Async 函式

前言

處理 JavaScript 中的異步程序總是需要面臨到各種不同應用情況的挑戰，在 ES6(ES2015)中所加入的 Promise 以及 Generator，讓新版本的 JavaScript 多了一些工具和語法，可以更方便的來作這些異步的程序的流程控制。但這些新的工具或結構，或許又讓開發者有了一些新的問題，例如下面幾項:

- 語法複雜，需要進一步學習與大幅度修改原有的程式碼
- 除錯不容易
- 錯誤處理不容易
- 對於情況控制(Conditionals)與迴圈/迭代處理仍然不是很理想

async 函式在 ECMAScript 2017(ES8) 後加入了標準之中，其目的非常的明確，就是為了要解決上述的問題，它可以更進一步的精簡整個語法。

async/await 的語法十分容易學習與使用，相較於 Promise 或 Generator 而言，開發者可以在很短的時間內理解用法，以及開始使用它們，甚至不需要對 Promise 或 Generator 有太深入的知識。

不過，async 函式的基礎仍然是 Promise，它的相對轉換的語法則是組合了 Promise 與 Generator，雖然它提供了更方便使用的語法，能夠很輕易的處理這些異步程序的流程控制，不過我們仍然需要對 Promise 有一定的理解，才能真正靈活地應用到各種情況中，說明白一點，Promise 的知識你仍然需要有的，沒有什麼誰取代誰的問題。

async 函式

一開始我們先以一個最簡單的 Promise 的範例來說明，我們使用 Fetch API 向伺服器要求待辦事項的資料，然後更新目前應用中的列表，程式碼範例如下:

```
fetch('http://example.com/items')
  .then(response => response.json())
  .then(data => {
    updateView(data)
  })
  .catch(error => {
    console.log('Update failed', error)
  })
```

Promise 的語法結構的涵意是 "我想要進行這個操作，然後(then)在下一步對操作得到的資料再進行處理" 這是一種連鎖語法的結構。

使用 async/await 來修改上面的範例，會變為下面這樣的程式碼，你可以注意到 then 已經不存在這個程式碼中:

```
const response = await fetch('http://example.com/items')
const data = await response.json()
updateView(data)
```

await 的語法涵意會是 "我想要得到這個操作的結果(值)"，這會感覺像是在撰寫同步的語句。

由於 await 運算子是被設計來等待 Promise 的，它只能在 async 函式 內使用。所以我們需要把它放在一個 async 函式 之中，像下面這樣的程式碼:

```
async function updateMyView() {
  const response = await fetch('http://example.com/items')
  const data = await response.json()
  updateView(data)
}
```

至於最一開始 Promise 裡的 catch 方法，可以用來作錯誤處理，在 async/await 語法裡，要使用 try/catch 來取代它，如以下的程式碼:

```
async function updateMyView() {
  try {
    const response = await fetch('http://example.com/items')
    const data = await response.json()
    updateView(data)
  } catch (error) {
    console.log('Update failed', error)
  }
}
```

當然，你也可以用 IIFE 或是箭頭函式的語法來搭配 async 函式，這可以因應不同的使用情況，以及讓語法更為簡化，像下面的程式碼:

```
;(async () => {
  try {
    const response = await fetch('http://example.com/items')
    const data = await response.json()
    updateView(data)
  } catch (error) {
    console.log('Update failed', error)
  }
})()
```

當然，由於 async 函式呼叫後會返回一個 Promise，你也可以使用原本的 catch 方法來作錯誤處理，像下面這樣的程式碼:

```
async function updateMyView() {
  const response = await fetch('http://example.com/items')
  const data = await response.json()
  updateView(data)
}

updateMyView().catch(error => {
  console.log('Update failed', error)
})
```

由上面的改寫範例，你可以看到我們雖然是在撰寫異步的程序，但寫出來的程式碼卻是像是在撰寫同步的程序。有被加上 await 的語句，會等待到該語句執行的結果得到後，才會接著處理下一步的程序語句。這使得整體的程式碼可閱讀性提高了，而且也變得更容易除錯。

> 註: 上面的例子可以使用 catch 方法，當然也可以使用 then 方法

> 註: await 不能在全域(頂級)作用域直接使用，它一定要在 async 函式內才能使用

async 函式的語法說明

根據 MDN 中有關於async 函式的說明:

> 當 async 函式被呼叫時，它會回傳一個 Promise，這與有沒有使用 await 無關

async 函式的語法即是使用 async 作為函式的修飾關鍵字詞，所宣告的函式，語法如下,，出自 MDN:

```
async function name([param[, param[, ... param]]]) {
   statements
}
```

async 函式被呼叫後，如果回傳一個值，就會被視為帶有該回傳值的實現(resolved)狀態的 Promise，反之如果拋出例外，就會被視為帶有被拋出值的拒絕(rejected)狀態的 Promise。所以這 async 函式呼叫的結果，與 Promise 兩者之間有互相對應的關係。

函式宣告有加上 async 關鍵字和沒加上是兩回事，async 函式在呼叫後會回傳一個 Promise，如下面的範例程式碼:

```
// Normal function
function func() {
  return 1
}

console.log(func()) // 1
```

```
// Async function
async function asyncFunc() {
  return 1
}

console.log(asyncFunc()) // Promise {<resolved>: 1}
```

