Mobx 源码解析

前言

在Git 找到Mobx 的源码(版本: 5)， 发现其是使用**TypeScript** 编写,因为我对Typescrit 没有项目经验，所以我先会将其编译成**JavaScript**，所以我们可以运行如下脚本或者从CDN直接下载一份编译过的源码，我们可以选择**umd** 规范脚本：

1. `git clone git@github.com:mobxjs/mobx.git`
2. `npm i`
3. `npm run small-build`

会在根目录下面生成两个文件夹 `.build.es5` 和 `.build.es6` 的文件夹，其分别对应的是 `TypeScript` 编译后的 `es5/6` 的脚本.

我们通过 `npm react-create-app mobx-learning` 来快速创建一个项目，然后将上面生成的 `.build.es6` 文件夹，拷贝到 `src` 目录下面.

Demo

首先我们从一个最基本的Demo开始，来看**Mobx** 的基本使用方式:
我们修改 `src/index.js` 如下：

```
import { observable, autorun } from './.build.es6/mobx'
const addBtn = document.getElementById('add')
const minusBtn = document.getElementById('minus')
const incomeLabel = document.getElementById('incomeLabel')
const nameInput = document.getElementById('name');
const bankUser = observable({
    name: 'Ivan Fan',
    income: 3,
    debit: 2
});

const incomeDisposer = autorun((reaction) => {
    incomeLabel.innerText = `${bankUser.name} income is ${bankUser.income}`
})
// incomeDisposer();
autorun(() => {
    console.log('账户存款:', bankUser.income);
});
autorun(() => {
    console.log('账户名称:', bankUser.name);
});
var nameDisposer = autorun(() => {
    console.log("name:" + bankUser.name)
});
addBtn.addEventListener('click', () => {
    bankUser.income++
})
minusBtn.addEventListener('click', () => {
    bankUser.income--
})
nameInput.addEventListener('change', (e) => {
    bankUser.name= e.target.value;
})
```

我们的界面非常简单，如图:

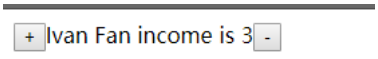


图1

两个Button , 一个label. 我们在js 文件中，我们给两个按钮添加了 **click** 事件，
事件的主体非常简单 `bankUser.income ++` `bankUser.income --` ,
就是对 `bankuser` 的 `income` 属性进行了自增或者自减，

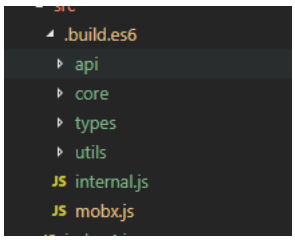
非常神奇， 当我们点击对应的按钮的时候， 中间的label 的内容发生了变化。但是我们在Button 的点击事件中并没有去操作**incomeLabel** 的内容，但是其内容确实随着点击事件，实时发生了变化。究其原因，只有以下代码对**incomeLabel** 的text 进行了处理：

```
const incomeDisposer = autorun(() => {
    incomeLabel.innerText = `Ivan Fan income is ${bankUser.income}`
})
```

这就是**Mobx** 的最简单神秘的功能，我们可以先从此开始深入研究它。

文件结构

我们打开 `.build.e6` 文件夹结构如下：



我们从上面的Demo 中引用 `mobx` 的方式 `import { observable, autorun } from './.build.es6/mobx'` 可知，
`mobx.js` 就是入口文件:

observable

从上面的JS文件中，我们发现其中引用了**mobx** 两个方法，
分别是observable 和 autorun,是的，是这样两个方法，让**incomeLabel** 在点击按钮的时候实时的发生了变化，所以我们接下来会对这两个方法进行深入分析，这一章节我们会先分析observable 先进行分析。
我们先打开Mobx的源码, 如果我们用**Vscode** 打开这个源码，我们可以用快捷键 `Ctrl + K  Ctrl + 0` 将代码都折叠起来， 然后在打开， 找到**exports** 的代码块，我们可以查看mobx 都暴露出了哪些方法：

```
exports.Reaction = Reaction$$1;
exports.untracked = untracked$$1;
exports.createAtom = createAtom$$1;
exports.spy = spy$$1;
exports.comparer = comparer$$1;
exports.isObservableObject = isObservableObject$$1;
exports.isBoxedObservable = isObservableValue$$1;
exports.isObservableArray = isObservableArray$$1;
exports.ObservableMap = ObservableMap$$1;
exports.isObservableMap = isObservableMap$$1;
exports.transaction = transaction$$1;
exports.observable = observable$$1;
exports.computed = computed$$1;
exports.isObservable = isObservable$$1;
exports.isObservableProp = isObservableProp$$1;
exports.isComputed = isComputed$$1;
exports.isComputedProp = isComputedProp$$1;
exports.extendObservable = extendObservable$$1;
exports.observe = observe$$1;
```

