Capítulo 5

Funciones

Una función nos permite envolver una secuencia de sentencias como una unidad que se puede llamar desde otra parte en un programa, tal vez varias veces. Las funciones permiten dividir un trabajo grande en trozos más pequeños que bien podrían ser escritos por diferentes personas separadas por el tiempo y el espacio. Una función oculta sus detalles de implementación a sus usuarios. Por todas estas razones, las funciones son una parte crítica de cualquier lenguaje de programación.

Ya hemos visto varias funciones. Ahora vamos a tomar tiempo para tratarlas más a fondo. El ejemplo de ejecución de este capítulo es un rastreador web, es decir, el componente de un motor de búsqueda web responsable de ir a buscar páginas web, el descubrimiento de los vínculos dentro de ellas, ir a buscar las páginas identificadas por esos enlaces, etc. Un rastreador web nos da una gran oportunidad para explorar la recursividad, las funciones anónimas, el manejo de errores y aspectos de las funciones que son exclusivas de **Go**.

5.1. Declaraciones de funciones

Una declaración de función tiene un nombre, una lista de parámetros, una lista opcional de resultados, y un cuerpo:

```
func name(parameter-list) (result-list) {
    body
}
```

La lista de parámetros especifica los nombres y tipos de los parámetros de la función, que son variables locales cuyos valores o argumentos son suministrados por el que llama. La lista de resultados especifica los tipos de los valores que devuelve la función. Si la función devuelve un resultado no identificado o ningún resultado en absoluto, los paréntesis son opcionales y por lo general se omiten. Dejando fuera la lista de resultados declara enteramente una función que no devuelve ningún valor y se llama sólo por sus efectos. En la función **hypot**,

```
func hypot(x, y float64) float64 {
    return math.Sqrt(x*x + y*y)
}

fmt.Println(hypot(3, 4)) // "5"
```

x e **y** son parámetros en la declaración, **3** y **4** son los argumentos de la llamada y la función devuelve un valor **float64**.

Igual que los parámetros, los resultados pueden tener nombre. En ese caso, cada nombre declara una variable local inicializada con el valor cero para su tipo.

Una función que tiene una lista de resultados debe terminar con una sentencia **return**, a menos que la ejecución claramente no pueda llegar al final de la función, tal vez porque la función termina con una llamada a **panic** o un bucle infinito **for** sin **break**.

Como hemos visto con **hypot**, una secuencia de parámetros o resultados del mismo tipo se pueden factorizar de modo que el tipo en sí se escriba solamente una vez. Estas dos declaraciones son equivalentes:

```
func f(i, j, k int, s, t string) { /* ... */ }
func f(i int, j int, k int, s string, t string) { /* ... */ }
```

Aquí hay cuatro formas de declarar una función con dos parámetros y un resultado, todos de tipo **int**. Puede ser utilizado el identificador en blanco para enfatizar que un parámetro no se utiliza.

```
func add(x int, y int) int { return x + y }
func sub(x, y int) (z int) { z = x - y; return }
func first(x int, _ int) int { return x }
func zero(int, int) int { return 0 }

fmt.Printf("%T\n", add) // "func(int, int) int"
fmt.Printf("%T\n", sub) // "func(int, int) int"
fmt.Printf("%T\n", first) // "func(int, int) int"
fmt.Printf("%T\n", zero) // "func(int, int) int"
```

El tipo de una función se llama a veces su firma (**signature**). Dos funciones tienen el mismo tipo o firma si tienen la misma secuencia de tipos de parámetros y la misma secuencia de tipos de resultados. Los nombres de los parámetros y resultados no afectan el tipo, ni tampoco si han sido o no declarados usando la forma factorizada.

Cada llamada a función debe proporcionar un argumento para cada parámetro, en el orden en que fueron declarados los parámetros. **Go** no tiene un concepto de valores de parámetros por defecto, ni ninguna forma de especificar argumentos por nombre, por lo que los nombres de los parámetros y resultados no importan al que llama, excepto como documentación.

Los parámetros son variables locales dentro del cuerpo de la función, con sus valores iniciales establecidos en los argumentos proporcionados por el que llama. Los parámetros de función y resultados mencionados son variables en el mismo bloque léxico de función como variables locales.

Los argumentos se pasan por valor, por lo que la función recibe una copia de cada argumento; las modificaciones a la copia no afectan al que llama. Sin embargo, si el argumento contiene algún tipo de referencia, como un puntero, **slice**, mapa, una función o un canal, entonces el que llama puede verse afectado por las modificaciones que haga la función de las variables de forma indirecta a las que se refiere el argumento.

Se puede encontrar de vez en cuando una declaración de función sin un cuerpo, lo que indica que la función está implementada en un lenguaje que no es **Go**. Tal declaración define la firma de la función.

```
package math
func Sin(x float64) float64 // implementado en lenguaje ensamblador
```

5.2. Recursividad

Las funciones pueden ser recursivas, es decir, pueden llamarse a sí mismas, ya sea directa o indirectamente. La recursividad es una técnica poderosa para muchos problemas, y por supuesto es esencial para el procesamiento de estructuras de datos recursivas. En la **Sección 4.4**, se utilizó la recursividad sobre un árbol para implementaba un mecanismo simple de ordenación. En esta sección, lo utilizaremos de nuevo para el procesamiento de documentos **HTML**.

El programa de ejemplo siguiente utiliza un paquete no estándar, **golang.org/x/net/html**, que proporciona un analizador de **HTML**. Los repositorios **golang.org/x/...** tienen paquetes diseñados y mantenidos por el equipo de **Go** para aplicaciones tales como la creación de redes, procesamiento de texto internacionalizado, plataformas móviles, manipulación de imágenes, criptografía y herramientas para desarrolladores. Estos paquetes no están en la librería estándar, ya que están todavía en desarrollo o porque rara vez los necesitan la mayoría de los programadores de **Go**.

Las partes de la **API golang.org/x/net/html** que necesitamos se muestran a continuación. La función **html.Parse** lee una secuencia de bytes, los analiza y devuelve la raíz de la estructura del documento **HTML**, que es un **html.Node**. **HTML** tiene varios tipos de nodos, texto, comentarios, etc., pero aquí nos interesa solamente los elementos de los nodos de la forma **<name key='value'>**.

```
golang.org/x/net/html
package html

type Node struct {
    Type NodeType
    Data string
    Attr []Attribute
    FirstChild, NextSibling *Node
}

type NodeType int32

const (
    ErrorNode NodeType = iota
    TextNode
    DocumentNode
    ElementNode
    CommentNode
    DoctypeNode
)

type Attribute struct {
    Key, Val string
}
func Parse(r io.Reader) (*Node, error)
```

La función **main** analiza la entrada estándar como **HTML**, extrae los enlaces usando una función recursiva **visit**, e imprime cada enlace descubierto:

```
gopl.io/ch5/findlinks1
// Findlinks1 imprime los enlaces en un documento HTML lee de la entrada estándar.
package main

import (
    "fmt"
    "os"
    "golang.org/x/net/html"
)

func main() {
    doc, err := html.Parse(os.Stdin)
    if err != nil {
```

```
            fmt.Fprintf(os.Stderr, "findlinks1: %v\n", err)
            os.Exit(1)
        }
        for _, link := range visit(nil, doc) {
            fmt.Println(link)
        }
    }
```

La función **visit** atraviesa un árbol de nodos **HTML**, extrae el enlace desde el atributo **href** de cada elemento de anclaje **<a href='...'>**, añade los enlaces a un **slice** de **strings**, y devuelve el **slice** resultante:

```
    // visit añade a los enlaces cada enlace que se encuentre en n y devuelve el resultado.
    func visit(links []string, n *html.Node) []string {
        if n.Type == html.ElementNode && n.Data == "a" {
            for _, a := range n.Attr {
                if a.Key == "href" {
                    links = append(links, a.Val)
                }
            }
        }
        for c := n.FirstChild; c != nil; c = c.NextSibling {
            links = visit(links, c)
        }
        return links
    }
```

Para descender por el árbol por un nodo **n**, **visit** se llama a sí misma recursivamente para cada uno de los hijos de **n**, que están en la lista enlazada **FirstChild.**

Vamos a ejecutar **findlinks** en la página principal **Go**, canalizando el resultado de **fetch** (**§1.5**) a la entrada de **findlinks**. Hemos editado la salida ligeramente por razones de brevedad.

```
    $ go build gopl.io/ch1/fetch
    $ go build gopl.io/ch5/findlinks1
    $ ./fetch https://golang.org | ./findlinks1
    #
    /doc/
    /pkg/
    /help/
    /blog/
    http://play.golang.org/
    //tour.golang.org/
    https://golang.org/dl/
    //blog.golang.org/
    /LICENSE
    /doc/tos.html
    http://www.google.com/intl/en/policies/privacy/
```

Nótese la variedad de formas de los enlaces que aparecen en la página. Más tarde veremos cómo resolverlos con relación a la base **URL**, **https://golang.org**, para hacer **URLs** absolutas.

