JChehe / blog Public

Code Issues 46 Pull requests Actions Projects Security Insights

master

blog / translation / 用Mocha和Chai对JavaScript进行单元测试.md

Jianchao Liu fix: 七牛图床迁移到腾讯云

1 contributor

用Mocha和Chai对JavaScript进行单元测试

标签： BDD chai mocha tdd 单元测试

本文由 伯乐在线 - 刘健超-J.c 翻译，等待校稿。未经许可，禁止转载！

英文出处：Unit Test Your JavaScript Using Mocha and Chai。欢迎加入翻译组。

你曾试过修改代码后，导致其它地方出现问题吗？

也许有大多数人试过。因为这是几乎不可避免的，特别在庞大的代码面前。由于代码间可能是环环相扣的，改变一处会影响另一处。

但如果这种情况不会发生呢？如果你有一种方法能知道改变后会出现的结果呢？这无疑是极好的。因为修改代码后无需担心会破坏什么东西，从而程序出现 bug 的概率更低，在 debug 上花费时间更少。

这就是单元测试的魅力。它能自动检测代码中的任何问题。在修改代码后进行相应测试，若有问题，能立刻知道问题是什么，问题在哪和正确的做法是什么。这完全消除任何猜测！

在本文，我会让你了解如何对 JavaScript 代码进行单元测试。而且，在本文出现的案例和技术可同时应用到基于浏览器的代码和 Node.js 的代码。

教程中的代码也可到我的 GitHub repo 中得到。

什么是单元测试

当你对代码库进行测试时，可先取一段代码（如函数），然后在特定情况下，验证其行为是否正确。而单元测试就是这方面的一种结构化和自动化的方法。当然，写的测试越多，获得的益处也更大。这也让你进行开发时会更加自信。

单元测试的核心思想是给函数特定的输入，测试其行为。也就是说，以特定的参数调用函数，然后检查是否得到正确的结果。

```
// Given 1 and 10 as inputs...
// 输入 1 和 10...
var result = Math.max(1, 10);

// ...we should receive 10 as the output
// ...应该输出 10
if(result !== 10) {
  throw new Error('Failed');
}
```

在实际中，测试有时会更复杂。例如，如果你的函数含有一个 Ajax 请求，那么测试就需要设定更多的东西。当然，“赋予特定的输入，期待得到一个特定的输出”原理仍然适用。

设置工具

在本文，我们选择 Mocha。它入门简单，能同时适用于基于浏览器的测试和 Node.js 的测试，而且与其它测试工具运行良好。

安装 Mocha 的最简单方式是 npm（为此，也需要安装 Node.js）。如果你不懂得如何在你的电脑上安装 npm 或 Node.js，可查看我的教程 A Beginner's Guide to npm — the Node Package Manager。

安装好 Node.js 后，在你的项目目录下打开 terminal 或 command line。

- 如果你想在浏览器上测试代码，执行 `npm install mocha chai --save-dev` 。
- 如果你想测试 Node.js 代码，除了执行上面那行命令，也要执行 `npm install -g mocha` 。

这就安装了 `mocha` 和 `chai` 包（package）了。Mocha 是一个运行测试的库，而 Chai 包含一些有用的功能，我们能利用这些功能对我们的测试结果进行验证。

Node.js vs Browser 测试对比

下面的案例是在浏览器上运行测试的。如果想为你的 Node.js 应用进行单元测试，要遵循以下步骤。

- 对于 Node，无需测试运行文件（test runner file）。
- 为了引入 Chari，需在测试文件顶部添加语句 `var chai = require('chai');` 。
- 用 `mocha` 命令执行单元测试，而不是打开浏览器。

设置目录结构

为了让文件结构更清晰，应将测试文件放在主代码文件的一个独立目录下。这是为了方便以后添加其它类型的测试（如集成测试（integration tests） 和 功能测试（functional tests））。

对于 JavaScript，最流行的实践方案是在项目根目录下创建一个 `test/` 文件夹。然后，将每个测试文件放置在该文件夹下，如 `test/someModuleTest.js` 。另一种方案是，在 `test/` 目录下，再创建文件夹。但我建议尽量保持简单——这样能保证在后面必要时进行（快速）修改。

