從菜鳥亂來的 CRUD 到 backend framework

by

Triton Ho 

Phase 1

- 這是菜鳥強行寫出來的 CRUD
- 即使沒有用戶身份驗證，也超過了 250 行

前言

- 教材都放到 https://github.com/TritonHo/demo
- 嚴禁問我在 non-Linux 上怎 compile 程式
- 即使你不用別人的 framework ，只要你忍受不了大量重覆性的程式碼，最後你還是會建立一套自己的 framework 的

Phase 1 問題

- 資料庫密碼放在程式碼中
- URL Routing 資訊散亂
- 要自己編寫簡單的 SQL
- 輸入處理跟商業邏輯混在一起
- 輸入檢查散落在 Create 和 Update
- Partial Update 的輸入處理冗長
- 輸出時，大量重覆的程式碼
- Handler 共用 global variable

密碼放在程式碼中

- 現在全宇宙的人都看到密碼
- 而且還永遠留存 ~

```
//the postgresql connection string
connectStr := "host=localhost" +
    " port=5432" +
    " dbname=demo_db" +
    " user=demo_user" +
    " password='user_password'" +
    " sslmode=disable"
```

URL Routing 資訊散亂

- 要知道 /v1/cats/ (GET) 是由誰來負責，要先看 main.go ，再細看 cat.go

```
//http.HandleFunc(pattern string, handler func(ResponseWriter, *Request))
http.HandleFunc(`/v1/cats/`, catHandler)
```

```
var uuidRegexp string = `[[:alnum:]]{8}-[[:alnum:]]{4}-4[[:alnum:]]{3}-[89AaBb][[:alnum:]]{3}-[[:alnum:]]{12}`
var catRegexp *regexp.Regexp = regexp.MustCompile("^/v1/cats/(" + uuidRegexp + ")$")

func catHandler(w http.ResponseWriter, r *http.Request) {
    switch r.Method {
    case "GET":
        if r.URL.Path == `/v1/cats/` {
            catGetAll(w, r)
        } else {
            catGetOne(w, r)
        }
    case "PUT", "PATCH":
        catUpdate(w, r)
    case "POST":
        catCreate(w, r)
    case "DELETE":
        catDelete(w, r)
    }
}
```

```
if catRegexp.MatchString(r.URL.Path) == false {
    //unmatched URL, directly return HTTP 404
    w.WriteHeader(http.StatusNotFound)
    return
}
```

要自己編寫簡單的 SQL

- 如果改動 table schema ，除了要改動 Model Class 之外，還要改動 5 句 SQL 指令

```
//load the object data from the database
rows, err := db.Query("SELECT id, name, gender, create_time, update_time FROM cats order by id desc")
if err != nil {
    w.Header().Set("Content-Type", "application/json; charset=utf-8")
    w.WriteHeader(http.StatusInternalServerError)
    w.Write([]byte(`{"error":"` + err.Error() + `"}`))
    return
}
defer rows.Close()
for rows.Next() {
    var cat model.Cat
    if err := rows.Scan(&cat.Id, &cat.Name, &cat.Gender, &cat.CreateTime, &cat.UpdateTime); err != nil {
        w.Header().Set("Content-Type", "application/json; charset=utf-8")
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte(`{"error":"` + err.Error() + `"}`))
        return
    }
    cats = append(cats, cat)
}
if err := rows.Err(); err != nil {
    w.Header().Set("Content-Type", "application/json; charset=utf-8")
    w.WriteHeader(http.StatusInternalServerError)
    w.Write([]byte(`{"error":"` + err.Error() + `"}`))
    return
}
```

輸入處理跟商業邏輯混在一起

- Phase1 是只支援 json 輸入的
  想想看：如果你的系統有十個 endpoint ，而你的老闆突然說明天前要支持 xml 輸入 ~

```
//bind the input
cat := model.Cat{}
if err := json.NewDecoder(r.Body).Decode(&cat); err != nil {
    w.Header().Set("Content-Type", "application/json; charset=utf-8")
    w.WriteHeader(http.StatusBadRequest)
    w.Write([]byte(`{"error":"` + err.Error() + `"}`))
    return
}
```

