

六 Todoify：现代的前端开发方式

在这一章结束的时候，你会学会下面这些技术：

1. 使用 Karma 测试运行器来自动的执行 JavaScript 测试
2. 使用 bower 作为前端 JavaScript 库的依赖管理工具
3. 使用 Gulp 作为构建脚本，使得很多构建任务自动化
4. 简单的测试驱动开发的方式
5. 如何开发一个 jquery 插件

Karma

Karma 是一个 JavaScript 的测试运行器，使用 Karma 可以很方便的运行测试（方便到你感觉不到它的实际存在）。运行器的意思是，Karma 本身不会去执行具体的测试，测试框架才会。Karma 的作用就是启动浏览器，然后启动测试框架去执行预先定义好的测试套件。

1. 基于真实的浏览器，并且支持多个浏览器
2. 自动监听测试/实现文件的变化，并在变化之后运行测试
3. 支持多种测试框架，比如 Jasmine，Mocha 等
4. 容易调试
5. 可以方便的与持续集成服务器集成

我们在本章会使用 Jasmine 作为测试框架和 Karma 一起来运行。当然，首先需要安装 Karma。Karma 是一个 Node.js 的包，也就是说，你的本地机器上需要安装 Node.js。

如果是 Mac OSX，安装过程十分简单：

```
$ brew install node
```

如果是 Linux，可以通过安装预编译包，或者通过源代码编译的方式。但是一般来说，预编译包的版本都会比较低，所以推荐使用源码编译的方式安装。安装完成之后，你会得到一个 npm 的可执行脚本。npm 是 Node 的包管理器（Node Package Manager），用来安装基于 Node 的程序包。

我们使用 npm 来安装 Karma：

```
$ npm install karma
```

安装完成之后，当前目录下会多出一个 node_modules 的目录，里边会有 Karma 的包。我们可以再安装一个 karma-cli 的包，-g 参数表示将 Karma 安装在全局环境中，以便其他的项目使用。

```
$ npm install karma-cli -g
```

karma-cli 会在当前目录的 node_modules 中查找 karma 的包，并尝试启动这个 Karma。如果当前目录没有，则会向全局目录查找。这样做的好处是，每个项目都可以使用不同版本的 Karma。

安装完成之后，你应该可以通过下面这条命令来查看 Karma 的版本：

```
$ karma --version  
Karma version: 0.10.10
```

前端依赖管理

Bower 是一个基于 Node.js 的依赖管理工具，它也是一个 npm 的包，因此安装十分简单，由于我们需要在所有项目中都可以使用 bower，因此将其安装在全局环境中：

```
$ npm install -g bower
```

安装完成之后，可以通过 bower search 来搜索需要的包，比如：

```
$ bower search backbone.js
```

通常你会得到类似与这样的结果：

Search results:

```
    jquery.backbone.js  
git://github.com/fanlia/jquery.backbone.js.git
```

典型的应用场景可能会是这样的，首先新建一个项目目录，然后在该目录中运行 bower 的 init 命令：

```
$ mkdir -p listing  
$ cd listing  
$ bower init
```

Bower 会问你一些问题，比如项目名称，项目入口点，作者信息之类，然后生成一个 bower.json 文件：

```
{  
  "name": "listing",  
  "version": "0.0.0",  
  "authors": [  
    "Qiu Juntao <juntao.qiu@gmail.com>"  
  ],  
  "main": "src/app.js",  
  "license": "MIT",  
  "ignore": [  
    "**/*.*",  
    "node_modules",  
    "bower_components",  
    "test",  
    "tests"  
  ]  
}
```

比如我们需要安装 jQuery 和 underscore.js，则很简单的运行 bower install 命令即可：

```
$ bower install jquery  
$ bower install underscore
```

而且，使用 bower 可以安装指定版本的包，比如先通过 info 子命令查看

```
$ bower info jquery#1.10.*

bower cached      git://github.com/jquery/jquery.git#1.10.2
bower validate    1.10.2 against
git://github.com/jquery/jquery.git#1.10.*

{
  name: 'jquery',
  version: '1.10.2',
  description: 'jQuery component',
  keywords: [
    'jquery',
    'component'
  ],
  main: 'jquery.js',
  license: 'MIT',
  homepage: 'https://github.com/jquery/jquery'
}
```

然后执行：

```
$ bower install jquery#1.10.*
```

即可安装 1.10.2 版本的 jQuery，默认的 bower 会下载安装最新版本的 jQuery。

如果需要团队中的其他成员可以在本地恢复我们的环境，需要在 bower.json 中指定 dependencies 小节：

```
"dependencies": {
  "jquery": "1.10.*",
  "underscore": "~1.5.2"
}
```

默认的，所有的 JavaScript 包都被安装到了本地的 bower_components 目录下。当然可以通过下面的方式来修改：

在当前目录创建一个.bowerrc 文件，然后修改该文件的内容为：

```
{
  "directory": "vendor"
}
```

然后执行 bower install，bower 会自动创建目录 vendor，并将内容下载到该目录中。

如果有了 bower.json 文件，那么即使本地的 bower_components（或者在.bowerrc 中自定义的目录）目录不存在，或者其中的包内容过期了，也可以使用 bower install 将其更新。

构建工具

```
$ npm install gulp gulp-util -g
```

搭建工程

首先创建基本的目录结构，我们需要为第三方的库放到一个单独的目录 vendor 中，而测试文件需要放到另一个 spec 目录中：

```
$ mkdir -p src/vendor
$ mkdir -p spec/
```

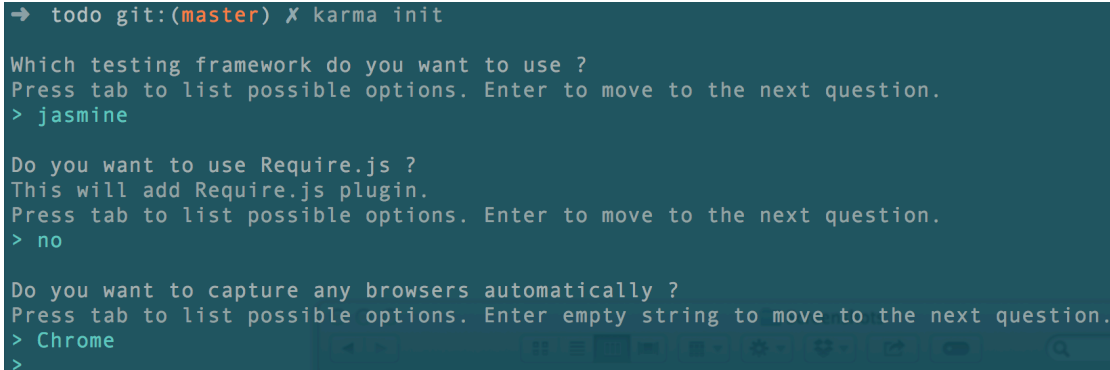
然后创建一个基本的 HTML 文件，这个文件将作为最终用来展示的入口：

```
$ touch index.html
```

我们在使用 Karma 之前，需要告诉 Karma，哪些文件是源代码，哪些文件是测试代码，这样 Karma 才可以动态的加载这些文件，并监听文件的改动。这个动作可以通过 init 子命令来完成：

```
$ karma init
```

