Capítulo 1

Tutorial

Este capítulo es un recorrido por los componentes básicos de **Go**. Esperamos proporcionar suficiente información y ejemplos para conseguir hacer cosas útiles lo más rápido posible. Los ejemplos aquí, y de hecho en todo el libro, están destinados a tareas que puede que se tengan que hacer en el mundo real. En este capítulo, "trataremos de dar una idea de la diversidad de los programas que se podrían escribir en **Go**, que van desde el procesamiento de archivos simple y un poco de gráficos a los clientes de Internet y servidores concurrentes. Ciertamente no ganamos al explicar todo en el primer capítulo, pero el estudio de este tipo de programas en un nuevo lenguaje puede ser una manera eficaz para empezar.

Cuando se aprender un nuevo lenguaje, hay una tendencia natural a escribir código, que se hubiera escrito en un lenguaje que ya se conoce. Ser conscientes de esta tendencia a medida que se aprende y tratar de evitarlo.

1.1. Hola Mundo

Comenzaremos con el ya tradicional ejemplo "hola, mundo", que aparece al comienzo del lenguaje de programación **C**, publicado en **1978**. **C** es una de las influencias más directas en **Go**, y "hola, mundo" ilustra una serie de ideas centrales.

```
gopl.io/ch1/helloworld
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

Go es un lenguaje compilado. La cadena de herramientas **Go** convierten un código de programa y las cosas de las que depende en instrucciones en lenguaje de máquina nativo de un ordenador. Estas herramientas son accedidas a través de un único comando llamado **go** que tiene una serie de subcomandos. El más simple de estos subcomandos es **run**, que compila el código fuente de uno o más archivos cuyos nombres terminan en **.go**, lo vincula con las bibliotecas y a continuación, ejecuta el archivo ejecutable resultante. (Utilizaremos **$** como el símbolo del sistema en todo el libro.)

```
$ go run helloworld.go
```

No es sorprendente que imprima

```
Hello, 世界
```

Go maneja Unicode nativo, por lo que se puede procesar texto en todas las lenguas del mundo.

Si el programa es más que un simple experimento es probable que se desee compilar una vez y guardar el resultado compilado para su uso posterior. Esto se hace con **go build**:

```
$ go build helloworld.go
```

Esto crea un archivo binario ejecutable llamado **helloworld** que se pueden ejecutar en cualquier momento sin más transformación:

```
$ ./helloworld
Hello, 世界
```

Hemos llamado a cada ejemplo significativo como un recordatorio de que se puede obtener el código del repositorio del código fuente del libro en **gopl.io**:

```
gopl.io/ch1/helloworld
```

Si se ejecuta **go get gopl.io/ch1/helloworld**, se obtendrá el código fuente y lo coloca en el directorio correspondiente. Hay más sobre este tema en la **Sección 2.6** y la **Sección 10.7**.

El código de **Go** está organizado en paquetes, que son similares a las bibliotecas o módulos en otros lenguajes. Un paquete se compone de uno o más ficheros fuente **.go** en un único directorio que definen lo que hace el paquete. Cada archivo fuente se inicia con una declaración **package**, aquí **package main**, que indica a qué paquete pertenece el archivo, seguido de una lista de otros paquetes que importa, y luego las declaraciones del programa que se almacenan en el archivo.

La biblioteca estándar de **Go** tiene más de **100** paquetes para tareas comunes como la entrada y la salida, clasificación y manipulación de texto. Por ejemplo, el paquete **fmt** contiene funciones para salida de impresión con formato y entrada de escaneo. **Println** es una de las funciones de salida básicas en **fmt** ; imprime uno o varios valores,

separados por espacios, con un carácter de nueva línea al final para que los valores aparezcan como una sola línea de salida.

Package main es especial. Define un programa ejecutable independiente, no una librería. Dentro del paquete **main** la función **main** es también especial, es donde comienza la ejecución del programa. Cualquiera cosa que haga **main** es lo que hace el programa. Por supuesto, **main** será normalmente llamará a las funciones en otros paquetes para hacer gran parte del trabajo, tales como la función **fmt.Println**.

Hay que indicar al compilador que paquetes son necesarios para este archivo fuente; este es el papel de la declaración **import** que sigue a la declaración **package**. El programa "hola, mundo " utiliza sólo una función de otro paquete, pero la mayoría de los programas importarán más paquetes.

Se deben importar exactamente los paquetes que se necesitan. Un programa no se compila si faltan importaciones o si las hay innecesarias. Este requisito estricto impide que se acumulen referencias a paquetes no utilizados a medida que los programas crecen.

Las declaraciones **import** deben seguir a la declaración **package**. Después de eso, un programa consiste en las declaraciones de funciones, variables, constantes y tipos (introducidos por las palabras clave **func**, **var**, **const**, y **type**); en su mayor parte, el orden de las declaraciones no importa. Este programa es de lo más corto posible, ya que declara una única función, que a su vez llama sólo a otra función. Para ahorrar espacio, a veces no vamos a mostrar el paquete y las declaraciones de importación en la presentación de ejemplos, pero están en el archivo de fuente y deben estar allí para compilar el código.

Una declaración de función consiste en la palabra clave **func**, el nombre de la función, una lista de parámetros (vacío para **main**), una lista de resultados (también vacía aquí), y el cuerpo de la función, las sentencias que definen lo que hay que hacer, encerrado entre llaves. Veremos más sobre las funciones en el **Capítulo 5**.

Go no requiere un punto y coma en los extremos de las sentencias o declaraciones, excepto cuando aparecen dos o más en la misma línea. En efecto, las nuevas líneas que siguen a ciertos **tokens** se convierten en un punto y coma, por lo que, en los saltos de línea, se coloca lo apropiado código **Go**. Por ejemplo, la llave de apertura **{** de la función debe estar en la misma línea que el final de la declaración **func**, no en una línea por sí mismo, y en la expresión **x + y** , se permite una nueva línea después, pero no antes del operador **+**.

Go toma una postura firme sobre el formato de código. La herramienta **gofmt** re-escribe el código en el formato estándar, y los subcomandos de herramientas **fmt** aplica **gofmt** a todos los archivos en el paquete especificado, o los que están en el directorio actual por defecto. Todos los archivos fuente **Go** en el libro se han ejecutado a través de **gofmt**, y se debe coger en el hábito de hacer lo mismo con el propio código. La declaración de un formato estándar por decreto elimina un gran debate acerca trivial inútil y, más importante, permite una variedad de transformaciones de código fuente automatizadas que serían inviables si se permitiera el formato arbitrario.

Muchos editores de texto pueden ser configurados para funcionar con **gofmt** cada vez que se guarda un archivo, de modo que su código fuente siempre tenga el formato correcto. Una herramienta relacionada, **goimports**, además, maneja la inserción y extracción de las declaraciones de importación, según sea necesario. No es parte de la distribución estándar, pero se puede obtener con este comando:

```
$ go get golang.org/x/tools/cmd/goimports
```

Para la mayoría de los usuarios, la manera habitual para descargar y generar paquetes, ejecutar sus pruebas, mostrar su documentación, etc., es con la herramienta **go**, que veremos en la **Sección 10.7**.

1.2. Argumentos de línea de comandos

La mayoría de los programas procesan alguna entrada para producir una salida; esto es más o menos la definición de la computación. Pero, ¿cómo obtiene un programa datos de entrada con los que operar? Algunos programas generan sus propios datos, pero normalmente, la entrada proviene de una fuente externa: un archivo, una conexión de red, la salida de otro programa, un usuario en un teclado, los argumentos de línea de comandos, o similares. Los próximos ejemplos tratarán algunas de estas alternativas, a partir de argumentos de línea de comandos.

El paquete **os** proporciona funciones y otros valores para tratar con el sistema operativo de una forma independiente de la plataforma. Los argumentos de línea de comandos están disponibles para un programa en una variable denominada **Args** que es parte del paquete **os**; de ahí su nombre en cualquier lugar fuera del paquete **os** es **os.Args**.

Las variables **os.Args** es un **slice** de **strings**. Los **slices** son un concepto fundamental en **Go**, y hablaremos mucho más sobre ellos pronto. Por ahora, pensar en un **slice** como una secuencia **s** de tamaño dinámico de elementos de arreglo donde los elementos individuales pueden ser accedidos como **s[i]** y una sub-secuencia contigua como **s[m:n]**. El número de elementos está dado por **len(s)**. Al igual que en la mayoría de los otros lenguajes de programación, toda la indexación en **Go** utiliza intervalos semiabiertos que incluyen el primer índice, pero excluyen el último, ya que simplifica la lógica. Por ejemplo, el **slice s[m:n]**, donde **0 ≤ m ≤ n ≤ len (s)**, contiene **n-m** elementos.

El primer elemento de **os.Args**, **os.Args[0]**, es el nombre del comando en sí; los otros elementos son los argumentos que se presentan en el programa cuando se inicia la ejecución. Una expresión **slice** de la forma **s[m:n]** da un **slice** que se refiere a los elementos **m** a través de n-1, por lo que los elementos que necesitamos para nuestro siguiente ejemplo son aquellos en el **slice os.Args[1:len(os.Args)]**. Si se omiten **m** o **n**, el valor predeterminado es **0** ó **len(s)**, respectivamente, por lo que se puede abreviar el **slice** deseado como **os.Args [1:]**.

Esto es una implementación del comando **echo** de **Unix**, que imprime sus argumentos de línea de comandos en una sola línea. Importa dos paquetes, que se dan como una lista entre paréntesis y no como declaraciones **import**

individuales. Es legal, pero se utiliza convencionalmente la forma de lista. El orden de las importaciones no importa; la herramienta **gofmt** ordena los nombres de los paquetes en orden alfabético. (Cuando hay varias versiones de un ejemplo, los numeraremos para que pueda estar seguro de que estamos hablando.)

```
gopl.io/ch1/echo1
// Echo1 Imprime los argumentos de línea de comandos.
package main

import (
    "fmt"
    "os"
)

func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

Los comentarios comienzan con **//**. Todo el texto de un **//** hasta el final de la línea es un comentario para los programadores y es ignorado por el compilador. Por convención, se describe cada paquete en un comentario inmediatamente anterior a su declaración del paquete; para un paquete **main**, este comentario es una o varias frases completas que describen el programa en su conjunto.

La declaración **var** declara dos variables **s** y **sep**, de tipo **string**. Una variable puede ser inicializada como parte de su declaración. Si no se ha inicializado de forma explícita, se inicializa implícitamente al valor cero de su tipo, que es **0** para tipos numéricos y el **string** vacío **""** para los **strings**. Así, en este ejemplo, la declaración implícita inicializa **s** y **sep** con strings vacíos. Veremos más sobre las variables y las declaraciones en el **Capítulo 2**.

Para los números, **Go** proporciona la aritmética usual y operadores lógicos. Sin embargo, cuando se aplica a los **strings**, el operador **+** concatena los valores, por lo que la expresión

```
sep + os.Args[i]
```

representa la concatenación de las cadenas **sep** y **os.Args[i]** . La sentencia que se utilizó en el programa,

```
s += sep + os.Args[i]
```

es una instrucción de asignación que concatena el antiguo valor de **s** con **sep** y **os.Args[i]** y lo asigna de nuevo a **s**; es equivalente a

```
s=s+sep + os.Args[i]
```

El operador **+=** es un operador de asignación. Cada operador aritmético y el lógico como **+** o ***** tiene un operador de asignación correspondiente.

El programa **echo** podría haber impreso su salida en un bucle de una sola pieza a la vez, pero en su lugar esta versión acumula una cadena añadiendo repetidamente nuevo texto hasta el final. El **string s** comienza la vida vacío, es decir, con el valor **""**, y en cada viaje a través del bucle le añade un poco de texto; después de la primera iteración, también se inserta un espacio de modo que cuando se termine el bucle, exista un espacio entre cada argumento. Este es un proceso cuadrático que podría ser costoso si el número de argumentos es grande, pero para **echo**, es poco probable. Mostraremos una serie de versiones mejoradas de **echo** en este capítulo y el siguiente va a hacer frente a cualquier ineficiencia real.

La variable **i** índice del bucle se declara en la primera parte del bucle **for**. El símbolo **:=** es parte de una declaración de variables corta, una sentencia que declara una o más variables y les da los tipos adecuados en función de los valores de la inicialización; veremos más sobre esto en el siguiente capítulo.

La sentencia de incremento **i++** añade **1** a **i**; es equivalente a **i+=1**, que es a su vez equivalente a **i = i + 1**. Hay una sentencia de decremento correspondiente **i--** que resta **1** a **i**. Estos son sentencias, no expresiones como son en la mayoría de los lenguajes de la familia **C**, ya que **j = i++** es ilegal, y son solo **postfix**, por lo que **--i** tampoco es legal.

El bucle **for** es la única sentencia de bucle en **Go**. Tiene una serie de formas, una de las cuales se ilustra aquí:

```
for initialization; condition; post {
    // cero o más sentencias
}
```

Los paréntesis no se usan nunca en torno a los tres componentes de un bucle **for**. Sin embargo, las llaves son obligatorias, y la llave de apertura debe estar en la misma línea que la sentencia **post**.

La sentencia opcional **initialization** se ejecuta antes de que empiece el bucle. Si está presente, debe ser una simple sentencia, es decir, una corta declaración de variable, una sentencia de incremento o asignación, o una llamada a función. **condition** es una expresión booleana que se evalúa al comienzo de cada iteración del bucle; si se evalúa como **true**, las sentencias controladas por el bucle se ejecutan. La sentencia **post** se ejecuta después del cuerpo del bucle, entonces la condición se evalúa de nuevo. El bucle termina cuando la condición se convierte en falsa.

