

Capítulo 4

Tipos compuestos

En el **Capítulo 3** vimos los tipos básicos que sirven como bloques de construcción para estructuras de datos en un programa **Go**; son los átomos de nuestro universo. En este capítulo, veremos los tipos compuestos, las moléculas creadas mediante la combinación de los tipos básicos de diversas maneras. Hablaremos de cuatro de estos tipos, arreglos, **slices**, mapas y estructuras, y al final del capítulo, mostraremos cómo pueden ser codificados los datos estructurados que utilizan estos tipos y como analizarlos a partir de datos **JSON** y utilizarlos para generar **HTML** a partir de plantillas.

Los arreglos y las estructuras son tipos agregados; sus valores son concatenaciones de otros valores en la memoria. Los arreglos son homogéneos, todos sus elementos tienen el mismo tipo, mientras que las estructuras son heterogéneas. Arreglos y estructuras son de tamaño fijo. Por el contrario, los **slices** y los mapas son estructuras de datos dinámicas que crecen a medida que se añaden valores.

4.1. Arreglos

Un arreglo es una secuencia de longitud fija de cero o más elementos de un tipo particular. Debido a su longitud fija, los arreglos son raramente utilizados en **Go** directamente. Los **slices**, que pueden crecer o reducirse, son mucho más versátiles, pero para entender los **slices** debemos comprender los arreglos primero.

Se accede a los elementos individuales de un arreglo con la notación de subíndice convencional, donde los subíndices van de cero a uno menos que la longitud de arreglo. La función incorporada **len** devuelve el número de elementos del arreglo.

```
var a [3]int // arreglo de 3 enteros
fmt.Println(a[0]) // imprimir el primer elemento
fmt.Println(a[len(a)-1]) // imprimir el último elemento, a[2]
// Imprimir los índices y elementos.
for i, v := range a {
    fmt.Printf("%d %d\n", i, v)
}
// Imprimir los elementos solamente.
for _, v := range a {
    fmt.Printf("%d\n", v)
}
```

De forma predeterminada, los elementos de una nueva variable de arreglo están configurados inicialmente con el valor cero para el tipo de elemento, que es **0** para los números. Podemos utilizar un literal de arreglo para inicializar un arreglo con una lista de valores:

```
var q [3]int = [3]int{1, 2, 3}
var r [3]int = [3]int{1, 2}
fmt.Println(r[2]) // "0"
```

En un literal de arreglo, si aparecen los puntos suspensivos **"..."** en lugar de su longitud, la longitud del arreglo se determina por el número de valores de inicialización. La definición de **q** se puede simplificar a:

```
q := [...]int{1, 2, 3}
fmt.Printf("%T\n", q) // "[3]int"
```

El tamaño de un arreglo es parte de su tipo, por lo que **[3]int** y **[4]int** son diferentes tipos. El tamaño debe ser una expresión constante, es decir, una expresión cuyo valor puede ser calculada cuando el programa está siendo compilado.

```
q := [3]int{1, 2, 3}
q=[4]int{1, 2, 3, 4} // error de compilación: No se puede asignar [4]int a [3]int
```

Como vemos, la sintaxis literal es similar para los arreglos, **slices**, mapas y estructuras. La forma específica anterior es una lista de valores en orden, pero también es posible especificar una lista de pares de índices y valores, de esta manera:

```
type Currency int

const (
    USD Currency = iota
    EUR
    GBP
    RMB
)
symbol := [...]string{USD: "$", EUR: "€", GBP: "£", RMB: "¥"}
fmt.Println(RMB, symbol[RMB]) // "3 ¥"
```

De esta forma, los índices pueden aparecer en cualquier orden y algunos pueden ser omitidos; como antes, los valores no especificados toman el valor cero para el tipo de elemento. Por ejemplo,

```
r := [...]int{99: -1}
```

define un arreglo **r** con **100** elementos, todos cero, excepto el último, que tiene un valor **-1**.

Si un tipo elemento del arreglo es comparable, el tipo de arreglo es comparable también, así que se pueden comparar directamente dos arreglos de ese tipo con el operador **==**, que informa de si todos los elementos correspondientes son iguales. El operador **!=** es su negación.

```
a := [2]int{1, 2}
b := [...]int{1, 2}
c := [2]int{1, 3}
fmt.Println(a == b, a == c, b == c) // "true false false"
d := [3]int{1, 2}
fmt.Println(a == d) // error de compilación: no se puede comparar [2]int == [3]int
```

Como un ejemplo más plausible, la función **Sum256** en el paquete **crypto/sha256** produce el hash criptográfico **SHA256** o **digest** de un mensaje almacenado en un **slice** de bytes arbitrario. El **digest** tiene **256 bits**, por lo que su tipo es **[32]byte**. Si dos **digests** son los mismos, es extremadamente probable que los dos mensajes sean los mismos; si los **digests** son diferentes, los dos mensajes son diferentes. Este programa imprime y compara los **digests SHA256** de "x" y "X":

```
gopl.io/ch4/sha256
import "crypto/sha256"

func main() {
    c1 := sha256.Sum256([]byte("x"))
    c2 := sha256.Sum256([]byte("X"))
    fmt.Printf("%x\n%x\n%t\n%T\n", c1, c2, c1 == c2, c1)
    // Output:
    // 2d711642b726b04401627ca9fbac32f5c8530fb1903cc4db02258717921a4881
    // 4b68ab3847feda7d6c62c1fbcbeebfa35eab7351ed5e78f4ddadea5df64b8015
    // false
    // [32]uint8
}
```

Las dos entradas difieren sólo en un único bit, pero aproximadamente la mitad de los bits son diferentes en los **digests**. Observar los verbos **printf**: **%x** para imprimir todos los elementos de un arreglo o un **slice** de bytes en hexadecimal, **%t** para mostrar un valor booleano, y **%T** para mostrar el tipo de un valor.

Cuando se llama una función, una copia de cada valor del argumento se asigna a la variable del parámetro correspondiente, por lo que la función recibe una copia, no el original. Pasar grandes arreglos de esta manera puede ser ineficiente, y cualquier cambio que haga la función a los elementos del arreglo afectará sólo a la copia, no al original. En este sentido, **Go** trata los arreglos como cualquier otro tipo, pero este comportamiento es diferente de otros lenguajes que implícitamente pasan arreglos por referencia.

Por supuesto, podemos pasar explícitamente un puntero a un arreglo de modo que cualquier modificación que la función haga a elementos del arreglo serán visibles para el que llama. Esta función pone a cero los contenidos de un arreglo **[32]byte**:

```
func zero(ptr *[32]byte) {
    for i := range ptr {
        ptr[i] = 0
    }
}
```

El literal de arreglo **[32]byte{}** produce un arreglo de **32 bytes**. Cada elemento del arreglo tiene el valor cero para byte, que es cero. Podemos utilizar este hecho para escribir una versión diferente de **zero**:

```
func zero(ptr *[32]byte) {
    *ptr = [32]byte{}
}
```

El uso de un puntero a un arreglo es eficiente y permite que la función llamada pueda mutar la variable que llama, pero los arreglos siguen siendo inherentemente inflexibles debido a su tamaño fijo. La función **zero** no acepta un puntero a una variable **[16]byte**, por ejemplo, ni hay ninguna manera de añadir o eliminar elementos del arreglo. Por estas razones, aparte de casos especiales como **SHA256** de hash de tamaño fijo, rara vez se utilizan los arreglos como parámetros de la función; En cambio, usamos **slices**.

4.2. Slices

Los **Slices** representan secuencias de longitud variable con todos los elementos del mismo tipo. Un tipo **slice** se escribe **[]T**, donde los elementos tienen el tipo **T**; se ve como un tipo de arreglo sin tamaño.

Los arreglos y los **slices** están íntimamente conectados. Un **slice** es una estructura de datos de peso ligero que da acceso a una subsecuencia de los elementos de un arreglo (o tal vez todos), que se conoce como arreglo subyacente

del **slice**. Un **slice** tiene tres componentes: un puntero, una longitud y una capacidad. El puntero apunta al primer elemento del arreglo que es accesible a través del **slice**, no es necesariamente el primer elemento del arreglo. La longitud es el número de elementos del **slice**; no puede exceder la capacidad, que es por lo general el número de elementos entre el inicio del **slice** y el final del arreglo subyacente. Las funciones incorporadas **len** y **cap** devuelven esos valores.

El mismo arreglo subyacente lo pueden compartir múltiples **slices** y pueden referirse a partes superpuestas de ese arreglo. La **Figura 4.1** muestra un arreglo de **strings** para los meses del año, y dos **slices** superpuestos del mismo. El arreglo se declara como

```
months := [...]string{1: "January", /* ... */ , 12: "December"}
```

así que **January** es **months[1]** y **December** es **months[12]**. Por lo general, el elemento del arreglo en el índice **0** contendría el primer valor, pero como los meses siempre están numerados desde **1**, podemos dejarlo fuera de la declaración e inicializarlo con una cadena vacía.

El operador de **slice** **s[i: j]**, donde $0 \leq i \leq j \leq \text{cap}(s)$, crea un nuevo **slice** que se refiere a los elementos **i** hasta **j-1** de la secuencia **s**, que puede ser una variable de arreglo, un puntero a un arreglo, u otro **slice**. El **slice** resultante tiene **j-i** elementos. Si **i** se omite, es **0**, y si **j** se omite, es **len(s)**. Así, el **slice** **months[1:13]** se refiere a toda la gama de meses válidos, al igual que el **slice** **months[1:]**; El **slice** **months[:]** se refiere a todo el arreglo. Vamos a definir **slices** superpuestos para el segundo trimestre y el verano del norte (**summer**):

