Go 博客

Go 切片：用法和本质

2011/01/05

引言

Go的切片类型为处理同类型数据序列提供一个方便而高效的方式。 切片有些类似于其他语言中的数组，但是有一些不同寻常的特性。 本文将深入切片的本质，并讲解它的用法。

数组

Go的切片是在数组之上的抽象数据类型，因此在了解切片之前必须要先理解数组。

数组类型定义了长度和元素类型。例如， `[4]int` 类型表示一个四个整数的数组。 数组的长度是固定的，长度是数组类型的一部分（ `[4]int` 和 `[5]int` 是完全不同的类型）。 数组可以以常规的索引方式访问，表达式 `s[n]` 访问数组的第 n 个元素。

```
var a [4]int
a[0] = 1
i := a[0]
// i == 1
```

数组不需要显式的初始化；数组的零值是可以直接使用的，数组元素会自动初始化为其对应类型的零值：

```
// a[2] == 0, int 类型的零值
```

类型 `[4]int` 对应内存中四个连续的整数：

Go的数组是值语义。一个数组变量表示整个数组，它不是指向第一个元素的指针（不像 C 语言的数组）。 当一个数组变量被赋值或者被传递的时候，实际上会复制整个数组。 （为了避免复制数组，你可以传递一个指向数组的指针，但是数组指针并不是数组。） 可以将数组看作一个特殊的struct，结构的字段名对应数组的索引，同时成员的数目固定。

数组的字面值像这样：

```
b := [2]string{"Penn", "Teller"}
```

当然，也可以让编译器统计数组字面值中元素的数目：

```
b := [...]string{"Penn", "Teller"}
```

这两种写法， `b` 都是对应 `[2]string` 类型。

切片

下一篇

JSON and Go

上一篇

Go: one year ago today

链接

golang.org
golang.org 中文版
安装 Go
Go 指南
Go 文档
Go 邮件列表（英文）
Go 邮件列表（中文）
Go+ 社区
Go 在 Twitter

博客索引

数组虽然有适用它们的地方，但是数组不够灵活，因此在Go代码中数组使用的并不多。 但是，切片则使用得相当广泛。切片基于数组构建，但是提供更强的功能和便利。

切片类型的写法是 `[]T` ， `T` 是切片元素的类型。和数组不同的是，切片类型并没有给定固定的长度。

切片的字面值和数组字面值很像，不过切片没有指定元素个数：

```
letters := []string{"a", "b", "c", "d"}
```

切片可以使用内置函数 `make` 创建，函数签名为：

```
func make([]T, len, cap) []T
```

其中T代表被创建的切片元素的类型。函数 `make` 接受一个类型、一个长度和一个可选的容量参数。 调用 `make` 时，内部会分配一个数组，然后返回数组对应的切片。

```
var s []byte
s = make([]byte, 5, 5)
// s == []byte{0, 0, 0, 0, 0}
```

当容量参数被忽略时，它默认为指定的长度。下面是简洁的写法：

```
s := make([]byte, 5)
```

可以使用内置函数 `len` 和 `cap` 获取切片的长度和容量信息。

```
len(s) == 5
cap(s) == 5
```

接下来的两个小节将讨论长度和容量之间的关系。

切片的零值为 `nil` 。对于切片的零值， `len` 和 `cap` 都将返回0。

切片也可以基于现有的切片或数组生成。切分的范围由两个由冒号分割的索引对应的半开区间指定。 例如，表达式 `b[1:4]` 创建的切片引用数组 `b` 的第1到3个元素空间（对应切片的索引为0到2）。

```
b := []byte{'g', 'o', 'l', 'a', 'n', 'g'}
// b[1:4] == []byte{'o', 'l', 'a'}, sharing the same storage as b
```

切片的开始和结束的索引都是可选的；它们分别默认为零和数组的长度。

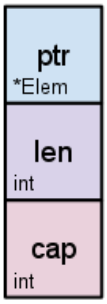
```
// b[:2] == []byte{'g', 'o'}
// b[2:] == []byte{'l', 'a', 'n', 'g'}
// b[:] == b
```

