

GO语言中的标识符与关键字

标识符

在编程语言中标识符就是程序员定义的具有特殊意义的词，比如变量名、常量名、函数名等等。 Go语言中标识符由字母数字和 `_` (下划线) 组成，并且只能以字母和 `_` 开头。举几个例子：`abc`, `_`, `_123`, `a123`。

关键字

关键字是指编程语言中预先定义好的具有特殊含义的标识符。关键字和保留字都不建议用作变量名。

Go语言中有25个关键字：

<code>break</code>	<code>default</code>	<code>func</code>	<code>interface</code>	<code>select</code>
<code>case</code>	<code>defer</code>	<code>go</code>	<code>map</code>	<code>struct</code>
<code>chan</code>	<code>else</code>	<code>goto</code>	<code>package</code>	<code>switch</code>
<code>const</code>	<code>fallthrough</code>	<code>if</code>	<code>range</code>	<code>type</code>
<code>continue</code>	<code>for</code>	<code>import</code>	<code>return</code>	<code>var</code>

此外，Go语言中还有37个保留字。

```
Constants:    true  false  iota  nil

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

Functions:    make  len   cap   new   append  copy   close  delete
              complex  real  imag
              panic  recover
```

GO语言中的变量

变量类型

变量（Variable）的功能是存储数据。不同的变量保存的数据类型可能会不一样。经过半个多世纪的发展，编程语言已经基本形成了一套固定的类型，常见变量的数据类型有：整型、浮点型、布尔型等。

Go语言中的每一个变量都有自己的类型，并且变量必须经过声明才能开始使用。

变量声明

Go语言中的变量需要声明后才能使用，同一作用域内不支持重复声明。并且Go语言的变量声明后必须使用。

标准声明

Go语言的变量声明格式为：

```
var 变量名 变量类型
```

变量声明以关键字 `var` 开头，变量类型放在变量的后面，行尾无需分号。举个例子：

```
var name string
var age int
var isOk bool
```

批量声明

每声明一个变量就需要写 `var` 关键字会比较繁琐，go语言中还支持批量变量声明：

```
var (
    a string
    b int
    c bool
    d float32
)
```

变量的初始化

Go语言在声明变量的时候，会自动对变量对应的内存区域进行初始化操作。每个变量会被初始化成其类型的默认值，例如：整型和浮点型变量的默认值为 `0`。字符串变量的默认值为 `空字符串`。布尔型变量默认为 `false`。切片、函数、指针变量的默认为 `nil`。

当然我们也可在声明变量的时候为其指定初始值。变量初始化的标准格式如下：

```
var 变量名 类型 = 表达式
```

举个例子：

```
var name string = "Q1mi"
var age int = 18
```

或者一次初始化多个变量

```
var name, age = "Q1mi", 20
```

类型推导

有时候我们会将变量的类型省略，这个时候编译器会根据等号右边的值来推导变量的类型完成初始化。

```
var name = "Q1mi"
var age = 18
```

短变量声明

在函数内部，可以使用更简略的 `:=` 方式声明并初始化变量。

```
package main

import (
    "fmt"
)
// 全局变量m
var m = 100

func main() {
    n := 10
    m := 200 // 此处声明局部变量m
    fmt.Println(m, n)
}
```

匿名变量

在使用多重赋值时，如果想要忽略某个值，可以使用 [匿名变量 \(anonymous variable\)](#)。匿名变量用一个下划线 `_` 表示，例如：

```
func foo() (int, string) {
    return 10, "Q1mi"
}
func main() {
    x, _ := foo()
    _, y := foo()
    fmt.Println("x=", x)
    fmt.Println("y=", y)
}
```

匿名变量不占用命名空间，不会分配内存，所以匿名变量之间不存在重复声明。（在 [Lua](#) 等编程语言里，匿名变量也被叫做哑元变量。）

注意事项：

1. 函数外的每个语句都必须以关键字开始（`var`、`const`、`func`等）
2. `:=` 不能使用在函数外。
3. `_` 多用于占位，表示忽略值。

GO语言中的常量

相对于变量，常量是恒定不变的值，多用于定义程序运行期间不会改变的那些值。常量的声明和变量声明非常类似，只是把 `var` 换成了 `const`，常量在定义的时候必须赋值。

```
const pi = 3.1415
const e = 2.7182
```

声明了 `pi` 和 `e` 这两个常量之后，在整个程序运行期间它们的值都不能再发生变化了。

多个常量也可以一起声明：

```
const (
    pi = 3.1415
    e = 2.7182
)
```

`const` 同时声明多个常量时，如果省略了值则表示和上面一行的值相同。例如：

```
const (
    n1 = 100
    n2
    n3
)
```

上面示例中，常量 `n1`、`n2`、`n3` 的值都是100。

iota常量计数器

`iota` 是go语言的常量计数器，只能在常量的表达式中使用。

`iota` 在`const`关键字出现时将被重置为0。`const`中每新增一行常量声明将使 `iota` 计数一次(`iota`可理解为`const`语句块中的行索引)。使用*iota*能简化定义，在定义枚举时很有用。

举个例子：

```
const (
    n1 = iota //0
    n2      //1
    n3      //2
    n4      //3
)
```

几个常见的 `iota` 示例：

使用 `_` 跳过某些值

```
const (
    n1 = iota //0
    n2        //1
    -
    n4        //3
)
```

`iota` 声明中间插队

```
const (
    n1 = iota //0
    n2 = 100 //100
    n3 = iota //2
    n4        //3
)
const n5 = iota //0
```

定义数量级（这里的 `<<` 表示左移操作，`1<<10` 表示将1的二进制表示向左移10位，也就是由 `1` 变成了 `100000000000`，也就是十进制的1024。同理 `2<<2` 表示将2的二进制表示向左移2位，也就是由 `10` 变成了 `1000`，也就是十进制的8。）

```
const (
    _ = iota
    KB = 1 << (10 * iota)
    MB = 1 << (10 * iota)
    GB = 1 << (10 * iota)
    TB = 1 << (10 * iota)
    PB = 1 << (10 * iota)
)
```

多个 `iota` 定义在一行

```
const (
    a, b = iota + 1, iota + 2 //1,2
    c, d           //2,3
    e, f           //3,4
)
```

GO语言的基本数据类型

整型

整型分为以下两个大类：按长度分为：`int8`、`int16`、`int32`、`int64` 对应的无符号整型：`uint8`、`uint16`、`uint32`、`uint64`

其中，`uint8` 就是我们熟知的 `byte` 型，`int16` 对应C语言中的 `short` 型，`int64` 对应C语言中的 `long` 型。

类型	描述
uint8	无符号 8位整型 (0 到 255)
uint16	无符号 16位整型 (0 到 65535)
uint32	无符号 32位整型 (0 到 4294967295)
uint64	无符号 64位整型 (0 到 18446744073709551615)
int8	有符号 8位整型 (-128 到 127)
int16	有符号 16位整型 (-32768 到 32767)
int32	有符号 32位整型 (-2147483648 到 2147483647)
int64	有符号 64位整型 (-9223372036854775808 到 9223372036854775807)

特殊整型

类型	描述
uint	32位操作系统上就是 <code>uint32</code> ，64位操作系统上就是 <code>uint64</code>
int	32位操作系统上就是 <code>int32</code> ，64位操作系统上就是 <code>int64</code>
uintptr	无符号整型，用于存放一个指针

注意：在使用 `int` 和 `uint` 类型时，不能假定它是32位或64位的整型，而是考虑 `int` 和 `uint` 可能在不同平台上的差异。

注意事项 获取对象的长度的内建 `len()` 函数返回的长度可以根据不同平台的字节长度进行变化。实际使用中，切片或 map 的元素数量等都可以用 `int` 来表示。在涉及到二进制传输、读写文件的结构描述时，为了保持文件的结构不会受到不同编译目标平台字节长度的影响，不要使用 `int` 和 `uint`。

数字字面量语法 (Number literals syntax)

Go1.13版本之后引入了数字字面量语法，这样便于开发者以二进制、八进制或十六进制浮点数的格式定义数字，例如：

`v := 0b00101101`，代表二进制的 101101，相当于十进制的 45。`v := 0o377`，代表八进制的 377，相当于十进制的 255。`v := 0x1p-2`，代表十六进制的 1 除以 2^2 ，也就是 0.25。

而且还允许我们用 `_` 来分隔数字，比如说：`v := 123_456` 表示 v 的值等于 123456。

我们可以借助fmt函数来将一个整数以不同进制形式展示。

```
package main

import "fmt"

func main(){
    // 十进制
    var a int = 10
```

```
fmt.Printf("%d \n", a) // 10
fmt.Printf("%b \n", a) // 1010 占位符%b表示二进制

// 八进制 以0开头
var b int = 077
fmt.Printf("%o \n", b) // 77

// 十六进制 以0x开头
var c int = 0xff
fmt.Printf("%x \n", c) // ff
fmt.Printf("%X \n", c) // FF
}
```

浮点型

Go语言支持两种浮点型数：`float32` 和 `float64`。这两种浮点型数据格式遵循 [IEEE 754](#) 标准：`float32` 的浮点数的最大范围约为 `3.4e38`，可以使用常量定义：`math.MaxFloat32`。`float64` 的浮点数的最大范围约为 `1.8e308`，可以使用一个常量定义：`math.MaxFloat64`。

打印浮点数时，可以使用 `fmt` 包配合动词 `%f`，代码如下：

```
package main
import (
    "fmt"
    "math"
)
func main() {
    fmt.Printf("%f\n", math.Pi)
    fmt.Printf("%.2f\n", math.Pi)
}
```

复数

`complex64` 和 `complex128`

```
var c1 complex64
c1 = 1 + 2i
var c2 complex128
c2 = 2 + 3i
fmt.Println(c1)
fmt.Println(c2)
```

复数有实部和虚部，`complex64` 的实部和虚部为32位，`complex128` 的实部和虚部为64位。

布尔值

Go语言中以 `bool` 类型进行声明布尔型数据，布尔型数据只有 `true (真)` 和 `false (假)` 两个值。

注意：

1. 布尔类型变量的默认值为 `false`。

2. Go 语言中不允许将整型强制转换为布尔型。

3. 布尔型无法参与数值运算，也无法与其他类型进行转换。

字符串

Go语言中的字符串以原生数据类型出现，使用字符串就像使用其他原生数据类型（int、bool、float32、float64等）一样。Go语言里的字符串的内部实现使用 [UTF-8](#) 编码。字符串的值为 双引号("") 中的内容，可以在Go语言的源码中直接添加非ASCII码字符，例如：

```
s1 := "hello"  
s2 := "你好"
```

字符串转义符

Go语言的字符串常见转义符包含回车、换行、单双引号、制表符等，如下表所示。

转义符	含义
\r	回车符（返回行首）
\n	换行符（直接跳到下一行的同列位置）
\t	制表符
\'	单引号
\"	双引号
\\\	反斜杠

举个例子，我们要打印一个Windows平台下的一个文件路径：

```
package main  
import (  
    "fmt"  
)  
func main() {  
    fmt.Println("str := \"c:\\Code\\lesson1\\go.exe\"")  
}
```

多行字符串

Go语言中要定义一个多行字符串时，就必须使用 [反引号](#) 字符：

```
s1 := `第一行  
第二行  
第三行  
'  
fmt.Println(s1)
```

反引号间换行将被作为字符串中的换行，但是所有的转义字符均无效，文本将会原样输出。

字符串的常用操作

方法	介绍
len(str)	求长度
+或fmt.Sprintf	拼接字符串
strings.Split	分割
strings.Contains	判断是否包含
strings.HasPrefix, strings.HasSuffix	前缀/后缀判断
strings.Index(), strings.LastIndex()	子串出现的位置
strings.Join(a[]string, sep string)	join操作

byte和rune类型

组成每个字符串的元素叫做“字符”，可以通过遍历或者单个获取字符串元素获得字符。字符用单引号（'）包裹起来，如：

```
var a = '中'  
var b = 'x'
```

Go语言的字符有以下两种：

1. `uint8` 类型，或者叫 `byte` 型，代表了 `ASCII码` 的一个字符。
2. `rune` 类型，代表一个 `UTF-8字符`。

当需要处理中文、日文或者其他复合字符时，则需要用到 `rune` 类型。`rune` 类型实际是一个 `int32`。

Go 使用了特殊的 `rune` 类型来处理 Unicode，让基于 Unicode 的文本处理更为方便，也可以使用 `byte` 型进行默认字符串处理，性能和扩展性都有照顾。

```
// 遍历字符串  
func traversalString() {  
    s := "hello沙河"  
    for i := 0; i < len(s); i++ { //byte  
        fmt.Printf("%v(%c) ", s[i], s[i])  
    }  
    fmt.Println()  
    for _, r := range s { //rune  
        fmt.Printf("%v(%c) ", r, r)  
    }  
    fmt.Println()  
}
```

输出：

```
104(h) 101(e) 108(l) 108(l) 111(o) 230(æ) 178(²) 153() 230(æ) 178(²) 179(³)  
104(h) 101(e) 108(l) 108(l) 111(o) 27801(沙) 27827(河)
```

因为UTF8编码下一个中文汉字由3~4个字节组成，所以我们不能简单的按照字节去遍历一个包含中文的字符串，否则就会出现上面输出中第一行的结果。

字符串底层是一个byte数组，所以可以和 `[]byte` 类型相互转换。字符串是不能修改的 字符串是由byte字节组成，所以字符串的长度是byte字节的长度。 `rune`类型用来表示utf8字符，一个rune字符由一个或多个byte组成。

修改字符串

要修改字符串，需要先将其转换成 `[]rune` 或 `[]byte`，完成后再转换为 `string`。无论哪种转换，都会重新分配内存，并复制字节数组。

```
func changeString() {
    s1 := "big"
    // 强制类型转换
    byteS1 := []byte(s1)
    byteS1[0] = 'p'
    fmt.Println(string(byteS1))

    s2 := "白萝卜"
    runeS2 := []rune(s2)
    runeS2[0] = '红'
    fmt.Println(string(runeS2))
}
```

类型转换

Go语言中只有强制类型转换，没有隐式类型转换。该语法只能在两个类型之间支持相互转换的时候使用。

强制类型转换的基本语法如下：

```
T(表达式)
```

其中，T表示要转换的类型。表达式包括变量、复杂算子和函数返回值等。

比如计算直角三角形的斜边长时使用math包的`Sqrt()`函数，该函数接收的是`float64`类型的参数，而变量a和b都是`int`类型的，这个时候就需要将a和b强制类型转换为`float64`类型。

```
func sqrtDemo() {
    var a, b = 3, 4
    var c int
    // math.Sqrt()接收的参数是float64类型，需要强制转换
    c = int(math.Sqrt(float64(a*a + b*b)))
    fmt.Println(c)
}
```

GO语言中的运算符

Go 语言内置的运算符有：

1. 算术运算符
2. 关系运算符
3. 逻辑运算符
4. 位运算符
5. 赋值运算符

算术运算符

运算符	描述
+	相加
-	相减
*	相乘
/	相除
%	求余

注意： `++` (自增) 和 `--` (自减) 在Go语言中是单独的语句，并不是运算符。

关系运算符

运算符	描述
<code>==</code>	检查两个值是否相等，如果相等返回 True 否则返回 False。
<code>!=</code>	检查两个值是否不相等，如果不相等返回 True 否则返回 False。
<code>></code>	检查左边值是否大于右边值，如果是返回 True 否则返回 False。
<code>>=</code>	检查左边值是否大于等于右边值，如果是返回 True 否则返回 False。
<code><</code>	检查左边值是否小于右边值，如果是返回 True 否则返回 False。
<code><=</code>	检查左边值是否小于等于右边值，如果是返回 True 否则返回 False。

逻辑运算符

运算符	描述
&&	逻辑 AND 运算符。如果两边的操作数都是 True，则为 True，否则为 False。
	逻辑 OR 运算符。如果两边的操作数有一个 True，则为 True，否则为 False。
!	逻辑 NOT 运算符。如果条件为 True，则为 False，否则为 True。

位运算符

位运算符对整数在内存中的二进制位进行操作。

运算符	描述
&	参与运算的两数各对应的二进位相与。（两位均为1才为1）
	参与运算的两数各对应的二进位相或。（两位有一个为1就为1）
^	参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为1。（两位不一样则为1）
<<	左移n位就是乘以2的n次方。“a<<b”是把a的各二进位全部左移b位，高位丢弃，低位补0。
>>	右移n位就是除以2的n次方。“a>>b”是把a的各二进位全部右移b位。

赋值运算符

运算符	描述
=	简单的赋值运算符，将一个表达式的值赋给一个左值
+=	相加后再赋值
-=	相减后再赋值
*=	相乘后再赋值
/=	相除后再赋值
%=	求余后再赋值
<<=	左移后赋值
>>=	右移后赋值
&=	按位与后赋值
=	按位或后赋值
^=	按位异或后赋值

流程控制是每种编程语言控制逻辑走向和执行次序的重要部分，流程控制可以说是一门语言的“经脉”。

Go语言中最常用的流程控制有 `if` 和 `for`，而 `switch` 和 `goto` 主要是为了简化代码、降低重复代码而生的结构，属于扩展类的流程控制。

GO语言中的逻辑控制

if else(分支结构)

if条件判断基本写法

Go语言中 `if` 条件判断的格式如下：

```
if 表达式1 {  
    分支1  
} else if 表达式2 {  
    分支2  
} else{  
    分支3  
}
```

当表达式1的结果为 `true` 时，执行分支1，否则判断表达式2，如果满足则执行分支2，都不满足时，则执行分支3。if判断中的 `else if` 和 `else` 都是可选的，可以根据实际需要进行选择。

Go语言规定与 `if` 匹配的左括号 `{` 必须与 `if和表达式` 放在同一行，`{` 放在其他位置会触发编译错误。同理，与 `else` 匹配的 `{` 也必须与 `else` 写在同一行，`else` 也必须与上一个 `if` 或 `else if` 右边的大括号在同一行。

举个例子：

```
func ifDemo1() {  
    score := 65  
    if score >= 90 {  
        fmt.Println("A")  
    } else if score > 75 {  
        fmt.Println("B")  
    } else {  
        fmt.Println("C")  
    }  
}
```

if条件判断特殊写法

if条件判断还有一种特殊的写法，可以在 `if` 表达式之前添加一个执行语句，再根据变量值进行判断，举个例子：

```
func ifDemo2() {  
    if score := 65; score >= 90 {  
        fmt.Println("A")  
    } else if score > 75 {  
        fmt.Println("B")  
    } else {  
        fmt.Println("C")  
    }  
}
```

思考题：上下两种写法的区别在哪里？

for(循环结构)

Go 语言中的所有循环类型均可以使用 `for` 关键字来完成。

for循环的基本格式如下：

```
for 初始语句; 条件表达式; 结束语句{  
    循环体语句  
}
```

条件表达式返回 `true` 时循环体不停地进行循环，直到条件表达式返回 `false` 时自动退出循环。

```
func forDemo() {  
    for i := 0; i < 10; i++ {  
        fmt.Println(i)  
    }  
}
```

for循环的初始语句可以被忽略，但是初始语句后的分号必须要写，例如：

```
func forDemo2() {  
    i := 0  
    for ; i < 10; i++ {  
        fmt.Println(i)  
    }  
}
```

for循环的初始语句和结束语句都可以省略，例如：

```
func forDemo3() {  
    i := 0  
    for i < 10 {  
        fmt.Println(i)  
        i++  
    }  
}
```

这种写法类似于其他编程语言中的 `while`，在 `while` 后添加一个条件表达式，满足条件表达式时持续循环，否则结束循环。

无限循环

```
for {  
    循环体语句  
}
```

for循环可以通过 `break`、`goto`、`return`、`panic` 语句强制退出循环。

for range(键值循环)

Go语言中可以使用 `for range` 遍历数组、切片、字符串、map 及通道（channel）。通过 `for range` 遍历的返回值有以下规律：

1. 数组、切片、字符串返回索引和值。
2. map返回键和值。
3. 通道（channel）只返回通道内的值。

switch case

使用 `switch` 语句可方便地对大量的值进行条件判断。

```
func switchDemo1() {  
    finger := 3  
    switch finger {  
        case 1:  
            fmt.Println("大拇指")  
        case 2:  
            fmt.Println("食指")  
        case 3:  
            fmt.Println("中指")  
        case 4:  
            fmt.Println("无名指")  
        case 5:  
            fmt.Println("小拇指")  
        default:  
            fmt.Println("无效的输入! ")  
    }  
}
```

Go语言规定每个 `switch` 只能有一个 `default` 分支。

一个分支可以有多个值，多个case值中间使用英文逗号分隔。

```
func testSwitch3() {  
    switch n := 7; n {  
        case 1, 3, 5, 7, 9:  
            fmt.Println("奇数")  
        case 2, 4, 6, 8:  
            fmt.Println("偶数")  
        default:  
            fmt.Println(n)  
    }  
}
```

分支还可以使用表达式，这时候switch语句后面不需要再跟判断变量。例如：

```
func switchDemo4() {
    age := 30
    switch {
        case age < 25:
            fmt.Println("好好学习吧")
        case age > 25 && age < 35:
            fmt.Println("好好工作吧")
        case age > 60:
            fmt.Println("好好享受吧")
        default:
            fmt.Println("活着真好")
    }
}
```

`fallthrough` 语法可以执行满足条件的case的下一个case，是为了兼容C语言中的case设计的。

```
func switchDemo5() {
    s := "a"
    switch {
        case s == "a":
            fmt.Println("a")
        fallthrough
        case s == "b":
            fmt.Println("b")
        case s == "c":
            fmt.Println("c")
        default:
            fmt.Println("...")
    }
}
```

输出：

```
a
b
```

goto(跳转到指定标签)

`goto` 语句通过标签进行代码间的无条件跳转。`goto` 语句可以在快速跳出循环、避免重复退出上有一定的帮助。

Go语言中使用 `goto` 语句能简化一些代码的实现过程。例如双层嵌套的for循环要退出时：

```
func gotoDemo1() {
    var breakFlag bool
    for i := 0; i < 10; i++ {
        for j := 0; j < 10; j++ {
            if j == 2 {
                // 设置退出标签
                breakFlag = true
                break
            }
            fmt.Printf("%v-%v\n", i, j)
        }
        // 外层for循环判断
        if breakFlag {
            break
        }
    }
}
```

```
    }
}
}
```

使用 `goto` 语句能简化代码：

```
func gotoDemo2() {
    for i := 0; i < 10; i++ {
        for j := 0; j < 10; j++ {
            if j == 2 {
                // 设置退出标签
                goto breakTag
            }
            fmt.Printf("%v-%v\n", i, j)
        }
    }
    return
    // 标签
breakTag:
    fmt.Println("结束for循环")
}
```

break(跳出循环)

`break` 语句可以结束 `for`、`switch` 和 `select` 的代码块。

`break` 语句还可以在语句后面添加标签，表示退出某个标签对应的代码块，标签要求必须定义在对应的 `for`、`switch` 和 `select` 的代码块上。举个例子：

```
func breakDemo1() {
BREAKDEMO1:
    for i := 0; i < 10; i++ {
        for j := 0; j < 10; j++ {
            if j == 2 {
                break BREAKDEMO1
            }
            fmt.Printf("%v-%v\n", i, j)
        }
    }
    fmt.Println("...")
}
```

continue(继续下次循环)

`continue` 语句可以结束当前循环，开始下一次的循环迭代过程，仅限在 `for` 循环内使用。

在 `continue` 语句后添加标签时，表示开始标签对应的循环。例如：

```
func continueDemo() {
forloop1:
    for i := 0; i < 5; i++ {
        // forloop2:
        for j := 0; j < 5; j++ {
            if i == 2 && j == 2 {
                continue forloop1
            }
            fmt.Printf("%v-%v\n", i, j)
        }
    }
}
```

GO语言中的Array(数组)

数组是同一种数据类型元素的集合。在Go语言中，数组从声明时就确定，使用时可以修改数组成员，但是数组大小不可变化。基本语法：

```
// 定义一个长度为3元素类型为int的数组a
var a [3]int
```

数组定义

```
var 数组变量名 [元素数量]T
```

比如：`var a [5]int`，数组的长度必须是常量，并且长度是数组类型的一部分。一旦定义，长度不能变。

`[5]int` 和 `[10]int` 是不同的类型。

```
var a [3]int
var b [4]int
a = b //不可以这样做，因为此时a和b是不同的类型
```

数组可以通过下标进行访问，下标是从`0`开始，最后一个元素下标是：`len-1`，访问越界（下标在合法范围之外），则触发访问越界，会panic。

数组的初始化

数组的初始化也有很多方式。

方法一

初始化数组时可以使用初始化列表来设置数组元素的值。

```
func main() {
    var testArray [3]int //数组会初始化为int类型的零值
    var numArray = [3]int{1, 2} //使用指定的初始值完成初始化
    var cityArray = [...]string{"北京", "上海", "深圳"} //使用指定的初始值完成初始化
    fmt.Println(testArray) // [0 0 0]
    fmt.Println(numArray) // [1 2 0]
    fmt.Println(cityArray) // [北京 上海 深圳]
}
```

