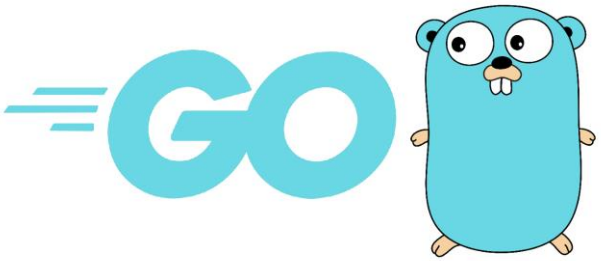
Go with Redis



Go with Redis

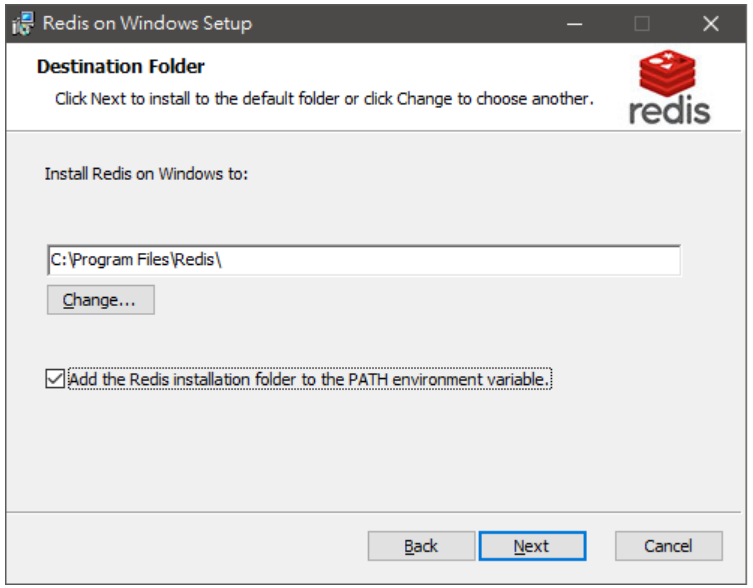
Golang 鍵值資料庫

- 如果想要做快取，資料只是 key-value 的格式，那就無須用到資料庫，可以使用 redis 來達到需求。
- Redis是**Re**mote **Di**ctionary **S**erver簡寫，意思為遠端字典Server。字典Dictionary 即是Key-Value對照表。
- Redis存放資料的速度極快，是物理層面上的快，因為Redis對記憶體(Memory)進行操作。
- 用以下數量級打個比方，會比較有感覺
  - 假如硬碟(Disk、Storage) 的讀寫速度是1、容量10000。那麼...記憶體(Mem、RAM) 的讀寫速度會是100、容量是100，CPU快取(Cache)的讀寫速度是10000、容量是1。
  - 之前提到的MySQL資料庫是對硬碟進行讀寫，Redis將資料存放於記憶體中的特點為：讀寫速度比存放於硬碟快、但容量較小，且資料在斷電就揮發(Volatile)不見了(若沒及時寫進硬碟中的話)。
  - 若有短時間內大量存取的需求，卻又不想如此頻繁的對硬碟讀寫，這種場合可以使用Redis資料庫暫時存放資料，等一段時間後再一齊寫入硬碟裡。

Go with Redis

安裝Redis

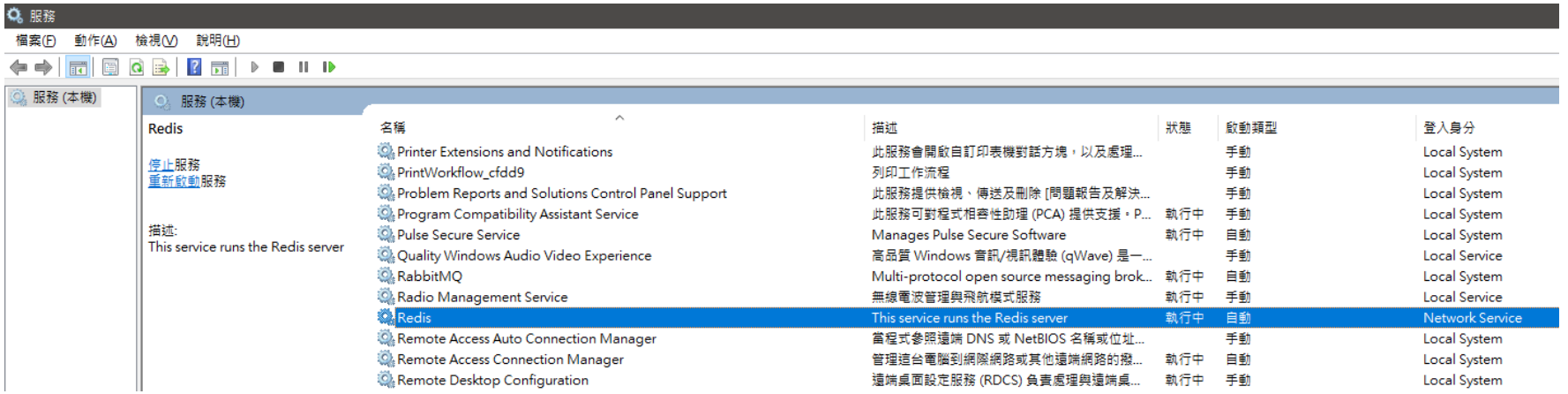
- Redis與MySQL同樣是資料庫，分成Server端與Client端，只是存放資料的形式不同。
- **Windows** 安裝Redis
  - 到Github下載 .msi安裝檔案，接著下一步、**勾選加入PATH環境變數**：



