第 3 章

Node.js 基于 Mocha 的测试驱动开发和行为驱动开发

众所周知，测试驱动开发（TDD）是一种主要的敏捷开发技术。它最强大之处是可以提升代码的质量，改进错误的检测方式，以及增强程序员的信心，使其获得更有效率的开发手段。

纵观历史，Web 应用已经越来越难以自动测试，开发者们严重依赖手动测试。但其实，一些特定的项目，比如独立的服务和 REST API 可以且必须用 TDD 来测试。同时，富用户界面（或富用户体验）应用也可以用 PhantomJS 这种无界面浏览器来进行测试。

行为驱动开发（BDD）的概念是基于 TDD 的。它鼓励产品负责人与开发者合作，这一点在语言上不同于 TDD。

多数时候，软件工程师需要使用测试框架，这和构建应用程序本身同等重要。为了让大家快速熟悉测试框架 Mocha，我们将介绍以下几点：

- 安装与理解 Mocha
- 用 assert（断言）进行测试驱动开发
- 用 Expect.js 进行行为驱动开发
- 项目：为博客写第一个 BDD 测试

本章的源代码可以在 practicalnode 的 GitHub 库的 `ch3` 文件夹中找到。[1]

[1] https://github.com/azat-co/practicalnode

安装与理解 Mocha

Mocha 是 Node.js 的一个成熟且强大的测试框架。安装它只需简单地运行以下命令：

```
$ npm install -g mocha@1.16.2
```

■**注意** 我们使用了 Mocha 的一个特定版本（在撰写本文时最新的是 1.16.2），以防止未来版本的差异性造成与本书例子不一致。

如果你遇到第 1 章和第 2 章讨论的缺少权限的问题，运行：

```
$ sudo npm install -g mocha@1.16.2
```

为了避免使用 `sudo` 命令，参见第 1 章关于如何正确安装 Node.js 的说明。

■**提示** 就像其他 NPM 模块一样，你可以在不同项目的 `node_modules` 目录下安装 Mocha 模块，并通过简单的命令指定这个模块，就可以使每一个项目拥有一个独立版本的 Mocha，命令如下：

```
$ ./node_modules/mocha/bin/mocha test_name
```

对于 Mac OS X / Linux 系统，可以参考本章的“将配置参数写入 Makefile”一节。

大家都已经听说过 TDD，以及它为什么是一种值得追随的好技术。TDD 主要的思想罗列如下：

- 定义一个单元测试
- 执行这个单元测试
- 验证这个测试是否通过

BDD 是 TDD 的一个专业版本，它指定了从业务需求的角度出发需要哪些单元测试。虽然，使用 Node.js 核心模块 `assert` 来写测试也是可行的，但是，在多数情况下，用一个框架会更好。我们将使用 Mocha 这个测试框架来实现 TDD 和 BDD，因为我们在 Mocha 模块中获得了许多免费的东西，如下所示：

- 获取测试报告
- 支持异步模式
- 丰富的可配置项

下面的清单是`$ mocha [options]`命令包含的一系列可选的参数。

- `-h` 或`–help`：输出 Mocha 的帮助信息
- `-V` 或`–version`：输出当前 Mocha 的版本号

- -r 或--require <name>：引用一个具名模块
- -R 或--reporter <name>：指定要使用的测试报告的样式方案
- -u 或--ui <name>：指定要使用的测试模式（例如，bdd、tdd）
- -g 或--grep <pattern>：只用匹配模式来运行匹配到的测试
- -i 或-invert：颠倒--grep 的匹配结果
- -t 或--timeout <ms>：用毫秒设置测试用例的超时时间（例如，5000）
- -s 或--slow <ms>：用毫秒设置测试的极限时间（例如，100）
- -w 或-watch：在终端监测测试文件的更改
- -c 或-colors：启用颜色高亮
- -C 或--no-colors：禁用颜色高亮
- -G 或--growl：启用 Mac OS X 的通知
- -d 或--debug：启用 Node.js 调试——$ node --debug
- --debug-brk：启用 Node.js 调试在第一行中断——$ node --debug-brk
- -b 或--bail：在第一次测试失败后退出
- -A 或--async-only：设置所有测试为异步模式
- --recursive：对子文件夹应用测试
- --globals <names>：提供以逗号分隔的全局名称
- --check-leaks：检查全局变量的泄漏
- --interfaces：输出可用的接口
- --reporters：输出可用的测试报告的样式方案
- --compilers <ext>:<module>,...：使用给定的模块来编译文件