方法二

按照上面的方法每次都要确保提供的初始值和数组长度一致，一般情况下我们可以让编译器根据初始值的个数自行推断数组的长度，例如：

```
func main() {
    var testArray [3]int
    var numArray = [...]int{1, 2}
    var cityArray = [...]string{"北京", "上海", "深圳"}
    fmt.Println(testArray) // [0 0 0]
    fmt.Println(numArray) // [1 2]
    fmt.Printf("type of numArray:%T\n", numArray) //type of numArray:[2]int
    fmt.Println(cityArray) // [北京 上海 深圳]
    fmt.Printf("type of cityArray:%T\n", cityArray) //type of cityArray:[3]string
}
```

方法三

我们还可以使用指定索引值的方式来初始化数组，例如：

```
func main() {
    a := [...]int{1: 1, 3: 5}
    fmt.Println(a) // [0 1 0 5]
    fmt.Printf("type of a:%T\n", a) //type of a:[4]int
}
```

数组的遍历

遍历数组a有以下两种方法：

```
func main() {
    var a = [...]string{"北京", "上海", "深圳"}
    // 方法1: for循环遍历
    for i := 0; i < len(a); i++ {
        fmt.Println(a[i])
    }

    // 方法2: for range遍历
    for index, value := range a {
        fmt.Println(index, value)
    }
}
```

多维数组

Go语言是支持多维数组的，我们这里以二维数组为例（数组中又嵌套数组）。

二维数组的定义

```
func main() {
    a := [3][2]string{
        {"北京", "上海"},
        {"广州", "深圳"},
        {"成都", "重庆"},
    }
    fmt.Println(a) //[[北京 上海] [广州 深圳] [成都 重庆]]
    fmt.Println(a[2][1]) //支持索引取值:重庆
}
```

二维数组的遍历

```
func main() {
    a := [3][2]string{
        {"北京", "上海"},
        {"广州", "深圳"},
        {"成都", "重庆"},
    }
    for _, v1 := range a {
        for _, v2 := range v1 {
            fmt.Printf("%s\t", v2)
        }
        fmt.Println()
    }
}
```

输出：

```
北京  上海
广州  深圳
成都  重庆
```

注意：多维数组只有第一层可以使用`...`来让编译器推导数组长度。例如：

```
//支持的写法
a := [...]string{
    {"北京", "上海"},
    {"广州", "深圳"},
    {"成都", "重庆"},
}

//不支持多维数组的内层使用...
b := [3][...]string{
    {"北京", "上海"},
    {"广州", "深圳"},
    {"成都", "重庆"},
}
```

数组是值类型

数组是值类型，赋值和传参会复制整个数组。因此改变副本的值，不会改变本身的值。

```
func modifyArray(x [3]int) {
    x[0] = 100
}

func modifyArray2(x [3][2]int) {
    x[2][0] = 100
}

func main() {
    a := [3]int{10, 20, 30}
    modifyArray(a) //在modify中修改的是a的副本x
    fmt.Println(a) // [10 20 30]
    b := [3][2]int{
        {1, 1},
        {1, 1},
        {1, 1},
    }
    modifyArray2(b) //在modify中修改的是b的副本x
    fmt.Println(b) // [[1 1] [1 1] [1 1]]
}
```

注意：

1. 数组支持“==”、“!=”操作符，因为内存总是被初始化过的。
2. `[n]*T` 表示指针数组，`*[n]T` 表示数组指针。

GO语言中的切片

切片（Slice）是一个拥有相同类型元素的可变长度的序列。它是基于数组类型做的一层封装。它非常灵活，支持自动扩容。

切片是一个引用类型，它的内部结构包含 地址、长度 和 容量。切片一般用于快速地操作一块数据集合。

引子

因为数组的长度是固定的并且数组长度属于类型的一部分，所以数组有很多的局限性。例如：

```
func arraySum(x [3]int) int{
    sum := 0
    for _, v := range x{
        sum = sum + v
    }
    return sum
}
```

这个求和函数只能接受 [3]int 类型，其他的都不支持。再比如，

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

数组a中已经有三个元素了，我们不能再继续往数组a中添加新元素了。

切片的定义

声明切片类型的基本语法如下：

```
var name []T
```

其中，

- name:表示变量名
- T:表示切片中的元素类型

举个例子：

```
func main() {
    // 声明切片类型
    var a []string           //声明一个字符串切片
    var b = []int{}           //声明一个整型切片并初始化
    var c = []bool{false, true} //声明一个布尔切片并初始化
    var d = []bool{false, true} //声明一个布尔切片并初始化
    fmt.Println(a)             //[]
    fmt.Println(b)             //[]
    fmt.Println(c)             // [false true]
    fmt.Println(a == nil)      //true
    fmt.Println(b == nil)      //false
    fmt.Println(c == nil)      //false
    // fmt.Println(c == d)     //切片是引用类型，不支持直接比较，只能和nil比较
}
```

切片的长度和容量

切片拥有自己的长度和容量，我们可以通过使用内置的 `len()` 函数求长度，使用内置的 `cap()` 函数求切片的容量。

切片表达式

切片表达式从字符串、数组、指向数组或切片的指针构造子字符串或切片。它有两种变体：一种指定low和high两个索引界限值的简单的形式，另一种是除了low和high索引界限值外还指定容量的完整的形式。

简单切片表达式

切片的底层就是一个数组，所以我们可以基于数组通过切片表达式得到切片。切片表达式中的 `low` 和 `high` 表示一个索引范围（左包含，右不包含），也就是下面代码中从数组a中选出 `1<=索引值<4` 的元素组成切片s，得到的切片 `长度=high-low`，容量等于得到的切片的底层数组的容量。

```
func main() {
    a := [5]int{1, 2, 3, 4, 5}
    s := a[1:3] // s := a[low:high]
    fmt.Printf("s:%v len(s):%v cap(s):%v\n", s, len(s), cap(s))
}
```

输出：

```
s:[2 3] len(s):2 cap(s):4
```

为了方便起见，可以省略切片表达式中的任何索引。省略了 `low` 则默认为0；省略了 `high` 则默认为切片操作数的长度：

```
a[2:] // 等同于 a[2:len(a)]
a[:3] // 等同于 a[0:3]
a[:] // 等同于 a[0:len(a)]
```

注意：

对于数组或字符串，如果 `0 <= low <= high <= len(a)`，则索引合法，否则就会索引越界（out of range）。

对切片再执行切片表达式时（切片再切片），`high` 的上限边界是切片的容量 `cap(a)`，而不是长度。常量索引必须是非负的，并且可以用int类型的值表示；对于数组或常量字符串，常量索引也必须在有效范围内。如果 `low` 和 `high` 两个指标都是常数，它们必须满足 `low <= high`。如果索引在运行时超出范围，就会发生运行时 `panic`。

```
func main() {
    a := [5]int{1, 2, 3, 4, 5}
    s := a[1:3] // s := a[low:high]
    fmt.Printf("s:%v len(s):%v cap(s):%v\n", s, len(s), cap(s))
    s2 := s[3:4] // 索引的上限是cap(s)而不是len(s)
    fmt.Printf("s2:%v len(s2):%v cap(s2):%v\n", s2, len(s2), cap(s2))
}
```

输出：

```
s:[2 3] len(s):2 cap(s):4
s2:[5] len(s2):1 cap(s2):1
```

完整切片表达式

对于数组，指向数组的指针，或切片a(注意不能是字符串)支持完整切片表达式：

```
a[low : high : max]
```

上面的代码会构造与简单切片表达式 `a[low: high]` 相同类型、相同长度和元素的切片。另外，它会将得到的结果切片的容量设置为 `max-low`。在完整切片表达式中只有第一个索引值（`low`）可以省略；它默认为0。

```
func main() {
    a := [5]int{1, 2, 3, 4, 5}
    t := a[1:3:5]
    fmt.Printf("t:%v len(t):%v cap(t):%v\n", t, len(t), cap(t))
}
```

输出结果：

```
t:[2 3] len(t):2 cap(t):4
```

完整切片表达式需要满足的条件是 `0 <= low <= high <= max <= cap(a)`，其他条件和简单切片表达式相同。

使用make()函数构造切片

我们上面都是基于数组来创建的切片，如果需要动态的创建一个切片，我们就需要使用内置的 `make()` 函数，格式如下：

```
make([]T, size, cap)
```

其中：

- T:切片的元素类型
- size:切片中元素的数量
- cap:切片的容量

举个例子：

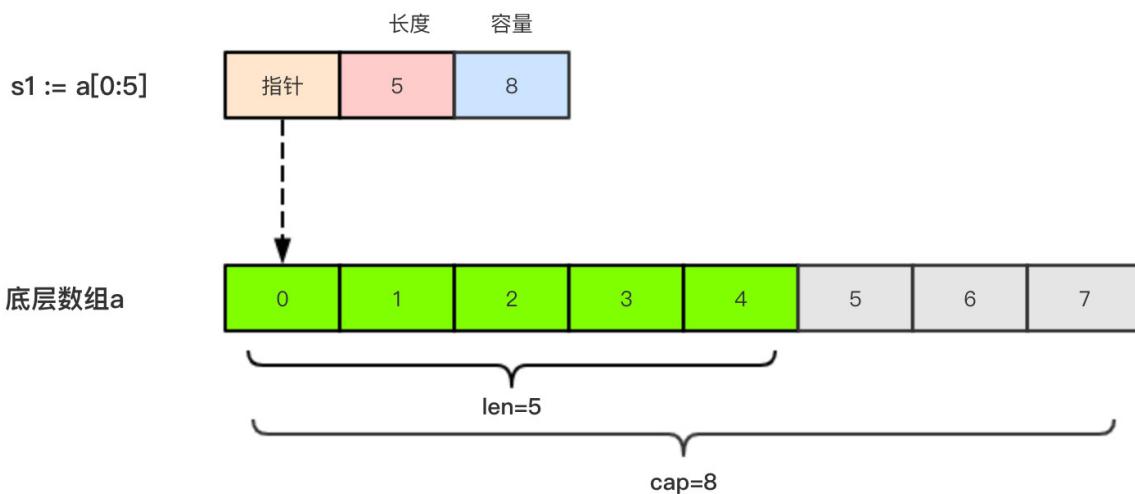
```
func main() {
    a := make([]int, 2, 10)
    fmt.Println(a)      // [0 0]
    fmt.Println(len(a)) // 2
    fmt.Println(cap(a)) // 10
}
```

上面代码中 `a` 的内部存储空间已经分配了10个，但实际上只用了2个。容量并不会影响当前元素的个数，所以 `len(a)` 返回2，`cap(a)` 则返回该切片的容量。

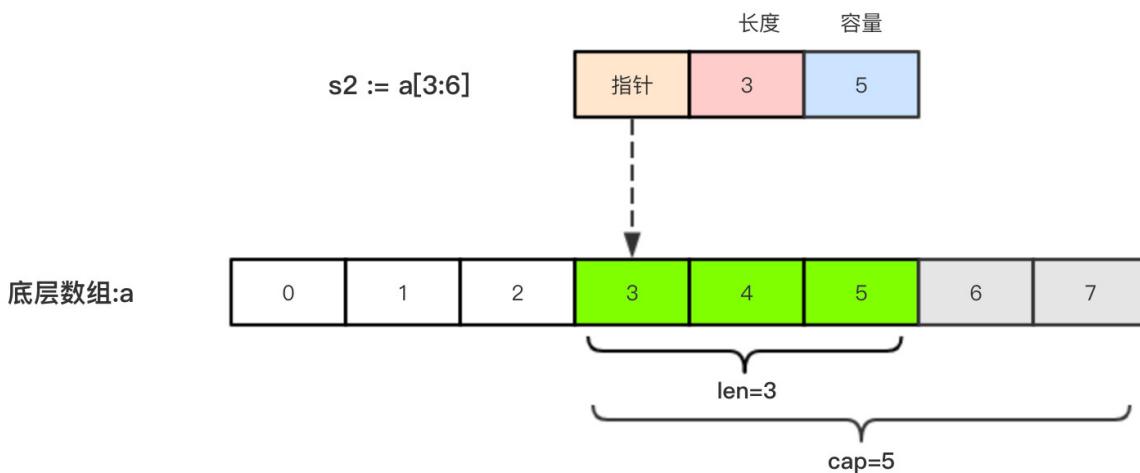
切片的本质

切片的本质就是对底层数组的封装，它包含了三个信息：底层数组的指针、切片的长度（len）和切片的容量（cap）。

举个例子，现在有一个数组 `a := [8]int{0, 1, 2, 3, 4, 5, 6, 7}`，切片 `s1 := a[:5]`，相应示意图如下。



切片 `s2 := a[3:6]`，相应示意图如下：



判断切片是否为空

要检查切片是否为空，请始终使用 `len(s) == 0` 来判断，而不应该使用 `s == nil` 来判断。

切片不能直接比较

切片之间是不能比较的，我们不能使用 `==` 操作符来判断两个切片是否含有全部相等元素。切片唯一合法的比较操作是和 `nil` 比较。一个 `nil` 值的切片并没有底层数组，一个 `nil` 值的切片的长度和容量都是0。但是我们不能说一个长度和容量都是0的切片一定是 `nil`，例如下面的示例：

```
var s1 []int          //len(s1)=0;cap(s1)=0;s1==nil
s2 := []int{}         //len(s2)=0;cap(s2)=0;s2!=nil
s3 := make([]int, 0)  //len(s3)=0;cap(s3)=0;s3!=nil
```

所以要判断一个切片是否是空的，要是用 `len(s) == 0` 来判断，不应该使用 `s == nil` 来判断。

切片的赋值拷贝

下面的代码中演示了拷贝前后两个变量共享底层数组，对一个切片的修改会影响另一个切片的内容，这点需要特别注意。

```
func main() {
    s1 := make([]int, 3) // [0 0 0]
    s2 := s1             // 将s1直接赋值给s2, s1和s2共用一个底层数组
    s2[0] = 100
    fmt.Println(s1) // [100 0 0]
    fmt.Println(s2) // [100 0 0]
}
```

切片遍历

切片的遍历方式和数组是一致的，支持索引遍历和 `for range` 遍历。

```
func main() {
    s := []int{1, 3, 5}

    for i := 0; i < len(s); i++ {
        fmt.Println(i, s[i])
    }

    for index, value := range s {
        fmt.Println(index, value)
    }
}
```

append()方法为切片添加元素

Go语言的内建函数 `append()` 可以为切片动态添加元素。可以一次添加一个元素，可以添加多个元素，也可以添加另一个切片中的元素（后面加...）。

```
func main(){
    var s []int
    s = append(s, 1)      // [1]
    s = append(s, 2, 3, 4) // [1 2 3 4]
    s2 := []int{5, 6, 7}
    s = append(s, s2...)  // [1 2 3 4 5 6 7]
}
```

注意：通过var声明的零值切片可以在 `append()` 函数直接使用，无需初始化。

```
var s []int
s = append(s, 1, 2, 3)
```

没有必要像下面的代码一样初始化一个切片再传入 `append()` 函数使用，

```
s := []int{} // 没有必要初始化
s = append(s, 1, 2, 3)

var s = make([]int) // 没有必要初始化
s = append(s, 1, 2, 3)
```

每个切片会指向一个底层数组，这个数组的容量够用就添加新增元素。当底层数组不能容纳新增的元素时，切片就会自动按照一定的策略进行“扩容”，此时该切片指向的底层数组就会更换。“扩容”操作往往发生在 `append()` 函数调用时，所以我们通常都需要用原变量接收 `append` 函数的返回值。

举个例子：

```
func main() {
    //append()添加元素和切片扩容
    var numSlice []int
    for i := 0; i < 10; i++ {
        numSlice = append(numSlice, i)
        fmt.Printf("%v len:%d cap:%d ptr:%p\n", numSlice, len(numSlice), cap(numSlice),
numSlice)
    }
}
```

输出：

```
[0] len:1 cap:1 ptr:0xc0000a8000
[0 1] len:2 cap:2 ptr:0xc0000a8040
[0 1 2] len:3 cap:4 ptr:0xc0000b2020
[0 1 2 3] len:4 cap:4 ptr:0xc0000b2020
[0 1 2 3 4] len:5 cap:8 ptr:0xc0000b6000
[0 1 2 3 4 5] len:6 cap:8 ptr:0xc0000b6000
[0 1 2 3 4 5 6] len:7 cap:8 ptr:0xc0000b6000
[0 1 2 3 4 5 6 7] len:8 cap:8 ptr:0xc0000b6000
[0 1 2 3 4 5 6 7 8] len:9 cap:16 ptr:0xc0000b8000
[0 1 2 3 4 5 6 7 8 9] len:10 cap:16 ptr:0xc0000b8000
```

从上面的结果可以看出：

1. `append()` 函数将元素追加到切片的最后并返回该切片。
2. 切片 `numSlice` 的容量按照 1, 2, 4, 8, 16 这样的规则自动进行扩容，每次扩容后都是扩容前的 2 倍。

`append()` 函数还支持一次性追加多个元素。例如：

```
var citySlice []string
// 追加一个元素
citySlice = append(citySlice, "北京")
// 追加多个元素
citySlice = append(citySlice, "上海", "广州", "深圳")
// 追加切片
a := []string{"成都", "重庆"}
citySlice = append(citySlice, a...)
fmt.Println(citySlice) // [北京 上海 广州 深圳 成都 重庆]
```

切片的扩容策略

可以通过查看 `$GOROOT/src/runtime/slice.go` 源码，其中扩容相关代码如下：

```
newcap := old.cap
doublecap := newcap + newcap
if cap > doublecap {
    newcap = cap
} else {
    if old.len < 1024 {
        newcap = doublecap
    } else {
        // Check 0 < newcap to detect overflow
        // and prevent an infinite loop.
        for 0 < newcap && newcap < cap {
            newcap += newcap / 4
        }
        // Set newcap to the requested cap when
        // the newcap calculation overflowed.
        if newcap <= 0 {
            newcap = cap
        }
    }
}
```

从上面的代码可以看出以下内容：

- 首先判断，如果新申请容量 (cap) 大于2倍的旧容量 (old.cap) ，最终容量 (newcap) 就是新申请的容量 (cap) 。
- 否则判断，如果旧切片的长度小于1024，则最终容量 (newcap) 就是旧容量 (old.cap) 的两倍，即 (`newcap=doublecap`) ，
- 否则判断，如果旧切片长度大于等于1024，则最终容量 (newcap) 从旧容量 (old.cap) 开始循环增加原来的1/4，即 (`newcap=old.cap,for {newcap += newcap/4}`) 直到最终容量 (newcap) 大于等于新申请的容量 (cap)，即 (`newcap >= cap`)
- 如果最终容量 (cap) 计算值溢出，则最终容量 (cap) 就是新申请容量 (cap) 。

需要注意的是，切片扩容还会根据切片中元素的类型不同而做不同的处理，比如 `int` 和 `string` 类型的处理方式就不一样。

使用`copy()`函数复制切片

首先我们来看一个问题：

```
func main() {
    a := []int{1, 2, 3, 4, 5}
    b := a
    fmt.Println(a) // [1 2 3 4 5]
    fmt.Println(b) // [1 2 3 4 5]
    b[0] = 1000
    fmt.Println(a) // [1000 2 3 4 5]
    fmt.Println(b) // [1000 2 3 4 5]
}
```

由于切片是引用类型，所以a和b其实都指向了同一块内存地址。修改b的同时a的值也会发生变化。

Go语言内建的 `copy()` 函数可以迅速地将一个切片的数据复制到另外一个切片空间中，`copy()` 函数的使用格式如下：

```
copy(destSlice, srcSlice []T)
```

其中：

- `srcSlice`: 数据来源切片
- `destSlice`: 目标切片

举个例子：

```
func main() {
    // copy() 复制切片
    a := []int{1, 2, 3, 4, 5}
    c := make([]int, 5, 5)
    copy(c, a) // 使用 copy() 函数将切片 a 中的元素复制到切片 c
    fmt.Println(a) // [1 2 3 4 5]
    fmt.Println(c) // [1 2 3 4 5]
    c[0] = 1000
    fmt.Println(a) // [1 2 3 4 5]
    fmt.Println(c) // [1000 2 3 4 5]
}
```

从切片中删除元素

Go语言中并没有删除切片元素的专用方法，我们可以使用切片本身的特性来删除元素。代码如下：

```
func main() {
    // 从切片中删除元素
    a := []int{30, 31, 32, 33, 34, 35, 36, 37}
    // 要删除索引为2的元素
    a = append(a[:2], a[3:]...)
    fmt.Println(a) // [30 31 33 34 35 36 37]
}
```

总结一下就是：要从切片 a 中删除索引为 `index` 的元素，操作方法是 `a = append(a[:index], a[index+1:]...)`

GO语言中的map

map是一种无序的基于 **key-value** 的数据结构，Go语言中的map是引用类型，必须初始化才能使用。

map定义

Go语言中 **map** 的定义语法如下：

```
map [KeyType]ValueType
```

其中，

- KeyType:表示键的类型。
- ValueType:表示键对应的值的类型。

map类型的变量默认初始值为nil，需要使用make()函数来分配内存。语法为：

```
make(map[KeyType]ValueType, [cap])
```

其中cap表示map的容量，该参数虽然不是必须的，但是我们应该在初始化map的时候就为其指定一个合适的容量。

map基本使用

map中的数据都是成对出现的，map的基本使用示例代码如下：

```
func main() {
    scoreMap := make(map[string]int, 8)
    scoreMap["张三"] = 90
    scoreMap["小明"] = 100
    fmt.Println(scoreMap)
    fmt.Println(scoreMap["小明"])
    fmt.Sprintf("type of a:%T\n", scoreMap)
}
```

输出：

```
map[小明:100 张三:90]
100
type of a:map[string]int
```

map也支持在声明的时候填充元素，例如：

```
func main() {
    userInfo := map[string]string{
        "username": "沙河小王子",
        "password": "123456",
    }
    fmt.Println(userInfo) //
```

判断某个键是否存在

Go语言中有个判断map中键是否存在特殊写法，格式如下：

```
value, ok := map[key]
```

举个例子：

```
func main() {
    scoreMap := make(map[string]int)
    scoreMap["张三"] = 90
    scoreMap["小明"] = 100
    // 如果key存在ok为true,v为对应的值；不存在ok为false,v为值类型的零值
    v, ok := scoreMap["张三"]
    if ok {
        fmt.Println(v)
    } else {
        fmt.Println("查无此人")
    }
}
```

map的3遍历

Go语言中使用 `for range` 遍历map。

```
func main() {
    scoreMap := make(map[string]int)
    scoreMap["张三"] = 90
    scoreMap["小明"] = 100
    scoreMap["娜扎"] = 60
    for k, v := range scoreMap {
        fmt.Println(k, v)
    }
}
```

但我们只想遍历key的时候，可以按下面的写法：

```
func main() {
    scoreMap := make(map[string]int)
    scoreMap["张三"] = 90
    scoreMap["小明"] = 100
    scoreMap["娜扎"] = 60
    for k := range scoreMap {
        fmt.Println(k)
    }
}
```

注意：遍历map时的元素顺序与添加键值对的顺序无关。

使用delete()函数删除键值对

使用 `delete()` 内建函数从map中删除一组键值对，`delete()` 函数的格式如下：

```
delete(map, key)
```

其中，

- map:表示要删除键值对的map
- key:表示要删除的键值对的键

示例代码如下：

```
func main(){
    scoreMap := make(map[string]int)
    scoreMap["张三"] = 90
    scoreMap["小明"] = 100
    scoreMap["娜扎"] = 60
    delete(scoreMap, "小明") //将小明:100从map中删除
    for k,v := range scoreMap{
        fmt.Println(k, v)
    }
}
```

按照指定顺序遍历map

```
func main() {
    rand.Seed(time.Now().UnixNano()) //初始化随机数种子

    var scoreMap = make(map[string]int, 200)

    for i := 0; i < 100; i++ {
        key := fmt.Sprintf("stu%02d", i) //生成stu开头的字符串
        value := rand.Intn(100)           //生成0~99的随机整数
        scoreMap[key] = value
    }
    //取出map中的所有key存入切片keys
    var keys = make([]string, 0, 200)
    for key := range scoreMap {
        keys = append(keys, key)
    }
    //对切片进行排序
    sort.Strings(keys)
    //按照排序后的key遍历map
    for _, key := range keys {
        fmt.Println(key, scoreMap[key])
    }
}
```

元素为map类型的切片

下面的代码演示了切片中的元素为map类型时的操作：

```
func main() {
    var mapSlice = make([]map[string]string, 3)
    for index, value := range mapSlice {
        fmt.Printf("index:%d value:%v\n", index, value)
    }
    fmt.Println("after init")
    // 对切片中的map元素进行初始化
    mapSlice[0] = make(map[string]string, 10)
    mapSlice[0]["name"] = "小王子"
    mapSlice[0]["password"] = "123456"
    mapSlice[0]["address"] = "沙河"
    for index, value := range mapSlice {
        fmt.Printf("index:%d value:%v\n", index, value)
    }
}
```

值为切片类型的map

下面的代码演示了map中值为切片类型的操作：

```
func main() {
    var sliceMap = make(map[string][]string, 3)
    fmt.Println(sliceMap)
    fmt.Println("after init")
    key := "中国"
    value, ok := sliceMap[key]
    if !ok {
        value = make([]string, 0, 2)
    }
    value = append(value, "北京", "上海")
    sliceMap[key] = value
    fmt.Println(sliceMap)
}
```

