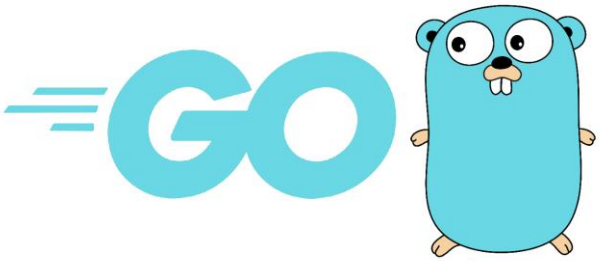
# Gin With Swagger



# Gin With Swagger

API文件

- 每位後端工具師在寫Web API時，一定覺得寫程式還要維護API文件，是件很繁瑣但又重要的工作。
- Swagger是一個能夠快速的生產出一份API Doc的工具。
- Swagger
  - 幾乎主流語言都有支援Swagger，Go也不例外。Swaggo/swag就是用Go撰寫的。
  - swag有支援gin、net/http。

# Gin With Swagger

安裝Swag

```
go get -u github.com/swaggo/swag/cmd/swag
```

```
go install github.com/swaggo/swag/cmd/swag
```

- 檢查swag，會出現版本跟指令，其實也就是init跟help： `swag -h`

初始化swag

- 在專案根目錄下執行： `swag init`
- 會看到在根目錄下產生出了doc資料夾，裡面會有：
  - docs.go
  - swagger.json
  - swagger.yaml

# Gin With Swagger

匯入套件跟註冊路由

- 首先匯入docs套件，mod名稱請改成自己的，修改一下code。

```go
package main

import (
    "github.com/gin-gonic/gin"
    swaggerFiles "github.com/swaggo/files"
    ginSwagger "github.com/swaggo/gin-swagger"

    "hello/hello/docs"
)

// @title Gin Swagger Demo
// @version 1.0
// @description Swagger API.
// @host localhost:8080
func main() {

    router := gin.Default()
    docs.SwaggerInfo.BasePath = "/api/v1"
    router.Group("/demo/v1").
        GET("/hello", hello).
        GET("/hi", hi)

    url := ginSwagger.URL("http://localhost:8080/swagger/doc.json") // The url pointing to API definition
    router.GET("/swagger/*any", ginSwagger.WrapHandler(swaggerFiles.Handler, url))

    router.Run()
}

// @Success 200 {string} string
// @Router /demo/v1/hello [get]
func hello(c *gin.Context) {
    c.JSON(200, "Hello - v1")
}

// @Success 200 {string} string
// @Router /demo/v1/hi [get]
func hi(c *gin.Context) {
    c.JSON(200, "Hi - v1")
}
```

https://go.dev/play/p/cD2IzJ8JVh3

# Gin With Swagger

Swagger 註解

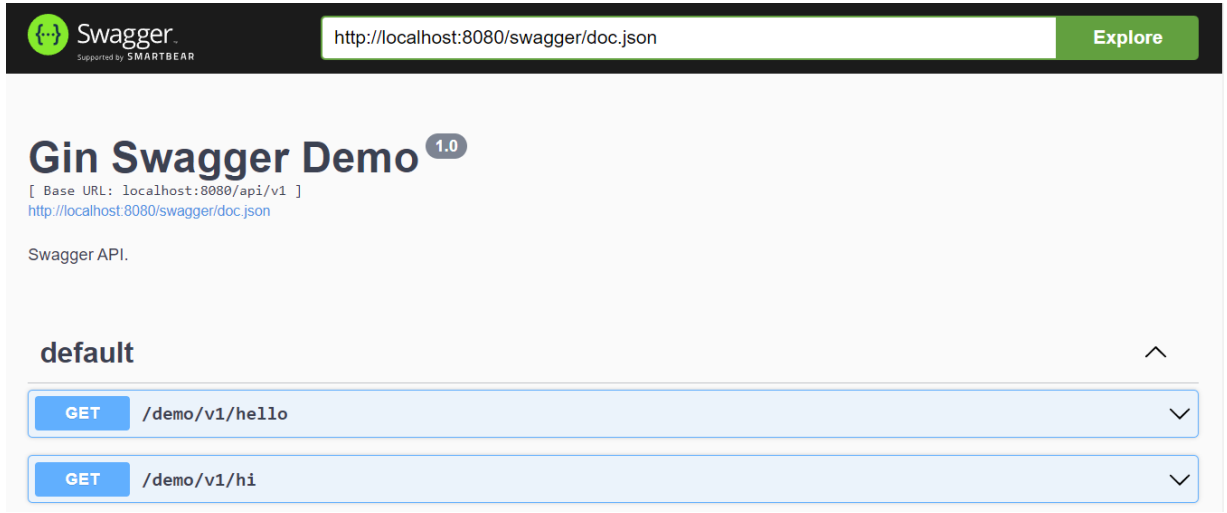
- Swagger能產生文件，其實全靠註解。只要遵從swagger格式寫出來的註解，就會產生對應文件說明。
- 一般說明(main.go)

```
// @title Gin swagger
// @version 1.0
// @description Gin swagger

// @contact.name nathan.lu
// @contact.url https://tedmax100.github.io/

// @license.name Apache 2.0
// @license.url http://www.apache.org/licenses/LICENSE-2.0.html

// @host localhost:8080
// schemes http
func main() { ... }
```