下面语法也是基于数组创建一个切片：

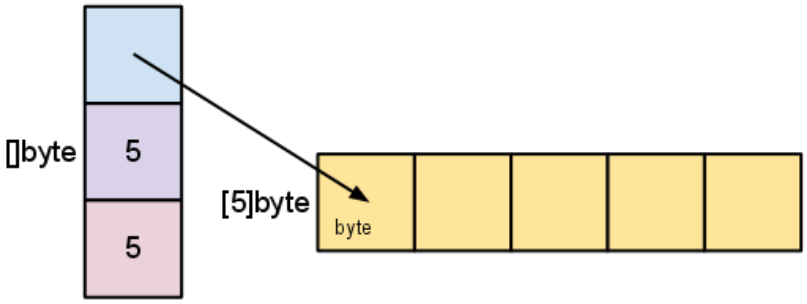
```
x := [3]string{"Лайка", "Белка", "Стрелка"}
s := x[:] // a slice referencing the storage of x
```

切片的内幕

一个切片是一个数组片段的描述。它包含了指向数组的指针，片段的长度， 和容量（片段的最大长度）。



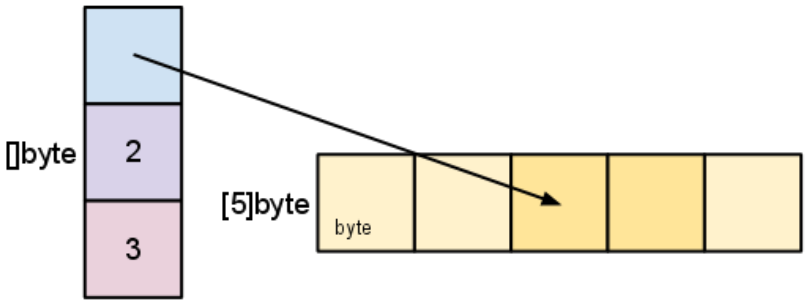
前面使用 `make([]byte, 5)` 创建的切片变量 `s` 的结构如下：



长度是切片引用的元素数目。容量是底层数组的元素数目（从切片指针开始）。 关于长度和容量和区域将在下一个例子说明。

我们继续对 `s` 进行切片，观察切片的数据结构和它引用的底层数组：

```
s = s[2:4]
```

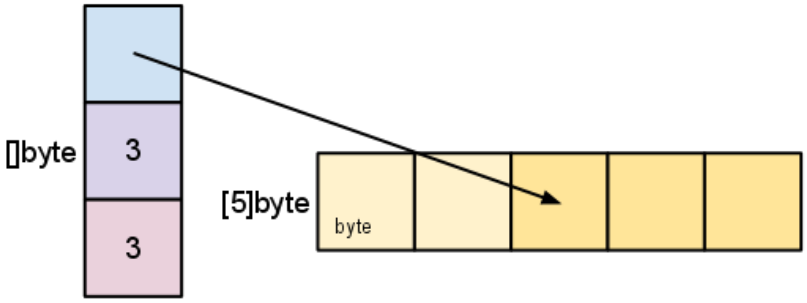


切片操作并不复制切片指向的元素。它创建一个新的切片并复用原来切片的底层数组。 这使得切片操作和数组索引一样高效。因此，通过一个新切片修改元素会影响到原始切片的对应元素。

```
d := []byte{'r', 'o', 'a', 'd'}
e := d[2:]
// e == []byte{'a', 'd'}
e[1] = 'm'
// e == []byte{'a', 'm'}
// d == []byte{'r', 'o', 'a', 'm'}
```

前面创建的切片 `s` 长度小于它的容量。我们可以增长切片的长度为它的容量：

```
s = s[:cap(s)]
```



切片增长不能超出其容量。增长超出切片容量将会导致运行时异常，就像切片或数组的索引超 出范围引起异常一样。同样，不能使用小于零的索引去访问切片之前的元素。

切片的生长（copy and append 函数）

要增加切片的容量必须创建一个新的、更大容量的切片，然后将原有切片的内容复制到新的切片。 整个技术是一些支持动态数组语言的常见实现。下面的例子将切片 `s` 容量翻倍，先创建一个2倍 容量的新切片 `t` ， 复制 `s` 的元素到 `t` ， 然后将 `t` 赋值给 `s` ：

```
t := make([]byte, len(s), (cap(s)+1)*2) // +1 in case cap(s) == 0
for i := range s {
        t[i] = s[i]
}
s = t
```

循环中复制的操作可以由 copy 内置函数替代。copy 函数将源切片的元素复制到目的切片。 它返回复制元素的数目。

```
func copy(dst, src []T) int
```