Cualquiera de estas partes puede ser omitida. Si no hay **initialization** y no hay **post**, el punto y coma también pueden omitirse:

```
// Un tradicional bucle "while"
for condition {
    // ...
}
```

Si se omite **condition** completamente en cualquiera de estas formas, por ejemplo:

```
// Un bucle infinito tradicional
for {
    // ...
}
```

El bucle es infinito, aunque los bucles de esta forma se pueden terminar de alguna otra manera, con una sentencia **break** o **return**.

Otra forma de bucle *for* itera sobre un rango de valores de un tipo de datos como un **string** o un **slice**. Para ilustrar, aquí ponemos una segunda versión de **echo**:

```
gopl.io/ch1/echo2
// Echo2 Imprime los argumentos de línea de comandos.
package main

import (
    "fmt"
    "os"
)

func main() {
    s, sep := "", ""
    for _, arg := range os.Args[1:] {
        s += sep + arg
        sep = " "
    }
    fmt.Println(s)
}
```

En cada iteración del bucle, **range** produce un par de valores: el índice y el valor del elemento en ese índice. En este ejemplo, no necesitamos el índice, pero la sintaxis de un bucle **range** requiere que, si tratamos con el elemento, debemos tratar con el índice también. Una idea sería asignar el índice a una variable temporal, obviamente, como **temp** y pasar por alto su valor, pero **Go** no permite variables locales no utilizadas, por lo que esto daría lugar a un error de compilación.

La solución es utilizar el identificador en blanco (**blank identifier)**, cuyo nombre es **_** (es decir, un carácter de subrayado). El identificador en blanco puede ser utilizado siempre que la sintaxis requiera un nombre de variable, pero la lógica del programa no la requiera, por ejemplo, para descartar un índice de bucle no deseado cuando se requiere sólo el valor del elemento. La mayoría de los programadores de **Go** probablemente usarían **range** y **_** para escribir el programa echo anterior, ya que la indexación sobre **os.Args** es implícita, no explícita, y por lo tanto más fácil de hacerlo bien.

Esta versión del programa utiliza una breve declaración de variable para declarar e inicializar **s** y **sep**, pero podría igualmente haber declarado las variables por separado. Hay varias formas de declarar una variable de **string**; estos son todos equivalentes:

```
s := ""
var s string
var s = ""
var s string = ""
```

¿Por qué se debe preferir una forma a otra? La primera forma, una declaración de variable corta es la más compacta, pero sólo podrá utilizarse dentro de una función, no para las variables a nivel de paquete. La segunda forma se basa en la inicialización por defecto al valor cero para **strings**, que es **""**. La tercera forma se utiliza muy poco, excepto cuando se declara múltiples variables. La cuarta forma es explícita sobre el tipo de la variable **s**, que es redundante cuando es la misma que la del valor inicial pero necesaria en casos en los que no son del mismo tipo. En la práctica, se debe utilizar por lo general una de las dos primeras formas, con la inicialización explícita para decir que el valor inicial es importante y la inicialización implícita para decir que el valor inicial no importa.

Como se señaló anteriormente, cada vez que rueda el bucle, el **string s** toma nuevos contenidos. La sentencia **+=** hace un nuevo **string** mediante la concatenación del **string** anterior, un carácter de espacio, y el siguiente argumento, a continuación, se asigna el nuevo **string s**. El antiguo contenido de **s** ya no está en uso, por lo que será recogido por el recogedor de basura en su debido momento.

Si la cantidad de datos en cuestión es grande, esto podría ser costoso. Una solución más simple y más eficiente sería utilizar la función **Join** del paquete **strings**:

```
gopl.io/ch1/echo3
func main() {
    fmt.Println(strings.Join(os.Args[1:], " "))
}
```

Por último, si no nos importa el por formato y solo queremos ver los valores, tal vez para la depuración, podemos formatear los resultados en **Println**:

```
fmt.Println(os.Args[1:])
```

El resultado de esta sentencia es como lo que se pueden conseguir a partir **strings.Join**, pero con las llaves alrededor. Cualquier **slice** se puede imprimir esta manera.

1.3. Búsqueda de líneas duplicadas

Los programas para la copia de archivos, impresión, búsqueda, ordenación, conteo todos tienen una estructura similar: un bucle sobre la entrada, algún cálculo sobre cada elemento, y la generación de salida sobre la marcha o al final. Mostraremos tres variantes de un programa llamado **dup**; inspirado en parte por el comando **uniq** de **Unix**, que presenta las líneas duplicadas adyacentes. Las estructuras y paquetes utilizados son los modelos que se pueden adaptar fácilmente.

La primera versión de **dup** imprime cada línea que aparece más de una vez en la entrada estándar, precedido por su cuenta. Este programa presenta la sentencia **if**, el tipo de datos **map** y el paquete **bufio**.

gopl.io/ch1/dup1

```
gopl.io/ch1/dup1
// Dup1 imprime el texto de cada línea que aparece más de una vez en
// la entrada estándar, precedido por su cuenta.
package main

import (
    "bufio"
    "fmt"
     "os"
)

func main() {
    counts := make(map[string]int)
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        counts[input.Text()]++
    }
    // NOTA: haciendo caso omiso de los posibles errores de input.Err()
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

Al igual que con **for**, nunca se usan paréntesis alrededor de la condición en una sentencia **if**, pero se requieren llaves para el cuerpo. Puede haber una parte opcional **else** que se ejecuta si la condición es **false**.

Un **map** contiene un conjunto de pares **clave/valor** y proporciona operaciones constantes en el tiempo para almacenar, recuperar, o probar un elemento en el conjunto. La clave puede ser de cualquier tipo cuyos valores puedan compararse con **==**, siendo los **strings** el ejemplo más común; el valor puede ser de cualquier tipo. En este ejemplo, las claves son **string** y los valores son **int**. La función incorporada **make** crea un nuevo **map** vacío; tiene otros usos también. Los **maps** se tratan en detalle en la **Sección 4.3**.

Cada vez **dup** lee una línea de entrada, la línea se utiliza como una clave en el **map** y se incrementa el valor correspondiente. La sentencia **counts[input.Text()]++** es equivalente a estas dos sentencias:

```
line := input.Text()
counts[line] = counts[line] + 1
```

No es un problema si el **map** todavía no contiene esa clave. La primera vez que se ve una nueva línea, la expresión **counts[line]** en el lado derecho se evalúa al valor cero para su tipo, que es **0** para **int**.

Para imprimir los resultados, utilizamos otro bucle basado en **range**, esta vez sobre el **map counts**. Como antes, cada iteración produce dos resultados, una clave y el valor del elemento de **map** para esa clave. El orden de la iteración de **map** no se especifica, pero en la práctica es aleatorio, variando de una ejecución a otra. Este diseño es intencional, ya que evita que los programas confíen en cualquier orden particular donde no se garantiza ninguno.

El paquete **bufio**, ayuda a que la entrada y la salida sea eficiente y conveniente. Una de sus características más útiles es un tipo llamado **Scanner** que lee la entrada y la rompe en líneas o palabras; es a menudo la manera más fácil de procesar la entrada que viene de forma natural en las líneas.

El programa utiliza una declaración breve de variables para crear una nueva variable **input** que hace referencia a un **bufio.Scanner**:

```
input := bufio.NewScanner(os.Stdin)
```

El escáner lee desde la entrada estándar del programa. Cada llamada a **input.Scan()** lee la siguiente línea y elimina el carácter de nueva línea del final; el resultado puede ser recuperado llamando a **input.Text()**. La función **Scan** devuelve **true** si hay una línea y **false** cuando no hay más entradas.

La función **fmt.Printf**, como **printf** en **C** y otros lenguajes, produce una salida con formato de una lista de expresiones. Su primer argumento es un **string** formateado que especifica cómo deben ser formateados los argumentos posteriores. El formato de cada argumento es determinado por un carácter de conversión, una letra después de un signo de porcentaje. Por ejemplo, **%d** formatea un operando entero usando la notación decimal y **%s** expande el valor de un operando **string**.

Printf tiene más de una docena de tales conversiones, que los programadores de **Go** llaman **verbos**. Esta tabla está lejos de ser una especificación completa, pero ilustra muchas de las características que están disponibles:

%d	Entero decimal
%x, **%d**, **%b**	Enteros en hexadecimal, octal, binario
%f, **%g**, **%e**	Número de coma flotante: 3.141593 3.141592653589793 3.141593e + 00
%t	Booleano: **true** o **false**
%c	**Rune** (punto de código Unicode)
%s	**String**
%q	**String** con comillas "abc" o **rune** 'c'
%v	Cualquier valor en un formato natural
%T	Tipo de cualquier valor
%%	Signo de porcentaje literal (sin operando)

El **string** de formato en **dup1** también contiene una tabulación **\t** y un salto de línea **\n**. Los literales de **string** pueden contener secuencias de escape para la representación de caracteres que no son visibles. **Printf** no escribe una nueva línea de forma predeterminada. Por convención, las funciones de formato cuyos nombres terminan en **f**, como **log.Printf** y **fmt.Errorf**, utilizan las reglas de formato de **fmt.Printf**, mientras que aquellos cuyos nombres terminan en **ln** como **Println**, formatean sus argumentos como por **%v**, seguido de una nueva línea.

Muchos programas leen ya sea desde su entrada estándar, como anteriormente, o de una secuencia de archivos con nombre. La próxima versión del **dup** puede leer desde la entrada estándar o manejar una lista de nombres de archivo, utilizando **os.Open** para abrir cada uno de ellos:

```
gopl.io/ch1/dup2
// Dup2 imprime el recuento y el texto de las líneas que aparecen más de una vez
// en la entrada. Se lee de la entrada estándar o de una lista de archivos con el nombre.
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    counts := make(map[string]int)
    files := os.Args[1:]
    if len(files) == 0 {
        countLines(os.Stdin, counts)
    } else {
        for _, arg := range files {
            f, err := os.Open(arg)
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            countLines(f, counts)
            f.Close()
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
```

```
}

func countLines(f *os.File, counts map[string]int) {
    input := bufio.NewScanner(f)
    for input.Scan() {
        counts[input.Text()]++
    }
    // NOTA: haciendo caso omiso de los posibles errores de input.Err()
}
```

La función **os.Open** devuelve dos valores. El primero es un archivo abierto (***os.File**) que se utiliza en la posterior lee por el **Scanner**.

El segundo resultado de **os.Open** es un valor incorporado del tipo **error**. Si **err** es igual al valor incorporado especial **nil**, el archivo se abre correctamente. El archivo se lee, y cuando se llega al final de la entrada, **Close** cierra el archivo y libera cualquier recurso. Por otro lado, si se **err** no es **nil**, algo salió mal. En ese caso, el valor del error describe el problema. Nuestro simple manejador de errores imprime un mensaje en el flujo de error estándar utilizando **Fprintf** y el verbo **%v**, que muestra un valor de cualquier tipo en un formato predeterminado, y **dup** luego continúa con el siguiente archivo; la sentencia **continue** va a la siguiente iteración del bucle **for**.

En aras de mantener ejemplos de código a un tamaño razonable, nuestros primeros ejemplos son intencionalmente un tanto atrevidos sobre la gestión de errores. Es evidente que hay que comprobar si hay un error de **os.Open**; Sin embargo, estamos ignorando la probable posibilidad de que podría producirse un error al leer el archivo con **input.Scan**. Veremos lugares en los que se han omitido la comprobación de errores, entraremos en los detalles de tratamiento de errores en la **Sección 5.4**.

Observar que la llamada a **countlines** precede a su declaración. Las funciones y otras entidades a nivel de paquete se pueden declarar en cualquier orden.

Un **map** es una referencia a la estructura de datos creada por **make**. Cuando se pasa un **map** a una función, la función recibe una copia de la referencia, por lo que cualquier cambio en la función llamada en la estructura de datos subyacente será visible a través de la referencia de **map** también. En nuestro ejemplo, los valores insertados en el **map counts** por **countLines** son vistos por **main**.

Las versiones de **dup** anteriores operan en un modo "**streaming**" en el que se lee la entrada y se separa en líneas lo que sea necesario, por lo que, en principio, estos programas pueden manejar una cantidad arbitraria de entrada. Un enfoque alternativo es leer la entrada completa en la memoria de una sola vez, dividirla en líneas a la vez, y luego procesar las líneas. La siguiente versión, **dup3**, funciona de esa manera. Se introduce la función **ReadFile** (del paquete **io/ioutil**), que lee todo el contenido de un archivo con el nombre y **strings.Split**, que divide un **string** en un **slice** de **substrings**. (**Split** es lo contrario de **strings.Join**, que hemos visto anteriormente.)

Hemos simplificado **dup3** un poco. En primer lugar, sólo lee los archivos con nombre, no la entrada estándar, ya que **ReadFile** requiere un argumento de nombre de archivo. En segundo lugar, se pasó el recuento de las líneas de nuevo en **main**, ya que ahora se necesita en un solo lugar.

```
gopl.io/ch1/dup3
package main

import (
    "fmt"
    "io/ioutil"
    "os"
    "strings"
)

func main() {
    counts := make(map[string]int)
    for _, filename := range os.Args[1:] {
        data, err := ioutil.ReadFile(filename)
        if err != nil {
            fmt.Fprintf(os.Stderr, "dup3: %v\n", err)
            continue
        }
        for _, line := range strings.Split(string(data), "\n") {
            counts[line]++
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

ReadFile devuelve un **slice** de **byte** que debe ser convertido en un **string** de modo que se pueda dividir por **strings.Split**. Vamos a tratar los **strings** y los **slices** de bytes en detalle en la **Sección 3.5.4**.

