Effective Go

Introduction

Go là một ngôn ngữ mới. Mặc dù nó vay mượn ý tưởng từ các ngôn ngữ hiện có, nhưng nó có những đặc điểm khác biệt khiến các chương trình Go hiệu quả có tính chất khác so với các chương trình viết bằng các ngôn ngữ họ hàng. Việc dịch thẳng một chương trình C++ hoặc Java sang Go thường sẽ không cho kết quả như ý—chương trình Java nên được viết bằng Java, không phải Go. Ngược lại, nếu suy nghĩ về vấn đề từ góc nhìn của Go, bạn có thể tạo ra một chương trình thành công nhưng lại rất khác biệt. Nói cách khác, để viết Go tốt, điều quan trọng là phải hiểu các đặc tính và phong cách của nó. Cũng quan trọng không kém là phải biết các quy ước đã được thiết lập khi lập trình Go, như đặt tên, định dạng, cấu trúc chương trình, v.v., để các chương trình bạn viết sẽ dễ dàng cho các lập trình viên Go khác hiểu.

Tài liệu này cung cấp các mẹo để viết mã Go rõ ràng, đúng phong cách. Nó bổ sung cho đặc tả ngôn ngữ, Tour of Go, và Cách viết mã Go, tất cả đều nên đọc trước.

Lưu ý bổ sung tháng 1 năm 2022: Tài liệu này được viết cho bản phát hành Go năm 2009, và chưa được cập nhật đáng kể kể từ đó. Mặc dù nó là hướng dẫn tốt để hiểu cách sử dụng ngôn ngữ, nhờ sự ổn định của Go, nhưng nó nói rất ít về các thư viện và không đề cập đến những thay đổi lớn của hệ sinh thái Go kể từ khi được viết, như hệ thống build, kiểm thử, modules, và đa hình. Không có kế hoạch cập nhật nó, vì đã có rất nhiều tài liệu, blog, sách mô tả tốt về cách sử dụng Go hiện đại. Effective Go vẫn hữu ích, nhưng người đọc nên hiểu rằng nó không phải là hướng dẫn đầy đủ. Xem issue 28782 để biết thêm.

Examples

Các mã nguồn package Go được thiết kế không chỉ là thư viện lõi mà còn là ví dụ về cách sử dụng ngôn ngữ. Hơn nữa, nhiều package chứa các ví dụ thực tế, tự chứa, có thể chạy trực tiếp từ trang web go.dev, như ví dụ này (nếu cần, nhấn vào chữ "Example" để mở). Nếu bạn có câu hỏi về cách tiếp cận một vấn đề hoặc cách triển khai một thứ gì đó, tài liệu, mã nguồn và ví dụ trong thư viện có thể cung cấp câu trả lời, ý tưởng và kiến thức nền.

Formatting

Các vấn đề về định dạng là những vấn đề gây tranh cãi nhất nhưng lại ít hệ quả nhất. Mọi người có thể thích nghi với các phong cách định dạng khác nhau nhưng tốt hơn là không phải làm vậy, và sẽ tiết kiệm thời gian hơn nếu mọi người tuân theo cùng một phong cách. Vấn đề là làm sao để đạt được điều này mà không cần một hướng dẫn định dạng dài dòng.

Với Go, chúng tôi áp dụng một cách tiếp cận khác thường và để máy xử lý hầu hết các vấn đề về định dạng. Chương trình `gofmt` (cũng có thể sử dụng qua `go fmt`, hoạt động ở cấp độ package thay vì file nguồn) sẽ đọc một chương trình Go và xuất ra mã nguồn với phong cách chuẩn về thụt lề và căn chỉnh dọc, giữ lại và nếu cần sẽ định dạng lại các chú thích. Nếu bạn muốn biết nên xử lý một tình huống bố cục mới như thế nào, hãy chạy `gofmt`; nếu kết quả không như ý, hãy sắp xếp lại chương trình của bạn (hoặc báo lỗi về `gofmt`), đừng tìm cách lách luật.

Ví dụ, bạn không cần tốn thời gian căn chỉnh chú thích trên các trường của một struct. `Gofmt` sẽ làm điều đó cho bạn. Với khai báo

```
type T struct {
name   string // name of the object
```

```
    value   int // its value
    }
```

gofmt sẽ căn chỉnh các cột, khoảng trắng:

```
    type T struct {
    name string // name of the object
    value int // its value
    }
```

Tất cả mã Go trong các package chuẩn đều đã được định dạng bằng gofmt.

Một số chi tiết về định dạng vẫn còn. Tóm tắt:
Indentation:
Chúng tôi sử dụng tab để thụt lề và gofmt sẽ xuất ra tab theo mặc định. Chỉ dùng dấu cách nếu thật sự cần thiết.
Line length:
Go không giới hạn độ dài dòng. Đừng lo lắng về việc tràn thẻ đục lỗ. Nếu một dòng cảm thấy quá dài, hãy xuống dòng và thụt lề thêm một tab.
Parentheses:
Go cần ít dấu ngoặc đơn hơn C và Java: các cấu trúc điều khiển (if, for, switch) không có dấu ngoặc đơn trong cú pháp. Ngoài ra, thứ tự ưu tiên toán tử ngắn hơn và rõ ràng hơn, nên
`x<<8 + y<<16`
có nghĩa đúng như cách bạn căn khoảng trắng, không như các ngôn ngữ khác.

Commentary

Go cung cấp chú thích kiểu C (`/* */`) và kiểu C++ (`//`). Chú thích dòng là phổ biến; chú thích khối thường xuất hiện như chú thích package, nhưng cũng hữu ích trong biểu thức hoặc để vô hiệu hóa một đoạn mã lớn.

Các chú thích xuất hiện trước các khai báo cấp cao nhất, không có dòng trống xen giữa, được xem là tài liệu cho chính khai báo đó. Những "chú thích tài liệu" này là tài liệu chính cho một package hoặc lệnh Go. Để biết thêm về chú thích tài liệu, xem "Go Doc Comments".

Names

Tên gọi quan trọng trong Go như bất kỳ ngôn ngữ nào khác. Chúng thậm chí còn có ý nghĩa, ngữ nghĩa: khả năng nhìn thấy tên bên ngoài package(public) được xác định bởi ký tự đầu tiên có viết hoa hay không. Vì vậy, đáng để dành chút thời gian nói về quy ước đặt tên trong chương trình Go.

Package names

Khi một package được import, tên package trở thành bộ truy cập cho nội dung của nó như sau

```
import "bytes"
```

package import có thể sử dụng `bytes.Buffer`. Sẽ hữu ích nếu mọi người dùng package đều dùng cùng một tên để tham chiếu nội dung, điều này ngụ ý rằng tên package tốt là: ngắn gọn, súc tích, gợi nhớ. Theo quy ước, các package

được đặt tên bằng chữ thường, tên một từ; không nên có dấu gạch dưới hoặc chữ hoa ghép. Nên nghiêng về phía ngắn gọn, vì mọi người sử dụng package của bạn sẽ phải gõ tên đó. Và đừng lo lắng về việc va chạm *a priori*. Tên package chỉ là tên mặc định cho các import; nó không cần phải duy nhất trên tất cả mã nguồn, và trong trường hợp hiếm hoi xảy ra va chạm, import package có thể chọn một tên khác để sử dụng cục bộ. Trong mọi trường hợp, sự nhầm lẫn là hiếm gặp vì tên tệp trong lệnh import xác định chính xác package nào đang được sử dụng.

Một quy ước khác là tên package là tên cơ sở của thư mục nguồn của nó; package trong `src/encoding/base64` được import là `"encoding/base64"` nhưng có tên là `base64`, không phải `encoding_base64` và không phải `encodingBase64`.

Người nhập khẩu một package sẽ sử dụng tên để tham chiếu đến nội dung của nó, vì vậy các tên xuất khẩu trong package có thể sử dụng thực tế đó để tránh sự lặp lại. (Đừng sử dụng ký hiệu `import .`, điều này có thể đơn giản hóa các bài kiểm tra phải chạy bên ngoài package mà chúng đang kiểm tra, nhưng nên được tránh trong mọi trường hợp khác.) Ví dụ, kiểu bộ đệm trong package `bufio` được gọi là `Reader`, không phải `BufReader`, vì người dùng thấy nó là `bufio.Reader`, đó là một cái tên rõ ràng, súc tích. Hơn nữa, vì các thực thể được nhập khẩu luôn được truy cập bằng tên package của chúng, `bufio.Reader` không bị xung đột với `io.Reader`. Tương tự, hàm để tạo các instances mới của `ring.Ring`—thường được gọi là `NewRing`, nhưng vì `Ring` là kiểu duy nhất được xuất khẩu bởi package, và vì package được gọi là `ring`, nó chỉ được gọi là `New`, mà khách hàng của package thấy là `ring.New`. Sử dụng cấu trúc package để giúp bạn chọn tên tốt.

Một ví dụ ngắn khác là `once.Do`; `once.Do(setup)` đọc tốt và sẽ không được cải thiện bằng cách viết `once.DoOrWaitUntilDone(setup)`. Tên dài không tự động làm cho mọi thứ trở nên dễ đọc hơn. Một chú thích tài liệu hữu ích thường có thể có giá trị hơn một cái tên dài thêm.

Getters

Go không cung cấp hỗ trợ tự động cho các getter và setter. Không có gì sai khi cung cấp các getter và setter của chính bạn, và thường là thích hợp để làm như vậy, nhưng không phải là cách diễn đạt điển hình cũng như không cần thiết phải đưa `Get` vào tên của getter. Nếu bạn có một trường gọi là `owner` (chữ thường, không xuất khẩu), phương thức getter nên được gọi là `Owner` (chữ hoa, xuất khẩu), không phải `GetOwner`. Việc sử dụng tên viết hoa cho xuất khẩu cung cấp móc để phân biệt trường với phương thức. Một hàm setter, nếu cần, có thể được gọi là `SetOwner`. Cả hai tên đều đọc tốt trong thực tế:

```
owner := obj.Owner()
if owner != user {
obj.SetOwner(user)
}
```

Interface names

Theo quy ước, các `one-method interfaces` được đặt tên bằng tên phương thức cộng với hậu tố -er hoặc sửa đổi tương tự để tạo ra một danh từ tác nhân: `Reader`, `Writer`, `Formatter`, `CloseNotifier` v.v.
type Stringer interface {
String() string
}

Có một số tên như vậy và thật hữu ích khi tôn trọng chúng và các tên hàm mà chúng đại diện. `Read`, `Write`, `Close`, `Flush`, `String` và vân vân có chữ ký và ý nghĩa chuẩn. Để tránh nhầm lẫn, đừng đặt tên cho phương thức của bạn

bằng một trong những tên đó trừ khi nó có cùng chữ ký và ý nghĩa. Ngược lại, nếu kiểu của bạn triển khai một phương thức có cùng ý nghĩa với một phương thức trên một kiểu đã biết, hãy đặt tên cho nó với cùng một tên và chữ ký; gọi phương thức chuyển đổi chuỗi của bạn là `String` không phải `ToString`.

MixedCaps

Cuối cùng, quy ước trong Go là sử dụng `MixedCaps` hoặc `mixedCaps` thay vì dấu gạch dưới để viết tên nhiều từ.

Semicolons

Giống như C, ngữ pháp chính thức của Go sử dụng dấu chấm phẩy để kết thúc câu lệnh, nhưng khác với C, các dấu chấm phẩy này không xuất hiện trong mã nguồn. Thay vào đó, lexer sử dụng một quy tắc đơn giản để tự động chèn dấu chấm phẩy khi quét, vì vậy văn bản đầu vào hầu như không có dấu chấm phẩy.

Quy tắc là như sau. Nếu token cuối cùng trước khi xuống dòng là một định danh (bao gồm các từ như `int` và `float64`), một literal cơ bản như số hoặc hằng chuỗi, hoặc một trong các token

```
break continue fallthrough return ++ -- ) }
```

lexer luôn chèn một dấu chấm phẩy sau token đó. Có thể tóm tắt là: "nếu xuống dòng sau một token có thể kết thúc một câu lệnh, hãy chèn dấu chấm phẩy".

Dấu chấm phẩy cũng có thể được bỏ qua ngay trước dấu ngoặc nhọn đóng, vì vậy một câu lệnh như

```
go func() { for { dst <- <-src } }()
```

không cần dấu chấm phẩy. Các chương trình Go chuẩn chỉ có dấu chấm phẩy ở những nơi như các mệnh đề vòng lặp `for`, để phân tách phần khởi đầu, điều kiện và phần tiếp tục. Chúng cũng cần thiết để phân tách nhiều câu lệnh trên một dòng, nếu bạn viết mã theo cách đó.

Một hệ quả của quy tắc chèn dấu chấm phẩy là bạn không thể đặt dấu ngoặc nhọn mở của một cấu trúc điều khiển (`if`, `for`, `switch`, hoặc `select`) ở dòng tiếp theo. Nếu bạn làm vậy, một dấu chấm phẩy sẽ được chèn trước dấu ngoặc nhọn, điều này có thể gây ra các hiệu ứng không mong muốn. Hãy viết như sau

```
if i < f() {
g()
}
```

chứ không phải như sau

```
if i < f() // wrong!
{ // wrong!
g()
}
```

Control structures

Các cấu trúc điều khiển của Go có liên quan đến C nhưng khác biệt ở những điểm quan trọng. Không có vòng lặp `do` hoặc `while`, chỉ có một vòng lặp `for` tổng quát hơn; `switch` linh hoạt hơn; `if` và `switch` chấp nhận một câu lệnh khởi tạo tùy chọn giống như `for`; các câu lệnh `break` và `continue` nhận một nhãn tùy chọn để xác định nơi cần dừng hoặc tiếp tục; và có các cấu trúc điều khiển mới bao gồm type switch và bộ ghép kênh giao tiếp nhiều chiều `select`. Cú pháp cũng hơi khác: không có dấu ngoặc đơn và thân lệnh luôn phải được bao bởi dấu ngoặc nhọn.

If

Trong Go, một câu lệnh `if` đơn giản trông như sau:

```
if x > 0 {
return y
}
```

Dấu ngoặc nhọn bắt buộc khuyến khích việc viết các câu lệnh `if` đơn giản trên nhiều dòng. Dù sao thì đây cũng là một phong cách tốt, đặc biệt khi thân lệnh chứa một câu lệnh điều khiển như `return` hoặc `break`.

