<https://blog.csdn.net/qq_39261142/article/details/107310638>

**redux的使用技巧**



[香香鸡](https://blog.csdn.net/qq_39261142) 2020-07-13 09:41:46  41  收藏

分类专栏： [react](https://blog.csdn.net/qq_39261142/category_9980188.html) [前端](https://blog.csdn.net/qq_39261142/category_9977194.html) 文章标签： [reactjs](https://www.csdn.net/tags/OtTaQg3sNDQ1LWJsb2cO0O0O.html) [javascript](https://www.csdn.net/tags/OtDaQg2sNzExLWJsb2cO0O0O.html)

版权

**Redux**

Redux在任何地方都可用，不只是运用于React

redux用于state的管理，管理的是在整个网站各个地方都通用的state，但是不去管理那些只作用于组件自身的state

npm install redux -save

**作用：将特定的state属性全局向需要用到该state的组件传递，该state属性全局唯一（类似static属性），只能通过action进行修改，且修改后影响全局**

**要点**

* 项目中所有的state都以一个对象树的形式存储在一个单一的store中
* 只有触发action才能更改state
* 通过编写reducers来实现action更改state的逻辑

import { createStore } from 'redux'

// 创建一个store，createStore必须接受一个reducer，reducer为一个方法

let store = createStore(counter)

// 声明一个reducer，接收旧的state和action，根据action来制定修改state的逻辑代码并返回新的state

const state = 0

function counter(state, action) {

switch(action.type) {

case '+':

return state + 1

case '-':

return state - 1

default:

return state // 因为不是任何时候state都需要修改的，因此需要不改变state的出口

}

}

// action是一个普通的JS对象，普遍约定action内必须使用一个string的type字段来表示将要执行的动作

const add = {type: '-'}

const dec = {type: '+'}

// 改变内部state的唯一方法树dispatch一个action

store.dispatch(add) // -1

store.dispatch(dec) // 0

// 创建监听，一般用于更改state时进行渲染更新

store.subscribe()

**store**

store = createStore(reducer)

store.getState() // 获取当前state内容

store.dispatch(action) // 方法更新state

store.subscribe(listener) // 注册监听器，会返回一个可以解绑监听器的函数，执行该函数则会解绑

* **store.getState( )**

返回当前应用的state树，与store的最后一个reducer返回值相同，注意，这个state是所有reducer传入的state，即获取到了所有全局的state的值

* **store.dispatch(action)**

触发state变化的唯一途径，会使用当前getState()和传入的action以同步方式调用store的reduce函数，返回值会被作为下一个state，成为getState()的新返回值，同时变化监听函数（change listener）被触发

如果action是一个函数，应该去执行，即store.dispatch(action())，因为执行了创建方法这样才能返回一个action对象

* **store.subscribe(listener)**

每当执行store.dispatch(action)时便会自动执行此方法，需要操作当前state时可用store.getState()获取当前state

**action**

action在声明reducer时并没有将具体的的action传入，真正将具体action传入reducer是在调用dispatch（action）的时候

使用action的两种方法：

* 直接使用一个对象

const action = {

type:''

}

* action创建函数，使用一个函数返回一个对象（约定传入参数放进payload属性里）

const action = (prams) => {

return {

type:''

payload: {

prams

}

}

}

**reducers**

**通过触发store的dispatch方法，会触发reducer方法，然后reducer就会拿到dispatch传入的action，再对action内的值进行判断并通过reducer更改state（会触发所有的reducer）**

* **设计state结构**

redux中所有state被保存在一个单一对象中，不同类型的state需要想办法进行区分

{

visibilityFilter: 'SHOW\_ALL',

todos: [

{

text: 'Consider using Redux',

completed: true,

},

{

text: 'Keep all state in a single tree',

completed: false

}

]

}

* **保证reducer纯净**

不可在reducer里执行如下操作：

* + 修改传入参数
  + 执行有副作用的参数，如API请求和路由跳转
  + 调用非纯函数，如Data.now()或Math.random()

