TypeScript && 测试

课前预习材料

1. TypeScript参考
2. vue中的TypeScript
3. 单元测试
4. 端到端测试参考

课堂目标

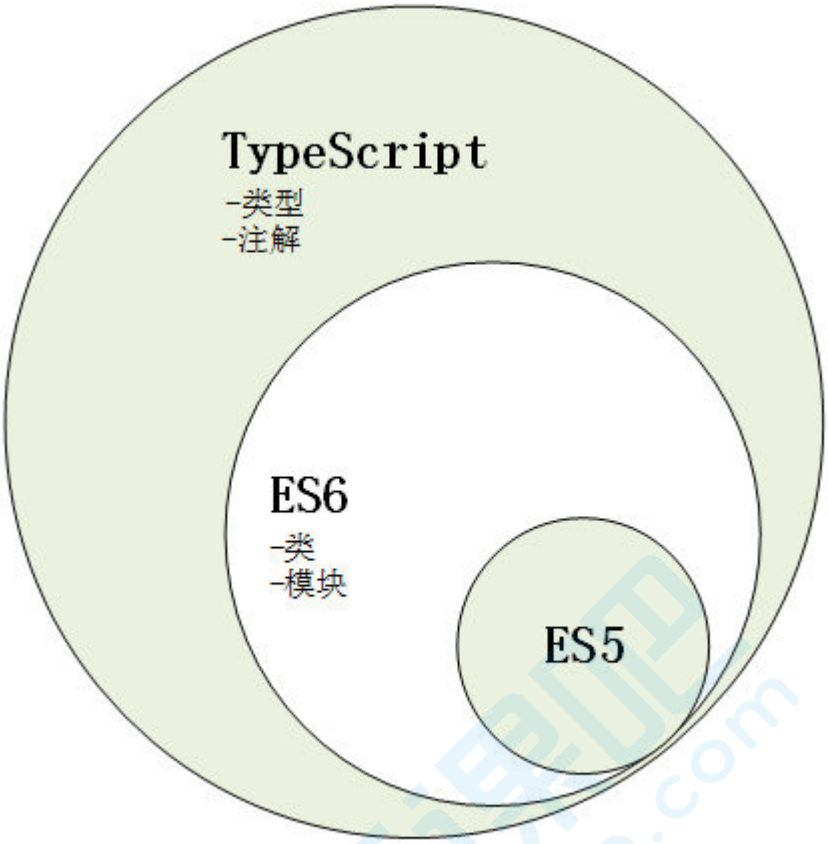
1. 掌握TypeScript
2. 能使用TypeScript编写vue应用
3. 掌握Vue测试
4. 写易于测试的Vue组件和代码

知识点

TypeScript

- 类型注解和编译时类型检查
- 基于类的面向对象编程
- 泛型
- 接口
- 声明文件

ts和es6



ES5、ES6和TypeScript

> typescript是angular2的开发语言，Vue3正在使用TS重写

类型注解和编译时类型检查

定义变量后，可以通过冒号来指定类型注解

```
// Hello.vue
let name = "xx"; // 类型推论
let title: string = "开课吧"; // 类型注解
name = 2;// 错误
title = 4;// 错误
```

数组类型

```
let names: string[];
names = ['Tom'];//或Array<string>
```

任意类型

```
let foo:any = 'xx'
foo = 3

//any类型也可用于数组
let list: any[] = [1, true, "free"];
list[1] = 100;
```

函数中使用类型

```
function greeting(person: string): string {
  return 'Hello, ' + person;
}
//void类型，常用于没有返回值的函数
function warnUser(): void { alert("This is my warning message"); }
```

函数

必填参：参数一旦声明，就要求传递，且类型需符合

```
function sayHello(name:string, age:number): string {
  console.log(name, age)
}
sayHello(11,12)
satHello('开课吧','十八')
```

可选参数：参数名后面加上问号，变成可选参数

```
function sayHello(name:string, age?:number): string {
  console.log(name, age)
}
```

参数默认值

```
function sayHello(name:string, age:number=20): string {
  console.log(name, age)
}
```

*函数重载：以参数数量或类型区分多个同名函数

```
function add(a: number, b: number): string;
function add(a: string, b: string): string;
function add(a: any, b: any): string {
  if (typeof a === "number") {
    return "数字相加: " + (a + b);
  } else {
    return "字符串拼接: " + (a + b);
  }
}
console.log(add(1,1));
console.log(add('foo','bar'));
```

