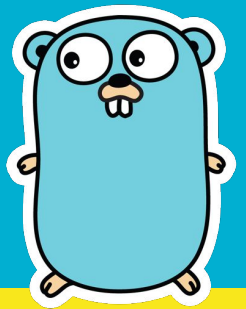
Go 程式設計課 03

—

Go與併發設計

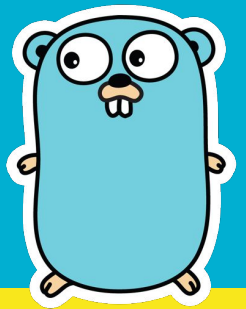
大綱 (1)

- Why we should Concurrency
- 基本資料結構
  - queue
  - stack
  - message queue
- race condition
- deadlock



大綱 (2)

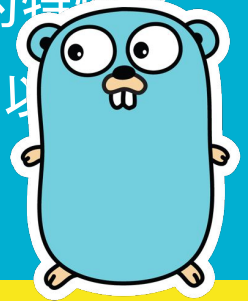
- 並行語法
- 完成 go 之旅的 concurrency
- CSP vs CPS (Continuation-Passing Style)
- thread pool vs event loop
- Go 實作
  - multi-thread stack / queue implement
  - GUI todo list



Why we should Concurrency

Why we should Concurrency?

- 目前的 CPU 時脈已達到上限，硬體大多往多核心發展，然而如果程式碼不可併發(concurrency)，則無法真正地發揮平行處理使效能提升
- Concurrency vs. Parallel
  - 多執行緒: 每個工作結束後，另外一個工作才會開始
  - 平行處理 : 在同個時間運行多個工作
- Golang 中多執行緒稱為" Goroutine"。Goroutine 是 Go 最重要的特性之一，它可以讓開發者輕易做到concurrency，而且它非常輕量，所以開很多 goroutine 也不會有什麼問題。

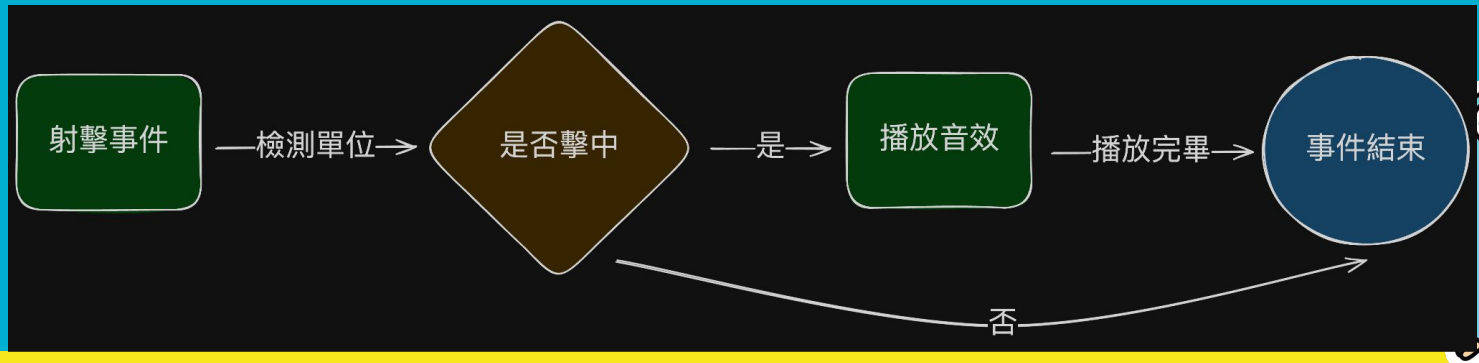


同步非同步概念

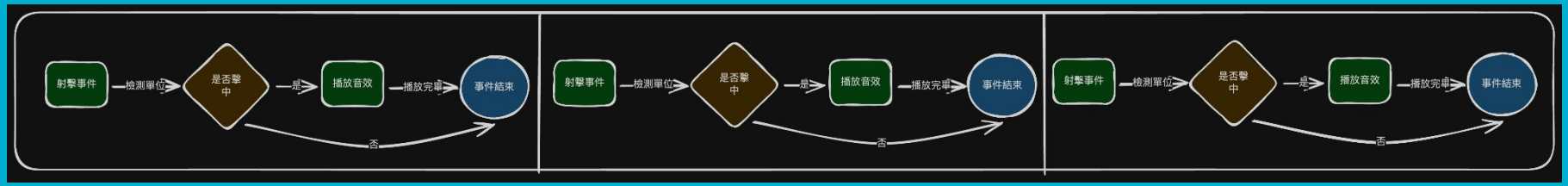
ref: https://ouch1978.github.io/blog/2022/09/25/understand-sync-async-and-multi-thread-with-one-pic

