### 切片的数据结构

切片本身并不是动态数组或者数组指针。它内部实现的数据结构通过指针引用底层数组，设定相关属性将数据读写操作限定在指定的区域内。切片本身是一个只读对象，其工作机制类似数组指针的一种封装。

切片（slice）是对数组一个连续片段的引用，所以切片是一个引用类型（因此更类似于 C/C++ 中的数组类型，或者 Python 中的 list 类型）。这个片段可以是整个数组，或者是由起始和终止索引标识的一些项的子集。需要注意的是，终止索引标识的项不包括在切片内。切片提供了一个与指向数组的动态窗口。

给定项的切片索引可能比相关数组的相同元素的索引小。和数组不同的是，切片的长度可以在运行时修改，最小为 0 最大为相关数组的长度：切片是一个长度可变的数组。

Slice 的数据结构定义如下:

type slice **struct** {

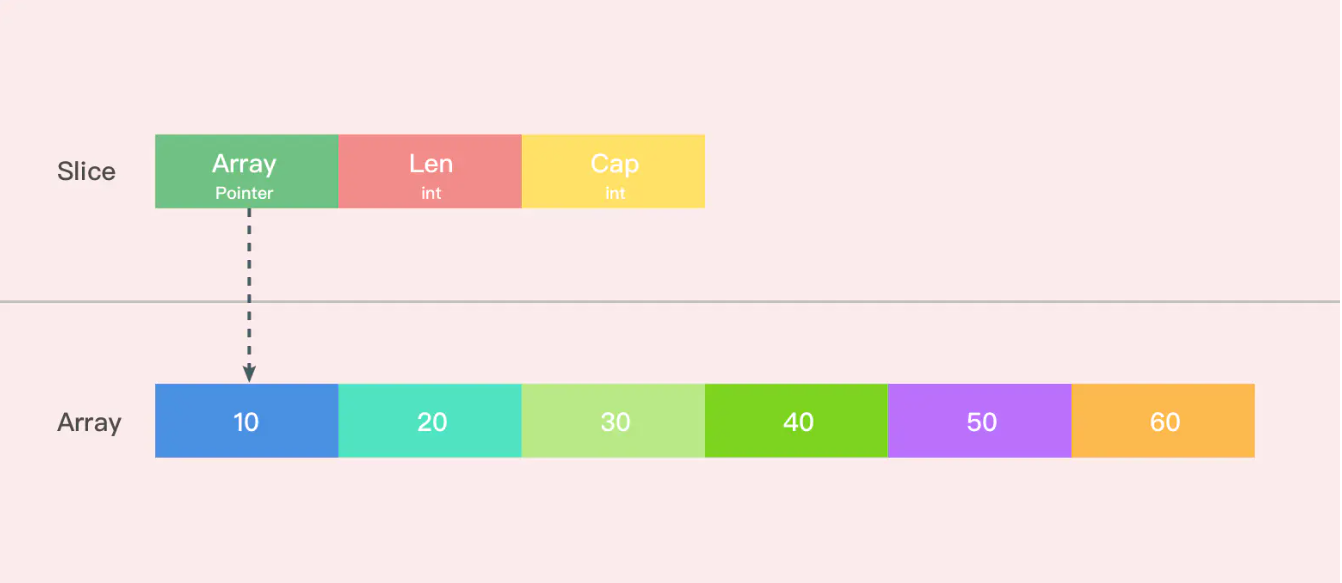
array unsafe.Pointer

len **int**

cap **int**

}

切片的结构体由3部分构成，Pointer 是指向一个数组的指针，len 代表当前切片的长度，cap 是当前切片的容量。cap 总是大于等于 len 的。

如果想从 slice 中得到一块内存地址，可以这样做：

s := make([]**byte**, 200)

ptr := **unsafe**.Pointer(&s[0])

如果反过来呢？从 Go 的内存地址中构造一个 slice

**var** ptr **unsafe**.Pointer

**var** s1 = **struct** {

addr uintptr

len **int**

cap **int**

}{ptr, length, length}

s := \*(\*[]**byte**)(**unsafe**.Pointer(&s1))

构造一个虚拟的结构体，把 slice 的数据结构拼出来。

当然还有更加直接的方法，在 Go 的反射中就存在一个与之对应的数据结构 SliceHeader，我们可以用它来构造一个 slice

**var** o []**byte**

sliceHeader := (\*reflect.SliceHeader)((**unsafe**.Pointer(&o)))

sliceHeader.Cap = length

sliceHeader.Len = length

sliceHeader.Data = uintptr(ptr)