类

自定义类型：通过类可以声明自定义类型，Hello.vue

面向对象：通过extends实现继承、使用public等访问修饰符实现封装、通过方法覆盖实现多态

接口

接口仅约束结构，不要求实现，使用更简单

```
// 接口中只需定义结构，不需要初始化
interface Feature {
  id: number;
  name: string;
  version: string;
}
```

泛型

可使用泛型（Generic）来创建可重用的组件，一个组件可以支持多种类型的数据

```
// 不用泛型
// interface Result {
//   ok: 0 | 1;
//   data: Feature;
// }

// 使用泛型
interface Result<T> {
  ok: 0 | 1;
  data: T[];
}
```

装饰器

装饰器实际上是工厂函数，传入一个对象，输出处理后的新对象。

测试分类

常见的开发流程里，都有测试人员，这种我们成为黑盒测试，测试人员不管内部实现机制，只看最外层的输入输出，比如你写一个加法的页面，会设计N个case，测试加法的正确性，这种代码里，我们称之为E2E测试

更负责一些的 我们称之为集成测试，就是集合多个测试过的单元一起测试

还有一种测试叫做白盒测试，我们针对一些内部机制的核心逻辑 使用代码 进行编写 我们称之为单元测试

这仨都是我们前端开发人员进阶必备的技能

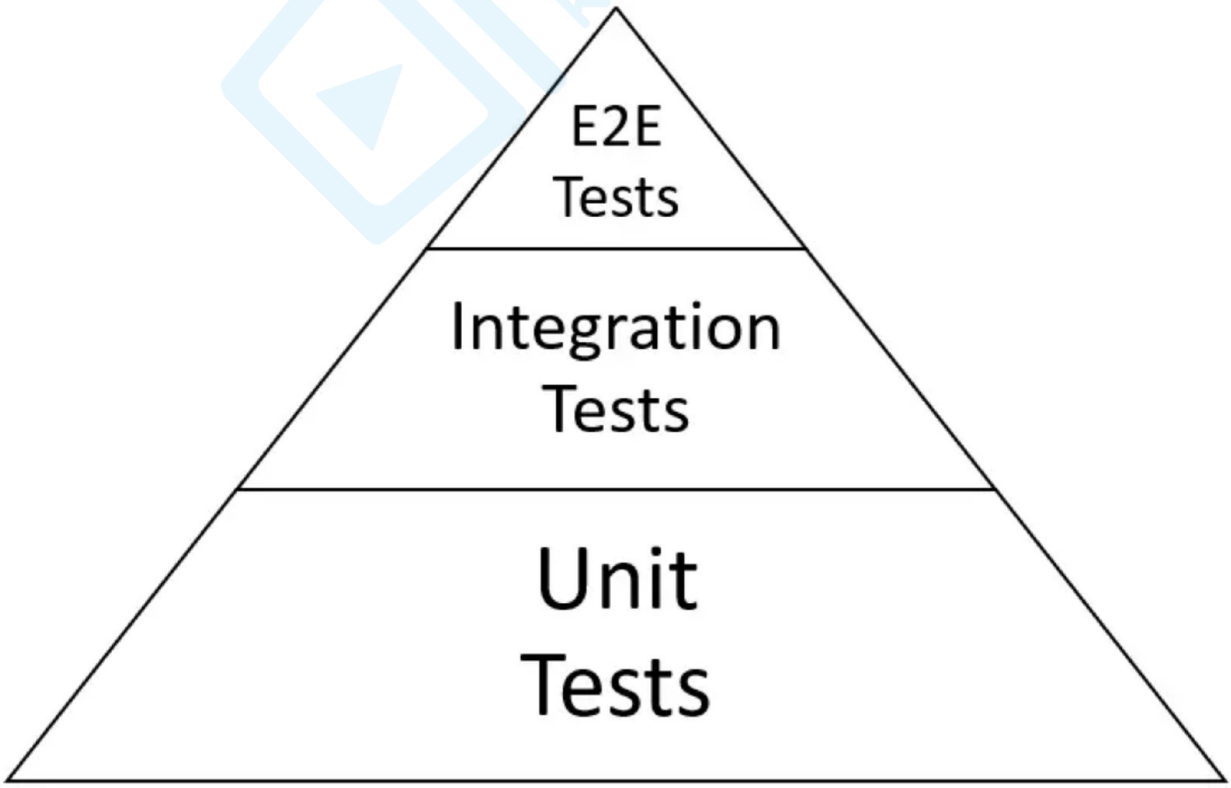
我们其实日常使用console，算是测试的雏形吧，console.log(add(1,2) == 3)

