Redux概念

Redux的官網中用一句話來說明Redux是什麼:

> Redux是針對JavaScript應用的可預測狀態容器

這句話雖然簡短，其實是有幾個涵義的:

- 可預測的(predictable): 因為Redux用了reducer與pure function的概念，每個新的state都會由舊的state建來一個全新的state，這樣可以作所謂的時光旅行除錯。因此，所有的狀態修改都是"可預測的"。
- 狀態容器(state container): `state` 是集中在單一個物件樹狀結構下的單一 `store` ， `store` 即是應用程式領域(app domain)的狀態集合。
- JavaScript應用: 這說明Redux並不是單指設計給React用的，它是獨立的一個函式庫，可通用於各種JavaScript應用。

有些人可能會認為Redux一開始就是Facebook所建立的專案，其實並不是，它主要是由Dan Abramov所開始的，Dan Abramov進入Facebook的React小組工作是最近的事情。他還有建立另外還有其他的相關專案，像React Hot Loader、React DnD，可能比當時的Redux專案還更廣為人知，在Facebook發表Flux架構不久之後，許多Flux的週邊函式庫，不論是加強版、進化版、大改版...非常的多。Redux一開始的對外展示的大型活動，是在2015年的React-Europe，影片Live React: Hot Reloading with Time Travel。影片中就有簡單的說明，Redux用了"Flux + Elm"的概念。

> Redux = Flux + Elm

當然除了Flux與Elm之外，還有其他的主要像RxJS中的概念與設計方式，Redux融合了各家的技術於一身，除了更理想的使用在Flux要解決的問題上之外，更延伸了一些不同的設計方式。

但是對初學者來說，它也不容易學習，網路上常常見到初學者報怨Redux實在有夠難學，這也並不是完全是Redux的問題，基本上來說Flux的架構原本就不是很容易理解，Redux還簡化了Flux的流程與開發方式。

所以我們要理解Redux是什麼，我們開始可以從這Flux與Elm兩大基礎來理解，以下分別說明一些基本的概念。

Flux

不論是Flux或其他以Flux架構為基礎延伸發展的函式庫(Alt、Reflux、Redux...)都是為了要解決同一個問題，這個問題在React應用規模化時會非常明顯，簡單以一句話來說就是:

> 應用程式領域(app domain)的狀態 - 簡稱為 `App state`

應用程式都需要有 `App state` (應用程式狀態)，不論是在一個需要使用者登入的應用，要有全域的記錄著使用者登入的狀態，或是在應用程式中不同操作介面(元件)或各種功能上的資料溝通，都需要用到它。如果你已經有一些程式語言或應用的開發經驗，你應該知道這會像是MVC設計模式中的Model(模型)部份該作的事情。

React應用為什麼會出現這個問題？原因主要是來自React元件的本身設計限制造成的。React被設計為一個相似於MVC架構中的View(視圖)函式庫，實際上它可以作的事情比MVC中的View(視圖)還要更多，但本質上的確React並不是一個完整的應用程式開發框架，裡面沒有額外的架構可以作類似Model(模型)或Controller(控制器)的事情。對小型的元件或應用而言，應用程式的資料都包含在裡面，也就是在View(視圖)之中。

有學過React的一些基礎的開發者應該會知道，在React中的元件是無法直接更動 `state` (狀態)值，要透過 `setState` 方法來進行更動，這有很大的原因是為了Virtual DOM(虛擬DOM)特性的所設計，這是其中一點。另外在元件的樹狀階層結構，父元件(擁有者)與子元件(被擁有者)的關係，子元件是只能由父元件以 `props` (屬性)來傳遞屬性值，子元件自己本身無法更改自己的 `props` ，這也是為什麼一開始在學習React時，都會看到大部份的範例只有在最上層的元件有 `state` 值，而且都是由它來負責進行當資料改變時的重新渲染工作，子元件通常只有負責呈現資料。

