Knockout教學

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **Knockout(以下簡稱ko)的資料繫結(data binding):**  透過ko的applyBindings方法,可以將資料模型(ViewModel)與視圖(View)做繫結,繫結後ko會自動對ViewModel及View中的改變做相依追踪(Dependency Tracking),使其資料在二者間自動同步。  換句話說,我們不需再寫任何額外的程式,即可以使ViewModel中的資料,自動更新到View中(反之亦然)!  **在View中做繫結(單向)**  在ViewModel中我們宣告的了一組資料模型(AppViewModel),其中有姓及名二個成員資料,而ko.applyBindings(new AppViewModel());則會使其資模型開始運作。  在View中,我們會使用data-bind屬性來做設定,使ko知道如何產生相應的行為,在右方例子中,我們宣告data-bind="text: firstName"來表示AppViewModel中的firstName成員,會成為<strong>標籤中text的值。也就是說,ko會轉換成如下的html:  <p>First name: <strong>Bert</strong></p>  <p>Last name: <strong>Bertington</strong></p>  所以我們會得到下方的執行結果。  **注:在本例中的繫結是由ViewModel到View的單向繫結,在ViewModel中的資料做了改變,會直接反應到View。** | <p>First name: <strong data-bind="text: firstName"></strong></p> <p>Last name: <strong data-bind="text: lastName"></strong></p> |
| 執行結果 | Javascript(ViewModel) |
| First name: **Bert**  Last name: **Bertington** | function AppViewModel() {     this.firstName = "Bert";     this.lastName = "Bertington"; }  // Activates knockout.js ko.applyBindings(new AppViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **雙向資料繫結(Observables):**  在View中的資料,若為可以編輯(如右邊例子),且在做過變更後能自動更新AppViewModel中的firstName及lastName,則必須要在ViewModel中使用observable方法來自動追踪並同步。  在這個例子中,若改變FirstName中的值,會自動更新所有firstName的內容(可看到第一行的內容也會跟著改變)。 | <p>First name: <strong data-bind="text: firstName"></strong></p> <p>Last name: <strong data-bind="text: lastName"></strong></p>  <p>First name: <input data-bind="value: firstName" /></p> <p>Last name: <input data-bind="value: lastName" /></p> |
| 執行結果 | Javascript(ViewModel) |
|  | function AppViewModel() {     this.firstName = ko.**observable**("Bert");     this.lastName = ko.**observable**("Bertington"); }  // Activates knockout.js ko.applyBindings(new AppViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **定義計算值資料(Computed value):**  ko.computed()可以用來定義一組計算出來的值(可能由數個ko.observable資組合而成),則該值會自動與其相關的各個值做追踪並同步。  以此例來說,fullName會自動追踪firstName及lastName的改變,而自動更新自己的值。 | <p>First name: <strong data-bind="text: firstName"></strong></p> <p>Last name: <strong data-bind="text: lastName"></strong></p>  <p>First name: <input data-bind="value: firstName" /></p> <p>Last name: <input data-bind="value: lastName" /></p>  <p>Full name: <strong data-bind="text: fullName"></strong></p> |
| 執行結果 | Javascript(ViewModel) |
|  | function AppViewModel() {     this.firstName = ko.observable("Bert");     this.lastName = ko.observable("Bertington");      this.fullName = ko.computed(function() {         return this.firstName() + " " + this.lastName();         }, this); }  // Activates knockout.js ko.applyBindings(new AppViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **行為(behavior):**  Ko亦可以幫我們處理事件,例如按鍵的click事件。  在data-bind中設定 click事件,並指定在ViewModel中的方法名稱,當按下按鍵後,會執行該方法。  在此例中的capitalizeLastName即為button的click事件要執行的方法。  另外,當我們要改變ViewModel 中成員的資料時,要以方法的型式去改變,例如:我們要改變lastName的值,要寫成this.lastName("xxxxx"),這樣才會被ko追踪到並處理同步。 | <p>First name: <strong data-bind="text: firstName"></strong></p> <p>Last name: <strong data-bind="text: lastName"></strong></p>  <p>First name: <input data-bind="value: firstName" /></p> <p>Last name: <input data-bind="value: lastName" /></p>  <p>Full name: <strong data-bind="text: fullName"></strong></p>  <button data-bind="click: capitalizeLastName">Go caps</button> |
| 執行結果 | Javascript(ViewModel) |
|  | function AppViewModel() {     this.firstName = ko.observable("Bert");     this.lastName = ko.observable("Bertington");      this.fullName = ko.computed(function() {         return this.firstName() + " " + this.lastName();         }, this);      this.capitalizeLastName = function() {         var currentVal = this.lastName();        // 讀取lastName裡的值         this.lastName(currentVal.toUpperCase()); // 寫入lastName的值     };     }  // Activates knockout.js ko.applyBindings(new AppViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **處理資料集合(Lists and Collections):**  我們時常會要處理或展示多筆資料的情形,讓使用者可以新增或刪除資料,在ko可透過observableArray()方法,配合foreach的繫結來處理。  foreach會依據指定的ViewModel中的集合成員(本例中為seats)做迴圈處理(重覆它所包含的標籤,本例中為<tr>...</tr>這組標籤),被包含的標籤中所使用的成員,是seats中的每一筆集合的子集合(例如seats中的每一筆,是由SeatReservation所組成,故它的子成員為name及meal。(當中的meal又是由mealName及price所組合,所以meal().mealName可取得餐點名稱,meal().price取得價格)  ko.observableArray()做用與ko.observable()相同,差別在於前者是針對陣列的管理  **注意: 當ViewModel中的成員,是由ko.observable()所建立時(例如meal),得在取得meal值時,請用meal(),而要取得meal內的price時,是用meal().price(不可用meal.price)** | <h2>Your seat reservations</h2> <table>     <thead><tr>         <th>Passenger name</th><th>Meal</th><th>Surcharge</th><th></th>     </tr></thead>     <tbody data-bind="foreach: seats">         <tr>             <td data-bind="text: name"></td>             <td data-bind="text: meal().mealName"></td>             <td data-bind="text: meal().price"></td>         </tr>         </tbody> </table> |
| 執行結果 | Javascript(ViewModel) |
|  | // SeatReservation的資料物件(value object),表示每個座位的內容(有名稱及餐點) function SeatReservation(name, initialMeal) {     var self = this;     self.name = name;     self.meal = ko.observable(initialMeal); } // Overall viewmodel for this screen, along with initial state function ReservationsViewModel() {     var self = this;  // 可供選擇的餐點(有餐點名稱及價格),這些資料應該來自主機,這邊只是示範     self.availableMeals = [         { mealName: "Standard (sandwich)", price: 0 },         { mealName: "Premium (lobster)", price: 34.95 },         { mealName: "Ultimate (whole zebra)", price: 290 }     ];         // seats表所有座位的list,每個座位(SeatReservation)都有名稱及餐點,每份餐點都有餐點名稱及價格(二階層資料結構)     self.seats = ko.observableArray([         new SeatReservation("Steve", self.availableMeals[2]),         new SeatReservation("Bert", self.availableMeals[1])     ]); }  ko.applyBindings(new ReservationsViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **加入資料到集合中(Adding items):**  在ViewModel中定義一個方法(addSeat),由於seats的內容是由observableArray所管理(參考前一例子),此時只要在seats的陣例中加入(或刪除),ko會自動追踪並同步View中有參考到seats的部份(foreach: seats) | <h2>Your seat reservations</h2> <table>     <thead><tr>         <th>Passenger name</th><th>Meal</th><th>Surcharge</th><th></th>     </tr></thead>     <tbody data-bind="foreach: seats">         <tr>             <td data-bind="text: name"></td>             <td data-bind="text: meal().mealName"></td>             <td data-bind="text: meal().price"></td>         </tr>         </tbody> </table>  <button data-bind="click: addSeat">Reserve another seat</button> |
| 執行結果 | Javascript(ViewModel) |