Partial Update 的輸入處理冗長

- 我們需要知道用戶正在改動那一個 attribute ，所以要建立一個 structure with pointer attribute 然後還要慢慢建立 SQL 耶（哭）

```
//since we have to know which field is updated, thus we need to use structure with pointer attribute
input := struct {
    Name   *string `json:"name"`
    Gender *string `json:"gender"`
}{}
```

```
//build the SQL for partial update
columnNames := []string{}
values := []interface{}{}
if input.Name != nil {
    columnNames = append(columnNames, `name`)
    values = append(values, input.Name)
}
if input.Gender != nil {
    columnNames = append(columnNames, `gender`)
    values = append(values, input.Gender)
}
colNamePart := ``
for i, name := range columnNames {
    colNamePart = colNamePart + name + ` = $` + strconv.Itoa(i+1) + `, `
}
q := `UPDATE cats SET ` + colNamePart[0:len(colNamePart)-2] + ` WHERE id = $` + strconv.Itoa(len(columnNames)+1)
values = append(values, id)
```

輸入檢查散落在 Create 和 Update

- 一些很簡單的檢查（像 Cat.Gender 一定是 MALE 或 FEMALE ），需要重覆地寫在 Create Handler 和 Update Handler 上

```
//perform basic checking on gender
if cat.Gender != `MALE` && cat.Gender != `FEMALE` {
    w.Header().Set("Content-Type", "application/json; charset=utf-8")
    w.WriteHeader(http.StatusBadRequest)
    w.Write([]byte(`{"error":"Gender must be MALE or FEMALE"}`))
    return
}
```

```
//perform basic checking on gender
if input.Gender != nil && *input.Gender != `MALE` && *input.Gender != `FEMALE` {
    w.Header().Set("Content-Type", "application/json; charset=utf-8")
    w.WriteHeader(http.StatusBadRequest)
    w.Write([]byte(`{"error":"Gender must be MALE or FEMALE"}`))
    return
}
```

輸出時，大量重覆的程式碼

- 很噁心耶 ~
- 而且，如果你作了十個 endpoint ，老闆決定明天追加 xml 輸出，怎麼辦？

```
//perform basic checking on gender
if input.Gender != nil && *input.Gender != `MALE` && *input.Gender != `FEMALE` {
    w.Header().Set("Content-Type", "application/json; charset=utf-8")
    w.WriteHeader(http.StatusBadRequest)
    w.Write([]byte(`{"error":"Gender must be MALE or FEMALE"}`))
    return
}
```

```
//output the result
w.Header().Set("Content-Type", "application/json; charset=utf-8")
if err != nil {
    w.WriteHeader(http.StatusInternalServerError)
    w.Write([]byte(`{"error":"` + err.Error() + `"}`))
} else {
    if affected, _ := result.RowsAffected(); affected == 0 {
        w.WriteHeader(http.StatusNotFound)
    } else {
        w.WriteHeader(http.StatusNoContent)
    }
}
```

Handler 共用 global variable

- 所有的 handler ，都是使用 main.go 的 db 物件
- 想想看：有笨蛋的 stupid.go 中寫了 db = nil
  那會如何 XD~

精簡程式碼之旅 ~

Phase 2

- 引入 mux library ，支援以 regular expression 作 routing
  - 所有 routing information 集中在 main.go ，清楚明快
  - 因為有了 RE ， Handler 不用檢查 URL 是否正確

```
router := mux.NewRouter()
uuidRegexp := `[[:alnum:]]{8}-[[:alnum:]]{4}-4[[:alnum:]]{3}-[89AaBb][[:alnum:]]{3}-[[:alnum:]]{12}`

router.HandleFunc("/v1/cats/", catGetAll).Methods("GET")
router.HandleFunc("/v1/cats/{catId:"+uuidRegexp+"}", catGetOne).Methods("GET")
router.HandleFunc("/v1/cats/{catId:"+uuidRegexp+"}", catUpdate).Methods("PUT")
router.HandleFunc("/v1/cats/{catId:"+uuidRegexp+"}", catDelete).Methods("DELETE")
router.HandleFunc("/v1/cats/", catCreate).Methods("POST")
```

```
func catGetOne(w http.ResponseWriter, r *http.Request) {
    //create the object and get the Id from the URL
    var cat model.Cat
    cat.Id = mux.Vars(r)[`catId`]
```

