The Little Go Book

by Karl Seguin

traducido por
Raúl Expósito

Sobre este Libro

Licencia

Última Versión

La última versión del código fuente de este libro puede encontrarse en: http://github.com/raulexposito/the-little-go-book

Introducción

Siempre he vivido una relación de amor-odio cuando se trata de aprender nuevos lenguajes. Por una parte, los lenguajes son tan fundamentales en lo que hacemos que incluso pequeños matices pueden provocar un impacto considerable. Ese momento *ajá* que se produce cuando algo encaja puede tener un efecto perdurable en nuestra forma de programar y puede redefinir nuestras expectativas en otros lenguajes. Por contra, el aprender nuevos lenguajes es algo que no termina nunca. Aprender nuevas palabras reservadas, tipos de datos, estilos de programación y descubrir librerías, comunidades y paradigmas supone un montón de trabajo a priori difícil de justificar. Comparados con cualquier otra cosa que tengamos que aprender, los nuevos lenguajes a menudo parecen ser una mala inversión de nuestro tiempo.

Dicho esto, *tenemos* que seguir adelante. *Tenemos* que dar los pasos porque, de nuevo, los lenguajes son el elemento principal de lo que hacemos. Aunque el aprendizaje sea continuo e infinito, éste ayuda a ampliar nuestra visión e impacta en nuestra productividad, la legibilidad de nuestro código, el rendimiento, la testeabilidad, la forma en la que gestionamos dependencias, la gestión de errores, la documentación, el profiling, nuestra presencia en comunidades, nuestro conocimiento de librerías estándar, y mucho más. ¿Hay alguna manera positiva de decir *la sarna con gusto no pica*?

Todo esto nos deja con una pregunta importante: **¿Por qué Go?** Para mí hay dos motivos convincentes: el primero es que es un lenguaje relativamente sencillo con una librería estandar relativamente simple. Go tiene la naturaleza de simplificar la complejidad que hemos visto incluida en los lenguajes de programación en el último par de décadas mediante el uso de varios mecanismos. El otro motivo es que para muchos desarrolladores puede servir como complemento para su arsenal.

Go fue construido para ser utilizado como un lenguaje orientado a software de sistemas (es decir, sistemas operativos, drivers de dispositivos), lo cual animó a desarrolladores de C y C++. Según el equipo de Go, y según la percepción que yo también tengo, los desarrolladores de aplicaciones, y no los desarrolladores de software de sistemas, se han convertido en los principales usuarios de Go. ¿Por qué? No puedo hablar con autoridad acerca de los desarrolladores de software de sistemas, pero para todos aquellos que construimos aplicaciones web, servicios, aplicaciones de escritorio y software por el estilo, nos ha cubierto la necesidad de disponer de algo que nos permita desarrollar aplicaciones de bajo y de alto nivel.

Quizá sea la mensajería, las cachés, los análisis de datos computacionalmente muy costosos, los interfaces por línea de comandos, los logs o la monitorización. No sé cómo etiquetarlo, pero a lo largo de mi vida laboral el software de sistemas ha ido creciendo en complejidad. Es *posible* construir estos sistemas con Ruby o Python o alguna otra cosa (como mucha gente hace), pero ese tipo de software puede beneficiarse de un sistema con un tipado rígido que brinde mejor rendimiento. Del mismo modo, *puedes* usar Go para construir aplicaciones web (como mucha gente hace), aunque yo todavía prefiero, con mucho margen, la expresividad de Node o Ruby para este tipo de desarrollos.

Hay otras áreas en las que Go es excelente. Por ejemplo, no existen dependencias a la hora de ejecutar programas compilados con Go. No hay que preocuparse sobre si los usuarios tienen Ruby o la JVM instalada o, si la tienen, qué versión tendrán. Por este motivo Go se está convirtiendo en un lenguaje popular con el que desarrollar aplicaciones de línea de comandos u otro tipo de programas de utilidades que necesiten ser distribuibles (por ejemplo, un recolector de log).

Hablando claramente, aprender Go es hacer un uso eficiente de tu tiempo. No es necesario invertir muchas horas para aprender o incluso dominar Go, y obtendrás algo práctico de tu esfuerzo.

Una Nota del Autor

He dudado sobre si escribir este libro o no por un par de motivos. El primero es que la propia documentación de Go, en particular Effective Go, es buena.

El otro es mi malestar por escribir un libro sobre un lenguaje de programación. Cuando escribí El Pequeño Libro de MongoDB podia dar por supuesto que la mayoría de los lectores entendían los conceptos básicos de bases de datos relaciones así como su modelado. Con El Pequeño Libro de Redis podía asumir que el lector estaba familiarizado con los conceptos de clave-valor y empezar por ahí.

Cuando pienso en los párrafos y capítulos que vienen adelante sé que no puedo hacer esas presunciones. ¿Cuánto tiempo será necesario utilizar para hablar sobre interfaces sabiendo que para algunos el concepto será nuevo, mientras que otros no necesitan saber más que que *Go tiene interfaces*? Me tomaré la libertad de pensar que me vas a hacer saber si algunas partes son demasido superficiales mientras que otras entran demasiado en detalle. Considéralo el precio a pagar por tener este libro.

Comenzando

Si buscas jugar un poco con Go deberías probar el Go Playground, el cual te permite ejecutar código en el navegador sin tener que instalar nada. Esta es, además, la forma más común de compartir código escrito en Go al pedir ayuda en los foros de Go y en lugares como StackOverflow.

La instalación de Go es bastante directa. Lo puedes instalar desde el código fuente, pero te aconsejo que utilices los binarios precompilados. Cuando vayas a la página de descargas verás instaladores para varias plataformas. Vamos a pasar de ellos y a aprender cómo instalar Go por nosotros mismos. Como verás no es difícil.

Excepto para los ejemplos sencillos, Go ha sido diseñado para funcionar siempre que tu código esté dentro de un workspace. El workspace es un directorio formado por los subdirectorios `bin`, `pkg` y `src`. Puede que estés tentado a forzar a Go a usar tu propio estilo - no lo hagas.

Por lo general coloco mis proyectos dentro de `~/code`. Por ejemplo, `~/code/blog` contiene mi blog. Para Go, mi workspace es `~/code/go` y mi blog vitaminado con Go debe estar en `~/code/go/src/blog`. Ya que hay mucho que escribir utilizo un enlace simbólico para poder acceder directamente desde `~/code/blog`:

```
ln -s ~/code/go/src/blog ~/code/blog
```

Para resumir, crea un directorio `go` con un subdirectorio `src` donde vayas a dejar tus proyectos,

OSX / Linux

Descarga el fichero `tar.gz` de tu plataforma. Para OSX, el que más te interesa es `go#.#.#.darwin-amd64-osx10.8.tar.gz`, donde `#.#.#` es la última versión de Go.

Descomprime el fichero en `/usr/local` con el comando `tar -C /usr/local -xzf go#.#.#.darwin-amd64-osx10.8.tar.gz`.

Configura dos variables de entorno:

1. `GOPATH` debe apuntar a tu workspace, para mí es `$HOME/code/go`.
2. Necesitamos añadir el binario de Go en nuestro `PATH`.

Para hacer esto desde la terminal lanzamos los comandos:

```
echo 'export GOPATH=$HOME/code/go' >> $HOME/.profile
echo 'export PATH=$PATH:/usr/local/go/bin' >> $HOME/.profile
```

Necesitarás activar ambas variables. Para ello puedes cerrar y reabrir tu terminal o puedes ejecutar el comando `source $HOME/.profile`.

Escribe `go version` y tendrás una salida del estilo a `go version go#.#.# darwin/amd64`.

Windows

Descarga la última versión del fichero en formato zip. Si tu máquina es x64 necesitarás bajar `go#.#.#.windows-amd64.zip`, donde `#.#.#` es la última versión de Go.

Descomprímelo en el directorio que prefieras. `c:\Go` es una buena elección.

Configura dos variables de entorno:

1. `GOPATH` debe apuntar a tu workspace. Deberá ser algo del estilo `c:\users\goku\work\go`.
2. Añade `c:\Go\bin` en tu variable de entorno `PATH`.

Las variables de entorno se pueden configurar a través del botón `Variables de entorno` de la pestaña `Avanzado` de la opción `Sistema` del panel de control. Esto puede variar según la versión de Windows.

Abre un terminal y escribe `go version`. Si todo ha ido bien tendrás una salida del estilo `go version go#.#.# windows/amd64`.

Capítulo 1 - Lo Básico

Go es un lenguaje compilado y estáticamente tipado, con una sintaxis similar a la de C y recolección de basura. ¿Qué significa todo esto?

Compilación

La compilación es el proceso de traducir el código fuente que escribas en un lenguaje de más bajo nivel – puede ser ensamblador (como en el caso de Go) u otro tipo de lenguaje intermedio (como en el caso de Java o C#)

Trabajar con lenguajes compilados puede ser poco gratificante ya que el proceso de compilación puede ser lento. Es difícil iterar rápidamente cuando tienes que esperar minutos u horas a que el código compile. La velocidad de compilación es una de las principales metas en el diseño de Go, lo cual es una buena noticia tanto para los que trabajan con proyectos grandes como para aquellos que estamos acostumbrados a tener el feedback inmediato que ofrecen los lenguajes interpretados.

Los lenguajes compilados suelen ser más rápidos y los ejecutables pueden funcionar sin depedencias adicionales (al menos esto es cierto para lenguajes como C, C++ y Go, los cuales compilan directamente en ensamblador)

Estáticamente Tipado

Estar tipado estáticamente implica que las variables deben ser de un tipo específico (int, string, bool, []byte, etc). Esto se consigue indicando el tipo a la hora de definir la variable o, en muchos casos, dejando al compilador que infiera el tipo (veremos ejemplos de esto en breve).

Se puede decir mucho más sobre el tipado estático, pero creo que se entiende mejor viendo directamente el código fuente. Si estás acostumbrado a trabajar con lenguajes dinámicamente tipados encontrarás algo torpe esta práctica. No estás equivocado, pero hay ventajas, especialmente cuando unes tipado estático con compilación. Ambos se combinan a menudo y, por lo general, cuando tienes uno tienes el otro, ya que le permite al compilador ser capaz de detectar problemas más allá de los errores sintáticos y de realizar mejores optimizaciones cuando el sitema de tipado es rígido.

Sintaxis Similar a C

Que Go tenga una sintaxis similar a C implica que si estás acostumbrado a utilizar otros lenguajes como C, C++, Java, JavaScript o C#, Go te resultará familiar – al menos de modo superficial. Por ejemplo, `&&` se utiliza como un AND booleano, `==` se emplea para comparar igualdad, `{` y `}` comienzan y finalizan un ámbito y los arrays empiezan por 0.

La sintaxis de C suele implicar que las líneas deban terminar en punto y coma y que debe haber paréntesis que delimitan condicionales. Go acaba con estas prácticas, aunque los paréntesis se siguen usando para controlar la precedencia de operadores. Por ejemplo, un `if` tiene el siguiente aspecto:

```
if name == "Leto" {
  print("the spice must flow")
}
```

Aunque en casos más complicados los paréntesis siguen siendo útiles:

```
if (name == "Goku" && power > 9000) || (name == "gohan" && power < 4000)  {
  print("super Saiyan")
}
```

Cabe indicar que Go es mucho más cercano a C que C# o Java - no sólo en términos de sintaxis, sino tambien en objetivos. Esto queda reflejado en la sequedad y simplicidad del lenguaje, el cual empezará a resultar muy sencillo a medida que lo vayas aprendiendo.

