Go

NachoPacheco

el lenguaje de programación

¡en Español!

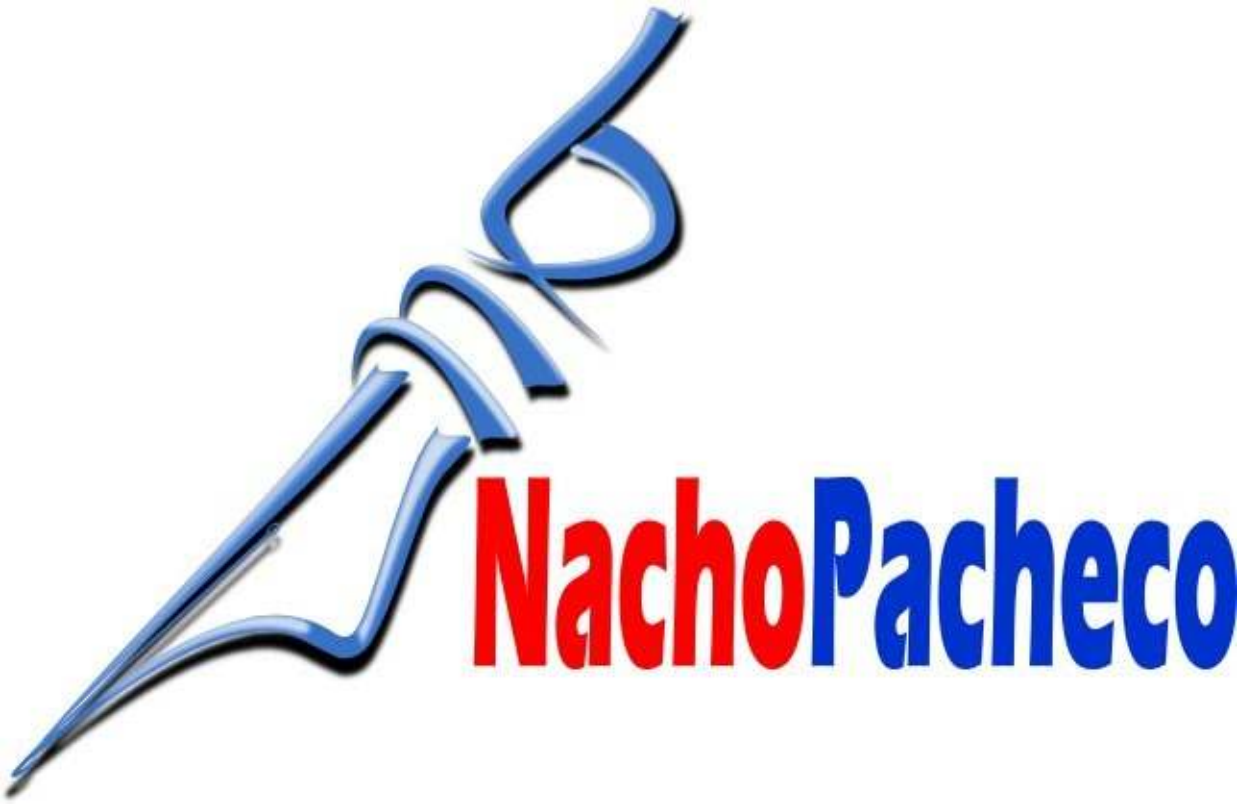
Tabla de contenido

¡Go en Español!

Go, el lenguaje de programación

Go es un lenguaje de programación de código abierto que facilita y simplifica la construcción de software eficiente y fiable.

Obra dedicada especialmente a Maru, mi amada e incondicional cómplice en esta gran aventura, a mis pequeños grandes cachorros —en estricto orden cronológico— Meli, Nacho, Nene, Lala y Judis; Y a todos (no los enumero porque son demasiados y no quiero omitir a nadie) quienes me han impulsado por este sendero.



Audiencia

Esta es *mi personal interpretación* de la **documentación del lenguaje de programación Go**; destinada a quienes les gusta la lectura en español porque se pueden concentrar en lo importante sin estar tratando de encontrar la mejor traducción del texto original —tal como me ocurre frecuentemente—; No obstante, si planeas compartir con todo el mundo el software que vas a desarrollar la norma es que lo escribas en Inglés —por cierto, **no olvides agregar toda la documentación en Español**—.

Este es un proyecto inconcluso cuya intensión es traducir toda la información relacionada con este lenguaje, su conclusión está supeditada a los donativos que se hagan para poder dedicar el tiempo necesario en completar la traducción. Por lo tanto, si te gusta y quieres apoyar el proyecto haz un donativo, luego, ponte en contacto, te haré llegar la traducción de los archivos fuente de Go para que tu instalación de Go ¡te hable en Español!

La esencia de este libro está en el sitio oficial de Go, donde puedes conseguir la información más reciente.

Historia

Segunda versión publicada el 24 de Marzo del 2015

- Actualización en la especificación del lenguaje sección literales compuestos
- Terminada la traducción de Primeros pasos

Primer versión publicada el 22 de Marzo del 2015

Construido con Go 1.5.1

¡Go en Español!

Puedes ver el documento original aquí

- El lenguaje de programación Go
- Instalando Go
  - [Primeros pasos][instalar]
- Aprendiendo Go
  - [Un paseo por Go][paseo]
  - [Cómo escribir código Go][comoescribirgo]
  - [Go eficiente][eficiente]
  - [Preguntas frecuentes (PF)][pf]
  - El wiki de Go
    - Más
- Referencias
  - Documentación de paquetes
  - Documentación de cmd
  - Especificación del lenguaje
  - El modelo de memoria Go
  - Historia de liberación
- Artículos
  - El blog de Go
    - PaseosXcódigo
    - Lenguaje
    - Paquetes
    - Herramientas
    - Más
- Charlas
  - Un video sobre una visita a Go
  - Código que crece con gracia
  - Patrones de concurrencia Go
  - Patrones avanzados de concurrencia Go
    - Más
- Documentación internacional

El lenguaje de programación Go

El lenguaje de programación Go es un proyecto de código abierto para hacer programadores más productivos.

Go es expresivo, conciso, limpio y eficiente. Sus mecanismos de concurrencia facilitan la escritura de programas consiguiendo lo mejor de máquinas multinúcleo y de la red, mientras su novel sistema de tipos permite la construcción de programas flexibles y modulares. Go compila rápidamente a código máquina aún con la comodidad de la recolección de basura y el poder de reflexión en tiempo de ejecución. Es un lenguaje tipado estáticamente, compilado y por lo tanto rápido, que se siente como un lenguaje interpretado y tipado dinámicamente.

Instalando Go

[Primeros pasos][instalar]

Instrucciones para descargar e instalar los compiladores, herramientas y bibliotecas de Go.

Aprendiendo Go

[Un paseo por Go][paseo]

Una introducción interactiva a Go en tres secciones. La primer sección cubre la sintaxis básica y las estructuras de datos; la segunda habla sobre los métodos e interfaces; y la tercera introduce las primitivas de concurrencia de Go. Cada sección concluye con unos cuantos ejercicios con los que puedes practicar lo que has aprendido. Puedes tomar el [paseo en línea][paseo] o [instalarlo localmente][paseolocal].

[Cómo escribir código Go][comoescribirgo]

Disponible también como video, este documento explica cómo utilizar la [orden go][ordengo] para descargar, construir e instalar paquetes, comandos y correr pruebas.

[Go eficiente][eficiente]

Un documento que da consejos para escribir claro e idiomático código Go. Todo nuevo programador de Go lo tiene que leer. Este complementa el paseo y la especificación del lenguaje, mismos que deberías leer primero.

[Preguntas frecuentes (PF)][pf]

Respuestas a preguntas comunes sobre Go.

El wiki de Go

Un wiki mantenido por la comunidad de Go.

Más

Ve la página Aprendiendo en el Wiki para más recursos sobre el aprendizaje de Go.

Referencias

Documentación de paquetes

La documentación de la biblioteca estándar de Go.

Documentación de cmd

La documentación para las herramientas de Go.

Especificación del lenguaje

La especificación oficial del lenguaje Go.

El modelo de memoria Go

Un documento que especifica las condiciones bajo las cuales se puede garantizar que en la lectura de una variable en una rutinago se observan los valores producidos al escribir a la misma variable en una diferente rutinago.

Historia de liberación

Un resumen de los cambios entre liberaciones de Go.

Artículos

El blog de Go

El blog oficial del proyecto Go, presenta noticias y reflexivos artículos escritos por el equipo e invitados de Go.

PaseosXcódigo

Visitas guiadas a programas Go.

- Funciones de primera clase en Go
- Generando texto arbitrario: un algoritmo de la cadena de Markov
- Compartiendo memoria mediante comunicación
- Escribiendo aplicaciones Web - construyendo una sencilla aplicación web.

Lenguaje

- JSON-RPC: un relato de interfaces
- Sintaxis de las declaraciones Go
- Defer, Panic y Recover
- Patrones de concurrencia Go: Cronometrando, continuando
- Sectores Go: uso y funcionamiento interno
- Un decodificador GIF: un ejercicio en interfaces Go
- Manejo de errores y Go
- Organizando código Go

Paquetes

- JSON y Go - usando el paquete json.
- Montones de datos - diseño y uso del paquete gob.
- Las leyes de reflexión - fundamentos del paquete reflect.
- El paquete image de Go - fundamentos del paquete image.
- El paquete image/draw de Go - fundamentos del paquete image/draw.

Herramientas

- Sobre la orden Go - por qué la escribimos, qué es, qué no es y cómo utilizarla.
- ¿C? ¿Go? ¡Cgo! - enlazando contra código C con cgo.
- Depurando código Go con GDB
- Godoc: documentando código Go - escribiendo buena documentación para godoc.

- Perfilando programas Go
- Detector de pugna de datos - un manual para el detector de pugna de datos.
- Introduciendo el detector de pugnas de Go - una introducción al detector de pugnas.
- Una guía rápida para el ensamblador de Go - una introducción al ensamblador usado por Go.

Más

Ve la página de Artículos en el Wiki para más artículos sobre Go.

Charlas

Un video sobre una visita a Go

Tres cosas que hacen que Go sea rápido, divertido y productivo: interfaces, reflexión y concurrencia. Construye un diminuto rastreador web para demostrarlo.

Código que crece con gracia

Uno de los objetivos clave del diseño de Go es la adaptabilidad del código; debe ser fácil tomar un sencillo diseño y construirlo en una manera limpia y natural. En esta charla Andrew Gerrand describe un servidor con una sencilla "ruleta de charlas" que empareja conexiones TCP entrantes y, entonces, usa los mecanismos concurrentes, las interfaces y la biblioteca estándar de Go para extenderlo con una interfaz web y otras características. Si bien, la función del programa cambia drásticamente, la flexibilidad de Go preserva el diseño original cuando crece.

Patrones de concurrencia Go

Concurrencia es la clave para diseñar servicios de red de alto rendimiento. Las primitivas de concurrencia en Go (rutinasgo y canales) proporcionan un medio sencillo y eficiente para expresar la ejecución concurrente. En esta charla vemos cómo se pueden solucionar elegantemente los delicados problemas de concurrencia con sencillo código Go.

Patrones avanzados de concurrencia Go

Esta conferencia amplía la charla sobre los *Patrones de concurrencia Go* profundizando en las primitivas de concurrencia Go.

Más

Ve el sitio de charlas Go y la página del wiki para más charlas sobre Go.

Documentación internacional

Ve la página Internacional en el Wiki para documentación en otros idiomas.

Primeros pasos

Puedes ver el documento original aquí

Descarga la distribución Go

Descarga Go — Haz clic aquí para visitar la página de descargas

Disponemos de distribuciones binarias oficiales para FreeBSD (versión 8-STABLE y superiores), sistemas operativos Linux, Mac OS X (Snow Leopard y superiores) y arquitecturas Windows con procesador x86 de 32 bits (`386`) y 64 bits (`amd64`).

Si una distribución binaria no está disponible para tu combinación de sistema operativo y arquitectura, intenta con la instalación desde fuente o instala gccgo en lugar de gc.

Requisitos del sistema

El compilador `gc` es compatible con los siguientes sistemas operativos y arquitecturas. Por favor, asegúrate de que tu sistema cumple estos requisitos antes de proceder. Si tu sistema operativo o arquitectura no está en la lista, es posible que `gccgo` pueda apoyar tu configuración; ve los detalles de configuración y uso de gccgo.

Sistema operativo	Arquitecturas	Notas
FreeBSD 8-STABLE o superior	amd64, 386, arm	Debian GNU/kFreeBSD no soportado; FreeBSD/ARM necesita FreeBSD 10 o posterior
Linux 2.6.23 o posterior con glibc	amd64, 386, arm	CentOS/RHEL 5.x no soportado; aún no hay una distribución binaria para ARM
Mac OS X 10.6 o posterior	amd64, 386	Utiliza el gcc † que viene con Xcode ‡
Windows XP o posterior	amd64, 386	Utiliza MinGW gcc †. No necesitas cygwin o msys.

† `gcc` sólo se requiere si vas a utilizar CGO.

‡ Sólo necesitas instalar las herramientas de línea de órdenes para Xcode. Si ya tienes instalado Xcode 4.3 +, lo puedes instalar desde la ficha Componentes del Panel de preferencias de descarga.

Instalando las herramientas Go

Si estás actualizando desde una versión anterior de Go primero debes quitar la versión existente.

Linux, Mac OS X y archivos comprimidos para FreeBSD

Descarga el archivo y extraelo en `/usr/local`, creando un árbol Go en `/usr/local/go`. Por ejemplo:

```
tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
```

Elije el archivo apropiado para tu instalación. Por ejemplo, si vas a instalar Go versión 1.4.2 para 64-bits x86 en Linux, el archivo que deseas se llama `go1.4.2.linux-amd64.tar.gz`.

(Normalmente estas órdenes se deben ejecutar como root o por medio de `sudo`).

Añade `/usr/local/go/bin` a la variable de entorno `PATH`. Lo puedes hacer añadiendo esta línea a tu `/etc/profile` (para una instalación en todo el sistema) o `$HOME/.profile`:

```
export PATH=$PATH:/usr/local/go/bin
```

Instalación en una ubicación personalizada

Las distribuciones binarias de Go asumen que se instalarán en `/usr/local/go` (o `c:\Go` en Windows), pero es posible instalar las herramientas de go en un lugar diferente. En ese caso, debes establecer la variable de entorno `GOROOT` para que apunte al directorio en el que lo hayas instalado.

Por ejemplo, si instaláste Go en tu directorio `home` debes agregar las siguientes órdenes a `$HOME/.profile` :

```
export GOROOT=$HOME/go
export PATH=$PATH:$GOROOT/bin
```

Nota:: `GOROOT` se debe establecer sólo cuando se instala en una ubicación personalizada.

Instalador de paquetes Mac OS X

Descarga el archivo del paquete, ábrelo y sigue las instrucciones para instalar las herramientas Go. El paquete instala la distribución Go en `/usr/local/go` .

El paquete debe poner el directorio `/usr/local/go/bin` en tu variable de entorno `PATH` . Posiblemente tengas que reiniciar cualquier sesión de terminal abierta para que el cambio surta efecto.

Windows

El proyecto Go ofrece dos opciones de instalación para los usuarios de Windows (además la instalación desde fuente): un archivo zip que requiere que establezcas algunas variables de entorno y un instalador MSI que configura tu instalación automáticamente.

Instalador MSI

Abre el archivo MSI y sigue las instrucciones para instalar las herramientas Go. De manera predeterminada, el instalador pone la distribución Go en `c:\Go` .

El instalador debe poner el directorio `c:\Go\bin` en tu variable de entorno `PATH` . Posiblemente tengas que reiniciar cualquier sesión abierta para que el cambio surta efecto.

Archivo zip

Descarga el archivo zip y extraelo en el directorio de tu elección (sugerimos `c:\Go`).

Si eliges un directorio distinto de `c:\Go` , debes establecer la variable de entorno `GOROOT` a la ruta elegida.

Añade el subdirectorio `bin` de tu Go raíz (por ejemplo, `c:\Go\bin`) a tu variable de entorno `PATH`.

Configurando las variables de entorno en Windows

En Windows, puedes establecer las variables de entorno a través del botón "Variables de entorno" en la ficha "Avanzado" del panel de control "Sistema". Algunas versiones de Windows proporcionan este panel de control a través de "Configuración Avanzada del Sistema" dentro de la opción "Configuración del sistema" del panel de control.

Probando tu instalación

Verifica que Go está instalado correctamente construyendo un sencillo programa, de la siguiente manera.

Crea un archivo llamado `hola.go` y pon el siguiente código en él:

```
package main

import "fmt"

func main() {
    fmt.Printf("hola, mundo\n")
}
```

Luego ejecútalo con la herramienta `go`:

```
$ go run hola.go
hola, mundo
```

Si ves el mensaje "hola, mundo" entonces tu instalación de Go está trabajando.

Configura tu ambiente de trabajo

Casi terminas. Solo necesitas configurar tu entorno.

Lee el documento Cómo escribir código Go, que proporciona **instrucciones de configuración esenciales** para el uso de las herramientas Go.

Desinstalando Go

Para eliminar de tu sistema una instalación de Go existente borra el directorio `Go`. Este suele ser `/usr/local/go` bajo Linux, Mac OS X y FreeBSD o `c:\Go` bajo Windows.

También debes eliminar el directorio `go/bin` de tu variable de entorno `PATH`. Bajo Linux y FreeBSD deberías editar `/etc/profile` o `$HOME/.profile`. Si instalaste Go con el paquete Mac OS X luego debes quitar el archivo `/etc/paths.d/go`. Los usuarios de Windows deben leer la sección sobre ajuste de las variables de entorno bajo Windows.

Consiguiendo ayuda

Para ayuda en tiempo real, pregunta a los serviciales gophers en el canal `#go-nuts` del servidor IRC Freenode.

La lista de correo oficial para discusión del lenguaje Go es Go Nuts.

Informa los errores utilizando el gestor de incidencias Go.

Cómo escribir código Go

Introducción

Este documento muestra el desarrollo de un sencillo paquete Go e introduce la herramienta go, la manera estándar para descargar, construir e instalar paquetes y órdenes Go.

La herramienta `go` requiere que organices tu código en una manera específica. Por favor lee cuidadosamente este documento. Este explica la manera más sencilla de configurar y ejecutar tu instalación de Go.

Disponemos de una explicación similar en video (en inglés).

Organizando el código

Ambiente de trabajo

La herramienta `go` está diseñada para trabajar con código de fuente abierta mantenido en repositorios públicos. Aunque no es necesario publicar tu código, el modelo para configurar el ambiente de trabajo es igual si lo haces o no.

El código Go se debe mantener dentro de un *ambiente de trabajo*. Un ambiente de trabajo es una jerarquía de directorios con tres ramas en su raíz:

- `src` contiene los archivos de código fuente Go organizados en paquetes (un paquete por directorio),

- `pkg` contiene objetos paquete, y
- `bin` contiene las órdenes ejecutables.

La herramienta `go` construye paquetes fuente e instala los binarios resultantes a los directorios `pkg` y `bin`.

El subdirectorio `src` típicamente contiene múltiples repositorios de control de versiones (tal como Git o Mercurial) que siguen la pista al código fuente del desarrollo de uno o más paquetes.

Para darte una idea de cómo se ve un ambiente de trabajo en la práctica, aquí tienes un ejemplo:

```
bin
    hola                        # orden ejecutable
    yasalio                     # orden ejecutable
pkg
    linux_amd64
        github.com/gitnacho/ejemplo
            utilcadenas.a       # objecto paquete
src
    github.com/gitnacho/ejemplo
        .git/                   # metadatos del repositorio git
        hola
            hola.go             # fuente de la orden
        yasalio
            main.go             # fuente de la orden
            main_test.go        # fuente de la prueba
        utilcadenas
            reverso.go          # fuente del paquete
            reverso_test.go     # fuente de la prueba
```

Este ambiente de trabajo contiene un repositorio (`ejemplo`) que comprende dos órdenes (`hola` y `yasalio`) y una biblioteca (`utilcadenas`).

Un ambiente de trabajo típico debería tener varios repositorios fuente conteniendo muchos paquetes y órdenes. La mayoría de los programadores Go mantienen *todo* su código fuente y dependencias de Go en un único ambiente de trabajo.

Las órdenes y bibliotecas se construyen de diferentes clases de paquetes fuente. Hablaremos sobre esta distinción más adelante.

La variable de entorno `GOPATH`

La variable de entorno `GOPATH` especifica la ubicación de tu ambiente de trabajo. Probablemente esta puede ser la única variable de entorno que necesitarás configurar cuando desarrollas código Go.

Para empezar, crea un directorio para tu ambiente de trabajo y configura `GOPATH` consecuentemente. Tu ambiente de trabajo se puede localizar en dónde quieras, no obstante, en este documento utilizaremos `$HOME/go`. Ten en cuenta que esta **no debe ser** la misma ruta que tu instalación de Go.

```
$ mkdir $HOME/go
$ export GOPATH=$HOME/go
```

Por comodidad, añade el subdirectorio `bin` del ambiente de trabajo a tu `PATH`:

```
$ export PATH=$PATH:$GOPATH/bin
```

Rutas de paquetes

A los paquetes de la biblioteca estándar se les asignan rutas cortas tal como `&quot;fmt&quot;` y `&quot;http/net&quot;`. Para tus propios paquetes, debes escoger una ruta base que sea poco probable pueda colisionar con futuras adiciones a la biblioteca estándar u otras bibliotecas externas.

Si mantienes tu código en algún repositorio fuente, entonces deberías utilizar la raíz de ese repositorio fuente como tu ruta base. Por ejemplo, si tu cuenta GitHub está en `github.com/usuario`, esa debería ser tu ruta base.

Ten en cuenta que no necesitas publicar tu código a un repositorio remoto antes de que lo puedas construir. solo es un buen hábito para organizar tu código como si algún día lo fueras a publicar. En la práctica puedes escoger cualquier nombre de ruta arbitrario, siempre y cuando sea único en la biblioteca estándar y en el ecosistema Go.

Utilizaremos `github.com/usuario` como nuestra ruta base. Crea un directorio dentro de tu ambiente de trabajo en el cual mantendrás el código fuente:

```
$ mkdir -p $GOPATH/src/github.com/usuario
```

Tu primer programa

Para compilar y correr un sencillo programa, primero escoge una ruta para el paquete (utilizaremos `github.com/usuario/hola`) y crea un directorio para el paquete correspondiente dentro de tu ambiente de trabajo:

```
$ mkdir $GOPATH/src/github.com/usuario/hola
```

Luego, crea un archivo llamado `hola.go` dentro de ese directorio, conteniendo el siguiente código Go.

```
package main

import "fmt"

func main() {
    fmt.Printf("Hola, mundo.\n")
}
```

Ahora puedes construir e instalar el programa con la herramienta `go`:

```
$ go install github.com/usuario/hola
```

Ten en cuenta que puedes ejecutar esta orden desde cualquier lugar en tu sistema. La herramienta `go` encuentra el código fuente buscando el paquete `github.com/usuario/hola` dentro del ambiente de trabajo especificado por `GOPATH`.

También puedes omitir la ruta del paquete si ejecutas `go install` desde el directorio del paquete:

```
$ cd $GOPATH/src/github.com/usuario/hola
$ go install
```

Estas instrucciones construyen la orden `hola`, produciendo un ejecutable binario. Esto instala el binario en el directorio `bin` del ambiente de trabajo como `hola` (o, bajo Windows, `hola.exe`). En nuestro ejemplo, este será `$GOPATH/bin/hola`, el cual está en `$HOME/go/bin/hola`.

La herramienta `go` solo producirá salida cuando ocurra un error, así que si estas órdenes no producen ninguna salida es porque se han ejecutado satisfactoriamente.

Ahora puedes ejecutar el programa escribiendo la ruta completa en la línea de órdenes:

```
$ $GOPATH/bin/hola
Hola, mundo.
```

O, suponiendo que añadíste `$GOPATH/bin` a tu `PATH`, solo escribe el nombre del binario:

```
$ hola
Hola, mundo.
```

Si estás utilizando un sistema de control de versiones, ahora sería un buen momento para iniciar un repositorio, añadir los archivos y enviar tu primer entrega. Una vez más, este paso es opcional: no es necesario utilizar el control de versiones para escribir código Go.

```
$ cd $GOPATH/src/github.com/usuario/hola
$ git init
Inicia repositorio Git vacío en /home/user/go/src/github.com/usuario/hola/.git

$ git add hola.go
$ git commit -m "entrega inicial"
[master (root-commit) 0b4507d] entrega inicial
 1 archivo cambió, 1 insercion(+)
  crea modo 100644 hola.go
```

Dejamos el envío del código a un repositorio remoto como ejercicio para el lector.

Tu primer biblioteca

Vamos a escribir una biblioteca y a usarla desde el programa `hola`.

De nuevo, el primer paso es escoger una ruta para el paquete (utilizaremos `github.com/usuario/utilcadenas`) y creamos el directorio del paquete:

```
$ mkdir $GOPATH/src/github.com/usuario/utilcadenas
```

A continuación, crea un archivo llamado `reverso.go` en ese directorio con el siguiente contenido.

```
// utilcadenas este paquete contiene útiles funciones para trabajar
// con cadenas de caracteres.
package utilcadenas

// Reverso invierte su argumento s dejándolo legible de izquierda
// a derecha.
func Reverso(s string) string {
    r := []rune(s)
    for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {
        r[i], r[j] = r[j], r[i]
    }
    return string(r)
}
```

Ahora, comprueba que el paquete compila con `go build`:

```
$ go build github.com/usuario/utilcadenas
```

O, si estás trabajando en el directorio fuente del paquete, justo:

```
$ go build
```

Esto no producirá un archivo. Para ello, tienes que utilizar `go install`, que coloca el objeto paquete dentro del directorio `pkg` del ambiente de trabajo.

Después de confirmar la construcción del paquete `utilcadenas`, modifica tu `hola.go` original (que está en `$GOPATH/src/github.com/usuario/hola`) para utilizarlo:

```
package main

import (
    "fmt"
    "github.com/usuario/utilcadenas"
)

func main() {
    fmt.Printf(utilcadenas.Reverso("!oG ,aloH¡"))
}
```

Siempre que la herramienta `go` instala un paquete o binario, también instala todas las dependencias que tenga. Por lo tanto cuando instalas el programa `hola`

```
$ go install github.com/usuario/hola
```

Automáticamente también instalará el paquete `utilcadenas`.

Al ejecutar la nueva versión del programa, deberías ver un nuevo mensaje, invertido:

```
$ hola
¡Hola, Go!
```

Después de los pasos anteriores, tu ambiente de trabajo se parecerá a este:

```
bin
        hola                    # orden ejecutable
pkg
        linux_amd64/            # esto reflejará tu SO y arquitectura
        github.com/usuario
                utilcadenas.a   # objecto paquete
src
        github.com/usuario
            hola
                hola.go         # fuente de la orden
        utilcadenas
                reverso.go      # fuente del paquete
```

Ten en cuenta que `go install` colocó el objeto `utilcadenas.a` en un directorio dentro de `pkg/linux_amd64` que refleja tu directorio fuente. Esto es a modo de que futuras invocaciones a la herramienta `go` puedan encontrar el objeto paquete y evitar recompilar innecesariamente el paquete. La parte `linux_amd64` está allí para ayudar en recompilaciones cruzadas, y refleja el sistema operativo y arquitectura de tu sistema.

Las órdenes ejecutables se enlazan estáticamente; los objetos paquete no es necesario que estén presentes para ejecutar los programas Go.

Nombre de paquetes

La primer declaración en un archivo fuente de Go tiene que ser:

```
package nombre
```

Dónde `nombre` es el nombre predeterminado del paquete para importaciones. (Todos los archivos en un paquete tienen que utilizar el mismo `nombre`).

Una convención en Go es que el nombre del paquete es el último elemento de la ruta de importación: el paquete importado como "`crypto/rot13`" se debería llamar `rot13`.

Las órdenes ejecutables siempre tienen que usar `package main`.

No hay ningún requisito para que los nombres de paquete sean únicos entre todos los paquetes enlazados a un solo binario, solo que las rutas de importación (sus nombres de archivo completos) sean únicos.

Ve Go eficiente para aprender más sobre las convenciones de nomenclatura en Go.

Probando

Go tiene una ligera plataforma de pruebas compuesta de la orden `go test` y el paquete `testing`.

Escribes una prueba creando un archivo con un nombre que termine en `_test.go` el cual contiene las funciones llamadas `TestXXX` con la firma `func (t *testing.T)`. La plataforma de pruebas corre cada una de esas funciones; si la función invoca a una función de fallo tal como `t.Error` o `t.Fail`, se considerada que la prueba falló.

Añade una prueba al paquete `utilcadenas` creando el archivo `$GOPATH/src/github.com/usuario/utilcadenas/reverso_test.go` conteniendo el siguiente código Go.

```
package utilcadenas

import "testing"

func TestReverso(t *testing.T) {
    casos := []struct {
        ingresada, deseada string
    }{
        {"Hola, mundo", "odnum ,aloH"},
        {"Hola, 世界", "界世 ,aloH"},
        {"", ""},
    }
    for _, c := range casos {
        obtuve := Reverso(c.ingresada)
        if obtuve != c.deseada {
            t.Errorf("Reverso(%q) == %q, deseada %q", c.ingresada, obtuve, c.deseada)
        }
    }
}
```

Entonces corre la prueba con `go test`:

```
$ go test github.com/usuario/utilcadenas
ok      github.com/usuario/utilcadenas 0.165s
```

Como siempre, si estás corriendo la herramienta `go` desde el directorio del paquete, puedes omitir la ruta del paquete:

```
$ go test
ok      github.com/usuario/utilcadenas 0.165s
```

Corre go help test y ve la documentación del paquete testing para más detalles.

Paquetes remotos

Una ruta de importación puede describir cómo obtener el código fuente del paquete utilizando un sistema de control de versiones tal como Git o Mercurial. La herramienta `go` utiliza esta propiedad para automáticamente descargar paquetes desde repositorios remotos. Por ejemplo, los fragmentos de código descritos en este documento también están mantenidos en un repositorio Git hospedado en Github, github.com/gitnacho/ejemplo. Si incluyes el URL del repositorio en la ruta de importación del paquete, `go get` lo descargará, construirá e instalará automáticamente:

```
$ go get github.com/gitnacho/ejemplo/hola
$ $GOPATH/bin/hola
¡Hola, ejemplos Go!
```

Si el paquete especificado no está presente en un ambiente de trabajo, `go get` lo colocará dentro del primer ambiente de trabajo especificado por `GOPATH`. (Si el paquete ya existe, `go get` omitirá la descarga remota y se comportará igual que `go install`).

Después de emitir la orden `go get` anterior, ahora el árbol de directorios del ambiente de trabajo se debería parecer a este:

```
bin
    hello                       # orden ejecutable
pkg
    linux_amd64
        github.com/gitnacho/ejemplo
            utilcadenas.a       # objeto paquete
        github.com/usuario
            utilcadenas.a       # objecto paquete
src
    github.com/gitnacho/ejemplo
        .git/                   # metadatos del repositorio Git
        hola
            hola.go             # fuente de la orden
        utilcadenas
            reverso.go          # fuente del paquete
            reverso_test.go     # fuente de pruebas
    github.com/usuario
        hola
            hola.go             # fuente de la orden
        utilcadenas
            reverso.go          # fuente del paquete
            reverso_test.go     # fuente de pruebas
```

La orden `hola` hospedada en Github depende del paquete `utilcadenas` dentro del mismo repositorio. Las importaciones en el archivo `hola.go` utilizan la misma convención de las rutas de importación, por lo tanto la orden `go get` también es capaz de localizar e instalar el paquete dependiente.

```
import "github.com/gitnacho/ejemplo/utilcadenas"
```

Esta convención es la manera más fácil de permitir que tus paquetes Go estén disponibles para que otros los utilicen. El Wiki de Go y godoc.org proporcionan listas de proyectos Go externos.

Para más información sobre el uso de repositorios remotos con la herramienta `go`, ve go help importpath.

Algo más

Suscríbete a la lista de correo golang-announce para recibir notificaciones de cuándo se liberará una nueva versión estable de Go.

Ve Go eficiente para ver algunos consejos sobre cómo escribir claro e idiomático código Go.

Toma Un paseo por Go para aprender adecuadamente el lenguaje.

Visita la página de documentación para un conjunto de artículos que profundizan en el lenguaje Go, sus bibliotecas y herramientas.

Consiguiendo ayuda

Para ayuda en tiempo real, pregunta a los serviciales gophers en el canal `#go-nuts` del servidor IRC Freenode.

La lista de correo oficial para discusión del lenguaje Go es Go Nuts.

Reporta fallos utilizando el Gestor de incidencias Go.

Go eficiente

Puedes ver el documento original aquí

Introducción

Go es un nuevo lenguaje. A pesar de que toma ideas prestadas de lenguajes existentes, tiene propiedades inusuales que hacen que los eficientes programas Go sean diferentes en carácter a los programas escritos en sus parientes. Una sencilla traducción de un programa C++ o Java a Go probablemente no pueda producir un resultado satisfactorio —los programas Java están escritos en Java, no en Go. Por otro lado, pensando sobre el problema desde la perspectiva de Go podría producir un programa exitoso pero bastante diferente. En otras palabras, para escribir buen Go, es importante entender sus propiedades y modismos. Además, es importante conocer las convenciones establecidas para programar en Go, tal como la nomenclatura, formateo, construcción del programa, etc., de modo tal que los programas que escribas los entiendan fácilmente otros programadores de Go.

Este documento proporciona consejos para escribir código Go claro e idiomático. Este complementa la especificación del lenguaje, Un paseo por Go y Cómo escribir código Go, mismos que deberías leer primero.

Ejemplos

La fuente de los paquetes Go se pretende sirva no solo como la biblioteca del núcleo sino también como ejemplos de cómo utilizar el lenguaje. Además, muchos de los paquetes contienen arquetipos ejecutables autocontenidos que puedes correr directamente desde el sitio web golang.org, tal como este (si es necesario, haz clic en la palabra "Ejemplo" para abrirlo).

Si tienes alguna pregunta sobre cómo abordar un problema o cómo se podría implementar algo, la documentación, el código y los ejemplos en la biblioteca te pueden proporcionar respuestas, ideas y antecedentes.

Formateo

El tema del formateo es el más contencioso pero al que se le presta menos importancia. La gente puede adoptar diferentes estilos de formateo pero sería mejor si no tuvieran que hacerlo y dedicarían menos tiempo al tema si todo mundo se adhiriese al mismo estilo. El problema es cómo abordar esta utopía sin una gran guía de estilo prescriptiva.

En Go adoptamos un enfoque inusual y dejamos que la máquina cuide de la mayoría de los problemas de formateo. El programa `gofmt` (también disponible como `go fmt`, el cual opera a nivel de paquete más que a nivel de archivo fuente) lee un programa Go y emite la fuente en un estilo de sangría y alineación vertical estándar, reteniendo y si es necesario reformateando los comentarios. Si quieres saber cómo manejar algunas nuevas situaciones de diseño, ejecuta `gofmt`; si el resultado no te parece bien, reacomoda tu programa (opcionalmente registra un fallo sobre `gofmt`) y no te preocupes más de ello.

Por ejemplo, no hay ninguna necesidad de desperdiciar tiempo alineando los comentarios en los campos de una estructura. `Gofmt` lo hará por ti. Dada la declaración:

```
type T struct {
    nombre string // nombre del objeto
    valor int // su valor
}
```

`gofmt` alineará las columnas:

```
type T struct {
    nombre string  // nombre del objeto
    valor int      // su valor
}
```

Todo el código Go en los paquetes estándar se ha formateado con `gofmt`.

Aquí tienes algunos detalles de formato. Muy brevemente:

Sangría

> *Utilizamos tabuladores para la sangría y de manera predeterminada `gofmt` los emite. Usa espacios solo si es necesario.*

Longitud de línea

> *Go no tiene ningún límite de longitud de línea. No te preocupes por desbordar una tarjeta perforada. Si sientes que una línea es demasiado larga, envuélvela y sángrala con un tabulador extra.*

Paréntesis

> *Go necesita menos paréntesis que C y Java: las estructuras de control (`if`, `for`, `switch`) no tienen paréntesis en su sintaxis. Además, la jerarquía del operador de precedencia es más corta y más clara, por lo tanto:*
>
> ```
> x<<8 + y<<16
> ```
>
> *significa lo que el espaciado implica, a diferencia de los otros lenguajes.*

Comentarios

Go proporciona comentarios de bloque al estilo de C `/* */` y comentarios de línea al estilo de C++ `//`. Los comentarios de línea son la norma; los comentarios de bloque generalmente aparecen como comentarios de paquete, pero son útiles dentro de una expresión o para desactivar grandes franjas de código.

El programa —y servidor web— `godoc` procesa archivos fuente de Go para extraer la documentación desde el contenido del paquete. Extrae los comentarios que aparecen antes de las declaraciones de nivel superior, sin tomar en cuenta los saltos de línea, junto con su declaración para que sirvan como el texto explicativo del elemento. La naturaleza y estilo de estos comentarios determina la calidad de la documentación que `godoc` produce.

Cada paquete debería tener un *comentario de paquete*, un comentario de bloque que preceda a la cláusula `package`. Para paquetes multiarchivo, el comentario de paquete únicamente debe estar presente en un archivo, en cualquiera de ellos funcionará. El comentario de paquete debe introducir el paquete y proporcionar información relevante sobre el paquete en su conjunto. Esta aparecerá primero en la página `godoc` y debería configurar la documentación detallada que sigue.

```
/*
regexp. Este paquete implementa una sencilla biblioteca para expresiones
        regulares.
La sintaxis aceptada de las expresiones regulares es:
regexp:
        Concatenación { '|' concatenación }
    concatenación:
        { cierre }
    cierre:
        término [ '*' | '+' | '?' ]
    término:
        '^'
        '$'
        '.'
        carácter
        '[' [ '^' ] rangos de caracteres ']'
        '(' regexp ')'
*/
package regexp
```

Si el paquete es sencillo, el comentario de paquete puede ser breve.

```
// path. Este paquete implementa útiles rutinas para manipular
// rutas a nombres de archivo separados por barras inclinadas.
```

Los comentarios no necesitan formateo extra tal como viñetas de estrellas. La salida generada incluso puede no presentarse en un tipo de letra de ancho fijo, así que no depende del espaciado para alineamiento de `godoc`, puesto que `gofmt`, cuida de ello. Los comentarios son texto sencillo no interpretado, así que HTML y otras anotaciones como `_esta_` se reproducirán *literalmente* y no se deberían utilizar. Un ajuste que hace `godoc` es mostrar texto sangrado en un tipo de letra de ancho fijo, propio para fragmentos de programa. El comentario principal del paquete `fmt` utiliza este buen efecto.

Dependiendo del contexto, `godoc` incluso podría reformatear los comentarios, así que asegúrate que se vean bien desde el principio: usa ortografía, puntuación y estructura de frases correcta, envuelve líneas largas, etc.

Dentro de un paquete, cualquier comentario precedido inmediatamente por una declaración de nivel superior sirve como *comentario doc* para esa declaración. Cada nombre exportado (con mayúscula inicial) en un programa debería tener un comentario doc.

