Promise

> 一個promise代表一個異步操作的最終結果 ~譯自Promises/A+

Promise語法結構提供了更多的程式設計的可能性，它是一個經過長時間實戰的結構，在許多知名的函式庫或框架中長久之來可以見到Promise物件的身影，例如Dojo、jQuery、YUI、 Ember、Angular、WinJS、Q等等，之後Promises/A+社區則提供了可供遵守的統一標準，而在新的ES6標準中包含了Promise的實作，即將提供原生的語言內建支援，這將是個開始，往後會有愈來愈多API以此為基礎架構在其上。

Promise是一個強大的異步程式設計語法結構，ES6標準中，內容只有一個建構函式與一個`then`方法、一個`catch`方法，再加上四個必定使用`Promise`關鍵字呼叫的靜態函式，`Promise.resolve`、`Promise.reject`、`Promise.all`、`Promise.race`。要把語法介紹頁面看完只有7頁，內容少之又少。但Promise結構不容易理解，原因在於同步與非同步的轉換的概念，以及其中很多設計的原理與規則。

本文的範例是介紹原生的ES6 Promise為主，從基本概念出發，以Promises/A+作為主要的規則說明，另外集合了很多的實例與深入設計原理的解說，其目的是希望入門Promise結構的學習者，能有一個基礎的知識。章節最後提供了幾種應用實例與程式碼片段，希望能以最接近實際應用的範例，真正學會與活用Promise結構。

開始之前

異步Callback(回調)

> 並非所有的回調函式就是異步的回調函式，但異步執行的函式一定是回調函式

需要再次強調，並非所有的使用callbacks(回調)函式的API都是異步的，相反其實大部份的callbacks(回調)函式的結構都是同步的，如果你自己寫的callbacks(回調)結構那更不用說一定是同步的。在JavaScript語言中，除了事件處理9成以上都是異步外，只有少部份的API是異步的，要讓callbacks(回調)的結構轉變為異步，只有以下幾種方式:

- 使用計時器(timer)函式: `setTimeout`, `setInterval`
- 特殊的函式: `nextTick`, `setImmediate`
- 執行I/O: 監聽網路、資料庫查詢或讀寫外部資源
- 訂閱事件

> 執行I/O的API通常會出現在伺服器端(Node.js)，例如讀寫檔案、資料庫互動等等，這些API都會經過特別的設計。瀏覽器端只有少數幾個。

那麼像那些為尚未支援ES6 Promise瀏覽器打造的polyfill(填充)函式庫，又是怎麼作的？

一方面的確是用上面說的這些特別的方式，尤其是計時器的函式，或是一些各別新式瀏覽器中獨自的功能。有個舊的專案asap，裡面很多研究出來的作法，後來延伸出一套知名的Promise函式庫 - Q之中。es6-promise專案中也有個自己的asap.js，它是來自另一個異步程式的工具函式庫RSVP.js。

Deferred物件

如果你有用過近幾年在jQuery中的ajax相關方法(1.5版本之後)，其實你就有用過它裡面Deferred(延期)的設計了。Deferred(延期)算是很早就被實作的一種技術，最知名的是由jQuery函式庫在1.5版本(2011年左右)中實作的Deferred物件，用於註冊多個callbacks(回調)進入callbacks(回調)佇列，呼叫callbacks(回調)佇列，以及在成功或失敗狀態轉接任何同步或異步函式，這個目的也就是Deferred物件或Promise物件實作出來的原因。

Deferred的設計在jQuery中API豐富而且應用廣泛，尤其是在ajax相關方法中，更是受到很多程式設計師的歡迎。jQuery所設計的Deferred物件中其中有一個 `deferred.promise()` 可以回傳Promise物件，歷經多次的改版，現在在3.0版本中的jQuery.Deferred物件與現在的Promises/A+與ES6標準，已經是相容的設計，你可以把它視為是Promise的超集或擴充版本。

Deferred(延期)的設計在不同函式庫中略有不同，例如知名的Q函式庫(或Angular中的$q)，它把Promise物件視為Deferred物件的一個屬性，在使用上兩者扮演不同的角色。Deferred(延期)並不在我們要討論的細節。不過，相對來說ES6中的Promise物件功能較少。

如果你有需要使用外部函式庫或框架中關於Promise的設計，以及使用它們裡面豐富的方法與樣式，不妨先花點時間了解一下ES6中的Promise特性，畢竟這是內建的語言特性，對於一般的異步程式設計也許已經很足夠。

Promise與異步程式設計

> promise物件的設計就是針對異步函式的，要不就是用一個回傳值來變成已實現狀態，要不就是用一個理由(錯誤)來變成已拒絕狀態

同步程序你應該很熟悉了，大部份你寫的程式碼都是同步的程序。一步一步(一行一行)接著執行。異步程序有一些你可能用過， `setTimeout` 、 `XMLHttpRequest` (AJAX)之類的API，或是DOM事件的處理，在設計上就是異步的。

我們關心的是以函式(方法)的角度來看異步或同步，函式相當於包裹著要要一起來作某件事的程序語句，雖然函式內的這些程序有可能是同步的也有可能是異步的。JavaScript中的程式執行的設計是以函式為執行上下文(EC)的一個單位，也只有函式可以進入異步的執行流程之中。

> 同步函式的結果要不就是回傳一個值，要不然就是執行到一半發生例外，中斷目前的程式然後拋出例外。

異步的函式結果又會是什麼？要不然就最後回傳一個值，要不然就執行到一半發生例外，但是異步的函式發生錯誤時怎麼辦，可以馬上中斷程式然後拋出例外嗎？不行。那該怎麼作？只能用別的方式來處理。也就是說異步的函式，除了與同步函式執行方式不同，同步函式會直接在呼叫堆疊中執行，而異步函式會先送到工作佇列(queue)中，然後用事件迴圈再回到呼叫堆疊執行。而且它們對於錯誤的處理方式也要用不同的方式。

> 同步函式的結果要不就是帶有回傳值的成功，要不就是帶有回傳理由的失敗。

以一個簡單的比喻來說，你開了一間冰店，可能有些原料是自己作的，但也有很多配料或食材，是由別人生產的。同步函式就像你自己作的配料，例如自己製作大冰塊、煮紅豆湯之類的，每個步驟都是你自己監管品質，中間如果發生問題(例外)，例如作大冰塊的冰箱壞了，你也可以第一時間知道，而且需要你自己處理，但作大冰塊這件事就會停擺，影響到後面的工作。異步函式是另一種作法，有些配料是向別的工廠叫貨，例如煉乳或黑糖漿，你可以先打電話請工廠進行生產，等差不多時間到了，這些工廠就會把貨送過來。當工廠發生問題時，你可能只是接獲工廠通知，有可能是要延期交貨或是改向別的工廠叫貨。

promise物件的設計就是針對異步函式的，promise物件最後的結果要不然就用一個回傳值來fulfilled(實現)，要不然就用一個理由(錯誤)來rejected(拒絕)。

你可能會認為這種用失敗(或拒絕)或成功的兩分法結果，似乎有點太武斷了，但在許多異步的結構中，的確是用成功或失敗來作為代表，例如AJAX的語法結構。promise物件用實現(解決)與拒絕來作為兩分法的分別字詞。對於有回傳值的情況，沒有什麼太多的考慮空間，必定都是實現狀態，但對於何時才算是拒絕的狀態，這有可能需要仔細考量，例如以下的情況:

好的拒絕狀態應該是:

- I/O操作時發生錯誤，例如讀寫檔案或是網路上的資料時，中途發生例外情況
- 無法完成預期的工作，例如 `accessUsersContacts` 函式是要讀取手機上的聯絡人名單，因為權限不足而失敗
- 內部錯誤導致無法進行異步的程序，例如環境的問題或是程式開發者傳送錯誤的傳入值

壞的拒絕狀態例如:

