目录

[基础 2](#_Toc495007848)

[思想 2](#_Toc495007849)

[一些概念 2](#_Toc495007850)

[action creator 2](#_Toc495007851)

[注意事项 2](#_Toc495007852)

[state 2](#_Toc495007853)

[服务器同构 2](#_Toc495007854)

[流程 2](#_Toc495007855)

[action 3](#_Toc495007856)

[reducer 3](#_Toc495007857)

[与flux比较 3](#_Toc495007858)

[dispatch 3](#_Toc495007859)

[mapStateToProps和mapDispatchToProps 3](#_Toc495007860)

[combineReducers 4](#_Toc495007861)

[createStore 5](#_Toc495007862)

[顶层api 6](#_Toc495007863)

[createStore(reducer,[initialState],enhancer) 6](#_Toc495007864)

[combineReducers(reducers) 7](#_Toc495007865)

[applyMiddleware(…middlewares) 8](#_Toc495007866)

[bindActionCreators(actionCreators,dispatch) 8](#_Toc495007867)

[compose(…functions) 10](#_Toc495007868)

[store 11](#_Toc495007869)

[getState 11](#_Toc495007870)

[dispatch(action) 11](#_Toc495007871)

[subscribe(listener) 11](#_Toc495007872)

[replaceReducer(nextReducer) 12](#_Toc495007873)

[react-redux 12](#_Toc495007874)

[provider 13](#_Toc495007875)

[connect 13](#_Toc495007876)

[getWrappedInstance(): ReactComponent 14](#_Toc495007877)

[常用工具 14](#_Toc495007878)

[redux saga 15](#_Toc495007879)

[常见错误 15](#_Toc495007880)

[No reducer provided for key "userReducer" 15](#_Toc495007881)

[Actions must be plain objects 15](#_Toc495007882)

[redux-thunk 16](#_Toc495007883)

[格式 16](#_Toc495007884)

[把dispatch传入return中 16](#_Toc495007885)

[return中的dispatch还可以调用其他的action函数 16](#_Toc495007886)

[可以获得state 16](#_Toc495007887)

[实现思路 17](#_Toc495007888)

# 基础

## 思想

简单状态（数据流）管理。

const sourceOfTruth = {}

const vmA = new Vue({

data: sourceOfTruth

})

const vmB = new Vue({

data: sourceOfTruth

})

这样其实就已经可以多个组件共享一个数据了。但是调试将会变为噩梦。任何时间，我们应用中的任何部分，在任何数据改变后，都不会留下变更过的记录。

所以有了flux的思想：

所有 store 中 state 的改变，都放置在 store 自身的 action 中去管理。这种集中式状态管理能够被更容易地理解哪种类型的 mutation 将会发生，以及它们是如何被触发。当错误出现时，我们现在也会有一个 log 记录 bug 之前发生了什么。

组件不允许直接修改属于 store 实例的 state，而应执行 action 来分发 (dispatch) 事件通知 store 去改变，我们最终达成了 Flux 架构。这样约定的好处是，我们能够记录所有 store 中发生的 state 改变，同时实现能做到记录变更 (mutation)、保存状态快照、历史回滚/时光旅行的先进的调试工具。

# 一些概念

## action creator

就是我们平时写的创建action的函数。

## 注意事项

### state

你可以决定 state 的格式。你可以使用普通对象或者 Immutable 这类的实现。如果你不知道如何做，刚开始可以使用普通对象

如果 state 是普通对象，永远不要修改它！比如，reducer 里不要使用 Object.assign(state, newData)，应该使用 Object.assign({}, state, newData)。这样才不会覆盖旧的 state。也可以使用 Babel 阶段 1 中的 ES7 对象的 spread 操作 特性中的 return { ...state, ...newData }。

### 服务器同构

对于服务端运行的同构应用，为每一个请求创建一个 store 实例，以此让 store 相隔离。dispatch 一系列请求数据的 action 到 store 实例上，等待请求完成后再在服务端渲染应用。

