6

Методы

С начала 1990-х годов объектно-ориентированное программирование (ООП) стало доминирующей парадигмой программирования в промышленности и образовании, и почти все широко используемые языки, разработанные с того времени, включают его поддержку. Не является исключением и Go.

Хотя общепринятого определения объектно-ориентированного программирования нет, для наших целей определим, что *объект* представляет собой просто значение или переменную, которая имеет методы, а *метод* — это функция, связанная с определенным типом. Объектно-ориентированная программа — это программа, которая использует методы для выражения свойств и операций каждой структуры данных так, что клиентам не требуется прямой доступ к представлению объекта.

В предыдущих главах мы регулярно использовали методы из стандартной библиотеки, такие как метод `Seconds` типа `time.Duration`:

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

Мы также определяли собственный метод в разделе 2.5 — метод `String` типа `Celsius`:

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

В этой главе, первой из двух, посвященных объектно-ориентированному программированию, мы покажем, как эффективно определять и использовать методы. Мы также рассмотрим два ключевых принципа объектно-ориентированного программирования — *инкапсуляцию* и *композицию*.

6.1. Объявления методов

Метод объявляется с помощью вариации объявления обычных функций, в котором перед именем функции появляется дополнительный параметр. Этот параметр присоединяет функцию к типу этого параметра.

Давайте напишем наш первый метод в простом пакете для геометрии на плоскости:

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

```
import "math"

type Point struct{ X, Y float64 }

// Традиционная функция
func Distance(p, q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

// То же, но как метод типа Point
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}
```

Дополнительный параметр `p` называется *получателем* метода приемника; это название унаследовано от ранних объектно-ориентированных языков, которые описывали вызов метода как "отправку сообщения объекту".

В Go не используется специальное имя, такое как `this` или `self`, для получателя; мы выбираем имя для получателя так же, как для любого другого параметра. Поскольку имя получателя будет использоваться часто, лучше выбрать что-то короткое и согласованно используемое во всех методах. Распространенным выбором является первая буква имени типа, как выше использовано имя `p` для `Point`.

В вызове метода аргумент получателя находится перед именем метода. Это напоминает объявление, в котором параметр получателя находится перед именем метода:

```
p := Point{1, 2}
q := Point{4, 6}
fmt.Println(Distance(p, q)) // "5", вызов функции
fmt.Println(p.Distance(q))  // "5", вызов метода
```

Нет никакого конфликта между двумя объявлениями функций по имени `Distance`, приведенными выше. Первое объявляет функцию уровня пакета под названием `geometry.Distance`. Второе объявляет метод типа `Point`, поэтому его имя — `Point.Distance`.

Выражение `P.Distance` называется *селектором* (selector), потому что оно выбирает подходящий метод `Distance` для получателя `p` типа `Point`. Селекторы используются также для выбора полей структурных типов, как в выражении `p.X`. Поскольку методы и поля находятся в одном и том же пространстве имен, объявление в нем метода `X` для структуры типа `Point` приведет к неоднозначности, и компилятор его отвергнет.

Поскольку каждый тип имеет собственное пространство имен для методов, мы можем использовать имя `Distance` для других методов, лишь бы они принадлежали различным типам. Давайте определим тип `Path`, который представляет собой последовательность отрезков линии, и определим метод `Distance` и для него:

```
// Path — путь из точек, соединенных прямолинейными отрезками.
type Path []Point
```

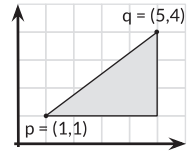
```
// Distance возвращает длину пути.
func (path Path) Distance() float64 {
    sum := 0.0
    for i := range path {
        if i > 0 {
            sum += path[i-1].(path[i])
        }
    }
    return sum
}
```

`Path` является именованным типом среза, а не структурой, как `Point`, но мы по-прежнему можем определить для него методы. Разрешая связывать методы с любым типом, Go отличается от многих других объектно-ориентированных языков программирования. Часто бывает удобно определить дополнительное поведение для простых типов, таких как числа, строки, срезы, отображения, а иногда даже функции. Методы могут быть объявлены для любого именованного типа, определенного в том же пакете, лишь бы его базовый тип не являлся ни указателем, ни интерфейсом.