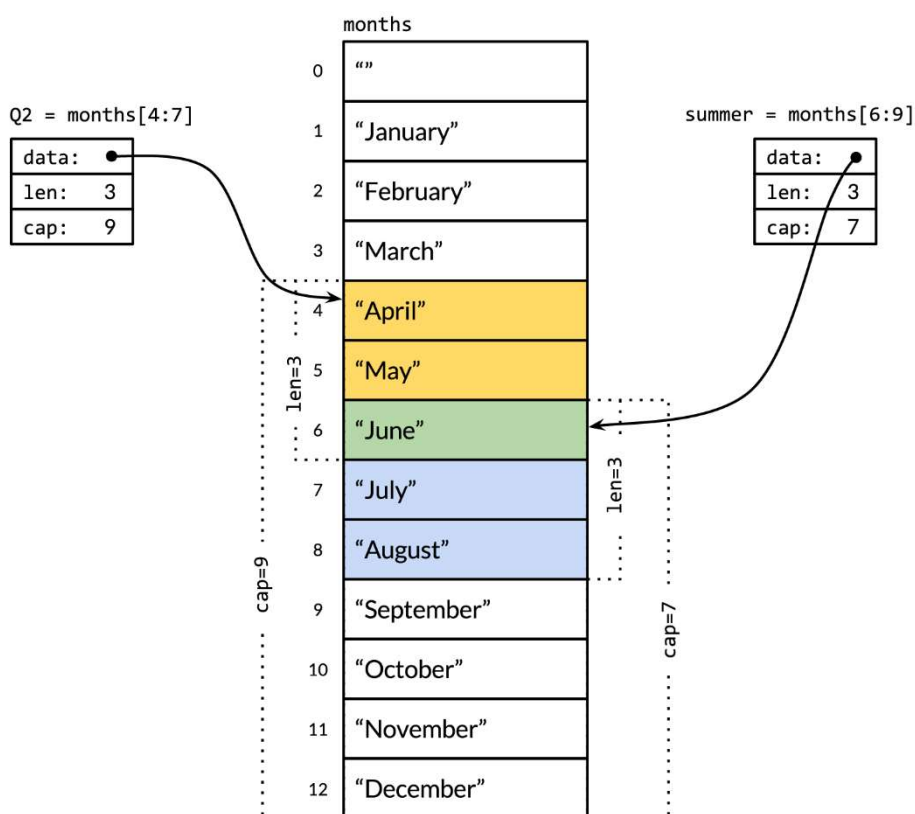


Figura 4.1. Dos **slices** superpuestos de una serie de meses.

```
Q2 := months[4:7]
summer := months[6:9]
fmt.Println(Q2) // ["April" "May" "June"]
fmt.Println(summer) // ["June" "July" "August"]
```

June se incluye en ambos y es la única salida de esta prueba (ineficaz) para los elementos comunes:

```
for _, s := range summer {
    for _, q := range Q2 {
        if s == q {
            fmt.Printf("%s appears in both\n", s)
        }
    }
}
```

Hacer **slicing** más allá de la **cap(s)** causa un pánico, pero hacer **slicing** más allá de **len(s)** extiende el **slice**, por lo que el resultado puede ser más largo que el original:

```
fmt.Println(summer[:20]) // panico: fuera de rango
endlessSummer := summer[:5] // extiende un slice (dentro de la capacidad)
fmt.Println(endlessSummer) // "[June July August September October]"
```

Como acotación al margen, observar la similitud de la operación de **substrings** en **strings** del operador de **slices** en **[]bytes**. Ambos están escritos **x[m:n]**, y ambos retornan una subsecuencia de los bytes originales, compartiendo la representación subyacente de modo que ambas operaciones toman tiempo constante. Con la expresión **x[m:n]** se obtiene un **string** si **x** es un **string**, o un **byte[]** si **x** es un **byte[]**.

Dado que una **slice** contiene un puntero a un elemento de un arreglo, al pasar un **slice** a una función permite que la función pueda modificar los elementos del arreglo subyacente. En otras palabras, la copia de una **slice** crea un alias (§2.3.2) para el arreglo subyacente. La función **reverse** invierte los elementos de un **slice []int** en su lugar, y se puede aplicar a los **slices** de cualquier longitud.

```
gopl.io/ch4/rev
// reverse invierte un slice de ints en el lugar.
func reverse(s []int) {
    for i, j := 0, len(s)-1; i < j; i, j = i+1, j-1 {
        s[i], s[j] = s[j], s[i]
    }
}
```

Aquí invertimos todo el arreglo **a**:

```
a := [...]int{0, 1, 2, 3, 4, 5}
reverse(a[:])
fmt.Println(a) // "[5 4 3 2 1 0]"
```

Una forma sencilla de rotar un **slice** dado por **n** elementos es aplicar la función **reverse** tres veces, primero a los principales **n** elementos, a continuación, a los elementos restantes, y finalmente al conjunto del **slice**. (Para girar a la derecha, tomar la tercera llamada en primer lugar.)

```
s := []int{0, 1, 2, 3, 4, 5}
// Rotar s a la izquierda dos posiciones.
reverse(s[:2])
reverse(s[2:])
reverse(s)
fmt.Println(s) // "[2 3 4 5 0 1]"
```

Observar cómo la expresión que inicializa el **slice s** difiere de la del arreglo **a**. Un literal de **slice** se parece a un literal de arreglo, una secuencia de valores separados por comas y entre llaves, pero no se da el tamaño. Esto crea implícitamente una variable de arreglo del tamaño correcto y produce un **slice** que apunta al mismo. Al igual que con los literales de arreglo, los literales de **slice** pueden especificar los valores en orden, o dar sus índices de forma explícita, o utilizar una mezcla de los dos estilos.

A diferencia de los arreglos, los **slices** no son comparables, por lo que no podemos usar **==** para probar si dos **slices** contienen los mismos elementos. La librería estándar proporciona la altamente optimizada función **bytes.Equal** para comparar dos **slices** de bytes (**[] byte**), pero para otros tipos de **slice**, tenemos que hacer la comparación nosotros mismos:

```
func equal(x, y []string) bool {
    if len(x) != len(y) {
        return false
    }
    for i := range x {
        if x[i] != y[i] {
            return false
        }
    }
    return true
}
```

En vista de lo natural, que es esta "profunda" prueba de igualdad, y que no es más costosa en tiempo de ejecución que el operador **==** de arreglos de **strings**, puede ser desconcertante que las comparaciones **slice** no funcionen también de esta manera. Hay dos razones por las cuales la equivalencia profunda es problemática. En primer lugar, a diferencia de elementos del arreglo, los elementos de un **slice** son indirectos, por lo que es posible que un **slice** se contenga a sí mismo. Aunque hay maneras de hacer frente a estos casos, ninguno es simple ni eficiente, y lo más importante, obvio.

En segundo lugar, como los elementos de un **slice** son indirectos, un valor fijo de **slice** puede contener diferentes elementos en diferentes momentos cuando el contenido del arreglo subyacente se modifica. Debido a una tabla hash como un tipo **map** de **Go** hace sólo las copias de poca profundidad de sus claves, se requiere que la igualdad para cada clave siga siendo la misma a lo largo de la vida útil de la tabla **hash**. La equivalencia profunda sería pues hacer **slices** inadecuados para su uso como claves de **maps**. Para los tipos de referencia como punteros y canales, el operador **==** prueba identidad de referencia, es decir, si las dos entidades se refieren a la misma cosa. Una análoga "superficial" prueba de igualdad de los **slices** podría ser útil, y resolvería el problema con los **maps**, pero el tratamiento inconsistente de los **slices** y los arreglos por el operador **==** sería confuso. La opción más segura es no permitir comparaciones **slice** por completo.

La única comparación **slice** legal es **nil**, como

```
if summer == nil { /* ... */ }
```

El valor cero de un tipo de **slice** es **nil**. Un **slice nil** no tiene ningún arreglo subyacente. El **slice nil** tiene una longitud y una capacidad cero, pero también hay **slices** no **nil** de longitud y capacidad cero, como `[]int{}` o `make([]int,3)[3:]`. Al igual que con cualquier tipo que pueden tener valores **nil**, el valor **nil** de un tipo de **slice** en particular puede ser escrito usando una expresión de conversión tal como `int[](nil)`.

```
var s []int // len(s) == 0, s == nil
s=nil // len(s) == 0, s == nil
s=[]int(nil) // len(s) == 0, s == nil
s=[]int{} // len(s) == 0, s != nil
```

Por lo tanto, si se necesita comprobar si un **slice** está vacío, utilizar `len(s) == 0`, no `s==nil`. Aparte de la comparación de igual a cero, un **slice nil** se comporta como cualquier otro **slice** de longitud cero; por ejemplo, `reverse(nil)` es perfectamente seguro. A no ser que claramente se documente lo contrario, las funciones **Go** deben tratar a todos los **slices** de longitud cero de la misma manera, ya sea **nil** o no **nil**.

La función incorporada **make** crea un **slice** de un tipo de elemento específico, y de longitud y capacidad específica. El argumento de la capacidad se puede omitir, en cuyo caso la capacidad es igual a la longitud.

```
make([]T, len)
make([]T, len, cap) // lo mismo que make([]T, cap)[:len]
```

Bajo el capó, **make** crea una variable de arreglo sin nombre y devuelve un **slice** en ella; el arreglo es accesible sólo a través del **slice** retornado. En la primera forma, la **slice** es una vista de todo el arreglo. En la segunda, el **slice** es una vista de sólo los primeros **len** elementos del arreglo, pero su capacidad incluye todo el arreglo. Los elementos adicionales se reservan para el crecimiento futuro.

4.2.1. La función append

La función incorporada **append** agrega elementos a **slices**:

```
var runes []rune
for _, r := range "Hello, 世界" {
    runes = append(runes, r)
}
fmt.Printf("%q\n", runes) // "['H' 'e' 'l' 'l' 'o' ', ' ' ' ' '世' '界']"
```

El bucle utiliza **append** para construir el **slice** de nueve runas codificadas por el **string** literal, aunque este problema específico se resuelve más convenientemente mediante la conversión incorporada `[]rune("Hello, 世界")`.

La función **append** es crucial para entender cómo funcionan los **slices**, así que vamos a echar un vistazo a lo que está pasando. Aquí hay una versión llamada **appendInt** que está especializada en **slices []int**:

```
gopl.io/ch4/append
func appendInt(x []int, y int) []int {
    var z []int
    zlen := len(x) + 1
    if zlen <= cap(x) {
        // No hay espacio para crecer. Extender el slice.
        z=x[:zlen]
    } else {
        // No hay espacio suficiente. Asignar un nuevo arreglo.
        // Crecer con duplicación, para amortizar la complejidad lineal.
        zcap := zlen
        if zcap < 2*len(x) {
            zcap = 2 * len(x)
        }
        z=make([]int, zlen, zcap)
        copy(z, x) // una función incorporada; véase el texto
    }
    z[len(x)] = y
    return z
}
```

Cada llamada a **appendInt** debe comprobar si el **slice** tiene capacidad suficiente para contener los nuevos elementos del arreglo existente. Si es así, se extiende el **slice** mediante la definición de un **slice** más grande (todavía dentro del arreglo original), copia el elemento **y** en el nuevo espacio, y retorna el **slice**. La entrada **x** y el resultado **z** comparten el mismo arreglo subyacente.

Si no hay suficiente espacio para el crecimiento, **appendInt** debe asignar un nuevo arreglo suficientemente grande como para contener el resultado, copiar los valores de **x** en él, y poner el nuevo elemento **y**. El resultado **z** ahora se refiere a un arreglo subyacente diferente del arreglo del que **x** se refiere.

Sería fácil de copiar los elementos con bucles explícitos, pero es más fácil utilizar la función incorporada **copy**, que copia los elementos de un **slice** a otro del mismo tipo. Su primer argumento es el destino y el segundo es la fuente, que se asemeja el orden de los operandos en una asignación como **dst = src**. Los **slices** pueden hacer referencia al mismo arreglo subyacente; incluso pueden solaparse. A pesar de que se utiliza aquí, **copy** devuelve el número de elementos copiados, que es la más pequeña de las dos longitudes de **slice**, por lo que no hay peligro de quedarse fuera del final o sobrescribir algo fuera de rango.

