---

link: null
title: 珠峰架构师成长计划
description: mobx是一个简单可扩展的状态管理库
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=113 sentences=323, words=1819

---

1. mobx

mobx是一个简单可扩展的状态管理库

2. mobx vs redux

mobx学习成本更低，性能更好的的状态解决方案

- 开发难度低
- 开发代码量少
- 渲染性能好

3. 核心思想

状态变化引起的副作用应该被自动触发

- 应用逻辑只需要修改状态数据即可,mobx回自动渲染UI，无需人工干预
- 数据变化只会渲染对应的组件
- MobX提供机制来存储和更新应用状态供 React 使用
- eact 通过提供机制把应用状态转换为可渲染组件树并对其进行渲染

4. 环境准备

4.1 安装依赖模块

```
pm i webpack webpack-cli babel-core babel-loader babel-preset-env babel-preset-react babel-preset-stage-0 babel-plugin-transform-decorators-legacy mobx mobx-react -D
```

4.2 webpack.config.js

```
const path=require('path');
module.exports = {
    mode: 'development',
    entry: path.resolve(__dirname,'src/index.js'),
    output: {
        path: path.resolve(__dirname,'dist'),
        filename:'main.js'
    },
    module: {
        rules: [
            {
                test: /\.jsx?$/,
                exclude: /node_modules/,
                use: {
                    loader: 'babel-loader',
                    options: {
                        presets: ['env','react','stage-0'],
                        plugins:['transform-decorators-legacy']
                    }
                }
            }
        ]
    },
    devtool:'inline-source-map'
}
```

4.3 package.json

```
"scripts": {
    "start": "webpack -w"
},
```

5. Decorator

5.1 类的修饰

- 修饰器（Decorator）函数，用来修改类的行为
- 修饰器是一个对类进行处理的函数。修饰器函数的第一个参数，就是所要修饰的目标类
- 修饰器本质就是编译时执行的函数
- 如果想添加实例属性，可以通过目标类的prototype对象操作

```
@testable
class Person{

}

function testable(target) {
    target.isTestable=true;
}
console.log(Person.isTestable)

@decorator
class A{}
---------------
class A{}
A = decorator(A);
```

5.2 修饰属性

```
class Circle{
    @readonly PI=3.14;
}

function readonly(target,name,descriptor) {
    console.log(descriptor);
    descriptor.writable=false;
}
let c1=new Circle();
c1.PI=3.15;
console.log(c1.PI);
```

5.3 修饰方法

- 修饰器不仅可以修饰类，还可以修饰类的属性。

```
class Calculator{

    add(a,b) {
        return a+b;
    }
}

function logger(target,name,descriptor) {
    let oldValue=descriptor.value;
    descriptor.value=function () {
        console.log(`${name}(${Array.prototype.join.call(arguments,',')})`);
        return oldValue.apply(this,arguments);
    }
}
let oldDescriptor=Object.getOwnPropertyDescriptor(Calculator.prototype,'add');
logger(Calculator.prototype,'add',oldDescriptor);
Object.defineProperty(Calculator.prototype,'add',oldDescriptor);

let calculator=new Calculator();
let ret = calculator.add(1,2);
console.log(ret);
```

6. Proxy

- Proxy 可以理解成，在目标对象之前架设一层"拦截"，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写
- get方法用于拦截某个属性的读取操作，可以接受三个参数，依次为目标对象、属性名和 proxy 实例本身
- set方法用来拦截某个属性的赋值操作，可以接受四个参数，依次为目标对象、属性名、属性值和 Proxy 实例本身

```
var proxy = new Proxy(target, handler);
```

```
let p1=new Proxy({name:'zfpx'},{
    get: function (target,key,receiver) {
        console.log(`getting ${key}`);
        console.log(receiver);
        return Reflect.get(target,key,receiver);
    },
    set: function (target,key,value,receiver) {
        console.log(`setting ${key}`);
        return Reflect.set(target,key,value,receiver);
    }
});
console.log(p1.name);
```

