Goroutine & Channel

Điều kiện để **gửi thông tin vào channel** trong Go là:

- **Phải có một goroutine khác đang nhận (receive) từ channel đó** hoặc channel phải có **buffer còn trống** (nếu là buffered channel).
- Nếu channel là **unbuffered** (không có buffer), lệnh gửi (`channel <- value`) sẽ **block** (chờ) cho đến khi có một goroutine khác nhận giá trị từ channel đó.
- Nếu channel là **buffered** (có buffer), lệnh gửi sẽ chỉ block khi buffer đã đầy.

Tóm lại:

- Channel unbuffered: cần có goroutine nhận thì mới gửi được.
- Channel buffered: gửi được nếu buffer chưa đầy, nếu đầy thì phải chờ có goroutine nhận bớt giá trị khỏi channel.

Ví dụ:

```
ch := make(chan int) // unbuffered
go func() {
    ch <- 1 // sẽ block nếu không có ai nhận
}()
<-ch // nhận, giải phóng block
```

```
ch := make(chan int, 2) // buffered
ch <- 1 // gửi được ngay
ch <- 2 // gửi được ngay
// ch <- 3 // sẽ block vì buffer đầy nếu chưa có ai nhận
```

Phần Channels of Channels:

Phần này thực sự rất hay, thể hiện sức mạch của combo Goroutine và channel. Có thể ứng dụng là: dùng Goroutine chính làm trung tâm điều phối công việc cho các luồng goroutine con, thực hiện các hàm/feature, rồi đợi nhận kết quả để tiếp tục công việc. Thay cho cách thiết kế truyền thống là 1 luồng chính duy nhất chạy từ trên xuống dưới.
Ví dụ như app web:

- Phía client gửi thông tin cần xử lý dạng json chứa các thông tin như: dữ liệu đầu vào cần xử lý(username, password, info...), loại tính năng muốn dùng phía server(xác thực, truy cập SQL, lấy bài đăng... cần thống nhất tên gọi với phía server để server gọi đúng tên tính năng đó), các tham số option khác. Hoạt động như 1 thống Redux - Action - State.
- Phía server phân tích JSON, dùng hàm phân phối trung gian để đưa tạo ra biến có kiểu như *Request{Input, Feature, ChannelResult}. Truyền biến này vào Các channel để giao việc cho các Goroutine worker đang chờ làm việc: `bỏ Intput, các tham số option vào hàm tính năng như Feature và RUN`, sau khi xong việc trả kết quả thông qua ChannelResult. Phía Luồng chính đợi nhận kết quả và gửi về cho Client(có thể gửi trực tiếp hay lại giao cho goroutine con làm việc này). Luồng chính chỉ làm: `giao việc và nhận kết quả`

Parallelization

Chủ đề Parallelization trong Golang nói về cách chia nhỏ một tác vụ lớn (ví dụ: xử lý một mảng số thực) thành nhiều phần nhỏ để thực hiện song song trên nhiều CPU core, giúp tăng hiệu suất. Mỗi phần được xử lý độc lập bởi một goroutine, và khi hoàn thành sẽ gửi tín hiệu qua channel để báo đã xong.

Ví dụ trong bài mô tả một kiểu dữ liệu Vector (mảng số thực), và một hàm DoAll sẽ chia mảng này thành nhiều phần, mỗi phần do một goroutine xử lý. Sau khi tất cả goroutine hoàn thành, chương trình mới tiếp tục.

Đoạn mã `i*len(v)/numCPU, (i+1)*len(v)/numCPU` chỉ là 1 thủ thuật nhỏ dùng để chia đều mảng v thành numCPU phần:

- i chạy từ 0 đến numCPU-1.
- i*len(v)/numCPU là chỉ số bắt đầu của phần thứ i.
- (i+1)*len(v)/numCPU là chỉ số kết thúc (không bao gồm) của phần thứ i.

Ví dụ: Nếu len(v) = 10, numCPU = 4, các phần sẽ là:

- i=0: 0*10/4=0 đến 1*10/4=2 (phần tử 0,1)
- i=1: 2 đến 5 (phần tử 2,3,4)
- i=2: 5 đến 7 (phần tử 5,6)
- i=3: 7 đến 10 (phần tử 7,8,9)

Nếu phép chia không đều (số lẻ), một số phần sẽ có nhiều phần tử hơn, nhưng tổng lại vẫn đủ hết các phần tử của mảng. Đây là cách chia đều nhất có thể mà không bị sót hoặc trùng phần tử.

