

[🏠 首页 \(/\)](#) / [网友博文 \(/articles\)](#) / Go 语言中的 Array, Slice, Map 和 Set

Go 语言中的 Array, Slice, Map 和 Set

📅 6月 30 2014 👤 Damon Zhao | 👁 阅读 1671 次 ❤️ 2 人喜欢 💬 1 条评论 (/articles/2335#commentForm) ☆ 收藏

Array(数组)

内部机制

在 Go 语言中数组是固定长度的数据类型，它包含相同类型的连续的元素，这些元素可以是内建类型，像数字和字符串，也可以是结构类型，元素可以通过唯一的索引值访问，从 0 开始。

数组是很有价值的数据结构，因为它的内存分配是连续的，内存连续意味着可是让它在 CPU 缓存中待更久，所以迭代数组和移动元素都会非常迅速。

数组声明和初始化

通过指定数据类型和元素个数(数组长度)来声明数组。

```
// 声明一个长度为5的整数数组
var array [5]int
```

一旦数组被声明了，那么它的数据类型跟长度都不能再被改变。如果你需要更多的元素，那么只能创建一个你想要长度的新的数组，然后把原有数组的元素拷贝过去。

Go 语言中任何变量被声明时，都会被默认初始化为各自类型对应的 0 值，数组当然也不例外。当一个数组被声明时，它里面包含的每个元素都会被初始化为 0 值。

一种快速创建和初始化数组的方法是使用数组字面值。数组字面值允许我们声明我们需要的元素个数并指定数据类型：

```
// 声明一个长度为5的整数数组
// 初始化每个元素
array := [5]int{7, 77, 777, 7777, 77777}
```

如果你把长度写成 ...，Go 编译器将会根据你的元素来推导出长度：

```
// 通过初始化值的个数来推导出数组容量
array := [...]int{7, 77, 777, 7777, 77777}
```

如果我们知道想要数组的长度，但是希望对指定位置元素初始化，可以这样：

```
// 声明一个长度为5的整数数组
// 为索引为1和2的位置指定元素初始化
// 剩余元素为0值
array := [5]int{1: 77, 2: 777}
```

使用数组

使用 `[]` 操作符来访问数组元素：

```
array := [5]int{7, 77, 777, 7777, 77777}
// 改变索引为2的元素的值
array[2] = 1
```

我们可以定义一个指针数组：

```
array := [5]*int{0: new(int), 1: new(int)}
// 为索引为0和1的元素赋值
*array[0] = 7
*array[1] = 77
```

在 Go 语言中数组是一个值，所以可以用它来进行赋值操作。一个数组可以被赋值给任意相同类型的数组：

```
var array1 [5]string
array2 := [5]string{"Red", "Blue", "Green", "Yellow", "Pink"}
array1 = array2
```

注意数组的类型同时包括数组的长度和可以被存储的元素类型，数组类型完全相同才可以互相赋值，比如下面这样就不可以：

```
var array1 [4]string
array2 := [5]string{"Red", "Blue", "Green", "Yellow", "Pink"}
array1 = array2
// 编译器会报错
Compiler Error:
cannot use array2 (type [5]string) as type [4]string in assignment
```

拷贝一个指针数组实际上是拷贝指针值，而不是指针指向的值：

```
var array1 [3]string
array2 := [3]string{new(string), new(string), new(string)}
*array2[0] = "Red"
*array2[1] = "Blue"
*array2[2] = "Green"
array1 = array2
// 赋值完成后，两组指针数组指向同一字符串
```

多维数组

数组总是一维的，但是可以组合成多维的。多维数组通常用于有父子关系的数据或者是坐标系数据：

```
// 声明一个二维数组
var array [4][2]int
// 使用数组字面值声明并初始化
array := [4][2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}}
// 指定外部数组索引位置初始化
array := [4][2]int{1: {20, 21}, 3: {40, 41}}
// 同时指定内外部数组索引位置初始化
array := [4][2]int{1: {0: 20}, 3: {1: 41}}
```

