1、builtin介绍

builtin 包为Go的预声明标识符提供了文档。此处列出的条目其实并不在builtin 包中，对它们的描述只是为了让 godoc 给该语言的特殊标识符提供文档。
这个包里面`描述了类型以及内置函数`，这些内置函数都是为了操作类型而设定的。虽然这是一个包但是并不需要导入这个包。

2、常量

详解常量

```
package main

import (
    "fmt"
)

// 常量
// 1、常量的定义和全局变量变量定义类似，只是把var改成了const
const (
    X = 1
    Y = 2
)

// 2、如果常量省略了值默认和上面一行(邻近的一行)的值是相同的，首行必须有值，因为他没有上一行可以推导出值
const (
    T1 = 10
    T2 = 19
    T3  //19
)

// 3、iota(类似枚举)

// const iota = 0 // 无类型整数值(系统定义)
// iota是一个预定义的标识符，代表顺序按行增加的无符号整数，每个const声明单元（被括号括起来）相互独立，分别从0开始。

// 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。
```

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

// 跳过
const (
    v1 = iota   // 0
    v2          // 1
    _           // 跳过
    v3          // 3
)

// 插入
const (
    b1 = iota
    b2 = 100
    b3      // 100 b2中间插入，此时b3的规则就是找上一条的值赋给自己
    b4      // 100
)

const (
    c1 = iota   // 0
    c2 = 100    // 100
    c3 = iota   // 2    // 接着上面的开始，c2相当于占了一个位置，所以这里是2
    c4          // 3
)

// 多个常量定义在一行，最好是每一行的个数相同，否则容易发生错误
// 这里有两列，且首行各自赋了iota，所以，每一列的常量都会累加
const (
    a, b = iota + 1, iota + 2 //1,2
    c, d                      //2,3
    e, f                      //3,4
)

// 为bool类型提供的两个系统常量，bool的值只能是true或false
const (
    true  = 0 == 0 // 无类型布尔值  --> 输出true时结果仍是true
    false = 0 != 0 // 无类型布尔值  --> 输出false时结果仍是false
)
```

3、new和make

func new

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

内建函数new分配内存。其第一个实参为类型，而非值。其返回值为指向该类型的新分配的零值的指针。

func make

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

内建函数make分配并初始化一个类型为切片、映射、或通道的对象。其第一个实参为类型，而非值。make的返回类型与其参数相同，而非指向它的指针。其具体结果取决于具体的类型：

make函数针对的都是引用类型的对象

```
切片：size指定了其长度。该切片的容量等于其长度。切片支持第二个整数实参可用来指定不同的容量；
      它必须不小于其长度，因此 make([]int, 0, 10) 会分配一个长度为0，容量为10的切片。
映射：初始分配的创建取决于size，但产生的映射长度为0。size可以省略，这种情况下就会分配一个
      小的起始大小。
通道：通道的缓存根据指定的缓存容量初始化。若 size为零或被省略，该信道即为无缓存的。
```

通道无法自动扩容，所以开发中必须要指定可用的大小

CSDN @Golang-Study

```go
func main() {

    // make有初始化功能(针对引用类型)，比如切片存放的类型是int，那么就将切片中存放的值设置为0
    // 其他类型也是一样的，以此类推。

    // make(type, len, cap)
    slice := make([]int, 2, 10)
    fmt.Printf( format: "%p, %v, %T\n", slice, slice, slice)  // 0xc00001a140, [0 0], []int

    ptr := new([]int)
    fmt.Printf( format: "地址%p, 值%v, 类型%T\n", ptr, ptr, ptr) // 0xc000008078, &[], *[]int
    *ptr = append(*ptr,  elems...: 10)
    fmt.Println(ptr)    // &[10]

}
```

CSDN @Golang-Study

```
/*
        new 和 make
        值类型、地址类型和引用类型

        值类型就是开辟在栈上的一块空间，在传递时就会拷贝一份新的不会影响到自身
        地址类型相对于值类型，它是地址的表示16进制，指向内存中的一块内存空间，这里面的是实际需要的数据