- 沒有找到值或是輸出是空白的情況，例如對資料庫查詢，目前沒有找到結果，回傳值是0。它不應該是個拒絕狀態，而是帶有0值的實現。
- 詢問類的函式，例如 `hasPermissionToAccessUsersContacts` 函式詢問是否有讀取手機上聯絡人名單的權限，當回傳的結果是false，也就是沒有權限時，應該是一個帶有false值的實現。

不同的想法會導致不同的設計，舉一個明確的實例來說明拒絕狀態的情境設計。jQuery的 `ajax()` 方法，它會呼叫fail處理函式的條件，除了網路連線的問題外，在雖然有回應但是是屬於失敗類的HTTP狀態碼時，也會一併包含進來。但另一個用於類似功能的 `fetch` 方法並沒有， `fetch` 使用Promise架構，只有在網路連線發生問題才會轉為rejected(拒絕)狀態。

> 註: 在JavaScript中函式的設計，必定有回傳值，沒寫只是回傳undefined，相當於 `return undefined`

Promises/A+標準

> 原生的ES6 Promise是符合Promises/A+標準的

所謂的Promises/A+標準，其實就只一頁幾千字的網頁而已，裡面的用語並不會太難懂。ES6標準中也有自己的Promise物件章節，但因為涉及實作技術，內容明顯用字遣詞艱澀許多。以下使用Promises/A+標準作為一個開始，來解說Promise的內容有什麼。

專門用語

- `promise` (承諾)是一個帶有遵照這個規格的then方法的物件
- `thenable` 是一個有定義then方法的物件
- `value` 合法的JavaScript值(包含undefined、thenable與promise)
- `exception` (例外)使用throw語句丟出來的值
- `reason` (理由)表明為什麼promise被拒絕(rejected)的值

> 註: 另外有個常見的專有名詞 `settled` (固定的) 一個promise最後的狀態，也就是fulfilled(已實現)或rejected(已拒絕)

> 註: `reason` (理由)通常是一個Error物件，用於錯誤處理。

Promise狀態

promise物件必定是以下三種狀態中的其中一種: pending(等待中)、fulfilled(已實現)或rejected(已拒絕)。

2.1.1 當處在pending(等待中)時，一個promise:

```
2.1.1.1 可能會轉變到不是fulfilled(已實現)就是rejected(已拒絕)狀態
```

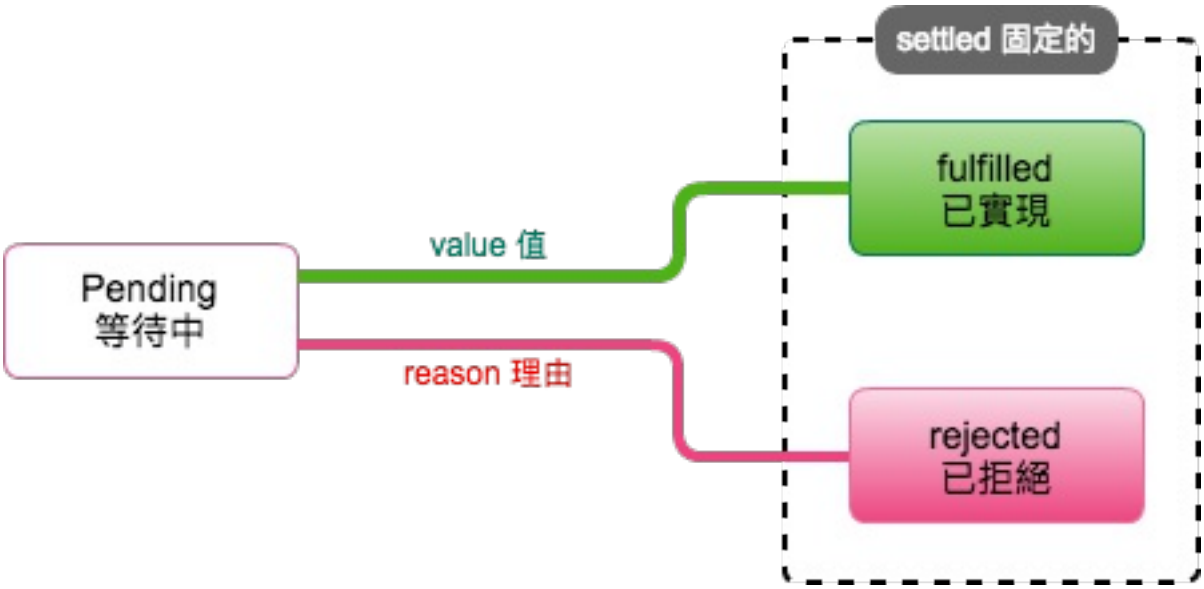
2.2.1 當處在fulfilled(已實現)時，一個promise:

```
2.2.1.1 必定不會再轉變到其他任何狀態
2.2.1.2 必定有不能再更動的值
```

2.3.1 當處在rejected(已拒絕)，一個promise:

```
2.3.1.1 必定不會再轉變到其他任何狀態
2.3.1.2 必定有不能再更動的值reason(理由)
```

這個用下面的圖解說明，應該可以很清楚的理解:



狀態是在Promise結構很重要的一個屬性，因為promise物件一開始都是代表懸而未決的值，所以一開始在promise物件在建立時，狀態都是pending(等待中)，之後可以轉變到fulfilled(已實現)就是rejected(已拒絕)其中一個，然後就固定不變了。通常有value(值)的情況是轉變到fulfilled(已實現)狀態，而如果是有reason(理由)時，代表要轉變到rejected(已拒絕)狀態。

不過，講是這樣講，實際上在實作時，promise物件一旦實體化完成回傳出來，就已經決定好是那一種狀態了，要不就是fulfilled(已實現)，要不就是rejected(已拒絕)。只是在實體化過程中，這個還不存在的(還沒回傳出來的)的promise物件的確是pending(等待中)狀態。下面實作時就看得到這個結果。

Promise物件建立與基本使用

> ES Promise的實作中，會確保Promise物件一實體化後就會固定住狀態，要不就是已實現，要不就是已拒絕

Promise物件的建立

一個簡單的Promise語法結構如下:

```
const promise = new Promise(function(resolve, reject) {
  // 成功時
  resolve(value)
  // 失敗時
  reject(reason)
});

promise.then(function(value) {
  // on fulfillment(已實現時)
}, function(reason) {
  // on rejection(已拒絕時)
})
```

首先先看Promise的建構函式，它的語法如下(來自MDN):

```
  new Promise( function(resolve, reject) { ... } )
```

用箭頭函式可以簡化一下:

```
  new Promise( (resolve, reject) => { ... } )
```

建構函式的傳入參數值需要一個函式。

這個函式中又有兩個傳入參數值，`resolve`(解決)與`reject`(拒絕)都是要求一定是函式類型。成功的話，也就是有合法值的情況下執行`resolve(value)`，promise物件的狀態會跑到fulfilled(已實現)固定住。失敗或是發生錯誤時用執行`reject(reason)`，reason(理由)通常是用Error物件，然後promise物件的狀態會跑到rejected(已拒絕)狀態固定住。

這個傳入函式稱為executor(執行者)函式，這是一種名稱為暴露的建構式樣式 Revealing Constructor Pattern的樣式。executor會在建構式回傳物件實體前立即執行，也就是說當傳入這個函式時，Promise物件會立即決定裡面的狀態，看是要執行`resolve`來回傳值，還是要用`reject`來作錯誤處理。也因為它與一般的物件實體化的過程不太一樣，所以常會先包到一個函式中，使用時再呼叫這個函式來產生promise物件，例如像下面這樣的程式碼:

```
function asyncFunction(value) {
    return new Promise(function(resolve, reject){
        if(value)
            resolve(value) // 已實現，成功
        else
            reject(reason) // 有錯誤，已拒絕，失敗
    });
}
```

