自动化测试

1. 课前准备

1. 了解自动化测试
2. jest
3. cypress

2. 课堂主题

1. 单测
2. E2E测试

3. 课堂目标

1. 掌握Vue测试
2. 写易于测试的Vue组件和代码

4. 知识点

为什么需要测试

为什么要进行测试

1. 测试可以确保得到预期的结果
2. 作为现有代码行为的描述
3. 促使开发者写可测试的代码，一般可测试的代码可读性也会高一点
4. 如果依赖的组件有修改，受影响的组件能在测试中发现错误

测试分类

单元测试：指的是以原件的单元为单位，对软件进行测试。单元可以是一个函数，也可以是一个模块或一个组件，基本特征就是只要输入不变，必定返回同样的输出。一个软件越容易些单元测试，就表明它的模块化结构越好，给模块之间的耦合越弱。React的组件化和函数式编程，天生适合进行单元测试

功能测试：相当于是黑盒测试，测试者不了解程序的内部情况，不需要具备编程语言的专门知识，只知道程序的输入、输出和功能，从用户的角度针对软件界面、功能和外部结构进行测试，不考虑内部的逻辑

集成测试：在单元测试的基础上，将所有模块按照设计要求组装成子系统或者系统，进行测试

冒烟测试：在正式全面的测试之前，对主要功能进行的与测试，确认主要功能是否满足需要，软件是否能正常运行

我们其实日常使用console，算是测试的雏形吧，console.log(add(1,2) == 3)