Debajo de las sábanas, **bufio.Scanner**, **ioutil.ReadFile**, y **ioutil.WriteFile** utilizan los métodos **Read** y **Write** de ***os.File**, pero es raro que la mayoría de los programadores necesiten acceder directamente a esas rutinas de bajo nivel. Las funciones de nivel superior como las de **bufio** e **io/ioutil** son más fáciles de usar.

1.4. GIF animados

El siguiente programa demuestra el uso básico de los paquetes de imagen estándar de **Go**, que usaremos para crear una secuencia de imágenes de mapa de bits y luego codificar la secuencia como una animación **GIF**. Las imágenes, llamadas figuras de **Lissajous**, eran un efecto visual básico en las películas de ciencia ficción de la década de **1960**. Son las curvas paramétricas producidas por oscilación armónica en dos dimensiones, tales como dos ondas sinusoidales alimentada en las entradas **x** e **y** de un osciloscopio. La **Figura 1.1** muestra algunos ejemplos.

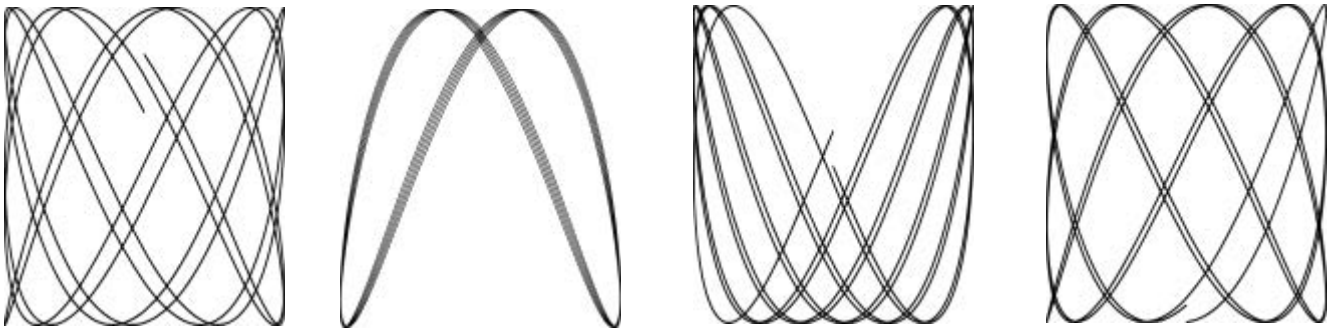

Figura 1.1. Cuatro figuras de **Lissajous**.

Hay muchas nuevas construcciones en este código, incluyendo declaraciones **const**, tipos **struct**, y literales compuestos. A diferencia de la mayoría de nuestros ejemplos, éste también implica cálculos de coma flotante. Aquí hablaremos de estos temas sólo brevemente aquí, dejando la mayoría de los detalles para los capítulos posteriores, ya que el objetivo principal en este momento es tener una idea del aspecto de **Go** y el tipo de cosas que se pueden hacer fácilmente con el lenguaje y sus librerías.

```
gopl.io/ch1/lissajous
    // Lissajous genera animaciones GIF de figuras de Lissajous al azar.
package main

import (
    "image"
    "image/color"
    "image/gif"
    "io"
    "math"
    "math/rand"
    "os"
)

var palette = []color.Color{color.White, color.Black}
const (
        whiteIndex = 0 // primer color en la paleta
        blackIndex = 1 // siguiente color en la paleta
)
func main() {
    lissajous(os.Stdout)
}
func lissajous(out io.Writer) {
    const (
            cycles = 5   // número de revoluciones del oscilador completa x
            res = 0.001  // resolución angular
            size = 100   // canvas de la imagen cubre [-size..+size]
            nframes = 64 // número de frames de animación
            delay = 8    // retraso entre frames en unidades de 10 ms
    )
    freq := rand.Float64() * 3.0 // frecuencia relativa del oscilador y
    anim := gif.GIF{LoopCount: nframes}
    phase := 0.0 // diferencia de fase
    for i := 0; i < nframes; i++ {
        rect := image.Rect(0, 0, 2*size+1, 2*size+1)
        img := image.NewPaletted(rect, palette)
        for t := 0.0; t < cycles*2*math.Pi; t += res {
```

```
            x := math.Sin(t)
            y := math.Sin(t*freq + phase)
            img.SetColorIndex(size+int(x*size+0.5), size+int(y*size+0.5),
                blackIndex)
        }
        phase += 0.1
        anim.Delay = append(anim.Delay, delay)
        anim.Image = append(anim.Image, img)
    }
    gif.EncodeAll(out, &anim) // NOTA: ignorar errores de codificación
}
```

Después de importar un paquete cuyo camino tiene varios componentes, como **image/color**, nos referimos al paquete con un nombre que viene del último componente. Por lo tanto la variable **color.White** pertenece al paquete **image/color** y **gif.GIF** pertenece a **image/gif** .

Una declaración **const** (**§3.6**) da nombres a las constantes, es decir, valores que se fijan en tiempo de compilación, como los parámetros numéricos de los ciclos, los **frames**, y el retardo. Tal como las declaraciones **var**, las declaraciones **const** pueden aparecer a nivel de paquete (por lo que los nombres son visibles en todo el paquete) o dentro de una función (por lo que los nombres son visibles sólo dentro de esa función). El valor de una constante debe ser un número, un **string** o un booleano.

Las expresiones **[]color.Color{...}** y **gif.GIF{...}** son literales compuestos (**§4.2**, **§4.4.1**), una notación compacta para crear instancias de tipos compuestos de **Go** de una secuencia de valores de elementos. Aquí, el primero es un **slice** y el segundo es un **struct**.

El tipo **gif.GIF** es un tipo de estructura (**§4.4**). Una estructura es un conjunto de valores denominados campos, a menudo de diferentes tipos, que se recogen en un único objeto que puede ser entendido como una unidad. La variable **anim** es una estructura de tipo **gif.GIF**. El literal **struct** crea un valor **struct** cuyo campo **LoopCount** se establece en **nframes**; todos los demás campos tienen el valor cero para su tipo. Los campos individuales de una estructura se pueden acceder mediante la notación de punto, al igual que en las dos últimas asignaciones que actualizan de forma explícita los campos **Delay** e **Image** de **anim** .

La función **lissajous** tiene dos bucles anidados. El bucle exterior tiene ejecuta **64** iteraciones, cada una produciendo un solo **frame** de la animación. Se crea una nueva imagen **201 X 201** con una paleta de dos colores, blanco y negro. Todos los píxeles se ajustan inicialmente al valor cero de la paleta (el color en la paleta de cero), que será blanco. Cada paso a través del bucle interno genera una nueva imagen mediante el establecimiento de algunos píxeles negros. El resultado se anexa, utilizando la función incorporada **append** (§4.2.1), a una lista de fotogramas de **anim**, junto con un retraso especificado de **80 ms**. Por último, la secuencia de **frames** y los retrasos se codifican en formato **GIF** y se escriben en el **stream** de salida **out** . El tipo de **out** es **io.Writer**, lo que nos permite escribir a una amplia gama de posibles destinos, como veremos pronto.