Tóm lại: Chủ đề này hướng dẫn cách chia nhỏ và xử lý song song một tác vụ lớn bằng goroutine và channel trong Go, và đoạn chia chỉ số là để phân chia công việc cho các goroutine.

A leaky buffer

Chủ đề **A leaky buffer** trong Go mô tả một kỹ thuật quản lý bộ nhớ tạm (buffer) hiệu quả khi làm việc với các tác vụ lặp đi lặp lại như nhận/gửi dữ liệu qua mạng.

- Nếu không dùng kỹ thuật này thì mỗi lần cần dùng đến Buffer như đọc file, hay các tác vụ đòi hỏi cấp phát bộ nhớ RAM. Thì cần yêu cầu cấp phát bộ nhớ, sau khi dùng xong bị GC dọn dẹp.

- Khi dùng kỹ thuật này như trong ví dụ là chuẩn bị trước 100 Buffer(ô nhớ), sau đó ghi lại address tham chiếu đến đó. Sau khi thực hiện xong tác vụ thì trả Buffer về danh sách chờ để phục vụ cho công việc tiếp theo. Nên tác vụ sẽ nhanh hơn do không cần đợi hệ thống cấp phát địa chỉ ô nhớ mỗi khi cần dùng.
  Ý chính của ví dụ:

- **freeList** là một buffered channel dùng để tái sử dụng các buffer đã dùng xong, tránh phải cấp phát và bỏ bộ nhớ liên tục gây mất time chờ xử lý.

- **client** lấy buffer từ freeList (nếu có), hoặc tạo mới nếu hết. Sau khi dùng xong, gửi buffer cho server xử lý.

- **server** xử lý xong sẽ cố gắng trả buffer về freeList để tái sử dụng. Nếu freeList đầy, buffer sẽ bị "rơi" (không tái sử dụng), và sẽ được gom rác tự động (garbage collector) dọn dẹp.

Tại sao gọi là "leaky"?

- Vì nếu freeList đầy, buffer sẽ không được giữ lại mà bị bỏ đi ("leak" ra ngoài freeList), nhưng không gây rò rỉ bộ nhớ thực sự vì Go có garbage collector.

Ý nghĩa:

- Giúp giảm số lần cấp phát/bỏ bộ nhớ, tăng hiệu suất.
- Đơn giản hóa quản lý bộ nhớ nhờ tận dụng channel và garbage collector.
- Không cần khóa (lock) phức tạp, tránh deadlock vì các select đều có default (không bao giờ block).

Tóm lại:
"A leaky buffer" là một kỹ thuật dùng buffered channel để quản lý pool bộ nhớ tạm, cho phép tái sử dụng hiệu quả mà không cần quản lý thủ công phức tạp. Nếu pool đầy, buffer dư sẽ bị bỏ đi và gom rác sẽ xử lý.

Error

1. Mở đầu: Xử lý lỗi trong Go

- Go sử dụng cơ chế trả về nhiều giá trị (multi-value return) để trả về cả kết quả và lỗi (error) cùng lúc.
- Kiểu lỗi chuẩn là interface `error` với phương thức `Error() string`, được gọi tự động khi `type triển khai` nó gây `panic`.
- Các hàm thư viện như `os.Open` trả về giá trị lỗi chi tiết (ví dụ: `*os.PathError`), giúp xác định nguyên nhân lỗi rõ ràng hơn là chỉ trả về nil hoặc thông báo chung chung.
- Lỗi nên có thông điệp rõ ràng, bao gồm nguồn gốc (tên hàm, package) để dễ truy vết.
- Có thể dùng `instance.(type assertion)` hoặc `type switch` để kiểm tra và xử lý lỗi cụ thể.
- Mỗi package nên có struct lỗi cục bộ để dễ dàng quản lý và xử lý lỗi nội bộ của package đó. Có thể thêm lỗi public thì dễ kiểm soát hơn.

---

1. Về `PathError.Err` và interface error

- Đúng như bạn nói, trường `Err` trong struct `os.PathError` có kiểu là `error` (interface).
- Điều này có nghĩa là **bất kỳ giá trị nào implement interface `error`** đều có thể được gán vào đây, không chỉ riêng một loại struct lỗi nào.
- `os.PathError` chỉ là một trong nhiều struct lỗi khác trong Go. Ngoài ra còn có các loại lỗi khác như `LinkError`, `SyscallError`, v.v.
- Việc dùng interface `error` giúp Go linh hoạt trong việc truyền tải thông tin lỗi, có thể lồng ghép nhiều lớp lỗi khác nhau.

