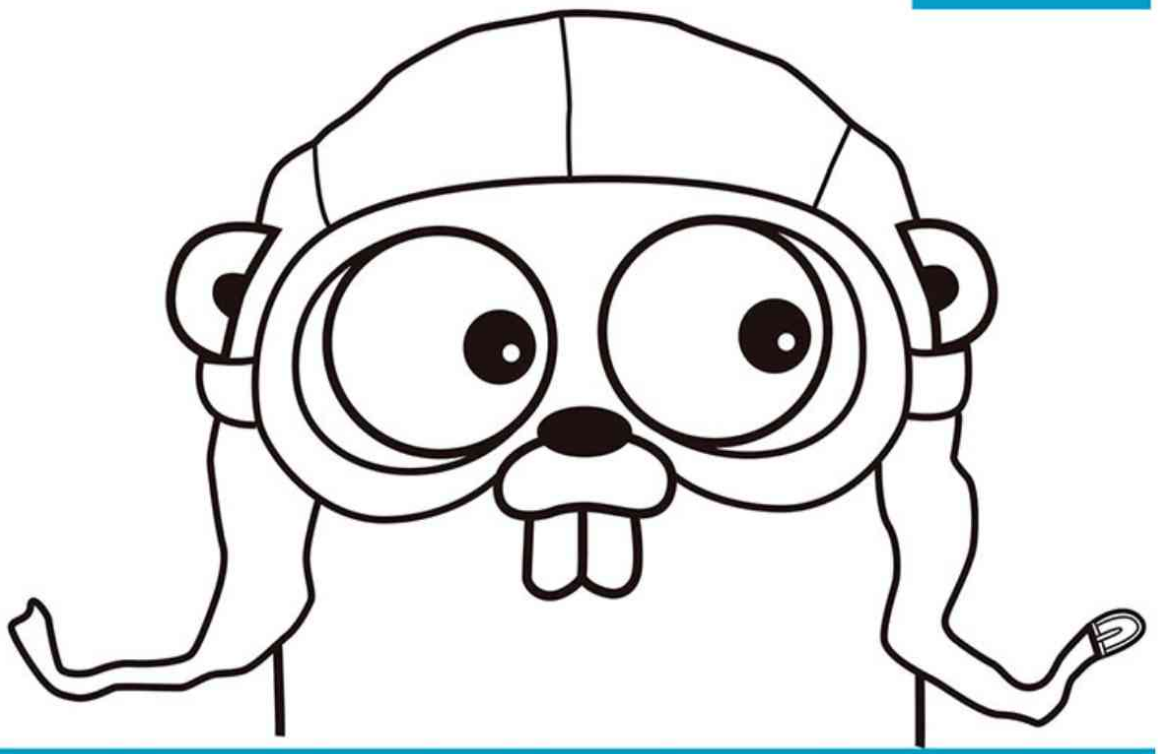
CONTENIDOS WEB



PROGRAMACIÓN EN GO

MARIO MACÍAS LLORET

Marcombo 1945-2020 75 años

PROGRAMACIÓN EN GO

Mario Macías Lloret

PROGRAMACIÓN EN GO

Mario Macías Lloret

1945-2020
Marcombo 75 años

Programación en Go

Primera edición, 2021

www.marcombo.com

Diseño de cubierta: ENEDENÚ DISEÑO GRÁFICO
Maquetación: Reverte-Aguilar
Correctora: Anna Alberola
Directora de producción: M.a Rosa Castillo
Producción del ePub: booqlab

escanear algún fragmento de esta obra.

ISBN: 978-84-267-3249-1

Este libro está dedicado a Isabel.
Sin tus ideas y ánimos, tu paciencia y generosidad,
estas palabras nunca habrían sido posibles.

TABLA DE CONTENIDO

INTRODUCCIÓN

Go es el lenguaje de moda entre informáticos. Sin duda, es uno de los lenguajes de programación que durante la última década han ganado más impulso entre los programadores de diversas disciplinas. Pese a ser un lenguaje relativamente "joven", no ha tardado en hacerse con una legión, ya no de adeptos, sino de auténticos *fans*.

Son muchas las causas del éxito de Go:

- Es **versátil.** Combina características de los lenguajes de bajo nivel, como C++, con características de lenguajes dinámicos, como Python o Ruby. Esto hace de Go un lenguaje tan idóneo para software de sistema (controladores, comandos de sistema, agentes de monitorización, incluso programación de sistemas embebidos gracias a la implementación de tinygo.org) como para la creación de grandes aplicaciones web y sistemas de servicios distribuidos.
- Es **rápido.** Es un lenguaje cuyos ejecutables se distribuyen en código nativo, sin necesidad de máquinas virtuales o intérpretes de lenguaje. Compila las técnicas de optimización más vanguardistas.
- Es **compacto.** Genera ejecutables pequeños que incluyen todo el código necesario, sin necesidad de bibliotecas externas o entornos de ejecución extra.
- Es **muy rápido compilando.** Está enfocado hacia las tendencias actuales de integración y despliegue continuo de aplicaciones, mediante las cuales el software está en continua actualización. Hoy, el tiempo de compilación es una métrica clave para la productividad de los equipos de desarrollo.
- Es **seguro.** A diferencia de otros lenguajes como C o C++, donde un apuntador a memoria "desbocado" puede suponer un grave fallo de seguridad, Go comprueba la seguridad de los accesos a memoria de tal

manera que un usuario malintencionado lo tendrá mucho más difícil para encontrar fallos de seguridad explotables.

•Es **sencillo.** Los equipos de desarrollo modernos pasan muchas horas revisando código de sus compañeros, con tal de reforzar unos estándares de calidad altos. Go es un lenguaje diseñado para ser fácil de leer y entender, lo cual incentiva unas prácticas y estilos globales y unificados, y evita proveer múltiples soluciones para una misma tarea.

•Es **completo.** La distribución estándar de Go proporciona casi todas las herramientas que un profesional necesita: gestores de dependencias, analizadores de rendimiento, formateadores y analizadores de código, depuradores, gestión de la documentación, una enorme biblioteca estándar de funcionalidades, etc.

•Es **código abierto.** El código de todas las herramientas oficiales de Go, así como su librería estándar, es abierto y libre de modificar y distribuir. Además, sus bibliotecas de terceros también son código abierto.

ACERCA DE ESTE LIBRO

Este libro pretende ser un punto tanto de contacto como de profundización en el lenguaje de programación Go. Está destinado tanto a personas con conocimientos básicos de programación como a profesionales con experiencia que quieran adentrarse en los paradigmas y filosofía de un nuevo lenguaje.

En ningún caso es una introducción a la programación para personas que nunca hayan programado, ni un curso de algoritmia básica. No obstante, muchos de los conceptos que se presentan se explican brevemente, de manera que todo el mundo pueda entenderlos.

Si usted es un programador experto, podrá sacar buen provecho de este libro, ya que no se limita a explicar las estructuras básicas de programación adaptadas a la sintaxis de Go, sino también su filosofía y los nuevos conceptos que hacen de Go un lenguaje único y especial.

Este libro no pretende ser un manual de referencia técnico, ni un compendio de todas las bibliotecas y funciones estándar de Go. Para ese cometido ya existe la documentación oficial. Este libro pretende ser una introducción ágil —sin descuidar la profundización— a las herramientas y características que le permitirán escribir programas en Go de manera productiva, en un breve periodo de tiempo.

Como autor, humildemente —pero no por ello sin ambición—, pretendo que este libro sea la herramienta que a mí me hubiera gustado tener para agilizar mi transición profesional de programador en C y Java hacia el lenguaje Go.

ORGANIZACIÓN DEL LIBRO

Los 20 capítulos de este libro se agrupan en cuatro partes diferenciadas.

La **primera parte** comprende los capítulos del 1 al 9, y muestra los constructos esenciales de Go, comunes a casi cualquier otro lenguaje de programación imperativo y estructurado. Los programadores expertos serán capaces de leer y asimilar de manera rápida las particularidades de la sintaxis de Go, mediante breves explicaciones y ejemplos concisos. Los programadores menos iniciados encontrarán explicaciones sencillas a muchos conceptos que puedan ser nuevos para ellos.

La **segunda parte** comprende los capítulos del 10 al 14. Los conceptos aquí explicados tienen sus equivalentes en otros lenguajes de programación, aunque en Go se abordan desde otro paradigma, que cambiará la manera en que diseñamos nuestro software respecto a cuando lo hacemos para lenguajes más clásicos.

Durante el transcurso de los capítulos del 10 al 14, el lector podrá empezar a intuir por qué Go es un lenguaje "diferente", y cómo aúna filosofías que se pensaban irreconciliables, al situarse entre los lenguajes de programación de bajo nivel y los lenguajes interpretados y dinámicos.

La **tercera parte** está compuesta por los capítulos 15 y 16, en los que el lector entrará de lleno en los conceptos y herramientas que hacen de Go un lenguaje único para la computación de altas prestaciones, tanto por la potencia de sus herramientas como por su casi insultante sencillez. El lector aprenderá a lanzar miles de tareas en paralelo, a coordinarlas y a establecer una comunicación sencilla y efectiva entre estas.

La **cuarta parte** está enfocada a la programación de aplicaciones en Go. Además de su lenguaje, su filosofía y sus detalles, este libro pretende servir también como punto de contacto con la programación efectiva de aplicaciones. Por ello, la cuarta y última parte de este libro muestra, paso a paso, cómo

utilizar estas funcionalidades de la biblioteca estándar de Go que rápidamente nos permitirán empezar a programar un amplio abanico de aplicaciones.

El capítulo 17 está dedicado a la creación de servicios web en Go: cómo entablar la comunicación entre programas situados remotamente. El capítulo 18 muestra cómo serializar estructuras de datos complejas en Go a formatos de texto, para su intercambio a través de Internet o su guardado en disco, por ejemplo. El capítulo 19 explica las funciones básicas que permitirán conectar nuestras aplicaciones en Go a bases de datos relacionales, para el guardado persistente de datos estructurados y relacionados. Por último, el capítulo 20 muestra el sistema de Go para llevar a cabo una de las prácticas actualmente esenciales durante el desarrollo y mantenimiento de aplicaciones: las pruebas de código automatizadas *(testing)*.

CONVENCIONES DE FORMATO

Este libro intercala texto explicativo con extractos de código, así como con capturas de sesiones interactivas de terminal.

El código fuente se muestra en una fuente de ancho fijo, con algunas palabras clave de Go resaltadas. Ejemplo:

```
for
{

var
i = 3
  funcion(i)
}
```

Para una mejor visualización del código en el formato de un libro, las líneas de código no ocuparán más de 60 caracteres de ancho, y las indentaciones/tabulaciones ocuparán 2 caracteres.

La entrada y salida de datos a través del terminal de línea de comandos se muestra en fuente de ancho fijo y sin resaltado. Los comandos que el usuario escribe desde el terminal están precedidos del símbolo del dólar, $, a semejanza de las líneas de comando Unix/Linux:

```
$ go run hola.go
Esto es lo que muestra el programa: ¡Hola!
```

En algunos momentos, se muestran plantillas que describen de manera genérica algunas partes de la sintaxis de Go. Las partes entre símbolos < y > deben substituirse por un texto que represente el concepto explicado, sin dichos símbolos. Las partes entre símbolos [y] representan partes opcionales, que pueden omitirse.

Por ejemplo, la plantilla

```
var <nombre> <tipo> [ = valor ]
```

podría coincidir con cualquiera de las siguientes líneas válidas de Go:

```
var i int
var nombre string
var pi float32 = 3.1416
```

Por brevedad, en los ejemplos de código se omiten algunas partes que serían necesarias en un programa completo, cuando estas no aportan información interesante al ejemplo o a la explicación, tales como:

- Cabeceras de la función principal: func main()
- Definición de paquetes: package main
- Importación de paquetes externos: import "net/http"

ACERCA DEL AUTOR

Nacido durante las últimas "hornadas" de la generación *baby boom,* tuve la suerte de entablar contacto con la informática desde pequeñito gracias a mi tío Antonio, que nos consiguió un micro-ordenador Sony MSX. Cuando fui capaz de aprender programación en el lenguaje BASIC de nuestro MSX, supe que quería dedicarme profesionalmente a ello. Creo que fue gracias a la determinación surgida de este fortísimo deseo que conseguí acabar, con mucha frustración y dificultad, mis estudios de secundaria. Los odiaba, pero me permitieron llegar a estudiar Ingeniería Informática en la universidad, una de las mejores cosas que me han pasado en la vida.

Después de trabajar unos años en empresas, volví al mundo académico para trabajar y realizar mi doctorado en Arquitectura de Computadores en el Barcelona Supercomputing Center y en la Universitat Politécnica de Catalunya donde, además, tuve el privilegio de dar clases de programación durante 10 años. Al poco tiempo de volver al mundo empresarial, entré en contacto con Go, nada más entrar en la empresa de monitorización New Relic (newrelic.com).

He pretendido aunar en este libro mi pasión por la programación y mi pasión por la docencia. Solo me queda agradecerle su lectura, y esperar que lo disfrute tanto como yo disfruté escribiéndolo.

Capítulo 1
INSTALACIÓN Y USO DE GO

1.1 INSTALANDO GO

Go está disponible para descarga gratuita desde su página web:

http://golang.org

En la página principal, haga clic en el enlace titulado "Download Go", que le llevará a la página de descargas (Figura 1.1), donde encontrará los diversos paquetes para su sistema operativo (Windows, Linux, Mac).

Tan solo instale el paquete descargado y siga las instrucciones de instalación. Si la instalación ha tenido éxito, puede abrir una sesión de línea de comandos y verificar que el ejecutable go está en su ruta de programas:

```
$ go version
go version go1.15 darwin/amd64
```

La instalación por defecto de Go consiste en:

- El compilador go, que le permite generar sus archivos ejecutables.
- Herramientas útiles para la programación: depuradores, analizadores de rendimiento, comprobadores de código...
- La biblioteca estándar de Go: una extensa colección de funcionalidades que usted puede incorporar en sus programas.
- La documentación estándar de Go, accesible a través del comando godoc.

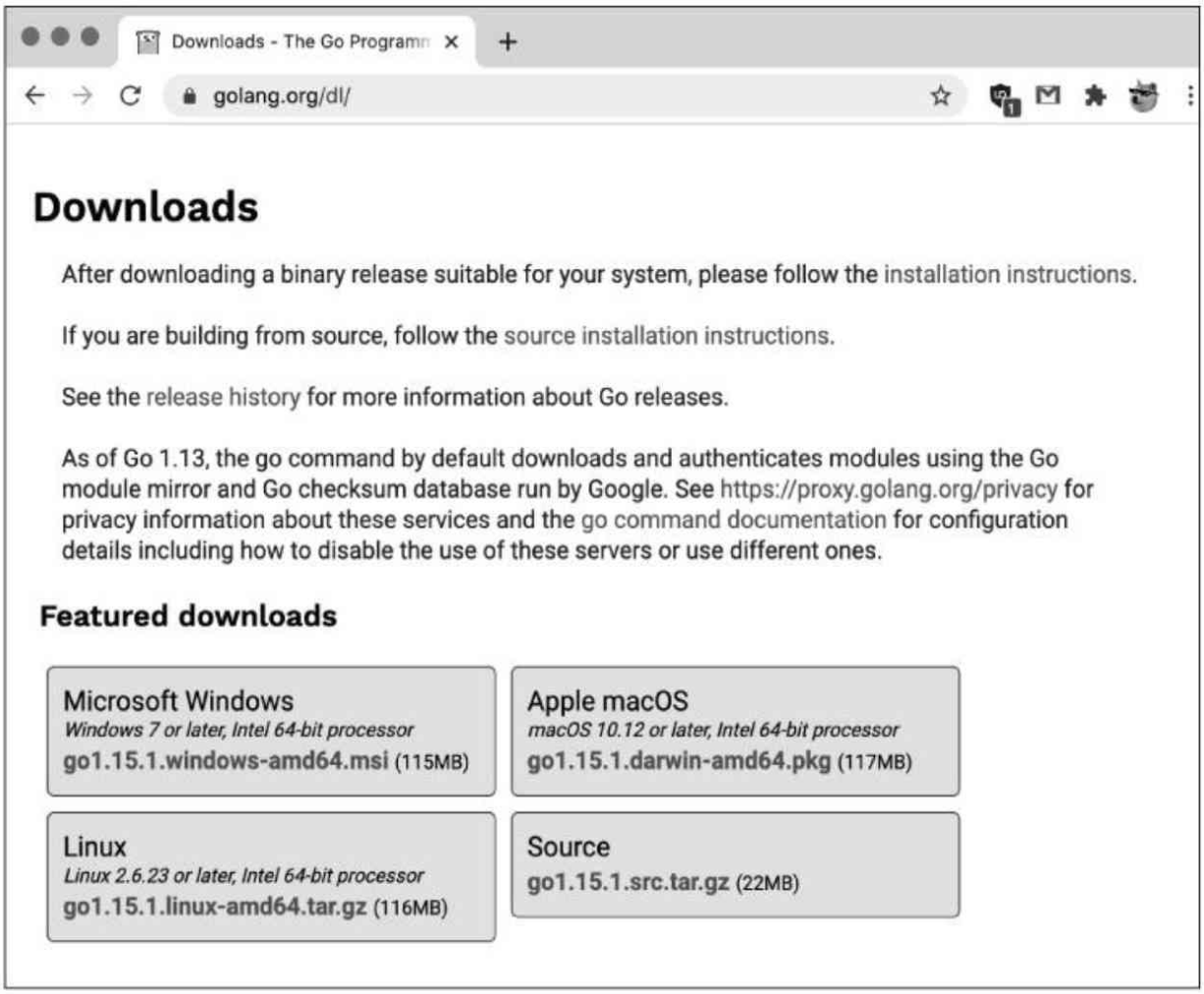


Figura 1.1 Página web oficial de descargas de Go.

1.2 COMANDOS BÁSICOS DE GO

A continuación, se muestran algunos de los comandos más habituales de Go, que pueden serle de utilidad durante su proceso de aprendizaje:

- go build
  - –Ejecutándose desde el directorio raíz de su proyecto Go, genera un archivo ejecutable del proyecto.
- go run <archivo>.go
  - –Para ejecutar su programa de Go directamente, sin generar un ejecutable. Útil durante la etapa de desarrollo.
- go fmt ./...
  - –Ejecutándose desde el directorio raíz de su proyecto, reformatea todos los archivos de código fuente según el formato estándar de Go. Su uso es, si no obligado *de facto,* sí muy recomendable.
- go vet ./...
  - –Ejecutándose desde el directorio raíz de su proyecto, busca patrones de código susceptibles de ocasionar errores o comportamientos incorrectos.
- godoc
  - –Abre un servidor web en la dirección http://localhost:6060/. Si lo abre desde su navegador, podrá navegar localmente por toda la documentación de las diferentes bibliotecas estándar de Go, disponibles en la instalación por defecto.
- go get
  - –Permite descargar bibliotecas y utilidades de línea de comandos suministradas por terceros.
- go mod
  - –Permite gestionar sus proyectos locales, así como sus dependencias hacia bibliotecas de terceros.

1.3 EDITANDO SU CÓDIGO EN GO

Para programar en Go, tan solo necesita un editor de texto plano, como un bloc de notas. Sin embargo, es aconsejable utilizar un editor de código o un entorno integrado de desarrollo, que proporciona funcionalidades tales como resaltado de código, comprobación *in situ* de errores, depuración visual, etc.

Go es compatible con la gran mayoría de editores de código y entornos de desarrollo, directamente o a través de extensiones. La tabla de la Figura 1.2 enumera, por orden alfabético, algunos de los entornos más populares para el desarrollo de proyectos en Go. Aunque su editor favorito no esté en la tabla, es posible que ofrezca soporte básico o resaltado de sintaxis para Go.

Nombre	Web	Código abierto	Gratuito
Atom	atom.io	Sí	Sí
Brackets	brackets.io	Sí	Sí
Goland	www.jetbrains.com/go/	No	No
Sublime Text	www.sublimetext.com	No	No
Visual Studio Code	code.visualstudio.com	Sí	Sí
Zeus IDE	www.zeusedit.com	No	No

Figura 1.2 Algunos editores y entornos de desarrollo populares para Go.

1.4 COMPILANDO Y EJECUTANDO SU PRIMER PROGRAMA

Comenzaremos a familiarizarnos con Go y sus herramientas mediante la creación y compilación de un programa básico. No se preocupe si no entiende todo su contenido, ya que será explicado en profundidad en los próximos capítulos.

Para empezar, abra un editor de texto plano y cree un archivo cuya extensión sea .go. Por ejemplo, hola.go. Copie el siguiente contenido:

```
package main

import "fmt"

// función principal "main"
func main() {
  // muestra un saludo en pantalla
  fmt.Println("¡Hola, geómidos!")
}
```

Del código anterior, quizás haya podido deducir algunos de los siguientes componentes de un programa básico en Go:

- El punto de entrada del programa se marca mediante func main() { . El programa comenzará ejecutando el código contenido en el bloque delimitado por llaves { y } .
- Cada fichero está encabezado por una directiva package, que otorga un nombre común para la agrupación (paquete) de todos los archivos Go en un directorio. El punto de entrada del programa debe estar encabezado por package main (paquete principal).
- Cuando se incorporan funcionalidades de la biblioteca estándar o de otras bibliotecas de terceros (por ejemplo, para mostrar un mensaje en

pantalla), deben incorporarse explícitamente los paquetes que las contienen mediante la directiva import. En el programa de ejemplo, la directiva import "fmt" permite usar funcionalidades de escritura de mensajes en pantalla (entre otras).

•Cualquier texto escrito después de dos barras // es ignorado por el compilador hasta el final de la línea. Allí puede dejar algunos comentarios que ayuden a documentar o entender el código.

–Puede usar comentarios de múltiples líneas si los sitúa entre los símbolos /* y */ .

•El comando fmt.Println muestra un mensaje en pantalla. Está compuesto por el nombre del paquete en el que está guardado (fmt) y el nombre de la función Println. Entre paréntesis, se escribe el texto a mostrar (que debe ir entre comillas dobles).

Para saber más sobre Println y sobre el paquete fmt, puede probar a ejecutar el comando godoc en su línea de comandos. Se ejecutará un servidor web local en el que puede abrir la siguiente dirección en su navegador:

```
http://localhost:6060/pkg/fmt
```

Dicha dirección le llevará a la documentación del paquete fmt, donde podrá encontrar información completa sobre las funciones que proporciona, muchas de las cuales se explicarán en el próximo capítulo y en el capítulo 14.

Para ejecutar el programa, vaya desde la línea de comandos al directorio que contiene el archivo hola.go y ejecute el siguiente comando:

```
$ go run hola.go
```

A continuación del comando escrito, verá el siguiente mensaje:

```
¡Hola, geómidos!
```

Esto permite ejecutar el programa en su ordenador personal (ya que tiene instalada la herramienta Go). Para generar un archivo ejecutable que pueda

ser ejecutado en otro ordenador, debe utilizar el comando go.build:

```
$ go build hola.go
```

El comando anterior creará un archivo ejecutable apto para la misma arquitectura y el mismo sistema operativo desde el cual compila. Si quisiera generar ejecutables para otros sistemas operativos y arquitecturas de procesador, puede usar las variables de entorno GOOS y GOARCH, respectivamente:

```
GOOS=windows GOARCH=386 go build hola.go
GOOS=linux GOARCH=arm64 go build hola.go
```

Los comandos anteriores generarían ejecutables para Windows con procesadores Intel de 32 bits, y para Linux con procesadores ARM de 64 bits, respectivamente.

Capítulo 2

SINTAXIS BÁSICA DE GO

Este capítulo explica las características básicas del lenguaje de programación Go. Los fundamentos del lenguaje Go no son, en concepto, muy diferentes de los de otros lenguajes de programación, como Java o C.

2.1 TIPOS DE DATOS BÁSICOS

Una **variable** es un espacio reservado en la memoria del ordenador para guardar datos cuyo valor puede cambiar durante la vida del programa: un número, una letra, un texto, un luchador de Street Fighter... Una variable irá asociada a un nombre para poder referirnos a ella durante el programa: matrícula, total, jugador1...

Go es un lenguaje con "tipado estático", lo que significa que cada variable tiene un tipo de dato asociado y solo podrá guardar valores de su tipo de dato. Go provee los siguientes tipos de datos básicos, a partir de los cuales es posible definir variables y ejecutar operaciones básicas:

- bool es el tipo *booleano*. Los posibles valores que puede tomar son dos: true (cierto) o false (falso).
- string permite guardar cadenas de texto, y acepta cualquier carácter Unicode mediante la codificación UTF-8 (es decir, todos los caracteres del alfabeto latino, más caracteres de otras culturas, incluso *emojis)*.
- int es un número entero con signo (es decir, permite guardar tanto números enteros positivos como negativos). Su tamaño depende de la plataforma para la cual se compila el programa. Esto significa que el compilador escogerá el tamaño de las variables int que mejor se adapten a la arquitectura del procesador para el cual se compila (generalmente, concordará con el número de bits del procesador: 32 o 64).
  - –Cuando sea necesario concretar con exactitud el tamaño de una variable, se pueden usar los siguientes tipos: int8, int16, int32, int64 para números enteros con signo de 8, 16, 32 y 64 bits, respectivamente.
- uint es un número entero sin signo (es decir, no acepta números negativos). Al igual que int, su tamaño dependerá de la plataforma (32 o 64 bits).
  - –Cuando se requiera concretar con exactitud el tamaño de una variable,

se pueden usar los siguientes tipos: uint8, uinti6, uint32 o uint64.

•byte es el equivalente de uint8.

•rune es un equivalente para int32 y se usa para representar un carácter (letras, números, signos de puntuación, *emojis,* y un largo etcétera).

•float32 y float64 representan números de coma flotante (también llamados "números reales") de 32 y 64 bits, respectivamente.

•complex64 y complex128 son números complejos cuyas partes real e imaginaria están representadas por números reales de 32 y 64 bits, respectivamente.

–Un literal de número complejo se escribe como en el siguiente ejemplo: 2.23 + 3i .

2.2 CADENAS DE TEXTO. EL TIPO string

Go permite definir cadenas de texto explícitamente, insertando un texto cualquiera entre comillas dobles:

```
texto := "- Hola, ¿cómo estás?"
```

Si una cadena ha de contener un texto mostrado en diversas líneas, puede introducir el carácter especial de nueva línea \n allá donde quiera que termine una línea y empiece otra:

```
texto := "- Hola, ¿cómo estás?\n- Estoy bien, gracias."
```

Si una cadena ha de contener comillas dobles en su interior, estas deben especificarse como un carácter especial \", para que Go no las confunda como el final de una cadena de texto:

```
texto := "Podría decirse que estoy \"bien\"..."
```

Cuando un texto contiene múltiples líneas o comillas dobles, puede resultar más limpio substituir el delimitador de comillas dobles por el de "acento grave"; esto le permitirá escribir cadenas en múltiples líneas, tomando los saltos de línea como literales. El equivalente a la cadena anterior, sería:

```
texto := `- Hola, ¿cómo estás?
- Estoy "bien", gracias.`
```

No debe confundir los delimitadores de cadenas con la comilla simple ', que sirve para especificar caracteres individuales, del tipo byte o rune.

2.3 DEFINICIÓN DE VARIABLES

La manera de definir una variable, dado un nombre y un tipo de dato, es:

var <nombre> <tipo> [= <valor>]

Por ejemplo:

var dias int **var** meses int = 12

Las líneas anteriores definirían una variable llamada dias, del tipo int, y otra variable llamada meses, del tipo int, a la que se le asigna un valor inicial de 12.

Si no se provee un valor inicial, las variables serán inicializadas automáticamente con el "valor cero" de cada tipo de dato: 0 para tipos numéricos, false para bool, y la cadena vacía "" para string.

En aras de la brevedad, se puede omitir tanto la palabra var como su tipo si se usa el operador de inicialización :=:

meses := 12

El compilador de Go inferirá el tipo de la variable meses a un int (ya que es un número sin parte decimal), y aunque el valor de la variable cambie, el tipo siempre deberá ser int.

El símbolo := solo se puede usar para definir una variable. Una vez definida, se usará el operador de asignación = si se desea cambiar su valor. Por ejemplo:

```
paso := 1 // definición de la variable con un valor inicial
paso = 2 // cambio del valor de la variable
```

Las guías de estilo de Go recomiendan declarar variables con el operador : = siempre que se pueda, aunque las variables globales y las que no tengan valor inicial deberán declararse mediante la forma precedida por var.

2.4 CONVERSIONES EXPLÍCITAS DE TIPOS

En Go, al contrario de lo que ocurre en otros lenguajes de tipado estático, si se quiere asignar una variable numérica a otra variable de otro tipo numérico, se deberá explicitar el tipo de destino de la siguiente manera:

```
var segundos int8 = 30 var
horas int
horas = int(segundos)
```

En el fragmento anterior, se indica a Go que el valor de la variable segundos, del tipo int8, se va a copiar en otra variable del tipo int. Aunque es obvio que cualquier valor del tipo int8 (8 bits) cabe en una variable del tipo int (32 o 64 bits), Go obliga a hacer explícita esta conversión.

Si la conversión se hace desde un tipo entero de tamaño superior al tipo de destino, se usan los bits menos significativos del tipo de origen que caben en el tipo de destino.

Si la conversión se hace desde un tipo de coma flotante, se trunca la parte con decimales y se asigna la parte entera. Por ejemplo:

```
distancia := 12.78
kms := int(distancia)
```

En el ejemplo anterior, la variable kms tomará el valor 12 (eliminando, sin redondeo alguno, el 0.78 de la variable original).

2.5 CONSTANTES

Una constante es un valor literal al que se le asigna un nombre, que no puede cambiar durante la vida del programa. Su definición es similar a la de una variable, pero reemplazando la palabra var por la palabra const. Por ejemplo:

```
const Pi = 3.1416
```

o

```
const Pi float64 = 3.1416
```

Cuando se definen múltiples constantes, se pueden agrupar semánticamente bajo la misma directiva const. Por ejemplo:

```
// Configuración de una tipografía
const (
  TipoFuente   = "Times New Roman"
  AlturaFuente = 12
  Subrayado    = false
  Negrita      = true
)
```

2.6 BASES DE NUMERACIÓN

Además de la comúnmente usada base decimal (en la que cada dígito puede tomar diez valores distintos, del 0 al 9), Go permite usar otros tipos de bases de numeración según el prefijo que añadamos a cada número (Figura 2.1).

Base	Prefijo	Descripción	Ejemplo
Binario	`0b`	Cada dígito puede ser 0 o 1	`0b10001101`
Octal	`0`	Cada dígito puede tomar valores del 0 al 7	`012072`
Hexadecimal	`0x`	Cada dígito puede tomar valores del 0 al 9, y de la A a la F representando los valores del 10 al 15	`0xAF32`

Figura 2.1 Bases de numeración en Go.

Cuando queramos agrupar bloques de dígitos, podemos insertar un guión bajo "_" entre estos. Go no lo tendrá en cuenta a la hora de establecer el valor. Ejemplos:

```
productoInteriorBruto := 1_419_000_000_000
bitsAgrupados := 0b_1000_1001_0110
```

2.7 OPERADORES NUMÉRICOS

Go permite hacer las siguientes operaciones con tipos de datos númericos, por orden de precedencia:

1. Agrupaciones de operaciones, delimitadas por paréntesis.
2. Multiplicaciones, divisiones (operadores * y /), así como el resto de la división entera (o módulo, operador %).
3. Sumas y restas (operadores + y -).

El orden de precedencia hace referencia a qué operaciones se evalúan primero, cuando una expresión compleja engloba múltiples operaciones. Primero, se evalúan las operaciones de mayor precedencia. En caso de múltiples operaciones con la misma precedencia, estas se evalúan según su posición en la expresión, de izquierda a derecha.

Por ejemplo, dada la siguiente expresión:

```
a := 8 + 3 * (1 + 2) % 5
```

1. Primero evaluaría la expresión entre paréntesis, ya que es la de mayor precedencia: a := 8 + 3 * 3 % 5
2. Luego evaluaría la multiplicación y el módulo. Al ser de la misma precedencia, primero evaluaría la multiplicación, ya que está más a la izquierda: a := 8 + 9 % 5
3. Y continuaría por el resto de la división entera: a := 8 + 9
4. Siendo la suma la operación de menos precedencia, sería la última en evaluarse: a := 12

Además de los anteriores operadores matemáticos, Go provee los operadores de incremento (++) y decremento (--), que van detrás de una variable que se quiere incrementar o decrementar, respectivamente:

```
a := 10
b := 20
a++
```

b--

Después de ejecutar el anterior programa, la variable a contendría el valor 11 y la variable b contendría el valor 19.
A diferencia de otros lenguajes de programación con los que el lector pueda estar familiarizado, los operadores de incremento y decremento no pueden ir dentro de otras expresiones.

2.8 OPERADORES NUMÉRICOS DE COMPARACIÓN

Go permite comparar expresiones numéricas (Figura 2.2). El resultado de una comparación será un bool (true o false). Cuando se mezclan con las operaciones numéricas del apartado anterior, las operaciones de comparación son las de menor precedencia.

Operador	Descripción	Ejemplo
==	Comprueba si el valor de la izquierda es igual al de la derecha	`3 + 5 == 8` (resultado: `true`)
!=	Comprueba si el valor de la izquierda es diferente al de la derecha	`3 + 5 != 8` (resultado: `false`)
<	Comprueba si el valor de la izquierda es menor al de la derecha	`3 < 3` (resultado: `false`)
<=	Comprueba si el valor de la izquierda es menor o igual al de la derecha	`3 <= 1 + 2` (resultado: `true`)
>	Comprueba si el valor de la izquierda es mayor al de la derecha	`3 > 3 - 1` (resultado: `true`)
>=	Comprueba si el valor de la izquierda es mayor o igual al de la derecha	`3 >= 2` (resultado: `false`)
<	Comprueba si el valor de la izquierda es menor al de la derecha	`3 < 2` (resultado: `false`)

Figura 2.2 Operadores de comparación.

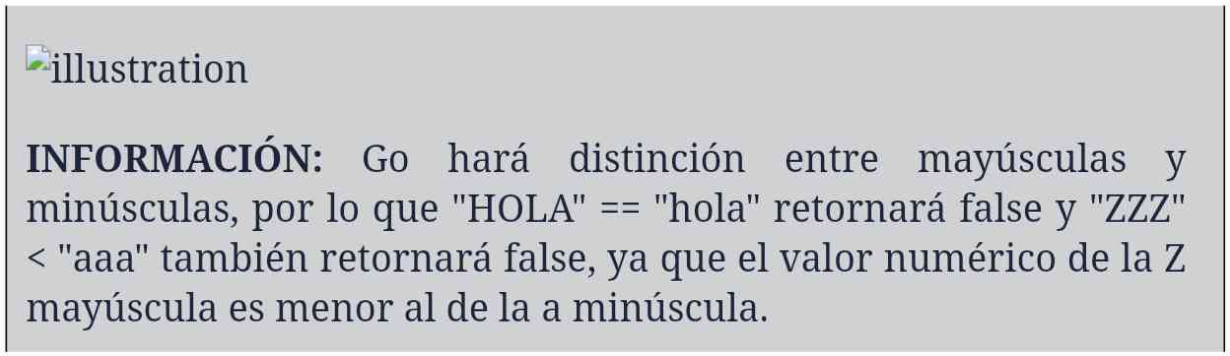
2.9 OPERADORES APLICABLES AL TIPO string

El operador + aplicado entre string, retorna un nuevo string con las cadenas concatenadas:

```
a := "el cocherito"
b := "leré"
concat := a + “ " + b
```

En el ejemplo anterior, la variable concat contendría la cadena "el cocherito leré".

Los operadores de comparación también pueden usarse para comparar cadenas de texto.



2.10 OPERADORES LÓGICOS CON bool

Cualquier variable o expresión que retorne un bool puede combinarse con otros booleanos mediante los siguientes operadores lógicos (Figura 2.3), resultando en otros valores booleanos. Cuando se mezclan con las operaciones de comparación del apartado anterior, las operaciones de lógica son las de menor precedencia.

Operador	Descripción	Ejemplo
`&&`	*AND*. Retorna `true` si —y solo si— ambos operadores booleanos son `true`	`6 == 6 && 3 > 2` (resultado: `true && true`, entonces `true`)
`||`	*OR*. Retorna `true` en caso de que uno de los operadores booleanos sea `true`	`false || 3 < 8` (resultado: `false || true`, entonces `true`)
`!` (unario)	*NOT*. Invierte `true` por `false` y `false` por `true` en el operador situado a la derecha	`!(7 < 1)` (resultado: `!false`, entonces `true`)

Figura 2.3 Operadores lógicos.

2.11 OPERADORES LÓGICOS A NIVEL DE BIT

Las variables numéricas también pueden verse como agrupaciones de dígitos binarios (bits) con los que podemos realizar las siguientes operaciones, que se aplican tras comparar los bits de igual posición en dos operandos (Figura 2.4).

Símbolo	Descripción	Ejemplo
&	*AND*. Resulta `1` cuando si —y solo si— ambos bits son `1`	`0b_1010 & 0b_1100 == 0b_1000`
|	*OR*. Resulta `1` cuando al menos uno de los bits es `1`	`0b_1010 | 0b_1100 == 0b_1110`
^	*XOR* (o exclusivo). Resulta `1` si los bits comparados son diferentes, o `0` si son iguales	`0b_1010 ^ 0b_1100 == 0b_0110`
^ (unario)	*NOT*. Cuando se aplica delante de un solo operador, invierte el valor de cada dígito	`^0b_100110 == 0b_011001`
<<	Desplazamiento a la izquierda. Desplaza cada bit del operador izquierdo tantas posiciones como el número del operador derecho. Los espacios libres a la derecha se rellenan con ceros	`0b_1001 << 2 == 0b_100100`
>>	Desplazamiento a la derecha. Desplaza cada bit del operador izquierdo tantas posiciones como el número del operador derecho. Los bits más a la derecha son eliminados	`0b_1001 >> 2 == 0b_0010`

Figura 2.4 Operadores lógicos a nivel de bit.

2.12 SALIDA ESTÁNDAR DE DATOS

El paquete fmt, del cual pudo ver una pequeña muestra en el capítulo anterior, permite mostrar datos en su terminal de línea de comandos (también llamado **salida estándar)** mediante las siguientes funciones:

```
fmt.Print
fmt.Println
fmt.Printf
```

fmt.Print y fmt.Println enviarán a la salida estándar los datos que sitúe entre paréntesis, y separados por comas, a continuación del nombre de la función. Para los datos que no sean cadenas de texto, Go hará una conversión genérica a texto antes de enviarlos a la salida estándar. Go, además, añadirá un espacio entre los diferentes datos dentro de una misma invocación a fmt.Print o fmt.Println.

Ejemplo:

```
x := 33
fmt.Println("Hola, número", x, "!")
```

Salida estándar:

```
Hola, número 33 !
```

La diferencia entre fmt.Print y fmt.Println es que fmt.Println añade un salto de línea al final. Es decir, sucesivas invocaciones a fmt.Print serían mostradas una detrás de otra, en la misma línea; mientras que sucesivas invocaciones a fmt.Println serían mostradas en diferentes líneas, una debajo de otra.

Cuando requiera un control más exhaustivo de cómo se deben mostrar los datos, la función fmt.Printf admite una cadena de texto en la que puede colocar unas "marcas de formato", conocidas como **verbos**. A continuación, separados por comas, se colocan los datos que Go debe introducir en el lugar de cada uno de los verbos.

Por ejemplo, el siguiente ejemplo intercala los verbos %v, que muestran el valor de una variable, y que serán substituidos por las variables a continuación del texto de formato, en orden de aparición:

```
cosa := "depósito"
x := 36
y := 84
fmt.Printf("Coordenadas de %v:  (%v, %v)\n", cosa, x, y)
```

Salida estándar:

```
Coordenadas de depósito: (36, 84)
```

Observe que fmt.Printf no añade ninguna nueva línea al final, por lo que si necesita que el siguiente texto aparezca en la línea siguiente, debe finalizar la cadena de formato con el carácter especial de nueva línea: '\n'.

La tabla de la Figura 2.5 muestra algunos otros verbos útiles aceptados por fmt.Printf. Para más detalles, visite la documentación del paquete fmt a través del comando godoc que se introdujo en el capítulo anterior.

Verbo	Descripción
%v	Valor en el formato por defecto (tal como se mostraría en `fmt.Print`)
%T	Tipo de datos
%d	Valor numérico en base decimal
%b	Valor numérico en base binaria
%o	Valor numérico en base octal
%x	Valor numérico en base hexadecimal (con letras a-f en minúscula)
%X	Valor numérico en base hexadecimal (con letras A-F en mayúscula)
%e	Notación científica (ejemplo: `-1.234e+10`)
%f	Número en coma flotante, sin exponente (ejemplo: `-12.34`)
%c	Un carácter individual
%s	Cadena de texto
%q	Cadena de texto delimitada por comillas dobles, con todos los caracteres especiales substituidos por marcas de Go (por ejemplo, en vez de mostrar un salto de línea, mostraría `\n`)

Figura 2.5 Algunos verbos útiles para fmt.Printf.

2.13 ENTRADA ESTÁNDAR DE DATOS

La **entrada estándar** permite a un programa de Go obtener datos desde el exterior, a través del teclado en la línea de comandos.

Go proporciona dos funciones para obtener datos desde la entrada estándar:

```
fmt.Scan
fmt.Scanf
```

fmt.Scan lee los datos del teclado y los guarda en las variables pasadas en la invocación. Cada variable debe ir precedida por el símbolo *ampersand,* & (sabrá el por qué cuando llegue al capítulo 4, sobre apuntadores). Por ejemplo:

```
var
edad int
fmt.Print("Edad? ")
fmt.Scan(&edad)
fmt.Println("Tienes", edad, "años")
```

Ejemplo de entrada y salida estándar:

```
Edad? 36
Tienes 36 años
```

En el ejemplo anterior, si los datos introducidos no fueran un número entero válido, ignoraría la entrada:

```
Edad? manuel
Tienes 0 años
```

fmt.Scanf permite especificar con más detalle el formato de la entrada, tomando como primer parámetro una cadena de texto en la que se pueden introducir los diversos verbos (como los de la tabla de la Figura 2.5), que se colocarían en sus respectivas variables, ya que tanto fmt.Scan como fmt.Scanf

aceptan múltiples variables:

```
var
hora, minuto, segundo int
fmt.Print("HH:MM:SS? ")
fmt.Scanf("%d:%d:%d", &hora, &minuto, &segundo)
fmt.Printf("%d horas, %d minutos, %d segundos",
  hora, minuto, segundo)
```

Ejemplo de entrada y salida estándar:

```
HH:MM:SS? 12:34:56
12 horas, 34 minutos, 56 segundos
```

Capítulo 3
CONTROL DE FLUJO

Los programas de ejemplo mostrados hasta ahora seguían un "flujo secuencial": las operaciones se ejecutan una a una, según su orden descendiente de escritura.

Como en los demás lenguajes de programación, Go permite alterar el flujo secuencial mediante los bloques de control de flujo, que se agrupan en dos tipos:

- **Condicionales.** Permiten que un bloque de instrucciones se ejecute o no, dependiendo de si se cumple una condición en el programa.
- **Iterativos.** Permiten que un bloque de instrucciones se ejecute repetidamente mientras se dé una condición.

Por "condición" entendemos cualquier expresión booleana que retorne true o false, como las mostradas en el capítulo anterior.

3.1 BLOQUES CONDICIONALES

3.1.1 if

El bloque if permite agrupar un conjunto de instrucciones que se ejecutarán si —y solo si— su condición asociada es true. Su estructura es:

```
if <condición> {
  // conjunto de instrucciones que se ejecutarán
  // si condición == true
}
```

Por ejemplo, el siguiente programa genera un número aleatorio, e informa de que el número generado es par. Dicho mensaje de información solo se mostrará si el número es realmente par:

```
package main

import (
  "fmt"
  "math/rand"
)

func main() {
  valor := rand.Int()
  if valor % 2 == 0 {
    // bloque a ejecutar solo si la condición es true
    fmt.Println("El número ", valor, " es par")
  }
  fmt.Println("Adiós!")
}
```

Nótese la importación del paquete math/rand y el uso del comando rand.Int() para la generación de números aleatorios.

