Web Programming 05/16

* Asynchronous Functions
* Callback Hell
* Promise
* Fetch
* Async / Await

追求現代化的網頁瀏覽體驗

- 每個人看到的網頁內容不同
- 單一網頁可以快速載入大量的資訊來源
- 利用標籤來切換部分網頁內容時不會整個白螢幕
- 網頁還沒有開啟完畢就可以與其中一些元素互動
- 各種比例的蓋板視窗、跳出視窗、提示標籤... 等

Synchronous?Asynchronous?

- 大家熟悉的電腦語言像是 C/C++, 當它在做一些計算的時候，你對它進行 I/O 他是不會理你的。但像是有 JS 的網頁，你卻常常可以一邊放音樂一邊在下面按讚。
- JavaScript is asynchronous? Try this:

```
function waitThreeSeconds(){
  var ms = 3000 + new Date().getTime();
  while(new Date() < ms) {}
  console.log("finished function");
}
function clickHandler() { console.log("click event!”); }
document.addEventListener('click', clickHandler);
console.log("started execution");
waitThreeSeconds();
console.log("finished execution");
```

Ref: https://www.udemy.com/understand-javascript/?siteID=je6NUbpObpQ-vRvfcTs4jcQ1bRNhRpaPtA&LSNPUBID=je6NUbpObpQ

Synchronous?Asynchronous?

- 事實上，JS Engine 跟一般的 C/C++ program 一樣都是 synchronous, 是 browser 用 asynchronous 的方式來 handle 從 JS code 裡面送出來的各種 asynchronous I/O, functions, etc.
  - e.g. setTimeout(foo, 1000);
  - e.g. document.addEventListener('click', clickHandler);
- 事件/Function 之間的互動因為 asynchronous 會變成十分的複雜
  - 一個程式按照此順序執行：“A -> X -> B -> Y -> C - > D”，但當 X, Y 是 asynchronous 且回覆的先後不一定時，你就得要去考慮各種排列組合對於後續程式的副作用

Asynchronous in JS is necessary

- When you run this code, 50 is returned !

```
var result = multiplyTwoNumbers(5, 10)
console.log(result)
// 50 gets printed out
```

- But what will be returned for this call?

```
var photo = downloadPhoto('http://coolcats.com/cat.gif')
// photo is 'undefined'!
```

Ref: http://callbackhell.com/

We need “callback” functions

- “handlePhoto” will be called (back) when download is completed, or error happens.

```
downloadPhoto('http://coolcats.com/cat.gif', handlePhoto)

function handlePhoto (error, photo) {
  if (error) console.error('Download error!', error)
  else console.log('Download finished', photo)
}

console.log('Download started')
```

// Note: Usually property like “photo.isReady” is set to signify the download completion

Callback function can be anonymous

- Since callback function is usually called just once, people tend to make it anonymous:
  - ```
    var src = 'http://coolcats.com/cat.gif';
    downloadPhoto(src, function(error, photo) {
      if (error)
        console.error('Download error!', error)
      else console.log('Download finished', photo)
    }
    console.log('Download started')
    ```
- This is plausible, and seems straightforward, as the code is written from top to down as how it should be executed intuitively.
- However, this may cause “callback hell”!
    ```

Callback Hell in JavaScript

callbackhell.js

```
var floppy = require('floppy');

floppy.load('disk1', function (data1) {
    floppy.prompt('Please insert disk 2', function() {
        floppy.load('disk2', function (data2) {
            floppy.prompt('Please insert disk 3', function() {
                floppy.load('disk3', function (data3) {
                    floppy.prompt('Please insert disk 4', function() {
                        floppy.load('disk4', function (data4) {
                            floppy.prompt('Please insert disk 5', function() {
                                floppy.load('disk5', function (data5) {
                                    floppy.prompt('Please insert disk 6', function() {
                                        floppy.load('disk6', function (data6) {
                                            //if node.js would have existed in 1995
                                        });
                                    });
                                });
                            });
                        });
                    });
                });
            });
        });
    });
});

```


Ref: http://blog.mclain.ca/tag/nintendo/

Conventional Way to Fix Callback Hell

1. Keep your code shallow
2. Modularize
3. Handle every single error

Ref: http://callbackhell.com/

Conventional Way to Fix Callback Hell
1. Keep your code shallow

- This example uses browser-request to make an AJAX request to a server. However, this style may goes into deep callback hell.

