Curso de Go

Roberto Costumero Moreno
ACM Capítulo de Estudiantes
Facultad de Informática, UPM

Abril 2010

Curso de Go

El logotipo de Go y la información correspondiente al presente manual ha sido obtenida, recopilada y modificada adecuadamente de la página oficial de Go, *http://golang.org*.

El logotipo de Go y la información referente al lenguaje están licenciadas por Google™ bajo una **Licencia Reconocimiento 3.0 de Creative Commons** y puede consultarse en la siguiente dirección: *http://creativecommons.org/licenses/by/3.0/es/*. El uso del logotipo y de la información están autorizados por la presente licencia mencionando su correcta atribución.

ACM Capítulo de Estudiantes
Facultad de Informática - Universidad Politécnica de Madrid
Campus de Montegancedo s/n
28660 Boadilla del Monte
MADRID (ESPAÑA)

Gracias a todos mis amigos y compañeros de ACM
por todo el apoyo y la ayuda que han prestado
para que este manual pudiera ser publicado.

Índice general

Capítulo 1

Introducción

El presente manual se ha realizado para el Curso de Go ofrecido por ACM Capítulo de Estudiantes de la Facultad de Informática de Madrid, en el año 2010.

Se pretende iniciar y dar toda la base teórica al lector para el aprendizaje del nuevo lenguaje de programación Go, desarrollado por Google ™.

1.1. ¿Qué es Go?

Go es un lenguaje de programación de sistemas que, según la página oficial, es expresivo, concurrente y tiene recolector de basura. Además, presume de ser un lenguaje simple, rápido, seguro, divertido y *open source*.

Go sigue una sintaxis tipo C, con lo que si se ha programado anteriormente en dicho lenguaje, la curva de aprendizaje de este nuevo lenguaje de programación es mucho más suave, aunque la diferencia entre ambos es bastante notable desde el primer momento.

Las principales características de Go son:

- Es un lenguaje compilado muy, muy rápido.
- Usa una codificación UTF-8 para todos los ficheros fuente, es decir, permite usar caracteres latinos, chinos, etc.
- Usa tipado fuerte y memoria virtual segura.
- Posee punteros, pero no aritmética de los mismos.
- Es un lenguaje 100 % concurrente.
- Posee hasta un servidor web empotrado.
- Es Open Source, con lo que cualquier persona puede colaborar en su desarrollo aportando ideas o implementando nuevas librerías.

1.2. ¿Quién lo desarrolla?

Go es un proyecto promovido por cinco personas: Rob Pike, Robert Griesemer y Ken Thompson, en primera instancia, a los que se unieron posteriormente Russ Cox e Ian Lance Taylor. Todos los anteriormente citados, forman parte de Google™. Varios de ellos desarrollaron el Sistema Operativo *Plan 9* y han retomado muchas de las ideas originales para la creación de este nuevo lenguaje de programación.

1.3. ¿Por qué crear un nuevo lenguaje?

Existen varias razones por las que crear un nuevo lenguaje de programación.

Para empezar, el mundo informático ha avanzado enormemente en la última década, a la par que no han aparecido nuevos lenguajes de programación para sistemas. Actualmente, nos podemos encontrar las siguientes casuísticas:

- Los ordenadores son mucho más rápidos, pero no así el desarrollo de software.
- Los sistemas software tienen una gran dependencia, por lo que a la hora de compilar es importante realizar un análisis eficiente de las dependencias entre los distintos ficheros, algo que no ocurre en los actuales “ficheros de cabecera” de C.
- Existe una tendencia creciente al uso de lenguajes de tipado dinámico, como Python y Javascript.
- La recolección de basura o la computación paralela, no están soportadas adecuadamente por los lenguajes de sistemas más populares.
- El aumento del número de núcleos en los ordenadores, ha provocado confusión y quebraderos de cabeza respecto a la programación concurrente y paralela.

1.4. Recursos

Pese a que Go es un lenguaje de reciente creación, ya existen numerosos sitios de información sobre el lenguaje, aunque no siempre son fáciles de encontrar debido a que el término ”Go” es muy común en inglés.

Algunos ejemplos:

1.4.1. golang.org

El sitio web oficial del lenguaje, no olvidéis su dirección: *http://golang.org*. Toda la información que queráis encontrar respecto al lenguaje, seguro que estará ahí.

Como curiosidad, comentar que la propia página web está hecha en Go, utilizando el servidor web empotrado y una serie de templates HTML que trae ”de serie”. A continuación, un enlace rápido a los sitios más importantes dentro de la página:

http://golang.org/doc Acceso al listado de los ficheros de documentación.

http://golang.org/cmd Acceso a la documentación sobre los comandos que pueden usarse.

http://golang.org/pkg Acceso a la documentación de todos los paquetes existentes en Go.

http://golang.org/src Acceso al código fuente de distintos ficheros de apoyo.

1.4.2. #go-nuts

Existe un canal de chat oficial en IRC para discutir acerca del lenguaje. Si quieres una sesión *live* sobre Go, entra en el canal **#go-nuts** en el servidor *irc.freenode.net*.

1.4.3. Lista de correo Go Nuts

La lista oficial de correo de Go, se puede encontrar en la siguiente dirección: *http://groups.google.com/group/golang-nuts?pli=1*. Podrás darte de alta, ya que es una lista abierta, y recibir y enviar correos con todas las dudas que te surjan.

1.4.4. Gestor de errores

Hay una página dedicada a gestionar todos los posibles errores del lenguaje, para ser así resueltos de forma eficiente. Si queréis echarle un vistazo, la podéis encontrar en: *http://code.google.com/p/go/issues/list*

1.4.5. http://go-lang.cat-v.org/

Un sitio web mantenido a partir de todas las aportaciones de la gente a través de la lista de correo oficial. Contiene muchas librerías actualmente en desarrollo, ports del lenguaje a otros entornos (entre ellos Windows), y sobre todo, archivos de coloreado de código fuente para un montón de programas de edición de texto.

1.5. ¿Cómo instalarlo?

Para instalar todas las librerías de Go y las herramientas propias del lenguaje, hay que seguir unos sencillos pasos. Los aquí mencionados son los que aparecen en la página oficial y están pensados para cualquier arquitectura basada en UNIX.

1. Inicializar la variable $GOROOT que indica el directorio raíz de Go. Típicamente es el directorio $HOME/go, aunque puede usarse cualquier otro.

2. Inicializar las variables $GOOS y $GOARCH. Indican la combinación de Sistema Operativo y Arquitectura utilizada. Los posibles valores son los que se observan en la tabla superior.

3. Inicializar la variable $GOBIN (opcional), que indica dónde serán instalados los binarios ejecutables. Por defecto es $HOME/bin. Tras la instalación, es conveniente agregarla al $PATH, para poder usar las herramientas desde cualquier directorio.

$GOOS	$GOARCH	SO y Arquitectura
darwin	386	Mac OS X 10.5 o 10.6 - 32-bit x86
darwin	amd64	Mac OS X 10.5 o 10.6 - 64-bit x86
freebsd	386	FreeBSD - 32-bit x86
freebsd	amd64	FreeBSD - 64-bit x86
linux	386	Linux - 32-bit x86
linux	amd64	Linux - 64-bit x86
linux	arm	Linux - 32-bit arm
nacl	386	Native Client - 32-bit x86

Cuadro 1.1: Tabla de valores de $GOOS y $GOARCH

Nota.- Hay que tener en cuenta que las variables $GOOS y $GOARCH indican el sistema contra el que se va a programar, no el sistema sobre el que se está programando, que aunque típicamente sea el mismo, no tiene por qué serlo. Es decir, estaremos todo el rato realizando una compilación cruzada.

Todo lo anterior se resume editando el fichero .bashrc, o cualquiera equivalente, añadiendo al final del fichero las siguientes líneas, con la configuración correcta:

```
export GOROOT=$HOME/go
export GOARCH=amd64
export GOOS=linux
export GOBIN=$HOME/bin
```

Para bajarse las herramientas del repositorio, hay que tener instalado mercurial (tener un comando *hg*). Si no se tiene instalado mercurial, se puede instalar con el siguiente comando:

```
$ sudo easy_install mercurial
```

Si no se consigue instalar, conviene visitar la página oficial de descarga de mercurial.[1]

Tras asegurarnos de que la variable $GOROOT apunta a un directorio que no existe o que se encuentre vacío, hacemos un *checkout* del repositorio con el comando:

```
$ hg clone -r release https://go.googlecode.com/hg/ $GOROOT
```

Una vez que nos hemos bajado todos los ficheros necesarios, hay que instalar Go. Para instalar Go hay que compilarlo, y para ello necesitaremos *GCC* y las librerías estándar de C, ya que el núcleo de Go está escrito en C, el generador de parsers *Bison*, la herramienta *make* y el editor de texto *ed*. En OS X, se puede instalar todo junto al paquete Xcode, mientras que en Linux, por lo general, bastará con el comando (o uno similar para tu distribución):

```
$ sudo apt-get install bison gcc libc6-dev ed make
```

[1] http://mercurial.selenic.com/wiki/Download

Para compilar la distribución de Go, hay que ejecutar:

```
$ cd $GOROOT/src
$ ./all.bash
```

Si *make all* funciona correctamente, finalizará imprimiendo como últimas líneas

```
--- cd ../test
N known bugs; 0 unexpected bugs
```

donde N es un número que varía de una distribución a otra de Go.

1.6. Compilando

Go viene con un conjunto de herramientas bastante completo. Entre las herramientas más importantes, están los dos tipos de compiladores que podemos usar para generar nuestros programas ejecutables: *gccgo y 6g/8g*.

Vamos a ver características de uno y otro, y un ejemplo de cómo realizar una compilación con el compilador nativo de Go, la versión 6g/8g. Una característica común de ambos, es que generan un código únicamente entre un 10 % y un 20 % más lento que código en C.

1.6.1. gccgo

El compilador *gccgo* es un *front-end* del famoso compilador de C, GCC. Posee las siguientes características:

- Es un compilador más tradicional.
- Soporta 32-bit y 64-bit bajo x86, además de ARM.
- Genera muy buen código, pero no tan rápido como su hermano *6g/8g*.
- Se puede enlazar con GCC, y así realizar una compilación con C.
- No soporta pilas segmentadas todavía y aloja cada goroutine - se verá más adelante - por hilo de ejecución.

Su uso, es exactamente igual al uso que se le da a GCC, sólo que invocándolo con el comando gccgo.

1.6.2. 6g/8g

El compilador *6g/8g* es un compilador experimental nativo de Go. *6g*, es el compilador asociado a la arquitectura amd64, y genera ficheros objeto con extensión ".6". *8g*, es el compilador asociado a la arquitectura 386, y genera ficheros objeto con extensión ".8".

- Es un compilador experimental.
- Soporta 32-bit y 64-bit bajo x86, además de ARM.
- Genera buen código de forma muy, muy rápida.
- No se puede enlazar con GCC, pero tiene soporte FFI.
- Posee un buen soporte de gouroutines, multiplexándolas en varios hilos de ejecución, e implementa las pilas segmentadas.

Para compilar un archivo cualquiera llamado **file.go**, usaremos el comando:

```
$ 6g file.go
```

y para enlazar el archivo y así generar el fichero ejecutable correspondiente, usaremos:

```
$ 6l file.6
```

Finalmente, para ejecutar el programa usaremos el comando:

```
$ ./6.out
```

Nota.- Hay que tener en cuenta, que si se usa la versión 32-bit del compilador, se cambiaría cada 6 por un 8.

Nota.- Para conseguir compilar un fichero con su propio nombre (y así no usar el fichero ejecutable por defecto (6.out), podemos pasarle el parámetro *-o fichero_salida* (ejemplo: 6l -o fichero file.6).

El *linker* de Go (*6l*), no necesita recibir ningún otro fichero del que dependa la compilación, como en otros lenguajes, ya que el compilador averigua qué ficheros son necesarios leyendo el comienzo del fichero compilado.

A la hora de compilar un fichero **A.go** que dependa de otro **B.go** que depende a su vez de **C.go**:

- Compila **C.go**, **B.go** y finalmente **A.go**.
- Para compilar **A.go**, el compilador lee **B.go**, no **C.go**

1.7. Otras herramientas

La distribución de Go viene con una serie de herramientas bastante útiles, aunque todavía le faltan otras importantes, como un depurador, que está en desarrollo. Así pues, contamos con: Godoc, Gofmt y con gdb[2].

[2] Aquellos que usen *gccgo* pueden invocar *gdb*, pero la tabla de símbolos será como la de C y no tendrán conocimiento del *run-time*

1.7.1. Godoc

Godoc es un servidor de documentación, análogo a **javadoc**, pero más fácil para el programador que éste último.

Como comentamos al principio del manual, la documentación oficial de Go, que se puede ver en su página oficial[3] se apoya precisamente en esta herramienta.

Es una herramienta muy útil para generar la documentación de nuestros programas y poder compartirlo con la gente de forma rápida y sencilla, a través de la web.

La documentación de las librerías estándar de Go, se basa en los comentarios del código fuente. Para ver la documentación de una librería, podemos verla online a través de la web *http://golang.org/pkg* o bien a través de la linea de comandos, ejecutando:

```
godoc fmt
godoc fmt Printf
```

Y así cambiando *fmt* por el nombre del paquete que queramos ver, y *Printf* por la función en cuestión a ser visualizada.

1.7.2. Gofmt

Gofmt es un formateador de código. Es una herramienta muy útil para ver correctamente el código. Todo el código que se puede observar en la página web oficial, está formateado con esta herramienta.

1.8. Manteniéndose al día

Go se actualiza periódicamente. Para mantenerse al día y conseguir que tu distribución funcione correctamente, debes actualizar cada vez que salga una nueva distribución, que se anuncia en la lista de correo oficial de Go. Para actualizar a la última distribución disponible, hay que poner los siguientes comandos:

```
$ cd $GOROOT/src
$ hg pull
$ hg update release
$ ./all.bash
```

[3] http://golang.org

Capítulo 2

Características y tipos básicos

2.1. Nuestro primer programa: Hello World!

Sin más preámbulos, y después de contar un poco qué es Go y por qué se da este manual, veamos el primer trozo de código: el típico programa Hello World!

```
package main

import "fmt"

func main() {
    fmt.Printf("Hello World!\n")
}
```

2.2. Garbage Collector

Go posee un Garbage Collector - Recolector de Basura - que identifica cuándo se deja de utilizar una variable o una declaración concreta, y libera la memoria asociada de forma automática.

Actualmente, el compilador de Go posee un Garbage Collector muy simple pero efectivo, basado en un ”marcado de barrido”, es decir, marca aquello que puede ser eliminado, y cuando se activa, se borra.

Está en desarrollo un Garbage Collector mucho más avanzado basado en las ideas del Garbage Collector de IBM™[1]. Esta nueva implementación pretende ser muy eficiente, concurrente y de baja latencia, con lo que nada más detectar que algo sobra, se elimine.

2.3. Las bases del lenguaje

Go está basado en una sintaxis tipo C, con lo que cualquier conocimiento previo de dicho lenguaje, será de mucha utilidad para seguir el curso.

[1] http://www.research.ibm.com/people/d/dfb/papers.html

Los ficheros fuente están codificados en UTF-8, lo que implica la posibilidad de usar cualquier tipo de caracteres tanto latinos, como árabes o chinos en un mismo fichero fuente. Los delimitadores de las declaraciones - o lo que es lo mismo, espacios en blanco - son tres: espacios, tabulaciones y saltos de línea.

Los identificadores de cualquier variable en un programa escrito en Go, deben de ser alfanuméricos, incluyendo el caracter '_', teniendo en cuenta que las letras y números deben ser aquellas definidas por Unicode.

2.4. Comentarios

Los comentarios son exactamente los mismos que en C++. Para obtener un comentario de una línea, utilizamos la combinación de caracteres //, mientras que para un comentario de varias líneas, se usa la combinación /* */.

```
// Esto es un comentario de una linea

/* Esto es un
   comentario que ocupa
   varias líneas */
```

Hay que recordar, que los comentarios de varias líneas, no admiten anidamiento, así pues, el siguiente comentario sería erróneo:

```
/* Esto es un
   comentario que ocupa
   varias líneas incorrecto.
   /*
      Con otro comentario anidado
      de varias líneas, que no es posible hacer.
   */
*/
```

2.5. Literales

Entendemos por *literales* todas aquellas expresiones que representan un valor concreto, ya sea en forma de cadena de caracteres o en forma numérica.