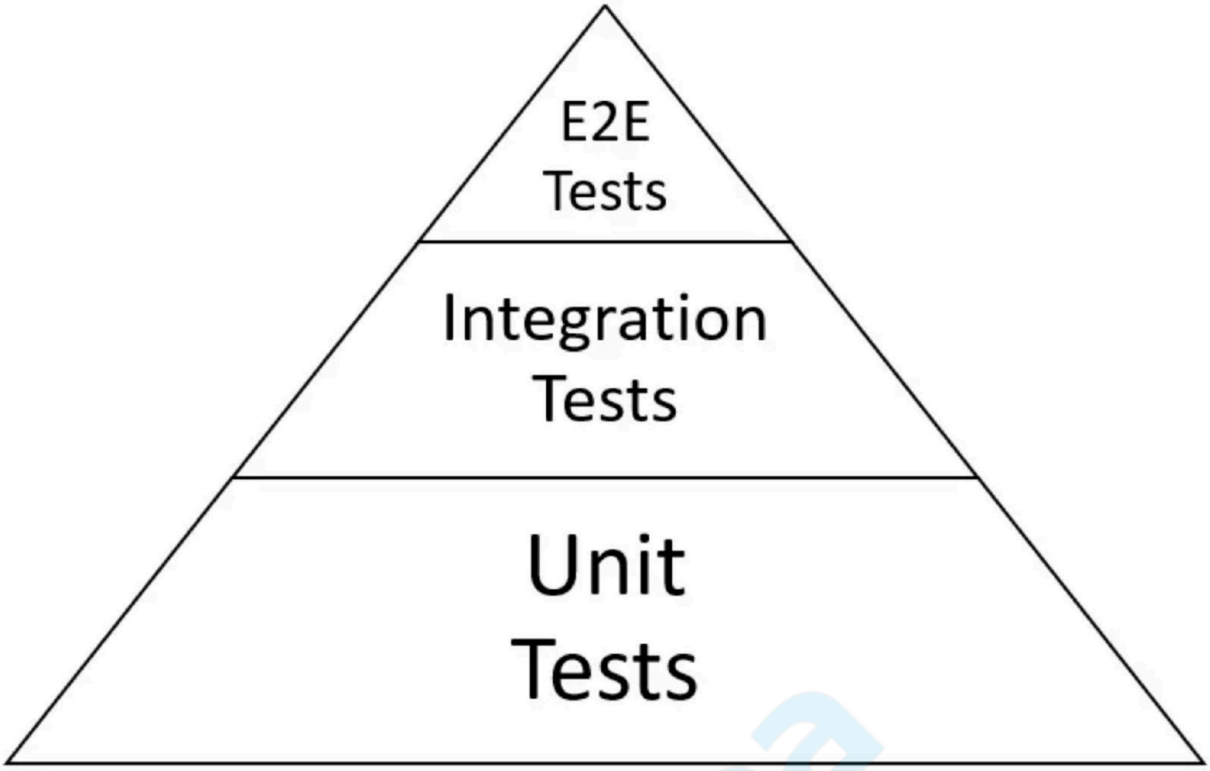
测试的好处

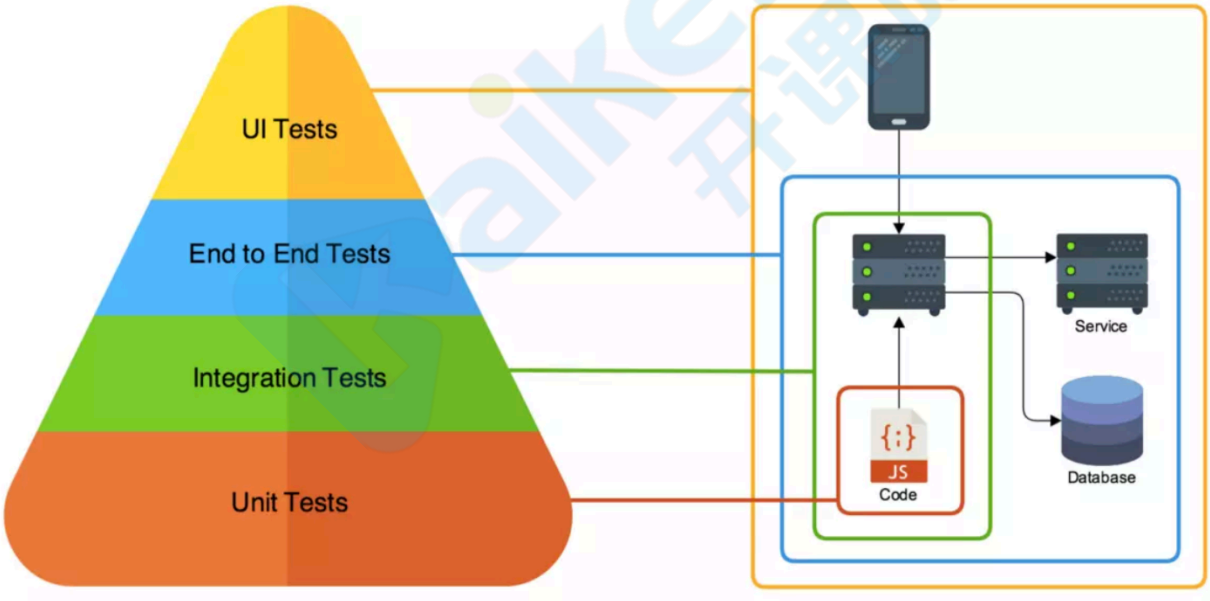
组件的单元测试有很多好处：

- 提供描述组件行为的文档
- 节省手动测试的时间
- 减少研发新特性时产生的 bug
- 改进设计
- 促进重构

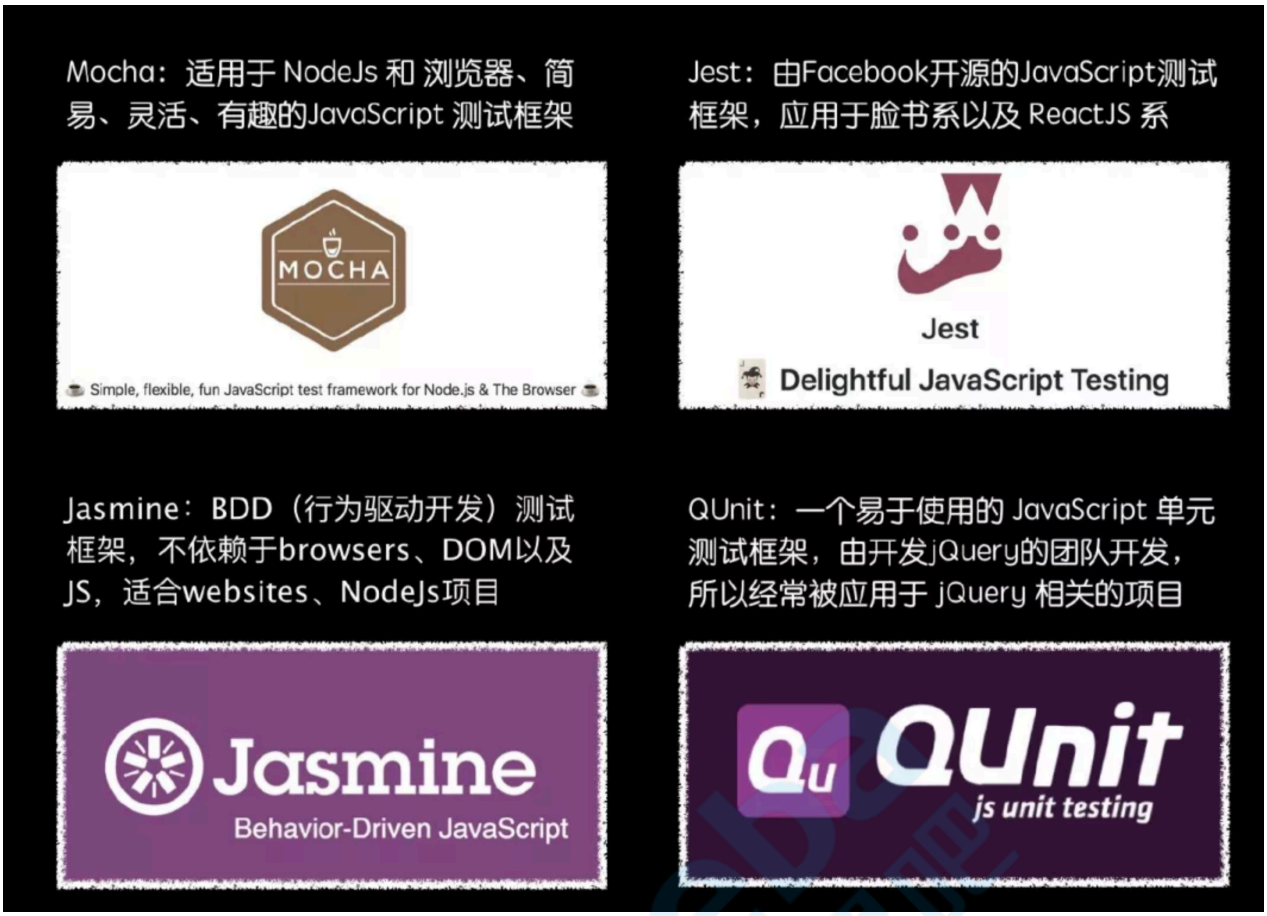
自动化测试使得大团队中的开发者可以维护复杂的基础代码。让你改代码不再小心翼翼

<<<<<<< HEAD


E2E
Tests
Integration
Tests
Unit
Tests


UI Tests
End to End Tests
Integration Tests
Unit Tests
Service
Database
JS
Code

测试工具



490726f153e030b5873a87dd451176953e055c80

单测

> 单元测试（unit testing），是指对软件中的最小可测试单元进行检查和验证。

在vue中，推荐用Mocha+chai 或者jest，咱们使用jest演示，语法基本一致

新建kaikeba.spec.js，.spec.js是命名规范，写下一下代码

```
function add(num1, num2) {
    return num1 + num2
}

describe('Kaikeba', () => {
    it('测试加法', () => {
        expect(add(1, 3)).toBe(3)
        expect(add(1, 3)).toBe(4)
        expect(add(-2, 3)).toBe(1)
    })
})
```

