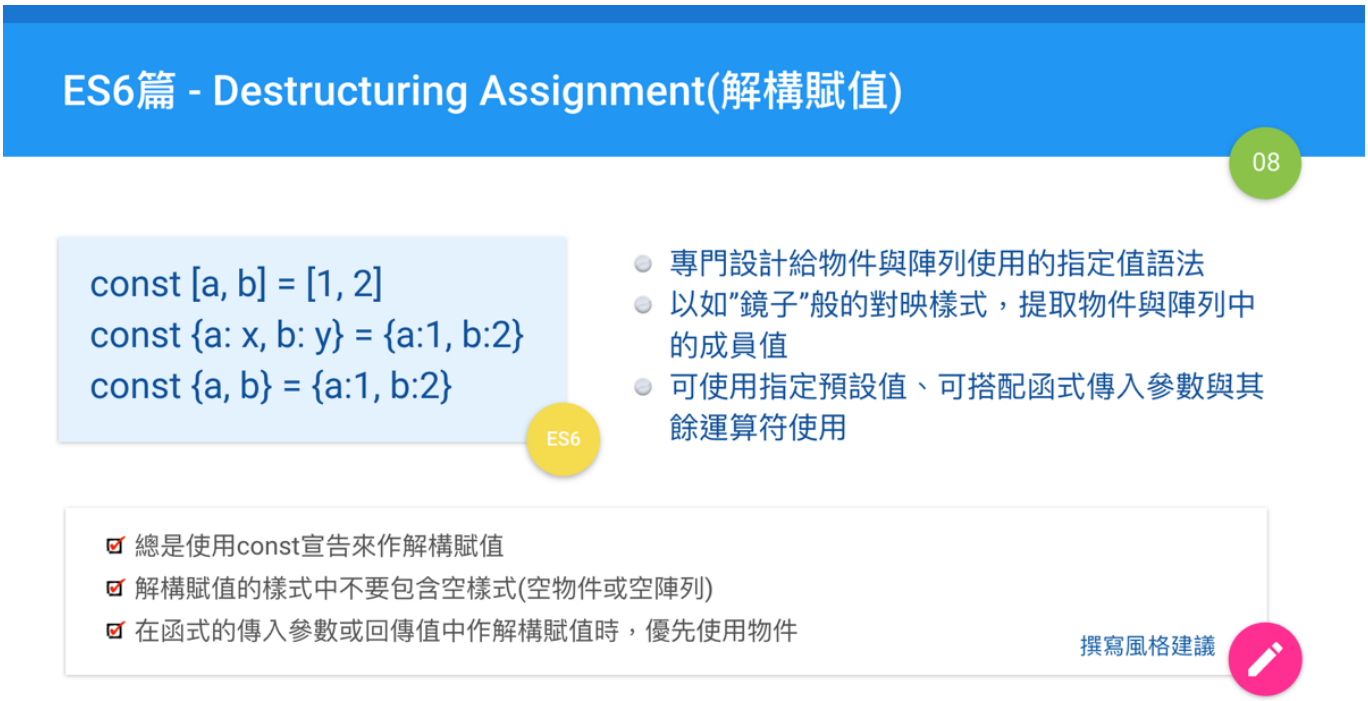
# ES6篇 - Destructuring Assignment(解構賦值)




本章的目標是對Destructuring Assignment(解構賦值)提供一些使用上的說明。這些語法在React、React Native、Redux等等新式的函式庫應用上非常的常見，是一個必學的語法。

> 註: 本文章同步放置於Github庫的這裡。

## 解構賦值介紹

"解構賦值"這個中文翻譯會是簡體中文的翻譯字詞，繁體中文通常會翻為"指定值"而不是"賦值"，因為網路上的中文翻譯常見的都是用這個翻譯，所以這裡用這個中文名詞。中文的意思為"解析結構來進行指定"。解構賦值是屬於ES6標準中指定運算的章節。

> destructuring: 變性、破壞性。使用"解構"是對照de–字頭有"脫離"、"去除"的意思。

> assignment: 賦值、指定。賦值通常指的是程式中使用等號(=)運算符的語句。

解構賦值的解說只有一小段英文，這一段是來自MDN(舊版的網站上)：

> The destructuring assignment syntax is a JavaScript expression that makes it possible to extract data from arrays or objects using a syntax that mirrors the construction of array and object literals.