En el programa anterior, el mensaje dentro del bloque if se ejecutará solo si el número aleatorio es divisible por dos (el resto de su división entera es 0).

Si no se quiere considerar el valor 0 como número par, la condición del bloque if podría completarse mediante el uso de los operadores lógicos vistos en el capítulo anterior:

```
if valor != 0 && valor%2 == 0
```

3.1.2 if ... else

Un bloque if puede ser inmediatamente continuado por un bloque else, que ejecutaría el bloque de instrucciones asociado si —y solo si— la condición del if es false. Basándonos en el anterior ejemplo:

```
func main() {
  valor := rand.Int()
  if valor % 2 == 0 {
    // bloque a ejecutar solo si la condición es true
    fmt.Println("El número", valor, "es par")


  } else {
    // bloque a ejecutar solo si la condición es false
    fmt.Println("El número", valor, "es impar")
  }
  fmt.Println("Adiós!")
}
```

Por brevedad, se ha omitido la definición del paquete y los import.

En el programa anterior, siempre se ejecutará uno de los dos bloques en exclusiva: o el código dentro del bloque if (en caso de que valor sea divisible por dos), o el código dentro del bloque else (en caso de que valor no sea divisible por dos); pero nunca se ejecutarán los dos bloques a la vez.

En los ejemplos anteriores, la variable valor existe y es accesible durante toda la vida de la función main, incluso si solo se necesita en el contexto de los bloques if y else. Es una buena práctica restringir el ciclo de vida de una variable lo máximo que se pueda.

Go permite definir una variable dentro de un if, limitando su ciclo de vida a la evaluación de la condición y las instrucciones de los bloques if y else:

```
if <definición de variable> ; <condición> {
  //
}
```

Aplicándolo al ejemplo anterior:

```
func main() {
  if valor := rand.Int(); valor % 2 == 0 {
    // bloque a ejecutar si la condición es true
    fmt.Println("El número", valor, "es par")
  } else {
    // bloque a ejecutar si la condición es false
    fmt.Println("El número", valor, "es impar")
  }
  fmt.Println("Adiós!")
}
```

La diferencia es que, mientras anteriormente se podía usar la variable valor en cualquier punto de la función main(), con esta nueva forma la variable valor deja de existir después del bloque else (por ejemplo, no podríamos imprimir su valor en el mensaje de despedida).

CONSEJO: Limitar al máximo la vida de una variable nos ahorrará muchos errores en nuestros programas, algunos de ellos difíciles de detectar.

3.1.3 switch - case

Cuando una variable puede tomar múltiples valores de una enumeración concreta, y se debe realizar una acción distinta para cada uno de estos valores, la forma más concisa de expresarlo es mediante un bloque switch/case:

```
switch <expresión a evaluar> {
  case <valor posible 1>:
    // código
  case <valor posible 2>:
    // código
    // ...
  default:
    // código a ejecutar si la expresión no toma
    // ninguno de los posibles valores anteriores
}
```

La orden switch también puede incluir una inicialización de variable, similar a la forma if anteriormente vista:

switch <inicialización variable> ; <expresión a evaluar>

El siguiente ejemplo hace uso del paquete runtime y sus variables globales runtime.GOARCH y runtime.GOOS para extraer información de la arquitectura y del sistema operativo en el que el programa se ejecuta. Como en muchos otros ejemplos a partir de ahora, se omite la definición del paquete, los import y la definición de la función main().

```
fmt.Print("La arquitectura de su procesador es ")

arch := runtime.GOARCH
switch arch {
  case "386":
    fmt.Println("x86 de 32 bits")
  case "amd64":
    fmt.Println("x86 de 64 bits")
  default:

    fmt.Println(arch)
}

fmt.Print("Su sistema operativo es ")

switch os := runtime.GOOS; os {
  case "darwin":
    fmt.Println("Mac OS X")
  case "linux":
    fmt.Println("GNU/Linux")
  case "hurd":
    fmt.Println("GNU/Hurd")
  default:
    fmt.Println(os)
}
```

El código anterior mostraría algo similar al siguiente texto (dependiendo del ordenador en el que lo ejecutara):

La arquitectura de su procesador es x86 de 64 bits

Su sistema operativo es windows

El primer bloque switch del ejemplo anterior evaluará el valor de la variable

arch y mostrará un mensaje distinto dependiendo de si esta variable contiene "386" o "amd64" (valores definidos en las cabeceras de sus respectivos bloques case). El bloque default: se ejecutará si el valor de la variable arch no coincide con ninguno de los valores definidos tras cada case (por ejemplo, si su procesador fuera un ARM, comúnmente usado en tablets y teléfonos inteligentes).

La variable arch, definida en la función main, existirá y será visible desde cualquier punto de la función main (a partir de la declaración de la variable). Sin embargo, observe que el segundo bloque switch define la variable os en la misma línea en la que se comprueba (separado por punto y coma, de manera análoga al bloque if de la sección anterior). En este caso, la existencia y visibilidad de la variable os se limitará al bloque switch.

Si el lector está familiarizado con otros lenguajes de programación, como Java o C, habrá notado la ausencia de la palabra break al final de cada bloque case. En aras de la brevedad, Go la omite. Cuando finaliza el código dentro de un bloque case, el flujo del programa continúa fuera del switch.

Si por algún motivo fuera necesario que, tras acabar el código de un case, el flujo continuara por el siguiente bloque case, se debe usar el comando fallthrough. Ejemplo:

```
switch letra {
  case 'A':
    fmt.Println("Mayúscula")
    fallthrough
  case 'a':
    fmt.Println("Primera del abecedario")
}
```

En el ejemplo anterior:

- Si letra == 'a' se mostrará Primera del abecedario.

•Si letra == 'A' se mostrará tanto Mayúscula como primera del abecedario.

Se pueden especificar diferentes valores, separados por comas, detrás de un case. Go ejecutará el código para ese caso si la expresión a comprobar coincide con alguno de esos valores:

```
switch letra {
  case 'A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o', 'u':
    fmt.Println("Vocal")
  default:
    fmt.Println("Cualquier carácter menos una vocal")
}
```

La orden switch permite un análisis de patrones más complejo (hasta cierto punto, equivalente a varios if-else encadenados). Por ejemplo:

```
fmt.Print("Escribe un carácter: ")
var c int8
fmt.Scanf("%c", &c)

switch {
case c >= 'A' && c <= 'Z':
    fmt.Println("Letra Mayúscula")
case c>= 'a' && c <= 'z':
    fmt.Println("Letra Minúscula")
case c >= '0' && c <= '9':
    fmt.Println("Dígito")
default:
    fmt.Println("Ni letra ni dígito")
}
```

Observe que, en este caso, la orden switch no especifica ninguna variable, ya que la comprobación de esta se hace en el case.

3.2 ÓRDENES ITERATIVAS (BUCLES for)

Un bucle es un conjunto de órdenes que se repite. El bucle más sencillo que Go permite especificar es:

```
for {
  // instrucciones a repetir
}
```

El bucle anterior es un "bucle infinito". Si un programa llega a ese punto, el programa jamás continuará más allá del bucle for, a no ser que el bucle se rompa con la instrucción break:

```
for {
  fmt.Print("Salir? (s/n): ")
  var c rune
  fmt.Scanf("%c\n", &c)
  if c == 'S' || c == 's' {
    break
  }
}
```

El ejemplo anterior repite un bucle en el que el programa pide un carácter al usuario, y se repite hasta que el usuario introduce el carácter 'S' o 's' , momento en el que el bucle se rompe mediante la instrucción break.

La instrucción continue rompe el flujo de cada iteración de un for. En este caso, el flujo del programa salta de nuevo al inicio del bucle, sin ejecutar las instrucciones restantes de la iteración en que se invoca.

```
for {
  fmt.Print("Salir? (s/n): ")
  var c rune
  fmt.Scanf("%c\n", &c)
  if c == 'N' || c == 'n' {
    continue
  }
  if c == 'S' || c == 's' {

    break
  }
  fmt.Println("carácter no reconocido")
}
fmt.Println("adiós!")
```

El ejemplo anterior se comportaría de la siguiente manera:

Salir? (s/n): a
carácter no reconocido
Salir? (s/n): n
Salir? (s/n): s
adiós!

De acuerdo con el código del ejemplo:

- Si el usuario introduce 'N' o 'n' , se ejecutará la instrucción continue, por lo que la ejecución volverá al inicio del for.
- Si el usuario introduce 'S' o 's' , se ejecutará la instrucción break, por lo que el for finaliza y la ejecución continúa por el mensaje adiós!
- Solo en el caso de que el usuario introdujera cualquier otro carácter, se mostraría el mensaje carácter no reconocido .

A pesar de lo que promulgan algunas escuelas de programación de corte académico, las órdenes break y continue son totalmente válidas y aceptadas

en las convenciones sobre estilo de Go. Sin embargo, a menudo resulta más limpia y útil la forma condicional de for:

```
for <condición> {
  // instrucciones a ejecutar mientras
  // la condición sea cierta
}
```

Por ejemplo:

```
var c rune
for c != 'S' && c != 's' {
  fmt.Print("Salir? (s/n): ")
  fmt.Scanf("%c\n", &c)
}
```

El programa anterior repetirá el código dentro del bucle mientras el usuario no introduzca 'S' ni 's'.
El lector que esté familiarizado con otros lenguajes de programación, se habrá dado cuenta de que esta forma de for "condicional" suele llamarse while en otros lenguajes de programación. Los diseñadores de Go decidieron que el código resultaría mucho más simple y legible si cualquier bucle usaba la misma orden for.
Hay una tercera forma de describir un for; la más similar al bucle for de los demás lenguajes:

```
for <inicio>; <condición>; <actualización> {
  // órdenes que se ejecutarán mientras la condición
  // sea cierta
}
```

Este tipo de for:

1. Ejecuta la orden de inicio una sola vez, antes de ejecutar la primera

iteración.

2. Antes de cada iteración, se comprobará si la condición es cierta; en caso de que no lo sea, el bucle acabará.

3. Después de cada iteración, antes de volver a comprobar si la condición sigue siendo cierta, se ejecutará la orden de actualización.

Por ejemplo, el siguiente bucle for mostrará una cuenta del 1 al 10:

```
for
i := 1; i <= 10; i++ {
  fmt.Println(i)
}
```

4. La variable i se inicia al valor 1 (observe que el alcance de esta se limita al bucle).

5. Se muestra el valor de la variable i .

6. Se incrementa la variable i .

7. Se comprueba si el valor de i es menor o igual que 10. Si es el caso, se vuelve al paso 2. Si no es el caso, se sale del bucle for.

En el capítulo 6, sobre estructuras de datos lineales, se mostrará otro uso del bucle for, comúnmente conocido como for-each, que facilita el recorrido por los valores de diferentes colecciones de datos.

3.3 CONTEXTO Y OCULTACIÓN DE VARIABLES

En cada instrucción condicional o cada bucle, cada vez que se define un nuevo contexto entre un par de llaves { y } es posible definir nuevas variables cuya vida y visibilidad se limitarán a ese contexto:

```
func main() {
  a := 3
  if a == 3 {
    i := 1
    fmt.Println("'i' solo es visible en este contexto:", i)
  }
  fmt.Println("'a' es visible en todo 'main':", a)
}
```

Si, en el ejemplo anterior, intentara mostrar el contenido de la variable i desde fuera del bloque if donde ha sido definida, el compilador le mostraría un mensaje de error.

Una característica de Go, que puede ser útil pero también puede ser una fuente de problemas, es el poder definir nuevas variables con nombres que ya existen en los contextos más globales. A pesar de tener el mismo nombre, serán variables distintas. Este concepto es conocido como "ocultación" o, en inglés, *shadowing*.

Ejemplo:

```
a := 0
b := 0

if true {
  a := 1
  b = 1
  a++
  b++
}

fmt.Printf("a = %d, b = %d\n", a, b)
```

En un despiste, se podría pensar que la salida estándar de este programa sería a = 2, b = 2. Sin embargo, es a = 0, b = 2, ya que la variable a ha sido redeclarada y ocultada (eclipsada) dentro de la condición (observe el sutil detalle del operador de declaración := usado con a, frente al de asignación = usado con b). Por tanto, la variable a mostrada en fmt.Printf no ha sido modificada en ningún momento.

Capítulo 4
APUNTADORES

La memoria de un ordenador podría abstraerse, como si fuera un conjunto de cajones colocados en una gigantesca estantería. Cada cajón tiene un número identificativo único, que permite al ordenador referirse a él a la hora de ir a buscar o guardar datos. Dicho número identificativo se conoce como "dirección de memoria".

Los apuntadores (o punteros) son variables que no guardan valores útiles como tales, sino las direcciones de memoria donde se encuentran dichos valores. Su nombre hace referencia al hecho de que no guardan una variable sino que "apuntan" a su dirección.

En Go, los apuntadores tienen un tipo asociado, es decir, solo pueden apuntar a variables de un tipo en concreto: un puntero a int solo podrá guardar la dirección de memoria de una variable del tipo int, un puntero a bool solo podrá guardar la dirección de memoria de un bool, etc.

4.1 DEFINICIÓN DE UN APUNTADOR

Un apuntador se define como una variable, añadiendo un asterisco delante del tipo de datos al que este apuntador puede apuntar:

```
var pi *int    // apuntador a ints var pb *bool  // apuntador a bools
```

4.2 LA REFERENCIA A nil

Los apuntadores definidos en la sección anterior no han sido inicializados a ningún valor. Apuntan al valor nil, que podría interpretarse como "a ninguna parte".

El valor nil se puede utilizar tanto para reiniciar el valor de un apuntador "a nada" como para comprobar si un apuntador apunta a algún lugar válido:

```
var pi *int
pi = nil
if pi == nil {
  fmt.Print("¡No puedo hacer nada con este apuntador ")
  fmt.Println("porque no apunta a nada!")
}
```

Cuando un apuntador hace referencia a la dirección nil, este no se puede utilizar para leer o modificar valores apuntados, ya que dicho valor apuntado no existe. Si tratáramos de hacerlo, el sistema de memoria segura de Go abortaría la ejecución del programa, mostrando un error similar al siguiente:

```
panic: runtime error: invalid memory address or nil pointer dere-
ference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0
pc=0x49169f]
```

4.3 APUNTANDO HACIA UNA VARIABLE

El operador *ampersand* (&) delante de una variable retorna la dirección de memoria de esta. Este valor se puede asignar directamente a un puntero:

```
i := 10 var
p *int
p = &i
```

En el ejemplo anterior, el apuntador p apuntará a la variable i (Figura 4.1). El código anterior se puede abreviar de la siguiente manera:

```
i := 10
p := &i
```

Ya que &i retorna un valor del tipo *int, p será declarado como *int (apuntador a int) y desde el principio apuntará al int i.

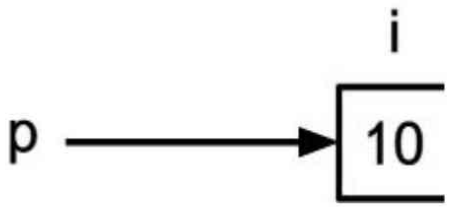


Figura 4.1 Apuntador *"p"* apuntando a la dirección de memoria de *"i"* (esquema visual).

4.4 LEYENDO O MODIFICANDO EL VALOR APUNTADO

A través de un apuntador, se puede leer o modificar el valor de la variable apuntada. El operador asterisco * delante del nombre del apuntador nos permitirá acceder al valor de la variable apuntada, tanto para su lectura como para su modificación:

```
i := 10
p := &i
a := *p
*p = 21
```

En el ejemplo anterior, donde el apuntador p apunta a la variable i, operando a través de este apuntador se está leyendo y modificando el valor de la variable i. La tercera línea (a := *p) tendría un resultado equivalente a a := i, y la cuarta línea (*p = 21) tendría un resultado equivalente a i = 21. La Figura 4.2 esquematiza la evolución de la memoria según las anteriores instrucciones se van ejecutando: qué variables se van creando y qué valores toman en cada momento.

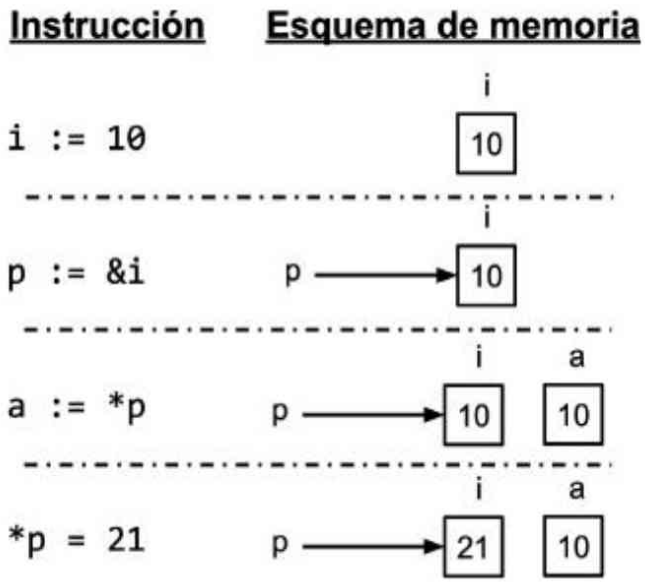


Figura 4.2 Uso del operador asterisco sobre punteros (esquema visual).

Viéndolo desde el punto de vista del ejemplo anterior, podría parecer que los apuntadores son un paso intermedio innecesario. Más adelante, este libro muestra la verdadera utilidad de los apuntadores cuando se usan para recorrer complejas estructuras de datos, para referirse a alguna de sus partes o para intercambiar datos con "funciones".

4.5 VALORES VERSUS REFERENCIAS

Las variables de Go, por defecto, son operadas "por valor". Esto significa que:

- El operador de asignación = **copia** el valor de la derecha hacia el valor de la variable de la izquierda. Tras la instrucción a = b, a y b serán dos variables con el mismo valor, pero no son la misma variable: ocupan zonas distintas de la memoria y la modificación de una no afecta al valor anterior de la otra.
- El operador de igualdad == entre dos variables resulta en true si ambas variables tienen un valor igual, aunque sean dos variables distintas.

Los punteros de Go nos permitirán operar "por referencia", es decir:

- El operador de asignación = apunta (o **referencia)** el apuntador de la izquierda hacia el apuntador de la derecha. Tras la instrucción p = &i , el apuntador p apunta hacia la misma dirección de memoria donde está i . Cualquier modificación de p afectará a tal valor y a cómo se accede a través de i.
- El operador de igualdad == entre dos apuntadores resulta en true si ambos apuntan a la misma dirección de memoria, o en false si apuntan a direcciones distintas (aunque esas direcciones contengan el mismo valor). Si se quisiera comparar la igualdad de dos valores apuntados por los apuntadores p1 y p2, se debería usar el operador asterisco para obtener el valor de ambos y, así, poder comparar valores en vez de direcciones:

```
i := 0
j := 0
p1 := &i
p2 := &j
if p1 == p2 {
  fmt.Println("p1 y p2 apuntan a la misma dirección")
} else {
```

```
  fmt.Println("p1 y p2 apuntan a direcciones distintas")
}
if *p1 == *p2 {
  fmt.Println("p1 y p2 apuntan a valores iguales")
} else {
  fmt.Println("p1 y p2 apuntan a valores distintos")
}
```

El ejemplo anterior mostraría la siguiente salida estándar:

p1 y p2 apuntan a direcciones distintas

p1 y p2 apuntan a valores iguales

Ya que p1 y p2 apuntan a dos variables con el mismo valor, pero en distintas zonas de memoria (Figura 4.3), tan solo mostraría el mensaje p1 y p2 apuntan a valores iguales, ya que apuntan a dos variables iguales, pero que no son la misma.

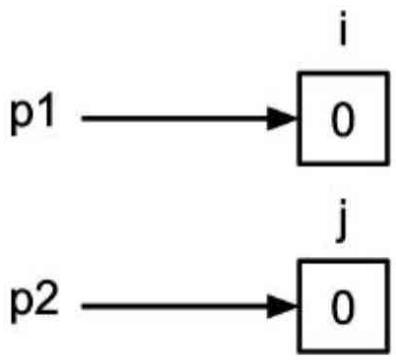


Figura 4.3 p1 y p2 apuntan a valores iguales, pero distintas variables (esquema visual).

La primera comprobación sería "por referencia", ya que se comparan direcciones de memoria; mientras que la segunda comprobación sería "por valor", ya que se comparan dos valores concretos.

Capítulo 5

FUNCIONES

Las funciones permiten reutilizar partes de código cuya ejecución puede invocarse desde diversos puntos del programa.

5.1 DEFINICIÓN E INVOCACIÓN

Un ejemplo sencillo de función se definiría de este modo:

func
<Nombre Función>() {
//
código a ejecutar en cada invocación

}

Por ejemplo, el siguiente código definiría una función llamada Hola que, al invocarse, muestra por pantalla el mensaje ¡Hola!:

func
Hola() {
fmt.Println("¡Hola!")
}

Las funciones se definen fuera del cuerpo de la función main, y suelen invocarse desde el código de las propias funciones, mediante el nombre de la función seguido de dos paréntesis ():

```
func main() {
  fmt.Println("Invocando una función:")
  Hola()
  fmt.Println("Invocando la misma función, otra vez:")
  Hola()
}
```

Si se añade la definición de la función Hola al mismo archivo que contiene la función main anterior y se ejecuta el programa, la salida estándar sería:

Invocando una función:
¡Hola!
Invocando la misma función, otra vez:

¡Hola!

Aunque una función ejecuta siempre el mismo código, se puede modificar ligeramente su comportamiento si se definen **argumentos** (o **parámetros)** entre los paréntesis de la cabecera de la función. Cada argumento consiste en un nombre seguido de un tipo (como si de una variable se tratara). Los diferentes argumentos se separan por comas.
Por ejemplo, se podría modificar el comportamiento de la anterior función Hola para especificar el nombre y apellidos de la persona a saludar mediante argumentos:

```
func
Hola(nombre string, apellido string) {
  fmt.Printf("¡Hola, %s %s!\n", nombre, apellido)
}
```

Para invocar una función con argumentos, entre los paréntesis de la invocación deberán situarse los valores para dichos argumentos, separados por comas, y ordenados según la definición en la cabecera:

```
func
main() {
  Hola("Marta", "García")
  Hola("Juan", "Martínez")
}
```

Salida estándar:

```
¡Hola, Marta García!
¡Hola, Juan Martínez!
```

Los valores a pasar como argumentos pueden ser valores literales, variables, u otras expresiones combinadas mediante operadores.

CONSEJO: Cuando en la definición de la cabecera de una función varios argumentos consecutivos son del mismo tipo, se puede omitir el tipo de los demás argumentos, excepto el último. La cabecera aconsejable de la función Hola sería:

```
func Hola(nombre, apellido string)
```

Go asumirá que tanto nombre como apellido son del tipo string.

5.2 RETORNO DE VALORES

Al igual que las funciones matemáticas, una función en Go puede devolver un valor, resultante de los cálculos o procesos que la función haga con sus argumentos. El tipo de retorno se declara al final de la cabecera de la función, cuando se cierran los paréntesis de los argumentos, y el retorno del valor resultante se especifica con la palabra return en el cuerpo de la función. Cuando el comando return se ejecuta, la función finaliza (no se ejecuta ninguna instrucción a continuación).

```
// Max retorna el mayor de los dos números recibidos
func Max(a, b int) int {
  if a > b {
    return a
  }
  return b
}
```

El ejemplo anterior retornará el primer argumento si este es mayor que el segundo, o el segundo argumento en caso contrario.

El valor retornado por una función se puede usar en medio de una expresión matemática, como si de una variable se tratara, o puede asignarse directamente a una variable:

```
m := Max(3, 5)          // m == 5

a := 10 + Max(m+2, 6)   // a == 17
```

5.3 RETORNO DE MÚLTIPLES VALORES

Una particularidad de Go es que sus funciones pueden retornar múltiples valores. Para ello, en vez de especificar un solo tipo de retorno, se especificarán varios tipos de retorno separados por comas.

En el siguiente ejemplo, la función MaxMin acepta dos argumentos y retorna dos valores. Primero, retornará el mayor de ambos valores; luego, retornará el menor. En la orden return, se deberán escribir ambos valores a retornar, separados por coma, y en orden según su significado (primero el máximo y luego el mínimo):

```
// MaxMin retorna dos números:
// Primero, el mayor de los argumentos
// Segundo, el menor de los argumentos
func MaxMin(a, b int) (int, int) {
  if a > b {
    return a, b
  }
  return b, a
}
```

Una función que retorna múltiples valores no se puede invocar en medio de una expresión matemática. Se deben recoger los valores especificando múltiples variables, separadas por comas, a la izquierda del operador de asignación (=) o de declaración (:=),

max, min := MaxMin(34, 55)

En el anterior ejemplo, el primer valor retornado se guardará en la variable max y el segundo valor retornado se guardará en la variable min.

TRUCO: La asignación múltiple de valores se puede realizar sin invocar ninguna función. Por ejemplo, el comando a, b = b, a intercambiaría el valor de dos variables.

5.4 RETORNO DE MÚLTIPLES VALORES NOMBRADOS

En funciones largas y complejas, por cuestiones de legibilidad puede convenir poner un nombre a cada uno de los valores de retorno y, a partir de ahí, usarlos como si fueran variables. En este caso, la orden return no necesita que se especifiquen los valores de retorno:

```
// MaxMin retorna dos números:
// Primero, el mayor de los argumentos
// Segundo, el menor de los argumentos
func MaxMin(a, b int) (max int, min int) {
  if a > b {
    max = a
    min = b
  } else {
    min = a
    max = b
  }
  return
}
```

No obstante, la invocación de la función no cambia:

```
mx, mn := MaxMin(3, 5)
```

5.5 EL IDENTIFICADOR VACÍO

El identificador vacío (representado por un guion bajo, _) permite descartar los valores retornados por una función que no se van a necesitar. Aquellos valores que no se van a usar se deben asignar a este identificador vacío, ya que **Go no permite declarar variables que no se van a usar** (el compilador retornaría error).

Considerando la función MaxMin del ejemplo anterior, el siguiente código:

```
m, _ := MaxMin(3, 6)
```

guardaría en la variable m el primer valor retornado por la función, y descartaría el segundo valor.

5.6 PASO POR VALOR VS. PASO POR REFERENCIA

Los argumentos en Go, por defecto, se pasan **por valor**, lo cual quiere decir que las variables o valores pasados en la invocación **se copian** en el espacio de memoria reservado a los argumentos dentro de la función.

La siguiente función, Incrementa, incrementa en 1 el valor pasado como argumento:

```
func Incrementa(a int) {
  a++
}
func main() {
  a := 3
  fmt.Println("a =", a)
  Incrementa(a)
  fmt.Println("a =", a)
}
```

Sin embargo, el anterior programa mostraría:

a = 3

a = 3

Esto es así porque Incrementa no trabaja con la misma variable a de la función main, sino con una copia de esta (Figura 5.1).

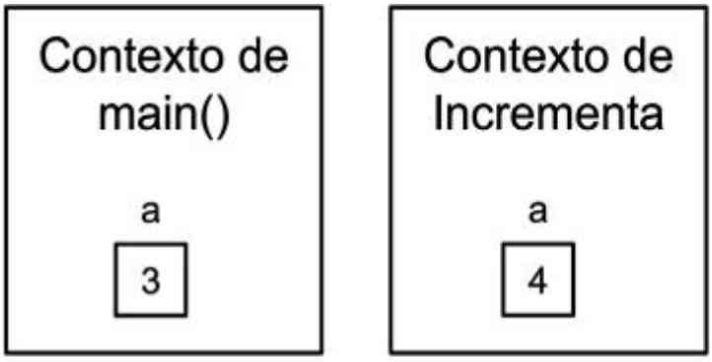


Figura 5.1 El incremento se realiza sobre una copia de la variable original

(esquema visual).

Cuando se pretende que una función trabaje sobre la misma zona de memoria que la variable que se le pasa en la invocación, se debe usar el paso de parámetros **por referencia.** En el caso que nos ocupa, la variable Incrementa no debería recibir una variable del tipo int sino un apuntador del tipo *int, y trabajar con operadores de punteros:

```
func Incrementa(a *int) {
  *a++
}
func main() {
  a := 3
  fmt.Println("a =", a)
  Incrementa(&a)

  fmt.Println("a =", a)
}
```

Salida estándar del anterior ejemplo:

a = 3

a = 4

A recalcar:

- Incrementa recibe un puntero a *int en vez de un simple int. Por tanto, la operación a++ (incrementa el valor de a) debe cambiarse por *a++ (incrementa el valor de la variable apuntada por a, Figura 5.2).
- La invocación a Incrementa no puede recibir directamente a, que es un int, sino &a (la dirección de a), que es un apuntador *int.

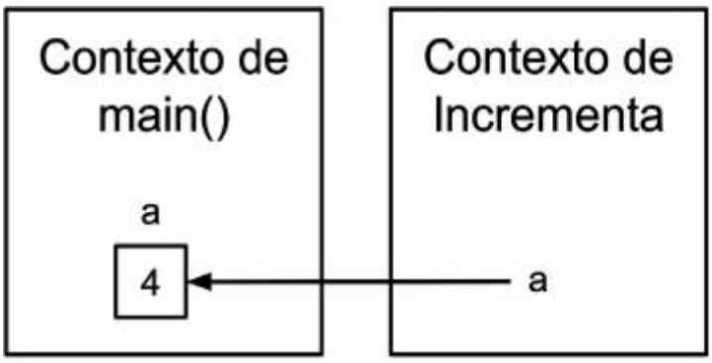


Figura 5.2 El incremento se realiza sobre una referencia hacia la variable original (esquema visual).

5.7 LITERALES DE FUNCIÓN

En Go, la signatura de una función (la definición de sus argumentos y tipos de retorno) es un tipo de dato por sí mismo. Esto significa que se puede declarar una variable o un argumento del tipo función. Por ejemplo, la siguiente variable apuntaría (las funciones son tratadas **por referencia)** a una función que no recibe ningún argumento y retorna un número entero:

```
var generador func() int
```

A esta variable se le puede asignar un literal de función en cualquier punto del programa, e invocar dicha función a través de la variable generador , como si fuera una función común:

```
func main() {
  var generador func() int

  generador = func() int {
    return 0
  }
  fmt.Println("generador de ceros:", generador())
}
```

Mostraría: generador de ceros: 0.
Los literales también pueden acceder a valores que están fuera de su ámbito:

```
contador := 0
generador = func() int {
    contador++
    return contador
}
fmt.Println("generador contador:",
                generador(), generador(), generador())
```

Mostraría: generador contador: 1, 2, 3.

Una potente característica de los literales es que una variable del tipo "signatura de función" acepta una referencia a cualquier función existente, siempre que esta comparta la misma signatura:

```
func Suma(a, b int) int {
  return a + b
}
func Multiplica(a, b int) int {
  return a * b
}
func main() {
  var operador func(int, int) int
  operador = Suma
  fmt.Println("suma =", operador(3, 4))
  operador = Multiplica
  fmt.Println("multiplica =", operador(3, 4))
}
```

Mostraría en pantalla:

suma = 7

multiplica = 12

En algunas situaciones muy concretas (véase el capítulo 15 sobre Gorrutinas, o la cláusula defer en el capítulo 13 sobre gestión de errores), puede ser útil invocar a un literal justo después de su declaración, sin necesidad de asignarlo a una variable:

```
func() {
  fmt.Println("Esto se invocará")
  fmt.Println("Directamente, pero en otro contexto")
}()
```

Observe los paréntesis de invocación () justo después de la declaración del literal.

INFORMACIÓN: Los literales de función también pueden recibir el nombre de *lambdas* o “clausuras” (en inglés, *closures).*

5.8 OTRAS CONSIDERACIONES

Hay algunos otros aspectos a considerar cuando se diseñan funciones en Go:

•Al igual que para la asignación de variables, no hay conversión de tipos implícita en el paso de argumentos. Por ejemplo, si una función acepta un argumento del tipo int64 y la variable que contiene el valor a pasar es del tipo int32, se tendrá que hacer la conversión explícita:

```
func EsPar(v int64) bool {
    return v%2 == 0
}
func main() {
    var val int32 = 123
    fmt.Println("Par?", EsPar(int64(val)))
}
```

•A diferencia de otros lenguajes, Go no soporta "sobrecarga" de funciones (diferentes funciones con el mismo nombre pero diferentes argumentos). Por ejemplo, el paquete rand proporciona funciones con funcionalidad similar pero con argumentos y tipos de retorno diferentes, y esta limitación de Go obliga a proporcionar nombres diferentes para cada función.

–func Int() int retorna un int aleatorio cualquiera.

–func Intn(n int) int retorna un int aleatorio entre 0 y n-1, ambos incluidos.

–func Int31n(n int32) int32 es análoga a Intn pero acepta y retorna int32 .

–func Int63n(n int64) int64 es análoga a Intn pero acepta y retorna int64 .

Capítulo 6
ESTRUCTURAS DE DATOS LINEALES

Hasta ahora, este libro ha mostrado variables que pueden guardar un valor concreto. Cuantos más valores sean necesarios, más variables habrá que definir. Sin embargo, este enfoque es inviable cuando un programa debe manejar grandes conjuntos de datos.

Los lenguajes de programación permiten crear agrupaciones de datos de un mismo tipo, y acceder a cada agrupación mediante el nombre que se le da y el índice que cada dato ocupa en ella. Estas agrupaciones son "lineales" porque los datos están ordenados según una sola dimensión: la de su índice.

Go permite definir dos tipos de agrupaciones lineales: los **vectores**, similares al concepto de vector de otros lenguajes de programación, y las **porciones**, que definen una vista sobre otros vectores y proporcionan mayor dinamismo y manejabilidad.

6.1 VECTORES

Un vector (comúnmente conocido por su denominación en inglés, *array)* es una agrupación de valores del mismo tipo que se guardan en una zona contigua de la memoria. Se pueden seleccionar y se puede acceder a ellos individualmente mediante el nombre del vector y un índice que identifica de manera única cada valor del vector.

En Go, una variable que contiene un vector se definirá mediante un nombre, un tamaño entre corchetes y el tipo de datos que contiene. Por ejemplo:

```
var arr [5]int
```

La definición anterior creará un vector de 5 números enteros, accesibles mediante los índices [0] a [4], con el valor cero por defecto.

Para declarar un vector de 5 enteros, con valores por defecto:

```
arr := [5]int{1, 2, 3, 4, 5}
```

El indicador de tamaño especial ... permite que el compilador infiera el tamaño del vector, según la cantidad de números que se indican en la declaración:

```
arr := [...]int{1, 2, 3, 4, 5}
```

Para acceder a los elementos de un vector, se utiliza el nombre del vector, así como un índice entre corchetes. El elemento al que se accede equivaldrá a una variable del tipo contenido en el vector:

```
arr := [3]int{1, 2, 3}
suma := arr[0] //
lee un int del vector

arr[2] = 0    //
escribe en un int del vector
```

Igualmente, se pueden definir matrices, es decir, vectores de múltiples dimensiones:

```
var tableroAjedrez [8][8]uint8
```

El uso de los vectores en Go está limitado a algunos casos específicos, ya que imponen las siguientes restricciones:

- El tamaño de un vector es parte del tipo. Eso significa que las siguientes variables v1 y v2 tienen diferente tipo:

```
v1 := [3]int{5, 4, 3} var v2 [10]int
```

y, por tanto, el compilador retornaría error al intentar hacer v2 = v1. Eso también significa que un vector no puede incrementar su tamaño, ya que cambiaría su tipo.

- Los operadores de asignación (=) o de declaración (:=) **copian por valor** los contenidos del vector a la derecha del operador hacia el vector a la izquierda. Esto se debe tener en cuenta porque el operador de asignación = o el paso de argumentos a una función serán costosos en tiempo de ejecución si el vector es grande. Es necesario considerar que el operador de igualdad == retornará true si los vectores a ambos lados del operador son iguales (comparando elemento por elemento).

Debido a las anteriores restricciones, los vectores en Go se utilizan para casos muy específicos. Para tratar colecciones de valores contiguos e indexados, Go provee una estructura que también puede indizarse, que es más sencilla y está indicada para la mayoría de casos: las **porciones** —o, como comúnmente se conocen en la comunidad Go, dominada por los anglicismos, *slices*—.

6.2 PORCIONES

Las porciones (slices) envuelven vectores (o partes de vectores) con una interfaz más flexible y genérica. Se podrían considerar **referencias** que en todo momento apuntan a un vector, y que permiten acceder a sus elementos también a través de un índice.

Sin embargo, las porciones presentan las siguientes diferencias con respecto a los vectores:

- Si asignas una variable del tipo porción a otra del mismo tipo (operadores = o :=), los valores de la porción no se copian, sino que ambas porciones apuntarán al mismo vector (las variables son **referencias** a porciones).
- Si se comparan dos porciones con el operador de igualdad == , la operación solo retornará true si ambas porciones apuntan al mismo vector, y false si apuntan a vectores distintos (aunque fueran iguales, elemento a elemento).
- El tamaño de una porción no es parte del tipo. El tipo de "porción de enteros" será definido como []int, y cualquier variable de este tipo podrá ser asignada a cualquier otra variable del mismo tipo, sea cual sea su tamaño.
- La corrección del acceso a los elementos de la porción (por ejemplo, comprobar si el índice está dentro de la longitud de la porción) se realizará en tiempo de ejecución del programa. Esto significa que un programa puede compilar correctamente y fallar durante su ejecución, si se intenta acceder a un índice inexistente.
- La **longitud** de una porción, que representa el número de elementos que esta contiene e indexa, no es un número fijo. Se pueden añadir nuevos elementos con la función append, incorporada en el lenguaje Go.
- La **capacidad** de la porción, que es igual o mayor a su longitud,

representa el espacio reservado en el vector subyacente. A medida que una porción crece, la longitud se irá incrementando hasta llegar a la capacidad total. A partir de ahí, Go reservará espacio para un vector con más capacidad, en el que poder seguir insertando valores. Antes, copiará todos los valores del anterior en el nuevo vector.

- A partir de una porción existente, se pueden crear nuevas "subporciones" (también conocidas como "vistas"), lo cual permite ver una porción grande como múltiples porciones más pequeñas (cuyo índice o apuntará al inicio de la subporción, no al de la porción padre).

Se puede decir que, mientras los vectores operan **por valor**, las porciones operan **por referencia.** No obstante, una porción, internamente, no se limita a un simple puntero, sino que debe guardar la siguiente información en todo momento (Figura 6.1):

- **Inicio de la porción.** Representado por un apuntador al inicio de la porción en el vector subyacente.
- **Longitud de la porción.** Representada por un número entero.
- **Capacidad de la porción.** Representada por otro número entero.

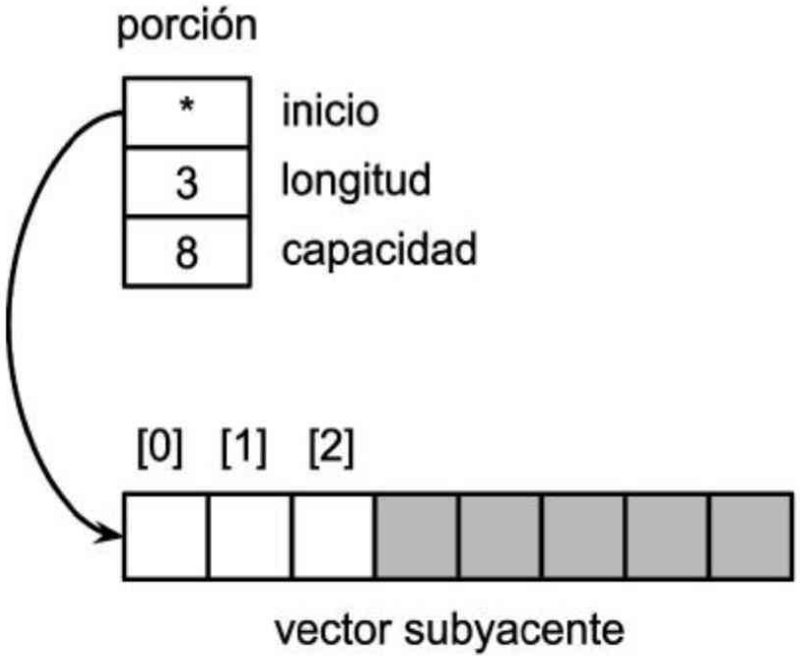


Figura 6.1 Estructura en memoria de una porción.

6.3 DECLARANDO VARIABLES A PORCIONES

Una porción se define de manera similar a un vector, pero sin especificar el tamaño entre corchetes:

```
var r []int
```

Esto declarará una variable r del tipo "porción de enteros"; al no estar inicializada, apunta a nil.
Para declarar una porción apuntando a algunos valores ya inicializados:

```
s := []int{1, 2, 3, 4}
```

En el ejemplo anterior, la longitud de la porción será 4 y, al igual que sucede con los vectores, se puede acceder a los distintos elementos con los índices: s[0], s[1], s[2] and s[3]. Cualquier acceso a un índice superior a 3 sería detectado en tiempo de ejecución, interrumpiría el programa y devolvería el siguiente mensaje de error:

```
panic: runtime error: index out of range
```

Observe también la diferencia al declarar un vector, que requiere que entre los corchetes se establezca la longitud del vector o el indicador ...:

```
p := []int{1, 2, 3}      // porción
v1 := [3]int{1, 2, 3}    // vector
v2 := [...]int{1, 2, 3} //vector
```

6.4 AÑADIR ELEMENTOS A UNA PORCIÓN. FUNCIÓN append

La función append añade nuevos elementos al final de una porción (incrementando su longitud). Es una función propia del lenguaje, por lo que no tendrá que hacer import de ningún paquete.

La manera genérica de usarla es la siguiente:

<porción> = append(<porción>, <elemento>, <elemento>)

donde el número de elementos a añadir es variable. Por ejemplo:

```
var p []int            // longitud: 0 (nil)
p = append(p, 1, 2, 3) // longitud: 3
p = append(p, 6)       // longitud: 4
```

Es necesario observar que la función append recibe, como primer argumento, la porción donde se van a añadir los números. Además, append retorna una variable del tipo porción (que se vuelve a asignar a la variable original). Esta variable, generalmente, será la misma porción que se ha pasado por argumento, excepto en el caso de que la porción argumento haya rebasado su capacidad y no se puedan añadir más elementos. En este caso:

1. Se reserva espacio en memoria para una nueva porción, con una capacidad que varía en función de la porción original:
   - –Si los elementos a añadir son tantos como elementos tiene la porción original, la capacidad de la nueva porción será la suma de ambas longitudes.
   - –Si la porción original tiene una capacidad menor a 1024 elementos, la capacidad de la nueva porción dobla a la de la porción original.
   - –En otro caso, la nueva porción tendrá un 25 % más de capacidad que la porción original.
2. Los elementos de la porción original (que se pasa por argumento) se

copian en la nueva porción.

3. Se añaden a la nueva porción los elementos a añadir mediante la función append .

4. Se retorna la nueva porción.

TRUCO: Si append recibe una porción nil como argumento, la función creará una nueva porción y la retornará. Para porciones de longitud variable que puedan estar vacías, declararlas con un valor inicial nil ahorrará una reserva de memoria a nuestro programa:

6.5 MEDIR DIMENSIONES CON len Y cap

La función len(<porción>) retorna la longitud de una porción: cuántos elementos contiene en un momento dado.

La función cap(<porción>) retorna la capacidad total de una función: cuántos elementos podría albergar hasta que fuera necesario reservar espacio para otra porción con mayor capacidad.