# Gin With Swagger

## Swagger 註解

- API Info
  - 執行生成api doc，並且執行api：`swag init; go run main.go`
  - 打開瀏覽器輸入：http://localhost:8080/swagger/index.html




  - 如果都沒錯誤，看到的就是這樣。

| 註解 | 描述 |
| --- | --- |
| **title** | 必須簡單API專案的標題或主要的業務功能 |
| **version** | 必須目前這專案/API的版本 |
| **description** | 簡單描述 |
| **tersOfService** | 服務條款 |
| **contact.name** | 作者名稱 |
| **contact.url** | 作者blog |
| **contact.email** | 作者email |
| **license.name** | 必須許可證名稱 |
| **license.url** | 許可證網址 |
| **host** | 服務名稱或者是ip |
| **BasePath** | 基本URL路徑, (/api/v1, /v2...) |
| **schemes** | 提供的協定, (http, https) |

# Gin With Swagger

Swagger 註解

- API Operation
  - helloHandler.go

```
// @Summary 說Hello
// @Id 1
// @Tags Hello
// @version 1.0
// @produce text/plain
// @Success 200 string string 成功後返回的值
// @Router /hello [get]
func GetHello(ctx *gin.Context) {...}

// @Summary Delete Hello
// @Id 1
// @Tags Hello
// @version 1.0
// @produce text/plain
// @param id path int true "id"
// @Success 200 string string 成功後返回的值
// @Router /hello/{id} [delete]
func DeleteHello(ctx *gin.Context) { ... }
```

# Gin With Swagger

## Swagger 註解

- API Operation
  - userHandler.go
    - 在這裡新增自定義的Header，通常就是放authorization token用或其他，只要標示是在header即可。

```
// @Summary User Login
// @Tags User
// @version 1.0
// @produce application/json
// @param email formData string true "email"
// @param password formData string true "password"
// @param password-again formData string true "password-again"
// @Success 200 string string 成功後返回的值
// @Router /user/login [post]
func UserLogin(ctx *gin.Context) { ... }

// @Summary Get User Info
// @Tags User
// @version 1.0
// @produce text/plain
// @param Authorization header string true "Authorization"
// @param uid path int true "uid"
// @Success 200 string string 成功後返回的值
// @Router /user/{uid} [get]
func GetUser(ctx *gin.Context) { ... }
```

| 註解 | 描述 |
|---|---|
| **summary** | 描述該API |
| **tags** | 歸屬同一類的API的tag |
| **accept** | request的context-type |
| **produce** | response的context-type |
| **param** | 參數按照 參數名 參數類型 參數的資料類型 是否必須 註解 (中間都要空一格) |
| **header** | response header return code 參數類型 資料類型 註解 |
| **router** | path httpMethod |

# Gin With Swagger

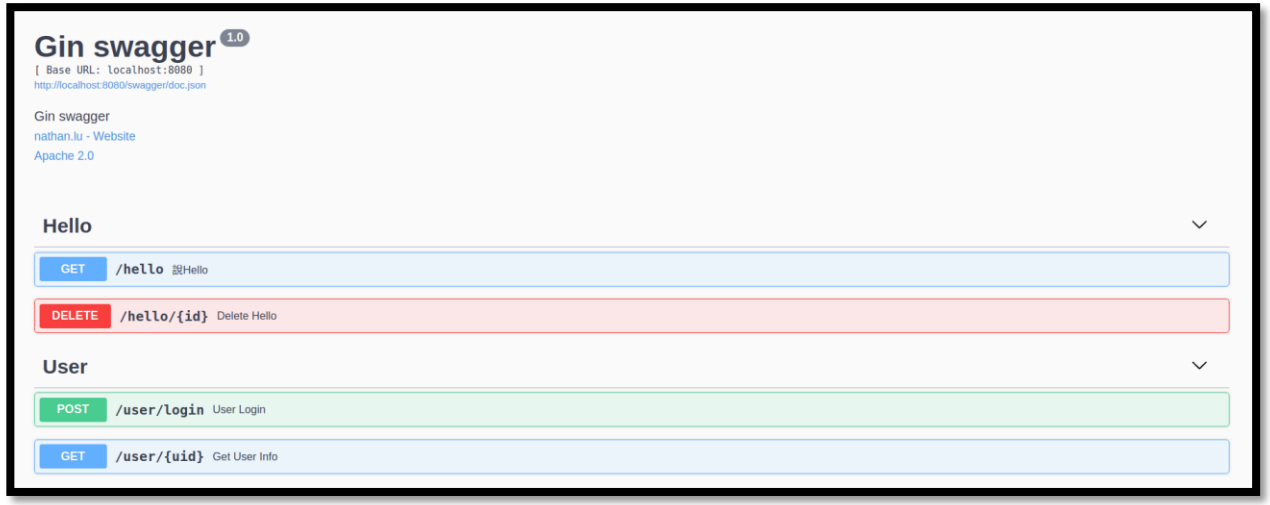
Swagger 註解

- API Operation
  - 參數類型
    - query (接在url後面的query string)
    - path (url內)
    - header (表頭內)
    - body
    - formData
  - Data Type
    - string (string)
    - integer (int, uint, uint32, uint64)
    - number (float32)
    - boolean (bool)
    - user defined struct

