Go语言高薪系列 之 切片(Slice)实现

本课作者 | 法师

1、切片原理

1、1 切片的原理

切片（ slice ）是 Go中一种比较特殊的数据结构，这种数据结构更便于使用和管理数据集合。

切片是围绕动态数组的概念构建的，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。

但是很多同学对 slice 的模糊认识，造成认为Go中的切片作为函数参数是地址传递，结果就是在实际开发中碰到很多坑，以至于出现一些莫名奇妙的问题，数组中的数据丢失了。

```
func Demo(slice []int) {
  slice = append(slice, 6, 6, 6)
  fmt.Println("函数中结果: ", slice)
}
func main() {
  //定义一个切片
  slice := []int{1, 2, 3, 4, 5}
  Demo(slice)
  fmt.Println("定义中结果: ", slice)
}
```

结果

```
函数中结果:  [1 2 3 4 5 6 6 6]
定义中结果:  [1 2 3 4 5]
```

在上面代码中并没有将切片（slice）的修改结果在主调函数中显示。

因为Go语言所有函数参数都是值传递。

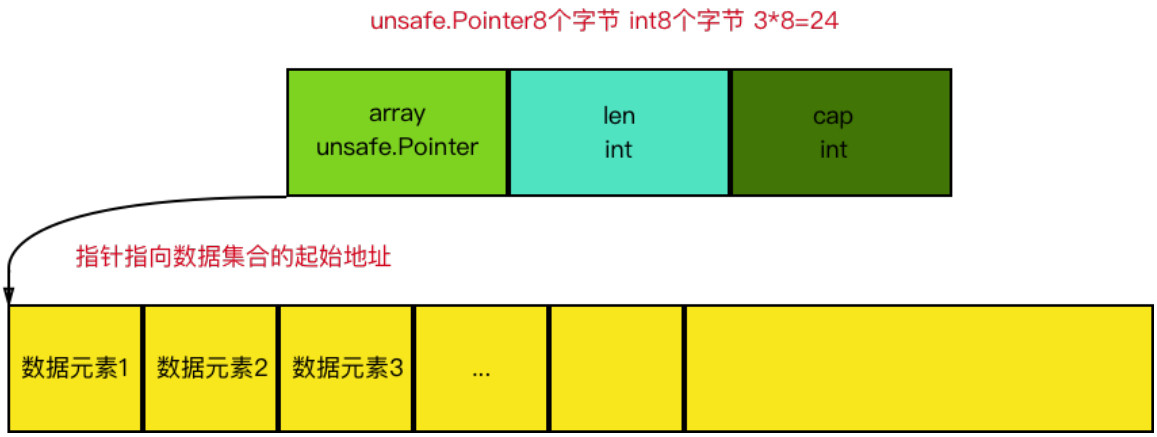
下面就开始详细理解下 slice，理解后会对开发出高效的程序非常有帮助。

```
//定义一个切片
var slice []int
//unsafe.Sizeof功能: 计算一个数据在内存中占的字节大小
size := unsafe.Sizeof(slice)
fmt.Println(slice)
//无论切片是否有数据 计算结果都为: 24
```

slice 的数据结构很简单，一个指向真实数据集合地址的指针ptr，slice 的长度 len 和容量 cap 。

Go语言中切片数据结构在源码包src的 /runtime/slice.go里面：

```go
//path:Go SDK/src/runtime/slice.go
type slice struct {
  array unsafe.Pointer
  len   int
  cap   int
}
```



1、2 切片的源码

在创建切片时，可以根据源码进行分析

```go
var slice []int = make([]int, 10, 20)
```

会被编译器翻译为 runtime.makeslice，并执行如下函数：

```go
func makeslice(et *_type, len, cap int) slice {
  maxElements := maxSliceCap(et.size)
  //判断len是否满足条件
  if len < 0 || uintptr(len) > maxElements {
    panic(errorString("makeslice: len out of range"))
  }
  //判断cap是否满足条件
  if cap < len || uintptr(cap) > maxElements {
    panic(errorString("makeslice: cap out of range"))
  }
  //根据cap大小，通过mallocgc创建内存空间
  p := mallocgc(et.size*uintptr(cap), et, true)
  //将地址作为其中一个结构体成员返回
  return slice{p, len, cap}
}
```

