**flux**

1. **简介：**

**2013和react一同诞生，flux和react的出现，代替了老一辈的backbone.js和ember.js等mvc框架，但是flux并不是mvc框架，并推翻了传统的mvc；**

**之前的前端mvc和服务端的mvc有一定的区别，并不是每次的页面改变都经历一次闭环的，而是出现了很多的v和m直接交互的场景，这样就造成了数据的比较难维护；而flux的出现让前端的应用更像服务端mvc，采用单向数据流的模式，每一次页面的改变都是一个闭环，action相当于用户的请求，而dispatcher相当于controller，store相当于Model,Flux相当于view；这样就形成了一个如果想要改变页面必须经过action动作；**

**dispatcher:处理动作分发，维持store之间的依赖关系；**

**store：负责存储数据和处理数据相关逻辑；**

**action:驱动dispatcher的js对象；**

**view：视图；**

1. **应用：**
2. **手动搭建一个不依赖任何插件和flux的flux框架**

**flux的配置的文件：**

1. **当store改变时，通知view重新获取数据（采用观察者模式）**

export var event = {

eventObj: {},

$on(eventName, fn) {

if (this.eventObj[eventName] === undefined) {

this.eventObj[eventName] = []

}

this.eventObj[eventName].push(fn)

},

$emit() {

let eventName = arguments[0]

let dataArr = [].concat(Array.prototype.slice(arguments, 1))

this.eventObj[eventName].forEach(function(i) {

i(...dataArr)

}, this);

},

destroy(eventName) {

if (this.eventObj[eventName] !== undefined) {

delete this.eventObj[eventName]

}

}

}

1. **创建当前应用的数据存储中心和数据改变逻辑函数注册**

export var state = {

num: 1,

getnum() {

return this.num

},

addNum(i) {

this.num += i

console.log(this.num)

}

}

1. **创建一个可以进行动作分发的dispatch函数**

export var dispatch = function(actions) {

switch (actions.type) {

case "addNum":

{

state.addNum(actions.text)

}

}

event.$emit("change")

}

1. **创建一个在view层可以进行执行的动作**

export var actions = {

addNum(text) {

dispatch({

type: "addNum",

text: text

})

}

}

**在页面中引用：**

**显示页面：**

import {event,state} from "../flux"

constructor(props) {

super(props)

// 自己封装

this.state={

num:state.getnum()

}

//每次数据改变后的通知

event.$on("change",()=>{

this.setState({

num:state.getnum()

})

})

}

**触发页面：**

import {actions} from "../../flux/flux.js"

render() {

return (

<div>

<p>这里是子组件</p>

<button onClick={()=>actions.addNum(3)}>点击增加</button>

</div>

)

}

1. **使用facebook的flux插件进行flux架构的开发：**
2. // 创建的store文件

import { EventEmitter } from "events"

export const store = Object.assign(EventEmitter.prototype, {

num: 1,

getnum() {

return this.num

},

addNum(i) {

this.num += i

// console.log(this.num)

},

// 跟view层的渲染有关事件处理函数

emitChange() {

this.emit("change")

},

onChange(cb) {

this.on("change", cb)

},

deleteChange(cb) {

this.removeListener("change", cb)

}

})

1. // 单独引入flux的dispatcher对象

import { Dispatcher } from "flux"

export var dispatcher = new Dispatcher()

export const dispatchToken = dispatcher.register((action) => {

switch (action.type) {

case "addNum":

{

store.addNum(action.text)

store.emitChange()

}

break;

}

})

1. // 创建的actions文件

export const actions = {

addNum(text) {

console.log(text)

dispatcher.dispatch({

type: "addNum",

text: text

})

}

}

**在页面中引用：**

**显示页面：**

import {event,state} from "../flux"

constructor(props) {

super(props)

this.state={

num:store.getnum()

}

store.onChange(()=>{

this.setState({

num:store.getnum()

})

})

}

**触发页面：**

import {actions} from "../../flux/flux.js"

render() {

return (

<div>

<p>这里是子组件</p>

<button onClick={()=>actions.addNum(3)}>点击增加</button>

</div>

)

}

1. **数据的只可读性处理**

* **使用symbol和get函数实现**

**store文件：**

// 用私有属性（symbol实现state只读）

const num=Symbol()

export const store = Object.assign(EventEmitter.prototype, {

[num]: 1, //私有属性

getnum() {

return this[num]

},

addNum(i) { //页面中获取时，调用此函数

this[num] =this[num]+ i

// console.log(this.num)

},

// 跟view层的渲染有关事件处理函数

emitChange() {

this.emit("change")

},

onChange(cb) {

this.on("change", cb)

},

deleteChange(cb) {

this.removeListener("change", cb)

}

})