Go with Redis

安裝Redis

- **Windows** 安裝Redis
  - 預設會將Redis Server開啟成一項 Windows服務：



  - Server端有了，再來啟動一個CMD來執行 Client端：

```
$ redis-cli
```

Go with Redis

安裝Redis

- **MacOS** 安裝Redis
  - 用Homebrew來快速方便的安裝：

```
$ brew install redis
```

  - 啟動Redis Server 端：

```
$ redis-server
```

  - 再另外起一個Terminal來執行Client：

```
$ redis-cli
```

Go with Redis

基本指令：查詢

- **Keys** 查詢指令
  - 是不是忘記自己下過哪些變數名稱了、怕怕的？可利用KEYS指令，透過 pattern(正規表達式) 來查詢。

```
> KEYS *
```

  - Keys指令的時間複雜度為O(n)，會佔用Redis單線程、卡到後面讀寫指令，在程式開發、測試時可以使用，但在大型正式運行的伺服器中千萬別使用KEYS相關指令，查詢時若大批的資料湧進會造成堵塞、炸裂。建議改用SCAN指令來查詢。

Go with Redis

基本指令：查詢

- **Scan** 查詢指令
  - 在當前的Redis資料庫中進行迭代。一開始先下SCAN 0，之後看回傳的數值，接著繼續SCAN迭代下去。

```
> SCAN 0
```

  - 此時若第一個回傳值為"15“

```
> SCAN 15
```

  - 持續步驟直到第一個回傳值為"0"

Go with Redis

基本指令：查詢

- **Type** 查詢類型
  - 忘記這個Key對應到的是甚麼物件類型了？查詢test的類型

```
> TYPE test
```

  - Redis鍵值資料庫中也有分格式，常用到的格式有：
    - **字串 String**：可存放數字、字串。
    - **列表 List**：有順序的列表，可從列表左端或右端操作、新增刪除。
    - **Hash、HashMap**：可存放 HashMap 或稱Dictionary(Key-Value對照表)。
    - **集合 Set**：存放唯一且不重複的項目。
    - **有序集合和 Sorted-Set**：存放唯一且不重複項目，透過Score分數來**排序**，是個極度有趣的格式。
  - 所有的指令在 Redis官網都有詳細記載。

Go with Redis

基本指令：新增、取得、刪除

- **String 數值、字串**
  - 雖然是字串，但實際上可以放數字。
  - SET 設置
    - 設置 test1的值為123

```
> SET test1 123
```

    - 設置 test1的值為字串

```
> SET test1 "Hello Redis"
```

  - INCR、DECR 遞增減
    - test1加1、test1減1

```
> INCR test1
> DECR test1
```

  - INCRBY、DECRBY 加減法
    - test1加50、test1減50

```
> INCRBY test1 50
> DECRBY test1 50
```

Go with Redis

基本指令：新增、取得、刪除

- **String 數值、字串**
  - StrLen 字串長度
    - 傳回test1字串的長度
    ```
    > STRLEN test1
    ```
  - Append
    - 透過append在字尾新增字串hello，接著傳回字串總長度
    ```
    > APPEND test "hello"
    ```
  - GET 取得
    ```
    > GET test1
    ```

Go with Redis

基本指令：新增、取得、刪除

- **List 列表**
  - 將hello塞入list1之中 `> LPUSH list1 hello`
    - LPUSH：往左側(頭)塞入元素a，若無list1則自動建立一個新的
    - RPUSH：往右側(尾)塞入元素a，若無list1則自動建立一個新的
    - LPUSHX：若無list1則塞入失敗，不會自動建立
    - 一次塞多個值進list1 `> LPUSH list1 b c d a a a`
  - 將list1最左側(頭)的元素POP彈出 `> LPOP list1`
  - 看 list1之中，第0到100的元素(含0含100) `> LRANGE list1 0 100`
    - 若要查看所有元素可使用：LRANGE list1 0 -1。 Index -1 代表最後一個項目

Go with Redis

基本指令：新增、取得、刪除

- **Hash、HashMap**
  - Hash即為**一組Key對應到一組Value**
  - HSET 設 hash1表裡面的 h1鍵 值為123 `> HSET hash1 h1 123`
  - HMSET 設置hashmap(一次設多個HSET) `> HMSET hash1 h2 234 h3 345 h4 456`
  - 取得hash1表中的 h1 的值 `> HGET hash1 h1`
  - 取得hash1表中全部的鍵值 `> HGETALL hash1`

  - 在Redis中 不支援巢狀Hash(hash中不能再設hash)，可以用Serialize或將兩個Hash Table當成 key:value 對應來達成或者使用ReJSON(Redis JSON)，可方便儲存JSON格式。

Go with Redis

基本指令：新增、取得、刪除

- **Set 集合**
  - Set 集合裡面的內容，是**唯一且不重複**的。譬如學生的學號：

```
> SADD set1 "S00001"
> SADD set1 "S00002" "S00003" "S00004" "S00005" ...

> SCARD set1
    返回集合總數量

> SDIFF set1
    返回、列出不同的Set
```

