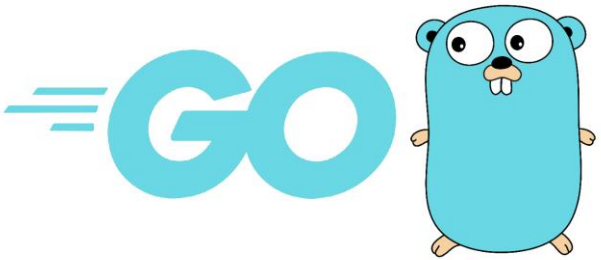
websocket



https://leesonhsu.blogspot.com/2018/07/socketwebsocketsocketio.html

websocket

用 golang 做一個 websocket server 的相關應用。使用的是 gorilla 團隊做的 gorilla/websocket，沒有太多的依賴，非常適合拿來當個應用的基礎 package 。

gorilla/websocket

- WebSocket是一種在單個TCP連接上進行全雙工通訊的協定。WebSocket通訊協定於2011年被IETF定為標準RFC 6455，並由RFC7936補充規範。WebSocket API也被W3C定為標準。
- WebSocket 使得用戶端和伺服器之間的資料交換變得更加簡單，允許伺服器端主動向用戶端推播資料。在WebSocket API中，瀏覽器和伺服器只需要完成一次交握，兩者之間就直接可以建立永續性連接，並進行雙向資料傳輸。
- 可以把它想像成類似 socket 協定的東西，只是 websocket 是基於 http 之上的協定，但兩者最終都還是 TCP 之上的東西。運用這個特性，在 web 上想要實現即時的推播，就不需要使用輪詢等相關技術，減少 http request 相關檔頭肥大的問題。

websocket

gorilla/websocket `go get github.com/gorilla/websocket`

- 運用 gorilla/websocket 寫一個簡單的 client / server 的應用，在開發之前，請記得先安裝，才有辦法執行下面範例，以及對應開發需求。

- **websocket server**

```go
func main() {
    upgrader := &websocket.Upgrader{
        //如果有 cross domain 的需求, 可加入這個, 不檢查 cross domain
        CheckOrigin: func(r *http.Request) bool { return true },
    }
    http.HandleFunc("/echo", func(w http.ResponseWriter, r *http.Request) {
        c, err := upgrader.Upgrade(w, r, nil)
        if err != nil {
            log.Println("upgrade:", err)
            return
        }
        defer func() {
            log.Println("disconnect !!")
            c.Close()
        }()
        for {
            mtype, msg, err := c.ReadMessage()
            if err != nil {
                log.Println("read:", err)
                break
            }
            log.Printf("receive: %s\n", msg)
            err = c.WriteMessage(mtype, msg)
            if err != nil {
                log.Println("write:", err)
                break
            }
        }
    })
    log.Println("server start at :8899")
    log.Fatal(http.ListenAndServe(":8899", nil))
}
```

