CHAPTER

16 测试

至此，每次书写完Node程序，都要通过执行程序来检测是否结果和我们预期的一样，从而判断工作是否正常。随着时间越来越长，这种确保程序工作正常并且没有引入新bug的测试方法是非常低效的。

自动化测试是一个过程，它通过执行一系列程序来检验函数工作是否正确。本章首先要介绍的自动化测试的方法是为每个测试创建一段小的 node程序，然后使用原生的`assert`模块来检验。

接着，我们会通过使用一个名为expect.js的项目来优化书写断言（assertion）的过程。紧接着，我们还会介绍如何使用测试框架 Mocha来组织测试代码。

最后，对于在Node和浏览器端都要运行的代码，本章会介绍如何将已有的测试也用到浏览器端。

简单测试

开始前，我们先要确定测试目标。换句话说，我们得确定哪些脚本或者功能需要测试。

测试目标

本章的测试目标是我们在第9章中书写的查询推文的应用。

这里，我们书写一个程序，来断言提交查询时，看看是否能够找到对应的HTML代码来证明查询的推文内容返回了。本例子，推文列表是通过一个或者多个<li>元素来构建的。通过断言查询关键字是否存在，以及<li>字符串是否为HTTP响应内容的一部分来判断应用程序是否工作正常，应当已经很充分了。

要开始自动化测试前，我们先运行应用，确保它能正常工作。然后，通过浏览器访问http://localhost:3000。

测试策略

最基本的测试方法就是书写一个新的node程序，当测试通过时就以状态码0退出程序，失败则以状态码1退出。

如果有未捕获的异常（uncaught exception）抛出，Node会自动以失败错误码退出程序，和我们的设计相符。除此之外，还能获得堆栈踪迹来帮助调试错误原因。因此，书写测试代码时，目标就是要断言条件是否通过，不通过就抛出异常。Node自带了一个assert模块，顾名思义，它的作用就是检查判断条件是否通过，如果不通过，就抛出一个AssertionError异常。

作为例子，我们书写一个测试程序，当时间戳是偶数时成功，是奇数时失败，同时抛出堆栈信息。

```
/**
 * 模块依赖
 */

var assert = require('assert');

/**
 * 断言条件
 */

var now = Date.now();
console.log(now);
assert.ok(now % 2 == 0);
```

291

assert.ok可以用来判断提供的值是否为真（哪怕真是值不是true，我们还是要断言它是true）。当数字除以2没有余数时，那么该数字就是偶数。

现在，我们多运行几次上述程序，看看时间戳：

```
 ∞ simple-testing node assert-example.js
1325520251830
 ∞ simple-testing node assert-example.js
1325520252742
 ∞ simple-testing node assert-example.js
1325520253637

node.js:134
        throw e; // process.nextTick error, or 'error' event on first tick
        ^
AssertionError: true == false
    at Object.<anonymous> (assert-example.js:14:8)
    at Module._compile (module.js:411:26)
    at Object..js (module.js:417:10)
    at Module.load (module.js:343:31)
    at Function._load (module.js:302:12)
    at Array.<anonymous> (module.js:430:10)
    at EventEmitter._tickCallback (node.js:126:26)
```

前两次中，由于时间戳是偶数，所以测试通过了。第三次，时间戳变成了奇数，所以抛出了异常，输出了堆栈信息。

测试程序

现在，我们使用superagent，发送GET请求来查询bieber关键字，然后分析响应结果：

```
/**
 * 模块依赖
 */

var request = require('superagent')
  , assert = require('assert')

/**
 * 测试 /search?q=<tweet>
 */

request.get('http://localhost:3000')
  .data({ q: 'bieber' })
  .exec(function (res) {

  // 断言响应状态码是否正确
  assert.ok(200 == res.status);

  // 断言关键字是否存在
  assert.ok(~res.text.toLowerCase().indexOf('bieber'));

  //断言列表项是否存在
  assert.ok(~res.text.indexOf('<li>'));
});
```

292

- throwException：断言Function在调用时是否会抛出异常。

```
expect(fn).to.throwException();
expect(fn2).to.not.throwException();
```

- within：断言数组是否在某个区间内。

```
expect(1).to.be.within(0, Infinity);
```

- greaterThan/above：断言 >。

```
expect(3).to.be.above(0); expect(5).to.be.greaterThan(3);
```