执行 npm run test:unit

api介绍

- `describe`： 定义一个测试套件
- `it`：定义一个测试用例
- `expect`：断言的判断条件
- `toBe`：断言的比较结果

测试Vue组件

一个简单的组件

```
<template>
    <div>
    <span>{{ message }}</span>
    <button @click="changeMsg">点击</button>
    </div>

</template>

<script>
  export default {
    data () {
      return {
```

```
        message: 'vue-text'
      }
    },
    created () {
      this.message = '开课吧'
    },
    methods:{
        changeMsg(){
            this.message = '按钮点击'
        }
    }
  }
</script>
```

```
// 导入 Vue.js 和组件，进行测试
import Vue from 'vue'
import KaikebaComp from '@/components/Kaikeba.vue'

// 这里是一些 Jasmine 2.0 的测试，你也可以使用你喜欢的任何断言库或测试工具。

describe('KaikebaComp', () => {
  // 检查原始组件选项
  it('由created生命周期', () => {
    expect(typeof KaikebaComp.created).toBe('function')
  })

  // 评估原始组件选项中的函数的结果
  it('初始data是vue-text', () => {
    expect(typeof KaikebaComp.data).toBe('function')

    const defaultData = KaikebaComp.data()
    expect(defaultData.message).toBe('hello!')
  })

})
```

检查mounted之后

```
it('mount之后测data是开课吧', () => {
  const vm = new Vue(KaikebaComp).$mount()
  expect(vm.message).toBe('开课吧')
})
```

用户点击

和写vue 没啥本质区别，只不过我们用测试的角度去写代码，vue提供了专门针对测试的`@vue/test-utils`

```
it('按钮点击后', () => {
  const wrapper = mount(KaikebaComp)
  wrapper.find('button').trigger('click')
  expect(wrapper.vm.message).toBe('按钮点击')
  // 测试html渲染结果
  expect(wrapper.find('span').html()).toBe('<span>按钮点击</span>')
})
```

测试覆盖率

jest自带覆盖率，如果用的mocha，需要使用istanbul来统计覆盖率

package.json里修改jest配置

```
  "jest": {
    "collectCoverage": true,
    "collectCoverageFrom": ["src/**/*.{js,vue}"],
  }
```

在此执行npm run test:unit

```
> vue-cli-service test:unit

 PASS  tests/unit/kaikeba.spec.js
 PASS  tests/unit/example.spec.js
----------------|----------|----------|----------|----------|-------------------|
File            |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------------|----------|----------|----------|----------|-------------------|
All files       |    15.79 |      100 |        0 |    15.79 |                   |
 src            |        0 |      100 |        0 |        0 |                   |
  main.js       |        0 |      100 |        0 |        0 |   1,2,3,4,6,8,11  |
  router.js     |        0 |      100 |        0 |        0 |      1,2,3,5,22   |
  store.js      |        0 |      100 |      100 |        0 |          1,2,4    |
 src/components |      100 |      100 |      100 |      100 |                   |
  Kaikeba.vue   |      100 |      100 |      100 |      100 |                   |
 src/views      |        0 |      100 |      100 |        0 |                   |
  Home.vue      |        0 |      100 |      100 |        0 |               10  |
----------------|----------|----------|----------|----------|-------------------|

Test Suites: 2 passed, 2 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        1.653s
Ran all test suites.
➜  vue-test git:(master) x
```

可以看到我们kaikeba.vue的覆盖率是100%，我们修改一下代码

```
<template>
  <div>
    <span>{{ message }}</span>
    <button @click="changeMsg">点击</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: "vue-text",
      count: 0
    };
```

```
  },
  created() {
    this.message = "开课吧";
  },
  methods: {
    changeMsg() {
      if (this.count > 1) {
        this.message = "count大于1";
      } else {
        this.message = "按钮点击";
      }
    },
    changeCount() {
      this.count += 1;
    }
  }
};
</script>
```

```
PASS tests/unit/kaikeba.spec.js
PASS tests/unit/example.spec.js
----------------|----------|----------|----------|----------|-------------------|
File            |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------------|----------|----------|----------|----------|-------------------|
All files       |    18.18 |       50 |        0 |    18.18 |                   |
 src            |        0 |      100 |        0 |        0 |                   |
  main.js       |        0 |      100 |        0 |        0 |    1,2,3,4,6,8,11 |
  router.js     |        0 |      100 |        0 |        0 |      1,2,3,5,22   |
  store.js      |        0 |      100 |      100 |        0 |            1,2,4  |
 src/components |    66.67 |       50 |      100 |    66.67 |                   |
  Kaikeba.vue   |    66.67 |       50 |      100 |    66.67 |            22,28  |
 src/views      |        0 |      100 |      100 |        0 |                   |
  Home.vue      |        0 |      100 |      100 |        0 |              10   |
----------------|----------|----------|----------|----------|-------------------|

Test Suites: 2 passed, 2 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        2.08s
Ran all test suites
```