同步與非同步模型

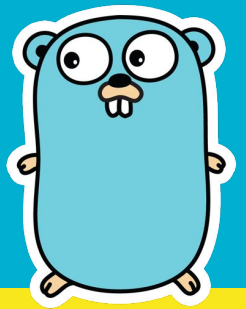
針對所有單位



同步模型

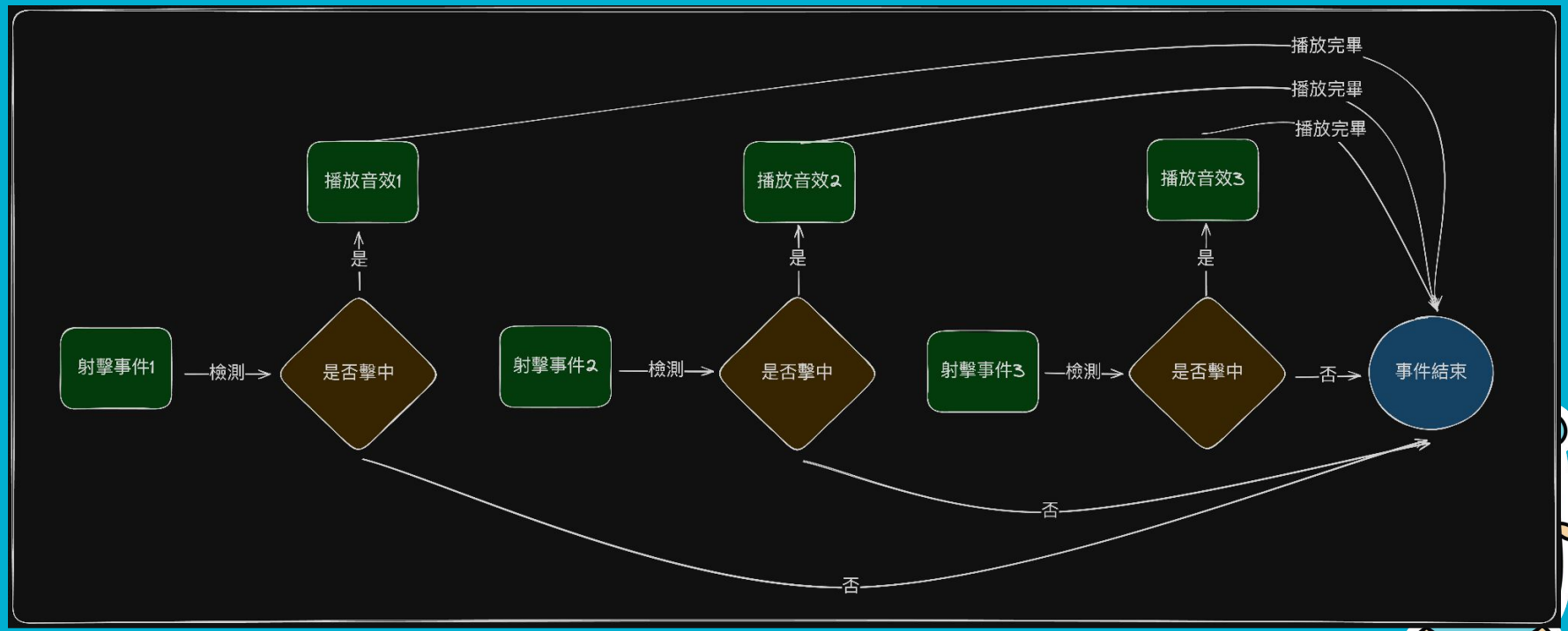


- 同步的運作方式應該是最常見的程式運作方式。
- 從圖裡我們可以看到，所有音效都需要等到前一個音效播放完後才能播放(一定得要等到該動作完成、取得回應之後，才能繼續下一個動作)
- 在這樣的運作方式下，會花大量時間在等待上。



非同步模型

- 非同步和同步最大的差別，就是不用等待上一個動作完成，就可以開始處理下一個動作。
- 從圖裡可以明顯的看得出來，非同步和同步相比，雖然都是只有一個執行者，但透過不等待音效完成，速度快了很多，也減少了大量延遲。
- 也因為等待的時間還可以處理其它工作的關係，真正閒置的時間也大幅的縮短。



同步與非同步的假設(Amdahl's law)

$$\text{理論加速比例} = \frac{1}{\text{不可平行化佔比} + \frac{\text{可平行化佔比}}{\text{平行程度}}}$$

例題:

假設一個程式可以被分成兩個部分：串行部分和可並行部分。串行部分佔整個程式執行時間的30%，而可並行部分佔70%。如果我們對可並行部分進行並行化處理，使其能夠完全並行執行，假設理想情況下，平行程度為5。根據阿姆達爾定律，計算並行化後的加速比。

Ans: 約 2.27 倍

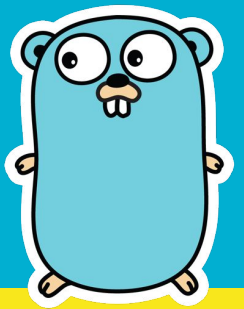
何為“可平行化”?

不佔用計算時間(CPU time)的行為

俗稱: IO （Input/Output）

常見為:

- 外接裝置調用
- 時間相關系統呼叫
- 內外部 API 呼叫
- 特殊庫呼叫(例如加解密)



做做實驗

```go
package main

import "fmt"

func syncFunc(number int) {
    fmt.Println("syncFunc", number)
}

func asyncFunc(number int) {
    fmt.Println("asyncFunc", number)
}

func main() {
    for i := 0; i < 5; i++ {
        syncFunc(i)
    }
}
```

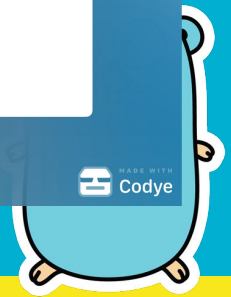
MADE WITH Codye

```go
package main

import "fmt"

func syncFunc(number int) {
    fmt.Println("syncFunc", number)
}

func asyncFunc(number int) {
    fmt.Println("asyncFunc", number)
}

func main() {
    for i := 0; i < 5; i++ {
        go asyncFunc(i)
    }
}
```

MADE WITH Codye

```go
package main

import "fmt"

func syncFunc(number int) {
    fmt.Println("syncFunc", number)
}

func asyncFunc(number int) {
    fmt.Println("asyncFunc", number)
}

func main() {
    for i := 0; i < 5; i++ {
        go asyncFunc(i)
    }

    fmt.Scanln()
}
```