上面的 async 函式相當於以下的兩種寫法，一樣是具有相同已實現內容的 Promise，範例程式碼如下:

```
// 原本的Async function
async function asyncFunc() {
  return 1
}

// 使用Promise.resolve
async function asyncFunc1() {
  return Promise.resolve(1)
}

// 使用new Promise
async function asyncFunc2() {
  return new Promise((resolve, reject) => {
    resolve(1)
  })
}

asyncFunc().then(console.log)
asyncFunc1().then(console.log)
asyncFunc2().then(console.log)
```

這當然是因為 async 函式可以包裹住在其中非 Promise 的程式語句，最後回傳一個 Promise。但這裡面仍然有一個小小的差異，上面範例中的 asyncFunc 函式，它的回傳是 `Promise {<resolved>: 1}` ，也就是已經實現的(resolved 或 fulfilled)的 Promise 狀態，但後面兩個回傳 Promise 的 async 函式，它們的 Promise 狀態都還是 `Promise {<pending>}` ，也就是說還沒真正固定狀態到已實現或已拒絕，需要再下一步的 then 接上去後，才會進行解析。不過，在最後的執行結果並無差異，這只是一個過程中的小細節而已。

async 函式會轉變成為一個異步的函式，但它與原本的 JavaScript 中函式裡的建構式不同，是使用新的 AsyncFunction 作為建構式來建立函式物件，也就是說它與原本的函式物件的結構並不相同。

async 函式的語法可以用於各種函式宣告的語法，沒有什麼限制或問題，例如以下幾個:

- 函式定義(FD)
- 函式表達式(FE)
- 箭頭函式
- 物件字面定義內的物件方法
- 類別定義中的方法

> 註: 在 JavaScript 中的函式主體內，最後沒有寫 return 回傳值，相當於 `return undefined` ，也算是有回傳值。

await 運算子

await 是一個運算子，使用這個關鍵字在表達式前作為修飾關鍵字詞，會讓表達式變為 "等待 Promise 解析的表達式"，await 只能在 async 函式內使用，不能在一般的函式內使用。

await 的語法如下，出自 MDN 這裡:

```
[rv] = await expression
```

> [rv] 回傳 Promise 物件的 resolved 值，如果當值不是 Promise 物件時，則會回傳該值本身。

await 是設計用來等待 Promise 的，因此在其後加上非異步的(同步)的表達式，並沒有什麼意義，就直接回傳它的值而已，對於 Promise 的最後狀態有下面兩種情況:

- 如果最後 Promise 是已實現的(fulfilled)狀態，await 的結果是這個已實現的(fulfillment)值

- 如果是 Promise 是已拒絕的(rejected)狀態，await 會拋出已拒絕的理由(reason)

await 會讓 JavaScript 程式進行等待(暫停)，等到後面接著表達式的 Promise 的狀態已經固定了(settled)，然後回傳這個 Promise 的 resolved(已實現) 值，才會再進行下一個語句，要注意這裡得到的是值(Promise 的已實現值，如果是已實現狀態的話)，而不是像 Promise 結構中一個可以往下傳的另一個 Promise 物件。

多個 await 表達式組合，是一種"順序操作(sequential)"的執行程序，就像是同步語句組合的語法，但實際上是非阻塞的、異步執行的語句組合。

`await 表達式` 也是一個表達式，它可以像一般的表達式在各種語法中使用，當然也這一定是在要 async 函式之中。

> await 這個英文字詞也是有 "等待、等候" 意思的動詞，它與另一個常用的英文字詞 wait 意思相近。wait 後面通常會加上 for，經常使用在等候某人、等公車、等聖誕老公公...等等，它也可以不需要加上後面的受詞。 await 則會用於比 wait 較為正式的場合或書面文章，它有預期某事物會延時發生的意思，後面必定要加上受詞，經常用於例如等候法院判決、等候合約審查、等候某人作決定...等等情況。

瀏覽器相容議題

根據Async functions - Can I Use在 2019/1 的資料，目前市場上大約有 85%的瀏覽器是原生支援 async 函式功能，由於 async 函式是基於 Promise 與 Generator 組合而成的、類似語法簡化的功能，如果瀏覽器原本就已經有支援上述兩種功能，要再加入 async 函式功能是相對容易的，因此主要的瀏覽器品牌在很早之前就推出實驗性質的、或是部份支援的功能。以下是各瀏覽器與 Node 的支援情況:

- Chrome v55(發佈 2016/12) - Google 瀏覽器的 V8 引擎在很早之前就開始支援 async 函式功能(註: V8 release v5.5, 2016/10)
- Node v7.6 (發佈 2017/2/22) - 不過 LTS 版本則是指 v8.0.0 (發佈 2017/5)後才正式支援
- Firefox v52 (發佈 2017/3)相關新聞或說明
- Safari v10.1 (發佈 2017/3)。相關新聞或說明
- Edge v15
- Opera v42

> 註: IE 瀏覽器迄今完全沒有支援，需進行填充或編譯工具先行編譯

> 註: Async 函式的標準內容是包含在 ECMAScript 2017(通常稱為 ES2017 或 ES8)之中，最終定案是在 2017/6。因此，上述的瀏覽器都是搶先實作這個功能。

babel 編譯工具需要使用 preset-es2017 ，使用的外掛是 plugin-transform-async-to-generator才能正確地編譯 async 函式向下相容至 ES5 的瀏覽器執行引擎環境(IE9 以上)。

> 註: IE8 是 ES3 的執行環境，仍然可以透過像 es5-shim進行填充來執行。

參考資源

- 14.6 Async Function Definitions - ECMAScript 2017