Phase 3

- 建立了 lib/config ，用來讀取環境變數
- 把密碼和用戶名稱都放到環境變數中
- handler 不再跟 main.go 放於同一 package 之下

```
//the postgresql connection string
connectStr := "host=" + config.GetStr(setting.DB_HOST) +
    " port=" + strconv.Itoa(config.GetInt(setting.DB_PORT)) +
    " dbname=" + config.GetStr(setting.DB_NAME) +
    " user=" + config.GetStr(setting.DB_USERNAME) +
    " password='" + config.GetStr(setting.DB_PASSWORD) + "'" +
    " sslmode=disable"
```

Phase 4

- 80/20 理論
  - ~~80% 的正妹都給 20% 的帥哥把走子~~ ~
  - 我們應該 automate（自動化）80% 的工作，集中精力解決剩下 20% 的工作
  - 我們應該用 20% 氣力去解決 80% 的工作 (easy tasks) ，然後用 80% 氣力去解決餘下 20% 的工作 (hard tasks)

Phase 4（續）

- 使用 struct tag(~= java annotation)，能讓我們告訴 ORM 一些額外資訊
- 因為使用了 ORM 來建立 SQL，之後要改動 table schema 也好，只需要動 Model Class 便行

```
type Cat struct {
    Id string `xorm:"pk" json:"id"`

    Name   string `json:"name"`
    Gender string `json:"gender"`

    CreateTime time.Time `xorm:"created" json:"createTime"`
    UpdateTime time.Time `xorm:"updated" json:"updateTime"`
}

func (c Cat) TableName() string {
    return "cats"
}
```

```
    //create the object slice
    cats := []model.Cat{}

    //load the object data from the database
    err := db.Find(&cats)
```

Phase 5

- 建立 lib/httputil 用作處理輸入

```
func Bind(r *http.Request, obj interface{}) error {
    decoder := json.NewDecoder(r.Body)
    if err := decoder.Decode(obj); err != nil {
        return err
    }

    return nil
}
```

reflection

- 簡單來說，能讓程式在 runtime 時知道一個 structure / object 的資訊
  - 例如有什麼連帶的 methods 和 attributes ，和其屬性
- 一般來說，雖然用了他會讓效能變慢，但是很多 library 都不能不用（特別是 ORM ）
- 除非有特殊原因，一般都不會在 handler 中使用 reflection
- 編寫使用 reflection 的 library 是極度痛苦的；但是完成後 library 的使用者卻極度快樂

Phase 5（續）

- 使用 reflection 來支持 Partial Update ，並且跟 ORM 整合

```go
//dbNameMap =  jsonName --> dbFieldName
//fieldNameMap =  jsonName --> fieldName
func getJsonTagMapping(obj interface{}) (dbNameMap map[string]string, fieldNameMap map[string]string) {
    //assumed a pointer will be passed
    immutable := reflect.ValueOf(obj).Elem()
    immutableType := immutable.Type()
    dbNameMap = map[string]string{}
    fieldNameMap = map[string]string{}

    for i := 0; i < immutable.NumField(); i++ {
        field := immutable.Field(i)
        fieldType := immutableType.Field(i)
        jsonName := getJsonTagName(&fieldType)

        if fieldType.Anonymous {
            //it is anonymous field, simply recursive dive in
            v := field.Addr().Interface()
            subDb, subField := getJsonTagMapping(v)
            for k, v := range subDb {
                dbNameMap[k] = v
            }
            for k, v := range subField {
                fieldNameMap[k] = v
            }
        } else {
            if field.CanSet() && jsonName != `` {
                dbNameMap[jsonName] = GetXormColName(&fieldType)
                fieldNameMap[jsonName] = fieldType.Name
            }
        }
    }
    return dbNameMap, fieldNameMap
}
```

Phase 6

- 使用並且延伸了 "gopkg.in/bluesuncorp/validator.v8"
- 簡單的 single attribute checking ，能放到 struct tag ，然後在 input binding 時檢查

```
type Cat struct {
    Id string `xorm:"pk" json:"id"`

    Name   string `json:"name"`
    Gender string `json:"gender" validate:"required,enum=MALE/FEMALE"`

    CreateTime time.Time `xorm:"created" json:"createTime" validate:"zerotime"`
    UpdateTime time.Time `xorm:"updated" json:"updateTime" validate:"zerotime"`
}
```

```
// bind the http request to a struct. JSON, form, XML are supported
func Bind(r *http.Request, obj interface{}) error {
    decoder := json.NewDecoder(r.Body)
    if err := decoder.Decode(obj); err != nil {
        return err
    }

    //perform basic validation on the input
    return validate.ValidateStructForCreate(obj)
}
```