Los comentarios doc trabajan mejor en frases completas, las cuales dejan una amplia variedad de presentaciones automatizadas. La primera frase debería ser un resumen de una oración que inicie con el nombre a declarar.

```
// Compile analiza una expresión regular y devuelve, de ser exitosa, un
// objeto Regexp que suele usarse para encontrar el texto buscado.
func Compile(str string) (regexp *Regexp, err error) {
```

Si el comentario siempre empieza con el nombre, la salida de `godoc` se puede correr convenientemente a travez de `grep`. Imagina que no puedes recordar el nombre "Compile" pero estás buscando la función para analizar expresiones regulares, así que ejecutas la orden:

```
$ godoc regexp | grep analiza
```

Si todos los comentarios doc en el paquete empiezan con "Esta función...", `grep` no te ayudará a recordar el nombre. Pero debido a que en el paquete cada comentario doc empieza con el nombre, verás algo cómo esto, lo cual te recuerda la palabra que estás buscando.

```
$ godoc regexp | grep analiza
    Compile analiza una expresión regular y devuelve, de ser exitosa, una Regexp
    analizada sintácticamente. Esto simplifica la iniciación segura de las variables glob
    que no se pueden analizar. Esto simplifica la iniciación segura de variables globales
$
```

La sintaxis de declaración Go te permite agrupar declaraciones. Un solo comentario doc puede introducir un grupo relacionado de constantes o variables. Debido a que se presenta la declaración completa, a menudo tal comentario puede ser superficial.

```
// Códigos de error devueltos por fallos al analizar una expresión.
var (
    ErrInternal      = errors.New("regexp: error interno")
    ErrUnmatchedLpar = errors.New("regexp: incompatible '('")
    ErrUnmatchedRpar = errors.New("regexp: incompatible ')'")
    ...
)
```

El agrupamiento también puede indicar relaciones entre elementos, tal como el hecho de que un conjunto de variables esté protegido por una exclusión mutua.

```
var (
    countLock   sync.Mutex
    inputCount  uint32
    outputCount uint32
    errorCount  uint32
)
```

Nombres

Los nombres son tan importantes en Go cómo en cualquier otro lenguaje. Incluso tienen efecto semántico: la visibilidad de un nombre fuera de un paquete está determinada por si su primer carácter está en mayúscula. Por tanto dedicaremos un poco de tiempo para hablar sobre las convenciones de nomenclatura en los programas Go.

Nomenclatura de paquetes

Cuando se importa un paquete, el nombre del paquete proviene de un método de acceso al contenido. Después de:

```
import "bytes"
```

El paquete importador puede hablar sobre `bytes.Buffer`. Es útil si todos los que utilizan el paquete pueden usar el mismo nombre para referirse a su contenido, lo cual implica que el nombre del paquete tendría que ser bueno: corto, conciso, evocador. Por convención, a los paquetes se les dan nombres de una sola palabra en minúsculas; no debería haber necesidad de guiones bajos o mayúsculas intercaladas. Errando por el lado de la brevedad, puesto que cualquiera que utilice tu paquete tendrá que escribir ese nombre. Y *a priori* no te preocupan las colisiones. El nombre del paquete es solo el nombre predeterminado para importaciones; este no tiene que ser único entre todo el código fuente y en el raro caso de una colisión el paquete importador puede elegir un nombre diferente para utilizarlo localmente. En cualquier caso, la confusión es rara porque el nombre de archivo en la importación determina justo qué paquete se está utilizando.

Otra convención es que el nombre del paquete es el nombre base de su directorio fuente; el paquete en `src/pkg/encoding/base64` se importa como `&quot;encoding/base64&quot;` pero se llama `base64`, no `encoding_base64` ni `encodingBase64`.

El importador de un paquete utilizará el nombre para referirse a su contenido, por lo tanto los nombres exportados en el paquete pueden utilizar este hecho para evitar confusiones. (No utilices la notación `import .`, la cual puede simplificar pruebas que debes correr fuera del paquete que estás probando, al contrario debes evitarla). Por ejemplo, el tipo lector de búfer en el paquete `bufio` se llama `Reader`, no `BufReader`, debido a que los usuarios ven `bufio.Reader`, porque es un nombre claro y conciso. Además, dado que las entidades importadas siempre van precedidas por el nombre del paquete, `bufio.Reader` no choca con `io.Reader`. Del mismo modo, la función para crear nuevas instancias de `ring.Ring` —que es la definición de un *constructor* en Go— normalmente se llamaría `NewRing`, pero puesto que `Ring` es el único tipo exportado por el paquete y dado que el paquete se llama `ring`, justamente estás llamando a `New`, cuyos clientes del paquete ven como `ring.New`. Usa la estructura del paquete para ayudarte a escoger buenos nombres.

Otro breve ejemplo es `unavez.Haz`; `unavez.Haz(ajusta)` se lee bien y no mejoraría al escribir `unavez.HazOEsperaHastaQueEsteHecho(ajusta)`. Los nombres largos no hacen las cosas más legibles automáticamente. A menudo, un útil comentario doc puede ser más valioso que un nombre extralargo.

Captadores

Go no proporciona soporte automático para captadores y definidores. No hay nada incorrecto en proporcionar captadores y definidores y a menudo es apropiado hacerlo, pero tampoco es idiomático ni necesario poner `Obt` al nombre del captador. Si tienes un campo llamado `propietario` (en minúsculas, no exportado), el método captador se tendría que llamar `Propietario` (en mayúsculas, exportado), no `ObtPropietario`. El uso de mayúsculas en los nombres para exportación proporciona el gancho para diferenciar un campo de un método. Una función definidora, si la necesitas, probablemente se llamará `EstPropietario`. Ambos nombres se leen bien en la práctica:

```
propietario := obj.Propietario()
if propietario != usuario {
    obj.EstPropietario(usuario)
}
```

Nombre de interfaces

Por convención, para denominar un método de interfaz se utiliza el nombre del método más un sufijo -er o modificación similar para construir un sustantivo del agente: `Reader`, `Writer`, `Formatter`, `CloseNotifier`, etc.

Para honrrarlos hay una serie de nombres productivos y nombres de funciones captadoras. `Read`, `Write`, `Close`, `Flush`, `String` y así por el estilo que tienen significado y firmas canónicas. Para evitar confusión, no denomines tus métodos con uno de esos nombres a no ser que tenga la misma firma y significado. En cambio, si tu tipo implementa un método con el mismo significado que un método en un tipo bien conocido, dale el mismo nombre y firma; ponle el nombre `String` a tu método convertidor a cadena no `ToString`.

Mayúsculas intercaladas

Finalmente, la convención en Go es utilizar `MayúsculasIntercaladas` o `mayúsculasIntercaladas` en lugar de guiones bajos para escribir nombres multipalabra.

Puntos y comas

Como en C, la gramática formal de Go usa puntos y comas para terminar declaraciones, pero a diferencia de C, esos puntos y comas no aparecen en la fuente. En su lugar el analizador léxico al escanear la fuente utiliza una sencilla regla para insertar puntos y comas automáticamente, así que de entrada, el texto mayoritariamente está libre de ellos.

La regla es esta. Si el último segmento antes de una nueva línea es un identificador (el cuál incluye palabras como `int` y `float64`), un literal básico tal como un número, una cadena constante o uno de los símbolos

```
break continue fallthrough return ++ -- ) }
```

El analizador léxico siempre inserta un punto y coma después del símbolo. Esto se podría resumir como, “si la nueva línea viene después de un segmento que pudiera acabar una declaración, inserta un punto y coma”.

Un punto y coma también se puede omitir inmediatamente antes de una llave de cierre, así que una declaración como

```
go func() { for { dst <- <-fnt } }()
```

no necesita punto y coma. Los programas idiomáticos de Go tienen puntos y comas solo en sitios como las cláusulas del bucle `for`, para separar el iniciador, la condición y los elementos de continuación. También son necesarios para separar múltiples declaraciones en una línea, debes escribir código de esa manera.

Una consecuencia de las reglas de inserción automática del punto y coma es que no puedes poner la llave de apertura de una estructura de control (`if` , `for` , `switch` o `select`) en la siguiente línea. Si lo haces, se insertará un punto y coma antes de la llave, el cual podría causar efectos no deseados. Escríbelos así

```
if i < f() {
    g()
}
```

no así

```
if i < f()  // ¡Incorrecto!
{           // ¡Incorrecto!
    g()
}
```

Estructuras de control

Las estructuras de control de Go están relacionadas a las de C pero difieren en importantes maneras. No hay bucles `do` o `while` , solo un ligeramente generalizado `for` ; `switch` es más flexible; `if` y `switch` aceptan una declaración de inicio opcional como la del `for` ; las declaraciones `break` y `continue` toman una etiqueta opcional para identificar qué interrumpir o continuar; y hay nuevas estructuras de control incluyendo un tipo `switch` y un multiplexor de comunicaciones multivía, `select` . La sintaxis también es ligeramente diferente: no hay paréntesis y los cuerpos siempre tienen que estar delimitados por llaves.

If

En Go una sencilla `if` tiene esta apariencia:

```
if x > 0 {
    return y
}
```

Las llaves obligatorias alientan la escritura de sencillas declaraciones `if` en múltiples líneas. Es buen estilo hacerlo en todo caso, especialmente cuando el cuerpo contiene una instrucción de control tal como `return` o `break` .

Dado que `if` y `switch` aceptan una declaración de iniciación, es común ver una usada para configurar una variable local.

```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```

En las bibliotecas de Go, encontrarás que cuando una declaración `if` no ifluye en la siguiente instrucción —es decir, el cuerpo termina en `break`, `continue`, `goto` o `return` — se omite el `else` innecesario.

```
f, err := os.Open(nombre)
if err != nil {
    return err
}
códigoUsando(f)
```

Este es un ejemplo de una situación común donde el código tiene que vigilar una secuencia de condiciones de error. El código se lee bien si el flujo de control es exitoso adelanta la página, eliminando casos de error cuando surgen. Puesto que los casos de error tienden a terminar en declaraciones `return`, el código resultante no necesita declaraciones `else`.

```
f, err := os.Open(nombre)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    f.Close()
    return err
}
códigoUsando(f, d)
```

Redeclaración y reasignación

Aparte: El último ejemplo en la sección anterior demuestra un detalle de cómo trabaja la declaración corta de variables `:=`. La declaración que llama a `os.Open` dice:

```
f, err := os.Open(nombre)
```

Esta declaración crea dos variables, `f` y `err`. Unas cuantas líneas más abajo, la llamada a `f.Stat` dice,

```
d, err := f.Stat()
```

la cual se ve como si declarara `d` y `err`. Observa que, no obstante, `err` aparece en ambas declaraciones. Esta duplicidad es legal: `err` fue creada en la primera declaración, pero únicamente *reasignada* en la segunda. Esto significa que la llamada a `f.Stat` utiliza la variable `err` existente declarada arriba y solo le da un nuevo valor.

En una declaración `:=` puede aparecer una variable `v` incluso si ya se ha declarado, siempre y cuando:

- Esa declaración esté en el mismo ámbito que la declaración existente de `v` (si `v` ya estuviera declarada en un ámbito exterior, la declaración creará una nueva variable),
- El valor correspondiente en la iniciación es asignable a `v` y
- Cuándo menos se crea una nueva variable en esa declaración.

Esta inusual propiedad es pragmatismo puro, facilitando la utilización de un solo valor de `err`, por ejemplo, en una larga cadena `if-else`. Verás que esto se utiliza a menudo.

Aquí, vale la pena recalcar que en Go el ámbito de los parámetros de función y el de los valores de retorno es el mismo que en el cuerpo de la función, incluso aunque léxicamente aparecen fuera de las llaves que encierran el cuerpo.

For

El bucle `for` de Go es similar —a pero no igual— al de C. Este unifica `for` y `while` y no hay `do-while`. Hay tres formas, solo una de las cuales tiene puntos y comas.

```
// Como un for C
for inicio; condición; incremento { }

// Como un while C
for condición { }

// Como un for(;;) C
for { }
```

Las instrucciones cortas facilitan la declaración de la variable inicial en el bucle.

```
suma := 0
for i := 0; i < 10; i++ {
    suma += i
}
```

Si estás recorriendo un arreglo, sector, cadena o mapa, o estás leyendo desde un canal, una cláusula `range` puede manejar el bucle.

```
for clave, valor := range mapaAnterior {
    nuevoMapa[clave] = valor
}
```

Si solo necesitas el primer elemento en el rango (la clave o índice), quita el segundo:

```
for clave := range m {
    if clave.expira() {
        delete(m, clave)
    }
}
```

Si solo necesitas el segundo elemento en el rango (el valor), usa el *identificador blanco*, un guión bajo, para desechar el primero:

```
suma := 0
for _, valor := range array {
    suma += valor
}
```

El identificador blanco tiene muchos usos, como se describe en una sección más adelante.

Para las cadenas, `range` hace más trabajo para ti, dividiendo los caracteres Unicode individuales mediante el análisis del UTF-8. Las codificaciones erróneas consumen un byte y producen la sustitución del rune U+FFFD. (El nombre —con tipo incorporado asociado— `rune` es terminología Go para un solo carácter Unicode. Ve los detalles en la especificación del lenguaje). El bucle:

```
for pos, carácter := range "日本\x80語" { // \x80 es una codificación UTF-8
                                        //  illegal
    fmt.Printf("el carácter %#U empieza en el byte %d\n", carácter, pos)
}
```

imprime:

```
el carácter U+65E5 '日' empieza en el byte 0
el carácter U+672C '本' empieza en el byte 3
el carácter U+FFFD '' empieza en el byte 6
el carácter U+8A9E '語' empieza en el byte 7
```

Finalmente, Go no tiene el operador coma, además, `++` y `--` son instrucciones, no expresiones. Por lo que si quieres usar múltiples variables en un `for` tendrás que utilizar asignación paralela (a pesar de que excluye a `++` y `--`).

```
// Reverso a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```

Switch

El `switch` de Go es más general que el de C. Las expresiones no es necesario que sean constantes o incluso enteros, los casos se evalúan de arriba hacia abajo hasta encontrar una coincidencia y si el `switch` no tiene una expresión este cambia a `true`. Por lo tanto es posible —e idiomático— escribir una cadena de `if` - `else` - `if` - `else` como un `switch`.

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

No hay compatibilidad automática hacia atrás, pero se pueden presentar casos en listas separadas por comas.

```
func seDebeEscapar(c byte) bool {
    switch c {
    case ' ', '?', '&', '=', '#', '+', '%':
        return true
    }
    return false
}
```

A pesar de que no son tan comunes en Go como en algunos otros lenguajes como C, las declaraciones `break` se pueden usar para terminar anticipadamente un `switch`. A veces, no obstante, es necesario romper un bucle redundante, no el `switch` y en Go se puede cumplimentar poniendo una etiqueta en el bucle y "rompiendo" hacia esa etiqueta. Este ejemplo muestra ambos usos.

```
Repite:
    for n := 0; n < len(src); n += tamaño {
        switch {
        case src[n] < tamañoUno:
            if soloValida {
                break
            }
            tamaño = 1
            actualiza(src[n])
        case src[n] < tamañoDos:
            if n+1 >= len(src) {
                err = errEntradaCorta
                break Repite
            }
            if soloValida {
                break
            }
            tamaño = 2
            actualiza(src[n] + src[n+1]<<shift)
        }
    }
```

Naturalmente, la instrucción `continue` también acepta una etiqueta opcional pero esta solo aplica en bucles.

Para cerrar esta sección, aquí está una rutina de comparación para sectores de byte que usa dos instrucciones `switch`:

```
// Compara devuelve un entero comparando lexicográficamente
// los dos sectores de byte.
// El resultado será 0 si a == b, -1 si a < b y +1 si a > b
func Compara(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }

    switch {
    case len(a) > len(b):
        return 1
    case len(a) < len(b):
        return -1
    }

    return 0
}
```

Switch de tipo

Un `switch` también puede descubrir dinámicamente el tipo de una variable de interfaz. Tal *switch de tipo* utiliza la sintaxis de una aserción de tipo con la palabra clave `type` dentro de los paréntesis. Si el `switch` declara una variable en la expresión, la variable tendrá el tipo correspondiente en cada cláusula. También es idiomático reutilizar el nombre en tales casos, en efecto declarando una nueva variable con el mismo nombre pero un diferente tipo en cada caso.

```
var t interface{}
t = funcionDeAlgunTipo()
switch t := t.(type) {
default:
    fmt.Printf("tipo inesperado %T\n", t)     // imprime %T cuando t tiene tipo
case bool:
    fmt.Printf("lógico %t\n", t)              // t es de tipo bool
case int:
    fmt.Printf("entero %d\n", t)              // t es de tipo int
case *bool:
    fmt.Printf("puntero a lógico %t\n", *t)   // t es de tipo *bool
case *int:
    fmt.Printf("puntero a entero %d\n", *t)   // t es de tipo *int
}
```

Funciones

Retorno de múltiples valores

Una de las características inusuales de Go es que las funciones y los métodos pueden devolver múltiples valores. Esta forma se puede usar para mejorar un par de torpes modismos en programas C: en devoluciones de error en banda tal como `-1` para `EOF` y modificando un argumento pasado por dirección.

En C, un error de escritura es señalado por un contador negativo con el código de error oculto en una ubicación volátil. En Go, `Write` puede devolver un contador *y* un error: “Sí, escribiste algunos bytes pero no todos porque llenaste el dispositivo”. La firma del método `Write` en archivos del paquete `os` es:

```
func (archivo *File) Write(b []byte) (n int, err error)
```

y como dice la documentación, devuelve el número de bytes escritos y un `error` distinto de `nil` cuando `n != len(b)`. Este es un estilo común; ve la sección sobre el manejo de errores para más ejemplos.

Un enfoque similar evita la necesidad de pasar un puntero a un valor de retorno para simular una referencia al parámetro. Aquí tienes una ingenua función para grabar un número desde una posición en el byte de un sector, devolviendo el número y la siguiente posición.

```
func sigEnt(b []byte, i int) (int, int) {
    for ; i < len(b) && !isDigit(b[i]); i++ {
    }
    x := 0
    for ; i < len(b) && isDigit(b[i]); i++ {
        x = x*10 + int(b[i]) - '0'
    }
    return x, i
}
```

La podrías utilizar para escanear los números en un sector de entrada `b` de la siguiente manera:

```
for i := 0; i < len(b); {
    x, i = sigEnt(b, i)
    fmt.Println(x)
}
```

Parámetros de retorno nombrados

Los “parámetros” de retorno o de resultado de una función Go pueden ser nombrados y los puedes utilizar como variables regulares, igual que los parámetros de entrada. Cuando son nombrados, al empezar la función se inician a los valores cero para sus tipos correspondientes; si la función ejecuta una instrucción `return` sin argumentos, los valores actuales de los parámetros de retorno serán utilizados como los valores devueltos.

Los nombres **no** son obligatorios pero pueden hacer que el código sea más corto y más claro: puesto que son documentación. Si nombramos los resultados de `sigEnt` se vuelve obvio cuál `int` devuelto es qué.

```
func sigEnt(b []byte, pos int) (valor, sigPos int) {
```

Debido a que los resultados nombrados se inician y vinculan sin adornos a un retorno, estos pueden simplificar además de clarificar. Aquí tienes una versión de `io.ReadFull` que los utiliza bien:

```
func ReadFull(r Reader, buf []byte) (n int, err error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:]
    }
    return
}
```

Defer

la instrucción `defer` de Go programa una llamada a función (la función *diferida*) para ejecutarla inmediatamente antes de que la función que está ejecutando `defer` regrese. Es una inusual pero eficiente manera para tratar situaciones como cuando existen recursos que se tienen que liberar a toda costa en la cual una función toma otra ruta para regresar. Los ejemplos canónicos son desbloquear un exclusión mutua o cerrar un archivo.

```
// Contenido regresa como cadena lo que contiene el archivo.
func Contenido(nombrearchivo string) (string, error) {
    f, err := os.Open(nombrearchivo)
    if err != nil {
        return "", err
    }

    defer f.Close()  // f.Close se ejecutará cuando haya terminado.
    var resultado []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        resultado = append(resultado, buf[0:n]...) // append se explica más adelante.
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err  // f se cerrará si regresamos aquí.
        }
    }
    return string(resultado), nil // f se cerrará si regresamos aquí.
}
```

Diferir una llamada a una función como `Close` tiene dos ventajas. Primero, garantiza que nunca olvidarás cerrar el archivo, una equivocación que se comete fácilmente si más tarde editas la función para añadir una nueva ruta de retorno. Segundo, significa que el `Close` se sitúa cerca del `Open`, lo cual es mucho más claro que colocarlo al final de la función.

Los argumentos de la función diferida (que incluyen al receptor si la función es un método) se evalúan cuando se ejecuta la función *diferida*, no al invocarla *llamada*. Más allá de evitar preocupaciones sobre el cambio de valores de las variables conforme se ejecuta la función, esto significa que una sola llamada diferida en el sitio puede posponer la ejecución de múltiples funciones. Aquí tienes un tonto ejemplo.

```
for i := 0; i &lt; 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

Las funciones diferidas se ejecutan en orden UEPS (último en entrar, primero en salir), así que este código provocará que se imprima `4 3 2 1 0` cuando la función regrese. Un más plausible ejemplo es una sencilla manera de rastrear la ejecución de una función a través del programa. Podríamos escribir un par de sencillas rutinas de rastreo como estas:

```
func sigue(s string)    { fmt.Println("entrando a:", s) }
func abandona(s string) { fmt.Println("dejando:", s) }
// Úsalas de la siguiente manera:
func a() {
    sigue("a")
    defer abandona("a")
    // hace algo....
}
```

Podemos hacer mejores funciones explotando el hecho de que los argumentos de las funciones diferidas se evalúan cuando se ejecuta el `defer`. La rutina de rastreo puede configurar el argumento de la rutina que deja de rastrear. Este ejemplo:

```
func sigue(s string) string {
    fmt.Println("entrando a:", s)
    return s
}

func abandona(s string) {
    fmt.Println("dejando:", s)
}

func a() {
    defer abandona(sigue("a"))
    fmt.Println("en a")
}

func b() {
    defer abandona(sigue("b"))
    fmt.Println("en b")
    a()
}

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

imprime esto:

```
entrando a: b
en b
entrando a: a
en a
dejando: a
dejando: b
```

Para los programadores acostumbrados a administrar recursos a nivel de bloque de otros lenguajes, `defer` les puede parecer extraño, pero sus más interesantes y potentes aplicaciones vienen precisamente del hecho de que no se basa en bloques sino que está basado en funciones. En la sección sobre `pánico` y `recuperación` veremos otro ejemplo de sus posibilidades.

Datos

Asignación con `new`

Go tiene dos primitivas de asignación, las funciones incorporadas `new` y `make`. Son dos cosas diferentes y aplican a diferentes tipos, lo cual puede ser confuso, pero las reglas son sencillas. Primero hablemos sobre `new`. Es una función incorporada que reserva memoria, pero a diferencia de su homónima en algunos otros lenguajes **no** *inicia* la memoria, solo la pone a *ceros*. Es decir, `new(T)` reserva almacenamiento establecido a cero para un nuevo elemento de tipo `T` y regresa su dirección, un valor de tipo `*T`. En terminología Go, regresa un puntero a un recién alojado valor cero de tipo `T`.

Puesto que la memoria devuelta por `new` se pone a cero, es útil que el valor cero de cada tipo se pueda utilizar sin más iniciación para organizar el diseño de las estructuras de datos. Esto significa para un usuario de la estructura de datos que puede crear una con `new` y conseguir que trabaje correctamente. Por ejemplo, la documentación para `bytes.Buffer` declara que “el valor de cero para `Buffer` es un búfer vacío listo para utilizarlo”. De modo parecido, `sync.Mutex` no tiene un método constructor explícito o `Init`. En cambio, el valor de cero para un `sync.Mutex` está definido que sea un mutex desbloqueado.

El valor cero es una propiedad útil que trabaja transitivamente. Considera esta declaración de tipo.

```
type BúferSincronizado struct {
    cerrado sync.Mutex
    búfer   bytes.Buffer
}
```

Los valores de tipo `BúferSincronizado` también están listos para utilizarse inmediatamente solo reservándolos o declarándolos. En el siguiente fragmento, ambos `p` y `v` trabajarán correctamente sin más complicaciones.

```
p := new(BúferSincronizado)  // tipo *BúferSincronizado
var v BúferSincronizado      // tipo  BúferSincronizado
```

Constructores y literales compuestos

A veces, el valor cero no es lo suficientemente bueno y es necesario un constructor para iniciarlo, como en este ejemplo derivado del paquete `os`.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }

    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
    f.nepipe = 0
    return f
}
```

Hay mucha redundancia allí. Lo podemos simplificar utilizando un *literal compuesto*, es decir, una expresión que crea una nueva instancia cada vez que es evaluada.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }

    f := File{fd, name, nil, 0}
    return &f
}
```

Ten en cuenta que, a diferencia de C, es perfectamente válido regresar la dirección de una variable local; el almacenamiento asociado a la variable sobrevive después de regresar de la función. De hecho, al tomar la dirección de un literal compuesto reserva una instancia fresca cada vez que se evalúa, por lo tanto podemos combinar estas últimas dos líneas.

```
return &File{fd, name, nil, 0}
```

Los campos de un literal compuesto se muestran en orden y todos deben estar presentes. Sin embargo, al etiquetar explícitamente los elementos en pares *campo* `:` *valor*, la iniciación puede aparecer en cualquier orden y los que falten se dejan con sus respectivos valores cero. De este modo podríamos decir:

```
return &File{fd: fd, name: name}
```

Como caso límite, si un literal compuesto no contiene campos en absoluto, este crea un valor cero para el tipo. Las expresiones `new(File)` y `&File{}` son equivalentes.

Los literales compuestos también se pueden crear para arreglos, sectores y mapas, con las etiquetas de campo cómo índices o claves de mapa según corresponda. En estos ejemplos, la iniciación trabaja independientemente de los valores de `Enone`, `Eio` y `Einval`, siempre y cuando sean distintos.

```
a := [...]string   {Enone: "sin error", Eio: "Eio", Einval: "argumento no válido"}
s := []string      {Enone: "sin error", Eio: "Eio", Einval: "argumento no válido"}
m := map[int]string{Enone: "sin error", Eio: "Eio", Einval: "argumento no válido"}
```

Asignación con `make`

Volviendo a la asignación. La función incorporada `make(T, args)` sirve a un propósito diferente de `new(T)`. Esta solamente crea sectores, mapas y canales y regresa un valor de tipo *T* (no **T*) `iniciado` (no `con ceros`). La razón para tal distinción es que estos tres tipos, bajo la cubierta, representan referencias a estructuras de datos que se tienen que iniciar antes de usarlas. Un sector, por ejemplo, es un descriptor de tres elementos que contiene un puntero al dato (dentro de un arreglo), su longitud y capacidad y hasta que esos elementos sean iniciados, el sector es `nil`. Para sectores, mapas y canales, `make` inicia la estructura de datos interna y prepara el valor para usarlo. Por ejemplo:

```
make([]int, 10, 100)
```

reserva memoria para un arreglo de 100 enteros y luego crea una estructura de sector con longitud 10 y una capacidad de 100 apuntando a los primeros 10 elementos del arreglo. (Cuando creas un sector, la capacidad se puede omitir; ve la sección sobre sectores para más información). En contraste, `new([]int)` regresa un puntero a una, recién asignada, estructura de sectores iniciada en ceros, es decir, un puntero a un sector con valor `nil`.

Estos ejemplos ilustran la diferencia entre `new` y `make`.

```
var p *[]int = new([]int)       // reserva la estructura del sector; *p == nil; raramente
var v  []int = make([]int, 100) // el sector v ahora se refiere a un nuevo arreglo de 100
// Innecesariamente complejo:
var p *[]int = new([]int)
*p = make([]int, 100, 100)
// Idiomático:
v := make([]int, 100)
```

Recuerda que `make` solo aplica a mapas, sectores y canales, además de que no regresa un puntero. Para obtener un puntero explícito asígnalo con `new` o toma la dirección de una variable explícitamente.

Arreglos

Los arreglos son útiles cuándo planeas detallados esquemas de memoria y a veces, pueden ayudar a evitar la asignación, pero principalmente son un bloque de construcción para sectores, el tema de la siguiente sección. Para preparar los fundamentos de ese tema, aquí tienes unas cuantas palabras sobre arreglos.

Hay importantes diferencias entre la manera en que trabajan los arreglos de Go a como lo hacen en C. En Go:

- Los arreglos son valores. Al asignar un arreglo a otro se copian todos los elementos.
- En particular, si pasas un arreglo a una función, esta recibe una *copia* del arreglo, no un puntero a él.
- El tamaño de un arreglo es parte de su tipo. Los tipos `[10]int` y `[20]int` son distintos.

La propiedad `valor` puede ser útil pero también muy cara; si quieres el comportamiento y eficiencia de C, puedes pasar un puntero al arreglo.

```
func Suma(a *[3]float64) (suma float64) {
    for _, v := range *a {
        suma += v
    }
    return
}

arreglo := [...]float64{7.0, 8.5, 9.1}
x := Suma(&arreglo)  // Observa el operador de dirección explícito
```

Pero incluso este estilo no es idiomático de Go. En su lugar usa sectores.

Sectores

Los sectores envuelven arreglos para dotarlos de una interfaz más general, potente y conveniente para secuencias de datos. Salvo los elementos con dimensión explícita tal como arreglos de transformación, la mayoría de la programación de arreglos en Go está hecha con sectores en lugar de arreglos sencillos.

Los sectores mantienen referencias a un arreglo subyacente y si asignas un sector a otro, ambos se refieren al mismo arreglo. Si una función toma un sector como argumento, los cambios que hace a los elementos del sector serán visibles al llamador, análogo a pasar un puntero al arreglo subyacente. Una función `Read` por lo tanto puede aceptar un sector como argumento en lugar de un puntero y un contador; la longitud dentro del sector impone un límite máximo de cuantos datos leer. Aquí está la firma del método `Read` del tipo `File` en el paquete `os`:

```
func (file *File) Read(buf []byte) (n int, err error)
```

El método regresa el número de bytes leídos y un valor de error, si lo hubiera. Para leer los primeros 32 bytes de un búfer `buf` más grande, el *sector* búfer (aquí utilizado como verbo).

```
n, err := f.Read(buf[0:32])
```

Tal sector es común y eficiente. De hecho, dejando aparte —por el momento— la eficiencia, el siguiente fragmento también lee los primeros 32 bytes del búfer.

```
var n int
var err error
for i := 0; i < 32; i++ {
    nbytes, e := f.Read(buf[i:i+1])  // Lee un byte.
    if nbytes == 0 || e != nil {
        err = e
        break
    }
    n += nbytes
}
```

La longitud de un sector se puede cambiar mientras todavía quepa dentro de los límites del arreglo subyacente; justo asignándolo a un sector de sí mismo. La *capacidad* de un sector, es accesible por medio de la función incorporada `cap`, esta informa la longitud máxima que el sector puede adoptar. Aquí está una función para añadir datos a un sector. Si el dato supera la capacidad, el sector es reasignado. El sector resultante es devuelto. La función utiliza el hecho que `len` y `cap` son legales cuando se aplican al sector `nil` y regresa 0.

```
func Append(sector, datos[]byte) []byte {
    l := len(sector)
    if l + len(datos) > cap(sector) {  // reasignado
        // Asigna el doble de lo necesario, para futuro crecimiento.
        nuevoSector := make([]byte, (l+len(datos))*2)
        // La función copy está predeclarada y trabaja para cualquier tipo de sector.
        copy(nuevoSector, sector)
        sector = nuevoSector
    }

    sector = sector[0:l+len(datos)]
    for i, c := range datos {
        sector[l+i] = c
    }
    return sector
}
```

Tenemos que regresar al sector más adelante porque, a pesar de que `Append` puede modificar los elementos del `sector`, el sector en sí mismo (la estructura de datos en tiempo de ejecución que contiene el puntero, longitud y capacidad) es pasado por valor.

La idea de añadir a un sector es tan útil que está capturada por la función incorporada `append`. Para entender el diseño de esa función, sin embargo, necesitamos un poco más de información, así que regresaremos a ella más adelante.

Sectores bidimensionales

En Go los arreglos y sectores son unidimensionales. Para crear el equivalente de un arreglo o sector 2D, es necesario definir un arreglo de arreglos o un sector de sectores, de la siguiente manera:

```
type Transformación [3][3]float64  // Un arreglo 3x3, en realidad un arreglo de arreglos.
type LíneasDeTexto [][]byte        // Un sector de sectores de byte.
```

Debido a que los sectores son arreglos de cierta longitud, es posible hacer que cada sector interior sea de una longitud diferente. Esta puede ser una situación común, como en nuestro ejemplo de `LíneasDeTexto`: cada línea tiene una longitud independiente.

```
texto := LíneasDeTexto{
    []byte("Ahora es el momento"),
    []byte("Para que todos los buenos gophers"),
    []byte("traigan un poco de diversión a la fiesta."),
}
```

A veces, es necesario reservar un sector 2D, una situación que puede surgir al procesar líneas de píxeles escaneados, por ejemplo. Hay dos maneras en que lo podemos lograr. Una es alojar cada sector independientemente; la otra es alojar un solo arreglo y apuntar a los sectores individuales en él. Cuál usar depende de tu aplicación. Si los sectores pueden crecer o reducirse, los deberías alojar independientemente para evitar sobrescribir la siguiente línea; si no, posiblemente sea más eficiente construir el objeto con una sola asignación. Para referencia, aquí tienes un croquis de los dos métodos. Primero, una línea a la vez:

```
// Asigna el sector de nivel superior.
foto := make([][]uint8, altura) // Un renglón por unidad de y.
// Recorre los renglones, alojando el sector de cada renglón.
for i := range foto {
    foto[i] = make([]uint8, ancho)
}
```

Y ahora como una asignación, dividido en líneas:

```
// Asigna el sector de nivel superior, igual a cómo lo hicimos antes.
foto := make([][]uint8, altura) // Un renglón por unidad de y.
// Reserva un gran sector que contiene todos los píxeles.
pixeles := make([]uint8, ancho*altura) // Tiene tipo []uint8 incluso aunque foto es [][]u
// Recorre las filas, dividiendo cada fila desde el frente del sector a los píxeles resta
for i := range foto {
    foto[i], pixeles = pixeles[:ancho], pixeles[ancho:]
}
```

Mapas

Los mapas son una conveniente y potente estructura de datos incorporada que asocia los valores de un tipo (la *clave*) con valores de otro tipo (el *elemento* o *valor*). La clave puede ser de cualquier tipo para el cual el operador de igualdad esté definido, tal como enteros, números de coma flotante y complejos, cadenas, punteros, interfaces (siempre y cuando la igualdad apoye el tipo dinámico), estructuras y arreglos. Los sectores no se pueden utilizar como claves de mapa, porque la igualdad no está definida en ellos. Como los sectores, los mapas contienen referencias a una estructura de datos subyacente. Si pasas un mapa a una función que cambia el contenido del mapa, los cambios serán visibles en el llamador.

Los mapas se pueden construir utilizando la sintaxis habitual del literal compuesto con pares clave-valor separados por comas, por tanto es fácil iniciarlos durante su construcción.

```
var zonaHoraria = map[string] int {
    "UTC":  0*60*60,
    "EST": -5*60*60,
    "CST": -6*60*60,
    "MST": -7*60*60,
    "PST": -8*60*60,
}
```

Asignar y recuperar valores del mapa se ve sintácticamente igual como se hace para los arreglos y sectores salvo que no es necesario que el índice sea un entero.

```
offset := zonaHoraria["EST"]
```

Un intento para recuperar un valor del mapa con una clave que no esté presente en él regresará el valor cero para el tipo de entrada en el mapa. Por ejemplo, si el mapa contiene enteros, al buscar una clave inexistente regresará `0`. Un conjunto se puede implementar como mapa con valor de tipo `lógico`. Pon la entrada del mapa a `true` para colocar el valor en el conjunto y entonces pruébalo por medio de indexación sencilla.

```
asistio := map[string]bool{
    "Ana": true,
    "José": true,
    ...
}

if asistio[persona] {  // será false si persona no está en el mapa
    fmt.Println(persona, "estaba en la reunión")
}
```

A veces necesitas distinguir entre una entrada ausente y un valor cero. ¿Hay una entrada para `&quot;UTC&quot;` o la cadena está vacía porque no está en el mapa en absoluto? Lo puedes diferenciar con una forma de asignación múltiple.

```
var segundos int
var ok bool
segundos, ok = zonaHoraria[zh]
```

Por obvias razones a esto se le conoce como el modismo de “coma ok”. En este ejemplo, si `zh` está presente, los `segundos` serán ajustados apropiadamente y `ok` será cierto; si no, `segundos` se pondrá a cero y `ok` será falso. Aquí está una función que junta todo con un buen informe de error:

```
func compensa(zh string) int {
    if segundos, ok := zonaHoraria[zh]; ok {
        return segundos
    }

    log.Println("zona horaria desconocida:", zh)
    return 0
}
```

Para probar la presencia en el mapa sin preocuparte del valor real, puedes utilizar el identificador blanco (`_`) en lugar de la variable habitual para el valor.

```
_, presente := zonaHoraria[zh]
```

Para eliminar una entrada del mapa, usa la función incorporada `delete` , cuyos argumentos son el mapa y la clave a eliminar. Es seguro usarla incluso si la clave ya no existe en el mapa.

```
delete(zonaHoraria, "PDT")  // Ahora en tiempo estándar
```

Impresión

La impresión formateada en Go usa un estilo similar al de la familia `printf` de C pero es más rica y más general. Estas funciones viven en el paquete `fmt` y tienen nombres capitalizados: `fmt.Printf` , `fmt.Fprintf` , `fmt.Sprintf` y así por el estilo. Las funciones de cadena (`Sprintf` , etc.) regresan una cadena en lugar de rellenar el búfer proporcionado.

No necesitas proporcionar una cadena de formato. Por cada `Printf` , `Fprintf` y `Sprintf` hay otro par de funciones, por ejemplo `Print` y `Println` . Estas funciones no toman una cadena de formato pero en cambio generan un formato predefinido para cada argumento. Las versiones `Println` también insertan un espacio entre argumentos y añaden un nuevo salto de línea al resultado mientras que las versiones `Print` solo añaden espacios si el operando en alguno de los lados es una cadena. En este ejemplo cada línea produce el mismo resultado.

```
fmt.Printf("Hola %d\n", 23)
fmt.Fprint(os.Stdout, "Hola ", 23, "\n")
fmt.Println("Hola", 23)
fmt.Println(fmt.Sprint("Hola ", 23))
```

Las funciones de impresión formateada `fmt.Fprint` y amigas toman como primer argumento cualquier objeto que implemente la interfaz `io.Writer`; las variables `os.Stdout` y `os.Stderr` son instancias familiares.

Aquí empiezan a divergir las cosas de C. Primero, los formatos numéricos tal como `%d` no toman banderas de signo o tamaño; en cambio, las rutinas de impresión utilizan el tipo del argumento para decidir estas propiedades.

```
var x uint64 = 1<<64 - 1
fmt.Printf("%d %x; %d %x\n", x, x, int64(x), int64(x))
```

imprime:

```
18446744073709551615 ffffffffffffffff; -1 -1
```

Si solo quieres la conversión predefinida, tal como decimal a enteros, puedes utilizar el formato multipropósito `%v` (por “valor”); el resultado es exactamente el que producirían `Print` y `Println`. Además, ese formato puede imprimir *cualquier* valor, incluso arreglos, sectores, estructuras y mapas. Aquí tienes una declaración de impresión para el mapa de husos horarios definido en la sección anterior.

```
fmt.Printf("%v\n", zonaHoraria)  // o solo fmt.Println(zonaHoraria)
```

La cuál produce:

```
map[CST:-21600 PST:-28800 EST:-18000 UTC:0 MST:-25200]
```

En los mapas, las claves se pueden organizar en cualquier orden, por supuesto. Cuando imprimas una estructura, el formato modificado `%+v` anota los campos de la estructura con sus nombres y para cualquier valor el formato alterno `%#v` imprime el valor en sintaxis Go completa.

```
type T struct {
    a int
    b float64
    c string
}

t := &T{ 7, -2.35, "abc\tdef" }

fmt.Printf("%v\n", t)
fmt.Printf("%+v\n", t)
fmt.Printf("%#v\n", t)
fmt.Printf("%#v\n", zonaHoraria)
```

imprime:

```
&{7 -2.35 abc   def}
&{a:7 b:-2.35 c:abc     def}
&main.T{a:7, b:-2.35, c:"abc\tdef"}
map[string] int{"CST":-21600, "PST":-28800, "EST":-18000, "UTC":0, "MST":-25200}
```

(Toma en cuenta los ampersands). La cadena de formato entrecomillada también está disponible con `%q` cuando se aplica a un valor de tipo `string` o `[]byte`. El formato alterno `%#q` en su lugar utilizará comillas inversas si es posible. (El formato `%q` también aplica a enteros y runes, produciendo una sola constante rune entrecomillada). Además, `%x` trabaja en cadenas, arreglos de bytes y sectores de bytes así como en enteros, generando una larga cadena hexadecimal y, con un espacio en el formato (`% x`) pone espacios entre los bytes.

