刘小夕 Lv5
2019年09月23日 阅读 2237

关注

【React系列】从零开始实现Redux

本篇文章会从零开始编写 Redux ，如果你对 Redux 的使用和源码已经非常熟悉，那么你只需要粗略浏览即可，如果你还不太清楚，那么跟着本篇文章一起来实现你的 Redux 吧。知其然，知其所以然。

本篇文章对应的代码在: github.com/YvetteLau/B... ，建议先 clone 代码，对照代码阅读本文。

1. Redux 是什么?

Redux 是 JavaScript 状态容器，提供可预测化的状态管理。 Redux 除了和 React 一起用外，还支持其它界面库。 Redux 体小精悍，仅有 2KB 。这里我们需要明确一点： Redux 和 React 之间，没有强绑定的关系。本文旨在理解和实现一个 Redux ，但是不会涉及 react-redux (一次深入理解一个知识点即可， react-redux 将出现在下一篇文章中)。

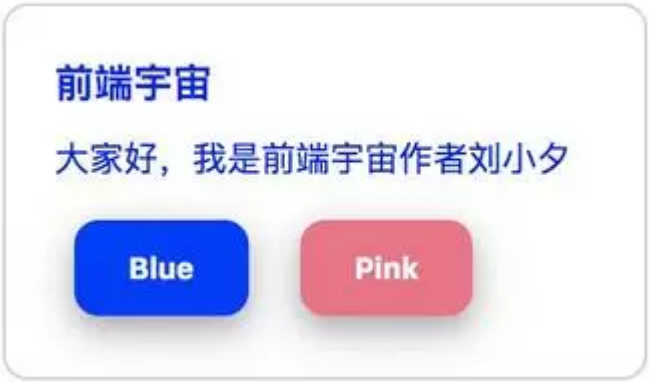
2. 从零开始实现一个 Redux

我们先忘记 Redux 的概念，从一个例子入手，使用 create-react-app 创建一个项目: to-redux 。

代码地址: myredux/to-redux 中。

将 public/index.html 中 body 修改为如下:

```html
<div id="app">
    <div id="header">
        前端宇宙
    </div>
    <div id="main">
        <div id="content">大家好，我是前端宇宙作者刘小夕</div>
        <button class="change-theme" id="to-blue">Blue</button>
        <button class="change-theme" id="to-pink">Pink</button>
    </div>
</div>
```



我们要实现的功能如上图所示，在点击按钮时，能够修改整个应用的字体的颜色。

修改 `src/index.js` 如下(代码: `to-redux/src/index1.js`):

```javascript
let state = {
    color: 'blue'
}
//渲染应用
function renderApp() {
    renderHeader();
    renderContent();
}
//渲染 title 部分
function renderHeader() {
    const header = document.getElementById('header');
    header.style.color = state.color;
}
//渲染内容部分
function renderContent() {
    const content = document.getElementById('content');
    content.style.color = state.color;
}

renderApp();

//点击按钮，更改字体颜色
document.getElementById('to-blue').onclick = function () {
    state.color = 'rgb(0, 51, 254)';
    renderApp();
}
document.getElementById('to-pink').onclick = function () {
    state.color = 'rgb(247, 109, 132)';
    renderApp();
}
```

这个应用非常简单，但是它有一个问题： `state` 是共享状态，但是任何人都可以修改它，一旦我们随意修改了这个状态，就可以导致出错，例如，在 `renderHeader` 里面，设置 `state = {}` ， 容易造成难以预料的错误。

不过很多时候，我们又的确需要共享状态，因此我们可以考虑设置一些门槛，比如，我们约定，不能直接修改全局状态，必须要通过某个途经才能修改。为此我们定义一个 `changeState` 函数，全局状态的修改均由它负责。

javascript

```
//在 index.js 中继续追加代码
function changeState(action) {
    switch(action.type) {
        case 'CHANGE_COLOR':
            return {
                ...state,
                color: action.color
            }
        default:
            return state;
    }
}
```

我们约定只能通过 `changeState` 去修改状态，它接受一个参数 `action` ,包含 `type` 字段的普通对象， `type` 字段用于识别你的操作类型(即如何修改状态)。

