ReactJS 技术栈入门

对react技术栈的学习和记录

安装

- 安装一个普通的 `react` 应用

> `npx create-react-app <react-name>`

`npx` 在检测到本地没有 create-react-app 的这个命令后， 会向网上的npm注册表中查找相应的命令进行安装，这样相比于vue来说不需要本地安装手脚架工具

API

ReactDOM.render

- 参数 (`element` ， `container`)

> `element` 需要渲染的内容，可以是原生的**dom**，也可以是**react**中的组件，以根的形式存在
> `container` 挂载的容器，就是渲染的**dom**会在**container**以子代标签的形式存在

一个 `react` 最小的应用

- 使用

```js
import ReactDOM from "react-dom";

ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

概念

- react有着不同类型的组件，不同类型的组件以类或者函数的形式存在，而在类中有一个渲染函数再 `return` ，在函数中直接 `return` html，而更好的编写是以 `JSX` 的形式返回。
- 使用JavaScript表达式直接使用一对 `{}` 使用任意的JavaScript表达式 ，而不像是 `vue` 里面的 `{{}}`
- `rander` 函数，每次响应式数据发生变化的时候，都会重新执行这个函数，但是每次更新不会全部更新，只会更新不同的地方

生命周期

- `componentDidMount()` 在组件已经渲染到DOM上运行
- `componentWillUnmount()` 组件被销毁的时候执行

JSX

- 在 `vs code` 中使用JSX， 使用 `Babel JavaScript` 插件正确高亮显示 `JSX`
- 推荐将 `JSX` 的内容放在()中
- `JSX` 也是一个函数表达式，会被编译成普通的函数进行调用
- 在 `JSX` 中属性包括原生属性，会使用小驼峰的形式
- `JSX` 会自动进行转义，不需要担心 `xss` 攻击

```jsx
const helloWord = (<h1>Hello World</h1>);
```

编译过程

- js

```jsx
const helloWorld = (<h1 className="title">Hello World</h1>);
```

- js

jsx

```
const helloWorld = React.createElement({
  'h1',
  {className, 'title'},
  'hello World'
})
```

组件

- 组件名称以大写字母开头
- 对于 `React.Compontent` 这个类实现类继承，在类里面，返回一个 `render` 函数
- 在组件内添加相应式数据，在组件的实现类里面重写构造函数，在 `this.state = {}` 里面添加数据和 `vue` 里面的 `data () { return {  } }` 一样

jsx

```
class MyButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {

    };
  }
  render() {
    return (<button>Click Me</button>);
  }
}
```

函数式组件

- 不需要render函数， 直接return JSX对象

jsx

```
function MyButton(props) {
  return (
    // 在onClick中没有箭头函数也没有, 也没有()执行符号
    <button className="square" onClick={props.changeCurrentHandle}>
      {props.value}
    </button>
```

```
    );
  }
```

在组件中添加事件

- 采用小驼峰命名方式
- 最好不要在自定义的组件上添加事件，需要在 `html` 上添加事件, 因为在自定义组件上添加事件会被当作传值给组件的属性, 当属性值为方法的时候也是可以被组件传递的

```jsx
<button className="square" onClick={() => { this.props.changeCurrentHar
  {this.props.value}
</button>
```

props

- 在组件中从父组件传递过来的数据
- 在每个组件中的props保存不变

state

- 会随着变化而变化
- 不要直接修改state的值， 使用setState
- state 更新是异步的，无法使用更新的值，来改变下一个状态
- setState 会重新执行render函数重新渲染

使用

遍历渲染

- 需要给循环渲染的列表元素给予key值
- key值帮助react识别哪些元素是被改变了
- key应该在需要循环的元素上添加
- 在兄弟节点中是唯一的，在全局可以是相同的

jsx

```jsx
function Component1 () {
return (
  <ul>
  {
    list.map((user, index) => (
      <li key={index}>{user.name}</li>
    );
  }
  <ul>
);
}
```

组件的传值

- 和vue一样，通过 `this.props` 属性获取父组件传过来的值

条件渲染

- 在return中进行判断

jsx

```jsx
function Component1(props) {
if (props.login) {
  return (<h1>logined</h1>);
}
return (<h1>not login</h1>);
}
```

this指向问题

- 在以类编写的组件中

jsx

```jsx
class MyBtn extends React.Component {
  clickHandle () {
    console.log(this) // undefined
  }
  render () {
```

