结课作业提交仓库

姓名：刘承杰
学校：南京大学
专业：软件工程
年级：2020级
QQ：329657173

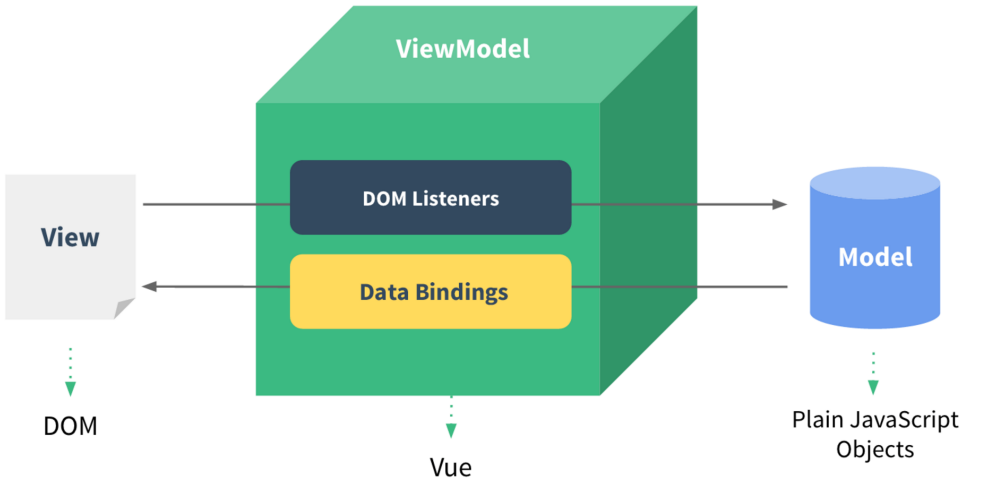
- 提交方式
  fork 本仓库提交你的代码即可

结课作业提交仓库
- 一、项目背景
- 二、项目目录
- 三、功能演示
- 四、具体实现
  - MVVM类
  - Compile类
  - Observer类
  - 发布订阅类Dep
  - Watcher类
- 五、测试
  - 1. 测试代码
  - 2. 测试结果

一、项目背景

MVVM 是一种软件架构模式，是 MVC 的改进版，MVVM 将其中 View 的状态和行为抽象化，让我们将视图的 UI和业务上的逻辑进行分开。简单来说，MVVM 是 Model-View-ViewModel 的简写。即是模型-视图-视图模型。

MVVM 框架主要完成了模板和数据之间的双向绑定。使用数据劫持的相关技术，对数据的变化的进行追踪，同时引入发布订阅模式完成对属性的依赖管理。另一方面则涉及模板的解析，提取模板中的指令和表达式，关联到数据，来初始化视图。模板和数据通过 watcher 衔接起来，这样数据的变化直接映射到视图；同时需要监听视图上的输入相关的事件，这样当输入变化时直接映射到数据模型。



本项目则实现上述 MVVM 框架的功能：

1. 实现数据劫持
2. 实现发布订阅模式
3. 实现数据单向绑定
4. 实现数据双向绑定
5. 项目实现
    - 使用 TypeScript 开发
    - 有 README
    - 有单元测试，且单元测试覆盖率达到 80%（本项目实际单测覆盖率为 97%）

二、项目目录

```
.
├── MVVM.html //项目html文件
├── MVVM框架要求.png
├── README.md //本文件
├── README.pdf //本文件pdf版本
├── __tests__ //测试文件
│   └── allInOne.test.ts
├── coverage
├── node_module
├── jest.config.js
├── out //ts导出的js文件
│   ├── MVVM.js
│   ├── compile.js
│   ├── observer.js
│   └── watcher.js
├── package-lock.json
├── package.json
├── src //ts源文件
│   ├── MVVM.ts
│   ├── compile.ts
│   ├── observer.ts
```

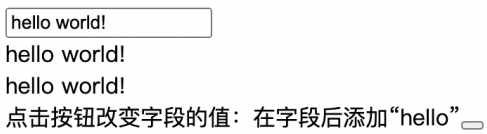
```
22 |   └── watcher.ts
23 └── tsconfig.json
```