Karma 会问一些问题，比如使用哪种浏览器，哪种测试框架等：



```
→ todo git:(master) X karma init

Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next question.
> jasmine

Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options. Enter to move to the next question.
> no

Do you want to capture any browsers automatically ?
Press tab to list possible options. Enter empty string to move to the next question.
> Chrome
>
```

填写完成之后，Karma 会在当前目录下生成一个 karma.conf.js 文件，这是一个可以被 Node.js 执行的 JavaScript 文件。karma.conf.js 中事实上只是为 config 设置了一些配置：

```
module.exports = function(config) {
  config.set({

  });
};
```

Karma 会在执行时读取这些定义，并应用这些配置。比如这里有一些常见的选项：

```
{
  frameworks: ['jasmine'],
  files: [
    "src/vendor/underscore/underscore.js",
    "src/vendor/jquery/jquery.js",
    'src/vendor/jasmine-jquery/lib/jasmine-jquery.js',
    "src/todoify.js",
    "spec/**/*.spec.js"
  ],
  exclude: [

  ],
  reporters: ['progress'],
```

```

port: 9876,
colors: true,
logLevel: config.LOG_INFO,
autoWatch: true,
browsers: ['Chrome'],
singleRun: false
}

```

frameworks 来配置测试框架（比如 Jasmine, Mocha 等）；files 来指定测试中需要加载哪些插件；autoWatch 选项指定是否监听文件变化，值为 true 时，当 files 中指定的任何文件有变化时，karma 会被触发，然后执行所有的测试。

你可以在随后编辑这个文件的内容。这时候可以做一个简单的测试，来验证 Karma 的安装情况。执行 start 子命令：

```

→ todo git:(master) X karma start
INFO [karma]: Karma v0.12.16 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 35.0.1916 (Mac OS X 10.9.3)]: Connected on socket 0jzU9N290xa_zVns1E88
Chrome 35.0.1916 (Mac OS X 10.9.3): Executed 0 of 0 ERROR (0.006 secs / 0 secs)

```

这时候会看到一个错误，Karma 抱怨找不到任何的测试，我们暂时不用管这个错误。Karma 还会启动配置过的（通过 browsers 选项）浏览器：



接下来，我们需要安装 jquery 和 underscore，我们需要这两个包被安装到当前目录的 src/vendor 目录下，因此需要编辑 .bowerrc 文件：

```

{
  "directory": "src/vendor"
}

```

然后运行 install：

```
$ bower install jquery underscore
```

当前的目录结构应该是这样的：

▼	todo	Today, 9:05 AM
	index.html	Today, 8:47 AM
	karma.conf.js	Today, 8:53 AM
▶	spec	Today, 8:47 AM
▼	src	Today, 9:09 AM
▼	vendor	Today, 9:09 AM
▶	jquery	Today, 9:09 AM
▶	underscore	Today, 9:09 AM

好了，有了这些基础设施之后，我们就可以来做真正的开发工作了。

测试驱动开发

简而言之，测试驱动开发是这样一种开发模式：

1. 在编写任何产品代码之前，先编写测试代码
2. 运行该测试，这时候测试必然会失败
3. 用最简单的方式来修复这个失败
4. 重构代码，使得代码更加清晰，容易扩展

这种开发方式的初衷是以最小的付出来完成功能，并且会获得很多额外的好处：

1. 编写完善的测试
2. 避免过度设计
3. 在修复 bug 的时候更加有信心，你可以立刻知道修改会引起哪些其他的影响

通常来看，测试驱动开发可以很大的提高代码质量，提高开发效率，降低修复 bug 时的成本。但是测试驱动开发对开发人员的要求也比较高，特别是在环境的配置上需要花费一些“额外”的时间。

比如：

1. 如何快速的运行测试
2. 如何在发生错误时快速的定位
3. 测试代码需要用哪种语言编写
4. 是否能在持续集成环境中也很容易的执行这些测试

这些问题在服务器端开发中已经得到了很好的解决，但是前端开发，直到最近才有了改善，我们这里会使用的 Karma 运行器，Jasmine 测试框架等，都为前端开发中使用测试驱动的方式提供了极大的便利。

实例 Todoify

Web 前端现在正处在类似于生物学上的寒武纪：众多的框架不断的被开发出来，有偏重于大而全的“重量级”框架，有强调只做一件事情的工具包，有严格遵循 MVC 的小巧框架，又有为客户端做过适配改进的 MVVM。

在进入这些令人眼花缭乱的世界之前，我们需要一起看一个简单的实例，在这个例子中，我们关注于功能以及小巧两个方面。在下面这个例子中，我们将开发一个 jQuery 的插件，这个插件可以将一个普通的 HTML 输入框变成一个待办事项的控件，使用这个控件，用户可以输入一条待办事项，或者删除待办事项列表中的一条记录等。如果用户预先已经有了一组待办事项的数据（一个数组或者列表对象），这个控件还可以用可视化的方式将他们展现出来。

在这个实例中，我们会用到最常用的，而且非常轻量级的两个 JavaScript 库：jQuery 和 underscore。

比如用户已经有了这样几条数据：

```
var mydata = [  
  "Hello, darkness",  
  "Tomorrow is another day",  
  "Never say never"  
];
```

那么，当使用了这个插件之后：

```
$("#item-input").todoify({  
  data: mydata,  
  container: "#item-todos"  
});
```

会得到这样一个列表：

Hello, darkness	X
Tomorrow is another day	X
Never say never	X

如果你之前没有任何的 jQuery 开发经验，不必担心，我们会从头开始介绍所有的知识：

jQuery 是一个 JavaScript 的函数库，虽然体积小巧，但是功能非常强大。它提供非常简洁的方式来选择 DOM 元素，操作 DOM 元素，注册事件，实现元素的动画，发送 ajax 请求等等。jQuery 事实上已经成为了前端开发的标准，事实上，已经很难见到一个不使用 jQuery 的网站了。

underscore 是另一个小巧的函数库，但是它更关注在 JavaScript 中对数据的操作上。使用 underscore 可以极大程度上精简对集合的操作，比如从集合中选择出符合某个条件的子集，去掉数组中的重复项，将数组中得所有元素都转化成另外一个形式等等。

underscore 的一些特性

underscore 对外暴露的对象名是一个下划线（_），也就是 underscore 这个单词本身的意义。

抽取对象数组中的某个属性，并组成一个新的数组。这个特性在很多情况下都非常使用，一般后台返回的数据可能包含很多与展现无关的部分，或者对当前组件无关的数据：

```
var vendors = [  
  {  
    id: 1,  
    name: "Dell",  
    address: "U.S"  
  }, {  
    id: 2,
```

```

        name: "HP",
        address: "U.S"
    }, {
        id: 3,
        name: "Lenovo",
        address: "China"
    }
];
var vendorNames = _.pluck(vendors, "name");
//["Dell", "HP", "Lenovo"]
var vendorIds = _.pluck(vendors, "id");
//[1, 2, 3]

```

取出一个对象的所有键组成的数组：

```

var vendor = {
    id: 3,
    name: "Lenovo",
    address: "China"
};
var keys = _.keys(vendor);
var values = _.values(vendor);

