Capítulo 3

Tipos de datos básicos

Con bits en la parte inferior, por supuesto, pero las computadoras operan fundamentalmente con números de tamaño fijo llamados palabras, que se interpretan como números enteros, números de coma flotante, conjuntos de bits o direcciones de memoria, luego se combinan para formar agregados más grandes que representan paquetes, píxeles, carteras, poesía, y todo lo demás. **Go** ofrece una variedad de maneras de organizar los datos, con un espectro de tipos de datos que en un extremo coinciden con las características del hardware y en el otro extremo proporcionan lo que los programadores necesitan para representar convenientemente complicadas estructuras de datos.

Los tipos de **Go** se dividen en cuatro categorías: **tipos básicos**, **tipos de agregados**, **tipos de referencia**, y **tipos de interface**. Los tipos básicos, el tema de este **Capítulo**, incluyen números, **strings** y valores booleanos. Los tipos de agregados, arreglos (**§4.1**) y estructuras (**§4.4**), forman tipos de datos más complejos mediante la combinación de varios valores de otros más simples. Los tipos de referencia son un grupo diverso que incluye punteros (**§2.3.2**), **slices** (**§4.2**), mapas (**§4.3**), funciones (**Capítulo 5**), y canales (**Capítulo 8**), pero lo que tienen en común es que todos se refieren a variables o estado del programa indirectamente, de modo que el efecto de una operación que se aplica a una referencia se observa por todas las copias de esa referencia. Finalmente, hablaremos de tipos de interface en el **Capítulo 7**.

3.1. Enteros

Los tipos de datos numéricos de **Go** incluyen varios tamaños de números enteros, números de coma flotante, y números complejos. Cada tipo numérico determina el tamaño y el signo de sus valores. Vamos a comenzar con los números enteros.

Go proporciona la aritmética de enteros con y sin signo. Hay cuatro tamaños distintos de enteros con signo, de **8**, **16**, **32** y **64** bits, representados por los tipos **int8**, **int16**, **int32** y **int64**, y las versiones sin signo correspondiente **uint8**, **uint16**, **uint32** y **uint64**.

También hay dos tipos llamados simplemente **int** y **uint** que son del tamaño natural o más eficiente para los números enteros con y sin signo en una plataforma en particular; **int** es de lejos el tipo numérico más utilizado. Ambos tipos tienen el mismo tamaño, de **32** ó **64** bits, pero no se deben hacer suposiciones acerca de cuál; diferentes compiladores pueden tomar decisiones diferentes, incluso en hardware idéntico.

El tipo de **rune** es un sinónimo de **int32** y convencionalmente indica que es un valor de un punto de código **Unicode**. Los dos nombres se pueden usar indistintamente. Del mismo modo, el tipo **byte** es un sinónimo de **uint8**, y hace hincapié en que el valor es una pieza de datos en bruto más que una pequeña cantidad numérica.

Por último, existe un tipo entero sin signo **uintptr**, cuya anchura no se ha especificado, pero es suficiente para contener todos los bits de un valor de puntero. El tipo **uintptr** se utiliza sólo para la programación de bajo nivel, como por ejemplo en el límite de un programa **Go** con una librería de **C** o un sistema operativo. Veremos ejemplos de esto cuando nos ocupemos del paquete **unsafe** en el **Capítulo 13**.

Independientemente de su tamaño, **int**, **uint** y **uintptr** son diferentes tipos de sus hermanos de tamaño de forma explícita. Por lo tanto, **int** no es del mismo tipo que **int32**, aunque el tamaño natural de los números enteros es de **32 bits**, y se requiere una conversión explícita para utilizar un valor **int** donde es necesario un **int32**, y viceversa.

Los números con signo se representan en la forma de complemento a 2, en el que el bit de orden superior está reservado para el signo del número y el rango de valores de un número de n bits es el de -2^{n-1} a $2^{n-1} - 1$. Los enteros sin signo utilizan la gama completa de bits para los valores no negativos y por lo tanto tienen el rango de **0** a $2^n - 1$. Por ejemplo, el rango de **int8** es de **-128** a **-127**, mientras que el rango de **uint8** es de **0** a **255**.

Los operadores binarios de **Go** para la aritmética, la lógica y la comparación se enumeran aquí en orden decreciente de prioridad:

```
*       /       %       <<      >>      y       & ^
+       ¬       |       ^
==      !=      <       <=      >       >=
&&
||
```

Sólo hay cinco niveles de prioridad de los operadores binarios. Los operadores en el mismo nivel se asocian a la izquierda, lo que hará necesario paréntesis para mayor claridad, para hacer que los operadores se evalúen en el orden deseado en una expresión como **mask &** (**1 << 28**).

Cada operador en las dos primeras líneas de la tabla de arriba, por ejemplo **+**, tiene un correspondiente operador de asignación como **+=** que puede ser utilizado para abreviar una sentencia de asignación.

Los operadores aritméticos de número entero **+**, **-**, ***** y **/** se pueden aplicar a números enteros, coma flotante, y complejos, pero el operador resto **%** sólo se aplica a los números enteros. El comportamiento de **%** para los números

negativos varía a través de los lenguajes de programación. En **Go**, el signo del resto es siempre el mismo que el signo del dividendo, por lo **-5%3** y **-5%-3** son ambos **-2**. El comportamiento de **/** depende de si sus operandos son enteros, por lo que **5.0/4.0** es **1.25**, pero **5/4** es **1** debido a que la división entera trunca el resultado a cero.

Si el resultado de una operación aritmética, ya sea con o sin signo, tiene más bits que pueden ser representados en el tipo del resultado, se llama **overflow**. Los bits de orden superior que no encajan son descartados silenciosamente. Si el número original es un tipo con signo, el resultado podría ser negativo si el bit más a la izquierda es un **1**, como en el **int8** de ejemplo aquí:

```
var u uint8 = 255
fmt.Println(u, u+1, u*u) // "255 0 1"
var i int8 = 127
fmt.Println(i, i+1, i*i) // "127 -128 1"
```

Se pueden comparar dos números enteros del mismo tipo utilizando los operadores de comparación binarios siguientes; el tipo de una expresión de comparación es un valor booleano.

```
==      Igual a
!=      No igual a
<       Menos de
<=      Menos de o igual a
>       Mayor que
>=      Mayor que o igual a
```

De hecho, todos los valores de tipo básico, booleanos, números y **strings**, son comparables, lo que significa que dos valores del mismo tipo pueden ser comparados con los operadores **==** y **!=**. Por otra parte, los números enteros, números de coma flotante y **strings** están ordenados por los operadores de comparación. Los valores de muchos otros tipos no son comparables, y no hay otros tipos ordenados. Presentamos las normas que rigen la comparabilidad de sus valores.

Hay también operadores unitarios de suma y resta:

```
+       Positivo unario (sin efecto)
-       negación unaria
```

Para enteros, **+x** es una abreviatura de **0+x** y **-x** es una abreviatura de **0-x**; para los números de coma flotante y complejos, **+x** es **x** y **-x** es la negación de **x**.

Go también proporciona los siguientes operadores binarios bit a bit, los primeros cuatro tratan a sus operandos como patrones de bits con ningún concepto de aritmética de acarreo o signo:

```
Y       AND bit a bit
|       OR bit a bit
^       XOR bit a bit
&^      borrado de bit (Y NOT)
<<      Desplazamiento a la izquierda
>>      Desplazamiento a la derecha
```

El operador **^** es **OR exclusiva** bit a bit (**XOR**) cuando se utiliza como un operador binario, pero cuando se utiliza como un operador de prefijo unario es negación bit a bit o complemento; es decir, se devuelve un valor de cada bit en su operando invertido. El operador **&^** es poco claro (Y **NOT**): en la expresión **z = x &^ y**, cada bit de **z** es **0** si el bit correspondiente de **y** es **1**; de lo contrario, es igual al bit correspondiente de **x**.

El código siguiente muestra cómo se pueden utilizar las operaciones a nivel de bit para interpretar un valor **uint8** como un conjunto compacto y eficiente de **8 bits** independientes. Utiliza verbos **Printf %b** para imprimir un número de dígitos binarios; **08** modifica **%b** (¡un adverbio!) Para rellenar el resultado con ceros a exactamente **8** dígitos.

```
var x uint8 = 1<<1 | 1<<5
var y uint8 = 1<<1 | 1<<2

fmt.Printf("%08b\n", x) // "00100010", el conjunto {1, 5}
fmt.Printf("%08b\n", y) // "00000110", el conjunto {1, 2}

fmt.Printf("%08b\n", x&y) // "00000010", la intersección {1}
fmt.Printf("%08b\n", x|y) // "00100110", la unión {1, 2, 5}
fmt.Printf("%08b\n", x^y) // "00100100", la diferencia simétrica {2, 5}
fmt.Printf("%08b\n", x&^y) // "00100000", la diferencia {5}

for i := uint(0); i < 8; i++ {
    if x&(1<<i) != 0 { // prueba de afiliación de miembro
        fmt.Println(i) // "1", "5"
    }
}

fmt.Printf("%08b\n", x<<1) // "01000100", el conjunto {2, 6}
fmt.Printf("%08b\n", x>>1) // "00010001", el conjunto {0, 4}
```

(La **Sección 6.5** muestra una implementación de conjuntos de números enteros que pueden ser mucho más grandes que un byte).