|  | function SeatReservation(name, initialMeal) {     var self = this;     self.name = name;     self.meal = ko.observable(initialMeal); } function ReservationsViewModel() {     var self = this;      self.availableMeals = [         { mealName: "Standard (sandwich)", price: 0 },         { mealName: "Premium (lobster)", price: 34.95 },         { mealName: "Ultimate (whole zebra)", price: 290 }     ];         self.seats = ko.observableArray([         new SeatReservation("Steve", self.availableMeals[2]),         new SeatReservation("Bert", self.availableMeals[1])     ]);      // 新增座位方法     self.addSeat = function() {         self.seats.push(new SeatReservation("", self.availableMeals[0]));     } }  ko.applyBindings(new ReservationsViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **讓集合資料可編輯(Editable items):**  要讓資料可以編輯,主要是在View中的html碼(如本例中的<td>裡的部份,改由<input>及<select>等標籤來處理)  Ko中可以支援View的功能非常多,在本例中示範下拉選單的用法  <select data-bind="options: $root.availableMeals, value: meal, optionsText: 'mealName'">  其中:  options是指定資料來源(其中的$root表是是由根物件,也就是ReservationsViewModel中的成員)  value是指定用什麼成員當做下拉選單的"值"(本例中是meal,為seats中的meal值)  optionsText是指定用options中的那一個成員來顯示在畫面上  另外,在SearReservation()中,增加一個formattedPrice的計算值,主要是用來示簵如何將一個值做格式化的顯示。 | <h2>Your seat reservations</h2> <table>     <thead><tr><th>Passenger name</th><th>Meal</th><th>Surcharge</th><th></th></tr>  </thead>     <tbody data-bind="foreach: seats">         <tr>             <td><input data-bind="value: name" /></td>             <td><select data-bind="options: $root.availableMeals, value: meal, optionsText: 'mealName'"></select></td>             <td data-bind="text: formattedPrice"></td>          </tr>         </tbody> </table>  <button data-bind="click: addSeat">Reserve another seat</button> |
| 執行結果 | Javascript(ViewModel) |
|  | function SeatReservation(name, initialMeal) {     var self = this;     self.name = name;     self.meal = ko.observable(initialMeal);      self.formattedPrice = ko.computed(function() {         var price = self.meal().price;         return price ? "$" + price.toFixed(2) : "None";             });   } function ReservationsViewModel() {     var self = this;      self.availableMeals = [         { mealName: "Standard (sandwich)", price: 0 },         { mealName: "Premium (lobster)", price: 34.95 },         { mealName: "Ultimate (whole zebra)", price: 290 }     ];         self.seats = ko.observableArray([         new SeatReservation("Steve", self.availableMeals[2]),         new SeatReservation("Bert", self.availableMeals[1])     ]);      // 新增座位方法     self.addSeat = function() {         self.seats.push(new SeatReservation("", self.availableMeals[0]));     } }  ko.applyBindings(new ReservationsViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **刪除資料及顯示加總:**  在這範例中主要是示範如何做資料刪除,在<tr>中加入一個<a>(也可用button之類的)  <a href="#" data-bind="click: $root.removeSeat">Remove</a>  並定義click事件去呼叫ReservationsViewModel中的removeSeat(seat),ko會將當前那筆資料當參數傳入。  另外,在顯示總計中,用了一個visible的設定,表示當指定條件(totalSurcharge() > 0)成立時,才會顯示出來。 | <h2>Your seat reservations</h2> <table>     <thead><tr><th>Passenger name</th><th>Meal</th><th>Surcharge</th><th></th></tr></thead>     <tbody data-bind="foreach: seats">         <tr>             <td><input data-bind="value: name" /></td>             <td><select data-bind="options: $root.availableMeals, value: meal, optionsText: 'mealName'"></select></td>             <td data-bind="text: formattedPrice"></td>             <td><a href="#" data-bind="click: $root.removeSeat">Remove</a></td>         </tr>         </tbody> </table> <button data-bind="click: addSeat">Reserve another seat</button> <h3 data-bind="visible: totalSurcharge() > 0">     Total surcharge: $<span data-bind="text: totalSurcharge().toFixed(2)"></span></h3> |
| 執行結果 | Javascript(ViewModel) |
|  | function SeatReservation(name, initialMeal) {     var self = this;     self.name = name;     self.meal = ko.observable(initialMeal);     self.formattedPrice = ko.computed(function() {         var price = self.meal().price;         return price ? "$" + price.toFixed(2) : "None";             });     } function ReservationsViewModel() {     var self = this;     self.availableMeals = [         { mealName: "Standard (sandwich)", price: 0 },         { mealName: "Premium (lobster)", price: 34.95 },         { mealName: "Ultimate (whole zebra)", price: 290 }     ];      self.seats = ko.observableArray([         new SeatReservation("Steve", self.availableMeals[0]),         new SeatReservation("Bert", self.availableMeals[0])     ]);     // 計算加總值      self.totalSurcharge = ko.computed(function() {        var total = 0;        for (var i = 0; i < self.seats().length; i++)            total += self.seats()[i].meal().price;        return total;     });         // 新增及刪除方法     self.addSeat = function() {         self.seats.push(new SeatReservation("", self.availableMeals[0]));     }     self.removeSeat = function(seat) { self.seats.remove(seat) } }  ko.applyBindings(new ReservationsViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **最多只能加入5個座位:**  在這範例中主要是示範如何做更細部的控制,例如:我只讓使用者最多加入5個座位  <button data-bind="click: addSeat, enable: seats().length < 5">Reserve another seat</button>  只要在加座位按鍵上,使用enable的設定,並指定條件(seats().length < 5),當條件成立是方為可點選的狀態。 | <h2>Your seat reservations (<span data-bind="text: seats().length"></span>)</h2> <table>     <thead><tr>         <th>Passenger name</th><th>Meal</th><th>Surcharge</th><th></th>     </tr></thead>     <tbody data-bind="foreach: seats">         <tr>             <td><input data-bind="value: name" /></td>             <td><select data-bind="options: $root.availableMeals, value: meal, optionsText: 'mealName'"></select></td>             <td data-bind="text: formattedPrice"></td>             <td><a href="#" data-bind="click: $root.removeSeat">Remove</a></td>         </tr>         </tbody> </table> <button data-bind="click: addSeat, enable: seats().length < 5">Reserve another seat</button> <h3 data-bind="visible: totalSurcharge() > 0">     Total surcharge: $<span data-bind="text: totalSurcharge().toFixed(2)"></span> </h3> |
| 執行結果 | Javascript(ViewModel) |
