Capítulo 6

Métodos

Desde principios de **1990**, la programación orientada a objetos (**POO**) ha sido el paradigma de programación dominante en la industria y la educación, y casi todos los lenguajes ampliamente desarrollados desde entonces han incluido soporte para el mismo. **Go** no es una excepción.

Aunque no existe una definición universalmente aceptada de la programación orientada a objetos, para nuestros propósitos, un objeto es simplemente un valor o variable que tiene métodos, y un método es una función asociada con un tipo particular. Un programa orientado a objetos es uno que utiliza métodos para expresar las propiedades y operaciones de cada estructura de datos para que los clientes no tengan que acceder a la representación del objeto directamente.

En los capítulos anteriores, hemos hecho uso regular de métodos de la librería estándar, como el método **Seconds** del tipo **time.Duration**:

```
const day = 24 * time.Hour
fmt.Println(day.Seconds()) // "86400"
```

y definimos un método propio en la **Sección 2.5**, un método **String** para el tipo **Celsius**:

```
func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }
```

En este capítulo, el primero de los dos en la programación orientada a objetos, mostraremos cómo definir y utilizar métodos con eficacia. También trataremos dos principios fundamentales de la programación orientada a objetos, la encapsulación y la composición.

6.1. Declaraciones de método

Un método se declara con una variante de la declaración de la función ordinaria en el que aparece un parámetro extra antes del nombre de la función. El parámetro fija la función al tipo de ese parámetro.

Vamos a escribir nuestro primer método en un sencillo paquete para geometría plana:

```
gopl.io/ch6/geometry
package geometry

import "math"

type Point struct{ X, Y float64 }
// Función tradicional
func Distance(p, q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

// Lo mismo, pero como un método del tipo Point
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}
```

El parámetro extra **p** se llama el **receptor** del método, un legado de los primeros lenguajes orientados a objetos que describían las llamadas a un método como "el envío de un mensaje a un objeto."

En **Go**, no se utiliza un nombre especial como **this** o **self** para el receptor; elegimos nombres de receptor tal como lo haríamos con cualquier otro parámetro. Dado que se utiliza con frecuencia el nombre del receptor, es una buena idea elegir algo corto y que sea consistente en los métodos. Una opción común es la primera letra del nombre del tipo, como **p** para el **Point**.

En una llamada a un método, el argumento del receptor aparece antes del nombre del método. Esto es paralelo a la declaración, en la que el parámetro del receptor aparece antes que el nombre del método.

```
p := Point{1, 2}
q := Point{4, 6}
fmt.Println(Distance(p, q)) // "5", llamada a la función
fmt.Println(p.Distance(q)) // "5", llamada de método
```

No hay ningún conflicto entre las dos declaraciones de funciones llamadas **Distance** anteriormente. La primera declara una función de nivel de paquete llamada **geometry.Distance**. La segunda declara un método del tipo **Point**, por lo que su nombre es **Point.Distance**.

La expresión **p.Distance** se llama un **selector**, porque selecciona el apropiado método **Distance** para el receptor **p** de tipo **Point**. Los selectores también se utilizan para seleccionar los campos de tipos de estructura, como en **p.X**. Dado

que los métodos y los campos habitan el mismo espacio de nombres, si se declara un método **X** en la estructura de tipo **Point** sería ambiguo y el compilador lo rechazaría.

Debido a que cada tipo tiene su propio espacio de nombres para los métodos, podemos utilizar el nombre **Distance** de otros métodos, siempre que pertenezcan a distintos tipos. Vamos a definir un tipo **Path** que representa una secuencia de segmentos de línea y darle también un método **Distance**.

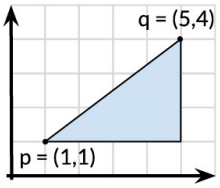
```
// Un Path es un camino que conecta puntos con líneas rectas.
type Path []Point
// Distance devuelve la distancia recorrida a lo largo del camino.
func (path Path) Distance() float64 {
    sum := 0.0
    for i := range path {
        if i > 0 {
            sum += path[i-1].Distance(path[i])
        }
    }
    return sum
}
```

Path es un tipo de **slice** con nombre, no es un tipo de estructura como **Point**, sin embargo, se le pueden definir métodos. Al permitir que los métodos se asocien con cualquier tipo, **Go** se diferencia de muchos otros lenguajes orientados a objetos. A menudo es conveniente definir comportamientos adicionales para los tipos simples, tales como números, **strings**, **slices**, mapas y a veces incluso funciones. Los métodos pueden ser declarados en cualquier tipo con nombre definido en el mismo paquete, siempre y cuando su tipo subyacente no sea ni un puntero ni una **interface**.