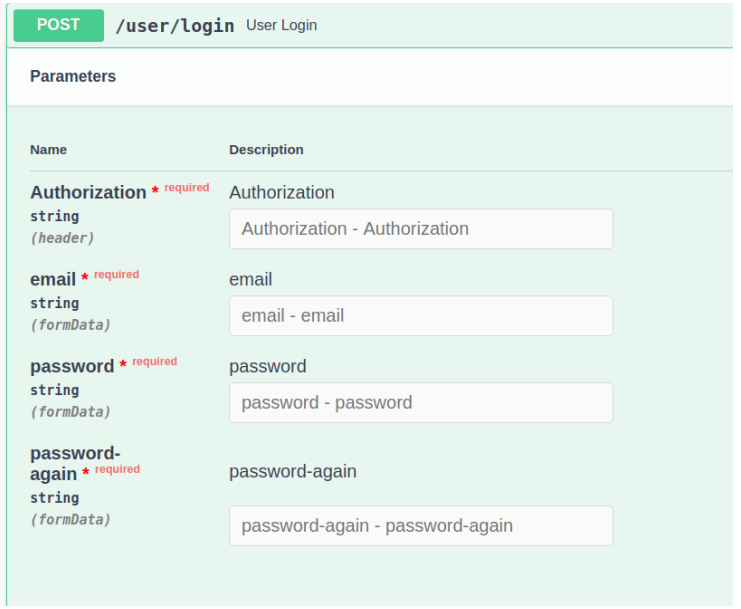
# Gin With Swagger

## Swagger 註解

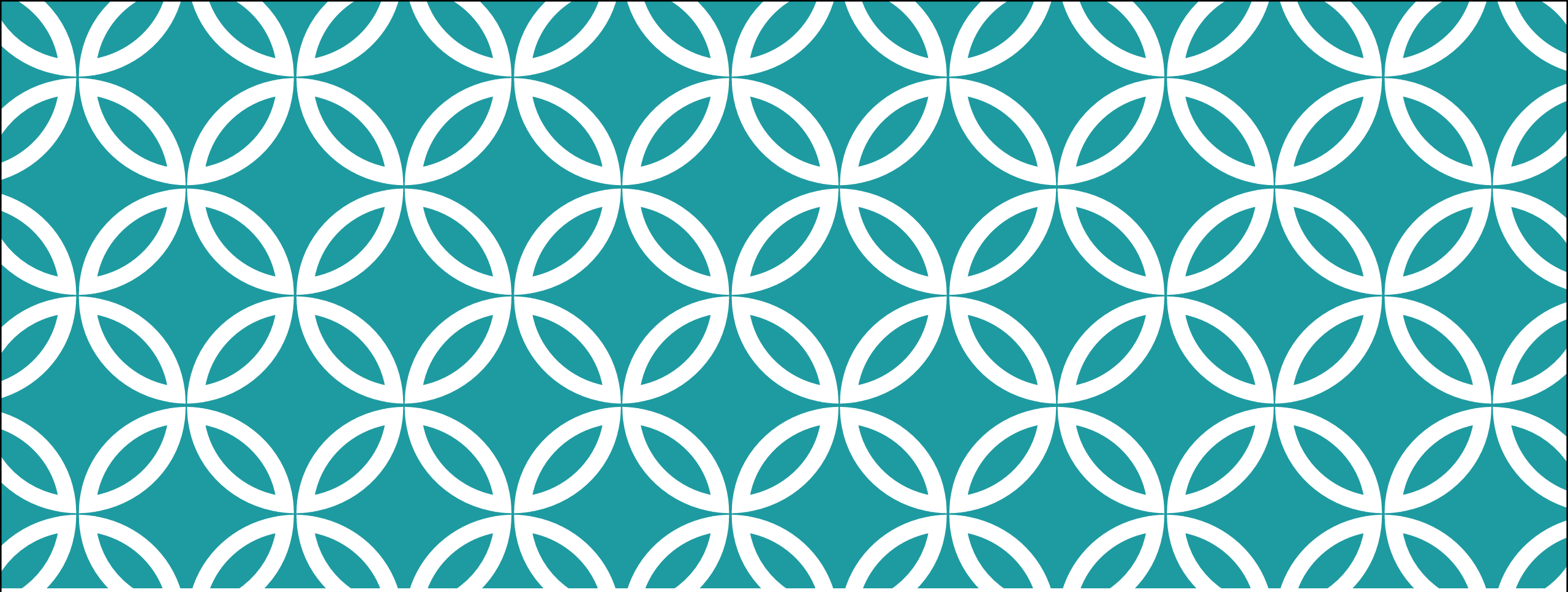
- API Operation
  - 再次執行生成api doc，並且執行api：`swag init; go run main.go`
  - 打開瀏覽器輸入：http://localhost:8080/swagger/index.html