        引用类型，不同于上面两种，在go中的各种数据类型中，只有切片、map和信道是引用类型。
        引用类型的内存模型不同于上面两种，这种类型本身和其他数据类型别无二致，但是内部维护的数据更为复杂；


            切片、map和管道是多种数据复合而成的一种数据类型

            切片、map和管道是通过结构体将多种数据组合而成的。

            切片主要有三种数据 --> 指向一块内存数组地址的16进制地址 / 切片的长度(数据量) / 切片的容量(可以自动扩容)
            map和管道类似切片，一个指向数据的指针，以及容量等

            这三种数据本身是结构体(值类型)，所以一旦要对变量本身操作，就必须要是传地址的(*[]int、*map[string]string等函数参数)
```

```
        比如 将切片传递给函数参数，在函数里面通过内置函数的append会返回一个切片，如果将返回的切片赋值给本身，那么此时原来的切片和当前的就不一样了，除非通过地址传递

        make内置函数是给切片、映射、或通道初始化的，主要是对他们内部维护的内存地址开辟指定空间,如果make，这块地址是nil，无法操作
    */
```

4、len和cap

func cap

```
func cap(v Type) int
```

内建函数cap返回 v 的容量，这取决于具体类型：

```
数组：v中元素的数量，与 len(v) 相同
数组指针：*v中元素的数量，与len(v) 相同
切片：切片的容量（底层数组的长度）；若 v为nil，cap(v) 即为零
信道：按照元素的单元，相应信道缓存的容量；若v为nil，cap(v)即为零
```

func len

```
func len(v Type) int
```

内建函数len返回 v 的长度，这取决于具体类型：

```
数组：v中元素的数量
数组指针：*v中元素的数量（v为nil时panic）
切片、映射：v中元素的数量；若v为nil，len(v)即为零
字符串：v中字节的数量
通道：通道缓存中队列（未读取）元素的数量；若v为 nil，len(v)即为零
```

CSDN @Golang-Study

```go
func testCap() {

    //数组
    var array1 [3]int
    array1[0] = 100
    fmt.Println(cap(array1)) // 3

    // 数组指针
    var arrayPtr *[3]int = new([3]int)
    (*arrayPtr)[1] = 999
    fmt.Println(cap(arrayPtr)) // 3

    // 切片
    var slice []int = make([]int, 3, 6)
    slice[2] = 89
    fmt.Println(cap(slice)) // 6

    // 信道
    var channel chan int = make(chan int, 10)
    fmt.Println(cap(channel)) // 10

    var chann *chan int = new(chan int)
    fmt.Println(cap(*chann)) // 0
}
```

```go
func testLen() {

    //数组
    var array1 [3]int
    array1[0] = 100
    fmt.Println(len(array1)) // 3

    // 数组指针
    var arrayPtr *[3]int = new([3]int)
    (*arrayPtr)[1] = 999
    fmt.Println(len(arrayPtr)) // 3

    // 切片
    var slice []int = make([]int, 3, 6)
    slice[2] = 89
    fmt.Println(len(slice)) // 3

    // 映射
    var mmp map[string]string = make(map[string]string, 3)
    mmp["name"] = "kiko"
    mmp["age"] = "23"
    fmt.Println(len(mmp)) // 2

    // 信道
    var channel chan int = make(chan int, 10)
    channel <- 10
    channel <- 30
    <-channel
    fmt.Println(len(channel)) // 1

    // 字符串
    str := "Hello"
    fmt.Println(len(str)) // 5

}
```

5、copy

copy函数`针对的是切片类型`的数据。拷贝之后的对象和原来的没有数据关联，改变其中一个不会影响另一个。

func copy

```go
func copy(dst, src []Type) int
```

内建函数copy将元素从来源切片复制到目标切片中，也能将字节从字符串复制到字节切片中。copy返回被复制的元素数量，它会是 len(src) 和 len(dst) 中较小的那个。来源和目标的底层内存可以重叠。

CSDN @Golang-Study

```go
func testCopy() {

    var src []int = []int{1, 2, 3, 4}

    // 目标切片对象必须初始化
    var dst []int = make([]int, 3)

    n := copy(dst, src)
```

```
    fmt.Println(n)   // 3
    fmt.Println(src) // [1 2 3 4]
    fmt.Println(dst) // [1 2 3]
}

func testStr() {

    str := "hello world!"
    var dst []byte = make([]byte, len(str))

