

测试框架 Mocha 实例教程

作者：阮一峰

分享

日期：2015年12月 3日

Mocha（发音"摩卡"）诞生于2011年，是现在最流行的JavaScript测试框架之一，在浏览器和Node环境都可以使用。

所谓"测试框架"，就是运行测试的工具。通过它，可以为JavaScript应用添加测试，从而保证代码的质量。

本文全面介绍如何使用 **Mocha**，让你轻松上手。如果你以前对测试一无所知，本文也可以当作JavaScript单元测试入门。值得说明的是，除了**Mocha**以外，类似的测试框架还有 **Jasmine**、**Karma**、**Tape** 等，也很值得学习。



simple, flexible, fun

一、安装

我为本文写了一个示例库 **Mocha-demos**，请先安装这个库。

```
$ git clone https://github.com/ruanyf/mocha-demos.git
```

如果你的电脑没装Git，可以直接下载[zip压缩包](#)，进行解压。

然后，进入 `mocha-demos` 目录，安装依赖（你的电脑必须有Node）。

```
$ cd mocha-demos
$ npm install
```

上面代码会在目录内部安装 Mocha，为了操作的方便，请在全面环境也安装一下 Mocha。

```
$ npm install --global mocha
```

二、测试脚本的写法

Mocha 的作用是运行测试脚本，首先必须学会写测试脚本。所谓“测试脚本”，就是用来测试源码的脚本。

下面是一个加法模块 `add.js` 的代码。

```
// add.js
function add(x, y) {
  return x + y;
}

module.exports = add;
```

要测试这个加法模块是否正确，就要写测试脚本。

通常，测试脚本与所要测试的源码脚本同名，但是后缀名为 `.test.js`（表示测试）或者 `.spec.js`（表示规格）。比如，`add.js` 的测试脚本名字就是 `add.test.js`。

```
// add.test.js
var add = require('./add.js');
var expect = require('chai').expect;

describe('加法函数的测试', function() {
  it('1 加 1 应该等于 2', function() {
    expect(add(1, 1)).to.be.equal(2);
  });
});
```

上面这段代码，就是测试脚本，它可以独立执行。测试脚本里面应该包括一个或多个 `describe` 块，每个 `describe` 块应该包括一个或多个 `it` 块。

`describe` 块称为"测试套件"（**test suite**），表示一组相关的测试。它是一个函数，第一个参数是测试套件的名称（"加法函数的测试"），第二个参数是一个实际执行的函数。

`it` 块称为"测试用例"（**test case**），表示一个单独的测试，是测试的最小单位。它也是一个函数，第一个参数是测试用例的名称（"1 加 1 应该等于 2"），第二个参数是一个实际执行的函数。

三、断言库的用法

上面的测试脚本里面，有一句断言。

```
expect(add(1, 1)).to.be.equal(2);
```

所谓"断言"，就是判断源码的实际执行结果与预期结果是否一致，如果不一致就抛出一个错误。上面这句断言的意思是，调用 `add(1, 1)`，结果应该等于2。

所有的测试用例（`it`块）都应该含有一句或多句的断言。它是编写测试用例的关键。断言功能由断言库来实现，**Mocha**本身不带断言库，所以必须先引入断言库。

```
var expect = require('chai').expect;
```

断言库有很多种，**Mocha**并不限制使用哪一种。上面代码引入的断言库是 `chai`，并且指定使用它的 `expect` 断言风格。

`expect` 断言的优点是很接近自然语言，下面是一些例子。

```
// 相等或不相等
expect(4 + 5).to.be.equal(9);
expect(4 + 5).to.be.not.equal(10);
expect(foo).to.be.deep.equal({ bar: 'baz' });

// 布尔值为true
```

```
expect('everything').to.be.ok;
expect(false).to.not.be.ok;

// typeof
expect('test').to.be.a('string');
expect({ foo: 'bar' }).to.be.an('object');
expect(foo).to.be.an.instanceof(Foo);

// include
expect([1,2,3]).to.include(2);
expect('foobar').to.contain('foo');
expect({ foo: 'bar', hello: 'universe' }).to.include.keys('foo');

// empty
expect([]).to.be.empty;
expect('').to.be.empty;
expect({}).to.be.empty;

// match
expect('foobar').to.match(/^foo/);
```

基本上，`expect` 断言的写法都是一样的。头部是 `expect` 方法，尾部是断言方法，比如 `equal`、`a` / `an`、`ok`、`match` 等。两者之间使用 `to` 或 `to.be` 连接。