https://go.dev/play/p/qFTFdp6EeUR

websocket

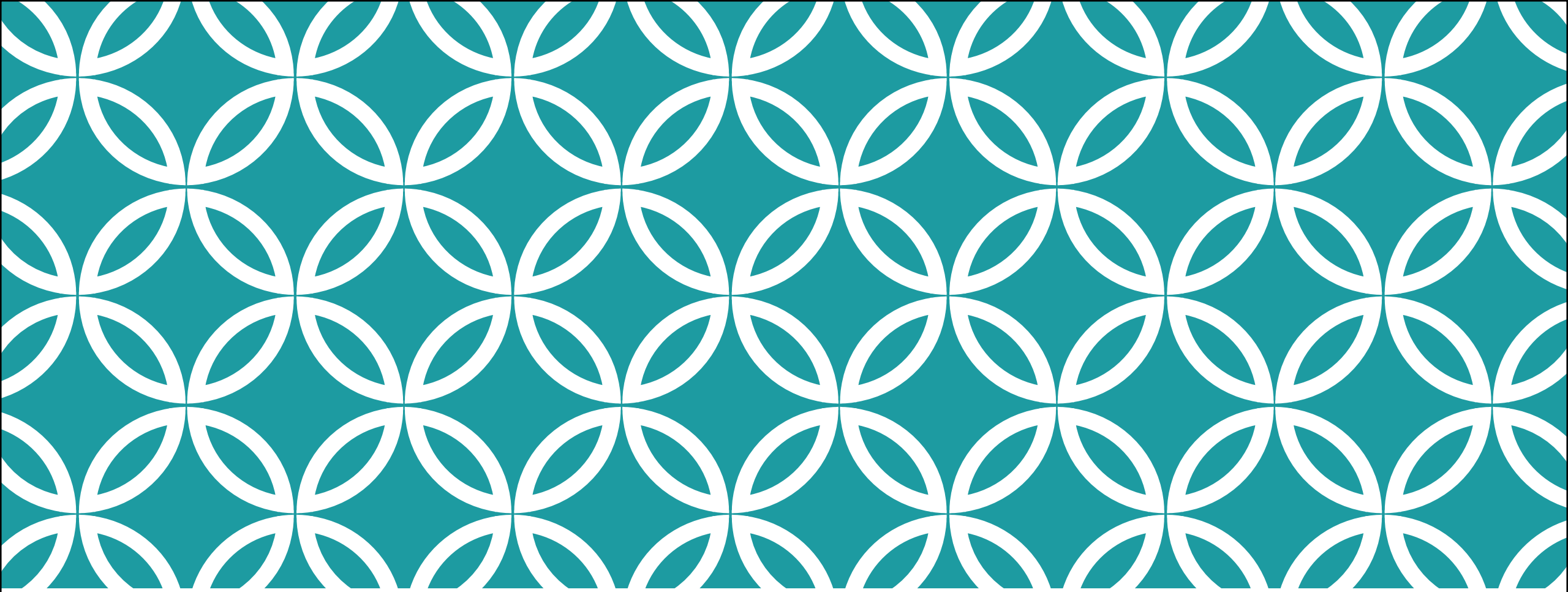
gorilla/websocket `go get github.com/gorilla/websocket`

- **websocket client**

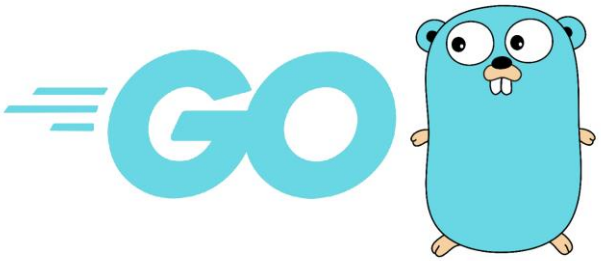
```go
func main() {

    c, _, err := websocket.DefaultDialer.Dial("ws://127.0.0.1:8899/echo", nil)
    if err != nil {
        log.Fatal("dial:", err)
    }
    defer c.Close()

    err = c.WriteMessage(websocket.TextMessage, []byte("hello Alvin"))
    if err != nil {
        log.Println(err)
        return
    }
    _, msg, err := c.ReadMessage()
    if err != nil {
        log.Println("read:", err)
        return
    }
    log.Printf("receive: %s\n", msg)

}
```