### 创建切片

make 函数允许在运行期动态指定数组长度，绕开了数组类型必须使用编译期常量的限制。

创建切片有两种形式，make 创建切片，空切片。

#### make 和切片字面量

func **makeslice**(et \*\_type, len, cap **int**) slice {

*// 根据切片的数据类型，获取切片的最大容量*

maxElements := maxSliceCap(et.size)

*// 比较切片的长度，长度值域应该在[0,maxElements]之间*

**if** len < 0 || uintptr(len) > maxElements {

panic(errorString("makeslice: len out of range"))

}

*// 比较切片的容量，容量值域应该在[len,maxElements]之间*

**if** cap < len || uintptr(cap) > maxElements {

panic(errorString("makeslice: cap out of range"))

}

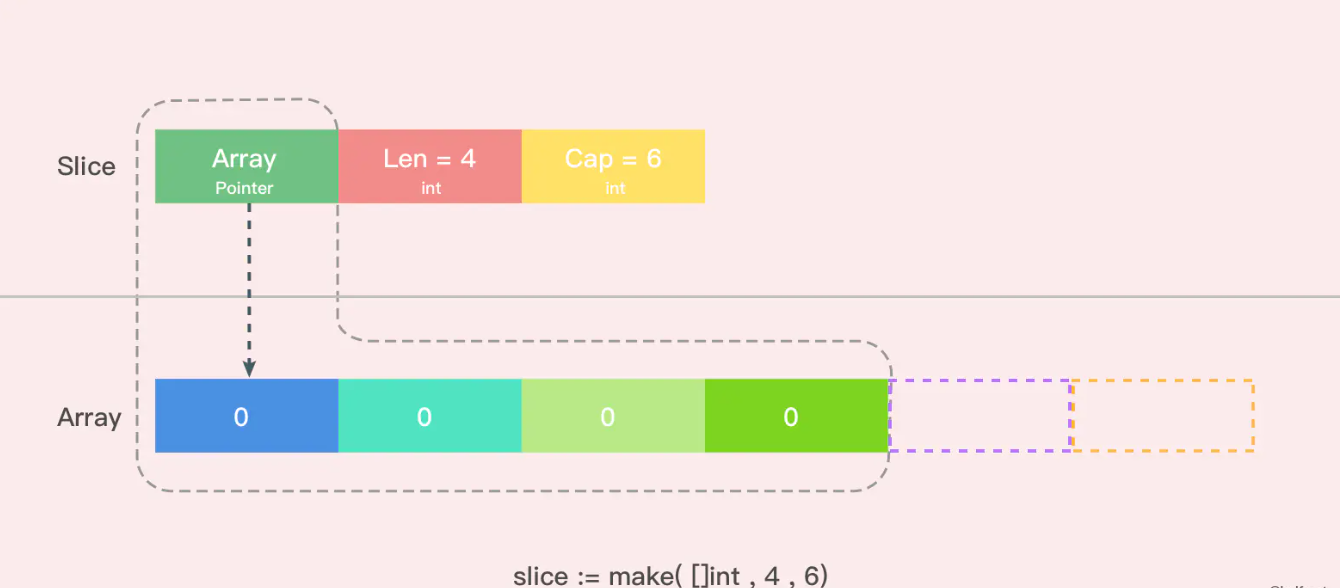
*// 根据切片的容量申请内存*

p := mallocgc(et.size\*uintptr(cap), et, true)

