

Table of Contents

Giới thiệu	1.1
Chương 1: Nền tảng ngôn ngữ Go	1.2
1.1 Nguồn gốc của ngôn ngữ Go	1.2.1
1.2 Sự tiến hóa của chương trình "Hello World"	1.2.2
1.3 Array, strings và slices	1.2.3
1.4 Functions, Methods và Interfaces	1.2.4
1.5 Khái niệm xử lý đồng thời và song song	1.2.5
1.6 Mô hình thực thi đồng thời	1.2.6
1.7 Error và Exceptions	1.2.7
Chương 2: Lập trình CGO	1.3
2.1 Quick Start	1.3.1
2.2 CGO Foundation	1.3.2
2.3 Chuyển đổi kiểu	1.3.3
2.4 Lời gọi hàm	1.3.4
2.5 Cơ chế bên trong CGO	1.3.5
2.9 Thư viện tĩnh và động	1.3.6
2.10 Biên dịch và liên kết các tham số	1.3.7
2.11 Lời nói thêm	1.3.8
Chương 3: Remote Procedure Call	1.4
3.1 Giới thiệu về RPC	1.4.1
3.2 Protobuf	1.4.2
3.3 Làm quen với gRPC	1.4.3
3.4 Một số vấn đề khác của gRPC	1.4.4
3.5 gRPC và Protobuf extensions	1.4.5
3.6 Công cụ grpcurl	1.4.6
3.7 Lời nói thêm	1.4.7
Chương 4: Go và Web	1.5
4.1 Giới thiệu chương trình Web	1.5.1
4.2 Routing trong Web	1.5.2
4.3 Middleware	1.5.3
4.4 Kiểm tra tính hợp lệ của request	1.5.4
4.5 Làm việc với Database	1.5.5
4.6 Giới hạn lưu lượng Service	1.5.6
4.7 Mô hình của các dự án web	1.5.7
4.8 Lời nói thêm	1.5.8
Chương 5: Hệ thống phân tán	1.6
5.1 Distributed id generator	1.6.1

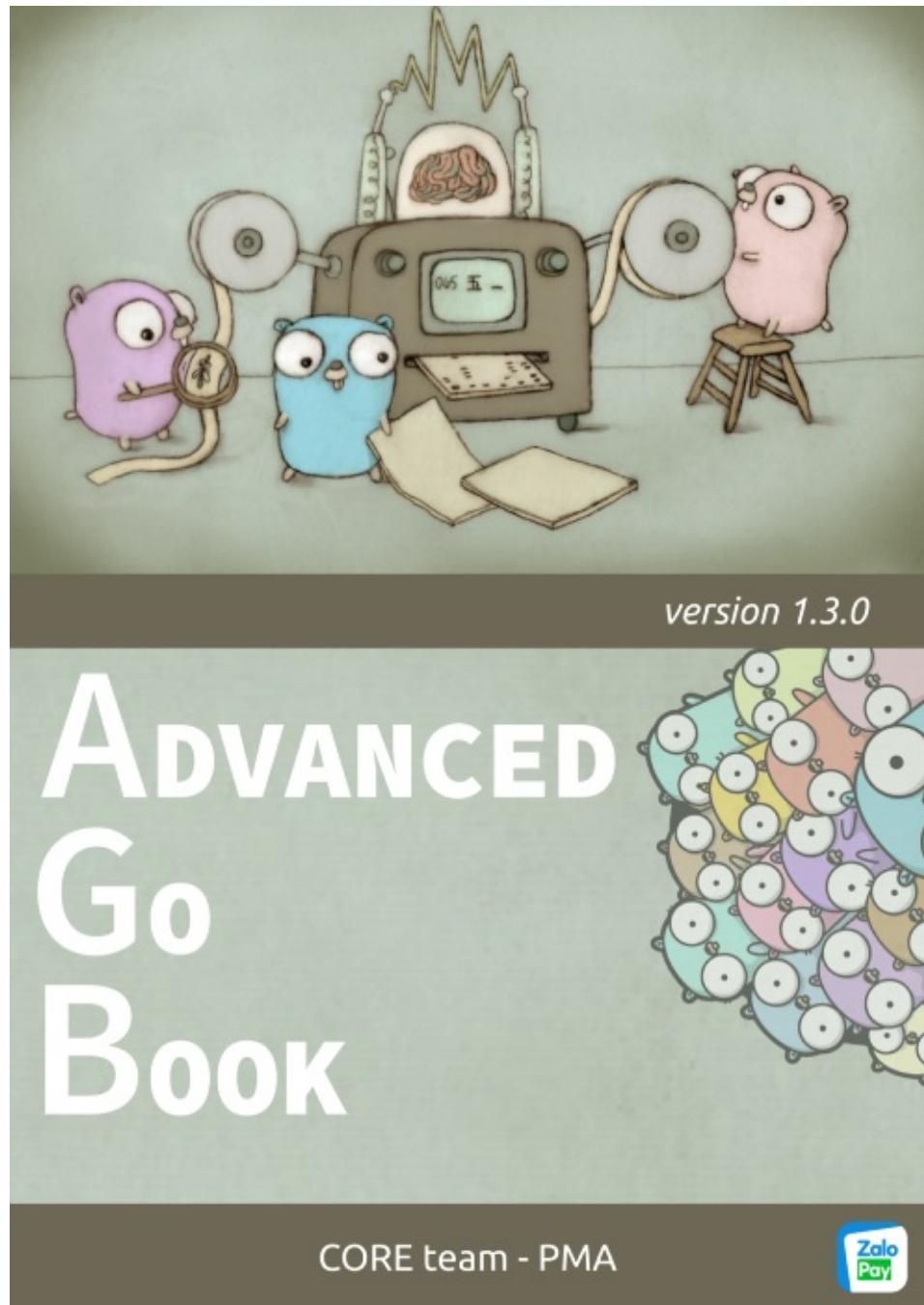
5.2 Lock phân tán	1.6.2
5.3 Hệ thống tác vụ có trì hoãn	1.6.3
5.4 Cân bằng tải	1.6.4
5.5 Quản lý cấu hình trong hệ thống phân tán	1.6.5
5.6 Trình thu thập thông tin phân tán	1.6.6
5.7 Lời nói thêm	1.6.7
Chương 6: Go Best Practices	1.7

Go Language Advanced Programming

repo status	active	version	1.3.0-release	contributors	5	last change	22/09/2019	open issues	1
-------------	--------	---------	---------------	--------------	---	-------------	------------	-------------	---

- Go Language Advanced Programming
 - Giới thiệu
 - Tại sao chúng tôi thực hiện bộ tài liệu này ?
 - Đối tượng sử dụng
 - Tài liệu tham khảo
 - Mục lục
 - Phương thức đọc
 - Tham gia phát triển
 - Nhóm phát triển
 - Cơ hội nghề nghiệp tại ZaloPay
 - Liên hệ

Giới thiệu



Ngôn ngữ [Golang](#) không còn quá xa lạ trong giới lập trình nữa. Đây là một ngôn ngữ dễ học, các bạn có thể tự học Golang cơ bản ở trang [Go by Example](#). Đa phần các tài liệu về Golang từ cơ bản hay đến nâng cao đều do các nhà lập trình viên nước ngoài biên soạn. Bộ tài liệu [Advanced Go Programming](#) được chúng tôi biên soạn hoàn toàn bằng Tiếng Việt sẽ trình bày về những chủ đề nâng cao trong Golang như CGO, RPC framework, Web framework, Distributed systems,... và kèm theo các ví dụ minh họa cụ thể theo từng chủ đề. Chúng tôi rất mong bộ tài liệu này sẽ giúp các bạn lập trình viên có thêm nhiều kiến thức mới và nâng cao kỹ năng lập trình Golang cho bản thân.

Tại sao chúng tôi thực hiện bộ tài liệu này ?

Chúng tôi thực hiện bộ tài liệu nhằm:

- Tạo ra bộ tài liệu về Go cho nội bộ ZaloPay sử dụng.

- Đây là cơ hội để mọi người biết tới technical stack của ZaloPay.
- Public ra bên ngoài để cộng đồng Golang Việt Nam có bộ tài liệu tiếng Việt do chính người Việt Nam biên soạn.
- Đồng thời tạo ra sân chơi mới có cơ hội giao lưu mở rộng mối quan hệ với các bạn có cùng đam mê lập trình.

Đối tượng sử dụng

Tất cả các bạn có đam mê lập trình Golang và đã nắm được cơ bản về lập trình Golang. Ngoài ra, trong bộ tài liệu này chúng tôi cũng có nhắc lại vài điểm cơ bản trong lập trình Golang.

Tài liệu tham khảo

Bộ tài liệu này được chúng tôi biên soạn dựa trên kinh nghiệm và kiến thức tích luỹ trong quá trình làm việc tại ZaloPay. Đồng thời chúng tôi có tham khảo các tài liệu bên ngoài như:

- [Advanced Go Programming](#).
- [Khoa học Distributed Systems](#) của Princeton.

Mục lục

Xem mục lục chính của bộ tài liệu [ở đây](#).

Phương thức đọc

- Đọc online: [GitBook](#)
- Tải file pdf: [pdf](#)
- Tải file epub: [epub](#)
- Tải file mobi: [mobi](#)

Tham gia phát triển

Chúng tôi biết tài liệu này còn nhiều hạn chế. Để trở nên hoàn chỉnh hơn trong tương lai, chúng tôi rất vui khi nhận được sự đóng góp từ mọi người.

Các bạn có thể đóng góp bằng cách:

- [Liên hệ](#) với chúng tôi.
- Trả lời các câu hỏi trong [issues](#).
- Tạo các issues gấp phai trên [issues](#).
- Tạo pull request trên repository của chúng tôi.
- ...

Nhóm phát triển

Dự án này được phát triển bởi các thành viên sau đây.

--	--	--	--	--



Cơ hội nghề nghiệp tại ZaloPay

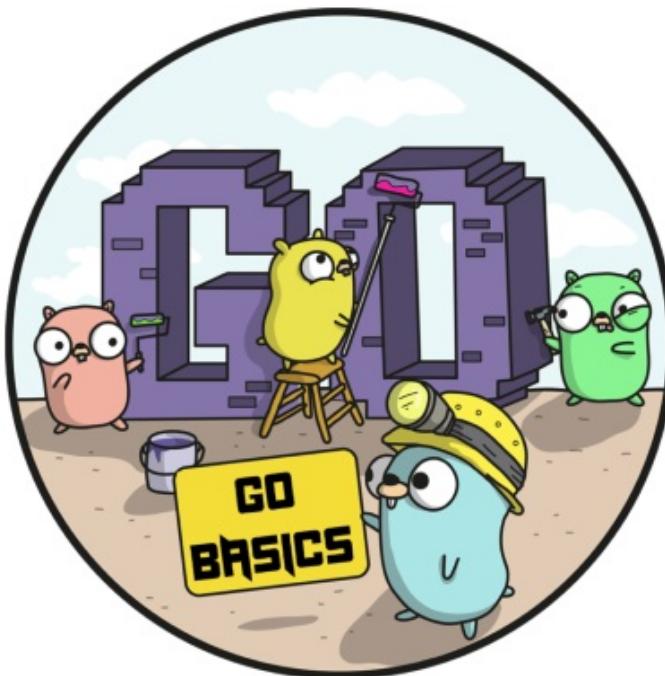


ZaloPay là một trong những ví điện tử được ưa chuộng hiện nay với nhiều tính năng và tiện ích hấp dẫn, giúp chúng ta giao dịch tài chính nhanh chóng hơn thông qua ứng dụng ZaloPay. Chúng tôi luôn mong muốn có thêm các thành viên mới gia nhập đội ngũ engineering, cùng giải quyết các bài toán hóc búa về high performance, fault tolerant và distributed transaction. Java, Golang và Rust là ngôn ngữ chính của chúng tôi.

Liên hệ

- Github: [ZaloPay Open Source](#)
- Facebook: [ZaloPay Engineering](#)
- Blog: [Medium ZaloPay Engineering](#)
- Bugs report: [issues](#)

Chương 1. Language Foundation



"Go is no Erlang, Smalltalk or Scheme, nothing pure. But it works great and is fun!" – Frank Mueller (@themue)"

Chương này bắt đầu bằng vài lời giới thiệu về lịch sử của ngôn ngữ Go và phân tích chi tiết cuộc cách mạng của chương trình "Hello World" với những thế hệ ngôn ngữ đi trước. Sau đó, một số cấu trúc dữ liệu sẽ được trình bày như arrays, strings, và slices, tính chất process-oriented và duck typing được thể hiện qua functions, methods, và interfaces, đặc biệt là mô hình concurrent programming và error handling cũng được giới thiệu sơ qua. Cuối cùng, một số trọng tâm trong việc phát triển chương trình trên các nền tảng macOS, Windows, và Linux, cũng như một vài editor và môi trường phát triển tích hợp (IDE) cũng được đề cập, bởi vì có công cụ tốt thì năng suất làm việc mới tăng lên.

Tài liệu này được là một trong những quyển sách nâng cao về Golang, vì vậy người đọc cần có một nền tảng Golang nhất định. Nếu bạn không biết nhiều về Golang, các bạn nên học Golang với một số gợi ý sau:

- Sau khi cài đặt Golang và tải hướng dẫn bằng `go get golang.org/x/tour`, có thể xem hướng dẫn [A Tour of Go](#) ngay ở local bằng lệnh `tour`.
- Bạn cũng có thể đọc hướng dẫn "[Ngôn ngữ lập trình Go](#)" được xuất bản bởi team Golang chính thức. "[Ngôn ngữ lập trình Go](#)" được gọi là *Kinh thánh Golang* trong cộng đồng Golang mà bạn phải đọc thật bài bản.

Trong khi học, hãy cố gắng giải quyết một số vấn đề nhỏ với Golang. Nếu bạn muốn tham khảo API, có thể mở truy vấn tài liệu tích hợp bằng lệnh `godoc`.

Liên kết

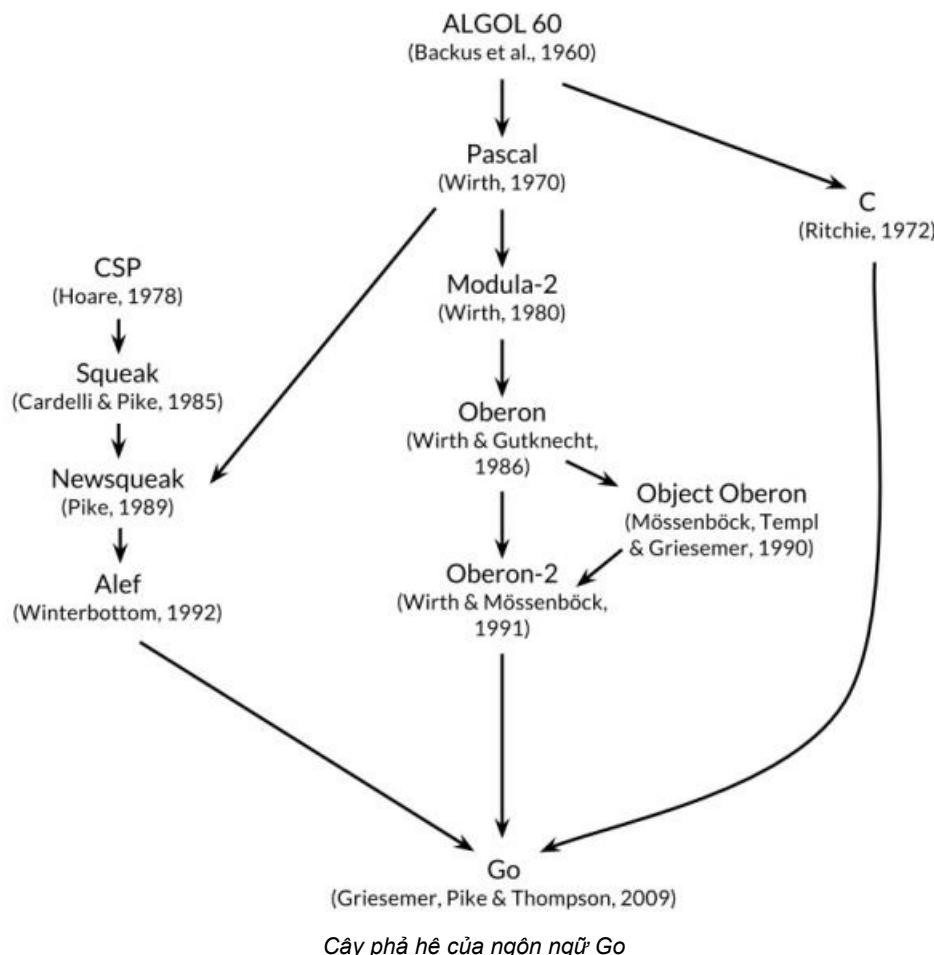
- Phần tiếp theo: [Nguồn gốc của ngôn ngữ Go](#)
- Phần trước: [Lời giới thiệu](#)
- [Mục lục](#)

1.1. Nguồn gốc của ngôn ngữ Go

Ngôn ngữ Go ban đầu được thiết kế và phát triển bởi một nhóm kỹ sư Google bao gồm **Robert Griesemer**, **Ken Thompson** và **Rob Pike** vào năm 2007. Mục đích của việc thiết kế ngôn ngữ mới bắt nguồn từ một số phản hồi về tính chất phức tạp của C++11 và nhu cầu thiết kế lại ngôn ngữ C trong môi trường network và multi-core.

Vào giữa năm 2008, hầu hết các tính năng được thiết kế trong ngôn ngữ được hoàn thành, họ bắt đầu hiện thực trình biên dịch (compiler) và Go runtime với **Russ Cox** là nhà phát triển chính. Trước năm 2010, ngôn ngữ Go dần dần được hoàn thiện. Vào tháng 9 cùng năm, ngôn ngữ Go chính thức được công bố dưới dạng Open source.

Ngôn ngữ Go thường được mô tả là "Ngôn ngữ tựa C" hoặc là "Ngôn ngữ C của thế kỷ 21". Từ nhiều khía cạnh, ngôn ngữ Go thừa hưởng những ý tưởng từ ngôn ngữ C, như là cú pháp, cấu trúc điều khiển, kiểu dữ liệu cơ bản, thủ tục gọi, trả về, con trỏ, v.v.., hoàn toàn kế thừa và phát triển ngôn ngữ C, hình bên dưới mô tả sự liên quan của ngôn ngữ Go với các ngôn ngữ khác.



Phía bên trái sơ đồ thể hiện tính chất **concurrency** của ngôn ngữ Go được phát triển từ học thuyết **CSP** công bố bởi **Tony Hoare** vào năm 1978. Học thuyết **CSP** dần dần được tinh chế và được ứng dụng thực tế trong một số ngôn ngữ lập trình như là **Squeak/NewSqueak** và **Alef**, cuối cùng là **Go**.

Chính giữa sơ đồ cho thấy tính chất hướng đối tượng và đóng gói của **Go** được kế thừa từ **Pascal** và những ngôn ngữ liên quan khác dẫn xuất từ chúng. Những từ khóa `package`, `import` đến từ ngôn ngữ Modula-2. Cú pháp hỗ trợ tính hướng đối tượng đến từ ngôn ngữ Oberon, ngôn ngữ Go được phát triển có thêm những tính chất đặc trưng như là `implicit interface` để chúng hỗ trợ mô hình **duck typing**.

Phía bên phải sơ đồ cho thấy ngôn ngữ **Go** kế thừa và cải tiến từ **C**, Cũng như **C**, **Go** là ngôn ngữ lập trình cấp thấp, nó cũng hỗ trợ con trỏ (pointer) nhưng ít nguy hiểm hơn **C**.

"Go is the result of C programmers designing a new programming language, and Rust is the result of C++ programmers designing a new programming language" - [link](#)

Một vài những tính năng khác của ngôn ngữ Go đến từ một số ngôn ngữ khác:

- Cú pháp `iota` được mượn từ ngôn ngữ **APL**.
- Những đặc điểm như là `lexical scope` và `nested functions` đến từ **Scheme**.
- Go hỗ trợ `slice` để truy cập phần tử nhanh và có thể tự động tăng giảm kích thước.
- Mệnh đề `defer` trong Go.

1.1.1. Khởi nguồn từ Bell Labs

Tính chất concurrency của Go đến từ học thuyết **Commutative sequential processes (CSP)** được công bố bởi Tony Hoare tại Bell Labs vào năm 1978. Bài báo khoa học về CSP nói rằng chương trình chỉ là một tập hợp các tiến trình được chạy song song, mà không có sự chia sẻ về trạng thái, sử dụng `channel` cho việc giao tiếp và điều khiển đồng bộ.

Học thuyết CSP của Tony Hoare chỉ là một mô hình lập trình với những khái niệm cơ bản về concurrency (tính đồng thời), nó cũng không hẳn là một ngôn ngữ lập trình. Qua việc thiết kế Go, Rob Pike đã tổng hợp nhiều thập kỷ trong việc ứng dụng học thuyết CSP trong việc xây dựng ngôn ngữ lập trình.

Ngôn ngữ **Erlang** là một hiện thực khác của học thuyết **CSP**, bạn có thể tìm kiếm thông tin về ngôn ngữ này trên [trang chủ Erlang](#).

Hình dưới chỉ ra lịch sử phát triển của ngôn ngữ Go qua codebase logs.

```
C:\go\go-tip>hg log -r 0:4
changeset: 0:f6182e5abf5e
user: Brian Kernighan <bwk>
date: Tue Jul 18 19:05:45 1972 -0500
summary: hello, world

changeset: 1:b66d0bf8da3e
user: Brian Kernighan <bwk>
date: Sun Jan 20 01:02:03 1974 -0400
summary: convert to C

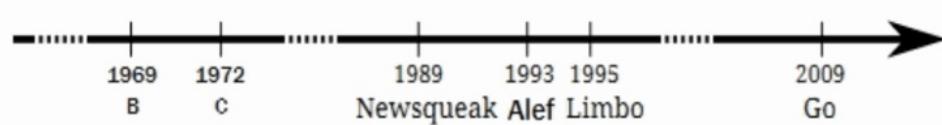
changeset: 2:ac3363d7e788
user: Brian Kernighan <research!bwk>
date: Fri Apr 01 02:02:04 1988 -0500
summary: convert to Draft-Proposed ANSI C

changeset: 3:172d32922e72
user: Brian Kernighan <bwk@research.att.com>
date: Fri Apr 01 02:03:04 1988 -0500
summary: last-minute fix: convert to ANSI C

changeset: 4:4e9a5b095532
user: Robert Griesemer <gri@golang.org>
date: Sun Mar 02 20:47:34 2008 -0800
summary: Go spec starting point.
```

Go language development log

Có thể nhìn thấy từ những submission log rằng ngôn ngữ Go được dần phát triển từ ngôn ngữ B - được phát minh bởi **Ken Thompson** và ngôn ngữ C được phát triển bởi **Dennis M.Ritchie**. Đó là thế hệ ngôn ngữ C đầu tiên, do đó nhiều người gọi Go là ngôn ngữ lập trình C của thế kỉ 21.



Lịch sử phát triển của lập trình concurrency trong Go

Trong vòng những năm gần đây, Go là một ngôn ngữ được ưa chuộng khi viết các chương trình Micro Services, vì những đặc tính nhỏ gọn, biên dịch nhanh, import thư viện từ github, cú pháp đơn giản nhưng hiện đại.

1.1.2. Hello World

Việc đầu tiên là cài đặt chương trình Go lang theo hướng dẫn trên trang chủ golang.org.

Để bắt đầu, chương trình đầu tiên thường in ra dòng chữ "Hello World", đoạn code bên dưới là chương trình này.

```
// package main chứa điểm thực thi đầu tiên của toàn chương trình
package main

// import gói thư viện "fmt" hỗ trợ in ra màn hình
import "fmt"

// main là hàm đầu tiên được chạy
func main() {

    // in ra dòng chữ "Hello World"
    fmt.Println("Hello World")
}
```

Lưu file trên thành `hello.go` và chạy bằng lệnh sau.

```
$ go run hello.go
Hello World
// hoặc có thể biên dịch ra file thực thi
$ go build
$ ./hello
Hello World
```

Liên kết

- Phần tiếp theo: [Sự tiến hóa của "Hello World"](#)
- Phần trước: [Chương 1](#)
- [Mục lục](#)

1.2. Sự tiến hóa của "Hello World"

Trong phần trước, chúng ta đã cùng tìm hiểu sơ lược về các ngôn ngữ cùng họ với Go và các ngôn ngữ lập trình khác được Bell Labs phát triển. Ở phần này, chúng ta sẽ nhìn lại dòng thời gian phát triển của từng ngôn ngữ và xem cách mà chương trình "Hello World" phát triển thành phiên bản của ngôn ngữ Go hiện tại và hoàn thiện những sự thay đổi mang tính cách mạng của nó.



1.2.1. Ngôn ngữ B - Ken Thompson, 1972

B là một ngôn ngữ lập trình đa dụng được phát triển bởi Ken Thompson thuộc Bell Labs, cha đẻ của ngôn ngữ Go, được thiết kế để hỗ trợ phát triển hệ thống UNIX. Tuy nhiên, B khá thiếu sự linh hoạt trong hệ thống kiểu khiến cho nó rất khó sử dụng.

Phiên bản "Hello World" sau đây nằm trong *A Tutorial Introduction to the Language B* được viết bởi Brian W. Kernighan (là người commit đầu tiên vào mã code của Go), chương trình như sau :

```
main() {
    extrn a, b, c;
    putchar(a); putchar(b); putchar(c);
    putchar('!*n');
}
a 'hell';
b 'o  w';
c 'orld';
```

Vì thiếu sự linh hoạt của kiểu dữ liệu trong B, các nội dung `a/b/c` cần in ra chỉ có thể được định nghĩa bằng các biến toàn cục, đồng thời chiều dài của mỗi biến phải được căn chỉnh (aligned) về 4 bytes (cảm giác giống như viết ngôn ngữ assembly vậy). Sau đó hàm `putchar` được gọi nhiều lần để làm nhiệm vụ output, lần gọi cuối với `!*n` để xuất ra một dòng mới.

Từ khi B được thay thế (bởi C), nó chỉ còn xuất hiện trong một số tài liệu và trở thành lịch sử.

1.2.2. Ngôn ngữ C - Dennis Ritchie, 1974 ~ 1989

C được phát triển bởi Dennis Ritchie trên nền tảng của B, trong đó thêm các kiểu dữ liệu phong phú hơn và đạt được mục tiêu lớn là viết lại UNIX. Có thể nói C chính là nền tảng phần mềm quan trọng nhất của ngành CNTT hiện đại. Hiện tại, gần như tất cả các hệ điều hành chính thống đều được phát triển bằng C, cũng như rất nhiều phần mềm cơ bản cũng được phát triển bằng C. Các ngôn ngữ lập trình của họ C đã thống trị trong nhiều thập kỷ và vẫn sẽ còn sức ảnh hưởng trong tương lai.

Trong hướng dẫn giới thiệu ngôn ngữ C được viết bởi Brian W. Kernighan vào khoảng năm 1974, phiên bản ngôn ngữ C đầu tiên của chương trình "Hello World" đã xuất hiện. Điều này cung cấp quy ước cho chương trình đầu tiên với "Hello World" cho hầu hết các hướng dẫn ngôn ngữ lập trình sau này.

```
// hàm không trả về kiểu giá trị một cách tường minh,
```

```
// mặc định sẽ trả về kiểu `int`
main()
{
    //`printf` không cần import khai báo hàm mà mặc định có thể được sử dụng
    printf("Hello World");

    // không cần một câu lệnh return nhưng mặc định sẽ trả về giá trị 0
}
```

Ví dụ này cũng xuất hiện trong bản đầu tiên của **C Programming Language** xuất bản năm 1978 bởi Brian W. Kerninghan và Dennis M. Ritchie (K&R).

Năm 1988, 10 năm sau khi giới thiệu hướng dẫn của K&R, phiên bản thứ 2 của **C Programming Language** cuối cùng cũng được xuất bản. Thời điểm này, việc chuẩn hóa ngôn ngữ ANSI C đã được hoàn thành sơ bộ, nhưng phiên bản chính thức của document vẫn chưa được công bố.

```
// thêm '#include <stdio.h>' là header file chứa câu lệnh đặc tả
// dùng để khai báo hàm `printf`
#include <stdio.h>

main()
{
    printf("Hello World\n");
}
```

Đến năm 1989, tiêu chuẩn quốc tế đầu tiên cho ANSI C được công bố, thường được nhắc tới với tên C89. C89 là tiêu chuẩn phổ biến nhất của ngôn ngữ C và vẫn còn được sử dụng rộng rãi. Phiên bản thứ 2 của **C Programming Language** cũng được in lại bản mới:

```
#include <stdio.h>
// 'void' được thêm vào danh sách các tham số hàm,
// chỉ ra rằng không có tham số đầu vào
main(void)
{
    printf("Hello World\n");
}
```

Tại thời điểm này, sự phát triển của ngôn ngữ C về cơ bản đã hoàn thành. C92/C99/C11 về sau chỉ hoàn thiện một số chi tiết trong ngôn ngữ. Do các yếu tố lịch sử khác nhau, C89 vẫn là tiêu chuẩn được sử dụng rộng rãi nhất.

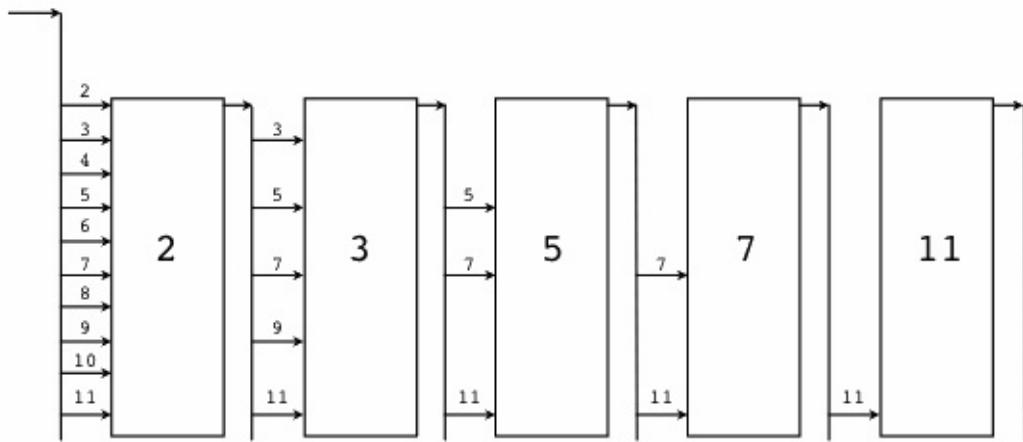
1.2.3. Newspeak - Rob Pike, 1989

Newspeak là thế hệ thứ 2 của ngôn ngữ chuột do Rob Pike sáng tạo ra, ông dùng nó để thực hành mô hình CSP lập trình song song. Newspeak nghĩa là ngôn ngữ squeak mới, với "squeak" là tiếng của con chuột, hoặc có thể xem là giống tiếng click của chuột. Ngôn ngữ lập trình squeak cung cấp các cơ chế xử lý sự kiện chuột và bàn phím. Phiên bản nâng cấp của Newspeak có cú pháp câu lệnh giống như của C và các biểu thức có cú pháp giống như Pascal. Newspeak là một ngôn ngữ chức năng (function language) thuần túy với bộ thu thập rác tự động cho các sự kiện bàn phím, chuột và cửa sổ.

Newspeak tương tự như một ngôn ngữ kịch bản có chức năng in tích hợp. Chương trình "Hello World" của nó không có gì đặc biệt:

```
// hàm 'print' có thể hỗ trợ nhiều tham số
print("Hello ", "World", "\n");
```

Bởi vì các tính năng liên quan đến ngôn ngữ Newsqueak và ngôn ngữ Go chủ yếu là đồng thời (concurrency) và pipeline nên ta sẽ xem xét các tính năng này thông qua phiên bản concurrency của thuật toán "sàng số nguyên tố". Nguyên tắc "sàng số nguyên tố" như sau:



Sàng số nguyên tố

Chương trình "sàng số nguyên tố" cho phiên bản concurrency của ngôn ngữ Newsqueak như sau:

```
// 'counter' dùng để xuất ra chuỗi gốc gồm các số tự nhiên vào các channel
counter := prog(c:chan of int) {
    i := 2;
    for(;;) {
        c <= i++;
    }
};

// Mỗi hàm 'filter' tương ứng với mỗi channel lọc số nguyên tố mới.
// Những channel lọc số nguyên tố này lọc các chuỗi input theo
// sàng số nguyên tố hiện tại và đưa kết quả tới channel đầu ra.
filter := prog(prime:int, listen, send:chan of int) {
    i:int;
    for(;;) {
        if((i = <-listen)%prime) {
            send <= i;
        }
    }
};

// Dòng đầu tiên của mỗi channel phải là số nguyên tố
// sau đó xây dựng sàng nguyên tố dựa trên số nguyên tố mới này
sieve := prog() of chan of int {
    // 'mk(chan of int)' tạo 1 channel, tương tự như 'make(chan int)' trong Go.
    c := mk(chan of int);

    begin counter(c);
    prime := mk(chan of int);
    begin prog(){
        p:int;
        newc:chan of int;
        for(;;){
            prime <= p =<- c;
            newc = mk();

            // 'begin filter(p,c,newc)' bắt đầu một hàm concurrency,
            // giống với câu lệnh 'go filter(p,c,newc)' trong Go.
            begin filter(p, c, newc);
            c = newc;
        }
    }
}
```

```

    }
}();

// 'become' dùng để trả về kết quả của hàm, tương tự như 'return'
become prime;
};

// kết quả là các số nguyên tố còn lại trên sàng
prime := sieve();

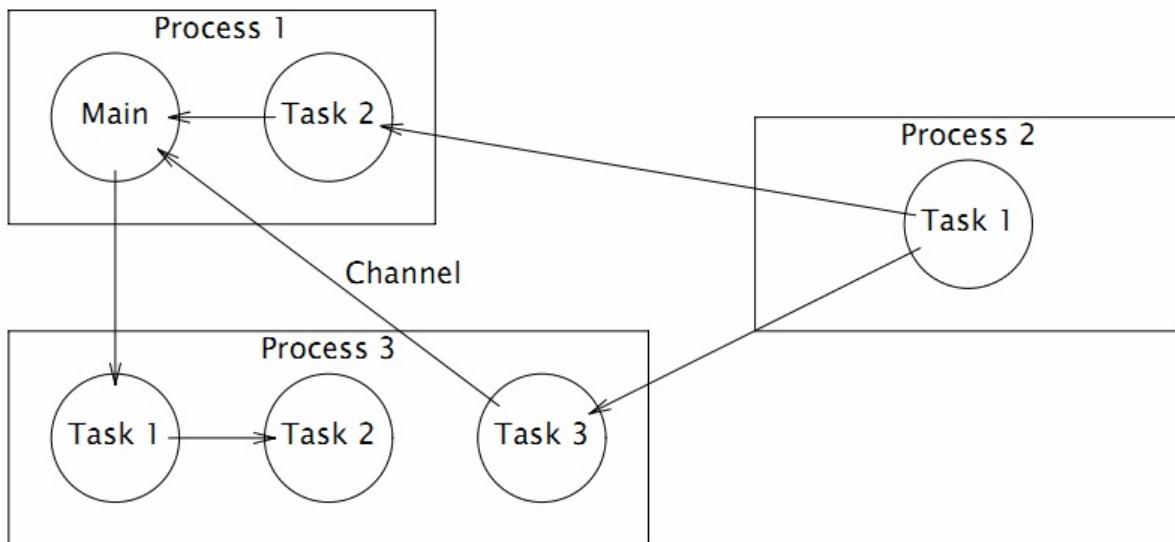
```

Cú pháp xử lý concurrency và channel trong ngôn ngữ Newspeak khá tương tự với Go, ngay cả cách khai báo kiểu dạng hậu tố của 2 ngôn ngữ này cũng giống nhau.

1.2.4. Alef - Phil Winterbottom, 1993

Trước khi xuất hiện ngôn ngữ Go, ngôn ngữ Alef có thể xem là ngôn ngữ xử lý concurrency hoàn hảo, hơn nữa cú pháp và runtime của Alef về cơ bản tương thích hoàn hảo với ngôn ngữ C. Tuy nhiên, do thiếu cơ chế phục hồi bộ nhớ tự động, việc quản lý tài nguyên bộ nhớ của cơ chế concurrency là vô cùng phức tạp. Hơn nữa, ngôn ngữ Alef chỉ cung cấp hỗ trợ ngắn hạn trong hệ thống Plan9 và các hệ điều hành khác không có môi trường phát triển Alef thực tế. Ngôn ngữ Alef chỉ có hai tài liệu công khai: **Alef Language Specification** và **the Alef Programming Wizard**. Do đó, không có nhiều thảo luận về ngôn ngữ Alef ngoài Bell Labs.

Hình sau đây là trạng thái concurrency của Alef:



Mô hình concurrency trong Alef

Chương trình "Hello World" cho phiên bản concurrency của ngôn ngữ Alef:

```

// Khai báo thư viện runtime chứa
// ngôn ngữ Alef
#include <alef.h>

void receive(chan(byte*) c) {
    byte *s;
    s = <- c;
    print("%s\n", s);
    terminate(nil);
}

void main(void) {
    chan(byte*) c;

    // tạo ra một channel chan(byte*)
}

```

```
// tương tự make(chan []byte) của Go
alloc c;

// receive khởi động hàm trong proc và thread
// tương ứng.
proc receive(c);
task receive(c);
c <- = "hello proc or task";
c <- = "hello proc or task";
print("done\n");

// kết thúc bằng lệnh terminate
terminate(nil);
}
```

Ngữ pháp của Alef về cơ bản giống như ngôn ngữ C. Nó có thể được coi là ngôn ngữ C ++ dựa trên ngữ pháp của ngôn ngữ C.

1.2.5. Limbo - Sean Dorward, Phil Winterbottom, Rob Pike, 1995

Limbo (Hell) là ngôn ngữ lập trình để phát triển các ứng dụng phân tán chạy trên máy tính nhỏ. Nó hỗ trợ lập trình mô-đun, kiểm tra kiểu mạnh vào thời gian biên dịch và thời gian chạy, liên lạc bên trong process thông qua channel, có bộ thu gom rác tự động. Có các loại dữ liệu trừu tượng đơn giản. Limbo được thiết kế để hoạt động an toàn ngay cả trên các thiết bị nhỏ mà không cần bảo vệ bộ nhớ phần cứng. Ngôn ngữ Limbo chạy chủ yếu trên hệ thống Inferno.

Phiên bản Limbo của chương trình "Hello World" như sau:

```
// tương tự 'package Hello' trong go
implement Hello;

// import các module khác
include "sys.m"; sys: Sys;
include "draw.m";

Hello: module
{
    // cung cấp hàm khởi tạo và kiểu khai báo dạng hậu tố
    // khác với Go không có tham số
    init: fn(ctx: ref Draw->Context, args: list of string);
};

init(ctx: ref Draw->Context, args: list of string)
{
    sys = load Sys Sys->PATH;
    sys->print("Hello World\n");
}
```

1.2.6. Ngôn ngữ Go - 2007 ~ 2009

Bell Labs sau khi trải qua nhiều biến động dẫn tới việc nhóm phát triển ban đầu của dự án Plan9 (bao gồm Ken Thompson) cuối cùng đã gia nhập Google. Sau khi phát minh ra ngôn ngữ tiền nhiệm là Limbo hơn 10 năm sau, vào cuối năm 2007, cảm thấy khó chịu với các tính năng "khủng khiếp" của C, ba tác giả gốc của ngôn ngữ Go đã tập hợp lại quyết định dùng 20% thời gian rảnh của mình để tạo ngôn ngữ một ngôn ngữ mới, chống lại sự thống trị của C/C++ ở Google lúc bấy giờ.

Đặc tả ngôn ngữ Go ban đầu được viết vào tháng 3 năm 2008 và chương trình Go gốc được biên dịch trực tiếp vào C và sau đó được dịch thành mã máy. Tháng 5 năm 2008, các nhà lãnh đạo Google cuối cùng đã phát hiện ra tiềm năng to lớn của ngôn ngữ Go và bắt đầu hỗ trợ cho dự án, cho phép các tác giả dành toàn bộ thời gian của mình để hoàn thiện ngôn ngữ. Sau khi phiên bản đầu tiên của đặc tả ngôn ngữ Go được hoàn thành, trình biên dịch ngôn ngữ Go cuối cùng có thể tạo ra mã máy trực tiếp (mà không phải thông qua C).

hello.go - Tháng 6 năm 2008

```
package main

func main() int {
    // vẫn còn dấu ';' cuối câu
    print "Hello World\n";

    // cần câu lệnh return để trả về giá trị
    // một cách tường minh
    return 0;
}
```

hello.go - 27 tháng 6 năm 2008

```
package main

func main() {
    print "Hello World\n";

    // loại bỏ câu lệnh return
    // chương trình trả về mặc định
    // bằng lệnh gọi 'exit(0)'
}
```

hello.go - 11 tháng 8 năm 2008

```
package main

func main() {
    // hàm built-in 'print' được đổi thành dạng hàm thông thường
    print("Hello World\n");
}
```

hello.go - 24 tháng 10 năm 2008

```
package main

import "fmt"

func main() {
    // 'printf' có thể định dạng chuỗi giống trong C
    // và được đặt trong package 'fmt' (viết tắt cho 'format')
    // phần đầu của tên hàm vẫn là chữ thường, lúc này tính năng export
    // vẫn chưa xuất hiện
    fmt.printf("Hello World\n");
}
```

hello.go - 15 tháng 1 năm 2009

```
package main

import "fmt"

func main() {
    // chữ 'P' viết hoa chỉ ra rằng hàm được export
    // các chữ viết thường chỉ ra hàm được dùng trong
    // nội bộ package
    fmt.Printf("Hello World\n");
}
```

hello.go - 11 tháng 12 năm 2009

```
package main

import "fmt"

func main() {
    // dấu ';' cuối cùng cũng được loại bỏ
    fmt.Printf("Hello World\n")
}
```

1.2.7. Hello World! - V2.0

Sau nửa thế kỷ phát triển, ngôn ngữ Go không chỉ có thể in được phiên bản Unicode của "Hello World", mà còn có thể cung cấp service tương tự cho người dùng trên toàn thế giới. Phiên bản sau đây in ra kí tự tiếng Việt "Xin chào" và thời gian hiện tại của mỗi client truy cập vào service.

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "time"
)

func main() {
    fmt.Println("Please visit http://127.0.0.1:12345/")

    // sử dụng giao thức http để in ra chuỗi bằng lệnh 'fmt.Fprintf'
    // thông qua log package
    http.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
        s := fmt.Sprintf("Xin chào - Thời gian hiện tại: %s", time.Now().String())
        fmt.Fprintf(w, "%v\n", s)
        log.Printf("%v\n", s)
    })

    // khởi động service http
    if err := http.ListenAndServe(":12345", nil); err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

Lúc này, Go cuối cùng đã hoàn thành việc chuyển đổi từ ngôn ngữ C của kỷ nguyên đơn lõi sang một ngôn ngữ lập trình đa dụng của môi trường đa lõi trong kỷ nguyên Internet thế kỷ 21.

Liên kết

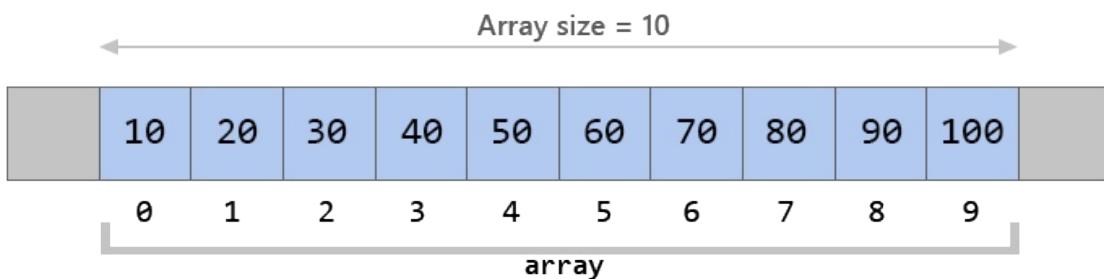
- Phản tiếp theo: [Array, strings và slices](#)
- Phản trước: [Nguồn gốc của ngôn ngữ Go](#)
- [Mục lục](#)

1.3. Array, strings và slices

`Arrays` và một số cấu trúc dữ liệu liên quan khác được sử dụng thường xuyên trong các ngôn ngữ lập trình. Chỉ khi chúng không đáp ứng được yêu cầu chúng ta mới cần nhắc sử dụng `linked lists` (danh sách liên kết) và `hash tables` (bảng băm) hoặc nhiều cấu trúc dữ liệu tự định nghĩa phức tạp khác.

`Arrays`, `strings` và `slices` trong ngôn ngữ Go là các cấu trúc dữ liệu liên quan mật thiết với nhau. Ba kiểu dữ liệu đó có cùng cấu trúc vùng nhớ lưu trữ bên dưới, và chỉ có những hành vi thể hiện ra bên ngoài khác nhau tùy thuộc vào ràng buộc ngữ nghĩa.

1.3.1. Array



Trong ngôn ngữ Go, `array` là một kiểu giá trị. Mặc dù những phần tử của array có thể được chỉnh sửa, phép gán của array hoặc khi truyền array như là một tham số của hàm thì chúng sẽ được xử lý toàn bộ, có thể hiểu là khi đó chúng được sao chép lại toàn bộ thành một bản sao mới xử lý trên bản sao đó (khác với kiểu truyền tham khảo).

Một array là một chuỗi độ dài cố định của các phần tử có kiểu dữ liệu nào đó, một array có thể bao gồm không hoặc nhiều phần tử. Độ dài của array là một phần thông tin được chứa trong nó, các array có độ dài khác nhau hoặc kiểu phần tử bên trong khác nhau được xem là các kiểu dữ liệu khác nhau, và không được phép gán cho nhau, vì thế array hiếm khi được sử dụng trong Go.

Cách định nghĩa một array:

```
// Định nghĩa một mảng kiểu int độ dài 3, các phần tử đều bằng 0
var a [3]int
// Định nghĩa một mảng có ba phần tử 1, 2, 3, do đó độ dài là 3
var b = [...]int{1, 2, 3}
// Mảng này có 3 phần tử theo thứ tự là 0, 2, 3
var c = [...]int{2: 3, 1: 2}
// Mảng này chứa dãy các phần tử là 1, 2, 0, 0, 5, 6
var d = [...]int{1, 2, 4: 5, 6}
```

Cấu trúc vùng nhớ của array thì rất đơn giản. Ví dụ cho một array `[4]int{2, 3, 5, 7}` thì cấu trúc bên dưới sẽ như sau:



Array layout

Khi biến array được gán hoặc truyền, thì toàn bộ array sẽ được sao chép. Nếu kích thước của array lớn, thì phép gán array sẽ chịu tổn phí lớn. Để tránh việc `overhead` (tổn phí) trong việc sao chép array, bạn có thể truyền con trỏ của array.

Lưu ý: con trỏ array thì không phải là một array.

```
// a là một array
var a = [...]int{1, 2, 3}
// b là một con trỏ tới array a
var b = &a
// in ra hai phần tử đầu tiên của array a
fmt.Println(a[0], a[1])
// truy xuất các phần tử của con trỏ array cũng giống như truy xuất các phần tử của array
fmt.Println(b[0], b[1])
// duyệt qua các phần tử trong con trỏ array, giống như duyệt qua array
for index, value := range b {
    // thay đổi từng phần tử trong b
    b[index] += 1
    fmt.Println(index, value)
}
// giá trị của các phần tử trong a bị thay đổi vì b
for index, value := range a {
    fmt.Println(index, value)
}
```

Kết quả là :

```
$ go run main.go
1 2
1 2
0 1
1 2
2 3
0 2
1 3
2 4
```

Hàm `len` có thể dùng để lấy thông tin về độ dài của array, và hàm `cap` sẽ tính toán độ dài tối đa của array. Nhưng trong kiểu array cả hai hàm này sẽ cùng trả về một giá trị giống nhau (điều này khác với slice).

Chúng ta có thể dùng vòng lặp `for` để duyệt qua các phần tử của array. Sau đây là những cách thường dùng để duyệt qua một array

```
for i := range a {
    fmt.Printf("a[%d]: %d\n", i, a[i])
}
for i, v := range b {
    fmt.Printf("b[%d]: %d\n", i, v)
}
for i := 0; i < len(c); i++ {
    fmt.Printf("c[%d]: %d\n", i, c[i])
}
```

`for range` là cách tốt nhất để duyệt qua các phần tử trong array, bởi vì cách này sẽ đảm các việc truy xuất sẽ không vượt quá giới hạn của array.

Các phần tử của array không nhất thiết là kiểu số học, nên cũng có thể là string, struct, function, interface, và channel, v.v..

```
// Mảng string
var s1 = [2]string{"hello", "world"}
```

```

var s2 = [...]string{"Hello!", "World"}
var s3 = [...]string{1: "Hello", 0: "World", }

// Mảng struct
var line1 [2]image.Point
var line2 = [...]image.Point{image.Point{X: 0, Y: 0}, image.Point{X: 1, Y: 1}}
var line3 = [...]image.Point{{0, 0}, {1, 1}]

// Mảng decoder của hình ảnh
var decoder1 [2]func(io.Reader) (image.Image, error)
var decoder2 = [...]func(io.Reader) (image.Image, error){
    png.Decode,
    jpeg.Decode,
}

// Mảng interface{}
var unknown1 [2]interface{}
var unknown2 = [...]interface{}{123, "Hello!"}

// Mảng channel
var chanList = [2]chan int{}

```

Chúng ta cũng có thể định nghĩa một array rỗng.

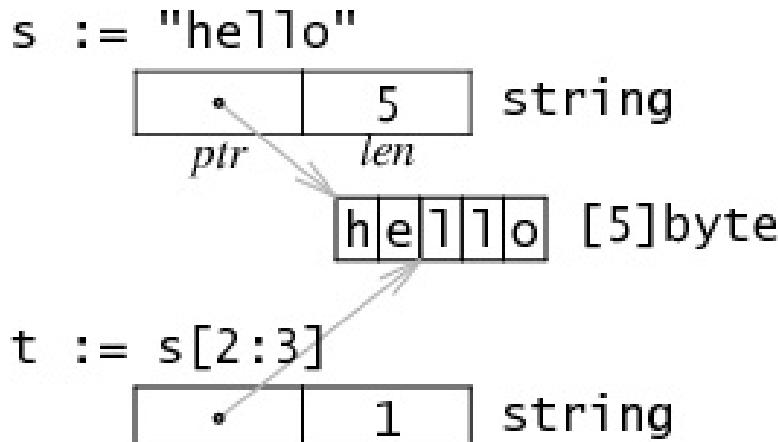
```

// Định nghĩa một array chiều dài 0
var d [0]int
// Tương tự trên
var e = [0]int{}
// Tương tự như trên
var f = [...]int{}

```

Một array có chiều dài 0 thì không chiếm không gian lưu trữ.

1.3.2. String



`string` cũng là một array của các `byte` dữ liệu, nhưng khác với array những phần tử của `string` là `immutable`.

Cấu trúc `reflect.StringHeader` được dùng để biểu diễn string :

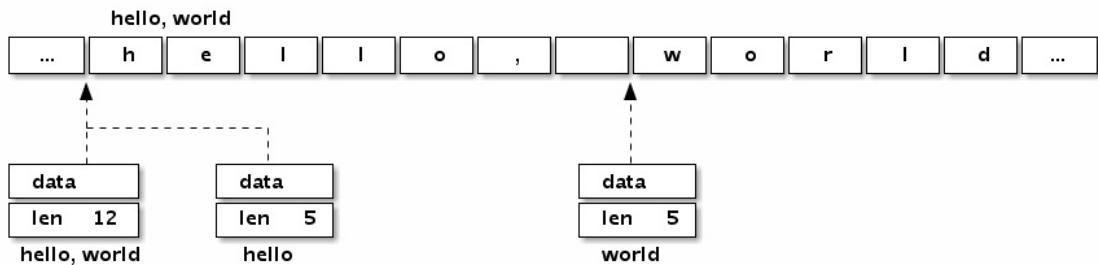
```

type StringHeader struct {
    // con trỏ địa chỉ vùng nhớ string
    Data uintptr
    // chiều dài của string
    Len int
}

```

Một string là một cấu trúc, do đó phép gán string thực chất là việc sao chép cấu trúc `reflect.StringHeader`, và không gây ra việc sao chép bên dưới phần dữ liệu.

Cấu trúc vùng nhớ tương ứng với string "Hello World" là:



String layout

Có thể thấy rằng bên dưới string "Hello World" là một array như sau:

```

var data = [...] byte {
    'H', 'e', 'l', 'l', 'o', ',', 'w', 'o', 'r', 'l', 'd',
}
    
```

Mặc dù string không phải là slice nhưng nó cũng hỗ trợ thao tác (slicing) cắt. Một vài phần của vùng nhớ cũng được truy cập bên dưới slice tại một số nơi khác nhau:

```

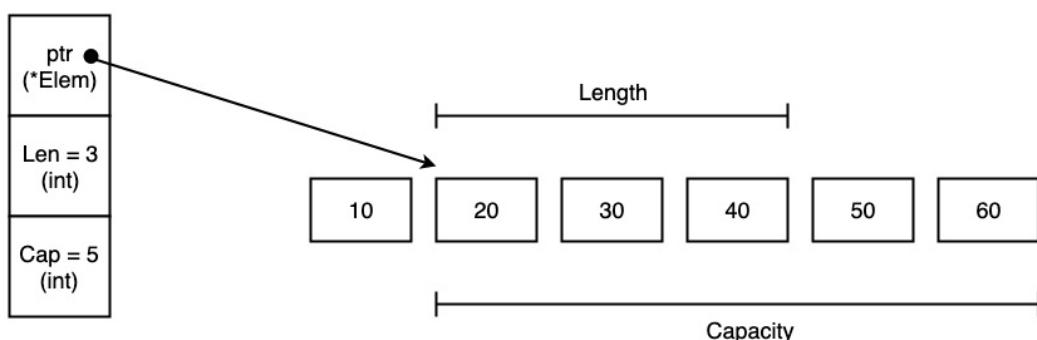
// s là biến string
var s = "hello world"
// lấy phần tử từ index 0 tới 4
hello := s[:5]
// lấy phần tử từ index 6 tới hết
world := s[6:]
// có thể thao tác trực tiếp
s1 := "hello world)[:5]
s2 := "hello world"[6:]
    
```

Tương tự như array, String cũng có một hàm built-in là `len` dùng để trả về chiều dài của string, ngoài ra bạn có thể dùng `reflect.StringHeader` để truy xuất chiều dài của string theo cách như sau

```

fmt.Println("len(s): ", (*reflect.StringHeader)(unsafe.Pointer(&s)).Len)
    
```

1.3.3. Slice



Cấu trúc Slice

`slices` thì phức tạp hơn, cấu trúc của chúng cũng như `string`, tuy nhiên việc giới hạn chỉ-đọc như `string` được lược bỏ.

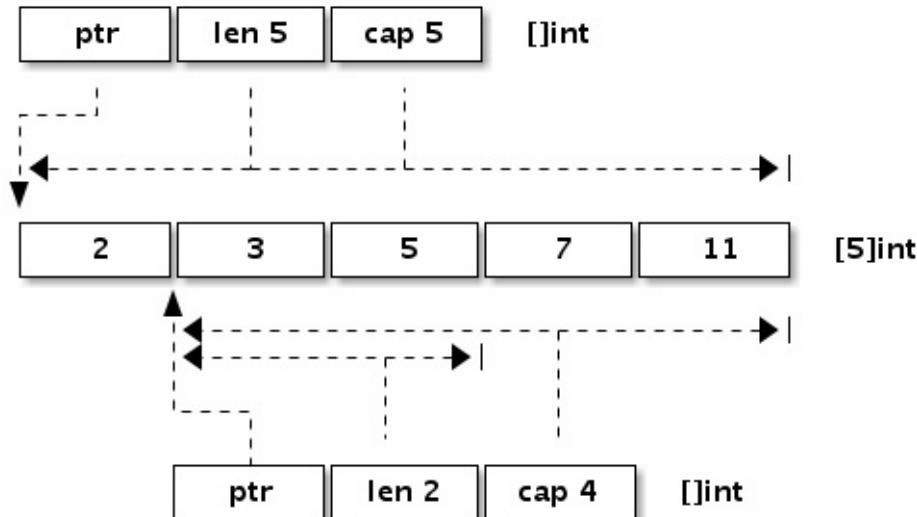
Cấu trúc của slice là `reflect.SliceHeader`

```
type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
```

Slice được xem là fat pointer, các bạn có thể đọc thêm ở bài viết sau để hiểu hơn về fat pointer trong Go. Cấu trúc slice bao gồm:

- `Data`: là con trỏ chứa địa chỉ của một array.
- `Len`: độ dài của slice.
- `Cap`: kích thước tối đa mà vùng nhớ trỏ tới slice được cấp phát.

Hình bên dưới sẽ miêu tả slice `x := []int{2,3,5,7,11}` và slice `y := x[1:3]`:



Slice layout

Các cách định nghĩa slice:

```
var (
    // nil slice
    a = []int
    // empty slice, khác với nil
    b = []int{}
    // có 3 phần tử trong slice, cả len và cap đều bằng 3
    c = []int{1,2,3}
    // có 2 phần tử trong slice, len bằng 2 và cap bằng 3
    d = c[:2]
    // có 2 phần tử trong slice, len bằng 2 và cap bằng 3
    e = c[0:2:cap(c)]
    // có 0 phần tử trong slice, len bằng 0 và cap bằng 3
    f = c[:0]
    // có 3 phần tử trong slice, len và cap bằng 3
    g = make ([]int,3)
    // có 2 phần tử trong slice, len bằng 2, cap bằng 3
    h = make ([]int,2,3)
```

```
// có 0 phần tử trong slice, len bằng 0, cap bằng 3
i = make ([]int, 0, 3)
)
```

Khi chúng ta sử dụng cú pháp tạo slice từ một slice cho trước như sau: **d = c[:2]** thì chúng ta nên lưu ý ở điểm sau. Slice sẽ không sao chép dữ liệu của slice gốc c qua d mà nó tạo ra một giá trị slice mới trả đến mảng ban đầu. Do đó, sửa đổi các phần tử của slice vừa được tạo sẽ sửa đổi các phần tử của slice gốc. Ví dụ minh họa:

```
old := []byte{'r', 'o', 'a', 'd'}
new := old[2:]
// new = []byte{'a', 'd'}
new[1] = 'm'
// new = []byte{'a', 'm'}
// old = []byte{'r', 'o', 'a', 'm'}
```

Các tác vụ cơ bản trong slice bao gồm:

- Duyệt qua các phần tử của slice
- Thêm phần tử vào slice
- Xóa phần tử trong slice

Duyệt qua slice

Duyệt qua slice thì tương tự như duyệt qua một arrays.

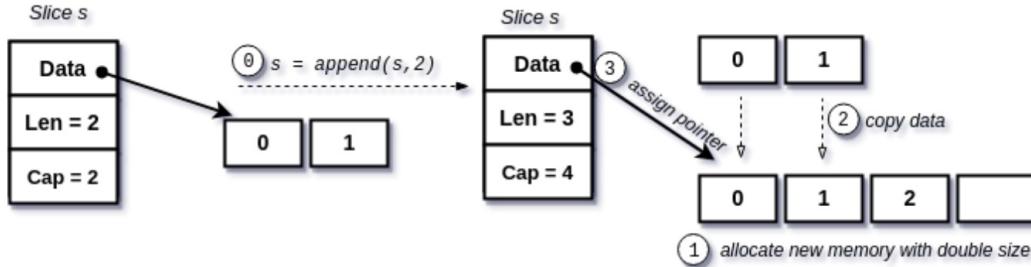
```
for i := range a {
    fmt.Printf("a[%d]: %d\n", i, a[i])
}
for i, v := range b {
    fmt.Printf("b[%d]: %d\n", i, v)
}
for i := 0; i < len(c); i++ {
    fmt.Printf("c[%d]: %d\n", i, c[i])
}
```

Thêm phần tử vào slice

Hàm `append` có thể thêm phần tử thứ `N` vào cuối cùng của slice:

```
var a []int
// nối thêm phần tử 1
a = append(a, 1)
// nối thêm phần tử 1, 2, 3
a = append(a, 1, 2, 3)
// nối thêm các phần tử 1, 2, 3 bằng cách truyền vào một mảng
a = append(a, []int{1, 2, 3}...)
```

Trong trường hợp slice ban đầu không đủ sức chứa khi thêm vào phần tử, hàm `append` sẽ hiện thực cấp phát lại vùng nhớ có kích thước gấp đôi vùng nhớ cũ và sao chép dữ liệu sang. Các bạn có thể xem đoạn mã nguồn về việc cấp phát lại vùng nhớ cho slice [ở đây](#).



Cấu trúc Slice

Ví dụ bên dưới cho thấy giá trị **cap** tăng gấp 2 khi thực thi hàm append vượt quá kích thước ban đầu.

```
func myAppend(sl []int, val int) []int{
    sl = append(sl, val)
    printSlice(sl)
    return sl
}

func printSlice(sl []int) {
    fmt.Printf("Slice %v\n", sl)
    fmt.Printf("Len %v, Cap %v\n\n", len(sl), cap(sl))
}

func main() {
    sl := make([]int, 1)
    printSlice(sl)
    for i := 1; i < 5; i++ {
        sl = myAppend(sl, i)
    }
}

/**
 * Slice [0]
 * Len 1, Cap 1
 *
 * Slice [0 1]
 * Len 2, Cap 2
 *
 * Slice [0 1 2]
 * Len 3, Cap 4
 *
 * Slice [0 1 2 3]
 * Len 4, Cap 4
 *
 * Slice [0 1 2 3 4]
 * Len 5, Cap 8
 */

```

Bên cạnh thêm phần tử vào cuối slice, chúng ta cũng có thể thêm phần tử vào đầu slice như sau

```
var a = []int{1, 2, 3}
// thêm phần tử 0 vào đầu slice a
a = append([]int{0}, a...)
// thêm các phần tử -3, -2, -1 vào đầu slice a
a = append([]int{-3, -2, -1}, a...)
```

Thêm phần tử vào đầu slice sẽ gây ra việc cấp phát lại vùng nhớ và làm những phần tử đang tồn tại trong slice sẽ được sao chép một lần nữa. Do đó, hiệu suất của việc thêm phần tử vào đầu slice sẽ thấp hơn thêm phần tử vào cuối slice.

Do hàm `append` sẽ trả về một slice mới, chúng ta có thể kết hợp nhiều hàm `append` để chèn một vài phần tử vào giữa slice.

```
// khai báo slice a
var a []int
// chèn x ở vị trí thứ i
a = append(
    a[:i],
    // tạo ra một slice tạm thời để nối với a[:i]
    append(
        []int{x}, a[i]...
    )...
)
// chèn một slice con vào slice ở vị trí thứ i
a = append(
    a[:i],
    append(
        []int{1, 2, 3},
        a[i]...
    )...
)
```

Bạn cũng có thể sử dụng hàm `copy` và `append` kết hợp với nhau để tránh việc khởi tạo những slice tạm thời như vậy:

```
// thêm phần tử 0 vào cuối slice a
a = append(a, 0)
// lùi những phần tử từ i trở về sau
copy(a[i+1:], a[i:])
// gán vị trí thứ i bằng x
a[i] = x
```

Chúng ta cũng có thể chèn nhiều phần tử vào vị trí chính giữa bằng việc kết hợp hàm `copy` và `append` như sau

```
// mở rộng không gian của slice a với array x
a = append(a, x...)
// sao chép len(x) phần tử lùi về sau
copy(a[i+len(x):], a[i:])
// sao chép array x vào giữa
copy(a[i:], x)
```

Xóa những phần tử trong slice

Có ba trường hợp xóa các phần tử:

- Ở đầu
- Ở giữa
- Ở cuối

Trong đó xóa phần tử ở cuối là nhanh nhất

```
a = []int{1, 2, 3}
// xóa một phần tử ở cuối
a = a[:len(a)-1]
// xóa N phần tử ở cuối
a = a[:len(a)-N]
```

Xóa phần tử ở đầu thì thực chất là di chuyển con trỏ dữ liệu về sau

```
a = []int{1, 2, 3}
// xóa phần tử đầu tiên
a = a[1:]
// xóa N phần tử đầu tiên
```

```
a = a[N:]
```

Khi xóa phần tử ở giữa, bạn cần dịch chuyển những phần tử ở phía sau lên trước, điều đó có thể được thực hiện như sau

```
a = []int{1, 2, 3, ...}
// xóa phần tử ở vị trí i
a = append(a[:i], a[i+1:]...)
// xóa N phần tử từ vị trí i
a = append(a[:i], a[i+N:]...)
// xóa phần tử ở vị trí i
a = a[:i+copy(a[i:], a[i+1:])]
// xóa N phần tử từ vị trí i
a = a[:i+copy(a[i:], a[i+N:])]
```

Kỹ thuật quản lý vùng nhớ trong slice

Hàm `TrimSpace` sau sẽ xóa đi các khoảng trắng. Hiện thực hàm này với độ phức tạp O(n) để đạt được sự hiệu quả và đơn giản.

```
func TrimSpace(s []byte) []byte {
    b := s[:0]
    // duyệt qua slice s để tìm phần tử thỏa điều kiện
    for _, x := range s {
        // kiểm tra điều kiện
        if x != ' ' {
            // tạo ra slice mới từ slice ban đầu thêm vào phần tử x
            b = append(b, x)
        }
    }
    return b
}
```

Thực tế, những giải thuật tương tự để xóa những phần tử trong slice thỏa một điều kiện nào đó, có thể được xử lý theo cách trên.

Lưu ý: hàm `append()` không cấp phát lại vùng nhớ khi chưa đạt tới sức chứa tối đa `cap`.

Hàm `Filter` sẽ lọc các phần tử thỏa điều kiện trong slice.

```
func Filter(s []byte, fn func(x byte) bool) []byte {
    b := s[:0]
    for _, x := range s {
        if !fn(x) {
            b = append(b, x)
        }
    }
    return b
}
```

Điểm chính của những tác vụ làm việc hiệu quả trên slice là hạn chế việc phải cấp lại vùng nhớ, cố gắng để hàm `append` sẽ không đạt tới `cap` sức chứa của slice, là giảm số lần cấp phát vùng nhớ và giảm kích thước vùng nhớ cấp phát tại mọi thời điểm.

Tránh gây ra memory leak trên slice

Những tác vụ trên slice sẽ không thay đổi vùng nhớ bên dưới slice, mà thực chất là thay đổi các tham số như `Data`, `Len`. Vùng nhớ bên dưới vẫn còn cho đến khi nào không còn được tham chiếu nữa.

Thỉnh thoảng toàn bộ dữ liệu bên dưới slice sẽ có thể mang một trạng thái đang được sử dụng vì chỉ có một phần nhỏ vùng nhớ được tham chiếu tới, nó sẽ trì hoãn quá trình tự động thu gom vùng nhớ để đòi lại vùng nhớ đã cấp phát.

Ví dụ là hàm `FindPhoneNumber` sẽ tải toàn bộ file vào bộ nhớ, và sau đó tìm kiếm số điện thoại đầu tiên xuất hiện trong file, và kết quả cuối cùng sẽ trả về một array.

```
func FindPhoneNumber(filename string) []byte {
    b, _ := ioutil.ReadFile(filename)
    return regexp.MustCompile("[0-9]+").Find(b)
}
```

Mã nguồn này sẽ trả về một mảng các `byte` trả tới toàn bộ file. Bởi vì slice tham khảo tới toàn bộ array gốc, cơ chế tự động thu gom rác không thể giải phóng không gian bên dưới array trong thời gian đó. Một yêu cầu kết quả nhỏ, những phải lưu trữ toàn bộ dữ liệu trong một thời gian dài. Mặc dù nó không phải là `memory leak` trong ngữ cảnh truyền thống, nó có thể làm chậm hiệu suất của toàn hệ thống.