GO语言中的函数

Go语言中支持函数、匿名函数和闭包，并且函数在Go语言中属于“一等公民”。

函数定义

Go语言中定义函数使用 `func` 关键字，具体格式如下：

```
func 函数名(参数)(返回值){
    函数体
}
```

其中：

- 函数名：由字母、数字、下划线组成。但函数名的第一个字母不能是数字。在同一个包内，函数名也称不能重名（包的概念详见后文）。
- 参数：参数由参数变量和参数变量的类型组成，多个参数之间使用 `,` 分隔。
- 返回值：返回值由返回值变量和其变量类型组成，也可以只写返回值的类型，多个返回值必须用 `()` 包裹，并用 `,` 分隔。
- 函数体：实现指定功能的代码块。

先来定义一个求两个数之和的函数：

```
func intSum(x int, y int) int {  
    return x + y  
}
```

函数的参数和返回值都是可选的，例如我们可以实现一个既不需要参数也没有返回值的函数：

```
func sayHello() {  
    fmt.Println("Hello 沙河")  
}
```

函数的调用

定义了函数之后，我们可以通过 `函数名()` 的方式调用函数。例如我们调用上面定义的两个函数，代码如下：

```
func main() {  
    sayHello()  
    ret := intSum(10, 20)  
    fmt.Println(ret)  
}
```

注意，调用有返回值的函数时，可以不接收其返回值。

参数

类型简写

函数的参数中如果相邻变量的类型相同，则可以省略类型，例如：

```
func intSum(x, y int) int {  
    return x + y  
}
```

上面的代码中，`intSum` 函数有两个参数，这两个参数的类型均为 `int`，因此可以省略 `x` 的类型，因为 `y` 后面有类型说明，`x` 参数也是该类型。

可变参数

可变参数是指函数的参数数量不固定。Go语言中的可变参数通过在参数名后加`...`来标识。

注意：可变参数通常要作为函数的最后一个参数。

举个例子：

```
func intSum2(x ...int) int {
    fmt.Println(x) //x是一个切片
    sum := 0
    for _, v := range x {
        sum = sum + v
    }
    return sum
}
```

调用上面的函数：

```
ret1 := intSum2()
ret2 := intSum2(10)
ret3 := intSum2(10, 20)
ret4 := intSum2(10, 20, 30)
fmt.Println(ret1, ret2, ret3, ret4) //0 10 30 60
```

固定参数搭配可变参数使用时，可变参数要放在固定参数的后面，示例代码如下：

```
func intSum3(x int, y ...int) int {
    fmt.Println(x, y)
    sum := x
    for _, v := range y {
        sum = sum + v
    }
    return sum
}
```

调用上述函数：

```
ret5 := intSum3(100)
ret6 := intSum3(100, 10)
ret7 := intSum3(100, 10, 20)
ret8 := intSum3(100, 10, 20, 30)
fmt.Println(ret5, ret6, ret7, ret8) //100 110 130 160
```

本质上，函数的可变参数是通过切片来实现的。

返回值

Go语言中通过`return`关键字向外输出返回值。

多返回值

Go语言中函数支持多返回值，函数如果有多个返回值时必须用 `()` 将所有返回值包裹起来。

举个例子：

```
func calc(x, y int) (int, int) {
    sum := x + y
    sub := x - y
    return sum, sub
}
```

返回值命名

函数定义时可以给返回值命名，并在函数体中直接使用这些变量，最后通过 `return` 关键字返回。

例如：

```
func calc(x, y int) (sum, sub int) {
    sum = x + y
    sub = x - y
    return
}
```

返回值补充

当我们的一个函数返回值类型为slice时，`nil`可以看做是一个有效的slice，没必要显示返回一个长度为0的切片。

```
func someFunc(x string) []int {
    if x == "" {
        return nil // 没必要返回[]int{}
    }
    ...
}
```

函数进阶

变量作用域

全局变量

全局变量是定义在函数外部的变量，它在程序整个运行周期内都有效。在函数中可以访问到全局变量。

```
package main

import "fmt"

//定义全局变量num
var num int64 = 10

func testGlobalVar() {
    fmt.Printf("num=%d\n", num) //函数中可以访问全局变量num
}

func main() {
    testGlobalVar() //num=10
}
```

局部变量

局部变量又分为两种： 函数内定义的变量无法在该函数外使用，例如下面的示例代码main函数中无法使用testLocalVar函数中定义的变量x：

```
func testLocalVar() {
    //定义一个函数局部变量x,仅在该函数内生效
    var x int64 = 100
    fmt.Printf("x=%d\n", x)
}

func main() {
    testLocalVar()
    fmt.Println(x) // 此时无法使用变量x
}
```

如果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num
var num int64 = 10

func testNum() {
    num := 100
    fmt.Printf("num=%d\n", num) // 函数中优先使用局部变量
}

func main() {
    testNum() // num=100
}
```

接下来我们来看一下语句块定义的变量，通常我们会在if条件判断、for循环、switch语句上使用这种定义变量的方式。

```
func testLocalVar2(x, y int) {
    fmt.Println(x, y) //函数的参数也是只在本函数中生效
    if x > 0 {
        z := 100 //变量z只在if语句块生效
        fmt.Println(z)
    }
    //fmt.Println(z)//此处无法使用变量z
}
```

还有我们之前讲过的for循环语句中定义的变量，也是只在for语句块中生效：

```
func testLocalVar3() {
    for i := 0; i < 10; i++ {
        fmt.Println(i) //变量i只在当前for语句块中生效
    }
    //fmt.Println(i) //此处无法使用变量i
}
```

函数类型与变量

定义函数类型

我们可以使用 `type` 关键字来定义一个函数类型，具体格式如下：

```
type calculation func(int, int) int
```

上面语句定义了一个 `calculation` 类型，它是一种函数类型，这种函数接收两个int类型的参数并且返回一个int类型的返回值。

简单来说，凡是满足这个条件的函数都是`calculation`类型的函数，例如下面的`add`和`sub`是`calculation`类型。

```
func add(x, y int) int {
    return x + y
}

func sub(x, y int) int {
    return x - y
}
```

`add`和`sub`都能赋值给`calculation`类型的变量。

```
var c calculation
c = add
```

函数类型变量

我们可以声明函数类型的变量并且为该变量赋值：

```

func main() {
    var c calculation           // 声明一个calculation类型的变量c
    c = add                     // 把add赋值给c
    fmt.Printf("type of c:%T\n", c) // type of c:main.calculation
    fmt.Println(c(1, 2))         // 像调用add一样调用c

    f := add                   // 将函数add赋值给变量f1
    fmt.Printf("type of f:%T\n", f) // type of f:func(int, int) int
    fmt.Println(f(10, 20))       // 像调用add一样调用f
}

```

高阶函数

高阶函数分为函数作为参数和函数作为返回值两部分。

函数作为参数

函数可以作为参数：

```

func add(x, y int) int {
    return x + y
}

func calc(x, y int, op func(int, int) int) int {
    return op(x, y)
}

func main() {
    ret2 := calc(10, 20, add)
    fmt.Println(ret2) //30
}

```

函数作为返回值

函数也可以作为返回值：

```

func do(s string) (func(int, int) int, error) {
    switch s {
    case "+":
        return add, nil
    case "-":
        return sub, nil
    default:
        err := errors.New("无法识别的操作符")
        return nil, err
    }
}

```

匿名函数和闭包

匿名函数

函数当然还可以作为返回值，但是在Go语言中函数内部不能再像之前那样定义函数了，只能定义匿名函数。匿名函数就是没有函数名的函数，匿名函数的定义格式如下：

```
func(参数)(返回值){  
    函数体  
}
```

匿名函数因为没有函数名，所以没办法像普通函数那样调用，所以匿名函数需要保存到某个变量或者作为立即执行函数：

```
func main() {  
    // 将匿名函数保存到变量  
    add := func(x, y int) {  
        fmt.Println(x + y)  
    }  
    add(10, 20) // 通过变量调用匿名函数  
  
    // 自执行函数：匿名函数定义完加()直接执行  
    func(x, y int) {  
        fmt.Println(x + y)  
    }(10, 20)  
}
```

匿名函数多用于实现回调函数和闭包。

闭包

闭包指的是一个函数和与其相关的引用环境组合而成的实体。简单来说，[闭包=函数+引用环境](#)。首先我们来看一个例子：

```
func adder() func(int) int {  
    var x int  
    return func(y int) int {  
        x += y  
        return x  
    }  
}  
func main() {  
    var f = adder()  
    fmt.Println(f(10)) //10  
    fmt.Println(f(20)) //30  
    fmt.Println(f(30)) //60  
  
    f1 := adder()  
    fmt.Println(f1(40)) //40  
    fmt.Println(f1(50)) //90  
}
```

变量 `f` 是一个函数并且它引用了其外部作用域中的 `x` 变量，此时 `f` 就是一个闭包。在 `f` 的生命周期内，变量 `x` 也一直有效。闭包进阶示例1：

```

func adder2(x int) func(int) int {
    return func(y int) int {
        x += y
        return x
    }
}
func main() {
    var f = adder2(10)
    fmt.Println(f(10)) //20
    fmt.Println(f(20)) //40
    fmt.Println(f(30)) //70

    f1 := adder2(20)
    fmt.Println(f1(40)) //60
    fmt.Println(f1(50)) //110
}

```

闭包进阶示例2：

```

func makeSuffixFunc(suffix string) func(string) string {
    return func(name string) string {
        if !strings.HasSuffix(name, suffix) {
            return name + suffix
        }
        return name
    }
}

func main() {
    jpgFunc := makeSuffixFunc(".jpg")
    txtFunc := makeSuffixFunc(".txt")
    fmt.Println(jpgFunc("test")) //test.jpg
    fmt.Println(txtFunc("test")) //test.txt
}

```

闭包进阶示例3：

```

func calc(base int) (func(int) int, func(int) int) {
    add := func(i int) int {
        base += i
        return base
    }

    sub := func(i int) int {
        base -= i
        return base
    }
    return add, sub
}

func main() {
    f1, f2 := calc(10)
    fmt.Println(f1(1), f2(2)) //11 9
    fmt.Println(f1(3), f2(4)) //12 8
    fmt.Println(f1(5), f2(6)) //13 7
}

```

闭包其实并不复杂，只要牢记 闭包=函数+引用环境。

defer语句

Go语言中的 `defer` 语句会将其后面跟随的语句进行延迟处理。在 `defer` 归属的函数即将返回时，将延迟处理的语句按 `defer` 定义的逆序进行执行，也就是说，先被 `defer` 的语句最后被执行，最后被 `defer` 的语句，最先被执行。

举个例子：

```
func main() {
    fmt.Println("start")
    defer fmt.Println(1)
    defer fmt.Println(2)
    defer fmt.Println(3)
    fmt.Println("end")
}
```

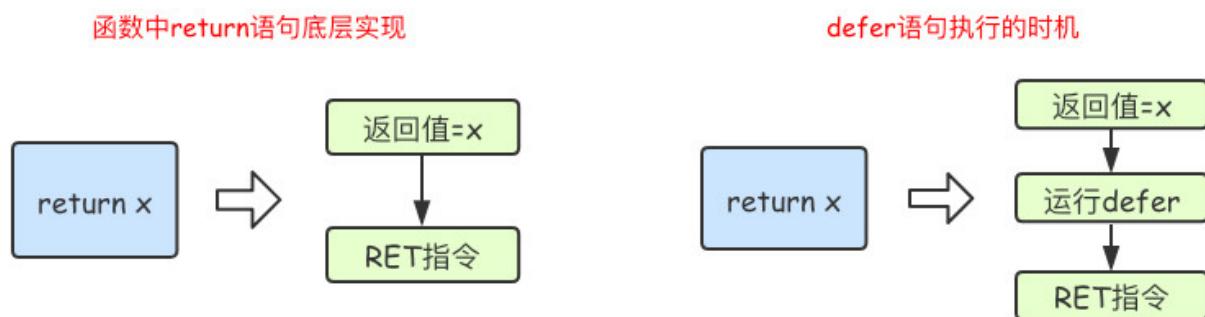
输出结果：

```
start
end
3
2
1
```

由于 `defer` 语句延迟调用的特性，所以 `defer` 语句能非常方便的处理资源释放问题。比如：资源清理、文件关闭、解锁及记录时间等。

defer执行时机

在Go语言的函数中 `return` 语句在底层并不是原子操作，它分为给返回值赋值和RET指令两步。而 `defer` 语句执行的时机就在返回值赋值操作后，RET指令执行前。具体如下图所示：



defer经典案例

阅读下面的代码，写出最后的打印结果。

```
func f1() int {
    x := 5
    defer func() {
        x++
    }()
    return x
}

func f2() (x int) {
    defer func() {
        x++
    }()
    return 5
}

func f3() (y int) {
    x := 5
    defer func() {
        x++
    }()
    return x
}

func f4() (x int) {
    defer func(x int) {
        x++
    }(x)
    return 5
}

func main() {
    fmt.Println(f1())
    fmt.Println(f2())
    fmt.Println(f3())
    fmt.Println(f4())
}
```

defer面试题

```
func calc(index string, a, b int) int {
    ret := a + b
    fmt.Println(index, a, b, ret)
    return ret
}

func main() {
    x := 1
    y := 2
    defer calc("AA", x, calc("A", x, y))
    x = 10
    defer calc("BB", x, calc("B", x, y))
    y = 20
}
```

问，上面代码的输出结果是？（提示：defer注册要延迟执行的函数时该函数所有的参数都需要确定其值）

内置函数介绍

内置函数	介绍
close	主要用来关闭channel
len	用来求长度，比如string、array、slice、map、channel
new	用来分配内存，主要用来分配值类型，比如int、struct。返回的是指针
make	用来分配内存，主要用来分配引用类型，比如chan、map、slice
append	用来追加元素到数组、slice中
panic和recover	用来做错误处理

panic/recover

Go语言中目前（Go1.12）是没有异常机制，但是使用 `panic/recover` 模式来处理错误。`panic` 可以在任何地方引发，但 `recover` 只有在 `defer` 调用的函数中有效。首先来看一个例子：

```
func funcA() {
    fmt.Println("func A")
}

func funcB() {
    panic("panic in B")
}

func funcC() {
    fmt.Println("func C")
}

func main() {
    funcA()
    funcB()
    funcC()
}
```

输出：

```
func A
panic: panic in B

goroutine 1 [running]:
main.funcB(...)
    .../code/func/main.go:12
main.main()
    .../code/func/main.go:20 +0x98
```

程序运行期间 `funcB` 中引发了 `panic` 导致程序崩溃，异常退出了。这个时候我们就可以通过 `recover` 将程序恢复回来，继续往后执行。

```
func funcA() {
    fmt.Println("func A")
}

func funcB() {
    defer func() {
        err := recover()
        //如果程序出现了panic错误,可以通过recover恢复过来
        if err != nil {
            fmt.Println("recover in B")
        }
    }()
    panic("panic in B")
}

func funcC() {
    fmt.Println("func C")
}
func main() {
    funcA()
    funcB()
    funcC()
}
```

注意：

1. `recover()` 必须搭配 `defer` 使用。
 2. `defer` 一定要在可能引发 `panic` 的语句之前定义。
-

GO语言中的指针

区别于C/C++中的指针，Go语言中的指针**不能进行偏移和运算，是安全指针**。

要搞明白Go语言中的指针需要先知道3个概念：指针地址、指针类型和指针取值。

任何程序数据载入内存后，在内存都有他们的地址，这就是指针。而为了保存一个数据在内存中的地址，我们就需要指针变量。

比如，“永远不要高估自己”这句话是我的座右铭，我想把它写入程序中，程序一启动这句话是要加载到内存（假设内存地址0x123456），我在程序中把这段话赋值给变量 `A`，把内存地址赋值给变量 `B`。这时候变量 `B` 就是一个指针变量。通过变量 `A` 和变量 `B` 都能找到我的座右铭。

Go语言中的指针不能进行偏移和运算，因此Go语言中的指针操作非常简单，我们只需要记住两个符号：`&`（取地址）和 `*`（根据地址取值）。

指针地址和指针类型

每个变量在运行时都拥有一个地址，这个地址代表变量在内存中的位置。Go语言中使用 `&` 字符放在变量前面对变量进行“取地址”操作。Go语言中的值类型（int、float、bool、string、array、struct）都有对应的指针类型，如：`*int`、`*int64`、`*string` 等。

取变量指针的语法如下：

```
ptr := &v // v的类型为T
```

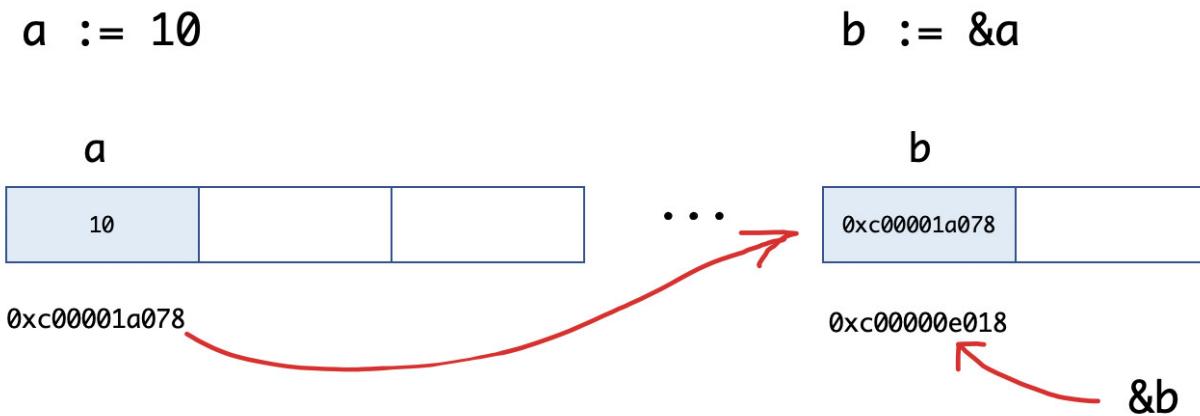
其中：

- `v`: 代表被取地址的变量，类型为 `T`
- `ptr`: 用于接收地址的变量，`ptr` 的类型就为 `*T`，称做 `T` 的指针类型。`*` 代表指针。

举个例子：

```
func main() {
    a := 10
    b := &a
    fmt.Printf("a:%d ptr:%p\n", a, &a) // a:10 ptr:0xc00001a078
    fmt.Printf("b:%p type:%T\n", b, b) // b:0xc00001a078 type:*int
    fmt.Println(&b)                  // 0xc00000e018
}
```

我们来看一下 `b := &a` 的图示：



指针取值

在对普通变量使用 `&` 操作符取地址后会获得这个变量的指针，然后可以对指针使用 `*` 操作，也就是指针取值，代码如下。

```
func main() {
    //指针取值
    a := 10
    b := &a // 取变量a的地址，将指针保存到b中
    fmt.Printf("type of b:%T\n", b)
    c := *b // 指针取值（根据指针去内存取值）
    fmt.Printf("type of c:%T\n", c)
    fmt.Printf("value of c:%v\n", c)
}
```

输出如下：

```
type of b:*int
type of c:int
value of c:10
```

总结：取地址操作符 `&` 和取值操作符 `*` 是一对互补操作符，`&` 取出地址，`*` 根据地址取出地址指向的值。

变量、指针地址、指针变量、取地址、取值的相互关系和特性如下：

- 对变量进行取地址（`&`）操作，可以获得这个变量的指针变量。
- 指针变量的值是指针地址。
- 对指针变量进行取值（`*`）操作，可以获得指针变量指向的原变量的值。

指针传值示例：

```
func modify1(x int) {
    x = 100
}

func modify2(x *int) {
    *x = 100
}

func main() {
    a := 10
    modify1(a)
    fmt.Println(a) // 10
    modify2(&a)
    fmt.Println(a) // 100
}
```

new和make

我们先来看一个例子：

```
func main() {
    var a *int
    *a = 100
    fmt.Println(*a)

    var b map[string]int
    b["沙河娜扎"] = 100
    fmt.Println(b)
}
```

执行上面的代码会引发panic，为什么呢？在Go语言中对于引用类型的变量，我们在使用的时候不仅要声明它，还要为它分配内存空间，否则我们的值就没办法存储。而对于值类型的声明不需要分配内存空间，是因为它们在声明的时候已经默认分配好了内存空间。要分配内存，就引出来今天的new和make。Go语言中new和make是内建的两个函数，主要用来分配内存。

new

new是一个内置的函数，它的函数签名如下：

```
func new(Type) *Type
```

其中，

- Type表示类型，new函数只接受一个参数，这个参数是一个类型
- *Type表示类型指针，new函数返回一个指向该类型内存地址的指针。

new函数不太常用，使用new函数得到的是一个类型的指针，并且该指针对应的值为该类型的零值。举个例子：

```
func main() {
    a := new(int)
    b := new(bool)
    fmt.Printf("%T\n", a) // *int
    fmt.Printf("%T\n", b) // *bool
    fmt.Println(*a)      // 0
    fmt.Println(*b)      // false
}
```

本节开始的示例代码中 `var a *int` 只是声明了一个指针变量a但是没有初始化，指针作为引用类型需要初始化后才会拥有内存空间，才可以给它赋值。应该按照如下方式使用内置的new函数对a进行初始化之后就可以正常对其赋值了：

```
func main() {
    var a *int
    a = new(int)
    *a = 10
    fmt.Println(*a)
}
```

make

make也是用于内存分配的，区别于new，它只用于slice、map以及chan的内存创建，而且它返回的类型就是这三个类型本身，而不是他们的指针类型，因为这三种类型就是引用类型，所以就没有必要返回他们的指针了。make函数的函数签名如下：

```
func make(t Type, size ...IntegerType) Type
```

make函数是无可替代的，我们在使用slice、map以及channel的时候，都需要使用make进行初始化，然后才可以对它们进行操作。这个我们在上一章中都有说明，关于channel我们会在后续的章节详细说明。

本节开始的示例中 `var b map[string]int` 只是声明变量b是一个map类型的变量，需要像下面的示例代码一样使用make函数进行初始化操作之后，才能对其进行键值对赋值：

```
func main() {
    var b map[string]int
    b = make(map[string]int, 10)
    b["沙河娜扎"] = 100
    fmt.Println(b)
}
```

new与make的区别

1. 二者都是用来做内存分配的。
2. make只用于slice、map以及channel的初始化，返回的还是这三个引用类型本身；
3. 而new用于类型的内存分配，并且内存对应的值为类型零值，返回的是指向类型的指针。

Go语言中没有“类”的概念，也不支持“类”的继承等面向对象的概念。Go语言中通过结构体的内嵌再配合接口比面向对象具有更高的扩展性和灵活性。

GO语言中的类型别名和自定义类型

自定义类型

在Go语言中有一些基本的数据类型，如 `string`、`整型`、`浮点型`、`布尔` 等数据类型，Go语言中可以使用 `type` 关键字来定义自定义类型。

自定义类型是定义了一个全新的类型。我们可以基于内置的基本类型定义，也可以通过struct定义。例如：

```
//将MyInt定义为int类型
type MyInt int
```

通过 `type` 关键字的定义，`MyInt` 就是一种新的类型，它具有 `int` 的特性。

类型别名

类型别名是 Go1.9 版本添加的新功能。

类型别名规定：TypeAlias只是Type的别名，本质上TypeAlias与Type是同一个类型。就像一个孩子小时候有小名、乳名，上学后用学名，英语老师又会给他起英文名，但这些名字都指的是他本人。

```
type TypeAlias = Type
```

我们之前见过的 `rune` 和 `byte` 就是类型别名，他们的定义如下：

```
type byte = uint8
type rune = int32
```

类型定义和类型别名的区别

类型别名与类型定义表面上看只有一个等号的差异，我们通过下面的这段代码来理解它们之间的区别。

```
//类型定义
type NewInt int

//类型别名
type MyInt = int

func main() {
    var a NewInt
    var b MyInt

    fmt.Printf("type of a:%T\n", a) //type of a:main.NewInt
    fmt.Printf("type of b:%T\n", b) //type of b:int
}
```

结果显示a的类型是 `main.NewInt`，表示main包下定义的 `NewInt` 类型。b的类型是 `int`。`MyInt` 类型只会在代码中存在，编译完成时并不会有 `MyInt` 类型。

GO语言中的结构体

Go语言中的基础数据类型可以表示一些事物的基本属性，但是当我们想表达一个事物的全部或部分属性时，这时候再用单一的基本数据类型明显就无法满足需求了，Go语言提供了一种自定义数据类型，可以封装多个基本数据类型，这种数据类型叫结构体，英文名称 `struct`。也就是我们可以通过 `struct` 来定义自己的类型了。

Go语言中通过 `struct` 来实现面向对象。

结构体的定义

使用 `type` 和 `struct` 关键字来定义结构体，具体代码格式如下：

```
type 类型名 struct {  
    字段名 字段类型  
    字段名 字段类型  
    ...  
}
```

其中：

- 类型名：标识自定义结构体的名称，在同一个包内不能重复。
- 字段名：表示结构体字段名。结构体中的字段名必须唯一。
- 字段类型：表示结构体字段的具体类型。

举个例子，我们定义一个 `Person`（人）结构体，代码如下：

```
type person struct {  
    name string  
    city string  
    age int8  
}
```

同样类型的字段也可以写在一行，

```
type person1 struct {  
    name, city string  
    age         int8  
}
```

这样我们就拥有了一个 `person` 的自定义类型，它有 `name`、`city`、`age` 三个字段，分别表示姓名、城市和年龄。这样我们使用这个 `person` 结构体就能够很方便的在程序中表示和存储人信息了。

语言内置的基础数据类型是用来描述一个值的，而结构体是用来描述一组值的。比如一个人有名字、年龄和居住城市等，本质上是一种聚合型的数据类型

结构体实例化

只有当结构体实例化时，才会真正地分配内存。也就是必须实例化后才能使用结构体的字段。

结构体本身也是一种类型，我们可以像声明内置类型一样使用 `var` 关键字声明结构体类型。