El siguiente programa utiliza la recursividad sobre el árbol de nodos **HTML** para imprimir la estructura del árbol en un esquema. A medida que encuentra cada elemento, empuja la etiqueta del elemento en una pila, y a continuación, imprime la pila.

```
    gopl.io/ch5/outline
    func main() {
        doc, err := html.Parse(os.Stdin)
        if err != nil {
            fmt.Fprintf(os.Stderr, "outline: %v\n", err)
            os.Exit(1)
        }
        outline(nil, doc)
    }

    func outline(stack []string, n *html.Node) {
        if n.Type == html.ElementNode {
            stack = append(stack, n.Data) // push tag
            fmt.Println(stack)
        }
        for c := n.FirstChild; c != nil; c = c.NextSibling {
            outline(stack, c)
        }
    }
```

Observar una sutileza: a pesar de que **outline** "empuja" un elemento en la pila, no hay **pop** correspondiente. Cuando **outline** se llama a sí mismo de forma recursiva, el destinatario recibe una copia de la pila (**stack**). A pesar de que el

destinatario de la llamada puede añadir elementos a este **slice**, modificando su arreglo subyacente y tal vez incluso asignar un nuevo arreglo, no modifica los elementos iniciales que son visibles para el que llama, así que cuando retorna la función, la pila del que llama es igual que era antes de la llamada.

Aquí está **outline** de **https://golang.org**, editado de nuevo por razones de brevedad:

```
$ go build gopl.io/ch5/outline
$ ./fetch https://golang.org | ./outline
[html]
[html head]
[html head meta]
[html head title]
[html head link]
[html body]
[html body div]
[html body div]
[html body div div]
[html body div div form]
[html body div div form div]
[html body div div form div a]
...
```

Como se puede ver mediante la experimentación con **outline**, la mayoría de los documentos **HTML** pueden ser procesados con sólo unos pocos niveles de recursividad, pero no es difícil de construir páginas web patológicas que requieren una recursión extremadamente profunda.

Muchas implementaciones de lenguajes de programación utilizan una pila de llamadas de función de tamaño fijo; tamaños de **64 KB** a **2 MB** son típicos. Las pilas de tamaño fijo imponen un límite en la profundidad de la recursividad, por lo que hay que tener cuidado para evitar un desbordamiento de pila cuando se atraviesan estructuras de datos de gran tamaño de forma recursiva; pilas de tamaño fijo pueden incluso suponer un riesgo para la seguridad. Por el contrario, las implementaciones típicas de **Go** utilizan pilas de tamaño variable que empiezan poco a poco y crecen como sea necesario, hasta un límite del orden de un gigabyte. Esto nos permite utilizar la recursividad de forma segura y sin preocuparnos del desbordamiento.

5.3. Múltiples valores devueltos

Una función puede devolver más de un resultado. Hemos visto muchos ejemplos de funciones de los paquetes estándar que devuelven dos valores, el resultado del cálculo deseado y un valor de error o booleano que indica si el cálculo funcionó. El siguiente ejemplo muestra cómo escribir uno para nuestro uso.

El programa siguiente es una variación de **findlinks** que hace la petición **HTTP** de manera que ya no es necesario ejecutar **fetch**. Debido a que las operaciones **HTTP** y de análisis pueden fallar, **findLinks** declara dos resultados: la lista de enlaces descubiertos y un error. Por cierto, el analizador **HTML** normalmente puede recuperarse de una mala entrada y construir un documento que contenga nodos de error, por lo que **Parse** raramente falla; cuando lo hace, es por lo general debido a que subyacen errores de **E/S**.

```
gopl.io/ch5/findlinks2
func main() {
    for _, url := range os.Args[1:] {
        links, err := findLinks(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "findlinks2: %v\n", err)
            continue
        }
        for _, link := range links {
            fmt.Println(link)
        }
    }
}


// findLinks realiza una solicitud HTTP GET para la url, analiza el
// la respuesta HTML, y extrae y devuelve los enlaces.
func findLinks(url string) ([]string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("getting %s: %s", url, resp.Status)
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
```

```
            return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
        }
        return visit(nil, doc), nil
    }
```

Hay cuatro sentencias **return** en **findLinks** , cada uno de los cuales devuelve un par de valores. Las tres primeras sentencias **return** causa que la función pase los errores subyacentes de los paquetes **http** y **html** al que llama. En el primer caso, el error se devuelve sin cambios; en el segundo y tercer, se aumenta con información de contexto adicional por **fmt.Errorf** (**§7.8**). Si **findLinks** tiene éxito, la sentencia final **return** devuelve el **slice** de enlaces, sin error.

Debemos asegurarnos de que **resp.Body** está cerrado para que los recursos de red se liberen correctamente, incluso en caso de error. El recolector de basura de **Go** recicla la memoria no utilizada, pero no asume que va a liberar los recursos del sistema operativo no utilizados, como archivos abiertos y conexiones de red. Ellos deben cerrarse de forma explícita.

El resultado de llamar a una función de múltiples valores es una tupla de valores. La llamada de una función de este tipo debe asignar explícitamente los valores a las variables si se van a utilizar cualquiera de ellas:

```
links, err := findLinks(url)
```

Para ignorar uno de los valores, se asigna al identificador en blanco:

```
links, _ := findLinks(url) // errores ignorados
```

El resultado de una llamada de varios valores puede ser devuelto el mismo en una llamada a función (valor múltiple), como en esta función que se comporta como **findLinks** pero registra su argumento:

```
func findLinksLog(url string) ([]string, error) {
    log.Printf("findLinks %s", url)
    return findLinks(url)
}
```

Una llamada de varios valores puede aparecer como un único argumento cuando se llama a una función de múltiples parámetros. Aunque rara vez se utiliza en la producción de código, esta función es conveniente algunas veces durante la depuración, ya que nos permite imprimir todos los resultados de una llamada mediante una sola sentencia. Las dos sentencias de impresión a continuación tienen el mismo efecto.

```
log.Println(findLinks(url))

links, err := findLinks(url)
log.Println(links, err)
```

Nombres bien elegidos pueden documentar la importancia de los resultados de una función. Los nombres son particularmente valiosos cuando una función devuelve varios resultados del mismo tipo, como

```
func Size(rect image.Rectangle) (width, height int)
func Split(path string) (dir, file string)
func HourMinSec(t time.Time) (hour, minute, second int)
```

pero no siempre es necesario nombrar multiples resultados, exclusivamente para la documentación. Por ejemplo, la convención dicta que un resultado final **bool** indica éxito; un resultado de error a menudo no necesita explicación.

En una función con resultados con nombre, los operandos de una instrucción de retorno pueden ser omitidos. Esto es llamado un retorno al descubierto (**bare return**).

```
// CountWordsAndImages hace una solicitud HTTP de una URL de documento HTML
// y devuelve el número de palabras e imágenes en el mismo.
func CountWordsAndImages(url string) (words, images int, err error) {
    resp, err := http.Get(url)
    if err != nil {
        return
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        err = fmt.Errorf("parsing HTML: %s", err)
        return
    }
    words, images = countWordsAndImages(doc)
    return
}
func countWordsAndImages(n *html.Node) (words, images int) { /* ... */ }
```

Un retorno al descubierto es un atajo para retornar cada una de las variables de resultado nombrados en orden, por lo que, en la función anterior, cada sentencia **return** es equivalente a

```
return words, images, err
```

En funciones como ésta, con muchas sentencias de retorno y varios resultados, los retornos al descubierto pueden reducir la duplicación de código, pero rara vez hacen el código más fácil de entender. Por ejemplo, no es evidente a primera vista que los dos primeros resultados son equivalentes a devolver **0, 0, err** (porque las variables de resultado

words e **images** se inicializan con sus valores cero) y que el último **return** es equivalente a **return words, images, nil**. Por esta razón, los retornos al descubierto es mejor utilizarlos con moderación.

5.4. Errores

Algunas funciones siempre tienen éxito en su tarea. Por ejemplo, **strings.Contains** y **strconv.FormatBool** tienen resultados bien definidos para todos los posibles valores de los argumentos y no pueden fallar, quitando los escenarios catastróficos e impredecibles como el mal funcionamiento de la memoria, en el que el síntoma está lejos de la causa y de la que hay poca esperanza de recuperación.

