

TB

# TechBridge 技術共筆部落格

var topics = ['Web前後端', '行動網路', '機器人/物聯網', '數據分析', '產品設計', 'etc.']

[首頁](#) / [關於我們](#) / [技術週刊](#) / [粉絲專頁](#) / [訂閱RSS](#)



2018-04-01 | ethereum, smart contract, dapp

## 在 Ethereum 上開發簡單的 Todo App

Like 7.3K   Share

### 前言

近一兩年區塊鏈的技術造成一股熱潮，由於加密貨幣在投資上的炒作，常看到的區塊鏈範例都是和虛擬貨幣相關連的服務，另外，技術

的應用焦點也常放在金融和會計業上，這是因為這兩個行業長久以來在市場上都有球員兼裁判的嫌疑（例如會計有資產信用背書和被雇用人這兩種矛盾的角色），因此需要一個「備受信任」的機制來重拾市場對它們的信任。而區塊鏈，或是廣義上來說的分散式賬本技術，正是一個有潛力的解法。不過就像《區塊鏈革命》這本書所說的，這個技術實現了「價值上的轉移」，理當會對更多的行業帶來影響。

Ethereum 提出 DApp(Decentralized App) 的想法，也就是藉著區塊鏈上佈署的智慧合約（Smart Contract）對區塊鏈資料進行操作，使得不但資料無法被竄改，連合約（程式碼）本身也無法被竄改，因此這些合約可以在沒有第三方（例如律師）的見證下具有信用。以下簡單的將傳統 web app 和 DApp 作類比：

—	—	—	—
Web app	front-end	back-end API	Database
DApp	front-end	smart contract	Blockchain(Ethereum)

由於這篇只會提到開發 Todo 程式的必要部分，如果需要更完整對於智慧合約的介紹和 Ethereum 的相關知識可以參考 Taipei Ethereum Meetup 的[部落格](#)或是 gasolin 網友編寫的[gitbook](#)。

另外，雖然這裡只是實作一個簡單的 Todo 程式，但是可以想像如果擴展成接案平台的核心，發包人和接案人商量一個完成任務的評估方式和報酬，例如完成幾個測試後會收到多少款項，接著將剛剛的合約寫入 DAapp，未來只要接案人的專案通過測試後就能自動完成收款。

以下對 DApp 的實作利用 Truffle 這個開發框架，以 Solidity（Ethereum 官方開發的編程語言）開發智慧合約，並利用 javascript 的 web3.js 套件和合約溝通，以下分別就這兩個工具介紹。另外如果使用 VSCode 開發，可以安裝 Solidity 的[開發工具](#)，方便檢查語法上的問題。

這是這篇所使用到的[程式碼](#)

# Truffle - Ethereum Development Framework

Truffle 是 Ethereum（以太坊）的開發框架，可以建立測試用的區塊鏈，並將寫好的智慧合約編譯佈署，由於以太坊的開發環境變動的相當快，因此務必注意到不同版本的支援問題。這裡使用最新的 truffle 版本(v4.1.3)，因為這個版本建立的測試用區塊鏈支援 websocket 連線，配合 web3.js v1.0 之後的版本可以利用 socket 監聽事件的觸發，這是比較有效率的作法。

## Step1. 安裝 truffle

```
1 npm install -g truffle@4.1.3
```

## Step2. 建立專案資料並初始化

```
1 mkdir EthereumTodo && cd EthereumTodo
2 truffle init
3 # 資料夾結構如下
4 # build/          合約編譯完才會產生，這裡會生成描述合約的 json 檔，
5 # contracts/      合約的檔案，以 .sol 為結尾
6 # migrations/     描述如何將合約佈署到區塊鏈
7 # test/           用來測試合約
8 # truffle.js      設定 truffle
```