El siguiente código ilustra cómo la longitud y la capacidad de una función varían cuando se añaden nuevos elementos al final, de acuerdo con las reglas explicadas en el apartado anterior:

```
var
sl []int
fmt.Printf("longitud %v. capacidad %v\n", len(sl), cap(sl))
sl = append(sl, 1, 2, 3, 4)
fmt.Printf("longitud %v. capacidad %v\n", len(sl), cap(sl))
sl = append(sl, 5)
fmt.Printf("longitud %v. capacidad %v\n", len(sl), cap(sl))
```

Salida en pantalla:

```
longitud 0. capacidad 0
longitud 4. capacidad 4
longitud 5. capacidad 8
```

Observaciones:

- La referencia sl inicialmente apunta a nil, ya que solo se declara la variable, sin inicialización alguna.
- Cuando se invoca a append por primera vez, se crea una porción en la que caben los sucesivos argumentos, se copian allí y retorna una nueva porción a asignar a la referencia sl .
- Como la longitud de la primera porción es igual a su capacidad total,

cuando se hace un segundo append se crea una nueva porción, con el doble de capacidad, se copian todos los elementos de la porción original y se añade el quinto elemento al final.

6.6 CONTROLAR EL TAMAÑO INICIAL CON make

Añadir elementos a una porción puede ser costoso si esta necesita ir creciendo y copiando valores entre porciones de distinta capacidad, como se describió anteriormente.

Si el programador conoce de antemano los requerimientos de capacidad de una porción, la función make permite mayor flexibilidad para crear porciones de una longitud y una capacidad dadas:

<porción> := make([]<tipo>, <longitud> [, <capacidad>])

La capacidad de la porción es opcional, pero la convención común es crear porciones de longitud cero con una capacidad específica, y luego agregar los elementos. Por ejemplo, el siguiente comando reservaría en memoria una porción vacía, con capacidad para 2048 números float32:

```
altaCapacidad := make([]float32, 0, 2048)
```

6.7 COPIA DE PORCIONES CON copy

Para copiar un vector, basta con asignar un vector a una nueva variable del mismo tipo, ya que estos trabajan por valor. Como las porciones funcionan por referencia, esto no es posible, ya que tendríamos dos variables apuntado a los mismos datos.

Cuando se quiera hacer un duplicado o una copia de una porción, Go provee la función copy:

```
func copy(destino, fuente []T) int
```

Dicha función copia los datos de la porción fuente a la porción destino, y retorna el número de elementos copiados.

Si fuente y destino tienen longitudes diferentes (cualesquiera que sean sus respectivas capacidades), solo se copiarán tantos elementos como tenga la más corta de las porciones.

Ejemplo de uso:

```
original := []int{1, 2, 3, 4, 5}
copia := make([]int, len(original))

n := copy(copia, original)
fmt.Println(n, "números copiados:", copia)
```

Salida por pantalla:

```
5 números copiados: [1 2 3 4 5]
```

6.8 USO DE PORCIONES EN FUNCIONES

Como para cualquier otro tipo de dato, las funciones pueden retornar vectores y porciones, así como aceptarlos como argumento.

Por cuestiones de eficiencia, se suele trabajar con porciones y no con vectores, ya que estos, al funcionar por valor, requerirán copiar grandes cantidades de memoria en cada invocación a la función.

No obstante, cuando en funciones se trabaja con porciones hay que tener en cuenta que, al trabajar mediante referencias, las modificaciones que se pudieran hacer a la porción original dentro de una función serán visibles desde la porción original que se pasa como argumento.

El siguiente código ilustra la diferencia entre el paso de vectores y porciones como argumento:

```
func Cero(vec [3]int, porc []int) {
  vec[0] = 0
  if len(porc) > 0 {
    porc[0] = 0
  }
}
func main() {
  v := [3]int{1, 2, 3}
  p := []int{1, 2, 3}
  Cero(v, p)
  fmt.Println("vector:", v)
  fmt.Println("porción:", p)
}
```

La función Cero recibe tanto un vector como una porción, y pone a cero el primer elemento de estos. Como ese vector siempre tendrá 3 —y solo 3— elementos, la orden vec[0] se puede ejecutar con total seguridad. Como no se puede asegurar que una porción tenga una longitud concreta (ni siquiera un

solo elemento), es aconsejable hacer antes la comprobación if len(porc) > 0, para evitar que el programa falle en tiempo de ejecución.
Tras invocar la función Cero con un vector y una porción de la misma longitud e iguales elementos, la salida por pantalla sería:

```
vector:  [1 2 3]
porción:  [0 2 3]
```

Se puede observar que el vector original no ha cambiado, ya que el paso de argumentos es por valor, y la función Cero trabaja con una copia del vector original. Sin embargo, la porción sí ha cambiado, puesto que el paso de argumentos es por referencia y la función Cero trabaja con una referencia a la porción original (aunque el código no esté usando sintaxis de punteros).

6.9 RECORRIENDO VECTORES Y PORCIONES

El lector que venga de otros lenguajes de programación clásicos, como C, estará tentado de recorrer un vector o porción de la siguiente manera:

```
ciudades := []string{"Tokyo", "Lleida", "Paris", "Madrid"}
for i := 0; i < len(ciudades); i++ {
  fmt.Printf("[%d] %s\n", i, ciudades[i])
}
```

Efectivamente, el código funcionaría correctamente, y mostraría lo siguiente:

[0] Tokyo
[1] Lleida
[2] Paris
[3] Madrid

Sin embargo, hay una manera de recorrer vectores y porciones que no solo es más clara sino también más óptima (ya que el compilador puede realizar algunas optimizaciones sobre esta):

```
for <índice>, <valor> := range <porcion> {
    // ...
}
```

Dicha estructura iterativa repetirá el código dentro del bloque for, accederá secuencialmente a cada elemento de la porción o vector, asignará el índice del elemento a la variable usada como índice y copiará el elemento correspondiente en la variable usada como valor.

La manera óptima y recomendada de recorrer la porción de ciudades del ejemplo anterior sería:

```
ciudades := []string{"Tokyo", "Lleida", "Paris", "Madrid"}
for i, ciudad := range ciudades {
  fmt.Printf("[%d] %s\n", i, ciudad)
}
```

Si no se necesita usar alguna de las variables índice o valor, esta se puede asignar al identificador vacío:

```
for _, ciudad := range ciudades {
  fmt.Println(ciudad)
}
```

6.10 CREANDO "VISTAS" DESDE LAS PORCIONES

Se puede crear una nueva porción a partir de una porción existente, especificando entre corchetes un rango de índices, separados por dos puntos (:):

<porción>[<inicio> : <fin>]

El índice de **inicio** es inclusivo (la nueva porción incluirá el elemento en esta posición). Si no se especifica, el índice de inicio por defecto es 0.

El índice de **fin** queda excluido (la nueva porción no incluirá el elemento en esta posición). Si se omite, el índice final es la longitud de la porción.

Es importante recalcar que:

- Una vista **no copia** el contenido de la porción base, sino que tanto la base como la vista operarán sobre la misma zona de memoria. Los cambios que se hagan a través de una vista serán visibles a través de la porción base, y viceversa.
- El índice 0 de una vista corresponderá al índice de la base a partir del cual se ha definido la vista. Por ejemplo, tras la operación vista := base[3:] , el elemento vista[0] corresponderá a base[3] . La longitud y capacidad de la porción vista también se verá recortada (Figura 6.2).

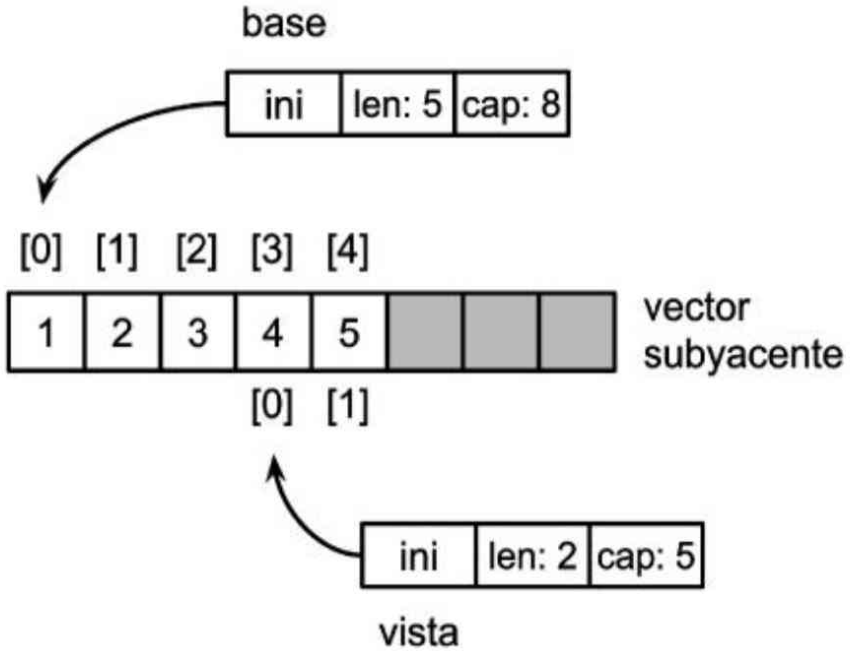


Figura 6.2 Dos porciones compartiendo una misma zona de memoria.

El siguiente código ilustra el comportamiento de las vistas de porciones:

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

vista1 := base[1:3] // incluye elementos 1 y 2
fmt.Println("vista 1:", vista1)
vista1[0] = 2 // equivalente a base[1] = 2
fmt.Println("base:", base)

vista2 := base[2:] // desde índice 2 hasta el final
fmt.Println("vista 2:", vista2)

vista3 := base[:3] // desde el inicio hasta indice 2
fmt.Println("vista 3:", vista3)

vista4 := base[:] // vista de inicio a fin
fmt.Println("vista 4:", vista4)
```

Salida:

```
base: [1 0 3 4 5]
vista 1: [0 3]
base: [1 2 3 4 5]
vista 2: [3 4 5]
vista 3: [1 2 3]
vista 4: [1 2 3 4 5]
```

En el ejemplo anterior, todas las vistas comparten memoria con la porción base, por lo que las operaciones a través de unas serán visibles en las otras

(Figura 6.3).

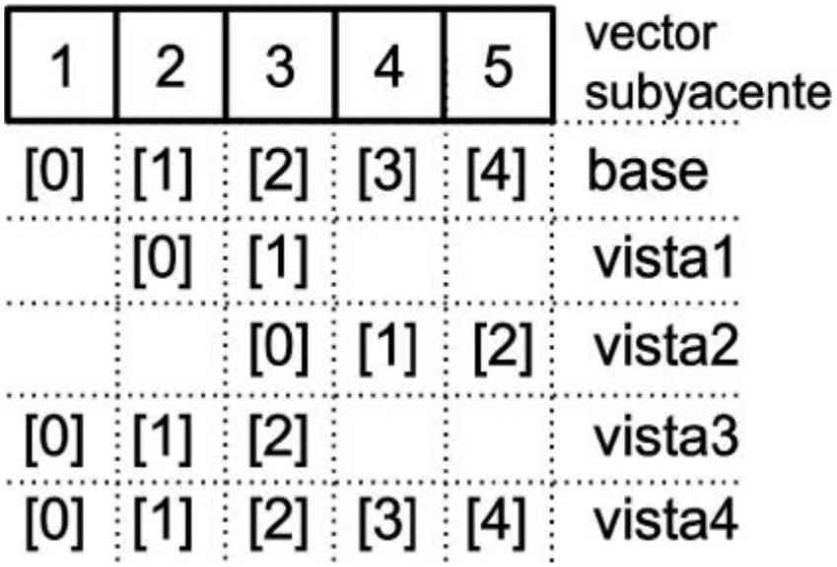


Figura 6.3 Múltiples vistas de porción sobre una zona de memoria común.

6.11 FUNCIONES CON NÚMERO VARIABLE DE ARGUMENTOS

Sin haber hecho hincapié en ello, este libro ha mostrado algunas funciones cuyo número de argumentos no está fijado, sino que puede variar. Por ejemplo, fmt.Println puede recibir diversos argumentos separados por comas. La función append introducida en este tema admite, además de una porción, un número variable de elementos a añadir.

Go permite definir funciones con un número variable de argumentos si en la cabecera de la función, donde se declara un argumento, el tipo de un argumento dado se prefija con tres puntos seguidos (...).

Por ejemplo:

func Suma(n ...int) int

La cabecera Suma define una función que acepta un número variable de argumentos; se puede invocar como Suma(), Suma(2, 3) o Suma(1, 2, 3, 4, 5, 6).

Para trabajar con este tipo de argumentos variables desde el cuerpo de la función, el argumento de "número variable" se utiliza como si de una porción se tratara: se pueden usar índices, recorrer mediante range, etc.

```
func Suma(n ...int) int {
  total := 0
  for _, s := range n {
    total += s
  }
  return total
}
```

Si se desea invocar a una función con número variable de argumentos, donde los argumentos están contenidos en otro vector o porción, es necesario

utilizar el **operador difusor** (...).

6.12 EL OPERADOR DIFUSOR

Si se pretendiera invocar la anterior función Suma sobre unos argumentos contenidos en un vector o porción, no se puede usar directamente esta porción, sino que se debe sufijar con el operador difusor, que consiste también en tres puntos: .... Por ejemplo:

```
valores := []int{1, 2, 3, 4, 5, 6}
total := Suma(valores...)
```

Se puede usar el operador difusor para concatenar dos porciones:

```
s1 := []int{1, 2, 3}
s2 := []int{4, 5, 6}
s1 = append(s, s2...)
fmt.Println("concatenación =", s1)
```

Salida:

```
concatenación = [1 2 3 4 5 6]
```

Capítulo 7
CADENAS DE TEXTO

Go provee soporte para cadenas de texto mediante el tipo de datos string. Internamente es una secuencia lineal de bytes que codifican un texto según el estándar **UTF-8:** los caracteres occidentales, así como los símbolos más comunes, suelen ocupar un byte (basándose en el antiguo estándar **ASCII);** y caracteres con tildes u otras decoraciones, así como otros caracteres no anglosajones, ocuparán rangos que van de uno a 4 bytes.

```
str1 := "hola mundo"
str2 := "hola 世界"
```

Como recordatorio de lo que brevemente se introdujo al inicio del libro, se pueden definir cadenas literalmente con dos tipos de delimitador:

- La **comilla doble recta** ("), que requiere que algunos caracteres especiales se deban eludir con el carácter de contrabarra (por ejemplo, \n para una nueva línea, o \" para una comilla doble que no deba confundirse con el delimitador de final de string). Requiere que una cadena literal sea definida en una única línea.

  Por ejemplo:

  fmt.Print("\"entrecomillado\"\nnueva línea")

  Mostraría en pantalla:

  "entrecomillado"
  nueva línea

- El **acento grave** (`) permite escribir cadenas de texto que ocupen múltiples líneas y no necesiten contrabarras para eludir algunos caracteres especiales. Por ejemplo:

```
fmt.Println(` Este texto se imprimirá
en cuatro líneas diferentes
y no necesitas eludir carácteres
como \, " o el salto de línea`)
```

7.1 DIFERENCIAS CON PORCIONES Y VECTORES

El tipo de dato string, aunque pueda ser visto como una secuencia lineal de datos y permita algunas operaciones comunes a estas (como indexado o creación de vistas), tiene su propio funcionamiento interno.

A diferencia de vectores y porciones, string es un tipo de dato **inmutable.** No se puede modificar el contenido de un string. Cualquier operación con cadenas de texto resultará en un nuevo string, copia del original, que incorporará los añadidos o modificaciones.

Es un tipo de dato que funciona **por referencia**, e internamente está formado por un apuntador al inicio de los datos y un número entero que indica la longitud. Esto significa que cualquier paso de argumentos o asignación de variable consistirá en copiar referencias al mismo string, pero nunca se copiará el contenido.

Sin embargo, la **comparación** de cadenas con el operador de igualdad == será carácter a carácter, como si de un vector por valor se tratara.

Además del operador de igualdad o diferencia !=, las cadenas se pueden comparar con los operadores <, <=, >, >=, que compararán la precedencia de dos cadenas según el orden de sus elementos más a la izquierda en el código UTF-8. Por ejemplo, la comparación "AAZ" < “AAa” retornará true, al ser los dos primeros caracteres iguales, y tener Z un código UTF-8 menor que a.

A pesar de ser un tipo de dato por referencia, el tipo string no puede apuntar a nil (al contrario que las porciones). Cuando se define una variable del tipo string y no se le asigna ningún valor, esta contendrá el string vacío "".

El siguiente código:

```
var nombre string

if nombre == "" {
    fmt.Println("nombre no tiene valor")
}
```

mostraría en pantalla nombre no tiene valor.

7.2 OBTENIENDO LA LONGITUD DE UN string

La función len, aplicada a un string, retorna la longitud de una cadena **en bytes.** Esto significa que len no devolverá siempre el número de caracteres de una cadena, sino los bytes que se requieren para su codificación UTF-8. Por ejemplo:

```
str1 := "hola mundo"
fmt.Printf("%q mide %v\n", str1, len(str1))

str2 := "hola 世界"
fmt.Printf("%q mide %v\n", str2, len(str2))
```

Mostrará:

```
"hola mundo" mide 10
"hola 世界" mide 11
```

La segunda frase, pese a ser más corta, retorna una longitud superior. Esto es debido a que los carácteres *kanji* asiáticos requieren varios bytes para ser codificados.

Las razones para el uso de UTF-8 son varias. La primera es que en la era de Internet y las aplicaciones multilenguaje es imprescindible poder representar los cientos de miles de símbolos y caracteres de las diversas lenguas (vivas y muertas) que el ser humano conoce.

La segunda razón es que todo este enorme número de caracteres no puede codificarse en un simple byte. Por ello, Go almacena un carácter individual con el tipo de dato básico rune, que ocupa 4 bytes. Sin embargo, si las cadenas de caracteres fueran meros vectores o porciones de []rune, se desperdiciaría mucho espacio en memoria, ya que la mayoría de caracteres de uso común podrían codificarse en un solo byte, y se usarían bytes extra solo para aquellos caracteres específicos de lenguas “no-anglosajonas” (recordemos que

Go fue creado por una multinacional americana).

Este es el estándar de codificación UTF-8, donde cada carácter puede ocupar de 1 a 4 bytes, y es el que Go utiliza internamente para organizar los bytes de un string.

Si en vez de la longitud en bytes de una cadena se pretende conocer la longitud en runas (en caracteres), Go proporciona la función utf8.RuneCountInString(<cadena>):

```
c := "hola 世界"
fmt.Printf("%q tiene %v runas\n", c,
                    utf8.RuneCountInString(str2))
```

Salida:

```
"hola 世界" tiene 7 runas
```

7.3 DE string A PORCIÓN

Un string permite leer sus bytes individualmente a través de un índice, o crear vistas de un string con un rango de índices. Sin embargo, estos accesos solo se permiten para la lectura de datos, nunca para la modificación, ya que el tipo string es inmutable.

Si se necesita modificar el contenido de un string, primero se debe convertir dicho string a una porción de byte o de rune, mediante la conversión explícita []byte("cadena") o []rune("cadena").

Las porciones de byte guardarán los datos de la cadena codificados en UTF-8. Las porciones de rune los guardarán un rune (4 bytes) por carácter, por lo que una porción de rune ocupará hasta 4 veces más memoria que el string original, pero nos facilitará el manejo de caracteres "no-anglosajones".

Cuando las mencionadas porciones de byte o rune sean modificadas, expandidas o recortadas mediante las operaciones comunes de porciones, estas se pueden convertir de vuelta a una cadena inmutable mediante la conversión explícita string(porcion).

Toda la tarea de codificación/decodificación la realiza Go internamente, de manera transparente para el programador:

```
cad := "una cadena"

// convirtiendo de string a porción
bytes := []byte(cad)
```

```
runas := []rune(cad)

// modificando el contenido de las porciones
bytes[0] = 'U'
runas = append(runas, '!')

// convirtiendo de nuevo de porción a cadena
str1 := string(bytes)
str2 := string(runas)

fmt.Println(str1)
fmt.Println(str2)
```

El anterior código crearía dos porciones a partir del original "una cadena". Modifica el primer carácter de una, añade un carácter al final de la otra y las convierte de vuelta a string, mostrando por pantalla:

Una cadena
una cadena!

Se puede asignar o añadir cualquier carácter a una porción de rune. Sin embargo, las porciones de byte solo aceptan caracteres cuya codificación UTF-8 cabe en un byte. Si en el ejemplo anterior se intentara hacer lo siguiente:

```
bytes[0] = 'む'
```

el compilador lanzaría un error similar a “constant 12416 overflows byte” (la constante que codifica el carácter Unicode es demasiado grande para un simple byte).

7.4 CONSTRUCCIÓN DINÁMICA DE CADENAS

En tiempo de ejecución, se puede construir una cadena a partir de una porción de byte o rune, donde se añaden, eliminan o modifican caracteres.

Esto puede ser útil en cadenas muy simples. Para usos más generales, hay otras maneras más ventajosas.

7.4.1 Concatenación de cadenas

El operador + permite concatenar dos cadenas:

```
str1 := "El operador de suma"
str2 := str1 + " es una manera sencilla " +
    "de concatenar cadenas"
```

Sin embargo, tiene sus limitaciones:

- Solo permite concatenar cadenas. Si se requiere concatenar, por ejemplo, un string y un int, primeramente se debe transformar el int a string (por ejemplo, usando la función strconv.Itoa).
- La operación de concatenación es rápida si puede definirse en una sola sentencia de código. Si se requiere concatenar muchas cadenas, haciendo uso de condicionales y bucles, cada una de las concatenaciones requerirá hacer una copia entera de la parte del string construido hasta el momento, incurriendo así en una enorme penalización de velocidad y uso de memoria.

7.4.2 Construcción con strings.Builder

Go proporciona un tipo de dato llamado strings.Builder, que permite componer cadenas paso a paso, adjuntando los diversos trozos de cadena al strings.Builder a través de sus funciones Write or WriteString. Cuando se haya finalizado la composición, la función String() retornará un string concatenando todas las cadenas anteriormente escritas. Este modo de trabajar es mucho más eficiente que el operador de concatenación +.
Por ejemplo:

```
var sb strings.Builder
sb.WriteString("strings.Builder\n")
l := sb.Len()
for i := 0; i < l; i++ {
  sb.WriteRune('=')
}
sb.WriteString("\nEs la forma más eficiente de construir ")
sb.WriteString("cadenas.\n")
sb.WriteString("Sin embargo, solo se pueden agregar ")
sb.Write([]byte("cadenas o porciones de bytes o runas. "))
sb.WriteString("Para añadir otros tipos, primero debes ")

sb.WriteString("transformarlos. Por ej. con strconv.Itoa(")
sb.WriteString(strconv.Itoa(1234))
sb.WriteByte(')')

fmt.Println(sb.String())
```

Construiría y mostraría en pantalla:

```
strings.Builder
===============
Es la forma más eficiente de construir cadenas.
```

Sin embargo, solo se pueden agregar cadenas o porciones de bytes o runas. Para añadir otros tipos, primero debes transformarlos. Por ej. con strconv.Itoa(1234)

El ejemplo anterior avanza algunos aspectos que serán explicados en capítulos posteriores:

- strings.Builder es un tipo de dato estructurado (struct) .
- Las funciones WriteString, Write, etc... no se invocan precedidas por el nombre de un paquete, sino que usan la variable sb como **receptor** de la función (sb.Write, sb.WriteString...). Esto quiere decir que estas funciones no trabajan en un ámbito global sino que tienen efecto sobre sb, una instancia concreta del tipo de dato strings.Builder .

7.4.3 Paquete fmt

fmt.Sprint, fmt.Sprintln y fmt.Sprintf (con S delante) son funciones con comportamiento análogo a fmt.Sprint, fmt.Sprintln y fmt. Sprintf, respectivamente. La diferencia es que en vez de mostrar un mensaje formateado por la salida estándar, estas funciones nos retornan dicho mensaje formateado en una cadena de texto.

Es la forma más expresiva, concisa y precisa para construir cadenas dinámicamente, aunque también es de las más lentas:

```
val := 3.1
str := fmt.Sprintf("val = %.2f\n", val)
//
str == "val = 3.10
```

De la misma manera, fmt.Sscan o fmt.Sscanf nos permiten extraer información formateada desde una cadena, como si de la entrada estándar (teclado) se tratara.

Capítulo 8
DICCIONARIOS (MAPAS)

Los diccionarios o mapas son estructuras de datos que permiten guardar elementos, pero no asociándolos a un índice lineal, como los vectores o porciones, sino a una "clave única" que los identifica de manera unívoca. Tanto la clave como el valor asociado pueden ser de cualquier tipo de dato.

Por ejemplo, un diccionario podría guardar un registro de pasaportes y sus respectivos años de caducidad, donde la clave del diccionario sea un string que contenga el número y las letras del pasaporte, y el valor asociado sea un int que guarde el año de caducidad.

En la jerga de los diccionarios, el pasaporte sería la **clave** y los datos de una persona serían el **valor.**

8.1 DECLARACIÓN DE MAPAS

La manera preferible de declarar un mapa es mediante una instancia literal, que puede definir un mapa vacío:

<variable> := map[<tipo_clave>]<tipo_valor>{}

o prerrellenado con algunas entradas:

```
<variable> := map[<tipo_clave>]<tipo_valor>{
    <clave_a> : <valor_a>,
    <clave_b> : <valor_b>,
    <clave_c> : <valor_c>,
    /* ... */
}
```

Ejemplos:

```
// Mapa vacío (clave: string, valor: int)

pasaportes := map[string]int{}

// Mapa inicializado con algunos valores
// Claves y valores son string

capitales := map[string]string{
  "España":      "Madrid",
  "Francia":     "París",
  "Reino Unido": "Londres",
  "Japón":       "Tokio",
}
```

Algunos detalles del ejemplo anterior a tener en cuenta son los siguientes:

- Incluso si pasaportes se crea vacío, su declaración require la abertura y

cerrado de llaves {} al final del tipo.

•En la definición del mapa de capitales, que ocupa múltiples líneas, incluso la última entrada ("Japón": "Tokio") debe llevar una coma al final.

8.2 ACCESO A ELEMENTOS

Para añadir elementos a un mapa, se debe seguir el siguiente patrón:

```
<mapa>[<clave>] = <valor>
```

Por ejemplo, usando el mapa capitales de la sección anterior:

```
capitales["Italia"] = "Roma"
```

Si se añade un valor con una clave ya existente, el valor anterior queda sobreescrito por el nuevo.

Para leer elementos, el nombre del mapa con la clave entre corchetes nos devolverá dicho elemento:

```
pais := "Francia"
capital := capitales[pais]
fmt.Printf("La capital de %s es %s", pais, capital)
```

Salida estándar:

```
La capital de Francia es París
```

Sin embargo, en el ejemplo anterior, si se buscara una clave inexistente en el mapa (por ejemplo, "Bélgica"), este devolvería un “valor cero”; es decir, 0 para números, nil para apuntadores u otras referencias, false para bool y la cadena vacía "" para string.

El valor cero puede ser un valor válido para un mapa, y su retorno no permite realmente saber si esa clave está registrada en el mapa (con el valor cero asociado) o si no lo está (por lo que retorna el valor cero). Para comprobar si una clave existe o no en el mapa, la operación de lectura puede, opcionalmente, devolver un segundo valor del tipo bool; será true si la clave está registrada, y será false si no lo está:

```
pais := "Narnia"
capital, ok := capitales[pais]
if ok {
  fmt.Println("La capital de", pais, "es", capital)
} else {
  fmt.Println("No he encontrado capital para", pais)
}
```

Salida estándar:

```
No he encontrado capital para Narnia
```

8.3 ELIMINANDO ENTRADAS CON delete

Go provee la función delete(<mapa>, <clave>) para eliminar entradas de un mapa, a través de su clave. Por ejemplo:

```
delete(capitales, "Reino Unido")
```

8.4 RECORRIENDO MAPAS CON range

De manera similar al recorrido de vectores y porciones, el operador range permite recorrer todos los valores de un mapa sin un orden específico:

```
for <clave>, <valor> := range <mapa> {
    /* código a repetir para cada par clave-valor */
}
```

Por ejemplo:

```
capitales := map[string]string{
  "España":      "Madrid",
  "Francia":     "París",
  "Reino Unido": "Londres",
  "Japón":       "Tokio",
}
for pais, capital := range capitales {
  fmt.Printf("La capital de %s es %s\n", pais, capital)
}
```

Salida:

```
La capital de Japón es Tokio
La capital de España es Madrid
La capital de Francia es París
La capital de Reino unido es Londres
```

Observe que, a diferencia de lo que sucede en el caso de vectores y porciones, un mapa no tiene por qué recorrer sus elementos en el mismo orden en el que fueron introducidos o insertados.

Si se desea ignorar los valores de un mapa y solo se quiere recorrer las claves, se puede omitir la segunda variable en la definición del bucle:

```
for pais := range capitales { ...
```

Si se desea ignorar las claves y solo se quiere recorrer los valores del mapa, se puede usar el identificador vacío _:

```
for _, capital := range capitales { ...
```

8.5 CONTANDO EL NÚMERO DE ELEMENTOS

Go provee la función len(<mapa>), que retorna el número de pares clave-valor en un mapa:

```
numero := len(capitales)
fmt.Println("contabilizadas", numero, "capitales")
```

Salida estándar:

```
contabilizadas 4 capitales
```

8.6 CONJUNTOS

Los conjuntos son estructuras de datos que permiten:

- Insertar elementos no duplicados
- Eliminar elementos
- Contar el número de elementos
- Comprobar si un elemento existe
- Recorrer los elementos uno a uno

Go no provee el tipo de dato Set, como otros lenguajes. Sin embargo, se puede aprovechar la estructura de datos que los mapas usan para guardar las claves y crear un mapa cuyas claves guarden los valores que queremos insertar en un conjunto y los valores guarden un tipo de dato vacío especial de Go: la "estructura vacía" struct{}, que se explicará con más detalle en el tema 11 sobre tipos de datos estructurados.

En el siguiente ejemplo, se muestran 6 números aleatorios pertenecientes a un sorteo de lotería. Es importante que dichos números no estén repetidos, así que se utiliza un conjunto construido a partir de un mapa cuya clave es un número entero y cuyo valor es una instancia de la estructura vacía struct{}, ya que no nos interesa ningún valor asociado a dichos números, sino los números en sí. Antes de insertar un número, nos aseguraremos de que no esté ya presente en el mapa. Si ya estuviera, este no se inserta y se prueba generando otro número aleatorio:

```
fmt.Println("Los números ganadores de la lotería son:")
numeros := map[int]struct{}{}
for len(numeros) < 6 {
  n := rand.Intn(49) + 1
  // solo se muestra el número si no ha salido antes
  if _, ok := numeros[n]; !ok {
    numeros[n] = struct{}{}
    fmt.Println("El", n, "...")
  }
}
```

Observe que la creación del mapa finaliza en dos pares de llaves {}{} seguidos. Uno pertenece a la definición de la estructura vacía struct{} y el otro indica la creación del mapa vacío.

El ejemplo anterior mostraría una secuencia similar a la siguiente:

Los números ganadores de la lotería son:
El 42 ...
El 24 ...
El 37 ...
El 8 ...
El 28 ...
El 12 ...
Felicidades a los premiados!

En el vector de números del ejemplo anterior veríamos:

valor, ok := numeros[n]

Siempre guardará struct{}{} en valor, por lo que es un valor poco interesante. En el ejemplo se usa _, ok := numeros[n] porque el valor se puede ignorar, y la parte interesante es el ok, que contendrá true si la clave n está ya en el conjunto numeros.

8.7 DETALLES INTERNOS DE map

Al igual que sucede en las porciones y cadenas de texto, el tipo de dato map funciona **por referencia.** Esto quiere decir que cuando una variable map se asigna a otra variable, cuando se invoca una función y se pasa un map como argumento, o una función retorna un map, en ningún momento se hace una copia del mapa original, por lo que diversas variables pueden estar trabajando sobre (y modificando) un mismo espacio de memoria.

Un diccionario, como interfaz genérica en las ciencias de la computación, puede ser implementado internamente de diversas formas, cada una con sus ventajas e inconvenientes. El tipo de dato map de Go implementa el tipo de diccionario más comúnmente usado: la "tabla de *hash*".

Una tabla de *hash* tiene una ventaja primordial con respecto a otras implementaciones de un diccionario: la velocidad. Generalmente, las operaciones de búsqueda, inserción y eliminación requieren un tiempo de ejecución constante. También se puede usar casi cualquier tipo de dato de Go como clave, ya que la "función de *hash*" que da nombre a la tabla, y que se usa para definir la posición de una clave en la tabla, se puede calcular con cualquier ristra de bits.

A medida que un mapa crece, según se van añadiendo nuevos pares clave-valor, el rendimiento de la tabla de *hash* puede ir degradándose. Llegado a cierto punto, Go puede decidir hacer un *rehash:* crear una nueva tabla de *hash* más grande y copiar allí los valores de la tabla de *hash* antigua con una indización mejorada para una mayor velocidad de acceso.

La operación de *rehash* es costosa, pero se puede minimizar si se conoce de antemano el tamaño aproximado de un mapa. Se puede crear un map mediante la función make, que permite especificar el número de elementos que ese mapa debería ser capaz de guardar, de manera eficiente y sin necesidad de hacer *rehashing:*

```
capitales := make(map[string]string, 1000)
```

Los mapas basados en *hash* tienen una limitación primordial con respecto a otras implementaciones más lentas de un diccionario, como las basadas en árboles binarios, y es que sus elementos no se recorrerán en un orden consistente, ya que siempre dependerá de factores internos y circunstanciales, como el tamaño de la tabla de *hash* o el orden en el que los elementos se han insertado.

Capítulo 9
ORGANIZACIÓN DE CÓDIGO
PAQUETES Y MÓDULOS

9.1 PAQUETES (package)

Un paquete (package) es un conjunto de archivos, bajo un mismo directorio, que agrupan definiciones de funciones, constantes y variables globales, así como tipos de datos definidos por el usuario (tal y como se mostrará en el próximo capítulo). Cada paquete se identifica mediante un nombre y una ruta al directorio que contiene dichos archivos.

Un directorio solo puede contener archivos de un mismo paquete, pero sus subdirectorios pueden contener otros paquetes.

Cada paquete debe tener un nombre que, según las recomendaciones de buenas prácticas de Go, debe ser breve, autoexplicativo y no ambiguo (elegir buenos nombres es todo un reto en Go).

La primera directiva en cada archivo .go debe ser la palabra package seguida del nombre del paquete. Los archivos del directorio que contiene la función main() deben definir el package main, como muestran los ejemplos completos de este libro.

Para usar los tipos de datos, variables, constantes y funciones de otros paquetes, el fichero .go que los utilice debe incorporar dichos paquetes con la directiva import:

```
import "fmt" import "net" import "math/rand"
```

Sin embargo, la forma recomendada de incorporar múltiples paquetes es agrupándolos bajo la misma directiva import:

```
import
(
  "fmt"
  "net"
  "math/rand"
)
```

Cabe observar una sutil diferencia: mientras package nombre no requiere poner el nombre del paquete entre comillas, la ruta de los paquetes tras import sí debe ir entre comillas.

Otro aspecto importante a considerar es que Go no permite incorporar paquetes que no se van a usar (lanzaría un error de compilación).

Cuando se utilizan las funciones, tipos y constantes proporcionadas por un paquete, estas deben ir precedidas por el nombre del paquete si se ejecutan desde fuera de este, como ya habrá visto en los ejemplos de este libro:

```
fmt.Println("Hola")
sb := strings.Builder{}
a := rand.Int()
```

9.2 MÓDULOS

A pesar de la gran riqueza y variedad de funcionalidades que proveen los paquetes estándar de Go, a veces es útil incorporar en nuestros programas bibliotecas de código hechas por terceras personas (o por nosotros mismos), potenciando así la compartición y reusabilidad del código.

Un módulo es una colección de paquetes de Go agrupados en un mismo árbol de directorios. Un módulo puede contener tanto una biblioteca de funciones como un programa completo con su función main.

El directorio raíz de cada módulo contiene un fichero llamado go.mod, que sigue el formato de la siguiente plantilla:

```
module <ruta del módulo>

go <versión>

require (
  <lista de dependencias a otros módulos>
)
```

La <ruta del módulo> suele ser una ruta web (sin el http) al repositorio de código donde el módulo está accesible. Por ejemplo:

```
module github.com/stretchr/testify
```

El sistema de módulos de Go requiere que el código fuente del módulo a importar sea accesible para el usuario de dicho módulo (no existen bibliotecas precompiladas como en C o Java). Es por eso que una gran cantidad de los módulos adoptados por la comunidad Go son distribuidos a través de la plataforma GitHub (github.com) o similares.

La línea go <versión> indica la versión mínima de Go requerida para compilar dicho módulo. Por ejemplo: go 1.14.

La sección require de go.mod permite definir dependencias a otros módulos, ya que nuestro módulo puede hacer uso del código de otros módulos. Dicha sección incluye una línea por módulo externo a utilizar, consistente en la ruta del módulo a importar seguido de la versión del código, que suele coincidir con el nombre de alguna rama o etiqueta del repositorio de código o el identificador del *commit*.

El siguiente ejemplo muestra el fichero go.mod, tomado de una biblioteca de pruebas ampliamente utilizada en Go:

```
module github.com/stretchr/testify

go 1.13

require (
  github.com/davecgh/go-spew v1.1.0
  github.com/pmezard/go-difflib v1.0.0
  github.com/stretchr/objx v0.1.0
  gopkg.in/yaml.v3 v3.0.0-20200313102051-9f266ea9e77c
)
```

Aunque un programa concreto no esté pensado para ser importado como dependencia en otros módulos, este deberá definir un fichero go.mod si desea incorporar módulos externos.

Existen otras alternativas a los módulos de Go para la definición e incorporación de dependencias, pero son totalmente obsoletas y su gestión es poco amigable.

9.3 CREANDO MÓDULOS Y PAQUETES

Esta sección explica, paso a paso, cómo crear un módulo de Go, así como un paquete que será importado desde el package main del mismo módulo para su uso.

El primer paso es crear el fichero go.mod. Es un fichero de texto plano que podría ser escrito a mano, pero la manera más sencilla de hacerlo es mediante el comando go mod init <ruta del modulo>.

Por ejemplo, se puede crear una carpeta ejemplo, que contendrá el módulo a crear. Desde el terminal, se debe ejecutar el siguiente comando en la carpeta ejemplo:

```
go mod init macias.info/go/ejemplo
```

Salida:

```
go: creating new go.mod: module macias.info/go/ejemplo
```

El comando anterior crearía un fichero go.mod con el nombre dado, sin ningún otro módulo a importar en la sección require:

```
module macias.info/go/ejemplo
```

go 1.14

El siguiente paso sería crear, en el mismo directorio ejemplo, un subdirectorio hola; y, en este, un archivo hola.go con el siguiente contenido:

```go
// Package hola implementa maneras de saludar
package hola

import "fmt"

// ConNombre retorna un efusivo saludo, dado un nombre
func ConNombre(nombre string) string {
  return fmt.Sprintf("¡Hola, %s!", nombre)
}
```

9.4 IMPORTANDO PAQUETES DEL MÓDULO LOCAL

Los paquetes estándar de Go, incorporados en toda instalación del entorno de desarrollo de Go, tienen rutas de acceso cortas, tales como fmt o math/ rand. Cuando se importa un paquete de un módulo propio o externo, este suele llevar delante la ruta del módulo, seguida de la ruta del paquete.

Por ejemplo, para importar el paquete creado en el apartado anterior, crearemos un fichero main.go en el directorio raíz del modulo macias.info/go/ejemplo (el mismo donde está el archivo go.mod) con el siguiente contenido:

```
package main

import (
  "fmt"

  "macias.info/go/ejemplo/hola"
)

func main() {
  fmt.Print("Cómo te llamas?: ")
  var nombre string
  fmt.Scanln(&nombre)
  fmt.Println(hola.ConNombre(nombre))
}
```

Ejemplo de entrada y salida estándar:

Cómo te llamas?: Manuel

¡Hola, Manuel!

Del anterior código, cabe destacar que:

- fmt es un paquete estándar de Go y no necesita más que su nombre para

que Go sepa dónde ir a buscarlo.

•Cualquier otro paquete deberá concatenar el nombre del módulo (macias.info/go/ejem-plo) y el nombre del paquete (/hola).

–Si el paquete a importar está en el directorio raíz del módulo, bastará con el nombre del módulo.

•Igual que para invocar las funciones del paquete fmt el nombre de dicho paquete se debe añadir (seguido de punto) antes de la función a invocar (por ejemplo, fmt.Println), el nombre del paquete local también debe usarse delante de la invocación a sus elementos (hola.ConNombre) .

El árbol de directorios y ficheros del módulo creado quedaría tal que:

```
ejemplo/
├── hola
│   └── hola.go
├── go.mod
└── main.go
```

El nuevo módulo se ejecutaría, desde el directorio de ejemplo, como cualquier otro programa de go:

```
go run main.go
```

(o go build -o mi-ejecutable main.go para generar un fichero ejecutable).

9.4.1 Dependencias circulares

Go está diseñado para conseguir un tiempo de compilación realmente rápido, así como para incentivar la simplicidad de los repositorios de código.

Como consecuencia, Go no soporta dependencias circulares. Esto significa que el paquete X no puede importar el paquete Y si el paquete Y, directa o indirectamente (a través de alguno de los paquetes importados por este) importa el paquete X.

9.5 INCORPORANDO PAQUETES DE MÓDULOS EXTERNOS

Vamos a añadir una funcionalidad nueva al ejemplo anterior. Además de saludar, el programa mostrará unas estadísticas sobre el nombre introducido. Dicha funcionalidad ya está implementada en un módulo externo y públicamente disponible, llamado github.com/mariomac/analizador, que proporciona la función analizador.PrintEstadistica.

La manera más sencilla de incorporar dicho módulo es, primero, importar el paquete github.com/mariomac/analizador en el fichero que lo va a usar (en este caso, main.go), e invocar directamente a la función a usar (al final de main):

```
package main

import (
  "fmt"

  "github.com/mariomac/analizador"
  "macias.info/go/ejemplo/hola"
)

func main() {
  fmt.Print("Cómo te llamas?: ")
  var nombre string
  fmt.Scanln(&nombre)
  fmt.Println(hola.ConNombre(nombre))

  analizador.PrintEstadistica(nombre)
}
```

Directamente, cuando se intente ejecutar el nuevo programa main.go, se

puede ver cómo Go va a descargar los nuevos paquetes de Internet:

```
$ go run main.go
go: finding module for package github.com/mariomac/analizador
go: downloading github.com/mariomac/analizador v1.0.0
go: found github.com/mariomac/analizador in github.com/mariomac/
analizador v1.0.0
```

El nuevo programa ahora implementa una nueva funcionalidad que se ha incorporado a través de una biblioteca disponible a través de un módulo en Internet.