Otro útil formato es `%T`, el cual imprime el *tipo* de un valor.

```
fmt.Printf("%T\n", zonaHoraria)
```

imprime:

```
map[string] int
```

Si quieres controlar el formato predefinido para un tipo personalizado, todo lo que se requiere es definir un método con la firma `String() string` en el tipo. Para nuestro sencillo tipo `T`, este podría tener esta apariencia.

```
func (t *T) String() string {
    return fmt.Sprintf("%d/%g/%q", t.a, t.b, t.c)
}

fmt.Printf("%v\n", t)
```

para imprimir en el formato:

```
7/-2.35/"abc\tdef"
```

(Si necesitas imprimir *valores* de tipo `T` así como punteros a `*T` , el receptor para `String` tiene que ser de tipo valor; este ejemplo utiliza un puntero porque es más eficiente e idiomático para tipos estructura. Ve la sección punteros versus receptores de valor para más información).

Nuestro método `String` es capaz de llamar a `Sprintf` porque las rutinas de impresión son completamente reentrantes y se pueden envolver de este modo. Sin embargo, hay un importante detalle para entender este enfoque: no construyas un método `String` que llame a `Sprintf` en una manera que se repita indefinidamente tu método `String` . Esto puede ocurrir si `Sprintf` intenta llamar a `print` para imprimir el receptor como cadena directamente, el cual a su vez invocará al método de nuevo. Es una fácil y común equivocación, tal como muestra este ejemplo.

```
type MiCadena string
func (m MiCadena) String() string {
    return fmt.Sprintf("MiCadena=%s", m) // Error: recurrirá indefinidamente.
}
```

También es fácil corregirlo: convierte el argumento al tipo string básico, el cual no tiene este método.

```
type MiCadena string

func (m MiCadena) String() string {
    return fmt.Sprintf("MiCadena=%s", string(m)) // BIEN: observa la conversión.
}
```

En la sección de iniciación veremos otra técnica que evita esta recursión.

Otra técnica de impresión es pasar los argumentos de una rutina de impresión directamente a otra rutina. La firma de `Printf` usa el tipo `...interface{}` en su argumento final para especificar que después del formato puede aparecer una cantidad arbitraria de parámetros (de cualquier tipo).

```
func Printf(format string, v ...interface{}) (n int, err error) {
```

Dentro de la función `Printf`, `v` actúa como una variable de tipo `[]interface{}` pero si esta se pasa a otra función con número de argumentos variable, actúa como una lista de argumentos regular. Aquí está la implementación de la función `log.Println` que utilizamos arriba. Esta pasa sus argumentos directamente a `fmt.Sprintln` para el formateo real.

```
// Println imprime al registro estándar a la manera de fmt.Println.
func Println(v ...interface{}) {
    std.Output(2, fmt.Sprintln(v...))  // Output toma los parámetros (int, string)
}
```

Escribimos `...` después de `v` en la llamada anidada a `Sprintln` para decir al compilador que trate a `v` como una lista de argumentos; de lo contrario solo pasa `v` como un solo argumento sector.

Incluso hay mucho más para imprimir de lo que hemos cubierto aquí. Ve la documentación `godoc` del paquete `fmt` para más detalles.

Por cierto, un parámetro `...` puede ser de un tipo específico, por ejemplo `...int` para una función `Min` que escoge el menor de una lista de enteros:

```
func Min(a ...int) int {
    min := int(^uint(0) >> 1)  // el int más grande
    for _, i := range a {
        if i < min {
            min = i
        }
    }
    return min
}
```

Append

Ahora tenemos la pieza perdida que necesitábamos para explicar el diseño de la función incorporada `append`. La firma de `append` es diferente de nuestra anterior función `Append` personalizada. Esquemáticamente, es así:

```
func append(sector []_T_, elementos ..._T_) []_T_
```

Dónde *T* es un marcador de posición para cualquier tipo dado. De hecho, en Go no puedes escribir una función donde el tipo `T` esté determinado por el llamador. Es por eso que `append` está incorporada: necesita respaldo del compilador.

Lo que hace `append` es añadir los elementos hasta el final del sector y regresar el resultado. Es necesario devolver el resutado porque, como con nuestra `Append` personalizada, el arreglo subyacente puede cambiar. Este sencillo ejemplo

```
x := []int{1,2,3}
x = append(x, 4, 5, 6)
fmt.Println(x)
```

imprime `[1 2 3 4 5 6]`. Así que `append` trabaja un poco como `Printf`, recolectando una arbitraria cantidad de argumentos.

Pero, ¿qué pasa si quisiéramos hacer lo que hace nuestra `Append` y anexar un sector a un sector? Fácil: usa `...` en el sitio de la llamada, tal como lo hicimos en la llamada a `Output` arriba. Este fragmento produce idéntico resultado que el anterior.

```
x := []int{1,2,3}
y := []int{4,5,6}
x = append(x, y...)
fmt.Println(x)
```

Sin esos `...`, no compila porque los tipos serían incorrectos; `y` no es de tipo `int`.

Iniciación

A pesar de que superficialmente la iniciación no se ve muy diferente en C o C++, la iniciación en Go es mucho más potente. Puedes construir estructuras complejas durante la iniciación y los problemas de ordenación entre objetos iniciados, incluso entre diferentes paquetes, se manejan correctamente.

Constantes

Las constantes en Go solo son eso —constantes. Estás se crean en tiempo de compilación, incluso cuando se definan como locales en funciones y solo pueden ser números, caracteres (runes), cadenas o lógicas. Debido a la restricción de tiempo de compilación, las expresiones que las definen tienen que ser expresiones constantes, evaluables por el compilador. Por ejemplo, `1&lt;&lt;3` es una expresión constante, mientras que `math.Sin(math.Pi/4)` no lo es porque la llamada a la función `math.Sin` necesita ocurrir en tiempo de ejecución.

En Go, las constantes enumeradas se crean utilizando el enumerador `iota`. Debido a que `iota` puede ser parte de una expresión y las expresiones se pueden repetir implícitamente es fácil construir intrincados conjuntos de valores.

```
type MagnitudByte float64

const (
    // descarta el primer valor asignándolo al identificador blanco
    _    = iota
    KB MagnitudByte = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

La habilidad de agregar un método tal como `String` a cualquier tipo definido por el usuario hace posible formatear automáticamente valores arbitrarios para impresión. A pesar de que a menudo verás aplicada esta técnica a estructuras, también es útil para tipos escalares como tipos de coma flotante tal como `MagnitudByte`.

```
func (b MagnitudByte) String() string {
    switch {
    case b >= YB:
        return fmt.Sprintf("%.2fYB", b/YB)
    case b >= ZB:
        return fmt.Sprintf("%.2fZB", b/ZB)
    case b >= EB:
        return fmt.Sprintf("%.2fEB", b/EB)
    case b >= PB:
        return fmt.Sprintf("%.2fPB", b/PB)
    case b >= TB:
        return fmt.Sprintf("%.2fTB", b/TB)
    case b >= GB:
        return fmt.Sprintf("%.2fGB", b/GB)
    case b >= MB:
        return fmt.Sprintf("%.2fMB", b/MB)
    case b >= KB:
        return fmt.Sprintf("%.2fKB", b/KB)
    }
    return fmt.Sprintf("%.2fB", b)
}
```

La expresión `YB` imprime `1.00YB`, mientras que `MagnitudByte(1e13)` imprime `9.09TB`.

Aquí el uso de `Sprintf` para implementar el método `String` en `MagnitudByte` es seguro (evita recurrir indefinidamente) no debido a una conversión sino a que se llama a `Sprintf` con `%f`, que no es una cadena de formato: `Sprintf` solo llamará al método `String` cuándo quiere una cadena y `%f` requiere un valor de coma flotante.

Variables

Las variables se pueden iniciar justo como las constantes pero el iniciador puede ser una expresión general calculada en tiempo de ejecución.

```
var (
    home   = os.Getenv("HOME")
    user   = os.Getenv("USER")
    gopath = os.Getenv("GOPATH")
)
```

La función init

Eventualmente, cada archivo fuente puede definir su propia *nil*adica función `init` para configurar cualquier estado requerido. (De hecho, cada archivo puede tener múltiples funciones `init`). Y finalmente significa que: `init` se llama después de que se han evaluado todas las declaraciones de variables en el paquete y estas son evaluadas solo después de iniciar todos los paquetes importados.

Más allá de las iniciaciones que no se pueden expresar como declaraciones, un uso común de las funciones `init` es para verificar o reparar opciones de estado del programa antes de comenzar su ejecución.

```
func init() {
    if usuario == "" {
        log.Fatal("$USER no configurado")
    }

    if home == "" {
        home = "/home/" + user
    }

    if gopath == "" {
        gopath = home + "/go"
    }

    // gopath se pudo haber reemplazado por la bandera --gopath en la línea de órdenes.
    flag.StringVar(&amp;gopath, "gopath", gopath, "sustituye al GOPATH predefinido")
}
```

Métodos

Punteros versus valores

Como vimos con `MagnitudByte`, los métodos se pueden definir para cualquier tipo nombrado (excepto un puntero o una interfaz); el receptor no tiene que ser una estructura.

Anteriormente, en la explicación de sectores, escribimos una función `Append`. En su lugar, la podemos definir como un método en sectores. Para ello, primero declaramos un tipo nombrado al cual podamos vincular el método y entonces hacemos que el receptor del método tenga un valor de ese tipo.

```
type SectorByte []byte

func (sector SectorByte) Append(datos []byte) []byte {
    // Cuerpo exactamente igual que arriba
}
```

Esto todavía requiere que el método devuelva el sector actualizado. Podemos eliminar esa torpeza redefiniendo el método para que tome un *puntero* a un `SectorByte` como su receptor, por tanto el método puede sobrescribir el sector llamador.

```
func (p *SectorByte) Append(datos []byte) {
    sector := *p
    // Cuerpo cómo el de arriba, sin el return.
    *p = sector
}
```

De hecho, incluso lo podemos hacer mejor. Si modificamos nuestra función para que se parezca al método `Write` estándar, de la siguiente manera:

```
func (p *SectorByte) Write(datos []byte) (n int, err error) {
    sector := *p
    // Una vez más como arriba.
    *p = sector
    return len(datos), nil
}
```

Entonces el tipo `*SectorByte` satisface la interfaz del `io.Write` estándar, lo cual es práctico. Por ejemplo, podemos imprimir en uno.

```
var b SectorByte
fmt.Fprintf(&amp;b, "Esta hora tiene %d días\n", 7)
```

Pasamos la dirección de un `SectorByte` porque únicamente un `*SectorByte` satisface al `io.Writer`. La regla sobre punteros versus valores para receptores es que los valores de los métodos se puedan invocar en punteros y valores, pero los métodos del puntero solo se pueden invocar en punteros.

Esta regla surge porque los métodos del puntero pueden modificar el receptor; invocándolos con un valor causaría que el método reciba una copia del valor, así que cualquier modificación sería desechada. El lenguaje por tanto rechaza esta equivocación. No obstante, hay una útil excepción. Cuando el valor es direccionable, el lenguaje cuida del caso común de invocar un método de puntero en un valor insertando automáticamente el operador de dirección. En nuestro ejemplo, la variable `b` es direccionable, así que podemos llamar a su método `Write` justo con `b.Write`. El compilador lo reescribirá por nosotros como `(&amp;b).Write`.

Por cierto, la idea de utilizar `Write` en un sector de bytes es central para la implementación de `bytes.Buffer`.

Interfaces y otros tipos

Interfaces

Las interfaces en Go proporcionan una manera de especificar el comportamiento de un objeto: si algo puede hacer *este*, entonces se puede utilizar *aquí*. Ya hemos visto un par de ejemplos sencillos; las impresiones personalizadas se pueden implementar con un método `String` mientras que `Fprintf` puede generar su salida a cualquier cosa con un método `Write`. Las interfaces con únicamente uno o dos métodos son comunes en código Go y normalmente se les otorga un nombre derivado del método, tal como `io.Writer` para algo que implementa `Write`.

Un tipo puede implementar múltiples interfaces. Por ejemplo, puedes ordenar una colección con las rutinas del paquete `sort` si este implementa la `sort.Interface`, la cual contiene `Len()`, `Less(i, j int) bool` y `Swap(i, j int)`, además puede tener un formateador personalizado. En este ejemplo inventado `Secuencia` satisface ambos.

```
type Secuencia []int

// Métodos requeridos por sort.Interface.
func (s Secuencia) Len() int {
    return len(s)
}

func (s Secuencia) Less(i, j int) bool {
    return s[i] < s[j]
}

func (s Secuencia) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

// Método para impresión - ordena los elementos antes de imprimir.
func (s Secuencia) String() string {
    sort.Sort(s)
    cadena := "["
    for i, elem := range s {
        if i > 0 {
            cadena += " "
        }
        cadena += fmt.Sprint(elem)
    }
    return cadena + "]"
}
```

Conversiones

El método `String` de `Secuencia` está recreando el trabajo que `Sprint` ya hace para sectores. Podemos compartir el esfuerzo si convertimos la `Secuencia` a un sencillo `[]int` antes de llamar a `Sprint` .

```
func (s Secuencia) String() string {
    sort.Sort(s)
    return fmt.Sprint([]int(s))
}
```

Este método es otro ejemplo de la técnica de conversión para llamar sin peligro a `Sprintf` desde un método `String` . Debido a que los dos tipos (`Secuencia` e `[]int`) son el mismo si ignoramos el nombre de tipo, es legal la conversión entre ellos. La conversión no crea un nuevo valor, solo actúa temporalmente como si el valor existente tuviera un nuevo tipo. (Hay otras conversiones legales, tal como de entero a coma flotante, esta sí crea un nuevo valor).

Es un modismo en programas Go convertir el tipo de una expresión para acceder a un diferente conjunto de métodos. Como ejemplo, podríamos utilizar el tipo existente `sort.IntSlice` para reducir el ejemplo entero a esto:

```
type Secuencia []int
// Método para impresión - ordena los elementos antes de imprimirlos.
func (s Secuencia) String() string {
    sort.IntSlice(s).Sort()
    return fmt.Sprint([]int(s))
}
```

Ahora, en vez de dejar que `Secuencia` implemente múltiples interfaces (ordenando e imprimiendo), utilizamos la habilidad de un elemento del dato para convertirlo a múltiples tipos (`Secuencia` , `sort.IntSlice` e `[]int`), cada uno de los cuales hace alguna parte del trabajo. Esto es muy inusual en la práctica pero puede ser eficiente.

Conversión de interfaz y aserción de tipo

Los switch de tipo son una forma de conversión: toman una interfaz y, para cada caso en el switch, en un sentido lo convierte al tipo de ese caso. Aquí está una versión simplificada de cómo el código bajo `fmt.Printf` cambia un valor a una cadena utilizando un switch de tipo. Si ya es una cadena, queremos el valor real de la cadena que tiene la interfaz, aunque esta tiene un método `String` queremos el resultado de llamar al método.

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

var valor interface{} // Valor proporcionado por el llamador.

switch cadena := valor.(type) {
case string:
    return cadena
case Stringer:
    return cadena.String()
}
```

El primer caso encuentra un valor concreto; el segundo convierte la interfaz a otra interfaz. Es perfectamente legal mezclar tipos de este modo.

¿Qué pasa si solo nos interesa un tipo? ¿Si sabemos que el valor tiene una `cadena` y únicamente la queremos extraer? Un caso del switch de tipo lo haría, pero también lo haría una *aserción de tipo*. Una aserción de tipo toma un valor interfaz y extrae de él un valor del

tipo especificado explícitamente. La sintaxis toma prestado de la cláusula de apertura un switch de tipo, pero con un tipo explícito en lugar de la palabra clave `type`:

```
valor.(nombreDeTipo)
```

y el resultado es un nuevo valor con el tipo estático `nombreDeTipo`. Ese tipo tampoco tiene que ser el tipo concreto soportado por la interfaz, o un segundo tipo interfaz en que el valor se pueda convertir. Para extraer la cadena sabemos que está en el valor, podríamos escribir:

```
cadena := valor.(string)
```

Pero, si resulta que el valor no contiene una cadena, el programa se detendrá con un error en tiempo de ejecución. Para resguardarte contra eso, usa el modismo "coma, ok" para probar, sin peligro, si el valor es una cadena:

```
cadena, ok := valor.(string)
if ok {
    fmt.Printf("el valor de la cadena es: %q\n", cadena)
} else {
    fmt.Printf("el valor no es una cadena\n")
}
```

Si la aserción de tipo falla, `cadena` todavía existe y sera de tipo cadena, pero esta tendrá el valor cero, una cadena vacía.

Como ilustración de su capacidad, aquí tienes una instrucción `if` - `else` equivalente al switch de tipo que abrió esta sección.

```
if cadena, ok := valor.(string); ok {
    return cadena
} else if cadena, ok := valor.(Stringer); ok {
    return cadena.String()
}
```

Generalidad

Si un tipo existe solo para implementar una interfaz y no tiene ningún método exportado además de la interfaz, no hay ninguna necesidad de exportar el tipo en sí mismo. Exportar solo la interfaz aclara que es el comportamiento el que importa, no la implementación y que

otras implementaciones con diferentes propiedades pueden reflejar el comportamiento del tipo original. También evita la necesidad de repetir la documentación en cada copia de un método común.

En tales casos, el constructor tendría que devolver un valor de interfaz en lugar del tipo implementado. Por ejemplo, en la bibliotecas `hash` tanto `crc32.NewIEEE` como `adler32.New` devuelven el tipo interfaz `hash.Hash32`. Sustituir el algoritmo `CRC-32` por `Adler-32` en un programa Go solo requiere cambiar la llamada al constructor; el resto del código no es afectado por el cambio en el algoritmo.

Un enfoque similar permite que los algoritmos de transmisión cifrada en varios paquetes `crypto` estén separados del bloque de cifradores encadenados. La interfaz `Block` en el paquete `crypto/cipher` especifica el comportamiento de un bloque cifrado, el cual proporciona encriptación de un solo bloque de datos. Entonces, por analogía con el paquete `bufio`, los paquetes de cifrado que implementan esta interfaz suelen poder construir cifradores de transmisión, representados por la interfaz `Stream`, sin conocer los detalles del bloque de encriptación.

Las interfaces `crypto/cipher` tienen la siguiente apariencia:

```
type Block interface {
    BlockSize() int
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
}

type Stream interface {
    XORKeyStream(dst, src []byte)
}
```

Aquí está la definición del modo contador (CTR), el cual cambia un bloque cifrado a un flujo cifrado; observa que se abstraen los detalles del bloque cifrado:

```
// NewCTR devuelve un Stream que cifra/descifra usando el Bloque dado
// en el modo contador. La longitud de iv tiene que ser igual al
// tamaño del bloque.
func NewCTR(block Block, iv []byte) Stream
```

`NewCTR` no solo aplica a un algoritmo de encriptación y fuente de datos específicos sino a cualquier implementación de la interfaz `Block` y a cualquier `Stream`. Debido a que regresa valores de la interfaz, reemplazando la encriptación CTR con otros modos de encriptación es un cambio localizado. Las llamadas al constructor se tienen que editar, pero porque el código circundante tiene que tratar el resultado solo como un `Stream`, no se nota la diferencia.

Interfaces y métodos

Debido a que casi cualquier cosa puede tener métodos adjuntos, casi cualquier cosa puede satisfacer una interfaz. Un ejemplo ilustrativo está en el paquete `http`, el cual define la interfaz `Handler`. Cualquier objeto que implemente `Handler` puede servir peticiones HTTP.

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

`ResponseWriter` en sí mismo es una interfaz que proporciona acceso a los métodos necesarios para devolver la respuesta al cliente. Estos métodos incluyen el método `Write` estándar, por lo tanto `http.ResponseWriter` puede utilizar cualquier `io.Writer`. `Request` es una estructura que contiene una representación de la petición del cliente analizada sintácticamente.

Por brevedad, ignoraremos las peticiones POST y asumiremos que las peticiones HTTP siempre son GET; esta simplificación no afecta la manera en que se configuran los controladores. Aquí tienes una trivial pero completa implementación de un controlador para contar las veces que la página es visitada.

```
// Sencillo contador del servidor.
type Contador struct {
    n int
}

func (cnt *Contador) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    cnt.n++
    fmt.Fprintf(w, "contador = %d\n", cnt.n)
}
```

(Manteniendo nuestro tema, observa cómo `Fprintf` puede imprimir a un `http.ResponseWriter`). Para referencia, aquí tienes cómo asociar tal servidor a un nodo en el árbol URL.

```
import "net/http"
...
cnt := new(Contador)
http.Handle("/contador", cnt)
```

Pero, ¿por qué hacer que `Contador` sea una estructura? Todo lo que necesitamos es un entero. (El receptor necesita ser un puntero para que el incremento sea visible al llamador).

```
// Sencillo contador del servidor.
type Contador int
func (cnt *Contador) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    *cnt++
    fmt.Fprintf(w, "contador = %d\n", *cnt)
}
```

¿Qué pasa si tu programa tiene algún estado interno que sea necesario notificar cuándo se haya visitado una página? Vincula un canal a la página web.

```
// Un canal que envía una notificación en cada visita.
// (Probablemente quieras que el canal se mentenga en disco).
type Chan chan *http.Request
func (ch Chan) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ch <- req
    fmt.Fprint(w, "notificación enviada")
}
```

Finalmente, digamos que deseas presentar los argumentos `/args` utilizados cuando invocaste el binario del servidor. Es fácil escribir una función para imprimir los argumentos.

```
func ArgServer() {
    fmt.Println(os.Args)
}
```

¿Cómo convertimos esto en un servidor HTTP? Podríamos hacer de `ArgServer` un método de algún tipo cuyo valor descartemos, pero hay una manera más limpia. Puesto que podemos definir un método para cualquier tipo excepto punteros e interfaces, podemos escribir un método para una función. El paquete `http` contiene este código:

```
// El tipo HandlerFunc es un adaptador para permitir el uso de
// funciones normales como controladores HTTP.  Si f es una función
// con la firma apropiada, HandlerFunc(f) es un
// objeto controlador que llama a f.
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP llama a f(c, req).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, req *Request) {
    f(w, req)
}
```

`HandlerFunc` es un tipo con un método, `ServeHTTP`, por lo tanto los valores de ese tipo pueden servir peticiones HTTP. Observa la implementación del método: el receptor es una función, `f` y el método llama a `f`. Esto puede parecer extraño pero no tan diferente,

digamos que, el receptor es un canal y enviamos el método en el canal.

Para hacer de `ArgServer` un servidor HTTP, primero lo modificamos para que tenga la firma correcta.

```
// Argumento servidor.
func ArgServer(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, os.Args)
}
```

`ArgServer` ahora tiene la misma firma que `HandlerFunc`, por lo tanto se puede convertir a ese tipo para acceder a sus métodos, tal como convertimos `Secuencia` a `IntSlice` para acceder a `IntSlice.Sort`. El código para configurarlo es conciso:

```
http.Handle("/args", http.HandlerFunc(ArgServer))
```

Cuando alguien visita la página `/args`, el controlador instalado en esa página tiene el valor `ArgServer` y el tipo `HandlerFunc`. El servidor HTTP invocará al método `ServeHTTP` de ese tipo, con `ArgServer` como el receptor, el cual en su momento llama a `ArgServer` (vía la invocación a `f(c, req)` dentro de `HandlerFunc.ServeHTTP`). Entonces los argumentos serán mostrados.

En esta sección hicimos un servidor HTTP a partir de una estructura, un entero, un canal y una función, todo porque las interfaces solo son conjuntos de métodos, los cuales se pueden definir para (casi) cualquier tipo.

El identificador blanco

Hasta ahora hemos mencionado un par de veces el identificador blanco, en el contexto de los bucles `for range` y mapas. El identificador blanco se puede asignar o declarar con cualquier valor de cualquier tipo, desechando el valor inofensivamente. Es un poco como escribir al archivo `/dev/null` en Unix: este representa un valor de solo escritura para utilizarlo como marcador de posición donde se necesita una variable pero el valor real es irrelevante. Tiene usos más allá de lo que ya hemos visto.

El identificador blanco en asignación múltiple

El uso de un identificador blanco en un bucle `for range` es un caso especial de una situación general: asignación múltiple.

Si una asignación requiere múltiples valores en el lado izquierdo, pero uno de los valores no será utilizado por el programa, un identificador blanco en el lado izquierdo de la asignación evita la necesidad de crear una variable inútil y aclara el hecho de que el valor será descartado. Por ejemplo, cuándo llamas a una función que regresa un valor y un error, pero solo el error es importante, usa el identificador blanco para desechar el valor irrelevante.

```
if _, err := os.Stat(ruta); os.IsNotExist(err) {
    fmt.Printf("%s no existe\n", ruta)
}
```

Ocasionalmente verás código que descarte el valor de error para ignorar el error; esta es una práctica terrible. Siempre revisa el error devuelto; este se proporciona por una razón.

```
// ¡Mal! Este código chocará si la ruta no existe.
ar, _ := os.Stat(ruta)
if ar.IsDir() {
    fmt.Printf("%s es un directorio\n", path)
}
```

Importaciones y variables no utilizadas

Es un error importar un paquete o declarar una variable sin utilizarla. Las importaciones no utilizadas hinchan el programa y entorpecen la compilación, mientras que una variable iniciada pero no utilizada por lo menos es un ciclo del reloj malgastado y quizás un indicativo de un defecto más grave. Cuando un programa está bajo activo desarrollo, no obstante, a menudo surgen importaciones y variables no utilizadas y puede ser molesto eliminarlas solo para acelerar la compilación, únicamente para necesitarlas otra vez más tarde. El identificador blanco proporciona una solución alternativa.

Este programa medio escrito tiene dos importaciones sin usar (`fmt` e `io`) y una variable inútil (`fd`), por lo tanto no compila, pero sería bueno ver qué tanto código es correcto.

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)

func main() {
    fd, err := os.Open("prueba.go")
    if err != nil {
        log.Fatal(err)
    }
    // PENDIENE: usar fd.
}
```

Para silenciar las quejas sobre importaciones no utilizadas, usa un identificador blanco para referir a un símbolo desde el paquete importado. De igual modo, asignar la variable `fd` no usada al identificador blanco acalla el error de variable no utilizada. Esta versión del programa sí compila.

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)

var _ = fmt.Printf // Para depuración; eliminar cuando esté hecho.
var _ io.Reader    // Para depuración; eliminar cuando esté hecho.

func main() {
    fd, err := os.Open("prueba.go")
    if err != nil {
        log.Fatal(err)
    }
    // PENDIENE: usar fd.
    _ = fd
}
```

Por convención, las declaraciones globales para silenciar errores de importación deberían venir justo después de las importaciones y estar comentadas, únicamente para facilitar su ubicación y como recordatorio para limpiar las cosas más tarde.

Efectos secundarios de importación

Una importación no utilizada tal como `fmt` o `io` en el ejemplo anterior eventualmente se tendría que utilizar o remover: las asignaciones al identificador blanco se asocian a código que todavía es un trabajo en progreso. Pero a veces es útil importar un paquete solo por sus efectos secundarios, sin ningún uso explícito. Por ejemplo, durante tu función `init`, el paquete `[net/http/pprof](/pkg/net/http/pprof/)` registra los controladores HTTP que proporcionan información de depuración. Esta tiene una API exportada, pero la mayoría de los clientes únicamente necesitan registrar el controlador y acceder a los datos a través de una página web. Para importar el paquete solo por sus efectos secundarios, renombra el paquete al identificador blanco:

```
import _ "net/http/pprof"
```

De esta forma aclaras que la importación del paquete se está haciendo solamente por sus efectos secundarios, porque no hay otro uso posible del paquete: en este archivo, este no tiene nombre. (Si lo tuviera y no utilizamos ese nombre, el compilador rechazaría el programa).

Revisando la interfaz

Como vimos en la explicación de las interfaces arriba, un tipo no necesita declarar explícitamente qué interfaz implementa. En cambio, un tipo implementa la interfaz justo al implementar los métodos de la interfaz. En la práctica, la mayoría de las conversiones de interfaz son estáticas y por tanto comprobadas en tiempo de compilación. Por ejemplo, al pasar un `*os.File` a una función que espera un `io.Reader` no compilará a no ser que `*os.File` implemente la interfaz `io.Reader`.

No obstante, algunas revisiones de la interfaz ocurren en tiempo de ejecución. Hay un ejemplo de esto en el paquete `[encoding/json](/pkg/encoding/json/)`, el cual define una interfaz `[Marshaler](/pkg/encoding/json/#Marshaler)`. Cuando el codificador JSON recibe un valor que implementa esa interfaz, el codificador invoca los valores del método `marshaling` para convertirlo a JSON en vez de hacer la conversión estándar. El codificador revisa esta propiedad en tiempo de ejecución con una aserción de tipo como esta:

```
m, ok := val.(json.Marshaler)
```

Si es necesario únicamente averigua si un tipo implementa cierta interfaz, sin de hecho utilizar la interfaz, quizás como parte de una comprobación de error, usa el identificador blanco para descartar el tipo/valor acertado:

```
if _, ok := val.(json.Marshaler); ok {
    fmt.Printf("el valor %v de tipo %T implementa json.Marshaler\n", val, val)
}
```

Un lugar donde surge esta situación es cuando es necesario garantizar dentro del paquete que implementa el tipo que de hecho satisface la interfaz. Si un tipo —por ejemplo, `[json.RawMessage](/pkg/encoding/json/#RawMessage)` — necesita una representación JSON personalizada, esta debe implementar `json.Marshaler`, pero no hay conversiones estáticas que provoquen que el compilador verifique esto automáticamente. Si el tipo falla inadvertidamente para satisfacer la interfaz, el codificador JSON todavía trabajará, pero no utilizará la implementación personalizada. Para garantizar que la implementación es correcta, se puede utilizar en el paquete una declaración global utilizando el identificador blanco:

```
var _ json.Marshaler = (*RawMessage)(nil)
```

En esta declaración, la asignación implica una conversión de un `*RawMessage` a un `Marshaler` la cual requiere que `*RawMessage` implemente a `Marshaler` y que la propiedad sea comprobada en tiempo de compilación. `json.Marshaler` debe cambiar la interfaz, este paquete ya no compilará y serás notificado sobre qué necesitas actualizar.

El aspecto del identificador blanco en este constructor indica que la declaración existe solo para el tipo que está comprobando, no para crear una variable. No obstante, esto no es para cada tipo que satisface una interfaz. Por convención, tales declaraciones solo se utilizan cuando no hay conversiones estáticas presentes en el código, el cual es un acontecimiento raro.

Incrustando

Go no proporciona la típica, idea conocida de subclases de tipo, pero tiene la habilidad de “tomar piezas prestadas” de una implementación *incrustando* tipos dentro de una estructura o interfaz.

Una interfaz incrustada es muy sencilla. Ya hemos mencionado antes las interfaces `io.Reader` e `io.Writer`; aquí están sus definiciones.

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

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

El paquete `io` también exporta muchas otras interfaces que especifican objetos que pueden implementar muchos de esos métodos. Por ejemplo, está `io.ReadWriter`, una interfaz que contiene ambos métodos `Read` y `Write`. Podríamos especificar `io.ReadWriter` al enumerar los dos métodos explícitamente, pero es más fácil y más sugerente incrustar las dos interfaces para formar una nueva, de esta manera:

```
// ReadWriter es la interfaz que combina las interfaces Reader y Writer.
type ReadWriter interface {
    Reader
    Writer
}
```

Esto solo dice que: un `ReadWriter` puede hacer lo que hace un `Reader` **y** lo que hace un `Writer`; es una unión de interfaces incrustadas (las cuáles tienen que ser conjuntos de métodos disjuntos). Únicamente se pueden incrustar interfaces en interfaces.

La misma idea básica aplica a estructuras, pero con más implicaciones de ámbito. El paquete `bufio` tiene dos tipos de estructuras, `bufio.Reader` y `bufio.Writer`, naturalmente, cada cual implementa las interfaces análogas del paquete `io`. Y `bufio` también implementa un escritor/lector con búfer, el cual trabaja combinando un lector y un escritor en una estructura utilizando la incrustación: esta enumera los tipos dentro de la estructura pero no les da nombre a los campos.

```
// ReadWriter almacena punteros a un Lector y a un Escritor.
// Este implementa io.ReadWriter.
type ReadWriter struct {
    *Reader  // *bufio.Reader
    *Writer  // *bufio.Writer
}
```

Los elementos incrustados son punteros a estructuras y naturalmente se tienen que iniciar para apuntar a estructuras válidas antes de que se puedan utilizar. La estructura `ReadWriter` se podría haber escrito de la siguiente manera

```
type ReadWriter struct {
    reader *Reader
    writer *Writer
}
```

Pero entonces para promover los métodos de los campos y para satisfacer las interfaces de `io`, también necesitaríamos proporcionar métodos de reenvío, tal como este:

```
func (rw *ReadWriter) Read(p []byte) (n int, err error) {
    return rw.reader.Read(p)
}
```

Al incrustar las estructuras directamente, evitamos esta contabilidad. Los métodos de los tipos incrustados vienen gratis, lo cual significa que `bufio.ReadWriter` no solo tiene los métodos `bufio.Reader` y `bufio.Writer`, sino que también satisface otras tres interfaces: `io.Reader`, `io.Writer` e `io.ReadWriter`.

Hay una importante manera en que la incrustación difiere de las subclases. Al incrustar un tipo, los métodos de ese tipo se vuelven métodos del tipo externo, pero cuando se invoca el receptor del método es el tipo interno, no el externo. En nuestro ejemplo, cuando se invoca al método `Read` de un `bufio.ReadWriter`, este tiene exactamente el mismo efecto como el método de reenvío escrito arriba; el receptor es el campo `reader` del `ReadWriter`, no el `ReadWriter` en sí mismo.

La incrustación también puede ser una simple comodidad. Este ejemplo muestra un campo incrustado junto a un campo nombrado regular.

```
type Empleo struct {
    Orden string
    *log.Logger
}
```

El tipo `Empleo` ahora tiene los métodos `Log`, `Logf` y otros de `*log.Logger`. Podríamos haberle dado el nombre de campo `Logger`, por supuesto, pero no es necesario hacerlo. Y ahora, una vez iniciado, podemos registrar el `Empleo`:

```
empleo.log("empezando ahora...")
```

El `Logger` es un campo regular de la estructura `Empleo`, por lo tanto lo podemos iniciar en la manera habitual dentro del constructor de `Empleo`, de esta manera:

```
func NuevoEmpleo(orden string, notario *log.Logger) *Empleo {
    return &Empleo{orden, notario}
}
```

o con un literal compuesto,

```
empleo := &Empleo{orden, log.New(os.Stderr, "Empleo: ", log.Ldate)}
```

Si necesitamos referirnos directamente a un campo incrustado, el nombre de tipo del campo, ignorando el calificador de paquete, sirve como nombre de campo, tal como lo hace el método `Read` de nuestra estructura `ReaderWriter`. Aquí, si necesitamos acceder al `*log.Logger` de una variable `empleo` de tipo `Empleo`, escribiríamos `empleo.Notario`, lo cual sería útil si quisiéramos refinar los métodos del `Logger`.

```
func (empleo *Empleo) Logf(format string, args ...interface{}) {
    empleo.Notario.Logf("%q: %s", empleo.Orden, fmt.Sprintf(format, args...))
}
```

La incrustación de tipos introduce el problema de conflictos de nombre pero las reglas para resolverlos son sencillas. Primero, un campo o método `X` oculta cualquier otro elemento `X` en una parte anidada más profunda del tipo. Si `log.Logger` contuviera un campo o método llamado `Orden`, el campo `Orden` del `Empleo` predominaría.

Segundo, si el mismo nombre aparece al mismo nivel de anidamiento, normalmente es un error; será erróneo incrustar `log.Logger` si la estructura `Empleo` contuviera otro campo o método llamado `Logger`. Sin embargo, si el nombre duplicado nunca se menciona en el programa fuera de la definición del tipo, está bien. Esta cualificación proporciona alguna protección contra cambios hechos a tipos incrustados externos; no hay ningún problema si se añade un campo que entre en conflicto con otro campo en otro subtipo si nunca se utiliza ese campo.

Concurrencia

Compartiendo por comunicación

La programación concurrente es un gran tema y este únicamente es un espacio para resaltar algo específico de Go.

La programación concurrente en muchos entornos se dificulta por las sutilezas requeridas para implementar el correcto acceso a variables compartidas. Go estimula un enfoque diferente en el cual los valores compartidos se pasan en canales y, de hecho, nunca se

comparten activamente entre hilos de ejecución separados. solo una rutinago tiene acceso al valor en cualquier momento dado. Las pugnas de datos no pueden ocurrir, por diseño. Para animar este modo de pensar lo hemos reducido a un eslogan:

No comunicar por medio de memoria compartida; en su lugar, compartir la memoria por comunicación.

Este enfoque se puede llevar demasiado lejos. La cantidad de referencia se puede hacer poniedo una exclusión mutua en torno a una variable entera, por ejemplo. Pero como enfoque de alto nivel, utilizar canales para controlar el acceso facilita la escritura de programas claros y correctos.

Una manera de pensar sobre este modelo es considerar un típico programa de un solo hilo corriendo en una CPU. Este no necesita primitivas de sincronización. Al correr otra instancia del mismo; tampoco necesita ninguna sincronización. Ahora, si permitimos que las dos se comuniquen; si la comunicación es el sincronizador, allí todavía no hay ninguna necesidad de otra sincronización. Las tuberías de Unix, por ejemplo, encajan en este modelo perfectamente. Aunque el enfoque de concurrencia en Go se origina en los procesos secuenciales de comunicación (PSC), que también se pueden ver como una generalización con seguridad de tipos de las tuberías Unix.

Rutinasgo

Se llaman *rutinasgo* porque los términos existentes como hilos, corutinas, procesos, etc., expresan connotaciones inexactas. Una rutinago tiene un modelo sencillo: es una función ejecutándose al mismo tiempo que otras rutinasgo en el mismo espacio. Es ligera, puesto que consta de un poco más de espacio que la asignación de la pila. Y la pila el inicio es pequeña, así que son baratas y crecen reservando (y liberando) almacenamiento en el montón cuando se requiere.

Las rutinasgo son multiplexadas en varios hilos del SO por lo que si una se debe bloquear, tal como cuándo espera E/S, las demás siguen funcionando. Su diseño esconde mucha de la complejidad de la creación y administración de hilos.

Prefija una función o llamada a método con la palabra clave `go` para ejecutar la llamada en una nueva rutinago. Cuando la llamada se completa, la rutinago sale, silenciosamente. (El efecto es similar a la notación `&` del intérprete de ordenes de Unix para ejecutar una orden en segundo plano).

```
go list.Sort()  // ejecuta list.Sort simultáneamente; no espera a que termine.
```

Una función literal se puede manejar en una invocación a una rutinago.

```
func Anuncia(mensaje string, demora time.Duration) {
    go func() {
        time.Sleep(demora)
        fmt.Println(mensaje)
    }()  // Observa los paréntesis - estos llaman a la función.
}
```

En Go, las funciones literales son cierres: la implementación se asegura de que las variables referidas por la función sobrevivan mientras están activas.

Estos ejemplos no son demasiado prácticos porque las funciones no tienen ninguna manera de señalar su conclusión. Para hacerlo, necesitamos canales.