## Step3. 查看版本資訊

```
1 truffle version
2 # Truffle v4.1.3 (core: 4.1.3)
3 # Solidity v0.4.19 (solc-js)
```

請務必先檢查支援的 Solidity 版本，由於目前這個語言變動很快，需要更加注意版本間的語法差異。

## Step4. 建立測試用區塊鏈

```
1 # ganache-cli 是 truffle 內附的指令，用來取代原先的 testrpc
2 # 用 --seed apple banana cherry，指定隨機生成的種子，這樣可以確
3 # 如此一來合約的佈署位址也會相同，這在測試環境上非常好用
4 ganache-cli --seed apple banana cherry
```

```
5
6  ## 執行結果會順便產生測試用的帳號和 key，帳號會在之後執行合約時用到
7  # Available Accounts
8  # =====
9  # (0) 0x1d489c3f8ed5ee71325a847888b2157c9ac29c05
10 # ...
11 # Private Keys
12 # =====
13 # (0) bea70301d065cf7946f25251c73dbfff93d4320715e43bdc0d5
14 # ...
15 # Listening on localhost:8545
```

## Step5. 設定 truffle 環境

```
1  // truffle.js
2  module.exports = {
3    networks: {
4      development: {
5        host: "localhost",
6        port: 8545,      // default port of ganache-cli
7        network_id: "*" // Match any network id
8      }
9    }
10  };
```

## 建立 Todo 合約

### Step1. 設定資料結構

```
1  // contracts/ToDoFactory.sol
2  // 指定編譯的版本
3  pragma solidity ^0.4.17;
4
5  // 指定合約的名稱，之後佈署或是測試時都是根據這個名稱（不是檔案名稱）
6  contract ToDoFactory {
7    struct Todo {
8      string taskName;
9      bool isCompleted;
10     bool isValid;
11   }
12
13   Todo[] todos;
14 }
```

以上簡單的設定在合約中資料儲存的方式，利用 `struct` 包裹每個 `todo` 應該包含的資料，並利用陣列儲存。Solidity 常用的型別包含 `int(uint, uint256), uint8, bool, address(8 bytes), byte`，而 `string` 相當於是 `byte[]`。由於儲存資料在區塊鏈上相當耗費成本（以 POW 機制來說，就是需要有人挖礦），智慧合約的執行上也必須消耗 **gas**，因此會盡可能的選用適當的型別以減少寫入的資料量和運算量，另外在 `struct` 中盡可能的將相同的資料型別排列在一起，也可以節省儲存的資料量。

除了 `Array` 之外，Solidity 中常用的還有 `mapping`，代表 key-value 之間的對應。例如

```
1 // 由 id(int) 對應到 todo(Todo)
2 mapping(int => Todo) idMapTodo;
```

## Step2. 加上操作資料的 Function(Method)

```
1 contract TodoFactory {
2
3     function addTodo(string _taskName) public {
4         Todo memory todo = Todo(_taskName, false, true);
5         uint todoId = todos.push(todo) - 1;
6     }
7 }
```

在 `addTodo` 中，我們先以輸入的 `_taskName` 初始化一個型態是 `Todo` 的物件，接著加進合約中的 `todos` 陣列，並以陣列索引當作 `id`。注意到上面的函式即使加上回傳值也無法回傳預期的結果，之後會再解釋這部份。

Solidity 中 `contract` 和 `function` 的關係，可以類比成 `class` 和 `function` 的關係，`contract` 也是可以被繼承的，而 `function` 可以加上一些 **modifier**，例如：

- `public`: 代表可以被外界調用
- `private`: 代表只能被此合約中的其他 `function` 調用

- `internal`: 代表可以被此合約和繼承的合約調用，像是 C++ 的 `protected`
- `external`: 代表只能被外界調用或是
- `view`: 代表此 function 不會對區塊鏈上的資料作任何的更改，像是 get function
- `pure`: 代表此 function 不會操作到區塊鏈上的任何資料，所以 pure function 不會消耗任何的 **gas**，可以想像這就是 util function

另外需要特別注意的是，Solidity 中執行函式的方式被分成 **Call** 和 **Transaction** 兩種（雖然程式碼都是 function）

- **Call**: 代表執行函式但是不會對區塊鏈作任何的修改，可以使用「回傳值」，通常會被這樣使用的函式包含 `view` 或是 `pure` 這兩個關鍵字，如果對於一個有寫入的函式使用 call 的方式執行，結果不會寫入任何的資料
- **Transaction**: 和 Call 相反，在執行上會寫入資料，因此需要等待礦工們將資料寫入，所以函式的「回傳值」僅代表 transaction hash，不會回傳預期的結果。