### 流程

当 store 创建后，Redux 会 dispatch 一个 action 到 reducer 上，来用初始的 state 来填充 store。你不需要处理这个 action。但要记住，如果第一个参数也就是传入的 state 如果是 undefined 的话（reducer函数中），reducer 应该返回初始的 state 值。

### action

action 具有 type 字段来表示它的类型。type 也可被定义为常量或者是从其它模块引入。最好使用字符串，而不是 Symbols 作为 action，因为字符串是可以被序列化的。除了 type 字段外，action 对象的结构完全取决于你。

### reducer

* 所有未匹配到的 action，必须把它接收到的第一个参数也就是那个 state 原封不动返回。（就是说要有一个default）
* 永远不能返回 undefined。当过早 return 时非常容易犯这个错误，为了避免错误扩散，遇到这种情况时 combineReducers 会抛异常。
* 如果传入的 state 就是 undefined，一定要返回对应 reducer 的初始 state。根据上一条规则，初始 state 禁止使用 undefined。使用 ES6 的默认参数值语法来设置初始 state 很容易，但你也可以手动检查第一个参数是否为 undefined。

## 与flux比较

### dispatch

redux只有一个store，没有Dispatch，只有一个挂在store下的dispatch函数。

当你在 reducer 内部调用 dispatch 时，将会抛出错误提示“Reducers may not dispatch actions.（Reducer 内不能 dispatch action）”。这就相当于 Flux 里的 “Cannot dispatch in a middle of dispatch（dispatch 过程中不能再 dispatch）”，但并不会引起对应的错误。在 Flux 里，当 Store 处理 action 和触发 update 事件时，dispatch 是禁止的。这个限制并不好，因为他限制了不能在生命周期回调里 dispatch action，还有其它一些本来很正常的地方。

在 Redux 里，只会在根 reducer 返回新 state 结束后再会调用事件监听器，因此，你可以在事件监听器里再做 dispatch。惟一使你不能在 reducer 中途 dispatch 的原因是要确保 reducer 没有副作用。如果 action 处理会产生副作用，正确的做法是使用异步 action 创建函数。

## mapStateToProps和mapDispatchToProps

**将state变成属性，传递到对应的视图组件todolist，todos是属性**

**const mapStateToProps = (state) => {**

**return {**

**todos: getVisibleTodos(state.todos, state.visibilityFilter)**

**}**

**}**

**调用相应的事件函数（action）,函数可以自定义，传入dispatch参数，只要返回的对象符合格式即可。**

**const mapDispatchToProps = (dispatch) => {**

**return {**

**onTodoClick: (id) => {**

**dispatch(toggleTodo(id))**

**}**

**}**

**}**

**const VisibleTodoList = connect(**

**mapStateToProps,**

**mapDispatchToProps**

**)(TodoList)**

**export default VisibleTodoList;**

## combineReducers

使用了以后，store树就会有对应的分支

**import** { *combineReducers* } **from 'redux'**;  
**import** *userReducer* **from './user'**;  
  
**export default** *combineReducers*({  
 **userStore**: *userReducer*,  
});

**user中**

**const** initialState = {  
 **isLoggedIn**: **false**,  
 **user**: {},  
 **status**: **null**,  
};

**export default function** user(state=initialState, action){

**使用时**

**function** *select*(store){  
 **return** {  
 isLoggedIn: store.**userStore**.isLoggedIn,  
 user: store.userStore.user,  
 status: store.userStore.registStatus,  
 }  
}