Los dos métodos **Distance** tienen diferentes tipos. No están relacionados entre sí en absoluto, aunque **Path.Distance** utiliza **Point.Distance** internamente para calcular la longitud de cada segmento que une los puntos adyacentes.

Vamos a llamar al nuevo método para calcular el perímetro de un triángulo rectángulo:

```
perim := Path{
    {1, 1},
    {5, 1},
    {5, 4},
    {1, 1},
}
fmt.Println(perim.Distance()) // "12"
```



En las dos llamadas de los métodos nombrados **Distance**, el compilador determina qué función debe llamar basándose en el nombre del método y en el tipo de receptor. En la primera, **path[i-1]** tiene el tipo **Point** de modo que se llama a **Point.Distance**; en el segundo, **perim** tiene el tipo **Path**, por lo que se llama a **Path.Distance**.

Todos los métodos de un tipo dado deben tener nombres únicos, pero diferentes tipos pueden utilizar el mismo nombre para un método, al igual que los métodos **Distance** para **Point** y **Path**; no hay necesidad de calificar los nombres de función (por ejemplo, **PathDistance**) para eliminar la ambigüedad. Aquí vemos la primera ventaja de usar métodos sobre las funciones ordinarias: los nombres de los métodos pueden ser más cortos. El beneficio se magnifica para las llamadas que se originan fuera del paquete, ya que pueden utilizar el nombre más corto y omitir el nombre del paquete:

```
import "gopl.io/ch6/geometry"

perim := geometry.Path{{1, 1}, {5, 1}, {5, 4}, {1, 1}}
fmt.Println(geometry.PathDistance(perim)) // "12", función independiente
fmt.Println(perim.Distance()) // "12", método de geometry.Path
```

6.2. Los métodos con un receptor de puntero

Debido a que al llamar a una función se realiza una copia de cada valor del argumento, si una función tiene que actualizar una variable, o si un argumento es tan grande que deseamos evitar copiarlo, se debe pasar la dirección de la variable usando un puntero. Lo mismo ocurre con los métodos que necesitan actualizar la variable del receptor: les asignamos el tipo puntero, como ***Point**.

```
func (p *Point) ScaleBy(factor float64) {
    p.X *= factor
    p.Y *= factor
}
```

El nombre de este método es **(*Point).ScaleBy**. Los paréntesis son necesarios; sin ellos, la expresión se analiza cómo ***(Point.ScaleBy)**.

En un programa realista, la convención dicta que, si cualquier método de **Point** tiene un receptor puntero, entonces todos los métodos de **Point** deben tener un receptor puntero, incluso los que realmente no lo necesitan. Hemos roto esta regla para **Point** de modo que podamos mostrar ambos tipos de método.

Los tipos con nombre (**Point**) y punteros a ellos (***Point**) son los únicos tipos que pueden aparecer en una declaración de receptor. Además, para evitar ambigüedades, las declaraciones de métodos no están permitidas en los tipos con nombre que son ellos mismos tipos de puntero:

```
type P *int
func (P) f() { /* ... */ } // error de compilación: tipo de receptor no válido
```

El método **(*Point).ScaleBy** se puede llamar al proporcionar un receptor ***Point**, de esta manera:

```
r := &Point{1, 2}
r.ScaleBy(2)
fmt.Println(*r) // "{2, 4}"
```

o esto:

```
p := Point{1, 2}
pptr := &p
pptr.ScaleBy(2)
fmt.Println(p) // "{2, 4}"
```

o esto:

```
p := Point{1, 2}
(&p).ScaleBy(2)
fmt.Println(p) // "{2, 4}"
```

Pero los dos últimos casos son torpes. Afortunadamente, el lenguaje nos ayuda aquí. Si el receptor **p** es una variable de tipo **Point**, pero el método requiere un receptor ***Point**, podemos emplear sus abreviaturas:

```
p.ScaleBy(2)
```

y el compilador realizará una implícita **&p** sobre la variable. Esto sólo funciona para las variables, incluidos los campos de **struct** como **p.X** y elementos de arreglo o **slice** como **perim[0]**. No podemos llamar a un método ***Point** en un receptor no direccionable **Point**, porque no hay manera de obtener la dirección de un valor temporal.

```
Point{1, 2}.ScaleBy(2) // error de compilación: no se puede tomar la dirección del literal Point
```

Pero podemos llamar un método **Point** como **Point.Distance** con un receptor ***Point**, debido a que hay una manera de obtener el valor de la dirección: sólo se tiene que cargar el valor apuntado por el receptor. El compilador inserta una implícita operación * para nosotros. Estas dos llamadas a funciones son equivalentes:

```
pptr.Distance(q)
(*pptr).Distance(q)
```

Vamos a resumir estos tres casos más, ya que son un punto de confusión frecuente. En cada expresión de llamada a un método válido, exactamente una de estas tres afirmaciones es verdadera.