Để khắc phục vấn đề này, bạn có thể sao chép dữ liệu cần thiết thành một slice mới.

```
func FindPhoneNumber(filename string) []byte {
    b, _ := ioutil.ReadFile(filename)
    b = regexp.MustCompile("[0-9]+").Find(b)
    return append([]byte{}, b...)
}
```

Vấn đề tương tự có thể gặp phải khi xóa những phần tử trong slice. Giả sử rằng con trả đổi tượng được lưu trữ trong cấu trúc của slice, sau khi xóa đi phần tử cuối, thì phần tử được xóa có thể còn được tham khảo bên dưới mảng slice, vùng nhớ có thể được giải phóng tự động trong thời gian đó (nó phụ thuộc vào cách hiện thực cơ chế thu hồi vùng nhớ)

```
var a []*int{ ... }
// phần tử cuối cùng dù được xóa nhưng vẫn được tham chiếu,
// do đó cơ chế thu gom rác tự động không thu hồi nó
a = a[:len(a)-1]
```

Cách giải quyết là đầu tiên thiết lập phần tử cần thu hồi về `nil` để đảm bảo giá trị thu gom tự động có thể tìm thấy chúng, sau đó xóa slices đó.

```
var a []*int{ ... }
// phần tử cuối cùng sẽ được gán giá trị nil
a[len(a)-1] = nil
// xóa phần tử cuối cùng ra khỏi slice
a = a[:len(a)-1]
```

Liên kết

- Phần tiếp theo: [Functions, Methods và Interfaces](#)
- Phần trước: [Sự tiến hóa của "Hello World"](#)
- [Mục lục](#)

1.4. Functions, Methods và Interfaces

Trong phần này chúng ta sẽ tìm hiểu cụ thể về các khái niệm cơ bản trong Golang: Function, Method và Interface.

1.4.1. Function

Hàm (function) là thành phần cơ bản của chương trình. Các hàm trong ngôn ngữ Go có thể có tên hoặc ẩn danh (anonymous function):

```
// hàm được đặt tên
func Add(a, b int) int {
    return a+b
}

// hàm ẩn danh
var Add = func(a, b int) int {
    return a+b
}
```

Một hàm trong ngôn ngữ Go có thể có nhiều tham số và nhiều giá trị trả về. Cả tham số và giá trị trả về trao đổi dữ liệu với hàm theo cách truyền vào giá trị (pass by value). Về mặt cú pháp, hàm cũng hỗ trợ số lượng tham số thay đổi, biến số lượng tham số phải là tham số cuối cùng và biến này phải là kiểu slice.

```
// Nhiều tham số và nhiều giá trị trả về
func Swap(a, b int) (int, int) {
    return b, a
}

// Biến số lượng tham số 'more'
// Tương ứng với kiểu [] int, là một slice
func Sum(a int, more ...int) int {
    for _, v := range more {
        a += v
    }
    return a
}
```

Khi đối số có thể thay đổi là một kiểu interface null, việc người gọi có phân giải (unpack) đối số đó hay không sẽ dẫn đến những kết quả khác nhau:

```
func main() {
    var a = []interface{}{123, "abc"}

    // tương đương với lời gọi trực tiếp `Print(123, "abc")`
    Print(a...) // 123 abc

    // tương đương với lời gọi `Print([]interface{}{123, "abc"})`
    Print(a)    // [123 abc]
}

func Print(a ...interface{}) {
    fmt.Println(a...)
}
```

Cả tham số truyền vào và các giá trị trả về đều có thể được đặt tên:

```
func Find(m map[int]int, key int) (value int, ok bool) {
    value, ok = m[key]
    return
}
```

Defer trong Function

Lệnh `defer` trì hoãn việc thực thi hàm cho tới khi hàm bao ngoài nó `return`. Các đối số trong lời gọi `defer` được đánh giá ngay lập tức nhưng lời gọi không được thực thi cho tới khi hàm bao ngoài nó `return`.

```
func main() {
    defer fmt.Println("world")

    fmt.Println("hello")
}
// kết quả: hello world
```

Mỗi lời gọi `defer` được push vào stack và thực thi theo thứ tự ngược lại khi hàm bao ngoài nó kết thúc.

Ta thường sử dụng `defer` cho việc đóng hoặc giải phóng tài nguyên:

- Đóng file giống như `try-finally`:

```
func main() {
    f, err := os.Create("file")
    if err != nil {
        panic("cannot create file")
    }

    // chắc chắn file sẽ được close dù hàm có bị panic hay return
    defer f.Close()
    fmt.Fprintf(f, "hello")
}
```

- Đóng file và xử lý panic giống như `try-catch-finally`:

```
func main() {
    defer func() {
        msg := recover()
        fmt.Println(msg)
    }()

    // . là folder hiện tại
    f, err := os.Create(".")
    if err != nil {
        panic("cannot create file")
    }
    defer f.Close()

    // không quan trọng chuyện gì xảy ra thì file cũng sẽ được close
    // để đơn giản nên ở đây bỏ qua bước kiểm ra close result
    fmt.Fprintf(f, "hello")
}
```

- Cũng giống như block `finally` thì lời gọi `defer` cũng có thể làm cho kết quả trả về thay đổi:

```
func yes() (text string) {
    defer func() {
        text = "no"
    }()
    return "yes"
```

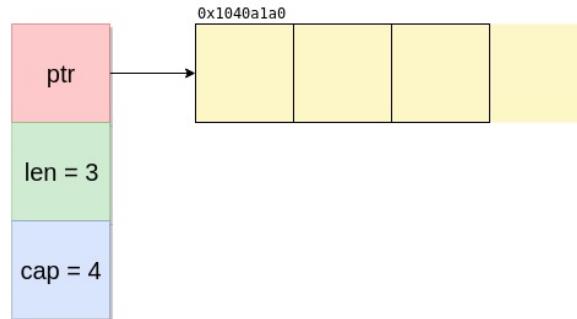
```

    }

func main() {
    fmt.Println(yes())
}

```

Slice trong Function



Mô hình slice

Mọi thứ trong Go đều được truyền theo kiểu pass by value, slice cũng thế. Nhưng vì giá trị của slice là một *header* (chứa con trỏ tới dữ liệu array bên dưới) nên khi truyền slice vào hàm, quá trình copy sẽ bao gồm luôn địa chỉ tới array chứa dữ liệu thực sự.

Ví dụ sau cho thấy ý nghĩa của việc truyền tham số kiểu slice vào hàm thay vì array:

```

// truyền vào array sẽ giúp
// nội dung của biến x không bị thay đổi
func once(x [3]int) {
    for i := range x {
        x[i] *= 2
    }
}

// truyền vào con trỏ ngầm định (slice)
// khiến nội dung của biến x bị thay đổi
func twice(x []int) {
    for i := range x {
        x[i] *= 2
    }
}

func main() {
    data := [3]int{8, 9, 0}

    once(data)
    fmt.Println(data)

    twice(data[0:])
    fmt.Println(data)

    // kết quả:
    // [8 9 0]
    // [16 18 0]
}

```

Tham số trả về được đặt tên

Cũng như tham số nhận vào, giá trị trả về cũng có thể được đặt tên, nhờ đó có thể đơn giản hóa lệnh return:

```

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:]
}

// hàm trả về n mà không cần phải chỉ rõ
return
}

```

1.4.2. Method

Go không có class, tuy nhiên chúng ta có thể định nghĩa các phương thức (Method) cho *type* (kiểu).

Phương thức là một hàm với đối số (argument) đặc biệt gọi là *receiver*.

```

type Vertex struct {
    X, Y float64
}

// method Abs() với receiver 'v'
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := Vertex{3, 4}
    fmt.Println(v.Abs())

    // kết quả:
    // 5
}

```

Phương thức (Method) là một tính năng của lập trình hướng đối tượng (OOP). Trong ngôn ngữ C++, phương thức tương ứng với một hàm thành viên của một class, liên kết với một đối tượng cụ thể. Tuy nhiên, phương thức trong ngôn ngữ Go được liên kết với kiểu, do đó liên kết tĩnh của phương thức có thể được tạo thành trong giai đoạn biên dịch.

Một chương trình hướng đối tượng sử dụng các phương thức để thể hiện những thao tác trên thuộc tính (properties) của nó, qua đó người dùng có thể sử dụng đối tượng mà không cần phải thao tác trực tiếp với đối tượng mà là thông qua các phương thức. C++ thường được xem là một dấu mốc mà lập trình hướng đối tượng bắt đầu phát triển mạnh mẽ, nó hỗ trợ các tính năng hướng đối tượng (như class) dựa trên cơ sở ngôn ngữ C. Kế đến là Java, ngôn ngữ được gọi là hướng đối tượng thuần túy vì các hàm của nó không thể tồn tại độc lập mà phải thuộc về một class nhất định.

Đối với một kiểu nhất định, tên của mỗi phương thức phải là duy nhất và các phương thức cũng như hàm đều không hỗ trợ overload.

Dưới đây là hiện thực các phương thức làm việc với File theo kiểu ngôn ngữ C:

```

type File struct {
    fd int
}

// mở file
func OpenFile(name string) (*File, error) {
    fmt.Println("Opening file ", name)
    return nil, nil
}

```

```
// đóng file
func (f *File) Close() error {
    fmt.Println("Close file")
    return nil
}

// đọc dữ liệu từ file
func (f *File) Read(offset int64, data []byte) int {
    fmt.Println("Read file")
    return 0
}
```

Ta sử dụng các phương thức này như sau:

```
func main() {
    var data []byte

    // khởi tạo một đối tượng File
    f, _ := OpenFile("foo.dat")

    f.Read(0, data)
    f.Close()
}
```

Trong một số tình huống, ta quan tâm nhiều hơn đến một chuỗi thao tác ví dụ như `Read` đọc một số mảng và sau đó gọi `Close` để đóng, trong ngữ cảnh này, người dùng không quan tâm đến kiểu của đối tượng, miễn là nó có thể đáp ứng được các thao tác của `Read` và `close`. Tuy nhiên trong các biểu thức phương thức của `ReadFile`, `CloseFile` có chỉ rõ kiểu `File` trong tham số kiểu sẽ khiến chúng không bị phụ thuộc vào đối tượng nào cụ thể. Việc này có thể khắc phục bằng cách sử dụng thuộc tính closure (closure property):

```
func main() {
    var data []byte

    // khởi tạo một đối tượng File
    f, _ := OpenFile("foo.dat")

    // một hàm closure có thể gọi tới đối tượng f ngoài hàm
    // sẽ liên kết với đối tượng f
    var Close = func() error {
        return (*File).Close(f)
    }

    // tương tự với hàm Close
    var Read = func (offset int64, data []byte) int {
        return (*File).Read(f, offset, data)
    }

    // xử lý file
    Read(0, data)
    Close()
}
```

Chúng ta có thể đơn giản hóa thành như sau:

```
func main() {
    var data []byte

    // mở đối tượng file
    f, _ := OpenFile("foo.dat")

    // ràng buộc với đối tượng f
    var Close = f.Close
```

```
// ràng buộc với đối tượng f
var Read = f.Read

// khi gọi không cần chỉ rõ đối tượng nữa
// vì đã được ràng buộc trước đó
Read(0, data)
Close()
}
```

Kế thừa phương thức

Go không hỗ trợ tính năng kế thừa như các ngôn ngữ hướng đối tượng truyền thống mà có cách của riêng mình. Tính kế thừa đạt được bằng cách xây dựng các thuộc tính ẩn danh trong struct:

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

type ColoredPoint struct {
    // thuộc tính ẩn danh
    Point

    // thuộc tính bình thường
    Color color.RGBA
}
```

Chúng ta có thể định nghĩa `ColoredPoint` như một struct có 3 trường, nhưng ở đây chúng ta sẽ dùng struct `Point` chứa `x` và `y` để thay thế.

```
// khai báo một đối tượng thuộc struct
var cp ColoredPoint

// có thể gán thẳng vào thuộc tính X
// không cần phải thông qua Point
cp.X = 1

// có thể truy cập X bằng cách này
fmt.Println(cp.Point.X)
// "1"

// hoặc gán vào Y thông qua Point
cp.Point.Y = 2

// và truy cập Y bằng cách này
fmt.Println(cp.Y)
// "2"
```

Có thể đạt được kết quả tương tự ngay cả với phương thức.

```
// lấy ví dụ với struct Mutex có sẵn
type Mutex struct {}
func (m *Mutex) Lock()
func (m *Mutex) Unlock()

// struct Cache kế thừa Mutex bằng cách
// khai báo một thuộc tính ẩn danh là sync.Mutex
type Cache struct {
    m map[string]string
    sync.Mutex
}

// Lookup tìm trên Cache với dữ liệu key và trả về value tương ứng
func (p *Cache) Lookup(key string) string {
```

```
// p có thể gọi thẳng tới phương thức Lock và Unlock
// nhờ kế thừa từ sync.Mutex
p.Lock()
defer p.Unlock()

return p.m[key]
}
```

Khả năng liên kết trực tiếp tới kiểu được kế thừa này được hoàn thành lúc biên dịch và không mất chi phí runtime.

Ví dụ trên có thể làm ta nghĩ rằng `sync.Mutex` là một lớp cơ sở và `Cache` là lớp kế thừa hoặc lớp con của nó. Tuy nhiên, phương thức được kế thừa theo cách này không thể hiện được tính đa hình bởi vì cái mà đối tượng `p` gọi tới là phương thức gốc mà không phải của nó (của struct Cache).

Nếu cần tính chất đa hình như các ngôn ngữ OOP khác, chúng ta cần triển khai nó với Interface.

1.4.3. Interface

Các interface trong Go cung cấp một cách để xác định hành vi của một đối tượng: nếu đối tượng đó có thể làm những việc *nhus thế này*, thì nó có thể được sử dụng ở đây.

Ngôn ngữ Go hiện thực mô hình hướng đối tượng thông qua cơ chế Interface.

Rob Pike, cha đẻ của ngôn ngữ Go, đã từng nói một câu nổi tiếng:

Languages that try to disallow idiocy become themselves idiotic

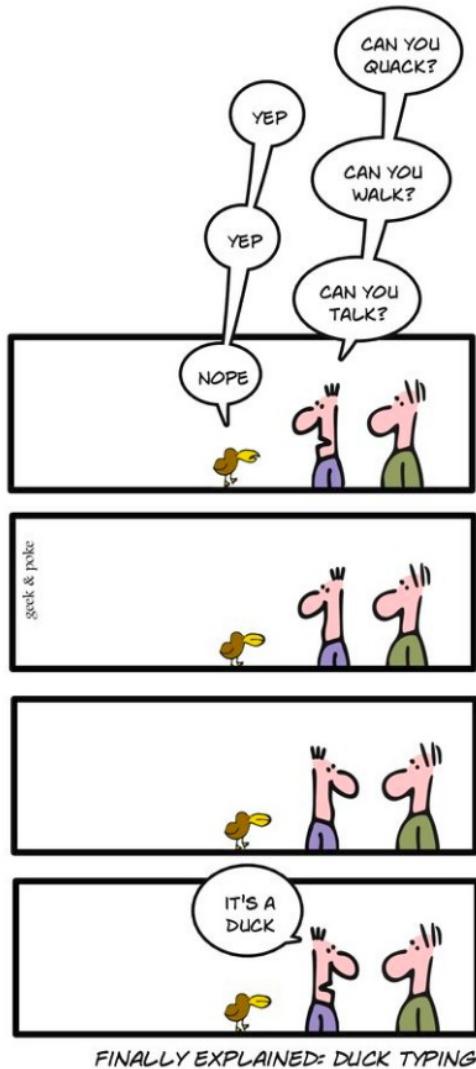
Các ngôn ngữ lập trình tĩnh nói chung có các hệ thống kiểu nghiêm ngặt, cho phép trình biên dịch đi sâu vào xem liệu lập trình viên có thực hiện bất kỳ động thái bất thường nào không. Tuy nhiên, một hệ thống kiểu quá nghiêm ngặt có thể làm cho việc lập trình trở nên quá cồng kềnh và khiến chúng ta phải mất nhiều thời gian cho nó.

Ngôn ngữ Go vì thế cố gắng cung cấp sự cân bằng giữa sự linh hoạt và tính an toàn: có cơ chế `duck-typing` thông qua interface nhưng đồng thời cũng kiểm tra kiểu nghiêm ngặt.

Duck typing

Duck-typing với ý tưởng đơn giản:

If something looks like a duck, swims like a duck and quacks like a duck then it's probably a duck.



Ví dụ có một interface con vịt, xác định khả năng Quacks :

```
type Duck interface {
    Quacks()
}
```

Và cách ta áp dụng *duck-typing*:

```
// một struct động vật bất kì
type Animal struct {}

// con này có khả năng `Quacks` như vịt
func (a Animal) Quacks() {
    fmt.Println("The animal quacks");
}

// hàm dành cho vịt
func Scream(duck Duck) {
    duck.Quacks()
}

func main() {
    // a là một vật thuộc struct Animal
    a := Animal{}
```

```
// vì a có khả năng `Quacks` như vịt nên
// ta có thể sử dụng nó như một con vịt trong hàm này
Scream(a)
}
```

Thiết kế này cho phép chúng ta tạo ra một interface mới thỏa mãn kiểu hiện có mà không phải hủy đi định nghĩa ban đầu của chúng, điều này đặc biệt linh hoạt và hữu ích khi các kiểu mà ta sử dụng đến từ những package không thuộc quyền kiểm soát của mình.

Chuyển đổi kiểu trong Go

Trong Golang, chuyển đổi kiểu ngầm định không được hỗ trợ với các kiểu cơ bản (kiểu không có interface): không thể gán giá trị của một biến kiểu `int` trực tiếp cho một biến kiểu `int64`.

Các yêu cầu về tính nhất quán của ngôn ngữ Go đối với kiểu cơ bản nghiêm ngặt là thế, nhưng nó lại khá linh hoạt để chuyển đổi kiểu giữa các interface: Chuyển đổi giữa đối tượng - interface hoặc chuyển đổi giữa interface - interface đều có thể là chuyển đổi ngầm định. Bạn có thể xem ví dụ sau:

```
var (
    // chuyển đổi ngầm định khi *os.File thỏa interface io.ReadCloser
    a io.ReadCloser = (*os.File)(f)

    // chuyển đổi ngầm định khi io.ReadCloser thỏa interface io.Reader
    b io.Reader = a

    // chuyển đổi ngầm định khi io.ReadCloser thỏa interface io.Closer
    c io.Closer = a

    // chuyển đổi tường minh khi io.Closer thỏa interface io.Reader
    d io.Reader = c.(io.Reader)
)
```

Một số sai lầm khi sử dụng Interface

Đôi khi đối tượng và interface quá linh hoạt dẫn đến việc chúng ta có thể mắc sai lầm khi struct khác vô tình điều chỉnh interface. Để khắc phục ta định nghĩa một phương thức đặc biệt để phân biệt các interface:

```
type runtime.Error interface {
    error

    // RuntimeError là một hàm rỗng được dùng chỉ với mục đích là
    // phân biệt lỗi runtime với các lỗi khác nhờ tính chất:
    // một type là runtime error chỉ khi nào nó có method RuntimeError
    RuntimeError()
}
```

Trong protobuf, interface `Message` cũng áp dụng một phương thức tương tự: định nghĩa một phương thức duy nhất `ProtoMessage` để ngăn các kiểu dữ liệu khác vô tình thỏa mãn interface:

```
type proto.Message interface {
    Reset()
    String() string
    ProtoMessage()
}
```

Interface `proto.Message` rất dễ bị "giả mạo", để tránh điều đó ta nên định nghĩa một phương thức riêng cho nó. Chỉ các đối tượng thỏa mãn phương thức riêng này mới có thể thỏa mãn interface đó và tên của phương thức riêng chứa tên đường dẫn tuyệt đối của package, vì vậy phương thức này chỉ có thể được hiện thực bên trong package để đáp ứng interface. `testing.TB` là interface trong package `test` sử dụng kỹ thuật này:

```
type testing.TB interface {
    Error(args ...interface{})
    Errorf(format string, args ...interface{})
    ...

    // Phương thức private ngăn user khác implement interface
    private()
}
```

Khả năng bị "làm giả" phương thức thuộc interface

Như đã đề cập trong phần Method, ta có thể kế thừa các phương thức của kiểu `án danh` bằng cách thêm các thuộc tính `án danh` thuộc kiểu đó vào struct. Vậy điều gì xảy ra nếu thuộc tính `án danh` này không phải là một kiểu bình thường mà là một kiểu interface?

Chúng ta có thể làm giả phương thức `private` của `testing.TB` bằng cách nhúng vào struct `TB` interface `án danh`:

```
package main

import (
    "fmt"
    "testing"
)

// TB có thể kế thừa phương thức `private` từ interface `testing.TB`
type TB struct {
    testing.TB
}

// phương thức thuộc struct TB
func (p *TB) Fatal(args ...interface{}) {
    fmt.Println("TB.Fatal disabled!")
}

func main() {
    // khởi tạo một đối tượng thuộc interface testing.TB
    var tb testing.TB = new(TB)

    // lúc này nó có thể sử dụng phương thức Fatal mà TB đã hiện thực
    tb.Fatal("Hello, playground")
}
```

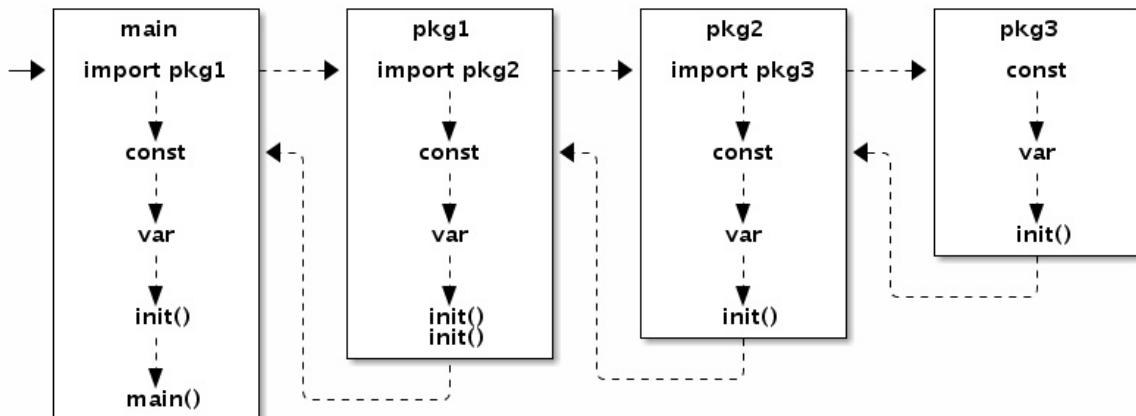
1.4.4. Luồng thực thi của một chương trình Go

Việc khởi tạo và thực thi chương trình Go luôn bắt đầu từ hàm `main.main`. Nếu package `main` có import các package khác, chúng sẽ được thêm vào package `main` theo thứ tự khai báo.

`init` không phải là hàm thông thường, nó có thể có nhiều định nghĩa, và các hàm khác không thể sử dụng nó.

- Nếu một package được import nhiều lần, sẽ chỉ được tính là một khi thực thi.
- Khi một package được import mà nó lại import các package khác, trước tiên Go sẽ import các package khác đó trước, sau đó khởi tạo các hằng và biến của package, rồi gọi hàm `init` trong từng package.
- Nếu một package có nhiều hàm `init` và thứ tự gọi không được xác định cụ thể thì chúng sẽ được gọi theo thứ tự xuất hiện. Cuối cùng, khi `main` đã có đủ tất cả hằng và biến ở package-level thì nó sẽ được khởi tạo bằng

cách thực thi hàm `init`, tiếp theo chương trình đi vào hàm `main.main` và bắt đầu thực thi. Hình dưới đây là sơ đồ nguyên lý một chuỗi bắt đầu của chương trình hàm trong Go:



Tiến trình khởi tạo package

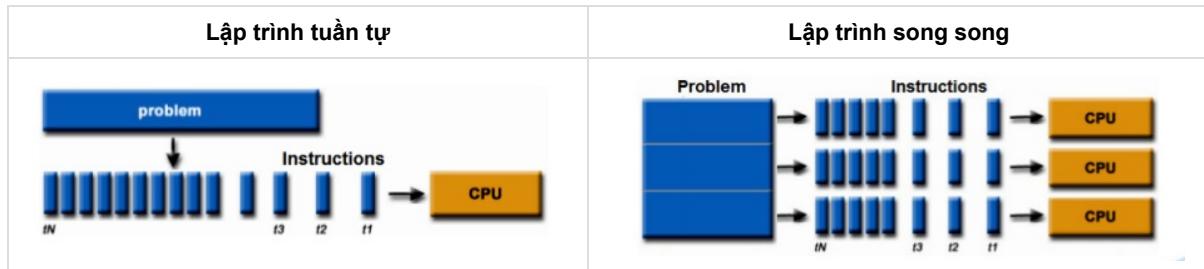
Cần lưu ý rằng trước khi hàm nào khác được thực thi thì tất cả code đều chạy trong cùng một Goroutine `main.main`, đây là thread chính của chương trình. Do đó, nếu một Goroutine khởi chạy trong hàm `main.main` thì nó chỉ có thể được thực thi sau khi vào chương trình đã thực thi xong `init`.

Liên kết

- Phần tiếp theo: [Mô hình lập trình đồng thời và lập trình song song](#)
- Phần trước: [Array, strings và slices](#)
- [Mục lục](#)

1.5. Mô hình lập trình đồng thời và lập trình song song

Thời gian đầu, CPU chỉ có một nhân duy nhất, các ngôn ngữ khi đó sẽ theo mô hình lập trình tuần tự, điển hình là ngôn ngữ C. Ngày nay, với sự phát triển của công nghệ đa xử lý, để tận dụng tối đa sức mạnh của CPU, mô hình lập trình song song hay [multi-threading](#)) thường thấy trên các ngôn ngữ lập trình ra đời. Ngôn ngữ Go cũng phát triển mô hình lập trình đồng thời rất hiệu quả với khái niệm Goroutines.



Ở phần này chúng ta sẽ đi tìm hiểu về mô hình lập trình đồng thời trong Go như thế nào. Trước hết chúng ta cùng nhắc lại một số kiến thức liên quan đến xử lý đồng thời và xử lý song song (parallelism).

1.5.1 Xử lý đồng thời là gì ?

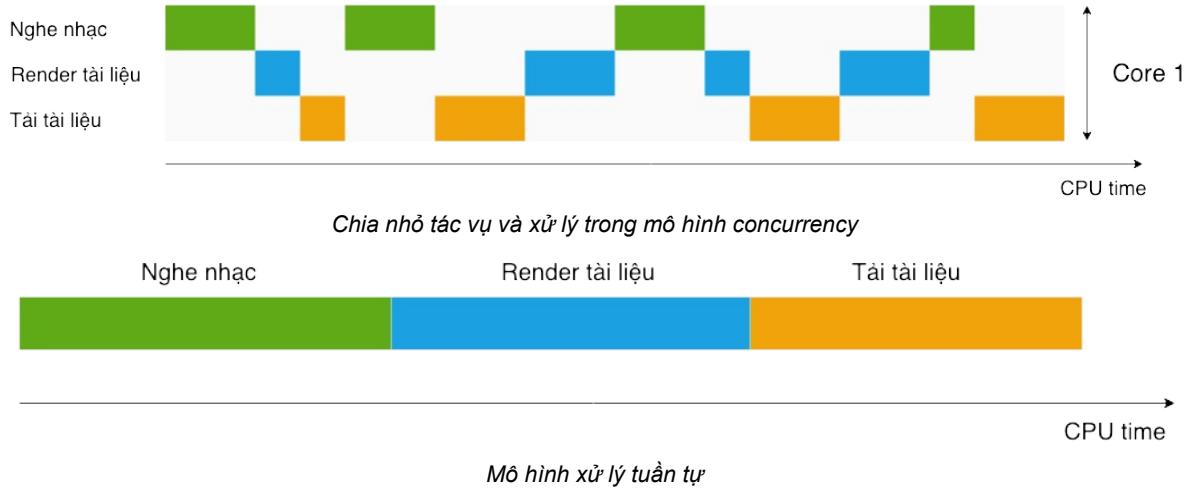
Xử lý đồng thời là khả năng phân chia và điều phối nhiều tác vụ khác nhau trong cùng một khoảng thời gian và tại một thời điểm chỉ có thể xử lý một tác vụ. Khái niệm này trái ngược với **xử lý tuần tự** (sequential processing). Xử lý tuần tự là khả năng xử lý chỉ một tác vụ trong một khoảng thời gian, các tác vụ sẽ được thực thi theo thứ tự hết tác vụ này sẽ thực thi tiếp tác vụ khác.

Concurrency is about dealing with lots of things at once-Rob Pike

Ví dụ như chúng ta vừa muốn nghe nhạc vừa đọc [Advanced Go book](#) và trong lúc đọc bạn muốn tải bộ tài liệu về từ [zalopay-oss](#). Nếu như theo mô hình xử lý tuần tự thì trèn duyệt web sẽ phải thực hiện việc nghe nhạc xong, rồi tới việc mở [Advanced Go book](#) online để đọc và sau khi đọc xong chúng ta mới có thể tải về được. Đối với mô hình xử lý đồng thời thì ta có thể làm 3 tác vụ trên trong cùng một khoảng thời gian. Chúng ta có thể vừa nghe nhạc vừa lướt đọc tài liệu mà vừa có thể tải bộ tài liệu này về máy. Vậy làm thế nào để có thể xử lý đồng thời như vậy ?

Tất cả các chương trình đang chạy trong máy tính chúng ta chạy đều do hệ điều hành quản lý, với mỗi chương trình đang chạy như vậy được gọi là một process (tiến trình) và được cấp một process id (PID) để hệ điều hành dễ dàng quản lý. Các tác vụ của tiến trình sẽ được CPU core (nhân CPU) của máy tính xử lý. Vậy làm sao 1 máy tính có CPU 1 nhân có thể làm được việc xử lý đồng thời nhiều tác vụ của các tiến trình cùng lúc. Bởi vì bản chất tại một thời điểm nhân CPU chỉ có thể xử lý một tác vụ.

Như câu nói của Rob Pike, ông đã sử dụng từ **dealing** (phân chia xử lý) để nói đến khái niệm concurrency. Thật như vậy, nhân CPU không bao giờ đợi xử lý xong một tác vụ rồi mới xử lý tiếp tác vụ khác, mà nhân CPU đã chia các tác vụ lớn thành các tác vụ nhỏ hơn và sắp xếp xen kẽ lẫn nhau. Nhân CPU tận dụng thời gian rảnh của tác vụ này để đi làm tác vụ khác, một lúc thì làm tác vụ nhỏ này, một lúc khác thì làm tác vụ nhỏ khác. Như vậy chúng ta sẽ cảm thấy máy tính xử lý nhiều việc cùng lúc tại cùng thời điểm. Nhưng bản chất bên dưới nhân CPU thì nó chỉ có thể thực thi một tác vụ nhỏ trong tác vụ lớn tại thời điểm đó.



1.5.2 Xử lý song song là gì ?

Xử lý song song là khả năng xử lý nhiều tác vụ khác nhau trong cùng một thời điểm, các tác vụ này hoàn toàn độc lập với nhau. Xử lý song song chỉ có thể thực hiện trên máy tính có số nhân lớn hơn 1. Thay vì một nhân CPU chúng ta chỉ có thể xử lý một tác vụ nhỏ tại một thời điểm thì khi số nhân CPU có nhiều hơn chúng ta có thể xử lý các tác vụ song song với nhau cùng lúc trên các nhân CPU.

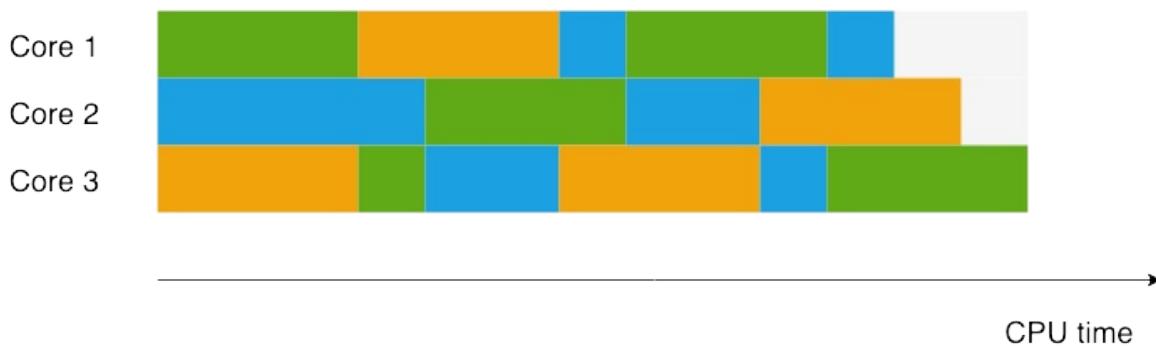
Parallelism is about doing lots of things at once-Rob Pike

Cũng lấy ví dụ nghe nhạc, đọc tài liệu và tải tài liệu ở trên, thì trong mô hình xử lý song song sẽ như sau.



Mô hình xử lý song song các tác vụ cùng một thời điểm

Trong thực tế, trên mỗi nhân của CPU vẫn xảy ra quá trình xử lý đồng thời miễn là tại một thời điểm không có xảy ra việc xử lý cùng một tác vụ trên hai nhân CPU khác nhau, mô hình trên vẽ lại như sau:



Mô hình xử lý song song

Chúng ta nên nắm được mô hình xử lý đồng thời khác với mô hình xử lý song song, tuy hai mô hình đều nêu lên việc xử lý nhiều tác vụ trong cùng một thời điểm. Trong một bài diễn thuyết của Rob Pike, ông cũng đã trình bày và phân biệt hai mô hình trên. Các bạn có thể xem buổi diễn thuyết [ở đây](#).

1.5.3 Xử lý đồng thời trong Golang

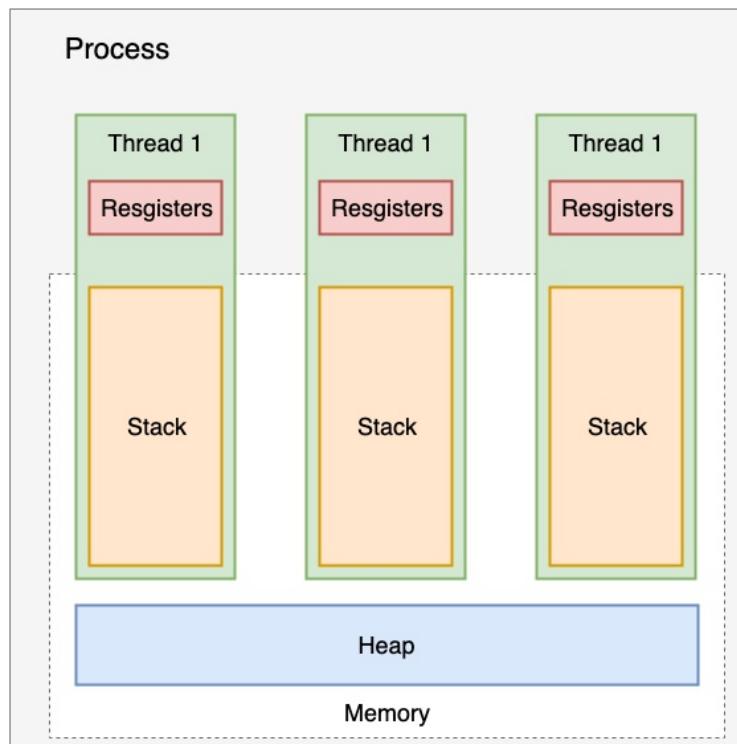
Trước khi tìm hiểu về cách Golang xử lý đồng thời như thế nào, chúng ta cùng nhắc lại một số khái niệm về tiến trình (process) và luồng (thread) trong hệ điều hành.

Process

Tiến trình có thể hiểu đơn giản là một chương trình đang chạy trong máy tính. Khi chúng ta mở trình duyệt web để đọc [Advanced Go book](#) thì đây được xem là một tiến trình. Khi chúng ta viết 1 chương trình máy tính bằng ngôn ngữ lập trình như C, Java, hay Go, sau khi tiến hành biên dịch và chạy chương trình thì hệ điều hành sẽ cấp cho chương trình một không gian bộ nhớ nhất định, PID (process ID),... Mỗi tiến trình có ít nhất một luồng chính (main thread) để chạy chương trình, nó như là xương sống của chương trình vậy. Khi luồng chính này ngừng hoạt động tương ứng với việc chương trình bị tắt.

Thread

Thread hay được gọi là tiểu trình nó là một luồng trong tiến trình đang chạy. Các luồng được chạy song song trong mỗi tiến trình và có thể truy cập đến vùng nhớ được cung cấp bởi tiến trình, các tài nguyên của hệ điều hành,...



Mô hình xử lý song song

Các thread trong process sẽ được cấp phát riêng một vùng nhớ `stack` để lưu các biến riêng của thread đó. Stack được cấp phát cố định khoảng `1MB-2MB`. Ngoài ra các thread chia sẻ chung vùng nhớ `heap` của process. Khi process tạo quá nhiều thread sẽ dẫn đến tình trạng `stack overflow`. Khi các thread sử dụng chung vùng nhớ sẽ dễ gây ra hiện tượng `race condition`. Ở phần sau chúng ta sẽ tìm hiểu cách Golang xử lý như thế nào để tránh lỗi race condition.

Ở các phần trên chúng ta đã cùng nhau thảo luận về mô hình xử lý đồng thời và xử lý song song các tác vụ, các tác vụ ở đây sẽ được thực hiện bởi các thread khác nhau. Vì vậy tương quan ở đây là khi chúng ta xử lý các tác vụ theo mô hình đồng thời hay song song cũng có nghĩa là có nhiều thread chạy đồng thời hay song song (multi-threading). Số lượng thread chạy song song trong cùng một thời điểm sẽ bằng với số lượng nhân CPU mà máy tính chúng ta có. Vì vậy khi chúng ta lập trình mà tạo quá nhiều thread thì cũng không có giúp cho chương trình chúng ta chạy nhanh hơn, mà còn gây ra lỗi và làm chậm chương trình. Theo kinh nghiệm lập trình chúng ta chỉ nên tạo số thread bằng số nhân CPU * 2.

Như mình đã trình bày ở phần trên thì khi xử lý đồng thời thì tại một thời điểm chỉ có một tác vụ được xử lý hay một thread được chạy trên một nhân CPU. Khi nhân CPU chuyển qua xử lý tác vụ khác cũng có nghĩa là thread khác được chạy. Thao tác đó được gọi là `context switch`. Các bạn có thể xem chi tiết [ở đây](#).

Goroutines và system threads

Goroutines là một đơn vị concurrency của ngôn ngữ Go. Hay nói cách khác Golang sử dụng goroutine để xử lý đồng thời nhiều tác vụ. Việc khởi tạo goroutines sẽ ít tốn chi phí hơn khởi tạo `thread` so với các ngôn ngữ khác. Cách khởi tạo goroutine chỉ đơn giản thông qua từ khóa `go`. Về góc nhìn hiện thực, goroutines và thread không giống nhau.

Đầu tiên, system thread sẽ có một kích thước vùng nhớ stack cố định (thông thường vào khoảng 2MB). Vùng nhớ stack chủ yếu được dùng để lưu trữ những tham số, biến cục bộ và địa chỉ trả về khi chúng ta gọi hàm.

Kích thước cố định của stack sẽ dẫn đến hai vấn đề:

- Stack overflow với những chương trình gọi hàm đệ quy sâu.
- Lãng phí vùng nhớ đối với chương trình đơn giản.

Giải pháp cho vấn đề này chính là cấp phát linh hoạt vùng nhớ stack:

- Một Goroutines sẽ được bắt đầu bằng một vùng nhớ nhỏ (khoảng 2KB hoặc 4KB).
- Khi gọi đệ quy sâu (không gian stack hiện tại là không đủ) Goroutines sẽ tự động tăng không gian stack (kích thước tối đa của stack có thể được đặt tới 1GB).
- Bởi vì chi phí của việc khởi tạo là nhỏ, chúng ta có thể dễ dàng giải phóng hàng ngàn goroutines.

Với ngôn ngữ lập trình khác như Java thì các thread được quản lý bởi hệ điều hành, có nghĩa là chương trình chúng ta đang xử lý đồng thời bị phụ thuộc vào hệ điều hành. Trong Golang sử dụng `Go runtime` có riêng cơ chế định thời cho Goroutines, nó dùng một số kỹ thuật để ghép `M Goroutines` trên `N thread` của hệ điều hành. Cơ chế định thời Goroutines tương tự với cơ chế định thời của hệ điều hành nhưng chỉ ở mức chương trình. Biến `runtime.GOMAXPROCS` quy định số lượng thread hiện thời chạy trên các Goroutines.

Chúng ta cùng xem qua bảng so sánh giữa Goroutines và Thread được tham khảo [ở đây](#):

Thread	Goroutines
Thread được quản lý bởi hệ điều hành và phụ thuộc vào số nhân CPU	Goroutines được quản lý bởi go runtime và không phụ thuộc vào số nhân CPU
Thread thường có kích cỡ stack cố định từ 1-2MB	Goroutines có kích cỡ stack từ 8KB (2 ở Go1.4) ở phiên bản mới
Vùng nhớ stack được cấp phát trong thời gian compile và không được tăng thêm	Vùng nhớ stack có thể tăng đến 1GB
Giao tiếp giữa các thread khá khó. Có độ trễ lớn trong việc tương tác giữa các thread	Goroutine sử dụng channels để tương tác với nhau với độ trễ thấp
Thread có định danh bằng TID trong mỗi process	Goroutines không có định danh vì Golang không có Thread Local Storage
Việc khởi tạo, giải phóng thread tốn nhiều thời gian	Goroutines được tạo và giải phóng bởi go runtime nên rất nhanh
Tốn nhiều chi phí context switch trong thread, do hệ điều hành xếp lịch	Tốn ít chi phí hơn do go runtime quản lý

Mô hình xử lý song song

1.5.4 Ví dụ Goroutine

Ví dụ 1:

```
func main() {
    // sử dụng từ khoá go để tạo goroutine
    go fmt.Println("Xin chào goroutine")
    fmt.Println("Xin chào main goroutine ")
}
```

Chương trình trên có lúc in ra cả hai câu không đúng như thứ tự trên, có lúc sẽ chỉ in được câu "Xin chào main goroutine". Chúng ta đã biết khi hàm main chạy xong thì chương trình sẽ dừng. Hàm main cũng là một goroutine và chạy đồng thời với hàm `fmt.Println`. Nên có trường hợp hàm main chạy xong và dừng trước khi hàm `fmt.Println` được chạy.

Chúng ta có thể làm như sau để có thể in ra cả hai câu:

```
func main() {
    // sử dụng từ khoá go để tạo goroutine
    go fmt.Println("Hello from another goroutine")
```

```

fmt.Println("Hello from main goroutine")

// chờ 1 giây để có thể chạy được goroutine
// của hàm fmt.Println trước khi hàm main kết thúc
time.Sleep(time.Second)
}

```

Sau khi chương trình chạy xong các goroutine sẽ bị huỷ.

Ví dụ 2:

Chúng ta có thể sử dụng goroutine bằng cách sau.

```

func MyPrintln(id int, delay time.Duration) {
    go func() {
        time.Sleep(delay)
        fmt.Println("Xin chào, tôi là goroutine: ", id)
    }()
}

func main() {
    for i := 0; i < 100; i++ {
        MyPrintln(i, 1*time.Second)
    }

    time.Sleep(10 * time.Second)
    fmt.Println("Chương trình kết thúc")
}

```

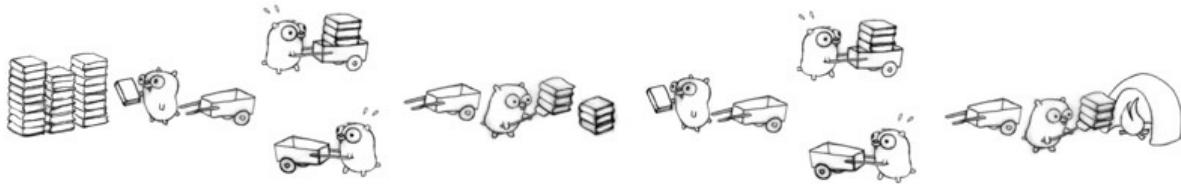
Ở phần tiếp theo chúng ta sẽ đi tới những ví dụ phức tạp hơn như cách xử lý race-condition trong Golang, sử dụng channel để chặn các goroutine,...

Liên kết

- Phần tiếp theo: [Mô hình thực thi đồng thời](#)
- Phần trước: [Functions, Methods và Interfaces](#)
- [Mục lục](#)

1.6. Mô hình thực thi đồng thời

Một điểm mạnh của Go là tích hợp sẵn cơ chế xử lý đồng thời (concurrency). Lý thuyết về hệ thống tương tranh của Go là CSP (Communicating Sequential Process) được đề xuất bởi Hoare vào năm 1978. CSP được áp dụng lần đầu cho máy tính đa dụng T9000 mà Hoare có tham gia. Từ NewSqueak, Alef, Limbo đến Golang hiện tại, Rob Pike, người có hơn 20 năm kinh nghiệm thực tế với CSP, rất quan tâm đến tiềm năng áp dụng CSP vào ngôn ngữ lập trình đa dụng. Khái niệm cốt lõi của lý thuyết CSP cũng được áp dụng vào lập trình concurrency trong Go.

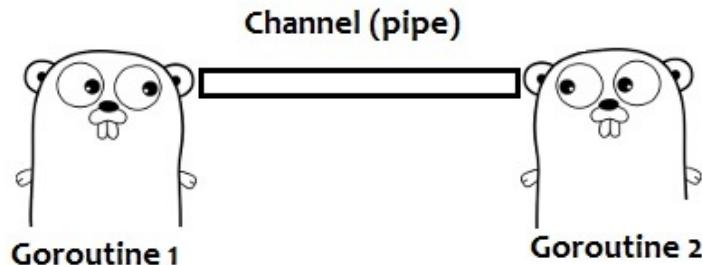


Go concurrency

Ở phần này chúng ta cùng xem qua các cách dùng goroutine cũng như các cách xử lý goroutine mà chúng ta thường gặp trong lúc lập trình.

1.6.1. Phiên bản concurrency với Hello World

Trong hầu hết các ngôn ngữ hiện đại, vấn đề chia sẻ tài nguyên được giải quyết bằng cơ chế đồng bộ hóa như khóa (lock) nhưng Golang có cách tiếp cận riêng là chia sẻ giá trị thông qua channel.



Goroutine trao đổi giá trị qua channel

Trên thực tế khi nhiều thread thực thi độc lập chúng hiếm khi chủ động chia sẻ tài nguyên. Tại bất kỳ thời điểm nào, tốt nhất là chỉ Goroutine sở hữu tài nguyên của chính mình. Golang có một triết lý được thể hiện bằng slogan:

Do not communicate by sharing memory; instead, share memory by communicating.

Do not communicate through shared memory, but share memory through communication.

Mặc dù các vấn đề tương tranh đơn giản như tham chiếu đến biến đếm có thể được hiện thực bằng `atomic operations` hoặc `mutex lock`, nhưng việc kiểm soát truy cập thông qua Channel giúp cho code của chúng ta clean và "Golang" hơn.

Áp dụng Mutex

Xem xét đoạn code sau:

```

func main() {
    var mu sync.Mutex

    // khởi chạy một goroutine chạy đồng thời với main
    go func(){
        fmt.Println("Hello World")

        // không thể đảm bảo hàm này sẽ chạy trước `mu.Unlock()` phía dưới
        mu.Lock()
    }()

    // xảy ra lỗi vì `mu` đang ở trạng thái unlocked
    mu.Unlock()
}

```

Ở đây, `mu.Lock()` và `mu.Unlock()` không ở trong cùng một Goroutine, vì vậy nó không đáp ứng được mô hình bộ nhớ nhất quán tuần tự (sequential consistency memory model).

Sửa lại đoạn code trên như sau:

```

func main() {
    var mu sync.Mutex

    // lệnh này khiến main thread bị block
    mu.Lock()

    go func(){
        fmt.Println("Hello World")

        // lệnh này Unlock cho main thread
        mu.Unlock()
    }()

    mu.Lock()
}

```

Áp dụng Channel

Đồng bộ hóa với mutex là một cách tiếp cận ở mức độ tương đối đơn giản. Bây giờ ta sẽ sử dụng một unbuffered channel để hiện thực đồng bộ hóa:

```

func main() {
    done := make(chan int)

    go func(){
        fmt.Println("Hello World")

        // chỉ sau khi goroutine này hoàn thành thao tác nhận
        // thì thao tác gửi ở main thread mới được kết thúc
        <-done
    }()

    // thao tác gửi qua channel `done`
    done <- 1
}

```

Cách này gặp bất cập với buffered channel vì lúc đó không có gì đảm bảo rằng goroutine sẽ in ra trước khi thoát `main`. Cách tiếp cận tốt hơn là hoán đổi hướng gửi và nhận của channel để tránh các sự kiện đồng bộ hóa bị ảnh hưởng bởi kích thước buffer của nó:

```

func main() {
    done := make(chan int, 1)
}

```

```

go func(){
    fmt.Println("Hello World")

    // gửi 1 giá trị vào channel thông báo kết thúc goroutine này
    done <- 1
}()

// main thread nhận giá trị từ channel và thoát khỏi
// trạng thái block
<-done
}

```

Dựa trên buffered channel, chúng ta có thể dễ dàng mở rộng thread print đến N. Ví dụ sau là mở 10 goroutine để in riêng biệt:

```

func main() {
    done := make(chan int, 10)

    // mở ra N goroutine
    for i := 0; i < cap(done); i++ {
        go func(){
            fmt.Println("Hello World")
            done <- 1
        }()
    }

    // đợi cả 10 goroutine hoàn thành
    for i := 0; i < cap(done); i++ {
        <-done
    }
}

```

Sử dụng sync.WaitGroup thay cho Channel

Một cách đơn giản hơn là sử dụng `sync.WaitGroup` để chờ một tập các sự kiện:

```

func main() {
    var wg sync.WaitGroup

    // mở N goroutine
    for i := 0; i < 10; i++ {
        // tăng số lượng sự kiện chờ, hàm này phải được
        // đảm bảo thực thi trước khi bắt đầu 1 goroutine chạy nền
        wg.Add(1)

        go func() {
            fmt.Println("Hello World")

            // cho biết hoàn thành một sự kiện
            wg.Done()
        }()
    }

    // đợi N goroutine hoàn thành
    wg.Wait()
}

```

1.6.2. Tác vụ Atomic

Tác vụ atomic trên một vùng nhớ chia sẻ thì đảm bảo rằng vùng nhớ đó chỉ có thể được truy cập bởi một Goroutine tại một thời điểm. Để đạt được điều này ta có thể dùng `sync.Mutex`.

Sử dụng sync.Mutex

```

import (
    // package cần dùng
    "sync"
)

// total là một atomic struct
var total struct {
    sync.Mutex
    value int
}

func worker(wg *sync.WaitGroup) {
    // thông báo hoàn thành khi ra khỏi hàm
    defer wg.Done()

    for i := 0; i <= 100; i++ {
        // chặn các Goroutines khác vào
        total.Lock()
        // bây giờ, lệnh total.value += i được đảm bảo là atomic (đơn nguyên)
        total.value += i
        // bỏ chặn
        total.Unlock()
    }
}

func main() {
    // khai báo wg để main Goroutine dừng chờ các Goroutines khác trước khi kết thúc chương trình
    var wg sync.WaitGroup
    // wg cần chờ 2 Goroutines khác
    wg.Add(2)
    // thực thi Goroutines thứ nhất
    go worker(&wg)
    // thực thi Goroutines thứ hai
    go worker(&wg)
    // wg bắt đầu đợi để 2 Goroutines kia xong
    wg.Wait()
    // in ra kết quả thực thi
    fmt.Println(total.value)
}

```

Trong một chương trình đồng thời, ta cần có cơ chế để `lock` và `unlock` trước và sau khi truy cập vào vùng `critical section`. Nếu không có sự bảo vệ biến `total`, kết quả cuối cùng có thể bị sai khác do sự truy cập đồng thời của nhiều thread.

Sử dụng sync/atomic

Thay vì dùng mutex, chúng ta cũng có thể dùng package `sync/atomic`, đây là giải pháp hiệu quả hơn đối với một biến số học.

```

import (
    "sync"
    // khai báo biến gói sync/atomic
    "sync/atomic"
)

// biến total được truy cập đồng thời
var total uint64

```

```

func worker(wg *sync.WaitGroup) {
    // wg thông báo hoàn thành khi ra khỏi hàm
    defer wg.Done()

    var i uint64
    for i = 0; i <= 100; i++ {
        // lệnh cộng atomic.AddUint64 total được đảm bảo là atomic (đơn nguyên)
        atomic.AddUint64(&total, i)
    }
}

func main() {
    // wg được dùng để dừng hàm main đợi các Goroutines khác
    var wg sync.WaitGroup
    // wg cần đợi hai Goroutines gọi lệnh Done() mới thực thi tiếp
    wg.Add(2)
    // tạo Goroutines thứ nhất
    go worker(&wg)
    // tạo Goroutines thứ hai
    go worker(&wg)
    // bắt đầu việc đợi
    wg.Wait()
    // in ra kết quả
    fmt.Println(total)
}

```

Để ghi và đọc atomic trên những đối tượng phức tạp hơn thì ta dùng kiểu [atomic.Value](#), ví dụ:

```

package main

import (
    "sync/atomic"
    "time"
)

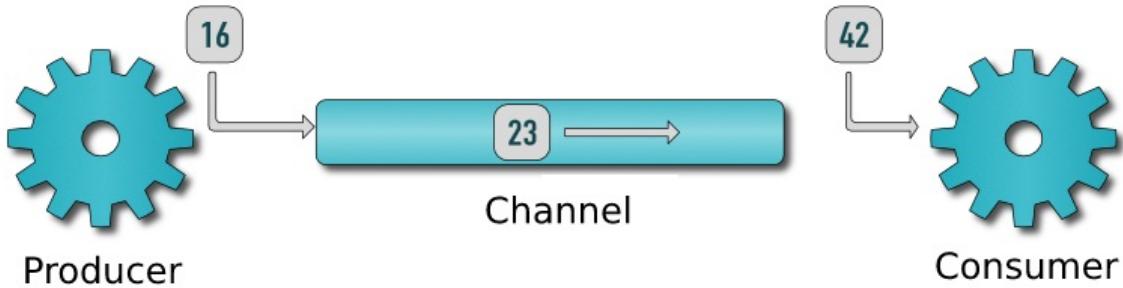
func loadConfig() map[string]string {
    return make(map[string]string)
}

func requests() chan int {
    return make(chan int)
}

func main() {
    // nắm giữ thông tin cấu hình của server
    var config atomic.Value
    // khởi tạo giá trị ban đầu
    config.Store(loadConfig())
    go func() {
        // cập nhật thông tin sau mỗi 10 giây
        for {
            time.Sleep(10 * time.Second)
            config.Store(loadConfig())
        }
    }()
    // tạo nhiều worker xử lý request
    // dùng thông tin cấu hình gần nhất
    for i := 0; i < 10; i++ {
        go func() {
            for r := range requests() {
                c := config.Load()
                // xử lý request với cấu hình c
                _, _ = r, c
            }
        }()
    }
}

```

1.6.3. Mô hình Producer Consumer



Mô hình Producer - Consumer

Ví dụ phổ biến nhất về lập trình concurrency là mô hình Producer Consumer, giúp tăng tốc độ xử lý chung của chương trình bằng cách cân bằng sức mạnh của các thread "sản xuất" (produce) và "tiêu thụ" (consume).

Producer tạo ra một số dữ liệu và sau đó đưa nó vào hàng đợi, cùng lúc đó consumer cũng lấy dữ liệu từ hàng đợi này ra để xử lý. Điều này làm cho produce và consume trở thành hai quá trình bất đồng bộ. Khi không có dữ liệu trong hàng đợi kết quả, consumer sẽ chờ đợi ở trạng thái "đói", còn khi dữ liệu trong hàng đợi bị đầy, producer phải đổi mới với vấn đề mất mát dữ liệu khi CPU phải loại bỏ dữ liệu trong đó để nạp thêm.

Golang hiện thực cơ chế này khá đơn giản:

```
// producer: liên tục tạo ra một chuỗi số nguyên dựa trên bội số factor và đưa vào channel
func Producer(factor int, out chan<- int) {
    for i := 0; ; i++ {
        out <- i*factor
    }
}

// consumer: liên tục lấy các số từ channel ra để print
func Consumer(in <-chan int) {
    for v := range in {
        fmt.Println(v)
    }
}
func main() {
    // hàng đợi
    ch := make(chan int, 64)

    // tạo một chuỗi số với bội số 3
    go Producer(3, ch)

    // tạo một chuỗi số với bội số 5
    go Producer(5, ch)

    // tạo consumer
    go Consumer(ch)

    // thoát ra sau khi chạy trong một khoảng thời gian nhất định
    time.Sleep(5 * time.Second)
}
```

Chúng ta có thể để hàm `main` giữ trạng thái block mà không thoát và chỉ thoát khỏi chương trình khi người dùng gõ `Ctrl-C`:

```

func main() {
    // hàng đợi
    ch := make(chan int, 64)

    go Producer(3, ch)
    go Producer(5, ch)
    go Consumer(ch)

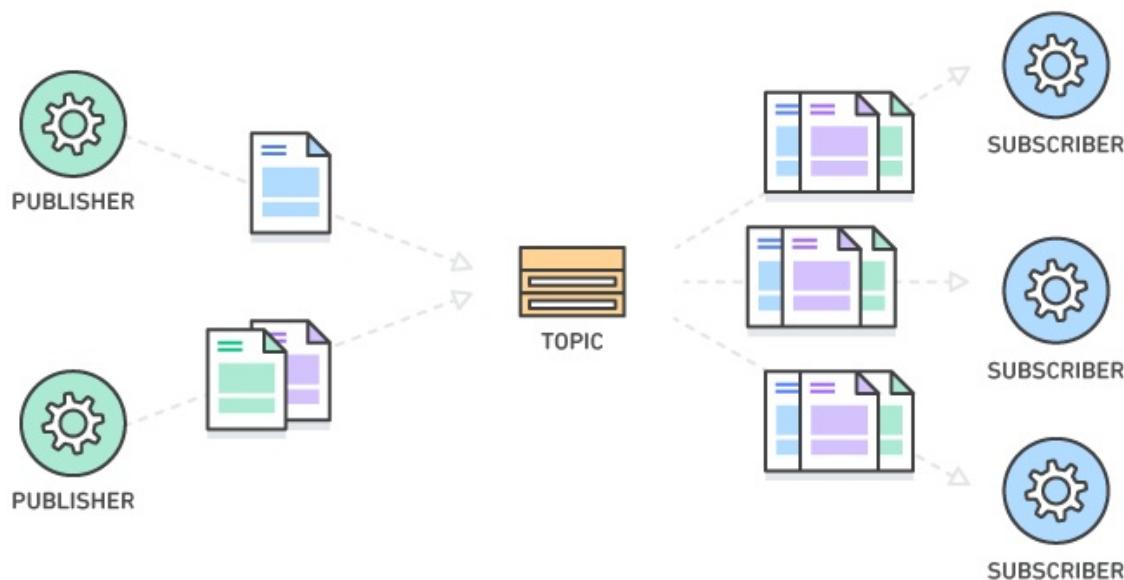
    // Ctrl+C để thoát
    sig := make(chan os.Signal, 1)
    signal.Notify(sig, syscall.SIGINT, syscall.SIGTERM)
    fmt.Printf("quit (%v)\n", <-sig)
}

```

Có 2 producer trong ví dụ trên và không có sự kiện đồng bộ nào giữa hai producer mà chúng concurrency. Do đó, thứ tự của chuỗi output ở consumer là không xác định.

1.6.4. Mô hình Publish Subscribe

Mô hình publish-and-subscribe thường được viết tắt là mô hình pub/sub. Trong mô hình này, producer trở thành publisher và consumer trở thành subscriber, đồng thời producer:consumer là mối quan hệ M:N.



Mô hình Publish - Subscribe

Trong mô hình producer-consumer truyền thống, thông điệp được gửi đến hàng đợi và mô hình publish-subscription sẽ publish thông điệp đến một topic.

Để hiện thực mô hình này ta implement package `pubsub` :

```

// package pubsub implements a simple multi-topic pub-sub library.
package pubsub

import (
    "sync"
    "time"
)

type (
    // subscriber thuộc kiểu channel

```

```

subscriber chan interface{}

// topic là một filter
topicFunc func(v interface{}) bool
)

type Publisher struct {
    // Read/Write Mutex
    m sync.RWMutex

    // kích thước hàng đợi
    buffer int

    // timeout cho việc publishing
    timeout time.Duration

    // subscriber đã subscribe vào topic nào
    subscribers map[subscriber]topicFunc
}

// constructor với timeout và độ dài hàng đợi
func NewPublisher(publishTimeout time.Duration, buffer int) *Publisher {
    return &Publisher{
        buffer:     buffer,
        timeout:   publishTimeout,
        subscribers: make(map[subscriber]topicFunc),
    }
}

// thêm subscriber mới, đăng ký hết tất cả topic
func (p *Publisher) Subscribe() chan interface{} {
    return p.SubscribeTopic(nil)
}

// thêm subscriber mới, subscribe các topic đã được filter lọc
func (p *Publisher) SubscribeTopic(topic topicFunc) chan interface{} {
    ch := make(chan interface{}, p.buffer)
    p.m.Lock()
    p.subscribers[ch] = topic
    p.m.Unlock()
    return ch
}

// hủy subscribe
func (p *Publisher) Evict(sub chan interface{}) {
    p.m.Lock()
    defer p.m.Unlock()

    delete(p.subscribers, sub)
    close(sub)
}

// publish ra 1 topic
func (p *Publisher) Publish(v interface{}) {
    p.m.RLock()
    defer p.m.RUnlock()

    var wg sync.WaitGroup
    for sub, topic := range p.subscribers {
        wg.Add(1)
        go p.sendTopic(sub, topic, v, &wg)
    }
    wg.Wait()
}

// đóng 1 đối tượng publisher và đóng tất cả các subscriber
func (p *Publisher) Close() {
    p.m.Lock()
    defer p.m.Unlock()
}

```

```

for sub := range p.subscribers {
    delete(p.subscribers, sub)
    close(sub)
}
}

// gửi 1 topic có thể duy trì trong thời gian chờ wg
func (p *Publisher) sendTopic(
    sub subscriber, topic topicFunc, v interface{}, wg *sync.WaitGroup,
) {
    defer wg.Done()
    if topic != nil && !topic(v) {
        return
    }

    select {
    case sub <- v:
    case <-time.After(p.timeout):
    }
}

```

Trong ví dụ sau đây, 2 subscriber đăng ký hết tất cả các topic với "golang":

```

import (
    "./pubsub"
    "time"
    "strings"
    "fmt"
)
func main() {
    // khởi tạo 1 publisher
    p := pubsub.NewPublisher(100*time.Millisecond, 10)

    // để đảm bảo p được đóng trước khi exit
    defer p.Close()

    // `all` subscribe hết tất cả topic
    all := p.Subscribe()

    // subscribe các topic có "golang"
    golang := p.SubscribeTopic(func(v interface{}) bool {
        if s, ok := v.(string); ok {
            return strings.Contains(s, "golang")
        }
        return false
    })

    // publish ra 2 topic
    p.Publish("hello, world!")
    p.Publish("hello, golang!")

    // print những gì subscriber `all` nhận được
    go func() {
        for msg := range all {
            fmt.Println("all:", msg)
        }
    }()

    // print những gì subscriber `golang` nhận được
    go func() {
        for msg := range golang {
            fmt.Println("golang:", msg)
        }
    }()
}

// thoát ra sau khi chạy 3 giây

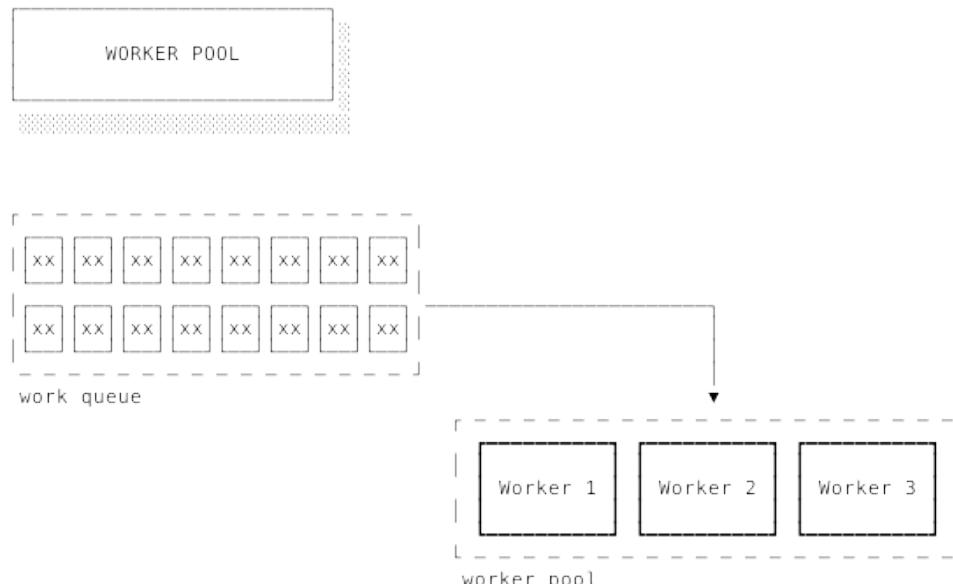
```

```
    time.Sleep(3 * time.Second)
}
```

Trong mô hình pub/sub, mỗi thông điệp được gửi tới nhiều subscriber. Publisher thường không biết hoặc không quan tâm subscriber nào nhận được thông điệp. Subscriber và publisher có thể được thêm vào động ở thời điểm thực thi, cho phép các hệ thống phức tạp có thể phát triển theo thời gian. Trong thực tế, những ứng dụng như dự báo thời tiết có thể áp dụng mô hình concurrency này.

1.6.5. Kiểm soát số lượng goroutine

Goroutine là một tính năng mạnh mẽ của Go, mất chi phí rất ít để sử dụng, nhưng tất nhiên nếu dùng với số lượng quá lớn sẽ chiếm gây nhiều lãng phí và cần có một cơ chế để kiểm soát. Một cách thông dụng để đạt được mục đích trên là dùng worker pool.



Mô hình Worker pool

Đầu tiên tạo ra các worker:

```
func worker(queue chan int, worknumber int, done chan bool) {
    for j := range queue {
        fmt.Println("worker", worknumber, "finished job", j)
        done <- true
    }
}
```

Sau đó có thể áp dụng như sau:

```
func main() {
    // queue of jobs
    q := make(chan int)

    // done channel lấy ra kết quả của jobs
    done := make(chan bool)

    // số lượng worker trong pool
    numberOfWorkers := 4
    for i := 0; i < numberOfWorkers; i++ {
        go worker(q, i, done)
    }
}
```

```
// đưa job vào queue
numberOfJobs := 17
for j := 0; j < numberOfJobs; j++ {
    go func(j int) {
        q <- j
    }(j)
}

// chờ nhận đủ kết quả
for c := 0; c < numberOfJobs; c++ {
    <-done
}
}
```

1.6.6. Dọn dẹp Goroutine

Sau khi job queue rỗng, ta sẽ phải dừng tất cả worker. Goroutine dù khá nhẹ nhưng vẫn không phải miễn phí, nhất là với các hệ thống lớn, dù chỉ là các chi phí nhỏ nhất cũng có thể trở nên khác biệt lớn nếu thay đổi.

Cách đơn giản là dùng kill channel để phát ra tín hiệu ngừng cho goroutine.