## createStore

一般会写一个configure函数，在需要的地方创建store

**import { createStore, applyMiddleware } from 'redux';**

**import createSagaMiddleware from 'redux-saga';**

**import logger from 'redux-logger';**

**import reducer from './reducers';**

**import rootSaga from './sagas';**

**export default function configureStore(initialState) {**

**const sagaMiddleware = createSagaMiddleware();**

**const store = createStore(**

**reducer,**

**initialState,**

**applyMiddleware(**

**sagaMiddleware, logger()**

**)**

**);**

**sagaMiddleware.run(rootSaga);**

**return store;**

**};**

**然后**

|  |
| --- |
| **<Provider store={configureStore()}>** |
|  |

|  |
| --- |
| **<App />** |
|  |

**</Provider>,**

# 顶层api



## createStore(reducer,[initialState],enhancer)

**参数**

reducer (Function): 接收两个参数，分别是当前的 **state 树和要处理的 action**，返回新的 state 树。

[initialState] (any): 初始时的 state。 在同构应用中，你可以决定是否把服务端传来的 state 水合（hydrate）后传给它，或者从之前保存的用户会话中恢复一个传给它。如果你使用 combineReducers 创建 reducer，它必须是一个普通对象，与传入的 keys 保持同样的结构。否则，你可以自由传入任何 reducer 可理解的内容。

enhancer (Function): Store enhancer 是一个组合 store creator 的高阶函数，返回一个新的强化过的 store creator。这与 middleware 相似，它也允许你通过复合函数改变 store 接口。



**返回值**

(Store): 保存了应用所有 state 的对象。改变 state 的惟一方法是 dispatch action。你也可以 subscribe 监听 state 的变化，然后更新 UI。

## combineReducers(reducers)

合并后的 reducer 可以调用各个子 reducer，并把它们的结果合并成一个 state 对象。**state 对象的结构由传入的多个 reducer 的 key 决定**。

通常的做法是命名 reducer，然后 state 再去分割那些信息，因此你可以使用 ES6 的简写方法：combineReducers({ counter, todos })。这与 combineReducers({ counter: counter, todos: todos }) 一样。

#### 参数

1. reducers (Object): 一个对象，它的值（value） 对应不同的 reducer 函数，这些 reducer 函数后面会被合并成一个。下面会介绍传入 reducer 函数需要满足的规则。

之前的文档曾建议使用 ES6 的 import \* as reducers 语法来获得 reducer 对象。这一点造成了很多疑问，因此现在建议在 reducers/index.js 里使用 combineReducers() 来对外输出一个 reducer。

在 reducer 层级的任何一级都可以调用 combineReducers。并不是一定要在最外层。实际上，你可以把一些复杂的子 reducer 拆分成单独的孙子级 reducer，甚至更多层。

#### 返回值

(Function)：一个调用 reducers 对象里所有 reducer 的 reducer，并且构造一个与 reducers 对象结构相同的 state 对象。所以最后传入createStore中的其实是这个函数。

## applyMiddleware(…middlewares)

**参数**

...middlewares (arguments): 遵循 Redux middleware API 的函数。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。该函数会被传入 被称为 next 的下一个 middleware 的 **dispatch 方法**，并返回一个接收 action 的新函数，这个函数可以直接调用 next(action)，或者在其他需要的时刻调用，甚至根本不去调用它。调用链中最后一个 middleware 会接受真实的 store 的 dispatch 方法作为 next 参数，并借此结束调用链。所以，middleware 的函数签名是 ({ getState, dispatch }) => next => action。

**返回值**

(Function) 一个应用了 middleware 后的 store enhancer。这个 store enhancer 就是一个函数，并且需要应用到 createStore。它会返回一个应用了 middleware 的新的 createStore。

## bindActionCreators(actionCreators,dispatch)

把 [action creators](http://www.redux.org.cn/docs/Glossary.html#action-creator) 转成拥有同名 keys 的对象，但使用 [dispatch](http://www.redux.org.cn/docs/api/Store.html#dispatch) 把每个 action creator 包围起来，这样可以直接调用它们。

#### 参数

1. actionCreators (Function or Object): 一个 [action creator](http://www.redux.org.cn/docs/Glossary.html#action-creator)，或者键值是 action creators 的对象。
2. dispatch (Function): 一个 [dispatch](http://www.redux.org.cn/docs/api/Store.html#dispatch) 函数，由 [Store](http://www.redux.org.cn/docs/api/Store.html) 实例提供。