這句後面的`mirrors the construction of array and object literals`，代表這個語法的使用方式 - 如同"**鏡子**"一般，對映出陣列或物件字面的結構。也就是一種樣式(pattern)對映的語法。它在處理具有多層次的巢狀的物件結構會顯得特別有用。

解構賦值(Destructuring Assignment)是一個在ES6的新特性，用於提取(extract)陣列或物件中的資料，新語法可以讓程式碼在撰寫時更為簡短與提高閱讀性。解構賦值的語法並不難，它的基本上是一種陣列與物件指

定值運算語法的簡短改進。

過去要對陣列或物件中的成員進行值的提取(extract)，要不就是使用迴圈語句，要不然就是要靠迭代的語法，在一些工具類的函式庫中，或是像TypeScript之類的超集語言，都有提供解構賦值的類似方法(通常叫作pick或pluck)或語法。ES6中內建了這種語法，所以不需要再使用這些函式庫中的方法。

解構賦值是針對陣列與物件這類的複合型資料，所設計的一種語法。在使用時有以下幾種常見的情況:

- 從陣列解構賦值
- 從物件解構賦值(或是從混用物件或陣列)
- 從非物件或非陣列解構賦值
- 解構賦值時給定預設值
- 搭配函式的傳入參數使用

解構賦值仍然是一種指定運算的語句，這一點請不要忘了。指定運算在等號(=)的左邊是要被指定的變數/常數識別名稱，右邊的則是要指定的值。不論語法看起來有多複雜，就是這種指定運算而已。

# 解構賦值的各種應用情況

## 從陣列解構賦值(Array destructuring)

從陣列解構賦值沒太多學問，唯一比較特別兩個是，第一可以用其餘運算符(Rest Operator)的語法，既然是其餘運算符，最後就會把其餘的對應值集合成一個陣列之中。第二是可以交換(Swap)陣列中的值的語法。下面是幾個幾個範例：

```
//基本用法
const [a, b] = [1, 2] //a=1, b=2

//先宣告後指定值，要用let才行
let a, b
[a, b] = [1, 2]

// 略過某些值
const [a, , b] = [1, 2, 3] // a=1, b=3

// 其餘運算
const [a, ...b] = [1, 2, 3] //a=1, b=[2,3]

// 失敗保護
const [, , , a, b] = [1, 2, 3] // a=undefined, b=undefined

// 交換值
const a = 1, b = 2;
[b, a] = [a, b] //a=2, b=1

// 多維複雜陣列
const [a, [b, [c, d]]] = [1, [2, [[[3, 4], 5], 6]]]

// 字串
const str = "hello";
const [a, b, c, d, e] = str
```

用法就是這麼簡單，用來賦值的等號符號(=)左邊按照你寫的變數或常數樣式，然後在右邊寫上要對映數值，就像之前說的"鏡子"般的樣式映對。當沒有對應位置的值時，就會被指定為`undefined`。

## 從物件解構賦值(Object destructuring)

物件除了有使用花括號({})來定義，其中也會包含屬性，物件屬性是識別名稱與值的組合。按照基本的原則，也是用像"鏡子"般的樣式映對，一樣看範例就很容易理解。這裡面有一個是超出ES6的語法(最後一個例子)，稱為Object Rest/Spread Properties：

```
// 基本用法
const { user: x } = { user: 5 } // x=5

// 失敗保護(Fail-safe)
const { user: x } = { user2: 5 } //x=undefined

// 賦予新的變數名稱
const { prop: x, prop2: y } = { prop: 5, prop2: 10 } // x=5, y=10

// 屬性賦值語法
const { prop: prop, prop2: prop2 } = { prop: 5, prop2: 10 } //prop = 5,
prop2=10

// 相當於上一行的簡短語法(Short-hand syntax)
const { prop, prop2 } = { prop: 5, prop2: 10 } //prop = 5, prop2=10

// ES7+的物件屬性其餘運算符
const {a, b, ...rest} = {a:1, b:2, c:3, d:4} //a=1, b=2, rest={c:3, d:4}
```

