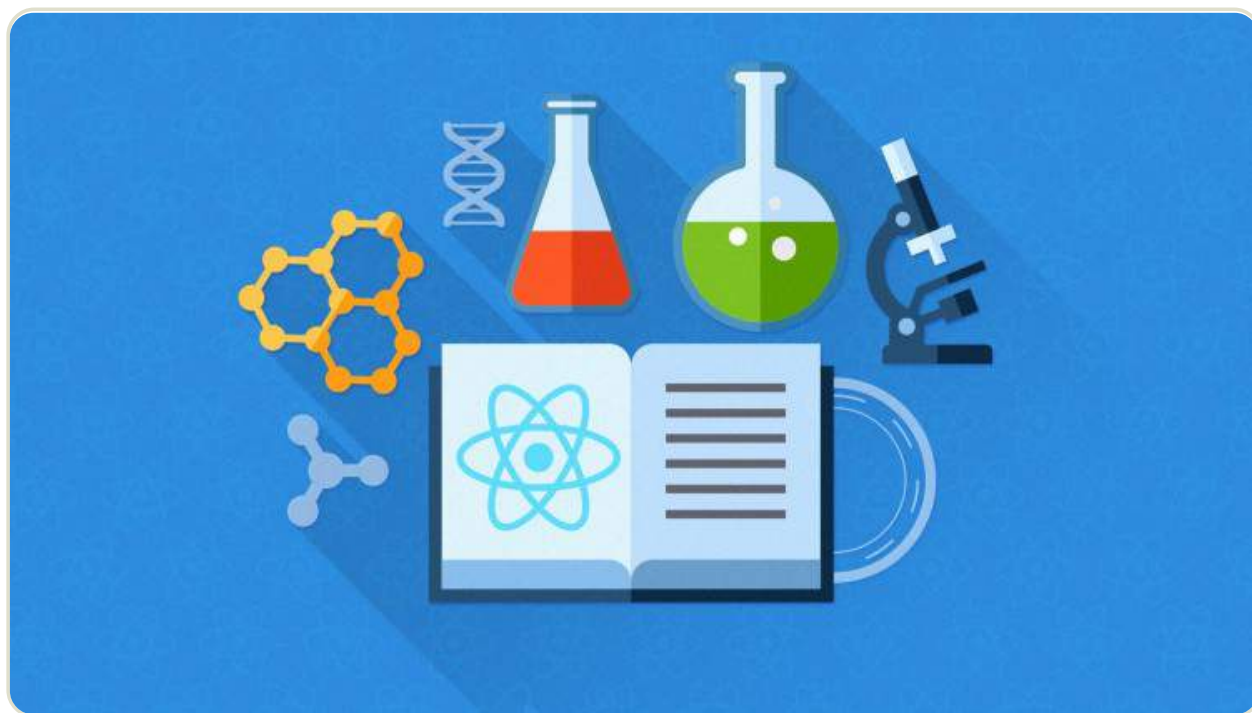


Redux 入门教程（二）：中间件与异步操作

作者：阮一峰

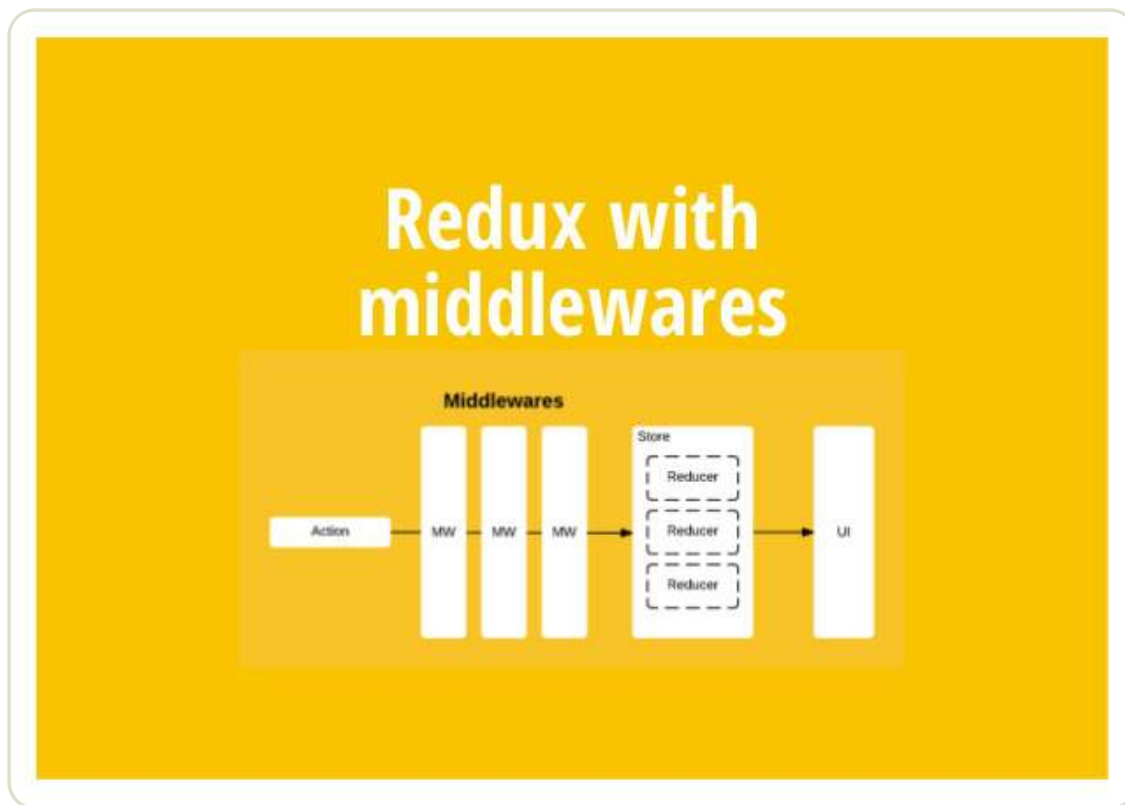
日期：2016年9月20日

上一篇文章，我介绍了 Redux 的基本做法：用户发出 Action，Reducer 函数算出新的 State，View 重新渲染。



但是，一个关键问题没有解决：异步操作怎么办？Action 发出以后，Reducer 立即算出 State，这叫做同步；Action 发出以后，过一段时间再执行 Reducer，这就是异步。

怎么才能 Reducer 在异步操作结束后自动执行呢？这就要用到新的工具：中间件（middleware）。



一、中间件的概念

为了理解中间件，让我们站在框架作者的角度思考问题：如果要添加功能，你会在哪个环节添加？

（1）**Reducer**：纯函数，只承担计算 **State** 的功能，不合适承担其他功能，也承担不了，因为理论上，纯函数不能进行读写操作。

（2）**View**：与 **State** 一一对应，可以看作 **State** 的视觉层，也不合适承担其他功能。

（3）**Action**：存放数据的对象，即消息的载体，只能被别人操作，自己不能进行任何操作。

想来想去，只有发送 **Action** 的这个步骤，即 `store.dispatch()` 方法，可以添加功能。举例来说，要添加日志功能，把 **Action** 和 **State** 打印出来，可以对 `store.dispatch` 进行如下改造。

```
let next = store.dispatch;
store.dispatch = function dispatchAndLog(action) {
  console.log('dispatching', action);
  next(action);
  console.log('next state', store.getState());