接下來只要依序啟動 server & client ，就可以開到 client 對 server 送了 hello Alvin，並且回覆相同字串給 client。。

https://go.dev/play/p/pjsRnnD7zZp



TCP與UDP



TCP與UDP

網際網路可以分為五層架構：

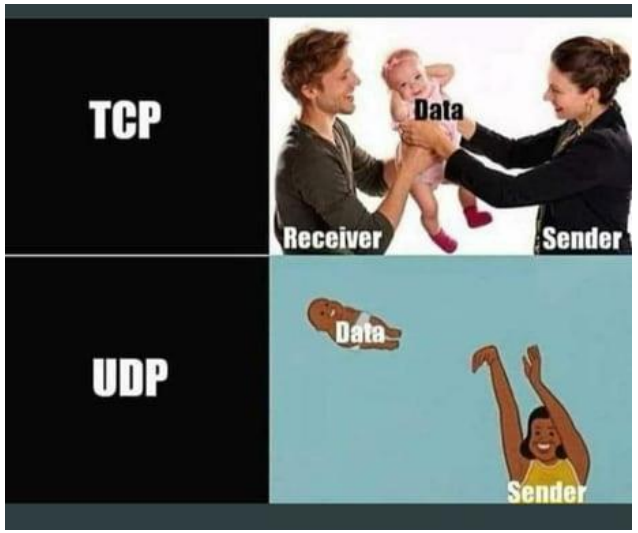
1. Application layer (應用層)
2. Transport layer (傳輸層)
3. Network layer (網路層)
4. Link layer (連接層)
5. Physical layer (物理層)

Application layer指的是網路的應用服務，比如瀏覽網頁所使用的 http(Hypertext transfer protocol, 超文本傳輸協定)和https(http + secure, 超文本傳輸**安全**協定)。只要遵守這些協定後，全世界的人都能正常的使用網頁服務。

TCP與UDP

什麼是 TCP 呢？

- 一提到 TCP (Transmission Control Protocol, 傳輸控制協定) 就一定得提一下 UDP。**TCP 可以保證資料傳輸的正確性、順序性**...等，意思就是說透過 TCP 傳送的資訊(比如 http, https, ftp,...)，並不需要考慮資料有錯誤發生，透過 TCP 傳送 A，那麼接收到的就是 A，先傳送 A 再傳送 B，那麼就是先接收到 A 再接收到 B。
- 而 UDP 則跟 TCP 完全不一樣，有可能透過 UDP 傳送了 A，但接收者就接收到 C，有可能先傳了 A 再傳 B，接收者卻先接收到 B 才接收到 A。**UDP 不但不能保證資料傳輸正確，甚至資料在中途就丟失了，還有可能資料傳輸時順序倒轉的。**



看這一張圖很快就能理解了。因此，只要是注重正確性的網路服務都會透過 TCP 實作，如剛剛提到的網頁 http, https 就算是一個字錯了也不允許，再來還有如郵件、FTP...等。

UDP 好像被講的一無事處，但其實並不然，**因為 TCP 太小心了，所以會有延遲，因為只要有一點錯就會重傳，而且使用前還要花時間建立連線**。UDP 就沒這個問題，他就像圖片底下那個拋嬰兒一樣非常隨便，也不管接收端有沒有收到，這有一個好處，比如講求即時性的應用，如：影片、直播、網路電話，即使錯了幾個畫面也無傷大雅，然而如果是延遲則會很嚴重影響體驗，這時就會優先考慮使用 UDP 實作。

TCP與UDP

Go net

- Go 主攻網路服務，net 套件相當豐富，Go 的官方網站在 net 套件的 Overview 寫了一段：

```
conn, err := net.Dial("tcp", "golang.org:80")
if err != nil {
    // handle error
}
fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
status, err := > bufio.NewReader(conn).ReadString('\n')
// ...
```

- 首先 Dial() 可以理解成撥號，有點像初始化的意思，可以對TCP、UDP這兩個傳輸層的協定進行撥號，也可以對 Network layer 的 IP4、IP6 撥號，那這個 Dial() 要怎麼用：`func Dial(network, address string) (Conn, error)`