O bien el argumento receptor tiene el mismo tipo que el parámetro del receptor, por ejemplo, ambos tienen tipo **T** o ambos tienen tipo ***T**:

```
Point{1, 2}.Distance(q) // Point
pptr.ScaleBy(2) // *Point
```

O el argumento de receptor es una variable de tipo **T** y el parámetro receptor tiene un tipo ***T**. El compilador toma implícita la dirección de la variable:

```
p.ScaleBy(2) // implicito (&p)
```

O el argumento receptor tiene tipo ***T** y el parámetro receptor tiene tipo **T**. El compilador implícitamente elimina la referencia del receptor, en otras palabras, carga el valor:

```
pptr.Distance(q) // implicito (*pptr)
```

Si todos los métodos de un tipo llamado **T** tienen un tipo de receptor de **T** en sí (no ***T**), es seguro para copiar las instancias de ese tipo; llamando a cualquiera de sus métodos necesariamente se hace una copia. Por ejemplo, los valores de **time.Duration** se copian, incluyéndolos como argumentos de funciones. Pero si cualquier método tiene un receptor de puntero, se debe evitar copiar instancias de **T**, ya que hacerlo puede violar las invariantes internas. Por ejemplo, la copia de una instancia de **bytes.Buffer** haría que el original y la copia de alias (**§2.3.2**) el mismo arreglo subyacente de bytes. Las llamadas a métodos posteriores tendrían efectos impredecibles.

6.2.1. Nil, constituirá un valor válido de receptor

Al igual que algunas funciones permiten los punteros **nil** como argumentos, también lo hacen algunos métodos para su receptor, especialmente si **nil** es un valor significativo a cero del tipo, al igual que con los mapas y los **slices**. En esta simple lista enlazada de enteros, **nil** representa la lista vacía:

```
// Un intlist es una lista enlazada de enteros.
// Una nil *IntList representa la lista vacía.
type IntList struct {
    Value int
    Tail *IntList
}

// Sum devuelve la suma de los elementos de la lista.
func (list *IntList) Sum() int {
    if list == nil {
        return 0
    }
    return list.Value + list.Tail.Sum()
}
```

Cuando se define un tipo cuyos métodos permiten **nil** como valor receptor, vale la pena señalar esto explícitamente en su comentario de documentación, como lo hicimos anteriormente.

Aquí hay una parte de la definición del tipo **Values** del paquete **net/url**:

```
net/url
package url

// Values asigna una clave de string a una lista de valores.
type Values map[string][]string
// Get devuelve el primer valor asociado con la clave dada,
// o "" si no hay ninguno.
func (v Values) Get(key string) string {
    if vs := v[key]; len(vs) > 0 {
        return vs[0]
    }
    return ""
}
// Add añade el valor de la clave.
// Se añaden a los valores existentes asociados con key.
func (v Values) Add(key, value string) {
    v[key] = append(v[key], value)
}
```

Expone su representación como un mapa, también proporciona métodos para simplificar el acceso al mapa, cuyos valores son **slices** de **strings,** esto es un **multimap**. Sus clientes pueden utilizar sus operadores intrínsecos (**make**, literales **slice**, **m[key]**, etc.), o sus métodos, o ambos, lo que prefieran:

```
gopl.io/ch6/urlvalues
m := url.Values{"lang": {"en"}} // construcción directa
m.Add("item", "1")
m.Add("item", "2")

fmt.Println(m.Get("lang")) // "en"
fmt.Println(m.Get("q")) // ""
fmt.Println(m.Get("item")) // "1" (primer valor)
fmt.Println(m["item"]) // "[1 2]" (mapa de acceso directo)

m=nil
fmt.Println(m.Get("item")) // ""
m.Add("item", "3") // pánico: asignación a la entrada en mapa nil
```

En la última llamada a **Get**, el receptor **nil** se comporta como un mapa vacío. Podríamos haberlo escrito equivalentemente como valores **Values(nil).Get("item"))**, pero **nil.Get("item")** no se compilará porque el tipo de **nil** no se ha determinado. Por el contrario, la llamada final a **Add** entra en pánico, ya que trata de actualizar un mapa **nil**.