當然，有一個很技巧性的方式，是把父元件中的方法定義由 `props` 傳遞給子元件，然後在子元件觸發事件時，呼叫這個父元件的方法，以此來達到子元件對父元件的溝通，間接來更動父元件中的 `state` 。不過這個作法並不直覺，需要事先規範好兩邊的方法。在簡單的應用程式中，這溝通方式還可行，但如果是在有複雜的元件階層結構時，例如層級很多或是不同樹狀結構中的子元件要互相溝通時，這個作法說實在是派不上用場的。

在複雜的元件樹狀結構時，唯一能作的方式，就是要將整個應用程式的資料整合在一起，然後獨立出來，也就是整個應用程式領域的資料圓份。另外還需要對於資料的所有更動方式，也要獨立出來。這兩者組合在一起，就是稱之為"應用程式領域的狀態"，為了區分元件中的狀態(state)，這個作為應用程式領域的持久性資料集合，會被稱為 `store` (儲存)。

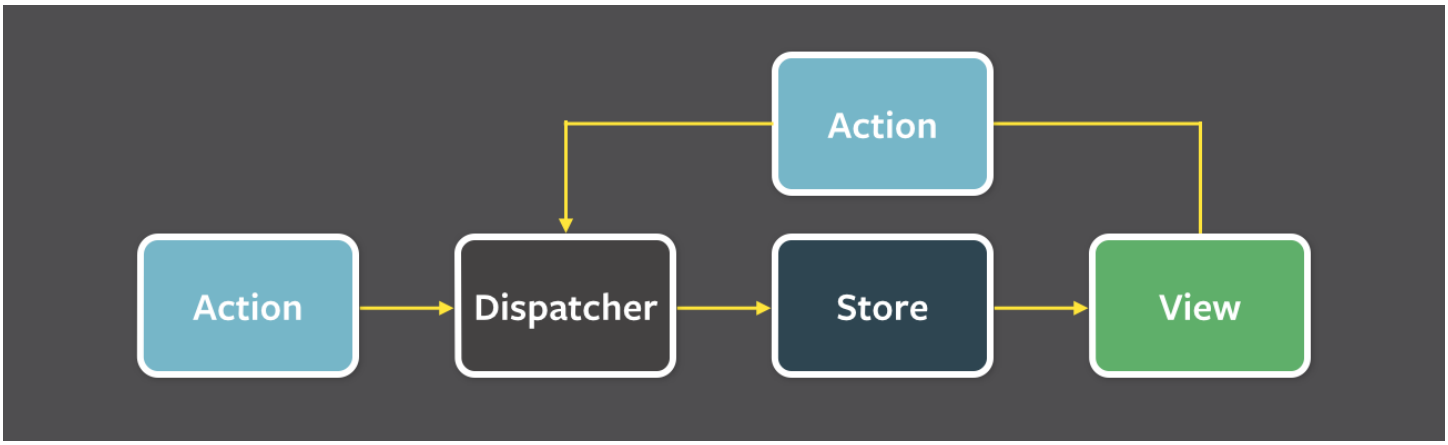
> store(儲存)並不是只有應用程式單純的資料集合而已，它還包含了所有對資料的更動方法

`store` (儲存)的角色並非只是元件中的 `state` (狀態)而已，它也不會只有單純的記錄資料，可能在現今的每種不同的Flux延伸的函式庫，對於 `store` 的定義與設計都有所不同。在Flux的架構中的 `store` 中，它包含了對資料更動的函式/方法，Flux稱這些函式/方法為Store Queries(儲存查詢)，也把它的角色定位為類似傳統MVC的Model(模型)，但與傳統的Model(模型)最大明顯不同之處的是， `store` 只能透過Action(動作)以"間接"的方式來自我更新。

`store` 的設計可以解決應用程式的狀態存放與更動的問題，但它並不是整個解決問題的完整核心，只能算是其中一個重要的參與成員。最困難的地方在於，要如何在觸發動作時，進行 `store` (儲存)的更動查詢，以及進行呈現資料的更動與最後作整個應用程式的渲染。這一連串的步驟，整合為一個資料流(Data Flow)，Flux的名稱來由其實就是拉丁文中的 `Flow` ，Flux用單向(unidirectional)資料流來設計整個資料流的運作，也就是說整個資料的流動方向都是一致的，從在網頁上呈現的操作介面元件，被觸發事件後，傳送動作到發送器，再到 `store` ，最後進行整個應用的重新渲染，都是往單一個方向執行。