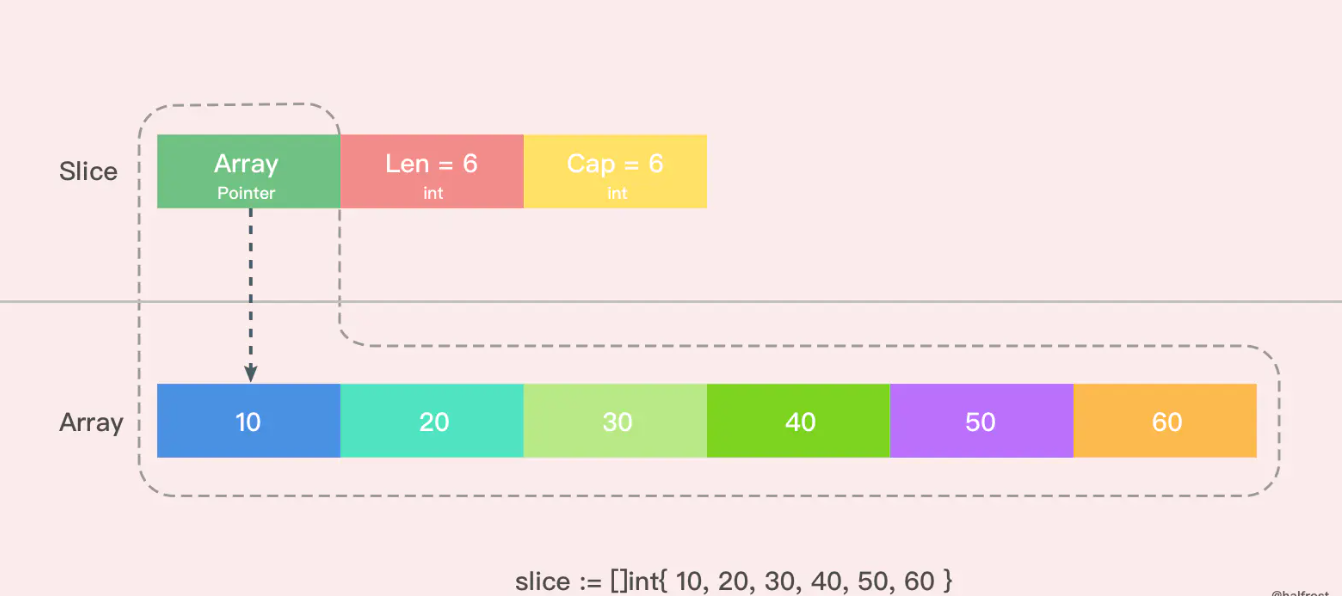
*// 返回申请好内存的切片的首地址*

**return** slice{p, len, cap}

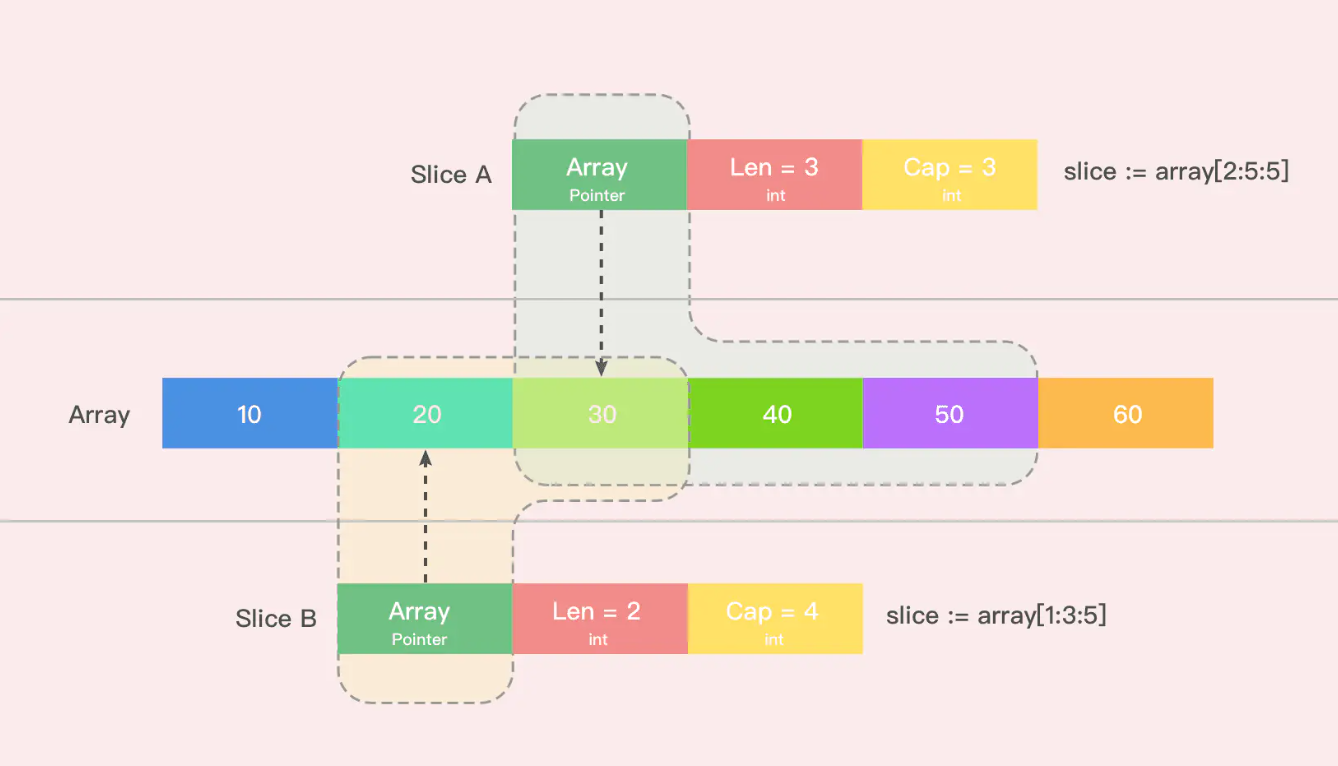
}



上图是用 make 函数创建的一个 len = 4， cap = 6 的切片。内存空间申请了6个 int 类型的内存大小。由于 len = 4，所以后面2个暂时访问不到，但是容量还是在的。这时候数组里面每个变量都是0 。