Debido a que **url.Values** es un tipo de mapa y un mapa se refiere a sus pares clave/valor indirectamente, las actualizaciones y supresiones que **url.Values.Add** hace a los elementos del mapa son visibles para el que llama. Sin embargo, al igual que con las funciones ordinarias, cualquier cambio en un método lo hace a la propia referencia, como poniéndolo a **nil**, o haciendo que se refiera a una estructura de datos de mapa diferente, no se reflejarán en el que llama.

6.3. Componer tipos por incrustación de struct

considerar el tipo **ColoredPoint**:

```
gopl.io/ch6/coloredpoint
import "image/color"

type Point struct{ X, Y float64 }
```

```
type ColoredPoint struct {
    Point
    Color color.RGBA
}
```

Podríamos haber definido **ColoredPoint** como una estructura de tres campos, pero en cambio, hemos incrustado un **Point** para proporcionar los campos **X** e **Y**. Como vimos en la **Sección 4.4.3**, la incrustación nos permite tomar un atajo sintáctico para la definición de un **ColoredPoint** que contiene todos los campos de **Point**, además de algo más. Si queremos, podemos seleccionar los campos de **ColoredPoint** que fueron aportados por el incrustado **Point** sin mencionar **Point**:

```
var cp ColoredPoint
cp.X = 1
fmt.Println(cp.Point.X) // "1"
cp.Point.Y = 2
fmt.Println(cp.Y) // "2"
```

Un mecanismo similar se aplica a los métodos de **Point**. Podemos llamar a los métodos del campo incrustado **Point** utilizando un receptor de tipo **ColoredPoint**, a pesar de que **ColoredPoint** no tiene métodos declarados:

```
red := color.RGBA{255, 0, 0, 255}
blue := color.RGBA{0, 0, 255, 255}
var p = ColoredPoint{Point{1, 1}, red}
var q = ColoredPoint{Point{5, 4}, blue}
fmt.Println(p.Distance(q.Point)) // "5"
p.ScaleBy(2)
q.ScaleBy(2)
fmt.Println(p.Distance(q.Point)) // "10"
```

Los métodos de **Point** han sido promovidos a **ColoredPoint**. De esta manera, la incrustación permite tipos complejos con muchos métodos construidos por la composición de varios campos, cada uno proporcionando unos métodos.

Los lectores familiarizados con lenguajes orientados a objetos basados en clases pueden verse tentados a ver a **Point** como una clase base y **ColoredPoint** como una subclase o clase derivada, o interpretar la relación entre estos tipos como si un **ColoredPoint** "es un" **Point**. Pero eso sería un error. Observar las llamadas a **Distance** anteriores. **Distance** tiene un parámetro de tipo **Point**, y **q** no es un **Point**, por lo que, aunque **q** tiene un campo incrustado de ese tipo, hay que seleccionarlo explícitamente. Si se intenta pasar **q,** sería un error:

```
p.Distance(q) // Error de compilación: No se puede utilizar q (ColoredPoint) como Point
```

Un **ColoredPoint** no es un **Point**, pero "tiene un" **Point**, y cuenta con dos métodos adicionales **Distance** y **ScaleBy** promovidos desde **Point**. Si se prefiere pensar en términos de implementación, el campo incrustado indica al compilador que genere métodos contenedores adicionales que deleguen a los métodos declarados equivalentes a éstos:

```
func (p ColoredPoint) Distance(q Point) float64 {
    return p.Point.Distance(q)
}
func (p *ColoredPoint) ScaleBy(factor float64) {
    p.Point.ScaleBy(factor)
}
```

Cuando es llamado **Point.Distance** por el primero de estos métodos contenedores, su valor receptor es **p.Point**, no **p**, y no hay manera que el método acceda a **ColoredPoint** donde **Point** está incrustado.