**显示页面：**

// 使用symbol借助flux插件实现数据只读

// store[Symbol()]++

console.log(store)

this.state={

num:store.getnum()

}

store.onChange(()=>{

this.setState({

num:store.getnum()

})

})

* **使用defineProperty属性实现：**

**store文件：**

const defineSymbol=Symbol()

const state = {

num: 1,

addNum(i) {

this[defineSymbol]("num",this.num+i)

},

[defineSymbol](arg,result){

Object.defineProperty(this,[arg],{

value:result

})

}

}

Object.keys(state).forEach(function(i) {

Object.defineProperty(state,i,{

writable:false

})

});

export let store=Object.assign(state,EventEmitter.prototype,{

emitChange() {

this.emit("change")

},

onChange(cb) {

this.on("change", cb)

},

deleteChange(cb) {

this.removeListener("change", cb)

}

})

**在页面中调用：（展示页面）**

this.state={

num:store.num

}

store.onChange(()=>{

this.setState({

num:store.num

})

})

1. **flux的不足之处：**

**无法自动刷新，每一次都需要手动的在页面进行事件的监听，从而改变store的值；**

**store中混杂了逻辑与状态，在开发过程不停的更换store的逻辑，容易出现bug；**

1. **flux的优势：**

**传统前端的mvc无法禁绝view和model之间的通讯问题，而flux的中store，在只有get方法，在页面中只能读取数据，而无法直接更改数据，每次数据的更变都必须通过action派发一个事件给dispatcher，由dispatcher去改变；**

**redux**

1. **简介：**

**如果把flux看成一个架构理念，那么redux就是flux的一种实现，除了redux之外，还有很多flux的框架，比如reflux、fluxible等等；2013年flux问世之后饱受争议，所以在2015年redux出现；在flux中只是提出了单向数据流，而redux在此基础上提出三个原则：**

**唯一数据源：**在redux的应用中只能有一个state数据源；

**保持状态只读:**在view层只能对state数据进行读取，无法改变；

**数据改变只能通过纯函数完成**:reducer函数只做数据的运算的不做数据的存储，所有的reducer函数只会接受值，进行运算，redux中的reducer接受两个参数，第一个参数是state当前的数据状态，第二个参数是action传入的值；将原始值和传入进行运算；

1. **理论讲解：**
2. **action:**
3. **定义：Action** 是把数据从应用（译者注：这里之所以不叫 view 是因为这些数据有可能是服务器响应，用户输入或其它非 view 的数据 ）传到 store 的有效载荷。它是 store 数据的**唯一**来源。一般来说你会通过 [store.dispatch()](http://www.redux.org.cn/docs/api/Store.html" \l "dispatch) 将 action 传到 store。
4. **action 内必须使用一个字符串类型的 type 字段来表示将要执行的动作**
5. **redux中的action和flux中的action的区别：**

在 Redux 中的 action 创建函数只是简单的返回一个 action，redux中action只需要将返回的对象传递给reducer函数去做处理；

在 [传统的 Flux](http://facebook.github.io/flux" \t "_blank) 实现中，当调用 action 创建函数时，一般会触发一个 dispatch

1. **reducer:**
2. **state的设计（应用数据库）**

尽可能地把 state 范式化，不存在嵌套。把所有数据放到一个对象里，每个数据以 ID 为主键，不同实体或列表间通过 ID 相互引用数据。把应用的 state 想像成数据库

1. **reducer函数的设计：**

保持 reducer 纯净非常重要。**永远不要**在 reducer 里做这些操作：

修改传入参数；

执行有副作用的操作，如 API 请求和路由跳转；

调用非纯函数，如 Date.now() 或 Math.random()。

**注意：**

在reducer中不要直接修改state，**。** 使用 [Object.assign()](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign" \t "_blank) 新建了一个副本。不能这样使用 Object.assign(state, {visibilityFilter: action.filter })，因为它会改变第一个参数的值。你**必须**把第一个参数设置为空对象。你也可以开启对ES7提案[对象展开运算符](http://www.redux.org.cn/docs/recipes/UsingObjectSpreadOperator.html)的支持, 从而使用 { ...state, ...newState } 达到相同的目的。

1. **对于reducer文件的拆分以及使用combineReducers()工具类**

随着应用的膨胀，我们还可以将拆分后的 reducer 放到不同的文件中, 以保持其独立性并用于专门处理不同的数据域。**每个 reducer 只负责管理全局 state 中它负责的一部分。每个 reducer 的**state**参数都不同，分别对应它管理的那部分 state 数据**

**注：**[combineReducers()](http://www.redux.org.cn/docs/api/combineReducers.html) 所做的只是生成一个函数，这个函数来调用你的一系列 reducer，每个 reducer **根据它们的 key 来筛选出 state 中的一部分数据并处理**，然后这个生成的函数再将所有 reducer 的结果合并成一个大的对象。