```
      return (
        <button onClick={ this.clickHandle }>btn1</button>
      );
    }
  }
  // 1. 改变this的指向, 在构造函数中
  class MyBtn extends React.Component {
    constructor() {
      this.clickHandle = this.clickHandle.bind(this)
    }
    clickHandle () {
      console.log(this) // Mybtn {}
    }
    render () {
      return (
        <button onClick={ this.clickHandle }>btn</button>
        // 或者
        <button onClick={ this.clickHandle.bind(this) }>btn</button>
      );
    }
  }
  // 2. 在函数定义的时候或者绑定事件的时候使用箭头函数
  class MyBtn extends React.Component {
    clickHandle = () => {
      console.log(this) // Mybtn {}
    }
    render() {
      return (
         <button onClick={ this.test }>btn</button>
        // 或者
         <button onClick={ () => this.test() }>btn</button>
      );
    }
  }
```

在React中可以使用到的第三方工具

CSS In Js

emotion 模块

- 安装

```
@emotion/styled @emotion/react
```

- vs code安装语法高亮插件

```
vscode-styled-components
```

时间处理

dayjs

qs

- 解析请求参数

Redux

- 单一数据源
- state是只读的
- 使用纯函数进行修改

安装

安装redux

```
yarn add reduex
```

安装react-redux

```
yarn add react-redux
```

store

- state值，获取，监听

state

- 数据源

action

- 将数据从应用 -> store 的载体
- 本质是一个javaScript 对象
- 必须要有type参数

reducer

- 本质是函数
- 需要有 `return state` ,才能被store接收

> 相当于action为对象的key，reducer表示key对应的方法，有action确定，reducer来执行

基本使用

在 `vue` 中 `state` 直接存放在 `store` 中，而在 `react` 中 `state` 相当于存放在 `reducer` 中，在 `reducer` 中的方法中改变值的变化，将 `state` 返回给 `store` ，而在外可以通过 `store` 监听和获取 `state`

1. 创建一个action，表示要执行的操作

```js
export const plusAction() = {
  return {
    type: 'PLUS' // require true
  }
}
export const minusAction() = {
  return {
    type: 'MINUS'
  }
}
```

2. 创建一个 `reducer` ，接收 `（state，action）` ，将 `state` 返回给 `store`

```js
const initialState = {
  value: 0
}
export const counterReducer = (state = initialValue, action) => {
  switch(action.type) {
    case 'PLUS':
      return {
        value: state.value + 1
      };
    case 'MINUS':
      return {
        value: state.value - 1
      };
    default:
      return state; // 返回默认值
  }
}
```

3. 创建一个 `store` ，接收 `reducer`

```js
const { createStore } from 'redux'
const { counterReducer } from './reducers/index.js'
export const store = createStore(counterReducer)
```

4. 导入 `store` ，当数据需要发生变换的时候， `dispatch` 派发（对象） `action` ，执行 `reducer` 里面相应的方法，返回 `store`

5. 监听 `state` 的变换，在 `componentWillDidMount` 生命周期的时候设置监听器，设置后每次 `dispatch` 都会监听变化

```jsx
import React from 'react'
import { store } from './store/index.js'
import { plusAction, minusAction } from './actions/index.js'

class IndexPage extends React.Component {
  plusClickHandle = () => {
    store.dispatch(plusAction())
  }
```

```
    minusClickHandle = () => {
      store.dispatch(minusAction())
    }
    componentWillDidMount() {
      store.subscribe(() => {
        console.log(store.getState().value)
      })
      this.setState({}) // 监听变化进行 更新
    }
    render() {
      return (
        <div class="index-page">
          <button onClick={ this.plusClickHandle }> + </button>
          <button onClick={ this.minusClickHandle }> - </button>
          <div>{store.getState().value}</div>
        </div>
      );
    }
  }
```

React-redux

- 在 `react` 中更好使用 `redux`

```
yarn add react-redux
```

`connect`

- 参数 `(mapStateToProps, mapDispatchToProps)`
- 可以在组件的props中获取 `state` ，和派发 `action`

基础使用

connect，会将state和dispatch合并到组件的props，这里的合并是平级的，也就是说，在state对象里面的属性值，会变成props里面的属性值

! 好像使用connect获取state值会有问题，如直接在jsx中使用this.props会有问题

1. 和之前使用类似需要创建 `reducer` 来创建 `store` 导出

```js
// reducers/index.js
const initialState = {
  counter: 0
};

export reducer = (state = initialState, action) => {
  switch(action.type) {
    case 'PLUS':
      return {
        counter: state.counter + 1
      };
    default:
      return state;
  }
}
```

```js
// store/index.js
import { createStore } from 'redux';
import { reducer } from './reducers/index.js';

export default createStore(reducer);
```

2. 在需要的使用的最外层包裹 `Provider` , 使用 `store` 作为参数传入

```jsx
import { Provider } from 'react-redux';