```
func main() {
    // channel để terminate các worker
    killsignal := make(chan bool)

    // queue các jobs
    q := make(chan int)
    // done channel nhận vào kết quả của các job
    done := make(chan bool)

    // số lượng worker trong pool
    numberOfWorkers := 4
    for i := 0; i < numberOfWorkers; i++ {
        go worker(q, i, done, killsignal)
    }

    // đưa job vào queue
    numberOfJobs := 17
    for j := 0; j < numberOfJobs; j++ {
        go func(j int) {
            q <- j
        }(j)
    }

    // chờ để nhận đủ kết quả
    for c := 0; c < numberOfJobs; c++ {
        <-done
    }

    // dọn dẹp các worker
    close(killsignal)
    time.Sleep(2 * time.Second)
}
```

Trong đó các worker được thiết kế như sau:

```
func worker(queue chan int, worknumber int, done, ks chan bool) {
    for true {
        // dùng select để chờ cùng lúc trên cả 2 channel
        select {
            // xử lý job trong channel queue
            case k := <-queue:
```

```

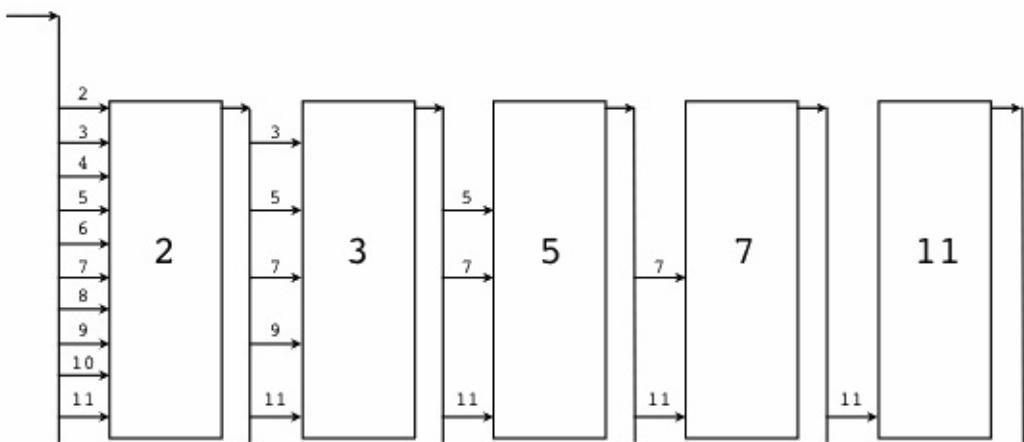
        fmt.Println("doing work!", k, "worknumber", worknumber)
        done <- true

        // nếu nhận được kill signal thì return
        case <-ks:
            fmt.Println("worker halted, number", worknumber)
            return
        }
    }
}

```

1.6.7. Sàng số nguyên tố

Trong phần 1.2, chúng tôi đã trình bày việc triển khai phiên bản concurrency của sàng số nguyên tố để chứng minh tính concurrency của Newsqueak. Nguyên tắc "sàng số nguyên tố" như sau:



Sàng số nguyên tố

Chúng ta cần khởi tạo một chuỗi các số tự nhiên $2, 3, 4, \dots$ (không bao gồm 0, 1):

```

// trả về channel tạo ra chuỗi số: 2, 3, 4, ...
func GenerateNatural() chan int {
    ch := make(chan int)
    go func() {
        for i := 2; ; i++ {
            ch <- i
        }
    }()
    return ch
}

```

Tiếp theo xây dựng một sàng cho mỗi số nguyên tố: đề xuất một số là bội số của số nguyên tố trong chuỗi đầu vào và trả về một chuỗi mới, đó là một channel mới.

```

// bộ lọc: xóa các số có thể chia hết cho số nguyên tố
func PrimeFilter(in <-chan int, prime int) chan int {
    out := make(chan int)
    go func() {
        for {
            if i := <-in; i%prime != 0 {
                out <- i
            }
        }
    }()
    return out
}

```

```

    }()
    return out
}

```

Bây giờ ta có thể sử dụng bộ lọc này trong hàm `main`:

```

func main() {
    // chuỗi số: 2, 3, 4, ...
    ch := GenerateNatural()
    for i := 0; i < 100; i++ {
        // số nguyên tố mới
        prime := <-ch

        // Bộ lọc dựa trên số nguyên tố mới
        fmt.Printf("%v: %v\n", i+1, prime)
        ch = PrimeFilter(ch, prime)

        // dựa trên chuỗi số còn lại trong channel để lọc
        // các số nguyên tố tiếp theo với các số được
        // trích xuất dưới dạng filter. Các channel tương ứng
        // với các sàng số nguyên tố khác nhau được kết nối liên tiếp nhau.
    }
}

```

1.6.8. Kẻ thắng làm vua

Có nhiều động lực để lập trình concurrency nhưng tiêu biếu là vì lập trình concurrency có thể đơn giản hóa các vấn đề. Lập trình concurrency cũng có thể cải thiện hiệu năng. Mở hai thread trên CPU đa lõi thường nhanh hơn mở một thread. Trên thực tế về mặt cải thiện hiệu suất, chương trình không chỉ đơn giản là chạy nhanh, mà trong nhiều trường hợp chương trình có thể đáp ứng yêu cầu của người dùng một cách nhanh chóng là điều quan trọng nhất. Khi không có yêu cầu từ người dùng cần xử lý, nên xử lý một số tác vụ nền có độ ưu tiên thấp.

Giả sử chúng ta muốn nhanh chóng tìm kiếm các chủ đề liên quan đến "golang", có thể mở nhiều công cụ tìm kiếm như Bing, Google hoặc Yahoo. Khi tìm kiếm trả về kết quả trước, ta có thể đóng các trang tìm kiếm khác. Do ảnh hưởng của môi trường mạng và thuật toán của công cụ tìm kiếm mà một số công cụ tìm kiếm có thể trả về kết quả tìm kiếm nhanh hơn. Chúng ta có thể sử dụng một chiến lược tương tự để viết chương trình này:

```

func main() {
    // tạo ra một channel với buffer đủ lớn để đảm bảo
    // không bị block do kích thước của buffer.
    ch := make(chan string, 32)

    // chạy nhiều goroutine dưới nền và gửi yêu cầu
    // tìm kiếm đến các công cụ tìm kiếm khác nhau
    go func() {
        ch <- searchByBing("golang")
    }()
    go func() {
        ch <- searchByGoogle("golang")
    }()
    go func() {
        ch <- searchByBaidu("golang")
    }()

    // khi bất kỳ công cụ tìm kiếm nào có kết quả
    // nó sẽ ngay lập tức gửi kết quả đến channel
    // ta chỉ lấy kết quả đầu tiên từ channel
    fmt.Println(<-ch)
}

```

Áp dụng ý tưởng trên có thể giúp cải thiện hiệu suất bằng cách chọn lấy kẻ chiến thắng trong cuộc đua thời gian.

1.6.9. Context package

Ở thời điểm phát hành Go1.7, thư viện tiêu chuẩn đã thêm một package context để đơn giản hóa hoạt động của dữ liệu, thời gian chờ và thoát giữa nhiều Goroutines. Package context định nghĩa kiểu Context, chứa deadline, cancelation signal và các giá trị request-scope giữa các API và giữa các process.

Chúng ta có thể sử dụng package context để hiện thực lại cơ chế kiểm soát timeout:

```
func worker(ctx context.Context, wg *sync.WaitGroup) error {
    defer wg.Done()

    for {
        select {
        default:
            fmt.Println("hello")
        case <-ctx.Done():
            return ctx.Err()
        }
    }
}

func main() {
    // nhận vào context parent (Background) và trả về context child (ctx) và hàm cancel
    // deadline 10 secs
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)

    var wg sync.WaitGroup

    // đơn giản hóa worker pool cho ngắn gọn
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go worker(ctx, &wg)
    }

    time.Sleep(time.Second)

    // mặc dù ctx sẽ expire theo timeout đã set trước đó
    // ta vẫn gọi cancel để đóng context child và các children của nó
    // để tránh giữ chúng tồn tại không cần thiết
    cancel()

    // sử dụng waitGroup thay cho done channel
    wg.Wait()
}
```

Golang tự động lấy lại bộ nhớ, do đó bộ nhớ thường không bị rò rỉ (memory leak). Trong ví dụ trước về sàng số nguyên tố, một Goroutine mới được đưa vào bên trong hàm `GenerateNatural` và Goroutine nền `PrimeFilter` có nguy cơ bị leak khi hàm `main` không còn sử dụng channel. Chúng ta có thể tránh vấn đề này với package context. Dưới đây là phần triển khai sàng số nguyên tố được cải thiện:

```
// trả về channel có chuỗi số: 2, 3, 4, ...
func GenerateNatural(ctx context.Context) chan int {
    ch := make(chan int)
    go func() {
        for i := 2; ; i++ {
            select {
            case <- ctx.Done():
                return
            case ch <- i:
            }
        }
    }()
    return ch
}
```

```

        }
    }()
    return ch
}

// bộ lọc: xóa các số có thể chia hết cho số nguyên tố
func PrimeFilter(ctx context.Context, in <-chan int, prime int) chan int {
    out := make(chan int)
    go func() {
        for {
            if i := <-in; i%prime != 0 {
                select {
                case <- ctx.Done():
                    return
                case out <- i:
                }
            }
        }
    }()
    return out
}

func main() {
    // Kiểm soát trạng thái Goroutine nền thông qua context
    ctx, cancel := context.WithCancel(context.Background())

    ch := GenerateNatural(ctx) // chuỗi số: 2, 3, 4, ...
    for i := 0; i < 100; i++ {
        prime := <-ch // số nguyên tố mới
        fmt.Printf("%v: %v\n", i+1, prime)
        ch = PrimeFilter(ctx, ch, prime) // Bộ lọc dựa trên số nguyên tố mới
    }

    cancel()
}

```

Khi hàm `main` kết thúc hoạt động, nó được thông báo bằng lệnh `cancel()` gọi đến Goroutine nền để thoát, do đó tránh khỏi việc leak Goroutine.

Concurrency là một chủ đề rất lớn, và ở đây chúng tôi chỉ đưa ra một vài ví dụ về lập trình concurrency rất cơ bản. Tài liệu chính thức cũng có rất nhiều cuộc thảo luận về lập trình concurrency, có khá nhiều cuốn sách thảo luận cụ thể về lập trình concurrency trong GoLang. Độc giả có thể tham khảo các tài liệu liên quan theo nhu cầu của mình.

Liên kết

- Phần tiếp theo: [Error và Exceptions](#)
- Phần trước: [Mô hình lập trình đồng thời và lập trình song song](#)
- [Mục lục](#)

1.7. Error và Exceptions

Error handling (xử lý lỗi) là một chủ đề quan trọng được đề cập trong mỗi ngôn ngữ lập trình. Go có một cơ chế xử lý lỗi đơn giản khác với `try catch` trên các ngôn ngữ lập trình khác dựa vào giá trị lỗi trả về của hàm. Ngoài ra, package `errors` giúp chúng ta định nghĩa các lỗi.

1.7.1. Ngữ cảnh thường gặp

Có một số hàm trong chương trình luôn yêu cầu phải chạy thành công. Ví dụ `strconv.Itoa` chuyển một số nguyên thành string, đọc và ghi phần tử từ array hoặc slice, đọc một phần tử tồn tại trong `map` và tương tự.

Những tác vụ như thế sẽ khó để có lỗi trong thời gian chạy trừ phi có `bug` trong chương trình hoặc những tình huống không thể đoán trước được như là memory leak tại thời điểm chạy. Nếu bạn thực sự bắt gặp một tình huống khác thường như thế, chương trình sẽ ngừng thực thi.

Khi một chương trình bị lỗi và dừng, chúng ta có thể cân nhắc một số khả năng xảy ra. Đối với các hàm xử lý lỗi tốt, nó sẽ trả về thêm một giá trị phụ, thường thì giá trị này được dùng để chứa thông điệp lỗi. Nếu chỉ có một lý do dẫn đến lỗi, giá trị thêm vào này có thể là một biến Boolean, thường đặt tên là `ok`. Ví dụ như bên dưới :

```
if v, ok := m["key"]; ok {
    return v
}
```

Nhưng thông thường sẽ có nhiều hơn một nguyên nhân gây ra lỗi, và nhiều khi user muốn biết nhiều thêm về lỗi đó. Nếu bạn chỉ sử dụng một biến boolean, thì bạn sẽ không giải quyết được yêu cầu trên. Trong ngôn ngữ C, một số nguyên `errno` được sử dụng mặc định để thể hiện lỗi, do đó bạn có thể định nghĩa nhiều loại error theo nhu cầu. Trong ngôn ngữ Go, có thể gọi `syscall.Errno` giống như với mã lỗi `errno` trong ngôn ngữ C.

Ví dụ, khi chúng ta dùng `syscall.Chmod` để thay đổi `mode` của một file, chúng ta có thể thấy thông tin về lỗi qua biến `err` như bên dưới :

```
err := syscall.Chmod(":invalid path:", 0666)
if err != nil {
    log.Fatal(err.(syscall.Errno))
}
```

Trong ngôn ngữ Go, `errors` được xem xét như là một kết quả đã được đoán trước; `exceptions` là một kết quả không thể đoán trước được, và một ngoại lệ có thể chỉ ra rằng một bug trong chương trình hoặc một vấn đề nào đó không được kiểm soát. Ngôn ngữ Go đề xuất dùng hàm `recover` để chuyển đổi exceptions thành error handling, chúng cho phép users thực sự quan tâm về những lỗi liên quan đến business.

Nếu một interface đơn giản ném tất cả những lỗi thông thường như là một ngoại lệ, chúng sẽ làm thông báo lỗi lộn xộn và không có giá trị. Như `main` trực tiếp bao gồm mọi thứ trong một hàm, nó không mang lại ý nghĩa gì.

```
func main() {
    defer func() {
        if r := recover(); r != nil {
            log.Fatal(r)
        }
    }()
}
```

Bao bọc một mã lỗi không phải là một kết quả cuối cùng. Nếu một ngoại lệ không thể đoán trước được, trực tiếp gây ra một ngoại lệ là một cách tốt nhất để xử lý chúng.

1.7.2. Chiến lược xử lý lỗi

Hãy minh họa cho ví dụ về sao chép file: một hàm cần phải mở hai file và sau đó sao chép toàn bộ nội dung của một file nào đó về một file khác.

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }

    written, err = io.Copy(dst, src)
    dst.Close()
    src.Close()
    return
}
```

Khi đoạn code trên chạy, sẽ tìm ẩn rủi ro. Nếu đầu tiên `os.open` gọi thành công, nhưng lệnh gọi thứ hai `os.create` gọi bị failed, nó sẽ trả về ngay lập tức mà không giải phóng tài nguyên file.

Mặc dù chúng ta có thể fix bug bằng việc gọi `src.Close()` trước lệnh `return` về mệnh đề `return` thứ hai; nhưng khi code trở nên phức tạp hơn, những vấn đề tương tự sẽ khó để tìm thấy và giải quyết. Chúng ta có thể sử dụng mệnh đề `defer` để đảm bảo rằng một file bình thường khi được mở cũng sẽ được đóng.

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }
    defer src.Close()

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }
    defer dst.Close()

    return io.Copy(dst, src)
}
```

Mệnh đề `defer` được thực thi khi ra khỏi tầm vực của hàm, chúng ta nghĩ về làm cách nào để đóng một file ngay khi mở file đó. Bất kể làm thế nào hàm được trả về, bởi vì mệnh đề `close` có thể luôn luôn được thực thi. Cùng một thời điểm, mệnh đề `defer` sẽ đảm bảo rằng `io.Copy` file có thể được đóng an toàn nếu một ngoại lệ xảy ra.

Như chúng ta đã đề cập trước đó, hàm `export` trong ngôn ngữ Go sẽ thông thường ném ra một ngoại lệ, và một ngoại lệ không được kiểm soát có thể xem là một bug trong một chương trình. Nhưng với những framework Web services, chúng thường cần sự truy cập từ bên thứ ba ở middleware.

Bởi vì thư viện middleware thứ ba có bug, khi mà một ngoại lệ ném một exception, web framework bản thân nó không chắc chắn. Để cải thiện sự bền vững của hệ thống, web framework thường thu hồi chính xác nhất có thể những ngoại lệ trong luồng thực thi của chương trình và sau đó sẽ gây exception về bằng cách `return error` thông thường.

Chúng ta hãy xem JSON parse là một ví dụ minh họa cho việc dùng ngữ cảnh của việc phục hồi. Cho một hệ thống JSON parser phức tạp, mặc dù một ngôn ngữ parse có thể làm việc một cách phù hợp, có một điều không chắc chắn rằng nó không có lỗi hỏng. Do đó, khi một ngoại lệ xảy ra, chúng ta sẽ không chọn cách crash parser. Thay vì thế chúng ta sẽ làm việc với ngoại lệ panic nhưng là một lỗi parsing thông thường và đính kèm chúng với một thông tin thêm để thông báo cho user biết mà báo cáo lỗi.

```
func ParseJSON(input string) (s *Syntax, err error) {
    defer func() {
        if p := recover(); p != nil {
            err = fmt.Errorf("JSON: internal error: %v", p)
        }
    }()
    // ...parser...
}
```

Gói `json` trong một thư viện chuẩn, nếu chúng gặp phải một error khi đệ quy parsing dữ liệu JSON bên trong, chúng sẽ nhanh chóng nhảy về mức cao nhất ở phía ngoài, và sau đó sẽ trả về thông điệp lỗi tương ứng.

Ngôn ngữ Go có cách hiện thực thư viện như vậy; mặc dù sử dụng package `panic`, chúng sẽ có thể được chuyển đổi đến một giá trị lỗi cụ thể khi một hàm được export.

1.7.3. Trường hợp dẫn đến lỗi sai

Thỉnh thoảng rất dễ cho những upper user hiểu rằng bên dưới sự hiện thực sẽ đóng gói lại error như là một loại error mới và trả kết quả về cho user.

```
if _, err := html.Parse(resp.Body); err != nil {
    return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
}
```

Khi một upper user bắt gặp một lỗi, nó có thể dễ dàng để hiểu rằng lỗi đó được gây ra trong thời gian chạy từ cấp business. Nhưng rất khó để có cả hai. Khi một upper user nhận được một sai sót mới, chúng ta cũng mất những error type bên dưới (chỉ những thông tin về mô tả sẽ bị mất).

Để ghi nhận thông tin về kiểu lỗi, chúng ta thông thường sẽ định nghĩa một hàm `WrapError` chúng bọc lấy lỗi gốc. Để tạo điều kiện cho những vấn đề như vậy, và để ghi nhận lại trạng thái của hàm khi một lỗi xảy ra, chúng ta sẽ muốn lưu trữ toàn bộ thông tin về hàm thực thi khi một lỗi xảy ra. Lúc này, để hỗ trợ transition như là RPC, chúng ta cần phải serialize error thành những dữ liệu tương tự như định dạng JSON, và sau đó khôi phục lại err từ việc decoding dữ liệu.

Để làm việc đó, chúng ta sẽ phải tự định nghĩa cấu trúc lỗi riêng ví dụ như github.com/chai2010/errors với những kiểu cơ bản sau:

```
type Error interface {
    Caller() []CallerInfo
    Wraped() []error
    Code() int
    error

    private()
}

type CallerInfo struct {
    FuncName string
    FileName string
    FileLine int
}
```

Trong số những `Error`, interface `error` là một mở rộng của kiểu `interface`, nó sẽ được dùng để thêm thông tin lời gọi hàm vào call stack, và hỗ trợ wrong muti-level gói lòng và hỗ trợ định dạng code. Cho tính dễ sử dụng, chúng ta có thể định nghĩa một số hàm giúp ích như sau:

```
func New(msg string) error
func NewWithCode(code int, msg string) error

func Wrap(err error, msg string) error
func WrapWithCode(code int, err error, msg string) error

func FromJson(json string) (Error, error)
func ToJson(err error) string
```

`New` dùng để tạo ra một loại error mới tương tự như `errors.New` trong thư viện chuẩn, nhưng với việc thêm vào thông tin gọi hàm tại thời điểm gây ra error. `FromJson` được dùng để khôi phục một kiểu đối tượng error từ chuỗi JSON. `NewWithCode` nó cũng gây dựng một error với một mã error, chúng cũng có thể chứa thông tin về gọi hàm call stack. `Wrap` và `WrapWithCode` là một hàm error secondary wrapper chúng sẽ gói error như là một error mới, nhưng sẽ giữ lại message error gốc. Đối tượng error sẽ trả về từ đây và có thể trực tiếp gọi `json.Marshal` để encode error như là JSON string.

Chúng ta có thể sử dụng wrapper function như sau:

```
import (
    "github.com/chai2010/errors"
)

func loadConfig() error {
    _, err := ioutil.ReadFile("/path/to/file")
    if err != nil {
        return errors.Wrap(err, "read failed")
    }

    // ...
}

func setup() error {
    err := loadConfig()
    if err != nil {
        return errors.Wrap(err, "invalid config")
    }

    // ...
}

func main() {
    if err := setup(); err != nil {
        log.Fatal(err)
    }

    // ...
}
```

Ở ví dụ trên, error sẽ được bao bọc trong hai lớp. chúng ta có thể duyệt quy trình đóng gói và bỏ qua.

```
for i, e := range err.(errors.Error).Wrapped() {
    fmt.Printf("wrapped(%d): %v\n", i, e)
}
```

Chúng ta có thể lấy thông tin gọi hàm cho mỗi wrapper error:

```
for i, x := range err.(errors.Error).Caller() {
```

```
    fmt.Printf("caller:%d: %s\n", i, x.FuncName)
}
```

Nếu chúng ta cần truyền một error thông qua network. chúng ta có thể encode `errors.ToJson(err)` như là JSON string.

```
// Gửi lỗi dưới dạng JSON
func sendError(ch chan<- string, err error) {
    ch <- errors.ToJson(err)
}

// Nhận lỗi dưới dạng JSON
func recvError(ch <-chan string) error {
    p, err := errors.FromJson(<-ch)
    if err != nil {
        log.Fatal(err)
    }
    return p
}
```

Cho web service dựa trên http protocol, chúng ta cũng có thể kết hợp trạng thái http với error.

```
err := errors.NewWithCode(404, "http error code")

fmt.Println(err)
fmt.Println(err.(errors.Error).Code())
```

Trong ngôn ngữ Go, error handling cũng có một coding style duy nhất. Sau khi kiểm tra nếu chức năng phụ bị failed, chúng ta thường đặt logic code tại sao sao chúng failed vào process trước khi code process thành công. Nếu một error gây ra function return, sau đó logic code về thành công sẽ không được đặt trên mệnh đề `else`, nó nên được đặt trực tiếp trong body của function.

```
f, err := os.Open("filename.ext")
if err != nil {
    // Trong trường hợp thất bại, trả về lỗi ngay lập tức
}

// Tiếp tục xử lý nếu không có lỗi
```

Cấu trúc code của hầu hết các hàm trong ngôn ngữ Go cũng tương tự, bắt đầu bởi một chuỗi khởi tạo việc kiểm tra để ngăn chặn lỗi xảy ra, theo sau bởi những logic thực sự trong function.

1.7.4. Trả về kết quả sai

Error trong ngôn ngữ Go là một kiểu interface. Thông tin về interface sẽ chứa kiểu dữ liệu nguyên mẫu, và kiểu dữ liệu gốc. Giá trị của interface chỉ tương ứng nếu như cả kiểu interface và giá trị gốc cả hai đều empty `nil`. Thực tế, khi kiểu của interface là empty, kiểu gốc sẽ tương ứng với interface sẽ không cần thiết phải empty.

Ví dụ sau, tôi thử cố gắng trả về một custom error type và trả về chỉ khi không có errors `nil`:

```
func returnsError() error {
    var p *MyError = nil
    if bad() {
        p = ErrBad
    }
    return p // Will always return a non-nil error.
}
```

Tuy nhiên, kết quả trả về cuối cùng sẽ thực sự không phải `nil`, nó là một lỗi thông thường, giá trị sai là `MyError` type của con trỏ `null`. Sau đây là một sự cải thiện của `returnsError` :

```
func returnsError() error {
    if bad() {
        return (*MyError)(err)
    }
    return nil
}
```

Do đó, khi đối mặt với giá trị `error` được return về, giá trị `error` return sẽ thích hợp khi được gán trực tiếp thành `nil`.

Ngôn ngữ Go sẽ có một kiểu dữ liệu mạnh, và cụ thể chuyển đổi sẽ được thực hiện giữa những kiểu khác nhau (và sẽ phải bên dưới cùng kiểu dữ liệu). Tuy nhiên, `interface` là một ngoại lệ của ngôn ngữ Go: non-interface kiểu đến kiểu `interface`, hoặc chuyển đổi từ `interface` type là cụ thể. Nó cũng sẽ hỗ trợ ducktype, dĩ nhiên, chúng sẽ thỏa mãn cấp độ 3 về bảo mật.

1.7.5. Phân tích ngoại lệ

`Panic` là một hàm dựng sẵn được dùng để dừng luồng thực thi thông thường và bắt đầu `panicking`. Khi hàm `F` gọi `panic`, hàm `F` sẽ dừng thực thi, bắt cứ hàm liên quan tới `F` sẽ thực thi một cách bình thường, và sau đó lệnh `return` `F` sẽ được gọi.

`Panic` được hỗ trợ để ném ra một kiểu ngoại lệ tùy ý (không chỉ là kiểu `error`), `recover` sẽ trả về một giá trị của lời gọi hàm và `panic` cũng như thông tin về kiểu tham số của hàm và những nguyên mẫu của hàm sẽ như sau:

```
func panic(interface{})
func recover() interface{}
```

Luồng thông thường trong ngôn ngữ Go là kết quả trả về của việc thực thi lệnh `return`. Đó không phải là một exception trong luồng, do đó luồng thực thi của ngoại lệ `recover` sẽ catch function trong process sẽ luôn luôn trả về `nil`. Cái khác là ngoại lệ exception. Khi một lời gọi `panic` sẽ ném ra một ngoại lệ, function sẽ kết thúc việc thực thi lệnh con, nhưng vì lời gọi `registered defer` sẽ vẫn được thực thi một cách bình thường và sau đó trả về `caller`. `Caller` trong hàm hiện tại, bởi vì trạng thái xử lý ngoại lệ chưa được bắt, `panic` sẽ tương tự như hành vi gọi hàm một cách trực tiếp. Khi một ngoại lệ xảy ra, nếu `defer` được thực thi lời gọi `recover`, nó có thể được bắt bằng việc trigger tham số `panic`, và trả về luồng thực thi bình thường.

`defer` sẽ thực hiện lệnh gọi `recover` nó thường gây khó khăn cho những người mới bắt đầu.

```
func main() {
    if r := recover(); r != nil {
        log.Fatal(r)
    }

    panic(123)

    if r := recover(); r != nil {
        log.Fatal(r)
    }
}
```

Cả hai lời gọi trên không có thể catch exceptions. Khi lời gọi `recover` đầu tiên được thực thi, hàm sẽ phải được trong một thứ tự thực thi bình thường, tại một điểm mà `recover` có thể trả về `nil`. Khi mà một exception xảy ra, lời gọi `recover` thứ hai sẽ không làm thay đổi việc thực thi, bởi vì lệnh gọi `panic` sẽ gây ra `defer` hàm sẽ trả về ngay lập tức sau khi thực thi `registered function`.

Trong thực tế, hàm `recover` sẽ có những yêu cầu nghiêm ngặt: chúng ta phải gọi lệnh `defer` để gọi chúng một cách trực tiếp từ hàm `recover`. Nếu hàm wrapper `defer` được gọi, `recover` sẽ catchup ngoại lệ sẽ bị fail. Ví dụ, thông thường chúng ta sẽ muốn gói hàm `MyRecover` và thêm những log cần thiết những thông tin bên trong, và sau đó gọi hàm `recover`. Đây là một hướng tiếp cận sai.

```
func main() {
    defer func() {
        // Không thể bắt ngoại lệ
        if r := MyRecover(); r != nil {
            fmt.Println(r)
        }
    }()
    panic(1)
}

func MyRecover() interface{} {
    log.Println("trace...")
    return recover()
}
```

Một cách tương tự, nếu chúng ta gọi `defer` trong hàm nested, `recover` sẽ cũng sẽ gây ra một ngoại lệ có thể được bắt.

```
func main() {
    defer func() {
        defer func() {
            // Không thể bắt ngoại lệ
            if r := recover(); r != nil {
                fmt.Println(r)
            }
        }()
    }()
    panic(1)
}
```

`defer` sẽ trực tiếp gọi two level nested function giống như wrapper function `recover` ở lớp một `defer` function. `MyRecover` là hàm trực tiếp trong statement sẽ work again.

```
func MyRecover() interface{} {
    return recover()
}

func main() {
    // có thể bắt ngoại lệ bình thường
    defer MyRecover()
    panic(1)
}
```

Tuy nhiên, nếu defer statement trực tiếp gọi hàm `recover`, thì ngoại lệ sẽ không được bắt một cách phù hợp.

```
func main(){
    defer recover()
    panic(1)
}
```

Nó sẽ phải tách biệt từ stack frame với một ngoại lệ bởi stack frame, do đó hàm `recover` sẽ có thể ném một ngoại lệ một cách bình thường. Hay nói cách khác, hàm `recover` sẽ bắt ngoại lệ của mức trên gọi hàm stack frame (chỉ là một layer `defer` function).

Đĩ nhiên, để tránh việc gọi `recover` không nhận ra được ngoại lệ, chúng ta nên tránh ném ra ngoại lệ `nil` như là một tham số.

```
func main() {
    defer func() {
        if r := recover(); r != nil { ... }
    }()
    panic(nil)
}
```

Khi chúng ta muốn trả về việc ném ngoại lệ vào error, nếu chúng ta muốn trung thành trả về thông tin gốc, bạn sẽ phải cần sử lý chúng một cách rời rạc cho những kiểu khác nhau.

```
func foo() (err error) {
    defer func() {
        if r := recover(); r != nil {
            switch x := r.(type) {
            case string:
                err = errors.New(x)
            case error:
                err = x
            default:
                err = fmt.Errorf("Unknown panic: %v", r)
            }
        }()
        panic("TODO")
    }
}
```

Dựa trên mẫu code trên, chúng ta có thể mô phỏng nhiều kiểu exception. Bởi việc định nghĩa những kiểu khác nhau của việc bảo vệ interface, chúng ta có thể phân biệt kiểu của ngoại lệ.

```
func main {
    defer func() {
        if r := recover(); r != nil {
            switch x := r.(type) {
            case runtime.Error:
                // ngoại lệ do quá trình chạy
            case error:
                // ngoại lệ do lỗi thông thường
            default:
                // ngoại lệ khác
            }
        }()
        // ...
    }
}
```

Nhưng làm như vậy sẽ đi ngược lại với triết lý lập trình đơn giản và dễ hiểu của Go.

Liên kết

- Phần tiếp theo: [Chương 2](#)
- Phần trước: [Mô hình thực thi đồng thời](#)
- [Mục lục](#)

Chapter 2: CGO Programming



CGO

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off
(Bjarne Stroustrup)

Sau nhiều thập kỷ phát triển, với số lượng lớn phần mềm được viết bằng C/C++, nhiều trong số đó được kiểm thử và tối ưu hiệu năng. Ngôn ngữ Go nên tận dụng ưu thế to lớn đó của C/C++.

Go hỗ trợ các lời gọi hàm từ C thông qua một công cụ gọi là `cgo`, do đó ta có thể sử dụng Go để đưa thư viện động của C (C dynamic library) sang các ngôn ngữ khác. Chương này sẽ đi sâu vào tìm hiểu các vấn đề liên quan đến lập trình với CGO.

Tuy nhiên ta cũng không nên lạm dụng CGO vì vấn đề hiệu suất, ví dụ nếu so sánh với Rust: `rustgo` chỉ chậm hơn gọi hàm Go trực tiếp khoảng 11%, nhưng nó lại nhanh hơn gấp 15 lần `cgo`, một vài con số cụ thể từ [link sau](#):

name	time/op
CallOverhead/Inline	1.67ns ± 2%
CallOverhead/Go	4.49ns ± 3%
CallOverhead/rustgo	4.58ns ± 3%
CallOverhead/cgo	69.4ns ± 0%

Liên kết

- Phần tiếp theo: [Quick Start](#)
- Phần trước: [Chương 1: Lời nói thêm](#)
- [Mục lục](#)

2.1. Quick Start

Trong phần này, chúng ta sẽ tìm hiểu cách sử dụng CGO cơ bản thông qua loạt ví dụ từ đơn giản đến phức tạp.

2.1.0. Cài đặt

Linux, Mac

- CGO được tích hợp sẵn, người dùng Linux và Mac chỉ cần cài đặt Go.

Windows

- CGO không tương thích với Cygwin
- Để dùng được CGO bạn phải cài đặt MingGW và sử dụng cmd của Windows

2.1.1. Chương trình CGO đơn giản

Đầu tiên là một chương trình CGO đơn giản nhất:

```
//main.go
package main

import "C"

func main() {
    println("hello cgo")
}
```

Chúng ta import package CGO thông qua câu lệnh `import "c"`. Chương trình trên chưa thực hiện bất kì thao tác nào với CGO, chỉ mới thông báo sẵn sàng cho việc lập trình với CGO. Mặc dù chưa sử dụng gì đến CGO nhưng lệnh `go build` vẫn sẽ gọi trình biên dịch `gcc` trong suốt quá trình biên dịch do đây được là một chương trình CGO hoàn chỉnh.

2.1.2. Xuất chuỗi dựa trên thư viện chuẩn của C

```
//main.go
package main

//#include <stdio.h>
import "C"

func main() {
    C.puts(C.CString("Hello World\n"))
}

// kết quả:
// Hello World
```

`import package "c"` để thực hiện các chức năng của CGO và include thư viện của ngôn ngữ C (trong comment code là cú pháp của CGO, không phải phần comment như thông thường). Tiếp theo, chuỗi string trong `c.cstring` của ngôn ngữ Go được chuyển đổi thành chuỗi string trong ngôn ngữ C bằng phương thức `c.puts` của package CGO.

Việc lỗi xảy ra khi không giải phóng chuỗi được tạo bằng C.CString của ngôn ngữ C sẽ dẫn đến rò rỉ bộ nhớ (memory leak). Nhưng đối với chương trình nhỏ ở trên điều này không đáng lo ngại vì hệ điều hành sẽ tự động lấy lại các tài nguyên của chương trình sau khi chương trình kết thúc.

2.1.3. Sử dụng hàm C tự khai báo

Phần trên chúng ta đã sử dụng các hàm đã có trong `stdio`. Nay giờ ta sẽ sử dụng một hàm `SayHello` của ngôn ngữ C. Chức năng hàm này là in ra chuỗi chúng ta truyền vào hàm. Sau đó gọi hàm `SayHello` trong hàm main:

```
//main.go
package main

/*
#include <stdio.h>
// Khai báo hàm trong ngôn ngữ C
static void SayHello(const char* s) {
    puts(s);
}
*/
import "C"

func main() {
    C.SayHello(C.CString("Hello World\n"))
}
```

Hoặc có thể đặt hàm `SayHello` trong file `hello.c` như sau:

```
//hello.c
#include <stdio.h>

void SayHello(const char* s) {
    puts(s);
}
```

Sau đó bên file `main.go` chúng chỉ cần khai báo hàm `SayHello` trong phần CGO như bên dưới.

```
//main.go
package main

//void SayHello(const char* s);
import "C"

func main() {
    C.SayHello(C.CString("Hello World\n"))
}
```

Hiện tại thư mục làm việc có cấu trúc như sau:

```
.
├── hello.c
└── main.go
```

Lưu ý : thay vì chạy lệnh `go run main.go` hoặc `go build main.go`, chúng ta phải sử dụng `go run "tên/của/package"` hoặc `go build "tên/của/package"`. Nếu đang đứng trong thư mục chứa mã nguồn thì bạn có thể chạy chương trình bằng lệnh `go run .` hoặc `go build .`

2.1.4. Module hóa C code

Trừu tượng và module hóa là cách để đơn giản hóa các vấn đề trong lập trình:

- Khi code quá dài, ta có thể đưa các lệnh tương tự nhau vào chung một hàm.
- Khi có nhiều hàm hơn, ta chia chúng vào các file hoặc module.

Trong ví dụ trước, ta trừu tượng hóa một module tên là `hello` và tất cả các interface của module đó được khai báo trong file header `hello.h`:

```
//hello.h
void SayHello(const char* s);
```

Và hiện thực hàm `SayHello` trong file `hello.c`:

```
//hello.c
#include "hello.h"
#include <stdio.h>

// Đảm bảo việc hiện thực hàm thỏa mãn interface của module.
void SayHello(const char* s) {
    puts(s);
}
```

Ngoài ra ta có thể hiện thực hàm này bằng C++ cũng được:

```
//hello.cpp
#include <iostream>

// extern giúp function C++ có được các liên kết (linkage)
// của C. Chỉ ra rằng liên hệ giữa hello.cpp và hello.h
// vẫn theo quy tắc của C
extern "C" {
    #include "hello.h"
}

void SayHello(const char* s) {
    std::cout << s;
}
```

Trong hàm main của Go ta gọi file header như sau:

```
//main.go
package main

//#include <hello.h>
import "C"

func main() {
    C.SayHello(C.CString("Hello World"))
}
```

Với việc lập trình C thông qua interface, ta có thể hiện thực module bằng nhiều ngôn ngữ khác nhau, miễn là đáp ứng được interface: SayHello có thể được viết bằng C, C++, Go hoặc kể cả Assembly.

2.1.5. Sử dụng Go để hiện thực hàm trong C

Trong thực tế, CGO không chỉ được sử dụng để gọi các hàm của C bằng ngôn ngữ Go mà còn có thể cho phép C gọi các hàm Go trong code của mình.

Trong ví dụ trước, chúng ta đã trùu tượng hóa một module có tên hello và tất cả các chức năng interface của module được xác định trong file header `hello.h`:

```
//hello.h
void SayHello(const char* s);
```

Bây giờ, chúng ta tạo một file `hello.go` và hiện thực lại hàm `SayHello` của interface bằng ngôn ngữ Go:

```
//hello.go
package main

import "C"

import "fmt"

//export SayHello
func SayHello(s *C.char) {
    fmt.Println(C.GoString(s))
}
```

Sử dụng chỉ thị `//export SayHello` của CGO để export hàm được hiện thực bằng Go sang hàm sử dụng được cho C.

Cần chú ý là ta sẽ có hai phiên bản `SayHello`: một là trong môi trường cục bộ của Go, hai là của C. Phiên bản `SayHello` của C được sinh ra bởi CGO cuối cùng cũng sẽ gọi phiên bản `SayHello` của Go thông qua `bridge code`.

Với việc lập trình ngôn ngữ C qua interface, ta có thể tự do hiện thực và đơn giản hóa việc sử dụng hàm. Bây giờ ta có thể dùng `SayHello` như là một thư viện:

```
package main

//#include <hello.h>
import "C"

func main() {
    C.SayHello(C.CString("Hello World\n"))
}
```

Cấu trúc thư mục lúc này:

```
.
├── hello.go
└── hello.h
└── main.go
```

2.1.6. Sử dụng Go để lập trình interface cho C

Để cho đơn giản chúng ta sẽ gộp tất cả thành một file `main.go` duy nhất như ví dụ dưới đây.

```
//main.go
package main

//void SayHello(char* s);
import "C"

import (
    "fmt"
)
```

```

func main() {
    C.SayHello(C.CString("Hello World\n"))
}

//export SayHello
func SayHello(s *C.char) {
    fmt.Println(C.GoString(s))
}

```

Tỉ lệ code C trong chương trình bây giờ ít hơn. Tuy nhiên vẫn phải sử dụng chuỗi trong C thông qua hàm `C.CString` chứ không thể dùng trực tiếp chuỗi của Go. Trong `Go1.10`, CGO đã thêm kiểu `_GoString_Prot` để thể hiện chuỗi trong ngôn ngữ Go. Sau đây là code đã được cải tiến:

```

// +build go1.10

package main

//void SayHello(_GoString_ s);
import "C"

import (
    "fmt"
)

func main() {
    C.SayHello("Hello World\n")
}

//export SayHello
func SayHello(s string) {
    fmt.Println(s)
}

```

Có vẻ như tất cả đều được viết bằng Go, nhưng việc triển khai từ hàm `main()` của ngôn ngữ Go đến phiên bản ngôn ngữ C đã tự động tạo ra hàm `SayHello`, rồi cuối cùng trở lại môi trường ngôn ngữ Go.

Liên kết

- Phần tiếp theo: [CGO Foundation](#)
- Phần trước: [Chương 1](#)
- [Mục lục](#)

2.2. CGO Foundation

Để sử dụng tính năng CGO, bạn cần cài đặt compiler C/C++ trên macOS và Linux là `gcc` còn trên Windows là `MinGW`. Đồng thời, cần đảm bảo rằng biến môi trường `CGO_ENABLED` được đặt thành 1.

2.2.1. Lệnh `import "C"`

Lệnh `import "C"` xuất hiện trong code nói cho compiler rằng tính năng CGO sẽ được sử dụng. Code trong cặp `/* */` trước lệnh đó là cú pháp để Go nhận ra code của C. Lúc này, ta có thể thêm các file code của C/C++ tương ứng trong thư mục hiện tại.

Ví dụ:

```
package main

/*
#include <stdio.h>

void printint(int v) {
    printf("printint: %d\n", v);
}
*/
import "C"

func main() {
    v := 42

    // int(v) chuyển đổi giá trị kiểu int trong Go đến giá trị
    // kiểu int trong ngôn ngữ C
    C.printint(C.int(v))
}
```

Tất cả các thành phần ngôn ngữ C trong file header sau khi được include sẽ được thêm vào package "C" ảo. Cần lưu ý rằng câu lệnh `import "C"` yêu cầu một dòng riêng và không thể được import cùng với các package khác.

Cần lưu ý rằng Go là ngôn ngữ ràng buộc kiểu mạnh, do đó tham số được truyền phải đúng kiểu khai báo, và phải được chuyển đổi sang kiểu trong C bằng các hàm chuyển đổi trước khi truyền, không thể truyền trực tiếp bằng kiểu của Go.

Các ký hiệu của C được import thông qua package C thì *không cần phải viết hoa*, không cần phải tuân theo quy tắc của Go.

Ví dụ tiếp theo ta định nghĩa kiểu `cchar` tương ứng với con trỏ char của C trong Go và sau đó thêm phương thức `GoString` để trả về chuỗi trong ngôn ngữ Go:

```
package cgo_helper

//#include <stdio.h>
import "C"

type CChar C.char

// nhận vào CChar của C và trả về
// chuỗi string của Go
func (p *CChar) GoString() string {
    return C.GoString((*C.char)(p))
}
```

```
func PrintCString(cs *C.char) {
    C.puts(cs)
}
```

Bây giờ có thể ta muốn sử dụng hàm này trong các package Go khác:

```
package main

//static const char* cs = "hello";
import "C"
import "./cgo_helper"

func main() {

    // C.cs là kiểu của package C xây dựng
    // trên kiểu *char (thực ra là *main.C.char)
    cgo_helper.PrintCString(C.cs)
    // kiểu *C.type của hàm PrintCString trong
    // cgo_helper là *cgo_helper.C.char
    // 2 kiểu này không giống nhau
    // --> Code lỗi
}
```

Các tham số được chuyển đổi kiểu rồi mới truyền vào có được không?

Câu trả lời là không, bởi vì các tham số của `cgo_helper.PrintCString` là kiểu `*c.char` được định nghĩa trong package riêng của nó và không thể truy cập trực tiếp từ bên ngoài.

Nói cách khác, nếu một package sử dụng trực tiếp một kiểu thuộc package C ảo (tương tự như `*c.char`) trong một interface chung thì các package khác sẽ không thể sử dụng trực tiếp các kiểu đó trừ khi package kia cũng cung cấp hàm tạo `*c.char type`.

2.2.2. Lệnh `#cgo`

Trước dòng `import "C"` ta có thể đặt các tham số cho quá trình biên dịch (compile phase) và quá trình liên kết (link phase) thông qua các lệnh `#cgo`.

- Các tham số của quá trình biên dịch chủ yếu được sử dụng để xác định các macro liên quan và đường dẫn truy xuất file header đã chỉ định.
- Các tham số của quá trình liên kết chủ yếu là để xác định đường dẫn truy xuất file thư viện và file thư viện sẽ được liên kết.

```
// Định nghĩa macro PNG_DEBUG, giá trị là 1
// #cgo CFLAGS: -DPNG_DEBUG=1 -I./include
// #cgo LDFLAGS: -L/usr/local/lib -lpng
// #include <png.h>
import "C"
```

Trong đoạn mã trên:

- Phần `CFLAGS`: `-D` định nghĩa macro `PNG_DEBUG`, giá trị là 1
- `-I` xác định thư mục tìm kiếm có trong file header.
- Phần `LDFLAGS`: `-L` chỉ ra thư mục truy xuất các file thư viện, `-l` chỉ định thư viện png là bắt buộc.

Do các vấn đề mà C/C ++ để lại, đường dẫn truy xuất file header C có thể là *relative path*, nhưng đường dẫn truy xuất file thư viện bắt buộc phải là *absolute path*. Absolute path của thư mục package hiện tại có thể được biểu diễn bằng biến `${SRCDIR}` trong thư mục truy xuất các file thư viện:

```
// #cgo LDFLAGS: -L${SRCDIR}/libs -lfoo
```

Đoạn code trên sẽ được phân giải trong link phase và trở thành:

```
// #cgo LDFLAGS: -L/go/src/foo/libs -lfoo
```

Lệnh `#cgo` chủ yếu ảnh hưởng đến một số biến môi trường của trình biên dịch như `CFLAGS`, `CPPFLAGS`, `CXXFLAGS`, `FFLAGS` và `LDFLAGS`. `LDFLAGS` được sử dụng để đặt tham số của liên kết, `CFLAGS` được sử dụng để đặt tham số biên dịch cho code ngôn ngữ C.

Đối với người dùng sử dụng C và C++ trong môi trường CGO, có thể có ba tùy chọn biên dịch khác nhau:

- `CFLAGS` cho các tùy chọn biên dịch theo ngôn ngữ C.
- `CPPFLAGS` cho các tùy chọn biên dịch cụ thể C++.
- `CXXFLAGS` cho các biên dịch C và C++.

Các lệnh `#cgo` cũng hỗ trợ tùy chọn biên dịch hoặc liên kết với các hệ điều hành hoặc một kiểu kiến trúc CPU khác nhau:

```
// tùy chọn cho Windows
// #cgo windows CFLAGS: -DX86=1

// tùy chọn cho non-windows platforms
// #cgo !windows LDFLAGS: -lm
```

Một ví dụ để xác định hệ thống nào đang chạy CGO:

```
package main

/*
#cgo windows CFLAGS: -DCGO_OS_WINDOWS=1
#cgo darwin CFLAGS: -DCGO_OS_DARWIN=1
#cgo linux CFLAGS: -DCGO_OS_LINUX=1

#if defined(CGO_OS_WINDOWS)
    const char* os = "windows";
#elif defined(CGO_OS_DARWIN)
    const char* os = "darwin";
#elif defined(CGO_OS_LINUX)
    const char* os = "linux";
#else
#    error(unknown os)
#endif
*/
import "C"

func main() {
    print(C.GoString(C.os))
}
```

Bằng cách này, chúng ta có thể biết được hệ thống mà code đang vận hành, nhờ đó áp dụng các kĩ thuật riêng cho các nền tảng khác nhau.

2.2.3. Biên dịch với tag

Build tag là một comment đặc biệt ở đầu file C/C++ trong môi trường Go hoặc CGO. Biên dịch có điều kiện tương tự như sử dụng macro `#cgo` để xác định các nền tảng khác nhau (ví dụ trên). Code được build sau khi macro của nền tảng tương ứng được xác định.

Ví dụ trình bày một cách khác khi các file nguồn sau sẽ chỉ được tạo khi debug build flag được thiết lập:

```
// +build debug

package main

var buildMode = "debug"
```

Có thể được build bằng lệnh sau:

```
go build -tags="debug"
go build -tags="windows debug"
```

Chúng ta có thể dùng `-tags` chỉ định nhiều build flag cùng một lúc thông qua các đối số dòng lệnh.

Ví dụ các build flag sau chỉ ra rằng việc build chỉ được thực hiện trong kiến trúc "linux/386" hoặc "non-cgo environment" trong nền tảng darwin.

```
// +build linux,386 darwin,!cgo
```

Trong đó, dấu phẩy (`,`) nghĩa là **và**. Khoảng trắng () nghĩa là **hoặc**.

Liên kết

- Phần tiếp theo: [Chuyển đổi kiểu dữ liệu](#)
- Phần trước: [Quick Start](#)
- [Mục lục](#)

2.3. Chuyển đổi kiểu dữ liệu

Ban đầu, CGO được tạo ra để thuận lợi cho việc sử dụng các hàm trong C (các hàm hiện thực khai báo Golang trong C) để sử dụng lại các tài nguyên của C. Ngày nay, CGO đã phát triển thành cầu nối giao tiếp hai chiều giữa C và Go. Để tận dụng tính năng của CGO, việc hiểu các quy tắc chuyển đổi kiểu giữa hai loại ngôn ngữ là điều quan trọng.

2.3.1. Các kiểu dữ liệu số học

Khi ta sử dụng các ký hiệu của C trong Golang, thường nó sẽ truy cập thông qua package "C" ào, chẳng hạn như kiểu `int` tương ứng với `c.int`. Một số kiểu trong C bao gồm nhiều từ khóa, nhưng khi truy cập chúng thông qua package "C" ào, phần tên không thể có ký tự khoảng trắng, ví dụ `unsigned int` không thể truy cập bằng `c.unsigned int`. Do đó, CGO cung cấp quy tắc chuyển đổi tương ứng cho các kiểu trong C:

Kiểu trong C	Kiểu trong CGO	Kiểu trong Go
char	C.char	byte
singed char	C.schar	int8
unsigned char	C.uchar	uint8
short	C.short	int16
unsigned short	C.ushort	uint16
int	C.int	int32
unsigned int	C.uint	uint32
long	C.long	int32
unsigned long	C.ulong	uint32
long long int	C.longlong	int64
unsigned long long int	C.ulonglong	uint64
float	C.float	float32
double	C.double	float64
size_t	C.size_t	uint

Bảng so sánh kiểu trong các ngôn ngữ Go và C

Mặc dù kích thước của những kiểu không chỉ rõ kích thước (trong C) như `int`, `short`, ..., kích thước của chúng đều được xác định trong CGO: kiểu `int` và `uint` của C đều có kích thước 4 byte, kiểu `size_t` có thể được coi là kiểu số nguyên không dấu `uint` của ngôn ngữ Go.

Mặc dù kiểu `int` và `uint` của C đều có kích thước cố định, nhưng với Go thì `int` và `uint` có thể là 4 byte hoặc 8 byte (tùy platform). Nếu cần sử dụng đúng kiểu `int` của C trong Go, bạn có thể sử dụng kiểu `goInt` được xác định trong file header `_cgo_export.h` được tạo ra bởi công cụ CGO. Trong file header này, mỗi kiểu giá trị cơ bản của Go sẽ xác định kiểu tương ứng trong C (kiểu có tiền tố "Go"). Ví dụ sau trong hệ thống 64-bit, file header `_cgo_export.h` định nghĩa các kiểu giá trị:

```

typedef signed char GoInt8;
typedef unsigned char GoUint8;
typedef short GoInt16;
typedef unsigned short GoUint16;
typedef int GoInt32;
typedef unsigned int GoUint32;
typedef long long GoInt64;
typedef unsigned long long GoUint64;
typedef GoInt64 GoInt;
typedef GoUint64 GoUint;
typedef float GoFloat32;
typedef double GoFloat64;

```

Trừ `GoInt` và `GoUint`, chúng tôi không khuyến khích bạn sử dụng trực tiếp `GoInt32`, `GoInt64` và các kiểu khác.

Một cách tốt hơn là sử dụng các kiểu có trong khai báo file header (chuẩn C99):

Kiểu trong C	Kiểu trong CGO	Kiểu trong Go
<code>int8_t</code>	<code>C.int8_t</code>	<code>int8</code>
<code>uint8_t</code>	<code>C.uint8_t</code>	<code>uint8</code>
<code>int16_t</code>	<code>C.int16_t</code>	<code>int16</code>
<code>uint16_t</code>	<code>C.uint16_t</code>	<code>uint16</code>
<code>int32_t</code>	<code>C.int32_t</code>	<code>int32</code>
<code>uint32_t</code>	<code>C.uint32_t</code>	<code>uint32</code>
<code>int64_t</code>	<code>C.int64_t</code>	<code>int64</code>
<code>uint64_t</code>	<code>C.uint64_t</code>	<code>uint64</code>

Bảng so sánh kiểu trong `stdint.h`

Như đã đề cập trước đó, nếu kiểu trong C bao gồm nhiều từ, nó không thể được sử dụng trực tiếp thông qua package "C" ào (ví dụ: `unsigned short` không thể được truy cập trực tiếp `C.unsigned short`). Tuy nhiên, sau khi định nghĩa lại kiểu trong bằng cách sử dụng `typedef`, chúng ta có thể truy cập tới kiểu gốc. Đối với các kiểu trong C phức tạp hơn thì nên sử dụng `typedef` để đặt lại tên cho nó, thuận tiện cho việc truy cập từ CGO.

2.3.2. Go Strings và Slices

Trong file header `_cgo_export.h` được tạo ra bởi CGO, kiểu trong C tương ứng cũng được tạo cho Go string, slice, dictionary, interface và channel:

```

typedef struct { const char *p; GoInt n; } GoString;
typedef void *GoMap;
typedef void *GoChan;
typedef struct { void *t; void *v; } GoInterface;
typedef struct { void *data; GoInt len; GoInt cap; } GoSlice;

```

Tuy nhiên, cần lưu ý rằng chỉ các string và slice là có giá trị sử dụng trong CGO, vì CGO tạo ra các phiên bản ngôn ngữ C cho một số hàm trong Go, vì vậy cả hai đều có thể gọi các hàm C trong Go, điều này được thực hiện ngay lập tức và CGO không cung cấp các hàm hỗ trợ liên quan cho các kiểu khác, đồng thời mô hình bộ nhớ dành riêng cho ngôn ngữ Go ngăn chúng ta duy trì các kiểu con trả về các vùng bộ nhớ Go quản lý, vì vậy mà môi trường ngôn ngữ C của các kiểu đó không có giá trị sử dụng.

Trong hàm C đã export, chúng ta có thể trực tiếp sử dụng các string và slice trong Go. Giả sử có hai hàm export sau:

```
//export helloString
func helloString(s string) {}

//export helloSlice
func helloSlice(s []byte) {}
```

File header `_cgo_export.h` được CGO tạo ra sẽ chứa khai báo hàm sau:

```
extern void helloString(GoString p0);
extern void helloSlice(GoSlice p0);
```

Nhưng lưu ý rằng nếu bạn sử dụng kiểu `GoString` thì sẽ phụ thuộc vào file header `_cgo_export.h` và file này có nội dung hay thay đổi do CGO sinh ra.

Phiên bản Go1.10 thêm một chuỗi kiểu `_GoString_`, có thể làm giảm code có rủi ro phụ thuộc file header `_cgo_export.h`. Chúng ta có thể điều chỉnh khai báo ngôn ngữ C của hàm `helloString` thành:

```
extern void helloString(_GoString_ p0);
```

Bởi vì `_GoString_` là kiểu định nghĩa trước, ta không thể truy cập rực tiếp các thông tin như length hay pointer của string qua kiểu này. Go1.10 thêm vào 2 hàm sau để bổ sung:

```
size_t _GoStringLen(_GoString_ s);
const char *_GoStringPtr(_GoString_ s);
```

2.3.3. Struct, Union, Enum

Các kiểu struct, Union và Enum của ngôn ngữ C không thể được thêm vào struct dưới dạng thuộc tính ẩn danh.

Struct

Trong Go, chúng ta có thể truy cập các kiểu struct như `struct xxx` tương ứng là `c.struct_xxx` trong ngôn ngữ C. Tổ chức bộ nhớ của struct tuân theo các quy tắc alignment. Trong môi trường ngôn ngữ Go 32 bit, struct của C tuân theo quy tắc alignment 32 bit và môi trường ngôn ngữ Go 64 bit tuân theo quy tắc alignment 64 bit. Đối với các struct có quy tắc alignment đặc biệt được chỉ định, chúng không thể được truy cập trong CGO.

Cách sử dụng struct đơn giản như sau:

```
/*
struct A {
    int i;
    float f;
};

import "C"
import "fmt"

func main() {
    var a C.struct_A
    fmt.Println(a.i)
    fmt.Println(a.f)
}
```

Nếu tên thành phần của struct tĩnh cờ là một từ khóa trong Go, bạn có thể truy cập nó bằng cách thêm một dấu gạch dưới ở đầu tên thành viên:

```
/*
struct A {
    int type;
    // type là một từ khóa trong Golang
};

import "C"
import "fmt"

func main() {
    var a C.struct_A
    fmt.Println(a._type)
    // _type tương ứng với type
}
```

Nhưng nếu có 2 thành phần: một thành phần được đặt tên theo từ khóa của Go và phần kia là trùng khi thêm vào dấu gạch dưới, thì các thành phần được đặt tên theo từ khóa ngôn ngữ Go sẽ không thể truy cập:

```
/*
struct A {
    int type; // type là một từ khóa trong Go
    float _type; // chặn CGO truy cập type trên kia
};

import "C"
import "fmt"

func main() {
    var a C.struct_A
    fmt.Println(a._type) // _type tương ứng với _type
}
```

Các thành phần tương ứng với **trường bit** (thuộc tính được định nghĩa với giá trị độ lớn kèm theo) trong cấu trúc ngôn ngữ C không thể được truy cập bằng ngôn ngữ Go. Nếu bạn cần thao tác với các thành phần này, bạn cần định nghĩa hàm hỗ trợ trong C.

```
/*
struct A {
    int size: 10; // Trường bit không thể truy cập
    float arr[]; // Mảng có độ dài bằng 0 cũng không thể truy cập được
};

import "C"
import "fmt"

func main() {
    var a C.struct_A
    fmt.Println(a.size) // Lỗi không thể truy cập trường bit
    fmt.Println(a.arr) // Lỗi mảng có độ dài bằng 0
}
```

Trong ngôn ngữ C, chúng ta không thể truy cập trực tiếp vào kiểu struct được định nghĩa bởi Go.

Union

Đối với các kiểu union, chúng ta có thể truy cập các kiểu `union xxx` tương ứng là `c.union_xxx` trong ngôn ngữ C. Tuy nhiên, các kiểu union trong C không được hỗ trợ trong Go và chúng được chuyển đổi thành các mảng byte có kích thước tương ứng.

```

/*
#include <stdint.h>

union B1 {
    int i;
    float f;
};

union B2 {
    int8_t i8;
    int64_t i64;
};
*/
import "C"
import "fmt"

func main() {
    var b1 C.union_B1;
    fmt.Printf("%T\n", b1) // [4]uint8

    var b2 C.union_B2;
    fmt.Printf("%T\n", b2) // [8]uint8
}

```

Nếu bạn cần thao tác biến kiểu lồng nhau trong C (union):

- Cách thứ nhất là định nghĩa hàm hỗ trợ trong C,
- Cách thứ hai là phân giải thủ công các thành phần đó thông qua "encoding/binary" của ngôn ngữ Go (không phải vấn đề big endian),
- Cách thứ ba là sử dụng package `unsafe` để chuyển sang kiểu tương ứng (đây là cách tốt nhất để thực hiện).

Sau đây cho thấy cách truy cập các thành viên kiểu union thông qua package `unsafe`:

```

/*
#include <stdint.h>

union B {
    int i;
    float f;
};
*/
import "C"
import "fmt"

func main() {
    var b C.union_B;
    fmt.Println("b.i:", *(*C.int)(unsafe.Pointer(&b)))
    fmt.Println("b.f:", *(*C.float)(unsafe.Pointer(&b)))
}

```

Mặc dù truy cập bằng package `unsafe` là cách dễ nhất và tốt nhất về hiệu suất, nó có thể làm phức tạp hóa vấn đề với các tình huống mà trong đó các kiểu union lồng nhau được xử lý. Đối với các kiểu này ta nên xử lý chúng bằng cách định nghĩa các hàm hỗ trợ trong C.

Enum

Đối với các kiểu liệt kê (enum), chúng ta có thể truy cập các kiểu `enum xxx` tương ứng là `c.enum_xxx` trong C.