同样通过 `[]` 操作符来访问数组元素：

```
var array [2][2]int
array[0][0] = 0
array[0][1] = 1
array[1][0] = 2
array[1][1] = 3
```

也同样的相同类型的多维数组可以相互赋值：

```
var array1 = [2][2]int
var array2 = [2][2]int
array[0][0] = 0
array[0][1] = 1
array[1][0] = 2
array[1][1] = 3
array1 = array2
```

因为数组是值，我们可以拷贝单独的维：

```
var array3 [2]int = array1[1]
var value int = array1[1][0]
```

在函数中传递数组

在函数中传递数组是非常昂贵的行为，因为在函数之间传递变量永远是传递值，所以如果变量是数组，那么意味着传递整个数组，即使它很大很大很大。。。

举个栗子，创建一个有百万元素的整形数组，在64位的机器上它需要8兆的内存空间，来看看我们声明它和传递它时发生了什么：

```
var array [1e6]int
foo(array)
func foo(array [1e6]int) {
    ...
}
```

每一次 `foo` 被调用，8兆内存将会被分配在栈上。一旦函数返回，会弹栈并释放内存，每次都需要8兆空间。

Go 语言当然不会这么傻，有更好的方法来在函数中传递数组，那就是传递指向数组的指针，这样每次只需要分配8字节内存：

```
var array [1e6]int
foo(&array)
func foo(array *[1e6]int){
    ...
}
```

但是注意如果你在函数中改变指针指向的值，那么原始数组的值也会被改变。幸运的是 `slice`(切片)可以帮我们处理好这些问题，来一起看看。

Slice(切片)

内部机制和基础

`slice` 是一种可以动态数组，可以按我们的希望增长和收缩。它的增长操作很容易使用，因为有内建的 `append` 方法。我们也可以通过 `relice` 操作化简 `slice`。因为 `slice` 的底层内存是连续分配的，所以 `slice` 的索引，迭代和垃圾回收性能都很好。

`slice` 是对底层数组的抽象和控制。它包含 Go 需要对底层数组管理的三种元数据，分别是：

- 指向底层数组的指针
- `slice` 中元素的长度
- `slice` 的容量(可供增长的最大值)

创建和初始化

Go 中创建 `slice` 有很多种方法，我们一个一个来看。

第一个方法是使用内建的函数 `make`。当我们使用 `make` 创建时，一个选项是可以指定 `slice` 的长度：

```
slice := make([]string, 5)
```

如果只指定了长度，那么容量默认等于长度。我们可以分别指定长度和容量：

```
slice := make([]int, 3, 5)
```

当我们分别指定了长度和容量，我们创建的 **slice** 就可以拥有一开始并没有访问的底层数组的容量。上面代码的 **slice** 中，可以访问3个元素，但是底层数组有5个元素。两个与长度不相干的元素可以被 **slice** 来用。新创建的 **slice** 同样可以共享底层数组和已存在的容量。

不允许创建长度大于容量的 **slice**：

```
slice := make([]int, 5, 3)
Compiler Error:
len larger than cap in make([]int)
```

惯用的创建 **slice** 的方法是使用 **slice** 字面量。跟创建数组很类似，不过不用指定 `[]` 里的值。初始的长度和容量依赖于元素的个数：

```
// 创建一个字符串 slice
// 长度和容量都是 5
slice := []string{"Red", "Blue", "Green", "Yellow", "Pink"}
```

在使用 **slice** 字面量创建 **slice** 时有一种方法可以初始化长度和容量，那就是初始化索引。下面是个例子：

```
// 创建一个字符串 slice
// 初始化一个有100个元素的空的字符串 slice
slice := []string{99: ""}
```

nil 和 empty slice

有的时候我们需要创建一个 **nil slice**，创建一个 **nil slice** 的方法是声明它但不初始化它：

```
var slice []int
```

创建一个 **nil slice** 是创建 **slice** 最基本的方法，很多标准库和内建函数都可以使用它。当我们想要表示一个并不存在的 **slice** 时它变得非常有用，比如一个返回 **slice** 的函数中发生异常的时候。

