

# API 自动化测试

## 一：背景介绍

**BDD**: 敏捷测试的一种方法，强调以需求为导向，从宏观出发，让开发者做“正确的事情”，有别于 **TDD** 的注重细节。

**BDD** 风格：

```
describe('description ...', function() {
  before(function(){
    //...
  });

  it('it shoud be ...', function() {
    var tmp=[1,2,3];
    tmp.indexOf(4).should.equal(-1);
  });

  after(function(){
    //...
  });
})
```

常用的 **BDD** 接口有：

- **describe()**: 描述场景，在里面可以设定 **Context**，可包括多个测试用例，也可以嵌套场景
- **it()**: 位于场景内，描述测试用例
- **before()**: 所有测试用例的统一前置动作
- **after()**: 所有测试用例的统一后置动作
- ... ....

在文本开始前，首先思考一个问题，为什么需要做 **API** 测试？相信很多测试人员以及开发人员都抱怨过，新开发的功能影响甚至破坏之前的功能，尤其是系统规模已经很庞大的时候，经常无法准确定位问题所在；环境、配置等的改变经常会又会使得系统的某些模块出现异常错误；一切开发工作都完成了，发现逻辑原来是错误的等；经过项目实践调查，90% 甚至以上影响系统正常工作的都是 **API** 的问题。基于上述原因，**API** 测试是至关重要的，它能够快速地帮助开发发现并定位问题，验证技术实现逻辑的正确性，以及可以成为需求说明的活文档。

那么为什么 **supertest**, **mocha**, **chai** 等这些工具呢？一般测试人员的代码开发能力相比开发人员较弱一些，加之复杂的环境，繁琐的代码等，会让测试人员还没上手就感到焦头烂额。**mocha**, **chai** 等进行 **BDD** 测试，也是目前比较流行的 **Nodejs** 的测试方法，配合 **supertest**，其环境搭建简单，上手容易，使用简单，不需要较复杂的配置，且使用灵活，无依赖性，比较适用于交付压力大，项目周期短的团队使用，

## 二：测试环境搭建

使用 **mkdir** 命令新建项目目录，然后到该目录下，就可以安装 **API** 测试所需要的安装包了。

**supertest** 是一个测试 **http** 请求的库。本文主要针对 **http** 请求的 **API** 测试，所以这

是一个非常理想的工具。

**supertest** 安装：

命令: `npm install supertest --save-dev`

创建一个目录 `simple_test`, 安装 `supertest`, 待成功安装之后, 若在项目中调用使用语句: `require("supertest");`

```
jizhou:~/Documents$ mkdir simple_test
jizhou:~/Documents$ cd simple_test/
jizhou:~/Documents/simple_test$ npm install supertest --save-dev
supertest@1.1.0 node_modules/supertest
├── methods@1.1.1
└── superagent@1.3.0 (extend@1.2.1, methods@1.0.1, cookiejar@2.0.1, component-emitter@1.1.2, reduce-component@1.0.1, mime@1.3.4, qs@2.3.3, formidable@1.0.14, readable-stream@1.0.27-1, debug@2.2.0, form-data@0.2.0)
jizhou:~/Documents/simple_test$ 
```

`mocha` 支持 BDD/TDD, 与 `Jasmine` 不同, `Mocha` 只关心总体的结构, 而对于实际的断言毫不关心。这不仅允许我们持续观察测试, 同时也允许我们自由选择使用自己喜欢的断言库。

**mocha** 安装：

命令: `npm install -g mocha` (带参数`-g` 为全局安装)

```
jizhou:~/Documents/simple_test$ npm install -g mocha
/usr/local/bin/mocha -> /usr/local/lib/node_modules/mocha/bin/mocha
/usr/local/bin/_mocha -> /usr/local/lib/node_modules/mocha/bin/_mocha
mocha@2.3.4 /usr/local/lib/node_modules/mocha
├── escape-string-regexp@1.0.2
├── supports-color@1.2.0
├── growl@1.8.1
├── diff@1.4.0
├── commander@2.3.0
├── jade@0.26.3 (commander@0.6.1, mkdirp@0.3.0)
├── debug@2.2.0 (ms@0.7.1)
├── mkdirp@0.5.0 (minimist@0.0.8)
└── glob@3.2.3 (inherits@2.0.1, graceful-fs@2.0.3, minimatch@0.2.14)
jizhou:~/Documents/simple_test$ 
```