我们希望点击按钮，可以修改整个应用的字体颜色。

javascript

```
//在 index.js 中继续追加代码
document.getElementById('to-blue').onclick = function() {
    let state = changeState({
        type: 'CHANGE_COLOR',
        color: 'rgb(0, 51, 254)'
    });
    //状态修改完之后，需要重新渲染页面
    renderApp(state);
}

document.getElementById('to-pink').onclick = function() {
    let state = changeState({
        type: 'CHANGE_COLOR',
        color: 'rgb(247, 109, 132)'
    });
    renderApp(state);
}
```

抽离 store

尽管现在我们约定了如何修改状态，但是 `state` 是一个全局变量，我们很容易就可以修改它，因此我们可以考虑将其变成局部变量，将其定义在一个函数内部(`createStore`)，但是在外部还需要使用 `state` ，因此我们需要提供一个方法 `getState()` ，以便我们在 `createStore` 外部获取到 `state` 。

```javascript
function createStore (state) {
    const getState = () => state;
    return {
        getState
    }
}
```

现在，我们可以通过 `store.getState()` 方法去获取状态(这里需要说明的是， `state` 通常是一个对象，因此这个对象在外部其实是可以被直接修改的，但是如果深拷贝 `state` 返回，那么在外部就一定修改不了，鉴于 `redux` 源码中就是直接返回了 `state` ，此处我们也不进行深拷贝(毕竟耗费性能)。

仅仅获取状态是远远不够的，我们还需要有修改状态的方法，现在状态是私有变量，我们必须要将修改状态的方法也放到 `createStore` 中，并将其暴露给外部使用。

```javascript
function createStore (state) {
    const getState = () => state;
    const changeState = () => {
        //...changeState 中的 code
    }
    return {
        getState,
        changeState
    }
}
```

现在， `index.js` 中代码变成下面这样(`to-redux/src/index2.js`):

```javascript
function createStore() {
    let state = {
        color: 'blue'
    }
    const getState = () => state;
    function changeState(action) {
        switch (action.type) {
            case 'CHANGE_COLOR':
                state = {
```

```
                    ...state,
                    color: action.color
                }
                return state;
            default:
                return state;
        }
    }
    return {
        getState,
        changeState
    }
}

function renderApp(state) {
    renderHeader(state);
    renderContent(state);
}
function renderHeader(state) {
    const header = document.getElementById('header');
    header.style.color = state.color;
}
function renderContent(state) {
    const content = document.getElementById('content');
    content.style.color = state.color;
}

document.getElementById('to-blue').onclick = function () {
    store.changeState({
        type: 'CHANGE_COLOR',
        color: 'rgb(0, 51, 254)'
    });
    renderApp(store.getState());
}
document.getElementById('to-pink').onclick = function () {
    store.changeState({
        type: 'CHANGE_COLOR',
        color: 'rgb(247, 109, 132)'
    });
    renderApp(store.getState());
}
const store = createStore();
renderApp(store.getState());
```

尽管，我们现在抽离了 `createStore` 方法，但是显然这个方法一点都不通用， `state` 和 `changeState` 方法都定义在了 `createStore` 中。这种情况下，其它应用无法复用此模式。

`changeState` 的逻辑理应在外部定义，因为每个应用修改状态的逻辑定然是不同的。我们将这部分逻辑剥离到外部，并将其重命名为 `reducer` (憋问为什么叫 `reducer`，问就是为了和 `redux` 保持一致)。 `reducer` 是干嘛的呢，说白了就是根据 `action` 的类型，计算出新状态。因为它不是在 `createStore` 内部定义的，无法直接访问 `state`，因此我们需要将当前状态作为参数传递给它。如下:

```javascript
function reducer(state, action) {
    switch(action.type) {
        case 'CHANGE_COLOR':
            return {
                ...state,
                color: action.color
            }
        default:
            return state;
    }
}
```

createStore 进化版

```javascript
function createStore(reducer) {
    let state = {
        color: 'blue'
    }
    const getState = () => state;
    //将此处的 changeState 更名为 `dispatch`
    const dispatch = (action) => {
        //reducer 接收老状态和action，返回一个新状态
        state = reducer(state, action);
    }
    return {
        getState,
        dispatch
    }
}
```

不同应用的 `state` 定然是不同的，我们将 `state` 的值定义在 `createStore` 内部必然是不合理的。

```javascript
function createStore(reducer) {
    let state;
    const getState = () => state;
    const dispatch = (action) => {
        //reducer(state, action) 返回一个新状态
```

```
            state = reducer(state, action);
        }
        return {
            getState,
            dispatch
        }
    }
```