Existen tres tipos de literales en Go:

- **Números tipo C:** Son aquellos números literales que se escriben igual que en C. La única diferencia es que en Go, un número literal no requiere signo ni ninguna marca sobre su tamaño (short, long...).

```
98
0x0FF
2.643e5
```

- **Strings tipo C:** Son aquellas cadenas de caracteres que se escriben igual que en C, pero en este caso deben estar representadas en UTF-8 (o cualquier otra representación Unicode). Tambien pueden representarse bytes con *xNN* con 2 dígitos o con *012* con 3 dígitos.

```
"Hello, world!\n"
"\\xFF"         // 1 byte
"\\u00FF"       // 1 caracter unicode, 2 bytes en UTF-8
```

- **Strings puros:** Son cadenas de caracteres que se imprimen tal cual son escritas en el código fuente, sin escapar ningún carácter. Se representan poniendo la cadena entre dos acentos graves ‘ ` ’.

```
`\n\.abc\t\`  ==  "\\n\\.abc\\t\\"
```

2.6. Vistazo rápido de la sintaxis

La sintaxis de Go es muy similar a la sintaxis utilizada en C. A la hora de declarar una variable o un tipo, se realiza de la misma forma que en C, únicamente que se invierte el orden del nombre de la variable y el tipo, quedando este último al final. Además, hay que añadir una palabra reservada al principio de las declaraciones. Veamos esto con un ejemplo, definiendo tres tipos de variables y un tipo *Struct*.

```
var a int          // a es un entero
var b, c *int      // b y c son punteros a enteros
var d []int        // d es un array de enteros

type S struct { a, b int }
// S es una estructura con dos atributos enteros, a y b.
```

Las estructuras de control del programa, también nos resultarán familiares si hemos trabajado con lenguages tipo C. Veamos un ejemplo con un *if* y un *for*.

```
if a == b {
    fmt.Print("a y b son iguales")
} else {
    fmt.Print("a y b son distintos")
}

for i = 0; i < 10; i++ { ... }
```

Nota.- No son necesarios los paréntesis en la comparación del *if* o del *for*, pero siempre son necesarias las llaves, que no pueden ser omitidas. Además, hay que tener en cuenta que la llave de apertura de un *if* debe ir en la misma linea que la sentencia, y que el *else* tiene que ir emparejado en la misma linea que el cierre de bloque del *if*. Más adelante se verá la estructura correcta.

Por último hay que notar en el código la ausencia del carácter ';' como indicativo del final de sentencia.

Cuando Go fue creado, los puntos y coma eran obligatorios en todas las instrucciones. Finalmente, se acordó que su uso sería opcional, y que sería el propio compilador el que añadiría los puntos y coma al final de las sentencias que considerara válidas. Aún así, el uso de puntos y coma en ciertas estructuras es obligatorio. Este esquema sigue las pautas acerca de este tipo de delimitadores del lenguaje *BCPL*, precursor de *B* y por lo tanto de C.

Así pues, resumiendo acerca del uso de los puntos y coma:

- Son opcionales en todos los programas al final de una sentencia, aunque deberían **no** ponerse. Únicamente son obligatorios a la hora de separar los elementos en la cláusula de un bucle *for* o elementos en un *if*.
- El compilador introducirá automáticamente los puntos y coma, por ello no es conveniente ponerlos, al final de una línea no vacía, si lo último que se encuentra es:
  - Un identificador o un literal.
  - Alguna de las palabras reservadas: break, continue, fallthrough o return.
  - Alguno de los siguientes tokens: ++, –,),], }.
- Se pueden poner los puntos y coma para separar instrucciones en una misma línea, pudiendo ser omitido antes de ')' o de '}'.

2.7. Tipos

Go posee únicamente tres tipos básicos: Números, Booleanos y Cadenas de caracteres.

2.7.1. Números

Existen tres tipos numéricos: Enteros, enteros sin signo y números flotantes.

Cada uno de estos tipos tiene asociadas una serie de variantes dependiendo del número de bits en el que sea almacenado. Veamos un cuadro resumen con los tipos numéricos existentes:

Enteros	Enteros sin signo	Flotantes
int	uint	float
int8	uint8 = byte	
int16	uint16	
int32	uint32	float32
int64	uint64	float64

Cuadro 2.1: Tabla de tipos numéricos

También existe el tipo *uintptr*, que sirve para almacenar número enteros lo suficientemente grandes como para necesitar un puntero.

Como puede deducirse de la tabla anterior, el número que acompaña a cada nombre de tipo, es el número de bits que ocupa en memoria. Así, podemos observar que los números flotantes no tienen representaciones válidas con 8 y 16 bits.

Los tipos que no tienen asociado ningún número, *int*, *uint* y *float*, se representan con un número de bits igual al ancho de la palabra de la máquina en la que se ha compilado el código. De esta forma se puede llegar a pensar que en un ordenador de 32-bit, los tipos *int* y *int32* son equivalentes, pero **no** es así. **Todos los tipos de la tabla son distintos**.

Debido a que Go es un lenguaje con tipado fuerte[2], no existe conversión implícita de tipos, aunque posteriormente veremos cómo pueden realizarse conversiones explícitas.

2.7.2. Bool

El tipo *bool* define el tipo booleano usual, con dos constantes predefinidas que son: *true* y *false*.

Hay que tener en cuenta, que a diferencia de otros lenguajes, en Go los *punteros* y los *enteros* **no son booleanos**.

2.7.3. String

El tipo *string* representa arrays invariables de bytes, o lo que es lo mismo, texto. Los strings están delimitados por su longitud, no por un carácter nulo como suele ocurrir en la mayoría de lenguajes. Esto hace que el tipo *string* sea mucho más seguro y eficiente.

Toda cadena de caracteres representada por el lenguaje, incluidas las cadenas de caracteres literales, tienen como tipo *string*.

Como se ha dicho en el primer párrafo, y al igual que ocurre con los números enteros, los strings son invariables. Esto significa, que se pueden reasignar variables de tipo *string* para que contengan otros valores, pero los valores de una variable de este tipo **no se pueden modificar**.

De la misma forma que 5 siempre es 5, ”Hola” siempre es ”Hola”.

Pese a que todo esto puede parecer incoherente o muy engorroso, las librerías de Go poseen un gran soporte para la manipulación de las cadenas de caracteres.

[2]**Tipado fuerte**: Un lenguaje tiene tipado fuerte cuando todas las variables tienen que declararse con un tipo asociado, y no puede cambiar su tipo en tiempo de ejecución.

2.8. Operadores

Los operadores en Go, como no podía ser de otra forma, son básicamente los mismos que existen en C. Existen operadores binarios y unarios. Veamos cuáles son los operadores binarios con una tabla, ordenada de mayor a menor precedencia.

Precedencia	Operadores	Comentarios
6	* / % << >> & &^	&^ significa "bit clear"
5	+ − | ^	^significa "xor"
4	== ! = < <= > >=	
3	<−	Usado para comunicación
2	&&	
1	||	

Cuadro 2.2: Tabla de operadores binarios

Los operadores unarios, tienen todos la misma precedencia y son: & ! * + − ^ <−.

El operador unario ^ significa "complemento".

2.8.1. Diferencias con C

Los programadores de C que comiencen a programar en Go, se encontrarán varios cambios en lo referente a los operadores, y que son de bastante importancia.

- Go posee menos niveles de precedencia.
- ^ sustituye a ∼, que pasa de ser un "or exclusivo" binario, a ser unario.
- ++ y −− no son operadores de expresiones.
  x++ es una instrucción, no una expresión;
  *p++ es (*p)++, no *(p++).
- &^ es un nuevo operador. Útil en expresiones constantes.
- <<, >>, etc. necesitan un contador de rotación sin signo.

2.8.2. Ejemplos

En el siguiente código hay varios ejemplos de operaciones válidas.

```
+x
25 + 6*x[i]
y <= f()
^a >> b
f() || g()
x == y + 2 && <- chan_ptr > 0
```

```
x &* 7     // x con los 3 bits inferiores a 0.
fmt.Printf("\%5.2g\n", 2*math.Sin(PI/8))

"Hola" + "," + "adios"
"Hola, " + str
```

2.9. Conversiones

Como ya se ha mencionado anteriormente, las conversiones entre distintos tipos deben realizarse de manera explícita, por lo que cualquier intento de conversión implícita fallará en tiempo de compilación.

Así pues, y tal y como se mencionó al ver los tipos de datos numéricos, convertir valores de un tipo a otro es una conversión explícita, que se realiza como si fuera la llamada a una función. Veamos algún ejemplo:

```
uint8(int_var)    // truncar al tamaño de 8 bits
int(float_var)    // truncar a la fracción
float64(int_var)  // conversión a flotante
```

También es posible realizar conversiones a *string*, de manera que podemos realizar las siguientes operaciones, entre otras:

```
string(0x1234)           // == "\u1234"
string(array_de_bytes)   // bytes -> bytes
string(array_de_ints)    // ints -> Unicode/UTF-8
```

2.10. Valores constantes y números ideales

Las constantes numéricas son ”números ideales”: aquellos números que no tienen tamaño ni signo, y por lo tanto no tienen modificadores *l*, *u*, o *ul*.

```
077            // Octal
0xFE8DACEFF    // Hexadecimal
1 << 100
```

Existen tanto números ideales enteros como de coma flotante. La sintaxis del literal utilizado, determinará su tipo.

```
1.234e5       // float
1e2           // float
630           // int
```

Los números constantes en coma flotante y los enteros pueden ser combinados según nos convenga. El tipo del resultado de las operaciones, así como las propias operaciones, dependerá del tipo de las constantes utilizadas.

Así pues tendremos:

```
2*3.14   // float: 6.28
3./2     // float 1.5
3/2      // int: 1

// Alta precisión:
const Ln2 = 0,693147180559945309417232121458\
          176568075500134360255254120680009
const Log2E = 1/Ln2    // mayor precisión
```

Nota.- Nótese la diferencia existente entre la imposibilidad de tener conversiones entre tipos implícita, que impide el lenguaje, y la conversión realizada en las operaciones, que sí que está definida. Esto es, a una variable de tipo *int*, no se le puede asignar el valor de una variable de tipo *float*, sin una conversión de tipos previa, pero sí una constante de tipo float.

La existencia de los números ideales tiene una serie de consecuencias, como la vista en la nota anterior. El lenguaje define y permite el uso de constantes sin una conversión explícita si el valor puede ser representado.

```
var million int = 1e6   // Variable int, constante float
math.Sin(1)             // Pasamos un int, espera un float
```

Las constantes deben ser representables en el tipo al que queremos asignarlas. Por ejemplo: ^0 es -1, que no es representable en el rango $0-255$.

```
uint8(^0)      // Mal: -1 no es representable
^uint8(0)      // OK
uint8(350)     // Mal: 350 no es representable
uint8(35.0)    // OK: 35.0 = 35
uint8(3.5)     // Mal: 3.5 no es representable
```

2.11. Declaraciones

Como ya hemos comentado anteriormente, todas las declaraciones en Go van precedidas de una palabra clave (var, const, type, func) y comparadas con C, parecen estar inversas, ya que primero va el nombre de la variable, y posteriormente va el tipo de la variable. Veamos algunos ejemplos:

```
var i int
const PI = 22./7.
type Point struct { x, y int }
func sum (a, b int) int { return a + b }
```

Seguro que te preguntarás... ¿Por qué está invertido el tipo y la variable? Es una cuestión de diseño, que otorga muchas facilidades para el compilador, pero que además permite definir en una misma instrucción varios punteros.

Por ejemplo:

```
var p, q *int
```

Además, las funciones son más cómodas de leer y son más consistentes con otras declaraciones. Más adelante iremos viendo más razones para la inversión del tipo en la declaración.

2.11.1. Variables

Las variables en Go se declaran precedidas de la palabra reservada *var*.

Pueden tener un tipo, una expresión que inicialice la variable, o ambas cosas. Las expresiones que inicialicen las variables deben encajar con las variables y su tipo, ya que recordemos que Go es un lenguaje de tipado fuerte.

Además, hay que tener en cuenta que podemos declarar varias variables de un mismo o distintos tipos en la misma línea, separando los nombres de las variables por comas.

Veamos algunos ejemplos:

```
var i int
var j = 365.245
var k int = 0
var l, k uint64 = 1, 2
var inter, floater, stringer = 1, 2.0, "Hola"
```

Dado que puede ser bastante tedioso estar escribiendo todo el rato la palabra reservada *var*, Go posee una característica muy interesante, que permite definir varias variables agrupadas por paréntesis, y separándolas por puntos y coma. He aquí uno de los pocos usos obligatorios de dicho delimitador.

```
var (
    i int;
    j = 365.245;
    k int = 0;
    l, k uint64 = 1, 2;
    inter, floater, stringer = 1, 2.0, "Hola"
)
```

Nota.- La última declaración de variables no tiene un ';' dado que no es obligatorio (ni aconsejable) ponerlo en dicha línea, ya que va antes de un carácter ')', tal y como se comentó en el apartado **2.6**.

Este mismo esquema, no sólo sirve para definir variables, sino que también puede ser usado para definir constantes (*const*) y tipos (*type*).

En las funciones, las declaraciones de la forma:

```
var v = valor
```

pueden ser acortadas a:

```
v := valor
```

Y teniendo las mismas características y tipado fuerte existente. He aquí, otro de los motivos para la inversión del nombre de la variable y el tipo de la misma.

El tipo es aquél que tenga el valor, que para números ideales sería *int* o *float* dependiendo del número utilizado. Este tipo de declaraciones de variables es muy usado y está disponible en muchos sitios, como en las cláusulas de los bucles *for*.

2.11.2. Constantes

Las declaraciones de constantes están precedidas por la palabra reservada *const*.

Todas las constantes son inicializadas en tiempo de compilación, lo que significa que todas las declaraciones de constantes deben ser inicializadas con una expresión constante. Además, de forma opcional, pueden tener un tipo especificado de forma explícita.

```
const Pi = 22./7.
const AccuratePi float64 = 355./113
const pepe, dos, comida = "Pepe", 2, "carne"

const (
   Lunes, Martes, Miercoles = 1, 2, 3;
   Jueves, Viernes, Sabado = 4, 5, 6
)
```

Iota

Las declaraciones de constantes pueden usar un contador especial, denominado *iota*, que comienza en 0 en cada bloque declarado con *const* y que se incrementa cada vez que encuenta un caracter punto y coma.

```
const (
    Lunes = iota;       // 0
    Martes = iota;      // 1
    Miercoles = iota    // 2
)
```

Existe una versión mucho más rápida para el uso del contador especial *iota*, que se basa en que la declaración de una variable pueda seguir un mismo patrón de tipo y expresión.

Sería algo así como lo que se puede observar en el siguiente ejemplo:

```
const (
    loc0, bit0 uint32 = iota, 1<<iota;  // 0, 1
    loc1, bit1;                         // 1, 2
    loc2, bit2                          // 2, 4
)
```

2.11.3. Tipos

El programador es libre de definir nuevos tipos a lo largo de su programa. Para declarar un nuevo tipo, se debe poner la palabra reservada *type*.

Más adelante, dedicaremos todo el capítulo 5 a hablar sobre los distintos tipos de datos compuestos que el usuario puede crear, por lo que un simple ejemplo nos dará una idea de cómo se realiza la declaración de tipos.

```
type Punto struct {
    x, y, z float
    nombre string
}
```

2.11.4. Operador new vs. función make

En Go existe un operador *new()* que básicamente reserva memoria para una variable.. La sintaxis, es igual que la llamada a una función, con el tipo deseado como argumento, de igual forma que en C++.

El operador *new()* devuelve un puntero al objeto que se ha creado en memoria. Hay que notar, que a diferencia de otros lenguajes, Go no posee un *delete* o un *free* asociado, ya que para eso está el Garbage Collector.

```
var p *Punto = new(Punto)
v := new(int)              // v tiene como tipo *int
```

Existe a su vez, un función llamada *make* que se usará generalmente en tres tipos de datos concretos: maps, slices y canales. Normalmente en estos tipos queremos que los cambios hechos en una variable no afecten al resto, y por lo tanto nos aseguramos que al crear este tipo de datos, se creen nuevas variables y no punteros a las mismas.

Así pues para crear variables de tipos de datos complejos usaremos *make* para los tipos anteriormente mencionados, que nos devolverá una estructura creada, o *new* si usamos otros tipos de datos, ya que nos devolverá un puntero a dicha estructura.

2.11.5. Asignaciones

Las asignaciones son fáciles y se realizan al igual que en la mayoría de lenguajes modernos. Para ello, se utiliza el operador '='. Hay que recordar, que el operador de igualdad en Go, es '=='.