Canales

Como los mapas, los canales se asignan con `make` y el valor resultante actúa como referencia a una estructura de datos subyacente. Si se proporciona un parámetro entero opcional, este establece el tamaño del búfer para el canal. El predeterminado es cero, para un canal síncrono o sin búfer.

```
ci := make(chan int)            // canal de enteros sin búfer
cj := make(chan int, 0)         // canal de enteros sin búfer
cs := make(chan *os.File, 100)  // canal de punteros a Archivos con búfer
```

Los canales sin búfer combinan el intercambio de comunicación de un valor con sincronización garantizando que dos cálculos (rutinasgo) se encuentran en un estado conocido.

Hay muchos buenos modismos utilizando canales. Aquí está uno para que podamos comenzar. En la sección anterior lanzamos una clase en segundo plano. Un canal te puede permitir lanzar una rutinago para esperar que la orden `sort` termine.

```
c := make(chan int)  // Reserva un canal.
// Inicia la orden sort en una rutinago; al completarse, lo señala en el canal.
go func() {
    list.Sort()
    c <- 1  // Envía una señal; el valor no importa.
}()
hazAlgoMientras()
<-c   // Espera a que Sort termine; desecha el valor enviado.
```

Los receptores siempre se bloquean hasta que hay datos por recibir. Si el canal es sin búfer, el emisor se bloquea hasta que el receptor ha recibido el valor. Si el canal tiene un búfer, el emisor solo se bloquea hasta que el valor se ha copiado al búfer; si el búfer está lleno,

significa que tiene que esperar hasta que algún receptor haya recuperado un valor.

Un canal con búfer se puede utilizar como un semáforo, por ejemplo para limitar la transmisión de datos. En este ejemplo, las peticiones entrantes se pasan al `controlador`, el cual envía un valor al canal, este procesa la petición y entonces recibe un valor desde el canal para preparar el "semáforo" para el siguiente consumidor. La capacidad del búfer del canal limita el número de llamadas simultáneas a `procesar`.

```
var sem = make(chan int, MaxOutstanding)
func controlador(r *Request) {
    sem <- 1     // Espera hasta drenar la cola activa.
    procesar(r)  // Puede tomar mucho tiempo.
    <-sem        // Hecho; permite ejecutar la siguiente petición.
}

func Serve(cola chan *Request) {
    for {
        req := <-cola
        go controlador(req)  // no espera a que termine el controlador.
    }
}
```

Una vez que los controladores `MaxOutstanding` están ejecutando `procesar`, alguno más lo bloqueará tratando de enviar al canal con el búfer lleno, hasta que uno de los controladores existentes termine y reciba desde el búfer.

Este diseño tiene un problema, si bien: `Serve` crea una nueva rutinago para cada petición entrante, incluso aunque únicamente `MaxOutstanding` de ellos se puedan ejecutar en cualquier momento. Como resultado, el programa puede consumir recursos ilimitados si las peticiones entran demasiado rápido. Podemos tratar esa deficiencia cambiando `Serve` a la entrada de la creación de las rutinasgo. Aquí tienes una solución obvia, pero ten cuidado esta tiene un fallo que arreglaremos posteriormente:

```
func Serve(cola chan *Request) {
    for req := range cola {
        sem <- 1
        go func() {
            procesar(req) // Lleno de errores; ve la explicación abajo.
            <-sem
        }()
    }
}
```

El fallo es que en un bucle `for` de Go, la variable del bucle se reutiliza en cada iteración, así que la variable `req` es compartida en todas las rutinasgo. Eso no es lo que deseamos. Necesitamos asegurarnos que `req` es única en cada rutinago. Aquí está una manera de

hacerlo, pasando el valor de `req` como un argumento para el cierre en la rutinago:

```
func Serve(cola chan *Request) {
    for req := range cola {
        sem <- 1
        go func(req *Request) {
            procesar(req)
            <-sem
        }(req)
    }
}
```

Compara esta versión con la anterior para ver la diferencia en cómo se declara y ejecuta el cierre. Otra solución es solo creando una nueva variable con el mismo nombre, como en este ejemplo:

```
func Serve(cola chan *Request) {
    for req := range cola {
        req := req // Crea una nueva instancia de req para la rutinago.
        sem <- 1
        go func() {
            procesar(req)
            <-sem
        }()
    }
}
```

Puede parecer extraño escribir

```
req := req
```

Pero hacerlo en Go es legal e idiomático. Consigues una versión fresca de la variable con el mismo nombre, ocultando deliberadamente la variable del bucle local pero únicamente para cada rutinago.

Volviendo al problema general de escribir el servidor, otro enfoque para manejar bien los recursos es empezar a corregir una serie de rutinasgo `controladora`s que están leyendo desde el canal de la petición. La cantidad de rutinasgo límitan el número de llamadas simultáneas a `procesar`. Esta función `Serve` también acepta un canal en el cual se producirá la salida; después de lanzar las rutinasgo estas bloquean la recepción de ese canal.

```
func controlador(cola chan *Request) {
    for r := range cola {
        procesar(r)
    }
}

func Serve(clientRequests chan *Request, quit chan bool) {
    // Inicia los controladores
    for i := 0; i < MaxOutstanding; i++ {
        go controlador(clientRequests)
    }

    <-quit  // Espera a que le digan que salga.
}
```

Canales de canales

Una de las propiedades más importantes de Go es que un canal es un valor de primera clase que se puede reservar y pasar como cualquier otro. Un uso común de esta propiedad es para implementar la demultiplexión segura y en paralelo.

En el ejemplo de la sección anterior, el `controlador` era un controlador idealizado para una petición pero no definimos el tipo que controla. Si ese tipo incluye un canal en el cual responder, cada cliente puede proporcionar su propia ruta para la respuesta. Aquí tienes una definición esquemática del tipo `Request`.

```
type Request struct {
    args        []int
    f           func([]int) int
    resultChan  chan int
}
```

El cliente proporciona una función y sus argumentos, así como un canal dentro del objeto petición en el cual recibir la respuesta.

```
func suma(a []int) (s int) {
    for _, v := range a {
        s += v
    }
    return
}

request := &Request{[]int{3, 4, 5}, suma, make(chan int)}
// envía la petición
clientRequests <- request
// Espera la respuesta.
fmt.Printf("respuesta: %d\n", <-request.resultChan)
```

En el lado del servidor, la función controladora es la única cosa que cambia.

```
func controlador(cola chan *Request) {
    for req := range cola {
        req.resultChan <- req.f(req.args)
    }
}
```

Allí claramente hay mucho más por mejorar para hacerlo realista, pero este código es una plataforma para un índice limitado en paralelo, un sistema RPC no bloqueante y allí no hay una exclusión mutua a la vista.

Paralelización

Otra aplicación de estas ideas es el cálculo en paralelo en una CPU de múltiples núcleos. Si el cálculo se puede romper en piezas separadas que se puedan ejecutar independientemente, se puede paralelizar, con un canal para señalar cuando cada pieza esté completa.

Digamos que tienes que ejecutar una costosa operación en un vector de elementos y que el valor de la operación en cada elemento es independiente, como en este ejemplo idealizado.

```
type Vector []float64

// Aplica la operación a v[i], v[i+1] ... hasta v[n-1].
func (v Vector) HazAlgo(i, n int, u Vector, c chan int) {
    for ; i < n; i++ {
        v[i] += u.Op(v[i])
    }
    c <- 1    // señala que esta pieza está terminada
}
```

Lanzamos las piezas independientemente en un bucle, una por CPU. Estas se pueden completar en cualquier orden pero este no importa; justo contamos las señales de conclusión drenando el canal después de lanzar todas las rutinasgo.

```
const NCPU = 4  // número de núcleos del CPU
func (v Vector) HazTodo(u Vector) {
    c := make(chan int, NCPU)  // Con búfer opcional pero perceptible.
    for i := 0; i < NCPU; i++ {
        go v.HazAlgo(i*len(v)/NCPU, (i+1)*len(v)/NCPU, u, c)
    }

    // Drena el canal.
    for i := 0; i < NCPU; i++ {
        <-c    // espera a que se complete una tarea
    }

    // Todo terminado.
}
```

La implementación actual en tiempo de ejecución de Go por omisión no paraleliza este código. Esta únicamente dedica un solo núcleo para procesar a nivel de usuario. Una arbitraria cantidad de rutinasgo se puede bloquear en llamadas al sistema, pero de manera predeterminada solo una puede ejecutar código a nivel de usuario en un momento dado. Esta debería ser inteligente pero un día será más lista y hasta entonces si quieres paralelismo en tu CPU tienes que decirle al tiempo de ejecución cuantas rutinasgo quieres que ejecuten código simultáneamente. Hay dos maneras relacionadas de hacer esto. O bien, ejecutas tu trabajo con la variable de entorno `GOMAXPROCS` configurada al número de núcleos a utilizar o importas el paquete `runtime` y llamas a `runtime.GOMAXPROCS(NCPU)`. Un útil valor podría ser `runtime.NumCPU()`, el cual informa el número de CPUs lógicos en la máquina local. Una vez más, este requisito se espera que sea retirado conforme a lo planificado y el tiempo de ejecución mejore.

Asegúrate de no confundir las ideas de concurrencia—estructuración de un programa ejecutando componentes independientemente —y paralelismo— ejecutando cálculos en paralelo para eficiencia en múltiples CPUs. A pesar de que las características de concurrencia de Go pueden facilitar la resolución de algunos problemas al estructurar cálculos en paralelo, Go es un lenguaje concurrente, no en paralelo y no todos los problemas de paralelización encajan en el modelo de Go. Para una explicación de la distinción, ve la charla mencionada en esta publicación del blog.

Un búfer agujereado

Las herramientas de la programación concurrente incluso pueden hacer que las ideas no concurrentes sean más fáciles de expresar. Aquí tienes un ejemplo extraído de un paquete RPC. La rutinago cliente itera recibiendo datos de alguna fuente, quizás una red. Para evitar alojar y liberar búferes, mantiene una lista libre y utiliza un canal con búfer para representarlos. Si el canal está vacío, se reserva un nuevo búfer. Una vez que el búfer del mensaje está listo, se envía al servidor en `serverChan` .

```
var listaLibre = make(chan *Buffer, 100)
var servidorDelCanal = make(chan *Buffer)

func cliente() {
    for {
        var b *Buffer
        // Graba un búfer si está disponible; si no lo reserva.
        select {
        case b = <-listaLibre:
            // Consigue uno; nada más por hacer.
        predefinido:
            // Ninguno libre, así que reserva uno nuevo.
            b = new(Buffer)
        }

        carga(b)             // Lee el siguiente mensaje de la red.
        servidorDelCanal <-b // Envía al servidor.
    }
}
```

El bucle del servidor recibe cada mensaje del cliente, lo procesa y regresa el búfer a la lista libre.

```
func servidor() {
    for {
        b := <-servidorDelCanal    // Espera el trabajo.
        procesar(b)
        // Reutiliza el búfer si hay espacio.
        select {
        case listaLibre <- b:
            // Búfer en la lista libre; nada más que hacer.
        predefinido:
            // Lista libre llena, justo te lleva al principio.
        }
    }
}
```

El cliente intenta recuperar un búfer desde `listaLibre` ; si ninguno está disponible, reserva uno fresco. El servidor lo envía a `listaLibre` la cual pone `b` al final en la lista libre a no ser que la lista esté llena, en cuyo caso se desecha el búfer para ser reclamado por el

recolector de basura. (Las cláusulas `default` en las declaraciones `select` se ejecutan cuando ningún otro caso está listo, significa que el `select` nunca se bloquea). Esta implementación construye un cubo agujereado de la lista libre en solo unas cuantas líneas, confiando en que el canal con búfer y el recolector de basura llevarán la contabilidad.

Errores

Las rutinas de la biblioteca a menudo tienen que regresar alguna clase de indicación de error al llamador. Anteriormente, cuando mencionamos que, el multivalor de retorno de Go facilita la devolución de una detallada descripción de error junto al valor de retorno normal. Es un buen estilo utilizar esta característica para proporcionar detallada información del error. Por ejemplo, cuando vemos, `os.Open` no solo regresa un puntero `nil` en caso de falla, también regresa un valor de error que describe qué estuvo mal.

Por convención, los errores tienen el tipo `error`, una sencilla interfaz incorporada.

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

Una biblioteca de escritura es libre de implementar esta interfaz con un modelo más rico bajo la cubierta, haciéndo posible no solo ver el error sino también para proporcionar algún contexto. Como mencionamos, junto al habitual valor de retorno `*os.File`, `os.Open` también regresa un valor de error. Si el archivo se abre exitosamente, el error será `nil`, pero cuando hay un problema, este tendrá un `os.PathError`:

```
// PathError registra un error, la operación y
// la ruta al archivo que lo causó.
type PathError struct {
    Op string    // "open", "unlink", etc.
    Path string  // El archivo asociado.
    Err error    // Devuelto por la llamada al sistema.
}

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

Los `errores` `PathError` generan una cadena como esta:

```
open /etc/passwx: no existe el archivo o directorio
```

Tal error, incluye el nombre del archivo problemático, la operación y el error del sistema operativo provocado, es útil incluso si se imprime fuera de la llamada que lo causó; es mucho más informativo que el sencillo "no existe el archivo o directorio".

Cuando es factible, las cadenas de error deberían identificar su origen, tal como cuando tienen un prefijo que nombra la operación o paquete que generó el error. Por ejemplo, en el paquete `image`, la representación de la cadena para un error de decodificación debido a un formato desconocido es "imagen: formato desconocido".

A los llamadores que les preocupan los detalles precisos del error pueden utilizar un switch de tipo o una aserción de tipo para buscar errores específicos y extraer los detalles. Para `PathErrors` esto podría incluir examinar el campo interno `Err` para fallos recuperables.

```
for try := 0; try < 2; try++ {
    archivo, err = os.Create(nombrearchivo)

    if err == nil {
        return
    }

    if e, ok := err.(*os.PathError); ok && e.Err == syscall.ENOSPC {
        deleteTempFiles()  // Recupera algo de espacio.
        continue
    }
    return
}
```

Aquí la segunda declaración `if` es otra aserción de tipo. Si esta falla, `ok` será falso y `e` será `nil`. Si tiene éxito, `ok` será cierto, lo cual significa que el error era de tipo `*os.PathError` y también `e` lo es, el cual podemos examinar para obtener más información sobre el error.

Pánico

La manera habitual de informar un error a un llamador es regresando un `error` como un valor de retorno extra. El método canónico `Read` es un caso bien conocido; este regresa un contador de byte y un `error`. ¿Pero qué pasa si el error es irrecuperable? A veces el programa sencillamente no puede continuar.

Para este propósito, hay una función `panic` incorporada que en efecto crea un error en tiempo de ejecución que detendrá el programa (pero ve la siguiente sección). La función toma un solo argumento de tipo arbitrario —a menudo una cadena— para imprimirla cuando el programa muere. También es una manera de indicar que algo imposible ha sucedido, tal como salir de un bucle infinito.

```
// Una ingenua implementación de la raíz cúbica utilizando el método de
// Newton.
func RaízCubica(x float64) float64 {
    z := x/3   // Valor inicial arbitrario
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z-x) / (3*z*z)
        if muyCerca(z, prevz) {
            return z
        }
    }

    // Un millón de iteraciones no han convergido; algo está mal.
    panic(fmt.Sprintf("La RaízCubica(%g) no convergió", x))
}


Este solo es un ejemplo pero las funciones de una biblioteca real deberían
evitar `panic`.  Si el problema se puede enmascarar o hacer funcionar de
otro modo, siempre es mejor dejar que las cosas continúen ejecutándose en
lugar de detener el programa entero.  Un posible contraejemplo es durante
el inicio: si la biblioteca verdaderamente no puede arrancar, podría ser
razonable invocar a pánico.

```go
var usuario = os.Getenv("USER")

func init() {
    if usuario == "" {
        panic("no hay valor para $USER")
    }
}
```

Recuperando

Cuándo se invoca a `panic`, incluyendo implícitamente los errores en tiempo de ejecución tal como la indexación fuera de límites de un sector o una fallida aserción de tipo, inmediatamente detiene la ejecución de la función actual y empieza a revertir la pila de la rutinago, ejecutando cualquier función diferida. Si esa reversión alcanza la parte superior de la pila en la rutinago, el programa muere. Sin embargo, es posible utilizar la función incorporada `recover` para recobrar el control de la rutinago y reanudar la ejecución normal.

Una llamada a `recover` detiene la reversión y regresa el argumento pasado a `panic`. Debido a que el único código que se ejecuta mientras la reversión está dentro de las funciones diferidas, `recover` solo es útil dentro de las funciones diferidas.

Una aplicación de `recover` es cerrar una rutinago que esté fallando dentro de un servidor sin matar las otras rutinasgo que se estén ejecutando.
```

```
func servidor(canalTrabajando <-chan *Trabajo) {
    for trabajo := range canalTrabajando {
        go hazloConSeguridad(trabajo)
    }
}

func hazloConSeguridad(trabajo *Trabajo) {
    defer func() {
        if err := recover(); err != nil {
            log.Println("trabajo fallido:", err)
        }
    }()
    haz(trabajo)
}
```

En este ejemplo, si `haz(trabajo)` entra en pánico, el resultado será registrado y la rutinago saldrá limpiamente sin perturbar a las otras. No hay necesidad de hacer nada más en el cierre diferido; al invocar a `recover` esta maneja la condición completamente.

Dado que `recover` siempre regresa `nil` a menos que la llamaras directamente desde una función diferida, el código diferido puede llamar a rutinas de la biblioteca en las que ellas mismas utilicen `panic` y `recover` sin fallar. Por ejemplo, la función diferida en `hazloConSeguridad` podría llamar a una función de registro antes de llamar a `recover` y el código de registro se ejecutaría sin afectación por el estado de panic.

Con nuestro patrón de recuperación en su sitio, la función `haz` (y cualquiera que la llame) puede salir limpiamente de cualquier mala situación al llamar a `panic`. Podemos utilizar esa idea para simplificar el manejo del error en software complejo. Démos una mirada a una versión idealizada de un paquete `regexp`, el cual informa análisis de errores llamando a `panic` con un tipo de error local. Aquí está la definición de `Error`, un método `error` y la función `Compila`.

```
// Error es el tipo de un error de analisis; este satisface la interfaz
// error.
type Error string

func (e Error) Error() string {
    return string(e)
}

// error es un método de *Regexp que informa los errores del análisis
// entrando en pánico con un Error.
func (regexp *Regexp) error(err string) {
    panic(Error(err))
}

// Compila regresa un representación procesada de la expresión regular.
func Compila(cadena string) (regexp *Regexp, err error) {
    regexp = new(Regexp)
    // hazProceso invoca a panic si hay un error de análisis.
    defer func() {
        if e := recover(); e != nil {
            regexp = nil    // Limpia el valor de retorno.
            err = e.(Error) // Reinvocará a panic si no un es un error
                            // de análisis.
        }
    }()
    return regexp.hazProceso(cadena), nil
}
```

Si `hazProceso` invoca a `panic`, el bloque de recuperación pondrá el valor de retorno a `nil` —las funciones diferidas pueden modificar los valores de retorno nombrados. Este entonces comprobará, en la asignación a `err`, que el problema era un error de análisis al asertar que este tiene el tipo `Error` local. Si no, la aserción de tipo fallará, causando un error en tiempo de ejecución que continúa revirtiendo la pila como si nada lo hubiera interrumpido. Esta comprobación significa que si pasa algo inesperado, tal como un índice fuera de límites, el código fallará incluso aunque estemos utilizando `panic` y `recover` para manejar los errores de análisis.

Con el manejo de errores en su sitio, el método `error` (debido a que es un método vinculando al tipo, está bien, incluso es natural, que este tenga el mismo nombre que el tipo `error` incorporado) facilita el análisis del informe de errores sin preocuparse de revertir a mano la pila del análisis:

```
if pos == 0 {
    re.error("'*' ilegal al inicio de la expresión")
}
```

Útil, aunque este patrón se debe utilizar solo dentro de un paquete. `Parse` cambia sus llamadas internas a `panic` en valores de `error`; no expone el resultado de `panic` a su cliente. Esta es una buena regla a seguir.

Por cierto, este modismo de rellamar a panic cambia el valor de panic si ocurre un error real. No obstante, ambas fallas la original y la nueva serán presentadas en el informe de la colisión, así que la causa raíz del problema todavía será visible. Por lo que este sencillo enfoque de rellamar a panic normalmente es suficiente —es una colisión después de todo— pero si solo quieres mostrar el valor original, puedes escribir un poco más de código para filtrar problemas inesperados y rellamar a `panic` con el error original. Esto lo dejamos como ejercicio para el lector.

Un servidor web

Terminaremos con un programa Go completo, un servidor web. Este, de hecho es una clase de reservidor web. Google proporciona un servicio en http://chart.apis.google.com que automáticamente da formato a datos en diagramas y gráficas. Aunque es difícil utilizarlo interactivamente, debido a que necesitas poner los datos en el URL como consulta. El programa proporciona una buena interfaz para un formulario de datos: dada una breve pieza de texto, este llama al servidor de gráficas para producir un código QR, un arreglo de cajas que codifican el texto. Puedes capturar esta imagen con la cámara de tu teléfono celular e interpretarla cómo, por ejemplo, una URL, ahorrándote el tener que escribir la URL en el minúsculo teclado del teléfono.

Aquí está el programa completo. Le sigue una explicación.

```
package main

import (
    "flag"
    "html/template"
    "log"
    "net/http"
)

var dir = flag.String("dir", ":1718", "dirección del servicio http") // Q=17, R=18

var plantilla = template.Must(template.New("qr").Parse(cadenaPlantilla))

func main() {
    flag.Parse()
    http.Handle("/", http.HandlerFunc(QR))
    err := http.ListenAndServe(*dir, nil)
    if err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}

func QR(w http.ResponseWriter, req *http.Request) {
    plantilla.Execute(w, req.FormValue("s"))
}

const cadenaPlantilla = `<!doctype html>
<html>
    <head>
        <title>Generador de enlaces QR</title>
    </head>
    <body>
        {{if .}}
            <img src="http://chart.apis.google.com/chart?chs=300x300&cht=qr&choe=UTF-8&ch
            <br>
            {{.}}
            <br>
            <br>
        {{end}}
        <form action="/" name=f method="GET">
            <input maxLength=1024 size=70 name=s value="" title="Texto QR a codificar">
            <input type=submit value="Muestra QR" name=qr>
        </form>
    </body>
</html>
```

Te debería ser fácil seguir las piezas hasta `main`. Una bandera configura un puerto HTTP predefinido para nuestro servidor. La variable `plantilla` es dónde empieza la diversión. Esta construye una plantilla HTML que será ejecutada por el servidor para mostrar la

página; más sobre esto en un momento.

La función `main` procesa las banderas y, utilizando el mecanismo del que hablamos anteriormente, vincula la función `QR` a la ruta raíz del servidor. Luego se invoca a `http.ListenAndServe` para arrancar el servidor; esta se bloquea mientras se ejecuta el servidor.

`QR` solo recibe la petición, la cual contiene datos del formulario y ejecuta la plantilla en los datos con el valor del formulario llamado `s`.

El paquete de plantillas `html/template` es potente; este programa solo toca una minúscula parte de su capacidad. En esencia, este reescribe al vuelo una pieza de texto HTML sustituyendo los elementos derivados desde los datos pasados a `plantilla.Execute`, en este caso el valor del formulario. En el texto de la plantilla (`cadenaPlantilla`), las piezas delimitadas por dobles llaves denotan acciones de la plantilla. La pieza desde `&quot;{{if .}}&quot;` hasta `&quot;{{end}}&quot;` solo se ejecuta si el valor del elemento de datos actual, llamado `.` (punto), no está vacío. Es decir, cuándo la cadena está vacía, esta pieza de la plantilla se suprime.

Los dos fragmento `{{html &quot;{{.}}&quot;}}` dicen que muestre el dato presentado a la plantilla en la cadena de consulta de la página web. El paquete de plantillas HTML automáticamente proporciona escape apropiado por lo tanto es seguro mostrar el texto.

El resto de la cadena de texto de la plantilla solo es el HTML a mostrar cuando se carga la página. Si esta es una explicación demasiado rápida, ve la documentación del paquete `template` para una descripción más minuciosa.

Y aquí lo tienes: un útil servidor web en unas cuantas líneas de código más algún dato derivado del texto HTML. Go es lo suficientemente potente para hacer que suceda mucho más en unas cuantas líneas.

Especificación del lenguaje de programación GO

Versión del 20 de Marzo, 2015

Puedes ver el documento original aquí

Introducción

Este es el manual de referencia para el lenguaje de programación Go. Para más información y otros documentos, ve [golang.org][golang].

Go es un lenguaje de propósito general diseñado con la programación de sistemas en mente. Es fuertemente tipado, cuenta con su propio recolector de basura y apoyo explícito a la programación concurrente. Los programas se construyen a partir de *paquetes*, cuyas propiedades permiten una eficiente gestión de las dependencias. Las implementaciones existentes utilizan el tradicional modelo de compilación/enlace para generar los binarios ejecutables.

La gramática es compacta y regular, lo que facilita el análisis de las herramientas automáticas tal como los entornos de desarrollo integrado.

Notación

La sintaxis se especifica utilizando la Forma Backus-Naur Extendida (EBNF por *Extended Backus-Naur Form*):

```
Producción  = nombre_producción "=" [ Expresión ] "." .
Expresión   = Alternativa { "|" Alternativa } .
Alternativa = Término { Término } .
Término     = nombre_producción | símbolo [ "…" símbolo ] | Grupo | Opción
              | Repetición .
Grupo       = "(" Expresión ")" .
Opción      = "[" Expresión "]" .
Repetición  = "{" Expresión "}" .
```

Las producciones son expresiones construidas a partir de términos y los siguientes operadores, en orden incremental de precedencia:

```
|   alternancia
()  agrupación
[]  opción (0 o 1 veces)
{}  repetición (de 0 a n veces)
```

Los nombres de producción en minúsculas se usan para identificar símbolos léxicos. Los no terminales están en MayúsculasIntercaladas. Los símbolos léxicos se encierran entre comillas dobles "" o comillas inversas ``.

La forma `a … b` representa el conjunto de caracteres de `a` hasta `b` como alternativas. Los puntos suspensivos horizontales `…` también se utilizan en otras partes de la especificación para denotar informalmente diversas enumeraciones o fragmentos de código que no se especifican más. El carácter `…` (a diferencia de los tres caracteres `...`) no es un símbolo del lenguaje Go.

Representación de código fuente

El código fuente es texto Unicode codificado en UTF-8. El texto no se canoniza, por lo que un único carácter acentuado es distinto del mismo carácter construido a partir de la combinación de un acento y una letra; estos son tratados como dos caracteres. Para simplificar, en este documento se utilizará el término no calificado *carácter* para referirse a un *carácter* Unicode en el texto fuente.

Cada carácter es distinto; por ejemplo, las letras mayúsculas y minúsculas son caracteres diferentes.

> Restricción de implementación: Por compatibilidad con otras herramientas, un compilador puede rechazar el carácter NUL (U+0000) en el texto fuente.
>
> Restricción de implementación: Por compatibilidad con otras herramientas, un compilador puede ignorar un byte de marca de orden (U+FEFF) codificado en UTF-8 si es el primer carácter Unicode en el texto fuente. Un byte de marca de orden se puede rechazar en cualquier otro lugar en la fuente.

Caracteres

Los siguientes términos se utilizan para referirse a clases específicas de caracteres Unicode:

```
nuevalínea         = /* el carácter Unicode U+000A */ .
carácter_unicode   = /* un carácter Unicode arbitrario excepto nuevalínea */ .
letra_unicode      = /* un carácter Unicode clasificado como "Letra" */ .
dígito_unicode     = /* un carácter Unicode clasificado como "Dígito decimal" */ .
```

El Estándar Unicode 7.0, en la Sección 4.5 "Categoría general" define un conjunto de categorías de caracteres. Go trata los caracteres en la categoría Lu, Ll, Lt, Lm o Lo como letras Unicode y los de la categoría Nd como dígitos Unicode.

Letras y dígitos

El carácter de subrayado `_` (U+005F) se considera una letra.

```
letra           = letra_unicode | "_" .
dígito_decimal  = "0" … "9" .
dígito_octal    = "0" … "7" .
dígito_hex      = "0" … "9" | "A" … "F" | "a" … "f" .
```

Elementos léxicos

Comentarios

Hay dos formas de comentarios:

1. *Comentarios de una línea* comienzan con la secuencia de caracteres `//` y terminan al final de la línea. Una línea de comentario actúa como una nuevalínea.
2. *Comentarios generales* comienzan con la secuencia de caracteres `/*` y continúan hasta alcanzar la secuencia de caracteres `*/`. Un comentario general que contiene uno o más saltos de línea actúa como un salto de línea, de lo contrario, actúa como un espacio.

Los comentarios no se anidan.

Símbolos

Los símbolos forman el vocabulario del lenguaje Go. Hay cuatro clases: *identificadores*, *palabras clave*, *operadores/delimitadores* y *literales*. El *espacio en blanco*, formado a partir de espacios (U+0020), tabuladores horizontales (U+0009), retornos de carro (U+000D) y saltos de línea (U+000A), se ignoran salvo los que separan a los símbolos que de otro modo se combinarían en un único símbolo. Además, un salto de línea o el final del archivo

pueden desencadenar la inserción de un punto y coma. Aunque la entrada se divide en símbolos, el siguiente símbolo es la secuencia de caracteres más larga que forma un símbolo válido.

Puntos y comas

La gramática formal utiliza un punto y coma `&quot;;&quot;` como terminador en una serie de resultados. Los programas Go pueden omitir la mayoría de esos puntos y comas utilizando las dos siguientes reglas:

1. Cuando la entrada se divide en símbolos, automáticamente se inserta un punto y coma en la cadena de componentes léxicos al final de una línea que no esté en blanco si el símbolo final de la línea es:
   - un identificador
   - un entero, número de coma flotante,

     ```
     [número imaginario](#imaginarios-literales),
     [rune](#rune-literales) o una
     [cadena de caracteres](#cadenas-literales) literal
     ```

   - una de las palabras clave `break`, `continue`, `fallthrough` o `return`
   - uno de los operadores y delimitadores `++`, `--`, `)`, `]` o `}`
2. Para permitir que las declaraciones complejas ocupen una sola línea, se puede omitir un punto y coma antes de un `&quot;)&quot;` o una `&quot;}&quot;` de cierre.

Para reflejar el uso idiomático, el código de los ejemplos de este documento omite el punto y coma utilizando estas reglas.

Identificadores

Los identificadores nombran entidades de programas tal como variables y tipos. Un identificador es una secuencia de una o más letras y dígitos. El primer carácter de un identificador debe ser una letra.

```
identificador = letra { letra | dígito_unicode } .
```

```
a
_x9
EstaVariableEsExportada
αβ
```

Algunos identificadores son predeclarados.

Palabras clave

Las siguientes palabras clave están reservadas y no se pueden utilizar como identificadores.

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

Operadores y delimitadores

Las siguientes secuencias de caracteres representan operadores, delimitadores y otros símbolos especiales:

```
+    &     +=    &=     &&    ==    !=    (    )
-    |     -=    |=     ||    <     <=    [    ]
*    ^     *=    ^=     <-    >     >=    {    }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     ...   .    :
     &^          &^=
```

Enteros literales

Un entero literal es una secuencia de dígitos que representa una constante entera. Un prefijo opcional establece una base no decimal: `0` para octal, `0x` o `0X` para hexadecimal. En hexadecimales literales, las letras `a-f` y `A-F` representan valores del 10 al 15.

```
ent_lit     = decimal_lit | octal_lit | hex_lit .
decimal_lit = ( "1" … "9" ) { dígito_decimal } .
octal_lit   = "0" { dígito_octal } .
hex_lit     = "0" ( "x" | "X" ) dígito_hex { dígito_hex } .
```

```
42
0600
0xBadFace
170141183460469231731687303715884105727
```

Números de coma flotante literales

Un número de coma flotante literal es una representación decimal de una constante de coma flotante. Tiene una parte entera, un punto decimal, una parte fraccional y una parte exponente. Las partes entera y fraccional comprenden los dígitos decimales; la parte del exponente es una `e` o `E` seguida de un exponente decimal opcionalmente con signo. Se puede omitir la parte entera o la parte fraccional; o bien se puede omitir el punto decimal o el exponente.

```
comaflotante_lit = decimales "." [ decimales ] [ exponente ] |
                   exponente decimal |
                   "." decimales [ exponente ] .
decimales        = dígito_decimal { dígito_decimal } .
exponente        = ( "e" | "E" ) [ "+" | "-" ] decimales .
```

```
0.
72.40
072.40  // == 72.40
2.71828
1.e+0
6.67428e-11
1E6
.25
.12345E+5
```

Imaginarios literales

Un imaginario literal es una representación decimal de la parte imaginaria de una constante compleja. Consiste de un número de coma flotante literal o un entero decimal seguido de la letra `i` minúscula.

```
imaginario_lit = (decimales | comaflotante_lit) "i" .
```

```
0i
011i  // == 11i
0.i
2.71828i
1.e+0i
6.67428e-11i
1E6i
.25i
.12345E+5i
```

Rune literales

Un rune literal representa una constante rune, un valor entero que identifica un carácter Unicode. Un rune literal se expresa como uno o más caracteres entre comillas simples. Dentro de las comillas, puede aparecer cualquier carácter excepto la comilla simple y el salto de línea. Un solo carácter entrecomillado representa el valor Unicode del carácter en sí mismo, mientras que las secuencias de varios caracteres que comienzan con una barra inversa codifican valores en varios formatos.

La forma más simple representa el único carácter dentro de las comillas; debido a que el texto fuente de Go son caracteres Unicode codificados en UTF-8, múltiples bytes codificados en UTF-8 pueden representar un único valor entero. Por ejemplo, la `&#39;a&#39;` literal contiene un solo byte que representa una `a` literal, el valor Unicode U+0061, el valor `0x61`, mientras que `&#39;ä&#39;` tiene dos bytes (`0xc3` `0xa4`) que representan una `a`-dieresis literal, U+00E4, el valor `0xe4`.

Varias barras inversas permiten escapar valores arbitrarios para que sean codificados como texto ASCII. Hay cuatro formas de representar el valor entero como una constante numérica: `\x` seguida de exactamente dos dígitos hexadecimales; `\u` seguida de exactamente cuatro dígitos hexadecimales, `\U` seguida de exactamente ocho dígitos hexadecimales y una barra inversa `\` seguida exactamente por tres dígitos octales. En cada caso el valor del literal es el valor representado por los dígitos en la base correspondiente.

Aunque todas estas representaciones resultan en un número entero, tienen diferentes rangos válidos. Los escapes octales deben representar un valor entre 0 y 255 inclusive. Los escapes hexadecimales satisfacen esta condición por su construcción. Los escapes `\u` y `\U` representan caracteres Unicode de modo que dentro de ellos algunos valores son ilegales, en particular los de arriba de `0x10FFFF` y mitades sucedáneas.

Después de una barra inversa, algunos escapes de un solo carácter representan valores especiales:

```
\a   U+0007 alerta o campana
\b   U+0008 retroceso
\f   U+000C avance de página
\n   U+000A avance de línea o nueva línea
\r   U+000D retorno de carro
\t   U+0009 tabulación horizontal
\v   U+000b tabulación vertical
\\   U+005c barra inversa
\'   U+0027 comilla simple (escape válido solo dentro de runes literales)
\"   U+0022 comilla doble (escape válido solo dentro de cadenas literales)
```

Todas las demás secuencias que comienzan con una barra inversa son literales ilegales dentro de rune.

```
rune_lit           = "'" ( valor_unicode | valor_byte ) "'" .
valor_unicode      = carácter_unicode | valor_u_bajo | valor_u_alto | carácter_escapado .
valor_byte         = byte_valor_octal | byte_valor_hex .
byte_valor_octal   = `\` dígito_octal dígito_octal dígito_octal .
byte_valor_hex     = `\` "x" dígito_hex dígito_hex .
valor_u_bajo       = `\` "u" dígito_hex dígito_hex dígito_hex dígito_hex .
valor_u_alto       = `\` "U" dígito_hex dígito_hex dígito_hex dígito_hex
                       dígito_hex dígito_hex dígito_hex dígito_hex .
carácter_escapado  = `\` ( "a" | "b" | "f" | "n" | "r" | "t" | "v" | `\` | "'" | `"` ) .
```

```
'a'
'ä'
'本'
'\t'
'\000'
'\007'
'\377'
'\x07'
'\xff'
'\u12e4'
'\U00101234'
'aa'         // ilegal: demasiados caracteres
'\xa'        // ilegal: muy pocos dígitos hexadecimales
'\0'         // ilegal: muy pocos dígitos octales
'\uDFFF'     // ilegal: medio sustituto
'\U00110000' // ilegal: carácter Unicode no válido
```

Cadenas literales

Una cadena literal representa una cadena constante obtenida a partir de la concatenación de una secuencia de caracteres. Hay dos formas: cadenas literales crudas y cadenas literales interpretadas. Las cadenas literales crudas son secuencias de caracteres entre comillas inversas ``. Dentro de las comillas, es lícito cualquier carácter excepto la comilla inversa. El valor de una cadena de texto literal es la cadena compuesta por los caracteres no interpretados (codificados implícitamente en UTF-8) entre comillas; en particular, las barras inversas no tienen un significado especial y la cadena puede contener saltos de línea. Los caracteres de retorno de carro ('\r') dentro de las cadenas literales se descartan del valor de la cadena cruda.

Las cadenas literales interpretadas son secuencias de caracteres entre comillas dobles `&quot;&quot;`. El texto entre las comillas, que por cierto, no puede contener saltos de línea, forma el valor literal, con barras inversas escapa caracteres interpretados como son los rune literales (excepto que `\&#39;` es ilegal y `\&quot;` es legal), con las mismas restricciones. Los tres dígitos octales (`\` *nnn*) y dos dígitos hexadecimales (`\x` *nn*)

representan *bytes* de escape individuales de la cadena resultante; todos los demás escapes representan la (posiblemente multi-byte) codificación UTF-8 de *caracteres* individuales. De este modo dentro de una cadena literal `\377` y `\xFF` representan un solo byte de valor `0xFF`=255, mientras `ÿ`, `\u00FF`, `\U000000FF` y `\xc3\xbf` representan los dos bytes `0xc3` y `0xbf` de la codificación UTF-8 del carácter U+00FF.