除了 make 函数可以创建切片以外，字面量也可以创建切片。

这里是用字面量创建的一个 len = 6，cap = 6 的切片，这时候数组里面每个元素的值都初始化完成了。需要注意的是 [ ] 里面不要写数组的容量，因为如果写了个数以后就是数组了，而不是切片了。



还有一种简单的字面量创建切片的方法。如上图。上图就 Slice A 创建出了一个 len = 3，cap = 3 的切片。从原数组的第二位元素(0是第一位)开始切，一直切到第四位为止(不包括第五位)。同理，Slice B 创建出了一个 len = 2，cap = 4 的切片。

#### nil 和空切片

nil 切片和空切片也是常用的。

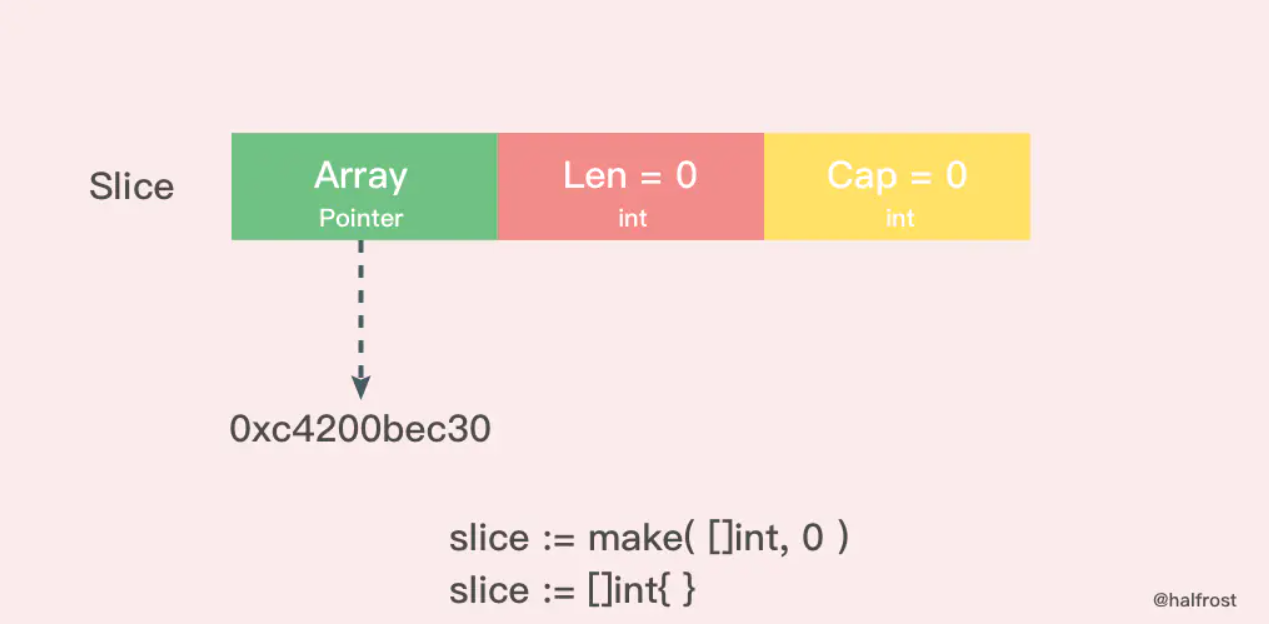
**var** slice []**int**

nil 切片被用在很多标准库和内置函数中，描述一个不存在的切片的时候，就需要用到 nil 切片。比如函数在发生异常的时候，返回的切片就是 nil 切片。nil 切片的指针指向 nil。

空切片一般会用来表示一个空的集合。比如数据库查询，一条结果也没有查到，那么就可以返回一个空切片。

silce := make( []**int** , 0 )

slice := []**int**{ }



空切片和 nil 切片的区别在于，空切片指向的地址不是nil，指向的是一个内存地址，但是它没有分配任何内存空间，即底层元素包含0个元素。

最后需要说明的一点是。不管是使用 nil 切片还是空切片，对其调用内置函数 append，len 和 cap 的效果都是一样的。

### 切片扩容

当一个切片的容量满了，就需要扩容了。怎么扩，策略是什么？

func **growslice**(et \*\_type, old slice, cap **int**) slice {

**if** raceenabled {

callerpc := getcallerpc(**unsafe**.Pointer(&et))

racereadrangepc(old.array, uintptr(old.len\***int**(et.size)), callerpc, funcPC(growslice))

}

**if** msanenabled {

msanread(old.array, uintptr(old.len\***int**(et.size)))