Respecto a las asignaciones hay que tener en cuenta, que Go posee tres tipos de asignaciones o asignaciones con tres tipos de características:

- **Asignaciones simples**: A una variable se le asigna un valor constante o el valor de otra variable.
- **Asignaciones múltiples**: A un grupo de variables se les asignan distintos valores simultáneamente.
- **Asignaciones múltiples de funciones**: Las funciones, como veremos más adelante, pueden devolver valores múltiples, que pueden ser "recogidos" por varias variables.

Veamos un ejemplo de lo anterior:

```
a = 5
b = a

x, y, z = f(), g(), h()
a, b = b, a

nbytes, error := Write(buffer)

// Nótese que nbytes y error estarían siendo declaradas
// en ese instante (operador :=). Funciona igual para '='
```

Capítulo 3

Estructuras de control

A lo largo de este capítulo, vamos a ir viendo las distintas estructuras de control que posee Go. Antes de comenzar, hay que realizar un apunte, concerniente especialmente a las sentencias *if*, *for* y *switch*:

- Son similares a las declaraciones homónimas en C, pero poseen diferencias significativas.
- Como ya se ha apuntado, no utilizan paréntesis y las llaves son obligatorias.
- Las tres sentencias admiten instrucciones de inicialización (operador ':=').

3.1. Sentencia condicional: if

La sentencia condicional *if* permite ejecutar una rama de código u otra dependiendo de la evaluación de la condición.

Esta sentencia, es similar a la existente en otros lenguajes, pero tiene el problema de que el *else* tiene que estar en la misma línea que la llave de cierre del *if*.

Además admite instrucciones de inicialización, separadas por un punto y coma. Esto es muy útil con funciones que devuelvan múltiples valores.

Si una sentencia condicional *if* no tiene ninguna condición, significa *true*, lo que en este contexto no es muy útil pero sí que lo es para *for* o *switch*.

Veámoslo con un ejemplo:

```
if x < 5 { f() }
if x < 5 { f() } else if x == 5 { g() }

if v := f(); v < 10 {
    fmt.Printf("\%d es menor que 10\n", v)
} else {
    fmt.Printf("\%d no es menor que 10", v)
}
```

```
// Se recogen los valores, y se comprueba
// que err sea un valor no nulo

if n, err = fd.Write(buffer); err != nil { ... }

// El else debe estar en la misma línea que
// la llave de cierre del if
if v := f(); v < 10 {
    fmt.Printf("\%d es menor que 10\n", v)
} else {
    fmt.Printf("\%d no es menor que 10", v)
}
```

3.2. Sentencia condicional: switch

El *switch* tiene la misma apariencia que en la mayoría de los lenguajes y, aunque es en apariencia similar al de C, tiene varias diferencias:

- Las expresiones no tienen por qué ser constantes ni siquiera números.
- Las condiciones no se evalúan en cascada automáticamente.
- Para conseguir evaluación en cascada, se puede usar al final de cada *case* la palabra reservada *fallthrough* finalizando la sentencia con un punto y coma.
- Múltiples casos que ejecuten una misma secuencia de instrucciones, pueden ser separados por comas.

```
switch count%7 {
    case 4, 5, 6: error()
    case 3: a *= v; fallthrough
    case 2: a *= v; fallthrough
    case 1: a *= v; fallthrough
    case 0: return a*v
}
```

El *switch* de Go es más potente que el de C. Además de que las expresiones pueden ser de cualquier tipo, sabemos que la ausencia de la misma significa *true*. Así, podemos conseguir una cadena *if−else* con un switch. Veamos otro ejemplo:

```
// Forma habitual
switch a {
    case 0: fmt.Printf("0")
    default: fmt.Printf("No es cero")
}
```

```
// If-else
a, b := x[i], y[j];
switch {
    case a < b: return -1
    case a == b: return 0
    case a > b: return 1
}

// Este último switch sería equivalente a
switch a, b := x[i], y[j]; { ... }
```

3.3. Bucles: for

Un bucle es una sentencia de los lenguajes de programación que permite ejecutar un mismo grupo de instrucciones *n* veces de forma consecutiva.

En Go sólo tenemos un tipo de bucles, a diferencia de otros lenguajes que suelen poseer varios. Todo lo que implique un bucle, en Go se ejecuta con una instrucción *for*. Una de las cosas más llamativas de los bucles en Go, es la posibilidad de usar asignaciones a varias variables, y hacer un bucle doble en una sola instrucción. Veamos algunos ejemplos:

```
// Forma clásica
for i := 0; i < 10; i++ {
    // ejecutamos instrucciones
}

// La falta de condición implica true (bucle infinito)
for ; ; { fmt.Printf("Bucle infinito") }
for { fmt.Printf("Bucle infinito, sin poner ;") }

// Equivalente al bucle while
for a > 5 {
    fmt.Printf("Bucle while\n")
    a--
}

// Bucle doble
for i,j := 0,N; i < j; i,j = i+1,j-1 {
    // ejecutamos instrucciones
}

// Sería equivalente a
for i := 0; i < N/2; i++{
    for j := N; j > i; j--{
        // ejecutamos instrucciones
    }
}
```

3.4. Instrucciones break y continue

Las instrucciones *break* y *continue* nos permiten controlar la ejecución de un bucle o una sentencia condicional, parándolo en un determinado momento, saliendo del mismo y continuando la ejecución del programa, o saltando a la siguiente iteración del bucle, respectivamente.

Funcionan de la misma forma que en C y se puede especificar una etiqueta para que afecte a una estructura externa a donde es llamado. Como se podría pensar al haber una etiqueta, sí, existe la sentencia *goto*.

```
Bucle: for i := 0; i < 10; i++ {
    switch f(i) {
        case 0, 1, 2: break Loop
    }
    g(i)
}
```

3.5. Funciones

Las funciones nos permiten ejecutar un fragmento de código que realiza una acción específica, de forma separada de un programa, que es donde se integra.

La principal ventaja del uso de funciones es la reutilización de código. Si vamos a realizar la misma tarea un gran número de veces, nos evitamos la duplicidad del código, y así la tarea de mantenimiento es mucho más sencilla.

Las funciones en Go comienzan con la palabra reservada *func*. Después, nos encontramos el nombre de la función, y una lista de parámetros entre paréntesis. El valor devuelto por una función, si es que devuelve algún valor, se encuentra después de los parámetros. Finalmente, utiliza la instrucción *return* para devolver un valor del tipo indicado.

Si la función devuelve varios valores, en la declaración de la función los tipos de los valores devueltos van en una lista separada por comas entre paréntesis.

```
func square(f float) float {
    return f*f
}

// Función con múltiples valores devueltos
func MySqrt(f float) (float, bool) {
    if f >= 0 {
        return math.Sqrt(f), true
    }
    return 0, false
}
```

Los parámetros devueltos por una función son, en realidad, variables que pueden ser usadas si se las otorga un nombre en la declaración. Además, están por defecto inicializadas a ”cero” (0, 0.0, false... dependiendo del tipo de la variable).

```
func MySqrt(f float) (v float, ok bool) {
    if f >= 0 {
        v, ok = math.Sqrt(f), true
    } else {
        v, ok = 0, false
    }
    return v, ok
}

// Función con múltiples valores devueltos
func MySqrt(f float) (v float, ok bool) {
    if f >= 0 {
        v, ok = math.Sqrt(f), true
    }
    return v, ok
}
```

Finalmente, un *return* vacío devuelve el actual valor de las variables de retorno. Vamos a ver dos ejemplos más de la función *MySqrt*:

```
func MySqrt(f float) (v float, ok bool) {
    if f >= 0 {
        v, ok = math.Sqrt(f), true
    }
    return           // Debe ser explícito
}

// Función con múltiples valores devueltos
func MySqrt(f float) (v float, ok bool) {
    if f < 0 {
        return       // Caso de error
    }
    return math.Sqrt(f), true
}
```

3.6. ¿Y qué hay del 0?

En esta sección vamos a tratar el tema de inicialización de las variables a un valor estándar “cero”. En Go, por temas de seguridad, toda la memoria utilizada es previamente inicializada. Todas las variables son inicializadas con su valor asignado o con un valor “cero” en el momento de su declaración. En caso de inicializar una variable con el valor “cero”, se usará el valor tomado como “cero” para su tipo concreto.

Como los valores dependerán del tipo, veamos cuáles son esos posibles valores:

- **int:** 0
- **float:** 0.0
- **bool:** false
- **string:** “”
- **punteros:** nil
- **struct:** valores “cero” de todos sus atributos.

Por ejemplo, el siguiente bucle imprimirá 0, 0, 0, 0, 0. Esto se debe a que la variable v se declara en cada pasada del bucle, con lo que se reinicializa.

```
for i := 0; i < 5; i++ {
    var v int
    fmt.Printf("\%d\n", v)
    v = 5
}
```

3.7. Defer

La instrucción *defer* es una instrucción poco común en lenguajes de programación, pero que puede resultar muy útil. En el caso de Go, la instrucción *defer* ejecuta una determinada función o método cuando la función que la engloba termina su ejecución.

La evaluación de los argumentos de *defer* se realiza en el momento de la llamada a dicha instrucción, mientras que la llamada a la función no ocurre hasta que se ejecuta el *return* correspondiente.

Es una instrucción muy útil para cerrar descriptores de ficheros o desbloquear mutex. Veamos un pequeño ejemplo, en el que cerraremos un fichero al terminar la ejecución de la función.

```
func data(name string) string {
    f := os.Open(name, os.O_RDONLY, 0)
    defer f.Close()
    contenido := io.ReadAll(f)
    return contenido
}
```

Cada *defer* se corresponde con una llamada a una única función. Cada vez que se ejecuta un *defer* una llamada a la función que deba ser ejecutada se encola en una pila LIFO, de tal forma que la última llamada será la primera en ejecutarse. Con un bucle, podemos cerrar todos los ficheros o desbloquear los mutex de un golpe al final. Veamos un ejemplo, que en este caso imprimirá los números 4, 3, 2, 1 y 0 en dicho orden.

```
func f(){
    for i := 0; i < 5; i++ {
        defer fmt.Printf("\%d", i)
    }
}
```

Veamos un ejemplo más completo para entender correctamente el funcionamiento de *defer*. Para ello, vamos a seguir la traza de ejecución de nuestro programa, imprimiendo la secuencia de funciones que son llamadas.

```
func trace (s string) { fmt.Print("entrando en: ", s, "\n") }

func untrace (s string) { fmt.Print("saliendo de: ", s, "\n") }

func a() {
    trace("a")
    defer untrace("a")
    fmt.Print("estoy en a\n")
}

func b() {
    trace("b")
    defer untrace("b")
    fmt.Print("estoy en b\n")
    a()
}

func main() {
    b()
}
```

El programa anterior, imprimirá los siguientes valores por pantalla:

```
entrando en: b
estoy en b
entrando en: a
estoy en a
saliendo de: a
saliendo de: b
```

Veamos ahora una versión más optimizada del mismo programa, que imprimirá exactamente los mismos valores por pantalla. En este ejemplo se puede observar que los argumentos de *defer* son evaluados en el momento, pero que la ejecución de la función no se realiza hasta finalizar su función “padre”.

```
func trace (s string) string {
    fmt.Print("entrando en: ", s, "\n")
    return s
}

func un(s string) { fmt.Print("saliendo de: ", s, "\n") }

func a() {
    defer un(trace("a"))
    fmt.Print("estoy en a\n")
}

func b() {
    defer un(trace("b"))
    fmt.Print("estoy en b\n")
    a()
}

func main() {
    b()
}
```

3.8. Funciones anónimas, λ-programación y closures

Los tres conceptos están muy relacionados entre sí. Las funciones anónimas y la λ-programación se refieren al mismo concepto, mientras que las *closures* son un uso concreto de las funciones anónimas.

Funciones anónimas o λ-programación Se basa en utilizar funciones que están definidas en tiempo de ejecución. Este tipo de funciones, no tienen nombres por sí mismas y se ejecutan en el contexto en el que son declaradas. A menudo, son referenciadas por variable para poder ser ejecutadas en otro contexto. Son típicas de lenguajes funcionales como, por ejemplo, Haskell.

Closures Se pueden definir las *closures* como funciones anónimas declaradas en el ámbito de otra función y que pueden hacer referencia a las variables de su función contenedora incluso después de que esta se haya terminado de ejecutar.

Dado que son dos conceptos bastante abstractos, lo mejor es que veamos un ejemplo de cada uno de ellos.

```
func f(){
    for i := 0; i < 10; i++{
        g := func(i int) { fmt.Printf("\%d", i) }
        g(i)
    }
}
```

```
func sumador(){
    var x int
    return func(delta int) int{
        x += delta
        return x
    }
}

var f = sumador()    // f es una función ahora.
fmt.Printf(f(1))
fmt.Printf(f(20))
fmt.Printf(f(300))
```

El anterior ejemplo, imprimirá los valores 1, 21 y 321, ya que irá acumulando los valores en la variable *x* de la función *f*.

Capítulo 4

Estructuración de programas

4.1. Paquetes y estructura de los ficheros fuente

Todos los programas en Go están construidos como paquetes, que pueden usar funciones de otros paquetes. Un programa está por lo tanto creado mediante el enlace de un conjunto de paquetes y cada paquete, se forma de uno o varios ficheros fuente.

La importación de paquetes en Go se realiza de forma "calificada", es decir, que cuando nos referimos al ítem de un paquete, hay que hacer una referencia explícita al nombre del paquete en cuestión. Así pues, tendremos una sintaxis similar a: *nombreDelPaquete.NombreDelItem*.

Todo fichero fuente contiene:

- Una cláusula de instanciación del paquete. Es decir, a qué paquete pertenece dicho fichero. El nombre usado es el nombre utilizado por defecto por los paquetes que lo importen.

```
package fmt
```

- Un conjunto opcional de declaraciones de importación de otros paquetes.

```
import "fmt"          // Usa el nombre por defecto
import my_fmt "fmt"   // Usa en este fichero el nombre my_fmt
```

- Puede existir alguna declaración global o declaración a "nivel de paquete".

```
package main     // Este fichero forma parte del paquete "main"

import "fmt"     // Este fichero utiliza el paquete "fmt"

const hola = "Hello, World!\n"

func main() {
    fmt.Print(hola)  // fmt es el nombre del paquete importado
}
```

4.2. Ámbito de una variable

El ámbito de una variable define la visibilidad de la misma por otras funciones, métodos, o incluso por otros paquetes, y que ésta pueda ser usada o no fuera de su entorno local.

Dentro de un paquete todas las variables globales, funciones, tipos y constantes son visibles desde todos los ficheros que formen dicho paquete.

Para los usuarios de dicho paquete - otros paquetes que lo importen -, los nombres deben comenzar por una letra mayúscula para que sean visibles. Esto se acepta para variables globales, funciones, tipos, constantes, además de métodos y atributos de una estructura, aplicados a variables globales y tipos. Su equivalente en C/C++ serían los modificadores: *extern*, *static*, *private* y *public*.

```
const hola = "Hola"          // Variable visible en el paquete
const Hola = "Ey, Hola!"     // Variable visible globalmente
const _Adios = "Ciao ciao"   // _ no es una letra mayúscula
```

4.3. Inicialización

Todos los programas en Go deben contener un paquete llamado *main* y tras la etapa de inicialización, la ejecución del programa comienza por la función *main()* de dicho paquete, de forma análoga a la función global *main()* en C o C++.

La función *main.main()* no acepta argumentos y no devuelve ningún valor. El programa finaliza su ejecución de forma inmediata y satisfactoriamente cuando la función *main.main()* termina.

Como se ha mencionado en el primer párrafo, antes de comenzar la ejecución existe una etapa de inicialización. Esta etapa se encarga de inicializar las variables globales antes de la ejecución de *main.main()*. Existen dos formas de hacer esta inicialización:

1. Una declaración global con un inicializador.
2. Una función *init()*. Pueden existir tantas como una por fichero fuente.

La dependencia entre los paquetes garantiza un orden de ejecución correcto. La inicialización siempre se realiza de forma “mono-hilo”, es decir, no existe ningún tipo de concurrencia en la etapa de inicialización para hacerla más segura. Veamos un ejemplo de inicialización:

```
package trascendental
import "math"
var Pi float64
```

```
func init() {
    Pi = 4 * math.Atan(1)   // init() calcula Pi
}

---------------------

package main
import (
    "fmt";
    "trascendental"
)

// La declaración calcula dosPi
var dosPi = 2 * trascendental.Pi

func main() {
    fmt.Printf("2*Pi = \%g\n", dosPi)
}
```

4.4. Testing

Go provee al programador de una serie de recursos muy interesantes. Entre ellos, cabe destacar el sistema de pruebas que ofrece al programador para generar las pruebas de su código y realizar un test automático.