Vì `if` và `switch` chấp nhận một câu lệnh khởi tạo, nên thường thấy một câu lệnh khởi tạo được sử dụng để thiết lập một biến cục bộ.

```
if err := file.Chmod(0664); err != nil {
log.Print(err)
return err
}
```

Trong các thư viện Go, bạn sẽ thấy rằng khi một câu lệnh `if` không tiếp tục sang câu lệnh tiếp theo—tức là thân lệnh kết thúc bằng `break`, `continue`, `goto`, hoặc `return`—thì phần `else` không cần thiết sẽ được lược bỏ.

```
f, err := os.Open(name)
if err != nil {
return err
}
codeUsing(f)
```

Đây là một ví dụ về tình huống phổ biến khi mã cần kiểm tra một chuỗi các điều kiện lỗi. Mã sẽ dễ đọc hơn nếu luồng điều khiển thành công chạy xuống dưới, loại bỏ các trường hợp lỗi khi chúng phát sinh. Vì các trường hợp lỗi thường kết thúc bằng câu lệnh `return`, nên mã kết quả không cần các câu lệnh `else`.

```
    f, err := os.Open(name)
    if err != nil {
    return err
    }
    d, err := f.Stat()
```

```
    if err != nil {
    f.Close()
    return err
    }
    codeUsing(f, d)
```

Redeclaration and reassignment

Một lưu ý: Ví dụ cuối cùng ở phần trước minh họa một chi tiết về cách hoạt động của dạng khai báo ngắn `:=`. Khai báo gọi `os.Open` như sau:

```
f, err := os.Open(name)
```

Câu lệnh này khai báo hai biến, `f` và `err`. Vài dòng sau, gọi `f.Stat` như sau:

```
d, err := f.Stat()
```

trông như khai báo `d` và `err`. Tuy nhiên, hãy chú ý rằng `err` xuất hiện ở cả hai câu lệnh. Việc trùng lặp này là hợp lệ: `err` được khai báo ở câu lệnh đầu tiên, nhưng chỉ *gán lại* ở câu lệnh thứ hai. Điều này có nghĩa là lần gọi `f.Stat` sử dụng biến `err` đã được khai báo ở trên, và chỉ gán cho nó một giá trị mới.

Trong một khai báo `:=`, một biến `v` có thể xuất hiện ngay cả khi nó đã được khai báo, với điều kiện:

- khai báo này nằm trong cùng phạm vi với khai báo hiện có của `v` (nếu `v` đã được khai báo ở phạm vi ngoài, khai báo này sẽ tạo một biến mới §),
- giá trị tương ứng trong phần khởi tạo có thể gán cho `v`, và
- có ít nhất một biến khác được tạo bởi khai báo này.

Tính chất đặc biệt này là thực dụng, giúp dễ dàng sử dụng một giá trị `err` duy nhất, ví dụ, trong một chuỗi `if-else` dài. Bạn sẽ thấy nó được sử dụng thường xuyên.

§ Cũng đáng lưu ý rằng trong Go, phạm vi của các tham số hàm và giá trị trả về giống như phạm vi của thân hàm, mặc dù chúng xuất hiện bên ngoài dấu ngoặc nhọn bao quanh thân hàm.

For

Vòng lặp `for` trong Go tương tự như trong C—nhưng không hoàn toàn giống. Nó thống nhất `for` và `while` và không có `do-while`. Có ba dạng, chỉ một trong số đó có dấu chấm phẩy.

```
// Giống như một vòng lặp for trong C
for init; condition; post { }

// Giống như một vòng lặp while trong C
for condition { }

// Giống như một vòng lặp for(;;) trong C
for { }
```

Các khai báo ngắn giúp dễ dàng khai báo biến chỉ số ngay trong vòng lặp.

```
sum := 0
for i := 0; i < 10; i++ {
sum += i
}
```

Nếu bạn đang lặp qua một array, slice, string hoặc map, hoặc đọc từ một chanel, một câu lệnh `range` có thể quản lý vòng lặp.

```
for key, value := range oldMap {
newMap[key] = value
}
```

Nếu bạn chỉ cần mục đầu tiên trong phạm vi (key hoặc index), hãy bỏ qua thứ hai:

```
for key := range m {
if key.expired() {
delete(m, key)
}
}
```

Nếu bạn chỉ cần mục thứ hai trong phạm vi (giá trị), hãy sử dụng *ký tự đại diện bỏ qua*, dấu gạch dưới, để loại bỏ mục đầu tiên:

```
sum := 0
for _, value := range array {
sum += value
}
```

Ký tự đại diện bỏ qua có nhiều cách sử dụng, như sẽ được mô tả trong một phần sau.

Đối với các chuỗi, `range` thực hiện nhiều công việc hơn cho bạn, phân tách các điểm mã Unicode riêng lẻ bằng cách phân tích UTF-8. Các mã hóa sai sẽ tiêu tốn một byte và tạo ra rune thay thế U+FFFD. (Tên (với kiểu tích hợp liên quan) `rune` là thuật ngữ của Go cho một điểm mã Unicode đơn lẻ. Xem cụ thể ngôn ngữ để biết chi tiết.) Vòng lặp:

```
    for pos, char := range "日本\\x80 語" { // \x80 là một mã hóa UTF-8 không hợp lệ
    fmt.Printf("character %#U starts at byte position %d\n", char, pos)
    }
```

in ra

```
character U+65E5 '日' starts at byte position 0
character U+672C '本' starts at byte position 3
character U+FFFD '' starts at byte position 6
character U+8A9E '語' starts at byte position 7
```

Cuối cùng, Go không có toán tử phẩy và `++` và `--` là các câu lệnh chứ không phải biểu thức. Do đó, nếu bạn muốn chạy nhiều biến trong một vòng lặp `for`, bạn nên sử dụng gán song song (mặc dù điều đó sẽ loại trừ `++` và `--`).

```
// Đảo ngược a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```

Switch

Go's `switch` tổng quát hơn so với C. Biểu thức không nhất thiết phải là hằng số hoặc số nguyên, các trường hợp được đánh giá từ trên xuống dưới cho đến khi tìm thấy trường hợp phù hợp, và nếu `switch` không có biểu thức thì nó sẽ kiểm tra trên giá trị `true`. Do đó, hoàn toàn có thể—và là phong cách Go—viết một chuỗi `if-else-if-else` dưới dạng một `switch`.

```
func unhex(c byte) byte {
switch {
case '0' <= c && c <= '9':
return c - '0'
case 'a' <= c && c <= 'f':
return c - 'a' + 10
case 'A' <= c && c <= 'F':
return c - 'A' + 10
}
return 0
}
```

Không có tự động rơi xuống trường hợp tiếp theo (fall through), nhưng các trường hợp có thể được liệt kê trong một danh sách phân tách bằng dấu phẩy.

```
func shouldEscape(c byte) bool {
switch c {
case ' ', '?', '&', '=', '#', '+', '%':
return true
}
return false
}
```

Mặc dù chúng không phổ biến trong Go như ở một số ngôn ngữ giống C khác, câu lệnh `break` có thể được sử dụng để kết thúc một `switch` sớm. Tuy nhiên, đôi khi dùng để thoát khỏi một vòng lặp bao quanh, không phải chỉ `switch`, và trong Go điều này có thể thực hiện bằng cách đặt một nhãn cho vòng lặp và "breaking" tới nhãn đó. Ví dụ này thể hiện cả hai cách sử dụng.

```
Loop:
    for n := 0; n < len(src); n += size {
        switch {
        case src[n] < sizeOne:
            if validateOnly {
                break
            }
            size = 1
            update(src[n])

        case src[n] < sizeTwo:
            if n+1 >= len(src) {
                err = errShortInput
                break Loop
            }
            if validateOnly {
                break
            }
            size = 2
            update(src[n] + src[n+1]<<shift)
        }
    }
```

Tất nhiên, câu lệnh `continue` cũng chấp nhận một nhãn tùy chọn nhưng chỉ áp dụng cho vòng lặp.

Để kết thúc phần này, dưới đây là một hàm so sánh hai slice byte sử dụng hai câu lệnh `switch`:

```
// Compare returns an integer comparing the two byte slices,
// lexicographically.
// The result will be 0 if a == b, -1 if a < b, and +1 if a > b
func Compare(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }
    switch {
    case len(a) > len(b):
        return 1
    case len(a) < len(b):
        return -1
    }
```

```
    return 0
}
```

Type switch

Một `switch` cũng có thể được sử dụng để phát hiện kiểu động của một biến interface. Một *type switch* sử dụng cú pháp của type assertion với từ khóa `type` bên trong dấu ngoặc đơn. Nếu `switch` khai báo một biến trong biểu thức, biến đó sẽ có kiểu tương ứng trong từng nhánh. Ngoài ra, phong cách Go là tái sử dụng tên biến trong các trường hợp này, thực chất là khai báo một biến mới cùng tên nhưng kiểu khác nhau ở mỗi nhánh.

```
    var t interface{}
    t = functionOfSomeType()
    switch t := t.(type) {
    default:
    fmt.Printf("unexpected type %T\n", t) // %T prints whatever type t has
    case bool:
    fmt.Printf("boolean %t\n", t) // t has type bool
    case int:
    fmt.Printf("integer %d\n", t) // t has type int
    case *bool:
    fmt.Printf("pointer to boolean %t\n", *t) // t has type *bool
    case *int:
    fmt.Printf("pointer to integer %d\n", *t) // t has type *int
    }
```

Functions

Multiple return values

Một trong những tính năng đặc biệt của Go là hàm và phương thức có thể trả về nhiều giá trị. Dạng này có thể cải thiện một số thói quen rườm rà trong C: trả về lỗi kiểu "in-band" như `-1` cho `EOF` và sửa đổi một tham số truyền vào qua địa chỉ(con trỏ).

Trong C, lỗi ghi được báo hiệu bằng một số âm với mã lỗi được lưu ở một vị trí biến động. Trong Go, `Write` có thể trả về số byte đã ghi *and* một lỗi: "Bạn đã ghi được một số byte nhưng không phải tất cả vì thiết bị đã đầy". Chữ ký của phương thức `Write` trên file trong package `os` là:

```
func (file *File) Write(b []byte) (n int, err error)
```

Và như tài liệu nói, nó trả về số byte đã ghi và một giá trị `error` khác nil khi `n != len(b)`. Đây là phong cách phổ biến; xem phần xử lý lỗi để biết thêm ví dụ.

Một cách tiếp cận tương tự loại bỏ nhu cầu truyền con trỏ đến giá trị trả về để mô phỏng tham số tham chiếu. Đây là một hàm đơn giản lấy một số từ một vị trí trong slice byte, trả về số đó và vị trí tiếp theo.

```
func nextInt(b []byte, i int) (int, int) {
    for ; i < len(b) && !isDigit(b[i]); i++ {
```

```
    }
    x := 0
    for ; i < len(b) && isDigit(b[i]); i++ {
        x = x*10 + int(b[i]) - '0'
    }
    return x, i
}
```

Bạn có thể dùng nó để quét các số trong một slice đầu vào `b` như sau:

```
for i := 0; i < len(b); {
        x, i = nextInt(b, i)
        fmt.Println(x)
    }
```

Named result parameters

Các tham số trả về hoặc "kết quả" của một hàm Go có thể được đặt tên và sử dụng như biến thông thường, giống như các tham số đầu vào. Khi được đặt tên, chúng sẽ được khởi tạo về giá trị zero của kiểu khi hàm bắt đầu; nếu hàm thực thi một câu lệnh `return` không có đối số, các giá trị hiện tại của các tham số kết quả sẽ được sử dụng làm giá trị trả về.
Tên không bắt buộc nhưng có thể làm mã ngắn gọn và rõ ràng hơn: chúng cũng là tài liệu. Nếu chúng ta đặt tên cho kết quả của `nextInt` thì sẽ rõ ràng giá trị `int` nào được trả về.

```
func nextInt(b []byte, pos int) (value, nextPos int) {
```

Vì kết quả được khởi tạo và gắn với return không đối số, chúng có thể đơn giản hóa cũng như làm rõ mã. Đây là một phiên bản của `io.ReadFull` sử dụng chúng hiệu quả.

```
func ReadFull(r Reader, buf []byte) (n int, err error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:]
    }
    return
}
```

Defer

Câu lệnh `defer` của Go lên lịch một lời gọi hàm (hàm *deferred*) để chạy ngay trước khi hàm thực thi `defer` trả về. Đây là một cách hiệu quả để xử lý các tình huống như tài nguyên cần được giải phóng bất kể hàm trả về theo đường nào. Ví dụ điển hình là mở khóa mutex hoặc đóng file.

```
// Contents returns the file's contents as a string.
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close()  // f.Close will run when we're finished.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append is discussed later.
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err  // f will be closed if we return here.
        }
    }
    return string(result), nil // f will be closed if we return here.
}
```

Việc defer gọi một hàm như `Close` có hai lợi ích. Thứ nhất, nó đảm bảo bạn sẽ không bao giờ quên đóng file, một lỗi dễ mắc phải nếu sau này bạn chỉnh sửa hàm để thêm một đường return mới. Thứ hai, nó giúp lệnh đóng nằm gần lệnh mở, rõ ràng hơn nhiều so với việc đặt nó ở cuối hàm.

Các đối số truyền vào hàm deferred (bao gồm cả receiver nếu là phương thức) được đánh giá tại thời điểm thực thi *defer*, không phải khi hàm được gọi *call*. Điều này giúp tránh lo lắng về việc biến thay đổi giá trị khi hàm thực thi, đồng thời cho phép một vị trí defer duy nhất defer nhiều lần gọi hàm. Đây là một ví dụ đơn giản.