```

会得到["id", "name", "address"]，而_.values(vendor)会得到[3, "Lenovo", "China"]。

template 方法提供一个简单的模板：模板定义 HTML 的结构，以及一些占位符。_.template()会将这个静态模板编译成一个函数。在这个生成的函数上，将数据以参数的形式传入，就会得到最终的字符串（也就是视图）：

```

var tpl = "<p>Vendor <%= name %>, from <%= address %></p>";
var tpl_func = _.template(tpl);
tpl_func({name: "Lenovo", address: "China"});

```

比如，这个例子中，模板 tpl 中定义了一个<p>元素，而且会预期传入的对象中包含 name 和 address 两个属性。调用 template 之后得到的函数为 tpl_func，将 data 传入这个函数会得到：

```
<p>Vendor Lenovo, from China</p>
```

而用下列数据调用 tpl_func：

```
tpl_func({name: "Dell", address: "U.S"});
```

则得到的输出为：

```
<p>Vendor Dell, from U.S</p>
```

默认的 template 使用的是 ERB 的模板语法，如果你更喜欢自己熟悉的模板技术，比如 Mustache，只需要做很简单的配置即可：

```

_.templateSettings = {
    interpolate: /\{\{(.+?)\}\}/g
};
var tpl = "<p>Vendor {{ name }}, from {{ address }}</p>";
var tpl_func = _.template(tpl);
tpl_func({name: "Lenovo", address: "China"});

```


当这两个即为轻巧的函数库结合在一起时，会发挥出强大的威力。

jQuery 插件基础知识

简单流程

通常使用 jQuery 的流程是这样的：通过选择器选择出一个 jQuery 对象（集合），然后为这个对象应用一些预定义的函数，如：

```
$(".article .title").mouseover(function() {  
    $(this).css({  
        "background-color": "red",  
        "color": "white"  
    });  
});
```

我们如果要定义自己的插件，预期其被调用的方式和此处的 `mouseover` 并无二致。这需要将我们定义的函数附加到 jQuery 对象的 `fn` 属性上：

```
$.fn.hltitle = function() {  
    this.mouseover(function() {  
        $(this).css({  
            "background-color": "red",  
            "color": "white"  
        });  
    })  
};  
  
$('.article .title').hltitle();
```

jQuery 的一个很明显的特点是其链式操作，即每次调用完成一个函数/插件之后仍然会返回 jQuery 对象本身，这个需要我们在插件函数的最后一行返回 `this`。这样插件的使用者会像使用其他函数/插件一样很方便的将调用连起来。

另外一个问题是注意命名冲突（在 JavaScript 中，`$` 是一个合法的标示符，而且被众多的 JavaScript 库在使用），所以可以通过匿名执行函数来避免：

```
(function($) {  
    $.fn.hltitle = function() {  
        //...  
    }  
})(jQuery);
```

需要注意的问题

上面是一个最简单的插件定义，为了插件更加灵活，我们需要尽可能多的将配置项暴露给插件的用户，比如提供一些默认选项，如果用户不提供配置，则插件按照默认配置来工作，但是用户可以通过修改配置来定制插件的行为：

```
(function($) {  
    $.fn.hltitle = function(options) {
```

```
var defaults = {
  "background-color": "red",
  "color": "white"
};
var settings = $.extend(defaults, options);
return this.mouseover(...);
};
}(jQuery));
```

Todoify

我们目前的一个大的方向就是使用 jQuery 的插件机制来完成 Todoify 的功能，这是目前唯一明确的背景，因此我们的第一个测试可以从这一点来出发：

第一个测试：运行起来

那么第一个测试就很明确了：

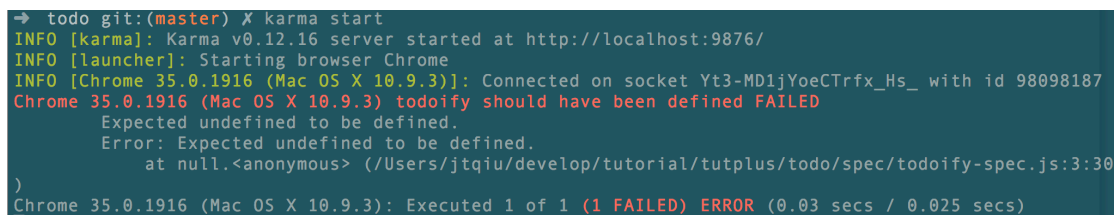
```
describe('todoify', function() {
  it('should have been defined', function() {
    expect($.fn.todoify).toBeDefined();
  });
});
```

我们测试 \$.fn 上已经附加了我们的插件函数，这样当别人使用插件时，就可以直接使用选择器选中的元素，然后调用 todoify 即可。将这个片段保存为 todoify-spec.js 即可。

此时，karma.conf.js 中的配置应该是这样的：

```
files: [
  "src/vendor/jquery/jquery.js",
  "src/vendor/underscore/underscore.js",
  "src/todoify.js",
  "spec/**/*.spec.js"
],
```

当然，这时候运行 Karma，会看到一个错误：



```
→ todo git:(master) X karma start
INFO [karma]: Karma v0.12.16 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 35.0.1916 (Mac OS X 10.9.3)]: Connected on socket Yt3-MD1jYoeCTrfx_Hs_ with id 98098187
Chrome 35.0.1916 (Mac OS X 10.9.3) todoify should have been defined FAILED
    Expected undefined to be defined.
    Error: Expected undefined to be defined.
      at null.<anonymous> (/Users/jtqiu/develop/tutorial/tutplus/todo/spec/todoify-spec.js:3:30)
)
Chrome 35.0.1916 (Mac OS X 10.9.3): Executed 1 of 1 (1 FAILED) ERROR (0.03 secs / 0.025 secs)
```

提示 todoify-spec.js 的第 3 行有错误，期望一个 undefined 的值为 defined。这个错误很容易修复，我们在 src/todoify.js 中加入这样的代码：

```
$.fn.todoify = function() {}
```

再次运行测试：

```

→ todo git:(master) X karma start
INFO [karma]: Karma v0.12.16 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 35.0.1916 (Mac OS X 10.9.3)]: Connected on socket Yt3-MD1jYoeCTrfx_Hs_ with id 980981
Chrome 35.0.1916 (Mac OS X 10.9.3) todoify should have been defined FAILED
Expected undefined to be defined.
Error: Expected undefined to be defined.function()
    at null.<anonymous> (/Users/jtqiu/develop/tutorial/tutplus/todo/spec/todoify-spec.js:3
)
再次运行测试:
Chrome 35.0.1916 (Mac OS X 10.9.3): Executed 1 of 1 (1 FAILED) ERROR (0.03 secs / 0.025 secs)
INFO [watcher]: Changed file "/Users/jtqiu/develop/tutorial/tutplus/todo/src/todoify.js".
Chrome 35.0.1916 (Mac OS X 10.9.3): Executed 1 of 1 SUCCESS (0.015 secs / 0.013 secs)

```

可以看到最后一行中，测试已经执行成功了。应该注意的是，这个过程中，Karma 一直运行在后台，一旦我们修改了测试或者实现代码，Karma 都会自动运行一次测试。这样我们可以得到实时的反馈：哪一行代码以何种方式运行失败了。

第二个测试：链式操作

jQuery 的一个强大特性是很方便的链式操作，我们需要 todoify 也支持这个特性，假如一个新的测试：

```

it('should be chainable after invoke', function() {
  var jq = $.fn.todoify();
  expect(jq.jquery).toBeDefined();
});