大家注意 `reducer` 的定义，在碰到不能识别的动作时，是直接返回旧状态的，现在，我们利用这一点来返回初始状态。

要想 `state` 有初始状态，其实很简单，咱们将初始的 `state` 的初始化值作为 `reducer` 的参数的默认值，然后在 `createStore` 中派发一个 `reducer` 看不懂的动作就可以了。这样 `getState` 首次调用时，可以获取到状态的默认值。

createStore 进化版2.0

javascript

```
function createStore(reducer) {
    let state;
    const getState = () => state;
    //每当 `dispatch` 一个动作的时候，我们需要调用 `reducer` 以返回一个新状态
    const dispatch = (action) => {
        //reducer(state, action) 返回一个新状态
        state = reducer(state, action);
    }
    //你要是有个 action 的 type 的值是 `@@redux/__INIT__${Math.random()}`，我敬你是个狠人
    dispatch({ type: `@@redux/__INIT__${Math.random()}` });

    return {
        getState,
        dispatch
    }
}
```

现在这个 `createStore` 已经可以到处使用了，但是你有没有觉得每次 `dispatch` 后，都手动 `renderApp()` 显得很蠢，当前应用中是调用两次，如果需要修改1000次 `state` 呢，难道手动调用1000次 `renderApp()` ？

能不能简化一下呢？每次数据变化的时候，自动调用 `renderApp()` 。当然我们不可能将 `renderApp()` 写在 `createStore()` 的 `dispatch` 中，因为其它的应用中，函数名未必叫 `renderApp()` ，而且有可能不止要触发 `renderApp()` 。这里可以引入 发布订阅模式 ，当状态变化时，通知所有的订阅者。

createStore 进化版3.0

```javascript
function createStore(reducer) {
    let state;
    let listeners = [];
    const getState = () => state;
    //subscribe 每次调用，都会返回一个取消订阅的方法
    const subscribe = (ln) => {
        listeners.push(ln);
        //订阅之后，也要允许取消订阅。
        //难道我订了某本杂志之后，就不允许我退订吗？可怕~
        const unsubscribe = () => {
            listeners = listeners.filter(listener => ln !== listener);
        }
        return unsubscribe;
    };
    const dispatch = (action) => {
        //reducer(state, action) 返回一个新状态
        state = reducer(state, action);
        listeners.forEach(ln => ln());

    }
    //你要是有个 action 的 type 的值正好和 `@@redux/__INIT__${Math.random()}` 相等，我敬你是个狠人
    dispatch({ type: `@@redux/__INIT__${Math.random()}` });

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

至此，一个最为简单的 `redux` 已经创建好了， `createStore` 是 `redux` 的核心。我们来使用这个精简版的 `redux` 重写我们的代码， `index.js` 文件内容更新如下(`to-redux/src/index.js`):

```javascript
function createStore() {
    //code(自行将上面createStore的代码拷贝至此处)
}
const initialState = {
    color: 'blue'
}

function reducer(state = initialState, action) {
    switch (action.type) {
        case 'CHANGE_COLOR':
            return {
```

```
                ...state,
                color: action.color
            }
        default:
            return state;
    }
}
const store = createStore(reducer);

function renderApp(state) {
    renderHeader(state);
    renderContent(state);
}
function renderHeader(state) {
    const header = document.getElementById('header');
    header.style.color = state.color;
}
function renderContent(state) {
    const content = document.getElementById('content');
    content.style.color = state.color;
}

document.getElementById('to-blue').onclick = function () {
    store.dispatch({
        type: 'CHANGE_COLOR',
        color: 'rgb(0, 51, 254)'
    });
}
document.getElementById('to-pink').onclick = function () {
    store.dispatch({
        type: 'CHANGE_COLOR',
        color: 'rgb(247, 109, 132)'
    });
}

renderApp(store.getState());
//每次state发生改变时，都重新渲染
store.subscribe(() => renderApp(store.getState()));
```

如果现在我们现在希望在点击完 `Pink` 之后，字体色不允许修改，那么我们还可以取消订阅:

javascript

