AJAX與Fetch API

AJAX與XMLHttpRequest

AJAX這個技術名詞的出現是在十年前(2005)，其中內容包含XML、JavaScript中的XMLHttpRequest物件、HTML與CSS等等技術的整合應用方式，這個名詞並非專指某項特定技術或是軟體，Google在時所推出的Gmail服務與地圖服務，獲得很大的成功，當時這個技術名詞以此作為主要的案例說明。實際上這個技術的實現是在更早之前(2000年之前)，一開始是微軟公司實作了一個Outlook與郵件伺服器溝通的介面，後來把它整合到IE5瀏覽器上。在2006年XMLHttpRequest正式被列入W3C標準中，現在已被所有的瀏覽器品牌與新版本所支援。

所謂的AJAX技術在JavaScript中，即是以XMLHttpRequest物件(簡稱為XHR)為主要核心的實作。正如它的名稱，它是用於客戶端對伺服器端送出httpRequest(要求)的物件，使用的資料格式是XML格式(但後來JSON格式才是最為流行的資料格式)。流程即是建立一個XMLHttpRequest(XHR)物件，打開網址然後送出要求，成功時最後由回調函式處理伺服器傳回的Response(回應)。整體的流程是很簡單的，但經過這麼長久的使用時間(11年)，它在使用上產生不少令人頭痛的問題，例如:

- API設計得過於高階(簡單)，所有的輸出與輸入、狀態，都只能與這個XHR物件溝通取得，進程狀態是用事件來追蹤。
- XHR是使用以事件為基礎(event-based)的模組來進行異步程式設計。
- 跨網站的HTTP要求(cross-site HTTP request)與CORS(Cross-Origin Resource Sharing)不易實作。
- 對非文字類型的資料處理上不易實作。
- 除錯不易。

XHR在使用上都是像下面的範例程式碼這樣，其實你可以把它視作一種事件處理的結構，大小事都是依靠XHR物件來作，語法中並沒有把每件事情分隔得很清楚，而比較像是擠在一團:

```
function reqListener() {
  const data = JSON.parse(this.responseText);
  console.log(data)
}

function reqError(err) {
  console.log('Fetch Error :-S', err)
}

const oReq = new XMLHttpRequest();
oReq.onload = reqListener
oReq.onerror = reqError
oReq.open('get', './sample.json', true)
oReq.send()
```

在今天瀏覽器功能相當強大，以及網站應用功能複雜的時代，XHR早就已經不敷使用，它在架構上明顯的有太多的問題，尤其在很多功能的應用情況，程式碼會顯得複雜且不易維護。除非你是有一定要使用原生JavaScript的強迫症，要不然現在要作AJAX功能時，程式設計師並不會使用原生XHR物件來撰寫，大部份時候會使用外部函式庫。因為一個AJAX的程式，並不是單純到只有對XHR的要求與回應這麼簡單，例如你可能會對伺服器要求一份資料，當成功得到資料後，後面還有需要進一步的資料處理流程，這樣就會涉及到異步程式的執行結構，原生XHR並沒有提供可用的方式，它只是單純的作與伺服器互動那件事而已。

XHR Level 2(第2級)

XHR並不是沒有在努力進步，在約5年前已經有制定XHR的第2級新標準，但它仍然與原有XHR向下相容，所以整體的模型架構並沒有重大的改變，只是針對問題加以補強或是擴充。目前XHR第2級在9成以上的瀏覽器品牌新版本都已經支援全部的功能，除了IE系列要版本10之後才支援，以及Opera Mini瀏覽器完全不支援，還有一小部份功能在不同瀏覽器上實作細節會有所不同。XHR第2級(5年前)相較於原有的XHR(11年前)多加了以下的功能，這也是現在我們已經可以使用到的XHR的新特性:

- 指定回應格式
- 上傳文件與blob格式檔案
- 使用FormData傳送表單
- 跨來源資源共享(CORS)
- 監視傳輸的進程

不過，XHR第2級的新標準並沒有太引人注目的新功能，它比較像是解決長期以來的一些嚴重問題的補強版本。而且，XHR在原本上的設計就是這樣，常被批評的是它的語法結構不論在使用與設定都相當的零亂。補強或擴充都還是跳脫不了基本的結構，現今是HTM5、CSS3與

ES6的時代，有許多新的技術正在蓬勃發展，說句實在話，就是XHR技術已經舊掉了，當時的設計不符合現在時代需求了，這也無關對或錯。

jQuery

外部函式庫例如jQuery很早就看到XHR物件中在使用的問題，使用像jQuery的函式庫來撰寫AJAX相關功能，不光是在解決不同瀏覽器中的不相容問題，或是提供簡化語法這麼簡單而已。jQuery它擴充了原有的XHR物件為jqXHR物件，並加入類似於Promise的介面與Deferred Object(延遲物件)的設計。

為何要加入類似Promise的介面？可以看看它的說明中，是為了什麼而加入的？

> 這些方法可以使用一個以上的函式傳入參數，當 `$.ajax()` 的要求結束時呼叫它們。這可以讓你在單一個(request)要求中指定多個callbacks(回調)，甚至可以在要求完成後指定多個callbacks(回調)。 ~譯自jQuery官網The jqXHR Object

原生的XHR根本就沒有這種結構，Promise的結構基本上除了是一種異步程式設計的架構，它也可以包含錯誤處理的流程。簡單地來說，jQuery的目標並不是只是簡化語法或瀏覽器相容性而已，它的目標是要"**取代以原生XHR物件的AJAX語法結構**"，雖然本質上它仍然是以XHR物件為基礎。

jQuery作得相當成功，十分受到程式設計師們的歡迎，它的語法結構相當清楚，可閱讀性與設定彈性相當高，常讓人忘了原來的XHR是有多不好使用。在Promise還沒那麼流行的前些年，裡面就已經有類似概念的設計。加上現在的新版本(3.0)已經支援正式的Promise標準，說實在沒什麼理由不去使用它。以下是jQuery中ajax方法的範例:

```
// 使用 $.ajax() 方法
$.ajax({

    // 進行要求的網址(URL)
    url: './sample.json',

    // 要送出的資料 (會被自動轉成查詢字串)
    data: {
        id: 'a001'
    },

    // 要使用的要求method(方法)，POST 或 GET
    type: 'GET',

    // 資料的類型
    dataType : 'json',
})
  // 要求成功時要執行的程式碼
  // 回應會被傳遞到回調函式的參數
  .done(function( json ) {
     $( '<h1>' ).text( json.title ).appendTo( 'body' );
     $( '<div class=\'content\'>').html( json.html ).appendTo( 'body' );
  })
  // 要求失敗時要執行的程式碼
  // 狀態碼會被傳遞到回調函式的參數
  .fail(function( xhr, status, errorThrown ) {
    console.log( '出現錯誤，無法完成!' )
    console.log( 'Error: ' + errorThrown )
    console.log( 'Status: ' + status )
    console.dir( xhr )
  })
  // 不論成功或失敗都會執行的回調函式
  .always(function( xhr, status ) {
    console.log( '要求已完成!' )
  })
```

把原生的XHR用Promise包裹住，的確是一個好作法，有很多其他的函式庫也是使用類似的作法，例如axios與SuperAgent，相較於jQuery的多功能，這些是專門只使用於AJAX的函式庫，另外這些函式庫也可以用在伺服器端，它們也是有一定的使用族群。

Fetch

Fetch是近年來號稱要取代XHR的新技術標準，它是一個HTML5的API，並非來自ECMAScript標準。在瀏覽器支援性的部份，首先由Mozilla與Google公司在2015年3月發佈Fetch實作消息，目前也只有Firefox與Chrome、Opera瀏覽器在新版本中原生支援，微軟的新瀏覽器Edge也