Go with Redis

基本指令：新增、取得、刪除

- **Sorted-Set 有序集合**
  - 有Score分數的集合。可以想像成：考完期中考的學生，各自都會對應到一個分數。分數可重複，學生不能重複：
    - S00001 拿到了99分
    - S00002 拿到了3分
    - S00003 也拿到了3分

```
> ZADD exam 99 "S00001"
> ZADD exam 3 "S00002"
> ZADD exam 3 "S00003"
```

  - 若加上**XX**的話，只會更新已存在的值，**若無鍵則不會新增**。

```
> ZADD exam XX 100 "Teacher"
```

  - 改成**NX**則相反：只新增鍵、不會更新已存在的

Go with Redis

基本指令：新增、取得、刪除

- **Sorted-Set 有序集合**
  - 查看來考試的學生數量 `> ZCARD exam`
  - 查看S00001的分數 `> ZSCORE exam S00001`
  - 有幾個人分數落在0~20分的 `> ZCOUNT exam 0 20`
  - 印出所有人的排名(從低分印到高分) `> ZRANGE exam 0 -1`
  - 印出所有人的排名與**分數**(從低分印到高分) `> ZRANGE exam 0 -1 WITHSCORES`
  - 把 S00001的分數+3分，直接變102分！ `> ZINCRBY exam 3 S00001`

Go with Redis

在Golang中使用Redis

- 一樣可以透過 Golang程式來模擬 Redis Client端的執行。而在golang github社群中有兩個主流的Redis函式庫。**Go-Redis 與 RediGo**。
- 安裝 go-redis
  - go-redis目前主要有**v6版**跟v8版，兩者的語法使用上不相同，這裡介紹v6的版本。
  - go get 全域安裝：
    - V6版本 `$ go get github.com/go-redis/redis`
    - V8版本 `$ go get github.com/go-redis/redis/v8`

Go with Redis

在Golang中使用Redis

- redis連線：
  - 這裡建立連線很簡單，應為在使用 docker 建立 redis 時，已經有 port-forward 到本機的6379 port 了，所以 Addr 填入 localhost:6379：

```
c := redis.NewClient(&redis.Options{
    Addr:     "localhost:6379",
    Password: "",
    DB:       0,
})
```

Go with Redis

在Golang中使用Redis

- Set
  - 這邊先 set 一筆資料，set 就好比 insert ，只要設定好一個 key 以及相對的 value，最後在設定資料過期的時間(不想讓資料過期就設為0)：

```
err := c.Set("user", "JC", 1000).Err() // => SET key value, 1000 代表過期秒數, 若不想設定, 則帶入0, 就不會過期
if err != nil {
    panic(err)
}
```

- Get
  - 再來就是 get 一筆資料，get 就好比 select，帶入自己想要查詢的 key，就會得到相對應的 value：

```
val, err := c.Get("user").Result() // Get value by key
if err != nil {
    panic(err)
}
fmt.Println("user:", val)
```

  - 再來如果帶入不存在的的 key 值，就會直接 panic 掉。

Go with Redis

在Golang中使用Redis

- Del
  - 刪除則是用 Del，可以執行以下程式，在用 Get 來確認是否能取得到值，如果取不到則是已經不存在該 key 值：

```
val, err := c.Del("user").Result() // Del value by key
if err != nil {
    panic(err)
}
fmt.Println("user:", val)
```

Go with Redis

在Golang中使用Redis

- 稍加修改Github上的Quickstart：

```go
package main

import (
    "fmt"
    "github.com/go-redis/redis"
)

func main() {
    c := NewClient()
    test(c)
}
func NewClient() *redis.Client { // 實體化redis.Client 並返回實體的位址
    client := redis.NewClient(&redis.Options{
        Addr:     "localhost:6379",
        Password: "", // no password set
        DB:       0,  // use default DB
    })

    pong, err := client.Ping().Result()
    fmt.Println(pong, err)
    return client
}

func test(c *redis.Client) { // 對該 redis.Client 進行操作
    err := c.Set("key", "value", 0).Err() // => SET key value 0 數字代表過期秒數, 在這裡0為永不過期
    if err != nil {
        panic(err)
    }

    val, err := c.Get("key").Result()   // => GET key
    if err != nil {
        panic(err)
    }
    fmt.Println("key", val)

    val2, err := c.Get("key2").Result() // => GET key2
    if err == redis.Nil {
        fmt.Println("key2 does not exist")
    } else if err != nil {
        panic(err)
    } else {
        fmt.Println("key2", val2)
    }
}
```

https://go.dev/play/p/5h3cUA6bm6x

Go with Redis

在Golang中使用Redis

- 抽獎遊戲製作： https://go.dev/play/p/aSxeLgx36j0
  - 既然Redis能在短時間內快速讀寫，可以透過它並且加上Gin網頁框架，來製作一個迷你遊戲。
  - 每位玩家註冊後預設有1000塊。可以投注任意正整數金額，投越多錢、中獎機率越高。伺服器每分鐘會抽一位幸運者出來，並把這局所有的錢給予這名幸運者。
  - 很適合用Redis中 Sorted-Set 這個類型的Score來計分，當作玩家擁有的錢。
  - 一開始先宣告會用到的物件，"玩家"以及"玩家下注"：

```
type User struct {
    Id      string `json:"Id"`  // 玩家 ID
    Balance int    `json:"balance"` // 玩家餘額
}

type UserBet struct {
    Id     string `json:"Id"`
    Round  int    `json:"round"`    // 局數
    Amount int    `json:"amount"`   // 下注金額
}
```