图2

暴露了一些列方法，我们后续会使用。
observable，翻译成中文就是可以观测的, 我们现在来调试这个方法， 我们可以 `const bankUser = mobx.observable({` 这一行打一个断点，然后 `F11` ,跳进去，发现源码对应的是一个createObservable 方法，也就是创建一个可以观察的对象：

```
var observable$$1 = createObservable;
function createObservable(v, arg2, arg3) {
    if (typeof arguments[1] === "string") {
        return deepDecorator$$1.apply(null, arguments);
    }
    if (isObservable$$1(v))
        return v;
    var res = isPlainObject$$1(v)
        ? observable$$1.object(v, arg2, arg3)
        : Array.isArray(v)
            ? observable$$1.array(v, arg2)
            : isES6Map$$1(v)
                ? observable$$1.map(v, arg2)
                : v;
    if (res !== v)
        return res;
    // otherwise, just box it
    fail$$1(process.env.NODE_ENV !== "production" &&
        "The provided value could not be converted into an observable. If you want just create an observable reference to the object use 'observable.box(value)'");
}
```

上面代码很简单，参数有三个，但是我们在调用的时候，值传递了一个参数， 所以我们暂且只要关心第一个参数 r .以下是这个functin 的基本逻辑：

1. 如果传入的第二个参数是一个字符串， 则直接调用deepDecorator$$1.apply(null, arguments);
2. 判断第一个参数是否已经是一个可观察的对象了，如果已经是可观察的对象了，就直接返回这个对象
3. 判断第一个参数是什么类型，然后调用不同的方法， 总共有三种类型: object , array , map (ES 的Map 数据类型)， 分别调用： `observable$$1.object` , `observable$$1.array` , `observable$$1.map` 方法， 那这个observable

1又是什么呢？在第一行‘*varobservable*

1 = createObservable; 表面就是`createObservable`方法。但是这个方法就短短几行代码，并没有`object, array, map`着三个方法， 我们发现在这个方法下面有**observableFa
Object.keys(observableFactories).forEach(function (name) { return (observable$$1[name] = observableFactories[name]); });`

因为在我们的Demo 中我们传递的是一个Object, 所以会调用 `observable$$1.object` 方法，接下来我们在继续分析这个方法, 其代码如下:

```
    object: function (props, decorators, options) {
        if (typeof arguments[1] === "string")
            incorrectlyUsedAsDecorator("object");
        var o = asCreateObservableOptions$$1(options);
        if (o.proxy === false) {
            return extendObservable$$1({}, props, decorators, o);
        }
        else {
            var defaultDecorator = getDefaultDecoratorFromObjectOptions$$1(o);
            var base = extendObservable$$1({}, undefined, undefined, o);
            var proxy = createDynamicObservableObject$$1(base);
            extendObservableObjectWithProperties$$1(proxy, props, decorators, defaultDecorator);
            return proxy;
        }
    },
```

`var o = asCreateObservableOptions$$1(options);` 生成的了一个简单的对象：

```
var defaultCreateObservableOptions$$1 = {
    deep: true,
    name: undefined,
    defaultDecorator: undefined,
    proxy: true
};
```

`o.proxy` 的值为 `true` , 所以会走 `else` 逻辑分支， 所以接下来我们一一分析 `else` 分支中的每一条代码。

1. `var defaultDecorator = getDefaultDecoratorFromObjectOptions$$1(o);` 这个是跟装饰器有关的逻辑，我们先跳过
2. `var base = extendObservable$$1({}, undefined, undefined, o);` 对o对象进行了加工处理，变成了一个 `Symbol` 数据类型。