在最近宣佈支援(新聞連結)(應該是Edge 14)，其他瀏覽器目前可以使用polyfill來作填充，提供暫時解決相容性的方案。另外，Fetch同樣要使用ES6 Promise的新特性，這代表如果瀏覽器沒有Promise特性，一樣也需要使用es6-promise來作填充。

Fetch並不是一個單純的XHR擴充加強版或改進版本，它是一個用不同角度思考的設計，雖然是可以作類似的事情。此外，Fetch還是基於Promise語法結構的，而且它的設計足夠低階，這表示它可以依照實際需求進行更多彈性設定。相對於XHR的功能來說，Fetch已經有足夠的相對功能來取代它，但Fetch並不僅於此，它還提供更多有效率與更多擴充性的作法。

> 註: 英文中 fetch/費曲/ 有"獲取"、"取回"的意思。它與get、bring單詞是近義詞。

Fetch基本語法

`fetch()` 方法是一個位於全域window物件的方法，它會被用來執行送出Request(要求)的工作，如果成功得到回應的話，它會回傳一個帶有Response(回應)物件的已實現Promise物件。 `fetch()` 的語法結構完全是Promise的語法，十分清楚容易閱讀，也很類似於jQuery的語法:

```
fetch('http://abc.com/', {method: 'get'})
.then(function(response) {
    //處理 response
}).catch(function(err) {
        // Error :(
})
```

但要注意的是fetch在只要在伺服器有回應的情況下，都會回傳已實現的Promise物件狀態(只要不是網路連線問題，或是伺服器失連等等)，在這其中也會包含狀態碼為錯誤碼(404, 500...)的情況，所以在使用時你還需要加一下檢查:

```
fetch(request).then(response => {
  //ok 代表狀態碼在範圍 200-299
  if (!response.ok) throw new Error(response.statusText)
  return response.json()
}).catch(function(err) {
        // Error :(
})
```

或是先用另一個處理狀態碼的函式，使用 `Promise.resolve` 與 `Promise.reject` 將回應的情況包裝為回傳不同狀態的Promise物件，然後再下個 `then` 方法再處理:

```
function processStatus(response) {
    // 狀態 "0" 是處理本地檔案 (例如Cordova/Phonegap等等)
    if (response.status === 200 || response.status === 0) {
        return Promise.resolve(response)
    } else {
        return Promise.reject(new Error(response.statusText))
    }
}

fetch(request)
    .then(processStatus)
    .then()
    .catch()
```

Fetch相關介面說明

fetch的核心由GlobalFetch、Request、Response與Headers四個介面(物件)與一個Body(Mixin混合)。概略的內容說明如下:

- GlobalFetch: 提供全域的 `fetch` 方法
- Request: 要求，其中包含 `method` 、 `url` 、 `headers` 、 `context` 、 `body` 等等屬性與 `clone` 方法
- Response: 回應，其中包含 `headers` 、 `ok` 、 `status` 、 `statusText` 、 `type` 、 `body` 等等屬性與 `clone` 方法
- Headers: 執行Request與Response中所包含的headers的各種動作，例如取回、增加、移除、檢查等等。設計這個介面的原因有一部份是為了安全性。
- Body: 同時在Request與Response中均有實作，裡面有包含主體內容的資料，是一種ReadableStream(可讀取串流)的物件

> 註: Mixin(混合)樣式是一種將多個物件(或類別)中會共同使用(分享)的方法或屬性另外用一個物件或介面整合包裝起來，然後讓其他的物件(或類別)來使用其中的方法或屬性的設計樣式。Mixins(混合)的主要目的是要讓程式碼功能可以達到重覆使用，但並不是透過類別繼承

