// 安装完成以后自带的一个createStore方法

import { createStore } from "redux";

// createStore 接收一个函数  可以在里面设置初始值  先当于vuex的state

const store = createStore(() => {

    const state = 0;

    return state;

});

export default store;

// 安装完成以后这个store就会有以下方法

// dispatch

// store.dispatch()可以改变值

// subscribe

// 用来重新rander

// store.subscribe(() => {

    // this.forceUpdate(); 强制刷新页面

    // this.setState({});

// });

// getState

// store.getState()可以获取到state里面的值

// replaceReducer

// Symbol

**初始化创建store 然后调用dispatch 执行action方法 store.subcribe可以监听 然后里面执行rander store.getState可以获取到值**

# connect

**class MyComp extends Component {**

**// content...**

**}**

**const Comp = connect(...args)(MyComp);**

**connect() 接**收四个参数，它们分别是 **mapStateToProps**，**mapDispatchToProps**，**mergeProps**和**option**

**第一个参数是一个函数 这个函数能够将store中的数据作为props绑定到组件上所以这个函数主要参数就是 Redux 的store还有组件自己的props**

**第二个参数是可以将action作为props绑定到组件上 由于mapDispatchToProps方法返回了increase和decrease属性的对象 这两个属性也会成为组件的props**

**第三个参数是用来做marge的 一般不需要传**

**第四个参数直接忽略**

<https://www.cnblogs.com/passkey/p/9910760.html>

https://www.cnblogs.com/shuitao/p/12071422.html

**redux中间件种类和各自的作用**

**redux-thunk 中间件  改造store.dispatch,解决异步操作**

### 它的引入是为了解决action里面无法进行异步操作的问题。它可以使action返回的是函数。

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;

检查参数 action 的类型，如果是函数的话，就执行这个 action，并把参数传递进去，否则就调用 next 让下一个中间件继续处理 action 。

**redux-promise 中间件**

这个的引入是解决action里面异步写法过于复杂的问题，使得store.dispatch方法可以接受 Promise 对象作为参数

import { isFSA } from 'flux-standard-action';

function isPromise(val) {

return val && typeof val.then === 'function';

}

export default function promiseMiddleware({ dispatch }) {

return next => action => {

if (!isFSA(action)) {

return isPromise(action)

? action.then(dispatch)

: next(action);

}

return isPromise(action.payload)

? action.payload.then(

result => dispatch({ ...action, payload: result }),

error => {

dispatch({ ...action, payload: error, error: true });

return Promise.reject(error);

}

)

: next(action);

};

}

 如果action不是一个promise，则判断其payload是否是一个promise，如果是，等待promise完成，然后将得到的结果作为payload的值触发

 如果action是一个promise，则将其resolve的值dispatch，否则，不做任何处理，交给下一个中间件

**redux-logger中间件 日志中间件**

### 能打印出先前的state 改变后的state和action给开发带来一些便利

### ****redux-saga中间件****

redux-saga相当于在Redux原有数据流中多了一层，通过对Action进行监听，从而捕获到监听的Action，通过更改的state驱动View的变更

### 它本身是基于es6的generator来实现的

Fork是非阻塞的 cell是阻塞的

Fork 创建一个新的进程 发送请求

Cell 发送api请求

Put 发送对应的dispatch 触发action

TakeEvery 监听对应的action

**Takelatest** 防止重复提交 只会触发最后一次dispatch

All 跟fork一样 同时并发多个action 没有顺序

### thunk 和 saga的区别，他们都能实现异步，不同点在于thunk是强耦合的。必须要写在action里面这样的话不便于维护，所以现在更多的是用saga。

### <https://www.jianshu.com/p/b17d8bec13f3>

### https://www.jianshu.com/p/ae7b5a2f78ae

# Redux框架之applyMiddleware

Middleware是无需引用大量代码第三方库实现异步 actions。这种方式可以让你像 dispatch 一般的 actions 那样 dispatch 异步 actions

每个 middleware 接受 Store 的 dispatch 和 getState 函数作为命名参数，并返回一个函数

import logger from "redux-logger";

import thunk from "redux-thunk";

applyMiddleware(logger, thunk)

利用传入的createStore和reducer创建一个store

接着创建一个middlewareAPI 里面有store的getState和dispatch

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

接着用compose将执行完的所有匿名函数，组装成一个新的函数，即新的dispatch return出去

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

}

}

}

https://www.jianshu.com/p/caff049d576c