Go with Redis

在Golang中使用Redis

- 抽獎遊戲製作：
  - 設定一些常數：

```
const (
    RoundSecond    = 60              // 每一局的時間
    DefaultBalance = 1000            // 玩家初始化金額
    UserMember     = "game"          // 儲存所有使用者的Balance      Redis:`Sorted-Set`  SCORE -> USER
    BetThisRound   = "bet_this_round" // 儲存目前局的下注狀況        Redis:`Sorted-Set`  SCORE -> USER
)
```

  - Gin的操作：

```
router.GET("/bet/:user", GetUserBalance) // 玩家註冊（不須密碼，填入帳號即可）`user`區分大小寫
router.GET("/bet/:user/:amount", Bet)    // 玩家對目前的局面進行下注，`amount`金額
```

  - 獲取bet的玩家帳號及金額：

```
user.Id = c.Param("user")
amountStr := c.Param("amount")
```

Go with Redis

在Golang中使用Redis

- 抽獎遊戲製作：
  - go-Redis 操作：
    - 下注前，先對用戶做查詢，查看玩家餘額足不足夠。

```
balance, err := RC.ZScore(UserMember, user.Id).Result() // => ZSCORE `Table` UserID
```

    - 如果成功下注，玩家餘額為目前餘額減去下注金額。
    - 並且用BetThisRound表來記錄此局此玩家的權重(籤數，越高越容易中獎)。

```
RC.ZIncrBy(UserMember, float64(-amount), user.Id)
RC.ZIncrBy(BetThisRound, float64(amount), user.Id)
```

Go with Redis

在Golang中使用Redis

- 抽獎遊戲製作：
  - go-Redis 操作：
    - 查詢BetThisRound獲取此局目前的獎金池。

```
bets, _ := RC.ZRangeWithScores(BetThisRound, 0, -1).Result()
for _, bet := range bets {
    var userBet UserBet
    userBet.Amount = int(bet.Score)
    prizePool += userBet.Amount
}
```

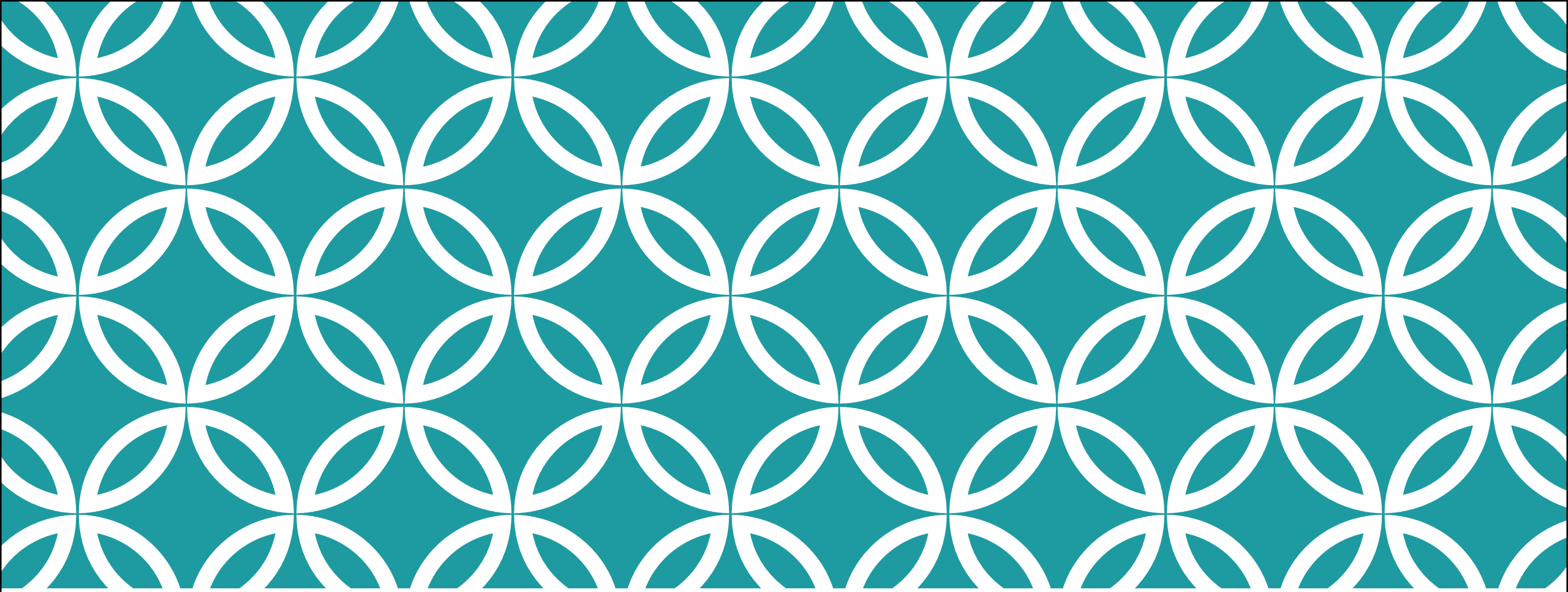
    - 抽出一個幸運兒。

```
winNum := rand.Intn(prizePool + 1)  // 亂數一個幸運號碼
for _, userBet := range userBets {
    winNum -= userBet.Amount
    if winNum <= 0 {
        winner = userBet.Id
        break
    }
}
```