```
var form = document.querySelector('form')
form.onsubmit = function (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

Conventional Way to Fix Callback Hell 1. Keep your code shallow

- To fix it, first name the functions and take advantage of function hoisting to keep the code shallow.

```
document.querySelector('form').onsubmit = formSubmit

function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse (err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err
  statusMessage.value = body
}
```

Conventional Way to Fix Callback Hell 2. Modularize

- Secondly, create JavaScript modules — by re-organizing functions into smaller files.

- The above example can be simplified as below when the callback functions are defined in another files.
/ / “formuploader.js” is where the modules are defined.

```
var formUploader = require('formuploader')
document.querySelector('form').onsubmit = formUploader.submit
```

Conventional Way to Fix Callback Hell 2. Modularize

- With the “formuploader.js” defined as:

```
module.exports.submit = formSubmit

function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse (err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err
  statusMessage.value = body
}
```

Conventional Way to Fix Callback Hell 3. Handle every single error

- Note: When dealing with callbacks, you are by definition dealing with tasks that get dispatched, go off and do something in the background, and then complete successfully or abort due to failure.

- What happens if any of them encounters errors? What happens if you don't handle the error and continue executing the callback functions?

- Important: Apply the Node.js style where the first argument to the callback is always reserved for an error.

```
var fs = require('fs')

fs.readFile('/Does/not/exist', handleFile)

function handleFile (error, file) {
  if (error) return console.error('Uhoh, there was an error', error)
  // otherwise, continue on and use `file` in your code
}
```

Introduction to “Promise”

- 傳統用 callback 來作為 asynchronous functions 的回應方法很容易讓 code 變得很混亂、破碎，且不容易將不同非同步事件之間的關係描述得很乾淨

- “Promise” 就是為了讓在 JavaScript 裡頭的 asynchronous 可以 handle 的比較乾淨
  - Promise 物件：定義好一個 asynchronous function 執行之後成功或是失敗的狀態處理
  - 使用情境：將 Promise 物件的 .then(), .catch() 加入此 asynchronous function 應該要被呼叫的地方
  - Note: ES6 已經把 promise.js 納入標準

“Promise” Example

- 想像一下你是個孩子，你媽承諾(Promise)你下個禮拜會送你一隻新手機。

- 現在你並不知道下個禮拜你會不會拿到手機。你媽可能真的買了新手機給你，或者因為你惹她不開心而取消了這個承諾

- 為了把這段人生中小小的事件定義好，你將問了媽媽以後會發生的各種情況寫成一個 JavaScript function:
  - var askMom = function willIGetNewPhone() { … }
    askMom();

Ref: https://andyyou.github.io/2017/06/27/js-promise/

“Promise” Example

- 基本上一個像是 willIGetNewPhone() 的非同步事件會有三種狀態:
  - Pending: 未發生、等待的狀態。到下週前，你還不知道這件事會怎樣。
  - Resolved 完成 / 履行承諾。你媽真的買了手機給你。
  - Rejected 拒絕承諾。沒收到手機，因為你惹她不開心而取消了這個承諾。
- 我們將把 willIGetNewPhone() 改宣告一個 Promise 物件，來定義上面三種狀態。

Ref: https://andyyou.github.io/2017/06/27/js-promise/

“Promise” Example

- 用 Promise 來定義 willIGetNewPhone()
  - **var isMomHappy = …;**
    var willIGetNewPhone

```
= new Promise(function (resolve, reject) {
    if (isMomHappy) {
      var phone = { … }; // your dream phone
      resolve(phone);
    } else {
      var reason = new Error('Mom is unhappy’);
      reject(reason);
    }
  });
```

Ref: https://andyyou.github.io/2017/06/27/js-promise/

“Promise” Example

- 定義 Promise 物件的語法：

```
new Promise(function (resolve, reject) { … });
```

  - 如果一個 Promise 執行成功要在內部 function 呼叫 resolve (成功結果)，如果結果是失敗則呼叫 reject (失敗結果)。
  - 一個 Promise 物件表達的是一件非同步的操作最終的結果，可以是成功或失敗。

Ref: https://andyyou.github.io/2017/06/27/js-promise/

“Promise” Example

- 用 Promise 定義好 willIGetNewPhone() 之後，我們就可以來改寫 askMom() —

```
var askMom = function () {
    willIGetNewPhone
        .then(function (fulfilled) {
            …
        })
        .catch(function (error) {
            …
        });
};
askMom();
```