Два рассмотренных метода `Distance` имеют различные типы. Они не связаны один с другим, хотя `Path.Distance` использует `Point.Distance` внутренне для вычисления длины каждого отрезка, соединяющего соседние точки.

Давайте вызовем новый метод для вычисления периметра прямоугольного треугольника:

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



В двух показанных выше вызовах методов `Distance` компилятор определяет, какая функция должна быть вызвана, основываясь на имени метода и типе получателя. В первом вызове `path[i-1]` имеет тип `Point`, поэтому вызывается `Point.Distance`; во втором `perim` имеет тип `Path`, поэтому вызывается метод `Path.Distance`.

Все методы данного типа должны иметь уникальные имена, но одно и то же имя метода может использоваться разными типами, как метод `Distance` использован для типов `Point` и `Path`. Нет необходимости квалифицировать имена функций (например, `PathDistance`) для устранения неоднозначности. Здесь мы видим первое преимущество использования методов по сравнению с обычными функциями: имена методов могут быть короче. Преимущество увеличивается для вызовов, выполняемых за пределами пакета, так как они могут использовать более короткое имя *и* опускать имя пакета:

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

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

6.2. Методы с указателем в роли получателя

Вызов функции создает копию каждого значения аргумента, поэтому, если функции необходимо обновить переменную или если аргумент является настолько большим, что желательно избежать его копирования, следует передавать адрес переменной с помощью указателя. То же самое справедливо и для методов, которым необходимо обновить переменную получателя: их следует присоединять к типу указателя, такому как `*Point`:

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

Именем данного метода является is `(*Point).ScaleBy`. Скобки необходимы — без них выражение будет трактоваться как `*(Point.ScaleBy)`.

В реальных программах соглашение диктует, что если какой-либо метод `Point` имеет получатель-указатель, то *все* методы `Point` должны иметь указатель в качестве получателя, даже те, которым это не требуется в обязательном порядке. Мы нарушили это правило для `Point`, чтобы показать вам обе разновидности методов.

Именованные типы (`Point`) и указатели на них (`*Point`) — единственные типы, которые могут появляться в объявлении получателя. Кроме того, чтобы избежать неоднозначности, объявления методов не разрешены для именованных типов, которые сами являются типами указателей:

```
type P *int
func (P) f() {/* ... */} // Ошибка компиляции: неверный тип получателя
```

Метод `(*Point).ScaleBy` может быть вызван с помощью предоставления получателя `*Point`, например, так:

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

Или так:

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

Или вот так:

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

Но последние два варианта несколько громоздки. К счастью, здесь нам помогает язык. Если получатель `p` является *переменной* типа `Point`, но методу необходим получатель `*Point`, можно использовать сокращенную запись

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

при которой компилятор выполнит неявное получение адреса `&p` этой переменной. Это работает только для переменных, включая поля структур наподобие `p.X` и элементы массивов или срезов наподобие `perim[0]`. Мы не можем вызвать метод `*Point` для неадресуемого получателя `Point`, так как нет никакого способа получения адреса временного значения.

```
Point{1, 2}.ScaleBy(2) // Ошибка компиляции: невозможно получить
                       // адрес литерала Point
```

Но мы *можем* вызвать метод `Point` наподобие `Point.Distance` с получателем `*Point`, поскольку есть способ получить значение из адреса: нужно просто загрузить значение, на которое указывает получатель. Компилятор, по сути, вставляет неявный оператор *. Два следующих вызова являются эквивалентными:

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

Подведем итоги, так как недостаточное понимание в этой области зачастую приводит к путанице. В каждом корректном выражении вызова метода истинным является только одно из трех перечисленных утверждений.

Либо аргумент получателя имеет тот же тип, что и параметр получателя, например оба имеют тип `T` или оба имеют тип `*T`:

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

Либо аргумент получателя является *переменной* типа `T`, а параметр получателя имеет тип `*T`. Компилятор неявно получает адрес переменной:

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

Либо аргумент получателя имеет тип `*T`, а параметр получателя имеет тип `T`. Компилятор неявно разыменовывает получателя, другими словами, загружает соответствующее значение:

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

Если все методы именованного типа `T` имеют тип получателя `T` (не `*T`), то копирование экземпляров этого типа безопасно; вызов любого из его методов обязательно делает копию. Например, значения `time.Duration` свободно копируются, в том числе в качестве аргументов функций. Но если какой-либо метод имеет в качестве получателя указатель, следует избегать копирования экземпляров `T`, потому

что это может нарушать внутренние инварианты. Например, копирование экземпляра `bytes.Buffer` может привести к тому, что оригинал и копия будут псевдонимами (раздел 2.3.2) одного и того же базового массива байтов, и последующие вызовы методов будут иметь непредсказуемые результаты.