Phase 7

- 理想的 Handler ，應該只有 business logic
  輸入和輸出都跟他沒關係
  - 只會返回 statusCode ， error / Output object
  - middleware 接受以上資料後，再作 HTTP Response

```go
func CatCreate(r *http.Request, urlValues map[string]string) (int, error, interface{}) {
    //bind the input
    cat := model.Cat{}
    if err := httputil.Bind(r, &cat); err != nil {
        return http.StatusBadRequest, err, nil
    }

    //generate the primary key for the cat
    cat.Id = uuid.NewV4().String()

    //perform the create to the database
    _, err := db.Insert(&cat)

    //output the result
    if err != nil {
        return http.StatusInternalServerError, err, nil
    } else {
        return http.StatusOK, nil, map[string]string{"Id": cat.Id}
    }
}
```

Phase 7（續）

```
func Wrap(f Handler) http.HandlerFunc {
    return func(res http.ResponseWriter, req *http.Request) {
        if statusCode, err, output := f(req, mux.Vars(req)); err == nil {
            SendResponse(res, statusCode, output)
        } else {
            SendResponse(res, statusCode, map[string]string{"error": err.Error()})
        }
    }
}
~
```

```
    router.HandleFunc("/v1/cats/", middleware.Wrap(handler.CatGetAll)).Methods("GET")
    router.HandleFunc("/v1/cats/{catId:"+uuidRegexp+"}", middleware.Wrap(handler.CatGetOne)).Methods("GET")
    router.HandleFunc("/v1/cats/{catId:"+uuidRegexp+"}", middleware.Wrap(handler.CatUpdate)).Methods("PUT")
    router.HandleFunc("/v1/cats/{catId:"+uuidRegexp+"}", middleware.Wrap(handler.CatDelete)).Methods("DELETE")
    router.HandleFunc("/v1/cats/", middleware.Wrap(handler.CatCreate)).Methods("POST")
```

Phase 8

- 直到 Phase 7 ， db 還是 global variable
- handler 真正需要的是 transaction ，不是 db
- 乾脆把 db 丟給 middleware ，然後以 parameter 方式把 transaction 丟給 Handler
- 延伸思考 1 ：不以 parameter ，改用 global map 的方式來丟，如何？
  註：請參看 "github.com/gorilla/context"
- 延伸思考 2 ： transaction 有什麼缺點？

Phase 8（續 1 ）

```
// a middleware to handle user authorization
func Wrap(f Handler) http.HandlerFunc {
    return func(res http.ResponseWriter, req *http.Request) {
        //prepare a database session for the handler
        session := db.NewSession()
        if err := session.Begin(); err != nil {
            SendResponse(res, http.StatusInternalServerError, map[string]string{"error": err.Error()})
            return
        }
        defer session.Close()

        //everything seems fine, goto the business logic handler
        if statusCode, err, output := f(req, mux.Vars(req), session); err == nil {
            //the business logic handler return no error, then try to commit the db session
            if err := session.Commit(); err != nil {
                SendResponse(res, http.StatusInternalServerError, map[string]string{"error": err.Error()})
            } else {
                SendResponse(res, statusCode, output)
            }
        } else {
            session.Rollback()
            SendResponse(res, statusCode, map[string]string{"error": err.Error()})
        }
    }
}
```

Phase 8（續 2 ）

- 即使 Handler 遇上問題，也不用自己來 Rollback 因為在 middleware 那一層解決了

```
func CatCreate(r *http.Request, urlValues map[string]string, session *xorm.Session) (int, error, interface{}) {
    //bind the input
    cat := model.Cat{}
    if err := httputil.Bind(r, &cat); err != nil {
        return http.StatusBadRequest, err, nil
    }

    //generate the primary key for the cat
    cat.Id = uuid.NewV4().String()

    //perform the create to the database
    _, err := session.Insert(&cat)

    //output the result
    if err != nil {
        return http.StatusInternalServerError, err, nil
    } else {
        return http.StatusOK, nil, map[string]string{"Id": cat.Id}
    }
}
```

現在是時候加回 authorization 功能了

Phase 9

- 範例中，我們是採用 jwt
- 除非有特殊原因 auth token 請放在 HTTP Authorization Header
- 這世界有 2 種 endpoint ：需要身份認證的，和不需身份認證的