Otras funciones siempre tienen éxito, siempre y cuando se cumplan las condiciones previas. Por ejemplo, la función **time.Date** siempre construye un **time.Time** a partir de sus componentes, año, mes, etc., a menos que el último argumento (la zona horaria) sea **nil**, en cuyo caso se entra en pánico. Este pánico es un signo seguro de un error en el código de llamada y nunca debe suceder en un programa bien escrito.

Para muchas otras funciones, incluso en un programa bien escrito, el éxito no está asegurado, ya que depende de factores ajenos del control del programador. Cualquier función que haga **E/S**, por ejemplo, debe hacer frente a la posibilidad de error, y sólo un programador ingenuo puede creer que una simple lectura o escritura no puede fallar. De hecho, cuando las operaciones más fiables fallan inesperadamente es cuando más necesitamos saber por qué.

Los errores son por lo tanto una parte importante de la API de un paquete o un interfaz de una aplicación, y el fallo es sólo uno de varios comportamientos esperados. Este es el enfoque que lleva **Go** con la gestión de errores.

Una función en la que el fallo es un comportamiento esperado devuelve un resultado adicional, convencionalmente el último. Si el fallo tiene sólo una causa posible, el resultado es un booleano, por lo general llamado **ok**, como en este ejemplo de una búsqueda en la caché que siempre tiene éxito a menos que no hubiera ninguna entrada para esa clave:

```
value, ok := cache.Lookup(key)
if !ok {
    // ...cache[key] no existe...
}
```

Más a menudo, y especialmente para las **E/S**, el fallo puede tener una variedad de causas por las que el que llama va a necesitar una explicación. En tales casos, el tipo del resultado adicional es **error**.

El tipo incorporado **error** es un tipo de **interface**. Veremos más de lo que esto significa y sus implicaciones para tratamiento de errores en el **Capítulo 7**. Por ahora es suficiente saber que un error puede ser **nil** o **no nil**, **nil** implica éxito y **no nil** implica fracaso, y un error **no-nil** tiene un **string** de mensaje de error que se puede obtener llamando a su método **Error** o imprimirlo llamando a **fmt.Println(err)** o **fmt.Printf ("%v", err)**.

Por lo general, cuando una función devuelve un error que no es **nil**, sus otros resultados no están definidos y deben ser ignorados. Sin embargo, algunas funciones pueden devolver resultados parciales en los casos de error. Por ejemplo, si se produce un error durante la lectura de un archivo, una llamada a **Read** devuelve el número de bytes que se leyeron y un valor del error describe el problema. Para el correcto comportamiento, los que llaman pueden necesitar procesar los datos incompletos antes de manipular el error, por lo que es importante que tales funciones estén claramente documentadas con sus resultados.

El enfoque de **Go** lo diferencia de muchos otros lenguajes en los que se reportan fallos usando excepciones, no valores normales. Aunque **Go** tiene un mecanismo de excepción de ordenaciones, como veremos en la **Sección 5.9**, se utiliza sólo para informar de los errores realmente inesperadas que indican un fallo, no a los errores de rutina que se esperan de un programa construido sólidamente.

La razón de este diseño es que las excepciones tienden a enredar la descripción de un error con el flujo de control requerido para manejar la situación, a menudo conduce a un resultado no deseado: los errores de rutina se comunican al usuario final en forma de un seguimiento de la pila incomprensible, lleno de información sobre la estructura del programa, pero que carecen de contexto inteligible sobre lo que salió mal.

Por el contrario, los programas **Go** utilizan mecanismos de flujo de control común como **if** y **return** para responder a errores. Este estilo innegablemente exige que se preste más atención a la lógica de gestión de errores, pero eso es precisamente el punto.

5.4.1. Estrategias de manejo de Error

Cuando una llamada a una función devuelve un error, es responsabilidad del que llama para comprobarlo y actuar en consecuencia. Dependiendo de la situación, puede haber una serie de posibilidades. Vamos a echar un vistazo a cinco de ellas.

En primer lugar, y más común, es propagar el error, de modo que un fallo en una subrutina se convierte en un fallo de la rutina de llamada. Vimos ejemplos de esto en la función **findLinks** de la **Sección 5.3**. Si la llamada a **http.Get** falla, **findLinks** devuelve el error **HTTP** al que llama sin más preámbulos:

```
resp, err := http.Get(url)
if err != nil {
    return nil, err
}
```

Por el contrario, si la llamada a **html.Parse** falla, **findLinks** no devuelve el error analizador de **HTML** directamente, porque carece de dos piezas cruciales de información: que se produjo el error en el analizador, y la dirección **URL** del documento que se está analizando. En este caso, **findLinks** construye un nuevo mensaje de error que incluye ambas piezas de información, así como el análisis de error subyacente:

```
doc, err := html.Parse(resp.Body)
resp.Body.Close()
if err != nil {
    return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
}
```

La función **fmt.Errorf** formatea un mensaje de error utilizando **fmt.Sprintf** y devuelve un nuevo valor de error. La usamos para construir errores descriptivos anteponiendo sucesivamente la información de contexto adicional para el mensaje de error original. Cuando el error es en última instancia manejado por la función **main** del programa, se debe proporcionar una **string** causal clara desde la raíz del problema al fallo en su conjunto, que recuerda a una investigación de accidentes de la **NASA**:

```
genesis: crashed: no parachute: G-switch failed: bad relay orientation
```

Dado que los mensajes de error están encolados con frecuencia, los **strings** de mensajes no deben ser objeto de capitalización y deben ser evitados los saltos de línea. Los errores resultantes pueden ser largos, pero serán autónomos cuando se encuentran por herramientas como **grep**.

Cuando se diseñan mensajes de error, son deliberados, de modo que cada uno es una descripción significativa del problema con suficiente y pertinente detalle, y son coherentes, por lo que los errores devueltos por la misma función o por un grupo de funciones en el mismo paquete son similares en forma y pueden ser tratados de la misma manera.

Por ejemplo, el paquete **os** garantiza que cada error devuelto por una operación de archivo, tales como los de los métodos **os.Open** o **Read**, **Write**, o **Close** de un archivo abierto, no describen sólo la naturaleza del fallo (permiso denegado, dicho directorio no existe , etc.), sino que también el nombre del archivo, por lo que el que llama no necesita incluir esta información en el mensaje de error que se construye.

En general, la llamada **f(x)** es responsable de informar de la operación intentada **f** y el valor del argumento **x** en su relación con el contexto del error. El que llama es responsable de añadir más información, pero la llamada **f(x)** no, por ejemplo, la dirección **URL** en la llamada a **html.Parse** anteriormente.

Vamos a pasar a la segunda estrategia para el manejo de errores. Para los errores que representan problemas transitorios o impredecibles, puede tener sentido volver a intentar la operación fallida, posiblemente con un retardo entre intentos, y tal vez con un límite en el número de intentos o el tiempo dedicado a tratar antes de renunciar por completo.

```
gopl.io/ch5/wait
// WaitForServer intenta contactar con el servidor de una URL.
// trata un minuto utilizando retroceso exponencial.
// informa de un error si todos los intentos fallan.
func WaitForServer(url string) error {
    const timeout = 1 * time.Minute
    deadline := time.Now().Add(timeout)
    for tries := 0; time.Now().Before(deadline); tries++ {
        _, err := http.Head(url)
        if err == nil {
            return nil // success
        }
        log.Printf("server not responding (%s); retrying...", err)
        time.Sleep(time.Second << uint(tries)) // retroceso exponencial
    }
    return fmt.Errorf("server %s failed to respond after %s", url, timeout)
}
```

En tercer lugar, si el progreso es imposible, el que llama puede imprimir el error y detener el programa de forma normal, pero este curso de acción general se debe reservar para el paquete **main** de un programa. Las funciones de la librería por lo general deben propagar los errores al que llama, a menos que el error sea un signo de una incoherencia interna, es decir, un bug.

```
// (En función main.)
if err := WaitForServer(url); err != nil {
    fmt.Fprintf(os.Stderr, "Site is down: %v\n", err)
    os.Exit(1)
}
```

Una forma más conveniente para lograr el mismo efecto es llamar **log.Fatalf**. Al igual que con todas las funciones de **log**, por defecto se prefija la hora y la fecha del mensaje de error.

```
if err := WaitForServer(url); err != nil {
    log.Fatalf("Site is down: %v\n", err)
}
```

El formato por defecto es útil en un servidor de larga duración, pero no tanto para una herramienta interactiva:

```
2006/01/02 15:04:05 Site is down: no such domain: bad.gopl.io
```

Para una salida más atractiva, podemos establecer el prefijo utilizado por el paquete **log** en el nombre del comando, y suprimir la visualización de la fecha y hora:

```
log.SetPrefix("wait: ")
log.SetFlags(0)
```

En cuarto lugar, en algunos casos, es suficiente sólo registrar el error y luego continuar, tal vez con funcionalidad reducida. Una vez más hay una elección de utilizar el paquete **log**, que añade el prefijo habitual:

```
if err := Ping(); err != nil {
    log.Printf("ping failed: %v; networking disabled", err)
}
```

e imprimir directamente el flujo de error estándar:

```
if err := Ping(); err != nil {
    fmt.Fprintf(os.Stderr, "ping failed: %v; networking disabled\n", err)
}
```

(Todas las funciones **log** añaden una nueva línea si una no está ya presente.)