---

2. Về `Err error // Returned by the system call.` và thông báo hệ thống

- Dòng chú thích này nói rằng trường `Err` thường chứa lỗi trả về từ **system call** (lời gọi hệ thống), ví dụ khi thao tác với file, mạng, v.v.
- Trong thực tế, giá trị của `Err` thường là một **hằng số lỗi hệ thống** (ví dụ: `syscall.ENOSPC`), đại diện cho các lỗi như "No space left on device".

- Các hằng số như `syscall.ENOSPC` là các giá trị đặc biệt (thường là số nguyên) được định nghĩa trong package `syscall`, và chúng cũng implement interface `error` (thông qua phương thức `Error()` trả về chuỗi mô tả lỗi).
- Khi bạn in lỗi ra, Go sẽ gọi `Error()` của từng thành phần, nên bạn sẽ thấy thông báo lỗi hệ thống (ví dụ: "no space left on device").

=> Kết luận:

- Phỏng đoán của bạn ở ý 1 là đúng: `Err` có thể chứa bất kỳ lỗi nào implement interface `error`.
- Tuy nhiên, trong thực tế, với các thao tác hệ thống, nó thường chứa lỗi hệ thống (như `syscall.Errno`), và các giá trị này có thể so sánh trực tiếp với các hằng số như `syscall.ENOSPC`.
- `syscall.ENOSPC` là một hằng số đại diện cho lỗi hệ thống, không phải là một property của struct nào, mà là một giá trị implement interface `error`.
- Đúng như bạn nói, các lỗi hệ thống như syscall.ENOSPC implement interface error, nên có thể gọi Error() để lấy chuỗi mô tả lỗi.
- Nếu lỗi được bọc nhiều lớp (nested error), mỗi lớp sẽ có Error() riêng, thường gọi tiếp Error() của lỗi bên trong để tạo thông báo đầy đủ.

Tóm lại:

- `os.PathError.Err` có thể chứa bất kỳ lỗi nào (miễn là implement interface `error`), nhưng thường là lỗi hệ thống.
- Các lỗi hệ thống như `syscall.ENOSPC` là các hằng số đặc biệt, cũng là một kiểu lỗi trong Go.

---

2. Panic: Khi lỗi không thể phục hồi

- Thông thường, lỗi nên được trả về cho caller xử lý.
- Tuy nhiên, với lỗi nghiêm trọng, không thể phục hồi (unrecoverable error), có thể dùng hàm built-in `panic`.
- `panic` sẽ dừng thực thi hàm hiện tại, bắt đầu quá trình "unwinding" stack, chạy các hàm `defer` và cuối cùng kết thúc goroutine (hoặc chương trình nếu là goroutine chính).
- Ví dụ: Nếu một thuật toán không hội tụ sau nhiều lần lặp, hoặc một hàm thiết lập(lúc khởi động app) ban đầu và quan trọng mà bị lỗi, có thể `panic` với thông báo lỗi.
- Lưu ý: Chỉ nên dùng `panic` khi thực sự không thể tiếp tục.

---

3. Recover: Khắc phục panic và tiếp tục chương trình

- Hàm built-in `recover` cho phép "bắt" lại panic trong các hàm được defer tại caller, giúp chương trình không bị dừng đột ngột.
- `recover` chỉ có tác dụng khi được gọi trực tiếp trong hàm defer.
- Ứng dụng: Có thể dùng để bảo vệ các goroutine, ghi log lỗi và cho phép các goroutine khác tiếp tục chạy.
- Mẫu phổ biến: hàm thực thi công việc bên trong có một hàm defer (closure) có gọi `recover` dự phòng panic, nếu có panic thì ghi log và kết thúc goroutine một cách an toàn.
- Trong các package phức tạp (ví dụ: regexp), có thể dùng panic để báo lỗi nội bộ, sau đó dùng recover để chuyển panic thành giá trị error trả về cho caller, không để panic "thoát" ra ngoài package.

LƯU Ý, CỐT LÕI VẤN ĐỀ PANIC VÀ RECOVER

- `panic` sẽ đưa lỗi ra caller gần nhất.
- Nếu caller không bắt bằng `recover`, panic tiếp tục lan ra các caller phía trên (unwind stack).
- Nếu không có ai bắt (`recover`), panic sẽ lên tới goroutine đang thực thi và kết thúc chương trình nếu nó là goroutine chính (in stack trace ra màn hình).
- `panic và recover` là 1 trong các cách thức các hàm/method truyền lỗi cho nhau để xử lý bên trong package phức tạp.