|  | function SeatReservation(name, initialMeal) {     var self = this;     self.name = name;     self.meal = ko.observable(initialMeal);     self.formattedPrice = ko.computed(function() {         var price = self.meal().price;         return price ? "$" + price.toFixed(2) : "None";             });     } function ReservationsViewModel() {     var self = this;     self.availableMeals = [         { mealName: "Standard (sandwich)", price: 0 },         { mealName: "Premium (lobster)", price: 34.95 },         { mealName: "Ultimate (whole zebra)", price: 290 }     ];      self.seats = ko.observableArray([         new SeatReservation("Steve", self.availableMeals[0]),         new SeatReservation("Bert", self.availableMeals[0])     ]);     // 計算加總值      self.totalSurcharge = ko.computed(function() {        var total = 0;        for (var i = 0; i < self.seats().length; i++)            total += self.seats()[i].meal().price;        return total;     });         // 新增及刪除方法     self.addSeat = function() {         self.seats.push(new SeatReservation("", self.availableMeals[0]));     }     self.removeSeat = function(seat) { self.seats.remove(seat) } }  ko.applyBindings(new ReservationsViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **設計一個Single Page Application (SPA):**  有關於SPA的說明,請見下方補充說明部份,或拜請google大神  **建立一個webmail client**  在本例中將會建立一個客戶端郵件顯示的功能,先利用<ul><li>標籤,建立出郵件分類的選項  其中:  $data是指folders陣列中的元素  Css 是用來設定樣式,在這例子中,當$data == $root.chosenFolderId()成立時,則會被加入"selected"的class值  並在ViewModel中,建一個WebmailViewModel的資料模型,成員包括  folders: 信件分類  chosenFolderId: 被選中的分類id  gotoFolder(): 當分類被點選時所觸發的事件,用來指定chosenFolderId的值 | <!-- 分類 --> <ul class="folders" data-bind="foreach: folders">     <li data-bind="text: $data,                     css: { selected: $data == $root.chosenFolderId() },                    click: $root.goToFolder"></li> </ul> |
| CSS |
| body { font-family: Helvetica, Arial}  .folders { background-color: #bbb; list-style-type: none; padding: 0; margin: 0; border-radius: 7px;  background-image: -webkit-gradient(linear, left top, left bottom, color-stop(0, #d6d6d6), color-stop(0.4, #c0c0c0), color-stop(1,#a4a4a4));  margin: 10px 0 16px 0;  font-size: 0px;  }  .folders li:hover { background-color: #ddd; }  .folders li:first-child { border-left: none; border-radius: 7px 0 0 7px; }  .folders li { font-size: 16px; font-weight: bold; display: inline-block; padding: 0.5em 1.5em; cursor: pointer; color: #444; text-shadow: #f7f7f7 0 1px 1px; border-left: 1px solid #ddd; border-right: 1px solid #888; }  .folders li { \*display: inline !important; } /\* IE7 only \*/  .folders .selected { background-color: #444 !important; color: white; text-shadow:none; border-right-color: #aaa; border-left: none; box-shadow:inset 1px 2px 6px #070707; }  .mails { width: 100%; table-layout:fixed; border-spacing: 0; }  .mails thead { background-color: #bbb; font-weight: bold; color: #444; text-shadow: #f7f7f7 0 1px 1px; }  .mails tbody tr:hover { cursor: pointer; background-color: #68c !important; color: White; }  .mails th, .mails td { text-align:left; padding: 0.4em 0.3em; white-space: nowrap; overflow: hidden; text-overflow: ellipsis; }  .mails th { border-left: 1px solid #ddd; border-right: 1px solid #888; padding: 0.4em 0 0.3em 0.7em; }  .mails th:nth-child(1), .mails td:nth-child(1) { width: 20%; }  .mails th:nth-child(2), .mails td:nth-child(2) { width: 15%; }  .mails th:nth-child(3), .mails td:nth-child(3) { width: 45%; }  .mails th:nth-child(4), .mails td:nth-child(4) { width: 15%; }  .mails th:last-child { border-right: none }  .mails tr:nth-child(even) { background-color: #EEE; }    .viewMail .mailInfo { background-color: #dae0e8; padding: 1em 1em 0.5em 1.25em; border-radius: 1em; }  .viewMail .mailInfo h1 { margin-top: 0.2em; font-size: 130%; }  .viewMail .mailInfo label { color: #777; font-weight: bold; min-width: 2.75em; text-align:right; display: inline-block; }  .viewMail .message { padding: 0 1.25em; } |
| 執行結果 | Javascript(ViewModel) |
|  | function WebmailViewModel() {     // 資料成員     var self = this;     self.folders = ['Inbox', 'Archive', 'Sent', 'Spam'];     self.chosenFolderId = ko.observable();      // Behaviours         self.goToFolder = function(folder) { self.chosenFolderId(folder); };     };  ko.applyBindings(new WebmailViewModel()); |
| 補充說明 |  |
| **Single page applications**  Many of the most modern, responsive, and engaging web-based UIs have gone beyond traditional Ajax and have become *single page applications*: the visitor can seemingly navigate within a single page at the speed of a native application. The best-known example is probably GMail, but these days it's an increasingly widespread technique.  Such applications use **hash-based** or **pushState navigation** to support back/forward gestures and bookmarking. If you're not familiar with how that technique works, [see this short explanation](http://learn.knockoutjs.com/).  **How hash-based/pushState navigation works**  For **hash-based** navigation, the visitor's position in a virtual navigation space is stored in the *URL hash*, which is the part of the URL after a 'hash' symbol (e.g.,/my/app/#category=shoes&page=4). Whenever the URL hash changes, the browser *doesn't*issue an HTTP request to fetch a new page; instead it merely adds the new URL to its back/forward history list and exposes the updated URL hash to scripts running in the page. The script notices the new URL hash and dynamically updates the UI to display the corresponding item (e.g., page 4 of the "shoes" category).  This makes it possible to support back/forward button navigation in a single page application (e.g., pressing 'back' moves to the previous URL hash), and effectively makes virtual locations bookmarkable and shareable.  **pushState** is an HTML5 API that offers a different way to change the current URL, and thereby insert new back/forward history entries, without triggering a page load. This differs from hash-based navigation in that you're not limited to updating the hash fragment — you can update the entire URL. | **單頁應用程式**  許多最現代、靈敏和基於Web的UI已超越傳統 Ajax，已成為單個頁面的應用程式(single page applications)：訪客可以在單一的頁面巡覽像本地應用程序的速度。最著名的例子可能是GMail，但這些日子裡，它已經是一種越來越普遍的技術。  **HASH-BASED與PUSHSTATE巡覽**  這種應用程式使用hash-based或pushState巡覽以支援前進/後退手勢和書籤。如果讀者不熟悉這種技術的工作原理，請參閱以下簡短的解釋。  hash-based巡覽：訪問者的位置是儲存在URL hash的虛擬巡覽空間裡，hash符號的後面是URL的一部分（例如，/my/app/#category=shoes&page=4），每當URL hash有變動，瀏覽器不會發出HTTP請求來獲取新的一頁。相反的是，它只會把新的URL加入它的前進或後退歷史清單中，並在此網頁運作的腳本中公開並更新新URL hash。該腳本通知新URL hash和動態更新UI並顯示對應的項目內容（例如，”#category=shoes&page=4”代表鞋子目錄第四頁）。  這樣就可以在單個頁面應用程式中支援前進與後退按鈕巡覽（例如，按下”後退鍵”移至前一個URL hash），和有效地使用虛擬位置並可和書籤共同。  pushState是一個提供了另一種不同的方法來更改當前URL的HTML5 API，它將插入新的前進和後退和歷史項目，而不會觸發頁面載入。這不同於URL hash巡覽，因為不限於更新hash片段，可以更新整個URL。 |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **加入郵件資料顯示(Grid of mail):**  在ViewModel中加入chosenFolderData,用來存放郵件的內容,當點選分類時,會透過:  $.get('/mail', { folder: folder }, self.chosenFolderData);  到主機抓取資料放到self.chosenFolderData中,取得的資料格式如下:    在View中的<table>用到了with: chosenFolderData,表示其內部被包的標籤中的成員,均參考chosenFolderData物件,因此foreach: mails即指的是由主機取得的資料(被放入了chosenFolderData中)  而mails展開後的成員(如上圖),則有from, to, subject, date...等 | <!-- 分類 --> <ul class="folders" data-bind="foreach: folders">     <li data-bind="text: $data,                     css: { selected: $data == $root.chosenFolderId() },                    click: $root.goToFolder"></li> </ul> <!-- 郵件列表 --> <table class="mails" data-bind="with: chosenFolderData">     <thead><tr><th>From</th><th>To</th><th>Subject</th><th>Date</th></tr></thead>     <tbody data-bind="foreach: mails">         <tr>             <td data-bind="text: from"></td>             <td data-bind="text: to"></td>             <td data-bind="text: subject"></td>             <td data-bind="text: date"></td>         </tr>          </tbody> </table> |
| 執行結果 | Javascript(ViewModel) |
|  | function WebmailViewModel() {     // 資料成員     var self = this;     self.folders = ['Inbox', 'Archive', 'Sent', 'Spam'];     self.chosenFolderId = ko.observable();     self.chosenFolderData = ko.observable();      // 行為(方法)     self.goToFolder = function(folder) {          self.chosenFolderId(folder);         $.get('/mail', { folder: folder }, self.chosenFolderData);     };      // 預設為顯示Inbox     self.goToFolder('Inbox'); };  ko.applyBindings(new WebmailViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **檢視郵件內容 (Viewing individual emails):**  做法類示上一個例子,增加一個方法      self.goToMail = function(mail) {          self.chosenFolderId(mail.folder);         self.chosenFolderData(null); // Stop showing a folder         $.get("/mail", { mailId: mail.id }, self.chosenMailData);     };  當郵件被點選後,會將chosenFolderData清空,再載入郵件內容做顯示 | <!-- 分類 --> <ul class="folders" data-bind="foreach: folders">     <li data-bind="text: $data,                     css: { selected: $data == $root.chosenFolderId() },                    click: $root.goToFolder"></li> </ul> <!-- 郵件列表 --> <table class="mails" data-bind="with: chosenFolderData">     <thead><tr><th>From</th><th>To</th><th>Subject</th><th>Date</th></tr></thead>     <tbody data-bind="foreach: mails">         <tr data-bind="click: $root.goToMail">             <td data-bind="text: from"></td>             <td data-bind="text: to"></td>             <td data-bind="text: subject"></td>             <td data-bind="text: date"></td>         </tr>          </tbody> </table> <!-- 被選中的郵件內容 --> <div class="viewMail" data-bind="with: chosenMailData">     <div class="mailInfo">         <h1 data-bind="text: subject"></h1>         <p><label>From</label>: <span data-bind="text: from"></span></p>         <p><label>To</label>: <span data-bind="text: to"></span></p>         <p><label>Date</label>: <span data-bind="text: date"></span></p>     </div>     <p class="message" data-bind="html: messageContent" /> </div> |
| 執行結果 | Javascript(ViewModel) |
|  | function WebmailViewModel() {     // 資料成員     var self = this;     self.folders = ['Inbox', 'Archive', 'Sent', 'Spam'];     self.chosenFolderId = ko.observable();     self.chosenFolderData = ko.observable();     self.chosenMailData = ko.observable();     // 行為(方法)     self.goToFolder = function(folder) {          self.chosenFolderId(folder);         self.chosenMailData(null); // Stop showing a mail         $.get('/mail', { folder: folder }, self.chosenFolderData);     };     self.goToMail = function(mail) {          self.chosenFolderId(mail.folder);         self.chosenFolderData(null); // Stop showing a folder         $.get("/mail", { mailId: mail.id }, self.chosenMailData);     };     // 預設為顯示Inbox     self.goToFolder('Inbox'); }; ko.applyBindings(new WebmailViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **使用客戶端導航(client-side navigation):**  在點選後重導到不同頁面的過程中,在上個例中是以改變資料內容來達成(改變chosenFolderData及chosenMailData),如此做會產生一個問題,就是當使用者按下瀏覽器"上一頁"時,會出現沒有上一頁的窘境而產生錯誤!  在本例中,我們運用另一套框架Sammy.js來輔助我們做客戶端導航中的路由設定(依照不同路徑URL來觸發不同的事件)  self.goToFolder = function(folder) { location.hash = folder }; self.goToMail = function(mail) { location.hash = mail.folder + '/' + mail.id };  首先,將這二行改成,當點選分類或郵件時,不做改變資料的動作,而是改變location.hash的內容(關於location.hash的說明教學,可以參考這篇: <http://ngarmchannel.blogspot.tw/2012/02/locationhash.html>)  利用Sammy定義出三組路由格式  // #Inbox這樣的各式,會觸發這段方法  this.get('#:folder', function() {  ...        });  // #Inbox/3這樣的各式,會觸發這段方法 this.get('#:folder/:mailId', function() {  ...        });  // 初始時,會跑這段,使用runRoute表示空的用戶端URL會被視同為#Inbox，即，它將載入並呈現Inbox。 this.get('', function() { this.app.runRoute('get', '#Inbox') }); 一旦location.hash改變,Sammy就會接手觸發相對應的方法,並將相關資料載入。這時,就算使用者按上一頁,也會正確的載入相對應的資料! | <script src="/scripts/lib/sammy.js" type="text/javascript"></script>  <!-- 分類 --> <ul class="folders" data-bind="foreach: folders">     <li data-bind="text: $data,                     css: { selected: $data == $root.chosenFolderId() },                    click: $root.goToFolder"></li> </ul> <!-- 郵件列表 --> <table class="mails" data-bind="with: chosenFolderData">     <thead><tr><th>From</th><th>To</th><th>Subject</th><th>Date</th></tr></thead>     <tbody data-bind="foreach: mails">         <tr data-bind="click: $root.goToMail">             <td data-bind="text: from"></td>             <td data-bind="text: to"></td>             <td data-bind="text: subject"></td>             <td data-bind="text: date"></td>         </tr>          </tbody> </table> <!-- 被選中的郵件內容 --> <div class="viewMail" data-bind="with: chosenMailData">     <div class="mailInfo">         <h1 data-bind="text: subject"></h1>         <p><label>From</label>: <span data-bind="text: from"></span></p>         <p><label>To</label>: <span data-bind="text: to"></span></p>         <p><label>Date</label>: <span data-bind="text: date"></span></p>     </div>     <p class="message" data-bind="html: messageContent" /> </div> |
| 執行結果 | Javascript(ViewModel) |