```
var 结构体实例 结构体类型
```

基本实例化

举个例子：

```
type person struct {
    name string
    city string
    age int8
}

func main() {
    var p1 person
    p1.name = "沙河娜扎"
    p1.city = "北京"
    p1.age = 18
    fmt.Printf("p1=%v\n", p1) //p1={沙河娜扎 北京 18}
    fmt.Printf("%#v\n", p1) //p1=main.person{name:"沙河娜扎", city:"北京", age:18}
}
```

我们通过`.`来访问结构体的字段（成员变量）,例如`p1.name`和`p1.age`等。

匿名结构体

在定义一些临时数据结构等场景下还可以使用匿名结构体。

```
package main

import (
    "fmt"
)

func main() {
    var user struct{Name string; Age int}
    user.Name = "小王子"
    user.Age = 18
    fmt.Printf("%#v\n", user)
}
```

创建指针类型结构体

我们还可以通过使用`new`关键字对结构体进行实例化，得到的是结构体的地址。格式如下：

```
var p2 = new(person)
fmt.Printf("%T\n", p2)      /*main.person
fmt.Printf("%#v\n", p2) //p2=&main.person{name:"", city:"", age:0}
```

从打印的结果中我们可以看出`p2`是一个结构体指针。

需要注意的是在Go语言中支持对结构体指针直接使用`.`来访问结构体的成员。

```
var p2 = new(person)
p2.name = "小王子"
p2.age = 28
p2.city = "上海"
fmt.Printf("p2=%#v\n", p2) //p2=&main.person{name:"小王子", city:"上海", age:28}
```

取结构体的地址实例化

使用 `&` 对结构体进行取地址操作相当于对该结构体类型进行了一次 `new` 实例化操作。

```
p3 := &person{}
fmt.Printf("%T\n", p3)      /*main.person
fmt.Printf("p3=%#v\n", p3) //p3=&main.person{name:"", city:"", age:0}
p3.name = "七米"
p3.age = 30
p3.city = "成都"
fmt.Printf("p3=%#v\n", p3) //p3=&main.person{name:"七米", city:"成都", age:30}
```

`p3.name = "七米"` 其实在底层是 `(*p3).name = "七米"`，这是Go语言帮我们实现的语法糖。

结构体初始化

没有初始化的结构体，其成员变量都是对应其类型的零值。

```
type person struct {
    name string
    city string
    age int8
}

func main() {
    var p4 person
    fmt.Printf("p4=%#v\n", p4) //p4=main.person{name:"", city:"", age:0}
}
```

使用键值对初始化

使用键值对对结构体进行初始化时，键对应结构体的字段，值对该字段的初始值。

```
p5 := person{
    name: "小王子",
    city: "北京",
    age: 18,
}
fmt.Printf("p5=%#v\n", p5) //p5=main.person{name:"小王子", city:"北京", age:18}
```

也可以对结构体指针进行键值对初始化，例如：

```
p6 := &person{  
    name: "小王子",  
    city: "北京",  
    age: 18,  
}  
fmt.Printf("p6=%#v\n", p6) //p6=&main.person{name:"小王子", city:"北京", age:18}
```

当某些字段没有初始值的时候，该字段可以不写。此时，没有指定初始值的字段的值就是该字段类型的零值。

```
p7 := &person{  
    city: "北京",  
}  
fmt.Printf("p7=%#v\n", p7) //p7=&main.person{name:"", city:"北京", age:0}
```

使用值的列表初始化

初始化结构体的时候可以简写，也就是初始化的时候不写键，直接写值：

```
p8 := &person{  
    "沙河娜扎",  
    "北京",  
    28,  
}  
fmt.Printf("p8=%#v\n", p8) //p8=&main.person{name:"沙河娜扎", city:"北京", age:28}
```

使用这种格式初始化时，需要注意：

1. 必须初始化结构体的所有字段。
2. 初始值的填充顺序必须与字段在结构体中的声明顺序一致。
3. 该方式不能和键值初始化方式混用。

结构体内存布局

结构体占用一块连续的内存。

```
type test struct {  
    a int8  
    b int8  
    c int8  
    d int8  
}  
n := test{  
    1, 2, 3, 4,  
}  
fmt.Printf("n.a %p\n", &n.a)  
fmt.Printf("n.b %p\n", &n.b)  
fmt.Printf("n.c %p\n", &n.c)  
fmt.Printf("n.d %p\n", &n.d)
```

输出：

```
n.a 0xc0000a0060
n.b 0xc0000a0061
n.c 0xc0000a0062
n.d 0xc0000a0063
```

【进阶知识点】关于Go语言中的内存对齐推荐阅读:[在 Go 中恰到好处的内存对齐](#)

空结构体

空结构体是不占用空间的。

```
var v struct{}
fmt.Println(unsafe.Sizeof(v)) // 0
```

面试题

请问下面代码的执行结果是什么？

```
type student struct {
    name string
    age  int
}

func main() {
    m := make(map[string]*student)
    stus := []student{
        {name: "小王子", age: 18},
        {name: "娜扎", age: 23},
        {name: "大王八", age: 9000},
    }

    for _, stu := range stus {
        m[stu.name] = &stu
    }
    for k, v := range m {
        fmt.Println(k, "=>", v.name)
    }
}
```

构造函数

Go语言的结构体没有构造函数，我们可以自己实现。例如，下方的代码就实现了一个 `person` 的构造函数。因为 `struct` 是值类型，如果结构体比较复杂的话，值拷贝性能开销会比较大，所以该构造函数返回的是结构体指针类型。

```
func newPerson(name, city string, age int8) *person {
    return &person{
        name: name,
        city: city,
        age:  age,
    }
}
```

调用构造函数

```
p9 := newPerson("张三", "沙河", 90)
fmt.Printf("%#v\n", p9) //&main.person{name:"张三", city:"沙河", age:90}
```

方法和接收者

Go语言中的 **方法 (Method)** 是一种作用于特定类型变量的函数。这种特定类型变量叫做 **接收者 (Receiver)**。

接收者的概念就类似于其他语言中的 `this` 或者 `self`。

方法的定义格式如下：

```
func (接收者变量 接收者类型) 方法名(参数列表) (返回参数) {
    函数体
}
```

其中，

- 接收者变量：接收者中的参数变量名在命名时，官方建议使用接收者类型名称首字母的小写，而不是 `self`、`this` 之类的命名。例如，`Person` 类型的接收者变量应该命名为 `p`，`Connector` 类型的接收者变量应该命名为 `c` 等。
- 接收者类型：接收者类型和参数类似，可以是指针类型和非指针类型。
- 方法名、参数列表、返回参数：具体格式与函数定义相同。

举个例子：

```
//Person 结构体
type Person struct {
    name string
    age  int8
}

//NewPerson 构造函数
func NewPerson(name string, age int8) *Person {
    return &Person{
        name: name,
        age:  age,
    }
}

//Dream Person做梦的方法
func (p Person) Dream() {
    fmt.Printf("%s的梦想是学好Go语言! \n", p.name)
}

func main() {
    p1 := NewPerson("小王子", 25)
    p1.Dream()
}
```

方法与函数的区别是，函数不属于任何类型，方法属于特定的类型。

指针类型的接收者

指针类型的接收者由一个结构体的指针组成，由于指针的特性，调用方法时修改接收者指针的任意成员变量，在方法结束后，修改都是有效的。这种方式就十分接近于其他语言中面向对象中的 `this` 或者 `self`。例如我们为 `Person` 添加一个 `SetAge` 方法，来修改实例变量的年龄。

```
// SetAge 设置p的年龄
// 使用指针接收者
func (p *Person) SetAge(newAge int8) {
    p.age = newAge
}
```

调用该方法：

```
func main() {
    p1 := NewPerson("小王子", 25)
    fmt.Println(p1.age) // 25
    p1.SetAge(30)
    fmt.Println(p1.age) // 30
}
```

值类型的接收者

当方法作用于值类型接收者时，Go语言会在代码运行时将接收者的值复制一份。在值类型接收者的方法中可以获取接收者的成员值，但修改操作只是针对副本，无法修改接收者变量本身。

```
// SetAge2 设置p的年龄
// 使用值接收者
func (p Person) SetAge2(newAge int8) {
    p.age = newAge
}

func main() {
    p1 := NewPerson("小王子", 25)
    p1.Dream()
    fmt.Println(p1.age) // 25
    p1.SetAge2(30) // (*p1).SetAge2(30)
    fmt.Println(p1.age) // 25
}
```

什么时候应该使用指针类型接收者

1. 需要修改接收者中的值
2. 接收者是拷贝代价比较大的大对象
3. 保证一致性，如果有某个方法使用了指针接收者，那么其他的方法也应该使用指针接收者。

任意类型添加方法

在Go语言中，接收者的类型可以是任何类型，不仅仅是结构体，任何类型都可以拥有方法。举个例子，我们基于内置的 `int` 类型使用 `type` 关键字可以定义新的自定义类型，然后为我们的自定义类型添加方法。

```
//MyInt 将int定义为自定义MyInt类型
type MyInt int

//SayHello 为MyInt添加一个SayHello的方法
func (m MyInt) SayHello() {
    fmt.Println("Hello, 我是一个int。")
}

func main() {
    var m1 MyInt
    m1.SayHello() //Hello, 我是一个int。
    m1 = 100
    fmt.Printf("%#v %T\n", m1, m1) //100 main.MyInt
}
```

注意事项：非本地类型不能定义方法，也就是说我们不能给别的包的类型定义方法。

结构体的匿名字段

结构体允许其成员字段在声明时没有字段名而只有类型，这种没有名字的字段就称为匿名字段。

```
//Person 结构体Person类型
type Person struct {
    string
    int
}

func main() {
    p1 := Person{
        "小王子",
        18,
    }
    fmt.Printf("%#v\n", p1)          //main.Person{string:"北京", int:18}
    fmt.Println(p1.string, p1.int) //北京 18
}
```

注意：这里匿名字段的说法并不代表没有字段名，而是默认会采用类型名作为字段名，结构体要求字段名称必须唯一，因此一个结构体中同种类型的匿名字段只能有一个。

嵌套结构体

一个结构体中可以嵌套包含另一个结构体或结构体指针，就像下面的示例代码那样。

```
//Address 地址结构体
type Address struct {
    Province string
    City     string
}
```

```
//User 用户结构体
type User struct {
    Name      string
    Gender    string
    Address   Address
}

func main() {
    user1 := User{
        Name:    "小王子",
        Gender:  "男",
        Address: Address{
            Province: "山东",
            City:     "威海",
        },
    }
    fmt.Printf("user1=%#v\n", user1) //user1=main.User{Name:"小王子", Gender:"男",
                                    Address:main.Address{Province:"山东", City:"威海"}}
}
```

嵌套匿名字段

上面user结构体中嵌套的 `Address` 结构体也可以采用匿名字段的方式，例如：

```
//Address 地址结构体
type Address struct {
    Province string
    City      string
}

//User 用户结构体
type User struct {
    Name      string
    Gender    string
    Address  //匿名字段
}

func main() {
    var user2 User
    user2.Name = "小王子"
    user2.Gender = "男"
    user2.Address.Province = "山东"      // 匿名字段默认使用类型名作为字段名
    user2.City = "威海"                  // 匿名字段可以省略
    fmt.Printf("user2=%#v\n", user2) //user2=main.User{Name:"小王子", Gender:"男",
                                    Address:main.Address{Province:"山东", City:"威海"}}
}
```

当访问结构体成员时会先在结构体中查找该字段，找不到再去嵌套的匿名字段中查找。

嵌套结构体的字段名冲突

嵌套结构体内部可能存在相同的字段名。在这种情况下为了避免歧义需要通过指定具体的内嵌结构体字段名。

```
//Address 地址结构体
type Address struct {
```

```

Province  string
City       string
CreateTime string
}

//Email 邮箱结构体
type Email struct {
    Account  string
    CreateTime string
}

//User 用户结构体
type User struct {
    Name   string
    Gender string
    Address
    Email
}

func main() {
    var user3 User
    user3.Name = "沙河娜扎"
    user3.Gender = "男"
    // user3.CreateTime = "2019" //ambiguous selector user3.CreateTime
    user3.Address.CreateTime = "2000" //指定Address结构体中的CreateTime
    user3.Email.CreateTime = "2000"   //指定Email结构体中的CreateTime
}

```

结构体的“继承”

Go语言中使用结构体也可以实现其他编程语言中面向对象的继承。

```

//Animal 动物
type Animal struct {
    name string
}

func (a *Animal) move() {
    fmt.Printf("%s会动! \n", a.name)
}

//Dog 狗
type Dog struct {
    Feet     int8
    *Animal //通过嵌套匿名结构体实现继承
}

func (d *Dog) wang() {
    fmt.Printf("%s会汪汪汪~\n", d.name)
}

func main() {
    d1 := &Dog{
        Feet: 4,
        Animal: &Animal{ //注意嵌套的是结构体指针
            name: "乐乐",
        },
}

```

```
}

d1.wang() //乐乐会汪汪~  
d1.move() //乐乐会动!  
}
```

结构体字段的可见性

结构体中字段大写开头表示可公开访问，小写表示私有（仅在定义当前结构体的包中可访问）。

结构体与JSON序列化

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。易于人阅读和编写。同时也易于机器解析和生成。JSON键值对是用来保存JS对象的一种方式，键/值对组合中的键名写在前面并用双引号 “” 包裹，使用冒号 : 分隔，然后紧接着值；多个键值之间使用英文 , 分隔。

```
//Student 学生  
type Student struct {  
    ID      int  
    Gender string  
    Name    string  
}  
  
//Class 班级  
type Class struct {  
    Title   string  
    Students []*Student  
}  
  
func main() {  
    c := &Class{  
        Title:    "101",  
        Students: make([]*Student, 0, 200),  
    }  
    for i := 0; i < 10; i++ {  
        stu := &Student{  
            Name:   fmt.Sprintf("stu%02d", i),  
            Gender: "男",  
            ID:     i,  
        }  
        c.Students = append(c.Students, stu)  
    }  
    //JSON序列化：结构体-->JSON格式的字符串  
    data, err := json.Marshal(c)  
    if err != nil {  
        fmt.Println("json marshal failed")  
        return  
    }  
    fmt.Printf("json:%s\n", data)  
    //JSON反序列化： JSON格式的字符串-->结构体  
    str := `{"Title":"101","Students":[{"ID":0,"Gender":"男","Name":"stu00"},  
{"ID":1,"Gender":"男","Name":"stu01"}, {"ID":2,"Gender":"男","Name":"stu02"},  
 {"ID":3,"Gender":"男","Name":"stu03"}, {"ID":4,"Gender":"男","Name":"stu04"},  
 {"ID":5,"Gender":"男","Name":"stu05"}, {"ID":6,"Gender":"男","Name":"stu06"},  
 {"ID":7,"Gender":"男","Name":"stu07"}, {"ID":8,"Gender":"男","Name":"stu08"},  
 {"ID":9,"Gender":"男","Name":"stu09"}]`
```

```

c1 := &Class{}
err = json.Unmarshal([]byte(str), c1)
if err != nil {
    fmt.Println("json unmarshal failed!")
    return
}
fmt.Printf("%#v\n", c1)
}

```

结构体标签 (Tag)

`Tag` 是结构体的元信息，可以在运行的时候通过反射的机制读取出来。`Tag` 在结构体字段的后方定义，由一对反引号包裹起来，具体的格式如下：

```
`key1:"value1" key2:"value2"`
```

结构体tag由一个或多个键值对组成。键与值使用冒号分隔，值用双引号括起来。同一个结构体字段可以设置多个键值对tag，不同的键值对之间使用空格分隔。

注意事项：为结构体编写 `Tag` 时，必须严格遵守键值对的规则。结构体标签的解析代码的容错能力很差，一旦格式写错，编译和运行时都不会提示任何错误，通过反射也无法正确取值。例如不要在key和value之间添加空格。

例如我们为 `Student` 结构体的每个字段定义json序列化时使用的Tag：

```

//Student 学生
type Student struct {
    ID      int      `json:"id"` //通过指定tag实现json序列化该字段时的key
    Gender string   //json序列化是默认使用字段名作为key
    name    string   //私有不能被json包访问
}

func main() {
    s1 := Student{
        ID:      1,
        Gender: "男",
        name:   "沙河娜扎",
    }
    data, err := json.Marshal(s1)
    if err != nil {
        fmt.Println("json marshal failed!")
        return
    }
    fmt.Printf("json str:%s\n", data) //json str:{"id":1,"Gender":"男"}
}

```

结构体和方法补充知识点

因为slice和map这两种数据类型都包含了指向底层数据的指针，因此我们在需要复制它们时要特别注意。我们来看下面的例子：

```

type Person struct {
    name  string
    age   int8
}

```

```
dreams []string
}

func (p *Person) SetDreams(dreams []string) {
    p.dreams = dreams
}

func main() {
    p1 := Person{name: "小王子", age: 18}
    data := []string{"吃饭", "睡觉", "打豆豆"}
    p1.SetDreams(data)

    // 你真的想要修改 p1.dreams 吗?
    data[1] = "不睡觉"
    fmt.Println(p1.dreams) // ?
}
```

正确的做法是在方法中使用传入的slice的拷贝进行结构体赋值。

```
func (p *Person) SetDreams(dreams []string) {
    p.dreams = make([]string, len(dreams))
    copy(p.dreams, dreams)
}
```

同样的问题也存在于返回值slice和map的情况，在实际编码过程中一定要注意这个问题。

Go语言的包 (package)

包介绍

包 (package) 是多个Go源码的集合，是一种高级的代码复用方案，Go语言为我们提供了很多内置包，如 `fmt`、`os`、`io` 等。

定义包

我们还可以根据自己的需要创建自己的包。一个包可以简单理解为一个存放 `.go` 文件的文件夹。该文件夹下面的所有go文件都要在代码的第一行添加如下代码，声明该文件归属的包。

```
package 包名
```

注意事项：

- 一个文件夹下面直接包含的文件只能归属一个 `package`，同样一个 `package` 的文件不能在多个文件夹下。
- 包名可以不和文件夹的名字一样，包名不能包含 `-` 符号。

- 包名为 `main` 的包为应用程序的入口包，这种包编译后会得到一个可执行文件，而编译不包含 `main` 包的源代码则不会得到可执行文件。

可见性

如果想在一个包中引用另外一个包里的标识符（如变量、常量、类型、函数等）时，该标识符必须是对外可见的（public）。在Go语言中只需要将标识符的首字母大写就可以让标识符对外可见了。

举个例子，我们定义一个包名为 `pkg2` 的包，代码如下：

```
package pkg2

import "fmt"

// 包变量可见性

var a = 100 // 首字母小写，外部包不可见，只能在当前包内使用

// 首字母大写外部包可见，可在其他包中使用
const Mode = 1

type person struct { // 首字母小写，外部包不可见，只能在当前包内使用
    name string
}

// 首字母大写，外部包可见，可在其他包中使用
func Add(x, y int) int {
    return x + y
}

func age() { // 首字母小写，外部包不可见，只能在当前包内使用
    var Age = 18 // 函数局部变量，外部包不可见，只能在当前函数内使用
    fmt.Println(Age)
}
```

结构体中的字段名和接口中的方法名如果首字母都是大写，外部包可以访问这些字段和方法。例如：

```
type Student struct {
    Name string //可在包外访问的方法
    class string //仅限包内访问的字段
}

type Payer interface {
    init() //仅限包内访问的方法
    Pay() //可在包外访问的方法
}
```

包的导入

要在代码中引用其他包的内容，需要使用 `import` 关键字导入使用的包。具体语法如下：

```
import "包的路径"
```

注意事项：

- import导入语句通常放在文件开头包声明语句的下面。
- 导入的包名需要使用双引号包裹起来。
- 包名是从`$GOPATH/src/`后开始计算的，使用`/`进行路径分隔。
- Go语言中禁止循环导入包。

单行导入

单行导入的格式如下：

```
import "包1"  
import "包2"
```

多行导入

多行导入的格式如下：

```
import (  
    "包1"  
    "包2"  
)
```

自定义包名

在导入包名的时候，我们还可以为导入的包设置别名。通常用于导入的包名太长或者导入的包名冲突的情况。具体语法格式如下：

```
import 别名 "包的路径"
```

单行导入方式定义别名：

```
import "fmt"  
import m "github.com/Q1mi/studygo/pkg_test"  
  
func main() {  
    fmt.Println(m.Add(100, 200))  
    fmt.Println(m.Mode)  
}
```

多行导入方式定义别名：

```
import (
    "fmt"
    m "github.com/Q1mi/studygo/pkg_test"
)

func main() {
    fmt.Println(m.Add(100, 200))
    fmt.Println(m.Mode)
}
```

匿名导入包

如果只希望导入包，而不使用包内部的数据时，可以使用匿名导入包。具体的格式如下：

```
import _ "包的路径"
```

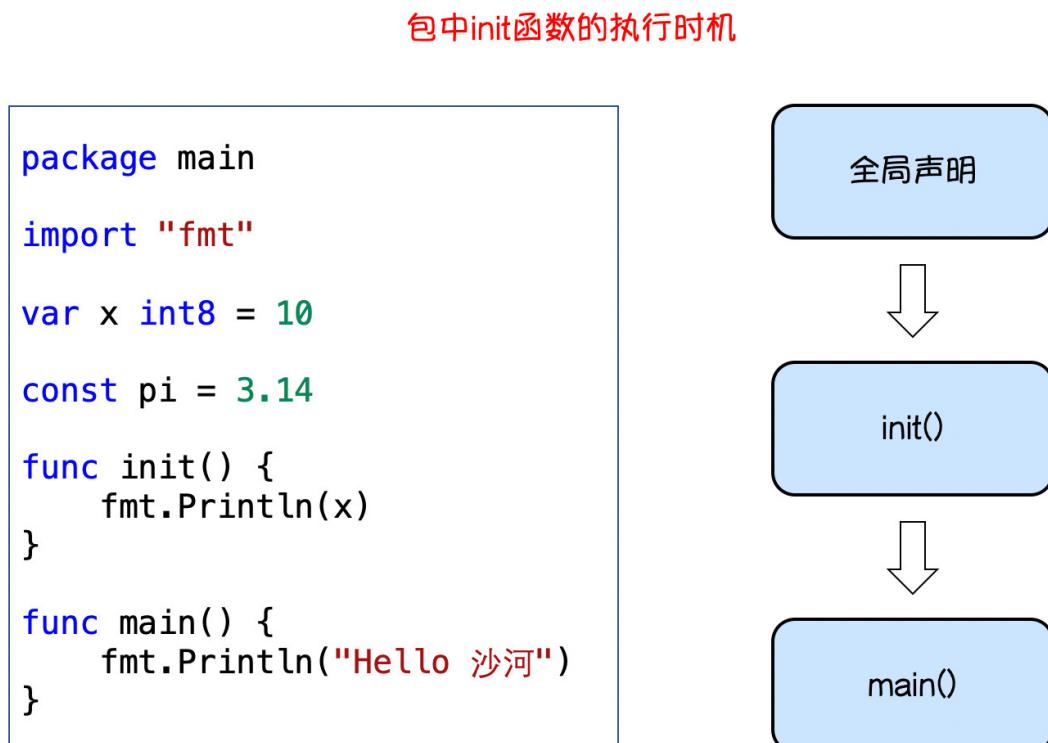
匿名导入的包与其他方式导入的包一样都会被编译到可执行文件中。

init()初始化函数

init()函数介绍

在Go语言程序执行时导入包语句会自动触发包内部 `init()` 函数的调用。需要注意的是：`init()` 函数没有参数也没有返回值。`init()` 函数在程序运行时自动被调用执行，不能在代码中主动调用它。

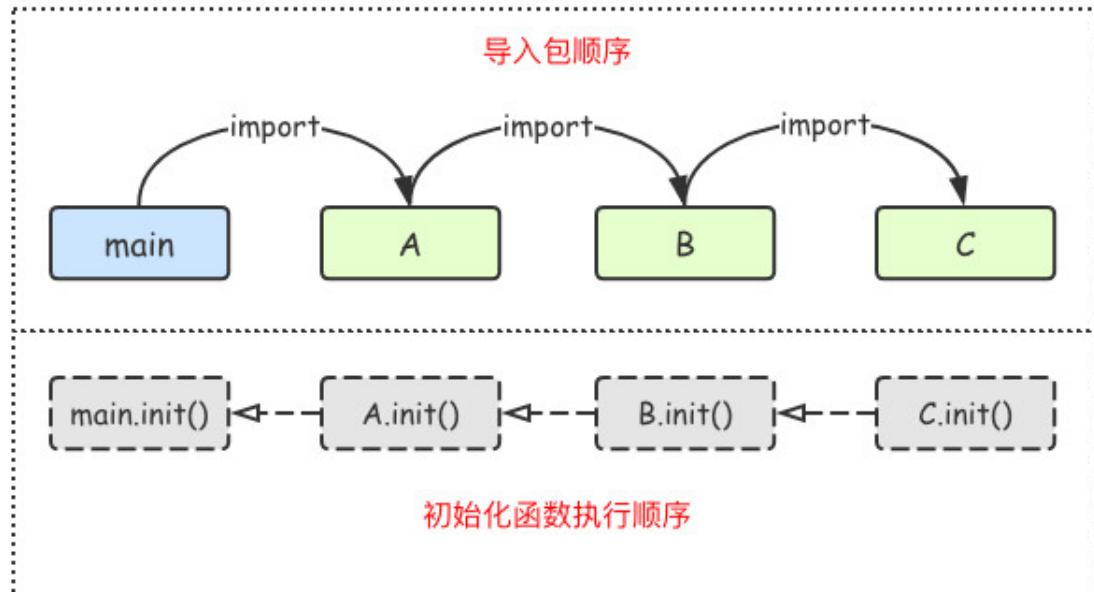
包初始化执行的顺序如下图所示：



init()函数执行顺序

Go语言包会从 `main` 包开始检查其导入的所有包，每个包中又可能导入了其他的包。Go编译器由此构建出一个树状的包引用关系，再根据引用顺序决定编译顺序，依次编译这些包的代码。

在运行时，被最后导入的包会最先初始化并调用其 `init()` 函数，如下图示：



GO语言中的接口

接口类型

在Go语言中接口（interface）是一种类型，一种抽象的类型。

`interface` 是一组 `method` 的集合，是 `duck-type programming` 的一种体现。接口做的事情就像是定义一个协议（规则），只要一台机器有洗衣服和甩干的功能，我就称它为洗衣机。不关心属性（数据），只关心行为（方法）。

为了保护你的Go语言职业生涯，请牢记接口（interface）是一种类型。

为什么要使用接口