以上可以參考這個討論串

除了預設的關鍵字，也可以利用關鍵字 `modifier` 宣告自訂的 **modifier**，類似 python 或是 js ES7 中的 decorator（裝飾字），例如

```
1  contract TodoFactory {
2
3      modifier isValidTodo(uint _todoId) {
4          // require 要求傳入的參數要是 true，否則中止操作，並退還執行
5          // 以下連結有更詳細的比較和說明
6          // https://medium.com/taipei-ethereum-meetup/比較-require
7          require(isTodoValid(_todoId));
8          _; // 這是語法上必要的
9      }
10 }
```



```
11     function deleteTodo(uint _todoId) public isValidTodo(_t
12         todos[_todoId].isValid = false;
13     }
14 }
```

在上述的程式碼中，首先利用自訂的 **modifier** 檢查想要刪除的元素是否有效，有效才會刪除。回到前面新增物件的部分，把陣列索引當作 **id** 是很奇怪的作法，因為實作上如果把陣列的元素移除，可能會想用其他的元素填補這個空隙，以節省儲存空間，這樣索引就會被更改，例如 [a, b, c] 移除 b 後會變成 [a, c]，但是因為區塊鏈的寫入成本極高，因此當刪除陣列元素時，不應該搬移其他的元素，所以直接將這個元素設定成 **invalid** 是成本較低的作法，當然也可以用 `delete` 刪除，但是會造成空隙。

另外 Solidity 中的函式可以回傳複數的值，回傳時類似 tuple 以 `()` 包裹，不過回傳的資料型別只能是原始的資料型別或是陣列，也就是說不能回傳 `struct` 或者是 `string[]`。

### Step3. 加上 event 標示已完成的事件

如同前面提到的，執行 `addTodo`, `deleteTodo`, `completeTodo` 的時候都會以 **Transaction** 的方式執行（不然不會寫入資料），因此為了讓其他人知道執行完畢，並收到執行的結果，必須使用 **event** 觸發的方式。

```
1  contract TodoFactory {
2
3      event OnTodoAdded(uint todoId);
4
5      function addTodo(string _taskName) public {
6          Todo memory todo = Todo(_taskName, false, true);
7          uint todoId = todos.push(todo) - 1;
8
9          // trigger event:
10         // 這樣才能取得原來的回傳值 todoId
11         // 在 v0.4.21 之後，必須寫成 emit OnTodoAdded(todoId)
12         OnTodoAdded(todoId);
13     }
14 }
```

`event` 的宣告就像是函式的 `header`，利用 `event` 可以讓多個 `client` 監聽智慧合約的變化，而且這些 `event` 一旦被觸發就會被紀錄在區塊鏈裡面，未來可以很輕易的查詢過去發生過的紀錄

## Step4. 編譯及佈署合約

原則上就是照抄 `migrations/1_initial_migration.js`

```
1 // migrations/4_deploy_todoFactory.js
2 const TodoFactory = artifacts.require('TodoFactory');
3
4 module.exports = function (deployer) {
5   deployer.deploy(TodoFactory);
6 }
```

接著依序編譯和佈署合約到自建的測試區塊鏈上

```
1 truffle compile
```

執行結果會在 `build/` 建立 `TodoFactory.json`

```
1 truffle migrate
2
3 ## 執行結果
4 # Running migration: 4_deploy_todoFactory.js
5 #   Deploying TodoFactory...
6 #   ... 0xfecf0206d68c496cf067e320a4d4b5d294dfe89979552f7
7 #   TodoFactory: 0x21e4624c5a0b3fda81d0833d412dded2bb3a7a
8 # Saving successful migration to network...
9 #   ... 0x6f592087ebfa7d5d77cce3f82c9d1222148c25499c348a5
10 # Saving artifacts...
```