Tóm lại:
panic → lan ra các caller → không có caller nào recover → lan tới goroutine → Kết thúc goroutine bị panic đó (nếu là goroutine chính thì app crash)

---

4. Lưu ý khi dùng panic/recover

- Chỉ nên dùng panic/recover để xử lý lỗi nội bộ bên trong package, không nên để panic thoát ra ngoài package. Tức là phải kiểm soát đảm bảo: `PACKAGE HOẠT ĐỘNG HAY KHÔNG HOẠT ĐỘNG`, không để package nó làm sập nơi gọi package.
- Khi recover, `nên kiểm tra loại lỗi` để chỉ xử lý các panic dự kiến, các lỗi không mong muốn vẫn để chương trình crash để dễ phát hiện bug.
- Việc dùng panic/recover giúp đơn giản hóa xử lý lỗi phức tạp, nhưng cần dùng đúng chỗ để tránh làm khó bảo trì mã nguồn.

Tóm lại:
Go khuyến khích trả về lỗi qua giá trị trả về, chỉ dùng panic cho lỗi nghiêm trọng. recover giúp kiểm soát panic, tránh crash toàn bộ chương trình, nhưng nên dùng cẩn trọng và chủ yếu trong phạm vi nội bộ package.

Web server

Tóm tắt nội dung

Đoạn tài liệu này trình bày một ví dụ hoàn chỉnh về một web server viết bằng Go. Server này cung cấp giao diện web để người dùng nhập một đoạn text, sau đó tạo ra mã QR tương ứng bằng cách sử dụng dịch vụ của Google Chart API. Kết quả là một trang web hiển thị hình ảnh QR code và chính đoạn text đó.

Phân tích chi tiết đoạn code Go

Giải thích từng phần

1. **Import các package cần thiết**
    - `flag`: Xử lý tham số dòng lệnh.
    - `html/template`: Xử lý template HTML an toàn.
    - `log`: Ghi log lỗi.
    - `net/http`: Xây dựng web server.
2. **Khai báo biến cấu hình**
    - `addr`: Địa chỉ và cổng mà server sẽ lắng nghe, mặc định là `:1718`.

3. **Khai báo template**
    - `templ`: Được tạo từ chuỗi `templateStr` và parse thành template Go. Sử dụng `template.Must` để panic nếu có lỗi khi parse.
4. **Hàm `main`**
    - `flag.Parse()`: Đọc các tham số dòng lệnh.
    - `http.Handle("/", http.HandlerFunc(QR))`: Đăng ký handler cho đường dẫn `/`, khi có request sẽ gọi hàm `QR`.
    - `http.ListenAndServe(*addr, nil)`: Khởi động server, lắng nghe trên địa chỉ đã cấu hình. Nếu có lỗi sẽ ghi log và kết thúc chương trình.
5. **Hàm handler `QR`**
    - Nhận request, lấy giá trị tham số `s` từ form (query string).
    - Gọi `templ.Execute(w, req.FormValue("s"))` để render template với dữ liệu là giá trị `s`.
6. **Chuỗi template HTML**
    - Nếu có dữ liệu (`{{if .}}`), sẽ hiển thị hình ảnh QR code (dùng Google Chart API) và chính đoạn text đó.
    - Luôn hiển thị form nhập liệu để người dùng nhập text mới.

Ý nghĩa và điểm nổi bật

- **Ngắn gọn, dễ hiểu:** Chỉ với vài chục dòng code đã xây dựng được một web server có giao diện động.
- **Sử dụng template an toàn:** `html/template` tự động escape dữ liệu đầu vào, tránh lỗi XSS.
- **Tận dụng dịch vụ bên ngoài:** Không cần tự tạo QR code, chỉ cần nhúng link tới Google Chart API.
- **Cấu trúc chuẩn Go:** Sử dụng handler, flag, log, template đúng chuẩn idiomatic Go.

Kết luận

Đây là ví dụ điển hình cho thấy Go có thể xây dựng ứng dụng web nhỏ gọn, dễ bảo trì, tận dụng tốt các thư viện chuẩn. Đoạn code này minh họa cách tổ chức một web server, xử lý template, và tương tác với dịch vụ bên ngoài một cách an toàn và hiệu quả.