只要传入参数相同，返回计算得到的下一个state就一定相同，没有特殊情况，没有副作用，没有API请求，没有变量修改，单纯执行计算

reducer的最终目的只是接受一个条件并根据条件修改state，其他不是以修改state为目的的操作应该都在外部操作，如action

* **Action**

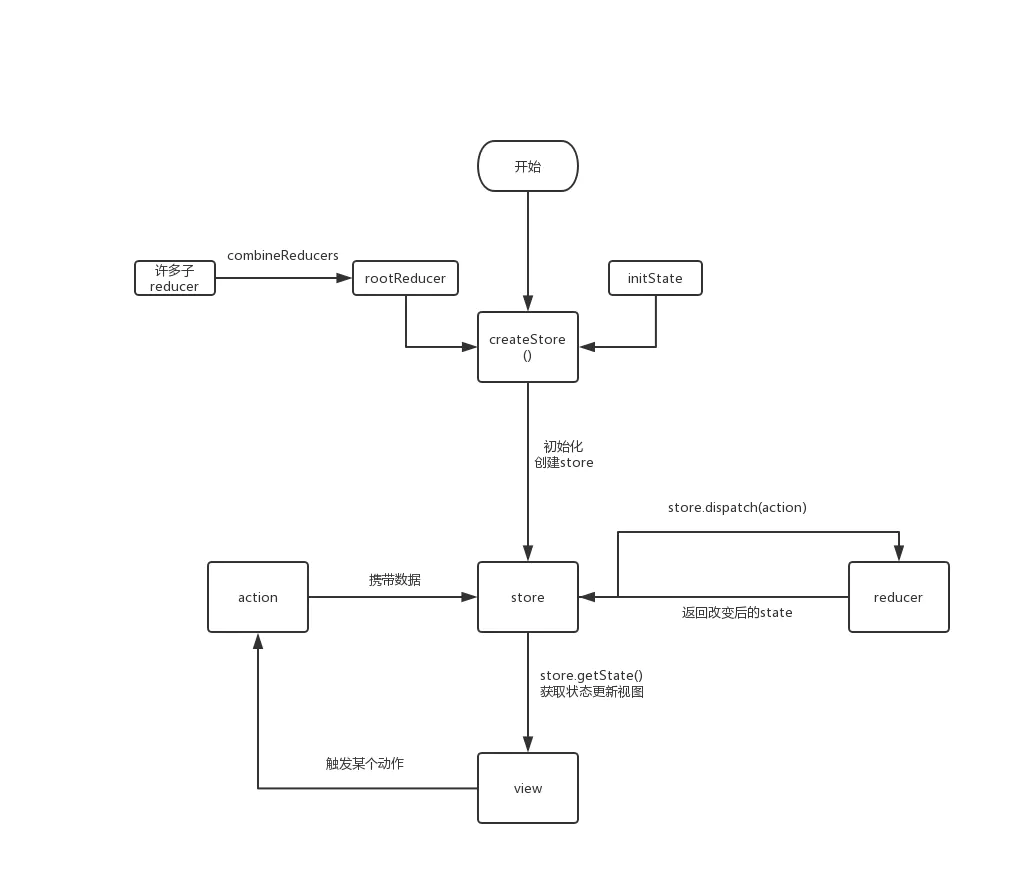
redux首次执行时，state为undefined，可另设置初始state

function todoApp(state = initialState, action) {

// 这里暂不处理任何 action，仅返回传入的 state

return state

}



**项目搭建**

* **管理reducers**  
  在项目src目录下创建reducers文件夹，用于统一存放reducer文件，创建index.js文件，用于统一导入reducer文件并导出

import reducerA from './reducerA'

import reducerB from './reducerB'

import {combineReducers} from 'redux' // redux专门用来提供合并reducers的工具

export default combineReducers({

reducerA, // 相当于 reducerA: rerducerA

reducerB

})

/\*

不可以直接导出，这样导出的不是一个方法，无法被识别

export default {

reducerA,

reducerB

}

\*/

​ 一开始进行reducer的编写时，只需写明state和声明一个简单方法导出即可，后续再加功能

const reducerA\_state = {

// ...