现在的代码，依然是测试没有报错，但是覆盖率只有66%了，而且没有覆盖的代码行数，都标记了出来，继续努力加测试吧

Jest详解

api

```
beforeAll(() => {
  console.log('global before all');
});

afterAll(() => {
  console.log('global after all');
});

beforeEach(() =>{
  console.log('global before each');
});

afterEach(() => {
  console.log('global after each');
});

describe('test1', () => {
  beforeAll(() => {
    console.log('test1 before all');
  });

  afterAll(() => {
    console.log('test1 after all');
  });

  beforeEach(() => {
    console.log('test1 before each');
  });

  afterEach(() => {
    console.log('test1 after each');
  });

  it('test sum', () => {
    expect(sum(2, 3)).toEqual(5);
  });

  it('test mutil', () => {
    expect(sum(2, 3)).toEqual(7);
  });

});
```

断言

1. expect(value)：要测试一个值进行断言的时候，要使用expect对值进行包裹
2. toBe(value)：使用Object.is来进行比较，如果进行浮点数的比较，要使用toBeCloseTo
3. not：用来取反
4. oEqual(value)：用于对象的深比较
5. toMatch(regexpOrString)：用来检查字符串是否匹配，可以是正则表达式或者字符串
6. toContain(item)：用来判断item是否在一个数组中，也可以用于字符串的判断
7. toBeNull(value)：只匹配null
8. toBeUndefined(value)：只匹配undefined
9. toBeDefined(value)：与toBeUndefined相反
10. toBeTruthy(value)：匹配任何使if语句为真的值
11. toBeFalsy(value)：匹配任何使if语句为假的值
12. toBeGreaterThan(number)： 大于
13. toBeGreaterThanOrEqual(number)：大于等于
14. toBeLessThan(number)：小于
15. toBeLessThanOrEqual(number)：小于等于
16. toBeInstanceOf(class)：判断是不是class的实例
17. anything(value)：匹配除了null和undefined以外的所有值
18. resolves：用来取出promise为fulfilled时包裹的值，支持链式调用
19. rejects：用来取出promise为rejected时包裹的值，支持链式调用
20. toHaveBeenCalled()：用来判断mock function是否被调用过
21. toHaveBeenCalledTimes(number)：用来判断mock function被调用的次数
22. assertions(number)：验证在一个测试用例中有number个断言被调用
23. extend(matchers)：自定义一些断言

方法

1. simulate(event, mock)：模拟事件，用来触发事件，event为事件名称，mock为一个event object
2. instance()：返回组件的实例
3. find(selector)：根据选择器查找节点，selector可以是CSS中的选择器，或者是组件的构造函数，组件的display name等
4. at(index)：返回一个渲染过的对象
5. get(index)：返回一个react node，要测试它，需要重新渲染
6. contains(nodeOrNodes)：当前对象是否包含参数重点 node，参数类型为react对象或对象数组
7. text()：返回当前组件的文本内容
8. html()： 返回当前组件的HTML代码形式
9. props()：返回根组件的所有属性
10. prop(key)：返回根组件的指定属性