#### 返回值

(Function or Object): 一个与原对象类似的对象，只不过这个对象中的的每个函数值都可以直接 dispatch action。如果传入的是一个函数，返回的也是一个函数。

惟一使用 bindActionCreators 的场景是当你需要把 action creator 往下传到一个组件上，却不想让这个组件觉察到 Redux 的存在，而且不希望把 Redux store 或 [dispatch](http://www.redux.org.cn/docs/api/Store.html#dispatch) 传给它。

实例

#### TodoActionCreators.js

export function addTodo(text) {

return {

type: 'ADD\_TODO',

text

};

}

export function removeTodo(id) {

return {

type: 'REMOVE\_TODO',

id

};

}

#### SomeComponent.js

import \* as TodoActionCreators from './TodoActionCreators';

let boundActionCreators = bindActionCreators(TodoActionCreators, dispatch);

console.log(boundActionCreators);

// {

// addTodo: Function,

// removeTodo: Function

// }

如果你使用 ES5，不能使用 import \* as 语法，你可以把 require('./TodoActionCreators') 作为第一个参数传给 bindActionCreators。惟一要考虑的是 actionCreators 的参数全是函数。模块加载系统并不重要。

## compose(…functions)

从右到左来组合多个函数。

这是函数式编程中的方法，为了方便，被放到了 Redux 里。

#### 参数

1. (arguments): 需要合成的多个函数。每个函数都接收一个**函数**作为参数，然后返回一个函数。

#### 返回值

(Function): 从右到左把接收到的函数合成后的最终函数。

compose 做的只是让你不使用深度右括号的情况下来写深度嵌套的函数。不要觉得它很复杂。其实就是先把这些函数变成一个数组，然后遍历一遍形成嵌套。

finalCreateStore = compose(

applyMiddleware(...middleware),

require('redux-devtools').devTools(),

require('redux-devtools').persistState(

window.location.href.match(/[?&]debug\_session=([^&]+)\b/)

),

createStore

);

// 不使用 compose 来写是这样子：

//

// finalCreateStore =

// applyMiddleware(middleware)(

// devTools()(

// persistState(window.location.href.match(/[?&]debug\_session=([^&]+)\b/))(

// createStore

// )

// )

// );

# store

Store 不是类。它只是有几个方法的对象。 要创建它，只需要把根部的 reducing 函数 传递给 createStore。

## getState

返回应用当前的 state 树。  
它与 store 的最后一个 reducer 返回值相同。

## dispatch(action)

这是触发 state 变化的惟一途径。

会使用当前 getState() 的结果和传入的 action 以同步方式的调用 store 的 reduce 函数。返回值会被作为下一个 state。从现在开始，这就成为了 getState() 的返回值，同时变化监听器(change listener)会被触发。

只接受action，不接受函数（可以使用redux实现控制翻转，从而表面看上去实现了dispatch一个函数。其实是在真正判断调用前通过中间件 return dispacht=>{return {type:。。。}} ）

**返回值**

(Object†): 要 dispatch 的 action。（有时候可以有很大用处）

## subscribe(listener)

这是一个底层 API。多数情况下，你不会直接使用它，会使用一些 React（或其它库）的绑定。如果你想让回调函数执行的时候使用当前的 state，你可以 把 store 转换成一个 Observable 或者写一个定制的 observeStore 工具。

#### 参数

listener (Function): 每当 dispatch action 的时候都会执行的回调。state 树中的一部分可能已经变化。你可以在回调函数里调用 [getState()](http://www.redux.org.cn/docs/api/Store.html#getState) 来拿到当前 state。store 的 reducer 应该是纯函数，因此你可能需要对 state 树中的引用做深度比较来确定它的值是否有变化。