}

export default (state = reducerA\_state, action) => {

switch(action.type) {

default:

return state

}

}

* **管理store**

在项目的src目录下创建store.js文件（注意只能存在一个store）

import {createStore} from 'redux'

import rootReducer from './reducers'

export default createStore(rootReducer)

* **管理actions**

在项目src目录下创建actions文件夹，用于统一存放action文件，一般新建一个actionType.js文件统一对action的type进行管理并导出，然后另创建具体的action.js文件使用action创建函数的方式导出具体action

如：实现一个商品计数器的action

// actionType.js

export default {

CART\_AMOUNT\_INCREMENT: 'CART\_AMOUNT\_INCREMENT',

CART\_AMOUNT\_DECREMENT: 'CART\_AMOUNT\_DECREMENT'

}

// /src/actions/cart.js

import actionType from './actionType'

export const increment = (id) =>{

return {

type: actionType.CART\_AMOUNT\_INCREMENT,

payload: {

id

}

}

}

export const decrement = (id) => {

return{

type: actionType.CART\_AMOUNT\_DECREMENT,

payload: {

id

}

}

}

​ 在reducer里导入的是actionType.js，在使用store.dispatch( )方法的组件中导入cart.js

// /src/reducers/cart.js

import actionType from '../actions/actionType'

const initState = [{

id: 1,

title: 'Apple',

price: 8888.66,

amount: 10

},{

id: 2,

title: 'Orange',

price: 4444.66,

amount: 12

}]

export default (state = initState, action) => {

switch(action.type) {

case actionType.CART\_AMOUNT\_INCREMENT:

return state.map(item => {

if(item.id === action.payload.id) {

item.amount += 1

}

return item

})

default:

return state

}

}

import store from '../../store'

import {increment, decrement} from '../../actions/cart'

// ...

<button onClick = {

() => {

this.store.dispatch(decrement(id))

}

}/>

<button onClick = {

() => {

this.store.dispatch(increment(id))