    // 字符串复制到字节切片中
    n := copy(dst, str)

    fmt.Println(n)   // 12
    fmt.Println(dst) // [104 101 108 108 111 32 119 111 114 108 100 33]
}
```

6、delete

delete函数根据键删除对应的键值对，针对map操作。

func delete

```
func delete(m map[Type]Type1, key Type)
```

内建函数delete按照指定的键将元素从映射中删除。若m为nil或无此元素，delete不进行操作。

CSDN @Golang-Study

7、close

func close

```
func close(c chan<- Type)
```

内建函数close关闭信道，该通道必须为双向的或只发送的。它应当只由发送者执行，而不应由接收者执行，其效果是在最后发送的值被接收后停止该通道。在最后的值从已关闭的信道中被接收后，任何对其的接收操作都会无阻塞的成功。对于已关闭的信道，语句：

```
x, ok := <-c
```

还会将ok置为false。

CSDN @Golang-Study

8、panic和recover

func panic

```
func panic(v interface{})
```

内建函数panic停止当前Go程的正常执行。当函数F调用panic时，F的正常执行就会立刻停止。F中defer的所有函数先入后出执行后，F返回给其调用者G。G如同F一样行动，层层返回，直到该Go程中所有函数都按相反的顺序停止执行。之后，程序被终止，而错误情况会被报告，包括引发该恐慌的实参值，此终止序列称为恐慌过程。

func recover

```
func recover() interface{}
```

内建函数recover允许程序管理恐慌过程中的Go程。在defer的函数中，执行recover调用会取回传至panic调用的错误值，恢复正常执行，停止恐慌过程。若recover在defer的函数之外被调用，它将不会停止恐慌过程序列。在此情况下，或当该Go程不在恐慌过程中时，或提供给panic的实参为nil时，recover就会返回nil。

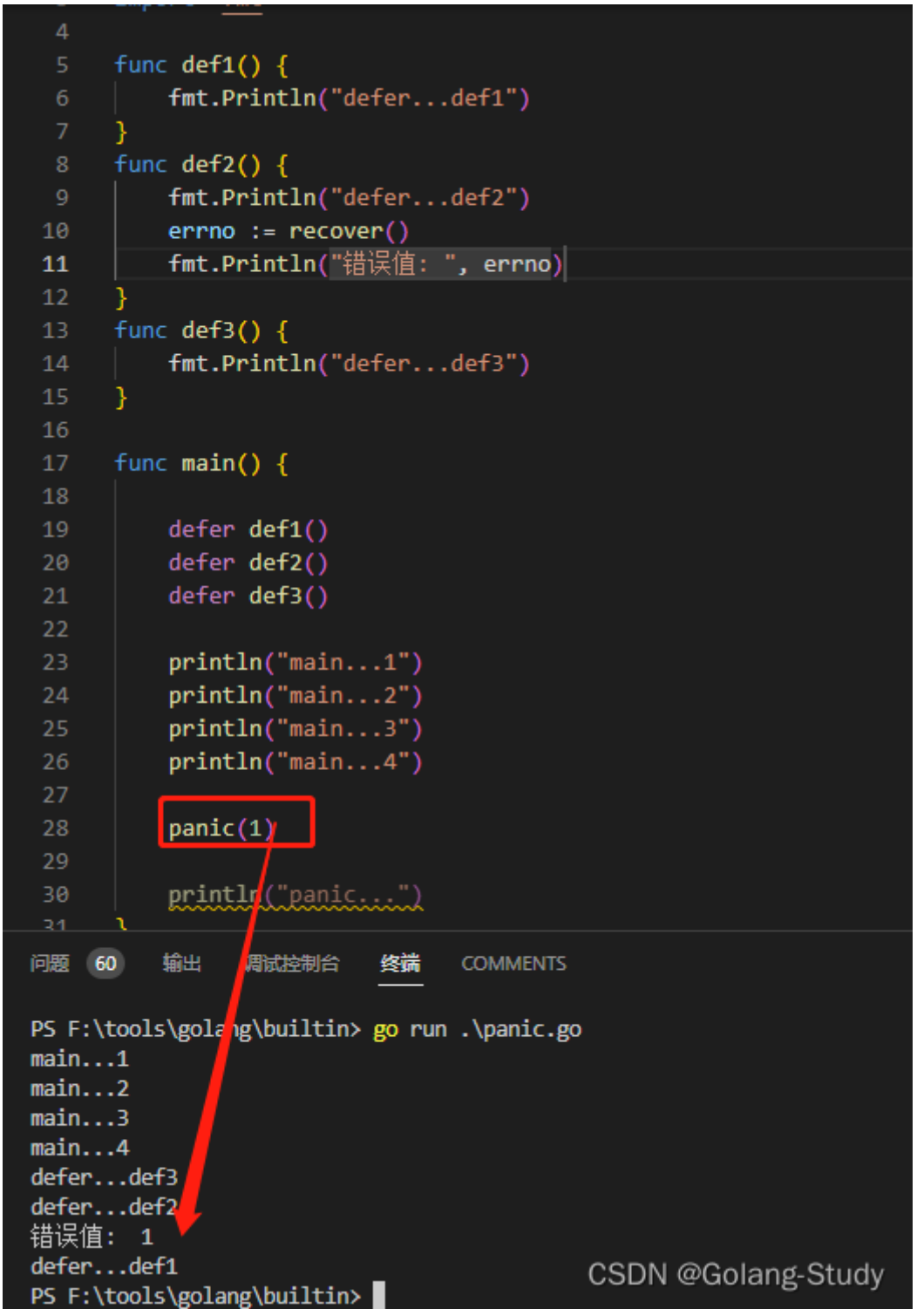
CSDN @Golang-Study