Promise建構函式與Promise.prototype物件的設計，主要是要讓設計師作Promisify用的，也就是要把原本異步或同步的程式碼或函式，打包成為Promise物件來使用。包裝後才能使用Promise的語法結構，也就是我們所說的異步程式的語法結構。

你可能會認為在 `new Promise(function(resolve, reject){...})` 中的 `resolve` 與 `reject` 兩個傳入參數值的名稱，就一定要是這樣。而且學到最後面，最會讓人搞混的是Promise物件中也有兩個靜態方法(說明在下面章節)，名稱也恰好是 `Promise.resolve` 與 `Promise.reject` 。答案並不是一定要這樣命名，它只是對應之後執行時，習慣上這樣命名而已。你可以執行下面的範例，看能不能執行:

```
const promise = new Promise(function(resolveParam, rejectParam) {
    //resolveParam(1)
    rejectParam(new Error('error!'))
})

promise.then((value) => {
    console.log(value) // 1
    return value + 1
}).then((value) => {
    console.log(value) // 2
    return value + 2
}).catch((err) => console.log(err.message))
```

為什麼可以換傳入參數值的名稱？要回答這個問題，要先來解說一下進入Promise建構函式的大概執行流程，當然這是很簡化的說明而已:

1. 先用一個內部的物件，我把它稱為雛形物件，然後狀態(state)設定為pending(等待中)，值(value)為undefined，理由(reason)為undefined
2. 再來初始化這個物件的工作，用 `init(promise, resolver)` 函式，傳入建構式中的傳入參數，也就是 `function(resolve, reject){...}` 當作 `resolver` (解決者，解決用函式)傳入參數。
3. 把真正實作的Promise中的兩個內部resolve函式，與reject函式，對映到 `init(promise, resolver)` 中執行。

下面是把步驟說明實作出來的範例，不過這個只是為了方便解說用的，並不是真正可用的Promise建構函式，參考自 es6-promise:

```
//內部用的雛形物件，實作上包含在建構式中用this
const pInternal = {
  state: 'pending',
  value: undefined,
  reason: undefined
}

//這個就是稱為executor的傳入參數
function resolver(resolve, reject){
  resolve(10)
  //reject(new Error('error occured !'))
}

//初始化內部雛形物件用的函式
function init(promise, resolver){
    resolver(function resolvePromise(value){
      _resolve(promise, value);
    }, function rejectPromise(reason) {
      _reject(promise, reason);
    });
```

```
    return promise
}

//隱藏在內部的私有函式
function _resolve(promise, value){
  console.log(value)
  promise.state = 'onFulfilled'
  promise.value = value
}

//隱藏在內部的私有函式
function _reject(promise, reason){
  console.log(reason)
  promise.state = 'onRejected'
  promise.reason = reason
}

//最後生成回傳的promis物件
const promise = init(pInternal, resolver)
console.log(promise)
```

以上面的範例來說，`resolver`函式在`init`中被呼叫時，`resolver`的第1個傳入參數，它的執行程式碼內容會被`init`中的`resolver`呼叫時的第一個傳入參數`resolvePromise`所取代，然後再加上`init`函式傳入參數`promise`物件，最後呼叫執行`_resolve(promise, value)`這個內部私有方法。也就是說這只是一個函式傳入參數的代換過程。所以如果你改成這樣也是一樣的結果:

```
//這個就是稱為executor的傳入參數
function resolver(rs, rj){
  rs(10)
  //rj(new Error('error occured !'))
}

//初始化內部雛形物件用的函式
function init(promise, resolver){
  //改用匿名函式
    resolver(function(value){
      _resolve(promise, value)
    }, function(reason) {
      _reject(promise, reason)
    });

    return promise
}
```

或用箭頭函式來更簡化程式碼:

```
//這個就是稱為executor的傳入參數
function resolver(rs, rj) {
    rs(10)
        //reject(new Error('error occured !'))
}

//初始化內部雛形物件用的函式
function init(promise, resolver) {
    //改用匿名函式與箭頭函式
    resolver((value) => _resolve(promise, value), (reason) => _reject(promise, reason))
```

```
    return promise
}
```

那麼executor(執行者，執行函式)是必要的傳入參數值嗎？是的，如果你沒傳入任何的參數，它會出現一個類型錯誤，錯誤訊息中有一個resolver(解決者，解決函式)字詞，它應該算是executor的別名，或是第一個函式型傳入參數的名稱，如果你傳入一個空白函式，雖然不會有錯誤發生，但會產生一個完全無三小路用的Promise物件:

```
const promise = new Promise()
//Uncaught TypeError: Promise resolver undefined is not a function

const promise = new Promise(function(){})
//不會有錯誤，但會產生一個完全無用的promise，無法改變狀態
//Promise {[[PromiseStatus]]: "pending", [[PromiseValue]]: undefined}
```

Promise物件中的兩個靜態方法，`Promise.resolve`與`Promise.reject`，它們的實作是接近上面範例中的`resolve`與`reject`內部方法，不要搞混了。下面會談到它們的用法。

你可能會覺得很奇怪，為何要用這種方式來建構一個promise的物件實體？有幾個原因:

1. 暴露的建構式樣式: Promise只是個用來包裹現有函式或程式語句的物件，所以把建構式外露出來給程式設計師自行定義其中的程式碼。這才稱之為"暴露的建構式樣式(Revealing Constructor Pattern)"。因為它需要用callbacks回調的傳入參數才能進行把回調函式進行異步執行，所以就會設計成這樣。
2. 封裝: Promise物件不外露狀態，也無法從外部程式碼中直接修改其狀態，狀態由executor的執行結果決定。此外，Promise物件一旦被固定住兩種其一的狀態，就無法再改變狀態，這也是一個需要致力保護的原則。
3. Throw safety: 確保在建構函式在執行過程時，如果有throw例外的情況，也是安全的，能作異步的錯誤處理。有一種混用同步/異步程式所產生問題，稱之為 Zalgo。

> 註: 有一些術語或形容詞的意思大概都是相同的，fulfilled(已實現)、resolve(解決)、successful(成功的)、completed(完成的)差不多意思。而rejected(已拒絕)、fail(失敗)、error(錯誤)是另一個意思。

then與catch

在Promise的標準中，一直不斷的提到一個方法 - `then`，中文是"然後、接著、接下來"的意思，這個是一個Promise的重要方法。有定義then方法的物件被稱之為`thenable`物件，標準中花了一個章節在講`then`方法的規格，它的語法如下(出自MDN):

```
p.then(onFulfilled, onRejected);

p.then(function(value) {
   // fulfillment
  }, function(reason) {
  // rejection
});
```

`then`方法一樣用兩個函式當作傳入參數，`onFulfilled`是當promise物件的狀態轉變為fulfilled(已實現)呼叫的函式，有一個傳入參數值可用，就是value(值)。而`onRejected`是當promise物件的狀態轉變為rejected(已拒絕)呼叫的函式，也有一個傳入參數值可以用，就是reason(理由)。

為什麼說它是"一樣"？因為比對到promise物件的建構式，與那個傳入的函式參數值的樣子非常像，也是兩個函式當作傳

入參數，只是名稱的定義上有點不同，但接近同意義。

那麼 `then` 方法最後的回傳值是什麼？是另一個新的promise物件。

> Promises/A+標準 2.2.7
>
> `then` 必須回傳一個promise。 `promise2 = promise1.then(onFulfilled, onRejected);`

這樣設計的目的，主要是要能作連鎖(chained)的語法結構，也就是稱之為合成(composition)的一種運算方式，在JavaScript中如果回傳值相同的函式，可以使用連鎖的語法。像下面這樣的程式碼:

```
const promise = new Promise(function(resolve, reject) {
  resolve(1)
})

promise.then(function(value) {
  console.log(value) // 1
  return value + 1
}).then(function(value) {
  console.log(value) // 2
  return value + 2
}).then(function(value) {
  console.log(value) // 4
})
```