创建 **empty slice** 的方法就是声明并初始化一下：

```
// 使用 make 创建
silce := make([]int, 0)
// 使用 slice 字面值创建
slice := []int{}
```

empty slice 包含0个元素并且底层数组没有分配存储空间。当我们想要表示一个空集合时它很有用处，比如一个数据库查询返回0个结果。

不管我们用 **nil slice** 还是 **empty slice**，内建函数 **append**，**len**和**cap**的工作方式完全相同。

使用 slice

为一个指定索引值的 **slice** 赋值跟之前数组赋值的做法完全相同。改变单个元素的值使用 `[]` 操作符：

```
slice := []int{10, 20, 30, 40, 50}
slice[1] = 25
```

我们可以在底层数组上对一部分数据进行 **slice** 操作，来创建一个新的 **slice**：

```
// 长度为5，容量为5
slice := []int{10, 20, 30, 40, 50}
// 长度为2，容量为4
newSlice := slice[1:3]
```

在 **slice** 操作之后我们得到了两个 **slice**，它们共享底层数组。但是它们能访问底层数组的范围却不同，**newSlice** 不能访问它头指针前面的值。

计算任意 **new slice** 的长度和容量可以使用下面的公式：

```
对于 slice[i:j] 和底层容量为 k 的数组
长度：j - i
容量：k - i
```

必须再次明确一下现在是两个 **slice** 共享底层数组，因此只要有一个 **slice** 改变了底层数组的值，那么另一个也会随之改变：

```
slice := []int{10, 20, 30, 40, 50}
newSlice := slice[1:3]
newSlice[1] = 35
```

改变 **newSlice** 的第二个元素的值，也会同样改变 **slice** 的第三个元素的值。

一个 **slice** 只能访问它长度范围内的索引，试图访问超出长度范围的索引会产生一个运行时错误。容量只可以用来增长，它只有被合并到长度才可以被访问：

```
slice := []int{10, 20, 30, 40, 50}
newSlice := slice[1:3]
newSlice[3] = 45
Runtime Exception:
panic: runtime error: index out of range
```

容量可以被合并到长度里，通过内建的 **append** 函数。

slice 增长

slice 比 数组的优势就在于它可以按照我们的需要增长，我们只需要使用 **append** 方法，然后 **Go** 会为我们做好一切。

使用 **append** 方法时我们需要一个源 **slice** 和需要附加到它里面的值。当 **append** 方法返回时，它返回一个新的 **slice**，**append** 方法总是增长 **slice** 的长度，另一方面，如果源 **slice** 的容量足够，那么底层数组不会发生改变，否则会重新分配内存空间。

```
// 创建一个长度和容量都为5的 slice
slice := []int{10, 20, 30, 40, 50}
// 创建一个新的 slice
newSlice := slice[1:3]
// 为新的 slice append 一个值
newSlice = append(newSlice, 60)
```

因为 newSlice 有可用的容量，所以在 append 操作之后 slice 索引为 3 的值也变成了 60，之前说过这是因为 slice 和 newSlice 共享同样的底层数组。

如果没有足够可用的容量，append 函数会创建一个新的底层数组，拷贝已存在的值和将要被附加的新值：

```
// 创建长度和容量都为4的 slice
slice := []int{10, 20, 30, 40}
// 附加一个新值到 slice，因为超出了容量，所以会创建新的底层数组
newSlice := append(slice, 50)
```

append 函数重新创建底层数组时，容量会是现有元素的两倍(前提是元素个数小于1000)，如果元素个数超过1000，那么容量会以 1.25 倍来增长。

slice 的第三个索引参数

slice 还可以有第三个索引参数来限定容量，它的目的不是为了增加容量，而是提供了对底层数组的一个保护机制，以方便我们更好的控制 append 操作，举个栗子：

```
source := []string{"apple", "orange", "plum", "banana", "grape"}
// 接着我们在源 slice 之上创建一个新的 slice
slice := source[2:3:4]
```

新创建的 slice 长度为 1，容量为 2，可以看出长度和容量的计算公式也很简单：

```
对于 slice[i:j:k] 或者 [2:3:4]
长度： j - i 或者 3 - 2
容量： k - i 或者 4 - 2
```