下面這個語法是個有錯誤的語法，這是一個常見錯誤的示範：

```
// 錯誤的示範:
let a, b
{ a, b } = {a: 1, b: 2}
```

> 註: 這個語法只能用`let`來宣告變數

因為在Javascript語言中，雖然使用花括號符號({})是物件的宣告符號，但這個符號用在程式語句中，也可以作為區塊語句的符號。當花括號({})是在語句的最開頭時，JS會當作是程式碼的區塊(block)符號來使用，這是一個常見的符號共用所造成的語法錯誤。

要修正這個語法可以在整從語句的外面框上圓括號符號(())，圓括號符號(())是群組運算符，具有控制表達式求值順序的作用，它也可以保護像這樣的語句，告訴JS這是一個完整的表達式而不是區塊語句，經過修正後正確的寫法如下：

```
let a, b
({ a, b } = {a: 1, b: 2}) //a=1, b=2
```

> 注意: 在一般情況下，你應該是在定義常數或變數的當下，就進行解構賦值。而不是拆成兩行來寫。

如果是在複雜的物件結構中，或是混合陣列在物件結構中，如果你能記住之前說的"鏡子樣式對映"原則，其實也很容易就能理解：

```
// 混用物件與陣列
const {prop: x, prop2: [, y]} = {prop: 5, prop2: [10, 100]}

console.log(x, y) // => 5 100

// 複雜多層次的物件
const {
  prop: x,
  prop2: {
    prop2: {
      nested: [ , , b]
    }
  }
} = { prop: "Hello", prop2: { prop2: { nested: ["a", "b", "c"]}}}

console.log(x, b) // => Hello c
```

## 從非陣列或非物件(原始資料類型值)解構賦值

從其他的資料類型進行陣列或物件解構，這並非這種語法設計的目的。解構賦值的語法是針對物件或陣列的資料結構所設計的，所以作這種賦值要不就是產生錯誤，要不然就是賦不到值(得到undefined)。

如果你用`null`或`undefined`這兩種當作值來指定時，會產生錯誤：

```
const [a] = undefined
const {b} = null
//TypeError: Invalid attempt to destructure non-iterable instance
```

如果是從其他的原始資料類型如布林、數字、字串等作物件解構，則會得到`undefined`值。

```
const {a} = false
const {b} = 10
const {c} = 'hello'

console.log(a, b, c) // undefined undefined undefined
```

從其他的原始資料類型布林、數字、字串等作陣列解構的話。唯一的例外只有字串類型的值可以解構出單字元的字串值，其他也是得到`undefined`值：

```
const [a] = false
const [b] = 10
const [c] = 'hello' //c="h"

console.log( a, b, c)
```

上面會有出現這樣的結果，是當一個值要被進行解構前，在這第一個階段時，它會先被轉成物件或陣列。因為`null`或`undefined`無法轉成物件(或陣列)，所以必定產生錯誤。下一個階段如果這個值轉換的物件或陣列，沒有附帶有對應的迭代器(Iterator)就無法被成功解構賦值，所以最後會回傳`undefined`。

> 註: 字串資料類型的值可以解出來指定給陣列，是因為在JS中的內部設計，一個字串值相當於多個單一字元的字串組成的陣列，字串有很多特性與陣列會很類似。只能用在陣列的解構賦值，無法用在物件的解構賦值。

## 解構賦值時的預設值

在等號(=)左邊的樣式(pattern)中是可以先給定預設值的，這是作為如果沒有賦到值時，也就是要對應的值不存在時的預設值。下面為例子:

```
const [missing = true] = []
console.log(missing)
// true

const { message: msg = 'Something went wrong' } = {}
console.log(msg)
// Something went wrong

const { x = 3 } = {}
console.log(x)
// 3
```

下面是個陷阱頭目，你可以試看看下面這個範例中到底是賦到了什麼值：

```
const { a ='hello' } = 'hello'
const [ b ='hello' ] = 'hello'

console.log( a, b)
```

預設值的觸發情況與之前在"傳入參數預設值"的章節說的情況有些類似，也要對照上一節的"從非陣列或非物件(原始資料類型值)解構賦值"說明，要觸發預設值必須是賦值時對應值是undefined或不存在的情況，但如果直接把整個左邊的樣式指定到一個`null`或`undefined`值時，會直接產生錯誤，而不是觸發預設值。

## 在函式傳入參數定義中使用

在函式傳入參數定義中也可以使用解構賦值，因為函式的傳入參數本身也有自己的預設值指定語法，這是ES6的另一個特性，所以使用上非常容易與解構賦值本身的預設值設定搞混。這地方會產生不少陷阱。