El bucle interior ejecuta los dos osciladores. El oscilador **x** es sólo la función seno. El oscilador **y** también es una sinusoide, pero su frecuencia relativa al oscilador **x** es un número aleatorio entre **0** y **3**, y su fase relativa al oscilador **x** es inicialmente cero, pero aumenta con cada fotograma de la animación. El bucle se ejecuta hasta que el oscilador **x** ha completado cinco ciclos completos. En cada paso, se llama a **SetColorIndex** para colorear el píxel correspondiente a negro (**x, y**), que está en la posición **1** de la paleta.

La función **main** llama a la función **lissajous**, dirigiéndola a escribir en la salida estándar, por lo que este comando produce un **GIF** animado con frames como los de la **Figura 1.1**:

```
$ go build gopl.io/ch1/lissajous
$ ./lissajous >out.gif
```

1.5. Fetching una URL

Para muchas aplicaciones, el acceso a la información a través de Internet es tan importante como el acceso al sistema de archivos local. **Go** ofrece una colección de paquetes, agrupados en **net**, que hacen que sea enviar y recibir información a través de Internet, realizar conexiones de red de bajo nivel, y configurar servidores, para lo cual las características de concurrencia **Go** son particularmente útiles (introducidos en el **Capítulo 8**).

Para ilustrar el mínimo necesario para recuperar información a través de **HTTP**, aquí hay un simple programa llamado **fetch** que recupera el contenido de cada **URL** y lo imprime como texto no interpretado; está inspirado por la utilidad invaluable **curl**. Obviamente por lo general se hace más con este tipo de datos, pero esto muestra la idea básica. Vamos a utilizar este programa con frecuencia en el libro.

```
gopl.io/ch1/fetch
// Fetch imprime el contenido que se encuentra en una URL.

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
)
```

```
func main() {
    for _, url := range os.Args[1:] {
        resp, err := http.Get(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
            os.Exit(1)
        }
        b, err := ioutil.ReadAll(resp.Body)
        resp.Body.Close()
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: reading %s: %v\n", url, err)
            os.Exit(1)
        }
        fmt.Printf("%s", b)
    }
}
```

Este programa introduce funciones a partir de dos paquetes, **net/http** y **io/ioutil**. La función **http.Get** realiza una petición **HTTP** y, si no hay error, devuelve el resultado en la estructura de respuesta **resp** . El campo **Body** de **resp** contiene la respuesta del servidor como un **stream** legible. A continuación, **ioutil.ReadAll** lee toda la respuesta; el resultado se almacena en **b** . El **stream Body** es cerrado para evitar fugas de recursos, y **Printf** escribe la respuesta en la salida estándar.

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://gopl.io
<html>
<head>
<title>The Go Programming Language</title>
...
```

Si la petición **HTTP** falla, **fetch** informa dwl fallo:

```
$ ./fetch http://bad.gopl.io
fetch: Get http://bad.gopl.io: dial tcp: lookup bad.gopl.io: no such host
```

En cualquiera de los casos de error, **os.Exit(1)** hace que el proceso salga con un código de estado de 1.

1.6. Fetching URL Concurrentemente

Uno de los aspectos más interesantes y novedosas de **Go** es su soporte para la programación concurrente. Este es un tema muy amplio, al que se dedican los **Capítulos 8** y **9**, así que por ahora sólo daremos una muestra de principales los mecanismos de concurrencia de **Go**, **goroutines** y canales.

El próximo programa, **fetchall**, hace la mismo captura de contenidos de una URL del ejemplo anterior, pero captura muchas **URL**, todo concurrentemente, por lo que el proceso durará lo que dure el fetch más largo no la suma de todas las capturas. Esta versión de **fetchall** descarta las respuestas, pero indica el tamaño y el tiempo transcurrido para cada una:

```
gopl.io/ch1/fetchall
package main

// FetchAll obtiene las direcciones URL de forma paralela e informa de sus tiempos y tamaños.

import (
    "fmt"
    "io"
    "io/ioutil"
    "net/http"
    "os"
    "time"
)

func main() {
    start := time.Now()
    ch := make(chan string)
    for _, url := range os.Args[1:] {
        go fetch(url, ch) // iniciar una goroutine
    }
    for range os.Args[1:] {
        fmt.Println(<-ch) // recibir en el canal ch
    }
    fmt.Printf("%.2fs elapsed\n", time.Since(start).Seconds())
}

func fetch(url string, ch chan<- string) {
```

```
    start := time.Now()
    resp, err := http.Get(url)
    if err != nil {
        ch <- fmt.Sprint(err) // enviar al canal ch
        return
    }
    nbytes, err := io.Copy(ioutil.Discard, resp.Body)
    resp.Body.Close() // No tener alguna fuga de recursos
    if err != nil {
        ch <- fmt.Sprintf("while reading %s: %v", url, err)
        return
    }
    secs := time.Since(start).Seconds()
    ch <- fmt.Sprintf("%.2fs %7d %s", secs, nbytes, url)
}
```

Aquí hay un ejemplo:

```
$ go build gopl.io/ch1/fetchall
$ ./fetchall https://golang.org http://gopl.io https://godoc.org
0.14s 6852 https://godoc.org
0.16s 7261 https://golang.org
0.48s 2475 http://gopl.io
0.48s elapsed
```

Una **goroutine** es una función de ejecución concurrente. Un **canal** es un mecanismo de comunicación que permite a una **goroutine** pasar valores de un especificado tipo a otra **goroutine**. La función **main** se ejecuta en una **goroutine** y la sentencia **go** crea **goroutines** adicionales.

La función **main** crea un canal de **strings** por medio de **make**. Para cada argumento de línea de comandos, la sentencia **go** en el primer bucle **range** comienza una nueva **goroutine** que llama a **fetch** asíncronamente para poder recuperar la **URL** usando **http.Get**. La función **io.Copy** lee el cuerpo de la respuesta y la descarta escribiéndola en el **stream** de salida **ioutil.Discard**. **Copy** devuelve el número de bytes, junto con cualquier error que se produzca. A medida que llega cada resultado, **fetch** envía una línea de resumen en el canal **ch**. El segundo bucle de **range** en **main** recibe e imprime estas líneas.

Cuando una **goroutine** intenta un envío o recepción en un canal, se bloquea hasta que otra **goroutine** intenta la operación de recibir o enviar correspondiente, momento en el que el valor se transfiere y ambas **goroutines** proceden. En este ejemplo, cada **fetch** envía un valor (**ch <- expresión**) en el canal **ch4** , y **main** recibe todos ellos (**<-ch**). Teniendo a **main** haciendo toda la impresión asegura que la salida de cada **goroutine** se procesa como una unidad, sin peligro de entrelazado si dos **goroutines** terminan al mismo tiempo.

1.7 Un servidor Web

Las librerías de **Go** hacen que sea fácil escribir un servidor web que responda a las peticiones de los clientes, como las realizadas por **fetch**. En esta sección, mostraremos un servidor mínimo que devuelve el componente de la ruta de la **URL** que se utiliza para acceder al servidor. Es decir, si la solicitud es para **http://localhost:8000/hello**, la respuesta será **URL.Path = "/hello"**.

