

Redux 入门教程（一）：基本用法

作者： 阮一峰

日期： 2016年9月18日

一年半前，我写了[《React 入门实例教程》](#)，介绍了 React 的基本用法。

React 只是 DOM 的一个抽象层，并不是 Web 应用的完整解决方案。有两个方面，它没涉及。

- 代码结构
- 组件之间的通信

对于大型的复杂应用来说，这两方面恰恰是最关键的。因此，只用 React 没法写大型应用。

为了解决这个问题，2014年 Facebook 提出了 [Flux](#) 架构的概念，引发了很多的实现。2015年，[Redux](#) 出现，将 Flux 与函数式编程结合在一起，很短时间就成为了最热门的前端架构。

本文详细介绍 Redux 架构，由于内容较多，全文分成三个部分。今天是第一部分，介绍基本概念和用法。



零、你可能不需要 Redux

首先明确一点，Redux 是一个有用的架构，但不是非用不可。事实上，大多数情况，你可以不用它，只用 React 就够了。

曾经有人说过这样一句话。

"如果你不知道是否需要 Redux，那就是不需要它。"

Redux 的创造者 Dan Abramov 又补充了一句。

"只有遇到 **React** 实在解决不了的问题，你才需要 **Redux** 。

简单说，如果你的UI层非常简单，没有很多互动，**Redux** 就是不必要的，用了反而增加复杂性。

- 用户的使用方式非常简单
- 用户之间没有协作
- 不需要与服务器大量交互，也没有使用 **WebSocket**
- 视图层（**View**）只从单一来源获取数据

上面这些情况，都不需要使用 **Redux**。

- 用户的使用方式复杂
- 不同身份的用户有不同的使用方式（比如普通用户和管理员）
- 多个用户之间可以协作
- 与服务器大量交互，或者使用了**WebSocket**
- **View**要从多个来源获取数据

上面这些情况才是 **Redux** 的适用场景：多交互、多数据源。

从组件角度看，如果你的应用有以下场景，可以考虑使用 **Redux**。

- 某个组件的状态，需要共享
- 某个状态需要在任何地方都可以拿到
- 一个组件需要改变全局状态
- 一个组件需要改变另一个组件的状态

发生上面情况时，如果不使用 **Redux** 或者其他状态管理工具，不按照一定规律处理状态的读写，代码很快就会变成一团乱麻。你需要一种机制，可以在同一个地方查询状态、改变状态、传播状态的变化。

总之，不要把 **Redux** 当作万灵丹，如果你的应用没那么复杂，就没必要用它。另一方面，**Redux** 只是 **Web** 架构的一种解决方案，也可以选择其他方案。

一、预备知识

阅读本文，你只需要懂 **React**。如果还懂 **Flux**，就更好了，会比较容易理解一些概念，但不是必需的。

Redux 有很好的[文档](#)，还有配套的小视频（[前30集](#)，[后30集](#)）。你可以先阅读本文，再去官方材料详细研究。

我的目标是，提供一个简洁易懂的、全面的入门级参考文档。

二、设计思想

Redux 的设计思想很简单，就两句话。

- （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。

请务必记住这两句话，下面就是详细解释。

三、基本概念和 API

3.1 Store

Store 就是保存数据的地方，你可以把它看成一个容器。整个应用只能有一个 **Store**。

Redux 提供 `createStore` 这个函数，用来生成 **Store**。

```
import { createStore } from 'redux';  
const store = createStore(fn);
```

上面代码中，`createStore` 函数接受另一个函数作为参数，返回新生成的 **Store** 对象。

3.2 State

Store 对象包含所有数据。如果想得到某个时点的数据，就要对 **Store** 生成快照。这种时点的数据集合，就叫做 **State**。

当前时刻的 **State**，可以通过 `store.getState()` 拿到。

```
import { createStore } from 'redux';
const store = createStore(fn);

const state = store.getState();
```

Redux 规定，一个 **State** 对应一个 **View**。只要 **State** 相同，**View** 就相同。你知道 **State**，就知道 **View** 是什么样，反之亦然。

3.3 Action

State 的变化，会导致 **View** 的变化。但是，用户接触不到 **State**，只能接触到 **View**。所以，**State** 的变化必须是 **View** 导致的。**Action** 就是 **View** 发出的通知，表示 **State** 应该要发生变化了。

Action 是一个对象。其中的 `type` 属性是必须的，表示 **Action** 的名称。其他属性可以自由设置，社区有一个[规范](#)可以参考。

```
const action = {
  type: 'ADD_TODO',
  payload: 'Learn Redux'
};
```