**注：这里其实实现的就是类似于vuex中模块式管理数据逻辑层；**

1. **store:**

维持应用的 state；

提供 [getState()](http://www.redux.org.cn/docs/api/Store.html" \l "getState) 方法获取 state；

提供 [dispatch(action)](http://www.redux.org.cn/docs/api/Store.html" \l "dispatch) 方法更新 state；

通过 [subscribe(listener)](http://www.redux.org.cn/docs/api/Store.html" \l "subscribe) 注册监听器;

通过 un[subscribe(listener)](http://www.redux.org.cn/docs/api/Store.html" \l "subscribe) 返回的函数注销监听器

**注：Redux 应用只有一个单一的 store**。当需要拆分数据处理逻辑时，你应该使用 [reducer 组合](http://www.redux.org.cn/docs/basics/Reducers.html" \l "splitting-reducers)而不是创建多个 store。

1. **实例应用：**
2. **手动搭建redux（单一文件搭建）：**

import { createStore } from "redux"

// action的类型命名管理

export const actionType = {

ADD: "add"

}

// state数据管理中心

const initState = {

num: 0

}

// 创建reducer纯函数

const reducer = (state = initState, action) => {

const { type, arg, text } = action

switch (type) {

case actionType.ADD:

{

const newnum = state[arg] + text

return {...state, [arg]: newnum }

}

default:

return state

}

}

// 抛出实例化后的store

export const store = new createStore(reducer)

// 创建action对象函数

export const actions = {

[actionType.ADD](i) {

return {

type: actionType.ADD,

arg: "num",

text: i

}

}

}

1. **多文件分离构造redux：**
2. **创建actionType文件：（用于存储所有的actions类型）**

// 针对某一个模块

export const ADD = "add"

// 针对另一个模块

export const LIST = "list"

1. **创建actions文件（用于存储所有的动作，提交到store的所有载荷）**

import { ADD, LIST } from "./ActionType"

// 其中一个模块的actions

export const addAction = (text) => {

return {

type: ADD,

arg: "num",

text: text

}

}

// 另外一个模块的actions

export const listAction = (text) => {

return {

type: LIST,

arg: "list",

text: text

}

}

1. **创建reducer文件：（更新state的纯函数）**

（使用combineReducers进行合并reducer）

import { ADD, LIST } from "./ActionType"

// 初始数据的存储

const initialState = {

num: 0,

list: [1, 2, 3]

}

import { combineReducers } from "redux"

// add模块

let addReduce = (state = initialState.num, action) => {

const { arg, type, text } = action

switch (type) {

case ADD:

{

console.log(22)

const num = state + text

return num

}

default:

return state

}

}

// list模块

let listReduce = (state = initialState.list, action) => {

const { arg, type, text } = action

switch (type) {

case LIST:

{

let arr = [...state]

arr.push(text)

return arr

}

default:

return state

}

}

// 用combineReducers进行多个reducer进行合并

const app = combineReducers({

num: addReduce,

list: listReduce

})

export default app

1. **创建store文件：（实例化redux的store，并存放初始state数据）**

import { createStore } from "redux"

import reducer from "./reducer.js"

export const store = createStore(reducer)

1. **显示页面的调用：**

import {store} from "../redux/store"

constructor(props) {

super(props)

this.onchange=this.onchange.bind(this)

this.state={

list:store.getState().list,

num:store.getState().num

}

store.subscribe(this.onchange)

}

onchange(){

this.setState({

list:store.getState().list,

num:store.getState().num

})

}

render() {

let $li=this.state.list.map((i,index)=>{

return <li key={index}>{i}</li>

})

return (

<div >

<span>主页</span>

<span>{this.state.num}</span>

<ul>

{$li}

</ul>

<Children></Children>

< /div>

)

}

componentWillUnmount(){

store.unsubscribe(this.onchange)

}

1. **调用页面的代码：**

import {store} from "../../redux/store"

import {addAction,listAction} from "../../redux/actions"

addfn(i){

store.dispatch(addAction(i))

// store.dispatch(actions[actionType.ADD](i))

}

addlistfn(i){

store.dispatch(listAction(i))

// store.dispatch(actions[actionType.ADD](i))

}

1. **react与redux的搭配：**

**声明：**Redux 和 React 之间没有关系。Redux 支持 React、Angular、Ember、jQuery 甚至纯 JavaScript。

在react中使用redux，需要安装react-redux；

npm install --save react-redux

**容器组件：（顶层组件）**

只在最顶层组件（如路由操作）里使用 Redux，数据的读取和更改都来自redux；

**展示组件：（傻瓜组件）**

其余内部组件仅仅是展示性的，所有数据都通过 props 传入。数据的读取和修改来自于props的数据和props传递的回掉函数；

**react和redux的链接：**

我们需要做出两个变化，将 App 组件连接到 Redux 并且让它能够 dispatch actions 以及从 Redux store 读取到 state。

首先，我们需要获取从之前安装好的 [react-redux](http://github.com/reactjs/react-redux" \t "_blank) 提供的 Provider，并且在渲染之前**将根组件包装进**<Provider>。