Para probar un paquete hay que escribir un conjunto de ficheros fuente dentro del mismo paquete, con nombres del tipo **_test.go*.

Dentro de esos ficheros se declararán las funciones globales que le darán soporte a la herramienta de test *gotest*. Las funciones deben seguir el siguiente patrón: Test[^a-z]*, es decir, la palabra "Test"seguida de un número indefinido de letras. Además deben de tener la declaración de la siguiente forma:

```
func TestXxxx (t *testing.T)
```

El paquete *testing* proporciona una serie de utilidades de *logging* y *error reporting*.

Veamos un ejemplo sacado de la librería *fmt* de Go:

```
package fmt
import (
    "testing"
)
```

```
func TestFlagParser(t *testing.T) {
    var flagprinter flagPrinter
    for _, tt := range flagtests {
        s := Sprintf(tt.in, &flagprinter)
        if s != tt.out {
            t.Errorf("Sprintf(\%q, &flagprinter) =>
                        \%q, want \%q", tt.in, s, tt.out)
        }
    }
}
```

Al utilizar la herramienta *gotest* se ejecutan todos los ficheros *_test.go. Al ejecutarlo sobre el fichero anteriormente nombrado *fmt_test.go*, obtenemos la siguiente salida:

```
mallocs per Sprintf(""): 2
mallocs per Sprintf("xxx"): 3
mallocs per Sprintf("\%x"): 5
mallocs per Sprintf("\%x \%x"): 7
PASS
```

4.5. Librerías

Las librerías son paquetes normales pero que por convención se suele llamar así a los paquetes que vienen integrados dentro del propio lenguaje.

Actualmente las librerías existentes son pocas, pero están creciendo a un ritmo vertiginoso. Veamos en una tabla algunos ejemplos:

Paquete	Propósito	Ejemplos
fmt	E/S formateada	Printf, Sprintf...
os	Interfaz del SO	Open, Read, Write...
strconv	números $<->$ string	Atoi, Atof, Itoa...
io	E/S genérica	Copy, Pipe...
flag	flags: –help, etc.	Bool, String...
log	Log de eventos	Log, Logf, Stderr...
regexp	Expresiones regulares	Compile, Match...
template	HTML, etc.	Parse, Execute...
bytes	byte arrays	Compare, Buffer...

Cuadro 4.1: Tabla de paquetes

4.5.1. Fmt

El paquete *fmt* implementa las funciones de entrada / salida formateada, de manera análoga a otras funciones en otros lenguajes como C.

Modificadores

Las funciones que utilizan utilizan un formato para imprimir los datos, usan una serie de modificadores para imprimir los distintos tipos de datos. A continuación hay una lista de los modificadores existentes:

- **General:**
  - %v imprime el valor en un formato por defecto.
    Cuando se imprimen structs, el flag + (%+v) añade los nombres de los campos.
  - %#v una representación en "sintaxis-Go" del valor
  - %T una representación en "sintaxis-Go" del tipo del valor
- **Boolean:**
  - %t la palabra true o false
- **Enteros:**
  - %b base 2
  - %c el caracter representado por su código Unicode
  - %d base 10
  - %o base 8
  - %x base 16, con letras minúsculas de a-f
  - %X base 16, con letras mayúsculas de A-F
- **Reales:**
  - %e notación científica, p.ej. -1234.456e+78
  - %E notación científica, p.ej. -1234.456E+78
  - %f coma flotante sin exponente, e.g. 123.456
  - %g cualquiera que %e o %f produzca pero de manera más compacta
  - %G cualquiera que %E o %f produzca pero de manera más compacta
- **Cadenas de caracteres o slices de bytes:**
  - %s los bytes sin interpretar de un string o slice
  - %q una cadena de caracteres con dobles comillas escapada de forma segura con una sintaxis Go
  - %x notación base 16 con dos caracteres por byte
- **Punteros:**
  - %p notación base 16, con un 0x precedente
- No hay flag 'u'. Los enteros se imprimen sin signo si son del tipo unsigned.

Fprintf

```
func Fprintf(w io.Writer, format string, a ...interface{})
            (n int, error os.Error)
```

La función *Fprintf* escribe en un elemento de tipo *io.Writer* (es decir, un descriptor de fichero o similares), con el formato indicado en el parámetro *format*. El resto de parámetros son las variables en cuestión que deben imprimirse.

Fprintf devuelve 2 valores:

- **n**: Indica el número de caracteres escritos correctamente.
- **error**: Contiene un valor significativo si ha habido algún error.

Veamos un ejemplo de cómo se usaría la función:

```
var x int = 2
var punt *int

fmt.Fprintf (os.Stdout, "La variable x que vale \%d,
             está en la posición \%p", x, &punt)
```

Printf

```
func Printf(format string, a ...interface{})
           (n int, errno os.Error)
```

La función *Printf* escribe por la salida estándar con el formato indicado en el parámetro *format*. El resto de parámetros son las variables en cuestión que deben imprimirse. Realmente, *Printf* realiza una llamada a *Fprintf* con el primer parámetro igual a *os.Stdout*.

Printf devuelve 2 valores:

- **n**: Indica el número de caracteres escritos correctamente.
- **errno**: Contiene un valor significativo si ha habido algún error.

Veamos un ejemplo de cómo se usaría la función:

```
var x int = 2
var punt *int

fmt.Printf ("La variable x que vale \%d,
             está en la posición \%p", x, &punt)
```

Sprintf

```
func Sprintf(format string, a ...interface{}) string
```

La función *Sprintf* escribe en un buffer intermedio la cadena de caracteres pasada con el formato indicado en el parámetro *format*. El resto de parámetros son las variables en cuestión que deben imprimirse.

Sprintf devuelve la cadena de caracteres que hemos formateado.

Veamos un ejemplo de cómo se usaría la función:

```
var x int = 2
var punt *int
var s string

s = fmt.Sprintf ("La variable x que vale \%d,
                  está en la posición \%p", x, &punt)
```

Fprint, Print y Sprint

```
func Fprint(w io.Writer, a ...interface{}) (n int, error os.Error)
func Print(a ...interface{}) (n int, errno os.Error)
func Sprint(a ...interface{}) string
```

Las funciones *Fprint*, *Print* y *Sprint* son generalizaciones de las funciones anteriormente descritas. Como se puede observar, cambian el nombre eliminando la 'f' final y el parámetro que se corresponde con el formato. Así pues, estas funciones imprimen en el formato por defecto que el lenguaje considere.

Veamos un ejemplo de cómo se usarían las funciones:

```
fmt.Fprint (os.Stderr, "Error: 12")
fmt.Print ("ACM da muchos cursos geniales\n")
s = fmt.Sprint ("Go mola un montón")
```

La salida del programa sería:

```
Error: 12ACM da muchos cursos geniales
Go mola un montón
```

Fprintln, Println y Sprintln

```
func Fprintln(w io.Writer, a ...interface{}) (n int, error os.Error)
func Println(a ...interface{}) (n int, errno os.Error)
func Sprintln(a ...interface{}) string
```

Las funciones *Fprintln*, *Println* y *Sprintln* son generalizaciones de las funciones con formato anteriormente descritas. Como se puede observar, estas funciones terminan en 'ln' lo que indica que al final del último operando que se le pase a la función, añaden un salto de línea automáticamente.

Veamos un ejemplo de cómo se usarían las funciones:

```
fmt.Fprintln (os.Stderr, "Error: 12")
fmt.Println ("ACM da muchos cursos geniales")
s = fmt.Sprintln ("Go mola un montón")
```

La salida del programa sería:

```
Error: 12
ACM da muchos cursos geniales
Go mola un montón
```

4.5.2. Os

El paquete *os* se compone de varios ficheros que tratan toda la comunicación con el Sistema Operativo. Vamos a ver algunas de las funciones más utilizadas de dicho paquete y que están repartidas en diversos ficheros.

ForkExec

```
func ForkExec(argv0 string, argv []string, envv []string,
                  dir string, fd []*File) (pid int, err Error)
```

La función *ForkExec* crea un nuevo hilo de ejecución dentro del proceso actual e invoca al comando Exec (ejecuta otro programa) con el programa indicado en *argv0*, con los argumentos *argv[]* y el entorno descrito en *envv*.

El array *fd* especifica los descriptores de fichero que se usarán en el nuevo proceso: fd[0] será el descriptor 0 (equivalente a os.Stdin o entrada estándar), fd[1] será el descriptor 1 y así sucesivamente. Pasándole un valor de *nil* hará que el proceso hijo no tenga los descriptores abiertos con esos índices.

Si *dir* no está vacío, el proceso creado se introduce en dicho directorio antes de ejecutar el programa.

ForkExec devuelve 2 valores:

- **pid**: Indica el id de proceso (pid) del proceso hijo.
- **err**: Contiene un valor significativo si ha habido algún error.

Veamos un ejemplo de cómo se usaría la función:

```
var argv []string { "/bin/ls", "-l" }
var env []string

pid, ok := os.ForkExec ("/bin/ls", argv, env, "/home", null)
```

Exec

```
func Exec(argv0 string, argv []string, envv []string) Error
```

La función *Exec* sustituye el proceso actual pasando a ejecutar otro programa mediante el comando Exec, que ejecuta otro programa indicado en *argv0*, con los argumentos *argv[]* y el entorno descrito en *envv*.

Si el comando se ejecuta correctamente, nunca retorna al programa principal. En caso de fallo devuelve un error.

ForkExec es, casi siempre, una mejor opción para ejecutar un programa, ya que tras su ejecución volvemos a tener control sobre el código ejecutado.

Exec devuelve un error en caso de que el comando nose ejecute correctamente.

Veamos un ejemplo de cómo se usaría la función:

```
var argv []string { "/bin/ls", "-l" }
var env []string

error := os.Exec ("/bin/ls", argv, env)
```

Wait

```
func Wait(pid int, options int) (w *Waitmsg, err Error)
```

La función *Wait* espera a que termine o se detenga la ejecución de un proceso con identificador *pid* y retorna un mensaje describiendo su estado y un error, si es que lo hay. El parámetro *options* describe las opciones referentes a la llamada *wait*.

Wait devuelve 2 valores:

- **w**: Un mensaje con el estado del proceso.
- **err**: Contiene un valor significativo si ha habido algún error.

Veamos un ejemplo de cómo se usaría la función:

```
var argv []string { "/bin/ls", "-l" }
var env []string

if pid, ok := os.ForkExec ("/bin/ls", argv, env, "/home", nil); ok {
    msg, ok := os.Wait(pid, 0)
}
```

Getpid

```
func Getpid() int
```

La función *Getpid* devuelve el identificador del proceso que ejecuta la función.

Getppid

```
func Getppid() int
```

La función *Getppid* devuelve el identificador del proceso padre del que ejecuta la función.

Exit

```
func Exit(code int)
```

La función *Exit* termina la ejecución del programa con un código de error indicado en el parámetro *code*.

4.5.3. Os - Entrada/salida en ficheros

La escritura en ficheros depende del paquete *os*. A pesar de ello, se ha creado una nueva sección en el manual para tratar las funciones de tratamiento de ficheros para una mejor comprensión de las mismas.

El tipo File

Los ficheros en un Sistema Operativo son identificados a través de identificadores de ficheros, que indican con un número cada fichero abierto en el sistema y accesible por cada proceso. El tipo *File* representa un descriptor de fichero abierto que trata como si fuera un objeto, y está definido dentro del paquete *os* como:

```
// File represents an open file descriptor.
type File struct {
    fd      int
    name    string
    dirinfo *dirInfo // nil a menos que sea un directorio
    nepipe  int
}
```

Los parámetros importantes son los dos primeros:

- **fd**: Indica el descriptor de fichero del archivo.
- **name**: Indica el nombre relativo del fichero (hello.go y no /home/yo/hello.go)

Echemos ahora un vistazo a las funciones que nos permiten manejar ficheros.

Fd

```
func (file *File) Fd() int
```

La función *Fd* devuelve el descriptor de fichero del archivo *file* que llama a la función.

Nota.- Nótese que Fd no recibe ningún parámetro, sino que es un objeto quien invocará a dicho método. Para una mejor comprensión se recomienda repasar este punto tras la lectura del capítulo 6.

Name

```
func (file *File) Name() string
```

La función *Name* devuelve el nombre relativo del fichero *file* que llama a la función.

Open

Las función *Open* que vamos a ver a continuación utiliza una serie de modificadores para abrir los ficheros en distintos modos. Los modificadores existentes son:

```
O_RDONLY  // Abre el fichero en modo sólo lectura
O_WRONLY  // Abre el fichero en modo sólo escritura
O_RDWR  // Abre el fichero en modo lectura-escritura
O_APPEND  // Abre el fichero en modo append (escribir al final)
O_ASYNC  // Genera una señal cuando la E/S está disponible
O_CREAT  // Crea un nuevo fichero si no existe
O_EXCL  // Usado con O_CREAT. El fichero debe no existir
O_SYNC  // Abre el fichero listo para E/S síncrona
O_TRUNC  // Si es posible trunca el fichero al abrirlo
O_CREATE  // Igual a O_CREAT
```

La función *Open* tiene el siguiente formato:

```
func Open(name string, flag int, perm int)
          (file *File, err Error)
```

La función abre el fichero indicado (con su ruta absoluta o relativa) en el parámetro *name*, del modo que se especifica en el parámetro *flag* usando los flags anteriormente descritos, con los permisos indicados en el último parámetro.

Open devuelve 2 valores:

- **file**: Un puntero a un objeto de tipo *File, que usaremos después para llamar a los distintos métodos.
- **err**: Un error en caso de que la apertura no haya funcionado correctamente.

Veamos un ejemplo de cómo se usaría la función:

```
var fichero *os.File
var error os.Error

fichero, error = os.Open("/home/yo/mifichero",
                         O_CREAT | O_APPEND, 660)
```

Close

```
func (file *File) Close() Error
```

La función *Close* cierra el fichero que invoca dicho método.

Close devuelve un error en caso de que la llamada no haya podido completarse correctamente.

Veamos un ejemplo de cómo se usaría la función:

```
var fichero *os.File
var error os.Error

fichero, error = os.Open("/home/yo/mifichero",
                         O_CREAT | O_APPEND, 660)
fichero.Close()
```

Read

```
func (file *File) Read(b []byte) (n int, err Error)
```

La función *Read* lee caracteres hasta el número máximo de bytes que haya en el parámetro *b*, que se consigue con la llamada *len(b)*.

Read devuelve 2 valores:

- **n**: El número de bytes leídos realmente.
- **err**: Contiene un valor significativo si ha habido algún error.

Veamos un ejemplo de cómo se usaría la función:

```
var fichero *os.File
var error os.Error
var bytes [256]byte
var num_leidos int

fichero, error = os.Open("/home/yo/mifichero",
                         O_CREAT | O_RDONLY, 660)
num_leidos, error = fichero.Read (byte)
fichero.Close()
```

Write

```
func (file *File) Write(b []byte) (n int, err Error)
```

La función *Write* escribe caracteres hasta el número de bytes que haya en el parámetro *b*.

Write devuelve 2 valores:

- **n**: El número de bytes escritos realmente.
- **err**: Contiene un valor significativo si ha habido algún error.

Veamos un ejemplo de cómo se usaría la función:

```
var fichero *os.File
var error os.Error
var bytes [256]byte = { 'H', 'o', 'l', 'a' }
var num_leidos int

fichero, error = os.Open("/home/yo/mifichero",
                         O_CREAT | O_APPEND, 660)
num_leidos, error = fichero.Write (byte)
fichero.Close()
```

WriteString

```
func (file *File) WriteString(s string) (ret int, err Error)
```

La función *WriteString* escribe una cadena de caracteres (string). Es mucho más cómodo usar esta función para escribir datos en un fichero.

WriteString devuelve 2 valores:

- **ret**: El número de caracteres escritos realmente.
- **err**: Contiene un valor significativo si ha habido algún error.