```
Cómo te llamas?: Antón
¡Hola, Antón!
La palabra "Antón" contiene:
        - 1 mayúsculas
        - 4 minúsculas
        - 2 vocales
        - 3 consonantes
Histograma de letras:
        A : 1 apariciones
        N : 2 apariciones
        O : 1 apariciones
        T : 1 apariciones
```

Si abre de nuevo el fichero go.mod, verá que la dependencia hacia el analizador ha sido automáticamente añadida:

```
module macias.info/go/ejemplo

go 1.14

require github.com/mariomac/analizador v1.0.0
```

Además, observará que un nuevo archivo llamado go.sum ha sido añadido automáticamente al directorio. Dicho archivo contiene *hashes* criptográficos sobre los módulos descargados, para verificar que en futuras descargas de un mismo módulo el contenido de este no ha cambiado (accidental o maliciosamente).

9.6 COPIAS LOCALES DE MÓDULOS. EL DIRECTORIO vendor

La naturaleza abierta del sistema de módulos de Go permite exponer módulos públicamente y con total libertad, al no tener que depender de un proveedor concreto.

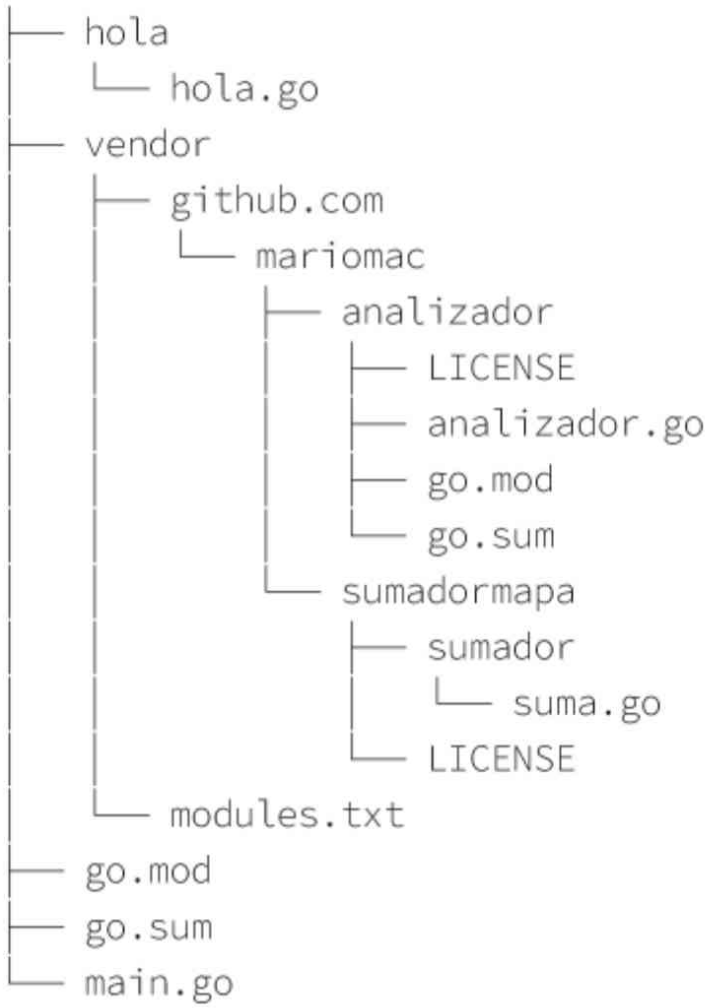
Dicha libertad y falta de control centralizado es una ventaja, pero también puede llevar a situaciones en las que un módulo del cual depende nuestro software deje de estar disponible, ya sea porque el servidor donde estaba alojado el código ha caído temporalmente, o porque el autor del módulo se ha enfadado con el mundo y ha decidido borrarlo.

Para evitar que estas situaciones puedan afectar al proceso de construcción de nuestro software, Go permite hacer una descarga local de los módulos externos, que serán guardados en el directorio vendor de nuestros proyectos. El contenido de la carpeta vendor se puede guardar y distribuir conjuntamente con nuestro software, siempre que la licencia de los módulos de terceros lo permita.

Para guardar una copia local de los módulos externos en un proyecto local, se debe ejecutar el siguiente comando desde el directorio raíz que contiene el fichero go.mod:

```
go mod vendor
```

Si se ejecutara el comando anterior en el proyecto de ejemplo de las secciones anteriores, el árbol de directorios del proyecto quedaría de la siguiente manera:



El módulo externo analizador se ha guardado en la correspondiente ruta dentro de la carpeta vendor. Además, hay otro módulo llamado sumadormapa que no estaba dentro del go.mod local. Esto se debe a que analizador requiere del modulo sumadormapa y, para asegurar el correcto funcionamiento de analizador, se debe guardar también una copia de sumadormapa.

El fichero vendor/modules.txt sirve como índice de los módulos descargados localmente, de tal manera que el programador pueda ver, de un solo vistazo, qué módulos contiene vendor.

9.7 ELEMENTOS PÚBLICOS Y PRIVADOS A NIVEL DE PAQUETE

Un mismo paquete puede contener diversos elementos, ya sean funciones, variables globales, constantes o incluso nuevos tipos de datos. De esos elementos, algunos son útiles para quien incorpora ese paquete en un código, y otros por sí solos no tienen más utilidad que implementar detalles y partes de otras funciones del paquete (con los que no se debe interactuar fuera del paquete).

Con tal de simplificar el uso de los paquetes, es conveniente hacer visibles solo aquellos elementos de un paquete con los que el usuario debe interactuar (elementos públicos), y dejar los pequeños detalles internos visibles solo dentro del mismo paquete, e inaccesibles desde otros paquetes (elementos privados).

Haciendo una analogía con el mundo real, piense en las partes de su ordenador con las que usted debe interactuar: pantalla, teclado, altavoces, ratón... todas ellas son visibles (públicas). Sin embargo, la circuitería interna, las baterías, los cables... todo eso está oculto tras una carcasa —no solo por cuestiones de estética, sino también para hacer la máquina más amigable al uso y para evitar que una manipulación accidental de dichos componentes internos pueda causar una avería o un mal funcionamiento—.

Go permite crear elementos públicos y privados según la siguiente regla:

- Cualquier función, variable global, constante o —como se verá en próximos capítulos— tipo de datos o atributo de una estructura cuyo nombre empiece en **minúscula** será considerado **privado a nivel de paquete.** Solo se podrá acceder a ese elemento desde el código del mismo paquete que lo define.
- Cualquier función, variable global, constante, tipo de dato o atributo de estructura cuyo nombre empiece en **mayúscula** será considerado **público.**

Pongamos como ejemplo un proyecto con la siguiente estructura de directorios, con un paquete llamado cubo y el paquete main:

```
.
├── cubo
│   └── cubo.go
└── main.go
```

El archivo cubo.go contiene una función pública (Volumen) y una función privada (eleva3):

```
package cubo

func eleva3(x float64) float64 {
  return x * x * x
}

func Volumen(lado float64) float64 {
  return eleva3(lado)
}
```

La función eleva3 puede invocarse desde la función Volumen, ya que ambas están en el mismo paquete. Cualquier parte del programa, incluyendo la función main, puede invocar la función Volumen, ya que esta es pública:

```
package main

import (
  "fmt"
  "macias.info/go/paquete/cubo"
)

func main() {
  fmt.Println("El volumen de un cubo de lado 3.5 es",
              cubo.Volumen(3.5))
}
```

Si se intentara invocar la función cubo.eleva3 desde fuera del paquete cubo, se obtendría el siguiente error de compilación:

```
./main.go:?:?: cannot refer to unexported name cubo.eleva3
```

9.8 ALIAS DE PAQUETE

Puede pasar que dos paquetes de diferentes módulos tengan el mismo nombre. Eso no supone ningún problema al incorporarlos mediante import, ya que ambos paquetes tendrán rutas distintas. Sin embargo, sí será un problema a la hora de usarlos para invocar sus elementos.

Para evitar este problema, se puede especificar un "alias" o seudónimo delante de la ruta de importación. En el archivo que incluye dicho alias, los elementos de ese paquete no se invocarán a través del nombre del paquete sino a través del alias. Por ejemplo, si se quisiera importar un paquete llamado cubo de una biblioteca externa y se debiera usar conjuntamente con el paquete cubo del ejemplo de la sección anterior, se haría de la siguiente manera:

```
package main

import (
  "fmt"
  "macias.info/go/paquete/cubo"


  calc "github.com/ejemplo/utils/cubo"
)

func main() {
  fmt.Println("El volumen de un cubo de lado 3.5 es",
              cubo.Volumen(3.5))
  fmt.Println("La raíz cúbica de 9 es", calc.Raiz(9))
}
```

En situaciones normales, la función calc.Raiz se invocaría como cubo. Raiz. En este ejemplo, se usa el alias calc definido delante de la ruta del apartado import.

9.9 LA FUNCIÓN init

Cada paquete en Go puede definir un tipo de función especial, con la siguiente cabecera:

```
func init()
```

Esta función puede declarar código de inicialización que se ejecutará una sola vez en el paquete, durante la inicialización o carga (sin necesidad de invocarlo explícitamente).

Cuando un programa comienza, antes de que la función main sea invocada, todos los paquetes son inicializados. Cada paquete se inicializa una —y solo una— vez, incluso si se ha importado desde múltiples puntos del programa.

Cuando un paquete se inicializa, se realizan los siguientes acciones:

1. Recursivamente, inicializa primero todos los paquetes que hayan sido importados desde el paquete y no hayan sido inicializados ya.
2. Computa y asigna los valores iniciales para las variables globales declaradas en el paquete.
3. Ejecuta la función init() del paquete.

Se pueden definir múltiples funciones init dentro de un mismo paquete, por motivos de legibilidad o modularidad.

Las funciones init deben definirse con extremo cuidado, y su uso se desaconseja o incluso se considera una mala praxis, ya que definen un comportamiento oculto que puede ser difícil de reproducir y depurar.

También hacen uso de variables globales que, incluso cuando no son utilizadas por el importador, no pueden ser eliminadas por los optimizadores de código.

En otros casos, borrar un paquete importado porque ya no se utiliza puede tener efectos inesperados, porque al dejar de invocar la función init algunos de sus comportamientos ocultos también dejan de tener efecto.

En estos casos muy muy puntuales (e indicativos de que hay algo mal en el

código), se puede forzar la importación de un paquete mediante un “alias vacío” _:

```
import _ "example.com/mi/codigo/paquete"
```

Al igual que con las variables, Go prohíbe definir un paquete que no se utiliza. Definiendo un alias mediante el identificador vacío permite inicializar un paquete no usado y evitar que el compilador lance un error.

Capítulo 10
DEFINICIÓN DE TIPOS DE DATOS

Del mismo modo que los paquetes permiten declarar funciones, constantes y variables globales, es posible declarar también nuevos tipos de datos mediante la directiva type:

```
type <Nombre> <definición>
```

Como forma más sencilla, es posible definir un nuevo tipo de datos a partir de un tipo existente. Por ejemplo, así se definiría el tipo de datos Edad, representado por un solo número entero de 0 a 255:

```
type Edad uint8
```

El nuevo tipo Edad es, en realidad, un uint8 que acepta valores entre 0 y 255 y permite realizar las mismas operaciones que un uint8. Entonces, ¿por qué no usar directamente un uint8 para guardar los datos pertenecientes a la edad? La razón principal es que el tipo de datos Edad aporta un nuevo significado semántico dentro del dominio del programa: es un número, pero sabemos que solo contiene datos relativos a la edad.

Se puede declarar e iniciar una variable del tipo Edad de la misma manera que para otros tipos de datos numéricos:

```
var
adulto Edad = 18
jubilado := Edad(65)
```

Al igual que para los demás tipos de datos numéricos, no hay conversiones implícitas. Si se desea asignar un uint8 a una variable del tipo Edad, se debe hacer una conversión explícita de tipos:

```
var
```

```
y Edad
num := 27
y = Edad(num)
```

Al igual que para funciones y valores globales, los tipos cuyo nombre comienza en **minúscula** tienen **visibilidad privada** (son visibles solo desde el paquete donde se han definido), y los tipos cuyo nombre comienza en **mayúscula** tienen **visiblidad pública** (se pueden usar desde cualquier punto del programa). Con la excepción, claro está, de los tipos de datos incluidos en el lenguaje Go:int, bool, string...

Del mismo modo, cuando los tipos de datos definidos se referencian desde paquetes diferentes al que los define, se debe prefijar el nombre del paquete al nombrar los tipos de datos, seguido de punto. Por ejemplo: strings.Builder.

10.1 TIPOS A PARTIR DE PORCIONES

De la misma manera que se pueden definir nuevos tipos a partir de tipos de datos básicos, también se pueden definir nuevos tipos a partir de porciones, y así darles un nuevo significado semántico:

```
type Año int
type Nacimientos []Año

func main() {
  censo := Nacimientos{1979, 1983, 1965}
  censo = append(censo, 1990)
  censo = append(censo, 1955)

  suma := Año(0)
  for _, a := range censo {
    suma += a
  }

  media := suma / Año(len(censo))
  fmt.Println("Año de nacimiento medio:", media)
}
```

Salida estándar:

Año de nacimiento medio: 1974

En el código anterior, se puede ver cómo el tipo Nacimientos puede ser utilizado como una porción []Año cualquiera:

- La instanciación de un literal con algunos valores predefinidos se hace anteponiendo el nombre a los valores entre llaves. La forma censo := make(Nacimientos, 0, 10) también sería válida.
- Se pueden añadir nuevos elementos mediante append.

- Se puede recorrer mediante el operador range.

Es preciso observar, en el cálculo de la media, que el resultado len(censo) debe convertirse explícitamente al tipo de datos Año, ya que Go no permitiría dividir un Año por un int. Una alternativa válida sería convertir suma a int mediante int(suma). La única diferencia sería que, mientras en el ejemplo actual media es del tipo Año (resultante de dividir dos Año), con la mencionada alternativa sería del tipo int.

10.2 TIPOS A PARTIR DE MAPAS

También es posible definir nuevos tipos de datos a partir de mapas, y usarlos como tales:

```
type Año int
type NombreNacimiento map[string]Año

func main() {
  artistas := NombreNacimiento{
    "Vincent Van Gogh": 1853,
    "Elvis Presley":    1935,
    "Salvador Dali":    1904,
  }
  artistas["Rick Astley"] = 1966
  for nombre, año := range artistas {
    fmt.Printf("%s nació en %d\n", nombre, año)
  }
}
```

Salida:

Vincent Van Gogh nació en 1853
Elvis Presley nació en 1935
Salvador Dali nació en 1904
Rick Astley nació en 1966

El tipo NombreNacimiento se instancia como un map cualquiera con unos valores predefinidos. Se le pueden añadir nuevas entradas especificando una clave entre corchetes, y sus pares clave-valor se pueden recorrer mediante range.

10.3 TIPOS FUNCIONALES

En el capítulo de funciones se vio que una función con una signatura concreta es un tipo de datos en sí misma. Para mejorar la legibilidad de nuestras API *(Application Programming Interface,* Interfaz de Programación de Aplicaciones), podemos dar a cada uno de esos tipos funcionales su propio nombre. Supongamos un tipo de datos Generador, del que puede formar parte cualquier función sin argumentos que retorna un int:

```
type Generador func() int
```

Una función sencilla que cumple la signatura del tipo Generador podría ser una función que simplemente genera un cero cada vez que se invoca:

```
func
Cero() int {

return
0
}
```

Un Generador de más utilidad podría ser un contador que cada vez que se invoca retorna un número igual al anterior más uno, de tal manera que una serie de invocaciones a la misma instancia del contador retornarían 1, 2, 3, 4... Para eso, es necesario que el estado del contador se guarde en alguna parte. Una opción sería utilizar una variable global, pero entonces no podríamos tener varios contadores a la vez con distintos valores. La manera funcional de Go sería:

1. Crear una función Contador que no es un Generador en sí, sino que retorna un Generador:

   ```
   func Contador() Generador
   ```

2. En el cuerpo de Contador, definir la variable local cuenta, inicializada a

cero.

3. Que la función Contador retorne una función del tipo func() int, que incrementa el contador definido como variable local de Contador y lo retorna.

La función Contador quedaría tal que:

```
func Contador() Generador {
  cuenta := 0
  return func() int {
    cuenta++
    return cuenta
  }
}
```

Un ejemplo de uso:

```
cnt := Contador()
fmt.Println(cnt())
fmt.Println(cnt())
fmt.Println(cnt())
cnt = Contador()  //
genera nuevo contador

fmt.Println(cnt())
```

Salida estándar:

```
1
2
3
1
```

En el ejemplo anterior, la primera línea de la función Contador fuerza a Go a

guardar espacio para una nueva instancia de la variable cuenta en una zona especial de la memoria (llamada *heap)*, y retorna una función (sin invocarla) con una referencia a esta variable en el *heap* (Figura 10.1). La nueva función queda referenciada a través de la variable cnt. La instancia de la variable cuenta seguirá en el *heap* mientras la variable cnt apunte al generador retornado.

La invocación al generador retornado por Contador debe hacerse a través de la variable donde se ha guardado: cnt(). Aunque parezca contraintuitivo según la forma en que hasta ahora este libro ha usado las funciones, cnt tiene un estado interno que no es global, sino local y diferente para cada generador obtenido tras múltiples invocaciones de Contador().

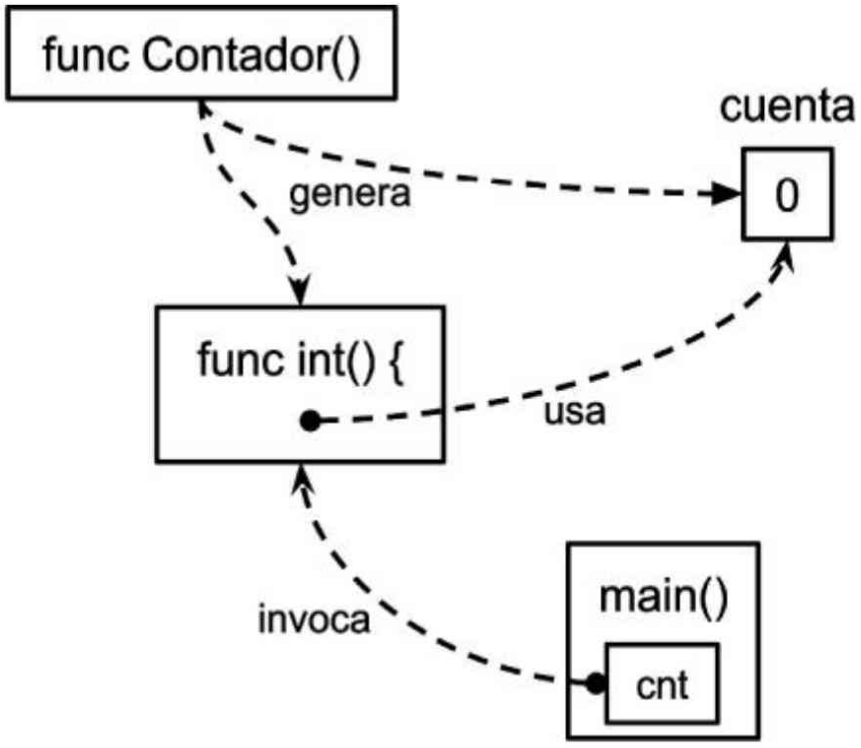


Figura 10.1 Una invocación a Contador genera y retorna una función que usa una variable creada desde fuera (esquema de memoria).

Otro ejemplo similar: una función que retorna un Generador de números

aleatorios. En este caso, podría ser útil pasar una "semilla" al Generador para inicializar el objeto generador de números aleatorios. La semilla puede pasarse por argumento no al Generador, sino a la función que lo crea y retorna:

```
func Aleatorio(semilla int64) Generador {
  rnd := rand.NewSource(semilla)
  return func() int {
    return int(rnd.Int63())
  }
}
```

Observe que el argumento se utiliza en la función que "genera el Generador" (valga la redundancia), pero el Generador retornado sigue sin aceptar ningún argumento, ya que es la manera en que dicho tipo de datos se ha definido. Como ejemplo de uso, la siguiente función GenerarTodo recibe como argumento un número variable de generadores y retorna una porción de int con un número generado por cada argumento, en el orden de invocación:

```
func GenerarTodo(gens ...Generador) []int {
  nums := make([]int, 0, len(gens))
  for _, fg := range gens {

    nums = append(nums, fg())
  }
  return nums
}
```

Observe que esta función no precisa de ninguna implementación concreta de Generador, sino que funciona de manera genérica. La función puede ser proporcionada por un módulo externo y el programador sabe que solo debe aportar nuevas funciones que cumplan con la signatura de Generador para

modificar los resultados de GenerarTodo.

Ejemplo de uso:

```
cnt := Contador()
rnd := Aleatorio(456)
for
i := 0; i < 5; i++ {
  fmt.Println(GenerarTodo(Cero, cnt, rnd))
}
```

Salida estándar:

```
[0 1 3253734790460108590]
[0 2 45774782476566703 29]
[0 3 878791956162078 0516]
[0 4 7718007074468753867]
[0 5 52937981495895779 01]
```

Observe que el primer argumento pasado a GenerarTodo no lleva paréntesis, ya que no pretendemos invocar la función Cero sino pasar una referencia a esta para que sea invocada por GenerarTodo.

10.4 RECEPTORES DE FUNCIÓN. MÉTODOS

En el capítulo sobre cadenas de texto se mostraron algunas funciones que no se invocan tras el nombre de un paquete, sino tras el nombre de una variable:

```
var
sb strings.Builder
sb.WriteString("hola")
sb.WriteString("!")
str := sb.String()
```

Este tipo de funciones suelen llamarse "métodos", por su similitud con los lenguajes orientados a objetos (pese a que Go no lo sea). La particularidad de los métodos es que pueden leer o modificar el estado de la variable sobre la cual se invoca.

Un método se define como una función que añade la información del "receptor de la función" como un argumento entre paréntesis, antes del nombre de la función:

```
func
(<nombre> <tipo>) NombreMétodo(<argumentos>) <tipo(s)
retorno>
```

Por ejemplo:

```
type PH float32

func (p PH) Categoria() string {
  switch {
  case p < 7:
    return "ácido"
  case p > 7:
    return "básico"
  default:
    return "neutro"
  }
}
```

El tipo PH es un número que representa el Potencial de Hidrógeno, que define cuán alcalina o ácida es una disolución. La ventaja de definirlo como un nuevo tipo de dato es que, además de ser usado como un simple float32, permite ampliar su funcionalidad mediante nuevos métodos, como el método Categoria, que retorna si un PH concreto es ácido, básico o neutro.
En Categoria, el receptor del método o función se referencia mediante (p PH) (tratado como si fuera un argumento más), y a partir de este se puede leer el valor del float32 definido en el tipo.
Ejemplo de uso del tipo de datos PH e invocación a sus métodos:

```
phs := []PH{
  PH(7), PH(1.2), PH(9),
}
for _, ph := range phs {
  fmt.Printf("Un pH == %v es %v\n", ph, ph.Categoria())
}
```

Salida estándar:

Un pH == 7 es neutro

Un pH == 1.2 es ácido

Un pH == 9 es básico

Al igual que con los argumentos comunes, el receptor de una función puede ser un valor o un apuntador. Los métodos que modifiquen el estado del receptor tienen que usar un puntero, al igual que con el paso de argumentos por referencia. El siguiente ejemplo implementaría un contador con una funcionalidad similar al contador-generador del apartado anterior, pero con la ventaja de que ahora el estado interno del contador puede ser compartido entre varios métodos:

```
type Contador int

func (c *Contador) Incrementa() {
  *c++
}

func (c *Contador) Reinicia(nuevoValor int) {
  *c = Contador(nuevoValor)
}
```

En este caso, es importante que el receptor de las funciones Incrementa y Reinicia sea un apuntador (y se use como tal en el cuerpo de las funciones), ya que ambas funciones modifican el estado del Contador. De no usar un apuntador como receptor, los métodos trabajarían sobre una copia del contador original, que no cambiaría tras las invocaciones.

var

c Contador

c.Incrementa()

c.Incrementa()

c.Incrementa()

```
  fmt.Println("valor:", c)
  c.Reinicia(77)
  fmt.Println("tras reinicio:", c)
```

Salida estándar:

```
  valor: 3
  tras reinicio: 77
```

10.5 TIPOS PSEUDOENUMERADOS

Un tipo de datos enumerado es un tipo de datos que solo acepta unos valores concretos y limitados, definidos por el usuario. Por ejemplo, un tipo de datos que definiera el palo de una baraja española podría ser un enumerado cuyos únicos valores fueran: Espadas, Bastos, Oros y Copas.

Muchos lenguajes de programación permiten la definición de tipos de datos enumerados, pero Go no. No obstante, es posible simular esta funcionalidad mediante la combinación de constantes y definición de tipos.

La técnica común para simular tipos de datos enumerados en Go consta de dos partes:

1. Definir un nuevo tipo de datos, basado en un tipo numérico entero o en string.
2. Definir, como constantes, los valores que ese tipo de datos puede tomar, donde cada constante tiene un valor diferente de las otras.

Por ejemplo:

```
type PaloBaraja int

const (
  Espadas PaloBaraja = 0
  Bastos  PaloBaraja = 1
  Oros    PaloBaraja = 2
  Copas   PaloBaraja = 3
)
```

Sin embargo, esta técnica no puede considerarse como una implementación completa de un tipo de datos enumerado, ya que nada impide que a una variable del tipo PaloBaraja se le asigne un valor cualquiera, más allá de los ya definidos Espadas, Bastos, Oros, o Copas:

var carta PaloBaraja = -22 **var**

```
i int = 3434
carta = PaloBaraja(i)
```

En el ejemplo anterior, la primera línea es válida porque cualquier literal de número entero puede ser directamente asignado durante la declaración de la variable carta. La última línea también es válida porque es legal convertir explícitamente de un int cualquiera hacia un PaloBaraja. Sin embargo, los valores asignados están fuera del contexto semántico que se le pretende dar a este tipo. Es por ello que esta sección se titula "pseudoenumerados".

Una ventaja de definir nuevos tipos para valores pseudo-enumerados es la posibilidad de definir métodos, mediante receptores de función, para añadir funcionalidades extra. Por ejemplo, si se usara una variable del tipo PaloBaraja como argumento de fmt.Println, esta mostraría un nada significativo valor entero entre 0 y 3. Sin embargo, cualquier tipo de datos en Go que implemente el método String() string será representado según el valor que dicho método retorne al invocarse sobre un valor dado.

El siguiente método haría que un PaloBaraja no se mostrara como un número en fmt.Print, sino como un texto:

```
func (f PaloBaraja) String() string {
  switch f {
  case Espadas:
    return "Espadas"
  case Bastos:
    return "Bastos"
  case Oros:
    return "Oros"
  case Copas:
    return "Copas"
  }
  return "inválido"
}
```

10.5.1 El operador iota

Para facilitar la asignación de números únicos a las constantes enumeradas, el operador iota asigna un número en incremento (empezando desde cero) a un grupo de constantes definidas bajo el mismo bloque const (...). El siguiente código ejemplificaría cómo funciona con unas simples constantes del tipo int:

```
const (
  Cero = iota
  Uno
  Dos
  Tres
)
```

El operador iota asigna 0 a la primera constante en el grupo (Cero). Luego se incrementa y, en la siguiente línea, asigna el valor 1 a la siguiente constante... y continúa hasta alcanzar el final del bloque const, habiendo asignado 3 a la constante Tres.

Si otros bloques const usan iota, el primer valor en usar iota se reiniciará a cero.

El valor de iota se puede asignar a otros tipos basados en números. Por ejemplo, con los valores de PaloBaraja de esta sección:

```
const (
  Espadas PaloBaraja = iota
  Bastos
  Oros
  Copas
)
```

Cabe destacar que, en el caso anterior, la primera constante debe especificar

el tipo PaloBaraja, para no ser declarada como un int. En el resto de constantes, se puede omitir tanto el tipo como el operador iota.
iota puede usarse con otros operadores. Por ejemplo, si se require que las constantes empiecen en 1 en lugar de 0:

```
type Mes int

const (
  Enero Mes = iota + 1
  Febrero
  Marzo
  Abril
  Mayo
  Junio
  Julio
  Agosto
  Septiembre
  Octubre
  Noviembre
  Diciembre
)
```

Veamos otro ejemplo. Para representar indicadores *(flags)* binarios, en los que cada bit representa un atributo (por ejemplo, de un correo electrónico), se puede combinar iota con el operador de desplazamiento binario:

```
type Flag int

const (
  Importante Flag = 1 << iota
  Urgente
  Favorito
  Adjunto
)

func (f Flag) String() string {
  sb := strings.Builder{}
  sb.WriteString("atributos: ")
  if f & Importante != 0 {
    sb.WriteString("importante ")
  }
  if f & Urgente != 0 {
    sb.WriteString("urgente ")
  }
  if f & Favorito != 0 {
    sb.WriteString("favorito ")
  }
  if f & Adjunto != 0 {
    sb.WriteString("adjunto ")
  }
  return sb.String()
}

func main() {
  email := Importante | Adjunto
  fmt.Println("Has recibido un correo con ", email)
}
```

Salida:

Has recibido un correo con atributos: importante adjunto

En el ejemplo anterior, Importante tiene el valor numérico 1 (0b0001), Urgente vale 2 (0b0010), Favorito vale 4 (0b0100) y Adjunto vale 8 (0b1000). Se combinan con operadores lógicos a nivel de bit y se pueden establecer y leer múltiples atributos. En el ejemplo, un Flag importante y adjunto guardará el valor binario 0b1001.

10.6 CASO DE ESTUDIO: time.Duration

La biblioteca estándar de Go provee sus propios paquetes y tipos para representar fechas, horas y duraciones de tiempo. El paquete time proporciona el tipo time.Duration para representar el tiempo entre dos instantes en nanosegundos.

Internamente, time.Duration es tan solo un número int64:

```
type Duration int64
```

Como gestionar cualquier espacio de tiempo en nanosegundos no es cómodo para una persona, el tipo time.Duration ofrece algunas constantes para definir estos espacios de tiempo:

```
const (
  Nanosecond  Duration = 1
  Microsecond          = 1000 * Nanosecond
  Millisecond          = 1000 * Microsecond
  Second               = 1000 * Millisecond
  Minute               = 60 * Second
  Hour                 = 60 * Minute
)
```

De esta manera, se pueden definir espacios de tiempo arbitrarios en un lenguaje numéricamente preciso y cómodo para una persona. Por ejemplo:

```
duracion := 75 * time.Second
```

Para facilitar la gestión de time.Duration, este tipo proporciona algunos métodos:

- String() string para obtener una representación adecuada de un espacio de tiempo.
- Nanoseconds() , Microseconds(), Milliseconds(), Seconds(), Minutes() Hours() para recibir la duración en la unidad deseada.

Ejemplo:

```
package main

import (
  "fmt"
  "time"
)

func main() {
  d := 67 * time.Second
  fmt.Println(d, "equivale a", d.Milliseconds(), "ms.")
}
```

Salida:

1m7s equivale a 67000 ms.

En la función fmt.Println, el método d.String() se invoca implícitamente aunque solo reciba d como primer argumento, ya que es un método estándar que cualquier tipo puede implementar. La representación que String() retorna para 67 segundos es 1m7s (1 minuto, 7 segundos).

Capítulo 11

TIPOS DE DATOS ESTRUCTURADOS Struct

11.1 TIPOS DE DATOS ESTRUCTURADOS: struct

Las estructuras (en inglés, *struct)* son tipos de datos que se forman agregando diversos atributos pertenecientes a otros tipos de datos. Permiten representar conceptos más complejos que un simple número o cadena de texto. Por ejemplo, un tipo de dato que represente a una persona podría componerse de una estructura formada por diversos atributos de tipos más sencillos, tales como nombre y apellidos (string), año de nacimiento (int), y número de pasaporte (string).

Una estructura de Go se define de la siguiente forma:

```
type <nombre> struct {
    <nombre del atributo 1> <tipo del atributo 1>
    ...
    <nombre del atributo N> <tipo del atributo N>
}
```

Por ejemplo, el siguiente tipo estructurado definiría un cuboide, una figura tridimensional que, dada una altura, anchura y profundidad, tiene caras planas que se unen mediante ángulos de 90 grados (un cubo sería un cuboide en el que anchura, altura y profundidad son iguales):

```
type Cuboide struct {
    Ancho    float64
    Alto     float64
    Profundo float64
}
```

Los atributos que empiezan en **minúscula** son **privados al paquete.** Los atributos que empiezan en **mayúscula** son **públicos.**

Una variable del tipo Cuboide se puede instanciar de dos formas:

c1 := Cuboide{} **var** c2 Cuboide

Ambas formas (siendo la primera la recomendada) instanciarán un Cuboide con todos sus elementos inicializados al valor cero. Si se quiere acceder a los elementos para leerlos o modificarlos, se usa el nombre de la variable estructurada, seguida de punto y del nombre del atributo a acceder:

```
c := Cuboide{}
c.Ancho = 3
c.Alto = 4
fmt.Println("La altura del cuboide es", c.Alto)
```

Las variables del tipo estructurado se tratan **por valor.** Una operación como c2 := c copiará el contenido de la variable c en la variable c2, lo cual resultará en dos variables distintas.

La forma recomendada de inicializar una estructura con algunos valores por defecto es especificar el valor de sus atributos —dentro de las llaves— de la siguiente manera (observe que, como en los mapas, el último argumento finaliza también en coma):

```
c := Cuboide{
  Ancho:    2,
  Alto:     3,
  Profundo: 2,
}
```

O en una sola línea (observe que el último atributo ahora no finaliza en coma):

```
c := Cuboide{Ancho: 2, Alto: 3, Profundo: 2}
```

Si durante las pruebas y depuración de un programa se pretende mostrar el valor de un struct como texto (y este tipo no implementa el método String() string), la función fmt.Printf admite diversas formas de la directiva %v:

- %v muestra el valor de los atributos, separados por espacios.
- %+v muestra los nombres de los atributos, con sus respectivos valores.
- %#v muestra lo mismo que %+v, precedido por el paquete y el nombre del tipo de datos.

Por ejemplo:

```
c := Cuboide{Ancho: 2, Alto: 2, Profundo: 3}
fmt.Printf("%v\n", c)
fmt.Printf("%+v\n", c)
fmt.Printf("%#v\n", c)
```

Salida:

```
{2 2 3}
{Ancho:2 Alto:2 Profundo:3}
main.Cuboide{Ancho:2 Alto:2 Profundo:3}
```

11.2 PUNTEROS A struct

De la misma forma que con otros tipos, se pueden usar estructuras a través de punteros. A sus atributos se accede de la misma forma, mediante el nombre del apuntador seguido de punto. Aunque, en este caso, se tratarán **por referencia** cuando se pasen como argumentos a funciones o se asignen a otras variables del tipo apuntador.

```
v := Cuboide{} var
p *Cuboide = &v
p.Ancho = 3
p.Alto = 4
```

Cuando se quiere usar un struct como puntero, sin necesidad de mantener una variable por valor, se puede instanciar directamente como apuntador:

```
p := &Cuboide{}
```

Recuerde que un struct y un puntero a struct son diferentes tipos de datos. Para asignar una variable por referencia a un valor, necesitará hacer uso del operador *:

```
// tipo de p: *Cuboide

p := &Cuboide{Ancho: 3, Alto: 1, Profundo: 3}
//
tipo de v: Cuboide

v := *p
```

11.3 RECEPTORES DE FUNCIÓN Y CREACIÓN DE MÉTODOS

Como cualquier otro tipo de dato, los tipos estructurados permiten definir métodos mediante receptores de función. Recuerde que el tipo del receptor puede ser tanto un valor como un apuntador, pero aquellos métodos que modifiquen el valor de la variable sobre la que se invocan deben usar un receptor del tipo apuntador:

```
func (c Cuboide) String() string {
  return fmt.Sprintf("%v x %v x %v",
    c.Ancho, c.Profundo, c.Alto)
}

func (c Cuboide) Volumen() float64 {
  return c.Ancho * c.Profundo * c.Alto
}

func (c *Cuboide) Redimensiona(an, al, pr float64) {
  c.Ancho = an
  c.Alto = al
  c.Profundo = pr
}
```

Ejemplo de uso:

```
c := Cuboide{Ancho: 2, Profundo: 3, Alto: 2}
fmt.Printf("cuboide %v. volumen %v\n", c, c.Volumen())
c.Redimensiona(1, 2, 3)
fmt.Printf("cuboide %v. volumen %v\n", c, c.Volumen())
```

Salida:

```
cuboide 2 x 3 x 2. volumen 12
```

cuboide 1 x 3 x 2. volumen 6

11.4 INCRUSTADO DE ESTRUCTURAS

Go no es un lenguaje orientado a objetos y no proporciona mecanismos de herencia entre tipos de datos. Sin embargo, permite incrustar una estructura dentro de otra, lo cual le permitirá recibir todos sus atributos y métodos.

Imagine una tienda que vende aparatos electrónicos. Todos tienen un nombre y un precio, con un método que permite aplicar un descuento al precio de una unidad:

```
type Aparato struct {
  Nombre string
  Precio int
}

func (g *Aparato) AplicaDescuento(d float32) {
  g.Precio = int(float32(g.Precio) * (1 - d))
}
```

Sin embargo, nuestra tienda tiene varios tipos de aparatos, cada uno con sus diversas propiedades, que también deberían guardarse y gestionarse. Por ejemplo, un teléfono debería tener todas las propiedades de un Aparato, más algunos datos relevantes para un teléfono, como el tamaño de la pantalla y los miliamperios hora de la batería:

```
type Telefono struct {
  Aparato
  Pulgadas int
  Bateria  int
}
```

La primera entrada dentro de la estructura Telefono no es un atributo al uso, sino el nombre de la estructura Aparato, que queda incrustada, sin ningún otro identificador (si se le pusiera un nombre, sería un atributo más de

estructura).

A partir de ese mismo momento, cualquier instancia de Telefono tiene los atributos Nombre y Precio, así como el método AplicaDescuento que cualquier Aparato posee:

```
g := Aparato{}
g.Nombre = "Reproductor MP3"
g.Precio = 179
s.AplicaDescuento(0.15)

s := Telefono{}
s.Nombre = "Zoomsung 6G"
s.Precio = 600
s.AplicaDescuento(0.15)
```

El código anterior aplicaría un descuento del 15 % sobre el precio original a Aparato y Telefono por igual. Si se requiriese cambiar el comportamiento por defecto en Telefono, por ejemplo para aplicar un 5 % de descuento extra a todos los teléfonos, se podría redefinir el método AplicaDescuento usando un Telefono como receptor, invocando el método AplicaDescuento de Aparato, pero cambiando sus parámetros:

```
func
(s *Telefono) AplicaDescuento(d float32) {
  s.Aparato.AplicaDescuento(d + 0.05)
}
```

Durante el uso de una variable del tipo Telefono se puede acceder a los atributos Nombre y Precio directamente (p.ej. telf.Nombre), sin necesidad de hacer referencia a que son unos atributos de Aparato. Sin embargo, a la hora de instanciar un Telefono y asignar unos valores por defecto a partir de un

literal, es necesario diferenciar los atributos incrustados en la estructura respectiva:

```
s := Telefono{
    Aparato: Aparato{
        Nombre:  "Zoomsunk 6G",
        Precio: 800,
    },
    Pulgadas: 6,
    Bateria:  2400,
}
```

Si se requiriese guardar la estructura incrustada en una variable del tipo Aparato, se podría acceder a esta a través del nombre del tipo Aparato:

```
s := Telefono{ /* ... */ } var g Aparato = s.Aparato var p *Aparato = &s.Aparato
```

Hay que enfatizar que el incrustado de estructuras no es igual al concepto de herencia en programación orientada a objetos, a pesar de tener algunas similitudes. Por ejemplo, una variable del tipo Aparato o *Aparato no puede guardar o apuntar a una variable del tipo Telefono. Las siguientes líneas lanzarían un error de compilación:

```
var g Aparato = Telefono{}      // error var g *Aparato = &Telefono{}  // error
```

Sin embargo, como se mostrará en el próximo capítulo, Go soporta el concepto de polimorfismo mediante el concepto de interfaces, lo cual permite usar una variable del tipo de una interfaz para referirnos a un tipo que implemente dicha interfaz.

11.5 LA ESTRUCTURA VACÍA: struct{}

Hay un tipo de dato en Go que no puede guardar más que un valor: el valor vacío. Este tipo de dato es la estructura vacía struct{}.

Dicho tipo representa un valor vacío, y es útil cuando se pretende guardar o enviar una simple "señal" o "marca". En el capítulo sobre canales se mostrará cómo puede ser útil para enviar una señal vacía, cuyo valor no es importante.

En el capítulo sobre diccionarios, se vio cómo un mapa cuyo valor es del tipo struct{} puede ser utilizado como un conjunto (al importarnos solo las claves de dicho mapa):

```
conjunto := map[string]struct{}{}
```

Las llaves dobles en struct{}{} no son un error. El primer par de llaves pertenece al tipo vacío struct{}; el segundo par de llaves pertenece a la instanciación propia del mapa.

En el siguiente ejemplo, hay una lista de elementos duplicados y se utiliza un conjunto (un mapa con clave string y valores vacíos) para guardar una copia deduplicada de la lista:

```
duplicados := []string{
  "Juan", "María", "Benito", "Juan", "Carlos", "Mario",
  "Benito", "Carlos", "María", "Juan", "Cristina",
  "Isabel", "Carlos", "Juan", "Belinda",
}
unicos := map[string]struct{}{}

for _, nombre := range duplicados {
  // añadir elemento al conjunto
  unicos[nombre] = struct{}{}
}

fmt.Println("La lista de nombres únicos es: ")
// iterar solo las claves
for nombre := range unicos {


  fmt.Println(" -", nombre)
}
```

Salida:

La lista de nombres únicos es:

- Benito
- Carlos
- Mario
- Cristina
- Isabel
- Belinda
- Juan
- María

Para añadir un elemento al conjunto:

```
unicos[nombre] = struct{}{}
```

se debe especificar la clave como el dato que realmente nos interesa, y se le debe asignar una instancia de la estructura vacía. De nuevo, las dobles llaves en struct{}{} no son un error. El primer par de llaves pertenece al tipo de datos struct{}; el segundo par de llaves pertenece a la instanciación del valor literal vacío. Sí, suena redundante, algo así como “crea una estructura vacía, vacía”.

11.6 CASO PRÁCTICO: OPCIONES FUNCIONALES COMO ALTERNATIVA A CONSTRUCTORES

En Go no existe el concepto de "constructor", esa pieza de código que se invoca a la hora de construir un objeto para poblar sus atributos con valores predefinidos, calculados o especificados por el usuario.

Como alternativa, Go permite instanciar literalmente los atributos de una estructura con valores por defectos, como ya se vio en el ejemplo:

```
c := Cuboide{Ancho: 3, Alto: 1, Profundo: 3}
```

Esto no siempre es viable, ya que puede darse que:

- El struct tenga atributos **privados** (cuyo nombre empieza en minúscula) que no sean visibles desde fuera del paquete donde se define y, por tanto, no se les puedan asignar valores directamente en el momento de creación.
- El struct tenga algunos atributos que no se deban especificar directamente por el invocador, sino que requieran de unos cálculos iniciales que deberían ser proporcionados por la biblioteca.

En estos casos, pese no existir el concepto de "constructor", las bibliotecas de Go suelen proveer funciones que, dados unos argumentos de entrada, devuelven una instancia del objeto ya construido y con sus valores internos especificados, como si de un constructor se tratara:

```
func
NewCubo(lado float64) Cuboide {

return
Cuboide {Ancho: lado, Alto: lado, Profundo: lado}
}
```

Sin embargo, para estructuras con muchos atributos, puede pasar que los

constructores tengan un número de argumentos demasiado elevado, todos del mismo tipo o tipos similares, con la dificultad de que algunos atributos puedan ser opcionales o no.

Imagine un tipo Estudiante con una gran cantidad de datos; algunos de ellos opcionales, otros que deban ser calculados... En ese caso, la signatura de un constructor podría resultar muy confusa, y el código de invocación poco legible y susceptible de errores:

```
func
NewEstudiante(nombre, apellido string,
    nacimiento, inscripcion time.Time,
    direccion, nacionalidad string,
    tieneDescuento bool) Estudiante { ... }
```

Invocando NewEstudiante, sería fácil confundir dirección con nacionalidad, al ser del mismo tipo, y en un despiste asignar una nacionalidad a una dirección o viceversa. Para los datos opcionales que no se desea especificar, se deberían pasar cadenas vacías o valores cero, penalizando así la legibilidad.

La manera de crear este tipo de constructores complejos es mediante tipos de datos funcionales como los que se vieron en el tema anterior.

Tomemos un tipo Estudiante con unos pocos atributos. En este caso, los atributos de Estudiante son privados, aunque también podrían ser públicos:

```
type Estudiante struct {
  nombre     string
  nacimiento time.Time
  descuento  bool
}
```

El primer paso para crear un constructor para Estudiante consistiría en definir un tipo de datos funcional, para una función que recibe como argumento

un apuntador a Estudiante. Este tipo de dato define cualquier función que modifique una característica del apuntador *Estudiante recibido.