7. mobx

7.1 observable

- MobX为现有的数据结构(如对象，数组和类实例)添加了可观察的功能。
- observable就是一种让数据的变化可以被观察的方法
- 先把数据转化成可以被观察的对象，那么对这些数据的修改就可以备监视

7.1.1 引用类型 (observable)

类型 描述 对象 数组

```
const {observable,isArrayLike}=require('mobx');
function observable2(target) {
    return new Proxy(target,{});
}
const p1=observable2({name:'zfpx'});
console.log(p1.name);
```

```
const {observable}=require('mobx');
function observable2(target) {
    return new Proxy(target,{

    });
}
const p1=observable([1,2,3]);
p1.push(4);
p1.pop();
console.log(p1);
console.log(Array.isArray(p1));
```

7.1.2 基本类型(observable.box)

类型 描述 String 字符串 Boolean 布尔值 Number 数字 Symbol 独一无二的值

```
const {observable}=require('mobx');
let num=observable.box(10);
let str=observable.box('hello');
let bool=observable.box(true);
console.log(num.get(),str.get(),bool.get());
num.set(100);
str.set('world');
bool.set(false);
console.log(num.get(),str.get(),bool.get());
```

7.1.3 decorator

```
import {observable} from 'mobx';
class Store {
    @observable name='zfpx';
    @observable age=9;
    @observable isMarried=false;

    @observable hobbies=[];
    @observable home={name:'北京'};
    @observable skills=new Map();
}
```

8. 使用对可观察对象做出响应

8.1 computed

- 计算值(computed values)是可以根据现有的状态或其它计算值衍生出的值
- 组合已有的可观察数据，成为新的可观察数据
- 既是反应又是可观察数据
- 可以作为函数使用也可以作为decorator使用
- 使用 .get() 来获取计算的当前值
- 使用 .observe(callback) 来观察值的改变。
- computed值可以引用其它computed的值，但是不能循环引用

```

let {observable,computed} = require('mobx');
class Store {
    @observable name='zfpx';
    @observable age=9;
    @observable area='010';
    @observable number="18910092296"

    @observable province="广东";
    @observable city="东莞";
    @computed get home() {
        return this.province+this.city;
    }
}

let store=new Store();
let cell = computed(function () {
    return store.area+'-'+store.number;
});
cell.observe(change=>console.log(change));
console.log(cell.get());
store.area='020';
store.number='15718856132';
console.log(cell.get());
console.log(store.home);
store.province='山东';
store.city='济南';
console.log(store.home);
```

8.2 autorun

- 如果使用修饰器模式，则不能再用observe方法了
- 当你想创建一个响应式函数，而该函数本身永远不会有观察者时,可以使用 mobx.autorun
- 当使用 autorun 时，所提供的函数总是立即被触发一次，然后每次它的依赖关系改变时会再次被触发
- 数据渲染后自动渲染

```
autorun(() => {

    console.log(store.home);
});

store.province='山东';
store.city='济南';
```

8.3 when

- when 观察并运行给定的 predicate，直到返回true。
- 一旦返回 true，给定的 effect 就会被执行，然后 autorunner(自动运行程序) 会被清理。
- 该函数返回一个清理器以提前取消自动运行程序。

```
when(predicate: () => boolean, effect?: () => void, options?)
```

```
let dispose = when(() => store.age>=18,()=>{
    console.log('你已经成年了!')
});
dispose();
store.age=10;
store.age=20;
store.age=30;
```

8.4 reaction

- autorun的变种， autorun会自动触发， reaction对于如何追踪 observable赋予了更细粒度的控制
- 它接收两个函数参数，第一个(数据 函数)是用来追踪并返回数据作为第二个函数(效果 函数)的输入
- 不同于 autorun的是当创建时效果 函数不会直接运行，只有在数据表达式首次返回一个新值后才会运行
- 可以用在登录信息存储和写缓存逻辑

```
reaction(() => [store.province,store.city],arr => console.log(arr.join(',')));
store.province='山东';
store.city='济南';
```

