Programming in Go

Matt Holiday
Christmas 2020

Odds and Ends

Enumerated types

There are no real enumerated types in Go

You can make an almost-enum type using a named type and constants:

```
type shoe int

const (
    tennis shoe = iota
    dress
    sandal
    clog
)
```

`iota` starts at 0 in each `const` block and increments once on each line;
here 0, 1, 2, ...

Enumerated types

Traditional flags are easy:

```
type Flags uint

const (
    FlagUp Flags = 1 << iota    // is up
    FlagBroadcast               // supports broadcast access
    FlagLoopback                // is a loopback interface
    FlagPointToPoint            // is a point-to-point link
    FlagMulticast               // supports multicast access
)
```

These flags take on the values in a power-of-two sequence: `0x01`, `0x02`, `0x04`, etc.

That makes them easy to combine, e.g. `FlagUp | FlagLoopback`

Enumerated types

Go also supports more complex `iota` expressions:

```
type ByteSize int64

const (
    _           = iota                  // ignore first value
    KiB ByteSize = 1 << (10 * iota)     // 2^10
    MiB                                 // 2^20 (1 << 10*2)
    GiB
    TiB
    PiB
    EiB
)
```

So `EiB` is set to $2^{60} = 1152921504606846976 \approx 10^{19}$

Variable argument lists

What if we don't know how many parameters a function needs?

```
fmt.Printf("%#v\n", myMap)

fmt.Printf("%s: %s\n", type, quantity)

a := sum(1, 2, 3)

b := sum(1, 2, 3, 4, 5)
```

All the formatted printing code uses variable argument lists

Variable argument lists

We use a special operator `...` before the parameter type

```
func sum(nums ...int) int {
    var total int

    for _, num := range nums {
        total += num
    }

    fmt.Printf("+/%v=%d\n", nums, total)
    return total
}

// prints +/[1 2 3 4 5] = 15
```

Only the **last** parameter may have this operator

Variable argument lists

Since the parameter looks like a slice, we can pass a slice

```
func main() {
    fmt.Println(add())
    fmt.Println(add(11))
    fmt.Println(add(1, 2, 3, 4))

    s := []int{1, 2, 3}

    fmt.Println(add(s...))
}

// prints 0, 11, 10, 6
```

The special operator `...` *after* the actual parameter “unpacks” it into the variable argument list

Sized integers

Sometimes we need to handle low-level protocols (TCP/IP, etc.)

```
type TCPFields struct {
    SrcPort    uint16
    DstPort    uint16
    SeqNum     uint32
    AckNum     uint32
    DataOffset uint8
    Flags      uint8
    WindowSize uint16
    Checksum   uint16
    UrgentPtr  uint16
}
```

So we need to work with integers that have a particular size and/or are unsigned

Bitwise operators

We can mask off bits inside a byte or word

```
package main
import "fmt"

func main() {
    a, b := uint16(65535), uint16(281)

    fmt.Printf("%016b %#04[1]x\n", a)             // 1111111111111111 0xffff
    fmt.Printf("%016b %#04[1]x\n", a &^ 0b1111)   // 1111111111110000 0xfff0
    fmt.Printf("%016b %#04[1]x\n", a &  0b1111)   // 0000000000001111 0x000f

    fmt.Printf("%016b %#04[1]x\n", b)             // 0000000100011001 0x0119
    fmt.Printf("%016b %#04[1]x\n", ^b)            // 1111111011100110 0xfee6
    fmt.Printf("%016b %#04[1]x\n", b |  0b1111)   // 0000000100011111 0x011f
    fmt.Printf("%016b %#04[1]x\n", b ^  0b1111)   // 0000000100010110 0x0116
}
```