}

}/>

**recat-redux**

npm install react-redux -s

react-redux是redux对react的官方绑定库，借助react-redux可以很方便的在react中使用redux

redux只对外暴露了以下方法，除了这些都是react-redux的内容

export {

createStore,

combineReducers,

bindActionCreators,

applyMiddleware,

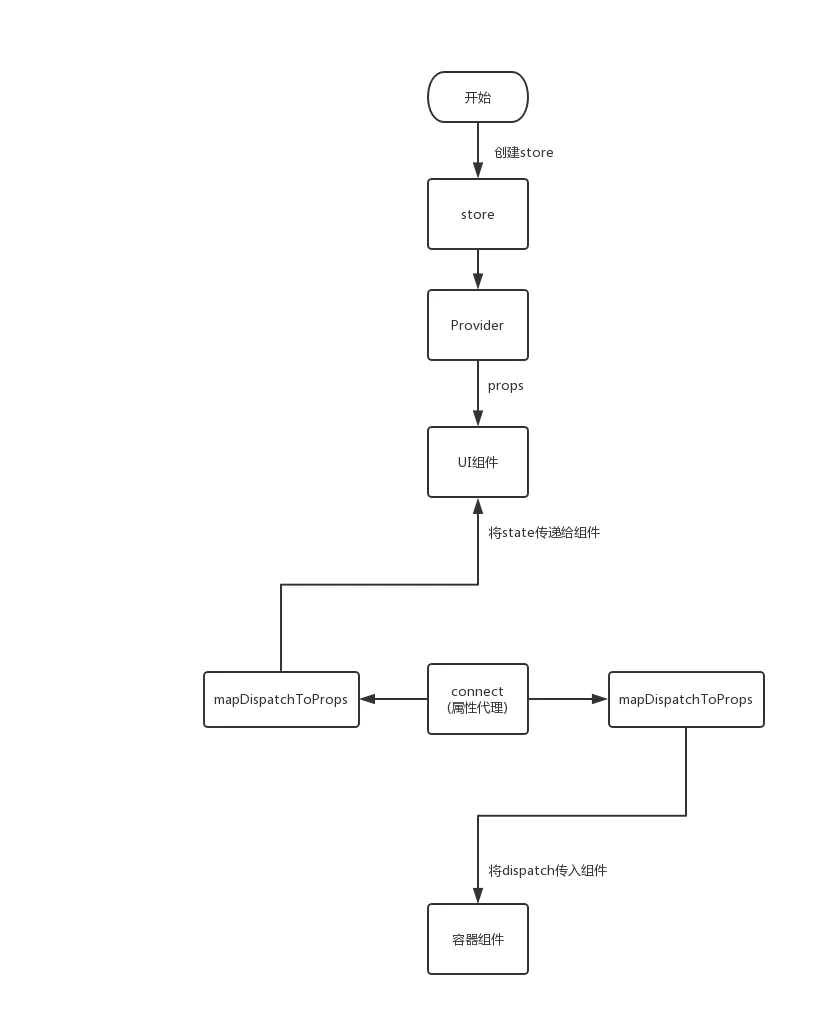
compose,

\_\_DO\_NOT\_USE\_\_ActionTypes

}

**使用步骤：**

1. 通过createStore()方法创建store；
2. 通过Provider组件将store注入到需要使用store的组件中；
3. 通过connect()连接UI组件和容器组件，从而更新state和dispatch(action)



**运用思想**

实际项目中，需要权衡是直接使用Redux还是用React-Redux

* React-Redux 将所有组件分成两大类：UI 组件（presentational component）和容器组件（container component）
* UI 组件负责 UI 的呈现，容器组件负责管理数据和逻辑，如果一个组件既有 UI 又有业务逻辑，则将它拆分成下面的结构：外面是一个容器组件，里面包了一个UI 组件，前者负责与外部的通信，将数据传给后者，由后者渲染出视图
* React-Redux 规定，所有的 UI 组件都由用户提供，容器组件则是由 React-Redux 自动生成，也就是说，用户负责视觉层，状态管理则是全部交给它

**UI组件**

* 只负责 UI 的呈现，不带有任何业务逻辑
* 没有状态（即不使用this.state这个变量）
* 所有数据都由参数（this.props）提供
* 不使用任何 Redux 的 API

**容器组件**

* 负责管理数据和业务逻辑，不负责 UI 的呈现
* 带有内部状态
* 使用 Redux 的 API

|  | **UI组件** | **容器组件** |
| --- | --- | --- |
| **作用** | 描述如何展现（骨架、样式） | 描述如何运行（数据获取、状态更新） |
| **直接使用 Redux** | 否 | 是 |
| **数据来源** | props | 监听 Redux state |
| **数据修改** | 从 props 调用回调函数 | 向 Redux 派发 actions |
| **调用方式** | 手动 | 通常由 React Redux 生成 |

**Provider和connect**

**Provider**

使用provider进行快捷组件交流，不需要多层传递数据

使用时需要将其包在要传递组件的最外层，由于<App/>为组件渲染根元素，因此任意子组件内都可直接使用provider传递的数据，无需层层相传，必须要要拥有store属性，值为创建的store

// /src/index.js

import React from 'react'

import { render } from 'react-dom'

import App from './App'

import store from './store'

import {Provider} from 'react-redux'

render(

<Provider store={store}>

<App/>,

</Provider>,

document.querySelector('#root')

)

**connect**

通过connect( )( )自动生成的容器组件（高阶组件），经过connect操作后会将dispatch方法传入该组件

使用了connect之后是自动订阅state的变化并进行重新渲染的，不需要再去通过store.subscribe(listener)去监听state的变化

import {connect} from 'react-redux'

export default VisibleMyComponent = connect()(myComponent) // 一般合起来写，作用和下面的一样

/\*

const VisibleMyComponent = connect()(myComponent)

export default VisibleMyComponent

使用装饰器的写法：

@connect() // connect()()有两层括号，使用装饰器后会少一个

class myComponent {...}

export default myComponent

\*/

class componentA {} // props为传递进来的组件

@connect()

class componentA {} // props为传递进来的组件和dispatch方法

connect( )接受两个参数：mapStateToProps和mapDispatchToProps，它们定义了 UI 组件的业务逻辑，前者负责输入逻辑，即将state映射到 UI 组件的参数（props），后者负责输出逻辑，即将用户对 UI 组件的操作映射成 Action

const VisibleMyComponent = connect(

mapStateToProps,

mapDispatchToProps

)(myComponent)