Por eficiencia, el nuevo arreglo suele ser algo mayor que el mínimo necesario para mantener **x** e **y**. La expansión del arreglo al duplicar su tamaño en cada expansión evita un número excesivo de asignaciones y se asegura que añadiendo un solo elemento toma tiempo constante de promedio. Este programa demuestra el efecto:

```
func main() {
    var x, y []int
    for i := 0; i < 10; i++ {
        y=appendInt(x, i)
        fmt.Printf("%d cap=%d\t%v\n", i, cap(y), y)
        x=y
    }
}
```

Cada cambio en la capacidad indica una asignación y una copia:

```
0 cap=1 [0]
1 cap=2 [0 1]
2 cap=4 [0 1 2]
3 cap=4 [0 1 2 3]
4 cap=8 [0 1 2 3 4]
5 cap=8 [0 1 2 3 4 5]
6 cap=8 [0 1 2 3 4 5 6]
7 cap=8 [0 1 2 3 4 5 6 7]
8 cap=16 [0 1 2 3 4 5 6 7 8]
9 cap=16 [0 1 2 3 4 5 6 7 8 9]
```

Vamos a echar un vistazo más de cerca a la iteración **i=3**. El **slice x** contiene los tres elementos **[0 1 2]**, pero tiene la capacidad de **4**, por lo que es un solo elemento de holgura en el extremo, y **appendInt** del elemento **3** puede proceder sin reasignación. El **slice** resultante y tiene una longitud y una capacidad de **4**, y tiene el mismo arreglo subyacente como el slice original **x**, como muestra la **Figura 4.2**.

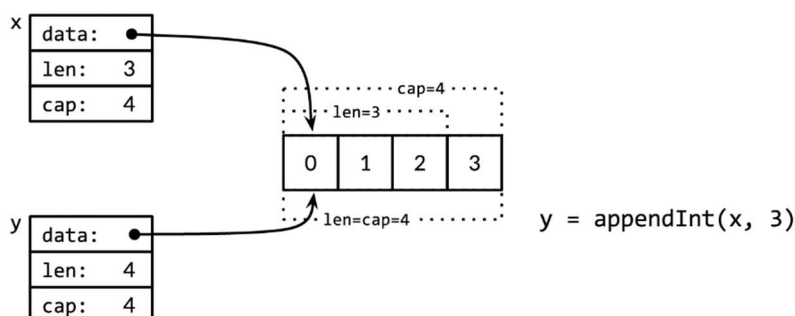


Figura 4.2. Añadir con espacio para crecer.

En la siguiente iteración, **i=4**, no hay holgura en absoluto, así **appendInt** asigna un nuevo arreglo de tamaño **8**, copia los cuatro elementos **[0 1 2 3]** de **x**, y añade **4**, el valor de **i**. El **slice** resultante y tiene una longitud de **5**, pero una capacidad de **8**; la holgura de **3** salvará a las próximas tres iteraciones de la necesidad de reasignar. Los **slices y** y **x** son vistas de diferentes arreglos. Esta operación se representa en la **Figura 4.3**.

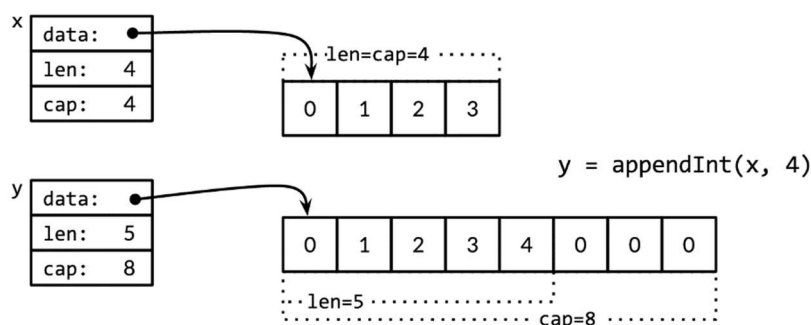


Figura 4.3. Añadir sin espacio para crecer.

La función incorporada **append** puede utilizar una estrategia de crecimiento más sofisticado que la simplista **appendInt**. Por lo general, no sabemos si una llamada dada a **append** provocará una reasignación, por lo que podemos asumir que el **slice** original hace referencia al mismo arreglo que el **slice** resultante, no se refiere a uno diferente. Del mismo modo, no debemos suponer que las operaciones sobre los elementos del antiguo **slice** estarán (o no) reflejados en el nuevo **slice**. Como resultado de ello, es habitual asignar el resultado de una llamada a **append** a la misma variable de **slice** cuyo valor pasamos a **append**:

```
runes = append(runes, r)
```

La actualización de la variable de **slice** se requiere no sólo al llamar a **append**, sino para cualquier función que pueda cambiar la longitud o la capacidad de un **slice** o hacer que se refiera a un arreglo subyacente diferente. Para utilizar **slices** correctamente, es importante tener en cuenta que, aunque los elementos del arreglo subyacente son indirectos, el puntero, la longitud y la capacidad del **slice** no lo son. Para actualizarlos se requiere una misión como la de arriba. En este sentido, los **slices** no son "puros" tipos de referencia, pero se asemejan a un tipo agregado como esta estructura:

```
type IntSlice struct {
    ptr *int
    len, cap int
}
```

La función **appendInt** añade un único elemento de un **slice**, pero la incorporada **append** nos permite añadir más de un elemento nuevo, o incluso todo un **slice**.

```
var x []int
x=append(x, 1)
x=append(x, 2, 3)
x=append(x, 4, 5, 6)
x=append(x, x...) // append the slice x
fmt.Println(x) // "[1 2 3 4 5 6 1 2 3 4 5 6]"
```

Con la pequeña modificación que se muestra a continuación, podemos coincidir con el comportamiento de la incorporada **append**. Los puntos suspensivos **"..."** en la declaración de **appendInt** hace la función **variadic**: acepta cualquier número de argumentos finales. Los puntos suspensivos correspondientes en la llamada anterior a **append** muestra cómo suministrar una lista de argumentos de un **slice**. Explicaremos este mecanismo en detalle en la **Sección 5.7**.

```
func appendInt(x []int, y ...int) []int {
    var z []int
    zlen := len(x) + len(y)
    // ...expandir z al menos en zlen...
    copy(z[len(x):], y)
    return z
}
```

La lógica de ampliar el arreglo **z** subyacente no ha cambiado y no se muestra.

4.2.2. Técnicas In-Place de Slice

Vamos a ver más ejemplos de funciones que, como **rotate** y **reverse**, modifican los elementos de un **slice** en su lugar. Dada una lista de strings, la función **noempty** devuelve los no vacíos:

```
gopl.io/ch4/nonempty
// Nonempty es un ejemplo de un algoritmo de slice in-place.
package main

import "fmt"
// nonempty devuelve un slice único que sostiene los strings no vacíos.
// El arreglo subyacente se modifica durante la llamada.
func nonempty(strings []string) []string {
    i := 0
    for _, s := range strings {
        if s != "" {
            strings[i] = s
            i++
        }
    }
    return strings[:i]
}
```

La parte sutil es que el segmento de entrada y salida del **slice** comparten el mismo arreglo subyacente. Esto evita la necesidad de asignar otro arreglo, aunque, por supuesto, los contenidos de **data** se sobrescriben en parte, como se evidencia por la segunda sentencia **print**:

```
data := []string{"one", "", "three"}
fmt.Printf("%q\n", nonempty(data)) // `["one" "three"]`
fmt.Printf("%q\n", data) // `["one" "three" "three"]`
```


Por lo tanto, lo que normalmente escribe: **data = nonempty(data)**.

La función **nonempty** también se puede escribir usando **append**:

```
func nonempty2(strings []string) []string {
    out := strings[:0] // zero-length slice of original
    for _, s := range strings {
        if s != "" {
            out = append(out, s)
        }
    }
    return out
}
```

Cualquiera que sea la variante que usemos, la reutilización de un arreglo de esta manera requiere que se produzca como máximo un valor de salida para cada valor de entrada, que es el caso de muchos algoritmos que filtran los elementos de una secuencia o combinan los adyacentes. Tal uso complejo de **slice** es la excepción y no la regla, pero puede ser claro, eficiente y útil en alguna ocasión.

Un **slice** puede ser usado para implementar una pila (**stack**). Dado un **slice** inicialmente vacío **stack**, podemos meter (**push**) un nuevo valor en el extremo del **slice** con **append**:

```
stack = append(stack, v) // push v
```

La parte superior de la pila es el último elemento:

```
top := stack[len(stack)-1] // parte superior de la pila
```

y disminuye la pila al sacar ese elemento

```
stack = stack[:len(stack)-1] // pop
```

Para eliminar un elemento de la mitad de un **slice**, preservando el orden de los elementos restantes, utilizar **copy** para deslizar los elementos de numeraciones superiores, uno por uno para llenar el vacío:

```
func remove(slice []int, i int) []int {
    copy(slice[i:], slice[i+1:])
    return slice[:len(slice)-1]
}
func main() {
    s := []int{5, 6, 7, 8, 9}
    fmt.Println(remove(s, 2)) // "[5 6 8 9]"
}
```

Y si no necesitamos preservar el orden, podemos simplemente mover el último elemento:

```
func remove(slice []int, i int) []int {
    slice[i] = slice[len(slice)-1]
    return slice[:len(slice)-1]
}
func main() {
    s := []int{5, 6, 7, 8, 9}
    fmt.Println(remove(s, 2)) // "[5 6 9 8]"
}
```

4.3. Mapas

La tabla **hash** es una de las estructuras de datos más ingeniosas y versátiles que existe. Se trata de una colección desordenada de pares **clave/valor** en el que todas las claves son distintas, y el valor asociado a una clave determinada se puede consultar, actualizar o eliminar usando un número constante de comparaciones de clave de media, no importa cuán grande sea la tabla **hash**.

En **Go**, un mapa (**map**) es una referencia a una tabla **hash**, un tipo **map** se escribe **map[K]V**, donde **K** y **V** son los tipos de sus claves y valores. Todas las claves en un mapa dado son del mismo tipo, y todos los valores son del mismo tipo, pero las claves no tienen por qué ser del mismo tipo que los valores. El tipo de clave **K** debe ser comparable usando **==**, por lo que el mapa puede comprobar si una clave determinada es igual a otra ya dentro de ella. Aunque los números de coma flotante son comparables, es una mala idea comparar **floats** para igualdad y, como hemos mencionado en el **Capítulo 3**, especialmente malo si **NaN** es un valor posible. No hay restricciones sobre el tipo de valor **V**.