如果我们试图设置比可用容量更大的容量，会得到一个运行时错误：

```
slice := source[2:3:6]
Runtime Error:
panic: runtime error: slice bounds out of range
```

限定容量最大的用处是我们在创建新的 slice 时候限定容量与长度相同，这样以后再给新的 slice 增加元素时就会分配新的底层数组，而不会影响原有 slice 的值：

```
source := []string{"apple", "orange", "plum", "banana", "grape"}  
// 接着我们在源 slice 之上创建一个新的 slice  
// 并且设置长度和容量相同  
slice := source[2:3:3]  
// 添加一个新元素  
slice = append(slice, "kiwi")
```

如果没有第三个索引参数限定, 添加 **kiwi** 这个元素时就会覆盖掉 **banana**。

内建函数 **append** 是一个变参函数, 意思就是你可以一次添加多个元素, 比如:

```
s1 := []int{1, 2}  
s2 := []int{3, 4}  
fmt.Printf("%v\n", append(s1, s2...))  
Output:  
[1 2 3 4]
```

迭代 slice

slice 也是一种集合, 所以可以被迭代, 用 **for** 配合 **range** 来迭代:

```
slice := []int{10, 20, 30, 40, 50}  
for index, value := range slice {  
    fmt.Printf("Index: %d Value: %d\n", index, value)  
}  
Output:  
Index: 0 Value: 10  
Index: 1 Value: 20  
Index: 2 Value: 30  
Index: 3 Value: 40  
Index: 4 Value: 50
```

当迭代时 **range** 关键字会返回两个值, 第一个是索引值, 第二个是索引位置值的拷贝。注意: 返回的是值的拷贝而不是引用, 如果我们把值的地址作为指针使用, 会得到一个错误, 来看看为啥:


```
slice := []int{10, 20, 30, 40}
for index, value := range slice {
    fmt.Printf("Value: %d Value-Addr: %X ElemAddr: %X\n", value, &value,
        &slice[index])
}
```

Output:

```
Value: 10 Value-Addr: 10500168 ElemAddr: 1052E100
Value: 20 Value-Addr: 10500168 ElemAddr: 1052E104
Value: 30 Value-Addr: 10500168 ElemAddr: 1052E108
Value: 40 Value-Addr: 10500168 ElemAddr: 1052E10C
```

`value` 变量的地址总是相同的因为它只是包含一个拷贝。如果想得到每个元素的真是地址可以使用 `&slice[index]`。

如果不需要索引值，可以使用 `_` 操作符来忽略它：

```
slice := []int{10, 20, 30, 40}
for _, value := range slice {
    fmt.Printf("Value: %d\n", value)
}
```

Output:

```
Value: 10
Value: 20
Value: 30
Value: 40
```

`range` 总是从开始一次遍历，如果你想控制遍历的`step`，就用传统的 `for` 循环：

```
slice := []int{10, 20, 30, 40}
for index := 2; index < len(slice); index++ {
    fmt.Printf("Index: %d Value: %d\n", index, slice[index])
}
```

Output:

```
Index: 2 Value: 30
Index: 3 Value: 40
```

同数组一样，另外两个内建函数 `len` 和 `cap` 分别返回 `slice` 的长度和容量。

多维 slice

也是同数组一样，`slice` 可以组合为多维的 `slice`：

```
slice := [][]int{{10}, {20, 30}}
```

需要注意的是使用 `append` 方法时的行为，比如我们现在对 `slice[0]` 增加一个元素：

```
slice := []int{{10}, {20, 30}}
slice[0] = append(slice[0], 20)
```

那么只有 `slice[0]` 会重新创建底层数组，`slice[1]` 则不会。

在函数间传递 slice

在函数间传递 `slice` 是很廉价的，因为 `slice` 相当于是指向底层数组的指针，让我们创建一个很大的 `slice` 然后传递给函数调用它：

```
slice := make([]int, 1e6)
slice = foo(slice)
func foo(slice []int) []int {
    ...
    return slice
}
```

