编写合约测试脚本

我们已经实现了合约的编译和部署的自动化，这将大大提升我们开发的效率。但流程的自动化并不能保证我们的代码质量。质量意识是靠谱工程师的基本职业素养，在智能合约领域也不例外：任何代码如果不做充分的测试，问题发现时通常都已为时太晚；如果代码不做自动化测试，问题发现的成本就会越来越高。

在编写合约时，我们可以利用 remix 部署后的页面调用合约函数，进行单元测试；还可以将合约部署到私链，用 geth 控制台或者 node 命令行进行交互测试。但这有很大的随意性，并不能形成标准化测试流程；而且手动一步步操作，比较繁琐，不易保证重复一致。

于是我们想到，是否可以利用现成的前端技术栈实现合约的自动化测试呢？当然是可以的，mocha 就是这样一个 JavaScript 测试框架。

安装依赖

开始编写测试脚本之前，我们首先需要安装依赖：测试框架 mocha。当然，作为对合约的测试，模拟节点 ganache 和 web3 都是不可缺少的；不过我们在上节课编写部署脚本时，已经安装了这些依赖（我们的 web3 依然是 1.0.0 版本）。

```
npm install mocha -save-dev
```

进行单元测试，比较重要的一点是保证测试的独立性和隔离性，所以我们并不需要测试网络这种有复杂交互的环境，甚至不需要本地私链保存测试历史。而 ganache 基于内存模拟以太坊节点行为，每次启动都是一个干净的空白环境，所以非常适合我们做开发时的单元测试。还记得 ganache 的前身叫什么吗？就是大名鼎鼎的 testRPC。

mocha 简介

mocha 是 JavaScript 的一个单元测试框架，既可以在浏览器环境中运行，也可以在 node.js 环境下运行。我们只需要编写测试用例，mocha 会将测试自动运行并给出测试结果。

mocha 的主要特点有：

- 既可以测试简单的 JavaScript 函数，又可以测试异步代码；
- 可以自动运行所有测试，也可以只运行特定的测试；
- 可以支持 before、after、beforeEach 和 afterEach 来编写初始化代码。

测试脚本示例

假设我们编写了一个 sum.js，并且输出一个简单的求和函数：

```
module.exports = function (...rest) {
    var sum = 0;
    for (let n of rest) {
        sum += n;
    }
    return sum;
};
```

这个函数非常简单，就是对输入的任意参数求和并返回结果。

如果我们想对这个函数进行测试，可以写一个 test.js，然后使用 Node.js 提供的 assert 模块进行断言：

```
const assert = require('assert');
const sum = require('./sum');

assert.strictEqual(sum(), 0);
assert.strictEqual(sum(1), 1);
assert.strictEqual(sum(1, 2), 3);
assert.strictEqual(sum(1, 2, 3), 6);
```

assert 模块非常简单，它断言一个表达式为 true。如果断言失败，就抛出 Error。

单独写一个 test.js 的缺点是没法自动运行测试，而且，如果第一个 assert 报错，后面的测试也执行不了了。

如果有很多测试需要运行，就必须把这些测试全部组织起来，然后统一执行，并且得到执行结果。这就是我们为什么要用 mocha 来编写并运行测试。

我们利用 mocha 修改后的测试脚本如下：

```
const assert = require('assert');
const sum = require('../sum');

describe('#sum.js', () => {

    describe('#sum()', () => {
        it('sum() should return 0', () => {
            assert.strictEqual(sum(), 0);
        });

        it('sum(1) should return 1', () => {
            assert.strictEqual(sum(1), 1);
        });

        it('sum(1, 2) should return 3', () => {
            assert.strictEqual(sum(1, 2), 3);
        });

        it('sum(1, 2, 3) should return 6', () => {
            assert.strictEqual(sum(1, 2, 3), 6);
        });
    });
```

```
});
```

这里我们使用 mocha 默认的 BDD-style 的测试。describe 可以任意嵌套，以便把相关测试看成一组测试。

describe 可以任意嵌套，以便把相关测试看成一组测试；而其中的每个 it 就代表一个测试。

每个 it("name", function() {...})就代表一个测试。例如，为了测试 sum(1, 2)，我们这样写：

```
it('sum(1, 2) should return 3', () => {
    assert.strictEqual(sum(1, 2), 3);
});
```