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



单测

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

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>
```

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

E2E测试

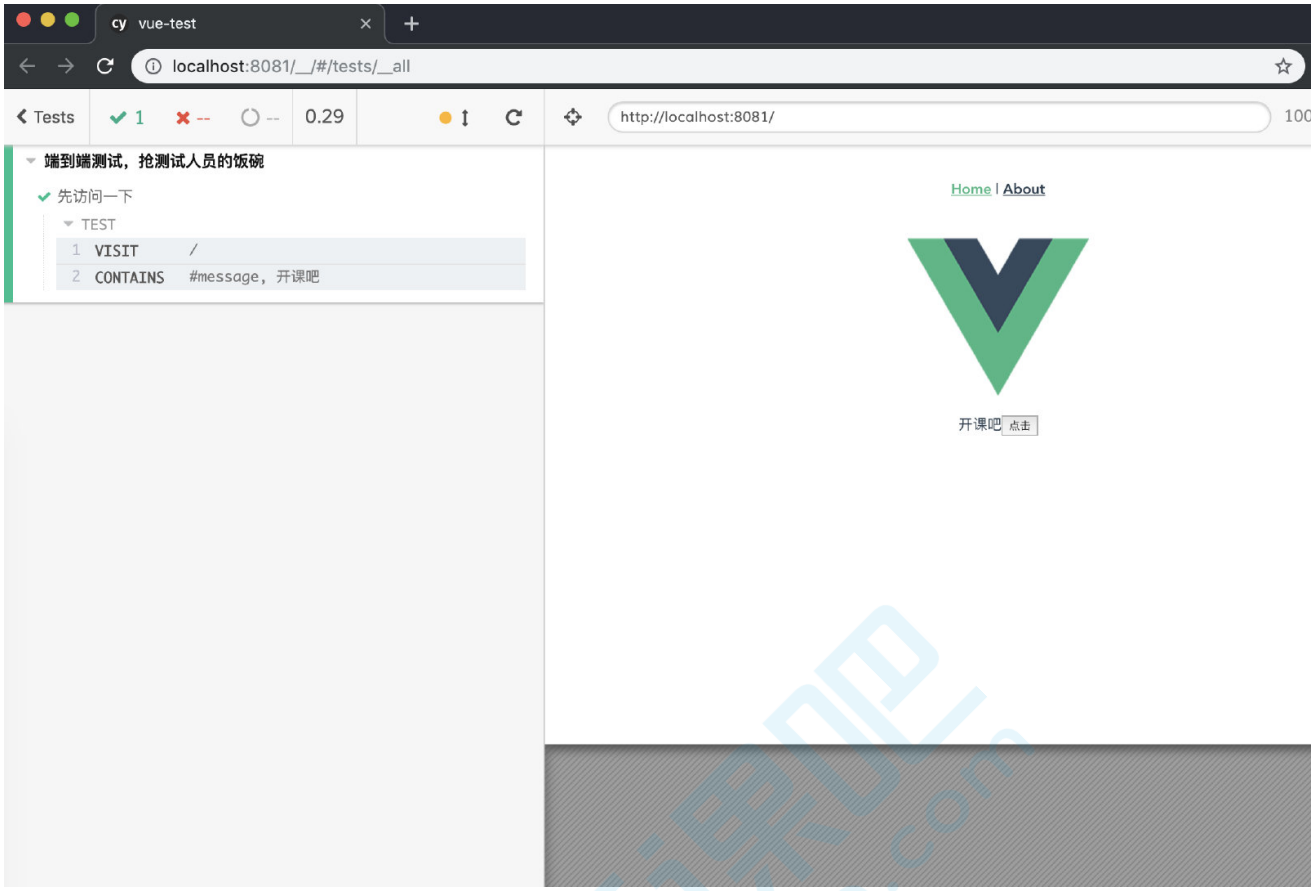
借用浏览器的能力，站在用户测试人员的角度，输入框，点击按钮等，完全模拟用户，这个和具体的框架关系不大，完全模拟浏览器行为

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')

  })
})
```