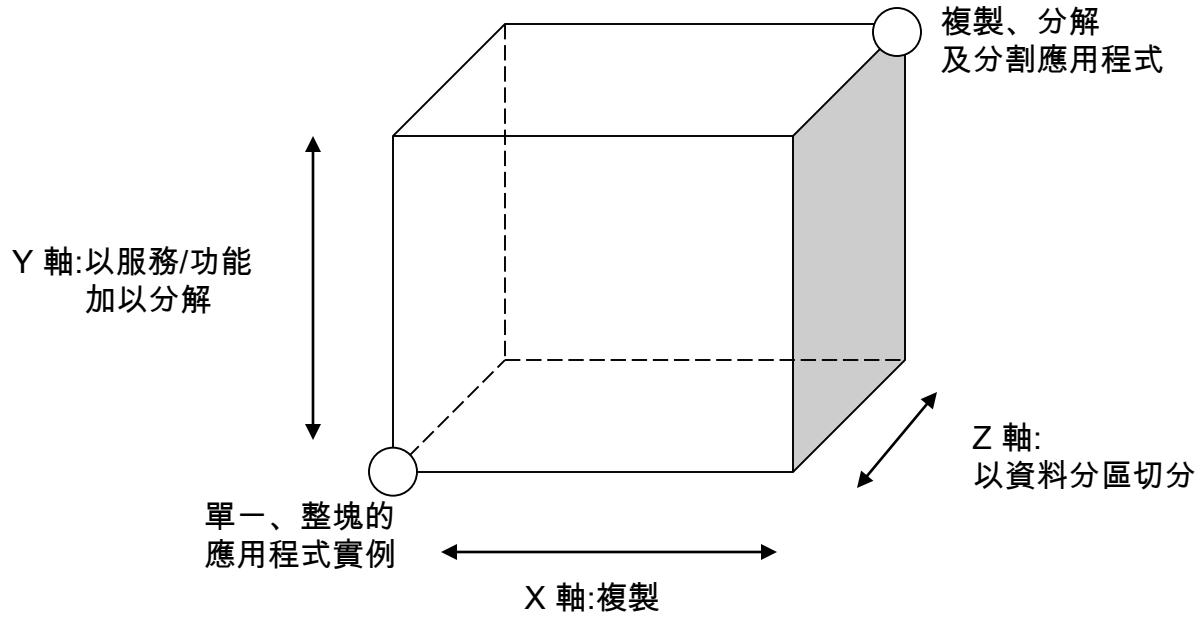
Node.js

Node.js

可擴展性是一種對軟體系統計算處理能力的設計指標，高擴展性代表一種彈性，在系統擴展成長過程中，軟體能夠保證旺盛的生命力，通過很少的改動，就能實現整個系統負載能力的線性增長。

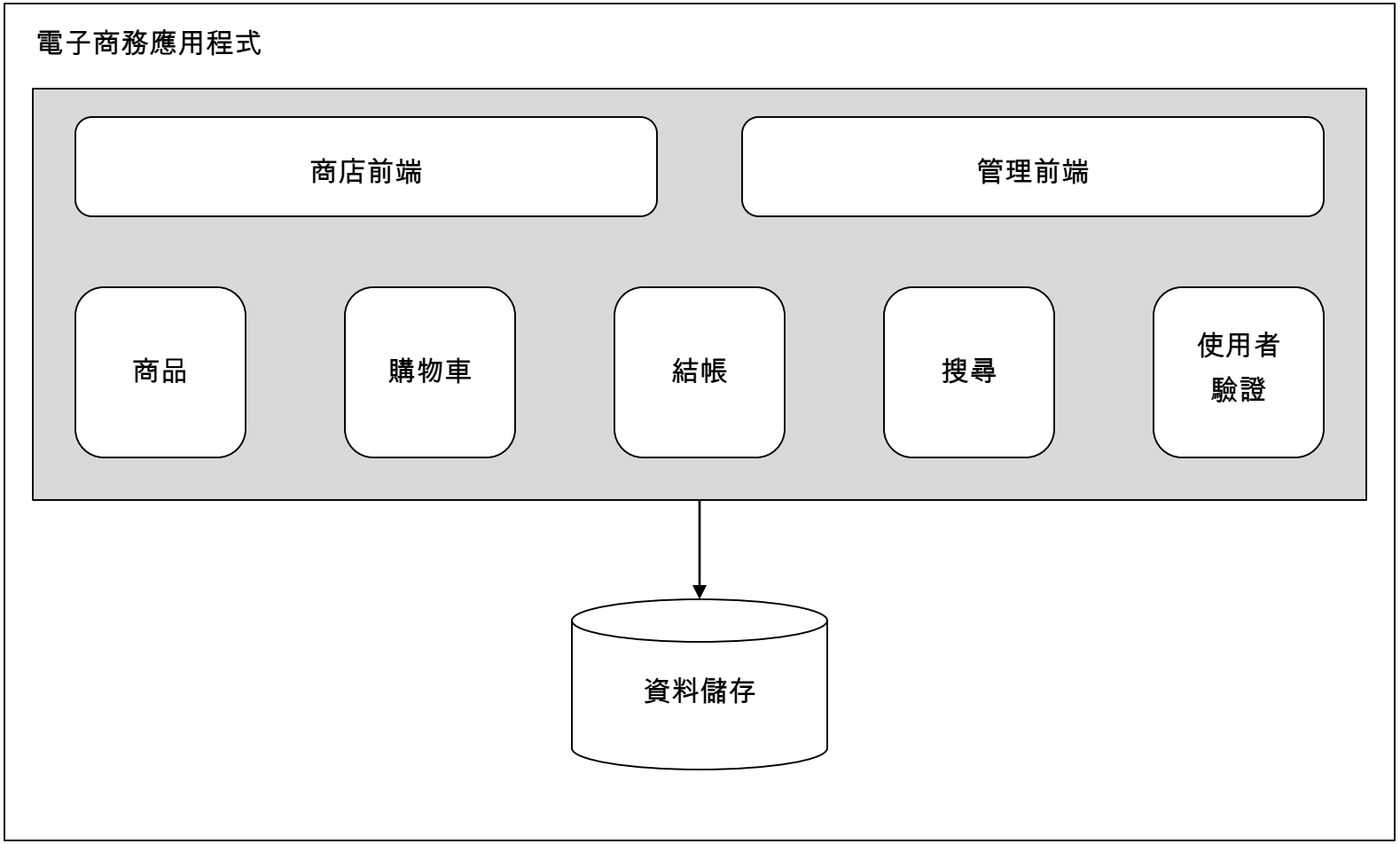
Node.js 雖然是執行於單一執行緒的環境中，但擁有非常彈性的擴展性，而負載能力也並非是擴展 Node.js 應用程式的唯一理由；相反的，它還能增強其他的特性，例如可用性與容錯性。

擴展性有三個維度，第一個需了解的基礎原則是負載分散，也就是將應用程式負載分散到多個程序及機器，這有很多種實現方式，可以藉由一個精巧模型加以呈現，稱之為擴展立方(scale cube)



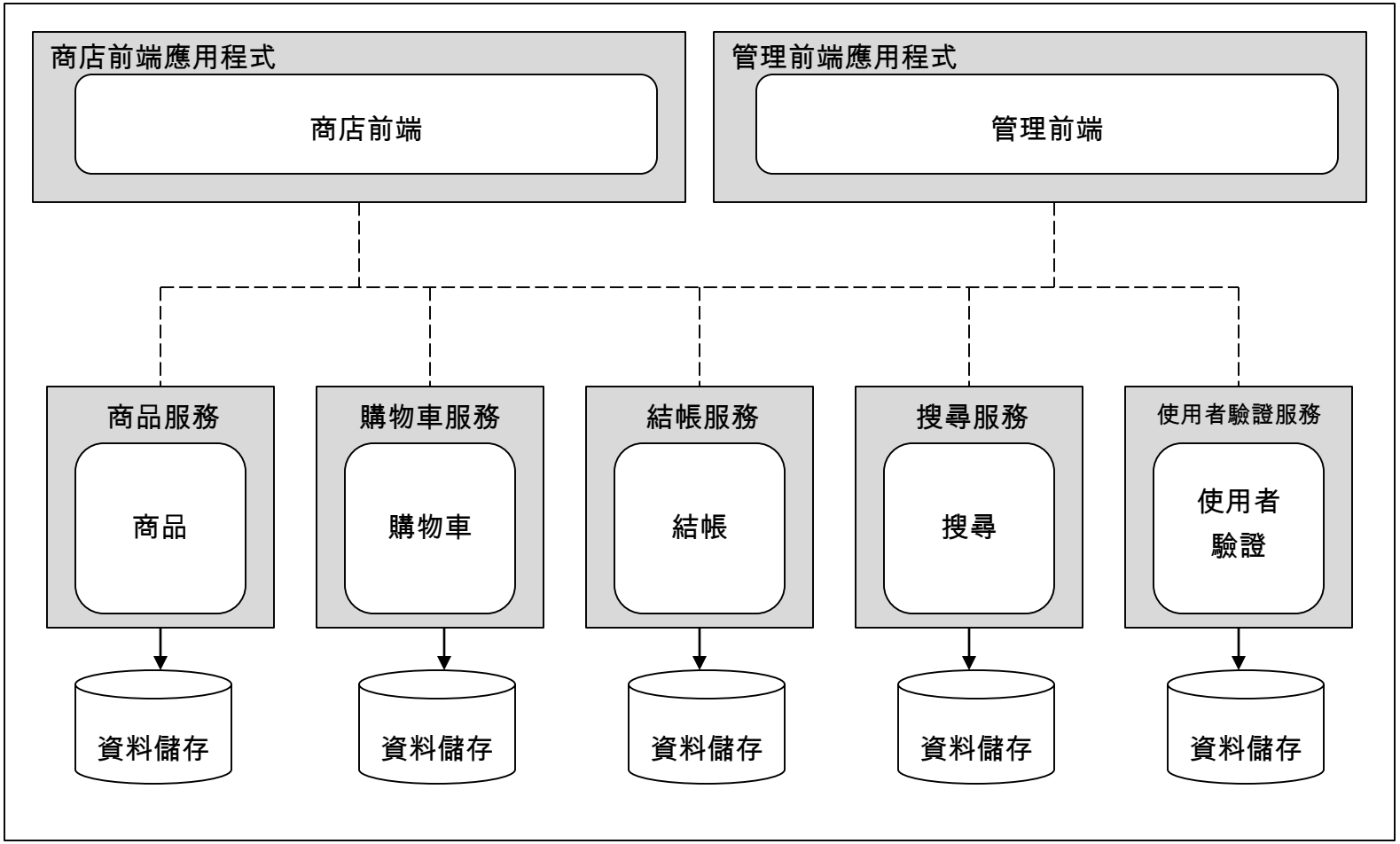
Node.js

單塊架構(monolithic)



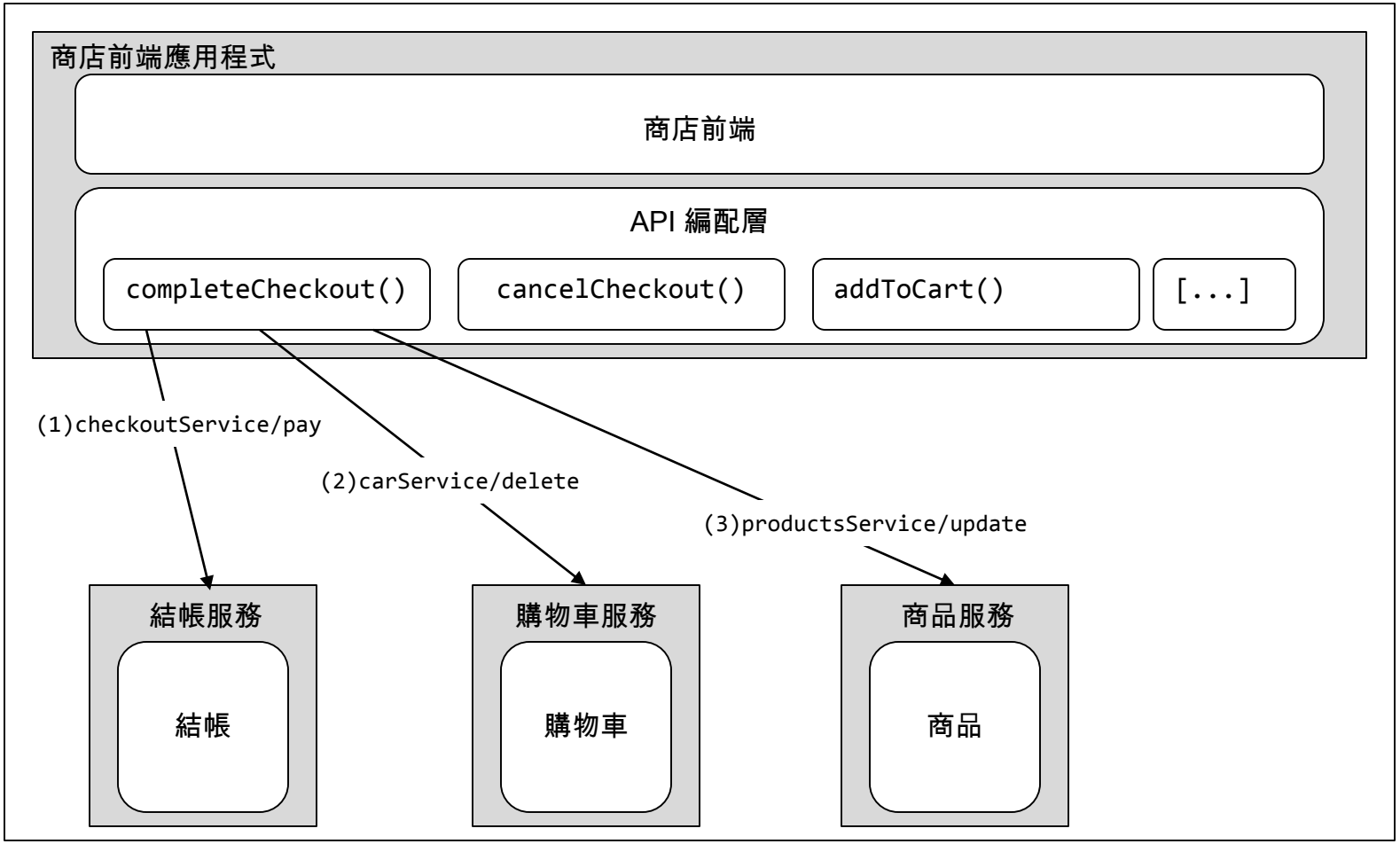
Node.js

微服務架構(microservice)



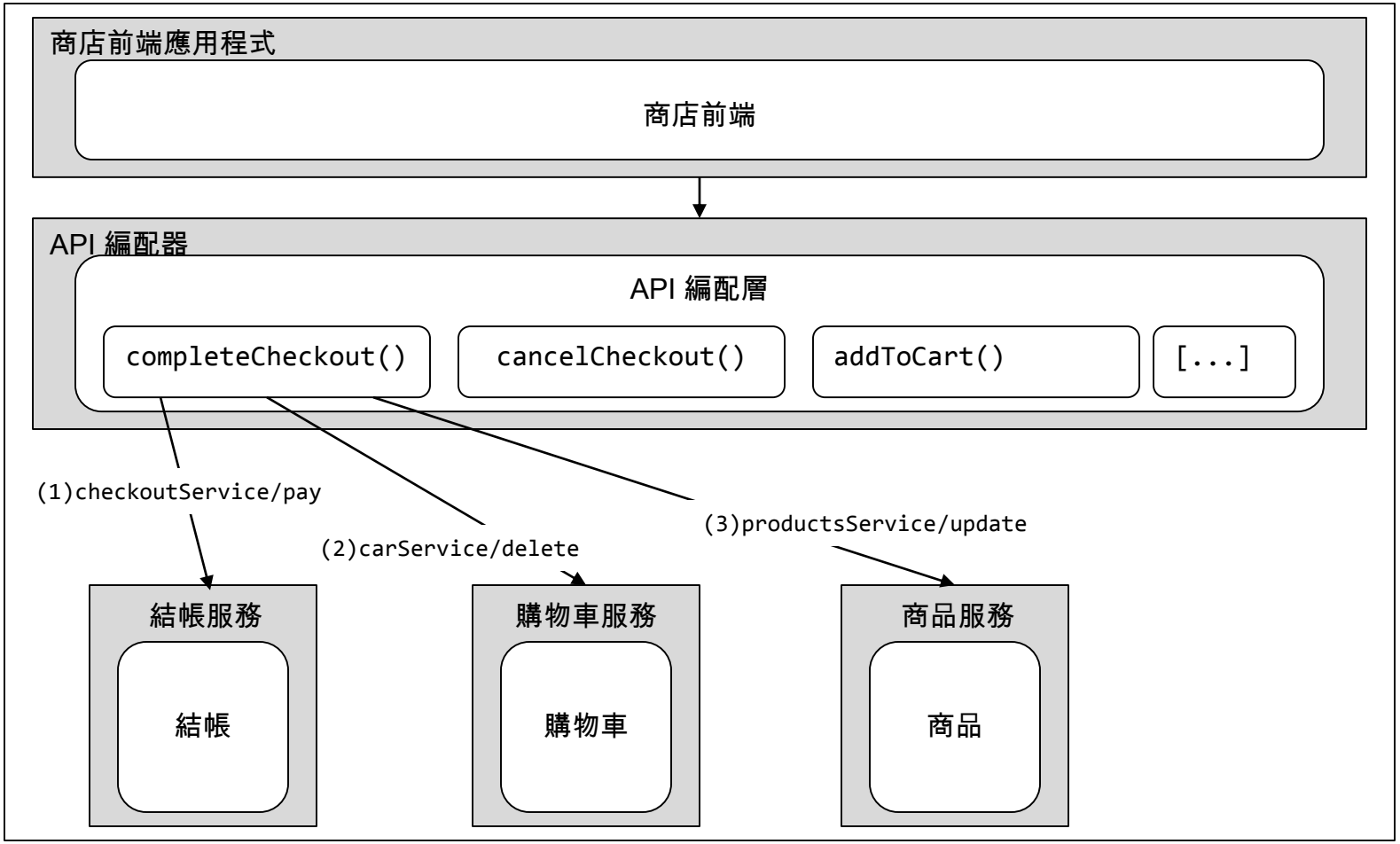
Node.js

微服務架構的整合: API 編配



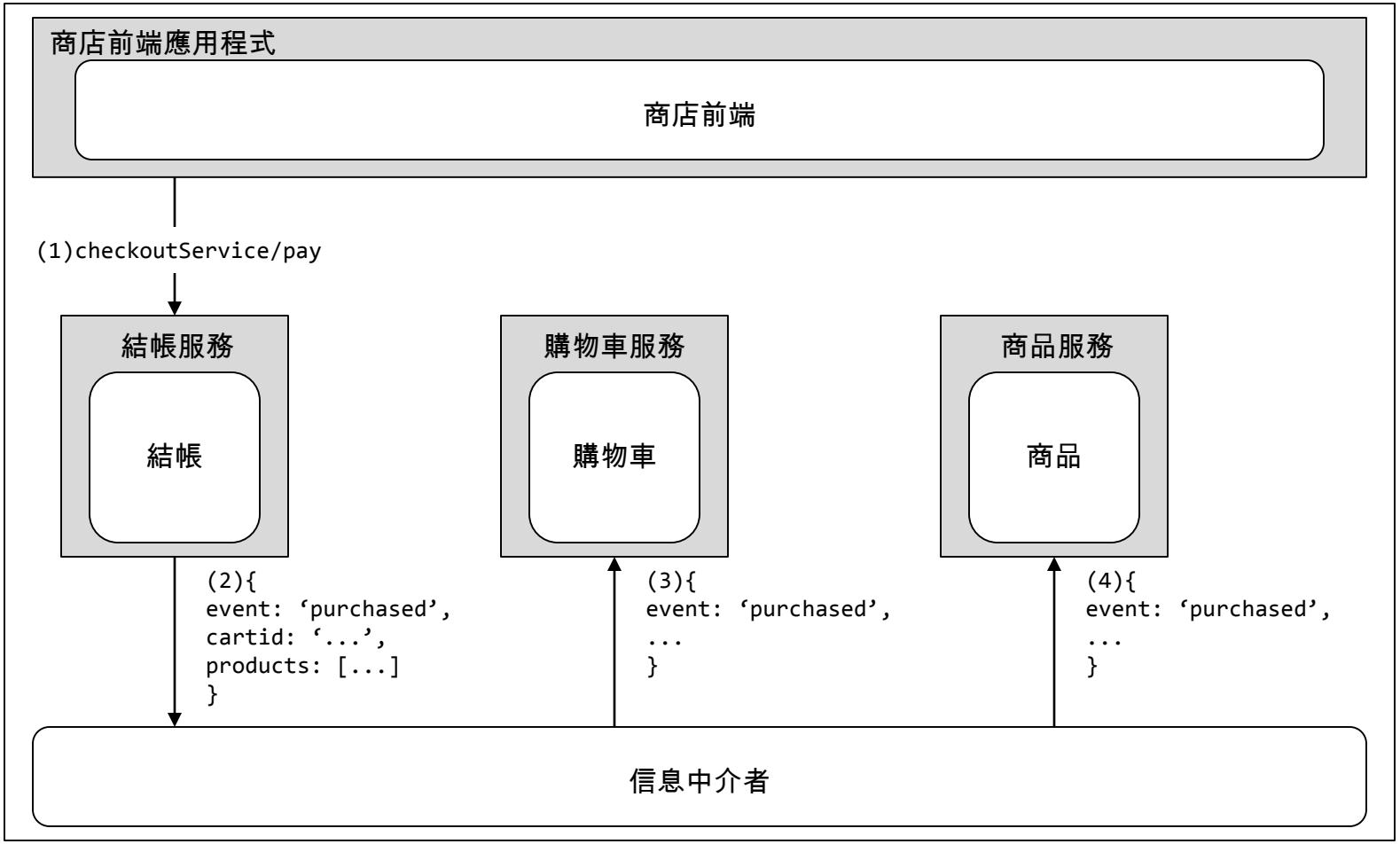
Node.js

微服務架構的整合: API 編配



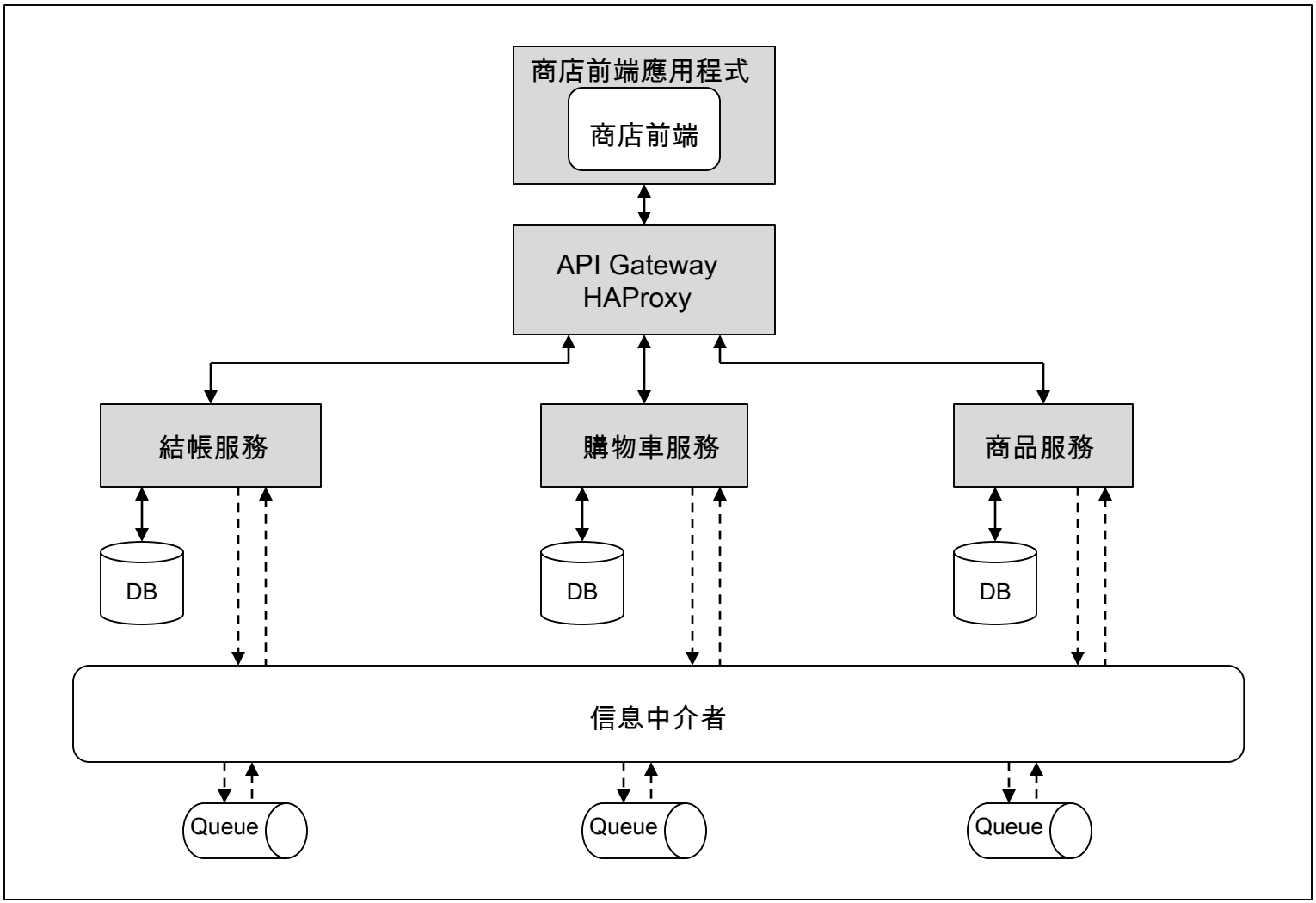
Node.js

微服務架構的整合：訊息中介者



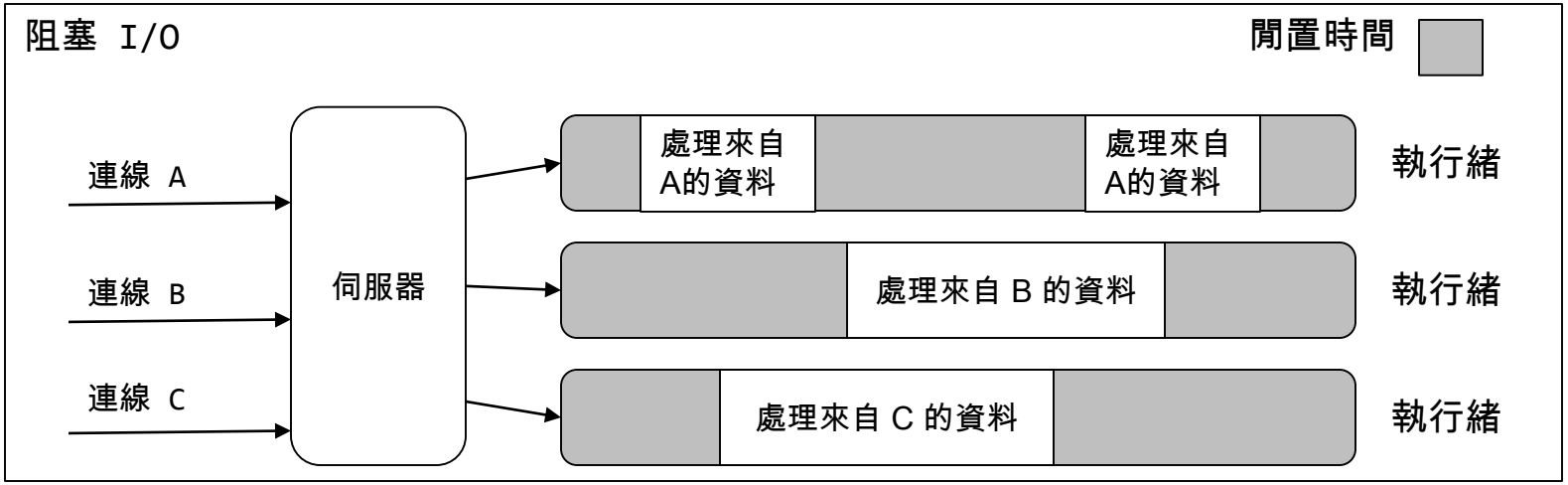
Node.js

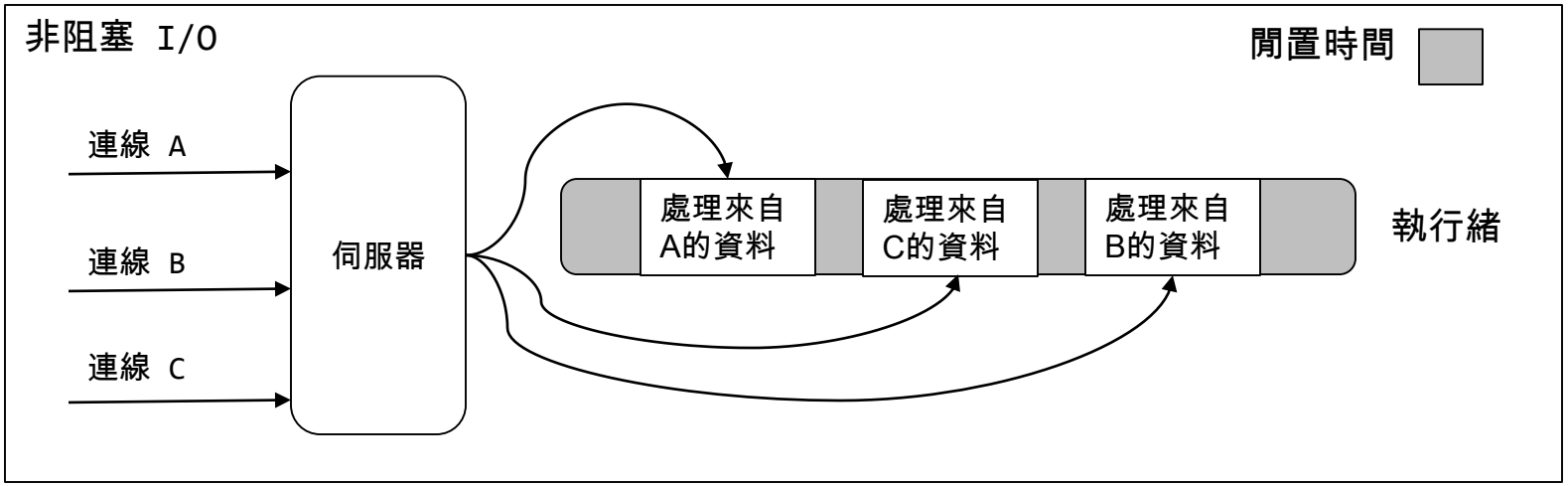
微服務架構的整合



Node.js

單一執行緒與非阻塞 I/O





Node.js

- 事件解多工器 DemultiPlexer

  忙碌等待(busy-waiting)並非處理非阻塞資源的最理想方案，不過令人慶幸的是，多數的當代作業系統都提供一項原生機制，能夠以有效率的方式處理並行以及非阻塞的資源，這項機制被稱為**同步事件解多工器**或**事件通知介面**。這種元件會將受監控資源的 I/O 事件收集起來並排入佇列，然後進行阻擋直到可以處理新事件為止。這項設計就被稱為**事件迴圈**。

- 非阻塞 I/O 引擎 – libuv

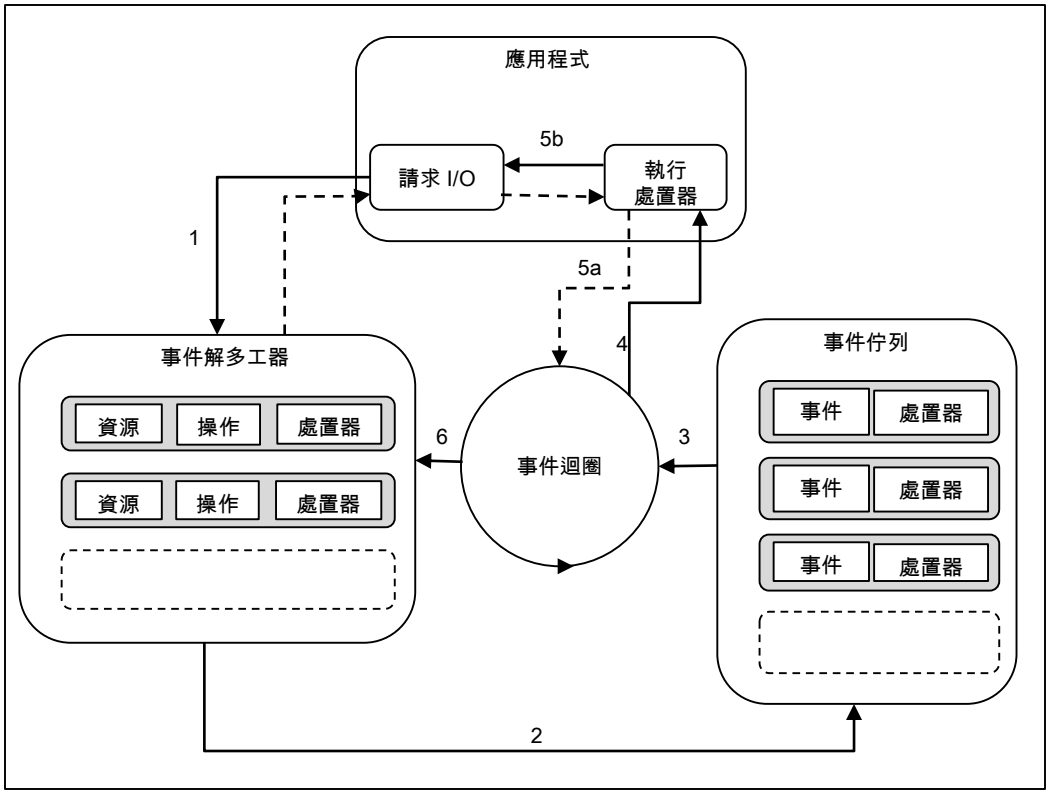
  每個作業系統都有自己的事件解多工器介面，例如 Linux 的 epoll、Mac OS X 的 kqueue，以及 Windows 的 I/O Completion Port API(IOCP)。此外，即便在相同的作業系統下，各 I/O 操作還是可以依資源型態而有全然不同的行為模式。例如，在 Unix 中，正規檔案系統的檔案並不支援非阻塞操作，因此為了模擬非阻塞行為模式，則必須在事件迴圈外部使用一種獨立的執行緒。由於這類矛盾存在於多種不同的作業系統，因此需要更高層級的抽象來建置事件解多工器。於是 Node.js 核心團隊便建立了一套名為 libuv 的 C 函式庫，目地是讓 Node.js 相容於所有的主流平台，並正規化各種資源型態的非阻塞行為模式。libuv 如今是作為 Node.js 的低階 I/O 引擎。

  除了抽象化底層系統的呼叫外，libuv 也實作了反應器模式，因此有提供 API 來產生事件迴圈、管理事件佇列、執行非同步 I/O 操作，以及將其他類型的任務排入佇列。

Node.js

- 反應器模式 (Reactor pattern)

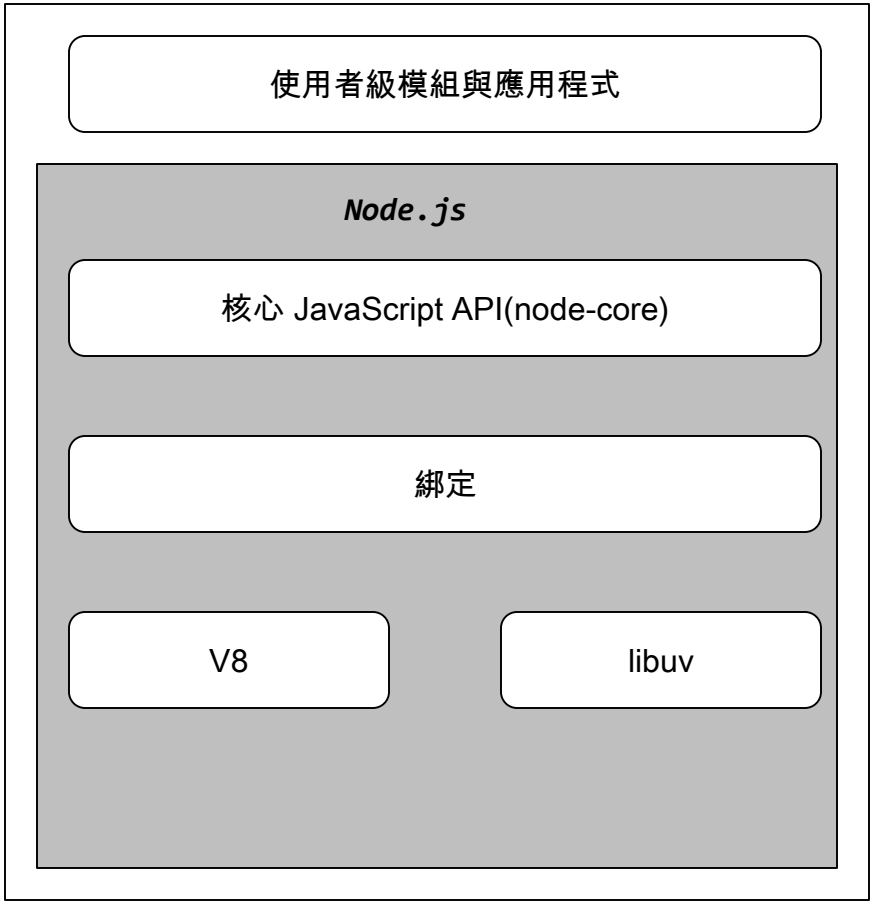
  反應器模式其背後的概念，是擁有一個與各 I/O 操作繫結的**處置器**(在 Node.js 裡是一個**回呼函式 callback**)，會在事件迴圈產生及處理事件時立即呼叫。



1. 應用程式提交請求給**事件解多工器**以產生新的 I/O 操作。此外，應用程式也指定一個**處置器 (handler)**，在操作完成時呼叫。提交新請求給事件解多工器為非阻塞呼叫，它會立即將控制權交回給應用程式。
2. 當一組 I/O 操作完成時，事件解多工器會將新事件推入事件佇列。
3. 此時，事件迴圈會迭代事件佇列裡的項目。
4. 針對每個事件，呼叫相關的處置器。
5. 處置器是應用程式的一部份，會在執行完成時(5a)將控制權交回給事件迴圈。不過，新的非同步操作可能會在處置器執行期間(5b)提出請求，使得新的操作在控制權交回給事件迴圈前，插入至事件解多工器(1)。
6. 當事件佇列裡的所有項目皆處理完畢後，事件解多器迴圈會再度阻塞，直到觸發下一個週期。

Node.js

- Node.js 結構



Node.js

- 回呼模式(callback pattern)

關於反應器模式下的處置器，其具體呈現即是回呼(callback)，而它們可說是形塑出 Node.js 程式設計風格的重要特質之一。回呼是傳送操作結果的函式，而這正是處理非同步操作所需要的，用於取代同步執行的 return 指令。JavaScript 是應用回呼的絕佳語言，因為函式是頭等類別物件，可以輕易的指派給變數、以參數傳遞、自其他函式呼叫回傳，或者是存放在某個資料結構裡。此外，閉包(Closure)是實作回呼的理想架構，使用閉包就能參照函式的原始執行環境。具體來說即是無論回呼何時或何處被呼叫，都能夠保持非同步操作的原始環境。

Node.js

- 延續傳遞風格(CPS)

JavaScript 的回呼是作為參數傳遞給其他函式，並在操作完成時呼叫以傳遞結果。在函式化的程式設計風格理，這種傳遞結果的方式稱為**延續傳遞風格(Continuation-passing style)**，簡稱 **CPS** 。這是一個通用的概念，而不一定與非同步操作相關。事實上，它只是指出操作結果是以傳給另一個函式(回呼)的方式來傳遞，而不是直接傳給呼叫者

```
function add(a, b) {
  return a + b;
}
```

結果由 return 指令回傳給呼叫者。此稱為**直接風格(direct style)**。是同步化程式設計理最常見的結果回傳方式。

```
function add(a, b, callback) {
  callback(a + b);
}

console.log('before');
add(1, 2, function(result) {
  console.log('Result: ' + result);
});
console.log('after');
```

這個 add() 函式是一個同步 CPS 函式，使用回呼回傳其運算的結果。

```
before
Result: 3
after
```

同步延續傳遞風格函式

Node.js

- 延續傳遞風格(CPS)

```
function addAsync(a, b, callback) {
  setTimeout(function() {
    callback(a + b);
  }, 0);
}

console.log('before');
addAsync(1, 2, function(result) {
  console.log(`Result: ${result}`);
});
console.log('after');
```

利用 setTimeout() 模擬非同步的回呼呼叫

```
function addAsync(a, b, callback) {
  process.nextTick(function() {
    callback(a + b);
  });
}
```

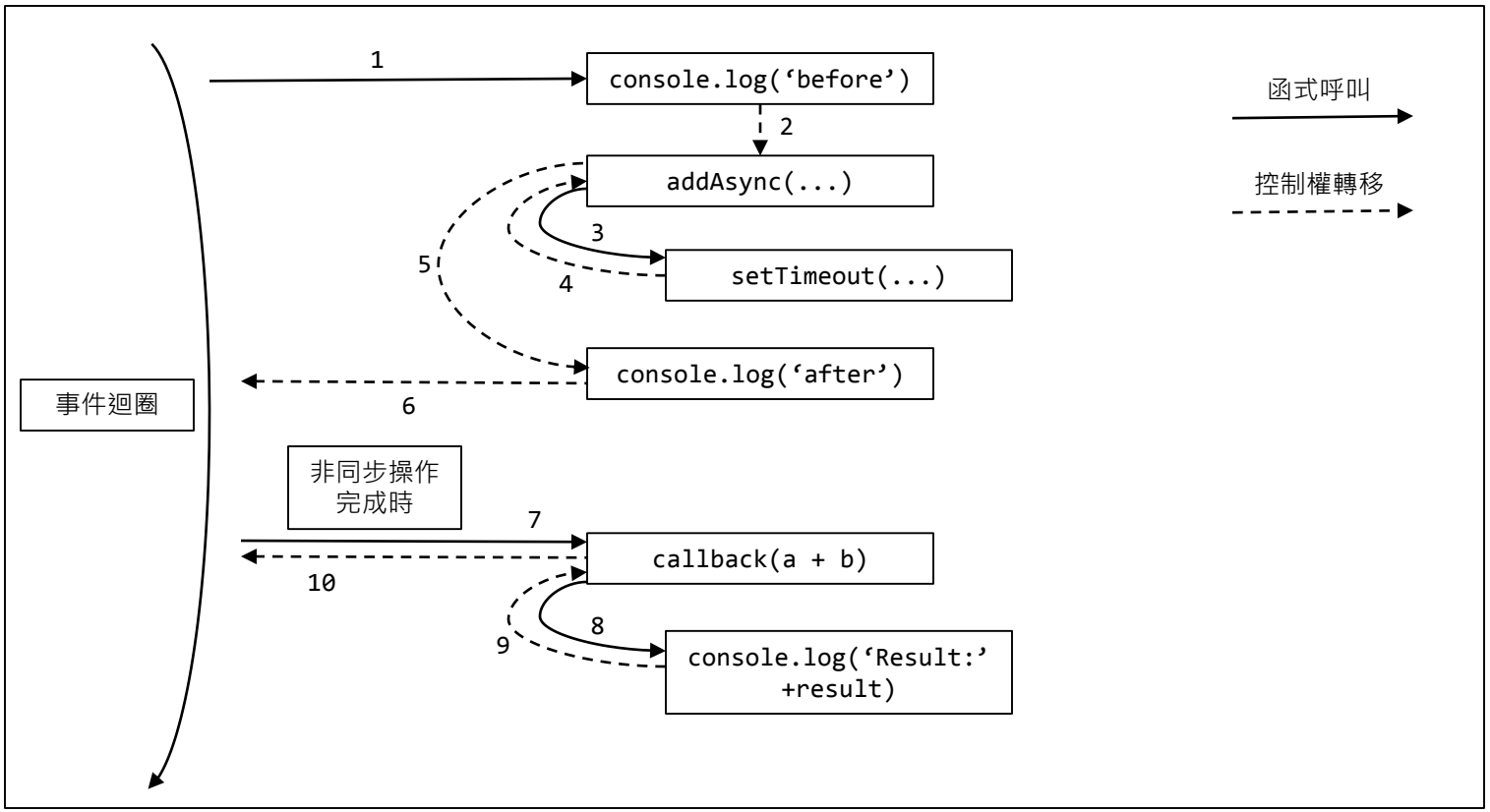
```
before
after
Result: 3
```

非同步延續傳遞風格函式

由於 setTimeout() 觸發了一項非同步操作，因此將不等待任何回呼的執行，而是立即將控制權交回給 addAsync()，接著再返回給該函式的呼叫者。此種性質在 Node.js 裡相當重要，因為他跳脫出了堆疊，在非同步請求發送時，控制權就立即交回給事件迴圈，讓來自佇列的新事件能夠被處理。

Node.js

- 延續傳遞風格(CPS)



當非同步操作完成時，便會執行非同步函式的回呼，也就是先前堆疊的跳脫處。這項執行會從事件迴圈開始，所以會有一個全新堆疊。這裡便顯示出 JavaScript 的好用之處，使用閉包就能夠輕易地保持非同步函式呼叫者的環境，即便在不同時間點或不同位置呼叫回呼也是如此。同步函式會阻塞直到完程操作。非同步函式則會立即返回，其操作結果會在事件迴圈循環的稍後傳遞給處置器(回呼)。

Node.js

- 同步或非同步?

我們已經認識到，指令順序的變化完全是依函式的性質，也就是依同步或非同步而定。這在整個應用程式的資料流中會有明顯的影響，並涉及到正確性及運作效率。我們必須對這兩種模式清楚的辨識其風險。一般來說，必須極力避免的狀況是讓某個 API 的性質出現不一致或混淆，因為這會導致一連串難以偵測與重現的問題。

```
const fs = require('fs');
const cache = {};

function inconsistentRead(filename, callback) {
  if(cache[filename]) {
    callback(cache[filename]);
  } else {
    fs.readFile(filename, 'utf8', (err, data) => {
      cache[filename] = data;
      callback(data);
    });
  }
}
```

一個無法預測的函式

同步呼叫

非同步呼叫

Node.js

- 同步或非同步?