MADE WITH Codye



實驗結果

```
syncFunc 0
syncFunc 1
syncFunc 2
syncFunc 3
syncFunc 4

Process finished with the exit code 0
```

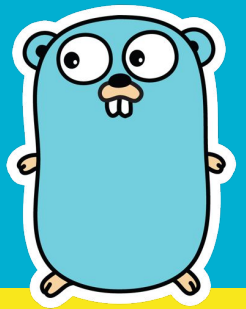
依序輸出

```
Process finished with the exit code 0
```

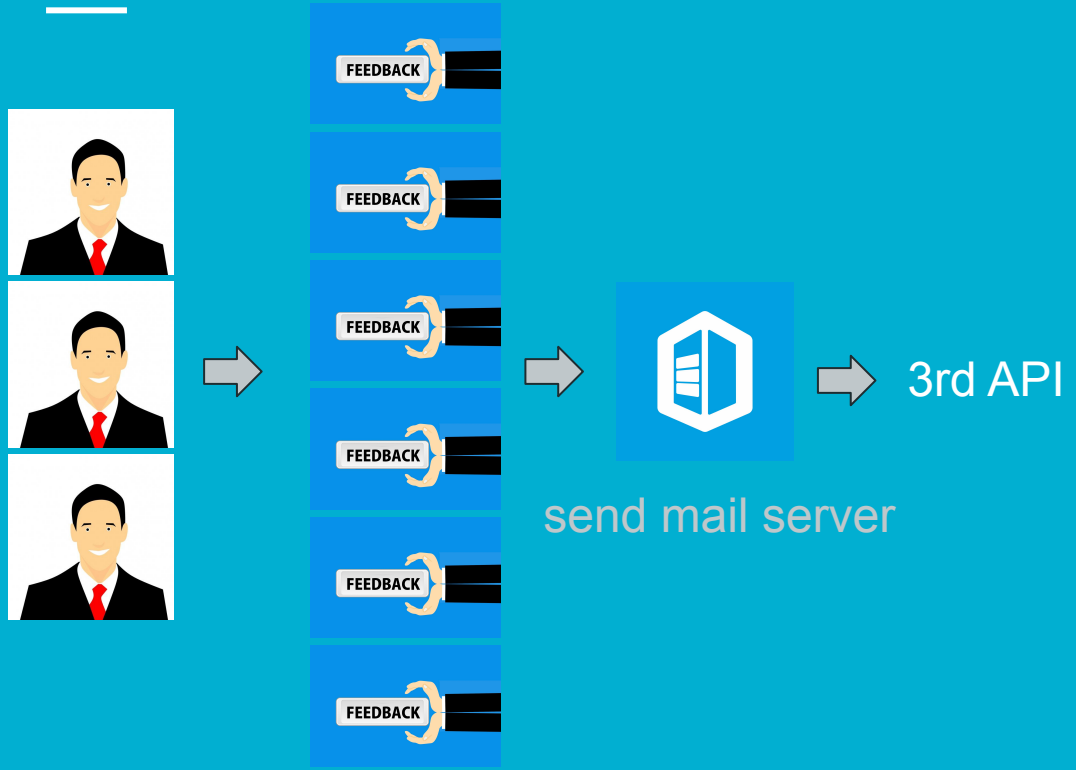
無輸出

亂序輸出

```
asyncFunc 1
asyncFunc 3
asyncFunc 4
asyncFunc 2
asyncFunc 0
```



實際案例 send mail server

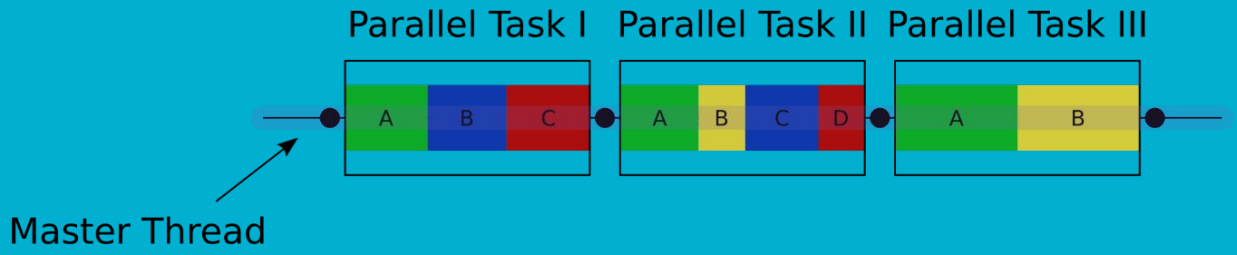


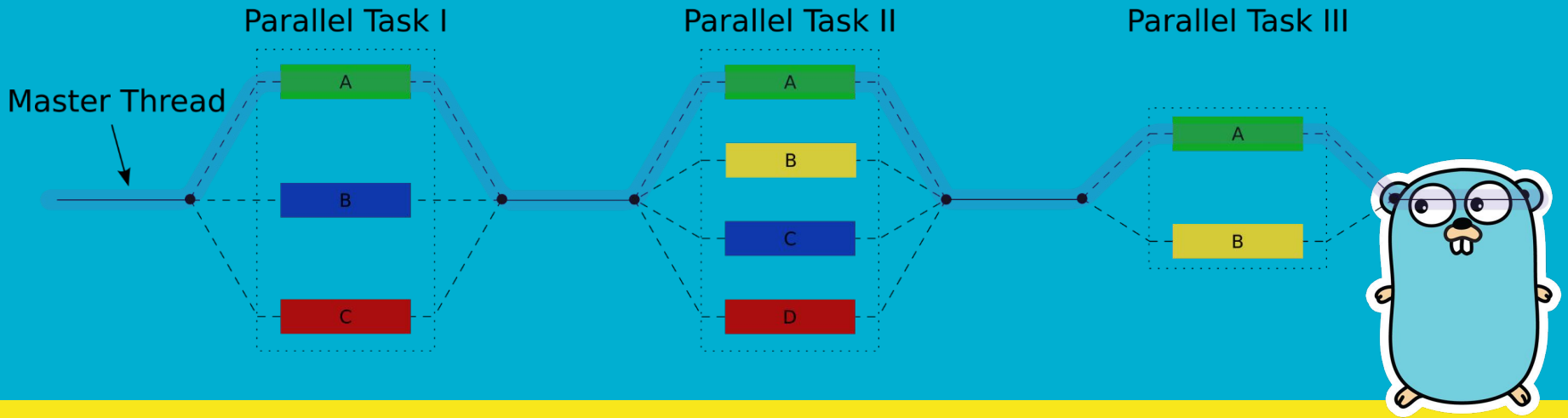

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

func asyncSendMail(number int, wg *sync.WaitGroup) {
    defer wg.Done()
    // delay few seconds
    time.Sleep(2 * time.Second)
    fmt.Println("Async Send Mail", number)
}

func main() {
    // get cpu time for this function
    start := time.Now()
    wg := sync.WaitGroup{}
    wg.Add(5)
    for i := 0; i < 5; i++ {
        go asyncSendMail(i, &wg)
    }
    wg.Wait()
    // get cpu time for this function
    elapsed := time.Since(start)
    fmt.Println("All Mail Sent", elapsed)
}
```

MADE WITH Codye

fork-join model

Image: from Wiki

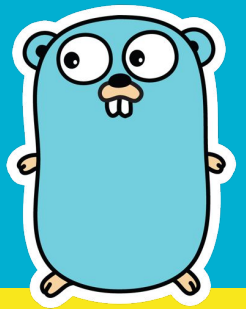




基本資料結構

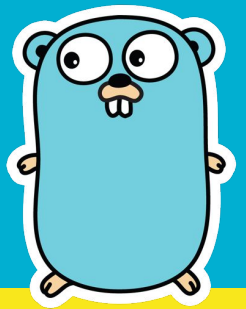
基本資料結構 - Queue

- 佇列 Queue 先加入的元素先取出 (先進先出)
- First in first out (FIFO)



基本資料結構 - Stack

- 堆疊 Stack 後加入的元素先取出 (後進先出)