- 在 .then 中使用 function (fulfilled){}，而這個 fulfilled 就是從 Promise 的 resolve(成功結果) 傳來的結果，範例中這個結果就是 phone 物件。
- 在 .catch 中我們使用了 function (error) {}。而這個 error 就是從 Promise 的 reject(失敗結果) 傳來的，即 reason。

Ref: https://andyyou.github.io/2017/06/27/js-promise/

Putting them together…

- 把 asynchronous function 包成 Promise 物件，
  用 .then() 與 .catch() 來處理成功與失敗的結果。
  - ```
    const handleResolved = () => {}
    const handleRejected = () => {}
    new Promise((resolve, reject) => {
      try {
        const data = getData();
        resolve(data);
      } catch (err) {
        reject(err);
      }
    })
    .then(handleResolved)
    .catch(handleRejected);
    ```

Ref: https://andyyou.github.io/2017/06/27/js-promise/
    ```

Asynchronous Functions (Promise) 的串連

- 假設有一連串的 asynchronous functions 要執行，可以把他們的 Promise 串聯起來 —

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => {…})
.catch(failureCallback);
```

- 其中 doSomething(), doSomethingElse(), doThirdThing() 就是用 Promise 包起來的三個 asynchronous functions

Ref: https://andyyou.github.io/2017/06/27/js-promise/

.catch() 後面還是可以接東西

- See this example —
  - ```
    new Promise((resolve, reject) => {
      console.log('Initial');
      resolve();
    }).then(() => {
      throw new Error('Something failed');
      console.log('Do this');
    }).catch(() => {
      console.log('Do that');
    }).then(() => {
      console.log('Do this whatever happened before');
    });
    ```
- Output —

  ```
  Initial
  Do that
  Do this whatever happened before
  ```

Ref: https://andyyou.github.io/2017/06/27/js-promise/
  ```

Promise.all()

- p1 & p2 都要成功
  - ```
    const p1 = new Promise((resolve, reject) => {
      setTimeout(resolve, 500, "one");
    });
    const p2 = new Promise((resolve, reject) => {
      setTimeout(resolve, 100, "two");
    });
    Promise.all([p1, p2]).then((value) => {
      console.log(value);
    });
    ```
  - Output
    Array ["one", "two"]

Ref: https://andyyou.github.io/2017/06/27/js-promise/
    ```

Promise.race()

- 看誰先成功
  - ```
    const p1 = new Promise((resolve, reject) => {
      setTimeout(resolve, 500, "one");
    });
    const p2 = new Promise((resolve, reject) => {
      setTimeout(resolve, 100, "two");
    });
    Promise.race([p1, p2]).then((value) => {
      console.log(value);
      // Both resolve, but p2 is faster
    });
    ```
  - Output
    two

Ref: https://andyyou.github.io/2017/06/27/js-promise/
    ```

Promise 好好學，寫 js 沒煩惱。

– ChenTsu Lin

一個很有用的 Promise — fetch()

- 有聽過/用過 XMLHttpRequest() 嗎?
  - 在 IE7 (2006) 年就有的 API
  - 命名有些問題
  - API 太過低階
  - e.g.

```
function reqListener () {
  console.log(this.responseText);
}
var oReq = new XMLHttpRequest();
oReq.addEventListener("load", reqListener);
oReq.open("GET", "http://www.example.org/example.txt");
oReq.send();
```

- 用 fetch() 取代 XMLHttpRequest() —> 回傳 Promise()

Ref: https://andyyou.github.io/2017/06/27/js-promise/

一個很有用的 Promise — fetch()

- 使用 fetch() 會容易很多！
  - ```
    fetch("https://www.google.com")
    .then((res)=>console.log(res.status))
    .catch((err) => console.log('Error :', err));
    // output: 200
    ```
- 常用的 res.text() res.json() 都是回傳Promise()
  - ```
    fetch("https://www.google.com")
    .then((res)=>console.log(res.text()))
    .catch((err) => console.log('Error :', err));
    // output a Promise()
    ```