- lessThan/below：断言 <。

```
expect(0).to.be.below(3); expect(1).to.be.lessThan(3);
```

改进了断言风格，接下来，我们要通过一个名为Mocha的框架重构测试代码的组织方式。

Mocha

Mocha是一个测试框架，简化了书写测试代码的过程，提供划分测试集、运行并同时输出有助于开发者理解的结果页。

相比把一份份测试代码写在不同文件中，会导致的文件系统中存有大量无序文件，通过Mocha，可以书写出如下形式的测试代码：

test.js

```
describe('a topic', function () {

  it('should test something', function () {

  });

  describe('another topic', function () {

    it('should test something else', function () {

    });

  });

});
```

295

与expect.js类似，在test.js中描述和组织测试代码的方式非常自然直观。

要运行测试代码只需通过mocha命令即可。在这之前，需要确保运行npm install -g mocha安装了mocha。然后，就通过如下方式来运行：

- be/equal：类似===。

```
expect(1).to.be(1);
expect(NaN).not.to.equal(NaN);
expect(1).not.to.be(true);
expect('1').to.not.be(1);
```

- eql：断言非严格相等，支持对象。

```
expect({ a: 'b' }).to.eql({ a: 'b' });
expect(1).to.eql('1');
```

- a/an：断言所属类型，支持数组和instanceof。

```
// typeof with optional array
expect(5).to.be.a('number');
expect([]).to.be.an('array'); // works
expect([]).to.be.an('object'); // works too, since it uses typeof`
// constructors
expect(5).to.be.a(Number);
expect([]).to.be.an(Array);
expect(tobi).to.be.a(Ferret);
expect(person).to.be.a(Mammal);
```

- match：断言字符串是否匹配一段正则表达式。

```
expect(program.version).to.match(/[0-9]+\.[0-9]+\.[0-9]+/);
```

- contain：断言字符串是否包含另一个字符串，内部使用indexOf方法。

```
expect([1, 2]).to.contain(1);
expect('hello world').to.contain('world');
```

- length：断言数组长度，内部使用.length方法。

```
expect([]).to.have.length(0);
expect([1,2,3]).to.have.length(3);
```

- empty：断言数组是否为空。

```
expect([]).to.be.empty();
expect([1,2,3]).to.not.be.empty();
```

- property：断言某个自身属性（或值）是否存在。

```
expect(window).to.have.property('expect');expect(window).to.have.
 property('expect', expect)
expect({a: 'b'}).to.have.property('a');
```

- key/keys：断言键是否存在，支持only修饰符。

294

```
js expect({ a: 'b' }).to.have.key('a');
expect({ a: 'b', c: 'd' }).to.only.have.keys('a', 'c');
expect({ a: 'b', c: 'd' }).to.only.have.keys(['a', 'c']);
expect({ a: 'b', c: 'd' }).to.not.only.have.key('a');
```

注意了，如果请求抛出异常，我们直接不做处理，往上抛即可，最终它会变成一个未捕获的异常，导致程序失败并退出。

记住在运行测试前要先安装superagent：

```
npm install superagent@0.4.1
```

expect.js

在此前的例子中，我们使用了assert.ok以及基本的JavaScript表达式。

在看测试代码时，你也许会发现很难看懂到底在测试什么。比如，断言一个字符串是否包含另一个字符串时，最简单的方法就是使用indexOf方法和~操作符，正如此前例子中处理的那样。

expect.js提供了一个简单的expect函数，可以将：

```
assert.ok(~res.text.indexOf('<li>'));
```

改写成：

```
expect(res.text).to.contain('<li>'));
```

通过近自然语言的表达，使得书写和理解测试代码变得更加容易。

expect.js可以通过NPM获取，其名为expect.js，同时，它的官方文档是https://github.com/learnboost/expect.js。

接下来，会介绍一些expect.js的基础API。

API一览

通过引入expect.js模块就可以使用expect函数：

```
var expect = require('expect.js')
```

293 expect.js可以和任何模块一起使用，如此前用到的assert。与所有assert模块提供的函数类似，在expect.js中，当断言失败时会抛出AssertionError。

下面是一些expect.js 0.1.2中最有用的方法。

- ok：断言值是否为真。

```
expect(1).to.be.ok();
expect(true).to.be.ok();
expect({}).to.be.ok();
expect(0).to.not.be.ok();
```

```
mocha test.js
```