如果 `expect` 断言不成立，就会抛出一个错误。事实上，只要不抛出错误，测试用例就算通过。

```
it('1 加 1 应该等于 2', function() {});
```

上面的这个测试用例，内部没有任何代码，由于没有抛出了错误，所以还是会通过。

四、Mocha的基本用法

有了测试脚本以后，就可以用Mocha运行它。请进入 `demo01` 子目录，执行下面的命令。

```
$ mocha add.test.js
```

加法函数的测试

✓ 1 加 1 应该等于 2

```
1 passing (8ms)
```

上面的运行结果表示，测试脚本通过了测试，一共只有1个测试用例，耗时是8毫秒。

`mocha` 命令后面紧跟测试脚本的路径和文件名，可以指定多个测试脚本。

```
$ mocha file1 file2 file3
```

Mocha默认运行 `test` 子目录里面的测试脚本。所以，一般都会把测试脚本放在 `test` 目录里面，然后执行 `mocha` 就不需要参数了。请进入 `demo02` 子目录，运行下面的命令。

```
$ mocha
```

加法函数的测试

- ✓ 1 加 1 应该等于 2
- ✓ 任何数加0应该等于自身

```
2 passing (9ms)
```

这时可以看到，`test` 子目录里面的测试脚本执行了。但是，你打开 `test` 子目录，会发现下面还有一个 `test/dir` 子目录，里面还有一个测试脚本 `multiply.test.js`，并没有得到执行。原来，Mocha默认只执行 `test` 子目录下面第一层的测试用例，不会执行更下层的用例。

为了改变这种行为，就必须加上 `--recursive` 参数，这时 `test` 子目录下面所有的测试用例----不管在哪一层----都会执行。

```
$ mocha --recursive
```

加法函数的测试

- ✓ 1 加 1 应该等于 2
- ✓ 任何数加0应该等于自身

乘法函数的测试

- ✓ 1 乘 1 应该等于 1

```
3 passing (9ms)
```

五、通配符

命令行指定测试脚本时，可以使用通配符，同时指定多个文件。

```
$ mocha spec/{my,awesome}.js  
$ mocha test/unit/*.js
```

上面的第一行命令，指定执行 `spec` 目录下面的 `my.js` 和 `awesome.js`。第二行命令，指定执行 `test/unit` 目录下面的所有js文件。

除了使用Shell通配符，还可以使用Node通配符。

```
$ mocha 'test/**/*.@(js|jsx)'
```

上面代码指定运行 `test` 目录下面任何子目录中、文件后缀名为 `js` 或 `jsx` 的测试脚本。注意，Node的通配符要放在单引号之中，否则星号（`*`）会先被Shell解释。

上面这行Node通配符，如果改用Shell通配符，要写成下面这样。

```
$ mocha test/{,*/}*. {js,jsx}
```

六、命令行参数

除了前面介绍的 `--recursive`，Mocha还可以加上其他命令行参数。请在 `demo02` 子目录里面，运行下面的命令，查看效果。

6.1 --help, -h

`--help` 或 `-h` 参数，用来查看Mocha的所有命令行参数。

```
$ mocha --help
```

6.2 --reporter, -R

`--reporter` 参数用来指定测试报告的格式，默认是 `spec` 格式。

```
$ mocha
# 等同于
$ mocha --reporter spec
```