- Last in first out (LIFO)



在Go實作 Queue/Stack

- Go 沒有內建的Deque 模組來協助實作 Stack 跟 Queue，所以我們只能用第三方套件或內建的 Slice 來手動實現

```go
package main

import (
    "errors"
    "fmt"
)

type stack []int

func (s *stack) IsEmpty() bool {
    return len(*s) == 0
}

func (s *stack) Push(val int) {
    *s = append(*s, val)
}

func (s *stack) Pop() {
    if s.IsEmpty() {
        errors.New("Stack is empty")
    }
    *s = (*s)[:len(*s)-1]
}

func main() {
    myStack := stack{}
    myStack.Push(1)
    myStack.Push(2)
    fmt.Println(myStack)
    myStack.Pop()
    fmt.Println(myStack)
    myStack.Pop()
    myStack.Pop()
}
```

Stack

```go
package main

import (
    "errors"
    "fmt"
)

type queue []int

func (s *queue) IsEmpty() bool {
    return len(*s) == 0
}

func (s *queue) Enqueue(val int) {
    *s = append(*s, val)
}

func (s *queue) Dequeue() {
    if s.IsEmpty() {
        errors.New("Queue is empty")
    }
    *s = (*s)[1:len(*s)]
}

func main() {
    myQueue := queue{}
    myQueue.Enqueue(1)
    myQueue.Enqueue(2)
    fmt.Println(myQueue)
    myQueue.Dequeue()
    fmt.Println(myQueue)
    myQueue.Dequeue()
    myQueue.Dequeue()
}
```

Queue

Message Queue

訊息佇列(message queue, mq)是許多分散式系統的關鍵元件，為不同元件提供了一種非同步通訊和協調的方式

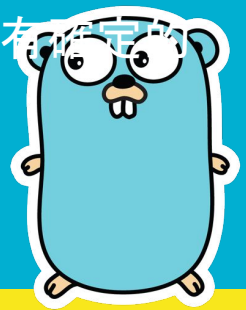
使用 Channel 在 Go 中建立一個簡單的mq