```
gopl.io/ch1/server1
// Server1 es un mínimo servidor "eco"

package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", handler) // cada solicitud llama al manejador
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

// manejador que hace eco del componente del trazado de la solicitud de URL r.
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}
```

El programa es sólo de unas pocas líneas, las funciones de biblioteca hacen la mayor parte del trabajo. La función **main** conecta con una función de manejo de direcciones **URL** entrantes que comienzan con **/**, que son todas las direcciones **URL**, e inicia un servidor escuchando peticiones entrantes en el puerto **8000**. La solicitud es representada como una estructura de tipo **http.Request**, que contiene una serie de campos relacionados, uno de los cuales es la

dirección **URL** de la petición entrante. Cuando llega una petición, se le da a la función de manejo, que extrae el componente de trazado (**/hello**) de la **URL** de la solicitud y la envía como respuesta, utilizando **fmt.Fprintf**. Los servidores web se explicarán en detalle en la **Sección 7.7**.

Vamos a iniciar el servidor en segundo plano. En **Mac OS X** o **Linux**, agregar un signo and (&) a la orden; en **Microsoft Windows**, habrá que ejecutar el comando sin el símbolo en una ventana de comandos independiente.

```
$ go run src/gopl.io/ch1/server1/main.go &
```

Luego podemos hacer las solicitudes de cliente de la línea de comandos:

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://localhost:8000
URL.Path = "/"
$ ./fetch http://localhost:8000/help
URL.Path = "/help"
```

Alternativamente, se puede acceder al servidor desde un navegador web, como se muestra en la **Figura 1.2**.

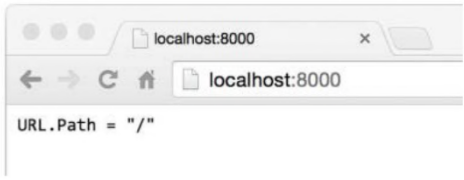

Figura 1.2. Una respuesta desde el servidor de eco.

Es fácil añadir características al servidor. Una adición útil es una **URL** específica que devuelve un estado de algún tipo. Por ejemplo, esta versión hace el mismo eco, pero también cuenta el número de peticiones; una petición a la **URL /count** devuelve el recuento hasta ahora, con exclusión de las que **/count** solicita a sí mismo:

```
gopl.io/ch1/server2
// Servidor 2 es un servidor mínimo de "eco" y contador.
package main

import (
    "fmt"
    "log"
    "net/http"
    "sync"
)

var mu sync.Mutex
var count int

func main() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/count", counter)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

// Manejador que hace eco de la componente de la ruta de la dirección URL solicitada.
func handler(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    count++
    mu.Unlock()
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}

// Contador que hace eco de la cantidad de llamadas hasta el momento.
func counter(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    fmt.Fprintf(w, "Count %d\n", count)
    mu.Unlock()
}
```

El servidor tiene dos manejadores, y la **URL** de la solicitud determina cuál se llama: una solicitud de **/count** invoca **counter** y todas las demás invocan a **handler**. Un modelo de manejador que termina en una barra coincide con cualquier **URL** que tenga el patrón como prefijo. Detrás de las escenas, el servidor ejecuta el manejador para cada petición entrante en una **goroutine** por separado para que pueda servir a múltiples peticiones simultáneamente. Sin embargo, si dos solicitudes simultáneas intentan actualizar **count** al mismo tiempo, puede que no se incremenea consistentemente; El programa tendría un error grave conocido como una condición de carrera (**race condition**) **(§9.1**). Para evitar este problema, hay que asegurarse de que a lo sumo accede una **goroutine** a la variable a la vez,

lo cual es el propósito de las llamadas a **mu.Lock() y mu.Unlock()** que coloca cada acceso de **count**. Veremos más de cerca la concurrencia con variables compartidas en el **Capítulo 9**.

Como un ejemplo más rico, la función **handler** puede informar sobre los datos de los encabezados y formar los datos que recibe, haciendo al servidor útil para inspeccionar y depurar peticiones:

```
gopl.io/ch1/server3

// Manipulador que hace eco de la petición HTTP.
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "%s %s %s\n", r.Method, r.URL, r.Proto)
    for k, v := range r.Header {
        fmt.Fprintf(w, "Header[%q] = %q\n", k, v)
    }
    fmt.Fprintf(w, "Host = %q\n", r.Host)
    fmt.Fprintf(w, "RemoteAddr = %q\n", r.RemoteAddr)
    if err := r.ParseForm(); err != nil {
        log.Print(err)
    }
    for k, v := range r.Form {
        fmt.Fprintf(w, "Form[%q] = %q\n", k, v)
    }
}
```

Esto utiliza los campos de la estructura **http.Request** para producir una salida como ésta:

```
GET /?q=query HTTP/1.1
Header["Accept-Encoding"] = ["gzip, deflate, sdch"]
Header["Accept-Language"] = ["en-US,en;q=0.8"]
Header["Connection"] = ["keep-alive"]
Header["Accept"] = ["text/html,application/xhtml+xml,application/xml;..."]
Header["User-Agent"] = ["Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5)..."]
Host = "localhost:8000"
RemoteAddr = "127.0.0.1:59911"
Form["q"] = ["query"]
```

Observar cómo la llamada a **ParseForm** está anidada dentro de una sentencia **if**. **Go** permite una sentencia simple, como una declaración de variable local para preceder a la condición **if**, lo que es particularmente útil para el tratamiento de errores de este ejemplo. Lo podríamos haber escrito así

```
err := r.ParseForm()
if err != nil {
    log.Print(err)
}
```

pero la combinación de las sentencias es más corta y reduce el alcance de la variable **err**, que es una buena práctica. Definiremos el alcance en la **Sección 2.7**.

En estos programas, hemos visto tres tipos muy diferentes utilizados como flujos de salida (**output streams**). El programa **fetch** copió los datos de respuesta **HTTP** a **os.Stdout**, un archivo, al igual que en el programa **lissajous**. El programa **fetchall** lanzó la respuesta lejos (mientras contaba su longitud) copiándola en el trivial **ioutil.Discard**. Y el servidor web utilizado anteriormente usó **fmt.Fprintf** para escribir a un **http.ResponseWriter** que representa el navegador web.

Aunque estos tres tipos difieren en los detalles de cómo lo hacen, todos ellos satisfacen una **interface** común, permitiendo que cualquiera de ellos pueda ser utilizado cuando sea necesario crear un flujo de salida. Esa interface llamada **io.Writer**, se trata en la **Sección 7.1**.

