

# Golang语法细节底层挖掘



Go1ang是一门语法简单但充满了丰富细节的语言

# 举个例子

---

```
a0 := [8]int{789, 0, 0, 0, 0, 0, 0, 0}
```

```
a1 := [8]int{0: 789}
```

```
a2 := [...]int{789, 0, 0, 0, 0, 0, 0, 0}
```

```
a3 := [...]int{0: 789, 7: 0}
```

```
var a4 [8]int
```

```
a4[0] = 789
```

```
a5 := *new([8]int)
```

```
a5[0] = 789
```

```
a6 := [8]int{}
```

```
a6[0] = 789
```

Go1ang中的nil

# Golang中的nil

---

```
func f1() *bool {  
    return nil  
}
```

```
func f2(length int) interface{} {  
    var ss []int = nil  
    if length >= 0 {  
        ss = make([]int, length)  
    }  
    return ss  
}
```

```
func main() {  
    var i1 interface{} = f1()  
    fmt.Println ( i1 == nil ) // false, why?  
    i2 := f2(-1)  
    fmt.Println ( i2 == nil ) // false, why?  
}
```

# Golang中的nil

---

官方文档：下列6大类型家族的零值为nil  
其它类型值都不可能为nil

指针类型 pointer	切片类型 slice	映射表类型 map
数据通道类型 channel	函数类型 function	接口类型 interface

但什么是nil?

nil和很多其它编程语言中的null是一回事儿吗?

# Golang中，nil值在使用时必须要有明确的类型

---

```
// nil为未确定类型值，它有很多可能类型，但是它没有默认类型  
// v := nil // 编译错误
```

```
p := (*struct{}) (nil) // var p *struct{} = nil  
s := []int(nil)        // var s []int = nil  
m := map[int]bool(nil) // var m map[int]bool = nil  
c := chan string(nil)  // var c chan string = nil  
f := (func()) (nil)    // var f func() = nil  
i := interface{}(nil)  // var i interface{} = nil
```

```
fmt.Printf("%#v\n", p) // (*struct {})(nil)  
fmt.Printf("%#v\n", s) // []int(nil)  
fmt.Printf("%#v\n", m) // map[int]bool(nil)  
fmt.Printf("%#v\n", c) // (chan string)(nil)  
fmt.Printf("%#v\n", f) // (func())(nil)  
fmt.Printf("%#v\n", i) // <nil> // 接口类型
```

## 每种类型的nil所占内存大小不一

---

```
var p *struct{} = nil
fmt.Println( unsafe.Sizeof( p ) ) // 8
```

```
var s []int = nil
fmt.Println( unsafe.Sizeof( s ) ) // 24
```

```
var m map[int]bool = nil
fmt.Println( unsafe.Sizeof( m ) ) // 8
```

```
var c chan string = nil
fmt.Println( unsafe.Sizeof( c ) ) // 8
```

```
var f func() = nil
fmt.Println( unsafe.Sizeof( f ) ) // 8
```

```
var i interface{} = nil
fmt.Println( unsafe.Sizeof( i ) ) // 16
```

on  
amd64



## 每种零值为nil的类型的底层结构（一）

类型	类型底层结构
指针	<code>uintptr</code> // 系统相关的无符号整数， // 在amd64上，等同于uint64)
映射表	<code>type mapStruct struct {     m *internalMap }</code>
数据通道	<code>type channelStruct struct {     c *internalChannel }</code>
函数	<code>type functionStruct struct {     f *internalFunction }</code>

## 每种零值为nil的类型的底层结构（二）

类型	类型底层结构
切片	<pre>type sliceStruct struct {     array *internalArray     len    int     cap    int }</pre>
接口	<pre>type interfaceStruct struct {     v *_value // 实际值     t *_type  // 实际值的类型信息 }</pre> <p>// 这只是一个简化定性的描述，实际实现要复杂一些 // v所指的值是用用户程序接触不到的，并且从不改变 // v本身的值在被赋值的时候是可能改变的</p>

Golang中，如果一个类型的值所占内存的每一个字节都是0，则此值即为此类型的零值。

---

```
uintptr(0)                                channelStruct {
                                           c: uintptr(0),
sliceStruct {                             }
    array:
uintptr(0),                               functionStruct {
    len:    0,                             f: uintptr(0),
    cap:    0,                             }
}
                                           mapStruct {
interfaceStruct {                         m: uintptr(0),
    v: uintptr(0),                         }
    t: uintptr(0),
}
```