9. action

- 前面的方式每次修改都会触发 autorun和 reaction执行
- 用户一次操作需要修改多个变量，但是视图更新只需要一次
- 任何应用都有动作,动作是任何用来修改状态的东西
- 动作会分批处理变化并只在(最外层的)动作完成后通知计算值和反应
- 这将确保在动作完成之前，在动作期间生成的中间值或未完成的值对应用的其余部分是不可见的*9.1 action #

```
let {observable,computed,autorun,when,reaction,action} = require('mobx');
class Store {
    @observable province="广东";
    @observable city="东莞";
    @action moveHome(province,city) {
         this.province=province;
         this.city=city;
    }
}
let store=new Store();
reaction(() => [store.province,store.city],arr => console.log(arr.join(',')));
store.moveHome('山东','济南');
```

** 9.2 action.bound #**

```
let {observable,computed,autorun,when,reaction,action} = require('mobx');
class Store {
    @observable province="广东";
    @observable city="东莞";
    @action.bound moveHome(province,city) {
         this.province=province;
         this.city=city;
    }
}
let store=new Store();
reaction(() => [store.province,store.city],arr => console.log(arr.join(',')));
let moveHome=store.moveHome;
moveHome('山东','济南');
```

** 9.3 runInAction #**

```
runInAction(() => {
    store.province='山东';
    store.city='济南';
});
```

10. mobx应用

- mobx-react 核心是将render方法包装为autorun
- 谁用到了可观察属性，就需要被observer修饰，按需渲染

```
cnpm i react react-dom mobx-react -S
```

** 10.1 计数器 #**

```
import React,{Component} from 'react';
import ReactDOM from 'react-dom';
import {observable,action} from 'mobx';
import PropTypes from 'prop-types';
import {observer} from 'mobx-react';

class Store {
    @observable number=0;
    @action.bound add() {
        this.number++;
    }
}
let store=new Store();

@observer
class Counter extends Component{
    render() {
        return (
            <div>
                <p>{store.number}p>
                <button onClick={store.add}>+button>
            div>
        )
    }
}
ReactDOM.render(<Counter/>,document.querySelector('#root'));
```

```
import React,{Component} from 'react';
import ReactDOM from 'react-dom';
import {observable,action} from 'mobx';
import PropTypes from 'prop-types';
import {observer} from 'mobx-react';

class Store {
    @observable counter={number:0};
    @action.bound add() {
        this.counter.number++;
    }
}
let store=new Store();
@observer
class Counter extends Component{
    render() {
        return (
            <div>
                <p>{this.props.counter.number}p>
                <button onClick={this.props.add}>+button>
            div>
        )
    }
}
ReactDOM.render(<Counter
    counter={store.counter}
    add={store.add}
/>,document.querySelector('#root'));
```