6.2.1. Значение `nil` является корректным получателем

Так же, как некоторые функции допускают нулевые указатели в качестве аргументов, так и некоторые методы допускают нулевые указатели в качестве их получателя, особенно если `nil` является полноценным нулевым значением типа, как в случае отображений и срезов. В приведенном далее простом связанном списке целых чисел значение `nil` представляет пустой список:

```
// IntList представляет собой связанный список целых чисел.
// Значение *IntList, равное nil, представляет пустой список.
type IntList struct {
    Value int
    Tail *IntList
}

// Sum возвращает сумму элементов списка.
func (list *IntList) Sum() int {
    if list == nil {
        return 0
    }
    return list.Value + list.Tail.Sum()
}
```

При определении типа, методы которого допускают нулевое значение получателя, желательно явно указать это в документирующих комментариях, как это сделано выше.

Вот часть определения типа `Values` из пакета `net/url`:

net/url

```
package url

// Values отображает строковый ключ на список значений.
type Values map[string][]string

// Get возвращает первое значение, связанное с данным ключом,
// или "", если такового нет.
func (v Values) Get(key string) string {
    if vs := v[key]; len(vs) > 0 {
        return vs[0]
    }
    return ""
}
```

```
// Add добавляет значение к ключу. Добавление выполняется к любым
// существующим значениям, связанным с ключом.
func (v Values) Add(key, value string) {
    v[key] = append(v[key], value)
}
```

Оно показывает, что представлением данного типа является отображение, а также предоставляет методы для доступа к отображению, значениями которого являются срезы строк — это *мультиотображение*. Клиенты могут использовать его встроенные операторы (`make`, литералы срезов, `m[key]` и т.д.) или его методы, или и то, и другое вместе, что им больше нравится:

gopl.io/ch6/urlvalues

```
m := url.Values{"lang": {"en"}} // Непосредственное создание
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" (первое значение)
fmt.Println(m["item"])     // "[1 2]" (непосредственное обращение)

m = nil
fmt.Println(m.Get("item")) // ""
m.Add("item", "3")         // Аварийная ситуация: присваивание
                           // записи в пустом отображении
```

В последнем вызове `Get` нулевой получатель ведет себя, как пустое отображение. Мы могли бы эквивалентно записать вызов как `Values(nil).Get("Item")`, но выражение `nil.Get("item")` не будет компилироваться, потому что тип `nil` не определен. Последний же вызов `Add` приводит к аварийной ситуации, поскольку он пытается обновить нулевое отображение.

Поскольку `url.Values` имеет тип отображения, а отображение обращается к своим парам "ключ–значение" косвенно, любые обновления и удаления, которые делают вызовы `url.Values.Add` с элементами отображений, видны вызывающей функции. Однако, как и в случае обычных функций, любые изменения, которые метод делает с самой ссылкой, такие как установка ее значения равным `nil` или ее перенаправление на другое отображение, не будет видимо вызывающей функции.

6.3. Создание типов путем встраивания структур

Рассмотрим тип `ColoredPoint`:

gopl.io/ch6/coloredpoint

```
import "image/color"
```

```
type Point struct{ X, Y float64 }
type ColoredPoint struct {
    Point
    Color color.RGBA
}
```

Мы могли бы определить `ColoredPoint` как структуру из трех полей, но вместо этого *встраиваем* `Point` для предоставления полей `X` и `Y`. Как мы видели в разделе 4.4.3, встраивание позволяет нам использовать синтаксические сокращения для определения структуры `ColoredPoint`, которая содержит все поля `Point` плюс еще некоторые. При желании мы можем выбрать поля `ColoredPoint`, которые были предоставлены встроенной структурой `Point` без упоминания имени `Point`:

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

Аналогичный механизм применим и к *методам* `Point`. Мы можем вызывать методы встроенного поля `Point` с использованием получателя типа `ColoredPoint`, несмотря на то что `ColoredPoint` не имеет объявленных методов:

```
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"
```

Методы `Point` были *повышены* до методов `ColoredPoint`. Таким образом, встраивание допускает наличие сложных типов со многими методами, при этом указанные типы создаются путем *композиции* нескольких полей, каждое из которых предоставляет несколько методов.