```

Karma 会报告一个错误：

```

INFO [watcher]: Changed file "/Users/jtqiu/develop/tutorial/tutplus/todo/spec/todoify-spec.js".
Chrome 35.0.1916 (Mac OS X 10.9.3) todoify should be chainable after invoke FAILED
TypeError: Cannot read property 'jquery' of undefined
    at null.<anonymous> (/Users/jtqiu/develop/tutorial/tutplus/todo/spec/todoify-spec.js:8
)
Chrome 35.0.1916 (Mac OS X 10.9.3): Executed 2 of 2 (1 FAILED) (0.019 secs / 0.016 secs)

```

第 8 行报错了，这是因为我们在第一步的实现仅仅是简单的让测试通过而已，并没有实质性的代码，所以需要加上一点真实的功能：

```

$.fn.todoify = function() {
  return this;
}

```

第三个测试：显示一个条目

那么接下来我们就需要做一些“实际”的事情了，首先我们需要插件可以显示一个条目：

```

it('should render an one-item list', function() {
  $('input').todoify({
    data: ['one-item'],
    to: '#todo-container'
  });

  expect($('#todo-container').find('.todo').length).toBe(1);
  expect($('#todo-container').find('.todo').text()).toBe('one-
item');
});

```

给定一个 input 元素，传入一个数组数据，并且传入一个 HTML 元素作为容器，我们预期这个元素中会多出一个 class 为 todo 的元素，并且这个元素中的文本正是传入的数据 one-item。

此时运行测试，会看到报告的错误，我们进一步加入实现代码：

```
$.fn.todoify = function(options){
  var settings = options || {};
  var todo = $('<span></span>').addClass('todo');

  todo.text(options.data[0]);
  $(options.to).append(todo);

  return this;
}
```

但是即使代码上看不出问题，测试仍然是失败的。追踪原因之后，我们会发现，测试中的 input 不知道从何而来。

理论上，这个 input 是我们在实际页面上调用的时候传入的页面上的 input 元素，那么在测试中，我们又如何得知这个元素的名字呢？这里我们需要使用在测试的页面上预先加入一些用于测试的元素，这些用于测试目的的数据被称为 fixture。

这里我们需要另外一个 JavaScript 库：Jasmine-jquery。使用 Jasmine-jquery 可以很容易的加载外部文件作为 fixture，以方便我们的单元测试。我们可以通过 bower 来安装这个库：

```
$ bower install jasmine-jquery#1.6.0
```

另外在 karma.conf.js 中加入对 jasmine-jquery 的引用：

```
files: [
  "src/vendor/jquery/jquery.js",
  "src/vendor/underscore/underscore.js",
  'src/vendor/jasmine-jquery/lib/jasmine-jquery.js',
  "src/todoify.js",
  "spec/**/*.spec.js",
  {pattern: 'spec/fixtures/*.html', included: false, served: true}
],
```

注意此处，我们加入了 pattern 这样一行，这一行告诉 Karma，加载 spec/fixtures 目录下 HTML 文件作为 fixture。

而在 spec/fixtures 目录下，我们会创建一个 todo.html，内容为：

```
<div>
  <input type="text" name="" id="" value="" />
  <div id="todo-container" />
</div>
```

这样，Jasmine-jquery 就会先将这个片段插入到测试页面上，然后测试就可以找到 input 元素，并将其变成一个 todo。这时，我们需要在测试中加入加载 HTML 片段的代码：

```
beforeEach(function(){
  var fixtures = jasmine.getFixtures();
```

```
jasmine.getFixtures().fixturesPath = 'base/spec/fixtures/';
fixtures.load('todo.html');
});
```

对应的实现代码也调整为：

```
$.fn.todoify = function(options) {
  var settings = $.extend({
    data: [],
    to: "body"
  }, options);

  var todo = $('<span></span>').addClass('todo');

  todo.text(settings.data[0]);
  $(settings.to).append(todo);

  return this;
}
```

第四个测试：显示多个条目

对于多个条目，测试非常类似：

```
it('should render multiple items', function() {
  $('input').todoify({
    data: ['one-item', 'two-item', 'three-item'],
    to: '#todo-container'
  });

  expect($('#todo-container').find('.todo').length).toBe(3);
});
```

此时，测试会抱怨：

```
INFO [watcher]: Changed file "/Users/jtqiu/develop/tutorial/tutplus/todo/spec/todoify-spec.js".
Chrome 35.0.1916 (Mac OS X 10.9.3) todoify with data should render multiple items FAILED
  Expected 1 to be 3.
  Error: Expected 1 to be 3.
    at null.<anonymous> (/Users/jtqiu/develop/tutorial/tutplus/todo/spec/todoify-spec.js:36:3)
Chrome 35.0.1916 (Mac OS X 10.9.3): Executed 4 of 4 (1 FAILED) (0.029 secs / 0.024 secs)
```

因此我们需要调整实现代码：

```
$.fn.todoify = function(options) {
  var settings = $.extend({
    data: [],
    to: "body"
  }, options);

  var render = function(item) {
    var todo = $('<span></span>').addClass('todo');
    todo.text(item);
    $(settings.to).append(todo);
  };
}
```

```

    settings.data.forEach(render);

    return this;
};

```

这时候，测试代码显得有些凌乱了，我们需要为各个测试用例分组，所有关于初始化的测试归位一组：

```

describe("initialize", function() {
    it('should have been defined', function() {
        expect($.fn.todoify).toBeDefined();
    });

    it('should be chainable after invoke', function() {
        var jq = $.fn.todoify();
        expect(jq.jquery).toBeDefined();
    });
});

```

而对应的，关于数据渲染的归位一组：

```

describe("with static data", function() {
    beforeEach(function() {
        var fixtures = jasmine.getFixtures();

        jasmine.getFixtures().fixturesPath = 'base/spec/fixtures/';
        fixtures.load('todo.html');
    });

    it('should render an one-item list', function() {
        $('input').todoify({
            data: ['one-item'],
            to: '#todo-container'
        });

        expect($('#todo-container').find('.todo').length).toBe(1);
        expect($('#todo-container').find('.todo').text()).toBe('one-
item');
    });

    it('should render multiple items', function() {
        $('input').todoify({
            data: ['one-item', 'two-item', 'three-item'],
            to: '#todo-container'
        });

        expect($('#todo-container').find('.todo').length).toBe(3);
    });
});

```

第五个测试：添加条目

我们刚刚完成了一个：测试失败-测试通过-代码重构的完整周期，相信读者已经对测试驱动开发的方式有一些认识了，所以很容易就可以想到如何编写添加条目的测试：

```

it('should be able to add new item to empty list', function() {

```

```

    $('input').todoify({
      data: [],
      to: '#todo-container'
    });

    expect($('#todo-container').find('.todo').length).toBe(0);
    $('input').val('new item').pressEnter();

    expect($('#todo-container').find('.todo').length).toBe(1);
    expect($('#todo-container').find('.todo').text()).toBe('new
item');
  });
};