```

/*
enum C {
    ONE,
    TWO,
}

```

```

};

*/
import "C"
import "fmt"

func main() {
    var c C.enum_C = C.TWO
    fmt.Println(c)
    fmt.Println(C.ONE)
    fmt.Println(C.TWO)
}

```

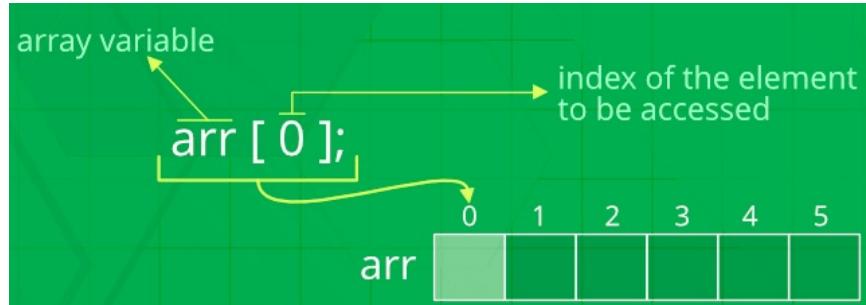
Trong ngôn ngữ C, kiểu `int` bên dưới kiểu liệt kê hỗ trợ giá trị âm. Chúng ta có thể truy cập trực tiếp các giá trị liệt kê được xác định bằng `C.ONE`, `C.TWO`,

2.3.4. Array, String và Slice

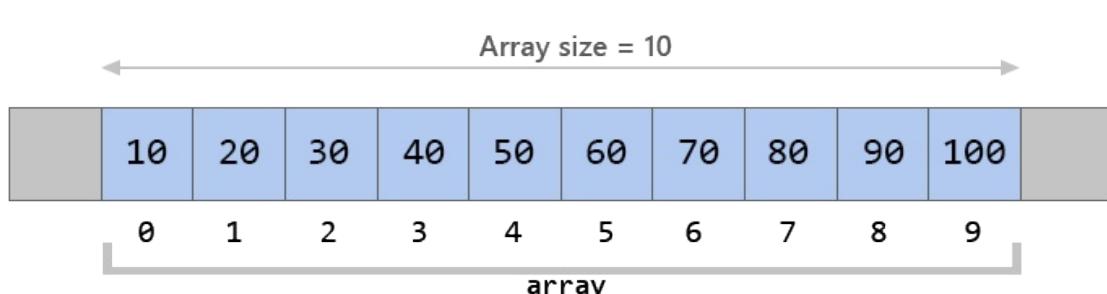
Chuỗi (string) trong C là một mảng kiểu char và độ dài của nó phải được xác định theo vị trí của ký tự NULL (đại diện kết thúc mảng). Không có kiểu slice trong ngôn ngữ C.

Array

Trong C, biến mảng thực ra tương ứng với một con trỏ trỏ tới một phần bộ nhớ có độ dài cụ thể của một kiểu cụ thể, con trỏ này không thể được sửa đổi, khi truyền biến mảng vào một hàm, thực ra là truyền địa chỉ phần tử đầu tiên của mảng.



Trong Go, mảng là một kiểu giá trị và độ dài của mảng là một phần của kiểu mảng. Chuỗi trong Go tương ứng với một vùng nhớ "chỉ đọc" có độ dài nhất định. Slice trong Go là phiên bản đơn giản hơn của mảng động (dynamic array).



Chuyển đổi giữa Go và C với các kiểu array, string và slice có thể được đơn giản hóa thành chuyển đổi giữa Go slice và C pointer trỏ tới vùng nhớ có độ dài nhất định.

Package C ảo của CGO cung cấp tập các hàm sau để chuyển đổi hai chiều array và string giữa Go và C:

```

// Go string -> C string
// C string được cấp phát trong C heap sử dụng malloc.
// Caller có trách nhiệm free nó sau khi sử dụng
// bằng cách như gọi C.free (nhớ include stdlib.h)
func C.CString(string) *C.char

// Go []byte slice -> C array
// C array được cấp phát trong C heap sử dụng malloc.
// Caller có trách nhiệm free nó sau khi sử dụng
// bằng cách như gọi C.free (nhớ include stdlib.h)
func C.CBytes([]byte) unsafe.Pointer

// C string -> Go string
func C.GoString(*C.char) string

// C data với length được chỉ định -> Go string
func C.GoStringN(*C.char, C.int) string

// C data với length được chỉ định -> Go []byte
func C.GoBytes(unsafe.Pointer, C.int) []byte

```

Khi string và slice của Go được chuyển đổi thành phiên bản trong C, hàm `malloc` của C cấp phát một vùng nhớ mới và cuối cùng có thể được giải phóng bằng `free`. Ngược lại khi một string hoặc array trong C được chuyển đổi thành kiểu tương ứng trong Go, vùng nhớ của dữ liệu được chuyển đổi được quản lý bởi ngôn ngữ Go.

Với các hàm chuyển đổi này, vùng nhớ trước chuyển đổi và sau chuyển đổi vẫn ở trong vùng nhớ cục bộ của vùng tương ứng của chúng. Ưu điểm của việc chuyển đổi này là quản lý interface và vùng nhớ rất đơn giản. Nhược điểm là cần cấp phát vùng nhớ mới và các hoạt động sao chép của nó sẽ dẫn đến chi phí phụ.

String và Slice

Các định nghĩa cho string và slice trong package `reflect`:

```

type StringHeader struct {
    Data uintptr
    Len  int
}

type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}

```

Nếu không muốn cấp phát vùng nhớ riêng, bạn có thể truy cập trực tiếp vào không gian bộ nhớ của C bằng Go:

```

/*
#include <string.h>
char arr[10];
char *s = "Hello";
*/
import "C"
import (
    "reflect"
    "unsafe"
    "fmt"
)

func main() {
    // chuyển đổi bằng reflect.SliceHeader
    var arr0 []byte
    var arr0Hdr = (*reflect.SliceHeader)(unsafe.Pointer(&arr0))
}

```

```

arr0Hdr.Data = uintptr(unsafe.Pointer(&C.arr[0]))
arr0Hdr.Len = 10
arr0Hdr.Cap = 10

// chuyển đổi slice
arr1 := (*[31]byte)(unsafe.Pointer(&C.arr[0]))[:10:10]

var s0 string
var s0Hdr = (*reflect.StringHeader)(unsafe.Pointer(&s0))
s0Hdr.Data = uintptr(unsafe.Pointer(C.s))
s0Hdr.Len = int(C.strlen(C.s))

sLen := int(C.strlen(C.s))
s1 := string((*[31]byte)(unsafe.Pointer(C.s))[sLen:sLen])

fmt.Println("arr1: ", arr1)
fmt.Println("s1: ", s1)

//kết quả:
//arr1: [0 0 0 0 0 0 0 0 0 0]
//s1: Hello
}

```

Vì chuỗi trong Go là chuỗi chỉ đọc, người dùng cần đảm bảo rằng nội dung của chuỗi C bên dưới sẽ không thay đổi trong quá trình sử dụng chuỗi đó trong Go và bộ nhớ sẽ không được giải phóng trước.

Trong CGO, phiên bản ngôn ngữ C của struct tương ứng với struct string và slice trên:

```

typedef struct { const char *p; GoInt n; } GoString;
typedef struct { void *data; GoInt len; GoInt cap; } GoSlice;

```

Trong C có thể dùng `GoString` và `GoSlice` để truy cập string và slice trong Go. Nếu là một kiểu mảng trong Go, bạn có thể chuyển đổi mảng thành một slice và sau đó chuyển đổi nó. Nếu không gian bộ nhớ bên dưới tương ứng với một string hoặc slice được quản lý bởi runtime của Go thì đối tượng bộ nhớ Go có thể được lưu trong một thời gian dài trong ngôn ngữ C.

Chi tiết về mô hình bộ nhớ CGO sẽ được thảo luận kĩ hơn trong các chương sau.

2.3.5. Chuyển đổi giữa các con trỏ

Trong ngôn ngữ C, các kiểu con trỏ khác nhau có thể được chuyển đổi tương ứng hoặc ngầm định. Việc chuyển đổi giữa các con trỏ cũng là vấn đề quan trọng đầu tiên cần được giải quyết trong code CGO.

Trong ngôn ngữ Go, nếu một kiểu con trỏ được xây dựng dựa trên một kiểu con trỏ khác, nói cách khác, hai con trỏ bên dưới là các con trỏ có cùng cấu trúc, thì chúng ta có thể chuyển đổi giữa các con trỏ bằng cú pháp cast trực tiếp. Tuy nhiên, CGO thường phải đối phó với việc chuyển đổi giữa hai kiểu con trỏ hoàn toàn khác nhau. Về nguyên tắc, thao tác này bị nghiêm cấm trong code Go thuần.

Một trong những mục đích của CGO là phá vỡ sự cấm đoán nói trên và khôi phục các thao tác chuyển đổi con trỏ tự do mà ngôn ngữ C nên có. Đoạn code sau trình bày cách chuyển đổi một con trỏ kiểu X thành một con trỏ kiểu Y:

```

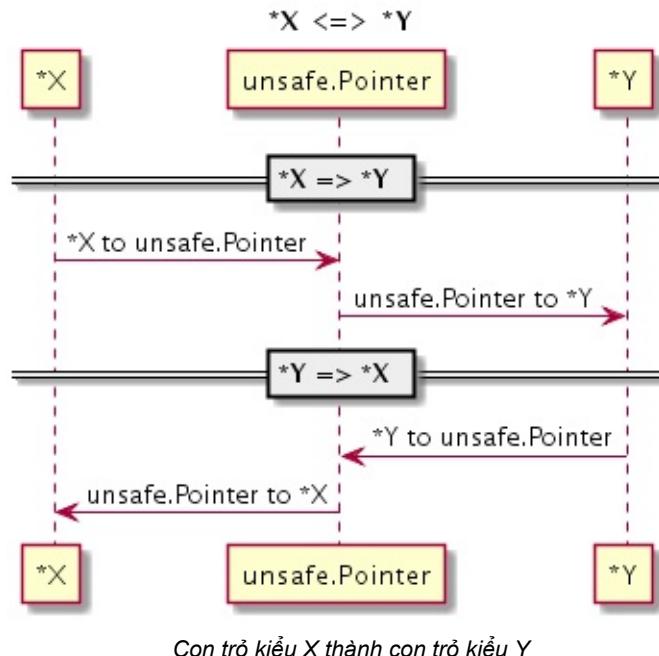
var p *X
var q *Y

q = (*Y)(unsafe.Pointer(p)) // *X => *Y
p = (*X)(unsafe.Pointer(q)) // *Y => *X

```

Để chuyển đổi con trỏ kiểu `X` thành con trỏ kiểu `Y`, chúng ta cần hiện thực hàm `unsafe.Pointer` chuyển đổi giữa các kiểu con trỏ khác nhau như một kiểu cầu nối trung gian. Kiểu con trỏ `unsafe.Pointer` tương tự với ngôn ngữ C với con trỏ `void*`.

Sau đây là sơ đồ quá trình chuyển đổi giữa các con trỏ:



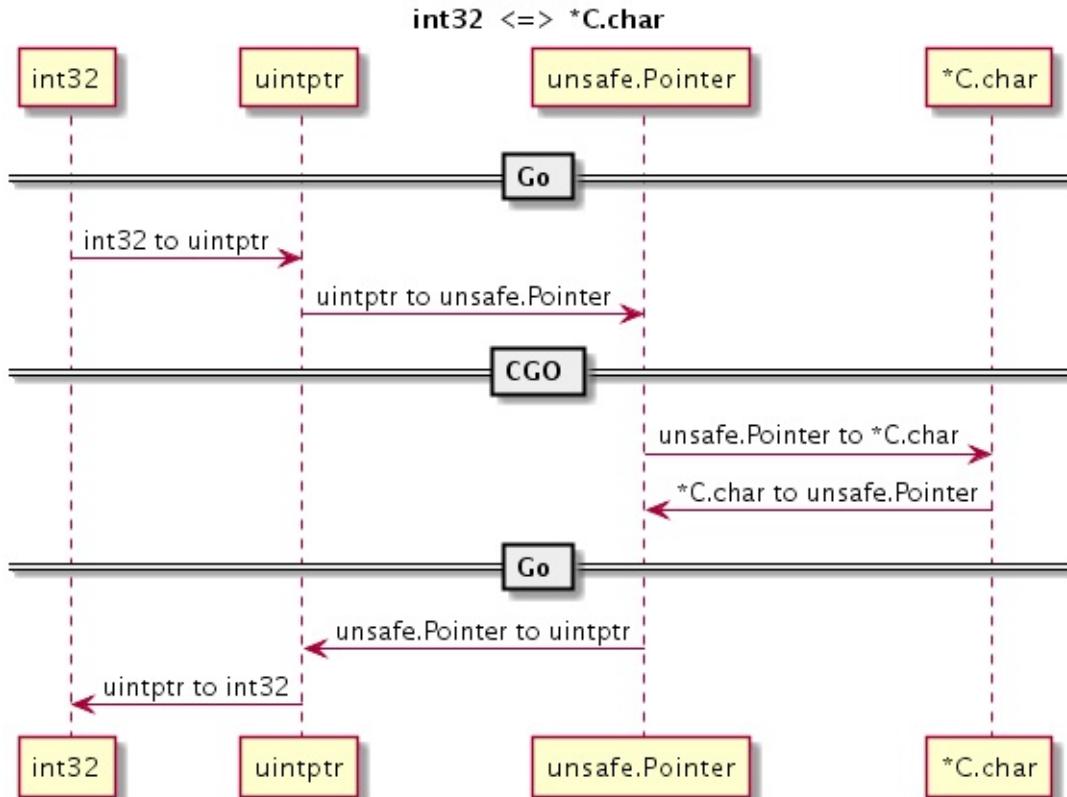
Bất kỳ kiểu con trỏ nào cũng có thể được chuyển sang kiểu con trỏ `unsafe.Pointer` để bỏ đi thông tin kiểu ban đầu, sau đó gán lại một kiểu con trỏ mới để đạt được mục đích chuyển đổi.

2.3.6. Chuyển đổi giá trị và con trỏ

Trong ngôn ngữ C, ta thường gặp trường hợp con trỏ được biểu diễn bởi giá trị thông thường, làm thế nào để hiện thực việc chuyển đổi giá trị và con trỏ cũng là một vấn đề mà CGO cần phải đối mặt.

Để kiểm soát chặt chẽ việc sử dụng con trỏ, ngôn ngữ Go không cho phép chuyển đổi các kiểu số trực tiếp thành các kiểu con trỏ. Tuy nhiên, Go đã đặc biệt định nghĩa một kiểu `uintptr` cho các kiểu con trỏ `unsafe.Pointer`. Chúng ta có thể sử dụng `uintptr` làm trung gian để hiện thực các kiểu số thành các kiểu `unsafe.Pointer`.

Biểu đồ sau đây trình bày cách hiện thực chuyển đổi lẫn nhau của kiểu `int32` sang kiểu con trỏ `char*` là chuỗi trong ngôn ngữ C:

*Int32 và char chuyển đổi con trỏ*

Việc chuyển đổi được chia thành nhiều giai đoạn: đầu tiên là kiểu `int32` sang `uintptr`, sau đó là `uintptr` thành kiểu con trỏ `unsafe.Pointer` và cuối cùng là kiểu con trỏ `unsafe.Pointer` thành kiểu `*c.char`.

2.3.7. Chuyển đổi giữa kiểu slice

Mảng cũng là một loại con trỏ trong ngôn ngữ C, vì vậy việc chuyển đổi giữa hai kiểu mảng khác nhau về cơ bản tương tự như chuyển đổi giữa các con trỏ. Tuy nhiên trong ngôn ngữ Go, slice thực ra là một con trỏ tới một mảng (fat pointer), vì vậy chúng ta không thể chuyển đổi trực tiếp giữa các kiểu slice khác nhau.

Tuy nhiên, package `reflection` của ngôn ngữ Go đã cung cấp sẵn cấu trúc cơ bản của kiểu slice nhờ đó chuyển đổi slice có thể được hiện thực:

```
var p []X
var q []Y

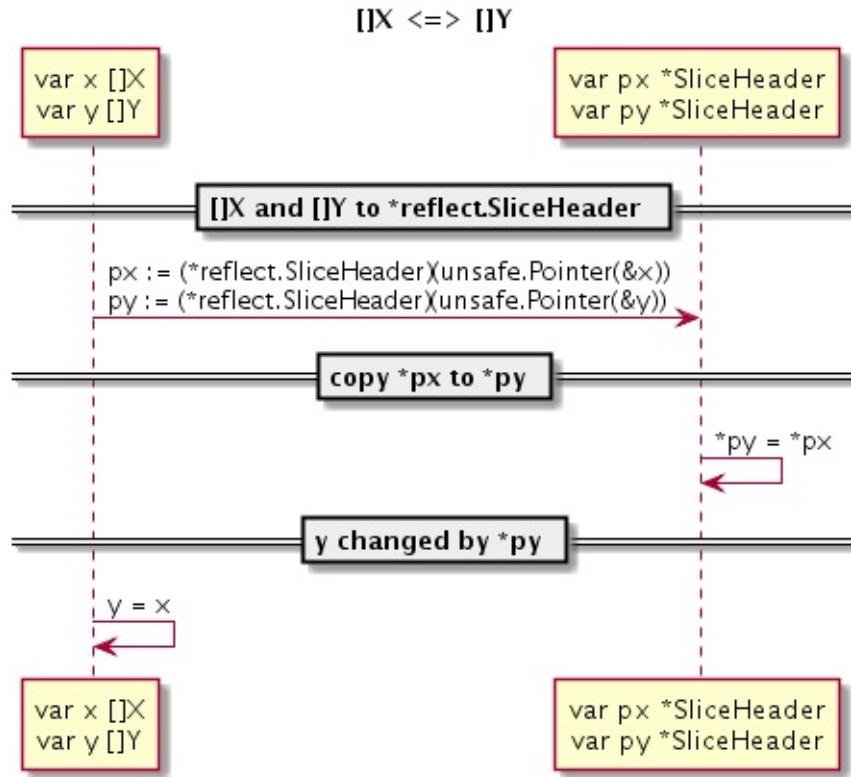
// tạo slice trống
pHdr := (*reflect.SliceHeader)(unsafe.Pointer(&p))
qHdr := (*reflect.SliceHeader)(unsafe.Pointer(&q))

// chuyển dữ liệu bên trong slice
pHdr.Data = qHdr.Data

// chuyển các thông tin về len và cap
pHdr.Len = qHdr.Len * unsafe.Sizeof(q[0]) / unsafe.Sizeof(p[0])
pHdr.Cap = qHdr.Cap * unsafe.Sizeof(q[0]) / unsafe.Sizeof(p[0])
```

Cần lưu ý rằng nếu X hoặc Y là kiểu null, đoạn code trên có thể gây ra lỗi chia cho 0 và code thực tế cần được xử lý khi thích hợp.

Sau đây cho thấy luồng cụ thể của thao tác chuyển đổi giữa các slice:



kiểu cắt X thành slice Y

Đối với các tính năng thường được sử dụng trong CGO, tác giả package github.com/chai2010/cgo, đã cung cấp các chức năng chuyển đổi cơ bản. Để biết thêm chi tiết hãy tham khảo code hiện thực.

Liên kết

- Phần tiếp theo: [Lời gọi hàm](#)
- Phần trước: [Chuyển đổi kiểu dữ liệu](#)
- [Mục lục](#)

2.4 Lời gọi hàm

Qua công cụ CGO, chúng ta không chỉ có thể gọi hàm của ngôn ngữ C bằng Go mà còn có thể export hàm của Go để sử dụng như là hàm ngôn ngữ C.

2.4.1 Go gọi hàm C

Đối với chương trình kích hoạt các tính năng CGO, CGO xây dựng một package C ảo. Hàm ngôn ngữ C có thể được gọi qua package C ảo này.

```
/*
static int add(int a, int b) {
    return a+b;
}
*/
import "C"

func main() {
    C.add(1, 1)
}
```

Code CGO ở trên trước tiên xác định hàm `add` hiển thị trong file hiện tại và sau đó chuyển sang `C.add`.

2.4.2 Giá trị trả về của hàm C

Đối với hàm của C có giá trị trả về, chúng ta có thể nhận giá trị trả về bình thường.

```
/*
static int div(int a, int b) {
    return a/b;
}
*/
import "C"
import "fmt"

func main() {
    v := C.div(6, 3)
    fmt.Println(v)
}
```

Hàm `div` ở trên thực hiện một phép toán chia số nguyên và trả về kết quả của phép chia.

Tuy nhiên, không có cách xử lý đặc biệt nào cho trường hợp số chia là 0. Vì ngôn ngữ C không hỗ trợ trả về nhiều kết quả nên nếu bạn muốn trả về lỗi khi số chia là 0 thì có thể xem `errno` là một biến toàn cục thread-safe có thể được sử dụng để ghi lại mã trạng thái của lỗi đây nhất.

CGO hỗ trợ các macro `errno` thuộc thư viện tiêu chuẩn : nếu có hai giá trị trả về khi CGO gọi hàm C thì giá trị trả về thứ hai sẽ tương ứng với trạng thái lỗi `errno`.

```
/*
#include <errno.h>

static int div(int a, int b) {
    if(b == 0) {
        errno = EINVAL;
```

```

        return 0;
    }
    return a/b;
}
*/
import "C"
import "fmt"

func main() {
    v0, err0 := C.div(2, 1)
    fmt.Println(v0, err0)

    v1, err1 := C.div(1, 0)
    fmt.Println(v1, err1)
}

```

Thực thi đoạn code trên sẽ cho output như sau:

```

2 <nil>
0 invalid argument

```

Chúng ta có thể xem hàm `div` tương ứng với một hàm trong Go như sau:

```

func C.div(a, b C.int) (C.int, error)

```

Tham số thứ hai trả về (giá trị `error`) có thể bỏ qua, được hiện thực bên dưới là kiểu `syscall.Errno`.

2.4.3 Giá trị trả về của hàm void

Trong C cũng có hàm không trả về kiểu giá trị (thay vào đó trả về `void`). Chúng ta không thể nhận được giá trị trả về của hàm kiểu `void` vì đó thực sự không phải giá trị!?

Như đã đề cập trong ví dụ trước, CGO hiện thực một phương pháp đặc biệt cho `errno` và có thể nhận về trạng thái lỗi của ngôn ngữ C thông qua giá trị trả về thứ hai. Tính năng này vẫn hợp lệ cho các hàm kiểu `void`. Đoạn code sau để lấy mã trạng thái lỗi của hàm không có giá trị trả về:

```

//static void noreturn() {}
import "C"
import "fmt"

func main() {
    _, err := C.noreturn()
    fmt.Println(err)
}

// kết quả: <nil>

```

Lúc này, chúng ta bỏ qua giá trị trả về đầu tiên và chỉ nhận được mã lỗi tương ứng với giá trị trả về thứ hai.

Chúng ta cũng có thể thử lấy giá trị trả về đầu tiên, cũng chính là kiểu tương ứng với kiểu `void` trong ngôn ngữ C:

```

//static void noreturn() {}
import "C"
import "fmt"

func main() {
    v, _ := C.noreturn()
    fmt.Printf("%#v", v)
}

```

Chạy code này sẽ thu được kết quả:

```
main._Ctype_void{}
```

Chúng ta có thể thấy rằng kiểu void của ngôn ngữ C tương ứng với kiểu trong package main `_Ctype_void`. Trong thực tế, hàm `noreturn` của ngôn ngữ C cũng được coi là một hàm với kiểu trả về `_Ctype_void`, do đó bạn có thể trực tiếp nhận giá trị trả về của hàm kiểu void:

```
//static void noreturn() {}
import "C"
import "fmt"

func main() {
    fmt.Println(C.noreturn())
}
```

Chạy code này sẽ cho ra kết quả sau:

```
[]
```

Trong thực tế, trong code được CGO tạo ra, kiểu `_Ctype_void` tương ứng với kiểu mảng có độ dài 0 (`[0]byte`), do đó output `fmt.Println` là một cặp dấu ngoặc vuông biểu thị một giá trị null.

2.4.4 C gọi hàm do Go export

CGO có một tính năng mạnh mẽ là export các hàm Go thành các hàm ngôn ngữ C. Trong trường hợp này, chúng ta có thể định nghĩa interface bằng C và sau đó triển khai nó thông qua Go. Trong phần đầu tiên của chương này cũng đã có một số ví dụ về hàm ngôn ngữ C gọi hàm do Go export.

Nhắc lại một chút về hàm `add` trong chương đầu:

```
import "C"

//export add
func add(a, b C.int) C.int {
    return a+b
}
```

Tên hàm `add` bắt đầu bằng một chữ cái viết thường và là một hàm private trong package của Go. Nhưng theo cách nhìn của ngôn ngữ C thì hàm `add` là hàm ngôn ngữ C có thể được truy cập toàn cục. Nếu có một hàm `add` cùng tên được export dưới dạng hàm ngôn ngữ C trong hai package ngôn ngữ Go khác nhau, vẫn đè về trùng tên sẽ xảy ra trong link phase.

Chúng ta có thể include file header `_cgo_export.h` để thêm tham chiếu đến các hàm export. Nếu bạn muốn sử dụng ngay lập tức hàm `add` của C được export trong file CGO hiện tại, bạn không thể tham khảo đến file `_cgo_export.h`, bởi vì việc tạo file `_cgo_export.h` cần phụ thuộc vào file hiện tại mà trong file hiện tại lại tham khảo tới `_cgo_export.h` (file chưa được tạo) thì sẽ gây ra lỗi.

```
#include "_cgo_export.h"

void foo() {
    add(1, 1);
}
```

Khi export interface của ngôn ngữ C, ta cần đảm bảo rằng các tham số hàm và kiểu giá trị trả về là kiểu "thân thiện" với C (xem lại [2.3](#)) đồng thời giá trị trả về không được trực tiếp hoặc gián tiếp chứa con trỏ vào khung gian bộ nhớ ngôn ngữ Go.

Liên kết

- Phần tiếp theo: [Cơ chế bên trong CGO](#)
- Phần trước: [Chuyển đổi kiểu dữ liệu](#)
- [Mục lục](#)

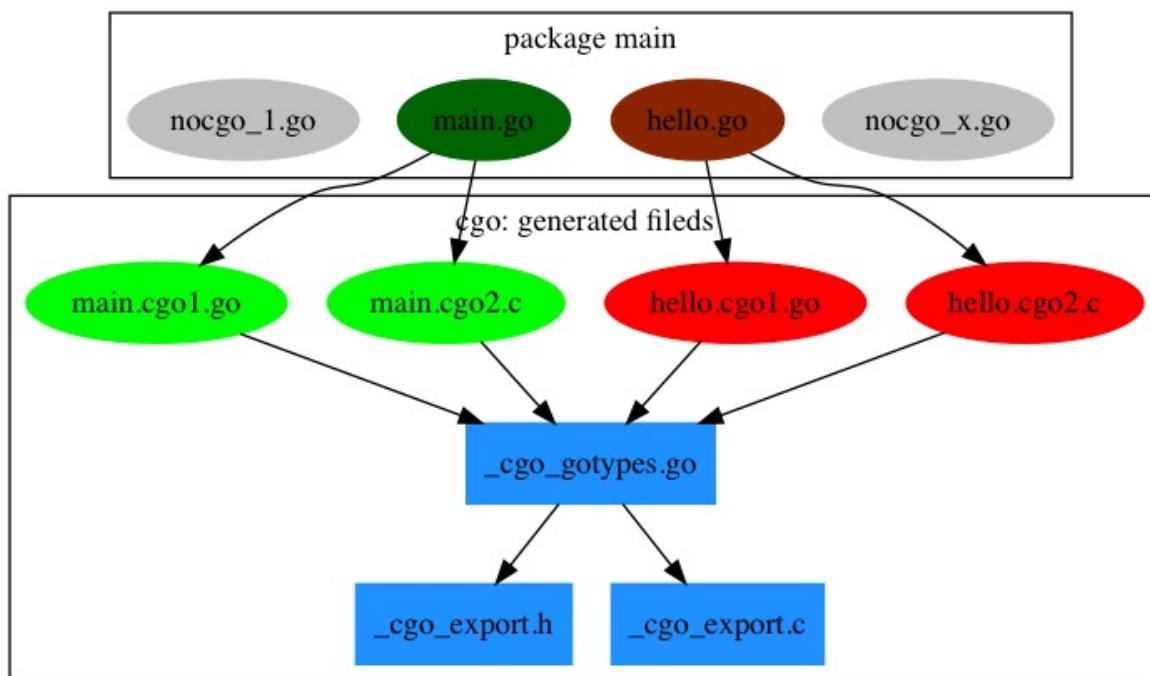
2.5. Cơ chế bên trong CGO

Để hiểu rõ hơn cách CGO vận hành, trong phần này chúng ta sẽ phân tích luồng hoạt động của các hàm ngôn ngữ Go và C từ code được tạo.

2.5.1. Các file trung gian được CGO tạo ra

Để hiểu cơ chế của CGO, trước tiên ta cần biết các file trung gian mà CGO tạo ra. Ta có thể thêm một thư mục `-work` chứa file trung gian output khi build package cgo và giữ file trung gian khi quá trình build hoàn tất. Nếu là một đoạn code cgo đơn giản, chúng ta cũng có thể trực tiếp xem file trung gian được tạo bằng cách gọi lệnh `go tool cgo`.

Trong file nguồn Go, nếu một lệnh `import "C"` thực thi thì lệnh cgo sẽ được gọi để tạo ra file trung gian tương ứng. Dưới đây là sơ đồ đơn giản mô tả các file trung gian được cgo tạo ra:



Các file trung gian được CGO tạo ra

Có 4 file Go trong package, trong đó các file nocgo chứa `import "C"` và hai file còn lại chứa code cgo. Lệnh cgo tạo ra hai file trung gian cho mỗi file chứa mã cgo. Ví dụ: `main.go` tạo ra hai file trung gian là `main.cgo1.go` và `main.cgo2.c`.

Kế đó file `_cgo_gotypes.go` được tạo cho toàn bộ package chứa một phần code hỗ trợ của Go. Đồng thời quá trình này cũng tạo ra các file `_cgo_export.h` và `_cgo_export.c`, để export các kiểu và hàm trong Go tới kiểu và hàm tương ứng trong C.

2.5.2. Go gọi hàm của C

Go gọi các hàm trong C là trường hợp ứng dụng phổ biến nhất của CGO. Chúng ta sẽ bắt đầu với ví dụ đơn giản nhất để phân tích chi tiết luồng hoạt động của quá trình này.

Đoạn code cụ thể như sau:

```
//main.go
package main

//int sum(int a, int b) { return a+b; }
import "C"

func main() {
    println(C.sum(1, 1))
}
```

Tiếp theo ta tạo một file trung gian trong thư mục _obj thông qua command line cgo:

```
$ go tool cgo main.go
```

Vào thư mục _obj để xem các file trung gian:

```
$ ls _obj
_cgo_.o
_cgo_export.c
_cgo_export.h
_cgo_flags
_cgo_gotypes.go
_cgo_main.c
main.cgo1.go
main.cgo2.c
```

Trong đó `_cgo_.o` , `_cgo_flags` và `_cgo_main.c` có code không liên quan logic trực tiếp với nhau, bạn có thể bỏ qua.

Trước tiên chúng ta hãy xem file `main.cgo1.go` chứa code Go sau khi file `main.go` expand các hàm và biến số liên quan trong package C ảo:

```
package main

//int sum(int a, int b) { return a+b; }
import _ "unsafe"

func main() {
    println(_Cfunc_sum(1, 1))
}
```

Lời gọi `C.sum(1, 1)` được thay thế thành `(_Cfunc_sum)(1, 1)` . Mỗi dạng `c.xxx` của hàm được thay thế bằng hàm Go thuần túy dạng `_Cfunc_xxx` , trong đó tiền tố `_Cfunc_` chỉ ra rằng đây là hàm C.

Hàm `_Cfunc_sum` được định nghĩa trong file `_cgo_gotypes.go` do CGO tạo ra như sau:

```
//go:cgo_unsafe_args
func _Cfunc_sum(p0 _Ctype_int, p1 _Ctype_int) (r1 _Ctype_int) {
    _cgo_runtime_cgocall(_cgo_506f45f9fa85_Cfunc_sum, uintptr(unsafe.Pointer(&p0)))
    if _Cgo_always_false {
        _Cgo_use(p0)
        _Cgo_use(p1)
    }
    return
}
```

Tham số của hàm `_Cfunc_sum` và kiểu `_Ctype_int` của giá trị trả về tương ứng với kiểu `c.int` , các quy tắc đặt tên `_Cfunc_xxx` là tương tự nhau và các tiền tố khác nhau được sử dụng để phân biệt giữa các hàm và kiểu.

Hàm `_cgo_runtime_cgocall` tương ứng với `runtime.cgocall`, khai báo của hàm như sau:

```
func runtime.cgocall(fn, arg unsafe.Pointer) int32
```

Tham số đầu tiên là địa chỉ của hàm ngôn ngữ C và tham số thứ hai là địa chỉ của struct tham số tương ứng với hàm ngôn ngữ C.

Trong ví dụ này, hàm được truyền vào: `_cgo_506f45f9fa85_Cfunc_sum` cũng là một hàm trung gian được CGO tạo ra.

Hàm được định nghĩa trong `main.cgo2.c1`:

```
void _cgo_506f45f9fa85_Cfunc_sum(void *v) {
    struct {
        int p0;
        int p1;
        int r;
        char __pad12[4];
    } __attribute__((__packed__)) *a = v;
    char *stktop = _cgo_topofstack();
    __typeof__(a->r) r;
    _cgo_tsan_acquire();
    r = sum(a->p0, a->p1);
    _cgo_tsan_release();
    a = (void*)((char*)a + (_cgo_topofstack() - stktop));
    a->r = r;
}
```

Tham số hàm này chỉ có một con trả trả về kiểu void và hàm không có giá trị trả về.

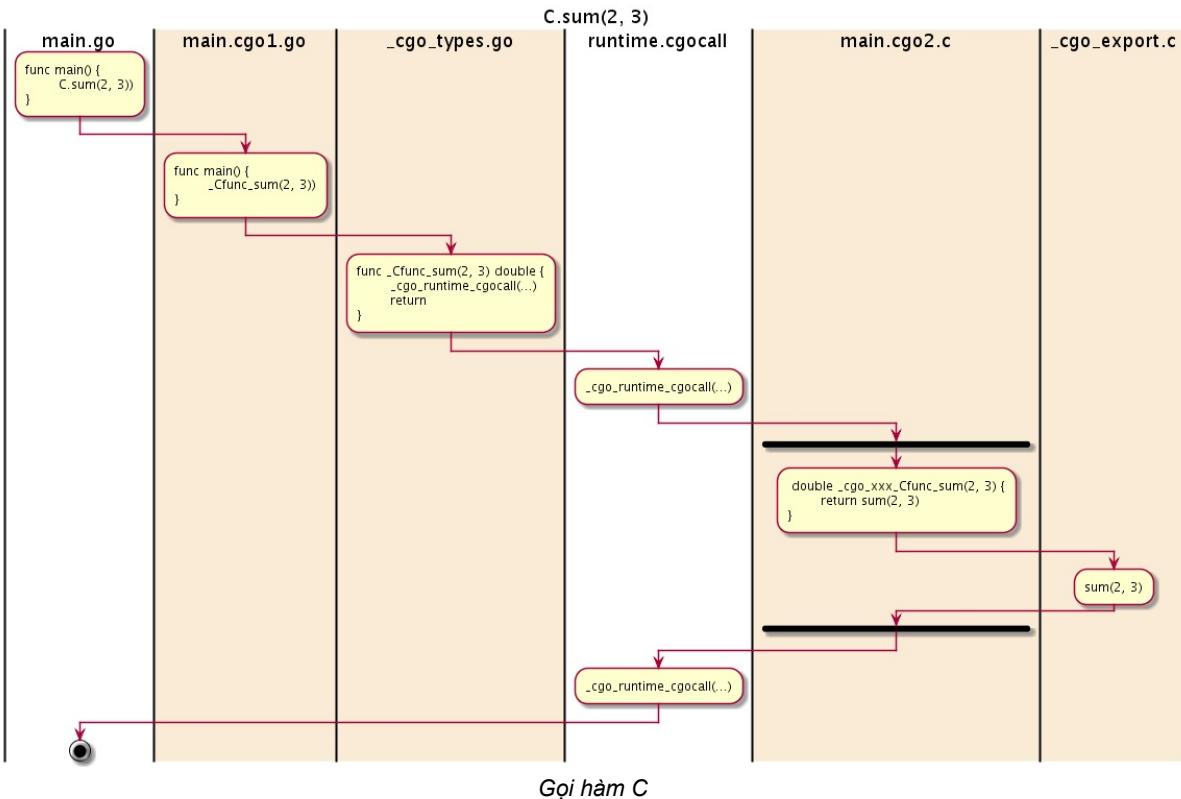
Struct được trả trả bởi con trả hàm `_cgo_506f45f9fa85_Cfunc_sum` là:

```
struct {
    int p0;
    int p1;
    int r;
    char __pad12[4];
} __attribute__((__packed__)) *a = v;
```

Thành phần `p0` tương ứng với tham số đầu tiên của `sum`, thành phần `p1` tương ứng với tham số thứ hai và thành phần `__pad12` được sử dụng để điền vào struct cho mục đích đảm bảo alignment của CPU.

Sau khi có được các tham số (trả trả struct), hàm `sum` của phiên bản ngôn ngữ C được gọi và giá trị trả về được lưu vào thành phần tương ứng trong thán struct.

Toàn bộ biểu đồ luồng hoạt động của cuộc gọi `C.sum` như sau:



Trong đó hàm `runtime.cgocall` là chia khóa để thực hiện cuộc gọi vượt ranh giới của hàm ngôn ngữ Go sang hàm ngôn ngữ C. Thông tin chi tiết có thể tham khảo <https://golang.org/src/cmd/cgo/doc.go>.

2.5.3. C gọi hàm của Go

Bây giờ chúng ta sẽ phân tích luồng của cuộc gọi ngược lại: C gọi đến hàm Go. Tương tự, ta cũng khởi tạo một hàm Go, tên file là sum.go:

```
package main

//int sum(int a, int b);
import "C"

//export sum
func sum(a, b C.int) C.int {
    return a + b
}

func main() {}
```

Các chi tiết về cú pháp của CGO không được mô tả ở đây. Để sử dụng hàm `sum` trong C, chúng ta cần biên dịch mã Go vào thư viện tĩnh của C:

```
$ go build -buildmode=c-archive -o sum.a sum.go
```

Nếu không có lỗi, lệnh biên dịch ở trên sẽ tạo ra một thư viện tĩnh `sum.a` và file header `sum.h`. File này sẽ chứa khai báo của hàm `sum` và thư viện tĩnh sẽ chứa hiện thực của hàm.

Để phân tích luồng hoạt động của cuộc gọi hàm từ phiên bản ngôn ngữ C ta cũng cần phải phân tích các file trung gian do CGO tạo ra:

```
$ go tool cgo sum.go
```

Thư mục _obj vẫn chứa các file trung gian được tạo tương tự như phần trước. Để ngắn gọn, ta bỏ qua một vài file không liên quan:

```
$ ls _obj
_cgo_export.c
_cgo_export.h
_cgo_gotypes.go
main.cgo1.go
main.cgo2.c
```

Trong đó nội dung của file `_cgo_export.h` và file do C tạo ra khi nó tạo thư viện tĩnh `sum.h` là giống nhau, đều khai báo hàm sum.

Vì ngôn ngữ C là bên gọi, chúng ta cần bắt đầu với việc hiện thực phiên bản ngôn ngữ C của hàm sum. Phiên bản này nằm trong file `_cgo_export.c` (file chứa phần hiện thực hàm của C tương ứng với hàm export của Go):

```
int sum(int p0, int p1)
{
    __SIZE_TYPE__ _cgo_ctxt = _cgo_wait_runtime_init_done();
    struct {
        int p0;
        int p1;
        int r0;
        char __pad0[4];
    } __attribute__((__packed__)) a;
    a.p0 = p0;
    a.p1 = p1;
    _cgo_tsan_release();
    crosscall2(_cgoexp_8313eaf44386_sum, &a, 16, _cgo_ctxt);
    _cgo_tsan_acquire();
    _cgo_release_context(_cgo_ctxt);
    return a.r0;
}
```

Hàm sum sử dụng một kỹ thuật tương tự như phần trước trình bày để đóng gói các tham số và trả về các giá trị của hàm thành một struct, sau đó truyền struct `runtime/cgo.crosscall2` vào hàm thực thi thông qua hàm `_cgoexp_8313eaf44386_sum`.

Hàm `runtime/cgo.crosscall2` được hiện thực bằng hợp ngữ và khai báo hàm của nó như sau:

```
func runtime/cgo.crosscall2(
    fn func(a unsafe.Pointer, n int32, ctxt uintptr),
    a unsafe.Pointer, n int32,
    ctxt uintptr,
)
```

Điểm cần chú ý ở đây là `fn` và `a`, `fn` là con trỏ tới hàm trung gian (proxy) và `a` là con trỏ tới struct tương ứng với đối số truyền đi khi gọi (và cũng chứa luôn giá trị trả về).

Hàm trung gian `_cgoexp_8313eaf44386_sum` có trong file `_cgo_gotypes.go`:

```
func _cgoexp_8313eaf44386_sum(a unsafe.Pointer, n int32, ctxt uintptr) {
    fn := _cgoexpwrap_8313eaf44386_sum
    _cgo_runtime_cgocallback(**(**unsafe.Pointer)(unsafe.Pointer(&fn)), a, uintptr(n), ctxt);
}

func _cgoexpwrap_8313eaf44386_sum(p0 _Ctype_int, p1 _Ctype_int) (r0 _Ctype_int) {
    return sum(p0, p1)
```

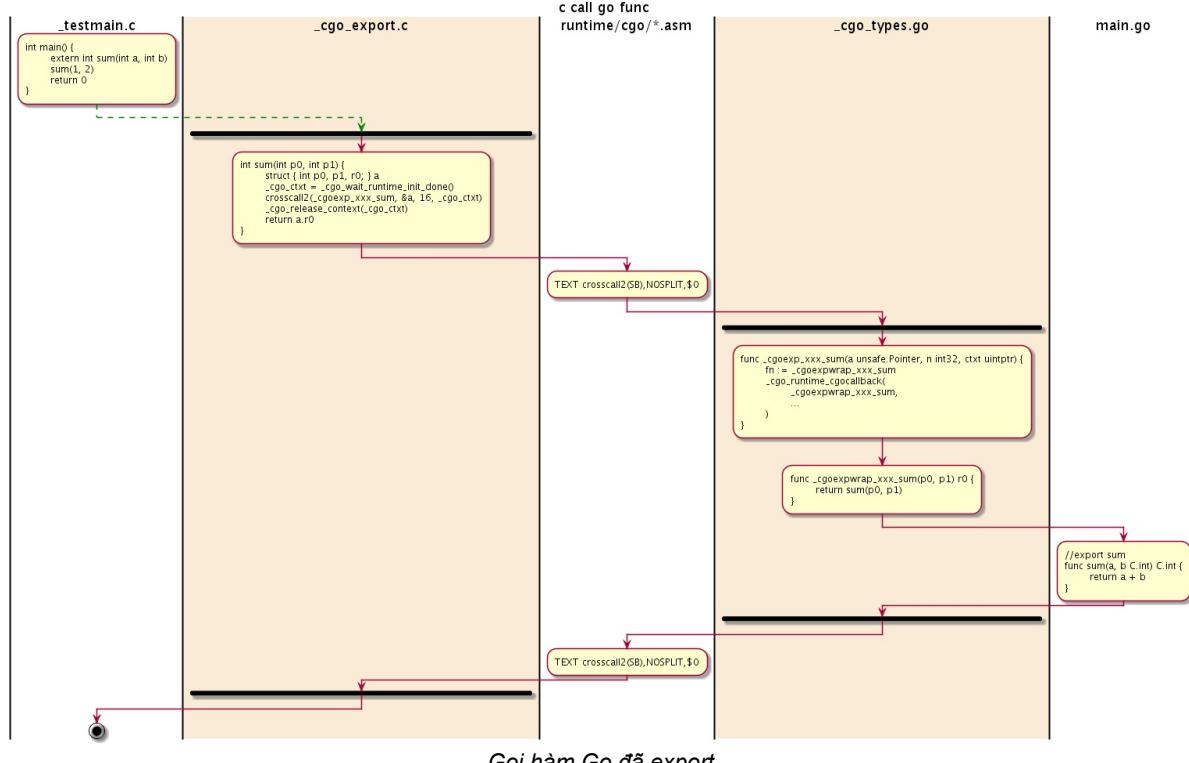
```
}
```

Hàm bao ngoài `_cgoexpwrap_8313eaf44386_sum` của `sum` được sử dụng như một con trỏ hàm. Hàm `_cgo_runtime_cgocallback` tương ứng với hàm `runtime.cgocallback`:

```
func runtime.cgocallback(fn, frame unsafe.Pointer, framesize, ctxt uintptr)
```

Các tham số là con trỏ hàm, tham số hàm và giá trị trả về tương ứng với con trỏ của struct, kích thước frame của lời gọi hàm và tham số ngũ cành.

Toàn bộ biểu đồ luồng cuộc gọi như sau:



Gọi hàm Go đã export

Trong đó, hàm `runtime.cgocallback` là chìa khóa để thực hiện cuộc gọi vượt ranh giới từ ngôn ngữ C sang Go. Chi tiết có thể được tìm thấy trong hiện thực `runtime.cgocallback.go`

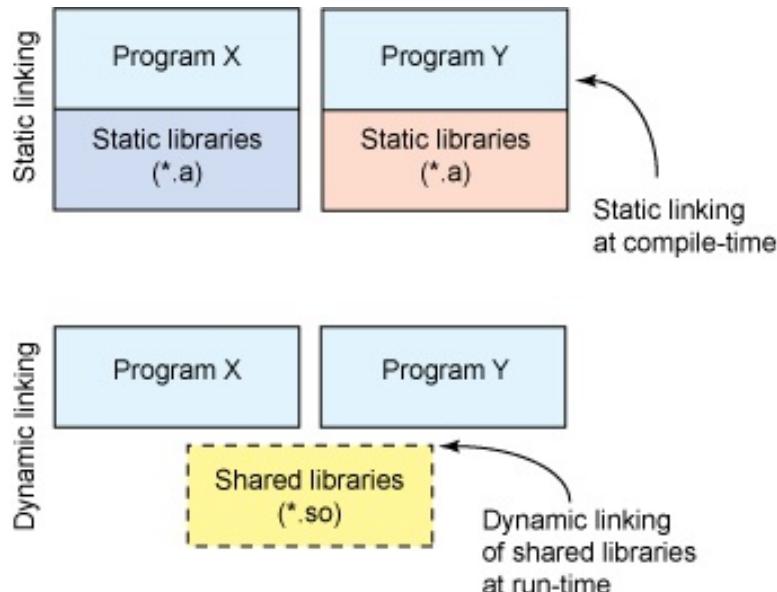
Liên kết

- Phản tiếp theo: [Tạo ra package qsort](#)
- Phản trước: [Lời gọi hàm](#)
- Mục lục

2.9. Thư viện tĩnh và động

Có ba cách để sử dụng mã nguồn C/C++ trong CGO:

1. Dùng trực tiếp mã nguồn (thêm dòng `import "c"` và chú thích mã nguồn C phía trên, hoặc bao gộp mã nguồn c/c++ trong package hiện tại).
2. Liên kết tĩnh mã nguồn (khai báo thư viện liên kết trong cờ `LDFLAGS`).
3. Liên kết động mã nguồn.



Chi tiết về sự khác biệt giữa thư viện tĩnh và động bạn đọc có thể xem thêm tại đây: [What-is-the-difference-between-static-and-dynamic-linking](#).

Sau đây chúng ta sẽ đi vào cách dùng thư viện tĩnh và thư viện động trong CGO.

2.9.1. Dùng thư viện C tĩnh

Nếu mã nguồn C/C++ được dùng trong CGO có kích thước nhỏ thì cách đưa trực tiếp chúng vào chương trình là một ý tưởng phổ biến nhất, nhưng nhiều lúc chúng ta không tự xây dựng mã nguồn, hoặc quá trình xây dựng mã nguồn C/C++ rất phức tạp thì đây là lúc thư viện C tĩnh phát huy thế mạnh của mình.

Ở ví dụ đầu tiên, chúng ta sẽ xây dựng một thư viện tĩnh đơn giản được gọi là `number`, chỉ có một hàm `number_add_mod` trong thư viện dùng để lấy modulo của một tổng hai số cho một số thứ ba, những files của thư viện `number` đặt trong cùng một thư mục:

number/number.h: header chứa prototype của hàm:

```
int number_add_mod(int a, int b, int mod);
```

number/number.c: phần hiện thực hàm như sau:

```
#include "number.h"

int number_add_mod(int a, int b, int mod) {
    return (a+b)%mod;
}
```

Bởi vì CGO dùng lệnh GCC để biên dịch và liên kết mã nguồn C và Go lại, do đó thư viện tĩnh cần phải tương thích với GCC compiler.

Thư viện tĩnh `libnumber.a` có thể được sinh ra bằng lệnh sau:

```
// di chuyển tới thư mục mã nguồn
$ cd ./number
// biên dịch ra file object từ file mã nguồn
$ gcc -c -o number.o number.c
// lệnh tạo ra thư viện tĩnh libnumber.a từ file object
// chi tiết về lệnh ar có thể xem tại https://linux.die.net/man/1/ar
$ ar rcs libnumber.a number.o
```

Sau khi sinh ra thư viện tĩnh mang tên `libnumber.a`, chúng ta dùng nó trong CGO.

`main.go` được tạo ra như sau:

```
package main

//#cgo CFLAGS: -I./number
//#cgo LDFLAGS: -L${SRCDIR}/number -lnumber
//
//#include "number.h"
import "C"
import "fmt"

func main() {
    fmt.Println(C.number_add_mod(10, 5, 12))
}
```

Hai lệnh `#cgo` trên dùng để biên dịch và liên kết mã nguồn với nhau:

- Cờ `CFLAGS -I./number` : khai báo đường dẫn đến thư mục mã nguồn.
- Cờ `LDFLAGS: -L${SRCDIR}/number -lnumber` : khai báo đường dẫn đến thư viện tĩnh `libnumber.a` với search path `-lnumber`.

Chú ý rằng: đường dẫn trong liên kết không thể dùng [relative path](#) mà phải dùng một [absolute path](#), ngoài ra đường dẫn không được chứa bất kỳ khoảng trắng nào.

Ví dụ : `LDFLAGS: -L/home/mypc/number -lnumber`

Kết quả như sau:

```
$ go run main.go
3
```

Nếu chúng ta sử dụng thư viện tĩnh từ bên thứ ba, chúng ta cần phải tải chúng và cài đặt thư viện tĩnh đến một nơi phù hợp, sau đó đặc tả location của header files và libraries qua cờ `CFLAGS` và `LDFLAGS` trong lệnh `#cgo`.

Trong môi trường Linux, có một lệnh `pkg-config` được dùng để truy vấn các tham số compile và link khi dùng các thư viện tĩnh, chúng ta có thể dùng lệnh `pkg-config` trực tiếp trong lệnh `#cgo` để generate compilation và linking parameters, bạn có thể customize lệnh `pkg-config` với biến môi trường `PKG_CONFIG`.

2.9.2. Sử dụng thư viện C động

Ý tưởng của thư viện động là shared library, các process khác nhau có thể chia sẻ trên cùng một tài nguyên bộ nhớ trên RAM hoặc đĩa cứng, nhưng hiện nay giá thành đĩa cứng và RAM cũng tương đối rẻ, nên hai vai trò sẽ trở nên không đáng quan tâm, do đó đâu là giá trị của thư viện động ở đây?

Từ góc nhìn của việc phát triển thư viện, thư viện động có thể tách biệt nhau và giảm thiểu rủi ro của việc xung đột trong khi liên kết, với những nền tảng như Windows, thư viện động là một cách khả thi để mở rộng các nền tảng biên dịch như `gcc`.

Trong môi trường `gcc` dưới MacOS hoặc Linux, chúng ta có thể sinh ra thư viện động của một số thư viện với những lệnh sau:

```
$ cd number
$ gcc -shared -o libnumber.so number.c
```

Bởi vì, base names của thư viện động và tĩnh là `libnumber`, chỉ phần hậu tố là sẽ khác. Do đó trong mã nguồn của ngôn ngữ Go sẽ giống chính xác với phiên bản thư viện tĩnh.

```
package main

//#cgo CFLAGS: -I./number
//#cgo LDFLAGS: -L${SRCDIR}/number -lnumber
//
//#include "number.h"
import "C"
import "fmt"

func main() {
    fmt.Println(C.number_add_mod(10, 5, 12))
}
```

`cgo` sẽ tự động tìm `libnumber.a` hoặc `libnumber.so` ở bước liên kết trong thời gian biên dịch.

Với nền tảng Windows, chúng ta có thể dùng công cụ `VC` để sinh ra thư viện động (sẽ có một số thư viện Windows phức tạp chỉ có thể được build với `vc`). Đầu tiên, chúng ta phải tạo một file định nghĩa cho `number.dll` để quản lý các kí hiệu dùng để export thư viện động.

Nội dung của file `number.def` như sau:

```
LIBRARY number.dll

EXPORTS
number_add_mod
```

Dòng đầu tiên `LIBRARY` sẽ chỉ ra tên của file và tên của thư viện động, và sau đó là mệnh đề `EXPORTS` theo sau bởi một danh sách các tên dùng để export.

Giờ đây, chúng ta có thể dùng những lệnh sau để tạo ra thư viện động (cần dùng `vc tools`).

```
$ cl /c number.c
$ link /DLL /OUT:number.dll number.obj number.def
```

Vào lúc này, một export library `number.lib` sẽ sinh ra `dll` cùng lúc. Nhưng trong `cgo`, chúng ta không thể dùng `link library` trong định dạng `lib`.

Để sinh ra định dạng `.a` cho việc export library cần dùng `mingw Toolbox dlltool command`:

```
$ dlltool -dllname number.dll --def number.def --output-lib libnumber.a
```

Một khi `libnumber.a` được sinh ra, có thể dùng `-lnumber` thông qua các link parameters.

Nên chú ý rằng, tại thời điểm thực thi, thư viện động cần được đặt ở cùng nơi để system có thể thấy. Trên Windows, bạn có thể đặt dynamic library và executable program trong cùng một thư mục, hoặc thêm đường dẫn tuyệt đối của thư mục trong khi dynamic library được đưa vào biến môi trường PATH. Trong MacOS, bạn cần phải thiết lập biến môi trường DYLD_LIBRARY_PATH. Trong hệ thống Linux, bạn cần thiết lập biến LD_LIBRARY_PATH.

2.9.3. Exporting thư viện C tĩnh

CGO không chỉ được dùng trong thư viện C tĩnh, các export functions được hiện thực bởi Go hoặc C static libraries. Chúng ta có thể dùng Go để hiện thực modulo addition function như phần trước ở ví dụ như sau đây.

Tạo `number.go` với nội dung như sau:

```
package main

import "C"

func main() {}

//export number_add_mod
func number_add_mod(a, b, mod C.int) C.int {
    return (a + b) % mod
}
```

Theo như mô tả của tài liệu CGO, chúng ta cần export C function trong main package. Với cách xây dựng thư viện C tĩnh, hàm main trong main package được bỏ qua, và hàm C sẽ đơn giản được export. Xây dựng các lệnh sau:

```
$ go build -buildmode=c-archive -o number.a
```

Khi sinh ra thư viện tĩnh `number.a`, cgo cũng sẽ sinh ra file `number.h`.

Nội dung của `number.h` sẽ như sau:

```
#ifdef __cplusplus
extern "C" {
#endif

extern int number_add_mod(int p0, int p1, int p2);

#ifndef __cplusplus
}
#endif
```

Phần ngữ pháp `extern "c"` được thêm vào để dùng đồng thời trên cả ngôn ngữ C và C++. Nội dung của phần lõi sẽ định nghĩa hàm `number_add_mod` để được export.

Sau đó chúng ta tạo ra file `_test_main.c` để kiểm tra việc sinh ra C static library (bên dưới là phần prefix dùng cho việc xây dựng C static library để bỏ qua file đó).

```
#include "number.h"

#include <stdio.h>

int main() {
    int a = 10;
    int b = 5;
    int c = 12;

    int x = number_add_mod(a, b, c);
    printf("(%d%d)%d = %d\n", a, b, c, x);
```

```
    return 0;  
}
```

Biên dịch và chạy chúng với những lệnh sau:

```
$ gcc -o a.out _test_main.c number.a  
$ ./a.out
```

2.9.4. Exporting thư viện C động

Quá trình exporting một thư viện động bằng CGO sẽ tương tự như một thư viện tĩnh, ngoại trừ build mode sẽ thay đổi c-shared và output file name được đổi thành `number.so`.

```
$ go build -buildmode=c-shared -o number.so
```

Nội dung của file `_test_main.c` sẽ không thay đổi, sau đó biên dịch và chạy chúng với các lệnh sau:

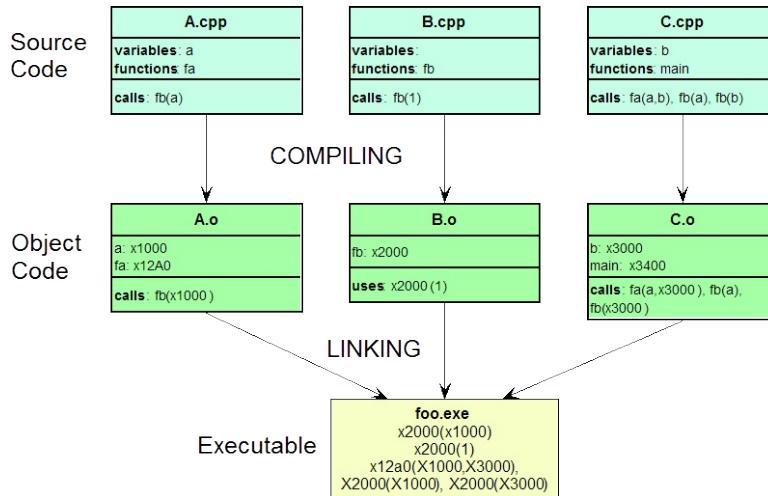
```
$ gcc -o a.out _test_main.c number.so  
$ ./a.out
```

Liên kết

- Phần tiếp theo: [Biên dịch và liên kết các tham số](#)
- Phần trước: [C++ Class Packaging](#)
- [Mục lục](#)

2.10. Biên dịch và liên kết các tham số

Biên dịch và liên kết các parameters là một vấn đề mà các lập trình viên C/C++ thường gặp phải. Trong phần này, chúng tôi sẽ giới thiệu ngắn các bước biên dịch và link parameters thường được dùng trong CGO.



2.10.1. Tham số biên dịch :

CGO cung cấp ba tham số `CFLAGS`/`CPPFLAGS`/`CXXFLAGS`. Trong đó:

- `CFLAGS` sẽ ứng với việc biên dịch ngôn ngữ C (.c).
- `CPPFLAGS` sẽ ứng với C/C++ (.c, .cc, .cpp, .cxx).
- `CXXFLAGS` ứng với C++ thuần (.cc, .cpp, *.cxx).

2.10.2. Tham số liên kết :

Trong CGO `${SRCDIR}` là một đường dẫn tuyệt đối trong thư mục hiện tại, các file định dạng của đối tượng C và C++ sau khi được biên dịch là như nhau, do đó `LDFLAGS` sẽ ứng với các C/C++ link parameters.

2.10.3. Lệnh pkg-config

Dùng công cụ `pkg-config` để thuận tiện cho việc compile và link parameter:

- Lệnh `#cgo pkg-config abc` để sinh ra lệnh compile và link parameter cho thư viện `abc`.
- Lệnh `pkg-config abc --cflags` sẽ sinh ra compiler parameters.
- Lệnh `pkg-config abc --libs` để sinh ra links parameters.

Có một số thư viện C/C++ non-standard mà `pkg-config` không hỗ trợ. Khi đó, chúng ta có thể hiện thực thủ công compilation và link parameter.

2.10.4. Chuỗi cài đặt package

Lệnh `go get package` sẽ liên kết các package phụ thuộc liên quan. Một chuỗi các package phụ thuộc sau như `pkgA -> pkgB -> pkgC -> pkgD`. Sau khi `go get A_package`, chúng sẽ get B, C, D package. Nếu việc build hỏng sau khi get `package_B`, nó sẽ dẫn tới việc hỏng toàn bộ chuỗi, kết quả là lệnh `get package_A` bị hỏng.

Sẽ có một số lý do cho việc hỏng chuỗi get, đây là một số lý do phổ biến:

- Một số hệ thống không hỗ trợ, việc biên dịch sẽ hỏng.
- Phụ thuộc vào `cgo`
 - User không có `gcc` được cài đặt sẵn.
 - Những thư viện độc lập không được cài đặt.
- Phụ thuộc vào `pkg-config`
 - Không được cài đặt trên windows.
 - Không có file `bc` được tìm thấy tương ứng.
- Phụ thuộc vào `custom pkg-config`: yêu cầu một số thiết lập thêm.
- Phụ thuộc vào `swig`: user chưa cài đặt `swig`, hoặc phiên bản không tương thích.

Liên kết

- Phản tiếp theo: [Lời nói thêm](#)
- Phản trước: [Thư viện tĩnh và động](#)
- [Mục lục](#)

2.11. Lời nói thêm

Thông qua CGO, bạn có thể kế thừa những phần mềm viết bằng C/C++ nổi tiếng hàng thập kỷ, có thể dùng Go để viết shared libraries trên giao diện C cho các hệ thống khác hoặc dùng mã nguồn được viết bởi Go để tích hợp tốt với các phần mềm có sẵn tạo thành một hệ sinh thái.

Bây giờ, các phần mềm chính thống thường được viết bởi ngôn ngữ C/C++. Sẽ rất nặng nề để có thể thay thế toàn bộ chúng bằng Go, do đó việc học cách chuyển đổi C/C++ sang Go bằng CGO là rất đáng cho dù bạn có là một nhà lập trình viên Go chuyên nghiệp.

Liên kết

- Phần tiếp theo: [Chương 3](#)
- Phần trước: [Biên dịch và liên kết các tham số](#)
- [Mục lục](#)

Chương 3: Remote Procedure Call



"Go is not meant to innovate programming theory. It's meant to innovate programming practice." – Samuel Tesla

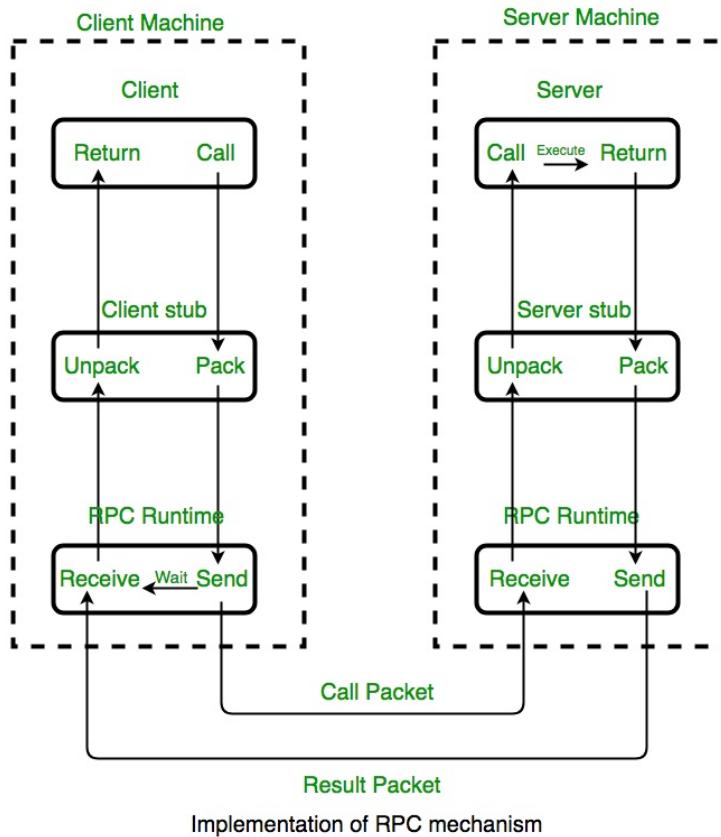
Remote Procedure Call là kỹ thuật cho phép chúng ta gọi hàm từ một process khác nằm cùng một máy hoặc ở hai máy tính khác nhau. Mục tiêu chính của phương pháp này là giúp lời gọi RPC tương tự như lời gọi thủ tục thông thường và ẩn đi phần xử lý kết nối mạng phức tạp. Chương này sẽ trình bày về cách sử dụng RPC, thiết kế RPC service, và hệ sinh thái RPC được xây dựng dựa trên nền tảng Protobuf của Google.

Liên kết

- Phần tiếp theo: [Giới thiệu về RPC](#)
- Phần trước: [Chương 2: Lời nói thêm](#)
- [Mục lục](#)

3.1. Giới thiệu về RPC

Remote Procedure Call (RPC) là phương pháp gọi hàm từ một máy tính ở xa để lấy về kết quả. Trong lịch sử phát triển của Internet, RPC đã trở thành một cơ sở hạ tầng không thể thiếu cũng giống như IPC (Inter Process Communication).



Mô hình giao tiếp client/server trong RPC

3.1.1 Chương trình RPC đầu tiên

Chương trình RPC đầu tiên được xây dựng dựa trên package `net/rpc` sẽ in ra chuỗi "Hello World" được tạo ra và trả về từ process khác:

server/main.go: chương trình phía server.

```
package main

import (
    "log"
    "net"
    "net/rpc"
)

// định nghĩa service struct
type HelloService struct{}`

// định nghĩa hàm service Hello, quy tắc:
// 1. Hàm service phải public (viết hoa)
```

```
// 2. Có hai tham số trong hàm
// 3. Tham số thứ hai phải kiểu con trỏ
// 4. Phải trả về kiểu error

func (p *HelloService) Hello(request string, reply *string) error {
    *reply = "Hello " + request
    // trả về error = nil nếu thành công
    return nil
}

func main() {
    rpc.RegisterName("HelloService", new(HelloService))
    // chạy rpc server trên port 1234
    listener, err := net.Listen("tcp", ":1234")
    // nếu có lỗi thì in ra
    if err != nil {
        log.Fatal("ListenTCP error:", err)
    }
    // vòng lặp để phục vụ nhiều client
    for {
        // accept một connection đến
        conn, err := listener.Accept()
        // in ra lỗi nếu có
        if err != nil {
            log.Fatal("Accept error:", err)
        }
        // phục vụ client trên một goroutine khác
        // để giải phóng main thread tiếp tục vòng lặp
        go rpc.ServeConn(conn)
    }
}
```

client/main.go: chương trình phía client.

```
package main

import (
    "fmt"
    "log"
    "net/rpc"
)

func main() {
    // kết nối đến rpc server
    client, err := rpc.Dial("tcp", "localhost:1234")
    // in ra lỗi nếu có
    if err != nil {
        log.Fatal("dialing:", err)
    }
    // biến chứa giá trị trả về sau lời gọi rpc
    var reply string
    // gọi rpc với tên service đã register
    err = client.Call("HelloService.Hello", "World", &reply)
    if err != nil {
        log.Fatal(err)
    }
    // in ra kết quả
    fmt.Println(reply)
}
```

Chạy server :

```
$ go run server/main.go
```

Chạy client :

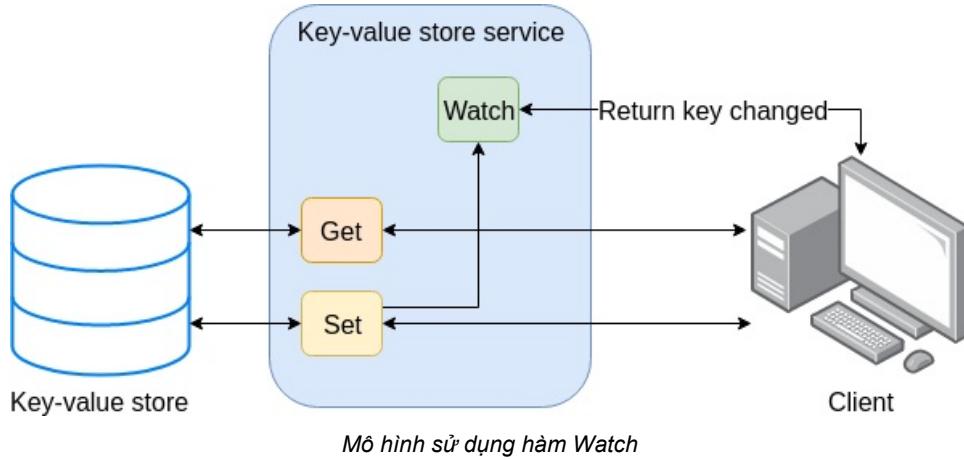
```
$ go run client/main.go
Hello World
```

Qua ví dụ trên, có thể thấy rằng việc dùng RPC trong Go thật sự đơn giản.

3.1.2 Hàm giám sát bằng RPC

Ta mong muốn khi hệ thống gặp phải những điều kiện nhất định thì có thể nhận về kết quả thông báo. Ví dụ sau sẽ xây dựng phương thức `watch` để làm điều đó.

Ý tưởng là giả lập một key-value store đơn giản, mỗi khi có sự thay đổi về value thì sẽ gửi về thông báo cho client.



Trước tiên xây dựng cơ sở dữ liệu Key-Value đơn giản thông qua RPC, xây dựng service như sau:

```
type KVStoreService struct {
    // map lưu trữ dữ liệu key value
    m      map[string]string

    // map chứa danh sách các hàm filter
    // được xác định trong mỗi lời gọi
    filter map[string]func(key string)

    // bảo vệ các thành phần khác khi được truy cập
    // và sửa đổi từ nhiều Goroutine cùng lúc
    mu     sync.Mutex
}

func NewKVStoreService() *KVStoreService {
    return &KVStoreService{
        m:      make(map[string]string),
        filter: make(map[string]func(key string)),
    }
}
```

Tiếp theo là phương thức Get và Set:

```
func (p *KVStoreService) Get(key string, value *string) error {
    p.mu.Lock()
    defer p.mu.Unlock()

    if v, ok := p.m[key]; ok {
        *value = v
        return nil
    }
}
```

```

        return fmt.Errorf("not found")
    }

func (p *KVStoreService) Set(kv [2]string, reply *struct{}) error {
    p.mu.Lock()
    defer p.mu.Unlock()

    key, value := kv[0], kv[1]

    if oldValue := p.m[key]; oldValue != value {
        // hàm filter được gọi khi value tương ứng
        // với key bị sửa đổi
        for _, fn := range p.filter {
            fn(key)
        }
    }

    p.m[key] = value
    return nil
}

```

Các filter sẽ được cung cấp trong phương thức `Watch`:

```

// Watch trả về key mỗi khi nhận thấy có thay đổi
func (p *KVStoreService) Watch(timeoutSecond int, keyChanged *string) error {
    // id là một string ghi lại thời gian hiện tại
    id := fmt.Sprintf("watch-%s-%03d", time.Now(), rand.Int())

    // buffered channel chứa key
    ch := make(chan string, 10)

    // filter để theo dõi key thay đổi
    p.mu.Lock()
    p.filter[id] = func(key string) { ch <- key }
    p.mu.Unlock()

    select {
    // trả về timeout sau một khoảng thời gian
    case <-time.After(time.Duration(timeoutSecond) * time.Second):
        return fmt.Errorf("timeout")
    case key := <-ch:
        *keyChanged = key
        return nil
    }

    return nil
}

```

Quá trình đăng ký và khởi động service `KVStoreService` bạn có thể xem lại phần trước. Hãy xem cách sử dụng phương thức `Watch` từ client:

```

func doClientWork(client *rpc.Client) {
    // khởi chạy một Goroutine riêng biệt để giám sát khóa thay đổi
    go func() {
        var keyChanged string
        // lời gọi `watch` synchronous sẽ block cho đến khi
        // có khóa thay đổi hoặc timeout
        err := client.Call("KVStoreService.Watch", 30, &keyChanged)
        if err != nil {
            log.Fatal(err)
        }
        fmt.Println("watch:", keyChanged)
    }()
}

```

```

err := client.Call(
    // giá trị KV được thay đổi bằng phương thức `Set`
    "KVStoreService.Set", [2]string{"abc", "abc-value"},
    new(struct{}),
)
// set lại lần nữa để giá trị value của key 'abc' thay đổi
err = client.Call(
    "KVStoreService.Set", [2]string{"abc", "another-value"},
    new(struct{}),
)
if err != nil {
    log.Fatal(err)
}

time.Sleep(time.Second*3)
}

```

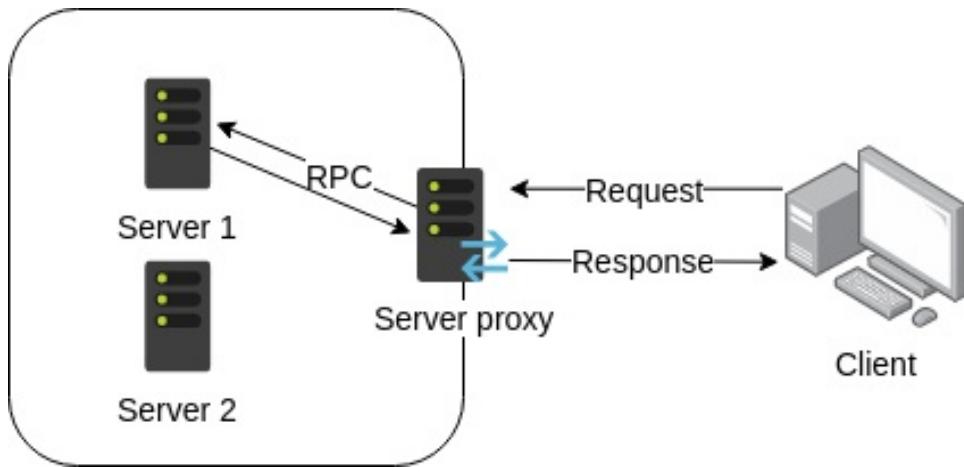
Kết quả nhận được ở client:

```
watch: abc
```

Server sẽ trả về key đã thay đổi (thông qua phương thức `watch`) cho client. Bằng cách này chúng ta có thể giám sát việc thay đổi trạng thái của key từ phía người gọi.

3.1.3 Reverse RPC

RPC thông thường chỉ do client gọi tới, server mới gửi lại phản hồi. Nhưng có một số trường hợp đặc biệt mà ta muốn server đóng vai trò người gọi (caller) để gọi cho client khi cần thiết. Mô hình này tương tự với reverse proxy: trong mạng nội bộ các service của chúng ta gọi nhau bằng RPC và không cho phép bên ngoài gọi trực tiếp vào đây. Khi đó cần một server đứng trung gian gọi tới các service trong nội bộ và gửi lại phản hồi cho các yêu cầu từ client bên ngoài.



Mô hình dùng reverse proxy

Lúc này **Server Proxy** đóng vai trò như server còn **Server 1** đóng vai trò như client trong một lời gọi RPC, hoặc ta có thể gọi quá trình này là Reverse RPC.

Đầu tiên client kết nối tới Server proxy bằng TCP, Server proxy sẽ chủ động kết nối với các Server trong nội bộ bằng RPC để nhận về kết quả và trả về cho client.

Sau đây là mã nguồn để khởi động một reverse RPC service:

server/main.go

```

func main() {
    rpc.Register(new(HelloService))

    for {
        // chủ động gọi tới client
        conn, _ := net.Dial("tcp", "localhost:1234")
        if conn == nil {
            time.Sleep(time.Second)
            continue
        }

        rpc.ServeConn(conn)
        conn.Close()
    }
}

```

Reverse RPC service sẽ không còn cung cấp service lắng nghe TCP, thay vào đó nó sẽ chủ động kết nối với server TCP của client. RPC service sau đó được cung cấp trên mỗi liên kết TCP được thiết lập.

RPC client cần cung cấp một service TCP có địa chỉ công khai để chấp nhận request từ RPC server:

client/main.go

```

func main() {
    // listen trên port 1234 chờ server gọi
    listener, err := net.Listen("tcp", ":1234")
    if err != nil {
        log.Fatal("ListenTCP error:", err)
    }

    clientChan := make(chan *rpc.Client)

    go func() {
        for {
            conn, err := listener.Accept()
            if err != nil {
                log.Fatal("Accept error:", err)
            }

            // khi mỗi đường link được thiết lập, đối tượng
            // RPC client được khởi tạo dựa trên link đó và
            // gửi tới client channel
            clientChan <- rpc.NewClient(conn)
        }
    }()
}

doClientWork(clientChan)
}

```

Client thực hiện lời gọi RPC trong hàm `doClientWork`:

```

func doClientWork(clientChan <-chan *rpc.Client) {
    // nhận vào đối tượng RPC client từ channel
    client := <-clientChan

    // đóng kết nối với client trước khi hàm exit
    defer client.Close()

    var reply string

    // thực hiện lời gọi rpc bình thường
    err := client.Call("HelloService.Hello", "hello", &reply)
    if err != nil {
        log.Fatal(err)
    }
}

```

```
    fmt.Println(reply)
}
```

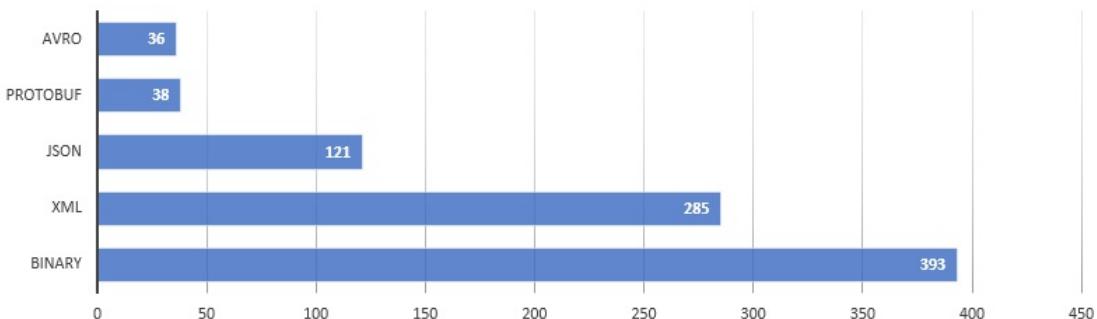
Liên kết

- Phần tiếp theo: [Protobuf](#)
- Phần trước: [Chương 2](#)
- [Mục lục](#)

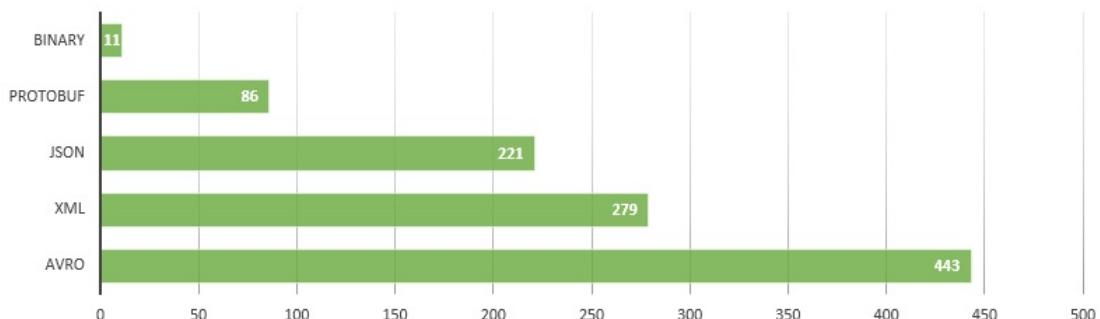
3.2. Protobuf

Protobuf hay Protocols Buffer là một ngôn ngữ dùng để mô tả các cấu trúc dữ liệu, chúng ta dùng protoc để biên dịch chúng thành mã nguồn của các ngôn ngữ lập trình khác nhau có chức năng serialize và deserialize các cấu trúc dữ liệu này thành dạng binary stream. So với dạng XML hoặc JSON thì dữ liệu đó nhỏ gọn gấp 3-10 lần và được xử lý rất nhanh.

Size comparison in bytes



Serialize/deserialize speed comparison in ms



Xem thêm: [Benchmarking Protocol Buffers, JSON and XML in Go](#).

Bạn đọc có thể cài đặt và làm quen với các ví dụ Protobuf trên [trang chủ](#) trước khi đi vào nội dung chính.

3.2.1 Kết hợp Protobuf với RPC

Đầu tiên chúng ta tạo file `hello.proto` chứa kiểu String được dùng trong RPC HelloService.

hello.proto:

```
// phiên bản proto3
syntax = "proto3";
// tên package được sinh ra
package main;
// message là một đơn vị dữ liệu trong Protobuf
message String {
    // chuỗi string được truyền vào hàm RPC
    string value = 1;
}
```

Để sinh ra mã nguồn Go từ file `hello.proto` ở trên, đầu tiên là cài đặt bộ biên dịch `protoc` qua liên kết [ở đây](#), sau đó là cài đặt một plugin cho Go thông qua lệnh:

```
$ go get github.com/golang/protobuf/protoc-gen-go
```

Chúng ta sẽ sinh ra mã nguồn Go bằng lệnh sau:

```
$ protoc --go_out=. hello.proto
// Trong đó,
// protoc: chương trình sinh mã nguồn
// go_out: chỉ cho protoc tải plugin protoc-gen-go, (cũng có java_out, python_out,...)
// --go_out=: sinh ra mã nguồn tại thư mục hiện tại
// hello.proto: file Protobuf
```

Sẽ có một file `hello.pb.go` được sinh ra, trong đó cấu trúc String được định nghĩa là:

hello.pb.go:

```
type String struct {
    Value string `protobuf:"bytes,1,opt,name=value" json:"value,omitempty"`
    //...
}

func (m *String) Reset()           { *m = String{} }
func (m *String) String() string { return proto.CompactTextString(m) }
func (*String) ProtoMessage()     {}
func (*String) Descriptor() ([]byte, []int) {
    return fileDescriptor_hello_069698f99dd8f029, []int{0}
}
//...
func (m *String) GetValue() string {
    if m != nil {
        return m.Value
    }
    return ""
}
//...
```

Ở phần 3.1 chúng ta đã xây dựng một RPC HelloService đơn giản dựa trên thư viện chuẩn `net/rpc` có kiểu dữ liệu request, reply do người dùng tự định nghĩa, bây giờ dựa trên kiểu String mới được sinh ra từ Protobuf, chúng ta có thể viết lại RPC HelloService như sau:

hello.go:

```
// RPC struct
type HelloService struct{}
// định nghĩa hàm Hello RPC, với tham số là kiểu String vừa định nghĩa trong Protobuf
func (p *HelloService) Hello(request *String, reply *String) error {
    // các hàm như .GetValue() đã được tạo ra trong file hello.pb.go
    reply.Value = "Hello, " + request.GetValue()
    // trả về nil khi thành công
    return nil
}
```

Chúng ta vẫn phải tự xây dựng hàm `Hello(request, reply)` bằng cách tự viết. Khi sử dụng Protobuf chúng ta có thể tự định nghĩa luôn service mình có những hàm rpc nào, nhận vào request và trả về reply ra sao. Chúng ta định nghĩa HelloService trong file proto như sau:

hello.proto

```
// ...
// định nghĩa service
service HelloService {
```

```
// định nghĩa lời gọi hàm RPC
rpc Hello (String) returns (String);
}
```

Chúng ta cần có một plugin để sinh ra mã nguồn service tương ứng với định nghĩa ở trên. Hiện nay Google đã phát triển bộ [gRPC plugin](#) giúp tạo ra mã nguồn tương ứng với file proto. Ở phần dưới sẽ trình bày cách xây dựng một plugin dựa trên mã nguồn gRPC plugin, chi tiết về gRPC chúng tôi sẽ đề cập ở các phần sau.

3.2.2 Viết plugin sinh mã nguồn RPC service

Từ mã nguồn [gRPC plugin](#), chúng ta có thể thấy hàm `generator.RegisterPlugin` được dùng để đăng ký plugin đó, Interface của một plugin sẽ như sau:

```
type Plugin interface {
    // Name() trả về tên của plugin.
    Name() string
    // Init() được gọi sau khi data structures built xong
    // và trước khi quá trình sinh code bắt đầu.
    Init(g *Generator)
    // Generate() là hàm sinh ra mã nguồn vào file
    Generate(file *FileDescriptor)
    // Hàm này được gọi sau khi Generate().
    GenerateImports(file *FileDescriptor)
}
```

Do đó, chúng ta có thể xây dựng một plugin mang tên `netrpcPlugin` để sinh ra mã nguồn RPC service cho Go từ file Protobuf.

