1801210682 吴欣雨

Go语言语法

目录

1801210682 吴欣雨

一. Go 语言基础语法

1. Go 程序可以由多个标记组成，可以是关键字，标识符，常量，字符串，符号。
2. 在 Go 程序中，一行代表一个语句结束。每个语句不需要像 C 家族中的其它语言一样以分号 ; 结尾，因为这些工作都将由 Go 编译器自动完成。
3. Go 语言的字符串可以通过 `+` 实现：
4. 下面列举了 Go 代码中会使用到的 25 个关键字或保留字：

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

5. 除了以上介绍的这些关键字，Go 语言还有 36 个预定义标识符

append	bool	byte	cap	close	complex	complex64	complex128	uint16
copy	false	float32	float64	imag	int	int8	int16	uint32
int32	int64	iota	len	make	new	nil	panic	uint64
print	println	real	recover	string	true	uint	uint8	uintptr

二. 数据类型

1. 在 Go 编程语言中，数据类型用于声明函数和变量。数据类型的出现是为了把数据分成所需内存大小不同的数据，编程的时候需要用大数据的时候才需要申请大内存，就可以充分利用内存。Go 语言按类别有以下几种数据类型：JavaScript中的Number类型包含整数、浮点数、科学计数法、NaN以及Infinity

1801210682 吴欣雨

序号	类型和描述
1	**布尔型** 布尔型的值只可以是常量 true 或者 false。一个简单的例子：var b bool = true。
2	**数字类型** 整型 int 和浮点型 float32、float64，Go 语言支持整型和浮点型数字，并且支持复数，其中位的运算采用补码。
3	**字符串类型:** 字符串就是一串固定长度的字符连接起来的字符序列。Go 的字符串是由单个字节连接起来的。Go 语言的字符串的字节使用 UTF-8 编码标识 Unicode 文本。
4	**派生类型:** 包括： (a) 指针类型（Pointer） (b) 数组类型 (c) 结构化类型(struct) (d) Channel 类型 (e) 函数类型 (f) 切片类型 (g) 接口类型（interface） (h) Map 类型

2. Go 也有基于架构的类型，例如：int、uint 和 uintptr。

3. ==运算符会自动转换数据类型再做比较，===不会转换数据

序号	类型和描述
1	**uint8** 无符号 8 位整型 (0 到 255)
2	**uint16** 无符号 16 位整型 (0 到 65535)
3	**uint32** 无符号 32 位整型 (0 到 4294967295)
4	**uint64** 无符号 64 位整型 (0 到 18446744073709551615)
5	**int8** 有符号 8 位整型 (-128 到 127)
6	**int16** 有符号 16 位整型 (-32768 到 32767)
7	**int32** 有符号 32 位整型 (-2147483648 到 2147483647)
8	**int64** 有符号 64 位整型 (-9223372036854775808 到 9223372036854775807)

4. 函数如果使用参数，该变量可称为函数的形参。调用函数，可以通过两种方式来传递参数：

三. Go 语言函数

1. 函数是基本的代码块，用于执行一个任务。Go 语言最少有个 main() 函数。
2. 可以通过函数来划分不同功能，逻辑上每个函数执行的是指定的任务。
3. 函数声明告诉了编译器函数的名称，返回类型，和参数。
4. Go 语言标准库提供了多种可动用的内置的函数。例如，len() 函数可以接受不同类型参数并返回该类型的长度。如果我们传入的是字符串则返回字符串的长度，如果传入的是数组，则返回数组中包含的元素个数。for…in语句可以循环一个变量的所有属性

5. Go 语言函数定义格式如下：

```
func function_name( [parameter list] ) [return_types] {
   函数体
}
```

函数定义解析：

- func：函数由 func 开始声明
- function_name：函数名称，函数名和参数列表一起构成了函数签名。
- parameter list：参数列表，参数就像一个占位符，当函数被调用时，你可以将值传递给参数，这个值被称为实际参数。参数列表指定的是参数类型、顺序、及参数个数。参数是可选的，也就是说函数也可以不包含参数。
- return_types：返回类型，函数返回一列值。return_types 是该列值的数据类型。有些功能不需要返回值，这种情况下 return_types 不是必须的。
- 函数体：函数定义的代码集合。