Se puede utilizar la función incorporada **make** para crear un mapa:

```
ages := make(map[string]int) // mapeo de strings a enteros
```

También podemos usar un literal **map** para crear un nuevo mapa con pares **clave/valor** iniciales:

```
ages := map[string]int{
    "alice": 31,
    "charlie": 34,
}
```


Esto es equivalente a

```
ages := make(map[string]int)
ages["alice"] = 31
ages["charlie"] = 34
```

por lo que una expresión alternativa para un nuevo mapa vacío es un **map[string]int{}**.

Se accede a los elementos del mapa a través de la notación usual de subíndice:

```
ages["alice"] = 32
fmt.Println(ages["alice"]) // "32"
```

y se borra con la función incorporada **delete**:

```
delete(ages, "alice") // eliminar elemento ages["alice"]
```

Todas estas operaciones son seguras, incluso si el elemento no está en el mapa; una búsqueda de mapa usando una clave que no esté presente devuelve el valor cero de su tipo, por lo que, por ejemplo, lo siguiente funciona incluso si **"bob"** no es una clave que esté en el mapa ya que el valor de **ages["bob"]** será **0**.

```
ages["bob"] = ages["bob"] + 1 // happy birthday!
```

Las formas de asignación cortas **x += y** y **x++** también funcionan con los elementos del mapa, por lo que podemos reescribir la declaración anterior como

```
ages["bob"] += 1
```

o incluso más concisa

```
ages["bob"]++
```

Sin embargo, un elemento de mapa no es una variable, y no se puede obtener su dirección:

```
_ = &ages["bob"] // Error de compilación: No se puede obtener la dirección de un elemento de mapa
```

Una de las razones por las que no podemos obtener la dirección de un elemento de un mapa es que al crecer un mapa podría causar un **re-hashing** de los elementos existentes en nuevos lugares de almacenamiento, lo que podría invalidar la dirección.

Para enumerar todos los pares **clave/valor** en el mapa, se utiliza un bucle **range**, basado en un bucle **for** similar a los que vimos en los **slices**. Sucesivas iteraciones del bucle hacen que las variables **name** y **age** se establezcan en el siguiente par **clave/valor**:

```
for name, age := range ages {
    fmt.Printf("%s\t%d\n", name, age)
}
```

El orden de la iteración del mapa no es específico, y las diferentes implementaciones podrían utilizar una función de hash diferente, dando lugar a una ordenación diferente. En la práctica, el orden es aleatorio, variando de una ejecución a la siguiente. Esto es intencional; haciendo variar la secuencia ayuda a los programas a ser robustos a través de las implementaciones. Para enumerar los pares **clave/valor** en orden, hay que ordenar las claves de forma explícita, por ejemplo, mediante la función **Strings** del paquete **sort** si las claves son **strings**. Este es un patrón común:

```
import "sort"

var names []string
for name := range ages {
    names = append(names, name)
}
sort.Strings(names)
for _, name := range names {
    fmt.Printf("%s\t%d\n", name, ages[name])
}
```

Ya que conocemos el tamaño final de **names** desde el principio, es más eficiente asignar un arreglo del tamaño requerido por adelantado. La siguiente sentencia crea un **slice** que está inicialmente vacío, pero tiene la capacidad suficiente para contener todas las claves del mapa **ages**:

```
names := make([]string, 0, len(ages))
```

En el primer bucle **range** anterior, se requieren solamente las claves del mapa **ages**, por lo que se omite la segunda variable del bucle. En el segundo bucle, se requieren sólo los elementos del **slice names**, por lo que utilizamos el identificador en blanco **_** para ignorar la primera variable, el índice.

El valor cero para un tipo mapa es **nil**, es decir, una referencia a ninguna tabla **hash** en absoluto.

```
var ages map[string]int
fmt.Println(ages == nil) // "true"
fmt.Println(len(ages) == 0) // "true"
```

La mayoría de las operaciones con los mapas, incluyendo las operaciones de búsqueda, borrado (**delete**), cálculo de la longitud (**len**) y bucles **range**, son seguros para funcionar en una referencia de mapa **nil**, ya que se comporta como un mapa vacío. Sin embargo, el almacenamiento en un mapa **nil** provoca un pánico:

```
ages["carol"] = 21 // pánico: asignación a un mapa nil
```

Se debe reservar el mapa antes de poder almacenar en él. Al acceder a un elemento del mapa por subíndice siempre produce un valor. Si la clave está presente en el mapa, se obtiene el valor correspondiente; si no, se obtiene el valor cero para el tipo de elemento, como hemos visto con **ages["bob"]**. Para muchos propósitos esto está muy bien, pero a veces se necesita saber si el elemento realmente estaba allí o no. Por ejemplo, si el tipo de elemento es numérico, puede que haya que distinguir entre un elemento inexistente y un elemento con el valor cero, mediante una prueba de la siguiente manera:

```
age, ok := ages["bob"]
if !ok { /* "bob" no es una clave en este mapa; age == 0. */ }
```

Veremos a menudo estos dos estados combinados, como este:

```
if age, ok := ages["bob"]; !ok { /* ... */ }
```

Al sub-indexar un mapa en este contexto se obtienen dos valores; el segundo es un booleano que informa si el elemento estaba presente. La variable booleana a menudo se llama **ok**, sobre todo si se utiliza inmediatamente en una condición **if**.

Al igual que los **slices**, los mapas no pueden ser comparados entre sí; la única comparación legal es con **nil**. Para probar si dos mapas contienen las mismas claves y los mismos valores asociados, debemos escribir un bucle:

```
func equal(x, y map[string]int) bool {
    if len(x) != len(y) {
        return false
    }
    for k, xv := range x {
        if yv, ok := y[k]; !ok || yv != xv {
            return false
        }
    }
    return true
}
```

Observar cómo usamos **!ok** para distinguir los casos de "desaparecidos" y de "presentes pero cero". Si hubiéramos escrito ingenuamente **xv != y[k]**, la llamada informaría incorrectamente que sus argumentos son iguales:

```
// True si la igualdad está escrita incorrectamente.
equal(map[string]int{"A": 0}, map[string]int{"B": 42})
```

Go no proporciona un tipo conjunto (**set**), pero ya que las claves de un mapa son distintas, un mapa puede servir para este propósito. Para ilustrar, el programa **dedup** lee una secuencia de líneas y se imprime sólo la primera ocurrencia de cada línea distinta. (Es una variante del programa **dup** que mostramos en la **Sección 1.3**.) El programa **dedup** utiliza un mapa cuyas claves representan el conjunto de líneas que ya han aparecido para asegurar que los sucesos posteriores no se imprimen.

```
gopl.io/ch4/dedup
func main() {
    seen := make(map[string]bool) // a set of strings
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        line := input.Text()
        if !seen[line] {
            seen[line] = true
            fmt.Println(line)
        }
    }
    if err := input.Err(); err != nil {
        fmt.Fprintf(os.Stderr, "dedup: %v\n", err)
        os.Exit(1)
    }
}
```

Los programadores de **Go** a menudo describen un mapa como un "conjunto de **strings**" sin más preámbulos, pero cuidado, no todos los valores **map[string]bool** son simples conjuntos; algunos pueden contener valores **true** y **false**.

A veces necesitamos un mapa o un conjunto cuyas claves son **slices**, pero como las claves de un mapa deben ser comparables, esto no puede expresarse directamente. Sin embargo, se puede hacer en dos pasos. En primer lugar, se define una función auxiliar **k** que mapea cada clave en un **string**, con la propiedad de que **k(x) == k(y)** si y sólo si consideramos **x** e **y** equivalentes. Entonces se crea un mapa cuyas claves son **strings**, aplicando la función de ayuda para cada clave antes de acceder al mapa.

El ejemplo siguiente utiliza un mapa para registrar el número de veces que **Add** se ha llamado con una lista dada de **strings**. Utiliza **fmt.Sprintf** para convertir un **slice** de **strings** en un solo **string** que es una adecuada clave de mapa, citando a cada elemento del **slice** con **%q** para guardar los límites fielmente:

```

var m = make(map[string]int)
func k(list []string) string { return fmt.Sprintf("%q", list) }
func Add(list []string) { m[k(list)]++ }
func Count(list []string) int { return m[k(list)] }

```

El mismo enfoque se puede utilizar para cualquier tipo de clave no comparable, no sólo **slices**. Es incluso útil para este tipo de clave comparable cuando se desea una definición de la igualdad que no sea **==**, tales como comparaciones entre mayúsculas y minúsculas para los **strings**. Y el tipo de **k(x)** no necesita ser un **string**; cualquier tipo comparable con la propiedad de equivalencia deseada lo va a hacer, tales como enteros, arreglos o estructuras.

Aquí vemos otro ejemplo de mapas en acción, un programa que cuenta las ocurrencias de cada punto de código **Unicode** distinto en su entrada. Puesto que hay un gran número de caracteres posibles, sólo aparecerá una pequeña fracción de un documento en particular, un mapa es una forma natural para mantener un registro de sólo los que se han visto y su conteo correspondiente.

```

gopl.io/ch4/charcount
// Charcount calcula la cantidad de caracteres Unicode
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
    "unicode"
    "unicode/utf8"
)

func main() {
    counts := make(map[rune]int) // conteo de caracteres Unicode
    var utfflen [utf8.UTFMax + 1]int // conteo de la longitud de la codificación UTF-8
    invalid := 0 // conteo de caracteres UTF-8 inválidos
    in := bufio.NewReader(os.Stdin)
    for {
        r, n, err := in.ReadRune() // retorna rune, nbytes, error
        if err == io.EOF {
            break
        }
        if err != nil {
            fmt.Fprintf(os.Stderr, "charcount: %v\n", err)
            os.Exit(1)
        }
        if r == unicode.ReplacementChar && n == 1 {
            invalid++
            continue
        }
        counts[r]++
        utfflen[n]++
    }
    fmt.Printf("rune\tcount\n")
    for c, n := range counts {
        fmt.Printf("%q\t%d\n", c, n)
    }
    fmt.Print("\nlen\tcount\n")
    for i, n := range utfflen {
        if i > 0 {
            fmt.Printf("%d\t%d\n", i, n)
        }
    }
    if invalid > 0 {
        fmt.Printf("\n%d invalid UTF-8 characters\n", invalid)
    }
}

```

El método **ReadRune** realiza la decodificación **UTF-8** y devuelve tres valores: la runa decodificada, la longitud en bytes de la codificación **UTF-8**, y un valor de error. El único error que esperamos es de fin de archivo. Si la entrada no tiene una codificación legal **UTF-8** de una runa, la runa devuelta es **unicode.ReplacementChar** y la longitud será **1**.

El programa **charcount** también imprime un recuento de las longitudes de las codificaciones **UTF-8** de las runas que aparecen en la entrada. Un mapa no es la mejor estructura de datos para esto; ya que se codifican longitudes que varían sólo de **1** a **utf8.UTFMax** (que tiene el valor **4**), un arreglo es más compacto.