```
for i := 0; i < 5; i++ {
defer fmt.Printf("%d ", i)
}
```

Các hàm deferred được thực thi theo thứ tự LIFO, nên đoạn mã này sẽ in ra `4 3 2 1 0` khi hàm trả về. Một ví dụ thực tế hơn là cách đơn giản để trace quá trình thực thi hàm trong chương trình. Ta có thể viết một cặp hàm trace đơn giản như sau:

```
func trace(s string)   { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }

// Use them like this:
func a() {
    trace("a")
    defer untrace("a")
    // do something....
}
```

Chúng ta có thể làm tốt hơn bằng cách tận dụng việc các đối số truyền vào hàm deferred được đánh giá khi thực thi `defer`. Hàm trace có thể thiết lập đối số cho hàm untrace. Ví dụ này:

```
func trace(s string) string {
    fmt.Println("entering:", s)
    return s
}

func un(s string) {
    fmt.Println("leaving:", s)
}

func a() {
    defer un(trace("a"))
    fmt.Println("in a")
}

func b() {
    defer un(trace("b"))
    fmt.Println("in b")
    a()
}

func main() {
    b()
}
```

Đối với lập trình viên quen với quản lý tài nguyên theo block ở các ngôn ngữ khác, `defer` có thể có vẻ lạ, nhưng ứng dụng thú vị và mạnh mẽ nhất của nó đến từ việc nó dựa trên hàm chứ không phải block. Trong phần về `panic` và `recover` chúng ta sẽ thấy thêm một ví dụ về khả năng này.

Data

Allocation with `new`

Go có hai primitive cấp phát bộ nhớ, hàm tích hợp `new` và `make`. Chúng làm những việc khác nhau và áp dụng cho các kiểu khác nhau, điều này có thể gây nhầm lẫn, nhưng quy tắc rất đơn giản. Đầu tiên hãy nói về `new`. Đây là một hàm tích hợp cấp phát bộ nhớ, nhưng không *khởi tạo* bộ nhớ, chỉ gán *zero*. Nghĩa là, `new(T)` cấp phát vùng nhớ zero cho một phần tử kiểu `T` mới và trả về địa chỉ của nó, một giá trị kiểu `*T`. Theo thuật ngữ Go, nó trả về một con trỏ tới giá trị zero mới được cấp phát kiểu `T`.

Vì bộ nhớ trả về bởi `new` là zero, nên khi thiết kế cấu trúc dữ liệu, bạn nên sắp xếp sao cho giá trị zero của mỗi kiểu có thể sử dụng được mà không cần khởi tạo thêm. Điều này giúp người dùng cấu trúc dữ liệu có thể tạo một instance bằng `new` và sử dụng ngay. Ví dụ, tài liệu của `bytes.Buffer` ghi rằng "giá trị zero của `Buffer` là một buffer rỗng sẵn sàng sử dụng." Tương tự, `sync.Mutex` không có constructor hoặc phương thức `Init` rõ ràng. Thay vào đó, giá trị zero của `sync.Mutex` được định nghĩa là một mutex chưa bị khóa.

Tính chất "giá trị zero có thể dùng được" này có tính chất lan truyền. Xem khai báo kiểu sau:

```
type SyncedBuffer struct {
lock sync.Mutex
buffer bytes.Buffer
}
```

Các giá trị kiểu `SyncedBuffer` cũng sẵn sàng sử dụng ngay khi được cấp phát hoặc chỉ khai báo. Trong đoạn mã tiếp theo, cả `p` và `v` đều hoạt động đúng mà không cần sắp xếp thêm.

```
p := new(SyncedBuffer) // type \*SyncedBuffer
var v SyncedBuffer // type SyncedBuffer
```

Constructors and composite literals

Đôi khi giá trị zero là không đủ và cần một constructor khởi tạo, như ví dụ này lấy từ package `os`.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
    f.nepipe = 0
    return f
}
```

Có khá nhiều mã lặp lại ở đây. Ta có thể đơn giản hóa bằng *composite literal*, là một biểu thức tạo một instance mới mỗi lần được đánh giá.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
    return nil
    }
    f := File{fd, name, nil, 0}
    return &f
}
```

Lưu ý rằng, khác với C, việc trả về địa chỉ của biến cục bộ là hoàn toàn hợp lệ; vùng nhớ liên quan đến biến sẽ tồn tại sau khi hàm trả về. Thực tế, việc lấy địa chỉ của một composite literal sẽ cấp phát một instance mới mỗi lần nó được đánh giá, nên ta có thể kết hợp hai dòng cuối lại.

```
return &File{fd, name, nil, 0}
```

Các trường của một composite literal được sắp xếp theo thứ tự và phải có mặt đầy đủ. Tuy nhiên, bằng cách gán nhãn các phần tử rõ ràng dưới dạng *trường:giá trị*, các giá trị khởi tạo có thể xuất hiện theo bất kỳ thứ tự nào, với các trường thiếu được để lại như giá trị zero tương ứng. Do đó ta có thể viết

```
return &File{fd: fd, name: name}
```

Như một trường hợp giới hạn, nếu một composite literal không có trường nào, nó tạo ra một giá trị zero cho kiểu. Các biểu thức `new(File)` và `&File{}` là tương đương.

Composite literal cũng có thể được tạo cho mảng, slice, và map, với các nhãn trường là chỉ số hoặc khóa map tương ứng. Trong các ví dụ này, việc khởi tạo hoạt động bất kể giá trị của `Enone`, `Eio`, và `Einval`, miễn là chúng khác nhau.

```
a := [...]string   {Enone: "no error", Eio: "Eio", Einval: "invalid argument"}
s := []string      {Enone: "no error", Eio: "Eio", Einval: "invalid argument"}
m := map[int]string{Enone: "no error", Eio: "Eio", Einval: "invalid argument"}
```

Allocation with `make`

Quay lại việc cấp phát. Hàm tích hợp `make(T, args)` phục vụ mục đích khác với `new(T)`. Nó chỉ tạo slice, map và channel, và trả về giá trị *đã khởi tạo* (không phải *zero*) kiểu `T` (không phải `*T`). Lý do là vì ba kiểu này thực chất là tham chiếu đến các cấu trúc dữ liệu cần được khởi tạo trước khi sử dụng. Ví dụ, slice là một descriptor gồm ba phần tử: con trỏ đến dữ liệu (trong một mảng), length và capacity, và cho đến khi các phần tử này được khởi tạo, slice là `nil`. Đối với slice, map và channel, `make` khởi tạo cấu trúc dữ liệu bên trong và chuẩn bị giá trị để sử dụng. Ví dụ,

```
make([]int, 10, 100)
```

cấp phát một mảng 100 phần tử int và tạo một slice có độ dài 10 và dung lượng 100 trỏ đến 10 phần tử đầu tiên của mảng. (Khi tạo slice, có thể bỏ qua dung lượng; xem phần về slice để biết thêm thông tin.) Ngược lại, `new([]int)` trả về một con trỏ đến một slice mới được cấp phát, giá trị zero, tức là con trỏ đến một slice nil.

Các ví dụ sau minh họa sự khác biệt giữa `new` và `make`.

```
var p *[]int = new([]int)       // allocates slice structure; *p == nil; rarely useful
var v  []int = make([]int, 100) // the slice v now refers to a new array of 100 ints

// Unnecessarily complex:
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// Idiomatic:
v := make([]int, 100)
```

Hãy nhớ rằng `make` chỉ áp dụng cho map, slice và channel và không trả về con trỏ. Để lấy con trỏ rõ ràng, hãy dùng `new` hoặc lấy địa chỉ biến một cách tường minh.

Arrays

Mảng hữu ích khi cần lên kế hoạch chi tiết bố trí bộ nhớ và đôi khi giúp tránh cấp phát, nhưng chủ yếu chúng là nền tảng cho slice, chủ đề của phần tiếp theo. Để chuẩn bị cho chủ đề đó, dưới đây là một số điểm về mảng.

Có những khác biệt lớn giữa cách mảng hoạt động trong Go và C. Trong Go:

- Mảng là giá trị. Gán một mảng cho mảng khác sẽ sao chép tất cả phần tử.
- Đặc biệt, nếu bạn truyền một mảng vào hàm, nó sẽ nhận *bản sao* của mảng, không phải con trỏ đến nó.
- Kích thước mảng là một phần của kiểu. Kiểu `[10]int` và `[20]int` là khác nhau.

Tính chất giá trị có thể hữu ích nhưng cũng tốn kém; nếu bạn muốn hành vi và hiệu suất giống C, bạn có thể truyền con trỏ đến mảng.

```
func Sum(a *[3]float64) (sum float64) {
    for _, v := range *a {
        sum += v
    }
    return
}

array := [...]float64{7.0, 8.5, 9.1}
x := Sum(&array)  // Note the explicit address-of operator
```

Nhưng ngay cả phong cách này cũng không phải là Go idiomatic(cách hay dùng trong Go). Hãy dùng slice thay thế.

Slices

Slices bao bọc mảng để cung cấp giao diện tổng quát, mạnh mẽ và tiện lợi hơn cho chuỗi dữ liệu. Ngoại trừ các trường hợp có kích thước rõ ràng như ma trận biến đổi, hầu hết lập trình mảng trong Go được thực hiện với slices thay vì mảng đơn giản.

Slices giữ tham chiếu đến mảng bên dưới, và nếu bạn gán một slice cho slice khác, cả hai sẽ tham chiếu cùng một mảng. Nếu một hàm nhận đối số là slice, các thay đổi nó thực hiện lên phần tử của slice sẽ hiển thị cho caller, tương tự như truyền con trỏ đến mảng bên dưới. Do đó, một hàm `Read` có thể nhận đối số là slice thay vì con trỏ và số lượng; độ dài của slice đặt giới hạn tối đa dữ liệu đọc. Đây là chữ ký của phương thức `Read` của kiểu `File` trong package `os`:

```
func (f *File) Read(buf []byte) (n int, err error)
```

Phương thức trả về số byte đã đọc và giá trị lỗi, nếu có. Để đọc vào 32 byte đầu tiên của buffer lớn hơn `buf`, *slice* (dùng như động từ) buffer.

```
n, err := f.Read(buf[0:32])
```

Việc slice như vậy là phổ biến và hiệu quả. Thực tế, nếu bỏ qua hiệu suất, đoạn mã sau cũng sẽ đọc 32 byte đầu tiên của buffer.

```
var n int
    var err error
    for i := 0; i < 32; i++ {
        nbytes, e := f.Read(buf[i:i+1])  // Read one byte.
        n += nbytes
        if nbytes == 0 || e != nil {
            err = e
            break
        }
    }
```

Độ dài của slice có thể thay đổi miễn là nó vẫn nằm trong giới hạn của mảng bên dưới; chỉ cần gán nó cho một slice của chính nó. *Dung lượng* của slice, truy cập qua hàm tích hợp `cap`, cho biết độ dài tối đa slice có thể nhận. Đây là một hàm để nối dữ liệu vào slice. Nếu dữ liệu vượt quá dung lượng, slice sẽ được cấp phát lại. Slice kết quả được trả về. Hàm sử dụng thực tế là `len` và `cap` hợp lệ khi áp dụng cho slice nil, và trả về 0.

```
func Append(slice, data []byte) []byte {
    l := len(slice)
    if l + len(data) > cap(slice) {  // reallocate
        // Allocate double what's needed, for future growth.
        newSlice := make([]byte, (l+len(data))*2)
        // The copy function is predeclared and works for any slice type.
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:l+len(data)]
    copy(slice[l:], data)
    return slice
}
```

Chúng ta phải trả về slice sau đó vì, mặc dù `Append` có thể thay đổi phần tử của `slice`, bản thân slice (cấu trúc dữ liệu runtime giữ con trỏ, độ dài và dung lượng) được truyền theo giá trị.

Ý tưởng nối vào slice hữu ích đến mức nó được tích hợp thành hàm `append`. Để hiểu thiết kế của hàm này, ta cần thêm một chút thông tin, sẽ đề cập sau.

Two-dimensional slices

Mảng và slice của Go là một chiều. Để tạo tương đương với mảng hoặc slice hai chiều, cần định nghĩa mảng của mảng hoặc slice của slice, như sau:

```
type Transform [3][3]float64  // A 3x3 array, really an array of arrays.
type LinesOfText [][]byte     // A slice of byte slices.
```

Vì slice có độ dài thay đổi, mỗi slice bên trong có thể có độ dài khác nhau. Đây là tình huống phổ biến, như ví dụ `LinesOfText`: mỗi dòng có độ dài riêng.

```
    text := LinesOfText{
    []byte("Now is the time"),
    []byte("for all good gophers"),
    []byte("to bring some fun to the party."),
}
```

Đôi khi cần cấp phát một slice hai chiều, ví dụ khi xử lý các dòng ảnh pixel. Có hai cách để làm điều này. Một là cấp phát từng slice riêng biệt; cách còn lại là cấp phát một mảng lớn và trỏ từng slice vào đó. Cách nào dùng tùy thuộc vào ứng dụng. Nếu slice có thể tăng giảm, nên cấp phát riêng để tránh ghi đè dòng tiếp theo; nếu không, có thể hiệu quả hơn khi xây dựng đối tượng bằng một lần cấp phát. Dưới đây là phác thảo hai phương pháp. Đầu tiên, từng dòng một:

```
// Allocate the top-level slice.
picture := make([][]uint8, YSize) // One row per unit of y.
// Loop over the rows, allocating the slice for each row.
for i := range picture {
    picture[i] = make([]uint8, XSize)
}
```

Và bây giờ là một lần cấp phát, chia thành các dòng:

```
// Allocate the top-level slice, the same as before.
picture := make([][]uint8, YSize) // One row per unit of y.
// Allocate one large slice to hold all the pixels.
pixels := make([]uint8, XSize*YSize) // Has type []uint8 even though picture is []
[]uint8.
// Loop over the rows, slicing each row from the front of the remaining pixels slice.
for i := range picture {
    picture[i], pixels = pixels[:XSize], pixels[XSize:]
}
```

Maps

Maps là một cấu trúc dữ liệu tích hợp tiện lợi và mạnh mẽ, liên kết giá trị của một kiểu (khóa) với giá trị của kiểu khác (giá trị). Khóa có thể là bất kỳ kiểu nào mà toán tử so sánh bằng được định nghĩa, như số nguyên, số thực, số phức, chuỗi, con trỏ, interface (miễn là kiểu động hỗ trợ so sánh bằng), struct và mảng. Slice không thể dùng làm khóa map vì không có phép so sánh bằng. Giống như slice, maps giữ tham chiếu đến cấu trúc dữ liệu bên dưới. Nếu bạn truyền một map vào hàm thay đổi nội dung map, thay đổi sẽ hiển thị ở caller.

Maps có thể được tạo bằng cú pháp composite literal thông thường với cặp khóa-giá trị phân tách bằng dấu hai chấm, nên rất dễ xây dựng khi khởi tạo.