```
import (
    // import gói thư viện để sinh ra plugin
    "github.com/golang/protobuf/protoc-gen-go/generator"
)
// định nghĩa struct netrpcPlugin xây dựng interface Plugin
type netrpcPlugin struct{ *generator.Generator }
// định nghĩa Name() function
func (p *netrpcPlugin) Name() string           { return "netrpc" }
// định nghĩa Init() function
func (p *netrpcPlugin) Init(g *generator.Generator) { p.Generator = g }
// định nghĩa GenerateImports()
func (p *netrpcPlugin) GenerateImports(file *generator.FileDescriptor) {
    if len(file.Service) > 0 {
        p.genImportCode(file)
    }
}
// định nghĩa Generate()
func (p *netrpcPlugin) Generate(file *generator.FileDescriptor) {
    for _, svc := range file.Service {
        p.genServiceCode(svc)
    }
}
```

Hiện tại, phương thức `genImportCode` và `genServiceCode` tạm thời như sau:

```
func (p *netrpcPlugin) genImportCode(file *generator.FileDescriptor) {
    p.P("// TODO: import code")
}

func (p *netrpcPlugin) genServiceCode(svc *descriptor.ServiceDescriptorProto) {
    p.P("// TODO: service code, Name = " + svc.GetName())
}
```

Để sử dụng plugin, chúng ta cần phải đăng ký plugin đó với hàm `generator.RegisterPlugin`, chúng có thể được xây dựng chúng nhờ vào hàm `init()`.

```
func init() {
    generator.RegisterPlugin(new(netrpcPlugin))
}
```

Bởi vì trong ngôn ngữ Go, package chỉ được import tĩnh, chúng ta không thể thêm plugin mới vào plugin đã có sẵn là `protoc-gen-go`. Chúng ta sẽ `re-clone` lại hàm main để build lại `protoc-gen-go`.

```
package main

import (
    "io/ioutil"
    "os"
    // import các package cần thiết
    "github.com/golang/protobuf/proto"
    "github.com/golang/protobuf/protoc-gen-go/generator"
)
// bắt đầu hàm main
func main() {
    // sinh ra một đối tượng plugin mới
    g := generator.New()
    // đọc lệnh từ console vào biến data
    data, err := ioutil.ReadAll(os.Stdin)
    // in ra lỗi nếu có
    if err != nil {
        g.Error(err, "reading input")
    }
    // unmarshal data thành cấu trúc Request
    if err := proto.Unmarshal(data, g.Request);
    // in ra lỗi nếu có
    err != nil {
        g.Error(err, "parsing input proto")
    }
    // kiểm tra xem tên file có hợp lệ không
    if len(g.Request.FileToGenerate) == 0 {
        g.Fail("no files to generate")
    }
    // đăng ký các tham số
    g.CommandLineParameters(g.Request.GetParameter())
    g.WrapTypes()
    // thiết lập tên package
    g.SetPackageName()
    g.BuildTypeNameMap()

    // sinh ra các file mã nguồn
    g.GenerateAllFiles()

    // trả về kết quả
    data, err = proto.Marshal(g.Response)
    if err != nil {
        g.Error(err, "failed to marshal output proto")
    }
    // ghi kết quả ra màn hình
    _, err = os.Stdout.Write(data)
    // in ra lỗi nếu có
    if err != nil {
        g.Error(err, "failed to write output proto")
    }
}
```

Để tránh việc trùng tên với `protoc-gen-go` plugin, chúng ta sẽ đặt tên cho plugin mới là `protoc-gen-go-netrpc`, và định biên dịch lại `hello.proto` với lệnh sau:

```
$ protoc --go-netrpc_out=plugins=netrpc:. hello.proto
```

Tham số `--go-netrpc_out` sẽ nói cho bộ biên dịch protoc biết là nó phải tải một plugin với tên gọi là `protoc-gen-go-netrpc`. Bây giờ, tiếp tục phát triển `netrpcPlugin` plugin với mục tiêu cuối cùng là sinh ra lớp Interface RPC. Đầu tiên chúng ta sẽ phải xây dựng `genImportCode`:

```
func (p *netrpcPlugin) genImportCode(file *generator.FileDescriptor) {
    p.P(`import "net/rpc"`)
}
```

Chúng ta sẽ định nghĩa kiểu `ServiceSpec` được mô tả như là thông tin thêm vào của service.

```
type ServiceSpec struct {
    // Tên của service
    ServiceName string
    // Danh sách cách Service method
    MethodList []ServiceMethodSpec
}

type ServiceMethodSpec struct {
    MethodName     string
    InputTypeName string
    OutputTypeName string
}
```

Chúng ta sẽ tạo ra một phương thức `buildServiceSpec`, nó sẽ parse thông tin thêm vào service được định nghĩa trong `ServiceSpec` cho mỗi service.

```
// phương thức buildServiceSpec
func (p *netrpcPlugin) buildServiceSpec(
    // tham số truyền vào thuộc kiểu ServiceDescriptorProto
    // mô tả thông tin về service
    svc *descriptor.ServiceDescriptorProto,
) *ServiceSpec {
    // khởi tạo đối tượng
    spec := &ServiceSpec{
        // svc.GetName(): lấy tên service được định nghĩa ở Protobuf file
        // sau đó chuyển đổi chúng về style CamelCase
        ServiceName: generator.CamelCase(svc.GetName()),
    }
    // mới mỗi phương thức RPC, ta thêm một cấu trúc tương ứng vào danh sách
    for _, m := range svc.Method {
        spec.MethodList = append(spec.MethodList, ServiceMethodSpec{
            // m.GetName(): lấy tên phương thức
            MethodName:     generator.CamelCase(m.GetName()),
            // m.GetInputType(): lấy kiểu dữ liệu tham số đầu vào
            InputTypeName: p.TypeName(p.ObjectNamed(m.GetInputType())),
            OutputTypeName: p.TypeName(p.ObjectNamed(m.GetOutputType())),
        })
    }
    // trả về cấu trúc trên
    return spec
}
```

Sau đó chúng ta sẽ sinh ra mã nguồn của service dựa trên thông tin mô tả đó, được xây dựng bởi phương thức `buildServiceSpec` :

```
func (p *netrpcPlugin) genServiceCode(svc *descriptor.ServiceDescriptorProto) {
    // gọi hàm được định nghĩa ở trên
    spec := p.buildServiceSpec(svc)
    // buf là biến chứa dữ liệu
```

```

var buf bytes.Buffer
// dùng tmplService cho việc sinh mã nguồn
t := template.Must(template.New("").Parse(tmplService))
// thực thi việc sinh mã nguồn
err := t.Execute(&buf, spec)
// in ra lỗi nếu có
if err != nil {
    log.Fatal(err)
}
// ghi buf.String() vào file
p.P(buf.String())
}

```

Chúng ta mong đợi vào mã nguồn cuối cùng được sinh ra như sau:

```

type HelloServiceInterface interface {
    Hello(in String, out *String) error
}

func RegisterHelloService(srv *rpc.Server, x HelloService) error {
    if err := srv.RegisterName("HelloService", x); err != nil {
        return err
    }
    return nil
}

type HelloServiceClient struct {
    *rpc.Client
}

var _ HelloServiceInterface = (*HelloServiceClient)(nil)

func DialHelloService(network, address string) (*HelloServiceClient, error) {
    c, err := rpc.Dial(network, address)
    if err != nil {
        return nil, err
    }
    return &HelloServiceClient{Client: c}, nil
}

func (p *HelloServiceClient) Hello(in String, out *String) error {
    return p.Client.Call("HelloService.Hello", in, out)
}

```

Để làm được như vậy, template của chúng ta được viết như sau:

```

const tmplService = `

{{$root := .}}


type {{.ServiceName}}Interface interface {
    {{- range $_, $m := .MethodList}}
    {{$m.MethodName}}(*{{$m.InputTypeName}}, *{{$m.OutputTypeName}}) error
    {{- end}}
}

func Register{{.ServiceName}}(
    srv *rpc.Server, x {{.ServiceName}}Interface,
) error {
    if err := srv.RegisterName("{{.ServiceName}}", x); err != nil {
        return err
    }
    return nil
}

type {{.ServiceName}}Client struct {
    *rpc.Client
}

```

```

}

var _ {{.ServiceName}}Interface = (*{{.ServiceName}}Client)(nil)

func Dial{{.ServiceName}}(network, address string) (
    *{{.ServiceName}}Client, error,
) {
    c, err := rpc.Dial(network, address)
    if err != nil {
        return nil, err
    }
    return &{{.ServiceName}}Client{Client: c}, nil
}

{{range $_, $m := .MethodList}}
func (p *{{$root.ServiceName}}Client) {{$m.MethodName}}(
    in *{{$m.InputTypeName}}, out *{{$m.OutputTypeName}},
) error {
    return p.Client.Call("{{$root.ServiceName}}.{{$m.MethodName}}", in, out)
}
{{end}}

```

Khi plugin mới của protoc được hoàn thành, mã nguồn có thể được sinh ra mỗi khi RPC service thay đổi trong `hello.proto` file. Chúng ta có thể điều chỉnh hoặc thêm nội dung của mã nguồn được sinh ra bằng việc cập nhật template plugin.

Liên kết

- Phản tiếp theo: [Làm quen với gRPC](#)
- Phản trước: [Giới thiệu về RPC](#)
- [Mục lục](#)

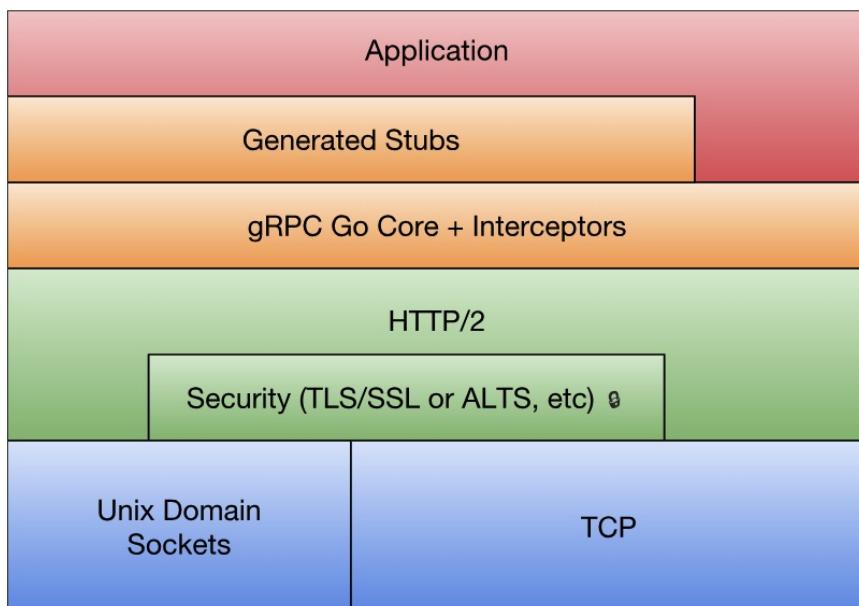
3.3 Làm quen với gRPC



gRPC là một framework RPC Open source đa ngôn ngữ được Google phát triển dựa trên [Protobuf](#) và giao thức [HTTP/2](#). Phần này sẽ giới thiệu một số cách sử dụng gRPC để xây dựng service đơn giản.

3.3.1 Kiến trúc gRPC

Kiến trúc gRPC trong Go:



gRPC technology stack

Lớp dưới cùng là giao thức TCP hoặc [Unix Socket](#). Ngay trên đây là phần hiện thực của giao thức [HTTP/2](#). Thư viện gRPC core cho Go được xây dựng ở lớp kế. Stub code được tạo ra bởi chương trình thông qua plug-in gRPC giao tiếp với thư viện gRPC core.

3.3.2 Làm quen với gRPC

Từ quan điểm của Protobuf, gRPC không gì khác hơn là một trình tạo code cho interface service.

Tạo file `hello.proto` và định nghĩa interface `HelloService`:

```

syntax = "proto3";

package main;

message String {
    string value = 1;
}
  
```

```
service HelloService {
    rpc Hello (String) returns (String);
}
```

Tạo gRPC code sử dụng hàm dựng sẵn trong gRPC plugin từ protoc-gen-go:

```
$ protoc --go_out=plugins=grpc:. hello.proto
```

gRPC plugin tạo ra các interface khác nhau cho server và client:

```
type HelloServiceServer interface {
    Hello(context.Context, *String) (*String, error)
}

type HelloServiceClient interface {
    Hello(context.Context, *String, ...grpc.CallOption) (*String, error)
}
```

gRPC cung cấp hỗ trợ context cho mỗi lệnh gọi phương thức thông qua tham số `context.Context`. Khi client gọi phương thức, nó có thể cung cấp thông tin context bổ sung thông qua các tham số tùy chọn của kiểu `grpc.CallOption`.

Chúng ta sử dụng struct `HelloServiceImpl` để thực hiện service `HelloService` dựa trên interface `HelloServiceServer`:

```
type HelloServiceImpl struct{}

func (p *HelloServiceImpl) Hello(
    ctx context.Context, args *String,
) (*String, error) {
    reply := &String{Value: "hello:" + args.GetValue()}
    return reply, nil
}
```

Quá trình khởi động của gRPC service tương tự như quá trình khởi động RPC service của thư viện chuẩn:

```
func main() {
    // khởi tạo một đối tượng gRPC service
    grpcServer := grpc.NewServer()

    // đăng ký service với grpcServer (của gRPC plugin)
    RegisterHelloServiceServer(grpcServer, new(HelloServiceImpl))

    // cung cấp gRPC service trên port `1234`
    lis, err := net.Listen("tcp", ":1234")
    if err != nil {
        log.Fatal(err)
    }
    grpcServer.Serve(lis)
}
```

Tiếp theo bạn đã có thể kết nối tới gRPC service từ client:

```
func main() {
    // thiết lập kết nối với gRPC service
    conn, err := grpc.Dial("localhost:1234", grpc.WithInsecure())
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
```

```
// xây dựng đối tượng `HelloServiceClient` dựa trên kết nối đã thiết lập
client := NewHelloServiceClient(conn)
reply, err := client.Hello(context.Background(), &String{Value: "hello"})
if err != nil {
    log.Fatal(err)
}
fmt.Println(reply.GetValue())
}
```

Có một sự khác biệt giữa gRPC và framework RPC của thư viện chuẩn: gRPC không hỗ trợ gọi asynchronous. Tuy nhiên, ta có thể chia sẻ kết nối HTTP/2 trên nhiều Goroutines, vì vậy có thể mô phỏng các lời gọi bất đồng bộ bằng cách block các lời gọi trong Goroutine khác.

3.3.3 gRPC streaming

RPC là lời gọi hàm từ xa, vì vậy các tham số hàm và giá trị trả về của mỗi cuộc gọi không thể quá lớn, nếu không thời gian phản hồi của mỗi lời gọi sẽ bị ảnh hưởng nghiêm trọng. Do đó, các lời gọi phương thức RPC truyền thống không phù hợp để tải lên và tải xuống trong trường hợp khối lượng dữ liệu lớn. Để khắc phục điểm này, framework gRPC cung cấp các chức năng stream cho phía server và client tương ứng.

Ta viết thêm phương thức channel hỗ trợ luồng hai chiều (Bidirect Streaming) trong `HelloService`:

```
service HelloService {
    rpc Hello (String) returns (String);

    // nhận vào tham số một stream và trả về giá trị là một stream.
    rpc Channel (stream String) returns (stream String);
}
```

Tạo lại code để thấy định nghĩa mới được thêm vào phương thức kiểu channel trong interface:

```
type HelloServiceServer interface {
    Hello(context.Context, *String) (*String, error)

    // tham số kiểu HelloService_ChannelServer được sử dụng
    // để liên lạc hai chiều với client.
    Channel(HelloService_ChannelServer) error
}

type HelloServiceClient interface {
    Hello(ctx context.Context, in *String, opts ...grpc.CallOption) (
        *String, error,
    )

    // trả về giá trị trả về thuộc kiểu `HelloService_ChannelClient`
    // có thể được sử dụng để liên lạc hai chiều với server.
    Channel(ctx context.Context, opts ...grpc.CallOption) (
        HelloService_ChannelClient, error,
    )
}
```

`HelloService_ChannelServer` và `HelloService_ChannelClient` thuộc interface:

```
type HelloService_ChannelServer interface {
    Send(*String) error
    Recv() (*String, error)
    grpc.ServerStream
}
```

```
type HelloService_ChannelClient interface {
    Send(*String) error
    Recv() (*String, error)
    grpc.ClientStream
}
```

Có thể thấy các interface hỗ trợ server và client stream đều có định nghĩa phương thức `Send` và `Recv` cho giao tiếp hai chiều của dữ liệu streaming.

Bây giờ ta có thể xây dựng các streaming service:

```
func (p *HelloServiceImpl) Channel(stream HelloService_ChannelServer) error {
    for {
        // Server nhận dữ liệu được gửi từ client
        // trong vòng lặp.
        args, err := stream.Recv()
        if err != nil {
            // Nếu gặp `io.EOF`, client stream sẽ đóng.
            if err == io.EOF {
                return nil
            }
            return err
        }

        reply := &String{Value: "hello:" + args.GetValue()}

        // Dữ liệu trả về được gửi đến client
        // thông qua stream và việc gửi nhận
        // dữ liệu stream hai chiều là hoàn toàn độc lập
        err = stream.Send(reply)
        if err != nil {
            return err
        }
    }
}
```

Client cần gọi phương thức `Channel` để lấy đối tượng stream trả về:

```
stream, err := client.Channel(context.Background())
if err != nil {
    log.Fatal(err)
}
```

Ở phía client ta thêm vào các thao tác gửi và nhận trong các Goroutine riêng biệt. Trước hết là để gửi dữ liệu tới server:

```
go func() {
    for {
        if err := stream.Send(&String{Value: "hi"}); err != nil {
            log.Fatal(err)
        }
        time.Sleep(time.Second)
    }()
}
```

Kế đến là nhận dữ liệu trả về từ server trong vòng lặp:

```
for {
    reply, err := stream.Recv()
    if err != nil {
        if err == io.EOF {
            break
        }
    }
}
```

```

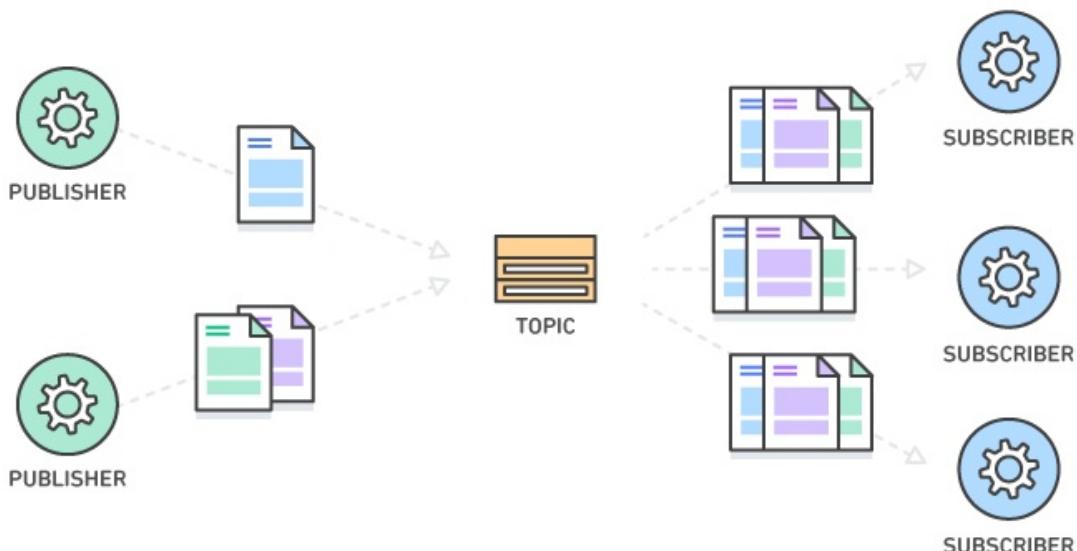
        }
        log.Fatal(err)
    }
    fmt.Println(reply.GetValue())
}

```

Bạn có thể xem code cụ thể tại [đây](#)

3.3.4 Mô hình Publishing - Subscription

Trong phần trước ta đã xây dựng phiên bản đơn giản của phương thức `Watch` dựa trên thư viện RPC sẵn của Go. Ý tưởng đó có thể sử dụng cho hệ thống publish-subscribe, nhưng bởi vì RPC thiếu đi cơ chế streaming nên nó chỉ có thể trả về 1 kết quả trong 1 lần. Trong chế độ publish-subscribe, hành động publish đưa ra bởi *caller* giống với lời gọi hàm thông thường, trong khi subscriber bị động thì giống với *receiver* trong gRPC client stream một chiều. Bây giờ ta có thể thử xây dựng một hệ thống publish - subscribe dựa trên đặc điểm stream của gRPC.



Nhắc lại mô hình Pub/Sub

Publishing - Subscription là một mẫu thiết kế thông dụng và đã có nhiều cài đặt của mẫu thiết kế này trong cộng đồng Open source. Đoạn code sau đây xây dựng cơ chế publish - subscription dựa trên package pubsub:

```

import (
    "github.com/moby/moby/pkg/pubsub"
    "time"
    "fmt"
    "strings"
)

func main() {
    // xây dựng một đối tượng để publish
    p := pubsub.NewPublisher(100*time.Millisecond, 10)

    // subscribe các topic "go"
    go := p.SubscribeTopic(func(v interface{}) bool {
        if key, ok := v.(string); ok {
            if strings.HasPrefix(key, "go:") {
                return true
            }
        }
    })
}

```

```

        }
        return false
    })

    // subscribe các topic "docker"
    docker := p.SubscribeTopic(func(v interface{}) bool {
        if key, ok := v.(string); ok {
            if strings.HasPrefix(key, "docker:") {
                return true
            }
        }
        return false
    })

    go p.Publish("hi")
    go p.Publish("go: https://go.org")
    go p.Publish("docker: https://www.docker.com/")
    time.Sleep(1)

    go func() {
        fmt.Println("go topic:", <-go)
    }()
    go func() {
        fmt.Println("docker topic:", <-docker)
    }()
}

<-make(chan bool)
}

```

Giờ chúng ta thử cung cấp một hệ thống publishing-subscription khác mạng dựa trên gRPC và pubsub package. Đầu tiên định nghĩa một service publish subscription interface bằng protobuf:

```

service PubsubService {
    // phương thức RPC thông thường
    rpc Publish (String) returns (String);

    // service server streaming
    rpc Subscribe (String) returns (stream String);
}

```

gRPC plugin sẽ tạo ra interface tương ứng cho server và client:

```

type PubsubServiceServer interface {
    Publish(context.Context, *String) (*String, error)
    Subscribe(*String, PubsubService_SubscribeServer) error
}
type PubsubServiceClient interface {
    Publish(context.Context, *String, ...grpc.CallOption) (*String, error)
    Subscribe(context.Context, *String, ...grpc.CallOption) (
        PubsubService_SubscribeClient, error,
    )
}

type PubsubService_SubscribeServer interface {
    Send(*String) error
    grpc.ServerStream
}

```

Bởi vì `Subscribe` là stream 1 chiều phía server nên chỉ có phương thức `Send` được tạo ra trong interface `HelloService_SubscribeServer`.

Sau đó có thể xây dựng các service publish và subscribe như sau:

```

type PubsubService struct {

```

```

    pub *pubsub.Publisher
}

func NewPubsubService() *PubsubService {
    return &PubsubService{
        pub: pubsub.NewPublisher(100*time.Millisecond, 10),
    }
}

```

Ké đến là các phương thức publishing và subscription:

```

func (p *PubsubService) Publish(
    ctx context.Context, arg *String,
) (*String, error) {
    p.pub.Publish(arg.GetValue())
    return &String{}, nil
}

func (p *PubsubService) Subscribe(
    arg *String, stream PubsubService_SubscribeServer,
) error {
    ch := p.pub.SubscribeTopic(func(v interface{}) bool {
        if key, ok := v.(string); ok {
            if strings.HasPrefix(key, arg.GetValue()) {
                return true
            }
        }
        return false
    })
    for v := range ch {
        if err := stream.Send(&String{Value: v.(string)}); err != nil {
            return err
        }
    }
    return nil
}

```

Hàm `main` cho phép đăng mới thông tin từ client tới server:

```

func main() {
    conn, err := grpc.Dial("localhost:1234", grpc.WithInsecure())
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    client := NewPubsubServiceClient(conn)

    _, err = client.Publish(
        context.Background(), &String{Value: "go: hello Go"},
    )
    if err != nil {
        log.Fatal(err)
    }
    _, err = client.Publish(
        context.Background(), &String{Value: "docker: hello Docker"},
    )
    if err != nil {
        log.Fatal(err)
    }
}

```

Chi tiết: [clientpub](#).

Sau đó có thể subscribe thông tin đó từ một client khác:

```
func main() {
    conn, err := grpc.Dial("localhost:1234", grpc.WithInsecure())
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    client := NewPubsubServiceClient(conn)
    stream, err := client.Subscribe(
        context.Background(), &String{Value: "go:"},
    )
    if err != nil {
        log.Fatal(err)
    }

    for {
        reply, err := stream.Recv()
        if err != nil {
            if err == io.EOF {
                break
            }
            log.Fatal(err)
        }

        fmt.Println(reply.GetValue())
    }
}
```

Chi tiết: [clientsub](#).

Cho đến giờ chúng ta đã xây dựng được service publishing và subscription khác mạng dựa trên gRPC. Trong phần kế tiếp chúng ta sẽ xét một số ứng dụng nâng cao hơn của Go trong gRPC.

Liên kết

- Phần tiếp theo: [Một số vấn đề khác của gRPC](#)
- Phần trước: [Protobuf](#)
- [Mục lục](#)

3.4 Một số vấn đề khác của gRPC

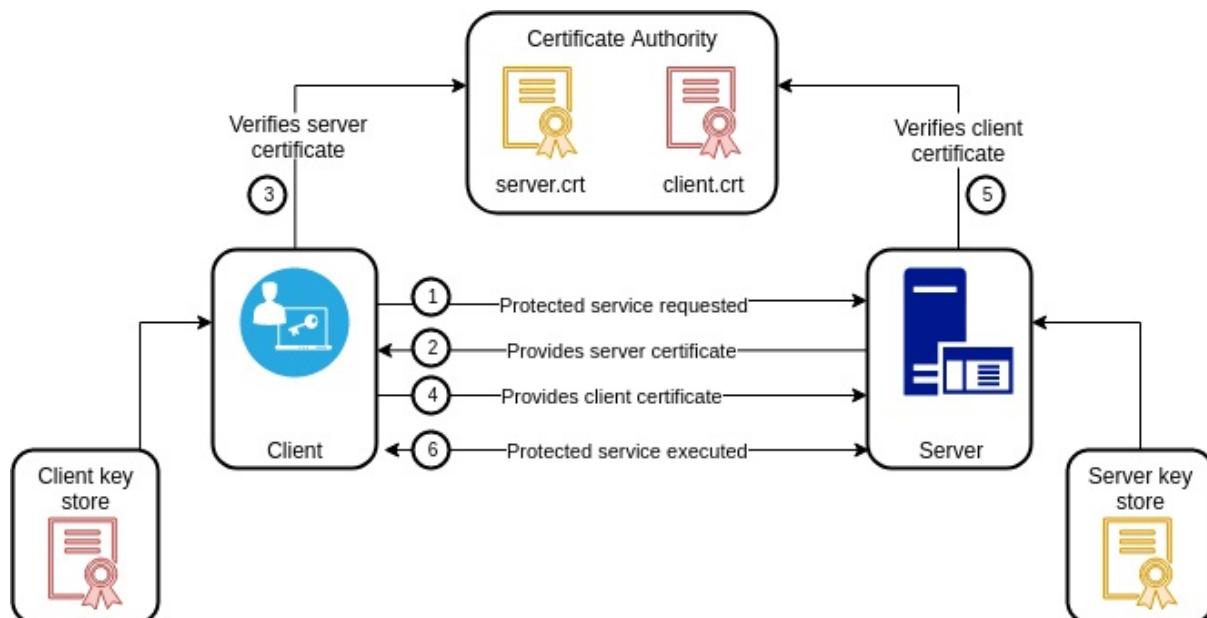
Để nâng cao khả năng bảo mật, ta có thể áp dụng chức năng xác thực vào hệ thống. Hai mức xác thực dùng trong RPC:

- Một là khi thiết lập kết nối giữa client - server (sử dụng chứng chỉ)
- Một là đối với mỗi lời gọi được thực hiện giữa client - server (sử dụng token).

Phần này sẽ ta sẽ tìm hiểu cách hiện thực các cơ chế xác thực này, sau đó là sử dụng interceptor cho request/response và gRPC kết hợp với web service.

3.4.1 Xác thực qua chứng chỉ (certificate)

gRPC được xây dựng dựa trên giao thức HTTP/2 và hỗ trợ TLS khá hoàn thiện. gRPC service trong chương trước chúng ta không hỗ trợ xác thực qua chứng chỉ, vì vậy client `grpc.WithInsecure()` có thể thông qua tùy chọn mà bỏ qua việc xác thực trong server được kết nối.



gRPC service không có xác thực chứng chỉ (certificate) sẽ phải giao tiếp hoàn toàn bằng plain-text với client và có nguy cơ cao bị giám sát bởi một bên thứ ba khác. Để khắc phục yếu điểm này, chúng ta có thể sử dụng mã hóa TLS trên server.

Đầu tiên tạo private key và certificate cho server và client riêng biệt bằng các lệnh sau:

```
$ openssl genrsa -out server.key 2048
$ openssl req -new -x509 -days 3650 \
    -subj "/C=GB/L=China/O=grpc-server/CN=server.grpc.io" \
    -key server.key -out server.crt

$ openssl genrsa -out client.key 2048
$ openssl req -new -x509 -days 3650 \
    -subj "/C=GB/L=China/O=grpc-client/CN=client.grpc.io" \
    -key client.key -out client.crt
```

Lệnh trên sẽ tạo ra 4 file: `server.key`, `server.crt`, `client.key` và `client.crt`. File private key có phần mở rộng `.key` và được cần được giữ bảo mật an toàn. File certificate có phần mở rộng `.crt` được hiểu như public key và không cần giữ bí mật.

Với certificate này ta có thể truyền nó vào tham số để bắt đầu một gRPC service:

```
func main() {
    // Khởi tạo đối tượng certificate từ file cho server
    creds, err := credentials.NewServerTLSFromFile("server.crt", "server.key")
    if err != nil {
        log.Fatal(err)
    }

    // Truyền certificate dưới dạng tham số
    // cho hàm khởi tạo một server
    server := grpc.NewServer(grpc.Creds(creds))

    ...
}
```

Server có thể được xác thực ở client dựa trên chứng chỉ của server và tên của nó:

```
func main() {
    // Client xác thực server bằng cách đưa vào
    // chứng chỉ CA root và tên của server
    creds, err := credentials.NewClientTLSFromFile(
        "server.crt", "server.grpc.io",
    )
    if err != nil {
        log.Fatal(err)
    }

    conn, err := grpc.Dial("localhost:5000",
        grpc.WithTransportCredentials(creds),
    )
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    ...
}
```

Khi client liên kết với server, trước tiên, nó sẽ yêu cầu chứng chỉ của server, sau đó sử dụng chứng chỉ CA root để xác minh chứng chỉ phía server mà nó nhận được.

Nếu chứng chỉ của client cũng được ký bởi CA root, server cũng có thể thực hiện xác thực chứng chỉ trên client. Ở đây ta sử dụng chứng chỉ CA root để ký chứng chỉ client:

```
$ openssl req -new \
    -subj "/C=GB/L=China/O=client/CN=client.io" \
    -key client.key \
    -out client.csr
$ openssl x509 -req -sha256 \
    -CA ca.crt -CAkey ca.key -CAcreateserial -days 3650 \
    -in client.csr \
    -out client.crt
```

Xem [Makefile](#)

Chứng chỉ root được cấu hình lúc khởi động server:

```

func main() {
    certificate, err := tls.LoadX509KeyPair("server.crt", "server.key")
    if err != nil {
        log.Fatal(err)
    }

    certPool := x509.NewCertPool()
    ca, err := ioutil.ReadFile("ca.crt")
    if err != nil {
        log.Fatal(err)
    }
    if ok := certPool.AppendCertsFromPEM(ca); !ok {
        log.Fatal("failed to append certs")
    }

    // Server cũng sử dụng hàm `credentials.NewTLS` để tạo chứng chỉ
    creds := credentials.NewTLS(&tls.Config{
        Certificates: []tls.Certificate{certificate},
        // Cho phép Client được xác thực
        ClientAuth:   tls.RequireAndVerifyClientCert,
        // Chọn chứng chỉ CA root thông qua ClientCA,
        // this is optional!
        ClientCAs:    certPool,
    })

    server := grpc.NewServer(grpc.Creds(creds))
    ...
}

```

Như vậy chúng ta đã xây dựng được một hệ thống gRPC đáng tin cậy để kết nối giữa Client và Server thông qua xác thực chứng chỉ từ cả 2 chiều.

3.4.2 Xác thực bằng token

Xác thực dựa trên chứng chỉ được mô tả ở trên là dành cho từng kết nối gRPC. Ngoài ra gRPC cũng hỗ trợ xác thực cho mỗi lệnh gọi gRPC, để việc quản lý quyền có thể thực hiện trên các kết nối khác nhau dựa trên user token.

Để hiện thực cơ chế xác thực cho từng phương thức gRPC, ta cần triển khai interface `grpc.PerRPCCredentials` :

```

type PerRPCCredentials interface {
    // GetRequestMetadata get request về metadata hiện tại,
    // refresh lại token. Hàm này nên được gọi từ transport layer
    // trên mỗi request và dữ liệu phải được điền vào header
    // hoặc trong context. Nếu status code được trả về,
    // nó sẽ được sử dụng làm status cho RPC.
    // uri là URI entry point của request.
    // ctx có thể dùng cho timeout và cancellation.
    GetRequestMetadata(ctx context.Context, uri ...string) (
        map[string]string,     error,
    )

    // RequireTransportSecurity thể hiện cho việc credentials
    // có yêu cầu kết nối bảo mật (TLS) không.
    // Nên là true để thông tin xác thực không có nguy cơ
    // bị xâm phạm và giả mạo.
    RequireTransportSecurity() bool
}

```

Ta có thể tạo ra struct `Authentication` để xác thực username và password:

```

type Authentication struct {
    User     string
    Password string
}

// trả về thông tin xác thực cục bộ gồm cả user và password.
func (a *Authentication) GetRequestMetadata(context.Context, ...string) (
    map[string]string, error,
) {
    return map[string]string{"user":a.User, "password": a.Password}, nil
}

// để code được đơn giản hơn nên `RequireTransportSecurity`
// không cần thiết.
func (a *Authentication) RequireTransportSecurity() bool {
    return false
}

```

Thông tin token có thể được truyền vào như tham số cho mỗi gRPC service được yêu cầu:

```

func main() {
    auth := Authentication{
        Login:    "gopher",
        Password: "password",
    }

    // đổi tượng `Authentication` được chuyển đổi thành tham số
    // của `grpc.Dial` bằng hàm `grpc.WithPerRPCCredentials`
    // Vì secure link không được kích hoạt nên cần phải
    // truyền vào `grpc.WithInsecure()` để bỏ qua bước
    // xác thực chúng chỉ bảo mật
    conn, err := grpc.Dial("localhost"+port, grpc.WithInsecure(), grpc.WithPerRPCCredentials(&auth))
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    ...
}

```

Kế đó trong mỗi phương thức của gRPC server, danh tính người dùng được xác thực bởi phương thức `Auth` của `Authentication`:

```

type grpcServer struct { auth *Authentication }

func (p *grpcServer) SomeMethod(
    ctx context.Context, in *HelloRequest,
) (*HelloReply, error) {
    if err := p.auth.Auth(ctx); err != nil {
        return nil, err
    }

    return &HelloReply{Message: "Hello " + in.Name}, nil
}

// phương thức thực hiện việc xác thực
func (a *Authentication) Auth(ctx context.Context) error {
    // meta information được lấy từ biến ngữ cảnh `ctx`
    md, ok := metadata.FromIncomingContext(ctx)
    if !ok {
        return fmt.Errorf("missing credentials")
    }

    var appid string
    var appkey string

```

```

if val, ok := md["login"]; ok { appid = val[0] }
if val, ok := md["password"]; ok { appkey = val[0] }

// Nếu xác thực thất bại sẽ trả về lỗi
if appid != a.Login || appkey != a.Password {
    return grpc.Errorf(codes.Unauthenticated, "invalid token")
}

return nil
}

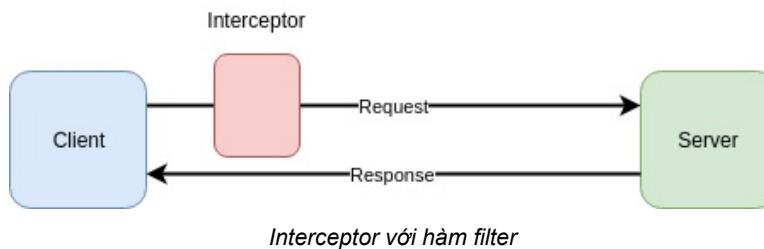
```

3.4.3 Interceptor

`Grpc.UnaryInterceptor` và `grpc.StreamInterceptor` trong gRPC hỗ trợ interceptor cho các phương thức thông thường và phương thức stream. Ở đây chúng ta hãy tìm hiểu về việc sử dụng interceptor cho phương thức thông thường.

Để sử dụng hàm interceptor `filter`, chỉ cần truyền nó vào lời gọi hàm khi bắt đầu một gRPC service:

```
server := grpc.NewServer(grpc.UnaryInterceptor(filter))
```



Để hiện thực một interceptor như vậy, ta tạo ra hàm `filter` như sau:

```

// ctx và req là tham số của phương thức RPC bình thường.
// info chỉ ra phương thức gRPC tương ứng
// hiện đang được sử dụng và tham số handler
// tương ứng với hàm gRPC hiện tại
func filter(ctx context.Context,
    req interface{},
    info *grpc.UnaryServerInfo,
    handler grpc.UnaryHandler,
) (resp interface{}, err error) {
    // ghi log tham số info
    log.Println("filter:", info)

    // gọi tới phương thức gRPC gắn với `handler`
    return handler(ctx, req)
}

```

Nếu hàm interceptor trả về lỗi thì lệnh gọi phương thức gRPC sẽ được coi là failure. Chúng ta lợi dụng điều này để thực hiện một số xác thực trên các tham số đầu vào và cả kết quả trả về của Interceptor.

Sau đây là một interceptor có chức năng thêm một exception cho phương thức gRPC:

```

func filter(
    ctx context.Context,
    req interface{},
    info *grpc.UnaryServerInfo,
    handler grpc.UnaryHandler,
) (resp interface{}, err error) {
    log.Println("filter:", info)
}

```

```
// nếu có exception thì throw về
defer func() {
    // recover bắt giá trị của goroutine bị panic
    if r := recover(); r != nil {
        err = fmt.Errorf("panic: %v", r)
    }
}()

return handler(ctx, req)
}
```

Ta áp dụng cho hàm `SayHello` của service `Greeter`:

```
type myGrpcServer struct{}

func (s *myGrpcServer) SayHello(ctx context.Context, in *HelloRequest) (*HelloReply, error) {
    // giả sử ở hàm này ta gặp panic, nó sẽ throw ra panic này
    panic("debug")
    return &HelloReply{Message: "Hello " + in.Name}, nil
}
```

Chi tiết về code có thể xem tại [đây](#)

Chạy thử chương trình của chúng ta:

```
$ make gen
$ go build
$ ./3-interceptor

2019/09/06 16:17:38 [server] filter: &{0xceb050 /main.Greeter/SayHello}
2019/09/06 16:17:38 [server] validate req
2019/09/06 16:17:38 [client] could not greet: rpc error: code = Unknown desc = panic: debug
```

Kết quả cho thấy request của client đi qua hàm filter (là interceptor) gặp hàm `SayHello` throw ra panic, filter lấy panic này trả về cho client.

Tuy nhiên, chỉ một interceptor có thể được gắn cho một service trong gRPC framework, cho nên tất cả chức năng interceptor chỉ có thể thực hiện trong một hàm. Package `go-grpc-middleware` trong project opensource [grpc-ecosystem](#) có hiện thực cơ chế hỗ trợ cho một chuỗi interceptor dựa trên gRPC.

Một ví dụ về cách sử dụng một chuỗi interceptor trong package `go-grpc-middleware`:

```
import "github.com/grpc-ecosystem/go-grpc-middleware"

myServer := grpc.NewServer(
    grpc.UnaryInterceptor(grpc_middleware.ChainUnaryServer(
        filter1, filter2, ...
    )),
    grpc.StreamInterceptor(grpc_middleware.ChainStreamServer(
        filter1, filter2, ...
    )),
)
```

Xem chi tiết: [go-grpc-middleware](#)

Liên kết

- Phần tiếp theo: [gRPC và Protobuf extensions](#)

- **Phần trước:** [Làm quen với gRPC](#)
- [Mục lục](#)

3.5. gRPC và Protobuf extensions

Hiện nay, cộng đồng Open source đã phát triển rất nhiều extensions xung quanh Protobuf và gRPC, tạo thành một hệ sinh thái to lớn. Ở phần này sẽ trình bày về một số extensions thông dụng.

3.5.1 Validator

Trong Protobuf chúng ta có thể quy định giá trị mặc định của các trường thông qua phần mở rộng, ví dụ:

hello.proto:

```
syntax = "proto3";
package main;
// import phần mở rộng của protobuf
import "google/protobuf/descriptor.proto";
// định nghĩa một số trường trong phần mở rộng
extend google.protobuf.FieldOptions {
    // những con số như: 50000, 50001 là duy nhất cho mỗi trường
    string default_string = 50000;
    int32 default_int = 50001;
}
// định nghĩa nội dung message
message Message {
    // default_string là giá trị mặc định cho name
    string name = 1 [(default_string) = "gopher"];
    // tương tự, age sẽ có giá trị 10 nếu không khởi tạo
    int32 age = 2[(default_int) = 10];
}
```

Trong cộng đồng Open source, thư viện [go-protoValidators](#) là một extension của Protobuf có chức năng validator rất mạnh mẽ dựa trên phần mở rộng tự nhiên của Protobuf. Để sử dụng validator đầu tiên ta cần phải tải plugin sinh mã nguồn bên dưới:

```
$ go get github.com/mwitkow/go-proto-validators/protoc-gen-govalidators
```

Sau đó thêm phần `validation rules` vào các thành viên của Message dựa trên rules của go-protoValidators validator.

hello.proto: (dùng thư viện [validator](#))

```
syntax = "proto3";
package main;
// import file validator.proto
import "github.com/mwitkow/go-proto-validators/validator.proto";
// định nghĩa message
message Message {
    // dấu ngoặc vuông mang ý nghĩa là phần tùy chọn
    string important_string = 1 [
        // regex sẽ validate trường important_string đúng theo syntax hay không
        (validator.field) = {regex: "[a-z]{2,5}"}
    ];
    int32 age = 2 [
        // tương tự, giá trị của a sẽ được validate lớn hơn 0 và nhỏ hơn 100
        (validator.field) = {int_gt: 0, int_lt: 100}
    ];
}
```

Tất cả những validation rules được định nghĩa trong message `FieldValidator` trong file `validator.proto`. Trong đó ta sẽ thấy một số trường được dùng ở ví dụ trên như sau:

`mwitkow/go-protoValidators/validator.proto:`

```
syntax = "proto2";
package validator;

import "google/protobuf/descriptor.proto";

extend google.protobuf.FieldOptions {
    optional FieldValidator field = 65020;
}

message FieldValidator {
    // sử dụng Golang RE2-syntax regex để match với nội dung các field
    optional string regex = 1;
    // giá trị của biến integer bình thường lớn hơn giá trị này.
    optional int64 int_gt = 2;
    // giá trị của biến integer bình thường nhỏ hơn giá trị này.
    optional int64 int_lt = 3;

    // ...
}
```

Phần chú thích của mỗi trường ở trên sẽ cho chúng ta thông tin về chức năng của chúng. Sau khi chọn được các chức năng validate cần thiết, chúng ta dùng lệnh sau để sinh ra mã nguồn validator:

```
$ protoc \
  --proto_path=${GOPATH}/src \
  --proto_path=${GOPATH}/src/github.com/google/protobuf/src \
  --proto_path=. \
  --govalidators_out=. --go_out=plugins=grpc:.\ \
  hello.proto

// Trong đó:
// - proto_path: đường dẫn đến tất cả các file .proto được sử dụng
// - govalidators_out: plugin sinh ra mã nguồn validator
// Chú ý:
// - Trong Windows, ta thay thế ${GOPATH} thành %GOPATH%
```

Lệnh trên sẽ gọi chương trình `protoc-gen-govalidators` để sinh ra file với tên `hello.validator.pb.go`, nội dung của nó sẽ như sau:

`hello.validator.pb.go:`

```
// định nghĩa chuỗi regex
var _regex_Message_ImportantString = regexp.MustCompile(`^[a-z]{2,5}$`)
// hàm Validate() sẽ chạy các rules và bắt lỗi nếu có
func (this *Message) Validate() error {
    // rule 1 kiểm tra ImportantString có theo regex hay không, nếu có lỗi sẽ ném ra
    if !_regex_Message_ImportantString.MatchString(this.ImportantString) {
        return go_proto_validators.FieldError("ImportantString", fmt.Errorf(
            `value '%v' must be a string conforming to regex "^[a-z]{2,5}$"`,
            this.ImportantString,
        ))
    }
    // rule 2 kiểm tra Age > 0 hay không, nếu có lỗi sẽ ném ra
    if !(this.Age > 0) {
        return go_proto_validators.FieldError("Age", fmt.Errorf(
            `value '%v' must be greater than '0'`, this.Age,
        ))
    }
    // rule 3 kiểm tra Age < 100 hay không, nếu có lỗi sẽ ném ra
```

```

    if !(this.Age < 100) {
        return go_proto_validators.FieldError("Age", fmt.Errorf(
            `value '%v' must be less than '100'`, this.Age,
        ))
    }
    // trả về nil nếu kiểm tra tất cả các rules trên đều hợp lệ
    return nil
}

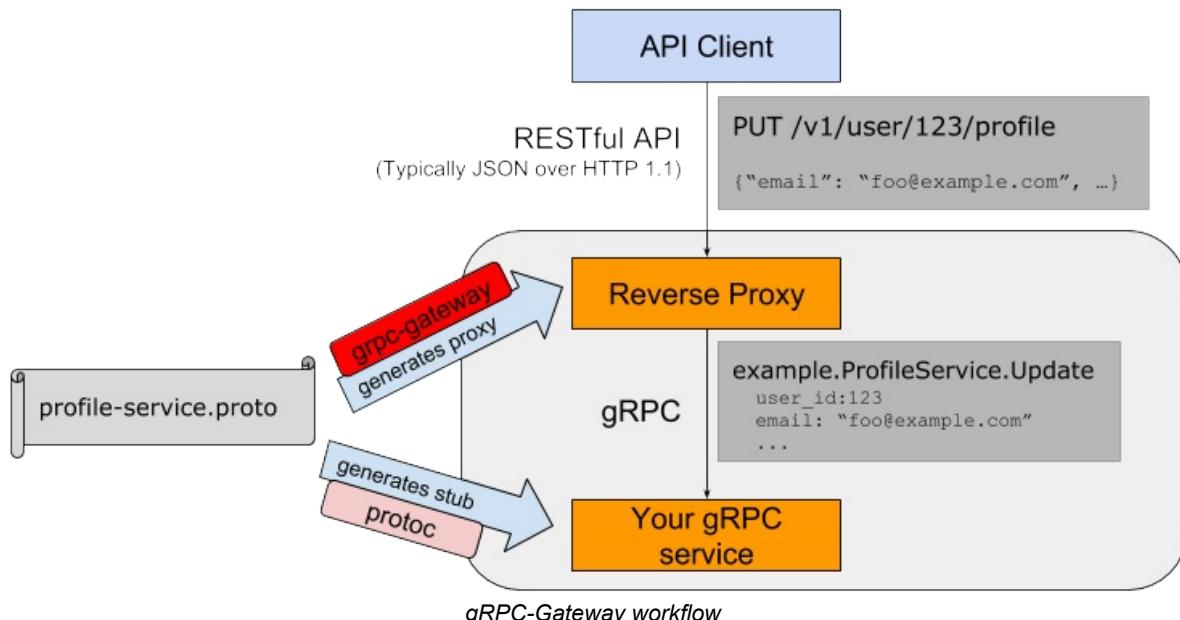
```

Thông qua hàm Validate() được sinh ra, chúng có thể được kết hợp với `gRPC interceptor`, chúng ta có thể dễ dàng validate giá trị của tham số đầu vào và kết quả trả về của mỗi hàm.

3.5.2 REST interface

Hiện nay RESTful JSON API vẫn là sự lựa chọn hàng đầu cho các ứng dụng web hay mobile. Vì tính tiện lợi và dễ dùng của RESTful API nên chúng ta vẫn sử dụng nó để frontend có thể giao tiếp với hệ thống backend. Nhưng khi chúng ta sử dụng framework gRPC của Google để xây dựng các service. Các service sử dụng gRPC thì dễ dàng trao đổi dữ liệu với nhau dựa trên giao thức HTTP/2 và protobuf, nhưng ở phía frontend lại sử dụng `RESTful API` API hoạt động trên giao thức HTTP/1. Vấn đề đặt ra là chúng ta cần phải chuyển đổi các yêu cầu RESTful API thành các yêu cầu gRPC để hệ thống các service gRPC có thể hiểu được.

Cộng đồng Open source đã xây dựng một project với tên gọi là `grpc-gateway`, nó sẽ sinh ra một proxy có vai trò chuyển các yêu cầu REST HTTP thành các yêu cầu gRPC HTTP2.



Trong file Protobuf (chỉ có ở proto3), chúng ta sẽ thêm thông tin phần routing ứng với các hàm trong gRPC service, để dựa vào đó grpc-gateway sẽ sinh ra mã nguồn proxy tương ứng.

`rest_service.proto:`

```

// phiên bản proto3
syntax = "proto3";
// tên package được sinh ra
package main;
// chú ý: import annotations.proto để dùng chức năng grpc-gateway
import "google/api/annotations.proto";
// định nghĩa message trao đổi
message StringMessage {

```

```

    string value = 1;
}
// định nghĩa RestService
service RestService {
    // định nghĩa hàm RPC Get trong service
    rpc Get(StringMessage) returns (StringMessage) {
        // nội dung phần option trong này định nghĩa Rest API ra bên ngoài
        option (google.api.http) = {
            // get: là tên phương thức được sử dụng
            get: "/get/{value}"
            // "/get/{value}" : là đường dẫn uri,
            // trong đó {value} được pass vào uri là nội dung StringMessage request
        };
    }
    // định nghĩa hàm RPC Post trong service
    rpc Post(StringMessage) returns (StringMessage) {
        option (google.api.http) = {
            // dùng phương thức post
            post: "/post"
            // StringMessage sẽ dưới dạng chuỗi Json khi gửi Request (vd: '{"value":"Hello, World"}')
            body: "*"
        };
    }
}

```

Chúng ta cài đặt plugin protoc-gen-grpc-gateway với những lệnh sau:

```
$ go get -u github.com/grpc-ecosystem/grpc-gateway/protoc-gen-grpc-gateway
```

Sau đó chúng ta sinh ra mã nguồn routing cho grpc-gateway thông qua plugin sau:

```

$ protoc -I/usr/local/include -I. \
-I${GOPATH}/src \
-I${GOPATH}/src/github.com/grpc-ecosystem/grpc-gateway/third_party/googleapis \
--grpc-gateway_out=. --go_out=plugins=grpc:.\
hello.proto

// Trong windows: Thay thế ${GOPATH} với %GOPATH%.

```

Plugin sẽ sinh ra hàm RegisterRestServiceHandlerFromEndpoint() cho RestService service như sau:

```

func RegisterRestServiceHandlerFromEndpoint(
    ctx context.Context, mux *runtime.ServeMux, endpoint string,
    opts []grpc.DialOption,
) (err error) {
    ...
}

```

Hàm RegisterRestServiceHandlerFromEndpoint được dùng để chuyển tiếp những request được định nghĩa trong REST interface đến gRPC service. Sau khi registering các Route handle, chúng ta sẽ chạy proxy web service trong hàm main như sau:

proxy/main.go:

```

func main() {
    // khai báo biến context để xử lý signal kết thúc goroutine
    ctx := context.Background()
    ctx, cancel := context.WithCancel(ctx)
    // hàm cancel() sẽ kích hoạt ctx.Done()
    defer cancel()
    // mux được dùng cho việc routing
    mux := runtime.NewServeMux()
}

```

```
// gọi hàm để đăng kí RestService cho proxy
err := RegisterRestServiceHandlerFromEndpoint(
    // truyền vào biến ctx, mux, và địa chỉ gRPC service
    ctx, mux, "localhost:5000",
    []grpc.DialOption{grpc.WithInsecure()},
)
// in ra lỗi nếu có
if err != nil {
    log.Fatal(err)
}
// bắt đầu lắng nghe http client trên port 8080
http.ListenAndServe(":8080", mux)
}

// $ go run proxy/main.go
```

Tiếp theo ta sẽ chạy gRPC service:

restservice/main.go:

```
// khai báo struct xây dựng RestService
type RestServiceImpl struct{}
// hàm Get RPC được xây dựng như sau
func (r *RestServiceImpl) Get(ctx context.Context, message *StringMessage) (*StringMessage, error) {
    return &StringMessage{Value: "Get hi:" + message.Value + "#"}, nil
}
// tương tự với hàm Post RPC được xây dựng với
func (r *RestServiceImpl) Post(ctx context.Context, message *StringMessage) (*StringMessage, error) {
    return &StringMessage{Value: "Post hi:" + message.Value + "@"}, nil
}
// hàm main của gRPC service
func main() {
    // khởi tạo một grpc Server mới
    grpcServer := grpc.NewServer()
    // register grpc Server với đối tượng xây dựng các hàm RPC
    RegisterRestServiceServer(grpcServer, new(RestServiceImpl))
    // listen gRPC Service trên port 5000, bỏ qua lỗi trả về nếu có
    lis, _ := net.Listen("tcp", ":5000")
    grpcServer.Serve(lis)
}

// $ go run restservice/main.go
```

Sau khi chạy hai chương trình gRPC và REST services, chúng ta có thể tạo request REST service với lệnh [curl](#):

```
// gọi service Get
$ curl localhost:8080/get/gopher
>{"value":"Get: gopher"}
// gọi service Post
$ curl localhost:8080/post -X POST --data '{"value":"grpc"}'
>{"value":"Post: grpc"}
```

Khi chúng ta publishing REST interface thông qua [Swagger](#), một swagger file có thể được sinh ra nhờ vào công cụ `grpc-gateway` bằng lệnh bên dưới:

```
// chạy lệnh sau để cài đặt nếu chưa có sẵn
$ go get -u github.com/grpc-ecosystem/grpc-gateway/protoc-gen-swagger
// lệnh sinh ra swagger file
$ protoc -I. \
    -I$GOPATH/src/github.com/grpc-ecosystem/grpc-gateway/third_party/googleapis \
    --swagger_out=. \
    hello.proto

// Trong đó,
```

```
// --swagger_out=: dùng plugin swagger để sinh ra swagger file tại thư mục hiện tại
```

File `hello.swagger.json` sẽ được sinh ra sau đó. Trong trường hợp này, chúng ta có thể dùng `swagger-ui` project để cung cấp tài liệu REST interface và testing dưới dạng web pages.

3.5.3 Dùng Docker grpc-gateway

Với những lập trình viên phát triển gRPC Services trên các ngôn ngữ không phải Golang như Java, C++, ... có nhu cầu sinh ra grpc gateway cho các services của họ nhưng gặp khá nhiều khó khăn từ việc cài đặt môi trường Golang, Protobuf, các lệnh generate,v.v.. Có một giải pháp đơn giản hơn đó là sử dụng Docker để xây dựng grpc-gateway theo bài hướng dẫn chi tiết sau [buildingdocker-grpc-gateway](#).

3.5.4 Nginx

Những phiên bản [Nginx](#) về sau cũng đã hỗ trợ gRPC với khả năng register nhiều gRPC service instance giúp load balancing (cân bằng tải) dễ dàng hơn. Những extension của Nginx về gRPC là một chủ đề lớn, ở đây chúng tôi không trình bày hết được, các bạn có thể tham khảo các tài liệu trên trang chủ của Nginx như [ở đây](#).

Liên kết

- Phần tiếp theo: [Công cụ grpcurl](#)
- Phần trước: [Một số vấn đề khác của gRPC](#)
- [Mục lục](#)

3.6. Công cụ grpcurl

Bản thân Protobuf đã có chức năng phản chiếu (reflection) lại file Proto của đối tượng khi thực thi. gRPC cũng cung cấp một package reflection để thực hiện các truy vấn cho gRPC service. Mặc dù gRPC có một hiện thực bằng C++ của công cụ `grpc_cli`, có thể được sử dụng để truy vấn danh sách gRPC hoặc gọi phương thức gRPC, nhưng bởi vì phiên bản đó cài đặt khá phức tạp nên ở đây chúng ta sẽ dùng công cụ `grpcurl` được hiện thực thuận bằng Golang. Phần này ta sẽ cùng tìm hiểu cách sử dụng công cụ này.

3.6.1 Khởi động một reflection service

Chỉ có duy nhất hàm `Register` trong package reflection, hàm này dùng để đăng ký `grpc.Server` với reflection service. Trong document của package có hướng dẫn như sau:

```
import (
    "google.golang.org/grpc/reflection"
)

func main() {
    s := grpc.NewServer()
    pb.RegisterYourOwnServer(s, &server{})

    // đăng ký reflection service trên gRPC server.
    reflection.Register(s)

    s.Serve(lis)
}
```

Nếu gRPC reflection service được khởi chạy thì các gRPC service có thể được truy vấn hoặc gọi ra bằng reflection service do package reflection cung cấp.

3.6.2 Xem danh sách service

Grpcurl là công cụ được cộng đồng Open source của Golang phát triển, quá trình cài đặt như sau:

```
$ go get github.com/fullstorydev/grpcurl
$ go install github.com/fullstorydev/grpcurl/cmd/grpcurl
```

Sử dụng phím tắt trong grpcurl là lệnh `list`, được sử dụng để lấy danh sách các service hoặc các phương thức trong service. Ví dụ, `grpcurl localhost:1234 list` là lệnh sẽ nhận được một danh sách các service gRPC trên port 1234 ở localhost.

Khi sử dụng grpcurl với giao thức TLS ta cần chỉ định các đường dẫn tới public key `-cert` và private key `-key`. Đối với gRPC service không có giao thức TLS, quy trình xác minh chứng chỉ TLS có thể bỏ qua bằng tham số `-plaintext`. Nếu đó là giao thức Unix Socket, cần chỉ định tham số `-unix`.

Nếu các file public và private key chưa được cấu hình và quá trình xác minh chứng chỉ bị bỏ qua, ta có thể sẽ gặp lỗi như sau:

```
$ grpcurl localhost:1234 list
Failed to dial target host "localhost:1234": tls: first record does not \
look like a TLS handshake
```

Nếu gRPC service bình thường nhưng được khởi động reflection service thì sẽ có thông báo lỗi:

```
$ grpcurl -plaintext localhost:1234 list
Failed to list services: server does not support the reflection API
```

Giả định rằng gRPC service đã được kích hoạt reflection service, file Protobuf của service như sau:

```
syntax = "proto3";

package HelloService;

message String {
    string value = 1;
}

service HelloService {
    rpc Hello (String) returns (String);
    rpc Channel (stream String) returns (stream String);
}
```

Kết quả với lệnh `list`:

```
$ grpcurl -plaintext localhost:1234 list
HelloService.HelloService
grpc.reflection.v1alpha.ServerReflection
```

Trong đó `HelloService.HelloService` là service được định nghĩa trong file protobuf. `ServerReflection` là reflection service được package reflection đăng ký. Thông qua service này chúng ta có thể truy vấn thông tin của tất cả các gRPC service bao gồm chính nó.

3.6.3 Danh sách các phương thức của service

Nếu tiếp tục sử dụng lệnh `list` ta có thể xem được cả danh sách các phương thức trong `HelloService`:

```
$ grpcurl -plaintext localhost:1234 list HelloService.HelloService
Channel
Hello
```

Từ kết quả cho thấy service này cung cấp 2 phương thức là `Channel` và `Hello`, tương ứng với các định nghĩa trong file Protobuf.

Nếu muốn biết chi tiết của từng phương thức, ta có thể sử dụng câu lệnh `describe`:

```
$ grpcurl -plaintext localhost:1234 describe HelloService.HelloService
HelloService.HelloService is a service:
service HelloService {
    rpc Channel ( stream .HelloService.String ) returns ( stream .HelloService.String );
    rpc Hello ( .HelloService.String ) returns ( .HelloService.String );
```

Kết quả là danh sách các phương thức có trong service cùng với mô tả các tham số input cũng như giá trị trả về tương ứng của chúng.

3.6.4 Lấy thông tin kiểu dữ liệu

Sau khi có được danh sách các phương thức và kiểu của giá trị trả về, chúng ta có thể tiếp tục xem thông tin chi tiết hơn về kiểu của các biến này. Sau đây là các sử dụng lệnh `describe` để xem thông tin của tham số

`HelloService.String :`

```
$ grpcurl -plaintext localhost:1234 describe HelloService.String
HelloService.String is a message:
message String {
    string value = 1;
}
```

Kết quả trả về đúng với mô tả trong file protobuf của service.

3.6.5 Lệnh gọi phương thức

Ta có thể gọi phương thức gRPC bằng cách truyền thêm tham số `-d` và một chuỗi json như input của hàm và gọi tới phương thức `Hello` trong `HelloService`, chi tiết như sau:

```
$ grpcurl -plaintext -d '{"value": "gopher"}' localhost:1234 HelloService>Hello
{
  "value": "hello:gopher"
}
```

Nếu có tham số `-d`, @ nghĩa là đọc vào tham số dạng json từ input chuẩn (stdin), cách này thường dùng để test các phương thức stream.

Ví dụ sau đây kết nối tới phương thức stream tên `Channel` và đọc tham số input stream từ input chuẩn:

```
$ grpcurl -plaintext -d @ localhost:1234 HelloService>HelloService/Channel
{"value":"gopher-vn"}
{
  "value": "hello:gopher-vn"
}

{"value": "vietnamese-vng"}
{
  "value": "hello:vietnamese-vng"
}
```

Liên kết

- Phần tiếp theo: [Lời nói thêm](#)
- Phần trước: [gRPC và Protobuf extensions](#)
- [Mục lục](#)

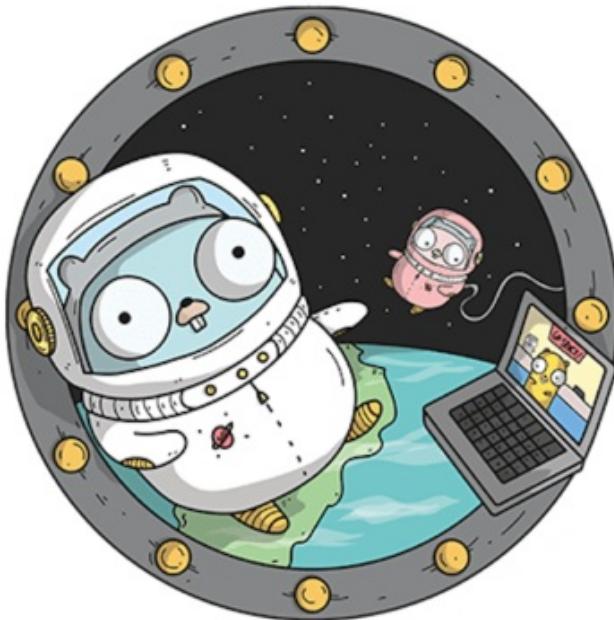
3.7. Lời nói thêm

Chúng ta nên đọc thêm các tài liệu và ví dụ được trang chủ [Protobuf](#) và [gRPC](#) cung cấp. Trong chương này tập trung vào RPC của thư viện chuẩn Go và framework gRPC dựa trên Protobuf. Song song đó là tìm hiểu cách tự mình tùy chỉnh framework RPC. Hiện tại, cộng đồng Open source có rất nhiều framework RPC riêng biệt, cũng như các hệ thống RPC để tùy biến trên các hệ thống phân tán. Người dùng nên chọn đúng công cụ theo nhu cầu thực tế của mình.

Liên kết

- Phản tiếp theo: [Chương 4](#)
- Phản trước: [Công cụ grpcurl](#)
- [Mục lục](#)

Chương 4: Go và Web



"I have reimplemented a networking project from Scala to Go. Scala code is 6000 lines. Go is about 3000. Even though Go does not have the power of abbreviation, the flexible type system seems to out-run Scala when the programs start getting longer. Hence, Go produces much shorter code asymptotically." – Petar Maymounko

Ngành công nghiệp phần mềm hiện đại không thể tách khỏi Web. Cho dù mục tiêu là trở thành một kỹ sư phần mềm, hay một nhà tư vấn về kỹ thuật thì kiến thức về Web nên là một trong những kiến thức nền tảng mà ta cần phải biết đến.

Liên kết

- Phản tiếp theo: [Giới thiệu về chương trình Web](#)
- Phản trước: [Chương 3: Lời nói thêm](#)
- [Mục lục](#)

4.1. Giới thiệu chương trình Web

Trong lập trình web, đối với các trang web nhỏ và đơn giản (như blog, portfolio, ...) ta sẽ sử dụng [template](#), đối với các trang web phức tạp và phát triển về lâu dài ta sẽ cần chia thành front end, back end riêng biệt để tiện cho các team làm việc độc lập và dễ dàng phát triển mở rộng.

Ngày nay có rất nhiều công nghệ hỗ trợ cho phát triển front end, mà nổi bật là Vue, React. Front end server được build để xử lý việc hiển thị giao diện còn các xử lý logic liên quan tới database nó sẽ phải gọi xuống api ở back end.

Back end cũng có thể chia thành nhiều service nhỏ (để scale được) và cung cấp các api viết bằng gRPC hoặc RESTful, tất cả có thể viết bằng nhiều ngôn ngữ như Java, C#, JS, PHP, Python, Rust,... nhưng ở đây chúng ta sẽ tìm hiểu về Go.

Phần này sẽ đề cập về cách xây dựng một chương trình web đơn giản bằng thư viện chuẩn của Go, sau đó giới thiệu các framework web trong cộng đồng Open source.

4.1.1 Dùng thư viện chuẩn net/http

Gói thư viện [net/http](#) đã cung cấp những hàm cơ bản cho việc routing URL, chúng ta sẽ dùng nó để viết một chương trình `http echo server`:

echo.go:

```
package main
// các gói thư viện cần import
import (
    "io/ioutil"
    "log"
    "net/http"
)
// hàm routing echo, gồm hai params
// r *http.Request : dùng để đọc yêu cầu từ client
// wr http.ResponseWriter : dùng để ghi phản hồi về client
func echo(wr http.ResponseWriter, r *http.Request) {
    // đọc thông điệp mà client gửi tới trong r.Body
    msg, err := ioutil.ReadAll(r.Body)
    // phản hồi về client lỗi nếu có
    if err != nil {
        wr.Write([]byte("echo error"))
        return
    }
    // phản hồi về client chính thông điệp mà client gửi
    writeLen, err := wr.Write(msg)
    // nếu lỗi xảy ra, hoặc kích thước thông điệp phản hồi khác
    // kích thước thông điệp nhận được
    if err != nil || writeLen != len(msg) {
        log.Println(err, "write len:", writeLen)
    }
}
// hàm main của chương trình
func main() {
    // mapping url ứng với hàm routing echo
    http.HandleFunc("/", echo)
    // địa chỉ http://127.0.0.1:8080/
    err := http.ListenAndServe(":8080", nil)
    // log ra lỗi nếu bị trùng port
    if err != nil {
        log.Fatal(err)
    }
}
```

Kết quả khi chạy chương trình:

```
$ go run echo.go &
$ curl http://127.0.0.1:8080/ -d '"Hello, World"'
"Hello, World"
```

4.1.2 Dùng thư viện bên ngoài

Bởi vì gói thư viện chuẩn `net/http` của Golang chỉ hỗ trợ những hàm routing và hàm chức năng cơ bản. Cho nên trong cộng đồng Golang có ý tưởng là viết thêm các thư viện hỗ trợ routing khác ngoài `net/http`.

Thông thường, nếu các dự án routing HTTP của bạn có những đặc điểm sau: `URI` cố định, và tham số không truyền thông qua `URI`, thì nên dùng thư viện chuẩn là đủ. Nhưng với những trường hợp phức tạp hơn, thư viện chuẩn `net/http` vẫn còn thiếu các chức năng hỗ trợ. Ví dụ, xét các route sau:

```
GET    /card/:id
POST   /card/:id
DELETE /card/:id
GET    /card/:id/name
GET    /card/:id/relations
```

Có thể thấy rằng, cùng là đường dẫn có chứa `/card/:id`, nhưng có phương thức khác nhau hoặc nhánh con khác nhau sẽ dẫn đến logic xử lý khác nhau, cách xử lý những đường dẫn trùng tên như vậy thường sẽ phức tạp. Khi đó chúng ta có thể nghĩ đến việc sử dụng một số framework routing bên ngoài từ cộng đồng Open source như [HttpRouter](#), [Gin](#), [Gorilla](#), [Revel](#), [Beego](#), [Iris](#),...

Liên kết

- Phần tiếp theo: [Routing trong Web](#)
- Phần trước: [Chương 4: Go và Web](#)
- [Mục lục](#)

4.2. Routing trong Web

Trong phần trước, chúng ta đã tìm hiểu cách dùng thư viện chuẩn [http/net](#) để hiện thực hàm routing đơn giản. Tuy nhiên một framework web sẽ có nhiều thành phần khác ngoài việc định tuyến như xử lý tham số URI, phương thức, mã lỗi.

4.2.1 RESTful API

[RESTful](#) là một tiêu chuẩn thiết kế API trong ngành công nghiệp web hiện đại. Ngoài những phương thức GET, POST thì RESTful cũng định nghĩa vài phương thức khác trong giao thức HTTP bao gồm:

Phương thức HTTP:

```
const (
    MethodGet     = "GET"
    MethodHead    = "HEAD"
    MethodPost    = "POST"
    MethodPut     = "PUT"
    MethodPatch   = "PATCH" // RFC 5789
    MethodDelete   = "DELETE"
    MethodConnect  = "CONNECT"
    MethodOptions  = "OPTIONS"
    MethodTrace    = "TRACE"
)
```

Nhìn vào những đường dẫn RESTful API sau:

```
// mỗi API sẽ có một phương thức tương ứng
// tham số được truyền vào thông qua URI

GET /repos/:owner/:repo/comments/:id/reactions
POST /projects/:project_id/columns
PUT /user/starred/:owner/:repo
DELETE /user/starred/:owner/:repo
```

Nếu hệ thống web của chúng ta cần có những API tương tự trên, việc sử dụng thư viện chuẩn net/http hiển nhiên là không đủ. Những API chứa parameters như trên của Github có thể được hỗ trợ hiện thực bởi thư viện [HttpRouter](#).

4.2.1 Tìm hiểu thư viện HttpRouter

Nhiều Open source web framework phổ biến của Go thường được xây dựng dựa trên [HttpRouter](#) như là [Gin](#) framework, hoặc hỗ trợ cho routing dựa trên những biến thể của HttpRouter. Khi sử dụng các framework đó, chúng ta cần phải tránh một số trường hợp mà nó dẫn đến xung đột routing khi thiết kế.

Ví dụ:

```
// xung đột trong trường hợp đặc biệt id là 'info'
// vì cùng phương thức nên cùng nằm trên một 'cây định tuyến'
// 'cây định tuyến' được nói ở phần sau
GET /user/info/:name
GET /user/:id

// không xung đột vì khác phương thức
// nên sẽ tạo ra hai 'cây định tuyến' cho hai phương thức khác nhau
GET /user/info/:name
```

```
POST /user/:id
// các lỗi trên sẽ bị bắt lỗi panic trong HttpRouter
```

HttpRouter hỗ trợ kí tự đặc biệt * trong đường dẫn.

Ví dụ:

```
Pattern: /src/*filepath
/src/           filepath = ""
/src/somefile.go    filepath = "somefile.go"
/src/subdir/somefile.go  filepath = "subdir/somefile.go"

// thiết kế này thường dùng để xây dựng một static file server
```

HttpRouter cũng hiện thực tùy chỉnh hàm callback trong một vài trường hợp đặc biệt như là lỗi 404:

Ví dụ:

```
r := httprouter.New()
r.NotFound = http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("oh no, not found"))
})
```

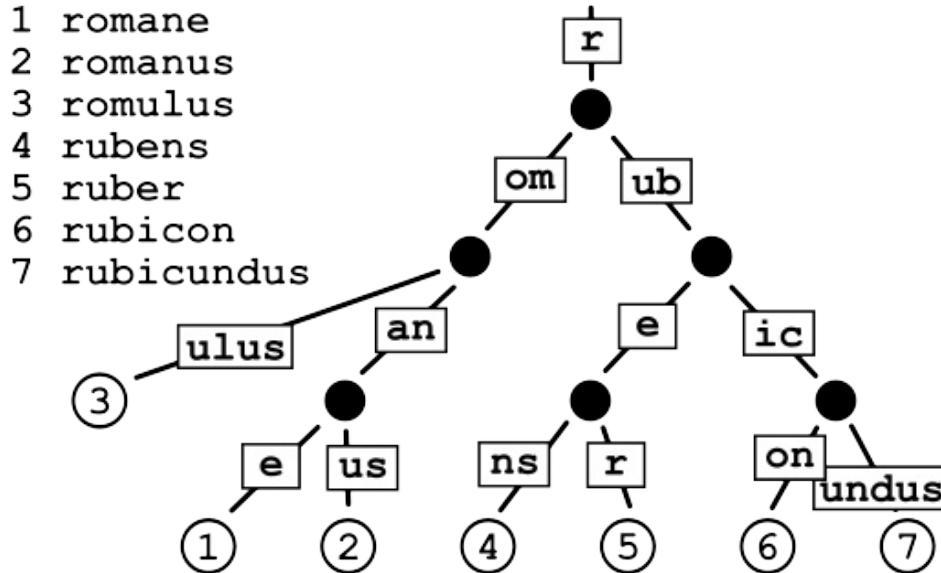
Hoặc tùy chỉnh hàm callback khi panic bên trong:

Ví dụ:

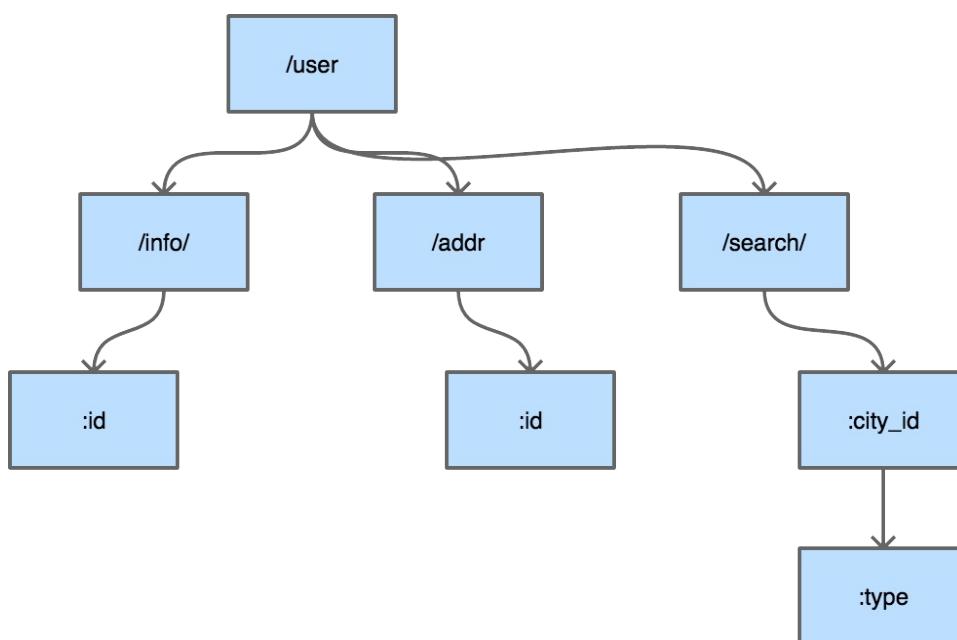
```
r.PanicHandler = func(w http.ResponseWriter, r *http.Request, c interface{}) {
    log.Printf("Recovering from panic, Reason: %#v", c.(error))
    w.WriteHeader(http.StatusInternalServerError)
    w.Write([]byte(c.(error).Error()))
}
```

4.2.2 Cấu trúc dữ liệu trong HttpRouter

Cấu trúc dữ liệu được dùng bởi HttpRouter và nhiều framework routing dẫn xuất khác là [Radix Tree](#). Cây Radix thường được dùng để truy xuất chuỗi, để xem chúng có nằm trong cây hay không và lấy thông tin gắn với chuỗi đó, phương pháp tìm kiếm theo chiều sâu sẽ bắt đầu từ node gốc, và thời gian xấp xỉ là $O(n)$, và n là chiều sâu của cây.



Kiểu chuỗi không phải là một kiểu số học nên không thể so sánh trực tiếp như kiểu số, và thời gian xấp xỉ của việc so sánh hai chuỗi là phụ thuộc vào độ dài của chuỗi, và sau đó dùng giải thuật như là [binary search](#) để tìm kiếm, độ phức tạp về thời gian có thể cao. Dùng Radix tree để lưu trữ và truy xuất chuỗi là một cách đảm bảo tối ưu về thời gian, mỗi phần trong đường dẫn được xem là một chuỗi và được lưu trữ trong cây Radix như ví dụ sau:



4.2.3 Xây dựng Radix tree

Hãy xét quy trình của một Radix tree trong HttpRouter. Phần thiết lập routing có thể như sau:

```

PUT /user/installations/:installation_id/repositories/:repository_id
GET /marketplace_listing/plans/
GET /marketplace_listing/plans/:id/accounts
GET /search
GET /status
  
```

```
GET /support
GET /marketplace_listing/plans/ohyes
```

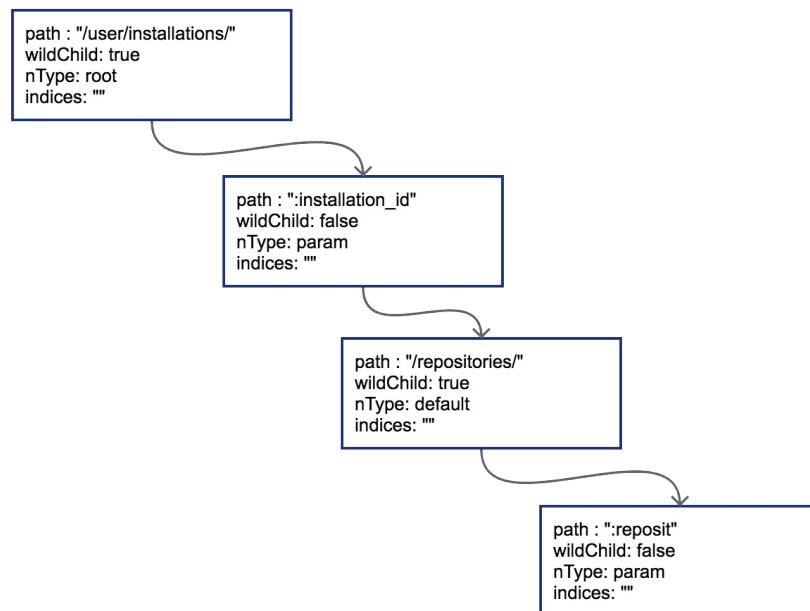
4.2.3.1 Khởi tạo

Radix tree có thể được lưu trữ trong cấu trúc của Router trong HttpRouter sử dụng một số cấu trúc dữ liệu sau:

```
// Router struct
type Router struct {
    // ...
    trees map[string]*node
    // Trong đó,
    // key: GET, HEAD, OPTIONS, POST, PUT, PATCH hoặc DELETE
    // value: node cha của cây Radix
    // ...
}
```

Mỗi phương thức sẽ tương ứng với một Radix tree độc lập và không chia sẻ dữ liệu với các cây khác. Đặc biệt đối với route chúng ta dùng ở trên, `PUT` và `GET` là hai Radix tree thay vì một. Đầu tiên, chèn route `PUT` vào Radix tree:

```
r := httprouter.New()
r.PUT("/user/installations/:installation_id/repositories/:repository", Hello)
```



Một cây từ điển nén được insert vào route

Kiểu của mỗi node trong Radix tree là `*httprouter.node`, trong đó, một số trường mang ý nghĩa sau:

```
path: // đường dẫn ứng với node hiện tại
wildChild: // cho dù là nút con tham số, nghĩa là nút có ký tự đại diện hoặc :id
nType: // loại nút có bốn giá trị liệt kê static/root/param/catchAll
static // chuỗi bình thường cho các node không gốc
root // nút gốc
param // nút tham số ví dụ :id
```

```
    catch // các nút ký tự đại diện, chẳng hạn như * anyway
  indices:
```

Tiếp theo, chúng ta chèn các route GET còn lại trong ví dụ để giải thích về quy trình chèn vào một node con.

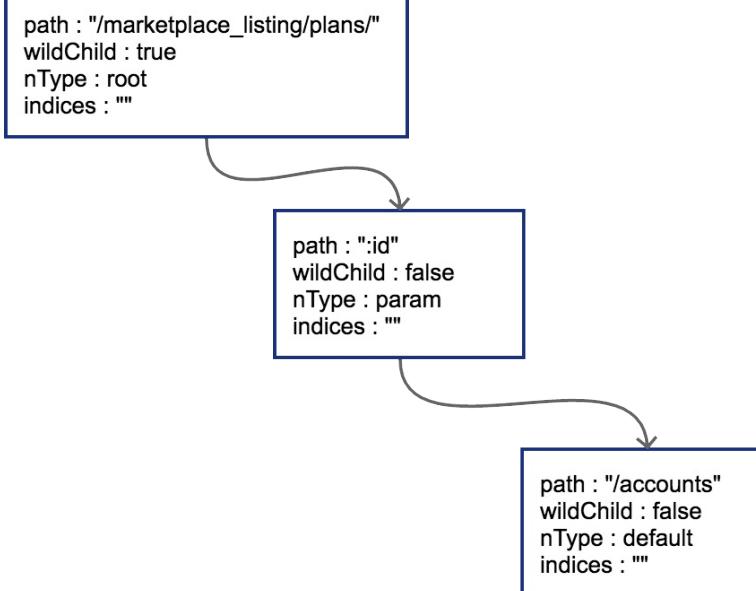
4.2.3.2 Chèn các route khác

Khi chúng ta chèn `GET /marketplace_listing/plans`, quá trình này sẽ tương tự như trước nhưng ở một Radix tree khác:

```
path : "/marketplace_listing/plans/"
wildChild : false
nType : root
indices : ""
```

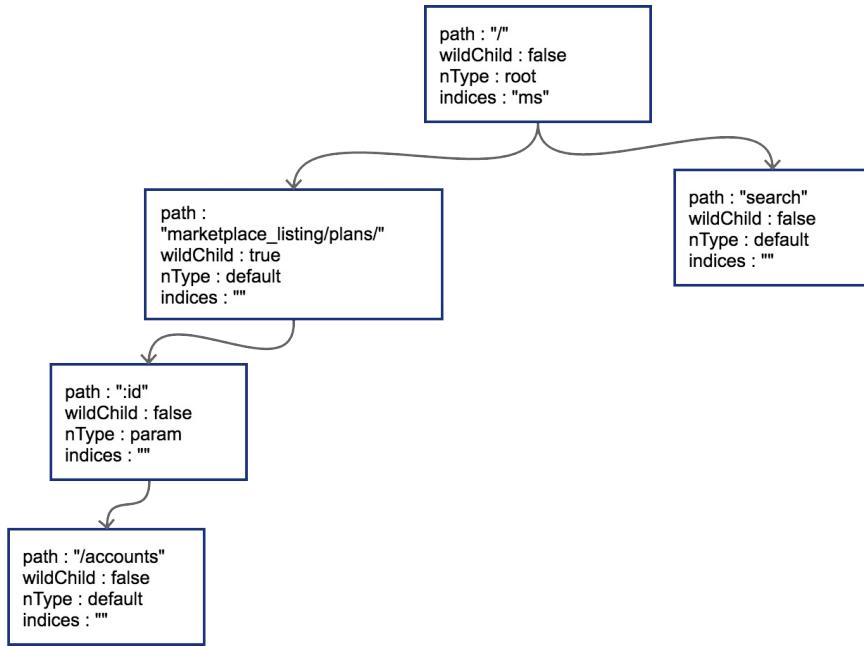
Chèn node đầu tiên vào Radix tree

Sau đó chèn đường dẫn `GET /marketplace_listing/plans/:id/accounts` cấu trúc Radix tree được hoàn thành sẽ như sau:



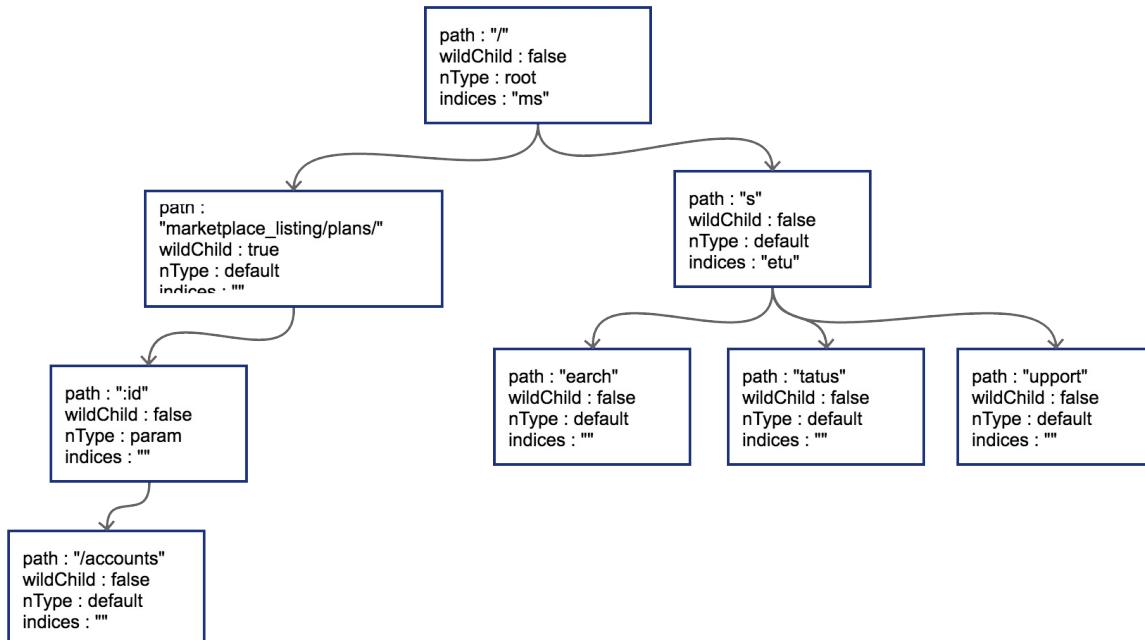
Chèn node thứ hai vào Radix tree

4.2.3.3 Phân nhánh Tiếp theo chúng ta chèn `GET /search`, sau đó sẽ sinh ra split tree như hình 5.6:



Chèn vào node thứ ba sẽ gây ra việc phân nhánh

Node gốc bây giờ sẽ bắt đầu từ ký tự /, chuỗi truy vấn phải bắt đầu từ node gốc chính, sau đó một route là `search` được phân nhánh từ gốc. Tiếp theo chèn `GET /status` và `GET /support` vào Redux tree. Lúc này, sẽ dẫn đến node `search` bị tách một lần nữa, và kết quả cuối cùng được nhìn thấy ở hình dưới:



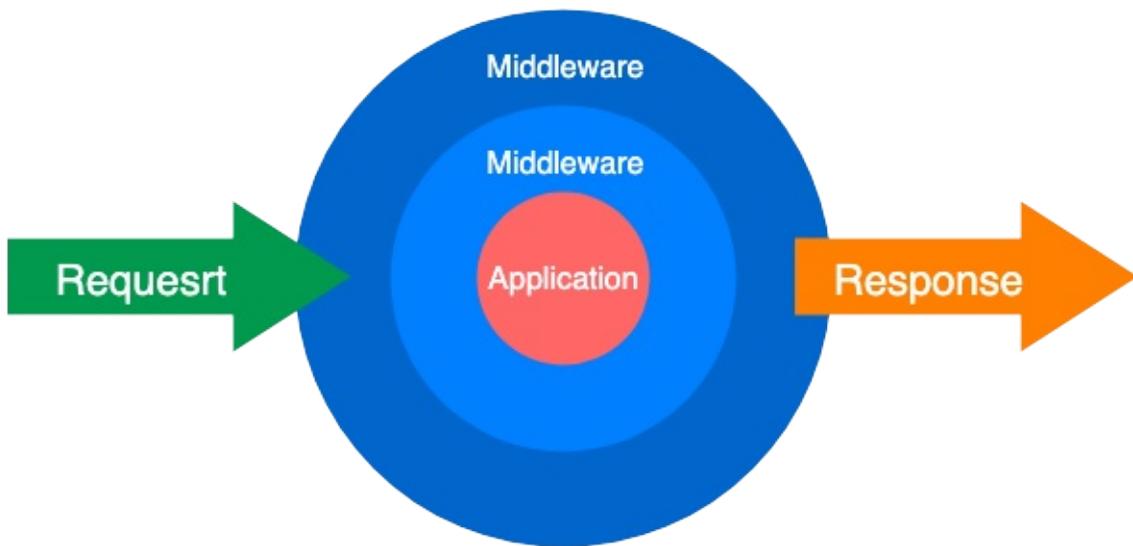
Sau khi chèn tất cả các node

Liên kết

- Phần tiếp theo: [Middleware](#)
- Phần trước: [Giới thiệu về chương trình Web](#)
- [Mục lục](#)

4.3 Middleware

Phần này sẽ phân tích tình huống dẫn tới việc sử dụng middleware, sau đó trình bày cách hiện thực một middleware đơn giản để tách biệt mã nguồn business và non-business. Trước hết chúng ta cùng nhắc lại khái niệm middleware là gì? Có thể nói ngắn gọn middleware là những đoạn mã trung gian nằm ở giữa request và response của ứng dụng web của chúng ta.



Middleware thường được dùng trong một số trường hợp chúng ta muốn ghi log hoạt động của hệ thống, báo cáo thời gian thực thi, xác thực,..

4.3.1 Tình huống đặt ra

Hãy nhìn vào đoạn mã nguồn sau:

main.go:

```
package main

func hello(wr http.ResponseWriter, r *http.Request) {
    wr.Write([]byte("hello"))
}

func main() {
    http.HandleFunc("/", hello)
    err := http.ListenAndServe(":8080", nil)
    //...
}
```

Bây giờ có một số nhu cầu mới, chúng tôi muốn tính được thời gian xử lý của Hello service được viết ở trên, nhu cầu này rất đơn giản, chúng tôi sẽ làm một số thay đổi nhỏ trên chương trình ở trên.

```
var logger = log.New(os.Stdout, "", 0)

func hello(wr http.ResponseWriter, r *http.Request) {
    timeStart := time.Now()
```

```

    wr.Write([]byte("hello"))
    timeElapsed := time.Since(timeStart)
    logger.Println(timeElapsed)
}

```

Đoạn mã nguồn thêm vào ở trên đã giải quyết được yêu cầu đặt ra, tuy nhiên trong quá trình phát triển, số lượng API ngày một tăng lên như sau:

main.go:

```

package main

func helloHandler(wr http.ResponseWriter, r *http.Request) {
    // ...
}

func showInfoHandler(wr http.ResponseWriter, r *http.Request) {
    // ...
}

func showEmailHandler(wr http.ResponseWriter, r *http.Request) {
    // ...
}

func showFriendsHandler(wr http.ResponseWriter, r *http.Request) {
    timeStart := time.Now()
    wr.Write([]byte("your friends is tom and alex"))
    timeElapsed := time.Since(timeStart)
    logger.Println(timeElapsed)
}

func main() {
    http.HandleFunc("/", helloHandler)
    http.HandleFunc("/info/show", showInfoHandler)
    http.HandleFunc("/email/show", showEmailHandler)
    http.HandleFunc("/friends/show", showFriendsHandler)
    // ...
}

```

Mỗi handler có một đoạn mã nguồn để ghi lại thời gian được đề cập từ trước. Mỗi lần chúng tôi thêm vào một route mới, cần phải sao chép những mã nguồn tương tự tới nơi chúng ta ta cần, bởi vì số lượng route ít, nên không phải là vấn đề lớn khi hiện thực.

Bây giờ, chúng ta cần bản báo cáo về dữ liệu thời gian chạy service cho hệ thống metrics. Nên cần phải thay đổi mã nguồn và gửi thời gian đến hệ thống metrics thông qua HTTP POST. Hãy thay đổi nó `helloHandler()`:

main.go:

```

func helloHandler(wr http.ResponseWriter, r *http.Request) {
    timeStart := time.Now()
    wr.Write([]byte("hello"))
    timeElapsed := time.Since(timeStart)
    logger.Println(timeElapsed)
    // Thêm phần upload thời gian
    metrics.Upload("timeHandler", timeElapsed)
}

```

Mỗi khi thay đổi, chúng ta có thể dễ dàng thấy rằng công việc phát triển sẽ rơi vào bế tắc. Bất kể nhu cầu phi chức năng hoặc thông kê trên hệ thống Web trong tương lai, các sửa đổi sẽ ảnh hưởng tới toàn bộ. Cũng như khi ta thêm một nhu cầu thống kê đơn giản, chúng ta cần phải thêm hàng tá những mã nguồn độc lập với business. Mặc dù dường như chúng không có lỗi trong thời gian đầu, nhưng sẽ thấy rõ hơn khi business càng phát triển.

4.3.2 Hiện thực middleware

Thực tế, vấn đề là chúng ta gây ra là đặt mã nguồn business và non-business cùng nhau. Trong hầu hết trường hợp, những yêu cầu non-business thường là làm một thứ gì đó trước khi xử lý HTTP request, và làm một thứ gì đó ngay sau khi chúng hoàn thành. Ý tưởng ở đây là tái cấu trúc lại mã nguồn để tách riêng mã nguồn của non-business riêng, như sau:

main.go (version2):

```
// hàm business logic của chúng ta
func hello(wr http.ResponseWriter, r *http.Request) {
    wr.Write([]byte("hello"))
}

// đây là một hàm thực hiện việc đóng gói hàm truyền vào
// để ghi nhận thời gian thực thi service
// hàm này được xem như là một Middleware
func timeMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(wr http.ResponseWriter, r *http.Request) {
        // ghi nhận thời gian trước khi chạy
        timeStart := time.Now()

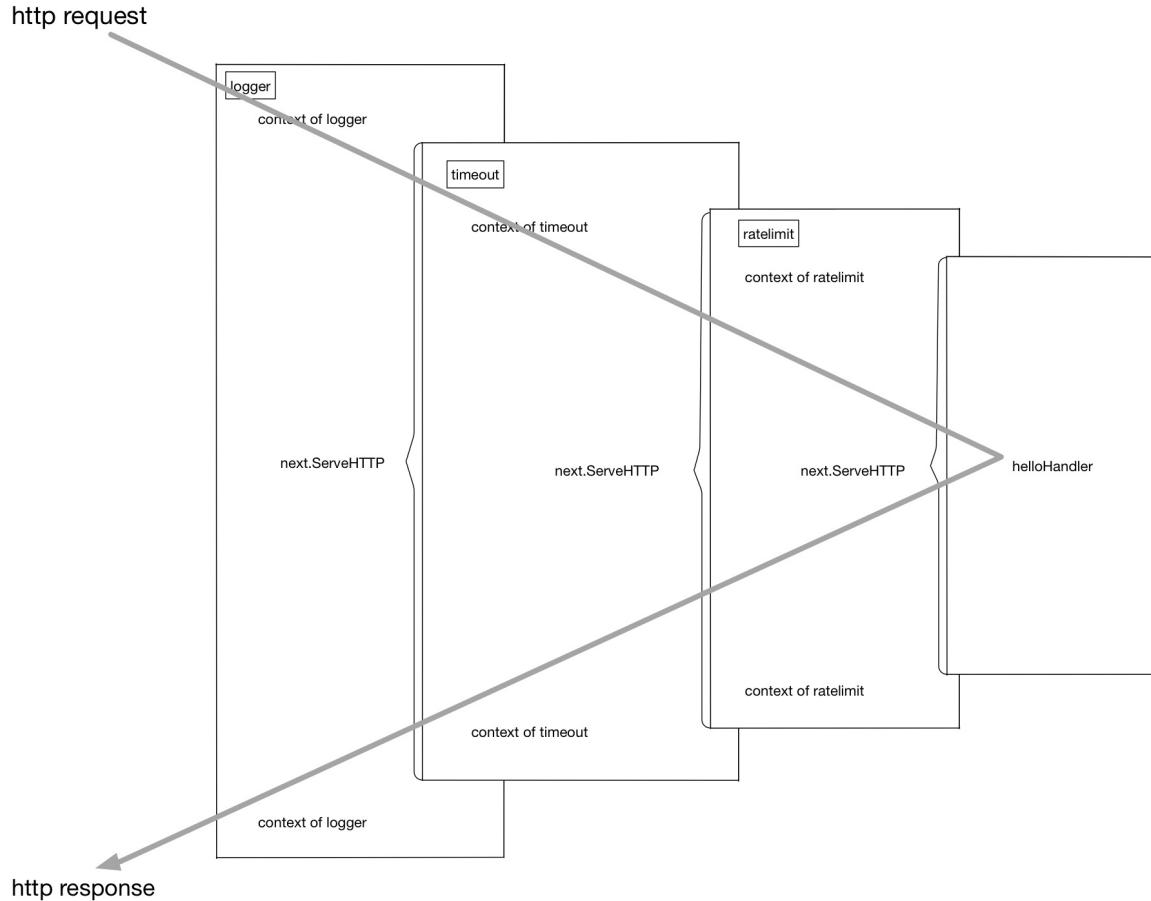
        // next là hàm business logic được truyền vào
        next.ServeHTTP(wr, r)
        // tính toán thời gian thực thi
        timeElapsed := time.Since(timeStart)
        // log ra thời gian thực thi
        logger.Println(timeElapsed)
    })
}

// Trong đó, http.Handler:
// type Handler interface {
//     ServeHTTP(ResponseWriter, *Request)
// }
func main() {
    http.Handle("/", timeMiddleware(http.HandlerFunc(hello)))
    err := http.ListenAndServe(":8080", nil)
    // ...
}
```

Bắt cứ hàm nào định nghĩa `ServeHTTP`, cũng đều là một đối tượng `http.Handler`. Những gì mà middleware làm là nhận vào hàm handler và trả về hàm handler khác kèm theo non-business logic. Chúng ta có thể dùng các middleware lồng vào nhau như bên dưới:

```
customizedHandler = logger(timeout(ratelimit(helloHandler)))
```

Ngũ cảnh của chuỗi các hàm trong quá trình thực thi có thể được thể hiện dưới đây:



Tuy nhiên cách dùng middleware lồng vào nhau như trên còn khá phức tạp.

4.3.3 Cách viết middleware thanh lịch hơn

Trong phần trước, sự tách biệt về mã nguồn hàm business và non-business function được giải quyết. Nhưng nếu bạn cần phải thay đổi thứ tự của những hàm đó, hoặc thêm, hoặc xóa middleware vẫn còn một số khó khăn, phần này chúng ta sẽ thực hiện việc tối ưu như sau.

Ví dụ:

```
r = NewRouter()
r.Use(logger)
r.Use(timeout)
r.Use(ratelimit)
r.Add("/", helloHandler)
```

Qua nhiều bước thiết lập, chúng ta có một chuỗi thực thi các hàm tương tự như ví dụ trước. Cách làm này giúp chúng ta dễ hiểu hơn. Nếu bạn muốn thêm hoặc xóa middleware, đơn giản thêm và xóa dòng ứng với lời gọi `use()`.

Từ góc nhìn về framework, làm sao để viết được hàm như vậy?

Hiện thực:

```
// định nghĩa kiểu interface
type middleware func(http.Handler) http.Handler
// cấu trúc Router
type Router struct {
```

```

// slice gồm các hàm middleware
middlewareChain [] middleware
// mapping cấu trúc routing với name
mux map[string] http.Handler
}
func NewRouter() *Router{
    return &Router{}
}
// mỗi khi gọi Use là thêm hàm middleware vào slice
func (r *Router) Use(m middleware) {
    r.middlewareChain = append(r.middlewareChain, m)
}
// mỗi khi gọi Add là thêm phần routing trong đó, áp dụng các middleware vào
func (r *Router) Add(route string, h http.Handler) {
    var mergedHandler = h
    // duyệt theo thứ tự ngược lại để apply middleware
    for i := len(r.middlewareChain) - 1; i >= 0; i-- {
        mergedHandler = r.middlewareChain[i](mergedHandler)
    }
    // cuối cùng register hàm handler vào name route tương ứng
    r mux[route] = mergedHandler
}

```

Liên kết

- Phần tiếp theo: [Kiểm tra tính hợp lệ của request](#)
- Phần trước: [Routing trong Web](#)
- [Mục lục](#)

4.4 Kiểm tra tính hợp lệ của request

Một nguyên tắc quan trọng trong lập trình web là không được hoàn toàn tin những gì mà user gửi lên, luôn phải có các cơ chế xác thực, kiểm tra tính hợp lệ của các request từ client để tránh nguy cơ bảo mật, phá rối hệ thống. Từ câu chuyện xác thực request đã nảy sinh các vấn đề xung quanh khác mà chúng ta sẽ giải quyết tiếp sau đây.

Có lẽ bạn đã bắt gặp đâu đó tám hình mà mọi người dùng để chế giễu cấu trúc của PHP:

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^([a-zA-Z\d]{2,64})$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if (!$_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```



'Hadouken' if-else

Thực tế đây là một trường hợp không liên quan gì tới ngôn ngữ mà chỉ là cách tổ chức code rườm rà khi gặp trường hợp mà nhiều field cần validate.

Trong phần này chúng ta sẽ dùng Go để viết một ví dụ validate và xem xét cải tiến nó theo 2 bước. Cuối cùng là phân tích cơ chế để hiểu rõ hơn cách một validator hoạt động.

4.4.1 Cải tiến 1: Tái cấu trúc hàm validation

Giả sử dữ liệu được liên kết tới một struct cụ thể thông qua binding bằng một thư viện Open source.

```
type RegisterReq struct {
    // tag giúp json package encode giá trị của Username
    // thành giá trị tương ứng với key username trong json obj
    Username     string `json:"username"`
    PasswordNew  string `json:"password_new"`
    PasswordRepeat string `json:"password_repeat"`
    Email        string `json:"email"`
}

// register nhận vào obj kiểu RegisterReq và thực hiện validate
// các trường trong đó.
```

```

func register(req RegisterReq) error{
    if len(req.Username) > 0 {
        if len(req.PasswordNew) > 0 && len(req.PasswordRepeat) > 0 {
            if req.PasswordNew == req.PasswordRepeat {
                if emailFormatValid(req.Email) {
                    createUser()
                    return nil
                } else {
                    return errors.New("invalid email")
                }
            } else {
                return errors.New("password and reinput must be the same")
            }
        } else {
            return errors.New("password and password reinput must be longer than 0")
        }
    } else {
        return errors.New("length of username cannot be 0")
    }
}

```

Giờ code của chúng ta có vẻ khá giống một "*Hadouken*" nhắc ở phần đầu rồi, vậy làm thế nào để tối ưu đoạn code trên?

Có một giải pháp đã được đưa ra trong [Refactoring.com - Guard Clauses](#), thử áp dụng cho trường hợp của chúng ta:

```

func register(req RegisterReq) error{
    if len(req.Username) == 0 {
        return errors.New("length of username cannot be 0")
    }

    if len(req.PasswordNew) == 0 || len(req.PasswordRepeat) == 0 {
        return errors.New("password and password reinput must be longer than 0")
    }

    if req.PasswordNew != req.PasswordRepeat {
        return errors.New("password and reinput must be the same")
    }

    if emailFormatValid(req.Email) {
        return errors.New("invalid email")
    }

    createUser()
    return nil
}

```

Nhờ bỏ đi cách viết if-else lồng nhau mà code trở nên "clean" hơn. Tuy vậy chúng ta vẫn phải viết khá nhiều hàm validate cho mỗi field trong một kiểu request.

Có một cách giúp chúng ta giảm khá nhiều code là sử dụng validator.

4.4.2 Cải tiến 2: Sử dụng validator



Thư viện [validator](#) hỗ trợ việc validate bằng cách sử dụng các tag lúc định nghĩa struct. Một ví dụ nhỏ:

```

import (
    "gopkg.in/go-playground/validator.v9"
    "fmt"
)

// RegisterReq là struct cần được validate
type RegisterReq struct {
    // gt = 0 cho biết độ dài chuỗi phải > 0, gt: greater than
    Username      string `json:"username" validate:"gt=0"`
    PasswordNew   string `json:"password_new" validate:"gt=0"`

    // eqfield kiểm tra các trường bằng nhau
    PasswordRepeat string `json:"password_repeat" validate:"eqfield=PasswordNew"`

    // kiểm tra định dạng email thích hợp
    Email         string `json:"email" validate:"email"`
}

// dùng 1 instance của Validate, cache lại struct info
var validate *validator.Validate

// validatefunc để wrap hàm validate.Struct
func validatefunc(req RegisterReq) error {
    err := validate.Struct(req)
    if err != nil {
        return err
    }
    return nil
}

func main() {
    validate = validator.New()

    // khởi tạo obj để test validator
    a := RegisterReq{
        Username      : "Alex",
        PasswordNew   : "",
        PasswordRepeat: "z",
        Email         : "z@z.z",
    }

    err := validatefunc(a)
    fmt.Println(err)
}

// kết quả:
// Key: 'RegisterReq.PasswordNew' Error:Field validation for 'PasswordNew' failed on the 'gt' tag
// Key: 'RegisterReq.PasswordRepeat' Error:Field validation for 'PasswordRepeat' failed on the 'eqfield' tag

```

Một lưu ý nhỏ là error message trả về cho người dùng thì không nên viết trực tiếp bằng tiếng Anh mà thông tin về error nên được tổ chức theo từng tag để người dùng theo đó tra cứu.

4.4.3 Cơ chế của validator

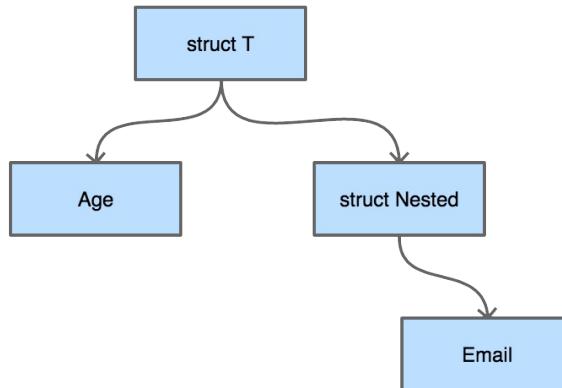
Từ quan điểm cấu trúc, mỗi struct có thể được xem như một cây. Giả sử chúng ta có một struct được định nghĩa như sau:

```

type Nested struct {
    Email string `validate:"email"`
}
type T struct {
    Age   int `validate:"eq=10"`
    Nested Nested
}

```

Sẽ được vẽ thành một cây như bên dưới:



Cây validator

Việc validate các trường có thể thực hiện khi đi qua cấu trúc cây này (bằng cách duyệt theo chiều sâu hoặc theo chiều rộng). Tiếp theo chúng ta sẽ minh họa cơ chế validate trên một cấu trúc như thế, mục đích để hiểu rõ hơn cách mà validator thực hiện.

Đầu tiên xác định 2 struct như hình trên:

```

package main

import (
    "fmt"
    "reflect"
    "regexp"
    "strconv"
    "strings"
)

type Nested struct {
    // validate định dạng email
    Email string `validate:"email"`
}

type T struct {
    // chỉ cho phép age = 10
    Age     int `validate:"eq=10"`
    Nested Nested
}
  
```

Định nghĩa hàm validate:

```

// validateEmail giúp xử lý các tag email
func validateEmail(input string) bool {
    if pass, _ := regexp.MatchString(
        `^(([^\<()\\[\]\\.,;:\\s@"]+([^\<()\\[\]\\.,;:\\s@"]+)*|(.+))@((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.)|(([a-zA-Z-0-9]+\\.)+[a-zA-Z]{2,}))$`, input,
    ); pass {
        return true
    }
    return false
}

// validate thực hiện công việc validate cho interface bất kỳ
// ở đây chỉ hiện thực cho kiểu T
func validate(v interface{}) (bool, string) {
    validateResult := true
    errmsg := "success"

    // xác định type và value của interface input
  
```

```

vt := reflect.TypeOf(v)
vv := reflect.ValueOf(v)

// lần lượt duyệt trên mỗi field của struct
for i := 0; i < vv.NumField(); i++ {
    // phân giải tag để áp dụng validate thích hợp
    fieldVal := vv.Field(i)
    tagContent := vt.Field(i).Tag.Get("validate")
    k := fieldVal.Kind()

    // điều kiện xét trên kiểu field của struct cần validate
    switch k {

        // trường hợp field là int
        case reflect.Int:
            // thực hiện validate cho tag eq=10
            val := fieldVal.Int()
            tagValStr := strings.Split(tagContent, "=")
            tagVal, _ := strconv.ParseInt(tagValStr[1], 10, 64)
            if val != tagVal {
                errmsg = "validate int failed, tag is: "+ strconv.FormatInt(
                    tagVal, 10,
                )
                validateResult = false
            }

        // trường hợp field là string
        case reflect.String:
            val := fieldVal.String()
            tagValStr := tagContent
            switch tagValStr {

                // nếu tag là email thì thực hiện validate tương ứng
                case "email":
                    nestedResult := validateEmail(val)
                    if nestedResult == false {
                        errmsg = "validate mail failed, field val is: "+ val
                        validateResult = false
                    }
            }

        // nếu có struct lồng bên trong thì truyền
        // xuống đệ quy theo chiều sâu
        case reflect.Struct:
            valInter := fieldVal.Interface()
            nestedResult, msg := validate(valInter)
            if nestedResult == false {
                validateResult = false
                errmsg = msg
            }
    }
}

return validateResult, errmsg
}

```

Sau đây là cách sử dụng trong hàm main:

```

func main() {
    // khởi tạo obj để test
    var a = T{Age: 10, Nested: Nested{Email: "abc@adfgom"}}

    validateResult, errmsg := validate(a)
    fmt.Println(validateResult, errmsg)
}

// kết quả:
// false validate mail failed, field val is: abc@adfgom

```

Thư viện validator được giới thiệu trong phần trước phức tạp hơn về mặt chức năng so với ví dụ ở đây. Nhưng nguyên tắc chung cũng là duyệt cây của một struct với reflection.

4.4.4 Xác thực request bằng JWT

Phần trên đã trình bày quá trình validate các thông tin về email và password khi đăng ký một tài khoản. Sau đó, nếu họ đăng nhập vào tài khoản bằng email và password thì trạng thái phiên làm việc của họ sẽ được giữ cho các yêu cầu kế tiếp. Có một số giải pháp để lưu trữ phiên làm việc bằng [session/cookie](#), một giải pháp khác là dùng cơ chế cấp token [JWT](#) sau khi đăng nhập, và dùng token này để xác thực các yêu cầu về sau.

Không chỉ lưu trữ phiên làm việc, token JWT cũng hay đi kèm trong các lệnh gọi API để xác thực phía client khi gọi đến web service. Sau đây là một đoạn chương trình middleware xác thực yêu cầu bằng JWT:

[auth.go](#):

```
var JwtAuthentication = func(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // danh sách các API không cần xác thực bằng token
        notAuth := []string{"/api/user/new", "/api/user/login"}
        requestPath := r.URL.Path
        for _, value := range notAuth {
            if value == requestPath {
                next.ServeHTTP(w, r)
                return
            }
        }

        response := make(map[string] interface{})
        tokenHeader := r.Header.Get("Authorization")
        // thiếu jwt token, trả về lỗi
        if tokenHeader == "" {
            response = u.Message(false, "Missing auth token")
            w.WriteHeader(http.StatusForbidden)
            w.Header().Add("Content-Type", "application/json")
            u.Respond(w, response)
            return
        }
        // thông thường chuỗi token có định dạng: Bearer {token-body}, nên cần tách phần token ra
        splitted := strings.Split(tokenHeader, " ")
        if len(splitted) != 2 {
            response = u.Message(false, "Invalid/Malformed auth token")
            w.WriteHeader(http.StatusForbidden)
            w.Header().Add("Content-Type", "application/json")
            u.Respond(w, response)
            return
        }
        // chuỗi jwt token trong phần header của request
        tokenPart := splitted[1]
        tk := &models.Token{}

        token, err := jwt.ParseWithClaims(tokenPart, tk, func(token *jwt.Token) (interface{}, error) {
            return []byte(os.Getenv("token_password")), nil
        })

        if err != nil {
            response = u.Message(false, "Malformed authentication token")
            w.WriteHeader(http.StatusForbidden)
            w.Header().Add("Content-Type", "application/json")
            u.Respond(w, response)
            return
        }

        if !token.Valid {
            response = u.Message(false, "Token is not valid.")
        }
    })
}
```

```
w.WriteHeader(http.StatusForbidden)
w.Header().Add("Content-Type", "application/json")
u.Respond(w, response)
return
}
ctx := context.WithValue(r.Context(), "user", tk.UserId)
r = r.WithContext(ctx)
// tiếp tục thực hiện request
next.ServeHTTP(w, r)
});
}
```

Liên kết

- Phần tiếp theo: [Làm việc với Database](#)
- Phần trước: [Middleware](#)
- [Mục lục](#)

4.5 Làm việc với Database

Phần này sẽ phân tích các thư viện `database/sql` tiêu chuẩn, giới thiệu một số **ORM** (Object Relational Mapping) và SQL Builder Open source được sử dụng rộng rãi. Cuối cùng là đánh giá công nghệ nào phù hợp nhất đứng ở góc độ phát triển ứng dụng doanh nghiệp.

4.5.1 Bắt đầu từ `database/sql`

Go cung cấp một package `database/sql` để làm việc với cơ sở dữ liệu cho người dùng. Package này cung cấp một interface và các hàm để vận hành cơ sở dữ liệu như quản lý nhóm kết nối, liên kết dữ liệu (data binding), transaction, xử lý lỗi, và vài chức năng khác.

Để giao tiếp với một cơ sở dữ liệu nhất định như MySQL, bạn phải cung cấp driver MySQL như sau:

```
import "database/sql"
import _ "github.com/go-sql-driver/mysql"

// Open để tạo ra một database handle
db, err := sql.Open("mysql", "user:password@/dbname")
```

Xem một chút về hàm `init`:

```
func init() {
    // Register giúp db driver available với "mysql".
    // nếu hàm này được gọi 2 lần cùng 1 tên db hoặc
    // driver nil sẽ gây ra panic.
    sql.Register("mysql", &MySQLDriver{})
}
```

Interface `Driver` trong package `sql`:

```
type Driver interface {
    Open(name string) (Conn, error)
}
```

`sql.Open()` trả về đối tượng `db` từ lời gọi hàm `conn`

```
type Conn interface {
    Prepare(query string) (Stmt, error)
    Close() error
    Begin() (Tx, error)
}
```

Trong thực tế, nếu nhìn vào code của `database/sql/driver/driver.go` sẽ thấy rằng tất cả các thành phần trong file đều là interface cả. Tuỳ vào kiểu trong này mà ta sẽ phải gọi tới những phương thức `driver` phù hợp.

Ở phía người dùng, trong process sử dụng package `database/sql`, ta có thể sử dụng các hàm được cung cấp trong những interface kề trên, hãy nhìn vào một ví dụ hoàn chỉnh sử dụng `database/sql` và `go-sql-driver/mysql`:

```
package main

import (
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
```

```

)
func main() {
    // db là một đối tượng của kiểu sql.DB,
    // tùy chọn kết nối có thể được đặt trong phương thức sql.DB, ở đây bỏ qua
    db, err := sql.Open("mysql", "user:password@tcp(127.0.0.1:3306)/hello")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    var (
        id int
        name string
    )

    // Query thực thi câu query và trả về các rows.
    rows, err := db.Query("select id, name from users where id = ?", 1)
    if err != nil {
        log.Fatal(err)
    }

    // Giải phóng kết nối khi rows.Close() thực thi
    defer rows.Close()

    // Next chuẩn bị row kết quả kế tiếp để đọc với Scan
    for rows.Next() {
        err := rows.Scan(&id, &name)
        if err != nil {
            log.Fatal(err)
        }
        log.Println(id, name)
    }

    // Err trả về lỗi nếu có trong quá trình lặp
    err = rows.Err()
    if err != nil {
        log.Fatal(err)
    }
}

```

Nếu bạn đọc muốn biết `database/sql` chi tiết hơn, có thể xem tại <http://go-database-sql.org/>.

Một vài hiện thực bao gồm các hàm, giới thiệu, cách sử dụng, các cảnh báo và các phản trực quan (counter-intuition) về thư viện (ví dụ như `sql.DB`, các truy vấn trong cùng goroutine có thể ở trên nhiều connections) đều được đề cập, và chúng sẽ không được nhắc tới nữa trong chương này.

Có thể thấy rằng hàm cung cấp `db` của thư viện chuẩn quá đơn giản. Chúng ta cần phải viết code SQL mỗi lần truy cập database để đọc dữ liệu, điều này có thể dẫn đến nguy cơ SQL Injection nếu xử lý không cẩn thận.

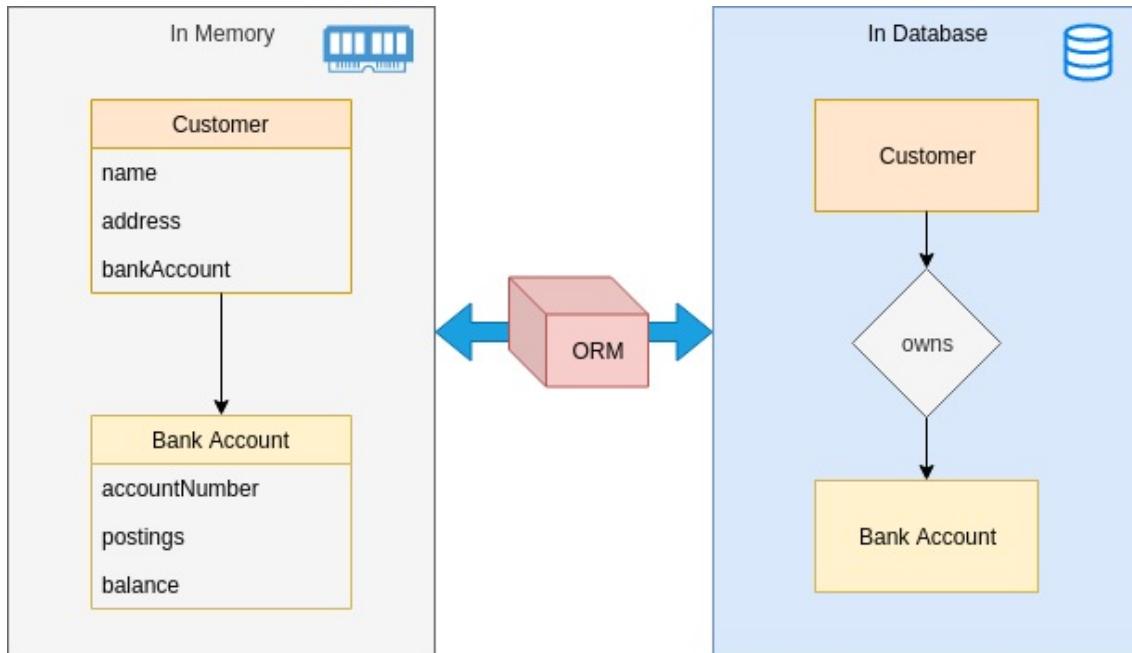
Sau đây sẽ là 2 cách khác để làm điều tương tự: SQL Builder và ORM.

4.5.2 Dùng ORM để tăng hiệu suất

Hãy xem định nghĩa của ORM trên wikipedia:

Object-relational mapping (ORM, O/RM, and O/R mapping tool) trong khoa học máy tính là một kĩ thuật lập trình cho phép chuyển đổi dữ liệu giữa các hệ thống kiểu không tương thích bằng ngôn ngữ hướng đối tượng. Điều này tạo ra một "cơ sở dữ liệu hướng đối tượng ảo" có thể được sử dụng từ trong ngôn ngữ lập trình.

Thông thường ORM thực hiện việc mapping từ database tới các class hoặc struct của chương trình.



Minh họa mapping giữa Database và Struct trong memory

Mục đích của ORM là che chấn lớp DB khỏi người sử dụng. ORM định nghĩa class hoặc struct, sau đó sử dụng một cú pháp cụ thể để tạo ra struct tương ứng 1-1. Sau đó, ta có thể thực hiện các thao tác khác nhau trên các đối tượng đã map từ các bảng trong cơ sở dữ liệu như SAVE, CREATE, DELETE,... . Đôi với những gì ORM đã thực hiện ở bên dưới, ta không cần phải rõ ràng. Khi sử dụng ORM, chúng ta thường sẽ không quan tâm cơ sở dữ liệu.

Ví dụ: ta có nhu cầu hiển thị cho người dùng danh sách sản phẩm mới nhất, giả định rằng `product` và `shop` có mối quan hệ 1:1, có thể thể hiện bằng đoạn code sau:

```
# mã giả
shopList := []
for product in productList {
    shopList = append(shopList, product.GetShop)
}
```

Công cụ như ORM là để bảo vệ cơ sở dữ liệu ngay từ điểm bắt đầu, cho phép vận hành cơ sở dữ liệu gần hơn với cách suy nghĩ của con người. Vì vậy, nhiều lập trình viên dù mới tiếp xúc với ORM cũng có thể code được.

Đoạn code trên sẽ phóng to yêu cầu đọc cơ sở dữ liệu theo hệ số của N. Nói cách khác, nếu danh sách sản phẩm có 15 SKU (Stock-Keeping Unit), mỗi lần người dùng mở trang, ít nhất 1 (danh sách mục truy vấn) + 15 (yêu cầu thông tin cửa hàng liên quan đến truy vấn) là bắt buộc. Ở đây N là 16. Nếu trang danh sách khá lớn, giả sử 600 mục, thì ta phải thực hiện ít nhất 1 + 600 truy vấn.

Nếu số lượng truy vấn đơn giản lớn nhất mà cơ sở dữ liệu có thể chịu được là 120 000 QPS và truy vấn trên chỉ là truy vấn được sử dụng phổ biến nhất, thì khả năng service có thể cung cấp là bao nhiêu? 200 QPS! Một trong những nguyên tắc cấm kỵ của các hệ thống trên Internet là sự khuếch đại số lượng thao tác đọc không cần thiết này.

Tất nhiên bạn có thể nói rằng đó không phải là vấn đề của ORM. Nếu viết bằng sql ta vẫn có thể viết được một chương trình giống vậy, hãy nhìn vào demo sau:

```
o := orm.NewOrm()
num, err := o.QueryTable("cardgroup").Filter("Cards__Card__Name", cardName).All(&cardgroups)
```

Nhiều ORM cung cấp kiểu truy vấn `Filter` này, nhưng trên thực tế, đằng sau ORM còn ẩn nhiều thao tác chi tiết khác, chẳng hạn như tạo ra câu lệnh SQL tự động `limit 1000`.

Có lẽ nhiều người trong chúng ta không hề biết có thao tác đó. Thực ra trong tài liệu chính thức của ORM đã nói qua rằng tất cả các truy vấn sẽ tự động `limit 1000` mà **không cần chỉ định rõ**, chính vì vậy mà điều này trở nên khó khăn đối với nhiều người chưa đọc tài liệu hoặc đọc mã nguồn của ORM. Những lập trình viên thích ngôn ngữ ràng buộc kiểu mạnh thường không thích những gì ngôn ngữ tự thực hiện ngầm định, chẳng hạn như chuyển đổi kiểu ngầm của các ngôn ngữ khác nhau trong thao tác gán để rồi mất đi độ chính xác trong chuyển đổi, điều này chắc chắn sẽ khiến họ đau đầu. Vì vậy, càng có ít thứ mà thư viện làm ẩn bên dưới thì càng tốt. Nếu ta cần thực hiện điều gì hãy thực hiện nó ở một nơi dễ thấy. Trong ví dụ trên, tốt hơn hết là loại bỏ hành vi tự hành động ngầm định này hoặc là bắt buộc người dùng phải truyền vào tham số `limit`.

Ngoài vấn đề `limit`, chúng ta hãy xem truy vấn này dưới đây:

```
num, err := o.QueryTable("cardgroup").Filter("Cards__Card__Name", cardName).All(&cardgroups)
```

Bạn có thấy rằng `Filter` này là một thao tác JOIN không? Rất khó để nhận ra vì ORM đã che giấu quá nhiều chi tiết khỏi thiết kế. Cái giá của sự tiện lợi là những hoạt động ẩn đằng sau nó hoàn toàn nằm ngoài kiểm soát. Một dự án như vậy sẽ trở nên ngày càng khó theo dõi và bảo trì chỉ sau một vài lần nâng cấp.

Tất nhiên chúng ta không thể phủ nhận được tầm quan trọng của ORM. Mục đích ban đầu của nó là loại bỏ việc triển khai cụ thể các hoạt động với database và lưu trữ dữ liệu. Nhưng một số công ty đã dần xem ORM là một thiết kế thất bại vì các chi tiết quan trọng bị ẩn giấu khá nhiều. Các chi tiết này rất quan trọng đối với sự phát triển về lâu dài của các hệ thống cần mở rộng quy mô.

4.5.3 Dùng SQL Builder để tăng hiệu suất

So sánh với ORM, SQL Builder đạt được sự cân bằng tốt hơn giữa SQL và khả năng bảo trì của dự án. Đầu tiên, sql builder không ẩn quá nhiều chi tiết như ORM nhưng cũng khá đơn giản để sử dụng:

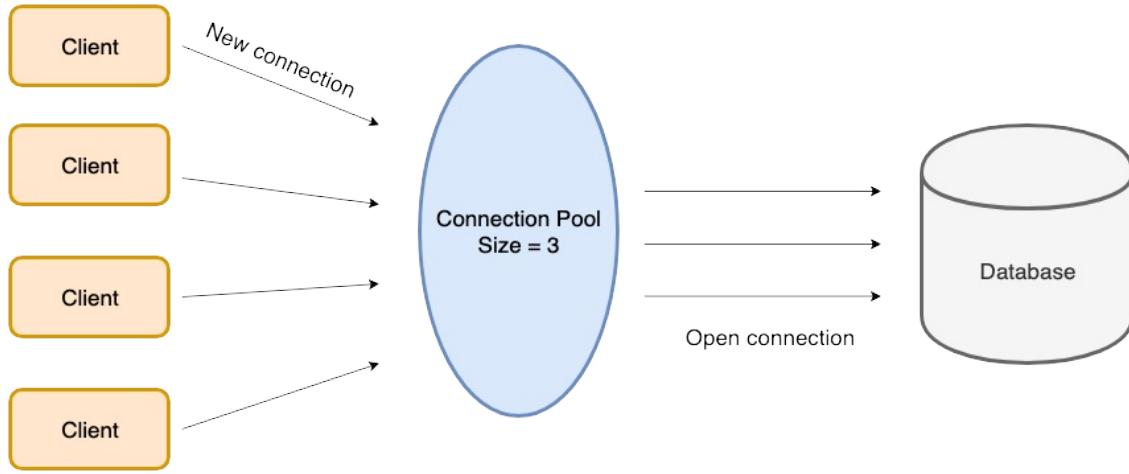
```
// câu truy vấn có điều kiện where
where := map[string]interface{} {
    "order_id > ?" : 0,
    "customer_id != ?" : 0,
}
limit := []int{0,100}
orderBy := []string{"id asc", "create_time desc"}

// get list kết quả từ các thành phần khởi tạo phía trên
orders := orderModel.GetList(where, limit, orderBy)
```

Việc code và đọc hiểu SQL Builder đều không gặp khó khăn gì. Chuyển đổi những dòng code này thành sql cũng không cần quá nhiều nỗ lực.

Nói một cách dễ hiểu, SQL Builder là một cách biểu diễn ngôn ngữ đặc biệt của sql trong mã. Nếu bạn không có DBA, nhưng R&D có khả năng phân tích và tối ưu hóa sql hoặc DBA của công ty bạn không phản đối các kiểu ngôn ngữ sql như thế này thì bạn sử dụng SQL Builder là một lựa chọn tốt.

4.5.4 Sử dụng connection pool để tăng hiệu suất



Minh họa Connection Pool

Một trong những kỹ thuật quan trọng khi làm việc với database là sử dụng **connection pool**. Một connection xem như là một bộ đệm duy trì các kết nối tới database. Một connection pool là tập hợp nhiều connection tới database.

Cơ chế hoạt động của connection pool khá đơn giản, khi một connection được tạo thì connection đó sẽ được đưa vào pool và được sử dụng lại cho các yêu cầu kết nối tiếp theo cho đến khi bị đóng hoặc hết thời gian chờ (timeout). Khi người dùng gửi yêu cầu kết nối đến hệ thống, hệ thống sẽ kiểm tra xem trong pool có connection nào chưa được sử dụng không. Có hai trường hợp xảy ra:

- Nếu có connection chưa được sử dụng, hệ thống sẽ cung cấp connection đó cho người dùng để xử lý các yêu cầu kết nối tới database.
- Nếu trong pool không rỗng hoặc không có connection nào đang rảnh và số lượng kết nối trong pool vẫn chưa vượt quá số lượng connection quy định (max connection) thì hệ thống sẽ tạo một connection mới tới database và cung cấp cho người dùng connection đó.
- Nếu trong pool đã hết connection rảnh và pool đã đạt số lượng connection cho phép tạo thì người dùng phải đợi cho đến khi có một connection rảnh được đưa vào pool.

Sử dụng connection pool có nhiều ưu điểm:

- Tăng hiệu suất khi làm việc với database, vì chúng ta có nhiều kết nối tới database cùng lúc mà không phải đợi tuần tự.
- Không phải tốn chi phí thời gian khởi tạo connection và đóng connection cho mỗi yêu cầu kết nối tới database vì trong pool đã có sẵn connection được khởi tạo rồi.
- Sử dụng tài nguyên hệ thống hợp lý, khi chúng ta có thể tận dụng lại các connection đã sử dụng và giới hạn được số lượng connection được mở.

Khi sử dụng package `database/sql`, thì mặc định package đã hỗ trợ chúng ta phần connection pool. Nhưng chúng ta có thể cấu hình lại connection pool để sử dụng hiệu quả hơn.

Sử dụng hàm SetMaxOpenConns Cấu hình số lượng connection lớn nhất có thể được mở.