三、功能演示

HTML源码如下

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>MVVM</title>
8  </head>
9  <body>
10     <div id = "app">
11         <input type = "text" v-model = "message">
12         <div>{{message}}</div>
13         {{message}}
14     </div>
15     点击按钮改变字段的值：在字段后添加“hello”<button id="btn1" onclick="changeMVVM()">
   </button><br>
16 </body>
17 </html>
18
19 <script src = "out/watcher.js"></script>
20 <script src = "out/observer.js"></script>
21 <script src = "out/compile.js"></script>
22 <script src = "out/MVVM.js"></script>
23 <script>
24     let vm = new MVVM({
25         el: '#app',
26         data: {
27             message: 'hello world!'
28         }
29     });
30     function changeMVVM () {
31         vm.$data.message += "hello";
32     }
33 </script>
```

初始进入该文件时显示如下，输入框和 message 字段的值实现绑定



对输入框内文字进行修改时，可以发现 message 同样也发生改变

hello world!!
hello world!!
hello world!!
点击按钮改变字段的值：在字段后添加“hello”

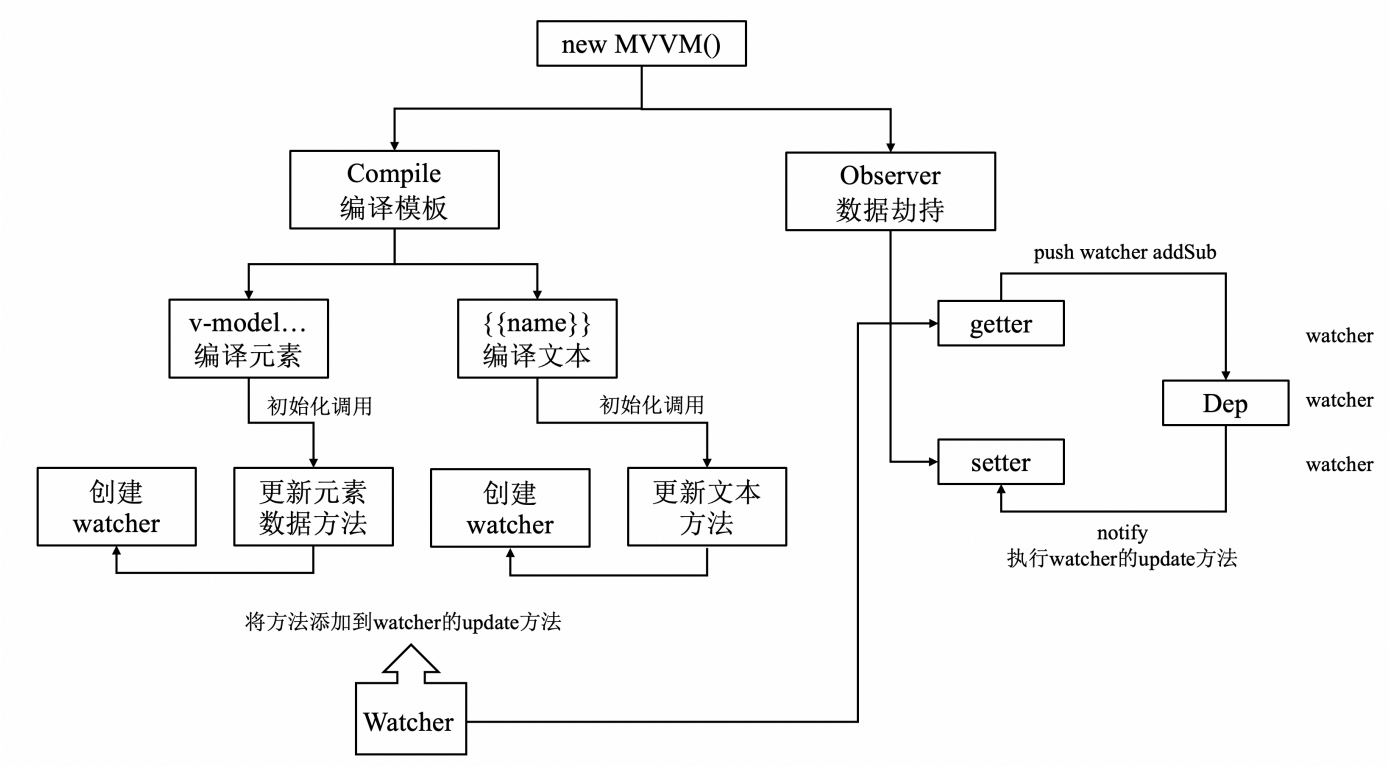
同样地，对 message 字段的值进行修改后，输入框内的值同样也被修改

hello world!!hello
hello world!!hello
hello world!!hello
点击按钮改变字段的值：在字段后添加“hello”

MVVM 架构的功能得以成功实现

四、具体实现

- MVVM 类负责创建 MVVM 实例
- Compile 类实现数据编译、单向绑定和双向绑定
- Observer 类实现数据劫持
- Dep 类和 Watcher 类共同实现发布订阅模式



MVVM类

```
import {Compile} from "./compile";
import {Observer} from "./observer";

export {MVVM}

class MVVM {
    $el;
```

```
    $data;

    constructor (options, frag) {
        // 先将可用的东西挂载在实例上，并获取模版实例
        this.$data = options.data;
        if (frag) {
            this.$el = frag;
        } else {
            this.$el = document.querySelector(options.el);
        }
        // 数据劫持，就是把对象的所有属性添加 set 和 get 方法
        new Observer(this.$data);