}

**if** et.size == 0 {

*// 如果新要扩容的容量比原来的容量还要小，这代表要缩容了，那么可以直接报panic了。*

**if** cap < old.cap {

panic(errorString("growslice: cap out of range"))

}

*// 如果当前切片的大小为0，还调用了扩容方法，那么就新生成一个新的容量的切片返回。*

**return** slice{**unsafe**.Pointer(&zerobase), old.len, cap}

}

*// 这里就是扩容的策略*

newcap := old.cap

doublecap := newcap + newcap

**if** cap > doublecap {

newcap = cap

} **else** {

**if** old.len < 1024 {

newcap = doublecap

} **else** {

**for** newcap < cap {

newcap += newcap / 4

}

}

}

*// 计算新的切片的容量，长度。*

**var** lenmem, newlenmem, capmem uintptr

**const** ptrSize = **unsafe**.Sizeof((\***byte**)(nil))

**switch** et.size {

**case** 1:

lenmem = uintptr(old.len)

newlenmem = uintptr(cap)

capmem = roundupsize(uintptr(newcap))

newcap = **int**(capmem)

**case** ptrSize:

lenmem = uintptr(old.len) \* ptrSize

newlenmem = uintptr(cap) \* ptrSize

capmem = roundupsize(uintptr(newcap) \* ptrSize)

newcap = **int**(capmem / ptrSize)

**default**:

lenmem = uintptr(old.len) \* et.size

newlenmem = uintptr(cap) \* et.size

capmem = roundupsize(uintptr(newcap) \* et.size)

newcap = **int**(capmem / et.size)

}

*// 判断非法的值，保证容量是在增加，并且容量不超过最大容量*

**if** cap < old.cap || uintptr(newcap) > maxSliceCap(et.size) {

panic(errorString("growslice: cap out of range"))

}

**var** p **unsafe**.Pointer

**if** et.kind&kindNoPointers != 0 {

*// 在老的切片后面继续扩充容量*

p = mallocgc(capmem, nil, false)

*// 将 lenmem 这个多个 bytes 从 old.array地址 拷贝到 p 的地址处*

memmove(p, old.array, lenmem)

*// 先将 P 地址加上新的容量得到新切片容量的地址，然后将新切片容量地址后面的 capmem-newlenmem 个 bytes 这块内存初始化。为之后继续 append() 操作腾出空间。*

memclrNoHeapPointers(**add**(p, newlenmem), capmem-newlenmem)

} **else** {

*// 重新申请新的数组给新切片*

*// 重新申请 capmen 这个大的内存地址，并且初始化为0值*

p = mallocgc(capmem, et, true)

**if** !writeBarrier.enabled {

*// 如果还不能打开写锁，那么只能把 lenmem 大小的 bytes 字节从 old.array 拷贝到 p 的地址处*

memmove(p, old.array, lenmem)

} **else** {

*// 循环拷贝老的切片的值*

**for** i := uintptr(0); i < lenmem; i += et.size {

typedmemmove(et, **add**(p, i), **add**(old.array, i))

}

}

}

*// 返回最终新切片，容量更新为最新扩容之后的容量*

**return** slice{p, old.len, newcap}

}

上述就是扩容的实现。主要需要关注的有两点，一个是扩容时候的策略，还有一个就是扩容是生成全新的内存地址还是在原来的地址后追加。

#### 扩容策略

先看看扩容策略。

func main() {

slice := []**int**{10, 20, 30, 40}

newSlice := append(slice, 50)

fmt.Printf("Before slice = %v, Pointer = %p, len = %d, cap = %d\n", slice, &slice, len(slice), cap(slice))

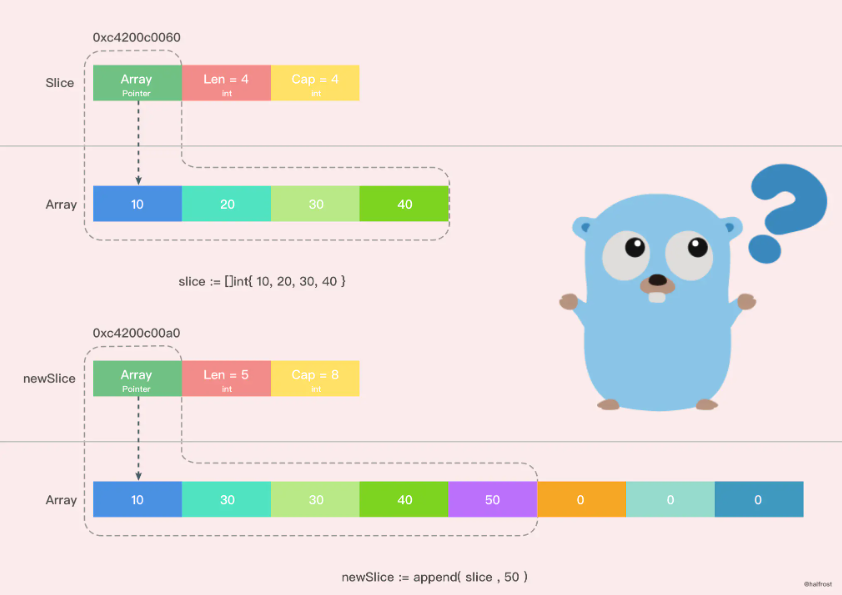
fmt.Printf("Before newSlice = %v, Pointer = %p, len = %d, cap = %d\n", newSlice, &newSlice, len(newSlice), cap(newSlice))