```
var timeZone = map[string]int{
    "UTC":  0*60*60,
    "EST": -5*60*60,
    "CST": -6*60*60,
    "MST": -7*60*60,
```

```
    "PST": -8*60*60,
}
```

Gán và truy xuất giá trị map có cú pháp giống như với mảng và slice, chỉ khác là chỉ số không cần là số nguyên.

```
offset := timeZone["EST"]
```

Nếu truy xuất giá trị map với khóa không tồn tại, sẽ trả về giá trị zero của kiểu phần tử trong map. Ví dụ, nếu map chứa số nguyên, tra cứu khóa không tồn tại sẽ trả về `0`. Một tập hợp (set) có thể được cài đặt bằng map với kiểu giá trị là `bool`. Đặt phần tử map thành `true` để thêm giá trị vào tập hợp, sau đó kiểm tra bằng chỉ số đơn giản.

```
    attended := map[string]bool{
    "Ann": true,
    "Joe": true,
    ...
}

if attended[person] { // will be false if person is not in the map
    fmt.Println(person, "was at the meeting")
}
```

Đôi khi bạn cần phân biệt giữa phần tử không tồn tại và giá trị zero. Có phần tử cho `"UTC"` hay đó là 0 vì không có trong map? Bạn có thể phân biệt bằng dạng gán nhiều giá trị.

```
var seconds int
var ok bool
seconds, ok = timeZone[tz]
```

Vì lý do rõ ràng, đây được gọi là idiom "comma ok". Trong ví dụ này, nếu `tz` tồn tại, `seconds` sẽ được gán đúng và `ok` là true; nếu không, `seconds` sẽ được gán giá trị 0 và `ok` sẽ là false. Dưới đây là một hàm kết hợp với báo lỗi đẹp:

```
   func offset(tz string) int {
    if seconds, ok := timeZone[tz]; ok {
        return seconds
    }
    log.Println("unknown time zone:", tz)
    return 0
}
```

Để kiểm tra sự tồn tại trong map mà không quan tâm giá trị thực tế, bạn có thể dùng blank identifier (`_`) thay cho biến thông thường cho giá trị.

```
_, present := timeZone[tz]
```

Để xóa một phần tử map, dùng hàm tích hợp `delete`, nhận map và khóa cần xóa. Làm vậy an toàn ngay cả khi khóa đã không còn trong map.

```
delete(timeZone, "PDT") // Now on Standard Time
```

Printing

In định dạng trong Go sử dụng phong cách giống họ `printf` của C nhưng phong phú và tổng quát hơn. Các hàm này nằm trong package `fmt` và có tên viết hoa: `fmt.Printf`, `fmt.Fprintf`, `fmt.Sprintf` và tương tự. Các hàm chuỗi (`Sprintf` v.v.) trả về một chuỗi thay vì điền vào một buffer được cung cấp.

Bạn không cần phải cung cấp chuỗi định dạng. Với mỗi hàm `Printf`, `Fprintf` và `Sprintf` đều có một cặp hàm khác, ví dụ như `Print` và `Println`. Các hàm này không nhận chuỗi định dạng mà tự động tạo định dạng mặc định cho từng đối số. Phiên bản `Println` cũng chèn khoảng trắng giữa các đối số và thêm ký tự xuống dòng vào cuối, trong khi phiên bản `Print` chỉ thêm khoảng trắng nếu toán hạng ở hai bên không phải là chuỗi. Trong ví dụ này, mỗi dòng sẽ cho ra kết quả giống nhau.

```
fmt.Printf("Hello %d\n", 23)
fmt.Fprint(os.Stdout, "Hello ", 23, "\n")
fmt.Println("Hello", 23)
fmt.Println(fmt.Sprint("Hello ", 23))
```

Các hàm in định dạng như `fmt.Fprint` và các hàm liên quan nhận đối số đầu tiên là bất kỳ đối tượng nào cài đặt interface `io.Writer`; các biến `os.Stdout` và `os.Stderr` là những ví dụ quen thuộc.

Tại đây mọi thứ bắt đầu khác với C. Đầu tiên, các định dạng số như `%d` không nhận cờ cho dấu hoặc kích thước; thay vào đó, các hàm in sử dụng kiểu của đối số để quyết định các thuộc tính này.

```
var x uint64 = 1<<64 - 1
fmt.Printf("%d %x; %d %x\n", x, x, int64(x), int64(x))
```

prints

```
18446744073709551615 ffffffffffffffff; -1 -1
```

Nếu bạn chỉ muốn chuyển đổi mặc định, như số thập phân cho số nguyên, bạn có thể sử dụng định dạng tổng quát `%v` (viết tắt của "value"); kết quả sẽ giống hệt như khi dùng `Print` và `Println`. Hơn nữa, định dạng này có thể in *bất kỳ* giá trị nào, kể cả mảng, slice, struct và map. Dưới đây là một câu lệnh in cho map múi giờ đã định nghĩa ở phần trước.

```
fmt.Printf("%v\n", timeZone)  // or just fmt.Println(timeZone)
```

which gives output:

```
map[CST:-21600 EST:-18000 MST:-25200 PST:-28800 UTC:0]
```

Đối với map, `Printf` và các hàm liên quan sẽ sắp xếp kết quả theo thứ tự từ điển của khóa.

Khi in struct, định dạng `%+v` sẽ chú thích các trường với tên của chúng, và với bất kỳ giá trị nào, định dạng thay thế `%#v` sẽ in giá trị theo cú pháp đầy đủ của Go.

```
type T struct {
    a int
    b float64
    c string
}
t := &T{ 7, -2.35, "abc\tdef" }
fmt.Printf("%v\n", t)
fmt.Printf("%+v\n", t)
fmt.Printf("%#v\n", t)
fmt.Printf("%#v\n", timeZone)
```

prints

```
&{7 -2.35 abc   def}
&{a:7 b:-2.35 c:abc     def}
&main.T{a:7, b:-2.35, c:"abc\tdef"}
map[string]int{"CST":-21600, "EST":-18000, "MST":-25200, "PST":-28800, "UTC":0}
```

(Lưu ý dấu &.) Định dạng chuỗi có dấu nháy này cũng có sẵn qua `%q` khi áp dụng cho giá trị kiểu `string` hoặc `[]byte`. Định dạng `%#q` sẽ dùng dấu nháy ngược nếu có thể. (Định dạng `%q` cũng áp dụng cho số nguyên và rune, tạo ra hằng rune có dấu nháy đơn.) Ngoài ra, `%x` hoạt động trên chuỗi, mảng byte và slice byte cũng như số nguyên, tạo ra chuỗi hex dài, và với dấu cách trong định dạng (`% x`) sẽ chèn dấu cách giữa các byte.

Một định dạng tiện lợi khác là `%T`, in ra *kiểu* của giá trị.

```
fmt.Printf("%T\n", timeZone)
```

prints

```
map[string]int
```

Nếu bạn muốn kiểm soát định dạng mặc định cho kiểu tự định nghĩa, chỉ cần định nghĩa một phương thức với chữ ký `String() string` trên kiểu đó. Với kiểu đơn giản `T`, có thể như sau.

```
func (t *T) String() string {
    return fmt.Sprintf("%d/%g/%q", t.a, t.b, t.c)
}
fmt.Printf("%v\n", t)
```

to print in the format

```
7/-2.35/"abc\tdef"
```

(Nếu bạn cần in *giá trị* của kiểu `T` cũng như con trỏ tới `T`, receiver cho phương thức `String` phải là kiểu giá trị; ví dụ này sử dụng con trỏ vì nó hiệu quả và đúng idiomatic hơn cho struct. Xem phần bên dưới về con trỏ so với receiver giá trị để biết thêm chi tiết.)

Phương thức `String` của chúng ta có thể gọi `Sprintf` vì các hàm in hoàn toàn reentrant và có thể được bọc như vậy. Tuy nhiên, có một chi tiết quan trọng cần hiểu về cách tiếp cận này: đừng xây dựng phương thức `String` bằng cách gọi `Sprintf` theo cách sẽ đệ quy vào chính phương thức `String` của bạn vô hạn. Điều này có thể xảy ra nếu lệnh gọi `Sprintf` cố in trực tiếp receiver dưới dạng chuỗi, điều này sẽ gọi lại phương thức. Đây là lỗi phổ biến và dễ mắc phải, như ví dụ sau.

```
type MyString string

func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", m) // Error: will recur forever.
}
```

Cũng dễ sửa: chuyển đổi đối số về kiểu chuỗi cơ bản, kiểu này không có phương thức.

```
type MyString string
func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", string(m)) // OK: note conversion.
}
```

Trong phần khởi tạo chúng ta sẽ thấy một kỹ thuật khác tránh được đệ quy này.

Một kỹ thuật in khác là truyền trực tiếp các đối số của hàm in cho một hàm in khác. Chữ ký của `Printf` dùng kiểu `...interface{}` cho đối số cuối để chỉ định rằng có thể truyền vào số lượng tham số tùy ý (và kiểu tùy ý) sau định dạng.

```
func Printf(format string, v ...interface{}) (n int, err error) {
```

Bên trong hàm `Printf`, `v` hoạt động như biến kiểu `[]interface{}` nhưng nếu truyền cho một hàm variadic khác, nó hoạt động như danh sách đối số thông thường. Đây là cài đặt của hàm `log.Println` đã dùng ở trên. Nó truyền các đối số trực tiếp cho `fmt.Sprintln` để định dạng thực tế.

```
// Println prints to the standard logger in the manner of fmt.Println.
func Println(v ...interface{}) {
    std.Output(2, fmt.Sprintln(v...))  // Output takes parameters (int, string)
}
```

Ta viết `...` sau `v` trong lệnh gọi lồng nhau tới `Sprintln` để báo cho trình biên dịch coi `v` là danh sách đối số; nếu không nó sẽ chỉ truyền `v` như một đối số slice duy nhất.

Còn nhiều điều về in hơn những gì đã đề cập ở đây. Xem tài liệu `godoc` của package `fmt` để biết chi tiết.

Nhân tiện, một tham số `...` có thể là kiểu cụ thể, ví dụ `...int` cho hàm min chọn số nhỏ nhất trong danh sách số nguyên:

```
func Min(a ...int) int {
    min := int(^uint(0) >> 1)  // largest int
    for _, i := range a {
        if i < min {
            min = i
        }
    }
    return min
}
```

Append

Bây giờ chúng ta có mảnh ghép còn thiếu để giải thích thiết kế của hàm tích hợp `append`. Chữ ký của `append` khác với hàm `Append` tự viết ở trên. Về mặt sơ đồ, nó như sau:

```
func append(slice []T, elements ...T) []T
```

Trong đó *T* là placeholder cho bất kỳ kiểu nào. Bạn không thể thực sự viết một hàm trong Go mà kiểu `T` được xác định bởi caller. Đó là lý do `append` là hàm tích hợp: nó cần sự hỗ trợ từ trình biên dịch.

Những gì `append` làm là nối các phần tử vào cuối slice và trả về kết quả. Kết quả cần được trả về vì, giống như hàm `Append` tự viết, mảng bên dưới có thể thay đổi. Ví dụ đơn giản này

```
x := []int{1,2,3}
x = append(x, 4, 5, 6)
fmt.Println(x)
```

in ra `[1 2 3 4 5 6]`. Vậy `append` hoạt động giống như `Printf`, thu thập số lượng đối số tùy ý.

Nhưng nếu muốn làm như `Append` và nối một slice vào slice khác thì sao? Đơn giản: dùng `...` tại vị trí gọi, giống như đã làm với lệnh gọi `Output` ở trên. Đoạn mã này cho kết quả giống hệt đoạn trên.

```
x := []int{1,2,3}
y := []int{4,5,6}
x = append(x, y...)
fmt.Println(x)
```

Nếu không có `...`, mã sẽ không biên dịch vì kiểu không đúng; `y` không phải là kiểu `int`.

Initialization

Mặc dù nhìn bề ngoài không khác nhiều so với khởi tạo trong C hoặc C++, khởi tạo trong Go mạnh mẽ hơn. Có thể xây dựng các cấu trúc phức tạp trong quá trình khởi tạo và các vấn đề về thứ tự giữa các đối tượng được khởi tạo, kể cả giữa các package khác nhau, đều được xử lý đúng.

Constants

Hằng số trong Go đúng nghĩa là hằng số. Chúng được tạo ra tại thời điểm biên dịch, kể cả khi được định nghĩa là biến cục bộ trong hàm, và chỉ có thể là số, ký tự (rune), chuỗi hoặc boolean. Do bị giới hạn ở thời điểm biên dịch, các biểu thức định nghĩa chúng phải là biểu thức hằng số, có thể đánh giá bởi trình biên dịch. Ví dụ, `1<<3` là biểu thức hằng số, còn `math.Sin(math.Pi/4)` thì không vì lệnh gọi hàm `math.Sin` cần thực hiện lúc chạy.

Trong Go, hằng số liệt kê được tạo bằng enumerator `iota`. Vì `iota` có thể là một phần của biểu thức và các biểu thức có thể được lặp lại ngầm định, rất dễ xây dựng các tập giá trị phức tạp.

```
type ByteSize float64

const (
    _           = iota // ignore first value by assigning to blank identifier
    KB ByteSize = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

Khả năng gắn phương thức như `String` cho bất kỳ kiểu tự định nghĩa nào giúp các giá trị tùy ý có thể tự động định dạng khi in. Dù thường thấy nhất với struct, kỹ thuật này cũng hữu ích cho các kiểu vô hướng như kiểu số thực như `ByteSize`.

```
func (b ByteSize) String() string {
    switch {
    case b >= YB:
        return fmt.Sprintf("%.2fYB", b/YB)
    case b >= ZB:
        return fmt.Sprintf("%.2fZB", b/ZB)
    case b >= EB:
```

```
        return fmt.Sprintf("%.2fEB", b/EB)
    case b >= PB:
        return fmt.Sprintf("%.2fPB", b/PB)
    case b >= TB:
        return fmt.Sprintf("%.2fTB", b/TB)
    case b >= GB:
        return fmt.Sprintf("%.2fGB", b/GB)
    case b >= MB:
        return fmt.Sprintf("%.2fMB", b/MB)
    case b >= KB:
        return fmt.Sprintf("%.2fKB", b/KB)
    }
    return fmt.Sprintf("%.2fB", b)
}
```

Biểu thức `YB` sẽ in ra `1.00YB`, còn `ByteSize(1e13)` sẽ in ra `9.09TB`.