前述函式使用 cache 變數來儲存不同檔案的讀取操作結果。不過切記這只是一項範例，它沒有錯誤管理，且快取邏輯也不是採用最佳的方式。但除此之外，上述函式是相當危險的。因為直到 fs.readFile() 函式回傳結果並設定快取前，它的行為都不同步。但對已存在快取裡的檔案，其後續所有請求卻是同步的：立即觸發對回呼的呼叫。

這個函式的回呼行為模式將很難以預測，因為會受到許多因素的影響，包含呼叫頻率、傳入的檔名，以及載入的時間等。

這類的問題在真實的應用程式造成的臭蟲可能會很複雜，且難以辨識與重現。如果在一個可能會有多個並行請求的網頁伺服器裡，使用類似的函式，則部分請求可能會直接停滯，卻沒有任何顯而易見的原因，也沒有任何錯誤事件紀錄下來，這勢必會成為相當麻煩的問題類型。

這種無法預期的函式在 Node.js 生態中，被比喻為**釋放魔鬼(Unleashing Zalgo)**。

Node.js

- 延遲執行

inconsistentRead() 函式的修正方式即是讓它變成純粹的非同步。我們可以使用小技巧將同步回呼送入事件迴圈排程，而非在同一次事件迴圈循環裡立即執行。在 Node.js 裡，一種可能的作法是利用 process.nextTick()，它可以延遲函式的執行，直到事件迴圈的下一次循環。它的功能很簡單，以回呼作為參數，再將之推入至事件佇列的頂端，也就是所有待處理的 I/O 事件之前，然後立即返回。當事件迴圈再度執行，回呼就會被呼叫。

```
const fs = require('fs');
const cache = {};
function inconsistentRead(filename, callback) {
  if(cache[filename]) {
    process.nextTick(function(){
      callback(cache[filename]);
    });
  } else {
    fs.readFile(filename, 'utf8', (err, data) => {
      cache[filename] = data;
      callback(data);
    });
  }
}
```

Node.js

- Node.js 回呼慣例

在 Node.js 裡，延續傳遞風格 API 與回呼皆依賴了一些慣例。這些慣例是應用於 Node.js 的核心 API，但所有使用者級模組與應用程式也應加以遵循。因此了解它們，並確認我們已遵循了設計非同步 API 所需的一切原則，是相當重要的。

- 回呼置後

在 Node.js 裡，若函式接受回呼，就必須作為最後一個輸入參數。以下列 Node.js 核心 API 為例:

```
fs.readFile(filename, [options], callback)
```

如上述函式的簽名(signature)，回呼永遠置於最後，即便有選擇性參數也是如此。此慣例的用意在於:當回呼被定義在固定位置時，函式呼叫會更容易閱讀。

Node.js

- 錯誤先行

在 CPS 下，錯誤是由操作結果所產生，也就是來自於回呼，在 Node.js 裡，所有 CPS 函式所產生的錯誤，一律是作為回呼的第一個參數傳遞，其他的結果則會從第二個參數以後開始傳遞。若操作結果順利無誤，則第一個參數會是 null 或 undefined。以下程式碼即呈現遵循此慣例的回呼定義方式:

```
fs.readFile('foo.txt', 'utf8', function(err, data) {
  if(err) {
    handleError(err);
  } else {
    processData(data);
  }
});
```

妥善的實作方式是永遠要檢查是否有錯誤存在，否則，就很難為程式除錯並發掘出可能的問題。另一個也必須留意的重要慣例是:錯誤必須為 Error 類型。意即單純的字串或數字都不應該被作為錯誤物件傳遞。

Node.js

- 錯誤傳遞

傳遞錯誤在同步且直接風格的函式裡是以常見的 throw 命令完成，此命令可在呼叫堆疊裡丟出錯誤。然而在非同步 CPS 裡，適切的錯誤傳遞，則是單純地傳遞錯誤給 CPS 鏈的下一個回呼。典型模是如下所述：

```
const fs = require('fs');

function readJSON(filename, callback) {
  fs.readFile(filename, 'utf8', function(err, data) {
    let parsed;
    if (err) return callback(err);
    try {
      parsed = JSON.parse(data);
    } catch(err) {
      return callback(err);
    }
    callback(null, parsed);
  });
};
```

傳遞錯誤並結束現有函式

剖析檔案內容

捕捉剖析的錯誤

無錯誤，僅傳遞資料

Node.js

- 未補獲異常

從前述的 readJSON() 函式裡看到，為避免任何異常被丟進 fs.readFile() 回呼裡，我們針對 JSON.parse() 放置了一個 try-catch 區塊。這是因為若在非同步回呼裡作丟棄，則事件迴圈將會丟出異常，且永遠不會傳遞給下一個回呼。

在 Node.js 裡，這是無法復原的狀態，應用程式將會停止運作並列印出錯誤至 stderr 介面。

```
SyntaxError: Unexpected token n in JSON at position 5
    at JSON.parse (<anonymous>)
    at H:\NodeJS\Projects\node-pattern\temp\cb.js:7:25
    at FSReqWrap.readFileAfterClose [as oncomplete] (fs.js:511:3)
```

Node.js

- 未補獲異常

前述的例子提到，應用程式會在異常抵達事件迴圈時中止，不過，事實上還是有一個最終機會能夠讓它在終止前執行一些清理或留下事件紀錄。當異常發生時，Node.js 會在程式結束前發出名為 uncaughtException 的特殊事件。以下程式碼即展示了一項簡單的使用案例：

```
process.on('uncaughtException', function(err){
  console.log(`This will catch at last the JSON parsing exception:
    ${err.message}`);
  process.exit(1);
});
```

若沒有這一行，應用程式將會繼續執行

```
This will catch at last the JSON parsing exception:
    Unexpected token n in JSON at position 5
```

Node.js

- 揭示模組模式(Revealing module pattern)

JavaScript 最大的問題之一是缺乏命名空間。全域範圍的程式內容，仍會受到內部區域範圍程式碼及相依性的汙染。而解決此問題的普遍技巧為揭示模組模式。

```
const module = (function() {
  let privateFoo = function() {...};
  let privateVar = [];

  let export = {
    publicFoo: function() {...},
    publicBar: function() {...}
  };

  return export;
})();
```

這個模式利用自我呼叫函式(IIFE)來產生私有範圍，並且只匯出欲公開的部份。在程式碼中，module 變數的內容只有匯出的 API，而其餘模組的內容都無法自外部存取。此模式背後的想法，便是 Node.js 模組系統的基礎。

Node.js

- Node.js 模組

在 JavaScript 社群中，有一名為 「CommonJS」的組織，其目標是為 JavaScript 生態建立標準，他們所訂立的模組設計規格如今也已被廣為採用。Node.js 的模組系統便是基於此規格，再加上一些自定的延伸。其設計的方式，就如同揭示模組模式。在此模式下，每個模組皆執行於私有範圍，因此任何區域變數都不會汙染全域命名空間。

Node.js 模組定義方式，就如同下面的範例:

```
const dependency = require('./anotherModule');        載入其他相依性模組

function log() {                                      私有函式
  console.log('Well done ' + dependency.username);
}
module.exports.run = function() {                     匯出公開使用的 API
  log();
};
```

必須謹記的重要觀念是:模組內的一切皆為私有，除非被指派給 module.exports 變數。之後 Node.js 使用 require() 函式載入模組時，該變數內容就會被**快取並回傳**。

Node.js

- Node.js 模組

即便模組所宣告的變數及函式是在區域範圍內定義的，都仍有可能定義為全域變數。事實上，模組系統有個名為 `global` 的特殊變數，就可以用於此目的。被指派給這個變數的一切事物，都將自動存在於全域性範圍中。

但要注意，汙染全域性範圍被認為是不好的實作方式，會失去使用模組系統所帶來的優勢。所以，只有在完全明白自己在做甚麼的情況下才使用它。

Node.js

- require( ) 函式

1. 接收到模組名稱，首先要作的就是解析模組的完整路徑，稱之為 id。
2. 若模組已在先前載入，則應該已存在於快取中。在此情況下，便立即回傳該模組。
3. 若模組尚未載入，便設定初次載入的環境。具體來說就是建立一個 ***module*** 物件，內含以一個空字面物件初始化的 exports 屬性。模組程式碼將利用這個屬性來匯出公開的 API。
4. 將 module 物件加入快取。
5. 自檔案讀取模組原始碼，並進行計值(evaluated)，模組將處理或置換 module.exports 物件，來匯出其公開的 API。
6. 最後，代表該模組公開 API 的 module.exports 內容，將回傳給呼叫者。

另一個必須重視的要點是，require() 函式是同步的，它會單純的使用直接風格來回傳模組內容，無須回呼。

Node.js

- module.exports 與 exports

對許多還不太了解 Node.js 的開發者而言，常見的混淆就是在揭露公開 API 時，使用 exports 與 module.exports 的差異。變數 exports 只是 module.exports 初始值的參照，此值本質上是一個簡單的字面物件，建立於模組載入之前。

這表示只能附加**新屬性**於 exports 變數所參照的物件

```
exports.hello = function() {
  console.log('Hello');
};
```

重新指派 exports 變數不會有任何效果，因為不會改變 module.exports 的內容，而僅會重新指派變數本身。因此下面的程式碼是錯誤的：

```
exports = function() {
  console.log('Hello');
};
```

這指定一個新物件給 exports 變數，失去了原先對 module.exports 的參照。

Node.js

- module.exports 與 exports

若想匯出初始字面物件以外的項目，例如函式、實例或字串等等，就必須重新指派 module.exports

```
module.exports = function() {
  console.log('Hello');
};
```

取代了初始的字面物件，匯出了一個函式。

Node.js

- 模組路徑解析

require() 函式接受到模組名稱，會開始解析模組完整的路徑，此路徑之後將用於載入程式碼，同時也是作為該模組的唯一識別碼。解析演算法可切分為以下三個主要分支:

- 檔案模組: 若 moduleName 開頭為「/」，即認定是模組的絕對路徑，並按原樣回傳。若以「./」起始，則 moduleName 會被認為是相對路徑，會從提出要求的模組開始推測。
- 核心模組: 若 moduleName 前沒有「/」或「./」，則演算法會先試著在核心 Node.js 模組裡搜尋。
- 套件模組: 若沒有在核心模組裡找到符合的 moduleName，則會在提出要求的模組目錄結構裡，搜尋 node_modules 目錄，直到出現符合模組，或全目錄搜尋完畢。

就檔案模組與套件模組而言，moduleName 可與個別檔案及目錄進行比對。具體來說，此演算法會試圖比對:

- <moduleName>.js
- <moduleName>/index.js
- <moduleName>/package.json 的 main 屬性裡指定的目錄/檔案

Node.js

- 執行程序的兩個階段

考慮到阻塞對 Node.js 事件循環的成本，您可能認為使用同步文件訪問方法都不是好主意。要了解何時可以使用同步的方式，您可以將 Node.js 程序視為兩個階段

- 在初始化階段，程序正在設置，引入程式庫，讀取配置參數(configuration)以及執行其他關鍵任務。如果在這個早期階段出現問題，可以做的不多，最好快速讓程式失敗。您應該考慮同步文件訪問的唯一時間，就是在程序的初始化階段。
- 第二階段是操作(operation)階段，當程序通過事件循環進行時。由於許多 Node.js 程序是聯網的，會接受網路連接(connections)，發出請求(requests)以及等待其他類型的 I/O. 在此階段，您永遠不應該使用同步文件訪問方法。

require() 函數是這個原則的一個例子，它同步評估(evaluates)目標模塊(module)的代碼並返回模塊物件(module object)。模塊一定得成功加載，否則程序將立即失敗。

根據經驗，如果你的程序沒有此程式就不可能繼續往下執行，那麼可以使用同步文件訪問。如果您的設計沒有此程序，而仍然可以繼續執行其它業務。那麼最好採取安全的方式，並堅持非同步 I/O.

Node.js

- 模組定義模式

模組系統除了作為載入相依性的機制外，同時也是定義 API 的工具。因為針對 API 的設計問題，主要的考量因素往往是私有與公開功能間的平衡。我們想要將 API 的資訊隱藏與可用性最大化，同時也想要維持程式的擴充性與重複使用性。以下是常見的模組定義模式，對此都有一定程度的平衡。

- 具名匯出

揭露公開 API 最基本的方式是使用具名匯出，藉由指派想予以公開的值給 exports(或 module.exports)所參照的物件屬性。在這種方式下，所匯出的物件會成為一個容器或命名空間，裝載著一組相應功能。

```
// 檔案 logger.js
exports.info = function(message) {
  console.log('info: ' + message);
};

exports.verbose = function(message) {
  console.log('verbose: ' + message);
};
```

Node.js

- 模組定義模式

匯出的函式接下來可作為已載入模組的屬性，如下列程式碼所示：

```
// 檔案 main.js
const logger = require('./logger');

logger.info('This is an informational message');
logger.verbose('This is a verbose message');
```

大多數的 Node.js 核心模組都使用了此模式。

CommonJS 規格只允許使用 exports 變數來揭露公開成員。因此，具名匯出模式是唯一真正相容於 CommonJS 規格的做法。至於 module.exports 則是 Node.js 的自訂延伸，為的是支援更多種模組定義方式。

Node.js

- 模組定義模式

  - 匯出函式

有一種相當受歡迎的模組定義模式，是重新指派整個 module.exports 變數給某函式。它的主要優點在於只會揭露單一功能，為模組提供了清晰的入口，使它易於瞭解與使用。

```
// 檔案 logger.js
module.exports = function(message) {
  console.log('info: ' + message);
};
```

擴充此模式的一種可能方式，是將匯出函式作為其它公開 API 的命名空間。這是相當強大的結合，因為它既為模組提供了清晰的單一入口(主匯出函式)，且又允許揭露其他次要或更進階的功能。此又稱為「子堆疊模式(substack)」，下列程式碼展示如何利用匯出函式作為命名空間，來擴充先前定義的模組:

```
module.exports.verbose = function(message) {
  console.log('verbose: ' + message);
};
```

Node.js

- 模組定義模式

```
// 檔案 main.js
const logger = require('./logger');

logger('This is an informational message');
logger.verbose('This is a verbose message');
```

只匯出一個函式，看起來似乎頗為受限，但其實這是強調單一功能的完美方案，這對模組而言尤其重要。它降低了次要面向的能見度，使之作為匯出函式的屬性被揭露出來。

這種定義模組的方式又稱為「子堆疊模式」(substack): 僅匯出單一函式來揭露模組的主要功能。使用匯出函式作為命名空間，來揭露所有附屬功能。

Node.js

- 模組定義模式

  - 匯出建構子

匯出建構子的模組為匯出函式模組的特別版本。差異在於這個新模式允許使用者既使用建構子來產生新實例，且又提供它們擴充自有原型及打造新類別的能力。

```
// 檔案 logger.js
function Logger(name) {
  this.name = name;
}
Logger.prototype.log = function(message) {
  console.log('[' + this.name + '] ' + message);
};
Logger.prototype.info = function(message) {
  this.log('info: ' + message);
};
Logger.prototype.verbose = function(message) {
  this.log('verbose: ' + message);
};
module.exports = Logger;
```

Node.js

- 模組定義模式

接著，可以如下使用前述模組。

```
// 檔案 main.js
const Logger = require('./logger');
const dbLogger = new Logger('DB');
dbLogger.info('This is an informational message');

const oracleLogger = new Logger('ORACLE');
oracleLogger.verbose('This is a verbose message');
```

```
[DB] info: This is an informational message
[ORACLE] verbose: This is a verbose message
```

匯出建構子同樣為模組提供了單一的入口，不過和子堆疊模式比較起來，它則揭露了更多模組內部。然而若從其他方面來看，它是更具備擴充自有功能的能力。

Node.js

- 模組定義模式

此模式的一種變體是增加一道防線，來防範未使用 new 指令的呼叫，這項小技巧讓我們能夠將模組作為「工廠」來使用。

```
function Logger(name) {
  if(!(this instanceof Logger)) {
    return new Logger(name);
  }
  this.name = name;
}
```

這項技巧很簡單，檢查 this 是否存在且為 Logger 的實例。若這些條件任一為否，就代表未使用 new 呼叫 Logger() 函式，因此便建立適切的新實例並回傳給呼叫者。此技巧能夠將模組作為工廠來使用。

```
const Logger = require('./logger');
const dbLogger = Logger('DB');
dbLogger.info('This is an informational message');
```

Node.js

- 模組定義模式
  - 匯出實例

我們可以利用 require() 的快取機制，來輕鬆定義狀態性實例，這由建構子或工廠所產生，可以在模組間共享的含狀態物件。

```
// 檔案 logger.js
function Logger(name) {
  this.count = 0;
  this.name = name;
}
Logger.prototype.log = function(message) {
  this.count++;
  console.log('[' + this.name + '] ' + message);
};
module.exports = new Logger('DEFAULT');
```

```
// 檔案 main.js
const logger = require('./logger');
logger.log('This is an informational message');
```

Node.js

- 模組定義模式

由於 logger 模組被快取，因此要求該模組的所有模組其實都會取得相同的物件實例，也就共享了它的狀態。此模組非常類似於單例(singleton)，然而它並不保證會像傳統的單例模式那樣，在整個應用程式中保持單一性。

這個模式還有一種擴充方式，是除了實例自身外，也揭露用於產生實例的建構子。這可以讓使用者建立相同物件的新實例，必要時甚至可以擴充。

```
module.exports.Logger = Logger;
```

```
// 檔案 main.js
const logger = require('./logger');
customLogger = new logger.Logger('CUSTOM');
customLogger.log('This an informational messsage');
```

從可用性方面來看，這類似於使用匯出函式作為命名空間。模組會匯出某物件的預設實例(基本功能)，而一些較進階的功能，例如產生新實例或擴充物件，則可以透過可見度較低的屬性來提供。

Node.js

- 觀察者模式(Observer pattern)

在 Node.js 裡，另一個重要的基礎模式是**觀察者模式**，這個模式連同反應器、回呼與模組等，是一同作為 Node.js 平台的基石，且為許多核心級與使用者級模組的必要機制。

觀察者是形塑 Node.js 反應性質的理想解決方案，還能作為回呼的完整補充。其正式定義如下：

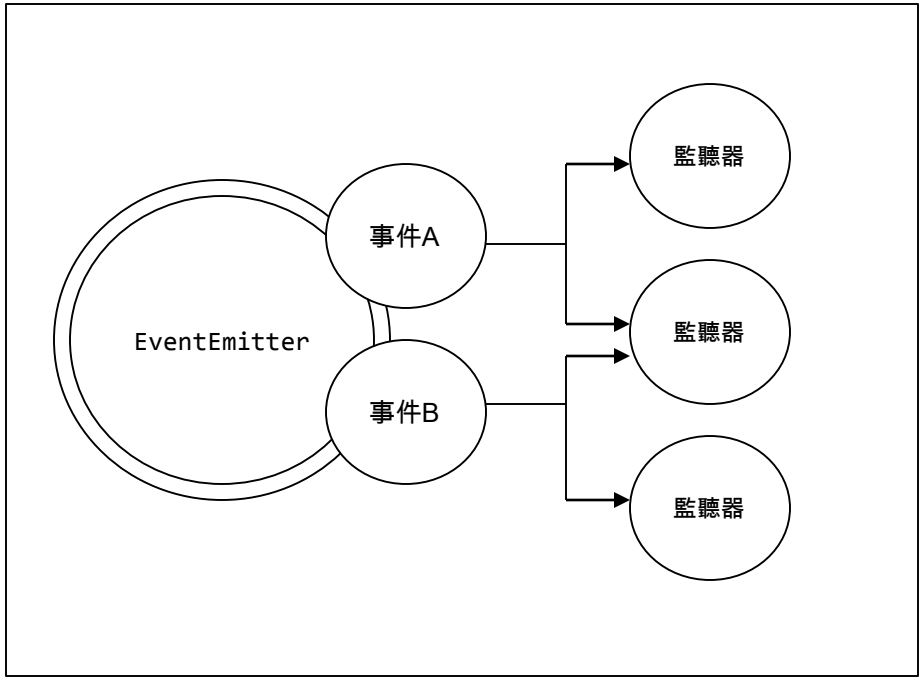
觀察者模式: 定義物件(稱為主體 subject)於狀態改變時，能夠通知多個觀察者(或監聽者)

與回呼模式的主要差異在於主體能夠通知多個觀察者，而在傳統的延續傳遞風格下，回呼通常只會傳遞結果給單一監聽器(即是回呼)。

Node.js

- 觀察者模式(Observer pattern)
  - EventEmitter

在傳統的物件導向程式設計中，觀察者模式需要介面、具體類別與繼承。然而在 Node.js 裡，一切更為簡單。觀察者模式已內建於核心中，而且可透過 EventEmitter 類別取用，EventEmitter 類別允許我們註冊一或多個函式為監聽器，然後會在特定事件類型觸發時呼叫。



Node.js

- 觀察者模式(Observer pattern)

EventEmitter 是一個原型，自 events 核心模組匯出。下列程式碼說明如何為其取得參照：

```
const EventEmitter = require('events').EventEmitter;
const eeinstance = new EventEmitter();
```

EventEmitter 的基本方法如下：

- on(event, listener): 此方法允許針對指定的事件類型(字串)註冊新監聽器(函式)。
- once(event, listener): 此方法註冊新監聽器，接著於事件首次發出後移除。
- emit(event, [arg1], [...]): 此方法產生新事件並提供附加參數傳送給監聽器。
- removeListener(event, listener): 此方法移除特定事件類型的監聽器。

上述的所有方法都會回傳 EventEmitter 實例來允許鏈接。listener 函式擁有簽名 function([arg1], [...])，因此會接受事件發出時所提供的參數。在監聽器內部，this 會指向產生事件的 EventEmitter 實例。(可以使用 ES6 箭頭函數作為偵聽器，但是，這樣做時，this 關鍵字將不再引用 EventEmitter 實例)

可以從這裡看見監聽器與傳統 Node.js 回呼的重大差異，特別是監聽器的第一個參數無須保留給錯誤事件，可以是任何在呼叫時傳給 emit() 的資料。

Node.js

- 觀察者模式(Observer pattern)

來看看如何實際使用 EventEmitter，我們將 add() 函式改用 EventEmitter。

```
const EventEmitter = require('events').EventEmitter;

function add(a, b) {
  let eventEmitter = new EventEmitter();
  setTimeout(function() {
    if (Math.random() < 0.5) {
      return eventEmitter.emit('error', new Error('error.'));
    }
    eventEmitter.emit('add', a + b);
  }, 0);
  return eventEmitter;
}
```

上述的函式將產生兩個事件：將結果透過 add 事件傳達， error 事件則會傳達錯誤事件。

Node.js

- 觀察者模式(Observer pattern)

現在來看看這個 add 函式可以如何應用:

```
add(1, 2)
  .on('add', function(result) {
    console.log(result);
  })
  .on('error', function(err) {
    console.log(err);
  });
```

這裡我們為 add() 函式中的 EventEmitter 所產生的兩個事件類型 add 與 error，都分別註冊一個監聽器。

EventEmitter 如同回呼，在錯誤狀況發生時不應只是丟出異常，因為若事件為非同步發出，它們就會消失在事件迴圈裡。反之，正確的使用慣例是發出一個名為 error 的特殊事件，再以參數傳第一個 Error 物件。我們的 add() 函式裡就是這麼作的。

而為 error 事件註冊監聽器一直都是比較好的實作方式，使 Node.js 對其做特殊處理。

Node.js

- 觀察者模式(Observer pattern)

來看看一個比較實際的例子，我們從最簡單的方式即是建立新實例然後直接使用。以下程式碼會在檔案內容中發現特定字串模式時，即時通知期訂閱者(findPattern.js):

```
const EventEmitter = require('events').EventEmitter;
const fs = require('fs');

function findPattern(files, regex) {
  const emitter = new EventEmitter();
  files.forEach(file => {
    fs.readFile(file, 'utf8', (err, content) => {
      if (err) return emitter.emit('error', err);

      emitter.emit('fileread', file);
      let match;
      if (match = content.match(regex)) {
        match.forEach(elem => emitter.emit('found', file, elem));
      }
    });
  });
  return emitter;
}
```

Node.js

- 觀察者模式(Observer pattern)

現在來看看這個 findPattern() 函式可以如何應用:

```
findPattern(
  ['fileA.txt', 'fileB.json'],
  /hello \w+/g
)
.on('fileread', file => console.log(`${file} was read`))
.on('found', (file, match) => console.log(`matched ${match} in file ${file}`))
.on('error', err => console.log(`Error emitted: ${err.message}`));
```

有時，直接由 EventEmitter 類別建立一個新的可觀察物件並不足夠，因為用此流程來提供建立新事件以外的功能不甚實際。因此，事實上較常見的作法是另外建立一種流程，**讓一般物件變為可觀察**，借由擴充 EventEmitter 類別就可以做到這點。

Node.js

- 觀察者模式(Observer pattern)

現在我們要在一個物件中實作 findPattern() 函式功能:

```
const EventEmitter = require('events').EventEmitter;
const fs = require('fs');

class FindPattern extends EventEmitter {
  constructor (regex) {
    super();
    this.regex = regex;
    this.files = [];
  }

  addFile(file) {
    this.files.push(file);
    return this;
  }
```

Node.js

- 觀察者模式(Observer pattern)

```
  find () {
    this.files.forEach(file => {
      fs.readFile(file, 'utf8', (err, content) => {
        if (err) {
          return this.emit('error', err);
        }
        this.emit('fileread', file);

        let match = null;
        if (match = content.match(this.regex)) {
          match.forEach(elem => this.emit('found', file, elem));
        }
      });
    });
    return this;
  }
}
```

Node.js

- 觀察者模式(Observer pattern)

我們在此使用 ES6 的新功能 class 定義了 FindPattern 建構式，繼承了 EventEmitter。藉由此方式，它便會成為一個完善的可觀察類別。以下即為使用範例：

```
const findPatternObject = new FindPattern(/hello \w+/g);
findPatternObject
  .addFile('fileA.txt')
  .addFile('fileB.json')
  .find()
  .on('found', (file, match) => console.log(`Matched "${match}" in file
${file}`))
  .on('error', err => console.log(`Error emitted ${err.message}`));
```

現在我們可以看到 FindPattern 擁有完整的一組方法，並且經由繼承 EventEmitter 功能成為可觀察的物件。

在 Node.js 生態體系裡是相當常見的模式，舉例來說，核心 http 模組的 Server 物件即定義了諸如 listen()、close()、setTimeout() 等方法，而在其內部也繼承了 EventEmitter 函式的方法，因此能夠產生事件。例如在接收到新請求時產生 request、或是在新連線建立時產生 connection，以及在伺服器關閉時產生 closed。

關於以物件擴充 EventEmitter，還有一項值得注意的範例是 Node.js 串流(Streams)。

Node.js

- 觀察者模式(Observer pattern)

如同回呼，事件也可以同步或非同步發出，因此絕對不要在相同的 EventEmitter 裡混用這兩種處理方式，這點非常重要。但更重要的是，在發出相同的事件類型時，要避免類似「釋放魔鬼」的相同問題。

同步或非同步發出事件的主要差異在於監聽器的註冊方式。當事件非同步發出時，使用者在任何時候都能註冊新的監聽器，即使是在 EventEmitter 初始化後，因為直到事件迴圈的下一個循環前都不會發出事件。這便是 findPattern() 函式的實際運作情形，也就是我們先前所定義的那個函式，它呈現出 Node.js 模組最常使用的處理方式。

相反的，事件若是同步發出，則所有監聽器必須在 EventEmitter 函式開始發出任何事件前註冊。來看看以下範例:

Node.js

- 觀察者模式(Observer pattern)

```
const EventEmitter = require('events').EventEmitter;

class SyncEmit extends EventEmitter {
  constructor(){
    super();
    this.emit('ready');
  }
}

const syncemit = new SyncEmit();
syncemit.on('ready', () => console.log('Object is ready to be used'));
```

若 ready 事件非同步發出，上述程式碼會完美運作；但若事件同步產生，則監聽器在事件傳送後註冊。結果會是監聽器永遠不會被呼叫，而程式碼也不會在終端機上印出任何東西。

有別於回呼，EventEmitter 在某些情況下，以同步方式來使用是合理的。因此在導入 EventEmitter 時，於說明文件裡明白指出其行為模式相當重要，可避免混淆及可能的誤用。

Node.js

- EventEmitter vs Callback

在定義非同步 API 時，常見選擇是決定使用 EventEmitter 還是簡單的接受回呼。一般的區分標準是根據語義:回呼應於必須非同步回傳結果時使用，而事件則應該用於當某件事情發生時的傳達行為。

然而除了這項簡單原則，其實兩種範式在多數時候是相同的，也可以達成相同結果。它們真正的差異在於可讀性、語義，以及實作或使用時所需的程式碼多寡。

乍看之下，回呼對於支援不同的事件類型，存在著一些限制。但事實上還是能夠以回呼參數來傳遞事件類型或接收數個回呼(每一個都對應不同的事件)，藉此區隔多個事件。然而，這樣的 API 設計並不優雅。反之，EventEmitter 能夠提供較佳的介面及更加簡潔的程式碼。

較適合使用 EventEmitter 的另一種可能狀況是:當相同事件可能發生多次，或完全不會發生的時候。由於回呼預期只會被呼叫一次，無論操作成功與否。因此若面臨可能存在重複性的情況，就得再次思考其語義性質。顯然它比較類似於一個必須傳達的事件而非一個結果。在這種狀況下，EventEmitter 就會是比較好的選擇。

最後，使用回呼的 API 只能通知特定回呼，但使用 EventEmitter 函式則能夠讓多個監聽器接收到相同的通知。

Node.js

- 檔案系統

  - 監看文件的變化

作為程序員，你有可能在某些時候訪問文件系統：讀取，寫入重命名和刪除文件。 觀察文件的變化是一個很方便的問題，因為它需要異步編碼，同時展示了重要的 Node.js 概念。

```
$ mkdir filesystem
$ cd filesystem
$ touch target.txt
$ echo, > target.txt
```

```
// filesystem/watcher.js
const fs = require('fs');
fs.watch('target.txt', () => console.log('File changed!'));
console.log('Now watching target.txt for changes...');
```

Node.js

- 檔案系統

  - 讀取命令行參數

現在讓我們通過將文件名稱作為命令行參數來使我們的程序更有用些。 這將使用 process 全域物件並了解 Node.js 如何處理異常狀況。

```
// watcher-argv.js
const fs = require('fs');
const filename = process.argv[2];
if(!filename) {
  throw Error('A file to watch must be specified!');
}
fs.watch(filename, () => console.log(`File ${filename} changed`));
console.log(`Now watching ${filename} for changes...`);
```

process 是一個全域物件，process.argv[2] 將取得命令行的第三個參數

使用 **throw** 拋出一個 **Error** 物件實例

Template literals

```
$ node watcher-argv target.txt
Now watching target.txt for changes...
```

target.txt 是命令行的第三個參數

Node.js

- 檔案系統

  - 產生一個子行程

現在讓我們產生一個子行程來回應文件的更改，進一步增強我們的文件監視範例的程序。我們將引用 Node.js 的子行程模塊(child-process)，並深入研究一些 Node.js 如何使用串流(Streams)來管理數據。

```
// watcher-spawn.js
const fs = require('fs');
const spawn = require('child_process').spawn;
const filename = process.argv[2];

if (!filename) {
  throw Error('A file to watch must be specified!');
}

fs.watch(filename, () => {
  const ls = spawn('cmd.exe', ['/c','dir', filename]);
  ls.stdout.pipe(process.stdout);
});
console.log(`Now watching ${filename} for changes...`);
```

當檔案有異動時，產生一個子行程執行 Windows 命令模式下的 dir 指令

使用串流將結果導至標準輸出

Node.js

- 檔案系統
  - 產生一個子行程

現在執行它，然後更動檔案內容。

```
$ chcp 65001                          切換到 Unicode，否則會看到一堆亂碼
$ node watcher-spawn target.txt
Now watching target.txt for changes...

Volume in drive H is 新增磁碟區          當 target.txt 有異動時
 Volume Serial Number is E417-6764

 Directory of H:\NodeJS\Projects\node-pattern\temp\filesystem

2019/01/29  下午 02:48                15 target.txt
               1 File(s)             15 bytes
               0 Dir(s)  165,675,839,488 bytes free
```

Node.js

- 檔案系統

  - 使用 EventEmitter 捕獲數據

Node.js 的核心模組很多都直接繼承 EventEmitter 物件，因此也都可以直接註冊使用監聽器

```
// filesystem/watcher-spawn-parse.js
const fs = require('fs');
const spawn = require('child_process').spawn;
const filename = process.argv[2];

if (!filename) {
  throw Error('A file to watch must be specified!');
}

fs.watch(filename, () => {
  const ls = spawn('cmd.exe', ['/c','dir', filename]);
  let output = '';
  ls.stdout.on('data', chunk => output += chunk);

  ls.stdout.on('close', () => {
    const parts = output.split("\r\n");
    console.log(parts[5]);
  });
});
```

chunk 是個 Buffer 類型的資料，Buffer 是 Node.js 表示二進制數據的方式,它指向 Node.js 核心分配的一塊 blob 記憶體，Buffers 無法調整大小，它們需要編碼和解碼才能轉換為字符串。每次將非字符串(non-string)加到字符串(string)時，就像這裡將 chunk 加到 output，將會隱式的調用物件的 toString() 方法。對於 Buffer，這意味著會使用預設的編碼（UTF-8）將內容複製到 Node.js 的 heap 堆疊中。

Windows line break

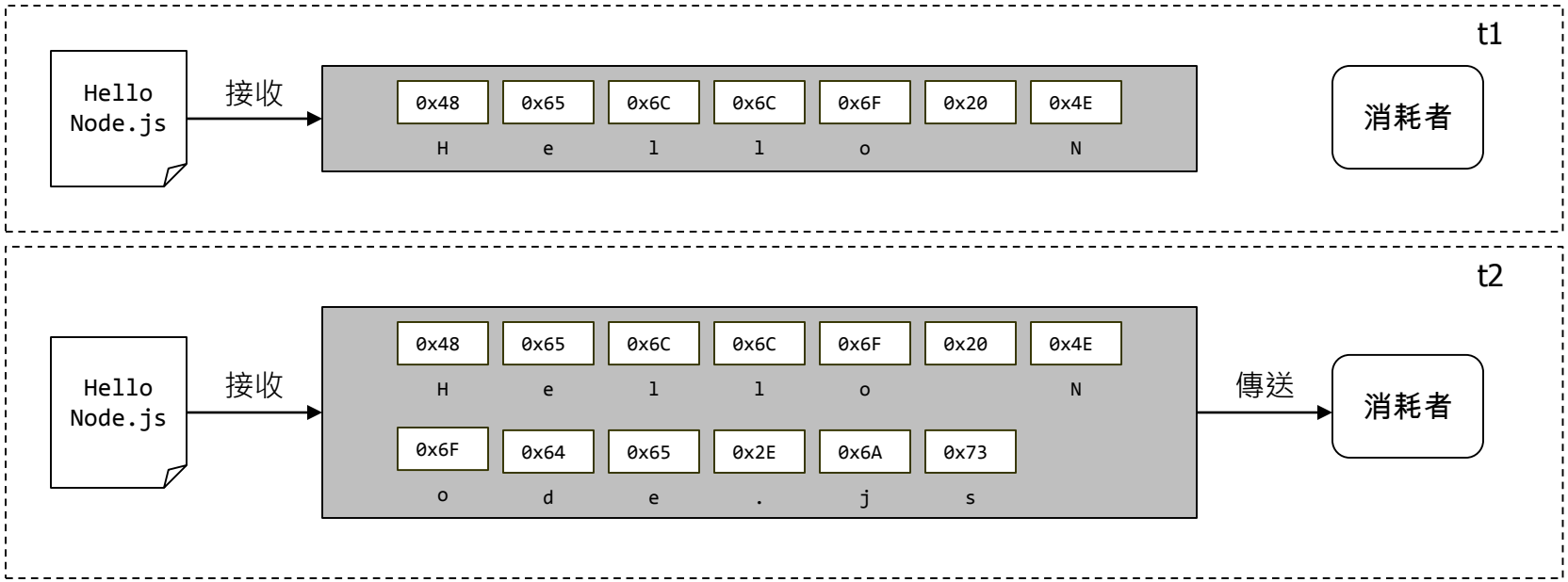
Node.js

- 串流 (Streams)

  串流是 Node.js 最重要的元件與模式之一。社群裡有句格言:「串流一切事物!」，足以說明串流在 Node.js 裡扮演的角色。

  對於 Node.js 這類基於事件的平台，最有效率的 I/O 處理方式就是即時化，也就是盡可能的盡快處裡輸入，並盡快送出應用程式的輸出結果。

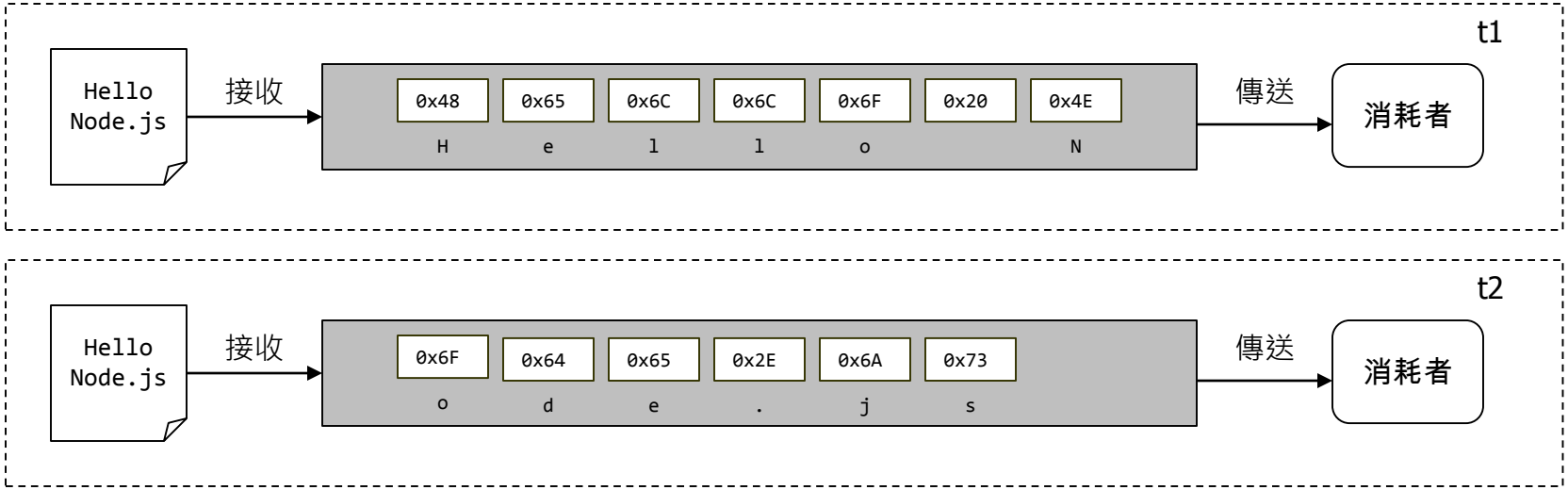
  - 緩衝(Buffering) vs 串流(Streaming)



在上圖中，我們看到時間 t1 時，會從某資源接收部份資料並儲存至緩衝區。而在時間 t2 時，接收到最後一個資料塊，即表示完成讀取操作，然後將整個緩衝區內容傳送給消耗者。

Node.js

- 緩衝(Buffering) vs 串流(Streaming)



串流則允許當一接收到來自某資源的資料，即馬上處裡。上圖顯示如何從某資源接收各個資料塊，並立即提供給消耗者。消耗者可以立即處理資料，而無須等待所有資料皆被收集到緩衝區時才處理。

這兩種處理方式有何不同?我們可以歸結為兩個主要範疇:

- 空間效率
- 時間效率

不過，Node.js 串流還有另一重要優勢:可組合性。

Node.js

- 空間效率

對於無法將完整資料一次載入並處理完畢的情況，我們仍可以透過串流來完成工作。舉例來說，試想必須讀取一個巨量檔案的狀況，可能是數百 MB 甚至是數 GB。這時若使用一個會將檔案完整讀取，並回傳巨量緩衝區的 API，顯然不是個好主意。想像一下，若並行讀取幾個巨量檔案，則應用程式將會很容易耗盡記憶體。此外，事實上 V8 的緩衝區也無法超過 0x3FFFFFFF 個位元組(這只比 1GB 小一點點)。因此，其實我們可能會在還沒耗盡實體記憶體之前就先出錯。

```
RangeError: File size is greater than possible Buffer: 0x3FFFFFFF bytes
```

例如，使用緩衝 API 作 gzip 壓縮

```
const fs = require('fs');
const zlib = require('zlib');
const file = process.argv[2];

fs.readFile(file, (err, buffer) => {
  zlib.gzip(buffer, (err, buffer) => {
    fs.writeFile(file + '.gz', buffer, err => {
      console.log('File successfully compressed');
    });
  });
});
```

使用緩衝 API，如果檔案超過 1GB，這將接收到超過緩衝區最大值的錯誤訊息。

為了簡化範例，此例省略錯誤處理

Node.js

- 空間效率

使用串流作 gzip 壓縮

```
const fs = require('fs');
const zlib = require('zlib');
const file = process.argv[2];

fs.createReadStream(file)
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream(file + '.gz'))
  .on('finish', () => console.log('File successfully compressed'));
```

這裡我們使用串流 API，使它能夠處理巨量檔案。就這樣嗎？，沒錯，是的。串流之所以神奇的原因，還包括它們的介面及可組合性，因而能提供清楚、優雅與簡潔的程式碼。最重要的是無論檔案大小為何，這個程式基本上都會以恆定的記憶體用量順利執行。

```
require('fs').createReadStream(process.argv[2]).pipe(process.stdout);
```

Node.js

- 空間效率

串流並不只是用來處理純 I/O 而已，通常它也可作為簡化及模組化程式碼的絕佳方案。在 Node.js 裡，它們無所不在，包含核心模組。不僅 fs 模組，就連 http 的請求及回應物件也都是串流。

Node.js 裡的每個串流，都來自 stream 核心模組的基本抽象類別實作，共有四種 stream 類別，而每個類別也都是 EventEmitter 的實例。事實上，串流可以產生好幾種事件類型，例如於 stream.Readable 串流完成讀取時的 end，或於出錯時的 error。

串流之所以如此靈活的原因之一，在於它們不只能處理二進位資料，而是幾乎能處理所有的 JavaScript 資料值，事實上它們支援兩種操作模式:

- 二進位模式:此模式資料以資料塊的形式進行串流，例如緩衝區(buffers)或字串
- 物件模式:此模式是串流資料會被視為物件序列(幾乎允許使用任何的 JavaScript 值)

```
require('fs').createReadStream(process.argv[2]).pipe(process.stdout);
```

```
require('fs').createReadStream(process.argv[2])
  .on('data', chunk => process.stdout.write(chunk))
  .on('error', err => process.stderr.write(`ERROR: ${err.message}\n`));
```

Node.js

- 時間效率

現在來思考一項應用程式案例，這個應用程式需要先壓縮某個檔案再將它上傳至遠端 HTTP 伺服器，接著解壓縮後再儲存到檔案系統中。若客戶端使用緩衝 API 來實作，則上傳動作會等到整個檔案皆已讀取並壓縮後才開始。另一方面，伺服器也會在所有資料都接收到時才會開始解壓縮。針對這項工作，更好的解決方案是使用串流。在客戶端機器上，串流允許你一從檔案系統讀取，就壓縮並傳送資料塊；就伺服器端而言，則可以一從遠端節點接收，就解壓縮每一塊資料。

```
const http = require('http');
const fs = require('fs');
const zlib = require('zlib');

const server = http.createServer((req, res) => {
  const filename = req.headers.filename;
  console.log('File request received: ' + filename);
  req
    .pipe(zlib.createGunzip())
    .pipe(fs.createWriteStream(filename))
    .on('finish', () => {
      res.writeHead(201, {'Content-Type': 'text/plain'});
      res.end("That's it\n");
      console.log(`File saved: ${filename}`);
    });
});

server.listen(3000, () => console.log('Listening'));
```

HTTP Server

伺服器接收來自遠端網路的資料塊，借助 Node.js 串流，在一接收到資料就進行解壓縮並儲存

Node.js

- 時間效率

```
const fs = require('fs');
const zlib = require('zlib');
const http = require('http');
const path = require('path');
const file = process.argv[2];
const server = process.argv[3];
const options = {
  hostname: server,
  port: 3000,
  path: '/',
  method: 'PUT',
  headers: {
    filename: path.basename(file),
    'Content-Type': 'application/octet-stream',
    'Content-Encoding': 'gzip'
  }
};
const req = http.request(options, res => {
  console.log(`Server response: ${res.statusCode}`);
});
fs.createReadStream(file)
  .pipe(zlib.createGzip())
  .pipe(req)
  .on('finish', () => {
    console.log('File successfull sent');
});
```

HTTP Client

使用串流從檔案讀取資料，接著當它從檔案系統讀取，就壓縮並傳送各個資料塊。

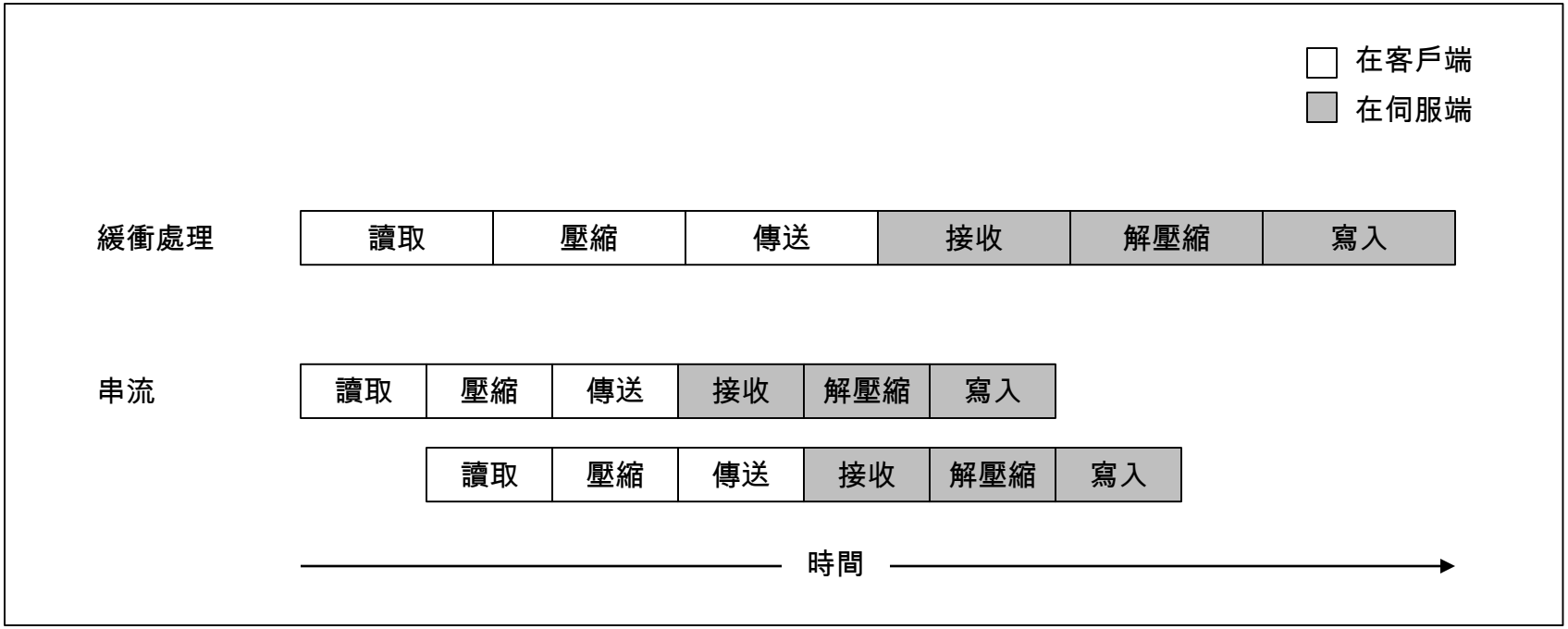
如何啟動:

```
node gzipSend <path to file> localhost
```