一個簡單的解構賦值用在函式的參數裡，這是很容易看得懂的語法：

```
function func({a, b}) {
  return a + b
}

func({a: 1, b: 2}) // 3
```

當你用上了解構賦值預設值的語句，而且只前面的a屬性有預設值，後面的b就沒有，這時候因為b沒有賦到值時，b會是`undefined`，任何數字加上`undefined`都會變成`NaN`：

```
function func({a = 3, b}) {
  return a + b
}

func({a: 1, b: 2}) // 3
func({b: 2}) // 5
func({a: 1}) // NaN
func({}) // NaN
func() // TypeError: Cannot match against 'undefined' or 'null'
```

上例中最後一個`func()`的呼叫明顯會產生錯誤，因為它相當於`let {a = 3, b} = undefined`的語句，這語句會產生錯誤。

在下面的例子中，當a與b兩個都有預設值時，`NaN`的情況不存在：

```
function func({a = 3, b = 5}) {
  return a + b
}

func({a: 1, b: 2}) // 3
func({a: 1}) // 6
func({b: 2}) // 5
func({}) // 8
func() // TypeError: Cannot match against 'undefined' or 'null'
```

上例中最後一個`func()`的呼叫明顯會產生錯誤，因為它相當於`let {a = 3, b = 5} = undefined`，也會產生錯誤。

實際上函式傳入參數自己也可以加預設值，如果再加上傳入參數的預設值，下面的例子會讓最後一種`func()`呼叫時與`func({})`呼叫有相同結果：

```
function func({a = 3, b = 5} = {}) {
  return a + b
}
```

```
func({a: 1, b: 2}) // 3
func({a: 1}) // 6
func({b: 2}) // 5
func({}) // 8
func() // 8
```

> 特別注意: 這個樣式很常見，在函式傳入參數預設值使用空物件算是一種保護性語法。

另一種使用傳入參數預設值的情況，是在傳入參數預設值中給了另一套預設值，但它只會在`func()`時發揮預設值指定的作用：

```
function func({a = 3, b = 5} = {a: 7, b: 11}) {
  return a + b
}

func({a: 1, b: 2}) // 3
func({a: 1}) // 6
func({b: 2}) // 5
func({}) // 8
func() // 18
```

你可以觀察一下，當對某個變數賦值時你給他null或undefined，到底是用預設值還是沒有值，這個範例的`g()`函式是個對照組：

```
function func({a = 1, b = 2} = {a: 10, b: 20}) {
  return a + b
}

func({a: 3, b: 5}) // 8
func({a: 3}) // 5
func({b: 5}) // 6
func({a: null}) // 2
func({b: null}) // 1
func({a: undefined}) // 3
func({b: undefined}) // 3
func({}) // 3
func() // 30
```

```
function g(a = 1, b = 2) {
  return a + b
}

g(3, 5) // 8
g(3) // 5
g(5) // 7
g(undefined, 5) // 6
```

```
g(null, 5) // 5
g() // 3
```

> 註：所以在作解構賦值時，給定null值時會導致預設值無用，請記住這一點。當數字運算時，null會轉為數字0。

## React中的實例應用

學再多的知識也比不上從真實的案例中來看這個語法的用處。真實的情況是在React或React Native的程式碼中，處處可見解構賦值的語法，這幾列出幾個例子與簡單的解說。如果你能真正看得懂這些語法的功用，相信你的React能力一定會更加提升。

因為怕混用各種特性語法會造成初學者一開始學習時的解讀困難，所以我先把搭配其餘運算符的部份留在之後的章節再說明。

### 從元件中解構出其他的物件值

以下的程式碼來自React Native的官方範例:

```
const NavigationExampleRow = require('./NavigationExampleRow');
const React = require('react');
const ReactNative = require('react-native');

// 這是取得ReactNative其中包含內建模組的一種語法
const {
  NavigationExperimental,
  ScrollView,
  StyleSheet,
} = ReactNative;

// 這是取得NavigationExperimental其中所包含的
// NavigationCardStack與NavigationStateUtils物件的語法
const {
  CardStack: NavigationCardStack,
  StateUtils: NavigationStateUtils,
} = NavigationExperimental;
```