##### 返回值

(Function): 一个可以解绑变化监听器的函数。如果需要解绑这个变化监听器，执行 subscribe 返回的函数即可。

function select(state) {

return state.some.deep.property

}

let currentValue

function handleChange() {

let previousValue = currentValue

currentValue = select(store.getState())

if (previousValue !== currentValue) {

console.log('Some deep nested property changed from', previousValue, 'to', currentValue)

}

}

let unsubscribe = store.subscribe(handleChange)

## replaceReducer(nextReducer)

替换 store 当前用来计算 state 的 reducer。

#### 参数

1. reducer (Function) store 会使用的下一个 reducer。

这是一个高级 API。只有在你需要实现代码分隔，而且需要立即加载一些 reducer 的时候才可能会用到它。在实现 Redux 热加载机制的时候也可能会用到。

# react-redux

react-redux就主要使用的就两个组件 Provider和connect



一共四个

export { Provider, createProvider, connectAdvanced, connect }

## provider

如果你**真的**不想把根组件嵌套在 <Provider> 中，你可以把 store 作为 props 传递到每一个被 connet() 包装的组件，但是我们只推荐您在单元测试中对 store 进行伪造 (stub) 或者在非完全基于 React 的代码中才这样做。正常情况下，你应该使用 <Provider>。

**属性**

* store ([*Redux Store*](http://rackt.github.io/redux/docs/api/Store.html)): 应用程序中唯一的 Redux store 对象
* children (ReactElement) 组件层级的根组件。

## connect

本身也是一个组件

返回一个注入了 state 和 action creator 的 React 组件。(对应前面两个函数)

最后面的是返回类型

**[mapStateToProps(state, [ownProps]): stateProps] (Function)**

如果指定了该回调函数中的第二个参数 ownProps，则该参数的值为传递到组件的 props，而且只要组件接收到新的 props，mapStateToProps 也会被调用。

**[mapDispatchToProps(dispatch, [ownProps]): dispatchProps] (Object or Function)**

如果你省略这个 mapDispatchToProps 参数，默认情况下，dispatch 会注入到你的组件 props 中。如果指定了该回调函数中第二个参数 ownProps，该参数的值为传递到组件的 props，而且只要组件接收到新 props，mapDispatchToProps 也会被调用。

如果传递的是一个对象，那么每个定义在该对象的函数都将被当作 Redux action creator，而且这个对象会与 Redux store 绑定在一起，其中所定义的方法名将作为属性名，合并到组件的 props 中。（会自动调用bindActionCreators）

如果传递的是一个函数，该函数将接收一个 dispatch 函数，然后由你来决定如何返回一个对象，这个对象通过 dispatch 函数与 action creator 以某种方式绑定在一起（提示：你也许会用到 Redux 的辅助函数 bindActionCreators()）。

**[mergeProps(stateProps, dispatchProps, ownProps): props] (Function)**

可以用这个回调函数，根据组件的 props 来筛选部分的 state 数据，或者把 props 中的某个特定变量与 action creator 绑定在一起。如果你省略这个参数，默认情况下返回 Object.assign({}, ownProps, stateProps, dispatchProps) 的结果。

**[options] (Object)**

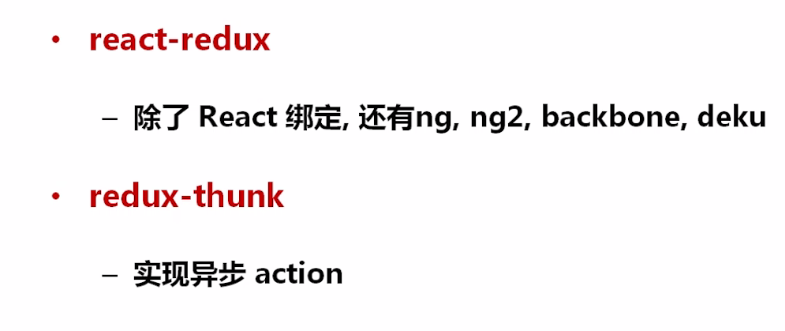
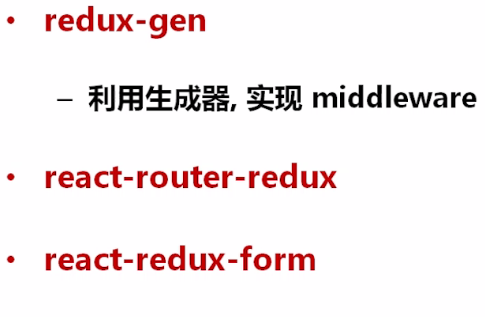
如果指定这个参数，可以定制 connector 的行为。