** 10.2 TODO #**

```
import React,{Component,Fragment} from 'react';
import ReactDOM from 'react-dom';
```

```
import {observable,action, computed} from 'mobx';
import PropTypes from 'prop-types';
import {observer,PropTypes as ObservablePropTypes} from 'mobx-react';
class Todo{
    id=Math.random();
    @observable text='';
    @observable completed=false;
    constructor(text) {
        this.text=text;
    }
    @action.bound toggle() {
        this.completed=!this.completed;
    }
}
class Store{
    @observable todos=[];
    @computed get left() {
        return this.todos.filter(todo=>!todo.completed).length;
    }
    @computed get filterTodos() {
        return this.todos.filter(todo => {
            switch (this.filter) {
                case 'completed':
                    return todo.completed;
                case 'uncompleted':
                    return !todo.completed;
                default:
                    return true;
            }
        });
    }
    @observable filter='all';
    @action.bound changeFilter(filter) {

        this.filter=filter;
        console.log(this.filter);
    }
    @action.bound addTodo(text) {
        this.todos.push(new Todo(text));
    }
    @action.bound removeTodo(todo) {
        this.todos.remove(todo);
    }
}
@observer
class TodoItem extends Component{
    static porpTypes={
        todo: PropTypes.shape({
            id: PropTypes.number.isRequired,
            text: PropTypes.string.isRequired,
            completed:PropTypes.bool.isRequired
        }).isRequired
    }
    render() {
        let {todo}=this.props;
        return (
            <Fragment>
                <input
                    type="checkbox"
                    onChange={todo.toggle}
                    checked={todo.completed} />
                <span className={todo.completed? 'completed':''}>{todo.text}span>

            Fragment>
        )
    }
}
@observer
class TodoList extends Component{
    static propsTypes={
        store: PropTypes.shape({
            addTodo:PropTypes.func,
            todos:ObservablePropTypes.observableArrayOf(ObservablePropTypes.observableObject)
        }).isRequired
    };
    state={text:''}
    handleSubmit=(event) => {
        event.preventDefault();
        this.props.store.addTodo(this.state.text);
        this.setState({text:''});
    }
    handleChange=(event) => {
        this.setState({text:event.target.value});
    }
    render() {
        let {filterTodos,left,removeTodo,filter,changeFilter}=this.props.store;
        return (
            <div className="todo-list">
                <form onSubmit={this.handleSubmit}>
                    <input placeholder="请输入待办事项" type="text" value={this.state.text} onChange={this.handleChange}/>
                form>
                <ul>
                    {
                        filterTodos.map(todo => (
                            <li key={todo.id}>
                                <TodoItem todo={todo} />
                                <button onClick={()=>removeTodo(todo)}>Xbutton>
                            li>
                        ))
                    }
                ul>
                <p>
                    <span>你还有{left}件待办事项!span>
                    <button
```

```
                    onClick={()=>changeFilter('all')}
                    className={filter==='all'?'active':''}>全部button>
                <button onClick={() => changeFilter('uncompleted')}
                    className={filter==='uncompleted'?'active':''}>未完成button>
                <button
                    onClick={()=>changeFilter('completed')}
                    className={filter==='completed'?'active':''}>已完成button>
            p>
        div>
    )
  }
}
let store=new Store();
ReactDOM.render(<TodoList store={store}/>,document.querySelector('#root'));
```

11.优化

** 11.1 observe #**

```
  constructor() {
      observe(this.todos,change => {
          console.log(change);
          this.disposers.forEach(disposer => disposer());
          this.disposers=[];
          for (let todo of change.object) {
              this.disposers.push(observe(todo,change => {
                  this.save();

              }));
          }
          this.save();
      });
  }
```

** 11.2 spy #**

```
spy(event => {

})
```

** 11.3 toJS #**

```
  constructor() {
      observe(this.todos,change => {
          console.log(change);
          this.disposers.forEach(disposer => disposer());
          this.disposers=[];
          for (let todo of change.object) {
              this.disposers.push(observe(todo,change => {
                  this.save();

              }));
          }
          this.save();
      });
  }
  save() {
      localStorage.setItem('todos',JSON.stringify(toJS(this.todos)));
  }
```