```
cadena_lit               = cadena_lit_cruda | cadena_lit_interpretada .
cadena_lit_cruda         = "`" { carácter_unicode | nuevalínea } "`" .
cadena_lit_interpretada  = `"` { valor_unicode | valor_byte } `"` .
```

```
`abc`  // misma que "abc"
`\n
\n`    // misma que "\\n\n\\n"
"\n"
""
"¡Hola, mundo!\n"
"日本語"
"\u65e5本\U00008a9e"
"\xff\u00FF"
"\uD800"       // ilegal: medio sustituto
"\U00110000"   // ilegal: carácter Unicode no válido
```

Todos estos ejemplos representan la misma cadena:

```
"日本語"                                 // UTF-8 texto ingresado
`日本語`                                 // UTF-8 texto ingresado como un
                                        // literal crudo
"\u65e5\u672c\u8a9e"                    // los caracteres Unicode explicitos
"\U000065e5\U0000672c\U00008a9e"        // los caracteres Unicode explícitos
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e"  // los bytes UTF-8 explícitos
```

Si el código fuente representa un carácter como dos caracteres, tal como implica una forma de combinar un acento y una letra, el resultado será un error si lo colocas en un rune literal (no es un único carácter) y aparecerá como dos caracteres si se coloca en una cadena literal.

Constantes

Hay *constantes lógicas*, *constantes rune*, *constantes enteras*, *constantes de coma flotante*, *constantes complejas* y *constantes de cadenas de caracteres*. Las constantes rune, enteras, de coma flotante y complejas se denominan colectivamente *constantes numéricas*.

Un valor constante está representado por un rune, número entero, de coma flotante, imaginario o cadena literal, un identificador denotando una constante, una expresión constante, una conversión con un resultado que es una constante o el valor resultante de alguna de las funciones integradas tal como `unsafe.Sizeof` aplicada a cualquier valor, `cap` o `len` aplicada a algunas expresiones, `real` e `imag` aplicada a una constante compleja y `complex` aplicada a constantes numéricas. Los valores lógicos de verdad están representados por las constantes predeclaradas `true` y `false`. El identificador predeclarado `iota` denota una constante entera.

En general, las constantes complejas son una forma de expresión constante y se explican en esa sección.

Las constantes numéricas representan valores de precisión arbitraria y no se desbordan.

Las constantes pueden ser tipadas o *sin tipo*. Las constantes literales, `true`, `false`, `iota` y ciertas expresiones constantes conteniendo solo operandos constantes sin tipo son sin tipo.

A una constante se le puede dar un tipo explícitamente por medio de una declaración de constante o conversión o implícitamente cuando se utiliza en una declaración de variable o una asignación o como un operando en una expresión. Es un error si el valor constante no se puede representar como un valor del tipo respectivo. Por ejemplo, a `3.0` se le puede dar cualquier tipo numérico tal como entero o de coma flotante, mientras que a `2147483648.0` (igual a `1&lt;&lt;31`) se le pueden dar los tipos `float32`, `float64` o `Uint32` pero no `int32` o `string`.

Una constante sin tipo tiene un *tipo predeterminado*, que es el tipo al que la constante se convierte implícitamente en contextos donde se requiere un valor tipado, por ejemplo, en una declaración corta de variable tal como `i := 0`, donde no hay ningún tipo explícito. El tipo predeterminado de una constante sin tipo es `bool`, `rune`, `int`, `float64`, `complex128` o `cadena`, respectivamente, dependiendo de si se trata de un valor lógico, rune, número entero, de coma flotante, complejo o cadena constante.

No hay constantes que denoten los valores infinitos IEEE-754 ni `not-a-number`, pero las funciones Inf, NaN, IsInf e IsNaN del paquete `math` devuelven y prueban estos valores en tiempo de ejecución.

> Restricción de implementación: A pesar de que en el lenguaje las constantes numéricas tienen precisión arbitraria, un compilador la puede implementar usando una representación interna con precisión limitada. Dicho esto, cada implementación debe:
>
> - Representar constantes enteras con al menos 256 bits.
> - Representar constantes de coma flotante, incluyendo las partes de una constante compleja, con una mantisa de al menos 256 bits y un exponente con signo de al menos 32 bits.
> - Dar un error si no puede representar con precisión una constante entera.
> - Dar un error si no puede representar una constante de coma flotante o compleja debido a desbordamiento.
> - Redondear a la constante representable más cercana si no puede representar una constante de coma flotante o compleja debido a los límites de precisión.

Estos requisitos se aplican tanto a las constantes literales como al resultado de la evaluación de expresiones constantes.

Variables

Una variable es una ubicación de almacenamiento para contener un *valor*. El conjunto de valores permisibles lo determina el *tipo de la variable*.

Una declaración de variable o, para los parámetros de función y resultados, la firma de una declaración de función o función literal reserva almacenamiento para una variable nombrada. Llamar a la función incorporada `new` o tomar la dirección de un literal compuesto reserva almacenamiento para una variable en tiempo de ejecución. Para referirse a tal variable anónima se usa una (posiblemente implícita) indirección de puntero.

Las variables *estructuradas* de tipo arreglo, sector y estructura tienen elementos y campos que se pueden encontrar individualmente. Cada uno de esos elemento actúa como una variable.

El *tipo estático* (o simplemente *tipo*) de una variable es el tipo determinado en su declaración, el tipo provisto en la llamada a `new` o al literal compuesto, o el tipo de un elemento de una variable estructurada. Las variables de tipo interfaz también tienen un *tipo dinámico* distinto, que es el tipo concreto del valor asignado a la variable en tiempo de ejecución (a menos que el valor sea el identificador predeclarado `nil`, mismo que no tiene tipo). El tipo dinámico puede variar durante la ejecución, pero los valores almacenados en variables interfaz siempre son asignables al tipo estático de la variable.

```
var x interface{}  // x es nil y tiene una interfaz de tipo estático
var v *T           // v tiene un valor nil, tipo estático *T
x = 42             // x tiene valor 42 y el tipo dinámico int
x = v              // x tiene un valor (*T)(nil) y el tipo dinámico *T
```

El valor de una variable se recupera haciendo referencia a la variable en una expresión; este es el valor más reciente asignado a la variable. Si una variable aún no tiene asignado un valor, su valor es el valor cero para su tipo.

Tipos

Un tipo determina el conjunto de valores y operaciones específicas para los valores de ese tipo. Los tipos podrán ser *nombrados* o *anónimos*. Los tipos con nombre se especifican mediante una (posiblemente cualificada) declaración *type nombre*; los tipos anónimos se especifican usando un *tipo literal*, que compone un nuevo tipo de los tipos existentes.

```
Tipo         = NombreTipo | TipoLit | "(" Tipo ")" .
NombreTipo   = identificador | IdentiCualificado .
TipoLit      = TipoArreglo | TipoEstructura | TipoPuntero | TipoFunción | TipoInterfaz |
               TipoSector | TipoMapa | TipoCanal .
```

Las instancias nombradas de tipos lógicos, numéricos y cadena son predeclaradas. Los *tipos compuestos* —arreglo, estructura, puntero, función, interfaz, sector, mapa y canal— se pueden construir mediante tipos literales.

Cada tipo `T` tiene un *tipo subyacente*: Si `T` es uno de los tipos predeclarados lógico, numérico, cadena o un tipo literal, el correspondiente tipo subyacente es `T` en sí mismo. De lo contrario, el tipo `T` subyacente es el tipo subyacente del tipo al que `T` se refiere en su declaración de tipo.

```
type T1 string
   type T2 T1
   type T3 []T1
   type T4 T3
```

El tipo subyacente de `string`, `T1` y `T2` es `string`. El tipo subyacente de `[]T1`, `T3` y `T4` es `[]T1`.

Conjunto de métodos

Un tipo puede tener un *conjunto de métodos* asociado a él. El conjunto de métodos de un tipo interfaz es su interfaz. El conjunto de métodos de cualquier otro tipo `T` se compone de todos los métodos declarados con el tipo receptor `T`. El conjunto de métodos del puntero al tipo correspondiente a `*T` es el conjunto de todos los métodos declarados con el receptor `*T` o `T` (es decir, que también contiene el conjunto de métodos de `T`). Las nuevas reglas se aplican a estructuras que contienen campos anónimos, como se describe en la sección sobre tipos estructura. Cualquier otro tipo tiene un conjunto de métodos vacío. En un conjunto de métodos, cada método debe tener un nombre de método único no blanco.

El conjunto de métodos de un tipo determina las interfaces que el tipo implementa y los métodos que se pueden llamar mediante un receptor de ese tipo.

Tipos lógicos

Un *tipo lógico* representa el conjunto de valores lógicos de verdad indicados por las constantes predeclaradas `true` y `false`. El tipo lógico predeclarado es `bool`.

Tipos numéricos

Un *tipo numérico* representa conjuntos de valores enteros o de coma flotante. Los tipos numéricos predeclarados independientes de la arquitectura son:

```
uint8       el conjunto de todos los enteros sin signo de 8 bits (0 a 255)
uint16      el conjunto de todos los enteros sin signo de 16 bits (0 a
                                                                  65535)
uint32      el conjunto de todos los enteros sin signo de 32 bits (0 a
                                                              4294967295)
uint64      el conjunto de todos los enteros sin signo de 64 bits
                                                 (0-18446744073709551615)

int8        el conjunto de todos los enteros con signo de 8 bits (-128 a
                                                                    127)
int16       el conjunto de todos los enteros con signo de 16 bits (-32768
                                                                 a 32767)
int32       el conjunto de todos los enteros con signo de 32 bits
                                               (-2147483648 a 2147483647)
int64       el conjunto de todos los enteros con signo de 64 bits
                          (-9223372036854775808 a 9223372036854775807)

float32     el conjunto de todos los números IEEE-754 de coma flotante de
                                                                 32 bits
float64     el conjunto de todos los números IEEE-754 de coma flotante de
                                                                 64 bits

complex64   el conjunto de todos los números complejos float32 con partes
                                                        real e imaginaria
complex128  el conjunto de todos los números complejos float64 con partes
                                                        real e imaginaria

byte        alias para uint8
rune        alias para int32
```

El valor de un entero *n*-bit es *n* bits de ancho y se representa usando aritmética de complemento a dos.

También hay una serie de tipos numéricos predeclarados con tamaños específicos de la implementación:

```
uint     entre 32 o 64 bits
int      mismo tamaño que uint
uintptr  un entero sin signo suficientemente grande para almacenar
         los bits no interpretados de un valor puntero
```

Para evitar problemas de portabilidad todos los tipos numéricos son distintos a excepción de `byte`, que es un alias para `uint8` y `rune`, que es un alias para `int32`. Se necesita una conversión cuando diferentes tipos numéricos se mezclan en una expresión o asignación. Por ejemplo, `int32` e `int` no son del mismo tipo a pesar de que pueden tener el mismo tamaño en una arquitectura particular.

Tipos cadena de caracteres (*string*)

Un *tipo cadena* representa el conjunto de valores de cadena de caracteres. Un valor de cadena es una (posiblemente vacía) secuencia de bytes. Las cadenas son inmutables: una vez creadas, no es posible cambiar el contenido de una cadena. El tipo predeclarado de cadena es `string`.

La longitud de una cadena `s` (su tamaño en bytes) se puede descubrir usando la función incorporada `len`. La longitud es una constante en tiempo de compilación si la cadena es una constante. Se puede acceder a los bytes de una cadena por índices enteros desde 0 hasta `len(s)-1`. Es ilegal tomar la dirección de tal elemento; si `s[i]` es el `i` enésimo byte de una cadena, `&s[i]` no es válido.

Tipos arreglo

Un arreglo o vector es una secuencia numerada de elementos de un solo tipo, llamado el tipo del elemento. Al número de elementos se le conoce como `longitud` y nunca es negativo.

```
TipoArreglo      = "[" LongitudArreglo "]" TipoElemento .
LongitudArreglo  = Expresión .
TipoElemento     = Tipo .
```

La longitud es parte del tipo arreglo; se debe evaluar a una constante no negativa representable por un valor de tipo `int`. La longitud del arreglo `a` se puede descubrir usando la función incorporada `len`. Los elementos se pueden encontrar por índices enteros desde 0 hasta `len(a)-1`. Los tipos arreglo siempre son de una sola dimensión, pero pueden estar compuestos para formar tipos multidimensionales.

```
[32]byte
[2*N] struct { x, y int32 }
[1000]*float64
[3][5]int
[2][2][2]float64  // igual que [2]([2]([2]float64))
```

Tipos sector

Un sector es un descriptor de un segmento contiguo de un *arreglo subyacente* y proporciona acceso a una secuencia numerada de elementos de ese arreglo. Un tipo sector denota el conjunto de todos los sectores de arreglos del tipo de elemento. El valor de un sector sin iniciar es `nil`.

```
TipoSector = "[" "]" TipoElemento .
```

Al igual que los arreglos, los sectores son indexables y tienen una longitud. La longitud de un sector `s` se puede descubrir por medio de la función incorporada `len`; a diferencia de los arreglos puede cambiar durante la ejecución. Los elementos se pueden encontrar por medio de índices enteros desde 0 hasta `len(s)-1`. El índice de un elemento dado del sector puede ser menor que el índice del mismo elemento del arreglo subyacente.

Un sector, una vez iniciado, siempre está asociado con un arreglo subyacente que mantiene sus elementos. Por lo tanto, un sector comparte almacenamiento con su arreglo y con otros sectores del mismo arreglo; por el contrario, distintos arreglos siempre representan almacenamiento distinto.

El arreglo subyacente a un sector se puede extender más allá del final del sector. La *capacidad* es una medida de esa extensión: es la suma de la longitud del sector y la longitud del arreglo más allá del sector; puedes crear un sector de mayor longitud que la capacidad *secccionando* uno nuevo desde el sector original. La capacidad de un sector `a` se puede descubrir usando la función incorporada `cap(a)`.

Un nuevo valor de sector iniciado, para un tipo de elemento `T` dado se crea usando la función integrada `make`, que toma un tipo sector y parámetros especificando la longitud y opcionalmente la capacidad. Un sector creado con `make` siempre asigna un nuevo arreglo oculto al cual el valor del sector devuelto refiere. Es decir, al ejecutar:

```
make([]T, longitud, capacidad)
```

produce el mismo sector como si asignaras un arreglo y después lo seccionaras, por lo tanto estas dos expresiones son equivalentes:

```
make([]int, 50, 100)
new([100]int)[0:50]
```

Como los arreglos, los sectores siempre son de una sola dimensión, pero pueden estar compuestos para construir objetos de mayores dimensiones. Con arreglos de arreglos, los arreglos internos siempre son, por construcción, de la misma longitud; sin embargo con los sectores de sectores (o arreglos de sectores), las longitudes internas pueden variar dinámicamente. Por otra parte, los sectores internos se deben iniciar individualmente.

Tipos estructura

Una estructura es una secuencia de elementos nombrados, llamados campos, cada uno de los cuales tiene un nombre y un tipo. Los nombres de campo se pueden especificar explícitamente (ListaIdentificador) o implícitamente (CampoAnónimo). En una estructura, no vacía los nombres de los campos deben ser únicos.

```
TipoEstructura   = "estructura" "{" { DeclaraCampo ";" } "}" .
DeclaraCampo     = ( ListaIdentificador Tipo | CampoAnónimo ) [ Etiqueta ] .
CampoAnónimo     = ["*"] NombreTipo .
Etiqueta         = cadena_lit .
```

```
// Una estructura vacía.
struct {}

// Una estructura con 6 campos.
struct {
    x, y int
    u float32
    _ float32  // relleno
    A *[]int
    F func()
}
```

Un campo declarado con un tipo pero no un nombre de campo explícito es un *campo anónimo*, también llamado campo *incrustado* o una incrustación del tipo en la estructura. Un tipo de campo se tiene que especificar como un nombre de tipo `T` o como un puntero a un nombre de tipo no interfaz `*T` y `T` en sí mismo no puede ser un tipo puntero. El nombre de tipo no cualificado actúa como el nombre del campo.

```
// Una estructura con cuatro campos anónimos de tipo T1, *T2,
// P.T3 y *P.T4
struct {
    T1       // el nombre del campo es T1
    *T2      // el nombre del campo es T2
    P.T3     // el nombre del campo es T3
    *P.T4    // el nombre del campo es T4
    x, y int // los nombres de campo son x e y
}
```

La siguiente declaración es ilegal ya que los nombres de campo deben ser únicos en un tipo estructura:

```
struct {
    T     // conflicto con los campos anónimos *T y *P.T
    *T    // conflicto con los campos anónimos T y *P.T
    *P.T  // conflicto con los campos anónimos T y *T
}
```

Un campo o método `f` de un campo anónimo en una estructura `x` se dice que fue *promovido* si `x.f` es un selector legal que denota ese campo o método `f`.

Los campos promovidos actúan como campos ordinarios de una estructura, excepto que no se pueden utilizar como nombres de campo en los literales compuestos de la estructura.

Dado un tipo estructura `S` y un tipo llamado `T`, los métodos promovidos se incluyen en el conjunto de métodos de la estructura de la siguiente manera:

- Si `S` contiene un campo anónimo `T`, el conjunto de métodos de ambos `S` y `*S` incluyen métodos promovidos con receptor `T`. El conjunto de métodos de `*S` también incluye métodos promovidos con receptor `*T`.
- Si `S` contiene un campo anónimo `*T`, ambos conjuntos de métodos de `S` y `*S` incluyen métodos promovidos con receptor `T` o `*T`.

Una declaración de campo puede estar seguida por una cadena literal opcional que hará las veces de *etiqueta*, que se convierte en un atributo para todos los campos en la declaración del campo correspondiente. Las etiquetas se hacen visibles a través de una interfaz de reflexión y toman parte en la identidad de tipo para estructuras pero de lo contrario se omiten.

```
// Una estructura que corresponde al protocolo TimeStamp con búfer.
// Las cadenas etiqueta definen el protocolo búfer para el número
// de campos.
struct {
    microsec    uint64 "campo 1"
    servidorIP6 uint64 "campo 2"
    proceso     string "campo 3"
}
```

Tipos puntero

Un tipo puntero denota el conjunto de todos los punteros a las variables de un determinado tipo, llamado el *tipo base* del puntero. El valor de un puntero no iniciado es `nil`.

```
TipoPuntero = "*" TipoBase .
TipoBase    = Tipo .
```

```
*Punto
*[4]int
```

Tipos función

Un tipo función indica el conjunto de todas las funciones con los mismos parámetros y tipo de resultados. El valor de una variable no iniciada de tipo función es `nil`.

```
TipoFunción        = "func" Firma .
Firma              = Parámetros [ Resultado ].
Resultado          = Parámetros | Tipo .
Parámetros         = "(" [ ListaDeParámetros [ "," ] ] ")".
ListaDeParámetros  = DeclaParámetro {"," DeclaParámetro }.
DeclaParámetro     = [ ListaIdentificador ] ["..."] Tipo .
```

Dentro de una lista de parámetros o resultados, los nombres (ListaIdentificador), deberían estar todos presentes o todos ausentes. Si están presentes, cada nombre se refiere a un elemento (parámetro o resultado) del tipo especificado y todos los nombres no vacíos de la firma deben ser únicos. Si esta ausente, cada tipo representa un elemento de ese tipo. Las listas de parámetros y resultados siempre están entre paréntesis excepto si hay exactamente un resultado anónimo que se pueda escribir como un tipo sin paréntesis.

El parámetro final en una firma de función puede tener un tipo prefijado con `...`. Una función con tal parámetro se llama *variadica* y se puede invocar con cero o más argumentos para ese parámetro.

```
func()
func(x int) int
func(a, _ int, z float32) bool
func(a, b int, z float32) (bool)
func(prefijo string, valores ...int)
func(a, b int, z float64, opcional ...interface{}) (éxitoso bool)
func(int, int, float64) (float64, *[]int)
func(n int) func(p *T)
```

Tipos interfaz

Un tipo interfaz especifica un conjunto de métodos conocido como su *interfaz*. Una variable de tipo interfaz puede almacenar un valor de cualquier tipo con un conjunto de métodos que es algún superconjunto de la interfaz. Tal tipo se dice que *implementa la interfaz*. El valor de una variable no iniciada de tipo interfaz es `nil`.

```
TipoInterfaz         = "interface" "{" { EspecificaMétodo ";" } "}" .
EspecificaMétodo     = NombreMétodo  Firma | NombreTipoInterfaz .
NombreMétodo         = identificador .
NombreTipoInterfaz   = NombreTipo .
```

Al igual que con todos los conjuntos de métodos, en un tipo interfaz, cada método debe tener un nombre único no vacío.

```
// Una sencilla interfaz Documento
interface {
    Lee(b Buffer) bool
    Escribe(b Buffer) bool
    Cierra()
}
```

Más de un tipo puede implementar una interfaz. Por ejemplo, si dos tipos `S1` y `S2` tienen el conjunto de métodos:

```
func (p T) Lee(b Buffer) bool { return … }
func (p T) Escribe(b Buffer) bool { return … }
func (p T) Cierra() { … }
```

(Donde `T` representa ya sea a `S1` o a `S2`), entonces la interfaz `Documento` la implementan tanto `S1` como `S2` , independientemente de cuales otros métodos puedan tener o compartir `S1` y `S2` .

Un tipo implementa cualquier interfaz que comprenda algún subconjunto de sus métodos y por lo tanto puede implementar varias interfaces distintas. Por ejemplo, todos los tipos implementan la *interfaz vacía*:

```
interface{}
```

Del mismo modo, considera esta especificación de interfaz que aparece dentro de una declaración de tipo para definir una interfaz llamada `Archivero` :

```
type Archivero interface {
    Bloquea()
    Desbloquea()
}
```

Si `S1` y `S2` también implementan

```
func (p T) Bloquea() { … }
func (p T) Desbloquea() { … }
```

implementan la interfaz `Archivero` así como la interfaz `Documento`.

Una interfaz `T` puede utilizar un (posiblemente cualificado) nombre de tipo interfaz `E` en lugar de una especificación de método. A esto se le conoce como *incrustación* de la interfaz `E` en `T`; esta añade todos los métodos (exportados y no exportados) de `E` a la interfaz `T`.

```
type LectorEscritor interface {
    Lee(b Buffer) bool
    Escribe(b Buffer) bool
}

type Documento interface {
    LectorEscritor  // igual que añadir los métodos de LectorEscritor
    Archivero       // igual que añadir los métodos de Archivero
    Cierra()
}

type DocumentoArchivado interface {
    Archivero
    Documento   // ilegal: Archivero, Desbloquea no es único
    Bloquea()   // ilegal: Bloquea no es único
}
```

Un tipo interfaz `T` no se puede incrustar a sí mismo o a cualquier tipo interfaz que incorpore `T`, recursivamente.

```
// ilegal: Mal no se puede incrustar
type Mal interface {
    Mal
}

// ilegal: Mal1 no se puede incrustar a sí misma usando Mal2
type Mal1 interface {
    Mal2
}

type Mal2 interface {
    Mal1
}
```

Tipos mapa

Un mapa es un grupo no ordenado de elementos de un tipo, conocido como el tipo del elemento, indexado por un conjunto de *claves* únicas de otro tipo, llamada la clave del tipo. El valor de un mapa sin iniciar es `nil`.

```
TipoMapa   = "map" "[" TipoClave "]" TipoElemento .
TipoClave  = Tipo .
```

Los operadores de comparación `==` y `!=` deben estar completamente definidos como operandos del tipo clave; por lo que, el tipo clave no debe ser una función, mapa o sector. Si el tipo clave es un tipo interfaz, estos operadores de comparación los deben definir los valores de las claves dinámicas; un fallo provocará pánico en tiempo de ejecución.

```
map[string]int
map[*T]struct{ x, y float64 }
map[string]interfaz{}
```

El número de elementos del mapa se conoce como su longitud. En un mapa `m` su longitud se puede descubrir usando la función incorporada `len` y puede cambiar durante la ejecución. Puedes añadir elementos durante la ejecución utilizando asignaciones y se recuperan con expresiones índice; se pueden remover con la función incorporada `delete`.

Un nuevo valor de mapa vacío se crea usando la función integrada `make`, que toma el tipo mapa y un indicio de capacidad opcional como argumentos:

```
make(map[string]int)
make(map[string]int, 100)
```

La capacidad inicial no inmoviliza su tamaño: los mapas crecen para acomodar el número de elementos almacenados en ellos, a excepción de los mapas `nil`. Un mapa `nil` es equivalente a un mapa vacío salvo que no se le pueden añadir elementos.

Tipos canal

Un canal proporciona un mecanismo para ejecutar funciones concurrentemente para comunicarse enviando y recibiendo valores de un tipo de elemento especificado. El valor de un canal sin iniciar es `nil`.

```
TipoCanal = ("chan" | "chan" "<-" | "<-" "chan") TipoElemento .
```

El operador opcional `<-` especifica la *dirección* del canal, *enviar* o *recibir*. Si no se da una dirección, el canal es *bidireccional*. Un canal puede estar limitado por conversión o asignación solo para enviar o solo para recibir.

```
chan T          // se puede utilizar para enviar y recibir valores
                // de tipo T
chan<- float64  // solo se puede utilizar para enviar float64
<-chan int      // solo se puede utilizar para recibir ints
```

El operador `<-` posibilita la asociación con el `canal` de la izquierda:

```
chan<- chan int    // igual que chan<- (chan int)
chan<- <-chan int  // igual que chan<- (<-chan int)
<-chan <-chan int  // igual que <-chan (<-chan int)
chan (<-chan int)
```

Puedes crear un nuevo valor de canal iniciado usando la función integrada `make`, que toma como argumentos el tipo canal y una capacidad opcional:

```
make(chan int, 100)
```

La capacidad, en número de elementos, establece el tamaño del búfer en el canal. Si la capacidad es cero o está ausente, el canal es sin búfer y la comunicación tiene éxito solo cuando ambos, el remitente y el receptor están listos. De lo contrario, el canal se almacenará en un búfer y la comunicación será éxitosa sin bloqueo si el búfer no está lleno (envía) o no está vacío (recibe). Un canal `nil` nunca está listo para comunicación.

Un canal se puede cerrar con la función incorporada `close`. La forma de asignación de múltiples valores del operador receptor informa si un valor recibido fue enviado antes de que se cerrara el canal.

Un solo canal se puede utilizar en instrucciones de envío, operaciones de recepción y llamadas a las funciones incorporadas `cap` y `len` por una serie de rutinasgo sin más sincronización. Los canales actúan como colas PEPS (*primero-en-entrar-primero-en-salir*). Por ejemplo, si una rutinago envía los valores en un canal y una segunda rutinago los recibe, los valores se reciben en el orden enviado.

Propiedades de tipos y valores

Identidad de tipo

Dos tipos son o bien *idénticos* o *diferentes*.

Dos tipos nombrados son idénticos cuando sus nombres de tipo se originan en la misma EspeciTipo. Un tipo nombrado y uno anónimo siempre son diferentes. Dos tipos anónimos son idénticos si los tipos literales correspondientes son idénticos, es decir, si tienen la misma estructura literal y los componentes correspondientes tienen tipos idénticos. En detalle:

- Dos tipos arreglo son idénticos si tienen elementos de tipos idénticos y los arreglos tiene la misma longitud.
- Dos tipos sector son idénticos si el tipo de sus elementos es idéntico.
- Dos tipos estructura son idénticos si tienen la misma secuencia de campos y si los campos correspondientes tienen los mismos nombres, tipos y etiquetas idénticas. Dos campos anónimos se considera que tienen el mismo nombre. Nombres de campo en minúsculas de diferentes paquetes siempre son diferentes.
- Dos tipos puntero son idénticos si tienen tipos base idénticos.
- Dos tipos función son idénticos si tienen el mismo número de parámetros y valores de resultado, los tipos de parámetros y resultados correspondientes son idénticos y, o bien ambas funciones son variadicas o ninguna de las dos. Los nombres de parámetros y resultados no es necesario que coincidan.
- Dos tipos interfaz son idénticos si tienen el mismo conjunto de métodos con los mismos nombres y tipos función idénticos. Los nombres de método en minúsculas de diferentes paquetes siempre son diferentes. El orden de los métodos es irrelevante.
- Dos tipos mapa son idénticos si tienen tipos de claves y valores idénticos.
- Dos tipos canal son idénticos si tienen tipos de valores idénticos y la misma dirección.

Dadas las declaraciones:

```
type (
    T0 []string
    T1 []string
    T2 struct{ a, b int }
    T3 struct{ a, c int }
    T4 func(int, float64) *T0
    T5 func(x int, y float64) *[]string
)
```

estos tipos son idénticos:

```
T0 y T0
[]int e []int
struct{ a, b *T5 } y struct{ a, b *T5 }
func(x int, y float64) *[]string y func(int, float64) (resultado *[]string)
```

`T0` y `T1` son diferentes porque tienen nombres de tipo con declaraciones distintas; `func(int, float64) *T0` y `func(x int, y float64) *[]string` son diferentes porque `T0` es diferente de `[]string`.

Asignabilidad

Un valor `x` es *asignable* a una variable de tipo `T` ("`x` es asignable a `T`") en cualquiera de estos casos:

- el tipo `x` es idéntico a `T`.
- los tipos `x`, `V` y `T` tienen idénticos tipos subyacentes y al menos uno de `V` o `T` no es un tipo nombrado.
- `T` es un tipo interfaz y `x` implementa a `T`.
- `x` es un valor de canal bidireccional, `T` es un tipo canal, los tipos `x`, `V` y `T` tienen tipos de elemento idénticos y al menos uno de `V` o `T` no es un tipo nombrado.
- `x` es el identificador predeclarado `nil` y `T` es un tipo puntero, función, sector, mapa, canal o interfaz.
- `x` es una constante sin tipo representable por un valor de tipo `T`.

Bloques

Un *bloque* es una secuencia posiblemente vacía de declaraciones e instrucciones entre llaves.

```
Bloque            = "{" ListaInstrucciones "}" .
ListaInstrucciones = { Declaración ";" } .
```

Además de los bloques explícitos en el código fuente, hay bloques implícitos:

1. El *bloque universal* abarca todo el texto fuente Go.
2. Cada paquete tiene un *bloque de paquete* que contiene todo el texto fuente Go para ese paquete.
3. Cada archivo tiene un *bloque de archivo* que contiene todo el texto fuente Go en ese archivo.
4. Cada declaración "if", "for" y "switch" se considera que están en su propio bloque implícito.
5. Cada cláusula de una declaración "switch" o "select" actúa como un bloque implícito.

Bloques anidados e influencia y determinación de ámbito.

Declaraciones y ámbito

Una *declaración* vincula a un identificador no blanco con una constante, tipo, variable, función, etiqueta o paquete. Cada identificador en un programa se debe declarar. Ningún identificador se puede declarar dos veces en el mismo bloque y ningún identificador se puede declarar en ambos el bloque de archivo y el bloque de paquete.

El identificador blanco se puede utilizar como cualquier otro identificador en una declaración, pero no introduce un vínculo y por tanto no se ha declarado. En el bloque de paquete, el identificador `init` solo lo pueden utilizar las declaraciones de función `init` e igual que el identificador blanco este no introduce un nuevo vínculo.

```
Declaración        = DeclaConst | DeclaTipo | DeclaVar .
DeclaNivelSuperior = Declaración | DeclaFunción | DeclaMétodo .
```

El *ámbito* de un identificador declarado es la extensión del texto fuente en el que el identificador denota la constante, tipo, variable, función, etiqueta o paquete especificado.

Go tiene ámbito léxico utilizando bloques:

1. El alcance de un identificador predeclarado es el bloque universal.
2. El ámbito de un identificador que denota una constante, tipo, variable o función (pero no método) declarado en el nivel superior (fuera de cualquier función) es el bloque del paquete.
3. El alcance del nombre del paquete de un paquete importado es el bloque de archivo del archivo que contiene la declaración de importación.
4. El ámbito de un identificador que denota un método receptor, parámetro de función o variable de resultado es el cuerpo de la función.
5. El ámbito del identificador de una constante o variable declarada dentro de una función se inicia al final de la EspeConst o EspeVar (DeclaVarCorta para declaraciones cortas de variables) y termina al final de la más interna que contiene el bloque.
6. El alcance de un identificador de tipo declarado dentro de una función comienza en la EspeciTipo del identificador y termina al final del bloque más interno.

Un identificador declarado en un bloque se puede volver a declarar en un bloque interno. Mientras que el identificador de la declaración interna esté a su alcance, este se refiere a la entidad declarada por la declaración interna.

La cláusula package no es una declaración; el nombre del paquete no aparece en ningún ámbito. Su propósito es identificar los archivos pertenecientes a un mismo paquete y especificar el nombre del paquete predeterminado para las declaraciones de importación.

Ámbito de etiquetas

Las etiquetas son declaradas por las instrucciones etiquetadas y se utilizan en las instrucciones "break", "continue" y "goto". Es ilegal definir una etiqueta que nunca se utiliza. En contraste con otros identificadores, las etiquetas no se restringen a bloques y no entran en conflicto con los identificadores que no son etiquetas. El ámbito de una etiqueta es el cuerpo de la función en la que se declara y excluye el cuerpo de cualquier función anidada.

Identificador blanco

El *identificador blanco* está representado por el carácter de subrayado `_`. Sirve como un marcador de posición anónimo en lugar de un identificador regular (no blanco) y tiene un significado especial en declaraciones, como un operando y en las asignaciones.

Identificadores predeclarados

Los siguientes identificadores están declarados implícitamente en el bloque universal:

```
Tipos:
    bool byte error complex128 complex64 float32 float64
    int int8 int16 int32 int64 runa string
    uint uint8 uint16 uint32 uint64 uintptr

Constantes:
    true false iota

Valor cero:
    nil

Funciones:
    append cap close complex copy delete imag len
    make new panic print println real recover
```

Identificadores exportados

Un identificador se puede *exportar* para permitir el acceso al mismo desde otro paquete. Un identificador se exporta si se cumplen las siguientes condiciones:

1. El primer carácter del nombre del identificador es una letra Unicode en mayúscula (Unicode clase "Lu"); y
2. El identificador se declara en el bloque de paquete o se trata de un nombre de campo o un nombre de método.

Todos los otros identificadores no se exportan.

Unicidad de identificadores

Dado un conjunto de identificadores, un identificador se llama *único* si es *diferente* de todos los otros en el conjunto. Dos identificadores son diferentes si se escriben diferente o si aparecen en diferentes paquetes y no se exportan. De lo contrario, son los mismos.

Declaración de constantes

Una declaración de constante vincula una lista de identificadores (los nombres de las constantes) a los valores de una lista de expresiones constantes. El número de identificadores debe ser igual al número de expresiones y el e*n*ésimo identificador de la izquierda está unido al valor de la e*n*ésima expresión de la derecha.

```
DeclaConst         = "const" ( EspeConst | "(" { EspeConst ";" } ")" ) .
EspeConst          = ListaIdentificador [ [ Tipo ] "=" ListaExpresión ] .
ListaIdentificador = identificador { "," identificador } .
ListaExpresión     = Expresión { "," Expresión } .
```

Si el tipo está presente, todas las constantes toman el tipo especificado y las expresiones deben ser asignables a ese tipo. Si se omite el tipo, las constantes toman los tipos individuales de las expresiones correspondientes. Si los valores de la expresión sin tipo son constantes, las constantes declaradas permanecen sin tipo y los identificadores constantes denotan los valores constantes. Por ejemplo, si la expresión es un literal de coma flotante, el identificador constante denota una constante de coma flotante, incluso si la parte fraccionaria literal es cero.

```
const Pi float64 = 3.14159265358979323846
const cero = 0.0         // constante de coma flotante sin tipo
const (
    tamaño int64 = 1024
    fda          = -1    // constante entera sin tipo
)

const a, b, c = 3, 4, "foo"  // a = 3, b = 4, c = "foo", constantes
                             // enteras sin tipo y cadena
const u, v float32 = 0, 3    // u = 0.0, v = 3.0
```

Dentro de paréntesis la lista de declaración `const` y la lista de expresiones se puede omitir en cualquiera menos la primera declaración. Tal lista vacía es equivalente a la sustitución textual de la primer lista de expresiones no vacía anterior y su tipo si los hubiere. La omisión de la lista de expresiones es equivalente a la repetición de la lista anterior. El número de

identificadores debe ser igual al número de expresiones en la lista anterior. Junto con el generator de constantes `iota` este mecanismo permite la declaración ligera de valores secuenciales:

```
const (
    Domingo      = iota
    Lunes
    Martes
    Miércoles
    Jueves
    Viernes
    DíaDelPartido
    númeroDeDías // esta constante no se exporta
)
```

Iota

Dentro de una declaración de constante, el identificador predeclarado `iota` representa constantes enteras sucesivas sin tipo. Se restablece a 0 cada vez que la palabra reservada `const` aparece en la fuente y se incrementa después de cada EspeConst. Se puede utilizar para construir un conjunto de constantes relacionadas:

```
const (  // iota se restablece a 0
    c0 = iota  // c0 == 0
    c1 = iota  // c1 == 1
    c2 = iota  // c2 == 2
)

const (
    a = 1 << iota  // a == 1 (iota se ha restablecido)
    b = 1 << iota  // b == 2
    c = 1 << iota  // c == 4
)

const (
    u         = iota * 42  // u == 0     (constante entera sin tipo)
    v float64 = iota * 42  // v == 42.0  (constante float64)
    w         = iota * 42  // w == 84    (constante entera sin tipo)
)

const x = iota  // x == 0 (iota se ha restablecido)
const y = iota  // y == 0 (iota se ha restablecido)
```

Dentro de una ListaExpresión, el valor de cada `iota` es el mismo, ya que solo se incrementa después de cada EspeConst:

```
const (
    bit0, mask0 = 1 << iota, 1<<iota - 1  // bit0 == 1, mask0 == 0
    bit1, mask1                           // bit1 == 2, mask1 == 1
    _, _                                  // salta iota == 2
    bit3, mask3                           // bit3 == 8, mask3 == 7
)
```

Este último ejemplo explota la repetición implícita de la lista de expresiones no vacía anterior.

Declaración de tipos

Una declaración de tipo vincula un identificador, el *nombre de tipo*, a un nuevo tipo que tiene el mismo tipo subyacente que un tipo existente y las operaciones definidas para el tipo existente también están definidas para el nuevo tipo. El nuevo tipo es diferente del tipo existente.

```
DeclaTipo    = "type" ( EspeciTipo | "(" { EspeciTipo ";" } ")" ) .
EspeciTipo   = identificador de Tipo .
```

```
type ArregloEntero [16]int

type (
    Punto struct{ x, y float64 }
    Polar Punto
)

type ÁrbolDeNodo struct {
    izquierda, derecha *ÁrbolDeNodo
    valor *Comparable
}

type Bloque interface {
    TamañoBloque() int
    Cifra(fnt, dst []byte)
    Descifra(fnt, dst []byte)
}
```

El tipo declarado no hereda ningún método ligado al tipo existente, pero el conjunto de métodos de un tipo interfaz o de elementos de un tipo compuesto se mantiene sin cambios:

```
// Un Excluyente es un tipo de dato con dos métodos, Bloquea y Desbloquea.
type struct Excluyente { /* Campos de exclusión mutua (mutex) */ }

func (m *Excluyente) Bloquea() { /* Implementación de Bloquea */ }
func (m *Excluyente) Desbloquea() { /* Implementación de Desbloquea */ }

// NuevoExcluyente tiene la misma composición que un objeto de
// exclusión mutua pero su conjunto de métodos está vacío.
type NuevoExcluyente Excluyente

// El conjunto de métodos del tipo base de PuntExcluyente permanece sin
// cambios, pero el conjunto de métodos está vacío.
type PuntExcluyente *Excluyente

// El conjunto de métodos de *ExcluyenteImprimible contiene los métodos
// Bloquea y Desbloquea unido a su campo anónimo Excluyente.
type ExcluyenteImprimible struct {
    Excluyente
}

// MiBloque es un tipo interfaz que tiene el mismo conjunto
// de métodos que Bloque.
type MiBloque Bloque
```

Puedes utilizar una declaración de tipo para definir un diferente valor lógico, numérico o tipo cadena y conectarle métodos:

```
type ZonaHoraria int

const (
    EST ZonaHoraria = -(5 + iota)
    CST
    MST
    PST
)

func (zh ZonaHoraria) String() string {
    return fmt.Sprintf("GMT+%dh", zh)
}
```

Declaración de variables

Una declaración de variable crea una o más variables, vinculándolas a sus correspondientes identificadores, dándoles un tipo y valor inicial a cada una.

```
DeclaVar    = "var" ( EspeVar | "(" { EspeVar ";" } ")" ) .
EspeVar     = ListaIdentificador ( Tipo [ "=" ListaExpresión ] | "=" ListaExpresión ) .
```

```
var i int
var U, V, W float64
var k = 0
var x, y float32 = -1, -2

var (
    i       int
    u, v, s = 2.0, 3.0, "bar"
)

var re, im = RaízCompleja(-1)
var _, encontrada = entradas[nombre]  // consulta mapa; solo está
                                      // interesado en "encontrada"
```

Si se da una lista de expresiones, las variables se inician con las expresiones siguiendo las reglas para asignaciones. De lo contrario, cada variable se inicia a su valor cero.

Si un tipo está presente, a cada variable se le da ese tipo. De lo contrario, a cada variable se le da el tipo del valor inicial correspondiente en la asignación. Si ese valor es una constante sin tipo, primero se convierte a su tipo predeterminado; si es un valor lógico sin tipo, primero se convierte al tipo `bool`. El valor predeclarado `nil` no se puede utilizar para iniciar una variable sin tipo explícito.