* [pure = true] *(Boolean)*: 如果为 true，connector 将执行 shouldComponentUpdate 并且浅对比 mergeProps 的结果，避免不必要的更新，前提是当前组件是一个“纯”组件，它不依赖于任何的输入或 state 而只依赖于 props 和 Redux store 的 state。*默认值为 true。*
* [withRef = false] *(Boolean)*: 如果为 true，connector 会保存一个对被包装组件实例的引用，该引用通过 getWrappedInstance() 方法获得。*默认值为 false*

## getWrappedInstance(): ReactComponent

仅当 connect() 函数的第四个参数 options 设置了 { withRef: true } 才返回被包装的组件实例。

# 常用工具

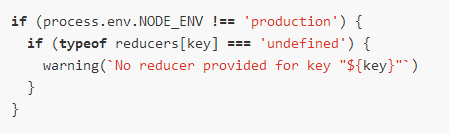


# redux saga

# 常见错误

## No reducer provided for key "userReducer"

在combineReducers.js中



## Actions must be plain objects

在没有使用thunk的情况下,action中,return(action)不能放在回调函数里.

# redux-thunk

## 格式

### 把dispatch传入return中

function makeASandwichWithSecretSauce(forPerson) {

// Invert control!

// Return a function that accepts `dispatch` so we can dispatch later.

// Thunk middleware knows how to turn thunk async actions into actions.

**return function (dispatch)** {

return fetchSecretSauce().then(

sauce => dispatch(makeASandwich(forPerson, sauce)),

error => dispatch(apologize('The Sandwich Shop', forPerson, error))

);

};

}

### return中的dispatch还可以调用其他的action函数

### 可以获得state

function makeSandwichesForEverybody() {

return **function (dispatch, getState)** {

if (**!getState().sandwiches.isShopOpen**) {

// You don’t have to return Promises, but it’s a handy convention

// so the caller can always call .then() on async dispatch result.

**return Promise.resolve();**

}

// We can dispatch both plain object actions and other thunks,

// which lets us compose the asynchronous actions in a single flow.

return dispatch(

makeASandwichWithSecretSauce('My Grandma')

).then(() =>

Promise.all([

dispatch(makeASandwichWithSecretSauce('Me')),

dispatch(makeASandwichWithSecretSauce('My wife'))

])

).then(() =>

dispatch(makeASandwichWithSecretSauce('Our kids'))

).then(() =>

dispatch(getState().myMoney > 42 ?

withdrawMoney(42) :

apologize('Me', 'The Sandwich Shop')

)

);

};

}

## 实现思路

异步以后回调纯action函数.

function makeASandwich(forPerson, secretSauce) {

return {

type: 'MAKE\_SANDWICH',

forPerson,

secretSauce

};

}

function makeASandwichWithSecretSauce(forPerson) {

// Invert control!

// Return a function that accepts `dispatch` so we can dispatch later.

// Thunk middleware knows how to turn thunk async actions into actions.

return function (dispatch) {

return fetchSecretSauce().then(

sauce => **dispatch(makeASandwich**(forPerson, sauce)**)**,

error => dispatch(apologize('The Sandwich Shop', forPerson, error))

);

};

}