A modo de experimento, ejecutamos **charcount** en este libro en un punto. A pesar de que es en su mayoría en inglés, por supuesto, tiene un buen número de caracteres no ASCII. Éstos son los diez primeros:

° 27 世 15 界 14 é 13 × 10 ≤ 5 × 5 ☺ 4 ♦ 4 □ 3

y aquí está la distribución de las longitudes de todas las codificaciones UTF-8:

```
len count
1 765391
2 60
3 70
4 0
```

El tipo del valor de un mapa puede ser en sí mismo un tipo compuesto, tal como un mapa o **slice**. En el siguiente código, el tipo de clave de **graph** es un **string** y el tipo de valor es un **map[string]bool**, que representa un conjunto de **strings**. Conceptualmente, **graph** asigna un **string** a un conjunto de **strings** relacionados, sus sucesores en un grafo dirigido.

```
gopl.io/ch4/graph
var graph = make(map[string]map[string]bool)
func addEdge(from, to string) {
    edges := graph[from]
    if edges == nil {
        edges = make(map[string]bool)
        graph[from] = edges
    }
    edges[to] = true
}
func hasEdge(from, to string) bool {
    return graph[from][to]
}
```

La función **addEdge** muestra la forma idiomática para rellenar un mapa con pereza, es decir, para inicializar cada valor tal como aparece su clave para la primera vez. La función **hasEdge** muestra cómo funciona el valor cero de una entrada de mapa que falta: aunque ninguno **from** **nor** **to** está presente, **graph[from][to]** siempre dará un resultado significativo.

4.4. Estructuras

A **struct** es un tipo de datos agregados que agrupa a cero o más valores nombrados de tipos arbitrarios como una sola entidad. Cada valor se denomina campo. El ejemplo clásico de una estructura de procesamiento de datos es un registro de empleado, cuyos campos son un identificador único **ID**, el nombre del empleado, dirección, fecha de nacimiento, cargo, salario, gerente, y similares. Todos estos campos se recogen en una sola entidad que se puede copiar como una unidad, pasar a funciones y ser devuelta por ellas, almacenada en arreglos, etc.

Estas dos sentencias declaran un tipo de estructura llamada **Employee** y una variable llamada **dilbert** que es una instancia de un **Employee**:

```
type Employee struct {
    ID int
    Name string
    Address string
    DoB time.Time
    Position string
    Salary int
    ManagerID int
}
var dilbert Employee
```

Se accede a los campos individuales de **dilbert** usando la notación de punto como **dilbert.Name** y **dilbert.DoB**. Debido a que **dilbert** es una variable, sus campos son variables también, así que se puede asignar a un campo:

```
dilbert.Salary -= 5000 // degradado, para escribir muy pocas líneas de código
```

o tomar su dirección y acceder a ella a través de un puntero:

```
position := &dilbert.Position
*position = "Senior " + *position // promovido, para la contratación externa de Elbonia
```

La notación de punto también trabaja con un puntero a una estructura:

```
var employeeOfTheMonth *Employee = &dilbert
employeeOfTheMonth.Position += " (proactive team player)"
```

La última sentencia es equivalente a

```
(*employeeOfTheMonth).Position += " (proactive team player)"
```

Teniendo en cuenta que un empleado tiene un identificador único **ID**, la función **EmployeeByID** devuelve un puntero a una estructura **Employee**. Podemos utilizar la notación de punto para acceder a sus campos:

```
func EmployeeByID(id int) *Employee { /* ... */ }
fmt.Println(EmployeeByID(dilbert.ManagerID).Position) // "Jefe de pelo puntiagudo"
id := dilbert.ID
EmployeeByID(id).Salary = 0 // Despedido por ninguna razón...
```

La última sentencia actualiza la estructura **Employee** que está apuntada por el resultado de la llamada a **EmployeeByID**. Si el tipo de resultado de **EmployeeByID** se cambió a **Employee** en lugar de ***Employee**, la sentencia de asignación no compilará ya que en el lado izquierdo no se identificó una variable.

Los campos se escriben generalmente uno por línea, con los nombres del campo precediendo a su tipo, pero los campos consecutivos del mismo tipo se pueden combinar, como con **Name** y **Address** aquí:

```
type Employee struct {
    ID int
    Name, Address string
    DoB time.Time
    Position string
    Salary int
    ManagerID int
}
```

El orden de los campos es importante para la identidad del tipo. Si hubiéramos combinado también la declaración del campo **Position** (también un **string**), o intercambiado **Name** y **Address**, estaríamos definiendo un tipo de estructura diferente. Por lo general, sólo combinamos las declaraciones de campos relacionados.

El nombre de un campo de estructura se exporta si comienza con una letra mayúscula; esto es un mecanismo principal de control de acceso de **Go**. Un tipo de estructura puede contener una mezcla de campos exportados y no exportados.

Los tipos **struct** tienden a ser más detallados, ya que a menudo implican una línea para cada campo. Aunque podríamos escribir todo el tipo cada vez que sea necesario, la repetición cansa. Los tipos **struct** generalmente aparecen después de la declaración de un tipo con nombre como **Employee**.

Un tipo de **struct** con nombre **S** no puede declarar un campo del mismo tipo **S**: un valor agregado no puede contenerse a sí mismo. (Una restricción análoga se aplica a los arreglos.) Pero **S** puede declarar un campo del tipo de puntero ***S**, lo que nos permite crear estructuras de datos recursivas como listas enlazadas y árboles. Esto se ilustra en el siguiente código, que utiliza un árbol binario para implementar una ordenación de inserción:

```
gopl.io/ch4/treesort
type tree struct {
    value int
    left, right *tree
}
// Sort ordena valores en el lugar.
func Sort(values []int) {
    var root *tree
    for _, v := range values {
        root = add(root, v)
    }
    appendValues(values[:0], root)
}
// appendValues añade los elementos de t a los valores en orden
// y retorna un slice como resultado.
func appendValues(values []int, t *tree) []int {
    if t != nil {
        values = appendValues(values, t.left)
        values = append(values, t.value)
        values = appendValues(values, t.right)
    }
    return values
}
func add(t *tree, value int) *tree {
    if t == nil {
        // Equivalente a return &tree{value: value}.
        t = new(tree)
        t.value = value
        return t
    }
    if value < t.value {
        t.left = add(t.left, value)
    } else {
        t.right = add(t.right, value)
    }
    return t
}
```

El valor cero de una estructura se compone de los valores cero de cada uno de sus campos. Por lo general es deseable que el valor cero sea natural o sensible por defecto. Por ejemplo, en **bytes.Buffer**, el valor inicial de la estructura es un buffer vacío listo para su uso, y el valor cero de **sync.Mutex**, que veremos en el **Capítulo 9**, es un producto listo para usar un **mutex** desbloqueado. A veces, este comportamiento inicial sensible que ocurre de libremente, pero a veces el diseñador de tipos tiene que trabajar en ello.

El tipo **struct** sin campos se llama una estructura vacía, se escribe como **struct{}**. Tiene tamaño cero y no lleva ninguna información, no obstante, puede ser útil. Algunos programadores de **Go** las utilizan en lugar de **bool** como el tipo de valor de un mapa que representa un conjunto, para hacer hincapié en que sólo las claves son significativas, pero el ahorro de espacio es marginal y la sintaxis más engorrosa, por lo que generalmente hay que evitarlo.

```
seen := make(map[string]struct{}) // conjunto de strings
// ...
if _, ok := seen[s]; !ok {
    seen[s] = struct{}{}
    // ...first time seeing s...
}
```

4.4.1. Literales de struct

Se puede escribir un valor de un tipo **struct** utilizando un literal **struct** que especifica los valores de sus campos.

```
type Point struct{ X, Y int }
p := Point{1, 2}
```

Hay dos formas de literal de estructura. La primera forma, que se muestra arriba, requiere que se especifique un valor para cada campo, en el orden correcto. Da trabajo al escritor (y al lector) al tener que recordar exactamente que son los campos, y hace que el código sea más frágil si más adelante el conjunto de campos crece o es reordenado. En consecuencia, esta forma tiende a ser utilizada sólo dentro del paquete que define el tipo **struct** o con tipos pequeños de **struct** para los cuales existe una convención de ordenamiento de campo obvio, como **image.Point{x, y}** o **color.RGBA{red, green, blue, alpha}**.

se utiliza más a menudo, una segunda forma, en la que un valor **struct** se inicializa haciendo una lista de algunos o todos los nombres de los campos y sus valores correspondientes, como en esta sentencia del programa **Lissajous** de la **Sección 1.4**:

```
anim := gif.GIF{LoopCount: nframes}
```

Si un campo se omite en este tipo de literal, se le establece el valor cero para su tipo. Debido a que se proporcionan nombres, no importa el orden de los campos.

Las dos formas no se pueden mezclar en el mismo literal. Tampoco se puede utilizar la primera forma de literal (basada en el orden) para esconder la regla de que los identificadores no exportados no se puede referenciar en otro paquete.

```
package p
type T struct{ a, b int } // a y b no están exportadas

package q
import "p"
var _ = p.T{a: 1, b: 2} // error de compilación: No se puede hacer referencia a, b
var _ = p.T{1, 2} // error de compilación: No se puede hacer referencia a, b
```

A pesar de que la última línea anterior no menciona los identificadores de campo no exportados, realmente son utilizados de forma implícita, por lo que no son permitidos.

Los valores **struct** se pueden pasar como argumentos a funciones y retornar de ellas. Por ejemplo, esta función escala un **Point** un factor especificado:

```
func Scale(p Point, factor int) Point {
    return Point{p.X * factor, p.Y * factor}
}

fmt.Println(Scale(Point{1, 2}, 5)) // "{5 10}"
```

Para una mayor eficiencia, los tipos **struct** más grandes suelen ser enviados o retornados de las funciones indirectamente usando un puntero,

```
func Bonus(e *Employee, percent int) int {
    return e.Salary * percent / 100
}
```

y esto es necesario si la función debe modificar su argumento, ya que, en un lenguaje de llamada por valor, como **Go**, la función llamada recibe sólo una copia de un argumento, no una referencia al argumento original.

```
func AwardAnnualRaise(e *Employee) {
    e.Salary = e.Salary * 105 / 100
}
```

Debido a que las estructuras son tan comúnmente tratadas a través de punteros, es posible utilizar esta notación abreviada para crear e inicializar un **struct** variable y obtener su dirección:

```
pp := &Point{1, 2}
```

Es exactamente equivalente a

```
pp := new(Point)
*pp = Point{1, 2}
```

Pero **&Point{1, 2}** se puede utilizar directamente dentro de una expresión, como una llamada de función.