上面代码中，**Action** 的名称是 `ADD_TODO`，它携带的信息是字符串 `Learn Redux`。

可以这样理解，**Action** 描述当前发生的事情。改变 **State** 的唯一办法，就是使用 **Action**。它会运送数据到 **Store**。

3.4 Action Creator

View 要发送多少种消息，就会有多种 **Action**。如果都手写，会很麻烦。可以定义一个函数来生成 **Action**，这个函数就叫 **Action Creator**。

```
const ADD_TODO = '添加 TODO';

function addTodo(text) {
  return {
    type: ADD_TODO,
    text
  }
}
```

```
}  
  
const action = addTodo('Learn Redux');
```

上面代码中，`addTodo` 函数就是一个 **Action Creator**。

3.5 store.dispatch()

`store.dispatch()` 是 **View** 发出 **Action** 的唯一方法。

```
import { createStore } from 'redux';  
const store = createStore(fn);  
  
store.dispatch({  
  type: 'ADD_TODO',  
  payload: 'Learn Redux'  
});
```

上面代码中，`store.dispatch` 接受一个 **Action** 对象作为参数，将它发送出去。

结合 **Action Creator**，这段代码可以改写如下。

```
store.dispatch(addTodo('Learn Redux'));
```

3.6 Reducer

Store 收到 **Action** 以后，必须给出一个新的 **State**，这样 **View** 才会发生变化。这种 **State** 的计算过程就叫做 **Reducer**。

Reducer 是一个函数，它接受 **Action** 和当前 **State** 作为参数，返回一个新的 **State**。

```
const reducer = function (state, action) {  
  // ...  
  return new_state;  
};
```

整个应用的初始状态，可以作为 **State** 的默认值。下面是一个实际的例子。

```
const defaultState = 0;
```

```
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
});
```

上面代码中，`reducer` 函数收到名为 `ADD` 的 `Action` 以后，就返回一个新的 `State`，作为加法的计算结果。其他运算的逻辑（比如减法），也可以根据 `Action` 的不同来实现。

实际应用中，`Reducer` 函数不用像上面这样手动调用，`store.dispatch` 方法会触发 `Reducer` 的自动执行。为此，`Store` 需要知道 `Reducer` 函数，做法就是在生成 `Store` 的时候，将 `Reducer` 传入 `createStore` 方法。

```
import { createStore } from 'redux';
const store = createStore(reducer);
```

上面代码中，`createStore` 接受 `Reducer` 作为参数，生成一个新的 `Store`。以后每当 `store.dispatch` 发送过来一个新的 `Action`，就会自动调用 `Reducer`，得到新的 `State`。

为什么这个函数叫做 `Reducer` 呢？因为它可以作为数组的 `reduce` 方法的参数。请看下面的例子，一系列 `Action` 对象按照顺序作为一个数组。

```
const actions = [
  { type: 'ADD', payload: 0 },
  { type: 'ADD', payload: 1 },
  { type: 'ADD', payload: 2 }
];

const total = actions.reduce(reducer, 0); // 3
```

上面代码中，数组 `actions` 表示依次有三个 `Action`，分别是加 0、加 1 和加 2。数组的 `reduce` 方法接受 `Reducer` 函数作为参数，就可以直接得到最终的状态 3。

3.7 纯函数

Reducer 函数最重要的特征是，它是一个纯函数。也就是说，只要是同样的输入，必定得到同样的输出。

纯函数是函数式编程的概念，必须遵守以下一些约束。

- 不得改写参数
- 不能调用系统 I/O 的API
- 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

由于 **Reducer** 是纯函数，就可以保证同样的**State**，必定得到同样的 **View**。但也正因为这一点，**Reducer** 函数里面不能改变 **State**，必须返回一个全新的对象，请参考下面的写法。

```
// 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];
}
```

最好把 **State** 对象设成只读。你没法改变它，要得到新的 **State**，唯一办法就是生成一个新对象。这样的好处是，任何时候，与某个 **View** 对应的 **State** 总是一个不变的对象。

3.8 store.subscribe()

Store 允许使用 `store.subscribe` 方法设置监听函数，一旦 **State** 发生变化，就自动执行这个函数。

```
import { createStore } from 'redux';
const store = createStore(reducer);

store.subscribe(listener);
```

显然，只要把 **View** 的更新函数（对于 **React** 项目，就是组件的 `render` 方法或 `setState` 方法）放入 `listen`，就会实现 **View** 的自动渲染。

`store.subscribe` 方法返回一个函数，调用这个函数就可以解除监听。

```
let unsubscribe = store.subscribe(() =>
  console.log(store.getState())
);

unsubscribe();
```