|  | function WebmailViewModel() {     // 資料成員     var self = this;     self.folders = ['Inbox', 'Archive', 'Sent', 'Spam'];     self.chosenFolderId = ko.observable();     self.chosenFolderData = ko.observable();     self.chosenMailData = ko.observable();     // Behaviours         self.goToFolder = function(folder) { location.hash = folder };     self.goToMail = function(mail) { location.hash = mail.folder + '/' + mail.id };     // Client-side routes         Sammy(function() {         this.get('#:folder', function() {             self.chosenFolderId(this.params.folder);             self.chosenMailData(null);             $.get("/mail", { folder: this.params.folder }, self.chosenFolderData);         });         this.get('#:folder/:mailId', function() {             self.chosenFolderId(this.params.folder);             self.chosenFolderData(null);             $.get("/mail", { mailId: this.params.mailId }, self.chosenMailData);         });         this.get('', function() { this.app.runRoute('get', '#Inbox') });     }).run();     };  ko.applyBindings(new WebmailViewModel()); |
| 補充說明 |  |
| 關於Sammy.js |  |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **自訂繫結(Custom bindings):**  Ko有一套非常完整的內建繫結(text, click, foreach...等),但不代表僅止於此,你也可以使用少量的程式碼來建立自訂繫結。在任何實際的程式中你會發現用來封裝常見的UI模式的繫結是非常好用的，這樣就能重覆的使用這些模型。  在本例中,我們將改善以下三件事   1. 使用動畫效果提示「You've used too many points! Please remove some.」 2. 改善「Finished」按鈕的樣式 3. 使用星星評分來換代下拉式選單給點數 | <h3 data-bind="text: question"></h3> <p>Please distribute <b data-bind="text: pointsBudget"></b> points between the following options.</p>  <table>     <thead><tr><th>Option</th><th>Importance</th></tr></thead>     <tbody data-bind="foreach: answers">         <tr>             <td data-bind="text: answerText"></td>             <td><select data-bind="options: [1,2,3,4,5], value: points"></select></td>         </tr>         </tbody> </table>  <h3 data-bind="visible: pointsUsed() > pointsBudget">You've used too many points! Please remove some.</h3> <p>You've got <b data-bind="text: pointsBudget - pointsUsed()"></b> points left to use.</p> <button data-bind="enable: pointsUsed() <= pointsBudget, click: save">Finished</button> |
| 執行結果 | Javascript(ViewModel) |
|  | function Answer(text) { this.answerText = text; this.points = ko.observable(1); }  function SurveyViewModel(question, pointsBudget, answers) {     this.question = question;     this.pointsBudget = pointsBudget;     this.answers = $.map(answers, function(text) { return new Answer(text) });     this.save = function() { alert('To do') };                             this.pointsUsed = ko.computed(function() {         var total = 0;         for (var i = 0; i < this.answers.length; i++)             total += this.answers[i].points();         return total;             }, this); }  ko.applyBindings(new SurveyViewModel("Which factors affect your technology choices?", 10, [    "Functionality, compatibility, pricing - all that boring stuff",    "How often it is mentioned on Hacker News",        "Number of gradients/dropshadows on project homepage",            "Totally believable testimonials on project homepage" ])); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **使用動畫效果提示:**  當訪客分配太多點數時，「You've used too many points! Please remove some.」提示訊息會出現，因為它是使用內建visible繫結。如果您想要使其淡入和淡出，可以編寫可重覆使用的自訂繫結，內部使用jQuery的淡入淡出函數來執行動畫。  你可以定義自訂繫結分配一個新屬性給ko.bindingHandlers物件。  **屬性會有兩個回呼函數：**  init：當繫結第一次發生時呼叫。通常用來設定狀態或註冊事件處理常式等。  update：當相關聯的資料有更新時呼叫。能用來更新符合的DOM元素。  ko.bindingHandlers.fadeVisible = {     init: function(element, valueAccessor) {        ...      },     update: function(element, valueAccessor) { ...      }  };  傳入的  Element:表示所繫結的的標籤元素  valueAccessor:表示繫結內容值(本例為pointsUsed() > pointsBudget )  在View中使用我們剛才自訂的繫結fadeVisible  <h3 data-bind="fadeVisible: pointsUsed() > pointsBudget">You've used too many points! Please remove some.</h3> | <p>Please distribute <b data-bind="text: pointsBudget"></b> points between the following options.</p>  <table>     <thead><tr><th>Option</th><th>Importance</th></tr></thead>     <tbody data-bind="foreach: answers">         <tr>             <td data-bind="text: answerText"></td>             <td><select data-bind="options: [1,2,3,4,5], value: points"></select></td>         </tr>         </tbody> </table>  <h3 data-bind="fadeVisible: pointsUsed() > pointsBudget">You've used too many points! Please remove some.</h3> <p>You've got <b data-bind="text: pointsBudget - pointsUsed()"></b> points left to use.</p> <button data-bind="enable: pointsUsed() <= pointsBudget, click: save">Finished</button> |
| 執行結果 | Javascript(ViewModel) |
|  | // ---------------------------------------------------------------------------- // Reusable bindings - ideally kept in a separate file  ko.bindingHandlers.fadeVisible = {     init: function(element, valueAccessor) {         // Start visible/invisible according to initial value         var shouldDisplay = valueAccessor();         $(element).toggle(shouldDisplay);     },     update: function(element, valueAccessor) {         // On update, fade in/out         var shouldDisplay = valueAccessor();         shouldDisplay ? $(element).fadeIn() : $(element).fadeOut();     }  };  // ---------------------------------------------------------------------------- // Page viewmodel  function Answer(text) { this.answerText = text; this.points = ko.observable(1); }  function SurveyViewModel(question, pointsBudget, answers) {     this.question = question;     this.pointsBudget = pointsBudget;     this.answers = $.map(answers, function(text) { return new Answer(text) });     this.save = function() { alert('To do') };                         this.pointsUsed = ko.computed(function() {         var total = 0;         for (var i = 0; i < this.answers.length; i++)             total += this.answers[i].points();         return total;             }, this); }  ko.applyBindings(new SurveyViewModel("Which factors affect your technology choices?", 10, [    "Functionality, compatibility, pricing - all that boring stuff",    "How often it is mentioned on Hacker News",        "Number of gradients/dropshadows on project homepage",            "Totally believable testimonials on project homepage" ])); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **整合第三方元件:**  如果想要在View裡使用其他JavaScript函式庫（例如，jQuery UI或YUI）且要繫結它們到ViewModel屬性，最簡單的方法就是建立自訂繫結。自訂繫結會介於ViewModel與第三方元件的中間。  這裡我們使用jQuery UI button widget（http://jqueryui.com/demos/button/）來改善【完成】按鈕。  在ViewModel最上方定義一個jqButton自訂繫結：  ko.bindingHandlers.jqButton = {     init: function(element) {        $(element).button(); // 將元素轉成jQuery UI按鍵樣式     },     update: function(element, valueAccessor) {  var currentValue = valueAccessor();         // 更新enable的值,若為false,則呼叫jQuery的button,使其設為disabled         $(element).button("option", "disabled", currentValue.enable === false);      }     };  在View中使用jqButton的自訂繫結  <button data-bind="jqButton: { enable: pointsUsed() <= pointsBudget }, click: save">Finished</button> | <h3 data-bind="text: question"></h3> <p>Please distribute <b data-bind="text: pointsBudget"></b> points between the following options.</p>  <table>     <thead><tr><th>Option</th><th>Importance</th></tr></thead>     <tbody data-bind="foreach: answers">         <tr>             <td data-bind="text: answerText"></td>             <td><select data-bind="options: [1,2,3,4,5], value: points"></select></td>         </tr>         </tbody> </table>  <h3 data-bind="fadeVisible: pointsUsed() > pointsBudget">You've used too many points! Please remove some.</h3> <p>You've got <b data-bind="text: pointsBudget - pointsUsed()"></b> points left to use.</p> <button data-bind="jqButton: { enable: pointsUsed() <= pointsBudget }, click: save">Finished</button> |