Node.js

- 時間效率

若選擇較大的檔案，就更容易感受到這種方式的優勢。但究竟為何資料流在這種範式下，比使用緩衝 API 更有效率?藉由下圖應該能得到一點提示：



這種狀況下，串流是完美的運作方式，唯一的注意事項是必須保持每塊資料在各個階段的抵達順序，而這 Node.js 串流會幫我們留意這部份。

Node.js

- 可組合性

目前為止我們看到的程式碼，是提供關於串流組合方式的概觀，拜 pipe() 方法之賜，讓我們能夠連接不同的處理單元，並讓每個單元負責單一功能，以實現完美的 Node.js 風格。之所以能這樣作是因為串流有統一的介面，而它們的 API 是彼此可通的。唯一的先決條件是管道的下一個串流必須支援上一個串流所產生的資料型態，它可以是二進位、文字，甚至物件。

為了示範這項特質，我們可以試著在之前的應用程式裡，加上加密層級。

HTTP Server

```
const crypto = require('crypto');
[...]
const server = http.createServer((req, res) => {
  [...]
  req
    .pipe(crypto.createDecipher('aes192', 'a_shared_secret'))
    .pipe(zlib.createGunzip())
    .pipe(fs.createWriteStream(filename))
    .on('finish', () => {
      [...]
    });
});
```

Node.js

- 可組合性

HTTP Client

```
const crypto = require('crypto');
[...]
fs.createReadStream(file)
  .pipe(zlib.createGzip())
  .pipe(crypto.createCipher('aes192', 'a_shared_secret'))
  .pipe(req)
  .on('finish', () => {
    console.log('File successfull sent');
  });
```

毫不費力地(就幾行程式碼而已)，就能夠為應用程式加入加密層級。所需的不過是重複使用現有的轉換串流，將它納入到既有的管道裡。按同樣方式，便能夠加入並結合其他串流，就像玩樂高積木一樣。

很明顯的，此種處理方式的主要優勢在於可重複使用性。不過就前述所呈現的程式碼來看，串流也能讓程式碼更清晰及更模組化。基於這些原因，串流並不只是用來處理 I/O 而已，通常它也可作為簡化及模組化程式碼的絕佳方案。

Node.js

- 非同步的批次處理及快取

在高負載的應用程式中，快取經常扮演著關鍵性的角色，而且幾乎在網路上四處可見。從網頁、圖片與樣式表等靜態資源，以及資料庫查詢結果這類純粹的資料，都經常會涉及到快取的使用。因此學習如何將快取應用於非同步操作中，以及如何從高請求處理量中獲益。

- 實作一個沒有快取及批次處理的伺服器

我們要先實作一個小型的測試伺服器，作為參考指標，藉此衡量各種不同的實作技巧所帶來的影響。

試想公司一個管理銷售狀況的網頁伺服器，具體來說是可以用於查詢特定商品類型的所有交易總額。對此，我們將使用 LevelUP 資料庫來加以實作。

LevelUP 是 Google LevelDB 的 Node.js 包裝，LevelDB 起初是為了在 Chrome 瀏覽器裡實作 IndexedDB 的鍵值存儲，但現在可不僅僅是這樣。由於它的極簡與可擴充性，LevelDB 已經被喻為「資料庫的 Node.js」。一如 Node.js，LevelDB 也提供極快速的執行效能，而且僅配置了最基本的功能集合，讓開發者可以在其上建置出任何型態的資料庫。

Node.js

- 非同步的批次處理及快取

LevelUP 雖然起初是 LevelDB 的包裝，但後續也支援了數種後端，包含記憶體式儲存、Riak 與 Redis 這類其它的 NoSQL 資料庫，以及 IndexedDB 與 localStorage 這類的網路儲存引擎。這讓我們能夠在伺服器及客戶端接使用相同的 API，以實現一些開發方案。

目前圍繞在 LevelUP 周邊的附加元件及模組，已發展出相當成熟的生態體系，使它從微小的核心一路擴展出各式功能，例如資料庫複製、次索引、線上更新、查尋引擎等等。此外，也有基於 LevelUP 的完整資料庫，例如 CouchDB 的移植版 PouchDB 與 CouchUP，甚至還有一款圖形資料庫 levelgraph，能夠在 Node.js 及瀏覽器上運作。

用 LevelUP 來實作我們的範例既簡單又富彈性。我們所採用的資料模型，是儲存在 sales 子集(sublevel，資料庫的一個區段)裡，一個簡單的交易清單。其組織方式如下格式:

```
transactionId → {amount, item}
```

transactionId 為鍵，其值則是一個 JSON 物件，內容為銷售總額(amount)與 item 類型。

Node.js

- 非同步的批次處理及快取

資料的處理過程相當簡單，實作的 API totalSales.js 如下:

```
const level = require('level');
const sublevel = require('level-sublevel');
const db = sublevel(level('example-db'), {valueEncoding: 'json'});
const salesDb = db.sublevel('sales');

module.exports = function totalSales(item, callback) {
  let sum = 0;
  salesDb.createValueStream()
    .on('data', function(data) {
      if(!item || data.item === item) {
        sum += data.amount;
      }
    })
    .on('end', function(){
      callback(null, sum);
    });
};
```

Node.js

- 非同步的批次處理及快取

totalSales 函式是模組的核心，也是唯一匯出的 API，其運作如下:

1. 首先從內含銷售交易的 salesDb 子集建立一個串流，該串流會從資料庫取得所有帳目。
2. data 事件從資料庫串流接收每一筆銷售資料。若 item 類型與呼叫函式傳入的類型相符，便將當前帳目的 amount 值加入至總和的 sum 值。若未傳入任何類型，則所有 item 類型都會被計入總和。
3. 最後，當收到 end 事件時，便呼叫 callback() 方法，並提供最終的 sum 作為結果。

這裡所建置的簡單查詢在執行效能絕對不是最好的。在真實世界的應用程式裡，通常會使用索引來協助查詢，或是在更好的情況下，能夠使用遞增的映射/歸納(incremental map/reduce)來即時的計算總和。不過，我們的範例裡，其實緩慢的查詢更加有用，因為可以突顯出經最佳化後的差異。

Node.js

- 非同步的批次處理及快取

再來，我們要使用 HTTP 伺服器來揭露 totalSales API，所以建置一個 app.js:

```
const http = require('http');
const url = require('url');
const totalSales = require('./totalSales');

http.createServer((req, res) => {
  const query = url.parse(req.url, true).query;
  totalSales(query.item, (err, sum) => {
    res.writeHead(200);
    res.end(`Total sales for item ${query.item} is ${sum}`);
  });
}).listen(8000, () => console.log('Started.'));
```

這個伺服器相當簡易，僅僅揭露了一個 totalSales API。

在首次啟動伺服器之前，需要先放置一些樣本資料在資料庫裡。

Node.js

- 非同步的批次處理及快取

在首次啟動伺服器之前，需要先放置一些樣本資料在資料庫裡，這裡是 populate_db.js。

```
const sublevel = require('level-sublevel');
const level = require('level');
const uuid = require('node-uuid');

const db = sublevel(level('example-db', {valueEncoding: 'json'}));
const salesDb = db.sublevel('sales');
const items = ['book', 'game', 'app', 'song', 'movie'];

for (let i = 100000; i > 0; i--) {
  salesDb.put(uuid.v4(), {
    amount: Math.ceil(Math.random() * 100),
    item: items[Math.floor(Math.random() * 5)]
  });
}
console.log('DB populated');
```

Node.js

- 非同步的批次處理及快取

現在，啟動伺服器所需的一切皆已準備就緒。一如往常，執行如下命令啟動伺服器：

```
node app
```

對伺服器進行查詢，只需使用瀏覽器前往以下 URL：

```
http://localhost:8000/?item=book
```

然而，為了明辨這個伺服器的執行效能，我們需要傳送多個請求。所以我們利用一個 loadTest.js 的小腳本，以 200 毫秒的間隔傳送請求。

```
const request = require('request');
const http = require('http');

const start = Date.now();
let count = 20;
const interval = 200;
let completed = count;
```

Node.js

- 非同步的批次處理及快取

```
const agent = new http.Agent();
agent.maxSockets = count;
const items = ['book', 'game', 'app', 'song', 'movie'];

const id = setInterval(() => {
  let query = 'item=' + items[Math.floor(Math.random() * 5)];
  request({
    url: 'http://localhost:8000?' + query,
    pool: agent
  },
  (err, res) => {
    if (err) return console.log(err);
    console.log(res.statusCode, res.body);
    if (!--completed) {
      console.log(`All completed in: ${Date.now() - start}ms`);
    }
  });
  if (!--count) {
    clearInterval(id);
  }
}, interval);
```

Node.js

- 非同步的批次處理及快取

現在執行如下命令：

```
node loadTest
```

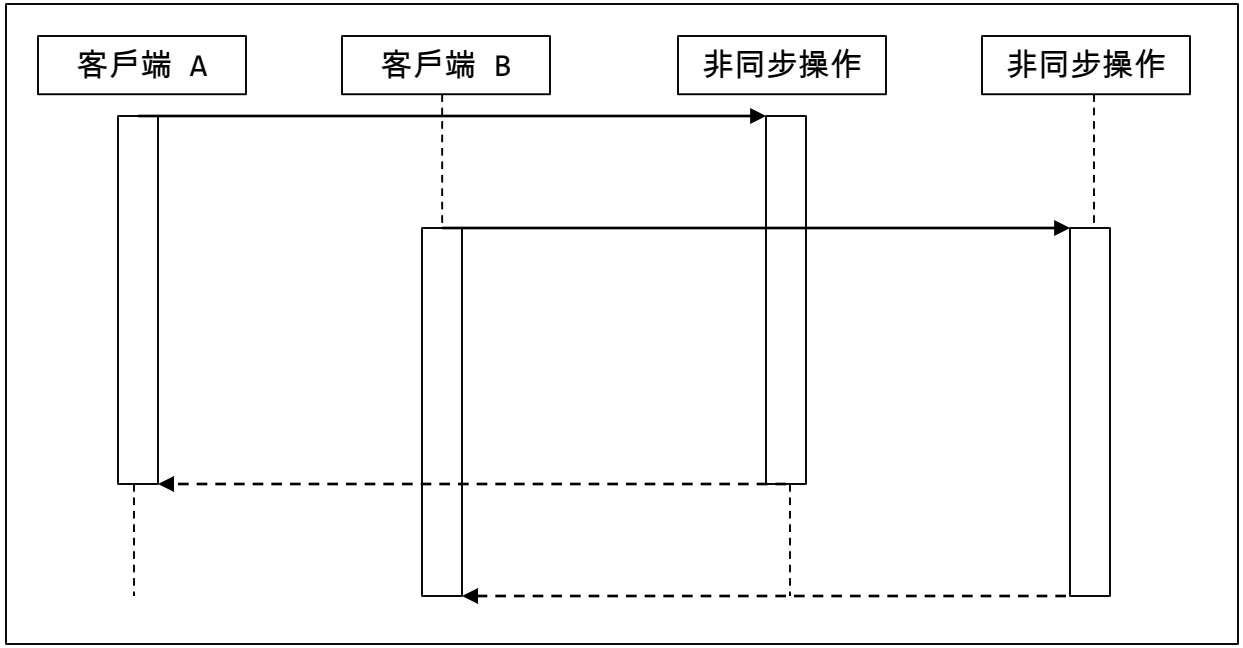
請留意這項測試的總執行時間，目前這 20 個請求會花上一點時間才完成。接下來我們要對它作最佳化，看看可以節省多少時間。

Node.js

- 非同步的批次處理及快取

  - 非同步請求的批次處理

在處理非同步操作時，最基本的快取實現機制，是對相同 API 的多次呼叫進行**批次處理**。其構想很簡單：當我們在呼叫一個非同步函式時，若有先前的呼叫仍在等待其結果，則我們可以將回呼掛接到正在執行的操作，而無須建立全新的請求。來看看以下的圖示：

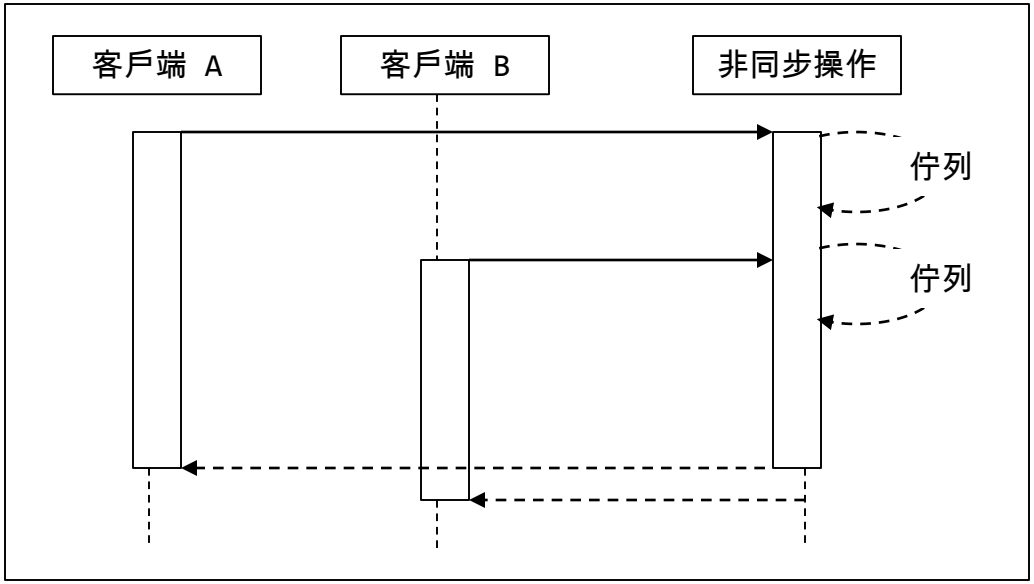


Node.js

- 非同步的批次處理及快取

  - 非同步請求的批次處理

前面的圖顯示有兩個客戶端(可以是兩個不同的物件，或兩個不同的網頁請求)皆以相同的參數、呼叫相同的非同步操作。可以想見，在一般情況下，這兩個客戶端會起始個別的操作，並且在不同的時間點完成，如前圖所示。然而，試想另一種情景，請見以下示意圖:



第二張圖顯示，這兩個以相同參數呼叫相同 API 的請求，能夠被批次處理。換句話說即是附加到相同的執行中操作，當操作完成時，兩個客戶端都會被通知。這是一種簡單、但極為強大的作法，能夠最佳化應用程式負載，卻不需要涉及記憶體管理及作廢策略等複雜的快取機制。

Node.js

- 非同步的批次處理及快取

  - 對網頁伺服器的請求做批次處理

  現在我們要在 totalSalse API 之上新增一個批次處理層。所使用的模式非常簡單:若 API 被呼叫時，已有其它完全相同的處理中請求，便會將回呼加入佇列。當非同步操作完成時，就會立即呼叫佇列裡的所有回呼。

  以下就來看看此模式是如何具體轉換成程式碼的。讓我們建立一個名為 totalSalesBatch.js 的新模組，在原始的 totalSales API 之上，實作一個批次處理層:

Node.js

- 非同步的批次處理及快取
  - 對網頁伺服器的請求做批次處理

```
const totalSales = require('./totalSales');
const queues = {};

module.exports = function totalSalesBatch(item, callback) {
  if (queues[item]) {
    console.log('Batching operation');
    return queues[item].push(callback);
  }

  queues[item] = [callback];
  totalSales(item, (err, res) => {
    const queue = queues[item];
    queues[item] = null;
    queue.forEach(cb => {
      cb(err, res);
    });
  });
};
```

Node.js

- 非同步的批次處理及快取

  - 對網頁伺服器的請求做批次處理

  totalSalesBatch() 函式是原始 totalSales() API 的代理器，其運作方式如下:

  1. 若佇列裡已經存在相同的 item 類型，表示針對此 item 的請求已經執行中。對此，我們只需要將 callback 加入到佇列中，並立即返回。
  2. 若該項目無相符的佇列，即表示必須建立新請求。對此，我們為特定的 item 建立一個新佇列，並加入 callback 函式。接著，便呼叫原始的 totalSales() API。
  3. 當原始的 totalSales() 完成請求後，便迭代該 item 的佇列，呼叫當中的每一個回呼，並帶入操作結果。

  totalSalesBatch() 函式的外在行為，乍看之下如同於原始的 totalSales() API。然而，現在若以相同的參數多次呼叫 API，將會批次處理，因而節省時間及資源。

Node.js

- 非同步的批次處理及快取

  - 對網頁伺服器的請求做批次處理

  現在讓我們在 app.js 裡，將 HTTP 伺服器所使用的 totalSales 模組替換為方才所做的批次化版本：

```
[...]
//const totalSales = require('./totalSales');
const totalSales = require('./totalSalesBatch');
[...]
```

現在再次啟動伺服器，並執行負載測試。我們首先會看到請求被批次回傳，印證了此模式確實可運作。接著，我們應該會察覺到測試的總執行時間大幅降低，一般來說應該會比原先的測試還要快上數倍！

這可是相當驚人的成果，證明只要加上一個簡單的批次處理層，便能夠大幅提升執行效能。無須耗費精力來實現完善但卻是複雜的快取機制。並且也無須費心於作廢策略的擬定。

批次處理請求的模式，會在高負載/慢 API 的應用程式裡發揮它的最大潛能。因為這類環境能夠在短時間內累積大量的待處理請求，以供批次處理。

Node.js

- 非同步的批次處理及快取

  - 非同步請求的快取處理

批次處理請求模式的一項問題，是 API 反應越快，則可批次化的請求量就越少。或許有人會認為，如果 API 已經很快，就無須再做最佳化。然而，再快速的 API 都還是會在某一時間點佔用資源，它仍然會是整體效能考量的一項重要因子。此外，事實上我們有時可以放心的假設某個 API 的呼叫結果並不會經常性變動，因此請求批次處理就不會是唯一的最佳化選項。基於上述考量，我們可以採行更積極的快取模式，來降低應用程式負載並提升其反應能力。

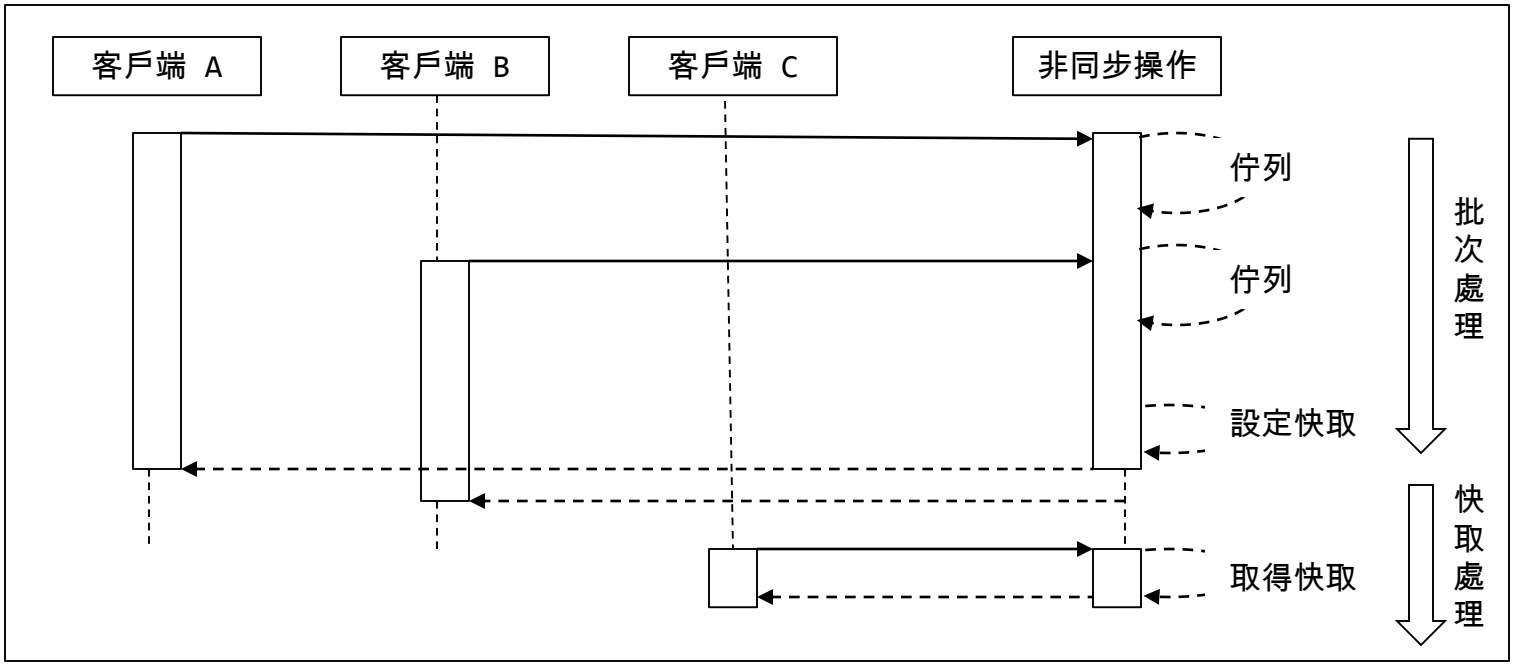
其構想也很簡單:當請求完成後，便將結果儲存在快取裡。快取的儲存形式可以是變數、資料庫項目，或者是特殊的快取伺服器。如此一來，當 API 再次被呼叫時，便能夠立即從快取內取得結果，而不是再增生另一個請求。

快取的概念對於有經驗的開發者並不是甚麼新鮮事物，但在非同步程式設計中，此模式有一項值得留意的重點，它是應該與批次處理結合，才是最佳的實作方式。因為在快取尚未建立前，即有可能會並行執行多個請求，而當這些請求完成時，快取可能會被重複建立多次。若加入批次處理便能防範此類狀況。

Node.js

- 非同步的批次處理及快取
  - 非同步請求的快取處理

基於以上的設想，非同步請求快取模式的最終結構如下圖所示：



Node.js

- 非同步的批次處理及快取

  - 非同步請求的快取處理

  上圖呈現了一個最佳化的非同步快取演算法，共有兩個階段：

  - 第一個階段完全相同於批次處理模式。當快取未設定時，所接收的請求都會被一併批次處理。當請求完成後，就會設定快取。
  - 當快取設定後，所有後續的請求皆由快取來處理。

  另一個需要考量的重要問題，是留意「釋放魔鬼」般的反模式。由於我們處理的是非同步 API，所以必須確認一定是以非同步方式回傳快取值，即便快取的存取只涉及同步操作。

Node.js

- 非同步的批次處理及快取
  - 對網頁伺服器的請求做快取處理

為了說明及量測非同步快取模式的優勢，現在我們要將先前所學的技巧應用在 totalSales() API 上。如同於對請求做批次處理的範例，我們必須建立一個代理器，為原始的 API 新增一個快取層。建立一個名為 totalSalesCache.js 的新模組:

```
const totalSales = require('./totalSales');

const queues = {};
const cache = {};

module.exports = function totalSalesCache(item, callback) {
  const cached = cache[item];
  if(cached) {
    console.log('Cache hit');
    return process.nextTick(callback.bind(null, null, cached));
  }
```

Node.js

- 非同步的批次處理及快取
  - 對網頁伺服器的請求做快取處理

```
  if(queues[item]) {
    console.log('Batching operation');
    return queues[item].push(callback);
  }

  queues[item] = [callback];
  totalSales(item, function(err, res) {
    if(!err) {
      cache[item] = res;
      setTimeout(() => {
        delete cache[item];
      }, 30 * 1000); //30秒後逾期
    }

    const queue = queues[item];
    queues[item] = null;
    queue.forEach(cb => cb(err, res));
  });
};
```

Node.js

- 非同步的批次處理及快取

  - 對網頁伺服器的請求做快取處理

  你應該立刻就能發現，上述程式碼有諸多部份，相同於我們先前的非同步批次處理版本。事實上，僅有下列差異:

  1. 當 API 被呼叫時，首先便需要檢查快取是否已設定。若是，即使用 callback() 回傳快取值，並透過 process.nextTick() 加以延遲。
  2. 以批次處理方式持續執行，不過這次當原始 API 順利完成後，是將結果存入快取。並且設定逾時，讓快取於 30 秒後失效。這可是相當簡單卻又極為高效的技巧!

  為了測試方才所建立的 totalSales 包裹程式，更新 app.js 模組如下:

```
[...]
//const totalSales = require('./totalSales');
//const totalSales = require('./totalSalesBatch');
const totalSales = require('./totalSalesCache');
[...]
```

Node.js

- 非同步的批次處理及快取

  - 對網頁伺服器的請求做快取處理

  現在可以再次啟動伺服器，並使用先前的 loadTest.js 腳本來加以測試。在預設的測試變數下，應該可以看到，相較於單純的批次處理，應該會有較少的執行時間。當然，這些變幅取決於諸多因素，例如所接收到的請求量，以及各個請求間的間隔等等。若請求量較高，且持續較長一段時間時，快取處理凌駕於批次處理的優勢就會更加顯著。

  將函式呼叫的結果存入於快取中，這種實作又稱為備忘(memorization)。在 npm 裡，能夠找到許多可輕易的實作非同步備忘的套件，其中一種相當完備的套件就名為 memoizee。

  - 實作快取機制的一些注意事項

  1. 當快取值數目龐大時，便很容易會消耗大量記憶體。對此，可以應用「最近少被使用」(Least Recently Used: LRU)演算法，保持恆定的記憶體使用量。
  2. 當應用程式被分拆為多個程序時，以簡單的變數來設定快取，可能會導致各個伺服器實例皆設定並使用各自的快取。若此種狀況不符需求，則解決方案就是以共享的方式儲存快取。常見的方案有 Redis 及 Memcached。
  3. 相較於自動逾期機制，手動式的快取作廢能夠視需要提供存活期較長的快取，或是提供更近期的資料，不過此舉勢必也會增加管理上的複雜度。

Node.js

- 非同步的批次處理及快取
  - 對網頁伺服器的請求做快取處理(使用 Redis)

Redis 是一個速度非常快的非關連式**記憶體資料庫**，它可以儲存**鍵(key)**與 7 種不同類型的**值(value)**之間的映射(mapping)，可以將存儲在記憶體的鍵值對數據持久化(persistence)到硬碟，可以使用複製特性來擴展讀取的性能，還可以使用客戶端分片(client-side sharding)來擴展寫入的特性，用戶可以很方便的將 Redis 擴展成一個能包函數百 GB 數據、每秒處理上百萬次請求的系統。

隨著 Redis 的高性能、低延遲系統的發展， Redis 的使用變得越來越廣泛。自 2017 年起，Redis 就擠進了 DB-Engine 排名榜的前 10 名。當你需要以接近實時的速度訪問快速變動的數據流時，Redis 這樣的鍵值數據庫就是最佳的選擇。所以，Redis 已經成為軟體開發維運人員所必須具備的「標準」技術。

Redis 是一個開源（BSD 許可）的軟體，除可當作資料庫(database)外，還可作為緩存(cache)和消息代理(message broker)。

Node.js

- 非同步的批次處理及快取
  - 對網頁伺服器的請求做快取處理(使用 Redis)

首先我們先建立一個 Redis 應用層幫我們處理 cache 事務，redisCache.js

```
exports.set = function set(conn, item, value, callback) {
  conn.set("demo:sales:cache:" + item, value, "EX", 30, (err, result) => {
    if (err) return callback(err);
    return callback(null, result);
  });
}

exports.get = function get(conn, item, callback) {
  conn.get("demo:sales:cache:" + item, (err, result) => {
    if (err) return callback(err);
    return callback(null, result);
  });
}
```

Node.js

- 非同步的批次處理及快取
  - 對網頁伺服器的請求做快取處理(使用 Redis)

再來，我們要將原先使用變數當 cache 的功能用 Redis 置換掉，這樣不僅將減少 Node.js 的記憶體使用，也可跨 Node.js 實例共用這個緩存，以下是 totalSalesRedisCache.js:

```
const redis = require("redis");
const cache = require("./redisCache");
const totalSales = require("./totalSales");
const client = redis.createClient({
  host: "10.11.25.137",
  port: 6379,
  password: "iscat",
  db: 0
});

const queues = {};
```

Node.js

- 非同步的批次處理及快取
  - 對網頁伺服器的請求做快取處理(使用 Redis)

```
module.exports = function totalSalesCache(item, callback) {
  cache.get(client, item, (err, cached) => {
    if (cached) {
      console.log(`Cache hit: ${item}:${cached}`);
      return callback(null, cached);
    }

    if (queues[item]) {
      console.log(`Batching operation: ${item}`);
      return queues[item].push(callback);
    }
```

Node.js

- 非同步的批次處理及快取
  - 對網頁伺服器的請求做快取處理(使用 Redis)

```
    queues[item] = [callback];
    totalSales(item, function(err, res) {
      if (!err) {
        cache.set(client, item, res, (err, result) => {
          if (err) console.log(`Redis cache error ${err}`);
        });
      }

      const queue = queues[item];
      queues[item] = null;
      queue.forEach(cb => cb(err, res));
    });
  });
};
```

Node.js

- 非同步的批次處理及快取
  - 對網頁伺服器的請求做快取處理(使用 Redis)

最後更新 app.js

```
[...]
//const totalSales = require('./totalSales');
//const totalSales = require('./totalSalesBatch');
//const totalSales = require('./totalSalesCache');
const totalSales = require('./totalSalesRedisCache');
[...]
```

現在不僅不會用到 Node.js 實例的記憶體當緩存，而且如果啟動另外一個 app 伺服器，將會使用到相同的 Redis 緩存，Node.js 的擴展性又更上一層。

這只是 Redis 最簡單的應用，後面我們將還會看到它的另外一種功能，訊息中介者(Message broker)。

Node.js

- 非同步的批次處理及快取

  - 使用承諾的批次及快取處理

在非同步控制流程模式裡，承諾(Promise)能夠大幅簡化非同步程式碼，不過在處理批次及快取處理時，它也能夠提供不錯的助益。若回想一下，承諾有兩項特性能夠讓我們加以應用：

- 多個 then() 監聽器可附加至相同的承諾。
- then() 監聽器保證只會被呼叫一次，並且即便是在承諾已解析(resolved)後才附加，也依然可以運作。不僅如此，then() 也保證一定以非同步的方式被呼叫。

前述的第一項特性便是在批次處理請求時所需要的，而第二項特性即表示承諾已經是已解析值的快取，也就能夠自然以一致的非同步方式，回傳快取值。基於以上設想，這就表示以承諾來實現批次及快取處理，是相當簡單明瞭的。

對此，我們可以試著利用承諾，建立 totalSales() API 的包裹程式，從中檢視它對於批次及快取層的實作細節。讓我們建立一個名為 totalSalesPromise.js 的新模組：

Node.js

- 非同步的批次處理及快取
  - 使用承諾的批次及快取處理

對此，我們可以試著利用承諾，建立 totalSales() API 的包裹程式，從中檢視它對於批次及快取層的實作細節。讓我們建立一個名為 totalSalesPromise.js 的新模組:

```
const { promisify} = require('util');
const totalSales = promisify(require('./totalSales'));

const cache = {};

module.exports = function totalSalesPromises(item) {
  if(cache[item]) {
    return cache[item];
  }
```

Node.js

- 非同步的批次處理及快取
  - 使用承諾的批次及快取處理

```
  cache[item] = totalSales(item)
    .then(res => {
      setTimeout(() => {delete cache[item]}, 30 * 1000);
      return res;
    })
    .catch(err => {
      delete cache[item];
      throw err;
    });
  return cache[item];
};
```

看著上述程式碼，首先映入眼前的就是此種方案的簡潔及優雅。承諾是極佳的工具，針對我們的需求而言，它提供了極其便利的優點。以下來看看上述程式碼發生了甚麼事：

Node.js

- 非同步的批次處理及快取

  - 使用承諾的批次及快取處理

1. 首先，我們需要使用 Node.js 8 所提供的工具函式 util.promisify()(如果使用的是較舊的 Node.js 版本，則可使用 npm 的一個套件 pify)，這個 promisify() 涵式可以承諾化原始的 totalSales() 函式。完成後，totalSales() 便會回傳承諾，而不是接收回呼。
2. 當 totalSalesPromises() 包裹程式被呼叫時，檢查指定的 item 類型是否已存在快取承諾，若有便回傳給呼叫者，要記得，**回傳的是承諾**。
3. 若指定的 item 類型不存在於快取承諾中，便呼叫已承諾化的原始 totalSales() API。
4. 當承諾解析時，設定快取清除時間為30秒，並回傳 res，將操作結果傳遞給附加在承諾上的其它 then() 監聽器。
5. 若承諾以錯誤回絕，便立即重設快取並將錯誤傳遞至承諾鏈，讓附加在相同承諾上的其他監聽器也會接收到錯誤。
6. 最後，回傳所產生的快取承諾。

以上程式碼相當簡單且直覺，而且**同時實現了批次及快取處理**。

Node.js

- 非同步的批次處理及快取
  - 使用承諾的批次及快取處理

為了試驗這個 totalSalesPromises() 函式，需要對 app.js 模組另做一個版本，因為該 API 是使用承諾而不是回呼。建立名為 appPromises.js 的 app 模組:

```
const http = require('http');
const url = require('url');
const totalSales = require('./totalSalesPromises');

http.createServer(function(req, res) {
  const query = url.parse(req.url, true).query;

  totalSales(query.item).then(function(sum) {
    res.writeHead(200);
    res.end(`Total sales for item ${query.item} is ${sum}`);
  });
}).listen(8000, () => console.log('Started.'));
```

Node.js

- 非同步的批次處理及快取

  - 使用承諾的批次及快取處理

  這個實作基本上與原始的 app 模組一致，差異則在於這是基於承諾的版本，因此呼叫方式也稍有不同。

  完成了!現在我們準備要試試這個新版的伺服器，執行如下命令:

```
node appPromises
```

藉由 loadTest 腳本，我們能夠確認新實作是否如預期般運作。執行時間應與使用 totalSalesCache() API 的版本相同。

Node.js

- 複製與負載平衡

傳統的多執行緒網頁伺服器通常只會在機器資源無法再行升級，或升級成本比起用另一部機器還高時，才會進行擴展。藉由多重執行緒，傳統的網頁伺服器可以利用伺服器的完整動力，使用所有可用的處理器與記憶體。然而，這便是單一的 Node.js 程序所無法做到的，因為它是單一執行緒，並且還有著 1GB 的記憶體限制(在 64 位元的機器上，可以增加到最大值 1.7GB)，因此，即便是在單一的機器環境下，Node.js 應用程式也比傳統的網頁伺服器更早面臨到是否需要擴展的議題，為的是完整利用現有的資源。

可別以為這是一種缺點，正好相反。由於條件限制而被迫進行擴充，反而有利於應用程式的某些運作機制，例如可用性與容錯性。以複製方式來擴展 Node.js 應用程式相對簡單而且常見，不見得是為了取得更多資源，也可能僅僅是為了增加富餘(redundant)，實現可容錯性的建置。

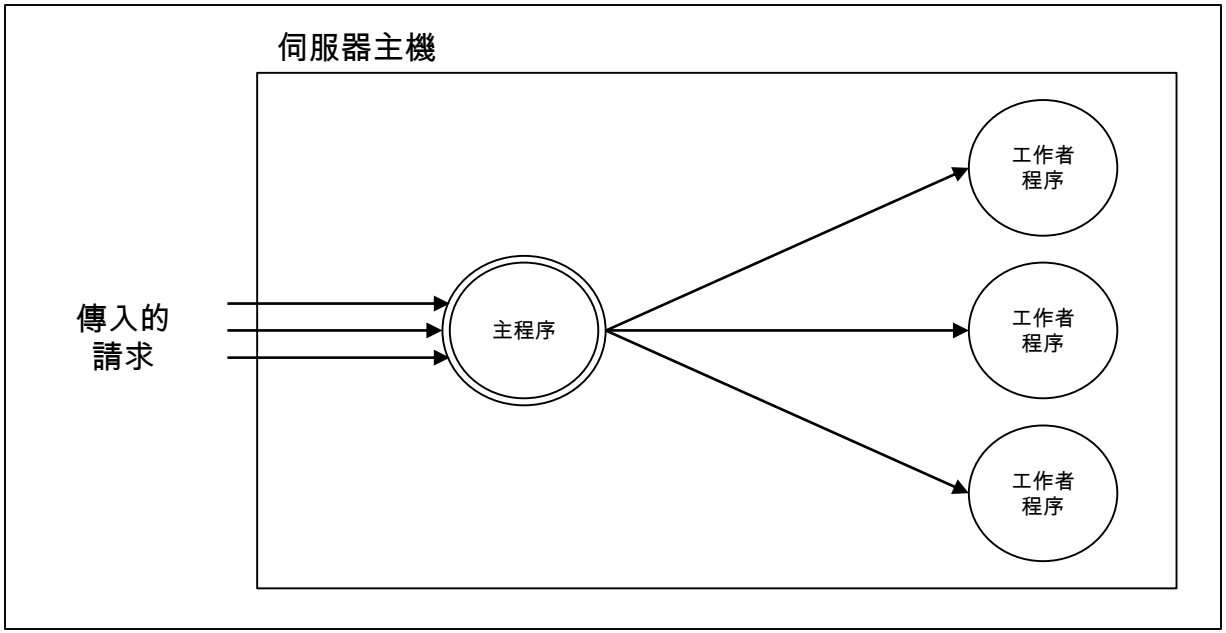
這也可以促使開發者在應用程式的早期階段就開始重視擴展性，確保應用程式不會依賴於任何無法在多個程序或多部機器之間共享的資源。也就是說，擴展應用程式的一項必要條件，就是各個實例不會將共用資訊儲存在無法共享的資源上，例如單機的記憶體或磁碟。舉例來說，在網頁伺服器中，我們不應儲存工作階段(session)資料於記憶體或磁碟中，因為此舉會在試圖擴展時遇到問題。相反的，若使用一個共享的資料庫則能夠確保，無論各個實例部署在何處，都能夠存取到相同的工作階段資訊。

在 Node.js 裡，垂直擴展(增加更多資源給單一機器)與水平擴展(增加更多機器)幾乎是相同的概念，因為事實上這兩種概念在 Node.js 裡的實作技巧相當類似。

Node.js

- 複製與負載平衡
  - cluster 模組

在 Node.js 模組裡，讓應用程式可在單一機器中透過多個實例來分散負載的最簡單模式，就是使用核心函式庫的 cluster 模組。cluster 模組能夠輕易地實現應用程式的分支化，並能夠自動地將傳入的連線轉遞給多個實例:



主程序(master process)負責生程多個工作者程序(worker process)，工作者程序就是應用程式的各個實例。而每個傳入的連線，將會分派在這些複製的工作者中，藉此分散負載。

Node.js

- 複製與負載平衡
  - cluster 模組

在 Node.js 裡， cluster 模組會讓多個工作者共享相同的伺服器 socket，而實際在各工作者之間實現負載平衡的則是作業系統。然而，這會有個問題，因為作業系統所採行的負載平衡演算法，並不會讓網路請求平均分散，而是對程序的執行進行排程。因此，實際的負載並不一定會平均分散在所有的實例上，通常是一部份的工作者會接收到大部份的負載。這種型態的行為模式對作業系統排程而言是合理的，因為它的重點就是最小化不同程序間的繁雜切換。

不過，Node.js 自版本 0.11.2 起，便導入了一個循環式負載平衡演算法(round robin load-balancing algorithm)至主程序中，使請求可在工作者之間平均分配。這個新的負載平衡演算法在除了 Windows 以外的所有平台上皆為預設啟用，其設定值可以透過 cluster.schedulingPolicy 進行全域性的變更，例如設定為 cluster.SCHED_RR(循環式)或 cluster.SCHED_NONE(由作業系統處理)。

Node.js

- 複製與負載平衡
  - cluster 模組

現在讓我們建置一個小型的 HTTP 伺服器，並使用 cluster 模組作複製及負載平衡。建立一個名為 app.js 的檔案，內含如下的程式碼：

```
const http = require('http');
const pid = process.pid;

http.createServer((req, res) => {
  for (let i = 1e7; i > 0; i--) {}
  console.log(`Handling request from ${pid}`);
  res.end(`Hello from ${pid}\n`);
}).listen(8080, () => {
  console.log(`Started ${pid}`);
});
```

這個 HTTP 伺服器會對任何傳入的請求，回應一個內含伺服器 PID 的訊息。藉此便能夠辨識出，回應請求的是哪一個應用程式實例。此外，為了模擬出更接近於真實狀況的 CPU 運作，我們執行一個空迴圈一千萬次。若沒有這項動作，則我們的測試規模會小到幾無伺服器負載可言。

這個即將被擴展的 app 模組可以是任何東西，例如可以使用諸如 express 等網頁框架來實作。

Node.js

- 複製與負載平衡
  - cluster 模組

現在可以使用流覽器或 curl 傳送請求至 http://localhost:8080，檢查是否如預期般運作。也可使用 Node.js 寫一個簡單的請求:

```
const request = require('request');
const url = 'http://localhost:8080';

for (let i = 100; i > 0; i--) {
  request.get(url, (err, res, body) => {
    console.log(body);
  });
}
```

Node.js

- 複製與負載平衡
  - cluster 模組

現在試著利用 cluster 模組來擴展我們的應用程式。建立一個名為 clusteredApp.js 的新模組。

```
const cluster = require('cluster');
const os = require('os');

if(cluster.isMaster) {
  // cluster.schedulingPolicy = cluster.SCHED_RR
  const cpus = os.cpus().length;
  console.log(`Clustering to ${cpus} CPUs`);
  for (let i = 0; i < cpus; i++) {
    cluster.fork();
  }
} else {
  require('./app');
}
```

使用 cluster 模組非常簡單，是的，就是這樣就完成了垂直擴展，充分應用了本機的資源。

Node.js

- 複製與負載平衡
  - cluster 模組

我們來分析這是怎麼回事:

1. 從命令列啟動 clusteredApp 時，實際上便是啟動了主程序。cluster.isMaster 變數會自動設為 true，而我們所需做的就只是使用 cluster.fork() 來分支化當前的程序。在這範例中，我們起始與系統 CPU 個數相同數目的工作者(worker)，為的是盡可能利用系統的完整運算能力。
2. 當主程序執行 cluster.fork() 時，現行的主模組(clusteredApp)便會再執行一次，然而這次是處於工作者模式(因為 cluster.isWorker 會被設為 true，而 cluster.isMaster 則為 false)。當應用程式作為工作者執行時，便會開始進行具體的工作內容。在這個範例裡，當載入 app 模組時，其實就是啟動一個新的 HTTP 伺服器。

cluster 模組使用方式正是基於一種遞迴模式，此舉能夠輕易地執行應用程式的多個實例。

Node.js

- 複製與負載平衡
  - cluster 模組

這裡要切記的是，每個工作者都是個別的 Node.js 程序，它們都有自己的事件迴圈、記憶體空間以及所載入的模組。

就底層而言，cluster 模組是使用 child_process.fork() API，因此，主程序與工作者之間也存在著溝通管道。可透過變數 cluster.workers 來存取工作者實例，所以要廣播一則信息給所有實例，只需執行如下程式碼:

```
if(cluster.isMaster){
  [...]
  Object.keys(cluster.workers).forEach(function(id){
    cluster.workers[id].send({id: id, message: 'Hello from master'});
  });
} else {
[...]
```

在工作者則可接收到訊息，可在 app.js 中加入如下的程式碼

```
process.on('message', function(data){
  console.log(data.message);
});
```

Node.js

- 複製與負載平衡
  - cluster 模組的可恢復性與可用性

如同我們曾經提及的，對應用程式進行擴展也有其它的優點，具體來說是即便處於故障或崩潰的情況，也能夠維持一定程度的服務。此種特性亦稱為**可恢復性(resiliency)**，而且對系統整體的**可用性**有所幫助。

為應用程式建立多個實例，就能夠建置出一個富餘的系統(redundant system)。意即若有個實例因故停擺，也依然有其它實例可回應這些請求。這種模式可以非常直接的以 cluster 模組實作，以下就來看看它是如何運作的！

透過 cluster 模組，我們所需做的就是在偵測到有任一工作者因錯誤終止時，便立即生成一個新的工作者。讓我們修改 clusteredApp.js 模組，來實作這項功能。

```
if(cluster.isMaster){
  [...]
  cluster.on('exit', (worker, code) => {
    if(code != 0 && !worker.exitedAfterDisconnect) {
      console.log(`Worker crashed. Starting a new worker`);
      cluster.fork();
    }
  });
} else {
```

Node.js

- 複製與負載平衡
  - cluster 模組的可恢復性與可用性

在前述程式碼中，若主程序接收到 'exit' 事件，便檢查該程序是正常結束，還是因錯誤而結束。透過檢查狀態 code 與旗標 worker.exitedAfterDisconnect 來進行確認，後者能夠判斷出工作者是否為主程序刻意終止的。如此我們確認該程序是因錯誤而終止，便啟動一個新的工作者。值得留意的是，雖然崩潰的工作者正重新啟動，但其它的工作者仍然能夠回應請求，所以不會影響應用程式的可用性。

要了解的是，在有工作者崩潰而重啟，有些請求會失敗，這是因為已建立的連線因崩潰而中斷。不幸的是，我們無法保證所有的請求都會成功，也沒有甚麼方案將這類問題處理得更好，因為所謂的程式崩潰本就是詭譎多變的。但儘管如此，我們的解決方案仍被證明是可行的，能夠為經常崩潰的應用程式提供相當強建的可用性。