En las operaciones de desplazamiento **x<<n** y **x>>n**, el operando **n** determina el número de posiciones de **bit** a cambiar y debe ser sin signo; el operando **x** puede ser sin signo o con signo. Aritméticamente, un desplazamiento a la izquierda **x<<n** es equivalente a la multiplicación por 2^n y un desplazamiento a la derecha **x>>n** es equivalente a la división por 2^n.

Los desplazamientos a la izquierda llenan los bits vacantes con ceros, al igual que los desplazamientos a la derecha de números sin signo, pero los desplazamientos a la derecha de números con signo llenan los bits vacantes con copias del bit de signo. Por esta razón, es importante usar la aritmética sin signo cuando se trata de un número entero como un patrón de bits.

Aunque **Go** proporciona números sin signo y aritmética, se tiende a utilizar la forma con signo **int** incluso para cantidades que pueden no ser negativas, tales como la longitud de un arreglo, aunque **uint** puede parecer una elección más obvia. De hecho, la función incorporada **len** devuelve un **int** con signo, como en este bucle que anuncia medallas de premios en el orden inverso:

```
medals := []string{"gold", "silver", "bronze"}
for i := len(medals) - 1; i >= 0; i-- {
    fmt.Println(medals[i]) // "bronze", "silver", "gold"
}
```

La alternativa sería calamitosa. Si **len** devuelve un número sin signo, entonces **i** también debería ser un **uint**, y la condición **i>=0** siempre sería verdad por definición. Después de la tercera iteración, en la que **i == 0**, la sentencia **i--** causaría que **i** convertirse no en **-1**, pero el máximo valor **uint** (por ejemplo, 2^{64}**-1**), y la evaluación de **medals[i]** iba a fallar en tiempo de ejecución, o de **panic** (**§5.9**), al tratar de acceder a un elemento fuera de los límites del **slice**.

Por esta razón, los números sin signo tienden a ser utilizados sólo cuando se requieren sus operadores bit a bit cuando son requeridos operadores aritméticos, como en la implementación de conjuntos de bits, análisis de archivos de formatos binarios, o para **hashing** y criptografía. Por lo general no se utilizan para cantidades no negativas.

En general, se requiere una conversión explícita para convertir un valor de un tipo a otro, y los operadores binarios para la aritmética y la lógica (excepto **shifts**) deben tener operandos del mismo tipo. Aunque en ocasiones esto se traduce en expresiones más largas, también elimina toda una clase de problemas y hace que los programas sean más fáciles de entender.

A modo de ejemplo familiar de otros contextos, considerar la siguiente secuencia:

```
var apples int32 = 1
var oranges int16 = 2
var compote int = apples + oranges // Error de compilación
```

Intentar compilar estas tres declaraciones produce un mensaje de error:

```
invalid operation: apples + oranges (mismatched types int32 and int16)
```

Esta falta de coincidencia de tipo puede ser arreglada de varias maneras, directamente mediante la conversión de todo a un tipo común:

```
var compote = int(apples) + int(oranges)
```

Tal como se describe en la **Sección 2.5**, para cada tipo **T**, la operación de conversión **T(x)** convierte el valor de **x** al tipo **T** si se permite la conversión. Muchas conversiones de enteros a enteros no implican un cambio en el valor; simplemente le indican al compilador cómo interpretar un valor. Pero una conversión que se estreche un gran número entero en uno más pequeño, o una conversión de número entero a coma flotante o viceversa, puede cambiar el valor o perder precisión:

```
f := 3.141 // un float64
i := int(f)
fmt.Println(f, i) // "3.141 3"
f=1.99
fmt.Println(int(f)) // "1"
```

La conversión de un **float** a un entero descarta cualquier parte fraccionaria, truncando hacia cero. Se deben evitar las conversiones en las que el operando está fuera de rango para el tipo de destino, debido a que el comportamiento depende de la implementación:

```
f := 1e100 // un float64
i := int(f) // resultado es dependiente de la implementación
```

Los literales enteros de cualquier tamaño y tipo se pueden escribir como números decimales ordinarios, o como números octales si empiezan con **0**, como en **0666**, o como hexadecimales si comienzan con **0x** o **0X**, como en **0xdeadbeef**. Los dígitos hexadecimales pueden ser mayúsculas o minúsculas. Hoy en día los números octales son utilizados para exactamente un propósito, los permisos de archivos en sistemas **POSIX,** pero los números hexadecimales son ampliamente utilizados para enfatizar el patrón de bits de un número sobre su valor numérico.

Al imprimir los números utilizando el paquete **fmt**, podemos controlar el radical o el formato con los verbos **%d**, **%o** y **%x**, como se muestra en este ejemplo:

```
o := 0666
fmt.Printf("%d %[1]o %#[1]o\n", o) // "438 666 0666"
x := int64(0xdeadbeef)
fmt.Printf("%d %[1]x %#[1]x %#[1]X\n", x)
// Output:
// 3735928559 deadbeef 0xdeadbeef 0XDEADBEEF
```

Observar el uso de dos **fmt**. Por lo general, un **string** de formato **Printf** que contiene múltiples verbos **%** requerirá el mismo número de operandos adicionales, pero los "adverbios" **[1]** después **%** le dice a **Printf** que debe utilizar el primer operando una y otra vez. En segundo lugar, el adverbio **#** para **%s** o **%x** o **%X** le dicen a **Printf** que emita un prefijo **0** o **0x** o **0X** respectivamente.

Los literales de la runa (**rune)** se escriben como un carácter entre comillas simples. El ejemplo más simple es un carácter ASCII como **'a'**, pero ' es posible escribir cualquier punto de código **Unicode** directamente o con escapes numéricos, como veremos en breve.

Las runas se imprimen con **%c**, o con **%q** si se desean las comillas:

```
ascii := 'a'
unicode := 'D'
newline := '\n'
fmt.Printf("%d %[1]c %[1]q\n", ascii) // "97 a 'a'"
fmt.Printf("%d %[1]c %[1]q\n", unicode) // "22269 D 'D'"
fmt.Printf("%d %[1]q\n", newline) // "10 '\n'"
```

3.2. Números de coma flotante

Go ofrece dos tamaños de números de coma flotante, **float32** y **float64**. Sus propiedades aritméticas se rigen por el estándar **IEEE 754** implementado por todas las **CPU** modernas.

Los valores de estos tipos numéricos se extienden de pequeño a grande. Los límites de los valores de coma flotante se pueden encontrar en el paquete **math**. La constante **math.MaxFloat32**, el mayor **float32**, es alrededor de **3.4e38** , y **math.MaxFloat64** es alrededor **1.8e308** . Los valores positivos son más pequeños cerca de **1.4e-45** y **4.9e-324**, respectivamente.

Un **float32** proporciona aproximadamente seis dígitos decimales de precisión, mientras que un **float64** proporciona alrededor de **15** dígitos; debe preferirse el **float64** para la mayoría de propósitos porque los cálculos con **float32** acumulan errores rápidamente a menos que se tenga cuidado, y el número entero positivo más pequeño que no se puede representar exactamente como **float32** no muy es grande:

```
var f float32 = 16777216 // 1 << 24
fmt.Println(f == f+1) // "true"!
```

Los números de coma flotante se pueden escribir literalmente usando decimales, por ejemplo:

```
const e = 2.71828 // (aproximadamente)
```

Los dígitos pueden omitirse antes del punto decimal (**.707**) o después de él (**1.**). Los números muy pequeños o muy grandes es mejor escribirlos en notación científica, con la letra **e** o **E** que precede al exponente decimal:

```
const Avogadro = 6.02214129e23
const Planck = 6.62606957e-34
```

Los valores de coma flotante son convenientemente impresos con el verbo **%g** de **Printf**, que elige la representación más compacta para una precisión adecuada, pero para las tablas de datos, las formas **%e** (exponente) o **%f** (sin exponente) pueden ser más apropiadas. Todos los tres verbos permiten anchura de campo y precisión numérica para ser controlada.

```
for x := 0; x < 8; x++ {
    fmt.Printf("x = %d e^x = %8.3f\n", x, math.Exp(float64(x)))
}
```

El código anterior imprime las potencias de **e** con tres cifras decimales de precisión, alineados en un campo de ocho caracteres:

```
x=0 e^x =    1.000
x=1 e^x =    2.718
x=2 e^x =    7.389
x=3 e^x =   20.086
x=4 e^x =   54.598
x=5 e^x =  148.413
x=6 e^x =  403.429
x=7 e^x = 1096.633
```

Además de una gran colección de funciones matemáticas habituales, el paquete **math** tiene funciones para crear y detectar los valores especiales definidos por **IEEE 754**: los infinitos positivos y negativos, que representan números de magnitud excesiva y el resultado de la división por cero; y **NaN** (**not a number** "no un número"), que es el resultado de operaciones de este tipo matemáticamente dudosas como **0/0** o **Sqrt(-1)**.

```
var z float64
fmt.Println(z, -z, 1/z, -1/z, z/z) // "0 -0 +Inf -Inf NaN"
```

La función **math.IsNaN** comprueba si su argumento es un valor no-un-número, y **math.NaN** devuelve un valor tal. Es tentador utilizar **NaN** como un valor centinela en un cálculo numérico, pero poner a prueba si un resultado de cálculo específico es igual a **NaN** está lleno de peligros, ya que cualquier comparación con **NaN** siempre produce **false**:

```
nan := math.NaN()
fmt.Println(nan == nan, nan < nan, nan > nan) // "false false false"
```

Si una función que devuelve un resultado de coma flotante puede fallar, es mejor reportar el fallo por separado, como esto:

```
func compute() (value float64, ok bool) {
    // ...
    if failed {
        return 0, false
    }
    return result, true
}
```

El siguiente programa ilustra el cálculo de gráficos de coma flotante. Traza una función de dos variables **z = f(x,y)** como una malla de alambre de superficie **3D**, usando gráficos vectoriales escalables (**SVG**), una notación estándar **XML** para dibujos lineales. La **Figura 3.1** muestra un ejemplo de la salida de la función **sin(r)/r**, donde **r** es **sqrt(x*x+y*y)**.

