ThinkMVC

A light-weight JavaScript framework

Introduction

It is a fact that demands for use experience of web applications have been escalating. Browsers have also been strongly developed to support more functions. In this trend, JavaScript takes much more responsibilities at the client side, and JavaScript code become much heavier. Tool libraries such as jQuery are not enough for accelerating the development. How to organize and structure the JavaScript code well and how to make code readable and extendable are challenges we are now facing. Besides that, we probably have to consider performance and memory leak issues.

Backbonejs is a very popular MV* framework. It provides such classes: event, model, view, collection and router. It deeply relies on another library underscorejs which provides a lot of functions. What I mean backbonejs deeply relies on underscorejs, it is that underscorejs should be loaded to your page before loading backbonejs. Backbonejs gets many benefits from it. When working with backbonejs, you need ensure that classes are created before they are used. So you should keep the definition and creation of classes in order. Generally code is developed by different people or teams, and may live in different resource files, or is combined in one file later. It is not easy to ensure this order especially classes are created in different files. In this case, you may need another library requirejs which allows the definition to happen in disorder but creation in order.

Another drawback of backbonejs, I think, it doesn't provide separate controllers which are only supposed to dispatch user requests. Backbone.View does this work instead but is also responsible for rendering pages. In my opinion, view's job is just render pages. The bridge between user requests from client and models is controllers.

ThinkMVC is a light-weight MVC framework and overcomes the drawbacks described above. ThinkMVC needs jQuery which can help querying DOM elements and bind DOM events, but it doesn't deeply replies on it, in fact it doesn't deeply relies on any other libraries. It splits code into three layers: model which organizes business logic code, view which renders pages and controller which responses and dispatches user requests.

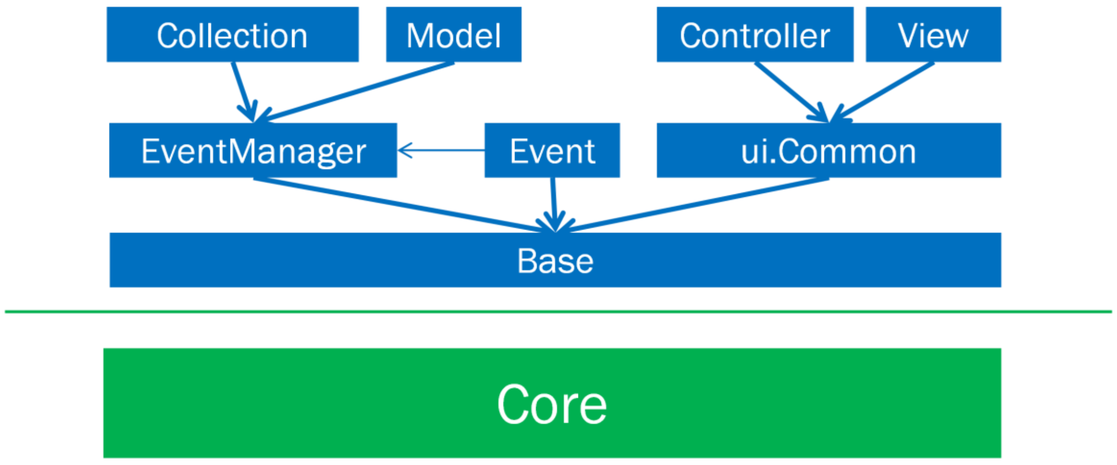
Also ThinkMVC tries to resolve some typical performance and memory leak issues. One performance issue is loading JavaScript resources inefficiently. A typical memory leak pattern is closure. We will discuss them in later chapters.

Well, if you get interested in ThinkMVC, let's begin the trip!

Architecture

Following chart briefly shows the architecture of the framework. It can be seen there are two layers: one is the core layer which offers a very limited set of APIs which are used to manage JavaScript resources and declare classes; the other layer provides a set of classes which can be extended by custom code.

You can organize your code by inheriting these classes: Model, Collection, Controller and View. Controllers listen to events from UI and hand off to models. Models handle the business logic and then notify views bound to them. Finally views update the UI.



Basics

In this section, I would like to introduce the essential stuff of ThinkMVC. You can learn how ThinkMVC works and how to write code with it.