Veamos un ejemplo de cómo se usaría la función:

```
var fichero *os.File
var error os.Error
var cadena String = "Hola"
var num_leidos int

fichero, error = os.Open("/home/yo/mifichero",
                         O_CREAT | O_APPEND, 660)
num_leidos, error = fichero.WriteString (cadena)
fichero.Close()
```

4.5.4. Otras librerías

Go es un lenguaje en desarrollo, y como tal sus librerías están todavía creciendo aunque lo hacen a pasos agigantados. Una de las mejores características de *Go* es que tiene unas librerías muy bien documentadas, tanto en el código como en la documentación online de la página web oficial.

A continuación se mencionan algunas de las librerías más comúnmente usadas, que el lector puede entender perfectamente al finalizar este curso.

- **bufio**: Es una librería con funciones encargadas de realizar E/S a través de un buffer.
- **crypto**: Una interesante librería con múltiples funciones criptográficas.
- **flag**: Esta librería se encarga de manejar todos los parámetros pasados al programa mediante la línea de comandos.
- **http**: Una librería que contiene un servidor http y todas las funciones necesarias para implementar un programa para la web.
- **image**: Contiene funciones para el manejo de imágenes, bitmaps...
- **math**: Librería de funciones matemáticas de todo tipo.
- **reflect**: La librería encargada de gestionar funciones con número variable de parámetros. (Se verá en detalle más adelante).
- **strings**: Una serie de funciones para manejo de cadenas de caracteres.
- **testing**: La librería más completa para realizar pruebas sobre tu propio programa.

Capítulo 5

Tipos de datos compuestos

Existen una serie de tipos de datos denominados compuestos, ya se son tipos de datos más complejos que los vistos anteriormente. Básicamente veremos Arrays, Slices y Structs, entre otras cosas.

5.1. Arrays

Los arrays son una estructura de datos que permiten tener una serie de datos del mismo tipo distribuidos uniformemente en un bloque de memoria.

Los arrays de Go son más cercanos a los de Pascal que a los de C. Más adelante, veremos los slices que son más parecidos a los arrays de C.

La declaración de los arrays se realiza con la palabra reservada *var* acompañada del nombre de la variable, y el tamaño y tipo de datos que tendrá el array. Todos los arrays deben tener un tamaño explícito.

```
var ar [3]int
```

La anterior declaración declarará un array de nombre *ar* con capacidad para 3 números enteros. Ya que no han sido inicializados, por defecto serán 0.

Si en algún momento deseamos averiguar el tamaño de un array, podemos hacer uso de la función *len()*.

```
len(ar) == 3
```

Los arrays son valores, no punteros implícitos como ocurre en C. De todas formas, se puede obtener la dirección de memoria donde se ha almacenado un array, que podría servir para pasar un array de forma eficiente a una función. Veamos un ejemplo:

```
func f(a [3]int) {
    fmt.Println(a)
}
```

```
func fp(a *[3]int) {
    fmt.Println(a)
}

func main() {
    var ar [3] int
    f(ar)    // Pasa una copia de ar
    fp(&ar)  // Pasa un puntero a ar
}
```

La salida del ejemplo anterior sería la siguiente:

```
 [0 0 0]
 &[0 0 0]
```

La función Println conoce la estructura de un array y cuando detecta uno, lo imprime de forma óptima.

Los arrays también tienen su propio literal, es decir, su forma de representar el valor real de un array. Veamos un pequeño ejemplo:

```
// Array de 3 enteros
[3]int { 1, 2, 3 }

// Array de 10 enteros, los 3 primeros no nulos
[10]int { 1, 2, 3 }

// Si no queremos contar el número de elementos
// '...' lo hace por nosotros.
[...]int { 1, 2, 3 }

// Si no se quieren inicializar todos,
// se puede usar el patrón 'clave:valor'
[10]int { 2:1, 3:1, 5:1, 7:1 }
```

Siguiendo las normas sobre los arrays que sabemos hasta ahora, podemos conseguir la dirección de memoria de un array literal para tener un puntero a una nueva instancia recien creada:

```
func fp(a *[3]int) {
    fmt.Println(a)
}

func main() {
    for i := 0; i < 3; i++ {
        fp(&[3]int {i, i*i, i*i*i})
    }
}
```

El resultado de la ejecución de dicho programa, nos imprimirá por pantalla:

```
&[0 0 0]
&[1 1 1]
&[2 4 8]
```

5.2. Slices

Un *slice* es una **referencia** a una sección de un array. Los slices se usan más comúnmente que los arrays.

Un *slice* se declara como un array nada más que éste no tiene tamaño asociado:

```
var a []int
```

Si queremos obtener el número de elementos que posee un *slice*, recurrimos a la función *len()*, de igual forma que si fuera un array.

Un *slice* puede crearse "troceando" un array u otro *slice*.

```
a = ar[7:9]
```

Esa instrucción nos generaría un slice a partir de un array, tomando los índices 7 y 8 del array. El número de elementos devuelto por *len(a) == 2*, y los índices válidos para acceder al slice *a*, son *0* y *1*.

De igual forma, podemos inicializar un *slice* asignándole un puntero a un array:

```
a = &ar     // Igual que a = ar[0:len(ar)]
```

Según un reciente cambio en la sintaxis referente a los slices, no es necesario poner implícitamente los dos valores, de comienzo y fin, sino que poniendo únicamente el valor de comienzo, nos creará un slice hasta el final del array o slice que estemos referenciando.

```
a = ar[0:]  // Igual que a = ar[0:len(ar)]
b = ar[5:]  // Igual que b = ar[5:len(ar)]
```

Al igual que los arrays, los slices tienen sus literales correspondientes, que son iguales pero no tienen tamaño asociado.

```
var slice = []int {1, 2, 3, 4, 5}
```

Lo que esta instrucción hace, es crear un array de longitud 5, y posteriormente crea un *slice* que referencia el array.

Podemos también reservar memoria para un slice (y su array correspondiente) con la función predefinida *make()*:

```
var s100 = make([]int, 100)  // slice: 100 enteros
```

¿Por qué usamos *make()* y no *new()*? La razón es que necesitamos construir un slice, no sólo reservar la memoria necesaria. Hay que tener en cuenta que *make([]int)* devuelve *[]int*, mientras que *new([]int)* devuelve **[]int)*.

La función *make()* se utiliza para slices, maps (que veremos en un momento) y para canales de comunicación.

Un *slice* se refiere a un array asociado, con lo que puede haber elementos más allá del final de un slice que estén presentes en el array. La función *cap()* (capacity) nos indica el número de elementos que el slice puede crecer. Veamos un ejemplo:

```
var ar = [10]int {0,1,2,3,4,5,6,7,8,9}
var a = &ar[5:7]    // Referencia al subarray {5, 6}

// len(a) == 2 y cap(a) == 5. Se puede aumentar el slice:

a = a[0:4]  // Referencia al subarray {5,6,7,8}

// Ahora: len(a) == 4. cap(a) == 5 .
```

¿Cómo es posible que cap(a) == 5? El aumento del slice puede ser de hasta 5 elementos. Teniendo en cuenta que si aumentamos el slice con *a[0:5]*, conseguiríamos un subarray con los valores del 5 al 9.

Los slices pueden ser utilizados como arrays crecientes. Esto se consigue reservando memoria para un slice con la función *make()* pasándole dos números - longitud y capacidad - y aumentándolo a medida que crezca:

```
var sl = make([]int, 0, 100)  // Len == 0, cap == 100

func appendToSlice(i int, sl []int) []int {
    if len(sl) == cap(sl) { error(...) }
    n := len(sl)
    sl = sl[0:n+1]  // Aumentamos el tamaño 1 unidad
    sl[n] = i
    return sl
}
```

La longitud de *sl* siempre será el número de elementos y crecerá según se necesite. Este estilo es mucho más “barato” e idiomático en Go.

Para terminar con los slices, hablaremos sobre lo “baratos” que son. Los slices son una estructura muy ligera que permiten al programador generarlos y aumentarlos o reducirlos según su necesidad. Además, son fáciles de pasar de unas funciones a otras, dado que no necesitan una reserva de memoria extra.

Hay que recordar que un *slice* ya es una referencia de por sí, y que por lo tanto, el almacenamiento en memoria asociado puede ser modificado. Por poner un ejemplo, las funciones de E/S utilizan slices, no contadores de elementos. Así pues, la función Read:

```
func Read(fd int, b []byte) int{
   var buffer[100]byte
   for i := 0; i < 100; i++ {
       // Rellenamos el buffer con un byte cada vez
       Read(fd, buffer[i:i+1])  // Aquí no hay reserva de memoria.
   }
}

// Igualmente, podemos partir un buffer:
header, data := buf[0:n], buf[n:len(buf)]
header, data := buf[0:n], buf[n:]  // Equivalente
```

De la misma forma que un buffer puede ser troceado, se puede realizar la misma acción con un string con una eficiencia similar.

5.3. Maps

Los *maps* son otro tipo de referencias a otros tipos. Los *maps* nos permiten tener un conjunto de elementos ordenados por el par "clave:valor", de tal forma que podamos acceder a un valor concreto dada su clave, o hacer una asociación rápida entre dos tipos de datos distintos. Veamos cómo se declararía un *map* con una clave de tipo *string* y valores de tipo *float*:

```
var m map[string] float
```

Este tipo de datos es análogo al tipo **map<string,float>* de C++ (Nótese el *). En un *map*, la función *len()* devuelve el número de claves que posee.

De la misma forma que los *slices* una variable de tipo *map* no se refiere a nada. Por lo tanto, hay que poner algo en su interior para que pueda ser usado. Tenemos tres métodos posibles:

1. Literal: Lista de pares "clave:valor" separados por comas.

   ```
   m = map[string] float { "1":1, "pi:3.1415 }
   ```

2. Creación con la función *make()*

   ```
   m = make(map[string] float)  // recordad: make, no new.
   ```

3. Asignación de otro map

   ```
   var m1 map[string] float
   m1 = m  // m1 y m ahora referencian el mismo map.
   ```

5.3.1. Indexando un map

Imaginemos que tenemos el siguiente map declarado:

```
m = map[string] float { "1":1, "pi":3.1415 }
```

Podemos acceder a un elemento concreto de un map con la siguiente instrucción, que en caso de estar el valor en el map, nos lo devolverá, y sino provocará un error.

```
uno   := m["1"]
error := m["no presente"]  //error
```

Podemos de la misma forma, poner un valor a un elemento. En caso de que nos equivoquemos y pongamos un valor a un elemento que ya tenía un valor previo, el valor de dicha clave se actualiza.

```
m["2"] = 2
m["2"] = 3   // m[2] vale 3
```

5.3.2. Comprobando la existencia de una clave

Evidentemente, visto el anterior punto, no es lógico andarse preocupando que la ejecución de un programa falle al acceder a una clave inexistente de un *map*. Para ello, existen métodos que permiten comprobar si una clave está presente en un *map* usando las asignaciones multivalor que ya conocemos, con la llamada fórmula "coma ok". Se denomina así, porque en una variable se coge el valor, mientras que se crea otra llamada *ok* en la que se comprueba la existencia de la clave.

```
m = map[string] float { "1":1, "pi":3.1415 }

var value float
var present bool

value, present = m[x]

// o lo que es lo mismo
v, ok := m[x]  // He aquí la fórmula "coma ok"
```

Si la clave se encuentra presente en el *map*, pone la variable booleana a *true* y el valor de la entrada de dicha clave en la otra variable. Si no está, pone la variable booleana a *false* y el valor lo inicializa al valor "cero"de su tipo.

5.3.3. Borrando una entrada

Borrar una entrada de un *map* se puede realizar mediante una asignación multivalor al map, de igual forma que la comprobación de la existencia de una clave.

```
m = map[string] float { "1":1, "pi":3.1415 }

var value float
var present bool
var x string = f()

m [x] = value, present
```

Si la variable *present* es *true*, asigna el valor *v* al *map*. Si *present* es *false*, elimina la entrada para la clave *x*. Así pues, para borrar una entrada:

```
m[x] = 0, false
```

5.3.4. Bucle for y range

El bucle *for* tiene una sintaxis especial para iterar sobre arrays, slices, maps, y alguna estructura que veremos más adelante. Así pues, podemos iterar de la siguiente forma:

```
m := map[string] float { "1":1.0, "pi":3.1415 }
for key, value := range m {
    fmt.Printf("clave \%s, valor \%g\n", key, value)
}
```

Si sólo se pone una única variable en el *range*, se consigue únicamente la lista de claves:

```
m := map[string] float { "1":1.0, "pi":3.1415 }
for key := range m {
    fmt.Printf("key \%s\n", key, value)
}
```

Las variables pueden ser directamente asignadas, o declaradas como en los ejemplos anteriores. Para arrays y slices, lo que se consigue son los índices y los valores correspondientes.

5.4. Structs

Las *structs* son un tipo de datos que contienen una serie de atributos y permiten crear tipos de datos más complejos. Su sintaxis es muy común, y se pueden declarar de dos formas:

```
var p struct { x, y float }

// O de forma más usual
type Punto struct { x, y float }
var p Point
```

Como casi todos los tipos en Go, las *structs* son valores, y por lo tanto, para lograr crear una referencia o puntero a un valor de tipo *struct*, usamos el operador *new(StructType)*. En el caso de punteros a estructuras en Go, no existe la notación $->$, sino que Go ya provee la indirección al programador.

```
type Point struct { x, y float }
var p Point
p.x = 7
p.y = 23.4
var pp *Point = new(Point)
*pp = p
pp.x = Pi    // equivalente a (*pp).x
```

Ya que los *structs* son valores, se puede construir un *struct* a "cero" simplemente declarándolo. También se puede realizar reservando su memoria con *new()*.

```
var p Point         // Valor a "cero"
pp := new(Point);   // Reserva de memoria idiomática
```

Al igual que todos los tipos en Go, los *structs* también tienen sus literales correspondientes:

```
p = Point { 7.2, 8.4 }
p = Point { y:8.4, x:7.2 }
pp := &Point { 23.4, -1 } // Forma correcta idiomáticamente
```

De la misma forma que ocurría con los arrays, tomando la dirección de un *struct* literal, da la dirección de un nuevo valor creado. Este último ejemplo es un constructor para *Point.

5.4.1. Exportación de tipos

Los campos (y métodos, que veremos posteriormente) de un *struct* deben empezar con una letra Mayúscula para que sea visible fuera del paquete. Así pues, podemos tener las siguientes construcciones de *struct*:

- Tipo y atributos privados:

```
type point struct { x, y float }
```

- Tipo y atributos exportados:

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

- Tipo exportado con una mezcla de atributos:

```
type point struct {
    X, Y float      // exportados
    nombre string   // privado
}
```

- Podríamos incluso tener un tipo privado con campos exportados.

5.4.2. Atributos anónimos

Dentro de un *struct* se pueden declarar atributos, como otro *struct*, sin darle ningún nombre al campo. Este tipo de atributos son llamados atributos anónimos y actúan de igual forma que si la *struct* interna estuviera insertada o "embebida.[en] la externa.

Este mecanismo permite una manera de derivar algunas o todas tus implementaciones desde cualquier otro tipo o tipos. Veamos un ejemplo:

```
type A struct {
    ax, ay int
}

type B struct {
    A
    bx, by float
}
```

B actúa como si tuviera cuatro campos: *ax*, *ay*, *bx* y *by*. Es casi como si *B* hubiera sido declarada con cuatro atributos, dos de tipo int, y otros dos de tipo float. De todas formas, los literales de tipo *B*, deben ser correctamente declarados:

```
b := B { A { 1, 2 }, 3.0, 4.0 }
fmt.Println( b.ax, b.ay, b.bx, b.by )
```

Pero todo el tema de los atributos anónimos es mucho más útil que una simple interpolación de los atributos: *B* también tiene el atributo *A*. Un atributo anónimo se parece a un atributo cuyo nombre es su tipo:

```
b := B { A { 1, 2 }, 3.0, 4.0 }
fmt.Println( b.A )
```

El código anterior imprimiría 1 2. Si *A* es importado de otro paquete, el atributo aún así debe seguir llamándose A:

```
import "pkg"
type C struct { pkg.A }
...
c := C { pkg.A{ 1, 2 } }
fmt.Println(c.A)  // No c.pkg.A
```

Para terminar, cualquier tipo nombrado, o puntero al mismo, puede ser utilizado como un atributo anónimo y puede aparecer en cualquier punto del *struct*.

```
type C struct {
    x float
    int
    string
}

c := C { 3.5, 7, "Hola" }
fmt.Println(c.x, c.int, c.string)
```

5.4.3. Conflictos

Pueden darse una serie de conflictos dada la posible derivación de unos *structs* en otros. De hecho, la mayoría de conflictos vendrán dados por tener dos atributos distintos pero con el mismo nombre. Para ello, se aplican las siguientes reglas en caso de que esto suceda:

1. Un atributo "externo" oculta un atributo "interno". Esto da la posibilidad de sobreescribir atributos y métodos.
2. Si el mismo nombre aparece dos veces en el mismo nivel, se provoca un error si el nombre es usado por el programa. Si no es usado, no hay ningún problema. No existen reglas para resolver dicha ambigüedad, con lo que debe ser arreglado.