> 單向資料流是Flux架構的核心設計

下面是個簡單的流程示意圖，上面有標出主要的參與成員，來自Flux官網:



這個資料流的位於最中心的設計是一個AppDispatcher(應用發送器)，你可以把它想成是個發送中心，不論來自元件何處的動作都需要經過它來發送。每個 `store` 會在AppDispatcher上註冊它自己，提供一個callback(回調)，當有動作(action)發生時，AppDispatcher(應用發送器)會用這個回調函式通知 `store` 。

由於每個Action(動作)只是一個單純的物件，包含actionType(動作類型)與資料(通常稱為payload)，我們會另外需要Action Creator(動作建立器)，它們是一些輔助函式，除了建立動作外也會把動作傳給Dispatcher(發送器)，也就是呼叫Dispatcher(發送器)中的dispatch方法。

> 註: Payload用在電腦科學的意思，是指在資料傳輸時的"有效資料"部份，也就是不包含傳輸時的頭部資訊或metadata等等用於傳輸其他資料。它的英文原本是指是飛彈或火箭的搭載的真正有效的負載部份，例如炸藥或核子彈頭，另外的不屬於payload的部份當然就是火箭傳送時用的燃料或控制零件。

Dispatcher(發送器)的用途就是把接收到的actionType與資料(payload)，廣播給所有註冊的callbacks。它這個設計並非是獨創的，這在設計模式中類似於pub-sub(發佈-訂閱)系統，Dispatcher則是類似Eventbus的概念。Dispatcher類別的設計很簡單，其中有兩個核心的方法，這兩個是互為相關的函式:

- `dispatch` 發送payload(相當於動作)給所有註冊的callbacks。元件觸發事件時用這個方式來發送動作。
- `register` 註冊在所有payload(相當於動作)發送時要呼叫的callbacks(回調)。這些callbacks(回調)就是上面說的會用來更動 `store` 的Store Queries(儲存查詢)。

在資料流的最後， `store` 要觸發最上層元件的 `setState` ，然後進行整體React的重新渲染工作。Flux提出的方式一種自訂事件的監聽方式，把 `store` 用 `EventEmitter.prototype` 物件進行擴充，讓 `store` 具有監聽事件的能力，然後在最上層元件中的生命週期中，加入有更動時的監聽事件。JavaScript中內建的Event、CustomEvent等介面，以及addListener、dispatch等方法，只能實作在具有事件介面的網頁DOM元素上。單純在JavaScript的物件上是沒有辦法使用，要靠額外的函式庫才能這樣作，這是一定要使用類似像EventEmitter這種函式庫的主要原因。

不過，你可能會覺得為什麼不乾脆一點直接對 `store` 上面作更動就好了，一定要拐這麼大一個彎，透過Action(動作)"間接"的方式來作自我更新？

我想原因之一，是要標準化Action(動作)的規格，也就是所有在應用程式中的元件，都得要按照這些動作來觸發事件，發送器中註冊的callback也是要寫成處理同一種規格的動作。Action(動作)主要由type(類型)與payload(有效資料)組成，Flux Standard Action(Flux標準動作)就是提出來要標準化Action(動作)的格式，有了統一格式的Action物件，在更新資料時所有更新方式會具統一性，這樣Flux才有辦法把整個資料流運作完成一個循環再接著下一個。就像網路的傳輸協定一樣，資料的格式與運作的流程，都有標準的規範，不是隨隨便便就可以進行傳輸。當然還有一些其它的原因，例如要避免Event Chains(事件連鎖)的發生。