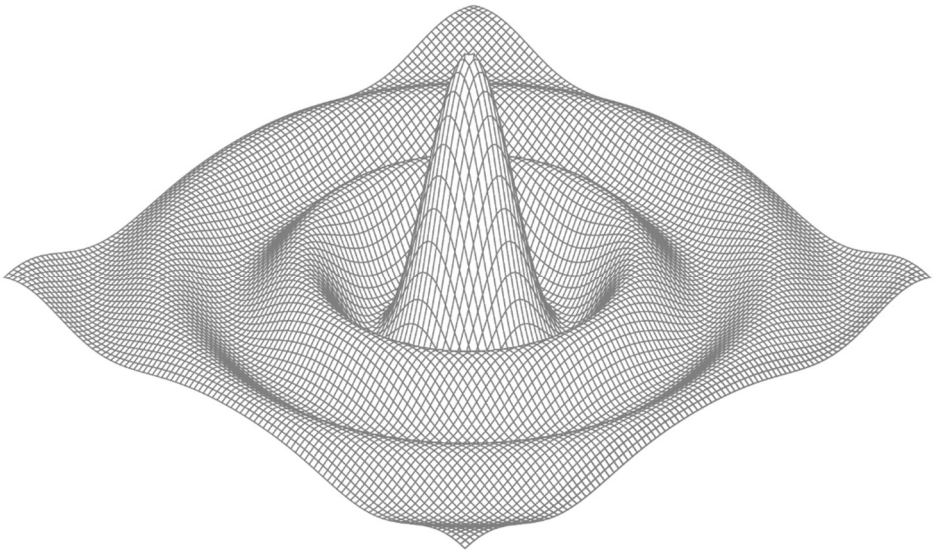

Figura 3.1. Un gráfico de superficie de la función **sin(r)/r**.

```
gopl.io/ch3/surface
// Surface calcula una representación SVG de una función de superficie 3D.
package main

import (
    "fmt"
    "math"
)

const (
    width, height = 600, 320 // tamaño del canvas en pixeles
    cells = 100 // numero de celdas de la malla
    xyrange = 30.0 // rango de ejes (-xyrange..+xyrange)
    xyscale = width / 2 / xyrange // pixeles por unidad x o y
    zscale = height * 0.4 // pixeles por unidad z
    angle = math.Pi / 6 // ángulo de los ejes x, y (=30°)
)
var sin30, cos30 = math.Sin(angle), math.Cos(angle) // sin(30°), cos(30°)

func main() {
    fmt.Printf("<svg xmlns='http://www.w3.org/2000/svg' "+
        "style='stroke: grey; fill: white; stroke-width: 0.7' "+
        "width='%d' height='%d'>", width, height)
    for i := 0; i < cells; i++ {
        for j := 0; j < cells; j++ {
            ax, ay := corner(i+1, j)
```

```
            bx, by := corner(i, j)
            cx, cy := corner(i, j+1)
            dx, dy := corner(i+1, j+1)
            fmt.Printf("<polygon points='%g,%g %g,%g %g,%g %g,%g'/>\n",
                ax, ay, bx, by, cx, cy, dx, dy)
        }
    }
    fmt.Println("</svg>")
}

func corner(i, j int) (float64, float64) {
    // Encontrar punto (x,y) en la esquina de celda (i,j).
    x := xyrange * (float64(i)/cells - 0.5)
    y := xyrange * (float64(j)/cells - 0.5)
    // Calcular la altura de la superficie z.
    z := f(x, y)
    // Projectar (x,y,z) isometricamente en un canvas 2-D SVG (sx,sy).
    sx := width/2 + (x-y)*cos30*xyscale
    sy := height/2 + (x+y)*sin30*xyscale - z*zscale
    return sx, sy
}
func f(x, y float64) float64 {
    r := math.Hypot(x, y) // distancia desde (0,0)
    return math.Sin(r) / r
}
```

Nótese que la función **corner** devuelve dos valores, las coordenadas de la esquina de la celda. La explicación de cómo funciona el programa requiere sólo la geometría básica, pero se puede saltar sobre ella, ya que el punto es para ilustrar el cálculo de coma flotante. La esencia del programa es el mapeo entre tres sistemas de coordenadas diferentes, que se muestran en la **Figura 3.2**. La primera es una cuadrícula de **2-D** de **100x100** células identificadas mediante coordenadas enteras **(i,j)**, a partir de **(0,0)** en la esquina trasera. Se traza desde la parte trasera a la parte delantera de modo que los polígonos de fondo pueden ser ocultados por otros de primer plano.

El segundo sistema de coordenadas es una malla de coordenadas **3-D** de coma flotante **(x,y,z)**, donde **x** e **y** son funciones lineales de **i** y **j**, trasladadas de manera que el origen está en el centro, y se escala por la constante **xyrange**. La altura **z** es el valor de la función de superficie **f(x,y)**.

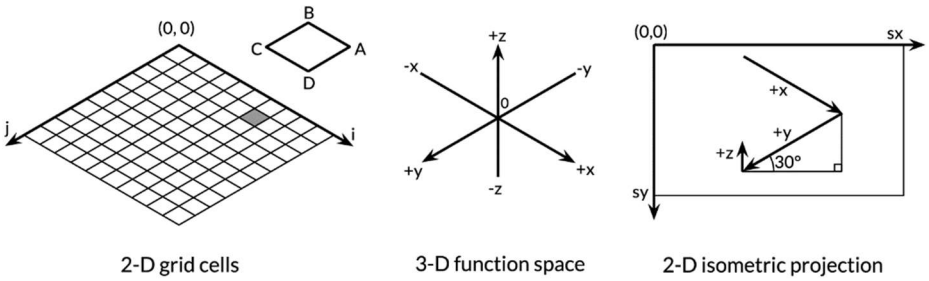

Figura 3.2. Tres sistemas de coordenadas diferentes.

El tercer sistema de coordenadas es el lienzo (**canvas)** de la imagen **2-D**, con **(0,0)** en la esquina superior izquierda. Los puntos en este plano se denotan como **(sx,sy)**. Utilizamos una proyección isométrica para mapear cada punto **3-D (x,y,z)** en el lienzo **2-D**. Cuando aparece un punto más a la derecha en el lienzo, mayor es su valor **x** o menor es su valor **y**. Y si un punto aparece más abajo en el lienzo, mayor será su valor **x** o valor **y**, y más pequeño su valor **z**. Los factores de escala horizontal o vertical para **x** e **y** se derivan del seno y el coseno de un ángulo de **30°**. El factor de escala para **z**, **0.4**, es un parámetro arbitrario.

Para cada celda de la cuadrícula **2-D**, la función **main** calcula las coordenadas en el lienzo de la imagen de las cuatro esquinas del polígono **ABCD**, donde **B** corresponde a **(i,j)** y **A**, **C** y **D** son sus vecinos, a continuación, imprime una instrucción **SVG** para dibujarlo.

3.3. Números complejos

Go ofrece dos tamaños para los números complejos, **complex64** y **complex128**, cuyos componentes son **float32** y **float64** respectivamente. La función incorporada **complex** crea un número complejo a partir de sus componentes real e imaginaria, y las funciones incorporadas en **real** e **imag** extraen los componentes:

```
var x complex128 = complex(1, 2) // 1+2i
var y complex128 = complex(3, 4) // 3+4i
fmt.Println(x*y) // "(-5+10i)"
fmt.Println(real(x*y)) // "-5"
fmt.Println(imag(x*y)) // "10"
```

Si un literal coma flotante o un literal entero decimal es seguido inmediatamente por **i**, como **3.141592i** o **2i**, se convierte en un literal imaginario, que denota un número complejo con un componente real cero:

```
fmt.Println(1i * 1i) // "(-1+0i)", i² = -1
```

Bajo las reglas de la aritmética constante, las constantes complejas se pueden añadir a otras constantes (entero o de coma flotante, real o imaginaria), lo que nos permite escribir números complejos de forma natural, como **1+2i**, o equivalentemente, **2i+1**. Las declaraciones de **x** y de **y** por encima se puede simplificar:

```
x := 1 + 2i
y := 3 + 4i
```

Los números complejos pueden ser comparados para igualdad con **==** y **!=**. Dos números complejos son iguales si sus partes reales son iguales y sus partes imaginarias son iguales.

