Slices

Uso y funcionamiento

¿Qué son?

El *slice* es una abstracción construida sobre el tipo de datos *array* de Go, por lo que para comprender su funcionamiento primero debemos entender el array.

Array

Una definición de tipo de *array* especifica una **longitud y un tipo de elemento**. Por ejemplo, el tipo `[4]int` representa una array de cuatro enteros.

- El tamaño del array es fijo; su longitud es parte de su tipo (`[4]int` y `[5]int` son tipos distintos e incompatibles).
- La expresión `s[n]`accede al elemento n, comenzando desde cero.
- Los arrays no necesitan inicializarse explícitamente; se les pone automáticamente el valor cero a cada elemento.

Ejemplo de definición

```
var a [4]int

a[0] = 1

i := a[0]

// i == 1
```

Propiedades

Un array de Go no es un puntero al primer elemento (como sería el caso en C).

- Esto significa que cuando asigna o pasa un array como valor hará una copia de su contenido.
    - Para evitar la copia, podría pasar un *puntero* al array, pero ese es un puntero a un array y no a un array.

Ejemplos,

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

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

En ambos casos, el tipo de `b` es `[2]string`.

Arrays vs. Slices

- **Arrays:**
  Son un poco inflexibles dado su tamaño fijo, por lo que no se ven con mucha frecuencia en el código Go.

- **Slices:**
  Están por todas partes dada su potencia y comodidad.

Definición

La especificación de tipo para un slice es `[]T`, donde `T` es el tipo de los elementos del slice. A diferencia de un tipo del array, un tipo de slice no tiene una longitud específica.

Un literal de slice se declara como un literal de array, excepto que omite el recuento de elementos, por ejemplo:

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

Se puede crear un slice con la función integrada `make`:

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

len y cap son opcionales

Ejemplo

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

Cuando se omite el argumento de capacidad, el valor predeterminado es la longitud especificada. Aquí hay una versión más reducida del mismo código:

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

La longitud y la capacidad de un slice se pueden inspeccionar utilizando las funciones integradas `len` y `cap`.

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

Segmentos de un slice

Se puede formar un slice "cortando" un slice o array existente. El corte se realiza especificando un rango semiabierto con dos índices separados por dos puntos. Por ejemplo, la expresión `b[1:4]` crea un slice que incluye los elementos del 1 al 3 de `b` (los índices del slice resultante serán del 0 al 2).

```
b := []byte{'g', 'o', 'l', 'a', 'n', 'g'}
// b[1:4] == []byte{'o', 'l', 'a'}, comparte el mismo almacenamiento que b
```

Los índices inicial y final de una expresión de slice son opcionales; por defecto, son cero y la longitud del segmento respectivamente:

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

Sintaxis alternativa

Se puede formar un slice "cortando" un slice o array existente. El corte se realiza especificando un rango semiabierto con dos índices separados por dos puntos. Por ejemplo, la expresión `b[1:4]` crea un slice que incluye los elementos del 1 al 3 de `b` (los índices del slice resultante serán del 0 al 2).

```
b := []byte{'g', 'o', 'l', 'a', 'n', 'g'}
// b[1:4] == []byte{'o', 'l', 'a'}, comparte el mismo almacenamiento que  b
```

Los índices inicial y final de una expresión de slice son opcionales; por defecto, son cero y la longitud del segmento respectivamente:

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

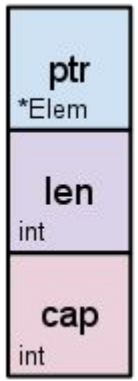
Segmentos de un slice

Esta es también la sintaxis para crear un slice dado un array:

```
x := [3]string{"España", "Francia", "Inglaterra"}
s := x[:] // un slice que referencia lo almacenado en el array
```