> 这一步操作非常重要，给一个空对象添加了一个 `$mobx$$1` (`var $mobx$$1 = Symbol("mobx administration");`)的属性， 其值是一个 `ObservableObjectAdministration` 类型对象，其 `write` 方法在后续数据拦截中会调用。

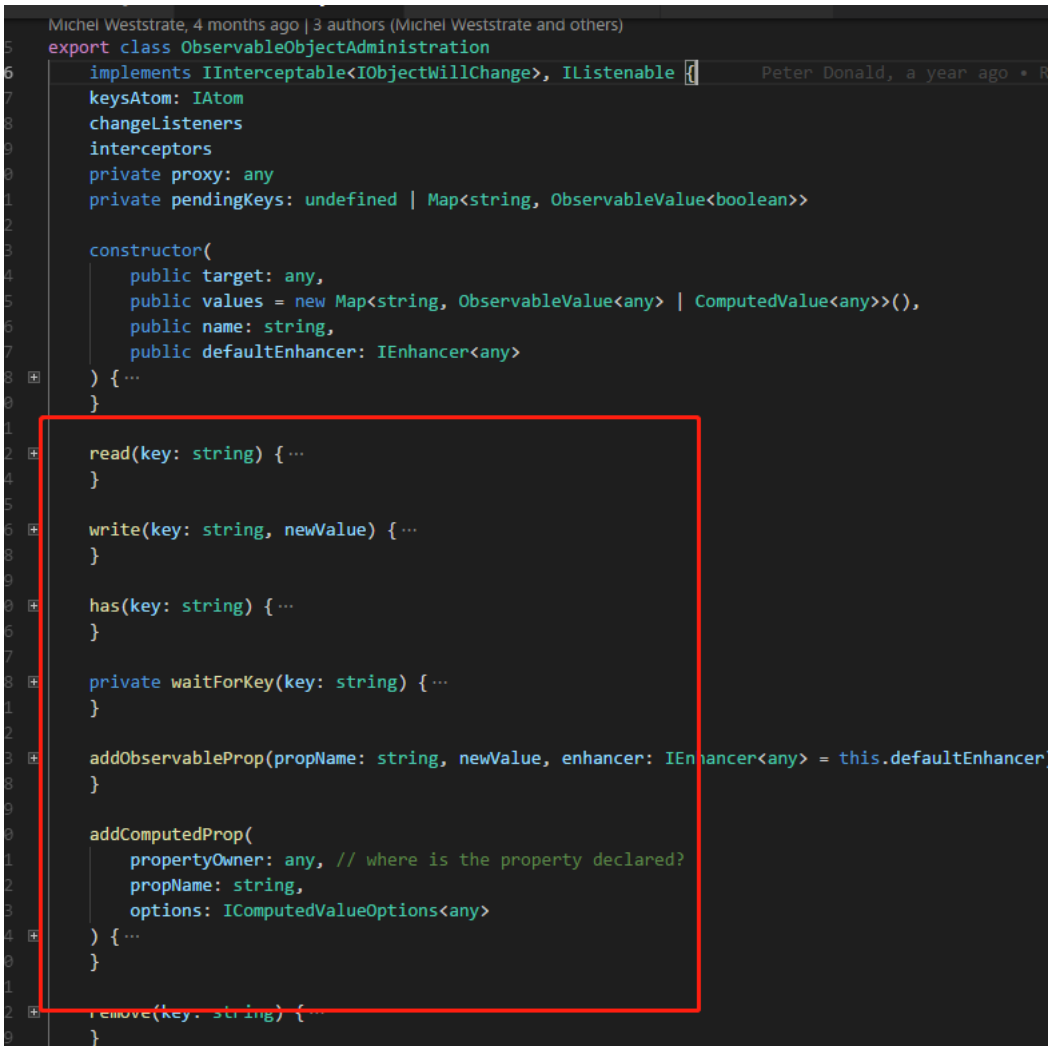


图3

3. `var proxy = createDynamicObservableObject$$1(base);` 这个方法，最为**核心**, 对这个对象进行了代理(Proxy)