Y en quinto y último lugar, en casos raros podemos ignorar un error en su totalidad:

```
dir, err := ioutil.TempDir("", "scratch")
if err != nil {
    return fmt.Errorf("failed to create temp dir: %v", err)
}
// ...usar temp dir...
os.RemoveAll(dir) // ignora errores; $TMPDIR es limpiado periodicamente
```

La llamada a **os.RemoveAll** puede fallar, pero el programa lo ignora porque el sistema operativo limpia periódicamente el directorio temporal. En este caso, descarta el error que fue intencional, pero la lógica del programa será la misma que cuando había olvidado tratar con él. Hay que tener un hábito con los errores después de cada llamada a una función, y cuando se ignora deliberadamente uno, documentar su intención claramente.

El tratamiento de errores en **Go** tiene un ritmo particular. Después de comprobar un error, el fallo es generalmente tratado con éxito antes. Si el fallo hace que la función retorne, la lógica para el éxito no se sangra dentro de un bloque **else**, sigue en el nivel externo. Las funciones tienden a exhibir una estructura común, con una serie de comprobaciones iniciales para rechazar errores, seguido por la sustancia de la función, mínimamente sangrada.

5.4.2. Final del archivo (EOF)

Por lo general, la variedad de errores que una función puede devolver es interesante para el usuario final, pero no para la lógica del programa que interviene. Sin embargo, en ocasiones, un programa debe tomar acciones diferentes dependiendo del tipo de error que se ha producido. Considerar la posibilidad de un intento de leer **n** bytes de datos de un archivo. Si **n** es elegido para ser la longitud del archivo, cualquier error representa un fallo. Por otro lado, si el que llama intenta varias veces leer fragmentos de tamaño fijo hasta que se agote el archivo, el que llama debe responder de manera diferente a una condición de fin de archivo que como lo hace para todos los demás errores. Por esta razón, el paquete **io** garantiza que cualquier daño causado por una condición de fin de archivo siempre se informe de un error distinto, **io.EOF**, que se define como sigue:

```
package io
import "errors"
// EOF es el error devuelto por Read cuando no hay más entradas disponibles.
var EOF = errors.New("EOF")
```

El que llama puede detectar esta condición mediante una comparación simple, como en el bucle siguiente, que lee runas de la entrada estándar. (El programa **charcount** en la **Sección 4.3** ofrece un ejemplo más completo.)

```
in := bufio.NewReader(os.Stdin)
for {
    r, _, err := in.ReadRune()
    if err == io.EOF {
        break // finished reading
    }
    if err != nil {
        return fmt.Errorf("read failed: %v", err)
    }
    // ...use r...
}
```

Como en una condición de fin de archivo no existe información, **io.EOF** tiene un mensaje de error fijo, "**EOF**". Para otros errores, es posible que necesitemos un informe de la calidad y la cantidad del error. En la **Sección 7.11**, presentaremos una manera más sistemática para distinguir ciertos valores de error de los demás.

5.5. Valores de función

Las funciones son valores de primera clase en **Go**: al igual que otros valores, los valores de función tienen tipos, y pueden ser asignados a las variables o pasados o retornados de funciones. Un valor de función puede ser llamado como cualquier otra función. Por ejemplo:

```
func square(n int) int { return n * n }
func negative(n int) int { return -n }
func product(m, n int) int { return m * n }

f := square
fmt.Println(f(3)) // "9"

f=negative
fmt.Println(f(3)) // "-3"
fmt.Printf("%T\n", f) // "func(int) int"

f=product // error de compilación: no se puede asignar f(int, int) int to f(int) int
```

El valor cero de un tipo de función es **nil**. Llamar a un valor de función **nil** provoca un pánico:

```
var f func(int) int
f(3) // panic: llamada a función nil
```

Los valores de función pueden compararse con **nil**:

```
var f func(int) int
if f != nil {
    f(3)
}
```

pero no son comparables, por lo que no pueden ser comparados entre sí o utilizarse como claves en un mapa.

Los valores de función nos permiten parametrizar nuestras funciones no sólo con datos, también con el comportamiento. Las librerías estándar contienen muchos ejemplos. Por ejemplo, **strings.Map** aplica una función a cada carácter de un **string**, uniendo los resultados para tomar otro **string**.

```
func add1(r rune) rune { return r + 1 }
fmt.Println(strings.Map(add1, "HAL-9000")) // "IBM.:111"
fmt.Println(strings.Map(add1, "VMS")) // "WNT"
fmt.Println(strings.Map(add1, "Admix")) // "Benjy"
```

La función **findLinks** de la **Sección 5.2** utiliza una función auxiliar, **visit**, para visitar todos los nodos en un documento **HTML** y aplicar una acción a cada uno de ellos. Con el uso de un valor de función, podemos separar la lógica de recorrido del árbol de la lógica de la acción que debe aplicarse a cada nodo, dejandonos reutilizar el recorrido con diferentes acciones.

```
gopl.io/ch5/outline2
// forEachNode llama a las funciones pre(x) y post(x) para cada nodo
// x en el árbol arraigado en n. Ambas funciones son opcionales.
// Se llama pre antes de que los hijos sean visitados (preorder) y
// se llama a post después (postorder).

func forEachNode(n *html.Node, pre, post func(n *html.Node)) {
    if pre != nil {
        pre(n)
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        forEachNode(c, pre, post)
    }
    if post != nil {
        post(n)
    }
}
```

La función **forEachNode** acepta dos argumentos de función, uno para llamar antes de que un nodo hijo sea visitado y uno para llamar después. Esta disposición da al que llama una gran cantidad de flexibilidad. Por ejemplo, las funciones **startElement** y **endElement** imprimen las etiquetas de inicio y final de un elemento **HTML**, como **...**:

```
var depth int

func startElement(n *html.Node) {
    if n.Type == html.ElementNode {
        fmt.Printf("%*s<%s>\n", depth*2, "", n.Data)
        depth++
    }
}
func endElement(n *html.Node) {
```

```
        if n.Type == html.ElementNode {
            depth--
            fmt.Printf("%*s</%s>\n", depth*2, "", n.Data)
        }
    }
```

Las funciones también indentan la salida usando otro truco **fmt.Printf**. El adverbio * en **%*s** imprime un **string** relleno con un número variable de espacios. La anchura y el **string** son proporcionados por los argumentos **depth*2** y **""**.

Si llamamos a **forEachNode** en un documento **HTML**, como aquí:

```
forEachNode(doc, startElement, endElement)
```

obtenemos una variación más elaborada en la salida de nuestro anterior programa **outline**:

```
$ go build gopl.io/ch5/outline2
$ ./outline2 http://gopl.io
<html>
    <head>
        <meta>
        </meta>
        <title>
        </title>
        <style>
        </style>
    </head>
    <body>
        <table>
            <tbody>
                <tr>
                    <td>
                        <a>
                            <img>
                            </img>
...
```

5.6. Funciones anónimas

Las funciones con nombre se pueden declarar sólo a nivel de paquete, pero podemos utilizar un literal de función para indicar un valor de la función dentro de cualquier expresión. Un literal de función se escribe como una declaración de función, pero sin un nombre después de la palabra clave **func.** Es una expresión, y su valor se llama una función anónima.

Los literales de función definen una función en su punto de uso. A modo de ejemplo, la llamada anterior a **strings.Map** se puede reescribir como

```
strings.Map(func(r rune) rune { return r + 1 }, "HAL-9000")
```

Más importante aún, las funciones definidas de esta manera tienen acceso a todo el entorno léxico, por lo que la función interna puede hacer referencia a las variables de la función envolvente, como muestra este ejemplo:

```
gopl.io/ch5/squares
// squares devuelve una función que devuelve
// el siguiente cuadrado de un número cada vez que se llama.
func squares() func() int {
    var x int
    return func() int {
        x++
        return x * x
    }
}
func main() {
    f := squares()
    fmt.Println(f()) // "1"
    fmt.Println(f()) // "4"
    fmt.Println(f()) // "9"
    fmt.Println(f()) // "16"
}
```

La función **squares** devuelve otra función, de tipo **func() int**. Una llamada a **squares** crea una variable local **x** y devuelve una función anónima, cada vez que se llama, en incrementando **x** y retornando su cuadrado. Una segunda llamada a **squares** crearía una segunda variable **x** y retornaría una nueva función anónima incrementando esa variable.