Node.js

- 複製與負載平衡
  - 零停機時間重啟

Node.js 應用程式可能會因為程式碼更新，而必須重新啟動。同樣的，這類情況也可以借助多個實例來維持應用程式的可用性。

當我們必須刻意重新啟動應用程式以進行更新時，被重啟的應用程式勢必會有一段時間，是無法回應請求的。若我們只是在更新個人的部落格，這點停機時間倒還無所謂；但對於標榜**服務層級協議(Service Level Agreement:SLA)**的專業應用程式，若是採**持續交付(continuous delivery)**流程的應用程式，則絕對是無法接受的。因此，可行的解決方案便是實作零停機時間的重啟，既能夠更新應用程式的程式碼，又不影響其可用性。

藉由 cluster 模組，這項任務同樣能夠輕鬆完成。具體的模式內容是一次只重新啟動一個工作者，讓其餘的工作者繼續運作以維持應用程式的可用性。

Node.js

- 複製與負載平衡
  - 零停機時間重啟

現在讓我們將這項新功能加入 clusteredApp.js 中，但由於我們的程式使用了 UNIX 信號，因此無法在 Windows 系統下適切的運作:

```
if(cluster.isMaster){
  [...]
  process.on('SIGUSR2', () => {
    const workers = Object.keys(cluster.workers);

    function restartWorker(i) {
      if (i >= workers.length) return;
      const worker = cluster.workers[workers[i]];
      console.log(`Stopping worker: ${worker.process.pid}`);
      worker.disconnect();

      worker.on('exit', () => {
        if (!worker.exitedAfterDisconnect) return;
        const newWorker = cluster.fork();
        newWorker.on('listening', () => {
          restartWorker(i + 1);
        });
      });
    }
    restartWorker(0);
  });
} else {
```

Node.js

- 複製與負載平衡
  - 零停機時間重啟

前述程式碼區塊的運作方式如下：

1. 在接收到 SIGUSR2 信號時重新啟動工作者。
2. 定義一個名為 restartWorker() 的迭代函式。它會對 cluster.workers 物件的項目，實作非同步循序迭代模式。
3. restartWorker() 函式的第一項任務是呼叫 worker.disconnect() 以適當地停止工作者。
4. 當欲終止的程序確時結束後，我們就能生成一個新的工作者。
5. 只有當新的工作者已準備好監聽新連線時，才呼叫迭代的下一步，繼續啟動下一個工作者。

由於我們的程式使用了 UNIX 信號，因此無法在 Windows 系統下適切的運作，在 UNIX 下可以用 ps –ef 找出主程序的 PID，然後可以用 kill –SIGUSR2 <PID> 傳送信號給主程序：

```
Stopping worker: 7888
Started 7480
Stopping worker: 3132
Started 8232
...
```

Node.js

- 複製與負載平衡
  - 零停機時間重啟

我們可以在 Windows 下模擬在 UNIX 送出 SIGUSR2 信號:

```
if(cluster.isMaster) {
  [...]
  setTimeout(() => {
    process.emit('SIGUSR2');
  }, 10000);
} else {
  [...]
}
```

這會在 10 秒鐘後，送出 SIGUSR2 信號，而零停機時間重啟工作者程序。

Node.js

- 以反向代理器(reverse proxy)擴展

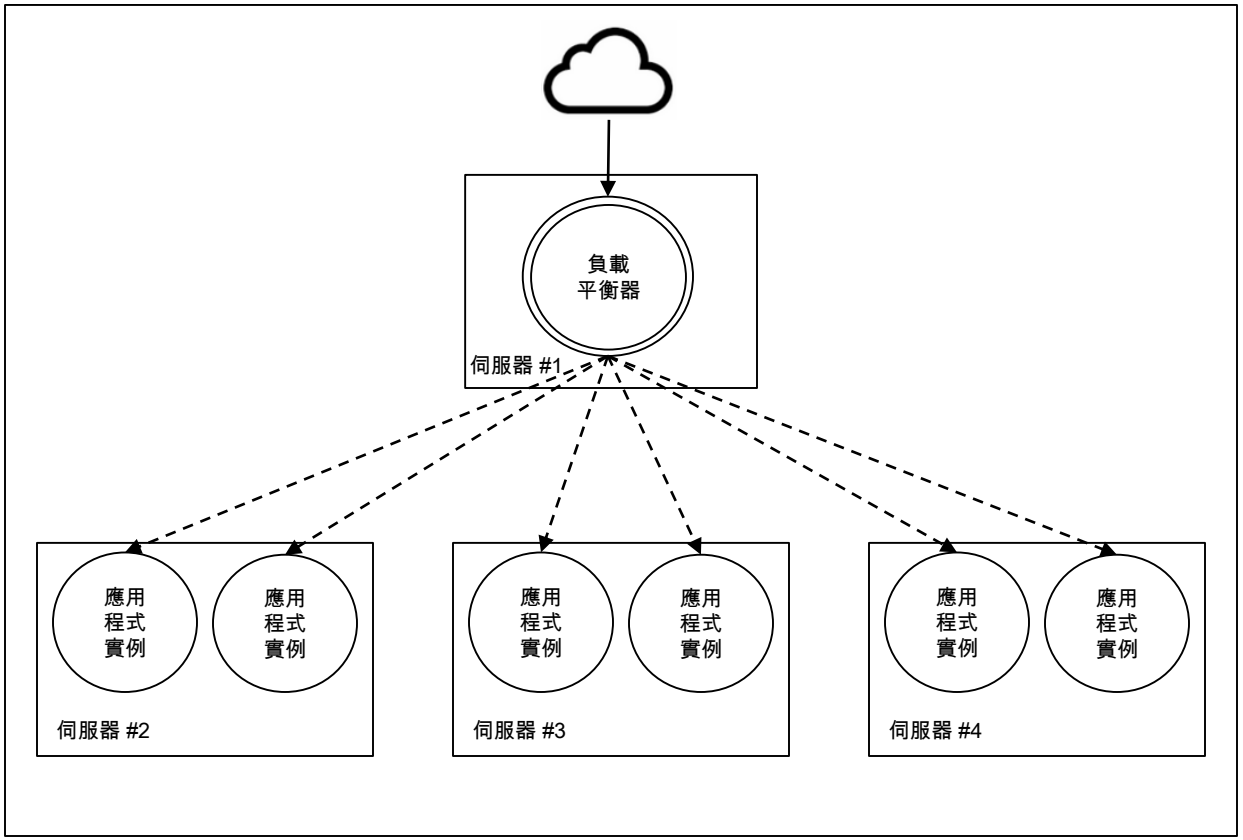
cluster 模組並非擴展 Node.js 應用程式的唯一選項。事實上，也有一些更傳統的技巧是常被使用的，因為這些技巧能夠為已經上線的環境，提供效能與可控制性的提升。

這項有別於 cluster 的方案，是在不同通訊埠或不同機器上啟動應用程式的多個實例，然後使用一個反向代理器(reverse proxy)(或閘道器 gateway)，來存取這些實例，藉此分散流量。在這樣的設定下，不再有一個主程序負責將請求分配給多個工作者，但會有多個更加獨立的程序執行於同一機器(使用不同的通訊埠)上，或者分散在相同網路中的不同機器上。我們以一個反向代理器作為應用程式的入口，這項特殊的裝置或服務是位於客戶端與應用程式實例之間，它會接收請求並轉遞至伺服器，然後回傳結果給客戶端。也就是說，它的對外表現就如同自己是伺服器一般。因此，反向代理器也能作為負載平衡器，將請求分派給多個應用程式實例。

Node.js

- 以反向代理器(reverse proxy)擴展

下面的圖片是說明一種典型的應用方式，將反向代理器置於前端，為後端的多個程序及多部機器提供負載平衡：



Node.js

- 以反向代理器(reverse proxy)擴展

有非常多的原因鼓勵我們的 Node.js 應用程式應採用這種作法，而非使用 cluster 模組:

- 反向代理器可以實現多部機器的分散負載，而不只是多個程序
- 常見的反向代理器也支援沾黏負載平衡(sticky load balancing)

  預設情況下，傳統負載均衡器會將每項請求單獨路由到負載最小的已註冊實例。但是，您可以使用粘性會話功能（也稱為會話關聯），使負載均衡器能夠將用戶會話綁定到特定的實例。這可確保在會話期間將來自用戶的所有請求發送到相同的實例中。
- 反向代理器背後的多個程序，可以是跨平台或是跨程式語言的
- 可以實作更強大的負載平衡演算法
- 許多反向代理器還提供一些其他的功能，例如 URL 重寫、快取、SSL 終端點，以及輸出靜態檔案(如同一般常見的網頁伺服器功能)等等

事實上，如有必要， cluster 模組也可以輕易的與反向代理器合併使用。例如，於單一機器裡使用 cluster 作垂直性的擴展，再使用反向代理器於不同節點之間作水平性的擴展。

Node.js

- 以反向代理器(reverse proxy)擴展

有很多種使用反向代理器實作負載平衡的方式，常見的解決方案有：

- Nginx(http://nginx.org)：這同時是一款網頁伺服器、反向代理器及負載平衡器，基於非阻塞式的 I/O 模型。
- HAProxy(http://www.haproxy.org)：這是一款針對 TCP/HTTP 流量的高負載平衡器。
- 基於 Node.js 的代理器：有許多解決方案是直接使用 Node.js 來實作反向代理器與負載平衡器。它們各有其優缺點，稍後我們會有更多說明。
- 基於雲端的代理器：在這個雲端運算的時代，透過某些服務實現負載平衡並不罕見。這種作法所需的維護工作最少，並且容易擴展，甚至有時還可以動態地依需求擴展。

Node.js

- 以反向代理器(reverse proxy)擴展
  - 使用 http-proxy 與 seaport 實作動態的負載平衡器。

現今流行的雲端基礎架構有一項重要優勢，就是能夠依據目前或預測性的流量，動態的調整應用程式的可用資源。此又稱為**動態擴展**，若實作得宜，可大幅降低 IT 建置成本，同時維持良好的可用性及回應能力。

具體的實作概念很簡單，如果應用程式正因高流量而導至效能減損時，便自動隨負載的增長，生成新的伺服器。我們也可以決定在某些時刻停用部份伺服器，例如半夜，因為我們知道那時流量很少，然後到早上再啟動伺服器。此種機制需要負載平衡器持續更新最新的網路拓撲資訊，掌握每時每刻的伺服器上線狀況。

解決這類問題的常用模式是使用名為**服務註冊表(Service Registry)**的中央倉庫，對執行中的伺服器及服務保持追蹤。每個應用程式實例都必須在上線時註冊到服務註冊表裡，並在實例停用時註銷。如此一來，負載平衡器就能持續取得最新的可用伺服器清單。

接下來的範例，我們將使用兩個 npm 套件來實作:

- http-proxy: 這個函式庫能夠簡化代理器與負載平衡器的建立。
- seaport: 一個極簡單的註冊服務表(Service Registry)。

Node.js

- 以反向代理器(reverse proxy)擴展

我們從服務 app.js 的實作開始，這些都是簡單的 HTTP 伺服器，但是我們需要每個伺服器啟動時，也將自己註冊到服務註冊表裡。

```
const http = require('http');
const seaport = require('seaport').connect('localhost', 9090);
const serviceType = process.argv[2];
const pid = process.pid;


const port = seaport.register(serviceType);


http.createServer((req, res) => {
  for(let i = 1e7; i > 0; i--) {};
  console.log(`Handling request from ${pid}`);
  res.end(`${serviceType} response from ${pid}\n`);
}).listen(port, () => {
  console.log(`Started ${pid}`);
});
```

初始化一個 seaport 客戶端，並連線至註冊伺服器

從命令列讀取 serviceType，作為伺服器所提供的服務類型。此只是為了模擬多服務環境，而不必真實作出多種伺服器

使用 seaport.register() 註冊服務，此函式會回傳一個 port 編號，讓 HTTP 伺服器在指定的通訊埠上啟動

註冊表會在與 HTTP 伺服器失去連線時，自動註銷對應的服務。這表示我們不需自己動手，當伺服器一停就會從註冊表中消失

Node.js

- 以反向代理器(reverse proxy)擴展

現在實作負載平衡器 loadBalancer.js:

```
const routing = [
  {
    path: '/api',
    service: 'api-service',
    index: 0
  },
  {
    path: '/',
    service: 'webapp-service',
    index: 0
  }
];
```

routing 陣列裡的各個項目會包含一個 service，也就是處理來自 path 請求的服務。至於 index 屬性則用於對該服務的請求做循環式(round robin)的處理。

Node.js

- 以反向代理器(reverse proxy)擴展

以下則是 loadBalancer.js 的後續部份:

```
const http = require('http');
const httpProxy = require('http-proxy');
const seaport = require('seaport').connect('localhost', 9090);
const proxy = httpProxy.createProxyServer({});

http.createServer((req, res) => {
  let route;
  routing.some( entry => {
    route = entry;
    return req.url.indexOf(route.path) === 0;
  });

  const servers = seaport.query(route.service);
  if(!servers.length) {
    res.writeHead(502);
    return res.end('Bad gateway');
  }

  route.index = (route.index + 1) % servers.length;
  proxy.web(req, res, {target: servers[route.index]});

}).listen(8080, () => {console.log('Started.')});
```

連線至註冊伺服器，以存取註冊表

實例化 http-proxy 物件，並啟動一個普通的網頁伺服器。

在網頁伺服器的請求處置器中，比對 URL 與路由表是否相符，其結果將會是一個包含服務名稱的描述項。如果都沒符合的項目 route 將會是最後一項，也就是 path '/'

自 seaport 獲得相關服務的伺服器清單。若名單為空，即回傳錯誤給客戶端。此外，為了達到最佳速度，seaport 會將註冊表置入於本地快取中，並與註冊表伺服器保持同步更新。因此 seaport.query() 事實上是一個同步呼叫。

最後，我們可以將請求輸送到它的目的地。這理以循環式(round robin)的方式處理，更新 route.index 以指向下一個伺服器。將請求(req)與回應(res)物件傳送給 proxy.web()，並依據索引值從清單選取伺服器。此舉會單純地將請求傳遞給指定的伺服器。

Node.js

- 以反向代理器(reverse proxy)擴展

可以看到，僅僅使用 Node.js 來實作負載平衡器非常簡單，而且還能擁有豐富的彈性。現在我們準備好執行它了，不過先得執行下列指令來安裝 seaport 伺服器。

```
npm install seaport -g
```

接著可以使用如下指令來啟動 seaport 服務註冊表:

```
seaport listen 9090
```

之後便可以啟動負載平衡器:

```
node loadBalancer
```

現在若試著存取負載平衡器所揭露的一些服務，將會回傳 HTTP 502 錯誤，這是因為我們還未啟動這些服務背後負責的伺服器。你可以試試以下指令；

```
curl localhost:8080/api
```

上述命令應該會回傳如下結果:

```
Bad gateway
```

Node.js

- 以反向代理器(reverse proxy)擴展

不過若是開始啟動服務的實例，結果就不一樣了。例如分別啟動兩個 api-service 與一個 webapp-service:

```
node app.js api-service
node app.js api-service
node app.js webapp-service
```

現在連續執行兩次:

```
curl localhost:8080/api
```

則應該會收到來自不同伺服器的訊息，這表示請求已平均分散在多個伺服器上:

```
api-service response from 8824


api-service response from 9204
```

這個模式的優勢是立即見效的。我們可以藉此動態的擴展基礎架構。無論是依需求，還是依據排程計畫。而負載平衡器將能夠自動載入新的清單，再也不必自己動手。

Node.js

- Networking with Sockets

Node.js 從開始就設計用於網絡編程的，因此就內置支持低階的 socket connections。TCP sockets 構成了現代網絡應用程序的支柱，理解它們將可為您提供良好的服務支援，進而可進行更複雜的網絡運用。

TCP socket 連接由兩個端點組成。一個端點綁定到一個編號的端口(numbered port)，而另一個端點連接到這個編號的端口。

這很像電話系統。一部電話長時間綁定一個給定的電話號碼。第二個電話撥打電話，連接到綁定號碼。一旦接聽電話，信息就可以雙向傳播。

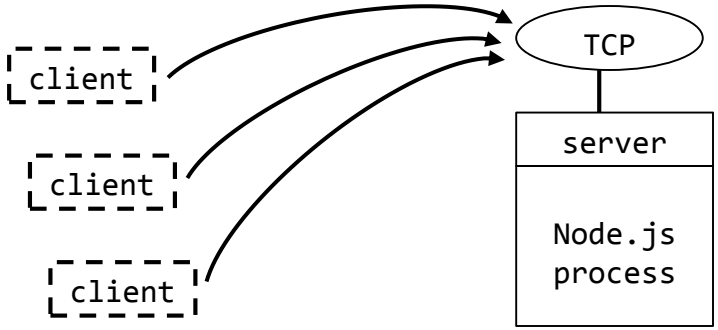
Node.js

- Networking with Sockets

首先我們需要啟動一個監聽器綁定到一個編號的端口：

```
const net = require('net');


net.createServer(connection => {
  // Use the connection object for data transfer
}).listen(60300);
```

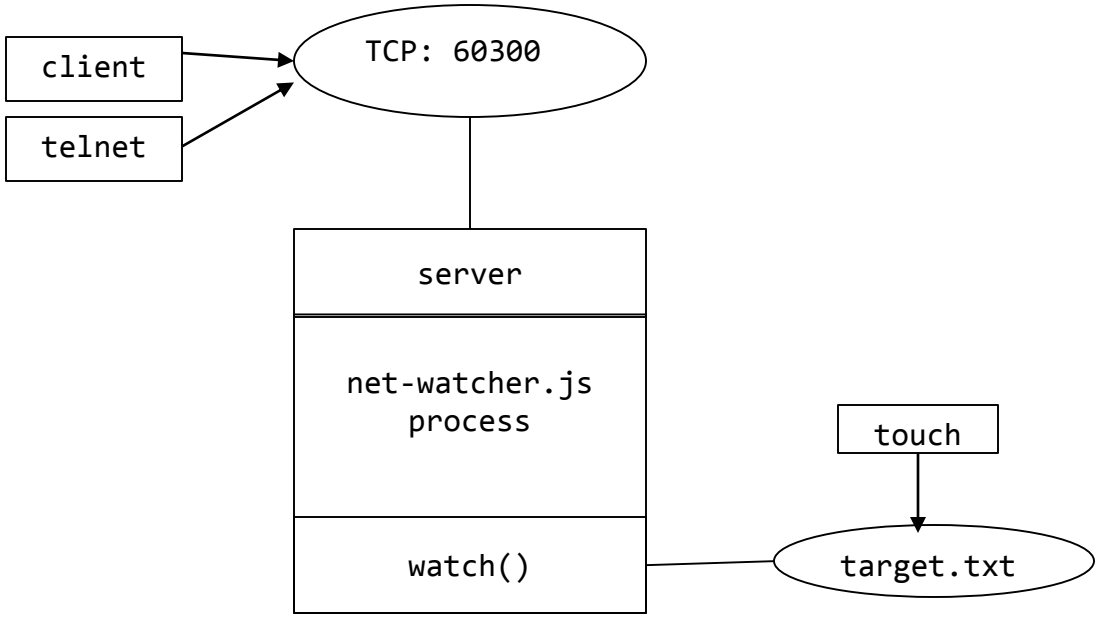
net.createServer() 方法接受一個處理器。只要另一個端點連接，就會調用此處理器。connection 是一個 Socket 物件，可用於發送或接收數據。listen() 綁定指定的 TCP 端口 60300。就是這樣，你已經建立一個 TCP 端口。



Node.js

- Networking with Sockets

現在讓我們建立一個實用的例子:



此例會監控檔案 target.txt，當它被更動時，會透過 TCP 端口送訊息給連上此端點的客戶端。客戶端可以使用一般的 TCP 客戶端的工具例如，telnet、nc 等，也可使用 Node.js 的核心庫 net。

Node.js

- Networking with Sockets

這裡是 net-watcher.js

```
const fs = require('fs');
const net = require('net');
const filename = process.argv[2];

if (!filename) {
  throw Error('Error: No filename specified');
}

net.createServer(connection => {
  console.log('Subscriber connected');
  connection.write(JSON.stringify({type: 'watching', file: filename}) + '\n');

  const watcher = fs.watch(filename, () => {
    connection.write(JSON.stringify({type: 'changed', timestamp: Date.now()}) + '\n');
  });

  connection.on('close', () => {
    console.log('Subscriber disconnected');
    watcher.close();
  });
  connection.on('error', err => console.log(err));
}).listen(60300, () => console.log('Listening for subscribers'));
```

Node.js

- Networking with Sockets

現在創建 target.txt 用來被監控，然後就可以啟動 net-watcher.js

```
touch target.txt
```

```
node net-watcher ./target.txt
```

現在可以使用 telnet 連上此 TCP 端口

```
telnet localhost 60300
```

現在應該可以接收到服務端送來的訊息:

```
{"type":"watching","file":"./target.txt"}
```

試著更動檔案內容並儲存，則會接到:

```
{"type":"changed","timestamp":1549936360764}
```

Node.js

- Networking with Sockets

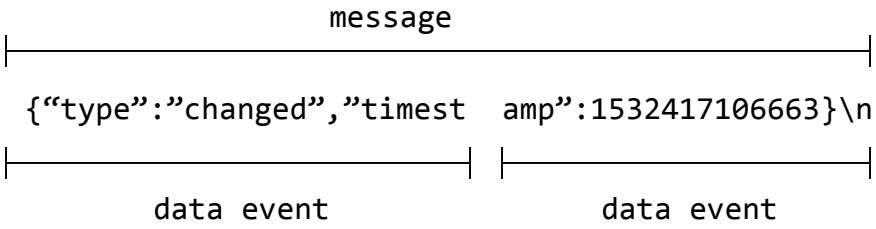
也可直接使用 Node.js 的核心模組，建立客戶端連結:

```
const net = require('net');
const client = net.connect({port: 60300});

client.on('data', data => {
  const message = JSON.parse(data);
  if (message.type === 'watching') {
    console.log(`Now watching: ${message.file}`);
  } else if (message.type === 'changed') {
    const date = new Date(message.timestamp);
    console.log(`File changed: ${date}`);
  } else {
    console.log(`Unrecognized message type: ${message.type}`);
  }
});
```

Node.js

- Networking with Sockets

看起來似乎很簡單，但是網路傳輸沒有這麼單純，在最好的情況下，消息將一次到達，但有時消息將分成幾部分或不同的數據事件，開發網絡應用程序，您需要在它們發生時處理這些分裂。



至於訊息的格式則需要雙方的協義。協議 (protocol) 是一組規則，用於定義系統中端點的通信方式。無論何時在 Node.js 中開發網路應用程序，您都在使用一個或更多協議。在這裡，我們基於通過 TCP 傳遞 JSON 消息來創建協議，每條消息都是一個 JSON 序列化物件(JSON-serialized object)，它是鍵值對的散列(a hash of key-value pairs)。

所以，使用一些網路的函式庫，才是聰明的選擇。

而這些作業的主角最主要就是用來交換的「訊息」，這就牽涉到應用程式擴展立方的 Y 軸。

Node.js

- 訊息與整合模式

如果說擴展性的內涵是切割分離，那麼系統整合的內涵就是合併了。我們已學到如何將應用程式分散於多部機器之中。然而，為了讓應用程式順利運作，這些個體都應該透過某種方式彼此通訊。也就是說，它們必須被整合起來。

整合一個分散的應用程式主要有兩種技巧:

- 一種是使用共享存儲作為中央的協調者及管理者
- 另一種則是透過訊息，在各個節點之間散播資料、事件及命令

訊息的使用不僅會對分散式系統的擴展，產生具體的影響，同時其內涵也是相當多變，甚至是有些複雜的。

訊息可應用於軟體系統的所有層級，例如網際網路上的通訊、本地程序間的資料傳遞、驅動程式與硬體之間的溝通，以及在應用程式裡，作為直接函式呼叫的替代方案(命令模式)。廣義來說，任何在邏輯個體或實體之間傳遞的資料都可以被稱為訊息。不過，在分散式架構裡，所謂的訊息系統則是專指網路架構中的資訊交換機制。

訊息系統的具體實作也是形形色色。可以選擇中介者架構或對等架構，也可能使用請求/回覆的通訊或是單行通訊，或是使用佇列，以更加可靠的方式傳遞訊息。由此可見，這其實是蠻浩大的主題。

Node.js

- 訊息與整合模式
  - 訊息系統的基礎

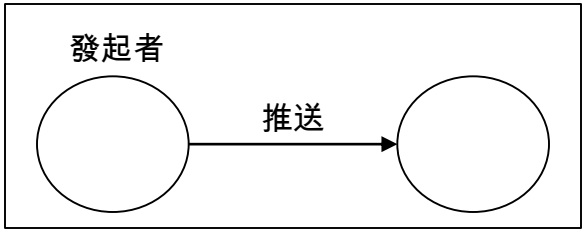
當涉及到訊息與訊息系統的議題時，我們便必須考量到如下四個基礎元素：

- 通訊的方向，例如單行，或是請求/回覆式的雙行通訊
- 訊息的目的，而這也涉及了訊息的確切內容
- 訊息的發送時機，例如立即性的傳送與接收，或是過一段時間後再進行(非同步)
- 訊息的傳遞方式，例如直接傳遞或是透過一個中介者

Node.js

- 訊息與整合模式
  - 單行

通訊方向是訊息系統中最根本的層面，而整個訊息系統的設計語義也經常源自於此。最簡單的通訊模式，是訊息以單行的方式從來源推送到目的地。

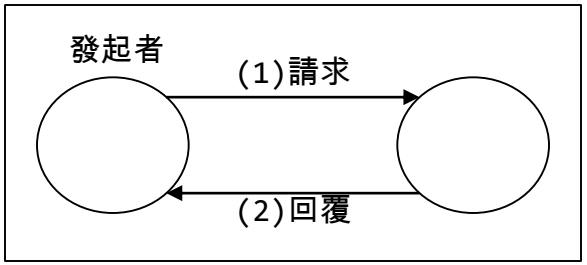


單行通訊的典型範例包含了:電子郵件、網頁伺服器透過 WebSocket 傳遞訊息給瀏覽器，以及系統將任務分派給多個工作者的行為。

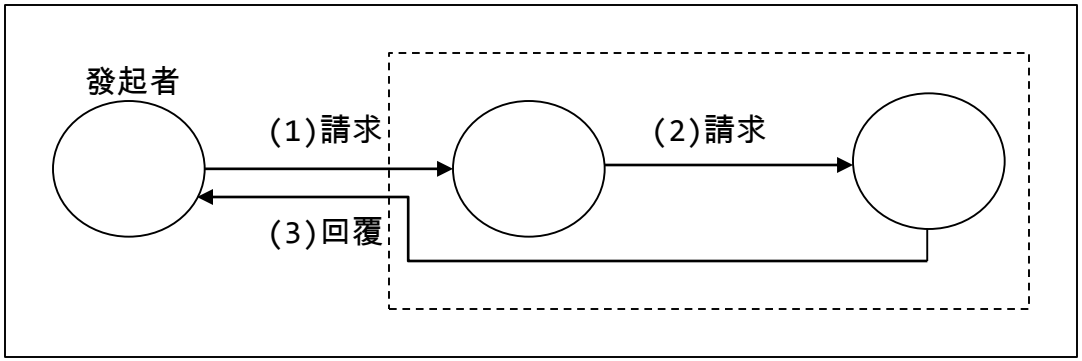
Node.js

- 訊息與整合模式
  - 請求/回覆模式

而另一方面，請求/回覆模式則比單行通訊更為常見許多，其典型範例就是網頁服務的呼叫。



請求/回覆模式或許看起來很簡單，但我們也即將發現，如果通信是非同步的，或者是需要呼叫多個節點時，它就會開始複雜起來了。



Node.js

- 訊息與整合模式
  - 請求/回覆模式

我們可以透過前圖瞭解到，為何請求/回覆模式的某些應用情境是比較複雜的。如果我們只考量任意兩個節點之間的通信方向，那麼絕對可以說那是單行。然而若從整體來看則會意識到，當發起者傳送請求後，便會由其他節點再傳送回覆給發起者，而這個傳送回覆的節點可能不同於一開始接收到請求的節點。在這些情況下，請求/回覆模式與純單行方式的真正差異在於，前者的訊息回覆必須確保會回到最初的訊息發起者。

- 訊息型態

所謂的信息基本上就是讓不同的軟體元件得以彼此互動的事物，其中的目的各有不同，可能是為了獲取其他系統或元件的一些資訊，以便遠端執行操作；或者是為了通知某些端點有某件事情剛發生。而訊息的內容也會因為通訊目地的不同而有所差異。一般而言，根據訊息的目地可區分出三種訊息型態:

- 命令訊息 (Command Message)
- 事件訊息 (Event Message)
- 文件訊息 (Document Message)

Node.js

- 訊息與整合模式

  - 命令訊息型態的目地，是為了在接收者中觸發某項任務的執行。對此，這個訊息必須包含執行任務所需的必要資訊，通常是操作名稱與參數清單。命令訊息可以用於實作遠端程序呼叫(Remote Procedure Call: RPC)系統、分散式運算、或者是單純的用來取得一些資料，REST 式的 HTTP 呼叫就是命令的一項簡單範例，每一個 HTTP 動詞皆具特定的意涵，並繫結於特定的操作: GET 取回資源、 POST 建立新資源、PUT 更新資源，而 DELETE 則是銷毀資源。

  - 事件訊息是用於通知其他元件有事情發生。通常其內容即包含事件型態，以及一些細節，例如相關的環境、主題及對象。在網站開發中，我們可以使用長期輪詢(long-polling)或 WebSocket，在瀏覽器上取得來自伺服器的是件通知，例如資料的變更，或系統狀態的改變等等。事件在分散式應用程式裡是相當重要的整合機制，因為它可以讓系統的所有節點皆保有一致的認知。

  - 文件訊息主要用於在多個元件及機器之間傳送資訊。由於命令訊息可能也包含部份資料，因此他與文件訊息主要的差異則在於，文件訊息不會指示接收者應執行那些操作。至於文件訊息與事件訊息的主要差異則在於，前者無關於任何特定事件的發生。除此之外，文件訊息經常被用於回覆命令訊息，內含請求的資料或操作的結果。

Node.js

- 訊息與整合模式

  - 非同步訊息與佇列

作為 Node.js 的開發者，我們應該已經瞭解非同步操作的好處，而同樣的道理也適用在訊息與通訊上。

同步通訊就像是在撥打一通電話，兩個端點之間必須接通，而訊息的交換應該是即時的。一般情況下，若我們於同時想再撥打一通電話給其他人，若不是使用另一支電話，就得先結束目前的通話才行。

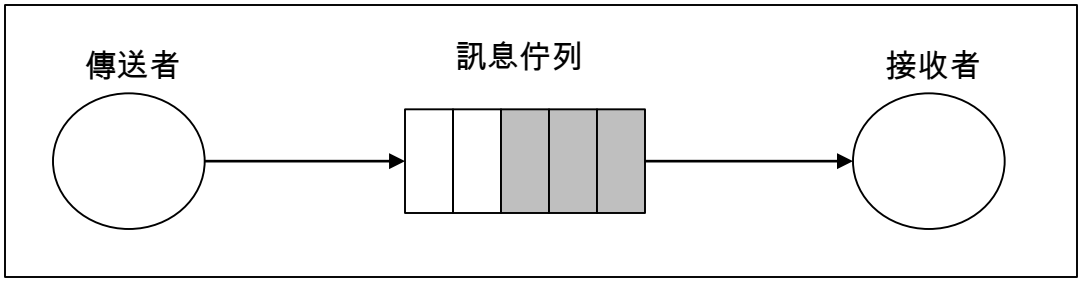
至於非同步通訊則類似於簡訊，接收者無須持續保持接通，我們可能會立即收到回應，或在未知的間隔後才收到，甚至完全不會收到。我們可能會傳送多封簡訊給多名接收者，然後以任意順序接收回覆(如果有的話)。簡而言之，使用更少資源就能達到較好的平行性。

Node.js

- 訊息與整合模式

  - 非同步訊息與佇列

非同步訊息的另一項重要優勢是訊息可以儲存，待稍後再作傳遞。當接收者過於忙碌，以致於無法即時處理新訊息時，這便是非常有用的優勢。在訊息系統中，我們可以透過**訊息佇列(message queue)**來實現這項機制，此種元件能夠協調傳送者與接收者之間的通訊，先保存所有的訊息再傳遞到目的地。



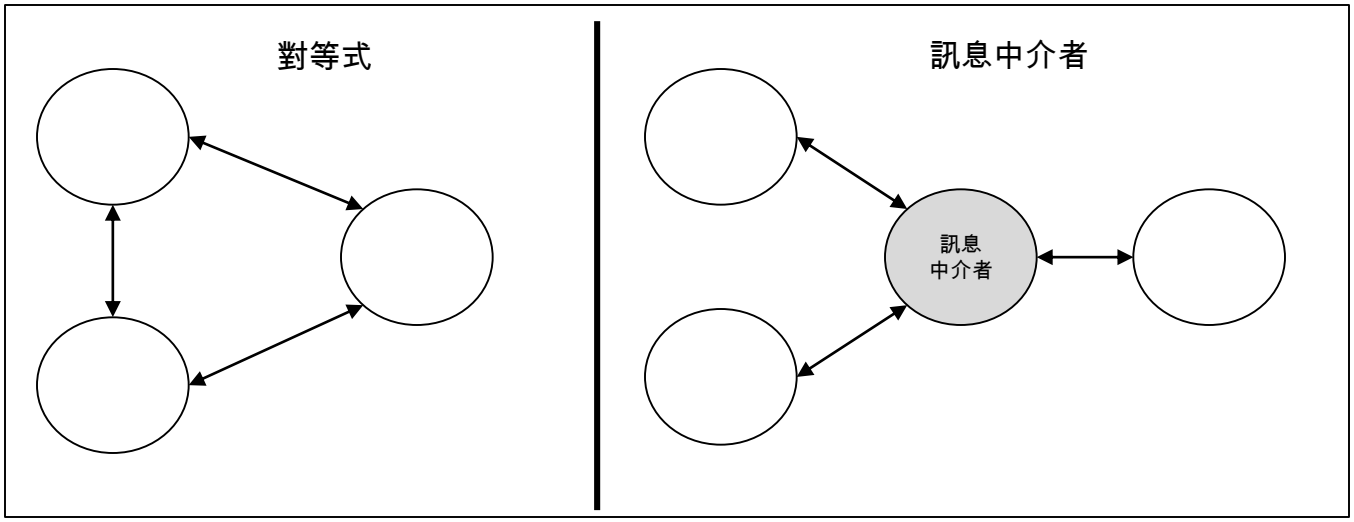
如此一來，若接收者因故崩潰、網路連線中斷或延遲，則訊息將會累積在佇列裡，當接收者恢復運作後便重新派送。佇列可以置於傳送者中，或者是置於傳送者與接收者之間，也就是一個獨立的元件，作為兩端之間通訊的**中介軟體**。

Node.js

- 訊息與整合模式

  - 對等式或基於中介者的訊息

訊息能夠以對等(peer-to-peer)的方式，直接傳遞給接收者；或者是透過一個名為訊息中介者(Message Broker)的中央媒介系統，來加以傳遞。中介者的主要作用是解除訊息接收者與傳送者之間的耦合。



Node.js

- 訊息與整合模式

  - 對等式或基於中介者的訊息

在對等式架構下，每一個節點都是直接將訊息傳送給接收者。這意謂著各個節點都必須知道接收者的位址及通訊埠，而且彼此之間皆必須採用一致的通訊協定及訊息格式。至於中介者則去除了這些對等關係所引致的複雜性，每個節點可以完全獨立，而且可以與各式節點通訊，而無須知道這些節點的細節。除此之外，中介者還能夠作為不同通訊協定之間的橋接器，例如常見的 RabbitMQ 中介者即支援了**進階訊息佇列通訊協定**(Advanced Message Queuing Protocol: AMQP)、**訊息佇列遙測傳輸**(Message Queue Telemetry Transport: MQTT)，以及**簡單/串流文字導向訊息通訊協定**(Simple/Streaming Text Oriented Message Protocol: STOMP)，讓使用不同訊息協定的應用程式能夠彼此互動。

MQTT 是一種輕量級的訊息通訊協定，是針對機器之間(物聯網)的通訊特別設計的。至於 AMQP 則是比較複雜的通訊協定，是為了對各式封閉的訊息中介軟體，提供一種開放原始碼的替代方案。STOMP 是基於文字的輕量級通訊協定，是依循 HTTP 設計概念而成。這三種都是應用程式層級的通訊協定，並且都是以 TCP/IP 為基礎。

去年我們導入的 Apache Kafka 也有中介者的功能，是一種串流式的訊息系統。

Node.js

- 訊息與整合模式

  - 對等式或基於中介者的訊息

除了避免耦合及確保互通性外，中介者還能夠提供更進階的功能，例如佇列保存、路由、訊息轉換及監控等等，以及各式各樣由中介者內建支援的訊息模式。當然，我們仍然可以強硬的使用對等架構來實作這些功能，但此舉很費工夫。不過另一方面，若不使用中介者，則也存在一些優點:

- 避免單點故障
- 減少了一個可能需要擴展的元件
- 訊息的交換沒有中間媒介，勢必會有利於傳輸速度

事實上，實作對等式的訊息系統，仍然可以不被任何特定技術、通訊協定或架構所捆綁，藉此獲得更多的彈性與助益。常見的低階訊息函式庫 ØMQ 即完美證明了，我們可以非常靈活的建構出自訂的對等式(或混合式)架構。

ØMQ 支持許多不同的消息傳遞模式，這些模式在 Node.js 中都運行良好：

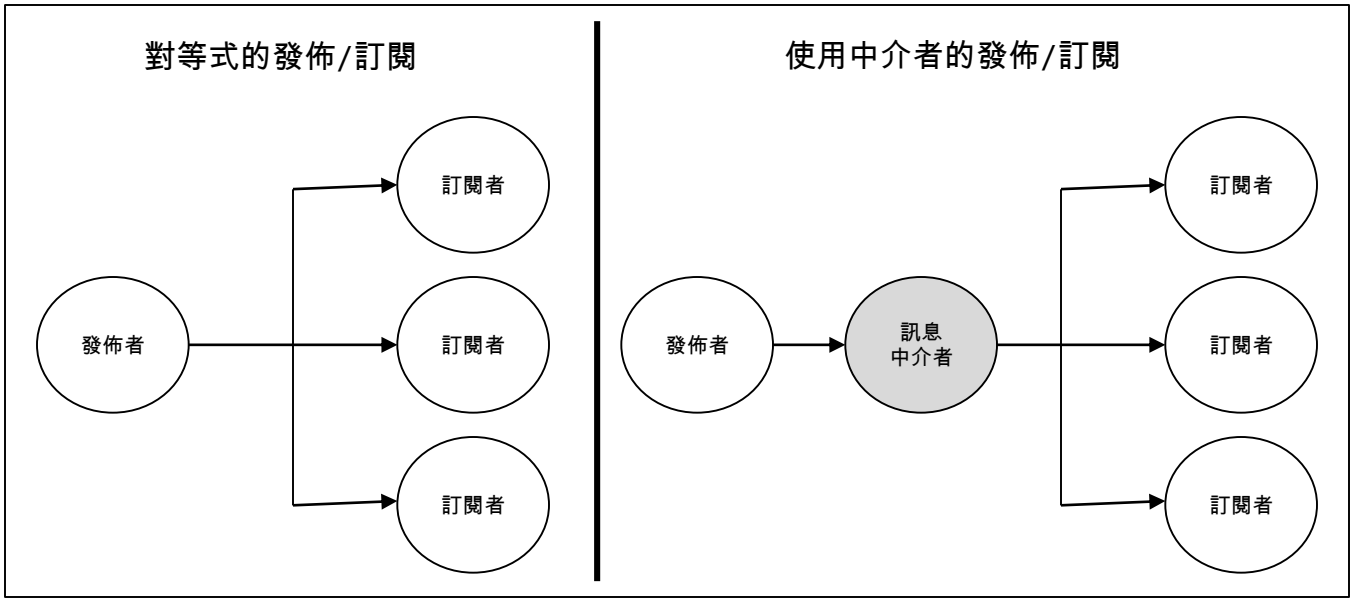
- 發布和訂閱消息(publish/subscribe)
- 回應請求(request/reply)
- 路由和處理消息(dealer/router)
- 推送和拉取消息(push/pull)

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式

發佈/訂閱模式(經常簡稱為 Pub/Sub)可能是最為人所知的一種單行訊息模式。它其實就是一種分散式的**觀察者模式**。我們會有多個**訂閱者**註冊想接收的訊息種類，至於**發佈者**則會將訊息散佈給相關的訂閱者。下圖呈現了發佈/訂閱模式的兩種變化形式:一種是對等式、另一種則是使用中介者:



Node.js

- 訊息與整合模式

  - 發佈/訂閱模式

發佈/訂閱如此特別的地方在於：發佈者無須預先知道訊息的接收者。如同先前所提及的，由於是訂閱者主動註冊想要接收的訊息，因此發佈者自身其實對接收者的數目一無所知。換句話說，發佈/訂閱模式是鬆散藕合的，使之成為一種最理想的模式，用於在變化多端的分散式系統中整合各個節點。

至於中介者的使用，更是進一步提升了節點之間的偶合程度，由於訂閱者只與中介者互動，所以不會知道哪一個節點是訊息發佈者。而且中介者也能作為一種訊息佇列系統，讓節點之間即便出現連線問題，也能確保遞送程序的可靠性。

為了展示一項真實範例，藉此瞭解發佈/訂閱模式能夠如何協助我們整合分散式的架構，以下我們將使用純粹的 WebSocket 來建置一個最基本的即時通訊應用程式。然後我們將執行多個實例來擴展此程式，並透過訊息系統讓這些實例彼此往來。

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (實作伺服器端)

我們需要 ws 套件，這是 Node.js 的純 WebSocket 實作。在 Node.js 下實作即時的應用程式非常簡單，以下便是會話功能的伺服器端內容(app.js):

```
const WebSocketServer = require('ws').Server;
const server = require('http').createServer(
  require('ecstatic')({root: `${__dirname}/www`})
);

const wss = new WebSocketServer({server: server});
wss.on('connection', ws => {
  console.log('Client connected');
  ws.on('message', msg => {
    console.log(`Message: ${msg}`);
    broadcast(msg);
  });
});

function broadcast(msg) {
  wss.clients.forEach(client => {
    client.send(msg);
  });
}

server.listen(process.argv[2] || 8080);
```

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (實作伺服器端)

如此而已！ 以上就是即時通訊程式的伺服器端程式碼。其運作方式如下：

1. 先建立一個 HTTP 伺服器，並附加一個名為 ecstatic 的中介軟體(Middleware)，來提供輸出靜態檔案的服務。此舉是為了供應應用程式客戶端所需要的資源(JavaScript 與 CSS)，所以我們會把 index.html 檔案放在當前路徑的 www 目錄下。
2. 建立一個 WebSocket 伺服器的新實例，然後將這著實例附加到現有的 HTTP 伺服器。所以這兩個伺服器會共用一個通訊埠。接著對 connection 事件附加事件監聽器，監聽傳入的 WebSocket 連線。
3. 當有新的客戶端連線到伺服器時，便監聽傳入的訊息，並將訊息廣播給所有已連線的客戶端
4. broadcast() 函式只是單純的迭代所有已連線的客戶端，並呼叫每一客戶端的 send() 函式。

以上就是 Node.js 的極簡魔法！ 當然，這也是因為我們所實作的伺服器相當迷你且基本，但至少它已經能夠執行我們所需要的工作。

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式（實作客戶端）

客戶端程式碼其實就是一個簡單的網頁，內含一些簡單的 JavaScript 程式碼 (www/index.html):

```
<!DOCTYPE html>
<html>
<head></head>
<body>
  Messages:
  <div id="messages"></div>
  <input type="text" placeholder="Send a message" id="msgBox">
  <input type="button" onclick="sendMessage()" value="Send">

  <script>
    const ws = new WebSocket('ws://' + window.document.location.host);
    ws.onmessage = function(message) {
      let msgDiv = document.createElement('div');
      msgDiv.innerHTML = message.data;
      document.getElementById('messages').appendChild(msgDiv);
    };
```

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (實作客戶端)

```
    function sendMessage() {
      let message = document.getElementById('msgBox').value;
      ws.send(message);
    }
  </script>
</body>
</html>
```

這個網頁無須太多的說明，但如果停止或重新啟動會話伺服器時，WebSocket 連線會被關閉而且不會自動重新連線(除非利用 Socket.io 這類高階函式庫)。這表示我們必須在伺服器重新啟動後，也在瀏覽器上做重新整理的動作，才能再次連線(或者實作出一種重新連線機制)。

現在可以啟動伺服器:

```
node app 8080
```

在瀏覽器上前往 http://localhost:8080，多打開幾個頁籤，送出訊息，將會即時顯示在各個頁籤的畫面上。

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式（實作客戶端）

接著我們要看看，當這個應用程式被擴展為不只一個實例時，會發生甚麼事。在其它的通訊埠上啟動另一個伺服器：

```
node app 8081
```

我們所期望的擴展結果，應該是連線到不同伺服器的不同客戶端，彼此之間也能夠交換會話訊息。但結果卻不是這樣，你可以試著在瀏覽器的另一個頁面籤開啟 http://localhost:8081，檢視其結果。

當在某一個實例裡傳送會話訊息時，這個訊息只會做到局部性的廣播，也就是只有該實例的客戶端會接收到訊息。這兩個實例之間無法彼此對話，所以我們必須設法整合。

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (建置 Redis 擔任訊息中介者)

Redis 可說是最重要的一種發佈/訂閱實作方式，它是相當高效且富含彈性的鍵值存儲，並且經常被用作為一種資料架構伺服器，雖然 Redis 更接近於是一種資料庫，而非訊息中介者，不過在它的許多功能之外，卻有一組指令是針對中央式的發佈/訂閱模式所設計。當然，若與更進階的訊息導向中介軟體相比，Redis 的實作非常簡化，然而這也是它之所以如此受歡迎的原因之一。事實上，許多現成的基礎架構經常都會提供 Redis，做為快取伺服器(caching server)或工作階段存儲(session store)。Redis 的效率及彈性，使它成為在分散式系統下分享資料的常見選項。因此，若有任何發佈/訂閱中介者的需求，最簡單而直接的選擇就是使用 Redis 來完成，而非另外設立一個專用的訊息中介者。以下就讓我們舉例說明 Redis 的極簡威力。