`then` 方法中的onFulfilled函式，也就是第一個函式傳入參數，它是有值時使用的函式，經過連鎖的結構，如果要把值往下傳遞，可以用回傳值的方式，上面的例子可以看到用 `return` 語句來回傳值，這個值可以繼續的往下面的 `then` 方法傳送。

onRejected函式，也就是 `then` 方法中第二個函式的傳入參數，也有用回傳值往下傳遞的特性，不過因為它是使用於錯誤的處理，除非你是有要用來修正錯誤之類的理由，不然說實在這樣作會有點混亂，因為不論是onFulfilled函式或onRejected函式的傳遞值，都只會記錄到新產生的那個Promise物件，對值來說並沒有區分是onFulfilled回傳的，還是onRejected回傳的。當一直有回傳值時就可以一直傳遞回傳值，當出現錯誤時，因為抓不到之前的值，會導致之前的值不見。

```
const p1 = new Promise((resolve, reject) => {
    resolve(4)
})

p1.then((val) => {
        console.log(val) //4
        return val + 2
    })
    .then((val) => {
        console.log(val) //6
        throw new Error('error!')
    })
    .catch((err) => {      //catch無法抓到上個promise的回傳值
        console.log(err.message)
        //這裡如果有回傳值，下一個then可以抓得到
        //return 100
    })
    .then((val) => console.log(val, 'done')) //val是undefined，回傳值消息
```

`then`方法中的兩個函式傳入參數，第1個onFulfilled函式是在promise物件有值的情況下才會執行，也就是進入到fulfilled(已實現)狀態。第2個onRejected函式則是在promise物件發生錯誤、失敗，才會執行。這兩個函式都可以寫出來，但為了方便進行多個不同程式碼的連鎖，通常在只使用`then`方法時，都只寫第1個函式傳入參數。

而錯誤處理通常交給另一個`catch`方法來作，`catch`只需要一個函式傳入參數，(出自MDN):

```
p.catch(onRejected);

p.catch(function(reason) {
   // rejection
});
```

`catch`方法相當於`then(undefined, onRejected)`，也就是`then`方法的第一個函式傳入參數沒有給定值的情況，它算是個`then`方法的語法糖。`catch`方法正如其名，它就是要取代同步`try...catch`語句用的異步例外處理方式。

> 註: 不過也因為`catch`方法與`try...catch`中的`catch`同名，造成IE8以下的瀏覽器產生衝突與錯誤。有些函式庫會用`caught`這個名稱來取代它，或是乾脆用`then`方法就好。

執行流程與錯誤處理

throw與reject

在promise建構函式中，直接使用`throw`語句相當於用`reject`方法的作用，一個簡單的範例如下:

```
// 用throw語句取代reject
const p1 = new Promise((resolve, reject) => {
    throw new Error('rejected!') // 用throw語句
    //相當於用以下的語句
    //reject(new Error('rejected!'))
})

p1.then((val) => {
        console.log(val)
        return val + 2
    }).then((val) => console.log(val))
    .catch((err) => console.log('error:', err.message))
    .then((val) => console.log('done'))

//最後結果:
//error: rejected!
//done
```

原先在錯誤處理章節中，對於`throw`語句的說明是，`throw`語句執行後會直接進到最近的`catch`方法的區塊中，執行區塊中程式碼後中斷之後程式碼。但這個理解是針對一般的同步程式結構，對異步的Promise結構並不適用。所以**並不會**中斷Promise結構的繼續執行，上面最後有個`then`中會輸出'done'字串，它依然會輸出。而且這裡的`throw`語句並不會瀏覽器的錯誤主控台出現錯誤訊息。

使用`throw`語句與用`reject`方法似乎是同樣的結果，都會導致promise物件的狀態變為rejected(已拒絕)，那麼這兩種方式有差異嗎？

有的，這兩種方式是有差異的。

首先，`throw`語句用於一般的程式碼中，它代表的意義是程式執行到某個時候發生錯誤，也就是`throw`語句會立即完成resolve(解決)，在then方法中按照規則，不論是onFulfilled函式或onRejected函式，只要丟出例外，就會導致新的promise物件的狀態直接變為Rejected(已拒絕)。

而`reject`則是一個一旦呼叫了就會讓Promise物件狀態變為Rejected(已拒絕)的方法，用起來像是一般的可呼叫的方法。

根據上面的解說，這兩種方式明顯是用在不同的場合的，實際上也無關優劣性。

其次，而在Promise中(建構函式或then)使用其他異步callback的API時會出現明顯的差異，這時候完全不能使用`throw`語句，以下的範例你可以試試:

```
const p1 = new Promise(function(resolve, reject){
     setTimeout(function(){
         // 這裡如果用throw，是完全不會拒絕promise
         reject(new Error('error occur!'))
         //throw new Error('error occur!')
     }, 1000);
});

p1.then((val) => {
        console.log(val)
        return val + 2
    }).then((val) => console.log(val))
    .catch((err) => console.log('error:', err.message))
    .then((val) => console.log('done'))
```

執行順序

如果你有看過其他的Promise教學，用`then`方法與`catch`來組成一個有錯誤處理的流程，例如像以下的範例程式，來自JavaScript Promises:

```
asyncThing1().then(function() {
  return asyncThing2();
}).then(function() {
  return asyncThing3();
}).catch(function(err) {
  return asyncRecovery1();
}).then(function() {
  return asyncThing4();
}, function(err) {
  return asyncRecovery2();
}).catch(function(err) {
  console.log("Don't worry about it");
}).then(function() {
  console.log("All done!");
});
```

在這篇文章中JavaScript Promises的也有一個這個範例的流程圖。

我要建議的是不要單純用只有onFulfilled函式的`then`方法，以及`catch`方法來看整體流程，其實會有點混亂。從完整的`then`方法可以把流程看得更清楚，每個`then`方法(或`catch`方法)都會回傳一個完整的Promise物件，當新的Promise往下個`then`方法傳遞時，因原來其中程式碼執行的不同，狀態也不同。

> Promises/A+標準 2.2.7
>
> `then`必須回傳一個promise。 `promise2 = promise1.then(onFulfilled, onRejected);`
>
> 2.2.7.1 當不論是onFulfilled或onRejected其中有一個是有回傳值`x`，執行Promise解析程序`[[Resolve]](promise2, x)`
>
> 2.2.7.2 當不論是onFulfilled或onRejected其一丟出例外`e`，promise2必須用`e`作為理由而拒絕(rejected)
>
> 2.2.7.3 當onFulfilled不是一個函式，而且promise1是fulfilled(已實現)時，promise2必須使用與promise1同樣的值被fulfilled(實現)
>
> 2.2.7.4 當onRejected不是一個函式，而且promise1是rejected(已拒絕)時，promise2必須使用與promise1同樣的理由被rejected(拒絕)

標準的2.2.7.4正好說明了，當你單純用只有onFulfilled函式的`then`方法時，如果發生在上面的Promise物件rejected(已拒絕)情況，為何會一直往下發生rejected(已拒絕)的連鎖反應，因為這個時候onRejected函式沒有傳入值，相當於undefined，也就是不算個函式，滿足了這個規則。會一直連鎖反應到某個`catch`方法(或是有onRejected函式的then方法)執行過後，才會又恢復使用2.2.7.1的正常規則。

標準的2.2.7.3恰好相反，它是在一堆只有`catch`方法的連鎖結構出現，`catch`方法代表沒有onFulfilled函式的`then`方法，只要最上面的Promise物件fulfilled(已實現)時，會一直連鎖反應到某個`then`方法，才會又恢復使用2.2.7.1的正常規則。當然，這種情況很少見，因為在整個結構中，我們使用一連串的`catch`方法是不太可能的，頂多只會使用一兩個而已。