6大类型家族底层结构的零值如上。

而官方文档：6大类型家族的零值为nil

结论: nil不是一个值, 它可能为下列这些值

---

```
uintptr(0) == nil
```

```
sliceStruct {
```

```
    array:
```

```
    uintptr(0),
```

```
    len:    0,
```

```
    cap:    0,
```

```
} == nil
```

```
interfaceStruct {
```

```
    v: uintptr(0),
```

```
    t: uintptr(0),
```

```
} == nil
```

```
channelStruct {
```

```
    c: uintptr(0),
```

```
} == nil
```

```
functionStruct {
```

```
    f: uintptr(0),
```

```
} == nil
```

```
mapStruct {
```

```
    m: uintptr(0),
```

```
} == nil
```

nil只不过是6大类型家族零值的简称,

否则不太容易描述表达清楚什么是这些类型的零值。

## 回到开始的问题: nil != nil, why?

---

```
var ppp *bool = nil
```


```
var iii interface{} = ppp
```

```
fmt.Println( iii == nil ) // false, why?
```


# nil != nil, why?

---

```
var ppp *bool = nil
    // 等价于 ppp := (*bool)nil
var iii interface{} = ppp
    // 等价于 iii := interface{}(ppp)
    // 或者: iii := interface{}((*bool)nil)
fmt.Println( iii == nil )
    // interface{}((*bool)nil) == interface{}(nil)
```



```
interfaceStruct {
    v: uintptr(0),
    t: (*_type) (*bool),
}
```



```
interfaceStruct {
    v: uintptr(0),
    t: uintptr(0),
}
```

显然两者的t属性不一致，此nil非彼nil

string与[]byte

# string与[]byte

---

内置函数copy和append要求两个参数须为切片并且两个切片的元素类型必须一致。但是，在第一个参数类型为[]byte的情况下，Golang允许第二个参数为string类型。

---

```
// 一个汉字3个byte
bytes := make([]byte, 9)

// copy([]byte, string)
copy(bytes, "西二旗")
// <=> copy(bytes, []byte("西二旗"))

// append([]byte, ...string)
bytes = append(bytes, ", 你早! "... )
// <=> append(bytes, []byte(", 你早! ")...)

fmt.Println(string(bytes)) // 西二旗, 你早!
```



# string与[]byte

---

当使用[]byte(string)或者string([]byte)在string和[]byte之间强制转换的时候，将复制一份底层的byte数组。

但是Golang官方编译器在两种情况下做了优化，在这两种情况下，强制转换并没有复制一份底层的byte数组。

---

```
text := "hello world!"
for i, b := range []byte(text) {
    fmt.Println(i, ":", b)
}
```

因为这里生成[]byte值永远得不到更改的机会，所以此优化是绝对安全的。

```
key := []byte{'k', 'e', 'y'}
m := map[string]string{}
m[string(key)] = "value"
fmt.Println(m[string(key)]) // value
```

即便切片key是一个包级（全局）变量，这里的优化仍将实施。并发程序要注意这点。

类型T和\*T的方法集

对于一个非指针非接口类型T，类型T和\*T的变量值可以互相调用定义在两个类型上的方法

```
type T struct{}
func (t T) f() {}
func (t *T) fp() {}
func main() {
    var t T
    var p = &t

    t.f()
    p.f() // 自动解引用 (*p).f()
         // 等价于 t.f()

    p.fp()
    t.fp() // 自动取地址 (&t).fp()
         // 等价于 p.fp()
}
```

但是Golang规范规定：

定义在类型T上的方法属于类型\*T的方法集中的一员，

反之却不然，

即定义在类型\*T上的方法并不属于类型T的方法集中的一员。

\*T的方法集：{ f, fp }，  
T的方法集：{ f }。

为什么定义在类型T上的方法属于类型\*T的方法集中的一员，但定义在类型\*T上的方法却不属于类型T的方法集合的一员？

---

因为对于一个指针值p，在编译阶段，对它的解引用\*p总是合法的。

但是，对于很多非指针值v，在编译阶段，取它的地址&v却并不总是合法的，所以当v调用\*T的方法时，难以将v自动转换为&v。

类型T的变量值(变量是总可以被取地址的)可以调用定义在类型\*T上的方法，纯粹是为了编程简洁方便，没有更多寓意。

```
// 直接值不能取地址
```

```
_ = &true  
_ = &"abc"  
n := 1
```

```
_ = &(n + 1)  
var i interface{} = n  
_ = &(i.(string))  
_ = &math.Int()
```

```
// 字符串字节元素不能取地址
```

```
s := "hello, world!"  
_ = &(s[5])
```

```
// map元素不能取地址(见后)
```

```
m := map[int]int{99:1}  
_ = &(m[99])
```

# 直接值不能取地址？为什么有很多&T{}这种用法？

```
pm    := &map[string]string{}
ps    := &[]string{}
pst   := &struct{}{}
pmt   := &MyType{}
```