其中 `0x21e4624c5a0b3fda81d0833d412dded2bb3a7a7c` 就是合約部署在區塊鏈上的位址

## Step5. 測試



由於已經佈署在區塊鏈上的合約無法再被修改，最多只能利用事先設定的函數調整參數，因此每一次的合約更新都會導致地址的改變，在實際的應用上代表著每次合約的更新都需要改變利用到的合約位址，這是非常麻煩的事情，因此佈署前的測試相當重要，另外測試對於 TDD (Test Drive Development) 或是 CI,CD 的流程也是不可或缺的。truffle 內建測試用的框架，利用 Mocha 和 Chai 這兩個在 javascript 中常用的套件（兩者所使用的版本可以從 [ganache-core](#) 查看）

以下的測試可以驗證合約是否正常發佈

```
1  // test/test_todoFactory.js
2  const TodoFactory = artifacts.require('TodoFactory');
3
4  contract('TodoFactory', function(accounts) {
5    before(async () => {
6      // 在所有測試開始前佈署合約
7      contract = await TodoFactory.deployed();
8    });
9
10   it('Should contract deployed properly', () => {
11     // 驗證合約是否已經被佈署
12     assert.isDefined(contract);
13   });
14 }
```

對於非同步的情形而言，可以使用 Promise 或者是 Callback 的語法，這裡用 Promise 配合 async-await 比較簡潔。不過因為目前 ganache-core 所使用的 chai 版本為 3.5，無法抓到非同步的錯誤，因此如果需要這方面的測試可以用以下的寫法

```
1  contract('TodoFactory', function(accounts) {
2    it('Should not complete invalid task', async () => {
3      contract.completeTodo(9527, (err) => {
4        assert.isDefined(err);
5      })
6    });
7  }
```

另外在測試的部分 **truffle** 對於 **Call** 和 **Transaction** 兩種執行方式並沒有區分，都是用一般函式的呼叫方式，差別在後者回傳的是 **transaction hash**，從 **hash** 可以取得觸發的 **event** 的資訊

```

1  // test/utils.js
2  function getEvents (tx, filter) {
3    const logs = tx.logs;
4    const events = _.filter(logs, filter);
5    return events;
6  }
7
8  // test/test_todoFactory.js
9  contract('TodoFactory', function(accounts) {
10    it('Should add new todo properly', async () => {
11      // addTodo 的呼叫是 Transaction 所以即使 addTodo 中有回傳
12      const tx1 = await contract.addTodo(Todo1.taskName);
13      const events1 = utils.getEvents(tx1, { event: 'OnTodo
14      todoId1 = events1[0].args.todoId;
15    });
16  }

```

另外如果要檢查執行時觸發的 **event** 可以參考 **stackoverflow** 上的討論來驗證

```

1  // test/utils.js
2  function assertEvent(contract, filter) {
3    return new Promise((resolve, reject) => {
4      const event = contract[filter.event]();
5      // event.watch, event.get, event.stopWatching
6      // 在 web3 中也有對應的 function 來監聽區塊鏈上的事件
7      event.watch();
8      event.get((error, logs) => {
9        const log = _.filter(logs, filter);
10        if (!_.isEmpty(log)) {
11          resolve(log);
12        } else {
13          reject(new Error("Failed to find filtered event f
14        }
15      });
16      event.stopWatching();
17    });
18  },
19
20  // test/test_todoFactory.js
21  contract('TodoFactory', function(accounts) {
22    it('Should delete todo properly', async () => {

```

```
22     it( 'should delete todo properly , async () => {  
23         await contract.deleteTodo(todoId1);  
24  
25         await utils.assertEvent(contract, { event: 'OnTodoDel  
26     });  
27 }
```