}
```

上面代码中，对 `store.dispatch` 进行了重定义，在发送 **Action** 前后添加了打印功能。这就是中间件的雏形。

中间件就是一个函数，对 `store.dispatch` 方法进行了改造，在发出 **Action** 和执行 **Reducer** 这两步之间，添加了其他功能。

二、中间件的用法

本教程不涉及如何编写中间件，因为常用的中间件都有现成的，只要引用别人写好的模块即可。比如，上一节的日志中间件，就有现成的[redux-logger](#)模块。这里只介绍怎么使用中间件。

```
import { applyMiddleware, createStore } from 'redux';
import createLogger from 'redux-logger';
const logger = createLogger();

const store = createStore(
  reducer,
  applyMiddleware(logger)
);
```

上面代码中，`redux-logger` 提供一个生成器 `createLogger`，可以生成日志中间件 `logger`。然后，将它放在 `applyMiddleware` 方法之中，传入 `createStore` 方法，就完成了 `store.dispatch()` 的功能增强。

这里有两点需要注意：

（1）`createStore` 方法可以接受整个应用的初始状态作为参数，那样的话，`applyMiddleware` 就是第三个参数了。

```
const store = createStore(
  reducer,
  initial_state,
  applyMiddleware(logger)
);
```

（2）中间件的次序有讲究。

```
const store = createStore(
  reducer,
  applyMiddleware(thunk, promise, logger)
);
```