整個的資料運作流程，大概是像下面這樣:

```
事件觸發 ->
由Action Creator呼叫Dispatcher.dispatch(action) ->
Dispatcher呼叫已註冊的callback ->
呼叫對應的Store Queries(儲存查詢) ->
觸發Store更動事件 ->
進行整個應用的重新渲染
```

下面整個流程的示意圖，來自Flux官網:

總結來說，Flux使用了單向資料流的設計架構，是為了要解決React的應用程式領域狀態的問題。Flux的實作並不容易，有許多細節與開發步驟上分割不明確的問題，所以在此並不討論Flux的實作部份。在Flux發表之後(約為2014年中)，陸陸續續出現了許多函式庫與框架，都是基於Flux的基本設計概念，改善簡化或自動化其中的部份實作步驟為主，Redux也是其中一套。在經過一段時間之後，目前較熱門的與較多人使用的，就是Redux，它有很多的設計概念都來自於Flux，能多理解Flux的基本設計概念，對於學習Redux是絕對有幫助的。

Elm

或許你有聽過函數式程式開發(functional programming, FP)的開發風格，FP是什麼？用下面的一句話來說明，摘譯自這篇教學文章:

> 函數式程式開發就是只使用"純粹函數"與"不可改變的值"來撰寫軟體應用的一種方式

FP是現今相當熱門的一種程式開發風格，在很早之前就已經有一些純函數式程式開發的語言例如Haskell與OCaml，Elm也是一個純函數式程式開發的語言，它是一個很年輕的語言，Elm是專門用來開發網站應用程式的程式語言，最終編譯為JavaScript在網頁上執行，它與JavaScript語言有多差異很大的設計，例如:

- Elm是強(靜態)資料類型的，它的資料類型也滿多樣的
- Elm是純FP的語言
- Elm-Architecture是包含在Elm的應用框架，它是單向資料流的架構

React與Flux中有許多設計，都有應用到FP的設計，與Elm中一部份設計相當類似。而Redux又使用更多Elm中的設計，尤其是Elm-Architecture而來的，例如:

- 不可改變性(Immutability): 所有的值在Elm中都是不可改變的，Redux中的pure function(純粹函式)與Reducer的設計很類似，React的設計中也有這類的概念。
- 時光旅行除錯(Time Traveling Debugger): 在Elm有這個設計，Redux學了過來。

> Redux作者使用了FP(函數式程式開發)與Elm的架構，改進或簡化原本的Flux架構

為何要學習Redux/Redux的優點

Redux是目前最熱門的、最多人使用的Flux架構類的函式庫，雖然Redux也可以用於其他的函式庫，但基本上它是專門為了React應用所打造的。如果你真的要學會React，並用它來開發一個稍有規模的應用，學習Redux說是一條必經之路，當然也有其他的Flux架構類函式庫可以選擇，不同的函式庫有可能使用的解決方式與樣式相差會非常大。目前來說Redux的開發社群是最龐大也是最活躍的，而且不見得其他的函式庫就會更容易學習與使用，這是一個選擇性的問題。畢竟用得人多，你會遇到的問題大概都有人遇過，也都能找得到解決方式，這是開放原始碼生態圈的紅利。

Redux會受歡迎不是沒有原因的，以下分析幾個Redux的優點:

使用了FP(函數式程式開發)與React可以配合得很好

Redux不同於Flux架構，它改採幾乎是純粹FP(函數式程式開發)的解決方式，目的是為了要簡化Flux中資源流的處理實作，但在另一部份的確可以與React中的元件渲染配合得很好，這証明了它的確是找到了一個較為理想的，與React應用能密切合作的解決方式。FP(函數式程式開發)也是目前JavaScript界的熱門主題，Redux的確吸引到不少開發者的目光。

時光旅行除錯/熱重新載入

Redux一開始就附了時光旅行除錯工具與熱重新載入(hot reloading)的搭配使用，提昇開發經驗，這有開發者有很大的吸引力，這也代表在Redux應用上的資料變動，可以更容易的測試與除錯，這是其他Flux架構類函式庫或框架中所沒有的見到的。