* **mapStateToProps()**

它是一个函数，建立一个从（外部的）state对象到（UI 组件的）props对象的映射关系，它接受state作为参数，返回一个对象

当在connect绑定函数时，对应的函数会自动传进来一个值，这个state为store.getState()的值，当在connect绑定该方法后，此组件的props将会在原有props的基础上添加该函数return对象里的属性

在组件renser()前被执行，因此每次render后的组件状态都与Store同步

// 若原组件的props内容为{a:1}

// 使用connect高阶组件化并设置mapStateToProps的组件props为{a:1, addPrpps:xxx, dispatch: function

const mapStateProps = (state, ownProps) => { // ownProps为传进该组件的的props

return {

addProps:state.myState

}

}

* **mapDispatchToProps()**

它可以是一个函数，也可以是一个对象，定义了哪些用户的操作应该当作 Action传给 Store，用来建立 UI 组件的参数到store.dispatch方法的映射，即将原本的this.props.dispatch()映射为返回的对象

在组件constructor()中被执行，因而只执行一次

为一个函数时：则本来应使用this.props.dispatch()方法，现在因为有了映射，可以直接this.props.add()和this.props.dec()执行对应的reducer功能

const increment = (id) =>{

return {

type: actionType.CART\_AMOUNT\_INCREMENT,

payload: {

id

}

}

}

const decrement = (id) => {

return{

type: actionType.CART\_AMOUNT\_DECREMENT,

payload: {

id

}

}

}

// 第一个参数用于接受store.dispatch()方法（可以为其它变量名）,第二个参数用于接受组件自身的props

const mapDispatchToProps = (dispatch, ownProps) => {

return {

add: (id) => { dispatch(increment(id)) },

dec: (id) => { dispatch(decrement(id)) }

}

}

​ 为一个对象时，键值内容应为一action对象或action创建函数，会主动识别传入的action并自动调用dispatch(action)方法并将返回值映射到props，此时调用dispatch方法时则使用this.props.increment()和this.props.decrement()，而非this.props.dispatch(this.props.increment())

const mapDispatchToProps = {

increment: (id) => {

return {

type: 'CART\_AMOUNT\_INCREMENT',

payload: {

id

}

}

},

decrement : (id) => {

return{

type: 'CART\_AMOUNT\_DECREMENT',

payload: {

id

}

}

}

}

**一般是在action先写好了mapStateToProps和mapDispatchToProps再导出到组件并传到connect里**

**▼演示范例：**

// 为了展示方便全写在一个文件里，实际在项目中需要分类各个文件

import React, { Component } from 'react'

import ReactDOM from 'react-dom'

import { createStore } from 'redux'

import { Provider, connect } from 'react-redux'

// 定义counter组件

class Counter extends Component {

render() {

const { value, onIncreaseClick } = this.props // 解构

return (

<div>

<span>{value}</span>

<button onClick={onIncreaseClick}>自增按钮</button>

</div>

)

}

}

// Action

const increaseAction = { type: 'increase' }

// Reducer 基于原有state根据action得到新的state

function counter(state = { count: 0 }, action) {

switch (action.type) {

case 'increase':

return { count: state.count + 1 }

default:

return state

}

}

// 根据reducer函数通过createStore()创建store

const store = createStore(counter)

// 将state映射到Counter组件的props

function mapStateToProps(state) {

return {

value: state.count

}

}

// 将action映射到Counter组件的props

function mapDispatchToProps(dispatch) {

return {

onIncreaseClick: () => dispatch(increaseAction)