El paquete **math/cmplx** proporciona funciones de librería para trabajar con números complejos, tales como las funciones complejas de raíces cuadradas y exponenciales.

```
fmt.Println(cmplx.Sqrt(-1)) // "(0+1i)"
```

El siguiente programa utiliza aritmética con **complex128** para generar un conjunto de **Mandelbrot**.

```
gopl.io/ch3/mandelbrot
// Mandelbrot emite una imagen PNG del fractal de Mandelbrot.
package main

import (
    "image"
    "image/color"
    "image/png"
    "math/cmplx"
    "os"
)

func main() {
    const (
        xmin, ymin, xmax, ymax = -2, -2, +2, +2
        width, height = 1024, 1024
    )

    img := image.NewRGBA(image.Rect(0, 0, width, height))
    for py := 0; py < height; py++ {
        y := float64(py)/height*(ymax-ymin) + ymin
        for px := 0; px < width; px++ {
            x := float64(px)/width*(xmax-xmin) + xmin
            z := complex(x, y)
            // Image point (px, py) represents complex value z.
            img.Set(px, py, mandelbrot(z))
        }
    }
    png.Encode(os.Stdout, img) // NOTE: ignoring errors
}

func mandelbrot(z complex128) color.Color {
    const iterations = 200
    const contrast = 15

    var v complex128
    for n := uint8(0); n < iterations; n++ {
        v=v*v + z
        if cmplx.Abs(v) > 2 {
            return color.Gray{255 - contrast*n}
        }
    }
    return color.Black
}
```

Los dos bucles anidados iteran sobre cada punto de una imagen de escala de grises **1024x1024** y que representa la parte de **-2** a **+2** del plano complejo. El programa comprueba repetidamente si al elevar al cuadrado y sumar el número que representa "se escapa" del círculo de radio **2**. Si es así, el punto es ensombrecido por el número de iteraciones que se tardó en escapar. Si no es así, el valor pertenece al conjunto de **Mandelbrot**, y sigue siendo un punto negro. Por último, el programa escribe en la salida estándar la imagen codificada **PNG** del icónico fractal, que se muestra en la **Figura 3.3**.

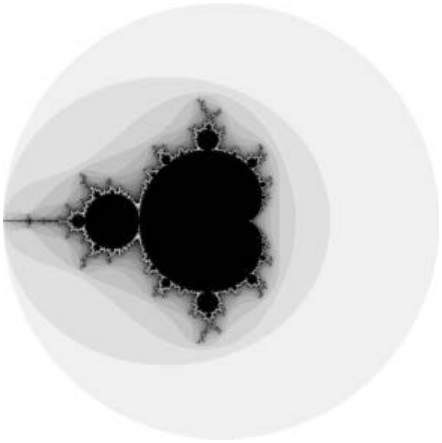

Figura 3.3. el conjunto de **Mandelbrot**.

3.4. Booleanos

Un valor de tipo **bool**, o booleano, tiene sólo dos valores posibles, **true** y **false**. Las condiciones en las sentencias **if** y **for** son booleanos y los operadores de comparación como **==** y **<** producen un resultado booleano. El operador unario **!** es la negación lógica, por lo que **!true** es **false**, o, por decirlo así, **(!true == false) == true** , aunque como una cuestión de estilo, siempre es conveniente simplificar las expresiones booleanas redundantes como **x == true** para **x**.

Los valores booleanos se pueden combinar con el **&& (AND)** y **|| (OR)**, que tienen operadores de comportamiento de cortocircuito: si la respuesta ya está determinada por el valor del operando de la izquierda, el operando de la derecha no se evalúa, lo que es seguro para escribir expresiones como ésta:

```
s != "" && s[0] == 'x'
```

donde **s[0]** entraría en pánico si se aplica a un string vacía.

Como **&&** tiene mayor prioridad que **||** (mnemotécnica: **&&** es la multiplicación booleana, **||** es la suma booleana), no se requieren paréntesis para las condiciones de esta forma:

```
if 'a' <= c && c <= 'z' ||
    'A' <= c && c <= 'Z' ||
    '0' <= c && c <= '9' {
    // ...ASCII letras o dígitos...
}
```

No hay una conversión implícita de un valor booleano a un valor numérico como **0** o **1**, o viceversa. Es necesario utilizar un **if** explícito, como en:

```
i := 0
if b {
    i=1
}
```

Puede ser que sea digno de escribir una función de conversión si se necesita esta operación a menudo:

```
// btoi devuelve 1 si b es verdadero y 0 si es falso.
func btoi(b bool) int {
    if b {
        return 1
    }
    return 0
}
```

La operación inversa es tan simple que no justifica una función, pero la simetría aquí está:

```
// Itob comprueba si i es diferente de cero.
func itob(i int) bool { return i != 0 }
```

3.5. Strings

Un **string** es una secuencia de bytes inmutable. Los **strings** pueden contener datos arbitrarios, incluyendo bytes con valor **0**, pero por lo general contienen texto legible. Los **strings** de texto se interpretan de forma convencional como secuencias codificadas en **UTF-8** de puntos de código Unicode (runas), que vamos a explorar en detalle más adelante.

La función incorporada **len** devuelve el número de **bytes** (no runas) en un **string**, y la operación de índice de **s[i]** recupera el **byte i**-ésimo del **string s**, donde **0 ≤ i < len(s)**.

```
s := "hello, world"
fmt.Println(len(s)) // "12"
fmt.Println(s[0], s[7]) // "104 119" ('h' and 'w')
```

Si se intenta acceder a un **byte** fuera de este rango resulta una situación de pánico:

```
c := s[len(s)] // pánico: Índice fuera de rango
```

El **byte i**-ésimo de un **string** no es necesariamente el **i**-ésimo carácter de un **string**, porque la codificación **UTF-8** de un punto de código no **ASCII** requiere dos o más **bytes**. En breve trataremos el trabajo con caracteres.

La operación de **substring s[i:j]** obtiene un nuevo **string** que consta de los bytes del **string** original comenzando en el índice **i** y continuando hasta el **byte** en el índice **j**, pero no incluyéndolo. El resultado contiene **j-i** bytes.

```
fmt.Println(s[0:5]) // "hello"
```

Una vez más, da como resultado un pánico si cualquier índice está fuera de los límites o si **j** es menor que **i**.

Se pueden omitir uno o ambos de los operandos **i** y **j**, en cuyo caso los valores por defecto de **0** (el principio del **string**) y **len(s)** (su extremo) se supone, respectivamente.

```
fmt.Println(s[:5]) // "hello"
fmt.Println(s[7:]) // "world"
fmt.Println(s[:]) // "hello, world"
```

El operador **+** realiza un nuevo **string** mediante la concatenación de dos **strings**:

```
fmt.Println("goodbye" + s[5:]) // "goodbye, world"
```

Los **strings** pueden ser comparados con los operadores de comparación como **==** y **<**; la comparación se realiza byte a byte, por lo que el resultado es el orden lexicográfico natural.

Los valores de **string** son inmutables: la secuencia de bytes contenidos en un valor de **string** no se puede cambiar, aunque, por supuesto, podemos asignar un nuevo valor a una variable **string**. Para añadir un **string** a otro, por ejemplo, podemos escribir:

```
s := "left foot"
t := s
s += ", right foot"
```

Esto no modifica el **string s** que se realizó originalmente, pero causa que **s** sostenga el nuevo **string** formado por la declaración **+=**; Mientras tanto, **t** todavía contiene el viejo **string**.

```
fmt.Println(s) // "left foot, right foot"
fmt.Println(t) // "left foot"
```

Dado que los **strings** son inmutables, las construcciones que tratan de modificar los datos un **string** no están permitidos:

```
s[0] = 'L' // Error de compilación: no se puede asignar a s[0]
```

Inmutabilidad significa que es seguro para dos copias de un **string** compartir la misma memoria subyacente, por lo que es barato para copiar secuencias de cualquier longitud. Del mismo modo, un **string s** y un **substring** como **s[7:]** pueden compartir de manera segura los mismos datos, por lo que la operación **substring** también es barata. No se asigna nueva memoria en cada caso. La **Figura 3.4** ilustra la disposición de un **string** y dos de sus **substrings** que comparten el mismo conjunto de bytes subyacente.

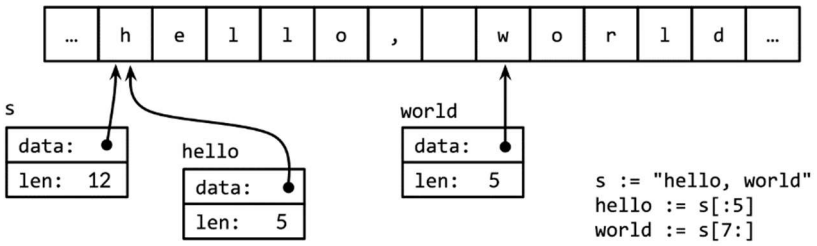

Figura 3.4. El string "hello, world" y dos substrings.

3.5.1. Los literales de String

Un valor de **string** se puede escribir como un literal de **string**, una secuencia de bytes entre comillas dobles:

```
"Hello,世界"
```

Dado que los archivos fuente de **Go** siempre están codificados en **UTF-8** y los **strings** de texto de **Go** se interpretan convencionalmente como **UTF-8**, podemos incluir los puntos de código **Unicode** en los literales de **string**.

