import "fmt"

// 定义全局变量
var num int16 = 10

func main() {
    var num int16 = 100 // 声明局部变量
    fmt.Printf("num: %v\n", num)
}
#结果
100
```

- 局部变量

if条件判读，for循环语句，switch语句，函数参数中定义的变量，也只能在相应的语句块中生效。

```
func test(x, y int) {
    fmt.Println(x, y) // 函数的参数只能在函数内生效

    for i := 0; i < 10; i++ {
        fmt.Printf("i: %v\n", i) // 变量i只能在for语句块中生效
    }
    // fmt.Printf("i: %v\n", i) // 此处无法使用变量i
```

```
    if x > 0 {
        z := 100
        fmt.Printf("z: %v\n", z) // 变量z只能在if语句块中生效
    }
    // fmt.Printf("z: %v\n", z) // 此处无法使用变量z
}
```

指定类型变量声明(指定类型)

指定变量类型，如果没有初始化，则变量默认为零值。
零值就是变量没有做初始化时系统默认设置的值。
数值类型（包括complex64/128）为 0
布尔类型为 false
字符串为 “”（空字符串）

以下几种类型为 nil：

```
var a *int
var a []int
var a map[string] int
var a chan int
var a func(string) int
var a error // error 是接口
```

示例：

```
// 形式一
var x int
x = 1
// 形式二
var x int = 1
// 形式三
var x = 1
```

无指定类型变量声明(无指定类型)

根据值自行判定变量类型。

```
var v_name = value
```

示例：

```
package main
import "fmt"
func main() {
    var d = true
    fmt.Println(d)//true
}
```

短变量声明

在 Go 函数中，我们可使用`name:= expression`的语法形式来声明和初始化局部变量。该语法糖的功能是声明（类型推断）和赋值。

```
x := 42
```

例如`x:=1`与下面几种形式是等价的

```
// 形式一
var x int
x = 1
// 形式二
var x int = 1
// 形式三
var x = 1
```

- 多变量声明中若其中一个变量是新的则可以使用两次

在多变量声明中，如果其中一个变量是新的，可以使用 := 两次。

```
x, y := 1, 2
y, z := 3, 4 // z 是新的变量
x, z := 5, 6 // 错误，x、z 均已定义过
```

- 函数内与短语句块中声明相同的名称

可以在短语句块中声明相同的名称，例如：if、for、switch 中，但它们有各自作用域。

```
func main() {
 x := 1
 if true {
  x := 2
  fmt.Printf("x = %d\n", x) // x = 2
 }
 fmt.Printf("x = %d\n", x) // x = 1
}
```

短变量声明注意事项

- 不能在函数外使用

不能在函数外使用 := ，因为在任何函数外，语句都应该以关键字开头，例如 type、var这样的关键字。

```
// 不合法
x := 42
// 合法
var y = 42

func main() {
 // 合法
 z:= 42
}
```

- 不能在同一作用域使用相同的语句两次

:= 代表引入一个新的变量，所以不能在同一作用域使用相同的 := 语句两次。

```
x := 1
x := 1 // 重复定义，错误
```

多变量声明

```
//类型相同多个变量, 非全局变量
var vname1, vname2, vname3 type
vname1, vname2, vname3 = v1, v2, v3

var vname1, vname2, vname3 = v1, v2, v3 // 和 python 很像,不需要显示声明类型，自动推断

vname1, vname2, vname3 := v1, v2, v3 // 出现在 := 左侧的变量不应该是已经被声明过的，否则会导致编译错误


// 这种因式分解关键字的写法一般用于声明全局变量
var (
    vname1 v_type1
    vname2 v_type2
)
```

示例:

```
package main

var x, y int
var (  // 这种因式分解关键字的写法一般用于声明全局变量
    a int
    b bool
    n,m ,z int
)

var c, d int = 1, 2
var e, f = 123, "hello"

//这种不带声明格式的只能在函数体中出现
//g, h := 123, "hello"

func main(){
```

```
    g, h := 123, "hello"
    println(x, y, a, b, c, d, e, f, g, h)
}
```

New函数

Go 内置的new函数是另一种创建变量的方式，表达式new(T)创建一个未命名的 T 类型变量，初始化为 T 类型的零值，并返回其地址（类型为 *T）。

下面两个`newInt`函数是等价的