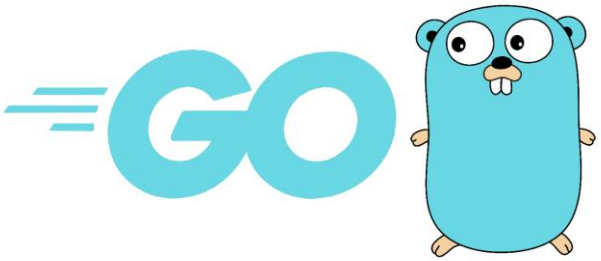
Go with Redis

在Golang中使用Redis

- 抽獎遊戲玩法：
  - 傳回的 Json 數值皆為使用者的餘額。程式執行起來後，開啟多個瀏覽器，每個瀏覽器作為一個獨立的玩家。
  - 註冊Jack帳號：http://127.0.0.1/bet/Jack
  - Jack下注50元：http://127.0.0.1/bet/Jack/50
  - 註冊Timmy帳號：http://127.0.0.1/bet/Timmy
  - Timmy下注333元：http://127.0.0.1/bet/Timmy/333
  - **(接著靜待一分鐘，抽出一名幸運兒。)**
  - 查看餘額Jack餘額：http://127.0.0.1/bet/Jack
  - 查看餘額Timmy餘額：http://127.0.0.1/bet/Timmy



Protobuf



Protobuf

Protobuf是由 Google 開發的一種可跨平台、跨語言的數據交換格式，是一種將結構化資料序列化(變成二進制)的方法。資料比json格式更小更輕便。Protobuf是Protocol Buffers的簡寫，翻成中文為協議緩衝區。

Protocol(協定、協議)：

- 約定、協商好、談妥的東西，或者叫做條款，例如：你給我1000元新台幣，我去幫你到店裡買一包七星菸抽其中一根給你。
- 可以是A、B服務之間傳遞的格式、交換、處理的事情。

Buffer：

- 一塊(特定大小的)空間。

Protocol Buffers：

- 講好的協議所用到的某一塊空間，在資訊世界裡，資料放在空間中就必需要有格式，首先來看看這個格式是什麼。

Protobuf

Protobuf 的格式

- Protobuf 的格式為 proto，目前有 proto2、proto3 兩種版本，兩者有一些差異，主要為 proto3 砍掉了一些較不嚴謹的功能。用法上會以proto3為主。首先創建一個檔案為 school.proto：

```
syntax = "proto3";
option go_package = ".;main";

message Teacher {
    string name = 1;
    int32 age = 2;
}
```

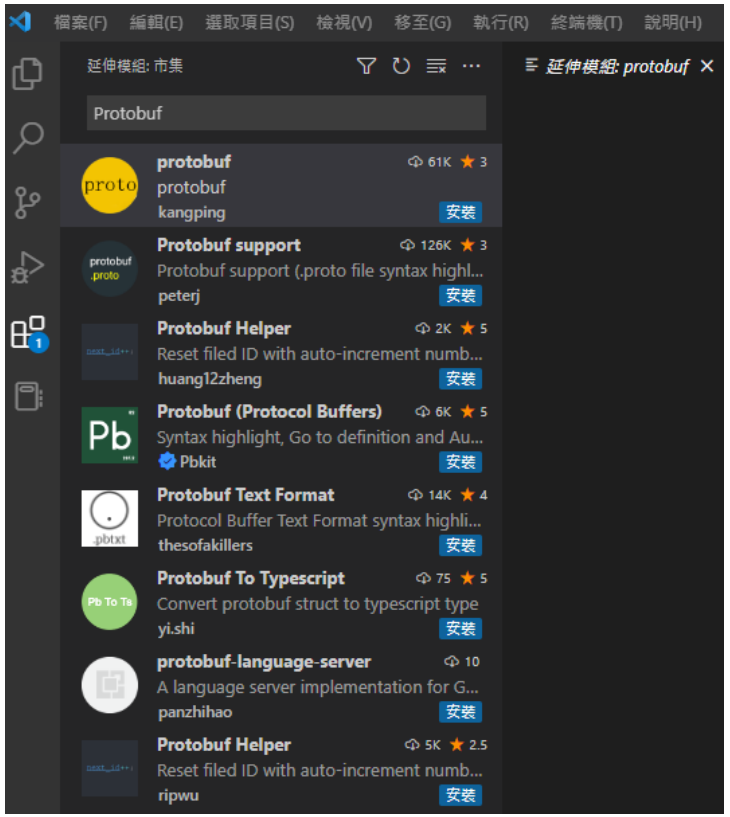
- 數字代表的意義
  - 上方的數字name = 1、age = 2的數字1與2，不是賦值的意思，不是說名字=1、年齡=2，而是編號、唯一識別碼，好讓程式識別這個變數，因為到時大家都被壓縮成二進制，認不出誰是誰，有編號要認人比較方便。
  - 在同一個message裡面識別碼不可重複，但不同message之間重複就沒關係了。這個識別碼編號方式也沒什麼硬性規定，通常會由上往下從1、2、3... 開始依序給。值得一提的是，編號1~15識別碼的區域會使用1 byte來作編碼，位於16~2047 之間的識別碼區域則會用到2 bytes來作編碼。所以可以將較常使用到的欄位盡量都放在1~15的位置，進而減少資料的傳輸量。