> 的方式。

與XHR有很大的明顯不同，每個XHR物件都是一個獨立的物件，麻煩的是每次作不同的Request(要求)或要處理不同的Response(回應)時，就得再重新實體化一個新的XHR物件，然後再設定一次。而fetch中則是可以明確地設定不同的Request(要求)或Response(回應)物件，提供了更多細部設定的彈性，而且這些設定過的物件都可以重覆再使用。Request(要求)物件可以直接作為fetch方法的傳入參數，例如下面的這個範例:

```
const req = new Request(URL, {method: 'GET', cache: 'reload'})

fetch(req).then(function(response) {
  //處理 response
}).catch(function(err) {
        // Error :(
})
```

另一個很棒的功能是你可以用原有的Request(要求)物件，當作其他要新增的Request(要求)物件的基本樣版，像下面範例中的新的 `postReq` 即是把原有的 `req` 物件的method改為'POST'而已，這可以很容易重覆使用原先設定好的Request(要求)物件。

```
const postReq = new Request(req, {method: 'POST'})
```

以下摘要Request(要求)物件中可以包含的屬性值，可以看到設定值相當多，可以依使用情況設定到很細:

- method: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`。
- url: 要求的網址。
- headers: 與要求相關的Headers物件。
- referrer - `no-referrer`, `client` 或一個網址。預設為 `client`。
- mode - `cors`, `no-cors`, `same-origin`, `navigate`。預設為 `cors`。Chrome(v47~)目前的預設值是 `same-origin`。
- credentials - `omit`, `same-origin`, `include`。預設為 `omit`。Chrome(v47~)目前的預設值是 `include`。
- redirect - `follow`, `error`, `manual`。Chrome(v47~)目前的預設值是 。`manual`。
- integrity - Subresource Integrity(子資源完整性, SRI)的值
- cache - `default`, `no-store`, `reload`, `no-cache`, 或 `force-cache`
- body: 要加到要求中的內容。注意，method為 `GET` 或 `HEAD` 時不使用這個值。

> 註: 由於不能在GET時使用body屬性，如果你需要在GET時用到query字串，解決方案請參考以下的相關問答集: 問答或問答

Request(要求)物件中可以包含 `headers` 屬性，它是一個以 `Headers()` 建構式進行實體化的物件，實體化後可以再使用其中的方法進行設定。例如以下的範例:

```
const httpHeaders = { 'Content-Type' : 'image/jpeg', 'Accept-Charset' : 'utf-8', 'X-My-Custom-Header' : 'fetch are cool' }
const myHeaders = new Headers(httpHeaders)

const req = new Request(URL, {headers: myHeaders})


const httpHeaders = new Headers()
httpHeaders.append('Accept', 'application/json')

const req = new Request(URL, {headers: httpHeaders})
```

> 註: Headers()建構式的傳入參數可以是其他的Headers物件，或是內含符合HTTP headers的位元組字串的物件。

> 註: Headers物件中還有一個很特殊的屬性guard，它與安全性有關，請參考Basic_concepts#Guard

當然fetch方法也可以不需要一定得要傳入Request(要求)實體物件，它可以直接使用相同結構的物件字面當作傳入參數，例如以下的範例:

```
fetch('./sample.json', {
        method: 'GET',
        mode: 'cors',
        redirect: 'follow',
        headers: new Headers({
                'Content-Type': 'text/json'
        })
}).then(function(response) {
```

```
  //處理 response
})
```

fetch的語法連鎖下一個 `.then` 方法，如果成功的話，會得到一個帶有Response(回應)物件值的已實現狀態的Promise物件。雖然在fetch API中也允許你自己建立一個Response(回應)物件實體，不過，Response(回應)物件通常都是從外部資源要求所得到，自訂Response(回應)物件算是會在特殊的情況下才會作的事情。

Response(回應)物件中包含的屬性摘要如下:

- type: `basic` , `cors`
- url: 回應網址
- useFinalURL: 布林值，代表這個網址是否為最後的網址(也可能是重新導向的網址)
- status: 狀態碼 (例如: 200, 404, 500...)
- ok: 代表成功的狀態碼 (狀態碼介於200-299)
- statusText: 狀態碼的文字 (例如: OK)
- headers: 與回應相關的Headers物件

