**Mobx使用详解**

**observable和autorun**

import { observable, autorun } from 'mobx';

const value = observable.box(0);

const number = observable.box(100);

autorun(() => {

console.log(value.get());

});

value.set(1);

value.set(2);

number.set(101);

可以看到，控制台中依次输出0,1,2。  
observable可以用来观测一个数据，这个数据可以数字、字符串、数组、对象等类型，而当观测到的数据发生变化的时候，如果变化的值处在autorun中，那么autorun就会自动执行。  
上例中的autorun函数中，只对value值进行了操作，而并没有number值的什么事儿，所以number.set(101)这步并不会触发autorun，只有value的变化才触发了autorun。

计算属性——computed（写到这里来了）

假如现在我们用一个数字，但我们对它的值不感兴趣，而只关心这个数字是否为正数。这个时候我们就可以用到computed这个属性了。

const number = observable(10);

const plus = computed(() => number.get() > 0);

autorun(() => {

console.log(plus.get());

});

number.set(-19);

number.set(-1);

number.set(1);

依次输出了true，false，true。  
第一个true是number初始化值的时候，10>0为true没有问题。  
第二个false将number改变为-19，输出false，也没有问题。  
但是当-19改变为-1的时候，虽然number变了，但是number的改变实际上并没有改变plus的值，所以没有其它地方收到通知，因此也就并没有输出任何值。  
直到number重新变为1时才输出true。

实际项目中，computed会被广泛使用到。

const price = observable(199);

const number = observable(15);

//computed的其它简单例子

const allPrice = computed(() => price.get() \* number.get());

顺便一提，computed属性和React Native中的ListView搭配使用很愉快。

action，runInAction和严格模式（useStrict）

mobx推荐被观测变量，修改的行为放在action中。

来看看以下例子:

import {observable, action} from 'mobx';

class Store {

@observable number = 0;

@action add = () => {

this.number++;

}

}

const newStore = new Store();

newStore.add();

以上例子使用了ES7的decorator，在实际开发中非常建议用上它，它可以给你带来更多的便捷

回到我们的例子，这个类中有一个add函数，用来将number的值加1，也就是修改了被观测的变量，根据规范，我们要在这里使用action来修饰这个add函数。  
勇于动手的你会发现，就算我把@action去掉，程序还是可以运行的。

class Store {

@observable number = 0;

add = () => {

this.number++;

}

}

这是因为我们使用了Mobx的非严格模式，如果在严格模式下，就会报错。  
接下来让我们来启用严格模式

import {observable, action, useStrict} from 'mobx';

useStrict(true);

class Store {

@observable number = 0;

@action add = () => {

this.number++;

}

}

const newStore = new Store();

newStore.add();

Mobx用严格模式的函数就是useStrict，注意和原生JS的"use strict"不是一个东西。  
现在再去掉@action就会报错了。  
实际开发的时候建议开起严格模式，这样不至于让你在各个地方很轻易地改变你所需要的值，降低不确定性。

action的写法大概有如下几种([摘自mobx英文文档](https://link.jianshu.com/?t=https://mobx.js.org/refguide/action.html)):

action(fn)

action(name, fn)

@action classMethod() {}

@action(name) classMethod () {}

@action boundClassMethod = (args) => { body }

@action(name) boundClassMethod = (args) => { body }

@action.bound classMethod() {}

@action.bound(function() {})

可以看到，action在修饰函数的同时，我们还可以给它设置一个name，这个name应该没有什么太大的作用，但可以作为一个注释更好地让其他人理解这个action的意图。

接下来说一个重点：**action只能影响正在运行的函数，而无法影响当前函数调用的异步操作**  
比如官网中给了如下例子

@action createRandomContact() {

this.pendingRequestCount++;

superagent

.get('https://randomuser.me/api/')

.set('Accept', 'application/json')

.end(action("createRandomContact-callback", (error, results) => {

if (error)

console.error(error);

else {

const data = JSON.parse(results.text).results[0];

const contact = new Contact(this, data.dob, data.name, data.login.username, data.picture);

contact.addTag('random-user');

this.contacts.push(contact);

this.pendingRequestCount--;

}

}));

}

重点关注程序的第六行。在end中触发的回调函数，被action给包裹了，这就很好验证了上面加粗的那句话，action无法直接影响当前函数调用的异步操作，而这个回调毫无疑问是一个异步操作，所以必须再用一个action来包裹住它，这样程序才不会报错。