除了 `spec` 格式，官方网站还提供了其他许多[报告格式](#)。

```
$ mocha --reporter tap

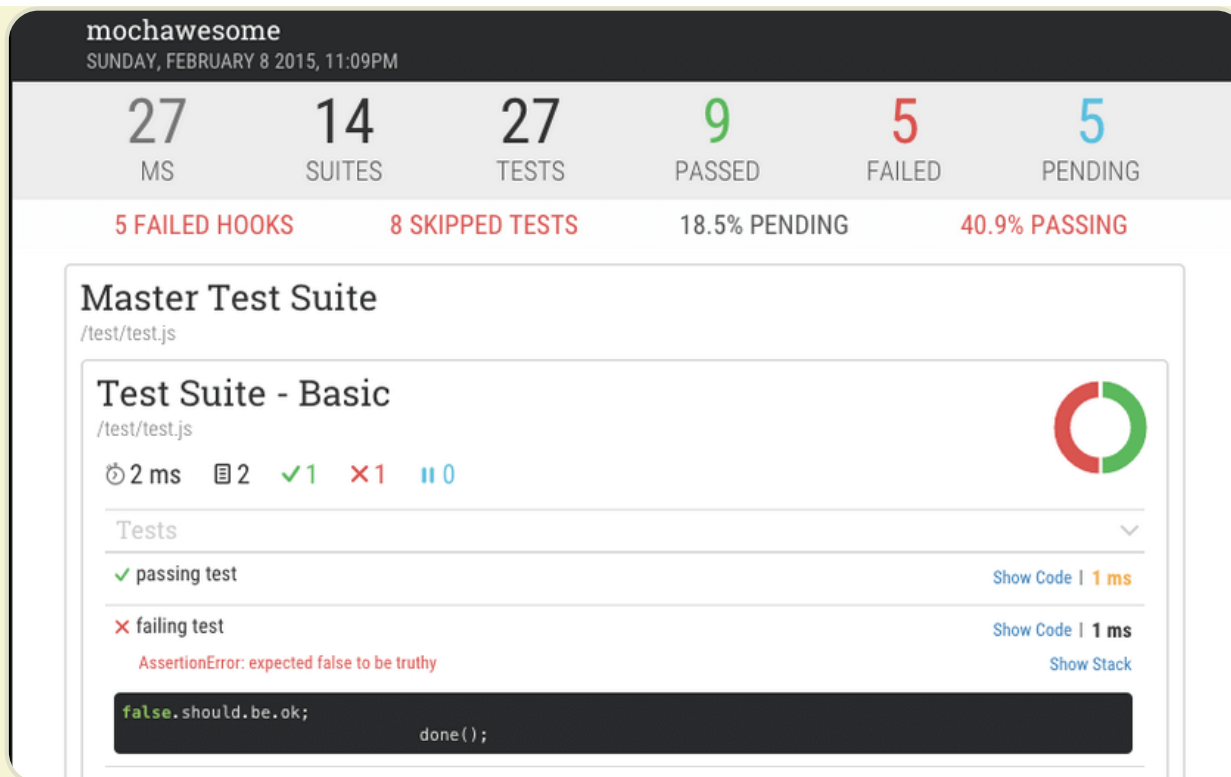
1..2
ok 1 加法函数的测试 1 加 1 应该等于 2
ok 2 加法函数的测试 任何数加0应该等于自身
# tests 2
# pass 2
# fail 0
```

上面是 `tap` 格式报告的显示结果。

`--reporters` 参数可以显示所有内置的报告格式。

```
$ mocha --reporters
```

使用 `mochawesome` 模块，可以生成漂亮的HTML格式的报告。



```
$ npm install --save-dev mochawesome  
$ ../node_modules/.bin/mocha --reporter mochawesome
```

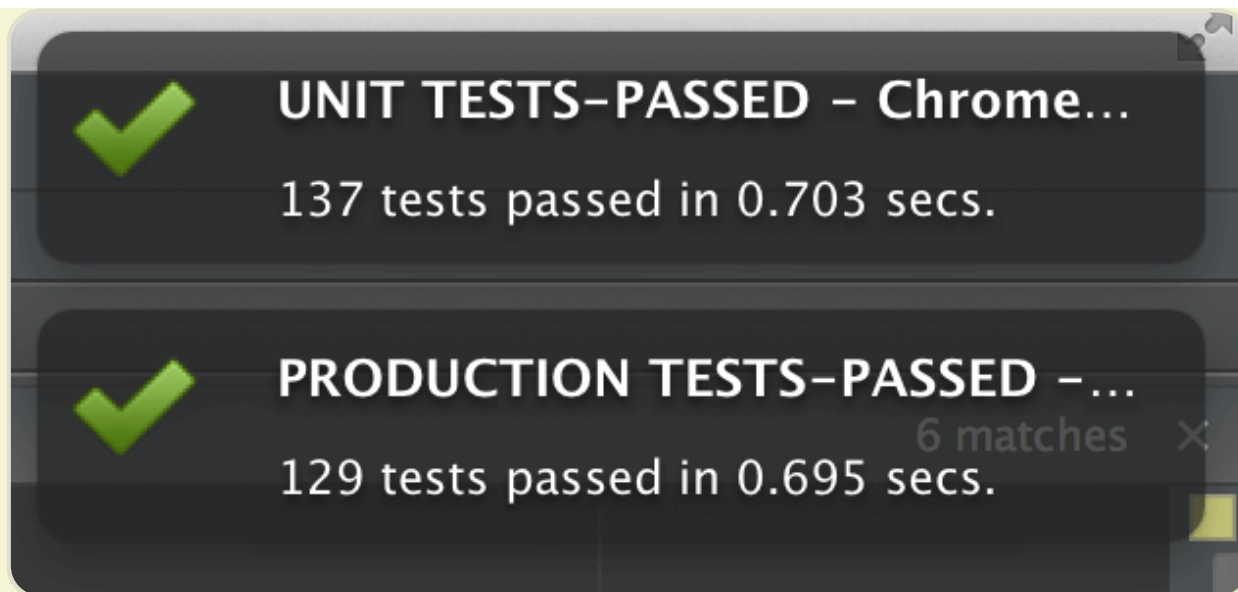
上面代码中，`mocha` 命令使用了项目内安装的版本，而不是全局安装的版本，因为 `mochawesome` 模块是安装在项目内的。

