純粹的好，Pure Function 知道

Pure Function 介紹，以及在React、Redux開發中的角色

Wendell Liu
May 17, 2017 · 11 min read

Eiskristalle

我自己的程式開發學習路上，存有一個很迷人的名詞，便是本篇的主角 — Pure Function。想了許久，終於找到一段時間能夠把自己對於Pure Function 的心得記錄成文。

另外，儘管我自己是在學習Functional Programming 時知道所謂Pure Function 的觀念，但請別把這篇文章視為Functional Programming 的傳教文，因為我認為Pure Function 不只是單一一種編程方式下專屬的概念。

本篇會提到

- Pure Function 是什麼
- Pure Function 對開發有什麼益處
- Pure Function 與React、Redux

Pure Function 是什麼

Pure Function 的定義十分簡單（所以更適合其字面語意）：

> *將相同的輸入丟入，永遠都會回傳相同的輸出，並且不對任何該函數以外的任何作用域產生影響。*

以上定義可以分成兩個部分，一是相同輸入將獲得相同輸出。對於一個函數 ，輸入便是指其參數（Parameter），而輸出自然就是其回傳值。

舉個例子，實作一個加法函數，其會收進二個參數，並且回傳其總和。在任何時候，給定1 與 1 則會回傳2，而給定8 與 7 則將會獲得15。這樣的結果不會受到其他作用域（scope）影響，無論外部做了什麼，此函數依舊遵照此模式。

定義的第二部分，延續上一段，此函數不但不會受到其他作用域的影響，也不會影響其他作用域的值，也就是沒有副作用（Side Effect）。

以下一個簡單的範例碼：

```
// impure function
let intercept = 2
function math(x) {
  return (3 * x) + intercept
}

const result = math(4) // 14

// pure function
function math(itr) {
  return function(x) {
    return (3 * x) + itr
  }
}

const result = math(2)(4) // 14
```

好啦，我知道這範例蠻白癡的，應該沒有人會在這個需求上寫成上面的函數。要寫出一個滿足Pure Function 的函數是不困難的，困難的是將程式碼轉換成一個個Pure Function 。

小結，Pure Function 就是一個誰找他他就回一樣的話，他作的事情不受人影響，也不影響任何人。套個行話，Pure Function就是個*邊緣人*。

. . .

Pure Function 對開發有什麼益處

有利重構（Refactor）

由於Pure Function 同輸入同輸出的特性，讓它具備高**可預測性**，這對於重構你的程式碼是十分有利的。在開發過程中，由於各種可或不可抗拒的因素，可能會求快而先寫出一個版本的程式碼，接下來或許會排時間進行重構來達到穩定性、可擴充性等更高品質的水準。

Workaround 雖可恥但有用。我自己在初期開發時，儘管預先知道是個簡單的初版，仍會盡可能將內容抽成各個函數，並確保其為Pure Function 。將來替換時，僅需要理解該函數的作用，便能合理且快速地重構它。

反之若函數是個Impure Function，除了理解這個函數變不容易外，更可怕的是改動他可能會影響到其他地方；讓重構需要經過更長遠的計畫、花更多的時間，才能夠進行重構。如此一來，快速開發所產生的債則會多上許多倍。

方便測試

上一段提到了Pure Function 的高可預測性，事實上這點就已經滿足**方便測試**的條件。一個容易預測的函數，會很容易想到Test Case ，以及對應的actual value 與expect value 。

然而Pure Function 的高測試性不僅限於此。由於Pure Function 不受到其他作用域影響，只有輸入值才會導致結果的改變。換句話說，我們將不必再費心去準備許多環境就能做到測試的目的。在同一段測試區塊中，也大幅減少事後還原的動作。

易與其他函數組合（Compose）

在Functional Programming 中，有一個運用數學上結合律（associative laws）的組合技巧（function compose）。

```
function someMath(n) {
  return 6 + (5 * n)
}

function g(n) {
  return n * 5
}

function f(n) {
  return n + 6
}

const z = someMath(x)

// 等同於
const y = g(x)
const z = f(y)

// 能利用組合率
z = f(g(x))
```

這樣的技巧是將資料（上例的x）視作進入一道管線（pipeline）。如果我們將對資料的處理流程梳理得更加明確時，能夠進一步組合更多數量的函數。以Elixir 的pipe operator來看更顯直觀：

```
iex> "Hello World"
  |> String.upcase
  |> String.split
  |> Enum.map(fn x -> x <> "oo" end)

["HELLOoo", "WORLDoo"]
```

對我而言，除了看起來酷炫以外，函數組合的優勢在於讓多個只有一些小功能，但可靠的函數組合成一個值得信賴的「管線」。而這管線中每一道閥，都建議是個Pure Function 。除了能便於達到所謂的「可靠」外（前文所敘之高預期性、高測試性），更重要的是我們並不希望管線中的函數會在不同的情境、時間下會獲得不同的結果，這會導致管線結構的不單純。進一步說，由Pure Function 組合出的管線（函數組），自然也是個Pure Function。

強化Pure Function 概念的一大好處是讓開發者更能夠不拘束地使用**函數組合**的技巧。

並行運作

在開多個執行緒來讓程式碼並行運作時，有一個可能會遇見的問題是Race Condition。產生Race Condition 的其中一個可能，發生於某a 執行緒改變了某一項外在條件，導致b 甚至更多執行緒的結果不合預期，甚至發生錯誤。