&T{} 是为了编程方便，添加的一个sugar，是下面形式的缩写，而不是临时值不能取地址的一个例外。

```
temp := T{}
&temp
```

```
// 编译器只会自动对变量取地址
// 而不会自动对直接值取地址
```

```
type T struct{}
func (t *T) f() {}
```

```
func main() {
    t := T{}
```

```
    (&t).f() // ok, 和上一句等价
    t.f()    // ok, 将自动取地址
```

```
    (&T{}).f() // ok
    // T{}.f() // error
                // 不会自动取地址
```

```
}
```

Go1ang中map元素属性是只读的

# Golang中的map元素属性是只读的

---

```
type Person struct {
    Age int
}

func (p *Person) GrowUp() {
    p.Age++
}

func main() {
    m := map[string]Person {
        "小明": Person{Age: 18},
    }
    // m["小明"].Age = 19 // 错误: 元素属性不可修改
    // m["小明"].Age++    // 错误: 元素属性不可修改
    // m["小明"].GrowUp() // 错误: 元素不可被取地址
    person := m["小明"] // 临时变量
    person.Age = 19
    m["小明"] = person // 覆盖旧值
}
```

## 切片元素属性却是可以修改的

---

```
type Person struct {  
    Name string  
    Age  int  
}  
  
func (p *Person) GrowUp() {  
    p.Age ++  
}  
  
func main() {  
    s := []Person {  
        Person{Name: "小明", Age: 18},  
    }  
  
    s[0].Age = 19 // ok      ⇔ (&s[0]).Age = 19  
    s[0].Age ++   // 20 now  ⇔ (&s[0]).Age ++  
    s[0].GrowUp() // 21 now  ⇔ (&s[0]).GrowUp()  
}
```



为什么不能直接修改/取址map元素属性？  
但是却可以直接修改/取址slice元素属性？

---

// 取一个下标不合法的slice元素，将运行时panic

```
person = s[999] // panic
```

```
age     = s[999].Age // panic
```

// 但取一个键值key不存在的map元素，将返回一个临时零值

```
person = m["大明"] // p == Person{Age: 0}
```

```
age = m["大明"].Age // age = 0
```

```
// m["大明"].Age ++ // 编译不通过
```

// 假如上一句编译可以通过

```
m["大明"].Age ++ // 0 + 1 → 1
```

// 但是变化仅仅施加在临时零值上

// 键值"大明"对应的元素依然是不存在的

```
age = m["大明"].Age // 依然为0
```

// age依然为0的原因是这里返回的是一个新的临时零值

// => 导致困惑

// 所以干脆禁止修改map元素属性

为什么不能直接修改/取址map元素属性？  
但是却可以直接修改/取址slice元素属性？

---

原因之二：目前Golang中的map实现不能保证元素地址固定不变。内部的hashable在扩容过程中将移动元素位置，所以如果map元素可以取地址，则保存外部指针变量中的地址将可能变为野指针。

map元素取不了地址，就不能更改其属性。

# Golang根据情况将局部变量分配到栈上或者堆上

---

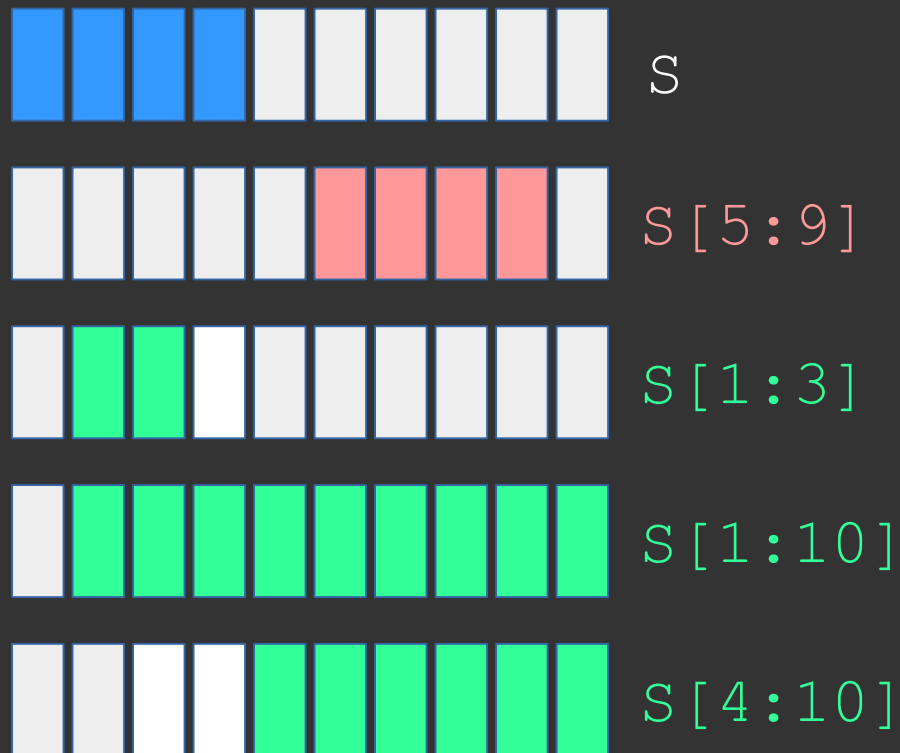
使用`go build -gcflags=-m`查看inline和内存开辟位置。

