17 SliceHeader：slice 如何高效处理数据？

在[第 4 讲|集合类型：如何正确使用 array、slice 和 map？]中，你已经学习了 slice（切片），并且知道如何使用。这节课我会详细介绍 slice 的原理，带你学习它的底层设计。

数组

在讲 slice 的原理之前，我先来介绍一下数组。几乎所有的编程语言里都存在数组，Go 也不例外。那么为什么 Go 语言除了数组之外又设计了 slice 呢？要想解答这个问题，我们先来了解数组的局限性。

在下面的示例中，a1、a2 是两个定义好的数组，但是它们的类型不一样。变量 a1 的类型是 [1]string，变量 a2 的类型是 [2]string，也就是说数组的大小属于数组类型的一部分，只有数组内部元素类型和大小一致时，这两个数组才是同一类型。

```
a1:=[1]string{"飞雪无情"}
a2:=[2]string{"飞雪无情"}
```

可以总结为，一个数组由两部分构成：数组的大小和数组内的元素类型。

```
//数组结构伪代码表示
array{
  len
  item type
}
```

比如变量 a1 的大小是 1，内部元素的类型是 string，也就是说 a1 最多只能存储 1 个类型为 string 的元素。而 a2 的大小是 2，内部元素的类型也是 string，所以 a2 最多可以存储 2 个类型为 string 的元素。**一旦一个数组被声明，它的大小和内部元素的类型就不能改变**，你不能随意地向数组添加任意多个元素。这是数组的第一个限制。

既然数组的大小是固定的，如果需要使用数组存储大量的数据，就需要提前指定一个合适的大小，比如 10 万，代码如下所示：

```
a10:=[100000]string{"飞雪无情"}
```

这样虽然可以解决问题，但又带来了另外的问题，那就是内存占用。因为在 Go 语言中，函数间的传参是值传递的，数组作为参数在各个函数之间被传递的时候，同样的内容就会被一遍遍地复制，**这就会造成大量的内存浪费**，这是数组的第二个限制。

虽然数组有限制，但是它是 Go 非常重要的底层数据结构，比如 slice 切片的底层数据就存储在数组中。

slice 切片

你已经知道，数组虽然也不错，但是在操作上有不少限制，为了解决这些限制，Go 语言创造了 slice，也就是切片。切片是对数组的抽象和封装，它的底层是一个数组存储所有的元素，但是它可以动态地添加元素，容量不足时还可以自动扩容，你完全可以把切片理解为动态数组。在 Go 语言中，除了明确需要指定长度大小的类型需要数组来完成，大多数情况下都是使用切片的。

动态扩容

通过内置的 append 方法，你可以向一个切片中追加任意多个元素，所以这就可以解决数组的第一个限制。

在下面的示例中，我通过内置的 append 函数为切片 ss 添加了两个字符串，然后返回一个新的切片赋值给 ss。

```
func main() {
   ss:=[]string{"飞雪无情","张三"}
   ss=append(ss,"李四","王五")
   fmt.Println(ss)
}
```

现在运行这段代码，会看到如下打印结果：

```
[飞雪无情 张三 李四 王五]
```

当通过 append 追加元素时，如果切片的容量不够，append 函数会自动扩容。比如上面的例子，我打印出使用 append 前后的切片长度和容量，代码如下：

```
func main() {
   ss:=[]string{"飞雪无情","张三"}
   fmt.Println("切片ss长度为",len(ss),",容量为",cap(ss))
   ss=append(ss,"李四","王五")
```

```
    fmt.Println("切片ss长度为",len(ss),",容量为",cap(ss))
    fmt.Println(ss)
}
```

其中，我通过内置的 len 函数获取切片的长度，通过 cap 函数获取切片的容量。运行这段代码，可以看到打印结果如下：

```
切片ss长度为 2 ,容量为 2
切片ss长度为 4 ,容量为 4
[飞雪无情 张三 李四 王五]
```

在调用 append 之前，容量是 2，调用之后容量是 4，说明自动扩容了。

> 小提示：append 自动扩容的原理是新创建一个底层数组，把原来切片内的元素拷贝到新数组中，然后再返回一个指向新数组的切片。

数据结构

在 Go 语言中，切片其实是一个结构体，它的定义如下所示：

```
type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
```

SliceHeader 是切片在运行时的表现形式，它有三个字段 Data、Len 和 Cap。

1. Data 用来指向存储切片元素的数组。
2. Len 代表切片的长度。
3. Cap 代表切片的容量。

通过这三个字段，就可以把一个数组抽象成一个切片，便于更好的操作，所以不同切片对应的底层 Data 指向的可能是同一个数组。现在通过一个示例来证明，代码如下：

```
func main() {
    a1:=[2]string{"飞雪无情","张三"}
    s1:=a1[0:1]
```

```
    s2:=a1[:]
    //打印出s1和s2的Data值，是一样的
    fmt.Println((*reflect.SliceHeader)(unsafe.Pointer(&s1)).Data)
    fmt.Println((*reflect.SliceHeader)(unsafe.Pointer(&s2)).Data)
}
```