El tipo de un campo anónimo puede ser un puntero a un tipo con nombre, en el que los campos de casos y métodos son promovidos indirectamente del objeto apuntado. La adición de otro nivel de indirección nos permite compartir estructuras comunes y variar las relaciones entre los objetos de forma dinámica. La declaración de **ColoredPoint** a continuación incorpora un ***Point**:

```
type ColoredPoint struct {
    *Point
    Color color.RGBA
}

p := ColoredPoint{&Point{1, 1}, red}
q := ColoredPoint{&Point{5, 4}, blue}
fmt.Println(p.Distance(*q.Point)) // "5"
q.Point = p.Point // p y q ahora comparten el mismo Point
p.ScaleBy(2)
fmt.Println(*p.Point, *q.Point) // "{2 2} {2 2}"
```

Un tipo de estructura puede tener más de un campo anónimo. Si hubiéramos declarado **ColoredPoint** como,

```
type ColoredPoint struct {
    Point
    color.RGBA
}
```

un valor de este tipo tendría todos los métodos de **Point**, todos los métodos de **RGBA**, y cualquier otro método declarado en **ColoredPoint** directamente. Cuando el compilador resuelve un selector tal como **p.ScaleBy** a un método, primero busca un método directamente declarado llamado **Scaleby**, a continuación, en los métodos promovidos una vez desde los campos incrustados en **ColoredPoint**, a continuación, en los métodos promovidos dos veces de campos incrustados dentro **Point** y **RGBA**, etc. El compilador informa de un error si el selector es ambiguo porque dos métodos fueron promovidos con el mismo rango.

Los métodos pueden ser declarados sólo en tipos con nombre (como **Point**) y punteros a ellos (*** Point**), pero gracias a la incrustación, es posible y, a veces útil que los tipos **struct** no identificados tengan también métodos.

Aquí hay un buen truco para ilustrarlo. En este ejemplo se muestra parte de un simple caché implementado utilizando dos variables a nivel de paquete, un **mutex** (**§9.2**) y el mapa que guarda:

```
var (
    mu sync.Mutex // guarda mapeo
    mapping = make(map[string]string)
)

func Lookup(key string) string {
    mu.Lock()
    v := mapping[key]
    mu.Unlock()
    return v
}
```

La versión de abajo es funcionalmente equivalente, pero agrupa a las dos variables relacionadas en una única variable de nivel de paquete, **cache**:

```
var cache = struct {
    sync.Mutex
    mapping map[string]string
} {
    mapping: make(map[string]string),
}

func Lookup(key string) string {
    cache.Lock()
    v := cache.mapping[key]
    cache.Unlock()
    return v
}
```

La nueva variable da nombres más expresivos a las variables relacionadas con **cache**, y porque el campo **sync.Mutex** está incrustado dentro de ella, los métodos **Lock** y **Unlock** son promovidos con el tipo de estructura sin nombre, lo que nos permite bloquear **cache** con una auto explicativa sintaxis.

6.4. Valores de método y expresiones

Por lo general, seleccionamos y llamamos un método en la misma expresión, como en **p.Distance()**, pero es posible separar estas dos operaciones. El selector **p.Distance** obtiene un valor de método, una función que une un método (**Point.Distance**) a un valor receptor específico **p**. Esta función puede ser invocada sin un valor receptor; sólo necesita los argumentos no receptores.

```
p := Point{1, 2}
q := Point{4, 6}
distanceFromP := p.Distance // valor del método
fmt.Println(distanceFromP(q)) // "5"
var origin Point // {0, 0}
fmt.Println(distanceFromP(origin)) // "2.23606797749979", √5
scaleP := p.ScaleBy // valor del método
scaleP(2) // p se convierte en (2, 4)
scaleP(3) // entonces (6, 12)
scaleP(10) // entonces (60, 120)
```

Los valores de método son útiles cuando una **API** de paquete llama a un valor de función, y el comportamiento deseado del cliente para esa función es llamar a un método en un receptor específico. Por ejemplo, la función **time.AfterFunc** llama a un valor de función después de un retardo especificado. Este programa lo utiliza para lanzar el cohete **r** después de **10** segundos:

```
type Rocket struct { /* ... */ }
func (r *Rocket) Launch() { /* ... */ }
r := new(Rocket)
time.AfterFunc(10 * time.Second, func() { r.Launch() })
```

La sintaxis valor método es más corta:

```
time.AfterFunc(10 * time.Second, r.Launch)
```