```
type Cat struct{}

func (c Cat) Say() string { return "喵喵喵" }

type Dog struct{}

func (d Dog) Say() string { return "汪汪汪" }
```

```
func main() {
    c := Cat{}
    fmt.Println("猫:", c.Say())
    d := Dog{}
    fmt.Println("狗:", d.Say())
}
```

上面的代码中定义了猫和狗，然后它们都会叫，你会发现main函数中明显有重复的代码，如果我们后续再加上猪、青蛙等动物的话，我们的代码还会一直重复下去。那我们能不能把它们当成“能叫的动物”来处理呢？

像类似的例子在我们编程过程中会经常遇到：

比如一个网上商城可能使用支付宝、微信、银联等方式去在线支付，我们能不能把它们当成“支付方式”来处理呢？

比如三角形，四边形，圆形都能计算周长和面积，我们能不能把它们当成“图形”来处理呢？

比如销售、行政、程序员都能计算月薪，我们能不能把他们当成“员工”来处理呢？

Go语言中为了解决类似上面的问题，就设计了接口这个概念。接口区别于我们之前所有的具体类型，接口是一种抽象的类型。当你看到一个接口类型的值时，你不知道它是什么，唯一知道的是通过它的方法能做什么。

接口的定义

Go语言提倡面向接口编程。

每个接口由数个方法组成，接口的定义格式如下：

```
type 接口类型名 interface{
    方法名1( 参数列表1 ) 返回值列表1
    方法名2( 参数列表2 ) 返回值列表2
    ...
}
```

其中：

- 接口名：使用 `type` 将接口定义为自定义的类型名。Go语言的接口在命名时，一般会在单词后面添加 `er`，如有写操作的接口叫 `Writer`，有字符串功能的接口叫 `Stringer` 等。接口名最好要能突出该接口的类型含义。
- 方法名：当方法名首字母是大写且这个接口类型名首字母也是大写时，这个方法可以被接口所在的包（package）之外的代码访问。
- 参数列表、返回值列表：参数列表和返回值列表中的参数变量名可以省略。

举个例子：

```
type writer interface{
    Write([]byte) error
}
```

当你看到这个接口类型的值时，你不知道它是什么，唯一知道的就是可以通过它的Write方法来做一些事情。

实现接口的条件

一个对象只要全部实现了接口中的方法，那么就实现了这个接口。换句话说，接口就是一个需要实现的方法列表。

我们来定义一个 `Sayer` 接口：

```
// Sayer 接口
type Sayer interface {
    say()
}
```

定义 `dog` 和 `cat` 两个结构体：

```
type dog struct {}

type cat struct {}
```

因为 `Sayer` 接口里只有一个 `say` 方法，所以我们只需要给 `dog` 和 `cat` 分别实现 `say` 方法就可以实现 `Sayer` 接口了。

```
// dog实现了Sayer接口
func (d dog) say() {
    fmt.Println("汪汪汪")
}

// cat实现了Sayer接口
func (c cat) say() {
    fmt.Println("喵喵喵")
}
```

接口的实现就是这么简单，只要实现了接口中的所有方法，就实现了这个接口。

接口类型变量

那实现了接口有什么用呢？

接口类型变量能够存储所有实现了该接口的实例。例如上面的示例中，`Sayer` 类型的变量能够存储 `dog` 和 `cat` 类型的变量。

```
func main() {
    var x Sayer // 声明一个Sayer类型的变量x
    a := cat{} // 实例化一个cat
    b := dog{} // 实例化一个dog
    x = a // 可以把cat实例直接赋值给x
    x.say() // 喵喵喵
    x = b // 可以把dog实例直接赋值给x
    x.say() // 汪汪汪
}
```

Tips：观察下面的代码，体味此处 `_` 的妙用

```
// 摘自gin框架routergroup.go
type IRouter interface{ ... }

type RouterGroup struct { ... }

var _ IRouter = &RouterGroup{} // 确保RouterGroup实现了接口IRouter
```

值接收者和指针接收者实现接口的区别

使用值接收者实现接口和使用指针接收者实现接口有什么区别呢？接下来我们通过一个例子看一下其中的区别。

我们有一个 `Mover` 接口和一个 `dog` 结构体。

```
type Mover interface {
    move()
}

type dog struct { }
```

值接收者实现接口

```
func (d dog) move() {
    fmt.Println("狗会动")
}
```

此时实现接口的是 `dog` 类型：

```
func main() {
    var x Mover
    var wangcai = dog{} // 旺财是dog类型
    x = wangcai          // x可以接收dog类型
    var fugui = &dog{}   // 富贵是*dog类型
    x = fugui            // x可以接收*dog类型
    x.move()
}
```

从上面的代码中我们可以发现，使用值接收者实现接口之后，不管是`dog`结构体还是结构体指针`*dog`类型的变量都可以赋值给该接口变量。因为Go语言中有对指针类型变量求值的语法糖，`dog`指针 `fugui` 内部会自动求值 `*fugui`。

指针接收者实现接口

同样的代码我们再来测试一下使用指针接收者有什么区别：

```

func (d *dog) move() {
    fmt.Println("狗会动")
}

func main() {
    var x Mover
    var wangcai = dog{} // 旺财是dog类型
    x = wangcai          // x不可以接收dog类型
    var fugui = &dog{}   // 富贵是*dog类型
    x = fugui            // x可以接收*dog类型
}

```

此时实现 `Mover` 接口的是 `*dog` 类型，所以不能给 `x` 传入 `dog` 类型的 `wangcai`，此时 `x` 只能存储 `*dog` 类型的值。

面试题

注意：这是一道你需要回答“能”或者“不能”的题！

首先请观察下面的这段代码，然后请回答这段代码能不能通过编译？

```

type People interface {
    Speak(string) string
}

type Student struct{}


func (stu *Student) Speak(think string) (talk string) {
    if think == "sb" {
        talk = "你是个大傻比"
    } else {
        talk = "您好"
    }
    return
}

func main() {
    var peo People = Student{}
    think := "bitch"
    fmt.Println(peo.Speak(think))
}

```

类型与接口的关系

一个类型实现多个接口

一个类型可以同时实现多个接口，而接口间彼此独立，不知道对方的实现。例如，狗可以叫，也可以动。我们就分别定义 `Sayer` 接口和 `Mover` 接口，如下：`Mover` 接口。

```
// Sayer 接口
type Sayer interface {
    say()
}

// Mover 接口
type Mover interface {
    move()
}
```

dog既可以实现Sayer接口，也可以实现Mover接口。

```
type dog struct {
    name string
}

// 实现Sayer接口
func (d dog) say() {
    fmt.Printf("%s会叫汪汪汪\n", d.name)
}

// 实现Mover接口
func (d dog) move() {
    fmt.Printf("%s会动\n", d.name)
}

func main() {
    var x Sayer
    var y Mover

    var a = dog{name: "旺财"}
    x = a
    y = a
    x.say()
    y.move()
}
```

多个类型实现同一接口

Go语言中不同的类型还可以实现同一接口 首先我们定义一个 `Mover` 接口，它要求必须由一个 `move` 方法。

```
// Mover 接口
type Mover interface {
    move()
}
```

例如狗可以动，汽车也可以动，可以使用如下代码实现这个关系：

```
type dog struct {
    name string
}

type car struct {
    brand string
}
```

```
// dog类型实现Mover接口
func (d dog) move() {
    fmt.Printf("%s会跑\n", d.name)
}

// car类型实现Mover接口
func (c car) move() {
    fmt.Printf("%s速度70迈\n", c.brand)
}
```

这个时候我们在代码中就可以把狗和汽车当成一个会动的物体来处理了，不再需要关注它们具体是什么，只需要调用它们的 `move` 方法就可以了。

```
func main() {
    var x Mover
    var a = dog{name: "旺财"}
    var b = car{brand: "保时捷"}
    x = a
    x.move()
    x = b
    x.move()
}
```

上面的代码执行结果如下：

```
旺财会跑
保时捷速度70迈
```

并且一个接口的方法，不一定需要由一个类型完全实现，接口的方法可以通过在类型中嵌入其他类型或者结构体来实现。

```
// WashingMachine 洗衣机
type WashingMachine interface {
    wash()
    dry()
}

// 甩干器
type dryer struct{}

// 实现WashingMachine接口的dry()方法
func (d dryer) dry() {
    fmt.Println("甩一甩")
}

// 海尔洗衣机
type haier struct {
    dryer //嵌入甩干器
}

// 实现WashingMachine接口的wash()方法
func (h haier) wash() {
    fmt.Println("洗刷刷")
}
```

接口嵌套

接口与接口间可以通过嵌套创造出新的接口。

```
// Sayer 接口
type Sayer interface {
    say()
}

// Mover 接口
type Mover interface {
    move()
}

// 接口嵌套
type animal interface {
    Sayer
    Mover
}
```

嵌套得到的接口的使用与普通接口一样，这里我们让cat实现animal接口：

```
type cat struct {
    name string
}

func (c cat) say() {
    fmt.Println("喵喵喵")
}

func (c cat) move() {
    fmt.Println("猫会动")
}

func main() {
    var x animal
    x = cat{name: "花花"}
    x.move()
    x.say()
}
```

空接口

空接口的定义

空接口是指没有定义任何方法的接口。因此任何类型都实现了空接口。

空接口类型的变量可以存储任意类型的变量。

```
func main() {
    // 定义一个空接口x
    var x interface{}
    s := "Hello 沙河"
    x = s
    fmt.Printf("type:%T value:%v\n", x, x)
    i := 100
    x = i
    fmt.Printf("type:%T value:%v\n", x, x)
    b := true
    x = b
    fmt.Printf("type:%T value:%v\n", x, x)
}
```

空接口的应用

空接口作为函数的参数

使用空接口实现可以接收任意类型的函数参数。

```
// 空接口作为函数参数
func show(a interface{}) {
    fmt.Printf("type:%T value:%v\n", a, a)
}
```

空接口作为map的值

使用空接口实现可以保存任意值的字典。

```
// 空接口作为map值
var studentInfo = make(map[string]interface{})
studentInfo["name"] = "沙河娜扎"
studentInfo["age"] = 18
studentInfo["married"] = false
fmt.Println(studentInfo)
```

类型断言

空接口可以存储任意类型的值，那我们如何获取其存储的具体数据呢？

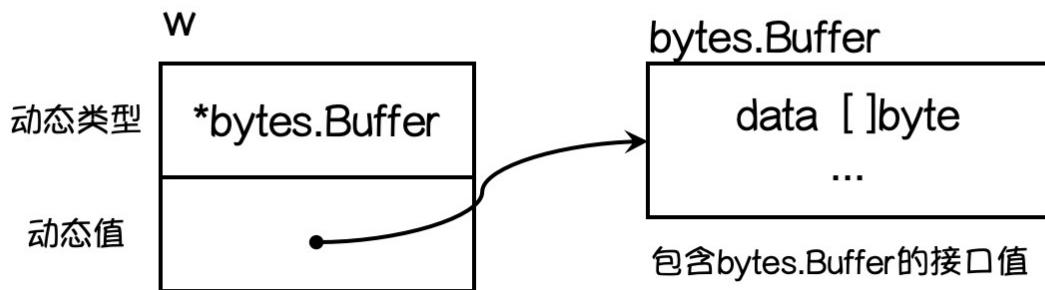
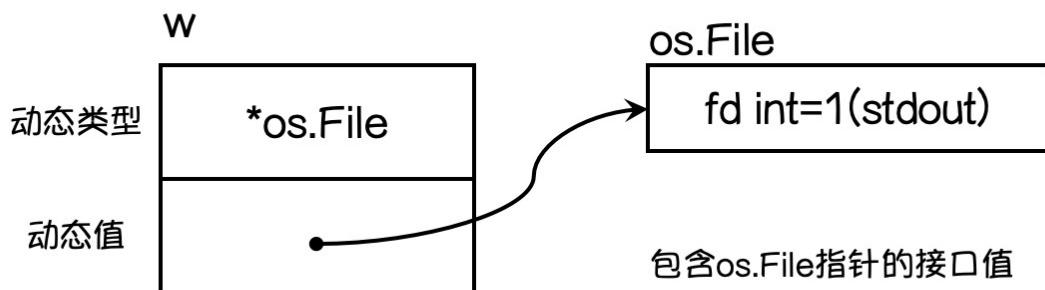
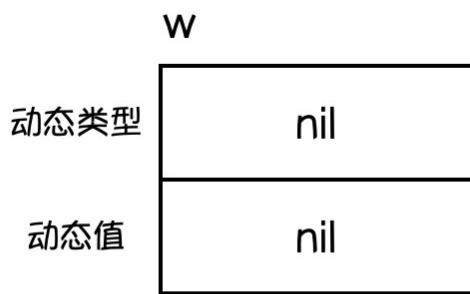
接口值

一个接口的值（简称接口值）是由 **一个具体类型** 和 **具体类型的值** 两部分组成的。这两部分分别称为接口的 **动态类型** 和 **动态值**。

我们来看一个具体的例子：

```
var w io.Writer
w = os.Stdout
w = new(bytes.Buffer)
w = nil
```

请看下图分解：



想要判断空接口中的值这个时候就可以使用类型断言，其语法格式：

```
x.(T)
```

其中：

- x：表示类型为 `interface{}` 的变量
- T：表示断言 `x` 可能是的类型。

该语法返回两个参数，第一个参数是 `x` 转化为 `T` 类型后的变量，第二个值是一个布尔值，若为 `true` 则表示断言成功，为 `false` 则表示断言失败。

举个例子：

```
func main() {
    var x interface{}
    x = "Hello 沙河"
    v, ok := x.(string)
    if ok {
        fmt.Println(v)
    } else {
        fmt.Println("类型断言失败")
    }
}
```

上面的示例中如果要断言多次就需要写多个 `if` 判断，这个时候我们可以使用 `switch` 语句来实现：

```
func justifyType(x interface{}) {
    switch v := x.(type) {
    case string:
        fmt.Printf("x is a string, value is %v\n", v)
    case int:
        fmt.Printf("x is a int is %v\n", v)
    case bool:
        fmt.Printf("x is a bool is %v\n", v)
    default:
        fmt.Println("unsupport type! ")
    }
}
```

因为空接口可以存储任意类型值的特点，所以空接口在Go语言中的使用十分广泛。

关于接口需要注意的是，只有当有两个或两个以上的具体类型必须以相同的方式进行处理时才需要定义接口。不要为了接口而写接口，那样只会增加不必要的抽象，导致不必要的运行时损耗。

GO语言中的反射

变量的内在机制

Go语言中的变量是分为两部分的：

- 类型信息：预先定义好的元信息。
- 值信息：程序运行过程中可动态变化的。

反射介绍

反射是指在程序运行期对程序本身进行访问和修改的能力。程序在编译时，变量被转换为内存地址，变量名不会被编译器写入到可执行部分。在运行程序时，程序无法获取自身的信息。

支持反射的语言可以在程序编译期将变量的反射信息，如字段名称、类型信息、结构体信息等整合到可执行文件中，并给程序提供接口访问反射信息，这样就可以在程序运行期获取类型的反射信息，并且有能力修改它们。

Go程序在运行期使用`reflect`包访问程序的反射信息。

在上一篇博客中我们介绍了空接口。空接口可以存储任意类型的变量，那我们如何知道这个空接口保存的数据是什么呢？反射就是在运行时动态的获取一个变量的类型信息和值信息。

reflect包

在Go语言的反射机制中，任何接口值都由是 **一个具体类型** 和 **具体类型的值** 两部分组成的(我们在上一篇接口的博客中有介绍相关概念)。在Go语言中反射的相关功能由内置的reflect包提供，任意接口值在反射中都可以理解为由 `reflect.Type` 和 `reflect.Value` 两部分组成，并且reflect包提供了 `reflect.TypeOf` 和 `reflect.ValueOf` 两个函数来获取任意对象的Value和Type。

TypeOf

在Go语言中，使用 `reflect.TypeOf()` 函数可以获得任意值的类型对象（`reflect.Type`），程序通过类型对象可以访问任意值的类型信息。

```
package main

import (
    "fmt"
    "reflect"
)

func reflectType(x interface{}) {
    v := reflect.TypeOf(x)
    fmt.Printf("type:%v\n", v)
}

func main() {
    var a float32 = 3.14
    reflectType(a) // type:float32
    var b int64 = 100
    reflectType(b) // type:int64
}
```

type name和type kind

在反射中关于类型还划分为两种：**类型 (Type)** 和 **种类 (Kind)**。因为在Go语言中我们可以使用type关键字构造很多自定义类型，而 **种类 (Kind)** 就是指底层的类型，但在反射中，当需要区分指针、结构体等大品种的类型时，就会用到 **种类 (Kind)**。举个例子，我们定义了两个指针类型和两个结构体类型，通过反射查看它们的类型和种类。

```
package main

import (
    "fmt"
    "reflect"
)

type myInt int64

func reflectType(x interface{}) {
    t := reflect.TypeOf(x)
    fmt.Printf("type:%v kind:%v\n", t.Name(), t.Kind())
}
```

```

func main() {
    var a *float32 // 指针
    var b myInt    // 自定义类型
    var c rune     // 类型别名
    reflectType(a) // type: Kind:ptr
    reflectType(b) // type:myInt kind:int64
    reflectType(c) // type:int32 kind:int32

    type person struct {
        name string
        age  int
    }
    type book struct{ title string }
    var d = person{
        name: "沙河小王子",
        age:  18,
    }
    var e = book{title: "《跟小王子学Go语言》"}
    reflectType(d) // type:person kind:struct
    reflectType(e) // type:book kind:struct
}

```

Go语言的反射中像数组、切片、Map、指针等类型的变量，它们的 `.Name()` 都是返回 空。

在 `reflect` 包中定义的Kind类型如下：

```

type Kind uint
const (
    Invalid Kind = iota // 非法类型
    Bool                // 布尔型
    Int                 // 有符号整型
    Int8               // 有符号8位整型
    Int16              // 有符号16位整型
    Int32              // 有符号32位整型
    Int64              // 有符号64位整型
    Uint               // 无符号整型
    Uint8              // 无符号8位整型
    Uint16             // 无符号16位整型
    Uint32             // 无符号32位整型
    Uint64             // 无符号64位整型
    Uintptr            // 指针
    Float32            // 单精度浮点数
    Float64            // 双精度浮点数
    Complex64          // 64位复数类型
    Complex128         // 128位复数类型
    Array              // 数组
    Chan               // 通道
    Func               // 函数
    Interface          // 接口
    Map                // 映射
    Ptr                // 指针
    Slice              // 切片
    String             // 字符串
    Struct             // 结构体
    UnsafePointer      // 底层指针
)

```

ValueOf

`reflect.ValueOf()` 返回的是 `reflect.Value` 类型，其中包含了原始值的值信息。`reflect.Value` 与原始值之间可以互相转换。

`reflect.Value` 类型提供的获取原始值的方法如下：

方法	说明
Interface() interface{}	将值以 <code>interface{}</code> 类型返回，可以通过类型断言转换为指定类型
Int() int64	将值以 <code>int</code> 类型返回，所有有符号整型均可以此方式返回
Uint() uint64	将值以 <code>uint</code> 类型返回，所有无符号整型均可以此方式返回
Float() float64	将值以双精度 (<code>float64</code>) 类型返回，所有浮点数 (<code>float32</code> 、 <code>float64</code>) 均可以此方式返回
Bool() bool	将值以 <code>bool</code> 类型返回
Bytes() []bytes	将值以字节数组 <code>[]bytes</code> 类型返回
String() string	将值以字符串类型返回

通过反射获取值

```
func reflectValue(x interface{}) {
    v := reflect.ValueOf(x)
    k := v.Kind()
    switch k {
    case reflect.Int64:
        // v.Int()从反射中获取整型的原始值，然后通过int64()强制类型转换
        fmt.Printf("type is int64, value is %d\n", int64(v.Int()))
    case reflect.Float32:
        // v.Float()从反射中获取浮点型的原始值，然后通过float32()强制类型转换
        fmt.Printf("type is float32, value is %f\n", float32(v.Float()))
    case reflect.Float64:
        // v.Float()从反射中获取浮点型的原始值，然后通过float64()强制类型转换
        fmt.Printf("type is float64, value is %f\n", float64(v.Float()))
    }
}

func main() {
    var a float32 = 3.14
    var b int64 = 100
    reflectValue(a) // type is float32, value is 3.140000
    reflectValue(b) // type is int64, value is 100
    // 将int类型的原始值转换为reflect.Value类型
    c := reflect.ValueOf(10)
    fmt.Printf("type c :%T\n", c) // type c :reflect.Value
}
```

通过反射设置变量的值

想要在函数中通过反射修改变量的值，需要注意函数参数传递的是值拷贝，必须传递变量地址才能修改变量值。而反射中使用专有的 `Elem()` 方法来获取指针对应的值。

```
package main

import (
    "fmt"
    "reflect"
)

func reflectSetValue1(x interface{}) {
    v := reflect.ValueOf(x)
    if v.Kind() == reflect.Int64 {
        v.SetInt(200) //修改的是副本, reflect包会引发panic
    }
}

func reflectSetValue2(x interface{}) {
    v := reflect.ValueOf(x)
    // 反射中使用 Elem()方法获取指针对应的值
    if v.Elem().Kind() == reflect.Int64 {
        v.Elem().SetInt(200)
    }
}

func main() {
    var a int64 = 100
    // reflectSetValue1(a) //panic: reflect: reflect.Value.SetInt using unaddressable value
    reflectSetValue2(&a)
    fmt.Println(a)
}
```

isNil()和isValid()

isNil()

```
func (v Value) IsNil() bool
```

`IsNil()` 报告v持有的值是否为nil。v持有的值的分类必须是通道、函数、接口、映射、指针、切片之一；否则 IsNil 函数会导致 panic。

isValid()

```
func (v Value) IsValid() bool
```

`IsValid()` 返回v是否持有一个值。如果v是Value零值会返回假，此时v除了 IsValid、String、Kind之外的方法都会导致 panic。

举个例子

`IsNil()` 常被用于判断指针是否为空；`IsValid()` 常被用于判定返回值是否有效。

```
func main() {
    // *int类型空指针
    var a *int
    fmt.Println("var a *int IsNil:", reflect.ValueOf(a).IsNil())
    // nil值
    fmt.Println("nil IsValid:", reflect.ValueOf(nil).IsValid())
    // 实例化一个匿名结构体
    b := struct{}{}
    // 尝试从结构体中查找"abc"字段
    fmt.Println("不存在的结构体成员:", reflect.ValueOf(b).FieldByName("abc").IsValid())
    // 尝试从结构体中查找"abc"方法
    fmt.Println("不存在的结构体方法:", reflect.ValueOf(b).MethodByName("abc").IsValid())
    // map
    c := map[string]int{}
    // 尝试从map中查找一个不存在的键
    fmt.Println("map中不存在的键: ", reflect.ValueOf(c).MapIndex(reflect.ValueOf("娜扎")).IsValid())
}
```

结构体反射

与结构体相关的方法

任意值通过 `reflect.TypeOf()` 获得反射对象信息后，如果它的类型是结构体，可以通过反射值对象 (`reflect.Type`) 的 `NumField()` 和 `Field()` 方法获得结构体成员的详细信息。

`reflect.Type` 中与获取结构体成员相关的的方法如下表所示。

方法	说明
Field(i int) StructField	根据索引，返回索引对应的结构体字段的信息。
NumField() int	返回结构体成员字段数量。
FieldByName(name string) (StructField, bool)	根据给定字符串返回字符串对应的结构体字段的信息。
FieldByIndex(index []int) StructField	多层成员访问时，根据 []int 提供的每个结构体的字段索引，返回字段的信息。
FieldByNameFunc(match func(string) bool) (StructField, bool)	根据传入的匹配函数匹配需要的字段。
NumMethod() int	返回该类型的方法集中方法的数目
Method(int) Method	返回该类型方法集中的第i个方法
MethodByName(string) (Method, bool)	根据方法名返回该类型方法集中的方法

StructField类型

`StructField` 类型用来描述结构体中的一个字段的信息。

`StructField` 的定义如下：