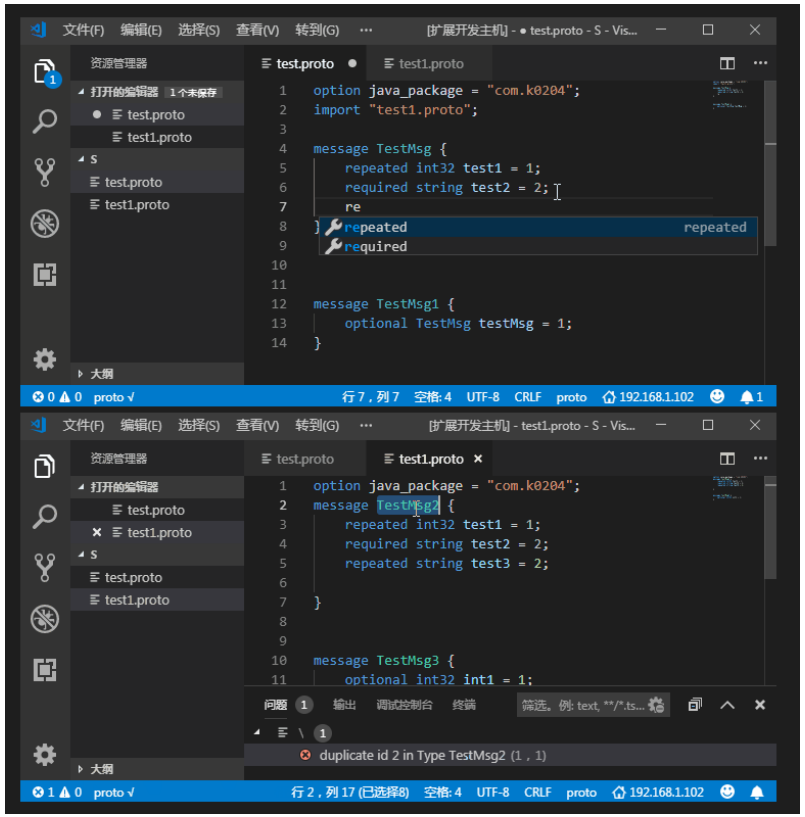
https://go.dev/play/p/rC5ns3F7X5B

Protobuf

vsCode IDE設定：

- 下載能支援proto語法偵測的套件，讓proto語法能夠標記。





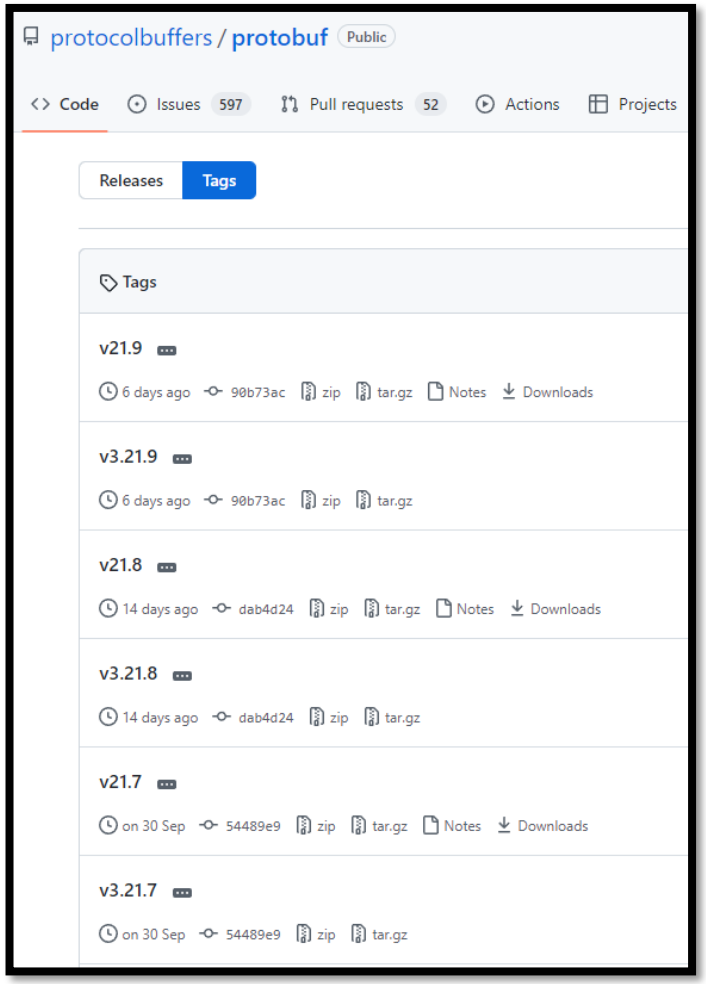
Protobuf

安裝方法

- **透過proto語法介面描述語言(IDL) 來自己定義自己想要的結構，**接著使用Protoc編譯器(Protocol-Buffer-Compiler)來將語法結構變成格式。
- 要在golang上使用proto，首先要下載Protoc編譯器，以及為了能順利**轉出golang的結構物件**，需下載支援產生go語言的插件protoc-gen-go。

