

Redux 入门教程（三）：React-Redux 的用法

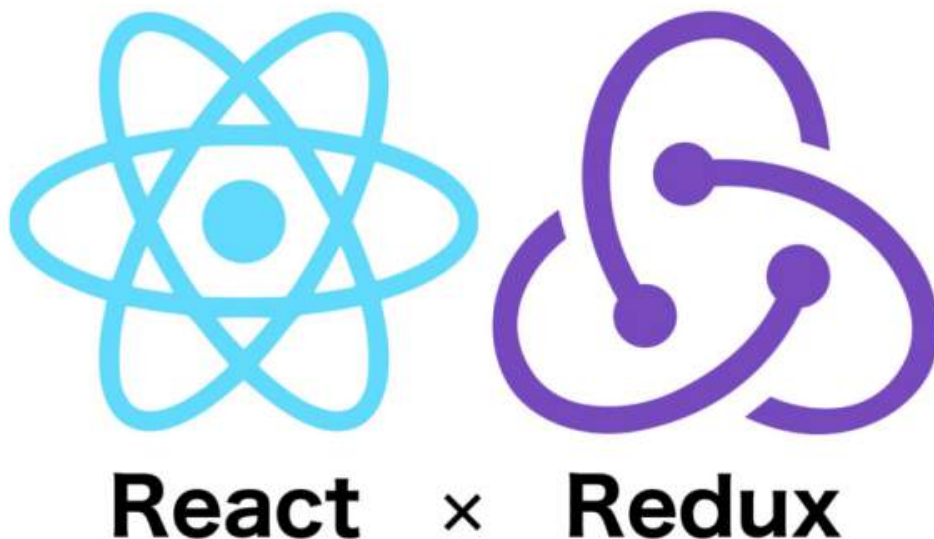
作者：阮一峰

日期：2016年9月21日

前两篇教程介绍了 Redux 的基本用法和异步操作，今天是最后一部分，介绍如何在 React 项目中使用 Redux。

为了方便使用，Redux 的作者封装了一个 React 专用的库 [React-Redux](#)，本文主要介绍它。

这个库是可以选用的。实际项目中，你应该权衡一下，是直接使用 Redux，还是使用 React-Redux。后者虽然提供了便利，但是需要掌握额外的 API，并且要遵守它的组件拆分规范。



一、UI 组件

React-Redux 将所有组件分成两大类：UI 组件（presentational component）和容器组件（container component）。

UI 组件有以下几个特征。

- 只负责 UI 的呈现，不带有任何业务逻辑
- 没有状态（即不使用 `this.state` 这个变量）
- 所有数据都由参数（`this.props`）提供

- 不使用任何 Redux 的 API

下面就是一个 UI 组件的例子。

```
const Title =  
  value => <h1>{value}</h1>;
```

因为不含有状态，UI 组件又称为"纯组件"，即它纯函数一样，纯粹由参数决定它的值。

二、容器组件

容器组件的特征恰恰相反。

- 负责管理数据和业务逻辑，不负责 UI 的呈现
- 带有内部状态
- 使用 Redux 的 API

总之，只要记住一句话就可以了：UI 组件负责 UI 的呈现，容器组件负责管理数据和逻辑。

你可能会问，如果一个组件既有 UI 又有业务逻辑，那怎么办？回答是，将它拆分成下面的结构：外面是一个容器组件，里面包了一个 UI 组件。前者负责与外部的通信，将数据传给后者，由后者渲染出视图。

React-Redux 规定，所有的 UI 组件都由用户提供，容器组件则是由 React-Redux 自动生成。也就是说，用户负责视觉层，状态管理则是全部交给它。

三、connect()

React-Redux 提供 connect 方法，用于从 UI 组件生成容器组件。connect 的意思，就是将这两种组件连起来。

```
import { connect } from 'react-redux'  
const VisibleTodoList = connect()(TodoList);
```

