Golang程序语言设计

> 根据前面所学的编程理论知识,来系统学习一门新的语言
>
> 从以下几个方面入手
>
> 1. 变量定义
> 2. 数据类型
> 3. 运算符,逻辑语句,递归
> 4. 函数
> 5. 复合类型
> 6. 面向对象
> 7. 并发

变量定义

Go语言中变量名由字母,数字,下划线组成,其中首字母不能为数字(与java一致)

使用**var**关键字进行声明

```
var abc int
var abc , aaa int //可一次声明多个变量
var aaa = "aaa" //省略类型    会进行类型推断
aaa:="fangcong" 省略var    //编译出错   因为aaa上面已经声明
bbb:=1     //编译通过
```

变量声明时,若不指定初始值则变量默认为零值 (**大部分语言大同小异**)

数值类型(包括负数) 为0

布尔类型为 false

字符串为 ""

指针类型,数组类型,map,管道,函数,error 为nil

常量定义

关键字 **const**

```
显式类型定义:  const b string = "abc"
隐式类型定义:  const b = "abc"
```

常量可以作为枚举

```
const (
    Unknown = 0
    Female = 1
    Male = 2
)
```

数据类型

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

值类型和引用类型

所有像 int、float、bool 和 string 这些基本类型都属于值类型，使用这些类型的变量直接指向存在内存中的值：

Fig 4.1: Value type

当使用等号 `=` 将一个变量的值赋值给另一个变量时，如：`j = i`，实际上是在内存中将 i 的值进行了拷贝：

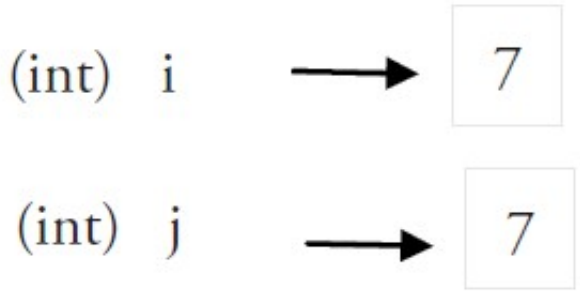


Fig 4.2: Assignment of value types

你可以通过 &i 来获取变量 i 的内存地址，例如：0xf840000040（每次的地址都可能不一样）。值类型的变量的值存储在栈中。

内存地址会根据机器的不同而有所不同，甚至相同的程序在不同的机器上执行后也会有不同的内存地址。因为每台机器可能有不同的存储器布局，并且位置分配也可能不同。

更复杂的数据通常会需要使用多个字，这些数据一般使用引用类型保存。

一个引用类型的变量 r1 存储的是 r1 的值所在的内存地址（数字），或内存地址中第一个字所在的位置。

Fig 4.3: Reference types and assignment

这个内存地址为称之为指针，这个指针实际上也被存在另外的某一个字中。

同一个引用类型的指针指向的多个字可以是在连续的内存地址中（内存布局是连续的），这也是计算效率最高的一种存储形式；也可以将这些字分散存放在内存中，每个字都指示了下一个字所在的内存地址。

当使用赋值语句 r2 = r1 时，只有引用（地址）被复制。

如果 r1 的值被改变了，那么这个值的所有引用都会指向被修改后的内容，在这个例子中，r2 也会受到影响。

我们知道可以在变量的初始化时省略变量的类型而由系统自动推断，声明语句写上 var 关键字其实是显得有些多余了，因此我们可以将它们简写为 a := 50 或 b := false。

a 和 b 的类型（int 和 bool）将由编译器自动推断。

这是使用变量的首选形式，但是它只能被用在函数体内，而不可以用于全局变量的声明与赋值。使用操作符 := 可以高效地创建一个新的变量，称之为初始化声明。

运算符

算术运算符

运算符	描述	实例
+	相加	A + B 输出结果 30
-	相减	A - B 输出结果 -10
*	相乘	A * B 输出结果 200
/	相除	B / A 输出结果 2
%	求余	B % A 输出结果 0
++	自增	A++ 输出结果 11
--	自减	A-- 输出结果 9

关系运算符

运算符	描述	实例
==	检查两个值是否相等，如果相等返回 True 否则返回 False。	(A == B) 为 False
!=	检查两个值是否不相等，如果不相等返回 True 否则返回 False。	(A != B) 为 True
>	检查左边值是否大于右边值，如果是返回 True 否则返回 False。	(A > B) 为 False
<	检查左边值是否小于右边值，如果是返回 True 否则返回 False。	(A < B) 为 True
>=	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。	(A >= B) 为 False
<=	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。	(A <= B) 为 True

逻辑运算符

运算符	描述	实例
&&	逻辑 AND 运算符。 如果两边的操作数都是 True，则条件 True，否则为 False。	(A && B) 为 False
||	逻辑 OR 运算符。 如果两边的操作数有一个 True，则条件 True，否则为 False。	(A || B) 为 True
!	逻辑 NOT 运算符。 如果条件为 True，则逻辑 NOT 条件 False，否则为 True。	

位运算符

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

运算符	描述	实例
&	按位与运算符"&"是双目运算符。 其功能是参与运算的两数各对应的二进位相与。	(A & B) 结果为 12, 二进制为 0000 1100
|	按位或运算符"|"是双目运算符。 其功能是参与运算的两数各对应的二进位相或	(A | B) 结果为 61, 二进制为 0011 1101
^	按位异或运算符"^"是双目运算符。 其功能是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。	(A ^ B) 结果为 49, 二进制为 0011 0001
<<	左移运算符"<<"是双目运算符。左移n位就是乘以2的n次方。 其功能把"<<"左边的运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。	A << 2 结果为 240 ，二进制为 1111 0000
>>	右移运算符">>"是双目运算符。右移n位就是除以2的n次方。 其功能是把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数。	A >>

赋值运算符

运算符	描述	实例
=	简单的赋值运算符，将一个表达式的值赋给一个左值	C = A + B 将 A + B 表达式结果赋值给 C
+=	相加后再赋值	C += A 等于 C = C + A
-=	相减后再赋值	C -= A 等于 C = C - A
*=	相乘后再赋值	C *= A 等于 C = C * A
/=	相除后再赋值	C /= A 等于 C = C / A
%=	求余后再赋值	C %= A 等于 C = C % A
<<=	左移后赋值	C <<= 2 等于 C = C << 2
>>=	右移后赋值	C >>= 2 等于 C = C >> 2
&=	按位与后赋值	C &= 2 等于 C = C & 2
^=	按位异或后赋值	C ^= 2 等于 C = C ^ 2
|=	按位或后赋值	C |= 2 等于 C = C | 2

条件语句

if 语句 不需要括号(与java不同)

```
if 布尔表达式 {
   /* 在布尔表达式为 true 时执行 */
}
```

go语言没有三目运算符

循环语句

for 循环

```
package main

import "fmt"

func main() {
    for true  {
        fmt.Printf("这是无限循环。\n");
    }



    slice := []int{0, 1, 2, 3}
```

```
    myMap := make(map[int]*int)

    for index, value := range slice {
        num := value
        myMap[index] = &num
    }
    fmt.Println("=====new map=====")
    prtMap(myMap)
}

// for range 可以通过for range方式遍历容器类型如数组、切片和映射。



func prtMap(myMap map[int]*int) {
    for key, value := range myMap {
        fmt.Printf("map[%v]=%v\n", key, *value)
    }
}
```

函数定义

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

函数定义解析：

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

go中函数有值传递和引用传递

默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到实际参数。

值传递

```
/* 定义相互交换值的函数 */
func swap(x, y int) int {
   var temp int

   temp = x /* 保存 x 的值 */
   x = y    /* 将 y 值赋给 x */
   y = temp /* 将 temp 值赋给 y*/

   return temp;
}
```