        // 将数据代理到实例上
        this.proxyData(this.$data);

        // 用数据和元素进行编译
        new Compile(this.$el, this);
    }

    // 数据代理
    proxyData (data) {
        Object.keys(data).forEach(key => {
            Object.defineProperty(this, key, {
                get () {
                    return data[key];
                },
                set (newVal) {
                    data[key] = newVal;
                }
            });
        });
    }
}

(window as any).MVVM = MVVM;
```

Compile类

```
export {Compile}
export {CompileUtil}

import {Watcher} from "./watcher";

class Compile {
    el;
    vm;

```

```
    constructor (el, vm) {
        // 检查el是否是DOM节点，如果不是则获取节点
        this.el = this.isElementNode(el) ? el : document.querySelector(el);
        this.vm = vm;
        if(this.el) {
            //获取文档碎片
            let fragment = this.nodeToFragment(this.el);
            //编译文档碎片
            this.compile(fragment);
            // 把编译好的文档碎片追加到页面中去
            this.el.appendChild(fragment);
        }
    }

    // 判断是否是DOM节点
    isElementNode (node): boolean {
        return node.nodeType === 1;
    }
    // 判断是不是指令
    isDirective (name): boolean {
        return name.includes('v-');
    }

    // 将DOM保存到内存中
    nodeToFragment (el) {
        const fragment = document.createDocumentFragment();
        while (el.firstChild) {
            fragment.appendChild(el.firstChild);
        }
        return fragment;
    }

    compile (fragment) {// 编译文档碎片方法
         // 获取子节点
         const childNodes = fragment.childNodes;
         // 识别节点类型并处理
         [...childNodes].forEach(child => {
             if (this.isElementNode(child)) {
                 // 元素节点处理
                 this.compileElement(child);
             } else {
                 // 文本节点处理
                 this.compileText(child)
             }
             // 如果还有子节点则需要递归调用
             if (child.childNodes && child.childNodes.length) {
                 this.compile(child)
             }
         })
```

```
    }

    compileElement (node) { // 编译元素节点
        // 带 v-model 的
        let attrs = node.attributes; // 取出当前节点的属性
        Array.from(attrs).forEach(attr => {
            // 判断属性名字是不是包含 v-
            let attrName = (attr as any).name;
            if(this.isDirective(attrName)) {
                // 取到对应的值，放在节点中
                let exp = (attr as any).value;
                let [, type] = attrName.split('-');
                // node this.vm.$date exp
                CompileUtil[type](node, this.vm, exp);
            }
        });
    }

    // 编译文本节点
    compileText (node) {
        const content = node.textContent; // 获取文本中的内容
        // 正则匹配{{**}}内容
        let reg = /\{\{([^}]+)\}\}/g;
        if(reg.test(content)) {
            // node this.vm.$date exp
            CompileUtil['text'](node, this.vm, content);
        }
    }
}

const CompileUtil = {
    // 获取实例上对应的数据
    getVal(vm, exp) {
        exp = exp.split('.');
        return exp.reduce((prev, next) => {
            return prev[next];
        }, vm.$data);
    },
    // 设置实例上对应的数据
    setVal(vm, exp, newVal) {
        exp = exp.split('.');
        return exp.reduce((prev, next, currentIndex) => {
            if (currentIndex === exp.length - 1) {
                return prev[next] = newVal;
            }
            return prev[next];
        }, vm.$data);
    },
    // 获取编译文本后的结果
```

```
    getTextVal(vm, exp) {
        return exp.replace(/\{\{([^}]+)\}\}/g, (...arg) => {
            return this.getVal(vm, arg[1]);
        });
    },
    //文本处理
    text(node, vm, exp) {
        let updateFn = this.updater['textUpdater'];
        let value = this.getTextVal(vm, exp);
        exp.replace(/\{\{([^}]+)\}\}/g, (...arg) => {
            new Watcher(vm, arg[1], newValue => {
                // 如果数据变化了，文本节点应该重新获取依赖的数据更新文本中的内容
                updateFn && updateFn(node, newValue);
            });
        });
        updateFn && updateFn(node, value);
    },
    // 输入框处理
    model(node, vm, exp) {
        let updateFn = this.updater['modelUpdater'];
        let value = this.getVal(vm, exp);
        // 添加监控，当数据变化时调用 watch 的回调
        new Watcher(vm, exp, newValue => {
            updateFn && updateFn(node, newValue);
        });
        // 添加输入框事件实现双向绑定
        node.addEventListener('input', e => {
            let newValue = e.target.value;
            this.setVal(vm, exp, newValue);
        });
        // 防止没有的指令解析时报错
        updateFn && updateFn(node, value);
    },
    updater: {
        // 文本更新
        textUpdater(node, value) {
            node.textContent = value;
        },
        // 输入框更新
        modelUpdater(node, value) {
            node.value = value;
        }
    }
};
```

Observer类