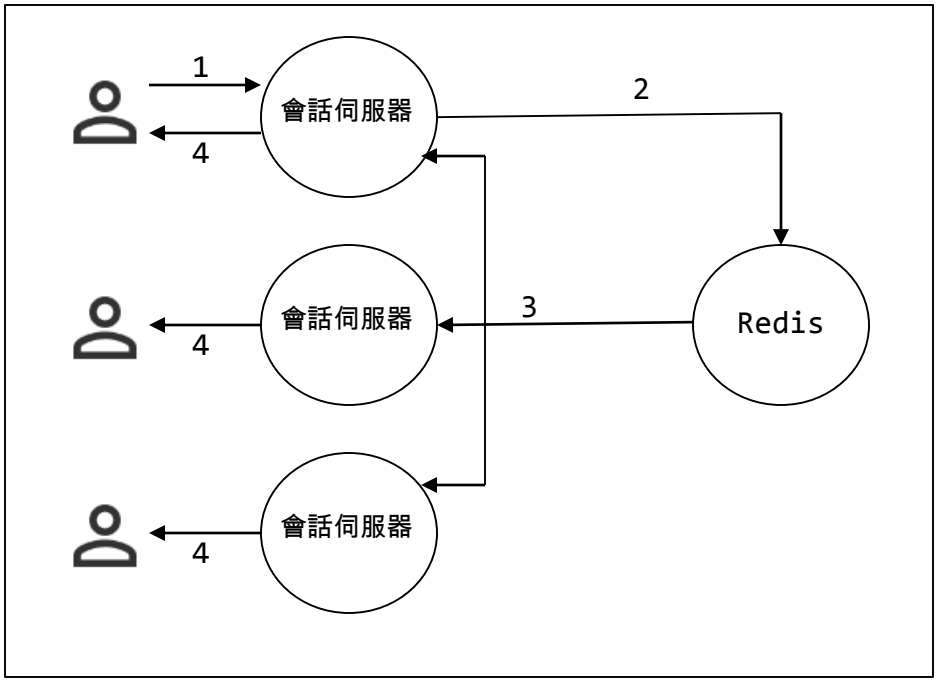
我們的計畫是使用 Redis 作為訊息中介者，以整合我們的會話伺服器。每一個實例會將來自客戶端的所有訊息發佈給中介者，同時訂閱所有來自其它伺服器實例的訊息。如我們所見，這個架構下的每一個伺服器都兼任訂閱者與發佈者。以下即是此架構的示意圖:

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (建置 Redis 擔任訊息中介者)

以下即是此架構的示意圖:



1. 訊息鍵入網頁文字框，再傳送至已連線的會話伺服器實例。
2. 訊息接著會被發佈給中介者。
3. 中介者發送訊息給所有訂閱者，也就是會話伺服器的所有實例。
4. 在每個實例裡，訊息則會再發送給所有已連線的客戶端。

Redis 能夠透過**通道**的定義，對發佈及訂閱作出進一步的分流。而所謂的通道，即是由字串定義而成，例如 chat.nodejs。此外也可以使用萬用模式，例如 chat.* 讓定義相符於多個通道。

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (建置 Redis 擔任訊息中介者)

來看看具體的作法，讓我們修改伺服器程式碼，加入發佈/訂閱的邏輯:

```
const WebSocketServer = require('ws').Server;
const redis = require('redis');
const redisSub = redis.createClient(6379, "10.11.25.137");
const redisPub = redis.createClient(6379, "10.11.25.137");

const server = require('http').createServer(
  require('ecstatic')({root: `${__dirname}/www`})
);

const wss = new WebSocketServer({server: server});
wss.on('connection', ws => {
  console.log('Client connected');
  ws.on('message', msg => {
    console.log(`Message: ${msg}`);
    redisPub.publish('demo:chat:messages', msg);
  });
});
```

```
const redisSub = redis.createClient({
  host: "10.11.25.137",
  password: "iscat",
  db: 0
});
```

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (建置 Redis 擔任訊息中介者)

```
redisSub.subscribe('demo:chat:messages');
redisSub.on('message', (channel, msg) => {
  wss.clients.forEach(client => {
    client.send(msg);
  });
});
server.listen(process.argv[2] || 8080);
```

1. 為了讓我們的 Node.js 應用程式能夠連線至 Redis 伺服器，這裡使用了 redis 套件，它是 Redis 的完整客戶端，支援所有的 Redis 命令。接著，實例化兩個連線，一個用於訂閱通道，另一個則用於發佈訊息。這番配置在 Redis 裡是必要的，因為若連線處於訂閱者模式，就只能使用與訂閱者有關的命令。這同時也表示，我們也需要另一個連線來發布訊息。
2. 當從已連線的客戶端接收新訊息時，就在 demo:chat:messages 通道發佈訊息。無須直接廣播訊息給伺服器自身的客戶端，因為伺服器已訂閱相同通道，而訊息將會透過 Redis 再回傳給發佈訊息的伺服器。這是非常簡單又有效的機制，很適用於我們的範例。
3. 由於伺服器也必須訂閱 demo:chat:messages 通道，因此這裡註冊了一個監聽器，接收所有發佈至該通道的訊息(不僅是伺服器自身，也包含了其它的會話伺服器)。當收到訊息時，便廣播給伺服器的所有客戶端。

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (建置 Redis 擔任訊息中介者)

少量的變更就足以整合所有的會話伺服器。為了實際驗證，現在便試著啟動應用程式的多個實例：

```
node app 8080
node app 8081
node app 8082
```

接著我們可以使用多個瀏覽器頁籤連線至各個實例，藉此確認傳送給某個伺服器的訊息，是否也會被不同伺服器的其他客戶端接收。從結果可知，我們已成功利用發佈/訂閱模式，對一個分散式的即時應用程式作出整合。

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (使用 ØMQ 對等式的訊息交換)

使用中介者可以大幅簡化訊息系統架構，不過這在某些情況下也並非最佳的解決方案。舉例來說，若我們非常在意任何可能的速度延遲、或者需要擴增一種極為複雜的分散式系統，甚至是為了避免單點故障問題。那麼，我們就得考慮其它的方案。

  - ØMQ 介紹

如果我們的專案可能需要實作出對等式的訊息交換，那麼首先應評估的解決方案是 ØMQ(又稱為 zmq、ZeroMQ 或 0MQ)。它是一套網路函式庫，提供多種基本工具、可用於建置出各種訊息模式。它提供了低階、高效，且極簡單的 API，能夠用於建置訊息系統所需的一切基礎，例如不可分割的訊息、負載平衡及佇列等。它也支援了多種協定，例如同程序通道(inproc://)、行程間通訊(ipc://)、PGM 協定的多點傳送(pgm:// 或 epgm://)，除此之外，當然也支援了一般的 TCP(tcp://)。

在 ØMQ 的諸多功能中，還包含了可實作出發佈/訂閱模式的工具，而這些工具就是我們所需要的。接下來我們要從現有的程式架構中移除中介者(Redis)，並借助 ØMQ 的發佈/訂閱 socket ，讓各個節點以對等形式通訊。

ØMQ socket 可以想成是一種強化的網路 socket，提供額外的抽象層，來實作出常見的訊息模式。例如發佈/訂閱、請求/回覆，或是單行通訊等等。

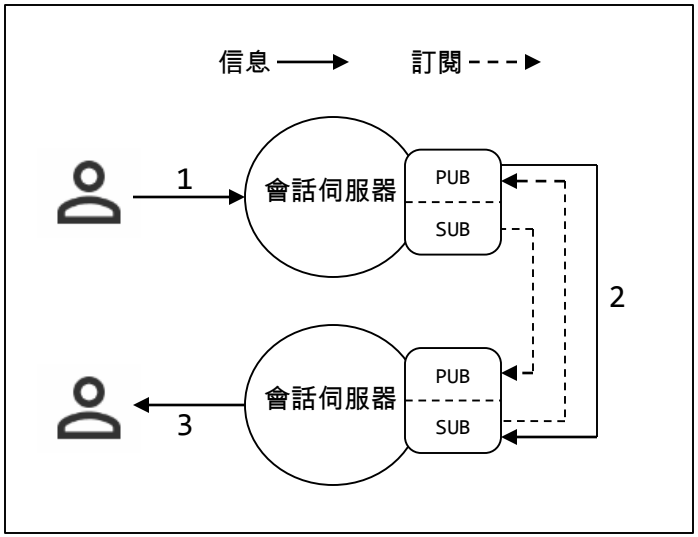
Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (使用 ØMQ 對等式的會話伺服器)

當從現有的架構中移除中介者後，即時通訊程式的每個實例都必須直接連線到其它實例，才能接收到其它實例所發佈的訊息。在 ØMQ 裡，有兩種專用於此目的的 socket: PUB 與 SUB 。典型的應用方式是將 PUB socket 綁定至某個通訊埠，以監聽來自其它實例的 SUB socket 訂閱。

訂閱可以被加以過濾，而過濾器是一個簡單的二進位緩衝區(因此可以是一個字串)，它會比對訊息的開頭(同樣是二進位緩衝區)。當訊息透過 PUB socket 傳送，便會廣播至所有已連線的 SUB socket。



這圖呈現出即時通訊程式的兩個實例，以及當中的資訊流動方式。若存在更多的實例，則也是相同的邏輯概念。在此架構下，每個節點都必須知悉系統裡的其他節點，才能建立所有必要的連線。此外，我們也注意到，當訂閱是從 SUB socket 到 PUB socket 時，則訊息便是反方向傳遞。

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (使用 ØMQ 對等式的會話伺服器)

以下就是使用 ØMQ 的 PUB/SUB socket 的會話伺服器:

```
const WebSocketServer = require('ws').Server;
const args = require('minimist')(process.argv.slice(2));
const zmq = require('zeromq');

const pubSocket = zmq.socket('pub');
pubSocket.bind(`tcp://127.0.0.1:${args['pub']}`);

const subSocket = zmq.socket('sub');
const subPorts = [].concat(args['sub']);
subPorts.forEach( p => {
  console.log(`Subscribing to ${p}`);
  subSocket.connect(`tcp://127.0.0.1:${p}`);
});
subSocket.subscribe('chat');

const server = require('http').createServer(
  require('ecstatic')({root: `${__dirname}/www`})
);
```

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式（使用 ØMQ 對等式的會話伺服器）

```
const wss = new WebSocketServer({server: server});
wss.on('connection', ws => {
  console.log('Client connected');
  ws.on('message', msg => {
    console.log(`Message: ${msg}`);
    broadcast(msg);
    pubSocket.send(`chat ${msg}`);
  });
});

subSocket.on('message', msg => {
  console.log(`From other server: ${msg}`);
  broadcast(msg.toString().replace(/^chat /, ""));
});

function broadcast(msg) {
  wss.clients.forEach(client => {
    client.send(msg);
  });
}
server.listen(args['http'] || 8080);
```

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (使用 ØMQ 對等式的會話伺服器)

很明顯地，上述程式碼開始變得有些複雜了，然而若考量到我們實作的是一個分散又對等的發佈/訂閱模式，那麼整體而言還是算簡單明瞭了。

1. 首先需要 zeromq 套件，它是 Node.js 的 ØMQ 綁定元件。此外還需要 minimist，這是一個命令列參數剖析器，我們需要透過它來輕易地取得參數。
2. 建立 PUB socket，並將這個 socket 綁定於 --pub 命令列參數所提供的通訊埠。
3. 建立 SUB socket，並連線到其它實例的 PUB socket。至於目標 PUB socket 的通訊埠則是由 --sub 命令列提供(可能不只一個)。接著以 chat 作為過濾器來建立訂閱，這表示我們只會接收到以 chat 為開頭的訊息。
4. 當我們的 WebSocket 接收到新訊息時，除了廣播給所有已連線的客戶端外，也透過 PUB socket 發佈出去。同樣以 chat 作為前置文字，再加上一個空白，如此一來該訊息就會發佈給所有以 chat 作為過濾條件的發佈者。
5. 監聽 SUB socket 所傳入的訊息，移除訊息內的 chat 前置文字，然後廣播給 WebSocket 的所有客戶端。

以上，我們便完成了一個簡單的分散式系統，而且是利用對等的發佈/訂閱模式整合而成!

Node.js

- 訊息與整合模式

  - 發佈/訂閱模式 (使用 ØMQ 對等式的會話伺服器)

現在可以逐一啟動這個應用程式的三個實例，並確認它們的 PUB socket 與 SUB socket 皆設定證確:

```
node app --http 8080 --pub 5000 --sub 5001 --sub 5002
node app --http 8081 --pub 5001 --sub 5000 --sub 5002
node app --http 8082 --pub 5002 --sub 5000 --sub 5001
```

上述的第一個命令將啟動一個監聽通訊埠 8080 的 HTTP 伺服器實例，並於通訊埠 5000 綁定 PUB socket，然後將 SUB socket 連線至通訊埠 5001 與 5002，也就是另外兩個實例的 PUB socket 監聽埠。至於第二及第三個命令則是相同的運作概念。

我們首先會注意到，若 SUB socket 的目標通訊埠尚不可用，則 ØMQ 也不會有所抱怨。若描述得更詳細一點，就是在第一個命令執行時，其實還沒有任何實例監聽通訊埠 5001 與 5002，但 ØMQ 卻不會丟出任何錯誤。這是因為 ØMQ 具備重新連線機制，會自動在特定間隔後試著與這些通訊埠建立連線。當有任何節點因故下線或重啟，這項功能就很有用了。相同的容錯邏輯也適用在 PUB socket 上，若無任何訂閱者，便會丟棄所有訊息，但仍持續運作。

現在可以透過瀏覽器檢驗方才所啟動的每一個伺服器實例，確認訊息是否都正確的廣播給所有的伺服器。

Node.js

- 訊息與整合模式

  - 可延續訂閱者 (Durable subscribers)

訊息系統有一個重要的抽象概念是**訊息佇列(message queue: MQ)**。透過訊息佇列，則訊息的傳送者與接收者便無須同時保持接通才能夠彼此通訊，因為佇列系統會先儲存資料，直到目的地能接收它們為止。此種行為模式不同於**設後不理(set and forget)**，這類情形的訂閱者只會在當下是保持連線的狀態才會接收到訊息。

若一個訂閱者使終都能夠可靠的接收到所有訊息，即便它沒有時時刻刻的監聽它們，那麼它就可以被稱為是一個**可延續訂閱者**。

MQTT 通訊協定針對傳送者與接收者之間的訊息交換，定義了一系列的服務品質(Quality of Service: QoS)。這些訂義非常適合用於描述任一訊息系統(不僅是 MQTT)的可靠性，其定義細節如下:

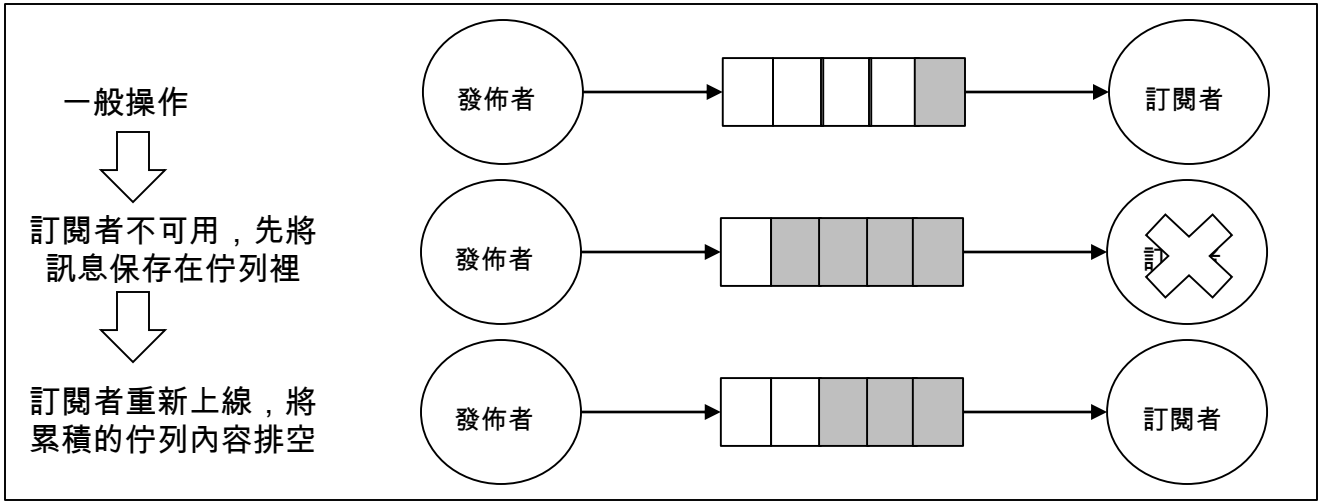
- **Qos0，至多一次:** 也就是「設後不理」，訊息不會保存，而遞送過程也不會加以確認。這表示當接收者崩潰或斷線時，訊息便有可能漏失。
- **Qos1，至少一次:** 確保訊息至少會被成功接收一次，但因此可能會出現重複情形。例如接收者在通知傳送前發生崩潰，而無法回送結果。在此情況下，訊息會被保存起來，然後再次傳送，直到取得確認結果。
- **Qos2，確切一次:** 這是最可靠的 Qos，它會確保訊息被成功接收一次，而且只有一次。然而為了確認訊息的遞送過程是否完整，相對的代價是較慢且繁雜的資料處理機制。

Node.js

- 訊息與整合模式

  - 可延續訂閱者 (Durable subscribers)

如同先前所述，為實現可延續訂閱者，我們的系統必須使用一個訊息佇列，在訂閱者未連線時先保存信息。佇列可暫時儲存在記憶體中，或是更加長久性的儲存在磁碟裡，如此一來，當中介者重啟或崩潰時，則依然能夠復原訊息。下圖呈現出藉由訊息佇列所實現的可延續訂閱者:



實作可延續訂閱者，很可能會是訊息佇列最重要的作用，不過它還有其它作用。先前使用的 Redis 內建的發佈/訂閱命令，是實作了「設後不理」機制(QoS0)。然而 Redis 還是可以透過其他命令的組合(而非直接使用其發佈/訂閱命令)，實作出可延續訂閱者。除此之外，ØMQ 也定義了一些支援可延續訂閱者的機制。但無論如何，具體的實作方式還是操之在我们的手上。

Node.js

- 訊息與整合模式

  - AMQP 介紹

一般來說，訊息佇列會被應用在所有無法接受訊息漏失的環境，也就是極為重要的應用程式，例如銀行業務或財務系統。這通常表示，典型的企業級訊息佇列其實是相當複雜的軟體設計，會使用堅固的協定及可靠的存儲，來確保即便有任何故障都能夠完成訊息的交換。因此企業級的訊息中介軟體多年來一直由 Oracle 與 IBM 這類軟體巨人所獨佔，而它們多半是實做了專有的通訊協定，嚴厲的綁住了客戶的選擇。所幸這幾年來，訊息系統已突破這道障礙而蓬勃發展，這歸功於 AMQP、STOMP 與 MQTT 這類的開放協定。為了瞭解訊息佇列系統的運作方式，以下我們會針對 AMQP 進行概述，藉此初步瞭解如何使用基於這類協定的 API。

AMQP(Advanced Message Queuing Protocol)是一項開放標準的協定，許多訊息佇列系統都支援了這項協定。事實上它除了是通訊協定外，還是一個涉及路由、過濾、佇列、可靠性與安全性的模型。在 AMQP 裡，共有三個基礎元件:

- **佇列(Queue):** 負責儲存訊息的資料架構，裡頭的訊息將由客戶端消耗。佇列訊息會被推送(或拉取)至一或多個消耗者，也就是我們的應用程式。若有多個消耗者皆連接至相同的佇列，則訊息會是負載平衡的。

Node.js

- 訊息與整合模式

  - AMQP 介紹

  佇列可以是以下幾種類型:

  - **可延續性(Durable)**: 意即若中介者重新啟動，則佇列也會自動建立。可延續性佇列並不表示先前的內容一定會被保留，只有被標示為需要保存的訊息，才會存入磁碟，並於重啟時復原。
  - **專用性(Exclusive)**: 意即佇列綁定於特定的訂閱者。若彼此的連線關閉，則佇列就會被銷毀。
  - **自動刪除(Auto-delete)**: 當沒有任何訂閱者連線時，便刪除佇列。

  - **交換(Exchange)** : 訊息發佈之處。依據所實作的演算法，將訊息輸送到一個或多個佇列。
    - **直接交換(Direct exchange)**: 比對整個路由鍵(例如 chat.msg)是否相符來輸送訊息。
    - **主題交換(Topic exchange)**: 使用萬用模式比對路由鍵(例如 chat.# 便吻合所有以 chat 為開頭的路由鍵)。
    - **擴散交換(Fanout exchange)**: 忽略任何的路由鍵，廣播訊息至所有連線的佇列。

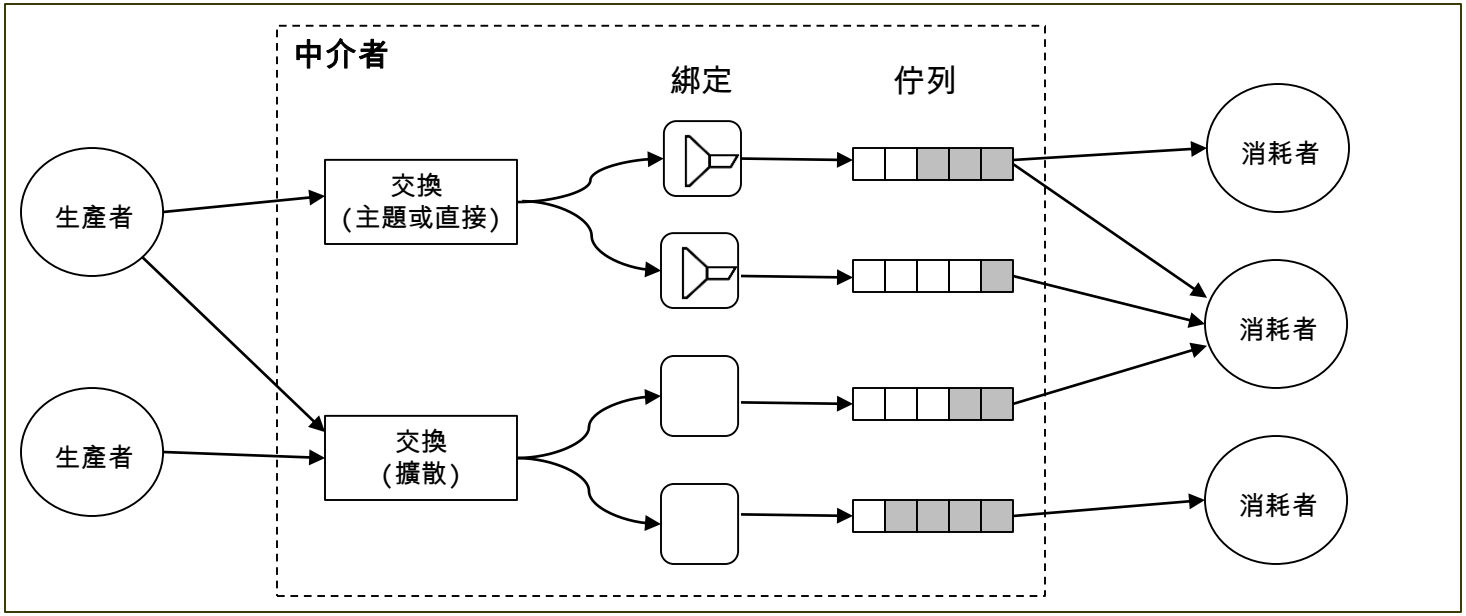
  - **綁定(Binding)**: 交換元件與佇列之間的連結。這裡也定義了路由鍵以過濾來自交換元件的訊息。

Node.js

- 訊息與整合模式

  - AMQP 介紹

以上這些元件是由一個**中介者(broker)**進行管理，它會揭露一個 API，用於相關的建置及處理。當連線到中介者時，客戶端會建立一個抽象化的**通道(channel)**，用於維護與中介者之間的通訊狀態。而在 AMQP 裡，「專用性」或「自動刪除」以外的佇列都可用於實作可延續性訂閱者。



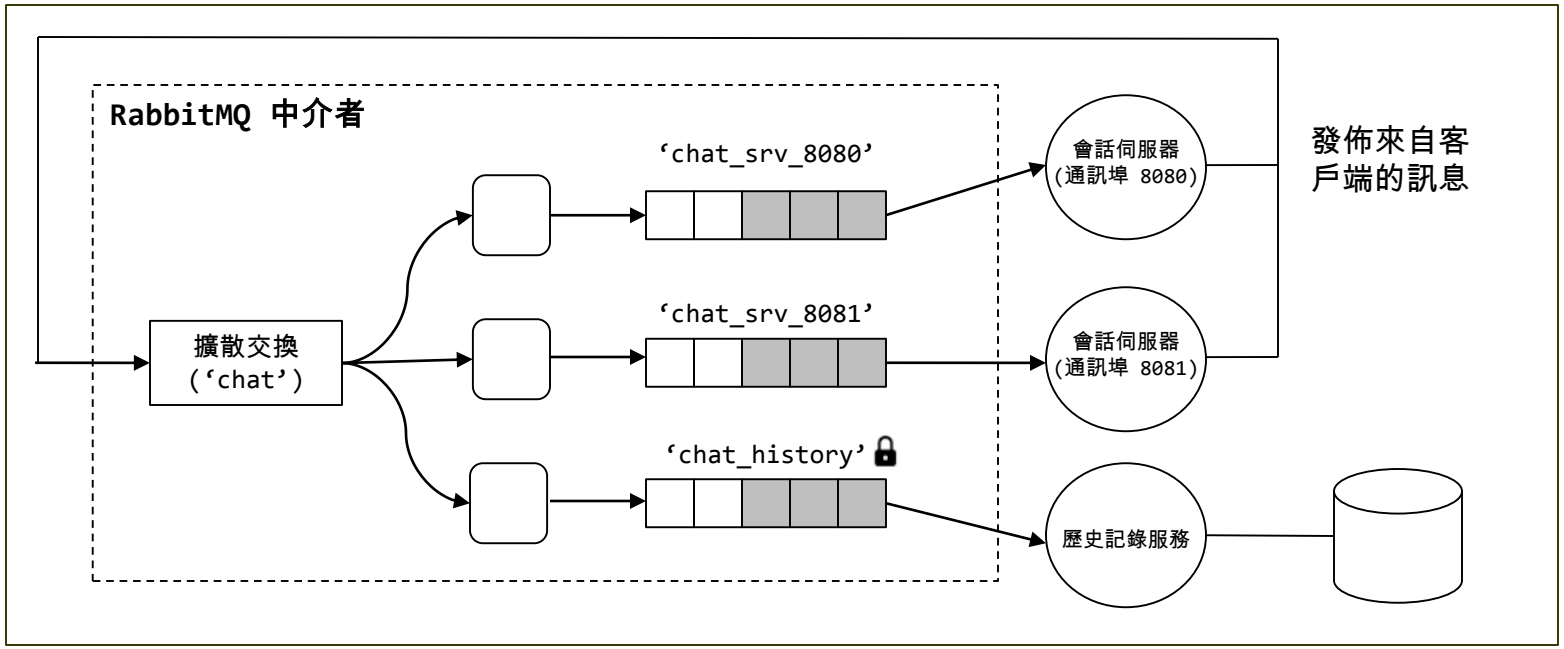
AMQP 模型比我們先前所使用過的任何訊息系統(Redis 與 ØMQ)都還要複雜。但若是因此打退堂鼓，採用簡單的發佈/訂閱機制，則難以實現 AMQP 所能夠提供的功能及可靠性。

Node.js

- 訊息與整合模式

  - 使用 AMOP 及 RabbitMQ 實作可延續訂閱者

當我們想要讓微服務(Microservice)架構中的多個服務保持資料同步時，這便是一種不能漏失掉任何信息的典型狀況。如果想透過一個中介者使所有服務皆保有一致的認知，那麼同時就必須防範任何訊息漏失的機會，否則將會導致狀態的不一致。以下我們要藉由微服務的概念，擴充這個小型的即時通訊程式。我們想要新增一個歷史記錄服務，以取出完整的會話紀錄。我們將使用 RabbitMQ 中介者與 AMQP，來整合歷史記錄服務與會話伺服器。



Node.js

- 訊息與整合模式

  - 使用 AMOP 及 RabbitMQ 實作可延續訂閱者

如圖架構所示，我們將使用擴散交換，沒有任何特殊的路由，也就不會有任何增加複雜性的設計。接著，我們將為會話伺服器的各個實例建立佇列。這些佇列是「專用性」的，因為我們不打算在此處裡任何離線訊息，這將是歷史記錄服務的工作。如果你有認真領會先前的說明，應該會注意到這表示我們的會話伺服器並非可延續訂閱者。因此一旦連線關閉，它們的佇列就會被銷毀。

反之，歷史記錄服務可不能漏失任何訊息，否則就無法體現它的存在目的，因此我們要為它建立的佇列必須是可延續的。若信息發佈時，歷史記錄服務是處於斷線狀態，則訊息將會保留在佇列裡，並於歷史記錄服務重新上線後遞送。

我們將利用 LevelUP 作為歷史服務的儲存引擎，此外也會使用 amqplib 套件，以 AMQP 通訊協定連線至 RabbitMQ。

Node.js

- 訊息與整合模式

  - 使用 AMOP 及 RabbitMQ 實作可延續訂閱者

現在便開始實作這個歷史記錄服務，我們將建立一個獨立的應用程式(其實就是一個典型的微服務) historySvc.js。這個模組分成兩個部份: 一個 HTTP 伺服器，用於揭露會話紀錄給客戶端；以及一個 AMQP 消耗者，負責擷取會話訊息，並將它們儲存在本地資料庫裡。

```
const level = require('level');
const timestamp = require('monotonic-timestamp');
const JSONStream = require('JSONStream');
const amqp = require('amqplib');
const db = level('./msgHistory');

require('http').createServer((req, res) => {
  res.writeHead(200);
  db.createValueStream()
    .pipe(JSONStream.stringify())
    .pipe(res);
}).listen(8090);
```

Node.js

- 訊息與整合模式
  - 使用 AMOP 及 RabbitMQ 實作可延續訂閱者

```
let channel, queue;
amqp.connect('amqp://10.11.25.137')
  .then(conn => conn.createChannel())
  .then(ch => {
    channel = ch;
    return channel.assertExchange('chat', 'fanout');
  })
  .then(() => channel.assertQueue('chat_history'))
  .then(q => {
    queue = q.queue;
    return channel.bindQueue(queue, 'chat');
  })
  .then(() => {
    return channel.consume(queue, msg => {
      const content = msg.content.toString();
      console.log(`Saving message: ${content}`);
      db.put(timestamp(), content, err => {
        if (!err) channel.ack(msg);
      });
    });
  }).catch(err => console.log(err));
```

Node.js

- 訊息與整合模式

  - 使用 AMQP 及 RabbitMQ 實作可延續訂閱者

很明顯就會看到 AMQP 需要一些設定過程，這是為了建立並連接此模型的所有元件。此外可以發覺到 amqplib 已支援承諾(Promises)功能，所以我們會充分利用，來流線化應用程式的非同步過程。以下就讓我們來深入瞭解這個應用程式的運作方式:

1. 我們首先使用一個 AMQP 中介者(RabbitMQ)建立連線。接著建立一個類似於工作階段(session)的通道，利用它來維護通訊狀態。
2. 接著，建立一個名為 chat 的交換(exchange)，如同先前曾提及的，這是一個 fanout(擴散)交換。assertExchange() 命令會確認該交換是否已存在於中介者中，若無便會加以建立。
3. 建立名為 chat_history 的佇列。由於佇列預設就是可延續的(durable)，除非被設為 exclusive(專用性)或 auto-delete(自動刪除)，所以我們不需在此指定任何額外選項。
4. 對佇列及交換作綁定。不用在此加上路由鍵這類的選項，因為先前的交換已經指定為擴散式，所以不會再執行任何過濾。
5. 最後，開始監聽來自佇列的訊息，將接收到的每個訊息儲存在 LevelDB 資料庫裡，並使用單向時間標記作為鍵值，讓訊息可依時間日期排序。其中應特別注意到，我們使用了 channel.ack(msg) 來確認每個訊息，而且只在訊息成功存入資料庫後才做出確認。如果中介者未收到 ACK(確認)，則訊息就會被保留在佇列裡以待再次處理。這是 AMQP 的另一項重大功能，能夠將服務的可靠性帶入全新境界。不過，如果不想明確做出確認，則可以傳入 {noAck:true} 選項給 channel.consume() API。

Node.js

- 訊息與整合模式

  - 使用 AMOP 及 RabbitMQ 實作可延續訂閱者

接著，我們將使用 AMQP 整合即時通訊程式，在會話伺服器裡使用 AMQP 進行整合，非常類似於我們在歷史服務裡的作法，我們只需要特別關注在於佇列是如何建立的，以及新訊息是如何發佈到交換(exchange)中的。以下就是 app.js:

```
const WebSocketServer = require('ws').Server;
const amqp = require('amqplib');
const JSONStream = require('JSONStream');
const request = require('request');
let httpPort = process.argv[2] || 8080;

const server = require('http').createServer(
  require('ecstatic')({root: `${__dirname}/www`})
);
```

Node.js

- 訊息與整合模式
  - 使用 AMOP 及 RabbitMQ 實作可延續訂閱者

```
let channel, queue;
amqp
  .connect('amqp://10.11.25.137')
  .then(conn => conn.createChannel())
  .then(ch => {
    channel = ch;
    return channel.assertExchange('chat', 'fanout');
  })
  .then(() => {
    return channel.assertQueue(`chat_srv_${httpPort}`, {exclusive: true});
  })
  .then(q => {
    queue = q.queue;
    return channel.bindQueue(queue, 'chat');
  })
  .then(() => {
    return channel.consume(queue, msg => {
      msg = msg.content.toString();
      console.log(`From queue ${msg}`);
      broadcast(msg);
    }, {noAck: true});
  })
  .catch(err => console.log(err));
```

Node.js

- 訊息與整合模式
  - 使用 AMOP 及 RabbitMQ 實作可延續訂閱者

```
const wss = new WebSocketServer({server: server});
wss.on('connection', ws => {
  console.log('Client connected');
  request('http://localhost:8090')
  .on('error', err => console.log(err))
  .pipe(JSONStream.parse('*'))
  .on('data', msg => ws.send(msg));

  ws.on('message', msg => {
    console.log(`Message: ${msg}`);
    channel.publish('chat', '', new Buffer(msg));
  });
});

function broadcast(msg) {
  wss.clients.forEach(client => client.send(msg));
}

server.listen(httpPort);
```

Node.js

- 訊息與整合模式

  - 使用 AMOP 及 RabbitMQ 實作可延續訂閱者

如同先前曾提及的，這個會話伺服器並不需要可延續訂閱者，因此「設後不理」的作法便已足夠。所以在建立佇列時，會傳入 {exclusive:true} 選項，指示該佇列的負責範圍僅限於當前的連線，並且會在伺服器關閉時一併銷毀。

信息的發佈非常簡單，只需要指定交換目標(chat)及路由鍵。由於我們是使用擴散交換，所以路由鍵的欄位為空(‘’)。

現在可以執行這個改良後的即時通信架構，讓我們啟動兩個會話伺服器，以及一個歷史記錄服務:

```
node app 8080
node app 8081
node historySvc
```

現在檢驗一下我們的系統，尤其是歷史記錄服務的部份。如果先中止歷史記錄服務，並且繼續透過即時通訊程式的網頁介面傳送訊息，將會看到當服務重新啟動後，便會收到先前所有漏失的訊息，充分展現出可延續訂閱者的作用。微服務的設計概念能夠讓系統即便少了某個元件(例如歷史紀錄服務)，其整體卻仍然能夠生存下來。它會暫時短少某些功能(沒有會話紀錄)，但仍然可以進行即時通訊，保障了應用程式的生命力!

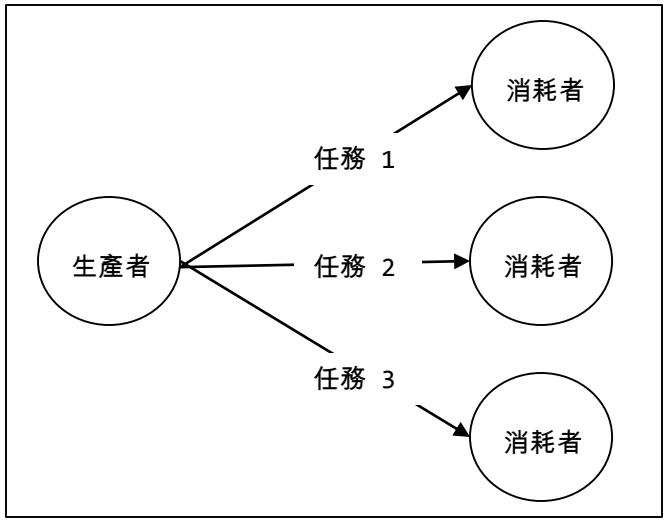
Node.js

- 管線與任務分散模式

前面我們曾學到如何將高成本的任務使用 child_process.fork() 分派給多個本地程序。雖然效能有所提升，但此舉並沒有將擴展範圍突破到單部機器之外。所以我們現在來看看如何在分散式架構下，運用網路上的多個遠端工作者。

其背後的構想是建立一個可分派任務至多部機器的訊息模式。這些任務可能是獨立性的，也可能是透過「分治法」切分的大行任務片段。

若檢視下圖的邏輯結構，應該可以識別出一種似曾相識的模式:

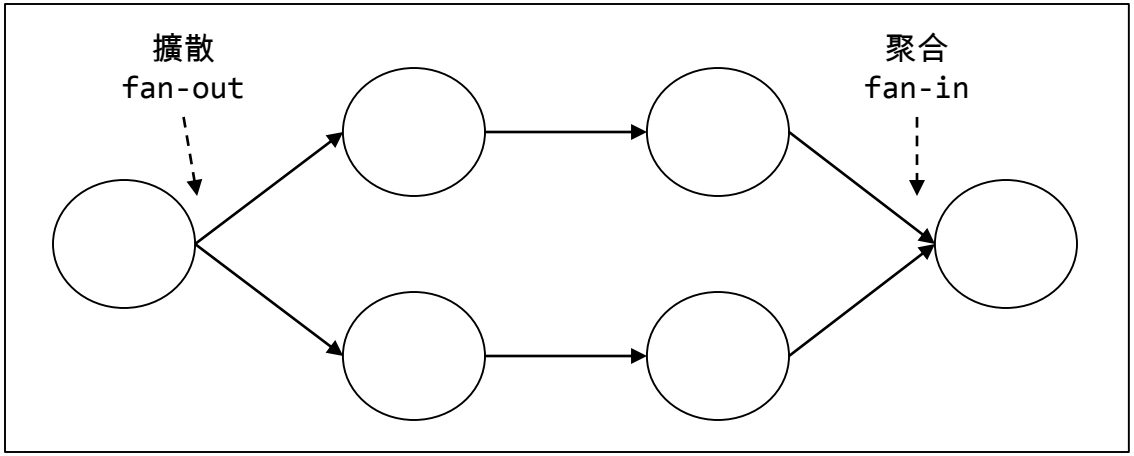


如左圖所見，發佈/訂閱模式並不適用於這類應用程式，因為我們絕對不希望多個工作者接收到同一項任務。相反的，我們需要一個類似負載平衡器的訊息分散模式，將各個訊息分派給不同的消耗者(在此例中亦稱為工作者)。在訊息系統的術語裡，此模式被稱為**競爭消耗者(competing consumers)**、**擴散分散(fanout distribution)**，或**吹風機(ventilator)**。

Node.js

- 管線與任務分散模式

不同於前面描述的 HTTP 負載平衡器，這裡的消耗者是更加主動的角色。多數時候並非生產者去連接消耗者，而是消耗者去連接生產者，以便接收新任務。這項特性在可擴展系統中是相當重要的優勢，讓我們可以輕鬆的增加工作者數目，而無須修改生產者或導入服務註冊表。此外，在一般的訊息系統裡，我們並不需要在生產者及工作者之間使用請求/回覆通訊。多數時候，更適當的作法是使用單行的非同步通訊，藉此獲得更好的平行性與擴展性。在這類架構下，訊息可能一直都是單向前進的**管線**(pipelines)，如下圖所示:



管線能夠讓我們建置出極為複雜的處理架構，卻沒有同步請求/回覆通訊所引致的負擔，也就能夠產生較低的延遲及較高的產出。在上圖中，我們可以看到訊息如何擴散到多個工作者，並且進一步轉遞給後續的處理單元，最中再聚合到單一節點，這個節點通常被相應地稱為**收風口**(**sink**)。

Node.js

- 管線與任務分散模式

我們將以此架構為基礎，實作對等式及以基於中介者的方式，而這類管線與任務散模式的結合，亦稱為**平行管道(parallel pipeline)**。

- ØMQ 擴散/聚合模式(fanout/fanin)

先前我們已看到 ØMQ 的一些能力，能夠用於建置對等式的分散架構，例如我們曾使用 PUB socket 及 SUB socket 散播一則訊息給多個消耗者。不過現在我們來看看另一組名為 PUSH 與 PULL 的 socket，藉此建置出平行管線。

Node.js

- 管線與任務分散模式

  - PUSH/PULL socket

  從字面上來看，我們很容易可以判定，PUSH socket 勢必很適合用於傳送訊息，而相對地 PULL socket 則是用於接收訊息。這個組合似乎簡單易懂，不過它們其實還有一些更細緻的特性，能夠完美應用在單行通訊系統的建置:

  - 兩者皆可在「連線」模式(connect mode)或「綁定」模式(bind mode)下運作。這表示可以建置一個 PUSH socket，綁定一個本地通訊埠，用於監聽來自 PULL socket 的連線；或者反過來，PULL socket 可以監聽來自 PUSH socket 的連線。雖然訊息永遠是從 PUSH 傳送到 PULL，但連線的發起端是可以變更的。綁定模式是「**可延續(durable)**」節點的最佳解決方案，例如任務生產者與收風口;反之連線模式(connect mode)則是「浮動」節點的最佳方案，例如任務工作者。如此一來，浮動式的節點可以任意地變化，而不會影響到可延續節點。
  - 若有多個 PULL socket 連線至同一個 PUSH socket，訊息將會平均分散給所有的 PULL socket，具體來說，這就是一種負載平衡(對等式的負載平衡)。另一方面，從多個 PUSH socket 接收訊息的 PULL socket，會使用**公平佇列**系統，循環式的處理傳入訊息，以實現「公平」的消耗過程。
  - 若透過 PUSH socket 發送訊息的同時，尚無任何已連線的 PULL socket，則訊息將不會漏失，而是會置入佇列，待節點上線並能夠拉取訊息時發送。

Node.js

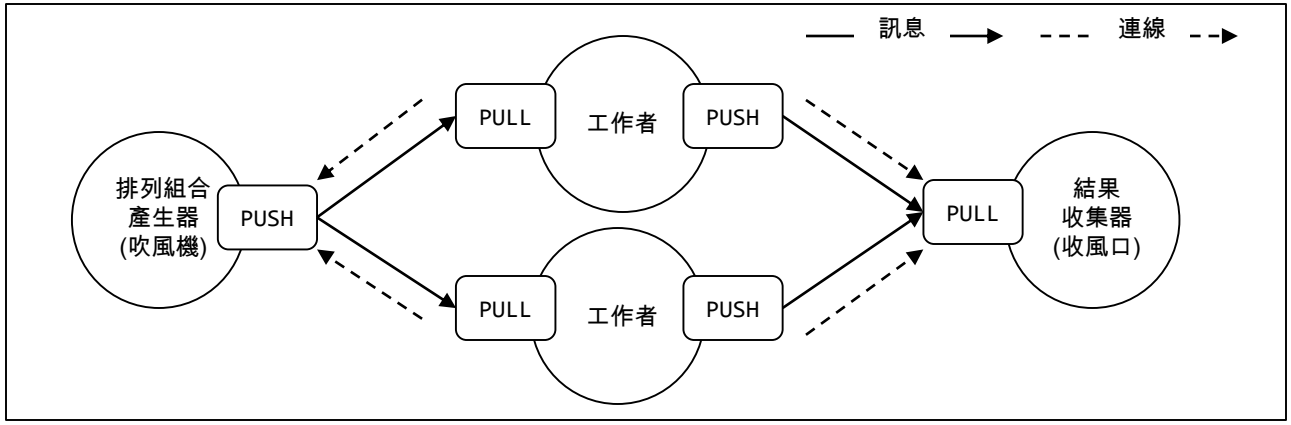
- 管線與任務分散模式

  - 使用 ØMQ 建置一個分散式的雜湊碼破解器

以下我們就要來建置一個範例應用程式，藉此瞭解前面所敘述的 PUSH/PULL socket 運作特性。

這個範例應用程式是一個「雜湊碼破解器」，能夠根據一組字母做出各種排列組合，藉此對輸入的雜湊碼(MD5、SHA1 等等)進行暴力破解。由於這可以說是一種「無障礙平行」的工作負載，所以自然就成了示範平行管線的完美選擇。

這個應用程式會實作一個典型的平行管線，其中有一個節點是用於建立任務並分派給多個工作者，以及一個節點是用於收集任務的執行結果。這個系統可以使用 ØMQ 實作，採如下的設計架構：



Node.js

- 管線與任務分散模式

  - 使用 ØMQ 建置一個分散式的雜湊碼破解器

在這個架構中，吹風機是用於產生字母表的所有排列組合，並將它們分派給多個工作者。工作者會在收到後計算出雜湊碼，然後試著與輸入的雜湊碼進行比對。若比對相符，便傳送結果至收集器節點(收風口)。

此架構的可延續節點是吹風機與收風口，而浮動節點則是工作者。意即每個工作者都是使用 PULL socket 連線至吹風機，並使用 PUSH socket 連線至收風口。這表示我們無須變更吹風機與收風口的任何參數，就能夠任意啟動及停止供作者。

以下就讓我們建立吹風機模組，ventilator.js:

```
const zmq = require('zeromq');
const variationsStream = require('variations-stream');
const alphabet = 'abcdefghijklmnopqrstuvwxyz';
const batchSize = 10000;
const maxLength = process.argv[2];
const searchHash = process.argv[3];
```

Node.js

- 管線與任務分散模式
  - 使用 ØMQ 建置一個分散式的雜湊碼破解器

```
const ventilator = zmq.socket('push');
ventilator.bindSync("tcp://*:5016");

let batch = [];
variationsStream(alphabet, maxLength)
  .on('data', combination => {
    batch.push(combination);
    if (batch.length === batchSize) {
      const msg = {searchHash: searchHash, variations: batch};
      ventilator.send(JSON.stringify(msg));
      batch = [];
    }
  })
  .on('end', () => {
    //send remaining combinations
    const msg = {searchHash: searchHash, variations: batch};
    ventilator.send(JSON.stringify(msg));
  });
```