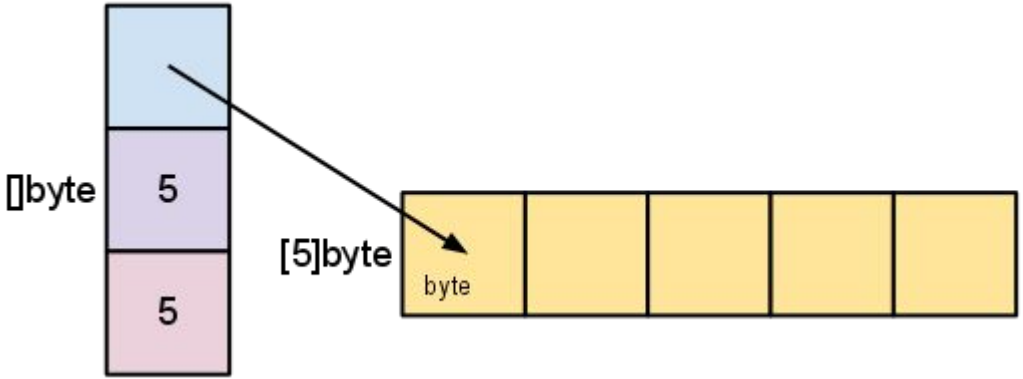
¿Qué es un slice al fin?

Un slice es un descriptor de un segmento de array. Consiste en un **puntero a al array, la longitud del slice y su capacidad** (la longitud máxima del segmento).



En nuestro ejemplo

Nuestra variable `s`, creada anteriormente por `make([]byte, 5)`, está estructurada así:

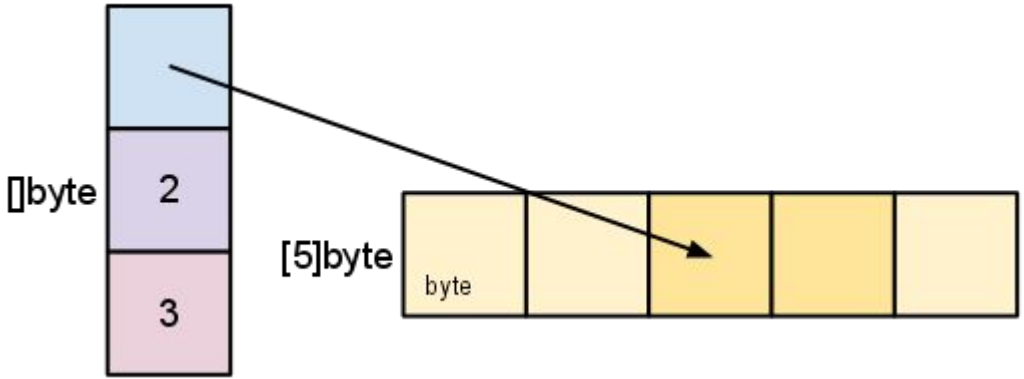


Longitud vs. capacidad

La *longitud* es el número de elementos a los que se refiere el slice. La *capacidad* es el número de elementos en el array subyacente (comenzando en el elemento al que hace referencia el puntero de segmento).

A medida que cortamos s, se ven los cambios en la estructura de datos del slice y su relación con el array subyacente:

s = s[2:4]



Referencia a los mismos datos

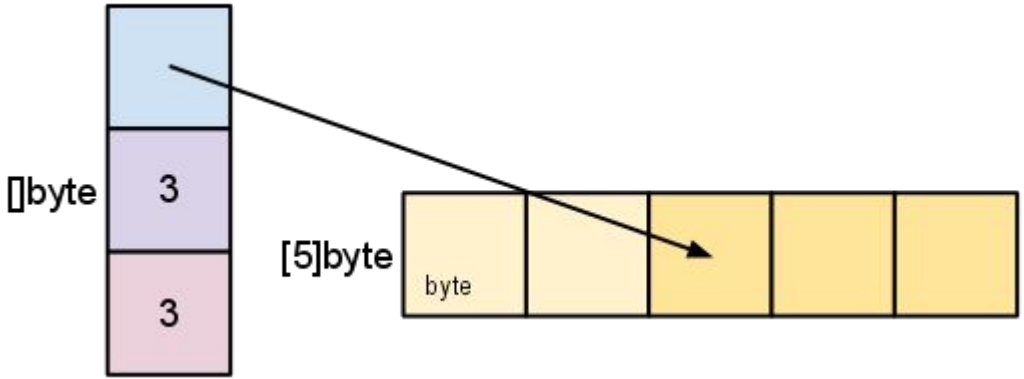
La creación de un sub-slice no copia los datos sino que crea un nuevo valor de segmento que apunta al array original. Esto hace que las operaciones de acceso a los segmentos sean tan eficientes como la manipulación de índices de los arrays. Por lo tanto, modificar los *elementos* de un nuevo sub-slice modifica los elementos del slice original:

```
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'}
```

Referencia a los mismos datos

Anteriormente cortamos s a una longitud más corta que su capacidad. Podemos hacer crecer s a su capacidad cortándolo nuevamente:

s = s[:cap(s)]



Un slice no puede crecer más allá de su capacidad. Intentar hacerlo provocará un error en el tiempo de ejecución, al igual que cuando se indexa fuera de los límites de un slice o un array. Del mismo modo, los slices no se pueden volver a dividir por debajo de cero para acceder a elementos anteriores del array.

