Go底层探索(三):切片

刘庆辉 猿码记 2023-01-31 03:45 Posted on 北京

收录于合集
#Go 101 #Go进阶 14

> **@注: 以下内容来自《Go语言底层原理剖析》、《Go 语言设计与实现》书中的摘要信息，本人使用版本(`Go1.18`)与书中不一致，源码路径可能会有出入。**

1. 介绍

切片是 `Go` 语言中非常常用的数据类型之一，使用方式和数组一样，但是其长度并不固定，我们可以向切片中追加元素，它会在容量不足时自动扩容。

1.1 声明

在 `Go` 语言中，切片类型的声明方式与数组有一些相似，不过由于切片的长度是动态的，所以声明时只需要指定切片中的元素类型：

```
[]int
[]interface{}
```

从切片的定义我们能推测出，切片在编译期间的生成的类型只会包含切片中的元素类型，即 `int` 或者 `interface{}` 等。 `cmd/compile/internal/types.NewSlice` 就是编译期间用于创建切片类型的函数：

```
func NewSlice(elem *Type) *Type {
 if t := elem.cache.slice; t != nil {
  if t.Elem() != elem {
```

```
    base.Fatalf("elem mismatch")
   }
   if elem.HasTParam() != t.HasTParam() || elem.HasShape() != t.HasShape() {
    base.Fatalf("Incorrect HasTParam/HasShape flag for cached slice type")
   }
   return t
  }

  t := newType(TSLICE)
  t.extra = Slice{Elem: elem}
  elem.cache.slice = t
  if elem.HasTParam() {
   t.SetHasTParam(true)
  }
  if elem.HasShape() {
   t.SetHasShape(true)
  }
  return t
 }
```

> 上述方法返回结构体中的 `Extra` 字段是一个只包含切片内元素类型的结构，也就是说切片内元素的类型都是在编译期间确定的，编译器确定了类型之后，会将类型存储在 `Extra` 字段中帮助程序在运行时动态获取。

2. 数据结构

编译期间的切片是 `cmd/compile/internal/types.Slice` 类型，但是在运行时切片会转换成 `reflect.SliceHeader` 结构体，如下:

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

- `Data` 是指向数组的指针;
- `Len` 是当前切片的长度;
- `Cap` 是当前切片的容量，即 `Data` 数组的大小。

2.1 存储原理

`Data` 是一片连续的内存空间，这片内存空间可以用于存储切片中的全部元素，数组中的元素只是逻辑上的概念，底层存储其实都是连续的，**所以我们可以将切片理解成一片连续的内存空间加上长度与容量的标识**。如下图所示:

> 从上图中，我们会发现切片与数组的关系非常密切，切片引入了一个抽象层，提供了对数组中部分连续片段的引用，而作为数组的引用，我们可以在运行区间可以修改它的长度和范围。当切片底层的数组长度不足时就会触发扩容，切片指向的数组可能会发生变化，不过在上层看来切片是没有变化的，上层只需要与切片打交道不需要关心数组的变化。

3. 初始化

Go 语言中包含三种初始化切片的方式:

- 通过下标的方式获得数组或者切片的一部分;
- 使用字面量初始化新的切片;
- 使用关键字 `make` 创建切片:

```
arr[0:3] or slice[0:3]
slice := []int{1, 2, 3}
```

```
slice := make([]int, 10)
```