newSlice[1] += 10

fmt.Printf("After slice = %v, Pointer = %p, len = %d, cap = %d\n", slice, &slice, len(slice), cap(slice))

fmt.Printf("After newSlice = %v, Pointer = %p, len = %d, cap = %d\n", newSlice, &newSlice, len(newSlice), cap(newSlice))

}

从图上我们可以很容易的看出，新的切片和之前的切片已经不同了，因为新的切片更改了一个值，并没有影响到原来的数组，新切片指向的数组是一个全新的数组。并且 cap 容量也发生了变化。这之间究竟发生了什么呢？

**Go 中切片扩容的策略是这样的：**

1、如果新扩容的容量是原来的容量的两倍，就直接是新的容量

2、如果切片的容量小于 1024 个元素，于是扩容的时候就翻倍增加容量（上面那个例子也验证了这一情况，总容量从原来的4个翻倍到现在的8个）。

3、一旦元素个数超过 1024 个元素，那么增长因子就变成 1.25 ，即每次增加原来容量的四分之一。

**注意：扩容扩大的容量都是针对原来的容量而言的，而不是针对原来数组的长度而言的。**

#### 新数组 or 老数组 ？

func main() {

array := [4]**int**{10, 20, 30, 40}

slice := array[0:2]

newSlice := append(slice, 50)

fmt.Printf("Before slice = %v, Pointer = %p, len = %d, cap = %d\n", slice, &slice, len(slice), cap(slice))

fmt.Printf("Before newSlice = %v, Pointer = %p, len = %d, cap = %d\n", newSlice, &newSlice, len(newSlice), cap(newSlice))