由於Response(回應)實作了Body介面(物件)，可以由Body的方法來取得回應回來的內容，但因為Body屬性值本身是個ReadableStream的物件，需要再依照不同的內容資料類型使用對應的方法，才能真正取到資料物件，其中最常使用的是json與text方法:

- arrayBuffer()
- blob()
- formData()
- json()
- text()

這幾個方法在使用過後，會產生帶有相關已解析資料值的已實現Promise物件，通常的作法還需要再下一個 `then` 方法中才取得到其中的已解析資料值(物件)。另一種作法是使用巢狀的Promise語法來取得資料，不過巢狀的Promise語法容易造成語法複雜，你可以獨立出來解析JSON資料物件的程式碼到另一個函式中會比較清楚。

> 註: arrayBuffer請參考ArrayBuffer
>
> 註: blob請參考Blob

不過要特別注意的是，Body實體的在Request(要求)與Response(回應)中的設計是"**只要讀取過就不能再使用**"，Request(要求)或Response(回應)物件其中都有一個 `bodyUsed` 只能讀不能寫的屬性，它在被讀取過會變成 `true` ，代表不能再被重覆使用。所以如果要重覆使用Body物件，必須在被讀取前(即 `bodyUsed` 被設定為 `true` 之前)，先呼叫Request(要求)或Response(回應)物件中的 `clone` 方法，另外拷貝出一個新的實體。

以下分別由幾種不同的資料類型來撰寫的樣式。

純文字/HTML格式文字

```
fetch('/next/page')
  .then(function(response) {
    return response.text()
  }).then(function(text) {
        console.log(text)
  }).catch(function(err) {
        // Error :(
  })
```

json格式

json方法會回傳一個帶有包含JSON資料的物件值的Promise已實現物件。

```
fetch('https://davidwalsh.name/demo/arsenal.json').then(function(response) {
        // 直接轉成JSON格式
        return response.json()
}).then(function(j) {
        // `j`會是一個JavaScript物件
        console.log(j)
}).catch(function(err) {
```

```
  // Error :(
})
```

blob(原始資料raw data)

```
fetch('https://davidwalsh.name/flowers.jpg')
        .then(function(response) {
          return response.blob();
        })
        .then(function(imageBlob) {
          document.querySelector('img').src = URL.createObjectURL(imageBlob);
        })
```

> 註: URL也是一個的Web API，在新式的瀏覽器上都有支援。請參考URL.createObjectURL

FormData

FormData是在要求時傳送表單資料時使用。以下為範例:

```
fetch('https://davidwalsh.name/submit', {
        method: 'post',
        body: new FormData(document.getElementById('comment-form'))
})
```

也可以使用JSON格式的物件資料來作要求:

```
fetch('https://davidwalsh.name/submit-json', {
        method: 'post',
        body: JSON.stringify({
                email: document.getElementById('email').value,
                answer: document.getElementById('answer').value
        })
})
```

> 註: FormData介面包含在XMLHttpRequest的新標準之中，目前只有Chrome與Firefox支援，請參考FormData

相較於jQuery.ajax

jQuery的ajax及相關方法的設計，已經很與fetch的語法結構很類似，不過它的回傳值仍然只是XHR物件的擴充jqXHR物件，需要經過轉換才能成為ES6的Promise物件。除此之外，有兩個重要的不同之處需要注意，來自window.fetch polyfill:

- `fetch` 方法回傳的Promise物件不會在有收到Response(回應)，但是是在HTTP錯誤狀態碼(例如404、500)的時候變成已拒絕(rejected)狀態。也就是說，它只會在網路出現問題或是被阻止進行Request(要求)時，才會變成已拒絕(rejected)狀態，其他都是已實現(fulfilled)。
- `fetch` 方法預設是不會傳送任何的認証証書(credentials)例如cookie到伺服器上的，這有可能會造成有管理使用者連線階段(session)的伺服器視為未經認証的Request(要求)。要加上傳送cookie可以用 `fetch(url, {credentials: 'include'})` 的語法來設置。