```

func main() {
    db, err := sql.Open("mysql",
        "username:password@tcp(127.0.0.1:3306)/hello")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    // mặc định là 0 (không giới hạn)
    // nếu giá trị truyền vào max <= 0 cũng sẽ là không giới hạn
    db.SetMaxOpenConns(10)
}

```

```
}
```

Sử dụng hàm SetMaxIdleConns Cấu hình số lượng connection rảnh có trong pool. Chỉ số này luôn nhỏ hơn hoặc bằng chỉ số MaxOpenConns. Nếu chúng ta cấu hình cao hơn thì thư việc sẽ tự điều chỉnh giảm lại cho phù hợp với chỉ số MaxOpenConns.

```
func main() {
    db, err := sql.Open("mysql",
        "username:password@tcp(127.0.0.1:3306)/hello")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    // mặc định là 2 connection rảnh
    // nếu giá trị truyền vào max <= 0 là không có connection rảnh được giữ lại
    db.SetMaxIdleConns(10)
}
```

Sử dụng hàm SetConnMaxLifetime Cấu hình thời gian tối đa của một connection được sử dụng lại. Sau khi hết thời gian quy định thì connection sẽ bị đóng lại.

```
func main() {
    db, err := sql.Open("mysql",
        "username:password@tcp(127.0.0.1:3306)/hello")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

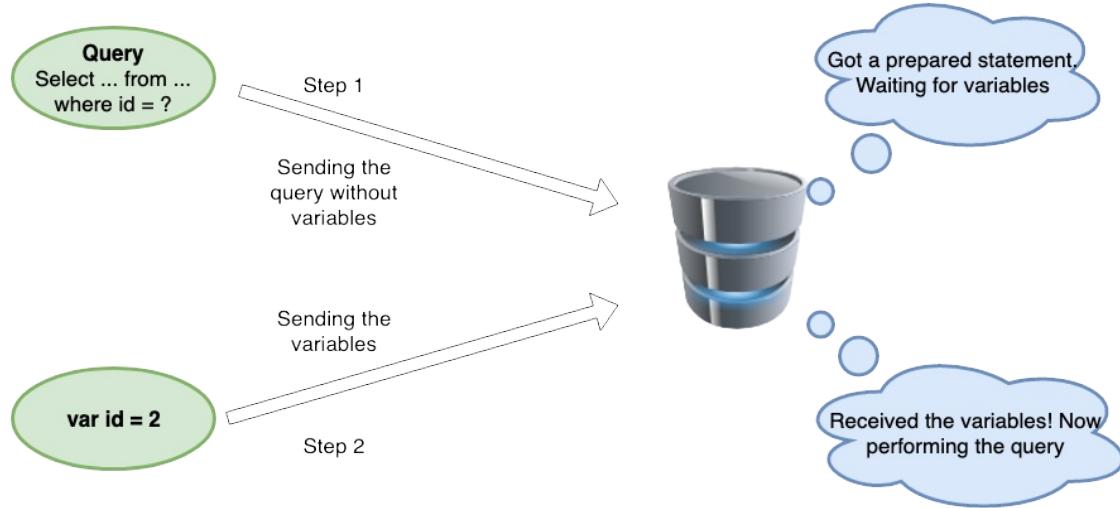
    // mặc định là các connection không bị đóng
    // nếu giá trị truyền vào max <= 0 là các connection sẽ được sử dụng lại mãi và không bị đóng
    db.SetConnMaxLifetime(10 * time.Hour)
}
```

Các bạn có thể tìm hiểu thêm về cấu hình connection pool [ở đây](#).

4.5.5 Sử dụng Prepared Statement để tăng hiệu suất

Để tối ưu hiệu năng của hệ thống, có rất nhiều cách để thực hiện nhưng hiệu quả nhất vẫn là tối ưu các câu truy vấn database. Một trong số này đó là sử dụng `prepared statement` để truy vấn.

Prepared statement là một tính năng được sử dụng để thực hiện lặp lại các câu lệnh SQL tương tự nhau với hiệu quả cao. Ví dụ minh họa sau:



Minh họa cơ chế Prepare statement

```

package main
import (
    _ "github.com/go-sql-driver/mysql"
    "database/sql"
    "log"
)

func main(){
    db, err := sql.Open("mysql", "root:root@tcp(127.0.0.1:3306)/hello")
    if err != nil {
        log.Fatal(err)
    }

    stmt, err := db.Prepare("SELECT * FROM accounts WHERE id = ?;")
    res,err:=stmt.Exec(2)
    res,err=stmt.Exec(3)

    if err!=nil{
        log.Fatal(err)
    }
    log.Println(res)
}

```

- **Prepare:** đầu tiên, ứng dụng tạo ra 1 statement template và gửi nó cho DBMS. Các giá trị không được chỉ ra và được gọi là parameters (dấu ? bên dưới) `SELECT * FROM accounts WHERE id = ?;`
- **Compile:** (parse, optimizes và translates) statement template , store kết quả mà không thực thi. Quá trình này do DBMS thực hiện.
- **Execute:** ứng dụng gửi giá trị của parametes của statement template và DBMS thực thi nó. Ứng dụng có thể thực thi statement nhiều lần với nhiều giá trị khác nhau.

Ưu điểm khi sử dụng prepared statement:

- Overhead của compile statement diễn ra 1 lần còn statement được thực thi nhiều lần. Về lý thuyết, khi sử dụng prepared statement, ta sẽ tiết kiệm được: `cost_of_prepare_preprocessing * (#statement_executions - 1)`. Nhưng thực tế, tùy từng loại query sẽ có cách optimize khác nhau ([chi tiết](#)).
- Chống [SQL injection](#).
- Phát hiện sớm các lỗi cú pháp trong câu statement.
- Có thể `cache prepared statement` và sử dụng lại sau này.

Các bạn có thể xem chi tiết hơn bài [blog](#) này.

Liên kết

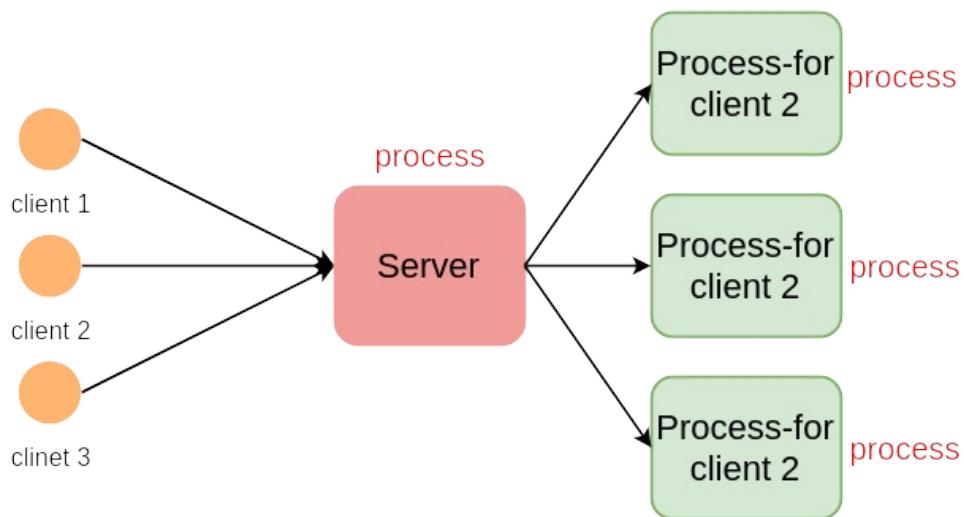
- Phần tiếp theo: [Giới hạn lưu lượng Service](#)
- Phần trước: [Kiểm tra tính hợp lệ của request](#)
- [Mục lục](#)

4.6 Giới hạn lưu lượng Service

Một chương trình máy tính có thể mắc phải một số các vấn đề bottleneck (tắc nghẽn):

- Bottleneck do CPU tính toán.
- Bottleneck do băng thông mạng.
- Đôi khi do external system gây ra tình trạng bottleneck trong chính hệ thống phân tán của nó.

Phần quan trọng nhất của một hệ thống Web là mạng. Cho dù đó là tiếp nhận, phân tích request của người dùng, truy cập bộ nhớ hay trả về dữ liệu response đều phải truy cập trực tuyến. Trước khi xuất hiện IO multiplexing interface `epoll/kqueue` do hệ thống cung cấp thì từng có một sự cố **C10k** trong máy tính đa lõi.



Vấn đề C10k xảy ra khi số lượng client giữ kết nối vượt 10000

Kể từ khi trên Linux có `epoll`, FreeBSD hiện thực `kqueue`, chúng ta có thể dễ dàng giải quyết vấn đề C10k với API do kernel cung cấp.

Thư viện `net` của Go đóng gói các syscall API khác nhau cho các nền tảng khác nhau. Thư viện `http` được xây dựng trên nền của thư viện `net`, trong Golang chúng ta có thể viết các service `http` hiệu suất cao với sự trợ giúp của thư viện chuẩn. Đây là một service `hello world` đơn giản:

```

package main

import (
    "io"
    "log"
    "net/http"
)

func sayhello(wr http.ResponseWriter, r *http.Request) {
    wr.WriteHeader(200)
    io.WriteString(wr, "hello world")
}

func main() {
    http.HandleFunc("/", sayhello)
    err := http.ListenAndServe(":9090", nil)
    if err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}

```

Chúng ta sẽ đo throughput của Web service này: sử dụng `wrk` trên máy tính cá nhân có cấu hình:

```
ThinkPad-T470
-----
OS: Ubuntu 18.04.2 LTS x86_64
Host: 20HES39900 ThinkPad T470
Kernel: 4.15.0-52-generic

CPU: Intel i5-7200U (4) @ 3.100GHz
GPU: Intel HD Graphics 620
Memory: 2977MiB / 15708MiB
```

Kết quả test:

```
$ wrk -c 10 -d 10s -t10 http://localhost:9090
Running 10s test @ http://localhost:9090
 10 threads and 10 connections
 Thread Stats      Avg      Stdev     Max   +/- Stdev
  Latency    360.57us  769.72us  14.27ms  91.45%
  Req/Sec     7.07k     0.91k    9.25k   76.10%
 704529 requests in 10.01s, 86.00MB read
Requests/sec: 70363.89
Transfer/sec: 8.59MB

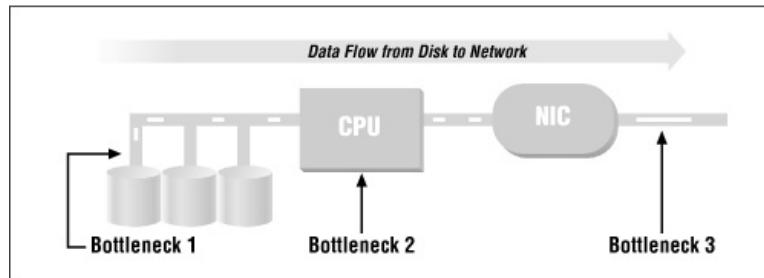
$ wrk -c 10 -d 10s -t10 http://localhost:9090
Running 10s test @ http://localhost:9090
 10 threads and 10 connections
 Thread Stats      Avg      Stdev     Max   +/- Stdev
  Latency    395.85us  0.90ms   18.93ms  91.88%
  Req/Sec     6.92k     0.95k    9.34k   75.30%
 688941 requests in 10.02s, 84.10MB read
Requests/sec: 68783.96
Transfer/sec: 8.40MB

$ wrk -c 10 -d 10s -t10 http://localhost:9090
Running 10s test @ http://localhost:9090
 10 threads and 10 connections
 Thread Stats      Avg      Stdev     Max   +/- Stdev
  Latency    374.92us  814.34us  13.76ms  91.47%
  Req/Sec     7.09k     1.03k    21.18k   82.82%
 706968 requests in 10.09s, 86.30MB read
Requests/sec: 70040.20
Transfer/sec: 8.55MB
```

Kết quả của thử nghiệm là khoảng 70.000 QPS và thời gian phản hồi là khoảng 15ms. Đối với một ứng dụng web, đây đã là một kết quả rất tốt. Đây chỉ là một máy tính cá nhân nên với server có cấu hình cao hơn sẽ đạt được kết quả còn tốt hơn nữa.

Chương trình của chúng ta chưa có chứa logic nghiệp vụ nên có thể dễ dàng đánh giá được QPS như thế, trên thực tế khi gặp một mô-đun có số lượng logic nghiệp vụ lớn và số lượng code lớn, thì vấn đề bottleneck có thể phải trả qua nhiều lần stress test mới có thể phát hiện ra.

Một số hệ thống thường bị bottleneck do mạng, chẳng hạn như dịch vụ CDN và dịch vụ Proxy. Một số là do CPU/GPU, như các dịch vụ xác minh đăng nhập và dịch vụ xử lý hình ảnh. Số khác do truy cập vào disk bị tắc nghẽn như hệ thống lưu trữ, cơ sở dữ liệu. Các nút thắt chương trình khác nhau được phản ánh ở những nơi khác nhau và các dịch vụ đơn chức năng được đề cập ở trên tương đối dễ phân tích.



3 nơi có khả năng tắc nghẽn: disk, CPU, NIC

Đối với trường hợp nút cỗ chai IO/Network, throughput ở chỗ NIC/disk IO (NIC - Network Interface Card) sẽ đầy trước CPU. Trong trường hợp này, ngay cả khi CPU được tối ưu hóa, throughput của toàn bộ hệ thống vẫn không thể cải thiện mà chỉ có thể tăng tốc độ đọc ghi của đĩa lên. Kích thước memory lớn có thể làm tăng băng thông của NIC để cải thiện hiệu suất tổng thể. Chương trình bottleneck ở CPU là khi mức sử dụng CPU đạt 100% trước lúc bộ nhớ và NIC đầy. CPU luôn ở tình trạng "busy" với nhiều tác vụ tính toán khác nhau trong khi các thiết bị IO thường tương đối nhàn rỗi (phản lớn thời gian ở trạng thái chờ).

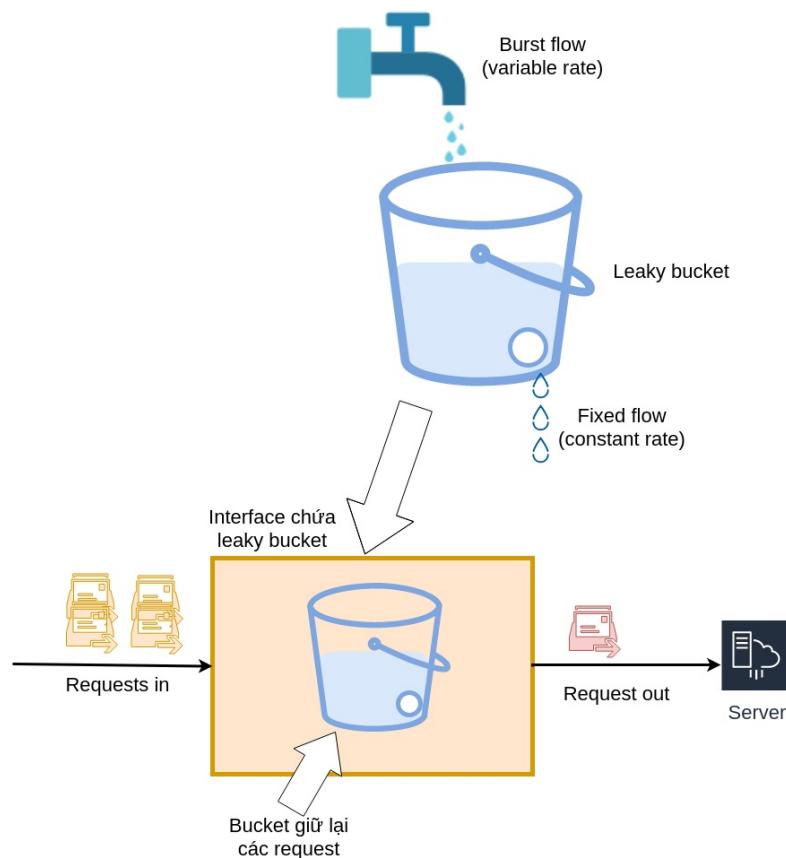
Bất kể là loại service nào, một khi tài nguyên sử dụng đạt đến giới hạn các request sẽ bị dồn lại, khi hết timeout mà không kịp xử lý sẽ dẫn đến treo hệ thống (không phản hồi) và ảnh hưởng trực tiếp tới người dùng cuối. Đối với các Web service phân tán, bottleneck không phải lúc nào cũng nằm trong hệ thống mà nó có thể nằm ở bên ngoài. Các hệ thống không chuyên về tính toán có xu hướng sẽ gặp vấn đề ở cơ sở dữ liệu quan hệ và lúc đó chính bản thân mô-đun Web đã bị bottleneck.

Không quan trọng service của chúng ta bị bottleneck tại đâu, vấn đề cuối cùng vẫn giống nhau đều nằm ở công việc quản lý lưu lượng (traffic).

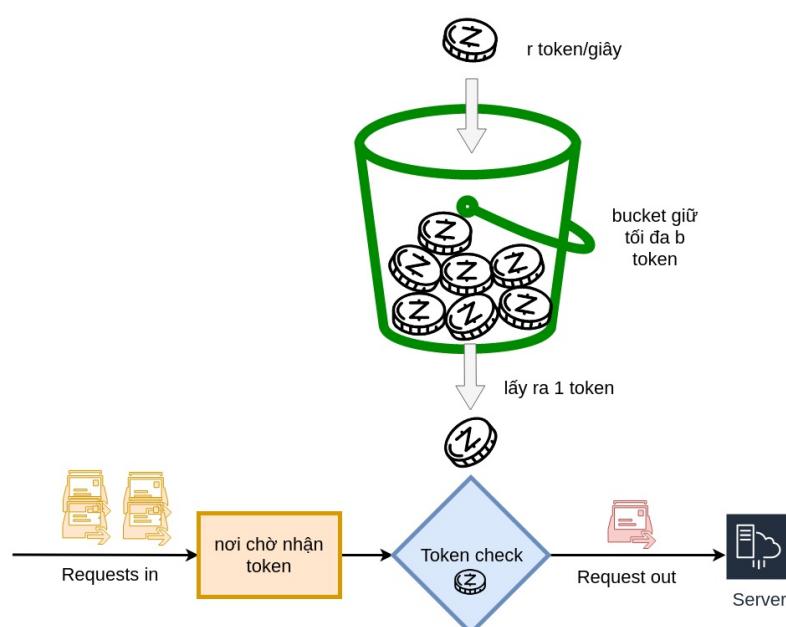
4.6.1 Tầm quan trọng của giới hạn lưu lượng

Có nhiều cách để giới hạn lưu lượng. Phổ biến nhất là leaky buckets và token buckets.

1. **Leaky bucket** có thể hiểu rằng chúng ta có một cái xô chứa đầy nước, và một giọt nước rò rỉ ra sau mỗi khoảng thời gian cố định. Nếu nhận được "giọt nước" thì có thể tiếp tục yêu cầu dịch vụ, ngược lại thì cần phải đợi đến lần nhỏ giọt tiếp theo.



1. **Token bucket** với nguyên tắc token được thêm vào bucket với tốc độ (rate) không đổi. Để có được token từ bucket, số lượng token có thể được điều chỉnh theo số tài nguyên cần sử dụng. Nếu không có token, ta có thể lựa chọn tiếp tục chờ hoặc từ bỏ. Hai phương pháp này nhìn thì tương tự nhau, nhưng thực ra là có một vài điểm khác biệt.



- Tốc độ mà leaky bucket bị rò rỉ (leak) là cố định còn token trong token bucket có thể được lấy ra chỉ khi có token trong bucket.

- Điều đó nghĩa là token bucket chỉ cho phép một mức độ đồng thời nhất định. Ví dụ cùng lúc có 100 yêu cầu người dùng gửi tới, miễn là có 100 token trong bucket thì tất cả 100 yêu cầu sẽ được đưa ra.
- Token bucket cũng có thể suy biến thành mô hình leaky bucket nếu không có token trong bucket.

Trong các ứng dụng thực tế, token bucket được sử dụng rộng rãi và hầu hết các limiter phổ biến hiện nay trong cộng đồng Open source đều dựa trên token bucket. Trên cơ sở này, có một phiên bản limiter là [juju/ratelimit](#) cung cấp một số phương thức thêm vào token với các đặc điểm khác nhau như sau:

```
// fillInterval với ý nghĩa mỗi token sẽ được đặt trong bucket
// một khoảng thời gian time.Duration, số lượng tối đa là
// capacity của bucket, phần vượt quá capacity sẽ bị loại bỏ.
// các bucket được khởi tạo ban đầu đều ở trạng thái đầy.
func NewBucket(fillInterval time.Duration, capacity int64) *Bucket

// khác biệt so với NewBucket() thông thường là cho phép đưa vào
// một kích thước quantum nhất định - quantum tokens ở mỗi
// khoảng thời gian fillInterval.
func NewBucketWithQuantum(fillInterval time.Duration, capacity, quantum int64) *Bucket

// NewBucketWithRate trả về một bucket token với số token được
// đưa vào mỗi giây đạt đến công suất tối đa rate.
// Do độ phân giải hạn chế của clock nên ở tốc độ cao thì tỷ lệ
// thực tế có thể khác tới 1% so với tỷ lệ được chỉ định.
// Ví dụ capacity=100, và rate=0.1, sẽ được một bucket có thể
// thêm vào 10 tokens mỗi giây.
func NewBucketWithRate(rate float64, capacity int64) *Bucket
```

Việc nhận token từ bucket cũng được cung cấp một số API:

```
func (tb *Bucket) Take(count int64) time.Duration {}
func (tb *Bucket) TakeAvailable(count int64) int64 {}
func (tb *Bucket) TakeMaxDuration(count int64, maxWait time.Duration)
    (time.Duration, bool,) {}
func (tb *Bucket) Wait(count int64) {}
func (tb *Bucket) WaitMaxDuration(count int64, maxWait time.Duration) bool {}
```

Tên và chức năng tương của chúng khá đơn giản nên ta sẽ không đi vào chi tiết ở đây. So với công cụ ratelimiter do thư viện Java của Google cung cấp là Guava nổi tiếng hơn trong cộng đồng Open source, thư viện này không hỗ trợ khởi tạo token và không thể sửa đổi dung lượng token ban đầu, do đó có thể không đáp ứng được hết các yêu cầu trong các trường hợp riêng lẻ.

4.6.2 Nguyên tắc

Mô hình token bucket thực ra là một quá trình cộng trừ vào một biến đếm toàn cục, nhưng việc sử dụng biến chung đòi hỏi chúng ta phải thêm các khóa đọc-ghi, chính vì vậy mà trở nên phức tạp. Nếu chúng ta đã quen thuộc với ngôn ngữ Go, bạn có thể nghĩ ngay đến việc dùng một buffered channel để hoàn thành thao tác tạo ra token bucket đơn giản:

```
var tokenBucket = make(chan struct{}, capacity)
```

tokenBucket thêm token vào theo thời gian, nếu bucket đầy thì bỏ qua:

```
fillToken := func() {
    ticker := time.NewTicker(fillInterval)
    for {
        select {
        case <-ticker.C:
            select {
```

```
        case tokenBucket <- struct{}{}:
            default:
                }
                fmt.Println("current token cnt:", len(tokenBucket), time.Now())
            }
        }
    }
}
```

Kết hợp vào code:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    var fillInterval = time.Millisecond * 10
    var capacity = 100
    var tokenBucket = make(chan struct{}, capacity)

    fillToken := func() {
        ticker := time.NewTicker(fillInterval)
        for {
            select {
            case <-ticker.C:
                select {
                case tokenBucket <- struct{}{}:
                default:
                }
                fmt.Println("current token cnt:", len(tokenBucket), time.Now())
            }
        }
    }

    go fillToken()
    time.Sleep(time.Hour)
}
```

Kết quả sau khi thực thi thu được:

```
current token cnt: 98 2019-06-21 13:49:01.932273966 +0700 +07 m==+0.98016304  
current token cnt: 99 2019-06-21 13:49:01.942302937 +0700 +07 m==+0.990192005  
current token cnt: 100 2019-06-21 13:49:01.952350569 +0700 +07 m==+1.000239633  
current token cnt: 100 2019-06-21 13:49:01.962330944 +0700 +07 m==+1.010220033  
current token cnt: 100 2019-06-21 13:49:01.972302925 +0700 +07 m==+1.020191986  
current token cnt: 100 2019-06-21 13:49:01.982292881 +0700 +07 m==+1.030181985  
current token cnt: 100 2019-06-21 13:49:01.992308344 +0700 +07 m==+1.040197432  
current token cnt: 100 2019-06-21 13:49:02.002350638 +0700 +07 m==+1.050239734  
current token cnt: 100 2019-06-21 13:49:02.012318649 +0700 +07 m==+1.060207734  
current token cnt: 100 2019-06-21 13:49:02.022282122 +0700 +07 m==+1.070171206
```

Trong thời gian 1s chương trình đưa ra 100 token. Tuy nhiên có thể thấy bộ đếm thời gian của Go có lỗi khoảng 0,001 giây, vì vậy nếu kích thước bucket lớn hơn 1000, có thể xảy ra một số lỗi. Mặc dù đối với phần lớn service thì lỗi này không đáng kể.

Thao tác cấp token của token bucket ở trên cũng dễ hiện thực hơn, để đơn giản hóa vấn đề, ta chỉ lấy một token như sau đây:

```
func TakeAvailable(block bool) bool{  
    var takenResult bool  
    if block {
```

```

        select {
            case <-tokenBucket:
                takenResult = true
            }
        } else {
            select {
            case <-tokenBucket:
                takenResult = true
            default:
                takenResult = false
            }
        }

        return takenResult
    }
}

```

Ở đây chú ý một chút, token bucket đưa token vào bucket theo các khoảng thời gian cố định. Giả sử:

- Lần cuối token được đưa vào là t1,
- Số token tại thời điểm đó là k1,
- Time interval là ti,
- Mỗi lần đưa x token vào bucket,
- Dung lượng của bucket là giới hạn.

Bây giờ nếu ai đó gọi `TakeAvailable` để lấy n token, ta sẽ ghi lại khoảng khắc này là t2. Vậy tại t2 nên có bao nhiêu token trong bucket token? Mã giả như sau:

```

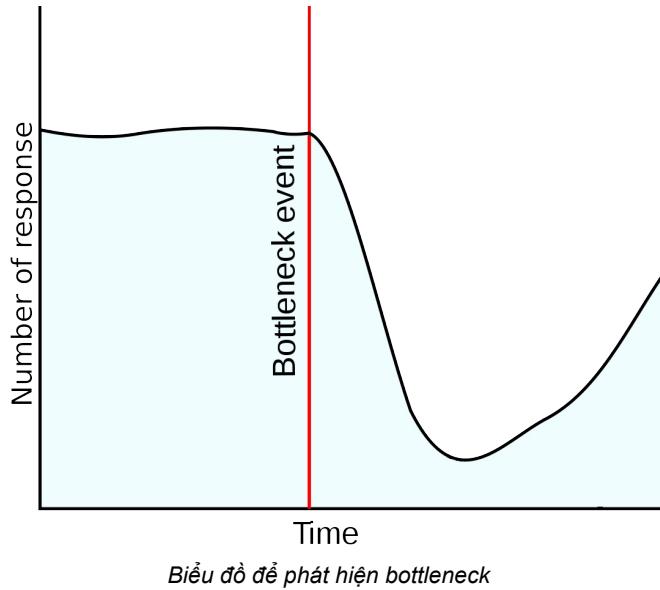
cur = k1 + ((t2 - t1)/ti) * x
cur = cur > cap ? cap : cur

```

Chúng ta sử dụng chênh lệch thời gian giữa t1, t2 kết hợp với các tham số ti, k1 thì có thể biết được số lượng token trong bucket trước khi lấy ra token. Về mặt lý thuyết là không cần thiết sử dụng hoạt động điền token vào channel ở ví dụ trước. Miễn là mỗi lần ta đều tính số lượng token trong bucket thì có thể nhận được số lượng token chính xác. Sau khi nhận được số lượng token rồi thì chỉ cần thực hiện những thao tác cần thiết như phép trừ số lượng token. Hãy nhớ sử dụng lock để đảm bảo an toàn với tính concurrency. Thư viện [juju/ratelimit](#) đang thực hiện theo cách này.

4.6.3 Vấn đề tắc nghẽn Service và Quality of Service

Trước đây chúng ta đã nói nhiều về việc bottleneck ở CPU, IO và một số loại khác nữa, vấn đề này có thể phát hiện tương đối nhanh chóng từ hầu hết các công ty có monitoring, nếu hệ thống gặp vấn đề về hiệu suất thì quan sát biểu đồ monitor về response là phương án nhanh nhất để phát hiện nguyên nhân.



Mặc dù các số liệu hiệu suất là quan trọng, QoS (Quality of Service) tổng thể của dịch vụ cũng cần được xem xét khi cung cấp dịch vụ cho người dùng. QoS bao gồm các số liệu như tính sẵn sàng (availability), thông lượng (throughput), độ trễ (latency, delay variation), mất mát dữ liệu, ...

Nhìn chung, ta có thể cải thiện việc sử dụng CPU của các dịch vụ Web bằng cách tối ưu hóa hệ thống, từ đó tăng throughput của toàn bộ hệ thống.

Nhưng khi throughput được cải thiện, chưa chắc đã có thể cải thiện trải nghiệm người dùng. Người dùng rất nhạy cảm với độ trễ. Dù throughput của hệ thống cao, nhưng nếu không phản hồi được trong một thời gian dài sẽ làm người dùng rất khó chịu. Do đó, trong các chỉ số hiệu suất dịch vụ Web của các công ty lớn, ngoài độ trễ phản hồi trung bình, thời gian phản hồi **95% (p95)** và **99% (p99)** cũng được lấy ra làm tiêu chuẩn hiệu suất. Thời gian phản hồi trung bình thường không ảnh hưởng nhiều đến việc cải thiện hiệu suất sử dụng CPU, quan trọng là thời gian phản hồi 99% so với 95% có thể tăng đáng kể. Từ đó ta có thể xem xét liệu chi phí cải thiện hiệu suất sử dụng CPU này có đáng hay không.

Liên kết

- Phản tiếp theo: [Mô hình của các dự án web](#)
- Phản trước: [Làm việc với Database](#)
- [Mục lục](#)

4.7. Mô hình của các dự án web

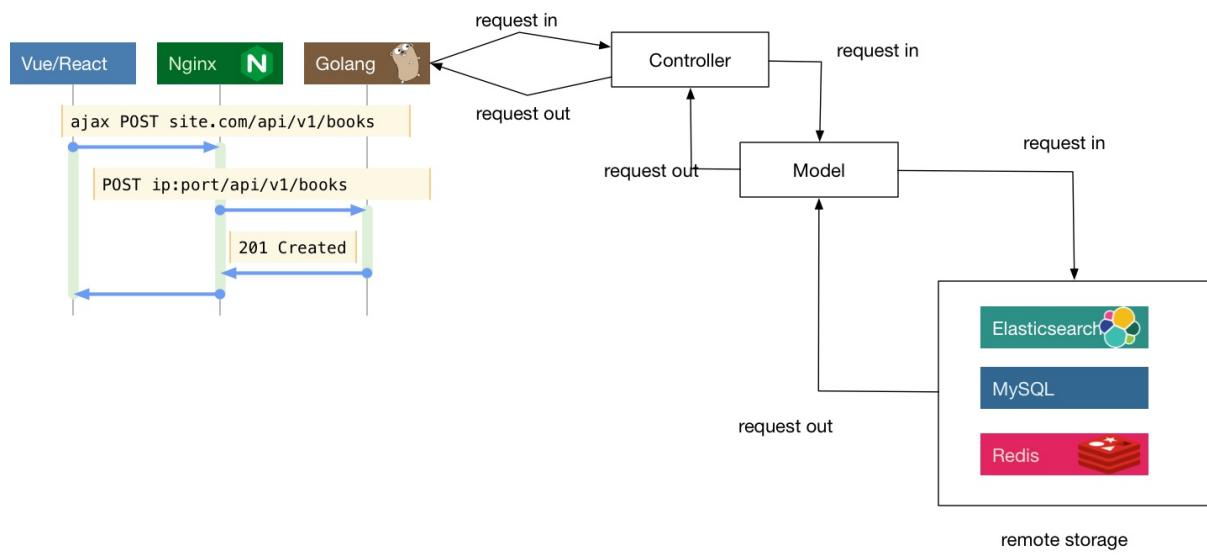
Phần này sẽ trình bày mô hình MVC và đi vào chi tiết các lớp trong một dự án web.

4.7.1 Mô hình MVC

MVC frameworks là những framework rất phổ biến trong việc phát triển web, khái niệm MVC được đề xuất đầu tiên bởi [Trygve Reenskaug](#) vào năm 1978, chương trình MVC gồm ba thành phần:

- **Model** : là nơi chứa những nghiệp vụ tương tác với dữ liệu hoặc hệ quản trị cơ sở dữ liệu (mysql, mssql...); nó sẽ bao gồm các class/function xử lý nhiều nghiệp vụ như kết nối database, truy vấn dữ liệu, thêm - xóa - sửa dữ liệu...
- **View** : là nơi có những giao diện như một nút bấm, khung nhập, menu, hình ảnh... nó đảm nhiệm nhiệm vụ hiển thị dữ liệu và giúp người dùng tương tác với hệ thống.
- **Controller** : là nơi tiếp nhận những yêu cầu xử lý được gửi từ người dùng, nó sẽ gồm những class/ function xử lý nhiều nghiệp vụ logic giúp lấy đúng dữ liệu thông tin cần thiết nhờ các nghiệp vụ lớp Model cung cấp và hiển thị dữ liệu đó ra cho người dùng nhờ lớp View.

Trải qua quá trình phát triển, phần back-end của chương trình ngày càng phức tạp. Để quản lý tốt hơn, những phần như thế sẽ thường phân chia ra thành nhiều kiến trúc con. Hình sau là một lưu đồ của hệ thống từ front-end tới back-end:



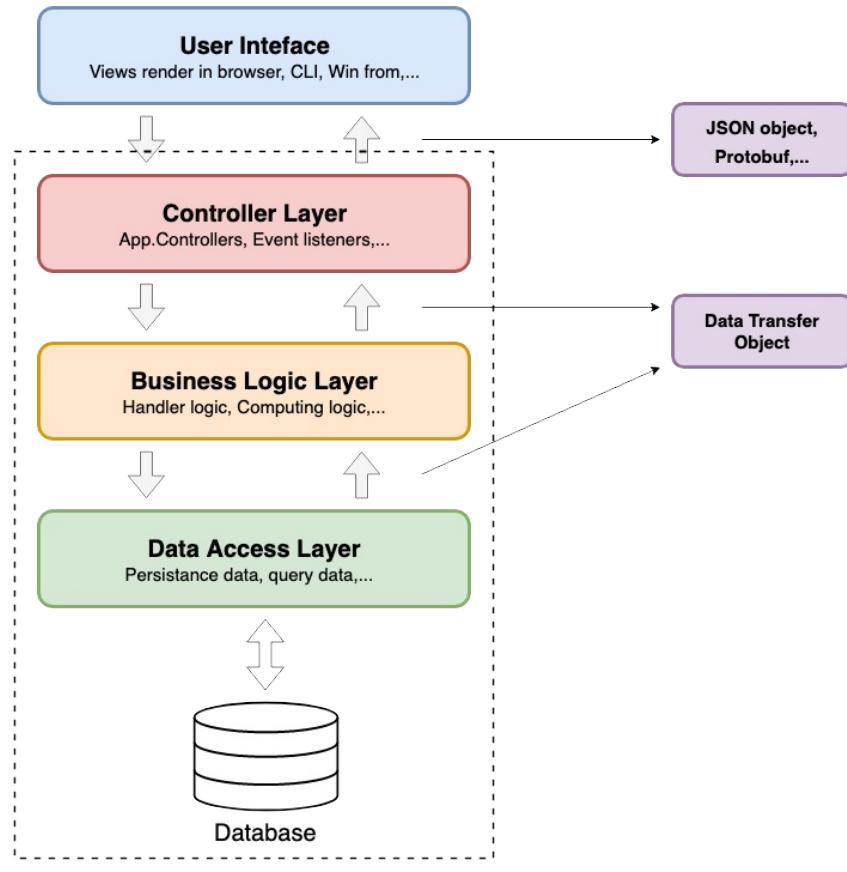
Kiến trúc một dự án web

Vue và **React** trong hình là hai frameworks front-end phổ biến trên thế giới, bởi vì chúng ta không tập trung nói về nó, do đó cấu trúc front-end của dự án không được nhấn mạnh trên lưu đồ. Thực tế trong vài dự án đơn giản, ngành công nghiệp không hoàn toàn tuân theo mô hình MVC, đặc biệt là phần M và C. Có nhiều công ty mà dự án của họ có rất nhiều phần logic bên trong lớp Controller, và chỉ quản lý phần lưu trữ dữ liệu ở lớp Model.

Tuy nhiên, theo như ý tưởng của người sáng lập MVC thì một business process cũng thuộc một loại "model". Nếu chúng ta đặt mã nguồn thao tác với dữ liệu và business process vào lớp M của MVC, thì lớp M sẽ quá cồng kềnh. Trong những dự án phức tạp, một lớp C hoặc M hiển nhiên là không đủ mà phải có nhiều lớp pure back-end API bên dưới nữa:

4.7.2 Mô hình 3 lớp

Giới thiệu mô hình 3 lớp



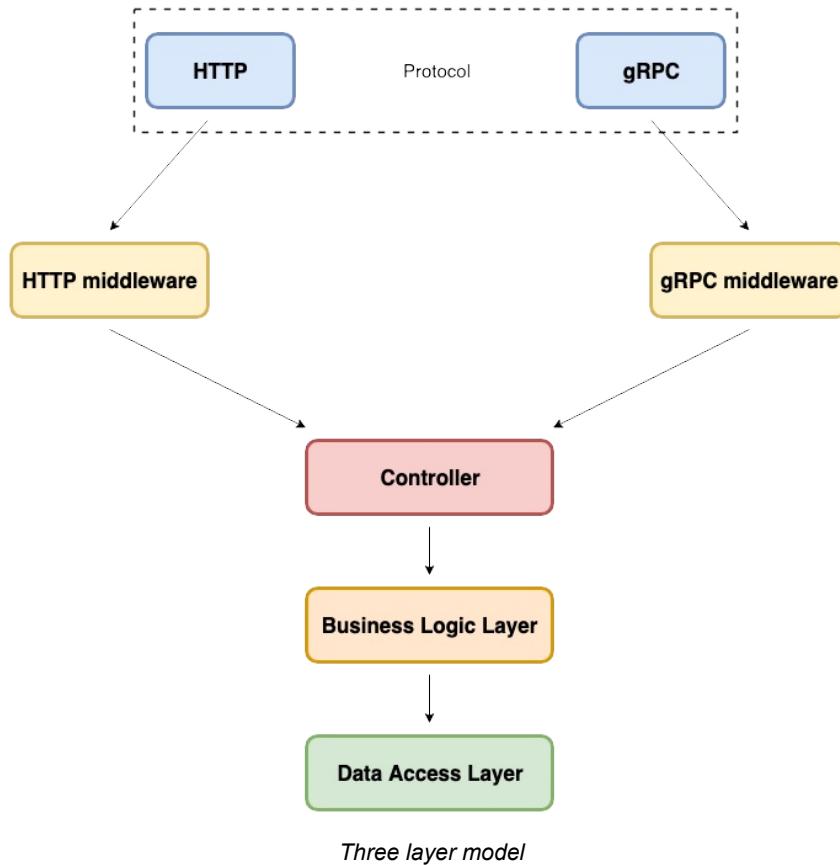
Three layer model

Mô hình 3 lớp là mô hình khá quen thuộc với chúng ta từ thời còn ngồi ghế nhà trường đại học. Khi chúng ta đã hiểu được tư tưởng của mô hình 3 lớp thì chúng ta có thể biến thể nó cho phù hợp với bài toán hay hệ thống của mình. Ở hình trên là một mô hình 3 lớp theo kinh nghiệm của mình hay làm:

- User Interface:** là phần giao diện tương tác với người dùng như web browser, winform, Đây là nơi người dùng gửi các yêu cầu RESTful hoặc yêu cầu gRPC tới hệ thống chúng ta.
- Controller Layer:** tương tự như ở trên, là một điểm đầu vào của service, là nơi nhận các gói tin yêu cầu và phản hồi về User Interface. Layer chịu trách nhiệm xử lý các logic routing, kiểm tra tham số, chuyển tiếp request, ...
- Business Logic Layer:** là lớp xử lý chính các business của hệ thống. Khi nhận các yêu cầu từ Controller layer, tùy vào loại yêu cầu sẽ có cách xử lý, tính toán khác nhau. Những yêu cầu cần đến dữ liệu hay thay đổi dữ liệu sẽ được lớp này đẩy xuống Data Access Layer tính toán.
- Data Access Layer:** là lớp duy nhất có thể truy vấn đến database của service, layer thực hiện các thao tác có liên quan đến dữ liệu như (select, insert, update, delete, ...).

Việc phân chia thành các lớp khác nhau giúp cho chúng ta dễ dàng phát triển và bảo trì hệ thống. Các lớp đảm nhận vai trò khác nhau, giảm sự phụ thuộc giữa các lớp khi hệ thống to ra. Đồng thời các lớp được phân tách giúp chúng ta có thể tái sử dụng lại khá nhiều, tiết kiệm được thời gian xây dựng. Như mình đã nói ở trên, tuy vào nghiệp vụ và hệ thống chúng ta sẽ lựa chọn cho mình một mô hình phù hợp, không nên cứng nhắc quá cho một mô hình. Một khi đã hiểu được ý tưởng thì chúng ta có thể linh hoạt trong việc thiết kế mô hình cho hệ thống.

Sử dụng thêm middleware



Bên cạnh các yêu cầu business logic, hệ thống web còn phải hiện thực phần non-business logic như xác thực token, gửi số liệu báo cáo, kiểm tra tín hợp lệ yêu cầu, ... để tách riêng hai phần này, các lớp middleware được thêm vào.

Liên kết

- Phần tiếp theo: [Lời nói thêm](#)
- Phần trước: [Mục lục](#)

4.8 Lời nói thêm

Kết thúc chương này hi vọng bạn đọc đã có một cái nhìn cụ thể về lập trình web cũng như nắm được một số framework hỗ trợ Golang trong lập trình web.

Bởi vì web là một lĩnh vực khá rộng, trong chương này chúng tôi chưa thể đề cập tới các vấn đề như lập trình front-end, back-end cụ thể như thế nào cũng như các vấn đề về xác thực người dùng. Bạn đọc có thể tham khảo thêm tại:

- [Building Web Apps with Go - codegansta](#)
- [Build web application with Golang - astaxie](#)

Liên kết

- Phần tiếp theo: [Chương 5: Hệ thống phân tán](#)
- Phần trước: [Mô hình của các dự án web](#)
- [Mục lục](#)

Chapter 5 Hệ thống phân tán



"We used it to write our own simple distributed computing software after realizing hadoop was too complicated (and thus bug prone) for our embarrassingly parallel needs. It took us less time to get the system written, stable and up and running then it had to get hadoop setup" – micro_cam in Hacker News

Ngôn ngữ Go ngày càng trở nên phổ biến và đang dần thay thế các ngôn ngữ truyền thống vì các ưu điểm vượt trội của nó. Bên cạnh đó, sự phát triển mạnh mẽ của điện toán đám mây như AWS, Azure,... đã mang lại nhiều lợi ích cho các doanh nghiệp. Và không thể không kể đến các hệ thống như Docker, Kubernetes được xây dựng bằng Go, nhờ chúng mà kỷ nguyên đám mây đã phát triển mạnh mẽ và nhanh chóng. Cùng với đó, nó lại kéo theo là các mô hình thiết kế hiện đại ra đời như serverless, microservices, ..., nơi mà phần cứng đã đạt giới hạn của nó, ta không thể tiếp tục mở rộng theo kiểu `vertical` mà thay vào đó phải tập trung theo `horizontal` hay chia nhỏ vấn đề lớn thành nhiều vấn đề nhỏ để giải quyết. Lúc này, ta sẽ gặp lại các câu hỏi quen thuộc trong hệ thống phân tán như:

- Làm sao tạo một ID duy nhất trong hệ thống phân tán?
- Làm sao tạo ra một `lock` phân tán trên nhiều hệ thống?
- Làm sao cân bằng tải trong hệ thống?
- Tính thống nhất khi nhiều hệ thống sử dụng chung cấu hình?
- Thu thập dữ liệu lớn?

Các vấn đề trên tuy không mới nhưng ở đây, chúng ta sẽ giải quyết chúng theo cách của Go.

Liên kết

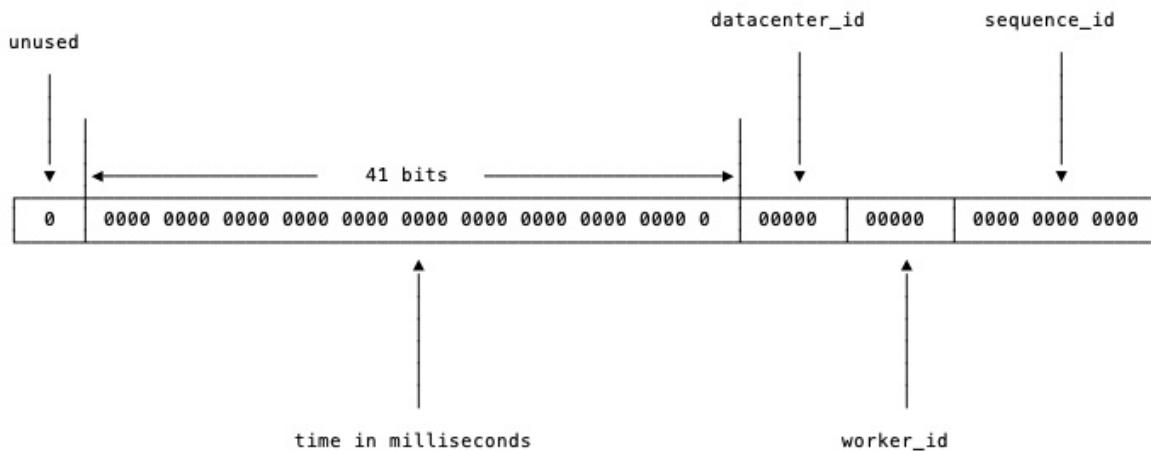
- Phần tiếp theo: [Distributed ID generator](#)
- Phần trước: [Chương 4: Lời nói thêm](#)
- [Mục lục](#)

5.1 Distributed ID generator

Đôi khi chúng ta cần tạo ra một ID tương tự như ID tăng tự động của MySQL và có tính chất không được trùng lặp. Chúng ta có thể sử dụng ID để hỗ trợ các ứng dụng trong kinh doanh. Điển hình, khi có chương trình khuyến mãi trong thương mại điện tử, một số lượng lớn đơn đặt hàng sẽ tràn vào hệ thống trong một khoảng thời gian ngắn, tạo ra khoảng 10 ngàn đơn mỗi giây, mỗi đơn sẽ tương ứng với một ID định danh.

Trước khi chèn vào cơ sở dữ liệu, chúng ta cần cung cấp cho các thông tin đó một ID, sau đó chèn nó vào cơ sở dữ liệu. Yêu cầu đối với ID này là có thông tin về thời gian. Nhờ vậy, cơ sở dữ liệu của chúng ta vẫn có thể sắp xếp các tin nhắn đó theo thứ tự thời gian.

Thuật toán [snowflake](#) của Twitter là giải pháp điển hình trong ngữ cảnh này. Hãy nhìn vào hình sau:



Phân phối Bit trong snowflake

Đầu tiên, ta xác định rằng giá trị là 64 bit, kiểu dữ liệu tương ứng bên Go là int64, chúng được chia thành bốn phần:

- Không sử dụng bit đầu tiên vì bit này là bit dấu.
- `timestamp` : sử dụng 41 bit để biểu thị timestamp khi nhận được yêu cầu, tính bằng milliseconds.
- `datacenter_id` : 5 chữ số để biểu thị id của trung tâm dữ liệu.
- `worker_id` : 5 chữ số để biểu thị id của server.
- `sequence_id` : cuối cùng là id tăng vòng lặp 12 bit (tăng từ 0 đến 111111111111 rồi trở về 0).

Cơ chế này có thể hỗ trợ chúng ta tạo ra $2^{12} = 4096$ tin nhắn trong cùng một millisecond trên cùng một server. Vậy chúng ta có tổng cộng có 4096 triệu tin nhắn trong một giây. Nó là hoàn toàn đủ để sử dụng trong các trường hợp cần một số lượng ID lớn trong thời gian ngắn.

`timestamp` gồm 41 chữ số nên có thể hỗ trợ chúng ta trong 69 năm. Tất nhiên, thời gian tính bằng mili giây của chúng ta có thể không thực sự bắt đầu từ năm 1970, vì nếu như vậy thì hệ thống sẽ chỉ hoạt động đến 2039/9/7 23:47:35, do đó, chúng chỉ nên tăng so với một mốc thời gian nhất định. Ví dụ: hệ thống của chúng ta là bắt đầu chạy vào 2018-08-01, thì chúng ta có thể coi mốc thời gian này là 2018-08-01 00: 00: 00.000 và tăng lên từ đó.

5.1.1 Phân công worker_id

`timestamp`, `datacenter_id`, `worker_id` và `sequence_id` là bốn trường, riêng `timestamp` và `sequence_id` được tạo bởi chương trình khi chạy. Còn `datacenter_id` và `worker_id` cần lấy trong giai đoạn triển khai và khi chương trình đã được khởi động, nó không thể thay đổi.

Nhìn chung, các server trong các trung tâm dữ liệu khác nhau sẽ cung cấp các API tương ứng để lấy id của trung tâm dữ liệu, vì vậy datacenter_id có thể dễ dàng lấy trong giai đoạn triển khai. Còn worker_id là một id mà chúng ta gán một cách "logically" cho từng máy. Chúng ta nên xử lý thế nào? Ý tưởng đơn giản là sử dụng các công cụ cung cấp tính năng ID tự tăng, chẳng hạn như MySQL:

```
mysql> insert into a (ip) values("10.1.2.101");
Query OK, 1 row affected (0.00 sec)
```

```
mysql> select last_insert_id();
+-----+
| last_insert_id() |
+-----+
|          2       |
+-----+
1 row in set (0.00 sec)
```

Khi bạn đã nhận được ID từ MySQL, worker_id sẽ được duy trì cục bộ, tránh được trường hợp làm mới mỗi khi bạn truy cập worker_id.

Tất nhiên, sử dụng MySQL đồng nghĩa với việc thêm một phụ thuộc bên ngoài vào service tạo ID. Chúng ta đều biết việc càng thêm nhiều phụ thuộc, khả năng phục vụ của service càng tệ.

Xem xét đến việc có một service tạo ID bị lỗi trong cluster, một phần của ID bị mất trong một khoảng thời gian, thì chúng ta cần gắn trực tiếp worker_id vào cấu hình của worker. Khi hệ thống đó hoạt động lại, đoạn script triển khai sẽ quy định giá trị worker_id.

5.1.2 Các Open source hiện có

5.1.2.1 Hiện thực theo snowflake chuẩn

[Snowflake](#) là một hiện thực Snowflake của Go khá nhẹ. Bạn có thể sử dụng phần định nghĩa của nó, xem dưới đây.

1 Bit Unused	41 Bit Timestamp	10 Bit NodeID	12 Bit Sequence ID
--------------	------------------	---------------	--------------------

Thư viện snowflake

Nó giống hoàn toàn như snowflake tiêu chuẩn. Và tương đối đơn giản để sử dụng như ví dụ sau:

main.go

```
package main

import (
    "fmt"
    "os"

    "github.com/bwmarrin/snowflake"
)

func main() {
    // Khởi tạo một node
    n, err := snowflake.NewNode(1)
    if err != nil {
        println(err)
        os.Exit(1)
    }
}
```

```

for i := 0; i < 3; i++ {
    // tạo ID
    id := n.Generate()
    fmt.Println("id", id)
    fmt.Println(
        "node: ", id.Node(),
        "step: ", id.Step(),
        "time: ", id.Time(),
        "\n",
    )
}
}

// output:
// id 1160090501362225152
// node: 1 step: 0 time: 1565422103621

// id 1160090501362225153
// node: 1 step: 1 time: 1565422103621

// id 1160090501362225154
// node: 1 step: 2 time: 1565422103621

```

Dĩ nhiên, thư viện này cũng cho phép chúng ta tùy chỉnh vài thông số, các trường có thể tùy chỉnh như:

```

// Epoch là thời gian bắt đầu
// Epoch được thiết lập theo twitter snowflake epoch vào thời điểm Nov 04 2010 01:42:54 UTC
// bạn có thể thiết lập Epoch theo thời gian của ứng dụng của bạn.
Epoch int64 = 1288834974657

// độ dài bit của máy chủ hoạt động
// nhớ rằng, bạn có tổng 22 bits chia sẻ giữa Node/Step
NodeBits uint8 = 10

// số bit để sử dụng cho Step
StepBits uint8 = 12

```

5.1.2.2 Sonyflake

[Sonyflake](#) là một dự án Open source của Sony. Ý tưởng cơ bản tương tự như snowflake, nhưng phân bổ bit hơi khác.

1 Bit Unused	39 Bit Timestamp	8 Bit Sequence ID	16 Bit Machine ID
--------------	------------------	-------------------	-------------------

Sonyflake

Thời gian ở đây chỉ sử dụng 39 bit, nhưng đơn vị thời gian trở thành 10ms. Về mặt lý thuyết, nó dài hơn thời gian của snowflake chuẩn đến 41 bit (174 năm).

Sequence ID giống với định nghĩa trước đó, Machine ID thực sự là id của Node. Sự khác biệt giữa [sonyflake](#) là các tham số cấu hình trong quá trình khởi động:

Cấu trúc của Settings như sau:

```

type Settings struct {
    // tương tự như Epoch
    // mặc định bắt đầu vào ngày 2014-09-01 00:00:00 +0000 UTC
    StartTime time.Time
    // hàm do người dùng định nghĩa
    // mặc định là 16bit cuối của IP gốc
    MachineID func() (uint16, error)
    // hàm kiểm tra MachineID có trùng hay không
}

```

```
    CheckMachineID func(uint16) bool
}
```

Sử dụng `CheckMachineID` là thiết kế khá thông minh. Nếu có một bộ lưu trữ tập trung và hỗ trợ kiểm tra sự trùng lặp, thì chúng ta có thể tùy chỉnh logic để kiểm tra xem `MachineID` có trùng không. Nếu công ty có cụm Redis được tạo sẵn, thì chúng dễ dàng kiểm tra trùng bằng các loại collection của Redis.

```
Redis 127.0.0.1:6379> SADD base64_encoding_of_last16bits MzI0Mgo=
(integer) 1
Redis 127.0.0.1:6379> SADD base64_encoding_of_last16bits MzI0Mgo=
(integer) 0
```

main.go

```
package main

import (
    "fmt"
    "os"
    "time"

    "github.com/sony/sonyflake"
)

// getMachineID hàm lấy MachineID
func getMachineID() (uint16, error) {
    var machineID uint16
    var err error

    // đọc MachineID từ file ở local
    machineID = readMachineIDFromLocalFile()
    if machineID == 0 {
        // nếu không có
        // gọi hàm generateMachineID để tạo ID
        machineID, err = generateMachineID()
        If err != nil {
            return 0, err
        }
    }

    return machineID, nil
}

// checkMachineID hàm kiểm tra MachineID có trùng không
func checkMachineID(machineID uint16) bool {
    // thêm machineID vào redis
    saddResult, err := saddMachineIDToRedisSet()
    if err != nil || saddResult == 0 {
        return true
    }

    // lưu lại machineID xuống file ở local
    err := saveMachineIDToLocalFile(machineID)
    if err != nil {
        return true
    }

    return false
}

func main() {
    t, _ := time.Parse("2006-01-02", "2018-01-01")

    // khởi tạo struct setting
    settings := sonyflake.Settings{
```

```
StartTime: t,
MachineID: getMachineID,
CheckMachineID: checkMachineID,
}

// tạo sonyflake struct
sf := sonyflake.NewSonyflake(settings)

// lấy ID
id, err := sf.NextID()
if err != nil {
    fmt.Println(err)
    os.Exit(1)
}

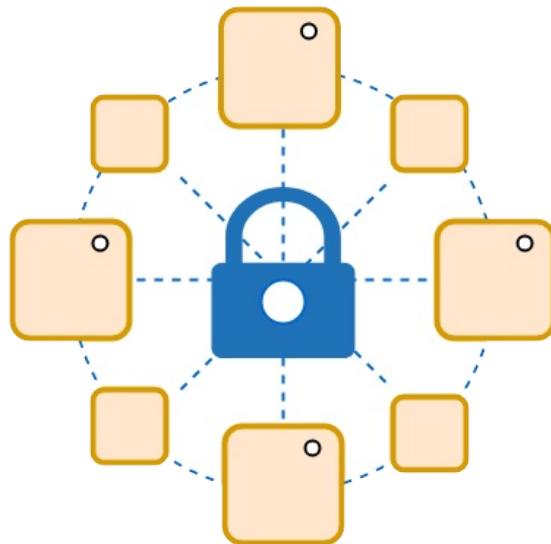
fmt.Println("ID: ", id)
}

// output:
// ID: 84989976554504193
```

Liên kết

- Phàn tiếp theo: [Lock phân tán \(Distributed lock\)](#)
- Phàn trước: [Chương 5: Distributed System](#)
- [Mục lục](#)

5.2 Lock phân tán



Distributed lock

Trước khi đi vào nội dung chính là `Distributed lock` chúng ta cùng xem xét bài toán sau. Khi một chương trình chạy đồng thời hoặc song song sửa đổi biến toàn cục, hành vi sửa đổi này cần phải được `lock` để tránh trường hợp `race conditions`. Tại sao bạn cần phải lock ? Hãy xem điều gì xảy ra khi trong bài toán đếm số một cách đồng thời mà không lock dưới đây.

main.go

```
package main

import (
    "sync"
    "fmt"
)

// biến đếm toàn cục
var counter int

func main() {
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            // tăng biến counter lên một đơn vị
            counter++
        }()
    }

    wg.Wait()
    fmt.Println("Counter: ", counter)
}
```

Khi ta chạy nhiều lần, các kết quả sẽ khác nhau:

```
$ go run local_lock.go
Counter: 945
$ go run local_lock.go
```

```
Counter 937
$ go run local_lock.go
Counter: 959
```

5.2.1 Lock quá trình đang thực hiện

Để có kết quả chính xác, lock phần code thực thi của bộ đếm như ví dụ dưới đây.

```
// ... bỏ qua phần trước
var wg sync.WaitGroup
var l sync.Mutex
for i := 0; i < 1000; i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        // lấy lock
        l.Lock()
        counter++
        // trả lock
        l.Unlock()
    }()
}
wg.Wait()
fmt.Println("Counter: ", counter)
```

Các lần chạy đều cho ra cùng một kết quả:

```
$ go run local_lock.go
1000
```

5.2.2 Sử dụng Trylock

Trong một số tình huống, chúng ta chỉ muốn một tiến trình thực thi một nhiệm vụ. Ở ví dụ đếm số ở trên, tất cả goroutines đều thực hiện thành công. Giả sử có goroutine thất bại trong khi thực hiện, chúng ta cần phải bỏ qua tiến trình của nó. Đây là lúc cần `trylock`.

Trylock, như tên của nó, cố gắng lock và nếu lock thành công thì thực hiện các công việc tiếp theo. Nếu lock bị lỗi, nó sẽ không bị chặn lại mà sẽ trả về kết quả lock. Trong lập trình Go, chúng ta có thể mô phỏng một trylock với channel có kích thước buffer là 1.

main.go

```
package main

import (
    "sync"
)

// Lock try lock
type Lock struct {
    c chan struct{}
}

// tạo một lock
func NewLock() Lock {
    var l Lock
    l.c = make(chan struct{}, 1)
    l.c <- struct{}{}
    return l
}
```

```

    return 1
}

// Lock try lock, trả về kết quả lock là true/false
func (l Lock) Lock() bool {
    lockResult := false
    select {
    case <-l.c:
        lockResult = true
    default:
    }
    return lockResult
}

// Unlock , giải phóng lock
func (l Lock) Unlock() {
    l.c <- struct{}{}
}

// biến đếm toàn cục
var counter int

func main() {
    var l = NewLock()
    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            if !l.Lock() {
                // log error
                fmt.Println("lock failed")
                return
            }
            counter++
            fmt.Println("current counter", counter)
            l.Unlock()
        }()
    }
    wg.Wait()
}

// output
// lock failed
// current counter 1
// lock failed
// lock failed
// lock failed

```

Bởi vì logic của chúng ta giới hạn bởi mỗi goroutine chỉ thực hiện logic sau khi nó Lock thành công. Còn đối với Unlock, nó đảm bảo rằng channel của Lock ở đoạn code trên phải trống, nên nó sẽ không bị chặn hoặc thất bại giữa chúng. Đoạn code trên sử dụng channel có kích thước 1 để mô phỏng một tryLock. Về lý thuyết, bạn có thể sử dụng [CAS](#) trong thư viện chuẩn để đạt được chức năng tương tự với chi phí thấp hơn. Bạn có thể thử dùng nó.

Trong một hệ thống đơn, trylock không phải là một lựa chọn tốt. Bởi vì khi có một lượng lớn lock goroutine có thể gây lãng phí tài nguyên trong CPU một cách vô nghĩa. Có một danh từ thích hợp được sử dụng để mô tả kịch bản khóa này: `livelock`.

`livelock` nghĩa là chương trình trông có vẻ đang thực thi bình thường, nhưng trên thực tế, CPU bị lãng phí khi phải lặp lại lock thay vì thực thi tác vụ, do đó việc thực thi của chương trình không hiệu quả. Vấn đề của livelock sẽ gây ra rất nhiều hậu quả xấu. Do đó, trong ngữ cảnh máy đơn, không nên sử dụng loại lock này.

5.2.3 Redis dựa trên setnx

Trong ngữ cảnh phân tán, chúng ta cũng cần một loại logic "ưu tiên". Làm sao để có được nó? Chúng ta có thể sử dụng lệnh `setnx` do Redis cung cấp:

main.go

```
package main

import (
    "fmt"
    "sync"
    "time"

    "github.com/go-redis/redis"
)

func incr() {
    client := redis.NewClient(&redis.Options{
        Addr:     "localhost:6379",
        // không cần khởi tạo password
        Password: "",
        // không dùng database
        DB:       0,
    })

    var lockKey = "counter_lock"
    var counterKey = "counter"

    // lấy lock
    resp := client.SetNX(lockKey, 1, time.Second*5)
    lockSuccess, err := resp.Result()

    if err != nil || !lockSuccess {
        fmt.Println(err, "lock result: ", lockSuccess)
        return
    }

    // tăng cntValue lên một đơn vị
    getResp := client.Get(counterKey)
    cntValue, err := getResp.Int64()
    if err == nil || err == redis.Nil {
        cntValue++
        resp := client.Set(counterKey, cntValue, 0)
        _, err := resp.Result()
        if err != nil {
            // log err
            fmt.Println("set value error!")
        }
    }
    fmt.Println("current counter is ", cntValue)

    delResp := client.Del(lockKey)
    // giải phóng lock
    unlockSuccess, err := delResp.Result()
    if err == nil && unlockSuccess > 0 {
        fmt.Println("unlock success!")
    } else {
        fmt.Println("unlock failed", err)
    }
}
```

```
func main() {
    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            incr()
        }()
    }
    wg.Wait()
}
```

Nhìn vào kết quả khi chạy:

```
$ go run redis_setnx.go
<nill> lock result: false
Current counter is 2028
Unlock success!
```

Thông qua mã nguồn và kết quả chạy thực tế, chúng ta có thể thấy rằng khi gọi `setnx` thực sự rất giống với một trylock. Nếu khóa bị lỗi, logic của tác vụ liên quan sẽ không được thực thi.

`setnx` tuyệt vời cho các ngữ cảnh cần tính đồng thời cao và được sử dụng để giành một số tài nguyên "độc nhất". Ví dụ: trong hệ thống kiểm tra giao dịch, người bán tạo một đơn đặt hàng và nhiều người mua sẽ giành lấy nó. Trong ví dụ này, chúng ta không có cách nào dựa vào thời gian cụ thể để phán đoán thứ tự. Bởi vì, dù đó là thời gian của thiết bị người dùng hay thời gian của mỗi server trong hệ thống phân tán, không có cách nào để đảm bảo thời gian chính xác sau khi tổng hợp lại. Ngay cả khi chúng ta đã đặt các service chung một server thì vẫn có sự khác biệt nhỏ.

Do đó, chúng ta cần dựa vào thứ tự của các yêu cầu này để node Redis thực hiện thao tác khóa chính xác. Nếu môi trường mạng của người dùng tương đối kém, thì họ chỉ cần tạo thêm yêu cầu. Các bạn có thể xem chi tiết phần `Distributed lock` bằng Redis [ở đây](#).

5.2.4 Sử dụng ZooKeeper

Chúng ta cùng xem qua một ví dụ về dùng lock trong `ZooKeeper`.

main.go

```
package main

import (
    "fmt"
    "time"

    "github.com/samuel/go-zookeeper/zk"
)

func main() {
    c, _, err := zk.Connect([]string{"127.0.0.1"}, time.Second) /*10)
    if err != nil {
        panic(err)
    }
```

```

// tạo lock
l := zk.NewLock(c, "/lock", zk.WorldACL(zk.PermsAll))
// sử dụng lock
err = l.Lock()
if err != nil {
    panic(err)
}
fmt.Println("lock succ, do your business logic")

time.Sleep(time.Second * 10)

// ...xử lý logic của bạn

// giải phóng lock
l.Unlock()
fmt.Println("unlock succ, finish business logic")
}

```

Lock dựa trên ZooKeeper khác với lock dựa trên Redis ở chỗ nó sẽ chặn cho đến khi lấy lock thành công, tương tự như `mutex.Lock`.