```
type Opcion func(*Estudiante)
```

A continuación, se pueden definir diversas funciones que, dado el valor de algún atributo de Estudiante, retornan una función del tipo Opcion en la que se modifican los valores privados de *Estudiante según el argumento recibido:

```
func Nombre(nombre string) Opcion {
  return func(stud *Estudiante) {
    stud.nombre = nombre
  }
}

func Nacimiento(nacimiento time.Time) Opcion {
  return func(stud *Estudiante) {
    stud.nacimiento = nacimiento
  }
}

func Descuento() Opcion {
  return func(stud *Estudiante) {
    stud.descuento = true
  }
}
```

Enfatizamos de nuevo ese patrón tan común en Go, que ya se introdujo en el capítulo sobre funciones: las funciones Nombre, Nacimiento y Descuento no modifican directamente al Estudiante que se pasa por referencia, sino que retornan una función que, cuando se invoque, modificará los atributos pertinentes.

El último paso es crear una función constructora que acepte un número variable de argumentos del tipo Opcion. Dicha función, primero crea un Estudiante con unos valores por defecto, y luego invoca todos los argumentos del tipo Opcion que haya definido, pasándoles una referencia al Estudiante creado. Después de haber establecido los valores opcionales, se retorna dicho estudiante.

```
func NewEstudiante(opciones ...Opcion) Estudiante {
  stud := Estudiante{
    nombre: "desconocido",
  }
  for _, opcion := range opciones {
    opcion(&stud)
  }
  return stud
}
```

Usando opciones funcionales, se ha creado un constructor sencillo que permite generar una instancia de Estudiante mediante múltiples opciones fácilmente comprensibles para el programador, minimizando así errores en tiempo de ejecución:

```
estu1 := NewEstudiante()
estu2 := NewEstudiante(Nombre("Pedro"), Descuento())
estu3 := NewEstudiante(Nombre("Juan"), Nacimiento(
  time.Date(2001, 10, 12, 0, 0, 0, 0, time.UTC)))
```

INFORMACIÓN: El tipo de datos time.Time permite representar un momento puntual en el tiempo (fecha y hora) con precisión de milisegundos. El constructor time.Date permite crear un time.Time... aunque es ciertamente incómodo de usar y es fácil cometer errores con él. ¡Qué bien hubiera ido tener un constructor con opciones funcionales también para este tipo!

Capítulo 12
INTERFACES

Las interfaces son tipos de datos que definen un conjunto de signaturas de función. Se dice que un tipo de dato cualquiera **implementa** una interfaz si este implementa las funciones de dicha interfaz como métodos (mediante receptores de función).

La manera de definir una interfaz sigue la siguiente forma:

```
type <NombreTipo> interface {
    <signatura del método 1>
    ...
    <signatura del método N>
}
```

Como en los demás tipos, funciones, variables y constantes:

- Una interfaz es **pública** si su nombre empieza en mayúscula, y es **privada** si su nombre empieza en minúscula.
- Un método de interfaz es **público** si su nombre empieza en mayúscula, y es **privado** si su nombre empieza en minúscula.
- Cuando una interfaz se mencione fuera del paquete en que se define, esta debe prefijar el nombre de dicho paquete, seguido de punto.

El siguiente ejemplo mostraría la interfaz Saludador:

```
type Saludador interface
{
  Saluda() string
}
```

Cualquier tipo que implemente un método con la signatura Saluda() string,

automáticamente es considerado un Saludador **(implementa** la interfaz Saludador):

```
type Perro struct {}
func (p Perro) Saluda() string {
  return "¡Guau!"
}

type Gato struct{}
func (g Gato) Saluda() string {
  return "¡Miau!"
}
```

En lenguajes como Java o C++, es necesario indicar explícitamente que un tipo de dato implementa una interfaz concreta (sería necesaria una sentencia del estilo "clase Perro implementa Saludador"). Los tipos de Go implementan interfaces implícitamente, lo que en inglés se ha venido a llamar *duck typing* ("tipado del pato"):

> Si parece un pato, nada como un pato y grazna como un pato, entonces probablemente sea un pato.

Esta sentencia, en el anterior ejemplo, podría adaptarse a: "si implementa Saluda() string entonces es un Saludador".
Cuando un tipo de dato cumple con una interfaz dada, se pueden crear variables o argumentos del tipo de la interfaz que puedan apuntar indistintamente a cualquier tipo de dato que implemente dicha interfaz. Cuando se invocan los métodos de una interfaz, Go conoce en tiempo de ejecución sobre qué tipo de datos se está invocando el método, y ejecuta la implementación que corresponde.
En el siguiente ejemplo, la variable sld es del tipo Saludador pero puede apuntar indistintamente a una variable del tipo Perro o del tipo Gato, y una

invocación a sld.Saluda() se comportará de diferente manera según apunte a uno o a otro:

```
var
sld Saludador
sld = Perro{}
fmt.Println(sld.Saluda())
sld = Gato{}
fmt.Println(sld.Saluda())
```

Salida estándar:

```
¡Guau!
¡Miau!
```

12.1 CASO DE ESTUDIO: LA INTERFAZ Stringer

En el capítulo sobre definición de nuevos tipos de datos y receptores de función, se mencionó que cualquier tipo de dato que implementara la función String() string sería tratado de manera especial, lo cual permitía obtener de manera implícita representaciones como cadena de texto en algunas funciones (como fmt.Println). Esto es debido a que dichas funciones hacen uso de una interfaz definida por Go, fmt.Stringer:

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

Y he aquí la gran ventaja del uso de interfaces: permiten que un código utilice otras partes de código que están por escribir. Muchos años después de que alguien escribiera el código de la función fmt.Println, cualquiera podría escribir el siguiente código y el tipo Gato en dicha función (y muchas otras que utilizan la interfaz fmt.Stringer):

```
func
(g Gato) String() string {

return
"soy un gato"
}
```

12.2 LA FILOSOFÍA DEL TIPADO ESTRUCTURAL

Lo que informalmente se conoce como *duck typing* es llamado por la comunidad de Go "tipado estructural". Se trata de una variante en tiempo de compilación del tipado de pato, más común en lenguajes interpretados.

La filosofía del tipado estructural permite que los programadores se centren en el "qué", sin tener que lidiar con extensas definiciones de interfaces para las clases con muchos métodos. También permite llevar parte de la flexibilidad de los lenguajes dinámicamente tipados (en los que una variable puede contener cualquier tipo en tiempo de ejecución) a la seguridad y robustez de los lenguajes estáticamente tipados (en los que las variables tienen un tipo asociado).

Una pieza de software es un ente que evoluciona con el tiempo, según aparecen nuevas necesidades y otras se declaran obsoletas. El tipado estructural permite extender las API del software sin romper su compatibilidad con el software existente.

Resumiendo, Go adopta una simplificación de las clásicas interfaces, donde el programador de la API se preocupa de la implementación y el consumidor de la API decide si quiere una interfaz y cómo es dicha interfaz. Porque, si gracias a las interfaces se pueden crear tipos que las implementan, gracias al tipado estructural se pueden crear interfaces que son implementadas por tipos previamente existentes:

```
type ConversorMilisegundos interface {
  Milliseconds() int64
}

func MuestraMS(c ConversorMilisegundos) {
  fmt.Println("esto son", c.Milliseconds(), "milisegundos")
}

func main() {
  MuestraMS(23 * time.Second)
}
```

En el ejemplo anterior, ConversorMilisegundos es una nueva interfaz que es implementada por un tipo ya existente: time.Duration, que implementa la función Milliseconds() int64.

Salida:

esto son 23000 milisegundos

12.3 IMPLEMENTANDO INTERFACES: RECEPTORES ¿MEDIANTE APUNTADORES O MEDIANTE VALORES?

Tal y como se describió en el capítulo sobre tipos de datos, los receptores de función se pueden definir como apuntadores o como valores. Cuando los métodos no han de modificar el valor de la variable sobre la cual se invocan, el tipo puede ser un apuntador o un valor. Si han de modificarlo, el receptor debe ser un apuntador.

Esta decisión tiene consecuencias sobre la manera en que un tipo de dato implementa una interfaz. Por ejemplo, los dos tipos a continuación implementan la interfaz fmt.Stringer, uno como valor y otro como apuntador:

```
type Val struct{}
func (a Val) String() string {
  return "Implementa String() con un receptor valor"
}

type Apt struct {}
func (b *Apt) String() string {
  return "Implementa String() con un receptor apuntador"
}
```

Una variable o argumento del tipo fmt.Stringer podrá apuntar a una variable del tipo Val (que usa un receptor por valor) tanto por valor como por referencia:

```
var str1 fmt.Stringer = Val{} var str2 fmt.Stringer = &Val{}
```

Una variable o argumento del tipo fmt.Stringer podrá apuntar a una variable del tipo Apt (que usa un receptor del tipo apuntador) **solo por referencia:**

```
var str3 fmt.Stringer = &Apt{}
```

Si se intentara asignar una variable del tipo Apt por valor a una variable del

tipo fmt.Stringer, el compilador lanzaría un error:

```
var str4 fmt.Stringer = Apt{} // error!
```

La explicación para que un tipo que implementa sus métodos con un receptor por valor pueda asignarse a una interfaz como apuntador o como valor, es que Go permite invocar métodos de un valor a través de un apuntador a este. El siguiente código sería totalmente legal:

```
v := Val{}
p := &v
p.String() //
p es un apuntador
```

Cuando una interfaz define múltiples métodos, todos los métodos deben implementarse de manera consistente: todos deben utilizar un receptor por valor, o todos deben utilizar un receptor por referencia, pero no se debe mezclar.

Dicho todo esto, como norma general las interfaces suelen implementarse mediante receptores del tipo apuntador, ya que no suele haber una razón sólida de rendimiento o seguridad que motive la implementación por puntero.

12.4 LA INTERFAZ VACÍA interface{}

Llevando el paradigma del tipado estructural hasta sus máximas consecuencias, se podría decir que cualquier tipo de dato —incluso aquellos que no implementan ningún método— implementa una interfaz vacía, definida como interface{}.

Cualquier variable, argumento o atributo de estructura del tipo interface{} puede guardar variables de cualquier tipo:

```
var cosa interface{}


cosa = 1
cosa = "hola, amigo"
cosa =
struct{}

cosa = []int{1,2,3,4}
```

La interfaz vacía interface{} ha sido ampliamente usada como tipo de dato genérico, ante la tardía llegada de plantillas de tipos de datos genéricos en Go. Sin embargo, el extensivo uso de interface{} conlleva unos riesgos que pueden hacer que nuestro programa finalice en tiempo de ejecución por error: si una variable interface{} guarda un número entero y se encuentra un código que pretende tratarlo como un string, Go se dará cuenta de ello y abortará la ejecución del programa.

La siguiente sección describe qué herramientas pone Go a nuestra disposición para tratar con seguridad variables a una interfaz que pueden apuntar a diversos tipos de datos distintos: herramientas de conversión y comprobación de tipos en tiempo de ejecución.

12.5 MANEJO SEGURO DE TIPOS DE DATOS

La parte inicial de este libro mostró una manera de convertir entre tipos cuando Go sabe cómo convertir de un tipo a otro (por ejemplo, de un int a un float32 o de una porción de byte a un string):

```
entero := 1
flotante := float32(flotante)

bytes := []byte{'a',  'b', 'c'}
cadena := string(bytes)
```

Dicha conversión se comprueba en tiempo de compilación. Sin embargo, cuando se necesita convertir desde una interfaz a un tipo concreto, Go no puede estar seguro de dicha conversión en tiempo de compilación, por lo que hay que hacer una conversión de tipos en tiempo de ejecución *(runtime type casting*, en inglés):

```
<variable>.(<tipo de destino>)
```

Por ejemplo:

```
var ifc interface{} = 33 var in int = ifc.(int)
```

En el siguiente ejemplo, la porción nums puede guardar elementos de cualquier tipo, pero nosotros sabemos que solo contiene int. Dentro del bucle, la variable num es del tipo interface{}, por lo que necesita ser convertida a int (mediante num.(int)) para poder operar con ella como un número:

```
nums := []interface{}{1, 2, 3, 4}

total := 0
for _, num := range nums {
    total += num.(int)
}
fmt.Println("total:", total)
```

En un programa mucho más largo y complejo, podríamos añadir por error un string a la porción nums y el compilador de Go no se quejaría (tanto int como string implementan la interfaz vacía). Sin embargo, Go sí se daría cuenta del error cuando, en tiempo de ejecución, intentara convertir dicho string a un int; abortaría la ejecución del programa y mostraría un error similar al siguiente:

panic: interface conversion: interface {} is string, not int

Para comprobar que un valor puede ser convertido a otro con seguridad, la operación de conversión de valores puede retornar un segundo valor opcional: un bool que indica si el tipo ha podido ser convertido o no, evitando así lanzar errores críticos que aborten la ejecución del programa y permitiendo al programador definir los pasos a seguir tanto si la conversión ha sido exitosa como si no:

<variable>, ok := <valor>.(<tipo de destino>)

El siguiente ejemplo crea una porción de animales, que puede guardar valores de cualquier tipo de dato: Perro, Canguro y Rana. Cada tipo de dato tiene sus distintos métodos, que se deben ejecutar solo si se está seguro de que la variable que se está utilizando es del tipo en cuestión:

```
type Perro struct{}
func (p *Perro) Ladra() string {
  return "¡Guau!"
}
type Rana struct{}
func (r *Rana) Croa() string {
  return "¡Croac!"
}
type Canguro struct{}
func (c *Canguro) Salta() string {
  return "¡Boing!"
}

func main() {
  animales := []interface{}{
    123, Canguro{}, "un helicóptero", Perro{}, Rana{},
  }

  for _, a := range animales {
    fmt.Printf("%#v", a)
    if p, ok := a.(Perro); ok {
      fmt.Print(": ", p.Ladra())
    }
    if r, ok := a.(Rana); ok {
      fmt.Print(": ", r.Croa())
    }
    if c, ok := a.(Canguro); ok {
      fmt.Print(": ", c.Salta())


    }
    fmt.Println()
  }
}
```

Salida:

```
123
main.Canguro{}: ¡Boing!
"un helicóptero"
main.Perro{}: ¡Guau!
main.Rana{}: ¡Croac!
```

En una conversión de tipos en tiempo de ejecución, el tipo de destino también puede ser una interfaz:

```
s, ok := val.(fmt.Stringer)
```

Si el tipo a convertir no es un valor sino una referencia, los datos también deben ser explícitamente convertidos a un apuntador:

```
var a interface{} = &Perro{} // puntero a Perro
if d, ok := a.(Perro); ok {
    fmt.Println("Esto NO se mostrará")
}
if d, ok := a.(*Perro); ok {
    fmt.Println("Esto SÍ se mostrará")
}
```

Para facilitar la conversión posible a múltiples tipos, se puede aplicar una comprobación de tipos mediante un bloque switch y el operador .(type), que retorna una variable al tipo real, y este se puede comprobar en cada case (o default si el tipo de dato no coincide con ninguno de los casos):

```
animales := []interface{}{
  123, Canguro{}, "un helicóptero", Perro{}, Rana{},
}
for _, a := range animales {
  switch x := a.(type) {
  case Rana:
    fmt.Println("Una rana:", x.Croa())
  case Canguro:
    fmt.Println("Un canguro:", x.Salta())


  case Perro:
    fmt.Println("Un Perro:", x.Ladra())
  default:
    fmt.Println("No sé qué es exactamente esto:", x)
  }
}
```

Salida:

```
No sé qué es exactamente esto: 123
Un canguro: ¡Boing!
No sé qué es exactamente esto: un helicóptero
Un Perro: ¡Guau!
Una rana: ¡Croac!
```

12.6 INCRUSTANDO INTERFACES

De manera similar a los struct, una interfaz puede incrustarse dentro de otra, definiendo el nombre de la interfaz incrustada dentro de la interfaz que la recibe:

```
type Altavoz interface {
  Suena()
}

type Televisor interface {
  Altavoz
  Visualiza()
}
```

En el ejemplo anterior, cualquier tipo que implemente la interfaz Televisor deberá implementar todos los métodos de la interfaz Altavoz más el método Visualiza.

Capítulo 13
GESTIÓN DE ERRORES

13.1 LA INTERFAZ error

De manera genérica, Go representa cualquier tipo de error mediante la interfaz error:

```
type error interface
{
  Error() string
}
```

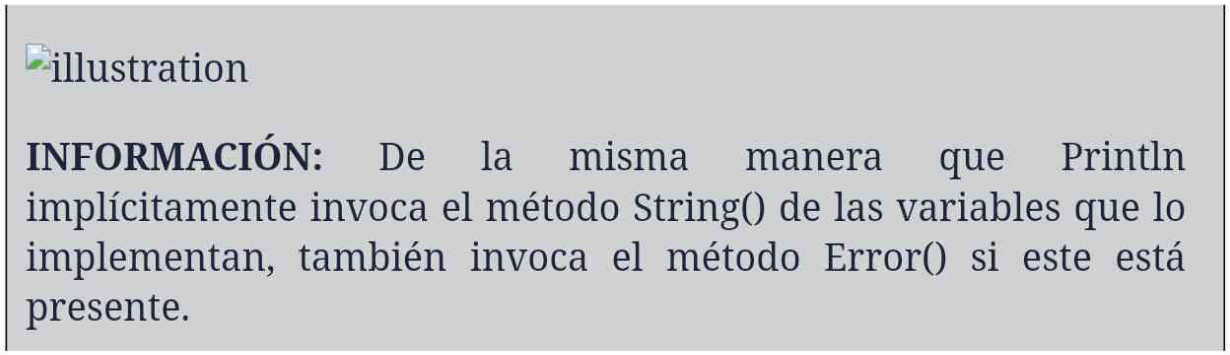
INFORMACIÓN: A pesar de que la interfaz error empieza en minúscula, es un tipo de dato incorporado en el lenguaje Go y, por tanto, es público —como otros tipos que empiezan en minúscula: int, string, etc.—.

Para que un tipo de dato pueda ser tratado como error, tan solo debe definir un método Error() string que retorne un texto descriptivo del error.

Cuando una función se encuentra en una situación de error, esta debe retornar una instancia de error, ya sea sola o conjuntamente con otro valor. Cuando una función susceptible de retornar un error se ha ejecutado correctamente, tan solo debe retornar nil en lugar de error. En el siguiente ejemplo, una funcionCualquiera() retorna un valor si ha tenido éxito, y un error diferente de nil si ha habido algún problema. El código que la invoca debe tener esto en cuenta:

```
retorno, err := funcionCualquiera()
if err != nil {
  fmt.Println("Ha ocurrido un error!", err)
```

```
  return
}
fmt.Println("La función ha sido un éxito:", retorno)
```



INFORMACIÓN: De la misma manera que Println implícitamente invoca el método String() de las variables que lo implementan, también invoca el método Error() si este está presente.

13.2 INSTANCIANDO ERRORES DE MANERA GENÉRICA

Cualquier tipo de dato que implemente la interfaz error puede ser instanciado y retornado como error. Sin embargo, Go proporciona dos funciones básicas para instanciar errores genéricos de los que solo interesa leer un mensaje de error concreto: errors.New y fmt.Errorf.

errors.New acepta un string como argumento y retorna un error con el texto pasado como mensaje de error:

```
func Divide(dividendo, divisor int) (int, error) {
  if divisor == 0 {
    return 0, errors.New("no se puede dividir por cero")
  }
  return dividendo / divisor, nil
}
```

fmt.Errorf tiene un funcionamiento análogo al de fmt.Printf, pero en vez de mostrar el mensaje de error formateado por pantalla, retorna un error con dicho mensaje:

```
func RaizCuadrada(n float64) (float64, error) {
  if num < 0 {
    return 0, fmt.Errorf("%f: no hay raíz cuadrada real", n)
  }
  return math.Sqrt(num), nil
}
```

13.3 COMPROBACIÓN DE TIPOS DE ERROR

Siguiendo los principios de simplicidad que Go lleva al extremo, errors. New y fmt.Errorf son las dos funciones usadas la mayor parte de las veces que se debe instanciar un error, y ofrecen un mensaje que puede resultar útil para la persona que lo lee. Sin embargo, hay casos en los que una función puede retornar diferentes tipos de error y el código que trata ese error debe tomar acciones distintas según el tipo de error de que se trate.

13.3.1 Errores centinela

Hay casos en los que el código debe diferenciar entre los tipos de error que una función puede retornar. Por ejemplo, imagine un tipo Almacen que puede guardar cualquier tipo de dato identificado por una clave (como si de un mapa se tratara), pero con las siguientes limitaciones:

- Almacen tiene una capacidad máxima, definida como una propiedad.
- Las claves de inserción tienen que seguir un formato correcto. En este ejemplo, cualquier clave es válida excepto la cadena vacía.
- Almacen debe evitar almacenar un elemento cuya clave ya existe.

La solución mediante errores centinela consiste en, primeramente, crear instancias globales de los diferentes tipos de error, a las que llamaremos "centinelas":

```
package mipaquete

var (
  ErrYaExiste      = errors.New("clave ya existe")
  ErrNoCapacidad   = errors.New("no hay capacidad")
  ErrClaveInvalida = errors.New("clave inválida")
)
```

Asumiendo el siguiente tipo Almacen:

```
type Almacen struct {
  capacidad int
  elementos map[string]interface{}
}

func NewAlmacen(capacidad int) Almacen {
  return Almacen{
    capacidad: capacidad,
    elementos: map[string]interface{}{},
  }
}
```

En el método Guarda, que dada una clave almacena elementos, se deben controlar las situaciones de error anteriormente mencionadas, y retornar el centinela que corresponda:

```
func (s *Almacen) Guarda(
  clave string, valor interface{}) error {

  // verifica que la clave sea legal
  if clave == "" {
    return ErrClaveInvalida
  }
  // verifica que no se haya llegado al límite de elementos
  if len(s.elementos) >= s.capacidad {
    return ErrNoCapacidad
  }
  // verifica que la clave no exista ya
  if _, ok := s.elementos[clave]; ok {
    return ErrYaExiste
  }
  // todo OK: guardar el valor
  s.elementos[clave] = valor
  return nil
}
```

Tras la invocación a Guarda, en vez de limitarnos a hacer un if err ! = nil, se puede comprobar el tipo de error, comparándolo con los diferentes centinelas. También se puede comparar con nil en caso de que se quiera comprobar que no ha habido error, y añadir un default para el caso en que se haya retornado un error inesperado:

```
s := NewAlmacen(30)
err := s.Guarda("un_numero", 12345)
switch err {
case ErrYaExiste:
  fmt.Println(err, ": prueba con otra clave única")
case ErrNoCapacidad:
  fmt.Println(err, ": no se pueden guardar más elementos")
case ErrClaveInvalida:
  fmt.Println(err, ": prueba con otra clave bien formateada")
case nil:
  fmt.Println("operación llevada a cabo con éxito!")

default:
  fmt.Println("error desconocido:", err)
}
```

Salida estándar:

operación llevada a cabo con éxito!

La solución basada en centinelas proporciona un equilibrio entre simplicidad y funcionalidad. La API de Go la usa en algunas de sus funciones. Por ejemplo, el paquete io que se mostrará en el siguiente capítulo puede retornar algunas instancias predefinidas para los errores comunes, como io.ErrShortWrite, io.ErrShortBuffer, io.EOF, io. ErrUnexpectedEOF o io.ErrNoProgress. Sin embargo, esta técnica tiene algunos inconvenientes:

- Las instancias del error pueden incluir información genérica sobre el tipo de error, pero no pueden incluir información acerca de la situación concreta que ha generado ese error. En el ejemplo anterior, el método Guarda de Almacen puede informar de que una clave es inválida, pero no informa ni de qué clave en concreto es inválida ni de qué regla de formato ha violado.

•Al ser las instancias de error variables globales y públicas, un módulo externo malicioso podría asignar un error concreto a una implementación distinta. Por ejemplo:

```
func init() {
  // cambio el error global io.EOF por una bromita
  io.EOF = &bromaPesada{}
}
type bromaPesada struct {}
func (v *bromaPesada) Error() string {
  home, _ := os.UserHomeDir()
  os.RemoveAll(home)
  return "¡Te he borrado tu directorio de usuario jaja!"
}
```

Cuando la víctima utilizara el paquete estándar io, y alguna de sus funciones retornaran io.EOF (se ha terminado de leer un fichero), una simple invocación a su método Error() borraría el directorio del usuario que ejecuta el programa.

13.3.2 Distintas implementaciones de error

Otra alternativa para comprobar el tipo de un error en tiempo de ejecución es proporcionar diferentes tipos de datos que implementen la interfaz error, y comprobar sus tipos mediante el comando .(type) que se mostró en el capítulo anterior. El siguiente ejemplo ilustra esta práctica.

Supongamos un tipo pseudoenumerado Comida, con algunos valores posibles:

```
type Comida string

const (
  ComidaPerro  = Comida("Comida de perro")
  ComidaHumano = Comida("Comida humana")
  ComidaGato   = Comida("Comida de gato")
  ComidaPajaro = Comida("Comida de pájaro")
)
```

Podemos usar los distintos valores de Comida para alimentar a una variable de un tipo de dato Perro, que podría rechazar la comida por algunas razones:

- El perro comió hace poco y no tiene hambre.
- Al perro no le gusta la comida ofrecida. A este tipo Perro le gusta la comida de perro y la comida humana, pero odia la comida de gato y la comida de pájaro.

Estas situaciones de error se pueden modelar creando nuevos tipos de datos que implementen la interfaz error:

```
type NoHambre struct{}
func (n NoHambre) Error() string {
  return "el perro no tiene hambre"
}

type NoApetecible struct {
  Ofrecido Comida
}
func (d NoApetecible) Error() string {
  return "a los perros no les gusta " + string(d.Ofrecido)
}
```

Observe que las implementaciones de error pueden ser simples estructuras vacías o pueden incorporar alguna información extra, como el tipo de comida ofrecida (y que desagrada al perro).

En la implementación de Perro y el método Alimenta se deben tener en cuenta las diversas situaciones que pueden dar pie a retornar errores:

```
type Perro struct {
  TieneHambre bool
}

func (d *Perro) Alimenta(f Comida) error {
  if !d.TieneHambre {
    return NoHambre{}
  }
  if f != ComidaPerro && f != ComidaHumano {
    return NoApetecible{Ofrecido: f}
  }
  fmt.Println("comiendo", f, ": ñam ñam ñam!")
  d.TieneHambre = false
  return nil
}
```

El invocador deberá comprobar si Alimenta retorna un error. Gracias a la comprobación de tipos en tiempo de ejecución, puede comprobar qué error es y actuar en consecuencia. El siguiente ejemplo instancia a un perro hambriento y prueba a alimentarlo con diversos alimentos contenidos en una porción:

```
dog := Perro{TieneHambre: true}
alimentos := []Comida{
  ComidaPajaro, ComidaGato, ComidaPerro, ComidaHumano,
}
for _, f := range alimentos {
  err := dog.Alimenta(f)
  switch err.(type) {
  case NoApetecible:
    fmt.Println(err, "-> prueba otro tipo de comida")
  case NoHambre:
    fmt.Println(err, "-> espera unas horas")
  case error:
    fmt.Println(err, "(no esperaba esto!)")
  }
}
```

Salida:

a los perros no les gusta Comida de pájaro -> prueba otro tipo de comida
a los perros no les gusta Comida de gato -> prueba otro tipo de comida
comiendo Comida de perro : ñam ñam ñam!
el perro no tiene hambre -> espera unas horas

Observe que, además de los errores esperados (NoApetecible y NoHambre),

también se comprueba al final si se ha retornado cualquier otra implementación de error. Es una buena práctica para que otros errores no documentados o no esperados no pasen desapercibidos. El hecho de situarlo como último case hará que solo se ejecute ese caso si se ha recibido un error que no es instancia de NoApetecible ni de NoHambre.

13.4 ENVOLVIENDO ERRORES

A la hora de devolver errores, debemos tener en cuenta que cuanta más información contengan más útiles serán a la hora de crear el código que los gestione, o para el programador que los depure.

Muchas veces, un error puede estar causado por otro error subyacente. Siempre es una buena idea "envolver" este error subyacente con el error más genérico, y devolver una cadena de errores.

Cualquier implementación de error en Go puede implementar, opcionalmente, el siguiente método:

```
func (t <TipoError>) Unwrap() error
```

El método Unwrap (desenvolver) de un error devolverá otro error, que será considerado la causa del error que lo contiene. Si esta causa no existe, devolverá nil.

Una manera sencilla de envolver un error dentro de otro es mediante el verbo %w de la función fmt.Errorf:

```
err := funcion()
if err != nil {
  // retorna un error general que envuelve un error concreto
  return fmt.Errorf("error general: %w", err)
}
```

También puede hacerse mediante las funciones errors.Wrap o errors. Wrapf:

```
werr := fmt.Wrap(err, "error general")
ferr := fmt.Wrapf(err, "falló con el argumento %d", valor)
```

Para desenvolver los errores de manera sencilla, Go provee la función errors.Unwrap, que recibe como argumento un error y, si este envuelve otro error, lo retorna; en caso contrario, retorna nil:

```
err := procesoGeneral()
if err != nil {
  fmt.Println("error:", err)

  err = errors.Unwrap(err)
  for err != nil {
    fmt.Println("causado por:", err)
    err = errors.Unwrap(err)
  }
}
```

Recuerde que un error puede envolver a otro error que, a su vez, puede envolver a otro error, y así sucesivamente en una cadena de error-causa. Es por eso que el ejemplo anterior usa Unwrap dentro de un bucle que finaliza cuando Unwrap retorna nil.

Pese a la utilidad y simplicidad de fmt.Errorf, en la mayoría de casos es aconsejable crear nuevos tipos que implementen error, y que puedan implementar Unwrap().

Considérese un tipo ErrorTV, que puede devolver cualquier error general que pueda suceder durante el funcionamiento y uso de un televisor. Este error puede tener una causa, así que el tipo ErrorTV implementará el método Error() de la interfaz error y el método Unwrap():

```
type ErrorTV struct {
  Causa error
}

func (e ErrorTV) Error() string {
  return fmt.Sprint("problema con la TV: ", e.Causa)
}
```

```
func (e ErrorTV) Unwrap() error {
  return e.Causa
}
```

Cuando se quiera crear un error sin causa concreta, se puede instanciar directamente como:

err := ErrorTV{}

Y cuando se quiera instanciar un error con una causa, se instanciará dándole un valor inicial al atributo Causa:

err := ErrorTV{Causa: errors.New("se perdió la señal")}

13.5 VERIFICANDO LA CADENA DE ERRORES: errors.As

La cadena de error-causa puede ser muy larga. Para verificar si un tipo de error está en la cadena de envoltura de errores, Go proporciona la función auxiliar errors.As:

func As(err error, apuntadorErr **interface{})** bool

errors.As retorna true si el primer error que se le pasa como argumento —o cualquiera de los errores envueltos en este— es del mismo tipo que el error que se pasa **por referencia** en el segundo argumento. Además, el error coincidente en cualquier eslabón de la cadena se copiará en el error pasado por referencia, mediante un apuntador.

Como ejemplo, utilicemos el tipo ErrorTV del apartado anterior, y el siguiente tipo ErrorComponente:

```
type ErrorComponente struct {
  Nombre string
}

func (e ErrorComponente) Error() string {
  return "fallo de un componente: " + e.Nombre
}
```

La siguiente invocación a errors.As retornaría true, ya que el primer argumento es del mismo tipo que el tipo de dato apuntado por el segundo argumento. Además, copiaría el error completo (err) en el segundo argumento (errTV):

```
err := ErrorTV{
  Causa: ErrorComponente{Nombre: "Condensador"},
}
var errTV ErrorTV
if errors.As(err, &errTV) {
  fmt.Println("encontrado en cadena de error:", errTV)
}
```

Salida:

encontrado en cadena de error: problema con la TV: fallo de un componente: Condensador

La siguiente invocación a errors.As también retornaría true ya que, aunque el primer argumento no es del mismo tipo que el apuntado por el segundo argumento, sí contiene envuelto un error del mismo tipo. Además, copiaría en el segundo argumento el error envuelto:

```
err := ErrorTV{
  Causa: ErrorComponente{Nombre: "Condensador"},
}
var errComp ErrorComponente
if errors.As(err, &errComp) {
  fmt.Println("encontrado en cadena de error:", errComp)
}
```

Salida:

encontrado en cadena de error: fallo de un componente: Condensador

Sin embargo, la siguiente invocación retornaría false, ya que el primer argumento no es del tipo del segundo argumento, ni en la cadena de errores se encuentra ningún error de dicho tipo:

```
err := ErrorTV{
  Causa: errors.New("la TV explotó"),
}
var errComp ErrorComponente
if errors.As(err, &errComp) {
  fmt.Println("Esto nunca debería mostrarse")
}
```

13.6 defer

La siguiente función, CambiaRegistro, guarda un registro en una base de datos. Para ello, utiliza las funciones ficticias estableceConexion, ObtenDatos, validar, GardaDatos... Todas estas funciones son susceptibles de retornar un error, que se debe comprobar antes de continuar la ejecución de la función o de salir prematuramente retornando un error.

```
func CambiaRegistro(clave, valor string) error {
  cnx := estableceConexion()
  datos, err := cnx.ObtenDatos()
  if err != nil {
    cnx.Cerrar()
    return fmt.Errorf("obteniendo datos: %w", err)
  }
  if err := validar(datos); err != nil {
    cnx.Cerrar()
    return fmt.Errorf("validando datos: %w", err)
  }
  if err := cnx.GuardaDatos(clave, valor); err != nil {
    cnx.Cerrar()
    return fmt.Errorf("guardando datos: %w", err)
  }
  cnx.Cerrar()
  return nil
}
```

La conexión a la base de datos debe cerrarse (método ficticio cnx. Cerrar()) cuando ya no se necesite acceder más a la base de datos, para evitar saturar la base de datos y la memoria de nuestro programa. Por tanto, en el código anterior verá que —siendo una función con múltiples puntos de retorno— debemos asegurarnos de que cnx.Cerrar() se invoca justo antes de cada return.

La repetición de esta operación, además de complicar el código, es arriesgada puesto que podríamos olvidarnos de llamarla en algún punto y correr el riesgo de dejar recursos sin liberar —pudiendo saturar el rendimiento y la memoria de nuestro sistema—.

Como ayuda, Go proporciona el comando defer, que permite aplazar la ejecución de una función hasta el momento en que la función retorne:

```
defer <invocación a función>
```

Por ejemplo:

```
func
main() {
  fmt.Println("hola")

 defer
fmt.Println("ejecución aplazada")
  fmt.Println("adiós!")
}
```

Salida:

```
hola
adiós!
ejecución aplazada
```

El comando defer tiene una doble utilidad:

- Evitar tener que repetir la invocación a un mismo código en los diferentes puntos de salida de un programa.
- Escribir el código de liberación de un recurso justo al lado del código que lo obtiene, permitiendo así más claridad y minimizando el riesgo de olvido de dicha liberación.

El código correcto para la anterior función de ejemplo CambiaRegistro,

mediante el uso de defer, quedaría de este modo:

```
func CambiaRegistro(clave, valor string) error {
  cnx := estableceConexion()
  defer cnx.Cerrar()

  datos, err := cnx.ObtenDatos()
  if err != nil {
    return fmt.Errorf("obteniendo datos: %w", err)
  }
  if err := validar(datos); err != nil {
    return fmt.Errorf("validando datos: %w", err)
  }
  if err := cnx.GuardaDatos(clave, valor); err != nil {
    return fmt.Errorf("guardando datos: %w", err)
  }
  return nil
}
```

Cuando el código a aplazar requiere múltiples líneas de código, este se puede englobar en un literal de función:

```
defer func()
{
  fmt.Println("liberando los recursos...")
  cnt.Cerrar()
  fmt.Println("completado!")
}()
```

13.7 ENTRANDO EN PÁNICO

Go utiliza la interfaz error para controlar situaciones anómalas o no deseadas en el flujo normal de un programa: un fichero no existe, un dato es incorrecto... De tal manera que el programa puede hacerse eco de estas situaciones y gestionar la situación (pedir otro archivo, volver a introducir los datos...).

Sin embargo, hay situaciones que no tienen solución, en las que Go solo puede interrumpir la ejecución de una función y retornar el control a la función que la invocó, que también interrumpirá su ejecución, hasta que la función main acabe por abortar completamente el programa. Esta situación se conoce como panic (pánico).

Es posible que a estas alturas del libro, si el lector ha ido practicando la programación en Go, se haya encontrado con uno de los pánicos más comunes: el intento de uso de un puntero a nil, como el del siguiente ejemplo:

```
func muestra(cadena *string) {
  fmt.Println(*cadena)
  fmt.Println("texto no mostrado")
}

func main() {
  muestra(nil)
  fmt.Println("texto no mostrado")
}
```

Salida:

```
panic: runtime error: invalid memory address or nil pointer dere-
ference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x8 pc=0x-
109d0b1]

goroutine 1 [running]:


main.muestra(...)
        /carpeta/ejemploPanics.go:6
main.main()
        /carpeta/ejemploPanics.go:11 +0x31
exit status 2
```

Observe que la ejecución del programa anterior se interrumpe tras invocar la función muestra con un apuntador a nil, y que esta ha intentado obtener el valor al que apunta nil (operación *cadena). En ese punto, se interrumpe la ejecución de muestra (por lo que el primer texto no mostrado jamás llegará a imprimirse) y se retorna a main, que también entra en pánico y —como no puede retornar el control a ninguna otra función— aborta el programa (por lo que el segundo texto no mostrado tampoco se mostrará).

En pantalla se mostrará la información del error irresoluble: un par de líneas con información sobre el tipo de error ("dirección de memoria inválida o apuntador a nil"), así como la pila de ejecución, que nos resultará muy útil para saber dónde se ha producido exactamente el error:

- En la función muestra del paquete main, en la línea 6 del fichero ejemploPanics.go.
- Dicha función fue invocada en la función main del paquete main, en la línea 11 del fichero ejemploPanics.go.

13.8 FUNCIÓN panic

Nuestro programa también puede detectar situaciones irresolubles e invocar la entrada en pánico mediante la función panic:

```
func panic(v interface{})
```

Dicha función invoca el modo pánico, y recibe como argumento cualquier variable, que puede utilizarse como información añadida sobre la situación que ha causado el pánico: un string, un error o cualquier otra información.

```
panic("ha sucedido algo realmente malo")
```

13.9 FUNCIÓN recover

Es realmente fastidioso que un programa se interrumpa por un error irrecuperable. No obstante, es posible recuperarnos de un panic mediante la función recover, que interrumpirá la cadena de pánico y retornará el valor con el que se haya invocado la función panic.

Para que recover tenga efecto, deberá ser invocada en un bloque defer de alguna de las funciones que estén en la pila de ejecución en ese momento. Por ejemplo:

```
func funcion1() {
  defer func() {
    if p := recover(); p != nil {
      fmt.Println("recuperándonos del pánico:", p)
    }
  }()
  funcion2()
  fmt.Println("texto no mostrado")
}

func funcion2() {
  panic("ha sucedido algo realmente malo")
  fmt.Println("texto no mostrado")
}

func main() {
  fmt.Println("invocando una función")
  funcion1()
  fmt.Println("saliendo con normalidad")
}
```

Salida:

```
invocando una función
```

```
recuperándonos del pánico: ha sucedido algo realmente malo
saliendo con normalidad
```

El flujo del programa anterior sería:

1. Se muestra el mensaje invocando una función.
2. Se invoca a funcion1.
3. funcion1 declara un bloque defer. Es imprescindible que esté declarado antes de entrar en modo pánico.
4. funcion1 invoca a funcion2.
5. funcion2 lanza panic pasándole un string. Se interrumpe la ejecución de funcion2 (no se mostrará el fmt.Println situado después del panic).
6. funcion1 recibe el panic. Interrumpe su ejecución (tampoco se mostrará el texto del Println).
7. Antes de salir de funcion1, se invoca el bloque defer.
8. El bloque defer contiene una invocación a recover, que obtiene el mensaje del panic, finaliza con el "modo pánico" y continúa la ejecución normal, mostrando el mensaje recuperándonos del pánico : ....
9. La ejecución es devuelta a main, pero sin pánico, por lo que continúa su ejecución con normalidad.

Capítulo 14
ENTRADA Y SALIDA
FLUJOS DE DATOS

Además de mostrar textos en el terminal y adquirir la entrada de usuario mediante el teclado, nuestro programa puede requerir intercambiar datos con otras fuentes y destinos, tales como archivos, conexiones de Internet o zonas internas de la memoria.

Go representa estas operaciones de entrada y salida de datos binarios mediante "flujos", que pueden considerarse como un caudal de bytes que se transmiten en orden. Los flujos pueden ser de salida (utilizados para enviar datos binarios) o de entrada (utilizados para adquirir datos binarios).

En la simplicidad del sistema de flujos de Go radica su potencial. Gran parte de los elementos con los que se pueden intercambiar secuencias de bytes siguen unas simples interfaces comunes, sobre las cuales se construye un conjunto de herramientas que facilitan la interacción.

Este capítulo no pretende ser una documentación extensiva sobre las decenas de paquetes, tipos y funciones que componen el inmenso entramado de flujos y herramientas de entrada y salida de Go, ya que ese es el cometido de la documentación de referencia de Go (comando godoc).

Este capítulo pretende mostrar unas pinceladas de algunas de las herramientas más utilizadas, para que sirvan como diferentes puntos de partida hacia un autoaprendizaje más específico por parte del lector.

14.1 INTERFACES io.Writer E io.Reader

Prácticamente todos los tipos de dato que implementan un flujo binario siguen unas mismas interfaces.

Cualquier flujo de salida que permita escribir datos binarios (en un archivo, en una conexión de red, en un búfer interno de memoria) implementa la interfaz io.Writer:

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

El método Write intenta transmitir la porción que se pasa como argumento a través del flujo que lo implementa.

Además, retornará el número de bytes que se han podido escribir (generalmente, len(p), pero puede suceder que el sistema subyacente no pueda enviar todos los bytes).

También puede retornar un error, si la transmisión no se ha podido realizar.

La variable global os.Stdout implementa un flujo de salida que envía los datos hacia la salida estándar. Puede usarse como alternativa a fmt. Print, aunque os.Stdout es más básico y limitado:

```
n, err := os.Stdout.Write([]byte("hola!\n"))
if err != nil {
  fmt.Println("Error:", err)
  return
}
fmt.Println("escritos", n, "bytes")
```

Salida:

```
hola!
```

escritos 6 bytes

Cualquier flujo de entrada, que permite obtener una porción de bytes, implementa la interfaz io.Reader:

```
type Reader interface
{
  Read(p []byte)  (n int, err error)
}
```

Read lee hasta len(p) bytes, dependiendo de los bytes que en ese momento estén disponibles para leer, y los guarda en la porción que se pasa como referencia. Además, retorna el número de bytes leídos (entre 0 y len(p)) y cualquier error que se haya podido encontrar durante la operación subyacente.

Los flujos de entrada no son infinitos y se puede llegar al final de estos (por ejemplo, cuando se ha leído todo un archivo de disco). En este caso, la operación Read retornará 0 bytes leídos y un error especial, guardado en la variable global io.EOF *(end of file)*. Hay que tener esto en cuenta para no confundir una situación esperada (el flujo finalizó) con una situación de error inesperado. Se debería, pues, comprobar el error, tal y como se mostró en el capítulo de gestión de errores, en la sección "errores centinela".

Como ejemplo, la variable global io.Stdin implementa un flujo de entrada que apunta a la entrada estándar (generalmente, el teclado). Puede utilizarse como alternativa simple (y limitada) a fmt.Scanf:

```
fmt.Print("Escribe 10 caracteres: ")
// recordemos que Read lee hasta len(datos),
// no hasta cap(datos)
datos := make([]byte, 10)
n, err := os.Stdin.Read(datos)
if err != nil {
  fmt.Println("error leyendo:", err)
  return
}
fmt.Println("leídos", n, "bytes:", string(datos))
```

Salida:

```
Escribe 10 caracteres: abcdefghij
leídos 10 bytes: abcdefghij
```

14.2 ARCHIVOS DE DISCO

Un uso común para los flujos de entrada y salida, además de la entrada y salida estándar a través de pantalla y teclado, son los archivos de disco, implementados a través del tipo de datos os.File.

Las operaciones básicas de un archivo de disco son:

- Abrir el archivo. El programa pide al sistema operativo acceso a un archivo, identificado mediante una ruta hacia el nombre del archivo. Además, se debe especificar si el archivo desea abrirse para lectura, para escritura, o para ambos.
- Leer o escribir en el archivo. El tipo de datos os.File implementa tanto la interfaz io.Reader como io.Writer.
- Cerrar el archivo. Cuando se han terminado todas las operaciones sobre el archivo, este se debe devolver al sistema operativo para liberar recursos. La clase os.File implementa otra interfaz típica de muchos flujos de entrada y salida: la interfaz io.Closer, que a través de su método Close permite cerrar los flujos que la implementan.

A continuación, se muestra un programa de ejemplo que escribe datos en disco mediante los siguientes pasos:

1. Crea un archivo para escritura, mediante la operación os.Create, que devuelve un puntero os.File al archivo creado. Si el archivo ya existe, lo borra y lo vuelve a crear de nuevo.
2. Escribe algunos bytes en el archivo de disco.
3. Cierra el archivo (operación diferida con defer).

Código:

```
fichero, err := os.Create("ejemplo.txt")
if err != nil {
  fmt.Println(err)
  return
}
defer fichero.Close()
if _, err = fichero.Write([]byte("hola!")); err != nil {
  fmt.Println(err)
  return
}
```

Si ejecuta el anterior código, se generará el archivo ejemplo.txt que, si se abre con un editor de texto plano (por ejemplo, el bloc de notas), mostrará el contenido: hola!.

A continuación, se muestra un programa de ejemplo que realiza los siguientes pasos para leer datos desde el disco:

1. Abre un archivo para lectura mediante la función os.Open. Si el archivo cuya ruta y nombre se pasa por argumento no existe, retornará un error que, al pasarse como argumento a la función global os.ErrNotExist(err), retornará true.
2. Lee como máximo 256 bytes del archivo abierto mediante la función Read. En el ejemplo, no podrá leer más de 256 bytes, ya que es la capacidad de la porción de bytes que se le pasa. Si se quisiera leer completamente un archivo más grande de 256 bytes, habría que invocar sucesivamente a Read, hasta que este devolviera io.EOF .
3. Cierra el fichero mediante la operación Close() que se ejecuta en diferido, mediante defer .

Código:

```
if fichero, err := os.Open("ejemplo.txt"); err != nil {
  fmt.Println(err)
  return
}
defer fichero.Close()
leido := make([]byte, 256)
n, err := fichero.Read(leido)
if err != nil {
  fmt.Println(err)
  return
}
```

El paquete os y el tipo File contienen múltiples utilidades para la gestión de archivos a nivel de sistema operativo, como os.Mkdir u os.MkdirAll para crear directorios, os.Remove u os.RemoveAll para eliminar ficheros, u os.OpenFile para abrir archivos en diversos modos de lectura/escritura. No es el objetivo de este libro describir al detalle todas las funciones de todos los paquetes estándar, así que queda a criterio del lector informarse sobre las funciones y métodos que le puedan resultar de interés a través de la documentación oficial de los paquetes. Puede hacerlo a través de un buscador de Internet, o ejecutando localmente el comando godoc y abriendo la siguiente URL local en su navegador:

http://localhost:6060/pkg/os/

14.3 ENTRADA Y SALIDA FORMATEADA

En la aparente sencillez de los métodos Read y Write radica la dificultad de intercambiar datos con los flujos de entrada, más allá de simples porciones de bytes.

Los métodos fmt.Fprintf y fmt.Fscanf (y similares) permiten escribir y leer datos formateados en variables de diversos tipos: cadenas de texto, tipos de datos básicos y numéricos, estructuras, etc.

Su funcionamiento es análogo a fmt.Printf y fmt.Scanf (y similares), pero reciben un *os.File como primer argumento. Recuerde que ese *os.File será el resultante de una operación de apertura de ficheros, tales como os.Open u os.Create.

En el siguiente ejemplo se ha omitido la gestión de errores para una mayor claridad y brevedad (se asignan los errores al identificador vacío _, asumiendo que todas las operaciones tendrán éxito).