  - 基本的API文件就出來了，但單元測試跟整合測試寫的完整點，比較實在，畢竟文件只能是串接開發時參考用，測試才能一直重複利用，好的測試本身就是個能讀的文件。



# GraphQL | GO

# GraphQL

如果有一個系統，它每天需要處理數十萬甚至數百萬的請求。一般情況下，會請求一個使用資料庫的 API ，該 API 會傳回大量的 JSON Response Body，其中包含了許多冗餘資訊。

此時，如果面對的是一個超大規模的應用程式，那麼傳送、接收這些冗餘資訊會產生很多額外的開銷，並且會由於負載的關係而阻塞頻寬。

事實上，GraphQL 能夠減少因冗餘資訊而產生的額外負擔，並且擁有能夠描述從伺服器端傳回資料的能力，這樣就可以僅關注當前任務，檢視或其他任何東西所需的資料、內容。而且，上述的功能僅僅是 GraphQL 能提供的眾多益處中的很小的一部分。

為 API 而生(而不是為資料庫而生)的查詢語言

- GraphQL 不是一個和傳統 SQL 一樣的查詢語言。它是位於 API 之前的一層抽象，並且不依賴於任何特定的資料或儲存引擎。可以先建立一個與現有服務互動的GraphQL服務，然後圍繞這個 GraphQL 進行構建，而無須擔心會修改現有的 RESTful API。

# GraphQL

REST 和 GraphQL 的區別

- 首先看看 RESTful 方法和 GraphQL 方法的區別。想像下正在構建一個能傳回服務，如果需要某些指定的資訊，通常來說，會建立一個 API 端點以允許以一個 ID 來檢索指定的位置：

```
# A dummy endpoint that takes in an ID path parameter
'http://api.tutorialedge.net/tutorial/:id'
```

- 如果給定的是一個合法的 ID ，它將會傳回一個結構，該結構可能會如下所示：

```
{
    "title": "Go GraphQL Tutorial",
    "Author": "Elliot Forbes",
    "slug": "/golang/go-graphql-beginners-tutorial/",
    "views": 1,
    "key" : "value"
}
```

- 現在，假設想建立一個控制元件，該控制元件會列出指定的作者撰寫的**前 5 個**文章。可以使用 /author/:id 這樣的 API 來檢索出**所有**由該作者撰寫的文章，然後再執行後續的呼叫獲取**排名前 5** 的文章。亦或者，可以建立一個新的 API 來傳回這些資料。上述的解決方案聽起來並沒有什麼特別之處，因為它們建立了大量無用的請求或者傳回了過多的冗餘資訊，這也暴露了RESTful方法的一些缺陷。

# GraphQL

REST 和 GraphQL 的區別

- 此時，就輪到 GraphQL 入場了。透過 GraphQL ，可以在查詢中精確定義想要傳回的資料。因此，如果需要上述的資訊，可以建立一個查詢，如下所示：

```
{
    tutorial(id: 1) {
        id title author {
            name tutorials
        }
        comments {
            body
        }
    }
}
```

- 隨後，它就會傳回該資訊的作者以及指定 id 下該作者所撰寫的其他列表。這些資料都是所需的，但卻不用透過傳送額外的 REST 請求來獲得！
- 目前為止，已經瞭解了 GraphQL 的基礎知識以及使用它的益處，接下來看看如何實際運用。

# GraphQL

Golang的 GraphQL函式庫主要有兩個：

- Graphql-go/**graphql**
  - **Code-First** 模式的函式庫，**無需編寫** GraphQL SDL(結構描述定義語言)，透過 Go 自帶的結構來描述 GraphQL 中的資料類型，由函式庫本身來轉換為 GraphQL Schema。

graph-gophers/**graphql-go**

- **Schema-First** 模式的函式庫，**需要先寫好** GraphQL SDL，然後在 Go 中寫相應的欄位的解析函數。

```go
package main

import (

"fmt"

"github.com/graphql-go/graphql"

)

func main() {

    fmt.Println(&graphql.Schema{})

}
```

```
API server listening at: 127.0.0.1:11559
&{map[] [] <nil> <nil> <nil> map[] map[] []}
Process exiting with code: 0
```

# GraphQL

GraphQL 既是一種用於API的查詢語言也是一個滿足資料查詢的運行時，GraphQL對API中的資料提供了一套易於理解的完整描述，使得客戶端能夠準確地獲得它需要的資料，而且沒有任何冗餘，也讓API更容易隨着時間推移而演進，還能用於構建強大的開發者工具。

基於node的伺服器端開發中，GraphQL技術較爲成熟常用。Golang作爲高性能的現代語言在web服務器開發中也有很高的使用率。根據GraphQL官方程式碼(**graphql-go/graphql**)：(Go/Golang 的 GraphQL 實現)