問題點

要求中斷或是設定timeout

Fetch目前沒有辦法像XHR可以中斷要求、或是設定timeout屬性，請參考Add timeout option #20的討論。這篇部落格JavaScript Fetch API in action有提供一個用Promise物件包裝的暫時解決方式，部份程式碼如下:

```
var MAX_WAITING_TIME = 5000;// in ms

var timeoutId = setTimeout(function () {
    wrappedFetch.reject(new Error('Load timeout for resource: ' + params.url));// reject on timeout
}, MAX_WAITING_TIME);

return wrappedFetch.promise// getting clear promise from wrapped
```

```
    .then(function (response) {
        clearTimeout(timeoutId);
        return response;
    });
```

進程事件(Progress events)

Fetch目前沒辦法觀察進程事件(或傳輸狀態)。在Fetch標準中有提供一個簡單的範例，但並不是太好的作法，程式碼如下:

```
function consume(reader) {
  var total = 0
  return pump()
  function pump() {
    return reader.read().then(({done, value}) => {
      if (done) {
        return
      }
      total += value.byteLength
      log(`received ${value.byteLength} bytes (${total} bytes in total)`)
      return pump()
    })
  }
}

fetch("/music/pk/altes-kamuffel.flac")
  .then(res => consume(res.body.getReader()))
  .then(() => log("consumed the entire body without keeping the whole thing in memory!"))
  .catch(e => log("something went wrong: " + e))
```

結論

Fetch在瀏覽器的實作與XHR不同，裡面的功能內容與API也相差很多，它有很多設計是為了新式的HTML5相關應用所設計的。現在已經有許多大公司的網站開始大量的使用Fetch API來取代XHR的作法，相信這個技術在這二、三年會愈來愈普及，畢竟AJAX技術對網站應用實在太重要，而這是一種扮演關鍵角色的技術。而且值得一提的是，現在在Chrome瀏覽器中在Service Worker技術實作中也提供了Fetch方法，但只限制在Service Worker中使用，這是一個非常新的應用技術，稱為Progressive Web App(漸進式的網路應用, PWA)。

補充: AJAX與Fetch函式庫比較表

本表主要參考自AJAX/HTTP Library Comparison。

名稱	Github星	瀏覽器支援	Node支援	Promise	原生	最後發佈日
XMLHttpRequest	-	是	-	-	是	-
Node HTTP	-	-	是	-	是	-
fetch	-	部份	-	是	是*	-
window.fetch polyfill	9269	全部	-	是	-	2016/5
node-fetch	894	-	是	是	-	2016/5
isomorphic-fetch	2587	是	是	是	-	2015/11
axios	2035	是	是	是	-	2016/7
SuperAgent	8175	是	是	是	-	2016/7
jQuery	40718	是	-	是*	-	2016/7

註記:

1. 以上統計數據為2016/7月
2. 瀏覽器是以目前各瀏覽器品牌的最新發佈穩定版本而言。
3. isomorphic-fetch是混合window.fetch polyfill(whatwg-fetch模組)與node-fetch的專案。
4. Promise為ES6新特性，瀏覽器支援一覽表。需要另外填充時使用es6-promise。
5. jQuery並非專門用於AJAX的函式庫，3.0版本後支援Promise目前標準。

參考資料

Fetch標準

- Fetch Standard @Github
- Fetch Standard

- Fetch API

教學

- Introduction to the Fetch API
- 深入浅出Fetch API
- Basic Fetch Request
- That's so fetch!
- fetch API
- JavaScript Fetch API in action
- Handling Failed HTTP Responses With fetch()

XHR&相關協定

- What is an Internet Protocol?
- XMLHttpRequest