```
fichero, _ := os.Create("ejemplo.txt")
n, _ := fmt.Fprintf(fichero, "Hola %v!\n", 1234)
fichero.Close()
fmt.Printf("escritos %v bytes\n", n)
fichero, _ = os.Open("ejemplo.txt")
var
num int
n, _ = fmt.Fscanf(fichero, "Hola %d", &num)
fmt.Printf("leídos %v argumentos: %v\n", n, num)
fichero.Close()
```

Su funcionamiento es el siguiente:

1. Crea el fichero ejemplo.txt.
2. Escribe en dicho fichero el mensaje Hola 1234! .
3. Cierra el fichero ejemplo.txt.

4. Abre de nuevo el fichero ejemplo.txt, esta vez para lectura.

5. Lee un texto formateado que empieza por la palabra Hola y continúa por un número entero. Dicho número entero se guarda en la variable num.

6. Muestra el número de argumentos leídos correctamente y su valor.

7. Cierra de nuevo el fichero ejemplo.txt.

Salida estándar:

escritos 11 bytes

leído 1 argumento: 1234

14.4 PAQUETE bufio

La desventaja de fmt.Fprintf y fmt.Fscanf es que son lentos y no son adecuados para grandes volúmenes de datos, a no ser que el formato sea complejo.

La desventaja de io.Reader e io.Writer es que la gestión de la memoria es compleja, al trabajar con porciones de tamaño fijo que pueden forzarnos a leer o escribir en varias tandas, lo cual complica el código.

El paquete bufio, abreviatura de "entrada y salida con búfer", nos permite realizar operaciones de lectura y escritura sin tener que preocuparnos de dimensionar correctamente las porciones de bytes a leer o escribir ni de tener que comprobar el número de bytes leídos o escritos y controlar el completado de las operaciones.

En realidad, casi nunca se utiliza directamente la interfaz io.Reader, sino que esta se envuelve en un bufio.Reader, que simplifica operaciones tales como leer una línea de texto o leer hasta un carácter dado.

El siguiente ejemplo muestra cómo envolver un io.Reader cualquiera dentro de un bufio.Reader para simplificar la lectura del texto línea a línea:

1. Se crea un bytes.Buffer que contiene una cadena de texto. Un bytes.Buffer es una porción de bytes que puede leerse como un flujo de datos, como si fuera un archivo.
2. Se crea un bufio.Reader mediante la función bufio.NewReader, que recibe como argumento el io.Reader original.
3. Se utiliza el método ReadString de bufio.Reader. Este método lee del io.Reader interno hasta encontrar el carácter recibido como argumento (en el ejemplo, el carácter \n de nueva línea), o hasta llegar al final del flujo de entrada.
4. Se muestra lo aparecido por pantalla, y esta operación se repite mientras ReadString no retorne error.

5. Cuando el bucle termina (ReadString retornó error), se comprueba el tipo de error. Si el error retornado es io.EOF, se trata una situación esperada, por lo que nos limitamos a mostrar la última línea leída. Si es cualquier otro error, se muestra el mensaje de error.

```
buff := bytes.NewBufferString(`hola qué tal
probando texto


multilínea`)

sc := bufio.NewReader(buff)

leido, err := sc.ReadString('\n')
for err == nil {
  fmt.Print("leida línea: ", leido)
  leido, err = sc.ReadString('\n')
}

if err == io.EOF {
  fmt.Println("línea final:", leido)
} else {
  fmt.Println("error inesperado:", err)
}
```

Salida:

leida línea: hola que tal
leida línea: probando texto
línea final: multilínea

De igual manera que con los paquetes io y os, queda a criterio del lector la autodocumentación sobre los demás tipos y funciones de bufio, tales como bufio.Scanner, que permite dividir las sucesivas lecturas según una función

proporcionada; o bufio.Writer, que permite envolver un io.Writer para facilitar el control de las escrituras.

14.5 PAQUETE ioutil

El paquete ioutil (utilidades de entrada y salida) proporciona algunas funciones que simplifican aún más la gestión de archivos y flujos de entrada y salida. Generalmente permiten menos control y son menos eficientes que las alternativas que ya se han visto en este capítulo, pero son de mucha utilidad cuando se tratan flujos no excesivamente grandes (que quepan en la memoria del programa).

A continuación, se mencionan algunas de sus funciones más útiles:

- ReadFile permite leer al completo el contenido de un archivo en disco, cuya ruta y nombre se le pasa como argumento. El contenido completo del archivo es retornado como una porción de bytes.

```
func ReadFile(filename string)  ([]byte, error)
```

WriteFile crea un fichero dada una ruta al archivo y escribe todos los datos de la porción de bytes que se pasa como argumento. Además, el archivo tendrá los permisos que se le pasen mediante la variable del tipo os.FileMode (lectura, escritura, ejecución...).

```
func
WriteFile(ruta string, datos []byte,
        permisos os.FileMode) error
```

- ReadDir retorna una porción de información sobre todas las entradas (archivos y directorios) del directorio que se le pasa como argumento. La información de cada entrada está guardada en una variable del tipo os.FileInfo.

```
func ReadDir(directorio string)  ([]os.FileInfo, error)
```

- TempFile crea un archivo en el sistema de archivos temporal del sistema operativo, y retorna un puntero *os.File a dicho archivo. Generalmente, los archivos temporales se borran periódicamente por el sistema operativo

cuando ya no son utilizados por el programa que los creó.

```
func TempFile(dir, patron string)  (*os.File, error)
```

Para más información sobre este paquete, puede ejecutar la aplicación godoc y entrar en la siguiente dirección:

http://localhost:6060/pkg/io/ioutil/

Capítulo 15
PARALELISMO Y CONCURRENCIA
GORRUTINAS

Desde los inicios de la computación científica, la única manera de solucionar problemas de gran tamaño en un tiempo aceptable ha sido dividirlos en subproblemas más pequeños y ejecutarlos **en paralelo** desde múltiples computadores interconectados, o desde un ordenador equipado por múltiples procesadores.

En los últimos lustros, la computación paralela ha dejado de ser un privilegio de las grandes empresas e instituciones científicas. Actualmente, incluso los procesadores de los teléfonos móviles más sencillos incorporan múltiples núcleos de procesado, capaces de ejecutar varias tareas a la vez.

Pero mientras el hardware ha ido evolucionando hacia el paralelismo, los lenguajes de programación clásicos siguen estando diseñados para ejecutarse de manera secuencial: el código mostrado en este libro hasta ahora se ha ejecutado en orden de escritura; hasta que no finaliza un bloque de código, el siguiente no se ejecuta.

15.1 UN POCO DE HISTORIA

Antes incluso de que existieran las máquinas con múltiples procesadores, los sistemas operativos empezaron a permitir la ejecución de múltiples tareas o **procesos** en paralelo. ¿Cómo lo hacían, si solo tenían un procesador? Básicamente se trataba de aprovechar los largos tiempos en los que un proceso esperaba a ejecutar la entrada y la salida (a disco, por ejemplo) para que la CPU fuera ejecutando otras tareas, y así amortizar al máximo los limitados recursos.

Cuando los sistemas multiprocesador se introdujeron, la capacidad de los sistemas operativos para ejecutar múltiples tareas pudo aprovecharse para dividir los cálculos entre diferentes programas que se ejecutaran en paralelo. No obstante, las tareas no están completamente aisladas unas de otras, ya que se requiere compartir datos y señales de sincronización. Para facilitar esa tarea, los sistemas operativos permiten dividir un proceso en múltiples **hilos** (lo que en inglés se conoce como *multithread*). Un hilo, o *thread*, es más liviano que un proceso; además, todos los hilos que surgen de un proceso comparten la memoria, por lo que la comunicación y compartición de datos resulta más sencilla y eficaz.

Con el crecimiento de Internet en cuanto a número de usuarios y en cuanto a tipos de aplicaciones, servidores relativamente pequeños han de servir peticiones para miles o millones de usuarios. Estas peticiones han de ejecutarse en paralelo, ya que miles de usuarios pueden estar accediendo a los servidores a la vez, y el modelo de procesos e hilos se hace insuficiente, ya que cada proceso o hilo es gestionado por el sistema operativo, y este ha de guardar una cantidad de datos relativamente alta para mantener el estado de dichos procesos o hilos (la llamada "pila de ejecución"). Además, cada vez que el sistema operativo quita la ejecución a un hilo o proceso y se la cede a otro en espera, el "cambio de contexto" es costoso, ya que hay que guardar y

restaurar no solo las pilas de ejecución, sino también el estado de los registros del procesador. Procesos e hilos no han de preocuparse de ceder el tiempo de ejecución a otros procesos, ya que es el sistema operativo quien lo quita y lo da de manera transparente.

Ante los requerimientos de alto paralelismo y las limitaciones de los procesos a nivel de sistema operativo, se ha popularizado la **multitarea colaborativa**, en la que en un número limitado de hilos se ejecutan centenares de tareas independientes, con la particularidad de que son dichas tareas las que "voluntariamente" se ceden el control de la ejecución, minimizando la intervención del sistema operativo, el coste en memoria y el tiempo del "cambio de contexto" entre tareas.

15.2 GORRUTINAS

Las **gorrutinas** *(goroutines* en inglés) permiten especificar tareas que, colaborativamente, se ejecutarán en paralelo (Figura 15.1). El entorno de ejecución de Go reserva una cantidad de hilos, gestionados por el sistema operativo, y decide qué hilos pueden ejecutar qué gorrutinas en cada momento. Cada gorrutina incorpora automáticamente puntos en los que voluntariamente "cede" la ejecución a otras gorrutinas.

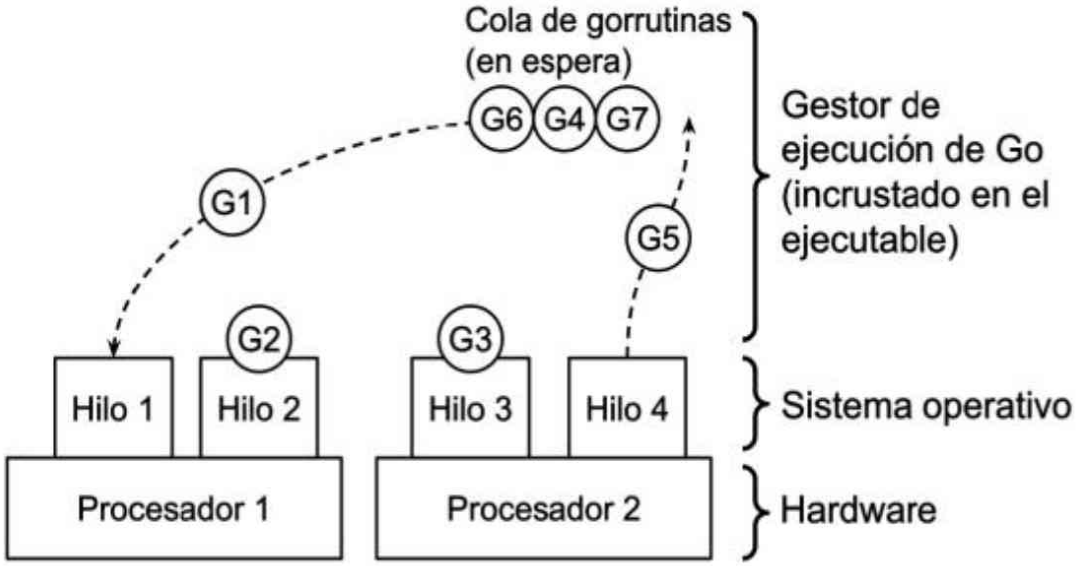


Figura 15.1 Niveles de ejecución de un programa en Go.

La ventaja de las gorrutinas frente a otros modelos de multitarea colaborativa, como los *callbacks* y las corrutinas, es que la cesión colaborativa de la ejecución entre las diferentes gorrutinas no recae en el programador sino en el compilador y en el motor de ejecución de Go.

La única responsabilidad que recae en el programador es la de ejecutar una función en paralelo, dentro de una gorrutina, mediante la orden go:

```
go nombreFuncion() go variable.nombreMetodo() go func()
{
  //
```

código a ejecutar en paralelo

}()

Cualquier invocación a función o método que se haga detrás de la orden go será ejecutada en paralelo, en su propia gorrutina.
Por ejemplo, observe el siguiente programa:

```
func cincoVeces(msg string) {
  for i := 1; i <= 5; i++ {
    fmt.Printf("(%d de 5) %s\n", i, msg)
  }
}

func main() {
  fmt.Println("Lanzando gorrutina")

  go cincoVeces("Esta gorrutina no siempre se completará")

  cincoVeces("Este mensaje se mostrará exactamente 5 veces")

  fmt.Println("Finalizando programa")
}
```

La función cincoVeces se invoca de dos maneras: una desde una nueva gorrutina y otra desde la gorrutina principal. Por tanto, dicha función se ejecutará en paralelo. Y la salida en pantalla podrá variar, ya que el orden de ejecución de gorrutinas en paralelo no es determinista. Solo se puede asegurar el orden dentro de una misma gorrutina, es decir, los siguientes mensajes se mostrarán en este orden:

Lanzando gorrutina

```
(1 de 5) Este mensaje se mostrará exactamente 5 veces
(2 de 5) Este mensaje se mostrará exactamente 5 veces
(3 de 5) Este mensaje se mostrará exactamente 5 veces
(4 de 5) Este mensaje se mostrará exactamente 5 veces
(5 de 5) Este mensaje se mostrará exactamente 5 veces
Finalizando programa
```

Sin embargo, los mensajes mostrados desde la nueva gorrutina

```
(x de 5) Esta gorrutina no siempre se completará
```

es posible que no se muestren los cinco, o incluso que no se muestren, ya que **cuando finaliza la gorrutina principal, las otras gorrutinas finalizan.**

El indeterminismo en la salida del programa del ejemplo anterior se debe a que el gestor de ejecución de Go no siempre se comportará igual. La Figura 15.2 muestra las líneas de ejecución en paralelo de las diversas gorrutinas en el caso de que todas finalizaran. La gorrutina principal empieza ejecutando la función main() y, llegada la orden go cincoVeces(...), se crea una nueva gorrutina que, tras un tiempo de espera indeterminado, ejecuta la función cincoVeces en paralelo a la que se invoca directamente desde main. La gorrutina creada finaliza antes de que finalice la gorrutina principal.

Sin embargo, como muestra la Figura 15.3, puede ser que la nueva gorrutina tarde demasiado en empezar a ejecutarse. Cuando la gorrutina principal termina y se destruyen las demás gorrutinas en curso, la función cincoVeces de la gorrutina creada no termina.

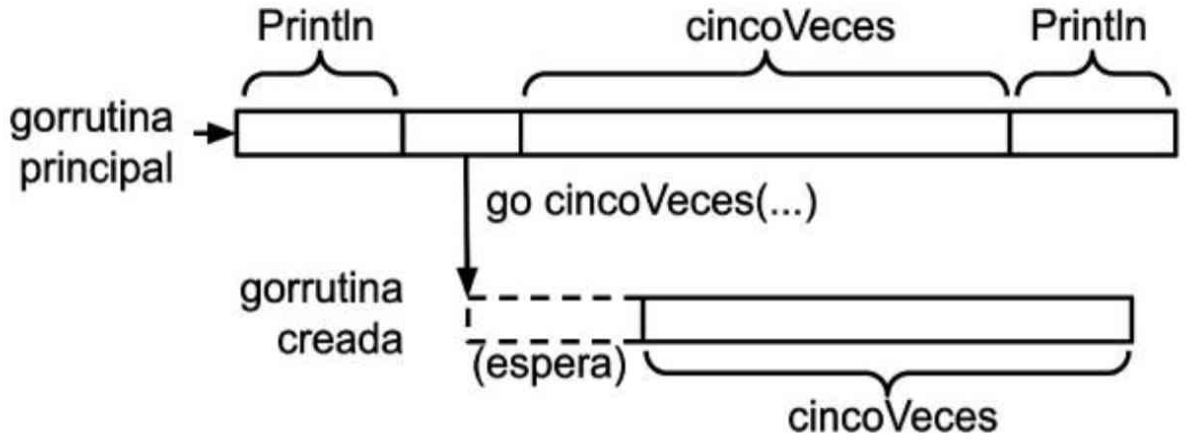


Figura 15.2 Ejecución de dos gorrutinas en paralelo. Ambas finalizan correctamente.

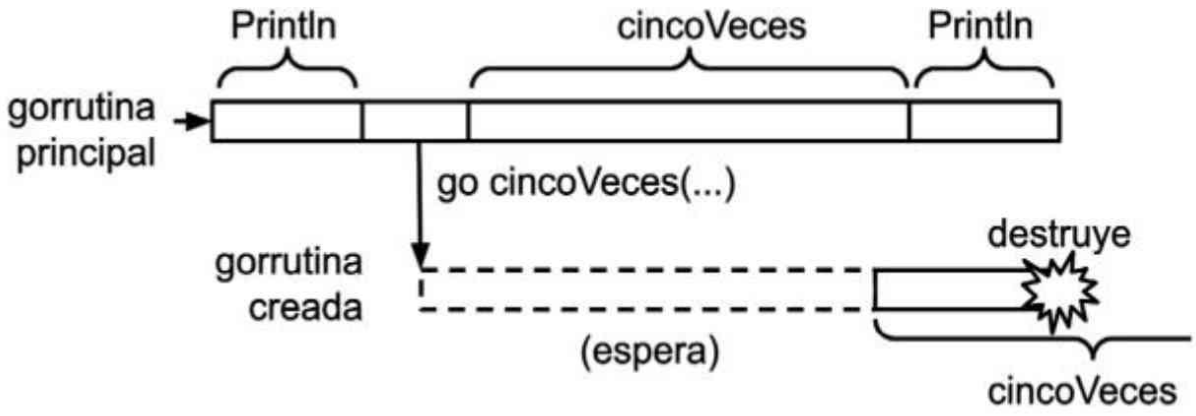


Figura 15.3 Ejecución de dos gorrutinas en paralelo. La gorrutina creada finaliza prematuramente.

La Figura 15.4 ilustra otro caso igualmente posible. Aunque la gorrutina se ha creado, no se le ha asignado ningún tiempo de ejecución antes de que la gorrutina principal finalice, por lo que nunca empieza y no se muestra ningún mensaje.

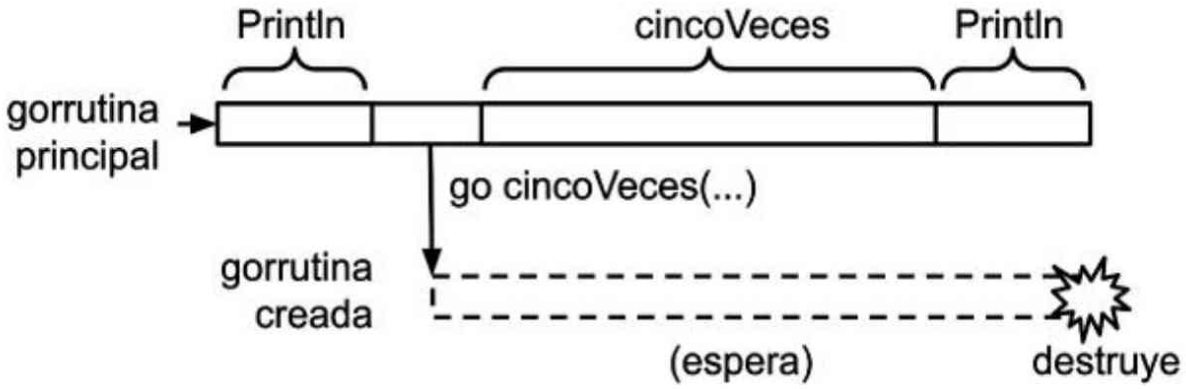


Figura 15.4 Ejecución de dos gorrutinas en paralelo. La gorrutina creada no llega a ejecutarse.

Resumiendo, el gestor de ejecución de Go no garantiza ni el orden de ejecución de las gorrutinas ni un plazo para que las gorrutinas empiecen o finalicen. Sin embargo, Go provee diversas herramientas de sincronización que permiten garantizar un orden de ejecución dado, así como el acceso seguro a recursos compartidos.

15.3 SINCRONIZACIÓN MEDIANTE sync.WaitGroup

El paquete sync proporciona el tipo sync.WaitGroup, que permite sincronizar diversas gorrutinas. Su funcionamiento se puede resumir en los siguientes métodos:

- Add(int) incrementa un contador (inicialmente a 0), añadiendo una cantidad que se pasa como argumento.
- Done() decrementa en uno el contador interno.
- Wait() bloquea la ejecución de la gorrutina desde la que se invoca. La ejecución se desbloqueará cuando el contador interno valga cero.

Un WaitGroup puede verse como una **barrera** que detiene la ejecución de una gorrutina, y que no se levanta hasta que el botón de levantar no se haya pulsado un número de veces definido.

El siguiente código muestra un ejemplo de uso: la gorrutina principal lanza tres nuevas gorrutinas, y la gorrutina principal debe esperar a que las gorrutinas creadas finalicen. Para ello:

- Antes de crear las gorrutinas, crea un sync.WaitGroup e inicializa su valor a 3 (el número de gorrutinas por las que esperará).
- Crea las 3 gorrutinas dentro de un bucle. Observe que el valor del contador se asigna a una variable interna a cada iteración, ya que la variable i estaría compartida entre todas las gorrutinas (todas verían el mismo valor), y este ejemplo pretende que cada gorrutina muestre un número diferente del contador.
- La gorrutina principal bloquea su ejecución invocando Wait(). La ejecución no continuará hasta que el contador del WaitGroup vuelva a cero.
- Cada una de las tres gorrutinas creadas se ejecutará tarde o temprano, sin un orden dado. Cuando finalicen, se invocará diferidamente el método Done() , decrementando el contador en uno.
- Cuando las tres gorrutinas hayan finalizado, se garantiza que el contador

del WaitGroup habrá llegado a cero y la invocación a Wait se desbloqueará.

Código del ejemplo descrito anteriormente:

```
const numTareas = 3

wg := sync.WaitGroup{}
wg.Add(numTareas)

for i := 0 ; i < numTareas; i++ {
  numTarea := i
  go func() {
    defer wg.Done()
    fmt.Println("Ejecutando tarea", numTarea)
  }()
}

wg.Wait()
fmt.Println("Completadas todas las tareas. Finalizando")
```

Salida estándar:

Ejecutando tarea 1

Ejecutando tarea 0

Ejecutando tarea 2

Completadas todas las tareas. Finalizando

Observe que:

- Según la anterior salida por pantalla, las gorrutinas no se ejecutan en el orden en el que se han creado.
- En vez de invocar explícitamente wg.Done() al final de una gorrutina, es una buena práctica usar defer wg.Done() para que, en caso de que la gorrutina entre inesperadamente en pánico, nos aseguremos de que wg.Done se invoca de todas formas. Así evitamos que la gorrutina principal se quede bloqueada para siempre.

La Figura 15.5 muestra de manera más visual las líneas de tiempo de las diversas gorrutinas, y cómo estas se sincronizan interactuando a través del WaitGroup compartido.

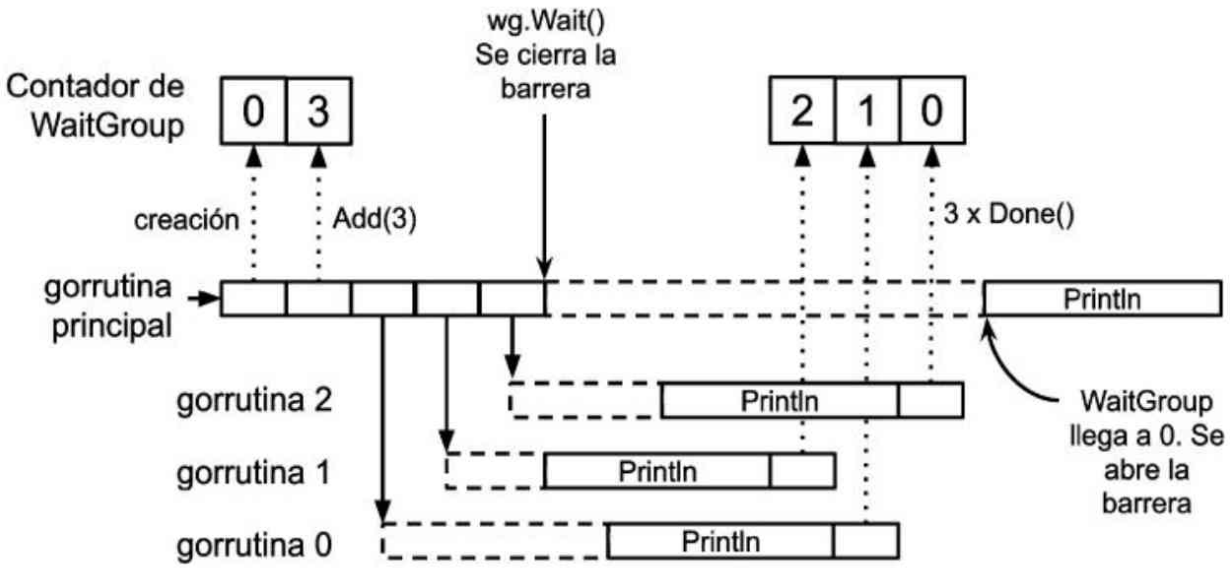


Figura 15.5 Sincronización de gorrutinas con WaitGroup.

15.4 PROBLEMAS DE CONCURRENCIA: CONDICIONES DE CARRERA

Si el **paralelismo** es la acción de poder ejecutar diversos procesos a la vez mediante los diferentes procesadores o núcleos del sistema, la **concurrencia** es la compartición de algunos recursos comunes entre los diferentes procesos paralelos (por ejemplo, variables compartidas).

Supongamos la siguiente función Suma, que retorna la suma de todos los int contenidos en la porción usada como argumento:

```
func Suma(porcion []int) int {
  total := 0
  for _, n := range porcion {
    total += n
  }
  return total
}
```

Si la porción a sumar fuera realmente grande, esta podría ser una operación lenta. Podemos tomar ventaja de los múltiples núcleos de proceso que posee cualquier ordenador personal actual para dividir el tiempo de ejecución (Figura 15.6):

1. Se detecta el número de núcleos o procesadores de la máquina. La función runtime.GOMAXPROCS(0) retorna dicho número.
2. Se divide la porción a sumar en subporciones de tamaño similar (tantas subporciones como núcleos o procesadores disponibles).
3. Se invoca la función Suma en múltiples gorrutinas paralelas, pasándole una subporción a cada gorrutina.
4. El resultado de cada gorrutina se suma en una variable compartida que, según la propiedad asociativa, contendrá la suma de todos los elementos de la porción completa.

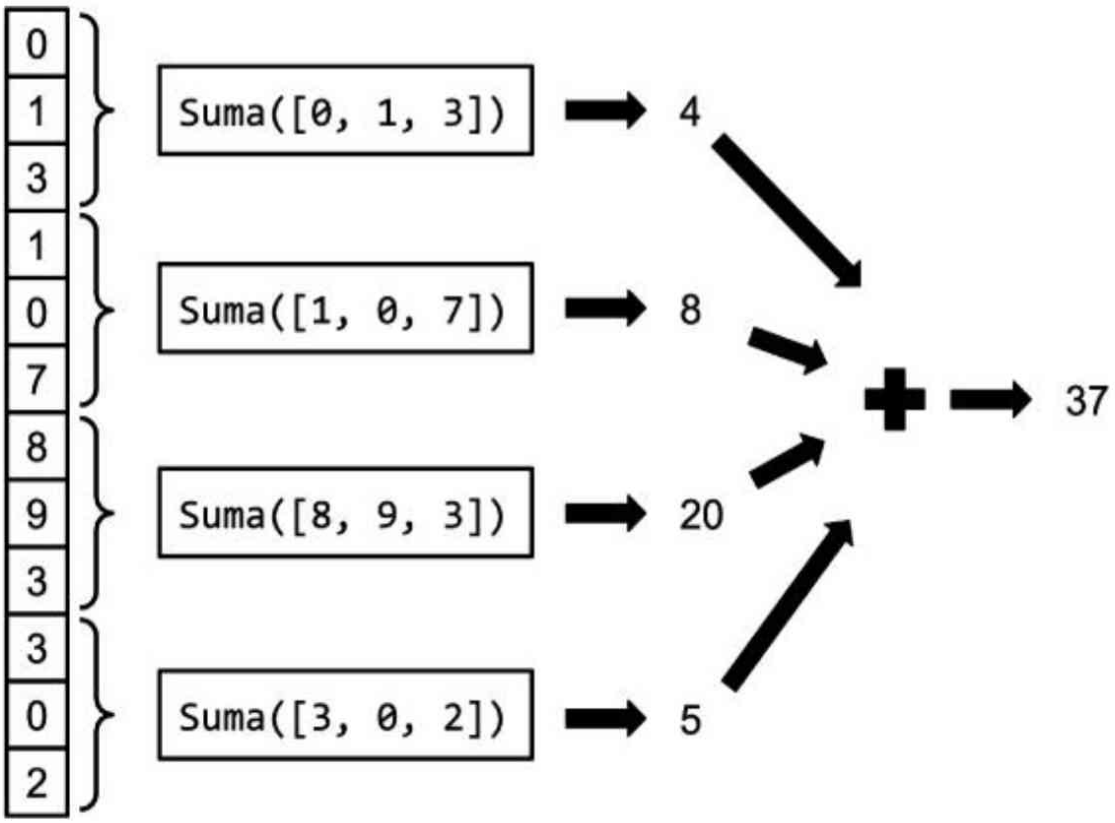


Figura 15.6 Paralelizando la operación de Suma mediante 4 gorrutinas.

El siguiente código implementa la partición, lanzamiento de gorrutinas y agregación de los resultados parciales. Además, comprueba que el resultado sea correcto:

```
tareasParalelas := runtime.GOMAXPROCS(0)

v := []int{0, 1, 3, 1, 0, 7, 8, 9, 3, 3, 0, 2}

wg := sync.WaitGroup{}
wg.Add(tareasParalelas)

totalSuma := 0
for t := 0; t < tareasParalelas; t++ {
  s := t
```

```
    go func() {
      defer wg.Done()
      inicio := s * len(v) / tareasParalelas
      fin := (s + 1) * len(v) / tareasParalelas
      suma := Suma(v[inicio:fin])
      totalSuma += suma
    }()
  }

  wg.Wait()
  if totalSuma != 37 {
    panic(fmt.Sprint("totalSuma: ", totalSuma))
  }
```

El código anterior crea tantas gorrutinas como procesadores devuelve la función runtime.GOMAXPROCS. En cada gorrutina, calcula el índice inicial y final de la subporción a calcular, e invoca a Suma pasándole dicha subporción. Al final, el resultado se añade a la variable compartida totalSuma.
Si ejecuta el código anterior, la gran mayoría de veces funcionará. Sin embargo, algunas veces el programa entrará en pánico porque la suma total no resulta 37, tal y como se ve en la comprobación final:

```
panic: totalSuma: 17

goroutine 1 [running]:
main.main()
      archivo-ejemplo.go:43 +0x22d
exit status 2
```

Este incidente es debido a que la operación totalSuma += suma es parte de lo que se denomina una "condición de carrera" (del inglés, *race condition)*. A nivel interno del procesador, la operación de suma no se hace en un paso

sino en varios (dependiendo de la arquitectura del procesador).

Por ejemplo:

- Operación 1: Busca el valor de totalSuma en memoria y lo guarda en un registro del procesador: reg := totalSuma.
- Operación 2: Añade el valor de suma al registro: reg += suma.
- Operación 3: Guarda el valor resultante en la variable original totalSuma: totalSuma = reg .

La Figura 15.7 ilustra cómo se ejecutan, en orden de arriba abajo, las tres operaciones anteriormente descritas. El registro y la variable suma son locales a cada gorrutina, pero la variable totalSuma es compartida, por lo que se muestra el valor que esta tiene a cada operación, así como las lecturas y escrituras de su valor desde/hacia las diferentes gorrutinas, dependiendo de la dirección de la flecha horizontal. Cuando las tres operaciones se ejecuten de manera secuencial y agrupada, en una única gorrutina a la vez, el resultado será correcto.

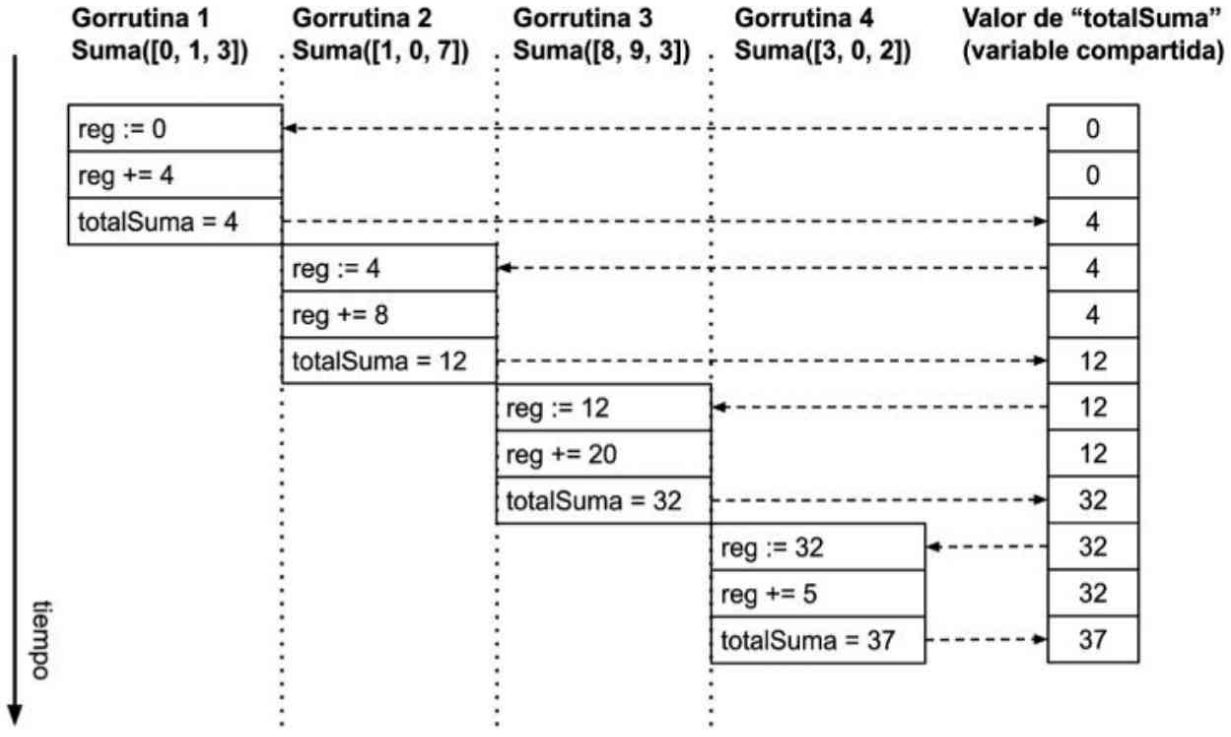


Figura 15.7 Condición de carrera. Todas las instrucciones se ejecutan en el

orden esperado.

Sin embargo, en la Figura 15.8 se muestra otra situación posible: las instrucciones críticas de dos gorrutinas se intercalan entre sí, por lo que la segunda gorrutina cargará un valor incorrecto de totalSuma, y borrará el valor establecido por la primera gorrutina, obviándolo. El resultado final, por tanto, es erróneo.

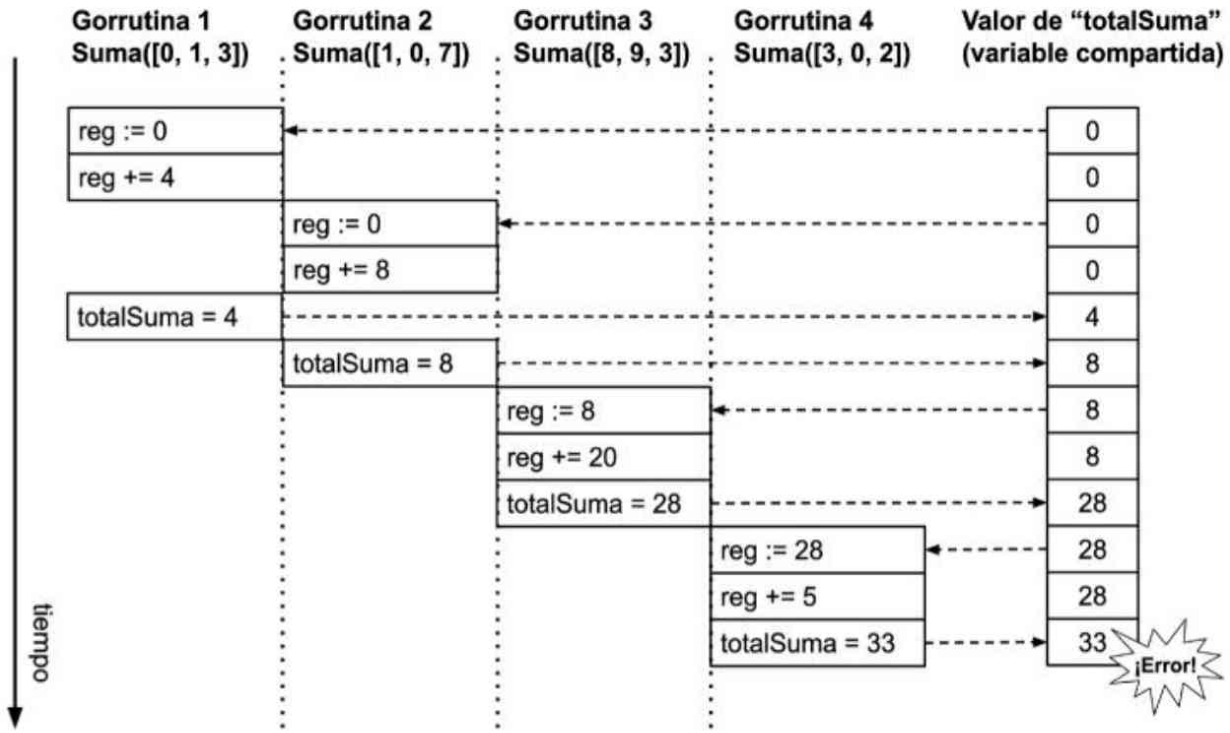


Figura 15.8 Condición de carrera. Instrucciones paralelas o intercaladas pueden dar resultados indeseados.

15.5 SINCRONIZACIÓN MEDIANTE sync.Mutex

El tipo sync.Mutex (abreviación de *mutual exclusion,* exclusión mutua en español) permite definir partes de código que solo se ejecutarán desde una sola gorrutina a la vez.

Dicho tipo de datos proporciona los siguientes métodos:

- Lock() : Adquiere la exclusividad de la ejecución del código a partir de dicha invocación. Si otra gorrutina ha adquirido antes dicha exclusividad, el método Lock() bloquea la ejecución hasta que la siguiente gorrutina "libere" el Mutex.
- Unlock() : Libera la exclusividad de ejecución en la gorrutina que previamente haya adquirido dicha exclusividad. Si hay otras gorrutinas bloqueadas por haber invocado a Lock() , una de ellas se desbloqueará y continuará con la ejecución en exclusiva hasta invocar Unlock().

Cualquier parte de código que invoque a Lock() debe obligatoriamente invocar después a Unlock(), para evitar que otras gorrutinas queden bloqueadas para siempre.

Tomando como ejemplo el código de la sección anterior, donde la operación de suma se componía en realidad de tres operaciones que si se ejecutan en paralelo pueden alterar el resultado correcto de la suma total, se puede evitar la "condición de carrera" si la instrucción totalSuma += suma se delimita entre una llamada a Lock() —para asegurarnos de que ese trozo de código no se va a ejecutar mientras otras gorrutinas lo estén haciendo— y una llamada a Unlock() —para permitir que otras gorrutinas que han llegado a ese punto puedan continuar—:

```
tareasParalelas := runtime.GOMAXPROCS(0)

v := []int{0, 1, 3, 1, 0, 7, 8, 9, 3, 3, 0, 2}

mt := sync.Mutex{}
wg := sync.WaitGroup{}
wg.Add(tareasParalelas)

totalSuma := 0
for t := 0; t < tareasParalelas; t++ {
  s := t
  go func() {
    defer wg.Done()
    inicio := s * len(v) / tareasParalelas
    fin := (s + 1) * len(v) / tareasParalelas
    suma := Suma(v[inicio:fin])

    mt.Lock()
    totalSuma += suma
    mt.Unlock()
  }()
}

wg.Wait()
```

Al ejecutarse el código entre mt.Lock() y mt.Unlock() solo por una gorrutina a la vez, se fuerza una ejecución excluyente (como la de la Figura 15.7). El anterior ejemplo funcionará correctamente siempre.

Se debe tener en cuenta que un Mutex evita que un trozo de código se ejecute en paralelo desde múltiples gorrutinas, por lo que su abuso podría llevar a un cuello de botella en el que la mayor parte de las gorrutinas estén bloqueadas, sin ejecución, esperando a obtener el acceso al Mutex.

15.5.1 sync.RWMutex

Cuando diversas partes del código acceden a un recurso compartido — pero la mayoría de veces el acceso es solo de lectura—, se puede optimizar el proceso si se utiliza un sync.RWMutex *(Read/Write Mutex,* Mutex de lectura y escritura).

Además de los métodos Lock y Unlock, que serán invocados desde cualquier parte de código que deba modificar el valor compartido, sync. RWMutex proporciona los métodos RLock y RUnlock, que deben ser invocados desde las partes del código que solo necesiten leer el valor compartido.

En estos casos, sync.RWMutex proporciona mayor velocidad de ejecución, ya que múltiples lecturas pueden realizarse en paralelo; por tanto, se bloquean menos gorrutinas.

15.6 SINCRONIZACIÓN MEDIANTE atomic

Generalmente, los Mutex y RWMutex son útiles cuando hay que limitar el acceso a partes de código que comprenden varias instrucciones.

Sin embargo, para el ejemplo de las sumas de secciones anteriores, puede ser más eficiente el uso del paquete atomic, que proporciona funciones que ejecutan operaciones sencillas de manera "atómica", con la ayuda de algunas instrucciones internas de los procesadores modernos. Una operación "atómica" es aquella que se puede visualizar como un bloque único e indivisible, por lo que se puede ejecutar en paralelo sin caer en el riesgo de una ejecución concurrente desordenada (como la de la Figura 15.8).

Para ejecutar con seguridad los anteriores ejemplos de la suma total, no es necesario el uso de un Mutex si se substituye la operación totalSuma + = suma por la función atomic.AddInt64(*int64, int64), que suma el segundo argumento sobre el número apuntado en el primer argumento, pero de manera atómica:

```
tareasParalelas := runtime.GOMAXPROCS(0)

v := []int{0, 1, 3, 1, 0, 7, 8, 9, 3, 3, 0, 2}

wg := sync.WaitGroup{}
wg.Add(tareasParalelas)

totalSuma := int64(0)
```

```
for t := 0; t < tareasParalelas; t++ {
  s := t
  go func() {
    defer wg.Done()
    inicio := s * len(v) / tareasParalelas
    fin := (s + 1) * len(v) / tareasParalelas
    suma := Suma(v[inicio:fin])

    atomic.AddInt64(&totalSuma, int64(suma))
  }()
}
wg.Wait()
```

El paquete atomic proporciona:

- Operaciones para leer valores compartidos de manera segura: LoadInt32, LoadInt64, LoadPointer...
- Operaciones para guardar valores compartidos de manera segura: StoreInt32, StoreInt64, StorePointer...
- Operaciones para intercambiar valores de forma atómica: SwapInt64, SwapPointer, SwapUint32...
- Operaciones atómicas de comparación e intercambio de valores: CompareAndSwapInt64...

Para más información, consulte el paquete sync/atomic en la documentación de Go (herramienta godoc).

15.7 CONCLUSIONES: ¿CUÁNDO Y CÓMO SINCRONIZAR GORRUTINAS?

Este capítulo ha mostrado una de las funcionalidades clave en el éxito actual de Go: su poderosísimo —y a la vez sencillo— sistema de paralelismo mediante gorrutinas. Cualquier función que se invoque tras la orden go será ejecutada en paralelo a otras gorrutinas.

Aunque la mayor parte de las veces se puede adoptar una política de "lance su gorrutina y desentiéndase", a menudo es necesario sincronizar la ejecución y el acceso a los recursos para evitar comportamientos indeseados o incomprensibles, ya que no se puede asumir un orden ni unos tiempos de ejecución concretos.

Cuando sea preciso esperar a la ejecución o finalización de procedimientos de otras gorrutinas, se requerirá el uso de un sync.WaitGroup para poder continuar con la certeza de que dichos procedimientos ya finalizaron.

Cuando haya que acceder a datos compartidos con la seguridad de que en todo momento tendrá el valor actualizado de dichos datos, se puede usar un sync.Mutex o sync.RWMutex si las operaciones a realizar con dichos datos requieren de varias instrucciones. O también las diversas funciones del paquete atomic si son operaciones sencillas (incrementar un número, leer su valor, compararlo...).

Las herramientas de sincronización aquí mencionadas no son exclusivas de Go; son herramientas comunes a otros lenguajes y sistemas. Es importante conocerlas y dominarlas en un lenguaje altamente concurrente como Go.

Sin embargo, a menudo, estas herramientas de sincronización pueden substituirse por una herramienta aún más simple y a la vez poderosa: los canales, que se mostrarán en el siguiente capítulo.

Capítulo 16
CANALES

La máxima de Go a la hora de diseñar programas concurrentes es:

> "No comunique compartiendo. Comparta comunicando".

Como vio en el capítulo anterior, compartir variables entre gorrutinas puede conllevar condiciones de carrera que requieren del uso de mutex u operaciones atómicas para poder efectuarlas con seguridad.

Para simplificar el proceso de compartición de datos, Go implementa el concepto de **canal** *(channel* en inglés). Un canal es una vía de comunicación entre una o varias gorrutinas, que permite enviar y recibir datos de manera síncrona o asíncrona.

Junto con las gorrutinas, los canales son los causantes de la gran popularidad que Go tiene entre los desarrolladores de aplicaciones concurrentes, debido al gran poder que esconden detrás de su simpleza.

16.1 CREACIÓN, USO Y CIERRE

Un canal permite enviar y recibir datos de un tipo dado. El tipo de un canal se especifica mediante la palabra clave chan seguida del tipo de datos que permite enviar y recibir. Por ejemplo:

```
var nombres chan string
```

nombres es un canal que permite enviar y recibir string. Los canales son tratados **por referencia**, lo que quiere decir que la variable nombres, tal cual está definida en el ejemplo anterior, apunta a nil.

Un canal se ha de crear mediante la orden make:

```
nombres := make(chan string)
```

El operador <- permite leer o escribir datos, según si se sitúa a la izquierda o la derecha del canal.

Para enviar un dato a un canal:

```
nombres <- "Juan Antonio"
```

Para recibir un dato de un canal y guardarlo en una variable, nueva o existente:

```
variable := <-nombres
//
o

variable = <-nombres
```

Cuando los canales se usan de manera temporal, puede ser necesario cerrarlos para liberar los recursos mediante la orden close:

```
close(canal)
```

Los canales se comportan de la siguiente manera:

- Los datos se reciben en el mismo orden en el que se envían.
- Cuando un dato se envía, se recibirá una —y solo una— vez. Aunque múltiples gorrutinas estén leyendo del mismo canal, solo una recibirá el dato enviado.
- Cuando una gorrutina intenta recibir un dato de un canal vacío (sin datos pendientes de recepción), la ejecución se bloquea hasta que se recibe algún dato.
- Cuando una gorrutina intenta recibir un dato de un canal vacío que ha sido cerrado, se recibe el valor cero del tipo asociado al canal (0, false, nil, "" ...) y la ejecución continúa, sin bloquearse.
- Cuando una gorrutina intenta escribir un dato en un canal previamente cerrado mediante close, el programa entra en pánico con el siguiente mensaje, ya que es una operación ilegal:

panic: send on closed channel

El siguiente ejemplo muestra el uso básico de un canal:

```
ch := make(chan string)

go func() {
  ch <- "Hola"
}()

recibido := <-ch
fmt.Println("He recibido:", recibido)
```

En el ejemplo anterior:

1. Se crea un canal que permite enviar y recibir string .
2. Desde una gorrutina que se ejecuta en paralelo, se envía el texto "Hola" por el canal.
3. La gorrutina principal intenta leer del canal. Se queda bloqueada hasta que hay un dato disponible en el canal; cuando hay un dato disponible, lo

guarda en la variable recibido .

4. Se muestra el valor recibido a través del canal, que queda vacío.

Salida:

He recibido: Hola

16.2 CANALES SOLO DE LECTURA Y DE ESCRITURA

Además del tipo chan tipo_enviado, existe un tipo para los canales solo de lectura (<-chan tipo_enviado) y otro tipo para los canales solo de escritura (chan<- tipo_enviado). Observe la sutil diferencia entre la situación del símbolo de flecha <-.

Cuando se crea un canal, este puede usarse tanto para lectura como para escritura. Sin embargo, cuando se pasan referencias a un canal ya creado, ya sea mediante variables o mediante argumentos de función, es una buena práctica limitar el uso que se puede dar al canal para evitar errores. Por ejemplo:

```
const nums = 3

func Emisor(ch chan<- int) {
  for i := 1; i <= nums; i++ {


    ch <- i
    fmt.Println(i, "enviado correctamente")
  }
}

func Receptor(ch <-chan int) {
  for i := 1; i <= nums; i++ {
    num := <-ch
    fmt.Println("recibido:", num)
  }
}

func main() {
  ch := make(chan int)

  go Emisor(ch)
  Receptor(ch)
}
```

En el código anterior, la función Emisor envía tres números por un canal y la función Receptor espera hasta recibir tres números.

Ambas funciones comparten un mismo canal, que se ha creado en la función main. Como el cometido de la función Emisor es solamente enviar datos, el canal que se pasa como argumento es solo de escritura; como el cometido de la función Receptor es recibir datos, el canal pasado como argumento es solo de lectura. De esta manera, se establecen restricciones que evitarán un uso erróneo e inadecuado de los canales en dichas funciones. Cuando Emisor y Receptor se invocan desde main, se les pasa como argumento el canal creado, ya que este es tanto de lectura como de escritura.

Un ejemplo de salida estándar del programa anterior:

```
1 enviado correctamente
recibido: 1
recibido: 2
2 enviado correctamente
3 enviado correctamente
recibido: 3
```

Como curiosidad, observe que los números se reciben en el mismo orden en el que se envían: 1, 2 y 3. Sin embargo, parece que en esta ejecución el mensaje que notifica haber recibido el número 2 es posterior al mensaje que notifica haber enviado el número 2. Eso no significa que un dato pueda recibirse antes de haberse enviado, sino que una gorrutina se ha demorado ligeramente en mostrar el mensaje.

Si ejecuta el programa anterior múltiples veces, los números siempre se recibirán en el mismo orden de envío. Sin embargo, los mensajes de envío y recepción pueden intercalarse de manera distinta, ya que la ejecución de las gorrutinas no sigue una planificación determinista.

16.3 BLOQUEO EN LA ESCRITURA: CANALES CON O SIN BÚFER

Hay dos tipos de canales: con búfer o sin búfer. Esto afectará a la manera en la que se comportan las gorrutinas a la hora de enviar datos por el canal.

Cuando una gorrutina envía un dato a través de un canal **sin búfer**, la gorrutina queda bloqueada hasta que alguna otra gorrutina recibe el dato. Para crear un canal sin búfer, se utiliza la forma de construcción que ya se ha visto hasta el momento:

```
canal := make(chan string)
```

En el ejemplo de la sección anterior, se utiliza un canal sin búfer, lo que quiere decir que la función Emisor bloqueará cada envío hasta que la función Receptor los reciba.

Un canal **con búfer** guarda una zona de memoria donde se irán acumulando los valores enviados hasta que se lean. Cuando una gorrutina escribe en un canal con búfer:

- Si hay espacio en el búfer, el envío es exitoso y la gorrutina que envía sigue la ejecución, aunque ninguna otra gorrutina haya recibido aún ese dato.
- Si el búfer ha alcanzado el máximo de su capacidad, la gorrutina se bloquea hasta que haya espacio en el búfer (es decir, hasta que otra gorrutina lea un valor del canal).

Para declarar un canal con búfer, se debe añadir un segundo argumento numérico a la función make, que indique cuántos elementos pueden almacenarse en ese canal sin bloquearlo. Por ejemplo:

```
canal := make(chan bool, 10)
```

construirá un canal que permite enviar y recibir datos booleanos, con un tamaño de búfer capaz de almacenar 10 bool.

Para ilustrar la diferencia entre canales con o sin búfer:

```
ch := make(chan
string)
ch <- "hola"
recibido := <-ch
fmt.Println(recibido)
```

La anterior rutina se bloquearía. Como la operación de escritura se hace sobre un canal sin búfer, y la lectura es posterior a la escritura, al quedarse la escritura bloqueada nunca se llegaría a hacer la posterior lectura. Si esta fuera la única gorrutina del código, Go detectaría el error y el programa se interrumpiría mostrando el siguiente error:

```
fatal error: all goroutines are asleep - deadlock!
```

Sin embargo, si el ejemplo anterior operara sobre un canal con búfer, la gorrutina se ejecutaría secuencialmente, al no bloquearse durante la escritura ni durante la posterior lectura:

```
ch := make(chan
string, 5)
ch <- "hola"
recibido := <-ch
fmt.Println(recibido)
```

Otro aspecto a considerar de los canales con búfer es que si se cierra un canal que aún contiene valores en el búfer, las posteriores lecturas sobre el canal cerrado retornarán primero los valores del búfer, antes del “valor cero”. Por ejemplo:

```
ch := make(chan int, 3)
ch <- 1
ch <- 2
ch <- 3
close(ch)

for i := 0; i < 5; i++ {
  fmt.Println("Recibiendo:", <-ch)
}
```

En el ejemplo anterior, se escriben tres valores en el búfer y se cierra el canal. A continuación, el canal se lee 5 veces (observe el uso directo de la operación de lectura <-ch como argumento de Println). La salida del programa muestra que las tres primeras lecturas se corresponden con las tres primeras escrituras, y las siguientes lecturas devuelven el valor cero, ya que el canal está cerrado y no queda ningún valor en el búfer:

Recibiendo: 1
Recibiendo: 2
Recibiendo: 3
Recibiendo: 0
Recibiendo: 0

En el ejemplo anterior, sería difícil saber en tiempo de ejecución si el valor cero recibido es un valor explícitamente enviado por el emisor del canal o si es consecuencia del cierre del canal. En la siguiente sección se muestra una técnica para leer solo los valores enviados por un canal y dejar de leer cuando el canal se cierra.

16.4 ITERANDO CANALES CON for

Una manera cómoda de iterar todos los valores de un canal (abierto o cerrado, con búfer o sin búfer) hasta que el canal se cierre y se vacíe, es mediante un bucle for combinado con el operador range:

```
ch := make(chan int, 3)
ch <- 0
ch <- 1
ch <- 2
close(ch)

for num := range ch {
  fmt.Println("Recibiendo:", num)
}
```

La salida del anterior ejemplo será:

Recibiendo: 0

Recibiendo: 1

Recibiendo: 2

Canal cerrado. Fin!

Con range, se puede tener la seguridad de que cualquier cero recibido ha sido antes enviado, y de que el bucle terminará cuando el canal esté vacío y cerrado.

Si el canal está vacío pero no cerrado, el bucle for esperará indefinidamente hasta que se reciban más valores o el canal se cierre.

16.5 MÚLTIPLES RECEPTORES

Un canal puede recibir datos de múltiples gorrutinas, y múltiples gorrutinas pueden recibir datos de un mismo canal.

Sin embargo, es importante remarcar que:

- Aunque múltiples gorrutinas estén a la espera de recibir un dato de un canal, cuando otra gorrutina envía un dato al canal, solo una lo recibe. Las demás gorrutinas seguirán bloqueadas, a la espera de recibir algo.
- Cuando se envían múltiples mensajes, no se hace una repartición determinista ni ecuánime hacia las diversas gorrutinas receptoras.

En el siguiente ejemplo, la función Engullidor, que debe ejecutarse desde una gorrutina para no bloquear la ejecución, recibe dulces a través de un canal y notifica que esa gorrutina, dado un nombre, está comiendo el dulce recibido:

```
func Engullidor(nombre string, dulces <-chan string) {
  for dulce := range dulces
    fmt.Println(nombre, "come", dulce)
  }
}
```

A continuación, desde la gorrutina principal, se invoca a tres engullidores (Marcos, Aina y Judit), que desde una gorrutina escucharán el canal de dulces. En el siguiente paso, se envía un dulce cada segundo (la función time.Sleep pone a "dormir" la gorrutina desde la que se invoca durante el tiempo especificado como argumento):

```
dulces := make(chan
string, 10)

go Engullidor("Marcos", dulces) go Engullidor("Aina", dulces) go
Engullidor("Judit", dulces)
```

```
//
No hay garantías de una entrega equitativa

dulces <- "Donut"
time.Sleep(time.Second)
dulces <- "Cruasán"
time.Sleep(time.Second)
dulces <- "Ensaimada"
time.Sleep(time.Second)
```

Si se ejecutara el código anterior en su respectiva función main,la salida por pantalla más común sería:

```
Marcos come Donut
Judit come Cruasán
Aina come Ensaimada
```

Sin embargo, cada ciertas ejecuciones podríamos encontrar la siguiente salida:

```
Marcos come Donut
Aina come Cruasán
Marcos come Ensaimada
```

Que Marcos haya comido dos dulces mientras que Judit no haya comido ninguno demuestra que, cuando se diseñen programas en los que múltiples consumidores leen de un mismo canal, **no se debe asumir una repartición "justa" o equitativa** de los mensajes.

16.6 SINCRONIZACIÓN MEDIANTE CANALES

Además de enviar y recibir datos, los canales también pueden usarse para sincronizar la ejecución de gorrutinas, debido al carácter bloqueante de sus operaciones de lectura.

De la misma forma que el tipo sync.WaitGroup (capítulo 15) permite bloquear la ejecución de una o varias gorrutinas hasta que otras gorrutinas lo desbloqueen, puede efectuarse una operación bloqueante de lectura de un canal, que quedará desbloqueada cuando alguien cierre ese canal. En este caso, un canal puede utilizarse como un sync.WaitGroup en el que solo es necesaria una invocación a Done para que se desbloquee.

El siguiente patrón de diseño es muy utilizado en Go para informar de que una tarea ejecutada asíncronamente en paralelo ha finalizado:

```
func TareaAsincrona() <-chan struct{} {
  ch := make(chan struct{})
  go func() {
    fmt.Println("haciendo alguna cosa en paralelo...")
    for i := 0; i < 3; i++ {
      fmt.Println(i, "...")
    }
    fmt.Println("finalizada tarea en paralelo")

    close(ch)
  }()
  return ch
}

func main() {
  espera := TareaAsincrona()

  <-espera

  fmt.Println("programa finalizado")
}
```

Salida:

haciendo alguna cosa en paralelo...

0 ...

1 ...

2 ...

finalizada tarea en paralelo

programa finalizado

En el programa anterior se invoca a una función TareaAsincrona, que retorna un canal del tipo chan struct{}. Este canal no envía ningún dato relevante, ya que el tipo struct{} está vacío, pero se utiliza para bloquear la función main hasta que alguien cierre ese canal y la ejecución del main pueda continuar.

La Figura 16.1 muestra el diagrama de tiempo y la interacción entre las gorrutinas del ejemplo anterior. La creación y lectura del canal espera sucede en la misma gorrutina, mientras el cierre de dicho canal se realiza desde la gorrutina secundaria (que también tiene acceso al canal, ya que ambos se crean dentro de la función TareaAsincrona).

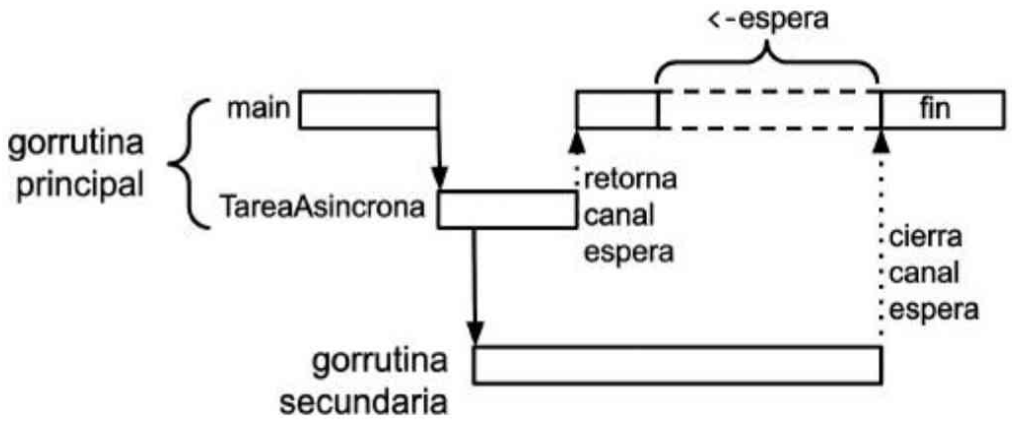


Figura 16.1 Uso de canales para bloquear/desbloquear gorrutinas. Diagrama temporal.

16.7 DEMULTIPLEXIÓN CON select

En determinados casos, puede requerirse que una gorrutina lea datos de varios canales distintos, y actúe según reciba datos de uno o de otro. El bloque select permite a una gorrutina atender simultáneamente los mensajes de varios canales. Su estructura es similar a la de un bloque switch:

```
select {
  case v := <-canal_1
    // acción a realizar si se ha recibido un valor de
    // canal_1, guardado en la variable "v"
  case v := <-canal_2
    // acción a realizar si se ha recibido un valor de
    // canal_2, guardado en la variable "v"
  default:
    // opcional: acción a realizar si no se ha recibido
    // ningún valor en los canales anteriores
}
```

Por ejemplo, la siguiente función CentralMensajeria escucha simultáneamente los remitentes que le pueden llegar por tres canales distintos.

```
func CentralMensajeria(sms, email, carta <-chan string) {
  for {
    select {
    case num := <-sms:
      fmt.Println("recibido SMS del número", num)
    case dir := <-email:
      fmt.Println("recibido email de dirección", dir)
```

```
    case rem := <-carta:
      fmt.Println("recibida carta de remitente", rem)
    }
  }
}

func main() {
  sms := make(chan string, 5)
  email := make(chan string, 5)
  carta := make(chan string, 5)

  go CentralMensajeria(sms, email, carta)

  sms <- "777889923"
  email <- "yahoo@google.com"
  carta <- "Banco Central Hispano"
  email <- "noreply@example.com"

  // damos un poco de margen para que los canales se lean
  // al completo
  time.Sleep(time.Second)
}
```

Salida:

recibido SMS del número 777889923
recibido email de dirección yahoo@google.com
recibida carta de remitente Banco Central Hispano
recibido email de dirección noreply@example.com

En el caso de que más de un canal contenga datos disponibles, Go seleccionará uno cualquiera, indeterminadamente, y dejará los datos del canal sin atender para la siguiente iteración.

16.8 CANCELANDO LECTURAS DESPUÉS DE UN TIEMPO DE ESPERA

En determinadas aplicaciones, es conveniente cancelar una espera si está llevando demasiado tiempo, para no dar al usuario la impresión de que la aplicación está bloqueada.

El constructo select se puede utilizar conjuntamente con la función time.After(time.Duration). Dicha función retorna un canal que recibe el tiempo actual después de la duración pasada como argumento. Por ejemplo, time.After(3*time.Second) retornaría un canal que, pasados tres segundos, recibiría el tiempo actual.

En el siguiente ejemplo una gorrutina dormirá 15 segundos (simulando un tiempo de cálculo o búsqueda muy alto), y enviará un valor por un canal. El receptor del valor escuchará dicho canal, pero también otro canal que especifica un tiempo máximo de espera:

```
ch := make(chan int)

go func() {
  fmt.Println(
    "Calculando la respuesta a la Gran Pregunta" +
      " de la Vida, el Universo, y Todo lo Demás")
  time.Sleep(15 * time.Second)
  ch <- 42
}()

fmt.Println("Esperando...")
select {
case ret := <-ch:
  fmt.Println("Recibido:", ret)
case <-time.After(2 * time.Second):
  fmt.Println("Error: tiempo de espera agotado")
}
```

Salida:

```
Esperando...
Calculando la respuesta a la Gran Pregunta de la Vida, el
Universo, y Todo lo Demás
Error: tiempo de espera agotado
```

Como el canal de tiempo de espera ha recibido un mensaje antes que la respuesta, se ha mostrado el error y el programa ha continuado sin esperar a "la respuesta a la Gran Pregunta de la Vida, el Universo, y Todo lo Demás".
Si, por ejemplo, la respuesta se obtuviera antes de esos dos segundos (o se ampliara la espera a más de 15 segundos), la salida por pantalla sería:

```
Esperando...
Calculando la respuesta a la Gran Pregunta de la Vida, el
Universo, y Todo lo Demás
Recibido: 42
```

16.9 CANCELANDO TAREAS MEDIANTE CONTEXTOS

A medida que indague en las API de Go, tanto estándar como módulos proporcionados por terceros, observará que muchas funciones y métodos reciben como primer argumento un objeto del tipo context.Context.

Como se intuye por su nombre, context.Context proporciona información acerca del contexto en el que una función se invoca. Pero la parte que nos interesa en esta sección es su utilidad de cancelar explícitamente tareas mediante canales.

El tipo context.Context dispone de un método llamado Done(), que retorna un canal que se cierra cuando el contexto ha finalizado (o ha sido cancelado). Cuando se leen datos de un canal, se puede leer también el canal retornado por Done() para tratar la cancelación de la tarea en cuestión. Por ejemplo, la siguiente función mostraría un mensaje después de 5 segundos, a no ser que el contexto pasado por argumento finalice antes:

```
func MuestraRetardada(ctx context.Context, msg string) {
  select {
  case <-time.After(5 * time.Second):
    fmt.Println(msg)
  case <-ctx.Done():
    // proceso interrumpido! La función continúa
  }
}
```

Generalmente, y si no se desea cancelar la ejecución en ningún caso, cuando se invocan funciones que reciben un contexto se puede pasar el contexto global del programa, que puede obtenerse mediante la función context.Background().

A partir de un contexto, pueden crearse nuevos subcontextos con nuevas características de cancelación. Para ello, el paquete context proporciona las

siguientes funciones:

- func WithCancel(Context) (Context, CancelFunc)
  - Esta función recibe un contexto y retorna dos valores. El primero es un contexto nuevo, que puede cancelarse explícitamente mediante la función que se retorna en segundo lugar (el tipo CancelFunc es una simple func()) .
- func WithDeadline(Context, time.Time) (Context, CancelFunc)
  - Recibe un contexto y una marca de tiempo (fecha y hora). Retorna un nuevo contexto que se cancela automáticamente llegado el momento pasado como segundo argumento. También retorna una función que permite cancelar el contexto explícitamente.
- WithTimeout(Context, time.Duration) (Context, CancelFunc)
  - Recibe un contexto y una duración. Retorna un contexto que se cancela automáticamente después de la duración pasada como argumento, contando desde el momento en que WithTimeout crea el contexto. También retorna una función que permite cancelar el contexto explícitamente.

El siguiente ejemplo invoca la función MuestraRetardada pasándole un contexto cancelable. En una gorrutina paralela, pide al usuario que pulse la tecla "Intro" si desea cancelar el mensaje:

```
func main() {
  ctx, cancela := context.WithCancel(context.Background())

  fmt.Println("Un mensaje se mostrará en 5 segundos...")
  go func() {
    fmt.Println("Pulsa INTRO para cancelar mensaje")
    fmt.Scanf("\n")
    cancela()
  }()

  MuestraRetardada(ctx, "Hola!!")
  fmt.Println("finalizando")
}
```

Con la función context.WithCancel se crea un nuevo contexto a partir del contexto global retornado por context.Background(). Ese contexto es el que se debe pasar como primer argumento a MuestraRetardada. En la gorrutina de fondo, se ejecuta Scanf, que bloqueará la ejecución de la gorrutina hasta que se pulse "Intro".

Si pasan 5 segundos, el select interno de MuestraRetardada tomará el camino en el que se muestra el mensaje. Entonces la gorrutina principal termina, así como la gorrutina de fondo (ya que el programa finaliza).

Si se pulsa "Intro" antes de 5 segundos, la función fmt.Scanf se desbloquea, se invoca la función cancela() y el select interno de MuestraRetardada ejecuta el case ←ctx.Done() que, al estar vacío, simplemente terminará la función, y finalizará la gorrutina principal sin haber mostrado ningún mensaje.

La salida estándar, si no se pulsa "Intro", será:

```
Un mensaje se mostrará en 5 segundos...
Pulsa INTRO para cancelar mensaje
Hola!!
finalizando
```

Si se pulsa “Intro” antes de 5 segundos, será:

Un mensaje se mostrará en 5 segundos...

Pulsa INTRO para cancelar mensaje

finalizando

Capítulo 17
SERVICIOS WEB

No solo las personas nos comunicamos a través de Internet. Las máquinas también lo hacen, mediante los llamados **servicios web:** mensajes que se envían con peticiones para hacer cálculos, buscar o guardar datos, validar pagos, etc.

En una petición o mensaje a un servicio web intervienen dos partes: el **servidor** (la máquina que recibe la petición y retorna un resultado) y el **cliente** (la máquina que realiza la petición, con los detalles sobre esta). En las arquitecturas orientadas a servicios, una máquina o programa puede ser a la vez servidor y cliente de otros servicios.

Si bien el tema de los servicios web daría para un libro entero, este capítulo pretende ser un punto de introducción para que el lector pueda familiarizarse con las bibliotecas estándar de Go para el manejo de servicios y clientes web.

Como este libro no asume ningún conocimiento específico por parte del lector en materia de servicios, las dos siguientes secciones explicarán brevemente dos de las especificaciones más ampliamente usadas para la implementación de servicios remotos: HTTP y REST.

17.1 HTTP EXPLICADO EN 3 MINUTOS

La web, establecida durante la última década del siglo xx, inicialmente se concibió como un conjunto de documentos disponibles y enlazados mediante localizadores (o direcciones) que siguen un formato concreto: el localizador uniforme de recursos, conocido como **URL**, por su nombre en inglés, *Uniform Resource Locator*.

Una URL generalmente sigue la siguiente estructura:

protocolo://servidor:puerto/ruta/al/recurso

donde:

- protocolo es la forma de comunicación en que los datos son transmitidos desde una máquina a la otra. Por ejemplo: http o https.
- servidor es el nombre, dominio, o dirección IP de la máquina que contiene el documento.
- puerto es un número que indica a través de qué puerto TCP se establecerá la conexión con el servidor. Si no se indica, se asume 80 si el protocolo es http, o 443 si es https.
- La ruta al recurso será una lista de directorios separados por una barra / , que indican un camino que lleva a un directorio o archivo concreto.

Por ejemplo:

http://macias.info/static/assets/others/phdthesis.pdf

El protocolo creado para la adquisición de documentos a través de la web es el Protocolo de Transferencia de Hipertexto, abreviado como **HTTP** por su denominación en inglés, *Hypertext Transfer Protocol*. Para máquinas que se comunican a través de Internet u otras redes públicas (y no fiables), es común el uso del protocolo **HTTPS**, donde la S significa "seguro".

En el protocolo HTTP, un **cliente** (la máquina que quiere acceder a un documento) envía una petición a un **servidor** (la máquina que contiene o gestiona

el documento), indicando la **ruta** (o *path)* a dicho documento, así como otros datos que no detallaremos, enviados como **cabeceras** (o *headers)*. El cliente también puede enviar un documento, como parte del llamado **cuerpo de la petición** *(request body)*.

En una cabecera, el cliente puede enviar el **método** de la petición, que generalmente indicará qué tipo de acción se desea realizar sobre un documento concreto: obtenerlo, borrarlo, modificarlo, moverlo, etc.

Una vez procesada la petición del cliente, el servidor retornará un **código de respuesta** numérico, indicando el resultado correcto o erróneo de la petición, más el cuerpo del documento pedido (o el cuerpo del mensaje de error).

17.2 REST EXPLICADO EN 3 MINUTOS

REST es el acrónimo de *Representational State Transfer* en inglés, algo así como "transferencia de estado mediante representaciones", y se ha convertido en un estándar *de facto* para la comunicación remota entre programas mediante el protocolo HTTP.

Si bien HTTP se diseñó inicialmente para la gestión y transferencia de documentos, su versatilidad permitió adoptarlo para poder gestionar remotamente el estado de otras entidades dentro del dominio de un programa (por ejemplo, los datos de los usuarios del sistema, libros en una tienda de libros, vídeos en un sistema de visionado en línea...).

Cada una de estas entidades, también conocidas como **recursos**, tiene su correspondencia con HTTP de la siguiente forma:

- Cada recurso se identifica mediante una **URL** única.
- Los documentos se transfieren en el **cuerpo** de la **petición** (envío) o de la **respuesta** (recepción).
- Se utilizan **códigos de respuesta** para informar sobre el resultado de la operación. Los códigos más comunes son:
  - 200 (OK). La petición se ha realizado correctamente.
  - 201 (Creado). El documento se ha creado correctamente.
  - 204 (Sin contenido). La petición es correcta pero no hay ningún documento que mostrar.
  - 400 (Petición errónea). El servidor no ha entendido la petición del cliente, porque el formato es incorrecto.
  - 401 (Desautorizado). El cliente no tiene permiso para hacer uso de ese servicio web. Generalmente deberá identificarse mediante las cabeceras HTTP.
  - 403 (Prohibido). El cliente, pese a estar identificado correctamente, no tiene permiso para acceder a un recurso concreto.

–404 (No encontrado). No existe recurso o documento asociado a la URL especificada.

–500 (Error interno del servidor). Un error ha sucedido en el servidor. El cliente no ha hecho nada mal, simplemente el servidor no ha funcionado correctamente.

•Entre muchos otros, se utilizan los siguientes **métodos** para gestionar los recursos o documentos:

–GET para obtener información acerca de un recurso (o una lista de recursos).

–POST para crear un nuevo recurso o documento. Se deberá enviar el documento en el cuerpo de la petición.

–PUT para modificar un recurso o documento existente. Se deberá enviar el documento en el cuerpo de la petición.

–DELETE para eliminar un recurso existente.

•Tanto en la respuesta como en la petición se pueden usar cabeceras de todo tipo para especificar información acerca del cliente, las peticiones, las respuestas, etc. Una de las cabeceras más comunes para la creación de peticiones es Accept, que especifica el formato de los datos que el cliente espera. En la parte de la respuesta, la cabecera Content-Type especifica el formato de los datos retornados por el servidor. Los valores más comunes para Accept y Content-Type son:

–text/plain, para texto en crudo, sin ningún formato concreto.

–text/html, para documentos en formato "marcas de hipertexto", HTML.

–image/jpeg, image/png, o similares, para imágenes.

–application/xml para documentos en formato "Lenguaje Extensible de Marcas", XML.

–application/json para documentos en formato "Notación de Objetos

JavaScript", JSON.

17.3 CREACIÓN DE UN SERVICIO HTTP EN GO

Para crear un servicio en Go, este capítulo mostrará el uso de distintos tipos de datos y funciones del paquete "net/http" de la biblioteca estándar.

17.3.1 Interfaz http.Handler

La interfaz http.Handler implementa un único método:

```
ServeHTTP(http.ResponseWriter, *http.Request)
```

El método ServeHTTP es el encargado de atender cualquier tipo de petición HTTP. Recibe dos argumentos. El primer argumento, del tipo http. ResponseWriter, se utiliza para enviar una respuesta al cliente —como el cuerpo del documento, el código de retorno, cabeceras opcionales, etc.—. El segundo argumento, del tipo *http.Request (apuntador), contiene información acerca de la petición del cliente —como cabeceras, URL, método (GET, POST...), cuerpo de la petición, etc.—.

Cualquier tipo de dato que implemente la interfaz http.Handler podrá usarse desde un servidor HTTP para atender peticiones según una lógica de negocio concreta.

17.3.2 Funciones http.ListenAndServe y http.ListenAndServeTLS

Estas funciones crean un servidor HTTP global al programa, que aceptará peticiones en un determinado puerto y las enviará al http.Handler que se pase como argumento.

http.ListenAndServe crea un servidor HTTP normal, sin ningún tipo de comunicación segura. Su signatura es:

```
func ListenAndServe(addr string, handler Handler) error
```

donde addr es una dirección de Internet que consta de un nombre de equipo o dirección IP, seguida de dos puntos : y un número de puerto donde realizar las escuchas. Si no se especifica ningún equipo o IP, Go escuchará en la dirección local (generalmente, localhost o 127.0.0.1), en el puerto especificado mediante dos puntos y el número de puerto (por ejemplo, ":8080").

La función http.ListenAndServe enviará las peticiones recibidas en la dirección y puerto especificados hacia la implementación de http. Handler pasada como argumento.

La función http.ListenAndServeTLS crea un servidor HTTPS, es decir, implementa una conexión segura. Su signatura es:

```
func
ListenAndServeTLS(addr, certFile, keyFile string, handler
Handler) error
```

donde certFile y keyFile son la ruta hacia el archivo que contiene los certificados y claves del servidor.

Para no dispersar la temática del libro, los ejemplos de este libro obviarán las conexiones seguras, que requerirían de la creación de certificados y claves válidas mediante una autoridad de certificación, y se centrarán en conexiones HTTP normales (asumiendo que las comunicaciones se realizan en una red

interna y segura).

Las funciones http.ListenAndServe y http.ListenAndServeTLS no deberían finalizar nunca (solo cuando se interrumpe el programa). Si las funciones acaban prematuramente (por ejemplo, debido a un error interno), retornan un error.

17.3.3 Ejemplo de servidor HTTP

El siguiente ejemplo provee una implementación sencilla de http. Handler que, para cada petición del cliente, retorna un documento web sencillo mostrando un mensaje de bienvenida e información sobre la URL del cliente:

```
type HolaServicio struct{}

func (hs *HolaServicio) ServeHTTP(
  rw http.ResponseWriter, req *http.Request) {

  rw.Header().Add("Content-Type", "text/html")

  documento := fmt.Sprintf(`
    <h1>Bienvenido!</h1>
    <p>Ruta de acceso: %s</p>
  `, req.URL.Path)

  rw.Write([]byte(documento))
}
```

El tipo HolaServicio implementa la interfaz http.Handler. En cada petición del usuario (método ServeHTTP), retornará una respuesta que contiene:

- Código de estado 200 OK. Si no se especifica nada (mediante el método WriteHeader(int)), 200 será el código por defecto.
- Cabecera Content-Type con valor text/html, que indica que el documento retornado por el servidor es un archivo en el formato HTML (usado para especificar el formato visual de las páginas web comunes).
- Cuerpo del mensaje como un documento HTML en el que se provee información de la ruta en URL indicada por el usuario (variable req.URL.Path) .

La anterior implementación de http.Handler se integra dentro de un servidor HTTP mediante la función http.ListenAndServe de la siguiente manera:

func

```
main() {
  panic(http.ListenAndServe(":8080", &HolaServicio{}))
}
```

Recuerde que, en principio, la invocación a panic no debería realizarse nunca, a no ser que http.ListenAndServe termine prematuramente y retorne un error.

Si después de ejecutar el servicio anterior abre su navegador web favorito e introduce una URL que empiece por http://localhost:8080 o http://127.0.0.1:8080 (máquina local escuchando en el puerto 8080), seguida de una ruta cualquiera, el navegador le mostrará un documento HTML de aspecto similar al de la Figura 17.1

Figura 17.1 Invocación a un servicio en Go, que retorna un documento HTML.

17.4 CREACIÓN DE UN CLIENTE HTTP EN GO

Go proporciona el tipo de dato estructurado http.Client, que se puede instanciar por defecto como:

```
cliente := http.Client{}
```

Sin embargo, es recomendable especificar un tiempo máximo de espera (atributo Timeout), tras el cual una petición a un servicio fallará si este no responde en el tiempo especificado. Por ejemplo:

```
client := http.Client{
  Timeout: 5 * time.Second,
}
```

Como siempre, puede consultar la documentación de la API de Go (comando godoc) para conocer al detalle más opciones de configuración.

La forma común de realizar una petición es mediante el tipo de datos http.-Request, que se puede construir mediante el método http. NewRequest, cuya signatura es:

```
func
NewRequest(metodo, url string,
          cuerpo io.Reader)  (*http.Request, error)
```

donde:

- metodo es el método de la petición. Pueden usarse directamente los valores "GET", "POST", "PUT" , etc., pero suele ser recomendable utilizar las constantes globales http.MethodGet, http.MethodPost, http.MethodPut, etc.
- url es la dirección completa (protocolo, servidor, puerto, ruta...) donde realizar la petición.
- cuerpo es un flujo de entrada (io.Reader) al contenido del cuerpo, usado en peticiones del tipo POST o PUT. En otro tipo de peticiones (como GET

o DELETE) puede ser nil.

Una vez se ha creado el apuntador *http.Request, este debe utilizarse en el método Do del tipo http.Client, cuya signatura es:

```
func
(c *http.Client) Do(
        req *http.Request)  (*http.Response, error)
```

Dicho método retornará una respuesta a la petición, que indicará valores tales como el código de respuesta, las cabeceras o el cuerpo del documento retornado.

17.4.1 Ejemplo de cliente HTTP

El siguiente ejemplo crea un cliente que invoca el servidor HTML del ejemplo anterior (cualquier otro servidor HTTP funcionaría, cambiando la URL), y muestra el código de respuesta (elemento StatusCode del tipo http.Response), la cabecera Content-Type (elemento Header, del tipo map) y el cuerpo del documento retornado (elemento Body, que implementa la interfaz io.Reader):

```
req, err := http.NewRequest(http.MethodGet,
  "http://localhost:8080/ruta/de/documento", nil)
if err != nil {
  panic(err)
}
client := http.Client{
  Timeout: 5 * time.Second,
}
resp, err := client.Do(req)
if err != nil {
  panic(err)
}
fmt.Println("Código de respuesta:", resp.StatusCode)
fmt.Println("Content-Type:", resp.Header["Content-Type"])
cuerpo, err := ioutil.ReadAll(resp.Body)
if err != nil {
  panic(err)
}
```

La gestión de errores se ha simplificado como una invocación a panic, en aras de la brevedad. Si el anterior servidor de ejemplo sigue ejecutándose, la salida del cliente de ejemplo sería similar a:

```
Código de respuesta: 200 200 OK
Content-Type: [text/html]
```

<h1>Bienvenido!</h1>

Ruta de acceso: /ruta/de/documento

17.5 EJEMPLO PRÁCTICO DE SERVICIO REST

Como ejemplo práctico, esta sección definirá un servicio básico REST en el que un cliente puede leer, añadir, modificar y borrar documentos en texto plano.

Un documento se identificará por la ruta de la URL. Así, por ejemplo, la URL http://localhost:8080/mi-documento se refiere a un documento alojado en la máquina local, que se identifica como mi-documento.

Para cada documento se pueden realizar las siguientes acciones:

- Leer documento
  - Método HTTP: GET
  - Retorna:
    - Código 200 (OK)
    - Cuerpo de la respuesta: un texto con el documento alojado en el servidor
  - Posibles errores:
    - 404 (no encontrado) si no hay ningún documento para esa ruta
- Borrar documento
  - Método HTTP: DELETE
  - Retorna: Código 200 (OK)
  - Posibles errores:
    - 404 (no encontrado) si no hay ningún documento para esa ruta
- Añadir documento
  - Método HTTP: POST
  - Cuerpo de la petición: Documento a añadir
  - Retorna: Código 200 (OK)
  - Posibles errores:
    - 400 (petición errónea) si ya existe un documento con esa ruta
- Modificar documento

–Método HTTP: PUT

–Cuerpo de la petición: Documento modificado

–Retorna: Código 200 (OK)

–Posibles errores:

0404 (no encontrado) si no hay ningún documento para esa ruta

En este servicio de pruebas, los documentos se guardarán en memoria, en un map[string]string, donde el identificador se guarda como clave, y el documento en sí se guarda como valor asociado a la clave:

```
type Rest struct
{
 entradas
map
[string]string
}
```

Para poder usar el tipo Rest en un servidor, haremos que implemente la interfaz http.Handler creando el método ServeHTTP, que será el encargado de gestionar las peticiones del cliente:

```
func (mr *Rest) ServeHTTP(
  rw http.ResponseWriter, req *http.Request) {

  // obtener identificador y documento (si hay alguno)
  // de la petición
  identificador := req.URL.Path
  var documento string
  if req.Body != nil {
    b, _ := ioutil.ReadAll(req.Body)
    documento = string(b)
  }

  // las respuestas a retornar no siguen un formato concreto
  // serán "texto llano"
  rw.Header().Add("Content-Type", "text/plain")

  // según el método de la petición, nuestro http.Handler
  // llevará a cabo una u otra acción
  switch req.Method {
  case http.MethodGet:
    mr.peticionGet(identificador, rw)
  case http.MethodDelete:
    mr.peticionDelete(identificador, rw)
  case http.MethodPut:
    mr.peticionPut(identificador, documento, rw)


  case http.MethodPost:
    mr.peticionPost(identificador, documento, rw)
  default:
    rw.WriteHeader(http.StatusBadRequest)
    fmt.Fprintln(rw, "inválido:", req.Method)
  }
}
```

La función ServeHTTP realiza las siguientes acciones:

- Identifica el documento leyendo la ruta de la URL de la petición (atributo req.URL.Path).
- Si la petición incluye algún documento en el cuerpo (atributo req. Body), este se lee y se guarda en un string. Por brevedad en el código de ejemplo, se descarta el posible error retornado por ioutil.ReadAll.
- Según el método de la petición retornado por req.Method (GET, DELETE, POST o PUT, definido en las constantes http.MethodGet, http.MethodDelete, http.MethodPost y http.MethodPut, respectivamente), se invoca a las respectivas funciones, mostradas más adelante, que definen la lógica de cada operación.
- Si el método no es ninguno de los anteriores, se retorna un código 400 (petición errónea), definido por la constante http. StatusBadRequest. Observe que se invoca la función fmt. Fprintln, que escribe sobre la respuesta http.ResponseWriter, ya que esta implementa la interfaz io.Writer.

A continuación, se muestra la función peticionGet, que realiza la lectura del documento a partir de una clave. Si la clave existe en el mapa entradas del servidor, escribe el documento en el cuerpo de la respuesta (el código de estado 200 OK se retorna por defecto). Si no existe un documento con esa clave, retorna 404 (definido en la constante http.StatusNotFound) y, como cortesía, un mensaje de error en el cuerpo de la respuesta:

```
func (mr *Rest) peticionGet(
  identificador string, rw http.ResponseWriter) {

  if documento, ok := mr.entradas[identificador]; ok {
    fmt.Fprintln(rw, documento)
  } else {
    rw.WriteHeader(http.StatusNotFound)
    fmt.Fprintln(rw, "no encontrado:", identificador)
  }
}
```

La función peticionDelete borra del mapa de memoria el documento cuya clave se ha enviado, y retorna OK en el cuerpo de la función. Si el documento a borrar no existe, retorna error 404 (no encontrado) y un mensaje de error en el cuerpo de la respuesta:

```
func (mr *Rest) peticionDelete(
  identificador string, rw http.ResponseWriter) {
  if _, ok := mr.entradas[identificador]; ok {
    delete(mr.entradas, identificador)
    fmt.Fprintln(rw, "OK")
  } else {
    rw.WriteHeader(http.StatusNotFound)
    fmt.Fprintln(rw, "no encontrado:", identificador)
  }
}
```

La función peticionPost crea un documento, previamente recibido a través del cuerpo de la petición. Este se añade en el mapa de entradas, a no ser que ya existiera un documento para dicha entrada, en cuyo caso se retornaría error 400 (petición errónea) y un mensaje de error:

```
func (mr *Rest) peticionPost(
  identificador, documento string, rw http.ResponseWriter) {

  if _, ok := mr.entradas[identificador]; ok {
    rw.WriteHeader(http.StatusBadRequest)
    fmt.Fprintln(rw, "ya existente:", identificador)
  } else {
    mr.entradas[identificador] = documento
    fmt.Fprintln(rw, "OK")
  }
}
```

La función peticiónPut modifica un documento ya existente. Si este no existe, devuelve error 404. Si existe, lo sobreescribe con el nuevo valor recibido en el cuerpo de la función y retorna 200 OK:

```
func (mr *Rest) peticionPut(
  identificador, documento string, rw http.ResponseWriter) {

  if _, ok := mr.entradas[identificador]; ok {
    mr.entradas[identificador] = documento
    fmt.Fprintln(rw, "OK")


  } else {
    rw.WriteHeader(http.StatusNotFound)
    fmt.Fprintln(rw, "no encontrado:", identificador)
  }
}
```

Una vez creado el servicio al completo, solo faltará crear un servidor HTTP y pasarle un apuntador a una variable del tipo Rest, como implementación de http.Handler:

```
func main() {
  rest := Rest{
    entradas: map[string]string{},
  }

  http.Handle("/", &rest)

  panic(http.ListenAndServe(":8080", nil))
}
```

17.5.1 Probando el servicio REST

Para probar el funcionamiento del servicio, se podría crear un cliente en Go que invocara los diversos métodos del servicio y comprobara sus salidas. Pero, en este caso, dada la sencillez del servicio, puede ser más rápido probarlo mediante uno de los tantos clientes REST disponibles gratuitamente. Puede encontrar decenas de ellos buscando "REST client" en la tienda de complementos de su navegador web favorito.

En este libro utilizaremos la herramienta de la línea de comandos curl, instalada por defecto en la mayoría de sistemas operativos, o disponible gratuitamente para descarga en la dirección curl.haxx.se.

Una vez compilado, y durante la ejecución del anterior programa servidor, desde la línea de comandos ejecutaremos el comando curl para pedir a nuestro servicio un documento llamado Japon:

```
$ curl http://localhost:8080/Japon

no encontrado: /Japon
```

Ha retornado un mensaje de error, ya que el servicio aún no guarda ningún documento. Si quiere ver más detalles sobre la petición y la respuesta, puede ejecutar curl con el argumento -v, que mostrará algunos datos extra, tanto de la petición (precedidos por el carácter >) como de la respuesta (precedidos por el carácter <):

```
$ curl -v http://localhost:8080/Japon

> GET /Japon HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.64.1
> Accept: */*
>
```

```
< HTTP/1.1 404 Not Found
< Content-Type: text/plain
< Date: Fri, 04 Sep 2020 16:25:38 GMT
< Content-Length: 22
<
no encontrado: /Japon
```

Probaremos a crear un nuevo documento, mediante una petición POST que contiene Osaka como cuerpo de la petición:

```
$ curl  -X POST -d "Osaka" http://localhost:8080/Japon
OK
```

Si se vuelve a ejecutar de nuevo el método GET para el documento Japon, verá que ahora sí recibe un documento asociado:

```
$ curl http://localhost:8080/Japon
Osaka
```

Si pretende añadir otro documento mediante una clave ya existente, también recibirá un mensaje de error:

```
$ curl  -X POST -d "Tokyo" http://localhost:8080/Japon
ya existente: /Japon
```

En este caso, si lo que se pretende es cambiarlo, se debe usar el método PUT:

```
$ curl -X PUT -d "Tokyo" http://localhost:8080/Japon
OK
$ curl http://localhost:8080/Japon
Tokyo
```

Mediante DELETE, borraremos un documento ya existente y este dejará de existir:

```
$ curl -X DELETE http://localhost:8080/Japon

OK
$ curl
http://localhost:8080/Japon

no encontrado: /Japon
```

Manualmente, hemos verificado que nuestro programa funciona. La tarea de verificación manual cada vez que usted haga una modificación a su servicio puede ser muy tediosa. Para ahorrarle tiempo, el capítulo 20 de este libro tratará sobre cómo definir pruebas automatizadas de software.

Capítulo 18
SERIALIZACIÓN DE DATOS

Cuando dos máquinas intercambian peticiones y documentos mediante HTTP, los datos casi nunca son simples textos en lenguaje humano, sino que están estructurados en un formato fácil de interpretar por una máquina.

El formato JSON (JavaScript Object Notation) es ampliamente utilizado para serializar peticiones y respuestas en clientes y servicios web, por su sencillez y su expresividad. Y, sobre todo, por estar basado en el lenguaje JavaScript —por lo que es fácilmente integrable en cualquier navegador web para la comunicación con servidores remotos—.

Por ejemplo, supongamos que pretendemos representar a un futbolista según las siguientes características:

- Nombre (texto)
- Año de nacimiento (número)
- Equipos en los que ha jugado (lista de textos)

La representación de un futbolista cualquiera como objeto JSON sería similar a:

```
{
  "nombre": "Carles Reixach",
  "nacimiento": 1947,
  "equipos": [
    "F.C. Barcelona",
    "Selección Española de Fútbol"
  ]
}
```

Obsérvese que la estructura de un objeto JSON no es muy distinta a la de un

tipo de dato estructurado en Go. Un futbolista como el del ejemplo anterior podría definirse mediante un nuevo tipo de dato de la siguiente manera:

```
type Futbolista struct {
  Nombre     string
  Nacimiento int
  Equipos    []string
}

var cr = Futbolista{
  Nombre:     "Carles Reixach",
  Nacimiento: 1947,
  Equipos: []string{
    "F.C. Barcelona",
    "Selección Española de Fútbol",
  },
}
```

Hay muchas similitudes entre la estructura de un documento JSON y la de un struct o porción de Go, por lo que el paquete encode/json proporciona funciones para automatizar y facilitar la **serialización** de Go a JSON y la **deserialización** de JSON a Go.

18.1 SERIALIZACIÓN DE TIPOS GO A JSON

La función json.Marshal recibe una variable cualquiera de Go (un struct o una porción) y retorna una porción de bytes con la representación JSON de la variable Go pasada como argumento (o con un error si algo no funcionó).

func Marshal(v **interface{})** ([]byte, error)

Por ejemplo:

```
cr := Futbolista{
  Nombre:     "Carles Reixach",
  Nacimiento: 1947,
  Equipos: []string{
    "F.C. Barcelona",
    "Selección Española de Fútbol",
  },
}


ftbl, err := json.Marshal(cr)
if err != nil {
  fmt.Println("Error serializando:", err)
  return
}
fmt.Println(string(ftbl))
```

Salida en pantalla en un JSON sin un formato/tabulación concreto:

{"Nombre":"Carles Reixach","Nacimiento":1947,"Equipos":[
"F.C. Barcelona","Selección Española de Fútbol"]}

Observe que se han respetado los nombres de los atributos, y que estos empiezan en mayúscula. Si se desea dar un nombre diferente a un atributo concreto, se pueden utilizar **anotaciones** consistentes en un string después del tipo de dato de cada elemento, que contenga una anotación en el formato

json:"nombre_del_atributo":

```
type Futbolista struct {
  Nombre     string   `json:"nombre"`
  Nacimiento int      `json:"nacimiento"
  Equipos    []string `json:"equipos"`
}
```

Con las anotaciones anteriores en la definición de Go, los atributos se serializan a JSON con minúscula, tal y como se ha especificado en cada etiqueta json:

{"nombre":"Carles Reixach","nacimiento":1947,"equipos":
["F.C. Barcelona","Selección Española de Fútbol"]}

Si se necesita especificar más opciones de serialización a JSON (por ejemplo, que el texto se muestre formateado e indentado), se puede utilizar el tipo de dato json.Encoder o la función json.MarshalIndent.
Como apunte final, es importante recalcar que solo se serializarán los elementos públicos de los struct (es decir, cuyos nombres empiecen en mayúscula).

18.2 DESERIALIZACIÓN DE JSON A TIPOS GO

La función json.Unmarshal lee una cadena JSON y copia sus valores al objeto que se le pasa como argumento (o retorna error si ha habido algún problema de formato o lectura):

func Unmarshal(datos []byte, destino **interface{})** error

El primer argumento recibe un documento JSON y el segundo recibe un apuntador a un tipo de dato capaz de guardar el documento JSON. Por ejemplo:

```
documento := []byte(`{
  "nombre": "Paco Garcia",
  "nacimiento": 2000,
  "liga": "Liga Peruana"
}`)
var f Futbolista
err := json.Unmarshal(documento, &f)
if err != nil {
  fmt.Println("Error deserializando:", err)
  return
}
fmt.Printf("deserializado: %#v\n", f)
```

Salida:

deserializado: main.Futbolista{Nombre:"Paco Garcia",
Nacimiento:2000, Equipos:[]string(nil)}

Observe el comportamiento del código anterior:

- El documento JSON original contenía un atributo "liga" que, al no estar definido en el tipo Futbolista, ha sido ignorado.
- Como el JSON original no contenía ningún valor para el atributo

"equipos", no se ha asignado ningún valor al atributo respectivo en Futbolista.

Si intentara deserializar el siguiente documento JSON:

```
{
  "nombre": "Paco Garcia",
  "nacimiento": "un dia cualquiera",
  "liga": "Liga Peruana"
}
```

La salida del programa anterior hubiera sido:

Error deserializando: json: cannot unmarshal string into Go struct field Futbolista.nacimiento of type int

ya que el atributo Nacimiento del struct Futbolista es del tipo int y en el JSON original es un string.

18.3 SERIALIZANDO Y DESERIALIZANDO DOCUMENTOS JSON SIN FORMATO

Los ejemplos anteriores requerían que los documentos JSON tuvieran un formato concreto que concordara con el formato de los tipos Go sobre los que se deserializaban.

Si los documentos JSON han de tener un formato libre, estos pueden serializarse desde y deserializarse hacia un map.

Por ejemplo, para leer un documento JSON de formato libre y guardarlo en un mapa:

```
documento := []byte(`{
  "receta": "Huevos Fritos",
  "ingredientes": [
    "Huevos",
    "Aceite",
    "Sal"
  ]
}`)
destino := map[string]interface{}{}
err := json.Unmarshal(documento, &destino)
if err != nil {
  fmt.Println("Error deserializando:", err)
  return
}
for c, v := range destino {
  fmt.Printf("%s --> %v\n", c, v)
}
```

Observe que, aunque map es un tipo de dato por referencia, la función json.Unmarshal requiere que se pase como puntero.

Salida estándar:

receta -- > Huevos Fritos

ingredientes -- > [Huevos Aceite Sal]

18.4 SERIALIZACIÓN DE PORCIONES Y ARRAYS

Las listas que contienen elementos de cualquier tipo también son documentos JSON válidos. Estas pueden serializarse y deserializarse mediante porciones y vectores de Go.

El siguiente ejemplo decodifica una lista JSON en una porción de string. Después, añade un nuevo elemento a la porción en Go y lo codifica de nuevo a JSON:

```
listaJson := []byte(`["hola", "que", "tal"]`)
var lista []string
if err := json.Unmarshal(listaJson, &lista); err != nil {
  panic(err)
}
fmt.Println("deserializado:", lista)
lista = append(lista, "amigo")
listaJson2, err := json.Marshal(lista)
if err != nil {
  panic(err)
}
fmt.Println("serializado:", string(listaJson2))
```

Salida estándar:

deserializado: [hola que tal]

serializado: ["hola","que","tal","amigo"]

18.5 SERIALIZACIÓN Y DESERIALIZACIÓN EN OTROS FORMATOS

Si bien JSON se ha convertido en un estándar en la programación web, existen otros formatos que gozan de amplia popularidad en otros entornos, que pueden deserializarse y serializarse en Go de manera similar a JSON.

Por ejemplo, para serializar y deserializar datos en XML (Lenguaje de Marcas Extensible), se utiliza el paquete encoding/xml, que funciona de manera análoga a encoding/json. Por ejemplo:

```
type Animal struct {
  Nombre string `xml:"nombre"`
  Tipo   string `xml:"tipo"`
}

func main() {
  a := Animal{Nombre: "Perro", Tipo: "Mamífero"}
  b, err := xml.Marshal(a)


  if err != nil {
    panic(err)
  }
  fmt.Println(string(b))
}
```

Salida estándar:

<Animal><nombre>Perro</nombre><tipo>Mamífero</tipo></Animal>

El formato YAML (Yet Another Markup Language, que podría traducirse como "otro lenguaje de marcas más"), pese a no ser un estándar, también ha ganado mucha tracción en los últimos años, debido a su uso extensivo como lenguaje de configuración de sistemas. Las bibliotecas estándar de Go no

proveen soporte para YAML, pero se puede incorporar mediante un módulo externo, añadiendo la siguiente línea en el fichero con">go.mod:

require gopkg.in/yaml.v2 v2.3.0

Una vez incorporado, se puede utilizar YAML en nuestro proyecto, de manera análoga a como se utilizan JSON y XML:

```
package main

import (
  "fmt"

  "gopkg.in/yaml.v2"
)

type Alumno struct {
  Nombre   string `yaml:"nombre"`
  Apellido string `yaml:"apellido"`
}

type Aula struct {
  Alumnos []Alumno `yaml:"alumnos"`
}

func main() {
  a := Aula{Alumnos: []Alumno{
    {Nombre: "Carolina", Apellido: "Martínez"},
```

```
    {Nombre: "Juan Francisco", Apellido: "Pérez"},
  }}
  txt, _ := yaml.Marshal(a)
  fmt.Println(string(txt))
}
```

Salida:

```
alumnos:
- nombre: Carolina
  apellido: Martínez
- nombre: Juan Francisco
  apellido: Pérez
```

Capítulo 19
CONEXIÓN A BASES DE DATOS SQL

La memoria de un programa es volátil, es decir, se borra cuando el programa finaliza. Cuando un programa precisa conservar la información entre ejecuciones del programa (incluso apagado del ordenador), es necesario usar mecanismos de **persistencia** de la información.

Un mecanismo de persistencia simple consiste en guardar los datos a conservar en disco, serializados en JSON, YAML, XML o similares. Esto es válido cuando los datos a guardar son relativamente sencillos y/o pequeños, y cuando son relativamente poco cambiantes.

Para grandes cantidades de datos cambiantes, se utilizan los llamados "sistemas gestores de bases de datos", que permiten almacenar ingentes cantidades de datos, así como establecer relaciones y restricciones entre estos.

Este capítulo muestra los paquetes estándar de Go para la conexión y operación con las llamadas **bases de datos relacionales**, mediante el paquete databases/sql, que proporciona una interfaz unificada para la conexión a diversas bases de datos SQL (Oracle, MySQL, PostgreSQL, etc.).

De entre todas las bases de datos disponibles, en los ejemplos de este capítulo se utilizará SQLite (www.sqlite.org), ya que esta puede ser incrustada directamente en nuestros programas Go, sin necesidad de instalar y configurar un servidor de bases de datos externo. SQLite no está pensada para ser una base de datos a gran escala (por ejemplo, para un servicio web con millones de usuarios), pero sí ofrece un muy bien rendimiento para aplicaciones locales, tales como aplicaciones de escritorio o móviles, o pequeños servicios.

Este capítulo no pretende enseñar lenguaje SQL (cuyas bases requerirían un libro entero). Se asume que el lector interesado en los contenidos de este capítulo tiene unas nociones básicas de SQL. De igual modo, el léxico SQL

aquí expuesto es tan sencillo que el lector no familiarizado con SQL podrá entenderlo con la ayuda de un manual para principiantes.

19.1 CARGA DE CONTROLADOR

Para facilitar la comunicación entre la biblioteca database/sql y una implementación concreta de un sistema de base de datos, se debe cargar un controlador *(driver)* que indique a la biblioteca database/sql cómo entablar comunicación con una base de datos concreta.

Para cargar el controlador de SQLite en nuestros programas Go, se debe importar el módulo externo github.com/mattn/go-sqlite3 de la siguiente manera:

```
import (
  "database/sql"
  _ "github.com/mattn/go-sqlite3"
)
```

Observe que se ha especificado un alias de paquete mediante el identificador vacío _ para evitar un error de "paquete no utilizado", ya que go-sqlite3 debe cargarse explícitamente (su función init() inicializa el controlador), pero no se usa directamente (nuestro código utilizará el paquete sql).

Si desea utilizar otras bases de datos SQL, puede utilizar alguno de los siguientes controladores (Figura 19.1):

Base de datos	Nombre de controlador	Módulo a importar
MySQL	`mysql`	github.com/go-sql-driver/mysql
Oracle	`godror`	github.com/godror/godror
Oracle	`oci8`	github.com/mattn/go-oci8
PostgreSQL	`pgx`	github.com/jackc/pgx
PostgreSQL	`postgres`	github.com/lib/pq
SQL Server	`sqlserver`	github.com/denisenkom/go-mssqldb
SQLite	`sqlite3`	github.com/mattn/go-sqlite3

Figura 19.1 Controladores para diversas bases de datos SQL.

La web oficial github.com/golang/go/wiki/SQLDrivers contiene un listado completo de todos los controladores registrados para un gran número de bases de datos.

19.2 ABRIENDO UNA BASE DE DATOS

La función sql.Open permite abrir una base de datos:

```
func
Open(
  nombreControlador, datosConexion string)  (*sql.DB, error)
```

El primer argumento debe ser el nombre del controlador, tal y como se especifica en la segunda columna de la tabla en la Figura 19.1. El segundo son los datos de conexión a una base de datos concreta. Suele consistir en la URL del servidor, las credenciales del usuario, el nombre de la base de datos, etc. La documentación oficial de cada base de datos explicará los detalles para dicha conexión.

En el caso de SQLite, los argumentos serán el nombre del controlador sqlite3 y un nombre de archivo de disco, donde se guardarán los datos:

```
bd, err := sql.Open("sqlite3", "archivo.db")
```

Si la base de datos se ha abierto con éxito, sql.Open retorna un apuntador a sql.DB, un controlador que permite operar con la base de datos.

19.3 MODIFICANDO LA BASE DE DATOS

El método Exec de sql.DB permite ejecutar comandos que modifican el contenido de la base de datos.

Por ejemplo, la siguiente invocación a Exec crearía una tabla SQL llamada Alumnos, con los atributos nota (número), id y nombre (textos):

```
result, err := bd.Exec(`CREATE TABLE IF NOT EXISTS
    Alumnos(
        id     VARCHAR PRIMARY KEY,
        nombre VARCHAR,
        nota   FLOAT
    )`)
```

El primer elemento retornado, del tipo sql.Result retorna información opcional acerca de las filas afectadas por dicha operación. Puede ignorarse si no resulta de interés para el programa.

Cuando una orden ha de incluir información proporcionada por el usuario, se deben usar los comodines "?", y se deben proporcionar los datos como argumentos extra. Por ejemplo, dado un tipo Alumno de Go:

```
type Alumno struct {
  Nombre string
  Id     string
  Nota   float64
}
```

la inserción de un alumno en la tabla Alumnos SQL debería parametrizarse de la siguiente manera:

```
func Inserta(bd *sql.DB, a Alumno) error {
  _, err := bd.Exec(`
        INSERT INTO Alumnos(id, nombre, nota)
        VALUES (?, ?, ?)`, a.Id, a.Nombre, a.Nota)
  return err
}
```

¿Por qué usar los comodines de interrogación y no componer directamente el comando SQL con fmt.Sprint? Principalmente porque si los datos del alumno son introducidos por un usuario, un usuario malintencionado podría escribir comandos SQL en los datos del alumno. Por ejemplo, un alumno llamado "); DROP TABLE Alumnos; borraría todos los datos de la base de datos. Este tipo de vulnerabilidad es muy grave y se conoce como “inyección de SQL”.
Al usar bd.Exec con comodines “?”, Go comprueba que los datos introducidos en esos comodines no son comandos SQL, protegiéndonos contra ese tipo de ataques.

19.4 CONSULTANDO DATOS

El comando Query permite ejecutar comandos SQL del tipo SELECT, y retorna un apuntador a un objeto sql.Rows, que permite iterar por los resultados encontrados en la base de datos. Query también permite argumentos especificados como comodines "?", para que nuestras consultas sean seguras contra ataques de inyección de SQL.

Ejemplo:

```
func BuscaPorNota(
  bd *sql.DB, min, max float64) ([]Alumno, error) {


  encontrados, err := bd.Query(`
        SELECT id, nombre, nota
        FROM Alumnos
        WHERE nota >= ? AND nota < ?
        ORDER BY nota
      `, min, max)
  if err != nil {
    return nil, err
  }
  var alumnos []Alumno
  for encontrados.Next() {
    a := Alumno{}
    err := encontrados.Scan(&a.Id, &a.Nombre, &a.Nota)
    if err != nil {
      return nil, err
    }
    alumnos = append(alumnos, a)
  }
  return alumnos, nil
}
```

El ejemplo anterior busca todos los alumnos cuya nota esté comprendida entre dos números. Para evitar ataques de inyección de SQL, la invocación a Query se hace mediante comodines, y los valores concretos son añadidos al

final, como argumentos extra.

El método Next() de la variable encontrados (del tipo *sql.Rows) retornará true por cada fila de resultados recorrida (false cuando ya se hayan leído todos los resultados).

Para cada fila retornada, el método Scan recibe apuntadores (como en fmt.Scan) a los valores de la fila, en el orden en el que se especificaron en el comando SELECT.

Resumiendo, la función BuscaPorNota ejecuta una consulta parametrizada sobre la tabla Alumnos; luego recorre todos los resultados de la consulta (podría ser que no hubiera resultados) y, para cada resultado, crea un objeto Alumno de Go, donde escribe los valores de dicho resultado.

19.5 DECLARACIONES PREPARADAS

La ejecución de los métodos Query o Exec del tipo sql.DB es relativamente lenta, ya que requiere que la base de datos interprete el mismo comando SQL cada vez que se ha de ejecutar. Si un mismo comando ha de repetirse muchas veces, su rendimiento mejorará si el proceso de interpretación del lenguaje SQL se realiza una sola vez y la ejecución se realiza múltiples veces.

El método Prepare de sql.DB permite preprocesar un comando SQL, que más adelante podrá parametrizarse. Retorna un apuntador al tipo sql. Stmt, que permite ejecutar métodos como Query o Exec tan solo especificando los argumentos del comando.

Para acelerar las funciones Inserta y BuscaPorNota de la sección anterior, se pueden guardar los comandos preprocesados y listos para ser utilizados en un nuevo tipo de dato:

```
type AccesoDatos struct {
  bd           *sql.DB
  inserta      *sql.Stmt
  buscaPorNota *sql.Stmt
}

func NuevoAccesoDatos(bd *sql.DB) (AccesoDatos, error) {
  var err error
  a := AccesoDatos{bd: bd}
  a.inserta, err = bd.Prepare(`
        INSERT INTO Alumnos(id, nombre, nota)
        VALUES (?, ?, ?)`)
  if err != nil {
    return a, err
  }
  a.buscaPorNota, err = bd.Prepare(`
        SELECT id, nombre, nota FROM Alumnos
        WHERE nota >= ? AND nota < ?
        ORDER BY nota
  `)
  return a, err
}
```

En este caso, los comandos SQL utilizados en Inserta y BuscaPorNota se preparan en un constructor, y se almacenan en el struct AccesoDatos.
Ahora, para insertar un dato, en vez de invocar el método Exec de sql. DB, se invoca Exec de sql.Stmt, que funciona de manera análoga, con la excepción de que no recibe el comando SQL sino directamente los parámetros:

```
func (d *AccesoDatos) Inserta(a Alumno) error {
  _, err := d.inserta.Exec(a.Id, a.Nombre, a.Nota)
  return err
}
```

Observe que, en el ejemplo, se ha cambiado una función global por un método que usa AccesoDatos como receptor.
De manera similar, la búsqueda por nota ahora invocaría a Query sobre el sql.Stmt guardado:

encontrados, err := d.buscaPorNota.Query(min, max)

El recorrido por las filas resultantes sería exactamente igual.

19.6 TRANSACCIONES

A menudo, una modificación sobre una base de datos puede requerir diversas operaciones de inserción, borrado o modificación. Pudiera ser, además, que si una de esas operaciones fallara, las demás operaciones debieran abortarse (incluso las que ya se han realizado).

Si se engloba un conjunto de operaciones bajo una transacción, estas no tendrán efecto sobre la base de datos hasta que no se realice una operación de Commit. En caso de querer abortar una transacción, se debe invocar al método Rollback, para que las operaciones anteriormente ejecutadas se deshagan.

Para crear una transacción, se debe ejecutar uno de los siguientes métodos del tipo db.DB:

```
func (db *DB) Begin()  (*Tx, error) func
(db *DB) BeginTx(
        ctx context.Context, opts *TxOptions)  (*Tx, error)
```

Begin devuelve una transacción por defecto (guardada en el tipo sql.Tx), mientras que BeginTx permite crear una transacción sujeta a un contexto, así como establecer algunas opciones de configuración.

El tipo sql.Tx tiene algunos métodos comunes a sql.DB y sql.Stmt, como Query, Exec, Prepare, etc. Aunque en el próximo ejemplo de uso es interesante el siguiente método:

```
func (tx *Tx) Stmt(stmt *Stmt) *Stmt
```

que adapta un sql.Stmt ya creado al contexto de una transacción concreta.

En el siguiente ejemplo se insertan todos los alumnos pasados como argumento. Si la inserción de un solo alumno falla, ningún alumno se insertará realmente en la base de datos, al ser todas las operaciones de inserción parte de una misma transacción:

```
func (d *AccesoDatos) InsertaTodos(aula []Alumno) error {
  tx, err := d.bd.Begin()
  if err != nil {
    return err
  }
  insertaTx := tx.Stmt(d.inserta)
  for _, a := range aula {
    _, err := insertaTx.Exec(a.Id, a.Nombre, a.Nota)
    if err != nil {
      tx.Rollback()
      return err
    }
  }
  return tx.Commit()
}
```

En el anterior ejemplo, se crea una transacción mediante d.bd.Begin(); la declaración preparada inserta se prepara para esa transacción concreta mediante tx.Stmt. En ese momento, no se usa d.inserta sino la declaración preparada insertaTx.

Si la inserción de un alumno (insertaTx.Exec) falla, se invoca al método tx.Rollback(), que deshace cualquier inserción anterior.

Solo en el caso de que todas las inserciones hayan sido exitosas, se invoca a tx.Commit() para que estas, definitivamente, tengan efecto sobre la base de datos.

19.7 RESERVA DE CONEXIONES

Establecer una conexión a una base de datos suele ser un proceso relativamente costoso, por lo que la biblioteca estándar de Go trata de reutilizar las conexiones entre distintas operaciones. Sin embargo, puede ser necesario guardar diversas conexiones en memoria, sobre todo en servidores que despachan peticiones simultáneas de múltiples usuarios.

El tipo sql.DB permite gestionar la reserva de conexiones (conocido en inglés como *connection pool)* mediante los siguientes métodos:

- func (db *DB) SetMaxIdleConns(n int)
  - Permite especificar el número máximo de conexiones **no usadas** (listas para ser reutilizadas). Por defecto, son 2.
- func (db *DB) SetMaxOpenConns(n int)
  - Permite especificar el número máximo de conexiones simultáneas, o o si no hay límite (valor por defecto: o) .
- func (db *DB) SetConnMaxIdleTime(d time.Duration)
  - Especifica el tiempo máximo que una conexión puede permanecer en el estado de "no usada" antes de cerrarse. El valor o indica que la conexión nunca expirará.
- func (db *DB) SetConnMaxLifetime(d time.Duration)
  - Especifica el tiempo máximo de vida de una conexión (esté siendo usada o no). El valor o indica que la conexión nunca expirará.

Capítulo 20
PRUEBAS AUTOMATIZADAS DE SOFTWARE

Las pruebas (o tests) automatizadas de código se han convertido en una parte esencial del proceso de creación y mantenimiento de software.

Una prueba automatizada es un código que ejecuta partes aisladas de un programa (funciones, métodos...), y compara los resultados de dicha ejecución (retorno de funciones, atributos de estructuras...) con los resultados esperados.

Cuando los resultados obtenidos tras la ejecución de prueba concuerdan con los resultados esperados, se dice que la prueba ha "pasado" o "finalizado exitosamente". En caso contrario, se dice que la prueba ha "fallado".

Go proporciona un sistema de pruebas simple, basado en convenciones comunes, que puede ampliarse o enriquecerse con módulos externos.

En este capítulo se mostrarán las partes básicas del sistema de pruebas o tests que vienen con el paquete testing de la librería estándar de Go.

20.1 CÓDIGO A PROBAR: LA FUNCIÓN FACTORIAL

Se define la función factorial de un número N como "el producto de todos los números enteros positivos desde 1 hasta N". Recursivamente, puede definirse como:

- Si N == 0, Factorial(N) = 1
- Si N != 0, Factorial(N) = N * Factorial(N-1)

Implementaremos este código en Go mediante la siguiente función (¡OJO! La implementación contiene un error introducido adrede):

```
package fact

func Factorial(n uint64) uint64 {
  if n == 0 {
    return 1
  }
  return n + Factorial(n-1)
}
```

El código anterior se ha escrito en un fichero llamado fact.go.

20.2 EL PAQUETE testing

Para probar la implementación de Factorial, se podría crear una función main() con diversas invocaciones a Factorial, mostrando en pantalla el resultado de dichas operaciones y comprobando manualmente que estas sean correctas. Sin embargo, estas pruebas manuales tienen los siguientes inconvenientes:

- Podríamos equivocarnos, por un despiste, a la hora de comprobar la corrección en los valores retornados por Factorial.
- Si cambiáramos el código, o cambiara alguno de los componentes o bibliotecas de las cuales depende nuestro código, deberíamos volver a repetir manualmente todos los errores. Además de ser pesado, aburrido y una pérdida de tiempo, esto es inviable a medida que nuestros programas crecen en tamaño y complejidad.

Para automatizar el proceso, se utilizará el sistema de pruebas estándar de Go:

1. En el mismo directorio donde se encuentra el archivo fact.go , crearemos un archivo llamado fact_test.go. Los archivos de pruebas en Go han de finalizar con el sufijo _test.go.
2. Cada archivo de pruebas alberga una o múltiples funciones cuyo nombre empieza por Test, con un único argumento del tipo *testing.T.
3. En cada función Test, se invoca a una parte aislada del código y se comprueba que esta funciona. En este caso, invocaremos a Factorial con un número cualquiera, y comprobaremos que el valor retornado coincide con el valor que nosotros hayamos calculado previamente.
4. Si se detecta que el test ha de fallar (porque el resultado no es el esperado), se invoca el método Error o Errorf del argumento *testing.T, que mostrará un mensaje que indique por qué la prueba ha fallado.

Código de la prueba en fact_test.go:

```
package fact

import "testing"

func TestFactorial(t *testing.T) {
  f := Factorial(3)
  if f != 6 {
    t.Errorf("Factorial(3): Esperaba 6. Retornó %d", f)
  }
}
```

Para ejecutar la prueba, nos podemos situar en la carpeta donde están las pruebas y ejecutar desde la línea de comandos:

go test .

La salida a este comando será algo parecido a:

```
--- FAIL: TestFactorial (0.00s)
    fact_test.go:13: Factorial(3): Esperaba 6. Retornó 7
FAIL
FAIL    /carpeta/testing/fact    0.523s
FAIL
```

¡Vaya! Nuestro primer test falló. Si revisamos el código de Factorial, podemos ver que en la implementación original se introdujo una suma donde debería ir una multiplicación. Lo corregimos, y el nuevo contenido de la función Factorial en el archivo fact.go será:

```
func Factorial(n uint64) uint64 {
  if n == 0 {
    return 1
  }
  return n * Factorial(n-1)
}
```

Tras ejecutar nuevamente go test ., se puede ver que la función Factorial se comporta según se espera. Salida estándar:

```
ok      /carpeta/testing/fact    0.326s
```

Para asegurarnos de que la función factorial funciona con varios números, se pueden incluir varios casos de prueba dentro de una misma función de test:

```
package fact

import "testing"

type ejemploTest struct {
  num      uint64
  esperado uint64
}

func TestFactorial(t *testing.T) {
  ejemplos := []ejemploTest{
    {num: 1,  esperado: 1},
    {num: 7,  esperado: 5040},
    {num: 10, esperado: 3628800},
    {num: 13, esperado: 6227020800},
  }
  for _, ej := range ejemplos {
    resultado := Factorial(ej.num)
    if resultado != ej.esperado {
      t.Errorf("Factorial(%d): Esperaba %d. Retornó %d",
        ej.num, ej.esperado, resultado)
    }
  }
}
```

20.3 PROBANDO SERVICIOS HTTP

El paquete net/http/httptest permite crear un servidor HTTP o HTTPS de pruebas, así como clientes que conecten a este servidor. Esto permite probar cualquier implementación de http.Handler sin necesidad de configurar un servidor Web completo.

El siguiente ejemplo prueba el servidor de documentos que se mostró como ejemplo en el capítulo 17 (tipo Rest). Para ello:

- Se invoca a la función httptest.NewServer, que recibe como argumento el http.Handler a probar (el servicio Rest del capítulo 17), y retorna un servidor de test.
- Se crea un cliente llamando a la función Client() del servidor de test retornado.
- Se invoca a los métodos del cliente http de test, tomando como URL el atributo URL del servidor de test, y añadiendo la ruta que se requiera.

Ejemplo de prueba del servidor Rest del capítulo 17:

```
func TestPost(t *testing.T) {
  // DADO un servicio REST de guardado de documentos
  handler := Rest{entradas: map[string]string{}}
  servidor := httptest.NewServer(&handler)

  // CUANDO se inserta un documento que no existe
  cliente := servidor.Client()
  resp, err := cliente.Post(
    servidor.URL+"/Japon",
    "text/plain",
    strings.NewReader("Tokyo"),
  )
  if err != nil {
    t.Error(err)
  }
  // ENTONCES el servidor responde 200 OK
  if resp.StatusCode != http.StatusOK {
    t.Error("Esperaba 200 OK pero obtuve", resp.StatusCode)
  }

  // Y CUANDO se lee de nuevo ese documento
  resp, err = cliente.Get(servidor.URL + "/Japon")
  if err != nil {
    t.Error(err)
  }
  documento, err := ioutil.ReadAll(resp.Body)
  if err != nil {
    t.Error(err)
  }
```

```
  // ENTONCES el documento retornado es igual
  // al documento enviado (quitando el salto de línea
  // "\n" que el servidor añade)
  documento = bytes.Trim(documento, "\n")
  if string(documento) != "Tokyo" {
    t.Error("Esperaba Tokyo pero obtuve", string(documento))
  }
}
```

20.4 PRUEBAS DE RENDIMIENTO

Las pruebas de rendimiento (o *benchmarks,* en inglés) permiten medir la velocidad de ejecución de nuestro código.

Para crear una prueba de rendimiento en Go, en el mismo fichero de pruebas se pueden añadir funciones cuyo nombre empieza por Benchmark y que reciben un apuntador *testing.B como argumento.

Dentro de un bucle que se repite N veces (donde N es un atributo de *testing.B), se invoca el código cuyo rendimiento se pretende medir:

```
func BenchmarkFactorial(b *testing.B) {
  for i := 0; i < b.N; i++ {
    _ = Factorial(21)
  }
}
```

Para ejecutar las funciones de prueba de rendimiento, se debe ejecutar go test con el argumento -bench:

```
$ go test -bench .
goos: darwin
goarch: amd64
BenchmarkFactorial-4    12740894                88.3 ns/op
PASS
ok      /carpeta/testing/fact    1.630s
```

La línea antes del PASS (resultado de los tests) indica que se ha ejecutado la función BenchmarkFactorial y que el código del bucle se ha ejecutado 12 740 894 veces. La media de tiempo de ejecución para cada invocación a Factorial es de 88.3 nanosegundos.

Probemos a optimizar la función Factorial mediante una implementación iterativa, mucho más rápida que la recursiva:

```
func Factorial(n uint64) uint64 {
  f := uint64(1)
  for i := uint64(2); i <= n; i++ {
    f *= i
  }
  return f
}
```

Al ejecutar de nuevo go test -bench ., el terminal muestra las siguientes líneas:

```
goos: darwin
goarch: amd64
BenchmarkFactorial-4    126372938                9.47 ns/op
PASS
ok      /carpeta/testing/fact    2.466s
```

De las líneas de arriba se deduce que:

- Las pruebas han pasado. En principio, la nueva función Factorial sigue siendo una implementación válida.
- La nueva implementación de Factorial es un orden de magnitud más rápida que la anterior. ¡Se ha pasado de 88.3 a 9.47 nanosegundos!

20.5 COBERTURA DE LAS PRUEBAS

Es muy importante remarcar que pasar un juego de pruebas no garantiza que este código sea correcto o esté exento de fallos. Las pruebas pueden no estar ejecutando todas las posibilidades e instrucciones del código que se está probando.

La herramienta go tool cover permite conocer el grado de **cobertura** de las pruebas, es decir, cuán profundamente se está probando un código.

Lo primero que se debe hacer es generar un perfil de cobertura de las pruebas. Esto se consigue con el argumento -coverprofile de go test:

```
$ go test -coverprofile cover.out .
ok  /carpeta/test/fact 0.2s coverage: 100.0% of statements
```

La salida al código anterior nos indica que durante la pruebas se ha ejecutado el 100 % de las instrucciones contenidas en el paquete fact. Además, ha generado un archivo cover.out con el perfil de la cobertura.

Para analizar cover.out, se debe utilizar la herramienta go tool cover:

```
go tool cover -func cover.out
```

Salida:

```
goos: darwin
goarch: amd64
BenchmarkFactorial-4    126372938                9.47 ns/op
PASS
ok      /carpeta/testing/fact    2.466s
```

La salida anterior nos muestra el porcentaje de cobertura de todos los archivos y funciones del código, así como el total de cobertura. Como el ejemplo de esta sección solo contiene una función, solo aparece una entrada.

Como experimento, añadiremos la siguiente función al archivo fact.go:

```
func
```

```
NoProbado() {
  fmt.Println("hola!")
}
```

Ahora, al ejecutar las pruebas y mostrar el perfil de cobertura, se verá lo siguiente:

```
goos: darwin
   /carpeta/test/fact/fact.go:5:  Factorial       100.0%
   /carpeta/test/fact/fact.go:13: NoProbado       0.0%
   total:                         (statements)    80.0%
ok      /carpeta/testing/fact    2.466s
```

La función NoProbado no se ha ejecutado durante las pruebas. Por tanto, su cobertura es del 0 % y la cobertura total del código de ejemplo ha bajado al 80 %.

Además del porcentaje de cobertura, es útil saber qué instrucciones han sido ejecutadas durante las pruebas y cuáles no, de manera que resulte más sencillo refinar las pruebas para ampliar el porcentaje de cobertura.

Si se ejecuta la función go cover con el parámetro -html en vez de -func:

```
go tool cover -html cover.out
```

se abrirá una sesión en el navegador web con una lista desplegable donde poder seleccionar los diferentes archivos que se han probado. Para cada archivo mostrará el código al completo, con las líneas de código resaltadas en diversos colores:

- Verde: Líneas de código ejecutadas durante las pruebas.
- Rojo: Líneas de código no ejecutadas durante las pruebas.
- Gris: Líneas de código no ejecutables (declaración del paquete, imports, cabeceras de función, etc.).