图 3-1 是通过$ mocha test-expect.js -R nyan 命令，来使用测试报告的样式方案-nyan 进行测试的例子。

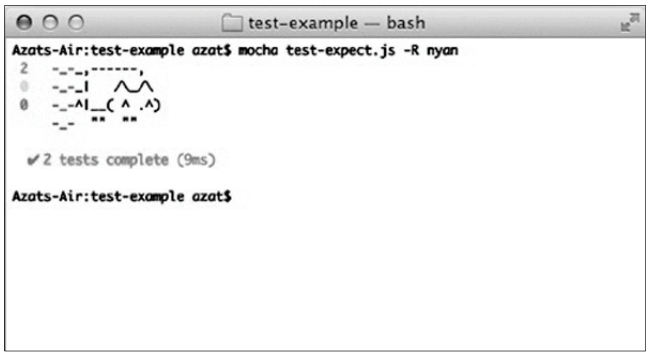

图 3-1　Mocha 测试报告的样式方案 nyan

选择一个测试框架时，通常会有几个备选项。Mocha 是其中非常强大和广泛应用的一个。当然，下面几个框架也值得考虑。

- NodeUnit[2]
- Jasmine[3]
- Vows[4]

理解 Mocha 的 hook 机制

hook 可以理解为是一些逻辑，通常表现为一个函数或者一些声明，当特定的事件触发时 hook 才执行。在第 7 章，我们将会写一些 hook 的代码，以此来理解 Mongoose 库的前置 hook。Mocha 拥有一些内置的 hook，在测试流程的不同时段触发，如在整个测试流程之前，或在每个独立测试之前等。

除了前置的 hook，`before()` 和 `beforeEach()` 以外，还有 `after()` 和 `afterEach()`。它们可以用来清除测试的设置信息，比如数据库数据之类的。

所有的 hook 都支持异步模式。测试也同样支持。例如，下述的测试进程是异步的，它并不用等待响应返回才完成测试：

```
describe('homepage', function(){
it('should respond to GET',function(){
    superagent
        .get('http://localhost:'+port)
        .end(function(res){
        expect(res.status).to.equal(200);
    })
})
```

但是，一旦给测试函数加上 `done` 参数，我们的测试用例就需要等 HTTP 请求返回响应：

```
describe('homepage', function(){
   it('should respond to GET',function(done){
      superagent
          .get('http://localhost:'+port)
          .end(function(res){
              expect(res.status).to.equal(200);
              done();
      })
})
```

测试用例可以嵌套在其他测试用例中，且像 `before` 和 `beforeEach` 这样的 hook 可以

[2] https://github.com/caolan/nodeunit

[3] http://pivotal.github.com/jasmine/

[4] http://vowsjs.org/

在不同的级别被混入到不同的测试用例中。在大型的测试文件中嵌套的结构是一个好主意。

开发者可以使用 `describe.skip()` 或 `it.skip()` 来跳过一个测试用例/进程，也可以使用 `describe.only()` 只执行某个特定的测试用例。

作为 BDD 的接口 `describe`、`it`、`before` 以及其他一些的替代，Mocha 支持更传统的 TDD 接口：

- `suite`：类似 `describe`
- `test`：类似 `it`
- `setup`：类似 `before`
- `teardown`：类似 `after`
- `suiteSetup`：类似 `beforeEach`
- `suiteTeardown`：类似 `afterEach`

用 assert 进行 TDD

assert 库是 Node.js 核心的一部分，这使得它易于访问。虽然它的功能很少，但对于某些情况，例如单元测试已经足够用了。

现在让我们用 assert 库编写第一个测试，在全局 Mocha 模块安装结束后，可以在 `test-example` 文件夹下创建一个测试文件：

```
$ mkdir test-example
$ subl test-example/test.js
```

> ■**注意** `subl` 是使用 Sublime Text 进行编辑的命令。你可以使用任何其他的编辑器，比如 Vi（`vi`）或者 TextMate（`mate`）。

在 `test.js` 中填入以下内容：

```
var assert = require('assert');
describe('String#split', function(){
    it('should return an array', function(){
       assert(Array.isArray('a,b,c'.split(',')));
    });
})
```

我们可以运行这个简单的 `test.js` 文件（在 `test-example` 文件夹中），以测试数据是否为数组类型，使用以下命令：

```
$ mocha test
```