> 註: catch方法通常會擺在整體流程的後面，這是因為擺前面會捕捉不到在它後面的then方法中的錯誤

我把上面的範例用箭頭函式簡化過，像下面這樣的範例:

```
async1()
    .then(() => async2())
    .then(() => async3())
    .catch((err) => errorHandler1())
    .then(() => async4(), (err) => errorHandler2())
    .catch((err) => console.log('Don\'t worry about it'))
    .then(() => console.log('All done!'))
```

然後再把`then`方法中的兩個函式傳入參數都補齊，`catch`方法也改用`then`方法來改寫。這樣作只是要方便解說這個規則影響流程是怎麼跑的，看得更清楚而已。實際使用你還是可以用上面的範例，當然前提是你已經很了解這些流程的規則。

```
async1()
    .then(() => async2(), undefined)
    .then(() => async3(), undefined)
    .then(undefined, (err) => errorHandler1())
    .then(() => async4(), (err) => errorHandler2())
    .then(undefined, (err) => console.log('Don\'t worry about it'))
    .then(() => console.log('All done!'), undefined)
```

情況: 當async1回傳的promise物件的狀態是rejected時。

1. 2.2.7.4規則，async2不會被執行，新的Promise物件直接是rejected狀態
2. 2.2.7.4規則，async3不會被執行，新的Promise物件直接是rejected狀態
3. 2.2.7.1規則，errorHandler1()被執行，新的Promise物件為onFulfilled狀態
4. 2.2.7.1規則，async4()被執行，新的Promise物件為onFulfilled狀態
5. 2.2.7.3規則，跳過輸出字串，新的Promise物件為onFulfilled狀態，回傳值繼續傳遞
6. 2.2.7.1規則，輸出字串

我想第3步是不易理解的，實際上它需要根據errorHandler1()函式的回傳值來決定新的Promise是哪一種狀態。上面只是一般的情況，當errorHandler1()只是簡單的回傳值或沒回傳值時。以下解說回傳值的差異情況。

根據連鎖反應，rejected狀態的Promise物件，會一直往下找到有 `onRejected` 函式的 `then` 方法，或是 `catch` 方法才會進入2.2.7.1規則，也就是正常的Promise處理程序，那麼在正常的處理程序中，對於處理時發生的情況，又是怎麼決定回傳的新Promise物件？下面是大概的摘要。

2.2.7.1規則裡面的執行原則，主要是由回傳值 `x` 來決定的，這裡所謂的 `x` 是經過onFulfilled或onRejected其中有一個被執行後的回傳值 `x` ，有幾種情況:

- `x` 不是函式或物件，直接用 `x` 實現(fulfill)promise物件(**注意: undefined也算，所以沒回傳值算這個規則**)
- `x` 是promise，看 `x` 最後是實現(fulfill)或拒絕，就直接對應到promise物件，回傳值或理由也會一併跟著
- `x` 是函式或物件，指定 `then = x.then` 然後執行 `then` 。其實它是在測試 `x` 是不是thenable物件，如果是thenable物件，會執行thenable物件中的then方法，一樣用類似then方法的兩選一來解析狀態，最後的狀態與值會影響到promise物件。一般情況就是如果不是thenable物件，也是直接用 `x` 作實現(fulfill)promise物件。

深入 then方法

> then方法實際上是整個Promise結構的流程運作主要核心

有些內容其實上面都有說過了，算是大概再整理一下。

一個promise物件經過then後，原本的promise物件的內容會改變？

不會。then方法會另外產生一個新的物件。以下為解說範例:

```
const promise = new Promise(function(resolve, reject) {
    resolve(1)
})

const p1 = promise.then((value) => {
    return value + 1
})

const p2 = promise.then((value) => {
    return value + 1
})

//延時執行
setTimeout(
() => {
  console.log(promise)
  console.log(p1)
  console.log(p2)
```

```
  console.log(p1 === p2) //false
}, 10000)
```

`then`方法中函式傳入參數的回傳值(onFulfilled)

`then`方法中onFulfilled函式回傳值，或是Promise建構函式中的resolve中的參數值，這是已經進入正常的"執行Promise解析程序"中。相較於onRejected函式，它通常只會用reason(理由)來處理，理由通常是個Error物件，用於錯誤處理中。當然，如果你的onRejected函式有回傳值，它的規劃也會和onFulfilled回傳值一模一樣。

上面已經有提到關於回傳值的情況，以及需要遵守的規則，你可以對照看看。這個章節的內容是要討論onFulfilled函式，你可以自訂哪一些回傳值。

`then`方法中onFulfilled函式回傳值可以有三種不同類型:

- 值
- promise物件
- thenable物件

值的話，就一般在JavaScript的各種值，這沒什麼好講的，因為`then`方法最後會回傳Promise物件，所以這個回傳值會跟著這個新Promise物件到下一個連鎖方法去，這個回傳值可以用下個`then`方法的onFulfilled函式取到第1個傳入參數值中。

promise物件的話，當你不希望`then`幫你回傳promise物件，自己建立一個，這也是合情合理。通常是使用promise建構式new一個，或是直接用`Promise.resolve(value)`靜態方法，產生一個fulfilled(已實現)狀態的promise物件。

thenable物件是最特殊的，根據標準的定義為:

> `thenable`是一個有定義then方法的物件或函式

按照標準上的解說，thenable是提供給沒有符合標準的其他實作函式庫或框架，利用合理的`then`方法來進行同化的的方式。以最簡單的情況說明，thenable物件是個單純物件，然後裡面有個then方法的定義而已，例如以下的範例:

```
const thenable1 = {
    then: function(onFulfill, onReject) {
        onFulfill('fulfilled!')
    }
}

const thenable2 = {
    then: function(resolve) {
        throw new TypeError('Throwing')
        resolve('Resolving')
    }
}
```

then中函式傳入參數的參數值(onFulfilled)

在開始討論前，稍作整理一下`then`方法中的傳入參數如下。

```
  promise.then(onFulfilled, onRejected)
```

then是promise物件中的方法，以onFulfilled與onRejected是作為兩個傳入參數，有幾個規則需要遵守:

- 當onFulfilled或onRejected不是函式時，忽略跳過
- 當promise是fulfilled時，執行onFulfilled函式，並帶有promise的value作為onFulfilled函式的傳入參數值
- 當promise是rejected時，執行onRejected函式，並帶有promise的reason作為onRejected函式的傳入參數值

then方法最後還要回傳另一個promise，也就是:

```
  promise2 = promise1.then(onFulfilled, onRejected)
```

以下假設只使用then方法中的第1個onFulfilled傳入參數(onRejected也是一樣)，那麼它可以有四種傳入的情況。在這個範例中，我們會把焦點放在其中有關於值的回傳情況。仔細看下面的範例中的 `doSomething2` 函式:

```
function doSomething1(){
  console.log('doSomething1 start')
  return new Promise(function(resolve, reject) {
      console.log('doSomething1 end')
      resolve(1)
  })
}

function doSomething2(){
  console.log('doSomething2')
  return 2
}

function finalThing(value){
  console.log('finalThing')
  console.log(value)
  return 0
}

//第1種傳入參數
doSomething1().then(doSomething2).then(finalThing)

//第2種傳入參數
doSomething1().then(doSomething2()).then(finalThing)

//第3種傳入參數
doSomething1().then(function(){doSomething2()}).then(finalThing)

//第3種傳入參數
doSomething1().then(function(){return doSomething2()}).then(finalThing)
```

第1種: 正常的函式傳入參數。最後的 `finalThing` 可以得到 `doSomething2` 回傳值 `2` 。

第2種: 雖然說 `then` 方法的規則，如果onFulfilled不是函式時會忽略，但這裡是執行 `doSomething2()` 函式，onFulfilled相當於 `doSomething2()` 的回傳值，JavaScript中的函式回傳值可以是個函式，沒執行過怎麼會知道它是不是回傳一個函式？所以會執行 `doSomething2()` ，但最後得到onFulfilled不是一個函式，所以忽略它。依照連鎖規則2.2.7.3(上面有說)，當onFulfilled不是函式，繼續用fulfilled狀態與帶值回傳新的Promise物件到下個then方法，最後的 `finalThing` 得到的值是 `doSomething1` 中的 `1` 。