Node.js

- 管線與任務分散模式

  - 使用 ØMQ 建置一個分散式的雜湊碼破解器

為避免產生太多組合，我們的產生器只使用了全小寫的英文字母，並限制了字母組合的長度。這項限制是透過命令列參數(maxLength)設定，至於欲破解的雜湊碼則同樣也是來自於命令列參數(searchHash)。此外我們使用了一個名為 varitions-stream 的函式庫，以串流介面來產生所有組合。

不過最重要的部份是如何分派任務給多個工作者:

1. 首先建立一個 PUSH socket，綁定於本地通訊埠 5016，讓工作者的 PULL socket 能夠與之連線並接收任務。

2. 收集已產生的組合，當達到 10,000 項時建立一個訊息(任務物件)，內含這些項目以及欲破解的雜湊碼。接著呼叫 ventilator.send()，以循環式(round robin)的分配方式，傳送訊息給下一個可用的工作者。

Node.js

- 管線與任務分散模式

  - 使用 ØMQ 建置一個分散式的雜湊碼破解器

接著是實作工作者(worker.js):

```
const zmq = require('zeromq');
const crypto = require('crypto');
const fromVentilator = zmq.socket('pull');
const toSink = zmq.socket('push');

fromVentilator.connect('tcp://localhost:5016');
toSink.connect('tcp://localhost:5017');
```

Node.js

- 管線與任務分散模式
  - 使用 ØMQ 建置一個分散式的雜湊碼破解器

```
fromVentilator.on('message', buffer => {
  const msg = JSON.parse(buffer);
  const variations = msg.variations;

  variations.forEach( word => {
    console.log(`Processing: ${word}`);
    const shasum = crypto.createHash('sha1');
    shasum.update(word);
    const digest = shasum.digest('hex');
    if (digest === msg.searchHash) {
      console.log(`Found! => ${word}`);
      toSink.send(`Found! ${digest} => ${word}`);
    }
  });
});
```

Node.js

- 管線與任務分散模式

  - 使用 ØMQ 建置一個分散式的雜湊碼破解器

  先前曾提到，工作者是此架構裡的浮動節點，因此工作者的 socket 應該連線至遠端節點，而不是監聽傳入的連線。根據上述原理，工作者 socket 的設計如下：

  - PULL socket 連線至吹風機，接收任務
  - PUSH socket 連線至收風口，傳送結果

  除此之外，這個工作者的工作非常簡單：針對每一個接收到的訊息，迭代裡頭的所有組合，逐一計算出它們的 SHA1 校驗碼，然後與 searchHash 進行比對。若比對相符，便將結果轉遞至收風口。

Node.js

- 管線與任務分散模式

  - 使用 ØMQ 建置一個分散式的雜湊碼破解器

這個範例的收風口是一個非常簡單的結果收集器，僅僅是將工作者所傳送的訊息列印在終端機上。sink.js 的內容如下:

```
const zmq  = require('zeromq');
const sink = zmq.socket('pull');
sink.bindSync("tcp://*:5017");

sink.on('message', buffer => {
  console.log('Message from worker: ', buffer.toString());
});
```

收風口也是此架構下的可延續節點，因此綁定其 PULL socket，而非主動連線至工作者的 PUSH socket。

Node.js

- 管線與任務分散模式

  - 使用 ØMQ 建置一個分散式的雜湊碼破解器

現在就準備來執行這個應用程式，讓我們先分別啟動工作者及收風口：

```
node worker
node worker
node sink
```

```
const crypto = require('crypto');  // 產生 SHA1 校驗碼

const hash = crypto.createHash('sha1')
                   .update('love')
                   .digest('hex');
console.log(hash);
```

接著便啟動吹風機，輸入組合的最大長度，以及欲破解的 SHA1 校驗碼。以下即為執行範例：

```
node ventilator 4 9f2feb0f1ef425b292f2f94bc8482494df430413
```

當上述命令執行時，吹風機會以四個字元的長度，產生所有可能的組合，再連同輸入的校驗碼分派給先前啟動的工作者。若成功破解，便會顯示結果在終端機上。你可在破解之前，試試看再增加一個工作者，它將會動態的加入破解工作者的行列！

```
Message from worker:  Found! 9f2feb0f1ef425b292f2f94bc8482494df430413 => love
```

Node.js

- 管線與任務分散模式

  - AMQP 的管線與競爭消耗者(competing consumers)

前面的展示是關於平行管線在對等式環境(peer-to-peer context)下的實作方式，而接下來我們要探索，如何將此模式應用在 RabbitMQ 這類功能成熟的訊息中介者上。

  - 對點式(point-to-point)通訊與競爭消耗者

在對等式(peer-to-peer)設計中，管線可以說是相當直覺的概念。然而若加上一個訊息中介者，則各個節點之間的關係就會變得有一點複雜。由於中介者是作為通訊的中間協調者，以至於來源節點與目的節點之間，經常對彼此所之甚少。舉例來說，當我們使用 AMQP 傳送一則訊息，並不會直接傳送到目的地，而是先傳遞給交換元件(exchange)，然後是佇列元件(queue)。最後，中介者會根據交換、綁定及目的地佇列所定義的規則，決定訊息的最終去向。

如果我們想要使用 AMQP 這類的系統，實作管線及任務分散模式，就必須確保每個訊息都只會有一個消耗者。但若交換元件綁定於多個佇列，這就會是難以確保的。上述問題的解決方案是直接傳送訊息至目的地佇列，完全繞過交換，如此一來就可以確保只有一個佇列會收到該訊息。此種通訊模式被稱為是**對點式(point-to-point)**。

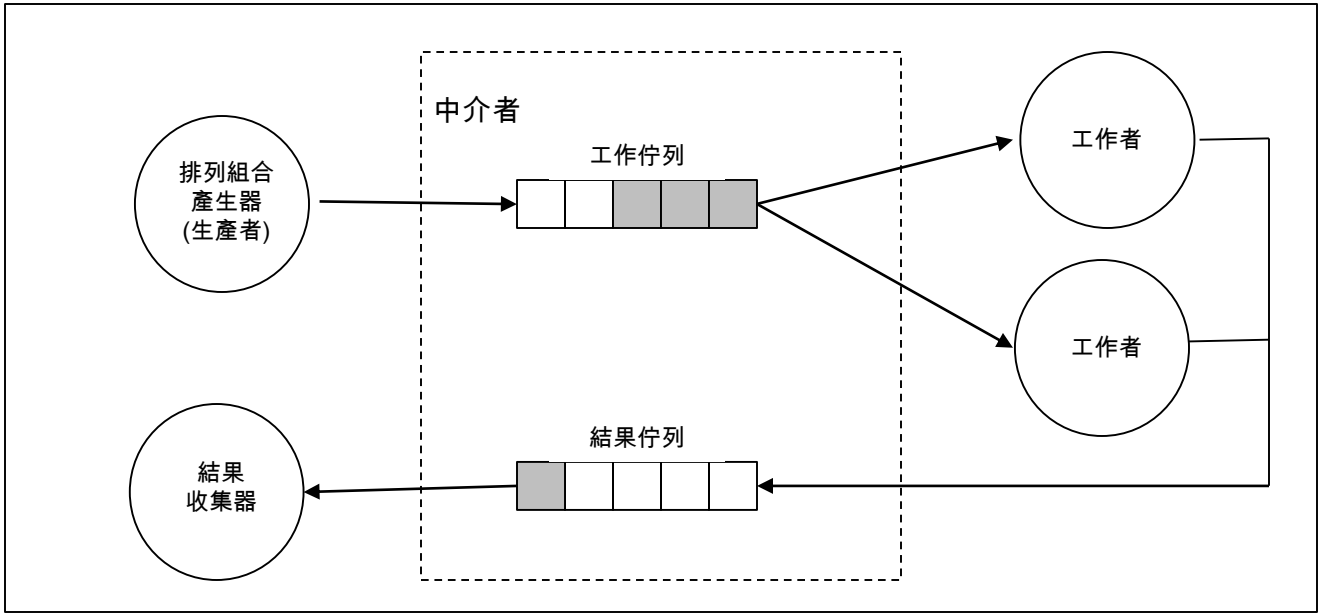
Node.js

- 管線與任務分散模式

  - AMQP 的管線與競爭消耗者(competing consumers)

當我們能夠直接傳送多個訊息至單一佇列，距離任務分散模式的實現就不遠了。事實上，接下來的工程就是，讓監聽相同佇列的多個消耗者之間，能夠平均分配訊息，以實作擴散(fan-out)分散。就訊息中介者的環境而言，此舉可以被稱為是**競爭消耗者**模式。

  - 使用 AMQP 實作雜湊碼破解器



Node.js

- 管線與任務分散模式

  - 使用 AMQP 實作雜湊碼破解器

如同先前曾提及的，我們需要使用單一佇列，將任務分派給多個工作者。我們在前圖稱其為「工作佇列」，在工作佇列的另一端，是多個工作者，也就是「競爭消耗者」。每一個工作者都會從佇列取得不同的訊息，藉此讓多項任務能夠在不同工作者之間平行執行。

工作者所產生的結果，就會被發佈到另一個佇列，我們稱其為「結果佇列」，接著由結果收集器消耗掉。這其實就等同於收風口或聚合。在此架構中，我們沒有使用到任何交換元件，而是將訊息直接傳送到目的地佇列，實作出對點式(point-to-point)的通訊。

我們就從生產者開始(排列組合產生器)。這裡的程式碼幾乎相同於先前的範例，除了關於訊息交換的部份。producer.js 的內容如下:

```
const amqp = require('amqplib');
const variationsStream = require('variations-stream');
const alphabet = 'abcdefghijklmnopqrstuvwxyz';
const batchSize = 10000;
const maxLength = process.argv[2];
const searchHash = process.argv[3];
const q = 'demo_jobs_queue';
```

Node.js

- 管線與任務分散模式
  - 使用 AMQP 實作雜湊碼破解器

```
let channel;


amqp
  .connect('amqp://10.11.25.137')
  .then(conn => {
    return conn.createChannel();
  })
  .then(ch => {
    channel = ch;
    produce();
  })
  .catch(err => console.log(err));
```

Node.js

- 管線與任務分散模式
  - 使用 AMQP 實作雜湊碼破解器

```
function produce() {
  let batch = [];
  variationsStream(alphabet, maxLength)
    .on('data', combination => {
      batch.push(combination);
      if (batch.length === batchSize) {
        const msg = {searchHash: searchHash, variations: batch};
        channel.sendToQueue(q, new Buffer(JSON.stringify(msg)));
        batch = [];
      }
    })
    .on('end', () => {
      //send remaining combinations
      const msg = {searchHash: searchHash, variations: batch};
      channel.sendToQueue(
        q,
        new Buffer(JSON.stringify(msg))
      );
      setTimeout(function () {
        channel.connection.close();
      }, 500);
    });
}
```

Node.js

- 管線與任務分散模式

  - 使用 AMQP 實作雜湊碼破解器

這裡沒有任何交換或綁定，使得 AMQP 的通訊設定變得非常簡單。甚至連佇列都沒有，因為我們只專注於訊息的發佈。

不過最重要的細節是 channel.sendToQueue() API，因為我們先前從未使用過它。如其名稱所示，它能夠直接遞送訊息給佇列(demo_jobs_queue)，繞過所有交換或路由。

在 demo_jobs_queue 的另一端，需要工作者監聽傳入的任務。worker.js 是我們實作的工作者:

```
const amqp = require('amqplib');
const crypto = require('crypto');
```

Node.js

- 管線與任務分散模式
  - 使用 AMQP 實作雜湊碼破解器

```
let channel, queue;
amqp
  .connect('amqp://10.11.25.137')
  .then(conn => conn.createChannel())
  .then(ch => {
    channel = ch;
    return channel.assertQueue('demo_jobs_queue');
  })
  .then(q => {
    queue = q.queue;
    consume();
  })
  .catch(err => console.log(err.stack));
```

Node.js

- 管線與任務分散模式
  - 使用 AMQP 實作雜湊碼破解器

```
function consume() {
  channel.consume(queue, function(msg) {
    const data = JSON.parse(msg.content.toString());
    const variations = data.variations;
    console.log('Processing...');
    variations.forEach( word => {
      const shasum = crypto.createHash('sha1');
      shasum.update(word);
      const digest = shasum.digest('hex');
      if (digest === data.searchHash) {
        console.log(`Found! => ${word}`);
        channel.sendToQueue('demo_results_queue',
          new Buffer(`Found! ${digest} => ${word}`));
      }
    });
    channel.ack(msg);
  });
}
```

Node.js

- 管線與任務分散模式

  - 使用 AMQP 實作雜湊碼破解器

這個工作者非常類似於先前使用 ØMQ 實作的工作者，除了關於訊息交換的部份。在前述程式碼中，可以看到我們先用 channel.assertQueue() 確認 demo_jobs_queue 是否存在，再開始使用 channel.consume() 監聽傳入的任務。接著，若比對相符，便經由 demo_results_queue 傳送結果給收集器，同樣是利用對點式的通訊。

若啟動多個工作者，則都會監聽相同的佇列，使訊息能夠在它們之間作負載平衡。

最後就是實作結果收集器，它也是一個非常簡單的模組，僅僅是將接收到的訊息列印到終端機上。collector.js 的實作如下：

Node.js

- 管線與任務分散模式
  - 使用 AMQP 實作雜湊碼破解器

```
const amqp = require('amqplib');

let channel, queue;
amqp
  .connect('amqp://10.11.25.137')
  .then(conn => conn.createChannel())
  .then(ch => {
    channel = ch;
    return channel.assertQueue('demo_results_queue');
  })
  .then(q => {
    queue = q.queue;
    channel.consume(queue, msg => {
      console.log('Message from worker: ', msg.content.toString());
    });
  })
  .catch(err => console.log(err.stack));
```

Node.js

- 管線與任務分散模式

  - 使用 AMQP 實作雜湊碼破解器

現在這個新系統皆已準備就緒，我們可以先執行兩個工作者，這兩個工作者都會連接至相同得佇列(demo_jobs_queue)，因此產生的訊息會在當中作負載平衡:

```
node worker
node worker
```

接著，我們可以執行 collector 模組，並接著執行 producer。別忘了輸入組合的最大長度，以及欲破解的校驗碼:

```
node collector
node producer 5 47e68180813c48be2408b98f5577fb058975820e
```

以上，我們僅僅使用 AMQP 便實作出訊息管線及競爭消耗模式。

Node.js

- 請求/回覆模式(Request/Reply patterns)

訊息系統的處理經常是使用單行非同步通訊，發佈/訂閱模式就是一項完美的範例。

單行通訊就平行性及效率而言，存在相當大的優勢。然而若只有單行通訊，並無法解決所有的整合及通訊問題。有些時候，老派的請求/回覆模式才是更好的解決方案。因此，我們應該要學習，如何在全是非同步單行通道的情況下，建置出一個抽象層，好讓我們能夠使用請求/回覆模式來交換訊息，而這便是接下來的重點。

- 關聯識別子(Correlation identifier)

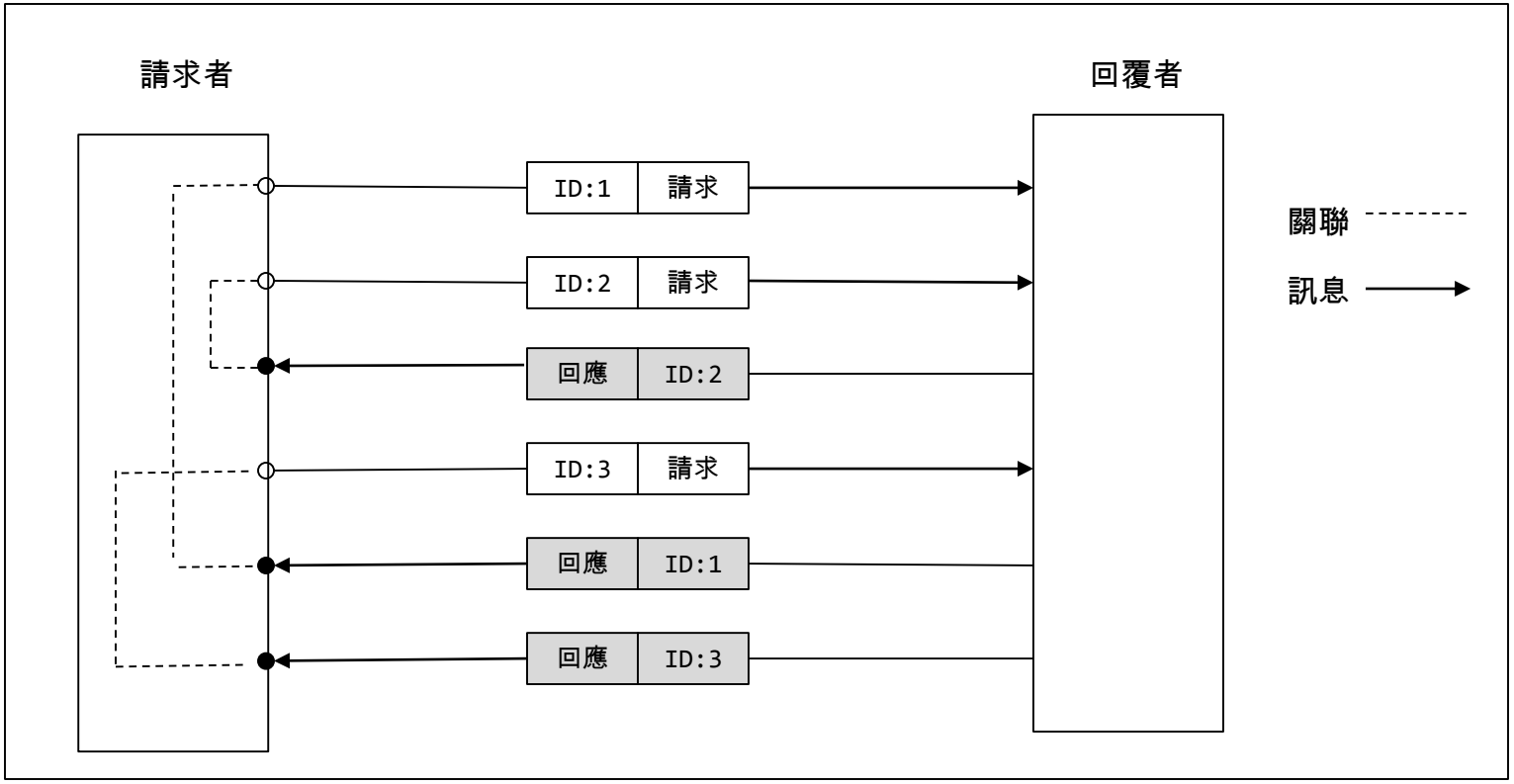
我們將要學習的第一種請求/回覆模式是關聯識別子，這個名稱同時也是指涉及請求/回覆抽象層的基本元素。

此模式會將各個請求標上一個識別子，而這個識別子隨後也會被接收者附加到回應裡。如此一來，請求的傳送者便能夠將這兩個訊息關聯起來，讓回應可以被傳遞給正確的處置者。此舉優雅的解決了單行非同步通道的問題，使訊息可隨時穿梭於不同的方向。

Node.js

- 請求/回覆模式(Request/Reply patterns)

讓我們看看下圖範例:



上圖情境展示出，關聯識別子能夠讓我們做出正確的請求/回應配對，即使它們的傳送及接收順序不一也沒有關係。

Node.js

- 請求/回覆模式(Request/Reply patterns)

  - 利用關聯識別子實作請求/回覆抽象層

接下來我們將針對一種最簡單的單行通道型態(one-way)進行示範，也就是對點式(兩個節點之間直接做連接 point-to-point)及訊息可以變換方向的單行通道。

關於簡單通道的一項例子是 WebSocket，它會在伺服器及瀏覽器之間建立對點式的連線，而訊息能夠以任意方向傳遞。至於其它例子還包含了 child_process.fork() 生成子程序時所建立的通信道。這兩個通道也是非同步的，它連接了子程序與父程序，且訊息能夠以任何方向傳遞。由於這應該是我們所能想到最簡單的通道型態，所以接下來的範例將以此為基礎。

我們準備要對一個應用程式建置抽象層，來包裹父程序與子程序之通道。這個抽象層會自動對每個請求標上關聯識別子，藉此比對傳入的回覆，再傳遞給相符的請求處置器，以實現請求/回覆的通訊模式。

在之前，我們已知父程序可以透過兩個基本函式，與子程序建立溝通管道:

- child.send(message)
- child.on('message', callback)

Node.js

- 請求/回覆模式(Request/Reply patterns)

  - 利用關聯識別子實作請求/回覆抽象層

  同樣地，子程序也可以使用下列基本函式與父程序溝通:

  - process.send(message)
  - process.on('message', callback)

  這表示父程序與子程序的通道介面都是相同的，因此我們可以建置出一個通道的抽象層，使通道的兩端接可傳送資料。

Node.js

- 請求/回覆模式(Request/Reply patterns)
  - 利用關聯識別子實作請求/回覆抽象層

以下就是這個抽象層，request.js:

```
const uuid = require('node-uuid');

module.exports = function(channel) {
  const idToCallbackMap = {};

  channel.on('message', message => {
    const handler = idToCallbackMap[message.inReplyTo];
    if (handler) {
      handler(message.data);
    }
  });
```

Node.js

- 請求/回覆模式(Request/Reply patterns)
  - 利用關聯識別子實作請求/回覆抽象層

```
 return function sendRequest(req, callback) {
    const correlationId = uuid.v4();
    idToCallbackMap[correlationId] = callback;
    channel.send({
      type: 'request',
      data: req,
      id: correlationId
    });
  };
};
```

Node.js

- 請求/回覆模式(Request/Reply patterns)

  - 利用關聯識別子實作請求/回覆抽象層

  這個請求抽象層的運作方式如下:

  1. 我們為請求函式建立一個閉包，而關鍵之處在於 idToCallbackMap 變數，用於儲存請求與回覆之間的對應關係。
  2. 在呼叫工廠時，首先會開始監聽傳入的訊息。若訊息的關聯識別子(位於 inReplyTo 屬性內)與 idToCallbackMap 變數內的任何識別子相符，便表示這是一則回覆訊息。因此便呼叫相關聯的回應處置器，並傳入訊息內的資料。
  3. 最後，我們回傳用於傳送新請求的函式。此函式會藉由 node-uuid 套件來產生關聯識別子，接著會將請求資料包裹進一封「信件」裡，內含關聯識別子與訊息類型。

Node.js

- 請求/回覆模式(Request/Reply patterns)
  - 利用關聯識別子實作請求/回覆抽象層

接著來看看另一個相對應的模組 reply.js，也就是回覆處置器的抽象層:

```
module.exports = function(channel) {
  return function registerHandler(handler) {
    channel.on('message', message => {
      if (message.type !== 'request') return;

      handler(message.data, reply => {
        channel.send({
          type: 'response',
          data: reply,
          inReplyTo: message.id
        });
      });
    });
  };
};
```

Node.js

- 請求/回覆模式(Request/Reply patterns)

  - 利用關聯識別子實作請求/回覆抽象層

這個 reply 模組也是一個工廠，會回傳一個函式，用於註冊新的回覆處置器。新處置器的註冊過程如下:

1. 監聽傳入的請求，在收到請求時，立即呼叫 handler，同時傳入訊息資料(message.data)及一個回呼函式(function(message){...})，該函式是用於收集處置器的回覆(延續傳遞風格 CPS)。
2. 當處置器作業完成後，呼叫回呼以傳回 reply 資料。接著建置一封信件，內含請求的關聯識別子(inReplyTo 屬性)及其他資料，然後發送出去。

此模式的神奇之處在於，在 Node.js 裡幾乎所有事務都已經是非同步的，所以在單行通道上建置非同步的請求/回覆通信並不困難，因此我們可以輕易的建置出抽象層來隱藏實作細節。

Node.js

- 請求/回覆模式(Request/Reply patterns)

  - 利用關聯識別子實作請求/回覆抽象層

現在我們準備要來試驗新的非同步請求/回覆抽象層。首先，我們需要建立一個示範用的回覆者 replier.js:

```
var reply = require('./reply')(process);

reply((req, callback) => {
  setTimeout(() => {
    callback({sum: req.a + req.b});
  }, req.delay);
});
```

這個回覆者會單純的對收到的兩個數字計算總和，然後在一段延遲時間(於請求中指定)後回傳結果。藉此讓我們檢驗，回應的順序是否可以不同於請求的順序，印證此模式的可行性。

Node.js

- 請求/回覆模式(Request/Reply patterns)

  - 利用關聯識別子實作請求/回覆抽象層

這項範例的最後步驟，是在 requestor.js 裡建立請求者，並透過 child_process.fork() 來啟動回覆者：

```
const replier = require('child_process')
                .fork(`${__dirname}/replier.js`);
const request = require('./request')(replier);

request({a: 1, b: 2, delay: 500}, res => {
  console.log(`1 + 2 = ${res.sum}`);
  replier.disconnect();
});

request({a: 6, b: 1, delay: 100}, res => {
  console.log(`6 + 1 = ${res.sum}`);
});
```

這個請求者會啟動回覆者，然後傳送其參考給 request 抽象層。我們接著執行兩個示範請求，藉此確認請求及回應的配對關係是否正確。

Node.js

- 請求/回覆模式(Request/Reply patterns)
  - 利用關聯識別子實作請求/回覆抽象層

執行 requestor.js 模組，輸出結果應如下所示:

```
6 + 1 = 7
1 + 2 = 3
```

這證實此模式完美的運作，回覆對應於正確的請求，無論它們傳送或接收順序為何。

Node.js

- 請求/回覆模式(Request/Reply patterns)

  - 回傳位址

  關聯識別子是在單行通道上建立請求/回覆通訊的基本模式。然而，若我們的訊息架構有一個以上的通道、佇列或請求者時，這就不敷使用了。因此，除了使用關聯識別子外，我們還必須瞭解如何運用**回傳位址(return address)**，確保回覆者可以傳送回應給正確的請求者。

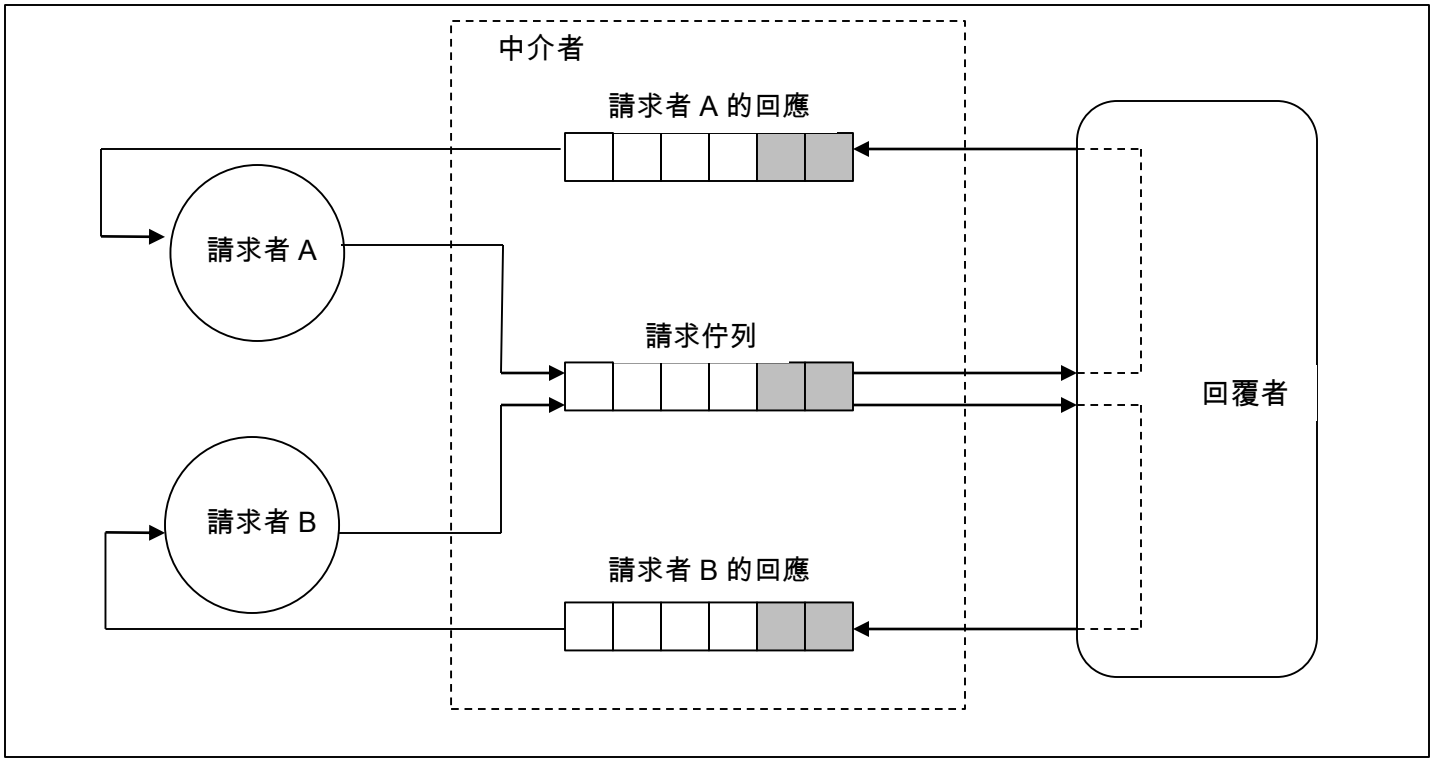
  - AMQP 的回傳位址模式

  在 AMQP 下，回傳位址是一個佇列，讓請求者監聽傳入的回覆。由於一個回覆應該僅僅對應一個請求者，因此回傳佇列應該是私有的，不會被其它的消耗者所用。根據前述特性，這表示我們需要建立單一的請求佇列，然後為各個請求者建立個別的回傳佇列，讓回覆者以對點通訊方式傳送回應。

Node.js

- 請求/回覆模式(Request/Reply patterns)
  - AMQP 的回傳位址模式

設計概念請參見下圖:



Node.js

- 請求/回覆模式(Request/Reply patterns)

  - AMQP 的回傳位址模式

在 AMQP 上建立請求/回覆模式，我們只需要在訊息屬性裡指定回應佇列的名稱即可。如此一來，回覆者便能夠知道回應訊息的目的地。概念很簡單，至於實作方式，就讓我們繼續看下去。

首先，我們要在 AMQP 上建置一個請求/回覆抽象層。這裡使用 RabbitMQ 作為中介者，不過應該也可以使用其他相容 AMQP 的中介者。讓我們從請求的抽象層開始實作 amqpRequest.js 模組。

```
const uuid = require('node-uuid');
const amqp = require('amqplib');

function AMQPRequest() {
  this.idToCallbackMap = {};
  this.channel = null;
  this.replyQueue = '';
}
```

Node.js

- 請求/回覆模式(Request/Reply patterns)
  - AMQP 的回傳位址模式

```
AMQPRequest.prototype.initialize = function () {
  return amqp
    .connect('amqp://10.11.25.137')
    .then(conn => conn.createChannel())
    .then(channel => {
      this.channel = channel;
      return channel.assertQueue('', {exclusive: true});
    })
    .then(q => {
      this.replyQueue = q.queue;
      return this._listenForResponses();
    })
    .catch( err => {
      console.log(err);
    });
};
```

Node.js

- 請求/回覆模式(Request/Reply patterns)
  - AMQP 的回傳位址模式

```
AMQPRequest.prototype._listenForResponses = function() {
  return this.channel.consume(this.replyQueue, msg => {
    const correlationId = msg.properties.correlationId;
    const handler = this.idToCallbackMap[correlationId];
    if (handler) {
      handler(JSON.parse(msg.content.toString()));
    }
  }, {noAck: true});
};

AMQPRequest.prototype.request = function (queue, message, callback) {
  const id = uuid.v4();
  this.idToCallbackMap[id] = callback;
  this.channel.sendToQueue(queue,
    new Buffer(JSON.stringify(message)),
    {correlationId: id, replyTo: this.replyQueue}
  );
};

module.exports = function() { return new AMQPRequest() };
```

Node.js

- 請求/回覆模式(Request/Reply patterns)

  - AMQP 的回傳位址模式

1. 首先，我們使用 channel.assertQueue(‘’, {exclusive: true}) 建立佇列以存放回應。我們在建立這個佇列時，並未指定任何名稱，這表示它會使用一個隨機的名稱。除此之外，這個佇列設定為「專用性」，這表示它綁定於作用中的 AMQP 連線，將於連線關閉時銷毀。這裡並不需要將佇列綁定交換元件，因為並無輸送或分派訊息至多個佇列的需求，而是直接遞送到我們的回應佇列裡。
2. 新的請求則是透過 request() 方法產生的，此函式會接收請求的佇列名稱(queue)以及欲傳送的訊息(message)。如同先前所學到的，我們也必須產生一個關聯識別子，與 callback 函式建立關聯。最後，我們傳送訊息，並設定 correlationId 與 replyTo 屬性作為中介資料(metadata)。
3. 這裡是使用 channel.sendToQueue() API 而非 channel.publish() 來傳送訊息，因為我們不打算使用交換元件來實作任務的發佈/訂閱模式，而是企圖以基本的對點方式(point-to-point)直接遞送至目的地佇列。
4. amqpRequest 原型的最後一項重點是監聽傳入的回應，_listenForResponses() 函式監聽來自回應佇列的訊息，並讀取每個傳入訊息的關聯識別子，然後與等待回覆的處置器進行比對。在取得相符的處置器後，便呼叫處置器並傳入回覆訊息。

Node.js

- 請求/回覆模式(Request/Reply patterns)
  - AMQP 的回傳位址模式

與 amqpRequest 模組搭配的另一個模組名為 amqpReply.js，我們要在此實作回應抽象層。

```
const amqp = require('amqplib');

function AMQPReply (qName) {
  this.qName = qName;
  this.channel = null;
  this.queue = '';
}
```

Node.js

- 請求/回覆模式(Request/Reply patterns)
  - AMQP 的回傳位址模式

```
AMQPReply.prototype.initialize = function () {
  return amqp
    .connect('amqp://10.11.25.137')
    .then(conn => conn.createChannel())
    .then(channel => {
      this.channel = channel;
      return this.channel.assertQueue(this.qName);
    })
    .then(q => this.queue = q.queue)
    .catch(err => console.log(err.stack));
};
```

Node.js

- 請求/回覆模式(Request/Reply patterns)
  - AMQP 的回傳位址模式

```
AMQPReply.prototype.handleRequest = function(handler) {
  return this.channel.consume(this.queue, msg => {
    const content = JSON.parse(msg.content.toString());
    handler(content, reply => {
      this.channel.sendToQueue(
        msg.properties.replyTo,
        new Buffer(JSON.stringify(reply)),
        {correlationId: msg.properties.correlationId}
      );
      this.channel.ack(msg);
    });
  });
};

module.exports = function(qName) {
  return new AMQPReply(qName);
};
```

Node.js

- 請求/回覆模式(Request/Reply patterns)

  - AMQP 的回傳位址模式

1. 首先使用 channel.assertQueue() API 建立一個佇列以接收傳入的請求，對此，我們使用一個簡單的可延續佇列(durable queue)。
2. handleRequest() 函式是真正值得關注的地方，這裡處理請求，以及如何回覆給正確的佇列。在進行回覆時，我們使用 channel.sendToQueue() 直接發佈訊息到 replyTo 屬性(也就是回傳位址)所指定的佇列裡。amqpReply 物件的另一項重要任務是設定回覆的 correlationId，如此一來接收者便可以將訊息與等待中的請求進行比對(取出回呼函式)。

經過前述的步驟，我們已完成了此系統的實作。不過接下來還要再建立示範用得請求者及回覆者，才能夠試驗這個新的抽象層。

Node.js

- 請求/回覆模式(Request/Reply patterns)

  - AMQP 的回傳位址模式

首先是 replier.js 模組:

```
const Reply = require('./amqpReply');
const reply = Reply('demo_requests_queue');

reply.initialize().then(() => {
  reply.handleRequest((req, callback) => {
    console.log('Request received', req);
    setTimeout(() => {
      callback({sum: req.a + req.b});
    }, req.delay);
  });
});
```

可以看到，我們所建置的抽象層，能夠隱藏關聯識別子及回傳位址的所有相關機制。我們只需要初始化一個新的 reply 物件，並指定想要接收請求者的佇列名稱 demo_requests_queue，程式碼的其餘部份相當簡單，這個示範用的回覆者只是單純的計算輸入的兩個數字總和，再透過回呼回傳結果。

Node.js

- 請求/回覆模式(Request/Reply patterns)
  - AMQP 的回傳位址模式

接著，我們在 requestor.js 模組裡實作示範用的請求者:

```
const req = require('./amqpRequest')();

req.initialize().then(() => {
  for (let i = 100; i > 0; i--) {
    sendRandomRequest();
  }
});

function sendRandomRequest() {
  const a = Math.round(Math.random() * 100);
  const b = Math.round(Math.random() * 100);
  req.request('demo_requests_queue', {a: a, b: b, delay: a + b}, res => {
    console.log(`${a} + ${b} = ${res.sum}`);
  });
}
```

Node.js

- 請求/回覆模式(Request/Reply patterns)

  - AMQP 的回傳位址模式

這個請求者會傳送 100 次的隨機請求到 demo_requests_queue 佇列。我們同樣可以在這裡看到，抽象層隱藏了非同步請求/回覆模式的所有細節，充分突顯了它的作用與意義。

現在，讓我們先後執行 replier 模組與 requestor 模組，來試驗這個系統:

```
node replier
node requestor
```

我們將看到一系列由請求者所發佈的操作，這些操作會被回覆者接收，接著再由回覆者傳送回應。

由於回覆者在初次啟動時，會建立一個可延續佇列，所以如果它被重新啟動，也不會有任何請求會被遺漏。所有的訊息都會被儲存在佇列裡，直到回覆者再次啟動。

Node.js

- 請求/回覆模式(Request/Reply patterns)

  - AMQP 的回傳位址模式

透過 AMQP 能夠無償實現的一項功能，就是回覆者是內建可擴展性的。可以試著啟動兩個或多個回覆者實例，再觀察請求是如何在實例之間做到負載平衡的。之所以能夠實現負載平衡，是因為每個回覆者在啟動時，都會監聽相同的可延續佇列。因此中介者會在所有的佇列消耗者之間，平均分派所有的訊息(競爭消耗者模式)。

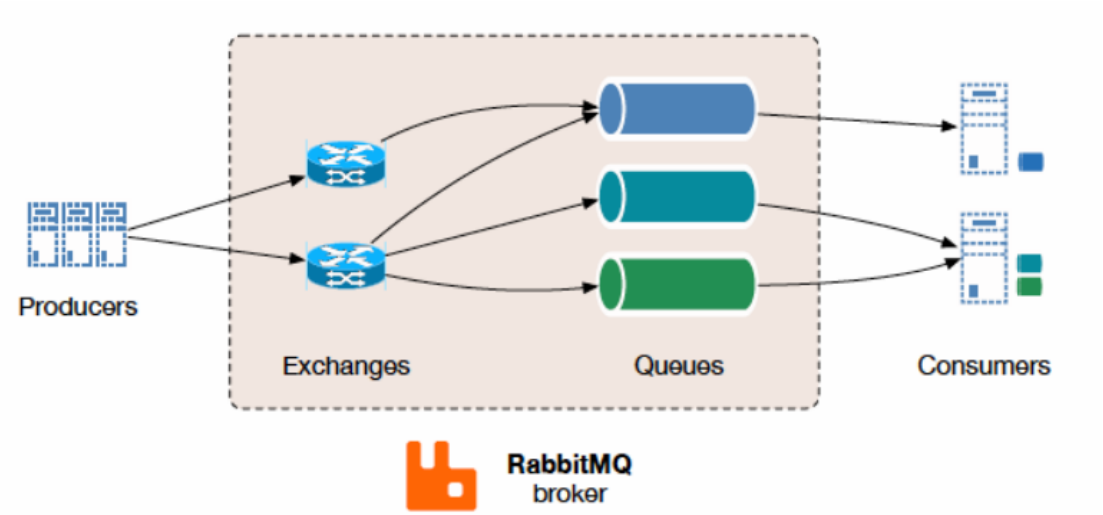
ØMQ 有一組 socket 是專門用於實作請求/回覆模式(REQ/REP)，不過它們是同步的(同時間只能一次請求/回覆)。

利用 AMQP 與完整的訊息中介者，可以實作出可靠且能夠擴展的應用程式，不過相應的代價是新增了一個同樣需要維護的節點。

Node.js

- 訊息與整合模式

  - RabbitMQ (AMQP)

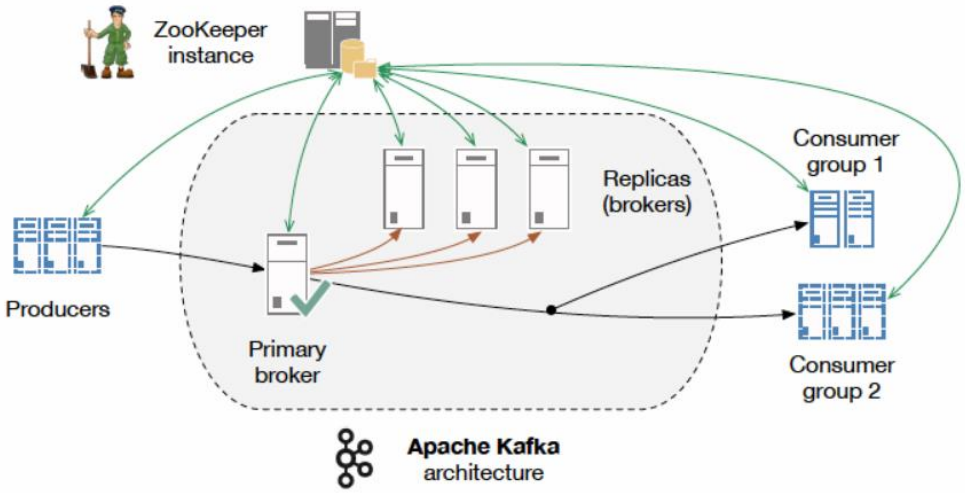


RabbitMQ 被設計為通用消息代理，採用點對點，請求/回覆和 pub-sub 通信樣式模式的多種變體。它使用智能代理/啞消費者模型，專注於向消費者提供一致的消息傳遞，消費者的消費速度與經紀人跟踪消費者狀態的速度大致相似。它是成熟的，在正確配置時表現良好，並且有許多可用的插件可以將它擴展到更多的用例和場景。當應用程序需要訪問流歷史時，RabbitMQ 通常與 Apache Cassandra 一起使用，對於需要“無限”隊列的應用程序，RabbitMQ 通常與 LevelDB 插件一起使用，但這兩個功能都不附帶 RabbitMQ 本身。

Node.js

- 訊息與整合模式

  - Apache Kafka



Apache Kafka 專為高容量發布/訂閱消息和流(Streams)而設計，旨在持久，快速和可擴展。從本質上講，Kafka 提供了一個持久的消息存儲，類似於在服務器集群中運行的日誌，它存儲稱為主題(Topic)的類別中的記錄流。Kafka 會在一段時間內保留所有消息，Kafka 不會嘗試跟踪每個消費者讀取的消息，消費者有責任在每個日誌（消費者狀態）中跟踪他們的位置。因此，通過合適的開發人員創建消費者代碼，Kafka 可以支持大量消費者並以極少的開銷保留大量數據。Kafka 是一個持久的消息存儲，客戶可以根據需要“重播”事件流，而不是像傳統的消息代理，一旦消息傳遞，它就會從隊列中刪除。Kafka 讓中間件保持單純而無知，將邏輯與智慧留給端點。

Node.js

- RabbitMQ 極簡應用
  - 發佈(Publisher)

```
const amqp = require('amqplib');
const q = 'demo_tasks';

// Publisher
amqp.connect('amqp://10.11.25.137')
  .then(function(conn) {
    return conn.createChannel();
  }).then(function(ch) {
    return ch.assertQueue(q).then(function(ok) {
      return ch.sendToQueue(q, Buffer.from('您好，這裡有事可做 ' + Date.now()));
    });
  }).catch(console.warn);
```

Node.js

- RabbitMQ 極簡應用
  - 訂閱(Consumer)

```
const amqp = require('amqplib');
const q = 'demo_tasks';

// Consumer
amqp.connect('amqp://10.11.25.137')
  .then(function(conn) {
    return conn.createChannel();
  }).then(function(ch) {
    return ch.assertQueue(q).then(function(ok) {
      return ch.consume(q, function(msg) {
        if (msg !== null) {
          console.log(msg.content.toString());
          ch.ack(msg);
        }
      });
    });
  }).catch(console.warn);
```

Node.js

ØMQ 支持許多不同的消息傳遞模式，這些模式在 Node.js 中運行良好，為學習三種基本的微觀模式(Microservices)提供了基礎：

- 發佈和訂閱(publish/subscribe)
- 請求回覆(request/reply)
- 路由和處理(dealer/router)
- 推送和拉取(push/pull)

```
$ npm install --save zeromq
$ mkdir microservices
$ cd microservices
$ node -p -e "require('zeromq').version"
4.2.2
```

Node.js

```
// microservices/zmq-watcher-pub.js
const fs = require('fs');
const zmq = require('zeromq');
const filename = process.argv[2];

const publisher = zmq.socket('pub');

fs.watch(filename, () => {
  publisher.send(JSON.stringify({
    type: 'changed',
    file: filename,
    timestamp: Date.now()
  }));
});

publisher.bind('tcp://*:60400', err => {
  if (err) {
    throw err;
  }
  console.log('Listening for zmq subscribers...');
});
```

publish/subscribe

伺服器只負責發佈消息，並不參與訊息的交換，應用到系統上，整體的概念就是各自有不同的子系統，透過通訊的方式將它們串在一起，這樣做有個好處就是效能好，再來就是設計得當的話，某子系統雖然停擺，但不會影響到所有的系統。分散式的系統可透過遠端的服務整合。ØMQ 也支援各種語言，所以也可跨語言整合應用系統。

```
$ node zmq-watcher-pub ../filesystem/target.txt
Listening for zmq subscribers...
```

Node.js

publish/**subscribe**

```
// microservices/zmq-watcher-sub.js
const zmq = require('zeromq');
const subscriber = zmq.socket('sub');

subscriber.subscribe('');

subscriber.on('message', data => {
  const message = JSON.parse(data);
  const date = new Date(message.timestamp);
  console.log(`File "${message.file}" changed at ${date}`);
});

subscriber.connect('tcp://localhost:60400');
```

```
$ node zmq-watcher-sub
File "../filesystem/target.txt" changed at Tue Jul 24 2018 15:59:34 GMT+0800 (台北標準時間)
```