四、Store 的实现

上一节介绍了 Redux 涉及的基本概念，可以发现 Store 提供了三个方法。

- store.getState()
- store.dispatch()
- store.subscribe()

```
import { createStore } from 'redux';
let { subscribe, dispatch, getState } = createStore(reducer);
```

createStore 方法还可以接受第二个参数，表示 State 的最初状态。这通常是服务器给出的。

```
let store = createStore(todoApp, window.STATE_FROM_SERVER)
```

上面代码中，window.STATE_FROM_SERVER 就是整个应用的状态初始值。注意，如果提供了这个参数，它会覆盖 Reducer 函数的默认初始值。

下面是 createStore 方法的一个简单实现，可以了解一下 Store 是怎么生成的。

```
const createStore = (reducer) => {
  let state;
  let listeners = [];

  const getState = () => state;

  const dispatch = (action) => {
    state = reducer(state, action);
    listeners.forEach(listener => listener());
  };
};
```



```
const subscribe = (listener) => {
  listeners.push(listener);
  return () => {
    listeners = listeners.filter(l => l !== listener);
  }
};

dispatch({});

return { getState, dispatch, subscribe };
};
```

五、Reducer 的拆分

Reducer 函数负责生成 State。由于整个应用只有一个 State 对象，包含所有数据，对于大型应用来说，这个 State 必然十分庞大，导致 Reducer 函数也十分庞大。

请看下面的例子。

```
const chatReducer = (state = defaultState, action = {}) => {
  const { type, payload } = action;
  switch (type) {
    case ADD_CHAT:
      return Object.assign({}, state, {
        chatLog: state.chatLog.concat(payload)
      });
    case CHANGE_STATUS:
      return Object.assign({}, state, {
        statusMessage: payload
      });
    case CHANGE_USERNAME:
      return Object.assign({}, state, {
        userName: payload
      });
    default: return state;
  }
};
```

上面代码中，三种 Action 分别改变 State 的三个属性。

- ADD_CHAT: chatLog 属性
- CHANGE_STATUS: statusMessage 属性

- **CHANGE_USERNAME:** `userName`属性

这三个属性之间没有联系，这提示我们可以把 **Reducer** 函数拆分。不同的函数负责处理不同属性，最终把它们合并成一个大的 **Reducer** 即可。

```
const chatReducer = (state = defaultState, action = {}) => {  
  return {  
    chatLog: chatLog(state.chatLog, action),  
    statusMessage: statusMessage(state.statusMessage, action),  
    userName: userName(state.userName, action)  
  }  
};
```

上面代码中，**Reducer** 函数被拆成了三个小函数，每一个负责生成对应的属性。

这样一拆，**Reducer** 就易读易写多了。而且，这种拆分与 **React** 应用的结构相吻合：一个 **React** 根组件由很多子组件构成。这就是说，子组件与子 **Reducer** 完全可以对应。

Redux 提供了一个 `combineReducers` 方法，用于 **Reducer** 的拆分。你只要定义各个子 **Reducer** 函数，然后用这个方法，将它们合成一个大的 **Reducer**。

```
import { combineReducers } from 'redux';  
  
const chatReducer = combineReducers({  
  chatLog,  
  statusMessage,  
  userName  
})  
  
export default todoApp;
```

上面的代码通过 `combineReducers` 方法将三个子 **Reducer** 合并成一个大的函数。

这种写法有一个前提，就是 **State** 的属性名必须与子 **Reducer** 同名。如果不同名，就要采用下面的写法。

```
const reducer = combineReducers({  
  a: doSomethingWithA,  
  b: processB,  
  c: c  
})
```

```
// 等同于
function reducer(state = {}, action) {
  return {
    a: doSomethingWithA(state.a, action),
    b: processB(state.b, action),
    c: c(state.c, action)
  }
}
```

总之，`combineReducers()` 做的就是产生一个整体的 **Reducer** 函数。该函数根据 **State** 的 **key** 去执行相应的子 **Reducer**，并将返回结果合并成一个大的 **State** 对象。

下面是 `combineReducer` 的简单实现。

```
const combineReducers = reducers => {
  return (state = {}, action) => {
    return Object.keys(reducers).reduce(
      (nextState, key) => {
        nextState[key] = reducers[key](state[key], action);
        return nextState;
      },
      {}
    );
  };
};
```