Dentro de un literal de **string** entre comillas dobles, pueden ser utilizadas secuencias de escape que comienzan con una barra invertida **** para insertar valores arbitrarios de bytes en el **string**. Una serie de manejadores de escape **ASCII** controlan los códigos como nueva línea, retorno de carro y tabulación:

```
\a      "alerta" o campana
\b      retroceso
\f      avance de página
\n      salto de línea
\r      retorno de carro
\t      tabulación
\v      tabulación vertical
\'      Comilla simple (sólo en la literal runa ' \ ' ' )
\"      Comillas dobles (sólo dentro de literales " ... ")
\\      barra invertida
```

También pueden ser incluidos bytes arbitrarios en los literales de **strings** utilizando escapes hexadecimales u octales. Un escape hexadecimal se escribe **\xhh**, con exactamente dos dígitos hexadecimales **h** (en mayúsculas o minúsculas). Un escape octal se escribe **\ooo** con exactamente tres dígitos octales **o** (**0** a **7**) no superior a **\377**. Ambos denotan un solo byte con el valor especificado. Más tarde, veremos cómo codificar los puntos de código **Unicode** numéricamente en los literales de **string**.

Un literal **string** en bruto se escribe **`...`**, usando comillas inversas en lugar de comillas dobles. Dentro de un literal de **string**, no se procesan secuencias de escape; los contenidos se toman literalmente, incluyendo las barras invertidas y saltos de línea, por lo que un literal **string** en bruto puede extenderse en varias líneas en el código fuente del programa. El único procesamiento es que los retornos de carro se eliminan de forma que el valor del **string** es el mismo en todas las plataformas, incluyendo aquellas que, por convención, ponen retornos de carro en archivos de texto.

Los literales de **string** en bruto son una forma conveniente de escribir expresiones regulares, que tienden a tener un montón de barras invertidas. También son útiles para las plantillas **HTML**, literales **JSON**, mensajes de uso de comandos, y similares, que a menudo se extienden a lo largo de múltiples líneas.

```
const GoUsage = `Go is a tool for managing Go source code.

Usage:
go command [arguments]
...`
```

3.5.2. Unicode

Hace mucho tiempo, la vida era simple y había, por lo menos en una visión provinciana, sólo un conjunto de caracteres para hacer frente: **ASCII**, el Código Estándar Americano para Intercambio de Información. **ASCII**, o más precisamente **US-ASCII**, utiliza **7 bits** para representar **128** "caracteres": las letras mayúsculas y minúsculas del inglés, dígitos y una variedad de caracteres de puntuación y control de dispositivos. Durante gran parte de los primeros días de la informática, esto era adecuado, pero dejó una fracción muy grande de la población del mundo sin poder utilizar sus propios sistemas de escritura en los ordenadores. Con el crecimiento de Internet, los datos en miles de idiomas se han convertido en algo mucho más común. ¿Cómo puede ser tratada esta rica variedad y, si es posible, de manera eficiente?

La respuesta es **Unicode** (**unicode.org**), que recoge todos los caracteres en todos los sistemas de escritura del mundo, además de acentos y otros signos diacríticos, como códigos de control como la tabulación y el retorno de carro, y un montón de elementos esotéricos, y asigna a cada uno un número estándar llamado un punto de código **Unicode** o, en la terminología **Go**, una runa (**rune**).

Unicode versión **8** define los puntos de código para más de **120.000** caracteres en más de **100** lenguas y escrituras. ¿Cómo están representados los programas de ordenador y los datos? El tipo de datos naturales para mantener una sola runa es **int32**, y que es lo que usa **Go**; tiene el sinónimo runa precisamente para este propósito.

Podríamos representar una secuencia de runas como una secuencia de valores **Int32**. En esta representación, que se llama **UTF-32** o **UCS-4**, la codificación de cada punto de código **Unicode** tiene el mismo tamaño, de **32 bits**. Esto es simple y uniforme, pero utiliza mucho más espacio de lo necesario ya que la mayoría de texto legible por ordenador está en **ASCII**, lo que requiere sólo **8 bits** o **1 byte** por carácter. Todos los carácteres de uso generalizado son de una cifra inferior a **65.536**, lo que encajaría en **16 bits**. ¿Podemos hacerlo mejor?

3.5.3. UTF8

UTF-8 es una codificación de longitud variable de puntos de código Unicode como bytes. **UTF-8** fue inventado por **Ken Thompson** y **Rob Pike**, dos de los creadores de **Go**, y ahora es un estándar **Unicode**. Utiliza entre **1** y **4 bytes** para representar cada runa, pero sólo **1 byte** para los caracteres **ASCII**, y sólo **2** o **3 bytes** para la mayoría de las runas de uso común. Los bits de orden superior del primer byte de la codificación de una runa indican cuántos bytes siguen. Un orden **0** indica **ASCII** de **7 bits**, donde cada runa lleva sólo **1 byte**, por lo que es idéntico al **ASCII** convencional. Un orden superior **110** indica que la runa lleva **2 bytes**; el segundo byte comienza con **10**. Las runas más grandes tienen codificaciones análogas.

```
0xxxxxx                             runas 0-127       (ASCII)
110xxxx 10xxxxxx                    128-2047          (valores <128 no utilizados)
1110xxx 10xxxxxx 10xxxxxx           2048-65535        (valores <2048 no utilizados)
11110xx 10xxxxxx 10xxxxxx 10xxxxxx  65536-0x10ffff    (otros valores no utilizados)
```

Una codificación de longitud variable se opone a la indexación directa para acceder al carácter **n**-ésimo de un **string**, pero **UTF-8** tiene muchas propiedades deseables para compensar. La codificación es compacta, compatible con **ASCII**, y con auto-sincronización: es posible encontrar el comienzo de un carácter de una copia de no más de tres bytes. Es también un código de prefijo, por lo que puede ser decodificado de izquierda a derecha sin ningún tipo de ambigüedad o adivinanza. La codificación de una runa es una **substring** de cualquier otra, o incluso de una secuencia de otros, por lo que se puede buscar una runa sólo buscando sus bytes, sin preocuparse por el contexto precedente. El orden de bytes lexicográfico es igual al orden de punto de código de **Unicode**, por lo que la clasificación **UTF-8** funciona de forma natural. No hay incrustados **NUL** (cero) bytes, que es conveniente para los lenguajes de programación que utilizan **NUL** para interrumpir los **strings**.

Los ficheros fuente de **Go** siempre están codificados en **UTF-8** y **UTF-8** es la codificación preferida para los **strings** de texto manipulados por los programas **Go**. El paquete **unicode** proporciona funciones para trabajar con runas individuales (como distinguir las letras de los números, o convertir una letra mayúscula a una minúscula), y el paquete **unicode/utf8** proporciona funciones para la codificación y decodificación de runas como bytes utilizando **UTF-8**.

Muchos caracteres Unicode son difíciles de escribir con un teclado o distinguir visualmente de los de aspecto similar; algunos son incluso invisibles. Los escapes **Unicode** en los literales de **string** de **Go** nos permiten especificarlos por su valor de punto de código numérico. Hay dos formas, **\uhhhh** por un valor de **16 bits** y **\Uhhhhhhhh** por un valor de **32 bits**, donde cada **h** es un dígito hexadecimal; la necesidad de la forma de **32 bits** surge con muy poca frecuencia. Cada una denota la codificación **UTF-8** del punto de código especificado. Así, por ejemplo, los siguientes literales de **strings** representan el misma **string** de seis bytes:

```
"世界"
"\xe4\xb8\x96\xe7\x95\x8c"
"\u4e16\u754c"
"\U00004e16\U0000754c"
```

Las tres secuencias de escape anteriores proporcionan notaciones alternativas para el primer **string**, pero los valores que denotan son idénticos.

Los escapes **unicode** también se pueden usar en los literales de runa. Estos tres literales son equivalentes:

```
'世' '\u4e16' '\U00004e16'
```

Una runa cuyo valor es inferior a **256** se puede escribir con un solo de escape hexadecimal, por ejemplo, **'\x41'** para **'A'**, pero para los valores más altos, debe ser utilizado un escape **\u** o **\U**. En consecuencia, **'\xe4\xb8\x96'** no es un literal de runa legal, a pesar de que esos tres bytes son una codificación válida **UTF-8** de un único punto de código.

Gracias a las agradables propiedades de **UTF-8**, muchas operaciones de **string** no requieren decodificación. Podemos comprobar si un **string** contiene a otro como un prefijo:

```
func HasPrefix(s, prefix string) bool {
    return len(s) >= len(prefix) && s[:len(prefix)] == prefix
}
```

o como un sufijo:

```
func HasSuffix(s, suffix string) bool {
    return len(s) >= len(suffix) && s[len(s)-len(suffix):] == suffix
}
```

o como un **substring**:

```
func Contains(s, substr string) bool {
    for i := 0; i < len(s); i++ {
        if HasPrefix(s[i:], substr) {
            return true
        }
    }
    return false
}
```

Usando la misma lógica para el texto **UTF-8** codificado como bytes sin formato. Esto no es cierto para otras codificaciones. (Las funciones anteriores se han extraído del paquete **strings**, aunque su implementación de **Contains** utiliza una técnica de **hashing** para buscar más eficientemente.)

