02

OPEN ORIENTED

凹 凸 实 验 室

Nerv + Mobx 使用

爽

mobx

02

mobx是什么?

一种简单、高扩展的状态管理库

mobx

用起来有点像vuex，把vuex搬到react里使用的感觉

核心概念

02

- (@)observable，初始化需要监听的数据
- (@)computed，类似于vue的computed
- autorun，被监听的数据改变时就会自动运行的函数
- (@)action，数据的变化最好都放action中，使得代码结构更优
- (@)observer，用于包裹react组件，数据更新会使组件重新渲染

使用

02

1、直接配合mobx-react，在nerv组件里使用

栗子栗子


```
@observer
class Demo1 extends Component {
  @observable num = 0
  @observable amount = 1
  @computed get total() {
    return this.num * this.amount;
  }
  @action handleNumChange (e){
    this.num = e.target.value
  }
  @action handleAmountClick() {
    this.amount++
  }

  componentDidMount() {
    autorun(() => {
      console.log('demo1', this.num, this.amount)
    })
    when(
      () => { return this.amount === 6},
      () => { console.log('when1', this.num, this.amount) }
    )
  }

  render() {
    return (
      <div className="mobx_item" style={{ width: '50%', float: 'left' }}>
        <p>这里是demo1</p>
        <input type='text' onChange={this.handleNumChange.bind(this)} />
        <button onClick={this.handleAmountClick.bind(this)}>加1加1</button>
        <p>总值是: {`${this.num} * ${this.amount} = ${this.total}`}</p>
      </div>
    );
  }
```

使用

02

2、将store分离出来，与组件解耦

栗子栗子

02

```
class NumStore {
  @observable num = 0
  @observable amount = 1
  @computed get total() {
    return this.num * this.amount;
  }

  constructor () {
    autorun(() => {
      console.log('demo2', this.num, this.amount)
    })
    when(
      () => { return this.amount === 6 },
      () => { console.log('when2', this.num, this.amount) }
    )
  }

  @action numChange(value) {
    this.num = value
  }
  @action amontAdd() {
    this.amount++
  }
}

const numStore = new NumStore()
```

```
@observer
class Demo2 extends Component {

  handleNumChange = (e) => {
    numStore.numChange(e.target.value)
  }

  handleAmountClick = () => {
    numStore.amontAdd()
  }

  render() {
    return (
      <div className="mobx_item" style={{ width: '50%', marginLeft: '50%'}}>
        <p>这里是demo2</p>
        <input type='text' onChange={this.handleNumChange} />
        <button onClick={this.handleAmountClick}>加1加1</button>
        <p>总值是: {`${numStore.num} * ${numStore.amount} = ${numStore.total}`}</p>
      </div>
    );
  }
}
```

注意事项

(@)action

- 应该开启严格模式，这样改变数据就只能在action中

- 异步action，回调函数也应该被包裹

- 若使用async/await，则需要使用runInAction函数

栗子栗子


```
@action asyncTest() {
  setTimeout(action(() => {
    this.str = '异步变化'
  }), 2000)
}
```

```
@action /*可选的*/ updateDocument = async () => {
  const data = await fetchDataFromUrl();
  /* 在严格模式下是强制的: */
  runInAction("update state after fetching data", () => {
    this.data.replace(data);
    this.isSaving = true;
  })
}
```

注意事项

自动渲染

- 某一被observable的变量改变后要触发组件的自动更新，则被observer的组件里，render函数中需要有对该变量的使用

- 自动渲染时，不会显式的调用React的生命周期方法如**componentShouldUpdate** 或 **componentWillUpdate**；而会有一个新的生命函数钩子， **componentWillReact**

构建大型项目

02

偏大型的项目如何组织代码?