TCP與UDP

Go net

```
func Dial(network, address string) (Conn, error)
```

- 其中第一個參數可以擺放連線方式，有的透過 tcp、有的透過 udp，有的則是由 ip 做連線。
- 而第二個參數則是放 IP 地址，或者，可以是一串網址，網址中可以包含 port，至於什麼是 port 呢？假如現在有一台伺服器使用了 140.120.1.20 這個 IP (每次上網時，網路提供商會提供一組 ip，可以要求網路提供商固定這個位址) ，別人可以連到這台伺服器上，但是如果一台伺服器只想要提供不同的服務要怎麼辦呢？這時 port 的概念就出現了。這個伺服器可以：
  - 用 port 443 開啟 https 的伺服器：https://140.120.1.20 = 140.120.1.20:443
  - 用 port 80 開啟 http 伺服器：http://140.120.1.20 = 140.120.1.20:80
  - 用 3306 開啟 MySQL 的資料庫伺服器：140.120.1.20:3306
  - 也可以將 ip 地址註冊成一串有意義的網址：140.120.1.20 = nchu.edu.tw
  - 透過這些 port ，伺服器可以知道接收著想要使用的是哪個服務。

TCP與UDP

使用 TCP 連線，連入 Go 官方網站

- 知道 port 的概念後馬上來試試看利用 tcp 連線來連上 golang.org 這個網站：

```
conn, err := net.Dial("tcp", "golang.org:80")
```

利用這個函式可以透過 tcp 和 golang.org:80 做連線。

回傳的 conn 是 connection 的縮寫，型態為 net.Conn。

而 net.Conn 有什麼方法可以來使用呢？參考文件。

```
執行結果：
HTTP/1.0 200 OK
Date: Tue, 29 Sep 2020 18:26:16 GMT
Expires: -1
...略...
```

```go
package main

import(
    "fmt"
    "net"
)

func main(){
    // 藉由 tcp 對 golang.org 建立連線
    // 採用 port 80 (http)
    // 好比在瀏覽器中打下 http://golang.org
    conn, err := net.Dial("tcp", "golang.org:80")
    if err != nil {
        panic(err)
    }

    // conn 的型態屬於 net.Conn
    // 但 conn 也同時滿足 io.Writer 這個 interface{}

    // func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
    // 利用 fmt.Fprintf 可以對 conn 寫入額外的資訊
    // 這會使 tcp 封包內加入一個請求 http 連線時的資訊
    fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")

    // 發送連線後，golang.org 的伺服器也會反回一個封包
    // conn 同時也滿足 io.Reader 這個 interface{}

    // 利用 .Read() 的方式可以讀取 conn 中的訊息

    // 新建一個 byte slice 長度 64
    // 每次從 conn 緩充 64 個 bytes 下來
    res := make([]byte, 64)
    // num 代表讀入的字數，直到讀到零為止
    for num, _ := conn.Read(res); num!=0; num, _ = conn.Read(res){
        // 將傳來的 []byte 透過 string() 轉型成字串印出
        fmt.Print(string(res))
    }
}
```

這些印出來的資訊就是一個 TCP 封包所含有的資訊，其中最底下的部份是 http 封包的內容。以伺服器端來說，golang.org 的伺服器會先產生一份 http 封包(Application layer)，該封包會被封裝進 TCP (Transport layer)的封包裡，接著又會往下封裝。送出封包後，接收端再一一拆包。透過 net/tcp 套件可以從 ip 的封包中拆出 tcp 封包使用