Veamos algún ejemplo:

```
type A struct { a int }
type B struct { a, b int }

type C struct { A; B }
var c C
```

Usando *c.a* es un error. ¿Lo sería *c.A.a* o *c.B.a*? Evidentemente no, ya que son perfectamente accesibles.

```
type D struct { B; b float }
var d D
```

El uso de *d.b* es correcto: estamos accediendo al float, no a *d.B.b*. La *b* interna se puede conseguir accediendo a través de *d.B.b*.

Capítulo 6

Orientación a Objetos

Pese a la ausencia de objetos según el sistema tradicional de orientación a objetos, Go es un lenguaje perfectamente orientado a objetos y lo hace de una forma más lógica según la definición formal de los objetos.

6.1. Métodos

Go no posee clases, pero su ausencia no impide que se puedan crear métodos específicos para un tipo de datos concreto. Es posible crear métodos para (casi) cualquier tipo de datos.

6.1.1. Métodos para structs

Los métodos en Go se declaran de forma independiente de la declaración del tipo. Éstos se declaran como funciones con un receptor explícito. Siguiendo con el ejemplo del tipo *Punto*:

```
type Punto struct { x, y float }

// Un método sobre *Punto
func (p *Punto) Abs() float {
    return math.Sqrt(p.x*p.x + p.y*p.y)
}
```

Cabe notar que el receptor es una variable explícita del tipo deseado (no existe puntero a *this*, sino que hay una referencia explícita al tipo **Punto*.

Un método no requiere un puntero a un tipo como receptor, sino que podemos usar un tipo pasado como valor. Esto es más costoso, ya que siempre que se invoque al método el objeto del tipo será pasado por valor, pero aún así es igualmente válido en Go.

```
type Punto3 struct { x, y float }

// Un método sobre Punto3
func (p Punto3) Abs() float {
    return math.Sqrt(p.x*p.x + p.y*p.y + p.z*p.z)
}
```

6.1.2. Invocación de métodos

Los métodos se invocan de una manera muy simple, tal y como se podría esperar en cualquier otro lenguaje orientado a objetos.

```
p := &Point { 3, 4 }
fmt.Print(p.Abs())    // Imprimirá 5
```

Ahora veamos un ejemplo de uso de un ejemplo con un tipo de datos que no sea de tipo *struct*.

```
type IntVector []int

func (v IntVector) Sum() (s int) {
    for i, x := range v {
        s += x
    }
    return
}

fmt.Println(IntVector { 1, 2, 3 }. Sum())
```

6.1.3. Reglas para el uso de métodos

Existen una serie de reglas concernientes a los métodos. Como se ha mencionado, los métodos están relacionados con un tipo concreto, digamos *Foo*, y están enlazados a su tipo de manera estática.

El tipo de un receptor en un método puede ser tanto **Foo* como *Foo*. Se pueden tener simultáneamente varios métodos *Foo* y otros cuantos métodos **Foo*.

Foo Por sí solo no puede ser de tipo puntero, aunque los métodos puedan obtener un receptor de tipo **Foo*.

Por último, hay que tener en cuenta que un tipo *Foo* debe estar definido en el mismo paquete que todos sus métodos.

6.1.4. Punteros y valores en los métodos

Go automáticamente indirecciona o derreferencia los valores cuando se invoca un método. Por ejemplo, aunque un método concreto tenga como receptor el tipo **Punto*, se puede invocar al método con un valor direccionable de tipo *Point*. Entenderemos esto mejor con un ejemplo:

```
p1 := Punto { 3, 4 }
fmt.Print(p1.Abs())   // Azúcar sintáctico para (&p1).Abs()
```

De forma similar, si los métodos están descritos para un tipo *Punto3*, se puede usar un valor de tipo **Punto3*:

```
p3 := &Punto3 { 3, 4, 5 }
fmt.Print(p3.Abs())   // Azúcar sintáctico para (*p3).Abs()
```

6.1.5. Atributos anónimos en los métodos

Naturalmente, cuando un atributo anónimo está embebido en un tipo *struct*, los métodos de ese tipo son también embebidos, es decir, se tiene una herencia de los métodos.

Este método ofrece una manera simple de emular algunos de los efectos de las subclases y la herencia utilizado por los lenguajes de programación orientados a objetos más comunes. Veamos un ejemplo de atributos anónimos:

```
type Punto struct { x, y float }
func (p *Punto) Abs() float { ... }

type PuntoNombre struct{
    Point
    nombre string
}

n := &PuntoNombre { Punto { 3, 4 }, "Pitágoras" }
fmt.Println(n.Abs())  // Imprime 5
```

La sobreescritura de los métodos funciona exactamente igual que con los atributos:

```
type PuntoNombre struct{
    Point
    nombre string
}

func (n *PuntoNombre) Abs() float {
    return n.Punto.Abs() * 100
}

n := &PuntoNombre { Punto { 3, 4 }, "Pitágoras" }
fmt.Println(n.Abs())  // Imprime 500
```

Veamos un ejemplo más avanzado acerca del uso de los métodos sobre atributos anónimos:

```
type Mutex struct { ... }
func (m *Mutex) Lock() { ... }

type Buffer struct {
    data [100]byte
    Mutex
}

var buf = new(Buffer)
buf.Lock()   // == buf.Mutex.Lock()
```

Nótese que el receptor de *Lock* es la dirección de memoria del atributo *Mutex*, no la estructura correspondiente.

6.1.6. Métodos con otros tipos

Los métodos, como ya hemos mencionado anteriormente no son exclusivos de las estructuras. Pueden ser definidos para cualquier tipo que no sea de tipo puntero.

El tipo debe estar definido de todas formas dentro del mismo paquete. No se puede por tanto escribir un método para el tipo *int*, pero sí que se puede declarar un nuevo tipo *int* y darle sus métodos correspondientes.

```
type Dia int

var nombreDia = []string {
    "Lunes", "Martes", "Miércoles", ...
}

func (dia Dia) String() string {
    return nombreDia[dia]
}
```

Imaginemos que ahora tenemos un tipo propio enumerado que conoce cómo debe imprimirse por pantalla él mismo:

```
const (
    Lunes Dia = iota;
    Martes
    Miercoles
    ...
)

var dia = Martes
fmt.Print(dia.String())  // Imprime Martes
```

6.1.7. El método String()

Existe en Go una función similar al *toString()* de otros lenguajes. Se basa en el hecho de que la función Print conoce la estructura de una función genérica String. Mediante una serie de técnicas que veremos más adelante, fmt.Print[ln] puede identificar valores que implementan el método *String()* tal y como hemos definido antes para el tipo *Dia*. Estos valores, son automáticamente formateados por el método que lo invoca:

```
fmt.Println(0, Lunes, 1, Martes) // Imprime 0 Lunes 1 Martes
```

Println es capaz de distinguir entre un 0 y un 0 de tipo Dia.

Así que, definiendo un método *String()* para tus tipos propios permitirá que estos sean impresos por pantalla de forma mucho más coherente sin realizar más trabajo.

6.1.8. Visibilidad de atributos y métodos

Por último, hablaremos de la visibilidad de los atributos y de los métodos de un tipo. Go, es bastante distinto a C++ en el area de la visibilidad. Las reglas que se aplican en Go son:

1. Go tiene visibilidad local a nivel de paquete (C++ a nivel de fichero).
2. La forma de escribir una variable o método determina su visibilidad (público o exportado / privado o local).
3. Las estructuras definidas en el mismo paquete, tienen acceso a cualquier atributo y método de cualquier otra estructura.
4. Un tipo de datos local puede exportar sus atributos y sus métodos.
5. No existe una herencia propiamente dicha, así que no existe la noción de *protected*.

Estas reglas tan simples parece que funcionan correctamente en la práctica.

6.2. Interfaces

Las interfaces forman parte del aspecto más inusual que podría esperarse en Go. Es posible que sean algo distinto a lo que el lector pueda haber visto hasta ahora, por lo que se recomienda estudiar este apartado cautelosamente y sin ningún tipo de preconcepción sobre interfaces.

6.2.1. Introducción sobre interfaces

Todos los tipos de datos que hemos visto hasta ahora han sido muy concretos: Todos implementaban algo.

Go proporciona un tipo extra que hay que considerar: los *interfaces*. Un interfaz es algo completamente abstracto y que por sí solo no implementa nada. Aunque se podría pensar que no sirve de nada, ya que no implementa nada, si que define una serie de propiedades que su implementación debe tener.

Las interfaces en Go son muy similares a las interfaces de Java, y aunque Java posee un tipo *interface*, Go implementa un nuevo concepto denominado 'ìnterface value.º valor de un interfaz.

6.2.2. Definición de interfaz

La palabra "interfaz" está algo sobrecargada en Go: Existe un concepto de interfaz, un tipo *interface* y existen valores de dicho tipo. Vayamos paso por paso, el concepto dice:

Definición de interfaz: Una interfaz es un conjunto de métodos.

Se puede tomar dicha definición de otra forma ya que los métodos implementados por un tipo concreto de datos como *struct*, conforman la interfaz de dicho tipo.

Veamos un pequeño ejemplo:

```
type Punto struct { x, y float }
func (p *Punto) Abs() float { ... }
```

Con ese tipo que ya hemos visto anteriormente, podemos definir que su interfaz consta de un único método:

```
Abs() float
```

No confundir el interfaz con la declaración de la función, ya que el interfaz abstrae completamente el receptor del mismo.:

```
func (p *Punto) Abs() float { ... }
```

Si volvemos atrás, se puede observar que teníamos el tipo *Punto* embebido en un tipo *NombrePunto*. Este último tendría la misma interfaz.

6.2.3. El tipo interface

Un tipo *interface* es una especificación de un interfaz, es decir, un conjunto de métodos que son implementados por otros tipos. Veamos un ejemplo de un tipo *interface* con lo que hemos visto hasta ahora:

```
type AbsInterface interface {
    Abs() float   // El receptor es implícito
}
```

Esta es la definición de una interfaz implementada por *Punto*, o en nuestra terminología: *Punto* implementa *AbsInterface*.

También, siguiendo la misma lógica: *NombrePunto* y *Punto3* implementan *AbsInterface*.

Veamos un pequeño ejemplo:

```
type MyFloat float

func (f MyFloat) Abs() float {
    if f < 0 { return -f }
    return f
}
```

MyFloat implementa *AbsInterface* a pesar de que el tipo nativo *float* no lo hace.

Una interfaz puede ser implementada por un número arbitrario de tipos. *AbsInterface* es implementada por cualquier tipo que tenga un método *Abs() float*, independientemente del resto de métodos que el tipo pueda tener.

Asimismo, un tipo puede implementar un número arbitrario de interfaces. *Punto* implementa, al menos, estas dos interfaces:

```
type AbsInterface interface { Abs() float }
type EmptyInterface interface { }
```

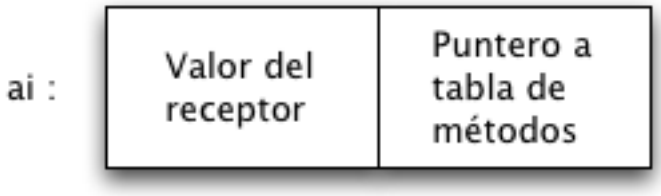
Todos los tipos implementarán una interfaz vacía *EmptyInterface*.

6.2.4. El valor interfaz

Una vez una variable es declarada con un tipo *interface*, puede almacenar cualquier valor que implemente dicha interfaz.

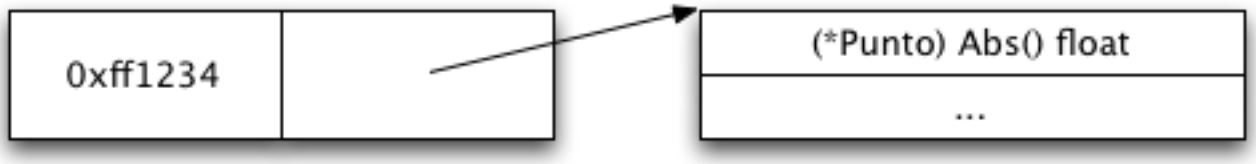
```
var ai AbsInterface

pp := new(Punto)
ai = pp  // OK: *Punto tiene Abs()
ai = 7.  // Error en tiempo de compilación: float no tiene Abs()
ai = MyFloat(-7.) // OK: MyFloat tiene Abs()

ai=&Point { 3, 4 }
fmt.Printf(ai.Abs()}
```

El anterior ejemplo imprimirá por pantalla *5*. Hay que tener en cuenta que *ai* no es un puntero. En realidad *ai* es una estructura de datos de varias palabras. Veamos un ejemplo con un gráfico para entenderlo mejor:

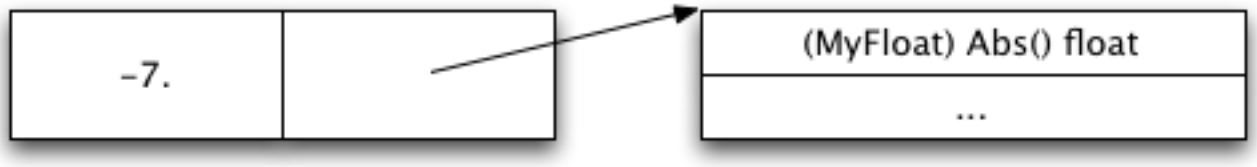


En distintos momentos de la ejecución del programa posee distintos valores y tipo:

```
ai = &Punto { 3, 4 }  (== (*Punto) (0xff1234)):
```



```
ai = MyFloat(-7.):
```



6.2.5. Hechos de los interfaces

Hay que tener en cuenta tres hecho sobre los interfaces en Go:

1. Los interfaces definen un conjunto de métodos. Son puros y completamente abstractos: No implementan nada, no tienen atributos. Go posee una clara separación entre interfaz e implementación.

2. Los valores de un interfaz son simplemente eso: valores. Contienen un valor concreto que implementa todos los métodos definidos en el interfaz. *Dicho valor puede o puede no ser un puntero*.

3. Los tipos implementan interfaces simplemente teniendo sus métodos. No es necesario declarar explícitamente que lo hacen. Por ejemplo, como ya hemos visto, todos los tipos implementan *EmptyInterface*.

6.2.6. Ejemplo: io.Writer

El paquete *io* concede una serie de métodos para el programador que pueden tratar la entrada y salida de datos. Para ver de forma más clara cómo son las interfaces, se ha elegido la interfaz *io.Wirter* por ser bastante representativa y muy usada en Go.

Si echamos un vistazo a la cabecera de la función *Fprintf* del paquete *fmt*, observamos que tiene la siguiente forma:

```
func Fprintf (w io.Writer, format string, a ...)
             (n int, error os.Error)
```

Es decir, dicha función lo que recibe es una variable u objeto que sea de tipo io.Writer, no necesariamente un fichero donde escribir. El parámetro a de tipo "..." debería obviarse por ahora ya que se explicará más adelante. Como se puede observar, Fprintf devuelve una tupla con el número de caracteres escritos y el error en caso de que hubiera alguno.

La interfaz io.Writer está definida de la siguiente manera:

```
type Writer interface {
   Write (p []byte) (n int, err os.Error)
}
```

Lo que quiere decir que todos los objetos que instancien dicha interfaz deberán tener un método Write que reciba un buffer de bytes y devuelva el número de caracteres escritos y un error en caso de que éste exista.

Si echamos un vistazo al paquete *bufio* que contiene métodos para realizar una entrada / salida con un buffer de datos, observamos que implementa un nuevo tipo de datos:

```
type Writer struct { ... }
```

Implementando dicho tipo de datos (*bufio.Writer*) el método *Writer* canónico visto en el interfaz del mismo nombre.

```
func (b *Writer) Write (p []byte)
                (n int, err os.Error)
```

Casi todos los paquetes de entrada / salida así como los que se encargan de la escritura en ficheros utilizan el interfaz Write.

El paquete io tiene declarados 4 interfaces distintos, uno para cada uso distinto:

- Reader
- Writer
- ReadWriter
- ReadWriteCloser

6.2.7. Comparación con C++

En C++ un tipo interfaz es como una clase abstracta pura, ya que se especifican los métodos pero no se implementa ninguno de ellos.