| 執行結果 | Javascript(ViewModel) |
|  | // ---------------------------------------------------------------------------- // Reusable bindings - ideally kept in a separate file ko.bindingHandlers.fadeVisible = {     init: function(element, valueAccessor) {         // Start visible/invisible according to initial value         var shouldDisplay = valueAccessor();         $(element).toggle(shouldDisplay);     },     update: function(element, valueAccessor) {         // On update, fade in/out         var shouldDisplay = valueAccessor();         shouldDisplay ? $(element).fadeIn() : $(element).fadeOut();     }  };  ko.bindingHandlers.jqButton = {     init: function(element) {        $(element).button(); // Turns the element into a jQuery UI button     },     update: function(element, valueAccessor) {         var currentValue = valueAccessor();         // Here we just update the "disabled" state, but you could update other properties too         $(element).button("option", "disabled", currentValue.enable === false);     }     };  // ---------------------------------------------------------------------------- // Page viewmodel  function Answer(text) { this.answerText = text; this.points = ko.observable(1); }  function SurveyViewModel(question, pointsBudget, answers) {     this.question = question;     this.pointsBudget = pointsBudget;     this.answers = $.map(answers, function(text) { return new Answer(text) });     this.save = function() { alert('To do') };                     this.pointsUsed = ko.computed(function() {         var total = 0;         for (var i = 0; i < this.answers.length; i++)             total += this.answers[i].points();         return total;             }, this); }  ko.applyBindings(new SurveyViewModel("Which factors affect your technology choices?", 10, [    "Functionality, compatibility, pricing - all that boring stuff",    "How often it is mentioned on Hacker News",        "Number of gradients/dropshadows on project homepage",            "Totally believable testimonials on project homepage" ])); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **開發自訂小工具(Widgets):**  最後來改善”重要性”的下拉式選單，改使用星星評分方式。你可以使用現有的外掛插件（例如，http://www.fyneworks.com/jquery/star-rating/）來替代，不過為了學習，讓我們從零開始。  在ViewModel上方自定義一個starRating繫結, 這段程式碼會安插數個<span>元素。(參考CSS部份)  ko.bindingHandlers.starRating = {     init: function(element, valueAccessor) {         $(element).addClass("starRating");         for (var i = 0; i < 5; i++)            $("<span>").appendTo(element); };  **呈現評分**  每當ViewModel變動時，星星評分要能自動更新狀態，我們可以使用update處理常式來處理合適的CSS類別給現有資料：  ko.bindingHandlers.starRating = {     init: function(element, valueAccessor) {         ...       },     update: function(element, valueAccessor) {         // Give the first x stars the "chosen" class, where x <= rating         var observable = valueAccessor();         $("span", element).each(function(index) {             $(this).toggleClass("chosen", index < observable());         });     }     };  **滑行時高亮度提示並將資料回傳VIEWMODEL**  當訪客滑過星星時應該高亮度來突顯它們，這可以幫助他們選擇。這裡使用jQuery的hover函數來轉換星星的狀態  當訪客點擊星星，將會儲存選擇的評分到ViewModel，然後自動更新UI。這裡使用jQuery的click函數來截取點擊：  ko.bindingHandlers.starRating = {     init: function(element, valueAccessor) {         $(element).addClass("starRating");         for (var i = 0; i < 5; i++)            $("<span>").appendTo(element);                 // 處理滑鼠事件Handle mouse events on the stars         $("span", element).each(function(index) {             $(this).hover(                 function() { $(this).prevAll().add(this).addClass("hoverChosen") },                  function() { $(this).prevAll().add(this).removeClass("hoverChosen") }                             ).click(function() {                  var observable = valueAccessor();  // Get the associated observable                 observable(index+1);               // Write the new rating to it             });         });                 },     update: function(element, valueAccessor) {         ...      }     }; | <h3 data-bind="text: question"></h3> <p>Please distribute <b data-bind="text: pointsBudget"></b> points between the following options.</p>  <table>     <thead><tr><th>Option</th><th>Importance</th></tr></thead>     <tbody data-bind="foreach: answers">         <tr>             <td data-bind="text: answerText"></td>             <td data-bind="starRating: points"></td>         </tr>         </tbody> </table>  <h3 data-bind="fadeVisible: pointsUsed() > pointsBudget">You've used too many points! Please remove some.</h3> <p>You've got <b data-bind="text: pointsBudget - pointsUsed()"></b> points left to use.</p> <button data-bind="jqButton: { enable: pointsUsed() <= pointsBudget }, click: save">Finished</button> |
| CSS |
| table { background-color: #cde; padding: 1em; border-radius: 0.5em; }  table th { text-align:left; }  table th:last-child { min-width: 130px; }  .starRating span { width:24px; height:24px; background-image: url(http://learn.knockoutjs.com/Content/TutorialSpecific/stars.png); display:inline-block; cursor: pointer; background-position: -24px 0; }  .starRating span.chosen { background-position: 0 0; }  .starRating:hover span { background-position: -24px 0; }  .starRating:hover span.hoverChosen { background-position: 0 0; } |
| 執行結果 | Javascript(ViewModel) |