Hacer crecer los slices

Para aumentar la capacidad de un slice, se debe crear un slice nuevo más grande y copiar en él el contenido del slice original. Esta técnica es la forma en que las implementaciones de arrays dinámicos de otros lenguajes funcionan detrás de escena. El siguiente ejemplo duplica la capacidad de `s` creando un nuevo slice, `t` copiando el contenido de `s` en `t` y luego asignando el valor del sector `t` a `s`:

```
t := make([]byte, len(s), (cap(s)+1)*2) // +1 en caso de que cap(s) == 0

for i := range s {

	t[i] = s[i]

}

s = t
```

Hacer crecer los slices

Esta misma operación común se facilita con la función ***copy*** built-in (nativa del lenguaje). Como sugiere el nombre, ***copy*** copia datos de un slice de origen a un slice de destino. Devuelve el número de elementos copiados.

func copy(dst, src []T) int

La función `copy` admite la copia entre segmentos de diferentes longitudes (solo copiará hasta el menor número de elementos). Además, `copy` puede manejar segmentos de origen y destino que comparten el mismo array subyacente, manejando correctamente los segmentos superpuestos.

Usando `copy`, podemos simplificar el fragmento de código anterior:

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

copy(t, s)

s = t
```

Agregar datos al final del slice

Una operación común es agregar datos al final de un slice. Esta función agrega elementos “bytes” a un “slice de bytes”, haciendo crecer el slice si es necesario, y devuelve el slice actualizado:

```
func AppendByte(slice []byte, data ...byte) []byte {
    m := len(slice)
    n := m + len(data)
    if n > cap(slice) { // si es necesario realocar
        // aloca el doble de lo que se necesita para futuros crecimientos
        newSlice := make([]byte, (n+1)*2)
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:n]
    copy(slice[m:n], data)
    return slice
}
```

Uno podría usar `AppendByte` así:

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

Agregar datos al final del slice

Funciones como `AppendByte` son útiles porque ofrecen un control completo sobre la forma en que crece el slice. Dependiendo de las características del programa, puede ser conveniente asignar en partes más pequeñas o más grandes, o poner un tope al tamaño de una reasignación.

Pero la mayoría de los programas no necesitan un control completo, por lo que Go proporciona una función built-in `append` que es buena para la mayoría de los propósitos; tiene el siguiente encabezado:

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

La función `append` añade los elementos `x` al final del slice `s` y aumenta el slice si se necesita una mayor capacidad.

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

Agregar un slice al final

Para agregar un slice a otro, se puede usar `...` para expandir el segundo argumento a una lista de argumentos.

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

La capacidad con append

```
func main() {
    slice := make([]int, 0, 5)
    var ptr *int
    for i := 0; i < 21; i++ {
        slice = append(slice, i)
        fmt.Printf("len=%d cap=%d\n", len(slice), cap(slice))
        if ptr != &slice[0] {
            fmt.Println("dirección del elemento 0 del slice:", &slice[0])
            ptr = &slice[0]
        }
    }
}
```

Ejemplo de ejecución del programa

len=1 cap=5

dirección del elemento 0 del slice: 0xc000126000

len=2 cap=5

len=3 cap=5

len=4 cap=5

len=5 cap=5

len=6 cap=10

dirección del elemento 0 del slice: 0xc00012c000

len=7 cap=10

len=8 cap=10

len=9 cap=10

len=10 cap=10

len=11 cap=20

dirección del elemento 0 del slice: 0xc00012e000

len=12 cap=20

len=13 cap=20

len=14 cap=20

len=15 cap=20

len=16 cap=20

len=17 cap=20

len=18 cap=20

len=19 cap=20

len=20 cap=20

len=21 cap=40

dirección del elemento 0 del slice: 0xc000130000