如果创建切片是基于新建内存空间

```go
var slice *[]int = new([]int)
```

会编译为：

```go
func newobject(typ *_type) unsafe.Pointer {
     //创建内存空间 并返回指针
     return mallocgc(typ.size, typ, true)
}
```

注意：mallocgc来自于Go SDK/src/runtime/malloc.go文件。

如果想实现slice开辟内存空间，不能使用mallocgc函数，因为首字母小写，不对外使用。

可以采用C语言和Go语言混合编程，使用C语言函数来完成开辟堆空间操作。

2、CGO混合编程

Go 有强烈的 C 背景，除了语法具有继承性外，其设计者以及其设计目标都与C语言有着千丝万缕的联系。

在Go与C语言互操作方面，Go更是提供了强大的支持。尤其是在Go中使用C，甚至可以直接在Go源文件中编写C代码，这是其他语言所无法望其项背的。

在如下一些场景中，可能会涉及到Go与C的互操作：

- 提升局部代码性能时，用C替换一些Go代码。C之于Go，好比汇编之于C。
- 嫌Go内存GC性能不足，自己手动管理应用内存。
- 实现一些库的Go Wrapper。比如Oracle提供的C版本OCI，但Oracle并未提供Go版本的以及连接DB的协议细节，因此只能通过包装C版本OCI版本的方式以提供Go开发者使用。

前置条件

要使用CGO特性，需要安装C / C构建工具链。

在macOS和Linux下安装GCC，使用命令安装：yum install gcc

在windows下是需要安装MinGW工具 http://www.mingw-w64.org/doku.php。

同时需要保证环境变量CGO_ENABLED被设置为1，这表示CGO是被启用的状态。在本地构建时CGO_ENABLED默认是启用的，当交叉构建时CGO默认是禁止的。

然后通过 import "C" 语句启用CGO特性。

2、1 import "C"

如果在Go代码中出现了 import "C"语句则表示使用了CGO特性，紧跟在这行语句前面的注释是一种特殊语法，里面包含的是正常的C语言代码。

当确保CGO启用的情况下，还可以在当前目录中包含C/C++对应的源文件。

```go
package main

//以块注释的形式写C语言代码

/*
#include <stdio.h>
//C语言函数格式
void Print()
{
 printf("法师好帅~");
}
*/
import "C"

func main() {
  C.Print()
}
```

C语言和Go语言混合编程需要安装GCC，同时编译速度相对会变慢。

2、2 #cgo语句

可以通过#cgo语句 设置编译阶段和链接阶段的相关参数。

编译阶段的参数主要用于定义相关宏和指定头文件检索路径。

链接阶段的参数主要是指定库文件检索路径和要链接的库文件。

```
// #cgo CFLAGS: -D PORT = 8080 -I./include
// #cgo LDFLAGS: -L/usr/local/lib -llibevent
// #include <libevent.h>
import "C"
```

CFLAGS部分，-D 部分定义了宏 PORT，值为8080；-I 定义了头文件包含的检索目录。

LDFLAGS部分，-L指定了链接时库文件检索目录，-l 指定了链接时需要链接libevent库。

2、3 类型转换

在Go语言中访问C语言的符号时，一般是通过虚拟的“C”包访问，比如**C.int**对应C语言的**int**类型。

Go语言中数值类型和C语言数据类型基本上是相似的，以下是对应关系表。

C语言类型	CGO类型	Go语言类型
char	C.char	byte
singed char	C.schar	int8
unsigned char	C.uchar	uint8
short	C.short	int16
unsigned short	C.short	uint16
int	C.int	int32
unsigned int	C.uint	uint32
long	C.long	int32
unsigned long	C.ulong	uint32
long long int	C.longlong	int64
unsigned long long int	C.ulonglong	uint64
float	C.float	float32
double	C.double	float64
size_t	C.size_t	uint