- 這個函式庫還封裝graphql-go-handler (https://github.com/graphql-go/graphql-go-handler)：
  - 透過 HTTP 請求處理 GraphQL 查詢的中間層。
- 在一切正常的情況下，已經配置完成一個極簡的 GraphQL 服務，並建立一個真實的請求傳送至該服務。

```go
package main

import (
    "encoding/json"
    "fmt"
    "log"

    "github.com/graphql-go/graphql"
)

func main() {
    // Schema
    fields := graphql.Fields{
        "hello": &graphql.Field{
            Type: graphql.String,
            Resolve: func(p graphql.ResolveParams) (interface{}, error) {
                return "world", nil
            },
        },
    }

    rootQuery := graphql.ObjectConfig{Name: "RootQuery", Fields: fields}
    schemaConfig := graphql.SchemaConfig{Query: graphql.NewObject(rootQuery)}
    schema, err := graphql.NewSchema(schemaConfig)
    if err != nil {
        log.Fatalf("failed to create new schema, error: %v", err)
    }

    // Query
    query := `{hello}`
    params := graphql.Params{Schema: schema, RequestString: query}
    r := graphql.Do(params)
    if len(r.Errors) > 0 {
        log.Fatalf("failed to execute graphql operation, errors: %+v", r.Errors)
    }

    rJSON, _ := json.Marshal(r)
    fmt.Printf("%s \n", rJSON) // {"data":{"hello":"world"}}
}
```

https://go.dev/play/p/j5vUZUBHP6v

# GraphQL

GraphQL 模式

- 需要分解上述例子中的程式碼，以便於之後的擴充套件。在 fields... 程式碼行開始處，定義了一個 schema。當透過GraphQL API執行查詢時，**實質上定義了物件中的哪些欄位是期望得到的。所以必須在 schema中定義這些欄位**。
- 在 Resolve... 程式碼行處，**定義了一個解析器函式，每當這個特定 field 被請求時都會觸發這個解析器函式**。到目前為止，僅僅傳回了一個 "world" 字串，**而在此之後，會實現查詢整個資料庫的能力**。

查詢(query)

- 在 query... 程式碼行開始處，定義了一個請求hello欄位的query。接著建立了一個params結構，**該結構包含了對之前定義的schema以及RequestString請求的引用**。最後，開始執行請求，並將請求的結果傳回到 r 中。然後處理可能出現的錯誤，並將結構解析成 JSON ，並將其列印到終端上。

根據範例得知，**在使用時需要有Schema、Query一起解析產生查詢文件對像後，使用查詢器對查詢文件對像進行解析**。

```go
package main

import (
    "encoding/json"
    "fmt"
    "log"

    "github.com/graphql-go/graphql"
)

func main() {
    // Schema
    fields := graphql.Fields{
        "hello": &graphql.Field{
            Type: graphql.String,
            Resolve: func(p graphql.ResolveParams) (interface{}, error) {
                return "world", nil
            },
        },
    }
    rootQuery := graphql.ObjectConfig{Name: "RootQuery", Fields: fields}
    schemaConfig := graphql.SchemaConfig{Query: graphql.NewObject(rootQuery)}
    schema, err := graphql.NewSchema(schemaConfig)
    if err != nil {
        log.Fatalf("failed to create new schema, error: %v", err)
    }

    // Query
    query := `        {            hello        }    `
    params := graphql.Params{Schema: schema, RequestString: query}
    r := graphql.Do(params)
    if len(r.Errors) > 0 {
        log.Fatalf("failed to execute graphql operation, errors: %+v", r.Errors)
    }
    rJSON, _ := json.Marshal(r)
    fmt.Printf("%s \n", rJSON) // {“data”:{“hello”:”world”}}
}
```

https://go.dev/play/p/-grQbV0Xepj

# GraphQL

更為複雜的例子

- 目前為止，已經有一個執行中、極簡的 GraphQL 服務，可以透過它來執行一些查詢。需要更進一步，建構一個更為複雜的例子。會建立一個 GraphQL 服務，它傳回一系列儲存於記憶體中的**教學以及相應的作者、評論**等資訊。首先，需要定義能夠表示 **Tutorial**、**Author**和 **Comment** 的結構：

```go
type Tutorial struct {
    ID       int
    Title    string
    Author   Author
    Comments []Comment
}

type Author struct {
    Name      string
    Tutorials []int
}

type Comment struct {
    Body string
}
```

```go
var tutorials []Tutorial

func populate() []Tutorial {
    author := &Author{Name: "Elliot Forbes", Tutorials: []int{1}}
    tutorial := Tutorial{
        ID:     1,
        Title:  "Go GraphQL Tutorial",
        Author: *author,
        Comments: []Comment{
            Comment{Body: "First Comment"},
        },
    }
    tutorials = append(tutorials, tutorial)
    return tutorials
}
```

- 緊接著，建立一個極簡的 populate 函式用於傳回元素為 Tutorial 型別的切片。右上程式碼將傳回一個簡單的列表，該列表會用於在之後進行的解析操作。

https://go.dev/play/p/a3XXQbvclKn

# GraphQL

更為複雜的例子：**建立一個新的物件型別**

- 首先使用 graphql.NewObject() 在 GraphQL 中建立一個新物件。使用 GraphQL 嚴格的型別來定義三種不同的型別。**這些型別將與已經定義好的三種結構相匹配**。
  - **Comment** 結構無疑是最簡單的，它只包含一個字串型別的欄位 Body ，所以能夠很容易地將其表示為 commentType ：

```
var commentType = graphql.NewObject(
    graphql.ObjectConfig{
        Name: "Comment",
        // we define the name and the fields of our
        // object. In this case, we have one solitary
        // field that is of type string
        Fields: graphql.Fields{
            "body": &graphql.Field{
                Type: graphql.String,
            },
        },
    },
)
```