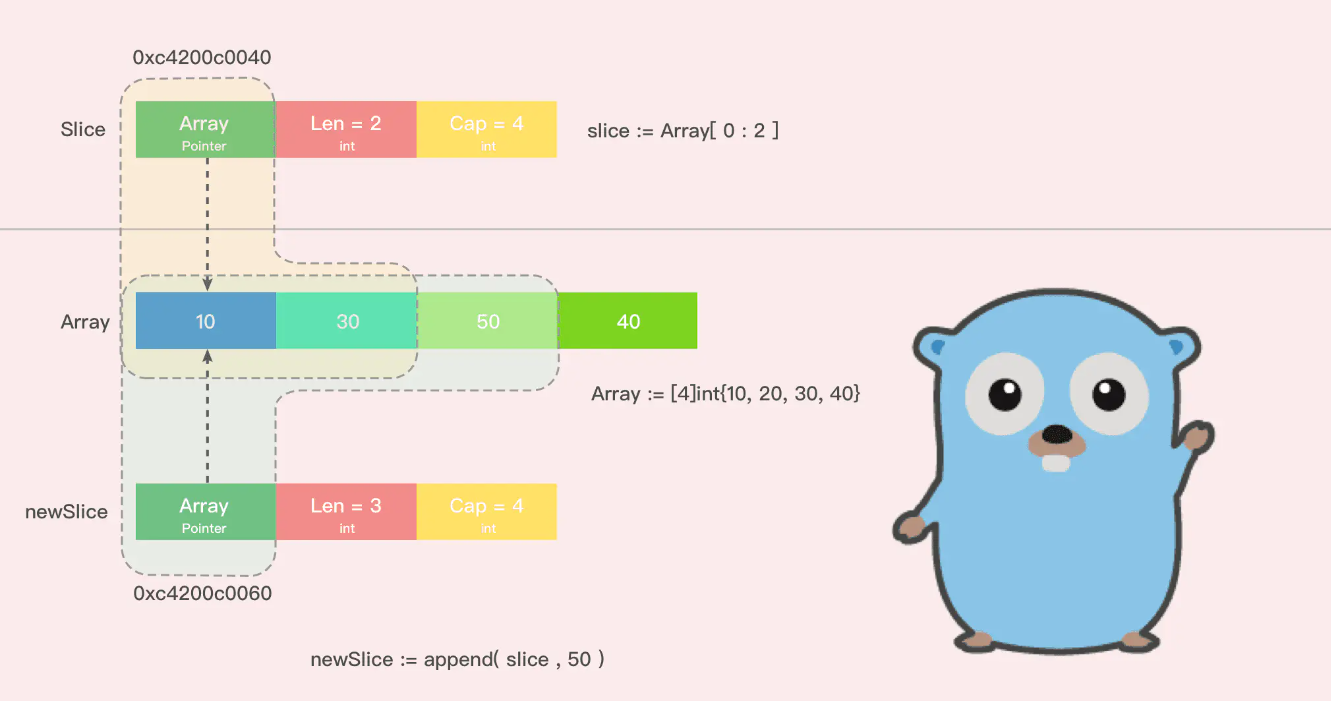
newSlice[1] += 10

fmt.Printf("After slice = %v, Pointer = %p, len = %d, cap = %d\n", slice, &slice, len(slice), cap(slice))

fmt.Printf("After newSlice = %v, Pointer = %p, len = %d, cap = %d\n", newSlice, &newSlice, len(newSlice), cap(newSlice))

fmt.Printf("After array = %v\n", array)

}

通过打印的结果，我们可以看到，在这种情况下，扩容以后并没有新建一个新的数组，扩容前后的数组都是同一个，这也就导致了新的切片修改了一个值，也影响到了老的切片了。并且 append() 操作也改变了原来数组里面的值。一个 append() 操作影响了这么多地方，如果原数组上有多个切片，那么这些切片都会被影响！无意间就产生了莫名的 bug！

这种情况，由于原数组还有容量可以扩容，所以执行 append() 操作以后，会在原数组上直接操作，所以这种情况下，扩容以后的数组还是指向原来的数组。

这种情况也极容易出现在字面量创建切片时候，第三个参数 cap 传值的时候，如果用字面量创建切片，cap 并不等于指向数组的总容量，那么这种情况就会发生。

slice := array[1:2:3]

上面这种情况非常危险，极度容易产生 bug 。

建议用字面量创建切片的时候，cap 的值一定要保持清醒，避免共享原数组导致的 bug。

**情况二：**

情况二其实就是在扩容策略里面举的例子，在那个例子中之所以生成了新的切片，是因为原来数组的容量已经达到了最大值，再想扩容， Go 默认会先开一片内存区域，把原来的值拷贝过来，然后再执行 append() 操作。这种情况丝毫不影响原数组。

所以建议尽量避免情况一，尽量使用情况二，避免 bug 产生。

### 切片拷贝

Slice 中拷贝方法有2个。

func **slicecopy**(to, fm slice, width uintptr) **int** {

*// 如果源切片或者目标切片有一个长度为0，那么就不需要拷贝，直接 return*

**if** fm.len == 0 || to.len == 0 {

**return** 0

}

*// n 记录下源切片或者目标切片较短的那一个的长度*

n := fm.len

**if** to.len < n {

n = to.len

}

*// 如果入参 width = 0，也不需要拷贝了，返回较短的切片的长度*

**if** width == 0 {

**return** n

}

*// 如果开启了竞争检测*

**if** raceenabled {

callerpc := getcallerpc(unsafe.Pointer(&to))

pc := funcPC(slicecopy)

racewriterangepc(to.array, uintptr(n\***int**(width)), callerpc, pc)

racereadrangepc(fm.array, uintptr(n\***int**(width)), callerpc, pc)

}

*// 如果开启了 The memory sanitizer (msan)*

**if** msanenabled {

msanwrite(to.array, uintptr(n\***int**(width)))

msanread(fm.array, uintptr(n\***int**(width)))