设置测试运行器（Test Runner）

为了能在浏览器上进行测试，我们需要创建一个简单的 HTML 页面作为测试运行页（test runner page）。该页面会加载 Mocha、测试库文件和实际测试文件。为了运行这些测试，我们只需在浏览器打开运行器（runner）。

如果你使用 Node.js，你可跳过这一步。Node.js 的单元测试能通过命令 `mocha` 运行，前提是按照我推荐的目录结构。

下面是我们用于测试运行器（test runner）的代码。我将其存为 `testrunner.html` 。

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mocha Tests</title>
    <link rel="stylesheet" href="node_modules/mocha/mocha.css">
  </head>
  <body>
    <div id="mocha"></div>
    <script src="node_modules/mocha/mocha.js"></script>
    <script src="node_modules/chai/chai.js"></script>
    <script>mocha.setup('bdd')</script>

    <!-- load code you want to test here -->

    <!-- load your test files here -->

    <script>
      mocha.run();
    </script>
  </body>
</html>
```

该测试运行器的几个重要点：

- 为了让测试结果拥有漂亮的样式，我们加载了 Mocha 的 CSS 文件。
- 创建了一个 ID 为 mochat 的 div 标签。测试结果将放在该标签内。
- 加载 Mocha 和 Chai 脚本文件。由于这两个文件是通过 npm 安装的，它们被放在 node_modules 目录的相应文件夹下。
- 通过调用 mocha.setup，开启 Mocha 的测试功能（testing helpers）。
- 然后，加载需要的测试项和相应测试的文件。尽管我们还没在这放置任何代码。
- 最后，调用了 mocha.run 执行相应测试。当然，要确保在资源和测试文件加载完成后才调用该函数。

基本的测试骨架

现在我们可以运行测试了，下面就开始写点测试相关的东西吧。

首先，创建 `test/arrayTest.js` 。每个文件名都有其具体含义，显然它是个测试文件，并会测试 array 的基本功能。

每个测试案例文件都会遵循以下基本模式。首先，有个 `describe` 块：

```
describe('Array', function() {
  // Further code for tests goes here
});
```

`describe` 用于把单独的测试聚合在一起。其第一个参数用于指示测试什么。在本例中，由于我们打算测试 array 功能，我传入一个 `'Array'` 字符串。

然后，在 `describe` 内需有 `it` 块：

```
describe('Array', function() {
  it('should start empty', function() {
    // Test implementation goes here
  });

  // We can have more its here
});
```

`it` 用于创建实际的测试。其第一个参数是对该测试的描述，且该描述的语言应该是人类可读的（而非编程语言）。如在本例中，“it should empty”能很好地描述了 array 的行为。实现该测试的具体代码则写在 `it` 的第二个参数 function 内。

所有 Mocha 测试都以同样的骨架编写，而且它们遵循这相同且基本的模式。

- 首先，使用 `describe` 表明我们测试什么，如“描述 array 该如何运行”。
- 然后，使用多个 `it` 函数创建独立的测试，每个 `it` 应该描述一个特定的行为，如上述的案例 “it should start empty（array 运行前应为空）”

编写测试代码

现在我们已经知道如何构造测试案例了，下面就开始更有趣的部分——实现测试。

由于我们的测试是 array 初始值应为空，即我们需要创建一个数组并确保它为空。实现该测试是非常简单的：

```
var assert = chai.assert;

describe('Array', function() {
  it('should start empty', function() {
    var arr = [];

    assert.equal(arr.length, 0);
  });
});
```

请注意首行代码，我们设置了 `assert` 变量。这样就不用每次需要它时输入 `chai.assert` 了。

在 `it` 函数里，我们创建了一个数组并检查其长度。尽管简单，但很好地展示了测试是如何工作的。

首先，你有东西需要被测试——这叫 被测系统（System Under Test，SUT）。若有需要，则对被测系统进行相应操作。对于上述案例，由于检查数组初始值是否为空，我们没做任何操作。

测试的最后步骤应该是验证——对结果进行断言（assertion）检查。对于上述案例，我们对此使用 `assert.equal` 。大多数断言函数的参数顺序是一致的：首先是“实际”值，然后是“期待”值。

实际值是测试代码的结果，因此，在该案例中是 `arr.length` 。

期待值是预想的结果。由于数组的初始值应为空，因此，在该案例中的期待值是 `0` 。

虽然 Chai 提供了两种不同的断言（assertion）编写方式，但现在为了保持简单，我们使用了 assert。当你能熟练编写测试时，你可能更想用 expect assertions ，因为它提供了更灵活的操作。

运行测试

为了运行该测试，我们需要将其添加到先前创建的测试运行器文件内。

对于 Node.js，可跳过此步骤，然后使用命令 `mocha` 执行测试。你会在 terminal 里看到测试结果。

向运行器添加该测试（针对浏览器端）：