從本文上面的說明對照這裡來看，其實一點都不難，只是名稱長了點，而且都是識別名稱(物件名)。物件的解構賦值容易讓人搞混，是因為它有簡寫法，這裡的例子剛好用了這兩種，一種是正常的樣式對映，另一種是簡寫法。

一般正常的用法像下面這樣，所以要被賦值的識別名(變數名)是在左邊樣式中的以"屬性對應的值"來代表:

```
const obj = {a: 1, b: 2}
const {a: x, b: y} = obj //x=1, y=2
```

上述的第二個解構賦值語法，就是這種正常型的用法，所以它提取到的是NavigationCardStack與NavigationStateUtils這兩個識別名，在這個程式檔案中就可以使用這兩個識別名，這兩個對應到的是NavigationExperimental元件(模組)裡的物件值。

下面的例子是一種簡寫法，此時只能在左邊的物件樣式中，寫要對應的物件值中的屬性名稱，會變為有點類似於把物件中的屬性直接提取出來為另一個變數名:

```
const obj = {a: 1, b: 2}
const {a, b} = obj //a=1, b=2

//上面的語句相當於下面的寫法
const {a: a, b: b} = obj
```

上述的第一個解構賦值語法，這是一般提取React Native中元件模組的方式，都是用簡寫法的解構賦值。React Native中的所有包含的元件模組定義程式檔，應該是在函式庫的這個檔案之中。

> 註: 上述的簡寫法可以這樣寫，是因為ES6中有另一個新特性，稱為"Object Literal Property Value Shorthand"(物件字面屬性值簡寫)

### 從props與state解構賦值

React的props或state裡面都有可能是巢狀的物件結構，使用解構賦值的確可以很容易的提取出裡面的屬性值，所以很常會看到類似像下面的程式碼(出自這篇文章):

```
class DataModal extends Component {
  render() {
    const { modalList, location: { pathname } } = this.props
    const { currIndex, showModal } = this.state
    // ..
  }
}
```

這個例子就不多說明了，這是很一般的物件解構賦值語法而已。主要是你在等號(=)右邊的物件值裡的結構要是等號左邊樣式的這樣，才能正確指定到屬性值。

### 使用於函式的傳入參數之中的解構賦值

在React中的以函式定義的方式來撰寫元件時(這種為無狀態元件)，經常會看到在傳入參數中使用解構賦值的語法，例如下面的例子(出自這篇文章):

```
function DataProcess({ onClickButton, stepCurrent }) {
  return (
    <div>
      <h2>Step {stepCurrent}</h2>
      <Button onClick={onClickButton}>Click</Button>
    </div>
```

```
  )
}
```

實際上它是一種解構賦值的語法，作為元件的函式通常是傳入的是props值。所以相當於以下的語法:

```
function DataProcess(props) {

  const onClickButton = props.onClickButton
  const stepCurrent = props.stepCurrent

  return (
    <div>
      <h2>Step {stepCurrent}</h2>
      <Button onClick={onClickButton}>Click</Button>
    </div>
  )
}
```

所以，如果你在函式的傳入參數中，使用了物件的樣式結構作為傳入參數的定義，當傳入一個物件值時，就會進行解構賦值。之後在函式區塊中，直接可以使用由這個樣式結構提取到的傳入物件中的屬性值。

## 撰寫風格建議

- 解構賦值時，優先使用const來宣告。(eslint: prefer-const)
- 解構賦值的樣式中不要包含空樣式(空物件或空陣列)。(eslint: no-empty-pattern)
- 在函式的傳入參數或回傳值中，要作解構賦值時，優先使用物件，而不要使用陣列。(Google 5.2.4, Airbnb 5.3)

## 結論

解構賦值是一個重要的ES6語法，它簡化了以往對陣列與物件結構中的成員值的提取工作，讓程式碼變得更清晰容易閱讀。當然，愈簡化的語法，對初學者來說可能會愈看不懂，你一定要先掌握住基本的使用原則以及基本的作用，這樣在不論是多複雜的物件結構，或是在合併使用了多種新式語法特性時，都能看得懂程式碼中的意義，而進一步能夠運用自如。

## 參考資料

- Several demos and usages for ES6 destructuring.
- Destructuring Assignment in ECMAScript 6
- Destructuring assignment MDN
- Destructuring assignmentのご利用は計画的に