En términos de Java, el tipo interfaz es mucho más parecido a una interfaz Java.

En Go, hay una gran diferencia en un aspecto: Un tipo no necesita declarar las interfaces que implementa ni siquiera heredar de un tipo interfaz, sino que si tiene los métodos que definen el interfaz, entonces dicho tipo implementa el interfaz.

6.2.8. Contenedores y la interfaz vacía

Los contenedores son estructuras de datos que permiten almacenar elementos de un determinado tipo o de cualquiera. Veamos cómo se realiza esto en una posible implementación de los vectores.

```
type Element interface {}

//Vector es el propio contenedor
type Vector struct {
   a []Element
}

// At() devuelve el i-ésimo elemento
func (p *Vector) At(i int) Element {
   return p.a[i]
}
```

Como se puede observar en el código anterior los vectores pueden contener elementos de cualquier tipo de datos ya que todos los tipos implementan la interfaz vacía (en este caso Element es una interfaz vacía). De hecho, cada elemento contenido en un vector podría ser de cualquier tipo.

6.2.9. Asertos de tipos

Una vez que se almacena un elemento en un *Vector*, dicho elemento se almacena como un valor *interfaz*. Es necesario por lo tanto "desempaquetarlo" para tener el valor original, usando los asertos de tipos (o type assertions). Su sintaxis es:

```
valor_interfaz.(tipo_a_extraer)
```

Veamos un ejemplo de cómo funcionarían los asertos:

```
var v vector.Vector

v.Set (0, 1234.)  // Se guarda como un valor de interfaz
i := v.At(0)      // Devuelve el valor como interface{}

if i != 1234. {}            // Error en tiempo de compilación
if i.(float) != 1234. {}    // OK
if i.(int) != 1234 {}       // Error en tiempo de ejecución
if i.(MyFloat) != 1234. {}  // Error: no es MyFloat
```

6.2.10. Conversión de una interfaz a otra

Hasta ahora únicamente nos hemos limitado a mover valores normales dentro y fuera de valores interfaz, pero los valores de este tipo que contengan los métodos apropiados, también pueden ser convertidos a otros tipos de interfaces.

De hecho, es lo mismo que desempaquetar el valor interfaz para extraer el valor concreto al que se hace referencia, para después volverlo a empaquetar para el nuevo tipo interfaz.

La viabilidad de la conversión depende del valor referenciado, no del tipo original de la interfaz.

Veamos un ejemplo de la conversión de intefaces, dadas las siguientes definiciones de variables y tipos:

```
var ai AbsInterface
type SqrInterface interface { Sqr() float }
var si SqrInterface
pp := new(Point) // asumamos que *Point tiene Abs, Sqr
var empty interface{}
```

Todas estas operaciones serían correctas:

```
empty = pp
        // Todo satisface empty
ai = empty.(AbsInterface)
        // El valor referenciado
        // implementa Abs(). En cualquier otro
        // caso, produce un fallo en tiempo de ejecución.
si = ai.(SqrInterface)
        // *Point tiene Sqr()
        // aunque AbsInterface no lo implemente
empty = si
        // *Point implementa el conjunto vacío
```

6.2.11. Probando interfaces con asertos

En muchas ocasiones querremos saber con qué tipo de interfaz estamos tratando, es por ello que podemos gracias a los asertos preguntar qué tipo de interfaz tiene una determinada variable.

Para ello se puede usar los asertos de tipo çoma ok”:

```
elem := vector.At(0)

if i, ok := elem.(int); ok {
   fmt.Printf ("int: \%d\n, i)
} else if f, ok := elem.(float); ok {
   fmt.Printf ("float: \%g\n", f)
} else {
   fmt.Print("tipo desconocido\n")
}
```

También podemos comprobar los interfaces con un switch de tipos:

```
switch v := elem.(type) {  // Literal "type"
case int:
   fmt.Printf ("int: \%d\n", v)
case float:
   fmt.Printf ("float: \%g\n", v)
default:
   fmt.Print("tipo desconocido\n")
```

Y yendo un paso más allá, podemos preguntar si un valor de tipo interfaz implementa un método. De esta forma, por ejemplo, es como *Print* decide si el tipo puede imprimirse así mismo o si debe usar una representación genérica del mismo.

```
type Stringer interface { String() string }

if sv, ok := v.(Stringer); ok {
   fmt.Printf("implementa String(): \%s\n", sv.String())
}
```

6.2.12. El paquete reflect

En Go existe un paquete *reflect* que utiliza la misma base de determinar los tipos de los parámetros en las llamadas a las funciones usando los asertos. Así es posible usar un nuevo argumento "..." que permite pasar argumentos indefinidos a una función. Veamos un ejemplo con *Printf*:

```
func Printf (format string, args ...) (n int, err os.Error)
```

El argumento "..." usado dentro de Printf (o en cualquier otro sitio) tiene el tipo *interface*{}, y *Printf* usa el paquete *reflect* para desempaquetar y descubrir la lista de argumentos.

Como resultado, tenemos que toda la familia de funciones *Print* saben los tipos de todos sus argumentos. Como conocen si el argumento es unsigned o long, no hay necesidad de *%u* o *%ld*, simplemente *%d*, simplificando mucho la lista de modificadores.

Capítulo 7

Concurrencia y comunicación

A medida que la informática avanza, se va haciendo más necesario el uso de elementos de comunicación y concurrencia. La comunicación es algo necesario para no formar parte de un sistema aislado, y así poder comunicar nuestro programa con otros programas, que puede que estén o no en nuestra misma máquina.

La concurrencia, por otra parte, nos permite evitar situaciones indeseadas y realizar programación paralela, haciendo que el aprovechamiento de los recursos sea mucho más eficaz. Go es un lenguaje creado con dos fines: uno de ellos es la simplicidad, y el otro es su eminente enfoque para sistemas concurrentes.

7.1. Goroutines

7.1.1. Definición

Podríamos definir de manera formal una Goroutine como:

"Una función de Go que permite ejecutar otra función de manera paralela al programa y en el mismo espacio de direcciones que realiza la llamada a la Goroutine."

Todos los programas en Go constan de una o varias Goroutines.

Aunque pueda llevar a confusión, una Goroutine no es lo mismo que un thread, un proceso, etc. Una Goroutine es una Goroutine.

Evidentemente, y aunque pueda parecer un sistema perfecto, tiene sus problemas de concurrencia asociados que se irán viendo más adelante. Por ahora asumamos que funciona tal y como se ha dicho.

7.1.2. Cómo crear una Goroutine

Crear una Goroutine es quizá lo más sencillo que haya. Basta con invocar una función anteponiéndole la palabra reservada *go*:

```
func IsReady (what strgin, minutes int64) {
   time.Sleep (minutes * 60 * 1e9)
   fmt.Println (what, "está listo")
}

go IsReady ("té", 6)
go IsReady ("café", 2)
fmt.Println ("Estoy esperando...")
```

El ejemplo anterior imprimiría:

```
Estoy esperando...    (de forma inmediata)
café está listo       (2 minutos después)
té está listo         (6 minutos después)
```

7.1.3. Verdades delas Goroutines

Las Goroutines son baratas, es decir, son muy fáciles de programar y de invocar.

Las Goroutines terminan su ejecución retornando de su función superior, o simplemente saliendo en algún punto intermedio del programa puesto a tal propósito. También pueden llamar a *runtime.Goexit()*, aunque es raramente necesario.

Las Goroutines pueden ejecutarse concurrentemente en diferentes procesadores con memoria compartida. Asimismo no hay por qué preocuparse del tamaño de la pila a la hora de invocarlas.

7.1.4. Pila de una Goroutine

En *gccgo*, al menos por ahora, las goroutines se compilan como si fueran pthreads. Sus pilas son grandes. Se pretende cambiar esto en un futuro próximo pero no se ha realizado todavía ya que necesitan realizarse cambios en el propio *gcc*.

Usando el compilador *6g*, las pilas son pequeñas (unos pocos kB) pero crecen a medida que sea necesario. De esta forma, las goroutines usan muy poca memoria pero si es necesario pueden de forma dinámica hacer crecer la pila para invocar a muchas más funciones.

Esto se ha diseñado y programado así para que el programador no tenga que preocuparse sobre el límite de la pila y la tarea le resulte mucho más cómoda.

7.1.5. Scheduling

Las goroutines están multiplexadas entre múltiples threads del sistema. Cuando una goroutine ejecuta una llamada bloqueante al sistema, el resto de goroutines no se bloquean y continúan su ejecución.

Se pretende realizar este mismo efecto para las goroutines dependientes de la CPU, pero por ahora si se quiere un paralelismo a nivel de usuario se necesita dar un valor a la variable de entorno *$GOMAXPROCS*, o se puede llamar en un programa a *runtime.GOMAXPROCS(n)*.

GOMAXPROCS le dice al planificador en tiempo de ejecución cuántas goroutines que no sean bloqueantes por llamadas al sistema, pueden correr a la vez. Esto implica, que para hacer una prueba real de paralelismo, necesitaremos adecuar el valor de *$GOMAXPROCS* al número de goroutines que tenga nuestro programa.

¿Cómo se puede dar un valor a dicha variable? Sencillo. En un terminal del sistema basta con poner las siguientes líneas:

```
$ GOMAXPROCS=4
$ export GOMAXPROCS
```

7.2. Channels

A menos que dos goroutines puedan comunicarse, éstas no pueden coordinarse o sincronizarse de ninguna forma. Por ello Go tiene un tipo denominado *channel* que prove comunicación y sincronización. También posee de estructuras de control especiales para los canales que hacen fácil la programación concurrente.

7.2.1. El tipo channel

En su forma más simple el tipo se declara como:

```
chan tipo_elemento
```

Dada una variable de ese estilo, puedes enviar y recibir elementos del tipo *tipo_elemento*.

Los canales son un tipo de referencia, lo que implica que se pude asignar una variable *chan* a otra y que ambas variables accedan al mismo canal. Esto también hace referencia a que hay que usar *make* para alojar un canal:

```
var c = make (chan int)
```

7.2.2. El operador $<-$

El operador $<-$ (más comúnmente conocido como "flechita") es un operador que nos permite pasar datos a los canales. La flecha apunta en la dirección en que se pretende que sea el flujo de datos.

Como operador binario, $<-$ envía a un canal:

```
var c chan int
c <- 1      // envía 1 al canal c
```

Como un operador unario prefijo, $<-$ recibe datos de un canal:

```
v = <- c    // Recibe un valor de c y se lo asigna a v.
<- c        // Recibe un valor de c, descartándolo
i := <- c   // Recibe un valor de c inicializando i
```

7.2.3. Semántica de canales

Por defecto la comunicación en Go se realiza de forma síncrona. Más tarde abordaremos la comunicacion asíncrona. Que la comunicación sea síncrona significa:

1. Una operación send en un canal bloquea la ejecución hasta que se ejecuta una operación receive en el mismo canal.
2. Una operación receive en un canal bloquea la ejecución hasta que una operación send es ejecutada en el mismo canal.

De esta forma, la comunicación es además una forma de sincronización: 2 goroutines intercambiando datos a través de un canal, se sincronizan en el momento en que la comunicación es efectiva.

7.2.4. Ejemplo de comunicación

Veamos un ejemplo bastante tonto de comunicación pero que servirá como toma de contacto con la sintaxis de los canales:

```
func pump (ch chan int) {
   for i := 0; ; i++ {
      ch <- i
   }
}

ch1 := make (chan int)
go pump (ch1)       // pump se bloquea; ejecutamos:
fmt.Println (<-ch1)     // Imprime 0
```

```
func suck (ch chan int) {
   for {
      fmt.Println (<-ch)
   }
}

go suck (ch1)    // Miles de números se imprimen

// Todavía se puede colar la función principal y pedir un valor
fmt.Println(<-ch)    // Imprime un valor cualquiera (314159)
```

7.2.5. Funciones que devuelven canales

En el ejemplo anterior, *pump* era una función generadora de valores para ser recogidos por otra función. El ejemplo anterior se puede simplificar bastante en el momento en el que se cree una función que pueda devolver el canal en sí para que se puedan recoger los datos.

```
func pump() chan int {
   ch := make (chan int)
   go func() {
      for i := 0; ; i++ {
         ch <- i
      }
   }()
   return ch
}

stream := pump()
fmt.Println(<-stream)   // Imprime 0
```

7.2.6. Rangos y canales

La claúsula *range* en bucles *for* acepta un canal como operando, en cuyo caso el bucle *for* itera sobre los valores recibidos por el canal.

Anteriormente reescribimos la función pump(), con lo que ahora reescribimos la función *suck* para que lance la goroutine también:

```
func suck (ch chan int) {
   go func() {
      for v:= range ch {
         fmt.Println(v)
      }
   }()
}

suck (pump())   // Ahora esta llamada no se bloquea
```

7.2.7. Cerrando un canal

Cuando queremos cerrar un canal por alguna razón usamos una función del lenguaje que nos permite realizar dicha acción.

```
close (ch)
```

De igual forma, existe otra función que nos permite comprobar si un canal está cerrado o no, usando la función *closed()*:

```
if closed (ch) {
   fmt.Println ("hecho")
}
```

Una vez que un canal está cerrado y que todos los valores enviados han sido recibidos, todas las operaciones receive que se realicen a continuación recuperarán un valor ”cero” del canal.

En la práctica, raramente se necesitará *close*.

Hay ciertas sutilezas respecto al cierre de un canal. La llamada a *closed()* sólo funciona correctamente cuando se invoca después de haber recibido un valor ”cero” en el canal. El bucle correcto para recorrer los datos de un canal y comprobar si se ha cerrado es:

```
for {
   v := <-ch
   if closed (ch) {
      break
   }
   fmt.Println (v)
}
```

Pero de una forma mucho más idiomática y simple, podemos usar la cláusula *range* que realiza esa misma acción por nosotros:

```
for v := range ch {
   fmt.Println (v)
}
```

7.2.8. Iteradores

Ahora que tenemos todas las piezas necesarias podemos construir un iterador para un contenedor. Aquí está el código para un *Vector*:

```
// Iteramos sobre todos los elementos

func (p *Vector) iterate (c chan Element) {
   for i, v := range p.a {   // p.a es un slice
      c <- v
   }
   close (c)     // Señala que no hay más valores
}

// Iterador del canal

func (p *Vector) Iter() chan Element {
   c := make (chan Element)
   go p.iterate (c)
   return c
}
```

Ahora que el *Vector* tiene un iterador, podemos usarlo:

```
vec := new (vector.Vector)
for i := 0 i < 100; i++ {
   vec.Push (i*i)
}

i:= 0
for x := range vec.Iter() {
   fmt.Printf ("vec[\%d] is \%d\n", i, x.(int))
   i++
}
```

7.2.9. Direccionalidad de un canal

En su forma más simple una variable de tipo canal es una variable sin memoria (no posee un buffer), síncrona y que puede ser utilizada para enviar y recibir.

Una variable de tipo canal puede ser anotada para especificar que únicamente puede enviar o recibir:

```
var recv_only <-chan int
var send_only chan <- int
```

Todos los canales son creados de forma bidireccional, pero podemos asignarlos a variables de canales direccionales. Esto es útil por ejemplo en las funciones, para tener seguridad en los tipos de los parámetros pasados:

```
func sink (ch <- chan int) {
   for { <- ch }
}

func source (ch chan<- int) {
   for { ch <- 1 }
}

var c = make (chan) int)  //bidireccional
go source (c)
go sink (c)
```

7.2.10. Canales síncronos

Los canales síncronos no disponen de un buffer. Las operaciones send no terminan hasta que un receive es ejecutado. Veamos un ejemplo de un canal síncrono:

```
c := make (chan int)
go func () {
   time.Sleep (60 * 1e9)
   x := <- c
   fmt.Println ("recibido", x)
}()

fmt.Println ("enviando", 10)
c <- 10
fmt.Println ("enviado", 10)
```

Salida:

```
enviando 10  (ocurre inmediatamente)
enviado 10   (60s después, estas dos lineas aparecen)
recibido 10
```

7.2.11. Canales asíncronos

Un canal asíncrono con buffer puede ser creado pasando a *make* un argumento, que será el número de elementos que tendrá el buffer.

```
c := make (chan int, 50)
go func () {
   time.Sleep (60 * 1e9)
   x := <-c
   fmt.Println ("recibido", x)
}()
```

```
fmt.Println ("enviando", 10)
c <- 10
fmt.Println ("enviado", 10)

enviando 10  (ocurre inmediatamente)
enviado 10   (ahora)
recibido 10   (60s después)
```

Nota.- El buffer no forma parte del tipo canal, sino que es parte de la variable de tipo canal que se crea.