4.4.2. Comparando Structs

Si todos los campos de una estructura son comparables, la estructura en sí es comparable, por lo que dos expresiones de ese tipo pueden ser comparadas usando **==** o **!=**. La operación **==** compara los campos correspondientes de las dos estructuras en orden, por lo que las dos expresiones impresas a continuación son equivalentes:

```
type Point struct{ X, Y int }

p := Point{1, 2}
q := Point{2, 1}
fmt.Println(p.X == q.X && p.Y == q.Y) // "false"
fmt.Println(p == q) // "false"
```

Se pueden utilizar tipos **struct** comparables, al igual que otros tipos comparables, como el tipo de la clave de un mapa.

```
type address struct {
    hostname string
    port int
}
hits := make(map[address]int)
hits[address{"golang.org", 443}]++
```

4.4.3. Incrustación de Struct y campos Anónimos

En esta sección, vamos a ver cómo un mecanismo inusual de incrustación de **struct** nos permite usar un tipo de estructura con nombre como un campo anónimo de otro tipo de estructura, proporcionando un conveniente atajo sintáctico para que una expresión de punto simple como **x.f** pueda representar una cadena de campos como **x.d.e.f**.

Considerar la posibilidad de un programa de dibujo **2-D** que proporciona una librería de formas, como rectángulos, elipses, estrellas, y ruedas. Aquí están dos de los tipos que se podrían definir:

```
type Circle struct {
    X, Y, Radius int
}

type Wheel struct {
    X, Y, Radius, Spokes int
}
```

Un **Circle** tiene campos para las coordenadas **X** e **Y** de su centro y un **Radius**. Una **Wheel** (rueda) tiene todas las características de un **Circle**, además de **Spokes**, el número de radios radiales inscritos. Vamos a crear una rueda:

```
var w Wheel
w.X = 8
w.Y = 8
w.Radius = 5
w.Spokes = 20
```

A medida que el conjunto de formas crece, estamos obligados a notar similitudes y repetición entre ellos, lo que puede ser conveniente para factorizar sus partes comunes:

```
type Point struct {
    X, Y int
}

type Circle struct {
    Center Point
    Radius int
}

type Wheel struct {
    Circle Circle
    Spokes int
}
```


La aplicación puede ser más clara, pero este cambio hace que el acceso a los campos de una *Wheel* sea más detallado:

```
var w Wheel
w.Circle.Center.X = 8
w.Circle.Center.Y = 8
w.Circle.Radius = 5
w.Spokes = 20
```

Go nos permite declarar un campo con un tipo, pero no un nombre; estos campos son llamados **campos anónimos**. El tipo del campo debe ser un tipo con nombre o un puntero a un tipo con nombre. A continuación, **Circle** y **Wheel** tienen un campo anónimo cada uno. Se dice que un **Point** está incrustado (**embedded**) dentro de **Circle**, y un **Circle** está incrustado dentro de **Wheel**.

```
type Circle struct {
    Point
    Radius int
}

type Wheel struct {
    Circle
    Spokes int
}
```

Gracias a la incrustación, se puede hacer referencia a los nombres de las hojas del implícito árbol, sin dar los nombres que intervienen:

```
var w Wheel
w.X = 8 // equivalente a w.Circle.Point.X = 8
w.Y = 8 // equivalente a w.Circle.Point.Y = 8
w.Radius = 5 // equivalente a w.Circle.Radius = 5
w.Spokes = 20
```

Las formas explícitas que se muestran en los comentarios anteriores siguen siendo válidas, lo que demuestra que "el campo anónimo" es un término equivocado. Los campos **Circle** y **Point** tienen nombres, el del tipo nombrado, pero esos nombres son opcionales en las expresiones de punto. Podemos omitir cualquiera o todos los campos anónimos a la hora de seleccionar sus subcampos.

Por desgracia, no existe una correspondiente forma abreviada de la sintaxis del literal de estructura, por lo que ninguno de ellos se podrá compilar:

```
w=Wheel{8, 8, 5, 20} // Error de compilación: campos desconocidos
w=Wheel{X: 8, Y: 8, Radius: 5, Spokes: 20} // Error de compilación: campos desconocidos
```

El literal de estructura debe seguir la forma de la declaración de tipo, por lo que se debe utilizar una de las dos formas siguientes, que son equivalentes entre sí:

```
gopl.io/ch4/embed
w=Wheel{Circle{Point{8, 8}, 5}, 20}
w=Wheel{
    Circle: Circle{
        Point: Point{X: 8, Y: 8},
        Radius: 5,
    },
    Spokes: 20, // NOTA: es necesaria la coma adicional aquí (y en Radius)
}
fmt.Printf("%#v\n", w)
// Salida:
// Wheel{Circle:Circle{Point:Point{X:8, Y:8}, Radius:5}, Spokes:20}
w.X = 42
fmt.Printf("%#v\n", w)
// Salida:
// Wheel{Circle:Circle{Point:Point{X:42, Y:8}, Radius:5}, Spokes:20}
```

Nótese cómo el adverbio **#** causa el verbo **%v** de **Printf** para mostrar valores en una forma similar a la sintaxis de **Go**. Para valores **struct**, esta forma incluye el nombre de cada campo.

Debido a que los campos "anónimos" tienen nombres implícitos, no se pueden tener dos campos anónimos del mismo tipo, ya que sus nombres podrían entrar en conflicto. Y debido a que el nombre del campo se determina implícitamente por su tipo, también lo es la visibilidad del campo. En los ejemplos anteriores, los campos anónimos **Point** y **Circle** se exportan. Si hubieran sido dejados (**point** y **circle**), podríamos usar la forma abreviada

```
w.X = 8 // equivalente a w.circle.point.X = 8
```

pero la forma larga explícita que se muestra en el comentario estaría prohibida fuera del paquete debido a que **circle** y **point** serían inaccesibles.

Lo que hemos visto hasta ahora de la incrustación de estructura es sólo una pizca de azúcar sintáctico en la notación de punto que se utiliza para seleccionar campos de **struct**. Más tarde, veremos que los campos anónimos no tienen

que ser solo los tipos **struct**; cualquier tipo con nombre o puntero a un tipo con nombre podrá ser. Pero ¿por qué se va a querer incrustar un tipo que no tiene subcampos?

La respuesta tiene que ver con los métodos. La notación abreviada utilizada para la selección de los campos de un tipo incrustado funciona para la selección de sus métodos también. En efecto, el tipo de estructura exterior no sólo puede tener campos de tipo incrustado, también sus métodos. Este mecanismo es la principal forma en la que los comportamientos de objetos complejos se componen de otros más simples. La composición es fundamental para la programación orientada a objetos en **Go**, la exploraremos más a fondo en la **Sección 6.3**.

4.5. JSON

JavaScript Object Notation (JSON) es una notación estándar para el envío y recepción de información estructurada. **JSON** no es el único ejemplo de notación. **XML (§7.14)**, **ASN.1**, y **Google Protocol Buffers** sirven propósitos similares y cada uno tiene su nicho, pero debido a su simplicidad, facilidad de lectura, y un soporte universal, **JSON** es el más ampliamente utilizado.

Go tiene un excelente soporte para la codificación y decodificación de estos formatos, proporcionado por los paquetes estándar de la librería de **encoding/json**, **encoding/xml**, **encoding/asn1**, etc., todos estos paquetes tienen **APIs** similares. En esta sección se presenta un breve resumen de las partes más importantes del paquete **encoding/json**.

JSON es una codificación de valores de **JavaScript**, **strings**, números, booleanos, arreglos y objetos, como texto **Unicode**. Es una representación legible y eficiente de los tipos de datos básicos del **Capítulo 3** y los tipos compuestos de este capítulo, arreglos, **slices**, estructuras, y mapas.

Los tipos **JSON** básicos son números (en notación decimal o científica), booleanos (**true** o **false**), y **strings**, que son secuencias de puntos de código **Unicode** entre comillas dobles, con una barra invertida de escape utilizando una notación similar a **Go**, aunque los escape numéricas **\Uhhhh** de **JSON** denotan códigos **UTF-16**, no runas.

Estos tipos básicos se pueden combinar de forma recursiva utilizando arreglos **JSON** y objetos. Un arreglo **JSON** es una secuencia ordenada de valores, escrita como una lista separada por comas entre corchetes; Los arreglos **JSON** se utilizan para codificar los arreglos y los **slices** de **Go**. Un objeto **JSON** es un mapeo de **strings** a valores, escrito como una secuencia de pares **name:value** separados por comas y rodeados de llaves; los objetos **JSON** se utilizan para codificar mapas **Go** (con claves de **strings**) y estructuras. Por ejemplo:

```
boolean    true
number     -273.15
string     "She said \"Hello, BF\""
array      ["gold", "silver", "bronze"]
object     {"year": 1980,
           "event": "archery",
           "medals": ["gold", "silver", "bronze"]}
```

Considerar una aplicación que reúne críticas de películas y ofrece recomendaciones. El tipo de datos **Movie** es una lista típica de los valores que se declaran a continuación. (Los literales de **string** después de las declaraciones **Year** y **Color** son etiquetas de campo, lo explicaremos en un momento).

```
gopl.io/ch4/movie
type Movie struct {
    Title string
    Year int `json:"released"`
    Color bool `json:"color,omitempty"`
    Actors []string
}
var movies = []Movie{
    {Title: "Casablanca", Year: 1942, Color: false,
     Actors: []string{"Humphrey Bogart", "Ingrid Bergman"}},
    {Title: "Cool Hand Luke", Year: 1967, Color: true,
     Actors: []string{"Paul Newman"}},
    {Title: "Bullitt", Year: 1968, Color: true,
     Actors: []string{"Steve McQueen", "Jacqueline Bisset"}},
    // ...
}
```

Las estructuras de datos de este tipo son un excelente complemento para **JSON**, y son fáciles de convertir en ambas direcciones. La conversión de una estructura de datos **Go** como **movies** a **JSON** se denomina **marshaling**. El **marshaling** se realiza mediante **json.Marshal**:

```
data, err := json.Marshal(movies)
if err != nil {
    log.Fatalf("JSON marshaling failed: %s", err)
}
fmt.Printf("%s\n", data)
```

Marshal produce una **slice** de bytes que contiene un **string** muy largo sin espacios en blanco extraños; hemos doblado las líneas para que quepan:

```
[{"Title": "Casablanca", "released": 1942, "Actors": ["Humphrey Bogart", "Ingrid Bergman"]}, {"Title": "Cool Hand Luke", "released": 1967, "color": true, "Actors": ["Paul Newman"]}, {"Title": "Bullitt", "released": 1968, "color": true, "Actors": ["Steve McQueen", "Jacqueline Bisset"]}]]
```

Esta representación compacta contiene toda la información, pero es difícil de leer. Para el consumo humano, una variante llamada **json.MarshalIndent** produce una salida perfectamente indentada. Dos argumentos adicionales definen un prefijo para cada línea de salida y un **string** para cada nivel de indentación:

```
data, err := json.MarshalIndent(movies, "", " ")
if err != nil {
    log.Fatalf("JSON marshaling failed: %s", err)
}
fmt.Printf("%s\n", data)
```

El código anterior imprime

```
[
  {
    "Title": "Casablanca",
    "released": 1942,
    "Actors": [
      "Humphrey Bogart",
      "Ingrid Bergman"
    ]
  },
  {
    "Title": "Cool Hand Luke",
    "released": 1967,
    "color": true,
    "Actors": [
      "Paul Newman"
    ]
  },
  {
    "Title": "Bullitt",
    "released": 1968,
    "color": true,
    "Actors": [
      "Steve McQueen",
      "Jacqueline Bisset"
    ]
  }
]
```

El **marshaling** utiliza los nombres de los campos de la estructura **Go** como los nombres de los campos de los objetos **JSON** (a través de la reflexión, que veremos en la **Sección 12.6**). Sólo los campos exportados se calculan, por lo que se optó por nombres en mayúsculas para todos los nombres de campo de **Go**.