```
type MessageQueue struct {
  queue chan string
}

func (q *MessageQueue) Enqueue(msg string) {
  q.queue <- msg
}

func (q *MessageQueue) Dequeue() string {
  return <-q.queue
}
```

race condition

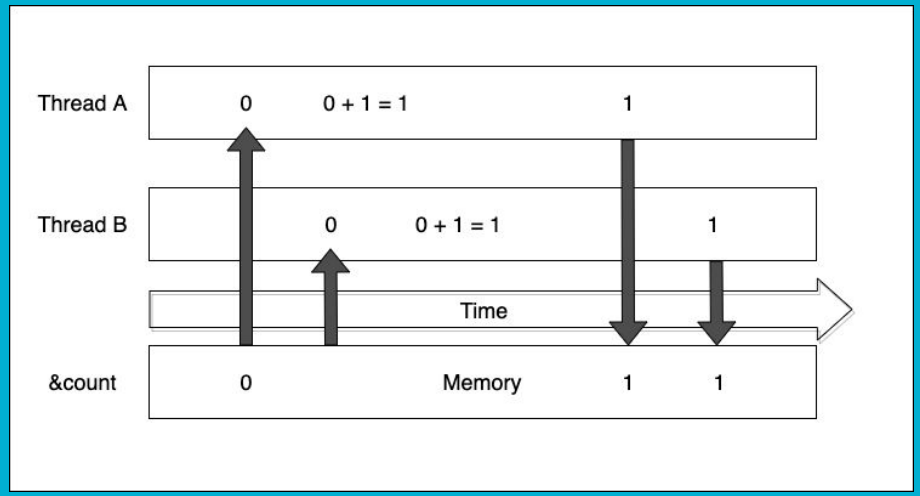
Race Condition

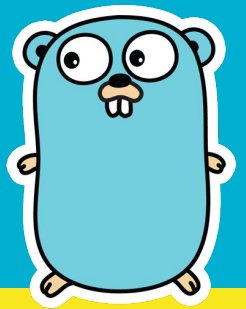
- 因concurrency的交錯執行使執行順序是不確定性的，而執行順序影響到執行結果，這就是競爭危機(race condition)
- A race condition or race hazard is the behavior of an electronics, software, or other system where the output is dependent on the sequence or timing of other uncontrollable events.
- 譬如說有兩個正在進行中的 goroutine 分別要對某變數 a 做 a = a * 2 還有 a = a + 1，這兩個 goroutine 不同的順序有可能導致最後 a 有不同的值，這就是 race condition，為了防止 race condition 要使用一些特別的方式讓他們有確定的順序，以免導致奇怪的 bug



Race Condition

- 如果在使用 goroutine 時沒有考慮到 race condition，那可能就會導致不正確的結果。
- 在Go中可以利用工具race condition detector，來偵測哪邊可能會產生 race condition

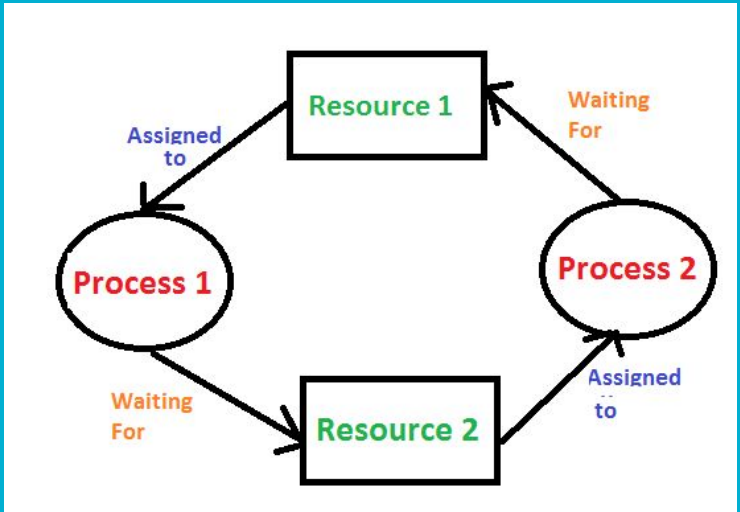




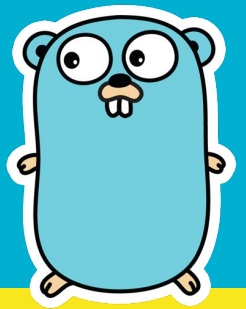
Deadlock

什麼是死鎖?

- 死鎖是指兩個或多個進程/線程在執行過程中,因爭奪資源而造成的一種互相等待的現象
- 如果沒有外力作用,這些進程/線程將無法繼續執行下去



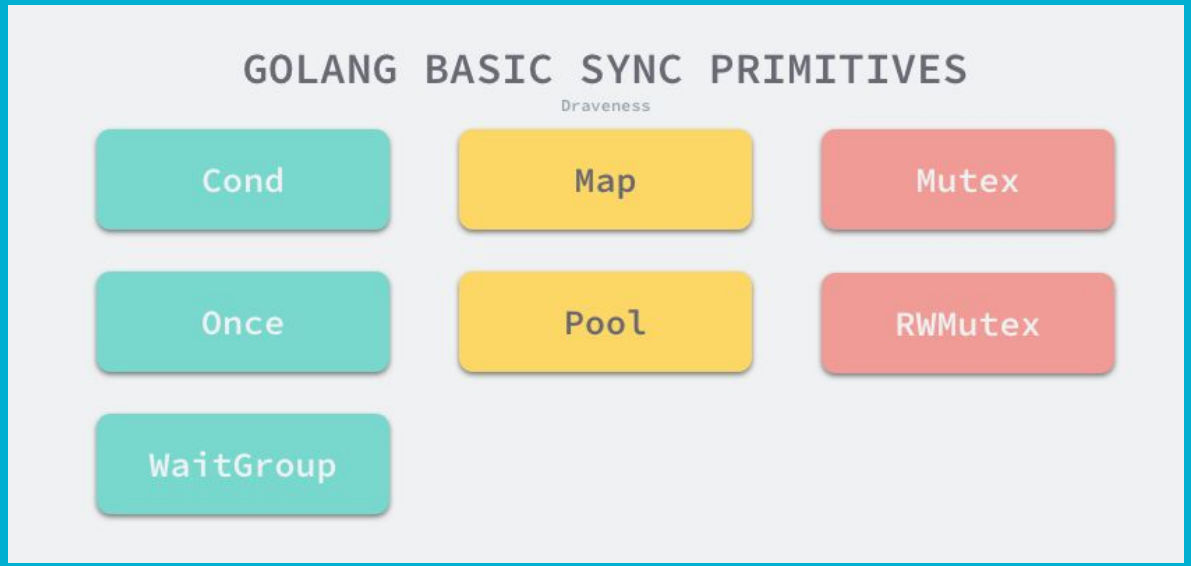
ref: https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/



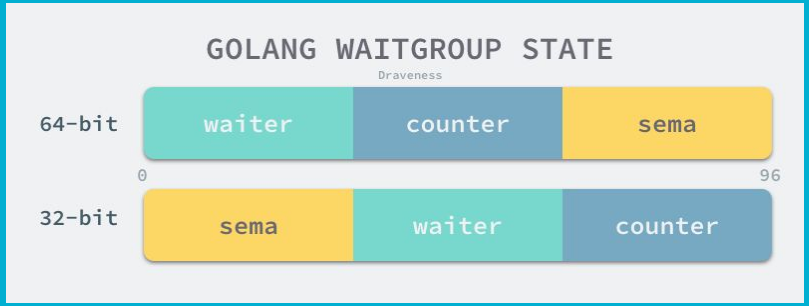
併行語法

Go並行語法

在多執行緒常見的問題，基本上都可以透過 channel, context, sync.WaitGroup, Select, sync.Mutex等方式解決



Wait Group (wg)



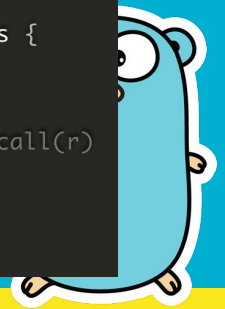
可以強迫 Goroutine 去等待其他 Goroutine

它能夠阻塞主線程的執行，直到所有的goroutine執行完畢。要注意goroutine的執行結果是亂序的，排程器無法保證goroutine執行順序，且進程結束時不會等待goroutine退出。https://ithelp.ithome.com.tw/articles/10226126

sync.WaitGroup 可以等待一組 Goroutine 的回應，

比較常見的使用場景是批次發出 HTTP 请求