Recolección de Basura

Algunas variables, al ser creadas, tienen un ciclo de vida claramente delimitado. Una variable local a una función, por ejemplo, desaparece cuando la función termina. En otros casos el escenario no es tan obvio – al menos para el compilador. Por ejemplo, el ciclo de vida de una variable devuelta por una función o referenciada por otras variables y objetos puede ser más complicado de determinar. Sin el recolector de basura es tarea de los desarrolladores el liberar la memoria asociada a estas variables cuando el desarrollador sepa que la variable va a dejar de ser utilizada. ¿Cómo? En C, ejecutando el comando `free(str);` sobre la variable.

Los lenguajes con recolectores de basura (e.j., Ruby, Python, Java, JavaScript, C#, Go) son capaces de realizar un seguimiento de las variables y liberarlas cuando no se utilicen más. La recolección de basura añade overhead, pero también elimina bugs devastadores.

Ejecutar Código Go

Vamos a comenzar nuestro camino creando un programa simple y aprendiendo a compilarlo y ejecutarlo. Para ello, abre tu editor de texto favorito y escribe el código siguiente:

```
package main

func main() {
  println("it's over 9000!")
}
```

Guarda el fichero con el nombre **`main.go`**. De momento, puedes guardarlo donde quieras, los ejemplos triviales no necesitan estar dentro del workspace de Go.

A continuación, abre una shell y cambia al directorio donde guardaste el archivo. En mi caso, esto implica escribir `cd ~/code`.

Para terminar, ejecuta el programa escribiendo:

```
go run main.go
```

Si todo ha funcionado deberías ver *it's over 9000!*.

Pero espera, ¿qué hay de la compilación?. `go run` es un comando muy práctico que compila *y* ejecuta tu código. Utiliza un directorio temporal para construir el programa, lo ejecuta y lo borra. Puedes ver la ubicación del fichero temporal ejecutando:

```
go run --work main.go
```

Para compilar explícitamente, utiliza `go build`:

```
go build main.go
```

Esto generará un `main` ejecutable que podrás utilizar donde necesites. No olvides que en Linux / OSX es necesario poner un prefijo al ejecutable con punto-barra, así que deberás escribir `./main`.

A la hora de desarrollar puedes utilizar tanto `go run` como `go build`. Sin embargo, cuando despliegues tu código, necesitarás generar un binario mediante `go build` y ejecutarlo.

Main

Espero que el código que acabamos de ejecutar sea entendible. Hemos creado una función y hemos mostrado un string con la función predefinida `println`. ¿El comando `go run` sabe qué tiene que ejecutar porque sólo tiene una elección? No. En Go, el punto de entrada de un programa es una función llamada `main` ubicada en el paquete `main`.

Hablaremos más de paquetes en el capítulo siguiente. De momento, mientras nos centramos en entender las bases de Go, vamos a escribir nuestro código en el paquete `main`.

Si quieres, puedes modificar el código anterior y cambiar el nombre del paquete. Si lo ejecutas mediante `go run` deberías obtener un mensaje de error. Tras esto, vuelve a dejar el nombre de `main` pero usa un nombre de función diferente, deberías ver otro mensaje de error. Prueba a hacer los mismos cambios pero ejecutando `go build` esta vez. Observa que el código compila, aunque no hay ningún punto de entrada para hacerlo funcionar. Esta situación es perfectamente normal cuando estás construyendo una librería.

Imports

Go incluye ciertas funciones predefinidas, como `println`, las cuales pueden ser utilizadas sin referenciar. No podemos ir muy lejos sin utilizar la librería estándar de Go y, eventualmente, sin usar librerías de terceros. En Go, la palabra clave `import` se utiliza para declarar qué paquetes van a ser utilizados por el código del fichero en el que estemos trabajando.

Vamos a cambiar nuestro programa:

```
package main

import (
  "fmt"
  "os"
)
```

```
func main() {
  if len(os.Args) != 2 {
    os.Exit(1)
  }
  fmt.Println("It's over ", os.Args[1])
}
```

El cual se puede ejecutar a través de:

```
go run main.go 9000
```

En este caso estamos usando dos paquetes estándar de Go: `fmt` y `os`. Hemos presentado, además, otra función predefinida: `len`. `len` devuelve el tamaño de un string, o el número de valores de un diccionario o, como podemos ver aquí, el número de elementos de un array. Si te estás preguntado que por qué espera 2 parámetros es porque el primer argumento – en el índice 0 – es siempre la ruta al ejecutable que está en marcha. (Puedes modificar el programa para que lo muestre y así lo puedas ver por ti mismo)

Es probable que hayas observado que ponemos el nombre del paquete antes que el nombre de la función: `fmt.Println`. Esta forma de trabajar es diferente a la que puedes conocer de otros lenguajes. Aprenderemos más sobre paquetes en los próximos capitulos: de momento, saber cómo importarlos y cómo usarlos es un buen punto de partida.

Go es estricto en lo que respecta a la importación de paquetes, tanto que tu código no compilará si importas un paquete que no utilizas. Prueba a ejecutar lo siguiente:

```
package main

import (
  "fmt"
  "os"
)

func main() {
}
```

Deberías obtener dos errores indicando que `fmt` y `os` están siendo importados pero no usados. ¿Puede ser esto molesto? Por supuesto que sí. A medida que vaya pasando el tiempo te acostumbrarás (aunque a veces puede seguir siendo molesto). Go es estricto en este sentido porque los import que no se utilicen pueden ralentizar la compilación; aunque es cierto que es un problema del cual la mayoría de nosotros no se preocupa.

Otro detalle a tener en cuenta es que la librería estándar de Go está bien documentada. Puedes echar un vistazo a http://golang.org/pkg/fmt/#Println para aprender más sobre la función `Println` que hemos usado. Puedes pinchar sobre el encabezado y ver el código fuente. Es más, en la parte de arriba de la página podrás saber más sobre las capacidades de formato de Go.

Si no tienes conexión a internet puedes obtener la documentación localmente ejecutando:

```
godoc -http=:6060
```

Y haciendo que tu navegador apunte a `http://localhost:6060`

Variables y Declaraciones

Estaría bien comenzar y terminar nuestro vistazo a las variables diciendo *puedes declarar y asignar una variable haciendo X = 4*. Por desgracia, las cosas son más complicadas en Go. Vamos a comenzar revisando ejemplos sencillos. Tras ello, en el próximo capítulo, aprenderemos a crear y utilizar estructuras. Quizá necesites algo de tiempo antes de sentirte cómodo con ellas.

Puede que pienses *¡Vaya! ¿Cómo de complicado puede ser eso?*. Vamos a comenzar revisando algunos ejemplos.

La forma más explícita de realizar declaraciones y asignaciones de variables en Go es aquella en la que hay que escribir más:

```
package main

import (
  "fmt"
)

func main() {
  var power int
  power = 9000
  fmt.Printf("It's over %d\n", power)
}
```

En este caso hemos declarado una variable `power` de tipo `int`. Por defecto, Go asigna un valor por defecto a las variables. A los enteros se les asigna `0`, a los booleanos `false`, a los strings `""`, etc. A continuación, hemos asignado el valor `9000` a nuestra varible `power`. Podemos unir las dos líneas:

```
var power int = 9000
```

Todavía hemos tenido que escribir mucho. Go tiene un operador muy práctico con el que declarar variables, `:=`, el cual puede inferir el tipo:

```
power := 9000
```

Es práctico y tambien se puede utilizar con funciones:

```
func main() {
  power := getPower()
}

func getPower() int {
  return 9001
}
```

Es importante que recuerdes que `:=` se utiliza para declarar la variable a la vez que se le asigna valor. ¿Por qué? Porque una variable no puede ser definida dos veces (al menos no en el mismo ámbito). Si tratas de ejecutar el siguiente código obtendrás un error:

```
func main() {
  power := 9000
  fmt.Printf("It's over %d\n", power)

  // ERROR DE COMPILACIÓN:
  // no hay variables nuevas a la izquierda de :=
  power := 9001
  fmt.Printf("It's also over %d\n", power)
}
```

El compilador se quejará con un *no hay variables nuevas a la izquierda de :=*. Esto significa que la primera vez que declaramos una variable debemos usar `:=`, pero para el resto de las asignaciones debemos usar el operador `=`. Esto tiene mucho sentido, pero te va a obligar a saber cuándo tienes que usar alguno de los dos.

Si lees el mensaje de error de cerca verás que *variables* es plural. Esto es así porque Go te permite asignar múltiples variables (usando tanto `=` como `:=`):

```
func main() {
  name, power := "Goku", 9000
  fmt.Printf("%s's power is over %d\n", name, power)
}
```

`:=` se puede utilizar ya que al menos una de las variables es nueva. Observa el siguiente ejemplo:

```
func main() {
  power := 1000
  fmt.Printf("default power is %d\n", power)

  name, power := "Goku", 9000
  fmt.Printf("%s's power is over %d\n", name, power)
}
```

Aunque es la segunda vez que se utiliza `:=` con `power`, el compilador no se quejará ya que verá que `name` es una variable nueva y, por tanto, permite `:=`. Sin embargo, no puedes cambiar el tipo de `power`. Fue declarada (implícitamente) como entera y, por ello, sólo se le pueden asignar enteros.

Lo último que debes saber por ahora es que, como ocurre con los imports, Go no te va a permitir tener variables sin utilizar. Por ejemplo:

```
func main() {
  name, power := "Goku", 1000
  fmt.Printf("default power is %d\n", power)
}
```

no compilará ya que `name` ha sido declarada pero no se utiliza. Puede, al igual que ocurre con los imports no utilizados, causar frustración, pero en líneas generales creo que ayuda a mantener la limpieza del código y la legibilidad.

Hay más cosas que aprender sobre declaaciones y asignaciones. De momento, recuerda que puedes usar `var NOMBRE TIPO` cuando declaras una variable y le asignas su valor por defecto, `NOMBRE := VALOR` cuando declaras y asignas valor a una variable, y `NOMBRE = VALOR` cuando asignas valor a una variable previamente declarada.

Declaración de Funciones

Es un buen momento para indicar que las funciones pueden devolver más de un valor. Vamos a echar un ojo a las siguientes tres funciones: una primera sin valor de retorno, una segunda con un valor de retorno, y una tercera con dos valores de retorno.

```
func log(message string) {
}

func add(a int, b int) int {
}

func power(name string) (int, bool) {
}
```

Utilizamos la última de este modo:

```
value, exists := power("goku")
if exists == false {
  // manejar esta situación de error
}
```

A veces sólo estás interesado en uno de los valores de retorno. En esos casos, puedes asignar los otros valores a _:

```
_, exists := power("goku")
if exists == false {
  // manejar esta situación de error
}
```

Esto es más que una mera convención. _, el identificador blanco, es especial en el sentido en que el valor de retorno no será asignado. Esto te permite utilizar _ una y otra vez independientemente del tipo devuelto.