Nguyên tắc này cũng dựa trên node (một server trong hệ thống phân tán) thứ tự tạm thời và quan sát API. Ví dụ, chúng ta sử dụng node `/lock`. Các Lock sẽ chèn giá trị của chính nó vào danh sách node bên dưới node này. Khi các node con ở dưới node này thay đổi, nó sẽ thông báo cho tất cả các chương trình quan sát giá trị của node. Lúc này, chương trình sẽ kiểm tra xem ID của node con gần node hiện tại nhất có giống với giá trị của chính nó không. Nếu chúng giống nhau thì việc lock diễn ra thành công.

Loại lock phân tán này phù hợp hơn cho các ngữ cảnh định thời tác vụ phân tán, nhưng nó không phù hợp trong các ngữ cảnh thường xuyên cần lock trong thời gian lâu. Theo bài báo [Chubby](#) của Google, các lock dựa trên các giao thức nhất quán cao áp dụng cho loại khóa "coarse-grained". Loại "coarse-grained" có nghĩa là khóa mất nhiều thời gian. Chúng ta nên xem xét liệu lock này có phù hợp để sử dụng với mục đích của chúng ta hay không.

5.2.5 Sử dụng etcd

[Etcd](#) là một thư viện thường được dùng trong các hệ thống phân tán, nó có chức năng gần giống với ZooKeeper và đã trở nên "hot" hơn trong hai năm qua. Dựa trên ZooKeeper, chúng tôi đã triển khai distributed lock. Với etcd, chúng ta cũng có thể thực hiện các chức năng tương tự:

main.go

```

package main

import (
    "log"

    "github.com/zieckey/etcdsync"
)

func main() {
    // khởi tạo lock
    m, err := etcdsync.New("/lock", 10, []string{"http://127.0.0.1:2379"})
    if m == nil || err != nil {
        log.Printf("etcdsync.New failed")
        return
    }
    // lock
    err = m.Lock()
    if err != nil {
        log.Printf("etcdsync.Lock failed")
        return
    }
}

```

```

    }

    log.Printf("etcdsync.Lock OK")
    log.Printf("Get the lock. Do something here.")

    // giải phóng lock
    err = m.Unlock()
    if err != nil {
        log.Printf("etcdsync.Unlock failed")
    } else {
        log.Printf("etcdsync.Unlock OK")
    }
}

```

Không có node thứ tự như ZooKeeper trong etcd. Vì vậy, việc thực hiện lock của nó khác với ZooKeeper. Quá trình lock cho etcdsync của đoạn code mẫu ở trên cụ thể như sau:

1. Kiểm tra xem có giá trị nào trong đường dẫn `/lock` không. Nếu có một giá trị, khóa đã bị người khác lấy.
2. Nếu không có giá trị, nó sẽ ghi giá trị của chính nó vào. Khi ghi giá trị thành công thì lock đã thành công. Giả sử có một node đang ghi thì node khác đến ghi, điều này khiến khóa bị lỗi. Tiếp bước 3.
3. Kiểm tra sự kiện trong `/lock`, cái mà đang bị kẹt tại lúc này.
4. Khi một sự kiện xảy ra trong đường dẫn `/lock`, process hiện tại được đánh thức. Kiểm tra xem sự kiện xảy ra có phải là sự kiện xóa không (chứng tỏ rằng lock đang được mở) hoặc sự kiện đã hết hạn (cho biết khóa đã hết hạn). Nếu vậy, quay lại bước 1 và thực hiện tuần tự các bước.

Điều đáng nói là trong [etcdv3 API](#) đã chính thức cung cấp API lock mà có thể sử dụng trực tiếp. Các bạn có thể tham khảo tài liệu etcdv3 để nghiên cứu thêm.

5.2.7 Làm sao để chọn đúng loại lock

Nếu bạn phát triển một dịch vụ phân tán, nhưng quy mô dịch vụ kinh doanh không lớn, thì sử dụng lock nào cũng như nhau. Nếu bạn có một cụm ZooKeeper, etcd hoặc Redis có sẵn trong công ty, hãy sử dụng cái có sẵn để đáp ứng nhu cầu kinh doanh của bạn mà không cần các công nghệ mới.

Nếu doanh nghiệp phát triển đến một mức độ nhất định, thì chúng ta cần xem xét ở nhiều khía cạnh. Đầu tiên là hãy xem xét liệu lock của bạn có cho phép mất dữ liệu trong bất kỳ điều kiện nào không. Nếu không, thì đừng sử dụng khóa `setnx` của Redis.

Nếu độ tin cậy của dữ liệu lock là rất cao, thì chỉ có khóa etcd hoặc ZooKeeper đảm bảo độ tin cậy của dữ liệu. Nhưng mặt trái của sự đáng tin cậy là throughput thấp hơn và latency cao hơn. Cần phải có những bài kiểm tra kỹ lưỡng theo từng cấp độ kinh doanh để đảm bảo rằng các lock phân tán bằng cụm etcd hoặc ZooKeeper có thể chịu được áp lực của các yêu cầu kinh doanh thực tế.

Lưu ý: sẽ không có cách nào để cải thiện hiệu suất của cụm etcd và Zookeeper bằng cách thêm các node. Để mở rộng quy mô theo chiều ngang, bạn chỉ có thể tăng số lượng cụm để hỗ trợ nhiều yêu cầu hơn. Điều này sẽ tăng thêm các chi phí vận hành, bảo trì và giám sát. Nhiều cụm có thể cần phải thêm proxy. Nếu không có proxy, dịch vụ cần được phân phối theo một ID nhất định. Nếu dịch vụ đã được mở rộng, bạn cũng nên xem xét việc di chuyển dữ liệu động. Đây không phải là điều dễ dàng.

Liên kết

- Phần tiếp theo: [Hệ thống tác vụ có trì hoãn](#)
- Phần trước: [Distributed ID generator](#)
- [Mục lục](#)

5.3 Hệ thống tác vụ có trì hoãn

Khi chúng ta xây dựng hệ thống, chúng ta thường xử lý các công việc trên thời gian thực. Người dùng gửi yêu cầu và sau đó được phản hồi ngay lập tức. Nhưng đôi khi bạn sẽ gặp các công việc không theo thời gian thực, chẳng hạn như đưa ra các thông báo quan trọng tại một thời điểm cụ thể. Hoặc bạn cần làm một cái gì đó cụ thể sau khi người dùng đã thực hiện một vài thứ trong X phút / Y giờ, chẳng hạn như thông báo, phát hành trái phiếu, ...

Nếu quy mô business còn nhỏ, chúng ta có thể sử dụng cơ sở dữ liệu để xử lý các công việc loại này, nhưng các công ty có quy mô lớn hơn sẽ tìm ra các giải pháp linh hoạt hơn để giải quyết vấn đề này.

Nhìn chung, có hai cách để giải quyết vấn đề trên:

1. Hiện thực một hệ thống phân tán để quản lý tác vụ theo thời gian tương tự như [crontab](#).
2. Hiện thực một hàng đợi tin nhắn (message queue) mà hỗ trợ các tin nhắn được định thời trước.

Hai ý tưởng trên đã tạo ra nhiều hệ thống khác nhau, nhưng bản chất là giống nhau. Ta cần hiện thực một bộ đếm thời gian. Bộ đếm thời gian không phải là hiếm trong ngữ cảnh có một server. Ví dụ, chúng ta thường gọi hàm `SetReadDeadline()` khi làm việc với thư viện mạng. Điều này thực sự tạo ra một bộ đếm thời gian (timer) cục bộ. Sau khi hết thời gian quy định, chúng ta sẽ nhận một thông báo từ bộ đếm thời gian nói rằng thời gian đã hết. Tại thời điểm này, nếu việc đọc chưa hoàn thành, thì coi như đã xảy ra sự cố mạng, do đó chúng ta có thể ngừng lại việc đọc.

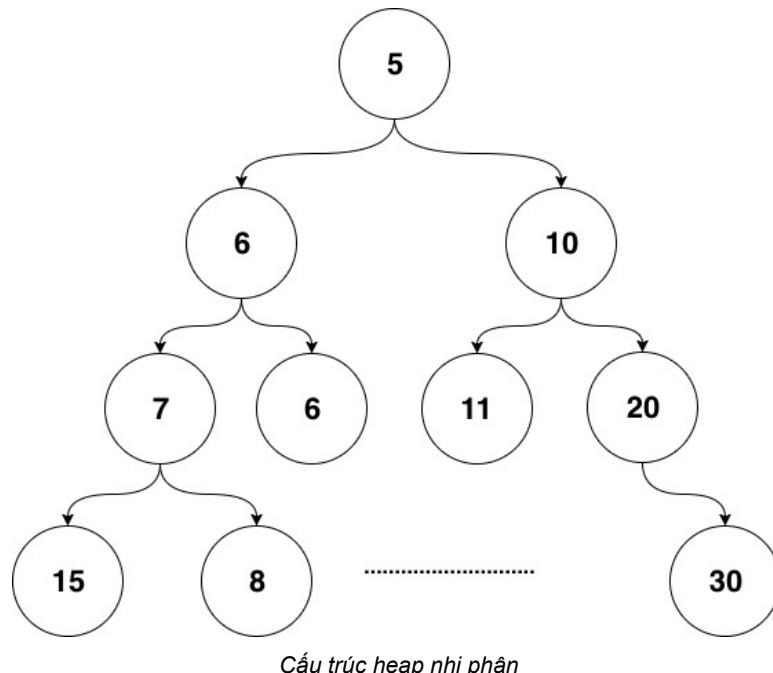
Hãy bắt đầu với bộ đếm thời gian và khám phá việc hiện thực hệ thống tác vụ có trì hoãn.

5.3.1 Hiện thực bộ đếm thời gian

Việc hiện thực các bộ đếm thời gian đã là một vấn đề quen thuộc. Phổ biến là time heap và time wheel.

5.3.1.1 Time heap

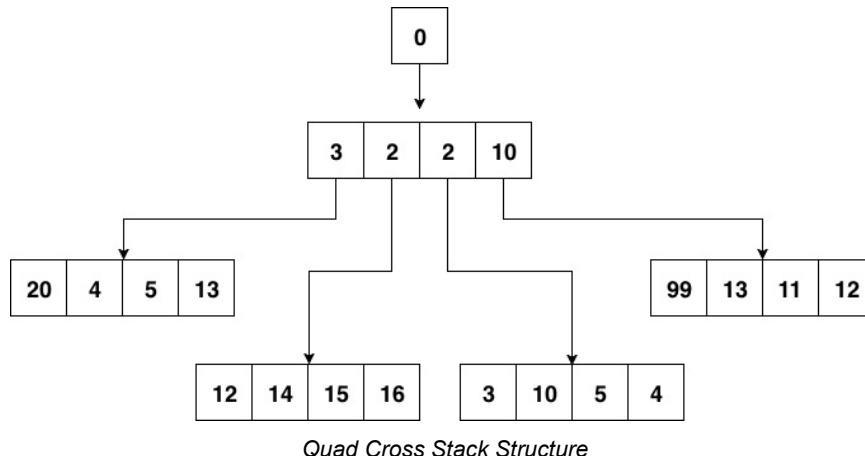
Time heap là phổ biến nhất và thường được hiện thực bằng min heap. Min heap là một cây nhị phân đặc biệt.



Những lợi ích của min heap là gì? Trong thực tế, đối với bộ đếm thời gian, nếu phần tử trên cùng lớn hơn thời gian hiện tại, thì tất cả các phần tử trong heap đều lớn hơn thời gian hiện tại. Hơn nữa, chúng ta không cần quan tâm gì về time heap. Độ phức tạp thời gian của việc kiểm tra này là $O(1)$.

Khi ta thấy các phần tử đầu của heap nhỏ hơn thời điểm hiện tại, thì có thể một loạt các sự kiện đã bắt đầu hết hạn, thì các lệnh pop-up và điều chỉnh heap diễn ra. Độ phức tạp thời gian của mỗi lần điều chỉnh heap là $O(\lg N)$.

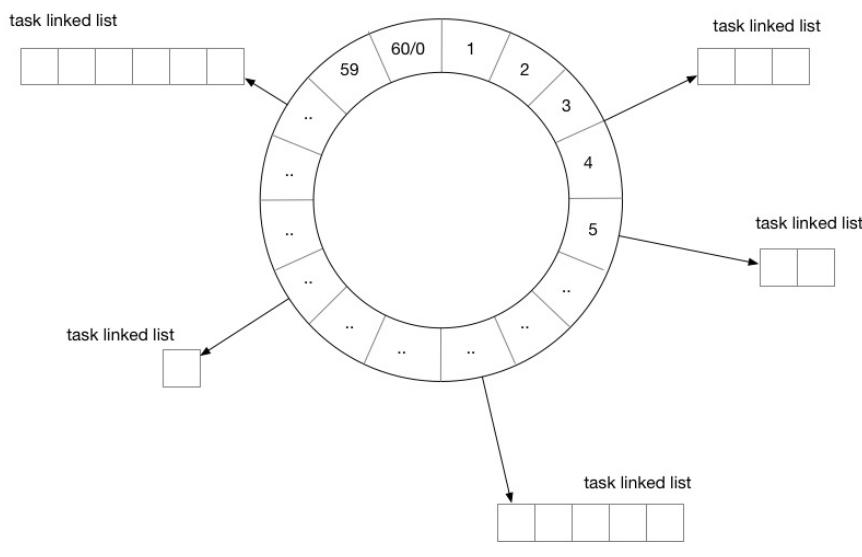
Bộ đếm thời gian tích hợp sẵn của Go được hiện thực với một time heap, nhưng thay vì sử dụng một heap nhị phân, có một giải pháp tốt hơn được sử dụng. Hãy nhìn vào min heap với bốn cạnh trông như thế nào:



Bản chất của min heap, node cha nhỏ hơn bốn node con của nó, không có mối quan hệ kích thước đặc biệt giữa các node con.

Không có sự khác biệt giữa thời gian quá hạn của phần tử và điều chỉnh heap trong `heap bốn node` và `heap nhị phân`.

5.3.1.2 Time Wheel



Khi sử dụng time wheel để hiện thực bộ đếm thời gian, chúng ta cần xác định $\frac{t}{\Delta t}$ của mỗi ô. Bánh xe thời gian có thể được tưởng tượng như một chiếc đồng hồ và trung tâm có kim giây theo chiều kim đồng hồ. Mỗi lần chúng ta chuyển sang một ô, chúng ta cần xem danh sách nhiệm vụ được gắn trên ô đó có nhiệm vụ đã đến hạn hay không.

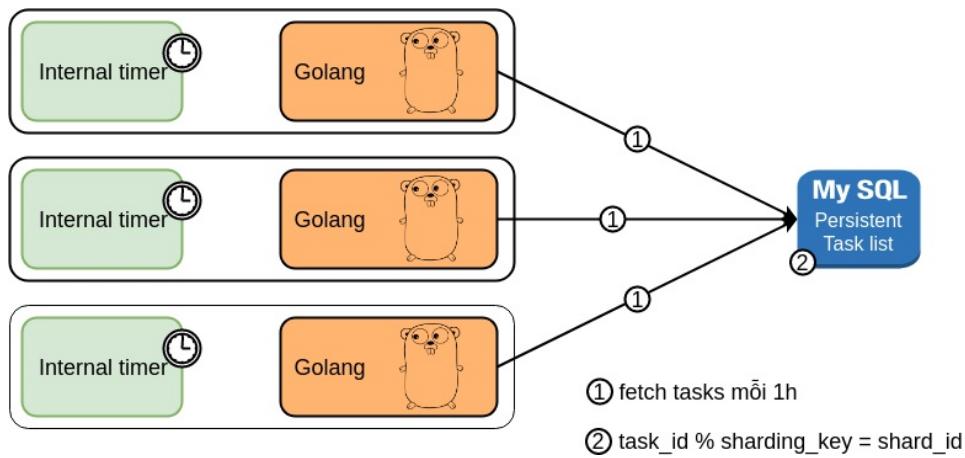
Về mặt cấu trúc, time wheel tương tự như bảng băm, nếu chúng ta định nghĩa thuật toán băm là: **thời gian kích hoạt % số phần tử của time wheel**. Thì đây là một bảng băm đơn giản. Trong trường hợp xung đột băm, một danh sách liên kết được sử dụng.

Ngoài time wheel một lớp, có một số time wheel trong thực tế sử dụng nhiều lớp. Tuy nhiên, tôi sẽ không đi vào chi tiết ở đây.

5.3.2 Phân phối công việc

Thông qua cách hiện thực bộ đếm thời gian cơ bản, nếu chúng ta đang phát triển một hệ thống trên một server, chúng ta có thể sử dụng chúng. Nhưng trong chương này chúng ta đang nói về ngữ cảnh hệ thống phân tán, vẫn còn một khoảng cách nhỏ để có thể áp dụng vào hệ thống phân tán.

Chúng ta cần phân bổ các công việc theo "thời gian" hoặc "tri hoãn" công việc (về cơ bản cũng là thời gian). Ý tưởng ở đây là:



Khi các tác vụ thời gian này được kích hoạt, bạn cần thông báo cho phía người dùng. Có hai cách để làm điều này:

1. Đóng gói các thông tin được kích hoạt bởi tác vụ dưới dạng tin nhắn và gửi nó vào một hàng đợi. Phía người dùng chỉ cần lắng nghe hàng đợi tin nhắn này.
2. Gọi một hàm callback do người dùng định cấu hình.

Cả hai cách này đều có ưu điểm và nhược điểm riêng. Nếu bạn sử dụng cách 1, thì khi hàng đợi tin nhắn bị lỗi, toàn bộ hệ thống sẽ `unavailable`. Tất nhiên, hàng đợi tin nhắn thường sẽ có giải pháp đảm bảo `high-availability`. Phần lớn thời gian, chúng ta không phải lo lắng về vấn đề này.

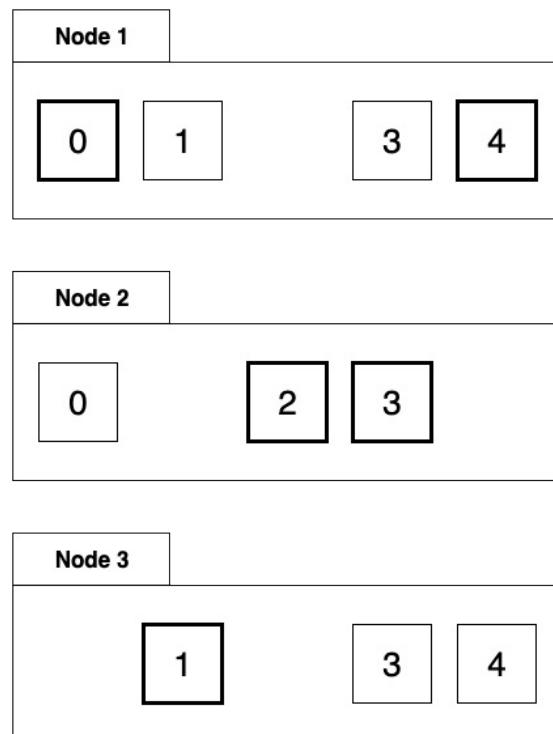
Thứ hai, nếu hàng đợi tin nhắn được sử dụng ở giữa quá trình của nghiệp vụ business, độ trễ của cho một xử lý business sẽ tăng lên. Nếu tác vụ được tính thời gian phải được hoàn thành trong vòng hàng chục millisecond đến vài trăm millisecond sau khi kích hoạt, thì hàng đợi tin nhắn sẽ có những rủi ro nhất định.

Nếu bạn áp dụng cách thứ hai, nó sẽ tăng gánh nặng của hệ thống tác vụ thời gian. Chúng tôi biết rằng việc thực thi đúng sơ nhất của bộ đếm thời gian trên một server là chức năng callback mất quá nhiều thời gian để thực thi, điều này sẽ ảnh hưởng đến việc thực thi tác vụ tiếp theo. Trong một kịch bản phân tán, mỗi quan tâm này vẫn được áp dụng. Một callback không tốt có thể trực tiếp kéo toàn bộ hệ thống nhiệm vụ theo thời gian đi xuống. Bên cạnh, chúng ta cũng cần xem xét việc thêm cài đặt thời gian `timeout` cho callback và xem xét cẩn thận khoảng thời gian chờ mà người dùng cấu hình.

5.3.3 Cân bằng dữ liệu và cân nhắc tính bất biến

Khi tác vụ của chúng ta thực hiện lỗi do một máy nào đó trong cụm có vấn đề, tác vụ cần được thực hiện lại. Theo chiến lược modulo phía trên, việc phân phối lại các tác vụ chưa được xử lý bởi server này sẽ rắc rối hơn. Nếu đó là một hệ thống đang thực sự chạy, bạn phải chú ý nhiều hơn đến việc cân bằng các nhiệm vụ trong trường hợp có lỗi xảy ra.

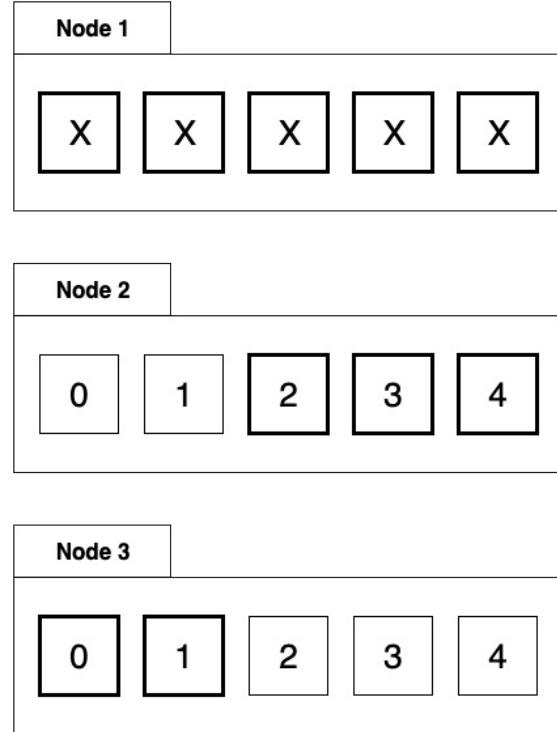
Chúng ta có thể tham khảo thiết kế phân phối dữ liệu của [ElasticSearch](#), mỗi dữ liệu của tác vụ có nhiều bản sao. Giả sử có hai bản sao như hình sau:



Task Data Distribution

Mặc dù có hai node cùng sở hữu một dữ liệu, dữ liệu sẽ có sự phân biệt: **bản chính** hay **bản phụ**. Bản chính là ô vuông có tô đậm viền trong hình và bản phụ có viền bình thường. Một tác vụ sẽ chỉ được thực hiện trên node có bản chính.

Khi có node bị lỗi, ta cần phân phối các dữ liệu của tác vụ trên node này. Ví dụ, node 1 bị treo, chúng ta cùng xem hình sau.

*Data distribution at fault*

Dữ liệu của node 1 sẽ được di chuyển đến node 2 và node 3.

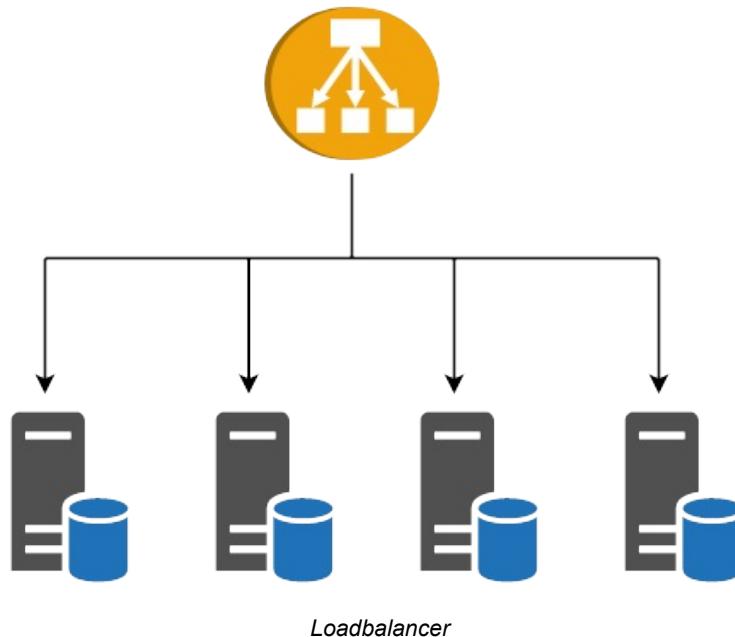
Tất nhiên, bạn cũng có thể sử dụng một ý tưởng phức tạp hơn một chút, chẳng hạn như phân chia vai trò của các node trong cụm và node điều phối sẽ phân phối lại các tác vụ trong trường hợp có lỗi. Xem xét tính *high availability*, node điều phối cũng cần 1 đến 2 Node dự phòng để ngăn ngừa tai nạn.

Như đã đề cập, chúng ta sẽ sử dụng hàng đợi tin nhắn để thông báo cho người dùng. Khi sử dụng hàng đợi tin nhắn, nhiều hàng đợi không hỗ trợ *exactly once*. Trong trường hợp này, chúng ta cần để người dùng tự xử lý việc nhận tin nhắn 2 lần hoặc thiếu tin nhắn (xử lý tạm thời).

Liên kết

- Phản tiếp theo: [Cân bằng tải](#)
- Phản trước: [Lock phân tán](#)
- [Mục lục](#)

5.4 Cân bằng tải



Phần này sẽ thảo luận về các phương pháp phổ biến trong cân bằng tải hệ thống phân tán.

5.4.1 Ý tưởng cân bằng tải

Cân bằng tải luôn là một vấn đề đáng chú trọng khi xây dựng một hệ thống phân tán. Có rất nhiều mô hình để đặt cân bằng tải: ở phía client, ở phía gateway, ở sidecar,... Mỗi mô hình đều có ưu nhược điểm riêng, tùy vào nhu cầu và điều kiện hiện tại của bạn để chọn một cách phù hợp nhất. Nhưng, dù nó đặt ở đâu, vấn đề của cân bằng tải vẫn luôn là giải thuật cân bằng tải như thế nào để mang lại tính cân bằng nhất cho các hệ thống con. Một giải thuật tốt sẽ đem lại một hiệu năng tốt cho toàn hệ thống, giảm khả năng `bottleneck` khi lượng truy cập lớn chỉ dồn vào service. Trong chương này, ta sẽ tìm hiểu về cách cân bằng tải và các lưu ý khi sử dụng các thuật toán cân bằng tải.

Khi có N node cùng cung cấp service và chúng ta cần chọn một trong số đó để thực hiện quy trình business. Có một số ý tưởng:

- Chọn theo thứ tự: lần gần nhất bạn chọn cái đầu tiên, thì lần này bạn chọn cái thứ hai, rồi cứ thế với cái tiếp theo. Nếu bạn đã đạt đến cái cuối cùng, thì cái tiếp theo bắt đầu từ cái đầu tiên. Trong trường hợp này, chúng ta có thể lưu trữ thông tin node dịch vụ trong một mảng. Sau khi mỗi yêu cầu được hoàn thành xuôi dòng, chúng ta di chuyển chỉ mục đi tiếp. Di chuyển trở lại đầu của mảng khi bạn di chuyển đến cuối.
- Chọn ngẫu nhiên: Chọn node một cách ngẫu nhiên. Giả sử rằng máy thứ X được chọn, thì x có thể được chọn từ hàm `rand.Intn()%n`.
- Sắp xếp các node theo một trọng lượng nhất định và chọn một node có trọng lượng lớn nhất hoặc nhỏ nhất.

Nếu yêu cầu không thành công, chúng ta vẫn cần cơ chế để thử lại. Đối với thuật toán ngẫu nhiên, có khả năng bạn sẽ chọn node lỗi lần nữa.

5.4.2 Cân bằng tải dựa trên thuật toán xáo trộn

Giả sử chúng ta cần chọn ngẫu nhiên node gửi yêu cầu và thử lại các node khác khi có lỗi trả về. Vì vậy, chúng ta thiết kế một mảng chỉ mục với kích thước bằng số node. Mỗi lần chúng ta có một yêu cầu mới, chúng ta xáo trộn mảng chỉ mục, sau đó lấy phần tử đầu tiên làm node dịch vụ. Nếu yêu cầu thất bại, ta chọn node tiếp theo. Cứ thử lại và tiếp tục, ...

```

var endpoints = []string {
    "100.69.62.1:3232",
    "100.69.62.32:3232",
    "100.69.62.42:3232",
    "100.69.62.81:3232",
    "100.69.62.11:3232",
    "100.69.62.113:3232",
    "100.69.62.101:3232",
}

// shuffle hàm xáo trộn chỉ mục
func shuffle(slice []int) {
    for i := 0; i < len(slice); i++ {
        a := rand.Intn(len(slice))
        b := rand.Intn(len(slice))
        slice[a], slice[b] = slice[b], slice[a]
    }
}

func request(params map[string]interface{}) error {
    var indexes = []int {0,1,2,3,4,5,6}
    var err error

    // gọi shuffle để xáo trộn các index
    shuffle(indexes)

    // số lần thử lại là 3
    maxRetryTimes := 3

    idx := 0
    for i := 0; i < maxRetryTimes; i++ {
        err = apiRequest(params, indexes[idx])
        if err == nil {
            break
        }
        idx++
    }

    if err != nil {
        // logging
        return err
    }

    return nil
}

```

Chúng ta duyệt qua các chỉ mục và hoán đổi chúng, tương tự như phương pháp xáo trộn mà chúng ta thường sử dụng khi chơi bài.

5.4.2.1 Vấn đề sinh số ngẫu nhiên

Thực sự không có vấn đề? Trong thực tế, vẫn còn vấn đề. Có hai cạm bẫy tiềm ẩn trong chương trình trên là:

1. Không có random seed. Khi không có `random seed`, trình tự của các lần random `rand.Intn()` là cố định.
2. Xáo trộn không đều, điều này sẽ khiến node đầu tiên của toàn bộ mảng có xác suất được chọn cao và phân phối tài giữa các node không cân bằng.

Điểm đầu tiên tương đối đơn giản nên chúng tôi không nêu ví dụ cụ thể. Về điểm thứ hai, chúng ta có thể sử dụng kiến thức về xác suất để chứng minh điều đó. Giả sử rằng mỗi lựa chọn là thực sự ngẫu nhiên, xác suất mà node ở vị trí đầu tiên không được chọn trong trao đổi `len(slice)` là $((6/7)*(6/7))^7 \approx 0.34$. Trong trường hợp phân phối đồng đều, chúng ta chắc chắn muốn xác suất phần tử đầu tiên được phân phối tại bất kỳ vị trí nào bằng nhau, do đó xác suất được chọn ngẫu nhiên phải xấp xỉ bằng $1/7=0.14$.

Rõ ràng, thuật toán xáo trộn được đưa ra ở đây có xác suất 30% không hoán đổi các yếu tố cho bất kỳ vị trí nào. Vì vậy, tất cả các yếu tố có xu hướng ở lại vị trí ban đầu của chúng. Bởi vì mỗi lần chúng ta nhập cùng một chuỗi cho mảng `shuffle`, phần tử đầu tiên có xác suất được chọn cao hơn. Trong trường hợp cân bằng tải, có nghĩa là tải máy đầu tiên trong mảng node sẽ cao hơn nhiều so với các máy khác (ít nhất gấp 3 lần).

5.4.2.2 Sửa thuật toán xáo trộn

Thuật toán `fishing-yates` đã chứng minh tính đúng đắn về mặt toán học. Ý tưởng chính của nó là chọn một giá trị ngẫu nhiên rồi đặt ở cuối mảng, và cứ thế tiếp tục. Ví dụ:

```
func shuffle(indexes []int) {
    for i:=len(indexes); i>0; i-- {
        lastIdx := i - 1
        idx := rand.Int(i)
        indexes[lastIdx], indexes[idx] = indexes[idx], indexes[lastIdx]
    }
}
```

Thuật toán đã được hiện thực trong thư viện chuẩn `math/rand` của Go:

```
func shuffle(n int) []int {
    b := rand.Perm(n)
    return b
}
```

Hiện tại, chúng ta có thể sử dụng `rand.Perm` để lấy mảng chỉ mục mà chúng ta muốn.

5.4.3 Kiểm tra tính ảnh hưởng của thuật toán cân bằng tải

Giả sử, ta cần chọn một node từ N node để gửi yêu cầu. Sau khi yêu cầu ban đầu kết thúc, các yêu cầu tiếp theo sẽ xáo trộn lại mảng, do đó không có mối quan hệ nào giữa hai yêu cầu. Ví thế, thuật toán ở trên sẽ không cần khởi tạo bất kì random seed nào.

Tuy nhiên, trong một số trường hợp đặc biệt, chẳng hạn như khi sử dụng ZooKeeper, khi máy khách khởi tạo việc lựa chọn node từ nhiều node dịch vụ, một kết nối được thiết lập cho node. Yêu cầu máy khách sau đó được gửi đến node. Node tiếp theo trong danh sách được chọn cho đến khi không còn node nào có sẵn. Lúc này, việc lựa chọn node kết nối ban đầu là "đúng chuẩn" ngẫu nhiên. Tuy nhiên, tất cả các máy khách sẽ kết nối với cùng một ZooKeeper khi chúng khởi động cùng lúc, lúc này sẽ không có tải cân bằng. Nếu doanh nghiệp của bạn cần phát triển tính năng hàng ngày, thì bạn phải xem xét liệu có một tình huống tương tự như trên xảy ra không. Cách đặt random seed cho thư viện rand:

```
rand.Seed(time.Now().UnixNano())
```

Lý do cho những kết luận này là phiên bản trước của thư viện Open source ZooKeeper được sử dụng rộng rãi đã mắc phải những lỗi trên và mãi đến đầu năm 2016, vấn đề mới được khắc phục.

5.4.4 Kiểm tra lại ảnh hưởng của thuật toán cân bằng tải

Chúng ta không xét trường hợp cân bằng tải có trọng số ở đây. Bây giờ, điều quan trọng nhất là sự cân bằng. Chúng ta chỉ đơn giản so sánh thuật toán xáo trộn trong phần mở đầu với kết quả của thuật toán Fisher-Yates:

main.go

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func init() {
    rand.Seed(time.Now().UnixNano())
}

func shuffle1(slice []int) {
    for i := 0; i < len(slice); i++ {
        a := rand.Intn(len(slice))
        b := rand.Intn(len(slice))
        slice[a], slice[b] = slice[b], slice[a]
    }
}

func shuffle2(indexes []int) {
    for i := len(indexes); i > 0; i-- {
        lastIdx := i - 1
        idx := rand.Intn(i)
        indexes[lastIdx], indexes[idx] = indexes[idx], indexes[lastIdx]
    }
}

func main() {
    var cnt1 = map[int]int{}
    for i := 0; i < 1000000; i++ {
        var sl = []int{0, 1, 2, 3, 4, 5, 6}
        shuffle1(sl)
        cnt1[sl[0]]++
    }

    var cnt2 = map[int]int{}
    for i := 0; i < 1000000; i++ {
        var sl = []int{0, 1, 2, 3, 4, 5, 6}
        shuffle2(sl)
        cnt2[sl[0]]++
    }

    fmt.Println(cnt1, "\n", cnt2)
}
```

Kết quả:

```
map[0:224436 1:128780 5:129310 6:129194 2:129643 3:129384 4:129253]
map[6:143275 5:143054 3:143584 2:143031 1:141898 0:142631 4:142527]
```

Dựa vào kết quả trên chúng ta thấy được sau khi sử dụng thuật toán Fisher-Yates thì sự cân bằng được tốt hơn, rải đều từ các node 0 -> node 6, trung bình khoảng 142000. Còn đối với thuật toán ban đầu thì ở node 0 đã chiếm hơn 220000, các node sau thì trung bình khoảng 128000.

Liên kết

- Phản tiếp theo: [Quản lý cấu hình trong hệ thống phân tán](#)
- Phản trước: [Hệ thống tác vụ có trì hoãn](#)
- Mục lục

5.5 Quản lý cấu hình trong hệ thống phân tán

Trong hệ thống phân tán, thường có những vấn đề gây phiền cho chúng ta. Mặc dù, hiện tại có một số cách để thay đổi cấu hình nhưng vẫn bị hạn chế bởi cách hoạt động nội bộ của hệ thống và lúc này sẽ không có cách nào để thay đổi cấu hình một cách thuận tiện nhất. Ví dụ: để giới hạn dòng chảy xuống `downstream`, chúng ta có thể tích luỹ dữ liệu lại và sau đó, nếu lượng tích luỹ đến ngưỡng về thời gian hay số lượng thì ta bắt đầu gửi đi, điều này tránh được việc gửi quá nhiều cho `downstream`. Với trường hợp này, ta lại rất khó để thay đổi cấu hình.

Do đó, trong chương này, ta sẽ tìm hiểu cách cấu hình cho hệ thống phân tán bằng Go và các vấn đề cần cân nhắc khi sử dụng cách cấu hình trực tuyến.

Quản lý cấu hình trong hệ thống phân tán là một vấn đề cực kì khó. Ngay cả một công ty công nghệ hàng đầu thế giới như Google cũng đã gặp vấn đề và làm cho họ giảm chất lượng dịch vụ rất nhiều. Chi tiết bài viết của Google [ở đây](#).

5.5.1 Thảo luận các ví dụ

5.5.1.1 Hệ thống báo cáo

Trong các hệ thống **OLAP** (Online analytical processing) hoặc một số nền tảng dữ liệu ngoại tuyến, sau một thời gian dài phát triển, các chức năng của toàn bộ hệ thống đã dần ổn định. Các dữ liệu đã có sẵn và hầu hết các thay đổi để hiển thị chỉ liên quan tới việc thay đổi câu truy vấn SQL. Lúc này, ta nghĩ tới việc có thể cấu hình được các câu truy vấn SQL mà không cần phải sửa đổi code.

Khi doanh nghiệp đưa ra các yêu cầu mới, việc chúng ta cần làm là cấu hình lại câu SQL cho hệ thống. Những thay đổi này có thể được thực hiện trực tiếp mà không cần khởi động lại.

5.5.1.2 Cấu hình mang tính doanh nghiệp

Nền tảng (Platform) của một công ty lớn luôn phục vụ cho nhiều business khác nhau và mỗi business được gán một ID duy nhất. Nền tảng này được tạo thành từ nhiều module và cùng chia sẻ một business. Khi công ty mở một dây chuyền sản phẩm mới, nó cần phải được thông qua bởi tất cả các hệ thống trong nền tảng. Lúc này, chắc chắn là sẽ tồn rất nhiều thời gian để nó có thể chạy được. Ngoài ra, các loại cấu hình toàn cục cần phải được quản lý theo cách thống nhất, các logic cộng và trừ cũng phải được quản lý theo cách thống nhất. Khi cấu hình này được thay đổi, hệ thống cần phải tự động thông báo cho toàn bộ hệ thống của nền tảng mà không cần sự can thiệp của con người (hoặc chỉ can thiệp rất đơn giản, chẳng hạn như kiểm toán nháp chuột một phát).

Ngoài quản lý trong lĩnh vực kinh doanh, nhiều công ty Internet còn phải kinh doanh theo quy định của thành phố. Khi doanh nghiệp được mở ở một thành phố, ID thành phố mới sẽ tự động được thêm vào danh sách trong hệ thống. Bằng cách này, quá trình kinh doanh có thể chạy tự động.

Một ví dụ khác, có nhiều loại hoạt động trong hệ điều hành của một công ty. Một số hoạt động có thể gặp những sự kiện bất ngờ (như khủng hoảng quan hệ công chúng), và hệ thống cần tắt chức năng liên quan lĩnh vực đó đi. Lúc này, một số công tắc sẽ được sử dụng để tắt nhanh các chức năng tương ứng. Hoặc nhanh chóng xóa ID của hoạt động mà bạn muốn khỏi danh sách chứa. Trong chương Web, chúng ta biết rằng đôi khi cần phải có một hệ thống để đo được lưu lượng truy cập vào các chức năng. Chúng ta có thể chủ động lấy thông tin này kết hợp với cấu hình hệ thống để tắt một tính năng trong trường hợp có lưu lượng lớn bất thường.

5.5.2 Sử dụng etcd để thực hiện cập nhật cấu hình

etcd là một kho lưu trữ key-value phân tán, nó có tính nhất quán mạnh mẽ, có khả năng chịu lỗi cao khi có một node trong cluster có sự cố hay lỗi do network. etcd được viết bằng ngôn ngữ Go, sử dụng thuật toán đồng thuận Raft để giao tiếp và bình chọn leader giữa các node trong hệ thống. Hiện nay có khá nhiều sản phẩm công nghệ sử dụng etcd như [Kubernetes](#), [Rook](#)....

Ở ví dụ này chúng ta sẽ sử dụng etcd để thực hiện đọc cấu hình và cập nhật tự động cấu hình cho các máy khách.

5.5.2.1 Định nghĩa cấu hình

Cấu hình đơn giản, bạn có thể lưu trữ nội dung hoàn toàn trong etcd. Ví dụ:

```
etcdctl get /configs/remote_config.json
{
  "addr" : "127.0.0.1:1080",
  "aes_key" : "01B345B7A9ABC00F0123456789ABCDAB",
  "https" : false,
  "secret" : "",
  "private_key_path" : "",
  "cert_file_path" : ""
}
```

5.5.2.2 Tạo ứng dụng khách etcd

Cấu trúc khởi tạo kết nối bằng package etcd cho người dùng.

```
cfg := client.Config{
    Endpoints:      []string{"http://127.0.0.1:2379"},
    Transport:       client.DefaultTransport,
    HeaderTimeoutPerRequest: time.Second,
}
```

5.5.2.3 Lấy cấu hình

```
resp, err = kapi.Get(context.Background(), "/path/to/your/config", nil)
if err != nil {
    log.Fatal(err)
} else {
    log.Printf("Get is done. Metadata is %q\n", resp)
    log.Printf("%q key has %q value\n", resp.Node.Key, resp.Node.Value)
}
```

Dùng phương thức `Get()` của KeysAPI trong etcd tương đối đơn giản. Các bạn có thể tham khảo thêm các API khác [ở đây](#).

5.5.2.4 Đăng ký tự động cập nhật cấu hình

```
kapi := client.NewKeysAPI(c)
w := kapi.Watcher("/path/to/your/config", nil)
go func() {
    for {
        resp, err := w.Next(context.Background())
        log.Println(resp, err)
        log.Println("new values is ", resp.Node.Value)
    }
}()
```

Bằng cách theo dõi những thay đổi sự kiện của đường dẫn cấu hình, khi có nội dung thay đổi trong đường dẫn, chúng ta có thể nhận được thông báo thay đổi cùng với giá trị đã thay đổi.

5.5.2.5 Chương trình hoàn chỉnh

```
package main

import (
    "log"
    "time"

    "golang.org/x/net/context"
    "github.com/coreos/etcd/client"
)

var configPath = `/configs/remote_config.json`
var kapi client.KeysAPI

type ConfigStruct struct {
    Addr      string `json:"addr"`
    AesKey    string `json:"aes_key"`
    HTTPS     bool   `json:"https"`
    Secret    string `json:"secret"`
    PrivateKeyPath string `json:"private_key_path"`
    CertFilePath  string `json:"cert_file_path"`
}

var appConfig ConfigStruct

func init() {
    cfg := client.Config{
        Endpoints:          []string{"http://127.0.0.1:2379"},
        Transport:           client.DefaultTransport,
        HeaderTimeoutPerRequest: time.Second,
    }

    c, err := client.New(cfg)
    if err != nil {
        log.Fatal(err)
    }
    kapi = client.NewKeysAPI(c)
    initConfig()
}

func watchAndUpdate() {
    w := kapi.Watcher(configPath, nil)
    go func() {
        for {
            resp, err := w.Next(context.Background())
            if err != nil {
                log.Fatal(err)
            }
            log.Println("new values is ", resp.Node.Value)

            err = json.Unmarshal([]byte(resp.Node.Value), &appConfig)
            if err != nil {
                log.Fatal(err)
            }
        }
    }()
}

func initConfig() {
    resp, err = kapi.Get(context.Background(), configPath, nil)
    if err != nil {
        log.Fatal(err)
    }
}
```

```

err := json.Unmarshal(resp.Node.Value, &appConfig)
if err != nil {
    log.Fatal(err)
}
}

func getConfig() ConfigStruct {
    return appConfig
}

func main() {
    // khởi tạo ứng dụng của bạn
}

```

Nếu là doanh nghiệp nhỏ, có thể sử dụng luôn ví dụ trên để hiện thực chức năng mà bạn cần.

Có một vài lưu ý ở đây, chúng ta sẽ làm rất nhiều thứ khi cập nhật cấu hình: phản hồi watch, phân tích json, và các hoạt động này không phải là `atomic`. Khi cấu hình bị thay đổi nhiều lần trong một quy trình dịch vụ, có thể xuất hiện sự không thống nhất logic giữa các yêu cầu xảy ra trước và sau khi cấu hình thay đổi. Do đó, khi bạn sử dụng cách tiếp cận trên để cập nhật cấu hình của mình, bạn cần sử dụng cùng một cấu hình trong suốt vòng đời của một yêu cầu. Cách thức thực hiện cụ thể nên lấy cấu hình một lần khi yêu cầu bắt đầu, và sau đó được truyền tiếp đi cho đến hết vòng đời của yêu cầu.

5.5.3 Sự phình to của cấu hình

Khi doanh nghiệp phát triển, áp lực lên hệ thống cấu hình có thể ngày càng lớn hơn và số lượng tệp cấu hình có thể là hàng chục nghìn. Máy khách cũng có hàng chục nghìn và việc lưu trữ nội dung cấu hình bên trong etcd không còn phù hợp nữa. Khi số lượng tệp cấu hình mở rộng, ngoài các vấn đề về thông lượng của hệ thống lưu trữ, còn có các vấn đề về quản lý đối với thông tin cấu hình. Chúng ta cần quản lý các quyền của cấu hình tương ứng và chúng ta cần cấu hình cụm lưu trữ theo lưu lượng truy cập. Nếu có quá nhiều máy khách, khiến hệ thống lưu trữ cấu hình không thể chịu được lượng lớn QPS, thì có thể cần phải thực hiện tối ưu hóa cache ở phía máy khách,

Đó là lý do tại sao các công ty lớn luôn phải phát triển một hệ thống cấu hình phức tạp cho doanh nghiệp của họ.

5.5.4 Quản lý phiên bản của cấu hình

Trong quy trình quản lý cấu hình, không thể tránh khỏi việc người điều hành thực hiện sai. Ví dụ: khi cập nhật cấu hình, một cấu hình không thể đọc được. Lúc này, chúng ta giải quyết bằng cách kiểm tra toàn bộ.

Đôi khi việc sai cấu hình có thể không phải là do vấn đề với định dạng, mà là vấn đề logic. Ví dụ: khi chúng ta viết SQL, chúng ta chọn ít trường hơn. Khi chúng ta cập nhật cấu hình, chúng ta vô tình làm mất một trường trong chuỗi json và khiến chương trình hiểu cấu hình mới và thực hiện một logic mới. Cách nhanh nhất và hiệu quả nhất để ngăn chặn những lỗi làm này nhanh chóng là quản lý phiên bản và hỗ trợ khôi phục theo phiên bản.

Khi cấu hình được cập nhật, chúng ta sẽ chỉ định số phiên bản cho từng nội dung của cấu hình, và luôn ghi lại nội dung và số phiên bản trước mỗi lần thay đổi, thực hiện quay ngược bản trước khi phát hiện sự cố với cấu hình mới.

Một cách phổ biến trong thực tế là sử dụng MySQL để lưu trữ các phiên bản khác nhau của tệp cấu hình hoặc chuỗi cấu hình. Khi bạn cần quay lại, chỉ cần thực hiện một truy vấn đơn giản.

5.5.5 Khả năng chịu lỗi ở máy khách

Sau khi cấu hình của hệ thống kinh doanh được chuyển đến trung tâm cấu hình, điều đó không có nghĩa là hệ thống của chúng ta đã hoàn thành nhiệm vụ. Khi trung tâm cấu hình ngừng hoạt động, chúng ta cũng cần một cơ chế chịu lỗi, ít nhất là để đảm bảo rằng doanh nghiệp vẫn có thể hoạt động trong thời gian này. Điều này đòi hỏi hệ thống phải lấy đủ thông tin cấu hình cần thiết trước khi trung tâm cấu hình ngừng hoạt động. Ngay cả khi thông tin này không đủ mới.

Cụ thể, khi cung cấp SDK đọc cấu hình cho một dịch vụ, tốt nhất là lưu cache cấu hình thu được trên đĩa của máy nghiệp vụ. Khi trung tâm cấu hình không hoạt động, bạn có thể trực tiếp sử dụng nội dung của đĩa cứng. Khi kết nối lại được với trung tâm cấu hình, các nội dung sẽ được cập nhật.

Hãy xem xét kỹ vấn đề thống nhất dữ liệu khi thêm cache. Các máy kinh doanh có thể không thống nhất về cấu hình do lỗi mạng, chúng ta có thể biết được nó đang diễn ra bằng hệ thống giám sát.

Chúng ta sử dụng một cách để giải quyết các vấn đề của việc cập nhật cấu hình, nhưng đồng thời chúng ta lại mang đến những vấn đề mới bằng việc sử dụng cách đó. Trong thực tế, chúng ta phải suy nghĩ rất nhiều về từng quyết định để chúng ta không bị thiệt hại quá nhiều khi vấn đề xảy ra.

Liên kết

- Phần tiếp theo: [Trình thu thập thông tin phân tán](#)
- Phần trước: [Cân bằng tải](#)
- [Mục lục](#)

5.6 Trình thu thập thông tin phân tán

Sự bùng nổ thông tin trong kỷ nguyên Internet là một vấn đề khiến nhiều người cảm thấy đau đầu. Vô số tin tức, thông tin và video đang xâm chiếm dần thời gian của chúng ta. Mặt khác, khi chúng ta thực sự cần dữ liệu, chúng ta cảm thấy rằng dữ liệu không dễ dàng gì có được. Ví dụ: chúng ta muốn biết về những gì mọi người đang thảo luận và quan tâm. Nhưng chúng ta không có thời gian để đọc từng diễn đàn được yêu thích, lúc này chúng ta muốn sử dụng công nghệ để đưa những thông tin cần vào cơ sở dữ liệu. Quá trình này có thể tốn một vài tháng hoặc một năm. Cũng có thể chúng ta muốn lưu những thông tin hữu ích mà vô tình gặp trên Internet như các cuộc thảo luận chất lượng cao của những người tài năng được tập hợp trong một diễn đàn rất nhỏ. Vào một lúc nào đó trong tương lai, chúng ta tìm lại được những thông tin đó và rút ra được những kết luận giá trị mà đến lúc này mới nhận ra.

Ngoài nhu cầu giải trí, có rất nhiều tài liệu mở quý giá trên Internet. Trong những năm gần đây, **Deep learning** đã và đang rất hot, và **machine learning** thường không quan tâm đến việc thiết lập ban đầu đúng hay không, các thông số có được điều chỉnh chính xác không, mà là ở giai đoạn khởi tạo ban đầu: không có dữ liệu.

Việc có một chương trình phục vụ việc thu thập thông tin hiện nay rất quan trọng.

5.6.1 Trình thu thập thông tin độc lập dựa trên Colly

Ví dụ sau đưa ra một ví dụ về trình thu thập thông tin đơn giản sử dụng thư viện **Colly**. Việc dùng Go sẽ cực kì thuận tiện để viết một trình thu thập thông tin cho trang web, chẳng hạn như việc thu thập thông tin trang web (www.abcdefg.com là trang web ảo):

main.go

```
package main

import (
    "fmt"
    "regexp"
    "time"

    "github.com/gocolly/colly"
)

var visited = map[string]bool{}

func main() {
    // tạo đối tượng collector
    c := colly.NewCollector(
        colly.AllowedDomains("www.abcdefg.com"),
        colly.MaxDepth(1),
    )

    // giả sử phù hợp với regex sau là trang chi tiết
    detailRegex, _ := regexp.Compile(`/go/go\?p=\d+$`)
    // giả sử phù hợp với regex sau là danh sách các trang
    listRegex, _ := regexp.Compile(`/t/\d+\#\w+`)

    // tắt cả các thẻ đều có hàm callback
    c.OnHTML("a[href]", func(e *colly.HTMLElement) {
        link := e.Attr("href")

        // duyệt qua từng trang chi tiết hoặc list trang
        if visited[link] && (detailRegex.Match([]byte(link)) || listRegex.Match([]byte(link))) {
            return
        }
    })
}
```

```

// nếu không phải trang chi tiết hoặc danh sách trang thì bỏ qua.
if !detailRegex.Match([]byte(link)) && !listRegex.Match([]byte(link)) {
    println("not match", link)
    return
}

// hầu hết các trang web đều có chức năng chặn
// cần có thời gian chờ.
time.Sleep(time.Second)
println("match", link)

visited[link] = true

time.Sleep(time.Millisecond * 2)
c.Visit(e.Request.AbsoluteURL(link))
})

err := c.Visit("https://www.abcdefg.com/go/go")
if err != nil {fmt.Println(err)}
}

```

5.6.2 Trình thu thập thông tin phân tán

Hãy tưởng tượng rằng hệ thống phân tích thông tin của bạn đang chạy rất nhanh. Tốc độ thu thập thông tin đã trở thành nút cổ chai. Mặc dù bạn có thể sử dụng tất cả các tính năng xử lý đồng thời tuyệt vời của Go để dùng hết hiệu suất CPU và băng thông mạng, nhưng bạn vẫn muốn tăng tốc độ thu thập thông tin của trình thu thập thông tin. Trong nhiều ngữ cảnh, tốc độ mang nhiều ý nghĩa:

- Đối với thương mại điện tử, cụ thể là cuộc chiến giá cả, tôi sẽ hy vọng mình có được giá mới nhất của đối thủ khi chúng thay đổi, và hệ thống sẽ tự động điều chỉnh giá của sản phẩm của tôi lại sao cho phù hợp.
- Đối với các dịch vụ cung cấp thông tin Feed, tính kịp thời của thông tin rất quan trọng. Nếu tin tức thu thập là tin tức của ngày hôm qua, nó sẽ không có ý nghĩa gì với người dùng.

Vì vậy, chúng ta cần hệ thống thu thập thông tin phân tán. Về bản chất, các trình thu thập thông tin phân tán là tập hợp của một hệ thống phân phối và thực thi tác vụ. Trong các hệ thống phân phối tác vụ phổ biến, sẽ có sự sai lệch tốc độ giữa upstream và downstream nên sẽ luôn tồn tại một hàng đợi tin nhắn.

Công việc chính của upstream là thu thập thông tin tất cả các "trang" đích từ một điểm bắt đầu được cấu hình sẵn. Nội dung html của trang danh sách sẽ chứa các liên kết đến các trang chi tiết. Số lượng trang chi tiết thường gấp 10 đến 100 lần so với trang danh sách, vì vậy chúng ta xem các trang chi tiết này như "tác vụ" và phân phối chúng thông qua hàng đợi tin nhắn.

Để thu thập thông tin trang, điều quan trọng là không có sự lặp lại xảy ra thường xuyên trong quá trình thực thi, vì nó sẽ tạo các kết quả sai (ví dụ trên sẽ chỉ thu thập nội dung trang chứ không phải phần bình luận).

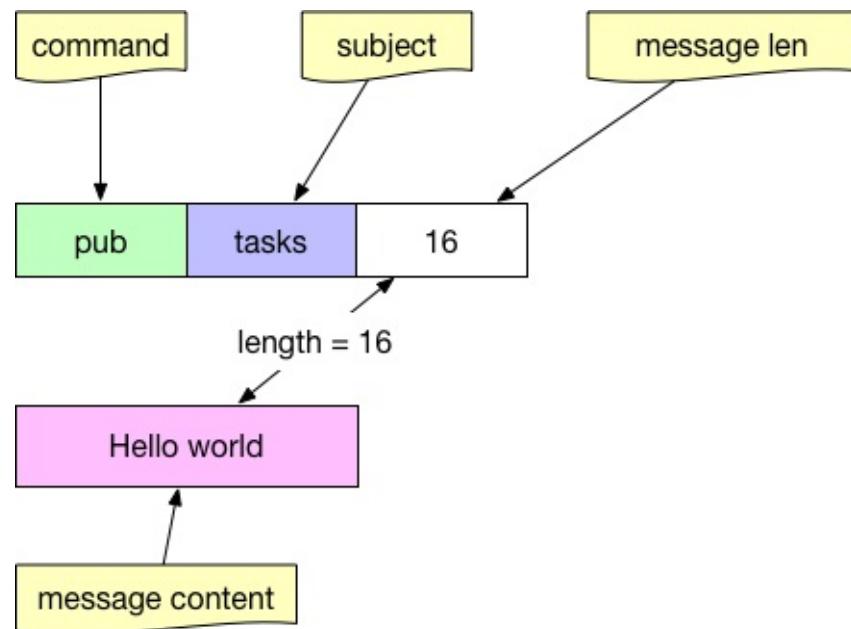
Trong phần này, chúng ta sẽ hiện thực trình thu thập thông tin đơn giản dựa trên hàng đợi tin nhắn. Cụ thể ở đây là sử dụng các NATS để phân phối tác vụ. Trong thực tế, tùy vào yêu cầu về độ tin cậy của thông điệp và cơ sở hạ tầng của công ty nên sẽ ảnh hưởng tới việc chọn công nghệ của từng doanh nghiệp.

5.6.2.1 Giới thiệu về NATS

NATS là một hàng đợi tin nhắn phân tán (distributed message queue) hiệu năng cao được lập trình bằng Go, ta nên sử dụng nó cho các tình huống yêu cầu tính đồng thời cao, thông lượng cao. Những phiên bản NATS ban đầu mang thiên hướng về tốc độ và không hỗ trợ tính persistence. Kể từ 16 năm trước, NATS đã hỗ trợ tính persistence dựa trên log thông qua NATS Streaming, cũng như nhắn tin đáng tin cậy. Dưới đây là những ví dụ đơn giản về NATS.

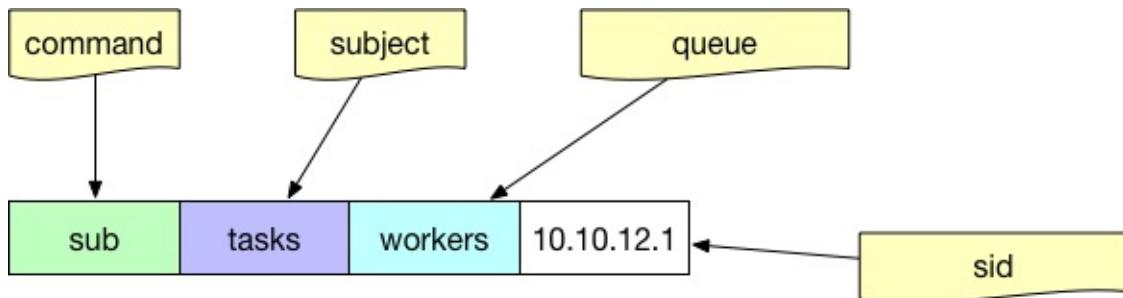
Máy chủ của NATS là gnatsd. Phương thức giao tiếp giữa máy khách và gnatsd là giao thức văn bản dựa trên tcp:

Gửi tin nhắn đi có chứa chủ đề cho một tác vụ:



Pub trong giao thức NATS

Theo dõi các tác vụ bằng chủ đề trên hàng đợi của các worker:



Sub trong giao thức NATS

Tham số hàng đợi là tùy chọn. Nếu bạn muốn cân bằng tài các tác vụ ở phía người dùng, thay vì tất cả mọi người nhận cùng một kênh, bạn nên gán một tên hàng đợi cho một người dùng.

Sản xuất tin nhắn

Sản xuất tin nhắn được chỉ định bằng `topic`:

```

nc, err := nats.Connect(nats.DefaultURL)
if err != nil {return}

// tham số đầu tiên tasks là tên topic
// tham số tiếp theo là nội dung tin nhắn gửi đi
err = nc.Publish("tasks", []byte("your task content"))

nc.Flush()
  
```

Tiêu thụ tin nhắn

Việc sử dụng trực tiếp API đăng ký của Nats không thể thoả được mục đích phân phối tác vụ, vì pub-sub là broadcast nên tất cả consumer sẽ nhận được cùng một thông điệp.

Ngoài cách đăng ký bình thường, Nats đã cung cấp chức năng đăng ký hàng đợi. Bằng cách cung cấp tên nhóm hàng đợi (tương tự như nhóm `consumer` trong Kafka), các tác vụ có thể được phân phối cho `consumer` một cách cân bằng.

```
nc, err := nats.Connect(nats.DefaultURL)
if err != nil {return}

// đăng ký hàng đợi đồng nghĩa với việc giúp cho hệ thống cân bằng tải cho việc phân phoitask.
// hng đợi trong Nats tương tự về mặt khái niệm với group consumer ở Kafka

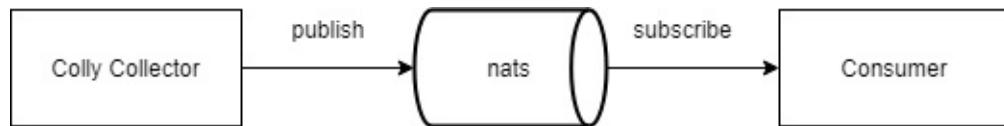
sub, err := nc.QueueSubscribeSync("tasks", "workers")
if err != nil {return}

var msg *nats.Msg
for {
    msg, err = sub.NextMsg(time.Hour * 10000)
    if err != nil {break}
    // xử lý tin nhắn nhận được
}
```

5.6.3 Tạo tin nhắn bằng cách kết hợp NATS và Colly

Bên dưới là một trình thu thập được tuỳ chỉnh cho trang web là `www.abcdefg.com`, `www.hijklmn.com` (ví dụ bên dưới), và sử dụng một phương thức `factory` đơn giản để ánh xạ trình thu thập tới đúng server. Khi trang web đang duyệt là một trang danh sách, ta cần phân tích tất cả các liên kết trong trang hiện tại và gửi liên kết của trang chi tiết đến hàng đợi tin nhắn.

Mô hình hoạt động:



main.go

```
package main

import (
    "fmt"
    "net/url"

    "github.com/gocolly/colly"
)

var domain2Collector = map[string]*colly.Collector{}
var nc *nats.Conn
var maxDepth = 10
var natsURL = "nats://localhost:4222"

func factory(urlStr string) *colly.Collector {
    u, _ := url.Parse(urlStr)
    return domain2Collector[u.Host]
}

func initABCDECollector() *colly.Collector {
    c := colly.NewCollector(
        colly.AllowedDomains("www.abcdefg.com"),
        colly.MaxDepth(maxDepth),
    )
    c.OnResponse(func(resp *colly.Response) {
        // thực hiện một số xác nhận
    })
}
```

```

    // ví dụ như xác nhận trang đã thu thập thông tin đã được lưu ở MySQL
  })

c.OnHTML("a[href]", func(e *colly.HTMLElement) {
  // chiến lược chống anti-reptile
  link := e.Attr("href")
  time.Sleep(time.Second * 2)

  // duyệt qua list trang thông thường
  if listRegex.Match([]byte(link)) {
    c.Visit(e.Request.AbsoluteURL(link))
  }
  // gửi các liên kết trong list trang vào hàng đợi tin nhắn
  if detailRegex.Match([]byte(link)) {
    err = nc.Publish("tasks", []byte(link))
    nc.Flush()
  }
})
return c
}

func initHijklCollector() *colly.Collector {
  c := colly.NewCollector(
    colly.AllowedDomains("www.hijklmn.com"),
    colly.MaxDepth(maxDepth),
  )

  c.OnHTML("a[href]", func(e *colly.HTMLElement) {
  })

  return c
}

func init() {
  domain2Collector["www.abcdefg.com"] = initV2exCollector()
  domain2Collector["www.hijklmn.com"] = initV2fxCollector()

  var err error
  nc, err = nats.Connect(natsURL)
  if err != nil {os.Exit(1)}
}

func main() {
  urls := []string{"https://www.abcdefg.com", "https://www.hijklmn.com"}
  for _, url := range urls {
    instance := factory(url)
    instance.Visit(url)
  }
}

```

5.6.4 Kết hợp trình tiêu thụ tin nhắn với Colly

Phía consumer sẽ đơn giản hơn, chúng ta chỉ cần đăng ký chủ đề tương ứng và truy cập trực tiếp vào trang chi tiết.

main.go

```

package main

import (
  "fmt"
  "net/url"

  "github.com/gocolly/colly"
)

```

```

var domain2Collector = map[string]*colly.Collector{}
var nc *nats.Conn
var maxDepth = 10
var natsURL = "nats://localhost:4222"

func factory(urlStr string) *colly.Collector {
    u, _ := url.Parse(urlStr)
    return domain2Collector[u.Host]
}

func initV2exCollector() *colly.Collector {
    c := colly.NewCollector(
        colly.AllowedDomains("www.abcdefg.com"),
        colly.MaxDepth(maxDepth),
    )
    return c
}

func initV2fxCollector() *colly.Collector {
    c := colly.NewCollector(
        colly.AllowedDomains("www.hijklmn.com"),
        colly.MaxDepth(maxDepth),
    )
    return c
}

func init() {
    domain2Collector["www.abcdefg.com"] = initABCDECollector()
    domain2Collector["www.hijklmn.com"] = initHIJKLCollector()

    var err error
    nc, err = nats.Connect(natsURL)
    if err != nil {os.Exit(1)}
}

func startConsumer() {
    nc, err := nats.Connect(nats.DefaultURL)
    if err != nil {return}

    sub, err := nc.QueueSubscribeSync("tasks", "workers")
    if err != nil {return}

    var msg *nats.Msg
    for {
        msg, err = sub.NextMsg(time.Hour * 10000)
        if err != nil {break}

        urlStr := string(msg.Data)
        ins := factory(urlStr)
        // vì phần downstream chắc chắn là trang chi tiết nên chỉ cần lấy thông tin từ trang này
        ins.Visit(urlStr)
        // prevent being blocked
        time.Sleep(time.Second)
    }
}

func main() {
    startConsumer()
}

```

Về cơ bản, khi lập trình thì các `producer` và `consumer` là giống nhau. Nếu chúng ta muốn có tính linh hoạt trong việc tăng và giảm số các trang web để thu thập thông tin trong tương lai, chúng ta nên suy nghĩ về các tham số và chiến lược cấu hình cho trình thu thập thông tin càng nhiều càng tốt.

Việc sử dụng hệ thống cấu hình đã được đề cập trong phần [cấu hình phân tán](#) nên các bạn có thể tự mình dùng thử nó.

Liên kết

- Phần tiếp theo: [Lời nói thêm](#)
- Phần trước: [Quản lý cấu hình trong hệ thống phân tán](#)
- [Mục lục](#)

5.7 Lời nói thêm

Hệ thống phân tán là một lĩnh vực lớn, những phần chúng tôi giới thiệu trong chương này chỉ là một cái nhìn tổng quan. Vì các hệ thống lớn thường có lưu lượng lớn và tính đồng thời cao, các giải pháp đơn giản thường không đáp ứng được yêu cầu. Để giải quyết vấn đề trong hệ thống có quy mô lớn, chúng ta phải phát triển nhiều hệ thống phân tán. Một số hệ thống rất đơn giản, chẳng hạn như trình tạo ID phân tán và một số hệ thống có thể rất phức tạp, chẳng hạn như công cụ tìm kiếm phân tán.

Dù cho đó là một hệ thống đơn giản hay phức tạp, nó cũng sẽ có giá trị quan trọng trong một ngữ cảnh cụ thể. Chúng tôi hy vọng người đọc sẽ tiếp xúc nhiều hơn với Open source, tích lũy các công cụ cho riêng mình, và "*standing on the shoulders of giants*" (học từ kinh nghiệm của những người đi trước).

Liên kết

- Phần tiếp theo: [Chương 6: Go best practice](#)
- Phần trước: [Trình thu thập thông tin phân tán](#)
- [Mục lục](#)