```
<!-- load your test files here -->
<script src="test/arrayTest.js"></script>
```

你一旦添加了脚本，就可以加载测试运行器页面了（若选择在浏览器进行测试）。

测试结果

当你运行这些测试，其测试结果看起来和下图类似：

注意：在 `describe` 和 `it` 函数的描述语句都在页面展示出来了——测试项（如：should start empty）都分组放在描述（如：Array）下。当然，也可以对 `describe` 块再嵌套，以创建更深的子分组。

下面看看测试失败是显示什么。

将测试的该行代码进行修改：

```
assert.equal(arr.length, 0);
```

将 0 改为 1。这无疑会导致测试失败，因为数组长度不再匹配期待值。

如果你再次运行测试，那么在测试结果中，运行错误的描述将以红色显示。

测试的一项好处是能帮助你更快地找到 bug，尽管错误信息在这并不是非常详细。

大多数断言函数都带有一个可选的 `message` 参数。该信息参数会在断言失败时显示。因此我们可以利用该参数，让错误信息更易被我们所理解。

我们能像下面那样向断言添加 `message` 参数：

```
assert.equal(arr.length, 1, 'Array length was not 0');
```

如果你再次运行测试，那么自定义的信息会取代默认的信息而显示出来。

OK，让我们将 `1` 改回 `0` ，确保测试通过。

综合案例

到目前为止，案例都是相当简单的。那么下面就让我们将学到的知识付诸实践，看看如何测试一段实际当中所用到的代码。

下面是一个将 CSS 类名添加到元素的函数。我们将该函数放进新文件 `js/className.js` 。

```
function addClass(el, newClass) {
  if(el.className.indexOf(newClass) === -1) {
    el.className += newClass;
  }
}
```

当元素的 `className` 属性不含有新类名时，才向元素添加新类名——毕竟谁想看到 `<div class="hello hello hello hello">` 。

在最好的情况下，我们要在编写代码前先为该函数编写测试。但 测试驱动开发（test-driven development） 是一个复杂的主题，因此我们现在仅专注于编写测试。

开始前，让我们重温单元测试的基本思想：赋予函数特定的输入，然后验证函数的行为是否符合预期。所以，该函数的输入和行为是什么呢？

给定一个元素和一个类名：

- 若元素的 `className` 属性未含有该类名，则应添加。
- 若元素的 `className` 属性已含有该类名，则不应添加。

将这两种情况转化为两个测试。在 `test` 目录下，创建新文件 `classNameTest.js` 并添加以下内容：

```
describe('addClass', function() {
  it('should add class to element');
  it('should not add a class which already exists');
});
```

我们也可以将措词稍微地改成“it should do X”，虽然可读性更强一点，但本质上仍然与我们上述语句的可读性一致。根据原来的措词联想到相应的测试也不难。

等等，测试函数跑去哪了？当我们省略 `it` 的第二个参数，Mocha 会在测试结果中标记这些测试为待测试项。这让设置多个测试变得更方便——就像一个备忘录，列着打算编写的测试。

接着实现第一个测试。

```
describe('addClass', function() {
  it('should add class to element', function() {
    var element = { className: '' };

    addClass(element, 'test-class');

    assert.equal(element.className, 'test-class');
  });

  it('should not add a class which already exists');
});
```

在该测试中，我们创建了 `element` 变量，并将其与字符串 `test-class` （作为元素的新类名） 作为参数传入 `addClass` 函数。然后，使用断言检查该类名是否已包含在值（element.className）里。

再一次，我们从初始的想法出发——给定一个元素和一个类名，将类名添加到 class 列表，然后以简单的方式将其转化为代码。

尽管该函数（addClass）是针对 DOM 元素的，但我们在此使用了一个简单JS对象（plain JS object，根据 jQuery 官方定义：含有零个或多个键值对的对象）。是的，有时我们可以利用 JavaScript 的动态特性，以上述方式简化测试。如果不这样做，我们就要创建一个实际的元素，这无疑会使测试代码变复杂。当然，这还有另一个好处，由于没使用 DOM，该测试也能在 Node.js 运行。

在浏览器运行测试

为了在浏览器运行测试，你需要在运行器添加 `className.js` 和 `classNameTest.js` 。