```go
func def1() {
    fmt.Println("defer...def1")
}
func def2() {
    fmt.Println("defer...def2")
}
func def3() {
    fmt.Println("defer...def3")
}

func main() {

    defer def1()
    defer def2()
    defer def3()

    println("main...1")
    println("main...2")
    println("main...3")
    println("main...4")

    panic(1)
```

问题 60 输出 调试控制台 终端 COMMENTS

```
PS F:\tools\golang\builtin> go run .\panic.go
main...1
main...2
main...3
main...4
defer...def3
defer...def2
defer...def1
panic: 1

goroutine 1 [running]:
main.main()
        F:/tools/golang/builtin/panic.go:26 +0xee
exit status 2
PS F:\tools\golang\builtin>
```

CSDN @Golang-Study


func def1() {
fmt.Println("defer...def1")
}
func def2() {
fmt.Println("defer...def2")
errno := recover()
fmt.Println("错误值: ", errno)
}
func def3() {
fmt.Println("defer...def3")
}
func main() {
defer def1()
defer def2()
defer def3()
println("main...1")
println("main...2")
println("main...3")
println("main...4")
panic(1)
println("panic...")
问题 60 输出 调试控制台 终端 COMMENTS
PS F:\tools\golang\builtin> go run .\panic.go
main...1
main...2
main...3
main...4
defer...def3
defer...def2
错误值: 1
defer...def1
PS F:\tools\golang\builtin>
CSDN @Golang-Study


func def2() {
fmt.Println("defer...def2")
errno := recover()
fmt.Println("错误值: ", errno)
}
func def3() {
fmt.Println("defer...def3")
}
func test() {
defer def1()
defer def2()
defer def3()
num1 := 1
num2 := 0
num := num1 / num2
fmt.Println(num)
}
发生恐慌的代码必须和defer在同一个函数中，否则即使处理了恐慌，也会在恐慌处停止执行
CSDN @Golang-Study
main...2
main...3
main...4
defer...def3
defer...def2
错误值: runtime error: integer divide by zero
defer...def1
panic...
CSDN @Golang-Study

9、print和println

func print

```
func print(args ...Type)
```

内建函数print以特有的方法格式化参数并将结果写入标准错误，用于自举和调试。

func println

```
func println(args ...Type)
```

println类似print，但会在参数输出之间添加空格，输出结束后换行。

CSDN @Golang-Study