Việc dùng `Sprintf` để cài đặt phương thức `String` của `ByteSize` là an toàn (tránh đệ quy vô hạn) không phải vì chuyển đổi mà vì nó gọi `Sprintf` với `%f`, không phải định dạng chuỗi: `Sprintf` chỉ gọi phương thức `String` khi nó muốn một chuỗi, còn `%f` muốn giá trị số thực.

Variables

Biến có thể được khởi tạo giống như hằng số nhưng giá trị khởi tạo có thể là biểu thức tổng quát tính toán lúc chạy.

```
var (
    home   = os.Getenv("HOME")
    user   = os.Getenv("USER")
    gopath = os.Getenv("GOPATH")
)
```

The init function

Cuối cùng, mỗi file nguồn có thể định nghĩa hàm `init` không đối số để thiết lập trạng thái cần thiết. (Thực tế mỗi file có thể có nhiều hàm `init`.) Và cuối cùng nghĩa là cuối cùng: `init` được gọi sau khi tất cả khai báo biến trong package đã đánh giá giá trị khởi tạo, và các giá trị đó chỉ được đánh giá sau khi tất cả các package import đã được khởi tạo.

Ngoài các khởi tạo không thể biểu diễn bằng khai báo, một cách dùng phổ biến của hàm `init` là kiểm tra hoặc sửa chữa tính đúng đắn của trạng thái chương trình trước khi thực thi thực sự bắt đầu.

```
func init() {
    if user == "" {
        log.Fatal("$USER not set")
    }
    if home == "" {
        home = "/home/" + user
    }
    if gopath == "" {
        gopath = home + "/go"
    }
```

```
    // gopath may be overridden by --gopath flag on command line.
    flag.StringVar(&gopath, "gopath", gopath, "override default GOPATH")
}
```

Methods

Pointers vs. Values

Như đã thấy với `ByteSize`, phương thức có thể được định nghĩa cho bất kỳ kiểu có tên nào (trừ con trỏ hoặc interface); receiver không nhất thiết phải là struct.

Trong phần thảo luận về slice ở trên, chúng ta đã viết hàm `Append`. Ta có thể định nghĩa nó thành phương thức trên slice. Để làm vậy, đầu tiên khai báo kiểu có tên để gắn phương thức, sau đó làm receiver của phương thức là giá trị kiểu đó.

```
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
    // Body exactly the same as the Append function defined above.
}
```

Điều này vẫn yêu cầu phương thức trả về slice đã cập nhật. Ta có thể loại bỏ sự rườm rà này bằng cách định nghĩa lại phương thức nhận *con trỏ* tới `ByteSlice` làm receiver, để phương thức có thể ghi đè slice của caller.

```
func (p *ByteSlice) Append(data []byte) {
    slice := *p
    // Body as above, without the return.
    *p = slice
}
```

Thực tế, ta còn có thể làm tốt hơn. Nếu sửa hàm để giống phương thức `Write` chuẩn, như sau,

```
func (p *ByteSlice) Write(data []byte) (n int, err error) {
    slice := *p
    // Again as above.
    *p = slice
    return len(data), nil
}
```

thì kiểu `*ByteSlice` sẽ thỏa mãn interface chuẩn `io.Writer`, rất tiện lợi. Ví dụ, ta có thể in vào một biến kiểu này.

```
    var b ByteSlice
    fmt.Fprintf(&b, "This hour has %d days\n", 7)
```

Ta truyền địa chỉ của `ByteSlice` vì chỉ `*ByteSlice` mới thỏa mãn `io.Writer`. Quy tắc về con trỏ và giá trị cho receiver là: phương thức giá trị có thể gọi trên cả con trỏ và giá trị, nhưng phương thức con trỏ chỉ có thể gọi trên con trỏ.

Quy tắc này xuất phát từ việc phương thức con trỏ có thể thay đổi receiver; gọi trên giá trị sẽ khiến phương thức nhận bản sao, nên mọi thay đổi sẽ bị bỏ qua. Ngôn ngữ do đó không cho phép lỗi này. Tuy nhiên, có một ngoại lệ tiện lợi. Khi giá trị có thể lấy địa chỉ, ngôn ngữ sẽ tự động chèn toán tử lấy địa chỉ khi gọi phương thức con trỏ trên giá trị. Trong ví dụ, biến `b` có thể lấy địa chỉ, nên ta có thể gọi phương thức `Write` chỉ với `b.Write`. Trình biên dịch sẽ viết lại thành `(&b).Write` cho ta.

Nhân tiện, ý tưởng dùng `Write` trên slice byte là trung tâm của cài đặt `bytes.Buffer`.

Interfaces and other types

Interfaces

Interfaces trong Go cung cấp cách chỉ định hành vi của một đối tượng: nếu một thứ có thể làm *việc này*, thì nó có thể được dùng *ở đây*. Ta đã thấy một vài ví dụ đơn giản; printer tùy chỉnh có thể được cài đặt bằng phương thức `String` trong khi `Fprintf` có thể xuất ra bất kỳ thứ gì có phương thức `Write`. Interface chỉ có một hoặc hai phương thức rất phổ biến trong mã Go, và thường được đặt tên theo phương thức, như `io.Writer` cho thứ gì đó cài đặt `Write`.

Một kiểu có thể cài đặt nhiều interface. Ví dụ, một collection có thể được sắp xếp bằng các hàm trong package `sort` nếu nó cài đặt `sort.Interface`, gồm `Len()`, `Less(i, j int) bool`, và `Swap(i, j int)`, và nó cũng có thể có formatter tùy chỉnh. Trong ví dụ này, `Sequence` thỏa mãn cả hai.

```
type Sequence []int

// Methods required by sort.Interface.
func (s Sequence) Len() int {
    return len(s)
}
func (s Sequence) Less(i, j int) bool {
    return s[i] < s[j]
}
func (s Sequence) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

// Copy returns a copy of the Sequence.
func (s Sequence) Copy() Sequence {
    copy := make(Sequence, 0, len(s))
    return append(copy, s...)
}

// Method for printing - sorts the elements before printing.
func (s Sequence) String() string {
    s = s.Copy() // Make a copy; don't overwrite argument.
    sort.Sort(s)
    str := "["
    for i, elem := range s { // Loop is O(N²); will fix that in next example.
        if i > 0 {
            str += " "
        }
```

```
        str += fmt.Sprint(elem)
    }
    return str + "]"
}
```

Conversions

Phương thức `String` của `Sequence` đang làm lại công việc mà `Sprint` đã làm cho slice. (Nó cũng có độ phức tạp $O(N^2)$, không tốt.) Ta có thể chia sẻ công sức (và tăng tốc) nếu chuyển đổi `Sequence` thành `[]int` trước khi gọi `Sprint`.

```
func (s Sequence) String() string {
    s = s.Copy()
    sort.Sort(s)
    return fmt.Sprint([]int(s))
}
```

Phương thức này là một ví dụ khác về kỹ thuật chuyển đổi để gọi `Sprintf` an toàn từ phương thức `String`. Vì hai kiểu (`Sequence` và `[]int`) giống nhau nếu bỏ qua tên kiểu, nên hợp pháp để chuyển đổi giữa chúng. Việc chuyển đổi không tạo giá trị mới, chỉ tạm thời coi giá trị hiện tại có kiểu mới. (Có các chuyển đổi hợp pháp khác, như từ số nguyên sang số thực, sẽ tạo giá trị mới.)

Đây là idiom trong chương trình Go để chuyển đổi kiểu biểu thức nhằm truy cập tập phương thức khác. Ví dụ, ta có thể dùng kiểu `sort.IntSlice` hiện có để rút gọn toàn bộ ví dụ thành:

```
type Sequence []int

// Method for printing - sorts the elements before printing
func (s Sequence) String() string {
s = s.Copy()
sort.IntSlice(s).Sort()
return fmt.Sprint(\[\]int(s))
}
```

Bây giờ, thay vì để `Sequence` cài đặt nhiều interface (sắp xếp và in), ta dùng khả năng một dữ liệu có thể chuyển đổi sang nhiều kiểu (`Sequence`, `sort.IntSlice` và `[]int`), mỗi kiểu làm một phần công việc. Điều này không phổ biến trong thực tế nhưng có thể hiệu quả.

Interface conversions and type assertions

Type switch là một dạng chuyển đổi: nó nhận một interface và, với mỗi case trong switch, về bản chất chuyển đổi nó sang kiểu của case đó. Dưới đây là phiên bản đơn giản hóa cách mã dưới `fmt.Printf` chuyển giá trị thành chuỗi bằng type switch. Nếu đã là chuỗi, chúng ta muốn giá trị chuỗi thực sự giữ bởi interface, còn nếu có phương thức `String` chúng ta muốn kết quả của phương thức đó.

```
type Stringer interface {
    String() string
}

var value interface{} // Value provided by caller.
switch str := value.(type) {
case string:
    return str
case Stringer:
    return str.String()
}
```

Trường hợp đầu tìm giá trị cụ thể; trường hợp hai chuyển interface thành interface khác. Việc trộn kiểu như vậy là hoàn toàn hợp lệ.

Nếu chỉ quan tâm một kiểu thì sao? Nếu biết giá trị chứa chuỗi và chỉ muốn lấy ra? Một type switch một case là đủ, nhưng type assertion cũng vậy. Type assertion lấy giá trị interface và trích xuất giá trị kiểu cụ thể. Cú pháp giống phần mở đầu của type switch, nhưng với kiểu cụ thể thay vì từ khóa `type`:

```
value.(typeName)
```

Kết quả là giá trị mới với kiểu tĩnh là kiểu chỉ định. Kiểu đó phải là kiểu cụ thể giữ bởi interface, hoặc interface khác mà giá trị có thể chuyển đổi sang. Để lấy chuỗi ta biết có trong giá trị, có thể viết:

```
str := value.(string)
```

Nhưng nếu giá trị không chứa chuỗi, chương trình sẽ crash với lỗi runtime. Để tránh điều đó, dùng idiom "comma, ok" để kiểm tra an toàn xem giá trị có phải chuỗi không:

```
str, ok := value.(string)
if ok {
    fmt.Printf("string value is: %q\n", str)
} else {
    fmt.Printf("value is not a string\n")
}
```

Nếu type assertion thất bại, `str` vẫn tồn tại và có kiểu string, nhưng là giá trị zero, chuỗi rỗng.

Để minh họa khả năng này, dưới đây là câu lệnh `if-else` tương đương với type switch mở đầu phần này.

```
if str, ok := value.(string); ok {
    return str
} else if str, ok := value.(Stringer); ok {
    return str.String()
}
```

Generality

Nếu một kiểu chỉ tồn tại để cài đặt một interface và sẽ không bao giờ có phương thức export ngoài interface đó, không cần export kiểu đó. Chỉ export interface giúp rõ ràng giá trị không có hành vi nào ngoài những gì mô tả trong interface. Nó cũng tránh phải lặp lại tài liệu trên mỗi instance của phương thức phổ biến.

Trong các trường hợp như vậy, constructor nên trả về giá trị interface thay vì kiểu cài đặt. Ví dụ, trong các thư viện hash, cả `crc32.NewIEEE` và `adler32.New` đều trả về kiểu interface `hash.Hash32`. Thay đổi thuật toán CRC-32 thành Adler-32 trong chương trình Go chỉ cần đổi lệnh gọi constructor; phần còn lại của mã không bị ảnh hưởng bởi thay đổi thuật toán.

Cách tiếp cận tương tự cho phép các thuật toán mã hóa luồng trong các package `crypto` khác nhau tách biệt với các block cipher mà chúng kết hợp. Interface `Block` trong package `crypto/cipher` chỉ định hành vi của block cipher, cung cấp mã hóa một block dữ liệu. Sau đó, tương tự như package `bufio`, các package cipher cài đặt interface này có thể dùng để xây dựng cipher luồng, đại diện bởi interface `Stream`, mà không cần biết chi tiết mã hóa block.
The `crypto/cipher` interfaces look like this:

```
type Block interface {
    BlockSize() int
    Encrypt(dst, src []byte)
    Decrypt(dst, src []byte)
}

type Stream interface {
    XORKeyStream(dst, src []byte)
}
```

Dưới đây là định nghĩa chế độ counter (CTR) stream, biến block cipher thành streaming cipher; lưu ý chi tiết block cipher được trừu tượng hóa:

```
// NewCTR returns a Stream that encrypts/decrypts using the given Block in
// counter mode. The length of iv must be the same as the Block's block size.
func NewCTR(block Block, iv []byte) Stream
```

`NewCTR` không chỉ áp dụng cho một thuật toán mã hóa và nguồn dữ liệu cụ thể mà cho bất kỳ cài đặt nào của interface `Block` và bất kỳ `Stream` nào. Vì trả về giá trị interface, thay đổi mã hóa CTR thành chế độ khác là thay đổi cục bộ. Chỉ cần sửa lệnh gọi constructor, vì mã xung quanh chỉ xử lý kết quả như một `Stream`, nó sẽ không nhận ra sự khác biệt.

Interfaces and methods

Vì gần như bất kỳ thứ gì cũng có thể gắn phương thức, gần như bất kỳ thứ gì cũng có thể thỏa mãn một interface. Một ví dụ minh họa là trong package `http`, định nghĩa interface `Handler`. Bất kỳ đối tượng nào cài đặt `Handler` đều có thể phục vụ HTTP request.

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

`ResponseWriter` bản thân là một interface cung cấp các phương thức cần thiết để trả về phản hồi cho client. Các phương thức này bao gồm phương thức chuẩn `Write`, nên một `http.ResponseWriter` có thể dùng ở bất kỳ đâu cần `io.Writer`. `Request` là struct chứa biểu diễn đã phân tích của request từ client.

Để đơn giản, giả sử HTTP request luôn là GET; điều này không ảnh hưởng đến cách thiết lập handler. Dưới đây là cài đặt đơn giản của handler đếm số lần trang được truy cập.

```
// Simple counter server.
type Counter struct {
    n int
}

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ctr.n++
    fmt.Fprintf(w, "counter = %d\n", ctr.n)
}
```

(Theo chủ đề, lưu ý `Fprintf` có thể in ra `http.ResponseWriter`.) Trong server thực tế, truy cập `ctr.n` cần bảo vệ khỏi truy cập đồng thời. Xem package `sync` và `atomic` để biết gợi ý.