6. 调用函数，可以通过两种方式来传递参数：

传递类型	描述
值传递	值传递是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。
引用传递	引用传递是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。

默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到实际参数。

四. 数据结构

1. Go 语言提供了数组类型的数据结构。数组是具有相同唯一类型的一组已编号且长度固定的数据项序列，这种类型可以是任意的原始类型例如整形、字符串或者自定义类型。

2. Go 语言数组声明需要指定元素类型及元素个数，语法格式如下：

```
var variable_name [SIZE] variable_type
```

3. 一个指针变量指向了一个值的内存地址。类似于变量和常量，在使用指针前你需要声明指针。指针声明格式如下：

```
var var_name *var-type
```

4. 当一个指针被定义后没有分配到任何变量时，它的值为 nil。nil 指针也称为空指针

5. nil在概念上和其它语言的null、None、nil、NULL一样，都指代零值或空值。一个指针变量通常缩写为 ptr。

6. Go 语言中数组可以存储同一类型的数据，但在结构体中我们可以为不同项定义不同的数据类型。

7. 结构体是由一系列具有相同类型或不同类型的数据构成的数据集合。结构体定义需要使用 type 和 struct 语句。struct 语句定义一个新的数据类型，结构体有中有一个或多个成员。type 语句设定了结构体的名称。结构体的格式如下：

```
type struct_variable_type struct {
   member definition;
   member definition;
   ...
   member definition;
}
```

8. 一旦定义了结构体类型，它就能用于变量的声明，语法格式如下：

```
variable_name := structure_variable_type {value1, value2...valuen}
或
variable_name := structure_variable_type { key1: value1, key2: value2..., keyn: valuen}
```

9. 如果要访问结构体成员，需要使用点号 `.` 操作符，格式为：

```
结构体.成员名"
```

10. 可以定义指向结构体的指针类似于其他指针变量，格式如下：

```
var struct_pointer *Books
```

11. Go 语言切片是对数组的抽象。Go 数组的长度不可改变，在特定场景中这样的集合就不太适用，Go中提供了一种灵活，功能强悍的内置类型切片(“动态数组"),与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。

12. 可以声明一个未指定大小的数组来定义切片：

```
var identifier []type
```

13. 切片是可索引的，并且可以由 len() 方法获取长度。切片提供了计算容量的方法 cap() 可以测量切片最长可以达到多少。

14. Go 语言中 range 关键字用于 for 循环中迭代数组(array)、切片(slice)、通道(channel)或集合(map)的元素。在数组和切片中它返回元素的索引和索引对应的值，在集合中返回 key-value 对的 key 值。

15. Map 是一种无序的键值对的集合。Map 最重要的一点是通过 key 来快速检索数据，key 类似于索引，指向数据的值。Map 是一种集合，所以我们可以像迭代数组和切片那样迭代它。不过，Map 是无序的，我们无法决定它的返回顺序，这是因为 Map 是使用 hash 表来实现的。

16. 可以使用内建函数 make 也可以使用 map 关键字来定义 Map:

```
/* 声明变量，默认 map 是 nil */
var map_variable map[key_data_type]value_data_type

/* 使用 make 函数 */
map_variable := make(map[key_data_type]value_data_type)
```

五. 错误处理

1. Go 语言通过内置的错误接口提供了非常简单的错误处理机制。error类型是一个接口类型，这是它的定义：

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

2. 我们可以在编码中通过实现 error 接口类型来生成错误信息。函数通常在最后的返回值中返回错误信息。使用errors.New 可返回一个错误信息：

```
func Sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, errors.New("math: square root of negative number")
    }
    // 实现
}
```