Se puede haber notado que el nombre del campo **Year** cambia a **released** en la salida, y **Color** cambió a **color**. Esto es debido a las etiquetas de campo. Una etiqueta de campo es un **string** de metadatos asociados en tiempo de compilación con el campo de una estructura:

```
Year int `json:"released"`
Color bool `json:"color,omitempty"`
```

Una etiqueta de campo puede ser cualquier literal de **string**, pero es interpretado de forma convencional como una lista separada por espacios de pares **clave:"valor"**; ya que contienen comillas dobles, las etiquetas de campos se escriben normalmente con literales de **string** en bruto. La clave **json** controla el comportamiento del paquete **encoding/json**, y otros paquetes **encoding/...** siguen esta convención. La primera parte de la etiqueta de campo **json** especifica un nombre alternativo **JSON** para el campo de **Go**. Las etiquetas de campo a menudo se utilizan para especificar un nombre de **JSON** idiomático como **total_count** para un campo de **Go** llamado **TotalCount**. La etiqueta para **Color** tiene una opción adicional, **omitempty**, que indica que no debe ser producida salida **JSON** si el campo tiene el valor cero para su tipo (**false**, aquí) o está vacío. Efectivamente, la salida **JSON** para **Casablanca**, una película en blanco y negro, no tiene campo **color**.

La operación inversa del **marshaling**, decodificar **JSON** llenando una estructura de datos **Go**, se llama **unmarshaling**, y se realiza por **json.Unmarshal**. El código de abajo hace **unmarshaling** de los datos de la película **JSON** en un **slice** de estructuras cuyo único campo es **Title**. Mediante la definición de estructuras de datos de **Go** adecuadas de esta manera, podemos seleccionar qué partes de la entrada de **JSON** se debe decodificar y que partes descartar. Cuando **Unmarshal** regresa, ha llenado en el **slice** la información de **Title**; otros nombres en el **JSON** se ignoran.

```

var titles []struct{ Title string }
if err := json.Unmarshal(data, &titles); err != nil {
    log.Fatalf("JSON unmarshaling failed: %s", err)
}
fmt.Println(titles) // "[{Casablanca} {Cool Hand Luke} {Bullitt}]"

```

Muchos servicios web proporcionan una interfaz **JSON**, al hacer una solicitud con **HTTP** devuelve la información deseada en formato **JSON**. Para ilustrar esto, vamos a interrogar al gestor de **GitHub** utilizando su interfaz de servicios web. En primer lugar, vamos a definir los tipos y constantes necesarias:

```

gopl.io/ch4/github
// Paquete github que provee una API a Go para el GitHub issue tracker.
// Ver https://developer.github.com/v3/search/#search-issues.
package github
import "time"
const IssuesURL = "https://api.github.com/search/issues"
type IssuesSearchResult struct {
    TotalCount int `json:"total_count"`
    Items []*Issue
}
type Issue struct {
    Number int
    HTMLURL string `json:"html_url"`
    Title string
    State string
    User *User
    CreatedAt time.Time `json:"created_at"`
    Body string // en formato Markdown
}
type User struct {
    Login string
    HTMLURL string `json:"html_url"`
}

```

Al igual que antes, los nombres de todos los campos del **struct** deben ser capitalizados incluso si sus nombres no son **JSON**. Sin embargo, el proceso de correspondencia que asocia nombres **JSON** con nombres de estructura **Go** durante **unmarshaling** es sensitivo a las mayúsculas y minúsculas, por lo que sólo es necesario utilizar una etiqueta de campo cuando hay un guión bajo en el nombre **JSON** pero no en el nombre de **Go**. Una vez más, estamos siendo selectivos sobre qué campos decodificar; la respuesta de búsqueda **GitHub** contiene mucha más información que la que mostramos aquí.

La función **SearchIssues** realiza una petición **HTTP** y decodifica el resultado como **JSON**. Dado que los términos de la consulta presentados por un usuario podría contener caracteres cómo **?** y **&** que tienen un significado especial en una **URL**, utilizamos **url.QueryEscape** para asegurarse de que se toman literalmente.

```

gopl.io/ch4/github
package github
import (
    "encoding/json"
    "fmt"
    "net/http"
    "net/url"
    "strings"
)

// SearchIssues pregunta al GitHub issue tracker.
func SearchIssues(terms []string) (*IssuesSearchResult, error) {
    q := url.QueryEscape(strings.Join(terms, " "))
    resp, err := http.Get(IssuesURL + "?q=" + q)
    if err != nil {
        return nil, err
    }
    // Debemos cerrar resp.Body en todas las ejecuciones de caminos.
    // (El capítulo 5 presenta 'defer', que hace esto más simple.)
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("search query failed: %s", resp.Status)
    }
    var result IssuesSearchResult
    if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {
        resp.Body.Close()
        return nil, err
    }
    resp.Body.Close()
    return &result, nil
}

```

Los ejemplos anteriores utilizan **json.Unmarshal** para descifrar todo el contenido de un **slice** de bytes como una sola entidad **JSON**. Para variar, este ejemplo utiliza un decodificador de **streaming**, **json.Decoder**, que permite a varias entidades **JSON** ser decodificadas en secuencia desde el mismo **stream**, a pesar de que no necesitamos esa característica aquí. Como era de esperar, hay un codificador de **streaming** correspondiente llamado **json.Encoder**.

La llamada a **Decode** rellena la variable **result**. Hay varias formas en las que puede dar formato a su valor muy bien. La más simple, demostrada por el comando **issues** a continuación, es como una tabla de texto con columnas de ancho fijo, pero en la siguiente sección veremos un enfoque más sofisticado basado en plantillas.

```
gopl.io/ch4/issues
// Issues imprime una tabla de incidencias de GitHub coincidentes con los términos de búsqueda.
package main
import (
    "fmt"
    "log"
    "os"
    "gopl.io/ch4/github"
)
func main() {
    result, err := github.SearchIssues(os.Args[1:])
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%d issues:\n", result.TotalCount)
    for _, item := range result.Items {
        fmt.Printf("#%-5d %9.9s %.55s\n",
            item.Number, item.User.Login, item.Title)
    }
}
```

Los argumentos de línea de comandos especifican los términos de búsqueda. El siguiente comando consulta el proyecto de **Go** de seguimiento de incidencias de la lista de errores abiertos relacionados con la decodificación **JSON**:

```
$ go build gopl.io/ch4/issues
$ ./issues repo:golang/go is:open json decoder
13 issues:
#5680 eaigner encoding/json: set key converter on en/decoder
#6050 gopherbot encoding/json: provide tokenizer
#8658 gopherbot encoding/json: use bufio
#8462 kortschak encoding/json: UnmarshalText confuses json.Unmarshal
#5901 rsc encoding/json: allow override type marshaling
#9812 klauspost encoding/json: string tag not symmetric
#7872 extempora encoding/json: Encoder internally buffers full output
#9650 cespere encoding/json: Decoding gives errPhase when unmarshalin
#6716 gopherbot encoding/json: include field name in unmarshal error me
#6901 lukescott encoding/json, encoding/xml: option to treat unknown fi
#6384 joeshaw encoding/json: encode precise floating point integers u
#6647 btracey x/tools/cmd/godoc: display type kind of each named type
#4237 gjemiller encoding/base64: URLEncoding padding is optional
```

La interfaz de servicios web en **GitHub** <https://developer.github.com/v3/> tiene muchas más funciones que estas, ya que no tenemos espacio aquí.

4.6. Las plantillas de texto y HTML

El ejemplo anterior hace sólo el formato más simple posible, para lo cual **Printf** es del todo adecuado. Pero a veces formatear debe ser más elaborado, y es deseable separar el formato del código de forma más completa. Esto se puede hacer con los paquetes **text/template** y **html/template**, que proporcionan un mecanismo para la sustitución de los valores de las variables en una plantilla de texto o **HTML**.

Una plantilla es un **string** o un archivo que contiene una o más partes encerradas entre llaves dobles, **{{...}}**, denominada **acciones**. La mayor parte del **string** se imprime, literalmente, pero las acciones desencadenan otros comportamientos. Cada acción contiene una expresión en el lenguaje de plantillas, una notación sencilla pero poderosa para los valores de impresión, seleccionando campos **struct**, llamando a funciones y métodos, expresando el flujo de control, como sentencias **if - else** y bucles **range**, y creando instancias de otras plantillas. A continuación, se muestra una simple **string** de plantilla:

```
gopl.io/ch4/issuesreport
const templ = `{{.TotalCount}} issues:
{{range .Items}}-----
Number: {{.Number}}
User: {{.User.Login}}
Title: {{.Title | printf "%.64s"}}
Age: {{.CreatedAt | daysAgo}} days
{{end}}`
```

Esta plantilla imprime en primer lugar el número de incidencias coincidentes, a continuación, imprime el número, el usuario, el título y la edad en días de cada uno. Dentro de una acción, hay una noción del valor actual, se hace referencia como "punto" y se escribe como ".", un punto. El punto se refiere inicialmente al parámetro de la plantilla, que será un **github.IssuesSearchResult** en este ejemplo. La acción **{{.TotalCount}}** expande al valor del campo **TotalCount**, impreso en la forma habitual. Las acciones **{{range .Items}}** y **{{end}}** crean un bucle, por lo que el texto entre ellas se expande en múltiples ocasiones, con el punto ligado a elementos sucesivos de **Items**.

Dentro de una acción, la notación **|** hace que el resultado de una operación el argumento de otra, de forma análoga a una tubería **shell Unix**. En el caso del **Title**, la segunda operación es la función **printf**, que es un sinónimo incorporado para **fmt.Sprintf** en todas las plantillas. Para **Age**, la segunda operación es la siguiente función, **daysAgo**, que convierte el campo **CreatedAt** en un tiempo transcurrido, usando **time.Since**:

```
func daysAgo(t time.Time) int {
    return int(time.Since(t).Hours() / 24)
}
```

Observar que el tipo de **CreatedAt** es **time.time**, no **string**. De la misma forma que un tipo puede controlar su formato de **string** (§2.5) mediante la definición de ciertos métodos, un tipo puede definir también métodos para controlar el comportamiento **marshaling** y **unmarshaling JSON**. El valor **JSON hecho marshaling** de un **time.Time** es un **string** en un formato estándar.