更簡化的程式碼，更多可能的延伸應用

Redux一開始的版本只有99行程式碼，這可能比一開始的Flux架構使用的API更要少。Redux一開始就可以很容易的使用於伺服器端渲染，而且也不限於使用於React應用上，這可以吸引更多的開發者使用。

更多的文件，發展良好的生態圈

Redux作者一開始就撰寫非常多的文件與教學，讓許多開發者能更快速地掌握Redux的應用技術，Redux作者也是技術討論區的常客，常常可以看到他在討論區上回覆相關的問題。Redux的專案也是相當活躍的，各種大大小小的問題，都有非常多的參與者在討論與解決，對於重大效能/臭蟲問題也是很快速地解決。

Redux的三大原則

Redux裡的強硬規則與設計不少，大部份都會與FP(函數式程式開發)，以及改變了原本的Flux架構中的設計有關。這三大基本原則主要是因為有可能怕初學者不理解Redux中的一些限制或設計，所以先寫出來說明，這裡面也會說明了Redux的設計原理基礎是如何，所以強烈建議所有的初學者一定要理解這三大原則中的意義。以下分別說明，主要以原文的標題與內容說明，儘可以說明的比較清楚些。

單一真相來源(Single source of truth)

> 你的整個應用中的state(狀態)，會儲存在單一個store(儲存)之中的一個物件樹狀結構裡。

Redux中只有用單一個物件大樹結構來的儲存整個應用的狀態，也就是整個應用中會用到的資料，稱之為 `store` (儲存)。但要注意的是 `store` (儲存)並不是只有單純的資料而已。 `store` 就是應用程式領域的狀態，它是類型MVC中的Model(模型的)設計的概念，這設計是由Flux架構而來的，在原本的Flux架構中是允許多個 `store` 的結構，Redux簡化為只有單一個。

Redux的單一個store的設計有一些好處，對開發者來說，它可以容易除錯與觀察狀態的變化，狀態儲存於物件樹狀結構中，也很容易作到重作/復原(Undo/Redo)的功能。因為只有一個 `store` ，但如果 `store` 裡要儲放多個不同的狀態物件，以及每次的更動資料，自然就會變成了物件樹狀結構(object tree)。

最後，如果你想要從store中取出目前的狀態資料，可以用store物件的 `getState()` 方法。

狀態是唯讀的(State is read-only)

> 唯一能更動狀態的是發送一個 `action` (動作)， `action` 是一個描述發生了什麼事的純物件

這裡指的"狀態"，是上面說的儲放在store中的狀態資料，你不能直接對其中的狀態資料更動，這與原先的React中的 `state` 與 `setState` 的概念有點像，Redux的意思是你不能直接更動 `store` 裡面所記錄的狀態值，只能"間接"地透過發送 `action` 物件來叫 `store` 更動狀態。"間接"地更動狀態是一個很關鍵的設計，這是Flux中單向資料流的重點之一，這對於每個動作發生，最終會影響到什麼狀態上的更動，一個接一個的順序等等的一種嚴格的設計。

"發生了什麼事"這句，是代表每個 `action` 都會有一個type(類型)，代表這個動作是要作什麼用的，或是現在是發生了什麼，例如是要新增一筆什麼資料，或是刪除整個資料等等，動作物件除了要說明它是要作什麼之外，也需要包含所影響的資料。

發送(emit)一個action，用的是 `store.dispatch(action)` 語法樣式，下面這個範例就是一個要更動狀態的程式碼:

```
store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
})
```

中間的那個純物件，就叫作 `action` (動作)，它是一個單純用於描述發生了什麼事與相關資料的純物件:

```
{
  type: 'COMPLETE_TODO',
  index: 1
}
```

還記得我們在React中的 `state` 與 `setState` 方法的設計嗎？ `state` 也是不能直接更動的，一定要透過 `setState` 方法才能更動它。那這是為什麼呢？因為 `setState` 不光只是更動 `state` 值，它還要作重新渲染的動作，React需要比對目前的狀態，與即將要變動的狀態，這樣才能進行動新渲染的工作。Redux的設計中 `store` 是與React中的 `state` 相比，它們之間有一些類似的設計。