你可以把所有子 **Reducer** 放在一个文件里面，然后统一引入。

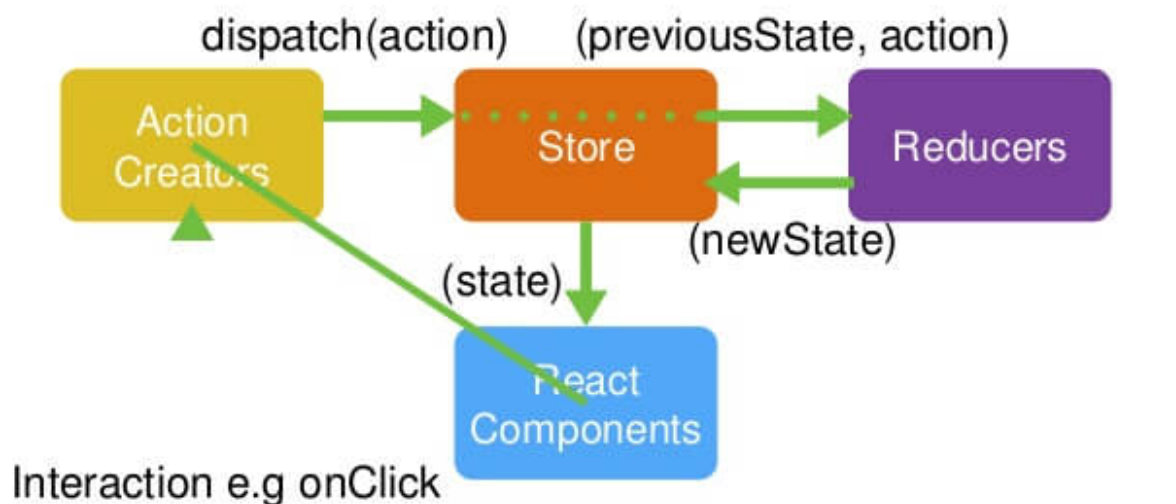
```
import { combineReducers } from 'redux'
import * as reducers from './reducers'

const reducer = combineReducers(reducers)
```

六、工作流程

本节对 **Redux** 的工作流程，做一个梳理。

Redux Flow



React + Redux

@nikgraf

首先，用户发出 Action。

```
store.dispatch(action);
```

然后，Store 自动调用 Reducer，并且传入两个参数：当前 State 和收到的 Action。Reducer 会返回新的 State。

```
let nextState = todoApp(previousState, action);
```

State 一旦有变化，Store 就会调用监听函数。

```
// 设置监听函数  
store.subscribe(listener);
```

listener 可以通过 `store.getState()` 得到当前状态。如果使用的是 React，这时可以触发重新渲染 View。

```
function listener() {  
  let newState = store.getState();
```

```
component.setState(newState);  
}
```

七、实例：计数器

下面我们来看一个最简单的实例。

```
const Counter = ({ value }) => (  
  <h1>{value}</h1>  
);  
  
const render = () => {  
  ReactDOM.render(  
    <Counter value={store.getState()}</>,  
    document.getElementById('root')  
  );  
};  
  
store.subscribe(render);  
render();
```

上面是一个简单的计数器，唯一的作用就是把参数 `value` 的值，显示在网页上。`Store` 的监听函数设置为 `render`，每次 `State` 的变化都会导致网页重新渲染。

下面加入一点变化，为 `Counter` 添加递增和递减的 `Action`。

```
const Counter = ({ value }) => (  
  <h1>{value}</h1>  
  <button onClick={onIncrement}>+</button>  
  <button onClick={onDecrement}>-</button>  
);  
  
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);
```

完整的代码请看[这里](#)。

Redux 的基本用法就介绍到这里，[下一次](#)介绍它的高级用法：中间件和异步操作。

（完）

文档信息

- 版权声明： 自由转载-非商用-非衍生-保持署名（[创意共享3.0许可证](#)）
- 发表日期： 2016年9月18日
- 更多内容： [档案](#) » JavaScript
- 博客文集： 《前方的路》， 《未来世界的幸存者》
- 社交媒体：  twitter,  weibo
- Feed订阅： 

打造中国最权威的《前端-全栈-工程化课程》

八年专注前端，珠峰培训让你高薪就业

快戳我！了解详情 

年薪50万不是梦

从前端小工到BAT中高级工程师的必备技能

 13大模块 / 52 个课时 / 3个月强化学习

相关文章

- 2017.04.16: [JavaScript 内存泄漏教程](#)

一、什么是内存泄漏？程序的运行需要内存。只要程序提出要求，操作系统或者运行时（runtime）就必须供给内存。

- 2017.03.18: [Reduce 和 Transduce 的含义](#)

学习函数式编程，必须掌握很多术语，否则根本看不懂文档。

- 2017.03.13: [Pointfree 编程风格指南](#)

本文要回答一个很重要的问题：函数式编程有什么用？

- 2017.03.09: [Ramda 函数库参考教程](#)

学习函数式编程的过程中，我接触到了 Ramda.js。

联系方式 | 2003 - 2017