引用传递

```
/* 定义交换值函数*/
func swap(x *int, y *int) {
   var temp int
   temp = *x    /* 保持 x 地址上的值 */
   *x = *y      /* 将 y 值赋给 x */
   *y = temp    /* 将 temp 值赋给 y */
}
```

数组

定义与java中相同

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

初始化数组

以下演示了数组初始化：

```
var balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

初始化数组中 {} 中的元素个数不能大于 [] 中的数字。

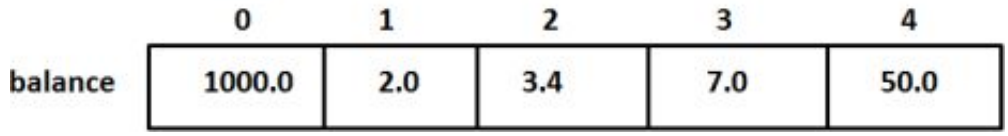
如果忽略 [] 中的数字不设置数组大小，Go 语言会根据元素的个数来设置数组的大小：

```
 var balance = [...]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

该实例与上面的实例是一样的，虽然没有设置数组的大小。

```
 balance[4] = 50.0
```

以上实例读取了第五个元素。数组元素可以通过索引（位置）来读取（或者修改），索引从0开始，第一个元素索引为 0，第二个索引为 1，以此类推。



结构体

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

结构体是由一系列具有相同类型或不同类型的数据构成的数据集合。

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

一旦定义了结构体类型，它就能用于变量的声明，语法格式如下：

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

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}


func main() {

    // 创建一个新的结构体
    fmt.Println(Books{"Go 语言", "www.runoob.com", "Go 语言教程", 6495407})

    // 也可以使用 key => value 格式
    fmt.Println(Books{title: "Go 语言", author: "www.runoob.com", subject: "Go
语言教程", book_id: 6495407})

    // 忽略的字段为 0 或 空
   fmt.Println(Books{title: "Go 语言", author: "www.runoob.com"})
}
```

如果要访问结构体成员，需要使用点号 . 操作符，格式为：

```
结构体.成员名
```

Slice

Go 语言切片是对数组的抽象。

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

你可以声明一个未指定大小的数组来定义切片：

```
var identifier []type
```

切片不需要说明长度。

或使用make()函数来创建切片:

```
var slice1 []type = make([]type, len)

也可以简写为

slice1 := make([]type, len)
```

也可以指定容量，其中capacity为可选参数。

```
make([]T, length, capacity)
```

这里 len 是数组的长度并且也是切片的初始长度。

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

直接初始化切片，[]表示是切片类型，{1,2,3}初始化值依次是1,2,3.其cap=len=3

```
s := arr[:]
```

初始化切片s,是数组arr的引用

```
s := arr[startIndex:endIndex]
```

将arr中从下标startIndex到endIndex-1 下的元素创建为一个新的切片

```
s := arr[startIndex:]
```

默认 endIndex 时将表示一直到arr的最后一个元素

```
s := arr[:endIndex]
```

默认 startIndex 时将表示从arr的第一个元素开始

```
s1 := s[startIndex:endIndex]
```

通过切片s初始化切片s1

```
s :=make([]int,len,cap)
```

通过内置函数make()初始化切片s,[]int 标识为其元素类型为int的切片

如果想增加切片的容量，我们必须创建一个新的更大的切片并把原分片的内容都拷贝过来。

下面的代码描述了从拷贝切片的 copy 方法和向切片追加新元素的 append 方法。

```
package main

import "fmt"

func main() {
   var numbers []int
   printSlice(numbers)

   /* 允许追加空切片 */
   numbers = append(numbers, 0)
   printSlice(numbers)

   /* 向切片添加一个元素 */
   numbers = append(numbers, 1)
   printSlice(numbers)

   /* 同时添加多个元素 */
   numbers = append(numbers, 2,3,4)
   printSlice(numbers)

   /* 创建切片 numbers1 是之前切片的两倍容量*/
   numbers1 := make([]int, len(numbers), (cap(numbers))*2)

   /* 拷贝 numbers 的内容到 numbers1 */
   copy(numbers1,numbers)
   printSlice(numbers1)
}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

Map

Map 是一种无序的键值对的集合。Map 最重要的一点是通过 key 来快速检索数据，key 类似于索引，指向数据的值。

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

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

如果不初始化 map，那么就会创建一个 nil map。nil map 不能用来存放键值对

```
package main

import "fmt"

func main() {
    var countryCapitalMap map[string]string /*创建集合 */
    countryCapitalMap = make(map[string]string)

    /* map插入key - value对,各个国家对应的首都 */
    countryCapitalMap [ "France" ] = "巴黎"
    countryCapitalMap [ "Italy" ] = "罗马"
    countryCapitalMap [ "Japan" ] = "东京"
    countryCapitalMap [ "India " ] = "新德里"

    /*使用键输出地图值 */
    for country := range countryCapitalMap {
        fmt.Println(country, "首都是", countryCapitalMap [country])
    }

    /*查看元素在集合中是否存在 */
    capital, ok := countryCapitalMap [ "American" ] /*如果确定是真实的,则存在,否则
不存在 */
    /*fmt.Println(capital) */
    /*fmt.Println(ok) */
    if (ok) {
        fmt.Println("American 的首都是", capital)
    } else {
        fmt.Println("American 的首都不存在")
    }
}
```

delete() 函数用于删除集合的元素, 参数为 map 和其对应的 key。实例如下：

```
package main
```

```
import "fmt"

func main() {
        /* 创建map */
        countryCapitalMap := map[string]string{"France": "Paris", "Italy":
"Rome", "Japan": "Tokyo", "India": "New delhi"}

        fmt.Println("原始地图")

        /* 打印地图 */
        for country := range countryCapitalMap {
                fmt.Println(country, "首都是", countryCapitalMap [ country ])
        }

        /*删除元素*/ delete(countryCapitalMap, "France")
        fmt.Println("法国条目被删除")

        fmt.Println("删除元素后地图")

        /*打印地图*/
        for country := range countryCapitalMap {
                fmt.Println(country, "首都是", countryCapitalMap [ country ])
        }
}
```

类型转换

类型转换用于将一种数据类型的变量转换为另外一种类型的变量。Go 语言类型转换基本格式如下：

```
type_name(expression)
```

type_name 为类型，expression 为表达式。

```
package main

import "fmt"

func main() {
   var sum int = 17
   var count int = 5
   var mean float32

   mean = float32(sum)/float32(count)
   fmt.Printf("mean 的值为: %f\n",mean)
}
```

指针的强制类型转换需要用到**unsafe**包中的函数实现

```
package main

import "unsafe"
import "fmt"

func main() {
    var a int =10
    var b *int =&a
    var c *int64 = (*int64)(unsafe.Pointer(b))
    fmt.Println(*c)
}
```

以上是强制类型转换，golang不像java和c++那样有隐式类型转换，也就是说int -> float 也需要强转

类型断言

```
package main

import "fmt"

func main() {
    var a interface{} =10
    switch a.(type){
    case int:
            fmt.Println("int")
    case float32:
            fmt.Println("string")
    }
}
```

```
package main

import "fmt"

func main() {
    var a interface{} =10
    t,ok:= a.(int)
    if ok{
        fmt.Println("int",t)
    }
    t2,ok:= a.(float32)
    if ok{
        fmt.Println("float32",t2)
    }
}
```