Mocha使用多种报告形式来展示测试通过和运行结果。

```
mocha test.js

  ..

    2 tests complete (0ms)
```

比如，有种报告形式是列表：

```
mocha -R list test.js

    a topic should test something: 0ms
    a topic another topic should test something else: 0ms

    2 tests complete (1ms)
```

正如下面将要介绍的，Mocha还有一种HTML报告形式，可以让你直接在浏览器查看运行。

测试异步代码

Mocha默认会在一个测试用例执行完毕后立刻执行另一个。然而，很多时候，当有异步事件发生时，我们希望能够延缓下一个测试用例的执行。

考虑如下例子：

```
it('should not throw', function () {
  setTimeout(function () {
    throw new Error('An error!');
  }, 100);
});
```

上述测试代码总会通过。原因就在于设置了计时器后代码就执行完毕了，Mocha会继续去执行下一个。由于在计时器设置后，没有立刻抛出异常，所以测试通过。

对于这类异常要在将来才会抛出（异步的行为）的测试，我们需要告诉Mocha，等到我们通知它说完成了才能认为该测试完成了。

要解决这个问题，我们只要简单地在回调函数中添加一个参数就可以了。也就是说，正如第2章中介绍的，我们将函数的参数数量（或者是`function#length`）设置为1： 296

```
it('should not throw', function (done) {
  setTimeout(function () {
    assert.ok(1 == 1);
  }, 100);
});
```

现在，测试代码看起来就像中间件一样。Mocha会一直等到done函数调用之后才会认为该测试已经结束。如果该函数在两秒内（默认）没有调用，就会抛出一个超时异常来告诉你哪个测试卡住了。你也可以通过mocha命令的-t选项来修改这个超时时长。还可以通过如下方式来自定义超时时长：

```
it('will fail', function () {
  this.timeout(100);
  setTimeout(function () {
    // the test will timeout before this occurs
  }, 1000);
});
```

为了让这个测试用例能够工作，我们在断言结束后调用done方法：

```
it('should not throw', function (done) {
  setTimeout(function () {
    assert.ok(1 == 1);
    done();
  }, 100);
});
```

我们可以使用Mocha将此前测试查询Bieber推文应用的代码改写为如下形式：

```
it('should find bieber tweets', function (done) {
  request.get('http://localhost:3000')
    .data({ q: 'bieber' })
    .exec(function (res) {

      //断言状态码是否正确
      assert.ok(200 == res.status);

      //断言查询关键字是否存在
      assert.ok(~res.text.toLowerCase().indexOf('bieber'));

      //断言列表项是否存在
      assert.ok(~res.text.indexOf('<li>'));

      done();
    });
});
```

297

有时，一个测试用例只有在当多个异步操作一起完成时才算通过。

这时我们就可以使用一个计数器来解决这个问题：

```
it('should complete three requests', function (done) {
  var total = 3;
  request.get('http://localhost:3000/1', function (res) {
    if (200 != res.status) throw new Error('Request error');--total || done();
  });
```

```
  request.get('http://localhost:3000/2', function (res) {
    if (200 != res.status) throw new Error('Request error');--total || done();
  });
  request.get('http://localhost:3000/3', function (res) {
    if (200 != res.status) throw new Error('Request error');;
    --total || done();
  });
});
```

注意了，Mocha非常“聪明”，它能够识别出未捕获的异常属于哪个测试用例。这是因为Mocha任何时候只执行一个测试用例，所以它能够准确地将通过process.on('uncaughtException')处理器捕获的未捕获的错误（uncaught Exception）链接到正确的测试用例上。

BDD风格

前面小节中使用的测试代码书写风格称为：行为驱动开发（BDD）。

在接下来的例子中，我们给jade提供一个包含段落的模板来测试Jade的处理行为。过程中还会用到expect.js。

首先，安装Jade、Mocha和expect.js：

```
$ npm install expect.js jade mocha
```

bdd.js

```
var expect = require('expect.js')
  , jade = require('jade');

describe('jade.render', function () {
  it('should render a paragraph', function () {
    expect(jade.render('p A paragraph')).to.be('<p>A paragraph</p>');
  });
});
```