https://go.dev/play/p/v52dBH3Nwk9

TCP與UDP

開設一個 tcp 伺服器

- 剛剛介紹的方法是利用 Dial 的方式與伺服器做連線。現在希望可以將手上的電腦變成一個可以監聽某個 port 的伺服器，至於要選哪一個 port 可以上維基百科查，因為 port 使用上有潛規則，盡量使用沒有被使用的 port 來練習：**監聽 port: 1450**
- server (伺服器端)
  - 伺服器端的部份以 net.Listen() 來實作，相關的用法請參考 net.Listener – Go：

```go
package main

import(
    "fmt"
    "net"
)

func main(){
    // 新增一個監聽
    // 監聽 tcp port 1450
    ln, err := net.Listen("tcp", ":1450")
    // defer 是一個特殊用法，會延遲函式執行
    // 也就是說 ln.Close() 會在 main() 即將結束時才執行
    defer ln.Close()

    if err != nil {
        panic("監聽 port 1450 失敗")
    }

    // 印出這個是 server (debug 方便)
    fmt.Println("SERVER")

    // server 是不能停止的，必需無時無刻監聽 port: 1450
    for {
        // Accept() 在沒有接到封包時會暫停，直到有接收到才會往下繼續執行
        conn, err := ln.Accept()
        if err != nil {
            fmt.Println("ln.Accept() 失敗")
            continue
        }
        // 處理 conn
        // 因為處理 conn 時會沒辦法繼續監聽，所以要另開一條執行緒來處理，這樣若有其他用戶同時需要伺服器的服務時才不會塞車
        go func (conn net.Conn){
            // 處理完後記得關閉 conn
            // 不然客戶端會不知道訊息傳完了沒
            defer conn.Close()
            req := make([]byte, 64)
            conn.Read(req)

            // 回傳說已接收到並且關閉連線
            fmt.Fprintf(conn, "伺服器端回傳伺服器端已接收到 %s", string(req))

            // 印出接收到的訊息
            fmt.Println("伺服器已接收到", string(req))
        }(conn)
    }
}
```

https://go.dev/play/p/z7eMbXOepD9

TCP與UDP

開設一個 tcp 伺服器

- client (用戶端)

```go
package main

import(
    "fmt"
    "net"
)

func main(){

    fmt.Println("CLIENT")

    // 如果要在相同的裝置上開用戶端，可以使用 localhost，用來代表現在這台主機
    conn, err := net.Dial("tcp", "localhost:1450")
    defer conn.Close()
    if err != nil {
        panic(err)
    }

    //發送訊息
    fmt.Fprintf(conn, "封印解除！")

    //接收伺服器回傳的訊息
    res := make([]byte, 64)
    conn.Read(res)
    fmt.Print(string(res))
}
```

- 執行
  - 執行時先啟動 server，再啟動 client

```
client 執行結果：
CLIENT
伺服器端回傳伺服器端已接收到 封印解除！

server 執行結果：
SERVER
伺服器已接收到 封印解除！

...沒有結束執行...
...因為伺服器終其一生都要等待被客戶觸發...
```

https://go.dev/play/p/CUjrZI2hEOJ

TCP與UDP

前面提到 UDP 雖然無法保證資料正確，但是 UDP 延遲低，在不講求資料正確性的情況下可以使用 UDP 來傳輸資料。

簡單的 UDP

- 伺服器(Server)
  - 利用 net.ListenPacket() 來建立一個監聽，其中，第一個回傳值的型態為 net.PacketConn 可以到官網進一步了解用法：net.PacketConn - Go

```go
package main

import(
    "fmt"
    "net"
)

func main(){
    // 與 tcp 不同, udp 要改用 net.ListenPacket() 來做監聽
    ln, err := net.ListenPacket("udp", ":689")
    defer ln.Close()
    if err != nil {
        panic("監聽 port 689 失敗")
    }
    fmt.Println("SERVER")
    for {
        buf := make([]byte, 1024)
        // ln.ReadFrom 分別回傳三個值
        // buf_len 代表讀入了幾個 byte
        // addr 代表客戶端的位址
        buf_len, addr, err := ln.ReadFrom(buf)

        if err != nil {
            continue
        }

        go func(ln net.PacketConn, addr net.Addr, buf []byte){
            fmt.Printf("用戶端位址: %s\n收到: %s\n", addr, buf)
            ln.WriteTo([]byte("伺服器端已收到資料!\n"), addr)
        }(ln, addr, buf[:buf_len])
    }
}
```