Para terminar, hay algo más que te gustará saber relacionado con la declaración de funciones. Se puede usar una sintaxis abreviada si los parámetros son del mismo tipo:

```
func add(a, b int) int {

}
```

A menudo utilizarás la capacidad de devolver más de un valor, y con frecuencia utilizarás _ para descartar algún valor. Nombrar los valores de retorno y la ligeramente abreviada declaración de parámetros no son tan comunes. De todos modos, antes o después te encontrarás con ellos así que es importante conocerlos.

Antes de Continuar

Hemos revisado una pequeña cantidad de piezas individuales que probablemente parezca que no encajan de momento. Poco a poco iremos construyendo ejemplos más grandes en los que las piezas comiencen a encajar.

Si provienes de un lenguaje dinámico quizá consideres que la complejidad de los tipos y las declaraciones son un paso atrás. No estoy en desacuerdo contigo. En determinados entornos los lenguajes dinámicos son, sin lugar a dudas, más productivos.

Probablemente te sientas cómodo con Go si procedes de un lenguaje estáticamente tipado. La inferencia de tipos y los múltiples valores de retorno son estupendos (aunque ciertamente no son exclusivos de Go). Iremos valorando su sintaxis clara y concisa a medida que vayamos aprendiendo más.

Capítulo 2 - Estructuras

Go no es un lenguaje orientado a objetos (OO) como C++, Java, Ruby o C#. No tiene objetos, ni herencia ni cosas por el estilo, así que tampoco tiene conceptos asociados con la OO como el polimorfismo o la sobrecarga.

Lo que tiene Go son estructuras que pueden ser asociadas con métodos. Go, además, posee un mecanismo sencillo y efectivo de composición. Como resultado el código es más sencilo, pero hay veces en las cuales se echan de menos algunas de las posibilidades que la OO ofrece. (Merece la pena destacar que la batalla *composición versus herencia* lleva mucho tiempo existiendo y que Go es el primer lenguaje que utilizo que defiende una posición concreta ante este dilema).

Aunque Go no hace OO del modo al que estamos acostumbrados, descubrirás un montón de similaridades entre la definición de una estructura y la definición de una clase. La definición de la estructura `Saiyan` es un ejemplo de ello:

```
type Saiyan struct {
  Name string
  Power int
}
```

Pronto veremos cómo añadir un método a esta estructura, lo cual se parece mucho a tener métodos en una clase. Antes de ponernos con ello vamos a meternos con las declaraciones.

Declaraciones e Inicializaciones

La primera vez que vimos variables y declaraciones sólo usamos tipos predefinidos como enteros o strings. Ahora que hemos hablado de estructuras vamos a ampliar el espectro para incluir punteros.

La forma más sencilla de crear un valor para nuestra estructura es:

```
goku := Saiyan{
  Name: "Goku",
  Power: 9000,
}
```

Nota: La última , del ejemplo anterior es obligatoria. Sin ella, el compilador dará un error. Acabarás apreciando que la consistencia sintáctica sea tan rígida, especialmente si has utilizado lenguajes que la tengan.

No tenemos por qué dar valor a todos los campos. Ambas declaraciones son válidas:

```
goku := Saiyan{}

// o

goku := Saiyan{Name: "Goku"}
goku.Power = 9000
```

Al igual que ocurre con las variables, los atributos también tienen un valor por defecto.

Es más, puedes omitir el nombre de los campos y basarte únicamente en el orden en el cual éstos están declarados (aunque en nombre de la claridad sólo deberias hacer esto en estructuras con pocos campos):

```
goku := Saiyan{"Goku", 9000}
```

Lo que hacen todos los ejemplos anteriores es declarar la variable `goku` y asignarle un valor.

Sin embargo, muchas veces no queremos una variable que esté asociada directamente con un valor, sino una variable que sea un puntero a ese valor. Un puntero es una dirección de memoria, es la ubicación donde podemos encontrar el valor. Es una indirección, la diferencia entre tener una casa y tener las señas de una casa.

¿Por qué podríamos querer tener un puntero al valor en vez del propio valor?. Tiene que ver con la forma en la que Go pasa parámetros a las funciones, ya que lo hace mediante copias. Sabiendo esto, ¿qué crees que aparecerá por consola?

```
func main() {
  goku := Saiyan{"Goku", 9000}
  Super(goku)
  fmt.Println(goku.Power)
}

func Super(s Saiyan) {
  s.Power += 10000
}
```

La respuesta es 9000, no 19000. ¿Por qué? Porque `Super` hizo cambios en una copia de nuestro `goku` original y, por eso, los cambios realizados en `Super` no se vieron reflejados en quien hizo la invocación. Para hacer que todo funcione como probablemente esperabas necesitamos pasar un puntero a nuestra variable:

```
func main() {
  goku := &Saiyan{"Goku", 9000}
  Super(goku)
  fmt.Println(goku.Power)
}

func Super(s *Saiyan) {
  s.Power += 10000
}
```

Hemos hecho dos cambios. El primero es la utilización del operador `&` para recuperar la dirección de nuestra variable (se llama operador *dirección de*). A continuación hemos modificado el tipo de parámetro que espera recibir `Super`. Antes esperaba un valor de tipo `Saiyan` pero ahora espera una dirección de tipo `*Saiyan`, donde `*X` significa *puntero a valor de tipo X*. Hay una relación obvia entre los tipos `Saiyan` y `*Saiyan`, pero son tipos distintos.

Observa que todavía estamos pasando una copia del valor de `goku` a `Super`, sólo que el valor de goku es ahora una dirección de memoria. Esta copia tiene la misma dirección que la original, que es en lo que se basa la indirección.

Imagina que copias las señas hacia un restaurante. Lo que tienes es una copia, pero esta copia también apunta al mismo restaurante que el original.

Podemos probar que es una copia tratando de cambiar el lugar donde apunta (no es algo que vaya a hacer lo que esperas que hiciese):

```
func main() {
  goku := &Saiyan{"Goku", 9000}
  Super(goku)
  fmt.Println(goku.Power)
}

func Super(s *Saiyan) {
  s = &Saiyan{"Gohan", 1000}
}
```

El código anterior, de nuevo, muestra 9000. Este comportamiento es propio de muchos lenguajes, incluidos Ruby, Python, Java y C#. Go y, en cierta medida, C#, simplemente hacen que el hecho sea visible.

Debería ser obvio que copiar un puntero es más barato en términos computacionales que copiar una estructura compleja. En una máquina de 64 bits un puntero ocupa 64 bits. Hacer copias puede ser caro si tenemos estructuras con muchos campos. El aporte real de los punteros es que te permiten compartir valores. ¿Queremos que `Super` modifique una copia de `goku` o que modifique al propio `goku`?

Con esto no quiero decir que siempre vayas a querer usar un puntero. Al final de este capítulo, cuando hayamos visto más sobre qué podemos hacer con estructuras, reexaminaremos la pregunta puntero-versus-variable.

Funciones sobre Estructuras

Podemos asociar un método con una estructura:

```
type Saiyan struct {
  Name string
  Power int
}

func (s *Saiyan) Super() {
  s.Power += 10000
}
```

En el código anterior podemos decir que el tipo `*Saiyan` es el **receptor** del método `Super`. Podemos llamar a `Super` de este modo:

```
goku := &Saiyan{"Goku", 9001}
goku.Super()
fmt.Println(goku.Power) // mostrará 19001
```

Constructores

Las estructuras no tienen constructores. Sin embargo, puedes crear una función que devuelva una instancia del tipo deseado en su lugar (como un patrón factory):

```
func NewSaiyan(name string, power int) *Saiyan {
  return &Saiyan{
    Name: name,
    Power: power,
  }
}
```

Nuestra factoría no tiene por qué devolver un puntero; este trozo de código es absolutamente válido:

```
func NewSaiyan(name string, power int) Saiyan {
  return Saiyan{
    Name: name,
    Power: power,
  }
}
```

New

A pesar de la falta de constructores, Go tiene la función predefinida `new` que se utiliza para reservar la memoria que un tipo concreto necesita. El resultado de `new(X)` es el mismo que `&X{}`:

```
goku := new(Saiyan)
// es lo mismo que
goku := &Saiyan{}
```

Cuál uses depende de ti, pero verás que la mayoría de la gente prefiere el segundo cuando hay campos que inicializar, ya que suele ser más fácil de leer:

```
goku := new(Saiyan)
goku.name = "goku"
goku.power = 9001

//vs

goku := &Saiyan {
  name: "goku",
  power: 9000,
}
```

Independientemente de la aproximación que sigas, utilizando el patrón factory que vimos anteriormente podrás aislar tu código de tener que saber y preocuparte sobre cuáles son los detalles de reserva de memoria.

Campos de una Estructura

`Saiyan`, en el ejemplo que hemos visto hasta ahora, tiene dos campos: `Name` y `Power`, de tipos **`string`** e **`int`**, respectivamente. Los campos pueden ser de cualquier tipo – incluyendo otras estructuras y tipos que no hemos explorado todavía tales como arrays, mapas, interfaces y funciones.

Por ejemplo, podemos hacer crecer nuestra definición de `Saiyan`:

```
type Saiyan struct {
  Name string
  Power int
  Father *Saiyan
}
```

la cual puede ser inicializada así:

```
gohan := &Saiyan{
  Name: "Gohan",
  Power: 1000,
  Father: &Saiyan {
    Name: "Goku",
    Power: 9001,
    Father: nil,
  },
}
```

Composición

Go incluye el concepto de composición, que consiste en incluir una estructura dentro de otra. En algunos lenguajes esto se conoce como trait o mixin. Los lenguajes que no tienen mecanismos de composición explícitos pueden hacerlo siguiendo el camino largo. En Java:

```
public class Person {
  private String name;

  public String getName() {
    return this.name;
  }
}

public class Saiyan {
  // Saiyan is said to have a person
  private Person person;

  // we forward the call to person
```

```
  public String getName() {
    return this.person.getName();
  }
  ...
}
```

Esto puede convertirse en algo bastante tedioso. Cada uno de los métodos de `Person` necesita ser duplicado en `Saiyan`. Go evita esta tediosidad:

```
type Person struct {
  Name string
}

func (p *Person) Introduce() {
  fmt.Printf("Hi, I'm %s\n", p.Name)
}

type Saiyan struct {
  *Person
  Power int
}

// cómo usarlo:
goku := &Saiyan{
  Person: &Person{"Goku"},
  Power: 9001,
}
goku.Introduce()
```

La estructura `Saiyan` posee un campo de tipo `*Person`. Dado que no le hemos puesto explícitamente un nombre podemos implícitamente acceder a los campos y funciones del tipo compuesto. Sin embargo, el compilador de Go *le puso* un nombre al campo. Ambas expresiones son perfectamente válidas:

```
goku := &Saiyan{
  Person: &Person{"Goku"},
}
fmt.Println(goku.Name)
fmt.Println(goku.Person.Name)
```

Las dos expresiones anteriores mostrarán "Goku".

¿Es la composición mejor que la herencia? Mucha gente piensa que es una forma más robusta de compartir código. Cuando utilizas herencia, tus clases están firmemente acopladas a su superclase y finalmente el esfuerzo se centra más en la jerarquía que en el comportamiento.