Por otro lado, si realmente nos interesan los caracteres **Unicode** individuales, tenemos que utilizar otros mecanismos. Considerar el **string** de nuestro primer ejemplo, que incluye dos caracteres de Asia oriental. La **Figura 3.5** ilustra su representación en la memoria. El **string** contiene **13 bytes**, pero interpretado como **UTF-8**, se codifica con sólo nueve puntos de código o runas:

```
import "unicode/utf8"
s := "Hello, 世界"
fmt.Println(len(s)) // "13"
fmt.Println(utf8.RuneCountInString(s)) // "9"
```

Para procesar esos caracteres, es necesario un decodificador **UTF-8**. El paquete **unicode/utf8** proporciona una forma que podemos utilizar la siguiente manera:

```
for i := 0; i < len(s); {
    r, size := utf8.DecodeRuneInString(s[i:])
    fmt.Printf("%d\t%c\n", i, r)
    i += size
}
```

Cada llamada a **DecodeRuneInString** retorna **r**, la propia runa, y **size**, el número de bytes ocupados por la codificación **UTF-8** de **r**. El tamaño (**size**) se utiliza para actualizar el índice de byte **i** de la siguiente runa en el **string**. Pero esto es poco elegante, y necesitamos bucles de este tipo todo el tiempo. Afortunadamente, el bucle **range** de **Go**, cuando se aplica a un **string**, realiza decodificación de **UTF-8** de forma implícita. A continuación, se muestra la salida del bucle en la **Figura 3.5**; observar cómo el índice salta más de **1** de cada runa no **ASCII**.

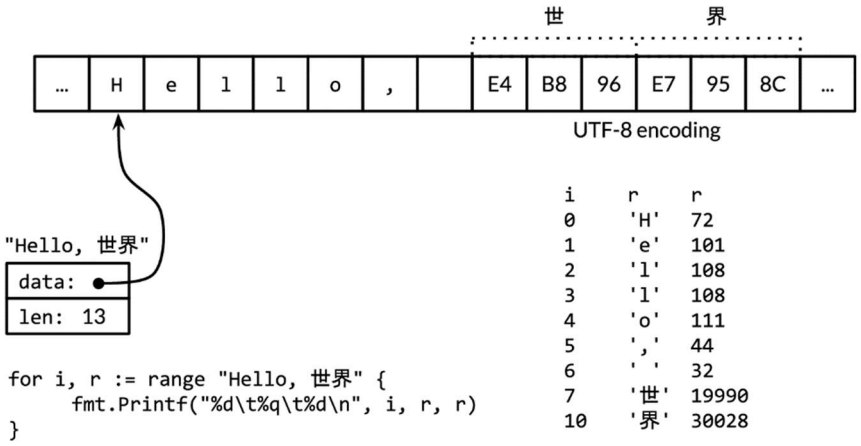

Figura 3.5. Un bucle **range** que decodifica un **string** codificado en **UTF-8**.

```
for i, r := range "Hello, 世界" {
    fmt.Printf("%d\t%q\t%d\n", i, r, r)
}
```

Podríamos usar un simple bucle **range** para contar el número de runas en un **string**, como esto:

```
n := 0
for _, _ = range s {
    n++
}
```

Al igual que con las otras formas del bucle **range**, podemos omitir las variables que no necesitamos:

```
n := 0
for range s {
    n++
}
```

O simplemente podemos llamar a **utf8.RuneCountInString(s)**.

Ya hemos mencionado anteriormente que es sobre todo una cuestión de convención en **Go** que los **strings** de texto se interpreten como secuencias codificadas en **UTF-8** de puntos de código **Unicode**, pero para el uso correcto de los bucles **range** en los **strings**, es más que una convención, es una necesidad. ¿Qué pasa si hacemos un **range** sobre un **string** que contiene datos binarios arbitrarios o, datos **UTF-8** que contienen errores?

Cada vez que un decodificador **UTF-8**, ya sea explícitamente en una llamada a **utf8.DecodeRuneInString** o implícitamente en un bucle **range**, consume un byte de entrada inesperado, se genera un carácter **Unicode** especial de reemplazo, **'\uFFFD'**, que generalmente está impreso como un signo de interrogación blanco dentro de una forma hexagonal o similar al diamante negro **  **. Cuando un programa encuentra este valor de runa, generalmente es una señal de que alguna parte anterior del sistema que genera los datos de **string** ha sido descuidada en el tratamiento de las codificaciones de texto.

UTF-8 es excepcionalmente conveniente como un formato de intercambio, pero dentro de un programa las runas pueden ser más convenientes debido a que son de tamaño uniforme y por lo tanto se indexan con facilidad en los arreglos y los **slices**.

Una conversión **[]rune** aplicada a un **string** codificado en **UTF-8** devuelve la secuencia de códigos de punto **Unicode** que codifica el **string**:

```
// "program" en japonés katakana
s := "プログラム"
fmt.Printf("% x\n", s) // "e3 83 97 e3 83 ad e3 82 b0 e3 83 a9 e3 83 a0"
r := []rune(s)
fmt.Printf("%x\n", r) // "[30d7 30ed 30b0 30e9 30e0]"
```

(El verbo **%x** en el primer **Printf** introduce un espacio entre cada par de dígitos hexadecimales.)

Si un **slice** de runas se convierte en un **string**, se produce una concatenación de las codificaciones de cada runa **UTF-8**:

```
fmt.Println(string(r)) // "プログラム"
```

La conversión de un valor entero a un **string** interpreta el número entero como un valor de runa, y se obtiene la representación **UTF-8** de la misma runa:

```
fmt.Println(string(65)) // "A", not "65"
fmt.Println(string(0x4eac)) // "京"
```

Si la runa no es válida, el carácter de reemplazo se sustituirá por:

```
fmt.Println(string(1234567)) // "�"
```

3.5.4. Strings y Slices de Byte

Hay cuatro paquetes estándar particularmente importantes para la manipulación de **strings**: **bytes**, **strings**, **strconv** y **unicode**. El paquete **strings** proporciona muchas funciones para buscar, reemplazar, comparar, recortar, dividir, y unir **strings**.

El paquete **bytes** tiene funciones similares para manipular **slices** de bytes, de tipo **[]byte**, que comparten algunas propiedades con los **strings**. Debido a que los **strings** son inmutables, la creación de **strings** de forma incremental puede implicar una gran cantidad de asignación y copia. En tales casos, es más eficaz utilizar el tipo **bytes.Buffer**, que mostraremos en un momento.

El paquete **strconv** proporciona funciones para convertir valores booleanos, enteros, y de coma flotante hacia y desde sus representaciones de **string**, y funciones para el encomillado y desencomillado de **strings**.

El paquete **unicode** proporciona funciones como e **IsDigit**, **IsLetter**, **IsUpper** y **isLower** para la clasificación de runas. Cada función toma un solo argumento runa y devuelve un booleano. Las funciones de conversión como **ToUpper** y **ToLower** convierten una runa en el caso que corresponda si se trata de una letra. Todas estas funciones utilizan las categorías estándar **Unicode** para las letras, dígitos, y demás. El paquete **strings** tiene funciones similares, también llamadas **ToUpper** y **ToLower**, que devuelven un nuevo **string** con la transformación especificada aplicada a cada carácter del **string** original.

La función **basename** por está inspirada en la utilidad de **shell Unix** del mismo nombre. En nuestra versión, **basename(s)** elimina cualquier prefijo de **s** que parezca un sistema de ficheros con componentes separados por barras, y se elimina cualquier sufijo que se parezca a un tipo de archivo:

```
fmt.Println(basename("a/b/c.go")) // "c"
fmt.Println(basename("c.d.go")) // "c.d"
fmt.Println(basename("abc")) // "abc"
```

La primera versión del **basename** hace todo el trabajo sin la ayuda de librerías:

```
gopl.io/ch3/basename1
// basename elimina los componentes de directorio y un .suffix.
// Por ejemplo, a => a, a.go => a, a/b/c.go => c, a/b.c.go => b.c
func basename(s string) string {
    // Descartar última '/' y todo antes.
    for i := len(s) - 1; i >= 0; i-- {
        if s[i] == '/' {
            s=s[i+1:]
            break
        }
    }
    // Preservar todo antes de la última '.'.
    for i := len(s) - 1; i >= 0; i-- {
        if s[i] == '.' {
            s=s[:i]
            break
        }
    }
    return s
}
```

Una versión más simple utiliza la función **strings.LastIndex** de librería:

```
gopl.io/ch3/basename2
func basename(s string) string {
    slash := strings.LastIndex(s, "/") // -1 if "/" not found
    s=s[slash+1:]
    if dot := strings.LastIndex(s, "."); dot >= 0 {
        s=s[:dot]
    }
    return s
}
```

Los paquetes **path** y **path/filepath** proporcionan un conjunto más general de funciones para manipular nombres jerárquicos. El paquete **path** funciona con rutas delimitada por barras en cualquier plataforma. No debe ser usado para nombres de archivo, pero es apropiado para otros dominios, como el componente de ruta de una **URL**. Por el contrario, **path/filepath** manipula los nombres de archivos usando las reglas para la plataforma de acogida, tales como **/foo/bar** para **POSIX** o **c:\foo\bar** en **Microsoft Windows**.