使用`-gcflags=" -m -m"` 将给出更多信息。

---

```
func f1() *int {  
    n := 100 // 开在堆上  
    return &n  
}  
  
func f2() {  
    for i := 0; i < 100; i++ { // i开在栈上  
        var k = i // k开在堆上  
        go func() {  
            k++ // 100个协程看到的是100个不同的k  
        }()  
    }  
}
```

# 有时子切片的长度可以大于原切片长度



取子切片时，Golang规定：

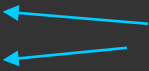
- 起始下标必须小于等于原切片的长度；
- 终止下标必须小于等于原切片的容量（可以大于原切片的长度）。

在Golang当前版本中，只有指针值的读写是跨平台（但不保证跨版本）原子性的，然并卵

```
var gameData *int = nil
```


```
func GameLoad() {  
    d := new(int)  
    *d = 123  
    gameData = d  
}
```

编译器可能会调换  
这两行的执行次序



```
func GameLoop() {  
    if gameData != nil {  
        fmt.Println(*data)  
    }  
    // ...  
}
```

即使data != nil  
\*data也可能为0



必须使用三种数据  
同步机制之一：

- 数据通道
- 加锁
- 原子操作

# 变量函数类型名可以为中文

---

```
const π = 3.1416
type 圆 struct {
    半径 float64
}
func 周长(一个圆 圆) float64 {
    return 2.0 * π * 一个圆.半径
}
func 面积(一个圆 圆) float64 {
    return π * 一个圆.半径 * 一个圆.半径
}
func main() {
    我的圆 := 圆{半径: 1.2}
    fmt.Println("我的圆的周长为: ", 周长(我的圆))
    fmt.Println("我的圆的面积为: ", 面积(我的圆))
}
```

很遗憾，目前中文被视为小写，所以中文开头的资源被不能导出



DataHub  
P2P数据交易平台



DataFoundry  
大数据PaaS平台



猓艺 (TapirGames)  
<http://gfw.tapirgames.com>



# append函数的返回切片值只可以和源切片共享或不共享底层数组

---

这个，资料太多了...

# channel 原理

---

- 每个channel维护3个队列：数据读协程队列，数据写协程队列，和数据缓冲循环队列。
- Golang channel的实现是（读写）事件触发式的，在处理事件时，整个channel数据结构都要加锁。
- 在任何时候，数据读协程队列和数据写协程队列必有一个为空。
- 

不太好讲清楚...

规则：

- \* nil通道
- \* closed的通道
- \* buffered的通道
- \* ...

# channel原理 / select-case原理

---

- 目前select-case的实现中，需要大量同时对涉及到的所有channel进行加锁操作和加入/移出队列操作，因此如果一个select块设计到的channel比较多，程序执行效率会很受影响。
- 一般来说，一个channel的协程队列遵循先入先出原则，但是select-case导致某些先加入队列的协程后来将被从队列移出取消。
- 
- 
- 
- 
- 
- 
- 不太好讲清楚...

# 类型T和\*T的方法集

---

欲在类型T和\*T上定义一些方法，T必须满足以下一些规则：

- 1) T不是一个无名类型；
- 2) T不是一个包外类型；
- 3) T的底层类型不是一个接口类型；
- 4) T的底层类型不是一个指针类型。

```
func ([]int) f1() {} // []int is an unnamed type
func (chan int) f2() {} // chan int is an unnamed type
func (func()) f3() {} // func() is an unnamed type
func (struct{}) f4() {} // struct {} is an unnamed type
type I interface{}
func (I) f5() {} // I is an interface type
func (string) f6() {} // non-local type string
func (int) f7() {} // non-local type int
type IntPtr *int
func (IntPtr) f8() {} // IntPtr is a pointer type
type MyInt int
func (**MyInt) f9() {} // *MyInt is an unnamed type
```