|  | // ---------------------------------------------------------------------------- // Reusable bindings - ideally kept in a separate file  ko.bindingHandlers.fadeVisible = {     init: function(element, valueAccessor) {         // Start visible/invisible according to initial value         var shouldDisplay = valueAccessor();         $(element).toggle(shouldDisplay);     },     update: function(element, valueAccessor) {         // On update, fade in/out         var shouldDisplay = valueAccessor();         shouldDisplay ? $(element).fadeIn() : $(element).fadeOut();     }  };  ko.bindingHandlers.jqButton = {     init: function(element) {        $(element).button(); // Turns the element into a jQuery UI button     },     update: function(element, valueAccessor) {         var currentValue = valueAccessor();         // Here we just update the "disabled" state, but you could update other properties too         $(element).button("option", "disabled", currentValue.enable === false);     }     }; |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **由主機載入與儲回資料:**  現在，你已經瞭解MVVM模式如何為動態UI有組織地整理用戶端程式碼，以及knockout.js的可觀察物件（observables）、繫結（bindings）、相依追蹤（dependency tracking）是如何運作。  幾乎所有網站應用程式，都需要由主機取得資料和傳送修改的資料回去主機。因為Knockout.js 是一個純用戶端函式庫，它能非常有彈性運作在任何主機端技術（例如，ASP.NET、ASP.NET MVC、Rails、PHP…等），以及任何架構模式、資料庫…等。只要主機端程式碼可以傳送與接收JSON資料，就能呈現以下教學。  **工作清單範例**  接下來將展示一個工作清單的例子,會由主機載入資料,做新增或刪除工作項目,並介紹二種不同的方式來將資料存回主機。 | <h3>Tasks</h3> <form data-bind="submit: addTask">     Add task: <input data-bind="value: newTaskText" placeholder="What needs to be done?" />     <button type="submit">Add</button> </form> <ul data-bind="foreach: tasks, visible: tasks().length > 0">     <li>         <input type="checkbox" data-bind="checked: isDone" />         <input data-bind="value: title, disable: isDone" />         <a href="#" data-bind="click: $parent.removeTask">Delete</a>     </li>  </ul> |
| CSS |
| ul { list-style-type: none; margin: 1em 0; background-color: #cde; padding: 1em; border-radius: 0.5em; }  ul li a { color: Gray; font-size: 90%; text-decoration: none }  ul li a:hover { text-decoration: underline }  input:not([type]), input[type=text] { width: 30em; }  input[disabled] { text-decoration: line-through; border-color: Silver; background-color: Silver; }  textarea { width: 30em; height: 6em; }  form { margin-top: 1em; margin-bottom: 1em; } |
| 執行結果 | Javascript(ViewModel) |
|  | function Task(data) {     this.title = ko.observable(data.title);     this.isDone = ko.observable(data.isDone); } function TaskListViewModel() {     // 資料成員     var self = this;     self.tasks = ko.observableArray([]);     self.newTaskText = ko.observable();     self.incompleteTasks = ko.computed(function() {         return ko.utils.arrayFilter(self.tasks(), function(task) { return !task.isDone() });     });     // 操作方法     self.addTask = function() {         self.tasks.push(new Task({ title: this.newTaskText() }));         self.newTaskText("");     };     self.removeTask = function(task) { self.tasks.remove(task) }; } ko.applyBindings(new TaskListViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **由主機載入資料:**  要由主機取得JSON資料最簡單的方法是透過一個Ajax請求，在此教學，我們會使用jQuery的$.getJSON和$.ajax函數來做此事。一旦你取得資料，可以使用它來更新ViewModel，並讓UI自動更新。  在對主機透過URL /tasks來請求，然後會回應JSON資料。新增程式碼到TaskListViewModel尾端並請求取得資料，並使用它來加入至tasks陣列：(jQueryj的語法及用法,請參考官方文件說明)      $.getJSON("/tasks", function(allData) {         var mappedTasks = $.map(allData, function(item) { return new Task(item) });         self.tasks(mappedTasks);     });  特別注意：你不是在載入資料之後才呼叫ko.applyBindings。很多Knockout.js新人每次載入一些資料就嘗試重新繫結它們的UI，但這不是很有用。這是沒有理由的重新繫結，只需更新現有的ViewModel就足以讓整個UI更新。  這段程式碼由主機取得原始資料陣列（JSON格式），然後利用jQuery的$.map的功能從每一個物件(或陣列元素)中來產生出Task實體。Task物件的結果陣列推入它到tasks陣列，這將使UI自動更新，因為它是個observable物件。  注意:這裡的連線,是連到官方網站的範例,取回的資料內容及格式如下:    **資料轉換**  前面程式碼展示如何讀取和映射資料，為我們提供了最大且直接控制的手動方法。如果你想要多一點自動化，可以參考knockout.mapping（http://knockoutjs.com/documentation/plugins-mapping.html）外掛，它能夠：   1. 自動建構observable物件資料的物件關係 2. 慣例式的映射，或使用明顯式組態的映射回呼函式 3. 稍後從主機取得更新後的資料，將會改變現有物件關係 | <h3>Tasks</h3> <form data-bind="submit: addTask">     Add task: <input data-bind="value: newTaskText" placeholder="What needs to be done?" />     <button type="submit">Add</button> </form> <ul data-bind="foreach: tasks, visible: tasks().length > 0">     <li>         <input type="checkbox" data-bind="checked: isDone" />         <input data-bind="value: title, disable: isDone" />         <a href="#" data-bind="click: $parent.removeTask">Delete</a>     </li>  </ul> |
| 執行結果 | Javascript(ViewModel) |
|  | function Task(data) {     this.title = ko.observable(data.title);     this.isDone = ko.observable(data.isDone); } function TaskListViewModel() {     // 資料成員     var self = this;     self.tasks = ko.observableArray([]);     self.newTaskText = ko.observable();     self.incompleteTasks = ko.computed(function() {         return ko.utils.arrayFilter(self.tasks(), function(task) { return !task.isDone() });     });     // 操作方法     self.addTask = function() {         self.tasks.push(new Task({ title: this.newTaskText() }));         self.newTaskText("");     };     self.removeTask = function(task) { self.tasks.remove(task) };      // Load initial state from server, convert it to Task instances, then populate self.tasks     $.getJSON("/tasks", function(allData) {         var mappedTasks = $.map(allData, function(item) { return new Task(item) });         self.tasks(mappedTasks);     });   } ko.applyBindings(new TaskListViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **將資料傳回主機(方法1:表單)**  要將資料傳送到主機,標準的做方就是用表單提交。  在這範例中,我們使用一個呈現Model資料的<form>表單，然後可以簡單讓使用者發送表單資料到伺服器，這是非常容易做到的。例如，在View加入以下程式碼：  <form action="/tasks/saveform" method="post">     <input type="hidden" name="tasks" data-bind="value: ko.toJSON(tasks)" />     <button type="submit">Save</button> </form>  現在嘗試修改畫面上的資料，新增、修改、完成、刪除等，相依追蹤會自動將最新JSON更新到<input>之內。當然使用者按下"Save"送出資料，伺服器端會接收JSON資料。  注:可將<input>改為<textarea>觀查其內容的改變。  Ko.JSON(tasks)的內容  [{"title":"Wire the money to Panama","isDone":true},{"title":"Get hair dye, beard trimmer, dark glasses and \"passport\"","isDone":false},{"title":"Book taxi to airport","isDone":false},{"title":"Arrange for someone to look after the cat","isDone":false}]  主機接到tasks的資料,再去解析json資料並完成工作。 | <h3>Tasks</h3> <form data-bind="submit: addTask">     Add task: <input data-bind="value: newTaskText" placeholder="What needs to be done?" />     <button type="submit">Add</button> </form> <ul data-bind="foreach: tasks, visible: tasks().length > 0">     <li>         <input type="checkbox" data-bind="checked: isDone" />         <input data-bind="value: title, disable: isDone" />         <a href="#" data-bind="click: $parent.removeTask">Delete</a>     </li>  </ul> You have <b data-bind="text: incompleteTasks().length">&nbsp;</b> incomplete task(s) <span data-bind="visible: incompleteTasks().length == 0"> - it's beer time!</span> <form action="/tasks/saveform" method="post">     <input type="hidden" name="tasks" data-bind="value: ko.toJSON(tasks)" />     <button type="submit">Save</button> </form> |