Vamos a continuar con otro ejemplo de **substring**. La tarea es tomar una representación **string** de un entero, como "**12345**", e insertar comas cada tres lugares, como en "**12,345**". Esta versión sólo funciona para números enteros; el manejo de números de coma flotante se deja como un ejercicio.

```
gopl.io/ch3/comma
// Inserta comas comas en un string decimal entero no negativo.
func comma(s string) string {
    n := len(s)
    if n <= 3 {
        return s
    }
    return comma(s[:n-3]) + "," + s[n-3:]
}
```

El argumento de **comma** es un **string**. Si su longitud es menor que o igual a **3**, no es necesario. De lo contrario, **comma** se llama a sí misma de forma recursiva con una **substring** formada por todos menos los tres últimos caracteres, y agrega una coma y los tres últimos caracteres en el resultado de la llamada recursiva.

Un **string** contiene un arreglo de bytes que, una vez creados, es inmutable. Por el contrario, los elementos de una **slice** de bytes se pueden modificar libremente.

Los **strings** pueden ser convertidos a **slices** de byte y viceversa:

```
s := "abc"
b := []byte(s)
s2 := string(b)
```

Conceptualmente, la conversión **[]byte(s)** asigna un nuevo arreglo de bytes que contiene una copia de los bytes de **s**, y produce un **slice** que hace referencia a la totalidad del arreglo. Un compilador optimizado puede ser capaz de evitar la asignación y la copia en algunos casos, pero en general, la copia es necesaria para asegurar que los bytes de **s** se mantienen sin cambios incluso si los de **b** son modificados posteriormente. La conversión desde **slice** de bytes de nuevo a **string** con **string(b)** también hace una copia, para garantizar la inmutabilidad del **string** resultante **s2**.

Para evitar las conversiones y asignación de memoria innecesaria, muchas de las funciones de utilidad en el paquete **bytes** directamente paraleliza sus contrapartes en el paquete **strings**. Por ejemplo, aquí hay una media docena de funciones de **strings**:

```
func Contains(s, substr string) bool
func Count(s, sep string) int
func Fields(s string) []string
func HasPrefix(s, prefix string) bool
func Index(s, sep string) int
func Join(a []string, sep string) string
```

y las correspondientes de **bytes**:

```
func Contains(b, subslice []byte) bool
func Count(s, sep []byte) int
func Fields(s []byte) [][]byte
func HasPrefix(s, prefix []byte) bool
func Index(s, sep []byte) int
func Join(s [][]byte, sep []byte) []byte
```

La única diferencia es que los **strings** se han sustituido por **slices** de bytes.

El paquete **bytes** proporciona el tipo **Buffer** para la manipulación eficiente de los **slices** de byte. Un **Buffer** comienza vacío, pero crece a medida que los datos de tipos como **string**, **byte** y **[]byte** se escriben en él. Como muestra el siguiente ejemplo, una variable **bytes.Buffer** no requiere inicialización debido a que se puede utilizar su valor cero:

```
gopl.io/ch3/printints
// intsToString es como fmt.Sprintf (valores), pero añade comas.
func intsToString(values []int) string {
    var buf bytes.Buffer
    buf.WriteByte('[')
    for i, v := range values {
        if i > 0 {
            buf.WriteString(", ")
        }
        fmt.Fprintf(&buf, "%d", v)
    }
    buf.WriteByte(']')
    return buf.String()
}

func main() {
    fmt.Println(intsToString([]int{1, 2, 3})) // "[1, 2, 3]"
}
```

Al añadir la codificación **UTF-8** de una runa arbitraria a un **bytes.Buffer**, es mejor utilizar el método **WriteRune** de **bytes.Buffer**, pero **WriteByte** está muy bien para los caracteres **ASCII** como '**[**' y '**]**'.

El tipo **bytes.Buffer** es extremadamente versátil, y cuando hablemos de interfaces en el **Capítulo 7**, veremos cómo se puede utilizar como un sustituto de un archivo cada vez que una función de **E/S** requiere un sumidero de bytes (**io.Writer**) como hizo anteriormente **Fprintf**, o una fuente de bytes (**io.Reader**).

3.5.5. Conversiones entre Strings y Números

Además de las conversiones entre **strings**, runas y bytes, a menudo es necesario convertir entre valores numéricos y sus representaciones de **string**. Esto se hace con las funciones del paquete **strconv**.

Para convertir un entero a un **string**, una opción es utilizar **fmt.Sprintf**; otra es utilizar la función **strconv.Itoa("entero a ASCII")**:

```
x := 123
y := fmt.Sprintf("%d", x)
fmt.Println(y, strconv.Itoa(x)) // "123 123"
```

Se pueden utilizar **FormatInt** y **FormatUint** para formatear números en una base diferente:

```
fmt.Println(strconv.FormatInt(int64(x), 2)) // "1111011"
```

Los verbos de **fmt.Printf %b**, **%d**, **%u** y **%x** son a menudo más convenientes que las funciones **Format**, sobre todo si queremos incluir información adicional además del número:

```
s := fmt.Sprintf("x=%b", x) // "x=1111011"
```

Para analizar un **string** que representa un número entero, utilizar las funciones de **strconv, Atoi** o **Parseint**, o **ParseUint** para enteros sin signo:

```
x, err := strconv.Atoi("123") // x es un int
y, err := strconv.ParseInt("123", 10, 64) // base 10, hasta 64 bits
```

El tercer argumento de **ParseInt** da el tamaño del tipo entero en el que el resultado debe encajar; por ejemplo, **16** implica **int16**, y el valor especial de **0** implica **int**. En cualquier caso, el tipo del resultado **y** siempre es **int64**, que luego se puede convertir a un tipo más pequeño.

A veces **fmt.Scanf** es útil para el análisis de entrada que consta de mezclas ordenadas de **strings** y números de todo en una sola línea, pero puede ser inflexible, especialmente con el manejo de entrada incompleta o irregular.

3.6. Constantes

Las constantes son expresiones cuyo valor es conocido por el compilador y cuya evaluación se garantiza que se produzca en tiempo de compilación, no en tiempo de ejecución. El tipo subyacente de cada constante es un tipo básico: booleano, **string** o número.

Una declaración **const** define los valores que se ven sintácticamente como variables con nombre, pero cuyo valor es constante, lo que evita cambios accidentales (o infames) durante la ejecución del programa. Por ejemplo, una constante es más apropiada que una variable para una constante matemática como **pi**, ya que su valor no va a cambiar:

```
const pi = 3.14159 // aproximadamente; math.pi es una mejor aproximación
```

Al igual que con las variables, puede aparecer una secuencia de constantes en una declaración; esto sería apropiado para un grupo de valores relacionados:

```
const (
    e = 2.71828182845904523536028747135266249775724709369995957496696763
    pi = 3.14159265358979323846264338327950288419716939937510582097494459
)
```

Se pueden evaluar muchos cálculos de constantes por completo en tiempo de compilación, lo que reduce el trabajo necesario en tiempo de ejecución y permite otras optimizaciones del compilador. Los errores detectados normalmente en tiempo de ejecución pueden ser reportados en tiempo de compilación cuando sus operandos son constantes, tales como la división por cero, **string** de indexación fuera de límites, y cualquier operación de coma flotante que dé lugar a un valor que no sea finito.

Los resultados de todas las operaciones aritméticas, operaciones de comparación y lógicas aplicados a operandos constantes son en sí mismos constantes, al igual que los resultados de las conversiones y las llamadas a ciertas funciones incorporadas tales como **len**, **cap**, **real**, **imag**, **complex** y **unsafe.Sizeof** (**§13.1**).

Como sus valores son conocidos por el compilador, las expresiones constantes pueden aparecer en tipos, específicamente como la longitud de un tipo de arreglo:

```
const IPv4Len = 4
// parseIPv4 analiza una dirección IPv4 (d.d.d.d).
func parseIPv4(s string) IP {
    var p [IPv4Len]byte
    // ...
}
```

Una declaración **constant** puede especificar un tipo, así como un valor, pero en ausencia de un tipo explícito, el tipo se deduce de la expresión en el lado derecho. En lo siguiente, **time.Duration** es un tipo con nombre cuyo tipo subyacente es **int64** , y **time.Minute** es una constante de ese tipo. Las constantes declaradas a continuación por lo tanto tienen el tipo **time.Duration** así, como lo revela **%T**:

```
const noDelay time.Duration = 0
const timeout = 5 * time.Minute
fmt.Printf("%T %[1]v\n", noDelay) // "time.Duration 0"
fmt.Printf("%T %[1]v\n", timeout) // "time.Duration 5m0s
fmt.Printf("%T %[1]v\n", time.Minute) // "time.Duration 1m0s"
```

Cuando una secuencia de constantes se declara como un grupo, la expresión del lado derecho se puede omitir para todos menos el primero del grupo, lo que implica que la expresión anterior y su tipo deben ser utilizados de nuevo. Por ejemplo:

```
const (
    a=1
    b
    c=2
    d
)
fmt.Println(a, b, c, d) // "1 1 2 2"
```

Esto no es muy útil si la expresión del lado derecho implícitamente copiada siempre evalúa la misma cosa. Pero ¿podría variar? Esto nos lleva a **iota**.