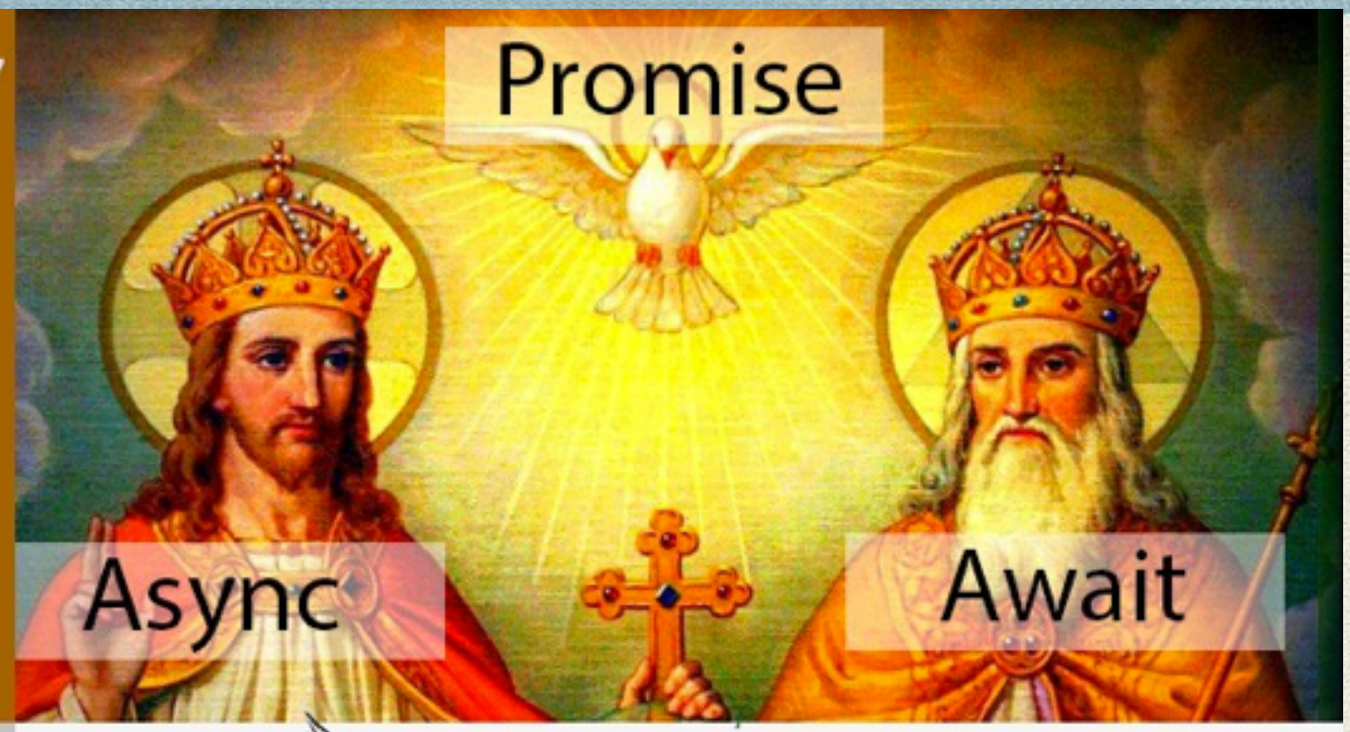
Ref: https://andyyou.github.io/2017/06/27/js-promise/

用 fetch() 送 POST

- fetch(url, object) —
  - ```
    fetch(url, {
      method: 'post',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        name: 'Hubot',
        login: 'hubot',
      }),
    });
    ```
- method: 可以是 get, post, put, delete
- headers: HTTP request header
- body: JSON.stringify 轉成 JSON 字串

Ref: https://andyyou.github.io/2017/06/27/js-promise/
    ```

ES7



ES6

Promise

ES5

Ref: https://medium.com/@peterchang_82818/javascript-es7-async-await-%E6%95%99%E5%AD%B8-703473854f29-tutorial-example-703473854f29

Can I use “async/await”?

- 雖然 ES7 原生支援 async/await, 但每個瀏覽器的支援程度不同，使用前還是要先查一下，否則就要用 Babel.
- node.js 要 7.6 版本之後才有支援

Async functions - OTHER

Usage: Global — % of all users: 84.51%

Async function make it possible to treat functions returning Promise objects as if they were synchronous.

Current aligned | Usage relative | Date relative | Show all

IE	Edge *	Firefox	Chrome	Safari	iOS Safari *	Opera Mini *	Chrome for Android	UC Browser for Android	Samsung Internet
			49						
			64		10.3				
	16	59	65	11	11.2				4
11	17	60	66	11.1	11.3	all	66	11.8	6.2
	18	61	67	TP					
		62	68						
			69						

Notes | Known issues (0) | Resources (6) | Feedback

啥？現看個例子...

- Promise
  - ```
    Doctor.findById()
      .then(doctor => {
        // some logic
        doctor…
        return somePromise;
      })
      .then(…);
    ```
- Async/Await
  - ```
    const f = async () => {
      const doctor = await Doctor.findById();
      // some logic
      doctor…
    }
    f();
    ```

Async/Await

- Async/Await 的目的在讓 Promise 概念的使用更直覺
  1. 在 top-level function 前面加個 async

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

  2. 用 await 把原先非同步結束後才要在 .then() 執行的程式碼直接隔開，平行的放到下面