https://go.dev/play/p/adyZg4Qqcdq

TCP與UDP

簡單的 UDP

- 用戶端(Client)

```go
package main

import (
    "fmt"
    "net"
    "time"
)

func main() {
    res, err := sendUDP("localhost:689", "Hello UDP!")
    if err != nil {
        fmt.Println(err.Error())
    }else{
        fmt.Println(res)
    }
}

func sendUDP(addr, msg string) (string, error) {
    conn, _ := net.Dial("udp", addr)

    _, err := conn.Write([]byte(msg))

    bs := make([]byte, 1024)

    // 設定 UDP 連線期限
    conn.SetDeadline(time.Now().Add(3 * time.Second))
    len, err := conn.Read(bs)
    if err != nil {
        return "", err
    }else{
        return string(bs[:len]), err
    }
}
```

https://go.dev/play/p/L8JO_Y-T0vA

TCP與UDP

使用 UDP 傳送圖片

- UDP 傳送封包時有一定限制，建議最大不要超過 512 bytes。
  若是使用 TCP 則不必擔心，TCP 有自己切封包的機制。
- **伺服器(Server) ：**

https://go.dev/play/p/MxVOWI6RFES

```go
package main

import(
    "fmt"
    "net"
    "io"
    "os"
)

func photo(path string, buf_channel chan []byte){
    file, err := os.Open(path)
    defer file.Close()
    if err != nil {
        fmt.Println(err)
        close(buf_channel)
    }

    buf := make([]byte, 512)
    for {
        n, err := file.Read(buf)
        if err != nil && err != io.EOF {
            panic(err)
        }
        if n == 0 {
            break
        }

        buf_channel <- buf
    }
    close(buf_channel)
}

func main(){
    ln, err := net.ListenPacket("udp", ":689")
    defer ln.Close()
    if err != nil {
        panic("監聽 port 689 失敗")
    }
    fmt.Println("SERVER")
    for {
        buf := make([]byte, 512)
        buf_len, addr, err := ln.ReadFrom(buf)

        if err != nil {
            continue
        }

        go func(ln net.PacketConn, addr net.Addr, buf []byte){
            fmt.Printf("用戶端位址: %s\n收到: %s\n", addr, buf)
            // 回傳圖片！
            buf_chan := make(chan []byte, 8)
            go photo("demo.jpg", buf_chan)
            for val := range buf_chan{
                ln.WriteTo(val, addr)
            }
        }(ln, addr, buf[:buf_len])
    }
}
```

TCP與UDP

使用 UDP 傳送圖片

- UDP 傳送封包時有一定限制，建議最大不要超過 512 bytes。
  若是使用 TCP 則不必擔心，TCP 有自己切封包的機制。
- **用戶端(Client)：**
- 運行結果：
  - UDP 在傳送時，會有傳錯的情況發生，但是 jpg, png 都無法容錯。
    於是又試了一些格式：

https://go.dev/play/p/IPb6jboeUGn

```go
package main

import (
    "fmt"
    "net"
    "time"
    "os"
)

func main() {
    res, err := sendUDP("localhost:689", "請求傳送圖片")
    if err != nil {
        fmt.Println(err.Error())
    }else{
        fmt.Println(res)
    }
}

func sendUDP(addr, msg string) (res string, err error) {
    err = nil
    conn, err := net.Dial("udp", addr)
    if err != nil{
        return "", err
    }

    _, err = conn.Write([]byte(msg))

    if err != nil{
        return "", err
    }

    buf := make([]byte, 512)

    conn.SetDeadline(time.Now().Add(5 * time.Second))
    res = ""
    file, err := os.Create("output.mp3")
    if err != nil{
        return "", err
    }

    for n, err := conn.Read(buf); ; n, err = conn.Read(buf){
        if err != nil{
            fmt.Println(err)
            break
        }
        file.Write(buf[:n])
    }

    file.Close()

    return res, nil
}
```