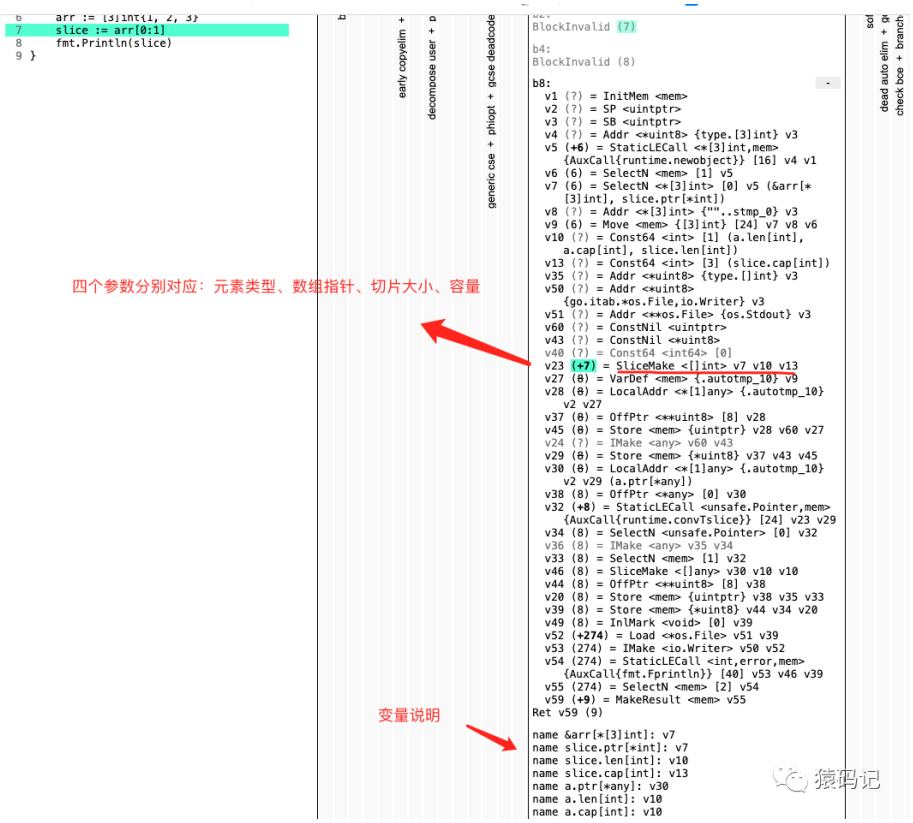
3.1 使用下标

使用下标创建切片是最原始也最接近汇编语言的方式，它是所有方法中最为底层的一种，编译器会将 `arr[0:3]` 或者 `slice[0:3]` 等语句转换成 `OpSliceMake` 操作；我们可以通过下面的代码来验证一下：

```
import "fmt"

func main() {
 arr := [3]int{1, 2, 3}
 slice := arr[0:1]
 fmt.Println(slice)
}
/** 生成SSA
➜  GOSSAFUNC=main go tool compile main.go
dumped SSA to /Users/liuqh/ProjectItem/go-app/ssa.html
*/
```

通过 `GOSSAFUNC` 变量编译上述代码可以得到一系列 `SSA` 中间代码，其中 `slice := arr[0:1]` 语句在 `decompose builtin` 阶段对应的代码如下图所示：



> `SliceMake` 操作会接受四个参数创建新的切片，元素类型、数组指针、切片大小和容量，这也是我们在数据结构一节中提到的切片的几个字段，需要注意的是使用下标初始化切片不会拷贝原数组或者原切片中的数据，它只会创建一个指向原数组的切片结构体，所以修改新切片的数据也会修改原切片。

```
func TestRun(t *testing.T) {
 oldArr := [5]int{1, 2, 3, 4, 5}
 // 使用下标创建切片
 newSlice := oldArr[0:3]
 fmt.Println("修改切片前-> oldArr:", oldArr, "newSlice:", newSlice)
 // 修改新切片值
 newSlice[0] = 100
 fmt.Println("修改切片后-> oldArr:", oldArr, "newSlice:", newSlice)
}
/** 输出
修改切片前-> oldArr: [1 2 3 4 5] newSlice: [1 2 3]
修改切片后-> oldArr: [100 2 3 4 5] newSlice: [100 2 3]
*/
```

3.2 使用字面量

当我们使用字面量 `[]int{1, 2, 3}` 创建新的切片时， `cmd/compile/internal/gc.slicelit` 函数会在编译期间将它展开成如下所示的代码片段：

```
var vstat [3]int
vstat[0] = 1
vstat[1] = 2
vstat[2] = 3
var vauto *[3]int = new([3]int)
*vauto = vstat
slice := vauto[:]
```

对上述代码分析：

1. 根据切片中的元素数量对底层数组的大小进行推断并创建一个数组；
2. 将这些字面量元素存储到初始化的数组中；
3. 创建一个同样指向 `[3]int` 类型的数组指针；
4. 将静态存储区的数组 `vstat` 赋值给 `vauto` 指针所在的地址；
5. 通过 `[:]` 操作获取一个底层使用 `vauto` 的切片；