用上节课学习的 unsafe.Pointer 把它们转换为 *reflect.SliceHeader 指针，就可以打印出 Data 的值，打印结果如下所示：

```
824634150744
824634150744
```

你会发现它们是一样的，也就是这两个切片共用一个数组，所以我们在切片赋值、重新进行切片操作时，使用的还是同一个数组，没有复制原来的元素。这样可以减少内存的占用，提高效率。

> 注意：多个切片共用一个底层数组虽然可以减少内存占用，但是如果有一个切片修改内部的元素，其他切片也会受影响。所以在切片作为参数在函数间传递的时候要小心，尽可能不要修改原切片内的元素。

切片的本质是 SliceHeader，又因为函数的参数是值传递，所以传递的是 SliceHeader 的副本，而不是底层数组的副本。这时候切片的优势就体现出来了，因为 SliceHeader 的副本内存占用非常少，即使是一个非常大的切片（底层数组有很多元素），也顶多占用 24 个字节的内存，这就解决了大数组在传参时内存浪费的问题。

> 小提示：SliceHeader 三个字段的类型分别是 uintptr、int 和 int，在 64 位的机器上，这三个字段最多也就是 int64 类型，一个 int64 占 8 个字节，三个 int64 占 24 个字节内存。

要获取切片数据结构的三个字段的值，也可以不使用 SliceHeader，而是完全自定义一个结构体，只要字段和 SliceHeader 一样就可以了。

比如在下面的示例中，通过 unsfe.Pointer 转换成自定义的 *slice 指针，同样可以获取三个字段对应的值，你甚至可以把字段的名称改为 d、l 和 c，也可以达到目的。

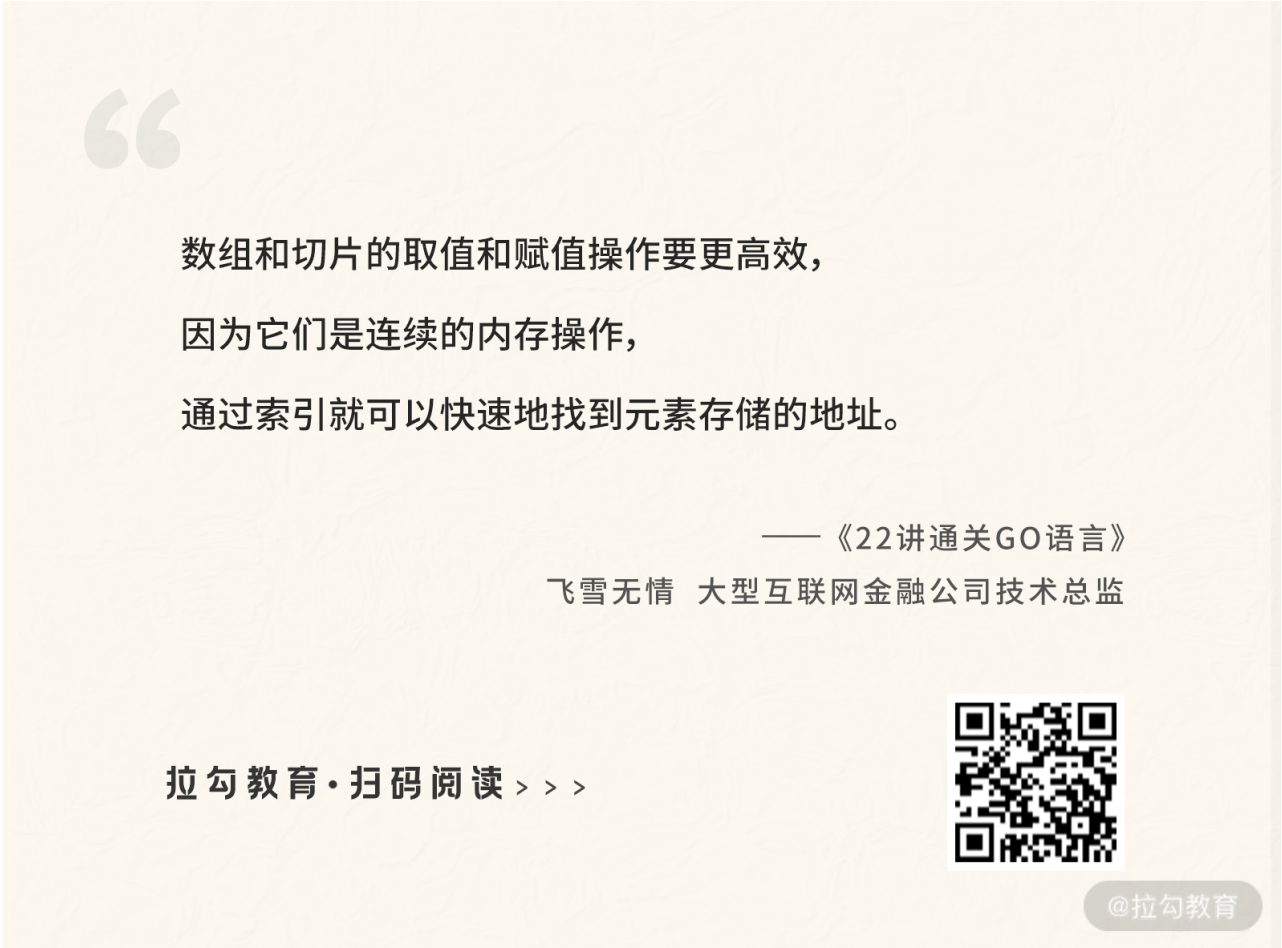
```
sh1:=(*slice)(unsafe.Pointer(&s1))
fmt.Println(sh1.Data,sh1.Len,sh1.Cap)
type slice struct {
    Data uintptr
    Len  int
```

```
    Cap  int
}
```