# GraphQL

更為複雜的例子：**建立一個新的物件型別**

- 首先使用 graphql.NewObject() 在 GraphQL 中建立一個新物件。使用 GraphQL 嚴格的型別來定義三種不同的型別。這些型別將與已經定義好的三種結構相匹配。
  - 接著需要處理 **Author** 結構，並將其定義為新的graphql.NewObject()，這會稍微複雜一點，因為該結構既包含String欄位，也包含一個 Int 值列表，這些值表示該作者所編寫的 ID 列表：

```
var authorType = graphql.NewObject(
    graphql.ObjectConfig{
        Name: "Author",
        Fields: graphql.Fields{
            "Name": &graphql.Field{
                Type: graphql.String,
            },
            "Tutorials": &graphql.Field{
                // we'll use NewList to deal with an array
                // of int values
                Type: graphql.NewList(graphql.Int),
            },
        },
    },
)
```

# GraphQL

更為複雜的例子：**建立一個新的物件型別**

- 首先使用 graphql.NewObject() 在 GraphQL 中建立一個新物件。使用 GraphQL 嚴格的型別來定義三種不同的型別。這些型別將與已經定義好的三種結構相匹配。
  - 最後定義了 **tutorial**結構，它會封裝一個 author、一個元素為 comment 的陣列、ID 以及 title：

```
var tutorialType = graphql.NewObject(
    graphql.ObjectConfig{
        Name: "Tutorial",
        Fields: graphql.Fields{
            "id": &graphql.Field{
                Type: graphql.Int,
            },
            "title": &graphql.Field{
                Type: graphql.String,
            },
            "author": &graphql.Field{
                // here, we specify type as authorType
                // which we've already defined.
                // This is how we handle nested objects
                Type: authorType,
            },
            "comments": &graphql.Field{
                Type: graphql.NewList(commentType),
            },
        },
    },
)
```

# GraphQL

更為複雜的例子：**更新模式**

- 到目前為止，已經定義了一個完整的 Type 系統，接下來，需要更新 Schema 以對映到這些型別上。
- 定義兩個不同的 Field：
  - 第一個：**tutorial 欄位**，該欄位允許根據傳入的 ID 引數檢索單個 tutorial 。
  - 第二個：**list欄位**，它允許檢索儲存於記憶體中的完整 tutorials 列表(list)。

```
// Schema
fields := graphql.Fields{
    "tutorial": &graphql.Field{
        Type: tutorialType,
        // it's Good form to add a description
        // to each field.
        Description: "Get Tutorial By ID",
        // We can define arguments that allow us to
        // pick specific tutorials. In this case
        // we want to be able to specify the ID of the
        // tutorial we want to retrieve
        Args: graphql.FieldConfigArgument{
            "id": &graphql.ArgumentConfig{
                Type: graphql.Int,
            },
        },
        Resolve: func(p graphql.ResolveParams) (interface{}, error) {
            // take in the ID argument
            id, ok := p.Args["id"].(int)
            if ok {
                // Parse our tutorial array for the matching id
                tutorials = populate()
                for _, tutorial := range tutorials {
                    if int(tutorial.ID) == id {
                        // return our tutorial
                        return tutorial, nil
                    }
                }
            }
            return nil, nil
        },
    },
    // this is our `list` endpoint which will return all
    // tutorials available
    "list": &graphql.Field{
        Type:        graphql.NewList(tutorialType),
        Description: "Get Tutorial List",
        Resolve: func(params graphql.ResolveParams) (interface{}, error) {
            tutorials = populate()
            return tutorials, nil
        },
    },
}
```

# GraphQL

更為複雜的例子：**測試**

- 先試新的 GraphQL 服務，使用最新提交的查詢。透過更改 main 函式中的 query 來嘗試 **list 模式**：

```
// Query
query := `
{
    list {
        id
        title
        comments {
            body
        }
        author {
            Name
            Tutorials
        }
    }
}
`
```

- 分析一下，在此查詢中有一個特殊的 root 物件。此時，描述了所期望的物件的 list 欄位。在按照 list 模式傳回的結果列表中，希望能夠看到 id、title、comments和author。當執行這個查詢後，將會看到如下輸出：

```
{"data":{"list":[{"author":{"Name":"Elliot Forbes","Tutorials":[1]},"comments":[{"body":"First Comment"}],"id":1,"title":"Go GraphQL Tutorial"}]}}
```

# GraphQL

更為複雜的例子：**測試**

- 正如所看到的，查詢以 JSON 的格式傳回了所有列表，這看起來和定義的初始查詢非常相似。現在透過 **tutorial 模式**來執行另一個查詢：

```
query := `
{
    tutorial(id:1) {
        title
        author {
            Name
            Tutorials
        }
    }
}
`
```