`Chai` 是一个非常好的 `assert` 库的替代品。

**chai** 安装：

命令: `npm install chai`

```
jizhou:~/Documents/simple_test$ npm install chai
chai@3.4.1 node_modules/chai
├── assertion-error@1.0.1
├── type-detect@1.0.0
└── deep-eql@0.1.3 (type-detect@0.1.1)
jizhou:~/Documents/simple_test$ 
```

`Grunt` 是一个基于任务的 `JavaScript` 项目命令行构建工具, 运行于 `Node.js` 平台

**grunt** 安装:

命令: `npm install grunt-cli -g`

```
jizhou:~/Documents/simple_test$ npm install grunt-cli -g
npm WARN deprecated lodash@2.4.2: lodash@<3.0.0 is no longer maintained. Upgrade to lodash@^3.0.0.
/usr/local/bin/grunt -> /usr/local/lib/node_modules/grunt-cli/bin/grunt
grunt-cli@0.1.13 /usr/local/lib/node_modules/grunt-cli
├─ resolve@0.3.1
└─ nopt@1.0.10 (abbrev@1.0.7)
└─ findup-sync@0.1.3 (lodash@2.4.2, glob@3.2.11)
jizhou:~/Documents/simple_test$
```

grunt-mocha-test:集成 Mocha 到 Grunt

命令: `npm install grunt-mocha-test`

```
jizhou:~/Documents/simple_test$ npm install grunt-mocha-test
grunt-mocha-test@0.12.7 node_modules/grunt-mocha-test
├─ hooker@0.2.3
└─ mkdirp@0.5.1 (minimist@0.0.8)
jizhou:~/Documents/simple_test$
```

至此环境已经搭建完成了，为了更好的管理测试文件，我们按照不同的文件用途来建立目录，目录结构为：

```
jizhou:~/Documents/simple_test$ mkdir test
jizhou:~/Documents/simple_test$ ls
node_modules test
jizhou:~/Documents/simple_test$ cd test
jizhou:~/Documents/simple_test/test$ mkdir config
jizhou:~/Documents/simple_test/test$ mkdir module
jizhou:~/Documents/simple_test/test$ mkdir scenario
jizhou:~/Documents/simple_test/test$ ls
config module scenario
jizhou:~/Documents/simple_test/test$
```

在项目名下的“`node_modules`”的文件夹就是我们刚才安装的环境搭建的包。所有我们要自己编写的与测试相关的文件都放在项目名称下的“`test`”这个文件夹里，“`config`”为配置文件夹，“`module`”为模块测试文件夹，“`scenario`”为场景测试的文件夹，分别存放测试的 `js` 文件。下面让我们跑一些简单的测试吧。

### 三： 测试 HTTP 请求

HTTP 请求指从客户端到服务器端的请求消息。一个完整的 HTTP 请求包括：一个请求行、若干消息头以及实体内容，而消息头和实体内容可以没有。请求发送之后有相应的返回码，返回状态以及返回内容。

介绍下常见的 HTTP 请求方式有以下几种：

`GET`:向特定的资源发出请求。

`POST`:向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。`POST` 请求可能会导致新的资源的创建和/或已有资源的修改。

`PUT`:向指定资源位置上传其最新内容。

`DELETE`:请求服务器删除 `Request-URI` 所标识的资源。

下面我们举几个简单的例子：

## 1. 异步调用

因为请求通常都是异步的，所以可以使用 `done` 异步方法来完成回调，通常在测试结束之后调用它。可用于下一用例的等待。

```
describe('read file', function () {
  it('content should not empty', function (done) {
    require('file').read('test.txt', function(err, res){
      res.should.not.equal(null);
      done();
    });
  });
});
```