Читатели, знакомые с объектно-ориентированными языками, основанными на классах, могут соблазниться провести параллели и рассматривать `Point` как базовый класс, а `ColoredPoint` — как подкласс (или производный класс) или интерпретировать связь между этими типами, как если бы `ColoredPoint` “являлся” `Point`. Но это было бы ошибкой. Обратите внимание на показанные выше вызовы `Distance`. Метод `Distance` имеет параметр типа `Point`, но `q` не является `Point`, так что хотя `q` и имеет встроенные поля этого типа, мы должны явно их указать. Попытка передачи `q` приведет к сообщению об ошибке:

```
p.Distance(q) // Ошибка компиляции: нельзя использовать
              // q (ColoredPoint) как Point
```

`ColoredPoint` не является `Point`, но "содержит" `Point` и имеет два дополнительных метода (`Distance` и `ScaleBy`), повышенных из `Point`. Если вы предпочитаете думать в терминах реализации, встроенное поле указывает компилятору на необходимость генерации дополнительных "методов-оберток", которые делегируют вызов объявленным методам что-то вроде такого кода:

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

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

Когда `Point.Distance` вызывается первым из этих методов-оберток, значением его получателя является `p.Point`, а не `p`, и при этом нет никакого способа, которым метод мог бы обратиться к структуре `ColoredPoint`, в которую встроена структура `Point`.

Типом анонимного поля может быть указатель на именованный тип; в этом случае поля и методы косвенно повышаются из указываемого объекта. Добавление еще одного уровня косвенности позволяет нам совместно использовать общие структуры и динамически изменять взаимоотношения между объектами. Приведенное ниже объявление `ColoredPoint` встраивает `*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 и q разделяют одну и ту же Point
p.ScaleBy(2)
fmt.Println(*p.Point, *q.Point)   // "{2 2} {2 2}"
```

Структурный тип может иметь более одного анонимного поля. Если мы объявим `ColoredPoint` как

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

то значение этого типа будет иметь все методы типа `Point`, все методы `RGBA` и любые дополнительные методы, непосредственно объявленные для `ColoredPoint`. Когда компилятор разрешает селектор как `p.ScaleBy` для вызова метода, сначала он ищет непосредственно объявленный метод с именем `ScaleBy`, затем — метод, однократно повышенный из встроенных полей `ColoredPoint`, после этого — дважды повы-

шенный метод из встроенных внутри `Point` и `RGBA`, и т.д. Компилятор сообщает об ошибке, если селектор является неоднозначным из-за того, что имеется два или более методов с одним именем с одинаковым рангом повышения.

Методы могут быть объявлены только для именованных типов (наподобие `Point`) и указателей на них (`*Point`), но благодаря встраиванию *неименованные* структурные типы также могут иметь методы (иногда это оказывается полезным).

Вот неплохая иллюстрация к сказанному. В приведенном примере показана часть простого кеша, реализованного с помощью двух переменных уровня пакета, мьютекса (раздел 9.2) и отображения, которое он защищает:

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

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

Приведенная ниже версия функционально эквивалентна, но группирует эти две связанные переменные вместе в одну переменную уровня пакета `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
}
```

Новая переменная дает связанным с кешем переменным более выразительные имена, а поскольку в нее встроено поле `sync.Mutex`, его методы `Lock` и `Unlock` повышаются до неименованного структурного типа, позволяя нам блокировать `cache` с помощью самоочевидного синтаксиса.