`copy` 函数支持不同长度的切片之间的复制（它只复制较短切片的长度个元素）。 此外， `copy` 函数可以正确处理源和目的切片有重叠的情况。

使用 `copy` 函数，我们可以简化上面的代码片段：

```
t := make([]byte, len(s), (cap(s)+1)*2)
copy(t, s)
s = t
```

一个常见的操作是将数据追加到切片的尾部。下面的函数将元素追加到切片尾部， 必要的话会增加切片的容量，最后返回更新的切片：

```
func AppendByte(slice []byte, data ...byte) []byte {
    m := len(slice)
    n := m + len(data)
    if n > cap(slice) { // if necessary, reallocate
        // allocate double what's needed, for future growth.
        newSlice := make([]byte, (n+1)*2)
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:n]
    copy(slice[m:n], data)
    return slice
}
```

下面是 `AppendByte` 的一种用法：

```
p := []byte{2, 3, 5}
p = AppendByte(p, 7, 11, 13)
// p == []byte{2, 3, 5, 7, 11, 13}
```

类似 AppendByte 的函数比较实用，因为它提供了切片容量增长的完全控制。 根据程序的特点，可能希望分配较小的活较大的块，或则是超过某个大小再分配。

但大多数程序不需要完全的控制，因此Go提供了一个内置函数 append ， 用于大多数场合；它的函数签名：

```
func append(s []T, x ...T) []T
```

append 函数将 x 追加到切片 s 的末尾，并且在必要的时候增加容量。

```
a := make([]int, 1)
// a == []int{0}
a = append(a, 1, 2, 3)
// a == []int{0, 1, 2, 3}
```

如果是要将一个切片追加到另一个切片尾部，需要使用 ... 语法将第2个参数展开为参数列表。

```
a := []string{"John", "Paul"}
b := []string{"George", "Ringo", "Pete"}
a = append(a, b...) // equivalent to "append(a, b[0], b[1], b[2])"
// a == []string{"John", "Paul", "George", "Ringo", "Pete"}
```

由于切片的零值 nil 用起来就像一个长度为零的切片，我们可以声明一个切片变量然后在循环 中向它追加数据：

```
// Filter returns a new slice holding only
// the elements of s that satisfy fn()
func Filter(s []int, fn func(int) bool) []int {
    var p []int // == nil
    for _, v := range s {
        if fn(v) {
            p = append(p, v)
        }
    }
    return p
}
```

可能的“陷阱”

正如前面所说，切片操作并不会复制底层的数组。整个数组将被保存在内存中，直到它不再被引用。 有时候可能会因为一个小的内存引用导致保存所有的数据。

例如， FindDigits 函数加载整个文件到内存，然后搜索第一个连续的数字，最后结果以切片方式返回。

```
var digitRegexp = regexp.MustCompile("[0-9]+")

func FindDigits(filename string) []byte {
    b, _ := ioutil.ReadFile(filename)
    return digitRegexp.Find(b)
}
```

这段代码的行为和描述类似，返回的 []byte 指向保存整个文件的数组。因为切片引用了原始的数组， 导致 GC 不能释放数组的空间；只用到少数几个字节却导致整个文件的内容都一直保存在内存里。

要修复整个问题，可以将感兴趣的数据复制到一个新的切片中：

```
func CopyDigits(filename string) []byte {
    b, _ := ioutil.ReadFile(filename)
    b = digitRegexp.Find(b)
    c := make([]byte, len(b))
    copy(c, b)
    return c
}
```

可以使用 append 实现一个更简洁的版本。这留给读者作为练习。

延伸阅读

实效 Go 编程 包含了对 切片 和 数组 更深入的探讨； Go 编程语言规范 对 切片类型 和 数组类型 以及与它们 相关的 辅助函数 进行了定义。

Andrew Gerrand 编写

相关文章

- HTTP/2 Server Push
- Introducing HTTP Tracing
- Generating code
- Arrays, slices (and strings): The mechanics of 'append'
- Introducing the Go Race Detector
- Go maps in action
- go fmt your code
- 组织 Go 代码
- Debugging Go programs with the GNU Debugger
- The Go image/draw package
- The Go image package
- 反射三法则
- Error handling and Go
- "First Class Functions in Go"
- Profiling Go Programs
- A GIF decoder: an exercise in Go interfaces
- Introducing Gofix
- Godoc: documenting Go code
- Gobs of data
- C? Go? Cgo!
- JSON and Go
- Go Concurrency Patterns: Timing out, moving on
- Defer, Panic, and Recover
- Share Memory By Communicating
- JSON-RPC: a tale of interfaces