Để gắn server này vào một node trên cây URL:

```
import "net/http"
...
ctr := new(Counter)
http.Handle("/counter", ctr)
```

Nhưng tại sao phải làm `Counter` là struct? Một số nguyên là đủ. (Receiver cần là con trỏ để phép tăng có hiệu lực với caller.)

```
// Simpler counter server.
type Counter int

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    *ctr++
    fmt.Fprintf(w, "counter = %d\n", *ctr)
}
```

Nếu chương trình có trạng thái nội bộ cần được thông báo khi một trang được truy cập? Gắn một channel vào trang web.

```
// A channel that sends a notification on each visit.
// (Probably want the channel to be buffered.)
type Chan chan *http.Request

func (ch Chan) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ch <- req
    fmt.Fprint(w, "notification sent")
}
```

Cuối cùng, giả sử muốn hiển thị trên `/args` các đối số dùng khi chạy server binary. Viết hàm in đối số rất dễ.

```
func ArgServer() {
    fmt.Println(os.Args)
}
```

Làm sao biến nó thành HTTP server? Có thể biến `ArgServer` thành phương thức của kiểu nào đó mà giá trị bị bỏ qua, nhưng có cách sạch hơn. Vì có thể định nghĩa phương thức cho bất kỳ kiểu nào trừ con trỏ và interface, ta có thể viết phương thức cho hàm. Package `http` chứa đoạn mã này:

```
// The HandlerFunc type is an adapter to allow the use of
// ordinary functions as HTTP handlers.  If f is a function
// with the appropriate signature, HandlerFunc(f) is a
// Handler object that calls f.
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, req).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, req *Request) {
    f(w, req)
}
```

`HandlerFunc` là kiểu có phương thức, `ServeHTTP`, nên giá trị kiểu này có thể phục vụ HTTP request. Xem cài đặt phương thức: receiver là hàm, `f`, và phương thức gọi `f`. Điều này có vẻ lạ nhưng không khác nhiều so với receiver là channel và phương thức gửi lên channel.

Để biến `ArgServer` thành HTTP server, đầu tiên sửa nó có đúng chữ ký.

```
// Argument server.
func ArgServer(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, os.Args)
}
```

`ArgServer` giờ có cùng chữ ký với `HandlerFunc`, nên có thể chuyển đổi sang kiểu đó để truy cập phương thức, giống như chuyển đổi `Sequence` sang `IntSlice` để truy cập `IntSlice.Sort`. Mã thiết lập rất ngắn gọn:

```
http.Handle("/args", http.HandlerFunc(ArgServer))
```

Khi ai đó truy cập trang `/args`, handler được gắn tại đó có giá trị là `ArgServer` và kiểu là `HandlerFunc`. HTTP server sẽ gọi phương thức `ServeHTTP` của kiểu đó, với `ArgServer` là receiver, điều này sẽ gọi `ArgServer` (thông qua lệnh gọi `f(w, req)` trong `HandlerFunc.ServeHTTP`). Các đối số sau đó sẽ được hiển thị.

Trong phần này, chúng ta đã tạo một HTTP server từ struct, số nguyên, channel và hàm, tất cả đều nhờ interface là tập hợp phương thức, có thể định nghĩa cho (gần như) bất kỳ kiểu nào.

The blank identifier

Chúng ta đã đề cập đến blank identifier vài lần, trong ngữ cảnh `for range` loop và map. Blank identifier có thể được gán hoặc khai báo với bất kỳ giá trị nào của bất kỳ kiểu nào, giá trị sẽ bị loại bỏ mà không gây hại. Nó giống như ghi vào file `/dev/null` của Unix: đại diện cho giá trị chỉ ghi, dùng làm placeholder khi cần biến nhưng giá trị thực tế không quan trọng. Nó còn nhiều ứng dụng khác ngoài những gì đã thấy.

The blank identifier in multiple assignment

Việc dùng blank identifier trong vòng lặp `for range` là trường hợp đặc biệt của tình huống tổng quát: gán nhiều giá trị.

Nếu một phép gán cần nhiều giá trị bên trái, nhưng một giá trị sẽ không được dùng, blank identifier ở bên trái giúp tránh tạo biến giả và làm rõ giá trị sẽ bị loại bỏ. Ví dụ, khi gọi hàm trả về giá trị và lỗi, nhưng chỉ quan tâm lỗi, dùng blank identifier để bỏ giá trị không cần thiết.

```
if _, err := os.Stat(path); os.IsNotExist(err) {
    fmt.Printf("%s does not exist\n", path)
}
```

Đôi khi bạn sẽ thấy mã bỏ qua giá trị lỗi để bỏ qua lỗi; đây là thói quen rất tệ. Luôn kiểm tra giá trị trả về lỗi; chúng được cung cấp vì lý do chính đáng.

```
// Bad! This code will crash if path does not exist.
fi, _ := os.Stat(path)
if fi.IsDir() {
    fmt.Printf("%s is a directory\n", path)
}
```

Unused imports and variables

Việc import package hoặc khai báo biến mà không dùng là lỗi. Import không dùng làm chương trình phình to và chậm biên dịch, còn biến khởi tạo mà không dùng ít nhất là lãng phí tính toán và có thể là dấu hiệu bug lớn hơn. Khi chương trình đang phát triển, import và biến không dùng thường xuất hiện và việc xóa chúng chỉ để biên dịch, rồi lại cần dùng sau, có thể gây phiền toái. Blank identifier cung cấp giải pháp tạm thời.
This half-written program has two unused imports (`fmt` and `io`) and an unused variable (`fd`), so it will not compile, but it would be nice to see if the code so far is correct.

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: use fd.
}
```

Để loại bỏ cảnh báo về import không dùng, dùng blank identifier để tham chiếu đến một symbol từ package import. Tương tự, gán biến không dùng `fd` cho blank identifier sẽ loại bỏ lỗi biến không dùng. Phiên bản này của chương trình sẽ biên dịch.

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)

var _ = fmt.Printf // For debugging; delete when done.
var _ io.Reader    // For debugging; delete when done.

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: use fd.
    _ = fd
}
```

Theo quy ước, khai báo toàn cục để loại bỏ lỗi import nên đặt ngay sau phần import và có chú thích, vừa dễ tìm vừa nhắc nhở dọn dẹp sau.

Import for side effect

Một import không dùng như `fmt` hoặc `io` trong ví dụ trên cuối cùng nên được dùng hoặc xóa: gán blank identifier xác định mã đang trong quá trình phát triển. Nhưng đôi khi hữu ích khi import package chỉ để lấy hiệu ứng phụ, không dùng trực tiếp. Ví dụ, trong hàm `init`, package `[net/http/pprof](/pkg/net/http/pprof/)` đăng ký HTTP handler

cung cấp thông tin debug. Nó có API export, nhưng hầu hết client chỉ cần đăng ký handler và truy cập dữ liệu qua web. Để import package chỉ lấy hiệu ứng phụ, đổi tên package thành blank identifier:

```
import _ "net/http/pprof"
```

Dạng import này làm rõ package được import chỉ để lấy hiệu ứng phụ, vì không thể dùng package theo cách khác: trong file này, nó không có tên. (Nếu có tên mà không dùng, trình biên dịch sẽ báo lỗi.)

Interface checks

Như đã thấy ở phần interface ở trên, một kiểu không cần khai báo rõ ràng rằng nó cài đặt interface. Thay vào đó, kiểu cài đặt interface chỉ bằng cách cài đặt các phương thức của interface. Trong thực tế, hầu hết chuyển đổi interface là tĩnh và được kiểm tra lúc biên dịch. Ví dụ, truyền `*os.File` cho hàm nhận `io.Reader` sẽ không biên dịch trừ khi `*os.File` cài đặt interface `io.Reader`.

Một số kiểm tra interface xảy ra lúc chạy. Một ví dụ là trong package `[encoding/json](/pkg/encoding/json/)`, định nghĩa interface `[Marshaler](/pkg/encoding/json/#Marshaler)`. Khi encoder JSON nhận giá trị cài đặt interface này, nó sẽ gọi phương thức marshaling của giá trị để chuyển thành JSON thay vì chuyển đổi chuẩn. Encoder kiểm tra thuộc tính này lúc chạy bằng type assertion như sau:

```
m, ok := val.(json.Marshaler)
```

Nếu chỉ cần kiểm tra kiểu có cài đặt interface mà không dùng interface, có thể dùng blank identifier để bỏ qua giá trị type-asserted:

```
if _, ok := val.(json.Marshaler); ok {
    fmt.Printf("value %v of type %T implements json.Marshaler\n", val, val)
}
```

Một trường hợp xuất hiện là khi cần đảm bảo trong package cài đặt kiểu rằng nó thực sự thỏa mãn interface. Nếu một kiểu—ví dụ `[json.RawMessage](/pkg/encoding/json/#RawMessage)`—cần biểu diễn JSON tùy chỉnh, nó nên cài đặt `json.Marshaler`, nhưng không có chuyển đổi tĩnh nào khiến trình biên dịch kiểm tra tự động. Nếu kiểu vô tình không thỏa mãn interface, encoder JSON vẫn hoạt động nhưng không dùng cài đặt tùy chỉnh. Để đảm bảo cài đặt đúng, có thể dùng khai báo toàn cục với blank identifier trong package:

```
var _ json.Marshaler = (*RawMessage)(nil)
```

Trong khai báo này, phép gán chuyển đổi `*RawMessage` sang `Marshaler` yêu cầu `*RawMessage` cài đặt `Marshaler`, và thuộc tính này sẽ được kiểm tra lúc biên dịch. Nếu interface `json.Marshaler` thay đổi, package này sẽ không biên dịch và ta sẽ biết cần cập nhật.

Việc xuất hiện blank identifier trong cấu trúc này cho thấy khai báo chỉ tồn tại để kiểm tra kiểu, không tạo biến. Đừng làm vậy cho mọi kiểu thỏa mãn interface. Theo quy ước, chỉ dùng khi không có chuyển đổi tĩnh nào trong mã, điều này

hiếm khi xảy ra.

Embedding

Go không cung cấp khái niệm kế thừa kiểu truyền thống, nhưng có khả năng "mượn" các phần của cài đặt bằng cách *nhúng* kiểu vào struct hoặc interface.

Nhúng interface rất đơn giản. Ta đã đề cập đến interface `io.Reader` và `io.Writer` trước đó; dưới đây là định nghĩa của chúng.

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Package `io` cũng export một số interface khác chỉ định đối tượng có thể cài đặt nhiều phương thức như vậy. Ví dụ, có interface `io.ReadWriter`, chứa cả `Read` và `Write`. Ta có thể chỉ định `io.ReadWriter` bằng cách liệt kê hai phương thức, nhưng dễ và gợi nhớ hơn khi nhúng hai interface để tạo interface mới như sau:

```
// ReadWriter is the interface that combines the Reader and Writer interfaces.
type ReadWriter interface {
    Reader
    Writer
}
```

Điều này nói đúng như nó thể hiện: Một `ReadWriter` có thể làm những gì `Reader` làm *và* những gì `Writer` làm; nó là hợp nhất của các interface được nhúng. Chỉ interface mới có thể nhúng vào interface.

Ý tưởng cơ bản tương tự áp dụng cho struct, nhưng với nhiều hệ quả hơn. Package `bufio` có hai kiểu struct, `bufio.Reader` và `bufio.Writer`, mỗi cái đều cài đặt interface tương ứng từ package `io`. Và `bufio` cũng cài đặt buffered reader/writer, bằng cách kết hợp reader và writer vào một struct bằng nhúng: liệt kê các kiểu trong struct mà không đặt tên trường.

```
// ReadWriter stores pointers to a Reader and a Writer.
// It implements io.ReadWriter.
type ReadWriter struct {
    *Reader  // *bufio.Reader
    *Writer  // *bufio.Writer
}
```

Các phần tử nhúng là con trỏ đến struct và tất nhiên phải được khởi tạo trỏ đến struct hợp lệ trước khi dùng. Struct `ReadWriter` có thể được viết như

```
type ReadWriter struct {
    reader *Reader
    writer *Writer
}
```

nhưng khi đó để promote các phương thức của trường và thỏa mãn interface `io`, ta cũng cần cung cấp các phương thức chuyển tiếp, như sau:

```
func (rw *ReadWriter) Read(p []byte) (n int, err error) {
    return rw.reader.Read(p)
}
```

Bằng cách nhúng struct trực tiếp, ta tránh được việc ghi mã chuyển tiếp. Các phương thức của kiểu nhúng được "kế thừa" miễn phí, nghĩa là `bufio.ReadWriter` không chỉ có các phương thức của `bufio.Reader` và `bufio.Writer`, mà còn thỏa mãn cả ba interface: `io.Reader`, `io.Writer`, và `io.ReadWriter`.

Có một điểm quan trọng mà nhúng khác với kế thừa. Khi nhúng một kiểu, các phương thức của kiểu đó trở thành phương thức của kiểu ngoài, nhưng khi gọi, receiver của phương thức là kiểu trong, không phải kiểu ngoài. Trong ví dụ, khi gọi phương thức `Read` của `bufio.ReadWriter`, nó có hiệu ứng giống hệt phương thức chuyển tiếp ở trên; receiver là trường `reader` của `ReadWriter`, không phải chính `ReadWriter`.