第3種: 正常的函式傳入參數，因為在函式中執行 `doSomething2()` ，這個onFulfilled最後的回傳值其實是undefined，但算有回傳值，回傳的新Promise物件也是fulfilled狀態，不過值變成 `undefined` 。最後的 `finalThing` 得到 `undefined` 值。

第4種: 正常的函式傳入參數， `then` 方法執行完onFulfilled最後的回傳值是 `doSomething2()` 的執行後的值也就是 `2` ，回傳的新Promise物件也是fulfilled狀態，最後的 `finalThing` 得到 `2` 值。

> 註: 在JavaScript中函式的設計，必定有回傳值，沒寫只是回傳undefined，相當於 `return undefined`

then中函式傳入參數的參數值(異步)(onFulfilled)

上面的概念如果在doSomething1、doSomething2與finalThing函式中，都有異步的callbacks(回調)時，這時除了值的情況，我們還會關心整體的執行順序。這個程式碼範例是來自We have a problem with promises，其實這已經是稍微進階的討論議題了。

```
function doSomething1(){
  console.log('doSomething1 start')
  return new Promise(function(resolve, reject) {
    setTimeout(function(){
      console.log('doSomething1 end')
      resolve(1)
    }, 1000)

  })
}

function doSomething2(){
  console.log('doSomething2 start')
  return new Promise(function(resolve, reject) {
    setTimeout(function(){
      console.log('doSomething2 end')
      resolve(2)
    }, 1000)

  })
}

function finalThing(value){
  console.log('finalThing start')
  return new Promise(function(resolve, reject) {
    setTimeout(function(){
      console.log('finalThing end')
      console.log(value)
      resolve(0)
    }, 1000)

  })
}

//第1種傳入參數，finalThing最後的值為2
doSomething1().then(doSomething2).then(finalThing)

//第2種傳入參數，finalThing最後的值為1
doSomething1().then(doSomething2()).then(finalThing)
```

```
//第3種傳入參數，finalThing最後的值為undefined
doSomething1().then(function(){doSomething2()}).then(finalThing)

//第4種傳入參數，finalThing最後的值為2
doSomething1().then(function(){return doSomething2()}).then(finalThing)
```

執行的結果會出乎想像，只有第1種與第4種，才是完整的Promise流程順序，也就是像下面的流程:

```
doSomething1 start
doSomething1 end
doSomething2 start
doSomething2 end
finalThing start
finalThing end
```

第2種中的 `then` 方法裡onFulfilled傳入參數的 `doSomething2()` 執行語句，它是一個同步的語句，所以會在 `doSomething1` 還沒執行完成時，就先被執行，所以流程會變為:

```
doSomething1 start
doSomething2 start
doSomething1 end
finalThing start
doSomething2 end
finalThing end
```

第3種在 `then` 方法裡裡onFulfilled的 `function(){doSomething2()}` 裡面的 `doSomething2()` 執行語句，也是一個同步的語句，但外團的匿名函式卻是一個異步函式，因為這樣會在 `doSomething1()` 結束才開始執行，但是也是在 `finalThing` 開始後才會結束。不過流程也是怪異:

```
doSomething1 start
doSomething1 end
doSomething2 start
finalThing start
doSomething2 end
finalThing end
```

從這個範例中，我認為並不需要太深究其中的順序的原因是為何。這範例其實是在告訴你不要亂用 `then` 方法中的傳入參數值，要不就是個堂堂正正的函式，要不然就寫好一個有回傳值的匿名函式。此外，如果你要在Promise結構中使用其他的異步API，更是要注意它們的執行順序，用想的還不如直接寫出來執行看看，有可能不見得是最後是你要的。

靜態方法 Promise.resolve與Promise.reject

> `Promise.reject` 或 `Promise.resolve` 只用於單純的傳入物件、值(理由)或外部的thenable物件，轉換為Promise物件的場合

`Promise.resolve` 是一個靜態方法，也就是說它一定加上 `Promise` 這樣呼叫的方法。 `Promise.resolve` 等於是要產生

fulfilled(已實現)狀態的Promise物件，它相當於直接回傳一個以`then`方法實現的新Promise物件，帶有傳入的參數值，這個方法的傳入值有三種，相當於`then`方法中的回傳值(上面有說)，語法如下:

```
Promise.resolve(value);
Promise.resolve(promise);
Promise.resolve(thenable);
```

直接用resolve代表這個新的Promise物件的狀態是直接設定為fulfilled(已實現)狀態，這方法只是方便產生根(第一個)Promise物件使用的，也就是除了用建構式的`new Promise()`外，提供另一種方式來產生新的Promise物件。下面為範例:

```
Promise.resolve("Success").then(function(value) {
  console.log(value) // "Success"
}, function(value) {
  // 不會被呼叫
})
```

`Promise.reject`剛好與`Promise.resolve`相反，等於是要產生rejected(已拒絕)狀態的新Promise物件。範例如下:

```
Promise.reject(new Error("fail")).then(function(error) {
  // 不會被呼叫
}, function(error) {
  console.log(error); // Stacktrace
});
```

`Promise.reject`與`Promise.resolve`的實作的程式碼隱藏了在內部關於Promise物件產生的過程，前面有提到我們不能用沒有executor傳入參數的Promise建構式來產生新的物件實體，會有錯誤發生。這兩個靜態方法，它們在實作時的確有產生沒有executor傳入參數的Promise雛形(中介)物件，只是它還不算真正回傳出來的Promise物件實體。理由主要當然是不讓程式設計師自由控制Promise的狀態，要改變Promise的狀態只有靠程式碼的執行結果才行。例如下面這一段程式碼(出自es6-promise(polyfill)):

```
var promise = new Constructor(noop); //noop是沒有內容的函式
_resolve(promise, object); //_resolve是內部函式，object是Promise.resolve傳入參數值
return promise;
```

`Promise.reject`與`Promise.resolve`與使用Promise建構式的方式，在使用上仍然有大的不同，executor傳入參數可以讓程式設師自訂Promise產生的過程，而且在產生的過程中是throw safety(安全)的。以下面的例子來說明:

```
//方式一: 使用Promise建構式

function initPromise1(someObject) {

    return new Promise(function(resolve) {
        if (typeof someObject !== 'object')
            throw new Error('error occured')
        else
            resolve(someObject);
    });
}

//方式二: 使用Promise.resolve+throw

function initPromise2(someObject) {
```

```
    if (typeof someObject !== 'object')
        throw new Error('error occured')
    else
        return Promise.resolve(someObject)
}

//方式三: 使用Promise.resolve+Promise.reject

function initPromise3(someObject) {
    if (typeof someObject !== 'object')
        return Promise.reject(new Error('error occured'))
    else
        return Promise.resolve(someObject)
}

//測試用
initPromise1(1).then((value) => {
    console.log(value)
}).catch((err) => {
    console.log(err.message)
})
```

如果此時傳入的`someObject`不是物件類型，方式二會直接throw出例外，造成程式中斷，與Promise結構中對錯誤的處理方式不同，只有方式一或方式二可以正常的處理錯誤。

上面的例子是太過簡單，如果initPromise函式中的程式碼很複雜的時候，你在回傳`Promise.reject`或`Promise.resolve`前發生例外，是完全沒辦法控制的。

結論是`Promise.reject`或`Promise.resolve`只用於單純的傳入物件、值或外部的thenable物件，轉換為Promise物件的場合。如果你要把一整段程式碼語句或函式轉為Promise物件，不要用這兩個靜態方法，要使用Promise建構式來產生物件才是正解。