Sobrecarga

Aunque la sobrecarga no sea un concepto específico de las estructuras merece la pena mencionarlo. Go, sencillamente, no soporta sobrecarga. Por este motivo verás (y escribirás) muchas funciones con la firma `Load`, `LoadById`, `LoadByName`, etc.

Sin embargo, puesto que la composición implícita es un truco del compilador, podemos "sobrecargar" las funciones de un tipo compuesto. Por ejemplo, nuestra estructura `Saiyan` puede tener su propia función `Introduce`:

```
func (s *Saiyan) Introduce() {
  fmt.Printf("Hi, I'm %s. Ya!\n", s.Name)
}
```

La versión original siempre estará accesible a través de `s.Person.Introduce()`.

Punteros versus Valores

A medida que escribas código en Go te preguntarás *¿debería ser esto un valor, o un puntero al valor?* Hay dos buenas noticias. Primera: la respuesta es la misma independientemente de la naturaleza de lo que quieras utilizar:

- Una asignación sobre una variable local
- Un campo de una estructura
- El valor de retorno de una función
- Los parámetros de una función
- El receptor de un método

La segunda: utiliza un puntero si no estás seguro.

Como hemos visto, pasar valores es una buena forma de conseguir que los datos sean inmutables (los cambios que una función haga sobre ellos no serán reflejados en el código que lo haya invocado). Lo más frecuente es que este comportamiento sea el que deseas, otras veces no.

Hay que considerar el coste computacional de crear una copia de una estructura grande incluso si no tienes intención de cambiar los datos que contiene. Por el contrario, puede que tengas estructuras pequeñas, tales como:

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

En estos casos, el coste de copiar la estrcutura probablemente sea desplazado por la capacidad de poder acceder a `X` e `Y` directamente sin realizar ningún tipo de indirección.

Una vez más todos estos casos son muy sutiles. A no ser que estés iterando con miles o cientos de miles de elementos no percibirás ninguna diferencia.

Antes de Continuar

Visto desde un punto de vista práctico este capítulo ha presentado las estructuras, cómo crear una instancia de una estructura desde una función y ha incluido los punteros a tu conocimiento sobre el sistema de tipado de Go. En los siguientes capítulos trabajaremos tanto con lo que ya sabemos de estructuras como con el resto de conceptos que hemos explorado.

Capítulo 3 - Mapas, Arrays y Slices

De momento hemos visto tipos simples y estructuras. Es el momento de conocer los arrays, los slices y los mapas.

Arrays

Si vienes de Python, Ruby, Perl, JavaScript o PHP (entre otros), probablemente estés acostumbrado a programar con *arrays dinámicos*, los cuales son arrays que se redimensionan cuando se añaden datos en ellos. En Go, como en muchos otros lenguajes, los arrays tienen un tamaño fijo. Declarar un array requiere especificar qué tamaño tiene y, una vez este tamaño ha sido especificado, no puede crecer.

```
var scores [10]int
scores[0] = 339
```

El array anterior puede almacenar hasta 10 puntuaciones usando los índices que van desde `scores[0]` hasta `scores[9]`. Intentar acceder a un índice del array fuera de este rango provocará un error en compilación o en ejecución.

Es posible inicializar el array con valores:

```
scores := [4]int{9001, 9333, 212, 33}
```

Podemos utilizar **`len`** para obtener la longitud del array. **`range`** puede ser utilizado para iterar sobre él:

```
for index, value := range scores {


}
```

Los arrays son eficientes a la par que rígidos, pero a menudo no podemos saber por adelantado qué cantidad de elementos vamos a tener que manejar, es por ello que utilizamos slices.

Slices

En Go raramente, si es que ocurre alguna vez, se utilizan arrays directamente: en su lugar utilizamos slices. Un slice es una estructura ligera que encapsula y representa una porción de un array. Hay varias formas de crear un slice, la primera es una ligera variación de cómo se crea un array:

```
scores := []int{1,4,293,4,9}
```

Al contrario que con la declaración de arrays, nuestro slice no ha sido declarado indicando una longitud en los corchetes. Vamos a ver otro modo de crear un slice, en este caso usando **`make`**, para entender por qué los dos son diferentes :

```
scores := make([]int, 10)
```

Usamos **`make`** en lugar de **`new`** porque la creación de un slice implica más cosas que simplemente reservar memoria (que es lo que hace **`new`**). En concreto, tenemos que reservar la memoria para un array subyacente y, además, tenemos que inicializar el slice. En el ejemplo anterior inicializábamos un slice con una longitud de 10 y una capacidad de 10. La

longitud es el tamaño del slice y la capacidad es el tamaño del array subyacente. Utilizando `make` podemos especificar ambos por separado:

```
scores := make([]int, 0, 10)
```

Esto crea un slice con longitud 0 y una capacidad de 10. (Si estás atento, quizá te habrás dado cuenta de que tanto `make` como `len` *están* sobrecargados. Go es un lenguaje que, para frustración de algunos, hace uso de características que los desarrolladores no tenemos capacidad de utilizar)

Vamos a ver algunos ejemplos con los que entender mejor la diferencia entre longitud y capacidad:

```
func main() {
  scores := make([]int, 0, 10)
  scores[5] = 9033
  fmt.Println(scores)
}
```

Nuestro primer ejemplo falla. ¿Por qué?, porque nuestro slice tiene una longitud de 0. Si, el array tiene una longitud de 10 elementos, pero necesitamos expandir nuestro slice explícitamente para poder acceder a esos elementos. Una forma de expandir un slice es a través de `append`:

```
func main() {
  scores := make([]int, 0, 10)
  scores = append(scores, 5)
  fmt.Println(scores) // muestra [5]
}
```

Pero esto cambia el propósito de nuestro código original. Hacer un append sobre un slice de tamaño 0 pondrá un valor en el primer elemento. Por la razón que sea, el código anterior quiere dar valor a la posición 5. Para conseguir esto podemos modificar nuestro slice:

```
func main() {
  scores := make([]int, 0, 10)
  scores = scores[0:6]
  scores[5] = 9033
  fmt.Println(scores)
}
```

¿Cuánto podemos redimensionar un slice? Hasta su capacidad, la cual en este caso es 10. Puede que estés pensando que *eso no resuelve el problema de la longitud fija de los arrays*. Esto convierte a `append` en algo especial. Si el array subyacente está lleno se creará uno más grande y se copiarán los valores sobre él (que es exactamente la forma de funcionar de los arrays dinámicos en lenguajes dinámicos como PHP, Python, Ruby, JavaScript, …). Este es el motivo por el cual, en el ejemplo anterior en el que usamos `append`, tenemos que reasignar el valor devuelto por `append` a nuestra variable `scores`: `append` puede haber creado un nuevo valor si el original se habia quedado sin espacio.

Si te dijera que Go hace crecer los arrays con un algoritmo que multiplica su tamaño por 2, ¿puedes imaginar cuál será la salida del siguiente programa?

```go
func main() {
  scores := make([]int, 0, 5)
  c := cap(scores)
  fmt.Println(c)

  for i := 0; i < 25; i++ {
    scores = append(scores, i)

    // Si la capacidad ha cambiado,
    // Go tiene que hacer crecer el array para volcar los datos
    if cap(scores) != c {
      c = cap(scores)
      fmt.Println(c)
    }
  }
}
```

La capacidad inicial de `scores` es 5. Para poder almacenar 25 valores tiene que expandirse 3 veces para tener la capacidad de 10, 20 y finalmente 40.

Como ejemplo final, ten en cuenta que:

```go
func main() {
  scores := make([]int, 5)
  scores = append(scores, 9332)
  fmt.Println(scores)
}
```

En este caso la salida será `[0, 0, 0, 0, 0, 9332]`. ¿Quizá pensaste que podría ser `[9332, 0, 0, 0, 0]`? Puede parecer lógico para un humano. Para un compilador, ese código indica que debe incluir un valor a un slice que tiene rellenos 5 valores.

Finalmente, hay cuatro formas comunes de inicializar un slice:

```go
names := []string{"leto", "jessica", "paul"}
checks := make([]bool, 10)
var names []string
scores := make([]int, 0, 20)
```

¿Cuándo usar cuál? La primera no debería tener mucha explicación: la deberías usar cuando sepas de antemano qué valores son los que debe haber en el array.

El segundo es útil para escribir en índices específicos de un slice. Por ejemplo:

```go
func extractPowers(saiyans []*Saiyans) []int {
  powers := make([]int, len(saiyans))
  for index, saiyan := range saiyans {
```

```
    powers[index] = saiyan.Power
  }
  return powers
}
```

El tercero sirve para crear un slice vacío y se utiliza junto con `append` cuando el número de elementos es desconocido.

La última versión nos permite indicar una capacidad inicial; es útil si tenemos una idea general de cuántos elementos vamos a necesitar.

Puedes utilizar `append` incluso cuando sabes el tamaño. Es, de largo, una mera cuestión de preferencia:

```
func extractPowers(saiyans []*Saiyans) []int {
  powers := make([]int, 0, len(saiyans))
  for _, saiyan := range saiyans {
    powers = append(powers, saiyan.Power)
  }
  return powers
}
```

Usar slices como wrappers de arrays es un concepto poderoso. Muchos lenguajes manejan el concepto de tener parte de un array. Los arrays tanto de JavaScript como de Ruby tienen un método `slice`. Puedes obtener un slice en Ruby usando `[INICIO..FIN]` o en Python a través de `[INICIO:FIN]`. Sin embargo, en estos lenguajes un slice no es más que un array nuevo con los valores del original copiados sobre él. Si usásemos Ruby, ¿cuál sería la salida del siguiente programa?

```
scores = [1,2,3,4,5]
slice = scores[2..4]
slice[0] = 999
puts scores
```

La respuesta es `[1, 2, 3, 4, 5]` debido a que `slice` es un array completamente nuevo con copias de los valores. Ahora, echa un vistazo al equivalente en Go:

```
scores := []int{1,2,3,4,5}
slice := scores[2:4]
slice[0] = 999
fmt.Println(scores)
```

La salida es `[1, 2, 999, 4, 5]`.