```
export {Observer}
export {Dep}

class Observer {
    constructor (data) {
        this.observe(data);
    }
    observe (data) {
        // 验证 data
        if(!data || typeof data !== 'object') {
            return;
        }

        // 要对这个 data 数据将原有的属性改成 set 和 get 的形式
        // 要将数据一一劫持，先获取到 data 的 key 和 value
        Object.keys(data).forEach(key => {
            // 劫持（实现数据响应式）
            this.defineReactive(data, key, data[key]);
            this.observe(data[key]); // 深度劫持
        });
    }
    defineReactive (object, key, value) { // 响应式
        let _this = this;
        // 每个变化的数据都会对应一个数组，这个数组是存放所有更新的操作
        let dep = new Dep();

        // 获取某个值被监听到
        Object.defineProperty(object, key, {
            enumerable: true,
            configurable: true,
            get () { // 当取值时调用的方法
                (Dep as any).target && dep.addSub((Dep as any).target);
                return value;
            },
            set (newValue) { // 当给 data 属性中设置的值适合，更改获取的属性的值
                if(newValue !== value) {
                    _this.observe(newValue); // 重新赋值如果是对象进行深度劫持
                    value = newValue;
                    dep.notify(); // 通知所有人数据更新了
                }
            }
        });
    }
}
```

发布订阅类Dep

```
class Dep {
    subs;

    constructor () {
        // 订阅的数组
        this.subs = [];
    }
    addSub (watcher) { // 添加订阅
        this.subs.push(watcher);
    }
    notify () { // 通知
        this.subs.forEach(watcher => watcher.update());
    }
}
```

Watcher类

```
import {Dep} from "./observer";
import {CompileUtil} from "./compile";

export {Watcher}

// 观察者的目的就是给需要变化的那个元素增加一个观察者，当数据变化后执行对应的方法
class Watcher {
    vm;
    exp;
    callback;
    preValue;

    constructor (vm, exp, callback) {
        this.vm = vm;
        this.exp = exp;
        this.callback = callback;
        this.preValue = this.getInitVal(); //获取初始值
    }
    getInitVal () {
       (Dep as any).target = this;
        let value = CompileUtil.getVal(this.vm, this.exp);
        (Dep as any).target = null;
        return value;
    }
    update () {
         // 更新时检查当前值是否有变化 有变化则更新数值
        let newValue = CompileUtil.getVal(this.vm, this.exp);
        if (newValue !== (this as any).oldValue){// 如果修改后的新旧值不等就执行回调
            this.callback(newValue);
```

```
        }
    }
}
```

五、测试

1. 测试代码

本项目的测试采用 ts-jest 测试框架进行单元测试，使用 jsdom 模拟 DOM 实例，具体测试代码如下

```
import {MVVM} from "../src/MVVM";

let app = document.createElement('div');
app.id = 'app';
let childNode = document.createElement('input');
childNode.setAttribute('v-model', 'message');
let text = document.createTextNode("{{message}}");
app.appendChild(childNode).appendChild(text);

let vm = new MVVM({
    data: {
        el: '#app',
        message: 'hello world!'
    }
}, app);

describe('test', () => {

    test('mvvm初始化', () => {
        expect(vm.$data.message).toBe(childNode.value);
    })

    test('修改数据模型，期望视图发生改变', () => {
        vm.$data.message = "hello mvvm";
        expect(vm.$data.message).toBe(childNode.value);
    });

    test('修改视图，期望数据模型发生改变', () => {
        childNode.value = "hello mvvm";
        childNode.dispatchEvent(new window.Event('input'));
        expect(vm.$data.message).toBe(childNode.value);
    });
})
```

2. 测试结果

下面两张图是运行上述单元测试结果，可以看到，测试用例全部通过且**单测覆盖率为 97%**，本实现符合要求。

All files

97.24% Statements `106/109` **96.29%** Branches `26/27` **95.12%** Functions `39/41` **97.22%** Lines `105/108`

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements		Branches		Functions		Lines	
MVVM.ts	85.71%	12/14	100%	1/1	60%	3/5	85.71%	12/14
compile.ts	98.33%	59/60	94.73%	18/19	100%	23/23	98.33%	59/60
observer.ts	100%	21/21	100%	6/6	100%	10/10	100%	20/20
watcher.ts	100%	14/14	100%	1/1	100%	3/3	100%	14/14