由于Mocha安装在项目路径中时并非是全局安装，所以，要在./node_modules/bin找到它并执行：

```
$ ./node_modules/.bin/mocha bdd.js
```

TDD风格

298

接下来要介绍的是测试驱动开发（TDD）。它和BDD类似，但是组织方式是使用测试集（suite）和测试（test）。

每个测试集都有setup和teardown函数。这些方法会在测试集中的测试执行前执行，它们的作用是为了避免代码重复以及最大限度使得测试之间相互独立。

现在，测试代码看起来就像中间件一样。Mocha会一直等到done函数调用之后才会认为该测试已经结束。如果该函数在两秒内（默认）没有调用，就会抛出一个超时异常来告诉你哪个测试卡住了。你也可以通过mocha命令的-t选项来修改这个超时时长。还可以通过如下方式来自定义超时时长：

```
it('will fail', function () {
  this.timeout(100);
  setTimeout(function () {
    // the test will timeout before this occurs
  }, 1000);
});
```

为了让这个测试用例能够工作，我们在断言结束后调用done方法：

```
it('should not throw', function (done) {
  setTimeout(function () {
    assert.ok(1 == 1);
    done();
  }, 100);
});
```

我们可以使用Mocha将此前测试查询Bieber推文应用的代码改写为如下形式：

```
it('should find bieber tweets', function (done) {
  request.get('http://localhost:3000')
    .data({ q: 'bieber' })
    .exec(function (res) {

      //断言状态码是否正确
      assert.ok(200 == res.status);

      //断言查询关键字是否存在
      assert.ok(~res.text.toLowerCase().indexOf('bieber'));

      // 断言列表项是否存在
      assert.ok(~res.text.indexOf('<li>'));

      done();
    });
});
```

297

有时，一个测试用例只有在当多个异步操作一起完成时才算通过。

这时我们就可以使用一个计数器来解决这个问题：

```
it('should complete three requests', function (done) {
  var total = 3;
  request.get('http://localhost:3000/1', function (res) {
    if (200 != res.status) throw new Error('Request error');--total || done();
  });
```

2 告诉Mocha采用什么样的测试风格（TDD、BDD或者export）。

3 载入测试代码。

4 运行Mocha。

expect.js可以在所有浏览器中运行，它可以很好地和Mocha配合使用。

建立项目

我们从创建一个test/文件夹开始，该文件夹包含Mocha在浏览器端运行所需文件（mocha.css和mocha.js）。

我们可以从node_modules/mocha目录中将这两个文件复制过来，或者从git仓库中直接下载。要了解如何下载这些文件，可以参考 http://mochajs.com。

我们还需要载入jQuery、expect.js以及测试代码，然后调用mocha.setup来设置测试风格。

最终，test/index.html代码如下所示：

test/index.html

```
<!doctype html>
<html>
  <head>
    <title>my tests</title>
    <link href="/mocha.css" rel="stylesheet" media="screen" />
    <script src="/jquery.js"></script>
    <script src="/mocha.js"></script>
    <script src="/expect.js"></script>
    <script>mocha.setup('bdd');</script>
    <script src="/my-test.js"></script>
    <script>window.onload = function () { mocha.run(); };</script>
  </head>
  <body>
    <div id="mocha"></div>
  </body>
</html>
```

300

如上述代码所示，Mocha会执行my-test文件，该文件中以BDD的风格做了一些简单的测试：

my-test.js

```
describe('my tests', function () {
  it('should not throw', function () {
    expect(1 + 1).to.be(2);
  });
});
```

托管目录

最简单的以网站的形式托管整个目录的方式是使用serve(1)命令。这是一个简单的命令行程序，serve内部使用connect的static中间件来托管指定的目录：

```
$ serve .
```

之后，简单地通过浏览器访问http://localhost:3000就可以了。

小结

本章一开始介绍了如何以最简单的方式书写测试程序：运行一个简单的测试脚本并查看其运行结果是否成功。

接着，为了验证在测试脚本中，某段条件判断是否通过，本章介绍了一个Node.js核心模块assert。

在这基础上，本章又介绍了使用expect.js和Mocha来提升测试代码风格以及组织形式。Expect可以让书写的测试代码清晰易懂，而Mocha提供了很好的组织测试代码的方法，同时还允许将测试代码在浏览器端运行。除此之外，对于测试异步代码，也游刃有余。