** 11.4 trace #**

```
trace
```

** 12. 优化 #**

- 把视图拆解的更细致
- 使用专门的视图渲染列表数据
- 尽可能晚的解构使用数据

```
import React,{Component,Fragment} from 'react';
import ReactDOM from 'react-dom';
import {trace,observable,action, computed, observe, spy,toJS} from 'mobx';
import PropTypes from 'prop-types';
import {observer,PropTypes as ObservablePropTypes} from 'mobx-react';
spy(event => {

})
class Todo{
  id=Math.random();
  @observable text='';
  @observable completed=false;
  constructor(text) {
      this.text=text;

  }
  @action.bound toggle() {
      this.completed=!this.completed;
  }
}
class Store{
  disposers=[];
  constructor() {
      observe(this.todos,change => {
          console.log(change);

          this.disposers.forEach(disposer => disposer());
          this.disposers=[];
          for (let todo of change.object) {
              this.disposers.push(observe(todo,change => {
                  this.save();

              }));
          }
          this.save();
```

```
        });
    }
    save() {
        localStorage.setItem('todos',JSON.stringify(toJS(this.todos)));
    }
    @observable todos=[];
    @computed get left() {
        return this.todos.filter(todo=>!todo.completed).length;
    }
    @computed get filterTodos() {
        return this.todos.filter(todo => {
            switch (this.filter) {
                case 'completed':
                    return todo.completed;
                case 'uncompleted':
                    return !todo.completed;
                default:
                    return true;
            }
        });
    }
    @observable filter='all';
    @action.bound changeFilter(filter) {

        this.filter=filter;
        console.log(this.filter);
    }
    @action.bound addTodo(text) {
        this.todos.push(new Todo(text));
    }
    @action.bound removeTodo(todo) {
        this.todos.remove(todo);
    }
}
@observer
class TodoItem extends Component{
    static porpTypes={
        todo: PropTypes.shape({
            id: PropTypes.number.isRequired,
            text: PropTypes.string.isRequired,
            completed:PropTypes.bool.isRequired
        }).isRequired
    }
    render() {
        trace();
        let {todo}=this.props;
        return (
            <Fragment>
                <input
                    type="checkbox"
                    onChange={todo.toggle}
                    checked={todo.completed} />
                <span className={todo.completed? 'completed':''}>{todo.text}span>

            Fragment>
        )
    }
}
@observer
class TodoFooter extends Component{
    static propTypes={

    };
    render() {
        trace();
        let {left,filter} = this.props.store;
        return (
            <div>
                    <span>你还有{left}件待办事项!span>
                    <button
                        onClick={()=>changeFilter('all')}
                        className={filter==='all'?'active':''}>全部button>
                    <button onClick={() => changeFilter('uncompleted')}
                        className={filter==='uncompleted'?'active':''}>未完成button>
                    <button
                        onClick={()=>changeFilter('completed')}
                        className={filter==='completed'?'active':''}>已完成button>
                div>
        )
    }
}
@observer
class TodoViews extends Component{
    render() {
        return (
                <ul>
                    {
                        this.props.store.filterTodos.map(todo => (
                            <li key={todo.id}>
                                <TodoItem todo={todo} />
                                <button onClick={()=>removeTodo(todo)}>Xbutton>
                            li>
                        ))
                    }
                ul>
        )
    }
}
@observer
class TodoHeader extends Component{
    state={text:''}
    handleSubmit=(event) => {
        event.preventDefault();
        this.props.store.addTodo(this.state.text);
```

```
      this.setState({text:''});
  }
  handleChange=(event) => {
      this.setState({text:event.target.value});
  }
  render() {
      return (
          <form onSubmit={this.handleSubmit}>
                  <input placeholder="请输入待办事项" type="text" value={this.state.text} onChange={this.handleChange}/>
              form>
      )
  }
}
@observer
class TodoList extends Component{
  static propsTypes={
      store: PropTypes.shape({
          addTodo:PropTypes.func,
          todos:ObservablePropTypes.observableArrayOf(ObservablePropTypes.observableObject)
      }).isRequired
  };

  render() {
      trace();
      return (
          <div className="todo-list">
              <TodoHeader store={this.props.store}/>
              <TodoViews store={this.props.store}/>
              <TodoFooter store={this.props.store}/>
          div>
      )
  }
}
let store=new Store();
ReactDOM.render(<TodoList store={store}/>,document.querySelector('#root'));
```