编写测试的原则是，一次只测一种情况，且测试代码要非常简单。我们编写多个测试来分别测试不同的输入，并使用 assert 判断输出是否是我们所期望的。

运行测试脚本

下一步，我们就可以用 mocha 运行测试了。打开命令提示符，切换到项目目录，然后创建文件夹 test，将 test.js 放入 test 文件夹下，执行命令：

```
./node_modules/mocha/bin/mocha
```

mocha 就会自动执行 test 文件夹下所有测试，然后输出如下：

```
  #sum.js
    #sum()
      ✓ sum() should return 0
      ✓ sum(1) should return 1
      ✓ sum(1, 2) should return 3
      ✓ sum(1, 2, 3) should return 6
```

```
  4 passing (7ms)
```

这说明我们编写的 4 个测试全部通过。如果没有通过，要么修改测试代码，要么修改 hello.js，直到测试全部通过为止。

编写合约测试脚本

测试时我们通常会把每次测试运行的环境隔离开，以保证互不影响。对应到合约测试，我们每次测试都需要部署新的合约实例，然后针对新的实例做功能测试。 Car 合约的功能比较简单，我们只要设计 2 个测试用例：

- 合约部署时传入的 brand 属性被正确存储；
- 调用 setBrand 之后合约的 brand 属性被正确更新；

新建测试文件 tests/car.spec.js，完整的测试代码如下。

```
const path = require('path');
const assert = require('assert');
const ganache = require('ganache-cli');
const Web3 = require('web3');

// 1. 配置 provider
const web3 = new Web3(ganache.provider());

// 2. 拿到 abi 和 bytecode
const contractPath = path.resolve(__dirname,
                            '../compiled/Car.json');
const { interface, bytecode } = require(contractPath);

let accounts;
let contract;
const initialBrand = 'BMW';
```

```
describe('contract', () => {
    // 3. 每次跑单测时需要部署全新的合约实例，起到隔离的作用
    beforeEach(async () => {
        accounts = await web3.eth.getAccounts();
        console.log('合约部署账户：', accounts[0]);
        contract = await new
            web3.eth.Contract(JSON.parse(interface))
            .deploy({ data: bytecode, arguments: [initialBrand] })
            .send({ from: accounts[0], gas: '1000000' });
        console.log('合约部署成功：',
                        contract.options.address); });
    // 4. 编写单元测试
    it('deployed contract', () => {
        assert.ok(contract.options.address);
    });
    it('should has initial brand', async () => {
        const brand = await contract.methods.brand().call();
        assert.equal(brand, initialBrand);
    });
    it('can change the brand', async ()=>{
        const newBrand = 'Benz';
        await contract.methods.setBrand(newBrand)
                                .send({from: accounts[0]});
        const brand = await contract.methods.brand().call();
        assert.equal(brand, newBrand);
    });
});
```

整个测试代码使用的断言库是 Node.js 内置的 assert 模块，assert.ok()用于判断表达式真值，等同于 assert()，如果为 false 则抛出 error；assert.equal()用于判断实际值和期望值是否相等（==），如果不相等则抛出 error。

beforeEach 是 mocha 里提供的声明周期方法，表示每次运行时每个 test 执行前都要做的准备操作。因为我们知道，在测试前初始化资源，测试后释放资源是非常常见的，所以 mocha 提供了 before、after、beforeEach 和 afterEach 来实现这些功能。

测试的关键步骤也用编号的数字做了注释，其中步骤 1、2、3 在合约部署脚本中已经比较熟悉，需要注意的是 ganache-cli provider 的创建方式。我们在脚本中引入 ganache，将模拟以太坊节点嵌入测试中，就不会影响我们外部运行的节点环境了。

测试中我们用到了 web3.js 中两个与合约实例交互的方法，之前我们已经接触过，以后在 DApp 开发时会大量使用：

- contract.methods.brand().call()，调用合约上的方法，通常是取数据，立即返回，与 v0.20.1 版本中的 .call() 相同；
- contract.methods.setBrand('xxx').send()，对合约发起交易，通常是修改数据，返回的是交易 Hash，相当于 v0.20.1 中的 sendTransaction()；send 必须指定发起的账户地址，而 call 可以直接调用。