```
<!-- load code you want to test here -->
<script src="js/className.js"></script>

<!-- load your test files here -->
<script src="test/classNameTest.js"></script>
```

正如 CodePen 所显示：一个测试通过，而另一个显示待测试。注意：为了让代码运行在 CodePen 环境下，代码需稍作调整。

<iframe height='268' scrolling='no' src='//codepen.io/SitePoint/embed/XXzXLX/?height=268&theme-id=0&default-tab=result' frameborder='no' allowtransparency='true' allowfullscreen='true' style='width: 100%;'>See the Pen Unit Testing with Mocha (1) by SitePoint (@SitePoint) on CodePen. </iframe>
See the Pen Unit Testing with Mocha (1) by SitePoint (@SitePoint) on CodePen.

<script async src="//assets.codepen.io/assets/embed/ei.js"></script>
接着，实现第二个测试...

```
it('should not add a class which already exists', function() {
  var element = { className: 'exists' };

  addClass(element, 'exists');

  var numClasses = element.className.split(' ').length;
  assert.equal(numClasses, 1);
});
```

经常运行测试是一种好习惯。因此，让我们现在运行测试看看会发生什么。

不出所料，两者均通过。

下面是在 CodePen 实现第二个测试的例子。

<iframe height='268' scrolling='no' src='//codepen.io/SitePoint/embed/pgdyzz/?height=268&theme-id=0&default-tab=result' frameborder='no' allowtransparency='true' allowfullscreen='true' style='width: 100%;'>See the Pen Unit Testing with Mocha (2) by SitePoint (@SitePoint) on CodePen. </iframe>
See the Pen Unit Testing with Mocha (2) by SitePoint (@SitePoint) on CodePen.

<script async src="//assets.codepen.io/assets/embed/ei.js"></script>
但事情没那么简单！该函数的第三种情况我们并没有考虑到，这也是该函数的一个非常严重的 Bug。虽然该函数只有三行代码，但你注意到了吗?

下面为第三种情况编写多一个案例，让这个 Bug 暴露出来。

```
it('should append new class after existing one', function() {
  var element = { className: 'exists' };

  addClass(element, 'new-class');

  var classes = element.className.split(' ');
  assert.equal(classes[1], 'new-class');
});
```

你可在下面的 CodePen 看到，这次测试失败了。导致该问题的原因很简单：元素上的 CSS 类名应以空格隔开。然而，现在实现的 `addClass` 并未加空格！

<iframe height='268' scrolling='no' src='//codepen.io/SitePoint/embed/oboxve/?height=268&theme-id=0&default-tab=result' frameborder='no' allowtransparency='true' allowfullscreen='true' style='width: 100%;'>See the Pen Unit Testing with Mocha (3) by SitePoint (@SitePoint) on CodePen. </iframe>
See the Pen Unit Testing with Mocha (3) by SitePoint (@SitePoint) on CodePen.

<script async src="//assets.codepen.io/assets/embed/ei.js"></script>
修复该函数，让测试通过。

```
function addClass(el, newClass) {
  if(el.className.indexOf(newClass) !== -1) {
    return;
  }

  if(el.className !== '') {
    //ensure class names are separated by a space
    newClass = ' ' + newClass;
  }