> 註: 在Promises/A+並沒有關於`Promise.reject`或`Promise.resolve`的定義，這兩個算是語法簡便使用的語法糖。

靜態方法 Promise.all與Promise.race

> Promise.all與Promise.race的參數值，通常使用陣列結構，而陣列中放的要不是就一般值，要不就是Promise物件

`Promise.all`是並行運算使用的靜態方法，它的語法如下(出自MDN):

```
Promise.all(iterable);
```

`iterable`代表可傳入陣列(Array)之類的物件，JavaScript內建的有實作`iterable`協定的有String、Array、TypedArray、Map與Set這幾個物件。一般使用上都只用到陣列而已。`Promise.all`方法會將陣列中的值並行運算執行，全部完成後才會接著下個`then`方法。在執行時有幾種情況:

- 陣列中的索引值與執行順序無關

- 陣列中的值如果不是Promise物件，會自動使用`Promise.resolve`方法來轉換
- 執行過程中只要有"**其中一個(any)**"陣列中的Promise物件執行，發生錯誤例外，或是有Promise的reject情況，立即變為rejected狀態，往下回傳
- 實現完成後，接下來的then方法會獲取到的值是陣列值

```
const p1 = Promise.resolve(3)
const p2 = 1337
const p3 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('foo'), 1000)
});

Promise.all([p1, p2, p3]).then((value) => {
    console.log(value)
}).catch((err) => {
    console.log(err.message)
})

//結果: [3, 1337, "foo"]
```

你可能會好奇，String(字串)類型是可以傳到`Promise.all`中的參數，傳入後會變成什麼樣子，當然實際上應該沒人這樣在用的。以下為一個簡單的範例:

```
Promise.all('i am a string').then((value) => {
    console.log(value)
}).catch((err) => {
    console.log(err.message)
})

//結果: ["i", " ", "a", "m", " ", "a", " ", "s", "t", "r", "i", "n", "g"]
```

`Promise.race`它的真正名稱應該是對比`Promise.all`的"any"，`Promise.all`指的是"**所有的**"陣列傳入參數的Promise物件都要解決(resolve)完了才進行下一步，`Promise.race`則是"**任何一個**"陣列傳入參數的Promise物件有解決，就會到下一步去。用"race(競賽)"這個字詞是比喻就像在賽跑一樣，只要有一個參賽者到達終點就行了。

`Promise.race`的規則與`Promise.all`相同，只不過實現的話，下一步的`then`方法只會獲取得勝的那個值，一個簡單的範例如下:

```
const p1 = Promise.resolve(3)
const p2 = 1337
const p3 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('foo'), 1000)
});

Promise.race([p1, p2, p3]).then((value) => {
    console.log(value)
}).catch((err) => {
    console.log(err.message)
})
```

其實上面這個`Promise.race`範例，你把p1與p2的位置對調，就會發現最後的結果正好會對調，也就是說正好是只使用`Promise.resolve`方法的轉換情況，是和陣列中的前後順序有關的。不過因為`Promise.race`只能選出一個優勝者，p1與p2應該算同時，所以也只能以陣列的順序為順序。

> 註: `Promise.race`應該要多一個規則，如果陣列中有同時實現的promise值，以陣列中的順序優先者為回傳值

下面的例子是有加上每個陣列中的Promise物件產生時間的不同，這當然就只會回傳最快實現的那個Promise物件，也就是p3。

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('p1'), 2000, )
});
const p2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('p2'), 1000, 'p2')
});
const p3 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('p3'), 500, 'p3')
});

Promise.race([p1, p2, p3]).then((value) => {
    console.log(value)
}).catch((err) => {
    console.log(err.message)
})
```

> 註: 在Promises/A+並沒有關於`Promise.all`或`Promise.race`的定義。

執行順序 Chaining(連鎖)或Branching(分支)

Chaining(連鎖)是我們在上面一直看到的用`then`與`catch`方法串接起來的結構，這個結構會依照同步流程的原則，一步一步往下執行。如果你已經看過上面的內容，相信你已經很熟悉了。像下面的例子這樣:

```
const p1 = new Promise((resolve, reject) => {
  resolve(1)
});


p1.then((value) => {
    console.log(value)
    return value+1
}).then((value) => {
    console.log(value)
    return value+2
}).then((value) => {
    console.log(value)
    return value+3
})
```

如果你以為相當下面這樣的程式碼，表示你對Chaining(連鎖)語法的認知是有問題的，這種叫作Branching(分支)的結構。因為經過`then`方法之後，雖然會產生一個新的Promise物件，原有的Promise物件並不會消失或被更改內容。每一次使用`p1`仍然是一樣的`p1`，每一段Promise語法由同樣的`p1`值各自發展，對程式來說它們是同步的語句:

```
const p1 = new Promise((resolve, reject) => {
```

```
  resolve(1)
});


p1.then((value) => {
    console.log(value)
    return value+1
})

p1.then((value) => {
    console.log(value)
    return value+2
})

p1.then((value) => {
    console.log(value)
    return value+3
})
```

Chaining(連鎖)語法真正的相等語法如下:

```
const p1 = new Promise((resolve, reject) => {
  resolve(1)
});


const p2 = p1.then((value) => {
    console.log(value)
    return value+1
})

const p3 = p2.then((value) => {
    console.log(value)
    return value+2
})

p3.then((value) => {
    console.log(value)
    return value+3
})
```

在Promise中使用連鎖的架構，就是一般稱為sequential(序列)執行的結構，而Branching(分支)並不是parallel(並行)的執行結構，真正的parallel(並行)結構應該是使用 `Promise.all` 方法的語法。序列執行可以保証一個Promise接著另一個執行，也就是"同步中的異步"結構。Branching(分支)的執行順序就很亂了，要看程式碼執行的順序來決定。

> sequential(序列)執行結構，也有另一個名稱是waterfall(瀑布)

反樣式(anti-pattern)

反樣式(anti-pattern)是指錯誤或不建議的用法，為避免很多潛在的問題，或是導正一般使用者的常見錯誤，網路上有很多整理好的反樣式，以下列出常見的幾個。

巢狀的(Nested) Promises

巢狀的(Nested) Promises無疑是讓錯誤處理變得更加困難，而且到底程式會怎麼執行，從程式碼中很難預期。

```
firstThingAsync().then((result1) => {
  //巢狀Promises，不建議使用
  secondThingAsync().then((result2) => {
    // 可以存取得到result1與result2
  })
},(err) => {console.log(err.message)}    //這裡捕捉不到錯誤
).catch((err) => {console.log(err.message)}) //這裡捕捉不到錯誤
```

解決之道

如果你是要並行處理 `firstThingAsync` 函式與 `secondThingAsync` 函式，可以用 `Promise.all` 方法。

```
Promise.all([firstThingAsync(), secondThingAsync()]).then(function(value) {
    console.log(value)
}).catch(function(err) {
    console.log(err.message)
})
```

如果你希望 `secondThingAsync` 函式是可以獲得 `firstThingAsync` 函式先執行完的結果result1，可以先解決 `firstThingAsync` 函式，得到Promise物件與結果result1後，再用 `Promise.all` 方法來保証 `secondThingAsync` 函式與結果result1可以並行。這個結構有點複雜，而且這是因為 `then` 方法可以回傳一個Promise物件。

```
firstThingAsync().then((result1) => {
  return Promise.all([result1, secondThingAsync(result1)])
  })
  .then((result2) => { console.log(result2)})
  .catch((err) => {console.log(err.message)})
```

忘了加catch方法

如果你把所有的 `then` 方法都只用於fulfilled(已實現)情況，而用 `catch` 方法用於rejected情況，這是個好主意，它可以讓你的程式碼更清楚易讀。

但至少每個連鎖的結構中，都至少要有個 `catch` 方法，而且最好是最後一個，因為 `catch` 方法只能捕捉到在前面步驟的錯誤。