`t,ok:= a.(int)`有两个返回值,第一个是对应类型的值,第二个是bool类型的,类型判断是否正确。

interface

Go 语言提供了另外一种数据类型即接口，它把所有的具有共性的方法定义在一起，任何其他类型只要实现了这些方法就是实现了这个接口。

```
package main

import (
    "fmt"
)

type Phone interface {
    call()
}

type NokiaPhone struct {
}

func (nokiaPhone NokiaPhone) call() {
    fmt.Println("I am Nokia, I can call you!")
}

type IPhone struct {
}

func (iPhone IPhone) call() {
    fmt.Println("I am iPhone, I can call you!")
}

func main() {
    var phone Phone

    phone = new(NokiaPhone)
    phone.call()

    phone = new(IPhone)
    phone.call()

}
```

方法和函数

方法是特殊的函数，定义在某一特定的类型上，通过类型的实例来进行调用，这个实例被叫接收者(receiver)。
函数将变量作为参数：**Function1(recv)**
方法在变量上被调用：**recv.Method1()**

Go语言**不允许为简单的内置类型添加方法**

```
package main

import(
  "fmt"
)


func Add(a ,b int){          //函数合法
  fmt.Println(a+b)
}

func (a int) Add (b int){    //方法非法！不能是内置数据类型
  fmt.Println(a+b)
}
```

面向对象

封装

Golang区分公有属性和私有属性的机制就是方法或属性是否首字母大写，如果首字母大写的方法就是公有的，如果首字母小写的话就是私有的。

继承

GO语言的继承方式采用的是匿名组合的方式：Woman 结构体中包含匿名字段Person，那么Person中的属性也就属于Woman对象。

```
package main

import "fmt"

type Person struct {
    name string
}

type Woman struct {
    Person
    sex string
}

func main() {
    woman := Woman{Person{"wangwu"}, "女"}
    fmt.Println(woman.name)
    fmt.Println(woman.sex)
}
```

多态

> 使用struct 和 interface 实现

> 是类型理论里, 类型多态性的一中体现　Structural subtyping

```
package main

import "fmt"

type Eater interface {
    Eat()
}

type Man struct {
}

type Woman struct {
}

func (man *Man) Eat() {
    fmt.Println("Man Eat")
}

func (woman *Woman) Eat() {
    fmt.Println("Woman Eat")
}

func main() {
    var e Eater

    woman := Woman{}
    man := Man{}

    e = &woman
    e.Eat()

    e = &man
    e.Eat()
}
```

错误处理

Go 语言通过内置的错误接口提供了非常简单的错误处理机制。

error类型是一个接口类型，这是它的定义：

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

我们可以在编码中通过实现 error 接口类型来生成错误信息。

函数通常在最后的返回值中返回错误信息。使用errors.New 可返回一个错误信息：

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

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

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

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

goroutine

定义：在go里面，每一个并发执行的活动成为goroutine。

详解：goroutine可以认为是轻量级的线程，与创建线程相比，创建成本和开销都很小，每个goroutine的堆栈只有几kb，并且堆栈可根据程序的需要增长和缩小(线程的堆栈需指明和固定)，所以go程序从语言层面支持了高并发。

程序执行的背后：当一个程序启动的时候，只有一个goroutine来调用main函数，称它为主goroutine，新的goroutine通过go语句进行创建。

在函数或者方法前面加上关键字go，即创建一个并发运行的新goroutine。

```
package main

import (
    "fmt"
    "time"
)

func HelloWorld() {
    fmt.Println("Hello world goroutine")
}

func main() {
    go HelloWorld()       // 开启一个新的并发运行
    time.Sleep(1*time.Second)
    fmt.Println("我后面才输出来")
}
```

输出：

```
Hello world goroutine
我后面才输出来
```

需要注意的是，main执行速度很快，一定要加sleep，不然你不一定可以看到goroutine里头的输出。

这也说明了一个关键点：**当main函数返回时，所有的gourutine都是暴力终结的，然后程序退出。**

多个goroutine的创建

```
package main

import (
    "fmt"
    "time"
)

func DelayPrint() {
    for i := 1; i <= 4; i++ {
        time.Sleep(250 * time.Millisecond)
        fmt.Println(i)
    }
}

func HelloWorld() {
    fmt.Println("Hello world goroutine")
}

func main() {
    go DelayPrint()    // 开启第一个goroutine
    go HelloWorld()    // 开启第二个goroutine
    time.Sleep(2*time.Second)
    fmt.Println("main function")
}
```

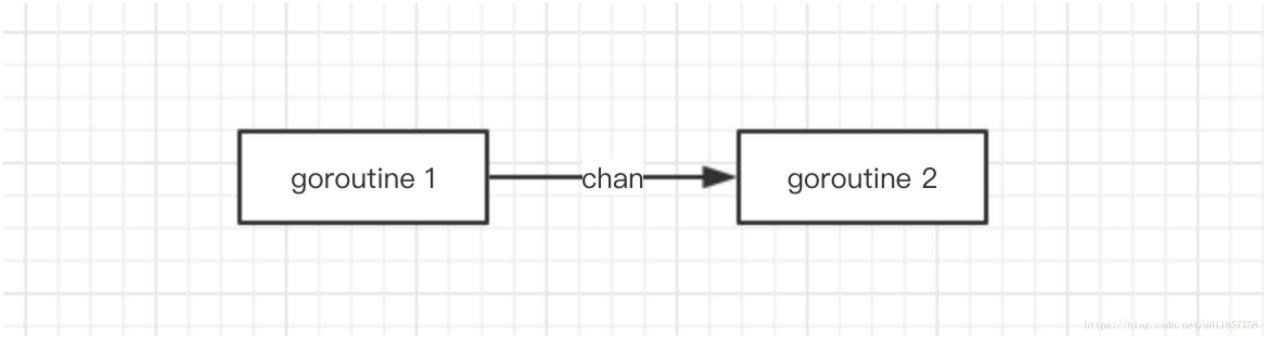
输出:

```
Hello world goroutine
1
2
3
4
5
main function
```

DelayPrint里头有sleep，那么会导致第二个goroutine堵塞或者等待吗?
答案是：no
疑惑：当程序执行go FUNC()的时候，只是简单的调用然后就立即返回了，并不关心函数里头发生的故事情节，所以不同的goroutine直接不影响，main会继续按顺序执行语句。

channel

如果说goroutine是Go并发的执行体，那么"通道"就是他们之间的连接。通道可以让一个goroutine发送特定的值到另外一个goroutine的通信机制。



声明

```
var ch chan int       // 声明一个传递int类型的channel
ch := make(chan int) // 使用内置函数make()定义一个channel

//=========

ch <- value           // 将一个数据value写入至channel，这会导致阻塞，直到有其他
goroutine从这个channel中读取数据
value := <-ch         // 从channel中读取数据，如果channel之前没有写入数据，也会导致阻
塞，直到channel中被写入数据为止

//=========

close(ch)             // 关闭channel
```

重要的四种通道使用

1. 无缓冲通道

无缓冲通道上的发送操作将会被阻塞，直到另一个goroutine在对应的通道上执行接收操作，此时值才传送完成，两个goroutine都继续执行。

```
package main

import (
    "fmt"
    "time"
)
var done chan bool
func HelloWorld() {
```

```
    fmt.Println("Hello world goroutine")
    time.Sleep(1*time.Second)
    done <- true
}
func main() {
    done = make(chan bool)  // 创建一个channel
    go HelloWorld()
    <-done
}
```