# Go1ang中的类型转换

---

```
package main

type MyInt int
func fA (i int) { }

type MySlice []int
func fB (is []int) { }

func main() {
    var mi MyInt
    fA(mi) // 编译错误：类型不匹配。不能隐式转换。
    fA(int(mi)) // ok，必须强制转换

    var mis MySlice
    fB(mis) // ok，隐式转换成功
    // 为什么这里没有类型不匹配的编译错误？
}
```

什么时候需要转换？

什么时候可以转换？

什么时候必须显式转换？

什么时候可以隐式转换？

# 什么时候需要类型转换

---

1. 对两个值进行比较(==或!=)但两个值的类型不同的时候，必须将其中一个值转化为另一个值的类型。
2. 赋值但源值和目标值的类型不同（或者源值类型未确定）的时候，必须将源值转化为目标值的类型。

特别地，赋值包括以下场合：

1. 传递参数/保存函数返回值
2. 发送/接收数据
3. {}初始化列表元素，数组和切片下标，map键值，计算表达式中的操作数，等。

(BTW 1，在Golang中，一切赋值都是值复制操作)

(BTW 2，在Golang中，没有其它语言中的引用类型，只有值类型，指针类型是特殊的值类型)

什么时候需要转换？

什么时候可以转换？

什么时候必须显式转换？

什么时候可以隐式转换？



# 有名类型(named type) vs 无名类型(unnamed type)

---

```
bool string
int8/uint8(byte)/int16/uint16/
int32(rune)/uint32/int64/uint64/int/uint
float32/float64/complex64/complex128
```

结构体	struct {...}	数组	[N]T
指针	*T	切片	[]T
映射表	map[T1]T2	数据通道	chan T
函数	func(Ta, Tb, ...) (T1, T2, ...)		
接口	interface{...}		

```
type MyType struct{...}
type MyMap map[int]int
type MySlice []string
type MyFunc func(Ta, Tb, ...) (T1, T2, ...)
```

(简而言之，用一个词表示的类型为有名类型，此词即为类型的名称)

# 类型的底层类型 (underlying type)

---

// 所有内置类型的底层类型均为其本身

bool / string / int / uint / int8 / float32 / ...

数组[5]int / 切片[]bool / 映射表map[string]int

结构体struct{} / 指针\*int / 函数func(int) error

数据通道chan string / 接口interface{f()} / ...

```
type MyInt int; type Age MyInt
```

// Age → MyInt → int // ←底层类型

```
type Map map[int]int; type Table Map
```

// Table → Map → map[int]int // ←底层类型

原则：溯源到内置类型（即无名类型或者内置简单类型）为止。

```
type AgeSlice []Age // []Age和[]int底层类型不一样
```

// AgeSlice → []Age → ~~[]int~~ // []Age ←底层类型

// 这里[]Age已经是一个无名类型，溯源到此为止。

## 直接值 (direct value) vs 变量 (variables)

---

// 直接值包括：无名常量、有名常量、临时值和中间计算结果等。

```
const BBB = true
const NNN = 123
const FFF = 3.1416
const STR = “西二旗”
```

// 变量

```
var succeeded = BBB
iii := NNN
jjj := 5 * iii / 3
aaa := math.Abs(-2.1)
sss := struct {} {}
var zeroSlice []int = nil
var none interface{} = zeroSlice
```

// 一个值要么是直接值，要么是变量。

BTW1: 任何直接值都不能被取地址。

BTW2: 所有变量都可以被取地址。

# 类型未确定值 (untyped value) vs 类型确定值 (typed value)

---

```
type MyBool bool
type MyInt int
type MyFloat float64
type MyString string
type MyStruct struct {}
```

直接值大多都为类型未确定值，除了少量有名常量直接值。

const B, N, F, S = false, 789, 7.89, “西二旗”

true, B	可能类型为bool, MyBool等。默认类型为bool
1.23, F	可能类型为float32, float64, MyFloat等。 默认类型为float64
123, N	可能为各种内置数值类型或MyInt。默认类型为int
“abc”, S	可能类型为string, MyString等， 默认类型为string
struct {} {}	可能类型为struct {}, MyStruct， 默认类型为struct {}
nil	可能类型为各种指针/切片/映射表/函数/ 数据通道/接口类型。nil没有默认类型！

## 类型未确定值 (untyped value) VS 类型确定值 (typed value)

---

```
type MyBool bool
type MyInt int
type MyStruct struct {}
```

所有变量都是类型确定值。  
有名常量也可能为类型确定值。

```
const AGE19 int32 = 19 // AGE19的类型塌缩为int32
```

```
var age = AGE19 // age的类型为AGE19的类型int32
```

```
age1 := int16(AGE19) // age1的类型为int16
```

```
var age2 = 19 // age2的类型为19的默认类型int
```

```
s := struct {} {} // struct {} {}的默认类型struct {}
```

```
var s2 MyStruct = s // s2的类型为MyStruct
```

```
gender1 := MyBool(true) // gender1的类型为MyBool
```

```
var name = “小明” // 类型为”小明”的默认类型string
```

```
var weight float32 = 60.5 // weight的类型为float32
```

```
var nothing = nil // 编译错误: nil没有默认类型
```

