**Redux 快速入门 @阮一峰**

[**Redux 基本用法**](https://www.ruanyifeng.com/blog/2016/09/redux_tutorial_part_one_basic_usages.html)

**Redux 设计思想**

1. Web 应用是一个状态机，视图与状态是一一对应的。
2. 所有的状态，保存在一个对象里面。

**Redux 基本概念及 API**

**Store**

1. 解释

* Store 是**用于保存数据/状态的容器**
* 一个应用只能有一个 Store

1. 语法：Store 的生成

|  |
| --- |
| JavaScript import { createStore } from "redux" const store = createStore(reducer) // 参数 reducer 是 Reducer 函数 |

**State**

1. 解释

* State 是**某个时刻 Store 中的数据/状态集合**
* 一个 State 对应一个 View（UI 界面），State 相同则 View 相同

1. 语法：获取当前时刻的 State

|  |
| --- |
| JavaScript const state = store.getState() |

**Action**

1. 解释

* Action 是 View（UI 界面）发出的通知，用于描述当前发生的事情
* Action 是**改变 State 的唯一办法**
* Action 是一个 plain JavaScript object
* Action **必须有 type 属性，可以有 error、payload、meta 属性**

1. 语法：Action 所需属性及其含义

* type（必选）：字符串常量，用于标识 Action 类型。
* Action 的 type 字段是唯一的
* 两个具有相同 type 的 Action 严格等价
* payload：可以是任意类型的值，用于携带与 Action 相关的具体数据。
* 如果 error: true，则 payload 应该是一个错误对象
* error：用于表示 Action 是否代表错误。
* 当 error: true 时，表示 Action 是一个错误，此时 payload 字段应该是一个错误对象，如 new Error("出错信息")
* 当 error 不是 true 时（包括 undefined，null），表示 Action 不应该是一个错误
* meta：可以是任意类型的值，用于存放与 Action 相关的额外信息。

1. 举例

|  |
| --- |
| JavaScript const action = {  type: 'ADD\_TODO',  payload: 'Learn Redux' }; |

**Action Creater**

1. 解释： Action Creater 是一个**创建 Action 的函数**
2. 举例

|  |
| --- |
| JavaScript const actionCreater = (type, payload) => ({ type, payload }) const action = actionCreater("ADD\_TODO", "Learn Redux") |

**Dispatch**

1. 解释

* store.dispatch 是 **View 发出 Action 的唯一方法**
* store.dispatch 接收一个 Action 参数

1. 举例

|  |
| --- |
| JavaScript store.dispatch({  type: 'ADD\_TODO',  payload: 'Learn Redux' }) |

**Subscribe & Unsubscribe**

1. 解释

* store.subscribe 为 State 设置**监听函数，一旦 State 变化，监听函数就会自动执行**
* store.subscribe 接受一个监听函数参数（listener)
* 对 React 而言，**只要将组件的 render 或 setState 方法放入监听函数中，就会实现 View 的自动渲染**
* store.subscribe 返回一个用于解除监听的函数（unsubscribe）

1. 举例

|  |
| --- |
| JavaScript let unsubscribe = store.subscribe(() =>  console.log(store.getState()) );  unsubscribe(); |

**Reducer**

1. 解释

* Reducer 是一个**函数，接收 Action 和当前 State，返回新的 State**
* 可以将 Reducer 理解为 State 的一个计算过程：Store 收到 View 发出的 Action → 计算出一个新的 State → View 更新
* store.dispatch 方法会自动触发 Reducer 函数的自动执行，也就是使用 createStore 创建 Store 时传入的函数参数
* Reducer 函数可以作为数组 reduce 方法的参数，用于计算多个 Action 影响后的 State。

1. 语法

|  |
| --- |
| JavaScript const reducer = function (state, action) {  // ...  return new\_state;  }; |

1. 举例：Reducer 函数的实现

|  |
| --- |
| JavaScript const defaultState = 0; // 初始状态  const reducer = (state = defaultState, action) => {  switch (action.type) {  case 'ADD':  return state + action.payload;  default:  return state;  }  };   const state = reducer(1, {  type: 'ADD',  payload: 2  }); |

1. 举例：Reducer 函数作为 array.reduce 的参数，计算多个 Action 影响后的 State

|  |
| --- |
| JavaScript const actions = [ { type: 'ADD', payload: 0 }, { type: 'ADD', payload: 1 }, { type: 'ADD', payload: 2 } ];  const total = actions.reduce(reducer, 0); // reducer 中的 state 对应 reduce 方法的 accumulator，action 对应 currentValue |