El ejemplo **squares** demuestra que los valores de la función no son sólo código, pueden tener estado. La función interna anónima puede acceder y actualizar las variables locales de la función envolvente **squares**. Estas referencias a variables ocultas son por lo qué clasificamos a las funciones como tipos de referencia y por qué los valores de función

no son comparables. Los valores de función como estos se implementan utilizando una técnica llamada **closures**, y los programadores **Go** a menudo utilizan este término para los valores de función.

Una vez más vemos un ejemplo donde el tiempo de vida de una variable no está determinado por su alcance: la variable **x** existe después de que **squares** ha retornado a **main**, a pesar de que **x** se oculta dentro de **f**.

Como un ejemplo un tanto académico de funciones anónimas, considerar el problema de calcular una secuencia de cursos de informática que satisfacen los requisitos previos de cada uno. Los requisitos previos se dan en la tabla **prereqs** de abajo, que es una correspondencia de cada curso a la lista de cursos que deben completarse antes que ellos.

```
gopl.io/ch5/toposort
// prereqs mapea cursos de informática a sus prerequisitos.
var prereqs = map[string][]string{
    "algorithms": {"data structures"},
    "calculus": {"linear algebra"},
    "compilers": {
        "data structures",
        "formal languages",
        "computer organization",
    },
    "data structures": {"discrete math"},
    "databases": {"data structures"},
    "discrete math": {"intro to programming"},
    "formal languages": {"discrete math"},
    "networks": {"operating systems"},
    "operating systems": {"data structures", "computer organization"},
    "programming languages": {"data structures", "computer organization"},
}
```

Este tipo de problema se conoce como clasificación topológica. Conceptualmente, la información de pre-requisito forma un grafo dirigido con un nodo para cada curso y ejes de cada curso a los cursos de los que depende. La gráfica no es cíclica: no hay camino de un curso que conduzca de nuevo a sí mismo. Podemos calcular una secuencia válida mediante la búsqueda primero en profundidad a través de la gráfica con el código de abajo:

```
func main() {
    for i, course := range topoSort(prereqs) {
        fmt.Printf("%d:\t%s\n", i+1, course)
    }
}
func topoSort(m map[string][]string) []string {
    var order []string
    seen := make(map[string]bool)
    var visitAll func(items []string)
    visitAll = func(items []string) {
        for _, item := range items {
            if !seen[item] {
                seen[item] = true
                visitAll(m[item])
                order = append(order, item)
            }
        }
    }
    var keys []string
    for key := range m {
        keys = append(keys, key)
    }
    sort.Strings(keys)
    visitAll(keys)
    return order
}
```

Cuando una función anónima requiere repetición, como en este ejemplo, primero debemos declarar una variable, y luego asignar la función anónima a esa variable. Si estos dos pasos se han combinado en la declaración, el literal de función no estarán dentro del alcance de la variable **visitAll** por lo que no tendría ninguna manera de llamarse a sí mismo de forma recursiva:

```
visitAll := func(items []string) {
    // ...
    visitAll(m[item]) // error de compilación: no definido: visitAll
    // ...
}
```

La salida del programa **toposort** se muestra a continuación. Es determinista, una propiedad a menudo deseable que no siempre vienen de forma gratuita. En este caso, los valores del mapa **prereqs** son **slices**, no más mapas, por lo que su orden de iteración es determinista, y se han ordenado las claves de **prereqs** antes de hacer las llamadas iniciales a **visitAll**.

```
1: intro to programming
2: discrete math
3: data structures
4: algorithms
5: linear algebra
6: calculus
7: formal languages
8: computer organization
9: compilers
10: databases
11: operating systems
12: networks
13: programming languages
```

Vamos a volver a nuestro ejemplo **findLinks**. Hemos movido la función de extracción de enlace de **links.Extract** a su propio paquete, ya que la utilizaremos de nuevo en el **Capítulo 8**. Hemos sustituido la función **visit** con una función anónima que se añade al slice **links** directamente, y se utiliza **forEachNode** para manejar el recorrido. Como **Extract** sólo necesita la función **pre**, pasa **nil** en el argumento **post**.

```
gopl.io/ch5/links
// El paquete links proporciona una función link-extraction.
package links

import (
    "fmt"
    "net/http"
    "golang.org/x/net/html"
)
// Extract realiza una solicitud GET HTTP a la URL especificada, analiza
// la respuesta como HTML, y devuelve los enlaces en el documento HTML.
func Extract(url string) ([]string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("getting %s: %s", url, resp.Status)
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    var links []string
    visitNode := func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "a" {
            for _, a := range n.Attr {
                if a.Key != "href" {
                    continue
                }
                link, err := resp.Request.URL.Parse(a.Val)
                if err != nil {
                    continue // ignorar las malas URLs
                }
                links = append(links, link.String())
            }
        }
    }
    forEachNode(doc, visitNode, nil)
    return links, nil
}
```

En lugar de añadir el valor del atributo en bruto **href** al **slice** links, esta versión lo analiza como una **URL** relativa a la URL base del documento, **resp.Request.URL**. El resultante **link** está en forma absoluta, adecuado para su uso en una llamada a **http.Get**.

El rastreo de la web es, en el fondo, un problema de recorrido del grafo. El ejemplo **topoSort** mostró un recorrido en profundidad primera transversal; para nuestro rastreador web, vamos a usar el recorrido de amplitud primera transversal, al menos inicialmente. En el **Capítulo 8**, exploraremos el recorrido concurrente transversal.

La siguiente función encapsula la esencia de un recorrido primero en amplitud. El que llama proporciona una lista inicial **worklist** de artículos para visitar y un valor de función **f** para llamar a cada elemento. Cada elemento está identificado por un **string**. La función **f** devuelve una lista de nuevos elementos a añadir a la **worklist**. La función **breadthFirst** retorna cuando todos los elementos han sido visitados. Mantiene un conjunto de **strings** para asegurar que no hay ningún elemento visitado dos veces.

```
gopl.io/ch5/findlinks3
// breadthFirst llama f para cada elemento en la lista de trabajo.
// Todos los objetos devueltos por f se añaden a la lista de trabajo.
// f se llama como máximo una vez por cada elemento.
func breadthFirst(f func(item string) []string, worklist []string) {
    seen := make(map[string]bool)
    for len(worklist) > 0 {
        items := worklist
        worklist = nil
        for _, item := range items {
            if !seen[item] {
                seen[item] = true
                worklist = append(worklist, f(item)...)
            }
        }
    }
}
```

Tal como explicamos de paso en el **Capítulo 3**, el argumento "**f(item)...**" hace que todos los elementos de la lista devueltos por **f** se añadirán a la **worklist**.

En nuestro rastreador, los elementos son direcciones **URL**. La función **crawl** que suministra **breadthFirst** imprime la **URL**, extrae sus enlaces, y los devuelve para que también sean visitados.

```
func crawl(url string) []string {
    fmt.Println(url)
    list, err := links.Extract(url)
    if err != nil {
        log.Print(err)
    }
    return list
}
```

Para iniciar el rastreador apagado, utilizaremos los argumentos de línea de comandos como las **URLs** iniciales.

```
func main() {
    // Rastrear la web breadth-first,
    // a partir de los argumentos de línea de comandos.
    breadthFirst(crawl, os.Args[1:])
}
```

Vamos a rastrear la web a partir de **https://golang.org**. Éstos son algunos de los enlaces resultantes:

```
$ go build gopl.io/ch5/findlinks3
$ ./findlinks3 https://golang.org
https://golang.org/
https://golang.org/doc/
https://golang.org/pkg/
https://golang.org/project/
https://code.google.com/p/go-tour/
https://golang.org/doc/code.html
https://www.youtube.com/watch?v=XCsL89YtqCs
http://research.swtch.com/gotour
https://vimeo.com/53221560
...
```

El proceso termina cuando todas las páginas web accesibles se han rastreado o la memoria de la computadora se ha agotado.

5.6.1. Advertencia: Captura de variables de iteración

En esta sección, echaremos un vistazo a las reglas de alcance del léxico de **Go** que pueden causar resultados sorprendentes. Instamos a entender el problema antes de continuar, porque la trampa puede atrapar incluso a los programadores experimentados.