看起來很簡單，但如果把 Publisher 停掉，這個 Subscriber 並不會查覺，所以會繼續等待。如果重新啟動 Publisher，這個 Subscriber 仍然會正常運作，從 ØMQ 的角度來看，哪個端點首先啟動並不重要，它會在任一端點啟動時自動建立並重新建立連接，這些特性構成了一個強大的平台，無需您的大量工作即可為您提供穩定性。

當設計網路應用程序時，您通常會有永久性的架構綁定部分，並會短時間連接到它們。 使用 ØMQ ，您可以決定系統的哪些部分將會需要交換，以及那種消息模式最適合您的需求。但是你不必同時決定它們，以後很容易改變主意。 ØMQ 為構建分佈式應用程序提供靈活，耐用的管道。

Node.js

request/**reply**

```
// microservices/zmq-filer-rep.js
const fs = require('fs');
const zmq = require('zeromq');

const responder = zmq.socket('rep');

responder.on('message', data => {
  const request = JSON.parse(data);
  console.log(`Received request to get: ${request.path}`);

  fs.readFile(request.path, (err, content) => {
    console.log('Sending response content.');
    responder.send(JSON.stringify({
      content: content.toString(),
      timestamp: Date.now(),
      pid: process.pid
    }));
  });
});

responder.bind('tcp://*:60401', err => {
  console.log('Listening for zmq requesters...');
});

process.on('SIGINT', () => {
  console.log('Shutting down...');
  responder.close();
});
```

Socket to reply to client requests

SIGINT， Unix 信號表示 process 收到用戶的中斷信號。 通常是透過在終端中按 Ctrl-C 來觸發。在這種情況下要做的事情是要求 responder 關閉任何未完成的連接。

Node.js

request/reply

```
// microservices/zmq-filer-req.js
const zmq = require('zeromq');
const filename = process.argv[2];

const requester = zmq.socket('req');

requester.on('message', data => {
  const response = JSON.parse(data);
  console.log('Received response:', response);
});

requester.connect('tcp://localhost:60401');

console.log(`Sending a request for ${filename}`);
requester.send(JSON.stringify({path: filename}));
```

```
$ node zmq-filer-req ../filesystem/target.txt
Sending a request for ../filesystem/target.txt
Received response: { content: '哈囉， World\r\nhi hi hi',
  timestamp: 1532479164548,
  pid: 2604 }
```

Node.js

```
for (let i = 1; i <= 5; i++){
  console.log(`Sending a request ${i} for ${filename}`);
  requester.send(JSON.stringify({path: filename}));
}
```

request/reply

將 send 改為循環連續送出需求，這會將 send 送入 Node.js Event-loop 中

```
$ node zmq-filer-req-loop ../filesystem/target.txt
Sending a request 1 for ../filesystem/target.txt
Sending a request 2 for ../filesystem/target.txt
Received response: { content: '哈囉, World\r\nhi hi hi',  timestamp: 1532481725790,  pid: 2604 }
Sending a request 3 for ../filesystem/target.txt
Sending a request 4 for ../filesystem/target.txt
Sending a request 5 for ../filesystem/target.txt
Received response: { content: '哈囉, World\r\nhi hi hi',  timestamp: 1532481725835,  pid: 2604 }
Received response: { content: '哈囉, World\r\nhi hi hi',  timestamp: 1532481725873,  pid: 2604 }
Received response: { content: '哈囉, World\r\nhi hi hi',  timestamp: 1532481725903,  pid: 2604 }
Received response: { content: '哈囉, World\r\nhi hi hi',  timestamp: 1532481725942,  pid: 2604 }
```

Requests 在循環中排隊，發送和接收可能是交錯的，具體取決於回應的速度

```
$ node zmq-filer-rep
Listening for zmq requesters...
Received request to get: ../filesystem/target.txt
Sending response content.
Received request to get: ../filesystem/target.txt
Sending response content.
Received request to get: ../filesystem/target.txt
Sending response content.
Received request to get: ../filesystem/target.txt
Sending response content.
Received request to get: ../filesystem/target.txt
Sending response content.
```

ØMQ REP/REQ socket 與 Node.js 結合使用有一個問題。 應用程序的每個端點一次只能處理一個請求或一個響應，沒有並行性。responder 一次只能知道一條消息。注意這裡，responder 程序在知道下一個排隊請求之前，已向每個請求發送了回應。 這意味著 Node.js 在處理每個請求的 fs.readFile() 時，事件循環保持旋轉。因此，簡單的 REQ/REP 可能不適合高性能的 Node.js 的需求。

Routing and Dealing Messages

- Routing and Dealing Messages

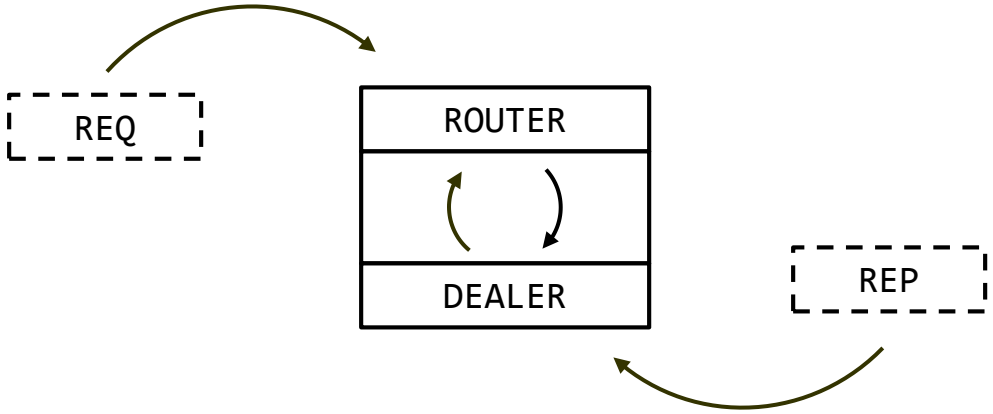
ROUTER socket 是並行的 REP socket，DEALER socket 則是並行的 REQ socket。 DEALER socket 可以並行發送多個請求。

```
const router = zmq.socket('router');
const dealer = zmq.socket('dealer');

router.on('message', (...frames) => dealer.send(frames));
dealer.on('message', (...frames) => router.send(frames));
```

在這裡，我們創建了一個 ROUTER socket 和一個 DEALER socket。 無論何時任何一方收到消息，它都會將 frames 送到另一個 socket。

這創建了一種直通關係，傳入 ROUTER 的請求將直接轉傳給 DEALER 以發送到連接它的端點。同樣，對 DEALER 的回覆將被轉發到 ROUTER，ROUTER 將每個回覆送回請求它的連接端點。

Routing and Dealing Messages



圖中心的方框是 Node.js 程序，傳入的 REQ socket 連接到 ROUTER。 當 REQ socket 發出請求時，ROUTER 將其反彈到 DEALER。然後，DEALER 選擇與其連接的下一個 REP socket（round-robin style）並轉發請求。

當 REP 連接產生回覆時，它遵循反向路由。 DEALER 收到回覆並將其反饋給 ROUTER。 ROUTER 查看消息的 frames 以確定其來源並將回覆發送回發送初始請求的連接的 REQ

從 REQ 和 REP socket 的角度來看，沒有任何改變。每個仍然一次處理一條消息，同時，ROUTER / DEALER 可以分配(round-robin)在兩端連接的 REQ 和 REP。

現在我們可以用以上的 REQ/REP， 與 ROUTER/DEALER socket 之上開發一個集群(clustered) Node.js 應用程序。

Routing and Dealing Messages

- Routing and Dealing Messages

在多線程系統(multithread)中，可以並行執行更多工作並激活更多線程(thread)。 但 Node.js 使用單線程事件循環(single-threaded event loop)，因此要在同一台計算機上利用多個核心或多個處理器，您必須啟動更多的 Node.js processes。

這稱為集群(cluster)，它是 Node.js 內置集群模塊(built-in cluster module)所做的事情。當有未使用的 CPU 容量可用時，群集是擴展 Node.js 應用程序的有用技術。

為了探索集群模塊的工作原理。我們將構建一個程序來管理 pool of worker processes 以響應 ØMQ 的請求。這將是我們之前的 responder 的替代品。它將使用 ROUTER，DEALER 和 REP socket 將請求分配給各個 worker。

總而言之，我們最終會得到一個簡短而強大的程序，它結合了基於群集的多進程工作(multiprocess work)來分配訊息和負載均衡。

```
const router = zmq.socket('router');
const dealer = zmq.socket('dealer');

router.on('message', (...frames) => dealer.send(frames));
dealer.on('message', (...frames) => router.send(frames));
```

Routing and Dealing Messages

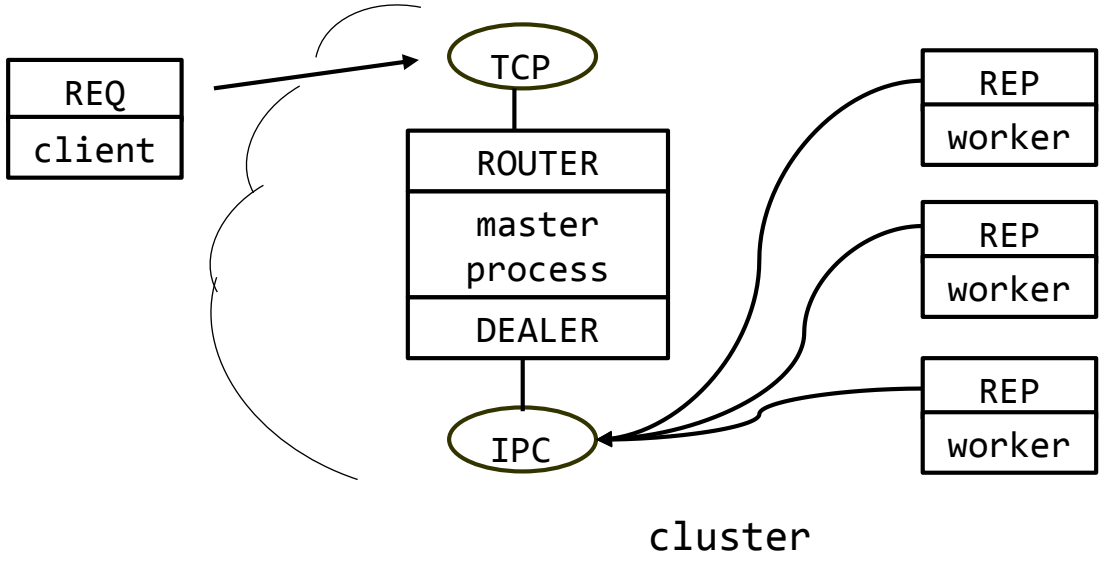
```
const cluster = require('cluster');

if (cluster.isMaster) {
  // Fork some worker processes.
  for (let i = 0; i < 10; i++) {
    cluster.fork();
  }
} else {
  // This is a worker process; do some work.
}
```

我們曾經使用 child_process 模塊的 spawn() 函數來啟動一個進程(process)。這非常適合從 Node.js 程序執行非 Node.js 進程。但是為了產生同一個 Node.js 程序的副本，forking 是一個更好的選擇，因為 fork() 是 spawn() 的一個特殊版本，它設置了進程間通信通道(IPC, interprocess communication channel)。

每次調用 cluster 模塊的 fork() 方法時。它創建一個運行與原始腳本相同的工作進程(worker process)。forked processes 的進程稱為 workers。 他們可以通過各種 events 與 master process 互通。

```
cluster.on('online', worker =>
  console.log(`Worker ${worker.process.pid} is online.`));

cluster.on('exit', (worker, code, signal) =>
  console.log(`Worker ${worker.process.pid} exited with code ${code}`));
```

Routing and Dealing Messages



我們的 Node.js 主程序將創建 ROUTER 和 DEALER sockets 並啟動 workers 工作程序。每個 worker 都將創建一個連接回 DEALER 的 REP socket。

Master process 是體系結構中最穩定的部分（它管理 workers），因此它負責進行 binding。worker processes 和 service 的 client 都連接到主服務器綁定的端點。請記住，訊息的流動由 socket 類型決定，而不是綁定或連接的 socket。

Routing and Dealing Messages

```
// microservices/zmq-filer-rep-cluster.js
const cluster = require('cluster');
const fs = require('fs');
const zmq = require('zeromq');

const numWorkers = require('os').cpus().length;

if (cluster.isMaster) {
  const router = zmq.socket('router').bind('tcp://*:60401');
  const dealer = zmq.socket('dealer').bind('tcp://127.0.0.1:60402');
  //const dealer = zmq.socket('dealer').bind('icp://filer-dealer.ipc');

  router.on('message', (...frames) => dealer.send(frames));
  dealer.on('message', (...frames) => router.send(frames));

  cluster.on('online', worker => console.log(`Worker ${worker.process.pid} is online.`));

  for (let i = 0; i < numWorkers; i++) {
    cluster.fork();
  }
} else {
```

Master process creates ROUTER and DEALER sockets and binds endpoints

Windows 沒有 IPC,用 TCP 取代

接下頁

Routing and Dealing Messages

```
  const responder = zmq.socket('rep').connect('tcp://127.0.0.1:60402');
  //const responder = zmq.socket('rep').connect('ipc://filer-dealer.ipc');

  responder.on('message', data => {
    const request = JSON.parse(data);
    console.log(`${process.pid} received request for: ${request.path}`);

    fs.readFile(request.path, (err, content) => {
      console.log(`${process.pid} sending response`);
      responder.send(JSON.stringify({
        content: content.toString(),
        timestamp: Date.now(),
        pid: process.pid
      }));
    });
  });
}
```

承上頁

Worker processes create a REP socket and connect to the DEALER

Read the file and reply with content

```
$ node zmq-filer-rep-cluster
Worker 5068 is online.
Worker 7212 is online.
Worker 1172 is online.
Worker 7780 is online.
```

```
$ node zmq-filer-req-loop ../filesystem/target.txt
```

Pushing and Pulling Messages

```
// microservices/zmq-watcher-push.js
const watch = require('node-watch');
const zmq = require('zeromq');
const filename = process.argv[2];

const pusher = zmq.socket('push');
pusher.bind('tcp://127.0.0.1:60401', err => {
  if (err) { throw err };
  console.log('Listening for zmq pullers...');
});

// Send message
watch(filename, () => {
  console.log('Sending file changed message ...')
  pusher.send(JSON.stringify({
    type: 'changed',
    file: filename,
    timestamp: Date.now()
  }));
});
```

使用 PUB/SUB，每個用戶將收到所有由 Publisher 發出的消息送。在 PUSH/PULL中，則只有一個 Puller 會收到 Pusher 發送的每條消息。

node-watch 取代 fs.watch

Watch file changed and send

```
$ node zmq-watcher-push ../filesystem/target.txt
Listening for zmq pullers..
```

Pushing and Pulling Messages

```
// microservices/zmq-watcher-pull.js
const zmq = require('zeromq');
const puller = zmq.socket('pull');

puller.connect('tcp://127.0.0.1:60401');

console.log('Puller connected to port 60401')

puller.on('message', data => {
  const message = JSON.parse(data.toString());
  const date = new Date(message.timestamp);
  console.log(`File "${message.file}" changed at ${date}`);
});

puller.connect('tcp://127.0.0.1:60401');
```

```
$ node zmq-watcher-pull
Puller connected to port 60401
File "../filesystem/target.txt" changed at Mon Jul 30 2018 09:09:15 GMT+0800 (台北標準時間)
```

只有一個 Puller 會收到訊息

Pushing and Pulling Messages

```
// microservices/zmq-watcher-pull-cluster.js
const cluster = require('cluster');
const zmq = require('zeromq');

const numWorkers = require('os').cpus().length;
let readyCount = 0;

if (cluster.isMaster) {
  const pusher = zmq.socket('push').bind('tcp://127.0.0.1:60401');
  const puller = zmq.socket('pull').bind('tcp://127.0.0.1:60402');

  cluster.on('online', worker => {
    console.log(`Worker ${worker.process.pid} is online.`);
  });

  for (let i = 0; i < numWorkers; i++) {
    cluster.fork();
  }

  puller.on('message', data => {
    const message = JSON.parse(data);
```

使用 Node.js Cluster,可以模擬並行處裡

這會由 worker 送回的信息

接下頁

Pushing and Pulling Messages

```
    if (message && message.ready) {
      if (readyCount++ === 3 ){
        for (let i = 0; i < 100; i++) {
          pusher.send(JSON.stringify({
            details: `Details about job ${i}`,
            job: i
          }));
        }
      }
    } else if (message && message.result) {
      console.log(message);
    }
  });
} else {
  } else {
  const workerPuller = zmq.socket('pull').connect('tcp://127.0.0.1:60401');
  const workerPusher = zmq.socket('push').connect('tcp://127.0.0.1:60402');

  workerPusher.send(JSON.stringify({
    ready: true
  }));
```

由 worker 送回的如果是 ready 訊息，readyCount 加 1，這表示 worker 已經準備號可以接受工作了。readyCount 等於 3 時就送出 100 個 jobs。

連結 Master 的 Pusher & Puller

送出 ready 訊息，我已經準備就緒可以接受工作。

接下頁

Pushing and Pulling Messages

```
    workerPuller.on('message', data => {
    const message = JSON.parse(data.toString());
    // Do something ...
    const result = "Results 處理結果.";

    workerPusher.send(JSON.stringify({
      result: result,
      timestamp: Date.now(),
      job: message.job,
      pid: process.pid
    }));
  });
}
```

承上頁

接受 Master 送來的訊息(Job)，並送回處理後的結果。包含 Worker 的 pid。

```
$ node zmq-watcher-pull-cluster
Worker 3952 is online.
Worker 5928 is online.
Worker 6236 is online.
Worker 7072 is online.
{ result: 'Results 處理結果.',
  timestamp: 1532921749230,
  job: 0,
  pid: 3952 }
.....
```

Worker 送回 Master 的處裡結果，訊息包含處裡的 Worker pid。

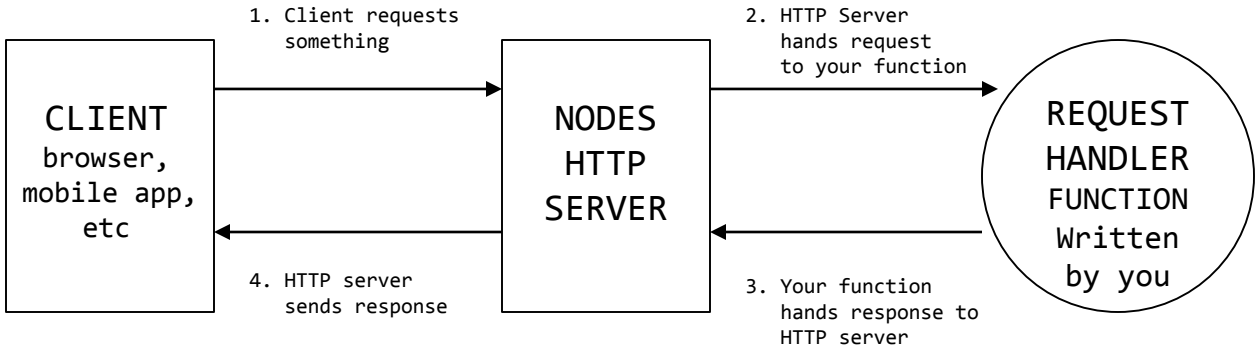
Base HTTP Server

```
// base-http-server.js
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});

  res.end('Hello Tainan!\n');
});

server.listen(60700, () => console.log("Ready."));
```

http.createServer() 接受一個回呼函數，將於伺服器接收到每一個 HTTP 請求 (Request) 時被呼叫。這個請求回呼接受兩個引數，Request 物件與 Response 物件。

```
$ node base-http-server
Ready.
```

```
$ curl http://localhost:60700
Hello Tainan!
```



Base HTTP Server

```
// server/base-http-server.js
const http = require('http');

const server = http.createServer((req, res) => {
  console.log(req.url);
  console.log(req.method);

  res.writeHead(200, {'Content-Type': 'text/plain'});

  res.end('Hello Tainan!\n');
});

server.listen(60700, () => console.log("Ready."));
```

Request 物件與 Response 物件是最關鍵的兩個物件

```
$ node base-http-server
Ready.
/hello
GET
/hello
POST
```

```
$ curl http://localhost:60700/hello
Hello Tainan!

$ curl -X POST http://localhost:60700/hello
Hello Tainan!
```

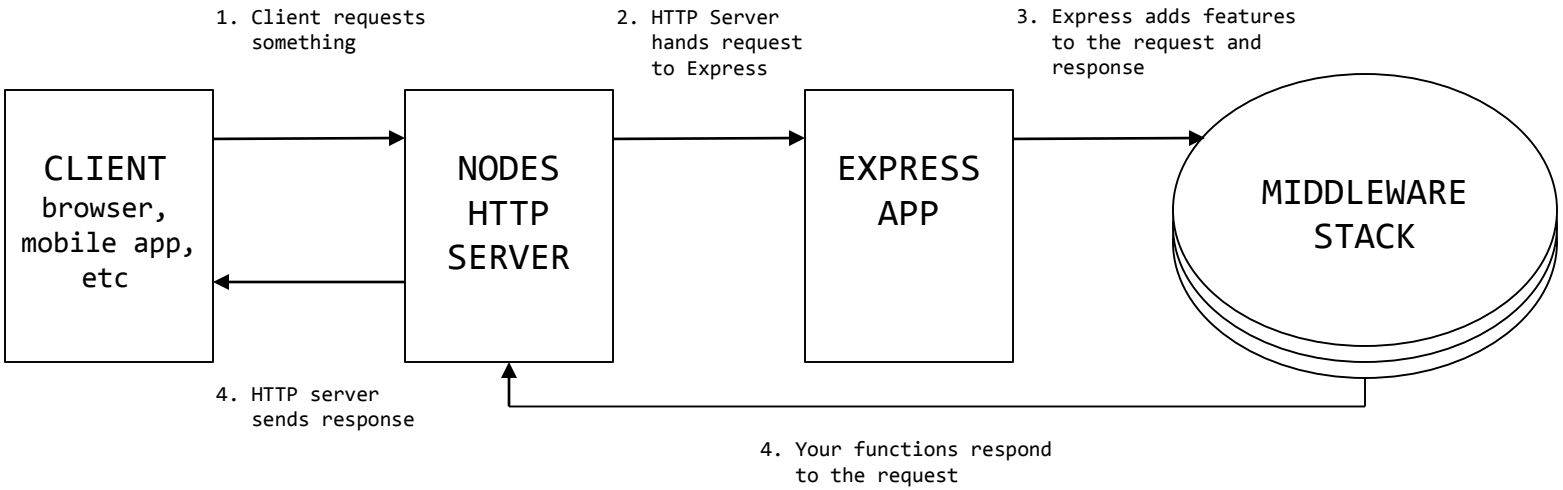
Base HTTP Server

```
// server/base-http-server.js
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});

  if (req.url === '/') {
    if (req.method === 'GET') {
      return res.end('Welcome!\n');
    }
  } else if (req.url.startsWith('/hello')) {
    return res.end('Hello user!\n');
  }

  res.end('Welcome!\n');
});

server.listen(60700, () => console.log("Ready."));
```

回應不同的網址，需要不同的程式邏輯，
所以我們需要使用 Framework。

```
$ curl http://localhost:60700
Welcome!

$ curl http://localhost:60700/hello
Hello user!
```

Express




```
$ npm install express --save
```

```
$ npm install -g express-generator@4
$ express --version
$ express app-server
$ cd app-server
$ npm install
$ npm start
```

express-generator 可讓我們快速的建立 express project

產生 express 專案目錄

安裝 express 相關的模組與 Middleware

Express

```
// app-server/server.js
const express = require('express');

const app = express();

app.get('/', (req, res) => {
  res.status(200).json({message: 'Hello Tainan!'});
});

app.get('/hello/:name', (req, res) => {
  res.status(200).json({'hello': req.params.name});
});

app.listen(60701, () => console.log('Ready.'));
```

Calls the express function to start a new Express application. app is just a function, a **request handler function**.

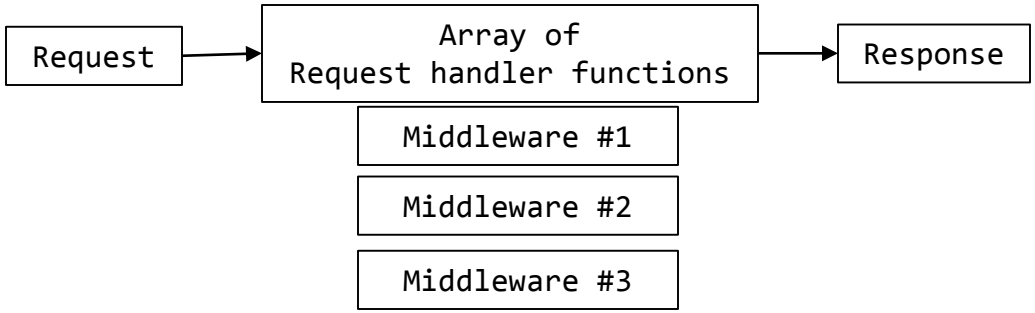

```
$ curl http://localhost:60701
{"message":"Hello Tainan!"}

$ curl http://localhost:60701/hello/Scott%20Tiger
{"hello":"Scott Tiger"}
```

Express

- Middleware

```
function requestHandler(request, response) {
  console.log("In comes a request to: " + request.url);
  response.end("Hello Tainan!");
}
```

如果沒有 Middleware，你必須在一個函數內處理所有的需求。每個 Request 只通過一個請求處理函數，並回應 Response，這並不是說主處理函數不能調用其他函數，而在結束時統籌由主函數回應每個請求。



使用 Middleware 可讓請求通過一系列函數，而不只是通過一個函數，而是稱為中間件堆棧(Middleware stack)的一系列函數。

Express

- Middleware

```
function myMiddleware(request, response, next) {
    ...
  next();
}
```

對請求和/或回應做些了什麼

完成後，調用 next() 來執行堆疊鏈中的下一個中間件

當你啟動 Server 時，會從最頂層的中間件開始，然後運行到底層。因此，如果您想添加簡單的 Logging，如下所示。

```
const express = require('express');
const http = require('http');
const app = express();

app.use((req, res, next) => {
  console.log(`In comes a ${req.method} to ${req.url}`);
  next();
});

app.use((req, res) => {
  res.status(200).json({message: 'Hello Tainan!'});
});

http.createServer(app).listen(60701, () => console.log('Ready.'));
```

這是 Middleware 基本模式，你可修改 Request 或 Response 物件，完成後 call next() 繼續往下執行。

Express

- Using static

```
const express = require('express');
const path = require('path');
const http = require('http');

const app = express();

app.use(express.static(path.join(__dirname, 'public')));

app.use((req, res) => {
  res.status(200).json({message: "Looks like you didn't find a static file"});
});

http.createServer(app).listen(60701, () => console.log('Ready.'));
```

Express.static 幫助您提供靜態文件，發送文件需要考慮許多情況和執行注意事項。上面的 Server，將顯示 public 目錄中的任何文件。 您可以隨意放置任何內容，服務器將發送它。 如果公用文件夾中不存在匹配的文件，它將繼續下一個中間件。 如果發送匹配的文件，express.static 將發送它並**停止中間件鏈(stop the middleware chain)**。

```
$ curl http://localhost:60701/employees.csv
empno,ename,job,mgr,hiredate,sal,comm,deptno
9990,瓊蝨,CLERK,,2016-08-24,1000,,20
```

Express

- express-generator

```
$ npm install -g express-generator

$ express --version
4.16.0

$ express --help
  Usage: express [options] [dir]
  Options:
        --version        output the version number
    -e, --ejs            add ejs engine support
        --pug            add pug engine support
        --hbs            add handlebars engine support
    -H, --hogan          add hogan.js engine support
    -v, --view <engine>  add view <engine> support (dust|ejs|hbs|hjs|jade|pug|twig|vash)
(defaults to jade)
        --no-view        use static html instead of view engine
    -c, --css <engine>   add stylesheet <engine> support (less|stylus|compass|sass) (defaults
to plain css)
        --git            add .gitignore
    -f, --force          force on non-empty directory
    -h, --help           output usage information
```

Express

- express-generator

```
$ express exp-example --view ejs --git

   create : exp-example\
   create : exp-example\public\
   create : exp-example\public\javascripts\
   create : exp-example\public\images\
   create : exp-example\public\stylesheets\
   create : exp-example\public\stylesheets\style.css
   create : exp-example\routes\
   create : exp-example\routes\index.js
   create : exp-example\routes\users.js
   .....

   change directory:
     > cd exp-example

   install dependencies:
     > npm install

   run the app:
     > SET DEBUG=exp-example:* & npm startusage information
```

Express

- nodemon

```
$ npm install -save nodemon

  // package.json
  "scripts": {
    "start": "nodemon server.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

nodemon 是一種 Node.js 工具，可以檢測目錄中的文件被更改時，自動重新啟動節點的應用程序，以幫助開發的進行。

RESTful Web Services

- Create, read, update, delete APIs

  在應用程需中有個常見的應用程序模式：create, read, update, and delete， 簡稱為 CRUD。它如何對應到 API，那就必須了解甚麼是 HTTP methods，這也稱為 HTTP verbs。

  - GET - Read

    用於取得資源。GET方法不應該改變你的應用程序的狀態，

    冪等性(idempotence)對 GET 請求很重要。冪等(Idempotent)是一個奇特的詞，意思是做一次應該與做多次沒有什麼不同。
  - POST - Create

    用於創建新資料，而不是修改已存在的資料。POST 是非冪等的，狀態將在您第一次 POST 時更改，也會在第二次和第三次更改，依此類推。
  - PUT - Update

    更好的名稱可能會更新或更改。PUT 是冪等的。
  - DELETE - Delete

    刪除資料。DELETE 是冪等的。

哪些 HTTP verbs 對應於哪些 CRUD 一直存在一些爭論。因為 PUT 可以像 POST 一樣創建記錄，你可以說 PUT 更適合創建。 POST 也可以更新記錄。HTTP 並沒有嚴格規定，由您自己決定要怎麼用。

RESTful Web Services

- HTTP status codes

HTTP 狀態範圍簡單的分類為：

1xx: 保留
2xx: 很好
3xx: 走開
4xx: 你搞砸了
5xx: 我搞砸了

- 200 range
  - 200: OK
  - 201: Created
  - 202: Accepted
  - 204: No Content (Delete version of 201)
- 300 range
  - 301: Moved Permanently
  - 303: See Other (resource is created and redirect to a new page.)
  - 307: Temporary Redirect
- 400 range
  - 401 and 403: Unauthorized and forbidden errors
  - 404: Not Found
- 500 range
  - 500: Internal Server Error

RESTful Web Services

- simple-crud-restapi

```
const express = require('express');
const http = require('http');
const app = express();

app.get("/", (req, res) => {
  res.status(200).send("你剛剛發了一個 GET 請求");
});

app.post("/", (req, res) => {
  res.status(201).send("一個 POST 請求？不錯");
});

app.put("/", (req, res) => {
  res.status(200).send("我不會看到很多 PUT 請求");
});

app.delete("/", (req, res) => {
  res.status(200).send("哦，小心，一個 DELETE ？");
});

http.createServer(app).listen(60701, () => console.log('Ready.'));
```

```
$ curl http://localhost:60701
你剛剛發了一個 GET 請求

 $ curl -X POST http://localhost:60701
一個 POST 請求？不錯

$ curl -X PUT http://localhost:60701
我不會看到很多 PUT 請求

 $ curl -X DELETE http://localhost:60701
哦，小心，一個 DELETE ？
```

```
$ curl -i http://10.11.100.91:60701
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 32
ETag: W/"20-c4qMXNeTDgVQIcNiJPcwbju6LJE"
Date: Mon, 27 Aug 2018 03:16:09 GMT
Connection: keep-alive

你剛剛發了一個 GET 請求
```

node-oracledb

- oracledb.getConnection(): 使用回呼(Callback)

```
const oracledb = require('oracledb');
const dbConfig = {
  user: "demo",
  password: "demo426",
  connectString: "10.11.25.139:1522/lx26.pec.com.tw"
};

oracledb.getConnection(dbConfig, (err, connection) => {
  if (err) {
    return console.log(err.message);
  }

  connection.execute('select user from dual', (err, result) => {
    if (err) {
      console.log(err.message);
      return connection.close();
    }
    console.log(result);
    connection.close();
  });
});
```

node-oracledb

- oracledb.getConnection(): 使用承諾(Promise)

```
module.exports = {
  user: "demo",
  password: "demo426",
  connectString: "10.11.25.139:1522/lx26.pec.com.tw"
};
```

```
const oracledb = require('oracledb');
const dbConfig = require('./dbconfig');

oracledb.getConnection(dbConfig)
  .then( connection => {
    return connection.execute('select user from dual')
      .then( result => {
        console.log(result);
        return connection.close();
      }).catch(err => {
        console.log(err.message);
        return connection.close();
      });
  }).catch(err => {
    console.log(err.message);
  });
```

node-oracledb

- Call Oracle Stored Procedure

```
const oracledb = require("oracledb");
const dbConfig = require("./dbconfig");

const sql = "BEGIN demo_testproc(:empno, :ename, :deptno); END;";
const bindvars = {
  empno: 9006,
  ename: { type: oracledb.STRING, dir: oracledb.BIND_OUT, maxSize: 20 },
  deptno: { type: oracledb.NUMBER, dir: oracledb.BIND_OUT }
};

let connection;
oracledb.getConnection(dbConfig)
  .then(conn => {
    connection = conn;
    return connection.execute(sql, bindvars);
  })
  .then(result => {
    console.log(result);
    connection.close();
  })
  .catch(err => {
    console.log(err.message);
    connection.close();
  });
```

```
CREATE OR REPLACE PROCEDURE demo_testproc
(
  empno_in   IN NUMBER,
  ename_out  OUT VARCHAR2,
  deptno_out OUT NUMBER)
AS
BEGIN
  SELECT ename, deptno
    INTO ename_out, deptno_out
    FROM emp
   WHERE empno = empno_in;
END;
/
```

node-oracledb

- oracledb.getConnection(): 使用 async/await

```
const oracledb = require('oracledb');
const dbConfig = require('./dbconfig');

(async () => {
  const connection = await oracledb.getConnection(dbConfig);
  const result = await connection.execute('select user from dual');
  console.log(result);
  connection.close();
})();
```

oracledb.getConnection() 如沒有 callback 函數，會返回 Promise 物件。

EMACScript 7 開始支援 async/await，使用 await 語法一定得包在 asnyc 函數中，可以直覺像以同步的寫法一樣；上面是簡化的程式碼，完全沒有考慮到發生錯誤時該如何處理。

await 語法後面期待的是一個 Promise 物件，會等待 Promise resolved 或 rejected 再繼續往下執行。所以只要是 Promise 都可以套入 async/await 的語法中。

Node.js version 8 開始支援 async/await，而且強烈的依賴 Promise。

node-oracledb

- async/await Example

```
function getRandomNumber() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      const randomValue = Math.random();
      const error = randomValue > .5 ? true : false;

      if (error) {
        reject(new Error('糟糕，有些東西錯了！'));
      } else {
        resolve(randomValue);
      }
    }, 1000);
  });
}
```

```
async function logNumber() {
  let number;
  console.log('調用 await 前: ', number);
  number = await getRandomNumber();
  console.log('調用 await 後: ', number);
}

console.log('調用 async 前');
logNumber();
console.log('調用 async 後');
```

```
調用 async 前
調用 await 前:  undefined
調用 async 後
調用 await 後:  0.22622288106213384
```

Await 不會阻止主線程。如果 Promise 沒有發生錯誤時，輸出將如右邊所示；如果 Promise 被拒絕，則會在主線程上拋出被拒絕的值。而處理錯誤的方式就是將 await 包在 try ... catch 中。

```
調用 async 前
調用 await 前:  undefined
調用 async 後
(node:8784) UnhandledPromiseRejectionWarning: Error:
糟糕，有些東西錯了！
.....
(node:8784) UnhandledPromiseRejectionWarning:
Unhandled promise rejection.
```

node-oracledb

- async/await Example

```
async function logNumber() {
  let number;
  console.log('調用 await 前: ', number);

  try {
    number = await getRandomNumber();
  } catch(err) {
    console.log(err.message);
    number = 42;
  }

  console.log('調用 await 後: ', number);
}
```

```
調用 async 前
調用 await 前:  undefined
調用 async 後
糟糕，有些東西錯了！
調用 await 後:  42
```

Await 處理錯誤的方式就是將 await 包在 try ... catch 中。

node-oracledb

- oracledb.getConnection(): async/await

```
const oracledb = require('oracledb');
const dbConfig = require('./dbconfig');

(async () => {
  let connection = null;
  let result = null;

  try {
    connection = await oracledb.getConnection(dbConfig);
    result = await connection.execute('select user from dual');
    console.log(result);
  } catch(err) {
    console.log(err);
  } finally {
    if (connection) {
      connection.close();
    }
  }
})();
```

```
if (connection) {
  try {
    await connection.close();
  } catch (err) {
    console.log(err)
  }
}
```

node-oracledb

- oracledb.createPool

```
const oracledb = require('oracledb');
const dbConfig = require('./dbconfig');

(async () => {
  let pool = null;
  let connectin = null;
  let result = null;

  try {
    pool = await oracledb.createPool(dbConfig);
    console.log(pool.poolAlias);
  } catch (err) {
    console.log(err);
    return;
  }

  try {
    connection = await oracledb.getConnection();
    result = await connection.execute('select user
from dual');
    console.log(result);
  } catch(err) {
    console.log(err);
  } finally {
    if (connection) {
      try {
        await connection.close();
      } catch (err) {
        console.log(err)
      }
    }
  }

  try {
    await oracledb.getPool().close();
  } catch (err){
    console.log(err);
  }
})();
```

default

Use default pool

Close connection

Close pool

```
module.exports = {
  user: "demo",
  password: "demo426",
  connectString: "10.11.25.139:1522/lx26.pec.com.tw",
  poolMin: 3,
  poolMax: 3,
  poolIncrement: 0
};
```

RESTful Web Services

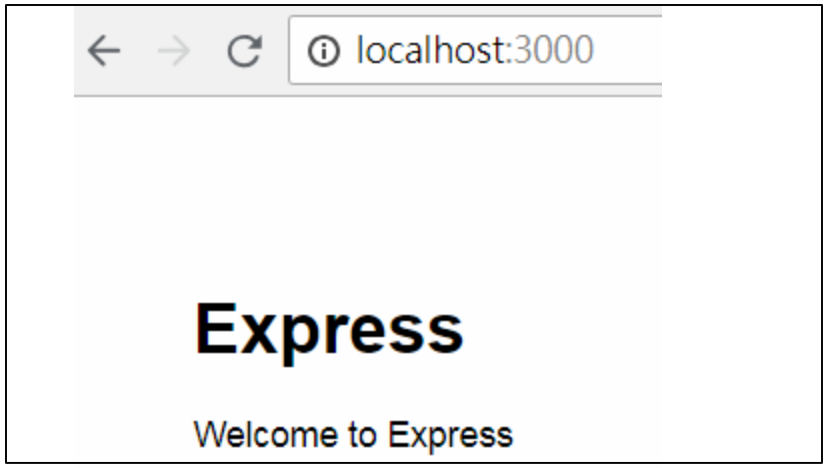
- RESTful Web Service 範例: Create Express APP

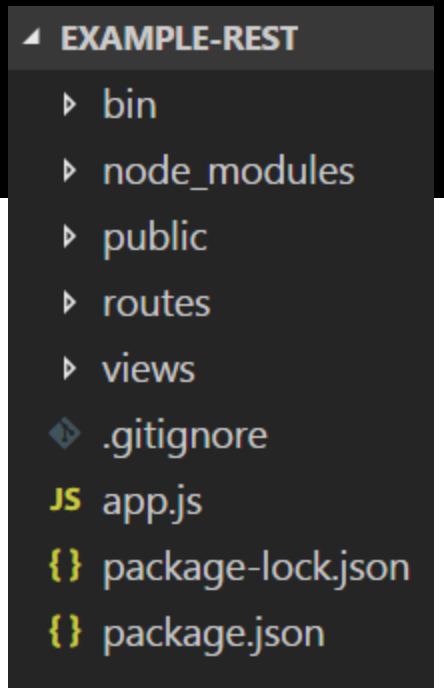
```
$ express --view ejs --git example-rest

   create : example-rest\
   create : example-rest\public\
   create : example-rest\public\javascripts\
   create : example-rest\public\images\
   create : example-rest\public\stylesheets\
   create : example-rest\public\stylesheets\style.css
   create : example-rest\routes\
   create : example-rest\routes\index.js
   .....

   change directory:
     > cd example-rest

   install dependencies:
     > npm install

   run the app:
     > SET DEBUG=example-rest:* & npm start
```

RESTful Web Services

- RESTful Web Service 範例: Install & Startup

```
$ cd example-rest

$ npm install
npm notice created a lockfile as package-lock.json. You should commit this file.
added 55 packages in 9.219s

$ npm start

> example-rest@0.0.0 start H:\NodeJS\Projects\example-rest
> node ./bin/www
```





RESTful Web Services

- RESTful Web Service 範例: Add route (routes/employees.js)

```
var express = require('express');
var router = express.Router();

/* GET employees listing. */
router.get('/', function(req, res, next) {
  res.status(200).json({message: "員工名單"});
});

module.exports = router;
```

修改 app.js，加入新增的 route，然後重新啟動服務。

```
var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
var employeesRouter = require('./routes/employees');
.....
app.use('/', indexRouter);
app.use('/users', usersRouter);
app.use('/v1/employees', employeesRouter);
```

RESTful Web Services

- RESTful Web Service 範例: 安裝 nodemon

```
$ npm install -g nodemon

$ nodemon --version
1.18.3
```

nodemon 是一個專為 Node.js 設計的模組，它會持續監視著你的程式碼，一旦你修改且保存後，會重新啟動你的的程式

修改 package.js，我們就可用 npm 直接啟動服務程序

```
"scripts": {
    "start": "nodemon ./bin/www"
  },
...
```

```
$ npm start

> example-rest@0.0.0 start H:\NodeJS\Projects\example-rest
> nodemon ./bin/www

[nodemon] 1.18.3
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node ./bin/www`
```

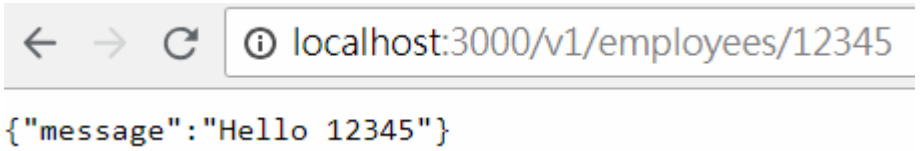
RESTful Web Services

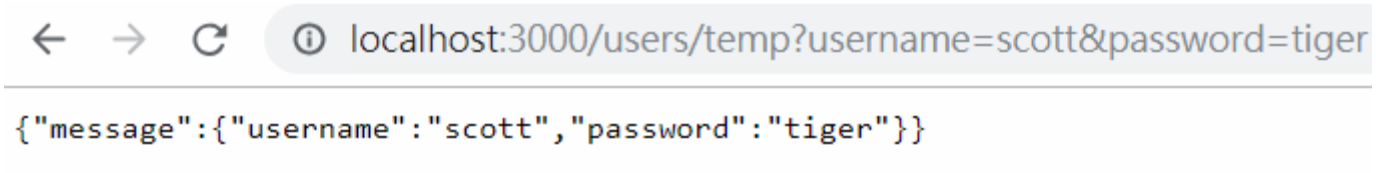
- RESTful Web Service 範例: Request 物件屬性 params

```
/* GET employee details */
router.get('/:empno', function(req, res, next) {
  const empno = req.params.empno || undefined;
  const message = `Hello ${empno}`;

  if (typeof empno !== 'string') {
    return res.status(400).json({message: "erros"});
  }

  res.status(200).json({message: message});
});
```

Template Strings

Request 物件提供很多的屬性，常用的有 url, params, url ...，Express 則另提供一些方便的屬性，例如 req.query 可以直接取的 url 的 query 字串。





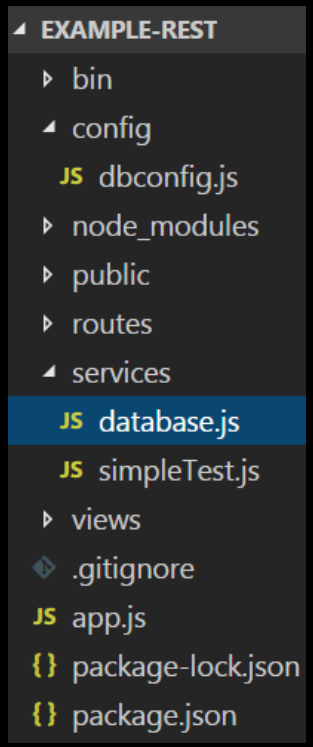
RESTful Web Services

- RESTful Web Service 範例: database.js (1/4)

```
$ npm install oracledb --save
```

```
const oracledb = require('oracledb');
const dbConfig = require('../config/config').oralx26;

async function initialize() {
  try {
    const pool = await oracledb.createPool(dbConfig);
  } catch(err) {
    console.log(err);
  }
}

async function close() {
  try {
    await oracledb.getPool().close();
  } catch(err) {
    console.log(err);
  }
}
```

```
// config/config.js
module.exports = {
  oralx26: {
    user: "demo",
    password: "demo426",
    connectString: "10.11.25.139:1522/lx26.pec.com.tw",
    poolMin: 1,
    poolMax: 2,
    poolIncrement: 1
  }
};
```



RESTful Web Services

- RESTful Web Service 範例: database.js (2/4)

```
function doExecute(statement, binds = [], opts = {}) {
  return new Promise(async (resolve, reject) => {
    let conn;

    opts.outFormat = oracledb.OBJECT;
    opts.autoCommit = true;

    try {
      conn = await oracledb.getConnection();
      const result = await conn.execute(statement, binds, opts);
      resolve(result);
    } catch (err) {
      reject({error: err});
    } finally {
      if (conn) {
        try {
          await conn.close();
        } catch (err) {
          console.log(err);
        }
      }
    }
  });
}
```

資料返回的格式
oracledb.ARRAY 或 oracledb.OBJECT

如果你不打算用 connection pool，則將 oracledb.getConnection() 改為 oracledb.getConnection(dbConfig)，並將 dbConfig 的 pool 設定去除。

RESTful Web Services

- RESTful Web Service 範例: database.js (3/4)