```
var d = math.Sin(0.5)  // d es float64
var i = 42             // i es int
var t, ok = x.(T)      // t es T, ok es bool
var n = nil            // ilegal
```

> Restricción de implementación: Un compilador puede hacer que sea ilegal declarar una variable dentro del cuerpo de la función si la variable no se usa nunca.

Declaración corta de variables

Una *declaración corta de variables* utiliza la sintaxis:

```
DeclaVarCorta = ListaIdentificador ":=" ListaExpresión .
```

Esta es la forma abreviada de una declaración de variables con expresión iniciadora pero sin tipos:

```
"var" ListaIdentificador = ListaExpresión .
```

```
i, j := 0, 10
f := func() int { return 7 }
ch := make(chan int)
r, w := os.Pipe(fd)  // os.Pipe() devuelve dos valores
_, y, _ := coord(p)  // coord() devuelve tres valores; solo está
                     // interesado en la coordenada y
```

A diferencia de las declaraciones de variable regulares, una declaración corta de variable puede redeclarar variables siempre y cuando se hayan declarado originalmente antes en el mismo bloque y con el mismo tipo y por lo menos una de las variables no blanca sea nueva. Como consecuencia, una redeclaración solo puede aparecer en una declaración multivariable corta. La redeclaración no introduce una nueva variable; simplemente asigna un nuevo valor a la original.

```
campo1, desplazamiento := siguienteCampo(cadena, 0)
campo2, desplazamiento := siguienteCampo(str, desplazamiento) // redeclara
                                                              // desplazamiento
a, a := 1, 2           // ilegal: doble declaración de a o no nueva variable si
                       // a fue declarada en otra parte
```

Las declaraciones cortas de variable solo pueden aparecer en el interior de funciones. En algunos contextos, como en los iniciadores de las instrucciones "if", "for" o "switch" se pueden utilizar para declarar variables locales temporales.

Declaración de funciones

Una declaración de función vincula un identificador, el *nombre de la función*, a una función.

```
DeclaFunción  = "func" NombreFunción ( Función | Firma ) .
NombreFunción = identificador .
Función       = Firma CuerpoFunción .
CuerpoFunción = Bloque .
```

Si la firma de la función declara parámetros de resultado, la lista de instrucciones del cuerpo de la función debe terminar en una instrucción de terminación.

```
func encuentraMarcador(c <-chan int) int {
    for i := range c {
        if x := <-c; esMarcador(x) {
            return x
        }
    }
    // No válido: falta la instrucción de retorno.
}
```

Una declaración de función puede omitir el cuerpo. Tal declaración provee la firma para una función implementada fuera de Go, tal como una rutina de ensamblador.

```
func min(x int, y int) int {
    if x < y {
        return x
    }
    return y
}

func vacíaCaché(inicio, fin uintptr)  // implementada externamente
```

Declaración de métodos

Un método es una función con un *receptor*. Una declaración de método vincula a un identificador, el *nombre del método*, con un método y asocia el método con el *tipo base* del receptor.

```
DeclaMétodo  = "func" Receptor NombreMétodo ( Función | Firma ) .
Receptor     = Parámetros .
```

El receptor se especifica a través de una sección de parámetros extra precediendo al nombre del método. Esa sección de parámetros debe declarar un solo parámetro, el receptor. Su tipo debe ser de la forma `T` o `*T` (posiblemente usando paréntesis) donde `T` es un nombre de tipo. El tipo denotado por `T` se conoce como el *tipo base* del receptor; no debe ser un tipo puntero o interfaz y se debe declarar en el mismo paquete que el método. El método se dice que está *unido* al tipo base y el nombre del método es visible solo dentro de selectores para ese tipo.

Un identificador receptor no blanco debe ser único en la firma del método. Si no se hace referencia al valor del receptor dentro del cuerpo del método, su identificador se puede omitir en la declaración. Lo mismo se aplica en general a los parámetros de funciones y métodos.

Para un tipo base, los nombres no blancos de métodos vinculados a él deben ser únicos. Si el tipo base es un tipo estructura, el método y los nombres de campos no blancos deben ser distintos.

Dado el tipo `Punto`, las declaraciones

```
func (p *Punto) Longitud() float64 {
    return math.Sqrt(p.x * p.x + p.y * p.y)
}

func (p *Punto) Escala(factor float64) {
    p.x *= factor
    p.y *= factor
}
```

vincula a los métodos `Longitud` y `Escala` con el tipo receptor `*Punto`, de tipo base `Punto`.

El tipo de un método es el tipo de una función con el receptor como primer argumento. Por ejemplo, el método `Escala` tiene el tipo:

```
func(p *Punto, factor float64)
```

Sin embargo, una función declarada de esta manera no es un método.

Expresiones

Una expresión especifica el cálculo de un valor mediante la aplicación de operadores y funciones a operandos.

Operandos

Los operandos denotan los valores elementales de una expresión. Un operando puede ser un literal, un (posiblemente cualificado) identificador no blanco que denota una constante, variable o función, una expresión método produciendo una función o una expresión entre paréntesis.

El identificador blanco puede aparecer como un operando solo en el lado izquierdo de una asignación.

```
Operando       = Literal | NombreOperando | ExpreMétodo | "(" Expresión ")" .
Literal        = LitBásico | LitCompuesto | FunciónLit .
LitBásico      = ent_lit | float_lit | imaginario_lit | rune_lit | cadena_lit .
NombreOperando = identificador | IdentiCualificado.
```

Identificadores cualificados

Un identificador cualificado es un identificador prefijado con un nombre de paquete. Tanto el nombre del paquete como el identificador no deben estar en blanco.

```
IdentiCualificado = NombrePaquete "." identificador .
```

Un identificador cualificado accede a un identificador en un paquete diferente, que se debe importar. El identificador se debe exportar y estar declarado en el bloque de paquete de ese paquete.

```
math.Sin    // denota la función Sin en el paquete math
```

Literales compuestos

Los literales compuestos construyen valores para estructuras, arreglos, sectores, mapas y crean un nuevo valor cada vez que se evalúan. Estos consisten en el tipo del valor seguido de una lista encerrada entre llaves de elementos compuestos. Un elemento puede ser una sola expresión o un par clave-valor.

```
LitCompuesto    = TipoLiteral ValorLiteral .
TipoLiteral     = TipoEstructura | TipoArreglo | "[" "..." "]" TipoElemento |
                  TipoSector | TipoMapa | NombreTipo .
ValorLiteral    = "{" [ ListaElementos [ "," ] ] "}" .
ListaElementos  = Elemento { "," Elemento } .
Elemento        = [ Clave ":" ] Valor .
Clave           = NombreCampo | ÍndiceElemento .
NombreCampo     = identificador .
ÍndiceElemento  = Expresión .
Valor           = Expresión | ValorLiteral .
```

El TipoLiteral debe ser un tipo estructura, arreglo, sector o mapa (la gramática impone esta restricción, excepto cuando el tipo se da como un NombreTipo). Los tipos de las expresiones deben ser asignables al campo respectivo, elemento y tipo clave del TipoLiteral; no hay conversión adicional. La clave se interpreta como un nombre de campo para las estructuras literales, un índice de arreglo y sectores literales, y una clave para

mapas literales. Para mapas literales, todos los elementos deben tener una clave. Es un error especificar múltiples elementos con el mismo nombre de campo o valor clave constante.

Para estructuras literales se aplican las siguientes reglas:

- Una clave debe ser un nombre de campo declarado en el TipoLiteral.
- Una lista de elementos que no contiene ninguna clave debe incluir un elemento para cada campo de estructura en el orden en que se declaran

  ```
  los campos.
  ```

- Si algún elemento tiene una clave, cada elemento debe tener una clave.
- Una lista de elementos que contiene claves no necesita tener un elemento para cada campo de la estructura. Los campos omitidos obtienen el

  ```
  valor cero para ese campo.
  ```

- Un literal puede omitir la lista de elementos; dichos literales evalúan al valor cero para su tipo.
- Es un error especificar un elemento de un campo no exportado de una estructura que pertenece a un paquete diferente.

Teniendo en cuenta las declaraciones

```
type Punto3D struct { x, y, z float64 }
type Línea struct { p, q Punto3D }
```

uno puede escribir

```
origen := Punto3D{}                                // valor cero para Punto3D
línea := Línea{origen, Punto3D{y: -4, z: 12.3}}  // valor cero para línea.q.x
```

Para los arreglos y sectores literales se aplican las siguientes reglas:

- Cada elemento tiene un índice entero asociado marcando su posición en el arreglo.
- Un elemento con una clave utiliza la clave como su índice; la clave debe ser una expresión constante entera.
- Un elemento sin una clave usa el índice del elemento anterior más uno.

  ```
  Si el primer elemento no tiene clave, su índice es cero.
  ```

Tomar la dirección de un literal compuesto genera un puntero a una única variable iniciada con el valor del literal.

```
var puntero *Punto3D = &Punto3D{y: 1000}
```

La longitud de un arreglo literal es la longitud especificada en el TipoLiteral. Si en el literal se proporcionan menos elementos que la longitud, los elementos que faltan se establecen al valor cero para el tipo de elemento del arreglo. Es un error proporcionar elementos con valores de índice fuera del rango índice del arreglo. La notación `...` especifica una longitud de arreglo igual al índice máximo de elementos más uno.

```
búfer  := [10]string{}               // len(búfer) == 10
setEnt := [6]int{1, 2, 3, 5}         // len(setEnt) == 6
días   := [...]string{"Sáb", "Dom"}  // len(días) == 2
```

Un sector literal describe todo el arreglo literal subyacente. Así, la longitud y capacidad de un sector literal son el índice del máximo elemento más uno. un sector literal tiene la forma

```
[]T{x1, x2, … xn}
```

y es la forma abreviada de una operación sector aplicada a un arreglo:

```
temp := [n]T{x1, x2, … xn}
temp[0 : n]
```

Dentro de un literal compuesto de arreglo, sector o mapa de tipo `T`, en que los elementos o claves de mapa en sí mismos son literales compuestos puedes omitir el tipo literal correspondiente si es idéntico al tipo de elemento o clave de mapa de `T`. Del mismo modo, los elementos que son direcciones de literales compuestos pueden omitir la `&T` cuando el tipo del elemento o clave es `*T`.

Dentro de un literal compuesto de elementos arreglo, sector o mapa de tipo `T`, en que ellos mismos son literales compuestos puedes omitir el tipo literal correspondiente si es idéntico al tipo de elemento `T`. Del mismo modo, los elementos que son direcciones de literales compuestos pueden omitir la `&T` cuando el tipo del elemento es `*T`.

```
[...]Punto{{1.5, -3.5}, {0, 0}}   // igual que [...]Punto{Punto{1.5, -3.5},
                                  //                      Punto{0, 0}}
[][]int{{1, 2, 3}, {4, 5}}        // igual que [][]int{[]int{1, 2, 3},
                                  //                   []int{4, 5}}
[][]Punto{{{0, 1}, {1, 2}}}       // igual que [][]Punto{[]Punto{Punto{0, 1},
                                  //                             Punto{1, 2}}}
map[string]Punto{"orig": {0, 0}}  // igual que map[string]Punto{"orig":
                                  //                              Punto{0, 0}}
[...]*Punto{{1.5, -3.5}, {0, 0}}  // igual que [...]*Punto{&Punto{1.5, -3.5},
                                  //                       &Punto{0, 0}}
map[Punto]string{{0, 0}: "orig"}  // igual que map[Punto]string{Punto{0, 0}:
                                  //                                  "orig"}
```

Surge un conflicto de ambigüedad cuando un literal compuesto usando la forma NombreTipo del TipoLiteral aparece como un operando entre la palabra clave y la llave de apertura del bloque de una declaración "if", "for" o "switch" y el literal compuesto no está entre paréntesis, corchetes o llaves. En este raro caso, la llave de apertura del literal es interpretado erróneamente como la introducción del bloque de instrucciones. Para resolver la ambigüedad, el literal compuesto debe aparecer entre paréntesis.

```
if x == (T{a,b,c}[i]) { … }
if (x == T{a,b,c}[i]) { … }
```

Ejemplos de literales válidos de arreglo, sector y mapa:

```
// Lista de números primos
primos := []int{2, 3, 5, 7, 9, 2147483647}

// vocales[v] es true si v es una vocal
vocales := [128]bool{'a': true, "e": true, "i": true, "o": true, "u": true, 'y': true}

// el arreglo [10]float32{-1, 0, 0, 0, -0.1, -0.1, 0, 0, 0, -1}
filtro := [10]float32{-1, 4: -0.1, -0.1, 9: -1}

// Frecuencias en Hz de la escala temperada (A4 = 440Hz)
frecuenciaNota := map[string]float32{
    "C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.83,
    "G0": 24.50, "A0": 27.50, "B0": 30.87,
}
```

Funciones literales

Un función literal representa una función anónima.

```
FunciónLit = "func" Función .
```

```
func(a, b int, z float64) bool { return a*b < int(z) }
```

Una función literal se puede asignar a una variable o invocarse directamente.

```
f := func(x, y int) int { return x + y }
func(ch chan int) { ch <- ACK }(canalRespuesta)
```

Las funciones literales son *cierres*: se pueden referir a variables definidas en funciones circundantes. Las variables se comparten entre las funciones circundantes y la función literal, además de que sobreviven siempre y cuando sean accesibles.

Expresiones primarias

Las expresiones primarias son los operandos de expresiones unarias y binarias.

```
ExprePrimaria =
    Operando |
    Conversión |
    ExprePrimaria Selector |
    ExprePrimaria Índice |
    ExprePrimaria Sector |
    ExprePrimaria TipoAserción |
    ExprePrimaria Argumentos .
Selector       = "." identificador .
Índice         = "[" Expresión "]" .
Sector         = "[" ( [ Expresión ] ":" [ Expresión ] ) |
                     ( [ Expresión ] ":" Expresión ":" Expresión )
                 "]" .
TipoAserción   = "." "(" Tipo ")" .

Argumentos     = "(" [ ( ListaExpresión | Tipo [ "," ListaExpresión ] )
                                            [ "..." ] [ "," ] ] ")" .
```

```
x
2
(s + ".txt")
f(3.1415, true)
Punto{1, 2}
m["foo"]
s[i : j + 1]
obj.color
f.p[i].x()
```

Selectores

Para una expresión primaria `x` que no es un nombre de paquete, la *expresión selectora*

```
x.f
```

denota el campo o método `f` del valor de `x` (o, a veces `*x` ; véase más adelante). El identificador de `f` se conoce como el (campo o método) *selector*; no debe ser el identificador blanco. El tipo de la expresión selectora es el tipo de `f` . Si `x` es el nombre del paquete, consulta la sección sobre identificadores cualificados.

Un selector `f` puede denotar un campo o método `f` de un tipo `T` , o bien se puede referir a un campo o método `f` anidado en un campo anónimo de `T` . El número de campos anónimos atravesados para alcanzar `f` se conoce como su *profundidad* en `T` . La profundidad de un campo o método `f` declarada en `T` es cero. La profundidad de un campo o método `f` declarado en un campo anónimo `A` en `T` es la profundidad de `f` en `A` más uno.

Las siguientes reglas se aplican a los selectores:

1. Para un valor `x` de tipo `T` o `*T` donde `T` no es un puntero o tipo interfaz, `x.f` denota el campo o método a la profundidad mínima en `T` donde hay tal `f` . Si no hay exactamente una `f` con menor profundidad, la expresión selectora es ilegal.
2. Para un valor `x` de tipo `I` , donde `I` es un tipo interfaz, `x.f` denota el método real con el nombre `f` del valor dinámico de `x` . Si no hay ningún método con el nombre `f` en el conjunto de métodos de `I` , la expresión selectora es ilegal.
3. Como excepción, si el tipo de `x` es un tipo puntero nombrado y `(*x).f` es una expresión selectora válida que denota un campo (pero no un método), `x.f` es la abreviatura de `(*x).f` .
4. En todos los demás casos, `x.f` es ilegal.
5. Si `x` es de tipo puntero y tiene el valor `nil` y `x.f` denota un campo de estructura, asignando o evaluando a `x.f` provoca pánico en tiempo de ejecución.
6. Si `x` es de tipo interfaz y tiene el valor `nil` , llamar o evaluar el método `x.f` provoca pánico en tiempo de ejecución.

Por ejemplo, dadas las declaraciones:

```
type T0 struct {
    x int
}

func (*T0) M0()

type T1 struct {
    y int
}

func (T1) M1()

type T2 struct {
    z int
    T1
    *T0
}

func (*T2) M2()

type Q *T2

var t T2     // con t.T0 != nil
var p *T2    // con p != nil y (*p).T0 != nil
var q Q = p
```

uno puede escribir:

```
t.z          // t.z
t.y          // t.T1.y
t.x          // (*t.TO).x
p.z          // (*p).z
p.y          // (*p).T1.y
p.x          // (*(*p).T0).x
q.x          // (*(*q).T0).x        (*q).x es un campo selector válido
p.M2()       // p.M2()              M2 espera receptor de *T2
p.M1()       // ((*p).T1).M1()      M1 espera receptor de T1
p.M0()       // ((&(*p).T0)).M0()   M0 espera receptor de *T0, ve
             //                        la sección de Llamadas
```

pero lo siguiente no es válido:

```
q.M0()       // (*q).M0 es válido pero no un campo selector
```

Expresiones método

Si `M` está en el conjunto de métodos del tipo `T`, `T.M` es una función que se puede llamar como una función regular con los mismos argumentos que `M` prefijados por un argumento adicional que es el receptor del método.

```
ExpreMétodo   = TipoReceptor "." NombreMétodo .
TipoReceptor  = NombreTipo | "(" "*" NombreTipo ")" | "(" TipoReceptor ")" .
```

Considera un tipo estructura `T` con dos métodos, `Mv`, cuyo receptor es de tipo `T` y `Mp`, cuyo receptor es de tipo `*T`.

```
type T struct {
    a int
}

func (tv  T) Mv(a int) int         { return 0 }  // receptor del valor
func (tp *T) Mp(f float32) float32 { return 1 }  // puntero al receptor

var t T
```

La expresión

```
T.Mv
```

produce una función equivalente a `Mv` pero con un receptor explícito como primer argumento; el cuál tiene la firma

```
func(tv T, a int) int
```

Esa función se puede invocar normalmente con un receptor explícito, por lo que estas cinco llamadas son equivalentes:

```
t.Mv(7)
T.Mv(t, 7)
(T).Mv(t, 7)
f1 := T.Mv; f1(t, 7)
f2 := (T).Mv; f2(t, 7)
```

Del mismo modo, la expresión

```
(*T).Mp
```

produce un valor función representando a `Mp` con la firma

```
func(tp *T, f float32) float32
```

Para un método con un receptor de valor, se puede derivar una función con un puntero receptor explícito, por lo tanto

```
(*T).Mv
```

produce un valor función que representa a `Mv` con la firma

```
func(tv *T, a int) int
```

Tal función direcciona al receptor para crear un valor a pasar como el receptor para el método subyacente; el método no sobrescribe el valor cuya dirección se pasa en la llamada a la función.

El caso final, una función receptor-valor para un método receptor-puntero, es ilegal porque los métodos receptor-puntero no están en el conjunto de métodos del tipo valor.

Los valores función derivados de los métodos se invocan con la sintaxis de llamada a función; el receptor se proporciona como el primer argumento de la llamada. Es decir, dada `f := T.Mv`, `f` es invocada como `f(t, 7)` no `t.f(7)`. Para construir una función que se vincule al receptor, usa una función literal o un valor método.

Es legal derivar un valor función desde un método de un tipo interfaz. La función resultante tiene un receptor explícito de ese tipo interfaz.

Valores método

Si la expresión `x` tiene tipo estático `T` y `M` está en el conjunto de métodos del tipo `T`, `x.M` se conoce como un *valor método*. El valor método `x.M` es un valor de función que es invocable con los mismos argumentos que una llamada al método `x.M`. La expresión `x` se evalúa y se guarda durante la evaluación del valor método; la copia guardada se utiliza entonces como el receptor en cualquier llamada que se pueda ejecutar después.

El tipo `T` puede ser un tipo interfaz o no interfaz.

Como en la explicación de la expresión método anterior, considera un tipo estructura `T` con dos métodos, `Mv`, cuyo receptor es de tipo `T` y `Mp`, cuyo receptor es de tipo `*T`.

```
type T struct {
    a int
}

func (tv  T) Mv(a int) int         { return 0 }  // receptor del valor
func (tp *T) Mp(f float32) float32 { return 1 }  // puntero al receptor

var t T
var pt *T

func makeT() T
```

La expresión

```
T.Mv
```

obtiene un valor de tipo función

```
func(int) int
```

Estas dos invocaciones son equivalentes:

```
t.Mv(7)
f := t.Mv; f(7)
```

Del mismo modo, la expresión

```
pt.Mp
```

obtiene un valor de tipo función

```
func(float32) float32
```

Como con los selectores, una referencia a un método no interfaz con un receptor de valor utilizando un puntero automáticamente desreferencia ese puntero: `pt.Mv` es equivalente a `(*pt).Mv`.

Al igual que con las llamadas a método, una referencia a un método no interfaz con un receptor puntero utilizando un valor direccionable automáticamente tomará la dirección de ese valor: `t.Mp` es equivalente a `(&t).Mp`.

```
f := t.Mv; f(7)   // como t.Mv(7)
f := pt.Mp; f(7)  // como pt.Mp(7)
f := pt.Mv; f(7)  // como (*pt).Mv(7)
f := t.Mp; f(7)   // como (&t).Mp(7)
f := makeT().Mp   // no válido: el resultado de makeT() no es direccionable
```

Aunque los ejemplos anteriores utilizan tipos no interfaz, también es legal crear un valor método desde un valor de tipo interfaz.

```
var i interface { M(int) } = miValor
f := i.M; f(7)  // como i.M(7)
```

Expresiones índice

Una expresión primaria de la forma

```
a[x]
```

denota el elemento del arreglo, puntero a arreglo, sector, cadena o mapa `a` indexado por `x`. El valor `x` se conoce cómo el *índice* del arreglo o *clave del mapa*, respectivamente. Se aplican las siguientes reglas:

Si `a` no es un mapa:

- el índice `x` debe ser de tipo entero o sin tipo; estar *en el rango* si `0 <= x < len(a)`, de lo contrario está *fuera de rango*
- una constante índice no debe ser negativa y representable por un valor de tipo `int`

Para `a` de tipo arreglo `A`:

- una constante índice debe estar en rango
- si `x` está fuera de rango en tiempo de ejecución, se produce pánico en tiempo de ejecución
- `a[x]` es el elemento del arreglo en el índice `x` y el tipo de `a[x]` es el tipo del elemento `A`

Para el puntero `a` al tipo arreglo:

- `a[x]` es la abreviatura de `(*a)[x]`

Para `a` de tipo sector `S`:

- si `x` está fuera de rango en tiempo de ejecución, se produce pánico en tiempo de ejecución

- `a[x]` es el elemento del sector en el índice `x` y el tipo de `a[x]` es el tipo del elemento `S`

Para `a` de tipo cadena:

- una constante índice debe estar en rango si la cadena `a` también es constante
- si `x` está fuera de rango en tiempo de ejecución, se produce pánico en tiempo de ejecución
- `a[x]` es el valor byte no constante en el índice `x` y el tipo de `a[x]` es `byte`
- nada se puede asignar a `a[x]`

Para `a` de tipo mapa `M` :

- el tipo de `x` debe ser asignable al tipo de la clave `M`
- si el mapa contiene una entrada con clave `x` , `a[x]` es el valor del mapa con clave `x` y el tipo de `a[x]` es el valor del tipo `M`
- si el mapa es `nil` o no contiene dicha entrada, `a[x]` es el valor cero para el valor del tipo `M`

De lo contrario `a[x]` es ilegal.

Una expresión índice en un mapa `a` de tipo `map[K]V` utilizado en una asignación o iniciación de la forma especial

```
v, ok = a[x]
v, ok := a[x]
var v, ok = a[x]
```

adicionalmente arroja un valor lógico sin tipo. El valor de `ok` es `true` si la clave `x` está presente en el mapa y `false` de lo contrario.

Asignar a un elemento de un mapa `nil` provoca pánico en tiempo de ejecución.

Expresiones sector

Las expresiones sector construyen una subcadena o un sector a partir de una cadena, arreglo, puntero a arreglo o sector. Hay dos variantes: una forma simple que especifica una cota baja y alta, y una forma completa que también especifica un límite en capacidad.

Expresiones sector simples

Para una cadena, arreglo, puntero a arreglo o sector `a` , la expresión primaria

```
a[bajo : alto]
```

construye una subcadena o un sector. Los *índices* `bajo` y `alto` seleccionan los elementos del operando `a` que aparecen en el resultado. El resultado tiene índices que comienzan en 0 y una longitud igual a `alto` - `bajo`. Después de seccionar el arreglo `a`

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

el sector `s` tiene el tipo `[]int`, longitud 3, capacidad 4 y los elementos:

```
s[0] == 2
s[1] == 3
s[2] == 4
```

Por conveniencia, cualquiera de los índices se puede omitir. A falta del índice `bajo` el valor predeterminado es cero; a falta del índice `alto` el predeterminado es la longitud del operando seccionado:

```
a[2:] // igual que a[2 : len(a)]
a[:3] // igual que a[0 : 3]
a[:]  // igual que a[0 : len(a)]
```

Si `a` es un puntero a un arreglo, `a[bajo : alto]` es la abreviatura de `(*a)[bajo : alto]`.

Para arreglos o cadenas, los índices están *en rango* si `0` <= `bajo` <= `alto` <= `len(a)`, de lo contrario estan *fuera de rango*. Para sectores, el índice del límite superior es la capacidad del sector `cap(a)` en lugar de la longitud. Una constante índice no debe ser negativa y representable por un valor de tipo `int`; para arreglos o cadenas constantes, las constantes índice también deben estar en rango. Si ambos índices son constantes, deben satisfacer `bajo <= alto`. Si los índices están fuera de rango en tiempo de ejecución, se produce pánico en tiempo de ejecución.

Excepto para las cadenas sin tipo, si el operando seccionado es una cadena o un sector, el resultado de la operación de seccionado es un valor no constante del mismo tipo que el operando. Para operandos cadena sin tipo el resultado es un valor no constante de tipo `cadena`. Si el operando seccionado es un arreglo, debe ser direccionable y el resultado de la operación de seccionado es un sector con el mismo tipo de elemento que el arreglo.

Si el operando seccionado de una expresión sector válida es un sector `nil`, el resultado es un sector `nil`. De lo contrario, el resultado comparte su arreglo subyacente con el operando.

Expresiones sector complejas

Para un arreglo, puntero a arreglo o sector `a` (pero no una cadena), la expresión primaria

```
a[bajo : alto : máx]
```

construye un sector del mismo tipo y con la misma longitud y elementos que la simple expresión sector `a[bajo : alto]`. Además, esta controla la capacidad del sector resultante estableciéndola en `máx - bajo`. Únicamente se puede omitir el primer índice; su valor predeterminado es 0. Después de seccionar el arreglo `a`

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

el sector `t` tiene el tipo `[]int`, longitud 2, capacidad 4 y los elementos

```
t[0] == 2
t[1] == 3
```

Cómo para las expresiones sector simples, si `a` es un puntero a un arreglo, `a[bajo : alto : máx]` es la abreviatura de `(*a)[bajo : alto : máx]`. Si el operando seccionado es un arreglo, debe ser direccionable.

Los índices están *en rango* si `0 <= bajo <= alto <= máx <= cap(a)`, si no están *fuera de rango*. Una constante índice no debe ser negativa y representable por un valor de tipo `int`; para arreglos, los índices constantes también deben estar en rango. Si varios índices son constantes, las constantes que están presentes deben estar en rango relativo de uno al otro. Si los índices están fuera de rango en tiempo de ejecución, se produce pánico en tiempo de ejecución.

Tipos aserción

Para una expresión `x` de tipo interfaz y un tipo `T`, la expresión primaria

```
x.(T)
```

afirma que `x` no es `nil` y que el valor almacenado en `x` es de tipo `T`. La notación `x.(T)` se llama *tipo aserción*.

Exactamente, si `T` no es un tipo interfaz, `x.(T)` afirma que el tipo dinámico de `x` es idéntico al tipo de `T`. En este caso, `T` debe implementar la (interfaz) del tipo `x`; de lo contrario el tipo aserción no es válido ya que no es posible para `x` almacenar un valor de

tipo `T`. Si `T` es un tipo interfaz, `x.(T)` afirma que el tipo dinámico de `x` implementa la interfaz `T`.

Si el tipo aserción se mantiene, el valor de la expresión es el valor almacenado en `x` y su tipo es `T`. Si el tipo aserción es falso, se produce pánico en tiempo de ejecución. En otras palabras, aunque el tipo dinámico de `x` es conocido solo en tiempo de ejecución, el tipo de `x.(T)` es conocido por ser `T` en un programa correcto.

```
var x interface{} = 7  // x tiene tipo dinámico int y valor 7

i := x.(int)           // i tiene tipo int y valor 7.

type I interface { m() }

var y I

s := y.(string)        // ilegal: "string" no implementa a I (falta el
                       // método m)

r := y.(io.Reader)     // r tiene tipo io.Reader e y debe implementar
                       // ambas I e io.Reader
```

Un tipo aserción usado en una asignación o iniciación de la forma especial

```
v, ok = x.(T)
v, ok := x.(T)
var v, ok = x.(T)
```

adicionalmente arroja un valor lógico sin tipo. El valor de `ok` es `true` si se mantiene la aserción. De lo contrario, es `false` y el valor de `v` es el valor cero para el tipo `T`. En este caso no ocurre ningún pánico en tiempo de ejecución.

Llamadas

Dada una expresión `f` de tipo función `F`:

```
f(a1, a2, … an)
```

invoca a `f` con argumentos `a1, a2, … an`. A excepción de un caso especial, los argumentos deben ser expresiones de un solo valor asignable a los tipos de parámetros de `F` y son evaluados antes de llamar a la función. El tipo de la expresión es el tipo de resultado de `F`. Una invocación a método es similar, pero el método en sí se especifica como un selector sobre un valor del tipo receptor para el método.

```
math.Atan2(x, y)  // invoca a la función

var pt *Punto

pt.Scale(3.5)     // llamada a método con pt como receptor
```

En una llamada a función, el valor de la función y los argumentos se evalúan en el orden habitual. Después de evaluarlos, los parámetros de la llamada se pasan por valor a la función y la función llamada comienza su ejecución. Los parámetros de retorno de la función de nuevo se pasan por valor a la función llamada cuando la función regresa.

Llamar a un valor de función `nil` provoca pánico en tiempo de ejecución.

Como caso especial, si los valores de retorno de una función o método `g` son iguales en número e individualmente asignables a los parámetros de otra función o método `f`, entonces la llamada `f(g(parámetros-de-g))` invocará a `f` después de vincular en orden los valores de retorno de `g` a los parámetros de `f`. La llamada a `f` no debe contener ningún otro parámetro salvo la llamada a `g` y `g` debe tener por lo menos un valor de retorno. Si `f` tiene un parámetro `...` final, este se asigna el valor de retorno de `g` que permanece después de la asignación de los parámetros normales.

```
func Divide(s string, pos int) (string, string) {
    return s[0:pos], s[pos:]
}

func Junta(s, t string) string {
    return s + t
}

if Junta(Divide(valor, len(valor)/2)) != valor {
    log.Panic("prueba fallida")
}
```

Una llamada al método `x.m()` es válida si el conjunto de métodos de (el tipo de) `x` contiene `m` y la lista de argumentos se puede asignar a la lista de parámetros de `m`. Si `x` es direccionable y el conjunto de métodos de `&x` contiene a `m`, `x.m()` es una abreviatura de `(&x).m()`:

```
var p Punto
p.Escala(3.5)
```

No hay ningún tipo método distinto y no hay métodos literales.

Pasando argumentos a parámetros `...`

Si `f` es variadica con un parámetro final `p` de tipo `...T`, entonces dentro de `f` de tipo `p` es equivalente al tipo `[]T`. Si `f` se invoca sin argumentos reales de `p`, el valor pasado a `f` es `nil`. De lo contrario, el valor pasado es un nuevo sector de tipo `[]T` con un nuevo arreglo subyacente cuyos elementos sucesivos son los argumentos reales, todos ellos deben ser asignables a `T`. Por tanto, la longitud y capacidad del sector es el número de argumentos ligados a `p` y pueden ser diferentes para cada sitio de la llamada.

Dada la función y llamadas

```
func Saluda(prefijo string, quién ...string)

Saluda("nadie")
Saluda("hola", "José", "Ana", "Elena")
```

dentro de `Saluda`, `quién` tendrá el valor `nil` en la primera llamada y `[]string{&quot;José&quot;, &quot;Ana&quot;, &quot;Elena&quot;}` en la segunda.

Si el argumento final es asignable a un tipo sector `[]T`, se puede pasar sin cambios como el valor de un parámetro `...T` si el argumento está seguido por `...`. En este caso no se crea un nuevo sector.

Dado el sector `s` y la llamada

```
s := []string{"Jaime", "Jazmín"}
Saluda("adiós:", s...)
```

dentro de `Saluda`, `quién` tendrá el mismo valor que `s` con el mismo arreglo subyacente.

Operadores

Los operadores combinan operandos en expresiones.

```
Expresión   = ExprUnaria | Expresión op_binario ExprUnaria .
ExprUnaria  = ExprePrimaria | op_unario ExprUnaria .
op_binario  = "||" | "&&" | op_rel | op_adi | op_mul .
op_rel      = "==" | "!=" | "<" | "<=" | ">" | ">=" .
op_adi      = "+" | "-" | "|" | "^" .
op_mul      = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .
op_unario   = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

Las comparaciones se dilucidan en otro lugar. Para otros operadores binarios, los tipos de los operandos deben ser idénticos a menos que la operación involucre desplazamientos o constantes sin tipo. Para las operaciones que solo implican constantes, consulta la sección sobre expresiones constantes.

Excepto para las operaciones de desplazamiento, si un operando es una constante sin tipo y el otro operando no, la constante se convierte al tipo del otro operando.

El operando de la derecha en una expresión de desplazamiento debe tener tipo entero sin signo o ser una constante sin tipo que se pueda convertir al tipo entero sin signo. Si el operando de la izquierda de una expresión de desplazamiento no constante es una constante sin tipo, el tipo de la constante es lo que sería si la expresión de desplzamiento fuera reemplazada solo por su operando izquierdo.

```
var s uint = 33
var i = 1<<s           // 1 tiene tipo int
var j int32 = 1<<s     // 1 tiene tipo int32; j == 0
var k = uint64(1<<s)   // 1 tiene tipo uint64; k == 1<<33
var m int = 1.0<<s     // 1.0 tiene tipo int
var n = 1.0<<s != i    // 1.0 tiene tipo int; n == false si los enteros
                       // son de 32 bits
var o = 1<<s == 2<<s   // 1 y 2 tiene tipo int; o == true si los enteros
                       // son de 32 bits
var p = 1<<s == 1<<33  // ilegal si los enteros son de 32 bits: 1 tiene
                       // tipo int, pero 1<<33 desborda el int
var u = 1.0<<s         // ilegal: 1.0 tiene tipo float64, no se puede
                       // desplazar
var u1 = 1.0<<s != 0   // ilegal: 1.0 tiene tipo float64, no se puede
                       // desplazar
var u2 = 1<<s != 1.0   // ilegal: 1.0 tiene tipo float64, no se puede
                       // desplazar
var v float32 = 1<<s   // ilegal: 1 tiene tipo float64, no se puede
                       // desplazar
var w int64 = 1.0<<33  // 1.0<<33 es una expresión de desplazamiento
                       // constante
```

Precedencia de operadores

Los operadores unarios tienen la precedencia más alta. Puesto que `++` y `--` forman declaraciones de operadores, no expresiones, caen fuera de la jerarquía del operador. Por consecuencia, la declaración `*p++` es la misma que `(*p)++`.

Hay cinco niveles de precedencia en los operadores binarios. Los operadores de multiplicación se vinculan fuertemente, seguidos de los operadores de adición, operadores de comparación `&amp;&amp;` (`AND` lógico) y finalmente `||` (`OR` lógico):

```
Precedencia              Operador
    5             *  /  %  <<  >>  &  &^
    4             +  -  |  ^
    3             ==  !=  <  <=  >  >=
    2             &&
    1             ||
```

Los operadores binarios de la misma precedencia se asocian de izquierda a derecha. Por ejemplo, `x / y * z` es lo mismo que `(x / y) * z`.

```
+x
23 + 3*x[i]
x <= f()
^a >> b
f() || g()
x == y+1 && <-chanPtr > 0
```

Operadores aritméticos

Los operadores aritméticos se aplican a los valores numéricos y producen un resultado del mismo tipo que el primer operando. Los cuatro operadores aritméticos estándar (`+` , `-` , `*` y `/`) se aplican a tipos numéricos enteros, de coma flotante y complejos; `+` también se aplica a cadenas de caracteres. Todos los demás operadores aritméticos solo se aplican a enteros.

```
+    suma              valores numéricos enteros, de coma flotante,
                                                    complejos y cadenas
-    diferencia        valores numéricos enteros, de coma flotante,
                                                              complejos
*    producto          valores numéricos enteros, de coma flotante,
                                                              complejos
/    cociente          valores numéricos enteros, de coma flotante,
                                                              complejos
%    residuo           enteros

&    AND a nivel de bits        enteros
|    OR a nivel de bits         enteros
^    XOR a nivel de bits        enteros
&^   limpieza de bit (y NOT)    enteros
<<   desplaza a la izquierda    entero << entero sin signo
>>   desplaza a la derecha      entero >> entero sin signo
```

Las cadenas se pueden concatenar usando el operador `+` o el operador de asignación `+=` :

```
s := "hola" + string(c)
s += " y adiós"
```

La adición de cadenas crea una nueva cadena concatenando los operandos.

Para dos valores enteros `x` e `y`, el cociente entero `q = x / y` y residuo `r = x % y` satisfacen las siguientes relaciones:

```
x = q*y + r  and  |r| < |y|
```

con `x / y` truncado hacia cero ("división truncada").

```
x     y     x / y     x % y
 5     3       1         2
-5     3      -1        -2
 5    -3      -1         2
-5    -3       1        -2
```

Como una excepción a esta regla, si el dividendo `x` es el valor negativo para el tipo int de `x`, el cociente `q = x / -1` es igual a `x` (y `r = 0`).

```
x, q
int8                     -128
int16                  -32768
int32             -2147483648
int64    -9223372036854775808
```

Si el divisor es una constante, esta no debe ser cero. Si el divisor es cero en tiempo de ejecución, se produce pánico en tiempo de ejecución. Si el dividendo no es negativo y el divisor es una constante potencia de 2, la división se puede reemplazar por un desplazamiento a la derecha y el cálculo del residuo se podrá sustituir por una operación `AND` bit a bit:

```
x     x / 4     x % 4     x >> 2     x & 3
 11      2         3          2         3
-11     -2        -3         -3         1
```

Los operadores de desplazamiento desplazan el operando de la izquierda por el valor de desplazamiento especificado por el operando de la derecha. Estos implementan desplazamiento aritmético si el operando de la izquierda es un entero con signo y los desplazamientos lógicos si se trata de un entero sin signo. No hay un límite superior para el valor del desplazamiento. Los desplazamientos se comportan como si el operando de la

izquierda se desplazara `n` veces por 1 para un valor de desplazamiento `n`. Como resultado, `x << 1` es lo mismo que `x*2` y `x >> 1` es lo mismo que `x/2`, pero truncado hacia el infinito negativo.

Para operandos enteros, los operadores unarios `+`, `-` y `^` se definen de la siguiente manera:

```
+x                          es 0 + x
-x    negación              es 0 - x
^x    complemento bit a bit es m ^ x  dónde m = "todos los bits puestos a 1"
                                        para x sin signo
                                    y  m = -1 para x con signo
```

Para números de coma flotante y complejos, `+x` es lo mismo que `x`, mientras que `-x` es la negación de `x`. El resultado de una división de coma flotante o compleja entre cero no se especifica más allá del estándar IEEE-754; cuando se produce pánico en tiempo de ejecución es específico de la implementación.