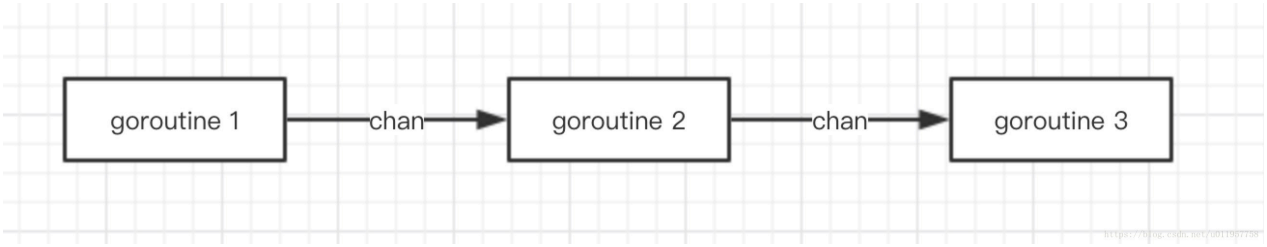
输出：

```
Hello world goroutine
```

由于main不会等goroutine执行结束才返回，前文专门加了sleep输出为了可以看到goroutine的输出内容，那么在这里由于是**阻塞**的，所以无需sleep。

2. 管道

通道可以用来连接goroutine，这样一个的输出是另一个输入。这就叫做管道。




```
package main

import (
    "fmt"
    "time"
)
var echo chan string
var receive chan string

// 定义goroutine 1
func Echo() {
    time.Sleep(1*time.Second)
    echo <- "咖啡色的羊驼"
}

// 定义goroutine 2
func Receive() {
    temp := <- echo // 阻塞等待echo的通道的返回
    receive <- temp
}
```

```
func main() {
    echo = make(chan string)
    receive = make(chan string)

    go Echo()
    go Receive()

    getStr := <-receive    // 接收goroutine 2的返回

    fmt.Println(getStr)
}
```

在这里不一定要去关闭channel，因为底层的垃圾回收机制会根据它**是否可以访问来决定是否自动回收它**。(这里不是根据channel是否关闭来决定的)

3. 单向通道类型

当程序则够复杂的时候，为了代码可读性更高，拆分成一个一个的小函数是需要的。

此时go提供了单向通道的类型，来实现函数之间channel的传递。

```
package main

import (
    "fmt"
    "time"
)

// 定义goroutine 1
func Echo(out chan<- string) {    // 定义输出通道类型
    time.Sleep(1*time.Second)
    out <- "咖啡色的羊驼"
    close(out)
}

// 定义goroutine 2
func Receive(out chan<- string, in <-chan string) { // 定义输出通道类型和输入类型
    temp := <-in // 阻塞等待echo的通道的返回
    out <- temp
    close(out)
}


func main() {
    echo := make(chan string)
    receive := make(chan string)

    go Echo(echo)
    go Receive(receive, echo)
```

```
    getStr := <-receive    // 接收goroutine 2的返回

    fmt.Println(getStr)
}
```

输出:

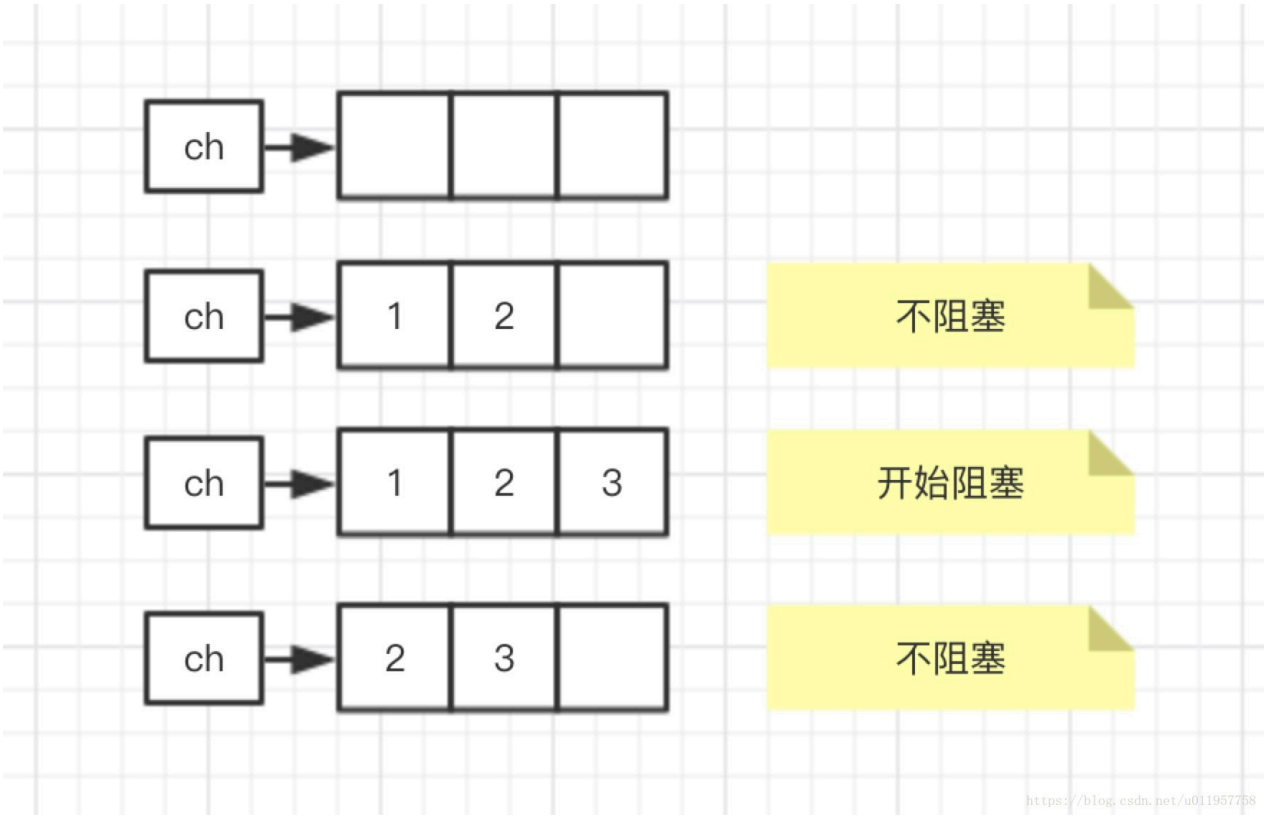
```
咖啡色的羊驼
```

4. 缓冲管道

goroutine的通道默认是是阻塞的，那么有什么办法可以缓解阻塞?
答案是：加一个缓冲区。

```
ch := make(chan string, 3) // 创建了缓冲区为3的通道

//=========
len(ch)   // 长度计算
cap(ch)   // 容量计算
```



goroutine死锁与友好退出

死锁现场一:

```go
package main

func main() {
    ch := make(chan int)
    <- ch // 阻塞main goroutine, 通道被锁
}
```

输出:

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
```

死锁线程二:

```go
package main

func main() {
    cha, chb := make(chan int), make(chan int)

    go func() {
        cha <- 1 // cha通道的数据没有被其他goroutine读取走，堵塞当前goroutine
        chb <- 0
    }()

    <- chb // chb 等待数据的写
}
```

为什么会产生死锁?

> 非缓冲通道上如果发生了流入无流出，或者流出无流入，就会引起死锁。
> 或者这么说：goroutine的非缓冲通道里头一定要一进一出，成对出现才行。
> 上面例子属于：一：流出无流入；二：流入无流出
>
> go 多个channel之间的关系多个无缓冲channel 设置值和取值的顺序 要一致，否则发生死锁

死锁的处理

1. 把没取走的取走:

```
package main

func main() {
    cha, chb := make(chan int), make(chan int)

    go func() {
        cha <- 1
        chb <- 0
    }()

    <- cha // 取走便是
    <- chb // chb 等待数据的写
}
```

2. 创建缓冲通道

```
package main

func main() {
    cha, chb := make(chan int, 3), make(chan int)

    go func() {
        cha <- 1 // cha通道的数据没有被其他goroutine读取走，堵塞当前goroutine
        chb <- 0
    }()

    <- chb // chb 等待数据的写
}
```

这样的话，cha可以缓存一个数据，cha就不会挂起当前的goroutine了。除非再放两个进去，塞满缓冲通道就会了。

go程序制作成docker镜像运行失败

在宿主机使用 go build 生成go的可执行文件，但是在写入dockerfile生成镜像后 启动失败

```
FROM alpine:latest

ADD nap-executor /usr/bin/
```

进入容器执行失败

```
/usr/bin/nap-executor -f /etc/nap-executor/config.json
```

解决方案：

在go build时 使用 go build -tags netgo

go中json序列化与反序列化

> # Go_Json_Unmarshal_Marshal
>
> func Marshal(v interface{}) ([]byte, error)
>
> func Unmarshal(data []byte,v interface{}) error

```
var deviceProperty proto.DeviceProperty
  if request.Property != "" {
    if err := json.Unmarshal([]byte(request.Property), &deviceProperty); err
!= nil {
      logs.Error(fmt.Sprintf("[ %s ]unmarshal device property %s failed, %s",
request.DeviceId, request.Property, err))
      *response = proto.RunCommandResponse{
        CommonResponse: proto.CommonResponse{
          Retcode:  -1,
          Message:  fmt.Sprintf("unmarshal device property %s failed, %s",
request.Property, err),
          DeviceId: request.DeviceId,
        },
        Output: []string{},
      }
      return nil
    }
  }
```

golang IO接口

> 在 io 包中最重要的是两个接口：Reader 和 Writer 接口。本章所提到的各种 IO 包，都跟这两个接口有关，也就是说，只要满足这两个接口，它就可以使用 IO 包的功能。

Reader 接口

Reader 接口的定义如下：

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

官方文档中关于该接口方法的说明：

> Read 将 len(p) 个字节读取到 p 中。它返回读取的字节数 n（0 <= n <= len(p)） 以及任何遇到的错误。即使 Read 返回的 n < len(p)，它也会在调用过程中占用 len(p) 个字节作为暂存空间。若可读取的数据不到 len(p) 个字节，Read 会返回可用数据，而不是等待更多数据。

> 当 Read 在成功读取 n > 0 个字节后遇到一个错误或 EOF (end-of-file)，它会返回读取的字节数。它可能会同时在本次的调用中返回一个non-nil错误,或在下一次的调用中返回这个错误（且 n 为 0）。 一般情况下, Reader会返回一个非0字节数n, 若 n = len(p) 个字节从输入源的结尾处由 Read 返回，Read可能返回 err == EOF 或者 err == nil。并且之后的 Read() 都应该返回 (n:0, err:EOF)。

> 调用者在考虑错误之前应当首先处理返回的数据。这样做可以正确地处理在读取一些字节后产生的 I/O 错误，同时允许EOF的出现。

根据 Go 语言中关于接口和满足了接口的类型的定义（Interface_types），我们知道 Reader 接口的方法集（Method_sets）只包含一个 Read 方法，因此，所有实现了 Read 方法的类型都满足 io.Reader 接口，也就是说，在所有需要 io.Reader 的地方，可以传递实现了 Read() 方法的类型的实例。

下面，我们通过具体例子来谈谈该接口的用法。

```
func ReadFrom(reader io.Reader, num int) ([]byte, error) {
  p := make([]byte, num)
  n, err := reader.Read(p)
  if n > 0 {
    return p[:n], nil
  }
  return p, err
}
```

ReadFrom 函数将 io.Reader 作为参数，也就是说，ReadFrom 可以从任意的地方读取数据，只要来源实现了 io.Reader 接口。比如，我们可以从标准输入、文件、字符串等读取数据，示例代码如下：

```
// 从标准输入读取
data, err = ReadFrom(os.Stdin, 11)

// 从普通文件读取，其中 file 是 os.File 的实例
data, err = ReadFrom(file, 9)

// 从字符串读取
data, err = ReadFrom(strings.NewReader("from string"), 12)
```

小贴士

io.EOF 变量的定义：`var EOF = errors.New("EOF")`，是 error 类型。根据 reader 接口的说明，在 n > 0 且数据被读完了的情况下，当次返回的 error 有可能是 EOF 也有可能是 nil。

Writer 接口

Writer 接口的定义如下：

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

官方文档中关于该接口方法的说明：

> Write 将 len(p) 个字节从 p 中写入到基本数据流中。它返回从 p 中被写入的字节数 n（0 <= n <= len(p)）以及任何遇到的引起写入提前停止的错误。若 Write 返回的 n < len(p)，它就必须返回一个 非nil 的错误。

同样的，所有实现了Write方法的类型都实现了 io.Writer 接口。

在上个例子中，我们是自己实现一个函数接收一个 io.Reader 类型的参数。这里，我们通过标准库的例子来学习。

在fmt标准库中，有一组函数：Fprint/Fprintf/Fprintln，它们接收一个 io.Wrtier 类型参数（第一个参数），也就是说它们将数据格式化输出到 io.Writer 中。那么，调用这组函数时，该如何传递这个参数呢?

我们以 fmt.Fprintln 为例，同时看一下 fmt.Println 函数的源码。

```
func Println(a ...interface{}) (n int, err error) {
  return Fprintln(os.Stdout, a...)
}
```

- os.File 同时实现了 io.Reader 和 io.Writer
- strings.Reader 实现了 io.Reader
- bufio.Reader/Writer 分别实现了 io.Reader 和 io.Writer
- bytes.Buffer 同时实现了 io.Reader 和 io.Writer
- bytes.Reader 实现了 io.Reader
- compress/gzip.Reader/Writer 分别实现了 io.Reader 和 io.Writer
- crypto/cipher.StreamReader/StreamWriter 分别实现了 io.Reader 和 io.Writer
- crypto/tls.Conn 同时实现了 io.Reader 和 io.Writer
- encoding/csv.Reader/Writer 分别实现了 io.Reader 和 io.Writer
- mime/multipart.Part 实现了 io.Reader
- net/conn 分别实现了 io.Reader 和 io.Writer(Conn接口定义了Read/Write)

除此之外，io 包本身也有这两个接口的实现类型。如：

```
实现了 Reader 的类型：LimitedReader、PipeReader、SectionReader
实现了 Writer 的类型：PipeWriter
```

以上类型中，常用的类型有：os.File、strings.Reader、bufio.Reader/Writer、bytes.Buffer、bytes.Reader

ReaderFrom 和 WriterTo 接口

ReaderFrom 的定义如下：

```
type ReaderFrom interface {
    ReadFrom(r Reader) (n int64, err error)
}
```

官方文档中关于该接口方法的说明：

> ReadFrom 从 r 中读取数据，直到 EOF 或发生错误。其返回值 n 为读取的字节数。除 io.EOF 之外，在读取过程中遇到的任何错误也将被返回。

> 如果 ReaderFrom 可用，Copy 函数就会使用它。

注意：ReadFrom 方法不会返回 err == EOF。

下面的例子简单的实现将文件中的数据全部读取（显示在标准输出）：