更動只能由純粹函式來進行(Changes are made with pure functions)

> 要指示狀態樹要如何依actions(動作)來改變，你需要撰寫純粹的歸納函式(reducers)

Redux中的reducer的原型會長得像下面這樣，你可以把它當作就是 `之前的狀態 + 動作 = 新的狀態` 的公式:

```
(previousState, action) => newState
```

> 註: 你可以參考Redux中Reducers這一章的內容，裡面有實例。

不過，Redux中的reducer要求的是一定是pure function(純粹函式)，也就是不能有副作用(side effect)的函式。因此由reducer所產生的新狀態，並不是直接修改之前的狀態而來，而是一個複製之前狀態，然後加上動作改變的部份，產生的一個新的物件，它這樣設計是有原因的。

Redux的store設計，並不是原本Flux的store，而是ReduceStore，這個ReduceStore是一個在Flux中的 `store` 進化版本，在說明中有一個叫作reduce的方法，說明如下:

> reduce(state: T, action: Object): T 歸納(Reduces)目前的state(狀態)與一個action(動作)到新的store中的state(狀態)。所有的子類都需要實作這個方法。這個方法必須是純粹而是無副作用。

那為何要用這個進化的ReduceStore？它最後有說明一段:

> 不需要發送更動事件注意所有繼承自ReduceStore的store，不需要手動發送(emit)在reduce()中的更動事件...state(狀態)會自動地比對在每個dispatch(發送)之前與之後，與自動地作發送更動事件...

`ReduceStore` 的設計與Redux最一開始的版本差不多是同時間發佈的，在開發者之間彼此有交流。Redux的store運用了類似於ReduceStore的設計，所以要更動Redux中的store，需要透過reducer，自然這是為了簡化原本在Flux資料流的運作實作流程。

reducer在Redux中扮演了十分重要的關鍵角色，它是一種store中所存放的狀態，要如何因應不同的動作而進行更新的函式，而store也是由reducer所建立，例如像下面的程式碼:

```
// @Reducer
//
// action payload = action.text
// 使用純粹函式的陣列unshift，不能有副作用
// state(狀態)一開始的值是空陣列`state=[]`
function todoApp(state = [], action) {
  switch (action.type) {
    case 'ADD_ITEM':
      return [action.text, ...state]
    default:
      return state
  }
}

// @Store
//
// 由reducer建立store
const store = createStore(todoApp)
```

針對應用中不同功能的狀態，可以分別寫出不同的reducer，Redux中提供了 `combineReducers` 函式可以合併多個reducer，例如以下的程式碼:

```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([ action.text ])
    default:
      return state
  }
}

function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
```

```
      return state - 1
    default:
      return state
  }
}

// rootReducer是個組合過的函式，
// 這裡用的是物件屬性初始設定簡寫法，
// combineReducers傳入參數是一個物件
const rootReducer = combineReducers({
  todos,
  counter
})
```

Action與Action Creator

這兩個是Flux架構中的參與成員，redux中有說明Action的定義:

> Actions are payloads of information that send data from your application to your store. 中譯: Actions(動作)是從你的應用送往store(儲存)的資訊負載

你可能會一直在Action(動作)這裡看到 `payload` 這個字詞，它是 負載 或 有效資料 的意思，這個字詞的意思解說你可以看一下，不難理解:

> Payload用在電腦科學的意思，是指在資料傳輸時的"有效資料"部份，也就是不包含傳輸時的頭部資訊或metadata等等用於傳輸其他資料。它的英文原本是指是飛彈或火箭的搭載的真正有效的負載部份，例如炸藥或核子彈頭，另外的不屬於payload的部份當然就是火箭傳送時用的燃料或控制零件。

這個Action是有一個固定格式的，叫作FSA, Flux Standard Action(Flux標準動作)，格式的範例會像下面這樣，是個JavaScript的物件字面定義:

```
{
  type: 'ADD_TODO',
  payload: {
    text: 'Do something.'
  }
}
```