Nhúng cũng có thể là tiện ích đơn giản. Ví dụ này cho thấy một trường nhúng bên cạnh trường có tên thông thường.

```
type Job struct {
    Command string
    *log.Logger
}
```

Kiểu `Job` giờ có các phương thức `Print`, `Printf`, `Println` và các phương thức khác của `*log.Logger`. Ta có thể đặt tên trường cho `Logger`, nhưng không cần thiết. Và giờ, sau khi khởi tạo, ta có thể ghi log cho `Job`:

```
job.Println("starting now...")
```

`Logger` là trường thông thường của struct `Job`, nên có thể khởi tạo theo cách thông thường trong constructor cho `Job`, như sau:

```
func NewJob(command string, logger *log.Logger) *Job {
    return &Job{command, logger}
}
```

hoặc với composite literal,

```

```

```
job := &Job{command, log.New(os.Stderr, "Job: ", log.Ldate)}
```

Nếu cần truy cập trực tiếp trường nhúng, tên kiểu của trường, bỏ qua qualifier package, đóng vai trò là tên trường, như trong phương thức `Read` của struct `ReadWriter`. Ở đây, nếu cần truy cập `*log.Logger` của biến `job`, ta viết `job.Logger`, hữu ích nếu muốn mở rộng phương thức của `Logger`.

```
func (job *Job) Printf(format string, args ...interface{}) {
    job.Logger.Printf("%q: %s", job.Command, fmt.Sprintf(format, args...))
}
```

Nhúng kiểu dẫn đến vấn đề trùng tên nhưng quy tắc giải quyết rất đơn giản. Đầu tiên, trường hoặc phương thức `X` sẽ che khuất bất kỳ mục `X` nào ở mức lồng ghép sâu hơn. Nếu `log.Logger` chứa trường hoặc phương thức tên `Command`, trường `Command` của `Job` sẽ chiếm ưu thế.

Thứ hai, nếu cùng một tên xuất hiện ở cùng cấp lồng ghép, thường là lỗi; sẽ báo lỗi nếu cố nhúng `log.Logger` nếu struct `Job` đã có trường hoặc phương thức tên `Logger`. Tuy nhiên, nếu tên trùng lặp không bao giờ được tham chiếu ngoài định nghĩa kiểu, thì không sao. Điều này giúp tránh các thay đổi đối với kiểu nhúng từ bên ngoài; không vấn đề gì nếu trường mới thêm vào xung đột với trường trong kiểu con khác nếu không trường nào được dùng.

Concurrency

Share by communicating

Lập trình đồng thời là một chủ đề lớn và ở đây chỉ có thể đề cập đến một số điểm nổi bật riêng của Go.

Lập trình đồng thời trong nhiều môi trường trở nên khó khăn bởi các chi tiết tinh vi cần thiết để truy cập đúng các biến chia sẻ. Go khuyến khích một cách tiếp cận khác, trong đó các giá trị chia sẻ được truyền qua channel và thực tế không bao giờ được chia sẻ chủ động bởi các luồng thực thi riêng biệt. Chỉ một goroutine có quyền truy cập giá trị tại một thời điểm. Thiết kế này loại bỏ khả năng xảy ra race condition. Để khuyến khích cách nghĩ này, chúng tôi đã tóm tắt thành một khẩu hiệu:

> Đừng chia sẻ bộ nhớ để giao tiếp; hãy giao tiếp để chia sẻ bộ nhớ.

Cách tiếp cận này có thể bị lạm dụng. Ví dụ, đếm tham chiếu có thể tốt nhất bằng cách đặt mutex quanh biến số nguyên. Nhưng ở cấp độ cao, dùng channel để kiểm soát truy cập giúp viết chương trình rõ ràng, đúng đắn dễ dàng hơn.

Một cách nghĩ về mô hình này là xem xét một chương trình đơn luồng chạy trên một CPU. Nó không cần primitive đồng bộ nào. Giờ chạy thêm một instance như vậy; nó cũng không cần đồng bộ. Giờ cho hai instance giao tiếp; nếu giao tiếp là bộ đồng bộ, vẫn không cần đồng bộ khác. Unix pipeline là ví dụ hoàn hảo cho mô hình này. Dù cách tiếp cận đồng thời của Go bắt nguồn từ Communicating Sequential Processes (CSP) của Hoare, nó cũng có thể xem là tổng quát hóa an toàn kiểu của Unix pipe.

Goroutines

Chúng được gọi là *goroutine* vì các thuật ngữ hiện có—thread, coroutine, process, v.v.—mang ý nghĩa không chính xác. Goroutine có mô hình đơn giản: là một hàm thực thi đồng thời với các goroutine khác trong cùng không gian địa chỉ.

Nó nhẹ, tốn ít hơn nhiều so với cấp phát stack. Stack bắt đầu nhỏ, nên rẻ, và tăng bằng cách cấp phát (và giải phóng) bộ nhớ heap khi cần.

Goroutine được multiplex lên nhiều thread hệ điều hành nên nếu một goroutine bị block, ví dụ khi chờ I/O, các goroutine khác vẫn tiếp tục chạy. Thiết kế này ẩn đi nhiều phức tạp của việc tạo và quản lý thread.

Thêm từ khóa `go` trước lời gọi hàm hoặc phương thức để chạy lời gọi trong goroutine mới. Khi lời gọi hoàn thành, goroutine sẽ thoát, không thông báo. (Hiệu ứng giống như ký hiệu `&` của shell Unix để chạy lệnh nền.)

```
go list.Sort() // run list.Sort concurrently; don't wait for it.
```

Hàm literal rất tiện khi gọi trong goroutine.

```
func Announce(message string, delay time.Duration) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }()  // Note the parentheses - must call the function.
}
```

Trong Go, hàm literal là closure: trình biên dịch đảm bảo các biến được tham chiếu bởi hàm tồn tại miễn là còn hoạt động.

Các ví dụ trên chưa thực tế vì hàm không có cách báo hiệu hoàn thành. Để làm vậy, chúng tôi cần channel.

Channels

Giống như map, channel được cấp phát bằng `make`, và giá trị trả về sẽ là một tham chiếu đến cấu trúc dữ liệu bên dưới. Nếu bạn cung cấp một tham số số nguyên tùy chọn, nó sẽ đặt kích thước bộ đệm (buffer) cho channel. Mặc định là 0, tức là channel không buffer (unbuffered) hoặc đồng bộ (synchronous).

```
ci := make(chan int) // unbuffered channel of integers
cj := make(chan int, 0) // unbuffered channel of integers
cs := make(chan *os.File, 100) // buffered channel of pointers to Files
```

Channel không buffer kết hợp giữa giao tiếp—trao đổi giá trị—và đồng bộ hóa—đảm bảo rằng hai phép tính (goroutine) ở trạng thái xác định.

Có rất nhiều idiom hay khi sử dụng channel. Dưới đây là một ví dụ khởi đầu. Ở phần trước, chúng ta đã khởi động một thao tác sắp xếp ở nền. Channel có thể cho phép goroutine khởi động chờ cho đến khi thao tác sắp xếp hoàn thành.

```
c := make(chan int)  // Allocate a channel.
// Start the sort in a goroutine; when it completes, signal on the channel.
go func() {
    list.Sort()
    c <- 1  // Send a signal; value does not matter.
```

```
}()
doSomethingForAWhile()
<-c   // Wait for sort to finish; discard sent value.
```

Receiver (bên nhận) luôn block cho đến khi có dữ liệu để nhận. Nếu channel không buffer, sender (bên gửi) sẽ block cho đến khi receiver nhận được giá trị. Nếu channel có buffer, sender chỉ block cho đến khi giá trị được copy vào buffer; nếu buffer đầy, sender sẽ phải chờ cho đến khi một receiver lấy ra một giá trị.

Channel có buffer có thể được sử dụng như một semaphore, ví dụ để giới hạn throughput. Trong ví dụ này, các request đến sẽ được chuyển cho `handle`, hàm này sẽ gửi một giá trị vào channel, xử lý request, và sau đó nhận một giá trị từ channel để "giải phóng semaphore" cho consumer tiếp theo. Dung lượng của buffer channel sẽ giới hạn số lượng lời gọi đồng thời tới `process`.

```
var sem = make(chan int, MaxOutstanding)

func handle(r *Request) {
    sem <- 1    // Wait for active queue to drain.
    process(r)  // May take a long time.
    <-sem       // Done; enable next request to run.
}

func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req)  // Don't wait for handle to finish.
    }
}
```

Khi đã có `MaxOutstanding` handler thực thi `process`, bất kỳ handler nào khác sẽ bị block khi cố gửi vào channel đã đầy, cho đến khi một handler hiện tại hoàn thành và nhận giá trị từ buffer.

Tuy nhiên, thiết kế này có một vấn đề: `Serve` tạo một goroutine mới cho mỗi request đến, mặc dù chỉ có tối đa `MaxOutstanding` goroutine có thể chạy cùng lúc. Kết quả là chương trình có thể tiêu tốn tài nguyên không giới hạn nếu request đến quá nhanh. Chúng ta có thể khắc phục điểm yếu này bằng cách thay đổi `Serve` để kiểm soát việc tạo goroutine.

```
func Serve(queue chan *Request) {
    for req := range queue {
        sem <- 1
        go func() {
            process(req)
            <-sem
        }()
    }
}
```

(Note rằng trong các phiên bản Go trước 1.22, mã này có một lỗi: biến vòng lặp được chia sẻ giữa tất cả các goroutine. Xem Go wiki để biết chi tiết.)

Một cách tiếp cận khác quản lý tài nguyên tốt là khởi động một số lượng cố định các goroutine `handle` tất cả đều đọc từ kênh yêu cầu. Số lượng goroutine giới hạn số cuộc gọi đồng thời đến `process`. Hàm `Serve` này cũng chấp nhận một kênh mà trên đó nó sẽ được thông báo để thoát; sau khi khởi động các goroutine, nó chặn việc nhận từ kênh đó.

```
func handle(queue chan *Request) {
    for r := range queue {
        process(r)
    }
}

func Serve(clientRequests chan *Request, quit chan bool) {
    // Start handlers
    for i := 0; i < MaxOutstanding; i++ {
        go handle(clientRequests)
    }
    <-quit  // Wait to be told to exit.
}
```

Channels of channels

Một trong những đặc tính quan trọng nhất của Go là channel là giá trị hạng nhất, có thể cấp phát và truyền đi như bất kỳ giá trị nào khác. Một ứng dụng phổ biến của đặc tính này là cài đặt demultiplex song song an toàn.

Trong ví dụ ở phần trước, `handle` là một handler lý tưởng cho một request nhưng chúng ta chưa định nghĩa kiểu dữ liệu mà nó xử lý. Nếu kiểu đó bao gồm một channel để trả lời, mỗi client có thể cung cấp đường dẫn riêng để nhận kết quả. Dưới đây là định nghĩa sơ đồ cho kiểu `Request`.

```
type Request struct {
    args        []int
    f           func([]int) int
    resultChan  chan int
}
```

Client sẽ cung cấp một hàm và các tham số của nó, cũng như một channel bên trong đối tượng request để nhận kết quả trả về.

```
func sum(a []int) (s int) {
    for _, v := range a {
        s += v
    }
    return
}

request := &Request{[]int{3, 4, 5}, sum, make(chan int)}
// Send request
clientRequests <- request
// Wait for response.
fmt.Printf("answer: %d\n", <-request.resultChan)
```

Ở phía server, chỉ có hàm handler là thay đổi.

```
func handle(queue chan *Request) {
    for req := range queue {
        req.resultChan <- req.f(req.args)
    }
}
```

Rõ ràng còn rất nhiều việc cần làm để hoàn thiện, nhưng đoạn mã này là một khung cho hệ thống RPC song song, không block, giới hạn tốc độ, và không cần mutex.

Parallelization

Một ứng dụng khác của các ý tưởng này là song song hóa một phép tính trên nhiều lõi CPU. Nếu phép tính có thể được chia thành các phần riêng biệt có thể thực thi độc lập, ta có thể song song hóa nó, sử dụng channel để báo hiệu khi mỗi phần hoàn thành.

Giả sử chúng ta có một phép toán tốn kém cần thực hiện trên một vector các phần tử, và giá trị của phép toán trên mỗi phần tử là độc lập, như trong ví dụ lý tưởng này.

```
type Vector []float64

// Apply the operation to v[i], v[i+1] ... up to v[n-1].
func (v Vector) DoSome(i, n int, u Vector, c chan int) {
    for ; i < n; i++ {
        v[i] += u.Op(v[i])
    }
    c <- 1    // signal that this piece is done
}
```

Chúng ta khởi động các phần độc lập trong một vòng lặp, mỗi phần cho một CPU. Chúng có thể hoàn thành theo bất kỳ thứ tự nào nhưng điều đó không quan trọng; chúng ta chỉ cần đếm các tín hiệu hoàn thành bằng cách rút channel sau khi đã khởi động tất cả các goroutine.

```
const numCPU = 4 // number of CPU cores

func (v Vector) DoAll(u Vector) {
    c := make(chan int, numCPU)  // Buffering optional but sensible.
    for i := 0; i < numCPU; i++ {
        go v.DoSome(i*len(v)/numCPU, (i+1)*len(v)/numCPU, u, c)
    }
    // Drain the channel.
    for i := 0; i < numCPU; i++ {
        <-c    // wait for one task to complete
    }
    // All done.
}
```

Thay vì tạo hằng số cho numCPU, chúng ta có thể hỏi runtime giá trị phù hợp. Hàm [runtime.NumCPU](/pkg/runtime#NumCPU) trả về số lõi CPU phần cứng trên máy, nên có thể viết

```
var numCPU = runtime.NumCPU()
```

Có hàm [runtime.GOMAXPROCS](/pkg/runtime#GOMAXPROCS), báo cáo (hoặc đặt) số lõi mà chương trình Go có thể chạy đồng thời. Mặc định là giá trị của runtime.NumCPU nhưng có thể ghi đè bằng biến môi trường hoặc gọi hàm với số dương. Gọi với 0 chỉ để truy vấn giá trị. Do đó, nếu muốn tôn trọng yêu cầu tài nguyên của người dùng, nên viết

```
var numCPU = runtime.GOMAXPROCS(0)
```

Hãy chắc chắn không nhầm lẫn giữa đồng thời—cấu trúc chương trình thành các thành phần thực thi độc lập—và song song—thực thi tính toán song song để tăng hiệu suất trên nhiều CPU. Dù tính năng đồng thời của Go giúp một số vấn đề dễ cấu trúc thành tính toán song song, Go là ngôn ngữ đồng thời, không phải song song, và không phải mọi vấn đề song song hóa đều phù hợp với mô hình Go. Để thảo luận về sự khác biệt, xem bài nói được dẫn trong blog post này.