E2E测试

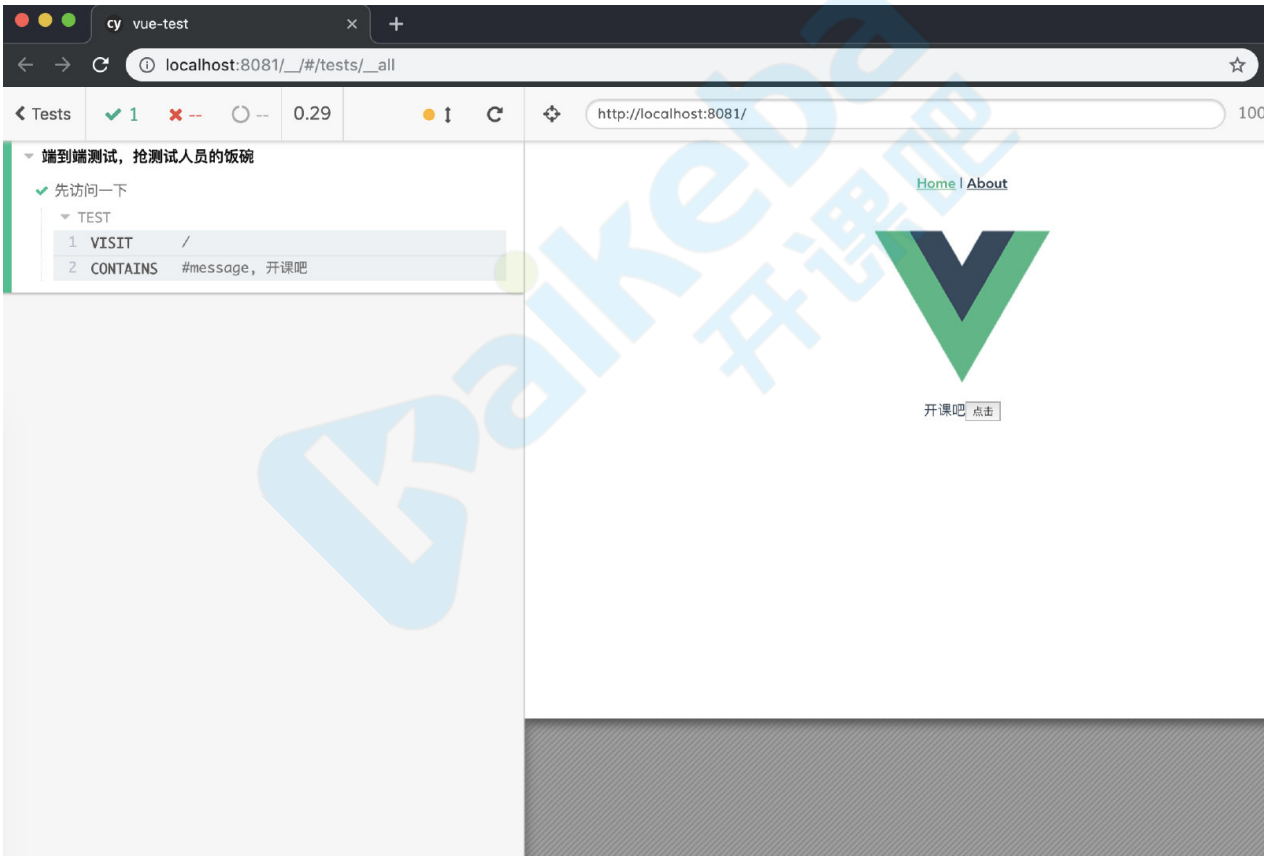
借用浏览器的能力，站在用户测试人员的角度，输入框，点击按钮等，完全模拟用户，这个和具体的框架关系不大，完全模拟浏览器行为

修改e2e/spec/test.js

```
// https://docs.cypress.io/api/introduction/api.html

describe('端到端测试，抢测试人员的饭碗', () => {
  it('先访问一下', () => {
    cy.visit('/')
    // cy.contains('h1', 'Welcome to Your Vue.js App')
    cy.contains('#message', '开课吧')

  })
})
```



可以看到是打开了一个浏览器进行测试

测试用户点击

```
// https://docs.cypress.io/api/introduction/api.html

describe('端到端测试，抢测试人员的饭碗', () => {
  it('先访问一下', () => {
```

```
    cy.visit('/')
    // cy.contains('h1', 'Welcome to Your Vue.js App')
    cy.contains('#message', '开课吧')

    cy.get('button').click()
    cy.contains('#message', '按钮点击')

  })
})
```

TDD

所以TDD 就是测试驱动开发模式，就是我们开发一个新功能，先把测试写好，然后测试跑起来，会报错，我们再开始写代码，挨个的把测试跑绿，功能也就完成了

React 自动化测试

React中，也是使用jest来做自动化测试，我们来体验一下

https://jestjs.io/docs/en/tutorial-react

Node自动化测试

node中单测，除了类似vue中的输入输出测试，node很多都是网络家口数据，我们如何是去测试这些数据呢

测试koa （回顾我们自己写的koa源码）

5. 扩展

6. 总结

7. 作业

8. 问答

9. 预告