6.4. Значения-методы и выражения-методы

Обычно мы выбираем и вызываем метод в одном выражении, как, например, в `p.Distance()`, но эти две операции можно разделить. Селектор `p.Distance` дает

нам *значение-метод* (method value), функцию, которая связывает метод (`Point.Distance`) со значением конкретного получателя `p`. Эта функция может быть вызвана без указания значения получателя; ей нужны только аргументы, не являющиеся получателем.

```
p := Point{1, 2}
q := Point{4, 6}

distanceFromP := p.Distance        // Значение-метод
fmt.Println(distanceFromP(q))      // "5"
var origin Point                   // {0, 0}
fmt.Println(distanceFromP(origin)) // "2.23606797749979", √5

scaleP := p.ScaleBy                // Значение-метод
scaleP(2)                          // p становится (2, 4)
scaleP(3)                          // затем (6, 12)
scaleP(10)                         // затем (60, 120)
```

Значения-методы полезны, когда API пакета требует значение-функцию, а для клиента желаемым поведением для этой функции является вызов метода для конкретного получателя. Например, функция `time.AfterFunc` вызывает значение-функцию после заданной задержки. Приведенная программа использует ее для запуска ракеты `r` через 10 секунд:

```
type Rocket struct { /* ... */ }
func (r *Rocket) Launch() { /* ... */ }

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

Синтаксис с использованием значения-метода оказывается более коротким:

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

Со значениями-методами тесно связаны *выражения-методы*. При вызове метода, в противоположность обычной функции, мы должны указать получателя с помощью синтаксиса селектора. Выражение-метод, записываемое как `T.f` или `(*T).f`, где `T` — тип, дает значение-функцию с обычным первым параметром, представляющим собой получатель, так что его можно вызывать, как обычную функцию:

```
p := Point{1, 2}
q := Point{4, 6}

distance := Point.Distance   // Выражение-метод
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)"
```

Выражения-методы могут быть полезны, когда требуется значение для представления выбора среди нескольких методов, принадлежащих одному типу, такое, чтобы выбранный метод можно было вызвать со многими различными получателями. В следующем примере переменная `op` представляет метод сложения либо вычитания типа `Point`, и `Path.TranslateBy` называет его для каждой точки в пути `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 {
        // Вызов либо path[i].Add(offset), либо path[i].Sub(offset).
        path[i] = op(path[i], offset)
    }
}
```

6.5. Пример: тип битового вектора

Множества в Go обычно реализуются как `map[T]bool`, где `T` является типом элемента. Множество, представленное отображением, очень гибкое, но для некоторых задач специализированное представление может его превзойти. Например, в таких областях, как анализ потока данных, где элементы множества представляют собой небольшие неотрицательные целые числа, множества имеют много элементов, а распространенными операциями являются объединение и пересечение множеств, идеальным решением оказывается *битовый вектор*.

Битовый вектор использует срез беззнаковых целочисленных значений, или "слов", каждый бит которых представляет возможный элемент множества. Множество содержит *i*, если *i*-й бит установлен. Приведенная далее программа демонстрирует простой тип битового вектора с тремя методами:

gopl.io/ch6/intset

```
// IntSet представляет собой множество небольших неотрицательных
// целых чисел. Нулевое значение представляет пустое множество.

type IntSet struct {
    words []uint64
```

```
}

// Has указывает, содержит ли множество неотрицательное значение 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 добавляет неотрицательное значение x в множество.
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 делает множество s равным объединению множеств s и 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)
        }
    }
}
```

Поскольку каждое слово имеет 64 бита, чтобы найти бит для значения `x`, мы используем частное `x/64` в качестве индекса слова, а остаток `x%64` — как индекс бита внутри этого слова. Операция `UnionWith` использует оператор побитового ИЛИ (`|`) для вычисления объединения 64 элементов за один раз. (Мы вернемся к выбору 64-битовых слов в упражнении 6.5.)

В данной реализации не хватает многих функций, которые хотелось бы иметь. Некоторые из них оставлены в качестве упражнений читателям, но без одной из них очень трудно обойтись: это вывод множества `IntSet` в виде строки. Давайте добавим к этому типу метод `String`, как мы уже делали для типа `Celsius` в разделе 2.5:

```
// String возвращает множество как строку вида "{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()
}
```

Обратите внимание на схожесть метода `String` в приведенном выше исходном тексте с `intsToString` в разделе 3.5.4; `bytes.Buffer` часто используется таким образом в методах `String`. Пакет `fmt` рассматривает типы с методом `String` специальным образом, так, чтобы значения сложных типов можно было выводить в удобочитаемом виде. Вместо вывода неформатированного представления значения (в данном случае — структуры) `fmt` вызывает метод `String`. Этот механизм опирается на интерфейсы и утверждения о типах, с которыми мы встретимся в главе 7.