<Provider store={ store }>
  <Home></Home>
</Provider>
```

3. 在需要改变state状态的组件中使用 `connect(null, mapDispathToProps)(Btn)` 使用第二个参数，在 `Btn` 组件中可以通过 `this.props` 获取dispatch相应的方法，执行方法就是相当于执行 `stroe.dispatch({type: 'PLUS'})`

jsx

```jsx
import React from 'react';

class Btn extends React.Component {

  clickHandle = () => {
    this.props.plusAction()
  }

  render() {
    return (
      <button onClick={ this.clickHandle }></button>
    );
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    plusAction: () => {
      dispatch({
        type: 'PLUS'
      });
    }
  };
};

export default connect(null, mapDispatchToProps)(Btn);
```

4. 在需要获取state值的地方使用 `connect(mapStateToProps)` 的第一个方法，就可以直接通过 `this.props` 获取

jsx

```jsx
import React from 'react';

class Text extends React.Component {
  render() {
    return (
      <div>{this.props.counter}</div>
    );
  }
}
```

```
const mapStateToProps = (state) => {
  return state;
}

export default connect(mapStateToProps)(Text);
```

reducer 拆分

在使用 connect 获取 state 时候， mapStateToProps 直接返回 state -> this.props -> {data1, data2} , mapStateToProps 返回 state.data1 -> this.props -> data1

- combineReducers

```js
import reducer1 from './reducers/reducer1.js'
import reducer2 from './reducers/reducer2.js'

const reducerRoot = combineReducers({
  data1: reducer1,
  data2: reducer2
})
```

- 其他用法与上面差不多

异步 action

redux-thunk

1. 创建 store ， 加入中间件

```js
import { createStore, applyMiddleWare } from 'redux';
import thunk from 'redux-thunk';
import { reducer } from './reducers/index.js';

export const store = createStore(reducer, applyMiddleWare(thunk));
```

2. 创建异步的 creator action

```js
export const fetchList = () => {
  return async (dispatch) => {
    // 获取数据
    const list = await getList(param)
    dispatch({
      type: 'GET_LIST',
      list
    })
  }
};
```

3. 使用 connect 进行 dispatch ，将数据存放在store中

```jsx
import { connect } from 'react-redux'
import { fetchList } from './actions'

const Btn = (props) => {
  return (
    <button onClick={props.getList}>click</button>
  )
}

const mapDispatchToProps = (dispatch) => {
  return {
    getList: () => {
      dispatch(fetchList)
    }
  }
}

export default connect(null, mapDispatchToProps)(Btn)
```

4. 获取在 store 中的 lis t数据

```jsx
import { connect } from 'react-redux'

const Text = (porps) => {
```

```
    return (<div>
      <ul>
        {
          props.list?.length ? props.list.map((item, index) => (
            <li key={index}>{item.title}</li>
          ))
        }
      </ul>
    </div>)
  }
  const mapStateToProps = (state) => {
    return state
  }
  export default connect(mapStateToProps, null)(Text)
```

安装

- 直接安装在浏览器环境下基于react-router的模块

```
yarn add react-router-dom
```

路由器

- 也就是路由的模式，一种是 `Browerhistory` 模式，一种是 `hashHistory` 模式
  1. `BrowerHistory` ，指向是真实的URL地址，在上传到服务器后，访问url的时候，会去访问真实的地址，但是该地址下没有当前资源会发生404的错误
  2. `HashHistory` ，如 `http://locahost:8000/#/` ,在# 后面的发送请求会自动忽略
- 在react中也是以组件的形式存在，需要导入，并包裹整个根组件

匹配

使用 `Switch` ，`Route` 两个组件来配合使用， `Switch` 就表示是切换的意思，要根据 `Route` 里面的属性path来匹配相应的组件，组件要么放在 `Route` 里面，要么以传入到属性 `component` 中

jsx

```
  const Test = props => {
    return (<div>Test<div>)
```

```
}

<Switch>
  <Route path="/test">
    <Test />
  </Route>
  // 或者
  <Route path="/test" component={Test}></Route>
</Switch>
```

> ! 从路由头开始匹配，而 `/` 会匹配所有的路由，一般将 `/` 的路由放在最后面，或者使用精准匹配(在路由Route中添加 `exact` 即可)的方式来解决

导航

- 路由跳转， `react-router-dom` 提供一些组件，来进行视图上的跳转
- 组件有 `Link` ， `NavLink` ， `Redirect`

> `Link` 会渲染成 `<a></a>` 的形式， `NavLink` 类似，不过有个激活状态表示，当路由是当前的 `NavLink` 的时候，会获得指定的类名

使用

基本使用

jsx

Hooks

useParams

- 获取参数

useRouteMatch

- 获取当前匹配到的路由信息
- 传入参数，可以对参数与当前匹配到的路由信息进行匹配，如果匹配成功会返回路由信息，匹配失败就会返回null