OOP 簡介

Sam Xiao, Sep.27, 2017

Overview

90 年代時，C++ 將 OOP 從原本小眾的 Smalltalk 帶進業界，OOP 從原本實驗室的概念，到現在每個主流程式語言都支援 OOP；但程式語言支援 OOP 是一回事，如何寫出 OOP 又是另外一回事，熟悉 OOP 已經算是每個 programmer 都必須具備的基本功。

最近幾年 FP 正夯，很多 junior programmer 開始跳過 OOP 直接學習 FP，但事實上 FP 與 OOP 本質也是一樣的，只是手段不一樣，OOP 靠的是 object，而 FP 靠的是 function。

> Function 可視為 `粒度` 更小的 object
>
> FP 可視為更嚴格的 OOP

所以學習 OOP 並不是過時的，`OOP 為體，FP為用`，在架構上使用 OOP 設計，但在實作上可視需求使用 OOP 或 FP。

> 沒有最完美的技術，只有最適合的技術，OOP 與 FP 都應該成為你 toolbox 中的重要武器，由 scenario 與 requirement 來決定該使用 OOP 或 FP。

Outline

軟體危機

軟體具有一些其他產業所沒有的特性，如 `容易實現`、`容易修改`，因此實務上會將 `容易變動` 的邏輯改用軟體實現，因此造就了特有的 `軟體危機`。

邏輯過於複雜

因為硬體的 `不易實現` 與 `不易修改` 的特性，大量邏輯改由軟體實現，且隨著時間不斷推進，人們的需求越來越多，邏輯只會越來越複雜。

需求變化太快

因為軟體主要用來實現 `容易變動` 的邏輯，因需求變動而改變規格，本來就是軟體的宿命，常常軟體才開發到一半，就因為需求變動而必須不斷地修改程式碼。

程式難以維護

程式越寫越大，無論新增功能或修改功能，常預期的功能寫好了，但原本的功能卻改壞了，也就是所謂的 `改 A 壞 B`。

> 這些都是軟體開發一定會面臨的問題，也是軟體工程一直想解決的課題。

軟體開發方法

為了解決軟體危機，人們開始發明各種開發方法：

結構化設計

> 以 `功能` 思考，使用 top down 方式寫程式
>
> 不能使用 `goto`
>
> 共用部分使用 function

代表語言：C

循序結構

> 程式碼由上而下，依序執行

選擇結構

> 根據 `條件式` 來選擇不同執行路徑，如 `if ... else`, `switch ... case`

迴圈結構

> 某一段程式碼反覆執行多次，如 `for` 、`while`

`結構化設計` 的時空背景，主要是針對 assembly 的大量使用 `goto` ，不使用 `function` 的程式風格提出解決方案，由於目前 programmer 並不是寫 assembly 出身， `結構化設計` 對大部分 programmer 已經不是問題。

> 基本上只要能寫出 `結構化` 的程式碼，就能解決所有的問題，因此很多人認為軟體 `入門簡單`，只要會結構化的這 3 招，就可以開發軟體了，因而能力都只停留在 `結構化` 程度而已。

模組化設計

> 模組 module
>
> 將高度相關的 function 集合在一起

代表語言 : C

> 模組化主要是搭配結構化設計，senior programmer 大都可以達到 `模組化` 要求。

物件導向設計

> 以 `模型` 思考，使用 `bottom up` 方式寫程式
>
> 模擬世界，加以處理

代表語言 : C++、Java、C#、TypeScript

建立模型，協助思考

若使用傳統以 `功能` 思考 :

1. 需要了解原 `功能` 程式碼的全部流程才能理解系統，新功能較不容易加上
2. 原來 `功能` 程式碼要加更多的 `if else` 判斷，執行不同功能
3. 原來 `功能` 程式碼因為加入新功能，行數越來越多

> 最後超過人類所能處理的複雜度而難以維護

若使用 `模型` 思考 :

1. 因為已經使用 `模型` 描述，因此較容易理解整個系統，新功能也較容易直接加到對的地方

2. 由於已經將整個系統拆成更小的物件表示，因此增加功能會直接加在相對應的物件上，可以少掉不少 `if else` 判斷
3. 因為整個系統由物件一起分工合作，因此程式碼行數將由眾多物件平均分擔

> 系統複雜度由模型加以簡化，隱藏了不必要的複雜度

依賴抽象，封裝變化

若使用傳統以 function 寫法：

1. 原來的程式碼要加更多的 `if else` 判斷，呼叫不同的 function
2. 只要 function 的 parameter 修改，則所有呼叫 function 的地方都要跟著修改

> 由於程式碼直接跟 function 耦合，只要需求改變，function 呼叫端必須跟著修改

若使用物件導向的 interface 寫法：

1. 原來的程式碼不需要增加 `if else` 判斷，已經將變化封裝成相同 interface 的不同 class
2. 若 parameter 修改，則使用 adapter class 加以轉換，由於 interface 沒變，則所有function 呼叫端不用修改

> 程式碼只與 interface 耦合，不與 function 耦合；呼叫端只認 interface，不認 function，只要 interface 不變，function 呼叫端不必修改

獨立模組，自給自足

`程式難以維護` 是軟體開發常被人詬病的問題，常因為程式越寫越大，最後已經超越人類所能處理的複雜度而無法維護，若我們能將程式分解成數個分開的部分，將有助於減少複雜度，這就是 `模組`。

> `模組化設計` 與 `物件導向設計` 都會建立模組，只是模組化設計是單一 file 或數個 file 建立模組；而物件導向設計是單一 class 或數個 class 建立模組。

模組人人會切，但最怕的是 A 模組的變動，必須配套 B 模組與 C 模組跟著修改，然後 B 模組的修改又造成連鎖反映導致 D 模組與 E 模組比需修改；當 A 模組修改，相關模組沒有跟著修改時，bug 就出現了，這也是 `改 A 壞 B` 主要發生的原因。

> 就類似我們修車時，最怕聽到的就是 A 零件壞掉，連 B 零件、C 零件也要跟著換；同樣的，當需求改變時，也最怕聽到的就是要修改 A 模組，連 B 模組、C 模組也要跟著修改。

因此我們在切模組時，應該切出能 `獨立運作` 的模組，也就是 A 模組的變動，並不需要任何模組的配套改變，就能達成所需功能；所有所需的變動，應該在 A 模組內完成。

非獨立模組：

- 將所需資料散佈在各模組
- A 模組所需的資料須由 B 模組的 function 提供
- A 模組若變動或抽換，B 模組也要跟著變動

獨立模組：

- 將所需資料放在同一模組內
- A 模組所需的資料完全由 A 模組的 function 自行提供
- A 模組若變動或抽換，與其他模組無關

解決問題

`物件導向設計` 能有效解決 `軟體危機`：

- `邏輯過於複雜`：改由人類容易理解的模型描述系統
- `需求變化太快`：將實作抽象化，使用端改依賴其抽象，需求變動不必修改使用端
- `程式難以維護`：將大系統切成數個獨立且自給自足的模組

> 物件導向思考方式其實比較人性化，但大部分 programmer 因為習慣 `功能` 思考的 `結構化` 與 `模組化` 設計，要改用 `模型` 思考 `物件導向` 設計反而比較困難。

Conclusion

- 儘管 OOP 已經 30 幾歲了，仍然是目前主流程式語言所支援的設計方式，尤其對於複雜度高的企業級程式非常有效。
- 程式語言支援 OOP 是一回事，能不能寫出 OOP 又是另外一回事。
- OOP 是以人的角度去看事情，其實非常人性化，只是大部分人已經被訓練以電腦角度看事情，因此反而學不好 OOP。