測試框架

- 使用 GoCheck 開發測試 - 具備測試基本功能(`Suite` , `before/after Suite/Test`)
- 使用 Ginkgo 開發測試 - 以 BDD 為設計基礎
    1. Test Case 比較清楚
    2. Matcher 內建比較多

同時使用多個 Testing Framework

Testing Framework 的比較

簡介

所有模組都是有狀態的，差別是否透過其它模組儲存狀態

測試範例

測試 Flag: 某此測試由 command flag 決定是否要執行(例連到資料庫的測試)

相依與測試

以下圖為例:

根據 Mathematical Induction，

1. 若 "Db Func 1", "Db Func 2" 測試通過，"Business Service 1", "Business Service 2" 才能測試通過，
2. 若 "Business Service 1", "Business Service 2" 測試通過，"RESTful Service" 才能測試通過。

反之，若 "Db Func 1" 測試不通過，"RESTful Service" 一定不會通過。

See Boolean Algebra

單元測試

一個執行檔的實作大致分三層:

層級	說明	責任	測試重點	實例
API 層	Socket RPC 或 RESTful API(範例)	以 Web MVC framework 來處理主要邏輯 1. 檢查輸入參數是否正確 2. 把輸入參數轉為內部方便使用的資料結構	1. 輸入參數的檢查是否正確? 例: 正規表達式是否正確的測試 2. 輸入資料的處理邏輯是否正確? 例: `23.83` 轉為 2 個 byte 3. 產生的 JSON 是否正確?	見 `common/gin/gin_util_test.go`
處理層	把從各個外部來源(資料庫、其它模組)的資料，在記憶體中計算、合併	1. 把外部資料來源的錯誤，格式化為 API 層可一致顯示的結構 2. 計算、合併 API 層所需要的資料	1. 測試合併的邏輯 2. 測試外部出錯時，產生的錯誤是否有正確反應給 API 層	
資料層	連結資料庫，使用資料庫提供的 DSL (範例)	1. CURD 資料庫	1. 確保各種 DSL 的組合是**正確的** 2. 確保 ORM 的設定是**正確的** 3. 確保 DB Script 是**正確的**	見 `common/db/nqm/agent_test.go`

資料層的嚴謹度最重，因為資料修正成本遠比程式修正成本高

若程式語言有 Java Annotation 或 GoLang Struct Tag
某些測試可以在該資料結構上測試，如電子郵件的格式正確與否。

API 層測試

直接以 RPC Client 或 HTTP Client 連接模組，
直接以「編譯後的執行檔」為測試的標地，同時也可確保執行檔是可用的。

目前都以 `<code>_it_test.go` 放 API 層測試(打執行檔)

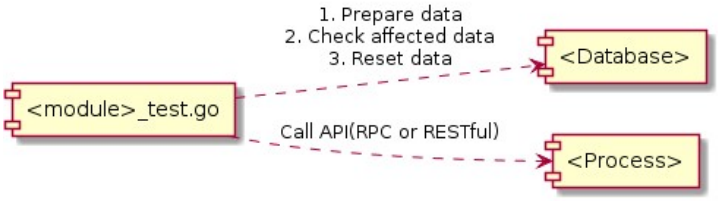
直接連接資料庫的模組

例: **Query**, **HBS**

透過直接寫入與讀取資料庫，也能較容易重設**資料庫狀態**

測試 API 時，測試的 Code(`XXX_test.go`) 會同時連接到執行的 process 與相依的資料庫

1. Process - 測試 API
2. 資料庫 - 便於準備、檢測與重設資料庫狀態



連接其它模組的模組

例: **Agent**, **Light**

測試 API 時，測試的 Code(`XXX_test.go`) 連接到假的相依服務

1. Process - 測試 API
2. Mock Service - 便於準備、模擬各種資料狀態

Mock Service 由資料驅動產生各式所需的狀態，例

- `/api/v1/resource/1` : 產生 `200` ，10 筆資料列表
- `/api/v1/resource/2` : 產生 `400` ，錯誤碼 1 的結果
- `/api/v1/resource/3` : 產生 `400` ，錯誤碼 2 的結果

Guru

- 就算程式碼很混亂，自動化測試也能有很大幫助
- 重構舊程式碼，先寫測試，再重構

文件、原始碼與測試

假設一個人不知道什麼是排序:

文件

告訴你什麼是排序法

原始碼

再怎麼 Clean Code，也不會懂快速排序演算法

- 規格如果有錯(不合邏輯)，程式碼也一定有錯
- 程式碼不是規格或文件，是為了實作規格的行為

```go
// Ignore overflow, or buggy code?
//
// 1. Spec 1: Overflow would be the expected behaviour
// 2. Spec 2: Overflow should raise panic
func Add(a, b int16) int16 {
  return a + b
}
func AddWithOverflow(a, b int16) int16 {
  return a + b
}
func AddOrPanicOnOverflow(a, b int16) int16 {
  /* Panic */
}
```

測試碼

```go
AssertEquals(QuickSort([]int{ 30, 20, 40, 6 }), []int{ 6, 20, 30, 40 })
```

程式碼是行為(現象)，行為(現象)不等於規格

比喻: **頂樓加蓋是行為(現象)，行為(現象)不等於合法**

自動化測試的價值

1. 確認下次要釋出的版本無誤，只呈現了自動化測試的 40% 價值
2. 日後的修改，提供了另外 40% 的價值
    1. 讓原有功能確保無誤
    2. 相依第三方函式庫的更新，確保最小範圍相容

3. 日後的除錯，快速利用自動化測試來排除，補足了 20% 的價值

為什麼寫不出自動化測試

1. 不明白系統規格
2. 自動化測試不是用來找出現在系統運作有什麼問題，也不是用來檢測系統的健康情況
    - 系統運作的問題，是由系統內建的健康檢測工具來處理
3. 自動化測試是用來**再三重覆**確認系統的規格，是否有照預期的行為運作
4. 不會寫程式
    - `定義`, `實作`, `測試`
    - 一個可運作的軟體，只看「實作」的結果。Construction is the central activity in software development.
        - 所有的工作、流程、規定，都是為了產生可用的程式
    - 比喻: 人不吃東西、不喝水就會死，但單以餵養人食物、水就代表人是健康的。

定義

文件 - 成為公司的財產

實作

提交程式碼到 VCS(Version control systems) - 成為公司的財產

測試

沒有自動化測試 - 你在浪費公司的錢

1. 在定義的角度，等於你沒有說明系統功能的文件、user guide 或 tutorial
2. 在實作的角度，等於你的原始碼，只有你的電腦上有，沒有用 CVS

測試不代表完美

1. 自動測試通過不代表系統 100% 正確，它只代表**最基礎、最簡單**的情境下，基本功能確認無誤
2. 若**最基礎、最簡單**的情境測試不通過，系統也等於是垃圾。

為何單元測試很重要

1. 單元測試通常用來測試系統下幾層的模組。越底層出錯，其上層將全部出錯。
    - 例: 若 MySql? 的 `SELECT` 有邏輯 Bug，影響層面相當廣泛
2. 單元測試的開發成本最低
3. 單元測試的執行速度最快

Last modified on 2018-11-21T14:05:59+08:00