如果Go语言在调用C语言函数时，需要传递参数或者接受返回值，都需要类型转换。

```
package main

//以块注释的形式写C语言代码

/*
#include <stdio.h>
//C语言函数格式
int add(int a, int b)
{
 return a + b;
}
*/
import "C"
import "fmt"

func main() {
  a := 10
```

```
  b := 20
  c := C.add(C.int(a), C.int(b))

  fmt.Println(c)
  fmt.Printf("%T\n", c)
  //转成Go语言整型
  fmt.Println(int(c))
}
```

结果

```
30
main._Ctype_int
30
```

3、切片的实现思路

经过前面两步操作，明白了切片 （slice）的底层原理，同时也了解了CGO混合编程。可以利用C语言相关函数实现切片（slice）。

定义切片结构体：

```
type Slice struct {
  Data unsafe.Pointer //万能指针类型 对应C语言中的void*
  len  int            //有效的长度
  cap  int            //有效的容量
}
```

使用C语言操作内存空间：

```
#include <stdlib.h>
//开辟堆空间 返回值为 void* 指针
malloc(字节大小)
//追加堆空间 实现切片中append的操作函数 返回值为 void* 指针
realloc(指针,字节大小)
//释放堆空间
free(指针)
```

操作函数：

```
//Create(长度 容量 数据)
func (s *Slice) Create(l int, c int, Data ...int)
//Print 打印切片
func (s *Slice) Print()
//Append 切片追加
func (s *Slice) Append(Data ...int)
//GetData 获取元素 GetData(下标)  返回值为int 元素
func (s *Slice) GetData(index int) int
//Search 查找元素 Search(元素)  返回值为int 下标
func (s *Slice) Search(Data int) int
//Delete 删除元素 Delete(下标)
func (s *Slice) Delete(index int)
//Insert 插入元素 Insert(下标 元素)
func (s *Slice) Insert(index int, Data int)
//Destroy 销毁切片
func (s *Slice) Destroy()
```

为了操作方便，以上采用整型进行代码的演示，如果为了切片的适用性，可以采用interface{}。

相当于整型，采用interface{}操作需要通过反射获取类型和值。

```
var i interface{}
i=10//只支持== !=

//类型断言  是基于接口类型数据的转换
value,ok:=i.(int)
if ok{
  fmt.Println("整型数据: ",value)
  fmt.Printf("%T\n",value)
}
//反射获取接口的数据类型
t:=reflect.TypeOf(i)
fmt.Println(t)

//反射获取接口类型数据的值
v:=reflect.ValueOf(i)
fmt.Println(v)
```

4、实例代码

代码实现：

偏移常量

```
//指针操作的偏移值
const TAG = 8
```

C语言代码

```
/*
#include <stdlib.h>
*/
import "C"
```

Create函数

```
//Create(长度 容量 数据)
func (s *Slice) Create(l int, c int, Data ...int) {
  //如果数据为空返回
  if len(Data) == 0 {
    return
  }
  //长度小于0 容量小于0 长度大于容量 数据大于长度
  if l < 0 || c < 0 || l > c || len(Data) > l {
    return
  }
  //ulonglong unsigned long long  无符号的长长整型
  //通过C语言代码开辟空间 存储数据
  //如果堆空间开辟失败 返回值为NULL 相当于nil 内存地址编号为0的空间
  s.Data = C.malloc(C.ulonglong(c) * 8)
  s.len = l
  s.cap = c

  //转成可以计算的指针类型
  p := uintptr(s.Data)
  for _, v := range Data {
    //数据存储
    *(*int)(unsafe.Pointer(p)) = v
    //指针偏移
    p += TAG
    //p+=unsafe.Sizeof(1)
  }
}
```

Print函数

```
//Print 打印切片
func (s *Slice) Print() {
  if s == nil {
    return
  }

  //将万能指针转成可以计算的指针
  p := uintptr(s.Data)
  for i := 0; i < s.len; i++ {
    //获取内存中的数据
    fmt.Print(*(*int)(unsafe.Pointer(p)), " ")
    p += TAG
  }
}
```