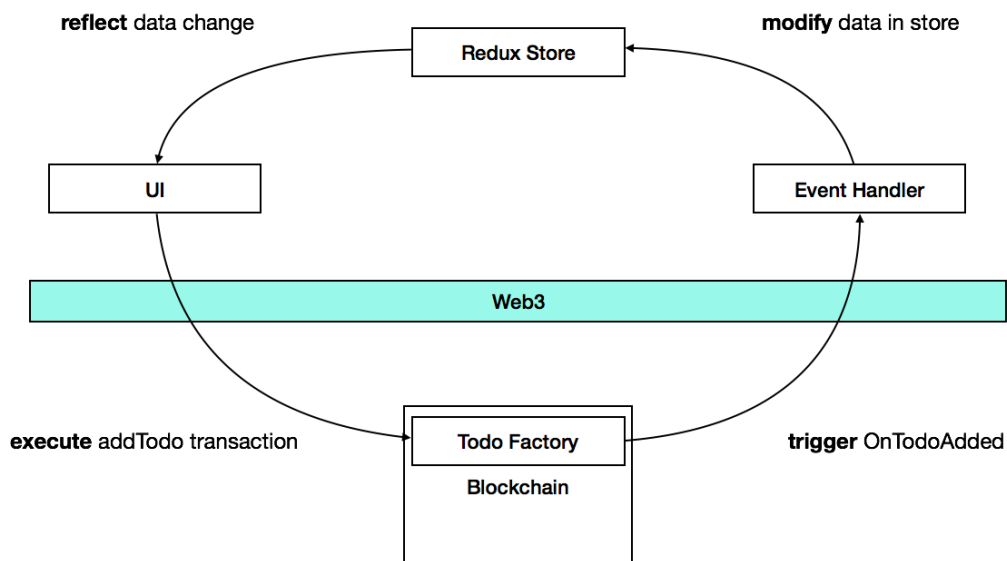
接著執行以下指令後可以得到測試的結果

```
1  truffle test
```

講完 DApp 「後端」 的部分後，接下來是利用前端來和智慧合約互動

## Web3.js

Web3.js 提供 javascript 用來和以太坊互動的 API，這邊使用的版本是 v1.0，v1.0 與之前的版本有相當大的差別，除了額外提供 Socket 接口監聽事件，API 的呼叫方式也完全不同，甚至有些連運作的邏輯也不同，所以在查詢資料上需要特別注意這點。前端的功能除了傳統 Todo App 的新增、刪除、標記完成任務的功能之外，還可以列出這個 DAPP 過去的操作紀錄。但這邊只會強調與 web3 相關的接口部分，其餘的部分請看完整的程式碼，運作流程如下：



## Step1. 安裝 web3

```
1 npm install web3
```

## Step2. 初始化 web3

```
1 // ethereum-todo/src/config/config-web3.js
2
3 // 使用 websocket
4 // localhost:8545 是利用 truffle 建立的測試用區塊鏈
5 // const web3 = new Web3(new Web3.providers.WebsocketProv
6 const web3 = new Web3('ws://localhost:8545');
7
8 // or 使用 http
9 // const web3 = new Web3(new Web3.providers.HttpProvider(
10 // const web3 = new Web3('http://localhost:8545');
```

除了自己架設測試的區塊鏈外，也可以使用公開的測試區塊鏈。

## Step3. 初始化合約

這邊需要用到合約的 ABI (Application Binary Interface) 以及在區塊鏈上佈署的位址，ABI 就是紀錄合約中使用到的函數和變數的文件，這裡有更詳細的說明，首先先將前一個部分編譯後的檔案 (`build/contracts/ToDoFactory.json`) 複製到專案資料夾

```
1 // ethereum-todo/src/contracts/todoContract
2 import web3 from 'config/config-web3';
3 import ToDoFactoryJSON from './ToDoFactory.json';
4
5 const CONTRACT_ADDRESS = '0x21e4624c5a0b3fda81d0833d412dde
6 const todoContract = new web3.eth.Contract(ToDoFactoryJSON
7
8 export default todoContract;
```

這樣就建立一個合約的實體可供操作

## Step4. 執行合約

重複前面提到的，**Transaction** 合約的執行需要消耗 **gas**，所以我們需要有一個帳號來花費 **gas** 執行這些合約，可以從 **truffle** 建立的區塊鏈中找到測試用的帳號（也就是執行時建立的那十個），並藉著 **web3** 提供的 **api** 查看帳號擁有的 **gas**：

```
1 // ethereum-todo/src/helpers/accountsHelper/balance.js
2 import web3 from 'config/config-web3';
3
4 export async function getBalanceAsync(address) {
5   const balance = await web3.eth.getBalance(address);
6   return balance;
7 }
8
9 // 接者可以在任何地方使用 getBalanceAsync 來取得特定使用者目前的
10 // const DEFAULT_USER = '0x1d489c3f8ed5ee71325a847888b215
11 //
12 // void async function() {
13 //   const balance = await getBalanceAsync(DEFAULT_USER);
14 //   console.log('Balance of account0', balance);
15 // }()
```