上面代码中，TodoList 是 UI 组件，VisibleTodoList 就是由 React-Redux 通过 connect 方法自动生成的容器组件。

但是，因为没有定义业务逻辑，上面这个容器组件毫无意义，只是 UI 组件的一个单纯的包装层。为了定义业务逻辑，需要给出下面两方面的信息。

- （1）输入逻辑：外部的数据（即 state 对象）如何转换为 UI 组件的参数
- （2）输出逻辑：用户发出的动作如何变为 Action 对象，从 UI 组件传出去。

因此，`connect` 方法的完整 API 如下。

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)
```

上面代码中，`connect` 方法接受两个参数：`mapStateToProps` 和 `mapDispatchToProps`。它们定义了 UI 组件的业务逻辑。前者负责输入逻辑，即将 `state` 映射到 UI 组件的参数（`props`），后者负责输出逻辑，即将用户对 UI 组件的操作映射成 `Action`。

四、`mapStateToProps`

`mapStateToProps` 是一个函数。它的作用就是像它的名字那样，建立一个从（外部的）`state` 对象到（UI 组件的）`props` 对象的映射关系。

作为函数，`mapStateToProps` 执行后应该返回一个对象，里面的每一个键值对就是一个映射。请看下面的例子。

```
const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}
```

上面代码中，`mapStateToProps` 是一个函数，它接受 `state` 作为参数，返回一个对象。这个对象有一个 `todos` 属性，代表 UI 组件的同名参数，后面的 `getVisibleTodos` 也是一个函数，可以从 `state` 算出 `todos` 的值。

下面就是 `getVisibleTodos` 的一个例子，用来算出 `todos`。

```
const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_ALL':
      return todos
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
    default:
      throw new Error('Unknown filter: ' + filter)
  }
}
```

`mapStateToProps` 会订阅 **Store**，每当 `state` 更新的时候，就会自动执行，重新计算 **UI** 组件的参数，从而触发 **UI** 组件的重新渲染。

`mapStateToProps` 的第一个参数总是 `state` 对象，还可以使用第二个参数，代表容器组件的 `props` 对象。

```
// 容器组件的代码
//   <FilterLink filter="SHOW_ALL">
//     All
//   </FilterLink>

const mapStateToProps = (state, ownProps) => {
  return {
    active: ownProps.filter === state.visibilityFilter
  }
}
```

使用 `ownProps` 作为参数后，如果容器组件的参数发生变化，也会引发 **UI** 组件重新渲染。

`connect` 方法可以省略 `mapStateToProps` 参数，那样的话，**UI** 组件就不会订阅**Store**，就是说 **Store** 的更新不会引起 **UI** 组件的更新。

五、mapDispatchToProps()

`mapDispatchToProps` 是 `connect` 函数的第二个参数，用来建立 **UI** 组件的参数到 `store.dispatch` 方法的映射。也就是说，它定义了哪些用户的操作应该当作 **Action**，传给 **Store**。它可以是一个函数，也可以是一个对象。

如果 `mapDispatchToProps` 是一个函数，会得到 `dispatch` 和 `ownProps`（容器组件的 `props` 对象）两个参数。

```
const mapDispatchToProps = (
  dispatch,
  ownProps
) => {
  return {
    onClick: () => {
      dispatch({
        type: 'SET_VISIBILITY_FILTER',
        filter: ownProps.filter
      });
    }
  };
};
```

从上面代码可以看到，`mapDispatchToProps` 作为函数，应该返回一个对象，该对象的每个键值对都是一个映射，定义了 **UI** 组件的参数怎样发出 **Action**。

如果 `mapDispatchToProps` 是一个对象，它的每个键名也是对应 UI 组件的同名参数，键值应该是一个函数，会被当作 **Action creator**，返回的 **Action** 会由 **Redux** 自动发出。举例来说，上面的 `mapDispatchToProps` 写成对象就是下面这样。