```
file, err := os.Open("writeAt.txt")
if err != nil {
    panic(err)
}
defer file.Close()
writer := bufio.NewWriter(os.Stdout)
writer.ReadFrom(file)
writer.Flush()
```

当然，我们可以通过 ioutil 包的 ReadFile 函数获取文件全部内容。其实，跟踪一下 ioutil.ReadFile 的源码，会发现其实也是通过 ReadFrom 方法实现（用的是 bytes.Buffer，它实现了 ReaderFrom 接口）。

如果不通过 ReadFrom 接口来做这件事，而是使用 io.Reader 接口，我们有两种思路：

1. 先获取文件的大小（File 的 Stat 方法），之后定义一个该大小的 []byte，通过 Read 一次性读取
2. 定义一个小的 []byte，不断的调用 Read 方法直到遇到 EOF，将所有读取到的 []byte 连接到一起

这里不给出实现代码了，有兴趣的可以实现一下。

提示

通过查看 bufio.Writer 或 strings.Buffer 类型的 ReadFrom 方法实现，会发现，其实它们的实现和上面说的第 2 种思路类似。

WriterTo的定义如下：

```
type WriterTo interface {
    WriteTo(w Writer) (n int64, err error)
}
```

官方文档中关于该接口方法的说明：

> WriteTo 将数据写入 w 中，直到没有数据可写或发生错误。其返回值 n 为写入的字节数。 在写入过程中遇到的任何错误也将被返回。

> 如果 WriterTo 可用，Copy 函数就会使用它。

读者是否发现，其实 ReaderFrom 和 WriterTo 接口的方法接收的参数是 io.Reader 和 io.Writer 类型。根据 io.Reader 和 io.Writer 接口的讲解，对该接口的使用应该可以很好的掌握。

这里只提供简单的一个示例代码：将一段文本输出到标准输出

```
reader := bytes.NewReader([]byte("Go语言中文网"))
reader.WriteTo(os.Stdout)
```

通过 io.ReaderFrom 和 io.WriterTo 的学习，我们知道，如果这样的需求，可以考虑使用这两个接口：“一次性从某个地方读或写到某个地方去。”

Closer接口

接口定义如下：

```
type Closer interface {
    Close() error
}
```

该接口比较简单，只有一个 Close() 方法，用于关闭数据流。

文件 (os.File)、归档（压缩包）、数据库连接、Socket 等需要手动关闭的资源都实现了 Closer 接口。

实际编程中，经常将 Close 方法的调用放在 defer 语句中。

ByteReader 和 ByteWriter

通过名称大概也能猜出这组接口的用途：读或写一个字节。接口定义如下：

```
type ByteReader interface {
    ReadByte() (c byte, err error)
}

type ByteWriter interface {
    WriteByte(c byte) error
}
```

在标准库中，有如下类型实现了 io.ByteReader 或 io.ByteWriter:

- bufio.Reader/Writer 分别实现了io.ByteReader 和 io.ByteWriter
- bytes.Buffer 同时实现了 io.ByteReader 和 io.ByteWriter
- bytes.Reader 实现了 io.ByteReader
- strings.Reader 实现了 io.ByteReader

接下来的示例中，我们通过 bytes.Buffer 来一次读取或写入一个字节（主要代码）：

```
var ch byte
fmt.Scanf("%c\n", &ch)

buffer := new(bytes.Buffer)
err := buffer.WriteByte(ch)
if err == nil {
  fmt.Println("写入一个字节成功！准备读取该字节……")
  newCh, _ := buffer.ReadByte()
  fmt.Printf("读取的字节：%c\n", newCh)
} else {
  fmt.Println("写入错误")
}
```

程序从标准输入接收一个字节（ASCII 字符），调用 buffer 的 WriteByte 将该字节写入 buffer 中，之后通过 ReadByte 读取该字节。

一般地，我们不会使用 bytes.Buffer 来一次读取或写入一个字节。那么，这两个接口有哪些用处呢?

在标准库 encoding/binary 中，实现Google-ProtoBuf中的 Varints 读取，ReadVarint 就需要一个 io.ByteReader 类型的参数，也就是说，它需要一个字节一个字节的读取。关于 encoding/binary 包在后面会详细介绍。

在标准库 image/jpeg 中，Encode函数的内部实现使用了 ByteWriter 写入一个字节。

使用gorm遇到的坑

> 今天在使用gorm做orm时，表1使用外键1关联表2，外键2关联表3，然后做插入时，插入表1的一条记录时会将附带的表2，表3
>
> 的记录也插入，但如果插入的记录有字段做了唯一性约束的话，如果跟以前的记录发生冲突时会插入不进去直接报错。

解决方法明天更新,回家还是不要想这些问题了

——后记，没有用关联表，使用的json存储，还是太菜了

ioutil 方便的io操作函数集

> 虽然 io 包提供了不少类型、方法和函数，但有时候使用起来不是那么方便。比如读取一个文件中的所有内容。为此，标准库中提供了一些常用、方便的IO操作函数。
>
> 说明：这些函数使用都相对简单，一般就不举例子了。

NopCloser 函数

有时候我们需要传递一个 io.ReadCloser 的实例，而我们现在有一个 io.Reader 的实例，比如：strings.Reader ，这个时候 NopCloser 就派上用场了。它包装一个io.Reader，返回一个 io.ReadCloser ，而相应的 Close 方法啥也不做，只是返回 nil。

比如，在标准库 net/http 包中的 NewRequest，接收一个 io.Reader 的 body，而实际上，Request 的 Body 的类型是 io.ReadCloser，因此，代码内部进行了判断，如果传递的 io.Reader 也实现了 io.ReadCloser 接口，则转换，否则通过ioutil.NopCloser 包装转换一下。相关代码如下：

```
    rc, ok := body.(io.ReadCloser)
    if !ok && body != nil {
    rc = ioutil.NopCloser(body)
    }
```

如果没有这个函数，我们得自己实现一个。当然，实现起来很简单，读者可以看看 NopCloser 的实现。

ReadAll 函数

很多时候，我们需要一次性读取 io.Reader 中的数据，通过上一节的讲解，我们知道有很多种实现方式。考虑到读取所有数据的需求比较多，Go 提供了 ReadAll 这个函数，用来从io.Reader 中一次读取所有数据。

```
  func ReadAll(r io.Reader) ([]byte, error)
```

阅读该函数的源码发现，它是通过 bytes.Buffer 中的 ReadFrom 来实现读取所有数据的。该函数成功调用后会返回 err == nil 而不是 err == EOF。(成功读取完毕应该为 err == io.EOF，这里返回 nil 由于该函数成功期望 err == io.EOF，符合无错误不处理的理念)

ReadDir 函数

笔试题：编写程序输出某目录下的所有文件（包括子目录）

是否见过这样的笔试题?

在 Go 中如何输出目录下的所有文件呢？首先，我们会想到查 os 包，看 File 类型是否提供了相关方法（关于 os 包，后面会讲解）。

其实在 ioutil 中提供了一个方便的函数：ReadDir，它读取目录并返回排好序的文件和子目录名（[]os.FileInfo ）。通过这个方法，我们可以很容易的实现“面试题”。

```
func main() {
  dir := os.Args[1]
  listAll(dir,0)
}

func listAll(path string, curHier int){
  fileInfos, err := ioutil.ReadDir(path)
  if err != nil{fmt.Println(err); return}

  for _, info := range fileInfos{
    if info.IsDir(){
      for tmpHier := curHier; tmpHier > 0; tmpHier--{
        fmt.Printf("|\t")
      }
```