7.2.12. Probando la comunicación

Si nos preguntamos si un receive puede en un momento determinado ejecutar sin quedarse bloqueado, existe un método que permite hacerlo de forma atómica, probando y ejecutando, utilizando para tal efecto el sistema ”coma ok”.

```
v, ok = <-c   // ok = true si v recibe un valor
```

¿Y qué hay de realizar una operación send sin bloquear? Hace falta averiguarlo y ejecutar. Para ello usamos una expresión booleana:

```
ok := c <- v

o también:

if c <- v {
   fmt.Println ("valor enviado")
}
```

Normalmente, se quiere una mayor flexibilidad, que conseguiremos con la siguiente estructura que veremos en el uso de canales.

7.3. Select

7.3.1. Definición

Select es una estructura de control en Go, análoga a un switch asociado a comunicación. En un *select* cada case debe ser una comunicación: send o receive.

```
var c1, c2 chan int

select {
case v := <- c1:
   fmt.Printf ("recibido \%d de c1\n", v)
case v := <- c2:
   fmt.Printf ("recibido \%d de c2\n", v)
}
```

Select ejecuta un case que pueda ejecutarse de forma aleatoria. Si ninguno de los case es posible de ejecutar, se bloquea hasta que uno lo sea. Una cláusula *default* es siempre ejecutable.

7.3.2. Semántica de select

Hagamos un rápido resumen del select:

- Cada case debe ser una comunicación
- Todas las expresiones del canal son evaluadas
- Todas las expresiones para ser enviadas son evaluadas
- Si alguna comunicación puede producirse, se produce; En otro caso se ignora.
  - Si existe una cláusula *default*, es ésta la que ejecuta
  - Si no existe *default*, el bloque select se bloquea hasta que una comunicación pueda realizarse; no hay reevaluación de los canales y/o posibles valores.
- Si hay múltiples casos listos, uno es seleccionado aleatoriamente. El resto no ejecuta.

7.3.3. Ejemplo: Generador aleatorio de bits

Un ejemplo bastante ilustrativo es el siguiente:

```
c := make (chan int)
go func() {
   for {
      fmt.Println (<-c)
   }
}()

for {
   select {
      case c <- 0:   // No hay sentencias, no existe el fall-through
      case c <- 1:
   }
}
```

Esto imprime: 0 1 1 0 0 1 1 1 0 1 ...

7.4. Multiplexación

Los canales son valores de ”primer tipo”, es decir, pueden ser enviados a través de otros canales. Esta propiedad hace que sea fácil de escribir un servicio multiplexador dado que el cliente puede proporcionar, junto con la petición, el canal al que debe responder.

```
chanOfChans := make (chan chan int)
```

O de forma más típica:

```
type Reply struct { ... }
type Request struct {
   arg1, arg2, arg3 some_type
   replyc chan *Reply
}
```

Vamos a ver un ejemplo cliente-servidor a lo largo de los siguientes puntos.

7.4.1. El servidor

```
type request struct {
   a, b int
   replyc chan int
}

type binOp func (a, b int) int

func run (op binOp, req *request) {
   req.replyc <- op(req.a, req.b)
}

func server (op binOp, service chan *request) {
   for {
      req := <- service  // Aquí se aceptan las peticiones
      go run(op, req)
   }
}
```

Para comenzar el servidor lo hacemos de la siguiente forma:

```
func startServer (op binOp) chan *request {
   req := make (chan *request)
   go server (op, req)
   return req
}

var adderChan = startServer(
   func a, b int) int { return a + b }
)
```

7.4.2. El cliente

```
func (r *request) String() string {
   return fmt.Sprintf ("\%d+\%d=\%d",
                       r.a, r.b, <-r.replyc)
}
req1 := &request{ 7, 8, make (chan int) }
req2 := &request{ 17, 18, make (chan int) }

adderChan <- req1
adderChan <- req2

fmt.Println (req2, req1)
```

7.4.3. La contrapartida

En el ejemplo anterior el servidor ejecuta en un bucle infinito, con lo que siempre está esperando peticiones. Para terminarlo de manera correcta, se le puede señalar con un canal. El siguiente servidor tiene la misma funcionalidad pero con un canal *quit* para terminar el proceso.

```
func server (op binOp, service chan *request,
             quit chan bool) {
   for {
      select{
         case req := <- service:  // Aquí se aceptan las peticiones
            go run(op, req)
         case <- quit:
            return
      }
   }
}
```

El resto del código es básicamente el mismo, sólo que con un canal más:

```
func startServer (op binOp) (service chan *request,
                             quit chan bool) {
   req := make (chan *request)
   quit = make (chan bool)
   go server (op, service, quit)
   return service, quit
}

var adderChan, quitChan = startServer(
   func a, b int) int { return a + b }
)
```

El cliente permanece inalterado ya que puede terminar el servidor cuando quiera con sólo ejecutar la sentencia:

```
quitChan <- true
```

7.5. Problemas de concurrencia

Como siempre que se programa de forma concurrente existen un montón de problemas pero Go trata de hacerse cargo de ellos.

El envío y la recepción (operaciones send y receive) son atómicas. La sentencia *select* está programada y definida muy cuidadosamente para que no contenga errores.

El problema viene principalmente dado porque las goroutines ejecutan en memoria compartida, las redes de comunicacción pueden colapsarse, que no existen debuggers multihilo...

El consejo por parte de los desarrolladores del lenguaje es no programar como si estuviéramos en C o C++, o incluso Java.

Los canales dan sincronización y comunicación en una misma estructura, lo cual los hace muy potentes, pero también hacen que sea fácil razonar si se están usando de manera correcta o no.

La regla es:

"No comunicar programas con memoria compartida, sino compartir memoria comunicándose"

El modelo básico a la hora de programar utilizando la filosofía cliente-servidor se resume en dos conceptos:

1. Usar un canal para enviar datos a una goroutine dedicada como servidor, ya que si una única goroutine en cada instante tiene un puntero a los datos, no hay problemas de concurrencia.
2. Usar el modelo de programación "un hilo por cliente" de forma generalizada, usado desde la década de los 80 y que funciona bastante bien.

Capítulo 8

Modelo de Memoria de Go

Este capítulo contiene información que afecta específicamente a la programación concurrente en Go. Especialmente se centra en cómo afectan las operaciones a las lecturas de datos de las variables en una goroutine de forma que se observe el valor correcto producido por una escritura de la misma variable en otra goroutine.

8.1. Lo que primero ocurre

En el ámbito de una única goroutine, las lecturas y escrituras deben comportarse como si fueran ejecutadas en el orden especificado por el programa. Esto es, los compiladores y procesadores pueden reordenar las lecturas y escrituras ejecutadas en el ámbito de una única goroutine sólo cuando la reordenación no cambia el comportamiento en esa goroutine definido por la especificación del lenguaje. Debido a esta reordenación, el orden de ejecución observado por una goroutine puede diferir del orden percibido por otra. Por ejemplo, si una goroutine ejecuta *a = 1; b=1;*, otra puede observar el valor actualizado de *b* antes que el de *a*.

Para especificar los requisitos de lecturas y escrituras, definimos ”lo que primero ocurre”, un orden parcial en la ejecución de operaciones de memoria en un programa escrito en Go. Si el evento e_1 ocurre antes que otro e_2, entonces decimos que e_2 ocurre después que e_1. También, si e_1 no ocurre antes que e_2 y no ocurre después que e_2, entonces decimos que e_1 y e_2 ocurren concurrentemente.

En una única goroutine, el orden de ”lo que primero ocurre” es el orden expresado en el programa.

La lectura *r* de una variable *v* puede observar una escritura *w* en *v* si las siguientes dos afirmaciones ocurren:

1. *w* ocurre antes que *r*.
2. No hay otra escritura *w* en *v* que ocurra después de *w* pero antes que *r*.

Para garantizar que una lectura *r* de una variable *v* observa un valor particular escrito por *w* en *v*, hay que asegurar que *w* es la única escritura que *r* puede observar. Esto es, *r* está garantizado que lee la escritura *w* si las siguientes afirmaciones se cumplen:

1. *w* ocurre antes que *r*.
2. Cualquier otra escritura en la variable compartida *v* ocurre o bien antes que *w* o bien después que *r*.

Este par de condiciones es más estricto que el primero; requiere que no haya otras escrituras ocurriendo concurrentemente con *w* o con *r*.

En una única goroutine no existe concurrencia, así pues las dos definiciones son equivalentes: una lectura *r* observa el valor escrito por la escritura más reciente *w* en *v*. Cuando varias goroutines acceden a una variable compartida *v*, deben utilizar eventos de sincronización para establecer qué es lo que ocurre primero y así asegurar que las lecturas observan los valores correctos de las variables compartidas.

La inicialización de una variable *v* con el valor ”cero” para el tipo de *v* se comporta como se ha definido en este modelo de memoria..

Las lecturas y escrituras de valores que ocupan más que el ancho de palabra de la máquina, se comportan como varias operaciones del ancho de la palabra de forma no especificada.

8.2. Sincronización

La sincronización entre las distintas goroutines es básica para que el programa en cuestión pueda ejecutar correctamente y dar resultados satisfactorios.

8.2.1. Inicialización

La inicialización del programa se ejecuta en una única goroutine y las nuevas goroutines creadas durante la inicialización no comienzan su ejecución hasta que la inicialización termina.

Si un paquete *p* importa un paquete *q*, la inicialización de las funciones de *q* terminan antes de que empiece la inicialización de cualquiera de las de *p*.

La ejecución de la función *main.main* comienza después de que todas las funciones *init* hayan terminado.

La ejecución de todas las goroutines creadas durante la ejecución de las funciones *init* ocurre después de que todas las demás funciones *init* hayan terminado.

8.2.2. Creación de Goroutines

La sentencia *go* que comienza una nueva goroutine se ejecuta antes de que la ejecución de la propia goroutine comience. Veamos un ejemplo al respecto:

```
var a string

func f() {
    print (a)
}

func hello() {
    a = ''Hello, World!"
    go f()
}
```

Una posible llamada a *hello* imprimirá *"Hello World!"* en algún momento en el futuro (quizá después de que *hello* haya terminado).

8.2.3. Comunicación mediante canales

La comunicación mediante canales es el método principal de sincronización entre gorotinnes. Cada send en un canal particular se corresponde con un receive en ese mismo canal, que se ejecuta normalmente en otra goroutine.

Un send en el canal se ejecuta antes de que la ejecución del receive correspondiente de dicho canal finalice.

Este programa garantiza la impresión de *"Hello, World!"*. La escritura de *a* ocurre antes del send en *c*, que ocurre antes de que el correspondiente receive en *c* termine, cosa que ocurre siempre antes que *print*.

```
var a string

func f() {
    a = ''Hello, World!"
    c <- 0
}

func main() {
    go f()
    <- c
    print (a)
}
```

Un receive de un canal sin buffer ocurre siempre antes de que el send en ese canal termine.

El siguiente programa también garantiza que se imprime *"Hello, World!*. La escritura en *a* ocurre antes que el receive en *c*, lo que ocurre antes de que se realice el correspondiente send *c*, lo que ocurre antes de que se ejecute el *print*.

```
var c = make (chan int)
var a string

func f() {
    a = ''Hello, World!"
    <- c
}

func main() {
    go f()
    c <- 0
    print (a)
}
```

Si el canal tuviera buffer (por ejemplo, *c = make (chan int, 1)*) entonces el programa no podría garantizar la impresión de *"Hello, World!"*. Puede imprimir la cadena vacía; No puede imprimir *"Hello, sailor"*, aunque tampoco puede terminar de forma inesperada.

8.3. Cerrojos - Locks

El paquete *sync* implementa dos tipos de cerrojos, *sync.Mutex* y *sync.RWMutex*.

Para cualquier variable *l* de tipo *sync.Mutex* o *sync.RWMutex* y para n ¡m, la n-ésima llamada a *l.Unlock()* ocurre antes de que la m-ésima llamada a *l.Lock()* termine.

El siguiente programa garantiza la impresión de *"Hello, World!"*. La primera llamada a *l.Unlock()* (en *f*) ocurre antes de que la segunda llamada a *l.Lock()* (en *main*) termine, lo que ocurre antes de que se ejecute *print*.

```
var l sync.Mutex
var a string

func f() {
    a = ''Hello, World!''
    l.Unlock()
}

func main() {
    l.Lock()
    go f()
    l.Lock()
    print(a)
}
```

Para cualquier llamada a *l.RLock* en una variable *l* de tipo *sync.RWMutex*, existe una n tal que la llamada a *l.RLock* termina después que la n-ésima llamada a *l.Unlock* y su correspondiente *l.RUnlock* ocurre antes de la n+1-ésima llamada a *l.Lock*.

8.4. El paquete once

El paquete *once* provee de un mecanismo seguro para la inicialización en presencia de varias goroutines. Múltiples threads pueden ejecutar *once.Do(f)* para una *f* particular, pero sólo una de ellas ejecutará la función *f()* y el resto de llamadas se bloquearán hasta que *f()* haya terminado su ejecución.

Una única llamada a f() desde once.Do(f) termina antes de que cualquier otra llamada a once.Do(f) termine.

En el siguiente programa la llamada a *twoprint* provoca que *"Hello, World!"* se imprima dos veces. La primera llamada a *twoprint* ejecuta *setup* una vez.

```
var a string

func setup() {
    a = ''Hello, World!''
}

func doprint() {
    once.Do(setup)
    print(a)
}

func twoprint() {
    go doprint()
    go doprint()
}
```

8.5. Sincronización incorrecta

En este apartado se va a tratar los problemas más comunes que pueden darse en una sincronización.

Nótese que una lectura *r* puede observar el valor escrito por una escritura *w* que ocurre concurrentemente con *r*. Aunque esto ocurra, no implica que las lecturas que ocurran después de *r* observen escrituras que hayan ocurrido antes que *w*.

En el siguiente programa puede ocurrir que *g* imprima *2* y luego *0*.

```
var a,b int

func f() {
    a = 1
    b = 2
}

func g() {
    print(b)
    print(a)
}

func main() {
    go f()
    g()
}
```

La realización de un doble intento de bloqueo para conseguir exclusión mutua, es un intento de eliminar la sobrecarga de la sincronización. Por ejemplo, el programa *twoprint* puede ser escrito incorrectamente como se puede observar a continuación, pero no garantiza que, en *doprint*, observando la escritura en la variable *done* no implica observar la escritura en *a*. Esta versión puede (incorrectamente) imprimir una cadena vacía en lugar de *"Hello, World!"*.

```
var a string
var done bool

func setup() {
    a = ''Hello, World!''
    done = true
}

func doprint() {
    if !done {
        once.Do(setup)
    }
    print(a)
}

func twoprint() {
    go doprint()
    go doprint()
}
```

Otro ejemplo de una mala sincronización es:

```
var a string
var done bool

func setup() {
    a = ''Hello, World!''
    done = true
}

func main() {
    go setup()
    for !done {
    }
    print(a)
}
```

De la misma forma de antes no existe garantía que en *main*, observando la escritura en *done* no implica observar la escritura en *a*, así que este programa también puede imprimir la cadena vacía. Peor aún, no hay garantías de que la escritura en *done* sea observable desde *main*, dado que no hay eventos de sincronización entre los dos threads. Asimismo el bucle en *main* no garantiza su finalización.

Hay sutiles variantes en este tema, como este programa:

```
type T struct {
    msg string
}

var g *T

func setup() {
    t := new (T)
    t.msg = ''Hello, World!''
    g = t
}

func main() {
    go setup()
    for g == nil {
    }
    print (g.msg)
}
```

Incluso si el *main* observa *g != nil* y sale de su bucle, no hay garantía de que observe el valor inicializado de *g.msg*

Para terminar, veamos un pequeño ejemplo de una incorrecta sincronización mediante canales. El siguiente programa, de la misma forma que los anteriores, puede no imprimir *"Hello, World!"*, imprimiendo una cadena vacía dado que la goroutina no termina su ejecución antes de que lo haga el *main*.

```
var c chan int

func hello() {
    fmt.Println(''Hello, World!'')
    c <- 0
}

func main() {
    go hello()
}
```

Dado que se realiza un send al canal pero no se realiza su receive correspondiente en main, no se garantiza que hello termine de ejecutar antes de que termine la ejecución de main, que aborta el programa.