上面代码中，`applyMiddleware` 方法的三个参数，就是三个中间件。有的中间件有次序要求，使用前要查一下文档。比如，`logger` 就一定要放在最后，否则输出结果会不正确。

三、applyMiddlewares()

看到这里，你可能会问，`applyMiddlewares` 这个方法到底是干什么的？

它是 `Redux` 的原生方法，作用是将所有中间件组成一个数组，依次执行。下面是它的源码。

```
export default function applyMiddleware(...middlewares) {
  return (createStore) => (reducer, ...args) => {
    const store = createStore(reducer, ...args);
    let dispatch = () => {
      throw new Error('Dispatching while providing listeners is not permitted')
```

```
return (createStore) => (reducer, preloadedState, enhancer) => {
  var store = createStore(reducer, preloadedState, enhancer);
  var dispatch = store.dispatch;
  var chain = [];

  var middlewareAPI = {
    getState: store.getState,
    dispatch: (action) => dispatch(action)
  };
  chain = middlewares.map(middleware => middleware(middlewareAPI));
  dispatch = compose(...chain)(store.dispatch);

  return {...store, dispatch}
}
```

上面代码中，所有中间件被放进了一个数组 `chain`，然后嵌套执行，最后执行 `store.dispatch`。可以看到，中间件内部（`middlewareAPI`）可以拿到 `getState` 和 `dispatch` 这两个方法。

四、异步操作的基本思路

理解了中间件以后，就可以处理异步操作了。

同步操作只要发出一种 **Action** 即可，异步操作的差别是它要发出三种 **Action**。

- 操作发起时的 **Action**
- 操作成功时的 **Action**
- 操作失败时的 **Action**

以向服务器取出数据为例，三种 **Action** 可以有两种不同的写法。

```
// 写法一：名称相同，参数不同
{ type: 'FETCH_POSTS' }
{ type: 'FETCH_POSTS', status: 'error', error: 'Oops' }
{ type: 'FETCH_POSTS', status: 'success', response: { ... } }

// 写法二：名称不同
{ type: 'FETCH_POSTS_REQUEST' }
{ type: 'FETCH_POSTS_FAILURE', error: 'Oops' }
{ type: 'FETCH_POSTS_SUCCESS', response: { ... } }
```

除了 **Action** 种类不同，异步操作的 **State** 也要进行改造，反映不同的操作状态。下面是 **State** 的一个例子。

```
let state = {
  // ...
  isFetching: true,
  didInvalidate: true,
```

```
    lastUpdated: 'xxxxxxx'  
  };
```

上面代码中，**State** 的属性 `isFetching` 表示是否在抓取数据。 `didInvalidate` 表示数据是否过时，`lastUpdated` 表示上一次更新时间。

现在，整个异步操作的思路就很清楚了。

- 操作开始时，送出一个 **Action**，触发 **State** 更新为"正在操作"状态，**View** 重新渲染
- 操作结束后，再送出一个 **Action**，触发 **State** 更新为"操作结束"状态，**View** 再一次重新渲染

五、redux-thunk 中间件

异步操作至少要送出两个 **Action**：用户触发第一个 **Action**，这个跟同步操作一样，没有问题；如何才能在操作结束时，系统自动送出第二个 **Action** 呢？

奥妙就在 **Action Creator** 之中。

```
class AsyncApp extends Component {  
  componentDidMount() {  
    const { dispatch, selectedPost } = this.props  
    dispatch(fetchPosts(selectedPost))  
  }  
  
  // ...
```

上面代码是一个异步组件的例子。加载成功后（`componentDidMount` 方法），它送出了（`dispatch` 方法）一个 **Action**，向服务器要求数据 `fetchPosts(selectedSubreddit)`。这里的 `fetchPosts` 就是 **Action Creator**。

下面就是 `fetchPosts` 的代码，关键之处就在里面。

Async Action Example

```
import fetch from 'isomorphic-fetch';

export function fetchFriends() {
  return dispatch => {
    dispatch({ type: 'FETCH_FRIENDS' });
    return fetch('http://localhost/api/friends/')
      .then(response => response.json())
      .then(json => {
        dispatch({ type: 'RECEIVE_FRIENDS', payload: json });
      });
  };
}
```