> 第 5 步中的 `[:]` 就是使用下标创建切片的方法，从这一点我们也能看出 `[:]` 操作是创建切片最底层的一种方法。

3.3 使用make

如果使用字面量的方式创建切片，大部分的工作都会在编译期间完成。但是当我们使用 `make` 关键字创建切片时，很多工作都需要运行时的参与；调用方必须向 `make` 函数传入切片的大小以及可选的容量，

类型检查期间的 `cmd/compile/internal/gc.typecheck1` 函数会校验入参:

```
func typecheck1(n *Node, top int) (res *Node) {
 switch n.Op {
 ...
 case OMAKE:
  args := n.List.Slice()

  i := 1
  switch t.Etype {
     // 类型是切片
  case TSLICE:
   if i >= len(args) {
    yyerror("missing len argument to make(%v)", t)
    return n
   }

   l = args[i]
   i++
   var r *Node
   if i < len(args) {
    r = args[i]
   }
   ...
     // 切片大小len,不能超过容量cap
   if Isconst(l, CTINT) && r != nil && Isconst(r, CTINT) && l.Val().U.(*Mpint).Cmp(r.Val().U.(*Mp
    yyerror("len larger than cap in make(%v)", t)
    return n
```

```
    }

    n.Left = l
    n.Right = r
    n.Op = OMAKESLICE
   }
  ...
 }
}
```

上述函数不仅会检查 `len` 是否传入，还会保证传入的容量 `cap` 一定大于或者等于 `len` 。除了校验参数之外，当前函数会将 `OMAKE` 节点转换成 `OMAKESLICE` ，中间代码生成的 `cmd/compile/internal/gc.walkexpr` 函数会依据下面两个条件转换 `OMAKESLICE` 类型的节点：

1. 切片的大小和容量是否足够小；
2. 切片是否发生了逃逸，最终在堆上初始化；

当切片发生逃逸或者非常大时，运行时需要 `runtime.makeslice` 在堆上初始化切片，如果当前的切片不会发生逃逸并且切片非常小的时候,那么切片运行时最终会被分配在栈中。

> 此临界值定义在 `cmd/compile/internal/ir.MaxImplicitStackVarSize` 变量中，默认为 `64 KB` ，可以通过指定编译时 `smallframes` 标识进行更新，因此， `make([]int64, 1023)` 与 `make([]int64, 1024)` 实现的细节是截然不同的。

4. 扩容原理

4.1 扩容触发

关键字 `append` 是触发切片扩容的主要方式，但不是每次使用 `append` 函数就一定会扩容，看下面代码示例：

```
func main() {
  // --------- 不会扩容 ---------
 // 定义切片a,容量为4
 a := make([]int, 3, 4)
```

```
    fmt.Printf("append前,a-> len:%v cap:%v value:%v \n", len(a), cap(a), a)
    a = append(a, 1)
    fmt.Printf("append后,a-> len:%v cap:%v value:%v \n", len(a), cap(a), a)

    // --------- 会扩容 ---------
    // 定义切片b，容量为3
    b := make([]int, 3, 3)
    fmt.Printf("append前,b-> len:%v cap:%v value:%v \n", len(b), cap(b), b)
    b = append(b, 1)
    fmt.Printf("append后,b-> len:%v cap:%v value:%v \n", len(b), cap(b), b)
}
/** 输出
append前,a-> len:3 cap:4 value:[0 0 0]
append后,a-> len:4 cap:4 value:[0 0 0 1]
append前,b-> len:3 cap:3 value:[0 0 0]
append后,b-> len:4 cap:6 value:[0 0 0 1]
*/
```

代码分析:

- 切片 `a` 容量为4，目前长度为3(初始化3个0)，所以还能再容纳一个元素，在执行 `append` 时，不会执行扩容，所以 `cap` 还是4；
- 切片 `b` 容量为3，目前长度也为3(初始化3个0)，如果再 `append` ,切片容量会不足，便会进行扩容。所以 `cap` 为6，至于为什么是6，看后续源码分析；