在 64 位的机器上，`slice` 需要 24 字节的内存，其中指针部分需要 8 字节，长度和容量也分别需要 8 字节。

Map

内部机制

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

`map` 是一种集合，所以我们可以像迭代数组和 `slice` 那样迭代它。不过，`map` 是无序的，我们无法决定它的返回顺序，这是因为 `map` 是使用 `hash` 表来实现的。

`map` 的 `hash` 表包含了一个桶集合(collection of buckets)。当我们存储，移除或者查找键值对(key/value pair)时，都会从选择一个桶开始。在映射(map)操作过程中，我们会把指定的键值(key)传递给 `hash` 函数(又称散列函数)。`hash` 函数的作用是生成索引，索引均匀的分布在所有可用的桶上。`hash` 表算法详见：July的博客——从头到尾彻底解析 `hash` 表算法

创建和初始化

Go 语言中有多种方法创建和初始化 `map`。我们可以使用内建函数 `make` 也可以使用 `map` 字面值：

```
// 通过 make 来创建
dict := make(map[string]int)
// 通过字面值创建
dict := map[string]string{"Red": "#da1337", "Orange": "#e95a22"}
```

使用字面值是创建 `map` 惯用的方法(为什么不使用`make`)。初始化 `map` 的长度依赖于键值对的数量。

`map` 的键可以是任意内建类型或者是 `struct` 类型，`map` 的值可以是使用 `==` 操作符的表达式。`slice`，`function` 和 包含 `slice` 的 `struct` 类型不可以作为 `map` 的键，否则会编译错误：

```
dict := map[[]string]int{}  
Compiler Exception:  
invalid map key type []string
```

使用 map

给 map 赋值就是指指定合法类型的键，然后把值赋给键：

```
colors := map[string]string{}  
colors["Red"] = "#da1337"
```

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，否则会报运行时错误：

```
var colors map[string]string  
colors["Red"] = "#da1337"  
Runtime Error:  
panic: runtime error: assignment to entry in nil map
```

测试 map 的键是否存在是 map 操作的重要部分，因为它可以让我们判断是否可以执行一个操作，或者是往 map 里缓存一个值。它也可以被用来比较两个 map 的键值对是否匹配或者缺失。

从 map 里检索一个值有两种选择，我们可以同时检索值并且判断键是否存在：

```
value, exists := colors["Blue"]  
if exists {  
    fmt.Println(value)  
}
```

另一种选择是只返回值，然后判断是否是零值来确定键是否存在。但是只有你确定零值是非法值的时候这招才管用：

```
value := colors["Blue"]  
if value != "" {  
    fmt.Println(value)  
}
```

当索引一个 map 取值时它总是会返回一个值，即使键不存在。上面的例子就返回了对应类型的零值。

迭代一个 map 和迭代数组和 slice 是一样的，使用 range 关键字，不过在迭代 map 时我们不使用 index/value 而使用 key/value 结构：

```
colors := map[string]string{  
    "AliceBlue":   "#f0f8ff",  
    "Coral":       "#ff7f50",  
    "DarkGray":    "#a9a9a9",  
    "ForestGreen": "#228b22",  
}  
for key, value := range colors {  
    fmt.Printf("Key: %s Value: %s\n", key, value)  
}
```

如果我们想要从 `map` 中移除一个键值对，使用内建函数 `delete`(要是也能返回移除是否成功就好了，哎。。。):

```
delete(colors, "Coral")
for key, value := range colors {
    fmt.Println("Key: %s Value: %s\n", key, value)
}
```

在函数间传递 map

在函数间传递 `map` 不是传递 `map` 的拷贝。所以如果我们在函数中改变了 `map`，那么所有引用 `map` 的地方都会改变：

```
func main() {
    colors := map[string]string{
        "AliceBlue":    "#f0f8ff",
        "Coral":        "#ff7f50",
        "DarkGray":     "#a9a9a9",
        "ForestGreen": "#228b22",
    }
    for key, value := range colors {
        fmt.Printf("Key: %s Value: %s\n", key, value)
    }
    removeColor(colors, "Coral")
    for key, value := range colors {
        fmt.Printf("Key: %s Value: %s\n", key, value)
    }
}
func removeColor(colors map[string]string, key string) {
    delete(colors, key)
}
```