或者

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

图 3-2 展示了以上 Mocha 命令运行的结果。

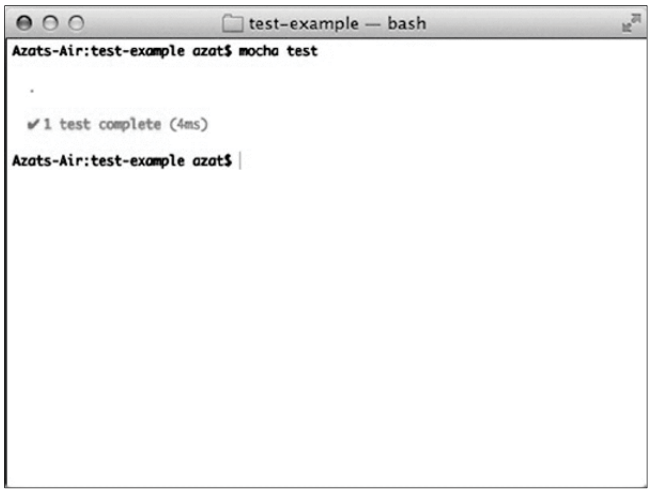

图 3-2　运行数组类型测试

我们还可以使用 it 方法为本例增加其他测试，来判断两个数组是否相等：

```
var assert = require('assert');
describe('String#split', function(){
    it('should return an array', function(){
        assert(Array.isArray('a,b,c'.split(',')));
    });
    it('should return the same array', function(){
      assert.equal(['a','b','c'].length, 'a,b,c'.split(',').length, 'arrays
          have equal length');
      for (var i=0; i<['a','b','c'].length; i++) {
      assert.equal(['a','b','c'][i], 'a,b,c'.split(',')[i], i + 'element is equal');
      };
    });
})
```

如你所见，一些代码是重复的，所以我们可以将代码抽象成 beforeEach 和 before 结构的：

```
var assert = require('assert');
var expected, current;
before(function(){
    expected = ['a', 'b', 'c'];
})
describe('String#split', function(){
    beforeEach(function(){
        current = 'a,b,c'.split(',');
    })
    it('should return an array', function(){
```

```
        assert(Array.isArray(current));
    });
    it('should return the same array', function(){
     assert.equal(expected.length, current.length, 'arrays have equal length');
     for (var i=0; i<expected.length; i++) {
         assert.equal(expected[i], current[i], i + 'element is equal');
     }
  })
})
```

断言库 Chai

在前面的 test.js 例子中，我们用到了 Node.js 的核心模块 assert。Chai 是这个模块的子集。我们可以用 Chai 改写这个例子，代码如下：

```
$ npm install chai@1.8.1
```

在 test-example/test.js 中引入：

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

以下是一些 Chai 的内置方法。

- assert(expressions, message)：如果表达式是错误的则抛出一个错误信息
- assert.fail(actual, expected, [message], [operator])：抛出一个带有实际值、期望值以及操作者的错误信息
- assert.ok(object, [message])：当传入的对象不等于（==）true 时抛出一个错误（在 JavaScript/Node.js 中，0 和空字符串被看作是布尔值 false）
- assert.notOk(object, [message])：当对象是 false 时，比如：false、0、""（空字符串）、null、undefined 或者 NaN，则抛出一个错误
- assert.equal(actual, expected, [message])：当实际值与期望值不相等（==）时抛出一个错误
- assert.notEqual(actual, expected, [message])：当实际值与期望值相等（==）时抛出一个错误，换句话说，断言实际值与期望值不相等（!=）
- .strictEqual(actual, expected, [message])：当实际值和期望值不深度相等（===）时抛出一个错误

完整的 Chai assert 模块 API（应用程序接口），可参考官方文档 http://chaijs.com/api/assert/。

■**注意** chai 与 Node.js 核心组件 assert 并不是 100%兼容的，因为前者拥有更多的方法。后面将要提到的 chai expect 模块以及独立的 expect.js 也同样如此。