由於Pure Function 不產生副作用也不受到外在影響的特性，不必擔心各個執行緒會影響他者運行的結果。我們能夠輕易地讓Pure Function 並行運作，不必擔憂產生Race Condition 的情況。

利於建快取（Cache）

Pure Function 的結果只取決於傳進的Parameter ，因此若結合閉包（closure）的作法，能夠妥善建立快取機制。判斷Parameter 是否是已經給進過的，若是則提取快取中的值，反之則加入快取並重新計算一個數值回傳。範例如下：

```
function cacheMath(n){
  const cache = {}
  return function(){
    if (n in cache) return cache[n]

    const result = doSomeComplexMathMethod(n)
    cache[n] = result
    return result
  }
}
```

可想而知的是，若回傳值除了受到n 亦會受到其他因素影響，由於回傳值的可預測性降低了，我們將無法有效的建立該快取制度。

專注於特定情境（Context）

最後這一點就相對開發時的心態。在理解Pure Function後，抱持著每個函數都盡可能達成其概念前提，這有助於開發者專注於各個情境。畢竟，真的很難在含有多個目的、情境的前提下，還能夠滿足Pure Function 的條件。

而這樣的專注能讓開發者較為不易陷入一個需求所導致的過度複雜思考，在起始時分離需求成不同目標，將目標轉換為不同的小函數，一一擊破。feature 一二三，寫code 多簡單。

. . .

與React、Redux 共舞

作為一個前端工程師，並且是React-Redux 開發為主的前端工程師，隨身攜帶一本Pure Function 指南也屬正常。底下會以一個React-Redux 架構開發者的角度，有什麼地方是與Pure Function 可以扯著寫成一段的。

React Component

如果不提萬惡的react context ，React Component 本身就是一個Pure Function 。Component 在概念上，呈現結果僅根據外在給的 `props` 以及 內部情境的 `state` 。當然這僅存於概念，你要在內部寫下多少Impure Function 來影響這個美麗的成果都是可以辦得到的。但React 各大介紹文中念茲在茲的 `view = f(x)` 想表述的概念其中有一應該就是Pure Function。

Redux Reducer

Redux 的架構是參考於**The Elm Architecture** （可以參考拙作 — 語言界的Redux？從學習Elm到函數式編程的啟發）。而Elm 作為一個Functional Programming Language ，自然沒在架構中放過Pure Function 。

Redux 的**Reducer** ，也就是Elm 的**Update** 乃是一個經典的Pure Function 案例。

```
const CounterReducer = (state, action) => {
  switch (action.type) {
    case INCREASE:
      return state + 1
    case DECREASE:
      return state - 1
    default:
      return state
  }
}
```

這是一個常見的Reducer 範例，拿到 `INCREASE` 就回傳當前的數值加一，反之拿到 `DECREASE` 則回傳當前的數值減一。整個Reducer 的結果取決於傳進的state 與action 的內容，是一個不折不扣的Pure Function 。

接下來Redux core 再將State 整個替換成Reducer 的回傳值，以更新Store 。

setState(object) -> setState(function)

沒看到Functional setState is the future of React 這篇文章前，還真的不知道原來React Component 的**setState** 除了能夠接受物件外，也能接受函數。這個函數是 `(state, props) => { doSomething() }` 的型態。

大部分的簡單情況下，我們只會直接拿某一值替換state 中的值。然而若是要寫得更加複雜，並且在不同component 間都會用到此邏輯，便能將他抽出成為一個獨立的函數，以達到程式碼獨立與DRY(Don't Repeat Youself.) 的目標。可想而知的是，要在各獨立的component 重複使用，此函數應盡可能是一個Pure Function。

除了上述兩個好處外，改用這樣的寫法也讓我們對component 的測試又多一個進入點，直接對functional setState 測試，更確保這一塊邏輯的穩固性。

另外，在react-future 中，也有一篇是延續此概念，且又更加擴充的 Stateful Function 。不得不說，真的永遠看不到Facebook 工程師的車尾燈。

Selector

React 與 Redux 的結合中，有一個重要的接口是 `react-redux` 所提供的 `connect` 。這讓React Component 能夠與Redux Store 串接。而通常會建議，把**取得Store 值的模式寫成函數**，以代入 `mapStateToProps` 。此函數就是**Selector** 。

Selector 該是一個Pure Function，否則如何能確保獲得的state 正確性？

另外，React 團隊也提供了 Reselect 使用，可以用這個lib 實作更複雜一些的Selector 。而Reselect 的文件中也提到：

> *A selector created with `createSelector` has a cache size of 1 and only returns the cached value when its set of arguments is the same as its previous set of arguments.*

看過原始碼後會發現， `createSelector` 確實實作了一套簡易的cache 機制，進而加速渲染的效率。這也是上方有提及Pure Function 其中一個優勢。

. . .

總結

Pure Function 是一個簡單但能夠帶來益處的概念，儘管在Functional Programming 特別強調，但我相信不僅限於FP 會需要。

在整個開發中肯定會有需要產生副作用的時刻，例如檔案I/O、資料庫寫入、DOM 操作等等。然而我們仍有必要盡可能將這些行為盡可能地收斂成最少數量的Impure Function，方便我們管理，維持系統的穩定性及可預測性。

Programming Functional Programming React Redux