本例中调用 `done()` 方法，完成异步回调。

## 2. POST 方法测试

本例中用 `POST` 来创建一个宠物对象，对将它指定到某个门店里，使用 `request.post` 向 `http://petstore.swagger.io/v2/pet` 发送请求，请求参数为 `send` 中的 `json` 对象，请求如果成功，返回的状态码 `200`，且拿到状态 `pet_id` 并将其存储起来，供后续的 API 调用。

```
describe('As a management for pet store, I need to manage kinds of pets.', function() {
  it('The management should add a new pet in my store successfully.', function(done) {
    request.post('/pet')
      .send({
        "id": 10,
        "category": {
          "id": 11,
          "name": "一号宠物店",
        },
        "name": "doggie",
        "photoUrls": [
          "http://www.mypet.picture.com"
        ],
        "tags": [
          {
            "id": 12,
            "name": "my pet"
          }
        ],
        "status": "available"
      })
      .expect(200, function(err, info){
        pet_id = info.res.body.id
        done();
        return pet_id;
      });
  });
});
```

以下是运行结果：

```
jizhou:~/Documents/simple_test$ mocha test/module/pet.js

As a management for pet store, I need to manage kinds of pets.
✓ The management should add a new pet in my store successfully. (837ms)

1 passing (843ms)

jizhou:~/Documents/simple_test$ █
```

第一行是测试场景，下面都是场景下的测试用例。可以看到该用例得到期望的返回结果，添加成功，测试通过。如果有数据库权限的话，这个时候去 `check` 数据库将会看到表中多了一行上面 `POST` 的数据。

### 3.GET 方法测试：

使用 `request.get` 发送请求，请求结果与 `.expect` 方法中的内容进行对比，若返回正确，则 API 测试通过，若其它结果，则测试失败。（更多的函数详情见 `supertest` 的官方文档）。

本例中我们使用了上面 `POST` 返回的 `pet_id` 作为筛选条件查询。

```
it('The management should get the pet which is just added by pet_id.', function(done) {
  request.get('/pet/'+pet_id)
    .expect(200, {
      "id": 10,
      "category": {
        "id": 11,
        "name": "一号宠物店",

        "name": "doggie",
        "photoUrls": [
          "http://www.mypet.picture.com"
        ],
        "tags": [
          {
            "id": 12,
            "name": "my pet"
          }
        ],
        "status": "available"
      }, done);
});

});
```

测试运行结果：

```
jizhou:~/Documents/simple_test$ mocha test/module/pet.js

As a management for pet store, I need to manage kinds of pets.
✓ The management should add a new pet in my store successfully. (524ms)
✓ The management should get the pet which is just added by pet_id. (1712ms)

2 passing (2s)

jizhou:~/Documents/simple_test$ █
```

#### 4. PUT 方法测试：

修改之前 `post` 的数据，设置变量 `setPet`，将宠物店的名字设为“二号宠物店”，然后发送 `put` 请求。

```
it('The management should update the new pet information by pet_id.', function(done) {
  var setPet = {
    "id": 10,
    "category": {
      "id": 11,
      "name": "二号宠物店",
      "name": "doggie",
      "photoUrls": [
        "http://www.mypet.picture.com"
      ],
      "tags": [
        {
          "id": 12,
          "name": "my pet"
        }
      ],
      "status": "available"
    }
    request.put('/pet/' + pet_id)
      .send(setPet)
      .expect(200, done)
  });
});
```

测试运行结果：

```
jizhou:~/Documents/simple_test$ mocha test/module/pet.js
```

```
As a management for pet store, I need to manage kinds of pets.
```

- ✓ The management should add a new pet in my store successfully. (545ms)
- ✓ The management should get the pet which is just added by pet\_id. (475ms)
- ✓ The management should update the new pet information by pet\_id. (466ms)

```
3 passing (1s)
```

```
jizhou:~/Documents/simple_test$ []
```

测试通过，说明修改成功。也可以使用 `Get` 请求作出进一步的验证。从返回的结果中查看宠物店的名字是否修改为“二号宠物店”。

```
it('The management should get the pet information after update by pet_id.', function(done) {
  request.get('/pet/' + pet_id)
    .expect(200, function(err, info){
      info.res.body.category.name.should.equal('二号宠物店')
      done();
    });
});
```

测试运行结果：

```
jizhou:~/Documents/simple_test$ mocha test/module/pet.js
```

As a management for pet store, I need to manage kinds of pets.