Relacionado con el valor de método está la expresión de método. Cuando se llama a un método, a diferencia de una función ordinaria, hay que suministrar el receptor de una manera especial usando la sintaxis del selector. Una expresión método, escrito **T.f** o **(*T).f** donde **T** es un tipo, obtiene un valor de función con un primer parámetro normal tomando el lugar del receptor, por lo que puede ser llamado en la forma habitual.

```
p := Point{1, 2}
q := Point{4, 6}
distance := Point.Distance // expresión de método
fmt.Println(distance(p, q)) // "5"
fmt.Printf("%T\n", distance) // "func(Point, Point) float64"
scale := (*Point).ScaleBy
scale(&p, 2)
fmt.Println(p) // "{2 4}"
fmt.Printf("%T\n", scale) // "func(*Point, float64)"
```

Las expresiones de método pueden ser útiles cuando se necesita un valor para representar una elección entre varios métodos que pertenecen al mismo tipo de manera que se pueda llamar al método elegido con muchos receptores diferentes. En el siguiente ejemplo, la variable **op** representa el método de adición o de sustracción de tipo **Point**, y **Path.TranslateBy** lo llama para cada punto de **Path**:

```
type Point struct{ X, Y float64 }

func (p Point) Add(q Point) Point { return Point{p.X + q.X, p.Y + q.Y} }
func (p Point) Sub(q Point) Point { return Point{p.X - q.X, p.Y - q.Y} }

type Path []Point

func (path Path) TranslateBy(offset Point, add bool) {
    var op func(p, q Point) Point
    if add {
        op = Point.Add
    } else {
        op = Point.Sub
    }
    for i := range path {
        // Llamar a cualquiera de path[i].Add(offset) o path[i].Sub(offset).
        path[i] = op(path[i], offset)
    }
}
```

6.5. Ejemplo: tipo vectorial de bit

Los conjuntos en **Go** normalmente se implementan como un **map[T]bool**, donde **T** es el tipo de elemento. Un conjunto representado por un mapa es muy flexible, pero, para ciertos problemas, una representación especializada puede superarla. Por ejemplo, en ámbitos tales como el análisis de flujo de datos donde el conjunto de elementos son pequeños números enteros no negativos, los conjuntos tienen muchos elementos, y las operaciones de conjuntos como unión e intersección son comunes, un vector de bits es ideal.

Un vector de bits utiliza un **slice** de valores enteros sin signo o "palabras,'' cada bit de los cuales representa un posible elemento del conjunto. El conjunto contiene **i** si se establece el i-ésimo bit. El programa siguiente muestra un simple tipo de vector de bit con tres métodos:

```
gopl.io/ch6/intset
// Un IntSet es un conjunto de pequeños números enteros no negativos.
// Su valor cero representa el conjunto vacío.
type IntSet struct {
    words []uint64
}

// Tiene informes si el conjunto contiene el valor no negativo x.
func (s *IntSet) Has(x int) bool {
    word, bit := x/64, uint(x%64)
    return word < len(s.words) && s.words[word]&(1<<bit) != 0
}
// Add añade la suma del valor no negativo x al conjunto.
func (s *IntSet) Add(x int) {
    word, bit := x/64, uint(x%64)
    for word >= len(s.words) {
        s.words = append(s.words, 0)
    }
    s.words[word] |= 1 << bit
}

// UnionWith pone s en la union de s y t.
func (s *IntSet) UnionWith(t *IntSet) {
```

```
        for i, tword := range t.words {
            if i < len(s.words) {
                s.words[i] |= tword
            } else {
                s.words = append(s.words, tword)
            }
        }
    }
```

Dado que cada palabra tiene **64** bits, para localizar el bit de **x**, usamos el cociente **x/64** como el índice de la palabra y el resto **x64%** como índice de bit dentro de esa palabra. La operación **UnionWith** utiliza el operador **OR** binario **|** para calcular la unión de los **64** elementos a la vez.

Esta implementación carece de muchas características deseables, algunos de los cuales se usaron en ejercicios, pero es difícil vivir sin: una forma de imprimir un **IntSet** como un **string**. Vamos a darle un método **String** como lo hicimos con **Celsius** en la **Sección 2.5**:

```
// String devuelve el conjunto como un string de la forma "{1 2 3}".
func (s *IntSet) String() string {
    var buf bytes.Buffer
    buf.WriteByte('{')
    for i, word := range s.words {
        if word == 0 {
            continue
        }
        for j := 0; j < 64; j++ {
            if word&(1<<uint(j)) != 0 {
                if buf.Len() > len("{") {
                    buf.WriteByte(' ')
                }
                fmt.Fprintf(&buf, "%d", 64*i+j)
            }
        }
    }
    buf.WriteByte('}')
    return buf.String()
}
```

Observar la similitud del método **String** anterior con **intsToString** en la **Sección 3.5.4**; **bytes.Buffer** es utilizado a menudo de esta manera en métodos **String**. El paquete **fmt** trata los tipos con un método **String** especialmente para que los valores de los tipos complicados puedan mostrarse de una manera fácil de usar. En lugar de imprimir la representación en bruto del valor (una estructura en este caso), **fmt** llama al método **String**. El mecanismo se basa en las interfaces y tipos aserción, que explicaremos en el **Capítulo 7**.