然后，测试结果报告就在 `mochaawesome-reports` 子目录生成。

6.3 --growl, -G

打开 `--growl` 参数，就会将测试结果在桌面显示。

```
$ mocha --growl
```

6.4 --watch, -w

`--watch` 参数用来监视指定的测试脚本。只要测试脚本有变化，就会自动运行 Mocha。

```
$ mocha --watch
```

上面命令执行以后，并不会退出。你可以另外打开一个终端窗口，修改 `test` 目录下面的测试脚本 `add.test.js`，比如删除一个测试用例，一旦保存，Mocha 就会再次自动运行。

6.5 --bail, -b

`--bail` 参数指定只要有一个测试用例没有通过，就停止执行后面的测试用例。这对[持续集成](#)很有用。

```
$ mocha --bail
```

6.6 --grep, -g

`--grep` 参数用于搜索测试用例的名称（即 `it` 块的第一个参数），然后只执行匹配的测试用例。

```
$ mocha --grep "1 加 1"
```

上面代码只测试名称中包含"1 加 1"的测试用例。

6.7 --invert, -i

`--invert` 参数表示只运行不符合条件的测试脚本，必须与 `--grep` 参数配合使用。

```
$ mocha --grep "1 加 1" --invert
```

七，配置文件mocha.opts

Mocha允许在 `test` 目录下面，放置配置文件 `mocha.opts`，把命令行参数写在里面。请先进入 `demo03` 目录，运行下面的命令。

```
$ mocha --recursive --reporter tap --growl
```

上面这个命令有三个参数 `--recursive`、`--reporter tap`、`--growl`。

然后，把这三个参数写入 `test` 目录下的 `mocha.opts` 文件。

```
--reporter tap  
--recursive  
--growl
```

然后，执行 `mocha` 就能取得与第一行命令一样的效果。

```
$ mocha
```

如果测试用例不是存放在`test`子目录，可以在 `mocha.opts` 写入以下内容。

```
server-tests  
--recursive
```

上面代码指定运行 `server-tests` 目录及其子目录之中的测试脚本。

八、ES6测试

如果测试脚本是用ES6写的，那么运行测试之前，需要先用Babel转码。进入 `demo04` 目录，打开 `test/add.test.js` 文件，可以看到这个测试用例是用ES6写的。

```
import add from '../src/add.js';
import chai from 'chai';

let expect = chai.expect;

describe('加法函数的测试', function() {
  it('1 加 1 应该等于 2', function() {
    expect(add(1, 1)).to.be.equal(2);
  });
});
```

ES6转码，需要安装Babel。

```
$ npm install babel-core babel-preset-es2015 --save-dev
```

然后，在项目目录下面，新建一个 `.babelrc` 配置文件。

```
{
  "presets": [ "es2015" ]
}
```

最后，使用 `--compilers` 参数指定测试脚本的转码器。

```
$ ../node_modules/mocha/bin/mocha --compilers js:babel-core/register
```

上面代码中，`--compilers` 参数后面紧跟一个用冒号分隔的字符串，冒号左边是文件的后缀名，右边是用来处理这一类文件的模块名。上面代码表示，运行测试之前，先用 `babel-core/register` 模块，处理一下 `.js` 文件。由于这里的转码器安装在项目内，所以要使用项目内安装的Mocha；如果转码器安装在全局，就可以使用全局的Mocha。