```

假设我们有一个空的列表，当在输入框中添加了新项目之后，列表应该会增加一个条目，并且条目的内容恰好为填入的内容“new item”。

注意此处的 `pressEnter`，这个函数模拟了用户按下回车键的动作，其实现为：

```

$.fn.pressEnter = function () {
  var e = $.Event("keypress");
  e.keyCode = 13;
  $(this).trigger(e);
};

```

同样，测试会报错，我们加入实现代码使得测试通过：

```

var eventHandler = function(event) {
  if(event.keyCode === 13) {
    var item = $(this).val();
    render(item);
    $(this).val('').focus();
  }
};

$(this).keypress(eventHandler);

```

第六个测试：添加额外的条目

这个测试事实上是上一个用例的扩展，如果我们在一个已有的列表中插入新的条目，应该不会影响已有的条目：

```

it('should be able to add new item to list has items', function() {
  $('input').todoify({
    data: ['one-item'],
    to: '#todo-container'
  });

  expect($('#todo-container').find('.todo').length).toBe(1);
  $('input').val('new item').pressEnter();

  expect($('#todo-container').find('.todo').length).toBe(2);
});

```

同样，这个测试与已有的添加单独条目到空表是有一定关系的，因此可以归位一组：

```
describe('manipulate todos', function() {
  it('should be able to add new item to empty list', function() {
    //...
  });

  it('should be able to add new item to list has items',
function() {
  //...
});
});
```

进一步改进

读者可以进一步练习测试驱动开发的方式来完成：

1. 删除一个条目
2. 用户自定义模板（目前是一个简单的 `span` 元素，如果用户需要更复杂的呢？）
3. 与后台服务通信

定制化UI之后的Todo

remove

我的Style和默认的有很大不同的

remove

目前，`todoify` 还没有与后台进行任何的通信，如果可以和后台的 `RESTful` 的 `API` 集成的话，这个插件将会有更多的使用场景。

简单来讲，只需要为插件提供更多选项，并提供回调函数即可，比如：

```
$("#input").todoify({
  resource: 'http://app/todos',
  onadd: function(item) {
    //...
  },
  ondelete: function(item) {
    //...
  }
});
```

然后加入对应的 `ajax` 调用即可。就目前来说，这个例子已经足够。

七 可以测试的 JavaScript 代码

在 Web 技术正处于井喷状态的今天，各种新的库，新的框架，新的工具层出不穷。虽然如此，但是对于一些小型的项目而言，可能并不一定要启用“重量级”的框架。

一直以来，对于小型或者微型的项目，开发人员倾向于使用 jQuery 来完成 DOM 的操作，事件绑定，发送网络请求等操作来完成页面的控制。大量的 jQuery 插件被开发出来，人们编写 JavaScript 的方式完全改变。但是随着代码量的增加，前端代码的复杂性的提升，一个明显的问题被人们重视起来：如何测试这些 JavaScript 代码。

我们这里将讨论一个具体的例子，先查看传统的方式如何实现。随后变换一个视角，从如何更方便测试的角度去重构代码，从而编写出更加清晰易读，更加容易维护的代码。

一个实例

现在有一个应用，系统中录入了很多的地点信息，用户可以搜索自己感兴趣的地点。用户还可以标记搜索结果中的地点。编辑为“喜欢”的地点会显示在右边栏中。



The screenshot shows a web browser at localhost:9292/#. Below the browser is a search interface. At the top, there's a search bar containing the text 'Mel' and a blue 'search' button. Below this, the results are divided into two columns. The left column, titled 'Search results:', contains a list of four items: 'Melbourne 3121', 'Melbourne 3000', 'East Melbourne', and 'West Melbourne'. Each item is in a light gray box with a 'Like' button to its right. The right column, titled 'Places I liked:', contains a list of two items: 'Melbourne 3000' and 'Melbourne 3121'.

在代码的编写过程中，我们暂时不需要考虑用户注册，用户登录，保存数据到数据库等实际的功能。而把目光仅仅关注在前端的这个简单界面上。

基本的功能可以分解为：

1. 用户填写地点名称，点击查找按钮时，发送请求到后台，并获得匹配的数据
2. 根据这些数据，动态的生成新的条目
3. 为每个新条目注册事件，当点击这些条目时，将条目添加到页面的另一个位置

如果页面的结构是这样的话：

```
<div id="container">
  <div id="searchForm">
    <input type="text" id="locationInput" value=""
placeholder="location for search"/>
    <input type="button" class="submit" id="searchButton"
value="search" />
  </div>
</div>
```

```

</div>

<div id="searchResults">
  <h4>Search results:</h4>
  <ul />
</div>

<div id="likedPlaces">
  <h4>Places I liked:</h4>
  <ul />
</div>
</div>

```

那么一个对应的实现很自然的与前面列出的几点相匹配。首先，选中几个主要的元素：

```

var loc = $("#locationInput");
var searchResults = $("#searchResults ul");
var liked = $("#likedPlaces ul");

```

然后为提交按钮注册事件：

```

$("#searchButton").on("click", function() {
  var location = $.trim(loc.val());

  $.ajax({
    url: '/locations/'+location,
    dataType: 'json',
    success: function(locations) {
      getTemplate('location-
detail.tmpl').then(function(template) {
        var tmpl = _.template(template);
        searchResults.html(tmpl({locations: locations}));
      });
    },
    error: function(xhr, status, error) {
      console.log("err: " + error);
    }
  });
});

```

当用户点击 search 按钮时，这段代码会发送请求到后台，如果成功，则先获取一个名为 location-detail.tmpl 的模板，然后将请求到的数据 locations 与这个模板结合，并将结果设置为 searchResults 元素的内容。

获取模板事实上也是发送一次请求，然后将请求到的文件内容缓存到前端：

```

var cache = {};

function getTemplate(template) {
  if(!cache[template]) {
    cache[template] = $.get("templates/" + template);
  }

  return cache[template];
}

```

当第二次点击 search 按钮的时候，就无需再发送一次请求，而直接从本地的 cache 对象中获取。

这里的模板 location-detail.tmpl 的内容为：

```
<% _.each(locations, function(location) { %>
  <li>
    <div class="title">
      <span>
        <%= location.name %>
      </span>
      <span>
        <a href="#" class="like">Like</a>
      </span>
    </div>
  </li>
<% }) %>
```

这是一段 underscore 的模板，内嵌的_.each 语句会遍历 locations 数组变量，然后将每一个元素的 name 字段取出来，添加到 title 下的 span 中作为输出。

在完成了对 searchResults 内容的填充之后，还需要绑定事件，这样当用户点击条目时，可以将条目标记为“喜欢”：

```
searchResults.on('click', '.like', function() {
  var loc = $(this).closest('.title').find('span:nth(0)').text();
  $('<li></li>', {text: loc}).appendTo(liked);
});
```

整体上来说，这段代码还算清晰，也可以很容易的读懂代码的意图。那么如何测试这段代码呢？

```
it("should return locations based on the search critria",
  function() {
    $("#locationInput").val("Melbourne");
    $("#searchButton").click();

    waitsFor(function() {
      return $("#searchResults ul li").length > 0;
    }, 2000);

    runs(function() {
      expect($("#searchResults ul li").length).toEqual(4);
      expect($("#likedPlaces ul li").length).toEqual(0);
    });
  });
```

假设后台会返回 4 个与 Melbourne 这个地方相关的数据，那么这个测试是有意义的，但是这个测试明显已经是一个端到端的测试了。除此之外，我们应该还可以看到这样几个明显的不好的地方：

1. 所有的逻辑都写在了\$(document).ready 中
2. DOM 操作和逻辑混合在一起
3. 如果有进一步的扩展，比如在页面的底部也放一个 search 按钮，代码不能重用

4. 无法完成单元测试

重构：更容易测试的代码

我们再来看看这个页面：

The screenshot shows a web application running on localhost:9292/#. It features a search bar with the text 'Mel' and a 'search' button. Below the search bar, there are two sections: 'Search results:' and 'Places I liked:'. The 'Search results:' section contains a table with four rows: 'Melbourne 3121', 'Melbourne 3000', 'East Melbourne', and 'West Melbourne', each with a 'Like' button. The 'Places I liked:' section contains a list of two items: 'Melbourne 3000' and 'Melbourne 3121'.