执行会得到以下结果：

```
Key: AliceBlue Value: #F0F8FF
Key: Coral Value: #FF7F50
Key: DarkGray Value: #A9A9A9
Key: ForestGreen Value: #228B22

Key: AliceBlue Value: #F0F8FF
Key: DarkGray Value: #A9A9A9
Key: ForestGreen Value: #228B22
```

可以看出来传递 `map` 也是十分廉价的，类似 `slice`。

Set

Go 语言本身是不提供 `set` 的，但是我们可以自己实现它，下面就来试试：

```
package main
import(
    "fmt"
    "sync"
```

```
)
type Set struct {
    m map[int]bool
    sync.RWMutex
}
func New() *Set {
    return &Set{
        m: map[int]bool{},
    }
}
func (s *Set) Add(item int) {
    s.Lock()
    defer s.Unlock()
    s.m[item] = true
}
func (s *Set) Remove(item int) {
    s.Lock()
    s.Unlock()
    delete(s.m, item)
}
func (s *Set) Has(item int) bool {
    s.RLock()
    defer s.RUnlock()
    _, ok := s.m[item]
    return ok
}
func (s *Set) Len() int {
    return len(s.List())
}
func (s *Set) Clear() {
    s.Lock()
    defer s.Unlock()
    s.m = map[int]bool{}
}
func (s *Set) IsEmpty() bool {
    if s.Len() == 0 {
        return true
    }
    return false
}
func (s *Set) List() []int {
    s.RLock()
    defer s.RUnlock()
    list := []int{}
    for item := range s.m {
        list = append(list, item)
    }
    return list
}
func main() {
    // 初始化
    s := New()

    s.Add(1)
    s.Add(1)
    s.Add(2)
    s.Clear()
    if s.IsEmpty() {
```

```

    fmt.Println("0 item")
}

s.Add(1)
s.Add(2)
s.Add(3)

if s.Has(2) {
    fmt.Println("2 does exist")
}

s.Remove(2)
s.Remove(3)
fmt.Println("list of all items", S.List())
}

```

注意我们只是使用了 `int` 作为键，你可以自己实现用 `interface{}` 作为键，做成更通用的 `Set`，另外，这个实现是线程安全的。

总结

- 数组是 `slice` 和 `map` 的底层结构。
- `slice` 是 Go 里面惯用的集合数据的方法，`map` 则是用来存储键值对。
- 内建函数 `make` 用来创建 `slice` 和 `map`，并且为它们指定长度和容量等等。`slice` 和 `map` 字面值也可以做同样的事。
- `slice` 有容量的约束，不过可以通过内建函数 `append` 来增加元素。
- `map` 没有容量一说，所以也没有任何增长限制。
- 内建函数 `len` 可以用来获得 `slice` 和 `map` 的长度。
- 内建函数 `cap` 只能作用在 `slice` 上。
- 可以通过组合方式来创建多维数组和 `slice`。`map` 的值可以是 `slice` 或者另一个 `map`。`slice` 不能作为 `map` 的键。
- 在函数之间传递 `slice` 和 `map` 是相当廉价的，因为他们不会传递底层数组的拷贝。

本文来自: *AriesDevil's Blog* ([/wr?u=http://se77en.cc](http://se77en.cc))

感谢作者: *Damon Zhao*

查看原文: *Go 语言中的 Array, Slice, Map 和 Set* (<http://se77en.cc/2014/06/30/farray-slice-map-and-set-in-golang/>)

♡ 2人喜欢

☆ 收藏

(<http://www.jiathis.com/share?uid=1895190>) 0

猜你喜欢

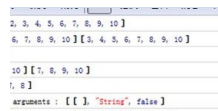
换一换



Go语言核心之美 3.2
—slice切片



Go语言中的 Array,
Slice, Map 和 Set |



Go语言学习笔记(四)
[array、slice、map]



Go数据类型(续)
— array、slice、



Go语言学习笔记1



GO语言基础教程:
array,slice,map

go (golang) 之slice的小想法1(同步sigmentfault)