```
requests := []*Request{...}
wg := &sync.WaitGroup{}
wg.Add(len(requests))

for _, request := range requests {
    go func(r *Request) {
        defer wg.Done()
        // res, err := service.call(r)
    }(request)
}
wg.Wait()
```



Wait Group範例

wg := sync.WaitGroup{}

wg.Add(<想要同時執行幾個 go func>)

go func(){

    defer wg.Done()

}()

wg.Wait() // 等 done 的次數達到目標

```go
package main

import (
        "fmt"
        "time"
        "sync"
)

func main() {
        fmt.Println("下課休息3秒鐘！")

        wg := sync.WaitGroup{}
        wg.Add(2)

        go rest(&wg)
        go rest(&wg)

        fmt.Println("開始休息")
        wg.Wait()
        fmt.Println("休息完畢準備上課")
}

func rest(wg *sync.WaitGroup) {
        time.Sleep(time.Second * 3)
        fmt.Println("學生休息完畢。")
        wg.Done()
}
```

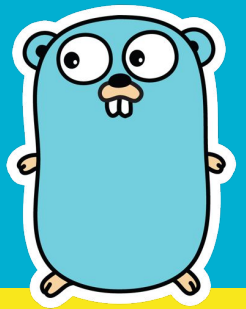
Mutex

1. 透過 mutex.Lock 上鎖,
   不會有別的 goroutine 進入上鎖區域

1. 透過 mutex.UnLock 解鎖, 結束上鎖區域

```
mutex.Lock()
...
會衝突的操作
...
mutex.UnLock()
```

```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    var count int
    var mutex sync.Mutex

    var wg sync.WaitGroup

    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            mutex.Lock()
            count++
            mutex.Unlock()
        }()
    }

    wg.Wait()
    fmt.Println("Final Count:", count)
}
```



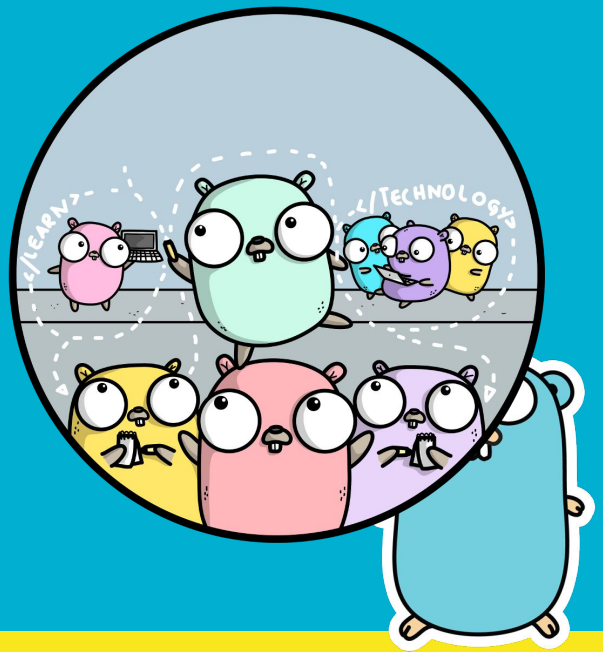
Chan (Channel)

- Goroutine的溝通主要可以透過channel、全域變數進行操作。
- 需要使用某種型別來創建通道，來定義傳送的資料類型
- 使用 make() 創建 Channel
- 用 `<-` 語法來接收或傳輸 item