```
type StructField struct {
    // Name是字段的名字。PkgPath是非导出字段的包路径，对导出字段该字段为""。
    // 参见http://golang.org/ref/spec#Uniqueness_of_identifiers
    Name      string
    PkgPath   string
    Type      Type      // 字段的类型
    Tag       StructTag // 字段的标签
    Offset    uintptr   // 字段在结构体中的字节偏移量
    Index    []int     // 用于Type.FieldByIndex时的索引切片
    Anonymous bool     // 是否匿名字段
}
```

结构体反射示例

当我们使用反射得到一个结构体数据之后可以通过索引依次获取其字段信息，也可以通过字段名去获取指定的字段信息。

```
type student struct {
    Name  string `json:"name"`
    Score int    `json:"score"`
}

func main() {
    stu1 := student{
        Name:  "小王子",
        Score: 90,
    }

    t := reflect.TypeOf(stu1)
    fmt.Println(t.Name(), t.Kind()) // student struct
    // 通过for循环遍历结构体的所有字段信息
    for i := 0; i < t.NumField(); i++ {
        field := t.Field(i)
        fmt.Printf("name:%s index:%d type:%v json tag:%v\n", field.Name, field.Index, field.Type,
        field.Tag.Get("json"))
    }

    // 通过字段名获取指定结构体字段信息
    if scoreField, ok := t.FieldByName("Score"); ok {
        fmt.Printf("name:%s index:%d type:%v json tag:%v\n", scoreField.Name, scoreField.Index,
        scoreField.Type, scoreField.Tag.Get("json"))
    }
}
```

接下来编写一个函数 `printMethod(s interface{})` 来遍历打印s包含的方法。

```
// 给student添加两个方法 Study和Sleep(注意首字母大写)
func (s student) Study() string {
    msg := "好好学习，天天向上。"
```

```
fmt.Println(msg)
return msg
}

func (s *student) Sleep() string {
    msg := "好好睡觉，快快长大。"
    fmt.Println(msg)
    return msg
}

func printMethod(x interface{}) {
    t := reflect.TypeOf(x)
    v := reflect.ValueOf(x)

    fmt.Println(t.NumMethod())
    for i := 0; i < v.NumMethod(); i++ {
        methodType := v.Method(i).Type()
        fmt.Printf("method name:%s\n", t.Method(i).Name)
        fmt.Printf("method:%s\n", methodType)
        // 通过反射调用方法传递的参数必须是 []reflect.Value 类型
        var args = []reflect.Value{}
        v.Method(i).Call(args)
    }
}
```

反射是把双刃剑

反射是一个强大并富有表现力的工具，能让我们写出更灵活的代码。但是反射不应该被滥用，原因有以下三个。

1. 基于反射的代码是极其脆弱的，反射中的类型错误会在真正运行的时候才会引发panic，那很可能是在代码写完的很长时间之后。
2. 大量使用反射的代码通常难以理解。
3. 反射的性能低下，基于反射实现的代码通常比正常代码运行速度慢一到两个数量级。

Go语言中的并发编程

并发与并行

并发：同一时间段内执行多个任务（你在用微信和两个女朋友聊天）。

并行：同一时刻执行多个任务（你和你朋友都在用微信和女朋友聊天）。

Go语言的并发通过 `goroutine` 实现。`goroutine` 类似于线程，属于用户态的线程，我们可以根据需要创建成千上万个 `goroutine` 并发工作。`goroutine` 是由Go语言的运行时（runtime）调度完成，而线程是由操作系统调度完成。

Go语言还提供 `channel` 在多个 `goroutine` 间进行通信。`goroutine` 和 `channel` 是 Go 语言秉承的 CSP (Communicating Sequential Process) 并发模式的重要实现基础。

goroutine

在java/c++中我们要实现并发编程的时候，我们通常需要自己维护一个线程池，并且需要自己去包装一个又一个的任务，同时需要自己去调度线程执行任务并维护上下文切换，这一切通常会耗费程序员大量的心智。那么能不能有一种机制，程序员只需要定义很多个任务，让系统去帮助我们把这些任务分配到CPU上实现并发执行呢？

Go语言中的 `goroutine` 就是这样一种机制，`goroutine` 的概念类似于线程，但 `goroutine` 是由Go的运行时 (runtime) 调度和管理的。Go程序会智能地将 `goroutine` 中的任务合理地分配给每个CPU。Go语言之所以被称为现代化的编程语言，就是因为它在语言层面已经内置了调度和上下文切换的机制。

在Go语言编程中你不需要去自己写进程、线程、协程，你的技能包里只有一个技能- `goroutine`，当你需要让某个任务并发执行的时候，你只需要把这个任务包装成一个函数，开启一个 `goroutine` 去执行这个函数就可以了，就是这么简单粗暴。

使用goroutine

Go语言中使用 `goroutine` 非常简单，只需要在调用函数的时候在前面加上 `go` 关键字，就可以为一个函数创建一个 `goroutine`。

一个 `goroutine` 必定对应一个函数，可以创建多个 `goroutine` 去执行相同的函数。

启动单个goroutine

启动goroutine的方式非常简单，只需要在调用的函数（普通函数和匿名函数）前面加上一个 `go` 关键字。

举个例子如下：

```
func hello() {
    fmt.Println("Hello Goroutine!")
}

func main() {
    hello()
    fmt.Println("main goroutine done!")
}
```

这个示例中hello函数和下面的语句是串行的，执行的结果是打印完 `Hello Goroutine!` 后打印 `main goroutine done!`。

接下来我们在调用hello函数前面加上关键字 `go`，也就是启动一个goroutine去执行hello这个函数。

```
func main() {
    go hello() // 启动另外一个goroutine去执行hello函数
    fmt.Println("main goroutine done!")
}
```

这一次的执行结果只打印了 `main goroutine done!`，并没有打印 `Hello Goroutine!`。为什么呢？

在程序启动时，Go程序就会为 `main()` 函数创建一个默认的 `goroutine`。

当 `main()` 函数返回的时候该 `goroutine` 就结束了，所有在 `main()` 函数中启动的 `goroutine` 会一同结束，`main` 函数所在的 `goroutine` 就像是权利的游戏中的夜王，其他的 `goroutine` 都是异鬼，夜王一死它转化的那些异鬼也就全部GG了。

所以我们要想办法让 `main` 函数等一等 `hello` 函数，最简单粗暴的方式就是 `time.Sleep` 了。

```
func main() {
    go hello() // 启动另外一个goroutine去执行hello函数
    fmt.Println("main goroutine done!")
    time.Sleep(time.Second)
}
```

执行上面的代码你会发现，这一次先打印 `main goroutine done!`，然后紧接着打印 `Hello Goroutine!`。

首先为什么会先打印 `main goroutine done!` 是因为在创建新的 `goroutine` 的时候需要花费一些时间，而此时 `main` 函数所在的 `goroutine` 是继续执行的。

启动多个goroutine

在 Go 语言中实现并发就是这样简单，我们还可以启动多个 `goroutine`。让我们再来一个例子：（这里使用了 `sync.WaitGroup` 来实现 `goroutine` 的同步）

```
var wg sync.WaitGroup

func hello(i int) {
    defer wg.Done() // goroutine结束就登记-1
    fmt.Println("Hello Goroutine!", i)
}

func main() {

    for i := 0; i < 10; i++ {
        wg.Add(1) // 启动一个goroutine就登记+1
        go hello(i)
    }
    wg.Wait() // 等待所有登记的goroutine都结束
}
```

多次执行上面的代码，会发现每次打印的数字的顺序都不一致。这是因为 10 个 `goroutine` 是并发执行的，而 `goroutine` 的调度是随机的。

goroutine与线程

可增长的栈

OS 线程（操作系统线程）一般都有固定的栈内存（通常为 2MB），一个 `goroutine` 的栈在其生命周期开始时只有很小的栈（典型情况下 2KB），`goroutine` 的栈不是固定的，它可以按需增大和缩小，`goroutine` 的栈大小限制可以达到 1GB，虽然极少会用到这么大。所以在 Go 语言中一次创建十万左右的 `goroutine` 也是可以的。

goroutine调度

GPM 是Go语言运行时 (runtime) 层面的实现，是go语言自己实现的一套调度系统。区别于操作系统调度OS线程。

- G 很好理解，就是个goroutine的，里面除了存放本goroutine信息外 还有与所在P的绑定等信息。
- P 管理着一组goroutine队列，P里面会存储当前goroutine运行的上下文环境（函数指针，堆栈地址及地址边界），P会对自己管理的goroutine队列做一些调度（比如把占用CPU时间较长的goroutine暂停、运行后续的goroutine等等）当自己的队列消费完了就去全局队列里取，如果全局队列里也消费完了会去其他P的队列里抢任务。
- M (machine) 是Go运行时 (runtime) 对操作系统内核线程的虚拟，M与内核线程一般是一一映射的关系，一个groutine最终是要放到M上执行的；

P与M一般也是一一对应的。他们关系是：P管理着一组G挂载在M上运行。当一个G长久阻塞在一个M上时，runtime会新建一个M，阻塞G所在的P会把其他的G挂载在新建的M上。当旧的G阻塞完成或者认为其已经死掉时回收旧的M。

P的个数是通过 `runtime.GOMAXPROCS` 设定（最大256），Go1.5版本之后默认为物理线程数。在并发量大的时候会增加一些P和M，但不会太多，切换太频繁的话得不偿失。

单从线程调度讲，Go语言相比起其他语言的优势在于OS线程是由OS内核来调度的，goroutine则是由Go运行时 (runtime) 自己的调度器调度的，这个调度器使用一个称为m:n调度的技术（复用/调度m个goroutine到n个OS线程）。其一大特点是goroutine的调度是在用户态下完成的，不涉及内核态与用户态之间的频繁切换，包括内存的分配与释放，都是在用户态维护着一块大的内存池，不直接调用系统的malloc函数（除非内存池需要改变），成本比调度OS线程低很多。另一方面充分利用了多核的硬件资源，近似的把若干goroutine均分在物理线程上，再加上本身goroutine的超轻量，以上种种保证了go调度方面的性能。

[深入Golang调度器之GMP模型 - sunsky303 - 博客园 \(cnblogs.com\)](#)

GOMAXPROCS

Go运行时的调度器使用 `GOMAXPROCS` 参数来确定需要使用多少个OS线程来同时执行Go代码。默认值是机器上的CPU核心数。例如在一个8核心的机器上，调度器会把Go代码同时调度到8个OS线程上（`GOMAXPROCS`是m:n调度中的n）。

Go语言中可以通过 `runtime.GOMAXPROCS()` 函数设置当前程序并发时占用的CPU逻辑核心数。

Go1.5版本之前，默认使用的是单核心执行。Go1.5版本之后，默认使用全部的CPU逻辑核心数。

我们可以通过将任务分配到不同的CPU逻辑核心上实现并行的效果，这里举个例子：

```
func a() {  
    for i := 1; i < 10; i++ {  
        fmt.Println("A:", i)  
    }  
}
```

```

func b() {
    for i := 1; i < 10; i++ {
        fmt.Println("B:", i)
    }
}

func main() {
    runtime.GOMAXPROCS(1)
    go a()
    go b()
    time.Sleep(time.Second)
}

```

两个任务只有一个逻辑核心，此时是做完一个任务再做另一个任务。将逻辑核心数设为2，此时两个任务并行执行，代码如下。

```

func a() {
    for i := 1; i < 10; i++ {
        fmt.Println("A:", i)
    }
}

func b() {
    for i := 1; i < 10; i++ {
        fmt.Println("B:", i)
    }
}

func main() {
    runtime.GOMAXPROCS(2)
    go a()
    go b()
    time.Sleep(time.Second)
}

```

Go语言中的操作系统线程和goroutine的关系：

1. 一个操作系统线程对应用户态多个goroutine。
2. go程序可以同时使用多个操作系统线程。
3. goroutine和OS线程是多对多的关系，即m:n。

channel

单纯地将函数并发执行是没有意义的。函数与函数间需要交换数据才能体现并发执行函数的意义。

虽然可以使用共享内存进行数据交换，但是共享内存存在不同的 `goroutine` 中容易发生竞态问题。为了保证数据交换的正确性，必须使用互斥量对内存进行加锁，这种做法势必造成性能问题。

Go语言的并发模型是 `CSP (Communicating Sequential Processes)`，提倡通过通信共享内存而不是通过共享内存而实现通信。

如果说 `goroutine` 是 Go 程序并发的执行体，`channel` 就是它们之间的连接。`channel` 是可以让一个 `goroutine` 发送特定值到另一个 `goroutine` 的通信机制。

Go 语言中的通道（channel）是一种特殊的类型。通道像一个传送带或者队列，总是遵循先入先出（First In First Out）的规则，保证收发数据的顺序。每一个通道都是一个具体类型的导管，也就是声明 `channel` 的时候需要为其指定元素类型。

channel类型

`channel` 是一种类型，一种引用类型。声明通道类型的格式如下：

```
var 变量 chan 元素类型
```

举几个例子：

```
var ch1 chan int    // 声明一个传递整型的通道  
var ch2 chan bool   // 声明一个传递布尔型的通道  
var ch3 chan []int  // 声明一个传递int切片的通道
```

创建channel

通道是引用类型，通道类型的空值是 `nil`。

```
var ch chan int  
fmt.Println(ch) // <nil>
```

声明的通道后需要使用 `make` 函数初始化之后才能使用。

创建 `channel` 的格式如下：

```
make(chan 元素类型, [缓冲大小])
```

`channel` 的缓冲大小是可选的。

举几个例子：

```
ch4 := make(chan int)  
ch5 := make(chan bool)  
ch6 := make(chan []int)
```

channel操作

通道有发送（send）、接收（receive）和关闭（close）三种操作。

发送和接收都使用 `<-` 符号。

现在我们先使用以下语句定义一个通道：

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

发送

将一个值发送到通道中。

```
ch <- 10 // 把10发送到ch中
```

接收

从一个通道中接收值。

```
x := <- ch // 从ch中接收值并赋值给变量x  
<-ch      // 从ch中接收值，忽略结果
```

关闭

我们通过调用内置的 `close` 函数来关闭通道。

```
close(ch)
```

关于关闭通道需要注意的事情是，只有在通知接收方goroutine所有的数据都发送完毕的时候才需要关闭通道。通道是可以被垃圾回收机制回收的，它和关闭文件是不一样的，在结束操作之后关闭文件是必须要做的，但关闭通道不是必须的。

关闭后的通道有以下特点：

1. 对一个关闭的通道再发送值就会导致panic。
2. 对一个关闭的通道进行接收会一直获取值直到通道为空。
3. 对一个关闭的并且没有值的通道执行接收操作会得到对应类型的零值。
4. 关闭一个已经关闭的通道会导致panic。

无缓冲的通道

无缓冲的通道又称为阻塞的通道。我们来看一下下面的代码：

```
func main() {  
    ch := make(chan int)  
    ch <- 10  
    fmt.Println("发送成功")  
}
```

上面这段代码能够通过编译，但是执行的时候会出现以下错误：

```
fatal error: all goroutines are asleep - deadlock!  
  
goroutine 1 [chan send]:  
main.main()  
.../src/github.com/Q1mi/studygo/day06/channel02/main.go:8 +0x54
```

为什么会出现 `deadlock` 错误呢？

因为我们使用 `ch := make(chan int)` 创建的是无缓冲的通道，无缓冲的通道只有在有人接收值的时候才能发送值。就像你住的小区没有快递柜和代收点，快递员给你打电话必须要把这个物品送到你的手中，简单来说就是无缓冲的通道必须有接收才能发送。

上面的代码会阻塞在 `ch <- 10` 这一行代码形成死锁，那如何解决这个问题呢？

一种方法是启用一个 `goroutine` 去接收值，例如：

```
func recv(c chan int) {
    ret := <-c
    fmt.Println("接收成功", ret)
}

func main() {
    ch := make(chan int)
    go recv(ch) // 启用goroutine从通道接收值
    ch <- 10
    fmt.Println("发送成功")
}
```

无缓冲通道上的发送操作会阻塞，直到另一个 `goroutine` 在该通道上执行接收操作，这时值才能发送成功，两个 `goroutine` 将继续执行。相反，如果接收操作先执行，接收方的 `goroutine` 将阻塞，直到另一个 `goroutine` 在该通道上发送一个值。

使用无缓冲通道进行通信将导致发送和接收的 `goroutine` 同步化。因此，无缓冲通道也被称为 **同步通道**。

有缓冲的通道

解决上面问题的方法还有一种就是使用有缓冲区的通道。我们可以在使用 `make` 函数初始化通道的时候为其指定通道的容量，例如：

```
func main() {
    ch := make(chan int, 1) // 创建一个容量为1的有缓冲区通道
    ch <- 10
    fmt.Println("发送成功")
}
```

只要通道的容量大于零，那么该通道就是有缓冲的通道，通道的容量表示通道中能存放元素的数量。就像你小区的快递柜只有那么多格子，格子满了就装不下了，就阻塞了，等到别人取走一个快递员就能往里面放一个。

我们可以使用内置的 `len` 函数获取通道内元素的数量，使用 `cap` 函数获取通道的容量，虽然我们很少会这么做。

for range 从通道循环取值

当向通道中发送完数据时，我们可以通过 `close` 函数来关闭通道。

当通道被关闭时，再往该通道发送值会引发 `panic`，从该通道取值的操作会先取完通道中的值，再然后取到的值一直都是对应类型的零值。那如何判断一个通道是否被关闭了呢？

我们来看下面这个例子：

```
// channel 练习
```

```

func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    // 开启goroutine将0~100的数发送到ch1中
    go func() {
        for i := 0; i < 100; i++ {
            ch1 <- i
        }
        close(ch1)
    }()
    // 开启goroutine从ch1中接收值，并将该值的平方发送到ch2中
    go func() {
        for {
            i, ok := <-ch1 // 通道关闭后再取值ok=false
            if !ok {
                break
            }
            ch2 <- i * i
        }
        close(ch2)
    }()
    // 在主goroutine中从ch2中接收值打印
    for i := range ch2 { // 通道关闭后会退出for range循环
        fmt.Println(i)
    }
}

```

从上面的例子中我们看到有两种方式在接收值的时候判断该通道是否被关闭，不过我们通常使用的是 `for range` 的方式。使用 `for range` 遍历通道，当通道被关闭的时候就会退出 `for range`。

单向通道

有的时候我们会将通道作为参数在多个任务函数间传递，很多时候我们在不同的任务函数中使用通道都会对其进行限制，比如限制通道在函数中只能发送或只能接收。

Go语言中提供了单向通道来处理这种情况。例如，我们把上面的例子改造如下：

```

func counter(out chan<- int) {
    for i := 0; i < 100; i++ {
        out <- i
    }
    close(out)
}

func squarer(out chan<- int, in <-chan int) {
    for i := range in {
        out <- i * i
    }
    close(out)
}

func printer(in <-chan int) {
    for i := range in {
        fmt.Println(i)
    }
}

```

```

func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    go counter(ch1)
    go squarer(ch2, ch1)
    printer(ch2)
}

```

其中，

- `chan<- int` 是一个只写单向通道（只能对其写入int类型值），可以对其执行发送操作但是不能执行接收操作；
- `<-chan int` 是一个只读单向通道（只能从其读取int类型值），可以对其执行接收操作但是不能执行发送操作。

在函数传参及任何赋值操作中可以将双向通道转换为单向通道，但反过来是不可以的。

通道总结

`channel` 常见的异常总结，如下图：

channel异常情况总结					
channel	nil	非空	空的	满了	没满
接收	阻塞	接收值	阻塞	接收值	接收值
发送	阻塞	发送值	发送值	阻塞	发送值
关闭	panic	关闭成功， 读完数据后 返回零值	关闭成功， 返回零值	关闭成功， 读完数据后 返回零值	关闭成功， 读完数据后 返回零值

关闭已经关闭的 `channel` 也会引发 `panic`。

worker pool (goroutine池)

在工作中我们通常会使用可以指定启动的goroutine数量- `worker pool` 模式，控制 `goroutine` 的数量，防止 `goroutine` 泄漏和暴涨。

一个简易的 `work pool` 示例代码如下：

```

func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Printf("worker:%d start job:%d\n", id, j)
        time.Sleep(time.Second)
        fmt.Printf("worker:%d end job:%d\n", id, j)
        results <- j * 2
    }
}

```

```
}
```



```
func main() {
    jobs := make(chan int, 100)
    results := make(chan int, 100)
    // 开启3个goroutine
    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }
    // 5个任务
    for j := 1; j <= 5; j++ {
        jobs <- j
    }
    close(jobs)
    // 输出结果
    for a := 1; a <= 5; a++ {
        <-results
    }
}
```

select多路复用

在某些场景下我们需要同时从多个通道接收数据。通道在接收数据时，如果没有数据可以接收将会发生阻塞。你也许会写出如下代码使用遍历的方式来实现：

```
for{
    // 尝试从ch1接收值
    data, ok := <-ch1
    // 尝试从ch2接收值
    data, ok := <-ch2
    ...
}
```

这种方式虽然可以实现从多个通道接收值的需求，但是运行性能会差很多。为了应对这种场景，Go内置了 `select` 关键字，可以同时响应多个通道的操作。

`select` 的使用类似于 `switch` 语句，它有一系列 `case` 分支和一个默认的分支。每个 `case` 会对应一个通道的通信（接收或发送）过程。`select` 会一直等待，直到某个 `case` 的通信操作完成时，就会执行 `case` 分支对应的语句。具体格式如下：

```
select{
    case <-ch1:
        ...
    case data := <-ch2:
        ...
    case ch3<-data:
        ...
    default:
        默认操作
}
```

举个小例子来演示下 `select` 的使用：

```
func main() {
    ch := make(chan int, 1)
    for i := 0; i < 10; i++ {
        select {
        case x := <-ch:
            fmt.Println(x)
        case ch <- i:
        }
    }
}
```

使用 `select` 语句能提高代码的可读性。

- 可处理一个或多个channel的发送/接收操作。
- 如果多个 `case` 同时满足， `select` 会随机选择一个。
- 对于没有 `case` 的 `select{}` 会一直等待， 可用于阻塞main函数。

并发安全和锁

有时候在Go代码中可能会存在多个 `goroutine` 同时操作一个资源（临界区），这种情况会发生 [竞态问题](#)（数据竞态）。类比现实生活中的例子有十字路口被各个方向的汽车竞争；还有火车上的卫生间被车厢里的人竞争。

举个例子：

```
var x int64
var wg sync.WaitGroup

func add() {
    for i := 0; i < 5000; i++ {
        x = x + 1
    }
    wg.Done()
}

func main() {
    wg.Add(2)
    go add()
    go add()
    wg.Wait()
    fmt.Println(x)
}
```

上面的代码中我们开启了两个 `goroutine` 去累加变量x的值，这两个 `goroutine` 在访问和修改 `x` 变量的时候就会存在数据竞争，导致最后的结果与期待的不符。

互斥锁

互斥锁是一种常用的控制共享资源访问的方法，它能够保证同时只有一个 `goroutine` 可以访问共享资源。Go语言中使用 `sync` 包的 `Mutex` 类型来实现互斥锁。使用互斥锁来修复上面代码的问题：

```
var x int64
var wg sync.WaitGroup
```

```

var lock sync.Mutex

func add() {
    for i := 0; i < 5000; i++ {
        lock.Lock() // 加锁
        x = x + 1
        lock.Unlock() // 解锁
    }
    wg.Done()
}

func main() {
    wg.Add(2)
    go add()
    go add()
    wg.Wait()
    fmt.Println(x)
}

```

使用互斥锁能够保证同一时间有且只有一个 `goroutine` 进入临界区，其他的 `goroutine` 则在等待锁；当互斥锁释放后，等待的 `goroutine` 才可以获取锁进入临界区，多个 `goroutine` 同时等待一个锁时，唤醒的策略是随机的。

读写互斥锁

互斥锁是完全互斥的，但是有很多实际的场景下是读多写少的，当我们并发的去读取一个资源不涉及资源修改的时候是没有必要加锁的，这种场景下使用读写锁是更好的一种选择。读写锁在Go语言中使用 `sync` 包中的 `RWMutex` 类型。

读写锁分为两种：读锁和写锁。当一个 `goroutine` 获取读锁之后，其他的 `goroutine` 如果是获取读锁会继续获得锁，如果是获取写锁就会等待；当一个 `goroutine` 获取写锁之后，其他的 `goroutine` 无论是获取读锁还是写锁都会等待。

读写锁示例：