注意在 v1.0.0 中，contract 后面要加上.methods 然后才能跟合约函数名，这与 v0.20.1 不同；类似，v1.0.0 中事件的监听也要 contract 后面加.events。

运行测试脚本

有了测试代码，就可以运行并观察结果。mocha 默认会执行 test 目录下的所有脚本，但我们也可以传入脚本路径，指定执行目录。如果你环境中全局安装了 mocha，可以使用如下命令运行测试：

```
mocha tests
```

如果没有全局安装 mocha，就使用如下命令运行测试：

```
./node_modules/.bin/mocha tests
```

如果一切正常，我们可以看到这样的输出结果：

```
  contract
合约部署账户：  0x1dD5C293Daf399Df299A7896Ce618142cAd0378f
(node:354) MaxListenersExceededWarning: Possible EventEmitter memory
ners added. Use emitter.setMaxListeners() to increase limit
合约部署成功：  0x4c3a244d8529927aD44c8707b302B30671DB2473
    ✓ deployed contract
合约部署账户：  0x1dD5C293Daf399Df299A7896Ce618142cAd0378f
合约部署成功：  0x222246dF0990178391c9B0CC4cF2D86E34E23B42
    ✓ should has initial brand
合约部署账户：  0x1dD5C293Daf399Df299A7896Ce618142cAd0378f
合约部署成功：  0xf559425569829a606123f27aa27AA1AC61B1d1ab
    ✓ can change the brand (117ms)


  3 passing (748ms)
```

完整的工作流

到目前为止，我们已经熟悉了智能合约的开发、编译、部署、测试，而在实际工作中，把这些过程串起来才能算作是真正意义上的工作流。比如修改了合约代码需要重新运行测试，但是重新运行测试之前需要重新编译，而部署的过程也是类似的，每次部署的都要是最新的合约代码。

通过 npm script 机制，我们可以把智能合约的工作流串起来，让能自动化的尽可能自动化，在 package.json 中作如下修改：

```
"scripts": {
      "compile": "node scripts/compile.js",
      "pretest": "npm run compile",
      "test": "mocha tests/",
      "predeploy": "npm run compile",
      "deploy": "node scripts/deploy.js"
},
```

上面的改动中，我们为项目增加了 3 条命令：compile、test、deploy，其中 pretest、predeploy 是利用了 npm script 的生命周期机制，把我们的 compile、test、deploy 串起来。

接下来我们可以使用 npm run test 运行测试，结果如下：

```
> contract_workflow@1.0.0 pretest /home/ubuntu/project/workflow_test
> npm run compile


> contract_workflow@1.0.0 compile /home/ubuntu/project/workflow_test
> node scripts/compile.js

Saving json file to  /home/ubuntu/project/workflow_test/compiled/Car.json

> contract_workflow@1.0.0 test /home/ubuntu/project/workflow_test
> ./node_modules/mocha/bin/mocha tests/



  contract
合约部署账户：  0xF32E39c8b69999a4305Dd878bA56e0c6E29b0ef0
(node:423) MaxListenersExceededWarning: Possible EventEmitter memory leak
ners added. Use emitter.setMaxListeners() to increase limit
合约部署成功：  0xA8bC40d15f863627D7ae21ccF384A7F6FF73e645
    ✓ deployed contract
合约部署账户：  0xF32E39c8b69999a4305Dd878bA56e0c6E29b0ef0
合约部署成功：  0x2bF2b282C02b4f59132f52f9c3BC3f43973Afff9
    ✓ should has initial brand (38ms)
合约部署账户：  0xF32E39c8b69999a4305Dd878bA56e0c6E29b0ef0
合约部署成功：  0xBa792454BBe4892d7668F8C027dC79C2d07a2836
    ✓ can change the brand (155ms)


  3 passing (860ms)
```

同理我们可以使用 npm run deploy 部署合约，结果如下：

```
> contract_workflow@1.0.0 predeploy /home/ubuntu/project/workflow_test
> npm run compile


> contract_workflow@1.0.0 compile /home/ubuntu/project/workflow_test
> node scripts/compile.js

Saving json file to  /home/ubuntu/project/workflow_test/compiled/Car.json

> contract_workflow@1.0.0 deploy /home/ubuntu/project/workflow_test
> node scripts/deploy.js

deploy time: 177.083ms
contract address:  0x00aC688114723873766aa7D9903750b11d31ae89
```