```go
package main

import "fmt"

func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum // 發送 sum 到 c
}

func main() {
    s := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)
    x, y := <-c, <-c // 從 c 接收

    fmt.Println(x, y, x+y)
}
```

小試身手

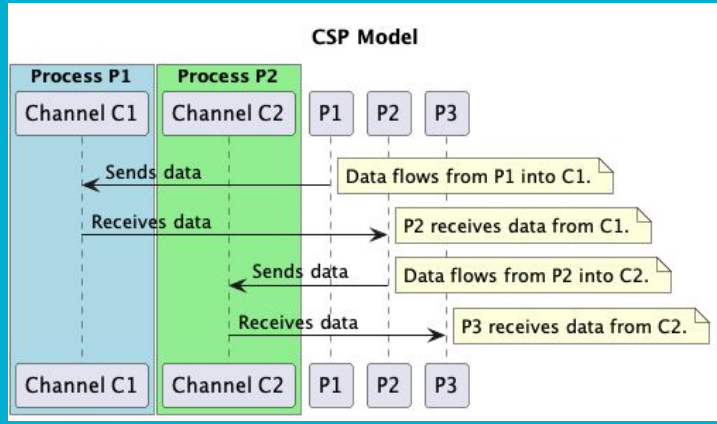
完成Go之旅的-Concurrency！

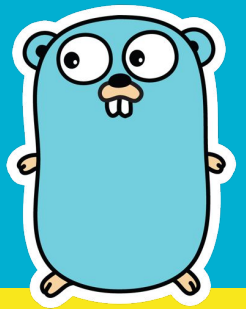


CSP vs CPS

CSP(communicating sequential processes)

- Go語言中的並發程序主要是通過基於CSP（communicating sequential processes）的goroutine和channel來實現。同時，也支持傳統的多線程共享內存的並發方式。
- Go 是第一個將 CSP 的這些思想引入，並且發揚光大的語言。僅管內存同步訪問控制（原文是 memory access synchronization）在某些情況下大有用處，Go 裏也有相應的 sync 包支持，但是這在大型程序很容易出錯。
- 每個東西都要完全獨立，透過訊息溝通來同步！



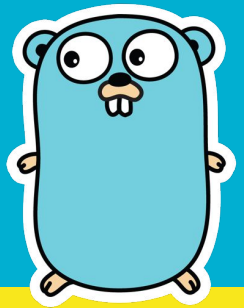


CPS (Continuation-Passing Style)

CPS（Continuation Passing Style）程式設計是一種程式設計風格,它的主要思想是將程式的控制流程**顯式傳遞**給下一個函數,而不是透過函數呼叫堆疊來控制。在 CPS 程式設計中,每個函數都需要一個額外的參數,這個參數被稱為 "continuation",它是一個函數,表示程式執行完當前函數後要繼續執行的程式碼。函數呼叫就變成了一個連續的函數呼叫鏈,每個函數都負責將結果傳遞給下一個函數,從而實現了控制流的顯式傳遞。

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function greet() {
  console.log('Hello');
  await delay(1000);
  console.log('World!');
}

greet();
```

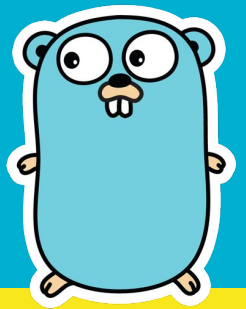
CSP 隱式傳遞 (Go)
vs
CPS 顯示傳遞 (JS)



Go 的同步思維

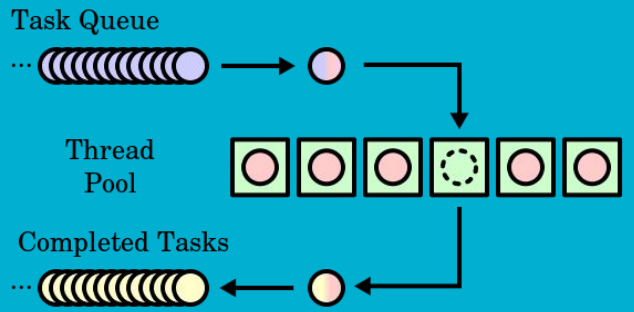
- 要用同步語法描述非同步溝通
- 盡可能用 multi-thread message queue 溝通而避免用記憶體溝通
- 少用記憶體溝通可以減少使用 sync.Wait , sync.Mutex 等語法

- **chan 即為原生的 multi-thread queue**
- **chan 經過 scheduler 優化, 是 lock-free 且 not busy**



thread pool vs event loop

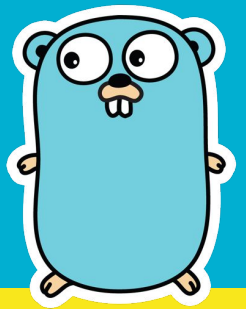
Thread Pool



在大量的執行緒環境中，建立和銷毀執行緒物件的開銷相當可觀，而且頻繁的執行緒物件建立和銷毀，會對高度並行化的應用程式帶來額外的延遲時間

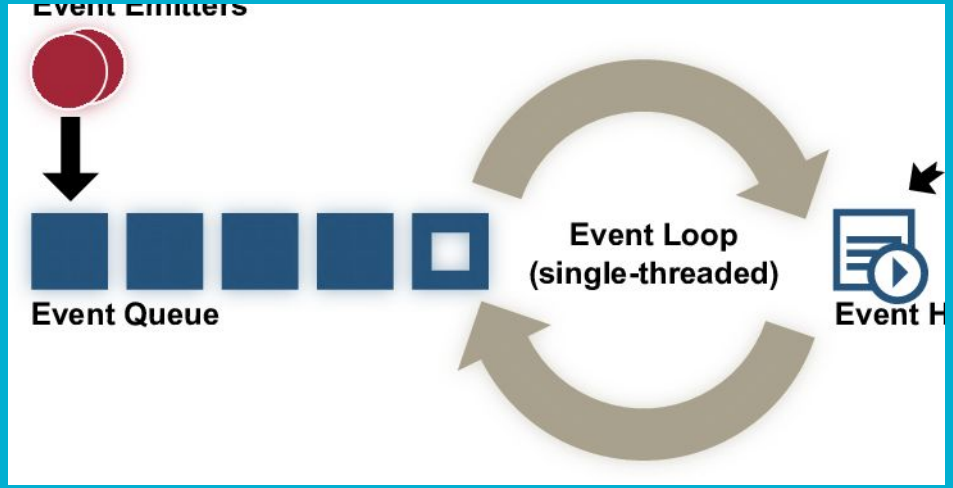
考慮到硬體的有效處理器數量有限，使用 thread pool 可控管執行分配到處理器的執行緒數量