Considerar la posibilidad de un programa que debe crear un conjunto de directorios y posteriormente eliminarlos. Podemos utilizar un **slice** de valores de función para mantener las operaciones de limpieza. (Por razones de brevedad, se han omitido todo el manejo de errores en este ejemplo.)

```
var rmdirs []func()
for _, d := range tempDirs() {
    dir := d // NOTA: ¡necesario!
    os.MkdirAll(dir, 0755) // crea directorio padre también
    rmdirs = append(rmdirs, func() {
        os.RemoveAll(dir)
    })
```

```
}
// ...hacer algún trabajo...
for _, rmdir := range rmdirs {
    rmdir() // limpieza
}
```

Es posible que nos preguntemos por qué se asignó la variable de bucle **d** a una nueva variable local **dir** dentro del cuerpo del bucle, en lugar de sólo nombrar la variable de bucle **dir** en esta variante sutilmente incorrecta:

```
var rmdirs []func()
for _, dir := range tempDirs() {
    os.MkdirAll(dir, 0755)
    rmdirs = append(rmdirs, func() {
        os.RemoveAll(dir) // NOTA: ¡incorrecto!
    })
}
```

La razón es una consecuencia de las reglas de alcance para las variables de bucle. En el programa inmediatamente anterior, el bucle **for** introduce un nuevo bloque de léxico en el que se declara la variable **dir**. Todos los valores de función creados por este bucle "capturan" y comparten la misma variable, una ubicación de almacenamiento direccionable, no su valor en ese momento en particular. El valor de **dir** se actualiza en iteraciones sucesivas, de manera que cuando las funciones de limpieza se llaman, la variable **dir** se ha actualizado varias veces por el bucle **for** completado. Por lo tanto **dir** mantiene el valor de la iteración final, y por lo tanto todas las llamadas a **os.RemoveAll** intentarán quitar el mismo directorio.

Con frecuencia, la variable interna introducida para funcionar con este problema, **dir** en nuestro ejemplo, se le da exactamente el mismo nombre que la variable externa de la que es una copia, lo que lleva a declaraciones de variables de aspecto extraño pero cruciales como esta:

```
for _, dir := range tempDirs() {
    dir := dir // declara dir interna, inicializada por externa dir
    // ...
}
```

El riesgo no es único para bucles basados en **range**. El bucle en el siguiente ejemplo adolece del mismo problema debido a la captura no intencionada de la variable de índice **i**.

```
var rmdirs []func()
dirs := tempDirs()
for i := 0; i < len(dirs); i++ {
    os.MkdirAll(dirs[i], 0755) // OK
    rmdirs = append(rmdirs, func() {
        os.RemoveAll(dirs[i]) // NOTE: incorrect!
    })
}
```

El problema de la iteración de captura variable se encuentra más a menudo cuando se utiliza la sentencia **go** (**Capítulo 8**) o con **defer** (que veremos en un momento), ya que ambos pueden retrasar la ejecución de un valor de función hasta después de que el bucle haya finalizado. Pero el problema no es inherente a **go** o **defer**.

5.7. Funciones variadic

Una función **variadic** es una que se puede llamar con un número de argumentos variable. Los ejemplos más conocidos son **fmt.Printf** y sus variantes. **Printf** requiere un argumento fijo al inicio, a continuación, acepta cualquier número de argumentos posteriores.

Para declarar una función **variadic**, el tipo del parámetro final es precedido por puntos suspensivos, "**...**'', lo que indica que la función puede ser llamada con cualquier número de argumentos de este tipo.

```
gopl.io/ch5/sum
func sum(vals ...int) int {
    total := 0
    for _, val := range vals {
        total += val
    }
    return total
}
```

La función **sum** anterior devuelve la suma de cero o más **int** argumentos. Dentro del cuerpo de la función, el tipo de **vals** es un slice **[]int**. Cuando **sum** se llama, cualquier número de valores puede ser proporcionado por su parámetro **vals**.

```
fmt.Println(sum()) // "0"
fmt.Println(sum(3)) // "3"
fmt.Println(sum(1, 2, 3, 4)) // "10"
```

Implícitamente, el que llama asigna un arreglo, copia los argumentos en él, y pasa un **slice** de todo el arreglo a la función. La última llamada por encima de este modo se comporta igual que la llamada a continuación, que muestra

cómo invocar una función **variadic** cuando los argumentos ya están en un **slice**: colocar unos puntos suspensivos después del argumento final.

```
values := []int{1, 2, 3, 4}
fmt.Println(sum(values...)) // "10"
```

Aunque el parámetro **...int** se comporta como un **slice** dentro del cuerpo de la función, el tipo de una función **variadic** es distinto del tipo de una función con un parámetro ordinario de **slice**.

```
func f(...int) {}
func g([]int) {}
fmt.Printf("%T\n", f) // "func(...int)"
fmt.Printf("%T\n", g) // "func([]int)"
```

Las funciones **variadic** a menudo se utilizan para formateo de **strings**. La función **errorf** por debajo construye un mensaje de error formateado con un número de línea al principio. El sufijo **f** es una convención de nomenclatura ampliamente seguida para las funciones **variadic** que aceptan un **string** de formato estilo **Printf**.

```
func errorf(linenum int, format string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, "Line %d: ", linenum)
    fmt.Fprintf(os.Stderr, format, args...)
    fmt.Fprintln(os.Stderr)
}
linenum, name := 12, "count"
errorf(linenum, "undefined: %s", name) // "Linea 12: no definido: count"
```

El tipo **interface{}** significa que esta función puede aceptar cualquier valor en sus argumentos finales, lo explicaremos en el **Capítulo 7**.

5.8. Las llamadas a funciones diferidas

Nuestros ejemplos **findLinks** utilizan la salida del **http.Get** como la entrada a **html.Parse**. Esto funciona bien si el contenido de la dirección **URL** solicitada es **HTML**, pero muchas páginas contienen imágenes, texto plano, y otros formatos de archivo. La alimentación de estos archivos en un analizador de **HTML** podría tener efectos indeseables.

El programa siguiente obtiene un documento **HTML** e imprime su título. La función **title** inspecciona la cabecera **Content-Type** de la respuesta del servidor y devuelve un error si el documento no es **HTML**.

```
gopl.io/ch5/title1
func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }
    // Prueba si el tipo de contenido es HTML (ejemplo, "text/html; charset=utf-8").
    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        resp.Body.Close()
        return fmt.Errorf("%s has type %s, not text/html", url, ct)
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    visitNode := func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "title" && n.FirstChild != nil {
            fmt.Println(n.FirstChild.Data)
        }
    }
    forEachNode(doc, visitNode, nil)
    return nil
}
```

Aquí vemos una sesión típica, con pequeñas modificaciones editadas:

```
$ go build gopl.io/ch5/title1
$ ./title1 http://gopl.io
The Go Programming Language
$ ./title1 https://golang.org/doc/effective_go.html
Effective Go - The Go Programming Language
$ ./title1 https://golang.org/doc/gopher/frontpage.png
title: https://golang.org/doc/gopher/frontpage.png
has type image/png, not text/html
```

Observar la llamada duplicada a **resp.Body.Close()**, lo que garantiza que **title** cierre la conexión de red en todas las rutas de ejecución, incluidos los fallos. A medida que las funciones se vuelven más complejas y tienen que manejar

más errores, como la duplicación de la lógica de limpieza, puede llegar a ser un problema de mantenimiento. Vamos a ver cómo el nuevo mecanismo **defer** hace las cosas más simples.

Sintácticamente, una sentencia **defer** es una función o un método de llamada ordinaria prefijado por la palabra clave **defer**. Las expresiones de función y argumentos se evalúan cuando se ejecuta la sentencia, pero la llamada real se aplaza hasta que la función que contiene la sentencia **defer** haya terminado, normalmente, mediante la ejecución de una sentencia **return,** o saliendo antes del final, o anormalmente, por pánico. Cualquier número de llamadas puede ser diferido; serán ejecutadas en el orden inverso el en el que fueron diferidas.