Ahora podemos demostrar **IntSet** en acción:

```
var x, y IntSet
x.Add(1)
x.Add(144)
x.Add(9)
fmt.Println(x.String()) // "{1 9 144}"

y.Add(9)
y.Add(42)
fmt.Println(y.String()) // "{9 42}"

x.UnionWith(&y)
fmt.Println(x.String()) // "{1 9 42 144}"

fmt.Println(x.Has(9), x.Has(123)) // "true false"
```

Una palabra de precaución: se declaró **String** y **Has** como métodos del tipo puntero ***IntSet** no por necesidad, sino por coherencia con los otros dos métodos, los cuales necesitan un receptor puntero debido a que asignan a **s.words**. En consecuencia, un valor **IntSet** no tiene un método **String**, en ocasiones, conduce a sorpresas como esta:

```
fmt.Println(&x) // "{1 9 42 144}"
fmt.Println(x.String()) // "{1 9 42 144}"
fmt.Println(x) // "{[4398046511618 0 65536]}"
```

En el primer caso, imprimimos un puntero ***IntSet**, que tiene un método **String**. En el segundo caso, llamamos a **String()** en una variable **IntSet**; el compilador inserta la operación implícita **&** que nos da un puntero, que tiene un método **String**. Sin embargo, en el tercer caso, debido a que el valor **IntSet** no tiene un método **String**, **fmt.Println** imprime en su lugar la representación de la estructura. Es importante no olvidar el operador **&**. Haciendo **String** un método de **IntSet**, no ***IntSet**, podría ser una buena idea, pero es un juicio de caso por caso.

6.6. Encapsulación

Una variable o método de un objeto se dice que está encapsulado si no es accesible para los clientes del objeto. La encapsulación, a veces llamada ocultación de información, es un aspecto clave de la programación orientada a objetos.

Go sólo tiene un mecanismo para controlar la visibilidad de nombres: los identificadores en mayúsculas son exportados desde el paquete en el que se definen, y los nombres en minúsculas no. El mismo mecanismo que limita el acceso a los miembros de un paquete también limita el acceso a los campos de una estructura o los métodos de un tipo. Como consecuencia, para encapsular un objeto, hay que hacerlo una estructura.

Esta es la razón por la que el tipo **IntSet** de la sección anterior fue declarado como un tipo de estructura a pesar de que sólo tiene un único campo:

```
type IntSet struct {
    words []uint64
}
```

En su lugar podríamos definir **IntSet** como un tipo de **slice** de la siguiente manera, aunque por supuesto tendríamos que reemplazar cada aparición de **s.words** por ***s** en sus métodos:

```
type IntSet []uint64
```

Aunque esta versión de **IntSet** sería esencialmente equivalente, permitiría a los clientes de otros paquetes leer y modificar el **slice** directamente. Dicho de otra manera, mientras que la expresión ***s** se pueda utilizar en cualquier paquete, **s.words** puede aparecer sólo en el paquete que define **IntSet**.

Otra consecuencia de este mecanismo basado en el nombre es que la unidad de encapsulación es el paquete, no el tipo como en muchos otros lenguajes. Los campos de un tipo de estructura son visibles para todos los códigos en el mismo paquete. Si el código aparece en una función o un método no hay ninguna diferencia.

La encapsulación proporciona tres beneficios. En primer lugar, como los clientes no pueden modificar directamente las variables del objeto, ahora necesitamos inspeccionar un menor número de sentencias para entender los posibles valores de esas variables.

En segundo lugar, ocultar los detalles de implementación evita que los clientes dependan de las cosas que pueden cambiar, lo que da al diseñador una mayor libertad para desarrollar la implementación sin romper la compatibilidad **API**.