  el.className += newClass;
}
```

修复后，最终在 CodePen 测试通过。

<iframe height='268' scrolling='no' src='//codepen.io/SitePoint/embed/BjmKBG/?height=268&theme-id=0&default-tab=result' frameborder='no' allowtransparency='true' allowfullscreen='true' style='width: 100%;'>See the Pen Unit Testing with Mocha (4) by SitePoint (@SitePoint) on CodePen. </iframe>

414 lines (268 sloc) | 22 KB

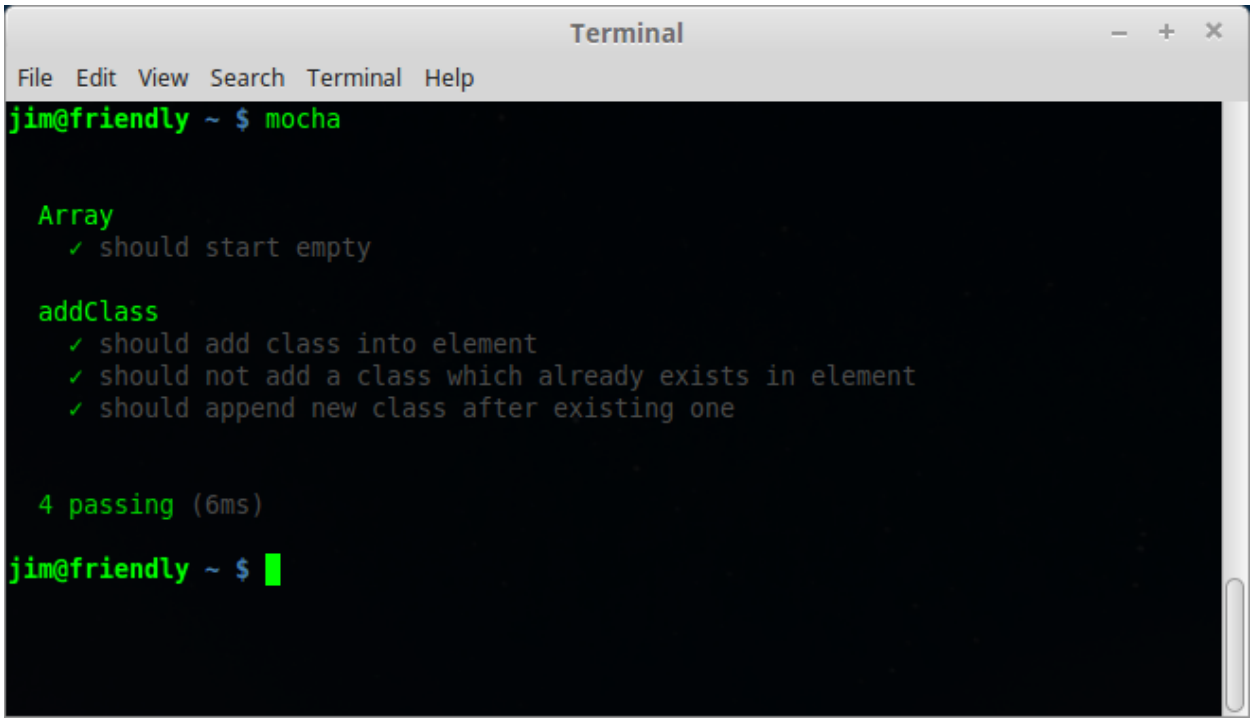
在 Node 运行测试

在 Node，由于 `className.js` 和 `classNameTest.js` 在不同文件下，我们需要一种方式将一个文件导出到另一个文件内。而标准的方式是通过 `module.exports` 。如果你需要复习相关知识，你可以看看 Understanding module.exports and exports in Node.js。

代码本质不变，只是结构稍微不同：

```
// className.js

module.exports = {
  addClass: function(el, newClass) {
    if(el.className.indexOf(newClass) !== -1) {
      return;
    }

    if(el.className !== '') {
      //ensure class names are separated by a space
      newClass = ' ' + newClass;
    }

    el.className += newClass;
  }
}


// classNameTest.js

var chai = require('chai');
var assert = chai.assert;

var className = require('../js/className.js');
var addClass = className.addClass;

// The rest of the file remains the same
// 文件其它部分保持不变
describe('addClass', function() {
  ...
});
```

正如你所看到的，测试通过。



下一步呢?

正如你所看到的，测试并不复杂或困难。与编写 JavaScript 应用的其它方面一样，有一些重复的基本模式。一旦你熟悉这些，你可以一次又一次使用它们。

但这些只是单元测试的皮毛，还有很多相关知识需要学习。

- 测试更复杂的系统
- 如何处理Ajax、数据库和其它“外部”的东西。
- 测试驱动开发

如果你想继续学习更多相关知识，可看看我编写的 免费的 JavaScript 单元测试快速入门系列。如果你觉得本文有用，你更应该点击 这里 看看。