Redux Universal

@nikgraf

```
const fetchPosts = postTitle => (dispatch, getState) => {
  dispatch(requestPosts(postTitle));
  return fetch(`/some/API/${postTitle}.json`)
    .then(response => response.json())
    .then(json => dispatch(receivePosts(postTitle, json)));
};

// 使用方法一
store.dispatch(fetchPosts('reactjs'));
// 使用方法二
store.dispatch(fetchPosts('reactjs')).then(() =>
  console.log(store.getState())
);
```

上面代码中，`fetchPosts` 是一个 **Action Creator**（动作生成器），返回一个函数。这个函数执行后，先发出一个 **Action**（`requestPosts(postTitle)`），然后进行异步操作。拿到结果后，先将结果转成 **JSON** 格式，然后再发出一个 **Action**（`receivePosts(postTitle, json)`）。

上面代码中，有几个地方需要注意。

- （1）`fetchPosts` 返回了一个函数，而普通的 **Action Creator** 默认返回一个对象。
- （2）返回的函数的参数是 `dispatch` 和 `getState` 这两个 **Redux** 方法，普通的 **Action Creator** 的参数是 **Action** 的内容。
- （3）在返回的函数之中，先发出一个 **Action**（`requestPosts(postTitle)`），表示操作开始。

（4）异步操作结束之后，再发出一个 **Action**（`receivePosts(postTitle, json)`），表示操作结束。

这样的处理，就解决了自动发送第二个 **Action** 的问题。但是，又带来了一个新的问题，**Action** 是由 `store.dispatch` 方法发送的。而 `store.dispatch` 方法正常情况下，参数只能是对象，不能是函数。

这时，就要使用中间件 `redux-thunk`。

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import reducer from './reducers';

// Note: this API requires redux@>=3.1.0
const store = createStore(
  reducer,
  applyMiddleware(thunk)
);
```

上面代码使用 `redux-thunk` 中间件，改造 `store.dispatch`，使得后者可以接受函数作为参数。

因此，异步操作的第一种解决方案就是，写出一个返回函数的 **Action Creator**，然后使用 `redux-thunk` 中间件改造 `store.dispatch`。

六、redux-promise 中间件

既然 **Action Creator** 可以返回函数，当然也可以返回其他值。另一种异步操作的解决方案，就是让 **Action Creator** 返回一个 **Promise** 对象。

这就需要使用 `redux-promise` 中间件。

```
import { createStore, applyMiddleware } from 'redux';
import promiseMiddleware from 'redux-promise';
import reducer from './reducers';

const store = createStore(
  reducer,
  applyMiddleware(promiseMiddleware)
);
```

这个中间件使得 `store.dispatch` 方法可以接受 **Promise** 对象作为参数。这时，**Action Creator** 有两种写法。写法一，返回值是一个 **Promise** 对象。

```
const fetchPosts =
(dispatch, postTitle) => new Promise(function (resolve, reject) {
  dispatch(requestPosts(postTitle));
  return fetch(`/some/API/${postTitle}.json`)
    .then(response => {
```

```
    type: 'FETCH_POSTS',
    payload: response.json()
  });
});
```

写法二，Action 对象的 payload 属性是一个 Promise 对象。这需要从 `redux-actions` 模块引入 `createAction` 方法，并且写法也要变成下面这样。

```
import { createAction } from 'redux-actions';

class AsyncApp extends Component {
  componentDidMount() {
    const { dispatch, selectedPost } = this.props
    // 发出同步 Action
    dispatch(requestPosts(selectedPost));
    // 发出异步 Action
    dispatch(createAction(
      'FETCH_POSTS',
      fetch(`/some/API/${postTitle}.json`)
        .then(response => response.json())
    ));
  }
}
```

上面代码中，第二个 `dispatch` 方法发出的是异步 Action，只有等到操作结束，这个 Action 才会实际发出。注意，`createAction` 的第二个参数必须是一个 Promise 对象。

看一下 `redux-promise` 的[源码](#)，就会明白它内部是怎么操作的。

```
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);
  };
}
```

从上面代码可以看出，如果 Action 本身是一个 Promise，它 resolve 以后的值应该是一个 Action 对象，会被 `dispatch` 方法送出（`action.then(dispatch)`），但 reject 以后不会有任何动作；如果 Action 对象的 payload 属性是一个 Promise 对象，那么无论 resolve 和 reject，`dispatch` 方法都会发出 Action。

中间件和异步操作，就介绍到这里。[下一篇文章](#)将是最后一部分，介绍如何使用 react-redux 这个库。

（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20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