1. 补充：array.reduce 的使用

* 语法：array.reduce(callback(accumulator, currentValue, currentIndex, array), initialValue)
* 参数
* callback: 一个在数组每一项上执行的函数。这个函数接收四个参数：
* accumulator: 累计器，累计回调的返回值。初始值是 initialValue，或者是数组中的第一个元素（如果没有提供 initialValue）。
* currentValue: 数组中正在处理的当前元素。
* currentIndex（可选）: 数组中正在处理的当前元素的索引。
* array（可选）: 调用 reduce 的数组。
* initialValue（可选）: 作为第一次调用回调函数时 accumulator 的初始值。如果没有提供初始值，accumulator 将使用数组中的第一个元素，并且从第二个元素开始迭代。
* 返回值：reduce 方法返回函数累计处理的结果。

**纯函数**

1. 解释

* 如果一个函数满足 **same input same output**，该函数就是一个纯函数
* 纯函数需要遵守的约束为：①不得改写参数 ②不能调用系统 I/O 的API ③不能调用 Date.now() 或者 Math.random() 等不纯的方法，因为每次会得到不一样的结果
* Reducer 是一个纯函数，因此**在函数中不能改变当前 State，只能返回一个全新的 State**。
* 基于纯函数的约束，可以认为 State 总是一个不变的对象。

1. 语法

|  |
| --- |
| JavaScript // State 是一个对象 function reducer(state, action) {  return Object.assign({}, state, { thingToChange });  // 或者  return { ...state, ...newState }; }  // State 是一个数组 function reducer(state, action) {  return [...state, newItem]; } |

**createStore 的简单实现**

|  |
| --- |
| JavaScript *const* createStore = (*reducer*) *=>* {  /\* 初始化状态和监听器 \*/  *let* state;  *let* listeners = [];   /\* ⭐ 获取当前状态 \*/  *const* getState = () *=>* state;   /\* ⭐ 分发动作 \*/  *const* dispatch = (*action*) *=>* {  state = reducer(state, *action*); // 更新状态  listeners.forEach(*listener* *=>* listener()); // 执行监听函数  };   /\* ⭐ 订阅状态变化 \*/  *const* subscribe = (*listener*) *=>* {  listeners.push(*listener*); // 订阅  return () *=>* { // 取消订阅函数  listeners = listeners.filter(*l* *=>* *l* !== *listener*);  }  };   /\* 初始化状态：令 state = initialState \*/  dispatch({});   return { getState, dispatch, subscribe }; };  /\* 以下是对应的 Reducer 函数 \*/ *const* reducer = (*state* = initialState, *action*) *=>* {  switch (*action*.type) {  // 处理不同的 action 类型  // ...  default:  return *state*;  } }; |

**Reducer 的拆分**

**Reducer 拆分的原因、方法及优点**

1. 原因

* Reducer 函数负责生成整个应用的 State 对象。随着应用规模的扩大，State 可能会变得复杂和庞大，导致单个 Reducer 函数过于复杂和臃肿。
* 拆分 Reducer 函数可以将不同部分的状态管理逻辑分解为多个独立的子 Reducer 函数，使代码结构更清晰、易于维护和扩展。

1. 方法

* 在复杂的应用中，State 对象往往包含多个字段和子对象。为了简化管理和维护，可以按照应用中不同的数据部分，将处理不同字段的逻辑封装为独立的子 Reducer 函数。
* 每个子 Reducer 负责管理和更新 State 的一个特定部分，例如一个数组、一个对象或者一个简单的值。然后使用 Redux 提供的 combineReducers 函数，将这些子 Reducer 合并为一个根 Reducer，这个根 Reducer 就可以管理整个应用的 State。
* 总而言之，**将 Reducer 函数拆分，不同子 Reducer 负责不同属性，最终将子函数合并为一个大 Reducer**。

1. 优点：可以让子 Reducer 与子组件对应，使得 Reducer 结构与 React 结构相吻合

**Reducer 拆分举例**

**Before**

|  |
| --- |
| JavaScript /\* 未拆分 \*/ *const* chatReducer1 = (*state* = defaultState, *action* = {}) *=>* {  *const* { type, payload } = *action*;  switch (type) {  case ADD\_CHAT: // 追加 chatLog  return *Object*.assign({}, *state*, {  chatLog: *state*.chatLog.concat(payload)  });  case CHANGE\_STATUS: // 更新 statusMessage  return *Object*.assign({}, *state*, {  statusMessage: payload  });  case CHANGE\_USERNAME: // 更新 userName  return *Object*.assign({}, *state*, {  userName: payload  });  default: return *state*;  } }; |