}

}

// 传入上面两个函数参数，将Counter组件变为App组件

const App = connect(

mapStateToProps,

mapDispatchToProps

)(Counter)

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

)

**中间件处理异步操作**

【这里的异步操作通常是指在action中执行ajax请求】

异步action（因为要保证reducer的纯净，因此不能外调api，所以在action里处理）

运行原理：actionCreator => 自动dispatch(actionCreator()) => reducer => store => view

// 当使用了connect，进行异步操作时，因为connect会立即dispatch设置的action，但是当异步return action时，就无法在第一时间找到dispatch的action，因此会报错，不能这样写

const action = () => {

setTimeout(() => {

return {

type: xxx,

}

},2000)

}

* 1

**为了能实现能获取到异步返回的action，需要使用redux的中间件middeware进行处理**

**applyMiddleware**

import { applyMiddleware } from 'redux'

作用：加载 middleware，将 action 进行多层组合，并且将dispatch和getState方法传入到 action 中，使 action 能通过 dispatch 向下调用新的 action或查看当前的state状态【要配合中间件才能起作用】

const action = () =>{

return {

type: xxx

}

}

const beforeAction = () => {

return (dispatch,getState) => {

console.log(getState()) // 获取state

dispatch(action()) // 手动dispatch调用下一个action

}

}

**实现原理：**

applyMiddleware 利用 createStore 和 reducer 创建了一个 store，然后 store 的 getState 方法和 dispatch 方法又分别被直接和间接地赋值给 middlewareAPI 变量

export default function applyMiddleware(...middlewares) {

return createStore => (...args) => {

// 利用传入的createStore和reducer和创建一个store

const store = createStore(...args)

let dispatch = () => {

throw new Error(

)

}

const middlewareAPI = {

getState: store.getState,

dispatch: (...args) => dispatch(...args)

}

// 让每个 middleware 带着 middlewareAPI 这个参数分别执行一遍

const chain = middlewares.map(middleware => middleware(middlewareAPI))

// 接着 compose 将 chain 中的所有匿名函数，组装成一个新的函数，即新的 dispatch

dispatch = compose(...chain)(store.dispatch)

return {

...store,

dispatch

}

}

}

上面的compose方法可以接受一组函数参数，从右到左来组合多个函数，然后返回一个组合函数

即compose(funcA, funcB, funcC)等价于compose(funcA(funcB(funcC())))

export default function compose(...funcs) {

if (funcs.length === 0) {

return arg => arg

}

if (funcs.length === 1) {

return funcs[0]

}

return funcs.reduce((a, b) => (...args) => a(b(...args)))

}

**redux-thunk**

redux-thunk是一个处理异步事件的中间件，类似的还有redux-promise，redux-saga等中间件，使用方法和redux-thunk接近，但是会有些不同

npm install redux-thunk -s

**实现原理：**

function createThunkMiddleware(extraArgument) {

return ({ dispatch, getState }) => next => action => {

if (typeof action === 'function') {

return action(dispatch, getState, extraArgument);

}

return next(action);

};

}

const thunk = createThunkMiddleware();

thunk.withExtraArgument = createThunkMiddleware;

export default thunk;

redux-thunk 中间件的功能很简单，首先检查参数 action 的类型，如果是函数的话，就执行这个 action ，并把 dispatch, getState, extraArgument 作为参数传递进去，否则就调用 next 让下一个中间件继续处理 action

每个中间件最里层处理 action 参数的函数返回值都会影响 Store 上的 dispatch 函数的返回值，但每个中间件中这个函数返回值可能都不一样。比如上面这个 react-thunk 中间件，返回的可能是一个 action 函数，也有可能返回的是下一个中间件返回的结果。因此，dispatch 函数调用的返回结果通常是不可控的，最好不要依赖于 dispatch 函数的返回值

**使用：**

// 直接在store文件里导入即可（同时要导入使用applyMiddleware方法）

import {createStore, applyMiddleware} from 'redux'

import thunk from 'redux-thunk'

import rootReducer from './reducers'