```
const mapDispatchToProps = {
  onClick: (filter) => {
    type: 'SET_VISIBILITY_FILTER',
    filter: filter
  };
}
```

六、<Provider> 组件

`connect` 方法生成容器组件以后，需要让容器组件拿到 `state` 对象，才能生成 UI 组件的参数。

一种解决方法是将 `state` 对象作为参数，传入容器组件。但是，这样做比较麻烦，尤其是容器组件可能在很深的层级，一级级将 `state` 传下去就很麻烦。

React-Redux 提供 **Provider** 组件，可以让容器组件拿到 `state`。

```
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import todoApp from './reducers'
import App from './components/App'

let store = createStore(todoApp);

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

上面代码中，`Provider` 在根组件外面包了一层，这样一来，`App` 的所有子组件就默认都可以拿到 `state` 了。

它的原理是 **React** 组件的 `context` 属性，请看源码。

```
class Provider extends Component {
  getChildContext() {
    return {
      store: this.props.store
    };
  }
  render() {
    return this.props.children;
  }
}
```

```
}

Provider.childContextTypes = {
  store: React.PropTypes.object
}
```

上面代码中，`store` 放在了上下文对象 `context` 上面。然后，子组件就可以从 `context` 拿到 `store`，代码大致如下。

```
class VisibleTodoList extends Component {
  componentDidMount() {
    const { store } = this.context;
    this.unsubscribe = store.subscribe(() =>
      this.forceUpdate()
    );
  }

  render() {
    const props = this.props;
    const { store } = this.context;
    const state = store.getState();
    // ...
  }
}

VisibleTodoList.contextTypes = {
  store: React.PropTypes.object
}
```

React-Redux 自动生成的容器组件的代码，就类似上面这样，从而拿到 `store`。

七、实例：计数器

我们来看一个实例。下面是一个计数器组件，它是一个纯的 UI 组件。

```
class Counter extends Component {
  render() {
    const { value, onIncreaseClick } = this.props
    return (
      <div>
        <span>{value}</span>
        <button onClick={onIncreaseClick}>Increase</button>
      </div>
    )
  }
}
```

上面代码中，这个 UI 组件有两个参数：`value` 和 `onIncreaseClick`。前者需要从 `state` 计算得到，后者需要向外发出 `Action`。

接着，定义 value 到 state 的映射，以及 onIncreaseClick 到 dispatch 的映射。

```
function mapStateToProps(state) {
  return {
    value: state.count
  }
}

function mapDispatchToProps(dispatch) {
  return {
    onIncreaseClick: () => dispatch(increaseAction)
  }
}

// Action Creator
const increaseAction = { type: 'increase' }
```

然后，使用 connect 方法生成容器组件。

```
const App = connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter)
```

然后，定义这个组件的 Reducer。

```
// Reducer
function counter(state = { count: 0 }, action) {
  const count = state.count
  switch (action.type) {
    case 'increase':
      return { count: count + 1 }
    default:
      return state
  }
}
```

最后，生成 store 对象，并使用 Provider 在根组件外面包一层。

```
import { loadState, saveState } from './localStorage';

const persistedState = loadState();
const store = createStore(
  todoApp,
  persistedState
);

store.subscribe(throttle(() => {
  saveState({
    todos: store.getState().todos,
```

```
    })  
  }, 1000))  
  
ReactDOM.render(  
  <Provider store={store}>  
    <App />  
  </Provider>,  
  document.getElementById('root')  
)
```

完整的代码看[这里](#)。




八、React-Router 路由库

使用 React-Router 的项目，与其他项目没有不同之处，也是使用 Provider 在 Router 外面包一层，毕竟 Provider 的唯一功能就是传入 store 对象。

```
const Root = ({ store }) => (  
  <Provider store={store}>  
    <Router>  
      <Route path="/" component={App} />  
    </Router>  
  </Provider>  
)
```

（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21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