Desbordamiento de enteros

Para valores enteros sin signo, las operaciones `+`, `-`, `*` y `<<` se calculan módulo 2 *n* , donde *n* es el número de bits del tipo entero sin signo. En términos generales, estas operaciones de enteros sin signo descartan bits altos sobre desbordamiento y los programas pueden depender de ``aproximaciones".

Para enteros con signo, las operaciones `+`, `-`, `*` y `<<` legalmente pueden desbordar y si el valor resultante existe y está definido deterministamente por la representación del entero con signo, la operación y sus operandos. No se lanza una excepción como resultado de desbordamiento. Un compilador no puede optimizar el código bajo el supuesto de que no se produce desbordamiento. Por ejemplo, no puede suponer que `x < x + 1` siempre sea cierto.

Operadores de comparación

Los operadores de comparación comparan dos operandos y producen un valor lógico sin tipo.

```
==    igual
!=    no igual
<     menor
<=    menor o igual
>     mayor
>=    mayor o igual
```

En cualquier comparación, el primer operando debe ser asignable al tipo del segundo operando o viceversa.

Los operadores de igualdad `==` y `!=` se aplican a los operandos que son *comparables*. Los operadores de ordenamiento `<` , `<=` , `>` y `>=` se aplican a los operandos que son *ordenados*. Estos términos y el resultado de las comparaciones se definen de la siguiente manera:

- Los valores lógicos son comparables. Dos valores lógicos son iguales si ambos son `true` o ambos son `false` .
- Los valores enteros son comparables y ordenados, de la forma habitual.
- Los valores de coma flotante son comparables y ordenados, como lo define la norma IEEE-754.
- Los valores complejos son comparables. Dos valores complejos `u` y `v` son iguales si ambos son `real(u) == real(v)` y `imag(u) == imag(v)` .
- Los valores de cadena son comparables y ordenados, léxicamente bit a bit.
- Los valores puntero son comparables. Dos valores puntero son iguales si apuntan a la misma variable o si ambos tienen valor `nil` . Los punteros a distinta variable de tamaño cero pueden o no ser iguales.
- Los valores de canal son comparables. Dos valores de canal son iguales si fueron creados por la misma llamada a `make` o si ambos tienen valor `nil` .
- Los valores de interfaz son comparables. Dos valores de interfaz son iguales si tienen idénticos tipos dinámicos y valores dinámicos iguales o si ambos tienen valor `nil` .
- Un valor `x` de tipo no interfaz `X` y un valor `t` de tipo interfaz `T` son comparables cuando los valores del tipo `X` son comparables y `X` implementa a `T` . Ellos son iguales si el tipo dinámico de `t` es idéntico a `X` y el valor dinámico de `t` es igual a `x` .
- Los valores estructura son comparables si todos sus campos son comparables. Dos valores estructura son iguales si sus correspondientes campos no blancos son iguales.
- Los valores de arreglo son comparables si el tipo de los valores de los elementos del arreglo son comparables. Dos valores de arreglo son iguales si sus correspondientes elementos son iguales.

Una comparación de dos valores interfaz con tipos dinámicos idénticos provoca pánico en tiempo de ejecución si los valores de ese tipo no son comparables. Este comportamiento se aplica no solo a las comparaciones de valores interfaz directos sino también al comparar arreglos de valores interfaz o estructuras con campos de valor interfaz.

Los valores de sector, mapa y función no son comparables. Sin embargo, como un caso especial, un valor de sector, mapa o función se puede comparar con el identificador predeclarado `nil` . La comparación de valores puntero, canal e interfaz a `nil` también se permite y sigue las reglas generales anteriores.

```
const c = 3 < 4            // c es true, la constante lógica sin tipo

type MiBooleano bool

var x, y int

var (
    // el resultado de una comparación es un bool sin tipo.
    // Aplican las reglas de asignación habituales.
    b3        = x == y     // b3 es de tipo bool
    b4 bool   = x == y     // b4 es de tipo bool
    b5 MiBooleano = x == y // b5 es de tipo MiBooleano
)
```

Operadores lógicos

Los operadores lógicos se aplican a valores booleanos y producen un resultado del mismo tipo que los operandos. El operando de la derecha se evalúa de forma condicional.

```
&&    AND condicional    p && q  es  "si p entonces q si no false"
||    OR condicional     p || q  es  "si p entonces true si no q"
!     NOT                !p      es  "no p"
```

Operadores de dirección

Para un operando `x` de tipo `T`, la operación de dirección `&x` genera un puntero de tipo `*T` a `x`. El operando debe ser *direccionable*, es decir, ya sea una variable, indirección de puntero o la operación de indexación de un sector; o un selector de campo de un operando direccionable a estructura; o una operación de indexación de arreglo de un arreglo direccionable. Como excepción al requisito de direccionamiento, `x` también puede ser un (posiblemente encerrado entre paréntesis) literal compuesto. Si la evaluación de `x` pudiera provocar pánico en tiempo de ejecución, la evaluación de `x` también lo haría.

Para un operando `x` de tipo puntero `*T`, la indirección de puntero `*x` denota la variable de tipo `T` apuntada por `x`. Si `x` es `nil`, un intento de evaluar `*x` provocará pánico en tiempo de ejecución.

```
&x
&a[f(2)]
&Punto{2, 3}
*p
*pf(x)
var x *int = nil
*x   // provoca pánico en tiempo de ejecución
&*x  // provoca pánico en tiempo de ejecución
```

Operador de recepción

Para un operando `ch` de tipo canal, el valor de la operación de recepción `<-ch` es el valor recibido desde el canal `ch`. La dirección del canal debe permitir operaciones de recepción y el tipo de la operación de recepción es el tipo de elemento del canal. Los bloques de expresión hasta un cierto valor están disponibles. Recibir desde un canal `nil` lo bloquea por siempre. Una operación de recepción en un canal cerrado siempre se puede proceder inmediatamente, dando el tipo de elemento de valor cero después de que se hayan recibido todos los valores previamente enviados.

```
v1 := <-ch
v2 = <-ch
f(<-ch)
<-estrobo  // Espera hasta un pulso del reloj
           // y desecha el valor recibido
```

Una expresión de recepción utilizada en una asignación o iniciación de la forma especial

```
x, ok = <-ch
x, ok := <-ch
var x, ok = <-ch
```

Adicionalmente arroja un resultado lógico sin tipo informando si la comunicación tuvo éxito. El valor de `ok` es `true` si el valor recibido fue entregado por una operación de envío correcta en el canal, o `false` si se trata de un valor cero generado debido a que el canal está cerrado y vacío.

Conversiones

Las conversiones son expresiones de la forma `T(x)`, donde `T` es un tipo y `x` es una expresión que se puede convertir al tipo `T`.

```
Conversión = Tipo "(" Expresión [ "," ] ")" .
```

Si el tipo comienza con el operador `*` o `<-`, o si el tipo comienza con la palabra clave `func` y no tiene una lista de resultados, este se debe encerrar entre paréntesis cuando sea necesario para evitar ambigüedades:

```
*Punto(p)        // igual que *(Punto(p))
(*Punto)(p)      // p se convierte a *Punto
<-chan int(c)    // igual que <-(chan int(c))
(<-chan int)(c)  // c se convierte a <-chan int
func()(x)        // firma de la función func() x
(func())(x)      // x se convierte a func()
(func() int)(x)  // x se convierte a func() int
func() int(x)    // x se convierte a func() int (sin ambigüedades)
```

Un valor constante `x` se puede convertir al tipo `T` en cualquiera de estos casos:

- `x` es representable por un valor de tipo `T`.
- `x` es una constante de coma flotante, `T` es un tipo de coma flotante y `x` es representable por medio de un valor de tipo `T` después de redondeado usando las reglas de redondeo al par IEEE 754. La constante `T(x)` es el valor redondeado.
- `x` es una constante entera y `T` es un tipo cadena. En este caso, aplica la misma regla que para la `x` no constante.

La conversión de una constante produce una constante tipada como resultado.

```
uint(iota)               // valor iota de tipo uint
float32(2.718281828)     // 2.718281828 de tipo float32
complex128(1)            // 1.0 + 0.0i de tipo complex128
float32(0.49999999)      // 0.5 de tipo float32
string('x')              // "x" de tipo string
string(0x266c)           // "♬" de tipo string
MiCadena("foo" + "bar")  // "foobar" de tipo MiCadena
string([]byte{'a'})      // no es una constante: []byte{'a'} no es una
                         // constante
(*int)(nil)              // no es una constante: nil no es una constante,
                         // *int no es un tipo booleano, numérico o string
int(1.2)                 // ilegal: 1.2 no se puede representar como un int
string(65.0)             // ilegal: 65.0 no es una constante entera
```

Un valor no constante `x` se puede convertir al tipo `T` en cualquiera de estos casos:

- `x` es asignable a `T`.
- `x` y `T` tienen idénticos tipos subyacentes.
- `x` y `T` son tipos puntero sin nombre y sus tipos puntero base tienen tipos subyacentes idénticos.
- ambos `x` y `T` son tipos enteros o de coma flotante.
- ambos `x` y `T` son tipos complejos.

- `x` es un número entero o un sector de bytes o runes y `T` es un tipo cadena.
- `x` es una cadena y `T` es un sector de bytes o runes.

Se aplican normas específicas para conversiones (no constantes) entre tipos numéricos o desde y hacia un tipo cadena. Estas conversiones pueden cambiar la representación de `x` e incurrir en un costo en tiempo de ejecución. Todas las demás conversiones solo cambian el tipo pero no la representación de `x`.

No hay ningún mecanismo lingüístico para convertir entre punteros y números enteros. Los paquetes `inseguros` implementan esta funcionalidad en circunstancias restringidas.

Conversiones entre tipos numéricos

Para la conversión de valores numéricos no constantes, se aplican las siguientes reglas:

1. Cuando conviertas entre tipos enteros, si el valor es un entero con signo, este signo se extiende implícitamente a precisión infinita; de lo contrario, se extiende a cero. Entonces se trunca para encajar en el tamaño del tipo de resultado. Por ejemplo, si `v := uint16(0x10F0)`, entonces, `uint32(int8(v)) == 0xFFFFFFF0`. La conversión siempre produce un valor válido; no hay ninguna indicación de desbordamiento.
2. Cuando se convierte un número de coma flotante a entero, la fracción se descarta (truncando a cero).
3. Al convertir un número entero o de coma flotante a un tipo de coma flotante, o un número complejo a otro tipo complejo, el valor del resultado se redondea a la precisión especificada por el tipo destino. Por ejemplo, el valor de una variable `x` de tipo `float32` se puede almacenar usando precisión adicional más allá de la de un número IEEE-754 de 32 bits, pero `float32(x)` representa el resultado de redondear el valor de `x` a precisión de 32 bits. Del mismo modo, `x + 0.1` puede utilizar más de 32 bits de precisión, pero `float32(x + 0.1)` no lo hace.

En todas las conversiones no constantes involucrando valores de coma flotante o complejos, si el tipo del resultado no puede representar el valor de la conversión es éxitosa, pero el valor del resultado depende de la implementación.

Conversiones a y desde un tipo cadena

1. La conversión de un valor entero con o sin signo a un tipo cadena produce una cadena que contiene la representación UTF-8 del entero. Los valores fuera de rango de caracteres Unicode válidos se convierten a `&quot;\uFFFD&quot;`.

```
string('a')       // "a"
string(-1)        // "\ufffd" == "\xef\xbf\xbd"
string(0xf8)      // "\u00f8" == "ø" == "\xc3\xb8"

type MiCadena string

MiCadena(0x65e5)  // "\u65e5" == "日" == "\xe6\x97\xa5"
```

1. La conversión de un sector de bytes a un tipo cadena produce una cadena cuyos bytes sucesivos son los elementos del sector.

```
string([]byte{'h', '\xc3', '\xb8', 'l', 'a'})     // "høla"
string([]byte{})                                  // ""
string([]byte(nil))                               // ""

type MisBytes []byte

string(MisBytes{'h', '\xc3', '\xb8', 'l', 'a'})   // "høla"
```

1. La conversión de un sector de runes a un tipo cadena produce una cadena que es la concatenación de los valores runes individuales convertidos a cadenas.

```
string([]rune{0x767d, 0x9d6c, 0x7fd4})   // "\u767d\u9d6c\u7fd4" == "白鵬翔"
string([]rune{})                         // ""
string([]rune(nil))                      // ""

type MisRunes []rune

string(MisRunes{0x767d, 0x9d6c, 0x7fd4})  // "\u767d\u9d6c\u7fd4" == "白鵬翔"
```

1. La conversión de un valor de tipo cadena a un sector de tipo bytes produce un sector cuyos elementos sucesivos son los bytes de la cadena.

```
[]byte("høla")    // []byte{'h', '\xc3', '\xb8', 'l', 'a'}
[]byte("")        // []byte{}

MisBytes("høla")  // []byte{'h', '\xc3', '\xb8', 'l', 'a'}
```

1. La conversión de un valor de tipo cadena a un sector de tipo runes produce un sector que contiene los caracteres Unicode individuales de la cadena.

```
[]rune(MiCadena("白鵬翔"))  // []rune{0x767d, 0x9d6c, 0x7fd4}
[]rune("")                 // []rune{}
MisRunes("白鵬翔")          // []rune{0x767d, 0x9d6c, 0x7fd4}
```

Expresiones constantes

Las expresiones constantes pueden contener solo operandos constantes y se evalúan en tiempo de compilación.

Las constantes booleanas sin tipo, numéricas y cadena se pueden utilizar como operandos siempre que sea legal el uso de un operando lógico, numérico o de tipo cadena, respectivamente. A excepción de las operaciones de desplazamiento, si los operandos de una operación binaria son diferentes clases de constantes sin tipo, la operación y, para operaciones no lógicas, el resultado utiliza el tipo que aparece más adelante en esta lista: número entero, rune, de coma flotante o complejo. Por ejemplo, una constante entera sin tipo dividida entre una constante compleja sin tipo produce una constante compleja sin tipo.

Una comparación constante siempre produce una constante lógica sin tipo. Si el operando de la izquierda de una expresión de desplazamiento constante es una constante sin tipo, el resultado es una constante entera; de lo contrario es una constante del mismo tipo que el operando de la izquierda, mismo que debe ser de tipo entero. La aplicación de todos los demás operadores a constantes sin tipo resultan en una constante sin tipo de la misma clase (es decir, una constante booleana, entera, de coma flotante, compleja o cadena).

```
const a = 2 + 3.0          // a == 5.0   (constante de coma flotante sin tipo)
const b = 15 / 4           // b == 3     (constante entera sin tipo)
const c = 15 / 4.0         // c == 3.75  (constante de coma flotante sin tipo)
const 0 float64 = 3/2      // 0 == 1.0   (tipo float64, 3/2 es una división
                           //                                      entera)
const Π float64 = 3/2.     // Π == 1.5   (tipo float64, 3/2. es la división de
                           //                                 coma flotante)
const d = 1 << 3.0         // d == 8     (constante entera sin tipo)
const e = 1.0 << 3         // e == 8     (constante entera sin tipo)
const f = int32(1) << 33   // ilegal     (constante 8589934592 desbordamiento
                           //                                       int32)
const g = float64(2) >> 1  // ilegal     (float64(2) es una constante de coma
                           //                               flotante con tipo)
const h = "foo" > "bar"    // h == true  (constante lógica sin tipo)
const j = true             // j == true  (constante lógica sin tipo)
const k = 'w' + 1          // k == 'x'   (constante rune sin tipo)
const l = "hi"             // l == "hi"  (constante cadena sin tipo)
const m = string(k)        // m == "x"   (tipo cadena)
const Σ = 1 - 0.707i       //            (constante compleja sin tipo)
const Δ = Σ + 2.0e-4       //            (constante compleja sin tipo)
const Φ = iota*1i - 1/1i   //            (constante compleja sin tipo)
```

La aplicación de la función incorporada `complex` a un número entero sin tipo, rune o a constantes de coma flotante produce una constante compleja sin tipo.

```
const ic = complex(0, c)   // ic == 3.75i  (constante compleja sin tipo)
const i0 = complex(0, 0)   // i0 == 1i     (tipo complex128)
```

Las expresiones constantes siempre se evalúan con exactitud; los valores intermedios y las constantes en sí mismas pueden requerir precisión significativamente mayor que la compatible con cualquier tipo predeclarado en el lenguaje. Las siguientes son declaraciones lícitas:

```
const Enorme = 1 << 100         // Enorme == 1267650600228229401496703205376
                                //               (constante entera sin tipo)
const Cuatro int8 = Enorme >> 98  // Cuatro == 4                (tipo int8)
```

El divisor de una operación de división o residuo constante no debe ser cero:

```
3.14 / 0.0   // ilegal: división entre cero
```

Los valores de constantes *tipadas* siempre se deben representar con toda precisión como valores del tipo constante. Las siguientes expresiones constantes son ilegales:

```
uint(-1)       // -1 no se puede representar como uint
int(3.14)      // 3.14 no se puede representar como uint
int64(Enorme)  // 1267650600228229401496703205376 no se puede representar
               //                                              como int64
Cuatro * 300   // operando 300 no se puede representar como int8 (tipo de
               //                                                 Cuatro)
Cuatro * 100   // producto 400 no se puede representar como int8 (tipo de
               //                                                 Cuatro)
```

La máscara utilizada por el operador complemento unario bit a bit `^` coincide con la regla para los no constantes: la máscara es toda 1s para las constantes sin signo y -1 para las constantes con signo y sin tipo.

```
^1         // número entero constante sin tipo, igual a -2
uint8(^1)  // ilegal: lo mismo que uint8(-2), -2 no se puede representar
                                                          como uint8
^uint8(1)  // constante uint8 con tipo, igual que
                                          0xFF ^ uint8(1) = uint8(0xFE)
int8(^1)   // igual que int8(-2)
^int8(1)   // igual que -1 ^ int8(1) = -2
```

> Restricción de implementación: Un compilador puede utilizar redondeo al calcular coma flotante sin tipo o expresiones constantes complejas; ve la restricción de implementación en la sección de constantes. Este redondeo puede causar una expresión constante de coma flotante para ser válida en el contexto de un número entero, incluso si sería integral cuando se calcula usando una precisión infinita.

Orden de evaluación

A nivel de paquete, la iniciación de dependencias determina el orden de evaluación de expresiones de iniciación individuales en las declaraciones de variables. De lo contrario, al evaluar los operandos de una expresión, asignación o instrucción return, todas las llamadas a función, las llamadas a métodos y operaciones de comunicación se evalúan en orden léxico de izquierda a derecha.

Por ejemplo, en la asignación (función local)

```
y[f()], ok = g(h(), i()+x[j()], <-c), k()

y [f ()], ok = g (h (), i () + `x` [j ()],

a := 1

f := func() int { a++; return a }

x := []int{a, f()}            // x puede ser [1, 2] o [2, 2]: el orden de
                              // evaluación entre a y f() no está
                              // especificado
m := map[int]int{a: 1, a: 2}  // m puede ser {2: 1} o {2: 2}: el orden de
                              // evaluación entre la asignación de los
                              // dos mapas no está especificado
n := map[int]int{a: f()}      // n puede ser {2: 3} o {3: 3}: el orden de
                              // evaluación entre la clave y el valor no
                              // está especificado
```

A nivel de paquete, la iniciación de dependencias redefine la regla de izquierda a derecha para las expresiones de iniciación individuales, pero no para operandos dentro de cada expresión:

```
var a, b, c = f() + v(), g(), sqr(u()) + v()

func f() int        { return c }
func g() int        { return a }
func sqr(x int) int { return x*x }

// las funciones u y v son independientes de todas las otras variables
// y funciones
```

Las llamadas a funciones suceden en el orden `u()`, `sqr()`, `v()`, `f()`, `v()` y `g()`.

Operaciones de coma flotante en una sola expresión se evalúan de acuerdo a la asociatividad de los operadores. Los paréntesis explícitos afectan a la evaluación por razones imperiosas de la asociatividad predeterminada. En la expresión `x + (y + z)` la adición `y + z` se realiza antes de sumar `x`.

Instrucciones

Control de ejecución de instrucciones.

```
Instrucción =
    Declaración | DeclaEtiquetada | SimpleInstru |
    InstruGo | InstruReturn | InstruBreak | InstruContinue | InstruGoto |
    InstruFallthrough | Bloque | InstruIf | InstruSwitch | InstruSelect |
    InstruFor |    InstruDefer .
SimpleInstru = InstruVacía | DeclaExpresión | InstruEnvío | InstruIncDec |
              Asignación | DeclaVarCorta .
```

Instrucción de terminación

Una instrucción de terminación es una de las siguientes:

1. Una instrucción "return" o "goto".
2. Una llamada a la función integrada `panic`.
3. Un bloque en el cual la lista de declaraciones termina en una declaración de terminación.
4. Una instrucción "if" en la que:
   - la rama "else" está presente y
   - ambas ramas tienen instrucciones de terminación.
5. Una instrucción "for" en la que:
   - no hay instrucciones "break" refiriéndose a la instrucción "for" y
   - la condición del bucle está ausente.
6. Una instrucción "switch" en la que:
   - ho hay instrucciones "break" refiriéndose a la instrucción "switch",
   - hay un caso `default` y
   - la lista de instrucciones en cada caso, incluyendo a `default`, terminan en una instrucción de terminación o una posiblemente etiquetada instrucción "fallthrough".
7. Una instrucción "select" en la que:
   - no hay declaraciones "break" refiriéndose a la instrucción "select" y
   - las listas de instrucciones en cada caso, incluyendo a `default` si está presente, terminan en una instrucción de terminación.

8. Una instrucción etiquetada vinculada a una instrucción de terminación.

Todas las demás declaraciones no están terminando.

Una lista de instrucciones termina en una instrucción de terminación si la lista no está vacía y su instrucción final está terminando.

Instrucciones vacías

La instrucción vacía no hace nada.

```
InstruVacía = .
```

Instrucciones etiquetadas

Una instrucción etiquetada puede ser el destino de una instrucción `goto`, `break` o `continue`.

```
InstruEtiquetada = Etiqueta ":" Declaración .
Etiqueta         = identificador .
```

```
Error: log.Panic("encontré un error")
```

Expresiones instrucción

A excepción de funciones integradas específicas, las llamadas a función, métodos y operaciones de recepción pueden aparecer en el contexto de una instrucción. Tales instrucciones pueden estar entre paréntesis.

```
InstruExpresión = Expresión .
```

Las siguientes funciones incorporadas no están permitidas en el contexto de una instrucción:

```
append cap complex imag len make new real
unsafe.Alignof unsafe.Offsetof unsafe.Sizeof
```

```
h(x+y)
f.Close()
<-ch
(<-ch)
len("foo")  // ilegal si len es la función integrada
```

Instrucciones de envío

Una instrucción de envío transmite un valor en un canal. La expresión canal debe ser de tipo canal, la dirección del canal debe permitir las operaciones de envío y el tipo de valor a enviar debe ser asignable al tipo de elemento del canal.

```
InstruEnvío = Canal "<-" Expresión .
Canal       = Expresión .
```

Tanto el canal como la expresión de valor se evalúan antes de que comience la comunicación. Se bloquea la comunicación hasta que pueda proceder el envío. Un envío en un canal sin búfer puede proceder si un receptor está listo. Un envío en un canal con búfer puede proceder si hay espacio en el búfer. Un envío en un canal cerrado procede provocando pánico en tiempo de ejecución. Un envío a un canal `nil` se bloquea para siempre.

```
ch <- 3  // envía el valor 3 al canal ch
```

Instrucciones IncDec

Las instrucciones "++" y "--" incrementan o disminuyen sus operandos por la constante sin tipo `1`. Al igual que con una asignación, el operando debe ser direccionable o una expresión de índice de mapa.

```
InstruIncDec = Expresión ( "++" | "--" ) .
```

Las siguientes instrucciones de asignación son semánticamente equivalentes:

```
Instrucción IncDec  Asignación
x++                 x += 1
x--                 x -= 1
```

Asignaciones

```
asignación    = ListaExpresión op_asignación ListaExpresión .
op_asignación = [ op_adi | op_mul ] "=" .
```

Cada operando del lado izquierdo debe ser direccionable, una expresión de índice de mapa o (solo para asignaciones `=`) el identificador blanco. Los operandos pueden estar entre paréntesis.

```
x = 1
*p = f()
a[i] = 23
(k) = <-ch  // lo mismo que: k = <-ch
```

Una *operación de asignación* `x` *op* `=` `y` donde *op* es una operación aritmética binaria es equivalente a `x` `=` `x` *op* `y` sino que solo evalúa a `x` una vez. La construcción *op* `=` es un solo símbolo. En las operaciones de asignación, tanto la lista de expresiones izquierda como la derecha deben contener exactamente una expresión de valor único y la expresión de la izquierda no debe ser el identificador blanco.

```
a[i] <<= 2
i &^= 1<<n
```

Una tupla de asignación establece los elementos individuales de una operación de varios valores a una lista de variables. Hay dos formas. En la primera, el operando de la derecha es una sola expresión de varios valores, tal como una llamada a función, una operación de canal o mapa, o un tipo aserción. El número de operandos en el lado de la mano izquierda debe coincidir con el número de valores. Por ejemplo, si `f` es una función que devuelve dos valores:

```
x, y = f()
```

asigna el primer valor a `x` y el segundo a `y`. En la segunda forma, el número de operandos de la izquierda debe ser igual al número de expresiones de la derecha, cada uno de los cuales debe ser de un solo valor y la e*n*ésima expresión de la derecha se asigna al e*n*ésimo operando de la izquierda:

```
uno, dos, tres = '一', '二', '三'
```

El identificador blanco proporciona una manera de ignorar los valores del lado derecho en una asignación:

```
_ = x       // evalúa x pero lo descarta
x, _ = f()  // evalúa f() pero desecha el valor del resultado
```

La asignación procede en dos fases. En primer lugar, los operandos de las expresiones de índice e indirecciones de puntero (incluyendo indirecciones de puntero implícitas en selectores) a la izquierda y las expresiones de la derecha todas se evalúan en el orden habitual. En segundo lugar, las tareas se llevan a cabo en orden de izquierda a derecha.

```
a, b       = b, a  // intercambia a y b
x         := []int{1, 2, 3}
i         := 0
i, x[i]    = 1, 2  // fija i = 1, x[0] = 2
i          = 0
x[i], i    = 2, 1  // fija x[0] = 2, i = 1
x[0], x[0] = 1, 2  // fija x[0] = 1, luego x[0] = 2 (por tanto x[0] == 2
                   //                                      al final)
x[1], x[3] = 4, 5  // fija x[1] = 4, luego ajusta el pánico x[3] = 5.

type Punto struct { x, y int }

var p *Punto

x[2], p.x   = 6, 7  // fija x[2] = 6, luego ajusta el pánico p.x = 7
i           = 2
x           = []int{3, 5, 7}

for i, x[i] = range x {  // fija i, x[2] = 0, x[0]
    break
}

// después de este bucle, i == 0 y x == []int{3, 5, 3}
```

En las asignaciones, cada valor debe ser asignable al tipo del operando al que se le asigna, con los siguientes casos especiales:

1. Cualquier valor tipado se puede asignar al identificador blanco.
2. Si se asigna una constante sin tipo a una variable de tipo interfaz o al identificador blanco, la constante primero se convierte a su tipo predeterminado.
3. Si se asigna un valor lógico sin tipo a una variable de tipo interfaz o al identificador blanco, esta primero se convierte al tipo `bool`.

Instrucciones `if`

Las instrucciones "if" especifican la ejecución condicional de dos ramas de acuerdo con el valor de una expresión booleana. Si la expresión se evalúa como verdadera, la rama "if" se ejecuta, de lo contrario, si está presente, se ejecuta la rama "else".

```
InstruIf = "if" [ SimpleInstru ";" ] Bloque Expresión [ "else" ( InstruIf |
                                                              Bloque ) ] .
```

```
if x > máx {
    x = máx
}
```

La expresión puede estar precedida por una declaración simple, que se ejecuta antes de evaluar la expresión.

```
if x := f(); x < y {
    return x
} else if x > z {
    return z
} else {
    return y
}
```

Instrucciones `switch`

Las instrucciones "switch" proporcionan la ejecución múltivía. Una expresión o especificador de tipo se compara con los "casos" en el interior del "switch" para determinar qué rama se ejecuta.

```
InstruSwitch = InstruExpreSwitch | InstruTipoSwitch .
```

Hay dos formas: expresiones switch y tipos switch. En una expresión switch, los casos contienen expresiones que se comparan con el valor de la expresión switch. En un tipo switch, los casos contienen tipos que se comparan contra el tipo de una expresión switch especialmente anotada.

Expresiones switch

En una expresión switch, se evalúa la expresión switch y las expresiones de casos, que no necesitan ser constantes, se evalúan de izquierda a derecha y de arriba hacia abajo; el primero que es igual a la expresión switch desencadena la ejecución de las instrucciones del caso asociado; los demás casos se omiten. En el supuesto de que no haya coincidencia

se ejecutan las instrucciones del caso " `default` ". Como máximo puede haber un caso `default` y puede aparecer en cualquier parte de la instrucción "switch". Una expresión switch faltante es equivalente al valor lógico `true` .

```
InstruExpreSwitch = "switch" [ SimpleInstru ";" ] [ Expresión ] "{" {
                                                    ExpreCláusulaCase } "}" .
ExpreCláusulaCase = ExpreCasoSwitch ":" ListaInstrucciones .
ExpreCasoSwitch   = "case" ListaExpresión | "default" .
```

En una cláusula `case` o `default` , la última instrucción no vacía puede ser una (posiblemente etiquetada) instrucción "fallthrough" para indicar que el control debe fluir desde el final de esta cláusula a la primera declaración de la siguiente cláusula. De lo contrario, el control fluye al final de la instrucción "switch". Una instrucción " `fallthrough` " puede aparecer como la última declaración de todas, pero la última cláusula de una expresión switch.

La expresión puede estar precedida por una instrucción simple, que se ejecuta antes de evaluar la expresión.

```
switch etiqueta {
  default: s3()
  case 0, 1, 2, 3: s1()
  case 4, 5, 6, 7: s2()
}

switch x := f(); {  // falta expresión switch significa "true"
  case x < 0: return -x
  default: return x
}

switch {
  case x < y: f1()
  case x < z: f2()
  case x == 4: f3()
}
```

Tipos switch

Un tipo switch compara tipos en lugar de valores. Por lo demás es similar a una expresión switch. Se caracteriza por una expresión switch especial que tiene la forma de un tipo aserción utilizando la palabra reservada `type` en lugar de un tipo real:

```
switch x.(tipo) {
// casos
}
```

Entonces los casos coinciden los tipos `T` reales contra el tipo dinámico de la expresión `x`. Al igual que con los tipos aserción, `x` debe ser de tipo interfaz y cada tipo no interfaz `T` enumerado en un caso debe implementar el tipo de `x`.

```
InstruTipoSwitch  = "switch" [ SimpleInstru ";" ] ProtecTipoSwitch "{" {
                                                    TipoCláusulaCase } "}" .
ProtecTipoSwitch  = [ identificador ":=" ] ExprePrimaria "." "(" "type" ")" .
TipoCláusulaCase  = TipoCasoSwitch ":" ListaInstrucciones .
TipoCasoSwitch    = "case" ListaTipo | "default" .
ListaTipo         = Tipo { "," Tipo } .
```

El ProtecTipoSwitch puede incluir una declaración corta de variable. Cuando se utiliza esta forma, la variable se declara al comienzo del bloque implícito en cada cláusula. En las cláusulas con un caso enumerando exactamente un tipo, la variable tiene ese tipo; de lo contrario, la variable tiene el tipo de la expresión en el ProtecTipoSwitch.

El tipo en un caso puede ser `nil`; ese caso se utiliza cuando la expresión en el ProtecTipoSwitch es un valor interfaz `nil`.

Dada una expresión `x` de tipo `interface{}`, el siguiente tipo switch:

```
switch i := x.(tipo) {
  case nil:
    printString("x es nil")                // el tipo de i es el tipo de
                                           //            x(interface{})
  case int:
    printInt(i)                            // el tipo de i es int
  case float64:
    printFloat64(i)                        // el tipo de i es float64
  case func(int) float64:
    printFunction(i)                       // el tipo de i es func(int) float64
  case bool, string:
    printString("tipo es bool o string")   // el tipo de i es el tipo de
                                           //            x(interface{})
default:
    printString("deconozco el tipo")      // el tipo de i es el tipo de
                                           //           x(interface{})
}
```

Se podría reescribir de la siguiente manera:

```
v := x  // x se evalúa exactamente una vez

if v == nil {
    i := v                                  // el tipo de i es el tipo de
                                            //            x(interface{})
    printString("x es nil")
} else if i, isInt := v.(int); isInt {
    printInt(i)                             // el tipo de i es int
} else if i, isFloat64 := v.(float64); isFloat64 {
    printFloat64(i)                         // el tipo de i es float64
} else if i, isFunc := v.(func(int) float64); isFunc {
    printFunction(i)                        // el tipo de i es func(int)
                                            //                      float64
} else {
    _, isBool := v.(bool)
    _, isString := v.(string)
    if isBool || isString {
        i := v                              // el tipo de i es el atipo de
                                            //            x(interface{})
        printString("tipo es bool o string")
    } else {
        i := v                              // el tipo de i es el tipo de
                                            //            x(interface{})
        printString("desconozco el tipo")
    }
}
```

El protector del tipo switch puede estar precedido por una declaración simple, que se ejecuta antes de que se evalúe el protector.

La declaración "fallthrough" no está permitida en un tipo switch.

Instrucciones `for`

Una instrucción "for" especifica la ejecución repetida de un bloque. La iteración es controlada por una condición, una cláusula "for" o una cláusula "range".

```
InstruFor  = "for" [ Condición | CláusulaFor | CláusulaRange ] Bloque .
Condición  = Expresión .
```

En su forma más simple, una instrucción "for" especifica la ejecución repetida de un bloque, siempre que una condición booleana se evalúe como verdadera. La condición se evalúa antes de cada iteración. Si la condición está ausente, es equivalente al valor lógico `true`.

```
for a < b {
    a *= 2
}
```

Una instrucción "for" con una *CláusulaFor* también es controlada por su condición, pero además, como asignación puede especificar una expresión *inicio* y una *destino*, así como una declaracion de incremento o decremento. La expresión de inicio puede ser una declaración corta de variables, pero la declaración destino no debe. Las variables declaradas por la declaración de inicio se vuelven a utilizar en cada iteración.

```
CláusulaFor  = [ ExpreInicio ] ";" [ Condición ] ";" [ ExpreDestino ] .
ExpreInicio  = SimpleInstru .
ExpreDestino = SimpleInstru .
```

```
for i := 0; i < 10; i++ {
    f(i)
}
```

Si no está vacía, la declaración de inicio se ejecuta una vez antes de evaluar la condición en la primera iteración; la declaración destino se evalúa después de cada ejecución del bloque (y únicamente si el bloque se ejecuta). Cualquier elemento de la CláusulaFor puede estar vacío pero los puntos y comas son obligatorios a menos que solo haya una condición. Si la condición está ausente, es equivalente al valor lógico `true`.

```
for cond { S() }    es lo mismo que    for ; cond ; { S() }
for      { S() }    es lo mismo que    for true     { S() }
```

Una instrucción "for" con una cláusula "range" itera a través todas las entradas de un arreglo, sector, cadena, mapa o los valores recibidos en un canal. Por cada entrada esta asigna *valores de iteración* a las *variables de iteración* correspondientes si están presentes y luego ejecuta el bloque.

```
CláusulaRange = [ ListaExpresión "=" | ListaIdentificador ":=" ]
                                                    "range" Expresión .
```

La expresión de la derecha en la cláusula "range" se conoce como la *expresión de rango*, que bien puede ser un arreglo, puntero a un arreglo, sector, cadena, mapa o canal que permite operaciones de recepción. Al igual que con una asignación, si están presentes los operandos de la izquierda deben ser direccionables o expresiones índice de mapa; estos denotan las variables de iteración. Si la expresión rango es un canal, por lo menos se

permite una variable de iteración, de lo contrario puede haber un máximo de dos. Si la última variable de iteración es el identificador blanco, la cláusula rango es equivalente a la misma cláusula sin ese identificador.

La expresión de rango se evalúa una vez antes de comenzar el bucle, con una excepción: si la expresión rango es un vector o un puntero a un arreglo y como máximo está presente una variable de iteración, sólo se evalúa la longitud de la expresión de rango; si esta longitud es constante, por definición no se evaluará la expresión en si misma.

Las llamadas a función de la izquierda son evaluadas una vez por iteración. Por cada iteración, los valores de iteración se producen como sigue si las respectivas variables de iteración están presentes:

```
Expresión range                          1er valor           2do valor
arreglo o sector  a  [n]E, *[n]E, o []E  índice    i  int   a[i]       E
cadena            s  tipo string         índice    i  int   ve rune abajo
mapa              m  map[K]V             clave     k  K     m[k]       V
canal             c  chan E, <-chan E    elemento  e  E
```

1. Para un arreglo, puntero a arreglo o valor del sector `a`, los valores índice de iteración se producen en orden creciente, comenzando en el índice del elemento 0. Si a lo sumo está presente una variable de iteración, el bucle de iteración range produce valores de 0 hasta `len(a)-1` y no tiene índice en el arreglo o sector en sí mismo. Para un sector `nil`, el número de iteraciones es 0.
2. Para un valor de cadena, la cláusula "range" itera sobre los caracteres Unicode en la cadena a partir del índice del byte 0. En sucesivas iteraciones, el valor del índice será el índice del primer byte de los sucesivos caracteres codificación UTF-8 en la cadena y el segundo valor, de tipo `rune`, será el valor del carácter correspondiente. Si la iteración se encuentra con una secuencia UTF-8 no válida, el segundo valor será `0xFFFD`, el carácter Unicode de reemplazo y la siguiente iteración avanzará un solo byte en la cadena.
3. La orden de iteración sobre mapas no se especifica y no se garantiza que sea la misma de una iteración a la siguiente. Si se eliminan entradas correlacionadas que aún no se han alcanzado durante la iteración, no se producirán los valores de iteración correspondientes. Si se crean entradas en el mapa durante la iteración, esa entrada se puede producir durante la iteración o se puede omitir. La elección puede variar para cada entrada creada y de una iteración a la siguiente. Si el mapa es `nil`, el número de iteraciones es 0.
4. Para canales, los valores de iteración producidos son los valores sucesivos enviados por el canal hasta que el canal esté cerrado. si el canal es `nil`, la expresión range se bloquea para siempre.

Los valores de iteración se asignan a las respectivas variables de iteración como en una instrucción de asignación.

Las variables de iteración se pueden declarar por cláusula "range" utilizando una forma de declaración corta de variables (`:=`). En este caso, sus tipos se fijan a los tipos de los respectivos valores de iteración y su ámbito es el bloque de la declaración "for"; estas se vuelven a utilizar en cada iteración. Si las variables de iteración se declaran fuera de la declaración "for", después de la ejecución sus valores serán los de la última iteración.

```
var datosdeprueba *struct {
    a *[7]int
}

for i, _ := range datosdeprueba.a {
    // datosdeprueba.a nunca se evalúa; len(datosdeprueba.a) es constante
    // i rangos desde 0 hasta 6
    f(i)
}

var a [10]string
for i, s := range a {
    // el tipo de i es int
    // el tipo de s es string
    // s == a[i]
    g(i, s)
}

var clave string
var val interface {}  // el tipo del valor de m es asignable a val

m := map[string]int{"lun":0, "mar":1, "mie":2, "jue":3, "vie":4,
                    "sáb":5, "dom":6}

for clave, val = range m {
    h(clave, val)
}
// clave == última clave encontrada en la iteración del mapa
// val == map[clave]

var ch chan Trabajo = productor()
for w := range ch {
    hazTrabajo(w)
}

// canal vacío
for range ch {}
```

Instrucciones `go`

Una instrucción " `go` " comienza la ejecución de una llamada a función como un hilo de control concurrente independiente, o *rutinago*, dentro del mismo espacio de direcciones.

```
InstruGo = "go" Expresión .
```

La expresión debe ser una llamada a función o método; no puede estar entre paréntesis. Las llamadas a funciones integradas están restringidas en cuanto a las declaraciones de expresión.