3. 在下面的例子中，我们在调用Sqrt的时候传递的一个负数，然后就得到了non-nil的error对象，将此对象与nil比较，结果为true，所以fmt.Println(fmt包在处理error时会调用Error方法)被调用，以输出错误，请看下面调用的示例代码：

```
result, err:= Sqrt(-1)

if err != nil {
   fmt.Println(err)
}
```

六. 并发

1. Go 语言支持并发，我们只需要通过 go 关键字来开启 goroutine 即可。goroutine 是轻量级线程，goroutine 的调度是由 Golang 运行时进行管理的。goroutine 类似于线程，但并非线程。可以将 goroutine 理解为一种虚拟线程。Go语言运行时会参与调度 goroutine，并将 goroutine 合理地分配到每个 CPU 中，最大限度地使用CPU性能。

2. goroutine 语法格式：

```
go 函数名( 参数列表 )
```

例如：

```
go f(x, y, z)
```

3. Go 允许使用 go 语句开启一个新的运行期线程， 即 goroutine，以一个不同的、新创建的 goroutine 来执行一个函数。 同一个程序中的所有 goroutine 共享同一个地址空间。

实例

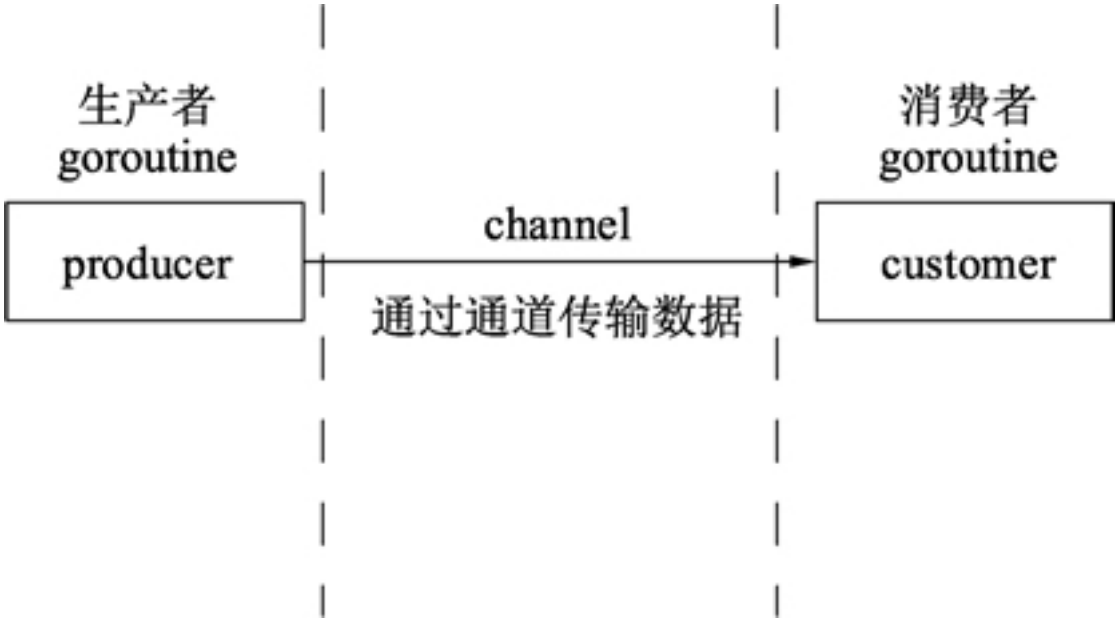
```
package main

import (
        "fmt"
        "time"
)

func say(s string) {
        for i := 0; i < 5; i++ {
                time.Sleep(100 * time.Millisecond)
                fmt.Println(s)
        }
}

func main() {
        go say("world")
        say("hello")
}
```

执行以上代码，你会看到输出的 hello 和 world 是没有固定先后顺序。因为它们是两个 goroutine 在执行：

```
world
hello
hello
world
world
hello
hello
world
world
hello
```