如果要划分组建的话，很自然的可以划分为三个：

1. 搜索框（search box）
2. 结果集（search results）
3. 喜欢的地方（liked）

The screenshot is annotated with red boxes and labels to identify components. A red box labeled 'Search Form' encompasses the search bar and the 'search' button. Another red box labeled 'Search Result' encompasses the 'Search results:' section. A third red box labeled 'Liked List' encompasses the 'Places I liked:' section.

搜索框

搜索框组件可以由一个输入框和一个按钮来生成，当点击按钮时，该组件会发起一个请求。如果再进一步，点击按钮时，会触发一个事件，而事件的响应则交给具体的监听器来完成。

那么搜索框的实现就非常简单了：

```

var SearchForm = function(form) {
  this.$element = $(form);
  this.$element.on('click', '.submit', _.bind(this._bindSubmit,
this));
};

SearchForm.prototype._bindSubmit = function(e) {
  var text = this.$element.find('input[type="text"]').val();
  $(document).trigger('search', [text]);
};

```

而且重要的是，我们可以很容易的编写对这个组件的单元测试：

```

describe("search form", function() {
  var form =
    $("<div><input type='text' /><input
class='submit' /></div>");
  var searchForm;

  beforeEach(function() {
    searchForm = new SearchForm(form);
  });

  it("should construct a new form", function() {
    expect(searchForm).toBeDefined();
  });

  it("should trigger search when I click submit", function() {
    var spyEvent = spyOnEvent(document, 'search');
    form.find('.submit').click();

    expect(spyEvent).toHaveBeenTriggered();
  });
});

```

当在“页面”上点击 submit 时，我们可以看到它触发了一个 search 的全局事件。至于谁来捕获这个事件，则可以交由下一个组件来处理。

发送请求

我们完全可以将发送请求这个动作独立出来，作为一个组件来进行测试。

```

it("should create new search", function() {
  expect(search).toBeDefined();
});

it("should fetch data from remote", function() {
  var r = search.fetch("Melbourne");
  expect($.ajax).toHaveBeenCalled();
  expect($.ajax.mostRecentCall.args[0]).toContain("Melbourne");
});

```

在第二个用例中，我们预期 jQuery 的 ajax 方法被调用了，并且当 fetch 的参数为 Melbourne 的时候，我们对 ajax 的调用也包含了该字符串。

当然，我们不需要真实的调用后台的服务，只需要做一个简单的 spy 即可：

```

var search;

beforeEach(function() {
    spyOn($, 'ajax').andCallFake(function(e) {
        return {
            then: function() {}
        }
    });

    search = new Search();
});

```

即当调用\$.ajax 的时候，jasmine 实际上会调用这个假的函数，这个函数会返回一个对象，对象中有一个名为 then 的空方法。

实现中，我们使用了 jQuery 的 promise 对象，使得代码更加简洁，这也是上边的测试中为何会出现 then 的原因。不过我在这里并不打算深入讨论 promise 异步模型，我们会留在后边的章节来讲解。

```

var Search = function() {};

Search.prototype.fetch = function(query) {
    var dfd;

    if(!query) {
        dfd = $.Deferred();
        dfd.resolve([]);
        return dfd.promise();
    }

    return $.ajax('/locations/' + query, {
        dataType: 'json'
    }).then(function(resp) {
        return resp;
    });
};

```

完成了搜索之后，我们来看看结果集。

结果集

结果集作为一个独立的组件，可以被设置值。既然是结果集，它自然接受一个数组作为参数，并将数组包装成可以 DOM 元素。另外，它还需要响应事件，当用户点击列表中的任意一个元素时，会触发一个全局的事件。

我们可以先从单元测试来看结果集组件需要哪些接口，首先是设置：

```

var ul;
var searchResults;

var results = [
    {name: "Richmond"},
    {name: "Melbourne"},
    {name: "Dockland"}
];

beforeEach(function() {

```

```

    ul = $("

</ul>");
    searchResults = new SearchResults(ul);
  });

```

下面的几个用例清楚的体现了结果集组件的对外接口：

```

it("should constructor a new search result", function() {
  expect(searchResults).toBeDefined();
});

it("should set search result", function() {
  searchResults.setResults(results);

  waitsFor(function() {
    return ul.find("li").length > 0;
  }, 1000);

  runs(function() {
    expect(ul.find("li").length).toBe(3);
    var location =
$.trim(ul.find("li").eq(0).find(".title").text());
    expect(location).toContain("Richmond");
  });
});

it("should like one of the search results", function() {
  searchResults.setResults(results);

  var spyEvent = spyOnEvent(document, 'like');

  waitsFor(function() {
    return ul.find("li").length > 0;
  }, 1000);

  runs(function() {
    ul.find('li').first().find('.like').click();
    expect(spyEvent).toHaveBeenTriggered();
  });
});

```

注意，我们现在可以在任何时刻设置结果集：setResults([])。这里的结果可能来源于一个静态数组，或者来源于网络上的一个 JSON 片段，可以是任意的数据源！结果集组件和其数据源完全解耦合了。

结果集组件的实现也变得非常高内聚：

```

var SearchResults = function(element) {
  this.$element = $(element);
  this.$element.on('click', '.like', _.bind(this._bindClick,
this));
};

SearchResults.prototype._bindClick = function(e) {
  var name = $(e.target).closest('.title').find('h4').text();
  $(document).trigger('like', [name]);
};

SearchResults.prototype.setResults = function(locations) {
  var template = $.get('templates/location-detail.tmpl');
  var that = this;

```

```

    template.then(function(tmpl) {
      var html = _.template(tmpl, {locations: locations});
      that.$element.html(html);
    });
  });
};

```

可以看到，加载模板的代码被放在了 `SearchResults` 内部。外界不再，也无需知道其内部的实现机制。

喜欢的地方

同样，我们可以从测试代码来入手：

```

describe("like list", function() {
  var ul;
  var like;

  beforeEach(function() {
    ul = $("<ul></ul>");
    like = new Like(ul);
  });

  it("should constructor a new list", function() {
    expect(like).toBeDefined();
  });

  it("should add new item", function() {
    like.add("juntao");
    expect(ul.find("li").length).toBe(1);
    expect(ul.find("li").eq(0).text()).toEqual("juntao");
  });
});