La producción de salida con una plantilla es un proceso de dos pasos. En primer lugar, debemos analizar la plantilla en una representación interna adecuada, y luego ejecutarla en las entradas específicas. El análisis tiene que hacerse sólo una vez. El código siguiente crea y analiza la plantilla **templ** definida anteriormente. Tener en cuenta el encadenamiento de llamadas a métodos: **template.New** crea y devuelve una plantilla; **Funcs** que añade **daysAgo** al conjunto de funciones accesibles dentro de esta plantilla, a continuación, devuelve esa plantilla; Por último, **Parse** se llama en el resultado.

```
report, err := template.New("report").
    Funcs(template.FuncMap{"daysAgo": daysAgo}).
    Parse(templ)
if err != nil {
    log.Fatal(err)
}
```

Dado que las plantillas suelen ser fijas en tiempo de compilación, el hecho de analizar una plantilla indica un error grave en el programa. La función de ayuda **template.Must** hace que el manejo de errores sea más conveniente: acepta una plantilla y un error, comprueba que el error es **nil** (y pánicos), y luego devuelve la plantilla. Volveremos a esta idea en la **Sección 5.9**.

Una vez creada la plantilla, aumentada con **daysAgo**, analizada y comprobada, podemos ejecutarla utilizando un **github.IssuesSearchResult** como fuente de datos y **os.Stdout** como el destino:

```
var report = template.Must(template.New("issuelist").
    Funcs(template.FuncMap{"daysAgo": daysAgo}).
    Parse(templ))
func main() {
    result, err := github.SearchIssues(os.Args[1:])
    if err != nil {
        log.Fatal(err)
    }
    if err := report.Execute(os.Stdout, result); err != nil {
        log.Fatal(err)
    }
}
```

El programa imprime un informe de texto sin formato como esto:

```
$ go build gopl.io/ch4/issuesreport
$ ./issuesreport repo:golang/go is:open json decoder
13 issues:
-----
Number: 5680
User: eaigner
Title: encoding/json: set key converter on en/decoder
Age: 750 days
-----
Number: 6050
User: gopherbot
Title: encoding/json: provide tokenizer
Age: 695 days
-----
...
```

Ahora volvamos al paquete **html/template**. Utiliza la misma **API** y expresión del lenguaje que **text/template** pero añade características para escapar automáticamente y adecuadamente al contexto de **strings** que aparecen dentro de **HTML**, **JavaScript**, **CSS** o **URLs**. Estas características pueden ayudar a evitar un problema de seguridad perenne de

la generación de **HTML**, un ataque de inyección, en el que un adversario incluye un valor de **string** como título de un tema para incluir código malicioso, que cuando se escapa indebidamente por una plantilla, da control sobre la página.

La plantilla siguiente imprime la lista de incidencias como una tabla **HTML**. Tener en cuenta la importación diferente:

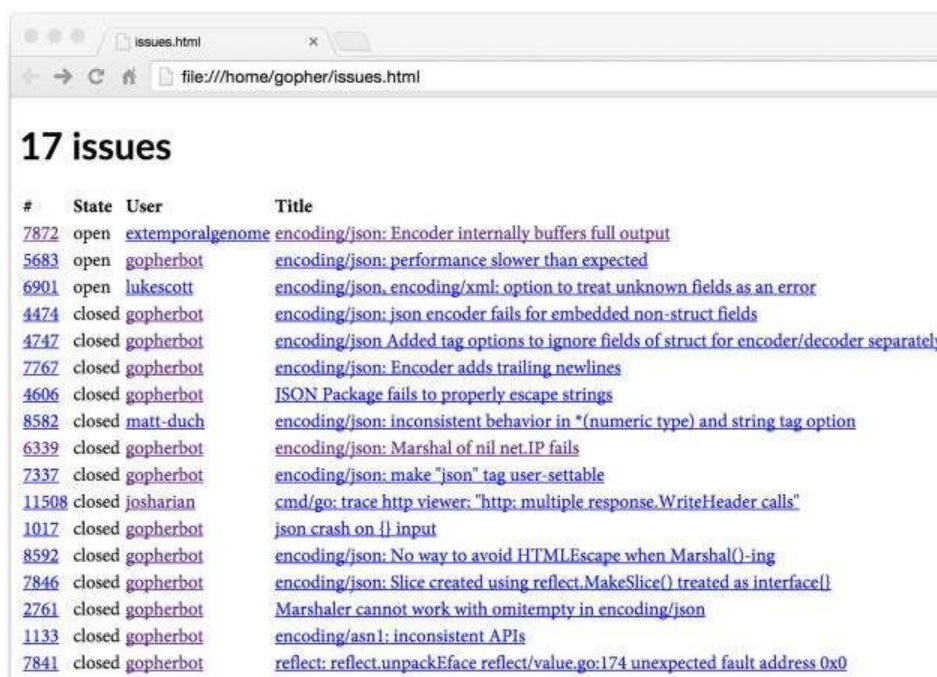
```
gopl.io/ch4/issueshtml
import "html/template"

var issueList = template.Must(template.New("issuelist").Parse(`
<h1>{{.TotalCount}} issues</h1>
<table>
<tr style='text-align: left'>
    <th>#</th>
    <th>State</th>
    <th>User</th>
    <th>Title</th>
</tr>
{{range .Items}}
<tr>
    <td><a href='{{.HTMLURL}}'>{{.Number}}</td>
    <td>{{.State}}</td>
    <td><a href='{{.User.HTMLURL}}'>{{.User.Login}}</a></td>
    <td><a href='{{.HTMLURL}}'>{{.Title}}</a></td>
</tr>
{{end}}
</table>
`))
```

El siguiente comando ejecuta la nueva plantilla con los resultados de una consulta ligeramente diferente:

```
$ go build gopl.io/ch4/issueshtml
$ ./issueshtml repo:golang/go commenter:gopherbot json encoder >issues.html
```

La **Figura 4.4** muestra el aspecto de la tabla en un navegador web. Los enlaces conectan a las páginas web correspondientes en **GitHub**.



#	State	User	Title
7872	open	extemporalgenome	encoding/json: Encoder internally buffers full output
5683	open	gopherbot	encoding/json: performance slower than expected
6901	open	lukescott	encoding/json, encoding/xml: option to treat unknown fields as an error
4474	closed	gopherbot	encoding/json: json encoder fails for embedded non-struct fields
4747	closed	gopherbot	encoding/json Added tag options to ignore fields of struct for encoder/decoder separately
7767	closed	gopherbot	encoding/json: Encoder adds trailing newlines
4606	closed	gopherbot	JSON Package fails to properly escape strings
8582	closed	matt-duch	encoding/json: inconsistent behavior in *(numeric type) and string tag option
6339	closed	gopherbot	encoding/json: Marshal of nil net.IP fails
7337	closed	gopherbot	encoding/json: make "json" tag user-settable
11508	closed	josharian	cmd/go: trace http viewer: "http: multiple response.WriteHeader calls"
1017	closed	gopherbot	json crash on [] input
8592	closed	gopherbot	encoding/json: No way to avoid HTMLEscape when Marshal()-ing
7846	closed	gopherbot	encoding/json: Slice created using reflect.MakeSlice() treated as interface[]
2761	closed	gopherbot	Marshaler cannot work with omitempty in encoding/json
1133	closed	gopherbot	encoding/asn1: inconsistent APIs
7841	closed	gopherbot	reflect: reflect.UnpackEface reflect/value.go:174 unexpected fault address 0x0

Figura 4.4. Una tabla **HTML** de las incidencias de proyectos **Go** relativos a codificación **JSON**.

Ninguna de las incidencias en la **Figura 4.4** representan un desafío para el **HTML**, pero podemos ver el efecto más claramente con incidencias cuyos títulos contienen meta-caracteres **HTML** como **&** y **<**. Hemos seleccionado dos de estos aspectos para este ejemplo:

```
$ ./issueshtml repo:golang/go 3133 10535 >issues2.html
```

La **Figura 4.5** muestra el resultado de esta consulta. Observar que el paquete **html/template** de forma automática escapó en **HTML** de los títulos de modo que aparecen literalmente. Si hubiéramos usado el paquete **text/template**, por error, el **string** de cuatro caracteres **"<"** se habría reproducido como menos que el carácter **'<'** y el **string** **"<link>"** se habría convertido en un elemento **link**, cambiando la estructura del documento **HTML** y quizás comprometiendo su seguridad.

Podemos suprimir este comportamiento de auto-escapar de los campos que contienen datos **HTML** de confianza mediante el uso del tipo de cadena denominada **template.HTML** en lugar de **string**. Existen tipos con nombre similares de confianza en **JavaScript**, **CSS** y **URLs**. El programa siguiente muestra el principio mediante el uso de dos campos con el mismo valor, pero diferentes tipos: **A** es un **string** y **B** es un **template.html**.



Figura 4.5. Meta-caracteres **HTML** en títulos de incidencias que se muestran correctamente.

gopl.io/ch4/autoescape

```
func main() {
    const templ = `

A: {{.A}}



B: {{.B}}

`
    t := template.Must(template.New("escape").Parse(templ))
    var data struct {
        A string // untrusted plain text
        B template.HTML // trusted HTML
    }
    data.A = "<b>Hello!</b>"
    data.B = "<b>Hello!</b>"
    if err := t.Execute(os.Stdout, data); err != nil {
        log.Fatal(err)
    }
}
```

La **Figura 4.6** muestra la plantilla de salida tal como aparece en un navegador. Podemos ver que **A** estaba sujeto a escapar, pero **B** no.

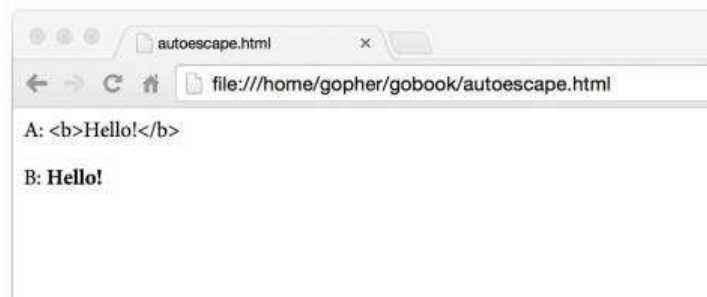


Figura 4.6. Los valores de **string** son de escape **HTML**, pero los valores **template.HTML** no.

Aquí tenemos espacio para mostrar sólo las características más básicas del sistema de plantillas. Como siempre, para más información, consultar la documentación del paquete:

```
$ go doc text/template
$ go doc html/template
```