```
function doExecuteMany(statement, binds = [], opts = {}) {
  return new Promise(async (resolve, reject) => {
    let conn;

    opts.outFormat = oracledb.OBJECT;
    opts.autoCommit = true;
    opts.batchErrors = true;

    try {
      conn = await oracledb.getConnection();
      const result = await conn.executeMany(statement, binds, opts);
      resolve(result);
    } catch (err) {
      reject(err);
    } finally {
      if (conn) {
        try {
          await conn.close();
        } catch (err) {
          console.log(err);
        }
      }
    }
  });
}
```

這可一次執行多筆，binds 的格式將會是 [[...],[...],[...]]

RESTful Web Services

- RESTful Web Service 範例: database.js (4/4)

```
module.exports.initialize = initialize;
module.exports.close = close;
module.exports.doExecute = doExecute;
module.exports.doExecuteMany = doExecuteMany;
```

RESTful Web Services

- RESTful Web Service 範例: 簡單的查詢測試 simpleTest.js

```
const oradb = require('./database');

(async function() {
  const statement = 'select * from emp where empno = :empno';

  try {
    await oradb.initialize();
    const result = await oradb.doExecute(statement, [7654]);
    console.log(result);
  } catch (err) {
    console.log(err);
  } finally {
    try {
      await oradb.close();
    } catch (err) {
      console.log(err);
    }
  }
})();
```

```
$ node simpleTest
{ outBinds: undefined,
  rowsAffected: undefined,
  metaData: [ { name: 'ENAME' } ],
  rows: [ { ENAME: '葉習堃' } ] }
```

oracledb.ARRAY => rows: [['葉習堃']]

oracledb.OBJECT

RESTful Web Services

- RESTful Web Service　範例: 測試多筆資料更新 demo_employees.js

```
module.exports = [
  {"EMPNO":7839,"ENAME":"KING","JOB":"PRESIDENT","MGR":null,"HIREDATE":"1981-11-16T16:00:00","SAL":5000,"COMM":null,"DEPTNO":10},
  {"EMPNO":7698,"ENAME":"BLAKE","JOB":"MANAGER1","MGR":7839,"HIREDATE":"1981-04-30T16:00:00","SAL":2850,"COMM":101,"DEPTNO":30},
  {"EMPNO":7782,"ENAME":"楊瑞　","JOB":"MANAGER","MGR":7839,"HIREDATE":"1981-06-08T16:00:00","SAL":2400,"COMM":null,"DEPTNO":10},
  {"EMPNO":7566,"ENAME":"楊瑞珉","JOB":"MANAGER","MGR":7839,"HIREDATE":"1981-04-01T16:00:00","SAL":2975,"COMM":null,"DEPTNO":20},
  {"EMPNO":7788,"ENAME":"SCOTT","JOB":"ANALYST","MGR":7566,"HIREDATE":"1982-12-08T16:00:00","SAL":3002,"COMM":null,"DEPTNO":20},
  {"EMPNO":7902,"ENAME":"FORD","JOB":"ANALYST","MGR":7566,"HIREDATE":"1981-12-02T16:00:00","SAL":3000,"COMM":null,"DEPTNO":20},
  {"EMPNO":7369,"ENAME":"SMITH","JOB":"CLERK","MGR":7902,"HIREDATE":"1980-12-16T16:00:00","SAL":8001,"COMM":null,"DEPTNO":20},
  {"EMPNO":7499,"ENAME":"ALLEN","JOB":"SALESMAN","MGR":7698,"HIREDATE":"1981-02-19T16:00:00","SAL":1600,"COMM":303,"DEPTNO":30},
  {"EMPNO":7608,"ENAME":"馬小九","JOB":"ANALYST","MGR":7788,"HIREDATE":"2010-06-28T03:11:23","SAL":1000,"COMM":null,"DEPTNO":10},
  {"EMPNO":7654,"ENAME":"葉習堃","JOB":"SALESMAN","MGR":7698,"HIREDATE":"1981-09-27T16:00:00","SAL":1250,"COMM":1400,"DEPTNO":30},
  {"EMPNO":7844,"ENAME":"하찮고","JOB":"SALESMAN","MGR":7698,"HIREDATE":"1981-09-07T16:00:00","SAL":1500,"COMM":0,"DEPTNO":30},
  {"EMPNO":7876,"ENAME":"ADAMS","JOB":"CLERK","MGR":7788,"HIREDATE":"1983-01-11T16:00:00","SAL":1100,"COMM":null,"DEPTNO":20},
  {"EMPNO":7900,"ENAME":"JAMES","JOB":"CLERK","MGR":7698,"HIREDATE":"1981-12-02T16:00:00","SAL":950,"COMM":null,"DEPTNO":30},
  {"EMPNO":7934,"ENAME":"楊喆","JOB":"CLERK","MGR":7782,"HIREDATE":"1982-01-22T16:00:00","SAL":1500,"COMM":null,"DEPTNO":10},
  {"EMPNO":9006,"ENAME":"林羿君1","JOB":"ANALYST","MGR":7788,"HIREDATE":"2007-01-30T16:00:00","SAL":45300,"COMM":203,"DEPTNO":20},
  {"EMPNO":7607,"ENAME":"バック","JOB":"分析師","MGR":7788,"HIREDATE":"2008-03-24T01:33:15","SAL":45000,"COMM":100,"DEPTNO":40},
  {"EMPNO":7609,"ENAME":"蔡大一","JOB":"分析師","MGR":7566,"HIREDATE":"2010-06-28T03:12:46","SAL":10000,"COMM":10,"DEPTNO":20},
  {"EMPNO":9011,"ENAME":"文英蔡","JOB":"總鋪師","MGR":7788,"HIREDATE":"2018-08-28T02:43:08","SAL":23010,"COMM":null,"DEPTNO":40},
  {"EMPNO":8907,"ENAME":"牸祢","JOB":"ANALYST","MGR":7566,"HIREDATE":"1982-12-08T16:00:00","SAL":9002,"COMM":null,"DEPTNO":10}
];
```

這是測試資料存儲在 demo_employees.js

RESTful Web Services

- RESTful Web Service 範例: 測試多筆資料更新 simpleExec.js (1/2)

```
const oradb = require('./database');
const employees = require('../data/demo_employees');

let row = [];
let binds = [];

employees.forEach(value => {
  row = [];
  Object.keys(value).forEach(key => row.push(value[key]));
  binds.push(row);
});

(async function() {
  const statement = `
    MERGE INTO emp_temp e
    USING (select :empno as empno, :ename as ename, :job as job,
                  :mgr as mgr, to_date(:hiredate,'yyyy-mm-dd"T"hh24:mi:ss') as hiredate,
                  :sal as sal, :comm as comm, :deptno as deptno
             from dual) p
    ON (e.empno = p.empno)
    WHEN MATCHED THEN
      UPDATE SET e.ename = p.ename, e.job = p.job, e.mgr = p.mgr, e.hiredate = p.hiredate,
                 e.sal = p.sal, e.comm = p.comm, e.deptno = p.deptno
    WHEN NOT MATCHED THEN
      INSERT (e.empno, e.ename, e.job, e.mgr, e.hiredate, e.sal, e.comm, e.deptno)
      VALUES (p.empno, p.ename, p.job, p.mgr, p.hiredate, p.sal, p.comm, p.deptno)
  `;
```

RESTful Web Services

- RESTful Web Service 範例: 測試多筆資料更新 simpleExec.js (2/2)

```
  try {
    await oradb.initialize();
    try {
      const result = await oradb.doExecuteMany(statement, binds, opts);
      console.log(result);
    } catch (err) {
      console.log(err);
    }
  } catch (err) {
    console.log(err);
  } finally {
    await oradb.close();
  }
})();
```

```
$ node simpleExec
{ rowsAffected: 19 }
```

受影響的筆數

RESTful Web Services

- RESTful Web Service 範例: 這 g_merge_sql.js (1/2)可產生 SQL Merge 指令

```
const metadata = {
  table_name: 'pec_hrc_users_t',
  keys: ["id_no", "emp_no"],
  columns: [
    "id_no","emp_no", "emp_name", "quit_kind", "manager_empno", "manage_code",
    "unit_code", "purchase_above", "workfile_leader", "expense_dept", "expense_name","pri_code"
  ]
};

const statement = g(metadata);
console.log(statement);

// function
function g(metadata) {
  const tablename = metadata.table_name;
  const keys = metadata.keys;
  const columns = metadata.columns;

  if(!(columns instanceof Array))
    return console.log('Error: Parameter error, Array expected.');

  let onkeys = keys.reduce((prev, curr) => {
      return `${prev}${prev == '' ? '':' and '}t.${curr} = p.${curr}`;
    },'');
```

RESTful Web Services

- RESTful Web Service 範例: g_merge_sql.js (2/2)

```
  let select = columns.reduce((prev, curr) => {
      return `${prev}${prev == 'select ' ? '':','}:${curr} as ${curr}`;
    },'select ');
  let update = columns.filter( item => {
      return keys.indexOf(item) === -1;
    }).reduce((prev, curr) => {
      return `${prev}${prev == '' ? '':','} t.${curr} = p.${curr}`;
    },'');
  let insert = columns.reduce((prev, curr) => {
      return `${prev}${prev == '' ? '':','} t.${curr}`;
    },'');
  let values = columns.reduce((prev, curr) => {
      return `${prev}${prev == '' ? '':','} p.${curr}`;
    },'');
  let merge = `
    MERGE INTO ${tablename}
    USING (${select} from dual) p
       ON (${onkeys})
    WHEN MATCHED THEN
      UPDATE SET ${update}
    WHEN NOT MATCHED THEN
      INSERT (${insert})
      VALUES (${values})
`;
  return merge;
}
```

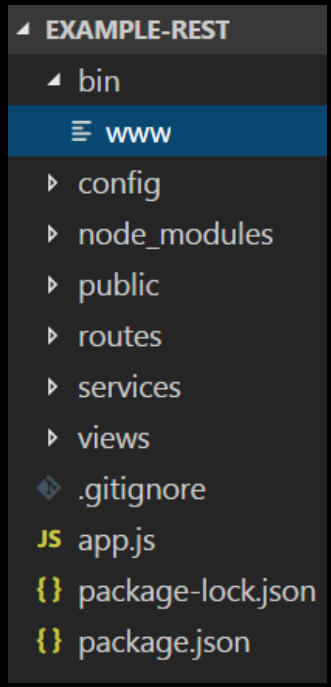
RESTful Web Services

- RESTful Web Service 範例: 修改 www (1/3)

```
async function dbInitialize() {
  try {
    console.log('Initializing database module');
    await database.initialize();
  } catch (err) {
    console.error(err);
    process.exit(1);
  }
}

async function dbClose(err) {
  let error = err;
  console.log('Close database connection pool');

  try {
    await database.close();
  } catch (err) {
    console.log("Encountered error", err);
    error = error || err;
  }
  if (error) {
    process.exit(1);
  } else {
    process.exit(0);
  }
}
```

如果不打算使用 Oracle connection pool, 則可不用修改此程序檔。

www 尾端加上 dbInitialize 與 dbClose 函數



RESTful Web Services

- RESTful Web Service 範例: 修改 www (2/3)

```
var app = require('../app');
var debug = require('debug')('example-rest:server');
var http = require('http');

/* Initialize Database connection pool */
const database = require('../services/database');
const dbConfig = require('../config/config').oralx26;
const defaultThreadPoolSize = 4;
process.env.UV_THREADPOOL_SIZE = dbConfig.poolMax + defaultThreadPoolSize;

dbInitialize();

.....
```

UV_THREADPOOL_SIZE 必須在使用 threadpool 的第一個調用之前設置。 這是因為 threadpool 是在第一次使用時創建的，一旦創建，它的大小是固定的。

www 前端調用 dbInitialize() 函數創建 database connection pool

```
DEMO      44   20 PEC\76070049P1  21764    INACTIVE   27983 node.exe            node.exe
          45   63 PEC\76070049P1  21766    INACTIVE   23061 node.exe            node.exe
          47  203 PEC\76070049P1  21768    INACTIVE    4447 node.exe            node.exe
```

在有非同步系統 API 的環境中，Node.js 會盡量用它的非同步 API，但如果在沒有非同步系統的環境中，則 Node.js 用 libuv 的 threadpool 來創建 Node.js 的非同步 API。因為 libuv 的 threadpool 有固定的大小，這意味著如果由於某種原因這些 API 需要很長時間處理，那麼在 libuv 的 threadpool 運行的其他 API 將會遇到性能下降的問題。為了緩解這個問題，一個可能的解決方案是通過將 UV_THREADPOOL_SIZE 環境變數設置為大於 4 的值（預設值）來增加 libuv 的 threadpool 的大小。

RESTful Web Services

- RESTful Web Service 範例: 修改 www (3/3)

```
process.on('SIGTERM', () => {
  console.log('Received SIGTERM');
  dbClose();
});
process.on('SIGINT', () => {
  console.log('Received SIGINT');
  dbClose();
});
process.on('uncaughtException', err => {
  console.log('Uncaught exception');
  console.log(err);
  dbClose(err);
});
process.on('unhandledRejection', (reason, promise) => {
  console.error("Unhandled Rejection at: ", promise, " reason: ", reason);
  dbClose(err);
});
```

www 最後加上幾個 Signal 監聽器,當 Node.js process 收到這些信號時，將會發出 Signal Events，收到這些事件時關閉 database connection pool。

```
[nodemon] 1.18.3
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node ./bin/www`
Initializing database module
Received SIGINT
要終止批次工作嗎 (Y/N)? y
```

當按下 CTRL-C 停止 APP Server 時

RESTful Web Services

- RESTful Web Service 範例: 修改 routes/employees.js (1/2)

```
var express = require('express');
const oradb = require('../services/database');
var router = express.Router();

/* GET employees listing. */
router.get('/', async function(req, res, next) {
  let error;
  let result = null;

  const statement = `select * from emp`;

  try {
    result = await oradb.doExecute(statement);
  } catch (err) {
    error = error || err;
  }

  if (error) {
    console.log(error);
    return res.status(401).json({status: "error", message: "Data not found"});
  } else {
    return res.status(200).json(result);
  }
});
```

RESTful Web Services

- RESTful Web Service 範例: 修改 routes/employees.js (2/2)

```
/* GET employee details */
router.get('/:empno', async function(req, res, next) {
  let error;
  let binds = [];
  let result = null;
  const statement = `select * from emp where empno = :empno`;
  const empno = req.params.empno || undefined;

  if (typeof empno !== 'string') {
    return res.status(404).json({message: "Data not found"});
  }

  binds.push(empno);

  try {
    result = await oradb.doExecute(statement, binds);
  } catch (err) {
    error = error || err;
  }
  if (error) {
    console.log(error);
    return res.status(404).json({message: "Data not found"});
  } else {
    if (result && result.rows.length > 0) {
      return res.status(200).json(result);
    } else {
      return res.status(404).json({message: "Data not found"});
    }
  }
});
```

RESTful Web Services

- RESTful Web Service Example: Modify Employees.js (POST)

```
/* POST Employees */
router.post('/', async function(req, res, next) {
  const data = req.body;
  const statement = `
    MERGE INTO emp e
    USING (select :empno as empno,
                  :ename as ename,
                  :job as job,
                  :mgr as mgr,
                  to_date(:hiredate,'yyyy-mm-dd"T"hh24:mi:ss') as hiredate,
                  :sal as sal,
                  :comm as comm,
                  :deptno as deptno
             from dual) p
    ON (e.empno = p.empno)
    WHEN MATCHED THEN
      UPDATE SET e.ename = p.ename,
                 e.job = p.job,
                 e.mgr = p.mgr,
                 e.hiredate = p.hiredate,
```

RESTful Web Services

- RESTful Web Service Example: Modify Employees.js (POST)

```
                e.sal = p.sal,
                e.comm = p.comm,
                e.deptno = p.deptno
  WHEN NOT MATCHED THEN
    INSERT (e.empno, e.ename, e.job, e.mgr, e.hiredate, e.sal, e.comm, e.deptno)
    VALUES (p.empno, p.ename, p.job, p.mgr, p.hiredate, p.sal, p.comm, p.deptno)
`;

if (typeof data !== 'object') {
  return res.status(400).json({message: "Unknown data"});
}

if (data instanceof Array && data.length > 0) {
  let error;
  let result;
  let row = [];
  let binds = [];
```

RESTful Web Services

- RESTful Web Service Example: Modify Employees.js (POST)

```
    data.forEach(value => {
      row = [];
      Object.keys(value).forEach(key => row.push(value[key]));
      binds.push(row);
    });

    try {
      result = await oradb.doExecuteMany(statement, binds);
    } catch (err) {
      error = error || err;
    }

    if (error) {
      console.log(error);
      return res.status(400).json({message: "Unknown data"})
    } else {
      console.log(`Employees upserted: ${JSON.stringify(result)}`);
      return res.status(201).json({status: "ok", message: `rowsAffected: ${result.rowsAffected}`});
    }
  } else {
    return res.status(400).json({message: "Unknown data"});
  }
});
```

RESTful Web Services

- RESTful Web Service Example: Modify Employees.js (POST)

```
$ curl -H 'Content-Type: application/json' \
       -X POST http://10.11.100.30:3000/v1/employees \
       -d @./demo_emp.json

{"status":"ok","message":"Rows upserted: 1"}
```

```
[
  {
    "empno": 9011,
    "ename": "文英蔡",
    "job": "總鋪師",
    "mgr": 7788,
    "hiredate": "2018-08-28T10:43:08",
    "sal": 23000,
    "comm": null,
    "deptno": 40
  }
]
```

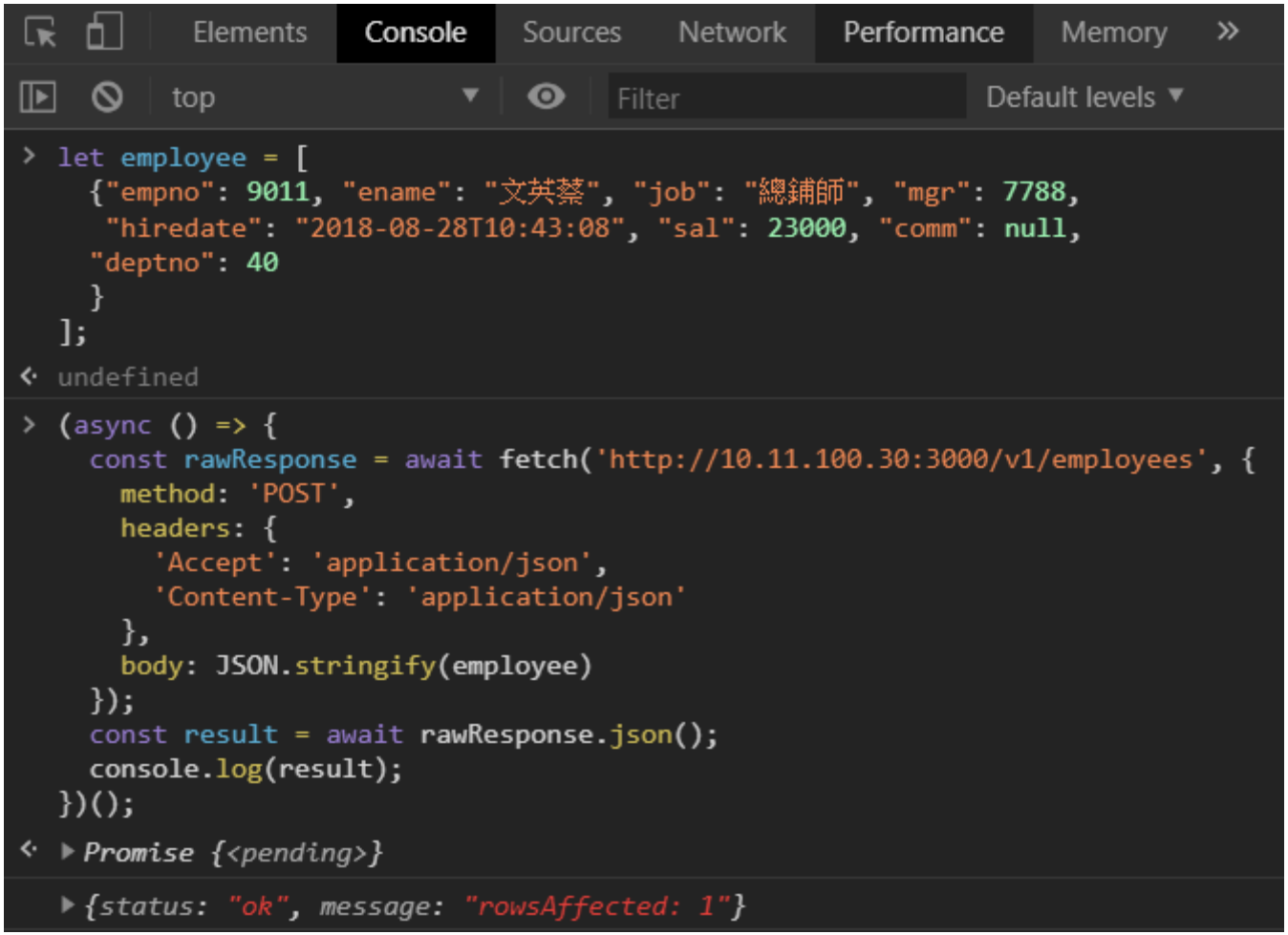
RESTful Web Services

- RESTful Web Service Example: Modify Employees.js (POST)

```
const employee = [
  {"empno": 9011, "ename": "文英蔡", "job": "總鋪師", "mgr": 7788,
   "hiredate": "2018-08-28T10:43:08", "sal": 23000, "comm": null,
   "deptno": 40
  }
];

(async () => {
  const rawResponse = await fetch('http://10.11.100.30:3000/v1/employees', {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(employee)
  });
  const result = await rawResponse.json();
  console.log(result);
})();
```

RESTful Web Services

- RESTful Web Service Example: Modify Employees.js (POST)



RESTful Web Services

- RESTful Web Service Example: CORS

```
undefined
(async () => {
  const rawResponse = await fetch('http://10.11.100.30:3000/v1/employees', {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(employee)
  });
  const result = await rawResponse.json();
  console.log(result);
})();
▶ Promise {<pending>}
Access to fetch at 'http://10.11.100.30:3000/v1/employees' from f?p=4500:1000:3905736601237:1
origin 'http://ords3.pec.com.tw:8080' has been blocked by CORS policy: Response to preflight
request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present
on the requested resource. If an opaque response serves your needs, set the request's mode to
'no-cors' to fetch the resource with CORS disabled.
▶Uncaught (in promise) TypeError: Failed to fetch                    VM84:12
```

RESTful Web Services

- RESTful Web Service Example: CORS

```
$ npm install cors --save
```

```
/* app.js */
.....
var logger = require('morgan');
var cors = require('cors');
.....
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
app.use(cors());
.....
```

```
(async () => {
  const rawResponse = await fetch('http://10.11.100.91:3000/v1/employees', {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(employee)
  });
  const result = await rawResponse.json();

  console.log(result);
})();
▶ Promise {<pending>}
▶ {status: "ok", message: "Rows upserted: 1"}
```

```
▼ Response Headers    view source
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 2680
Content-Type: application/json; charset=utf-8
Date: Wed, 29 Aug 2018 05:08:35 GMT
ETag: W/"a78-iDQvKA0/bj2/D26cbYjsHd0taP8"
X-Powered-By: Express
```

RESTful Web Services

- RESTful Web Service Example: Authorization JSON Web Token

```
$ npm install jsonwebtoken --save
```

```
const jwt = require('jsonwebtoken');
const jwtSecretKey = require('../config/config').jwtSecretKey;

let token = jwt.sign({id: 'someRoleId'}, jwtSecretKey);

console.log(`Token: ${token}`);

jwt.verify(token, jwtSecretKey, (err, payload) => console.log(payload));
```

```
$ node generateToken
Token:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6InNvbWVSb2xlSWQiLCJpYXQiOjE1NTI0NTU3MTd9.zk3cFaHl
ZYybHpf2brJYJgamzzdlpbuEXFjXuEboos8
{ id: 'someRoleId', iat: 1552455717 }
```

```
// config/config.js
module.exports = {
  oralx26: {
    ...
  },
  jwtSecretKey: "gIql2aCxjiyYfHQlZfDmCv6ezp3waoGAeqsi6ejZtQY"
};
```

RESTful Web Services

- RESTful Web Service Example: libs/auth.js

```
const jwt = require('jsonwebtoken');
const jwtSecretKey = require('../config/config').jwtSecretKey;

function auth(role) {
  return function (req, res, next) {
    let token;
    let payload;
    let _id = role || undefined;

    if (typeof _id !== 'string'){
      return res.status(401).send({ message: 'Unknown parameters.' });
    }

    if (!req.headers.authorization) {
      return res.status(401).send({ message: 'You are not authorized' });
    }
```

RESTful Web Services

- RESTful Web Service Example: libs/auth.js

```
    token = req.headers.authorization.split(' ')[1];

    try {
      payload = jwt.verify(token, jwtSecretKey);
      if (payload.id === _id) {
        next();
      } else {
        return res.status(401).json({ message: 'Authentication failed' });
      }
    } catch (e) {
      if (e.name === 'TokenExpiredError') {
        return res.status(401).json({ message: 'Token Expired' });
      } else {
        return res.status(401).json({ message: 'Authentication failed' });
      }
    }
  }
}

module.exports = auth;
```

RESTful Web Services

- RESTful Web Service Example: modify routes/employees.js

```
const express = require('express');
const oradb = require('../services/database');
const auth = require('../libs/auth');
const router = express.Router();

/* GET employees listing. */
router.get('/', auth('someRoleId'), async function(req, res, next) {
  let error;
  .....
```

RESTful Web Services

- RESTful Web Service Example: Authorization JSON Web Token

```
curl -H 'Content-Type: application/json' \
     -H 'Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6InNvbWVSb2xlSWQiLCJpYXQiOjE1NTI0NTYyOTN9.V4cPJHvn
FssCtPCbJirwZg8j9oBL_L-FIpJUoGKe4k4' \
     -X POST http://10.11.100.30:3000/v1/employees \
     -d @./demo_emp.json
```

```
(async () => {
  const rawResponse = await fetch('http://localhost:3000/v1/employees', {
    method: 'GET',
    headers: {
      'Content-Type': 'application/json',
      'Authorization': 'Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6InNvbWVSb2xlSWQiLCJpYXQiOjE1NTI0NTYyOTN9.V4cPJHvn
FssCtPCbJirwZg8j9oBL_L-FIpJUoGKe4k4'
    }
  });
  const result = await rawResponse.json();
  console.log(result);
})();
```

Node.js

- 非同步控制流程模式

在同步的程式設計風格，轉向 Node.js 這種基於延續傳遞風格與非同步 API 的平台，可能會很挫敗。撰寫非同步程式碼是完全不同的體驗，尤其是在控制流程的部份。簡單點的問題像是迭代處理一連串檔案、循序執行任務，或等待一連串操作完成，都需要開發者採取新的處理方式與技巧，才能避免寫出既沒效率又難以閱讀的程式碼。最常見的錯誤之一就是掉進「回呼地獄」問題的陷阱，以及大量的巢狀設計，讓程式碼肆無忌憚的水平擴增而非垂直成長，導至連簡單的日常處理都變得難以閱讀及維護。

延續傳遞風格(CPS)並非實作非同步 API 的唯一方式，ECMAScript 6 的承諾(promises)、產生器(generators)與 ECMAScript 7 的 async await 都是強大且富彈性的替代方案。

- 非同步程式設計的難處

在 JavaScript 裡確實很容易讓非同步程式碼失控。閉包與就地定義匿名函式可擁有流暢的程式設計體驗，讓開發者不需要突然切進程式碼的其它位置。這完美符合 KISS 原則，維持程式碼的流暢與簡單，並且不費很多時間就能得到可用的結果。然而，這也犧牲了模組性、可重複使用性與可維護性這類特性。遲早會引發失控繁衍的巢狀回呼，使函式體積無謂的增長，以及糟糕的程式碼組織。在多數時刻裡，產生閉包並不是出於功能上的需求，也不是特別針對非同步程式設計的問題，而是比較偏向於紀律議題。能夠體認到程式以經逐件的變得大而無當，或者，更好的情況是能夠預先知道它可能變得大而無當，進而選取最適切的解決方案來採取行動。了解某些設計模式預防這類問題，即是新手與專家的區別。

Node.js

- 非同步控制流程模式

我們從建立一個簡單的例子開始 ：

```
function waitFor(ms, callback) {
  setTimeout(() => {
    if (Math.random() < 0.2) {
      return callback(new Error(`wait for ${ms} ERROR!`));
    }
    callback(null, `wait for ${ms} result`);
  }, ms);
}
```

記得，這裡的 callback 使用的是 Node.js 的風格，錯誤先行。我們將用 waitFor() 函式來產生許多非同步的任務。

Node.js

- 非同步控制流程模式

```
const task1 = function(callback) {
  waitFor(800, callback);
}
const task2 = function(callback) {
  waitFor(1000, callback);
}
const task3 = function(callback) {
  waitFor(300, callback);
}
```

這裡產生了三個非同步的任務。

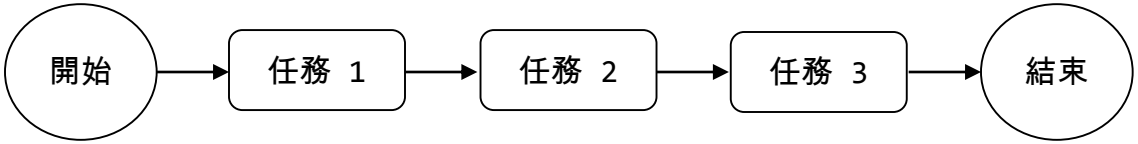
Node.js

- 非同步控制流程模式

  - 循序執行

  我們首先要開始探索非同步的控制流程模式，以分析循序執行作為起點。

  循序執行一組任務意即一次執行一個，之後再接另一個。執行的順序必須保持一致，因為前項任務的結果可能影響下一項任務的執行。



儘管使用直接風格的阻塞 API 時，循序執行不會有甚麼問題，但在使用非同步 CPS 時，它便經常成為回呼地獄的主要肇因。

Node.js

- 非同步控制流程模式

我們先使用一般的純 JavaScript 非同步的 CPS 模式，即回呼模式。

```
task1((err, data) => {
  if (err) {
    console.log(err.message);
  } else {
    console.log(data);
    task2((err, data) => {
      if (err) {
        console.log(err.message);
      } else {
        console.log(data);
        task3((err, data) => {
          if (err) {
            console.log(err);
          } else {
            console.log(data);
          }
        })
      }
    })
  }
});
```

Node.js

- 非同步控制流程模式

  - 回呼地獄

即使前面的程式碼，我們所使用的演算法相當直覺，但最終的程式碼還是存在一些內縮的層次，使其變得不易閱讀。而且使用的非同步 CPS，會牽涉到閉包的大量使用，如果閉包被錯誤使用將可能會導致極惡劣的程式碼。

大量的閉包與就地回呼定義，都會讓程式碼變得難以閱讀及管理，也就是所謂的回呼地獄。它是 Node.js 及 JavaScript 公認最嚴重的反模式之一。由於巢狀結構太深，無法輕易的追蹤到各個函式的結束處及起始處。

另一個問題則肇因於各作用範圍裡使用的變數名稱重疊。我們經常必須使用類似或者完全相同的名稱來描述變數內容，其中最常見的例子就是各個回呼所接收的錯誤參數，這常會導致混淆並提高產生程式缺陷的可能性。

我們也要留意，若使用閉包則在執行效率與記憶體消耗方面得付出一些代價。除此之外，它還會產生難以識別的記憶體洩漏，因為一個具作用的閉包所參考的所有環境，都會保留在垃圾回收區裡。

Node.js

- 非同步控制流程模式

藉由前面回呼模式循序執行流程，我們可以學到的簡單規則基本概念就如以下模式：

```
function task1(callback) {
  asyncOperation(() => {
    task2(callback);
  });
}

function task2(callback) {
  asyncOperation(() => {
    task3(callback);
  });
}

function task3(callback) {
  asyncOperation(() => {
    callback();
  });
}

task1(() => {
  // task1, task2, task3 完成
});
```

Node.js

- 非同步控制流程模式

基於此概念，我們可以修改前面的例子。

```
const task1 = function(callback) {
  waitFor(800, (err, data) => {
    if (err) {
      return console.log(err.message);
    }
    console.log(data);
    task2(callback);
  });
}

const task2 = function(callback) {
  waitFor(1000, (err, data) => {
    if (err) {
      return console.log(err.message);
    }
    console.log(data);
    task3(callback);
  });
}
```

Node.js

- 非同步控制流程模式

```
const task3 = function(callback) {
  waitFor(300, (err, data) => {
    if (err) {
      return console.log(err.message);
    }
    console.log(data);
    callback();
  });
}

// 開始執行
task1(() => {
  console.log('end');
});
```

上述模式說明在一般非同步操作完成時，各項任務呼叫下一項任務的方式。這個模式的重點在於模組化的任務，表示處理非同步程式碼不見得一定需要閉包。

Node.js

- 非同步控制流程模式

若預先知道有那些任務以及有多少任務要執行，上述模式可以完美運作，它可以直接寫死任務呼叫順序。但若想對一組集合裡的各個項目執行非同步操作呢？這種狀況可不能寫死任務順序，而必須動態的建置。

我們將使用循序非同步演算法，這個模式適用於所有需要非同步迭代一組元素或任務清單的情況，這個模式可概括如下：

```
function iterate(index) {
  if(index === tasks.length) {
    return finish();
  }
  const task = tasks[index];
  task(() => {
    iterate(index + 1);
  });
}

function finish() {
  //iteration completed
}

iterate(0);
```

Node.js

- 非同步控制流程模式

基於此概念，我們修改前面的例子。

```
const series = [1000, 800, 600, 400, 200, 900, 700, 500, 300, 100];
.....

function start(tasks, callback) {
  function iterate(index) {
    if (index === tasks.length) {
      return callback();
    }
    waitFor(tasks[index], (err, data) => {
      if (err) {
        console.log(err.message);
        return callback(err);
      }
      console.log(data);
      iterate(index + 1);
    });
  }
  iterate(0);
}

start(series, () => console.log('Done'));
```

Node.js

- 非同步控制流程模式

有一點必須留意，若 task() 為同步操作，則這類演算法就會變成真正的遞迴。在這種情況下，堆疊不會在每個循環週期都跳脫，並且可能還會遇到最大呼叫堆疊的限制。

這裡所展示的模式功能相當強大，因為它在很多情況下都適用。例如，我們可以映射(map)陣列值或傳遞操作結果給迭代裡的下一個元素，藉此實作出一種歸納演算法。我們也可以在遇到特殊情況時提早結束迴圈，甚至還可以迭代無窮數量的元素。

我們也可以選擇它包進擁有如下簽名的函式裡，進而通用化這項解決方案：

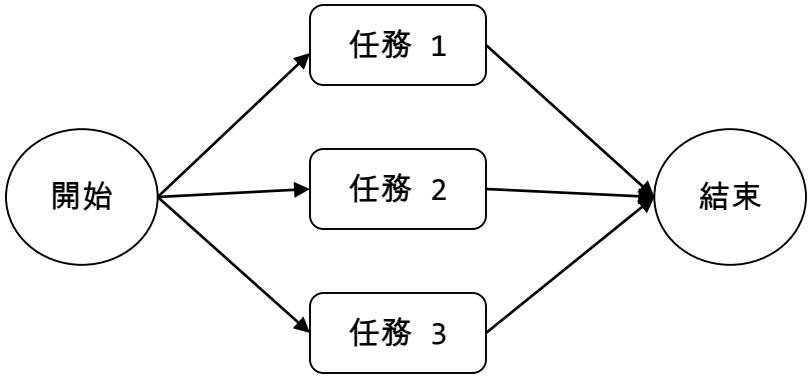
```
iterateSeries(collection, iteratorCallback, finalCallback);
```

Node.js

- 非同步控制流程模式

  - 平行執行

在某些情況下，一組非同步任務的執行順序並不重要，只要在所有任務都完成時能夠被通知就好。而這類情況則比較適合使用平行執行流程來處理，如下圖所示：



基於 Node.js 的非阻塞性質，即便只有一個執行緒，依然可以實現並行。**事實上，「平行」一詞在此並不適切，因為在此並不是指任務同時執行，而是藉由底層的非阻塞 API，在事件迴圈中交錯執行。**

如我們所知，當一項任務在要求新的非同步操作時會將控制權還給事件迴圈，藉此讓事件迴圈也能夠執行其他任務。這類流程比較適切的字眼是「**並行(concurrency)**」，不過出於簡化考量，我們還是稱之為「平行(parallel)」。

Node.js

- 非同步控制流程模式

在 Node.js 裡，我們只能平行執行非同步操作，因為它們的並行是由非阻塞 API 的內部負責處理的。在 Node.js 裡，同步(阻塞)操作無法並行，除非以非同步操作讓它們交錯執行，或者以 setTimeout() 或 setImmediate() 延遲。

對於平行執行流程，我們可以抽取出一個模式，它能夠針對不同情況改編或重複使用。如下程式碼即呈現此模式的通用版本：

```
const tasks = [...];

let completed = 0;
tasks.forEach(task => {
  task(() => {
    if(++completed === tasks.length) {
      finish();
    }
  });
});

function finish() {
  //all the tasks completed
}
```

Node.js

- 非同步控制流程模式

基於此模式，以下例子。

```
const series = [1000, 800, 600, 400, 200, 900, 700, 500, 300, 100];
.....

function start(tasks, callback) {
  let completed = 0, errored = false;

  function done(err, data) {
    if (err) {
      errored = true;
      return callback(err);
    }
    console.log(data);
    if(++completed === tasks.length && !errored) {
      return callback(null, 'Done');
    }
  }

  tasks.forEach(task => {
    waitFor(task, done);
  });
}
```

Node.js

- 非同步控制流程模式

```
start(series, (err, data) => {
  if (err) {
    return console.log(err.message);
  }
  console.log(data);
});
```

這個模式只要一點小修改，只需要一個變數 completed 計算回呼被成功呼叫的次數。至於如果回呼有錯誤，則可視需求修改。

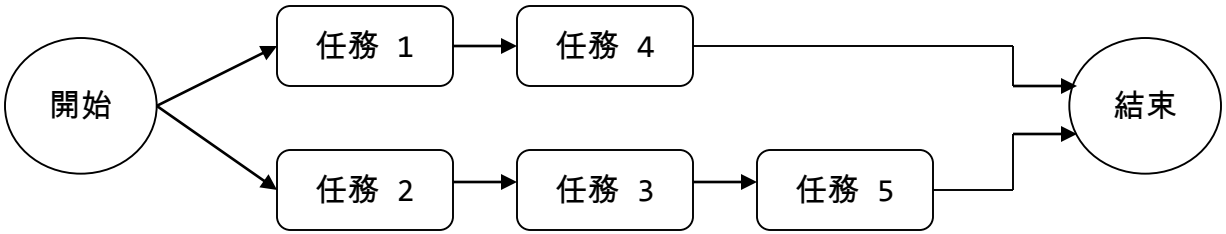
當使用阻塞 I/O 與多重執行緒，來平行的執行一組任務可能會讓人很頭痛。然而，我們已經知道在 Node.js 裡並非如此，平行執行多項非同步任務其實相當直覺，而且資源消耗也很低廉。這是 Node.js 最重要的優勢之一，因為它讓平行化成為普及的實作方式，而非僅僅是在確實需要時才能使用的複雜技巧。

Node.js

- 非同步控制流程模式

  - 限制平行執行

持續升成多個平行任務，若未加以控制，經常會導至超載問題。如果一次進行數千個檔案讀取、URL 存取或資料庫查詢，最常見的結果就是耗盡資源。例如，開啟過多檔案，以致於用盡應用程式所有可用的檔案描述項。而在網頁應用程式裡，這也有可能會導致**阻斷式服務(Denial of Service；Dos)**攻擊的風險。因此在所有類似狀況下，最好都限制可同時執行的任務數量。我們可以為伺服器的合理負載先做出預測，藉此確保應用程式不至於耗盡資源。下圖說明我們擁有 5 項平行執行的任務，而並行限制為 2 的情況：



上圖應該很清楚表達出我們的演算法會如何運作：

1. 一開始，在不超出並行限制下，盡可能生成多個任務。
2. 接著，當每次有一項任務完成時，在未達任務數量限制的前提下，可以再生成其它任務。

Node.js

- 非同步控制流程模式
  - 限制並行

下面展示的模式，是在有限的的並行下，平行執行一組指定數目的任務：

```
const tasks = [...];

let concurrent = 2, running = 0, completed = 0, index = 0;
function next() {
  while(running < concurrent && index < tasks.length) {
    task = tasks[index++];
    task(() => {
      if (++completed === tasks.length) {
        return finish();
      }
      running--;
      next();
    });
    running++;
  }
}
next();

function finish() {
  // all tasks finished
}
```

Node.js

- 非同步控制流程模式
  - 限制並行

```
let concurrency = 2, running = 0, completed = 0, index = 0;
function next() {
  while(running < concurrency && index < tasks.length) {
    waitFor(tasks[index++], (err, data) => {
      if (err) {
        console.log(err.message);
      } else {
        console.log(data);
      }
      if (++completed === tasks.length) {
        return finish();
      }
      running--;
      next();
    });
    running++;
  }
}
next();

function finish() {
  console.log('all tasks finished');
}
```

Node.js

- 非同步控制流程模式

  - 限制並行

限制平行的數量，我們也可以利用另一項機制:**佇列(queues)**，來限制多項任務的並行。以下就來了解它的運作方式:

我們現在要實作一個名為 TaskQueue 的簡單類別，讓我們的演算法能夠結合佇列功能。現在建立名為 taskQueue.js 的新模組，然後從定義它的建構子開始:

```
module.exports = class TaskQueue {
  constructor(concurrency) {
    this.concurrency = concurrency;
    this.running = 0;
    this.queue = [];
  }
```

這個建構子只接收一個參數作為並行限制，除此之外，它還會初始化其它所需要的實例變數。

Node.js

- 非同步控制流程模式
  - 限制並行

```
  pushTask(task) {
    this.queue.push(task);
    this.next();
  }

  next() {
    while(this.running < this.concurrency && this.queue.length) {
      const task = this.queue.shift();
      task(() => {
        this.running--;
        this.next();
      });
      this.running++;
    }
  }
};
```

Node.js

- 非同步控制流程模式
  - 限制並行

你或許注意到了，這個方法有點類似先前限制並行的模式。它基本上是從佇列啟動多項任務，但不超出並行限制。當每項任務完成時，便更新 running 任務計數，然後再次呼叫 next() 啟動其它任務。TaskQueue 類別能夠動態增加新項目到佇列裡，並且擁有一個管控中心，使所有函式執行實例都受控於共同的並行限制。

```
const TaskQueue = require('./tasksQueue');
const runQueue = new TaskQueue(2);
const tasks = [1000, 800, 600, 400, 200, 900, 700, 500, 300, 100];

function waitFor(ms, callback) {
  setTimeout(() => {
    if (Math.random() < 0.2) {
      return callback(new Error(`wait for ${ms} ERROR!`));
    }
    callback(null, `wait for ${ms} result`);
  }, ms);
}
```

Node.js

- 非同步控制流程模式
  - 限制並行

```
let completed = 0, hasErrors = false;
tasks.forEach(value => {
  runQueue.pushTask(done => {
    waitFor(value, (err, data) => {
      if (err) {
        hasErrors = true;
        console.log(err.message);
      } else {
        console.log(data);
      }
      if (++completed === tasks.length) {
        finish();
      }
      done();
    });
  });
});

function finish() {
  console.log('all tasks finished.');
}
```

Node.js

- 非同步控制流程模式
  - 限制並行

這個函式的實作相當簡單，而且非常類似無窮平行執行的演算法。我們將並行控制權委派給 TaskQueue 物件，而唯一要作的事就是檢查所有任務皆已完成。

你也可以將此程序包在 CPS 函式中，當所有任務皆已完成則呼叫你的回呼函式：

```
function run(callback) {
  let completed = 0, hsaErrors = false;
  ....
  ....
  if (++completed === tasks.length && !hasErrors ) {
    callback();
  }
  ....
}

run(() => console.log('all tasks finish'));
```

Node.js

- Redis Geo 類型

  隨著智能手機的普及，基於地理位置的服務變得越來越受歡迎。Redis 從 3.2 版開始正式引入 Geo(地理位置)相關的 API，用於支持存儲和查詢這些地理位置相關場景中的座標。

```
const redis = require('redis');
const client = redis.createClient({
  host: '10.11.25.137',
  password: 'iscat',
  db: 0
});
```

Node.js

- Redis Geo 類型

```
restaurants = [
  {name: 'home', longitude: 120.233599 , latitude: 22.995926},
  {name: '老四川巴蜀麻辣燙', longitude: 120.231207 , latitude: 22.989103 },
  {name: '伊都日本料理', longitude: 120.229222 , latitude: 22.990931 },
  {name: '台南老爺行旅', longitude: 120.233347 , latitude: 22.989967},
  {name: '南紡購物中心', longitude: 120.233036 , latitude: 22.990974},
  {name: '阿官火鍋', longitude: 120.228230 , latitude: 22.992096},
  {name: '童樂島親子餐廳', longitude: 120.230976, latitude: 22.993017},
  {name: '覺丸拉麵', longitude: 120.229882 , latitude: 22.996256 },
  {name: '銀川日式料理', longitude: 120.228020 , latitude: 23.002491 },
  {name: '王品牛排', longitude: 120.234468 , latitude: 22.995926}
];

restaurants.forEach(restauant => {
  client.geoadd('restaurants:tainan', restauant.longitude, restauant.latitude,
restauant.name, redis.print);
});
```

Node.js

- Redis Geo 類型

```
client.geodist('restaurants:tainan', 'home', '南紡購物中心', 'm',
  (err, result) => {
    console.log(`home 到 南紡購物中心 距離 ${result} 公尺`);
});

client.georadius('restaurants:tainan', 120.233599, 22.995926, '400', 'm',
  (err, result) => {
    console.log(`目前這個位置 120.233599, 22.995926(例手機) 半徑 400 公尺內的餐
廳 => ${result}`);
});

client.georadiusbymember('restaurants:tainan', '南紡購物中心', '500', 'm',
  (err, result) => {
    console.log(`南紡購物中心半徑 500 公尺內的餐廳 => ${result}`);
});
```

Node.js

- Redis Geo 類型

```
// show all
client.zrange('restaurants:tainan', 0, -1, (err, result) => {
  result.forEach(restaurant => {
    client.geopos('restaurants:tainan', restaurant, (err, result) => {
      console.log(`${restaurant} ${result}`);
    });
  });
});
```

```
// remove member
client.zrem('restaurants:tainan', 'home', redis.print);
```