用 Expect.js 进行 BDD

Expect.js 是一种 BDD 语言。它的语法是链式风格的，比起核心 assert 模块更加贴近自然语言。有以下两种方式来使用 expect.js：

1. 安装为本地模块
2. 作为 chai 库的一个接口安装

前者的话，简单运行以下命令即可：

```
$ mkdir node_modules
$ npm install expect.js@0.2.0
```

然后，在 Node.js 测试文件中添加代码 `var expect = require('expect.js')` 即可；前面的测试例子可以用 expect.js 改写为 BDD 的模式，代码如下：

```
var expect = require('expect.js');
var expected, current;
before(function(){
expected = ['a', 'b', 'c'];
})
describe('String#split', function(){
  beforeEach(function(){
    current = 'a,b,c'.split(',');
  })
  it('should return an array', function(){
    expect(Array.isArray(current)).to.be.true;
  });
  it('should return the same array', function(){
    expect(expected.length).to.equal(current.length);
    for (var i=0; i<expected.length; i++) {
      expect(expected[i]).equal(current[i]);
    }
  })
})
```

作为 chai 库的接口方式，运行以下命令：

```
$ mkdir node_modules
$ npm install chai@1.8.1
```

然后，在 Node.js 测试文件里用 `var chai = require('chai'); var expect = chai.expect;` 调用该模块。例如：

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

> ■**注意** 只有当你要安装 NPM 模块的文件夹下，既没有 `node_modules` 文件夹也没有 `package.json` 文件时，才需要使用`$ mkdir node_modules` 命令。要获取更多信息，请参考第 1 章。

Expect.js 的语法

Expect.js 库应用十分广泛，它拥有很好的仿自然语言的方法。通常写同一个断言会有几个方法，比如 `expect(response).to.be(true)`和 `expect(response).equal(true)`。以下列举了 Expect.js 的一些主要方法/属性。

- `ok`：检测是否为真
- `true`：检测对象是否为真
- `to.be`、`to`：作为连接两个方法的链式方法
- `not`：链接一个否定的断言，比如 `expect(false).not.to.be(true)`
- `a/an`：检测类型（也适用于数组类型）
- `include/contain`：检测数组或字符串是否包含某个元素
- `below/above`：检测是否大于或小于某个限定值

■**注意** 同样的，独立的 `expect.js` 模块与对应的 Chai 版本略有区别。

大家可以参考完整的 chai expect.js 文档[5]或 expect.js 文档[6]。

项目：为博客开发一个 BDD 测试

这个迷你项目的目标是为本书前面的博客项目加上几个测试。我们不会进行 UI 测试，但是可以发送几个 HTTP 请求，然后解析从应用程序 REST 端返回的响应数据（可以查看第 2 章对博客程序的描述）。

本章的源代码在 practicalnode 的 GitHub 库中的 `ch3/blog-express` 文件夹下[7]。

首先，让我们复制 Hello World 项目作为 Blog 的基础。然后，将依赖信息加入到 `package.json` 文件，同时使用`$ npm install mocha@1.16.2 --save-dev` 命令在博客项目目录下安装 Mocha。`--save-dev` 标识将这个模块归类为开发依赖模块（`package.json` 中对应的 `devDependencies` 字段）。接下来，修改这个命令，将模块名和版本号替换为 expect.js (0.2.0)和 superagent (0.15.7)[8]。后者是简化发起请求的库，与 `superagent` 类似的库有以下几个。

[5] http://chaijs.com/api/bdd/

[6] https://github.com/LearnBoost/expect.js/

[7] https://github.com/azat-co/practicalnode

[8] https://npmjs.org/package/superagent

- request[9]：最受关注的 NPM 模块的第三名（本书编写时）
- 核心模块 http：底层且笨重的模块
- supertest：一个基于 superagent 的断言模块

以下是更新过的 package.json 代码：

```
{
  "name": "blog-express",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js",
    "test": "mocha test"
  },
  "dependencies": {
    "express": "4.1.2",
    "jade": "1.3.1",
    "stylus": "0.44.0"
  },
  "devDependencies": {
    "mocha": "1.16.2",
    "superagent": "0.15.7",
    "expect.js": "0.2.0"
  }
}
```

现在，用$ mkdir tests 命令创建一个测试文件夹，并且在你的编辑器里打开 tests/index.js 文件。本测试用例开始前需要启动服务器，代码如下：

```
var boot = require('../app').boot,
  shutdown = require('../app').shutdown,
  port = require('../app').port,
  superagent = require('superagent'),
  expect = require('expect.js');
describe('server', function () {
  before(function () {
      boot();
  });
describe('homepage', function(){
  it('should respond to GET',function(done){
    superagent
      .get('http://localhost:'+port)
      .end(function(res){
          expect(res.status).to.equal(200);
          done()
    })
  })
});
after(function () {
```