```
      fmt.Println(info.Name(),"\\")
      listAll(path + "/" + info.Name(),curHier + 1)
    }else{
      for tmpHier := curHier; tmpHier > 0; tmpHier--{
        fmt.Printf("|\t")
      }
      fmt.Println(info.Name())
    }
  }
}
```

ReadFile 和 WriteFile 函数

ReadFile 读取整个文件的内容，在上一节我们自己实现了一个函数读取文件整个内容，由于这种需求很常见，因此 Go 提供了 ReadFile 函数，方便使用。ReadFile 的实现和ReadAll 类似，不过，ReadFile 会先判断文件的大小，给 bytes.Buffer 一个预定义容量，避免额外分配内存。

ReadFile 函数的签名如下:

```
func ReadFile(filename string) ([]byte, error)
```

函数文档:

> ReadFile 从 filename 指定的文件中读取数据并返回文件的内容。成功的调用返回的err 为 nil 而非 EOF。因为本函数定义为读取整个文件，它不会将读取返回的 EOF 视为应报告的错误。(同 ReadAll)

WriteFile 函数的签名如下:

```
func WriteFile(filename string, data []byte, perm os.FileMode) error
```

函数文档:

> WriteFile 将data写入filename文件中，当文件不存在时会根据perm指定的权限进行创建一个,文件存在时会先清空文件内容。对于 perm 参数，我们一般可以指定为：0666，具体含义 os 包中讲解。

demo 将爬虫返回的网页存入demo.html中

```
func (request *Request) root() {
  req, _ := http.NewRequest("GET", request.url, nil)
  req.Header.Set("user-agent", rand_ua())
  req.Header.Set("Host", "www.biquge.com.cn")
  req.Header.Add("Accept-Charset", "utf-8")
  resp, err := (&http.Client{}).Do(req)
  if err != nil {
    fmt.Println(err.Error())
  }
  defer resp.Body.Close()
```

```
  if resp.StatusCode == 200 {

    respByte, _ := ioutil.ReadAll(resp.Body)
    temp := string(respByte)
    fmt.Println(temp)
    path := "./demo.html"
    err := ioutil.WriteFile(path, []byte(temp), 0666)

    if err != nil {
      panic(err)
    }
    /*if err := request.rootDoc.ReadFromString(temp); err != nil {
      fmt.Println(err.Error())
      return
    }*/

  } else {
    fmt.Println("request error status_code:" + strconv.Itoa(resp.StatusCode))
    return
  }
}
```

小提示

ReadFile 源码中先获取了文件的大小，当大小 < 1e9 时，才会用到文件的大小。按源码中注释的说法是 FileInfo 不会很精确地得到文件大小。

TempDir 和 TempFile 函数

操作系统中一般都会提供临时目录，比如 linux 下的 /tmp 目录（通过 os.TempDir() 可以获取到）。有时候，我们自己需要创建临时目录，比如 Go 工具链源码中（src/cmd/go/build.go），通过 TempDir 创建一个临时目录，用于存放编译过程的临时文件：

```
  b.work, err = ioutil.TempDir("", "go-build")
```

第一个参数如果为空，表明在系统默认的临时目录（ os.TempDir ）中创建临时目录；第二个参数指定临时目录名的前缀，该函数返回临时目录的路径。

相应的，TempFile 用于创建临时文件。如 gofmt 命令的源码中创建临时文件：

```
  f1, err := ioutil.TempFile("", "gofmt")
```

参数和 ioutil.TempDir 参数含义类似。

这里需要**注意**：创建者创建的临时文件和临时目录要负责删除这些临时目录和文件。如删除临时文件：

```
  defer func() {
    f.Close()
    os.Remove(f.Name())
  }()
```

Discard 变量

Discard 对应的类型（`type devNull int`）实现了 io.Writer 接口，同时，为了优化 io.Copy 到 Discard，避免不必要的工作，实现了 io.ReaderFrom 接口。

devNull 在实现 io.Writer 接口时，只是简单的返回（标准库文件：src/pkg/io/ioutil.go)。

```
  func (devNull) Write(p []byte) (int, error) {
    return len(p), nil
  }
```

而 ReadFrom 的实现是读取内容到一个 buf 中，最大也就 8192 字节，其他的会丢弃（当然，这个也不会读取）。

fmt 格式化io

> fmt 包实现了格式化I/O函数，类似于C的 printf 和 scanf. 格式“占位符”衍生自C，但比C更简单。

Sample

```
type user struct {
  name string
}

func main() {
  u := user{"tang"}
  //Printf 格式化输出
  fmt.Printf("% + v\n", u)      //格式化输出结构
  fmt.Printf("%#v\n", u)        //输出值的 Go 语言表示方法
  fmt.Printf("%T\n", u)         //输出值的类型的 Go 语言表示
  fmt.Printf("%t\n", true)      //输出值的 true 或 false
  fmt.Printf("%b\n", 1024)      //二进制表示
  fmt.Printf("%c\n", 11111111) //数值对应的 Unicode 编码字符
  fmt.Printf("%d\n", 10)        //十进制表示
  fmt.Printf("%o\n", 8)         //八进制表示
  fmt.Printf("%q\n", 22)        //转化为十六进制并附上单引号
  fmt.Printf("%x\n", 1223)      //十六进制表示，用a-f表示
  fmt.Printf("%X\n", 1223)      //十六进制表示，用A-F表示
  fmt.Printf("%U\n", 1233)      //Unicode表示
  fmt.Printf("%b\n", 12.34)     //无小数部分，两位指数的科学计数法6946802425218990p-
49
  fmt.Printf("%e\n", 12.345)    //科学计数法，e表示
  fmt.Printf("%E\n", 12.34455) //科学计数法，E表示
```

```
    fmt.Printf("%f\n", 12.3456)  //有小数部分，无指数部分
    fmt.Printf("%g\n", 12.3456)  //根据实际情况采用%e或%f输出
    fmt.Printf("%G\n", 12.3456)  //根据实际情况采用%E或%f输出
    fmt.Printf("%s\n", "wqdew")  //直接输出字符串或者[]byte
    fmt.Printf("%q\n", "dedede") //双引号括起来的字符串
    fmt.Printf("%x\n", "abczxc") //每个字节用两字节十六进制表示，a-f表示
    fmt.Printf("%X\n", "asdzxc") //每个字节用两字节十六进制表示，A-F表示
    fmt.Printf("%p\n", 0x123)    //0x开头的十六进制数表示
}
```

sync.WaitGroup

在有多个goroutine 工作线程工作时，我们main线程需要等待工作线程完成才能结束时,这个时候就需要**sync.WaitGroup**

效果与java中的**CountDownLatch**相同

`WaitGroup` 对象内部有一个计数器，最初从0开始，它有三个方法：`Add(), Done(), Wait()` 用来控制计数器的数量。`Add(n)` 把计数器设置为`n` ， `Done()` 每次把计数器`-1` ， `wait()` 会阻塞代码的运行，直到计数器地值减为0。

```
func main() {
    wg := sync.WaitGroup{}
    wg.Add(100)
    for i := 0; i < 100; i++ {
        go func(i int) {
            fmt.Println(i)
            wg.Done()
        }(i)
    }
    wg.Wait()
}
```

这里首先把`wg` 计数设置为100， 每个for循环运行完毕都把计数器减一，主函数中使用`Wait()` 一直阻塞，直到wg为零——也就是所有的100个for循环都运行完毕。相对于使用管道来说， `WaitGroup` 轻巧了许多。

note:

1. 计数器不能为负值
2. WaitGroup对象不是一个引用类型 (**一定要通过指针传值，不然进程会进入死锁状态**)

Context

`context`是`Go`并发编程中常用到一种编程模式。

context常用的使用姿势：

1. web编程中，一个请求对应多个goroutine之间的数据交互
2. 超时控制
3. 上下文控制

context接口