在 **web3** 中 **Call** 和 **Transaction** 分別對應

`contract.methods.myMethod.call` 和

`contract.methods.myMethod.send` 兩種呼叫方式，後者在之前的版本是 `sendTransaction`。兩者的使用如下

```
1 // ethereum-todo/src/helpers/todoHelpers/todoAction.js
2 import todoContract from 'contracts/todoContract';
3
4 // 第一個測試帳號的 gas 數目相當多，很適合用來測試合約的執行
5 const DEFAULT_USER = '0x1d489c3f8ed5ee71325a847888b2157c9
6
7 export async function getTodoAsync(todoId) {
8   // 估計需要消耗的 gas
9   const gas = await todoContract.methods.getTodo(todoId).
10   console.log('Get Todo: Estimated gas', gas);
11   // 因為 getTodo 不會修改到資料，所以用 call
12   const result = await todoContract.methods.getTodo(todoId
13     from: DEFAULT_USER,
14     gas:200000,
15   });
16   return result;
17 }
18
```

```

19 export async function addTodoAsync(taskName) {
20   // 估計需要消耗的 gas，因為必須寫入字串，所以很可能會消耗超過預計
21   // 故調高 gas limit 到 200000

22   const gas = await todoContract.methods.addTodo(taskName)
23   console.log('AddTodo: Estimated gas', gas);
24   // 因為 addTodo 會修改資料，所以必須用 send
25   // 如果這邊改成 call，依然可以執行，但不會有資料的寫入
26   await todoContract.methods.addTodo(taskName).send({
27     from: DEFAULT_USER,
28     gas: 200000,
29   });
30 }

```

無論用 `call` 或是 `send` 都可以指定消耗的 **gas** 最大值（稱為 **gas limit** 這裡是 200000），**gas limit** 的設計是為了防止智慧合約在執行時產生無窮迴圈的情形，因為所有的運算都需要消耗 **gas**，一旦消耗 **gas** 的總量到達 **gas limit**，就會終止執行。

## Step5. 監聽執行的結果

還是一樣 `Transaction` 的執行需要等到礦工們寫入資料才算真的完成，因此只能利用監聽事件的方式來確定。在這個專案中將這些監聽的處理放在 `src/events` 資料夾下，未來或許放在 `middleware` 是比較漂亮的方式。

```

1  // ethereum-todo/src/events/todoEvents.js
2  import todoContract from 'contracts/todoContract';
3  import * as todoHelper from 'helpers/todoHelpers';
4  import { addTodo, deleteTodo, completeTodo } from 'contract';
5
6  import store from '../store';
7
8  // Event
9  // OnTodoAdded 是 event 的名稱
10 todoContract.events.OnTodoAdded({
11 }, async (error, result) => {
12   if (error) {
13     console.log(error);
14     return;
15   }
16   // result
17   // {
18   //   raw: {

```

```

19    //      data: "0x0000000000000000000000000000000000000000000000000000000000000000
20    //      topics: ["0x6edbfefb4adc3e180444860a21cd838446f0
21    //    },

22    //    returnValues: {
23    //      todoId: 1
24    //    },
25    //    address: "0x21e4624c5A0B3fdA81D0833d412DDED2bb3A7a
26    //    blockHash: "0x9186b52740bff34239c92137ae1ecb7205a0
27    //    blockNumber: 61,
28    //    event: "OnTodoAdded",
29    //    signature: "0x6edbfefb4adc3e180444860a21cd838446f0
30    //    transactionHash: "0x997195a40d9ad102b0c18b730711fe
31    //    transactionIndex: 0,
32    //    type: "mined"
33    //  }
34    const { returnValues: { todoId } } = result;
35
36    // 因為事件的回傳值只有 todoId，因此還需要取得完整的 todo 資料
37    const todo = await todoHelper.getTodoAsync(todoId);
38    store.dispatch(addTodo(todoId, todo[0], todo[1]));
39    console.log('Add', todoId);
40  });