```

var (
    x      int64
    wg    sync.WaitGroup
    lock  sync.Mutex
    rwlock sync.RWMutex
)

func write() {
    // lock.Lock() // 加互斥锁
    rwlock.Lock() // 加写锁
    x = x + 1
    time.Sleep(10 * time.Millisecond) // 假设读操作耗时10毫秒
    rwlock.Unlock()                // 解写锁
    // lock.Unlock()              // 解互斥锁
    wg.Done()
}

func read() {
    // lock.Lock() // 加互斥锁

```

```

rwlock.RLock()           // 加读锁
time.Sleep(time.Millisecond) // 假设读操作耗时1毫秒
rwlock.RUnlock()          // 解读锁
// lock.Unlock()          // 解互斥锁
wg.Done()
}

func main() {
start := time.Now()
for i := 0; i < 10; i++ {
    wg.Add(1)
    go write()
}

for i := 0; i < 1000; i++ {
    wg.Add(1)
    go read()
}

wg.Wait()
end := time.Now()
fmt.Println(end.Sub(start))
}

```

需要注意的是读写锁非常适合读多写少的场景，如果读和写的操作差别不大，读写锁的优势就发挥不出来。

sync.WaitGroup

在代码中生硬的使用 `time.Sleep` 肯定是不合适的，Go语言中可以使用 `sync.WaitGroup` 来实现并发任务的同步。`sync.WaitGroup` 有以下几个方法：

方法名	功能
(wg * WaitGroup) Add(delta int)	计数器+delta
(wg *WaitGroup) Done()	计数器-1
(wg *WaitGroup) Wait()	阻塞直到计数器变为0

`sync.WaitGroup` 内部维护着一个计数器，计数器的值可以增加和减少。例如当我们启动了N个并发任务时，就将计数器值增加N。每个任务完成时通过调用Done()方法将计数器减1。通过调用Wait()来等待并发任务执行完，当计数器值为0时，表示所有并发任务已经完成。

我们利用 `sync.WaitGroup` 将上面的代码优化一下：

```
var wg sync.WaitGroup

func hello() {
    defer wg.Done()
    fmt.Println("Hello Goroutine!")
}

func main() {
    wg.Add(1)
    go hello() // 启动另外一个goroutine去执行hello函数
    fmt.Println("main goroutine done!")
    wg.Wait()
}
```

需要注意 `sync.WaitGroup` 是一个结构体，传递的时候要传递指针。

sync.Once

说在前面的话：这是一个进阶知识点。

在编程的很多场景下我们需要确保某些操作在高并发的场景下只执行一次，例如只加载一次配置文件、只关闭一次通道等。

Go语言中的 `sync` 包中提供了一个针对只执行一次场景的解决方案- `sync.Once`。

`sync.Once` 只有一个 `Do` 方法，其签名如下：

```
func (o *Once) Do(f func()) {}
```

备注：如果要执行的函数 `f` 需要传递参数就需要搭配闭包来使用。

加载配置文件示例

延迟一个开销很大的初始化操作到真正用到它的时候再执行是一个很好的实践。因为预先初始化一个变量（比如在 `init` 函数中完成初始化）会增加程序的启动耗时，而且有可能实际执行过程中这个变量没有用上，那么这个初始化操作就不是必须要做的。我们来看一个例子：

```
var icons map[string]image.Image

func loadIcons() {
    icons = map[string]image.Image{
        "left":  loadImage("left.png"),
        "up":    loadImage("up.png"),
        "right": loadImage("right.png"),
        "down":  loadImage("down.png"),
    }
}

// Icon 被多个goroutine调用时不是并发安全的
func Icon(name string) image.Image {
    if icons == nil {
        loadIcons()
    }
    return icons[name]
```

```
}
```

多个 `goroutine` 并发调用 `Icon` 函数时不是并发安全的，现代的编译器和 CPU 可能在保证每个 `goroutine` 都满足串行一致的基础上自由地重排访问内存的顺序。`loadIcons` 函数可能会被重排为以下结果：

```
func loadIcons() {
    icons = make(map[string]image.Image)
    icons["left"] = loadIcon("left.png")
    icons["up"] = loadIcon("up.png")
    icons["right"] = loadIcon("right.png")
    icons["down"] = loadIcon("down.png")
}
```

在这种情况下就会出现即使判断了 `icons` 不是 `nil` 也不意味着变量初始化完成了。考虑到这种情况，我们能想到的办法就是添加互斥锁，保证初始化 `icons` 的时候不会被其他的 `goroutine` 操作，但是这样做又会引发性能问题。

使用 `sync.Once` 改造的示例代码如下：

```
var icons map[string]image.Image

var loadIconsOnce sync.Once

func loadIcons() {
    icons = map[string]image.Image{
        "left":  loadIcon("left.png"),
        "up":    loadIcon("up.png"),
        "right": loadIcon("right.png"),
        "down":  loadIcon("down.png"),
    }
}

// Icon 是并发安全的
func Icon(name string) image.Image {
    loadIconsOnce.Do(loadIcons)
    return icons[name]
}
```

并发安全的单例模式

下面是借助 `sync.Once` 实现的并发安全的单例模式：

```
package singleton

import (
    "sync"
)

type singleton struct {}

var instance *singleton
var once sync.Once

func GetInstance() *singleton {
    once.Do(func() {
        instance = &singleton{}
    })
}
```

```
        }
    return instance
}
```

`sync.Once` 其实内部包含一个互斥锁和一个布尔值，互斥锁保证布尔值和数据的安全，而布尔值用来记录初始化是否完成。这样设计就能保证初始化操作的时候是并发安全的并且初始化操作也不会被执行多次。

sync.Map

Go语言中内置的map不是并发安全的。请看下面的示例：

```
var m = make(map[string]int)

func get(key string) int {
    return m[key]
}

func set(key string, value int) {
    m[key] = value
}

func main() {
    wg := sync.WaitGroup{}
    for i := 0; i < 20; i++ {
        wg.Add(1)
        go func(n int) {
            key := strconv.Itoa(n)
            set(key, n)
            fmt.Printf("k=%v, v=%v\n", key, get(key))
            wg.Done()
        }(i)
    }
    wg.Wait()
}
```

上面的代码开启少量几个 `goroutine` 的时候可能没什么问题，当并发多了之后执行上面的代码就会报 `fatal error: concurrent map writes` 错误。

像这种场景下就需要为map加锁来保证并发的安全性了，Go语言的 `sync` 包中提供了一个开箱即用的并发安全版map-`sync.Map`。开箱即用表示不用像内置的map一样使用make函数初始化就能直接使用。同时 `sync.Map` 内置了诸如 `Store`、`Load`、`LoadOrStore`、`Delete`、`Range` 等操作方法。

```
var m = sync.Map{}

func main() {
    wg := sync.WaitGroup{}
    for i := 0; i < 20; i++ {
        wg.Add(1)
        go func(n int) {
            key := strconv.Itoa(n)
            m.Store(key, n)
            value, _ := m.Load(key)
            fmt.Printf("k=%v, v=%v\n", key, value)
        }(i)
    }
    wg.Wait()
}
```

```

        wg.Done()
    }(i)
}
wg.Wait()
}

```

原子操作

在上面的代码中的我们通过锁操作来实现同步。而锁机制的底层是基于原子操作的，其一般直接通过CPU指令实现。Go语言中原子操作由内置的标准库 `sync/atomic` 提供。

atomic包

方法	解释
<pre>func LoadInt32(addr *int32) (val int32) func LoadInt64(addr *int64) (val int64) func LoadUint32(addr *uint32) (val uint32) func LoadUint64(addr *uint64) (val uint64) func LoadUintptr(addr *uintptr) (val uintptr) func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)</pre>	读取操作
<pre>func StoreInt32(addr *int32, val int32) func StoreInt64(addr *int64, val int64) func StoreUint32(addr *uint32, val uint32) func StoreUint64(addr *uint64, val uint64) func StoreUintptr(addr *uintptr, val uintptr) func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)</pre>	写入操作
<pre>func AddInt32(addr *int32, delta int32) (new int32) func AddInt64(addr *int64, delta int64) (new int64) func AddUint32(addr *uint32, delta uint32) (new uint32) func AddUint64(addr *uint64, delta uint64) (new uint64) func AddUintptr(addr *uintptr, delta uintptr) (new uintptr)</pre>	修改操作
<pre>func SwapInt32(addr *int32, new int32) (old int32) func SwapInt64(addr *int64, new int64) (old int64) func SwapUint32(addr *uint32, new uint32) (old uint32) func SwapUint64(addr *uint64, new uint64) (old uint64) func SwapUintptr(addr *uintptr, new uintptr) (old uintptr) func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)</pre>	交换操作
<pre>func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool) func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool) func CompareAndSwapUint32(addr *uint32, old, new uint32) (swapped bool) func CompareAndSwapUint64(addr *uint64, old, new uint64) (swapped bool) func CompareAndSwapUintptr(addr *uintptr, old, new uintptr) (swapped bool) func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)</pre>	比较并交换操作

示例

我们填写一个示例来比较下互斥锁和原子操作的性能。

```

package main

import (
    "fmt"
    "sync"
    "sync/atomic"
    "time"
)

type Counter interface {
    Inc()
}

```

```
Load() int64
}

// 普通版
type CommonCounter struct {
    counter int64
}

func (c CommonCounter) Inc() {
    c.counter++
}

func (c CommonCounter) Load() int64 {
    return c.counter
}

// 互斥锁版
type MutexCounter struct {
    counter int64
    lock    sync.Mutex
}

func (m *MutexCounter) Inc() {
    m.lock.Lock()
    defer m.lock.Unlock()
    m.counter++
}

func (m *MutexCounter) Load() int64 {
    m.lock.Lock()
    defer m.lock.Unlock()
    return m.counter
}

// 原子操作版
type AtomicCounter struct {
    counter int64
}

func (a *AtomicCounter) Inc() {
    atomic.AddInt64(&a.counter, 1)
}

func (a *AtomicCounter) Load() int64 {
    return atomic.LoadInt64(&a.counter)
}

func test(c Counter) {
    var wg sync.WaitGroup
    start := time.Now()
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            c.Inc()
            wg.Done()
        }()
    }
    wg.Wait()
}
```

```

    end := time.Now()
    fmt.Println(c.Load(), end.Sub(start))
}

func main() {
    c1 := CommonCounter{} // 非并发安全
    test(c1)
    c2 := MutexCounter{} // 使用互斥锁实现并发安全
    test(&c2)
    c3 := AtomicCounter{} // 并发安全且比互斥锁效率更高
    test(&c3)
}

```

`atomic` 包提供了底层的原子级内存操作，对于同步算法的实现很有用。这些函数必须谨慎地保证正确使用。除了某些特殊的底层应用，使用通道或者sync包的函数/类型实现同步更好。

Go语言实现TCP通信

TCP协议

TCP/IP(Transmission Control Protocol/Internet Protocol) 即传输控制协议/网间协议，是一种面向连接（连接导向）的、可靠的、基于字节流的传输层（Transport layer）通信协议，因为是面向连接的协议，数据像水流一样传输，会存在黏包问题。

TCP服务端

一个TCP服务端可以同时连接很多个客户端，例如世界各地的用户使用自己电脑上的浏览器访问淘宝网。因为Go语言中创建多个goroutine实现并发非常方便和高效，所以我们可以每建立一次链接就创建一个goroutine去处理。

TCP服务端程序的处理流程：

1. 监听端口
2. 接收客户端请求建立链接
3. 创建goroutine处理链接。

我们使用Go语言的net包实现的TCP服务端代码如下：

```

// tcp/server/main.go

// TCP server端

// 处理函数
func process(conn net.Conn) {
    defer conn.Close() // 关闭连接
    for {
        reader := bufio.NewReader(conn)

```

```

var buf [128]byte
n, err := reader.Read(buf[:]) // 读取数据
if err != nil {
    fmt.Println("read from client failed, err:", err)
    break
}
recvStr := string(buf[:n])
fmt.Println("收到client端发来的数据: ", recvStr)
conn.Write([]byte(recvStr)) // 发送数据
}

func main() {
    listen, err := net.Listen("tcp", "127.0.0.1:20000")
    if err != nil {
        fmt.Println("listen failed, err:", err)
        return
    }
    for {
        conn, err := listen.Accept() // 建立连接
        if err != nil {
            fmt.Println("accept failed, err:", err)
            continue
        }
        go process(conn) // 启动一个goroutine处理连接
    }
}

```

将上面的代码保存之后编译成 `server` 或 `server.exe` 可执行文件。

TCP客户端

一个TCP客户端进行TCP通信的流程如下：

1. 建立与服务端的链接
2. 进行数据收发
3. 关闭链接

使用Go语言的net包实现的TCP客户端代码如下：

```

// tcp/client/main.go

// 客户端
func main() {
    conn, err := net.Dial("tcp", "127.0.0.1:20000")
    if err != nil {
        fmt.Println("err :", err)
        return
    }
    defer conn.Close() // 关闭连接
    inputReader := bufio.NewReader(os.Stdin)
    for {
        input, _ := inputReader.ReadString('\n') // 读取用户输入
        inputInfo := strings.Trim(input, "\r\n")
    }
}

```

```

if strings.ToUpper(inputInfo) == "Q" { // 如果输入q就退出
    return
}
_, err = conn.Write([]byte(inputInfo)) // 发送数据
if err != nil {
    return
}
buf := [512]byte{}
n, err := conn.Read(buf[:])
if err != nil {
    fmt.Println("recv failed, err:", err)
    return
}
fmt.Println(string(buf[:n]))
}
}

```

将上面的代码编译成 `client` 或 `client.exe` 可执行文件，先启动server端再启动client端，在client端输入任意内容回车之后就能在server端看到client端发送的数据，从而实现TCP通信。

GO语言中TCP黏包问题的解决

黏包示例

服务端代码如下：

```

// socket_stick/server/main.go

func process(conn net.Conn) {
    defer conn.Close()
    reader := bufio.NewReader(conn)
    var buf [1024]byte
    for {
        n, err := reader.Read(buf[:])
        if err == io.EOF {
            break
        }
        if err != nil {
            fmt.Println("read from client failed, err:", err)
            break
        }
        recvStr := string(buf[:n])
        fmt.Println("收到client发来的数据：", recvStr)
    }
}

func main() {

    listen, err := net.Listen("tcp", "127.0.0.1:30000")
    if err != nil {

```

```

        fmt.Println("listen failed, err:", err)
        return
    }
    defer listen.Close()
    for {
        conn, err := listen.Accept()
        if err != nil {
            fmt.Println("accept failed, err:", err)
            continue
        }
        go process(conn)
    }
}

```

客户端代码如下：

```

// socket_stick/client/main.go

func main() {
    conn, err := net.Dial("tcp", "127.0.0.1:30000")
    if err != nil {
        fmt.Println("dial failed, err", err)
        return
    }
    defer conn.Close()
    for i := 0; i < 20; i++ {
        msg := `Hello, Hello. How are you?`
        conn.Write([]byte(msg))
    }
}

```

将上面的代码保存后，分别编译。先启动服务端再启动客户端，可以看到服务端输出结果如下：

```

收到client发来的数据: Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How
are you?Hello, Hello. How are you?Hello, Hello. How are you?
收到client发来的数据: Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How
are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello,
Hello. How are you?Hello, Hello. How are you?
收到client发来的数据: Hello, Hello. How are you?Hello, Hello. How are you?
收到client发来的数据: Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How
are you?
收到client发来的数据: Hello, Hello. How are you?Hello, Hello. How are you?

```

客户端分10次发送的数据，在服务端并没有成功的输出10次，而是多条数据“粘”到了一起。

为什么会出现粘包

主要原因就是tcp数据传递模式是流模式，在保持长连接的时候可以进行多次的收和发。

“粘包”可发生在发送端也可发生在接收端：

- 由Nagle算法造成的发送端的粘包：Nagle算法是一种改善网络传输效率的算法。简单来说就是当我们提交一段数据给TCP发送时，TCP并不立刻发送此段数据，而是等待一小段时间看看在等待期间是否还有要发送

的数据，若有则会一次把这两段数据发送出去。

2. 接收端接收不及时造成的接收端粘包：TCP会把接收到的数据存在自己的缓冲区中，然后通知应用层取数据。当应用层由于某些原因不能及时的把TCP的数据取出来，就会造成TCP缓冲区中存放了几段数据。

解决办法

出现“粘包”的关键在于接收方不确定将要传输的数据包的大小，因此我们可以对数据包进行封包和拆包的操作。

封包：封包就是给一段数据加上包头，这样一来数据包就分为包头和包体两部分内容了(过滤非法包时封包会加入“包尾”内容)。包头部分的长度是固定的，并且它存储了包体的长度，根据包头长度固定以及包头中含有包体长度的变量就能正确的拆分出一个完整的数据包。

我们可以自己定义一个协议，比如数据包的前4个字节为包头，里面存储的是发送的数据的长度。

```
// socket_stick/proto/proto.go
package proto

import (
    "bufio"
    "bytes"
    "encoding/binary"
)

// Encode 将消息编码
func Encode(message string) ([]byte, error) {
    // 读取消息的长度，转换成int32类型（占4个字节）
    var length = int32(len(message))
    var pkg = new(bytes.Buffer)
    // 写入消息头
    err := binary.Write(pkg, binary.LittleEndian, length)
    if err != nil {
        return nil, err
    }
    // 写入消息实体
    err = binary.Write(pkg, binary.LittleEndian, []byte(message))
    if err != nil {
        return nil, err
    }
    return pkg.Bytes(), nil
}

// Decode 解码消息
func Decode(reader *bufio.Reader) (string, error) {
    // 读取消息的长度
    lengthByte, _ := reader.Peek(4) // 读取前4个字节的数据
    lengthBuff := bytes.NewBuffer(lengthByte)
    var length int32
    err := binary.Read(lengthBuff, binary.LittleEndian, &length)
    if err != nil {
        return "", err
    }
    // Buffered返回缓冲中现有的可读取的字节数。
    if int32(reader.Buffered()) < length+4 {
        return "", err
    }
    message := make([]byte, length)
    _, err = reader.Read(message)
    if err != nil {
        return "", err
    }
    return string(message), nil
}
```

```

}

// 读取真正的消息数据
pack := make([]byte, int(4+length))
_, err = reader.Read(pack)
if err != nil {
    return "", err
}
return string(pack[4:]), nil
}

```

接下来在服务端和客户端分别使用上面定义的 `proto` 包的 `Decode` 和 `Encode` 函数处理数据。

服务端代码如下：

```

// socket_stick/server2/main.go

func process(conn net.Conn) {
    defer conn.Close()
    reader := bufio.NewReader(conn)
    for {
        msg, err := proto.Decode(reader)
        if err == io.EOF {
            return
        }
        if err != nil {
            fmt.Println("decode msg failed, err:", err)
            return
        }
        fmt.Println("收到client发来的数据：", msg)
    }
}

func main() {

    listen, err := net.Listen("tcp", "127.0.0.1:30000")
    if err != nil {
        fmt.Println("listen failed, err:", err)
        return
    }
    defer listen.Close()
    for {
        conn, err := listen.Accept()
        if err != nil {
            fmt.Println("accept failed, err:", err)
            continue
        }
        go process(conn)
    }
}

```

客户端代码如下：

```

// socket_stick/client2/main.go

func main() {

```

```
conn, err := net.Dial("tcp", "127.0.0.1:30000")
if err != nil {
    fmt.Println("dial failed, err:", err)
    return
}
defer conn.Close()
for i := 0; i < 20; i++ {
    msg := `Hello, Hello. How are you?`
    data, err := proto.Encode(msg)
    if err != nil {
        fmt.Println("encode msg failed, err:", err)
        return
    }
    conn.Write(data)
}
}
```

Go语言实现UDP通信

UDP协议

UDP协议（User Datagram Protocol）中文名称是用户数据报协议，是OSI（Open System Interconnection，开放式系统互联）参考模型中一种无连接的传输层协议，不需要建立连接就能直接进行数据发送和接收，属于不可靠的、没有时序的通信，但是UDP协议的实时性比较好，通常用于视频直播相关领域。

UDP服务端

使用Go语言的 `net` 包实现的UDP服务端代码如下：

```
// UDP/server/main.go

// UDP server端
func main() {
    listen, err := net.ListenUDP("udp", &net.UDPAddr{
        IP:   net.IPv4(0, 0, 0, 0),
        Port: 30000,
    })
    if err != nil {
        fmt.Println("listen failed, err:", err)
        return
    }
    defer listen.Close()
    for {
        var data [1024]byte
        n, addr, err := listen.ReadFromUDP(data[:]) // 接收数据
        if err != nil {
            fmt.Println("read udp failed, err:", err)
            continue
        }
        fmt.Printf("data:%v addr:%v count:%v\n", string(data[:n]), addr, n)
    }
}
```

```
_ , err = listen.WriteToUDP(data[:n] , addr) // 发送数据
if err != nil {
    fmt.Println("write to udp failed, err:", err)
    continue
}
}
```

UDP客户端

使用Go语言的 `net` 包实现的UDP客户端代码如下：

```
// UDP 客户端
func main() {
    socket, err := net.DialUDP("udp", nil, &net.UDPAddr{
        IP:   net.IPv4(0, 0, 0, 0),
        Port: 30000,
    })
    if err != nil {
        fmt.Println("连接服务端失败, err:", err)
        return
    }
    defer socket.Close()
    sendData := []byte("Hello server")
    _, err = socket.Write(sendData) // 发送数据
    if err != nil {
        fmt.Println("发送数据失败, err:", err)
        return
    }
    data := make([]byte, 4096)
    n, remoteAddr, err := socket.ReadFromUDP(data) // 接收数据
    if err != nil {
        fmt.Println("接收数据失败, err:", err)
        return
    }
    fmt.Printf("recv:%v addr:%v count:%v\n", string(data[:n]), remoteAddr, n)
}
```

GO语言的单元测试

go test工具

Go语言中的测试依赖 `go test` 命令。编写测试代码和编写普通的Go代码过程是类似的，并不需要学习新的语法、规则或工具。

`go test`命令是一个按照一定约定和组织的测试代码的驱动程序。在包目录内，所有以 `_test.go` 为后缀名的源代码文件都是 `go test` 测试的一部分，不会被 `go build` 编译到最终的可执行文件中。

在 `*_test.go` 文件中有三种类型的函数，单元测试函数、基准测试函数和示例函数。

类型	格式	作用
测试函数	函数名前缀为Test	测试程序的一些逻辑行为是否正确
基准函数	函数名前缀为Benchmark	测试函数的性能
示例函数	函数名前缀为Example	为文档提供示例文档

`go test` 命令会遍历所有的 `*_test.go` 文件中符合上述命名规则的函数，然后生成一个临时的main包用于调用相应的测试函数，然后构建并运行、报告测试结果，最后清理测试中生成的临时文件。

测试函数

测试函数的格式

每个测试函数必须导入 `testing` 包，测试函数的基本格式（签名）如下：

```
func TestName(t *testing.T){
    // ...
}
```

测试函数的名字必须以 `Test` 开头，可选的后缀名必须以大写字母开头，举几个例子：

```
func TestAdd(t *testing.T){ ... }
func TestSum(t *testing.T){ ... }
func TestLog(t *testing.T){ ... }
```

其中参数 `t` 用于报告测试失败和附加的日志信息。`testing.T` 的拥有的方法如下：

```
func (c *T) Error(args ...interface{})
func (c *T) Errorf(format string, args ...interface{})
func (c *T) Fail()
func (c *T) FailNow()
func (c *T) Failed() bool
func (c *T) Fatal(args ...interface{})
func (c *T) Fatalf(format string, args ...interface{})
func (c *T) Log(args ...interface{})
func (c *T) Logf(format string, args ...interface{})
func (c *T) Name() string
func (t *T) Parallel()
func (t *T) Run(name string, f func(t *T)) bool
func (c *T) Skip(args ...interface{})
func (c *T) SkipNow()
func (c *T) Skipf(format string, args ...interface{})
func (c *T) Skipped() bool
```

测试函数示例

就像细胞是构成我们身体的基本单位，一个软件程序也是由很多单元组件构成的。单元组件可以是函数、结构体、方法和最终用户可能依赖的任意东西。总之我们需要确保这些组件是能够正常运行的。单元测试是一些利用各种方法测试单元组件的程序，它会将结果与预期输出进行比较。

接下来，我们定义一个 `split` 的包，包中定义了一个 `Split` 函数，具体实现如下：

```
// split/split.go

package split

import "strings"

// split package with a single split function.

// Split slices s into all substrings separated by sep and
// returns a slice of the substrings between those separators.
func Split(s, sep string) (result []string) {
    i := strings.Index(s, sep)

    for i > -1 {
        result = append(result, s[:i])
        s = s[i+1:]
        i = strings.Index(s, sep)
    }
    result = append(result, s)
    return
}
```

在当前目录下，我们创建一个 `split_test.go` 的测试文件，并定义一个测试函数如下：

```
// split/split_test.go

package split

import (
    "reflect"
    "testing"
)

func TestSplit(t *testing.T) { // 测试函数名必须以Test开头，必须接收一个*testing.T类型参数
    got := Split("a:b:c", ":")           // 程序输出的结果
    want := []string{"a", "b", "c"}       // 期望的结果
    if !reflect.DeepEqual(want, got) { // 因为slice不能比较直接，借助反射包中的方法比较
        t.Errorf("expected:%v, got:%v", want, got) // 测试失败输出错误提示
    }
}
```

此时 `split` 这个包中的文件如下：

```
split $ ls -l
total 16
-rw-r--r--  1 liwenzhou  staff  408  4 29 15:50 split.go
-rw-r--r--  1 liwenzhou  staff  466  4 29 16:04 split_test.go
```

在 `split` 包路径下，执行 `go test` 命令，可以看到输出结果如下：

```
split $ go test
PASS
ok      github.com/Q1mi/studygo/code_demo/test_demo/split      0.005s
```

一个测试用例有点单薄，我们再编写一个测试使用多个字符切割字符串的例子，在 `split_test.go` 中添加如下测试函数：

```
func TestMoreSplit(t *testing.T) {
    got := Split("abcd", "bc")
    want := []string{"a", "d"}
    if !reflect.DeepEqual(want, got) {
        t.Errorf("expected:%v, got:%v", want, got)
    }
}
```

再次运行 `go test` 命令，输出结果如下：

```
split $ go test
--- FAIL: TestMultiSplit (0.00s)
    split_test.go:20: expected:[a d], got:[a cd]
FAIL
exit status 1
FAIL    github.com/Q1mi/studygo/code_demo/test_demo/split      0.006s
```

这一次，我们的测试失败了。我们可以为 `go test` 命令添加 `-v` 参数，查看测试函数名称和运行时间：

```
split $ go test -v
== RUN TestSplit
--- PASS: TestSplit (0.00s)
== RUN TestMoreSplit
--- FAIL: TestMoreSplit (0.00s)
    split_test.go:21: expected:[a d], got:[a cd]
FAIL
exit status 1
FAIL    github.com/Q1mi/studygo/code_demo/test_demo/split      0.005s
```

这一次我们能清楚的看到是 `TestMoreSplit` 这个测试没有成功。还可以在 `go test` 命令后添加 `-run` 参数，它对应一个正则表达式，只有函数名匹配上的测试函数才会被 `go test` 命令执行。

```
split $ go test -v -run="More"
== RUN TestMoreSplit
--- FAIL: TestMoreSplit (0.00s)
    split_test.go:21: expected:[a d], got:[a cd]
FAIL
exit status 1
FAIL    github.com/Q1mi/studygo/code_demo/test_demo/split      0.006s
```

现在我们回过头来解决我们程序中的问题。很显然我们最初的 `split` 函数并没有考虑到 `sep` 为多个字符的情况，我们来修复下这个 Bug：

```
package split

import "strings"

// split package with a single split function.

