Redux 整合 React 檔案組織重構

模組系統中的 import/export

`import`加上萬用字元(`*`)這會這匯入的模組整個成為一個模組物件進行匯入，下面的範例的 actions 會是一個物件:

```
import * as actions from './actions'
```

預設的(Default import)，這種方式只能用在預設的輸出名稱上:

```
import defaultMember from 'src/my_lib'
```

命名空間的(Namespace) import:

這會這匯入的模組整個成為一個模組物件進行匯入，其中有輸出的函式與值都會變為此物件的屬性值之一:

```
import * as my_lib from 'src/my_lib'
```

識別名(Named) imports:

以模組中的識別名稱撰寫在花括號中指定要匯入的。

```
import { name1, name2 } from 'src/my_lib'
```

在匯入時可以更變名稱:

```
// Renaming: import `name1` as `localName1`
import { name1 as localName1, name2 } from 'src/my_lib'

// Renaming: import the default export as `foo`
import { default as foo } from 'src/my_lib'
```

default 是個關鍵字詞，直接從一個模組匯入，就是指要匯入它的 default 的部份(函式、類或表格達式)，下面兩個語法實際上是相等的:

```
import { default as foo } from 'lib'
import foo from 'lib'
```

default 的匯入與其它部份(識別名稱匯入)是分開的，如果要同時在一個語句中作這兩件時，就要用合併的語法，default 必定要寫在前面:

```
import theDefault, * as my_lib from 'src/my_lib'

import theDefault, { name1, name2 } from 'src/my_lib'
```

有識別名的輸出(named export)

內嵌名稱輸出(Inline named exports)，直接在定義語句中加上 export:

```
export var myVar1 = ···;
export let myVar2 = ···;
export const MY_CONST = ···;

export function myFunc() {
    ···
}
export function* myGeneratorFunc() {
    ···
}
export class MyClass {
    ···
}
```

名稱輸出於從屬語句(Named exporting via a clause)，在檔案的最後加上 export 語句，注意要使用花括號:

```
const MY_CONST = ···;
function myFunc() {
    ···
}

export { MY_CONST, myFunc };
```

預設輸出(export default)

default 與 export 搭配時為一個語言中的關鍵字，通常是模組只有單一個函式或類別時，才會使用 export default。

有三種情況可使用`export default`:

- 函式(包含匿名函式)
- 類別(包含沒有識別名的類別)
- 表格達式

```
// 函式
export default function myFunc() {}
export default function () {}

export default function* myGenFunc() {}
export default function* () {}

// 類別
export default class MyClass {}
export default class {}

// 表格達式
export default foo;
export default 'Hello world!';
export default 3 * 7;
export default (function () {});
```

因為`default`是個關鍵字，下面兩個語句的寫法也是相等的：

```
//------ module1.js ------
export default function foo() {} // function declaration!

//------ module2.js ------
function foo() {}
export { foo as default };
```

> 特別注意：以 var、let 與 const 開頭定義的語句，不能使用`export default`，只能用於`export`

重新輸出(Re-exporting)

從別的模組重新輸出成另一個模組，用的是`export...from`語法：

```
// 全部重新輸出
export * from 'src/other_module';

// 部份重新輸出
export { foo, bar } from 'src/other_module';

// 部份重新輸出，並更動名稱
export { foo as myFoo, bar } from 'src/other_module';
```

特別注意：`export *`會忽略掉`export default`的部份，default 的部份要像下面這樣寫才可以：

```
export { default } from 'foo';
export { myFunc as default } from 'foo';
```

多個模組檔案的重新輸出寫法:

```
export { default as Comp1 } from './Comp1.jsx'
export { default as Comp2 } from './Comp2.jsx'
export { default as Comp3 } from './Comp3.jsx'
```

從某個目錄的檔案 index.js 匯入:

```
// 在某個目錄下有個index.js
export * from 'ThingA'
export * from 'ThingB'
export * from 'ThingC'

// 匯入部份的寫法
import { ThingA, ThingB, ThingC } from 'lib/things'
```

另一個類似的寫法:

In a Thing,

```
export default function ThingA() {}
```

In things/index.js,

```
export { default as ThingA } from './ThingA'
export { default as ThingB } from './ThingB'
export { default as ThingC } from './ThingC'
```