這樣一個用於描述動作的單純物件字面定義，就稱為Action(動作)。

為什麼要先寫出明確的Actions(動作)，也就是把所有的元件會用到的Actions(動作)，全部集中寫到一個檔案中？這也是個硬規則，就像你如果參加奧運的體操比賽，每種項目都有規定的動作，在一定的時間內只能作這些動作，按照表定執行。主要還是因為Redux並不知道你的應用程式裡會作什麼動作，需要有一個明確說明有哪些動作的地方，在運作時以這個對照表為基準。

當然，Actions(動作)必需要有type(類型)，而且在同一個應用中的type(類型)名稱是不能重覆的，它的概念有點類似於資料表中的主鍵屬性。

> 那麼Action Creator(動作生成器)又是什麼？

在程式語言的函式庫中，如果是個英文的名詞，通常都是代表某種物件或資料格式，例如Action(動作)就是個單純的物件。如果叫什麼xxxxter或xxxxtor的，中文翻譯是"器"、"者"，通常就是個函式或方法，像上面的reducer和這裡的Action Creator，都是一種函式。

Action creator的設計也是由Flux架構來的產物，它是一種輔助用的函式，用來建立Action的。但因為設計的不同，在Redux中的Action creator比在Flux更簡單，它通常只用來回傳Action物件而已，當然它本身是個函式，在回傳前是可以再針對回傳的動作資料先進行運算或整理的，例如像下面這樣的函式:

```
export function addTodo(text) {
  return { type: ADD_TODO, text }
}
```

這個 `addTodo` 函式，有一個傳入參數，這個傳入參數就會用於組成Action物件中的 `payload` (有效資料)。

如果一個Action物件簡單到連payload(有效資料)都沒有，通常會是個固定payload(有效資料)的動作，例如每動作一次+1或-1，或是每動作一次在true或false值切換，那麼在Redux中允許連Action或Action Creator都可以不用寫了。但是這種情況大概只有在很小的應用，或是學習階段的範例才會這樣，如果應用還是有一定程度的複雜度，一定都是要寫出來的。

當然，Action Creator自然有它很重要的作用，其中之一就是處理有副作用的執行，例如計時器、Fetch/Ajax等等，因為reducer是一個強制無法有副作用的純粹函式，所以Redux中的副作用會寫在在Action Creator裡，不過這需要再配合中介軟體(middleware)來執行，之後的章節會再說明。

> 註: Action Creator在Redux中並沒有要求一定要是個純粹函式，只是不建議在裡面直接執行有副作用的函式，之後的章節會有說明。請參考這篇在stackoverflow的Reduce作者的回答。

reducer(歸納器)

reducer(歸納器)這種函式的名稱，是由陣列的一個迭代方法reduce(歸納)而來，你可以參考MDN中的相關說明，以下的內容與範例出自從ES6開始的JavaScript學習生活這本電子書中:

reduce(歸納)這個方法是一種應用於特殊情況的迭代方法，它可以藉由一個回調(callback)函式，來作前後值兩相運算，然後不斷縮減陣列中的成員數量，最終回傳一個值。reduce(歸納)並不會更動作為傳入的陣列(呼叫reduce的陣列)，所以它也沒有副作用。

```
const aArray = [0, 1, 2, 3, 4]

const total = aArray.reduce(function(pValue, value, index, array){
    return pValue + value
})

console.log(aArray) // [0, 1, 2, 3, 4]
console.log(total) // 10
```

按照這個邏輯，reduce(歸納)具有分散運算的特點，可以用於下面幾個應用之中:

- 兩相比較最後取出特定的值(最大或最小值)
- 計算所有成員(值)，總合或相乘
- 其它需要兩兩處理的情況(組合巢狀陣列等等)

副作用與純粹函式

> 當一個函式是純粹的時候，我們可以說 輸出 只取決於 輸入

對於函式來說，具有副作用代表著可能會更動到外部環境，或是更動到傳入的參數值。函式的區分是以 純粹(pure)函式 與 不純粹(impure)函式 兩者來區分，但這不光只有無副作用的差異，還有其他的條件。