下面是另外一个例子，使用Mocha测试CoffeeScript脚本。测试之前，先将 `.coffee` 文件转成 `.js` 文件。

```
$ mocha --compilers coffee:coffee-script/register
```

注意，Babel默认不会对Iterator、Generator、Promise、Map、Set等全局对象，以及一些全局对象的方法（比如 `Object.assign`）转码。如果你想要对这些对象转码，就要安装 `babel-polyfill`。

```
$ npm install babel-polyfill --save
```

然后，在你的脚本头部加上一行。

```
import 'babel-polyfill'
```

九、异步测试

Mocha默认每个测试用例最多执行2000毫秒，如果到时没有得到结果，就报错。对于涉及异步操作的测试用例，这个时间往往是不够的，需要用 `-t` 或 `--timeout` 参数指定超时门槛。

进入 `demo05` 子目录，打开测试脚本 `timeout.test.js`。

```
it('测试应该5000毫秒后结束', function(done) {  
  var x = true;  
  var f = function() {  
    x = false;  
    expect(x).to.be.not.ok;  
    done(); // 通知Mocha测试结束  
  };  
  setTimeout(f, 4000);  
});
```

上面的测试用例，需要4000毫秒之后，才有运行结果。所以，需要用 `-t` 或 `--timeout` 参数，改变默认的超时设置。

```
$ mocha -t 5000 timeout.test.js
```

上面命令将测试的超时时限指定为5000毫秒。

另外，上面的测试用例里面，有一个 `done` 函数。`it` 块执行的时候，传入一个 `done` 参数，当测试结束的时候，必须显式调用这个函数，告诉Mocha测试结束了。否则，Mocha就无法知道，测试是否结束，会一直等到超时报错。你可以把这行删除试试看。

Mocha默认会高亮显示超过75毫秒的测试用例，可以用 `-s` 或 `--slow` 调整这个参数。

```
$ mocha -t 5000 -s 1000 timeout.test.js
```

上面命令指定高亮显示耗时超过1000毫秒的测试用例。

下面是另外一个异步测试的例子 `async.test.js`。

```
it('异步请求应该返回一个对象', function(done){
  request
    .get('https://api.github.com')
    .end(function(err, res){
      expect(res).to.be.an('object');
      done();
    });
});
```

运行下面命令，可以看到这个测试会通过。

```
$ mocha -t 10000 async.test.js
```

另外，Mocha内置对Promise的支持，允许直接返回Promise，等到它的状态改变，再执行断言，而不用显式调用 `done` 方法。请看 `promise.test.js`。

```
it('异步请求应该返回一个对象', function() {
  return fetch('https://api.github.com')
    .then(function(res) {
      return res.json();
    }).then(function(json) {
      expect(json).to.be.an('object');
    });
});
```

十、测试用例的钩子

Mocha在 `describe` 块之中，提供测试用例的四个钩子：`before()`、`after()`、`beforeEach()` 和 `afterEach()`。它们会在指定时间执行。

```
describe('hooks', function() {

  before(function() {
    // 在本区块的所有测试用例之前执行
  });

  after(function() {
    // 在本区块的所有测试用例之后执行
  });

  beforeEach(function() {
    // 在本区块的每个测试用例之前执行
  });

  afterEach(function() {
    // 在本区块的每个测试用例之后执行
  });

  // test cases
});
```

进入 `demo06` 子目录，可以看到下面两个例子。首先是 `beforeEach` 的例子 `beforeEach.test.js`。

```
// beforeEach.test.js
describe('beforeEach示例', function() {
  var foo = false;

  beforeEach(function() {
    foo = true;
  });

  it('修改全局变量应该成功', function() {
    expect(foo).to.be.equal(true);
  });
});
```

上面代码中，`beforeEach` 会在 `it` 之前执行，所以会修改全局变量。

另一个例子 `beforeEach-async.test.js` 则是演示，如何在 `beforeEach` 之中使用异步操作。

```
// beforeEach-async.test.js
describe('异步 beforeEach 示例', function() {
  var foo = false;

  beforeEach(function(done) {
    setTimeout(function() {
      foo = true;
      done();
    }, 50);
  });

  it('全局变量异步修改应该成功', function() {
    expect(foo).to.be.equal(true);
  });
});
```

十一、测试用例管理

大型项目有很多测试用例。有时，我们希望只运行其中的几个，这时可以用 `only` 方法。`describe` 块和 `it` 块都允许调用 `only` 方法，表示只运行某个测试套件或测试用例。