Append函数

```
//切片追加
func (s *Slice) Append(Data ...int) {
  if s == nil {
    return
  }
  if len(Data) == 0 {
    return
  }

  //如果添加的数据超出了容量
  if s.len+len(Data) > s.cap {
    //扩充容量
    //C.realloc(指针,字节大小)
    s.Data = C.realloc(s.Data, C.ulonglong(s.cap)*2*8)
    //改变容量的值
    s.cap = s.cap * 2
  }

  p := uintptr(s.Data)
  for i := 0; i < s.len; i++ {
    //指针偏移
    p += TAG
  }

  //添加数据
  for _, v := range Data {
    *(*int)(unsafe.Pointer(p)) = v
    p += TAG
  }
  //更新有效数据（长度）
  s.len = s.len + len(Data)
}
```

GetData函数

```
//获取元素 GetData(下标)  返回值为int 元素
func (s *Slice) GetData(index int) int {
  if s == nil || s.Data == nil {
    return 0
  }
  if index < 0 || index > s.len-1 {
    return 0
  }

  p := uintptr(s.Data)
  for i := 0; i < index; i++ {
    p += TAG
```

```
  }
  return *(*int)(unsafe.Pointer(p))
}
```

Search函数

```
//查找元素 Search(元素)返回值为int 下标
func (s *Slice) Search(Data int) int {
  if s == nil || s.Data == nil {
    return -1
  }

  p := uintptr(s.Data)
  for i := 0; i < s.len; i++ {
    //查找数据  返回第一次元素出现的位置
    if *(*int)(unsafe.Pointer(p)) == Data {
      return i
    }
    //指针偏移
    p += TAG
  }
  return -1
}
```

Delete函数

```
//删除元素 Delete(下标)
func (s *Slice) Delete(index int) {
  if s == nil || s.Data == nil {
    return
  }
  if index < 0 || index > s.len-1 {
    return
  }

  //将指针指向需要删除的下标位置
  p := uintptr(s.Data)
  for i := 0; i < index; i++ {
    p += TAG
  }

  //用下一个指针对应的值为当前指针对应的值进行赋值
  temp := p
  for i := index; i < s.len; i++ {
    temp += TAG
    *(*int)(unsafe.Pointer(p)) = *(*int)(unsafe.Pointer(temp))
    p += TAG
  }

  s.len--
}
```

Insert函数

```
//插入元素 Insert(下标 元素)
func (s *Slice) Insert(index int, Data int) {
  if s == nil || s.Data == nil {
    return
  }
  if index < 0 || index > s.len-1 {
    return
  }

  //调用追加方法
  if index == s.len-1 {
```

```
    s.Append(Data)
    return
  }

  //获取插入数据的位置
  p := uintptr(s.Data)
  for i := 0; i < index; i++ {
    p += TAG
  }
  //获取末尾的指针位置

  temp := uintptr(s.Data)
  temp += TAG * uintptr(s.len)

  //将后面数据依次向后移动
  for i := s.len; i > index; i-- {
    //用前一个数据为当前数据赋值
    *(*int)(unsafe.Pointer(temp)) = *(*int)(unsafe.Pointer(temp - TAG))
    temp -= TAG
  }

  //修改插入下标的数据
  *(*int)(unsafe.Pointer(p)) = Data
  s.len++
}
```

Destroy函数

```
//销毁切片
func (s *Slice) Destroy() {
  //调用C语言  适释放堆空间
  C.free(s.Data)
  s.Data = nil
  s.len = 0
  s.cap = 0
  s = nil
}
```

测试代码

```
package main

import "fmt"

func main() {

  var s Slice
  //创建切片
  s.Create(5, 5, 1, 2, 3, 4, 5)
  s.Append(6, 7, 8)
  s.Append(9,10,11)
  //打印切片
  s.Print()

  fmt.Println("\n长度: ", s.len)
  fmt.Println("容量: ", s.cap)

  //根据下标获取元素
  value:=s.GetData(11)
  fmt.Println("值: ",value)

  //根据元素获取下标
  index:=s.Search(666)
  fmt.Println("下标: ",index)

  //删除元素
  s.Delete(5)
  s.Delete(5)
```

```
  s.Print()
  fmt.Println()

  //插入元素
  s.Insert(8,666)
  s.Print()

  //slice销毁
  fmt.Println(s)
  s.Destroy()
  fmt.Println(s)

}
```