- ✓ The management should add a new pet in my store successfully. (478ms)
- ✓ The management should get the pet which is just added by pet\_id. (454ms)
- ✓ The management should update the new pet information by pet\_id. (458ms)
- ✓ The management should get the pet information after update by pet\_id. (627ms)

4 passing (2s)

```
jizhou:~/Documents/simple_test$ █
```

## 四：项目应用

上一章节我们讲了发送简单的请求，其实 API 测试真正的价值在于对复杂业务场景进行测试，以验证逻辑的正确性。下面我们以实际项目中一个简单的场景为背景，用 BDD 的方法来讲述以上框架及技术在具体项目中的应用。

### 1. 场景测试

这个场景类似于滴滴打车，用户发出需要打车的请求，后台服务器便会创建订单，附近的司机搜索订单，并接单的过程。注：为了方便测试，对测试场景进行特殊设置，使得用户和司机的距离在 10 公里以内，因此测试数据中司机和用户的当前位置的经纬度也必须在不超过 10 公里的范围内。

主要测试代码分析如下：

```
describe('Scenario:In order to go home from office, I want to call a taxi by App,  
         this order be pushed to some drivers and the one accept it', function (done) {  
  var orderId;  
  
  it('Given the user opened the App and created a order', function (done) {  
    this.timeout(10000);  
    request  
      .post('/mobile/order')  
      .send({  
        "pickupLatitude": 34.225274,  
        "pickupLongitude": 108.970724,  
        "pickupAddress": "YanTa District, Xi'An city",  
        "backLatitude": 34.200922,  
        "backLongitude": 108.895712,  
        "backAddress": "GaoXin District, Xi'An city",  
        "contactNumber": "13708211782",  
      })  
      .set('Content-Type', 'application/json')  
      .set('EVA-ACCESS-TOKEN', token.owner_immediate_order)  
      .expect(201, function (err, res) {  
        res.body.should.to.have.property('id');  
        orderId = res.body.id;  
        done();  
      });  
  });  
});
```

用户通过手机 App 发出打车请求，包括 user 要求被接的地址和要送达的地址以及 user 的联系方式，后台系统根据这些信息创建订单。图中的 post 请求用来创建了这个订单，创建订单这个动作必须需要 App 使用者的权限发出，代码中有对 token 的设置。如果订单

创建成功会返回 201。在这里我们保存了订单的 id，以便后续逻辑使用。

下图为用户通过 APP 创建订单成功后查看新创建的订单的状态，若状态为“NEW”，则创建成功且返回码为 200。若不为“NEW”，则获取订单状态失败。

```
it('When search the order and the orderStatus is NEW.', function (done) {
  request
    .get('/mobile/order/' + orderId + '/status')
    .set('EVA-ACCESS-TOKEN', token.owner_immediate_order)
    .expect(200)
    .expect(function (res) {
      if (!(res.body.orderStatus.should.equal('NEW')))
        throw new Error("Failed to get the orderStatus")
    })
    .end(done);
});
```

下图为司机上传自己当前的位置，后台 API 则根据司机的位置搜索附近的订单。可以看到代码中的 token 为司机的 token，且 set 中设置了司机的位置的经纬度。且在请求成功之后会返回之前我们新建的订单的订单号。

```
it('And the driver search the order nearby.', function (done) {
  request
    .post('/mobile/driver/order-nearby')
    .set('EVA-ACCESS-TOKEN', token.driver)
    .send({
      "latitude": 34.205057,
      "longitude": 108.97977,
      "cityName": "Xi'An"
    })
    .expect(200)
    .expect(function (res) {
      res.body.uuid.should.to.equal(parseInt(orderId));
    })
    .end(done);
});
```

当司机搜索到订单之后便可以接单了，下图为接单的 API 的测试，可以看到接单 URL 的参数为刚才我们创建的 orderId，且只有司机才有权限接单。

```
it('Then the driver accept this order and notify the user.', function (done) {
  request
    .put('/mobile/driver/accept-order/' + orderId)
    .set('EVA-ACCESS-TOKEN', token.driver)
    .expect(200)
    .end(done);
});
```

后续还有对接单之后订单状态以及各个细节的检查，基于篇幅问题，这里不再赘述。  
测试运行结果：

```
jizhou:~/Documents/simple_test$ mocha test/scenario/take_taxi.js
```

Scenario: In order to go home from office, I want to call a taxi by App, this order be pushed to some drivers and the one accept it.