4.2 容量判定

当切片的容量不足时， `append` 函数在运行时会调用 `runtime.growslice` 函数为切片扩容；

> 扩容是为切片分配新的内存空间,并拷贝原切片中元素到新空间的过程。

`runtime.growslice` 源码如下:

```
func growslice(et *_type, old slice, cap int) slice {
 newcap := old.cap
 doublecap := newcap + newcap
 if cap > doublecap {
    // 如果新申请容量cap大于2倍的旧容量old.cap，则最终容量newcap是新申请的容量cap
  newcap = cap
 } else {
  if old.len < 1024 {
      // 如果旧切片的长度小于1024，则最终容量是旧容量的2倍
   newcap = doublecap
  } else {
      // 如果旧切片长度大于或等于1024，
   for 0 < newcap && newcap < cap {
        // 则最终容量从旧容量开始循环增加原来的1/4,直到最终容量大于或等于新申请的容量为止
    newcap += newcap / 4
   }
      // 容量计算值溢出,则最终容量就是新申请容量
   if newcap <= 0 {
    newcap = cap
   }
  }
 }
```

上面的代码显示了扩容的核心逻辑， Go 语言中切片扩容的策略整理如下：

- 如果新申请容量（ cap ）大于2倍的旧容量（ old.cap ），则最终容量（ newcap ）是新申请的容量（ cap ）。
- 如果旧切片的长度小于 1024 ，则最终容量是旧容量的2倍，即 newcap=doublecap 。
- 如果旧切片长度大于或等于1024，则最终容量从旧容量开始循环增加原来的 1/4 ,直到最终容量大于或等于新申请的容量为止,即 newcap ≥ cap 。
- 如果最终容量计算值溢出，即超过了int的最大范围，则最终容量就是新申请容量。

> growslice 函数会根据切片的类型，分配不同大小的内存。为了对齐内存，申请的内存可能大于 实际的类型大小×容量大小 。

5.1 代码示例

先看一段代码，试着分析会输出什么?

```
func TestCopySlice(t *testing.T) {
 // 定义切片
 oldSli := []int{100, 200, 300}
 // 复制切片
 copySli := oldSli
 fmt.Println("修改前->copySli:", copySli, "oldSli:", oldSli)
 // 修改复制后的切片
 copySli[0] = 99
 fmt.Println("修改后->copySli:", copySli, "oldSli:", oldSli)
}
```

输出:

```
// 修改前->copySli: [100 200 300] oldSli: [100 200 300]
// 修改后->copySli: [99 200 300] oldSli: [99 200 300]
```

切片的复制其实也是值复制，但这里的值复制指对于运行时 `SliceHeader` 结构的复制,但底层指针仍然指向相同的底层数据的数组地址，因此可以理解为数据进行了引用传递。如下图所示:

> 切片的这一特性使得即便切片中有大量数据，在复制时的成本也比较小，这与数组有显著的不同。

5.2 使用Copy

复制的切片不会改变指向底层的数据源，但有些时候我们希望建一个新的数组，**并且与旧数组不共享相同的数据源**,这时可以使用 `copy` 函数。

还是上面的示例,修改使用 `copy` ：

```
func TestCopySlice(t *testing.T) {
 // 定义切片
 oldSli := []int{100, 200, 300}
 // 使用copy复制切片
 copySli := make([]int, len(oldSli), cap(oldSli))
 copy(copySli, oldSli)
 fmt.Println("修改前->copySli:", copySli, "oldSli:", oldSli)
 // 修改复制后的切片
 copySli[0] = 99
 fmt.Println("修改后->copySli:", copySli, "oldSli:", oldSli)
}
```

输出:

```
// 修改前->copySli: [100 200 300] oldSli: [100 200 300]
// 修改后->copySli: [99 200 300] oldSli: [100 200 300]
```

> `copy` 函数在运行时主要调用了 `memmove` 函数，用于实现内存的复制。如果采用协程调用的方式 `go copy (numbers1, numbers)` 或者加入了 `race` 检测，则会转而调用运行时 `runtime.slicecopy` 函数；无论是编译期间拷贝还是运行时拷贝，两种拷贝方式都会通过 `runtime.memmove` 将整块内存的内容拷贝到目标的内存区域中