Bitwise operators

We can combine the TCP declaration and an enumerated type:

```
// Flags that may be set in a TCP segment.
const (
    TCPFlagFin = 1 << iota
    TCPFlagSyn
    TCPFlagRst
    TCPFlagPsh
    TCPFlagAck
    TCPFlagUrg
)

// true if both flags are set
synAck := tcpHeader.Flags & (TCPFlagSyn|TCPFlagAck) == (TCPFlagSyn|TCPFlagAck)
```

Checking for bit flags this way is pretty common in low-level code

Bitwise operators

We can (logical) shift bits inside a byte or word

```
package main
import "fmt"

func main() {
    a, b, c := uint16(1024), uint16(255), uint16(0xff00)

    fmt.Printf("%016b %#04[1]x\n", a)             // 0000010000000000 0x0400
    fmt.Printf("%016b %#04[1]x\n", a << 3)        // 0010000000000000 0x2000
    fmt.Printf("%016b %#04[1]x\n", a << 13)       // 0000000000000000 0x0000

    fmt.Printf("%016b %#04[1]x\n", b)             // 0000000011111111 0x00ff
    fmt.Printf("%016b %#04[1]x\n", b << 2)        // 0000001111111100 0x03fc
    fmt.Printf("%016b %#04[1]x\n", b >> 2)        // 0000000000111111 0x003f
    fmt.Printf("%016b %#04[1]x\n", c >> 2)        // 0011111111000000 0x3fc0
}
```

Sized integers

The 32-bit values get truncated; high bit set $\implies$ negative

```
package main
import "fmt"

func main() {
    var a, b uint32 = 66000, 2000000

    m, n := int16(a), int16(b) // 464, -31616

    fmt.Printf("%032b %016b  %4d\n", a, uint16(m), m)
    fmt.Printf("%032b %016b  %4d\n", b, uint16(n), n)
}
```

```
00000000000000010000000111010000 0000000111010000   464
00000000000111101000010010000000 1000010010000000  -31616
```

Sized integers

Arithmetic with sized integers may overflow

```
package main
import "fmt"

func main() {
    a, b := uint16(8), uint16(128)    // compare to uint8
    x, y := uint16(a*a), uint16(b*b)

    fmt.Printf("%5d %#04[1]x\n", a)    //     8 0x0008   //     8 0x0008
    fmt.Printf("%5d %#04[1]x\n", x)    //    64 0x0040   //    64 0x0040
    fmt.Printf("%5d %#04[1]x\n", b)    //   128 0x0080   //   128 0x0080
    fmt.Printf("%5d %#04[1]x\n", y)    // 16384 0x4000   //     0 0x0000
}
```

The last multiplication doesn't fit (high bits disappear to the left)

Signed integers

There's one more negative number (e.g., -128, ..., -1, 0, 1, ..., 127)

```
package main
import "fmt"

func main() {
    a := int8(-128)    // try -127, -1 for comparison
    b, c := -a, a/-1
    d, e := a+1, a-1
    fmt.Printf("%4d %#02x\n", a, uint8(a)) // -128 0x80 // -127 0x81 // -1 0xff
    fmt.Printf("%4d %#02x\n", b, uint8(b)) // -128 0x80 //  127 0x7f //  1 0x01
    fmt.Printf("%4d %#02x\n", c, uint8(c)) // -128 0x80 //  127 0x7f //  1 0x01
    fmt.Printf("%4d %#02x\n", d, uint8(d)) // -127 0x81 // -126 0x82 //  0 0x00
    fmt.Printf("%4d %#02x\n", e, uint8(e)) //  127 0x7f // -128 0x80 // -2 0xfe
}
```

Weird things happen when we do 8-bit math with -128, be careful

Goto considered harmful *

Every once in a long while, `goto` is simply easier to understand

```
readFormat:
    err = binary.Read(buf, binary.BigEndian, &header.format)

    if err != nil {
        return &header, nil, HeaderReadFailed.from(pos, err)
    }

    if header.format == junkID {
        . . .           // find size & consume WAVE junk header
        goto readFormat
    }

    if header.format != fmtID {
        return &header, nil, InvalidChunkType
    }
```