其實thread pool就是producer consumer pattern的一種形式。consumer就是一堆threads，當queue中一有工作進來，一個空閒的thread就會取出來做處理。

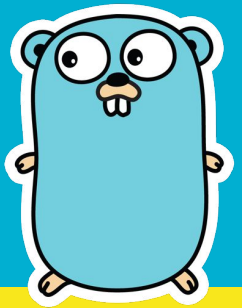


Event Loop

- 基於 event driven
- 持續處理進入的事件
- 如果壅塞先放在 queue

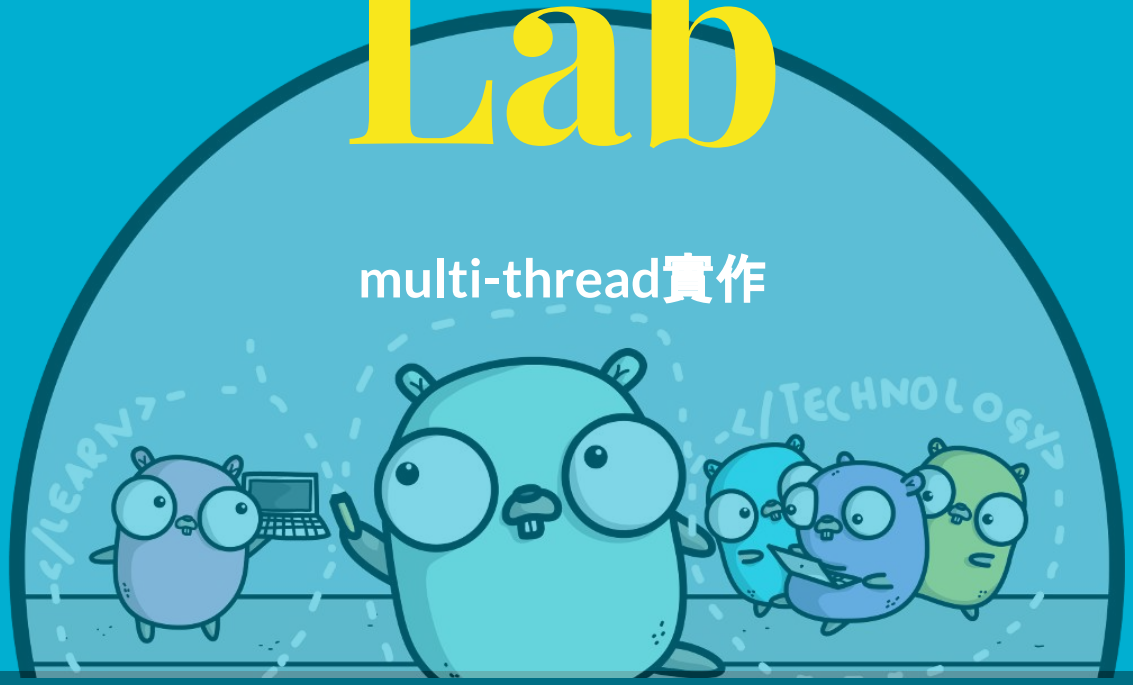


ref:
https://stackoverflow.com/questions/67554089/what-is-the-difference-between-callback-queue-and-event-queue



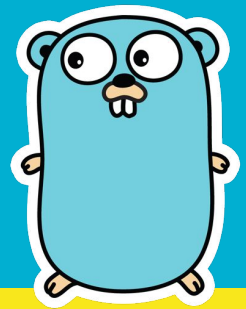
Lab

multi-thread實作

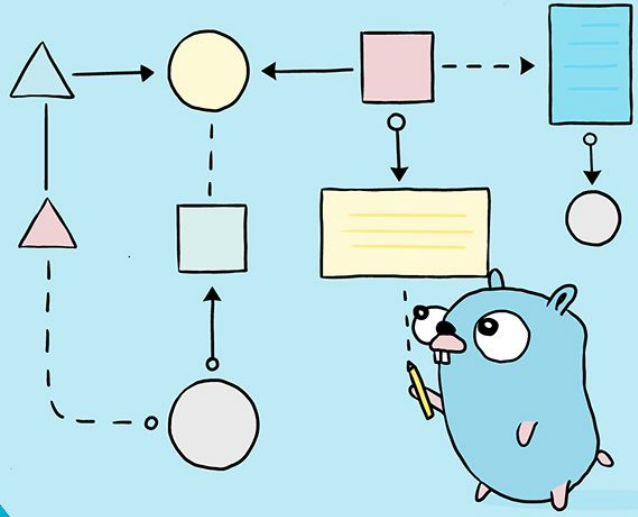


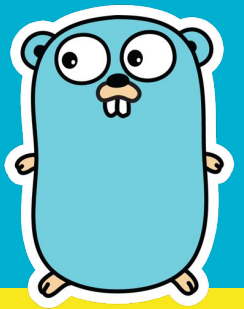
multi-thread stack / queue implement

Link: https://github.com/leon123858/go-tutorial/tree/main/data-structure



Q&A 時間





作業: GUI todolist

Link: https://github.com/leon123858/go-tutorial/tree/main/gui

- 具備 Add todo item
- 具備 clear all item
- 具備 set Done
- 具備 set not done
- 具備 export todoList 成多種格式