```

## Step6. 獲取過去的事件

區塊鏈可以視為一個保存操作紀錄並且確保這些紀錄無法被竄改的資料庫，因此上述的操作事件都會被紀錄在區塊鏈上，web3 提供 `getPastEvents` 這個 api 來取得過去的事件（web3 在 v1.0 版本前要取得過去的事件需要持續監聽，而非直接傳回結果）

```

1  // ethereum-todo/src/helpers/todoHelpers/eventLogs.js
2  import todoContract from 'contracts/todoContract';
3
4  export async function getAllEventsAsync() {
5    const events = await todoContract.getPastEvents('allEve
6      // 也就是取得從第一個區塊到最新區塊的所有事件
7      fromBlock: 0,
8      toBlock: 'latest'
9    });
10   // events 是一個陣列，陣列元素與前述監聽事件的回傳值相同
11   return events;
12 }

```



透過以上的兩個部分已經可以利用智慧合約在區塊鏈上寫一個 Todo DApp，不過使用的方式仍然相當侷限，也忽略不少實際上可能會碰到的問題，事實上光是如何適當的儲存資料在區塊鏈上就是一大挑戰，這個專案可以當作簡單基底，繼續深入研究。

## 參考資料

- [Ethereum區塊鏈！智能合約\(Smart Contract\)與分散式網頁應用\(DApp\)入門](#)
- [What is different between a transaction and a call](#)
- [比較 require, assert, 和 revert 及其運作的方式](#)
- [How to listen for contract events in javascript tests](#)
- [在公開測試鏈上部署合約。](#)
- [深入智能合約 ABI](#)

關於作者：

@cychien 喜愛閱讀，熱愛動手作，相信未來建立在對歷史的理解上

喜歡我們的文章嗎？歡迎分享按讚給予我們支持和鼓勵！

Like 1   Share



訂閱 TechBridge Weekly 技術週刊，每週發送最精華的技術開發、產品設計的資訊給您

[馬上訂閱技術週刊](#)

PS. 我們討厭垃圾信，所以我們只提供有價值的內容給您 :)

### TechBridge Weekly 技術週刊編輯團隊

TechBridge Weekly 技術週刊團隊是一群對用技術改變世界懷抱熱情的團隊。本技術共筆部落格初期專注於Web前後端、行動網路、機器人/物聯網、資料科學與產品設計等技術分享。This is TechBridge Weekly Team Tech Blog, which focus on web, mobile, robotics, IoT, Data Science technology sharing.

### Share this post

[Star](#)

[關於我們](#) / [技術日報](#) / [技術週刊](#) / [粉絲專頁](#) / [訂閱RSS](#)

Like 7.3K   Share

### 更多優質技術文章

← [用 Javascript 進行邏輯迴歸分析](#)

[淺談 React Fiber 及其對 lifecycles 造成的影響](#) →

## 留言討論

[Comments](#)[Community](#)[1 Login](#)[Recommend](#)[Tweet](#)[Share](#)[Sort by Best](#)

LOG IN WITH


OR SIGN UP WITH DISQUS [?](#)

Be the first to comment.

ALSO ON TECHBRIDGE WEEKLY 技術共筆部落格

**軟體開發的未來，是大斗內時代？ | TechBridge 技術共**

1 comment • a year ago

 **Hu Chun** — 這篇文章思路清晰，看到作者國中生跪了。**一起來了解 Web Authentication |**

1 comment • 3 months ago

 **朱柏翰** — w96jo Avatar2j6xu4x8 !**PWA 實戰經驗分享 | TechBridge 技術共筆部落**

2 comments • a year ago

 **李偉成** —**如何充滿熱情地學習 - 以資料結構為例 | TechBridge 技**

2 comments • a year ago

 **Po-Jen Lai** — 不好意思，All content copyright [TechBridge 技術共筆部落格](#) © 2017 • All rights reserved.