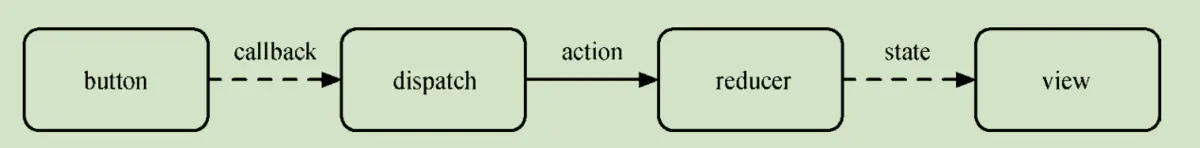
export default createStore(

rootReducer,

applyMiddleware(thunk)

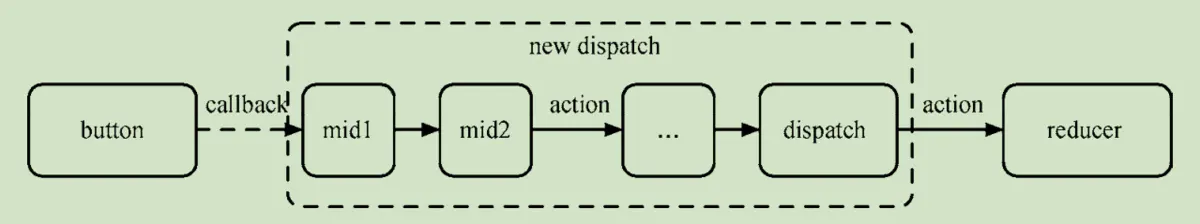
)

使用中间件处理前：



使用中间件处理后：

将action传递给中间件，每一个中间件可以处理action并返回新的action，最后一个中间件处理完action后才会将最终的action传递给reducer处理，因此保证了reducer处理的是异步执行完步后的action，不会因为立即执行dispatch而没有及时拿到到action报错



一般而言，会把同步action和异步操作分开，同步action用于定义action.type等数据，而异步只引用同步

const action = () =>{

return {

type: actionType.TYPE\_DEMO\_B

}

}

export const asynAction= () => {

setTimeout(() =>{

dispatch(action())

},2000)

}

多个中间件处理action的情况：只需导出最外层的中间件并在外部dispatch该中间件即可

const action\_1 = () => {

console.log('执行中间件1')

return {

type: actionType.TYPE\_DEMO\_C

}

}

const action\_2 = () => {

return (dispatch) => {

setTimeout(() => {

console.log('执行中间件2')

dispatch(action\_1())

},2000)

}

}

const action\_3 = () => {

return (dispatch) => {

console.log('执行中间件3')

dispatch(action\_2())

}

}

export const action\_4 = () => {

return (dispatch) => {

setTimeout(() => {

console.log('执行中间件4')

dispatch(action\_3())

},2000)

}

}

/\*

外部引入action\_4并dispatch(action\_4())，执行结果：

执行中间件4

执行中间件3

执行中间件2

执行中间件1

\*/

**使用redux-thunk在真实开发中实现数据请求的写法：**

function getWeather(url, params) {

return (dispatch, getState) => {

fetch(url, params)

.then(result => {

dispatch({

type: 'GET\_WEATHER\_SUCCESS', payload: result,

})

})

.catch(err => {

dispatch({

type: 'GET\_WEATHER\_ERROR', error: err,

})

})

}

}

可以在请求前后另外dispatch一些操作

比如在请求开始前将一个表示加载中的state设置成true，在请求完成时设置成flase，就实现需要在请求加载中时做的一些操作的效果

function getWeather(url, params) {

return (dispatch, getState) => {

dispatch({

type: 'GET\_WEATHER\_START'

})

fetch(url, params)

.then(result => {

dispatch({

type: 'GET\_WEATHER\_SUCCESS', payload: result,

})

})

.catch(err => {

dispatch({

type: 'GET\_WEATHER\_ERROR', error: err,

})

})

dispatch({

type: 'GET\_WEATHER\_END'

})

}

}