进入 `demo07` 子目录，测试脚本 `test/add.test.js` 就使用了 `only`。

```
it.only('1 加 1 应该等于 2', function() {
  expect(add(1, 1)).to.be.equal(2);
});

it('任何数加0应该等于自身', function() {
  expect(add(1, 0)).to.be.equal(1);
});
```

上面代码中，只有带有 `only` 方法的测试用例会运行。

```
$ mocha test/add.test.js
```

加法函数的测试

✓ 1 加 1 应该等于 2

1 passing (10ms)

此外，还有 `skip` 方法，表示跳过指定的测试套件或测试用例。

```
it.skip('任何数加0应该等于自身', function() {  
  expect(add(1, 0)).to.be.equal(1);  
});
```

上面代码的这个测试用例不会执行。

十二、浏览器测试

除了在命令行运行，Mocha还可以在浏览器运行。

passes: 2 failures: 0 duration: 0.02s 100%

加法函数的测试

✓ 1 加 1 应该等于 2

✓ 任何数加0等于自身

首先，使用 `mocha init` 命令在指定目录生成初始化文件。

```
$ mocha init demo08
```

运行上面命令，就会在 `demo08` 目录下生成 `index.html` 文件，以及配套的脚本和样式表。

```
<!DOCTYPE html>  
<html>  
  <body>  
    <h1>Unit.js tests in the browser with Mocha</h1>  
    <div id="mocha"></div>  
    <script src="mocha.js"></script>
```



```
<script>
  mocha.setup('bdd');
</script>
<script src="tests.js"></script>
<script>
  mocha.run();
</script>
</body>
</html>
```

然后，新建一个源码文件 `add.js`。

```
// add.js
function add(x, y) {
  return x + y;
}
```

然后，把这个文件，以及断言库 `chai.js`，加入 `index.html`。

```
<script>
  mocha.setup('bdd');
</script>
<script src="add.js"></script>
<script src="http://chaijs.com/chai.js"></script>
<script src="tests.js"></script>
<script>
  mocha.run();
</script>
```

最后，在 `tests.js` 里面写入测试脚本。

```
var expect = chai.expect;

describe('加法函数的测试', function() {
  it('1 加 1 应该等于 2', function() {
    expect(add(1, 1)).to.be.equal(2);
  });

  it('任何数加0等于自身', function() {
    expect(add(1, 0)).to.be.equal(1);
    expect(add(0, 0)).to.be.equal(0);
  });
});
```

```
});
```

现在，在浏览器里面打开 `index.html`，就可以看到测试脚本的运行结果。

十三、生成规格文件

Mocha支持从测试用例生成规格文件。

加法函数的测试

1 加 1 应该等于 2

```
expect(add(1, 1)).to.be.equal(2);
```

任何数加0应该等于自身

```
expect(add(1, 0)).to.be.equal(1);
```

乘法函数的测试

1 乘 1 应该等于 1

```
expect(multiply(1, 1)).to.be.equal(1);
```

进入 `demo09` 子目录，运行下面的命令。

```
$ mocha --recursive -R markdown > spec.md
```

上面命令根据 `test` 目录的所有测试脚本，生成一个规格文件 `spec.md`。





`-R markdown` 参数指定规格报告是markdown格式。

如果想生成HTML格式的报告 `spec.html`，使用下面的命令。

```
$ mocha --recursive -R doc > spec.html
```

(完)

文档信息

- 版权声明：自由转载-非商用-非衍生-保持署名（[创意共享3.0许可证](#)）
- 发表日期：2015年12月 3日
- 更多内容：档案 » JavaScript
- 购买文集： 《如何变得有思想》
- 社交媒体： twitter,  weibo
- Feed订阅：

相关文章

- **2016.04.12:** [跨域资源共享 CORS 详解](#)

CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）。

- **2016.04.08:** [浏览器同源政策及其规避方法](#)

浏览器安全的基石是"同源政策"（same-origin policy）。很多开发者都知道这一点，但了解得不全面。

- **2016.03.12:** [Node 应用的 Systemd 启动](#)

前面的文章介绍了 Systemd 的操作命令和基本用法，今天给出一个实例，如何使用 Systemd 启动一个 Node 应用。

- **2016.02.13:** [React 测试入门教程](#)

越来越多的人，使用React开发Web应用。它的测试就成了一个大问题。

联系方式 | ruanyifeng.com 2003 - 2016