**After**

|  |
| --- |
| JavaScript /\* 拆分后 \*/ /\* 注意事项  1. 以下写法要求 state 字段名必须与子 Reducer 同名。如果不同名，  则 key 表示 state 字段名，value 表示子 Reducer 名  2. combineReducer 将子 Reducer 合并一个整体的 Reducer \*/ import { combineReducers } from 'redux';  /\* 整体 Reducer \*/ *const* chatReducer2 = combineReducers({  chatLog,  statusMessage,  userName }) /\* 以上等价于 \*/ *const* chatReducer3 = (*state* = {}, *action*) *=>* {  return {  chatLog: chatLog(*state*.chatLog, *action*),  statusMessage: statusMessage(*state*.statusMessage, *action*),  userName: userName(*state*.userName, *action*)  } }  /\* 子 Reducer \*/ // chatLog reducer, 这里的 state 实际上对应整体 state 的 chatLog 字段 *function* chatLog(*state* = [], *action*) {  switch (*action*.type) {  case ADD\_CHAT:  return *state*.concat(*action*.payload);  default:  return *state*;  } }  // statusMessage reducer, 这里的 state 实际上对应整体 state 的 statusMessage 字段 *function* statusMessage(*state* = '', *action*) {  switch (*action*.type) {  case CHANGE\_STATUS:  return *action*.payload;  default:  return *state*;  } }  // userName reducer, 这里的 state 实际上对应整体 state 的 userName 字段 *function* userName(*state* = '', *action*) {  switch (*action*.type) {  case CHANGE\_USERNAME:  return *action*.payload;  default:  return *state*;  } } |

**Reducer 拆分后的模块化组织**

|  |
| --- |
| JavaScript /\* Reducer 的模块化逻辑 \*/ import { combineReducers } from 'redux' import \* as reducers from './reducers'  const reducer = combineReducers(reducers) |

**combineReducers 的简单实现**

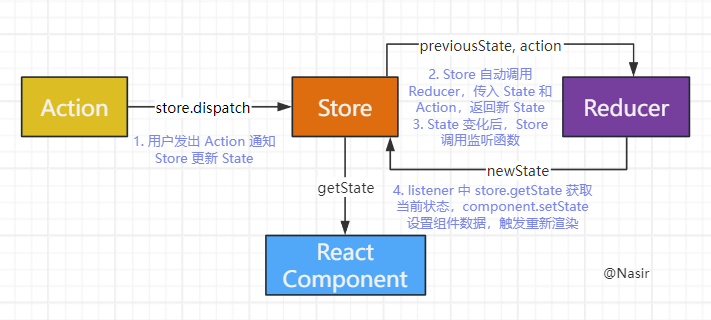
|  |
| --- |
| JavaScript /\* combineReducers 的简单实现 \*/ *const* combineReducers = *subReducers* *=>* {  return (*state* = {}, *action*) *=>* {  return *Object*.keys(*subReducers*).reduce(  (*nextState*, *key*) *=>* {  *nextState*[*key*] = *subReducers*[*key*](*state*[*key*], *action*);  return *nextState*;  },  {}  );  }; }; |

**Redux 实例：计数器**

|  |
| --- |
| JavaScript *const* Counter = ({ *value*, *onIncrement*, *onDecrement* }) *=>* (  <div>  <h1>{*value*}</h1>  <button onClick={*onIncrement*}>+</button>  <button onClick={*onDecrement*}>-</button>  </div> );  *const* reducer = (*state* = 0, *action*) *=>* {  switch (*action*.type) {  case 'INCREMENT': return *state* + 1;  case 'DECREMENT': return *state* - 1;  default: return *state*;  } };  *const* store = createStore(reducer);  *const* render = () *=>* {  ReactDOM.render(  <*Counter*  value={store.getState()}  onIncrement={() *=>* store.dispatch({ type: 'INCREMENT' })}  onDecrement={() *=>* store.dispatch({ type: 'DECREMENT' })}  />,  document.getElementById('root')  ); };  render(); store.subscribe(render); |

**Redux 工作流**

View 发出 Action，Reducer 函数算出新的 State，View 重新渲染



**Redux 中间件与异步操作**

[**React-Redux 的用法**](https://www.ruanyifeng.com/blog/2016/09/redux_tutorial_part_three_react-redux.html)