A leaky buffer

Công cụ lập trình đồng thời thậm chí có thể làm cho các ý tưởng không đồng thời dễ biểu diễn hơn. Dưới đây là ví dụ trừu tượng từ package RPC. Goroutine client lặp nhận dữ liệu từ nguồn nào đó, có thể là mạng. Để tránh cấp phát và giải phóng buffer, nó giữ một danh sách free, và dùng channel buffer để biểu diễn. Nếu channel rỗng, cấp phát buffer mới. Khi buffer message sẵn sàng, gửi cho server qua serverChan.

```
var freeList = make(chan *Buffer, 100)
var serverChan = make(chan *Buffer)

func client() {
    for {
        var b *Buffer
        // Grab a buffer if available; allocate if not.
        select {
        case b = <-freeList:
            // Got one; nothing more to do.
        default:
            // None free, so allocate a new one.
            b = new(Buffer)
        }
        load(b)              // Read next message from the net.
        serverChan <- b      // Send to server.
    }
}
```

Vòng lặp server nhận mỗi message từ client, xử lý, và trả buffer về danh sách free.

```
func server() {
    for {
        b := <-serverChan    // Wait for work.
        process(b)
        // Reuse buffer if there's room.
        select {
        case freeList <- b:
            // Buffer on free list; nothing more to do.
        default:
            // Free list full, just carry on.
        }
    }
}
```

Client cố lấy buffer từ `freeList`; nếu không có, cấp phát mới. Server gửi lại buffer vào `freeList` trừ khi danh sách đầy, khi đó buffer sẽ bị bỏ và garbage collector sẽ thu hồi. (Các nhánh `default` trong lệnh `select` thực thi khi không có case nào sẵn sàng, nghĩa là `select` không bao giờ block.) Cài đặt này xây dựng danh sách free kiểu leaky bucket chỉ trong vài dòng, dựa vào channel buffer và garbage collector để quản lý.

Errors

Các hàm thư viện thường phải trả về một dạng chỉ báo lỗi nào đó cho caller. Như đã đề cập trước đó, khả năng trả về nhiều giá trị của Go giúp dễ dàng trả về mô tả lỗi chi tiết cùng với giá trị trả về thông thường. Việc sử dụng tính năng này để cung cấp thông tin lỗi chi tiết là một phong cách tốt. Ví dụ, như chúng ta sẽ thấy, `os.Open` không chỉ trả về con trỏ `nil` khi thất bại, mà còn trả về một giá trị lỗi mô tả nguyên nhân.

Theo quy ước, lỗi có kiểu `error`, một interface tích hợp đơn giản.

```
type error interface {
    Error() string
}
```

Người viết thư viện có thể tự do cài đặt interface này với mô hình phong phú hơn bên dưới, cho phép không chỉ xem lỗi mà còn cung cấp thêm ngữ cảnh. Như đã đề cập, cùng với giá trị trả về thông thường `*os.File`, `os.Open` cũng trả về một giá trị lỗi. Nếu file mở thành công, lỗi sẽ là `nil`, nhưng khi có vấn đề, nó sẽ chứa một `os.PathError`:

```
// PathError records an error and the operation and
// file path that caused it.
type PathError struct {
    Op string    // "open", "unlink", etc.
    Path string  // The associated file.
    Err error    // Returned by the system call.
}

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

`PathError`'s `Error` generates a string like this:

```
open /etc/passwx: no such file or directory
```

Một lỗi như vậy, bao gồm tên file gặp sự cố, thao tác và lỗi hệ điều hành liên quan, rất hữu ích ngay cả khi được in ra ở xa nơi gọi gây ra lỗi; nó cung cấp nhiều thông tin hơn so với thông báo đơn giản "no such file or directory".

Khi có thể, chuỗi lỗi nên xác định nguồn gốc của nó, ví dụ bằng tiền tố tên thao tác hoặc package sinh ra lỗi. Ví dụ, trong package `image`, chuỗi lỗi khi giải mã định dạng không xác định là "image: unknown format".

Caller quan tâm đến chi tiết lỗi có thể sử dụng type switch hoặc type assertion để tìm kiếm lỗi cụ thể và trích xuất thông tin chi tiết. Với `PathErrors`, điều này có thể bao gồm việc kiểm tra trường `Err` bên trong cho các lỗi có thể phục hồi.

```
for try := 0; try < 2; try++ {
    file, err = os.Create(filename)
    if err == nil {
        return
    }
    if e, ok := err.(*os.PathError); ok && e.Err == syscall.ENOSPC {
        deleteTempFiles()  // Recover some space.
        continue
    }
    return
}
```

Câu lệnh `if` thứ hai ở đây là một type assertion khác. Nếu thất bại, `ok` sẽ là false và `e` sẽ là `nil`. Nếu thành công, `ok` sẽ là true, nghĩa là lỗi thuộc kiểu `*os.PathError`, và khi đó `e` cũng vậy, cho phép chúng ta kiểm tra thêm thông tin về lỗi.

Panic

Cách thông thường để báo lỗi cho caller là trả về một giá trị `error` như một giá trị trả về bổ sung. Phương thức `Read` chuẩn là một ví dụ nổi tiếng; nó trả về số byte và một `error`. Nhưng nếu lỗi không thể phục hồi thì sao? Đôi khi chương trình đơn giản là không thể tiếp tục.

Với mục đích này, Go cung cấp hàm tích hợp `panic`, về bản chất tạo ra một lỗi runtime sẽ dừng chương trình (xem phần tiếp theo). Hàm này nhận một đối số bất kỳ—thường là một chuỗi—để in ra khi chương trình dừng. Đây cũng là cách để báo hiệu rằng đã xảy ra điều gì đó không thể xảy ra, ví dụ như thoát khỏi một vòng lặp vô hạn.

```
// A toy implementation of cube root using Newton's method.
func CubeRoot(x float64) float64 {
    z := x/3   // Arbitrary initial value
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z-x) / (3*z*z)
        if veryClose(z, prevz) {
            return z
        }
```

```
    }
    // A million iterations has not converged; something is wrong.
    panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))
}
```

Đây chỉ là ví dụ, nhưng các hàm thư viện thực tế nên tránh sử dụng `panic`.
Nếu có thể che giấu hoặc xử lý vấn đề, luôn tốt hơn để chương trình tiếp tục chạy thay vì dừng toàn bộ. Một ngoại lệ có thể là khi khởi tạo: nếu thư viện thực sự không thể tự thiết lập, có thể hợp lý khi panic.

```
var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

Recover

Khi gọi `panic`, kể cả ngầm định do lỗi runtime như truy cập ngoài chỉ số slice hoặc type assertion thất bại, nó ngay lập tức dừng thực thi hàm hiện tại và bắt đầu unwind stack của goroutine, chạy bất kỳ hàm deferred nào trên đường đi. Nếu unwind đến đỉnh stack goroutine, chương trình sẽ dừng. Tuy nhiên, có thể dùng hàm tích hợp `recover` để lấy lại quyền kiểm soát goroutine và tiếp tục thực thi bình thường.

A call to `recover` stops the unwinding and returns the argument passed to `panic`. Because the only code that runs while unwinding is inside deferred functions, `recover` is only useful inside deferred functions.

Một ứng dụng của `recover` là tắt goroutine lỗi trong server mà không làm chết các goroutine khác.

```
func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)
    }
}

func safelyDo(work *Work) {
    defer func() {
        if err := recover(); err != nil {
            log.Println("work failed:", err)
        }
    }()
    do(work)
}
```

Trong ví dụ này, nếu `do(work)` panic, kết quả sẽ được log và goroutine sẽ thoát sạch sẽ mà không ảnh hưởng đến goroutine khác. Không cần làm gì thêm trong closure deferred; gọi `recover` xử lý hoàn toàn tình huống.

Vì `recover` luôn trả về nil trừ khi gọi trực tiếp từ hàm deferred, mã deferred có thể gọi hàm thư viện cũng dùng `panic` và `recover` mà không bị lỗi. Ví dụ, hàm deferred trong `safelyDo` có thể gọi hàm log trước khi gọi `recover`, và mã log đó sẽ chạy không bị ảnh hưởng bởi trạng thái panic.

Với mẫu recover này, hàm `do` (và bất kỳ hàm nào nó gọi) có thể thoát khỏi bất kỳ tình huống xấu nào bằng cách gọi `panic`. Có thể dùng ý tưởng này để đơn giản hóa xử lý lỗi trong phần mềm phức tạp. Xem phiên bản lý tưởng hóa của package `regexp`, báo lỗi phân tích cú pháp bằng cách gọi `panic` với kiểu lỗi cục bộ. Dưới đây là định nghĩa `Error`, phương thức `error`, và hàm `Compile`.

```
// Error is the type of a parse error; it satisfies the error interface.
type Error string
func (e Error) Error() string {
    return string(e)
}

// error is a method of *Regexp that reports parsing errors by
// panicking with an Error.
func (regexp *Regexp) error(err string) {
    panic(Error(err))
}

// Compile returns a parsed representation of the regular expression.
func Compile(str string) (regexp *Regexp, err error) {
    regexp = new(Regexp)
    // doParse will panic if there is a parse error.
    defer func() {
        if e := recover(); e != nil {
            regexp = nil    // Clear return value.
            err = e.(Error) // Will re-panic if not a parse error.
        }
    }()
    return regexp.doParse(str), nil
}
```

Nếu `doParse` panic, khối recover sẽ đặt giá trị trả về thành nil—hàm deferred có thể sửa giá trị trả về đã đặt tên. Nó sẽ kiểm tra, khi gán cho `err`, rằng vấn đề là lỗi phân tích cú pháp bằng cách type assertion sang kiểu cục bộ `Error`. Nếu không phải, type assertion sẽ thất bại, gây lỗi runtime tiếp tục unwind stack như chưa có gì ngắt quãng. Kiểm tra này đảm bảo nếu có gì bất ngờ xảy ra, như truy cập ngoài chỉ số, mã vẫn sẽ lỗi dù dùng `panic` và `recover` để xử lý lỗi phân tích cú pháp.

Với xử lý lỗi này, phương thức `error` (vì là phương thức gắn với kiểu, hoàn toàn hợp lý, thậm chí tự nhiên, khi trùng tên với kiểu tích hợp `error`) giúp dễ dàng báo lỗi phân tích cú pháp mà không phải tự unwind stack:

```
if pos == 0 {
    re.error("'*' illegal at start of expression")
}
```

Nhân tiện, idiom re-panic này thay đổi giá trị panic nếu có lỗi thực sự. Tuy nhiên, cả lỗi gốc và lỗi mới đều sẽ hiển thị trong báo cáo crash, nên nguyên nhân gốc vẫn thấy được. Do đó, cách re-panic đơn giản này thường đủ—dù sao cũng

là crash—nhưng nếu muốn chỉ hiển thị giá trị gốc, có thể viết thêm mã để lọc vấn đề bất ngờ và re-panic với lỗi gốc. Để lại cho bạn đọc tự thực hiện.

A web server

Hãy kết thúc với một chương trình Go hoàn chỉnh, một web server. Chương trình này thực ra là một dạng web re-server. Google cung cấp dịch vụ tại `chart.apis.google.com` tự động định dạng dữ liệu thành biểu đồ và đồ thị. Tuy nhiên, khó dùng tương tác vì cần đưa dữ liệu vào URL dưới dạng truy vấn. Chương trình này cung cấp giao diện thân thiện hơn cho một dạng dữ liệu: cho một đoạn văn bản ngắn, nó gọi server chart để tạo mã QR, một ma trận ô vuông mã hóa văn bản. Ảnh này có thể quét bằng camera điện thoại và giải mã thành, ví dụ, một URL, giúp bạn không phải gõ URL trên bàn phím nhỏ của điện thoại.
Here's the complete program. An explanation follows

```
package main

import (
    "flag"
    "html/template"
    "log"
    "net/http"
)

var addr = flag.String("addr", ":1718", "http service address") // Q=17, R=18

var templ = template.Must(template.New("qr").Parse(templateStr))

func main() {
    flag.Parse()
    http.Handle("/", http.HandlerFunc(QR))
    err := http.ListenAndServe(*addr, nil)
    if err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}

func QR(w http.ResponseWriter, req *http.Request) {
    templ.Execute(w, req.FormValue("s"))
}

const templateStr = `
<html>
<head>
<title>QR Link Generator</title>
</head>
<body>
{{if .}}
<img src="http://chart.apis.google.com/chart?chs=300x300&cht=qr&choe=UTF-8&chl={{.}}" />
<br>
{{.}}
<br>
<br>
{{end}}
```

```
<form action="/" name=f method="GET">
    <input maxLength=1024 size=70 name=s value="" title="Text to QR Encode">
    <input type=submit value="Show QR" name=qr>
</form>
</body>
</html>
`
```

Các phần đến `main` khá dễ theo dõi. Cờ duy nhất đặt cổng HTTP mặc định cho server. Biến template `templ` là nơi thú vị. Nó xây dựng một template HTML sẽ được server thực thi để hiển thị trang; hơn về điều đó trong một lúc.

Hàm `main` phân tích cờ và, sử dụng cơ chế đã nói ở trên, gắn hàm `QR` vào đường dẫn gốc của server. Sau đó gọi `http.ListenAndServe` để khởi động server; nó block trong khi server chạy.

`QR` chỉ nhận request, chứa dữ liệu form, và thực thi template trên dữ liệu trong form value tên `s`.

Package template `html/template` rất mạnh; chương trình này chỉ chạm đến một phần khả năng của nó. Về bản chất, nó viết lại một đoạn HTML động bằng cách thay thế các phần tử lấy từ dữ liệu truyền vào `templ.Execute`, ở đây là giá trị form. Trong template (`templateStr`), các phần nằm giữa dấu ngoặc kép kép là hành động template. Đoạn từ `{{if .}}` đến `{{end}}` chỉ thực thi nếu giá trị dữ liệu hiện tại, gọi là `.` (dot), không rỗng. Nghĩa là, khi chuỗi rỗng, phần này của template sẽ bị ẩn.

Hai đoạn `{{.}}` dùng để hiển thị dữ liệu truyền vào template—chuỗi truy vấn—trên trang web. Package template HTML tự động cung cấp escape phù hợp để văn bản an toàn khi hiển thị.

Phần còn lại của template chỉ là HTML hiển thị khi trang tải. Nếu giải thích này quá nhanh, xem tài liệu để biết thêm chi tiết.

Vậy là bạn đã có: một web server hữu ích chỉ với vài dòng mã cộng thêm một ít HTML động. Go đủ mạnh để làm được nhiều việc chỉ trong vài dòng mã.