```
const unsub = store.subscribe(() => renderApp(store.getState()));
document.getElementById('to-pink').onclick = function () {
    //code...
    unsub(); //取消订阅
}
```

顺便说一句：`reducer` 是一个纯函数(纯函数的概念如果不了解的话，自行查阅资料)，它接收先前的 `state` 和 `action`，并返回新的 `state`。不要问为什么 `action` 中一定要有 `type` 字段，这仅仅是一个约定而已(`redux` 就是这么设计的)

遗留问题：为什么 `reducer` 一定要返回一个新的 `state`，而不是直接修改 `state` 呢。欢迎在评论区留下你的答案。

前面我们一步一步推演了 `redux` 的核心代码，现在我们来回顾一下 `redux` 的设计思想:

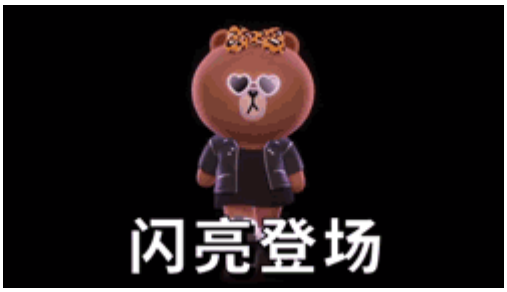
`Redux` 设计思想

- `Redux` 将整个应用状态(`state`)存储到一个地方(通常我们称其为 `store`)
- 当我们需要修改状态时，必须派发(`dispatch`)一个 `action` (`action` 是一个带有 `type` 字段的对象)
- 专门的状态处理函数 `reducer` 接收旧的 `state` 和 `action`，并会返回一个新的 `state`
- 通过 `subscribe` 设置订阅，每次派发动作时，通知所有的订阅者。

咱们现在已经有一个基础版本的 `redux` 了，但是它还不能满足我们的需求。我们平时的业务开发不会像上面所写的示例那样简单，那么就会有一个问题: `reducer` 函数可能会非常长，因为 `action` 的类型会非常多。这样肯定是不利于代码的编写和阅读的。

试想一下，你的业务中有一百种 `action` 需要处理，把这一百种情况编写在一个 `reducer` 中，不仅写得人恶心，后期维护代码的同事更是想杀人。

因此，我们最好单独编写 `reducer`，然后对 `reducer` 进行合并。有请我们的 `combineReducers` (和 `redux` 库的命名保持一致) 闪亮登场~



combineReducers

首先我们需要明确一点： `combineReducers` 只是一个工具函数，正如我们前面所说，它将多个 `reducer` 合并为一个 `reducer`。 `combineReducers` 返回的是 `reducer`，也就是说它是一个高阶函数。

我们还是以一个示例来说明，尽管 redux 不是非得和 react 配合，不过鉴于其与 react 配合最为适合，此处，以 react 代码为例：

这一次除了上面的展示以外，我们新增了一个计数器功能(使用 React 重构 ===> to-redux2):

```javascript
//现在我们的 state 结构如下:
let state = {
    theme: {
        color: 'blue'
    },
    counter: {
        number: 0
    }
}
```

显然，修改主题和计数器是可以分割开得，由不同的 reducer 去处理是一个更好的选择。

> store/reducers/counter.js

负责处理计数器的state。

```javascript
import { INCRENENT, DECREMENT } from '../action-types';

export default counter(state = {number: 0}, action) {
    switch (action.type) {
        case INCRENENT:
            return {
                ...state,
                number: state.number + action.number
            }
        case DECREMENT:
            return {
                ...state,
                number: state.number - action.number
            }
        default:
            return state;
    }
}
```

> store/reducers/theme.js

负责处理修改主题色的state。

```javascript
import { CHANGE_COLOR } from '../action-types';

export default function theme(state = {color: 'blue'}, action) {
    switch (action.type) {
        case CHANGE_COLOR:
            return {
                ...state,
                color: action.color
            }
        default:
            return state;
    }
}
```

每个 `reducer` 只负责管理全局 `state` 中它负责的一部分。每个 `reducer` 的 `state` 参数都不同，分别对应它管理的那部分 `state` 数据。

```javascript
import counter from './counter';
import theme from './theme';

export default function appReducer(state={}, action) {
    return {
        theme: theme(state.theme, action),
        counter: counter(state.counter, action)
    }
}
```

`appReducer` 即是合并之后的 `reducer`，但是当 `reducer` 较多时，这样写也显得繁琐，因此我们编写一个工具函数来生成这样的 `appReducer`，我们把这个工具函数命名为 `combineReducers` 。

我们来尝试一下编写这个工具函数 `combineReducers`：

> 思路：

1. `combineReducers` 返回 `reducer`
2. `combineReducers` 的入参是多个 `reducer` 组成的对象
3. 每个 `reducer` 只处理全局 `state` 中自己负责的部分