3.6.1. El generador de constante iota

Una declaración **const** puede utilizar el generador de constante **iota**, que se utiliza para crear una secuencia de valores relacionados sin detallar cada uno de forma explícita. En una declaración **const**, el valor de **iota** comienza en cero y se incrementa en uno por cada elemento de la secuencia.

Aquí hay un ejemplo del paquete **time**, que define constantes con nombre de tipo **Weekday** de los días de la semana, a partir de cero para **Sunday**. Los tipos de esta forma a menudo se llaman enumeraciones, o **enums**, para abreviar.

```
type Weekday int
const (
    Sunday Weekday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
)
```

Esto declara **Sunday** con el valor **0**, **Monday** con el valor 1, y así sucesivamente.

Podemos utilizar **iota** en expresiones más complejas también, como en este ejemplo del paquete **net** donde se le da a cada uno los **5 bits** más bajos de un entero sin signo de un nombre distinto y con interpretación booleana:

```
type Flags uint
const (
    FlagUp Flags = 1 << iota // is up
    FlagBroadcast // compatible con la capacidad de acceso de difusión
    FlagLoopback // es una interfaz de bucle invertido
    FlagPointToPoint // pertenece a un enlace point-to-point
    FlagMulticast // soporta capacidad de acceso de multidifusión
)
```

Con los incrementos **iota**, si a cada constante se le asigna el valor de **1 << iota**, evalúa sucesivas potencias de dos, cada una correspondiente a un solo bit. Podemos utilizar estas constantes dentro de las funciones de esa prueba, o establecer o borrar uno o más de estos bits:

```
gopl.io/ch3/netflag
    func IsUp(v Flags) bool { return v&FlagUp == FlagUp }
    func TurnDown(v *Flags) { *v &^= FlagUp }
    func SetBroadcast(v *Flags) { *v |= FlagBroadcast }
    func IsCast(v Flags) bool { return v&(FlagBroadcast|FlagMulticast) != 0 }

func main() {
    var v Flags = FlagMulticast | FlagUp
    fmt.Printf("%b %t\n", v, IsUp(v)) // "10001 true"
    TurnDown(&v)
    fmt.Printf("%b %t\n", v, IsUp(v)) // "10000 false"
    SetBroadcast(&v)
    fmt.Printf("%b %t\n", v, IsUp(v)) // "10010 false"
    fmt.Printf("%b %t\n", v, IsCast(v)) // "10010 true"
}
```

Como un ejemplo más complejo de **iota**, esta declaración nombra las potencias de **1024**:

```
const (
    _ = 1 << (10 * iota)
    KiB // 1024
    MiB // 1048576
    GiB // 1073741824
    TiB // 1099511627776 (excede 1 << 32)
    PiB // 1125899906842624
    EiB // 1152921504606846976
    ZiB // 118059162071741130342 4 (excede 1 << 64)
    YiB // 120892581961462917470617 6
)
```

El mecanismo **iota** tiene sus límites. Por ejemplo, no es posible generar las potencias más familiares de **1000** (**KB**, **MB**, y así sucesivamente), ya que no hay ningún operador de exponenciación.

3.6.2. Constantes sin tipo

Las constantes en **Go** son un poco inusuales. A pesar de que una constante puede tener cualquiera de los tipos de datos básicos como **int** o **float64**, incluyendo tipos básicos nombrados como **time.Duration**, muchas constantes no están comprometidas con un tipo particular. El compilador representa estas constantes no comprometidas con mucha mayor precisión numérica que los valores de los tipos básicos, y la aritmética en ellos es más precisa que la máquina aritmética; se pueden suponer al menos **256 bits** de precisión. Hay seis sabores de estas constantes no comprometidas, llamados booleanos sin tipo, enteros sin tipo, runa sin tipo, coma flotante sin tipo, complejo sin tipo y **string** sin tipo.

Al aplazar este compromiso, las constantes sin tipo no sólo conservan su precisión más alta hasta más tarde, pueden participar en muchas más expresiones que las constantes comprometidas sin necesidad de conversiones. Por ejemplo, los valores **ZiB** y **YiB** en el ejemplo anterior son demasiado grandes para almacenar en cualquier variable entera, pero son constantes legítimas que pueden usarse en expresiones como ésta:

```
fmt.Println(YiB/ZiB) // "1024"
```

Como otro ejemplo, la constante de coma flotante **math.Pi** se puede usar dondequiera que se necesite cualquier valor de coma flotante o complejo:

```
var x float32 = math.Pi
var y float64 = math.Pi
var z complex128 = math.Pi
```

Si **math.Pi** hubiera estado comprometido con un tipo específico, como **float64**, el resultado no sería tan preciso, y se requeriría conversiones de tipos para usarlo cuando es buscado un valor **float32** o **complex128**:

```
const Pi64 float64 = math.Pi
var x float32 = float32(Pi64)
var y float64 = Pi64
var z complex128 = complex128(Pi64)
```

Para los literales, la sintaxis determina el sabor. Todos los literales **0**, **0.0**, **0i** y **'\u0000'** denotan constantes del mismo valor, pero con diferentes sabores: entero sin tipo, coma flotante sin tipo, complejo sin tipo y runa sin tipo, respectivamente. Del mismo modo, **true** y **false** son booleanos sin tipo y los literales de **string** son **strings** sin tipo.

Recordemos que **/** puede representar un número entero o una división de coma flotante en función de sus operandos. En consecuencia, la elección del literal puede afectar el resultado de una expresión de división constante:

```
var f float64 = 212
fmt.Println((f - 32) * 5 / 9) // "100"; (f - 32) * 5 es  un float64
fmt.Println(5 / 9 * (f - 32)) // "0"; 5/9 es un entero sin tipo, 0
fmt.Println(5.0 / 9.0 * (f - 32)) // "100"; 5.0/9.0 es un float sin tipo
```

Sólo las constantes pueden ser sin tipo. Cuando una constante sin tipo se asigna a una variable, como en la primera declaración de abajo, o aparece en el lado derecho de una declaración de variable con un tipo explícito, como en las otras tres líneas, la constante se convierte implícitamente al tipo de esa variable si es posible.

```
var f float64 = 3 + 0i // complejo sin tipo -> float64
f=2 // entero sin tipo -> float64
f=1e123 // coma flotante sin tipo -> float64
f = 'a' // runa sin tipo -> float64
```

Las declaraciones anteriores son por lo tanto equivalentes a las siguientes:

```
var f float64 = float64(3 + 0i)
f=float64(2)
f=float64(1e123)
f=float64('a')
```

Ya sea implícita o explícita, la conversión de una constante de un tipo a otro requiere que el tipo de destino pueda representar el valor original. El redondeo es permitido para números reales y para complejos de coma flotante:

```
const (
    deadbeef = 0xdeadbeef // entero sin tipo con valor 3735928559
    a=uint32(deadbeef) // uint32 con valor 3735928559
    b=float32(deadbeef) // float32 con valor 3735928576 (redondeado)
    c=float64(deadbeef) // float64 con valor 3735928559 (exacto)
    d=int32(deadbeef) // error de compilación: constante supera int32
    e=float64(1e309) // error de compilación: constante supera float64
    f=uint(-1) // error de compilación: constante supera uint
)
```

En una declaración de variable sin un tipo explícito (incluyendo las declaraciones de variables cortas), el sabor de la constante sin tipo determina implícitamente el tipo predeterminado de la variable, como en estos ejemplos:

```
i := 0 // entero sin tipo; implícito int(0)
r := '\000' // runa sin tipo; implícito rune('\000')
f := 0.0 // coma flotante sin tipo; implícito float64(0.0)
c := 0i // complejo sin tipo; implícito complex128(0i)
```

Tener en cuenta la asimetría: los enteros sin tipo se convierten a **int**, cuyo tamaño no está garantizado, pero los números de coma flotante y complejos sin tipo se convierten a los tipos de tamaño de forma explícita **float64** y **complex128**. El lenguaje no tiene un **float** sin tamaño y tipos análogos complejos a un **int** sin tamaño, ya que es muy difícil escribir algoritmos numéricos correctos sin conocer el tamaño de los tipos de datos de coma flotante.

Para dar a la variable un tipo diferente, hay que convertir explícitamente la constante sin tipo al tipo deseado o indicar el tipo deseado en la declaración de variable, como en estos ejemplos:

```
var i = int8(0)
var i int8 = 0
```

Estos valores por defecto son particularmente importantes cuando se convierte una constante sin tipo a un valor de interface (véase el **Capítulo 7**), puesto que determinan su tipo dinámico.

```
fmt.Printf("%T\n", 0) // "int"
fmt.Printf("%T\n", 0.0) // "float64"
fmt.Printf("%T\n", 0i) // "complex128"
fmt.Printf("%T\n", '\000') // "int32" (rune)
```

Hemos cubierto los tipos de datos básicos de **Go**. El siguiente paso es mostrar cómo se pueden combinar en grupos más grandes, como los arreglos y estructuras, y luego en estructuras de datos para la resolución de problemas de programación real; ese es el tema del **Capítulo 4**.