当然如果在非严格模式下……那当我没说。

如果你使用async function来处理业务，那么我们可以使用runInAction这个API来解决之前的问题。

import {observable, action, useStrict, runInAction} from 'mobx';

useStrict(true);

class Store {

@observable name = '';

@action load = async () => {

const data = await getData();

runInAction(() => {

this.name = data.name;

});

}

}

runInAction有点类似action('名字', (error, results) => {})的语法糖，调用后，这个action方法会立刻执行。

**结合React使用**

在React中，我们一般把页面相关的数据放到state中，在需要改变这些数据的时候，我们会去用setState这个方法来进行改变。  
设想一个最简单的场景，页面上有个数字0和一个按钮。点击一次按钮让这个数字增加1，就让我们要用Mobx来处理这个试试。

import React from 'react';

import { observable, useStrict, action } from 'mobx';

import { observer } from 'mobx-react';

useStrict(true);

class MyState {

@observable num = 0;

@action addNum = () => {

this.num++;

};

}

const newState = new MyState();

@observer

export default class App extends React.Component {

render() {

return (

<div>

<p>{newState.num}</p>

<button onClick={newState.addNum}>+1</button>

</div>

)

}

}

上例中我们使用了一个MyState类，在这个类中定义了一个被观测的num变量和一个action函数addNum用来改变这个num值。  
之后我们实例化一个对象，叫做newState，之后在我的React组件中，我只需要用@observer修饰一下组件类，便可以愉悦地使用这个newState对象中的值和函数了。

跨组件交互

在不使用其它框架、类库的情况下，React要实现跨组件交互这一功能有些繁琐。通常我们需要在父组件上定义一个state和一个修改该state的函数。然后把state和这个函数分别传到两个子组件里，在逻辑简单，子组件很少的时候还好，当业务很复杂后，这么写就非常繁琐，且难以维护。而用Mobx就可以很好地解决这个问题。来看看以下的例子：

class MyState {

@observable num1 = 0;

@observable num2 = 100;

@action addNum1 = () => {

this.num1 ++;

};

@action addNum2 = () => {

this.num2 ++;

};

@computed get total() {

return this.num1 + this.num2;

}

}

const newState = new MyState();

const AllNum = observer((props) => <div>num1 + num2 = {props.store.total}</div>);

const Main = observer((props) => (

<div>

<p>num1 = {props.store.num1}</p>

<p>num2 = {props.store.num2}</p>

<div>

<button onClick={props.store.addNum1}>num1 + 1</button>

<button onClick={props.store.addNum2}>num2 + 1</button>

</div>

</div>

));

@observer

export default class App extends React.Component {

render() {

return (

<div>

<Main store={newState} />

<AllNum store={newState} />

</div>

);

}

}

有两个子组件，Main和AllNum (均采用无状态函数的方式声明的组件)  
在MyState中存放了这些组件要用到的所有状态和函数。  
之后只要在父组件需要的地方实例化一个MyState对象，需要用到数据的子组件，只需要将这个实例化的对象通过props传下去就好了。

那如果组件树比较深怎么办呢？  
我们可以借助React15版本的新特性context来完成。它可以将父组件中的值传递到任意层级深度的子组件中。

接下来看看网络请求的情况。

useStrict(true);

class MyState {

@observable data = null;

@action initData = async() => {

const data = await getData("xxx");

runInAction("说明一下这个action是干什么的。不写也可以", () => {

this.data = data;

})

};

}

严格模式下，只能在action中修改数据，但是action只能影响到函数当前状态下的情景，也就是说在await之后发生的事情，这个action就修饰不到了，于是我们必须要使用了runInAction(详细解释见上文)。  
当然如果你不开启严格模式，不写runInAction也不会报错。

**个人强烈建议开启严格模式，这样可以防止数据被任意修改，降低程序的不确定性**

**关于@observer的一些说明**

通常，在和Mobx数据有关联的时候，你需要给你的React组件加上@observer，你不必太担心性能上的问题，加上这个@observer不会对性能产生太大的影响，而且@observer还有一个类似于pure render的功能，甚至能起到性能上的一些优化。

所谓pure render见下例:

@observer

export default class App extends React.Component {

state = {

a: 0,

};

add = () => {

this.setState({

a: this.state.a + 1

});

};

render() {

return (

<div>

{this.state.a}

<button onClick={this.add}>+1</button>

<PureItem />

</div>

);

}

}

@observer

class PureItem extends React.Component {

render() {

console.log('PureItem的render触发了');

return (

<div>你们的事情跟我没关系</div>

);