4. 通道（channel）是用来传递数据的一个数据结构。通道可用于两个 goroutine 之间通过传递一个指定类型的值来同步运行和通讯。操作符 <- 用于指定通道的方向，

发送或接收。如果未指定方向，则为双向通道。这让编程模型更倾向于在 goroutine 之间发送消息，而不是让多个 goroutine 争夺同一个数据的使用权。程序可以将需要并发的环节设计为生产者模式和消费者的模式，将数据放入通道。通道的另外一端的代码将这些数据进行并发计算并返回结果，如下图所示。




```
ch <- v    // 把 v 发送到通道 ch
v := <-ch  // 从 ch 接收数据
           // 并把值赋给 v
```

5. 声明一个通道很简单，我们使用chan关键字即可，通道在使用前必须先创建：

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

6. 默认情况下，通道是不带缓冲区的。发送端发送数据，同时必须又接收端相应的接收数据。以下实例通过两个 goroutine 来计算数字之和，在 goroutine 完成计算后，它会计算两个结果的和：

实例

```
package main

import "fmt"

func sum(s []int, c chan int) {
        sum := 0
        for _, v := range s {
                sum += v
        }
        c <- sum // 把 sum 发送到通道 c
}

func main() {
        s := []int{7, 2, 8, -9, 4, 0}

        c := make(chan int)
        go sum(s[:len(s)/2], c)
        go sum(s[len(s)/2:], c)
        x, y := <-c, <-c // 从通道 c 中接收

        fmt.Println(x, y, x+y)
}
```

```
-5 17 12
```

7. 通道可以设置缓冲区，通过 make 的第二个参数指定缓冲区大小：

```
ch := make(chan int, 100)
```

8. 带缓冲区的通道允许发送端的数据发送和接收端的数据获取处于异步状态，就是说发送端发送的数据可以放在缓冲区里面，可以等待接收端去获取数据，而不是立刻需要接收端去获取数据。不过由于缓冲区的大小是有限的，所以还是必须有接收端来接收数据的，否则缓冲区一满，数据发送端就无法再发送数据了。

9. 如果通道不带缓冲，发送方会阻塞直到接收方从通道中接收了值。如果通道带缓冲，发送方则会阻塞直到发送的值被拷贝到缓冲区内；如果缓冲区已满，则意味着需要等待直到某个接收方获取到一个值。接收方在有值可以接收之前会一直阻塞。

实例

```
package main

import "fmt"

func main() {
    // 这里我们定义了一个可以存储整数类型的带缓冲通道
        // 缓冲区大小为2
        ch := make(chan int, 2)

        // 因为 ch 是带缓冲的通道，我们可以同时发送两个数据
        // 而不用立刻需要去同步读取数据
        ch <- 1
        ch <- 2

        // 获取这两个数据
        fmt.Println(<-ch)
        fmt.Println(<-ch)
}
```

```
1
2
```

10. 如果通道接收不到数据后 ok 就为 false，这时通道就可以使用 `close()` 函数来关闭。

1801210682 吴欣雨

实例

```
package main

import (
        "fmt"
)

func fibonacci(n int, c chan int) {
        x, y := 0, 1
        for i := 0; i < n; i++ {
                c <- x
                x, y = y, x+y
        }
        close(c)
}

func main() {
        c := make(chan int, 10)
        go fibonacci(cap(c), c)
        // range 函数遍历每个从通道接收到的数据，因为 c 在发送完 10 个
        // 数据之后就关闭了通道，所以这里我们 range 函数在接收到 10 个数据
        // 之后就结束了。如果上面的 c 通道不关闭，那么 range 函数就不
        // 会结束，从而在接收第 11 个数据的时候就阻塞了。
        for i := range c {
                fmt.Println(i)
        }
}
```

```
0
1
1
2
3
5
8
13
21
34
```