安裝protoc

- 首先下載符合自己電腦(Windows、MacOS、Linux)的編譯器。解壓縮後按照readme的步驟安裝，**把bin資料夾底下的東西放到PATH(GoRoot)中，include資料夾底下放的是可選擇安裝的proto插件，建議也一併放入到位。**



https://github.com/protocolbuffers/protobuf/releases

Protobuf

安裝protoc

- Windows 可以放：

```
C:\Users\USER\go\bin          放bin資料夾底下的東西
C:\Users\USER\go\include      放include資料夾底下的東西
```

或者：

```
C:\Go\bin          放bin資料夾底下的東西
C:\Go\include      放include資料夾底下的東西
```

- 也可以直接放入：

```
C:\WINDOWS\System32
```

- MacOS 可以放在：

```
/usr/local/bin          放bin資料夾底下的東西
/usr/local/include      放include資料夾底下的東西
```

安裝protoc-gen-go plugin

```
$ go get -u github.com/golang/protobuf/protoc-gen-go
```

產生結構物件 - pb.go檔案

```
$ protoc --go_out=:. school.proto
```

- 下完這行指令就能看到資料夾底下有school.pb.go的程式出現了！

Protobuf

安装protoc(問題處理)

```
protobuf默认情况下执行  --go_out是报错的，

'protoc-gen-go' 不是内部或外部命令，也不是可运行的程序
或批处理文件。
--go_out: protoc-gen-go: Plugin failed with status code 1.
```

```
go get -u google.golang.org/protobuf/cmd/protoc-gen-go
go install google.golang.org/protobuf/cmd/protoc-gen-go
```

Protobuf

在程式中使用物件

- 有了school.pb.go，等於是電腦自動完成這物件的結構。

```go
package main

import (
    "fmt"
)

func main() {
    teacher := &Teacher{Name: "Jack", Age: 32}
    fmt.Printf("%+v\n", teacher)

    tName := teacher.GetName()
    tAge := teacher.GetAge()
    fmt.Println(tName, tAge)
}
```

```
go run hello.go school.pb.go
```

```
name:"Jack" age:32
Jack 32
```

- 不過要注意的是，Teacher物件中的Age屬性，格式是int32。可以進去school.pb.go看這支程式裡面所寫的所有用法、func。

https://go.dev/play/p/g6RuLV2ZE4m

Protobuf

嘗試將內容進行序列化：

```go
package main

import (
    "log"

    "github.com/golang/protobuf/proto"
)

func main() {
    s := &Teacher{
        Name: "Jack",
        Age:  32,
    }

    data, err := proto.Marshal(s)
    if err != nil {
        log.Fatal(err)
    }

    log.Println(data)
}
```

```
C:\Users\teacher\go\src\hello>go run hello.go school.pb.go
2023/05/25 21:19:17 [10 4 74 97 99 107 16 32]
```

https://go.dev/play/p/KEgnF9taa-G

Protobuf

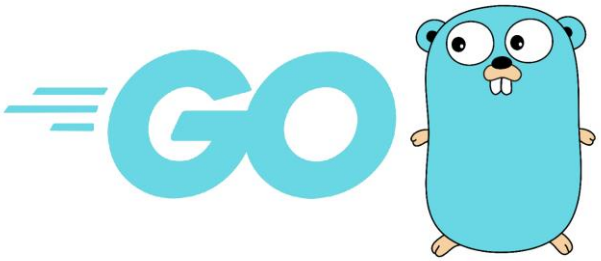
假設接收到以上的資料，也可以把資料再反序列化回來：

```go
package main

import (
    "log"

    "github.com/golang/protobuf/proto"
)

func main() {
    s := &Teacher{
        Name: "Jack",
        Age:  32,
    }

    data, err := proto.Marshal(s)
    if err != nil {
        log.Fatal(err)
    }

    ss := &Teacher{}
    err = proto.Unmarshal(data, ss)
    if err != nil {
        log.Fatal(err)
    }

    log.Println(ss)
}
```

```
C:\Users\teacher\go\src\hello>go run hello.go school.pb.go
2023/05/25 21:23:09 name:"Jack"  age:32
```

https://go.dev/play/p/H06fygkrml8

gRPC



gRPC

gRPC 是使用 Protobuf 來進行序列化協定設計，使得 Server 端和 Client 端在處理序列化更為迅速。

gRPC 也是基於 HTTP/2 設計的， HTTP/2 通訊協定在傳送和接收時都表現的精簡且有效率。

gRPC package

- 在 Go 語言裡要使用 gRPC 前，必須先載入套件：

```
go get -u google.golang.org/grpc
go get -u google.golang.org/grpc/cmd/protoc-gen-go-grpc
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc
```