```
type Context interface {

    Deadline() (deadline time.Time, ok bool)

    Done() <-chan struct{}

    Err() error

    Value(key interface{}) interface{}
}
```

`Context`接口包含四个方法：

字段	含义
Deadline	返回一个time.Time，表示当前Context应该结束的时间，如果没有设定期限，将返回`ok == false`。
Done	当Context被取消或者超时时候返回的一个close的channel，告诉给context相关的函数要停止当前工作然后返回了。(这个有点像全局广播)
Err	context被取消的原因
Value	context实现共享数据存储的地方，是协程安全的（还记得之前有说过map是不安全的？所以遇到map的结构,如果不是sync.Map,需要加锁来进行操作）

官方提供了4个Context实现

实现	结构体	作用
emptyCtx	type emptyCtx int	完全空的Context，实现的函数也都是返回nil，仅仅只是实现了Context的接口
cancelCtx	type cancelCtx struct { Context mu sync.Mutex done chan struct{} children map[canceler]struct{} err error }	继承自Context，同时也实现了canceler接口
timerCtx	type timerCtx struct { cancelCtx timer *time.Timer // Under cancelCtx.mu. deadline time.Time }	继承自**cancelCtx**，增加了timeout机制
valueCtx	type valueCtx struct { Context key, val interface{} }	存储键值对的数据

为了更方便的创建Context，包里头定义了Background来作为所有Context的根，它是一个emptyCtx的实例。

```
var (
    background = new(emptyCtx)
    todo       = new(emptyCtx)
)

func Background() Context {
    return background
}
```

你可以认为所有的Context是树的结构，Background是树的根，当任一Context被取消的时候，那么继承它的Context 都将被回收。

WithCancel

`WithCancel`函数用来创建一个可取消的`context`，即`cancelCtx`类型的`context`。`WithCancel`返回一个`context`和一个`CancelFunc`，调用`CancelFunc`即可触发`cancel`操作。

吃汉堡比赛，奥特曼每秒吃0-5个，计算吃到10的用时

example:

```
func main() {
  ctx, cancel := context.WithCancel(context.Background())
  eatNum := chiHanBao(ctx)
  for n := range eatNum {
    if n >= 10 {
      cancel()
      break
```

```
    }
  }

  fmt.Println("正在统计结果。。。")
  time.Sleep(1 * time.Second)
}

func chiHanBao(ctx context.Context) <-chan int {
  c := make(chan int)
  // 个数
  n := 0
  // 时间
  t := 0
  go func() {
    for {
      //time.Sleep(time.Second)
      select {
      case <-ctx.Done():
        fmt.Printf("耗时 %d 秒，吃了 %d 个汉堡 \n", t, n)
        return
      case c <- n:
        incr := rand.Intn(5)
        n += incr
        if n >= 10 {
          n = 10
        }
        t++
        fmt.Printf("我吃了 %d 个汉堡\n", n)
      }
    }
  }()

  return c
}
```

result:

```
我吃了 1 个汉堡
我吃了 3 个汉堡
我吃了 5 个汉堡
我吃了 9 个汉堡
我吃了 10 个汉堡
正在统计结果。。。
耗时 6 秒，吃了 10 个汉堡
```

WithTimeout

执行一段代码，控制**执行到某个时间**的时候，整个程序结束。

```go
func main() {
  ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
  chiHanBao(ctx)
  defer cancel()
}

func chiHanBao(ctx context.Context) {
  n := 0
  for {
    select {
    case <-ctx.Done():
      fmt.Println("stop \n")
      return
    default:
      incr := rand.Intn(5)
      n += incr
      fmt.Printf("我吃了 %d 个汉堡\n", n)
    }
    time.Sleep(time.Second)
  }
}
```

WithValue

携带关键信息，为全链路提供线索，比如接入elk等系统，需要来一个trace_id，那WithValue就非常适合做这个事。

```go
func main() {
  ctx := context.WithValue(context.Background(), "trace_id", "88888888")
  // 携带session到后面的程序中去
  ctx = context.WithValue(ctx, "session", 1)

  process(ctx)
}

func process(ctx context.Context) {
  session, ok := ctx.Value("session").(int)
  fmt.Println(ok)
  if !ok {
    fmt.Println("something wrong")
    return
  }

  if session != 1 {
    fmt.Println("session 未通过")
    return
  }
```

```
  traceID := ctx.Value("trace_id").(string)
  fmt.Println("traceID:", traceID, "-session:", session)
}
```

note:

Context要是全链路函数的第一个参数。

总结：

`context`主要用于父子任务之间的同步取消信号，本质上是一种协程调度的方式。另外在使用`context`时有两点值得注意：上游任务仅仅使用`context`通知下游任务不再需要，但不会直接干涉和中断下游任务的执行，由下游任务自行决定后续的处理操作，也就是说`context`的取消操作是无侵入的；`context`是线程安全的，因为`context`本身是不可变的（`immutable`），因此可以放心地在多个协程中传递使用。

golang 操作xml类型文件

> etree包是一个轻量级的纯go包，它以元素树的形式表示XML。它的设计灵感来自Python ElementTree 模块。
>
> 该软件包的一些功能和特性：
>
> - 将XML文档表示为元素树，以便于遍历。
> - 从头开始导入，序列化，修改或创建XML文档。
> - 向文件，字节片，字符串和io接口读写XML。
> - 使用轻量级的类似XPath的查询API执行简单或复杂的搜索。
> - 使用空格或制表符自动缩进XML，以提高可读性。
> - 完全实施；仅取决于标准的go库。
> - 构建在go encoding / xml 包之上。

```
import "github.com/beevik/etree"
```

创建文档

```
doc := etree.NewDocument()
doc.CreateProcInst("xml", `version="1.0" encoding="UTF-8"`)
doc.CreateProcInst("xml-stylesheet", `type="text/xsl" href="style.xsl"`)

people := doc.CreateElement("People")
people.CreateComment("These are all known people")

jon := people.CreateElement("Person")
jon.CreateAttr("name", "Jon")
```

```
sally := people.CreateElement("Person")
sally.CreateAttr("name", "Sally")

doc.Indent(2)
doc.WriteTo(os.Stdout)
```

Output:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="style.xsl"?>
<People>
  <!--These are all known people-->
  <Person name="Jon"/>
  <Person name="Sally"/>
</People>
```

读取文件

```
doc := etree.NewDocument()
if err := doc.ReadFromFile("bookstore.xml"); err != nil {
    panic(err)
}
```

也可以从字符串或者字节数组或者从流中读取数据到一个Document对象

也支持xpath表达式进行选取元素

(但值得注意的是，盗版就是盗版，不如python正版库，也许是我自己不会用 TnT)

python

```
span_list =root.xpath("//div[@id='list']/dl/dd/a/@href")
        for span in span_list:

            self.get_context(self.go_url("https://www.biquge.com.cn/"+span))
```

golang

```
span_list := doc.FindElements(`//div[@id='list']/dl/dd/a`) //这里写上@href就匹配不到
  for _, t := range span_list {
    span := t.SelectAttr("href").Value
  }
```

地址: https://github.com/beevik/etree

go设置代理

```
go env -w GOPROXY=https://goproxy.cn
```

如果上面的不生效 直接执行

```
GOPROXY=https://goproxy.cn go get ./...
```

go关闭验证包的有效性

```
go env -w GOSUMDB=off
```

更新依赖包

```
GOPROXY=direct go get -u github.com/sky-cloud-tec/proto
```