Esto cambia tu forma de pensar a la hora de programar. Por ejemplo, hay funciones que reciben la posición como un parámetro. En JavaScript, si quieres buscar el primer espacio en un string (sí, ¡los slices funcionan también con strings!) a partir de los primeros cinco caracteres, deberíamos escribir:

```
haystack = "the spice must flow";
console.log(haystack.indexOf(" ", 5));
```

En Go, usamos slices:

```
strings.Index(haystack[5:], " ")
```

Puedes ver en el ejemplo anterior que `[X:]` es un atajo que significa *desde X hasta el fin* mientras que `[:X]` es un atajo para *desde el comienzo hasta X*. Go, al contrario que otros lenguajes, no soporta valores negativos. Si queremos todos los valores de un slice excepto el último usamos:

```
scores := []int{1, 2, 3, 4, 5}
scores = scores[:len(scores)-1]
```

A continuación tienes una forma eficiente con la que borrar un valor de un slice sin ordenar:

```
func main() {
  scores := []int{1, 2, 3, 4, 5}
  scores = removeAtIndex(scores, 2)
  fmt.Println(scores)
}

func removeAtIndex(source []int, index int) []int {
  lastIndex := len(source) - 1
  //intercambiamos el último valor con aquel que queremos eliminar
  source[index], source[lastIndex] = source[lastIndex], source[index]
  return source[:lastIndex]
}
```

Para terminar, ahora que sabes slices, podemos echar un vistazo a otra función comúnmente usada: `copy`. `copy` es una de esas funciones que hacen destacar cómo los slices cambian la forma en la que programamos. Por lo general, un método que copie valores de un array a otro tiene 5 parámetros: `origen`, `comienzo en origen`, `contador`, `destino` y `comienzo en destino`. Gracias a los slices únicamente necesitamos dos:

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

func main() {
  scores := make([]int, 100)
  for i := 0; i < 100; i++ {
    scores[i] = int(rand.Int31n(1000))
  }
  sort.Ints(scores)

  worst := make([]int, 5)
  copy(worst, scores[:5])
```

```
  fmt.Println(worst)
}
```

Tómate algún tiempo y juega con este código. Pueba a hacer cambios. Comprueba qué ocurre cuando modificas la copia a algo como `copy(worst[2:4], scores[:5])`, o ¿qué ocurre si tratas de copiar más o menos de `5` valores en `worst`?

Mapas

Los mapas en Go son lo que en otros lenguajes se llaman tablas hash o diccionarios. Funcionan tal y como esperas: defines una clave y un valor y puedes recuperar, establecer y borrar valores en base a ella.

Los mapas se crean con la función **`make`** al igual que los slices. Vamos a ver un ejemplo:

```
func main() {
  lookup := make(map[string]int)
  lookup["goku"] = 9001
  power, exists := lookup["vegeta"]

  // muestra 0, false
  // 0 es el valor por defecto para un entero
  fmt.Println(power, exists)
}
```

Usamos **`len`** para recuperar el número de claves. Para borrar un valor en base a su clave usamos `delete`:

```
// devuelve 1
total := len(lookup)

// no devuelve nada, puede ser invocado en una clave que no exista
delete(lookup, "goku")
```

Los mapas crecen dinámicamente. Sin embargo, podemos pasarle un segundo argumento a **`make`** con el que indicar un tamaño inicial:

```
lookup := make(map[string]int, 100)
```

Definir un tamaño inicial puede ayudar a mejorar el rendimiento, así que si tienes una idea sobre cuántas claves vas a tener, es bueno indicarlo al crear el mapa.

Puedes definir del modo siguiente un mapa como campo en una estructura:

```
type Saiyan struct {
  Name string
  Friends map[string]*Saiyan
}
```

Una forma de inicializar el mapa anterior es la siguiente:

```
goku := &Saiyan{
  Name: "Goku",
  Friends: make(map[string]*Saiyan),
}
goku.Friends["krillin"] = ... //TODO: cargar o crear a Krillin
```

Hay otra forma más de declarar e inicializar valores en Go:

```
lookup := map[string]int{
  "goku": 9001,
  "gohan": 2044,
}
```

Podemos iterar sobre un mapa usando un bucle `for` en combinación con la palabra clave `range`:

```
for key, value := range lookup {
  ...
}
```

La iteración en mapas no está ordenada, de tal modo que cada iteración devolverá las parejas clave-valor en un orden aleatorio.

Punteros versus Valores

Acabamos el capítulo 2 revisando cuándo deberíamos usar valores o punteros como parámetro. Ha llegado el momento de que tengamos la misma conversación con respecto a los arrays y los mapas. ¿Cuál de estos dos deberías utilizar?

```
a := make([]Saiyan, 10)
//o
b := make([]*Saiyan, 10)
```

Muchos desarrolladores creen que pasar `b` a, o devolver `b` desde una función va a ser más eficiente. Sin embargo, lo que está siendo pasado/devuelto es una copia del slice, que ya es de por sí una referencia. Así que no hay diferencia con respecto a pasar/devolver un slice.

Verás la diferencia cuando modifiques los valores del slice o del mapa. En ese momento aplica la lógica que vimos en el capítulo 2. Así que la decisión sobre si definir un array de punteros o un array de valores depende de cómo utilices los valores individales, no de cómo utilices el array o el mapa.

Antes de Continuar

Los arrays y los mapas en Go son muy parecidos a los de otros lenguajes. Si estás acostumbrado a los arrays dinámicos puede que algo no te termine de encajar, pero la función `append` te debería ayudar. Más allá de la sintaxis superficial de los arrays están los slices, los cuales son muy prácticos y tienen un gran impacto en la claridad del código.

Hay casos límite que no hemos cubierto, pero no vamos a entrar en ellos. Si lo haces por tu cuenta, espero que el poso que te haya quedado con este capítulo te ayude a entenderlos mejor.

Capítulo 4 - Organización del Código e Interfaces

Ha llegado el momento de ver cómo organizar el código.

Paquetes

Para poder utilizar complicadas librerías y tener organizado el código fuente de nuestra aplicación es necesario que aprendamos qué son los paquetes. Los nombres de los paquetes en Go siguen la estructura de directorios de tu workspace. Si estamos desarrollando una aplicación de compras, seguramente empecemos con un nombre de paquete llamado "shopping" y pongamos nuestro código fuente en `$GOPATH/src/shopping/`.

De todos modos no querremos poner todo dentro de ese directorio. Por ejemplo, quizá queramos aislar la lógica de la base de datos dentro de un directorio propio. Para conseguir esto, crearemos un subdirectorio en `$GOPATH/src/shopping/db`. El nombre del paquete dentro de este subdirectorio será simplemente `db`, y para acceder a él desde otro paquete, incluido el propio paquete `shopping`, debemos importar `shopping/db`.

En otras palabras: cuando nombras un paquete mediante la palabra clave **`package`**, estás dando únicamente un valor, no una jerarquía completa (por ejemplo, "shopping" o "db"). Es necesario especificar la ruta completa a la hora de importar un paquete.

Vamos a intentarlo. Dentro del directorio `src` de tu workspace (el cual configuramos en el apartado Comenzando de la Introducción) crea un nuevo directorio llamado `shopping` y un subdirectorio dentro llamado `db`.

Crea un fichero llamado `db.go` dentro de `shopping/db` con el siguiente contenido:

```
package db

type Item struct {
  Price float64
}

func LoadItem(id int) *Item {
  return &Item{
    Price: 9.001,
  }
}
```

Observa que el nombre del paquete es el mismo que el del directorio. Aparte, obviamente, no estamos accediendo a la base de datos, esto no es más que un ejemplo sobre cómo organizar el código.

Ahora, crea un fichero llamado `pricecheck.go` dentro del directorio `shopping`. Su contenido es:

```
package shopping

import (
  "shopping/db"
```

```
)

func PriceCheck(itemId int) (float64, bool) {
  item := db.LoadItem(itemId)
  if item == nil {
    return 0, false
  }
  return item.Price, true
}
```

Es tentador pensar que importar `shopping/db` es algo especial ya que nos encontramos dentro del paquete/directorio `shopping`. En realidad, estamos importando `$GOPATH/src/shopping/db`.

Si estás construyendo un paquete no necesitas saber más que lo que ya has visto. Para generar un ejecutable sigues necesitando un **`main`**. Mi forma favorita de hacerlo es creando un subdirectorio llamado **`main`** dentro de `shopping` con un fichero llamado **`main.go`** y el siguiente contenido:

```
package main

import (
  "shopping"
  "fmt"
)

func main() {
  fmt.Println(shopping.PriceCheck(4343))
}
```

Puedes ejecutar tu código accediendo a tu proyecto `shopping` y escribiendo:

```
go run main/main.go
```

Importaciones Cíclicas

Te irás encontrando con ellas a medida que vayas desarrollando software más complejo. Ocurren cuando el paquete A importa el paquete B a la vez que el paquete B importa el paquete A (ya sea directa o indirectamente a través de otro paquete). Esto es algo que el compilador no permite.

Vamos a cambiar nuestro ejemplo para provocar el error.

Mueve la definición de `Item` de `shopping/db/db.go` a `shopping/pricecheck.go`. Tu fichero `pricecheck.go` debe quedar así:

```
package shopping

import (
```

```
  "shopping/db"
)

type Item struct {
  Price float64
}

func PriceCheck(itemId int) (float64, bool) {
  item := db.LoadItem(itemId)
  if item == nil {
    return 0, false
  }
  return item.Price, true
}
```

Si tratas de ejecutar el código te encontrarás con un par de errores de `db/db.go` indicando que `Item` no ha sido definido. Esto tiene sentido, ya que `Item` ha dejado de existir en el paquete `db`; ha sido movido al paquete `shopping`. Necesitamos modificar `shopping/db/db.go` a:

```
package db

import (
  "shopping"
)

func LoadItem(id int) *shopping.Item {
  return &shopping.Item{
    Price: 9.001,
  }
}
```

Ahora, al tratar de ejecutar el código, obtenemos un intimidante error *importación cíclica no permitida*. Lo solucionaremos introduciendo otro paquete que contenga las estructuras compartidas. Tu estructura de directorios debería tener el siguiente aspecto:

```
$GOPATH/src
  - shopping
    pricecheck.go
    - db
      db.go
    - models
      item.go
    - main
      main.go
```

`pricecheck.go` seguirá importando `shopping/db`, pero `db.go` importará `shopping/models` en lugar de `shopping` para así romper el ciclo. ya que hemos movido la estructura compartida `Item` a `shopping/models/item.go`, necesitamos cambiar `shopping/db/db.go` para referenciar la estructura `Item` desde el paquete `models`:

```
package db

import (
  "shopping/models"
)

func LoadItem(id int) *models.Item {
  return &models.Item{
    Price: 9.001,
  }
}
```

A menudo necesitarás compartir algo más que sólamente `models`, así que quizá tengas otro directorio similar llamado `utilities`. La regla a tener en cuenta con respecto a los paquetes compartidos es que no deberían importar nada procedente del paquete `shopping` o de ninguno de sus subpaquetes. Dentro de un par de secciones veremos las interfaces, las cuales nos ayudarán a desenredar estos tipos de dependencias.

Visibilidad

Go utiliza una regla muy sencilla para especificar qué tipos y qué funciones son visibles fuera del paquete. Si el nombre del tipo o de la función comienza con una letra mayúscula es visible. Si comienza con una letra minúscula no lo es.

Esto también aplica a los campos de las estructuras. Si el nombre de un campo de una estructura comienza con una letra minúscula sólo el código que se encuentre dentro del mismo paquete será capaz de acceder a ella.

Por ejemplo, si nuestro fichero `items.go` tiene una función con este aspecto:

```
func NewItem() *Item {
  // ...
}
```

podrá ser invocada mediante `models.NewItem()`. Pero si la función hubiese sido llamada `newItem`, no seríamos capaces de acceder a ella desde un paquete diferente.

Prueba a cambiar el nombre de varias funciones, tipos y campos del ejemplo de las compras. Por ejemplo, si renombras el campo `Price` de `Item` a `price` deberías obtener un error.

Gestión de Paquetes

El comando `go` que hemos estado utilizando con `run` y `build` también puede ser utilizado con `get`, que sirve para conseguir dependencias de terceros. `go get` funciona con varios protocolos aunque para este ejemplo, en el cual obtendremos una librería de Github, necesitarás tener `git` instalado en tu ordenador.

Asumiendo que ya tienes git instalado, escribe lo siguiente desde un terminal:

```
go get github.com/mattn/go-sqlite3
```

`go get` descarga los archivos y los almacena en tu workspace. Puedes comprobarlo echando un vistazo al directorio `$GOPATH/src`, verás que junto al proyecto que hemos creado antes encontrarás un directorio llamado `github.com`. Dentro de él verás un directorio `mattn` que contiene un directorio `go-sqlite3`.

Acabamos de comentar cómo importar paquetes existentes en nuestro workspace. Para utilizar el paquete `go-sqlite3` recién obtenido realizaremos el siguiente import:

```
import (
  "github.com/mattn/go-sqlite3"
)
```

Sé que parece una URL pero, en realidad, únicamente importa el paquete `go-sqlite3`, el cual se espera en la ruta `$GOPATH/src/github.com/mattn/go-sqlite3`.

Gestión de Dependencias

`go get` guarda un par de ases bajo la manga. Si usamos `go get` en un proyecto se escanearán todos los ficheros buscando `imports` con librerías de terceros y las descargará. En cierta medida, nuestro propio código fuente se convierte en un `Gemfile`, un `package.json`, un `pom.xml` o un `build.gradle`.

Si utilizas `go get -u` actualizarás todos los paquetes (también puedes actualizar un paquete específico usando `go get -u NOMBRE_COMPLETO_DEL_PAQUETE`).

Es probable que, con el tiempo, encuentres `go get` algo insuficiente por una razón: no hay forma de especificar una revisión ya que siempre utiliza master/head/trunk/default. Esto puede ser un problema incluso mayor si tienes dos o más proyectos que necesitan versiones diferentes de la misma librería.

Para solucionar este problema puedes usar una herramienta de gestión de dependencias de terceros. Todavía son jóvenes pero las dos más prometedoras son goop y godep. Hay una lista más completa en la go-wiki.

Interfaces

Los interfaces son tipos que especifican un contrato pero no contienen implementación. He aquí un ejemplo:

```
type Logger interface {
  Log(message string)
}
```

Puede que te estés preguntando para qué puede servir esto. Los interfaces sirven para desacoplar tu código de implementaciones específicas. Por ejemplo, podríamos tener distintos tipos de loggers:

```
type SqlLogger struct { ... }
type ConsoleLogger struct { ... }
type FileLogger struct { ... }
```

Programar contra interfaces en lugar de contra implementaciones concretas nos permite poder cambiar (y probar) cuál usar sin tener que modificar nuestro código.

¿Cómo se utilizan? Exactamente igual que cualquier otro tipo. Puede ser el campo de una estructura:

```
type Server struct {
  logger Logger
}
```

o el parámetro de una función (o un valor de retorno):

```
func process(logger Logger) {
  logger.Log("hello!")
}
```

En lenguajes como C# o Java es necesario indicar explícitamente cuándo una clase implementa una interfaz:

```
public class ConsoleLogger : Logger {
  public void Logger(message string) {
    Console.WriteLine(message)
  }
}
```

En Go esto ocurre de forma implícita. Si tu estructura tiene una función llamada `Log` con un parámetro de tipo `string` y ningún valor de retorno, entonces puede ser utilizada como un `Logger`, lo que evita el tener que escribir tanto a la hora de usar interfaces:

```
type ConsoleLogger struct {}
func (l ConsoleLogger) Log(message string) {
  fmt.Println(message)
}
```

Esto facilita tener interfaces pequeños y muy enfocados en una tarea concreta. La librería estándar está llena de interfaces. Por ejemplo, el paquete `io` tiene varios de ellos, como `io.Reader`, `io.Writer` e `io.Closer`. Si escribes una función que espera un parámetro sobre el que invocar a `Close()` al terminar, entoces definitivamente deberías recibir `io.Closer` en lugar del tipo concreto que estés usando.

Los interfaces también pueden participar en composiciones, de hecho, los interfaces pueden estar compuestos por otros interfaces. Por ejemplo, `io.ReadCloser` es un interfaz formado por los interfaces `io.Reader` e `io.Closer`.

Para acabar, los interfaces son ampliamente utilizados para evitar imports cíclicos. No poseen implementacion, lo que reduce las dependencias que puedan necesitar.

Antes de Continuar

Pasarás a encontrarte cómodo con la forma en la que estructurar tu código tras haber escrito un par de programas que no sean triviales. Lo más importante es recordar la estrecha relación que mantienen los nombres de los paquetes y la jerarquía de directorios (no sólo en el proyecto sino en todo el workspace).

La forma en la que Go gestiona la visibilidad de los tipos es directa a la par que efectiva, y además es consistente. Hay pocas cosas que no hayamos visto todavía, como las constantes y las variables globales, pero puedes descansar tranquilo, su visibilidad sigue las mismas reglas de nombrado.

Para terminar, puede que te lleve algún tiempo encontrar la utilidad de los interfaces si son nuevos para ti. Sin embargo, la primera vez que te encuentres con una función que espera algo como `io.Reader` agradecerás al autor que no haya pedido más de lo que realmente necesitaba.

Capítulo 5 - Exquisiteces

En este capítulo hablaremos de algunas características de Go que no encajan en ningún otro lugar.

Gestión de Errores

La mejor forma de gestionar errores en Go es utilizando valores de retorno, no empleando excepciones. Utiliza como ejemplo la función `strconv.Atoi`, la cual recibe un string y trata de convertirlo a entero;

```
package main

import (
  "fmt"
  "os"
  "strconv"
)

func main() {
  if len(os.Args) != 2 {
    os.Exit(1)
  }

  n, err := strconv.Atoi(os.Args[1])
  if err != nil {
    fmt.Println("no es un número válido")
  } else {
    fmt.Println(n)
  }
}
```

Puedes crear tus propios tipos de error, el único requisito es que cumplan con el contrato de la interfaz predefinida `error`, el cual es:

```
type error interface {
  Error() string
}
```

Con mayor asiduidad crearemos nuestros propios errores importando el paquete `errors` y usando la función **`New`**:

```
import (
  "errors"
)


func process(count int) error {
```

```
  if count < 1 {
    return errors.New("Invalid count")
  }
  ...
  return nil
}
```

Existe un patrón en la librería estándar de Go a la hora de utilizar variables de error. Por ejemplo, el paquete `io` declara la variable `EOF` tal y como sigue:

```
var EOF = errors.New("EOF")
```

Estamos ante una variable del paquete (está definida fuera de cualquier función) la cual es públicamente accesible (la primera letra está en mayúsculas). Hay varias funciones que pueden devolver este error, por ejemplo aquellas que leen ficheros o la entrada de STDIN. En este sentido tú también deberías utilizar este error. Como consumidores podemos utilizar este singleton:

```
package main

import (
  "fmt"
  "io"
)

func main() {
  var input int
  _, err := fmt.Scan(&input)
  if err == io.EOF {
    fmt.Println("no more input!")
  }
}
```

Como nota final, en Go existen las funciones `panic` y `recover`. `panic` es similar a lanzar una excepción y `recover` es parecido a un `catch`. Ambas son raramente utilizadas.

Defers

Hay determinados recursos que necesitan ser liberados explícitamente, incluso aunque Go tenga un recolector de basura. Por ejemplo, necesitamos cerrar ficheros con `Close()` después de haber terminado con ellos. Este tipo de código siempre es peligroso. Por un lado, mientras estamos escribiendo una función, es fácil olvidarse de usar `Close()` con algo que declaramos 10 líneas más arriba. Por otro lado, una función puede tener más de un punto de retorno. La solución de Go es la palabra clave `defer`:

```
package main
```

```
import (
  "fmt"
  "os"
)

func main() {
  file, err := os.Open("un_fichero_que_leer")
  if err != nil {
    fmt.Println(err)
    return
  }
  defer file.Close()
  // leer el fichero
}
```

Si tratas de ejecutar el código anterior seguramente obtengas un error (el fichero no existirá). La clave es mostrar cómo funciona `defer`. Lo que pongas después de `defer` será aplazado y se ejecutará al terminar el método, incluso si lo hace forzosamente. Esto te permite indicar que quieres liberar recursos cerca del lugar donde los inicializaste y despreocuparte de los múltiples puntos de retorno.

go fmt

La mayoría de los programas escritos en Go siguen las mismas reglas de formato y de nombrado, entre ellas que se deben usar tabuladores para indentar y que las llaves deben comenzar en la misma línea en la que empieza el bloque.

Lo sé, tienes tu propio estilo y quieres seguir usándolo. Esto mismo es lo que yo he hecho durante mucho tiempo, pero estoy contento de poder decir que lo he dejado de hacer. Un gran motivo es el comando `go fmt`. Es simple de usar y tiene autoridad, lo cual evita discusiones sobre preferencias sin importancia.

Cuando te encuentras en un proyecto puedes aplicar las reglas de formateo sobre él y sus subproyectos ejecutando:

```
go fmt ./...
```

Dale una oportunidad. Hará algo más que indentar tu código: alineará las declaraciones de los campos y ordenará alfabéticamente los imports.

If Inicializado

Go soporta un comando if ligeramente modificado, en el cual un valor puede ser inicializado antes que la condición que va a ser evaluada:

```
if x := 10; count > x {
  ...
}
```

Lo anterior es un ejemplo únicamente ilustrativo. Para ser más realistas, podrías hacer algo como:

```
if err := process(); err != nil {
  return err
}
```

Lo más interesante es que el ámbito de las variables se reduce al if, y que por tanto se puede acceder a ellas dentro de un **else if** o de un **else**, pero no fuera.

Interfaces Vacías y Conversiones

En la mayoría de los lenguajes orientados a objetos hay una clase base predefinida, a menudo llamada `object`, que actúa como superclase de todas las clases. Esto es lo que ocurre con una interfaz vacía sin métodos: **interface**{}. Puesto que las interfaces se implementan implícitamente, todos los tipos encajan con el contrato de una interfaz vacía.

Si quisiéramos podríamos crear una función llamada `add` con la siguiente firma:

```
func add(a interface{}, b interface{}) interface{} {
  ...
}
```

Para convertir una variable a un tipo específico debes usar `.(TIPO)`:

```
return a.(int) + b.(int)
```

También puedes crear switchs de tipos:

```
switch a.(type) {
  case int:
    fmt.Printf("a es un entero y vale %d\n", a)
  case bool, string:
    // ...
  default:
    // ...
}
```

Verás y utilizarás interfaces vacías más de lo que podrías creer a primera vista. Admitámoslo, no sirve para escribir código limpio. Convertir tipos de uno a otro constantemente es feo y hasta peligroso pero a veces, en lenguajes estáticos, es la única opción.

Strings y Arrays de Bytes

Los strings y los arrays de bytes están muy relacionados. Podemos convertir de uno a otro fácilmente:

```
stra := "the spice must flow"
byts := []byte(stra)
strb := string(byts)
```

De hecho, este tipo de conversión es común también entre otros tipos. Algunas funciones esperan explícitamente un `int32` o un `int64` o sus homólogos sin signo. Puede que te descubras a ti mismo haciendo cosas como esta:

```
int64(count)
```

Es algo que probablemente acabes haciendo con frecuencia cuando utilices bytes y strings. Observa que cuando utilizas `[]byte(X)` o `string(X)` estás haciendo una copia de los datos. Esto es imprescindible ya que los strings son inmutables.

Los strings están hechos de `runas`, que son letras unicode. Si recuperas la longitud de un string es probable que no obtengas el valor que esperas. El código siguiente muestra 3:

```
fmt.Println(len("□")) // quizá no lo veas, aquí debería aparecer un carácter chino
```

Si iteras sobre un string utilizando **`range`** obtendrás las runas, no los bytes, aunque cuando conviertes un string en un `[]byte` obtienes los datos correctos.

Funciones con Tipo

Las funciones son ciudadanos de primer nivel y tienen tipo:

```
type Add func(a int, b int) int
```

los cuales pueden ser utilizados en cualquier sitio – como tipo de un campo, como parámetro o como valor de retorno.