```
firstThingAsync().then(function() {
    return secondThingAsync()
}).then(function() {
    return thirdThingAsync()
}).catch((err) => {
    console.log(err.message)
})
```

then方法中沒用return值

在JavaScript中的函式，你最後沒寫 `return值` 的語句，它照樣會 `return undefined` 。 `then` 方法中的函式傳入參數回

傳值，攸關這個準備要回傳的新Promise物件的狀態值，關係重大。像下面這樣的範例是不建議使用的反樣式，而且這是很常發生的錯誤:

```
somePromise().then(() => {
  someOtherPromise()
}).then( () => {
  // 你覺得someOtherPromise會回傳給你嗎？ 我想不會
})
```

解決之道

then方法中的函式傳入參數，總是要有回傳值，要不然，就throw出錯誤也行，最後有catch方法可以接住錯誤。

```
somePromise().then(() => {
  return someOtherPromise()
}).then(() => {

}).catch((err) => {
    console.log(err.message)
})
```

理由(reason)不是一個Error物件

實例&程式碼片段

Delay(延時)

https://www.w3.org/2001/tag/doc/promises-guide#example-delay

包裝XMLHttpRequest

forEach/for/while

Promisify

```
function promiseCall(f, ...args) {
    try {
        return Promise.resolve(f(...args));
    } catch (e) {
        return Promise.reject(e);
    }
}
```

回調地獄(Callback hell)的改寫

你或許有聽過Promise是可以解決一種名稱為"回調地獄(Callback hell)"或是""的結構，

Promise libraries take the typical callback function: function(err, response) { } And split those arguments into separate

then/catch chainable callbacks: .then(function(response) { }).catch(function(err) { })

https://thomashunter.name/blog/the-long-road-to-asyncawait-in-javascript/
http://stackoverflow.com/questions/32038961/avoiding-promise-anti-patterns-with-node-asynchronous-functions
https://medium.com/@rdsubhas/es6-from-callbacks-to-promises-to-generators-87f1c0cd8f2e#.9fqj27hm1
http://jamesknelson.com/grokking-es6-promises-the-four-functions-you-need-to-avoid-callback-hell/

不適合使用Promise情況

取代Promise方案

外部函式庫議題

在ES6標準正式定案將Promise特性加入後，再加上各家瀏覽器紛紛發佈已附有實作Promise特性的新版本後，約莫2015年中開始，有許多Promise相關的外部函式庫，有逐漸的發佈趨緩，或是停止再維護的現象。以下列出幾套常見有遵照Promise標準，而且有進行擴充的函式庫(單純只加速或polyfill的不列入)，以及它們的發佈情況:

- q: 維護停止，最後發佈版本在2015/5
- bluebird: 維護持續(唯一)，最後發佈版本在2016/6
- when: 維護停止，最後發佈版本在2015/12
- then-promise: 維護停止，最後發佈版本在2015/12
- rsvp.js: 維護停止，最後發佈版本在2016/2
- vow: 維護停止，最後發佈版本在2015/12

> 註: 維護停止，代表6個月左右都沒有新的發佈版本

由列表中可以看到至2016/7目前唯一還在在更新版本的，就只有bluebird一套而已。下面分析為何會有這個現象發生，以及到底你要選擇用原生的ES6 Promise還是要使用外部函式庫的一些建議。

首先根據一篇在2015/4月網路上的問答為何原生的ES6 Promise會比bluebird慢而且更耗記憶體中回答，這是由bluebird函式庫的作者回答的內容來看，當時的原生ES6 Promise，又慢又耗記憶體的主因如下:

1. 尚未最佳化，實作當時也只是JavaScript程式碼
2. 原生ES6 Promise使用非常耗資源的 `new Promise` 與executor方式來建立根(第一個)Promise物件

當時2015年的效能評比大概的資料還在這裡。

- 序列執行(sequential): 原生ES6 Promise的執行時間需要bluebird的4倍，記憶體需要3.6倍
- 並行執行(parallel): 原生ES6 Promise的執行時間需要bluebird的9倍，記憶體需要5.7倍

看到這個結果，當然有很多人會存疑，為何bluebird會快成這樣，或是原生ES6 Promise會慢成這樣，就算是都用JavaScript程式碼實作，這麼大的差距也是很誇張。實際上在設計的本質上並不相同，bluebird以較有效率而且節省資源的方式來設計整體的執行流程，其他的函式庫都沒有辦法這麼快又節省資源。根據這篇bluebird的issue中的開發者指出，以Q與bluebird的設計比較，Q會慢bluebird幾十倍是因為Q針對瀏覽器提供某些安全上的設計，bluebird則是極致的追求執行效率，假設所有程式碼都是在完全可信任的環境例如Node.js中執行。或許也因為如此，在網路上很多類似的問答中，通常建議只在伺服器端(Node.js)使用bluebird，而瀏覽器端建議使用原生ES6 Promise或Q。

當然，經過一年之後的原生ES6 Promise，其效率與記憶體消耗問題已有了很大的改善，根據最近Chrome瀏覽器使用

的V8引擎在版本53發佈的消息，目前的原生ES6 Promise進行最佳化後，效能改善了20-40%。也就是說未來的原生ES6 Promise在經過最佳化後，與外部函式庫的執行效率並不會相差太遠，這個消息無疑為原生ES6 Promise打了一劑強心針。

所以如果你要要使用外部函式庫，其理由可能是以下幾個:

- 為了能使用豐富的API，但不見得是為了效率，效率有可能在將來並不是唯一的重點。
- 資源存取類型的函式庫或模組，現在都會提供具有Promise的API，例如資料庫、檔案處理、網路資源存取的模組。例如MongoDB Node.JS Driver就是雙模式的。有一些AJAX的函式庫也都是用Promise的架構，例如axios，或是像新式的標準Fetch API，也是基於Promise的。
- 工具類型的函式庫，現在也提供相容於Promise的物件，例如jQuery(3.0之後)。

雖然jQuery 3.0中實作了符合Promises/A+標準的Promise物件，但是jQuery並非單純用於Promise架構的外部函式庫，而且3.0也才剛發佈不久，再加上長期以來，jQuery有很多自己設計的擴充API，API的名稱與標準中有些不同。如果你原本就有使用jQuery，是可以使用它新版本中的deferred物件與Promise物件的功能，但要仔細的理解與原生ES6 Promise的差異性。

原生ES6 Promise會被認為是一個基礎，如果是在很簡單的執行流程，或許就已經夠用了。學習語法和使用方法後，以這個基礎你可以因應不同的需求，使用專屬的資源存取模組(例如資料庫)。而這些專門使用在Promise擴充的函式庫(例如bluebird)，雖然提供了豐富API，但使用它們的情況相信會愈來愈少，這也是為什麼後來這些函式庫都停止維護的主要原因。

參考資源

標準

- Promises/A+
- ECMA-262
- Writing Promise-Using Specifications(W3C)

其他樣式、反樣式

- Promise Anti-Patterns

** https://gist.github.com/domenic/3889970 ** http://www.datchley.name/es6-promises/ ** http://exploringjs.com/es6/ch_promises.html

http://www.datchley.name/promise-patterns-anti-patterns/

https://sdgluck.github.io/2015/08/24/promise-ponderings-patterns-apologies/#section4.1

http://stackoverflow.com/questions/37651780/why-does-the-promise-constructor-need-an-executor

http://stackoverflow.com/questions/28687566/difference-between-defer-promise-and-promise/28692824#28692824

http://www.html5rocks.com/en/tutorials/es6/promises/

http://www.2ality.com/2014/10/es6-promises-api.html

https://blog.domenic.me/the-revealing-constructor-pattern/

https://davidwalsh.name/promises

https://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html

http://de.slideshare.net/domenicdenicola/callbacks-promises-and-coroutines-oh-my-the-evolution-of-asynchronicity-in-javascript

http://liubin.org/promises-book/

http://www.mattgreer.org/articles/promises-in-wicked-detail/

https://zeit.co/blog/async-and-await

http://solutionoptimist.com/2013/12/27/javascript-promise-chains-2/