Como ejemplo, considerar el tipo **bytes.Buffer**. Se utiliza con frecuencia para acumular secuencias muy cortas, por lo que es una optimización rentable para reservar un poco de espacio adicional en el objeto para evitar la asignación de memoria en este caso común. Como **Buffer** es un tipo de estructura, este espacio tiene la forma de un campo adicional de tipo **[64]byte** con un nombre en minúsculas. Cuando se añadió este campo, como no se exportó, los clientes de **Buffer** fuera del paquete **bytes** no estaban al tanto del cambio, excepto un rendimiento mejorado. **Buffer** y su método **Grow** se muestran a continuación, simplificado para mayor claridad:

```
type Buffer struct {
    buf []byte
    initial [64]byte
    /* ... */
}

// Grow amplía la capacidad del buffer, si es necesario,
// para garantizar espacio para otro n bytes. [...]
func (b *Buffer) Grow(n int) {
    if b.buf == nil {
        b.buf = b.initial[:0] // utilizar el espacio preasignado inicialmente
    }
    if len(b.buf)+n > cap(b.buf) {
        buf := make([]byte, b.Len(), 2*cap(b.buf) + n)
        copy(buf, b.buf)
        b.buf = buf
    }
}
```

El tercer beneficio de encapsulación, y en muchos casos lo más importante, es que evita que los clientes definan variables de un objeto arbitrariamente. Debido a que las variables del objeto solamente pueden establecerse por funciones en el mismo paquete, el autor de ese paquete puede garantizar que todas las funciones mantengan los objetos invariantes internamente. Por ejemplo, el tipo **Counter** a continuación es válido para que los clientes puedan incrementar el contador o para que puedan restablecerlo a cero, pero no para establecerlo a algún valor arbitrario:

```
type Counter struct { n int }

func (c *Counter) N() int { return c.n }
func (c *Counter) Increment() { c.n++ }
func (c *Counter) Reset() { c.n = 0 }
```

Las funciones que simplemente acceden o modifican los valores internos de un tipo, tales como los métodos del tipo **Logger** del paquete **log**, a continuación, se llaman **getters** y **setters**. Sin embargo, al nombrar a un método **getter**,

por lo general omitimos el prefijo **Get**. Esta preferencia por la brevedad se extiende a todos los métodos, no sólo los descriptores de acceso de campo, también para otros prefijos redundantes, así como **Fetch**, **Find**, y **Lookup**.

```
package log
type Logger struct {
    flags int
    prefix string
    // ...
}

func (l *Logger) Flags() int
func (l *Logger) SetFlags(flag int)
func (l *Logger) Prefix() string
func (l *Logger) SetPrefix(prefix string)
```

El estilo de **Go** no prohíbe campos exportados. Por supuesto, una vez exportados, un campo puede no ser des-exportado sin un cambio incompatible con la **API**, así que la elección inicial debe ser deliberada y debe tener en cuenta la complejidad de los invariantes que se deben mantener, la probabilidad de cambios en el futuro, y la cantidad del código de cliente que se vería afectado por un cambio.

La encapsulación no siempre es deseable. Al revelar su representación como un **int64** número de nanosegundos, **time.Duration** nos permite usar todas las operaciones aritméticas y de comparación con duraciones habituales, e incluso definir las constantes de este tipo:

```
const day = 24 * time.Hour
fmt.Println(day.Seconds()) // "86400"
```

Como otro ejemplo, contrastar **IntSet** con el tipo **geometry.Path** del comienzo de este capítulo. **Path** se definió como un tipo de **slice**, permitiendo a sus clientes construir instancias utilizando la sintaxis literal de **slice**, para iterar sobre sus puntos usando un bucle **range**, y etc., mientras que estas operaciones se les niega a los clientes de **IntSet**.

Aquí está la diferencia crucial: **geometry.Path** es intrínsecamente una secuencia de puntos, ni más ni menos, y no previene la adición de nuevos campos a la misma, así que tiene sentido para el paquete **geometry** revelar que **Path** es un **slice**. Por el contrario, un **IntSet** simplemente pasa a ser representado como un **slice []uint64**. Podría haber sido representado mediante **[]uint**, o algo completamente diferente para los conjuntos que son escasos o muy pequeños, y quizás podría beneficiarse de características adicionales como un campo adicional para registrar el número de elementos en el conjunto. Por estas razones, tiene sentido para **IntSet** ser opaco.

En este capítulo, hemos aprendido cómo asociar métodos con los tipos con nombre, y cómo llamar a esos métodos. Aunque los métodos son cruciales para la programación orientada a objetos, son sólo la mitad de la imagen. Para completarla, necesitamos las **interfaces**, el tema del siguiente capítulo.