純粹函式(pure function)即滿足以下三個條件的函式，以下的定義是來自於Redux的概念:

- 給定相同的輸入(傳入值)，一定會回傳相同輸出值結果(回傳值)
- 不會產生副作用
- 不依賴任何外部的狀態

一個典型的純粹函式的例子如下:

```
const sum = function(value1, value2) {
  return value1 + value2
}
```

套用上面說的條件定義，你可以用下面觀察來理解它是不是一個純粹函式:

- 只要每次給定相同的輸入值，就一定會得到相同的輸出值: 例如傳入1與2，就一定會得到3
- 不會改變原始輸入參數，或是外部的環境，所以沒有副作用
- 不依賴其他外部的狀態，變數或常數

那什麼又是一個不純粹的函式？看以下的範例就是，它需要依賴外部的狀態/變數值:

```
let count = 1

let increaseAge = function(value) {
  return count += value
}
```

在JavaScript中不純粹函式很常見，像我們一直用來作為輸出的 `console.log` 函式，或是你可能會在很多範例程式看到的 `alert` 函式，都是"不"純粹函式，這類函式通常沒有回傳值，都是用來作某件事，像 `console.log` 會更動瀏覽器的主控台(外部環境)的輸出，也算是一種副作用。

每次輸出值都不同的不純粹函式一類，最典型的就是 `Math.random` ，這是產生隨機值的內建函式，既然是隨機值當然每次執行的回傳值都不一樣。

例如在陣列的內建方法中，有一些是有副作用，而有一些是無副作用的，這個部份需要查對應表才能夠清楚。不會改變傳入的陣列的，會在作完某件事後回傳一個新陣列的方法，就是無副作用的純粹函式(方法)，而會改變原陣列就算是不純粹函式(方法)了。

下面是兩個在陣列中作同樣事情的不同方法，都是要取出只包含陣列的前三個成員的陣列。一個用splice，另一用是slice，看起來都很像，連這兩個方法的名稱都很像，但卻是完全屬於不同的種類:

```
// 不純粹(impure)，splice會改變到原陣列
const firstThree = function(arr) {
  return arr.splice(0,3)
}

// 純粹(pure)，slice會回傳新陣列
const firstThree = function(arr) {
  return arr.slice(0,3)
}
```

其他有許多內建的或常用的函式都是免不了有副作用的，例如這些應用:

- 會改變傳入參數(物件、陣列)的函式(方法)
- 時間性質的函式，setTimeout等等
- I/O相關
- 資料庫相關
- AJAX

純粹函式當然有它的特別的優點:

- 程式碼閱讀性提高
- 較為封閉與固定，可重覆使用性高
- 輸出輸入單純，易於測試、除錯
- 因為輸入->輸出結果固定，可以快取或作記憶處理，在高花費的應用中可作提高執行效率的機制

最後，並不是說有副作用的函式就不要使用，而且要很清楚的理解這個概念，然後儘可能在你自己的撰寫的一般功能函式上使用純粹函式，以及讓必要有副作用的函式得到良好的管控。現在已經有一些新式的函式庫或框架(例如Redux)，會特別要求在某些地方只能使用純粹函式，而具有副作用的不純粹函式只能在特定的情況下才能使用。

> 註: 雖然在副作用與純粹函式的介紹中，我們有提到一些呼叫外部API(console.log/alert)、時間(Date())、隨機(Math.random)也屬於有副作用的呼叫，但以等級來區分它們只算是"輕度"或"微量"的副作用，這些在reducer或Action Creators能不能用？答案是可以用但也不要用，它會影響到純粹函式的一些最佳化。以在副作用的主題來說，異步執行才是"中度"或"一般"等級的副作用，我們談到副作用通常是指這個等級的。當然也有"重度"等級的副作用，那是另一個層次的特殊應用情況討論，例如組合出來的複雜異步執行流程結構。