- 這麼一來，就可以用指令來產生 _grpc.pb.go 檔。

gRPC

Go proto

- 在開發 gRPC 的 API 之前，需要先定義好自己的 Protobuf ，需要開一個以 .proto 為結尾的檔案：

```
syntax = "proto3";

option go_package = ".;student";

package student;

service StudentServer {
   rpc GetStudentData (GetStudentDataReq) returns (GetStudentDataRes) {}
}

message GetStudentDataReq {
   int64  student_id = 1;
   string class = 2;
}

message GetStudentDataRes {
   string student_name = 1;
   int64 student_heigh = 2;
   int64 student_weight = 3;
}
```

需要先在 service 裡頭先定義 api，並且包含 request 和 response，而在定義 request 和 response 裡頭的參數時，有比較嚴格的規定，在參數名稱前，須先定義它的型態，也就是說輸入的型態要符合才可以，而在參數後面會有一個 = <數字>，這是有嚴格規定的，必須按照順序來填，第一個就等於 1，以此類推。這樣就完成了proto檔案了。

- 而剛剛有提到產生 ._grpc.pb.go 檔，就是以 proto 檔來產出，在終端機下指令：

```
protoc --go_out=. --go-grpc_out=require_unimplemented_servers=false:. *.proto
```

https://go.dev/play/p/klhQiqvf0IY

gRPC

gRPC Server

- 使用 gRPC 來建立一套 Server，以及在 Client 端如何去呼叫 Server 端。先建立一個 Server 端，一樣先建立 proto 檔案，可以參考前面：

```
syntax = "proto3";

option go_package = ".;student";

package student;

service StudentServer {
   rpc GetStudentData (GetStudentDataReq) returns (GetStudentDataRes) {}
}

message GetStudentDataReq {
   int64  student_id = 1;
   string class = 2;
}

message GetStudentDataRes {
   string student_name = 1;
   int64 student_heigh = 2;
   int64 student_weight = 3;
}
```

https://go.dev/play/p/klhQiqvf0IY

gRPC

gRPC Server

- 要啟動的服務，包括 GetStudentData API，接著就要建立 main function，將服務啟動：
- 如此一來，就可以啟動這個 Server。

```go
package main

import (
    "context"
    "fmt"
    "net"
    "os"
    "os/signal"
    "syscall"

    "google.golang.org/grpc"
    "google.golang.org/grpc/reflection"

    "student"
)

type studentServer struct {
}

func main() {
    var (
        err              error
        shutdownObserver = make(chan os.Signal, 1)
    )
    // 設定要監聽的 port
    lis, err := net.Listen("tcp", ":3010")
    if err != nil {
        panic(err)
    }

    // 使用 gRPC 的 NewServer meethod 來建立 gRPC Server
    grpcServer := grpc.NewServer()
    sv := &studentServer{}
    student.RegisterStudentServerServer(grpcServer, sv)

    // 在 gRPC 伺服器上註冊反射服務。
    reflection.Register(grpcServer)

    go func(gs *grpc.Server, c chan<- os.Signal) {
        err := gs.Serve(lis)

        if err != nil {
            shutdownObserver <- syscall.SIGINT
        }
    }(grpcServer, shutdownObserver)

    signal.Notify(shutdownObserver, syscall.SIGHUP, syscall.SIGINT, syscall.SIGQUIT, syscall.SIGTERM)

    //阻塞直到有信號傳入
    s := <-shutdownObserver
    fmt.Println(`Receive signal:`, s)

    //停止GRPC服務
    grpcServer.GracefulStop()
}

func (s *studentServer) GetStudentData(ctx context.Context, in *student.GetStudentDataReq) (r *student.GetStudentDataRes, err error) {
    fmt.Println(in) //執行結果
    r = &student.GetStudentDataRes{
        StudentName:   "Alvin",
        StudentHeigh:  190,
        StudentWeight: 50,
    }

    return r, err
}
```

https://go.dev/play/p/tTo4ZpGTyBQ

gRPC

gRPC Client

- 在 Client 端，要呼叫服務端，作法很簡單：

```go
package main

import (
    "context"
    "fmt"
    "log"

    "google.golang.org/grpc"

    "student"
)

var client student.StudentServerClient

func main() {

    // 建立連線
    conn, err := grpc.Dial("localhost:3010", grpc.WithInsecure())
    if err != nil {
        fmt.Println("連線失敗：", err)
    }

    // 最後關閉連線
    defer conn.Close()

    // 用 proto 提供的 NewStudentServerClient, 來建立 client
    client = student.NewStudentServerClient(conn)

    GetStudent(1, "A")
}

func GetStudent(studentId int64, class string) {
    res, err := client.GetStudentData(context.Background(),
        &student.GetStudentDataReq{
            StudentId: studentId,
            Class:     class,
        })
    if err != nil {
        log.Fatalf("GetStudentData error: %v", err)
    }
    fmt.Println(res) //執行結果
}
```

- 這樣就可以建立 Client 端連線了，使用 Server 端提供的服務。
- 使用 gRPC，建出一套 Server 和 Client，使用起來不會太過複雜，如果是想以 Go 語言來開發網站，點對點的即時通訊，非常適合於微服務，會帶來非常高的效率。

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