什么时候需要转换？

什么时候可以转换？

什么时候必须显式转换？

什么时候可以隐式转换？

# 什么时候可以类型转换(赋值篇)

当两个值类型确定并且两者类型的底层类型相同的时候，两者可以互相转换为对方的类型。只要两者类型中有一个为无名类型，就可以隐式转换；否则在这两个有名类型之间必须显式强制转换。

```
type MyString string;    type Text MyString
var ss string; var ms MyString; var tt Text
```

⊗ `tt = ss; ss = tt` // 编译错误：必须显式强制转换

➡➡ `ss, ms, tt = string(tt), MyString(ms), Text(ss)`

⊗ `ss = string(true)` // 编译错误：底层类型不一样

```
type Age int; type Ages []Age; type Years Ages
var vvv []Age; var aaa Ages; var yyy Years
```

➡➡ `vvv = aaa; yyy = vvv` // 隐式转换ok

⊗ `aaa = yyy; yyy = aaa` // 编译错误：必须显式强制转换

➡➡ `aaa = Ages(yyy); yyy = Years(aaa)`

```
var ints []int; // []Age和[]int底层类型不一样
```

⊗ `vvv = []Age(ints)` // 编译错误

# 什么时候可以类型转换(赋值篇)

源值为一个未确定类型的直接值，并且此直接值的可能类型中包含了目标值类型的时候，可以隐式转换将其转换为目标值类型。

```
type Age int
const NNN = 123 // NNN和123都为类型未确定直接值。
               // 它们的可能类型包括int、Age、float32等
```

```
var iii int; var age Age; var fff float32
```

→ → `iii = NNN; age = 123; fff = NNN`

⊗ `iii = false` // 编译错误: `false`的可能类型不含`int`

```
type Ages []Age
var vvv []Age; var ages Ages
// []Age{}的可能类型为[]Age、Ages等，不包括[]int
```

→ → `vvv = []Age{}; ages = []Age{};`

```
// []int{}的可能类型不含[]Age和Ages
```

⊗ `ages = []int{} // 编译错误`

⊗ `ages = Ages([]int{})` // 即使强制转换也不行



# 什么时候可以类型转换(赋值篇)

当目标值类型为一个接口类型，并且源值的类型实现了此接口类型的时候。此时源值可以隐式转换为此接口类型。

```
type I interface {f(); g();}
type T struct {}
func (t T) f() {}
func (t T) g() {}
var t T
```

➡➡ `var i0 I = t // 因为T实现了I`

// 任何类型都实现了interface{}类型，包括接口I类型。

```
b, k, s := true, 12, "ab" // bool, int, string
```

➡➡ `var i1, i2, i3 interface{} = b, k, s`

➡➡ `var i4, i5, i6 interface{} = t, i0, i1`

// bool, int, string, interface{} 都没有实现I接口

⊗ `i0 = b; i0 = k; i0 = s; i0 = i1 // 编译错误`

# 什么时候可以类型转换(赋值篇)

---

特例1: 数值类型（各种整数，各种浮点数）的变量值之间可以互相转化。

特例2: `string`和`[]byte`类型的值之间可以互相转化；  
`string`和`[]rune`类型的值之间可以互相转化。

特例3: 数值类型（各种整数，各种浮点数）可以单方向转换为`string`。

以上特例均需显式强制转换。

特例 4 : `byte`和`uint8`是一个类型，它们的值之间不需转换，或者说它们的值之间可以互相隐式转换。

特例 5 : `rune`和`int32`是一个类型，它们的值之间不需转换，或者说它们的值之间可以互相隐式转换。

或许还有漏掉的特例...