```

它有一个 `add` 的接口，可以供外部调用，将元素添加到自身上。在实现上，它将一个 `ul` 包装起来，然后当 `add` 被调用时，包装一个 `li` 元素，并添加在自身的 `ul` 上。

```

var Like = function(element) {
  this.$element = $(element);
  return this;
};

Like.prototype.add = function(item) {
  var element = $("<li></li>", {text: item}).addClass('like');
  element.appendTo(this.$element);
};

```

这样，当我们需要为 `like` 注册事件时，就完全不需要修改别的文件，影响范围也非常的小。事实上，只要对外的接口：`add` 方法不变，`Like` 可以被实现成任意的形式。

放在一起

如果我们将所有的组件放在一起，会得到这样的一段代码：

```

$(function() {
  var searchForm = new SearchForm("#searchForm");
  var searchResults = new SearchResults("#searchResults ul");

```



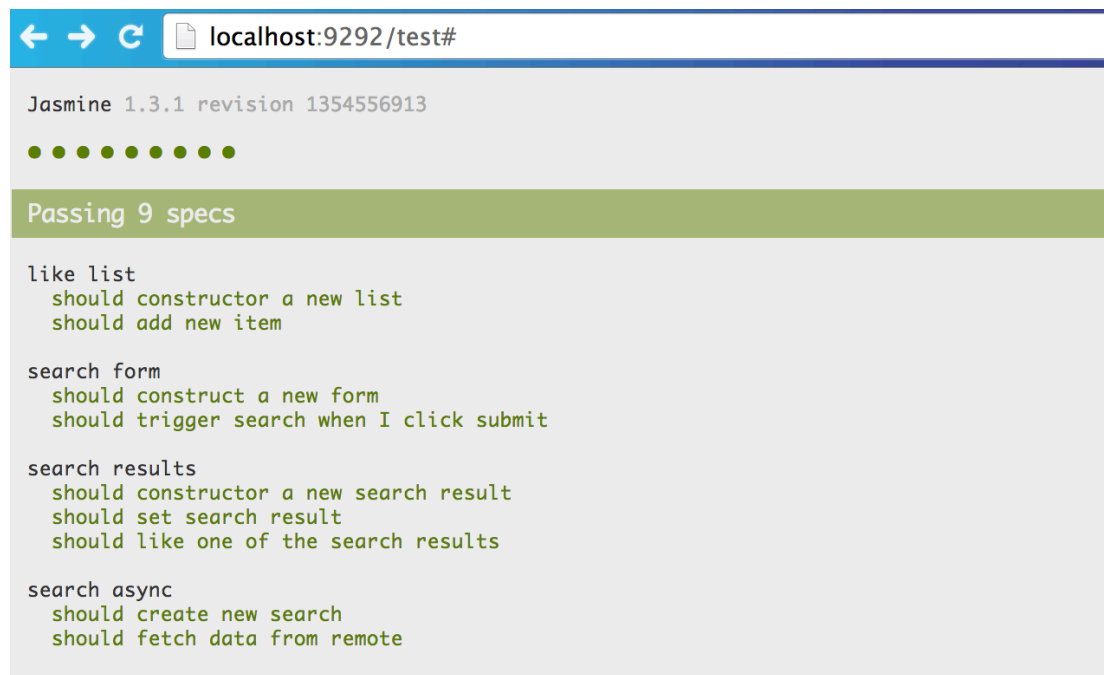
```
var liked = new Like("#likedPlaces ul");
var search = new Search();

$(document).on('search', function(event, query) {
    search.fetch(query).then(function(locations) {
        searchResults.setResults(locations);
    });
});

$(document).on('like', function(e, name) {
    liked.add(name);
});
});
```

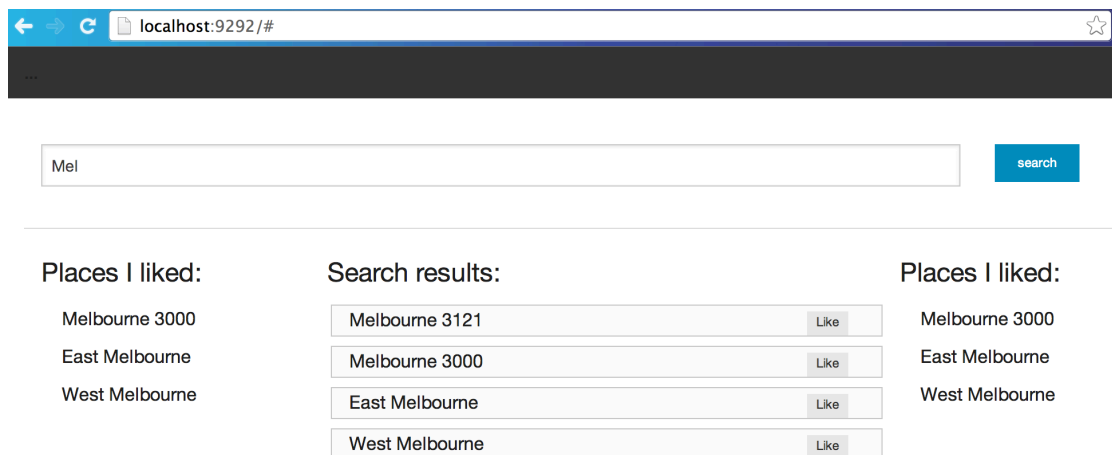
代码当然比刚开始的时候更加清晰了，每个部分都是独立的组件，互相之间并不会直接依赖。但是，你可能已经发现了，代码量反而增多了！我们由 1 个文件变成了 5 个文件，而且有了众多的测试代码。

但是这一切都是值得的，当你需要重用某个组件的时候，拿去用就是了！而且有了测试的保证，每个组件都可以做到更加健壮，更加灵活。



在最后，我们获得了 9 个测试，4 个组件。这些组件完全可以灵活的被组装起来。

由于有了基本的组件，新的实现可以更快速的响应需求的变化。比如，我们需要在界面的左边添加另外一个“喜欢的地方”的组件：



只需要调整对应的 DOM 元素：

```
<div>
  <div id="likedPlaces1">
    <h4>Places I liked:</h4>
    <ul />
  </div>

  <div id="searchResults">
    <h4>Search results:</h4>
    <ul />
  </div>

  <div id="likedPlaces2">
    <h4>Places I liked:</h4>
    <ul />
  </div>
</div>
```

然后在 JavaScript 中加入：

```
var liked1 = new Like("#likedPlaces1 ul");
var liked2 = new Like("#likedPlaces2 ul");

//...

$(document).on('like', function(e, name) {
  liked1.add(name);
  liked2.add(name);
});
```

即可完成新的需求。

关注点分离：另一种实现方式

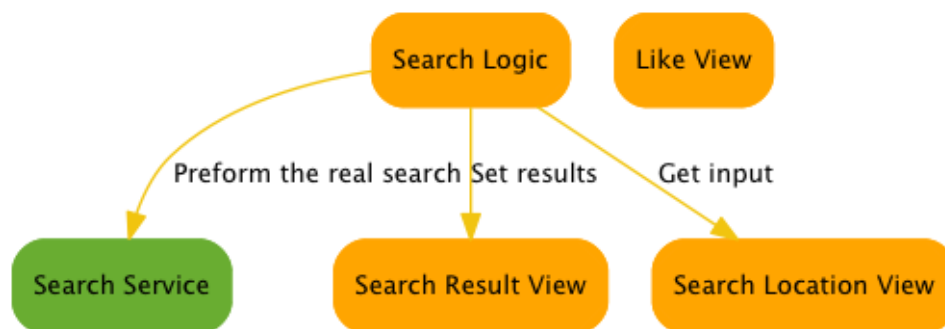
用面向对象的方式将界面元素抽象为独立的组件，使得每一个组件都可以被独立测试，并完成自我管理，这种方式当然比将所有内容混在一起要好得多。但是我们还可以从另外的角度出发，继续改善既有代码。