}

}

如果去掉子组件的@observer，按钮每次点击，控制台都会输出 *PureItem的render触发了* 这句话。

React组件中可以直接添加@observable修饰的变量

@observer

class MyComponent extends React.Component {

state = { a: 0 };

@observable b = 1;

render() {

return(

<div>

{this.state.a}

{this.b}

</div>

)

}

}

添加@observer之后，组件会多一个生命周期componentWillReact。组件中被observable观测的数据改变后，就会触发这个生命周期。  
注意setState并不会触发这个生命周期！state中的数据和observable数据并不算是一类。

**另外被observable观测数据的修改是同步的，不像setState那样是异步，这点给我们带了很大便利**。

**Observable Object和Observable Arrays**

本章主要对官方文档Observable Types这一节中的前两章进行了翻译概述。有兴趣的同学可以直接阅读官方文章 [Mobx官方文档——Observable Types](https://link.jianshu.com/?t=https://mobx.js.org/refguide/object.html" \t "_blank)

**Observable Objects**

如果使用observable来修饰一个Javascript的简单对象，那么其中所有属性都将变为可观察的，如果其中某个属性是对象或者数组，那么这个属性也将被observable进行观察，说白了就是递归调用。  
Tips: 简单对象是指不由构造函数创建，而是使用Object作为其原型，或是干脆没有原型的对象。  
需要注意，只有对象上已经存在的属性，才能被observable所观测到。  
若是当时不存在，后续添加的属性值，则需要使用extendObservable来进行添加。

let observableObject = observable({value: 3222});

extendObservable(observableObject, {

newValue: 2333

});

如果是由构造函数创建的对象，那么必须要在它的构造函数中使用observable或extendObservable来观测对象。  
如下所示：

function MyObject(name) {

extendObservable(this, {

name,

});

}

var obj = new MyObject("aaa");

如果对象中的属性是由构造函数创建的对象，那么它也不会被observable给转化。

对象中带有getter修饰的属性会被computed自动转换。

其实observable函数的自动转化已经能够解决至少95%的问题了，如果想要更详细地了解，可以去看 [modifiers这一章](https://link.jianshu.com/?t=https://mobx.js.org/refguide/modifiers.html)

最后附一个购物车的[例子](https://link.jianshu.com/?t=https://github.com/cqm1994617/react-mobx-cart)

**Observable Arrays**

与对象类似，数组同样可以使用observable函数进行转化。

ES5中原生数组对象中存在一定的限制，所以Mobx将会创建一个类数组对象来代替原始数组。在实际使用中，这些类数组的表现和真正的原生数组极其类似，并且它支持原生数组的所有API，包括数组索引、长度获取等。  
但是注意一点，sort和reverse方法返回的是一个新的Observable Arrays，对原本的类数组不会产生影响，这一点和原生数组不一样。

**请记住，这个类数组不管和真实的数组有多么相似，它都不是一个真正的原生数组，所以毫无疑问Array.isArray(observable([]))的返回值都是false。**当你需要将这个Observable Arrays转换成真正的数组时，可以使用slice方法创建一个浅拷贝。换句话来说，Array.isArray(observable([]).slice())会返回true。

除了原生数组支持的API外，Observable Arrays还支持以下API：

* intercept(interceptor)  
  这个方法可以在所有数组的操作被应用之前，将操作拦截。具体的请看[Intercept & Observe](https://link.jianshu.com/?t=https://mobx.js.org/refguide/observe.html" \t "_blank)
* observe(listener, fireImmediately? = false)  
  用来监听数组的变化(类似ES7中的observe，可惜这个ES7中的observe将被废弃)，它返回一个用以注销监听器的函数。
* clear()  
  清空数组
* replace(newArray)  
  用一个新数组中的内容替换掉原有的内容
* find(predicate: (item, index, array) => boolean, thisArg?, fromIndex?)  
  基本上与ES7 Array.find的提案相同，不过多了fromIndex参数。
* remove(value)  
  移除数组中第一个值等于value的元素，如果移除成功，则会返回true
* peek()  
  和slice类似，但它不会创建保护性拷贝，所以性能比slice会更好。如果你能够确定，转换出的数组肯定仅以只读的方式使用，那么可以使用这个API

**总结**

Mobx入门上手非常简单，只要记住少量概念就可以完成许多基础业务了。但深入学习下去，还是要接触许多概念的。例如Modifier、Transation等等。  
最后与Redux做一个简单的对比

1. Mobx写法上更偏向于OOP
2. 对一份数据直接进行修改操作，不需要始终返回一个新的数据
3. 对typescript的支持更好一些
4. 相关的中间件很少，逻辑层、业务整合是一个问题