需要 Auth 的 endpoint wrapper

- 跟之前的版本差不多，增加了對 jwt 的檢查

```go
// a middleware to handle user authorization
func Wrap(f Handler) http.HandlerFunc {
    return func(res http.ResponseWriter, req *http.Request) {
        userId, err := auth.Verify(req.Header.Get("Authorization"))
        if err != nil {
            SendResponse(res, http.StatusUnauthorized, map[string]string{"error": err.Error()})
            return
        } else {
            //please think carefully on this design, as it has potential security problem
            if newToken, err := auth.Sign(userId); err != nil {
                SendResponse(res, http.StatusInternalServerError, map[string]string{"error": err.Error()})
                return
            } else {
                res.Header().Add("Authorization", newToken) // update JWT Token
            }
        }

        //prepare a database session for the handler
        session := db.NewSession()
        if err := session.Begin(); err != nil {
            SendResponse(res, http.StatusInternalServerError, map[string]string{"error": err.Error()})
            return
        }
        defer session.Close()

        //everything seems fine, goto the business logic handler
        if statusCode, err, output := f(req, mux.Vars(req), session, userId); err == nil {
            //the business logic handler return no error, then try to commit the db session
            if err := session.Commit(); err != nil {
                SendResponse(res, http.StatusInternalServerError, map[string]string{"error": err.Error()})
            } else {
                SendResponse(res, statusCode, output)
            }
        } else {
            session.Rollback()
            SendResponse(res, statusCode, map[string]string{"error": err.Error()})
        }
    }
}
```

不需要 Auth 的 endpoint wrapper

- 現在 db 物件是 /lib/middleware 的 global variable ，所以你還是需要 Plain Wrapper
- 你應該視情況自行增加 middleware 種類

```
//do nothing and provide injection of database object only
//normally it is used by public endpoint
func Plain(f PlainHandler) http.HandlerFunc {
    return func(res http.ResponseWriter, req *http.Request) {
        f(res, req, mux.Vars(req), db)
    }
}
~
```

別把 user password 存到資料庫

- 範例中，我們用上了 bcrypt
  postgreSQL 也有 bcrypt extension ，但是建議在 application tier 上運算

```
    if digest, err := bcrypt.GenerateFromPassword([]byte(user.Password), bcrypt.DefaultCost); err != nil {
        middleware.SendResponse(w, http.StatusInternalServerError, map[string]string{"error": err.Error()})
        return
    } else {
        user.PasswordDigest = string(digest)
    }
```

```
user := model.User{}
found, err := db.Where("email = ?", input.Email).Get(&user)
if err != nil {
    middleware.SendResponse(w, http.StatusInternalServerError, map[string]string{"error": err.Error()})
    return
}
if found == false || bcrypt.CompareHashAndPassword([]byte(user.PasswordDigest), []byte(input.Password)) != nil {
    middleware.SendResponse(w, http.StatusUnauthorized, map[string]string{"error": "Incorrect Email / Password"})
    return
}
```

正確的 Model

- Model 只應包括你需要長期保存的 attribute
  如果你的 Input / Output 需要額外 attribute ，請使用 OOP inheritance / golang anonymous field

```
type User struct {
    Id string `xorm:"pk" json:"id" validate:"fixed"`

    Email          string `json:"email" validate:"required,fixed"`
    PasswordDigest string `json:"-"`

    FirstName string `json:"firstName"`
    LastName  string `json:"lastName" validate:"required"`

    CreateTime time.Time `xorm:"created" json:"createTime" validate:"zerotime"`
    UpdateTime time.Time `xorm:"updated" json:"updateTime" validate:"zerotime"`
}
```

```
input := struct {
    model.User       `xorm:"extends"`
    Password         *string `xorm:"-" json:"password" validate:"omitempty,min=1"`
    OriginalPassword *string `xorm:"-" json:"originalPassword" validate:"omitempty,min=1"`
}{}
```

Phase 10

- 如果 n 是程式總行數， Debug 困難度大約是正比於 n^2 – 2^n 之間
  - 別把了看起來酷而把本來數行的邏輯強行用一句來寫
- 所以，每一個 endpoint 都會用上的程式碼，應該要抽出來變成 procedure

Phase 11

- Golang 是獨立 executable ，而沒有像 nginx / tomcat 這樣的 Container
  - 所以，如果 handler 發生 panic（例如不小心的 1 除以 0 ）時，如果沒有任何 Recovery 處理，將會讓整個 executable 當掉
- 所以，我們用別人寫好了的 Recovery middleware

完