```
func newInt() *int {
 return new(int)
}

func newInt() *int {
 var x int
 return &x
}
```

很明显，`new`函数的设计同样是为了方便程序员的使用。

“...”

在 Go 函数定义中，我们可以使用...表示可变参数，用于表示可以接受任意个数但相同类型的参数;
...T语法糖本质上代表的是一个切片，其元素类型为T。因此，...interface{}类型等价于[]interface{}，这也是为什么Println函数可以接受任意数量，任意类型的参数原因。

最经典的例子就是`fmt`包下的`Println`函数。

```
func Println(a ...interface{}) (n int, err error) {}
```

Println函数我们可以称之为可变参函数,可变参函数具有以下特征:

- 可变参必须定义在函数参数列表最后一个，也只能有一个可变参类型定义。
- 函数调用时，可变参可以不填，此时函数内部会将其当做 nil 切片处理。
- 可变参数必须是相同类型，如果需要不同类型就定义为 interface{}。

"..."可用于初始化数组

```
    //形式1
    a := []int{1, 2, 5}
    fmt.Println(a)                  //输出 :[1 2 5]
    fmt.Println(reflect.TypeOf(a)) //输出 : []int
    //形式2
    b := [6]int{1: 1, 2: 2, 5: 6}
    fmt.Println(b)                  //输出 : [0 1 2 0 0 6]
    fmt.Println(reflect.TypeOf(b)) //输出 : [6]int

    //经验证,"..."创建的是数组，并不是切片，等效于形式2
    x := [...]int{1: 1, 2: 2, 5: 6}
    fmt.Println(x)                  //输出 : [0 1 2 0 0 6]
    fmt.Println(reflect.TypeOf(x)) //输出 : [6]int
    fmt.Println(x[5])               //输出 : 6
```

```
    m := [...]int{1, 2, 6}
    fmt.Println(m)                  //输出 : [1 2 3]
    fmt.Println(reflect.TypeOf(m)) //输出 : [3]int
```

for range

可以使用 for range 来快速遍历可迭代对象，例如数组、切片、map、channel、字符串等。

- for range遍历切片/数组/字符串的三种方式

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

// 遍历一：不关心索引和数据的情况
for range a {
}

// 遍历二：只关心索引的情况
for index := range a {
 fmt.Println(index)
}

// 遍历三：关心索引和数据的情况
for index, value := range a {
 fmt.Println(index, value)
}
```

- for range遍历map的三种方式

```
m := map[int]string{1: "Golang", 2: "Python", 3: "Java"}
// 遍历一：不关心 key 和 value 的情况
for range m {
}

// 遍历二：只关心 key 的情况
for key := range m {
 fmt.Println(key)
}

// 遍历二：关心 key 和 value 的情况
for key, value := range m {
 fmt.Println(key, value)
}
```

- for range遍历channel的方式

```
ch := make(chan int, 10)

// 遍历一：不关心 channel 数据
for range ch {
}

// 遍历二：关心 channel 数据
for data := range ch {
 fmt.Println(data)
}
```

Go 编译器会将不同的 for range 遍历方式转换成不同的控制逻辑，简化使用逻辑，使得程序员能够更方便地对可迭代对象进行遍历处理。

接收者方法

在 Go 中，对于自定义类型 T，为它定义方法时，其接收者可以是类型 T 本身，也可能是 T 类型的指针 *T。

- 接收者为原类型时调用方法正常

```
type Instance struct{}

func (ins Instance) Foo() string {
    return "123"
}
func main() {
    var a = Instance{}.Foo()
    fmt.Println(a) //输出123

    var b Instance
    c := b.Foo()
    fmt.Println(c) //输出123
}
```

- 接收者为指针类型时直接调用方法失败

```
func (ins *Instance) Foo() string {
    return "123"
}
func main() {
    var a = Instance{}.Foo() //编译失败,cannot call pointer method Instance{}
}
```

- 接收者为指针类型时直接调用方法成功

a值属于 Instance 类型，而非 *Instance，却能调用 Foo 方法，这是为什么呢？这其实就是 Go 编译器提供的语法糖：
当一个变量可变时，我们对类型 T 的变量直接调用 *T 方法是合法的，因为 Go 编译器隐式地获取了它的地址。变量可变意味着变量可寻址，因此，上文提到的 Instance{}.Foo() 会得到编译错误，就在于 Instance{} 值不能寻址。

```
type Instance struct{}

func (ins *Instance) Foo() string {
    return "123"
}
func main() {
    var a Instance
    b := a.Foo()
    fmt.Println(b)//输出 : 123
}
```