Then to consume all the things elsewhere,

```
import * as things from './things'
things.ThingA()
```

Or to consume just some of things,

```
import { ThingA, ThingB } from './things'
```

混合的輸出

通常來說，一個模組不是只有 export default 單一個部份，就是一個個 export 每個部份，在這兩種方式只選擇一種。

有些框架或函式庫，會採用混合兩種方式的寫法，例如 underscore 或 jQuery 的寫法：

```
//------ underscore.js ------
export default function(obj) {
  //···
}
export function each(obj, iterator, context) {
  //···
}
export { each as forEach }

//------ main.js ------
import _, { each } from 'underscore'
//···
```

_就是預設的函式，each 是另一個輸出的部份。

參考資料

- http://exploringjs.com/es6/ch_modules.html#sec_importing-exporting-details
- https://developer.mozilla.org/zh-TW/docs/Web/JavaScript/Reference/Statements/export
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import

redux

先提出基礎概念供參考：組織程式碼的四大策略

以下說明組織 Redux 程式碼的一些組織說明。首先，根據 Redux 官方的這篇有關如何組職程式碼的問答文章中(中文的翻譯在此)，一共列了三種常見的方式：

Rails 風格

又可以稱為"依照類別型(by type)"的集中組織方式。用 actions, constants, reducers, containers, components 等目錄區分，文章名可以依功能或應用命名區分，這大概是最常見的一種。範例如下：

```
actions/
    CommandActions.js
    UserActions.js
components/
    Header.js
    Sidebar.js
    Command.js
    User.js
    UserProfile.js
    UserAvatar.js
containers/
    App.js
```

```
    Command.js
    User.js
reducers/
    index.js
    command.js
    user.js
routes.js
index.js
rootReducer.js
```

應用領域風格(Domain)

又可以稱為"依照功能(by feature)"的集中組織方式。先以功能或應用領域不同的目錄區分，目錄裡有各自的 reducer、action 等等文章，可以用文章命名再作類別型區分。範例如下:

```
app/
    Header.js
    Sidebar.js
    App.js
    rootReducer.js
    routes.js
product/
    Product.js
    ProductContainer.js
    ProductActions.js
    ProductList.js
    ProductItem.js
    ProductImage.js
    productReducer.js
user/
    User.js
    UserContainer.js
    UserActions.js
    UserProfile.js
    UserAvatar.js
    userReducer.js
```

鴨子(Ducks)

鴨子是一種模組化 Redux 的程式碼組織方法，它是把 reducers, constants, action types 與 actions 打包成模組來用。鴨子可以減少很多目錄與文章結構，例如下面的目錄:

```
|_ containers
|_ constants
|_ reducers
|_ actions
```

改用鴨子後就會變成只有兩個目錄，也就是說把 constants, reducers, actions 都合並為模組就是:

```
|_ containers
|_ modules
```

鴨子有一些優點，也有一些明顯的缺點。它在小型應用中是很理想的作法，你不用為了要加一個功能，至少需要開三、四個程式碼文章。它仍然有自訂的空間，詳細請參考這篇文章。

結論

很明顯的，並沒有最好的一種程式碼組織法，實際上認真考慮起來，每種都有它的優點與缺點。程式碼組織方法是為了人而定的，只要對共享的與要重覆使用的程式碼文章能區分得清楚，團隊間協同都能充份理解，實際上用哪一種都可以。官方的文章只有說要同時考量 actions 與 reducers，以及 selectors 與 reducers 最好在一起宣告，並沒有說一定要用哪一種或哪一種比較好。以上的組織方式都可以混用，但最好以其中一種為主要組織方式。

參考資源

- https://jaysoo.ca/2016/02/28/applying-code-organization-rules-to-concrete-redux-code/
- https://medium.com/@scbarrus/the-ducks-file-structure-for-redux-d63c41b7035c#.73rvf6tnz
- https://hackernoon.com/my-journey-toward-a-maintainable-project-structure-for-react-redux-b05dfd999b5#.2bl4xoaut
- http://marmelab.com/blog/2015/12/17/react-directory-structure.html
- https://www.reddit.com/r/reactjs/comments/47mwdd/a_better_file_structure_for_reactredux/
- http://redux.js.org/docs/faq/CodeStructure.html