关注点分离是一种常用的系统解耦的方式，即将不同职责的代码归类起来。既然我们无法避免对 DOM 元素的操作，那么就将这些与 DOM 相关的操作封装到表示视图（view）的抽象中。

在应用程序中，我们事实上非常关心的并不是对页面元素的操作，比如我们的例子中，我们关注的是：当用户输入关键字，点击搜索，得到结果集这个过程。这个过程也正是这个应用程序存在的核心意义。这个过程我们可以单独抽取出来，形成一个对象。

最后，我们需要依赖后端的服务程序，这个服务程序是独立存在的。搜索服务本身只被依赖，它可以给任何需要的地方提供搜索服务。

一个简单的示意图应该是这样的：



SearchLogic 将这些不同的部分组装起来，而这些组件之间可能并不知道对方的存在。

搜索服务

搜索服务应该是最简单，而且最独立的模块，只需要指定一个搜索的端点（数据的提供者），然后暴露一个 search 的接口即可：

```
function SearchService(url) {
  this.search = function(location, successcb, errorcb) {
    $.ajax({
      url: url + location,
      dataType: 'json',
      success: successcb,
      error: errorcb
    });
  };
}
```

由于一个搜索服务是一个异步的调用，我们将搜索成功以及搜索失败的函数作为参数传入，这样即便于测试，也可以获得更大的灵活性。

结果视图

结果视图作为一个视图，仅仅需要正确的渲染自身即可。另外，当发生意外错误时，结果视图需要显示一个错误信息。

```
function SearchResultView(container) {
  this.render = function(data) {
    $(container).html('');
  };
}
```

```

$(data).each(function(index, loc) {
    var li = $("- </li>").html(loc.name);
    $(container).append(li);
    li.on('click', function(e) {
        var loc = $(e.target).text();
        $(document).trigger('like', [loc]);
    });
});

this.renderError = function() {
    $(container).text("something went wrong");
};
}

```

应该注意的是，我们将所有 DOM 相关的操作封装在此处，而至于何时展现，则应该剥离到别的对象中。

render 方法可以遍历传入的 data，然后创建新的 li 元素，绑定事件。注意此处，我们只是简单的自定义了一个事件，并在点击时将这个事件发起到 document 上。

搜索框视图

搜索框视图的实现非常简单，它需要一个获取框中内容的接口，这样调用者可以在任何时刻调用这个接口来获得搜索条件。另外，它需要公开一个接口，方便别人注册在其上，当点击搜索按钮时，调用这个注册过的回调函数。

```

function SearchLocationView() {
    this.getLocation = function() {
        return $("#location").val();
    };

    this.addSearchHandler = function(callback) {
        $("#search").on('click', callback);
    };
}

```

搜索逻辑

有了这些简单逻辑之后，我们的应用的核心部分的代码就会变成：

```

function SearchLocationLogic(formView, resultView, service) {
    this.launch = function() {
        formView.addSearchHandler(this.updateSearchResults);
    };

    this.updateSearchResults = function() {
        var location = formView.getLocation();
        if(location) {
            service.search(location, resultView.render,
resultView.renderError);
        }
    };
}

```

当逻辑部分启动时，它会为搜索框注册一个回调函数，当点击搜索框的搜索按钮时，这个回调会被调用。将这个函数定义为一个命名函数（而不是一个匿名函数）的好处是，我们可以在不触发点击事件的情况下测试这个代码块。

这个函数会先从搜索框视图中获取关键字，然后发起一次对搜索服务的调用，调用的回调则分别指向结果视图的 `render` 和 `renderError`。

放在一起

这时候，应用程序的入口将会变为：

```
$(function() {
    var searchResults = new SearchResultView("#searchResults ul");
    var searchLocation = new SearchLocationView();

    var searchService = new
SearchService("http://localhost:9292/locations/");
    var searchLogic = new SearchLocationLogic(searchLocation,
searchResults, searchService);
    searchLogic.launch();

    var liked = new LikeView("#liked ul");
    $(document).on('like', function(e, loc) {
        liked.render(loc);
    });
});
```

此处的 `LikeView` 是一个更加简单的独立视图：

```
function LikeView(container) {
    this.render = function(data) {
        var li = $("- </li>").text(data);
        $(container).append(li);
    };
}

```

越小巧，犯错的可能也越小，而且一旦出现错误，也可以很容易定位并修复。

更容易测试的代码

由于视图的分离，应用程序的核心逻辑被包装到了搜索逻辑部分，如果我们可以保证这部分代码的质量，视图部分事实上是无需测试的（视图已经被简化为了简单的值-对象，类似于 Java 中的 POJO）

对于逻辑部分的测试，我们需要创建一些 mock 对象：

```
var formView;
var searchResultView;
var searchService;

beforeEach(function() {
    formView = jasmine.createSpyObj('SearchLocationView',
['getLocation']);
```

```

    searchResultView = jasmine.createSpyObj('SearchResultView',
[ 'render', 'renderError' ]);
    searchService = jasmine.createSpyObj('SearchService',
[ 'search' ]);
  });

```

然后我们只需要验证各个组件间的交互是正确的即可：

```

it("do search logic", function() {
  var logic = new SearchLocationLogic(formView, searchResultView,
searchService);
  logic.updateSearchResults();

  expect(formView.getLocation).toHaveBeenCalled();
});

```

即当调用 updateSearchResults 时，需要保证搜索框视图的 getLocation 被调用了。另外一个测试场景是：

```

it("search for something", function() {
  formView = jasmine.createSpy('SearchLocationView');
  formView.getLocation =
jasmine.createSpy('getLocation').andCallFake(function() {
    return "Melbourne";
  });

  var logic = new SearchLocationLogic(formView, searchResultView,
searchService);
  logic.updateSearchResults();

  expect(searchService.search).toHaveBeenCalled();
  expect(searchService.search.mostRecentCall.args[0])
    .toEqual("Melbourne");
  expect(searchService.search.mostRecentCall.args[1])
    .toEqual(searchResultView.render);
  expect(searchService.search.mostRecentCall.args[2])
    .toEqual(searchResultView.renderError);
});

```

即确保调用 updateSearchResults 时，传递的参数是正确的。

而测试搜索服务这种独立的模块则更加容易：

```

describe("search service", function() {
  it("call ajax underline", function() {
    var spy = spyOn($, 'ajax');
    var service = new
SearchService("http://whatsoever.service");

    service.search("terms");
    expect($.ajax).toHaveBeenCalled();
  });
});

```

应该注意的是，我们此处无需测试任何的视图代码。视图本身对 DOM 的增删查改无需特别测试，而关于事件的触发等，我们可以移至更高层级的测试中，比如基于 Selenium 的测试。