Теперь можно продемонстрировать `IntSet` в действии:

```
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"
```

Небольшое предостережение: мы объявили `String` и `Has` как методы типа указателя `*IntSet` не по необходимости, а для обеспечения согласованности с двумя другими методами, которым в качестве получателя нужен указатель, поскольку в этих методах выполняется присваивание `s.words`. Следовательно, *значение* `IntSet` не имеет метода `String`, что иногда приводит к таким сюрпризам, как этот:

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

В первом случае мы выводим указатель `*IntSet`, у которого есть метод `String`. Во втором случае мы вызываем `String()` для переменной `IntSet`; компилятор вставляет неявную операцию `&`, давая нам указатель, который имеет метод `String`. Но в третьем случае, поскольку значение `IntSet` не имеет метода `String`, `fmt.`

`Println` выводит представление структуры. Важно не забывать оператор `&`. Сделать `String` методом `IntSet`, а не `*IntSet`, может быть хорошей идеей, но это зависит от конкретных обстоятельств.

Упражнение 6.1. Реализуйте следующие дополнительные методы:

```
func (*IntSet) Len() int      // Возвращает количество элементов
func (*IntSet) Remove(x int)  // Удаляет x из множества
func (*IntSet) Clear()        // Удаляет все элементы множества
func (*IntSet) Copy() *IntSet // Возвращает копию множества
```

Упражнение 6.2. Определите вариативный метод `(*IntSet).AddAll(...int)`, который позволяет добавлять список значений, например `s.AddAll(1,2,3)`.

Упражнение 6.3. `(*IntSet).UnionWith` вычисляет объединение двух множеств с помощью оператора `|`, побитового оператора ИЛИ. Реализуйте методы `IntersectWith`, `DifferenceWith` и `SymmetricDifference` для соответствующих операций над множествами. (Симметричная разность двух множеств содержит элементы, имеющиеся в одном из множеств, но не в обоих одновременно.)

Упражнение 6.4. Добавьте метод `Elems`, который возвращает срез, содержащий элементы множества и годящийся для итерирования с использованием цикла по диапазону `range`.

Упражнение 6.5. Типом каждого слова, используемого в `IntSet`, является `uint64`, но 64-разрядная арифметика может быть неэффективной на 32-разрядных платформах. Измените программу так, чтобы она использовала тип `uint`, который представляет собой наиболее эффективный беззнаковый целочисленный тип для данной платформы. Вместо деления на 64 определите константу, в которой хранится эффективный размер `uint` в битах, 32 или 64. Для этого можно воспользоваться, возможно, слишком умным выражением `32<<(^uint(0)>>63)`.

6.6. Инкапсуляция

Переменная (или метод объекта) называется *инкапсулированной*, если она недоступна клиенту этого объекта. Инкапсуляция, иногда именуемая *сокрытием информации*, является ключевым аспектом объектно-ориентированного программирования.

Go имеет только один механизм для управления видимостью имен: идентификаторы, начинающиеся с прописной буквы, экспортируются из пакета, в котором они определены, а начинающиеся со строчной буквы — нет. Такой же механизм, как и ограничивающий доступ к членам пакета, ограничивает доступ и к полям структуры или методам типа. Как следствие, для инкапсуляции объекта мы должны сделать его структурой.

По этой причине тип `IntSet` из предыдущего раздела был объявлен как структурный, несмотря на то что в нем имеется только одно поле:

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

Мы могли бы вместо этого определить `IntSet` как тип среза, как показано ниже, хотя, конечно, должны были бы при этом заменить в его методах каждое вхождение `s.words` на `*s`:

```
type IntSet []uint64
```

Хотя эта версия `IntSet` была бы, по сути, эквивалентна имеющейся, она позволяла бы клиентам из других пакетов читать и модифицировать срез непосредственно. Иначе говоря, в то время как выражение `*s` может быть использовано в любом пакете, `s.words` может появиться только в пакете, в котором определен `IntSet`.

Еще одним следствием механизма экспорта, основанного на именах, является то, что единицей инкапсуляции является пакет, а не тип, как во многих других языках программирования. Поля структурного типа являются видимыми для всего кода в том же пакете. Находится ли этот код в функции или методе, не имеет никакого значения.