> 小提示：我们还是尽可能地用 SliceHeader，因为这是 Go 语言提供的标准，可以保持统一，便于理解。

高效的原因

如果从集合类型的角度考虑，数组、切片和 map 都是集合类型，因为它们都可以存放元素，但是数组和切片的取值和赋值操作要更高效，因为它们是连续的内存操作，通过索引就可以快速地找到元素存储的地址。



> 小提示：当然 map 的价值也非常大，因为它的 Key 可以是很多类型，比如 int、int64、string 等，但是数组和切片的索引只能是整数。

进一步对比，在数组和切片中，切片又是高效的，因为它在赋值、函数传参的时候，并不会把所有的元素都复制一遍，而只是复制 SliceHeader 的三个字段就可以了，共用的还是同一个底层数组。

在下面的示例中，我定义了两个函数 arrayF 和 sliceF，分别打印传入的数组和切片底层对应的数组指针。

```
func main() {
   a1:=[2]string{"飞雪无情","张三"}
   fmt.Printf("函数main数组指针：%p\n",&a1)
   arrayF(a1)
   s1:=a1[0:1]
   fmt.Println((*reflect.SliceHeader)(unsafe.Pointer(&s1)).Data)
   sliceF(s1)
}
func arrayF(a [2]string){
   fmt.Printf("函数arrayF数组指针：%p\n",&a)
}
func sliceF(s []string){
   fmt.Printf("函数sliceF Data：%d\n",(*reflect.SliceHeader)(unsafe.Pointer(&
}
```

然后我在 main 函数里调用它们，运行程序会打印如下结果：

```
函数main数组指针：0xc0000a6020
函数arrayF数组指针：0xc0000a6040
824634400800
函数sliceF Data：824634400800
```

你会发现，同一个数组在 main 函数中的指针和在 arrayF 函数中的指针是不一样的，这说明数组在传参的时候被复制了，又产生了一个新数组。而 slice 切片的底层 Data 是一样的，这说明不管是在 main 函数还是 sliceF 函数中，这两个切片共用的还是同一个底层数组，底层数组并没有被复制。

> 小提示：切片的高效还体现在 for range 循环中，因为循环得到的临时变量也是个值拷贝，所以在遍历大的数组时，切片的效率更高。

切片基于指针的封装是它效率高的根本原因，因为可以减少内存的占用，以及减少内存复制时的时间消耗。

string 和 []byte 互转

下面我通过 string 和 []byte 相互强制转换的例子，进一步帮你理解 slice 高效的原因。

比如我把一个 []byte 转为一个 string 字符串，然后再转换回来，示例代码如下：

```
s:="飞雪无情"

b:=[]byte(s)

s3:=string(b)

fmt.Println(s,string(b),s3)
```

在这个示例中，变量 s 是一个 string 字符串，它可以通过 []byte(s) 被强制转换为 []byte 类型的变量 b，又可以通过 string(b) 强制转换为 string 类型的变量 s3。打印它们三个变量的值，都是

“飞雪无情”。

Go 语言通过先分配一个内存再复制内容的方式，实现 string 和 []byte 之间的强制转换。现在我通过 string 和 []byte 指向的真实内容的内存地址，来验证强制转换是采用重新分配内存的方式。如下面的代码所示：

```
s:="飞雪无情"

fmt.Printf("s的内存地址：%d\n", (*reflect.StringHeader)(unsafe.Pointer(&s)).Da

b:=[]byte(s)

fmt.Printf("b的内存地址：%d\n",(*reflect.SliceHeader)(unsafe.Pointer(&b)).Data

s3:=string(b)

fmt.Printf("s3的内存地址：%d\n", (*reflect.StringHeader)(unsafe.Pointer(&s3))
```