# 什么时候可以类型转换(比较篇)

---

(两个非接口值比较时，将逐字节比较两者所占用的内存。关于两个接口值的比较，见后。)

两个值的是否可以比较，取决于两个值的类型是否相同或者是否有一个可以隐式转换为另一个值的类型。关于是否可以隐式转换，参见前面针对赋值所列出的规则。

几种一个比较值可以隐式转换为另一个比较值的类型的情形：

- 两个值的底层类型一致，并且其中一个值的类型为无名类型。  
(实际上此时并不需要转换)
- 两个值中的一个类型确定，一个类型不确定，类型不确定的值隐式转换为类型确定值的类型。
- 两个值中的一个类型实现了另一个接口类型，实现类型的值隐式转换为接口类型。
- 还有一种只针对比较的特殊情形：两个值都为类型不确定值，这时只有在两个值的可能类型有交集的情况下才允许比较，此种情形没有太大实用价值。

# 什么时候可以类型转换(比较篇)

---

下列类型的值不支持比较（截至Go1.7）：

- 函数 `func(...)(...)`
- 切片 `[]T`
- 映射表 `map[Tkey]Tvalue`

但是上述类型值可以（仅能）和各自的零值`nil`比较。

包含上述类型属性或者元素的类型也不支持比较。比如：

- `[5][]int`
- `struct {f func() }`
- `chan map[string]bool`

上述不支持比较类型不能用作映射表`map`的键值类型`Tkey`。

包含上述不支持比较类型的值的接口值在比较时(或者用作一个映射表的键值时)，在编译阶段难以侦测此种情况，所以将在运行时产生`panic`。

# 接口类型值比较规则

前面提到：两个非接口值比较时，将逐字节比较两者所占用的内存。  
两个接口值比较时，规则略微复杂。

```
type interfaceStruct struct { // 接口类型底层结构
    v *_value // 实际值
    t *_type  // 实际值的类型
}
```

对于两个接口值a和b，只要它们满足下列条件之一，即可比较：1) 两者中至少有一个为nil接口值；  
2) 否则，它们的接口类型定义的方法集属于超集/子集的关系。

如果两个接口值a和b可以比较，只有在下列情况下两者才相等：

- 1) 如果两者皆为nil接口值，则两者相等；
- 2) 如果两者皆不为nil接口值，并且两者的t属性相同，  
并并且两者的v属性都是空指针；
- 3) 如果两者皆不为nil接口值，并且两者的t属性相同，  
并并且的v属性都不为空，并并且 \*a.v == \*b.v。

Golang中，如果两个同类型值相等，它们所占内存每个字节都相等

```
var p *struct{}
fmt.Println( p == nil ) // true
var s []int
fmt.Println( s == nil ) // true
var m map[int]bool
fmt.Println( m == nil ) // true
var c chan string
fmt.Println( c == nil ) // true
var f func()
fmt.Println( f == nil ) // true
var i interface{}
fmt.Println( i == nil ) // true
fmt.Println( *new(*struct{}) == nil ) // true
fmt.Println( *new([]int) == nil ) // true
fmt.Println( *new(map[int]bool) == nil ) // true
fmt.Println( *new(chan string) == nil ) // true
fmt.Println( *new(func()) == nil ) // true
fmt.Println( *new(interface{}) == nil ) // true
```

Golang中，未指定初始值的变量的值为其类型的零值。

虽然切片/映射表/函数类型不支持比较，但是它们的值可以和nil比较。

## BTW: nil甚至不是关键字

---

```
package main

import "fmt"

func main() {
    nil := 123
    fmt.Println(nil) // 123

    // nil的意义改变了，下面均编译出错：
    // var _ []bool = nil
    // var _ map[string]bool = nil
    // var _ interface{} = nil

    // 类似的还有iota
}
```

# nil != nil, why?

---

```
var ppp *int = nil
```

```
var iii interface{} = ppp
```

```
fmt.Println( ppp == nil ) // true
```

```
fmt.Println( iii == ppp ) // true
```

```
fmt.Println( iii == nil ) // false, why?
```

```
fmt.Println( ppp == interface{}(nil) ) // false, ?
```



# nil != nil, why?

---

```
var ppp *int = nil
    // 类型未确定值nil被转换为(*int)(nil)
var iii interface{} = ppp
    // ppp将转换为interface{}(ppp)

fmt.Println( ppp == nil ) // true
    // 比较时，类型未确定值nil将转换为(*int)(nil)

fmt.Println( iii == ppp ) // true
    // 比较时，ppp将转换为interface{}(ppp)

fmt.Println( iii == nil ) // false, why?
    // 比较时，nil将转换为interface{}(nil)

fmt.Println( ppp == interface{}(nil) ) // false, ?
    // 比较时，ppp将转换为interface{}(ppp)
```

interface{} ( (\*int)(nil) ) != interface{}(nil)

# nil != nil, why?

---

```
type interfaceStruct struct { // 接口类型底层结构
    v *_value // 实际值
    t *_type  // 实际值的类型
}
```

```
// interface{} ( (*int) (nil) )
interfaceStruct {
    v: uintptr(0),
    t: (*_type) (*int), // 伪代码
}
```

// 两者的t属性不一致

```
// interface{} (nil)
interfaceStruct {
    v: uintptr(0),
    t: uintptr(0),
}
```

```
interface{} ( (*int) (nil) ) != interface{} (nil)
```