El mecanismo de interface de **Go** es el tema del **Capítulo 7**, pero para dar una idea de lo que es capaz de hacer, veamos lo fácil que es combinar el servidor web con la función **lissajous** de manera que los archivos **GIF** animados se escriban no en la salida estándar, sino al cliente **HTTP**. Sólo hay que añadir estas líneas al servidor web:

```
handler := func(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
}
http.HandleFunc("/", handler)
```

o de manera equivalente:

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
})
```

El segundo argumento de la llamada a la función **HandleFunc** es un literal de función, es decir, una función anónima definida en el punto de uso. Lo explicaremos más a fondo en la **Sección 5.6**.

Una vez hecho este cambio, visitar **http://localhost:8000** en el navegador. Cada vez que se carga la página, se verá una nueva animación como la de la **Figura 1.3**.

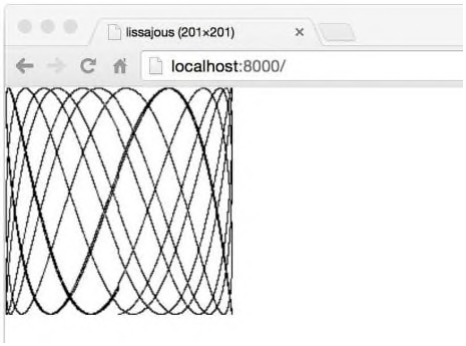

Figura 1.3. las figuras de **Lissajous** animadas en un navegador.

1.8. Cabos sueltos

Hay mucho más de **Go** que lo que hemos tratado en esta introducción rápida. Estos son algunos de los temas que hemos tocado ligeramente u omitido por completo, con el tratamiento suficiente serán familiares cuando haya breves apariciones antes del tratamiento completo.

Control de flujo: Hemos tratado las dos sentencias de flujo de control fundamentales, **if** y **for**, pero no la sentencia **switch**, que es una rama de múltiples vías. Aquí vemos un pequeño ejemplo:

```
switch coinflip() {
case "heads":
    heads++
case "tails":
    tails++
default:
    fmt.Println("landed on edge!")
}
```

El resultado de la llamada a **coinflip** se compara con el valor de cada **case**. Los casos se evalúan de arriba a abajo, por lo que la primera coincidencia se ejecuta. El caso opcional **default** coincide si ninguno de los otros casos lo hace; se puede colocar en cualquier lugar. Los casos no caen a través de uno a otro, como en lenguajes como **C** (aunque existe la sentencia **fallthrough** que es poco usada que anula este comportamiento).

Un **switch** no necesita un operando; sólo se puede enumerar los casos, cada una de las cuales es una expresión booleana:

```
func Signum(x int) int {
    switch {
    case x > 0:
        return +1
    default:
        return 0
    case x < 0:
        return -1
    }
}
```

Esta forma se llama un interruptor sin etiqueta (**tagless switch**); que es equivalente a un **switch true**.

Al igual que las sentencias **for** y **if**, un **switch** puede incluir una sentencia simple opcional, una corta declaración de variable, un incremento o sentencia de asignación, o una llamada de función, que puede ser utilizada para establecer un valor antes de que se ponga a prueba.

Las sentencias **break** y **continue** modifican el control del flujo. Un **break** causa que el control se reanude en la siguiente sentencia después de las sentencias **for**, **switch** o **select** (que veremos más adelante), y como vimos en la **Sección 1.3**, un **continue** hace que el bucle **for** más interior inicie su siguiente iteración. Las sentencias pueden ser etiquetadas de manera que **break** y **continue** puedan referirse a ellas, por ejemplo, para salir de varios bucles anidados a la vez o para iniciar la siguiente iteración del bucle más externo. Hay incluso una sentencia **goto**, a pesar de que está destinada al código generado por la máquina, no para el uso regular de los programadores.

Tipos con nombre: Una declaración de tipo hace posible dar un nombre a un tipo existente. Como los tipos **struct** son a menudo largos, casi siempre son nombrados. Un ejemplo conocido es la definición de un tipo **Point** para un sistema de gráficos **2-D**:

```
type Point struct {
    X, Y int
}
var p Point
```

Las declaraciones de tipos y tipos con nombre se describen en el **Capítulo 2**.

Punteros: **Go** proporciona punteros, es decir, valores que contienen la dirección de una variable. En algunos lenguajes, en particular **C**, los punteros son sin restricciones relativamente. En otros lenguajes, los punteros se disfrazan de "referencias'' y no se pueda hacer mucho con ellos, excepto pasarlos. **Go** toma una posición en algún punto intermedio. Los punteros son visibles de forma explícita. El operador **&** proporciona la dirección de una variable y el operador ***** recupera la variable a la que el puntero se refiere, pero no hay aritmética de punteros. Explicaremos los punteros en la **Sección 2.3.2**.

Métodos e interfaces: Un método es una función asociada con un tipo con nombre; **Go** es inusual en que los métodos pueden ser conectado a casi cualquier tipo con nombre. Los métodos se describen en el **Capítulo 6**. Las interfaces son tipos abstractos que nos permiten tratar diferentes tipos concretos de la misma manera basándose en lo que los métodos tienen, no la forma en que están representados o implementados. Las interfaces son el tema del **Capítulo 7**.

Paquetes: **Go** viene con una extensa librería estándar de útiles paquetes, y la comunidad **Go** ha creado y compartido muchos más. La programación es a menudo más el uso de paquetes existentes que la escritura de código original uno mismo. A lo largo del libro, vamos a señalar un par de docenas de los paquetes estándar más importantes, pero hay muchos más que no tenemos espacio para mencionar, y no podemos ofrecer algo remotamente parecido a una referencia completa para cualquier paquete.

Antes de embarcarse en cualquier programa nuevo, es una buena idea ver si ya existen paquetes que nos pueden ayudar a hacer el trabajo más fácilmente. Se puede encontrar un índice de los paquetes de librerias estándar en **https://golang.org/pkg** y los paquetes aportados por la comunidad en **https://godoc.org**. La herramienta **go doc** hace que estos documentos sean accesibles fácilmente desde la línea de comandos:

```
$ go doc http.ListenAndServe
package http // import "net/http"
func ListenAndServe(addr string, handler Handler) error
    ListenAndServe listens on the TCP network address addr and then
    calls Serve with handler to handle requests on incoming connections.
...
```

Comentarios: Ya hemos mencionado los comentarios de la documentación en el principio de un programa o paquete. Es también un buen estilo escribir un comentario antes de la declaración de cada función para especificar su comportamiento. Estas convenciones son importantes, ya que son utilizadas por herramientas como **go doc** y **godoc** para localizar y presentar la documentación (**§10.7.4**).

Para comentarios que abarcan varias líneas o aparecen dentro de una expresión o sentencia, también existe la conocida notación de otros lenguajes **/ * ... * /**. Este tipo de comentarios se utilizan a veces en el comienzo de un archivo con un gran bloque de texto explicativo para evitar un **//** en cada línea. En un comentario, **//** y **/*** no tienen ningún significado especial, por lo que los comentarios no se anidan.