运行它们，你会发现打印出的内存地址都不一样，这说明虽然内容相同，但已经不是同一个字符串了，因为内存地址不同。

> 小提示：你可以通过查看 runtime.stringtoslicebyte 和 runtime.slicebytetostring 这两个函数的源代码，了解关于 string 和 []byte 类型互转的具体实现。

通过以上的示例代码，你已经知道了 SliceHeader 是什么。其实 StringHeader 和 SliceHeader 一样，代表的是字符串在程序运行时的真实结构，StringHeader 的定义如下所示：

```
// StringHeader is the runtime representation of a string.
type StringHeader struct {
   Data uintptr
   Len  int
```

```
}
```

也就是说，在程序运行的时候，字符串和切片本质上就是 StringHeader 和 SliceHeader。这两个结构体都有一个 Data 字段，用于存放指向真实内容的指针。所以我们打印出 Data 这个字段的值，就可以判断 string 和 []byte 强制转换后是不是重新分配了内存。

现在你已经知道了 []byte(s) 和 string(b) 这种强制转换会重新拷贝一份字符串，如果字符串非常大，由于内存开销大，对于有高性能要求的程序来说，这种方式就无法满足了，需要进行性能优化。

如何优化呢？既然是因为内存分配导致内存开销大，那么优化的思路应该是在不重新申请内存的情况下实现类型转换。

仔细观察 StringHeader 和 SliceHeader 这两个结构体，会发现它们的前两个字段一模一样，那么 []byte 转 string，就等于通过 unsafe.Pointer 把 *SliceHeader 转为 *StringHeader，也就是 *[]byte 转 *string，原理和我上面讲的把切片转换成一个自定义的 slice 结构体类似。

在下面的示例中，s4 和 s3 的内容是一样的。不一样的是 s4 没有申请新内存（零拷贝），它和变量 b 使用的是同一块内存，因为它们的底层 Data 字段值相同，这样就节约了内存，也达到了 []byte 转 string 的目的。

```
s:="飞雪无情"

b:=[]byte(s)

//s3:=string(b)

s4:=*(*string)(unsafe.Pointer(&b))
```

SliceHeader 有 Data、Len、Cap 三个字段，StringHeader 有 Data、Len 两个字段，所以 *SliceHeader 通过 unsafe.Pointer 转为 *StringHeader 的时候没有问题，因为 *SliceHeader 可以提供 *StringHeader 所需的 Data 和 Len 字段的值。但是反过来却不行了，因为 *StringHeader 缺少 *SliceHeader 所需的 Cap 字段，需要我们自己补上一个默认值。

在下面的示例中，b1 和 b 的内容是一样的，不一样的是 b1 没有申请新内存，而是和变量 s 使用同一块内存，因为它们底层的 Data 字段相同，所以也节约了内存。

```
s:="飞雪无情"

//b:=[]byte(s)

sh:=(*reflect.SliceHeader)(unsafe.Pointer(&s))

sh.Cap = sh.Len

b1:=*(*[]byte)(unsafe.Pointer(sh))
```

> 注意：通过 unsafe.Pointer 把 string 转为 []byte 后，不能对 []byte 修改，比如不可以进行 b1[0]=12 这种操作，会报异常，导致程序崩溃。这是因为在 Go 语言中 string 内存是只读的。

通过 unsafe.Pointer 进行类型转换，避免内存拷贝提升性能的方法在 Go 语言标准库中也有使用，比如 strings.Builder 这个结构体，它内部有 buf 字段存储内容，在通过 String 方法把 []byte 类型的 buf 转为 string 的时候，就使用 unsafe.Pointer 提高了效率，代码如下：

```
// String returns the accumulated string.
func (b *Builder) String() string {
    return *(*string)(unsafe.Pointer(&b.buf))
}
```

string 和 []byte 的互转就是一个很好的利用 SliceHeader 结构体的示例，通过它可以实现零拷贝的类型转换，提升了效率，避免了内存浪费。

总结

通过 slice 切片的分析，相信你可以更深地感受 Go 的魅力，它把底层的指针、数组等进行封装，提供一个切片的概念给开发者，这样既可以方便使用、提高开发效率，又可以提高程序的性能。

Go 语言设计切片的思路非常有借鉴意义，你也可以使用 uintptr 或者 slice 类型的字段来提升性能，就像 Go 语言 SliceHeader 里的 Data uintptr 字段一样。

在这节课的最后，给你留一个思考题：你还可以找到哪些通过 unsafe.Pointer、uintptr 提升性能的例子呢？欢迎留言讨论。

下节课我们将进入工程管理模块，首先学习“质量保证：Go 语言如何通过测试保证质量？”记得来听课！

上一页 下一页