数组和切片下标越界检查优化 (Go 1.7)

# 数组和切片下标越界检查优化(golang1.7)

---

可以使用-B编译参数来关闭下标越界检查

```
go build -gcflags=-B main.go
```

可以使用-d=ssa/check\_bce/debug=1

参数查看哪些代码行需要检查下标越界(1.7 amd64)

```
go build -gcflags="-d=ssa/check_bce/debug=1"
```

---

```
func f1(s []int) {  
    _ = s[0]  
    _ = s[1]  
    _ = s[2]  
}
```

```
func f2(s []int) {  
    _ = s[2]  
    _ = s[1]  
    _ = s[0]  
}
```

# 数组和切片下标越界检查优化(golang1.7)

---

```
func f3(s []int) {  
    for i := 0; i < len(s); i++ {  
        _ = s[i]  
        _ = s[i:len(s)]  
        _ = s[:i+1]  
    }  
}
```

```
func f4(s []int) {  
    for i := range s {  
        _ = s[i]  
        _ = s[i:len(s)]  
        _ = s[:i+1]  
    }  
}
```

```
func f5(s []int) {  
    for i := len(s) - 1; i >= 0; i-- {  
        _ = s[i] // not smart enough?  
        _ = s[i:len(s)]  
    }  
}
```

# 数组和切片下标越界检查优化(golang1.7)

---

```
func f6(s []int) {  
    if len(s) > 2 {  
        _, _, _ = s[0], s[1], s[2]  
    }  
}
```

```
func f7(s []int, index int) {  
    if index >= 0 && index < len(s) {  
        _ = s[index]  
        _ = s[index:len(s)]  
    }  
}
```

```
func f8(a [5]int) {  
    _ = a[4]  
}
```

```
func f9(s []int, index int) {  
    _ = s[index]  
    _ = s[index]  
}
```

# 数组和切片下标越界检查优化(golang1.7)

---

```
func f10(s []int, index int) {  
    _ = s[:index]  
    _ = s[index:] // not smart enough?  
}
```

```
func f11(index int) {  
    s := []int{0, 1, 2, 3, 4, 5, 6}  
    _ = s[:index]  
    _ = s[index:]  
}
```

```
func f12(index int) {  
    s := []int{0, 1, 2, 3, 4, 5, 6}  
    s = s[:4]  
    _ = s[:index]  
    _ = s[index:] // not smart enough?  
}
```

# 数组和切片下标越界检查优化(golang1.7)

---

```
func f10(s []int, index int) {  
    _ = s[:index]  
    _ = s[index:] // not smart enough? No!  
}
```

```
func f11(index int) {  
    s := []int{0, 1, 2, 3, 4, 5, 6}  
    _ = s[:index]  
    _ = s[index:]  
}
```

在确保一个切片的长度和容量相等的情况下，才实施左边中间样列的优化。

```
func f12(index int) {  
    s := []int{0, 1, 2, 3, 4, 5, 6}  
    s = s[:4]  
    _ = s[:index]  
    _ = s[index:] // not smart enough? No!  
}
```



# 数组和切片下标越界检查优化(golang1.7)

---

```
func f10(s []int, index int) {  
    _ = s[:index]  
    _ = s[index:] // not smart enough? No!  
}  
  
// 颠倒两行的顺序  
func f10b(s []int, index int) {  
    _ = s[index:]  
    _ = s[:index] // not smart enough? Yes.  
}
```

## 1.7 BCE还有改进的余地

Go1ang根据情况将局部变量分配到栈上或者堆上

# Go1ang根据情况将局部变量分配到栈上或者堆上

func f3() {  
    for i:=0; i<9; i++ {  
        k := i // k开在  
栈上  
        go func() {  
            \_ = k + k  
        } ()  
    }  
}

func f4() {  
    for i:=0; i<9; i++ {  
        k := i // k开在堆  
上  
        go func() {  
            \_ = k + k  
        } ()  
        k++  
    }  
}

func f5() {  
    for i:=0; i<9; i++ {  
        k := i // k开在  
堆上  
        go func() {  
            \_ = &k  
        } ()  
    }

func f6() {  
    for i:=0; i<9; i++ {  
        k := i // k开在堆  
上  
        go func() {  
            k++  
        } ()  
    }

# Go语言根据情况将局部变量分配到栈上或者堆上

---

But, who cares?

Go语言 运行时完全可以把所有变量都分配到堆上，只不过为了程序优化才保留了栈。

Go运行时的底层实现经常会改变，但底层的改变并不会对上层的规则产生影响。



猢艺 (TapirGames)  
<http://gfw.tapirgames.com>