构建大型项目

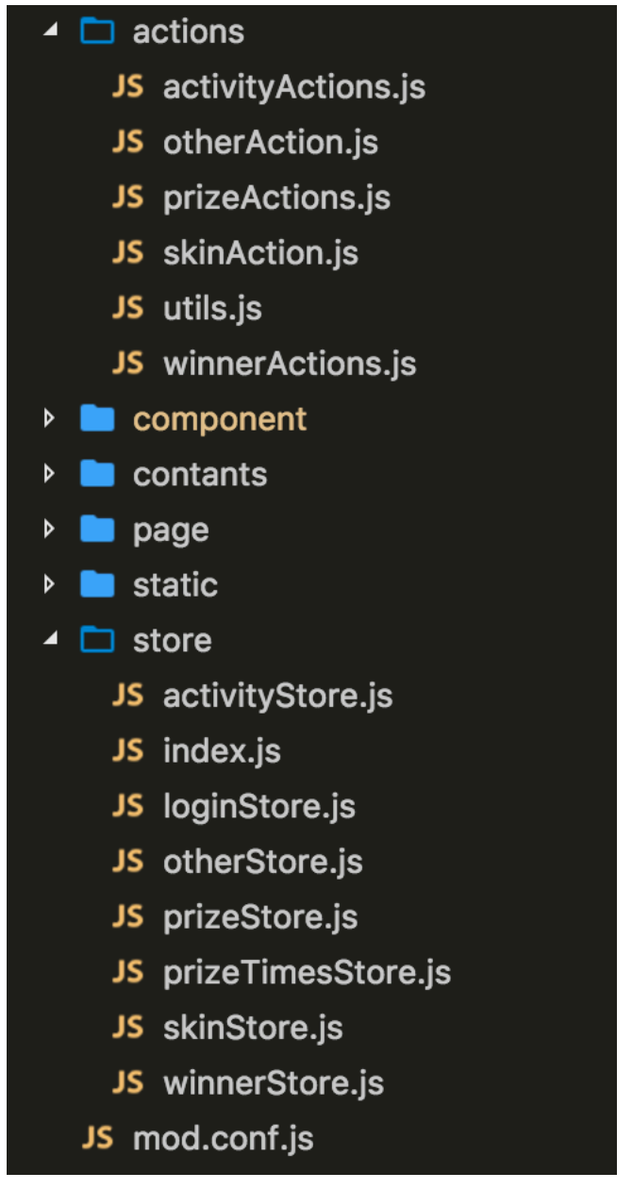
- 参考了mobx里awesome list里面的例子

  https://github.com/rwieruch/favesound-mobx

- Actions（类似redux-react里的action），Store分离

- 通过provider、inject来跨组件引入store

栗子栗子



- 有点类似redux的组织方式，不过会比其更松散，自由度更高

简单总结

02

所以和redux究竟有什么差别，适合场景在哪

简单总结

Redux

- 数据流流动很自然，有更好的可预测性和错误定位能力

- 是单一数据源，状态对象是不可变的

- 更好的**可扩展性**和**可维护性**所需要的代价是更复杂，冗余的代码，通常需要搭配许多中间件的使用

而mobx

02

Mobx

- 数据流流动不那么自然，用到数据才会引发绑定，局部更新更精确

- 一般由多个stroe各自管理自身状态，状态是可变的

- 代码量更少，灵活性更高，可以更自由地组织代码，使用更方便；但也会造成**可扩展性**和**可维护性**的下降

如何抉择

02

- 中小型应用，数据流不太复杂的情况下，可以考虑使用mobx，获取更高的灵活性和便捷性

- 大型复杂应用，数据流及其复杂，应该使用redux，保证项目的可扩展性和可维护性，hold住多人协作

参考资料

- 官方文档：https://suprise.github.io/mobx-cn/fp.html

- Mobx 思想的实现原理，及与 Redux 对比

- MobX vs Redux: Comparing the Opposing Paradigms - React Conf 2017 纪要

THANKS

FOR YOUR WATCHING

02

OPEN ORIENTED

凹 凸 实 验 室