```
package main

import (
  "fmt"
)

type Add func(a int, b int) int

func main() {
  fmt.Println(process(func(a int, b int) int{
      return a + b
  }))
}

func process(adder Add) int {
  return adder(1, 2)
}
```

El empleo de funciones como ésta puede ayudar a desacoplar código de implementaciones específicas igual que como lo hacíamos con los interfaces.

Antes de Continuar

Hemos revisado algunos aspectos de la programación con Go. Lo más destacable es que hemos visto cómo funciona la gestión de errores y cómo liberar recursos como conexiones y ficheros abiertos. A mucha gente no le gusta cómo Go gestiona los errores ya que puede parecer que estemos dando un paso atrás. A veces estoy de acuerdo, aunque también encuentro que produce código que es más fácil de seguir. El uso de `defer` es infrecuente pero es una aproximación práctica a la gestión de recursos. De hecho, no está limitado sólo a la gestión de recursos, puedes usar `defer` con otro propósito, como por ejemplo dejar en el log la salida de una función cuando esta termine.

Ciertamente no hemos revisado todas las exquisiteces que Go ofrece, pero deberías sentirte lo suficientemente cómodo como para derribar aquellos muros que te puedan ir apareciendo.

Capítulo 6 - Concurrencia

A menudo se describe a Go como un lenguaje con el que es fácil utilizar concurrencia. Esto es así porque utiliza una sintaxis sencilla sobre dos poderosos mecanismos: las go-rutinas y los canales

Go-rutinas

Una go-rutina es similar a un hilo sólo que es gestionada por Go, no por el sistema operativo. El código de una go-rutina puede ejecutase concurrentemente con otro código. Vamos a echar un vistazo a este ejemplo:

```
package main

import (
  "fmt"
  "time"
)

func main() {
  fmt.Println("inicio")
  go process()
  time.Sleep(time.Millisecond * 10) // esto es malo, ¡no lo hagas!
  fmt.Println("terminado")
}

func process() {
  fmt.Println("procesando")
}
```

Hay algunas cosas interesantes aquí, aunque la más importante es cómo iniciamos una go-rutina. Simplemente utilizamos la palabra clave `go` seguida de la función que queremos ejecutar. Si sólo queremos ejecutar un poco de código, como en el caso anterior, podemos usar una función anónima.

```
go func() {
  fmt.Println("procesando")
}()
```

Las go-rutinas son fáciles de crear y tienen muy poca sobrecarga. De hecho, se pueden ejecutar muchas go-rutinas en un mismo hilo. Como resultado tenemos que una go-rutina tiene una sobrecarga de unos pocos KB. De este modo podemos tener millones de go-rutinas funcionando sobre hardware moderno.

Es más, la complejidad de asignar go-rutinas a hilos y cómo planificarlas está escondida. Podemos decir "ejecuta este código concurrentemente" y dejar que sea Go quien se preocupe que hacer que eso ocurra.

Si volvemos al ejemplo anterior veremos que invocamos a `Sleep` unos pocos milisegundos. Esto lo hacemos para evitar que la función main acabe antes de que la go-rutina tenga la posibilidad de ejecutarse (el proceso principal no

espera a que las go-rutinas acaben antes de terminar). Para solucionar esto necesitamos coordinar nuestro código.

Sincronización

La creación de go-rutinas es trivial y son tan baratas que podemos arrancar muchas. Sin embargo, es necesario coordinar el código concurrente. Para ayudar a solucionar este problema Go provee `canales`. Pero creo que es importante entender algunos conceptos básicos antes de echar un vistazo a los `canales`.

Escribir código concurrente implica tener que prestar atención a dónde y cómo se leen y escriben valores. En cierta manera es como programar sin recolector de basura – requiere que pienses en los datos desde una perspectiva distinta, estando siempre alerta de daños posibles. Observa el siguiente ejemplo:

```
package main

import (
  "fmt"
  "time"
)

var counter = 0

func main() {
  for i := 0; i < 2; i++ {
    go incr()
  }
  time.Sleep(time.Millisecond * 10)
}

func incr() {
  counter++
  fmt.Println(counter)
}
```

¿Cuál crees que será la salida?

Si crees que la salida será `1,2`, te equivocas y tienes razón a la vez. Es cierto que si ejecutas el código anterior probablemente obtendrás esa salida. Sin embargo, la realidad es que el comportamiento no está definido. ¿Por qué?, porque varias (en este caso dos) go-rutinas están potencialmente escribiendo sobre la misma variable `counter` a la misma vez. O, lo que es peor, una go-rutina puede estar leyendo `counter` mientras la otra actualiza su valor.

¿Es esto relamente peligroso?. Sí, desde luego. `counter++` puede parecer únicamente una línea de código, pero en realidad está compuesta de varias instrucciones en ensamblador – la naturaleza exacta depende de la plataforma sobre la que estés trabajando. Es cierto que, en este ejemplo, lo más normal es que todo vaya bien. Sin embargo, otro posible resultado podría ser que ambas go-rutinas leyesen `counter` cuando su valor es cero y se generase una

salida de `1, 1`. Hay otras posibilidades peores, como que el ordenador se cuelgue o que accedamos a otra posición de memoria con datos y los incrementemos.

La única acción concurrente que puedes hacer con seguridad sobre una variable es leerla. Puedes tener todos los lectores que quieras, pero las escrituras necesitan estar sincronizadas. Hay varias formas de lograr esto, incluyendo el uso de operaciones realmente atómicas que necesitan instrucciones específicas de la CPU. Sin embargo, la aproximación más frecuente es usar un mutex:

```
package main

import (
  "fmt"
  "time"
  "sync"
)

var (
  counter = 0
  lock sync.Mutex
)

func main() {
  for i := 0; i < 2; i++ {
    go incr()
  }
  time.Sleep(time.Millisecond * 10)
}

func incr() {
  lock.Lock()
  defer lock.Unlock()
  counter++
  fmt.Println(counter)
}
```

Un mutex organiza el acceso a un código protegido. El motivo por el cual sencillamente definimos nuestro bloqueo como `lock sync.Mutex` se debe a que el valor por defecto de `sync.Mutex` es el de "desbloqueado"

¿Parece bastante sencillo? El ejemplo anterior es engañoso. Hay un montón de bugs serios que pueden aparecer a la hora de hacer programación concurrente. El primero de ellos es que no siempre resulta obvio saber qué parte del código necesita ser protegida. Puede ser tentador proteger una gran cantidad de código, pero menoscaba la razón principal por la que hacemos programación concurrente. Necesitamos protecciones lo más pequeñas posible: de otro modo convertiremos una autopista de 10 carriles en una que, de repente, se convierte en una de un solo carril.

El otro problema tiene que ver con los deadlocks. Con un único bloqueo no hay problema, pero puede ser peligroso si tenemos dos o más en el mismo código, ya que se puede dar el caso de que la go-rutina A posea el bloqueo A pero

necesite acceder al bloqueo B, mientras que la go-rutina B posee el bloqueo B pero necesita acceder al bloqueo A.

De hecho *es* posible tener un deadlock con un único bloqueo si nos olvidamos de liberarlo. No es tan peligroso como tener varios bloqueos pero puede ocurrir. Prueba a ejecutar el siguiente programa:

```
package main

import (
  "time"
  "sync"
)

var (
  lock sync.Mutex
)

func main() {
  go func() { lock.Lock() }()
  time.Sleep(time.Millisecond * 10)
  lock.Lock()
}
```

Hay mucho más sobre programación concurrente que lo que hemos visto hasta ahora. Existe, ya que podemos hacer varias lecturas a la vez, otro mutex conocido como mutex de lectura-escritura, el cual presenta dos funciones de bloqueo: por un lado se pueden bloquear lectores y, por otro, escritores. En Go, `sync.RWMutex` es un bloqueador de este estilo que, junto a los métodos `Lock` y `Unlock` de `sync.Mutex`, también incluye los métodos `RLock` y `RUnlock`, donde `R` significa `Lectura`. Aunque los mutex de lectura-escritura se utilizan comúnmente, suponen una carga extra para los desarrolladores: debemos prestar atención no sólo a quiénes están accediendo a los datos sino también a cómo lo hacen.

Gran parte de la programación concurrente tiene que ver con la coordinación de varias go-rutinas. Por ejemplo, dejar al proceso durmiendo 10 milisegundos no es una solución particularmente elegante, ¿qué ocurre si la go-rutina necesita más de 10 milisegundos?, ¿qué pasa si tarda menos y simplemente estamos desperdiciando ciclos de reloj?, Es más, ¿qué ocurre si, en vez de esperar a que las go-rutinas acaben, simplemente hacemos que digan a las demás *hey, itengo datos nuevos para que te encargues de procesarlos!*?

Todo esto puede llevarse a cabo con `canales`. Es cierto que, para los casos más sencillos, considero que **deberías** usar primitivas como `sync.Mutex` y `sync.RWMutex`, pero como verás en la próxima sección, los `canales` te permiten hacer una programación concurrente más limpia y con menor propensión a errores.

Canales

El reto de la programación concurrente aparece a la hora de tener que compartir datos. Si tus go-rutinas no comparten datos entonces no necesitas preocuparte de sincronzarlas aunque esto, por supuesto, no es una solución que puedas

utilizar en todas las aplicaciones. De hecho, muchos sistemas se construyen precisamente con la meta opuesta en mente: la de compartir datos. Una caché en memoria o una base de datos son buenos ejemplos.

Los canales ayudan a que la programación concurrente sea más sencilla haciendo desaparecer los datos compartidos. Un canal es un medio de comunicación entre go-rutinas que se utiliza para enviar datos. En otras palabras, una go-rutina que quiera enviar datos a otra debe hacerlo mediante un canal. El resultado es que, en un momento puntual, sólo una go-rutina tiene acceso a los datos.

Los canales, como todo lo demás, tienen tipos. En concreto, tienen el mismo tipo que los datos que se van a mandar a través de ellos. Por ejemplo, para crear un canal que envíe enteros debemos usar:

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

El tipo del canal es `chan int`. Por lo tanto, para mandar el canal a una función como parámetro la firma debe ser:

```
func worker(c chan int) { ... }
```

Los canales permiten dos operaciones: enviar y recibir. Enviamos datos a un canal a través de:

```
CANAL <- DATOS
```

Y recibimos con:

```
VARIABLE := <-CANAL
```

La flecha indica la dirección en la que se mueven los datos. Los datos entran en el canal al enviar y salen del canal al recibir.

Lo último que debes saber antes de que veamos nuestro primer ejemplo es que enviar y recibir a y desde un canal es bloqueante. Esto es, cuando recibimos información de un canal, la go-rutina no continuará hasta que los datos estén disponibles. Del mismo modo, cuando enviamos datos mediante un canal, la ejecución no continuará hasta que los datos hayan sido recibidos.

Considera un sistema en el que manejemos los datos de entrada en go-rutinas separadas, lo que es un requisito frecuente. Si hacemos que el procesamiento intensivo se realice en la go-rutina que acepta los datos de entrada podemos correr el riesgo de dar una respuesta lenta a los clientes. Primero, vamos a escribir un worker, que bien podría ser una mera función, pero voy a hacer que sea parte de una estructura ya que antes no hemos usado go-rutinas de este modo:

```
type Worker struct {
  id int
}

func (w Worker) process(c chan int) {
  for {
    data := <-c
    fmt.Printf("worker %d got %d\n", w.id, data)
  }
}
```

Nuestro worker es sencillo. Espera hasta que los datos estén disponibles y después los procesa con la función "process". Para ello utiliza un bucle, en el cual está siempre esperando por más datos que procesar.

Para usarlo, lo primero que necesitamos es iniciar algunos workers:

```
c := make(chan int)
for i := 0; i < 4; i++ {
  worker := Worker{id: i}
  go worker.process(c)
}
```

Y después les damos algo de trabajo:

```
for {
  c <- rand.Int()
  time.Sleep(time.Millisecond * 50)
}
```

Aquí tienes el código completo para hacerlo funcionar:

```
package main

import (
  "fmt"
  "time"
  "math/rand"
)

func main() {
  c := make(chan int)
  for i := 0; i < 5; i++ {
    worker := &Worker{id: i}
    go worker.process(c)
  }

  for {
    c <- rand.Int()
    time.Sleep(time.Millisecond * 50)
  }
}

type Worker struct {
  id int
}

func (w *Worker) process(c chan int) {
```

```
    for {
      data := <-c
      fmt.Printf("worker %d got %d\n", w.id, data)
    }
  }
```

No sabemos qué worker va a recuperar los datos. Lo que sí sabemos, ya que es lo que Go garantiza, es que los datos enviados a un canal sólo se recibirán en un receptor.

Observa que lo único que se encuentra compartido es el propio canal, el cual puede enviar y recibir datos concurrentemente. Los canales incluyen toda la sincronización que necesitamos y nos aseguran que, en cualquier instante de tiempo, una y sólo una go-rutina tiene acceso a unos datos concretos.

Canales con Buffer

Partiendo del código anterior, ¿qué ocurre si recibimos más datos de los que podemos manejar? Puedes simular este comportamiento haciendo que el worker se duerma tras recibir datos:

```
for {
  data := <-c
  fmt.Printf("worker %d got %d\n", w.id, data)
  time.Sleep(time.Millisecond * 500)
}
```

El código del main, el lugar donde se generan los números aleatorios, se queda bloqueado ya que envía datos al canal pero no hay receptores disponibles.

En los casos en los cuales necesites garantizar que los datos están siendo procesados seguramente necesites bloquear al cliente. En otros casos probablemente te convenga relajar estas garantías. Hay unas pocas estrategias populares para hacer esto. La primera es almacenar los datos en un buffer: si no hay workers disponibles, almacenamos temporalmente los datos en una especie de cola. Los canales tienen la capacidad de poder almacenar datos en un buffer, y de hecho podemos darle una longitud cuando creamos nuestro canal con **make**.

```
c := make(chan int, 100)
```

Puedes hacer este cambio, pero observa que el procesamiento seguirá causando estragos. Los canales con buffers no añadirán más capacidad: únicamente facilitan una cola en la que almacenar trabajos pendientes que ayuden a lidiar con picos de trabajo. En nuestro ejemplo estamos empujando continuamente más datos de los que nuestros workers pueden gestionar.

Sin embargo, puede tener sentido comprobar que nuestro canal con buffer está, de hecho, almacenando datos en el buffer utilizando **len**:

```
for {
  c <- rand.Int()
  fmt.Println(len(c))
  time.Sleep(time.Millisecond * 50)
```

```
}
```

Puedes obsevar cómo crece y crece hasta que se satura, momento en el que enviar datos al canal hará que se bloquee de nuevo.

Select

Incluso con buffers, llega el momento en el cual necesitamos comenzar a descartar mensajes. No podemos usar una cantidad infinita de memoria con la esperanza de que un worker la libere. Por este motivo podemos usar `select`.

Sintácticamente, un `select` se parece un poco a un switch. Con él podremos indicar qué hacer cuando el canal no esté dispinible para enviar más datos a través de él. Vamos a eliminar el buffer del canal para ver con más claridad cómo funciona `select`:

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

A continuación modificamos nuestro bucle `for`:

```
for {
  select {
  case c <- rand.Int():
    //aquí pondríamos el código
  default:
    //podemos dejarlo vacío para descartar los datos
    fmt.Println("dropped")
  }
  time.Sleep(time.Millisecond * 50)
}
```

Estamos empujando 20 mensajes por segundo, pero nuestros workers sólo pueden procesar 10 por segundo, así que la mitad de los mensajes se descartan.

Este es sólo el comienzo de lo que podemos lograr con `select`. Uno de sus propósitos principales es gestionar varios canales, haciendo que `select` bloquee algunos mientas habilita otros. En caso de no haber canales disponibles se ejecutará el caso `default`. Un canal es escogido aleatoriamente cuando varios están disponibles.

Es difícil terminar con un ejemplo que demuestre que este comportamiento no es una característica avanzada. No obstante, la próxima sección nos ayudará a ilustrar este caso.

Timeout

Hemos visto cómo los mensajes que se encuentran en buffers pueden ser descartados, aunque otra opción popular es especificar un timeout. Podemos bloquear por un tiempo, pero no para siempre. Esto es algo fácil de lograr con Go. Debo admitir que la sintaxis puede ser difícil de leer pero es una funcionalidad tan limpia y útil que no puedo dejarla pasar.

Podemos usar la función `time.After` para indicar un máximo de tiempo de bloqueo. Vamos a ver qué hay detrás de la magia. Para ello, nuestro emisor se transforma en:

```
for {
  select {
  case c <- rand.Int():
  case <-time.After(time.Millisecond * 100):
    fmt.Println("timed out")
  }
  time.Sleep(time.Millisecond * 50)
}
```

`time.After` devuelve un canal, así que podemos hacer un **`select`** de él, e indicar qué hacer cuando el tipo indicado ha pasado. Esto es todo, no tiene más magia que esa. Si tienes curiosidad, aquí tienes una implementación de cómo podría ser la función `after`:

```
func after(d time.Duration) chan bool {
  c := make(chan bool)
  go func() {
    time.Sleep(d)
    c <- true
  }()
  return c
}
```

Volviendo a nuestro **`select`**, hay un par de cosas con las que podemos jugar. La primera es, ¿qué ocurre si entramos en el caso `default`?, ¿Lo puedes imaginar?. Inténtalo. Si no estás seguro sobre qué va a pasar, recuerda que el caso `default` se lanza en cuanto nos quedamos sin canales disponibles.

Además, `time.After` es un canal de tipo `chan time.Time`. En el ejemplo anterior simplemente descartamos el valor que fue enviado al canal. Si lo quieres puedes recibirlo:

```
case t := <-time.After(time.Millisecond * 100):
  fmt.Println("timed out at", t)
```

Presta mucha atención a nuestro **`select`**. Observa que enviamos a `c` pero recibimos desde `time.After`. **`select`** funciona independientemente de lo que estemos recibiendo, enviando, o de cualquier combinación de canales:

- Se elige el primer canal disponible.
- Si hay varios disponibles se escoge uno aleatoriamente.
- Si no hay canales disponibles se ejecuta el caso `default`.
- Si no hay caso `default`, el select se bloquea.

Para terminar, es común ver un **`select`** dentro de un **`for`**. Por ejemplo:

```
for {
  select {
```

```
    case data := <-c:
      fmt.Printf("worker %d got %d\n", w.id, data)
    case <-time.After(time.Millisecond * 10):
      fmt.Println("Break time")
      time.Sleep(time.Second)
    }
  }
```

Antes de Continuar

Si eres nuevo en el mundo de la programación concurrente todo esto puede parecerte apabullante. Categóricamente necesita mucha más atención y cuidado, aunque Go hace que sea más sencillo.

Las go-rutinas abstraen de un modo eficiente todo lo necesario para ejecutar código concurrente. Los canales ayudan a eliminar algunos bugs importantes que surgen cuando se comparten datos. Esto no elimina los bugs, pero cambia la forma en la cual se aborda la programación concurrente. Puedes comenzar a pensar en concurrencia en términos de envíos de mensajes en lugar de pensar en secciones críticas.

Dicho esto, personalmente utilizo mucho las primitivas de sincronización que existen en los paquetes `sync` y `sync/atomic`. Considero que es importante sentirse cómodo con ambas aproximaciones. Te animo a que en un principio te centres en los canales, pero que tengas en mente que si tienes un ejemplo sencillo con un cerrojo que va a durar poco tiempo, un mutex o un mutex de lectura-escritura pueden ser buenas soluciones.

Conclusión

He podido oír opiniones que dicen que Go es un lenguaje *aburrido*. Aburrido porque es fácil de aprender, fácil de utilizar y, lo más importante, fácil de leer. Quizá yo mismo he causado algún perjuicio, hemos *pasado* tres capítulos hablando sobre los tipos de datos y cómo declarar variables después de todo.

Si tienes background con algún lenguaje estáticamente tipado mucho de lo que hemos hablado habrá sido, en el mejor de los casos, un recordatorio. Que Go haga que los punteros o que los slices sean poco más que wrappers no es agobiante para desarrolladores Java o C# experimentados.

Si has trabajado principalmente con lenguajes dinámicos puede que lo veas ligeramente diferente. *Es* un pequeño reto que aprender, no es sólo conocer la variedad de sintaxis que hay sobre declaración e inicialización de variables. Aunque sea un fan de Go, creo que el aprendijaze siempre va paralelo a la simplicidad. De todos modos hay que aprender algunas reglas básicas (como que sólo puedes declarar la variable una vez y que `:=` declara la variable) y tener unos conocimientos básicos (como que `new(X)` o `&X{}` sólo reservan memoria mientras que los slices, los mapas y los canales requieren inicialización y, por ello, el uso de `make`).

Tras esto, Go nos da una forma sencilla a la par que efectiva de organizar nuestro código. Los interfaces, la gestión de errores basada en valores de retorno, hacer `defer` en la gestión de recursos y tener una forma sencilla de conseguir composición son algunos ejemplos.

Por último, aunque no por ello menos importante, tenemos el soporte nativo de concurrencia. Hay poco más que decir de las go-rutinas aparte de que son eficientes y simples (simples de utilizar). Son una buena abstracción. Los canales son más complicados. Creo que es importante conocer las bases antes de comenzar a usar wrappers de alto nivel. *Creo* que aprender programación concurrente con canales es útil. Aún así, los canales han sido implementados de una forma que, para mí, no suponen una abstracción sencilla: tienen sus propios principios para construtir software sobre ellos. Digo esto porque cambian la forma en la que piensas y escribes software concurrente, aunque dado lo difícil que es la programación concurrente, es sin duda un buena herramienta.