Go语言中的 Array, Slice和 Map

Golang Array 数组 和 Slice 切片

go (golang) 之slice的小想法1 | Go语言中文网 | Go语言学习园地

[go语言]slice和map | Golang中文社区(Go语言构... Go语言学习园地

[GoLan学习总结]Go语言中的Sli... Go语言中文网 | Go语言学习园地

go基础——03(array、slice、map)


学习Golang语言(6):类型--切片

Go语言_array,slice,map

GO语言基础入门--类型

« (/articles/2334) 上一篇: google hosts 2015 (/articles/2334)
» (/articles/2336) 下一篇: 阿里云上Ubuntu14.04-64位安装Gogs (/articles/2336)

1 条评论



qkb_75_go (/user/qkb_75_go):

表格排版功能很强大呀，所有的表格只有一行内容反复叠加，这是怎么做到的？

(/user/qkb_75_go)

1楼, 2015-02-05 回复

文章点评：（您需要 登录 后才能评论 没有账号 (/user/register) ？）

编辑 预览

我有话要说.....

- 支持 Markdown 格式, ****粗体****、~~~~删除线~~~~、`单行代码`
- 支持 @ 本站用户；支持表情（输入：提示），见 [Emoji cheat sheet \(http://www.emoji-cheat-sheet.com/\)](http://www.emoji-cheat-sheet.com/)

提交

[最新主题 \(/topics\)](/topics) | [最新资源 \(/resources\)](/resources) | [最新评论](#)

- [请教一个动态调整加载css模板思路~ \(/topics/1929\)](/topics/1929)
- [字符串数组传递问题 \(/topics/1928\)](/topics/1928)
- [搭建Spark集群？没想到你是这样的k8s \(/topics/1927\)](/topics/1927)
- [http.ReadRequest后的Body字段为空，请大家帮忙 \(/topics/1926\)](/topics/1926)
- [Go语言的指针类型有什么作用？ \(/topics/1925\)](/topics/1925)
- [golang的服务控制实践 \(/topics/1924\)](/topics/1924)
- [请教Go Thrift 服务端模式的问题 \(/topics/1923\)](/topics/1923)
- [高速数据流存储 \(/topics/1922\)](/topics/1922)
- [类型断言不能操作 \[\]interface{} \(/topics/1921\)](/topics/1921)
- [专访：听道哥聊互联网江湖——探底黑产，描绘作战地图 \(/topics/1920\)](/topics/1920)



对开门冰箱



国外免费聊天室



海尔冰箱价格



在家赚钱的方法

🔥 开源项目 (/projects)



(/p/thyme)
自动追踪使用应用程序的时



(/p/atlantis-docker)
基于 Docker 的开源 PaaS .



(/p/administrative-divisions)
Go 实现的中国行政区划查询



(/p/urllooker)
企业级 URL 监控 URLooker



(/p/unik)
Unikernel 编译和部署平台



关于 (/wiki/about) | API (/api) | 贡献者 (/wiki/contributors) | 帮助推广 (/wiki) | 反馈 (/topics/node/16) | Github (http://github.com/studygolang) | 新浪微博 (http://weibo.com/studygolang) | 内嵌Wide (/wide/playground) | 免责声明 (/wiki/duty)

©2013-2016 studygolang.com 采用 Go语言 (http://golang.org) + MYSQL 构建 (http://www.mysql.com/) 当前在线: 30人 历史最高: 300人 运行时间: 376h0m14.23262567s
网站编译信息 版本: V2.0.0/master-ccf3f3ff2bee3099ad59f5698257b47668f57978, 时间: 2016-07-24 18:11:02.885687475 +0800 CST
Go语言中文网, 中国 Golang 社区, 致力于构建完善的 Golang 中文社区, Go语言爱好者的学习家园。京ICP备14030343号-1



(http://www.ucal.cn?fr=studygolang)

云栖社区

我们的云中江湖
首发战我

(http://click.aliyun.com/m/4526/)



code=3lfz4at7pxfma)

(https://portal.qiniu.com/signup?