- ✓ Given the user opened the App and created a order. (817ms)
- ✓ When search the order and the orderStatus is NEW. (479ms)
- ✓ And the order type is IMMEDIATE. (455ms)
- ✓ And the driver search the order nearby. (450ms)
- ✓ Then the driver accept this order and notify the user. (568ms)
- ✓ Then the orderStatus is OCCUPIED. (452ms)

6 passing (3s)

```
jizhou:~/Documents/simple_test$ █
```

## 2. 参数配置

在测试的过程中，测试人员经常需要测试很多环境，自动化测试也是一样，需要运行多个环境，因此就需要在 `config` 的文件夹中去设置。

在 `config` 中新建一个 `host_config.js` 的配置文件，列出所有需要测试的环境的地址：

```
module.exports = {  
  host : {  
    local : 'http://localhost:8081/rest/',  
    staging : 'http://[IP]:8081/rest',  
    inactive_production : 'http://[IP]:8081/gaia/rest',  
  },  
  env: process.env.NODE_ENV || 'local'  
}
```

图中 `{IP}` 可以填写真实项目的 IP 地址，测试文件中 API 的 url 均是相对，因此可以在文件开始引入 `host_config.js` 文件，使用 `var config=require('../config/host_config')`。

使用命令 `export NODE_ENV=staging`，再运行相关的测试文件，这样测试的便是在 `staging` 环境下的 API 了。此方法免去了每次运行时去修改各个 js 文件中的地址，同时方便运行多个环境下的测试，只需要做一个简单的配置即可。

在上述场景测试演示过程中，大家会发现出现了几个 `token`，都是通过变量调用的。在大型项目中，可能有多种甚至十几种不同的用户角色。因此参数设置对于提高测试的效率以及灵活性有很大的用处。

## 3. 与 CI 集成

CI（持续集成），是由一系列最佳实践构成：源代码的版本控制和管理、自动化构建、代码审查、自动化测试、自动部署和持续反馈等。通过持续地对代码和测试尽早频繁地集成来尽早地发现代码中的问题。

仍以上述项目为例，我们期望当后台开发人员提交代码后，自动在各个环境中执行 API 测试，以期望可以尽早尽快地发现 API 中存在的问题。因此可以在 Jenkins 中做一些简单的配置便可以实现自动化运行 API 测试了，如下图所示：

Project name

Description

[\[Plain text\]](#) [Preview](#)

Discard Old Builds [?](#)

GitHub project

This build is parameterized [?](#)

Permission to Copy Artifact [?](#)

Delivery Pipeline configuration [?](#)

Stage Name

Task Name

Disable Build (No new builds will be executed until the project is re-enabled.) [?](#)

Execute concurrent builds if necessary [?](#)

**Execute shell** [?](#)

Command

See [the list of available environment variables](#)

[Delete](#)

图中上半部分是一些简要说明信息，下半部分是测试运行的核心命令，{username}，{IP}，{password}可以填写真实的 IP 以及用户名密码。通过设置 NODE\_ENV=staging 且运行 grunt 命令，在 staging 环境上实现自动运行测试代码的效果，其它环境类似。根据上述的配置在项目的各个环境上，当开发提交一次版本，随着部署的完成，所有 API 测试都会在各个环境中运行一遍。

测试运行完成之后，在 console 中查看测试运行日志，如下图所示：

```
+ grunt
Running "mochaTest:test" (mochaTest) task
```

As a management for pet store, I need to manage kinds of pets.

- ✓ The management should add a new pet in my store successfully. (1217ms)
- ✓ The management should get the pet which is just added by pet\_id. (515ms)
- ✓ The management should update the new pet information by pet\_id. (451ms)
- ✓ The management should get the pet information after update by pet\_id. (481ms)

Scenario: In order to go home from office, I want to call a taxi by App, this order be pushed to some drivers and the one accept it.

- ✓ Given the user opened the App and created a order. (630ms)
- ✓ When search the order and the orderStatus is NEW. (536ms)
- ✓ And the order type is IMMEDIATE. (506ms)
- ✓ And the driver search the order nearby. (450ms)
- ✓ Then the driver accept this order and notify the user. (448ms)
- ✓ Then the orderStatus is OCCUPIED. (473ms)

10 passing (6s)

Done, without errors.

通过 **grunt** 命令，运行了该项目下所有测试用例，均成功通过。