```javascript
//reducers 是一个对象，属性值是每一个拆分的 reducer
export default function combineReducers(reducers) {
```

```
    return function combination(state={}, action) {
        //reducer 的返回值是新的 state
        let newState = {};
        for(var key in reducers) {
            newState[key] = reducers[key](state[key], action);
        }
        return newState;
    }
}
```

子 `reducer` 将负责返回 `state` 的默认值。比如本例中， `createStore` 中 `dispatch({type: @@redux/INIT${Math.random()} })` ，而传递给 `createStore` 的是 `combineReducers(reducers)` 返回的函数 `combination` 。

根据 `state=reducer(state,action)` ， `newState.theme=theme(undefined, action)` ， `newState.counter=counter(undefined, action)` ， `counter` 和 `theme` 两个子 `reducer` 分别返回 `newState.theme` 和 `newState.counter` 的初始值。

利用此 `combineReducers` 可以重写 `store/reducers/index.js`

javascript

```
import counter from './counter';
import theme from './theme';
import { combineReducers } from '../redux';
//明显简洁了许多~
export default combineReducers({
    counter,
    theme
});
```

我们写的 `combineReducers` 虽然看起来已经能够满足我们的需求，但是其有一个缺点，即每次都会返回一个新的 `state` 对象，这会导致在数据没有变化时进行无意义的重新渲染。因此我们可以对数据进行判断，在数据没有变化时，返回原本的 `state` 即可。

> combineReducers 进化版

javascript

```
//代码中省略了一些判断，默认传递的参数均是符合要求的，有兴趣可以查看源码中对参数合法性的判断及处理
export default function combineReducers(reducers) {
    return function combination(state={}, action) {
        let nextState = {};
        let hasChanged = false; //状态是否改变
        for(let key in reducers) {
            const previousStateForKey = state[key];
            const nextStateForKey = reducers[key](previousStateForKey, action);
```

```
            nextState[key] = nextStateForKey;
            //只有所有的 nextStateForKey 均与 previousStateForKey 相等时，hasChanged 的值才是 fa
            hasChanged = hasChanged || nextStateForKey !== previousStateForKey;
        }
        //state 没有改变时，返回原对象
        return hasChanged ? nextState : state;
    }
}
```

applyMiddleware

官方文档中，关于 `applyMiddleware` 的解释很清楚，下面的内容也参考了官方文档的内容:

日志记录

考虑一个小小的问题，如果我们希望每次状态改变前能够在控制台中打印出 `state`，那么我们要怎么做呢?

最简单的即是:

javascript

```
//...
<button onClick={() => {
    console.log(store.getState());
    store.dispatch(actions.add(2));
}}>+</button>
//...
```

当然，这种方式肯定是不可取的，如果我们代码中派发100次，我们不可能这样写一百次。既然是状态改变时打印 `state`，也是说是在 `reducer` 调用之前打印 `state`，`reducer` 实在 `dispatch` 中调用的，那么我们可以重写 `store.dispatch` 方法，在派发前打印 `state` 即可。

javascript

```
let store = createStore(reducer);
const next = store.dispatch; //next 的命令是为了和中间件的源码一致
store.dispatch = action => {
    console.log(store.getState());
    next(action);
}
```

崩溃信息

假设我们不仅仅需要打印 `state`，还需要在派发异常出错时，打印出错误信息。

```javascript
const next = store.dispatch; //next 的命名是为了和中间件的源码一致
store.dispatch = action => {
    try{
        console.log(store.getState());
        next(action);
    } catct(err) {
        console.error(err);
    }
}
```

而如果我们还有其他的需求，那么就需要不停的修改 `store.dispatch` 方法，最后导致这个这部分代码难以维护。

因此我们需要分离 `loggerMiddleware` 和 `exceptionMiddleware` .