| 執行結果 | Javascript(ViewModel) |
|  | function Task(data) {     this.title = ko.observable(data.title);     this.isDone = ko.observable(data.isDone); } function TaskListViewModel() {     // 資料成員     var self = this;     self.tasks = ko.observableArray([]);     self.newTaskText = ko.observable();     self.incompleteTasks = ko.computed(function() {         return ko.utils.arrayFilter(self.tasks(), function(task) { return !task.isDone() });     });     // 操作方法     self.addTask = function() {         self.tasks.push(new Task({ title: this.newTaskText() }));         self.newTaskText("");     };     self.removeTask = function(task) { self.tasks.remove(task) };      // Load initial state from server, convert it to Task instances, then populate self.tasks     $.getJSON("/tasks", function(allData) {         var mappedTasks = $.map(allData, function(item) { return new Task(item) });         self.tasks(mappedTasks);     });   } ko.applyBindings(new TaskListViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **將資料傳回主機(方法2:Ajax)**  替代<form>表單的另一方法，就是使用Ajax請寫傳送Model資料。例如，先移除或註解前面新增的<form>表單，然後加入一個<button>元素程式碼：  <button data-bind="click: save">Save</button>  然後在TaskListViewModel實作save方法：      self.save = function() {         $.ajax("/tasks", {             data: ko.toJSON({ tasks: self.tasks }),             type: "post", contentType: "application/json",             success: function(result) { alert(result) }         });     };  Ajax請求成功，只簡單使用alert來顯示伺服器的回應訊息，這能讓我們知道伺服器是真的有接收及瞭解JSON資料。 | <h3>Tasks</h3> <form data-bind="submit: addTask">     Add task: <input data-bind="value: newTaskText" placeholder="What needs to be done?" />     <button type="submit">Add</button> </form> <ul data-bind="foreach: tasks, visible: tasks().length > 0">     <li>         <input type="checkbox" data-bind="checked: isDone" />         <input data-bind="value: title, disable: isDone" />         <a href="#" data-bind="click: $parent.removeTask">Delete</a>     </li>  </ul> You have <b data-bind="text: incompleteTasks().length">&nbsp;</b> incomplete task(s) <span data-bind="visible: incompleteTasks().length == 0"> - it's beer time!</span> <button data-bind="click: save">Save</button> |
| 執行結果 | Javascript(ViewModel) |
|  | function Task(data) {     this.title = ko.observable(data.title);     this.isDone = ko.observable(data.isDone); } function TaskListViewModel() {     // 資料成員     var self = this;     self.tasks = ko.observableArray([]);     self.newTaskText = ko.observable();     self.incompleteTasks = ko.computed(function() {         return ko.utils.arrayFilter(self.tasks(), function(task) { return !task.isDone() });     });     // 操作方法     self.addTask = function() {         self.tasks.push(new Task({ title: this.newTaskText() }));         self.newTaskText("");     };     self.removeTask = function(task) { self.tasks.remove(task) };      // Load initial state from server, convert it to Task instances, then populate self.tasks     self.save = function() {         $.ajax("/tasks", {             data: ko.toJSON({ tasks: self.tasks }),             type: "post", contentType: "application/json",             success: function(result) { alert(result) }         });     };      // Load initial state from server, convert it to Task instances, then populate self.tasks     $.getJSON("/tasks", function(allData) {         var mappedTasks = $.map(allData, function(item) { return new Task(item) });         self.tasks(mappedTasks);     });   } ko.applyBindings(new TaskListViewModel()); |

|  |  |
| --- | --- |
| 說明 | HTML(View) |
| **追踨刪除:**  如果使用者刪除了一些資料,如何將主機上的資料也刪除掉?如果資料在用戶端是直接刪除,到了主機,就必須用比對的方式才能知道那些資料不見了,這樣做會增加複雜度。如果用戶端能明確記錄被刪除的資料再通知主機,這種方式會比較理想。  我們可以改寫刪除方法,透過destroy方法來刪除項目,如下:  // self.removeTask = function(task) { self.tasks.remove(task) }; self.removeTask = function(task) { self.tasks.destroy(task) };  它不實際的從tasks中刪除陣列元素,它只是增加一個\_destroy的屬性值(為true)到Task中。而ko中的相關繫結會自動忽略有這個屬性值為true的內容(例如:foreach),因此,也不會在畫面上出現。  當資料被提交到主機後,這些帶有\_destroy的項目,就能很容易的被找到,並做相對應的處理。  另外要注意的一點是,由於tasks裡的資料並未真的被刪除,因此在計算未完成的工作項目計算時,就要特別處理:  return ko.utils.arrayFilter(self.tasks(), function(task) { return !task.isDone() && !task.\_destroy }); | <h3>Tasks</h3> <form data-bind="submit: addTask">     Add task: <input data-bind="value: newTaskText" placeholder="What needs to be done?" />     <button type="submit">Add</button> </form> <ul data-bind="foreach: tasks, visible: tasks().length > 0">     <li>         <input type="checkbox" data-bind="checked: isDone" />         <input data-bind="value: title, disable: isDone" />         <a href="#" data-bind="click: $parent.removeTask">Delete</a>     </li>  </ul> You have <b data-bind="text: incompleteTasks().length">&nbsp;</b> incomplete task(s) <span data-bind="visible: incompleteTasks().length == 0"> - it's beer time!</span> <button data-bind="click: save">Save</button> |
| 執行結果 | Javascript(ViewModel) |
|  | function Task(data) {     this.title = ko.observable(data.title);     this.isDone = ko.observable(data.isDone); } function TaskListViewModel() {     // 資料成員     var self = this;     self.tasks = ko.observableArray([]);     self.newTaskText = ko.observable();     self.incompleteTasks = ko.computed(function() {         return ko.utils.arrayFilter(self.tasks(), function(task) { return !task.isDone() && !task.\_destroy });     });     // 操作方法     self.addTask = function() {         self.tasks.push(new Task({ title: this.newTaskText() }));         self.newTaskText("");     };     // self.removeTask = function(task) { self.tasks.remove(task) };     self.removeTask = function(task) { self.tasks.destroy(task) };      // Load initial state from server, convert it to Task instances, then populate self.tasks     self.save = function() {         $.ajax("/tasks", {             data: ko.toJSON({ tasks: self.tasks }),             type: "post", contentType: "application/json",             success: function(result) { alert(result) }         });     };      // Load initial state from server, convert it to Task instances, then populate self.tasks     $.getJSON("/tasks", function(allData) {         var mappedTasks = $.map(allData, function(item) { return new Task(item) });         self.tasks(mappedTasks);     });   } ko.applyBindings(new TaskListViewModel()); |