Una sentencia **defer** se utiliza a menudo con operaciones emparejadas como abrir y cerrar, conectar y desconectar o bloquear y desbloquear para garantizar que los recursos se liberan en todos los casos, independientemente de la complejidad del flujo de control. El lugar adecuado para una sentencia **defer** que libera un recurso es inmediatamente después de que el recurso se haya adquirido con éxito. En la función **title** a continuación, una sola llamada diferida sustituye a las dos llamadas anteriores a **resp.Body.Close()**:

```
gopl.io/ch5/title2
func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }
    defer resp.Body.Close()
    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        return fmt.Errorf("%s has type %s, not text/html", url, ct)
    }
    doc, err := html.Parse(resp.Body)
    if err != nil {
        return fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    // ...imprime elemento del título del doc...
    return nil
}
```

El mismo patrón se puede utilizar para otros recursos como las conexiones de red, por ejemplo para cerrar un archivo abierto:

```
io/ioutil
package ioutil
func ReadFile(filename string) ([]byte, error) {
    f, err := os.Open(filename)
    if err != nil {
        return nil, err
    }
    defer f.Close()
    return ReadAll(f)
}
```

o para desbloquear un **mutex** (**§9.2**):

```
var mu sync.Mutex
var m = make(map[string]int)
func lookup(key string) int {
    mu.Lock()
    defer mu.Unlock()
    return m[key]
}
```

La sentencia **defer** también se puede utilizar para emparejar acciones "en la entrada" y "en la salida" al depurar una función compleja. La función **bigSlowOperation** por debajo llama a **trace** inmediatamente, que hace la acción "en la entrada" a continuación, devuelve un valor de la función que, cuando se le llama, hace lo que corresponde a la acción "en la salida". Al aplazar la llamada a la función retornada de esta manera, podemos instrumentar el punto de entrada y todos los puntos de salida de una función en una única sentencia e incluso pasar valores, como el tiempo **start**, entre las dos acciones. Pero no hay que olvidar los paréntesis finales en la sentencia **defer**, o la acción "en la entrada" ocurrirá en la salida y ¡la acción de salida no ocurrirá en absoluto!

```
gopl.io/ch5/trace
func bigSlowOperation() {
    defer trace("bigSlowOperation")() // no olvidar los paréntesis extras
    // ...lots of work...
    time.Sleep(10 * time.Second) // simula operación lenta con sleeping
}
func trace(msg string) func() {
    start := time.Now()
    log.Printf("enter %s", msg)
    return func() { log.Printf("exit %s (%s)", msg, time.Since(start)) }
}
```

Cada vez que se llama a **bigSlowOperation** se, registra su entrada y salida y el tiempo transcurrido entre ellas. (Utilizamos **time.Sleep** para simular una operación lenta.)

```
$ go build gopl.io/ch5/trace
$ ./trace
2015/11/18 09:53:26 enter bigSlowOperation
2015/11/18 09:53:36 exit bigSlowOperation (10.000589217s)
```

Las funciones diferidas se ejecutan después de que las sentencias **return** han actualizado las variables del resultado de las funciones. Debido a que una función anónima puede tener acceso a las variables de la función que la encierra, incluyendo los resultados con nombre, una función anónima diferida puede observar los resultados de la función.

Considerar la función **double**:

```
func double(x int) int {
    return x + x
}
```

Al nombrar la variable de resultado y añadir una sentencia **defer**, podemos hacer que la función imprima sus argumentos y resultados cada vez que se llama.

```
func double(x int) (result int) {
    defer func() { fmt.Printf("double(%d) = %d\n", x, result) }()
    return x + x
}
_=double(4)
// Salida:
// "double(4) = 8"
```

Este truco es demasiado para una función tan simple como **double**, pero puede ser útil en funciones con muchas sentencias **return**.

Una función anónima diferida puede incluso cambiar los valores que la función que la encierra devuelve al que llama:

```
func triple(x int) (result int) {
    defer func() { result += x }()
    return double(x)
}
fmt.Println(triple(4)) // "12"
```

Dado que las funciones diferidas no son ejecutadas hasta el verdadero final de la ejecución de una función, una sentencia **defer** en un bucle merece un examen adicional. El código de abajo podría quedarse sin descriptores de archivos ya que ningún archivo se cerrará hasta que todos los archivos hayan sido procesados:

```
for _, filename := range filenames {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close() // NOTA: arriesgado; podría quedarse sin descriptores de archivos
    // ...procesando f...
}
```

Una solución es mover el cuerpo del bucle, incluyendo la sentencia **defer**, en otra función que se llama en cada iteración.

```
for _, filename := range filenames {
    if err := doFile(filename); err != nil {
        return err
    }
}
func doFile(filename string) error {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close()
    // ...procesando f...
}
```

El ejemplo siguiente es una mejora del programa **fetch** (**§1.5**) que escribe la respuesta **HTTP** a un archivo local en lugar de a la salida estándar. Se deriva el nombre del archivo desde el último componente de la ruta **URL**, que se obtiene mediante la función **path.Base**.

```
gopl.io/ch5/fetch
// Fetch descarga la URL y devuelve el
// nombre y la longitud del archivo local.
func fetch(url string) (filename string, n int64, err error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", 0, err
```

```
    }
    defer resp.Body.Close()
    local := path.Base(resp.Request.URL.Path)
    if local == "/" {
        local = "index.html"
    }
    f, err := os.Create(local)
    if err != nil {
        return "", 0, err
    }
    n, err = io.Copy(f, resp.Body)
    // Cerrar el archivo, pero se prefieren el error desde Copy, si los hubiere.
    if closeErr := f.Close(); err == nil {
        err = closeErr
    }
    return local, n, err
}
```

La llamada diferida a **resp.Body.Close** debe ser familiar por ahora. Es tentador utilizar una segunda llamada diferida, a **f.Close**, para cerrar el archivo local, pero esto sería sutilmente malo porque **os.Create** abre un archivo para escritura, creándolo si es necesario. En muchos sistemas de archivos, **NFS** en particular, los errores de escritura no se informan de inmediato, puede pospuesto hasta que el archivo esté cerrado. Si no se revisa el resultado de la operación de cierre podría causar pérdida de datos importantes desapercibidos. Sin embargo, si ambos **io.Copy** y **f.Close** fallan, es preferible informar del error de **io.Copy** ya que ocurrió por primera vez y es más probable que nos diga la causa raíz.

5.9. Pánico

El sistema de tipo de **Go** atrapa muchos errores en tiempo de compilación, pero otros, como un acceso fuera de límites de un arreglo o una referencia a un puntero **nil**, requieren controles en tiempo de ejecución. Cuando el **runtime** de **Go** detecta estos errores, entra en pánico.

Durante un típico pánico, se detiene la ejecución normal, todas las llamadas de funciones diferidas en esa **goroutine** se ejecutan, y el programa se bloquea con un mensaje de registro. Este mensaje de registro incluye el valor de pánico, que suele ser un mensaje de error de algún tipo, y, para cada **goroutine**, un seguimiento de la pila (**stack trace**) que muestra la pila de llamadas a funciones que estaban activas en el momento del pánico. Este mensaje de registro a menudo tiene suficiente información para diagnosticar la causa raíz del problema sin ejecutar el programa de nuevo, por lo que siempre debe incluirse en un informe de errores acerca de un programa en pánico.

No todos los pánicos vienen del tiempo de ejecución. La función incorporada **panic** puede ser llamada directamente; acepta cualquier valor como argumento. Un pánico es a menudo la mejor cosa que se puede hacer cuando ocurre alguna situación "imposible", por ejemplo, la ejecución alcanza un caso que lógicamente no puede suceder:

```
switch s := suit(drawCard()); s {
case "Spades": // ...
case "Hearts": // ...
case "Diamonds": // ...
case "Clubs": // ...
default:
    panic(fmt.Sprintf("invalid suit %q", s)) // Joker?
}
```

Es una buena práctica para afirmar que las condiciones previas de una función se sostienen, pero esto se puede hacer fácilmente en exceso. A menos que se puede brindar un mensaje de error más informativo o detectar un error antes, no tiene sentido hacer valer la condición de que el **runtime** compruebe por nosotros.

```
func Reset(x *Buffer) {
    if x == nil {
        panic("x is nil") // innecesario!
    }
    x.elements = nil
}
```

Aunque el mecanismo de pánico de **Go** se asemeja a las excepciones en otros lenguajes, las situaciones en las que el pánico se utiliza son bastante diferentes. Como un pánico hace que el programa se bloquee, se utiliza generalmente para errores graves, tales como una inconsistencia lógica en el programa; los programadores diligentes consideran cualquier accidente como prueba de un error en su código. En un programa robusto, los errores "esperados", de esos que surgen de la entrada incorrecta, mala configuración, o en su defecto de **E/S**, deben ser manejados con gracia; se tratan mejor con el uso de valores de **error**.

Considerar la función **regexp.Compile**, que compila una expresión regular en una forma eficiente para la coincidencia. Se devuelve un **error** si se llama con un patrón mal formado, pero la comprobación de este error es innecesaria y onerosa si el que llama sabe que una llamada en particular no puede fallar. En tales casos, no es razonable para el que llama manejar un error de pánico, ya que cree que es imposible.