```javascript
let store = createStore(reducer);
const next = store.dispatch; //next 的命名是为了和中间件的源码一致
const loggerMiddleware = action => {
    console.log(store.getState());
    next(action);
}
const exceptionMiddleware = action => {
    try{
        loggerMiddleware(action);
    }catch(err) {
        console.error(err);
    }
}
store.dispatch = exceptionMiddleware;
```

我们知道，很多 `middleware` 都是第三方提供的，因此需要将 `dispatch` 和 `getState` 肯定是需要作为参数传递给 `middleware` ，进一步改写:

```javascript
const loggerMiddleware = ({dispatch, getState}) => action => {
    const next = dispatch;
    console.log(getState());
    next(action);
}
const exceptionMiddleware = ({dispatch, getState}) => action => {
    try{
        loggerMiddleware({dispatch, getState})(action);
    }catch(err) {
        console.error(err);
    }
```

```
}

//使用
store.dispatch = exceptionMiddleware({dispatch, getState})(action);
```

现在还有一个小小的问题， exceptionMiddleware 中的 loggerMiddleware 是写死的，这肯定是不合理的，我们希望这是一个参数，这样使用起来才灵活，没道理只有 exceptionMiddleware 需要灵活，而不管 loggerMiddleware ，进一步改写如下：

```javascript
const loggerMiddleware = ({dispatch, getState}) => next => action => {
    console.log(getState());
    return next(action);
}
const exceptionMiddleware = ({dispatch, getState}) => next => action => {
    try{
        return next(action);
    }catch(err) {
        console.error(err);
    }
}
//使用
const next = store.dispatch;
const logger = loggerMiddleware({dispatch: store.dispatch, getState: store.getState});
store.dispatch = exceptionMiddleware(store)(logger(next));
```

现在，我们已经有了通用 middleware 的编写格式了。

middleware 接收了一个 next() 的 dispatch 函数，并返回一个 dispatch 函数，返回的函数会被作为下一个 middleware 的 next()

但是有一个小小的问题，当中间件很多的时候，使用中间件的代码会变得很繁琐。为此， redux 提供了一个 applyMiddleware 的工具函数。

上面我们能够看出，其实我们最终要改变的就是 dispatch ，因此我们需要重写 store ，返回修改了 dispatch 方法之后的 store .

所以，我们可以明确以下几点：

1. applyMiddleware 返回值是 store
2. applyMiddleware 肯定要接受 middleware 作为参数
3. applyMiddleware 要接受 {dispatch, getState} 作为入参，不过 redux 源码中入参是 createStore 和 createStore 的入参，想想也是，没有必要在外部创建出 store ，毕竟

部创建出的 `store` 除了作为参数传递进函数，也没有其它作用，不如把 `createStore` 和 `createStore` 需要使用的参数传递进来。

```javascript
//applyMiddleWare 返回 store.
const applyMiddleware = middleware => createStore => (...args) => {
    let store = createStore(...args);
    let middle = loggerMiddleware(store);
    let dispatch = middle(store.dispatch); //新的dispatch方法
    //返回一个新的store---重写了dispatch方法
    return {
        ...store,
        dispatch
    }
}
```

以上是一个 `middleware` 的情况，但是我们知道， `middleware` 可能是一个或者是多个，而且我们主要是要解决多个 `middleware` 的问题，进一步改写。

```javascript
//applyMiddleware 返回 store.
const applyMiddleware = (...middlewares) => createStore => (...args) => {
    let store = createStore(...args);
    let dispatch;
    const middlewareAPI = {
        getState: store.getstate,
        dispatch: (...args) => dispatch(...args)
    }
    //传递修改后的 dispatch
    let middles = middlewares.map(middleware => middleware(middlewareAPI));
    //现在我们有多个 middleware，需要多次增强 dispatch
    dispatch = middles.reduceRight((prev, current) => current(prev), store.dispatch);
    return {
        ...store,
        dispatch
    }
}
```

不知道大家是不是理解了上面的 `middles.reduceRight` ， 下面为大家细致说明一下：

```javascript
/*三个中间件*/
let logger1 = ({dispatch, getState}) => dispatch => action => {
    console.log('111');
    dispatch(action);
    console.log('444');
}
let logger2 = ({ dispatch, getState }) => dispatch => action => {
```

```
        console.log('222');
        dispatch(action);
        console.log('555')
    }
    let logger3 = ({ dispatch, getState }) => dispatch => action => {
        console.log('333');
        dispatch(action);
        console.log('666');
    }
    let middle1 = logger1({ dispatch, getState });
    let middle2 = logger2({ dispatch, getState });
    let middle3 = logger3({ dispatch, getState });

    //applyMiddleware(logger1,logger2,logger3)(createStore)(reducer)
    //如果直接替换
    store.dispatch = middle1(middle2(middle3(store.dispatch)));
```

观察上面的 `middle1(middle2(middle3(store.dispatch)))` ，如果我们把 `middle1` , `middle2` , `middle3` 看成是数组的每一项，如果对数组的API比较熟悉的话，可以想到 `reduce` ，如果你还不熟悉 `reduce` ，可以查看MDN文档。