[9] https://npmjs.org/package/request

```
  shutdown();
});
});
```

当我们的测试用例引入app.js时，app.js对外暴露两个方法，boot和shutdown。

所以，我们可以将

```
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

重构为：

```
var server = http.createServer(app);
var boot = function () {
  server.listen(app.get('port'), function(){
     console.info('Express server listening on port ' + app.get('port'));
  });
}
var shutdown = function() {
  server.close();
}
if (require.main === module) {
  boot();
}
else {
  console.info('Running app as a module')
  exports.boot = boot;
  exports.shutdown = shutdown;
  exports.port = app.get('port');
}
```

简单地运行$ mocha tests 命令来启动这个测试。服务器会启动，然后响应主页的（/route）请求，如图 3-3 所示。

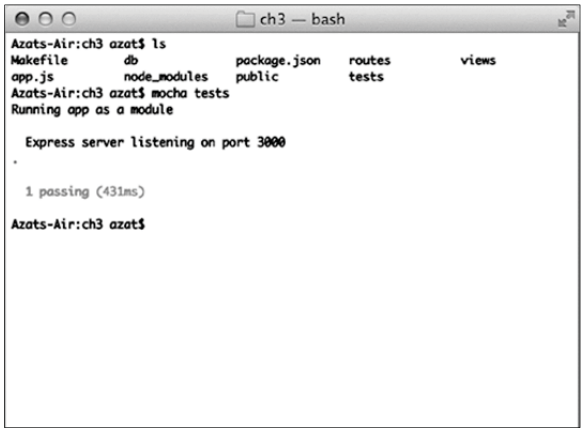

图 3-3 正在运行 $ mocha tests

将配置参数写入 Makefile

mocha 模块接受很多自定义参数。将这些参数集中写入 Makefile（生成文件）是个不错的主意。例如，我们可以用 test 和 test-w 命令来测试 test 文件夹下的所有文件，同时使用不同模式，分别测试 module-a.js 和 module-b.js 文件，配置如下：

```
REPORTER = list
MOCHA_OPTS = --ui tdd --ignore-leaks

test:
      clear
      echo Starting test ******************************************************
      ./node_modules/mocha/bin/mocha \
      --reporter $(REPORTER) \
      $(MOCHA_OPTS) \
      tests/*.js
      echo Ending test
test-w:
      ./node_modules/mocha/bin/mocha \
      --reporter $(REPORTER) \
      --growl \
      --watch \
      $(MOCHA_OPTS) \
      tests/*.js
test-module-a:
        mocha tests/module-a.js --ui tdd --reporter list --ignore-leaks
test-module-b:
        clear
        echo Starting test ******************************************************
        ./node_modules/mocha/bin/mocha \
        --reporter $(REPORTER) \
        $(MOCHA_OPTS) \
        tests/module-b.js
        echo Ending test

.PHONY: test test-w test-module-a test-module-b
```

使用$ make <mode>来运行 Makefile，本例中使用$ make test 即可。更多关于 Makefile 的信息，请参考 http://www.cprogramming.com/tutorial/makefiles.html 中的 Understanding Make 部分，以及 http://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html 中的 Using Make and Writing Makefiles 部分。

对于我们的博客应用，可以保持 Makefile 如下即可：

```
REPORTER = list
MOCHA_OPTS = --ui bdd -c

test:
    clear
    echo Starting test ******************************************************
    ./node_modules/mocha/bin/mocha \
    --reporter $(REPORTER) \
```

```
    $(MOCHA_OPTS) \
    tests/*.js
    echo Ending test
.PHONY: test
```

■**注意** 我们在生成文件里指定了本地的 Mocha 模块，所以需要在 `package.json` 中加入依赖信息，然后安装到 `node_modules` 文件夹中。

现在，可以用`$ make test` 命令来运行这些测试了，相比起简单的`$ mocha tests` 命令，我们追加了更多的配置参数，如图 3-4 所示。

图 3-4 运行 make test 命令

小结

本章，我们将 Mocha 安装为一个命令行工具，并且学习了它的相关参数，而后用 assert 以及 Expect.js 库写了一些简单的测试用例，同时我们通过把 `app.js` 改写为一个模块的方式，为博客程序创建了第一个测试。在第 10 章，我们将会利用分布式持续集成服务 TravisCI，并修改它的 `yml` 配置文件，在 GitHub 的虚拟云空间中实现持久多并发测试。在下一章，我们将了解 Web 应用输出 HTML 的本质——模板引擎。我们将深入学习 Jade 和 Handlebars 模板引擎，然后为博客添加页面。