Getting set up

Before showing code examples, we need import the framework JavaScript resource into the page. You will find it can't be easier. See following simple HTML page.

```
<!DOCTYPE HTML>
<html>
   <head>
      <title>Demo</title>
      <script type="text/javascript" data-config="http://host/config.js" src="
http://host/thinkmvc.js" async></script>
   </head>
   <body data-page="home">
   </body>
</html>
```

In order not to block the page rendering, better to assign an attribute 'async' to the script markup. You may notice another attribute 'data-config' on the script markup. It tells the framework that the config.js needs be downloaded to the page. The structure of config.js is something like as below.

```
// config.js
TM.configure({
   baseUrl: 'http://host',

   dependencies: {
      myApp: ['jquery']
   },

   modules: {
      myApp: '/app.js',
      jquery: 'http://code.jquery.com/jquery-2.0.3.js'
   },
    pages: {
        home: {
            controller: 'com.HomeController',
            module: ' myApp'
        }
    }
});
```

TM is the global object which represents the core layer of ThinkMVC. It only exports two APIs: configure and declare. As you can see from above code, the API 'configure' configures the whole application with following stuff.

1. Define the controller for each page which is indicated by the attribute 'data-page' bound on body markup, e.g.

   ```
   home: {
       controller: 'com.HomeController',
       module: ' myApp'
   }
   ```

   The page 'gateway' is managed by the controller '*com.GatewayController*'. This controller is from a JavaScript module named 'myApp'.

2. So where is myApp from? In the modules section, the module 'myApp' is mapped to a relative path '/app.js', the framework will load it by accessing the absolute url http://host/app.js combined by baseUrl and relative module path. Of course you can assign an absolute path for the module, and the framework won't combine with baseUrl then.

   ```
   baseUrl: 'http://host',
   modules: {
       myApp: '/app.js',
       jquery: 'http://code.jquery.com/jquery-2.0.3.js'
   }
   ```

3. The module your controller may depend on other modules, namely before running the code of controller, other modules should be loaded and executed. We can manage the dependencies among modules in the section 'dependencies'.

   ```
   dependencies: {
       myApp: ['jquery']
   }
   ```

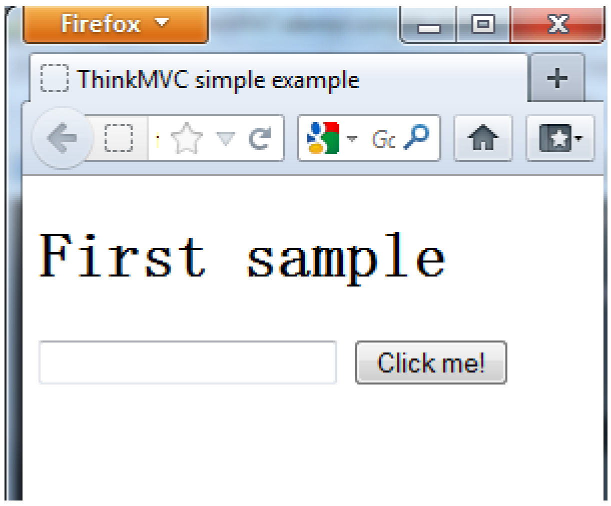
Create your first class

After we configure the application and bind the controller 'com.HomeController' to the page 'home', next job is to create the controller class.

```
TM.declare('com.HomeController').inherit('thinkmvc.Controller');
```

Above code declares a class 'HomeController' under the namespace 'com', and this class inherits from 'thinkMVC.Controller'. It is meaningless if we create a class that just inherits from the super controller class. Let's do more work in a very simple sample.

Suppose on the home page, customers can enter their names and click a button, and then the names will be shown on the page.



HTML code:

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta charset="utf-8"/>
        <title>ThinkMVC simple example</title>
        <script  src="../thinkmvc.js"  data-config="simple_exam_config.js"  type="text/javascript"
async></script>
    </head>
    <body data-page="home">
        <h1>First sample</h1>
        <div id="enter-fields">
            <h2 style="display: none;" id="yourName"></h2>
            <input type="text" id="username"/>
            <button id="btn">Click me!</button>
        </div>
    </body>
</html>
```

simple_exam_config.js:

```
TM.configure({
    baseUrl: '',

    dependencies: {
        simpleExam: ['jquery']
    },

    modules: {
        jquery: 'http://code.jquery.com/jquery-2.0.3.min.js',
        simpleExam: 'simple_exam.js'
    },

    pages: {
        home: {
            controller: 'com.HomeController',
            module: 'simpleExam'
        }
    }
});
```

simple_exam.js:

```
TM.declare('com.HomeController').inherit('thinkmvc.Controller').extend({
    events: {
        'click #btn': 'showUserName'
    },

    rootNode: '#enter-fields',

    selectors: {
        userName: '#username',
        yourName: '#yourName'
    },

    initialize: function() {
        this.invoke(' thinkmvc.Controller:initialize');
        this._el.$userName.val('').focus();
    },

    showUserName: function() {
        var el = this._el, userName = el.$userName.val(),
            message = userName ? 'Hello, ' + userName : 'Please enter your name.';
        el.$yourName.html(message).show();
    }
});
```

Look at the simple_exam.js, in the home controller we define the event ‘click’ for the button (‘#btn’ is button’s selector by which jQuery can find it), and its callback’s name is ‘showUserName’.

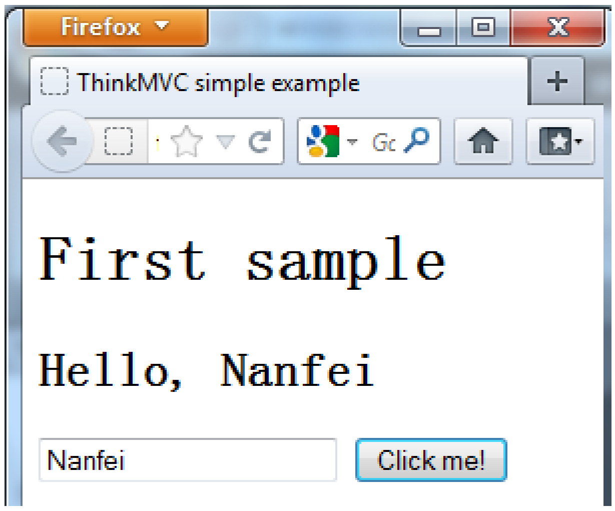
We also define some UI element variables in selectors. The purpose is to cache jQuery querying results so you don’t need query them for times in methods. E.g. { userName: ‘#username’ } tells the controller to look for the element whose id is ‘username’ and assign the result to userName. In methods you can access the element via this._el.$userName which is a jQuery object.

ThinkMVC asks jQuery to bind window events and query UI elements. There is really a lot of knowledge related to this, so it’s better for you to learn more from jQuery site if you are unfamiliar with jQuery.

The first instance of ThinkMVC classes

You may be curious about the above simple sample that we just create a class but we don’t explicitly create the instance of the controller class in anywhere. However the code works as expected, when you click the button on the page, you get a message. So where is first instance created? ThinkMVC itself does!

Remember the attribute ‘data-page’ bound to the body markup? In the config.js, the controller is bound to the page, so ThinkMVC knows the controller class path of which the instance should be created. It is the first instance that ThinkMVC creates after the page HTML is parsed.



First sample

Hello, Nanfei

Initialization

In the simple_exam.js of above sample, a method ‘initialize’ is defined for the controller class. This method is automatically called when the instance is created. In the method, two things are done.

1. The method ‘invoke is called with the argument ‘initialize’. This method calls the parent’s method ‘initialize’ under current execution context.

   Usually we need override the parent’s methods but run them as a part in overridden methods, calling ‘invoke’ is a good choice.

2. Another thing done in the method ‘initialize’ of child class is empty the input field and make it focused.

Models and Views

The sample I show to you is quite simple, let’s add more to it. There is a requirement that the more information should be displayed when users enter their names, e.g. age and gender. If user’s age is greater than 20, education status is required to show, too. So look, we have some business logic to handle now. To make the code clean, it’s better to separate it to different parts according to their functions. We can put the business logic code to a model.

```
TM.declare('com.models.User').inherit('thinkmvc.Model').extend({
  viewPath: 'com.views.UserView',

  initialize: function(name) {
    this.invoke('thinkmvcModel:initialize');
    this._name = name;
  },

  retrieveDetails: function() {
    var msg, name = this._name;
    if (!name) {
      msg = 'Please enter your name.';
    } else {
      var age = name === 'Nanfei' ? 31 : (name === 'Eric' ? 16 : 27);
      msg = 'Hello, ' + name + ', your age is ' + age;
```

```
            if (age > 20) {
                msg += ', your edu is master';
            }
        }

        this.trigger('show-details', msg);
    }
});
```

The view ‘com.views.UserView’ is bound to User model via the attribute ‘viewPath’. Now we create the view class. We bind an event ‘show-details’ to User model, when the model triggers this event, the corresponding callback will be invoked.

```
TM.declare('com.views.UserView').inherit('thinkmvc.View').extend({
    events: {
        'show-details': 'showDetails'
    },

    selectors: {
        yourName: '#yourName'
    },

    showDetails: function(evt) {
        this._el.$yourName.html(evt.data).show();
    }
});
```

Next work is to do a change to the controller’s method ‘showUserName’ as below. In the method, we create an instance of User model and call its API ‘retrieveDetails’.

```
    showUserName: function() {
        var user = this.U.createInstance('com.models.User', this._el.$userName.val());
        user.retrieveDetails();
    }
```