```javascript
    //applyMiddleware(logger1,logger3,logger3)(createStore)(reducer)

    //reduceRight 从右到左执行
    middles.reduceRight((prev, current) => current(prev), store.dispatch);
    //第一次 prev: store.dispatch    current: middle3
    //第二次 prev: middle3(store.dispatch) current: middle2
    //第三次 prev: middle2(middle3(store.dispatch))  current: middle1
    //结果 middle1(middle2(middle3(store.dispatch)))
```

阅读过 `redux` 的源码的同学，可能知道源码中是提供了一个 `compose` 函数，而 `compose` 函数中没有使用 `reduceRight` ，而是使用的 `reduce` ，因而代码稍微有点不同。但是分析过程还是一样的。

compose.js

```javascript
    export default function compose(...funcs) {
        //如果没有中间件
        if (funcs.length === 0) {
            return arg => arg
        }
        //中间件长度为1
        if (funcs.length === 1) {
            return funcs[0]
        }
```

```
    return funcs.reduce((prev, current) => (...args) => prev(current(...args)));
}
```

关于 `reduce` 的写法，建议像上面的 `reduceRight` 一样，进行一次分析

使用 `compose` 工具函数重写 `applyMiddleware` 。

```javascript
const applyMiddleware = (...middlewares) => createStore => (...args) => {
    let store = createStore(...args);
    let dispatch;
    const middlewareAPI = {
        getState: store.getstate,
        dispatch: (...args) => dispatch(...args)
    }
    let middles = middlewares.map(middleware => middleware(middlewareAPI));
    dispatch = compose(...middles)(store.dispatch);
    return {
        ...store,
        dispatch
    }
}
```

bindActionCreators

`redux` 还为我们提供了 `bindActionCreators` 工具函数，这个工具函数代码很简单，我们很少直接在代码中使用它， `react-redux` 中会使用到。此处，简单说明一下:

```javascript
//通常我们会这样编写我们的 actionCreator
import { INCRENENT, DECREMENT } from '../action-types';

const counter = {
    add(number) {
        return {
            type: INCRENENT,
            number
        }
    },
    minus(number) {
        return {
            type: DECREMENT,
            number
        }
    }
}
```

```
export default counter;
```

在派发的时候，我们需要这样写:

```javascript
import counter from 'xx/xx';
import store from 'xx/xx';

store.dispatch(counter.add());
```

当然，我们也可以像下面这样编写我们的 actionCreator:

```javascript
function add(number) {
    return {
        type: INCRENENT,
        number
    }
}
```

派发时，需要这样编写:

```javascript
store.dispatch(add(number));
```

以上代码有一个共同点，就是都是 `store.dispatch` 派发一个动作。因此我们可以考虑编写一个函数，将 `store.dispatch` 和 `actionCreator` 绑定起来。

```javascript
function bindActionCreator(actionCreator, dispatch) {
    return  (...args) => dispatch(actionCreator(...args));
}
function bindActionCreators(actionCreator, dispatch) {
    //actionCreators 可以是一个普通函数或者是一个对象
    if(typeof actionCreator === 'function') {
        //如果是函数，返回一个函数，调用时，dispatch 这个函数的返回值
        bindActionCreator(actionCreator, dispatch);
    }else if(typeof actionCreator === 'object') {
        //如果是一个对象，那么对象的每一项都要都要返回 bindActionCreator
        let boundActionCreators = {}
        for(let key in actionCreator) {
            boundActionCreators[key] = bindActionCreator(actionCreator[key], dispatch);
        }
        return boundActionCreators;
```

```
    }
}
```

在使用时:

```javascript
let counter = bindActionCreators(counter, store.dispatch);
//派发时
counter.add(number);
counter.minus(number);
```

这里看起来并没有精简太多，后面在分析 `react-redux` 时，我们会说明为什么我们需要这个工具函数。

至此，我们的 `redux` 基本已经编写完毕。与 `redux` 的源码相比，还相差一些内容，例如一些校验，还有 `createStore` 提供的 `replaceReducer` 方法，以及 `createStore` 的第二个参数和第三个参数我们没有提及，稍微看一下代码就能懂，此处不再一一展开。

参考链接

1. React.js 小书
2. redux中文文档
3. 完全理解 redux（从零实现一个 redux）

关注公众号，加入技术交流群