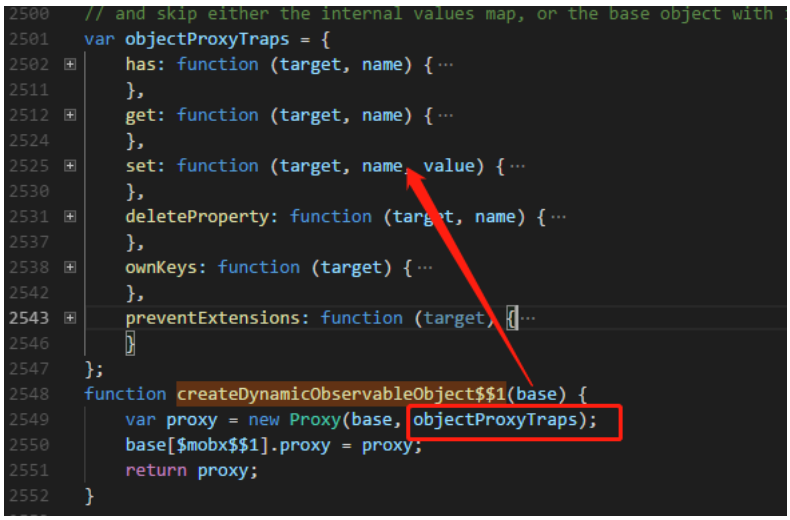
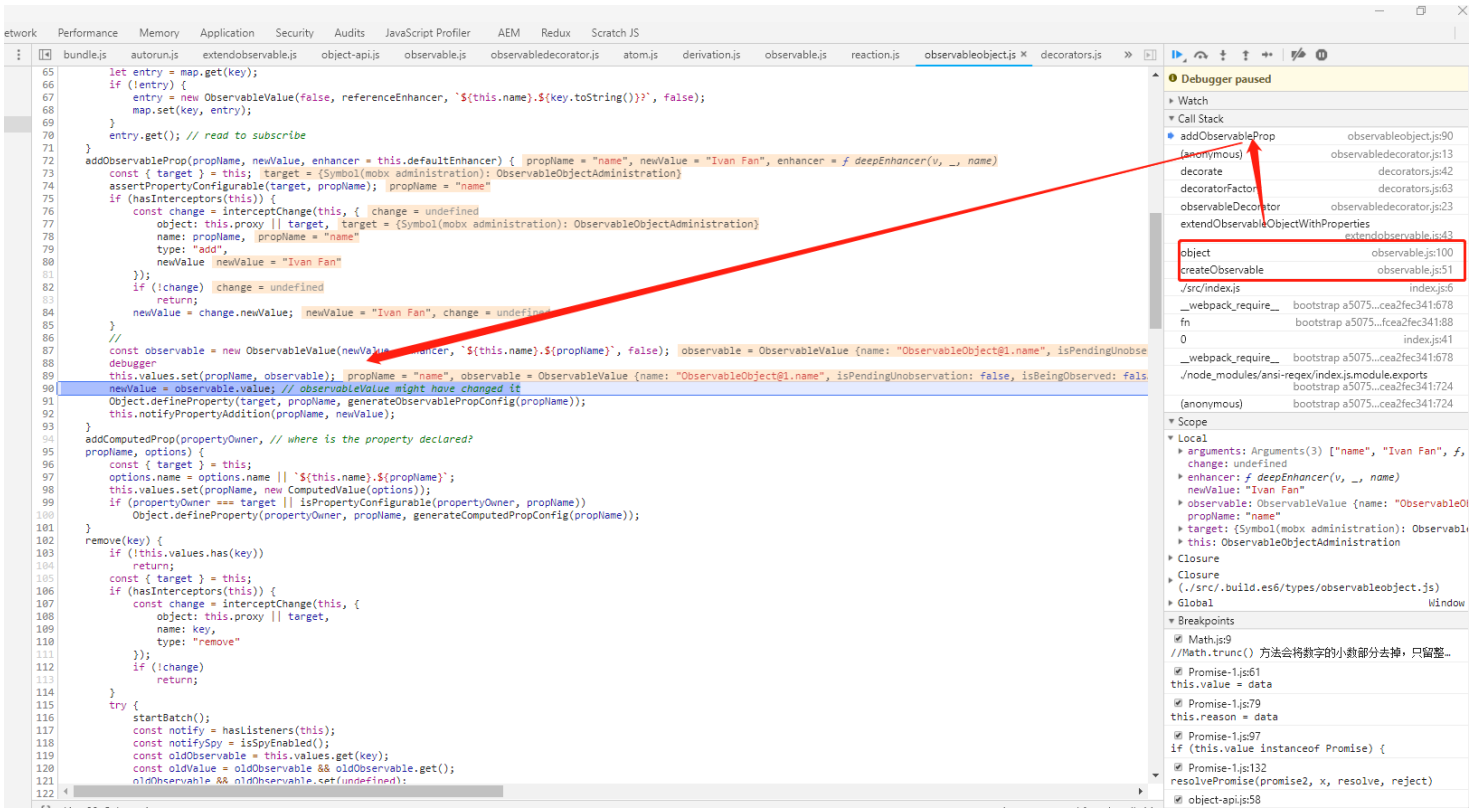