El valor de la función y los parámetros se evalúan como de costumbre en la llamada a la rutinago, pero a diferencia de una llamada regular, la ejecución del programa no espera a que la función invocada termine. En lugar de ello, la función comienza a ejecutarse de forma independiente en una nueva rutinago. Cuando la función termina, su rutinago también termina. Si la función tiene algúnos valores de retorno, serán descartados en cuanto la función termine.

```
go Server()
go func(ch chan<- bool) { for { sleep(10); ch <- true; }} (c)
```

Instrucciones `select`

Una instrucción "select" elige cual de un conjunto de posibles operaciones de envío o recepción procederá. Es similar a una declaración "switch" pero en la cual todos los casos se refieren a operaciones de comunicación.

```
InstruSelect         = "select" "{" { CláusulaComunicación } "}" .
CláusulaComunicación = CasoComunicación ":" ListaInstrucciones .
CasoComunicación     = "case" ( InstruEnvío | InstruRecep ) | "default" .
InstruRecep  = [ ListaExpresión "=" | ListaIdentificador ":=" ] ExpreRecep .
ExpreRecep   = Expresión .
```

Un caso con una InstruRecep puede asignar el resultado de una ExpreRecep a una o dos variables, las cuales se pueden declarar utilizando una declaración corta de variables. La ExpreRecep debe ser una (posiblemente entre paréntesis) operación de recepción. Como máximo puede haber un caso `default` y puede aparecer en cualquier parte de la lista de casos.

La ejecución de una declaración "select" procede en varias etapas:

1. Para todos los casos en la declaración, los operandos de canales de operaciones de recepción y el canal y las expresiones de envío del lado de la mano derecha son evaluadas exactamente una vez, en el orden de la fuente, al entrar en la declaración

"select". El resultado es un conjunto de canales para recibir desde o enviar a, y los valores correspondientes a enviar. Cualquier efecto secundario en la evaluación se producirá independientemente de que (si las hay) se seleccione la operación de comunicación para proceder. Las expresiones en el lado izquierdo de una InstruRecep con una declaración corta de variables o asignación aún no se han evaluado.

2. Si una o más de las comunicaciones puede proceder, se elige una sola, la que pueda proceder a través de una selección pseudoaleatoria uniforme. De lo contrario, si hay un caso predeterminado, se elige ese caso. Si no hay ningún caso predeterminado, la declaración "Select" se bloquea hasta que al menos una de las comunicaciones pueda continuar.
3. A menos que el caso seleccionado sea el caso predeterminado, se ejecuta la operación de comunicación respectiva.
4. Si el caso seleccionado es una InstruRecep con una declaración corta de variable o es una asignación, las expresiones de la parte izquierda se evalúan y el valor (o valores) recibidos serán asignados.
5. Se ejecuta la lista de instrucciones del caso seleccionado.

Puesto que la comunicación en canales `nil` nunca puede proceder, una "select" con únicamente canales `nil` y sin un caso predeterminado se bloquea por siempre.

```
var a []int
var c, c1, c2, c3, c4 chan int
var i1, i2 int

select {
  case i1 = <-c1:
    print("recibí ", i1, " desde c1\n")
  case c2 <- i2:
    print("envié ", i2, " a c2\n")
  case i3, ok := (<-c3):  // igual que: i3, ok := <-c3
    if ok {
        print("recibí ", i3, " desde c3\n")
    } else {
        print("c3 está cerrado\n")
    }
  case a[f()] = <-c4:
    // mismo que:
    // case t := <-c4
    //    a[f()] = t
  default:
    print("no hay comunicación\n")
}

for {  // envía secuencias aleatorias de bits a c
    select {
      case c <- 0:  // observa: no hay instrucción,
                    // no hay fallthrough,
                    // no hay repliegue de casos
      case c <- 1:
    }
}

select {}  // bloquea por siempre
```

Instrucciones `return`

Una instrucción "return" en una función `F` termina la ejecución de `F` y opcionalmente proporciona uno o más valores como resultado. Cualquier función diferida por `F` se ejecuta antes de que `F` regrese a su invocador.

```
InstruReturn = "return" [ ListaExpresión ] .
```

En una función sin un tipo de resultado, una instrucción "return" no deberá especificar ningún valor de resultado.

```
func sinResultado() {
    return
}
```

Hay tres formas de devolver valores de una función con un tipo de resultado:

1. El valor o valores de retorno se pueden enumerar explícitamente en la instrucción "return". Cada expresión debe ser de un solo valor y asignable al elemento correspondiente del tipo de resultado de la función.

   ```
   func fSimple() int {
       return 2
   }

   func fCompleja() (re float64, im float64) {
       return -7.0, -4.0
   }
   ```

2. La lista de expresiones en la instrucción "return" puede ser una sola llamada a una función multivalores. El efecto es como si cada valor devuelto por la función se asignara a una variable temporal con el tipo del valor respectivo, seguido de una instrucción "return" enumerando estas variables, momento en que se aplican las reglas del caso anterior.

   ```
   func fCompleja2() (re float64, im float64) {
       return fCompleja()
   }
   ```

3. La lista de expresiones puede estar vacía si el tipo de resultado de la función especifica nombres para sus parámetros de resultado. Los parámetros de resultado actúan como variables locales ordinarias y la función le puede asignar valores cuando sea necesario. La instrucción "return" devuelve los valores de esas variables.

   ```
   func fCompleja3() (re float64, im float64) {
       re = 7.0
       im = 4.0
       return
   }

   func (devnull) Escribe(p []byte) (n int, _ error) {
       n = len(p)
       return
   }
   ```

Independientemente de la forma en que se declaren, todos los valores de los resultados se inician a los valores cero para su tipo al entrar a la función. Una instrucción "return" que especifica los resultados establece los parámetros de resultado antes de ejecutar cualquiera de las funciones diferidas.

> Restricción de implementación: Un compilador podrá invalidar una lista de expresiones vacía en una instrucción "return" si una entidad diferente (constante, tipo o variable) con el mismo nombre que un parámetro de resultado está en el ámbito del lugar de la devolución.

```
func f(n int) (res int, err error) {
    if _, err := f(n-1); err != nil {
        return  // func f (n int) (res int, err error) {
    if _, err: = f (n-1); err! = nil {
        return // instrucción return no válida: err es ensombrecido
    }
    return
}
```

Instrucciones `break`

Una instrucción "break" termina la ejecución de la instrucción "for", "switch" o "select" más interna dentro de la misma función.

```
InstruBreak = "break" [ Etiqueta ] .
```

Si hay una etiqueta, debe ser la de una instrucción envolvente "for", "switch" o "select", que es en dónde se tiene que terminar la ejecución.

```
BucleExterno:
    for i = 0; i < n; i++ {
        for j = 0; j < m; j++ {
            switch a[i][j] {
            case nil:
                estado = Error
                break BucleExterno
            case item:
                estado = Encontrado
                break BucleExterno
            }
        }
    }
```

Instrucciones `continue`

Una instrucción "continue", comienza la siguiente iteración del bucle "for" más interno en su posterior instrucción. El bucle "for" debe estar dentro de la misma función.

```
InstruContinue = "continue" [ Etiqueta ] .
```

Si hay una etiqueta, debe ser la de una instrucción "for" envolvente y que sea en la que tiene que avanzar la ejecución.

```
BucleRenglón:
    for y, renglón := range renglones {
        for x, datos := range renglón {
            if datos == finDeRenglón {
                continue BucleRenglón
            }
            renglón[x] = datos + parcial(x, y)
        }
    }
```

Instrucciones `goto`

Una instrucción "goto" transfiere el control a la declaración con la etiqueta correspondiente dentro de la misma función.

```
InstruGoto = "goto" Etiqueta .
```

```
goto Error
```

La ejecución de la instrucción "goto" no debe provocar que ninguna variable entre en el ámbito que no estaba ya en su alcance en el punto de la `goto`. Por ejemplo, veamos esta declaración:

```
goto L  // MAL
v := 3
L:
```

es errónea porque salta a la etiqueta `L` omitiendo la creación de `v`.

Una instrucción "goto" declarada fuera de un bloque no puede saltar a una etiqueta dentro de este bloque. Por ejemplo, veamos esta declaración:

```
if n%2 == 1 {
    goto L1
}

for n > 0 {
    f()
    n--
L1:
    f()
    n--
}
```

es errónea porque la etiqueta `L1` está dentro del bloque de la instrucción "for", pero la instrucción "`goto`" no.

Instrucciones `fallthrough`

Una instrucción "fallthrough" transfiere el control a la primera declaración de la siguiente cláusula caso en una expresión "switch". Solo se puede utilizar como la instrucción no vacía final en dicha cláusula.

```
InstruFallthrough = "fallthrough" .
```

Instrucciones `defer`

Una instrucción "defer" invoca a una función cuya ejecución se difiere hasta momentos antes de que la función regrese, ya sea porque la función envolvente ejecutó una instrucción return, se alcanzó el final del cuerpo de la función o porque la rutinago correspondiente está entrando en pánico.

```
InstruDefer = "defer" Expresión .
```

La expresión debe ser una llamada a función o método; no puede estar entre paréntesis. Las llamadas a funciones integradas están restringidas en cuanto a las declaraciones de expresión.

Cada vez que se ejecuta una declaración "defer", el valor de la función y los parámetros de la llamada se evalúan de la manera usual y se guarda de nuevo, pero la función real no se invoca. En su lugar, las funciones diferidas se invocan inmediatamente antes de que la función envolvente regrese, en orden inverso al que fueron diferidas. Si un valor de función diferida evalúa a `nil`, se entra en pánico cuando la función se invoca, no cuando se ejecute la instrucción "defer".

Por ejemplo, si la función diferida es una función literal y la función que la rodea tiene parámetros nombrados como resultado esos están en el ámbito de la función literal, la función diferida puede acceder y modificar los parámetros de resultado antes de que sean devueltos. Si la función diferida no tiene ningún valor de retorno, estos serán descartados al terminar la función. (Ve también la sección sobre el manejo de pánico).

```
bloquea(l)

defer desbloquea(l)  // el desbloqueo ocurre antes que la
                     // función envolvente regrese

// imprime 3 2 1 0 antes de que la función envolvente regrese
for i := 0; i <= 3; i++ {
    defer fmt.Print(i)
}

// f devuelve 1
func f() (result int) {
    defer func() {
        result++
    }()
    return 0
}
```

Funciones integradas

Las funciones incorporadas son predeclaradas. Se llaman como cualquier otra función, pero algunas de ellas aceptan un tipo en lugar de una expresión como primer argumento.

Las funciones incorporadas no tienen tipos estándar Go, por lo que sólo pueden aparecer en expresiones de llamada; no se pueden utilizar como valores de función.

`close`

Para un canal `c`, la función incorporada `close(c)` registra que no más valores serán enviados en el canal. Es un error si `c` únicamente es un canal de recepción. Enviar a o cerrar un canal cerrado provoca pánico en tiempo de ejecución. Cerrar el canal `nil` también provoca pánico en tiempo de ejecución. Después de llamar a `close` y después de que se hayan recibido todos los valores enviados anteriormente, las operaciones de recepción devolverán el valor cero para el tipo de canal sin bloquearlo. Las operaciones de recepción multivalor devuelven un valor recibido junto con una indicación de si el canal está cerrado.

Longitud y capacidad

Las funciones incorporadas `len` y `cap` toman argumentos de varios tipos y devuelven un resultado de tipo `int`. La implementación garantiza que el resultado siempre encaja en un `int`.

```
Llamada   Tipo de argumento   Resultado
len(s)    tipo string         longitud de la cadena en bytes
          [n]T, *[n]T         longitud del arreglo    (== n)
          []T                 longitud del sector
          map[K]T             longitud del mapa (el número de claves definido)
          chan T              número de elementos en cola en el búfer del canal
cap(s)    [n]T, *[n]T         Longitud del arreglo (== n)
          []T                 capacidad del sector
          chan T              capacidad del búfer del canal
```

La capacidad de un sector es el número de elementos para los cuales hay espacio reservado en el arreglo subyacente. En cualquier momento mantiene la siguiente relación:

```
0 <= len(s) <= cap(s)
```

La longitud de un sector `nil`, mapa o canal es 0. La capacidad de un sector o canal `nil` es 0.

La expresión `len(s)` es constante si `s` es una cadena constante. Las expresiones `len(s)` y `cap(s)` son constantes si el tipo de `s` es un arreglo o puntero a un arreglo y la expresión `s` no contiene llamadas a función en un canal receptor o (no constante); en ese caso, `s` no se evalúa. De lo contrario, las invocaciones de `len` y `cap` no son constantes y `s` se evalúa.

```
const (
    c1 = imag(2i)                    // imag(2i) = 2.0 es una constante
    c2 = len([10]float64{2})         // [10]float64{2} no contiene
                                     // llamadas a función
    c3 = len([10]float64{c1})        // [10]float64{c1} no contiene
                                     // llamadas a función
    c4 = len([10]float64{imag(2i)}) // imag(2i) es una constante y no
                                     // se emite llamada a función
    c5 = len([10]float64{imag(z)})   // no válida: imag(z) es una
                                     // llamada a función (no constante)
)

var z complex128
```

Asignación

La función incorporada `new` toma un tipo `T`, reserva almacenamiento para una variable de ese tipo en tiempo de ejecución y devuelve un valor de tipo `*T` que apunta a ella misma. La variable se inicia como se describe en la sección sobre los valores iniciales.

```
new(T)
```

Por ejemplo:

```
type S struct { a int; b float64 }
new(S)
```

reserva almacenamiento para una variable de tipo `S`, lo inicia (`a=0`, `b=0.0`) y devuelve un valor de tipo `*S` que contiene la dirección de esa ubicación.

Creando sectores, mapas y canales

La función integrada `make` toma un tipo `T`, el cual debe ser un tipo sector, mapa o canal, opcionalmente seguido de una lista de expresiones de tipos específicos. Devuelve un valor de tipo `T` (no `*T`). La memoria se inicia como se describe en la sección sobre los valores iniciales.

```
Llamada          Tipo de T  Resultado
make(T, n)       sector     sector de tipo T con longitud n y capacidad n
make(T, n, m)    sector     sector de tipo T con longitud n y capacidad m
make(T)          mapa       mapa de tipo T
make(T, n)       mapa       mapa de tipo T con espacio inicial para n elementos
make(T)          canal      canal sin búfer de tipo T
make(T, n)       canal      canal con búfer de tipo T, tamaño del búfer n
```

Los argumentos de tamaño `n` y `m` deben ser de tipo entero o sin tipo. Un argumento constante para tamaño debe no ser negativo y representable por un valor de tipo `int`. Si se proporcionan ambos `n` y `m` y son constantes, entonces `n` debe ser mayor que `m`. Si `n` es negativo o mayor que `m` en tiempo de ejecución, se produce pánico en tiempo de ejecución.

```
s := make([]int, 10, 100)       // sector con len(s) == 10, cap(s) == 100
s := make([]int, 1e3)           // sector con len(s) == cap(s) == 1000
s := make([]int, 1<<63)         // ilegal: len(s) no es representable por un valor de tip
s := make([]int, 10, 0)         // ilegal: len(s) > cap(s)
c := make(chan int, 10)         // canal con un búfer de tamaño 10
m := make(map[string]int, 100)  // mapa con espacio inicial para 100 elementos
```

Anexando a y copiando sectores

Las funciones incorporadas `append` y `copy` ayudan en las operaciones de sector comunes. Para ambas funciones, el resultado es independiente de si la memoria referenciada por los argumentos se superpone.

La función variadica `append` añade cero o más valores `x` a `s` de tipo `S`, el cual debe ser un tipo de sector y devolver el sector resultante, también de tipo `S`. Los valores `x` se pasan a un parámetro de tipo `...T` dónde `T` es el tipo de elemento de `S` y se aplican las respectivas reglas para el paso de parámetros. Como caso especial, `append` también acepta un primer argumento asignable al tipo `[]byte` con un segundo argumento de tipo cadena seguido por `...`. Esta forma agrega los bytes de la cadena.

```
append(s S, x ...T) S  // T es el tipo de elemento de S
```

Si la capacidad de `s` no es lo suficientemente grande como para ajustarse a los valores adicionales, `append` asigna un nuevo arreglo subyacente lo suficientemente grande para que se ajuste tanto a los elementos existentes del sector como a los valores adicionales. De lo contrario, `append` reutiliza el arreglo subyacente.

```
s0 := []int{0, 0}
s1 := append(s0, 2)                // agrega un único elemento
                                   // s1 == []int{0, 0, 2}

s2 := append(s1, 3, 5, 7)          // agrega múltiples elementos
                                   // s2 == []int{0, 0, 2, 3, 5, 7}

s3 := append(s2, s0...)            // agrega un sector
                                   // s3 == []int{0, 0, 2, 3, 5, 7, 0, 0}

s4 := append(s3[3:6], s3[2:]...)   // agrega superponiendo un sector
                                   // s4 == []int{3, 5, 7, 2, 3, 5, 7, 0, 0}

var t []interface{}

t = append(t, 42, 3.1415, "foo")   // t == []interface{}{42, 3.1415, "foo"}

var b []byte

b = append(b, "bar"...)            // añade contenido a la cadena
                                   // b == []byte{'b', 'a', 'r' }
```

La función `copy` copia elementos del sector desde una fuente `src` a un destino `dst` y devuelve el número de elementos copiados. Ambos argumentos deben tener idéntico tipo de elemento `T` y deben ser asignables a un sector de tipo `[]T`. El número de elementos

copiados es el mínimo de `len(src)` y `len(dst)`. Como caso especial, `copy` también acepta un argumento destino asignable al tipo `[]byte` con un argumento fuente de tipo cadena. Esta forma copia los bytes de la cadena al sector de byte.

```
copy(dst, src []T) int
copy(dst []byte, src string) int
```

Ejemplos:

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
var b = make([]byte, 5)
n1 := copy(s, a[0:])            // n1 == 6, s == []int{0, 1, 2, 3, 4, 5}
n2 := copy(s, s[2:])            // n2 == 4, s == []int{2, 3, 4, 5, 4, 5}
n3 := copy(b, "¡Hola, mundo!")  // n3 == 5, b == []byte("¡Hola")
```

Eliminando elementos de mapa

La función incorporada `delete` elimina el elemento con la clave `k` de un mapa `m`. El tipo de `k` debe ser asignable al tipo de la clave de `m`.

```
delete(m, k)  // elimina el elemento m[k] del mapa m
```

Si el mapa `m` es `nil` o el elemento `m[k]` no existe, `delete` no es operativo.

Manipulando números complejos

Tres funciones para montar y desmontar números complejos. La función incorporada `complex` construye un valor complejo a partir de una parte real de coma flotante y una parte imaginaria, mientras que `real` e `imag` extraen las partes real e imaginaria de un valor complejo.

```
complex(parteReal, parteImaginaria floatT) tipoComplejo
real(tipoComplejo) floatT
imag(tipoComplejo) floatT
```

El tipo de los argumentos y valor de retorno corresponde. Para `complex`, los dos argumentos deben ser del mismo tipo, coma flotante y el tipo de retorno es de tipo complejo con los correspondientes componentes de coma flotante: `complex64` para `float32`, `complex128` para `float64`. Juntas las funciones `real` e `imag` forman la inversa, por lo que para un complejo de valor de `z`, `z == complex(real(z), imag(z))`.

Si todos los operandos de estas funciones son constantes, el valor de retorno es una constante.

```
var a = complex(2, -2)             // complex128
var b = complex(1.0, -1.4)         // complex128
x := float32(math.Cos(math.Pi/2))  // float32
var c64 = complex(5, -x)           // complex64
var im = imag(b)                   // float64
var rl = real(c64)                 // float32
```

Manejando pánicos

Dos funciones incorporadas, `panic` y `recover`, ayudan en la presentación de informes y manipulación de pánicos en tiempo de ejecución y las condiciones de error definidas por el programa.

```
func panic(interface{})

func recover() interface{}
```

Durante la ejecución de una función `F`, una llamada explícita a `panic` o un pánico en tiempo de ejecución finaliza la ejecución de `F`. Cualquier función diferida por `F` se ejecuta entonces de la manera usual. A continuación, las funciones diferidas a cargo del llamador `F&#39;s` se ejecutan y así sucesivamente hasta cualquier diferida por la función de nivel superior en la rutinago en ejecución. En ese momento, el programa se termina y se reporta la condición de error, incluyendo el valor del argumento para entrar en `pánico`. Esta secuencia de terminación se conoce como *entrar en pánico* (`panicking` en inglés).

```
panic(42)
panic("inalcanzable")
panic(Error("no se puede interpretar"))
```

La función `recover` permite que un programa maneje el comportamiento de una rutinago entrando en pánico. Supongamos que una función `G` difiere una función `D` que llama a `recover` y se produce pánico en una función en la misma rutinago en la que se está ejecutando `G`. Cuando el funcionamiento natural de las funciones diferidas alcanza a `D`, el valor de retorno de `D` en la llamada a `recover` será el valor que se pase a la llamada de `panic`. Si `D` regresa normalmente, sin necesidad de iniciar un nuevo `pánico`, la secuencia de pánico se detiene. En ese caso, el estado de las funciones llamadas entre `G` y la llamada a `pánico` se descartan y se reanuda la ejecución normal. Cualquier función diferida por `G` antes de `D` entonces se ejecutan y la ejecución de `G` termina volviendo a su llamador.

El valor de retorno de `recover` es `nil` si se cumple alguna de las siguientes condiciones:

- el argumento para `panic` era `nil` ;
- la rutinago no es presa del pánico;
- `recover` no fue llamado directamente por una función diferida.

La función `protege` en el siguiente ejemplo invoca al argumento de la función `g` y protege a los llamadores de pánico en tiempo de ejecución planteados por `g` .

```
func protege(g func()) {
    defer func() {
        log.Println("hecho")  // Println ejecuta normalmente incluso
                              // si hay un pánico
        if x := recover(); x != nil {
            log.Printf("pánico en tiempo de ejecución: %v", x)
        }
    }()
    log.Println("inicio")
    g()
}
```

Proceso de arranque

Las implementaciones actuales proporcionan varias funciones incorporadas útiles durante el proceso de arranque (`bootstrapping` en inglés). Estas funciones están completamente documentadas pero no se garantiza que permanezcan en el lenguaje. No devuelven un resultado.

```
Función    Comportamiento
print      imprime todos los argumentos; el formato de los argumentos es
           específico de la implementación
println    como print pero imprime espacios entre argumentos y una nueva
           línea al final
```

Paquetes

Los programas Go se construyen enlazando *paquetes*. Un paquete a su vez, se construye a partir de uno o más archivos fuente que juntos declaran constantes, tipos, variables y funciones propias del paquete y que son accesibles en todos los archivos del mismo paquete. Estos elementos se pueden exportar y utilizar en otros paquetes.

Organización de archivos fuente

Cada archivo fuente consta de una cláusula `package` que define el paquete al que pertenece, seguida de una serie posiblemente vacía de declaraciones de importación que declaran los paquetes que desea utilizar, seguida de una serie posiblemente vacía de declaraciones de constantes, tipos, variables y funciones.

```
ArchivoFuente    = CláusulaPackage ";" { InstruImport ";" } { DeclaNivelSuperior ";" } .
```

Cláusula `package`

Una cláusula `package` comienza cada archivo fuente y define el paquete al que pertenece el archivo.

```
CláusulaPackage  = "package" NombrePaquete .
NombrePaquete    = identificador .
```

El NombrePaquete no debe ser el identificador blanco.

```
package math
```

Un conjunto de archivos que comparten el mismo NombrePaquete forman la implementación de un paquete. Una implementación puede requerir que todos los archivos fuente para un paquete habiten en el mismo directorio.

Instrucciones `import`

Una instrucción `import` indica que el archivo fuente conteniendo la declaración depende de la funcionalidad de la importación del paquete *importado* (inicio y ejecución del programa) y permite el acceso a los identificadores exportados de ese paquete. La `import` nombra un identificador (NombrePaquete) que se utilizará para acceder y una RutaImportación que especifica el paquete a ser importado.

```
InstruImport       = "import" ( EspecImport | "(" { EspecImport ";" } ")" ) .
EspecImport        = [ "." | NombrePaquete ] RutaImportación .
RutaImportación    = cadena_lit .
```

El NombrePaquete se utiliza en identificadores cualificados para acceder a los identificadores exportados del paquete dentro del archivo fuente importado. Se declara en el bloque de archivo. Si se omite el NombrePaquete, toma como predeterminado el identificador especificado en la cláusula `package` del paquete importado. Si aparece un

punto explícito (`.`) en lugar de un nombre, todos los identificadores exportados del paquete declarados en ese bloque de paquete serán declarados en el bloque de archivo del archivo fuente importador y se debe acceder a ellos sin un calificador.

La interpretación de la RutaImportación depende de la implementación, pero típicamente es una subcadena del nombre de archivo completo del paquete compilado y puede ser relativa a un repositorio de paquetes instalados.

> Restricción de implementación: Un compilador puede restringir las RutaImportación a cadenas no vacías utilizando sólo caracteres pertenecientes a Unicode categorías generales L, M, N, P y S (los caracteres gráficos sin espacios) y también puede excluir los caracteres `!&quot;#$%&amp;&#39;()*,:;&lt;=&gt;?[\]^` {|}` y el carácter de reemplazo Unicode U+FFFD.

Supongamos que hemos compilado un paquete que contiene la cláusula de paquete `package math`, que exporta la función `Sin` y el paquete compilado está instalado en el archivo identificado por `&quot;lib/math&quot;`. Esta tabla demuestra cómo se accede a `Sin` en los archivos que importan el paquete después de los distintos tipos de declaración de importación.

```
Declaración de importación    Nombre local de Sin
import   "lib/math"           math.Sin
import m "lib/math"           m.Sin
import . "lib/math"           Sin
```

Una declaración de importación declara una relación de dependencia entre el importador y el paquete importado. Es ilegal que un paquete directa o indirectamente se importe a sí mismo o importar directamente un paquete sin hacer referencia a cualquiera de sus identificadores exportados. Para importar un paquete únicamente por sus efectos secundarios (iniciación), utiliza el identificador blanco como nombre de paquete explícito:

```
import _ "lib/math"
```

Un paquete de ejemplo

Aquí tienes un paquete Go completo que implementa un tamiz primo concurrente.

```
package main

import "fmt"

// Envía la secuencia 2, 3, 4, … al canal 'ch'.
func genera(ch chan<- int) {
    for i := 2; ; i++ {
        ch <- i  // Envía 'i' al canal 'ch'.
    }
}

// Copia los valores del canal «src» al canal 'dst',
// quitando los divisibles por 'primo'.
func filtra(src <-chan int, dst chan<- int, primo int) {
    for i: = range src { // Bucle sobre los valores recibidos de «src».
        if i%primo != 0 {
            dst <- i  // Envía 'i' al canal 'dst'.
        }
    }
}

// El tamiz primo: filtra la conexión en cadena y los procesa juntos.
func tamiz() {
    ch := make(chan int)  // Crea un nuevo canal.

    go genera(ch)       // Inicia genera() como un subproceso.
    for {
        primo := <-ch
        fmt.Print(primo, "\n")
        ch1 := make(chan int)
        go filtra(ch, ch1, primo)
        ch = ch1
    }
}

func main() {
    tamiz()
}
```

Iniciación y ejecución de programa

El valor cero

Cuando se asigna almacenamiento a una variable, ya sea a través de una declaración o una llamada a `new` o cuando se crea un nuevo valor, bien a través de un literal compuesto o por medio de una llamada a `make` y no se proporciona iniciación explícita, se da a la variable o valor un valor predeterminado. Cada elemento de una variable o valor tal se

establece al *valor cero* para su tipo: `false` para lógicos, `0` para números enteros, `0.0` para número de coma flotante, `&quot;&quot;` para cadenas y `nil` para punteros, funciones, interfaces, sectores, canales y mapas. Esta iniciación se realiza de forma recursiva, por lo que, por ejemplo, cada elemento de un arreglo de estructuras tendrá sus campos a cero si no se especifica ningún valor.

Estas dos simples declaraciones son equivalentes:

```
var i int
var i int = 0
```

Después de:

```
type T struct { i int; f float64; siguiente *T }

t := new(T)
```

se cumple lo siguiente:

```
t.i == 0
t.f == 0.0
t.siguiente == nil
```

Lo mismo sería cierto después de:

```
var t T
```

Iniciación del paquete

Dentro de un paquete, las variables a nivel de paquete se inician en *orden de declaración* pero después de cualquiera de las variables de que *dependen*.

Específicamente, una variable a nivel de paquete se considera *lista para iniciación* si aún no se ha iniciado y, o bien no tiene ninguna expresión de iniciación o su expresión de iniciación no tiene ninguna dependencia de variables sin iniciar. La iniciación procede por iniciar repetidamente la siguiente variable a nivel de paquete que se declaró más temprano y está lista para iniciación, hasta que no haya más variables listas para iniciación.

Si algunas variables están todavía sin iniciar cuando termine este proceso, esas variables son parte de uno o más ciclos de iniciación y el programa no es válido.

El orden de la declaración de variables declaradas en varios archivos es determinado por el orden en que los archivos se presentan al compilador: Las variables declaradas en el primer archivo se declaran antes de cualquiera de las variables declaradas en el segundo archivo y así sucesivamente.

El análisis de dependencia no se basa en los valores reales de las variables, sólo en *referencias* léxicas a ellos en la fuente, analizados transitivamente. Por ejemplo, si la expresión de iniciación para una variable `x` se refiere a una función cuyo cuerpo se refiere a la variable `y` entonces `x` depende de `y`. Específicamente:

- Una referencia a una variable o función es un identificador que denota esa variable o función.
- Una referencia a un método `m` es un valor método o expresión método de la forma `t.m`, donde el tipo (estático) de `t` no es un tipo interfaz y el método `m` está en el conjunto de métodos de `t`. Es irrelevante que el valor resultante de la función `t.m` sea invocado.
- Una variable, función o método `x` depende de una variable `y` si la expresión de iniciación de `x` o el cuerpo (para funciones y métodos) contiene una referencia a `y` o a una función o método que dependa de `y`.

El análisis de dependencias se realiza por paquete; únicamente se consideran referencias a variables, funciones y métodos declarados en el paquete actual.

Por ejemplo, dadas las declaraciones:

```
var (
    a = c + b
    b = f()
    c = f()
    d = 3
)

func f() int {
    d++
    return d
}
```

el orden de iniciación es `d`, `b`, `c`, `a`.

Las variables también se pueden iniciar utilizando funciones con nombre `init` declaradas en el bloque del paquete, sin argumentos y sin parámetros de resultado.

```
func init() { … }
```

Se pueden definir múltiples funciones como esta, incluso dentro de un solo archivo fuente. El identificador `init` no está declarado y por lo tanto las funciones `init` no se pueden referir desde cualquier parte de un programa.

Un paquete sin importaciones se inicia mediante la asignación de valores iniciales a todas sus variables a nivel de paquete seguida por una llamada a todas las funciones `init` en el orden en que aparecen en la fuente, posiblemente en varios archivos, tal como se presentan al compilador. Si un paquete tiene importaciones, los paquetes importados se inician antes de iniciar el propio paquete. Si varios paquetes importan un paquete, el paquete importado se iniciará únicamente una vez. La importación de paquetes, por construcción, garantiza que no puede haber ninguna dependencia de iniciación cíclica.

La iniciación de variables del paquete y la invocación a las funciones `init` ocurre secuencialmente en una sola rutinago, un paquete a la vez. Una función `init` puede lanzar otras rutinagos, mismas que se pueden ejecutar simultáneamente con el código de iniciación. Sin embargo, la iniciación siempre secuencia las funciones `init`: no invocará a la siguiente hasta que la anterior haya regresado.

Para garantizar un comportamiento de iniciación reproducible, se anima a construir sistemas para presentar varios archivos pertenecientes a un mismo paquete con el fin de nombrar archivo léxicos a un compilador.

Ejecución del programa

Un programa completo se crea enlazando un solo paquete no importado llamado el *package main* con todos los paquetes que importa, transitivamente. El paquete principal debe tener el nombre de paquete `main` y declarar una función `main` que no tiene argumentos y no devuelve nada.

```
func main() { … }
```

La ejecución del programa comienza iniciando el paquete principal y luego invocando a la función `main`. Cuando esa invocación a función regresa, el programa se cierra. No espera a que otras rutinasgo (no `main`) terminen.

Errores

El tipo predeclarado `error` se define como:

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

Es la interfaz convencional para representar una condición de error, el valor `nil` representa la ausencia de errores. Por ejemplo, una función para leer datos de un archivo se puede definir así:

```
func Lee(f *File, b []byte) (n int, err error)
```

Pánicos en tiempo de ejecución

Errores de ejecución como el intento de indexar un arreglo fuera de límites desencadena un pánico en tiempo de ejecución equivalente a llamar a la función incorporada `panic` con un valor de tipo interfaz `runtime.Error` definido en la implementación. Ese tipo se ajusta al tipo interfaz `error` predeclarado. Los valores de error exactos que representan condiciones de error en distintos tiempo de ejecución no están especificados.

```
package runtime

type Error interface {
    error
    // y tal vez otros métodos
}
```

Consideraciones del sistema

Paquete `unsafe`

El paquete integrado `unsafe`, conocido por el compilador, ofrece facilidades para la programación de bajo nivel incluyendo las operaciones que violan el sistema de tipos. Un paquete usando `unsafe` **debe** revisar manualmente la seguridad de tipos y puede no ser portátil. El paquete proporciona la siguiente interfaz:

```
package unsafe

type ArbitraryType int  // abreviatura de un tipo Go arbitrario; no es
                        // un tipo real

type Pointer *ArbitraryType

func Alignof(variable ArbitraryType) uintptr
func Offsetof(selector ArbitraryType) uintptr
func Sizeof(variable ArbitraryType) uintptr
```

Un `Pointer` es un tipo puntero pero un valor `Pointer` no se puede desreferenciar. Cualquier puntero o valor del tipo subyacente `uintptr` se puede convertir a un tipo `Pointer` y viceversa. El efecto de convertir entre `Pointer` y `uintptr` está definido por la implementación.

```
var f float64

bits = *(*uint64)(unsafe.Pointer(&f))
type ptr unsafe.Pointer

bits = *(*uint64)(ptr(&f))
var p ptr = nil
```

Las funciones `Alignof` y `Sizeof` toman una expresión `x` de cualquier tipo y devuelven la alineación o tamaño, respectivamente, de una hipotética variable `v` como si `v` hubiese sido declarada a través de `var v = x`.

La función `Offsetof` toma un (posiblemente entre paréntesis) selector `s.f`, que denota un campo `s` de la estructura denotada por `s` o `*s` y devuelve el desplazamiento del campo en bytes relativo a la estructura `dirección`. Si `f` es un campo incrustado, este debe ser accesible sin indirecciones de puntero a través de los campos de la estructura. Para una estructura `s` con el campo `f`:

```
uintptr(unsafe.Pointer(&s)) + unsafe.Offsetof(s.f) == uintptr(unsafe.Pointer(&s.f))
```

La arquitectura de las computadoras puede requerir que las direcciones de memoria estén *alineadas*; es decir, que las direcciones de una variable sean un múltiplo de un factor, el tipo *alineación* de la variable. La función `Alignof` toma una expresión que denota una variable de cualquier tipo y devuelve la alineación (del tipo) de la variable en bytes. Para una variable `x`:

```
uintptr(unsafe.Pointer(&x)) % unsafe.Alignof(x) == 0
```

Las llamadas a `Alignof`, `Offsetof` y `Sizeof` son expresiones constantes de tiempo de compilación de tipo `uintptr`.

Garantías de tamaño y alineación

Para los tipos numéricos, los siguientes tamaños están garantizados:

```
tipo                                tamaño en bytes
byte, uint8, int8                    1
uint16, int16                        2
uint32, int32, float32               4
uint64, int64, float64, complex64    8
complex128                          16
```

Las siguientes propiedades mínimas de alineación están garantizadas:

1. Para una variable `x` de cualquier tipo: `unsafe.Alignof(x)` es al menos 1.
2. Para una variable `x` de tipo estructura: `unsafe.Alignof(x)` es el más grande de todos los valores `unsafe.Alignof(x.f)` para cada campo `f` de `x`, pero al menos 1.
3. Para una variable `x` de tipo arreglo: `unsafe.Alignof(x)` es el mismo que `unsafe.Alignof(x[0])`, pero por lo menos 1.

Una estructura o tipo arreglo tiene tamaño cero si no contiene campos (o elementos, respectivamente) que tengan un tamaño mayor que cero. Dos distintas variables de tamaño cero pueden tener la misma dirección en memoria.

AVISO LEGAL:

Esta es una traducción **NO OFICIAL** de la licencia, está aquí para que los lectores hispanohablantes se den una idea de su contenido, **LA ÚNICA LICENCIA OFICIAL PARA FINES LEGALES ES LA ORIGINAL EN INGLÉS QUE APARECE [Abajo][Lic]**.

Licencia

La redistribución y uso en formas fuente y binaria, con o sin modificaciones, están permitidas siempre y cuendo se cumplan las siguientes condiciones:

- Las redistribuciones del código fuente deben conservar el copyright anterior notar, esta lista de condiciones y el siguiente descargo de responsabilidad.
- Las redistribuciones en formato binario deben reproducir el anterior aviso de copyright, esta lista de condiciones y el siguiente descargo de responsabilidad en la documentación y/u otros materiales proporcionados con la distribución.
- Ni el nombre de Google Inc. ni los nombres de sus colaboradores se pueden usar para respaldar o promocionar productos derivados de este software sin permiso previo por escrito.

ESTE SOFTWARE LO PROPORCIONAN LOS PROPIETARIOS DEL COPYRIGHT Y COLABORADORES "TAL CUAL" Y CUALQUIER GARANTÍA EXPRESA O IMPLÍCITA, INCLUYENDO, PERO NO LIMITADO A, LAS GARANTÍAS DE COMERCIALIZACIÓN Y APTITUD PARA UN PROPÓSITO PARTICULAR. EN NINGÚN CASO EL AUTOR PROPIETARIO O SUS COLABORADORES SERÁN RESPONSABLES POR DAÑOS DIRECTOS, INDIRECTOS, ESPECIALES, EJEMPLARES O EMERGENTES (INCLUYENDO, PERO NO LIMITADO A LA OBTENCIÓN DE BIENES Y SERVICIOS; PÉRDIDA DE USO, DATOS O BENEFICIOS; O INTERRUPCIÓN COMERCIAL) CAUSADOS COMO EN CUALQUIER TEORÍA DE RESPONSABILIDAD, YA SEA POR CONTRATO, RESPONSABILIDAD OBJETIVA O AGRAVIO (INCLUYENDO NEGLIGENCIA) POR DAÑOS OCASIONADOS POR EL USO DE ESTE SOFTWARE, INCLUSO SI SE HA ADVERTIDO DE LA POSIBILIDAD DE TALES DAÑOS.

Licencia oficial

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Google Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Glosario

rutinasgo

`goroutines` en inglés. Es una función ejecutándose al mismo tiempo que otras `rutinasgo` en el mismo espacio.

4. Go eficiente 1. Documentación 5. Especificación del lenguaje

semiware

`middleware` en inglés. Es un software que asiste a una aplicación para comunicarse con otras aplicaciones, redes, dispositivos o con el sistema operativo.

URL

Son las siglas de `uniform resource locator` en inglés, o localizador uniforme de recursos.

3. Cómo escribir código Go 4. Go eficiente

vector

`array` en inglés. En Go, un `arreglo`, `matriz` o `vector` es una zona de almacenamiento continuo, que contiene una serie de elementos del mismo tipo, los elementos del arreglo.

4. Go eficiente 5. Especificación del lenguaje