}

size := uintptr(n) \* width

**if** size == 1 {

*// TODO: is this still worth it with new memmove impl?*

*// 如果只有一个元素，那么指针直接转换即可*

\*(\*byte)(to.array) = \*(\*byte)(fm.array) *// known to be a byte pointer*

} **else** {

*// 如果不止一个元素，那么就把 size 个 bytes 从 fm.array 地址开始，拷贝到 to.array 地址之后*

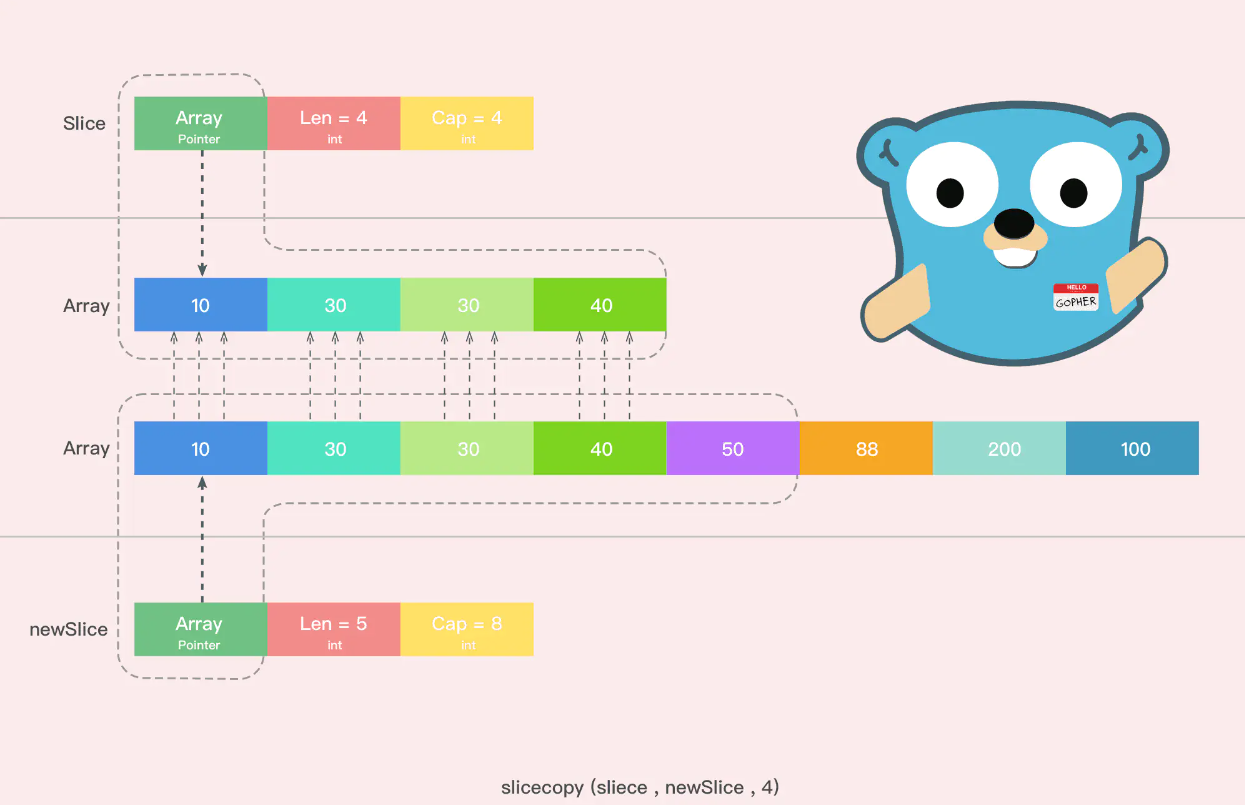
memmove(to.array, fm.array, size)

}

**return** n

}

在这个方法中，slicecopy 方法会把源切片值(即 fm Slice )中的元素复制到目标切片(即 to Slice )中，并返回被复制的元素个数，copy 的两个类型必须一致。slicecopy 方法最终的复制结果取决于较短的那个切片，当较短的切片复制完成，整个复制过程就全部完成了。



还有一个拷贝的方法，这个方法原理和 slicecopy 方法类似，不在赘述了，注释写在代码里面了。

func **slicestringcopy**(to []**byte**, fm **string**) **int** {

*// 如果源切片或者目标切片有一个长度为0，那么就不需要拷贝，直接 return*

**if** **len**(fm) == 0 || len(to) == 0 {

**return** 0

}

*// n 记录下源切片或者目标切片较短的那一个的长度*

n := len(fm)

**if** **len**(to) < n {

n = len(to)

}

*// 如果开启了竞争检测*

**if** raceenabled {

callerpc := getcallerpc(**unsafe**.Pointer(&to))

pc := funcPC(slicestringcopy)

racewriterangepc(**unsafe**.Pointer(&to[0]), uintptr(n), callerpc, pc)

}

*// 如果开启了 The memory sanitizer (msan)*

**if** msanenabled {

msanwrite(**unsafe**.Pointer(&to[0]), uintptr(n))

}

*// 拷贝字符串至字节数组*

memmove(**unsafe**.Pointer(&to[0]), stringStructOf(&fm).str, uintptr(n))

**return** n

}

说到拷贝，切片中有一个需要注意的问题。

func main() {

slice := []**int**{10, 20, 30, 40}

**for** **index**, value := range slice {

fmt.Printf("value = %d , value-addr = %x , slice-addr = %x\n", value, &value, &slice[**index**])

}

}

从上面结果我们可以看到，如果用 range 的方式去遍历一个切片，拿到的 Value 其实是切片里面的值拷贝。所以每次打印 Value 的地址都不变。