- 當再一次執行它，會看到它成功地檢索到了記憶體中 ID=1 的資料：

```
{"data":{"tutorial":{"author":{"Name":"Elliot Forbes","Tutorials":[1]},"title":"Go GraphQL Tutorial"}}}
```

- 從輸出結果上看，的 list 和 tutorial 模式能夠正常工作。可以嘗試在 populate 函式中更新列表，使其可以傳回更多的資料。一旦完成了這一步，就可以嘗試使用查詢，並加深對查詢的理解。

# GraphQL

一個簡單的 MySQL 資料庫

- 將記憶體中的資料儲存換成 MySQL 或 MongoDB。GraphQL 的一個很棒的特性是它不受任何特定的資料庫技術集的限制。可以建立一個與 NoSQL 以及 SQL 資料庫交互的 GraphQL API。
- 將使用 SQLite3(本地 SQL 資料庫)來示範如何交換更有效的資料來源。
- 建立新資料庫：
  - 打開一個互動式 shell，使用它來操作和查詢 SQL 資料庫。建立一個資料表(tutorials)開始：

```
sqlite> CREATE TABLE tutorials (id int, title string);
```

  - 然後想在資料庫中插入幾行，以便可以驗證List查詢更改是否有效：

```
sqlite> INSERT INTO tutorials VALUES (1, "First Tutorial");
sqlite> INSERT INTO tutorials VALUES (2, "Second Tutorial");
sqlite> INSERT INTO tutorials VALUES (3, "third Tutorial");
```

# GraphQL

一個簡單的 MySQL 資料庫

- 更新程式碼以連接到這個新資料庫。首先需要在main.go檔案增加一個新的引入，這將允許與SQLite3資料庫進行溝通：

```
package main

import (
    "database/sql"
    "encoding/json"
    "fmt"
    "log"

    "github.com/graphql-go/graphql"
    _ "github.com/mattn/go-sqlite3"
)
...
```

- 現在已經增加了這個新套件，可以嘗試透過**更新list**在main函數中定義的Field來使用它。

https://go.dev/play/p/yGOJwkhe1tl

# GraphQL

一個簡單的 MySQL 資料庫

- 從 GraphQL 的角度來看，只是換掉了傳回記憶體中**教學清單**的程式，現在查詢資料庫並傳回教學清單之前填滿內容：

```
"list": &graphql.Field{
    Type:        graphql.NewList(tutorialType),
    Description: "Get Tutorial List",
    Resolve: func(params graphql.ResolveParams) (interface{}, error) {
        db, err := sql.Open("sqlite3", "./tutorials.db")
        if err != nil {
            log.Fatal(err)
        }
        defer db.Close()
        // perform a db.Query insert
        var tutorials []Tutorial
        results, err := db.Query("SELECT * FROM tutorials")
        if err != nil {
            fmt.Println(err)
        }
        for results.Next() {
            var tutorial Tutorial
            err = results.Scan(&tutorial.ID, &tutorial.Title)
            if err != nil {
                fmt.Println(err)
            }
            log.Println(tutorial)
            tutorials = append(tutorials, tutorial)
        }
        return tutorials, nil
    },
},
```

- 進行這些更改後，可以查詢list，看看這是否有效。

# GraphQL

一個簡單的 MySQL 資料庫

- 更新**query**內部功能，使得它查詢list和檢索id以及title像這樣：

```
// Query
query := `
    {
        list {
            id
            title
        }
    }
`
params := graphql.Params{Schema: schema, RequestString: query}
r := graphql.Do(params)
if len(r.Errors) > 0 {
    log.Fatalf("failed to execute graphql operation, errors: %+v", r.Errors)
}
rJSON, _ := json.Marshal(r)
fmt.Printf("%s \n", rJSON)
```

- 當執行它時，應該看到 sqlite3 資料庫傳回了 3 行，可以看到來自 GraphQL 查詢的 JSON 回應。

```
2021/12/30 14:44:08 {1 First Tutorial { []} []}
2021/12/30 14:44:08 {2 Second Tutorial { []} []}
2021/12/30 14:44:08 {3 third Tutorial { []} []}
{"data":{"list":[{"id":1,"title":"First Tutorial"},{"id":2,"title":"Second Tutorial"},{"id":3,"title":"third Tutorial"}]}}
```

# GraphQL

一個簡單的 MySQL 資料庫：**檢索單筆資料**

- 剛剛替 GraphQL API 使用外部資料來源，但現在們看一個更簡單的範例。
- 更新tutorial模式，以便它現在引用新的 sqlite3 資料來源：

```
"tutorial": &graphql.Field{
    Type:        tutorialType,
    Description: "Get Tutorial By ID",
    Args: graphql.FieldConfigArgument{
        "id": &graphql.ArgumentConfig{
            Type: graphql.Int,
        },
    },
    Resolve: func(p graphql.ResolveParams) (interface{}, error) {
        id, ok := p.Args["id"].(int)
        if ok {
            db, err := sql.Open("sqlite3", "./tutorials.db")
            if err != nil {
                log.Fatal(err)
            }
            defer db.Close()
            var tutorial Tutorial
            err = db.QueryRow("SELECT ID, Title FROM tutorials where ID = ?", id).Scan(&tutorial.ID, &tutorial.Title)
            if err != nil {
                fmt.Println(err)
            }
            return tutorial, nil
        }
        return nil, nil
    },
},
```