Инкапсуляция имеет три преимущества. Во-первых, поскольку клиенты не могут изменять переменные объекта непосредственно, необходимо изучать меньше инструкций, чтобы понять, какими могут быть возможные значения этих переменных.

Во-вторых, сокрытие деталей реализации устраняет зависимость клиентов от сущностей, которые могут изменяться, что дает проектировщику большую свободу в развитии реализации без нарушения совместимости API.

В качестве примера рассмотрим тип `bytes.Buffer`. Он часто используется для накопления очень коротких строк, так что выгодной оптимизацией является резервирование некоторого дополнительного пространства в объекте, чтобы избежать излишнего перераспределения памяти в этом распространенном случае. Поскольку `Buffer` представляет собой структурный тип, это пространство принимает вид дополнительного поля `[64]byte` с именем, начинающимся со строчной буквы. После добавления этого поля в силу его неэкспортируемости клиенты `Buffer` вне пакета `bytes` ничего не знают о каких-либо изменениях, за исключением повышения производительности. `Buffer` и его метод `Grow` приведены ниже (немного упрощены для ясности):

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

// Grow увеличивает при необходимости емкость буфера,
// чтобы гарантировать наличие места для еще n байтов. [...]
func (b *Buffer) Grow(n int) {
    if b.buf == nil {
        b.buf = b.initial[:0] // Изначально используется
    }                         // предварительно выделенная память.
    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
    }
}
```

Третье, и во многих случаях наиболее важное, преимущество инкапсуляции состоит в том, что она не позволяет клиентам произвольным образом устанавливать значения переменных объекта. Поскольку переменные объекта могут устанавливаться только функциями из одного пакета, автор этого пакета может гарантировать, что все функции поддерживают внутренние инварианты объектов. Например, показанный ниже тип `Counter` позволяет клиентам выполнять приращение счетчика или сбрасывать его значение до нуля, но не устанавливать его равным некоторому произвольному значению:

```
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 }
```

Функции, которые просто получают доступ ко внутренним значениям типа или изменяют их, такие как методы типа `Logger` из пакета `log`, показанные ниже, называются методами *получения* и *установки* значения (getter и setter). Однако при именовании метода получения значения мы обычно опускаем префикс `Get`. Это предпочтение краткости относится ко всем методам (не только к методам доступа к полям), а также к прочим избыточным префиксам, таким как `Fetch`, `Find` или `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)
```

Стиль Go не запрещает экспортировать поля. Конечно, после экспорта поле не может стать неэкспортируемым без внесения несовместимых изменений в API, поэтому первоначальный выбор должен быть преднамеренным. Также должны быть тщательно рассмотрены вопрос о сложности инвариантов, которые должны поддерживаться, вероятность будущих изменений и количество клиентского кода, который будет затронут внесением изменений.

Инкапсуляция желательна не всегда. Открывая представление числа наносекунд как `int64`, `time.Duration` позволяет использовать все обычные арифметические операции и операции сравнения при работе с периодами времени и даже для определения констант этого типа:

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

В качестве другого примера сравним `IntSet` с типом `geometry.Path` в самом начале этой главы. `Path` был определен как тип среза и позволяет своим клиентам создавать экземпляры с использованием синтаксиса литералов срезов, выполнять итерации по его точкам с помощью цикла по диапазону и так далее, тогда как для клиентов `IntSet` эти операции недоступны.

Вот принципиальное отличие: `geometry.Path`, по сути, является последовательностью точек, не больше и не меньше, и добавление в него новых полей не предвидится, так что для пакета `geometry` имеет смысл показать, что `Path` является срезом. В противоположность этому `IntSet` просто случайно представлен в виде среза `[]uint64`. Он мог бы иметь представление с использованием `[]uint` или чего-то совершенно иного для разреженных или очень малых множеств. Или, быть может, определенное преимущество дадут дополнительные возможности, такие как еще одно поле для записи количества элементов множества. По этим причинам имеет смысл сделать `IntSet` непрозрачным.

Из этой главы вы узнали, как связать методы с именованными типами и как вызывать эти методы. Хотя методы имеют решающее значение для объектно-ориентированного программирования, они представляют только половину картины. Чтобы завершить ее, нужны *интерфейсы*. О них речь пойдет в следующей главе.