```
async function name([param[, param[, ... param]]]) {
  let ret = await doSomethingTakesTime();
  // this part does not run until await finishes
}
```

  3. 基本上 async 匡了一個 top-level async 的區域，然後裡面透過 await 變成是 synchronous

更多 Async / Await 的例子

- Simple case

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}


async function add1(x) {
  const a = await resolveAfter2Seconds(20);
  const b = await resolveAfter2Seconds(30);
  return x + a + b;
}
```

```
add1(10).then(v => {
  console.log(v);  // prints 60 after 4 seconds.
});


async function add2(x) {
  const p_a = resolveAfter2Seconds(20);
  const p_b = resolveAfter2Seconds(30);
  return x + await p_a + await p_b;
}

add2(10).then(v => {
  console.log(v);  // prints 60 after 2 seconds.
});
```

Ref: https://developer.mozilla.org/zh-TW/docs/Web/JavaScript/Reference/Statements/async_function

Async / Await 的 Error Handling

```
1   const asyncRunFail = async () => {
2     let mingRun = await runPromise('小明', 2000, false);
3     let auntieRun = await runPromise('漂亮阿姨', 2500);
4     return `${mingRun}, ${auntieRun}`
5   }
6   asyncRunFail().then(string => {
7     console.log(string);
8   }).catch(response => {
9     console.log(response);
10    // 小明 跌倒失敗(rejected)
11  })
```

Ref: https://wcc723.github.io/javascript/2017/12/30/javascript-async-await/

更多 Promise 與 Async / Await 的比較

- Promise

```
function getProcessedData(url) {
  return downloadData(url) // returns a promise
    .catch(e => {
      return downloadFallbackData(url); // returns a promise
    })
    .then(v => {
      return processDataInWorker(v); // returns a promise
    });
}
```

- Async / Await

```
async function getProcessedData(url) {
  let v;
  try {
    v = await downloadData(url);
  } catch(e) {
    v = await downloadFallbackData(url);
  }
  return processDataInWorker(v);
}
```

Ref: https://developer.mozilla.org/zh-TW/docs/Web/JavaScript/Reference/Statements/async_function

Await sequentially and parallelly

- Sequentially

```
var result1 = await sleep(2);
var result2 = await sleep(result1);
var result3 = await sleep(result2);

//result1 is 4
//result2 is 16
//result3 is 256
```

- Parallelly

```
var results = await Promise.all([sleep(1), sleep(2)]);
//results is [1,4]
```

Ref: https://developer.mozilla.org/zh-TW/docs/Web/JavaScript/Reference/Statements/async_function

Nested Await

```
for(var i =0 ; i<3; i++){
    var result = await sleep(i);

for(var j =0 ; j<result; j++){
        console.log('   i:'+i+', j:'+j+': ', await sleep(j));
    }
}

// i:1, j:0:  0
// i:2, j:0:  0
// i:2, j:1:  1
// i:2, j:2:  4
// i:2, j:3:  9
```

Ref: https://developer.mozilla.org/zh-TW/docs/Web/JavaScript/Reference/Statements/async_function

補充：Generator Function

- MDN 上有一句話：”async/await 函式的目的在於簡化同步操作 promise 的表現，以及對多個 Promise 物件執行某些操作。就像 Promise 類似於具結構性的回呼函式，同樣地，async/await 好比將 generator 與 promise 組合起來。”

- “function” 後面加個 * 就變 generator
  - ```
    function* idMaker() {
      var index = 0;
      while (index < index+1)  yield index++;
    }
    var gen = idMaker();
    console.log(gen.next().value); // 0
    console.log(gen.next().value); // 1
    console.log(gen.next().value); // 2
    console.log(gen.next().value); // 3
    ```

Ref: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*
    ```

After Class Readings

- (React) Routing
  - Quick start: https://reacttraining.com/react-router/web/guides/quick-start
  - 作者錄的介紹投影片：https://www.youtube.com/watch?v=cKnc8gXn80Q
- Redux
  - 官方網站的介紹：https://redux.js.org/introduction
  - 作者錄的教學系列：https://egghead.io/courses/getting-started-with-redux
  - ChentsuLin 大大的中文翻譯：https://chentsulin.github.io/redux/