# GraphQL

一個簡單的 MySQL 資料庫：**檢索單筆資料**

- 打開一個到現有資料庫的新連接，然後在該資料庫中查詢一筆資料，該資料ID的值等於ID傳入的query：

```
// Query
query := `
    {
        tutorial(id: 1) {
            id
            title
        }
    }
`
```

- 當執行它時，應該看到解析器函數已成功連接到 SQLite 資料庫並檢索到現有資料：

```
{"data":{"tutorial":{"id":1,"title":"First Tutorial"}}}
```

- 已經不再解析和傳回記憶體中的教學清單，而是連接到資料庫進行 SQL 查詢並從中填滿教學列表。在檢索到結果後，GraphQL會處理所有事情。如果想傳回每個教學的作者和評論，可以在SQLite資料庫中建立更多的資料表來儲存。然後，可以簡單地對資料庫執行額外的 SQL 查詢，以根據ID評論檢索作者。

# GraphQL

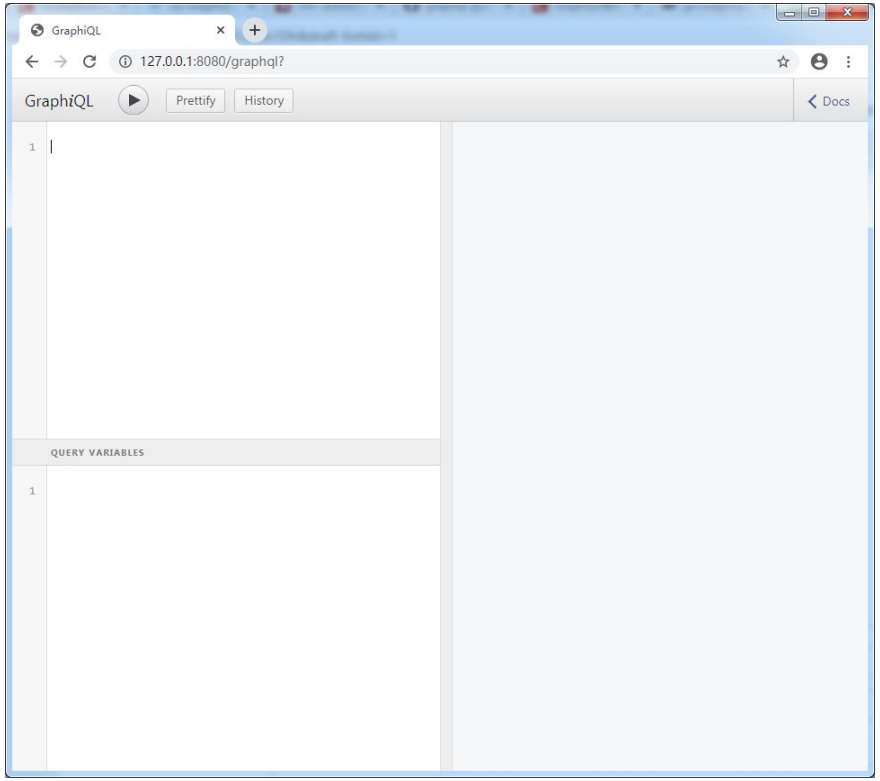
官網上的hello world是控制台輸出的，其實是可以從瀏覽器輸出。很多都是配合gin框架完成。其實官網上有提供handler函式庫，配合net/http就可以了：

```go
package main

import (
    "net/http"

    "github.com/graphql-go/graphql"
    "github.com/graphql-go/handler"
)

// 處理查詢請求
var queryHello = graphql.Field{
    Name:        "QueryHello",
    Description: "Query Hello",
    Type:        graphql.String,
    // Resolve是一個處理請求的函數，具體處理邏輯可在此進行
    Resolve: func(params graphql.ResolveParams) (interface{}, error) {
        return "hello,world", nil
    },
}

// 定義根查詢節點
var rootQuery = graphql.NewObject(graphql.ObjectConfig{
    Name:        "RootQuery",
    Description: "Root Query",
    Fields: graphql.Fields{
        "hello": &queryHello, // 這裡的hello,可以試著改變一下,比如改成test,看看GraphiQL哪裡會有變化
    },
})

// 定義Schema用於http handler處理
var schema, _ = graphql.NewSchema(graphql.SchemaConfig{
    Query:    rootQuery, // 查詢用
    Mutation: nil,       // 需要通過GraphQL做增刪改，可以定義Mutation
})

// main
func main() {
    h := Register()
    http.Handle("/graphql", h)
    http.ListenAndServe(":8080", nil)
}

// 初始化handler
func Register() *handler.Handler {
    h := handler.New(&handler.Config{
        Schema:   &schema,
        Pretty:   true,
        GraphiQL: true,
    })
    return h
}
```

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

# GraphQL

打開瀏覽器，輸入http://127.0.0.1:8080/graphql，就可以看到GraphiQL界面：




輸入{hello}，就可以在右側看到hello world了：