图4

对这个对象的属性的 `get` ， `set` ， `has` ， `deleteProperty` ， `ownKeys` ， `preventExtensions` 方法进行了代理拦截，这个是**Mobx** 事件数据添加的一个核心点。

4. 第三点的 `proxy` 其实只是初始化了一个简单的代理对象，但是没有与我们需要观察的 `target` (也就是 `mobx.observable` 方法传递进来的需要被观察的对象)关联起来， `extendObservableObjectWithProperties$$1(proxy, props, decorators, defaultDecorator);` 方法会遍历 `target` 的属性，将其赋值给 `proxy` 对象， 然后我们 `mobx.observable` 里的对象都被代理了，也就是实现了对属性操作的拦截处理。

5. 在第四点 `extendObservableObjectWithProperties$$1` 方法中， 最终会给原始的对象的属性进行**装饰**，通过查看function 的 call stack 得知，最后对调用 ObservableObjectAdministration 的addObservableProp 方法， 针对每一个**propName**(原始对象的Key)生成一个 ObservableValue 对象，并且保存在ObservableObjectAdministration 对象的**values**中



从图三中发现， 真正实现数据拦截的就是 `objectProxyTraps` 拦截器， 下一章节，我们需要对这个拦截器进行深入分析，着重看 `get` , `set` 如何实现了数据拦截。