// Split slices s into all substrings separated by sep and
```

```
// returns a slice of the substrings between those separators.
func Split(s, sep string) ([]string) {
    i := strings.Index(s, sep)

    for i > -1 {
        result = append(result, s[:i])
        s = s[i+len(sep):] // 这里使用len(sep)获取sep的长度
        i = strings.Index(s, sep)
    }
    result = append(result, s)
    return
}
```

这一次我们再来测试一下，我们的程序。注意，当我们修改了我们的代码之后不要仅仅执行那些失败的测试函数，我们应该完整的运行所有的测试，保证不会因为修改代码而引入了新的问题。

```
split $ go test -v
===[ RUN TestSplit
--- PASS: TestSplit (0.00s)
===[ RUN TestMoreSplit
--- PASS: TestMoreSplit (0.00s)
PASS
ok      github.com/Q1mi/studygo/code_demo/test_demo/split      0.006s
```

这一次我们的测试都通过了。

测试组

我们现在还想要测试一下 `split` 函数对中文字符串的支持，这个时候我们可以再编写一个 `TestChineseSplit` 测试函数，但是我们也可以使用如下更友好的一种方式来添加更多的测试用例。

```
func TestSplit(t *testing.T) {
    // 定义一个测试用例类型
    type test struct {
        input string
        sep   string
        want  []string
    }
    // 定义一个存储测试用例的切片
    tests := []test{
        {input: "a:b:c", sep: ":", want: []string{"a", "b", "c"}},
        {input: "a:b:c", sep: ",", want: []string{"a:b:c"}},
        {input: "abcd", sep: "bc", want: []string{"a", "d"}},
        {input: "沙河有沙又有河", sep: "沙", want: []string{"河有", "又有河"}},
    }
    // 遍历切片，逐一执行测试用例
    for _, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(got, tc.want) {
            t.Errorf("expected:%v, got:%v", tc.want, got)
        }
    }
}
```

我们通过上面的代码把多个测试用例合到一起，再次执行 `go test` 命令。

```
split $ go test -v
== RUN TestSplit
--- FAIL: TestSplit (0.00s)
    split_test.go:42: expected:[河有 又有河], got:[ 河有 又有河]
FAIL
exit status 1
FAIL    github.com/Q1mi/studygo/code_demo/test_demo/split      0.006s
```

我们的测试出现了问题，仔细看打印的测试失败提示信息：`expected:[河有 又有河]`，`got:[河有 又有河]`，你会发现 `[河有 又有河]` 中有个不明显的空串，这种情况下十分推荐使用 `%#v` 的格式化方式。

我们修改下测试用例的格式化输出错误提示部分：

```
func TestSplit(t *testing.T) {
    ...
    for _, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(got, tc.want) {
            t.Errorf("expected:%#v, got:%#v", tc.want, got)
        }
    }
}
```

此时运行 `go test` 命令后就能看到比较明显的提示信息了：

```
split $ go test -v
== RUN TestSplit
--- FAIL: TestSplit (0.00s)
    split_test.go:42: expected:[]string{"河有", "又有河"}, got:[]string{"", "河有", "又有河"}
FAIL
exit status 1
FAIL    github.com/Q1mi/studygo/code_demo/test_demo/split      0.006s
```

子测试

看起来都挺不错的，但是如果测试用例比较多的时候，我们是没办法一眼看出来具体是哪个测试用例失败了。我们可能会想到下面的解决办法：

```
func TestSplit(t *testing.T) {
    type test struct { // 定义test结构体
        input string
        sep   string
        want  []string
    }
    tests := map[string]test{ // 测试用例使用map存储
        "simple":     {input: "a:b:c", sep: ":", want: []string{"a", "b", "c"}},
        "wrong sep":  {input: "a:b:c", sep: ",", want: []string{"a:b:c"}},
        "more sep":   {input: "abcd", sep: "bc", want: []string{"a", "d"}},
        "leading sep": {input: "沙河有沙又有河", sep: "沙", want: []string{"河有", "又有河"}},
    }
    for name, tc := range tests {
```

```

got := Split(tc.input, tc.sep)
if !reflect.DeepEqual(got, tc.want) {
    t.Errorf("name:%s expected:%#v, got:%#v", name, tc.want, got) // 将测试用例的name格式化输出
}
}
}
}

```

上面的做法是能够解决问题的。同时Go1.7+中新增了子测试，我们可以按照如下方式使用 `t.Run` 执行子测试：

```

func TestSplit(t *testing.T) {
    type test struct { // 定义test结构体
        input string
        sep   string
        want  []string
    }
    tests := map[string]test{ // 测试用例使用map存储
        "simple":     {input: "a:b:c", sep: ":"}, want: []string{"a", "b", "c"}},
        "wrong_sep":  {input: "a:b:c", sep: ","}, want: []string{"a:b:c"}},
        "more_sep":   {input: "abcd", sep: "bc"}, want: []string{"a", "d"}},
        "leading_sep": {input: "沙河有沙又有河", sep: "沙"}, want: []string{"河有", "又有河"}},
    }
    for name, tc := range tests {
        t.Run(name, func(t *testing.T) { // 使用t.Run()执行子测试
            got := Split(tc.input, tc.sep)
            if !reflect.DeepEqual(got, tc.want) {
                t.Errorf("expected:%#v, got:%#v", tc.want, got)
            }
        })
    }
}

```

此时我们再执行 `go test` 命令就能够看到更清晰的输出内容了：

```

split $ go test -v
===[ RUN   TestSplit
===[ RUN   TestSplit/leading_sep
===[ RUN   TestSplit/simple
===[ RUN   TestSplit/wrong_sep
===[ RUN   TestSplit/more_sep
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/leading_sep (0.00s)
        split_test.go:83: expected:[]string{"河有", "又有河"}, got:[]string{"", "河有", "又有河"}
    --- PASS: TestSplit/simple (0.00s)
    --- PASS: TestSplit/wrong_sep (0.00s)
    --- PASS: TestSplit/more_sep (0.00s)
FAIL
exit status 1
FAIL    github.com/Q1mi/studygo/code_demo/test_demo/split          0.006s

```

这个时候我们要把测试用例中的错误修改回来：

```
func TestSplit(t *testing.T) {
    ...
    tests := map[string]test{ // 测试用例使用map存储
        "simple": {input: "a:b:c", sep: ":", want: []string{"a", "b", "c"}},
        "wrong sep": {input: "a:b:c", sep: ",", want: []string{"a:b:c"}},
        "more sep": {input: "abcd", sep: "bc", want: []string{"a", "d"}},
        "leading sep": {input: "沙河有沙又有河", sep: "沙", want: []string{"", "河有", "又有河"}},
    }
    ...
}
```

我们都知道可以通过 `-run=RegExp` 来指定运行的测试用例，还可以通过 `/` 来指定要运行的子测试用例，例如：`go test -v -run=Split/simple` 只会运行 `simple` 对应的子测试用例。

测试覆盖率

测试覆盖率是你的代码被测试套件覆盖的百分比。通常我们使用的都是语句的覆盖率，也就是在测试中至少被运行一次的代码占总代码的比例。

Go提供内置功能来检查你的代码覆盖率。我们可以使用 `go test -cover` 来查看测试覆盖率。例如：

```
split $ go test -cover
PASS
coverage: 100.0% of statements
ok      github.com/Q1mi/studygo/code_demo/test_demo/split      0.005s
```

从上面的结果可以看到我们的测试用例覆盖了100%的代码。

Go还提供了一个额外的 `-coverprofile` 参数，用来将覆盖率相关的记录信息输出到一个文件。例如：

```
split $ go test -cover -coverprofile=c.out
PASS
coverage: 100.0% of statements
ok      github.com/Q1mi/studygo/code_demo/test_demo/split      0.005s
```

上面的命令会将覆盖率相关的信息输出到当前文件夹下面的 `c.out` 文件中，然后我们执行 `go tool cover -`

`html=c.out`，使用 `cover` 工具来处理生成的记录信息，该命令会打开本地的浏览器窗口生成一个HTML报告。

```
package split

import "strings"

// split package with a single split function.

// Split slices s into all substrings separated by sep and
// returns a slice of the substrings between those separators.
func Split(s, sep string) (result []string) {
    i := strings.Index(s, sep)

    for i > -1 {
        result = append(result, s[:i])
        s = s[i+len(sep):] // 这里使用len(sep)获取sep的长度
        i = strings.Index(s, sep)
    }
    result = append(result, s)
    return
}
```

上图中每个用绿色标记的语句块表示被覆盖了，而红色的表示没有被覆盖。

基准测试

基准测试函数格式

基准测试就是在一定工作负载之下检测程序性能的一种方法。基准测试的基本格式如下：

```
func BenchmarkName(b *testing.B){
    // ...
}
```

基准测试以 `Benchmark` 为前缀，需要一个 `*testing.B` 类型的参数 `b`，基准测试必须要执行 `b.N` 次，这样的测试才有对照性，`b.N` 的值是系统根据实际情况去调整的，从而保证测试的稳定性。`testing.B` 拥有的方法如下：

```
func (c *B) Error(args ...interface{})
func (c *B) Errorf(format string, args ...interface{})
func (c *B) Fail()
func (c *B) FailNow()
func (c *B) Failed() bool
func (c *B) Fatal(args ...interface{})
func (c *B) Fatalf(format string, args ...interface{})
func (c *B) Log(args ...interface{})
func (c *B) Logf(format string, args ...interface{})
func (c *B) Name() string
func (b *B) ReportAllocs()
func (b *B) ResetTimer()
func (b *B) Run(name string, f func(b *B)) bool
```

```
func (b *B) RunParallel(body func(*PB))
func (b *B) SetBytes(n int64)
func (b *B) SetParallelism(p int)
func (c *B) Skip(args ...interface{})
func (c *B) SkipNow()
func (c *B) Skipf(format string, args ...interface{})
func (c *B) Skipped() bool
func (b *B) StartTimer()
func (b *B) StopTimer()
```

基准测试示例

我们为split包中的 `Split` 函数编写基准测试如下：

```
func BenchmarkSplit(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Split("沙河有沙又有河", "沙")
    }
}
```

基准测试并不会默认执行，需要增加 `-bench` 参数，所以我们通过执行 `go test -bench=Split` 命令执行基准测试，输出结果如下：

```
split $ go test -bench=Split
goos: darwin
goarch: amd64
pkg: github.com/Q1mi/studygo/code_demo/test_demo/split
BenchmarkSplit-8      10000000          203 ns/op
PASS
ok      github.com/Q1mi/studygo/code_demo/test_demo/split      2.255s
```

其中 `BenchmarkSplit-8` 表示对 `Split` 函数进行基准测试，数字 `8` 表示 `GOMAXPROCS` 的值，这个对于并发基准测试很重要。`10000000` 和 `203ns/op` 表示每次调用 `Split` 函数耗时 `203ns`，这个结果是 `10000000` 次调用的平均值。

我们还可以为基准测试添加 `-benchmem` 参数，来获得内存分配的统计数据。

```
split $ go test -bench=Split -benchmem
goos: darwin
goarch: amd64
pkg: github.com/Q1mi/studygo/code_demo/test_demo/split
BenchmarkSplit-8      10000000          215 ns/op          112 B/op       3
allocs/op
PASS
ok      github.com/Q1mi/studygo/code_demo/test_demo/split      2.394s
```

其中，`112 B/op` 表示每次操作内存分配了 112 字节，`3 allocs/op` 则表示每次操作进行了 3 次内存分配。我们将我们的 `Split` 函数优化如下：

```

func Split(s, sep string) (result []string) {
    result = make([]string, 0, strings.Count(s, sep)+1)
    i := strings.Index(s, sep)
    for i > -1 {
        result = append(result, s[:i])
        s = s[i+len(sep):] // 这里使用len(sep)获取sep的长度
        i = strings.Index(s, sep)
    }
    result = append(result, s)
    return
}

```

这一次我们提前使用make函数将result初始化为一个容量足够大的切片，而不再像之前一样通过调用append函数来追加。我们来看一下这个改进会带来多大的性能提升：

```

split $ go test -bench=Split -benchmem
goos: darwin
goarch: amd64
pkg: github.com/Q1mi/studygo/code_demo/test_demo/split
BenchmarkSplit-8          100000000           127 ns/op      48 B/op       1
allocs/op
PASS
ok      github.com/Q1mi/studygo/code_demo/test_demo/split      1.423s

```

这个使用make函数提前分配内存的改动，减少了2/3的内存分配次数，并且减少了一半的内存分配。

性能比较函数

上面的基准测试只能得到给定操作的绝对耗时，但是在很多性能问题是发生在两个不同操作之间的相对耗时，比如同一个函数处理1000个元素的耗时与处理1万甚至100万个元素的耗时的差别是多少？再或者对于同一个任务究竟使用哪种算法性能最佳？我们通常需要对两个不同算法的实现使用相同的输入来进行基准比较测试。

性能比较函数通常是一个带有参数的函数，被多个不同的Benchmark函数传入不同的值来调用。举个例子如下：

```

func benchmark(b *testing.B, size int){/* ... */}
func Benchmark10(b *testing.B){ benchmark(b, 10) }
func Benchmark100(b *testing.B){ benchmark(b, 100) }
func Benchmark1000(b *testing.B){ benchmark(b, 1000) }

```

例如我们编写了一个计算斐波那契数列的函数如下：

```

// fib.go

// Fib 是一个计算第n个斐波那契数的函数
func Fib(n int) int {
    if n < 2 {
        return n
    }
    return Fib(n-1) + Fib(n-2)
}

```

我们编写的性能比较函数如下：

```
// fib_test.go

func benchmarkFib(b *testing.B, n int) {
    for i := 0; i < b.N; i++ {
        Fib(n)
    }
}

func BenchmarkFib1(b *testing.B) { benchmarkFib(b, 1) }
func BenchmarkFib2(b *testing.B) { benchmarkFib(b, 2) }
func BenchmarkFib3(b *testing.B) { benchmarkFib(b, 3) }
func BenchmarkFib10(b *testing.B) { benchmarkFib(b, 10) }
func BenchmarkFib20(b *testing.B) { benchmarkFib(b, 20) }
func BenchmarkFib40(b *testing.B) { benchmarkFib(b, 40) }
```

运行基准测试：

```
split $ go test -bench=.
goos: darwin
goarch: amd64
pkg: github.com/Q1mi/studygo/code_demo/test_demo/fib
BenchmarkFib1-8      1000000000          2.03 ns/op
BenchmarkFib2-8      300000000          5.39 ns/op
BenchmarkFib3-8      200000000          9.71 ns/op
BenchmarkFib10-8     5000000          325 ns/op
BenchmarkFib20-8     30000          42460 ns/op
BenchmarkFib40-8     2          638524980 ns/op
PASS
ok      github.com/Q1mi/studygo/code_demo/test_demo/fib 12.944s
```

这里需要注意的是，默认情况下，每个基准测试至少运行1秒。如果在Benchmark函数返回时没有到1秒，则b.N的值会按1,2,5,10,20,50, ...增加，并且函数再次运行。

最终的BenchmarkFib40只运行了两次，每次运行的平均值只有不到一秒。像这种情况下我们应该可以使用 `-benchtime` 标志增加最小基准时间，以产生更准确的结果。例如：

```
split $ go test -bench=Fib40 -benchtime=20s
goos: darwin
goarch: amd64
pkg: github.com/Q1mi/studygo/code_demo/test_demo/fib
BenchmarkFib40-8      50          663205114 ns/op
PASS
ok      github.com/Q1mi/studygo/code_demo/test_demo/fib 33.849s
```

这一次 `BenchmarkFib40` 函数运行了50次，结果就会更准确一些了。

使用性能比较函数做测试的时候一个容易犯的错误就是把 `b.N` 作为输入的大小，例如以下两个例子都是错误的示范：

```
// 错误示范1
func BenchmarkFibWrong(b *testing.B) {
    for n := 0; n < b.N; n++ {
        Fib(n)
    }
}

// 错误示范2
func BenchmarkFibWrong2(b *testing.B) {
    Fib(b.N)
}
```

重置时间

`b.ResetTimer` 之前的处理不会放到执行时间里，也不会输出到报告中，所以可以在之前做一些不计划作为测试报告的操作。例如：

```
func BenchmarkSplit(b *testing.B) {
    time.Sleep(5 * time.Second) // 假设需要做一些耗时的无关操作
    b.ResetTimer()             // 重置计时器
    for i := 0; i < b.N; i++ {
        Split("沙河有沙又有河", "沙")
    }
}
```

并行测试

`func (b *B) RunParallel(body func(*PB))` 会以并行的方式执行给定的基准测试。

`RunParallel` 会创建出多个 `goroutine`，并将 `b.N` 分配给这些 `goroutine` 执行，其中 `goroutine` 数量的默认值为 `GOMAXPROCS`。用户如果想要增加非CPU受限（non-CPU-bound）基准测试的并行性，那么可以在 `RunParallel` 之前调用 `SetParallelism`。`RunParallel` 通常会与 `-cpu` 标志一同使用。

```
func BenchmarkSplitParallel(b *testing.B) {
    // b.SetParallelism(1) // 设置使用的CPU数
    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            Split("沙河有沙又有河", "沙")
        }
    })
}
```

执行一下基准测试：

```
split $ go test -bench=.
goos: darwin
goarch: amd64
pkg: github.com/Q1mi/studygo/code_demo/test_demo/split
BenchmarkSplit-8           100000000          131 ns/op
BenchmarkSplitParallel-8    500000000          36.1 ns/op
PASS
ok      github.com/Q1mi/studygo/code_demo/test_demo/split      3.308s
```

还可以通过在测试命令后添加 `-cpu` 参数如 `go test -bench=. -cpu 1` 来指定使用的CPU数量。

Setup与TearDown

测试程序有时需要在测试之前进行额外的设置（setup）或在测试之后进行拆卸（teardown）。

TestMain

通过在 `*_test.go` 文件中定义 `TestMain` 函数来可以在测试之前进行额外的设置（setup）或在测试之后进行拆卸（teardown）操作。

如果测试文件包含函数: `func TestMain(m *testing.M)` 那么生成的测试会先调用 `TestMain(m)`, 然后再运行具体测试。 `TestMain` 运行在主 `goroutine` 中, 可以在调用 `m.Run` 前后做任何设置（setup）和拆卸（teardown）。退出测试的时候应该使用 `m.Run` 的返回值作为参数调用 `os.Exit`。

一个使用 `TestMain` 来设置Setup和TearDown的示例如下:

```
func TestMain(m *testing.M) {
    fmt.Println("write setup code here...") // 测试之前的做一些设置
    // 如果 TestMain 使用了 flags, 这里应该加上flag.Parse()
    retCode := m.Run() // 执行测试
    fmt.Println("write teardown code here...") // 测试之后做一些拆卸工作
    os.Exit(retCode) // 退出测试
}
```

需要注意的是：在调用 `TestMain` 时, `flag.Parse` 并没有被调用。所以如果 `TestMain` 依赖于command-line标志(包括 testing 包的标记), 则应该显示的调用 `flag.Parse`。

子测试的Setup与Teardown

有时候我们可能需要为每个测试集设置Setup与Teardown, 也有可能需要为每个子测试设置Setup与Teardown。下面我们定义两个函数工具函数如下：

```
// 测试集的Setup与Teardown
func setupTestCase(t *testing.T) func(t *testing.T) {
    t.Log("如有需要在此执行:测试之前的setup")
    return func(t *testing.T) {
        t.Log("如有需要在此执行:测试之后的teardown")
    }
}

// 子测试的Setup与Teardown
func setupSubTest(t *testing.T) func(t *testing.T) {
    t.Log("如有需要在此执行:子测试之前的setup")
    return func(t *testing.T) {
        t.Log("如有需要在此执行:子测试之后的teardown")
    }
}
```

使用方式如下：

```

func TestSplit(t *testing.T) {
    type test struct { // 定义test结构体
        input string
        sep   string
        want  []string
    }
    tests := map[string]test{ // 测试用例使用map存储
        "simple": {input: "a:b:c", sep: ":", want: []string{"a", "b", "c"}},
        "wrong sep": {input: "a:b:c", sep: ",", want: []string{"a:b:c"}},
        "more sep": {input: "abcd", sep: "bc", want: []string{"a", "d"}},
        "leading sep": {input: "沙河有沙又有河", sep: "沙", want: []string{"", "河有", "又有河"}},
    }
    teardownTestCase := setupTestCase(t) // 测试之前执行setup操作
    defer teardownTestCase(t)          // 测试之后执行teardown操作

    for name, tc := range tests {
        t.Run(name, func(t *testing.T) { // 使用t.Run()执行子测试
            teardownSubTest := setupSubTest(t) // 子测试之前执行setup操作
            defer teardownSubTest(t)          // 测试之后执行teardown操作
            got := Split(tc.input, tc.sep)
            if !reflect.DeepEqual(got, tc.want) {
                t.Errorf("expected:%#v, got:%#v", tc.want, got)
            }
        })
    }
}

```

测试结果如下：

```

split $ go test -v
===[ RUN  TestSplit
===[ RUN  TestSplit/simple
===[ RUN  TestSplit/wrong_sep
===[ RUN  TestSplit/more_sep
===[ RUN  TestSplit/leading_sep
--- PASS: TestSplit (0.00s)
    split_test.go:71: 如有需要在此执行: 测试之前的setup
    --- PASS: TestSplit/simple (0.00s)
        split_test.go:79: 如有需要在此执行: 子测试之前的setup
        split_test.go:81: 如有需要在此执行: 子测试之后的 teardown
    --- PASS: TestSplit/wrong_sep (0.00s)
        split_test.go:79: 如有需要在此执行: 子测试之前的setup
        split_test.go:81: 如有需要在此执行: 子测试之后的 teardown
    --- PASS: TestSplit/more_sep (0.00s)
        split_test.go:79: 如有需要在此执行: 子测试之前的setup
        split_test.go:81: 如有需要在此执行: 子测试之后的 teardown
    --- PASS: TestSplit/leading_sep (0.00s)
        split_test.go:79: 如有需要在此执行: 子测试之前的setup
        split_test.go:81: 如有需要在此执行: 子测试之后的 teardown
    split_test.go:73: 如有需要在此执行: 测试之后的 teardown
===[ RUN  ExampleSplit
--- PASS: ExampleSplit (0.00s)
PASS
ok      github.com/Q1mi/studygo/code_demo/test_demo/split      0.006s

```

示例函数

示例函数的格式

被 `go test` 特殊对待的第三种函数就是示例函数，它们的函数名以 `Example` 为前缀。它们既没有参数也没有返回值。标准格式如下：

```
func ExampleName() {
    // ...
}
```

示例函数示例

下面的代码是我们为 `Split` 函数编写的一个示例函数：

```
func ExampleSplit() {
    fmt.Println(split.Split("a:b:c", ":"))
    fmt.Println(split.Split("沙河有沙又有河", "沙"))

    // Output:
    // [a b c]
    // [ 河有 又有河]
}
```

为你的代码编写示例代码有如下三个用处：

1. 示例函数能够作为文档直接使用，例如基于web的godoc中能把示例函数与对应的函数或包相关联。
2. 示例函数只要包含了 `// Output:` 也是可以通过 `go test` 运行的可执行测试。

```
split $ go test -run Example
PASS
ok      github.com/Q1mi/studygo/code_demo/test_demo/split      0.006s
```

3. 示例函数提供了可以直接运行的示例代码，可以直接在 [golang.org](#) 的 `godoc` 文档服务器上使用 `Go Playground` 运行示例代码。下图为 `strings.ToUpper` 函数在 Playground 的示例函数效果。

```
func ToUpper
```

```
func ToUpper(s string) string
```

`ToUpper` returns a copy of the string `s` with all Unicode letters mapped to their upper case.

▼ Example

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.ToUpper("Gopher"))
}
```

Run

Format

Share