接着，我们**想要通过**[react-redux](http://github.com/reactjs/react-redux" \t "_blank)**提供的**connect()**方法将包装好的组件连接到Redux**。尽量只做一个顶层的组件，或者 route 处理。从技术上来说你可以将应用中的任何一个组件 connect() 到 Redux store 中，但尽量避免这么做，因为这个数据流很难追踪。

**任何一个从**connect()**包装好的组件都可以得到一个**[dispatch](http://www.redux.org.cn/docs/api/Store.html" \l "dispatch)**方法作为组件的 props，以及得到全局 state 中所需的任何内容。** connect() 的唯一参数是 **selector**。此方法可以从 Redux store 接收到全局的 state，然后返回组件中需要的 props。最简单的情况下，可以返回一个初始的 state （例如，返回认证方法），但最好先将其进行转化。

**实例代码如下：**

**actions：**

import { ADD, LIST } from "./ActionType"

// 其中一个模块的actions

export const addAction = (text) => {

return {

type: ADD,

arg: "num",

text: text

}

}

// 另外一个模块的actions

export const listAction = (text) => {

return {

type: LIST,

arg: "list",

text: text

}

}

**reducer：**

import { ADD, LIST } from "./ActionType"

// 初始数据的存储

const initialState = {

num: 0,

list: [1, 2, 3]

}

import { combineReducers } from "redux"

// add模块

let addReduce = (state = initialState.num, action) => {

const { arg, type, text } = action

switch (type) {

case ADD:

{

console.log(22)

const num = state + text

return num

}

default:

return state

}

}

// list模块

let listReduce = (state = initialState.list, action) => {

const { arg, type, text } = action

switch (type) {

case LIST:

{

let arr = [...state]

arr.push(text)

return arr

}

default:

return state

}

}

// 用combineReducers进行多个reducer进行合并

const app = combineReducers({

num: addReduce,

list: listReduce

})

export default app

**actionsType:**

// 针对某一个模块

export const ADD = "add"

// 针对另一个模块

export const LIST = "list"

**容器组件：**

import React, { Component } from 'react'

import Children from "./base/children.jsx"

import {connect} from "react-redux"

class Main extends Component {

constructor(props) {

super(props)

}

render() {

let $li=this.props.list.map((i,index)=>{

return <li key={index}>{i}</li>

})

return (

<div >

<span>主页</span>

<span>{this.props.num}</span>

<ul>

{$li}

</ul>

<Children dispatch={this.props.dispatch}></Children>

< /div>

)

}

componentWillUnmount(){

store.unsubscribe(this.onchange)

}

}

const select=(state)=>{

let obj=Object.assign({},state)

return obj

}

const containerName = connect(

select

)(Main)

export default containerName

**显示组件：**

import React, { Component } from 'react'

import {addAction,listAction} from "../../redux/actions"

// import {store,actions,actionType} from "../../redux/index.js"

class Children extends Component {

constructor(props) {

super(props)

console.log(this.props)

}

addfn(i){

this.props.dispatch(addAction(i))

// store.dispatch(actions[actionType.ADD](i))

}

addlistfn(i){

this.props.dispatch(listAction(i))

// store.dispatch(actions[actionType.ADD](i))

}

render() {

return (

<div>

<p>这里是子组件</p>

<button onClick={()=>this.addfn(3)}>点击增加</button>

<button onClick={()=>this.addlistfn(3)}>点击增加数组</button>

</div>

)

}

}

export default Children

1. **异步action：**
2. **创建store文件时：**

import { connect } from "react-redux"

import { createStore, combineReducers, applyMiddleware } from "redux"

import { tmReducers } from "./tm/"

import { zfbReducers } from "./zfb/"

import thunkMiddleware from 'redux-thunk'

const appReducers = combineReducers({

tm: tmReducers,

zfb: zfbReducers

})

export default new createStore(appReducers, applyMiddleware(thunkMiddleware))

1. **请求数据的actions：**

// 数据请求的action，不同状态下再次去请求其他的actions

export const t\_fetch = () => {

return (dispatch) => {

// 数据请求开始

dispatch(t\_fetchStart())

fetch("/some/path").then((data) => {

data.json().then((redata) => {

console.log(redata)

// 数据请求成功

dispatch(t\_fetchOk(redata.custom))

})

}).catch((error) => {

// 数据请求失败

dispatch(t\_fetchNo())

})

}

}