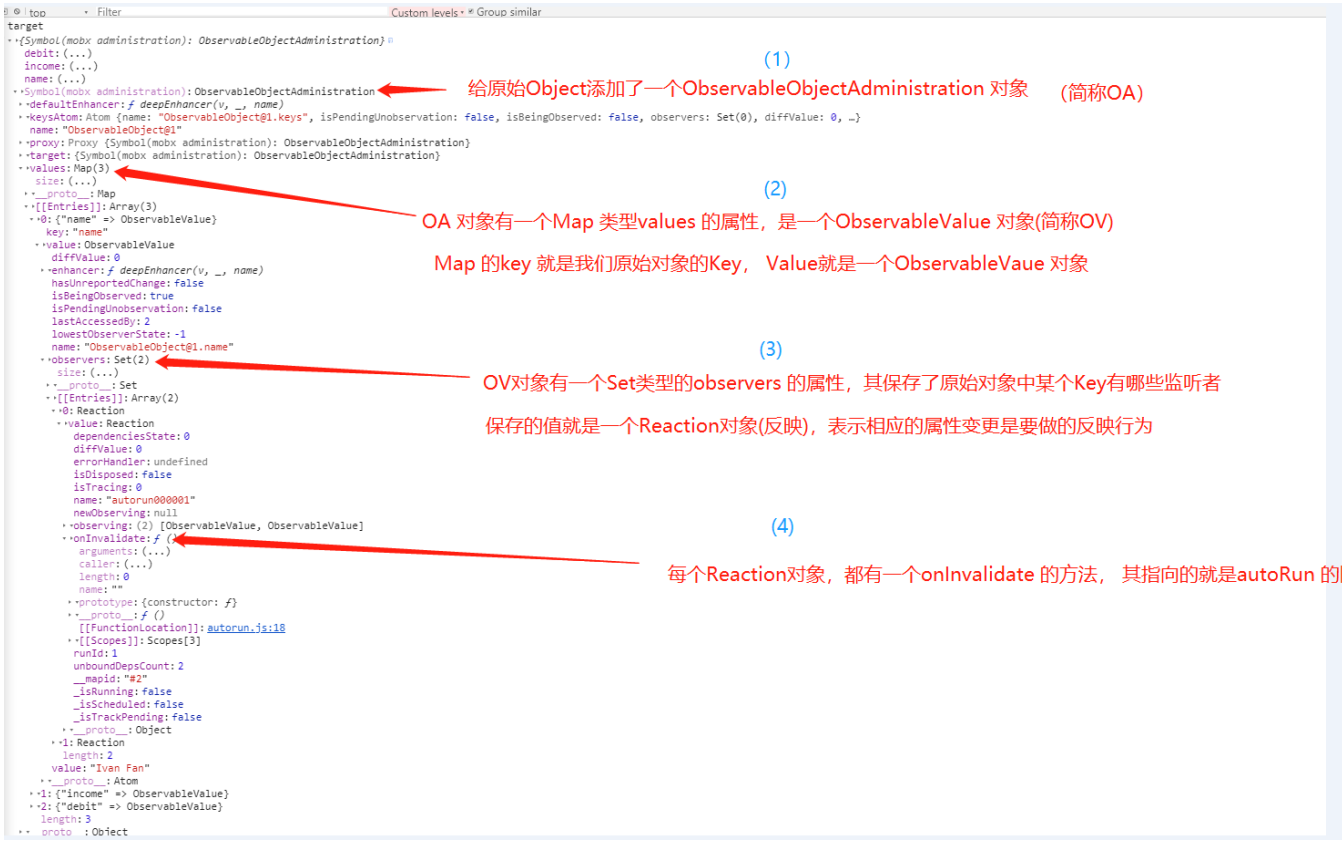
5. `return proxy;` 最终将返回一个已经被代理过的对象，替换原生对象。

> bankUser 对象就是一个已经被代理了的对象，并且包含了一个 `Symbol` 类型的新的属性。

```
const bankUser = mobx.observable({
    name: 'Ivan Fan',
    income: 3,
    debit: 2
});
```

总结

1. observable 首先传入一个原始对象(可以传入多种类型的数据: `array` , `map` , `object` ， 现在只分析object 类型的情况)
2. 创建一个空的Object 对象，并且添加一些默认属性(`var base = extendObservable$$1({}, undefined, undefined, o);`), 包括一个 `Symbol` 类型的属性，其值是一个 `ObservableObjectAdministration` 类型的对象.
3. 将这个对象用**ES6** 的Proxy 进行了代理， 会拦截这个对象的一些列操作(`get` , `set` ...) `var proxy = new Proxy(base, objectProxyTraps);`
4. 将原始对象，进行遍历，将其所有的自己的属性挂载在新创建的空对象中
5. 返回已经加工处理的对象 `bankUser`
6. 后续就可以监听这个对象的相应的操作了。
7. 加工后的对象如下图所示, 后面操作的对象，就是如下这个对象，但是**observable** 方法，其实只是做到了如下图的第二步(2), 第三步(3)的 observers属性还是一个没有任何值的Set 对象，在后续分析**autorun** 方法中，会涉及到在什么时候去给它赋值



(1)

给原始Object添加了一个ObservableObjectAdministration 对象 (简称OA)

(2)

OA 对象有一个Map 类型values 的属性，是一个ObservableValue 对象(简称OV)

Map 的key 就是我们原始对象的Key， Value就是一个ObservableVaue 对象

(3)

OV对象有一个Set类型的observers 的属性，其保存了原始对象中某个Key有哪些监听者

保存的值就是一个Reaction对象(反映)，表示相应的属性变更是要做的反映行为

(4)

每个Reaction对象，都有一个onInvalidate 的方法， 其指向的就是autoRun 的回调函数