Como la mayoría de las expresiones regulares son literales en el código fuente del programa, el paquete **regexp** proporciona una función contenedora **regexp.MustCompile** que hace esta comprobación:

```
package regexp

func MustCompile(expr string) *Regexp {
    re, err := Compile(expr)
    if err != nil {
        panic(err)
    }
    return re
}
```

La función contenedora hace que sea conveniente para los clientes inicializar una variable de nivel de paquete con una expresión regular compilada, como esta:

```
var httpSchemeRE = regexp.MustCompile(`^https?:`) // "http:" or "https:"
```

Por supuesto, **MustCompile** no debe ser llamada con valores de entrada que no son de confianza. El prefijo **Must** es una convención de nomenclatura común para este tipo de funciones, como **template.Must** en la **Sección 4.6**.

Cuando se produce una situación de pánico, todas las funciones diferidas se ejecutan en orden inverso, empezando por la función superior en la pila y procediendo hasta **main**, como muestra el programa siguiente:

```
gopl.io/ch5/defer1
func main() {
    f(3)
}

func f(x int) {
    fmt.Printf("f(%d)\n", x+0/x) // panico si x == 0
    defer fmt.Printf("defer %d\n", x)
    f(x - 1)
}
```

Cuando se ejecuta, el programa imprime lo siguiente a la salida estándar:

```
f(3)
f(2)
f(1)
defer 1
defer 2
defer 3
```

Se produce un pánico durante la llamada a **f(0)**, haciendo que las tres llamadas diferidas a **fmt.Printf** se ejecuten. A continuación, el **runtime** termina el programa, imprime el mensaje de pánico y hace un volcado de pila al **stream** de error estándar (simplificado para mayor claridad):

```
panic: runtime error: integer divide by zero
main.f(0)
    src/gopl.io/ch5/defer1/defer.go:14
main.f(1)
    src/gopl.io/ch5/defer1/defer.go:16
main.f(2)
    src/gopl.io/ch5/defer1/defer.go:16
main.f(3)
    src/gopl.io/ch5/defer1/defer.go:16
main.main()
    src/gopl.io/ch5/defer1/defer.go:10
```

Tal como veremos pronto, es posible que una función pueda recuperarse de una situación de pánico por lo que no termina el programa.

Para el propósito de diagnóstico, el paquete **runtime** permite al programador el volcado de la pila utilizando la misma maquinaria. Al aplazar una llamada a **printStack** en **main**,

```
gopl.io/ch5/defer2
func main() {
    defer printStack()
    f(3)
}

func printStack() {
    var buf [4096]byte
    n := runtime.Stack(buf[:], false)
    os.Stdout.Write(buf[:n])
}
```

el siguiente texto adicional (una vez más simplificado para mayor claridad) se imprime en la salida estándar:

```
goroutine 1 [running]:
main.printStack()
    src/gopl.io/ch5/defer2/defer.go:20
main.f(0)
    src/gopl.io/ch5/defer2/defer.go:27
main.f(1)
    src/gopl.io/ch5/defer2/defer.go:29
main.f(2)
    src/gopl.io/ch5/defer2/defer.go:29
main.f(3)
    src/gopl.io/ch5/defer2/defer.go:29
main.main()
    src/gopl.io/ch5/defer2/defer.go:15
```

Los lectores familiarizados con excepciones en otros lenguajes pueden sorprenderse de que **runtime.Stack** pueda imprimir información sobre las funciones que parecen ya han sido "desenrolladas". El mecanismo de pánico de **Go** ejecuta las funciones diferidas antes de que se desenrolle la pila.

5.10. Recover

Darse por vencido es por lo general la respuesta adecuada a una situación de pánico, pero no siempre. Puede ser que sea posible recuperarse de alguna manera, o al menos limpiar el desastre antes de salir. Por ejemplo, un servidor web que se encuentra con un problema inesperado podría cerrar la conexión en lugar de dejar colgado el cliente, y durante el desarrollo, es posible que informe del error al cliente también.

Si la función incorporada **recover** se llama dentro de una función diferida y la función que contiene la sentencia **defer** es presa del pánico, **recover** termina el actual estado de pánico y devuelve el valor de pánico. La función que fue presa del pánico no continua donde lo dejó, pero retorna normalmente. Si se llama a **recover** en cualquier otro momento, no tiene ningún efecto y retorna **nil**.

Como ejemplo, consideremos el desarrollo de un programa de análisis de un lenguaje. Incluso cuando parece estar funcionando bien, dada la complejidad de su trabajo, los errores todavía pueden estar al acecho en casos en las oscuras esquinas. Podríamos preferir que, en lugar de estrellarse, el analizador convierta estos pánicos en errores de análisis ordinarios, tal vez con un mensaje adicional exhortando al usuario un informe de error.

```
func Parse(input string) (s *Syntax, err error) {
    defer func() {
        if p := recover(); p != nil {
            err = fmt.Errorf("internal error: %v", p)
        }
    }()
    // ...parser...
}
```

La función diferida en **Parse** se recupera de una situación de pánico, utilizando el valor de pánico para construir un mensaje de error; una versión más elegante podría incluir toda la pila de llamadas usando **runtime.Stack**. La función diferida a continuación, asigna a **err** el resultado, que será devuelto al que llama.

La recuperación de manera indiscriminada de pánico es una práctica dudosa debido a que el estado de las variables de un paquete tras una emergencia es raramente bien definido o documentado. Tal vez una actualización crítica de una estructura de datos estaba incompleta, una conexión de archivo o la red se abrió, pero no se cerró, o una cerradura fue adquirida pero no liberada. Por otra parte, mediante la sustitución de un fallo con, por ejemplo, una línea en un archivo de registro, la recuperación indiscriminada puede provocar errores que pueden pasar desapercibidos.

Para recuperarse de una situación de pánico dentro del mismo paquete puede ayudar el simplificar el manejo de errores complejos o inesperados, pero como regla general, no se debería intentar recuperarse de un pánico de otro paquete. Las **API** públicas deben informar de cualquier fallo como **errors**. Del mismo modo, no debe recuperarse de una situación de pánico que pueden pasar a través de una función que no se mantiene, tales como una llamada de retorno proporcionada, ya que no se puede razonar acerca de su seguridad.

Por ejemplo, el paquete **net/http** proporciona un servidor web que envía las solicitudes entrantes a funciones de controlador proporcionadas por el usuario. En lugar de dejar que el pánico en uno de estos controladores mate el proceso, el servidor llama a **recover**, imprime un seguimiento de la pila, y sigue sirviendo. Esto es conveniente en la práctica, pero sí tiene riesgo de fugas de recursos o salir del controlador en un estado fallido no especificado que podría conducir a otros problemas.

Por todas las razones anteriores, es más seguro recuperar selectivamente en todo caso. En otras palabras, recuperar sólo de pánicos que fueron destinados a ser recuperados, debe ser raro. Esta intención puede ser codificada mediante el uso de un tipo distinto, dejados por el valor de pánico y probando si el valor devuelto por **recover** tiene ese tipo. (Vemos una manera de hacer esto en el siguiente ejemplo.) Si es así, se informa de pánico como un error ordinario; si no, llamamos a **panic** con el mismo valor para reanudar el estado de pánico.

El ejemplo siguiente es una variación del programa **title** que informa de un error si el documento **HTML** contiene múltiples elementos **<title>**. Si es así, se aborta la recursividad llamando a **panic** con un valor del tipo especial **bailout**.

```
gopl.io/ch5/title3
// soleTitle devuelve el texto del primer elemento de título no vacío
// en el documento, y un error si no era exactamente uno.
func soleTitle(doc *html.Node) (title string, err error) {
    type bailout struct{}
    defer func() {
        switch p := recover(); p {
        case nil:
            // sin pánico
        case bailout{}:
            // panico "esperado"
            err = fmt.Errorf("multiple title elements")
        default:
            panic(p) // pánico inesperado; continuar pánico
        }
    }()
    // Fianza de recursividad si encontramos más de un título no vacío.
    forEachNode(doc, func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "title" &&
            n.FirstChild != nil {
            if title != "" {
                panic(bailout{}) // multiples elementos title
            }
            title = n.FirstChild.Data
        }
    }, nil)
    if title == "" {
        return "", fmt.Errorf("no title element")
    }
    return title, nil
```

La función manejadora diferida llama a **recover**, comprueba el valor de pánico, e informa de un error ordinario si el valor era **bailout{}**. Todos los demás valores no **nil** indican un pánico inesperado, en cuyo caso el controlador llama a **panic** con ese valor, deshaciendo el efecto de **recover** y reanudando el estado original de pánico. (En este ejemplo se violan nuestros consejos acerca de no usar los errores de pánicos "esperados", proporciona una ilustración compacta de la mecánica.)

En algunas condiciones no hay recuperación. Por ejemplo, un mal funcionamiento de la memoria, hace que el **runtime** de **Go** termine el programa con un error fatal.